Heterogeneous Decomposition of Degree-Balanced Search Trees and Its Applications

Shan Leung Woo
maverick@cs.cmu.edu

CMU-CS-09-133

May 25, 2009

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee

Guy E. Blelloch, Co-Chair
Bruce M. Maggs, Co-Chair
Daniel D. Sleator
Richard Cole, New York University

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

This research was sponsored by the National Science Foundation under contract no. EIA-9706572 (PSciCo), Appalachian Regional Commission under contract no. CO-14574, National Science Foundation under contract no. CCR-0122581 (ALADDIN), and National Science Foundation under contract no. CNS-0435382 (NETS-NR). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Abstract

This thesis introduces the concept of heterogeneous decompositions of a degree-balanced search tree and applies this concept to establish the following three results.

(1) Any leaf-store or node-store degree-balanced search tree can support a constant number of dynamic fingers in the worst case without storing extra pointers in its nodes nor restructuring after a finger search. Each dynamic finger is represented as a logarithmic-sized data structure that contains pointers pointing into the tree, which is maintained using dictionary algorithms that exploit this representation of dynamic fingers.

(2) By construction, there exists a static binary search tree algorithm with the dynamic finger property in the worst case. This algorithm is primarily intended to serve as an alternate proof that the dynamic optimality conjecture implies the dynamic finger conjecture—in view of the fact that the earlier explicit proof of this implication is the highly-nontrivial proof of the dynamic finger theorem due to Cole.

(3) By construction, there exists a static $O(\lg \lg n)$-competitive binary search tree algorithm with the dynamic finger property in the amortized case. As a corollary, if the splay trees of Sleator and Tarjan are $O(1)$-competitive even in the presence of splits and joins, then the multi-splay trees of Wang, Derryberry, and Sleator have the dynamic finger property in the amortized case.

Contents

List of Figures

List of Tables

1 Introduction

THIS THESIS introduces the concept of heterogeneous decompositions of a degree-balanced search tree and applies this concept to establish the following three results.

(1) Any leaf-store or node-store degree-balanced search tree can support a constant number of dynamic fingers in the worst case without storing extra pointers in its nodes nor restructuring after a finger search. Each dynamic finger is represented as a logarithmic-sized data structure that contains pointers pointing into the tree, which is maintained using dictionary algorithms that exploit this representation of dynamic fingers.

(2) By construction, there exists a static binary search tree algorithm with the dynamic finger property in the worst case. This algorithm is primarily intended to serve as an alternate proof that the dynamic optimality conjecture implies the dynamic finger conjecture [ST85b]—in view of the fact that the earlier explicit proof of this implication is the highly-nontrivial proof of the dynamic finger theorem due to Cole [Col00].

(3) By construction, there exists a static $O(\lg \lg n)$-competitive binary search tree algorithm with the dynamic finger property in the amortized case. As a corollary, if the splay trees of Sleator and Tarjan [ST85b] are $O(1)$-competitive even in the presence of splits and joins, then the multi-splay trees of Wang, Derryberry, and Sleator [WDS06] have the dynamic finger property in the amortized case.

To ensure that these results are presented in the thesis in a self-contained manner, this chapter presents the four background topics that these results depend upon. We will use §1.1–§1.4 to provide a technical overview of each topic with a brief account of its history mixed in. A short survey on the history of finger search will then be provided in §1.5. Even though some of the definitions appearing in this chapter are new, we acknowledge that all technical results presented in this chapter are essentially our adaptation of the literature and due credits will be given throughout. The reader can find our own work starting from §2.

1.1 Degree-Balanced Search Trees

The term "degree-balanced search trees" was coined in the thesis of Overmars [Ove83, §3.2] to denote a major class of worst-case balanced search trees available at the time. A search tree in this class maintains its balance by ensuring that (i) all external nodes appear at the same depth, and (ii) the degree of an internal node ranges from a constant $a \geq 2$ to another constant $b \geq (2a-1)$. A sole exception is allowed at the root, whose degree can be as low as 2 regardless of both the value of a and the number of keys in the tree. Together these invariants imply that the height of a degree-balanced search tree with n keys is $O(\log_a n)$, which is the definition of being "balanced" for any worst-case search tree.

Embedded in the above description of degree-balanced search trees is Assumption 1.1 below. Note that this assumption implies that b is a constant multiple of a. However, we specifically do *not* assume that a and b are small; an explicit assumption has to be made when it is needed. We will also make two other assumptions throughout this thesis, namely Assumption 1.2 and Assumption 1.3 below. The former specifies several important properties of keys and also allows us to avoid dealing with duplicate keys in any of our search trees no matter it is degree-balanced or not. The latter is a technical assumption that will be explained in §1.1.1.2 when we deal with search tree representations, but it is stated here with the other two global assumptions of this thesis for the sake of coherence.

Assumption 1.1 Both degree bounds of a degree-balanced search tree are fixed but possibly large constants.

Assumption 1.2 All keys appearing within a search tree are unique elements drawn from a totally ordered set. Each key can be encoded and stored using $O(1)$ words and two keys can be compared in $O(1)$ time. The

only operation that can be performed on keys are comparisons unless other operations are explicitly allowed.

Assumption 1.3 Any search tree is in the node-store representation unless it is specified to be otherwise.

Before we go on, let us make a few further remarks regarding the first two assumptions. First, as we will see, within this thesis we will be frequently working inside some context in which both a and b are small. When this is true, or in general when the base of a logarithm is small, we will simply use the binary logarithm lg to suppress the base in our asymptotic bounds. Related to this, we are also lax about logarithms when they concern natural numbers such as the number of keys in a search tree. Our hope is the $O(\log_a n)$ bound above would have raised no eyebrows, even though n may in fact be 0 or 1 in the expression.

We also acknowledge that Assumption 1.1 and Assumption 1.2 are certainly *not* applicable to all degree-balanced search trees. For example, Brodal [Bro98] has a design in which the ratio between the two degree bounds as well as the two bounds themselves are doubly-exponential functions on the height of a node. Also, certain applications such as sorting can benefit from using search trees that handle duplicate keys correctly. Finally, notice that Assumption 1.2 restricts our attention in this thesis to search trees that are comparison-based. Readers who are interested in search trees that can take advantage of the RAM model of computation are referred to the work of Andersson and Thorup [AT07].

1.1.1 Search Trees and Their Representations

Since we are going to discuss various search tree designs and their relative strengths, it will be useful for us to define a *minimal* pointer-based search tree as the baseline in our comparisons. We will also take this chance to spell out our search tree terminology for the sake of completeness and definiteness—the latter being particularly important because some of the terms we use are new and some of the existing ones do not have a standard definition in the literature. We stress that the search trees within this section (§1.1.1) are *not* necessarily degree-balanced and all definitions here are applicable to all search trees whether they are balanced in any way or not. Finally, we must caution that the statements within each numbered list in this thesis are supposed to be read sequentially like a paragraph; otherwise there can be free variables.

1.1.1.1 Search Trees

(1) For $b \geq 2$, a b-way search tree T is a structure made up of nodes that are connected by links. A node can either be internal or external and any b-way search tree has at least one external node.

(2) If $b > 2$, then T and each of its internal nodes are said to be "multiway"; otherwise, they are said to be "two-way" or "binary". Note that this characterization is solely based on the chosen value of b.

(3) A particular node of T is distinguished as the root of T. This node is considered to be the topmost node of T and it can be either internal or external.

(4) An internal node u of T has between 2 and b child nodes (children) below it and they are arranged from left to right. Each of these nodes can be either internal or external. The child nodes of u are connected to u by links and they call u their parent. The degree of u is the number of children of u and is denoted $\Lambda(u)$. An internal node u is a k-node iff $\Lambda(u) = k$.

(5) An internal node u of T has $(\Lambda(u) - 1)$ keys drawn from a totally ordered set. (See Assumption 1.2.) The keys of u are arranged from left to right in-between the $\Lambda(u)$ children of u. The number of keys in u is denoted $\#(u)$ and it is also the size of u. It is easy to verify that $\#(u)$ ranges from 1 to $(b-1)$.

(6) An internal node is a leaf iff all of its children are external; otherwise it is a junction. Note that all leaves and junctions are internal nodes.

(7) An external node does not contain any key and has no children. Note that the root of a search tree can be external and this happens when the tree is a sole external node, which implies that the tree has no keys.

(8) Counting from left to right, the i-th external position of T is the position of the i-th external node of T. By structural induction, it can be shown that any b-way search tree with n keys has $(n+1)$ external positions regardless of the value of b.

(9) For any node u of T, a descendant of u is either a child of u or a descendant of a child of u. The node u is an ancestor of any of its descendants. Note that u is *not* a descendant *nor* an ancestor of itself.

(A) For any node u of T, the subtree of T rooted at u is denoted $T|^u$ and it comprises (i) the node u itself, plus (ii) all descendants of u, together with (iii) the links going among these nodes. A subtree of u is a subtree of T rooted at a child of u. Note that even if u is external, the subtree $T|^u$ still exists but u itself has no subtrees because it has no children.

(B) For T to be a valid search tree, a so-called "search tree relation" must hold among the keys in u and the keys in the subtrees of u. Moreover, it can shown that a subtree of T is also a b-way search tree. We will postpone the definition of this relation into §1.1.1.2 until further notation has been set up.

(C) Suppose T' is a subtree of T rooted at any node of T. The size of T' is denoted $|T'|$ and it is the total size of the internal nodes in T'. A subtree is empty iff its size is zero, which implies that an empty subtree must be an external node. Note that if $b = 2$, then the size of a subtree is also the number of internal nodes in it.

(D) For any node u of T, the depth of u is denoted $\mathrm{depth}(u)$ and it is the number of links on the simple path going from the root of T to u. The depth of T, denoted $\mathrm{depth}(T)$, is the maximum depth among all nodes in T. Note that the depth of the root of T is 0 and it is easy to verify that T is empty iff it has depth 0.

(E) The height of an external node is 0, and the height of an internal node is one plus the maximum height among its children. The height of a node u is denoted $\mathrm{ah}(u)$. Note that the height of an internal node is at least 1. Suppose T' is $T|^u$ for some node u in T. The height of T' is denoted $\mathrm{ah}(T')$ and it is $\mathrm{ah}(u)$. It is easy to verify that T' is empty iff it has height 0 and that the height of T' equals to the depth of T' by viewing the subtree T' as a b-way search tree itself.

With the above definitions in place, let us turn to the representation of search trees after the following remarks.

☞ In our usage, strictness and properness will be specifically noted. This is applicable to paths, sequences, sets, trees, etc. and we have already seen this in the usage of "subtree" in the above. However, we also note that some concepts have been or will be defined to be inherently strict or proper—an example being the descendants of a node, which do not include the node itself.

☞ The intention behind using $\mathrm{ah}(u)$ instead of $\mathrm{h}(u)$ is to better distinguish it within expressions once another notion of height is introduced in §2. The symbol "ah" is our mnemonic of actual height.

☞ Although one of Λ and # is arguably redundant, we are going to use them as a crude form of type-safety to guard against off-by-one errors. In particular, the keys of a node will always be indexed over # and the children over Λ when they are referred to directly. To help the reader distinguish them better, Λ is meant to resemble the two links of a degree-two node. . . to a certain degree. ☺

☞ We will borrow two conventions from [CLRS01, pp. 19–20, 438–439]. First, each of our arrays is left-to-right 1-based with a fixed number of allocated locations. The i-th location of an array A is denoted $A[i]$. Each location can store a fixed number of words and the size of an array is the number of locations allocated. Second, we distinguish among four types of names by their typesetting—constants are LIKE-THIS; fixed mathematical functions are like-this(x); identifiers or fields of an object are *like-this*$[x]$; and pseudocode procedures are LIKE-THIS(x).

1.1.1.2 Representations

Historically, there have been two common representations of search trees. Nievergelt and Wong [NW73], for example, distinguished between leaf-trees and node-trees when they studied the path lengths of binary search trees. Although Tarjan [Tar83a, pp. 9, 45] would call these representations "exogenous" and "endogenous" respectively, we allow ourselves to simply say leaf-store and node-store to avoid any casual mistakes. As the two representations share a large amount of detail and it will be useful to understand them both, let us start with their common points below.

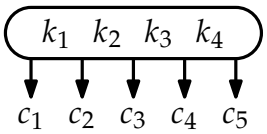
- common -

(1) A pointer to the root of T is stored in a field $root[T]$. Unless specifically indicated, this is the only field we store about T.

(2) An internal node u of T is represented as either two (leaf-store) or three (node-store) arrays, with each array entry defaulting to NIL.

(3) The first array is the key array and it has $(b-1)$ locations. It contains the fields to store the information that encodes each key of u. (See Assumption 1.2.) These fields are denoted $key_i[u]$ for $1 \le i \le (b-1)$ and we will simply drop the subscript when $b = 2$.

(4) The second array is the child array and it has b locations. It contains the fields to store the pointers that implements the links to the children of u. These fields are denoted $c_i[u]$ for $1 \le i \le b$. When $b = 2$, we will also use $left[u]$ and $right[u]$ to denote $c_1[u]$ and $c_2[u]$ respectively. On our side is a figure depicting the fields of a node when $b = 5$, with key_i abbreviated into k_i.



(5) In order for T to be a search tree, for $1 \le i \le \#(u)$, all keys in the subtree rooted at $c_i[u]$ must be less than $key_i[u]$, which in turn must be less than all keys in the subtree rooted at $c_{i+1}[u]$. This is the "search tree relation" we mentioned on page 5.

(6) Any non-NIL field must be packed towards the beginning of an array and all unused fields must be reset to NIL. Thus, the i-th key of u is

$key_i[u]$, the i-th child of u is $c_i[u]$, and the i-th subtree of u is the subtree rooted at $c_i[u]$.

(7) Though $\#(u)$ is not stored as a field in u, we remark that it can be computed by scanning through the key array until the first NIL is reached or all $(b-1)$ fields have been checked. The running time to compute $\#(u)$ and by extension $\Lambda(u)$ is $O(b)$.

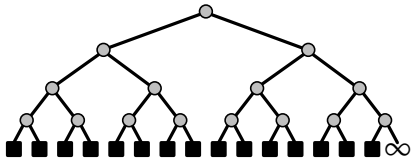
- leaf-store specific *(cont.)* -

(8) Each external node in a leaf-tree stores an item associated with a key and a terminal bit to mark the exterior of T.

(9) Counting from the left, the item at the i-th external node is associated with the i-th key of T. Observe that if this key is stored in an internal node u, then the corresponding external node is the rightmost external node of the subtree preceding this key in u. This is also the node preceding u in an in-order traversal of T.

(A) The terminal bit is a specification of whether a child pointer $c_i[u]$ is pointing to an internal or an external node. We remark that the idea of distinguishing a pointer by a bit stored at its destination will be used in several occasions inside this thesis.

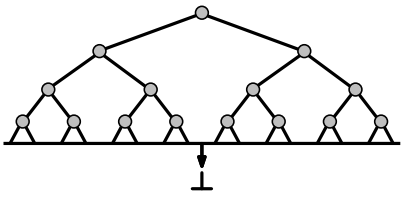
(B) Since there are $(n+1)$ external nodes but only the leftmost n of them have items to store, the rightmost external node will be assigned as the ∞ sentinel. Notice that this implies an empty leaf-tree is represented by the ∞ sentinel and has height 0.



- node-store specific *(cont.)* -

(9) If T is a node-tree, then the third array of u is the item array and it has $(b-1)$ locations. It contains the fields to store the items associated with $key_1[u]$ through $key_{\#(u)}[u]$. However, since we do not concern ourselves with these items, these fields will not be named.

(A) The external nodes in a node-tree serve no purpose. Similar to an approach taken in [CLRS01, p. 275], we represent all external nodes as *one* sentinel denoted $\perp$. Notice that this implies an empty node-tree is represented by the $\perp$ sentinel and has height 0.



(B) Finally, since $\perp$ is unique, we can test if a node is $\perp$ in $O(1)$ time and thus there is no need for terminal bits in a node-tree.

Leaf-Store vs. Node-Store. Having seen the two representations, let us remark about a space-time tradeoff that exists between leaf-trees and node-

trees. Suppose a key of our interest has been located in an internal node u. Since the item associated with this key is stored along with u in a node-tree, it can be readily retrieved in $O(1)$ time. But to do so in a leaf-tree would require reaching the corresponding external node, which can be *unboundedly* far away from u. As such, the running time in this scenario is clearly in favor of node-trees.

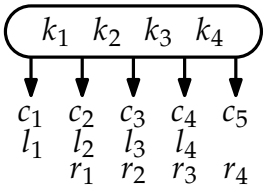
Node-trees, however, have two relative drawbacks. First, its space efficiency is lower because a total of $(n+1)$ of its pointers are used to point to $\perp$. This is true even if we were not to implement $\perp$, as the storage for these pointers would still have been allocated. Second, the decision procedure in a node-tree actually requires an *extra* comparison (equality) at each node and this makes the running time comparison above a bit unfair. The extra comparison is most evident when $b = 2$ in which case we are essentially making a three-way comparison at each internal node. This will be pointed out again when we review a search procedure for node-store degree-balanced search trees on page 12.

Node-Store is Default. At this point, let us remind the reader about Assumption 1.3 on page 3, which states that every search tree in this thesis is a node-tree unless it is specified to be otherwise. We remark that unlike the other two assumptions we made, this assumption is *not* a simplification in this thesis. In fact, quite the opposite is true—leaf-store is simply a special case of node-store from the perspective of our work, as will become clear in the chapters to come.

1.1.1.3 More About Search Trees

Using the notation developed in §1.1.1.2, let us present the remaining definitions related to search trees. Note that these definitions are independent of the representation used. Also, as a large number of the following definitions are symmetric with respect to left vs. right, we will define only the version on the right hand side when symmetry applies.

Generalized Left/Right and Key Pointers. To make multi-way nodes easier to handle, let us introduce the following definitions. As a visual aid, on our side is a b-way node u for $b = 5$ and $\#(u) = 3$, which implies that $key_4[u]$ and $c_5[u]$ are NIL. Note that the figure abbreviates *key*, *left*, and *right* into k, l, and r respectively. In what follows, let u be an internal node of T.



(1) For $1 \le i \le (b-1)$, the i-th left and the i-th right children of u are $c_i[u]$ and $c_{i+1}[u]$ respectively. These two children are also respectively the left and right children of $key_i[u]$.

(2) The child array will be aliased with the correct offset so that these two children can also be referred to as $left_i[u]$ and $right_i[u]$. When $b = 2$, $left_1[u]$ and $right_1[u]$ may also be collapsed into $left[u]$ and $right[u]$.
(3) A key pointer pointing to $key_i[u]$ is a tuple comprising a pointer to u and the position i, written as (u, i). Contrast this with the node pointers that are stored in, say, $root[T]$ and $c_i[u]$.

The benefit of making the above definitions is twofold. First, they allow us to think of a multiway node *as if* it were a binary node, albeit with an offset to indicate which key we are centering on. For example, instead of thinking about the subtree succeeding $key_i[u]$ in u, we can think of it as the i-th right subtree of u. And—this being our most important criteria—we know the root of this subtree is denoted $right_i[u]$ *without* hesitating if we have made an off-by-one mistake in the subscript. We also note that since $left_i[u]$ and $right_i[u]$ are notions associated with $key_i[u]$, they should also be indexed over $\#(u)$ but not $\Lambda(u)$. This explains the comment on page 5 in which we mentioned that the children of a node should be indexed over Λ *when* they are referred to directly. (Both $left_i$ and $right_i$ are aliases.)

Second, the above definitions enable our degree-balanced search tree algorithms to more closely resemble their two-way specializations, which are usually more intuitive. To see this, observe that $c_i[u]$ is a "left" child iff $i < \Lambda(u)$ and a "right" child iff $i > 1$. In other words, a child is a left (resp. right) child of u iff it is not the rightmost (resp. leftmost) child of u. Note that we would classify some child of u as both left *and* right simultaneously and this is not a bug. This notation actually eliminates many negations in our algorithms, which makes our algorithms easier to reason about.

Remark 1.4. The reader may notice that our key pointer notation (u, i) does not reveal that we are storing a pointer to u instead of the value of u itself. Indeed, all nodes are passed by reference in this thesis and therefore the distinction on the level of indirection will only be made in our definitions. For example, in the case of a key pointer, we have explicitly defined the first value in the tuple to be a pointer to a node.

Paths. We will often refer to certain paths in a search tree by a nominal and, in all but one case, the starting node of the path. In what follows, let u denote any node of T and let x be $key_i[u]$ for some i ranging from 1 to $\#(u)$. Note that these paths are illustrated in Figure 1.1.

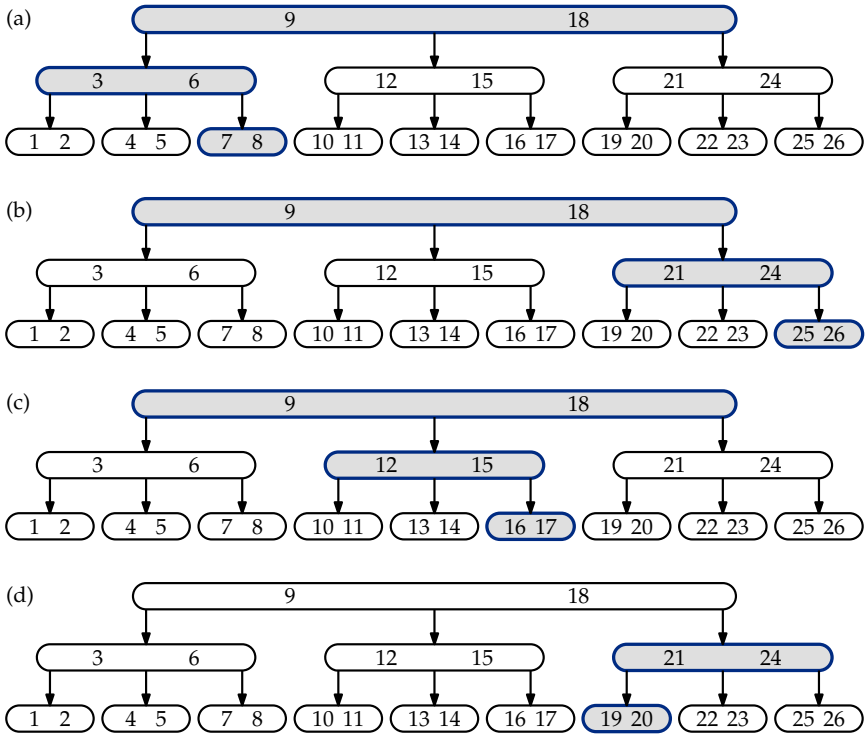


Figure 1.1 – Some examples to the path definitions starting on page 9: (a) access path of 8 when top-down; ancestral path of 8 when bottom-up; (b) right spine of root; also 2-nd right spine of root; (c) 1-st right spine of root; (d) 2-nd right-left spine of the root.

(1) The access path of u or x is the unique simple path from $root[T]$ to u.
(2) The ancestral path of u or x is the access path of u but with its nodes appearing in the opposite order. Note that this path contains u and it is specifically not named "the ancestor path" because we have defined ancestor as a proper notion.
(3) The right spine of u or $T|^u$ is the node u followed by, if present, the right spine of the rightmost child of u.
(4) The i-th right spine of u or $T|^u$ is the node u followed by, if present, the right spine of $right_i[u]$. The right spine of x is this path also. We remark that the i-th left spine is defined with respect to $left_i[u]$.
(5) The i-th right-left spine of u or $T|^u$ is the left spine of the i-right child of u. The right-left spine of x is this path also. Notice that this path does *not* contain u and it is the sole exception here.
(6) Finally, following [CLRS01, p. 298], we will measure the length of a path in a search tree by the number of *nodes* on it.[1]

1. This is also why we would not define search trees as graphs. The subject of the latter has a time-honored tradition of measuring the length of a path by the number of edges on it.

Family Relations. Besides the child relation which we have been focusing on, several other family-inspired relations also appear in a search tree. In what follows, let p be an internal node of T and let u and w be any two nodes of T. Also, let x be $key_i[u]$ for some i ranging from 1 to $\#(u)$.

(1) The node p is the parent of u or x iff u is a child of p. We remark that by "the parent of a node (or of a key)", we are always referring to a node—which is p in this case.
(2) Suppose u is $c_j[p]$ for some j ranging from 1 to $\Lambda(p)$. The right sibling of u is $c_{j+1}[p]$ if $(j+1) \leq \Lambda(p)$; otherwise u has no right sibling.
(3) Listing all nodes in T at the same depth as u from left to right, the right peer of u, if present, is the node succeeding u. Note that the right peer of u may or may not be the right sibling of u, meaning they may not have a common parent.
(4) Suppose u is $c_j[p]$ for some j ranging from 1 to $\Lambda(p)$. The right parent key of u or x is defined as follows—if u is a left child of p, then it is $key_j[p]$; otherwise, it is the right parent key of p if this key exists. An alternative definition that is not based on structural induction is as follows—the right parent key of u or x is the leftmost key on the access path of u that is also to the right of u. In Figure 1.1, the right parent key of 8 is 9 and the right parent key of 10 is 12.
(5) The right parent of u or x is the node containing the right parent key of u. Notice that it may contain keys that are to the *left* of u when $b > 2$.
(6) When the right parent of u or x does not exist, its right parent and its right parent key are both defined to be ∞. Observe that this happens *iff* u is on the right spine of T. We remark that ∞ is merely a concept in our analysis and it is *not* stored in any form. Moreover, note that this also explains why the rightmost external node of a leaf-tree is named the ∞ sentinel.
(7) To simplify our logic, we further define $key_0[u]$ and $key_{\#(u)+1}[u]$ to be the left and right parent keys of u respectively. These are again merely concepts in our analysis and are not stored in any form. Note that the subscript of the latter is *not* defined as b. It is easy to verify that the former is smaller than $key_1[u]$ and the latter is larger than $key_{\#(u)}[u]$. Therefore, the relation $key_i[u] < key_j[u]$ iff $i < j$ still holds.
(8) A right ancestor key of u or x is either the right parent key of u or a right ancestor key of the right parent key of u. An ancestor key of u or x is either a left or right ancestor key of u.
(9) A right ancestor of u or x is an ancestor of u that contains a right ancestor key of u.

(A) Suppose w is the right parent of u. The node w is classified as trivial iff (i) w is the parent of u, or (ii) w is ∞ and u is $root[T]$. This classification also applies to the right parent key, the right ancestor key, and the right ancestor as well. In Figure 1.1, the left parent of 8 is trivial, but its right parent is nontrivial. Note that if u has a nontrivial right parent, then its left parent is trivial. However, the converse is not always true unless $b = 2$. As a counterexample, both left and right parents of 5 are trivial in Figure 1.1.

1.1.2 Dictionary Operations

Having defined our search tree representation, let us review how to perform all three dictionary operations on a node-store degree-balanced search tree. (This marks the last time we will hint at node-store.) Not only can we use this to introduce a number of relevant terms, we can also point out how several of the above definitions actually apply when degree-balanced search trees are concerned. As all three algorithms start at the root of the tree, we classify them as "root-start".

Let us begin with an observation—in a degree-balanced search tree, all leaves appear at the same depth and a node u is a leaf iff $c_1[u] = \perp$, which is a worst-case $O(1)$-time test. In what follows, let T be a degree-balanced search tree and let x be a key that may or may not appear in T.

Search. To search for x, start with $root[T]$ as the current node u. As an option, also initialize an empty stack to store tuples comprising a node pointer and a child position. This stack will be called the parent stack.

⟪ search for x at subtree rooted at u ⟫

1> If u is external, i.e., $u = \perp$, then stop and report that x is not found.

2> Identify the position i of the leftmost key that is greater than or equal to x by, for example, scanning the keys in u from left to right. If all keys in u are smaller than x, then let i be $(\#(u) + 1)$.
(Post: $key_{i-1}[u] < x \leq key_i[u]$)

3> Push the tuple (u, i) into the parent stack, if maintained.

4> Test if $key_i[u] = x$.

4.1> If so, return the tuple (u, i) as the key pointer to x.

4.2> Otherwise, descend by setting u to $c_i[u]$ and going back to step 1. ∎

Remark 1.5. The path traversed by the above algorithm is the access path of x, which is represented by the parent stack we built in step 3. The tuple at the top of the stack is a key pointer to x, and each remaining tuple in the stack stores a pointer to an ancestor of x and the position of the child

we descended into. As an aside, the test in step 4 is the extra comparison we mentioned on page 8. In a leaf-tree, we always descend into $c_i[u]$.

Insert. To insert x into T, first search for it in T while maintaining the parent stack. If x is found, then report it is already present and stop. This is because we do not handle duplicate keys in a search tree as stated in Assumption 1.2. Otherwise, the search will have terminated at the leaf v^* at the top of the parent stack. We proceed by inserting x and (technically) its left child $\perp$ into v^* using the following procedure, in which x_L will be $\perp$ initially.

⟪ insert (x_L, x) into top node of parent stack ⟫

1> Pop the parent stack to obtain (u, i).

2> By shifting, add x to u as its new i-th key and attach w to u as its new i-th child.

3> If $\Lambda(u) \leq b$, stop. No further work needs to be done.

4> Otherwise, u is said to have become over-full. Any over-full node can be handled by promoting one of its keys in a node fission.

⟪ fission of u ⟫

4.1> Let m be $\lceil \frac{\#(u)}{2} \rceil$ and let u_m be $key_m[u]$ (a median key of u).

4.2> Split u at u_m into the following three parts.

(1) Let u_L be a *new* node with the first $(m-1)$ keys and the first m children of u.

(2) Let u_m be a standalone key.

(3) Let u_R be what remained at u.

Notice that u_R is still attached as the i-th child of its parent, if present.

4.3> We begin to promote u_m by testing if the parent stack is empty.

4.3.1> If so, we know u (now u_R) is the root. Create a new root containing u_m as its only key, with u_L and u_R being its first and second children respectively. This case is terminal.

4.3.2> Otherwise, the parent p of u will be at the top of the parent stack. Insert (u_L, u_m) into p recursively.

Delete. To delete x from T, first search for it in T while maintaining the parent stack. If x is absent, report this and stop. Otherwise, by peeking, suppose the top of the parent stack is (u, i). Run the following procedure to determine a particular leaf to be denoted v^*. (We note that the idea of this procedure first appeared in the work of Hibbard in [Hib62].)

⟪ Hibbard replacement ⟫

1> If u is a leaf, then let v^* be u and proceed to delete x from v^*. This case is terminal.

2> Otherwise, we know that u has some children. Continue to search for x in $right_i[u]$ and build up the parent stack. The search will fail at a leaf v^*, which will be the last node of the left spine of $right_i[u]$. In our definition, this spine is also known as the i-th right-left spine of u.

3> Let x^+ be $key_1[v^*]$, which will also be the successor key of x. Replace x with x^+ inside u and proceed *as if* the key to be deleted is x^+ from v^*. ∎

Observe that after the above procedure, v^* denotes the leaf containing the correct key to be deleted. Proceed with the following procedure using the correct key.

⟪ delete x from top node in parent stack ⟫

1> Pop the parent stack to obtain $x = (u, i)$ and let x_L be $left_i[u]$.

2> By shifting, remove x from u and also detach x_L from u.

3> If $\Lambda(u) \geq a$, stop. No further work needs to be done.

4> Test if the parent stack is empty. If so, u is the root and we stop after handling one of the following two cases.

4.1> If u is nonempty, stop. The root can have as few as one key.

4.2> Otherwise, stop after deleting u and letting $root[T]$ point to x_L.
(Pre: parent stack is nonempty ⇒ u has parent ⇒ u has a sibling)

5> In this case, u is said to have became under-full. Any under-full node can be handled by demoting a key from its parent in a node fusion with a sibling.

⟪ fusion of u with a sibling w ⟫

5.1> By symmetry assume that u has a sibling w on the right. Pop the parent stack to obtain (p, j). Notice that u and w are, respectively, $left_j[p]$ and $right_j[p]$.

5.2> Begin to demote $key_j[p]$ by first removing it from p and also detaching u from p. Note that w is still attached to p as (the new) $left_j[p]$.

5.3> Using $key_j[p]$ as the splitting key, join u into the left end of w. This finishes the demotion of $key_j[p]$ and the fusion of u with w.

6> At this point one of the following two cases applies.

(1) If $\Lambda(w) > b$, then subject w to a fission. This case is terminal because (i) $\Lambda(p)$ has been restored to its original value due to the promoted key, and (ii) $\Lambda(w)$ and the degree of the new node created in the fission are both upperbounded by $\frac{a+b}{2} < b$.

(2) Otherwise, $\Lambda(w) \leq b$, but $\Lambda(p)$ has just been decremented due to the demotion of $key_j[p]$. Let x_L and u be, respectively, u and p and then go back to step 3. Notice that from the perspective of p, both $key_j[p]$ and $left_j[p] = u$ have both been taken care of just like what step 2 would have done. ∎

Running Time. On a degree-balanced search tree with n keys, the running time of any of the three operations above is $O(b \log_a n)$ because we spend $O(b)$ time at each node and we visit $O(\log_a n)$ nodes. By construction, the degree invariant is preserved by both the insertion and the deletion algorithms. For the depth invariant, observe that a degree-balanced search tree can change its height only at its root because any restructuring must start at the leaf level and propagate up. Note that if a multiway node is implemented as a sorted array—which they are in this thesis—then the multiway decision in step 2 of the search algorithm can be improved from a left-to-right scan to a binary search. Although this does improve the search time, insertions and deletions see no improvement precisely because sorted arrays are used.

1.1.3 Splits, Joins, and Sorted List Operations

Besides the three dictionary operations, degree-balanced search trees also support splits and joins in $O(b \log_a n)$ time. The algorithms for these operations are largely similar to their AVL-trees [AVL62] equivalents devised in the thesis of Crane [Cra72, §1.7]. Instead of spelling out these algorithms, here we will give their specifications only.[2]

Furthermore, we also note that degree-balanced search trees support sorted list operations such as merging, union, intersection, and difference. The algorithms for these operations will be covered in §1.2.4.2.

- ☞ In a join, we are given a key x and two degree-balanced search trees T_L and T_R, with the condition that every key in T_L is less than x, which is in turn less than every key in T_R. Let n_L and n_R be the number of key(s) in T_L and T_R respectively. The result of joining the triple (T_L, x, T_R) is a new degree-balanced search tree T that contains the $n = (n_L + 1 + n_R)$ keys involved.
- ☞ A common variation of join is catenation in which we are given only T_L and T_R. The nontrivial case is often implemented by first removing the largest key of T_L or the smallest key in T_R, which is then used to join what remains of the two trees.
- ☞ In a split, we are given a degree-balanced search tree T of size n and a key x that may or may not be in T. The result of the split is a triple

[2] Interested readers can solve Problem 18.2 in [CLRS01], which specifically asks about these algorithms on $(2,4)$-trees augmented in a certain way. We do note that there are algorithms that do not depend on any augmentation such as the one used in the problem and still run in asymptotically the same amount of time.

$(T_L, x^?, T_R)$ with T_L and T_R taking their meanings from the above. The option $x^?$ will either be a key pointer to x if x is in T, or NIL if otherwise.

1.1.4 History of Degree-Balanced Search Trees

The earliest degree-balanced search trees are the $(2,3)$-trees [AHU74, §4] designed by Hopcroft in 1970 (year from [AHU82, p. 197]). As their name suggests[3], the degree of an internal node in a $(2,3)$-tree ranges from two to three. It's commonly agreed that $(2,3)$-trees are simple and highly intuitive, and as Hopcroft observed, they are very versatile for internal memory applications because they support dictionary operations as well as splits and joins in logarithmic time.

Independent of the then-unpublished $(2,3)$-trees, Bayer and McCreight [BM72] introduced the class of $(a, 2a-1)$-trees in 1972 and named the trees in this class "B-trees". Unlike $(2,3)$-trees, B-trees are designed in consideration of the external memory model in which we measure the running time of an algorithm by the number of pages accessed in the underlying hardware. By treating each page as a node, the value of a can be made quite large in a B-tree and therefore the height of a B-tree can be significantly smaller than that of a corresponding $(2,3)$-tree. As the number of pages accessed is proportional to the height of the tree, in practice the performance advantage of B-trees is very significant in the external memory setting. B-trees have many variants and we refer interested readers to both [Knu98, §6.2.4, pp. 486–489] and [Com79] for more information.

A particularly notable refinement of B-trees is the class of weak B-trees by Huddleston and Mehlhorn [HM82]. These trees—known as (a,b)-trees in [HM82]—are highly similar to B-trees but with the critical requirement that $b \geq 2a$. It follows that the minimal weak B-tree in terms of the degree bounds is a $(2,4)$-tree. Huddleston and Mehlhorn were able to show that the extra degree(s) in a weak B-tree makes it robust, meaning that if we discount the search time at the beginning of all inserts and deletes, then the restructuring cost of a length-m access sequence is $O(m)$. This is in stark contrast with B-trees, which at size n can incur $\Theta(mb \log_a n)$ time with the same access sequence. B-trees and generally the class of $(a, 2a-1)$-trees are thus said to be "fragile".[4] Weak B-trees are in fact parameterized into several variants and they can be found in [HM82] as well as the thesis of Huddleston [Hud81]. We also note that a highly-related variant known

3. The choice of using parenthesis in the notation of (a,b)-tree is customary, even though square brackets may better reflect the closed nature of the degree interval.

4. Indeed, weak B-trees are robust, but (the presumably strong) B-trees are fragile.

as "hysterical B-trees" have been independently introduced by Maier and Salveter [MS81]. (The choice of this name is explained in [MS81].)

The importance of degree-balanced search trees in practice is hard to be overstated. By 1979, a mere seven years after their introduction in [BM72], B-trees and its variants were already considered to "have become, *de facto*, the standard organization for indexes in a database system" [Com79, p. 1]. This trend has stayed true ever since and a recent survey on external memory algorithms by Vitter [Vit08, p. 6] confirms that B-trees are still considered to be "the most widely used online external memory data structure for dictionary operations and one-dimensional range queries".

B-trees have also gained extra momentum in the last few years due to their cache-oblivious adaptation by Bender, Demaine, and Farach-Colton [BDFC05]. Compared to their traditional variants, cache-oblivious B-trees adapt to the underlying hardware without knowing the actual page size. Consequently, a single implementation can be deployed on multiple platforms *sans* the time-consuming process of hardware-specific tuning.

On the internal memory side, degree-balanced search trees can also be said to be immensely popular in applications, albeit in the form of red-black trees of Guibas and Sedgewick [GS78]. For example, our recent inspection of the source code of three major open-source operating system kernels[5] shows that red-black trees are used as their standard dictionary data structure.

Technically speaking, red-black trees are not really degree-balanced since they are *binary* search trees. However, red-black trees can be and are often understood via their isometry to $(2,4)$-tree [Tar83a, p. 49; Sed98, §13.4] because they did start out as a binarization of B-trees. (See the sequence [Bay71], [Bay72], and [GS78].) Having said that, we should be careful not to overlook the many benefits that this binarization brings. Besides leading to beautiful and simple adaptations such as AA-trees [And93] (as named in [Wei99]) and the recent left-leaning variant [Sed08], red-black trees also have a distinct advantage over $(2,4)$-trees regarding the amount of restructuring in the worst case [Oli80; Tar83b; OW92]. (The reader, of course, already knows that this advantage does not hold in the amortized case.)

5. Linux http://www.gelato.unsw.edu.au/lxr/source/include/linux/rbtree.h
FreeBSD http://svn.freebsd.org/viewvc/base/stable/7/sys/sys/tree.h
OpenBSD http://www.openbsd.org/cgi-bin/man.cgi/usr/include/sys/tree.h

1.2 The Dynamic Finger Property

This thesis is primarily concerned with the dynamic finger property of various search trees. Although this property has its underpinnings in an *operation* known as "finger search" invented over 30 years ago by Guibas et al. [GMPR77], for the sake of clarity let us first define this property on a clean slate in §1.2.1 and then come back to finger search in §1.2.2.

We will start with a few preliminary definitions. Consider a data structure representing the sorted list A of n items with keys $\langle\langle a_1, a_2, \ldots, a_n \rangle\rangle$ and let x and y be two not-necessarily-distinct keys of A.

(1) The rank of x with respect to A is its position in A. In other words, if x is a_i, then the rank of x with respect to A is i. Any key that is not in A does not have a rank with respect to A in our definition.
(2) The key space of A is an imaginary space spanned by the minimum and the *current* maximum ranks of A. Notice that the former is always 1, but the latter can change with the number of keys in A.
(3) The distance between x and y with respect to A—denoted $\text{dist}_A(x, y)$—is the absolute difference between their ranks with respect to A. This distance is also known as the rank distance between the two keys and must range from 0 to $(n-1)$. Note that when the context is clear, we will drop the subscript in dist_A.

1.2.1 Definition

In what follows, we will offer two definitions of the dynamic finger property. The first definition is phrased in the lingo of competitive analysis pioneered by Sleator and Tarjan [ST85a]. This is what we prefer and what we will be using for the rest of this thesis. But to address what might appear to be over-simplifications of this first definition, we also offer an operational definition that fully justifies these simplifications from a theoretical standpoint.

Competitive Analysis. Recall that in the competitive analysis of online algorithms, we are given a length-m access sequence $\sigma = \langle\langle \sigma_i \rangle\rangle$ where each σ_i is one of n different possible accesses (a.k.a. requests). We are interested to serve σ by an online algorithm ALG, and we compare its cost to that of an offline optimal algorithm OPT.

For any algorithm Q, let $Q(\sigma_i)$ denote the cost incurred by Q to serve σ_i for $1 \le i \le m$ and let $Q(\sigma)$ denote the total cost summing over σ. For good measure, $Q(\sigma_i)$ is required to be at least 1 and thus $Q(\sigma) = \Omega(m)$. If

there exist both a function $f(n)$ and a constant c such that for all possible choices of $m = \Omega(n)$ and σ of length m the relation

$$ALG(\sigma) \leq f(n) \times OPT(\sigma) + c \tag{1.1}$$

holds, then ALG is said to be $f(n)$-competitive against OPT. Since we will be ignoring constant multiples, we also allow ourselves to say "$O(g(n))$-competitive" as long as there is some $f(n) = O(g(n))$ that satisfies (1.1). Finally, notice that we only consider long access sequences, meaning $m = \Omega(n)$ as stated above.

Dynamic Finger Property. For our purpose, we consider our data structure to be an online algorithm that maintains a dictionary A with a fixed set of n keys. The access sequence σ is a sequence of m accesses, with each access σ_i being one of the keys of A. The cost of the data structure to serve σ_i is the cost incurred in searching for σ_i. For a data structure in the comparison-based model, this cost is the number of comparisons made in the search for σ_i.

The offline algorithm in this context is not exactly an optimal algorithm *per se*. Instead, it is used to model the desired upperbound on the cost of serving the access σ_i. Let d_1 be n and let d_i be $\text{dist}_A(\sigma_{i-1}, \sigma_i)$ for $i \geq 2$. The offline algorithm is considered to incur a cost of exactly $\lg \max(2, d_i)$ to serve σ_i. We say that the dynamic finger budget of σ_i is $\lg \max(2, d_i)$, which we abbreviate to $\lg d_i$.

With the above definitions, a data structure is said to have the dynamic finger property if it is $O(1)$-competitive against the above offline algorithm. Simply put, this means a data structure with the dynamic finger property serves σ in $O(\sum_{i=1}^{m} \lg d_i)$ time when $m = \Omega(n)$. Splay trees [ST85b], for example, have the dynamic finger property as shown by a renowned result of Cole [Col00].

Technicalities. Note that in our formulation of (1.1), we have intentionally considered only the cost of serving the *entire* access sequence. This allows the performance guarantee of our data structure to be either worst-case or amortized. When we want to highlight the difference between the two, we will say that a data structure has the dynamic finger property in the worst or amortized case. Furthermore, by taking expectation over the random bits used to build the data structure, we may have even chosen to allow randomization in (1.1). Finally, the conditions $m = \Omega(n)$ and $Q(\sigma_i) \geq 1$ together allow us to absorb an $O(n)$ term in (1.1). This term plays a significant role in what follows.

An Operational Definition. On the surface, the above definition of the dynamic finger property may appear to have the following issues.

(1) All searches must be successful.
(2) The key set is fixed and there is no support for insertions and deletions.
(3) Even if insertions and deletions are supported, as a consequence of issue (2), only the time spent in the searches has been accounted.

To see how these issues can be addressed, let us consider an alternate definition that is based on the operations of a more realistic data structure.

For simplicity of exposition, we will assume that there are two *implicit* sentinels $-\infty$ and ∞ at the two ends of A. It is crucial to note that they do not affect the rank of any keys, and although we will denote their ranks as $-\infty$ and ∞, these should be considered to be one less/greater than the minimum/current maximum ranks in A.

To address issue (1), let us define the rank of a search *operation* to be the rank of the key at which the search terminates. This key, possibly a sentinel, is also known as the terminating key of the search. Denoting the search target by x, this key is defined with respect to the direction of the search and is (i) the least key larger than x either when this is the first search or when the most-recently searched target is less than x, or (ii) the greatest key less than x otherwise. With this definition, notice that even unsuccessful searches have ranks.

To address issue (2), we further define the rank of an insert operation to be the rank of the inserted key *after* it is inserted. This is also the rank of the key that is displaced by the inserted key. The rank of a delete operation is then defined symmetrically. Notice that all three dictionary operations are now supported and each of them has a notion of rank. An insert or a delete will be referred to as an update, and an update or a search will then be referred to as an access.

Now consider a length-m access sequence S made to a data structure representing an initially-empty A. Let s_i denote the key at the rank of the i-th access, and let N denote the maximum size of our data structure throughout S. We define the search sequence of S to be the length-m sequence $\sigma = \langle\langle s_i \rangle\rangle$. Notice that right before the time of every access, s_i is a key that is currently in A.

To address issue (3), we will use the notion of robust balancing introduced on page 16. Recall that a degree-balanced search tree supports robust balancing if the total restructuring cost of a length-m access sequence is $O(m)$. Notice that the notion of robustness is in fact applicable

to dictionary data structure, although it won't be a useful notion, say, for a hash table. In any case, as long as our data structure is robust, the search cost and the restructuring cost can be accounted for separately. This is precisely we show next.

Consider the search sequence $\langle\langle \sigma_i \rangle\rangle$ transcribed from any given access sequence S of length m. Let d_1 be 0 and let d_i be $\text{dist}_A(\sigma_{i-1}, \sigma_i)$ for $i \geq 2$. We say that a data structure "has the dynamic finger property" if (i) it is robust, and (ii) the search time incurred in the i-th access is asymptotically within the dynamic finger budget of $\lg \max(2, d_i)$, or $\lg d_i$ for short. Again, in general we allow this time bound to be worst-case or amortized, and when needed it can even be in the expected case by taking expectation over the random bits used to build the data structure. Summing the search cost and the restructuring cost, the running time bound on S for such a data structure is $O(\sum_{i=1}^{m} \lg d_i) + O(m) = O(\sum_{i=1}^{m} \lg d_i)$. This also justifies the running time bound in the competitive definition by setting n in (1.1) to N and using the search sequence here as the access sequence there. The reader is reminded that from this point on, we will switch back to the competitive lingo.

1.2.2 All About Fingers: Dynamic, Static, and Finger Search

Now that we are familiar with the dynamic finger property, let us use this section to address a number of "finger" concepts in one place.

Static Finger Property. Consider a data structure implementing a dictionary A of n keys and an access sequence σ of length m. Let f^* be a fixed key in A and let s_i be $\text{dist}_A(f^*, \sigma_i)$ for $1 \leq i \leq m$. Similar to how we defined the dynamic finger property, the offline algorithm in this case is considered to incur a cost of exactly $\lg \max(2, s_i)$ to serve σ_i. This cost is also known as the the static finger budget of σ_i. A data structure is said to have the static finger property if it is $O(1)$-competitive against this offline static finger algorithm. In other words, for $m = \Omega(n)$, the running time of such a data structure to serve σ is $O(\sum_{i=1}^{m} \lg \max(2, s_i))$.

The static finger property is implied by the dynamic finger property and therefore it is weaker. A proof of this can be found in [Wan06, Lemma 11, p. 14]. Combining this with the dynamic finger theorem of Cole [Col00], we can show that splay trees have the static finger property. We do note that prior to Cole's result, Sleator and Tarjan [ST85b] have already shown that splay trees have the static finger property when $m = \Omega(n \lg n)$.

Finger and Finger Pointer. In the context of the dynamic finger property, we usually *imagine* that there is a finger pointing at the key of the previous

access. The first access σ_1 defines the initial location of a dynamic finger, and any subsequent access σ_i moves it from σ_{i-1} to σ_i. This is in contrast with a static finger, which always stays at f^* as defined above in the context of the static finger property.

Corresponding to the imaginary concept of a finger is an *explicit* implementation of a finger pointer in some data structures. We will see two examples of this in §1.2.3. However, we must note that a finger pointer can also be more sophisticated than a pointer in the sense of the word in some programming languages. We will see a perfect and simple example of this when we show how heterogeneous red-black trees attain the dynamic finger property in §1.3.2.

Finger Search. For those data structures that do implement explicit finger pointers, finger search refers to a search procedure that takes a finger pointer in addition to a search target. In the context of degree-balanced search trees, this is in contrast with the logarithmic-time search procedure in §1.1.2 since the latter only takes a search target. In many cases, a finger search procedure can in fact take *any* finger pointer regardless of whether we are after the dynamic or the static finger property, and the running time of finger searching from a given finger f to x is logarithmic to $\text{dist}(f, x)$.

It is important to observe that $\text{dist}(f, x)$ is at most the size of the data structure. Therefore, a finger search always runs in logarithmic time and is never slower than the "typical search" asymptotically. Furthermore, it can yield an asymptotically smaller running time bound on access sequences where many accesses are close to the finger used in the corresponding access. We will study these sequences in §1.2.5.

Dynamic Finger Tree vs. Finger Search Tree. We make a distinction between two types of search trees, both of which support finger search and have the dynamic finger property.

(1) A dynamic finger tree is a search tree that supports finger searching from a dynamic finger. While some dynamic finger trees can support multiple dynamic fingers, often we have to limit the number of fingers to $O(1)$ in order for the running time bound to stay under the dynamic finger budget. As an example, the design of Kosaraju in [Kos81] is a representative dynamic finger tree that supports a constant number of dynamic fingers.

(2) A finger search tree is a search tree that supports finger searching from *any* key in the tree. Moreover, the starting keys in consecutive finger searches do not need to be related at all. A finger search tree can naturally simulate a dynamic finger tree by implementing the semantics

of any number of dynamic fingers, but the opposite is not true. In other words, finger search trees are more powerful than dynamic finger trees when finger searches are our only concern. As an example, the design of Brown and Tarjan in [BT80] is a representative finger search tree.

Usage. Since the two types are quite different in regard to their support of finger search, from now only finger search trees will be said to support finger search. When the context demands, however, we will still say a dynamic finger tree "supports a dynamic finger" or "supports a constant number of dynamic fingers", just as we have demonstrated above.

1.2.3 Two Example Data Structures

To better understand how data structures attain the dynamic finger property, let us turn to two simple examples. Incidentally, both of these data structures support finger searches from any key. But to retain simplicity, we will use them by implementing a finger pointer to the most-recently accessed key and start searches from it.

1.2.3.1 Sorted Arrays

Consider a sorted array A of n keys serving an access sequence σ of length m. To serve σ_1, we employ a binary search in A. This takes $O(\lg n)$ time and initializes a finger pointer at σ_1. Let f denote the array index of the key currently under the finger pointer.

Consider serving σ_i for $i \geq 2$ and assume the nontrival case where $\sigma_i \neq \sigma_{i-1}$. Furthermore, by symmetry assume $\sigma_{i-1} < \sigma_i$. We will use the following two-step procedure which Bentley and Yao [BY76] called an unbounded binary search.

⟪ forward unbounded binary search for σ_i from $A[f]$ ⟫

1> Sequentially add to f the doubling offsets 2^0, 2^1, 2^2, ... until either we have $\sigma_i \leq A[f + 2^h]$ or we hit the array boundary in which case $(f + 2^h) > n$. This step is known as a doubling scan (a.k.a. doubling search).

2> A binary search between the indices $(f + \lfloor 2^{h-1} \rfloor + 1)$ and $\min(f + 2^h, n)$ will yield the index of σ_i, which is where we address the finger pointer next.

To analyze the running time of this algorithm to serve σ, let r_i be the rank of σ_i with respect to A, which in this case is simply the index of σ_i in A. Observe that (i) h is proportional to $\lg(r_i - r_{i-1})$, which is evident

in the stopping condition of the doubling scan, and (ii) both the doubling scan and the binary search take $O(h)$ time. Therefore, the running time is $O(\sum_{i=1}^{m} \lg d_i)$, with d_1 being n and d_i being $|r_i - r_{i-1}|$ for $i \geq 2$. This shows that any sorted array has the dynamic finger property when we search it using an unbounded binary search. The reader is invited to contrast this with the bounded binary search which we used to establish the initial position of f.

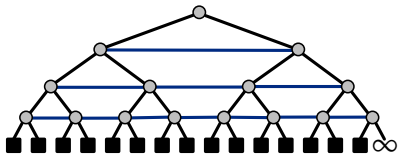
Bootstrapping. Before we go on, let us point out that the algorithm above is still using a linear scan to identify h and so we may *recursively* bootstrap this phase using another unbounded binary search. Even though this will not yield an asymptotic improvement since the bottleneck in step 2 remains, bootstrapping for a *constant* number of times can still be useful and later on we will see it used to our benefit in §5.

Remark 1.6. The above algorithm is Algorithm B_1 in the classic paper on unbounded searching by Bentley and Yao [BY76]. However, it was simply called "binary search" in the paper and the prefix was added by us to distinguish it from the bounded version of binary search. If we bootstrap Algorithm B_1 once in the first phase, then we will arrive their Algorithm B_2, namely "double binary search".

1.2.3.2 Level-Linked Degree-Balanced Search Trees

Our second example is the class of level-linked degree-balanced search trees, which must be in the *leaf-store* representation for a reason that will be explained later. Each internal node u is augmented to store three extra pointers, all defaulting to NIL. The first is a parent pointer $p[u]$ that points to the parent of u, if present. The remaining two are the left and right level pointers of u and they point to the left and right peers of u, if present. The two fields that store the level pointers will be unnamed since they are used only once in the algorithm below.

As an example, on our side is a minimal level-linked $(2,4)$-tree of 15 keys in which every internal node happens to have degree 2. Notice that all edges in the figure are bidirectional *except* the ones between the internal and the external nodes.



Algorithm. Consider a level-linked leaf-store degree-balanced search tree T of size n. We maintain a pointer to a leaf f of T that is the parent of the external node f' containing the most-recently accessed item. Notice that

when f' is the rightmost child of f, the key corresponding to f' is actually in the right parent of f but not in f itself.

To establish the initial location of f, we use a root-start search to serve σ_1 and leave f at the last leaf we visit. This takes $\Theta(b \log_a n)$ time. Let x be the target key of a subsequent search. The search for x will start at the current location of f and follow a particular path from f to the external node of x, also leaving f at the last leaf we visit. We remark that this path is not always one of the shortest path(s) between f and the external node of x, but it is simple in the technical sense of the word and its length can only be off by an additive constant in the worst case. Since this algorithm always starts a search from a leaf, we will also call it a leaf-start search in contrast to the root-start search on page 12.

The following is an algorithm that determines our desired path between f and the external node of x when $key_1[f] \le x$. The case of $x \le key_{\#(f)}[f]$ can be handled symmetrically. The two cases indeed overlap in the trivial case when $key_1[f] \le x \le key_{\#(f)}[f]$, but this does not affect the correctness of our algorithm.

Our algorithm maintains a node pointer q and the invariant that the node pointed by q contains a key less than or equal to x. Initially, q is set to f and this satisfies our invariant since $key_1[f] \le x$. Let u denote the node pointed by q and let w denote be the right peer of u, if present.

⟪ finger search for x in T using f ⟫

- ascend q to exit node -

1> Locate an internal node known as the exit node as follows.

1.1> Stop if u is $root[T]$ or any other node on the right spine of T. Note that either of these conditions can be tested in $O(1)$ time by utilizing the parent and right peer pointers. The exit node is u, and the invariant continues to hold because we did not move q.

1.2> Since u is not on the right spine of T, we know w exists. Test if $x < key_1[w]$. If so, stop and the exit node is u. The invariant continues to hold because we did not move q also.

1.3> Reaching this step, we must have $x \not< key_1[w] \Rightarrow key_1[w] \le x$. Start by computing $\#(p[u])$ in $O(b)$ time.

1.3.1> If u is a left child of $p[u]$, i.e., $u \ne right_{\#(p[u])}[p[u]]$, then move q upward to $p[u]$. The invariant continues to hold because both u and w are children of $p[u]$ and the key separating u and w in $p[u]$ is less than $key_1[w]$, which in turn is less than or equal to x.

1.3.2> Otherwise, move q sideways to w. The invariant continues to hold because $key_1[w] \le x$.

1.4> Now that q has been updated, go back to step 1.1.
(Post: u on right spine of $T \vee x < key_1[w]$)
- locate exit key in u -
2> Scan for the rightmost key in u that is less than or equal to x. Suppose this key is $key_j[u]$. This key is guaranteed to exist inside u due to the invariant and is known as the exit key of the search.
- descend from u -
3> Start by testing if $key_j[u] = x$.
3.1> If so, search for the external node containing x by descending into the subtree rooted at $left_j[u]$. The target node will end up being the rightmost external node of this subtree.
3.2> Otherwise, we must have $key_j[u] < x$ and we test if $j < \#(u)$.
3.2.1> If so, search for the external node of x by descending into the subtree rooted at $right_j[u]$.
3.2.2> Otherwise, *in addition* to searching the subtree rooted at $right_j[u]$, also search the subtree rooted at $left_1[w]$ iff w exists.

Analysis. Let us start by observing the path followed by q. In step 1, it is critical that q moves upward whenever u is a left child of its parent. This ensures that q never moves sideways twice, implying that q ascends at least half of the times within this step.

Fix u at its final value, namely the exit node, and let the height of u be h. With the above observation, the length of the path between f and u is at most $2h$ and so the running time of step 1 is $O(bh)$. As the running time of step 2 is $O(b)$, the time it takes us to locate the exit key is $O(bh)$. Furthermore, the length of the path from u to the external node of x is at most $(h+2)$, and therefore the time spent in step 3 is also $O(bh)$.

What remains is to relate h and $\mathrm{dist}(f,x)$ when $h = \omega(1)$. Observe that q never moves left in our algorithm. Therefore, there is a subtree T^* of height $\Theta(h)$ between f and x. We finish off by noting that T^* has $\Omega(a^h)$ keys, meaning $\mathrm{dist}(f,x) = \Omega(a^h)$. Therefore, h is $O(\log_a \mathrm{dist}(f,x))$ and the total running time is $O(b \log_a \mathrm{dist}(f,x))$. □

Relating to Unbounded Binary Search. Although our algorithm seems to be plagued with details of handling multiway nodes, it is in fact largely similar to how we finger searched sorted arrays in §1.2.3.1. To see this, let us interpret the path of q as follows. Observe that the size of T^* increases roughly geometrically as q ascends—this is akin to a doubling scan. Once q reaches the exit key, we continue the search by descending into the appropriate subtree(s)—this is akin to a binary search. The only difference between the two algorithms is that the series appearing here is only roughly

geometric. However, if we consider executing the algorithm from the leftmost leaf on the minimal tree displayed in the beginning of this section, then the series is in fact exactly geometric.

Leaf-Store vs. Node-Store. It is easy to see that the above algorithm will fail on a level-linked degree-balanced search tree in the *node-store* representation. The reason is not so much related to the algorithm itself; instead it is because the pointer structure of a node-tree no longer guarantees a path of length $O(\lg d)$ exists between two *items* that are d ranks apart. This is most evident by considering the items of the rightmost key of $root[T]$ and its successor in both representations. But in §3, we will show how a degree-balanced search tree in either representation can support a constant number of dynamic fingers but with no level-linking at all.

1.2.4 Three Example Applications

To show the benefit of data structures sporting the dynamic finger property, let us consider three of its simplest applications. Note that data structures with the dynamic finger property are in fact used in many applications, especially in computational geometry. Some references can be found in [BLMTT03].

1.2.4.1 Scanning

Suppose we use a search tree to represent the sorted list $A = \langle\!\langle a_1, a_2, \cdots, a_n \rangle\!\rangle$. Consider the scanning access sequence $\sigma = \langle\!\langle a_1, a_2, \cdots, a_n \rangle\!\rangle$, i.e., we access each element of A in the sorted order, unbeknownst to the search tree. If our search tree has the dynamic finger property, then we can easily upperbound the running time by $O(n)$. But if not, say we are using a classical red-black tree [GS78], then the best upperbound we can have is $O(n \lg n)$.

In-order Traversal vs. Scanning. To be sure, we do not expect any application to scan the keys of a search tree by accessing it once at each key, even though this is exactly what happened above. If the particular scanning access sequence is known in advance, and if the interface exposes the internals of a search tree, then an in-order traversal suffices to scan it.

However, notice that while the in-order traversal algorithm can exploit this particular access sequence, the access sequence is *not* known in advance. From the data structure's standpoint, it can only serve the access sequence one access at a time. This is the essence of being an online algorithm. The very fact that only some search trees can exploit an "easy" access sequence precisely shows that they have a desirable property absent

in the others. Splay trees [ST85b], for example can serve the scanning access sequence in linear time and in fact there has even been four proofs of this: [Tar85; Sun92; Col00; Elm04].

1.2.4.2 Merging Two Sorted Lists

Consider merging two *disjoint* sorted lists C and D of sizes $m \leq n$, each represented using a degree-balanced search tree of the same name. For simplicity, let us assume that these degree-balanced search trees are level-linked leaf-trees and their degree bounds are $O(1)$.

One correct algorithm would be to insert each element from C into D. This would yield an upperbound of $O(m \lg n)$ on the running time regardless of the order of the insertions. However, since we are using degree-balanced search trees with the dynamic finger property, a similar algorithm that orders the insertions in the symmetric order can be shown to run in $O(\sum_{i=1}^{m} \lg d_i)$ time, where d_1 is n and d_i for $i \geq 2$ is the difference in rank between the $(i-1)$-th and i-th elements of C in the *merged* list.

⟪ merging sorted lists C and D ⟫

1> A scan of C will yield its keys in the symmetric (sorted) order. The time incurred in this part of the algorithm is $O(m)$.

2> As each key of C becomes available, we insert it into D. We break down the insertion into two sub-steps

2.1> Locate the insertion point by searching for the key in D, thereby incurring $O(\lg d_i)$ time before the i-th insertion actually takes place.

2.2> Perform the actual insertion and the necessary restructuring. ▮

To analyze the running time, observe that the total amount of time incurred in the first sub-step is $O(\sum_{i=1}^{m} \lg d_i)$. And if the degree-balanced search tree is robust, then we can follow Huddleston and Mehlhorn [HM82, Theorem 6] and show that the *additional* time incurred in the second sub-step is $O(\lg n)$. This term is then absorbed by the above because $n = d_1$. But even not, we can still use a similar result by Brown and Tarjan [BT80, Theorem 1], which applies to level-linked $(2,3)$-trees and can be readily adapted to B-trees.

Our $O(\sum_{i=1}^{m} \lg d_i)$ upperbound is clearly no worse than $O(m \lg n)$. And thanks to the intuitive notion of a dynamic finger, our algorithm is also conceptually much simpler than an earlier AVL-tree [AVL62] merging algorithm by Brown and Tarjan [BT79]. Using the fact that $\sum_{i=1}^{m} d_i = O(m+n)$ and the concavity of logarithm, it can be shown that this bound is maximized when each d_i is $\frac{m+n}{m}$ and therefore our upperbound is $O(m \lg \frac{n}{m})$.

Note that this upperbound is particularly favorable when $m = \Theta(n)$, in which case it simplifies into $O(n)$.

Optimality and Similars. Finally, we also note that this bound is also information-theoretically optimal because there are $\binom{m+n}{m}$ possible interleavings among the elements of C and D and $\lg\binom{m+n}{m} = \Theta(m \lg \frac{n}{m})$. In fact, similar algorithms can be devised for other sorted list operations, which showcase the versatility of degree-balanced search trees sporting the dynamic finger property. In particular, union can be supported by discarding duplicates, intersection by retaining only duplicates, and difference by discarding the intersection. For more details on these and also the earliest appearances of the merging algorithm above, see [BT80] and also [HM82] and the references therein.

1.2.4.3 Inversion-Sensitive Sorting

Consider sorting a sequence of n unsorted keys $A = \langle\!\langle a_1, a_2, \cdots, a_n \rangle\!\rangle$. It is well-known that this requires $\Omega(n \lg n)$ comparisons in the comparison-based model. However, if the sequence is almost sorted, then we can circumvent this lowerbound by being sensitive to the presortedness of the sequence. In this example, we will use the number of inversions of a sequence as the measure.

An inversion happens when a larger key appears before a smaller key. More precisely, if $j < k$ but $a_j > a_k$, then the pair (a_j, a_k) induces an inversion in A. For any sequence of length n, the number of inversions can range from 0 to $\frac{n(n-1)}{2}$. The former happens when the sequence is sorted, whereas the latter happens also when the sequence is sorted but in the reverse order. The number of inversions of a sequence also happens to be the number of adjacent exchanges required to sort it, *à la* bubble sort.

Let Inv denote the number of inversions in A. Consider inserting the keys from A into an initially empty search tree sequentially. Essentially, this is an insertion sort and any balanced search tree can perform this in $O(n \lg n)$ time. However, if we are using a search tree with the static (or dynamic) finger property, then we can upperbound the running time by $O(n \lg \frac{Inv}{n})$ by placing a *static* finger at the rightmost leaf of the tree and use it for insertions. The analysis is as follows.

(1) Let s_i denote the number of keys after the insertion point of a_i till the end of the tree. Observe that besides being the static finger budget, s_i is

also the number of inversions due to a_i because a_i is smaller than s_i of the keys preceding it in A. Therefore,

$$\sum_{i=1}^{n} s_i = Inv. \tag{1.2}$$

(2) Using the static finger property and the amortization technique used in the merging example, we can again bound the total running time by $O(\sum_{i=1}^{n} \lg s_i)$.
(3) Using (1.2) and the concavity of logarithm, we can show by induction that $\sum_{i=1}^{n} \lg s_i$ is maximized when each s_i is $\frac{Inv}{n}$.
(4) We bound the maximum, $\sum_{i=1}^{n} \lg \frac{Inv}{n} = \lg \prod_{i=1}^{n} \frac{Inv}{n}$, as follows.

$$\left(\prod_{i=1}^{n} \frac{Inv}{n}\right)^{\frac{1}{n}} \leq \frac{\sum_{i=1}^{n} \frac{Inv}{n}}{n} = \frac{Inv}{n} \qquad \text{by A.M.-G.M. inequality}$$

$$\Rightarrow \quad \frac{1}{n} \lg \prod_{i=1}^{n} \frac{Inv}{n} \leq \lg \frac{Inv}{n} \qquad \text{by monotonicity of lg}$$

$$\Rightarrow \quad \lg \prod_{i=1}^{n} \frac{Inv}{n} \leq n \lg \frac{Inv}{n}$$

Note that our $O(n \lg \frac{Inv}{n})$ bound degenerates into $O(n \lg n)$ in the worst case, and no smaller bound can be expected since this algorithm is comparison-based. □

Adaptive Sorting. Before we finish, let us observe that the above algorithm uses a static finger and thus we have not fully exploited the dynamic finger property of the underlying search tree. This is most evident when we observe an inherent asymmetry in the notion of inversion. Few would agree that it is difficult to sort a reversely-sorted sequence, and yet there are $\Theta(n^2)$ inversions in it and we only have an $O(n \lg n)$ bound. Had a dynamic finger been used instead, we could have sorted this particular sequence in $O(n)$ time.

However, there are actually two good reasons why we would choose to present this algorithm—and this is besides the fact that it is simple to describe and analyze. First, it appeared in the very same paper [GMPR77] that introduced finger search, meaning it is one of the earliest applications of finger search. Second, it is also historically the first *adaptive* sorting algorithm and has a whole field of follow-up works. For example, the "local insertion sort" of Mannila [Man85] is precisely what we would have described if a dynamic finger was used instead!

Given its importance in both theory and practice, adaptive sorting is in fact a vast field. For more information about it, we refer the reader to a survey on the subject by Estivill-Castro and Wood [ECW92] as well as the following works as some starting points: [Man85], [PM95], [BCDI07], and [EF08].

1.2.5 A Data Compression Perspective

Let us end our introduction of the dynamic finger property with some intuition from a data compression perspective. Our goal is to relate the dynamic finger property with difference coding, and then show how other forms of compression give rise to other interesting properties. For simplicity, in this section we will consider a search tree containing a static set of keys $\{1, 2, \ldots, n\}$.

1.2.5.1 Gamma Codes

We will start by reviewing how a particular variant of γ codes [Eli75] encodes a natural number d into a bit string known as its codeword. Let the binary representation of d be $\mathrm{Bin}(d)$ and let the number of digits in $\mathrm{Bin}(d)$ be h. Notice that h is $\Theta(\lg d)$. The γ code of d consists of $2h$ bits: $(h-1)$ ones, followed by a zero, followed by $\mathrm{Bin}(d)$. For example, if d is 42, then $\mathrm{Bin}(d)$ is 101010 and h is 6. The codeword is therefore 111110101010.

A key property of γ codes is that they are prefix codes. This means that it is impossible to find $d_1 \neq d_2$ such that the codeword of d_1 is a prefix of the codeword of d_2. As an application, this means we can use γ code to encode a sequence of natural numbers into a uniquely-decodable bit string by simply catenating their γ codes.

1.2.5.2 Difference Coding

Consider the following method of compressing a length-m access sequence σ into a uniquely-decodable bit string S.

1> Record the γ code of σ_1.
2> For each subsequent σ_i, first record a sign bit indicating if σ_i is larger than σ_{i-1}, then record the γ code of their absolute difference.

The decompression procedure is straightforward, but what is the length of S? Let d_1 be σ_1 and d_i be $|\sigma_i - \sigma_{i-1}|$ for $i \geq 2$. By the property of γ codes, we know that σ_i can be encoded in $\Theta(\lg d_i)$ bits, and thus S has $\Theta(\sum_{i=1}^{m} \lg d_i)$ bits in total. Since S can be uniquely decoded back into σ, we have not lost any information in the compression and we can say that the information content of σ is upperbounded by $O(\sum_{i=1}^{m} \lg d_i)$ bits.

Of course, this also happens to be the total dynamic finger budget of σ, which upperbounds the time it takes to serve σ by any data structure with the dynamic finger property. To explain this apparent coincidence, let us offer the following interpretations.

(1) Recall that we assumed the set of keys is $\{1,2,\ldots,n\}$. This implies that the rank of σ_i is simply σ_i. When a data structure with dynamic finger property serves σ_i, the distance travelled by the dynamic finger is precisely $|\sigma_i - \sigma_{i-1}|$. Not surprisingly, this difference is what difference coding captures.
(2) The particular variant of γ code we have chosen corresponds to an encoding of the binary decisions in an unbounded binary search. In particular, the first h bits come from a doubling scan in which each 1 is "double" and the final 0 is "stop", and the last h bits come from a binary search, with 0 being "smaller" and 1 being "larger". (In fact, we could have define γ codes using the algorithm of unbounded binary search.)

Having seen the above, it should be clear that the actual compression is due to difference coding, and γ-coding does not actually reduce the amount of *redundancy* in σ at all. This leads us to the following question—what are the access sequences that have redundancy to be exploited?

1.2.5.3 Locality of References

Spatial. To answer the above question, let us start by re-examining the access sequences of the three applications in §1.2.4, all of which we know how to exploit.

(1) In the scanning access sequence, notice that each access is only one key apart from the previous. Serving this sequence hardly involves any searching at all.
(2) To merge two sorted lists of length $m \leq n$, we only need to perform m insertions into the longer lists from left to right. Furthermore, the insertion points are on average at most $\frac{m+n}{n}$ keys apart.
(3) When we use insertion sort to sort an n-sequence with Inv inversions, on average each insertion point is at most $\frac{Inv}{n}$ away from the largest number seen so far. If Inv happens to be $o(n^2)$, then the average search space of each insertion will be $o(n)$.

The common trait shared by these access sequences is their *spatial* locality of references.[6] Indeed, for each of these access sequences, we notice that the distance between two consecutive accesses is more restricted than it is possible in the general case. The presence of spatial locality in these access sequences is why a data structure with the dynamic finger property can have an asymptotically improved running time. This observation also brings us to the one issue that no introduction to the dynamic finger property can end without.

Temporal. As it turns out, difference coding may not be always effective—much less *optimal*. Here is another type of locality which difference coding fails to capture at all.

Consider the access sequence $\sigma_{ws} = \langle\!\langle 1, n, 1, n, 1, n, \cdots \rangle\!\rangle$. Difference coding saves very little here because consecutive accesses are $(n-1)$ apart in the key space. On the other hand, σ_{ws} actually exhibits a *temporal* locality of references, which is what any data structure with the working set property can exploit. The data compression perspective also suggests that there *must* be a method in which σ_{ws} can be succinctly compressed:

1> At the first occurrence of any key x, record 0 (written as + below) followed by the γ code of x in full.

2> At the next occurrence of the same key x, say as σ_i, record 1 (written as − below) and the γ code of one plus the number of *unique* keys that has occurred since the last occurrence of this key. This number, commonly denoted $t_i(x)$, is the working set number of x at the time of σ_i and ranges from 1 to n.

Applying this on σ_{ws}, the resulting sequence before γ-coding would be $\langle\!\langle +1, +n, -2, -2, -2, -2 \rangle\!\rangle$ followed by any number of repeated -2's and is thus highly compressible. And as an example data structure that has the working set property, we note that splay trees [ST85b] can serve any σ of length m in $O(n \lg n + \sum_{i=1}^{m} \lg \max(2, t_i(\sigma_i)))$ time. (It is an interesting open question whether the $O(n \lg n)$ term can be reduced to $O(n)$.)

Other Types of Locality. Having read the above discussion, we note that many interesting questions can be asked. For example, does some other type of locality exist? And is there an optimal method to compress any access sequence? And if so, do we have any data structure that can match such a compression scheme? However, as important as these questions are, they are beyond our scope and it's time for us to turn to our next topic.

6. The term "locality of references" actually originates in the work on computer systems in the late 1950s. Denning [Den05] has an interesting account of it.

1.3 Heterogeneous Finger Search Trees

Highly related to this thesis is the class of heterogeneous finger search trees introduced by Tarjan and Van Wyk [TVW88]. A heterogeneous finger search tree is a worst-case balanced search tree with its two spines inverted. On the right spine, this means instead of having each node point to its rightmost child, we have each node point to its own parent. And instead of keeping a pointer to the root, we keep two pointers to the bottommost internal node on the two spines from which the algorithms on these trees start. These two internal nodes—as opposed to the leftmost and the rightmost external nodes—will be referred to as the two "ends" of the tree. Note that the number of pointers in each node on the spine actually remains the *same*.

As observed by Tarjan and Van Wyk, in principle the technique of spine inversion can be applied to any balanced search tree that is robust. Indeed, this approach has also been applied on AVL-trees by Tsakalidis [Tsa85], who essentially inverted the left spine of an AVL-tree and "robustify" it by allowing an extra height imbalance on the spine (see §1.5.3 for a better comparison). Moreover, due to the performance guarantees to be described below, we can also choose to work with either leaf-trees or node-trees depending on the requirements of an application. This is why even though Tarjan and Van Wyk based their work on leaf-store red-black trees, we will use node-store red-black trees in this thesis instead.

Before we go on, observe that in our description above, the root can no longer reach its leftmost and the rightmost children due to spine inversion. But in some applications that require augmentation, it may be necessary to allow the root to reach these two children by maintaining two extra pointers. An example of this can be found in the thesis of Booth [Boo90], and we note that for our purpose we do not need these two pointers.

1.3.1 Representation and Operations

To give a flavor of how a heterogeneous red-black tree works, let us review the algorithms for search and join. Note that our description below assumes familiarity with red-black trees as described in, say, [CLRS01, §13]. The reader is simply reminded that the external nodes are considered to be black and their black-height is defined to be 0.

1.3.1.1 Representation

The representation of a heterogeneous red-black tree is basically the same as a red-black tree but with the nodes on the two spines pointing upward.

To implement the inverted right spine, we will simply define $right[u]$ to be the parent of u for any internal node u on the spine. The inverted left spine is defined symmetrically. And to aid our presentation below, we will let $p[u]$ to denote the correct child pointer which we use to point to the parent. Notice that the above definition is stated for *node-trees*. For a leaf-tree, we would have to maintain two extra pointers to the leftmost and the rightmost external nodes since their parents no longer have pointers to them. (Or we may let these two external node have parent pointers as in [TVW88] and define them to be the two ends of the tree.)

1.3.1.2 Search

To search for a key x in a heterogeneous red-black tree T, we start two searches simultaneously from the two ends towards the root and stop once either search finds x. By symmetry, we will describe only the search starting from the left end of T.

The idea is to scan the left spine bottom-up until we hit the rightmost node u whose key is still less than or equal to x. This key is often called the turn key. In other words, we have $key[u] \leq x$ and, if $p[u]$ exists, $x < key[p[u]]$ also. At this point, one of the following three possibilities can happen.

(1) If $key[u] = x$, then we are done.
(2) If $key[u] \neq x$ but $u = root[T]$, then x is not in the left subtree of T and we can terminate the search on this side.
(3) If both of the above do not apply, then x can only be in the right subtree of u because our stopping condition implies $x < key[p[u]]$. We descend (turn) right from u and continue searching downward as in a normal red-black tree.

Running Time. We claim that if x is at rank d, then the above algorithm runs in $\Theta(\lg d)$ time. The reasoning here—not surprisingly—is highly similar to that of an unbounded binary search. Suppose the black-height of u is h, i.e., u is the h-th black node we meet on the left spine. Observe that the above algorithm runs in $\Theta(h)$ time, and as long as u is not the left end, the right subtree of the left child of u separates the left end of T from x. Since we are dealing with a red-black tree, this subtree has $\Theta(2^h)$ keys and gives us the desired lowerbound on d. By combining a similar argument on the right hand side, this gives us the running time bound $\Theta(\lg \min(d, n-d))$ with n being the number of keys in T.

1.3.1.3 Join

Given two heterogeneous red-black trees T_L and T_R and a key x in-between, we can join them together using the following algorithm.

⟪ join (T_L, x, T_R) ⟫

1> Scan the black nodes on the right spine of T_L and the left spine of T_R in lock-step and terminate once we hit the root of either tree. By symmetry, assume we hit the root of T_L first and let v be the last black node we visited on the left spine of T_R. Notice that T_L and v share the same black height, which will be denoted h_L.

2> Determine if the black height of T_R is also h_L by checking if v has a black parent or a black grandparent.

3> If the black height of T_R is h_L, then finish with following procedure.

- form (T_L, x, T_R) -

3.1> Make x the new root by pointing both roots of T_L and T_R towards x.

3.2> Color x red if both roots are black, or else color it black.

3.3> Un-invert the right spine of T_L and the left spine of T_R.

4> Otherwise, the black height of T_R is larger than h_L and we will join T_L to the left spine of T_R as follows.

- replace $v^* \in T_R$ by (T_L, x, v^*) and restructure -

4.1> If the root of T_L is black, let v^* simply be v. Otherwise, color the root of T_L black and let v^* be the lowest black ancestor of u. Notice that in either cases, v^* is black and has the same black height as T_L.

4.2> Put x in a new red node u with v^* as its right child, and let the root of T_L points to u from the left, and let u points to $p[v^*]$.

4.3> Un-invert the right spine of T_L and the left spine of T_R up to and including v^*.

4.4> Start restructuring at x if it has a red parent. ∎

Running Time. First, we claim that the restructuring in step 4.4 takes amortized $O(1)$ time. This can be proved by defining the potential function of the tree as (i) the black height of the tree, plus (ii) the number of black nodes with *no* black children. The time spent in the steps of the algorithm is proportional to h_L, but this can be entirely charged to the potential released by T_L. By also considering the similar case where T_R has a smaller black height, the total running time is therefore amortized $O(1)$.

1.3.1.4 Other Operations

Heterogeneous red-black trees actually support all three dictionary operations as well as splits and joins. However, instead of reviewing all of them in detail, here we will simply give the specifications and the running

Potential	black height of T + #black nodes w. no black children (used in this thesis)	#black nodes w. two black children + 2 #black nodes w. two red children
Insert	$O(1)$	$O(1)$
Delete	$O(\lg \min(d, n-d))$	$O(1)$
Join/Catenate	$O(1)$	$O(\lg \min(\|T_L\|, \|T_R\|))$
Split	$O(\lg \min(\|T_L\|, \|T_R\|))$	$O(\lg \min(\|T_L\|, \|T_R\|))$

Table 1.1 – Running Time Bounds of Heterogeneous Red-Black Trees

time bounds the operations that we did not review. Interested readers are referred to [TVW88, Appendix A] and [Boo90, §2] for the details.

We note that at least two different potential functions can be used to analyze heterogeneous red-black trees and both of them have appeared in [TVW88]. For this thesis, we will stick with the one we used above in the analysis of joins. But for completeness we will also state the bounds for the other more classical potential function when we summarize the running time bounds in Table 1.1.

Insert and Delete. Given a heterogeneous red-black tree of size n and assuming the location of the update is known, the amortized time to insert a key is $O(1)$, and the amortized time to delete the key at rank d is $O(\lg \min(d, n-d))$. Note that the former will degenerate into the latter if we have to search for the update location first.

Catenate. Given two heterogeneous red-black trees T_L and T_R, the amortized time to catenate them is $O(1)$. The resulting tree contains all keys of T_L and all keys of T_R, in order. This uses the fact that we can delete the rightmost key of T_L in amortized $O(1)$ time and use it in a join.

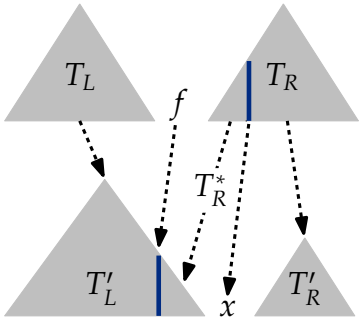
Split. Given a heterogeneous red-black tree T of size n and a key x, the amortized time to split it at x is $O(\lg \min(|T_L|, |T_R|))$. The result is a triple $(T_L, x^?, T_R)$, with T_L and T_R being heterogeneous red-black trees containing the keys of T that are respectively smaller than and greater than x. The option $x^?$ will be x if it is in T; or NIL if otherwise.

1.3.2 Dynamic Finger Property

While the above bounds implies that a heterogeneous red-black tree only has two *static* fingers at the two ends, it can actually implement the semantics of a dynamic finger using splits and joins. In other words, a heterogeneous red-black tree can also be considered to have the dynamic finger property when used in the following way, which we call the triplet representation. (We acknowledge that this is a slight abuse of the word "triplet"; however, we deem it is better than saying a doublet is a triple.)

Initialization. Suppose we are given a red-black tree T containing n keys and we would like to initialize a finger at a key f in T. First, split T at f using the red-black tree split algorithm to obtain the triple (T_L, f, T_R). Then, convert T_L and T_R into heterogeneous red-black trees by inverting their spines. Notice that these two steps run in $O(\lg n)$ time.

Finger Search. From now on, we will maintain this triple as our data structure. Suppose the finger needs to be moved to another key $x \neq f$. By symmetry let us assume $f < x$. First, split T_R at x to create (T_R^*, x, T_R'). Then, join (T_L, f, T_R^*) to create the new T_L'. The amortized time of these two steps are both $O(\lg |T_R^*|)$. Since T_R^* contains exactly all the keys that sits between f and x in T, this becomes $O(\lg \operatorname{dist}_T(f, x))$ as we desired.



Remark 1.7. Although we will not get into details, it is not hard to see how the above construction can be generalized to handle a *constant number* of dynamic fingers. In general, k fingers can be supported by maintaining a tuple of $(2k+1)$ values. The odd positions are the heterogeneous red-black trees and the even positions are the keys at the dynamic fingers.

1.3.3 Binarization

In the format presented above, searching a heterogeneous red-black tree involves starting from the two ends of the tree. Because of this, it can be argued that these trees are not search trees in the strictest sense of the phrase. Fortunately, a construction that appeared in a recent paper by Demaine, Harmon, Iacono, Kane, and Pătraşcu [DHIKP09] actually shows how to simulate a heterogeneous red-black tree using a binary search tree algorithm. This puts heterogeneous red-black trees—or rather the applications of them—firmly in the domain of binary search trees. Their method is very easy to understand and we will present it here for completeness.

Construction. Let T be the heterogeneous red-black tree to be simulated and let T^* be its binary search tree simulation. The following algorithm constructs T^* from T.

⟪ T (hrb) → T^4 (h24) → T_Z^4 (z24) → T^* (bst) ⟫

1> Color the root of T black if it is red.

- un-binarization -

2> Convert T into a heterogeneous $(2,4)$-tree T^4 by collapsing the red nodes into their black ancestors. Let the height of T^4 be h and notice

that the length of the two spines of T^4 is h exactly. Also, let l_i and r_i be, respectively, the left and right spinal nodes of T^4 at height i.

- pull the two spines and interweave -

3> Convert T^4 into 4-way search tree T^4_Z as follows.

3.1> Form the zig-zag skeleton of T^4_Z by interweaving the two spines *bottom-up* while preserving the symmetric order. Specifically, start with l_1 as the root. Then, for $1 \le i < h$, l_i will have r_i as its rightmost child, and in turn r_i will have l_{i+1} as its leftmost. Notice that the formation of the skeleton displaces the rightmost/leftmost child of l_i/r_i for $1 < i < h$.

3.2> For each l_i where $1 < i < h$, reconnect its rightmost child as the leftmost child of l_{i+1} in T^4_Z. Notice that the leftmost child of l_{i+1} in T^4 was l_i and thus this child position was left empty in T^4_Z at the skeletal formation.

3.3> Similarly for $1 < i < h$, we reconnect the leftmost child of r_i as the rightmost child of r_{i+1}.

- binarization -

4> Convert T^4_Z back into a red-black tree T^* by expanding each node with degree larger than 2 into a black parent and its corresponding red children. Notice that T^* is *not* a red-black tree in the sense of a balanced search tree. The red-black encoding is simply used to encode 3- and 4-nodes in the binarization.

Simulation. To see that T^* can simulate T, first observe the followings.

(1) A spinal node at height h in T has a depth of either $(2h-1)$ or $2h$ in T^* and appears on the zig-zag skeleton.

(2) Each subtree that is hanging off a left spinal node u of T is now hanging off a skeletal node of T^*.
   (a) If the subtree is not rooted at the rightmost child of u, then it is rooted at the same child position of u in T^*.
   (b) Otherwise, the subtree is rooted at the leftmost child position of the leftmost child of the rightmost child of u in T^*.

   The cases on the right spine is symmetric.

(3) The parent-child relationship within each subtree hanging off the spines of T remain the same in T^*.

Since all operations on T starts from the two ends, we can simulate the upward traversal of the spines by descending through the zig-zag skeleton from the root of T^*. Once the turn key has been identified on the skeleton, we can descend into the correct subtree depending on the condition in observation (2). From that point on, the subtree we are operating on is exactly the same in T and T^*. It follows that T^* can simulate T with only constant slow-down.

1.3.4 Homogenity and Finger Search Trees

To enhance our understanding of heterogeneous finger search trees, let us understand why they are called "heterogeneous" in the first place. Recall that in §1.2.2, we made a distinction between finger search trees and dynamic finger trees. A finger search tree can be finger searched from any key in the tree, whereas a dynamic finger tree can finger searched only from a dynamic finger. Out of the homogeneity among the locations from which a finger search can start, Tarjan and Van Wyk [TVW88] called the former a homogeneous finger search tree. This is to draw a better contrast with a heterogeneous finger search tree, on which we always start searching from its two ends. We remark that since we have already reserved the term "finger search trees" for the homogeneous type, we will not be using this adjective very often except when drawing parallels.

Although a homogeneous finger search tree is simply a finger search tree, the exact relationship between a dynamic finger tree and a heterogeneous finger search tree is more complicated. First of all, a heterogeneous finger search tree is *not* a dynamic finger tree since technically it only has two static fingers on the two ends. But from §1.3.2 we know a heterogeneous finger search tree that supports splits and joins can be used to implement the semantics of a dynamic finger due to the triplet representation. Furthermore, by suitably generalizing the construction in §1.3.3, we know any heterogeneous finger search tree can in fact be simulated by a binary search tree. Putting these two facts together, we can actually obtain a dynamic finger tree from an underlying balanced search tree T as follows.

⟪ transforms balanced search tree T into a dynamic finger tree T' ⟫

1> Convert T into its triplet representation (T_L, f, T_R) in which T_L and T_R are both heterogeneous finger search trees.
2> Simulate T_L and T_R as the binary search trees T_L^* and T_R^*.
3> Maintain a binary search tree T' with its root containing f and let T_L^* and T_R^* be respectively its left and right subtrees. ▮

With the above transformation, T' is in fact a faithful representation of T as long as each insertion or deletion happens at the dynamic finger. Furthermore, if k dynamic fingers have to be supported, we can also organize them as a balanced binary search tree of k keys and hang the $(k+1)$ binary search tree simulations at the corresponding external positions.

Relative Strength. Without doubt, homogeneous finger search trees are more powerful than heterogeneous finger search trees when the ability to

search by key is concerned. However, Tarjan and Van Wyk [TVW88] have discovered how the latter can be augmented to support other useful operations such as search by rank or search by a secondary heap order. While technically a homogeneous finger search tree can support these operations if it also supports splits and joins, this is because it can then be used to simulate a heterogeneous finger search tree in the straightforward manner. Tipping the balance, some homogeneous finger search trees support an operation called excisions and there is no efficient analogue in heterogeneous finger search trees.

Excision. Also known as a three-way split, an excision is the splitting of an inner portion of a finger search tree T of size n specified by two of its keys $x \leq y$. Suppose the number of keys in T within the closed interval $[x, y]$ is d. After the excision, these keys will be in a new tree of size d and the other $(n - d)$ keys will remain in T.

Hoffman, Mehlhorn, Rosenstiehl, and Tarjan [HMRT86] have demonstrated how excisions can be used in an optimal linear time algorithm for the so-called "Jordan sorting" problem[7]. Their algorithm is based on *circularly* level-linked $(2,4)$-trees. Using robust balancing, the running time of an excision is amortized $O(\lg \min(d, n - d))$. Hoffman et al. has also noted that their data structure can be used to speed up an $O(V \lg V)$-time planarity algorithm of Hopcroft and Tarjan [HT71] to run in optimal $O(V)$ time. This matches an optimal planarity algorithm that was also due to Hopcroft and Tarjan [HT74] and was based on a complicated application of depth-first search (see also the thesis of Tarjan [Tar71]).

Even though the Jordan sorting algorithm of Hoffman et al. [HMRT86] may have popularized the excision operation, we must also note that excision is not exactly required by the Jordan sorting problem. For example, Aurenhammer [Aur87] have proposed another linear time algorithm that is based on computing the convex hull of a carefully constructed polygon. Furthermore, Fung, Nicholl, Tarjan, and Van Wyk [FNTVW90] also have a simpler (but not simple) linear time algorithm that is based on none other than the heterogeneous finger search trees of Tarjan and Van Wyk [TVW88].

7. In a Jordan sorting problem, we would like to sort a permutation of size n that corresponds to the ordering of the intersections of a Jordan curve and a straight line as they appear on curve. Using the restriction that the curve cannot self-intersect, it can be shown that there are only c^n different possible inputs for some constant c. This implies that the Jordan sorting problem is strictly easier than the general sorting problem since the latter can have $n!$ different possible inputs.

1.4 Tangolike Trees

The final topic we will introduce in this chapter is the class of tangolike trees. Search trees in this class all follow a schema that was first used in the tango trees of Demaine, Harmon, Iacono, and Pătraşcu [DHIP04]. Since this schema is best-understood via a lowerbound in the binary search tree model, let us start with a review of the model itself.

1.4.1 The Binary Search Tree Model

The term "binary search tree model" refers to a concept that was first alluded to in the seminal paper on splay trees by Sleator and Tarjan [ST83]. Many readers of this thesis will be familiar with their dynamic optimality conjecture in [ST85b], which states that for any access sequence σ, the running time of a splay tree of size n is within $O(n)$ plus a constant multiple of the running time required by *any* search tree. Put in the lingo of competitive analysis also by the same authors [ST85a], this conjecture simply means splay trees are $O(1)$-competitive against an offline optimal binary search tree.

Implicit in the above restatement of the conjecture is a proper definition of search trees. To ensure the comparison of running times is meaningful, Sleator and Tarjan [ST85b] spelled out several requirements that any search tree used in the comparison has to follow. In our usage, these requirements are said to form a binary search tree model, and the restructuring algorithm used in any search tree that fits these requirements is said to be a binary search tree algorithm. Since the restructuring algorithms in many search trees are not specifically named, we will also allow ourselves to say a search tree is a binary search tree algorithm if it uses a binary search tree algorithm to restructure itself.

Historically, several different binary search tree models have been put forward by researchers in the field. Besides the unnamed model of Sleator and Tarjan, we also have the "standard search algorithm" model of Wilber [Wil86] as well as another unnamed model of Lucas [Luc88b, p. 3]. Although their details differ, these three models are equivalent up to constant factors in the sense that an algorithm running in one model can be converted to run in another with only a constant slowdown. Since the choice of model does not affect asymptotic analysis, for our purpose we will simply say a restructuring algorithm is in *the* binary search tree model as long as it is in one of these models.

The Reorganization Tree Cost Model. For concreteness, we will restate the model of Lucas here. Following the suggestion of Harmon [Har06, Appendix A], this model is also called the reorganization tree cost model.

(1) A search tree is a node-store binary search tree T with $\{1,2,\ldots,n\}$ as its fixed set of keys. It can have any initial shape, and in particular this shape is *not* chosen by the restructuring algorithm.
(2) A internal node u of T can only be addressed via a pointer to u, but the actual address of u cannot be obtained using any method.
(3) Each node can have any amount of extra storage for augmentation but *no* extra pointers besides the existing two. Together with the restriction in (2), this means the extra storage cannot be used to address nodes.
(4) An access sequence σ is $\{1,2,\ldots,n\}^m$ for any natural number m. Notice this implies all accesses are successful.
(5) An access σ_i starts with a finger visiting the root of T.
(6) The finger is allowed to traverse on the pointers of T in *either* direction to visit any set of internal nodes, but it must use the pointers of T to get there. In other words, it cannot go from one node to another when there is no pointer between them in either direction.
(7) During the tour, any node currently under the finger can be rotated with its parent, if present.
(8) To satisfy the access, the finger must visit the internal node containing σ_i at least once during the tour.
(9) The cost of serving σ_i, denoted $T(\sigma_i)$, is the number of *unique* nodes visited by the finger. Note that this is independent of the number of rotations performed.
(A) Finally, the cost of serving σ is defined to be $\sum_{i=1}^{m} T(\sigma_i)$. Note that we do not impose any restriction on the length of σ.

Before we go on, let us present a few remarks concerning this model.

☞ Since the finger is required to traverse *along* the pointers of T and it has to visit the node of σ_i at least once, we observe that the nodes visited by the finger form a connected subtree τ_i that contains the access path of σ_i. This subtree is known as the reorganization tree of σ_i, in reference to the fact that it can be reorganized by rotations during the tour.
☞ Binary search tree algorithms in this model can be specified at a high level that does *not* involve specifying the actual rotations. All we have to specify is how to compute the node membership and the final shape of τ_i from any given σ_i. The actual rotations can then be obtained using the fact that any two search trees of size τ_i can be rotated into each other in $O(\tau_i|)$ rotations [CW82; STT86]. In other words, we will think

of decomposing T into $(|\tau_i|+2)$ parts, namely τ_i itself and the $(|\tau_i|+1)$ subtrees at the external positions of τ_i. After σ_i is served, we simply transform τ_i from its current shape to the new shape via rotations. Notice that this keeps the shapes of $(|\tau_i|+1)$ subtrees intact.

Example Binary Search Tree Algorithms. One example binary search tree algorithm that fits this model is the venerable splaying algorithm used by Sleator and Tarjan [ST85b]. The reorganization tree is the access path itself and its final shape can be described by a rotate-to-root procedure followed by a path compression in a certain form.

Another example is the binarization of heterogeneous red-black trees we discussed in §1.3.3. Note that heterogeneous red-black trees in their original form are not in the binary search tree model. In fact, Demaine et al. [DHIKP09] devised this binarization precisely because they want to use the capabilities of heterogeneous red-black trees inside their new "GreedyOSS" binary search tree algorithm.

1.4.2 The Interleave Bound

Having defined the binary search tree model, let us turn to a lowerbound that forms the theoretical underpinnings of all tangolike trees. We note that this bound first appeared in the work of Demaine, Harmon, Iacono, and Pătraşcu [DHIP04] and is a slight variation of Wilber's first bound in [Wil86].

The Interleave Bound. The interleave bound of an access sequence σ is a function of σ and a reference tree P. The latter is a node-store binary search tree with n internal nodes fixed at any shape of *our choice*. Each internal node of P has a preferred direction of either left or right at all time, and initially every internal node prefers left. Even though P usually only exists in our imagination, let us also use $root[P]$ to denote its root.

We say that a node u prefers left (resp. right) if the most recent access within $P|^u$ is in the left (resp. right) subtree of u. We leave the preference of u flexible in the case when the most recent access within $P|^u$ is the key in u itself. The two common choices are to force u to prefer left after it has been accessed, or to simply force u to change its preference whatever it was.

To build up our intuition on node preferences, let us suppose that P has just served σ_i. Let $u_0 = root[P], \ldots, u_d = \sigma_i$ denote the node(s) on the access path of σ_i in P. Observe that for $1 \le j \le d$, the node u_{j-1} prefers towards the direction of u_j, and the target node u_d may prefer left or right depending on the definition used. Any node with a *new* preferred direction after P has served σ_i is said to have "switched due to σ_i". Notice that if a node

does not appear on the access path of σ_i, then its preference cannot have switched due to σ_i.

Let m be the length of σ and let $\mathrm{IB}(\sigma_i, P)$ be the number of switches induced on P by σ_i. The interleave bound $\mathrm{IB}(\sigma, P)$ is simply $\sum_{i=1}^{m} \mathrm{IB}(\sigma_i, P)$. We claim that the cost of any binary search tree algorithm serving σ is at least $\frac{1}{2}\mathrm{IB}(\sigma, P)$, and a proof of this can be found in [DHIP04, Appendix A]. Notice that the claim is meant to hold only after the serving costs are *aggregated* over the entire σ. In other words, the number of switches induced by one particular access is not a lowerbound of any sort.

We also note that this bound can be reduced by $O(n) + O(m)$ due to, respectively, our arbitrary choice for the initial node preferences and the flexible preference of the access node itself. However, the cost of serving σ in our model definition in page 43 can fully absorb this reduction when $m = \Omega(n)$, which is precisely what we required in our competitiveness definition on page 19.

Bit-Reversal Sequence. Before we go on, let us follow Wilber [Wil89] and show a simple application of the interleave bound. Let A be the sequence $\langle\!\langle 0, 1, 2, \ldots, 2^h - 1 \rangle\!\rangle$ for any h. Let a_i be the i-th element of A and let b_i be the sequence of binary digits that represents a_i, with zeros padded on the left to ensure that b_i is a sequence of exactly h binary digits. Let d_i be the reversal of b_i and let e_i be the integer whose binary representation is d_i. Finally, let σ_i be $(2e_i + 1)$. We call the sequence $\langle\!\langle \sigma_i \rangle\!\rangle$ the bit-reversal sequence of h bits, denoted $\mathrm{BR}(h)$. As an example, Table 1.2 illustrates $\mathrm{BR}(4)$, which is contained in its last column.

A remarkable fact about the bit-reversal sequence is that it is one of the "hardest" access sequences for a binary search tree. Consider a reference tree in the form of a complete binary search tree of height $(h + 1)$ and consider how we descend through the reference tree when it serves σ_i. If the first digit of d_i is a 0, we descend left; otherwise, we descend right. After h descents, we will reach the leaf containing σ_i. Observe that at every descent, the preference of the current key changes. This is evident by inspecting the digits of b_i as a binary counter, with the least significant bit corresponding to the root of the reference tree. We conclude that the number of switches per access is h. This gives us the

a_i	b_i	d_i	e_i	σ_i
0	0,0,0,0	0,0,0,0	0	1
1	0,0,0,1	1,0,0,0	8	17
2	0,0,1,0	0,1,0,0	4	9
3	0,0,1,1	1,1,0,0	12	25
4	0,1,0,0	0,0,1,0	2	5
5	0,1,0,1	1,0,1,0	10	21
6	0,1,1,0	0,1,1,0	6	13
7	0,1,1,1	1,1,1,0	14	29
8	1,0,0,0	0,0,0,1	1	3
9	1,0,0,1	1,0,0,1	9	19
10	1,0,1,0	0,1,0,1	5	11
11	1,0,1,1	1,1,0,1	13	27
12	1,1,0,0	0,0,1,1	3	7
13	1,1,0,1	1,0,1,1	11	23
14	1,1,1,0	0,1,1,1	7	15
15	1,1,1,1	1,1,1,1	15	31

Table 1.2 – Bit-Reversal Sequence of 4 Bits

desired lowerbound of $\Omega(n \lg n)$ for *any* binary search tree of size n to serve $\mathrm{BR}(h)$, where n is the size of the reference tree $(2^{h+1} - 1)$.

A Little Bit of History. As we mentioned, the interleave bound is due to Demaine, Harmon, Iacono, and Pătraşcu [DHIP04] and in fact appeared in the very same paper that defined tango trees. Just as the authors themselves have pointed out, this bound is a slight variation of a lowerbound first proved by Wilber in [Wil86]. In fact, the two bounds differ precisely in their model of search trees—Wilber's bound is based on leaf-trees and the interleave bound adapts it to node-trees. Incidentally, Wilber actually proved *two* lowerbounds in [Wil86] and the interleave bound corresponds to the first of the two. Wilber conjectured that his second bound is stronger than his first one. Although there are intuitions why this may be true, so far it remains an open conjecture.

For many years, the two lowerbounds of Wilber were the only candidates when lowerbounds in the binary search tree model are concerned. However, subsequent to the interleave bound, two highly-similar lowerbound frameworks were independently proposed by Harmon [Har06] (see also [DHIKP09]) and Derryberry, Sleator, and Wang [DSW05]. Since these two frameworks encompass all known lowerbounds in the binary search tree model as of this writing, an open problem in the field is whether there can be any lowerbound outside, i.e., higher than the ones provable using these frameworks.

1.4.3 Terminologies

Having introduced the notion of preference to define the interleave bound, let us define a number of related terms before we go on. For a given reference tree P, let u and v be any two of its internal nodes and let x be the key in u. Also, let σ_i be the most recent access served by P.

(1) The reference depth of u or x is the depth of u in the reference tree P. We note that in many cases P is usually specified using the context. Furthermore, although we do not support updating P dynamically, in such a setting the reference depth of u or x can change over time.
(2) If v is the child at the preferred direction of its parent u, then v is the preferred child of u. This child is also denoted $\checkmark_i(u)$.
(3) The preferred subtree of u is $P|^{\checkmark_i(u)}$.
(4) Oppositely defined are, respectively, the nonpreferred child and the nonpreferred subtree of u.

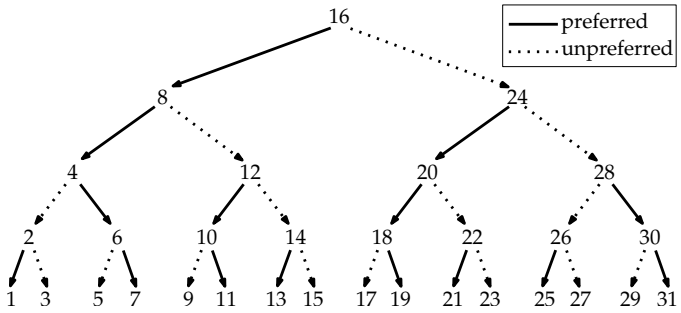


Figure 1.2 – Example reference tree of 31 nodes (see Example 1.1)

(5) For convenience in structural inductions, $root[P]$ is defined to be a *nonpreferred* child.

(6) Due to the way they are usually drawn in figures, the child pointer from u to $\checkmark_i(u)$ is said to be solid and the other child pointer of u is said to be dashed.

(7) A solid path is a maximal path of *internal* nodes in P that comprises only solid pointers. The solid path of u or x is the solid path that contains u and will be denoted $\mathrm{SP}(u)$ or $\mathrm{SP}(x)$.

(8) For any subtree P' of P, the root solid path of P' is the solid path of the root of P'. Note that the root solid path of $P|^u$ is simply $\mathrm{SP}(u)$.

(9) The preferred leaf of u or x or $\mathrm{SP}(u)$ is the last and hence deepest node of $\mathrm{SP}(u)$. Notice that given our assumption that u is internal, this is always a leaf according to our definition of a leaf[8] on page 4.

Example 1.1. *Switches on a Reference Tree.*

Figure 1.2 shows a reference tree P of 31 internal nodes in the form of a complete binary search tree, with the external nodes omitted for clarity. In its current state, the preferred leaf of 8 is 7, and if the next access is 17, then a left-to-right switch at 16 and a right-to-left switch at 18 will be made in P. Note that P could have taken on *any* shape, but in §1.4.4 we will explain why a complete binary search tree is a common choice.

1.4.4 Interleave Bound and Tangolike Trees

With the reference tree framework set up, tangolike trees are in fact quite easy to understand. In particular, a tangolike tree T is a simulation of a reference tree P but with a different cost model. Suppose we are serving

8. For an interesting example, consider a reference tree in the form of a red-black tree of size 2 with a black root and a red nonpreferred child on the right. The preferred leaf of the root is the root itself, but this root may not be designated as a leaf in some definitions.

an access σ_i using P and T. Since P is a node-tree, the cost incurred by P is simply one plus the reference depth of σ_i. The cost incurred by T, however, is one plus the number of switches induced by σ_i on P. Notice that this can be significantly lower than the cost incurred by P, but it is also upperbounded by the latter because at most one switch occurs at each depth. For flexibility, we allow T to be either worst-case or amortized. Summing over the entire σ, the cost incurred by T is thus proportional to the total number of switches induced on P by σ. This is in turn proportional to the interleave bound $\mathrm{IB}(\sigma, P)$.

It would be truly wonderful if the simulation would introduce only a constant slowdown for this would imply that T is $O(1)$-competitive. Unfortunately, this is not true and in the worst case the slowdown can be logarithmic in the height of P. Yet this amount of slowdown is still sufficient to guarantee that T is $O(\lg \lg n)$-competitive, for we merely need to pick any P that has $O(\lg n)$ height. This explains why P is commonly chosen to be a complete binary search tree—just as we did in Example 1.1—even though in theory P can be any balanced binary search tree and the competitiveness result will hold. Having said this, the reader is reminded that P can be *any* binary search tree in our discussion to follow. Assuming it is balanced or even complete would be unnecessarily restrictive.

1.4.5 Structure of Tangolike Trees

To define the structure of a tangolike tree, let us fix a moment in time and suppose our reference tree P has just served σ_i for some i. Notice that this fixes the preferences of all nodes in P.

Two Forests. Overall, a tangolike tree T is an interconnected forest of binary search trees which simulates the solid path decomposition of P. Let r denote $root[P]$. The solid path decomposition of P is the union of (i) the singleton set $\{\mathrm{SP}(r)\}$ and (ii) the solid path decomposition of the nonpreferred subtrees hanging off $\mathrm{SP}(r)$.

Observe that any path in a binary search tree *is also* a binary search tree in itself. By applying this view to the solid paths, P is a forest of binary search trees interconnected by dashed pointers into a binary search tree. Each solid path in P is then implemented by another binary search tree in the forest of T. Following Demaine et al. [DHIP04], each of these binary search trees is called an auxiliary tree. Note that an auxiliary tree contains exactly the same set of keys as its corresponding solid path, but in general they do not have the same shape.

Before we go on, let us introduce an extra notation here. Suppose u is a node in T and contains x as its key. The auxiliary tree of u or x will be denoted $\mathrm{ST}(u)$ or $\mathrm{ST}(x)$.

Interconnections. To define the interconnections between the auxiliary trees of T, let us take a closer look at the solid paths hanging off $\mathrm{SP}(r)$.

Let k denote the size of $\mathrm{SP}(r)$. Observe that $\mathrm{SP}(r)$ has k nonpreferred subtrees and one preferred external node. As P is a node-tree, the latter is actually $\perp$. By decomposing P into its solid paths, observe that the $(k+1)$ external positions of $\mathrm{SP}(r)$ will be occupied by the root solid paths of these $(k+1)$ subtrees in some order. Note that *at least two* of the solid paths hanging off $\mathrm{SP}(r)$ have length zero because at least one of the nonpreferred subtrees is also an external node. To distinguish between a preferred and a nonpreferred external node, we will introduce a second sentinel $\widetilde{\perp}$ to P. When a leaf of P should point to an external node in a particular direction, it will point to $\perp$ if the direction is preferred; otherwise it will point to $\widetilde{\perp}$.

The situation in $\mathrm{ST}(r)$ is essentially the same as in $\mathrm{SP}(r)$ because they are both binary search trees and have exactly the same set of keys. In particular, the i-th external position of $\mathrm{ST}(r)$ will be occupied by an auxiliary tree that represents the solid path at the i-th external position of $\mathrm{SP}(r)$. This gives a precise one-one correspondence between the external positions of $\mathrm{SP}(r)$ and $\mathrm{ST}(r)$, which in turn recursively defines the interconnections among the binary search trees in the forest of T. As a corollary, this correspondence also proves that T is a binary search tree.

Root Bits. Of course, only one type of pointer is allowed by the binary search tree model. To implement the notion of dashed pointers, we will allocate a root bit in each node of T and also introduce $\widetilde{\perp}$ to T as a representation of an empty auxiliary tree. The root bit of an internal node u is marked iff u is the root of a nonempty auxiliary tree, and we also think of $\widetilde{\perp}$ as $\perp$ with its root bit marked. With this definition of root bits, a child pointer is dashed iff it points to a marked node. Any other pointer in T is solid and resides entirely within an auxiliary tree.

1.4.6 Restructuring Algorithm

Since a tangolike tree implements the solid path decomposition of a reference tree, it has to restructure if an access causes the decomposition to change. Let us develop the restructuring algorithm by considering how to serve an access σ_i using a reference tree P and a tangolike tree T.

1.4.6.1 Inside P

Although serving σ_i using P is the most straightforward, for our purpose let us also record any node from which we descend towards its nonpreferred child and the direction of the descent. By definition, these are precisely the nodes that will switch due to σ_i. We make three observations here.

(1) In our descent towards σ_i, the moment we follow a dashed pointer from a node, we know it is a switching node and we also know the direction of the switch.
(2) We always stay on a solid path until we hit a switching node, at which point we jump to another solid path via its dashed pointer.
(3) After σ_i is served and v has switched its preference, the solid subpath that starts at $\checkmark_{i-1}(v)$ will *stay* solid. This is because none of the nodes on this subpath is on the access path of σ_i and thus cannot have a new preference.

Using the above observations, the following is an algorithm that updates the solid path decomposition of P assuming that we have located σ_i in P with the additional information collected during the descent. Note that in the algorithm we will speak as if we are really manipulating the pointers in the solid path decomposition of P. This will ease our future task of converting it to restructure T instead.

⟪ update solid path decomposition of P after serving σ_i ⟫

1> Let q be a node pointer that is initially pointing at the node of σ_i. We will ascend towards $root[P]$ and maintain the invariant that q stays on $\mathrm{SP}(\sigma_i)$ throughout by patching $\mathrm{SP}(\sigma_i)$.
2> Ascend q until it hits the beginning node of $\mathrm{SP}(\sigma_i)$.
3> If $q \neq root[P]$, then by observation (2), the current parent of q must be in another solid path and is the switching node v. By symmetry, assume this is a left-to-right switch. Note that we know the direction of the switch due to observation (1).

- switch v left-to-right in $\mathrm{SP}(v)$ -

3.1> Split $\mathrm{SP}(v)$ into two by changing the left pointer of v from solid to dashed. This separates out the subpath that starts at $\checkmark_{i-1}(v)$. By observation (3), this subpath will stay solid in the new decomposition.
3.2> Join what remained of $\mathrm{SP}(v)$ with $\mathrm{SP}(\sigma_i)$ using a solid pointer. This extends $\mathrm{SP}(\sigma_i)$ upward and allows the ascent to continue without breaking the invariant.
3.3> Go back to step 2.
4> The ascent terminates once $q = root[P]$. If needed, perform the final switch at the node of σ_i by using step 3.1 and step 3.2. ∎

Before we go on, let us note that technically there is no restriction on how the switches should be ordered. However, the sequential nature of our algorithm imposes a particular one. We will explain our choice in §1.4.7.2.

1.4.6.2 Inside T

Being a simulation of the solid path decomposition of P, the happenings inside T are in fact highly similar to the above. There are two major differences, however, and both of them require us to augment each node u of T with an extra field $refdep[u]$ to store the reference depth of u. Let us look at these two differences now.

Switching Node and Direction. While descending in P, it is straightforward to identify a switching node and its new preferred direction as in observation (1). Unfortunately, this does not carry over to T. Although each auxiliary tree ST has the same set of keys as its corresponding solid path SP, in general the two of them do not have the same shape. The trick is to recall that both of them are binary search trees and thus their external positions correspond to the same set of intervals. Therefore if a search for σ_i in SP ends at its i-th external position EP, meaning $\sigma_i \notin SP$, then a search for σ_i in ST will have to end at its i-th external position ET as well.

Furthermore, the very *same* procedure is in fact applicable to both trees even though it degenerates into a trivial statement when running on SP. Let S be either SP or ST and let E be the corresponding external position.

⟪ determine the node and direction of a switch in S ⟫

1> By keeping track of the predecessor and the successor of the current node as we descend through S, let u and w be the predecessor and the successor of E.

2> If u does not exist in S, then w is the leftmost internal node of S and this must be a right-to-left switch at w. The case when w does not exist is symmetric. And as long as S is nonempty, it cannot be the case that both u and w do not exist.

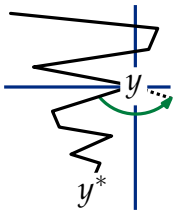
3> If both u and w exist in S, then the switch will happen at either u or w whichever is deeper in P. Furthermore, the switch is right-to-left iff E is a left child. ▮

Observe that the above procedure is well-specified as long as we have the reference depths of u and w for the test in step 3. In the case of SP, this is easy since we can keep track of the current depth as we descend through SP. But in the case of ST, there is no efficient substitute and this gives us the first reason why we need to store the *refdep* field in each node of T.

Actual Switch in T. The second difference between the situations in P and T is how we actually perform a switch on our way up. Consider a left-to-right switch at a node v containing y as its key. Let K be the set of keys on $\mathrm{SP}(y)$ that are *referentially*-deeper than y, meaning their reference depths are greater than that of y. To switch v, first we have to take out K from $\mathrm{ST}(y)$ as in step 3.1 on page 50.

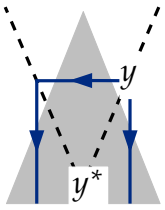
Let u and w denote the *reference* left and right parents of v if present in P and let x and z be their keys respectively. By viewing $\mathrm{SP}(y)$ as a binary search tree, observe that (i) K is the set of keys between x and y, and (ii) this set of keys is contiguous in rank in $\mathrm{SP}(y)$ and hence also in $\mathrm{ST}(y)$. Furthermore, by symmetry the set of keys K' that will replace K also have the same properties with respect to y and z. This highlights the need to locate K in $\mathrm{ST}(y)$ efficiently, and we will show how to do this using a small detour.

Bitonicity. Observe that the reference depths in any solid path SP are strictly bitonic—if we scan SP left-to-right, the reference depth increases until we reach a key y^* with the maximum reference depth and then decreases afterwards. The key y^* is also the key at the preferred leaf of SP and y^* is to the left of y iff this is a left-to-right switch.



To see how a switch at y affects $\mathrm{SP}(y)$, it is useful to picture the keys of $\mathrm{SP}(y)$ arranged in a "V" shape from left to right, with y^* at the bottom and y at some height on the rising stroke. Observe that any key to the right of y will stay in $\mathrm{SP}(y)$ because it is referentially-higher than y. If only we could split what's on the left of y at x, then we would have successfully taken K out from $\mathrm{SP}(y)$ because K is precisely the set of keys between x and y.

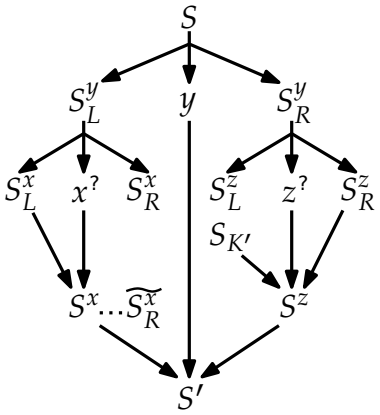
Alas, x is not necessarily in $\mathrm{SP}(y)$—this happens when y is on the left spine of $\mathrm{SP}(y)$. Worse, precisely because x can be in a solid path other than $\mathrm{SP}(y)$, it would be difficult if we need the value of x to switch at y in bottom-up setting. Fortunately, we did keep track of the reference depth of y during the descent and the keys in K are precisely those that are referentially-deeper than y. Therefore, if we can split $\mathrm{SP}(y)$ by the reference depths, then we can take K out by splitting $\mathrm{SP}(y)$ at the reference depth of y instead.



But wait! So far we have been considering $\mathrm{SP}(y)$, but actually we want to do this on $\mathrm{ST}(y)$ instead. Fortunately, $\mathrm{SP}(y)$ and $\mathrm{ST}(y)$ have exactly the same set of keys and thus the bitonicity condition also holds in $\mathrm{ST}(y)$. By changing our mental picture, the discussion above is therefore applicable to $\mathrm{ST}(y)$. All we need is the ability to split $\mathrm{ST}(y)$ by the reference depth at $\mathit{refdep}[y]$. (Notice, however, that y^* is not necessarily a leaf in $\mathrm{ST}(y)$.)

Left-to-Right Switch. Now that we understand the bitonicity condition, let us state the algorithm to perform a left-to-right switch at y in $\mathrm{ST}(y)$. Let S and S' be $\mathrm{ST}(y)$ respectively before and after the switch, and let $S_{K'}$ be the auxiliary tree corresponding to K'.

⟪ switch v left-to-right in auxiliary tree S ⟫



1> Split S by the keys at y into (S_L^y, y, S_R^y).

2> Split S_L^y by the reference depths at $\mathit{refdep}[y]$ into $(S_L^x, x^?, S_R^x)$, taking care to break ties so that $x^?$ is referentially-higher than y. The option $x^?$ will be x if x is in S, or NIL if otherwise. Notice that S_R^x contains exactly K.

3> Split S_R^y by the reference depths at $\mathit{refdep}[y]$ into $(S_L^z, z^?, S_R^z)$, taking care to break ties so that $z^?$ is referentially-higher than y. The option $z^?$ will be z if z is in S, or NIL if otherwise. Notice that S_L^z will be $\widetilde{\perp}$ since this is a left-to-right switch.

4> Mark the root of S_R^x and unmark the root of $S_{K'}$. Note that if S_R^x is $\perp$, this means changing it $\widetilde{\perp}$. This also applies to $S_{K'}$ symmetrically.

5> Join $(S_L^x, x^?, \widetilde{\perp})$ into S^x and hang S_R^x at the rightmost external position of S^x.

6> Join $(S_{K'}, z^?, S_R^z)$ into S^z.

7> Join (S^x, y, S^z) into S' to finish the switch.

Running Time and Competitiveness. To analyze the running time of serving σ_i, let us assume our auxiliary trees are implemented with a binary search tree that supports search, join, both types of split—all in either worst-case or amortized logarithmic time. The descent towards σ_i is the same as in a normal binary search tree. Although we must take care to identify the switching nodes and the direction of each switch using the algorithm on page 51, this does not add to our asymptotic running time. As the number of auxiliary trees we traverse is $O(\lg n)$ and the number of nodes in each auxiliary tree is also $O(\lg n)$, the search time is upper-

bounded by $O(\lg n \lg\lg n)$. Since each switch comprises three splits and three joins, the restructuring afterwards takes $O(\lg\lg n)$ time per auxiliary tree and thus $O(\lg n \lg\lg n)$ in total. The $O(\lg\lg n)$-competitiveness also follows immediately from the interleave bound.

1.4.7 Existing Tangolike Trees and Their History

Having defined the class of tangolike trees, let us end this section by looking at several designs in this class and how they relate to each other.

1.4.7.1 Tango Trees

The first provably $O(\lg\lg n)$-competitive binary search trees are the tango trees of Demaine, Harmon, Iacono, and Pătraşcu [DHIP04] and the name "tangolike trees" was chosen as our tribute to these trees. The simplest design in the class, tango trees deploy red-black trees [GS78] as their auxiliary trees. Besides its reference depth, each node u in an auxiliary tree ST is further augmented to store the maximum reference depth appearing in $ST|^u$. The switch procedure is largely similar to the algorithm on page 53.

Tango trees are most remarkable from the perspective of the famous "dynamic optimality conjecture". Recall that this conjecture of Sleator and Tarjan [ST85b] states that splay trees are $O(1)$-competitive against an offline optimal binary search tree. Note that if by being balanced means a binary search tree of size n has height $O(\lg n)$, then any balanced binary search tree is automatically $O(\lg n)$-competitive. However, prior to tango trees, *none* of the binary search tree algorithms known at the time has been shown to have a competitive ratio of $o(\lg n)$. This includes splay trees, which at the time have already been proven to have numerous optimal [ST85b; Tar85; Col00; Iac02] or close-optimal [Luc88a; Sun92; CH93] properties. (See also [Pet08] for a new result after 2004.) Indeed, one of the major open problems in the field is whether splay trees are $o(\lg n)$-competitive.

However, tango trees are also known to be $\Theta(\lg\lg n)$-competitive even at the time of publication. In fact, access sequences that are tight for tango trees are easy to construct. Although this rules out the possibility that tango trees are dynamically optimal, a previously inaccessible design space has been opened up for several follow-up works.

Super Tango Trees. Besides appearing in [DHIP04; DHIP07], tango trees also form a major part of the thesis of Harmon [Har06]. Inside, Harmon described another data structure known as "super tango trees" which simultaneously maintains a tango tree and an *explicit* reference tree. Even

though this puts super tango trees out of the binary search tree model, super tango trees may be considered to be easier to reason about.

1.4.7.2 Multi-Splay Trees

The success of tango trees was quickly followed-up in the work of multi-splay trees by Wang, Derryberry, and Sleator [WDS06]. As the name suggests, these trees are based on splay trees [ST85b] and indeed one may think of multi-splay trees as tango trees with each red-black tree replaced by a splay tree. Besides its reference depth, each node u in an auxiliary tree ST is augmented with an extra field $mindep[u]$ to store the minimum reference depth appearing in $ST|^u$. The switch procedure is also largely similar to the algorithm on page 53.

Although on the surface there seems to be little changes between the two, in comparison multi-splay trees do have some theoretical advantage over tango trees. Specifically, Wang et al. have shown that the worst-case and amortized running times of a multi-splay tree of size n is, respectively, $O(\lg^2 n)$ and $O(\lg n)$. The latter in particular proves that our estimate of $O(\lg n \lg\lg n)$ on page 53 is loose. Therefore, unlike tango trees which are $\Theta(\lg\lg n)$-competitive, multi-splay trees are *possibly* $O(1)$-competitive.

Furthermore, in his thesis, Wang [Wan06] has shown that multi-splay trees enjoy several optimal properties such as working set and another property known as "deque" (as defined in [Tar85]). While possessing these properties are not sufficient for being $O(1)$-competitive, they certainly are necessary. Indeed, splay trees and tango trees as the only candidates for dynamic optimality as of 2006; but subsequently a new binary search tree algorithm known as "GreedyOSS" has been proposed by Demaine, Harmon, Iacono, Kane, and Pătraşcu [DHIKP09] as a third candidate.

To wrap up, we note that Wang, Derryberry, and Sleator [WDS06] have also proposed a generalization of the binary search tree model that supports updates. By allowing the reference tree to restructure as a red-black tree [GS78], multi-splay trees can support insertions and deletions and are also $O(\lg\lg n)$-competitive in this model. This is the primary reason why we wanted to keep the shape of the reference tree flexible on page 48. Multi-splay trees also affect another aspect of our description of tangolike trees. In particular, the somewhat peculiar ordering of switches in our restructuring algorithm on page 50 is the one used in the existing analysis of multi-splay trees. (For example, the proof of the scanning theorem in [WDS06] depends on this order critically.)

1.4.7.3 Chain-Splay Trees

A perhaps less-visible follow-up work to tango trees is the chain-splay trees of Georgakopoulos [Geo05] (see also [Geo08] for an updated version of this design). Similar to multi-splay trees, chain-splay trees can also be seen as tango trees that use splay trees instead of red-black trees. The crucial difference is that a chain-splay tree does not require its nodes to store any extra fields other than *refdep*, which is favorable when compared to multi-splay trees because the nodes in the latter would also store the *mindep* field.

However, it is not known if this advantage in space would end up helping or hurting chain-splay trees in their actual running times. Specifically, when Wang et al. [WDS06] proved that a multi-splay tree of size n can serve an access in amortized $O(\lg n)$ time, they needed a property of the multi-splaying algorithm which intuitively *seems* to require the *mindep* field. Since the restructuring algorithm of a chain-splay tree does have access to this field, it can end up performing two extra splays when compared to the corresponding multi-splay tree. As the proofs of Wang et al. do not cover these two splays, it is not clear if chain-splay trees share any other property with multi-splay trees besides the $O(\lg \lg n)$-competitiveness. Indeed, as of this writing, it is not known if a chain-splay tree can serve an access in amortized $o(\lg n \lg \lg n)$ time. For a discussion of this finer point, we refer the reader to the thesis of Wang [Wan06, §3.5].

1.4.7.4 Improved Tango Trees

As we noted, the fact that tango trees are not $O(1)$-competitive has already been pointed out by Demaine, Harmon, Iacono, and Pătraşcu in [DHIP04]. But subsequently, the authors have also sketched an unnamed variant of tango tree in [DHIP07] based on another type of auxiliary tree. Demaine et al. have shown that this variant can serve each access in worst-case $O(\lg n)$ time. Seeing that this variant is unnamed *and* so is the type of the auxiliary trees used in it, we took the liberty to call this variant "improved tango trees" to distinguish it from the original tango trees. The description of improved tango trees by Demaine et al. is very concise and we will reproduce it here in full [DHIP07, pp. 243–244].

> [...] Namely, we can replace the auxiliary tree data structure with a balanced BST supporting search, split and concatenate operations in the worst-case dynamic finger bound, $O(1 + \lg r)$ worst-case time, where r is 1 plus the rank difference between the accessed element and the previously accessed element. For

example, one such data structure maintains the previously accessed element at the root and has subtrees hanging off the spine with size roughly exponentially increasing with distance from the root. Following the analysis in Section 3.4, the cost of accessing an element in the tree of auxiliary trees then becomes $O\left(\sum_{i=1}^{k}(1+\lg r_i)\right)$, where $r_1, r_2, \ldots, r_k$ are the numbers of nodes along the k preferred paths we visit; thus $r_1 + r_2 + \cdots + r_k = \Theta(\lg n)$. This cost is maximized up to constant factors when $r_1 = r_2 = \cdots = r_k = \Theta\left(\frac{\lg n}{k}\right)$, for a cost of $O\left(k\left(1+\lg\frac{\lg n}{k}\right)\right)$. Because $k = O(\lg n)$, we obtain an $O(\lg n)$ worst-case time bound per access. [...]

1.4.7.5 Poketrees

While not a tangolike tree in our definition, a highly-related work is the "poketrees" by Kujala and Elomaa [KE06]. A poketree is essentially a direction implementation of a reference tree augmented with extra pointers between its *own* nodes and these pointers are known as "dynamic links".

One way to think of the dynamic links of a poketree is to observe that the key space spanned by a node on a solid path decreases as we get deeper. When we view it this way, the switching node on the root solid path is simply the deepest node that still spans the search target. Kujala and Elomaa have therefore arranged the dynamic links so that the switching node within a solid path of size k can be located in worst-case $O(\lg k)$ time. While it is already nontrivial to update the dynamic links to reflect node preferences after each access, Kujala and Elomaa have managed to take advantage of the existing pointers of the underlying reference tree so that any search runs in *worst-case* $O(\lg n)$ time. In fact, their design even supports insertions and deletions just like multi-splay trees. We refer the reader to [KE06] for details.

1.5 A Brief History of Finger Search

To bring this introduction to its end, let us briefly survey the history of finger search and see how the four topics in §1.1–§1.4 fit into the picture. Unfortunately, this also means we can no longer control the model nor the representation we are working with. Therefore, in this section, an access means either a search or an update and the representations in our description can have technically insignificant differences to the original. In what follows, let n denote the number of keys in a data structure and let d denote the rank distance between the target and the finger *being used* in an

dictionary operation. (The emphasis is important for data structures that support more than one finger.)

1.5.1 The First Year

Finger search was originally introduced on a variant of leaf-store B-trees by Guibas, McCreight, Plass, and Roberts [GMPR77]. In this variant, any leaf can be designated as a finger through the creation of a "finger path" from the leaf up to the root. Each internal node is augmented with five pointers, three of which are used in level-linking. The remaining two are paired with an extra bit at each node to facilitate a special kind of threading that we will not describe. Unlike the root-start access algorithms in §1.1.2, an access in this design takes a finger as an additional input and starts at the leaf underneath the finger. Finger searching are then performed in a way similar to the leaf-start algorithm in §1.2.3.2.

The focus of Guibas et al., however, was in supporting updates so that their running time bound is asymptotically the same as that of searches. In order to do this, a regularity condition is imposed on the number of keys appearing in the nodes on a finger path. By suitably partitioning the possible node sizes into digits, they showed how to manipulate a finger path as a form of redundant counter[9]. For example, a carry is a key promotion and a borrow is a key demotion. But in order for this regularity condition to hold, each node in their design is required to contain between $\lfloor \frac{m}{3} - 1 \rfloor$ and m keys for some $m \geq 24$. In other words, the smallest such node is an $(8, 25)$ node.

Guibas et al. have also specifically described how multiple fingers can be supported in their design. With k fingers set up, the *worst-case* running time of any access is $O(k + \lg d)$. We note that the $O(k)$ term in this bound is not related to identifying which finger should be used. Instead, this term accounts for the time in updating the other $(k-1)$ fingers during an access.

Groundbreaking as it is, the design of Guibas et al. does have two issues. First, the running time can be dominated by the number of fingers which we denoted as k. Notice that this possibility only becomes more likely when d is small and finger search is most desirable. Second, the

9. A redundant counter is a representation of a number in certain forms of a positional number system in which one number has multiple representations. By imposing certain regularity conditions on the digits, it can be shown that the carry induced by incrementing or decrementing *any* digit of a redundant counter only affects a constant number of digits in total. Notice that this cannot be achieved with, say, the typical binary or decimal representation of a number. The subject has a rich history but it is out of the scope of this thesis. We refer the reader to [CK77], [KT96; Kap97], [Oka96], and [Bro96] for some references and applications.

fingers in this design are in fact *nonmovable* and each finger can require $\Theta(\lg n)$ time to construct and to destruct. In applications such as adaptive sorting described in §1.2.4.3, determining the finger locations can become an interesting challenge.

1.5.2 A Divergence

The design of Guibas et al. in [GMPR77] spurred several follow-ups that can be classified as the beginning of two different lines of work. As we will see, an interesting divergence would emerge among these two lines.

Worst-Case Dynamic Finger Trees. In one line of work, Kosaraju [Kos81] introduced a clever data structure in which each access runs in *worst-case* $O(\lg k + \lg d)$ time and can be implemented in either the leaf-store or the node-store representation. Not only does his design improve the dependency on the number of fingers k from linear to logarithmic, the fingers in this data structure are in fact *dynamic*[10]. The latter in particular means that we can move a finger used in an access to the location of the access itself with no change to the asymptotic time bound.

We note that the keys under the fingers are stored in another balanced search tree and the $O(\lg k)$ term here represents the time spent to search for the closest finger to use in an access. Furthermore, we also note that if we restrict to the case of one finger, Kosaraju's data structure can be viewed as a dynamic finger tree augmented with only $\Theta(\lg n)$ extra pointer(s). This is a crucial point that we will revisit in §4 where Kosaraju's design as well as its relationship to our own dynamic finger tree design is discussed.

Amortized Finger Search Trees. In another line of work, Brown and Tarjan [BT80] applied the technique of level-linking used by[11] Guibas et al. to $(2,3)$-trees [AHU74]. A similar approach is then taken by Huddleston and Mehlhorn in [HM82] on weak B-trees, which were introduced earlier by the same authors in [HM81].

As we have seen in §1.2.3.2, a level-linked degree-balanced search tree must be a leaf-tree and can readily support finger searching in worst-case $O(\lg d)$ time because of its pointer structure. It is also a finger search tree—meaning that a finger search can start at any given leaf—and is thus more versatile than the data structure designed by Kosaraju. Its *amortized* update time bound, however, is where the critical difference lies in comparison

10. "Movable" is what the literature at the time would say, e.g., see the title of [BT78].
11. We refrain to use "introduced by" here because so far we have not been able to track down if Guibas et al. [GMPR77] originated the idea of level-linking. This is something we hope to know more about one day.

to the worst-case bound guaranteed by Kosaraju's design. Having said that, amortized level-linked degree-balanced search trees are hard to beat in terms of their simplicity. This is true when compared to the design of Kosaraju in [Kos81], *more* so with respect to the design of Guibas et al. in [GMPR77], and *much more* so with respect to a worst-case level-linked degree-balanced search tree design that we will get to.

1.5.3 More on Dynamic Finger Trees

Swinging back to dynamic finger trees, Tsakalidis [Tsa85] has introduced a variant of AVL-trees [AVL62] with an inverted spine which he called "inclined AVL-trees". Since a static finger is available at one end of the tree, a dynamic finger traveling in a fixed direction can be supported by repeated splits. Although Tsakalidis did not mention joins nor inverting both spines, these can be seen as easy extensions. Therefore, we would also say his design can be used to support a dynamic finger with no restriction in its travel direction. Note that as with almost all dynamic finger trees, both leaf-store and node-store representations can be used his design.

As the reader will certainly recall from §1.3, inverted spines with splits and joins is also the key idea in the design of heterogeneous red-black trees by Tarjan and Van Wyk [TVW88]. This is where we see a tradeoff between worst-case and amortized designs. The design of Tsakalidis, in particular, has worst-case guarantees because a regularity constraint is imposed on the height-imbalance of the nodes on the inverted spine; but enforcing this constraint would require a careful manipulation of up to $\Theta(\lg n)$ extra pointers on the spine after each update. This is in contrast with the amortized design of Tarjan and Van Wyk in which no constraint has to be enforced—in fact, the robustness of red-black trees is all it needs. However, the above comparison is made from the perspective of finger search alone. To make this more fair, we must note that the repertoire of operations considered by Tarjan and Van Wyk is much broader than the ones considered by Tsakalidis. In particular, besides splits and joins, through augmentation the former also includes search by rank and search by a secondary heap value.

The technique of inverted spines is also a crucial idea in a design of *purely-functional* catenable sorted lists by Kaplan and Tarjan [KT96] (see also the thesis of Kaplan [Kap97]). In fact, two designs of such lists are described in [KT96], and the second can be said to organize a forest of $(2,3)$-trees using a purely-functional data structure that resembles an inverted spine. Both designs of Kaplan and Tarjan in [KT96] have *worst-case* guarantees

	Insert	Delete
Harel [Har80]	$O(\lg^* n)$	$O(\lg^* n)$
Fleischer [Fle96]	$O(\lg^* n)$	not supported
Brodal [Bro98]	$O(1)$	$O(\lg^* n)$
Brodal et al. [BLMTT03]	$O(1)$	$O(1)$

Table 1.3 – Update time of four major worst-case finger search tree designs

for the dictionary operations as well as splits and joins, which for purely-functional data structures are highly nontrivial feats. However, the details of their inner-workings are also nontrivial and we note that a much simpler amortized design has been proposed by Hinze and Paterson in [HP06].

Last but not least, we must also note that the splay trees of Sleator and Tarjan [ST85b] are also dynamic finger trees due to a renowned result by Cole [Col00]. As is natural for splay trees, the performance guarantee of a finger search is amortized, and we also note that it continues to hold even in the presence of updates at the finger. At present, it is not known if splay trees can support more than one dynamic finger, and the problem of supporting multiple fingers becomes particularly hard when updates have to be considered. For example, an open question in the field is a highly special case of the latter in which a splay tree is used as a doubly-ended queue, or deque for short [Tar85; Sun92; Elm04; Pet08].

1.5.4 More on Finger Search Trees

Deterministic. To swing back to finger search trees, let us start with Table 1.3 which lists four major worst-case finger search tree designs. As is usual for finger search trees, all of these designs are leaf-trees and support finger search in worst-case $O(\lg d)$ time. What's different among them is the update time, and in one case whether deletion is supported or not. We note that the update times in the table are all *worst-case* and are stated assuming the location of an update is known. Otherwise, all $O(1)$ entries should simply be replaced with $O(\lg d)$. Also note that the last design by Brodal, Lagogiannis, Makris, Tsakalidis, and Tsichlas [BLMTT03] is the convergence of the two lines of work we mentioned in §1.5.2 and can be considered as a perfect solution to the problem first studied by Guibas et al. [GMPR77] over a quarter century ago.

Randomized. Optimal as it is, the design of Brodal et al. is in fact rather complicated—as the authors themselves would point out in [BLMTT03, p. 417]. This is a perfect example where allowing randomization can give rise to a tremendously simpler data structure.

Let us start by noting that we can use a randomized method of Seidel and Aragon [SA96] to maintain a binary search tree under updates such that it remains a *random* binary search tree throughout. We will follow Seidel and Aragon and call any binary search tree maintained with their method a "treap". What's interesting from our perspective is that their analysis of treaps actually shows a random binary search tree is also a finger search tree supporting $O(1)$ updates in the *expected case*. More precisely, Seidel and Aragon have shown that by taking expectation over the random bits used in the maintenance of a treap, (i) the expected update time of a treap is $O(1)$ assuming the location of the update is known, and (ii) the expected length of the simple path between two keys that are d ranks apart in a treap is $O(\lg d)$.

It is important to realize that the mere existence of the path in (ii) does not mean we can traverse it efficiently—this is true even if parent pointers are implemented. In fact, Seidel and Aragon gave three methods to perform finger searching in a treap, each requiring a different augmentation. Their last method, which also uses the least augmentation among the three, only requires a parent pointer and a "spinal" bit in each node to indicate whether a node is on the two spines.

As it turns out, the spinal bit can be eliminated with an improved algorithm as described by Brodal in [Bro04]. Small as a one-bit improvement may seem, this new algorithm actually performs equally well in both the unweighted and the *weighted* case of the problem—just like the other two method of Seidel and Aragon. Although we will not discuss the weighted case here, let us simply note that the analysis of the weighted case is one of the distinguishing features of [SA96] when compared to the work of Martínez and Roura in [MR98]. The latter is specifically focused on the maintenance of a random binary search tree under updates and the analysis inside does not treat finger searching *per se*. Readers who are interested in the intricacies of this subject and how random binary search trees relate to finger search are referred to [Vui80], [AS89; SA96], [RM96; MR98], [DN04], and [KP07] for starting points and references.

Besides treaps, we must also note that the skip lists of Pugh [Pug90b; Pug90a] can also be seen as randomized finger search trees with $O(1)$ updates. The truth is, even though technically a skip list is not a search tree, there are many interesting parallels between it and a random binary search tree. For one thing, skip lists have also been extended to handle arbitrarily weighted keys by Bagchi, Buchsbaum, and Goodrich [BBG05]. We believe the time bounds in [BBG05, Table 1] will give the reader a

useful perspective of what's common among a skip list and a random binary search tree and exactly where they differ.

Finally, we remark that the work of Bagchi et al. in [BBG05] also includes *deterministic* biased skip lists. Curiously, this varaiant actually has matching worst-case guarantees as its randomized counterpart. Unfortunately, as the authors themselves have noted, their current design is different from what we expect a *derandomized* skip list might be. In particular, even if we were to "unbias" a deterministic biased skip list by letting all keys have the same weight, the time bound on an update would still be worst-case $O(\lg n)$. Indeed, had this been worst-case $O(1)$, which is what a derandomized skip list would have, this design would match the optimal finger search tree design of Brodal et al. [BLMTT03] while being much simpler.

RAM. So far we have only been looking at pointer-based designs, but finger searching has also been considered with the power of RAM. In particular, Dietz and Raman [DR94] started by refining a bucketing technique that was used earlier by Levcopoulos and Overmars [LO88] and also by Dietz and Sleator [DS87], and then applied it on a level-linked $(2,3)$-tree [BT80]. The result is a comparison-based design that supports finger searching in worst-case $O(\lg d)$ time and update in worst-case $O(1)$ time. The only reason why this design is not a pointer algorithm[12] is because it uses a table-lookup technique to speed up the bucket updates.

Perhaps not surprisingly, the above design does not really make full use of the power of RAM. Indeed, Andersson and Thorup [AT07] have an improved finger search tree in the RAM model that has worst-case $O(1)$ update cost and worst-case $O(\sqrt{\lg d / \lg \lg d})$ finger search cost. Note that this is *optimal* since it matches a corresponding lowerbound on predecessor search by Beame and Fich [BF02]. Furthermore, we also note that the result of Andersson and Thorup can be strengthened by breaking down into cases and can also be implemented in AC^0 operations but with a weaker bound. For the exact statement, the reader is referred to [AT07, Theorem 1.5].

Unified Structures. We will close this review with a class of data structures that are not exactly finger search trees but are nonetheless relevant to the dynamic finger property.

In §1.2.5.3, we have seen evidence that the dynamic finger property and the working set property do not capture each other. But before we use this to conclude that there are two irreconcilable types of locality, let us look at

12. This terminology was suggested by Ben-Amram [BA95] in his survey on the meaning of the term "pointer machine".

a new class of data structures that also be said to "support finger search" but in the following technical sense.

Instead of finger searching for x from a given finger, we will consider a model in which we can finger search *simultaneously* from all keys. However, the finger search that was started from the key f will incur an additional cost of $O(\lg t(f))$, with $t(f)$ being the working set number of f as defined on page 33. The cost to search for x is the minimum taken over the cost of all searches, or more succinctly $O(\min_f(\lg t(f) + \lg \mathrm{dist}(f, x)))$. Notice that this can be no worse than the cost incurred by finger searching with a dynamic finger as well as the cost incurred by a data structure with the working set property.

Miraculous as it may seem, a neat data structure that was designed by Iacono [Iac01] does have this guarantee. Furthermore, it has also been extended by Badoiu, Cole, Demaine, and Iacono [BCDI07] to support updates. Iacono calls the property captured by his data structure the unified property. As it turns out, this property has a natural interpretation using the data compression perspective in §1.2.5. However, it is also known that if a data structure is *tight* with respect to the above bound, then it is not $O(1)$-competitive against an optimal binary search tree. We note that quite a bit more about the unified property is known and interested readers are referred to the forth-coming thesis of Derryberry [Der08].

2 Heterogeneous Decompositions

THE MAIN CONCEPT introduced by this thesis is the heterogeneous decompositions of a degree-balanced search tree and we will define it in this chapter. To develop our intuition, let us start by handling a special case in §2.1 where we show how a complete binary search tree can be heterogeneously decomposed. Then we will use this special case to guide us when we generalize our definitions to degree-balanced search trees in §2.2. Besides presenting these definitions, this chapter also serves two other purposes. First, although the concept of heterogeneous decompositions is very much inspired by the heterogeneous finger search trees of Tarjan and Van Wyk [TVW88], there are also some critical differences among them. This will be explained in §2.3. Second, we will explore the relationship between finger search and heterogeneous decompositions in §2.4 by showing how to think about a finger search via excisions on heterogeneous decompositions. Such excision arguments will be used many times in the remaining chapters of this thesis.

Lists and Types. Before we proceed, let us extend our algorithmic notation to include catenable lists that support linear-time access from both ends. More specifically, the running time to access the j-th value of such a list is $O(j)$, where j can be counted from either ends. Note that this does not preclude our lists to have an asymptotically faster running time such as $O(\lg j)$. The length of a list is the number of values appearing in it. For a

list Q' to be a sublist of a list Q, the values of Q' must appear *contiguously* inside Q. As an example as well as an illustration of our notation, if the variables a through e denote some values, then $Q = \{a; \{b; c\}; d; \{e\}\}$ is a list of length 4 with its second and last values being lists themselves. The list $\{\{b; c\}; d\}$ is a sublist of Q, but the lists $\{a, d\}$ and $\{\{c\}; d\}$ are not. We will use $\textsc{Concat}(\cdot, \cdot)$ to denote the procedure that catenates two lists. Although we will not specify the actual implementation here, we require Concat to run in worst-case $O(1)$ time but it is not required to preserve its two inputs. One data structure that fits these requirements is a doubly-linked list.

It will also be convenient to introduce two type variables so that we can better distinguish among keys and nodes inside a list. We will let α denote the type of a key and τ denote the type of a node. In our usage, a tree will also be of type τ, which means we will be representing a tree using its root. We remark that we have intentionally kept our use of types at its most rudimentary; in particular, we will allow a list to contain multiple types of values even though the lists in our applications can all be homogeneously typed if only we choose to treat them more properly.

2.1 For Complete Binary Search Trees

Given a complete binary search tree T, we can heterogeneously decompose T with respect to any one of its keys. This means that if T has size n, then it has n different heterogeneous decompositions. But with our interest in finger search, most of the time we take the decomposition with respect to the key that is currently under a dynamic finger. The tree T is called the reference tree of the decomposition. Within this section (§2.1) we will restrict T to be a complete binary search tree.

2.1.1 Heterogeneous Decomposition

Given a complete binary search tree T and one of its keys f, the heterogeneous decomposition of T with respect to f is a triple (L, f, R), where L and R are lists of odd lengths comprising two alternating types of values. Appearing in the same order as they do in T, the odd positions of L/R are the left/right subtrees hanging off the access path of f and the even positions of L/R are the left/right ancestor keys of f. Although the above already defines L and R very rigidly, let us use an algorithm in §2.1.1.1 as our definition. This will allow us to gain some more intuition about the content of L and R, which will be useful when we generalize them in §2.2. Let us also remark that this is one of the few places in which we specify our algorithms in the style of [CLRS01].

```
HETERO-DECOMP-CBST(u, f)
1  if f = key[u]
2     then return ({left[u]}, key[u], {right[u]})                ▷ base case
3  elseif f < key[u]
4     then (L', ·, R') ← HETERO-DECOMP-CBST(left[u], f)          ▷ descend left
5          return (L', f, CONCAT(R', {key[u]; right[u]}))
6     else (L', ·, R') ← HETERO-DECOMP-CBST(right[u], f)         ▷ descend right
7          return (CONCAT({left[u]; key[u]}, L'), f, R')
```

Figure 2.1 – Procedure HETERO-DECOMP-CBST

2.1.1.1 Definition

Given the root u of a complete binary search tree T and one of its keys f, the procedure HETERO-DECOMP-CBST in Figure 2.1 returns the triple that represents the heterogeneous decomposition of T with respect to f. By inspecting the procedure, it is easy to see that we are recursively searching for f in T. Let (L, f, R) be the final triple returned by the call. We make the following observations.

(1) Since f is one of the keys in T, this procedure always terminates at the base case in line 2.

(2) Each time when we make a recursive call in line 4/line 6, we keep track of u by its key and whichever subtree of u that does *not* contain f by its root using a suitable CONCAT in line 5/line 7. Note that $key[u]$ is a right/left ancestor key of f.

(3) Being a top-down search for f, *every* left or right ancestor key of f will appear as $key[u]$ at some point in the recursion.

(4) The catenations in line 5/line 7 preserve the symmetric order inside R/L in the following sense—if $\{T_1'; x; T_2'\}$ is a sublist of R/L, then every key in T_1' is smaller than x, which in turn is smaller than every key in T_2'.

(5) Both the leftmost and rightmost values of both L and R are of type τ and some of them can be $\perp$.

Convenience Measures. Before we go on, let us deploy a few tricks to simplify the presentation to follow.

Sentinels. We add sentinels to the two ends of L and R as well as the key space of T. For L, we add the sentinels $-\infty$ and f; for R, we add f and ∞; for T, we add $-\infty$ and ∞. Note that the sentinels are *not* part of L, R, or T; in particular, the leftmost and rightmost values of both L and R do not change. However, now that we have the sentinels defined, we can refer to the keys preceding and succeeding them in L and R inside our analysis.

Furthermore, the ranks of the $-\infty$ and the ∞ sentinels with respect to T are defined to be 0 and one plus the current size of T.

Inner/Outer vs. Left/Right. Since we will be dealing with left-right symmetry extensively, we will start referring to the direction towards f as inner and the other direction as outer. For example, in the situation of line 5 in HETERO-DECOMP-CBST, we see that $right[u]$ will succeed $key[u]$ in R. Using the notation introduced here, $key[u]$ is the inner key of the subtree rooted at $right[u]$ in R and this subtree is the outer subtree of $key[u]$ in R.

Right Majorization. When symmetry applies, we will only deal with the situation on the right hand side and mark a statement or a theorem as such using the $(\mathcal{R})$ symbol. The left hand side symbol is $(\mathcal{L})$ and will *only* be called upon when we need to refer to the left hand side of a theorem. In other words, say if we refer to Theorem 2.2 below, then we mean Theorem 2.2 $(\mathcal{R})$ since this theorem is specified as $(\mathcal{R})$. To refer to its left hand variant, we will have to specifically refer to Theorem 2.2 $(\mathcal{L})$.

2.1.1.2 Analysis

Theorem 2.1 Given the root u of a complete binary search tree T and one of its keys f, the procedure HETERO-DECOMP-CBST(u, f) runs in $O(\lg|T|)$ time. Furthermore, the total size of the triple returned is $O(\lg|T|)$.

Proof. Since HETERO-DECOMP-CBST recurs only on successively deeper keys of T in line 4 and line 6, the running time bound follows. The space bound follows from the time bound and the fact that R/L can only grow by two values in line 4/6. □

Theorem 2.2 $(\mathcal{R})$ Let (L, f, R) be the heterogeneous decomposition of a complete binary search tree T with respect to one of its keys f and let u^* be a node in T containing the key x. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then (i) x is either f or a right ancestor key of f, and (ii) T' is the right subtree of u^*, and (iii) z is the right parent key of x. Note that x and z may be sentinels.

Proof. If x is f itself, then the first point is trivially true. Otherwise, observe that the test on line 3 guarantees that for $x = key[u]$ to be added to R' on line 5, f is in the left subtree of x and thus x is a right ancestor key of f.

To show the second point, we merely need to inspect line 2 or line 5 of HETERO-DECOMP-CBST. The former applies if x is the f sentinel; the latter if otherwise.

For the last point, first assume that z is not the ∞ sentinel. From observation (2), z is a right ancestor key of f; and if x is not the f sentinel, then x is one also. Since we never append to the left to R in the procedure,

z is a right ancestor key of every key to the left of it in R. As we did not skip any node on the access path of f, all right ancestor keys of f are in R. Combining the two, z is the right parent key of x whether x is the f sentinel or not.

Finally, if z is the ∞ sentinel, then T' is the rightmost subtree occurring in R. This happens either because of line 2 in which x is in the root, or because of line 5 in which x is the *first* key from which we descend left. In either case, x is on the right spine of T and our definition of the right parent key of x is precisely z. □

Theorem 2.3 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a complete binary search tree T with respect to one of its keys f and let u^* be a node in T containing the key x. If x is either f or one of its right ancestor keys in T and z is the right ancestor key of x, then $\{x; right[x]; z\}$ is a sublist of R. Note that x and z may be sentinels.

Proof. Consider when HETERO-DECOMP-CBST(u^*, f) is the most recent call and let (L^x, x, R^x) be the triple returned by this call. Note that this call must happen because of the identity of x and observation (3) on page 67.

Assuming that z is not the ∞ sentinel, we must have descended left from z on line 4 some time ago and there was not another left descend until we reach the present call. If x is f, then we will return $\{right[x]\}$ as R^x on line 2. otherwise, we will descend left from x on line 4 and then append $\{x; right[x]\}$ to R' from the recursive call. In both cases, when we rewind back to the call in which u contains z, the rightmost value of R' from the recursive call on line 4 will be $right[x]$ and we will append $\{z; right[z]\}$ to it on line 5. This proves that R contains $\{right[x]; z\}$ as a sublist if x is f, or $\{x; right[x]; z\}$ if otherwise.

If z is the ∞ sentinel, then by its definition x must be on the right spine of T. This means we have always descended right until reaching the current call. The two cases considered in the above analysis still applies, except that nothing will be appended to the right of R^x afterwards. Since R^x is the rightmost sublist of R, the theorem follows. □

Remark 2.4. Theorem 2.2 and Theorem 2.3 and their generalizations to be given as Theorem 2.12 and Theorem 2.13 can be considered to be the necessary and the sufficient condition for a list $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ to appear in R.

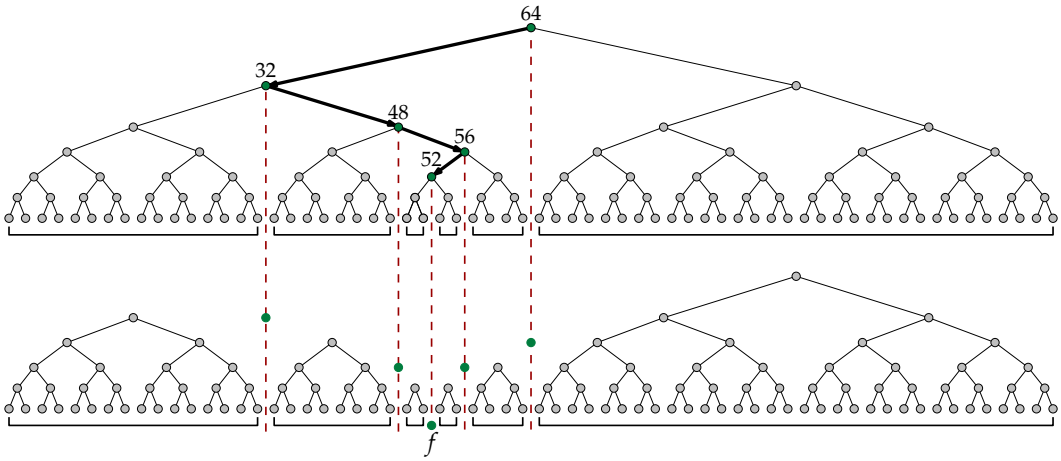


Figure 2.2 – Heterogeneous decomposition of a complete binary search tree of 127 keys with respect to the key f at rank 52, with the keys on the access path of f drawn at their respective heterogeneous heights (see Example 2.1 and Example 2.2)

Example 2.1. *Heterogeneous Decompositions*.

Refer to Figure 2.2. Assuming the set of keys in the reference tree is $\{1, 2, \ldots, 127\}$, the heterogeneous decomposition of the tree with respect to f is the triple

$$(\{T_L^{32}; 32; T_L^{48}; 48; T_L^{52}\}, 52, \{T_R^{52}; 56; T_R^{56}; 64; T_R^{64}\})$$

where T_L^x and T_R^x denote the left and right subtrees of the key x respectively.

The figure is vertically divided into two parts. The upper part shows the reference tree with the ancestor keys of f labeled with their ranks and with the edges on the access path of f turned into arrows pointing towards f. The lower part shows a collection of subtrees interleaved with the ancestor keys of f while keeping the horizontal position of all keys intact. These are precisely the subtrees and keys appearing in the decomposition. The heights of the ancestor keys will be explained in §2.1.2.

In the figure, any filled area is an abstract representation of a key and the black strokes indicate the link structure and the node boundaries. As a visual aid, the ancestors keys of f are colored green in both the upper and lower parts. Note that the areas corresponding to these keys in the lower part of the figure are *not* surrounded by black strokes. Furthermore, dashed vertical lines in red have been drawn underneath these keys to assist the identification of subtrees in the decomposition. Finally, brackets have also been drawn at the bottom of both parts to indicate the partitioning of the key space by f and its ancestor keys.

2.1.2 Heterogeneous Height

Let T continue to denote a complete binary search tree and let f be one of its keys. A notion of heterogeneous height with respect to f is associated with each key on the access path of f; any other key in T does not have a heterogeneous height. Note that since T is a complete binary search tree, the keys on the access path of f are either f itself or an ancestor key of f.

To prevent any confusion, we will always use the longer term "heterogeneous height" in full and reserve the shorter term "height" for its original meaning as defined on page 5. The reader is reminded that the latter is denoted $\mathrm{ah}(\cdot)$, which is our mnemonic of "actual height".

2.1.2.1 Definition

Let (L, f, R) be the heterogeneous decomposition of T with respect to one of its keys f and let z be an ancestor key of f. The heterogeneous height of z with respect to f will be denoted as $\mathrm{hh}_f^T(z)$.

(1) As the base case, $\mathrm{hh}_f^T(f)$ is defined to be 0.

(2) If the inner subtree of z in the decomposition is T', then $\mathrm{hh}_f^T(z)$ is defined to be $(\mathrm{ah}(T') + 1)$. The superscript will be dropped when the context makes it clear, but the subscript will never be dropped. Note that z may be the $-\infty$ or ∞ sentinel.

(3) In the setting of (2) above, we say that z is supported by T' in the decomposition and the amount of support by T' is $\mathrm{ah}(T')$.

2.1.2.2 Analysis

Even though their proofs are very simple, the following theorems about heterogeneous heights will be used many times later in this thesis. The reader is encouraged to use Example 2.2 to familiarize them.

Theorem 2.5 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a complete binary search tree T with respect to one of its keys f. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then $\mathrm{hh}_f(z) = \mathrm{ah}(x)$. Note that x and z may be sentinels.

Proof. Let u^* be the node containing x in T. By Theorem 2.2, T' is the right subtree of u^*. The theorem follows because $\mathrm{hh}_f(z)$ is defined to be $(\mathrm{ah}(T') + 1)$ and this value is $\mathrm{ah}(x)$. □

Theorem 2.6 Let (L, f, R) be the heterogeneous decomposition of a complete binary search tree T with respect to one of its keys f. Consider any key z in L or R. If $\mathrm{hh}_f(z)$ is h, then $\mathrm{dist}_T(f, z) \geq 2^{h-1}$. In the asymptotic notation, $\lg \mathrm{dist}_T(f, z) = \Omega(\mathrm{hh}_f(z))$.

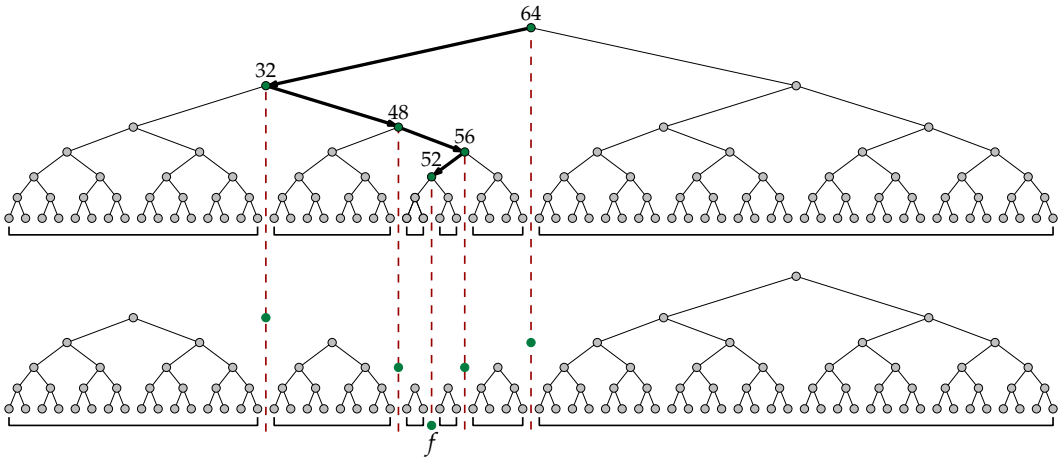


Figure 2.3 – Duplicate of Figure 2.2 on page 70

Example 2.2. *Heterogeneous Heights*.

Figure 2.3 is a duplicate of Figure 2.2, which shows the heterogeneous decomposition of a complete binary search tree of 127 keys with respect to the key f at rank 52. Using the definition above, the heterogeneous heights of the keys at rank 32, 48, 52, 56, and 64 are 5, 3, 0, 3, and 4 respectively. Note that these are exactly the ancestor keys of f in the tree.

In the lower part of the figure, these keys have been drawn at their respective heterogeneous heights. Note that the base of the figure where the brackets are drawn is at height 0 and the leaves of the subtrees are drawn at height 1.

To better understand the meaning of heterogeneous heights, Table 2.1 is an illustration of applying Theorem 2.5 to the example in Figure 2.3.

	$-\infty$	L		f	R		∞
rank		32	48	52	56	64	
ah		↙6	↙5	↙3↘	4↘	7↘	
hh	6	5	3	0	3	4	7

Table 2.1 – A visualization of applying Theorem 2.5 to the example in Figure 2.3

Proof. By definition, for z to attain a heterogeneous height of h, the inner subtree T' of z in the decomposition must have height $(h-1)$. Since T' is a subtree of a complete binary search tree, it has exactly $(2^{h-1}-1)$ keys. The theorem follows because $\text{dist}_T(f,z) \geq 1+|T'|$. □

Remark 2.7. Note that Theorem 2.6 does not allow z to be the f sentinel and the inequality $\mathrm{dist}_T(f,z) \geq 2^{h-1}$ is in fact *tight*. This can be witnessed when f has rank 1 and z has rank 2, which implies that $h = 1$ and $\mathrm{dist}_T(f,z) = 1$.

Interpretation. Let us interpret Theorem 2.6 from the perspective of finger searching in T. Suppose we are searching from f for a key $\sigma_i \geq f$ and z happens to be some key in the semi-open interval $(f,\sigma_i]$ inside the key space of T. Observe that this setting implies

$$\mathrm{dist}_T(f,\sigma_i) \geq \mathrm{dist}_T(f,z). \tag{2.1}$$

Furthermore, by Theorem 2.6, we also have

$$\mathrm{dist}_T(f,z) \geq 2^{\mathrm{hh}_f(z)-1}. \tag{2.2}$$

Combining the two above yields

$$\mathrm{dist}_T(f,\sigma_i) \geq 2^{\mathrm{hh}_f(z)-1}, \tag{2.3}$$

or in the asymptotic notation

$$\lg \mathrm{dist}_T(f,\sigma_i) = \Omega(\mathrm{hh}_f(z)). \tag{2.4}$$

In other words, Theorem 2.6 basically says that $\mathrm{hh}_f(z)$ is a lowerbound on the dynamic finger budget of finger searching from f to σ_i. As simple as it is, this is arguably the *most important* concept in the whole chapter.

2.1.3 Heterogeneous Spines

Let T continue to denote a complete binary search tree and let f be one of its keys. We will further define the notion of heterogeneous spines of T with respect to f.

2.1.3.1 Definition

Let (L,f,R) be the heterogeneous decomposition of T with respect to one of its keys f. Each of L and R has its corresponding heterogeneous spine. By symmetry, we will define the right heterogeneous spine only.

The right heterogeneous spine is simply R with each of its subtree T' replaced by a list of keys on the inner spine of T'. Let $\{x;T'\}$ with type $\{\alpha;\tau\}$ be a sublist of R in which x can be the f sentinel. We replace T' by a sorted list of the innermost key of the topmost $(\mathrm{ah}(x) - \mathrm{hh}_f(x) - 1)$ nodes on the inner spine of T'. In general, this list corresponds to a prefix of

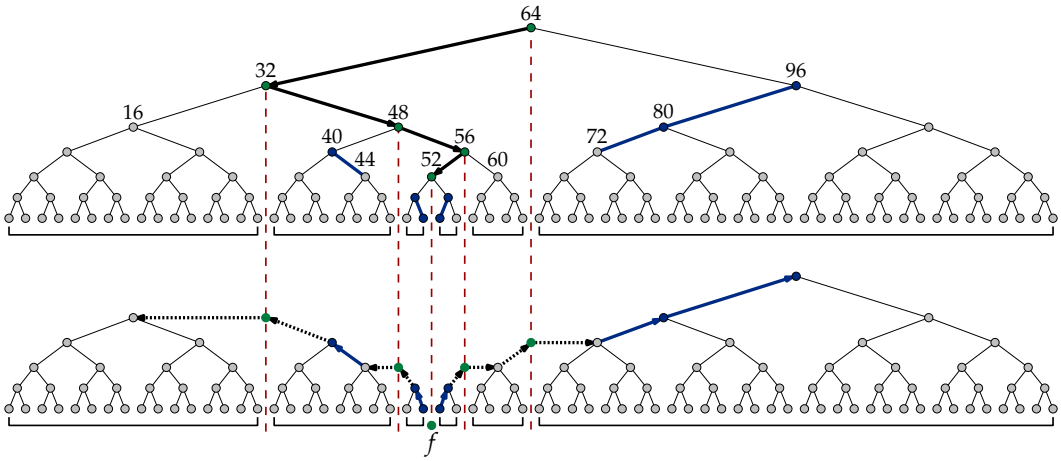


Figure 2.4 – Heterogeneous spines of a complete binary search tree of 127 keys with respect to the key f at rank 52 shown as two overlay paths (see Example 2.3)

Example 2.3. *Heterogeneous Spines*.

Refer to Figure 2.4. Assuming the set of keys in the reference tree is $\{1,2,\ldots,127\}$, the left and right heterogeneous spines of the tree with respect to f are

$$\{\{\};32;\{40\};48;\{50;51\}\} \text{ and } \{\{53;54\};56;\{\};64;\{80;96\}\}$$

respectively. Note that the keys 32, 48, 56, and 64 are primary, and the remaining keys inside the nested lists are secondary.

The figure is based on Figure 2.2 and we have additionally colored the secondary keys as well as their inner child edges blue. Furthermore, we have added an overlay path in the lower part. First, each inner child edge of a secondary key has been turned into an arrow pointing away from f. Then, for each primary key x, we also add two dashed arrows pointing away from f. Let u be the node containing x. The first is an rising arrow that points to x from the root of inner subtree of u. The second is an level arrow that points from x to a node at height $\mathrm{hh}_f(x)$ on the inner spine of the outer subtree of u. Notice that the arrows appear to connect into two paths emanating from f. This will be explained in §2.3.

the inner spine of T'. This spine prefix is said to be "owned" by x and is simply "the spine prefix of x".

Primary vs. Secondary. We make the following distinction among the keys that appear inside the right heterogeneous spine. The keys that were retained from R will be classified as primary, whereas the keys that appear inside the nested lists which were used to replace the subtrees in R will be classified as secondary. Note that the former are the right ancestor keys of

f. In addition, when the $-\infty$, f, and ∞ sentinels are needed, they will all be classified as primary.

2.1.3.2 Analysis

Theorem 2.8 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a complete binary search tree T with respect to one of its keys f. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then the list of secondary keys representing T' in the right heterogeneous spine of T with respect to f has length $(\mathrm{hh}_f(z) - \mathrm{hh}_f(x) - 1)$. Note that x and z may be sentinels.

Proof. By definition, T' is represented by a list of $(\mathrm{ah}(x) - \mathrm{hh}_f(x) - 1)$ secondary keys. By Theorem 2.5, $\mathrm{ah}(x) = \mathrm{hh}_f(z)$ and the theorem follows. □

Theorem 2.9 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a complete binary search tree T with respect to one of its keys f. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then the deepest and highest keys in the list of secondary keys representing T' in the right heterogeneous spine of T with respect to f have height $(\mathrm{hh}_f(x) + 1)$ and $(\mathrm{hh}_f(z) - 1)$ respectively. Note that x and z may be sentinels and the list of secondary keys may be empty.

Proof. Let u^* be the node containing x in T. By Theorem 2.2, T' is the right subtree of u^*. Since $\mathrm{ah}(T') = (\mathrm{ah}(x) - 1)$, including the topmost $(\mathrm{ah}(x) - \mathrm{hh}_f(x) - 1)$ keys on the inner spine of T' is equivalent to excluding its bottommost $\mathrm{hh}_f(x)$ keys. Therefore, among the keys that have been included on the spine, the deepest key has height $(\mathrm{hh}_f(x) + 1)$. For the highest key, by definition it has height $\mathrm{ah}(T')$ and $\mathrm{hh}_f(z)$ is precisely defined to be $(\mathrm{ah}(T') + 1)$. Note that this means the heterogeneous heights of f and all primary keys as well as the heights of all secondary keys are unique. □

Remark 2.10. Observe that Theorem 2.9 implies that the *only* subtree in R that has its entire inner spine represented is the innermost subtree in R. This is because it is the only subtree whose inner key has a heterogeneous height of 0, meaning that the deepest key in the list of secondary keys representing this subtree in R has height 1. Furthermore, had we defined $\mathrm{hh}_f(x)$ to be one less than its current value, we will lose the uniqueness property implied by Theorem 2.9 unless some (rather unpleasant) measures are taken.

Usage. From now on, whenever we consider a heterogeneous decomposition (L, f, R), we will be representing L and R in the format of heterogeneous spines and call them as such. Every subtree in L and R will be

represented by a suitably-long list of secondary keys on its inner spine and we will call this list the inner spine representation of the subtree.

2.2 For Degree-Balanced Search Trees

Generalizing our definitions in §2.1 to handle degree-balanced search trees requires relatively little effort because we already have both the conceptual and the notational frameworks set up. Specifically, from the interpretation of Theorem 2.6, what we are after is a set of definitions that will give rise to a version of this theorem for degree-balanced search trees. This means the heterogeneous height of a key should be a lowerbound on the dynamic finger budget to search for that key or any key in its outer direction. As it will turn out, our generalized notation of left/right introduced in §1.1.1 will make the theorems in this section (§2.2) look highly similar to their special cases in §2.1. Note that while some of the writings in this section may start like a rehash of those in §2.1, they are repeated here only for the sake of self-containment. In what follows, any degree-balanced search tree is an (a, b)-tree for any legitimate values of a and b.

2.2.1 Heterogeneous Decomposition

Given a degree-balanced search tree T and one of its keys f, the heterogeneous decomposition of T with respect to f is a triple (L, f, R), where L and R are lists of odd lengths comprising two alternating types of values. Appearing in the same order as they do in T, the odd positions of L/R are the left/right subtrees hanging off the access path of f and the even positions of L/R are the left/right ancestor keys of f. Underneath the above description lies one particular definition of "hanging off" since the nodes in T are multiway. To define this properly, let us start with an algorithm in §2.2.1.1 that computes the decomposition.

2.2.1.1 Definition

Given the root u of a degree-balanced search tree T and one of its keys f, the procedure Hetero-Decomp-Dbst in Figure 2.5 returns the triple that represents the heterogeneous decomposition of T with respect to f. The possibility of having multiple keys inside a node makes our procedure a bit more complicated than the corresponding procedure Hetero-Decomp-Cbst in §2.1.1.1. However, similar to Hetero-Decomp-Cbst, this procedure is based on a recursive search for f starting from $root[T]$ while keeping track of the subtrees hanging off the access path of f. As we have hinted, the question is what makes a desirable definition of "hanging off".

```
HETERO-DECOMP-DBST(u, f)
1  (i, j) ← BRACKET(u, f)
2  if i = j
3     then return ({left_i[u]}, key_j[u], {right_j[u]})          ▷ f = key_j[u]
4     else (L', ·, R') ← HETERO-DECOMP-DBST(c_j[u], f)           ▷ c_j[u] subtends f
5          if i ≥ 1
6             then L' ← CONCAT({left_i[u]; key_i[u]}, L')        ▷ c_j[u] not leftmost
7          if j ≠ b and key_j[u] ≠ NIL
8             then R' ← CONCAT(R', {key_j[u]; right_j[u]})       ▷ c_j[u] not rightmost
9          return (L', f, R')
```

Figure 2.5 – Procedure HETERO-DECOMP-DBST

```
BRACKET(u, f)
1  (i, j) ← (0, 1)
2  while key_j[u] < f
3        do (i, j) ← (i + 1, j + 1)
   (POST: key_i[u] < f ≤ key_j[u])
4  if f = key_j[u]
5     then return (j, j)          ▷ case (i)
6     else return (i, j)          ▷ case (ii)
```

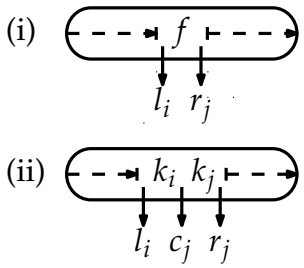


Figure 2.6 – Procedure BRACKET

For the moment, let us first understand what HETERO-DECOMP-DBST actually does. It starts off with the procedure BRACKET in Figure 2.6 which satisfies the following contract. Given a node u and a key f that may or may not be in u, BRACKET(u, f) returns a tuple (i, j) such that (i) if f is in u, then $i = j$ and $key_j[u] = f$; (ii) otherwise, $j = (i + 1)$ and $c_j[u]$ subtends f, meaning that f is in the subtree rooted at $c_j[u]$. Note that in the second case, $key_i[u]/key_j[u]$ is a left/right ancestor key of f. Furthermore, in both cases, the subtrees rooted at $left_i[u]$ and $right_j[u]$ do *not* contain f.

Back to HETERO-DECOMP-DBST. If f happens to be in u, then we will return the triple $(\{left_i[u]\}, key_j[u], \{right_j[u]\})$ in line 3. Since $key_j[u]$ is f and $i = j$, that means we are using L and R to keep track of *only* the two subtrees that are adjacent to f in u. But what about the other subtrees of u? In fact, the same question must also be asked in line 6 and line 8, where we seem to be dropping some subtrees of u altogether...

In our lingo, any maximal subtree of T that does not appear in the heterogeneous decomposition of T with respect to f is said to be "invisible from f", or "invisible" for short. An example that illustrates this visibility

issue will be shown in §2.2.2 where we define the heterogeneous heights. We say a subtree is hanging off the access path of f iff it is rooted at (i) a left or right child of f, or (ii) a left/right child of a left/right ancestor key of f. Using the contract of BRACKET(u, f), we see that these are precisely $left_i[u]$ and $right_j[u]$ in both cases.

2.2.1.2 Analysis

Theorem 2.11 Given the root u of a degree-balanced search tree T and one of its keys f, the procedure HETERO-DECOMP-DBST(u, f) runs in $O(b \log_a |T|)$ time. Furthermore, the total size of the triple returned is $O(\log_a |T|)$.

Proof. The proof is equivalent to the proof of Theorem 2.1 but with the b factor in the running time solely due to the $O(b)$ time required by BRACKET in line 1. Note that in line 7, we could have computed $\#(u)$ in $O(b)$ time and test for $j \leq \#(u)$ instead; however, here we are making use of the left packing property of the keys array. Also, note that the size of the triple returned does *not* have this factor. □

Theorem 2.12 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a degree-balanced search tree T with respect to one of its keys f and let u^* be a node in T containing the key x as its j-th key. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then (i) x is either f or a right ancestor key of f, and (ii) T' is the j-th right subtree of u^*, and (iii) z is the right parent key of x. Note that x and z may be sentinels.

Proof. If x is f itself, then the first point is trivially true. Otherwise, observe that the test on line 7 and the contract of BRACKET guarantee that for $x = key_j[u]$ to be added to R' on line 8, f is in the left subtree of x and thus x is a right ancestor key of f.

To show the second point, we merely need to inspect line 3 or line 8 of HETERO-DECOMP-DBST. The former applies if x is the f sentinel; the latter if otherwise.

The proof of the last point is similar to the proof of Theorem 2.2 due the observation we used to prove the first point here. This is a generalization of observation (2) on page 67. □

Theorem 2.13 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a degree-balanced search tree T with respect to one of its keys f and let u^* be a node in T containing the key x as its j-th key. If x is either f or one of its right ancestor keys in T and z is the right ancestor key of x, then $\{x; right_j[x]; z\}$ is a sublist of R. Note that x and z may be sentinels.

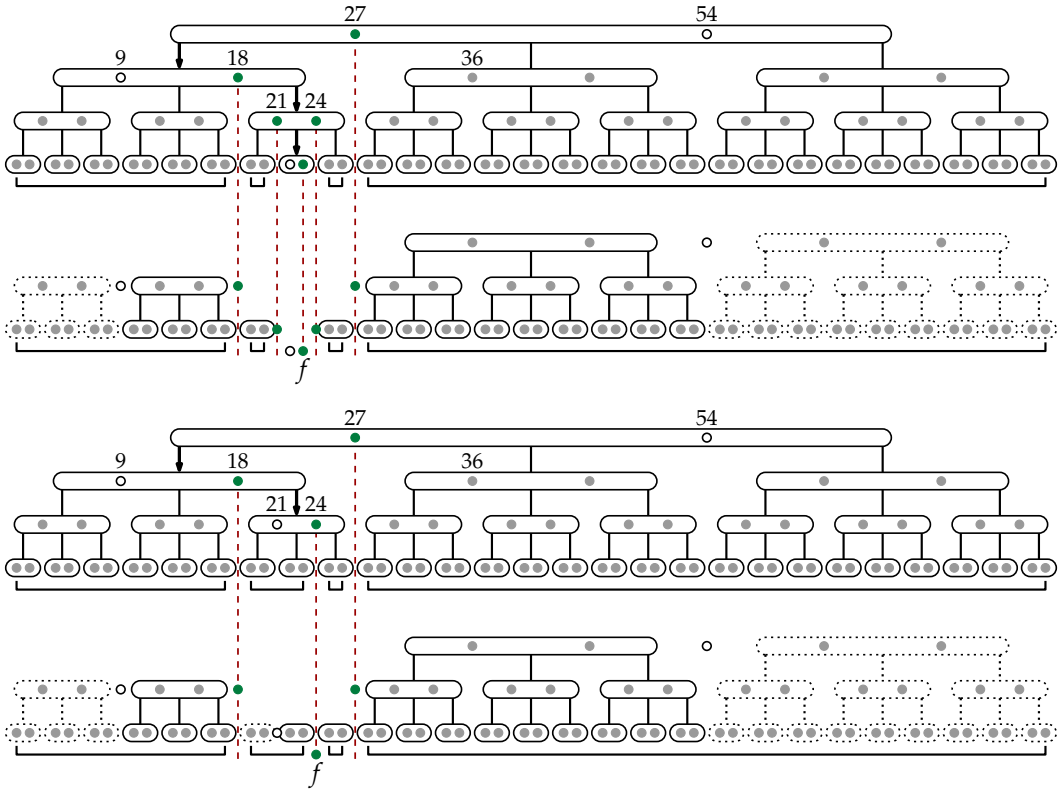


Figure 2.7 – Heterogeneous decompositions of a maximal $(2,3)$ degree-balanced search tree of 80 keys with respect to the key f at rank 23 (upper) and at rank 24 (lower), with the keys on the access path of f drawn at their respective heterogeneous heights (see Example 2.4 and Example 2.5)

Example 2.4. *Heterogeneous Decompositions*.

Refer to Figure 2.7. Assuming the set of keys in the reference tree is $\{1,2,\ldots,80\}$ and letting T_L^x and T_R^x denote the left and right subtrees of the key x respectively, the heterogeneous decompositions of the tree with respect to f in the two halves are respectively the triples

$$(\{T_L^{18};18;T_L^{21};21;T_L^{23}\},23,\{T_R^{23};24;T_R^{24};27;T_R^{27}\})$$

and

$$(\{T_L^{18};18;T_L^{24}\},24,\{T_R^{24};27;T_R^{27}\}).$$

The color scheme is based on the one described in Example 2.1 and in addition we have drawn each non-ancestor key on the access path of f as a hollow circle. Note that the nodes on the access path of f do not appear in the decomposition and their keys are exposed. We will explain the "invisible" subtrees that are drawn with a dashed stroke later in Example 2.5.

Proof. The proof is similar to the one for Theorem 2.3 except that a recursive call to HETERO-DECOMP-DBST can be considered as both a left *and* a right descend simultaneously here. This is because a single node in T can contain both a left and a right parent key of f. □

2.2.2 Heterogeneous Height

Let T continue to denote a degree-balanced search tree and let f be one of its keys. A notion of heterogeneous height with respect to f is associated with each key on the access path of f; any other key in T does not have a heterogeneous height. Note that since T is a degree-balanced search tree, the keys on the access path of f can be divided into three groups—the key f itself, the ancestor keys of f, and the non-ancestor keys of f on this path.

2.2.2.1 Definition

The definition of heterogeneous heights for degree-balanced search trees remains the same as the one for complete binary search trees in §2.1.2. However, we must additionally define the heterogeneous height of a non-ancestor key on the access path of f. For any such key y, we define $\mathrm{hh}_f^T(y)$ to be $(\mathrm{ah}(y) - 1)$. Note that in terms of support, this is equivalent to saying that y is supported by a subtree of height $(\mathrm{ah}(y) - 2)$.

2.2.2.2 Analysis

It may seem strange that the heterogeneous height of a non-ancestor key on the access path of f is defined without considering its position in its node relative to f. Say if f is in the leftmost subtree of a multiway node u and y is a non-ancestor key of f in u, then $\mathrm{dist}_T(f, y)$ would get larger if y is closer to the right end of u. However, as the following theorems and Example 2.5 will show, this definition is again tight from the perspective of lowerbounding the dynamic finger budget.

Theorem 2.14 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a degree-balanced search tree T with respect to one of its keys f. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then $\mathrm{hh}_f(z) = \mathrm{ah}(x)$. Furthermore, if $y > x$ is in the same node as x, then $\mathrm{hh}_f(y) = \mathrm{ah}(T')$. Note that x and z may be sentinels.

Proof. The proof here is mostly similar to the proof of Theorem 2.5. Let u^* be the node containing x in T and let x be the j-th key of u^*. By Theorem 2.12, T' is the j-th right subtree of u^*. The first part of the theorem follows because $\mathrm{hh}_f(z)$ is defined to be $(1 + \mathrm{ah}(T'))$ and this value is $\mathrm{ah}(x)$. To show that $\mathrm{hh}_f(y) = \mathrm{ah}(T')$, we merely need to note that T' is the j-th

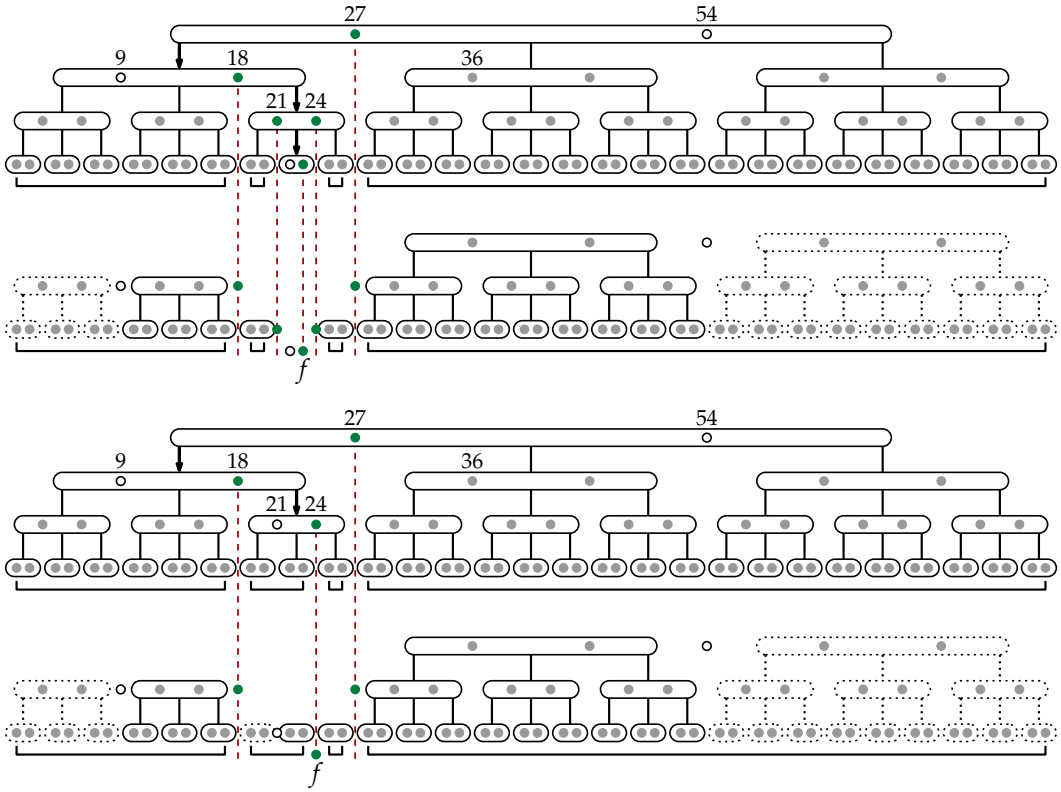


Figure 2.8 – Duplicate of Figure 2.7 on page 79

Example 2.5. *Heterogeneous Heights*.

Figure 2.8 is a duplicate of Figure 2.7, which shows the heterogeneous decompositions of a maximal (2,3) degree-balanced search tree of 80 keys with respect to the key f at rank 23 and at rank 24 in its upper and lower halves. Using the definition above, the heterogeneous heights of the keys at rank 9, 18, 21, 22, 23, 24, 27 and 54 in the upper half are 2, 2, 1, 0, 0, 1, 2 and 3 respectively. For the lower half, only the keys at rank 9, 18, 21, 24, 27, and 54 have their heterogeneous heights defined and they are 2, 2, 1, 0, 2, and 3 respectively.

The reason why some subtrees are drawn in a dashed stroke and some keys are drawn as hollow circles is because they are invisible from f in the decomposition. Let T_L^x denote the left subtree of the key x. A perfect example here is T_L^{21}. When f is 23, T_L^{21} is visible because 21 is a left ancestor key of f. However, when f is 24, the key 21 ceases to be a left ancestor key of f any more and our definition of heterogeneous height puts this key at height 1. Notice that both 21 and T_L^{21} are completely hidden by T_L^{24} in the sense that they do not have any part that is higher than T_L^{24}. Being "invisible from f" seems to be a suitable description here.

right subtree of u^*, meaning that $\mathrm{ah}(T') = \mathrm{ah}(u) - 1 = \mathrm{ah}(y) - 1$. The last value is precisely our definition of $\mathrm{hh}_f(y)$. □

Theorem 2.15 Let (L, f, R) be the heterogeneous decomposition of a degree-balanced search tree T with respect to one of its keys f and let a be the degree lowerbound of any non-root node in T. Consider any key z in L or R. If $\mathrm{hh}_f(z)$ is h, then $\mathrm{dist}_T(f, z) \geq a^{h-1}$. Furthermore, consider any non-ancestor key y on the access path of f. If $\mathrm{hh}_f(y)$ is h, then $\mathrm{dist}_T(f, y) \geq a^h$. In the asymptotic notation, $\log_a \mathrm{dist}_T(f, z) = \Omega(\mathrm{hh}_f(z))$ and $\log_a \mathrm{dist}_T(f, y) = \Omega(\mathrm{hh}_f(y))$.

Proof. First let us consider z. By definition, for z to attain a heterogeneous height of h, the inner subtree T' of z in the decomposition must have height $(h-1)$. Since T' is a *strict* subtree of a degree-balanced search tree with a degree lowerbound of a on any non-root node, T' has at least $(a^{h-1} - 1)$ keys. The theorem follows in this case because $\mathrm{dist}_T(f, z) \geq 1 + |T'|$.

For y, by definition it is not in L and R. By symmetry, suppose y is in the same node as the key $x < y$ and x is either f itself or a right ancestor key of f. Furthermore, suppose $\{x; T''\}$ of type $\{\alpha; \tau\}$ is a sublist of R, meaning x may be regarded as the f sentinel. By Theorem 2.12, T'' is the right subtree of x, which implies that T'' is to the left of y. By Theorem 2.14, $h = \mathrm{ah}(T'')$. Using a calculation similar to the above, T'' has at least $(a^h - 1)$ keys. The theorem follows in this case because $\mathrm{dist}_T(f, y) \geq 1 + |T''|$. □

Remark 2.16. Note that Theorem 2.15 does not allow z to be the f sentinel and the inequality $\mathrm{dist}_T(f, z) \geq a^{h-1}$ is in fact tight. Of course, the example in Remark 2.7 suffices to show this when $a = 2$, but this is true *even if* $a > 2$. This can be witnessed when T has height at least 2, f has rank $(a-1)$, and z has rank a, which implies that $h = 1$ and $\mathrm{dist}_T(f, z) = 1$. The inequality $\mathrm{dist}_T(f, y) \geq a^h$ is also tight. This can be witnessed when f has rank 1 and y has rank 2, which implies that $h = 0$ and $\mathrm{dist}_T(f, y) = 1$. Note that for y to exist, a must be at least 3 and thus f and y are in fact in the same node.

2.2.3 Heterogeneous Spines

Let T continue to denote a degree-balanced search tree and let f be one of its keys. We will further define the notion of heterogeneous spines of T with respect to f.

2.2.3.1 Definition

Let (L, f, R) be the heterogeneous decomposition of T with respect to one of its keys f. Each of L and R has its corresponding heterogeneous spine.

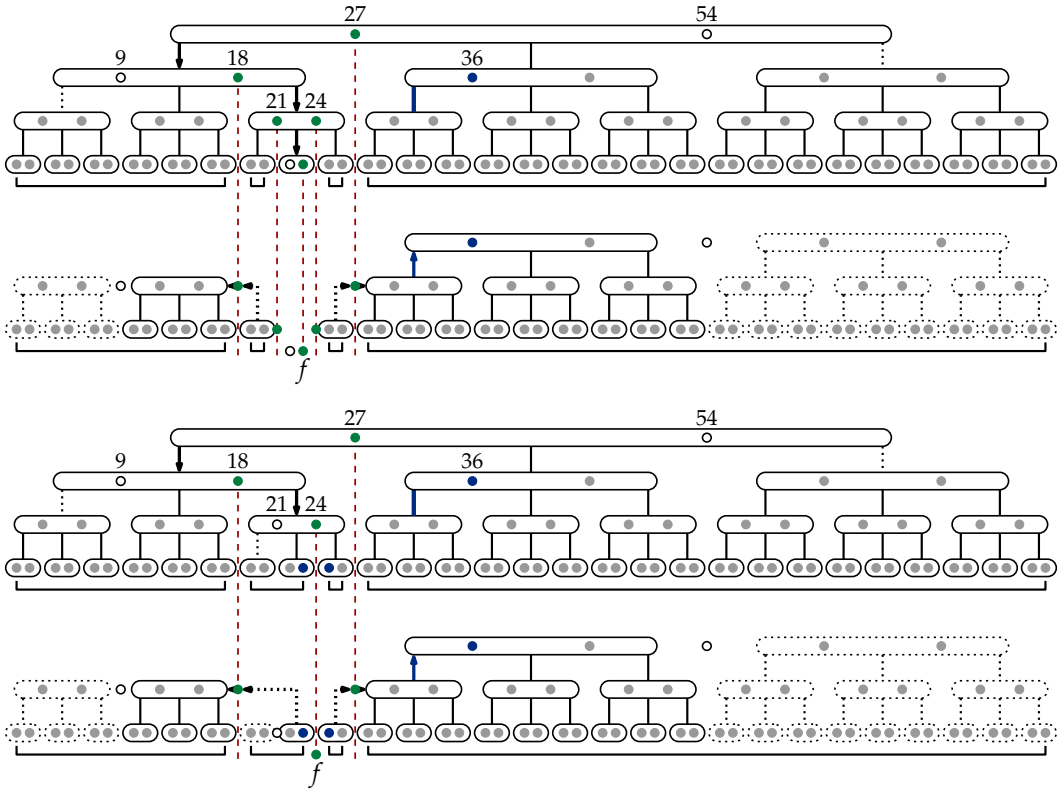


Figure 2.9 – Heterogeneous spines of a maximal $(2,3)$ degree-balanced search tree of 80 keys with respect to the key f at rank 23 (upper) and at rank 24 (lower) shown as two overlay paths (see Example 2.6)

Example 2.6. *Heterogeneous Spines*.

Refer to Figure 2.9. Assuming the set of keys in the reference tree is $\{1,2,\ldots,80\}$, the left and right heterogeneous spines of the tree with respect to f in the upper and the lower halves are respectively

$$\{\{\};18;\{\};21;\{\}\} \text{ and } \{\{\};24;\{\};27;\{36\}\}$$

and

$$\{\{\};18;\{23\}\} \text{ and } \{\{25\};27;\{36\}\}.$$

The figure is based on Figure 2.7 and we have modified it exactly the same way as we derived Figure 2.4 from Figure 2.2. Additionally, the non-ancestor keys on the access path of f has been drawn as hollow circles and the subtrees that are invisible from f as well as the edges leading to them in the reference tree have been drawn in a dashed stroke.

The definition here is exactly the same as the one in §2.1.3.1. Note that each node on a spine prefix has been defined to be represented by its *innermost* key. This specification is redundant for complete binary search trees, but it is critical for degree-balanced search trees.

2.2.3.2 Analysis

The theorems below are verbatim copy of the ones in §2.1.3.2 except that T is now a degree-balanced search tree. As this change does not affect our definition in §2.2.3.1, the proofs are exactly the same and thus omitted.

Theorem 2.17 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a degree-balanced search tree T with respect to one of its keys f. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then the list of secondary keys representing T' in the right heterogeneous spine of T with respect to f has length $(\mathrm{hh}_f(z) - \mathrm{hh}_f(x) - 1)$. Note that x and z may be sentinels.

Theorem 2.18 ($\mathcal{R}$) Let (L, f, R) be the heterogeneous decomposition of a degree-balanced search tree T with respect to one of its keys f. If $\{x; T'; z\}$ of type $\{\alpha; \tau; \alpha\}$ is a sublist of R, then the deepest and highest keys in the list of secondary keys representing T' in the right heterogeneous spine of T with respect to f have height $(\mathrm{hh}_f(x) + 1)$ and $(\mathrm{hh}_f(z) - 1)$ respectively. Note that x and z may be sentinels and the list of secondary keys may be empty.

2.3 Connecting to Heterogeneous Finger Search Trees

Before we go on, let us highlight a connection between the heterogeneous decompositions of a complete binary search tree T and the triplet representations of T as defined in §1.3.2. Let us stress that T is *not* a degree-balanced search tree.

Recall that if we want to implement the semantics of a dynamic finger on T, we can first decompose T into its triplet representation (T_L, f, T_R) by splitting T at one of its keys f to obtain T_L and T_R. Then we invert the two spines of both T_L and T_R and turn them into heterogeneous finger search trees. Through repeated splits and joins, this triplet can be maintained such that f always corresponds to the key under the dynamic finger.

2.3.1 A Canonical Transformation

Let (L, f, R) be the heterogeneous decomposition of T with respect to one of its keys f. Using the Split procedure for $(2,3)$-trees, let T_L and T_R be the pair of $(2,3)$-trees we obtained by splitting T at f. We claim that T_L / T_R can be obtained from L / R using the following ($\mathcal{R}$) procedure.

⟪ transform heterogeneous spine R into left spine of $(2,3)$-tree T_R ⟫

1> If R currently has length one, then stop.

2> Otherwise, let $\{T'_1; x; T'_2\}$ with type $\{\tau; \alpha; \tau\}$ be the innermost sublist of R and let w be the node at height $\mathrm{hh}_f(x)$ on the inner spine of T'_2.

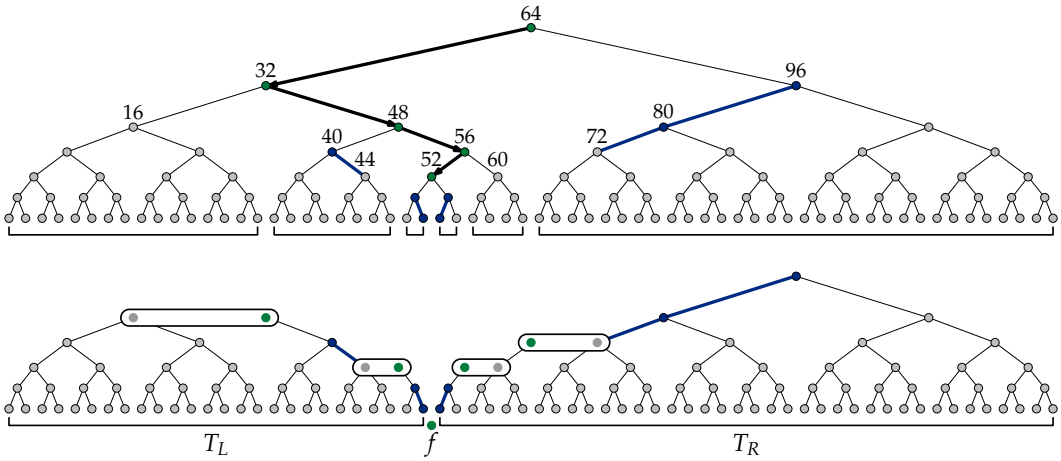


Figure 2.10 – Between the heterogeneous decomposition of a complete binary search tree of 127 keys with respect to the key f at rank 52 and the triplet representation of the tree as (T_L, f, T_R) (see §2.3.1)

Observe that by Theorem 2.9, w is the innermost child of the node containing the deepest secondary key that represents T'_2 in R.

3> Remove T'_1 and x from R, making T'_2 the innermost value of R.

4> Construct a 3-node u^* containing the keys x followed by $key[w]$ and from left to right attach T'_1, the left subtree of w, and the right subtree of w to u^*. Note that w must be a 2-node since it resides in T'_2 and this procedure has not operated on T'_2 yet.

5> Modify T'_2 by replacing w with u^* and replace the list of secondary keys representing T'_2 in R using $\textsc{Concat}(T'_1, \textsc{Concat}(\{(u^*, 1)\}, T'_2))$. Note that T'_1 and T'_2 inside the $\textsc{Concat}$ procedure are in their inner spine representations. Furthermore, note that the catenations do not affect the applicability of Theorem 2.9 on, if present, the outer key of T'_2 in R and its outer subtree.

6> Go back to step 1.

When the above procedure ends, only one value will remain in R and it will be the inner spine representation of the (now modified) T'_2 in step 5. Let T'_R denote this modified T'_2. It is straightforward to verify that T'_R is a valid $(2,3)$-tree and that the definition of heterogeneous heights puts each of the 3-nodes constructed in step 4 at exactly the same location as $\textsc{Split}$ would have. In other words, T'_R is exactly the same as T_R.

Furthermore, since the key pointed by $(u^*, 1)$ in step 5 is in fact the primary key x which we removed in step 3, the final value in R is simply a catenation of the original values in R. As this final value represents the entire inner spine of T'_R due to the observations in Remark 2.10 and step 2, we see that the original R is in fact a representation of the left spine of T_R.

Observe that whether we invert the spines or not does not matter as long as R is understood as a list in the same direction as the spine. In this sense, the heterogeneous decomposition of T with respect to f is also a *decomposition* of the triplet (T_L, f, T_R) where T_L and T_R are decomposed on their inner spines in a particular way specified by our definitions. We offer Figure 2.10 for a quick visualization of this fact. We remark that this is only one of the several reasonable decompositions of the triplet.

2.3.2 The Issue of Stability

It is important to observe that even though the heterogeneous spines of a complete binary search tree T can be transformed into the inner spines of a corresponding triplet, this does not mean they are equivalent objects. More precisely, once a finger search has been performed using the triplet, in general transforming the updated heterogeneous spines using the above procedure is *not* going to yield the inner spines of the updated triplet. This is because the heterogeneous decomposition with respect to a fixed key is *unique*, which implies that the above transformation always produces a pair of $(2,3)$-trees in a fixed shape that we will call their canonical shape. But as we will explain below, if we restrict the two trees in the triplet representation of T to their canonical shape, then the triplet can no longer be used to implement the semantics of a dynamic finger.

A Counterexample. Consider two heterogeneous decompositions of T, one of which is taken with respect to one of its keys f and the other is to another key x that is d keys further down in the key space of T. Using the uniqueness observation above, this would give rise to two triplets (T_L^f, f, T_R^f) and (T_L^x, x, T_R^x) in their respective canonical shapes. The questions are (i) how do the shapes of these two triplets relate to each other, and (ii) how does the difference between the two shapes relate to d?

Let T'_L and T'_R be respectively the left and right subtree of T. It turns out that both of these questions can be answered by inspecting a special case when f is at the root of T and d is 1, meaning that x is in the leftmost leaf of T'_R. While it is easy to see that T_L^f / T_R^f is just T'_L / T'_R in another name, the situation is more complicated when we get to T_L^x / T_R^x.

Let f^- be the predecessor of f in the key space of T and let the height of T be h. For $1 \le i \le (h-1)$, let $r(i)$ be the node at height i on the left spine of T'_R and let $T'_R(i)$ denote the right subtree of $r(i)$. Observe that T_L^x is T'_L with its rightmost leaf replaced by a 3-node that contains f^- and f. Therefore, the shapes of T_L^x and T_L^f are in fact quite similar. However, observe that the left spine of T_R^x is exclusively made up of 3-nodes, with the node at height

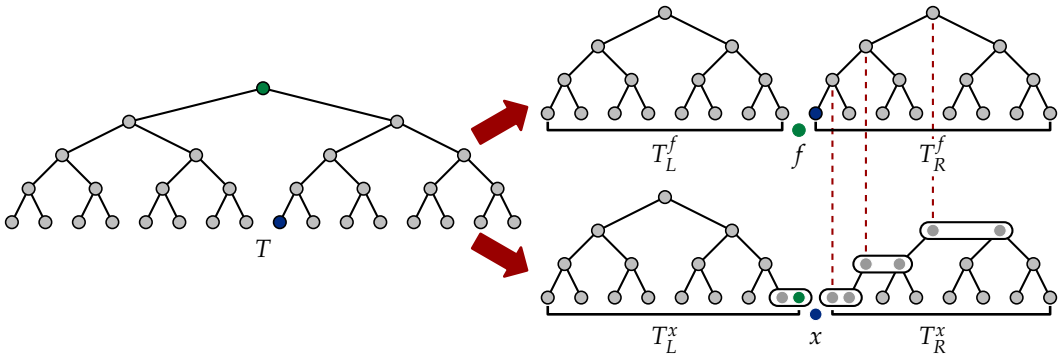


Figure 2.11 – Transforming a complete binary search tree T of height 5 into its triplet representations (T_L^f, f, T_R^f) and (T_L^x, x, T_R^x) where f is the key at the root of T and x is the successor of f

i being the 3-node that comes from combining $key[r(i+1)]$ with the root of $T'_R(i+1)$. This makes the shape of T_R^x very different from that of T_R^f as the latter is a binary tree. Figure 2.11 is an illustration of the above description when T is a complete binary search tree of height 5.

By now it should be obvious that the difference between the canonical shapes of (T_L^f, f, T_R^f) and (T_L^x, x, T_R^x) is *not* related to d in any straightforward sense. In fact, we have just seen an example where d is merely 1 but the "amount" of difference is proportional to $h = \Theta(\lg|T|)$. By going back and forth between f and x, this clearly shows that the triplet representation is not conducive to the dynamic finger property if we restrict the two trees to their canonical shape.

Robustness and Heterogeneous Finger Search Trees. Observe that if we compare the shapes of T_R^f and T_R^x, the situation is akin to repeatedly deleting and re-inserting the smallest key into a $(2,3)$-tree containing the keys of T_R^f. And because $(2,3)$-trees are not robust, we can be forced to incur an $\omega(1)$ cost in the example. This is precisely why a heterogeneous finger search tree has to be based on a balanced search tree that is robust. For example, Tarjan and Van Wyk [TVW88] used the robust red-black trees [GS78], whereas Tsakalidis [Tsa85] had to "robustify" AVL-trees [AVL62] by allowing a height imbalance of 2 on the inverted spine.

Given the discussion above, it may seem that using heterogeneous decompositions to reason about finger search would be a lost cause because this is equivalent to restricting the two trees in the triplet representation to their canonical shape. Fortunately, while the heterogeneous decomposition is not "robust", our representation of the corresponding heterogeneous spines is—this is true even if the reference tree is a complete binary search tree. As a preview, we can see that in our example every primary key in the heterogeneous spine corresponding to the left spine of T_R^x is already

present in the heterogeneous spine corresponding to the left spine of T_R^f, except that they are secondary keys in the latter. If we can transform these secondary keys into primary keys in an amount of time that is proportional to the dynamic finger budget, then perhaps the fragility issue can be circumvented. This is the very idea that we will rely on in §3.

Stability and Unique Representation. Although we will not get into the details, let us point out that this section actually touches on two highly-related issues in the design of dynamic data structures. The first is the issue of stability and the second is the issue of unique vs. redundant representations. Interested readers can start with the thesis of Acar [Aca05] and the thesis of Golovin [Gol08], which explore these two issues respectively.

2.4 Excision Arguments

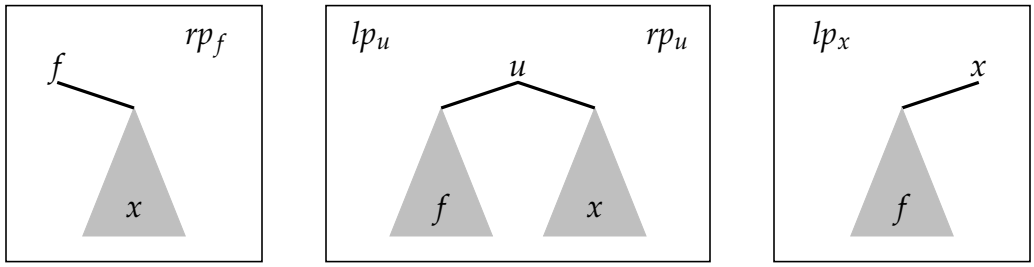
To see how the concept of heterogeneous decompositions can help us understand finger searching, let us present a particular way of thinking about it which we call an "excision argument". The reader is reminded that an excision refers to the splitting of an inner portion of a sorted list, with the region to be excised specified by two keys in the list. We have discussed this operation on page 41.

Consider a complete binary search tree T and two of its keys $f < x$. Let u be the lowest common ancestor of f and x, which means that f and x are both in $T|^u$. Suppose we take two heterogeneous decompositions of T, one at f which gives (L_f, f, R_f) and the other at x which gives (L_x, x, R_x). While we have focused on their worst-case differences in §2.3.2, here we are mostly interested in their similarity. We remark that Figure 2.12 depicts the relative locations of the keys in the discussion below.

2.4.1 Type U

Suppose f is in the left subtree of u and x is in the right subtree of u. Let lp_u and rp_u be the left and right parent keys of u, with lp_u being $-\infty$ if u is on the left spine of T and rp_u being ∞ if u is on the right spine. We claim that the two decompositions are identical to each other except in the region spanning from lp_u to rp_u, both inclusive.

To see this on the right hand side, observe that rp_u is on the access paths of both f and x and so it remains a right ancestor key in either cases. Because of this, any of the right ancestor keys of rp_u will stay as well. Therefore, any subtrees and right ancestor keys that appear on the right of rp_u in R_f and R_x are identical. A symmetric argument applies on the left to lp_u.



Type F: $[f, rp_f]$ Type U: $[lp_x, rp_x]$ Type X: $[lp_x, x]$

Figure 2.12 – The three types of excision arguments and their excision keys

From the above argument, we see that the heterogeneous height of any right ancestor key of rp_u will not change. However, the heterogeneous height of rp_u can. Say f is in the leftmost leaf of $T|^u$ and x is in its rightmost leaf, then the heterogeneous heights of rp_u with respect to these two keys are $\mathrm{ah}(u)$ and 0 respectively. The situation on the left is symmetric, with lp_u being the only key that can change its heterogeneous height.

2.4.2 Type F

Suppose f is in u and x is in the right subtree of f. Let rp_f be the right parent key of f, with rp_f being ∞ if f is on the right spine of T. We claim that the two decompositions are identical to each other except in the region spanning from f to rp_f, both inclusive.

On the right hand side, the situation is exactly the same as in a Type U excision; however, the left hand side is *subtly* different. In particular, f will take the place of lp_f in the argument, meaning that the left parent key of f will *not* change its heterogeneous height. In other words, the leftmost key that will change its heterogeneous height is f.

2.4.3 Type X

Suppose x is in u and f is in the left subtree of x. Let lp_x be the left parent key of x, with lp_x being $-\infty$ if x is on the left spine of T. We claim that the two decompositions are identical to each other except in the region spanning from lp_x to x, both inclusive. The argument here is symmetric to the one for Type F above.

2.4.4 Usage

What we have seen above are the three cases of an excision argument when we are finger searching from f to x. In Figure 2.12, note that f and x can be any key in the subtree represented by the shaded regions. Notice that in each of the three cases, we have identified the two excision keys and the

region bracketed by them is precisely the part of the heterogeneous decomposition that can see any change in the finger search. Most importantly, we can actually turn this into a procedure that transforms the heterogeneous decomposition taken at f (source) into the one taken at x (target). This procedure can be given as follows.

1> Find the lowest common ancestor of f and x to identify the type of the excision. This determines the two excision keys.
2> Remove the region between these two excision keys in the source decomposition and replace it with the corresponding region from the target decomposition.
3> Compute the new heterogeneous heights of the two excision keys and adjust their spine prefixes accordingly.

3

The Hands Revisited

In our previous work [BMW03], we have introduced a data structure called "the hands" that enables a degree-balanced search tree to support a constant number of dynamic fingers without storing extra pointers in its nodes nor restructuring after a finger search. Gratefully, the intuition we obtained during the development of this data structure eventually led us to the concept of heterogeneous decompositions, which then became the basis of this thesis.

In this chapter, we will revisit the hands and show how our newfound concept of heterogeneous decompositions clarifies the design and analysis of this data structure. Along the way, we will demonstrate that the hands can simply be viewed as an efficient worst-case implementation of the heterogeneous spines of the reference tree with respect to the key under a dynamic finger. This means that (i) the hands can be maintained in worst-case $O(\lg d)$ time when its corresponding dynamic finger is moved d ranks away, and (ii) the hands can be updated in real time when the reference tree is being restructured during an update.

Let us note that although the name of the data structure remains the same as in [BMW03], one of its invariants has been improved and leads to what we believe to be a simplier data structure. Also, even though "the hands" is apparently a plural form, it actually refers to a single data structure and thus we will continue to allow ourselves to refer to it as a singular.

3.1 Motivation

The hands was originally developed when we were investigating how to perform set operations such as intersection using sorted lists in a manner that is both optimal in time and compact in space. To understand how the hands is related to this problem, let us start with a review of the early history of optimal merging algorithms.

Optimal Merging Algorithms. Historically, the first *comparison*-optimal merging algorithm was due to Hwang and Lin [HL72], who were interested in how to merge two sorted lists that are stored on tapes. The number of comparsions made by their algorithm is $\left(\lceil \lg \binom{m+n}{n} \rceil + \min(m,n)\right)$, where $m \leq n$ are the lengths of the two input lists. Since their application requires the merged list to be written back onto an output tape, any merging algorithm must incur at least $\Theta(m+n)$ time. Notice that this time bound can asymptically absorb the second term in the above.

It is also not difficult to see that the remaining term, $\lceil \lg \binom{m+n}{n} \rceil$, is a lowerbound on any comparison-based merging algorithm. This is because any such algorithm must be able to distinguish among the $\binom{m+n}{n}$ possible interleavings of the elements in the two lists. However, because their application is tape-based, Hwang and Lin had no need to consider any efficient representation of the lists other than the linear representation. Unfortunately, it happens that if we are to naively implement their algorithm with arrays in a memory-based application, then the running time may asymptotically exceed the number of comparisons made. This means that there might still be room for improvement in this scenario.

Indeed, subsequently the merging problem in the memory-based setting has been studied by Brown and Tarjan [BT79], who proposed the first *time*-optimal merging algorithm that is based on the venerable AVL-trees [AVL62]. Their algorithm assumes that each of the two input lists is stored in a separate leaf-store AVL-tree in memory, and it inserts the items from the smaller tree into the larger tree in the sorted order so that the larger tree eventually becomes the merged list.[1] What Brown and Tarjan have shown is how to do these m insertions in $O\left(\lg \binom{m+n}{n}\right) = O\left(m \lg \frac{n}{m}\right)$ time, which makes their algorithm the first optimal merging algorithm in which the two input lists and the output list are represented with data structures of the same type.

1. Although it was not explicitly mentioned by Brown and Tarjan [BT79], we may even consider the output of their algorithm to be a *new* AVL-tree assuming that the technique of path-copying is used.

However, according to Brown and Tarjan themselves in [BT80, p. 613], the merging algorithm in [BT79] is "not obvious and the time bound requires an involved proof". As such, in a later joint work of theirs [BT80], Brown and Tarjan proposed yet another merging algorithm based on $(2,3)$-trees [AHU74] that are modified with the technique of level-linking. As we have demonstrated in §1.2.4.2, optimal merging can be performed very intuitively with level-linked degree-balanced search trees by implementing a simple dynamic finger *pointer* on the larger tree. But with the finger search algorithm analyzed in [BT80], which we have also covered in §1.2.3.2 when we described level-linking, the search tree must be in the leaf-store representation for the algorithm to be optimal.

Space Efficiency vs. Simplicity. Putting the representation issue aside, let us observe a curious mismatch in concepts underlying the approach taken by Brown and Tarjan in [BT80]. In particular, notice that what their merging algorithm needs is a *single* dynamic finger, but what it uses are homogeneous finger search trees, which are arguably designed for problems that require finger searching from any key.

Once we have phrased it in this way, it is suggestive that a more apt solution for the merging problem could be the heterogeneous finger search trees that were later introduced by Tarjan and Van Wyk [TVW88]. Indeed, when a pair of these trees is used in the triplet representation as described in §1.3.2, it is as if we have a dynamic finger on the search tree represented by the triplet. This immediately gives us a second time-optimal merging algorithm that is also based on the intuitive notion of finger search.

Furthermore, this approach has a distinct advantage in terms of space—whereas level-linking introduces a linear overhead to a degree-balanced search tree due to the parent and level pointers, as we have pointed out in §1.3.1, the number of pointers in a heterogeneous red-black tree is the same as its corresponding red-black tree. In fact, even when we have to use degree-balanced search trees as the basis, the overhead due to the extra number of nodes in a triplet representation is only logarithmic. This makes a truly compact solution to the optimal merging problem.

Unfortunately, heterogeneous finger search trees are not exactly as easy to implement as level-linked degree-balanced search trees and this might potentially make the above solution slightly less desirable. In fact, Booth—then a doctoral student of Tarjan—has noted in her thesis [Boo90, §2.3] that heterogeneous finger search trees "have a reputation for being complicated and impractical, hard to understand and hard to implement" (presumably because they are often used with augmentation). Regardless of whether

this reputation is justified or not when it comes to the merging problem, the required restructuring of the triplet after each finger search does have some nontrivial details that must be handled carefully.

Finger Search vs. Update. Although we can agree that such details only need to be taken care of once per implementation—and hence not being a significant concern, in our investigation we have also observed that the above solution may be improved at the conceptual level. To illustrate this, let us consider a finger search spanning a rank distance of d on a level-linked $(2,4)$-tree and a corresponding triplet based on heterogeneous red-black trees. While the former can be shown to run in worst-case $O(\lg d)$ time, we can only prove an equivalent but *amortized* bound with the latter. The amortization, however, has little to do with the finger searching itself—in both cases, we need to traverse only $\Theta(\lg d)$ pointers; instead, the worst-case guarantee is lost due to the restructuring of the triplet after a finger search, which in the worst case may propagate all the way to the root.

Our observation is that this restructuring is not related to an update of the triplet and it is performed purely to restore the red-black tree invariants after splits and joins. This is most evident if we observe that there are applications—intersection being a prime example—in which the set of items in the triplet remains the same throughout the entire sequence of finger searches. Together, these two observations suggest that maybe we should look for a solution such that if the time due to the restructuring in updates is discounted, then each finger search has a *worst-case* time guarantee. From a data structure design standpoint, this means we are trying to decouple the concern of finger searching from the concern of updates.

An Auxiliary Data Structure. With the decoupling idea in our mind, a careful scrutiny of the earlier merging algorithm by Brown and Tarjan in [BT79]—which does not rely on any pointers other than the ones already in an AVL-tree—would reveal an interesting property of the utilization of the parent and level pointers during a finger search in a level-linked (a,b) degree-balanced search tree. In particular, although these pointers are always present in the tree and thus occupying linear space, a finger search spanning a rank distance of d can only use $\Theta(\log_a d)$ of such pointers. Since initially we were interested in supporting only one dynamic finger, this gave us the idea of getting rid of the parent and level pointers altogether and instead simulating them using an auxiliary data structure that is maintained *outside* of the search tree. Our goal is to minimize the size of this data structure while making sure that (i) it can be used to compute every

parent or level pointer when its corresponding dynamic finger is used in a finger search, and (ii) it can be updated in real time when the search tree has to be restructured after an update.

As documented in [BMW03], our actual thought process started with how to scan a complete binary search tree in worst-case $O(1)$ time per key. Observe that in the absence of parent and level pointers, the choice between leaf-store and node-store makes little difference to our problem. In fact, a leaf-tree can be seen a special case of a node-tree in which only the keys at the leaves can be accessed. Our solution to the scanning problem happens to be a simple logarithmic-sized data structure, which can easily be symmetrized into a pair of inter-linked data structures that supports bidirectional stepping. At the time, it was thought that this pair of data structures is a set of related fingers and this explains why we dubbed it "the hands". But somewhat curiously, although we have designed the hands from the perspective of scanning, we discovered that the hands can actually support finger searching without any modification to its structure. Understanding exactly *why* the latter happens would lead us to extract the concept of heterogeneous decompositions from the hands. The rest, as they say, is history.

3.2 Overview

Using the concept of heterogeneous spines from §2, the definition of the hands is straightforward—it is simply a stack of stack implementation of the heterogeneous spines themselves. Since this cannot be simplified by much even if we restrict our attention to the special case of complete binary search trees, we will define the hands for degree-balanced search trees immediately in §3.3. The same is true for the logarithmic-time algorithm that builds the hands on any given key and we will be describe it in §3.4.

Unfortunately, we have not been able to finish rewriting the algorithms to manipulate the hands within the timeframe of this thesis. Instead, we have attached [BMW02] as §A to serve the purpose of describing these algorithms. The final version of this document will contain their complete descriptions.

Global Variables. Although global variables are considered harmful, let us make a few deliberated exceptions here. Within this chapter, let T be the underlying search tree and let f be one of its keys. We will specify whether T is a complete binary search tree or an (a, b) degree-balanced search tree in each section. Also, let (L, f, R) be the heterogeneous decom-

position of T with respect to f. Note that T is both the reference tree of the decomposition and the underlying tree of the hands in this chapter.

Catenable Stacks with Splits. All stacks in this chapter are a form of catenable stacks that support splits. Let us spell out our notation and the interface of such stacks here. As it will turn out, pointer loops are possible on the hands, meaning that the hands itself is inherently an ephemeral data structure; therefore, many of the operations in the interface below are stated to modify their inputs.

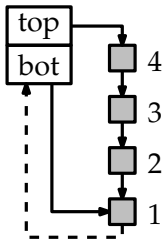
Notation. A catenable stack Q is made up of cells and each cell contains a value, which itself can be a tuple, say. The length of Q is the number of cells in Q and is denoted $|Q|$. The cells of Q are counted from the bottom of the stack and the i-th cell of Q is denoted $Q[i]$.

Interface. We require our catenable stacks to support each of the following operations in worst-case $O(1)$ time.

- ☞ CreateStack() returns an empty stack.
- ☞ IsEmpty(Q) tests if the stack Q is empty.
- ☞ Push(Q, v) pushes the value v into a new top cell of the stack Q.
- ☞ Pop(Q) pops the top cell of the stack Q and returns the value inside this cell. This operation assumes Q is nonempty.
- ☞ Top(Q) and Bot(Q) respectively return the address of the top and the bottom cell in the stack Q. They both assume Q is nonempty.
- ☞ Concat(Q_1, Q_2) puts the stack Q_1 on top of another stack Q_2. Assuming that Q_1 is not empty, this operation changes Q_2 and the new top cell of Q_2 is the top cell of Q_1. This operation destroys Q_1.
- ☞ Split(Q, p) splits the stack Q at one of its cells pointed by the pointer p and returns a new stack Q' spanning from the top cell of Q to the cell pointed by p. After the split, Q no longer contains the cells that have been split into Q'.

Remark 3.1. To make sure we understand the last two operations, notice that Split reverses the effect of Concat. In particular, suppose Q_1 and Q_2 are stacks and Q_1 is nonempty. If p is the address returned by Bot(Q_1), then Concat(Q_1, Q_2) followed by Split(Q_2, p) will return Q_1 as its result and restores Q_2 to its original state.

Possible Implementation. One possible pointer structure that implements the above interface is depicted in the figure on our side.



In the figure, a stack of length 4 is shown with its cells labelled with their positions in the stack. The data structure has a record that stores two pointers to the top and bottom cells of the stack if the stack is nonempty; otherwise, these two pointers are NIL. Each cell stores its value and a pointer to the next cell down the stack. The pointer in the bottom cell may be implemented to either point to the record or store NIL. (Hence this pointer is drawn in a dashed stroke.)

Bottom-Up Traversal. We note that in §3.4 we will describe an algorithm that traverses *up* a stack. Although this is not an operation supported by the above interface, we can simulate this traversal by inverting a stack Q into a temporary stack Q' using PUSH and POP; then we can traverse Q in the bottom-up manner as we invert Q' back into Q. The running time for a bottom-up traversal of Q is $\Theta(|Q|)$. The actual implementation of our algorithm using this simulation is straightforward. (Alternatively, we may use a doubly-linked list to implement a catenable stack. This is the approach taken in [BMW03]; however, it will not be as space-efficient.)

3.3 The Structure of the Hands

The hands on f in a degree-balanced search tree T consists of two parts, namely the left hand and the right hand. Intuitively, the left/right hand represents the left/right heterogeneous spine of T with respect to f, but there are also additional pointers going between the two parts. The hands is implemented as a pair of catenable stacks (lps, rps), each of which is used to organize a collection of other catenable stacks. Since the definitions of *lps* and *rps* are symmetric, we will only describe the structure of the latter. Our focus here are the names and types; the content of the right hand will be specified using the three invariants in §3.3.1.

(1) For $1 \le i \le |rps|$, the cell $rps[i]$ contains a triple of pointers. In order, they are the key pointer, the spine pointer, and the cross pointer of $rps[i]$.
(2) The key pointer of $rps[i]$ is denoted $key[rps[i]]$ and it points to a key in T. Suppose this key is contained in the node u in T and it is $key_j[u]$. We further let $node[rps[i]]$ to denote u and $pos[rps[i]]$ to denote j.
(3) The spine pointer of $rps[i]$ is denoted $spine[rps[i]]$ and it points to a catenable stack with exactly the same type as *rps* itself. This stack is called the spine prefix stack of $rps[i]$.
(4) The cross pointer of $rps[i]$ is denoted $cross[rps[i]]$ and it can either be NIL or point to another cell, which is called the cross cell of $rps[i]$.

3.3.1 Invariants

Let us specify the content of *rps* via the following three invariants. The first two can be verified to be consistent with our definition of heterogeneous spines in §2 and the third is the trick to an efficient catenable stacks implementation of the heterogeneous spines. In what follows, let ℓ be $|rps|$.

Invariant 3.2 (Primary $\mathcal{R}$) At the two ends, $key[rps[\ell]]$ points to f in T and $key[rps[1]]$ points to a key on the right spine of T. For $1 \le i \le (\ell - 1)$, $key[rps[i]]$ points to the right parent key of $key[rps[i+1]]$ in T.

Invariant 3.3 (Secondary $\mathcal{R}$) For $1 \le i \le \ell$, $spine[rps[i]]$ has length $(\mathrm{ah}(k_i) - \mathrm{hh}_f(k_i) - 1)$. For $1 \le j \le |spine[rps[i]]|$, $spine[rps[i]][j]$ contains the triple of pointers (k_j, s_j, x_j) with x_j being NIL, s_j pointing to an empty stack, and k_j pointing to the innermost key of the j-th node on the inner spine of the outer subtree of $key[rps[i]]$ in R.

Invariant 3.4 (Cross $\mathcal{R}$) The cross pointer $cross[rps[\ell]]$ is NIL. For $1 \le i \le (\ell - 1)$, if $key[rps[i]]$ is the trivial right parent key of $key[rps[i+1]]$, then $cross[rps[i]]$ is NIL; otherwise $cross[rps[i]]$ points to a cell in *lps* whose key pointer points to a key in the left child of $key[rps[i]]$ in T.

Sentinels. As it will be convenient, we will imagine a sentinel cell $rps[0]$ containing the triple $(k_0, s_0, x_0) = (\infty, \text{NIL}, \text{NIL})$ underneath the bottom of *rps*. Note that this is consistent with Invariant 3.2 because we have defined the right parent key of any key on the right spine of T to be ∞. Also, note that a corresponding sentinel using $-\infty$ will also be added to *lps* as $lps[0]$.

3.3.2 Analysis

Theorem 3.5 ($\mathcal{R}$) Suppose $|rps|$ is ℓ. For $1 \le i \le \ell$, the length of $spine[rps[i]]$ is $(\mathrm{hh}_f(key[rps[i-1]]) - \mathrm{hh}_f(key[rps[i]]) - 1)$ where k_0 is ∞.

Proof. By Invariant 3.3, the length of $spine[rps[i]]$ is

$$(\mathrm{ah}(key[rps[i]]) - \mathrm{hh}_f(key[rps[i]]) - 1).$$

Let j be $pos[rps[i]]$. If $i \ge 2$, then by Invariant 3.2 $key[rps[i-1]]$ is the right parent key of $key[rps[i]]$ in T. By Theorem 2.13, this means

$$\{key[rps[i]]; right_j[node[rps[i]]]; key[rps[i-1]]\}$$

is a sublist of R. By Theorem 2.17, this implies that $|spine[rps[i]]|$ is

$$(\mathrm{hh}_f(key[rps[i-1]]) - \mathrm{hh}_f(key[rps[i]]) - 1)$$

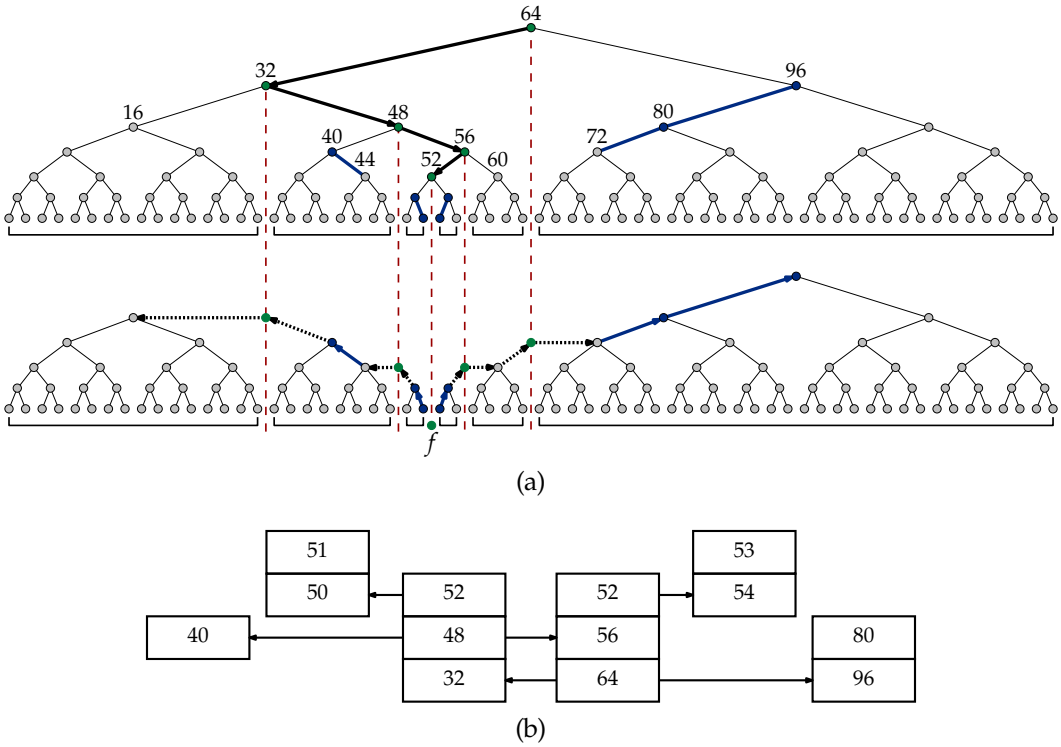


Figure 3.1 – (a): Duplicate of Figure 2.4 on page 74, which shows the heterogeneous spines of a complete binary search tree of 127 keys with respect to the key f at rank 52 shown as two overlay paths; (b): The corresponding hands on f

and the theorem follows. If $i = 1$, then by Theorem 2.14 with $x = key[rps[1]]$, we know $\mathrm{hh}_f(\infty) = \mathrm{ah}(x)$. Since the former is $\mathrm{hh}_f(key[rps[0]])$ and the latter is $\mathrm{ah}(key[rps[1]])$, the theorem also follows. □

Theorem 3.6 The number of cells in the hands on any key in a degree-balanced search tree of height h is $O(h)$.

Proof. By symmetry, let us only compute the total number of cells on the right hand side. Suppose $|rps|$ is ℓ. For $1 \le i \le \ell$, we have one cell due to rps_i and $(\mathrm{ah}(key[rps[i]]) - \mathrm{hh}_f(key[rps[i]]) - 1)$ cells due to $spine[rps[i]]$. Applying Theorem 3.5 to the latter, the total number of cells is therefore

$$\sum_{i=1}^{\ell} \big(1 + (\mathrm{hh}_f(key[rps[i-1]]) - \mathrm{hh}_f(key[rps[i]]) - 1)\big)$$
$$= \mathrm{hh}_f(key[rps[0]]) - \mathrm{hh}_f(key[rps[\ell]]). \tag{3.1}$$

Since $key[rps[\ell]]$ is f due to Invariant 3.2, we know that $\mathrm{hh}_f(key[rps[\ell]])$ is 0. Furthermore, since $key[rps[0]]$ is the ∞ sentinel in Theorem 3.5, we conclude $\mathrm{hh}_f(key[rps[0]]) = \mathrm{ah}(key[rps[1]]) = O(h)$. □

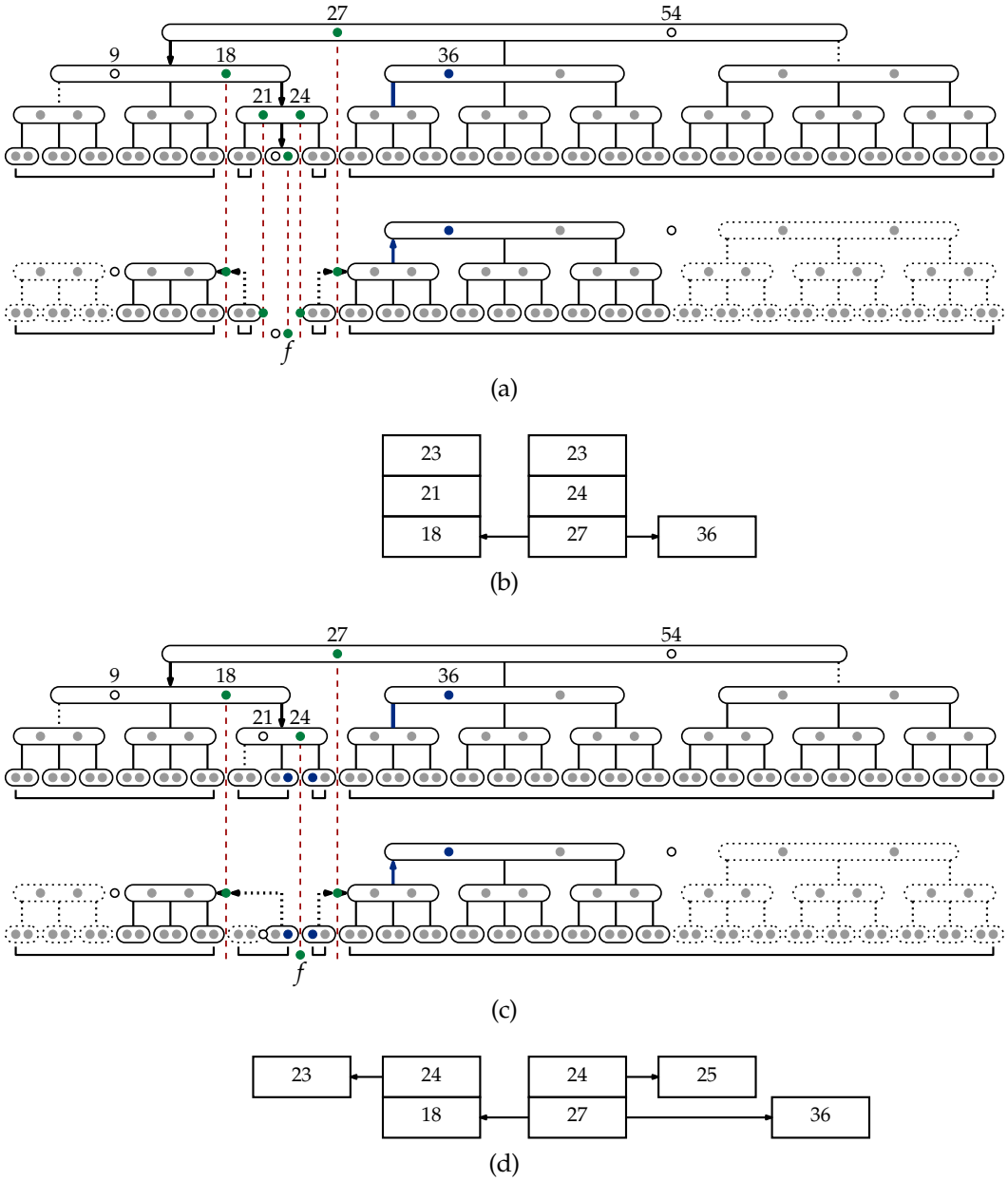


Figure 3.2 – (a): Duplicate of Figure 2.9 on page 83, which shows the heterogeneous spines of a maximal $(2,3)$ degree-balanced search tree of 80 keys with respect to the key f at rank 23 shown as two overlay paths; (b): The corresponding hands on f; (c) and (d): ditto when f is at rank 24

Theorem 3.7 If $node[lps[j]]$ and $node[rps[i]]$ both point to the same node u in T (but possibly to a different key of u), then no cross pointer in lps points to $rps[i]$ and no cross pointer in rps points to $lps[j]$.

Proof. We will prove this by contradiction. First, consider the case when a cross pointer in rps points $lps[j]$. Let the cell containing this cross pointer be $rps[i']$. We will break this down into three cases.

Case ($i' < i$): This condition implies that the cell $rps[i'+1]$ exists. By Invariant 3.4, $node[rps[i'+1]]$ is not a child of $node[rps[i']]$; also, $node[lps[j]] = u$ is the left child of $key[rps[i']]$. By Invariant 3.2, $key[rps[i']]$ is the right parent key of $key[rps[i'+1]]$ and thus $node[rps[i'+1]]$ is also in the left subtree of $key[rps[i']]$. Since $node[rps[i'+1]]$ is not a child of $node[rps[i']]$, $node[rps[i'+1]]$ cannot be u and thus $node[rps[i'+1]]$ must be deeper than u in T. However, u is also on rps due to $rps[i]$, which together with the condition $i' < i$ implies that $(i'+1) < i$. This in turn implies that u is deeper than $node[rps[i'+1]]$ by repeated applications of Invariant 3.2. The contradiction is in the relative depth of u with respect to $node[rps[i'+1]]$.

Case ($i' = i$): By Invariant 3.4, $cross[rps[i']] = cross[rps[i]]$ must point to a cell in lps that points to a child of $node[rps[i]]$. This means $node[lps[j]] = u$ is a child of $node[rps[i]]$. However, $node[rps[i]]$ is actually u itself from the condition of the theorem statement—contradiction.

Case ($i' > i$): By repeated applications of Invariant 3.2, $node[rps[i']]$ is deeper than u in T. By Invariant 3.4, $cross[rps[i']]$ must point to a cell in lps that points to a child of $node[rps[i']]$. But since $cross[rps[i']]$ points to $lps[j]$ and $node[lps[j]] = u$, it means u is deeper than $node[rps[i']]$ instead. The contradiction is the relative depth of u with respect to $node[rps[i']]$.

Finally, the analysis for the case when a cell in lps points $rps[i]$ is symmetric to the above. □

Theorem 3.8 The cross pointers in the hands do not cross each other, i.e., there do not exist positions $i < i'$ and $j < j'$ such that the $cross[lps[j]]$ points to $rps[i']$ and $cross[rps[i]]$ points to $lps[j']$.

Proof. Suppose by way of contradiction there exists a pair of cross pointers that cross each other as specified by the four locations in the statement of the theorem. By symmetry, let us assume that $node[rps[i]]$ is higher than $node[lps[j]]$ in T, i.e.,

$$\mathrm{ah}(node[rps[i]]) > \mathrm{ah}(node[lps[j]]). \tag{3.2}$$

By Invariant 3.4, $node[rps[i]]$ is the parent of $node[lps[j']]$, meaning that

$$\mathrm{ah}(node[rps[i]]) = \mathrm{ah}(node[lps[j']]) + 1. \tag{3.3}$$

Furthermore, by repeated applications of Invariant 3.2 ($\mathcal{L}$) and the condition that $j < j'$, we also know that

$$\mathrm{ah}(\mathit{node}[\mathit{lps}[j]]) > \mathrm{ah}(\mathit{node}[\mathit{lps}[j']]). \tag{3.4}$$

However, these three relations cannot simultaneously hold. □

3.4 The Ability to Build

The easiest way to build the hands is to do so in a number of elementary steps, each of which performs a single task that is easy to understand and easy to analyze. In what follows, we will give such an algorithm that builds the hands on f in a degree-balanced search tree T. This algorithm assumes it is given a node u in T such that f is in the subtree $T|^u$. We can therefore pass $\mathit{root}[T]$ as u to build the hands from scratch. In the general case when u is not $\mathit{root}[T]$, we say that the hands returned by the algorithm is "the hands on f restricted to u (or $T|^u$)".

Before we go on, let us remark that the algorithm below makes several passes when it builds the hands and these passes can actually be condensed into a *single* top-down pass. However, both the algorithm and its proof of correctness will be considerably more complex.

3.4.1 Algorithm

⟪ build the hands on f in $T|^u$ ⟫

- pass 1 -

1> Compute the height h of $T|^u$ by traversing down its left spine. Note that traversing this path is a correct way to compute h because T is a degree-balanced search tree. The running time of this step is $O(h)$.

- pass 2 -

2> Search for f starting from u while keeping track of (i) the current depth with respect to u, and (ii) the access path of f in a parent stack. To actually build the hands, the search procedure in §1.1.2 should be modified in two places.

2.1> During the descend, for each right parent key k of f from which we descend into its left subtree, push $(k, \textsc{Nil}, \textsc{Nil})$ into *rps* and remember its height in T. The latter of which can be deduced from h and the current depth.

2.2> When we reach f, push $(f, \textsc{Nil}, \textsc{Nil})$ into *rps* and also remember its height in T.

Note that in both places the first value of the triple pushed into *rps* is a key pointer and not the key itself. As the two modifications only

introduces an extra $O(1)$ amount of work in each recursive call, this step takes $O(b \cdot h)$ time. Also, note that the current *rps* satisfies Invariant 3.2 by construction.

- pass 3 -

3> Let ℓ be $|rps|$. Since we know the height of each key appearing in *rps*, we can apply Theorem 2.13 and then Theorem 2.14 to compute their heterogeneous heights, which give us the values of $|spine[rps[i]]|$ for $1 \le i \le \ell$. The running time of this step is $O(\ell)$, which is $O(h)$ by Theorem 3.6.

4> For $1 \le i \le \ell$, build $spine[rps[i]]$ incrementally by traversing down the right-left spine of $key[rps[i]]$ in T for $|spine[rps[i]]|$ nodes. For each visited node, push a triple comprising its innermost key, NIL, and NIL into $spine[rps[i]]$. This step takes $O(h)$ time by Theorem 3.6 and now *rps* satisfies Invariant 3.3.

- pass 4 and pass 5 -

5> Symmetrically build *lps* using steps 2 to 4. Note that step 4 takes $O(b \cdot h)$ time for *lps* because we need to compute $\#(u)$ for each visited node u in order to descend into its rightmost child. In any case, all three steps takes $O(b \cdot h)$ time in total. Note that so far all cross pointers in the cells of both *lps* and *rps* are NIL.

- pass 6 -

6> To set the cross pointers in *rps*, we initialize a pointer *rp* to the bottom cell of *rps* and another pointer *lp* to NIL, which is considered to be poining at the sentinel $lps[0]$. We will also initialize a pointer p to the bottom cell of the parent stack in step 2 and let $node[p]$ be the node pointed by the cell at p.

We will maintain an invariant that $node[rp]$ is a descendant of $node[lp]$. Observe that this invariant is true with the initial values of *lp* and *rp*. This is because $node[lps[0]]$ is the $-\infty$ sentinel of T, which we consider to be a parent of $root[T]$, and thus $node[rps[1]]$ is a descendant of it.

7> If $node[p]$ is not $node[rp]$, then advance p up the parent stack until this is true. This step will correctly finish since rp also points to a node on the access path of f.

8> Test if *rp* is pointing to the top cell of *rps*.

8.1> If so, then there is nothing left to do in this pass. This is because $cross[rps[\ell]]$ has already been initialized to NIL and this satisfies Invariant 3.4. Go to step 9.

8.2> Otherwise, suppose *rp* is pointing at $rps[i]$ for some $i \le (\ell - 1)$. From the test result, we know $rps[i+1]$ exists. Furthermore, the existence

of $rps[i+1]$ also implies that a cell exists above p in the parent stack. Let c^* be the node pointed by this cell. Note that c^* is a child of $node[rps[i]]$. Test if $node[rps[i+1]]$ is c^*.

8.2.1> If so, then the placement of $node[rps[i]]$ and c^* into rps in step 2 implies that we have descended left from both $node[rps[i]]$ and c^*. This in turn implies that $node[rps[i]]$ is a trivial parent and therefore $cross[rps[i]]$ should remain NIL. In other words, no work has to be done.

8.2.2> Otherwise, advance lp up lps until it hits the cell pointing to c^* and then set $cross[rps[i]]$ to lp. This will finish correctly because (i) we must have descended (right) into the rightmost child of c^* for it to *not* appear in rps, but descending right also means it will appear in lps, and (ii) the local invariant governing lp and rp implies that lp was still at a cell in lps that points to an ancestor of c^*.

8.3> Having handled one of the above two cases, advance rp up rps by one cell and go back to step 7. Note that after advancing rp in this step, we have restored the local invariant above—in the first case we did not advance lp and so the invariant holds true; in the second case $node[rp]$ is deeper than c^* in T because c^* is a child of $node[rps[i]]$ and $node[rps[i+1]]$ is in the rightmost subtree of c^*.

- pass 7 -

9> Repeat step 6 to step 7 to set the cross pointers in lps.

3.4.2 Analysis

Theorem 3.9 Given a node u at height h in a degree-balanced search tree T and a key f in $T|^u$, the hands on f restricted to u can be built in $O(b \cdot h)$ time.

Proof. Note that we have already analyzed the running time of the first five passes of the above algorithm and they all run within the desired time bound. For the sixth and seventh passes, observe that each of lp, rp, and p only gets advanced up in its corresponding stack and that each advancement takes $O(1)$ time. By viewing $T|^u$ as a degree-balanced search tree, Theorem 3.6 applies and thus both of these passes finish in $O(h)$ time. □

4

A Worst-Case Dynamic Finger Tree Algorithm

Besides giving us an easy way to describe and analyze "the hands" data structure that was covered in §3, the concept of heterogeneous decompositions has also enabled us to design a new binary search tree algorithm with the dynamic finger property in the worst case. As opposed to using pseudocode, this time our algorithm is actually specified using a complete implementation in the Standard ML programing language [MTHM97]. The purpose of this chapter is to (i) explain why such a binary search tree algorithm is interesting from both a historical and a theoretical standpoint, and (ii) present the design of our algorithm and annotate a relevant subset of its source code to make it easier to understand. (The complete source code is available upon request.)

Before we get to our motivation, let us point out that the PDF document of this thesis contains two layers that are invisible by default. Both of them depict a binary search tree with 127 keys maintained using our algorithm. The first layer overlays the 127 shapes of this tree in the sorted order on pp. 1–127 of this document; the second layer is identical to the first except it only appears on odd-numbered pages for double-sided printing.

4.1 Motivation

Intellectual Curiosity. The primary motivation behind this work is to answer the following question—why would Sleator and Tarjan [ST85b] conjecture that splay trees have the dynamic finger property in the first place?

To really understand this question, let us describe a relatively common pattern in the study of dynamic optimality for splay trees. The first part of this pattern involves defining a desirable property that can be exhibited on *some* binary search tree algorithm, from which we infer that an optimal binary search tree algorithm must also have this property. Then we would attempt to prove that splay trees also have this property, because it is a necessary condition for splay trees to remain its candidacy of dynamically optimality. We feel that any work following this pattern would seem to require that we pick a particular property which is already known to be exhibited by some binary search tree algorithms. For if we have picked a property that is unknown to be achievable by binary search trees, then we would have no idea if splay trees really "should" have this property or not.

Now of course, part of the above reasoning relies on our faith in splay trees being $O(1)$-competitive. Otherwise the norm among previous works should really be the identification of some desirable property, followed by the design of a binary search tree algorithm with the property, and then a proof that splay trees do not have this property. However, when it comes to the dynamic finger property, we have discovered that the situation in the literature is interesting.

First, we note that at the time when Sleator and Tarjan [ST85b] put forward the dynamic finger conjecture, there were already numerous works on finger searching in various balanced search trees. For some examples, we list [GMPR77], [BT80], and [HM82]. However, to the best of our knowledge, none of these works fits into the binary search tree model. This is of course most certainly known to Sleator and Tarjan as well. But in view of this, Sleator and Tarjan [ST85b, p. 685] did go on to point out that dynamic optimality conjecture implies the dynamic finger conjecture. In other words, a binary search tree algorithm that has the dynamic finger property must be already known at the time, at least to the authors themselves.

We went on to search the literature for such a binary search tree algorithm. In the most definitive word on the dynamic finger property, Cole [Col00] have remarked that the proof that dynamic optimality implying the dynamic finger property is "nontrivial". This assures us that there must be some technical difficulty in obtaining such a binary search tree

algorithm. However, while researching for §5 of this thesis, we also met with what may be the most recent mentioning of such an algorithm, which is by Demaine, Harmon, Iacono, and Pătraşcu [DHIP07, p. 243]:

> [...] a balanced BST supporting search, split and concatenate operations in the worst-case dynamic finger bound, $O(1 + \lg r)$ worst-case time, where r is 1 plus the rank difference between the accessed element and the previously accessed element. For example, one such data structure maintains the previously accessed element at the root and has subtrees hanging off the spine with size roughly exponentially increasing with distance from the root. [...]

While the fact that no citation is offered may suggest that there has been no explicit description of such an algorithm, the reader may have already begun to see how such an algorithm may work and may have perhaps even seen such an algorithm.

Indeed, the most relevant work we found was a design by Kosaraju [Kos81]. Although his design may essentially be described as above, with the subtrees hanging off the spine being $(2,3)$-trees of increasing heights, we must note that the interaction between the subtrees hanging off the spine is highly nontrivial. In fact, the details lying underneath the adjective "roughly exponentially" makes up for the majority of the paper. However, in that form that it is written in [Kos81], Kosaraju's algorithm does not seem to be in the binary search tree model. According to our analysis, the problem has to do with the fact that his algorithm is designed to support efficient insertions and deletions. In particular, Kosaraju would allow consecutive $(2,3)$-trees to have equivalent heights, even though they are supposed to get exponentially larger as the distance from the root increases. To control the amount of redundancy in the sizes, he would then maintain a regularity condition so that, among other sub-conditions, there can only be a $O(1)$ number of subtrees sharing a particular height. But to implement the regularity condition, he ended up using spinal nodes that have five pointers, which can give an ability that may not be feasible in the binary search tree model. In other words, it is not entirely clear that this is the design that Sleator and Tarjan had in mind. This marks the beginning of our effort.

To this end, we have made the simplifying assumption that the tree contains a fixed set of $(2^h - 1)$ keys for any $h \geq 1$. Our algorithm is deterministic and worst-case. This is in contrast to designs like splay trees,

which are amortized, and treaps [SA96], which are randomized. However, we do note that both of these designs have numerous advantages over our design, for they are easier to implement, sport other desirable properties, and support more operations such as insertions and deletions.

We will end this with a somewhat whimsical note. Before we started working on our own design, we were aware of exactly one simple and explicitly-stated binary search tree algorithm that has the dynamic finger property. It is, in fact, the splay tree algorithm. ☺

Stress Test. Another reason why we have developed this program is to catch bugs in our own reasoning. While this is arguably backwards when it comes to the pursuit of correctness, forcing ourselves to implement our own theory have proved to be immensely useful in revealing corner cases that the author has, in all honesty, simply missed. Of course, even though we have done extensive testing with our code as described in §4.8, we still cannot be sure that there can be no bug in our code and hence our reasoning. But in contrast to having programmed our algorithms using pseudocode, we feel that this boosts our confidence in the correctness in two aspects. First, pseudocode—by its very nature—cannot be executed on a computer. Even though we can dry run it on small instances, some corner cases can only reveal themselves in larger nontrivial instances, either causing an incorrect output or even crashing our programs. Second, leaving a specification of an algorithm in pseudocode form means leaving room for typographical mistakes even when the algorithm to be described is completely correct to begin with. This can be witnessed by a cursory glance over the errata of [CLRS01][1], which reveals a fair number of such typos.

Having made the above observations, we have come to the conclusion that we should attempt to leave our algorithmic specification in a state that is as mechanically-verifiable as possible. For this work, we have chosen to specify our algorithm in the Standard ML programming language [MTHM97]. As we will see, the Standard ML type checker actually enforces certain invariants due to the way the datatypes in our program are chosen, and this has been a significant practical advantage. Another benefit of presenting our algorithm in Standard ML is that we can verify that our program is purely-functional by mere inspection. (Standard ML have constructs that cause side-effects, but it is easy to check that we did not use any of them.) However, we do note that all binary search tree

[1] http://www.cs.dartmouth.edu/~thc/clrs-2e-bugs/bugs.php

algorithms on a static set of keys are inherently purely-functional simply because the reorganization tree can be implemented by copying.

4.2 Design

Intuitively, our dynamic finger tree is an implementation of the heterogeneous spines of a complete binary search tree T with respect to one of its key f. The idea is to have f always at the root and use the two spine structures of a binary search tree to implement the heterogeneous spines. (For the moment, we may think of the spine structures simply as the spines. This is a complication that we will explain later.) The subtrees that are hanging off the ancestral paths of f in the reference tree will then be hung off the two spine structures of the dynamic finger tree. While this is similar to the design of the hands, where two parent stacks and a collection of spine prefix stacks are used to implement the heterogeneous spines, there are several major aspects where the dynamic finger tree is different from the hands, all of which are consequences of staying in the binary search tree model.

The first difference lies in the fact that catenable stacks have abilities that do not come easily in the binary search tree model. In particular, they can be catenated in worst-case $O(1)$ time. Recall that in the algorithm to perform a forward step in the hands, we rely on this ability to prepend the spine prefix stack to the parent stack even though a spine prefix stack can grow to unbounded length. To allow prepending to the spine structures of the dynamic finger tree in worst-case $O(1)$ time, we must therefore plan to use incremental prepending, which correctly implements a catenation because every secondary key residing in the spine prefix goes before the primary keys in the ancestral path. Fortunately, with the help of an extra bit at each node, this turns out to be both easy and advantageous when we get to the next difference below.

The second difference is lies the fact that we cannot have extra pointers such as the cross pointers, which as we recall is critical for the $O(1)$ time absorption in the nontrivial parent case. But as we will see in the source code, cross pointers are in fact not needed in a dynamic finger tree because the merging is done incrementally. Whenever we need to undo the effect of a partially completed incremental merge, it would be the case that the amount of work that is already done is proportional to the dynamic finger budget. (In fact, this is another way to implement the hands, although it certainly adds to the complexity.)

The third difference is due to the use of incremental merging. Because we will not have enough time to move to the spines all nodes that should be merged in, some nodes that correspond to the primary keys will have to reside in places other than the two spines of the dynamic finger tree. This necessitates the storage of an extra bit at each node to help us differentiate nodes that contain the primary keys, and is the reason why we keep referring to the "spine structures" instead of just "spines". (In other words, the two parent stacks in the hands are now implemented by the two spine structures of the dynamic finger tree.) A clean way to accomplish this turns out to be using this bit to indicate whether a node in the dynamic finger tree is the root of a subtree that contains the keys of the reference subtree hanging off the access path of f. Such a subtree is called a boxed subtree, which we will explain soon.

The fourth difference is how we represent the spine prefixes themselves. Recall that in the hands, we store each key in a spine prefix in a cell of the spine prefix stacks and maintain pointers to these stacks in the cells of the parent stacks. Not having the space to store the spine prefixes anywhere, we resort to "storing" them by skewing the boxed subtrees themselves. The notion of skewness is very easy to understand. Consider a complete binary search tree T_{ref} which is a reference subtree hanging off the access path of f. The initial shape of the boxed subtree $T_{\square}$ that contains the keys in T_{ref} is exactly the same as T_{ref}. To right skew $T_{\square}$ once is to rotate its root t to the right, thus making the left child of t as the new root. This is to signify that the spine prefix of T_{ref} now contains the key inside t. (It is important to note that the new root of $T_{\square}$ is *not* yet part of the spine prefix.) Further skewing $T_{\square}$ means a longer spine prefix. (In the pointer view, we can think that the pointer that leads to the root of a boxed subtree is in fact a pointer to the deepest key of the spine prefix.)

A further complication arises because a complete binary search tree of height h can only be right skewed $(h-1)$ times because each skew shortens its left spine by one key. However, a boxed subtree of height h, which initially starts in the shape of a complete binary search tree, can be skewed h times when it corresponds to the left or right reference subtree of f. Therefore, we have to introduce another bit to indicate whether the current root of a boxed subtree is in the spine prefix or not.

To finish the overview, we note that our algorithm is conceptually similar to the one for the hands and it is based on unrolling the decomposition by means of absorption. The running time analysis follows analogously since we will be simulating all operations of the hands in real-time. With-

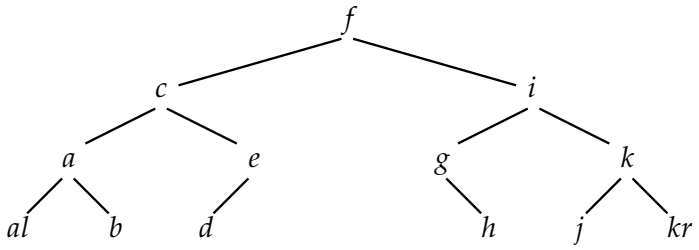


Figure 4.1 – Node naming convention used in our code

out getting bog down by further details of the representation, which will become clear once we get pass the ability to build the dynamic finger tree in §4.5, we will end this with Figure 4.1, which shows the alphabets we used to name the nodes in our code, and Figure 4.2, which shows the call graph of the functions that we are going to describe.

4.3 Datatype and Utilities

Datatype. We will start by understanding our main datatype. We define the type of an α tree as usual, but with two twists. Usually, an α tree is defined to be either an external node `E`, or it is an internal node that is represented by a tuple made of a left α subtree, a key of type α, and a right α subtree. Both internal nodes and leaves are referred to as nodes. In our dynamic finger trees, however, we need to store additional bits of information into the nodes. First we extend the tuple defining an internal node by a `HeadBit`, which is either `H` (set, or head) or `T` (clear, or tail). The purpose of the head bit will be explained later, and we merely remark that setting it denotes that the key is a nontrivial parent key. Then, in contrast to introducing two bits into the tuple explicitly, we differentiate among four types of internal nodes by using four different constructors `X` (exposed), `B` (balanced), `S` (skewed), and `Z` (maximally skewed).

```
16 datatype HeadBit = H | T
17 datatype 'a Tree =
18   (* non-boxed types *)
19     E                                                          (* Empty *)
20   | X of 'a Tree * 'a * HeadBit * 'a Tree                  (* eXposed node *)
21   (* boxed types *)
22   | B of 'a Tree * 'a * HeadBit * 'a Tree               (* Balanced, S_h^0 *)
23   | S of 'a Tree * 'a * HeadBit * 'a Tree   (* Skewed, for i=[1,h-1]: S_h^i *)
24   | Z of 'a Tree * 'a * HeadBit * 'a Tree        (* maZimally skewed, S_h^h *)
```

Boxed and Non-Boxed Types. We will refer to the `E` and `X` types of nodes as non-boxed types, while the `B`, `S`, and `Z` types are said to be boxed. We may think of a non-boxed node as the usual kind of nodes used in typical search tree implementations. The three types of boxed nodes are

introduced to represent the root of the boxed subtrees hanging off the two spine structures of the dynamic finger tree. The distinction among these three types will be explained when we get to the function `skewR`.

Sentinels as Exceptions. Besides the five types of nodes, we also have two exceptions that are related to sentinels. At present, we do not use the $-\infty$ and ∞ sentinels in our dynamic finger trees. Instead, whenever a sentinel is reached, we raise the corresponding exception. This does not cause any problem because we can safely assume all accesses are within $[n]$ in the binary search tree model. We note that this is also the reason why we have specialized our α type to `int`, even though the majority of our code can easily be modified to not relying on this specialization at all. (This basically would be to changing our code to use a generic comparator function since we currently use < on the `int` type.)

```
34 exception PositiveSentinel of int Tree
35 exception NegativeSentinel of int Tree
```

Building a Complete Binary Search Tree. At this point, we are ready to demonstrate how to build a complete binary search tree of height h, which is also the starting point of our procedure to build an dynamic finger tree. The procedure is a simple divide-and-conquer based on recursion. Note that the head bits in all nodes are initialized to `T` by default. Also, note that even though we are building a complete binary search tree for the moment, later on it will be restructured like a binary search tree. In particular, the number of internal nodes in the tree we return will be $(2^h - 1)$, which is computed by the `twoHMO` on the input h. The number of nodes, which includes that of leaves, is $(2^{h+1} - 1)$.

```
60 fun buildBst h = let
61   fun builder l r =
62       if l = r then X (E,l,T,E)
63       else let val m = Int.div (l + r, 2) in
64              X (builder l (m - 1),m,T,builder (m + 1) r) end
65 in builder 1 (twoHMO h) end
```

Rotations. We have also introduced four reparenthetization functions to perform rotations quickly. Note that these functions rotate a node only if it is an exposed node, and will throw an exception otherwise.

```
81 fun rotateL (xl,xk,xh,X (yl,yk,yh,yr)) = (X (xl,xk,xh,yl),yk,yh,yr)
82   | rotateL _ = raise Fail "rotateL on non-X edge"
83 fun rotateR (X (xl,xk,xh,yl),yk,yh,yr) = (xl,xk,xh,X (yl,yk,yh,yr))
84   | rotateR _ = raise Fail "rotateR on non-X edge"
```

The two functions `rotateL'` and `rotateR'`, which are not shown here, are defined to be `rotateR` and `rotateL` for symmetry. (In our coding style, the primed variant of a function is the inverse function.)

4.4 Skewing

The most important concept for our dynamic finger trees is the skewness of a boxed subtree, which models the length of the spine prefix of the corresponding reference subtree and increases as the heterogeneous height of the corresponding right ancestor of f drops. Although we have not yet seen how the first boxed nodes are introduced into dynamic finger trees by the function `buildDftAtWalker`, we note that skewing only occurs at the root of a boxed subtree.

Recall that the right skewing operation is simply a right rotation at the root of a boxed subtree in the dynamic finger tree, with its left child becoming the new root. A boxed subtree T that corresponds to a reference subtree of height h can have $(h+1)$ different possible amount of skewness—or simply "skew(s)"—ranging from 0 to h. Initially, T has 0 skews and its initial shape is a complete binary search tree of height h. The root of T will have the type `B`, and for convenience we will refer to T in this configuration as a B_h. Once T has acquired a skewness of i for $1 \le i < h$, the root of T will have the type `S` and we refer to T as a S_h^i. Finally, as we have discussed in §4.2, T can be skewed h times but we only have $(h-1)$ possible shapes. Therefore, when T has been skewed h times, we let its root take on the type `Z` and refer to it as a Z_h.

To prepare for the possible merge of the spine prefix with the parent stack, perhaps due to a forward step, we will also need to start boxing the subtrees inside T as it increases its skewness. In general, this means the right subtree of the left child of the original root will now become a boxed left subtree of the right child of the new root. However, care must be taken in these cases to box the right child in the first skew and also to avoid boxing an `E` in the skew before the final skew:

(1) A boxed subtree B_1 will be skewed into a Z_1 immediately.
(2) After the initial skewing a boxed subtree B_2, we must not box the left subtree of the right child of the new root, as this subtree is in fact an `E`.
(3) Otherwise, we are skewing a boxed subtree B_h for $h > 2$ and the right subtree of the right child of the new root will also be boxed. This kind of boxed subtree will also be referred to as an end boxed subtree.
(4) When B_h for $h > 2$ is getting skewed for the h-th time, we will return it as a Z_h to signify this.

(5) A similar precaution to (2) must be taken in general when a B_h for $h > 2$ is getting skewed for the $(h-1)$-th time since the right subtree of the left child before the skew is also an `E`.

```
126 fun skewR tree = case tree of
127       E   => E
128     | X _ => raise Fail "skewR X is always a bug"
129     | B (n as (E          ,_,_,E))                    (*    B_1       *)
130     =>Z  n                                            (* => Z_1       *)
131     | B (X (E ,xk,T,   E)    ,yk,T,X yrn )            (*    B_2       *)
132     =>S (   E ,xk,T,X (E     ,yk,T,B yrn))            (* => S_2^1     *)
133     | B (X (xl,xk,T,X    xrn),yk,T,X yrn )            (*    B_h       *)
134     =>S (   xl,xk,T,X (B xrn ,yk,T,B yrn))            (* => S_h^1     *)
135     | S (n as (E          ,_,_,_))                    (*    S_h^{h-1} *)
136     =>Z  n                                            (* => Z_h       *)
137     | S (n as (X (E,_,_,E),_,_,_))                    (*    S_h^{h-2} *)
138     =>S (rotateR n)                                   (* => S_h^{h-1} *)
139     | S (X (xl,xk,T,   X xrn),yk,T,yr    )            (*    S_h^i     *)
140     =>S (   xl,xk,T,X (B xrn ,yk,T,yr   ))            (* => S_h^{i+1} *)
141     | Z _ => raise Fail "skewR Z is always a bug"
142     | _   => raise Fail "skewR unknown BS pat"
```

Finally, note that we will thrown an exception should we ever try to skew a `Z`. This is because conceptually a `Z` is already at its maximum skewness and we should never increase its skewness.

The function `skewR'` is the inverse of the function `skewR`. The special cases can be derived similar to those of `skewR`.

```
144 fun skewR' tree = case tree of
145       E   => E
146     | X _ => raise Fail "skewR' X is always a bug"
147     | B _ => raise Fail "skewR' B is always a bug"
148     | S (   E ,xk,T,X (E     ,yk,T,B yrn))            (*    S_2^1     *)
149     =>B (X (E ,xk,T,   E)    ,yk,T,X yrn )            (* => B_2       *)
150     | S (n as (E,_,_,X (E,_,_,_)))                    (*    S_h^{h-1} *)
151     =>S (rotateR' n)                                  (* => S_h^{h-2} *)
152     | S (   xl,xk,T,X (B xrn ,yk,T,B yrn))            (*    S_h^1     *)
153     =>B (X (xl,xk,T,   X xrn),yk,T,X yrn )            (* => B_1       *)
154     | S (   xl,xk,T,X (B xrn ,yk,T,yr   ))            (*    S_h^i     *)
155     =>S (X (xl,xk,T,   X xrn),yk,T,yr    )            (* => S_h^{i-1} *)
156     | Z (n as (E,_,_,E          ))                    (*    Z_1       *)
157     =>B  n                                            (* => B_1       *)
158     | Z (n as (E,_,_,_          ))                    (*    Z_h       *)
159     =>S  n                                            (* => S_h^{h-1} *)
160     | _   => raise Fail "skewR' unknown SZ pat"
```

4.5 The Ability to Build

Two Forward Operators. We will start with two convenience operators known as the forward pipe operator `|>` and the forward composition operator `>>`. These are not in the Standard ML programming language but their definitions are simply:

```
1  fun (x |> f) = f x
2  fun (f >> g) = g o f
```

Although it will not matter in our code here, the reader is referred to `infix.sml` for the relative precedence of these operators.

4.5.1 Building a Finger Search Tree

`buildDftAt`. First comes a wrapper function that calls `buildDftAtWalker` and `buildDftRotator` for the actual work. We will explain these functions next.

```
318 fun buildDftAt x tree =
319   buildDftAtWalker x E tree E |> #3 |> buildDftAtRotator
```

`buildDftAtWalker`. The function `buildDftAtWalker` takes four arguments. The first is simply the target key x where we should build our dynamic finger tree at. The third is the root of the subtree we should look at. As seen in `buildDftAt`, initially this is simply the root of a complete binary search tree. The second and the fourth will be explained below.

Conceptually, we can think of `buildDftAtWalker` as having a descend phase, a destination phase and an ascend phase. In the descend phase, we search for x as usual, but we do two extra actions before each descend. Suppose this is a right descend. First, we box and keep track of the left subtree of the current node in *ilst*, which is the second argument of `buildDftAtWalker`. The fourth argument is simply the mirror of the second. Second, we skew the subtree that we have been keeping track in *irst*. Observe that, due to the symmetry of the first move above, this is the right subtree of the last node we descended left (all subsequent descends are to the right). From the above two actions, we see that the left subtree of a node where we descended right will be skewed k times if we continue by descending left k times before the next right descend. Finally, as a side note, notice that although the initial value of these two arguments are set to `E`, they will be set after the first right descend and the first left descend. For convenience in this function, we have allowed `skewR` to skew an `E` without throwing an exception.

In the destination phase, we have reached the target node that contains x. We will simply skew the two subtrees of the target node to their maximum before the next phase.

In the ascend phase, we also perform two extra actions in each ascend. The first is simply a rotation to rotate the target node to the root. The second is to set the head bits of the nontrivial ancestors. Since we did not keep track of the direction of the descend, we would provide the answer to both cases. The first (last) value of the tuple to be returned is the head bit of the parent if it was a right (left) descend. Say we have just performed a right descend and we are about to return. If we are at a right child, then the parent is a trivial ancestor. Otherwise, we are at a left child, and the parent will be a nontrivial ancestor.

```
fun buildDftAtWalker x = let
  (* dn: skew the hanging subtrees towards `x' accordingly *)
  (* up: rotate `x' to root *)
  fun walker ilst (tree as X (E,_,T,E)) irst =       (* bottom node stays T *)
      (T, ilst, tree, irst, T)
    | walker ilst (        X (l,k,T,r)) irst =
      if x = k then let           (* skew our two subtrees all the way to Z *)
          fun skewer skewFn = let
            fun worker tree = case tree of
                        Z _ => tree                          (* termination *)
                      | X n => worker (B n)               (* start by boxing *)
                      | _   => worker (skewFn tree)          (* recursion *)
          in worker end
        in                                              (* internal marks H *)
          (T, ilst, X (skewer skewL l,k,H,skewer skewR r), irst, T)
        end
      else if x < k then let            (* skew ilst and pass r as new irst *)
          val (_, olst, c, orst, h) = walker (skewL ilst) l (box r)
        in     (* ilst touched; L => (c,orst) are new (l,r); irst untouched *)
          (H, olst, X (rotateR (c,k,h,orst)), irst, T)        (* LP marks H *)
        end
      else let                          (* skew rlst and pass l as new ilst *)
          val (h, olst, c, orst, _) = walker (box l) r (skewR irst)
        in     (* ilst untouched; R => (olst,c) are new (l,r); irst touched *)
          (T, ilst, X (rotateL (olst,k,h,c)), orst, H)        (* RP marks H *)
        end
    | walker _ _ _ = raise Fail "buildDftAt walker unknown pat, probably X:H"
in walker end
```

`buildDftAtRotator`. The function `buildDftAtRotator` takes one argument, which is the root of the partially build dynamic finger tree piped from the third position of the tuple returned by `buildDftAtWalker`. Recall that the target key x is already rotated to the root. It's easy to verify that rotating x to the root would cause the left-right and the right-left spines to contain exactly, up to the two (now boxed) subtrees of x before the rotation, the left and right ancestors of x.

Assuming that we do not have the concern of incremental merging, this function simply rotates the two children of the root to exhaust the left-right and the right-left spine until we hit the two boxed subtrees. This would put the left (right) ancestors on the left (right) spine, with the right child relation being the "right parent of" relation. Indeed, we can think of this as inverting the heterogeneous spines of x and storing them on the two spines of the dynamic finger tree.

Alas, incremental merging is a real concern when we get to the functions `prependL'` and `prependR`. The idea of `buildDftAtRotator` is invert the spines while performing an "incremental unmerge" to mimic the effect of partially completed incremental merges. This brings us to the dynamic finger tree equivalent of Invariant 3.4, which for the reason of synchronization with §3 is displayed as Invariant 4.4.

We will explain the significance of Invariant 4.4 when we get to the function `prependL'`. For now, we simply note it is easy to maintain it as we invert the right-left spine into the right spine—whenever we see the (T,T,H) pattern at the (g,i,k) position, we will first right rotate at i to bring g up as usual, but then perform an additional left rotation at i to bury i as the left child of k.

```
293 fun buildDftAtRotator tree = let
294   val (l,k,h,r) = get4 tree
295   fun rotatorL ls = case ls of
296         E             => ls                                  (* singleton *)
297       | Z _           => ls                                  (* no parent *)
298       | X (_,_,_,E  ) => ls                                       (* done *)
299       | X (_,_,_,Z _) => ls                          (* stop at own child *)
300       | X (X (ll,lk,H,lr),k,T,X (rl,rk,T,rr))                   (* H,T,T *)
301       =>X (X (ll,lk,H,X (lr,k,T,rl)),rk,T,rr) |> rotatorL
302       | X (n as (_,_,_,X _)) => X (rotateL n) |> rotatorL
303       | _ => raise Fail "buildDftAt rotatorL unknown pat"
304   fun rotatorR rs = case rs of
305         E             => rs                                  (* singleton *)
306       | Z _           => rs                                  (* no parent *)
307       | X (E  ,_,_,_) => rs                                       (* done *)
308       | X (Z _,_,_,_) => rs                          (* stop at own child *)
309       | X (X (ll,lk,T,lr),k,T,X (rl,rk,H,rr))                   (* T,T,H *)
310       =>X (ll,lk,T,X (X (lr,k,T,rl),rk,H,rr)) |> rotatorR
311       | X (n as (X _,_,_,_)) => X (rotateR n) |> rotatorR
312       | _ => raise Fail "buildDftAt rotatorR unknown pat"
313 in
314   X (rotatorL l,k,h,rotatorR r)
315 end
```

The function `get4` simply returns the tuple that represents an internal node.

Remark 4.1. Technically we can combine `buildDftAtWalker` and `buildDftAtRotator` into one function, although we see that abstracting them as two separate functions actually facilitates code reuse.

4.5.2 Spine Structures and Invariants

At this point we are ready to state the dynamic finger tree equivalent of Invariant 3.2 and Invariant 3.3. But first, let us define the i-th node on the right spine structure of a dynamic finger tree to be the i-th node we visit when we perform an in-order traversal of all exposed nodes in the right subtree of the root. In other words, the traversal would regard the root of boxed subtrees as leaves and return, and had we not performed the incremental unmerge, the i-node on the right spine structure is the i-th node on the right spine of the right child of the root. The root then is considered to be the 0-th node on the right spine structure.

For $i > 0$, the i-th subtree hanging off the right spine structure is the left subtree of the i-th node on the right spine structure. Note that these subtrees are all boxed, with the exception when it is an `E`. Note also that, due to the root being the 0-th node, the i-th subtree hanging off the right spine structure in fact corresponds to right reference subtree of the $(i-1)$-th key. We will say that this boxed subtree, i.e., the i-th one, is hanging off the $(i-1)$ key even though in general this boxed subtree is the left subtree of the right child of the key. Finally, suppose there are k nodes on the right spine structure, which means the rightmost node t on the right spine structure is the $(k-1)$-th. We further define the k-th subtree hanging off the right spine structure to be the right subtree of t, which is also either an `E` or a boxed node as it is the end boxed subtree. This also corresponds to the only case ($i = k$) when the i-th subtree is actually a child of the $(i-1)$-th key on the right spine structure.

With the above definitions, we state the three remaining invariants besides Invariant 4.4.

Invariant 4.2 (Primary) Suppose the right spine structure has k nodes. The 0-th node on the right spine structure contains the most recently accessed key. For $1 \leq i \leq k$, the i-th node on the right spine structure contains the right parent key of the key in the $(i-1)$-th node in the reference tree.

Invariant 4.3 (Secondary) Suppose the right spine structure has k nodes. For $1 \leq i \leq k$, suppose T_i is the i-th subtree hanging off the right spine

structure. Let w_{i-1} and w_i be the $(i-1)$-th and the i-th nodes on the right spine structure, with w_k being the sentinel ∞ as in §2.1.3. Then T_i is $(\mathrm{hh}_f(w_i) - \mathrm{hh}_f(w_{i-1}) - 1)$-skewed, where f is the key at the root of the dynamic finger tree.

Invariant 4.4 (Incremental Prepend) Let the right spine of an dynamic finger tree contains the non-root grandparent g, the parent p and the child c. If the head bit of c is set, then the head bits of g and p cannot be both clear.

Invariant 4.5 (Incremental Merging Lower) If the right child i of the root has its head bit set, and if i has a left child g, then g cannot be exposed.

4.6 The Ability to Step

Recall from §3 that the ability to step is a primitive used in the search. We will start by discussing the function `stepR` before we go into the functions it depends on.

`stepR`. The function `stepR` can be understood in two parts. In the first part, we use the functions `incMergeR`, `extendR`, and `prependR` to operate on the right subtree of the current root. The current root at f is thus implicitly "popped". This will give us the new *root*, which contains the key at g, along with its new right subtree. In the second part, we operate on the left subtree of the current root using the corresponding functions `prependL'`, `extendL'`, and `incMergeL'`, which are of course used in this reversed order. However, as we will see, we will need to fix the head bit on the new left child before we use `extend'`. This is accomplished by the function `fixHeadL`.

```
451 fun stepR (tree as X (_,_,_,E)) = raise PositiveSentinel tree
452   | stepR tree = let
453       val (l,f,t,r) = get4 tree
454       val ltree = X (l,f,t,E) |> prependL' >> extendL'          (* clear R *)
455       val root = r |> incMergeR >> extendR >> prependR
456     in case fixHeadL ltree root of
457          X (l,f,t,r) => X (incMergeL' l,f,t,r)
458        | _ => raise Fail "stepR fixHeadL only returns X"
459     end
460
461 ;use "visualizations.sml";
```

Assuming that each of the above functions takes worst-case $O(1)$ time, we conclude that `stepR` runs in worst-case $O(1)$ time as well.

`incMergeR`. The functio `incMergeR` is very easy to understand. Whenever we see a (T,T,H) pattern in the (i,j,k) position, we know that there is

an incremental merging going on and the left-right spine of k is to be prepended in front of k. (Recall j is the left child of k.) Therefore, we simply rotate j up as a step in the incremental merge to satisfy Invariant 4.5 since i will become the new root.

```
409 fun incMergeR (X (il,i,T,X (kn as (X (_,j,T,_),k,H,_)))) =
410               X (il,i,T,X (rotateR kn)              )
411   | incMergeR itree = itree                            (* nothing to merge *)
```

We note that this is the reason for the name "head bit". It signifies that k is the head of the second part of the incremental merge.

`extendR`. The function `extendR` extends the spine prefix of the key in i, which in general corresponds to the boxed subtree in the j position, i.e., it is the boxed subtree hanging off i. This can be very confusing because it is not at all clear why there is a boxed subtree at the j position. To clarify this, we must note that `extendR` is called after `incMergeR`, which means that at this point the key in the k position is in fact the key following the key at i in the right spine structure. (For instance, if the incremental merge had an effect, the key in the k position was in the j position back in `incMergeR`.) We also note that a special case occurs if i is already the rightmost key on the right spine structure, in which case the boxed subtree hanging off i is in fact k, the end boxed subtree. Both of these cases restore Invariant 4.3 at j.

```
338 fun extendR (X (il,i,it,X (      j,k,kt,kr))) =             (* i's box is j *)
339             X (il,i,it,X (skewR j,k,kt,kr))
340   | extendR (X (il,i,it,      k)) =                         (* i's box is k *)
341             X (il,i,it,skewR k)
342   | extendR (ztree as Z _) = ztree                               (* i degen *)
343   | extendR _ = raise Fail "extendR unknown pat"
```

Finally, we note that this function corresponds to decrementing the heterogeneous height of the right parent of the new root, which is in the i position. The consequence of which is to extend its spine prefix by one key, which we did.

`prependR`. The last function we need for adjusting the right subtree is the function `prependR`. Its purpose is to restore Invariant 4.2 and Invariant 4.4. Observe that when the function gets called, we have already performed the pop and the extend parts of the forward step. What's missing is to prepend the spine prefix of the original root to the front of the right spine. We see that this spine prefix corresponds to the left boxed subtree of i, which must have already been maximally skewed since it is the boxed subtree hanging off f before the forward step. Invariant 4.4 actually gives us a good idea of

what to do. Suppose the boxed subtree is a Z_h, which will be exposed since the corresponding secondary keys are now primary. If $h \geq 2$, then we will incrementally merge the first two keys into the right spine of the dynamic finger tree. If $h = 1$, then one key will be merged. Otherwise $h = 0$ and no keys need to be merged. In all three cases, we will set the head bit of i to indicate that i is the head of the second part of the concatenation. Finally, we note that there is a special end case when f was in fact the rightmost key on the right spine structure. In this case, we will simply expose the boxed subtree hanging off it, which is at position i.

```
377 fun prependR itree = case itree of
378       X (Z (E,g,T,X (hl,h,T,   hr)),i,_,ir  )                    (* Z>1 *)
379     =>X (   E,g,T,X (hl,h,T,X (hr  ,i,H,ir)))                  (* mark H *)
380     | X (Z (E,g,T,   E),i,_,ir )                              (* Z=1, h degen *)
381     =>X (   E,g,T,X (E ,i,H,ir))                               (* mark H *)
382     | X (E,i,_,ir) => X (E,i,H,ir)                            (* Z=0; mark H *)
383     | Z (n as (E,_,_,_)) => X n                             (* Z at end; unbox *)
384     | E => E                                                    (* empty rps *)
385     | _ => raise Fail "prependR unknown pat"
```

`prependL'`. The effect of the function `prependL'` can easily be understood by reversing the function `prependR`. Recall that, before the prepend that we are trying to undo, we have just initiated an incremental merge to prepend the Z_h at the e position in front of c. A trick we deployed is to mark the head bit of c to signify that it is the head of the second part of the concatenation. It is important to note this bit is set even if there was nothing to be prepended. We can therefore use the first head bit we see on the left spine structure to determine the head of the second part and "unprepend" accordingly, thanks to Invariant 4.4. Finally, we remark that the three end cases are merely a boxing operation and could be collapsed into one had we not been so keen on explicitly matching all possible cases.

```
365 fun prependL' tree = case tree of
366       X (_,_,H,E) => tree                                       (* Z=0 *)
367     | X (X (X (al,a,H,      ar),c,T,cr),f,T,E )                 (* Z>1 *)
368     =>X (      al,a,H,Z (X (ar ,c,T,cr),f,T,E))
369     | X (X (cl,c,H,   E),f,T,E )                                (* Z=1 *)
370     =>X (   cl,c,H,Z (E ,f,T,E))
371     | X (n as (E                   ,f,T,E)) => Z n            (* Z=1 end *)
372     | X (n as (X (B (E,a,T,E),c,T,E),f,T,E)) => Z n            (* Z=2 end *)
373     | X (n as (X (X (_,a,T,_),c,T,E),f,T,E)) => Z n            (* Z>2 end *)
374     | _ => raise Fail "prependL' unknown pat"
```

`extendL'`. The operation of `extendL'` is the exact opposite of `extendR`. We merely remark that we are "unextending" the boxed subtree hanging off c.

```
330 fun extendL' (X (X (al,a,at,       b),c,ct,cr)) =          (* c's box is b *)
331               X (X (al,a,at,skewL' b),c,ct,cr)
332   | extendL' (X (       a,c,ct,cr)) =                      (* c's box is a *)
333               X (skewL' a,c,ct,cr)
334   | extendL' (ztree as Z _) = ztree                            (* c degen *)
335   | extendL' _ = raise Fail "extendL' unknown pat"
```

fixHeadL. The function `fixHeadL` is a distinctive function in the forward step for it is not mirroring some function that operates on the right spine structure. Instead, it uses the newly built right spine structure to help determine if we should clear the head bit of the key at f, which after the forward step will be the key at the left child of the root. This is one of the few places where we clear the head bit, which was set by the function `prependR` for the purpose of our current discussion of a forward step. The idea is to look at the right child of the new root, and see if it corresponds to a nontrivial parent key. If so, using the fact that the reference tree is a complete binary search tree, we know that the left child of the new root should correspond to a trivial parent key. It could also be possible that g is on the right spine of the reference tree, in which case the the right child is either a maximally skewed boxed subtree, or `E`. If none of the the above applies, we leave the head bit as it was. Finally, `fixHeadL` is also responsible for piecing together the left and the right spine structures.

```
418 fun fixHeadL (ltree as X (ll,l,_,lr)) (X (E,f,ft,fr)) =
419     (case fr of
420        X (_,_,H,_)            (* only one parent is nontrivial and it is RP *)
421            => X (X (ll,l,T,lr),f,ft,fr)
422      | Z _ => X (X (ll,l,T,lr),f,ft,fr)          (* LP trivial, RP sentinel *)
423      | E   => X (X (ll,l,T,lr),f,ft,fr)            (* at the rightmost node *)
424      | _   => X (ltree        ,f,ft,fr))
425   | fixHeadL (ltree as Z _) (X (E,f,ft,fr)) = X (ltree,f,ft,fr)
426   | fixHeadL _ _ = raise Fail "fixHeadL unknown pat"
```

incMergeL'. The last function in a forward step is `incMergeL'`. It may appear that we have already performed an unprepend before. However, observe that this is for the spine prefix of f before the backward step that we are trying to undo. This spine prefix corresponds to the Z_h at the e position before. Now that we have performed an unprepend, we still need to undo the possible incremental merge which would bring the key at b to the a position. Hence we will check to see if the head bit of the current al position is set (this was the key at a), and if so, rotate the key at the current a position down to the b position.

```
404 fun incMergeL' (X (X (bn as (X (al,a,H,ar),b,T,br)),c,T,cr)) =
405                X (X (rotateL' bn)                 ,c,T,cr)
```

```
406   | incMergeL' ctree = ctree                                (* nothing to unmerge *)
```

We note that the code above names the nodes as they were before the incremental merge was performed, hence b seems to be on the left spine and has a as its left child.

4.7 The Ability to Search

As we may recall from §3, our search function is built upon four abstractions based on absorption. We will first discuss them in the context of dynamic finger trees before we show how we perform a search.

4.7.1 Abstractions

promoteRoot. The function `promoteRoot` is responsible for promoting the key at the root to one less than its (actual) height, therefore eliminating its spine prefix. In the context of dynamic finger trees, this corresponds to eliminating the skewness of the boxed subtrees hanging off it and can be done by successively called the functions `extendL'` and `extendR'`. There are a total of six terminal cases because of our treatment of the end cases. We merely remark that in all these six cases, the skewness of the boxed subtree hanging off the root has been completely eliminated.

```
464 fun promoteRoot tree = case tree of
465       X (E,_,_,_) => tree                                (* 3 leaf cases *)
466     | X (_,_,_,E) => tree
467     | X (X (_,_,_,E),_,_,X (E,_,_,_)) => tree
468     | X (B _,_,_,_) => tree                                  (* 3 B cases *)
469     | X (_,_,_,B _) => tree
470     | X (X (_,_,_,B _),_,_,X (B _,_,_,_)) => tree
471     | _ => tree |> extendL' >> extendR' |> promoteRoot       (* otherwise *)
```

demoteRoot. The function `demoteRoot` is the opposite of `promoteRoot`. The only difference is that we will set the head bit of the root key after the demotion if the key resides in a non-bottom internal node in the reference tree. Observe that, from what we understand from the forward step, a key at a non-bottom internal node will has its head bit marked until it rolls over the left side of the root of the dynamic finger tree. It is until then that we will check if we should clear its head bit. This explains why we mark its head bit here.

```
474 fun demoteRoot tree = case tree of
475       X (E,_,_,_) => tree                                (* 3 leaf cases *)
476     | X (_,_,_,E) => tree
477     | X (X (_,_,_,E),_,_,X (E,_,_,_)) => tree
478     | X (l as Z _,k,_,r) => X (l,k,H,r)                (* 3 Z cases, mark H *)
```

```
479     | X (l,k,_,r as Z _) => X (l,k,H,r)
480     | X (l as X (_,_,_,Z _),k,_,r as X (Z _,_,_,_)) => X (l,k,H,r)
481     | _ => tree |> extendL >> extendR |> demoteRoot            (* otherwise *)
```

absorbTrivialRP. We use the function `absorbTrivialRP` to absorb the key at the root and advance to its trivial right parent key, which will be at the new root. The first thing we do is to promote the root key, thus eliminated the skewness of the two boxed subtrees hanging off it. A key step is to perform an `incMergeR` now to ensure that the key at the k position is in fact the right parent key of the key at the i position.

We will then dispatch based on whether the left child is exposed or not. If not, we are in the easy case since there is no left parent key. (This happens when f is on the left spine of the reference tree and is the second case in the code.) In this case, we merely need to prepare a new boxed subtree nb and put it as the left child of the new root, which contains the key at i. Otherwise, the preparation of nb proceeds by first building a `B` node that contains the root's left child, the key at the root, and the root's right child. We further have to determine the head bit lh of the new left child. Fortunately, it simply is the opposite of what the new right child would have. At this point we assemble the new left child using nb as its left subtree. Notice that nb will therefore be the boxed subtree hanging off the new root, exactly as it should be. There are two final steps. First, we must call `extendL'` on the new left child because its key has just increased in its heterogeneous height. We can witness this by noting that its support, which comes from nb, is one more than it was from lr (the left subtree of nb now). Second, we must also call `incMergeL'` on nl to make sure that it does not violate Invariant 4.4 should we have cleared its head bit lh.

```
502 fun absorbTrivialRP dftP =
503     case promoteRoot dftP of          (* turn associated trees into B or E *)
504       X (l,k,_,r) =>
505       (case (l, incMergeR r) of                    (* restore RS invariant *)
506          (X (ll,lk,H,lr), X (rl,rk,T,rr)) => let             (* LP present *)
507            val nb = B (unBE lr,k,T,unBE rl)
508            (* if we have a new trivial RP, then LP remains nontrivial *)
509            val lh = case rr of X (_,_,T,_) => H | _ => T
510            val nl = X (ll,lk,lh,nb) |> extendL'
511          (* if lh is cleared, check and restore LS invariant *)
512          in X (incMergeL' nl,rk,T,rr) end
513        | (_             , X (rl,rk,T,rr)) => let              (* LP absent *)
514            val nb = B (unBE l,k,T,unBE rl)
515          in X (nb,rk,T,rr) end
516        | _ => raise Fail "absorbTrivialRP unknown l,r pat")
517     | _ => raise Fail "absorbTrivialRP non-X pat"
```

Finally, we note that the function `unBE` exposes a `B` node to become an `X` while passing `E` through.

`absorbNonTrivialRP`. The function `absorbNonTrivialRP` is in fact largely similar to `absorbTrivialRP`, with only a complication at the preparation of the new left child nl. Ideally, it is simply a node consisting of the left child of the root l, the key k at the root and also the subtree hanging off to the right of the root rl. Observe that when the right parent key rp is nontrivial, there can be an unbounded number of left ancestor keys that are in the left reference subtree of rp. Fortunately, these left ancestor keys must appear before the left parent key lrp of rp in the left spine structure, and the head bit of lrp must have set when we prepended them on. In other words, we just need to proceed similar to `prependL'` and rotate all keys up to the first head off the left spine. The only difference now is that after the rotation(s), we must also call `extendL'` on lrp, which will now be in the c position.

```
542 fun absorbNonTrivialRP dftP =
543     case promoteRoot dftP of           (* turn associated trees into B or E *)
544       X (l,k,_,r) =>
545       (case (l, incMergeR r) of                     (* restore RS invariant *)
546          (X (ll,lk,T,lr), X (rl,rk,H,rr)) => let      (* LP must be present *)
547            (* if we have a new trivial RP, then new LP is nontrivial *)
548            val h = case rr of X (_,_,T,_) => H | _ => T
549            (* construct new LP: prepend' all nodes up to the first H into S *)
550            val nl = case X (l,k,T,unBE rl) of
551                       (* if we still have an LP, then we need to extendL' it *)
552                        X (X (X (al,a,H,      ar),c,T,cr),f,T,fr )     (* S>1 *)
553                      =>X (      al,a,h,S (X (ar ,c,T,cr),f,T,fr)) |> extendL'
554                      | X (X (cl,c,H,   E),f,T,fr )                   (* S=1 *)
555                      =>X (   cl,c,h,S (E ,f,T,fr)) |> extendL'
556                      | X n => S n (* no H means we have absorbed the last LP *)
557                      | _   => raise Fail "absorbNonTrivialRP nl unknown pat"
558          (* if we cleared H of new LP, check and restore LS invariant *)
559          in X (incMergeL' nl,rk,T,rr) end
560        | _ => raise Fail "absorbNonTrivialRP unknown l,r pat")
561     | _ => raise Fail "absorbNonTrivialRP non-X pat"
```

4.7.2 The Search Algorithm

With the four abstractions laid out, we are ready to describe the search algorithm.

`search`. This is simply the wrapper to implement the semantics of a dynamic finger we defined in §1. If it is determined that the finger has to move based on the last search target stored in `lastTarget`, it calls `searchForward` to perform the actual work.

```
local
  val lastTarget = ref 0              (* the first search is always forward *)
in
fun search x tree = let
  val k = getK tree
in
  (if x = !lastTarget orelse x = k then tree                        (* no op *)
   else if !lastTarget < x then                                    (* forward *)
     if x < k then tree                              (* x is still not present *)
     else searchForward  x tree
   else                                                           (* backward *)
     if k < x then tree                              (* x is still not present *)
     else searchBackward x tree)
  before lastTarget := x
end
end                                                                 (* local *)
```

`searchForward`. The function `searchForward` proceeds exactly as in the forward search algorithm for the hands. First, we locate the exit key w using the function `exitForward` and restore Invariant 4.3 to obtain a dynamic finger tree at w via `demoteRoot`. Then, if the target x is in the right reference subtree of w, we make a forward step. To locate the turn key w', we use `exitForward` again. If w' is x, then we can restore our invariants via `demoteRoot` again. Otherwise, we use the function `reassembleForward` to restore Invariant 4.3.

```
fun searchForward x dft = let
  val exitDft = dft |> exitForward x |> demoteRoot
in
  if x <= getK exitDft then exitDft
  else let                                        (* x is in the right subtree *)
      val turnP = exitDft |> stepR |> exitForward x
    in
      if x <= getK turnP then demoteRoot turnP
      else let
          val destDft = reassembleForward x turnP
        in
          if x <= getK destDft then destDft
          else raise PositiveSentinel destDft
        end
    end
end
```

`exitForward`. The function `exitForward` scans for the exit key by successive absorptions using `absorbTrivialRP` and `absorbNonTrivialRP` appropriately. Assuming we do not hit the easy case when the target key is one of the primary keys, this means we check if x is less than the right parent key of

the current root. The first time we hit this, we know that x must be in the right reference subtree of the current root key and we return the current root. Note that we do not call `demoteRoot` in this function since we also use it to locate the turn key, from which we simply descend and restore our invariants directly.

```
644 fun exitForward x dftP =
645     if x = getK dftP then dftP    (* do note demote: this happens in lg^2 d *)
646     else case getR dftP of
647            X (_,rk,rh,_) =>
648            if x < rk then dftP                                (* root is exit *)
649            else dftP                                (* recurse after absorption *)
650                 |> (case rh of T => absorbTrivialRP | H => absorbNonTrivialRP)
651                 |> exitForward x
652          | _ => dftP               (* boxes, root has no RP => root is exit *)
```

`reassembleForward`. The last function `reassembleForward` is perhaps conceptually the most complicated of all. Its purpose is to complete building the dynamic finger tree at the target key x. Recall that if this function gets called, then we have already obtained a tree $turnP$ rooted at the turn key k. Note that this tree does not yet fully satisfy our invariants and hence is not a dynamic finger tree. We may think of it as a partially-built dynamic finger tree, however. To complete building it, we will dispatch based on the the right child r of $turnP$ and invoke the appropriate excision argument.

First Case. In the first case, r is a `B` which means k is on the right spine of the reference tree and x is in the reference subtree r_{ref} corresponding to the boxed subtree rooted at r. In this case, we will unbox r (`X rn` in code) and call `buildDftAtWalker` on it. We will pass the subtree hanging off on the left of k, namely lr, as the second argument so that it can get appropriately skewed inside `buildDftAtWalker` and come back as $olst$. This subtree also corresponds to the left reference subtree l_{ref} of k. The key k, which is now a left parent key, will also get its head bit h computed.

Let's look at the node represented by the tuple $ttuple$ assembled in the code right before we perform the left rotation using `rotateL`. The node contains k, with its right child being nc. The left child contains lk, with the left child being ll and the right child being $olst$. Observe that the subtree rooted at nc is in fact the dynamic finger tree build at x using the reference subtree rooted at r_{ref}. Comparing the heterogeneous decompositions of the whole reference tree and that of the subtree rooted at r_{ref}, we see that the only difference is that the former is the latter with some extra keys and subtrees on the left. The rightmost extra key corresponds to k, and to its left the rest of the decomposition. To get the former decomposition

using the latter, we only need to get the heterogeneous height of k correct, which in turns means we have to compute the correct spine prefix for l_{ref}. The rest of the decomposition to the left of l_{ref} is the same as it is in the decomposition at the turn key, which we already have from $turnP$. But of course, the spine prefix of l_{ref} has already been computed correctly when we use `buildDftAtWalker` to skew lr into $olst$. In other words, the only thing that is missing is a left rotation to bring nc to the root, followed by a call to `buildDftAtRotator` to invert the left-right and the right-left spine of the new root. (The fact that we do not have to perform `incMergeL'` lies in the case condition that k is on the right spine of the reference tree and hence all left ancestors are trivial and no incremental merging is in progress.)

Second Case. The second case is only slightly more complicated because now we have to deal with both the left parent and the right parent of the turn key. In this case, x is in the reference subtree r_{ref} corresponding to the boxed subtree rooted at rl, which we unbox into `X rln`. The boxed subtree rooted at $rbox$ corresponds to the right reference subtree of rk, which is the right parent key of k. Depending on whether rr is the rightmost node or not, it is either rrl or rr itself. As we are descending from k to r_{ref} in the reference tree, we have to skew $rbox$ once before we pass to `buildDftAtWalker` as the input subtree on the right. The left input subtree remains lr, and we obtain the values of h, $olst$, c, and $orst$. At this point, again depending on whether rr is the rightmost node or not, we assemble the new right-right child nrr by replacing rrl or rr itself with $orst$.

The piecing together of the new root can be understood based on its two sides. On the right side, we let rk be the child key with nrr as its right child because rk is the leftmost extra primary key in the right of the heterogeneous decomposition. Its heterogeneous height, or more relevantly the skewness of the boxed subtree $orst$ that represents its right reference subtree, has already been adjusted by `buildDftAtWalker`. On the left side, we let k be the child key. (Recall that k is the rightmost extra key in the left of the heterogeneous decomposition.) The right child of k is cl, and the left child consists of the left subtree ll, the key lk and the right subtree $olst$, which was lr before we adjust its skewness by `buildDftAtWalker`. Lastly, we restore Invariant 4.4 by calling `incMergeL'` to rotate the node containing lk in the a position in the case when k is a trivial ancestor key of x. We then pass the job to `buildDftAtRotator` to finish inverting the left-right and the right-left spine of the new root.

```
596 fun reassembleForward x turnP = let
597   val (l,k,_,r) = get4 turnP
```

```
598   val (ll,lk,lh,lr) = get4 l
599   val newRoot = case incMergeR r of                (* restore RS invariant *)
600     B rn => let                                                  (* no rbox *)
601       val (h,olst,nc,_,_) = buildDftAtWalker x lr (X rn) E
602       val ttuple = (X (ll,lk,lh,olst),k,h,nc)
603     in X (rotateL ttuple) end                              (* nc is now root *)
604   | X (B rln,rk,rh,rr) => let
605       val rbox = case rr of X (rrl,_,_,_) => rrl | _ => rr
606       val (h,olst,nc,orst,_) = buildDftAtWalker x lr (X rln) (skewR rbox)
607       val nrr = case rr of X (_,rrk,rrh,rrr) => X (orst,rrk,rrh,rrr)
608                          | _                 => orst
609       val (ncl,nck,nch,ncr) = get4 nc
610     in
611       X (incMergeL' (X (X (ll,lk,lh,olst),k,h,ncl)),nck,nch,X (ncr,rk,H,nrr))
612     end
613   | _ => raise Fail "reassembleForward unknown r pat"
614 in
615   buildDftAtRotator newRoot
616 end
```

4.8 Testing Procedures

We have performed extensive testing with our code using the Standard ML of New Jersey[2], up to the limit allowed by the memory of our experiment machine. Without showing the actual test programs, which can be found in `regressions.sml`, we remark that we have tested our code with up to $N = (2^{13} - 1)$ keys in the following ways:

- ☞ Using a helper function `fullFromReflection` which computes the left subtree of a dynamic finger tree at f by taking the mirror image of the right subtree of the dynamic finger tree at $(N - f + 1)$, we check that the dynamic finger trees built by `buildDftAt` satisfy the symmetry condition.
- ☞ Using `buildDftAt`, we build a dynamic finger tree at each key from 1 to N. Then we use `stepR` to repeatedly step from the dynamic finger tree at 1 to N and check that the results match. We also check `stepL` similarly, and also check that `stepL` is the inverse function of `stepR` by taking a backward step after a forward step at each key from 1 to $(N - 1)$.
- ☞ For all $1 \le i < j \le N$, we build a dynamic finger tree at i using `buildDftAt`, then perform a forward search to j using `searchForward`. Then we compare the output with a dynamic finger tree built at j by `buildDftAt`.

2. http://www.smlnj.org/

In other words, all possible forward searches are tested. We test all possible backward searches similarly.

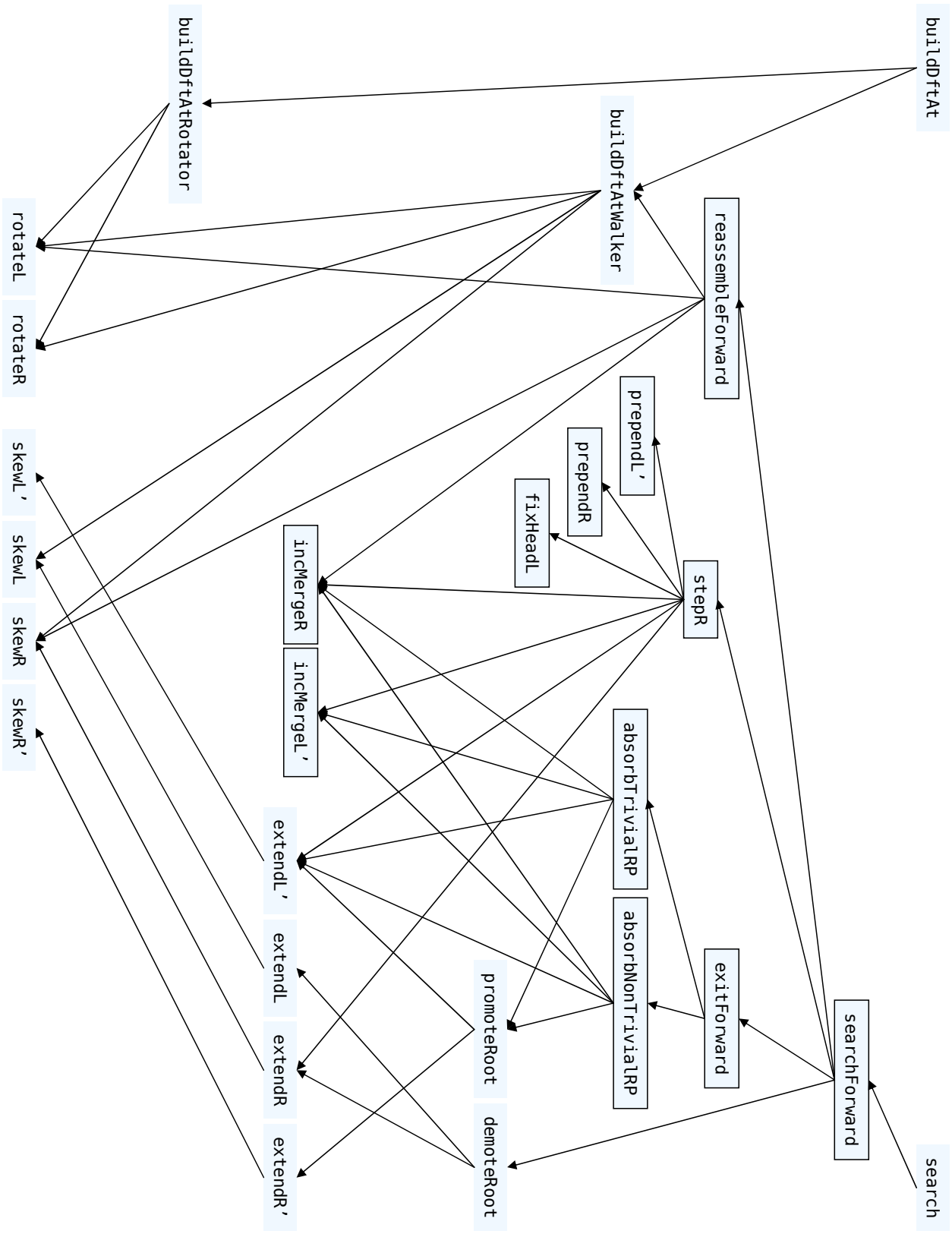


Figure 4.2 – Call graph for the functions involved in the forward direction (backward suppressed by boxes around the forward variant)

5

An $O(\lg \lg n)$-Competitive Dynamic Finger Tree Algorithm

At the time when this thesis was proposed, the author conjectured that multi-splay trees have the dynamic finger property and proposed that heterogeneous decompositions can be a useful perspective to look at the dynamic finger property of tangolike trees. Let us admit upfront that we have *not* been able to prove or disprove our own conjecture. Instead we will use this chapter to report some progress that we have made towards this goal. In particular, we will describe what we believe to be the first $O(\lg \lg n)$-competitive binary search tree algorithm that has the dynamic finger property in the amortized case, which we dubbed "handy tango trees". To simplify our exposition, we will make the following assumption about the reference tree.

Assumption 5.1 The reference tree is a complete binary search tree of height h.

5.1 Tango Trees vs. Dynamic Finger Property

Perhaps the best way to start is to show two existing tangolike tree designs that do not have the dynamic finger property.

5.1.1 Vanilla Tango Trees

It is easy to see that tango trees do not have the dynamic finger property by considering repeated accesses to the smallest key x_1. Observe that the top auxiliary red-black tree T^* that contains x_1 has exactly $\lg n$ keys and it stores x_1 at the deepest level. Consequently, we will need $\Theta(\lg\lg n)$ time to search for x_1 inside T^*, and our easy theorem follows.

Theorem 5.2 Tango trees do not have the dynamic finger property.

We may be tempted to conjecture that for a tangolike tree to have the dynamic finger property, at least the top auxiliary tree must have the dynamic finger property. However, using the understanding we gained from heterogeneous decompositions, we see that the ranks of the keys in the top auxiliary tree are approximately geometrically separated, and as such the dynamic finger budget may in fact be quite large even if the finger does not move over a large number of keys within it. An extreme case is analyzed in Example 5.1.

Example 5.1.

Suppose the left spine of the reference tree is solid and we scan each of these keys once from left to right. Clearly, there is no switches induced. The i-th key has rank 2^{i-1}. We compute that the difference in rank between the $(i+1)$-th key and the i-th is $(2^i - 2^{i-1}) = 2^{i-1}$, which gives a dynamic finger budget of $\Theta(i)$. Scanning the left spine would therefore have a budget of $\Theta(h^2)$, where h is the height of the reference tree. This budget is sufficient even for a search tree that incurs linear time.

The above example actually leads us to the following theorem for access sequences that induce no switches.

Theorem 5.3 Suppose the root solid path ends at a leaf f. Consider the access sequence σ that contains only keys on this path. We can serve σ within the dynamic finger budget using a tangolike design as long as the top auxiliary tree can serve a key x in $O(k)$ time, where k is the number of keys between f and x inside the top auxiliary tree.

Proof. Take the heterogeneous decomposition of the reference tree with respect to f. The runtime condition of the top auxiliary tree simply says that the k-th closest primary key x on the heterogeneous spines can be served in $O(k)$ time.

We lowerbound the dynamic finger budget as follows. Observe that the heterogeneous height of the primary keys in the right heterogeneous

spine is non-decreasing with bounded multiplicity. (Or we can rely on Assumption 5.1, in which case the heterogeneous height of the primary keys are unique and strictly increasing.) Therefore, if x is the k-th key, we know $\mathrm{hh}(x) = \Omega(k)$. By Theorem 2.15, we know that $\lg \mathrm{dist}(f, x) = \Omega(\mathrm{hh}(x))$, which is $\Omega(k)$ from the above. Suppose σ_{i-1} and σ_i are both on the right heterogeneous spine. If $\sigma_{i-1} < \sigma_i$, then $\mathrm{dist}(\sigma_{i-1}, \sigma_i) \geq \frac{1}{2}\mathrm{dist}(f, \sigma_i)$. Otherwise, $\sigma_{i-1} \geq \sigma_i$ and $\mathrm{dist}(\sigma_{i-1}, \sigma_i) \geq \mathrm{dist}(f, \sigma_i)$. Suppose σ_{i-1} and σ_i are on the right and left heterogeneous spines respectively. Then $\mathrm{dist}(\sigma_{i-1}, \sigma_i) \geq \mathrm{dist}(f, \sigma_i)$. In all three cases, the dynamic finger budget is $\Omega(\lg \mathrm{dist}(f, \sigma_i)) = \Omega(k)$ and is sufficient to cover our $O(k)$ cost. □

Observations. Using the static finger theorem of splay trees, Theorem 5.3 shows that multi-splay trees serves any access sequence that induces no switches within the dynamic finger budget. In particular, note that the top splay tree can serve x in $O(\lg k)$ time, which is better than what we need. Therefore, by considering such sequences alone, we cannot conclude that the dynamic finger property is necessary for the top auxiliary tree. However, it does suggest that when an arbitrary access sequence gets into a segment where there is no switches, the keys with lower heterogeneous heights with respect to to the leaf at the current solid path should be made cheaper to access.

5.1.2 Improved Tango Trees

In all fairness, Demaine et al. have already observed that tango trees are not dynamically optimal in their original conference paper [DHIP04]. In particular, they have noted that if we use any non-self-adjusting binary search tree to represent an auxiliary tree, then there must exist a key x at depth $\Omega(\lg \lg n)$ in the top auxiliary tree. Since there are only $O(1)$ switches per access in the access sequence we considered in §5.1.1, repeated accesses to x shows that tango trees are in fact $\Theta(\lg \lg n)$-competitive.

Demaine et al. have subsequently proposed to replace each red-black tree with a certain variant of search tree in the journal version of their paper [DHIP07, p. 243]. The reader may recall that we have reproduced their relatively short description of this variant in full on page 56 and called it the "improved tango trees". Furthermore, the reader may also recall that we have partially quoted this on page 107 as well when we discussed possible candidates of binary search tree algorithms that have the dynamic finger property. Indeed, the description of the "balanced BST" in the quote seems to fit the binary search tree algorithm we presented in §4. This motivates us to represent each auxiliary tree using a generalized version of

our dynamic finger trees with support for splits and joins. (Recall that our design in §4 only handles search.) To avoid any possible confusion, let us call this hypothetical version of dynamic finger trees "generalized dynamic finger trees". It must be stressed that our generalized dynamic finger trees merely represent only one possibility of what the above "balanced BST" might be.

Unfortunately, if we are allowed to make the following two assumptions on how generalized dynamic finger trees may work, we do not get a tangolike design with the dynamic finger property. In particular, our analysis below shows that such a tangolike design incurs $\Omega(\lg n \lg \lg n)$ time per access when serving a certain form of the bit-reversal sequence, which we define next.

Assumption 5.4 Splitting a generalized dynamic finger tree at a key that is d rank away from the finger takes $\Theta(\lg d)$ time.

Assumption 5.5 After splitting a generalized dynamic finger tree at the key x, the fingers in the two resultant generalized dynamic finger trees will be at the predecessor and the successor of x.

Having made these two assumptions, let us consider using a tangolike tree to serve an arbitrary repetition of the bit-reversal sequence. To develop our intuition, we will first look at a particular key f defined as follows. Starting at the root, we alternately descend left and right until we reach a leaf. Let the key in this leaf be f, and let σ_i be an access to f. (If h is 4 as in Table 1.2, then f is 11.) Let the auxiliary tree that contains f at the access σ_i be denoted T_f. Notice that T_f currently has $(h+1)$ keys.

Observe that up until the next access to f, each key on the access path of f will have changed its preference once and each will be due to a different access. Furthermore, the preference changes occur in the top-to-bottom order, i.e., the topmost key (the root) will change its preference first, followed by the second topmost etc. All these are due to the property of the bit-reversal sequence. (Figuratively, we see that the bit-reversal sequence keeps "peeling" the access path of f at the top.)

Suppose the topmost key on the access path of f has switched due to an access sometime after σ_i. In a tangolike design, this corresponds to a split of T_f at this key, which by construction is the rightmost key of T_f. The finger is now at the new rightmost key of T_f by Assumption 5.5 and T_f now contains h keys. Forward into the future, suppose the second topmost key on the access path of f has switched due to another subsequent access. How much time would it take to split T_f at this key, which by construction

is the leftmost key of T_f? By Assumption 5.4, it takes $\Theta(\lg h)$ time and we note that T_f now contains $(h-1)$ keys.

It is not hard to see that f has been chosen so that the lone finger in T_f has to alternate between the two ends. The time we spent on splitting T_f aggregates to $\sum_{i=0}^{h-1} \Theta(\lg(h-i)) = \Theta(h \lg h)$, and this is divided among the h subsequent accesses at an average of $\Theta(\lg h)$ for each of these accesses. Notice that this is the cost incurred due to splitting T_f alone and each of these accesses incur some other costs due to the splitting of other auxiliary trees, among other operations.

One may hope that in general the access paths to most leaves do not see this extreme behavior we exhibited using f. However, we observe that each time when there is a turn in the upper half of an access path to any leaf, we incur a cost logarithmic to the size of the whole path due to Assumption 5.4. In particular, say this is a right turn where we are at a left child and about to descend right. In this case, the finger must have been in the rightmost of the auxiliary tree before we reach the left child, where the finger is now. As we will descend right, this left child is at the leftmost of the auxiliary tree. As we only consider the turns in the top half of the reference tree, this turn will cost us $\Theta(\lg h)$ time. The question is, therefore, how many such turns do we see when serving one repetition of the bit-reversal sequence?

The probabilistic method allows us to compute this easily. Consider the address of a random leaf f. A turn corresponds to the event that the $(i+1)$-th bit differs from the i-th bit in the address. This happens with probability $\frac{1}{2}$ at each bit independently for $1 \le i < h$. Therefore, the expected number of turns in the upper half of the access path of f is $\frac{1}{2} \times \frac{h}{2} = \Theta(h)$, and the cost due to splitting the auxiliary tree that contains f over the course of serving one repetition of the bit-reversal sequence is $\Theta(h \lg h)$. Finally, since the bit-reversal sequence visits every leaf once per repetition, the expected number of turns is the average number of turns when summing over all leaves. In other words, to serve one repetition of the bit-reversal sequence, we incur $\Theta(h \lg h)$ time per access.

Theorem 5.6 Let h be the height of the reference tree and let n be the number of keys in it, i.e., $n = (2^h - 1)$. Any tangolike design using generalized dynamic finger trees satisfying Assumption 5.4 and Assumption 5.5 as its auxiliary trees will incur $\Theta(\lg n \lg \lg n)$ time per access to serve the bit-reversal sequence.

Observations. Theorem 5.6 suggests that having one finger per auxiliary tree may not be sufficient for a tangolike design to have the dynamic finger property. Otherwise, the time bound would have been $O(\lg n)$ instead.

However, we must note that Assumption 5.4 is very strong because it forces us to spend $\Theta(\lg d)$ time to split at d keys away. It is totally conceivable that a tighter analysis is possible on some (stronger) variants of search trees, such as splay trees. Furthermore, after splitting a splay tree, the keys at the roots of the two resultant splay trees do not satisfy Assumption 5.5 as well. If the finger should be considered to be at the two new roots of the splay trees after the split, then we must note that the new finger is most certainly not the predecessor/successor of the splitting key. (Although we have no good intuition on whether the finger should be considered to be at the two new roots or the predecessor/successor of the original root, the latter seems more intuitive and happens to satisfy Assumption 5.5.)

In any case, Theorem 5.6 does not rule out the possibility of multi-splay trees having dynamic finger property at all. However, it does suggest that we should not merely rely on the dynamic finger property of splay trees, for otherwise it is as if we are assuming Assumption 5.4 in the auxiliary trees.

We further remark that when this research was started, it was an open problem whether any *online* binary search tree algorithm can be split into singletons in linear time. For more information, see the discussion at the end of [Col90] as well as [Luc88a] and [Har06, §2.3.7]. However, recently Demaine, Harmon, Iacono, Kane, and Pătraşcu [DHIKP09] have resolved this question by showing how to simulate a heterogeneous $(2,4)$-tree with a binary search tree algorithm. Notice that this simulation is a sufficient but not necessary condition for our proof to fail. In other words, even though this simulation does not give the worst-case bounds as asserted in [DHIP07], it does give us a binary search tree algorithm that has two fingers and thus invalidating Assumption 5.4.

5.2 Wilber I vs. the Dynamic Finger Budget

Upon further investigation, we have identified four obstacles that any tangolike design must overcome in order to have the dynamic finger property. Let us explain these issues before we go on to describe handy tango trees.

5.2.1 Dynamic Finger Budget Can Be Too High

Consider an extreme example where we keep alternating between the smallest and the largest key of the tree. The number of switch per search is exactly 1, but the dynamic finger budget is $\lg n$. If we were to retain $O(\lg\lg n)$-competitiveness, then we only have $O(\lg\lg n)$ time to work with. This already means that the dynamic finger tree in §4 is too slow.

5.2.2 Switching At $\Theta(\lg\lg n)$ Time Is Too slow

Let us first recall that tangolike trees have a very specific structure to guarantee their competitiveness. In particular, each solid path in the reference tree is represented by an auxiliary tree. A switch is then implemented by several splits and joins among the involved auxiliary trees. If we are not aiming for the dynamic finger property, then it suffices to bound each split and join in time that is proportional to the logarithm of the size of an auxiliary tree. But what if we encounter a search sequence such as the alternation between the predecessor and the successor of the root of the reference tree? This sequence has a low $O(m)$ dynamic finger budget, but the two auxiliary trees involved both have size $\Theta(\lg n)$. If a switch requires time that is proportional to the logarithm of the size of the auxiliary trees, then we are already off by a $\Theta(\lg\lg n)$ factor from the dynamic finger budget.

5.2.3 Must Avoid The Dynamic Optimality Dragon

Now it should be clear that we must more tightly analyze the running time of a switch, but it turns out we must also be careful not to *hope* for an analysis that is too tight in the following pragmatic sense. Recall that the number of switches of a search sequence is a lowerbound on the running time of any binary search tree algorithm on that sequence, and that we have seen binary search tree algorithms that achieve the dynamic finger budget of any search sequence. A simple but useful consequence of the above is that the number of switches of a search sequence is upperbounded by the dynamic finger budget of that sequence. It also follows that the bit-reversal sequence, which induces $\Omega(n\lg n)$ switches, has a dynamic finger budget of $\Theta(n\lg n)$. Therefore, when we serving this sequence, we must implement each switch in $O(1)$ time. However, realize that any auxiliary tree that supports switching in $O(1)$ time regardless of the search sequence would immediately give a tangolike design that is dynamically optimal! Therefore, we must aim for an analysis that is tight enough for access

sequences like bit-reversal, but not so tight lest we are attempting to solve the even harder problem of dynamic optimality.

5.2.4 Amortization Is Necessary

Consider the sequence where we first scan through the n keys, followed by a search to the predecessor of the root and then a search to the successor. The dynamic finger budget for the last search is $\Theta(1)$. However, observe that we have set up every key on the left spine of the right subtree of the root to prefer to the right. Therefore, to get to the successor of the root, we have exactly $\lg n$ switches to perform. In other words, any tangolike design that represents a solid path using an auxiliary tree cannot have the dynamic finger property in the worst-case.

5.3 Handy Tango Trees—The Leaf-Only Case

As we will explain in the future writeup of handy tango trees, search targets that are in the junctions of the reference tree creates a complication that is best treated separately. For now, we merely note that it is an artifact due to how the typical search algorithm in a tangolike design interacts with the auxiliary tree tree we will use here. To separate this concern as cleanly as possible, let us handle the case when all search targets are in the leaves of the reference tree. This is formalized by making the following assumption within this section.

Assumption 5.7 All search targets are in the leaves of the reference tree.

The following recursive definition captures an intuitive concept on the reference tree when Assumption 5.7 holds. We will be making use of this concept in this chapter.

Definition 5.8 Consider a subtree T in the heterogeneous decomposition of the reference tree with respect to any key f. We say that the subtree T is leaf-decomposed iff (i) the most-recently searched target x in T is at a leaf of T, and (ii) each of the subtrees in the heterogeneous decomposition of T with respect to x is itself leaf-decomposed.

We remark that f in the above definition can be any key in the reference tree, i.e., it can be either in a leaf or in a junction. To check our understanding of this definition, observe that if a subtree in a given heterogeneous decomposition is leaf-decomposed, then the most-recently searched target on *each* solid path in that subtree is in the leaf of that path (hence the name). The reader is invited to contrast this with the situation when Assumption 5.7 does not hold.

5.3.1 Auxiliary Tree

Recall that an auxiliary tree in a tangolike design corresponds to a solid path in the reference tree. For now, we will focus on one such solid path and show how it is represented.

Let f be the key at the leaf of the considered path. We categorize the keys on this path into five groups, with the first group being the key f itself. The next two are the trivial and nontrivial right ancestors of f in the solid path, and the remaining two the corresponding groups among the left ancestors of f. Notice that every key on the solid path is in exactly one of these five groups.

Our auxiliary tree design is in fact a *segmented* representation of a variant of the left and right parent stacks of f. Recall that the right parent stack of f is a stack in which f is at the top and each non-top cell contains the right parent of the key in the cell above. By considering each nontrivial ancestor in the stack as a partition point, the trivial ancestors will be organized into contiguous but possibly empty segments.

On the schematic level, our auxiliary tree is defined as the following three-layer structure.

(1) The top layer contains only the root, which contains the key f.
(2) The middle layer are two heterogeneous red-black trees, one on each side of the root. The one on the right stores the nontrivial right ancestors of f. We call this tree the "nontrivial subtree" of the auxiliary tree.
(3) The bottom layer consists of two collections of heterogeneous red-black trees, hanging from the leaves of the two nontrivial subtrees. Each of these trees stores the corresponding segment of the trivial ancestors of f, and will be referred to as a "trivial subtree".

Of course, either of the two nontrivial subtrees or any one of the trivial subtrees may be empty. This brings us to the first of several technical issues in our auxiliary tree design.

5.3.1.1 Root Bits, Trivial Bits, and Nontrivial Bits

Just as in any tangolike design, the root of each auxiliary tree will be marked to distinguish them from the other nodes so that we know when we have traversed from one auxiliary tree into another. This is achieved by storing a root bit in each node and mark it at the roots of the auxiliary trees.

For handy tango trees, we must also mark the root of each heterogeneous red-black tree using a separate nontrivial bit so that we know whether the heterogeneous red-black tree we are traversing is a trivial sub-

tree or not. This is crucial because some of these heterogeneous red-black trees are in fact empty and will leave no trace in a handy tango tree at all. As an example, suppose f is the smallest key in a tall reference tree. Since f does not have any nontrivial right ancestor, the heterogeneous red-black tree at its right child position is in fact representing the trivial right ancestors of f. By keeping track of the nontrivial bit when we descend from f, we know we have skipped over an empty nontrivial subtree.

As it turns out, the nontrivial bit can be implemented in multiple ways. In this thesis, we allocate two extra bits at each node—the nontrivial bit and the trivial bit. One of these bits will be marked at the roots of the heterogeneous red-black trees appropriately.

5.3.1.2 Augmentation

Similar to tango trees and multi-splay trees, each node w in the auxiliary tree is also augmented to store an extra *reference depth* field. This field stores the depth of the key of w in the reference tree, denoted $\mathit{refdep}(w)$. Since we are dealing with a static reference tree as per Assumption 5.1, the content of this field does not change.

The augmentation in multi-splay trees requires two fields and can be particularly tricky to use correctly. Handy tango trees are more intuitive in this regard for two reasons: (i) we store the left and right ancestors separately, and (ii) we search from the two ends of an auxiliary tree, instead from the root. We remark that the second point above affords us great convenience. Consider a heterogeneous red-black tree on the right hand side of an auxiliary tree. Observe that as we scan from left to right in this tree, the reference depth of the keys decreases. In other words, the keys in this tree are in fact also *sorted* in their reference depth, albeit in the reverse order. (The situation is even simpler if we consider a heterogeneous red-black tree on the left hand side of an auxiliary tree.) Consequently, any comparison-based algorithm that operates on the keys of the tree can also operate on the reference depths of the keys simply by changing the comparison function only. For example, this means we can split the tree at a given reference depth, just as we can split the tree at a given key. We will be using this handy ability frequently in what follows.

5.3.1.3 Binary Search Tree Simulation

Lastly and most importantly, notice that we have been describing the auxiliary trees as if heterogeneous red-black trees are in the binary search tree model. Of course, this is not literally true since heterogeneous red-black trees are usually understood as red-black trees with inverted spines and

we search such trees from its two ends. Fortunately, recall that in §1.3.3, we presented a recent result by Demaine et al. [DHIKP09, Lemma 2.4] that allows us to simulate a heterogeneous red-black tree by a binary search tree algorithm with only constant slowdown. Indeed, we will be using this simulation for the rest of this chapter and continue to speak as if we are operating on the heterogeneous red-black trees directly. We do caution that the trivial and nontrivial bits are to be marked on the roots of the binary search trees that are used in the simulation. (The actual root of the heterogeneous red-black tree is buried deeply in the binary search tree simulation.)

5.3.2 Switch

As usual in a tangolike design, the search algorithm is structured around the switch primitive. In this section, we will deal with how to switch a given auxiliary tree T at one of its key w. Let f be the key at the root of T.

First of all, notice that by Assumption 5.7, w cannot be f since we will never switch at a leaf of the reference tree. Now observe that if w is to the right of f, then this can only be a left-to-right switch because w is a right ancestor of f. If w has a left parent in the reference tree, then we will denote its key by lpw. Observe that lpw is the rightmost key to the left of f that has a reference depth smaller than that of w. Furthermore, lpw must be a nontrivial left ancestor of f, since f is in the left reference subtree of w. In the case when w does not have a left parent in the reference tree, let lpw denote $-\infty$ in what follows. Similarly, we define rpw as the right parent of w in the reference tree if it exists, or ∞ otherwise.

Intuition. In the case of a left-to-right switch, we would like to separate out the keys in T that are below w in the reference tree, and replace them with another set of keys that corresponds to the keys in the solid path of the right reference subtree of w. Notice that the these two sets of keys are in the open intervals (lpw, w) and (w, rpw) respectively. If the parent stacks are represented by a non-segmented representation such a list, then the switch algorithm would be straighforward. However, we are using a segmented representation and this adds a bit of complexity to our algorithm, which we describe in §5.3.2.1–§5.3.2.5.

5.3.2.1 RHS Splits

If w is a nontrivial ancestor, then it is in the nontrivial subtree RN of T. We split RN at w to obtain RN_l, w, and RN_r.

Otherwise, w is in a trivial subtree RT. We split RT at w to obtain RT_l, w, and RT_r. If RT has a parent subtree in T, then it must be the nontrivial subtree RN. We further split RN at w to obtain RN_l and RN_r. Finally, we hang RT_l and RT_r at the rightmost and the leftmost of RN_l and RN_r respectively.

5.3.2.2 Existence Test of lpw

Before we can deal with the left hand side, first we need to give a test for the existence of lpw in T. Fortunately, this is easy since the left ancestor keys get deeper as we scan them from left to right. Therefore, the left parent of w is in T iff the left subtree of T is not empty and its leftmost key has a reference depth that is smaller than that of w. Since we are using heterogeneous red-black trees as components to store the left subtree of T, the above can be tested in worst-case $O(1)$ time in all three possible cases.

5.3.2.3 LHS Splits

If lpw exists inside T, then we also need to split the left hand side of T using lpw. This is performed in a fashion similar to the right hand side. The only difference is that this time we are splitting at the reference depth of w using the augmentation field, which gives us the effect of splitting at lpw. (In fact, we do not know the value of lpw before the splitting.) Let the variables LN_l, LN_r, LT_l, and LT_r denote their corresponding parts.

5.3.2.4 Auxiliary Tree of f

To obtain the auxiliary tree for f after the switch, first we let f be the root of the resulting auxiliary tree. The right subtree of f is RN_l if it exists, or RT_l if otherwise. We should note that none of the keys in RN_l and RT_l will change their triviality nature after the switch and hence no adjustment is needed. The left hand side is treated symmetrically.

5.3.2.5 Auxiliary Tree of x

The auxiliary tree T_x of x before the switch is rooted at the external position immediately to the right of w before the splits. After the splits, this position is now the leftmost external position of RT_r if exists. Otherwise, it is either at the leftmost external position of RN_r, or the leftmost external position of the leftmost trivial subtree underneath RN_r. In all three cases, the root of T_x—containing x itself—can be reached in worst-case $O(1)$ time.

At this point, we will proceed to join the remaining pieces of T together with T_x in the exact opposite order of how we split T. However, some care

must be taken in the joins because some keys will change their triviality nature after the switch.

In particular, lpw will become a trivial left ancestor if it is the parent of w in the reference tree. This can be tested by checking if the reference depths of the two differ by one.

Similarly, w will become a trivial left ancestor if x is to the right of the root r of the right reference subtree of w. To test for this, notice that r is either the rightmost or the leftmost ancestor of x depending on the relative order between r and x. In either cases, r can be reached in worst-case $O(1)$ time in T_x by noting that r has the smallest reference depth among all keys in T_x.

Fortunately, all of the above happen at the end of some heterogeneous red-black trees and we only need to pick the correct tree to join when we consider lpw and w.

5.3.3 The Search Algorithm

In any tangolike design, it is straightforward to search for a given target x by descending through the auxiliary trees. But during the descend, the search algorithm must also identify the correct key to switch and the direction of the switch in each auxiliary tree it traverses. Once x is reached, the algorithm will then switch these keys bottom-up. Below we will describe how the search algorithm identifies the key and the direction of the switch in an auxiliary tree T. But first, let us define a useful notation that we will be using throughout the analysis.

5.3.3.1 Segmented Notation

We will be frequently dealing with the rank of a key in the two subtrees of an auxiliary tree. In the most general case, each of these subtrees consists of a nontrivial subtree at the top and a collection of trivial subtrees on the bottom. To make it convenient to reason about the rank of a key in this segmented representation, we define the following notation using a tuple.

Definition 5.9 Consider an auxiliary tree T and let R be the right subtree of T. Assuming the right nontrivial subtree of T exists.

- ☞ The $(i,0)$-th key from the left and right of R is the i-th key in the nontrivial subtree, counting from the left and right respectively.
- ☞ The (i,j)-th key from the left of R is, counting from the left, the j-th key of the trivial subtree hanging at the i-th leftmost external position of the nontrivial subtree.

☞ The (i, j)-th key from the right of R is, counting from the right, the j-th key of the trivial subtree hanging at the i-th rightmost external position of the nontrivial subtree.

If the right nontrivial subtree of T does not exist, then the last two definitions above degenerate to the case where i is always 0.

To make sure we understand this definition and facilitate reuse, we now prove the following simple lemma.

Lemma 5.10 Suppose w is the k-th leftmost (rightmost) right ancestor of f in T and w is also the (i, j)-th key from the left (right) of the right subtree of T. Then both i and j are no larger than k.

Proof. First of all, observe that i cannot be larger than k since in the worst-case every trivial subtree is empty. Now if $i = 0$, then $j = k$, or else $i > 0$ and we have $j < k$. □

5.3.3.2 Switch Direction

Recall that a switch happened in the reference tree once we descend into an external position of T. In handy tango trees, it is easy to determine the direction of the switch because our auxiliary trees store the left and right ancestors separately. In particular, it is a left-to-right switch iff the external position is to the right of the root of T.

5.3.3.3 Switch Key

Suppose we have to perform a left-to-right switch in T and let f be the root of T. As this is a left-to-right switch, we know that $f < x$ and the switch key is the lowest common ancestor w of f and x in the reference tree. It follows that w is the largest right ancestor of f that is no larger than x.

Our auxiliary tree design together with Assumption 5.7 makes it particularly easy to find w in T. Suppose w is the (i, j)-th key from the left in the right subtree of T. What we want is the key at the lexicographically largest (i, j) position.

High Level Search. Consider the most general case when w is in a trivial subtree underneath the nontrivial subtree. We first search for x in the nontrivial subtree, while keeping track of the largest key nw that is no larger than x in this tree. Since x is not in T, we will reach an external position, in which we find a trivial subtree. Similarly we search for x in the trivial subtree, while keeping track of the largest key tw that is no larger than x in this tree. Eventually we will reach yet another external position.

If we find any key in the trivial subtree that is not larger than x, then tw is w. Otherwise, every key in the trivial subtree is larger than x, and we

conclude that nw is w. What we need is the low level search primitive that allows us to identify nw and tw.

Low Level Search. Below we will describe how to search for the largest key that is no larger than a given target x from the left end of a heterogeneous red-black tree. The search from the right end is similar.

The idea is to keep track of our candidate by means of a pointer, which initially points at the leftmost node of T. Each time we ascend from a node, we place the pointer at the node we ascend from. Note that in the ascend phase, we either reach the root of T or not. If so, then we know that the external position is not in the left subtree of T and we simply terminate. Otherwise, we will start descending from a node on the left spine of T. Each time we descend right from a node, we place the pointer at the node we descend from. Eventually we will reach an external position and the pointer will point to w.

The correctness of this algorithm hinges on two facts:

(1) Assumption 5.7 guarantees that we must reach an external position.
(2) By definition w is the left parent of the external position and the pointer is updated each time we descend right.

5.3.4 $O(\lg\lg n)$-Competitiveness

To show that handy tango trees are $O(\lg\lg n)$-competitive, we merely need to show that each switch takes amortized $O(\lg\lg n)$ time. Allowing for an $O(n)$ potential drop due to amortization, the competitiveness follows from Wilber's lowerbound.

Theorem 5.11 In a handy tango tree, each switch takes amortized $O(\lg\lg n)$ time.

Proof. Each switch involves an $O(1)$ number of splits and joins. Each split involves a heterogeneous red-black tree that is $O(\lg n)$ in size and therefore runs in amortized $O(\lg\lg n)$ time. Each join takes amortized $O(1)$ time. □

Although the above analysis already suffices to establish the $O(\lg\lg n)$-competitiveness, we still need more refined estimates to establish the dynamic finger property.

5.3.4.1 Free Joins

To handle the running time incurred by the joins, let us amortize the cost of joins into that of the splits in heterogeneous red-black trees using the following deductions.

(1) Modulo the $O(n)$ potential drop, the time spent in the joins is proportional to the number of joins since each join only takes amortized $O(1)$ time.
(2) The number of joins is in turn proportional to the number of switches, which is a lowerbound on the running time of any binary search tree algorithms.
(3) This lowerbound is of course asymptotically no higher than any upperbound we can prove on the running time of any binary search tree algorithm.

In other words, the time spent in the joins will be fully absorbed by the constant in the Big-O notation of an upperbound, which we can safely assume to contain an additive $O(n)$ term to account for the size of the data structure. We note that some authors may say that each join is amortized $O(0)$ time, although we will not be using this notation.

5.3.4.2 Costly Splits

Before we go on, we will also state and prove a refinement of Theorem 5.11 as the following benign-looking lemma.

Lemma 5.12 Consider a left-to-right switch on the key w in an auxiliary tree T rooted at f. If w is the (i,j)-th key from the left (right) of the right subtree of T, then the switch takes amortized $O(\lg i + \lg j)$ time.

Proof. In the worst-case, there are two splits in the right subtree of T and in total they take amortized $O(\lg i + \lg j)$ time. But we still have to split the left subtree of T at the left parent lpw of w. We claim that lpw is either the $(i,0)$-th or $(i+1,0)$-th key from the right (left) of the left subtree of T.

To see this, first observe that lpw is a nontrivial left ancestor of f because f is in the left subtree of w. Thus the second position of the tuple is indeed always 0.

Next we consider the nontrivial ancestors bottom-up (top-down). Observe that between two consecutive nontrivial right ancestors u and v, there must be a left nontrivial ancestor because (i) we must have descended left and then immediately right at both u and v, thereby implying the existence of a chain of left ancestor(s) between u and v, and (ii) the lowest left ancestor on this chain is a nontrivial left ancestor. In other words, the nontrivial left and right ancestors must appear in alternation.

Therefore, depending on whether f is a left or right child, lpw is respectively either the $(i+1,0)$-th or $(i,0)$-th key from the right (left) of the left subtree of T. It then follows that the two splits at lpw take amortized $O(\lg i)$ time. □

5.3.5 No Working Set Property

Before we prove that handy tango trees have the dynamic finger property, we must point out that these trees do not have the working set property. The weakness of handy tango trees is precisely the overly-rigid structure in the heterogeneous red-black trees.

Theorem 5.13 Handy tango trees do not have the working set property.

Proof. We give a simple alternating search sequence that incurs $\Omega(\lg \lg n)$ time in handy tango trees.

Recall that h is the height of the reference tree and let h be large. We define the key f as the leaf reached from the root by alternating between left and right at each descend. Let w be the right nontrivial ancestor of f at depth $\lceil \frac{h}{2} \rceil$. By picking the initial descend direction from the root, we can ensure that w is a right child of its parent. It follows that f is in the left subtree of w. Let x be the key reached from the root of the right subtree of w by first descending left and then alternating between right and left at each descend.

Notice that both f and x have $\Theta(h)$ nontrivial ancestors on both left and right. Furthermore, w is a $\Theta(h)$-th nontrivial ancestor of both f and x, each at its corresponding side.

The search sequence is simply an alternation between f and x. The working set budget for each search is clearly $\Theta(1)$. However, notice that since w is basically in the middle of the corresponding nontrivial subtree at each switch, it takes $\Omega(\lg \lg n)$ time to split at w. □

The above, of course, means that handy tango trees are not dynamically optimal. In contrast, multi-splay trees do have the working set property [Wan06, Corollary 3, p. 34], among several others.

5.3.6 Dynamic Finger Property

We will now analyze the running time of handy tango trees from the perspective of the dynamic finger property. Consider the heterogeneous decomposition of the reference tree with respect to to a key f and suppose we are searching for a key x to the right of f. By Assumption 5.7, x is in one of the subtrees T_w^R in the heterogeneous decomposition. Our analysis consists of three parts and each part corresponds one of the following groups of switches induced by the search of x.

Group I The topmost switch at w in the reference tree will be analyzed on its own. Note that this is the only switch that does not occur inside T_w^R.

Group II The right-to-left switch(es) on the left spine of T_w^R will form the second group. The size of this group ranges from zero to one less than the height of T_w^R. The latter is because every key on the left spine could have preferred to the right but we do not count the leaf.

Group III The topmost left-to-right switch on the left spine of T_w^R forms the third group. Note that this group can be empty.

Group IV The remaining switch(es) will form the fourth group. The size of this group ranges from zero to two less than the height of T_w^R. The latter can be justified similar to the second group, except that we do not count the switch that occurs on the left spine.

Switch Classification. Before we proceed, let us give some intuition on the seemingly-obscure definitions above and justify that we have included every switch in one of the groups. We claim that the switches in these four groups occur in increasing reference depth. In other words, if we were to search for x from the root of the reference tree, then the switches induced by the search occur from group I to IV in order.

The first group is obvious from its definition. Now consider the following question: how many left-to-right switches can there be on the left spine of T_w^R? We claim that there can be at most one. Suppose we consider the keys on the left spine from top to bottom. The very first time when such a switch happens, say at key w', we will have already descended right from there. In other words, we have departed from the left spine. Therefore, there cannot be a switch at any key on the left spine that is below w'. And if this switch happens, it is the only possible switch in the third group. (Indeed, the word "topmost" in the definition of the third group is redundant.)

The second group consisting of right-to-left switches is the most interesting. By the same argument above, all the switches on the left spine occur above the only possible left-to-right switch at w'. Clearly, they are all right-to-left switches and belong to the second group. Any remaining switches must not occur on the left spine and will go into the fourth group as claimed.

5.3.6.1 Group I

Recall that w is in fact the exit key in the search for x. Suppose w is the k-th nearest right ancestor of f. Observe that the heterogeneous height increases at each right ancestor because of Assumption 5.1. Therefore, $\mathrm{hh}_f(w)$ is at least k. By Theorem 2.6, k is $O(\lg d_i)$. Applying Lemma 5.10 on k to obtain i and j, followed by Lemma 5.12 immediately yields the lemma below.

Lemma 5.14 The switch in the first group runs in amortized $O(\lg \lg d_i)$ time.

5.3.6.2 Group II

Bounding the time spent by the switches in this group require two steps.

First Step. First we show that each of the switches runs in amortized $O(1)$ time. Of course, this is in general not possible with our current understanding of binary search trees. However, the following two properties hold for each switch in this group.

Let T be the auxiliary tree that contains the switching key w', let rpw' be the right parent of w' in the reference tree if exists, and let y be the leaf of the solid path represented by T. Recall that in a right-to-left switch, we would split T at w and rpw', if exists.

(1) The switching key w' is the leftmost key in the solid path represented by T. Observe that w is either the leftmost key of the left nontrivial subtree of T, or the leftmost key of the leftmost trivial subtree.
(2) The key rpw' is the rightmost nontrivial right ancestor of y and hence it is the rightmost key of the right nontrivial subtree of T.

As both w' and rpw' are at the extreme ends of their corresponding heterogeneous red-black trees, we have the following lemma.

Lemma 5.15 Each switch in the second group runs in amortized $O(1)$ time.

Remark 5.16. Note that rpw' itself may have an unbounded number of right ancestors. However, they are all trivial right ancestors of y and are stored in a segment to the right of rpw'. This is a trick afforded by our segmented representation.

Second Step. Now that we have shown Lemma 5.15, the second step is to absorb the running time incurred by the switches in this group into the constant in the Big-O notation. The argument is exactly the same as the "free" joins in §5.3.4.

5.3.6.3 Group III

Recall that if this group is not empty, its only member is the left-to-right switch at w' on the left spine of T^R_w. Let the auxiliary tree that contains w' be T, and let the leaf of the solid path represented by T be v. Observe that v is to the left of w' because w' currently prefers to the left.

Recall that we are searching for x and the most-recently search target is f. We claim the analysis in for the left-to-right switch in Group I applies

here as well. The reason is simply because v is to the right of f, and so $\text{dist}(f, x) \geq \text{dist}(v, x)$. By letting w' be the k-th nearest right ancestor of v, we have the following lemma.

Lemma 5.17 The switch in the third group runs in amortized $O(\lg \lg d_i)$ time.

5.3.6.4 Group IV

This group of switches are in fact easy to handle within the dynamic finger budget. Observe that there are at most $(\text{ah}(w') - 2)$ switches in this group, and more importantly that the dynamic finger budget is $\Omega(\text{ah}(w'))$ because of the left reference subtree of w'. We claim that the running time incurred by this group is $O(\text{ah}(w'))$. The analysis below is similar to a proof by Demaine et al. [DHIP07, p. 243].

Suppose there are a total of p switches in this group. Let us follow the reference path from the root of right subtree of w' to x and number the switches top-down. Observe that there are $(p+1)$ solid paths on this path. Let r_ℓ for $\ell \in [1, p]$ be the number of keys we traversed before the ℓ-th switch. It follows that

$$\sum_{\ell=1}^{p} r_\ell \leq \text{ah}(w') - 1 = O(\text{ah}(w')). \tag{5.1}$$

We claim that the ℓ-th switch takes amortized $O(\lg r_\ell)$ time. By symmetry, assume this is a left-to-right switch. Let T_ℓ be the auxiliary tree in the switch and let v_ℓ be the leaf of the solid path represented by T_ℓ. Suppose the switch key is the k-th *farthest* right ancestor of v_ℓ. Our claim follows by applying Lemma 5.10 and Lemma 5.12.

Totalling the p switches, we have incurred amortized $\sum_{\ell=1}^{p} O(\lg r_\ell)$ time. By the concavity of the lg function and (5.1), this sum is maximized when each r_ℓ is the same. This gives us an upperbound of $O(p \lg \frac{\text{ah}(w')}{p})$. By calculus, this is maximized when $p = \frac{\text{ah}(w')}{2}$ and the value is $O(\frac{\text{ah}(w')}{2})$. Our claim follows and we have the lemma below.

Lemma 5.18 The switches in the fourth group run in amortized $O(\lg d_i)$ time.

Theorem 5.19 Handy tango trees have the dynamic finger property in the leaf-only case.

Proof. This follows immediately from Lemma 5.14, Lemma 5.15, Lemma 5.17, and Lemma 5.18. □

A

Reprint of CMU-CS-02-184

THE FOLLOWING PAGES are directly imported from [BMW02], which is the full version of [BMW03] that appeared in SODA 2003. At the moment, this the most complete writeup of the hands. We intend to furnish a more complete version of §3 in the final publication of this thesis.

Space-Efficient Finger Search on Degree-Balanced Search Trees

Guy E. Blelloch Bruce M. Maggs
Shan Leung Maverick Woo

September 2002
CMU-CS-02-184

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

An extended abstract of this paper has appeared in ACM-SIAM Symposium on Discrete Algorithms (SODA) 2003. This paper was last revised on April 15, 2003.

Abstract

We show how to support the finger search operation on degree-balanced search trees in a space-efficient manner that retains a worst-case time bound of $O(\log d)$, where d is the difference in rank between successive search targets. While most existing tree-based designs allocate linear extra storage in the nodes (e.g., for side links and parent pointers), our design maintains a compact auxiliary data structure called the "hand" during the lifetime of the tree and imposes *no* other storage requirement within the tree.

The hand requires $O(\log n)$ space for an n-node tree and has a relatively simple structure. It can be updated synchronously during insertions and deletions with time proportional to the number of structural changes in the tree. The auxiliary nature of the hand also makes it possible to introduce finger searches into any existing implementation without modifying the underlying data representation (e.g., any implementation of Red-Black trees can be used). Together these factors make finger searches more appealing in practice.

Our design also yields a simple yet optimal in-order walk algorithm with *worst-case* $O(1)$ work per increment (again without any extra storage requirement in the nodes), and we believe our algorithm can be used in database applications when the overall performance is very sensitive to retrieval latency.

This work was supported in part by the National Science Foundation under grants CCR-0205523 and CCR-9900304 and also through the Aladdin Center (www.aladdin.cs.cmu.edu) under grants CCR-0085982 and CCR-0122581. The second author is also affiliated with Akamai Technologies.

Keywords: Data Structure, Balanced Search Trees, Finger Search

1 Introduction

The problem of maintaining a sorted list of unique, totally-ordered elements is ubiquitous in Computer Science. When efficient element *access* (insert, delete, or search) is needed, one of the most common solutions is to use some form of balanced search trees to represent the list. Over the years, many forms of balanced search trees have been devised, analyzed and implemented.

Balanced search trees are very versatile representations of sorted lists. In addition to providing element access in logarithmic time, certain forms also allow efficient aggregated operations like set intersection and union. For example, Brown and Tarjan [6] have shown a merging algorithm using AVL trees [1] with an optimal $O(m \log \frac{n}{m})$ time bound, where m and n are the sizes of the two lists with $m \leq n$.

Their merging algorithm is, however, "not obvious and the time bound requires an involved proof" [7, p. 613]. As such, in their subsequent paper, Brown and Tarjan [7] proposed a new structure by introducing extra pointers into a 2-3 tree [2] and called it a *level-linked 2-3 tree*. The merging algorithm on level-linked 2-3 trees is simple and intuitive and it uses the idea of finger searches, which we will define shortly. But there is a trade-off in this design. Each node in a level-linked 2-3 tree contains not only a key and two child pointers, but also a parent pointer and two side links. Considering this relatively high space requirement and the elegance of their simple yet optimal merging algorithm, it is natural to wonder if finger searches can be supported in a more *space-efficient* manner on any existing balanced search trees such as 2-3 trees. This is the motivation of our work.

Finger search. Consider a sorted list A of n elements $a_1, \ldots, a_n$ represented by a search structure. Let the *rank* of an element be its position in the list and let $\delta_A(a_i, a_j)$ be $|\, i - j \,|$, i.e., the difference in the ranks of a_i and a_j w.r.t. the elements of A. We say that the search structure has the *finger search property* if searching for a_j takes $O(\log \delta_A(a_i, a_j))$ time, where a_i is the most recently found element. The time bound can be worst-case, expected-case or amortized and we will distinguish them explicitly when needed. (As usual, we let $\log x$ denote $\log_2 \max(2, x)$ and we will simply say $O(\log d)$ when the elements a_i and a_j are not made explicit.)

A *finger* is a reference to an element in the list and historically it is often realized by a simple pointer to an element. (Indeed some papers mandate this representation in their definitions, e.g., see [5].) Typically, we maintain the invariant that the finger is on the most recently found element and we refer to this element as the "current" element. The finger search operation uses the finger as an extra hint to search for its new target and also shifts the finger to the element found. (Section 2 has a precise definition that is appropriate when the search target is absent from the list.) In the worst scenario, finger searching matches the $O(\log n)$ time bound of a classical search; but in applications like merging where there is a locality of reference in the sequence of search targets, finger searching yields a significantly tighter time bound.

Finger search was introduced on a variant of B-trees [3] by Guibas et al. [10] in 1977. Since then, finger search based on modification of balanced search trees has been studied by many researchers, e.g., Brown and Tarjan [7, 2-3 trees], Huddleston and Mehlhorn [12, (a, b) trees], Tsakalidis [23, AVL trees], Tarjan and Van Wyk [22, heterogeneous finger search trees] and Seidel and Aragon [20, treaps]. In their original paper on splay trees, Sleator and Tarjan [21] conjectured that the splay operation has the finger search property. Known as the Dynamic Finger Conjecture, it was subsequently proven by Cole [8]. There are other designs that are not entirely based on balanced search trees as well. For example, Kosaraju [15] designed a more general structure with the finger search property using on a collection of 2-3 trees. Skip Lists by Pugh [19] also support finger searching. More recently, Brodal [5] has investigated finger search trees designed to improve insertion and deletion time. Of special note are the purely-functional catenable sorted lists of Kaplan and Tar-

jan [13]. Their design not only has the finger search property, but it also requires very little space overhead. We will contrast our design with theirs in Section 6.

Challenges and results. Supporting finger search in balanced search trees can be challenging. The main difficulty is in shifting the finger fast enough to achieve a *worst-case* $O(\log d)$ time bound. Observe that if we have to strictly adhere to the unique path induced by the tree, then two elements with similar rank can be stored far apart. As an extreme example, consider the root element and its successor: the tree path has length $\Theta(\log n)$, but we only have $O(1)$ time.

One way to circumvent this apparent difficulty is to store extra information in the nodes so that we do not have to adhere to the tree path. For example, this approach has been taken by Brown and Tarjan [7] who added a parent pointer and two side links to each node. (Side links are pointers to the previous and next node at the same depth.) With these extra pointers, it can be shown that there exists a path of length $O(\log d)$ between two nodes differing in rank by d. Finger search can now be supported by taking this new path. The problem with this design is that a total of $3n$ extra pointers are introduced and the size of the tree is doubled, assuming the key has the same size as a pointer. In fact, among the many other tree-based designs mentioned above, this $O(n)$ extra storage requirement is a common trait.

Our design is an attempt to avoid this $O(n)$ storage requirement but at the same time retain the structural simplicity of balanced search trees. To this end, we base our design on *degree-balanced* search trees [18][1] and we assume a compact k-ary node with only $(k-1)$ keys and k child pointers. Since any extra storage we need must be stored in some *auxiliary* data structures outside of the tree, our goal is to minimize the amount of auxiliary storage while supporting the finger search operation in worst-case $O(\log d)$ time.

As we will show in this paper, our design requires $O(\log n)$ space on a degree-balanced search tree with n nodes and supports finger searches in worst-case $O(\log d)$ time. The finger searches can go in both forward and backward directions without any restriction. We also show that once the finger has been placed on the position of change, insertions and deletions can be implemented in time proportional to the number of structural changes in the tree. This allows us to transfer any results previously proven on these two operations, such as an amortized $O(1)$ time bound and the actual distribution of work at different depths of the tree [12]. In the development of our finger search algorithm, we also obtain a simple in-order walk algorithm with worst-case $O(1)$ work per increment. We believe that this improvement over the previous amortized $O(1)$ bound can be used in database applications when the overall performance is very sensitive to retrieval latency. (The focus of this paper is not on database applications, but we have documented our idea in Appendix C.)

Design overview. We notice that if supporting finger searches is really possible under our storage restrictions, then we must be able to support a special case of it: an in-order walk with *worst-case* $O(1)$ work per increment. Our solution is to eagerly schedule the in-order walk and walk the path in advance. We call this the *eager walk* technique. Because we can only see a constant number of nodes at a time, we also need to keep track of our progress and so we have devised a simple data structure called the *hand* for this purpose. We will document these two ideas along with our in-order walk algorithm in Section 3.

Having solved the in-order walk problem, we then go back to finger searches. Notice that in the in-order walk, the future search targets are known in advance. However, this is not true in finger searches. Our understanding of eager walk suggests that we want to start shifting the finger *before* the actual search target arrives. For finger searches, that means we want to cache some portion of

[1] Apparently the term *degree-balanced* search trees was coined in this monograph. However, it does not cover the details of such search trees. See other references in Section 2 for more information.

the tree so that when the actual search target arrives, the cache will contain a prefix of the path from the finger to the target. If the length of the prefix is chosen to be long enough, then we will be able to finish shifting the finger over the rest of the path in $O(\log d)$ time. As it turns out, the hand is precisely such a cache despite being initially designed just for the in-order walk. At this point, we will also bring in a connection between the hand and the inverted spine technique used in heterogeneous finger search trees by Tarjan and Van Wyk [22]. Using this connection, our finger search algorithm becomes relatively straightforward. Section 4 will be devoted to presenting this connection.

In our presentation in Sections 3 and 4, we will assume for simiplicity that the finger only goes forward. In Section 5, we will handle the backward direction by using two hands and also show how the hands can be updated during insertions and deletions. In addition, we also analyze how the hands can be used during splits and joins. Finally, we will contrast our design with Kaplan and Tarjan's [13] in Section 6 and then conclude with some remarks to deal with practical issues that may arise when using the hands.

2 Notations and definitions

Lists and elements. In the rest of this paper, all lists are sorted and have unique elements drawn from $(\mathbb{Z}, \leq)$ and the variables a through e will range over $\mathbb{Z}$ without any further quantification. (It would be more general to leave the domain as any total order. For example, some total orders such as $(\mathbb{R}, \leq)$ do not have a natural notion of immediate successor. However, this issue does not come up in this paper.)

Finger destination. To handle the possible case that the search target is not in the list, we define a^+ to be the smallest element in the list that is larger than or equal to a (much like the limit notation). When a is larger than all elements in the list, let a^+ be a sentinel denoted by ∞. We can symmetrically define a^-. With these two definitions, a finger search for a should place f at a^+ if $a^f \leq a$ (forward), or a^- otherwise (backward), where a^f is the element under f when the finger search is started. Note that if a is in the list, then a^+ and a^- are both equal to a and therefore the finger will be placed at a in either case. This allows us to say the finger will be placed on a^+ (or a^-) when we are finger searching for a.

Trees and nodes. In a search tree T_A representing a list A of n elements $a_1, \ldots, a_n$, the node containing a_i will simply be called x_i and the variables w through z will range over nodes. (Notice that a node can contain multiple keys, in which case multiple x_i's can correspond to the same node. However, we will only use the x_i notation when referring to nodes by their ranks.) When referring to a node x_i, we use x_i^{--} and x_i^{++} to denote its *predecessor* x_{i-1} and *successor* x_{i+1} respectively. We denote the *depth* of a node x simply as $depth(x)$, with the depth of the root defined to be 1. The depth of the tree $depth(T_A)$ is the maximum depth among all nodes. We regard nodes without children as leaves.

As stated, our design is based on degree-balanced search trees. All the leaves in such a tree are at the same depth and its balance is maintained by varying the degree of internal nodes between fixed constants. 2-3 trees [2], B-trees [3] and (a, b) trees [12] are all variants of degree-balanced search trees. Red-Black trees [11] can also be viewed as degree-balanced easily via the isomorphism with 2-3-4 trees. We sometimes simplify our presentation by assuming a complete binary search tree (BST), but we also show how to account for this assumption to retain full generality.

A k-ary node x has $(2k-1)$ fields. The keys are sorted elements from A and are denoted as x^j, for $j = 1, 2, \ldots, k-1$ and the children are denoted as $x[j]$, for $j = 1, 2, \ldots, k$. We define the *j-th left child* to be $x[j]$ for $j = 1, 2, \ldots, k-1$ and denote it by $x^j[L]$. The corresponding *j-th right child* is then $x[j+1]$ and denoted by $x^j[R]$. If x is a leaf, then the child pointers are all nil. For

binary nodes, we simply drop the superscript. We say that the finger is *under* a node x when the finger is pointing at a key inside the sub-tree rooted at x.

A node is *overfull* if it has at least k keys. In a degree-balanced search tree, there are no overfull nodes and different nodes can have different arity. During an update, any overfull node will be split into two.

Spines and relatives. We first define spines on binary trees and we give only the version for the right (forward) direction.

The *right spine* of a binary node is defined to be the list of node(s) starting at the node itself, followed by the right spine of its right child, if it exists. The *right-left spine* of a node is the node itself and left spine of its right child. (Our notation stresses the direction taken to traverse the spine and is consistent if we view the right spine as the right-right spine.) Now given any spine of a node, its *atlas* is the second node on the spine (a child) and its *tail* is the last node. Suppose we have three nodes x, y, z in a tree. If x is on the left-right spine of y, then we say y is the *right parent* of x. The *right ancestors* of x will then be y and the right ancestors of y. If y is the right parent of x and the left parent of z with x and z at the same depth, then we say z is the *right peer* of x. In the special case when y is the parent of both, then we say z is the *right brother* of x instead. Figure 1 illustrates some of these concepts on a complete BST. Note that the dashed arrows are only for the purpose of illustration. In particular, our work does *not* make use of such pointers in the nodes. The right-left spine of 8 has also been highlighted.

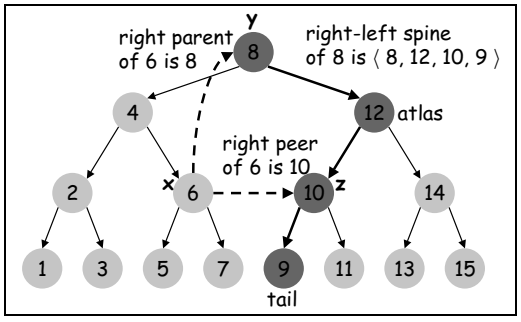


Figure 1: Parent, Peer, and Spine

The definitions for nodes of any higher degree is straightforward using our j-th child quantification. If a k-ary node y is the right parent of x and x is in $y^j[L]$, then we say y^j is the *right parent key* of x.

Deques. We will use doubly-linked queues (*deques*) as a building block of the hand. A deque is made up of *cells* and we denote a deque with k cells by $\langle c_k, \ldots, c_1 \rangle_{\dashv}$, with the back on the right hand side. A deque supports the following operations in $O(1)$ time: MakeDeque, Push, Pop, Inject, Eject, Front, Back, and Prepend. Note that Inject and Eject operate on the back of a deque and a deque can be used as a catenable stack. With an additional pointer to a cell, a deque also supports Split in $O(1)$ time. For more information on deques, refer to Knuth [14].

3 In-order walk

In this section, we motivate and present the design of the hand by developing an in-order walk algorithm with worst-case $O(1)$ work per increment. Our goal is to develop our understanding of the hand through this discussion. To simplify our presentation, we start by working with a complete BST and then generalize to handle all degree-balanced search trees.

3.1 Design

The simpliest in-order walk algorithm is the straightforward recursive solution, which takes amortized $O(1)$ time per increment. To achieve the worst-case $O(1)$ bound, we need to schedule the discovery of nodes that will be processed later in order to avoid traversing a long path during an increment. We refer to this discovery activity as an eager walk. To avoid confusion, in this section

we say that we "visit" a node when it is the actual node being processed by the in-order walk and we "explore" a node when it is being discovered due to the eager walk.

Now, let's look at each increment individually. Suppose we are currently visiting the node x and so the next node to visit is x^{++}. Observe that in a search tree, there are only two possible positions for x^{++} to appear. If x is not a leaf, then x^{++} is the tail y on the right-left spine of x. Otherwise, it is the right parent z of x. (If z does not exist also, then we must be at the rightmost node of the tree. We let the root have an imaginary parent labelled ∞ and end the in-order walk there.) Figure 2 illustrates our situation.

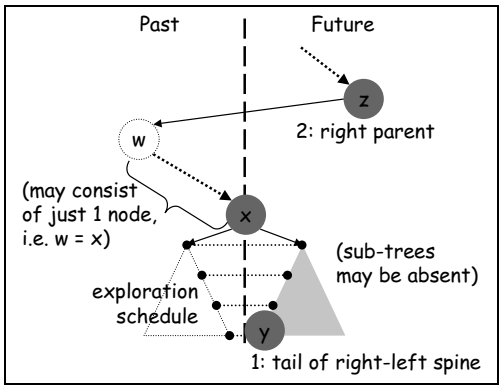


Figure 2: Possible locations of x^{++} and spine exploration schedule

To handle the first case, we must traverse the full right-left spine of x before we visit x. Since we have only a constant amount of time in each increment but the spine can be long, we can only afford to explore a constant number of nodes at a time and perform this multiple times. As we need the spinal nodes in the bottom-to-top order in the in-order walk, we associate a stack with x and we push the right-left spinal nodes of x, beginning with the atlas, onto the stack as we discover them. The scheduling on a degree-balanced search tree is intuitive because all of the leaves are at the same depth and so the left-right spine of x has the same length as the right-left spine. Since all the nodes on the left-right spine must be visited before x, a natural choice is to explore one right-left spinal node when we visit one left-right spinal node. This way, by the time we have visited the tail of the left-right spine, we will have explored the tail of the right-left spine, namely y.

The second case is simpler. To go up the tree, we use a stack to keep track of the ancestors as we descend between visits. But as we show in Figure 2, x can have any number of left ancestors (up to the atlas w). To get to z in constant time, we keep track of only the right ancestors, i.e., we push a node when we descend left and pop it out when we return to it and descend right. Now z will be at the top of the stack when we visit x. (We note that the idea of right parent stack has been used before, e.g., see Brown and Tarjan [6].)

The right parent stack is related to our eager walk as well. Notice that as we approach y in the eager walk, all the nodes we explored are right ancestors of y. Since the right ancestors of x are also right ancestors of y, we are in fact building the upper part of the right parent stack for y. A catenable stack will be perfect for our purpose because when we catenate the right-left spine of x onto its right parent stack, we will immediately have the right parent stack of y. However, we will need INJECT and EJECT in Section 5.4 to handle insertions and deletions. Hence, we will use a deque as a catenable stack.

3.2 The "hand" data structure

The hand is an auxiliary data structure designed to keep track of our progress in the eager walk. For our in-order walk algorithm, it is a deque named Rps. Stored inside the cells of Rps are pointer pairs of the form (node, spine), where node is a pointer to a node in the underlying tree and spine is a (null) pointer that can be used to point to a deque containing similar pointer pairs so that we can prepend the deque pointed by spine onto Rps.

Let the underlying tree be a complete BST T and Rps be a deque consisting of k pointer pairs $\langle (x_k, s_k), \ldots, (x_1, s_1) \rangle_{\dashv}$. Rps must obey these two invariants:

Invariant 3.1 (node) *x_1 is on the right spine of T and $\forall j \in \{2, \ldots, k\} : x_{j-1}$ is the right parent of x_j in T.*

Invariant 3.2 (spine) *$\forall j \in \{1, \ldots, k\} : s_j$ is a deque of* (node, spine) *pairs representing a prefix of x_j's right-left spine, with the atlas stored at the back. The length of s_j is $depth(x_{j+1}) - depth(x_j) - 1$, where $depth(x_{k+1})$ is defined to be $depth(T) + 1$.*

We now relate these two invariants with our design. First of all, the top node x_k in Rps will always correspond to the node x that we are currently visiting. Together with Invariant 3.1, Rps is indeed the right parent stack of x. Now consider the node x_{j-1}. By Invariant 3.1, it is the right parent of x_j. By Invariant 3.2, the length of its associated spine prefix s_{j-1} is $depth(x_j) - depth(x_{j-1}) - 1$. If z is the last node on the prefix, then z is at depth $depth(x_j) - 1$ and therefore $z[L]$ is the right peer of x_j. Since z is stored at the top of s_{j-1}, we will be able to reach the right peer of x_j in $O(1)$ time once we reach the cell containing s_{j-1}. A special case to notice is s_k. By Invariant 3.2 and our definition of $depth(x_{k+1})$, its length is $depth(T) - depth(x_k)$. This is precisely the length of its full right-left spine and this also reflects our design. (In our usage, "prefix" is not necessarily strict and so the full spine is a prefix of itself.)

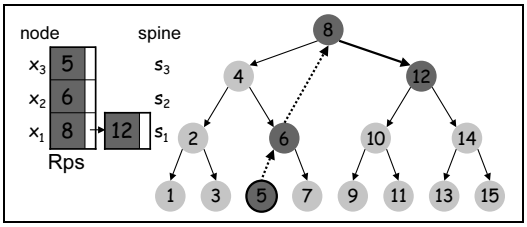


Figure 3: An example hand on 5

The two invariants not only allow us to execute our desired schedule while we are visiting the nodes on the left-right spine of x_{k-1}, but also give us a very strong hint as to why the hand will facilitate finger search. By traversing down Rps, we can reach the right ancestors of the current node *as if* we had right parent pointers. Moreover, the right peer of any node on Rps can be reached with an additional $O(1)$ time, *as if* we had forward side links on each of the right ancestors. The power of these pointers has already been demonstrated by Brown and Tarjan [7] in level-linked 2-3 trees: these pointers are exactly the pointers introduced to facilitate finger searches.

Figure 3 illustrates an example hand at node 5 in a complete BST with 15 nodes. Notice that we have added two dotted arrows pointing upward in the tree to reflect the nature of the right parent stack. As a demonstration of Invariant 3.2, note that the right peers of nodes 5 and 6 are precisely one node away from the end of the spine prefix associated with their right parents.

Using Invariant 3.2, we can immediately bound the size of the hand by the depth of the tree.

Theorem 3.1 (Hand Size) *The hand for a complete BST T has at most $depth(T)$ cells.*

Proof Suppose Rps has k cells. The total number of cells in the hand is $\sum_{j=1}^{k}(1 + |s_j|)$. By Invariant 3.2, this is $k + (depth(x_{k+1}) - depth(x_1) - k)$ which is at most $depth(x_{k+1}) - 1 = depth(T)$. ■

3.3 Algorithm

To start the in-order walk, we first build the hand on the leftmost node of the tree by pushing the left spine of the tree into Rps. We associate an empty deque with each spinal node and use an empty Rps to indicate termination. (The actual algorithm for increment is very succinct, but we have grouped together all the pseudo-codes in this paper into Appendix A. Please refer to the pseudo-code of INCREMENT and EXTENDRIGHTLEFTSPINE.) The correctness of our algorithm follows from the discussion in Section 3.1 and it clearly takes $O(1)$ time per invocation. Note that a hand can be built on any node in $O(\log n)$ time. (Refer to BUILDHAND for how this can be done.) In our case it is built on the leftmost node.

3.4 Extending to k-ary nodes

The in-order walk algorithm above works on a complete BST. When generalizing it to degree-balanced search trees, our j-th child quantification is very handy. We will consider x^j as a binary node, with $x^j[L]$ and $x^j[R]$ as its two children. Suppose we are now visiting the rightmost node of the sub-tree rooted at $x^j[L]$. By a quantified version of Invariant 3.2, at this point we will have all but the tail y of the right-left spine of x^j. The increment to x^j will complete the spine and the increment to y will put us in the *same* situation as if we are visiting the leftmost node of the sub-tree rooted at $x^{j+1}[L]$. The remaining details are straightforward. (See Appendix B for a demonstration.)

Theorem 3.2 (In-order Walk) *In-order walk on a degree-balanced search tree can be performed with worst-case $O(1)$ time per increment, using $O(\log n)$ space and $O(\log n)$ pre-processing (to obtain the initial hand).*

4 Finger search

In this section, we demonstrate how the hand allows us to perform finger searches in a degree-balanced search tree. Again we will simplify our presentation by working with a complete BST and limiting the finger searches to go in the forward direction.

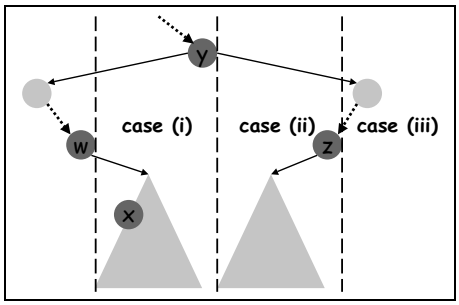


Figure 4: Possible destination of finger

We now consider a finger search for element a with a finger f at node w. Let y be the right parent of w and z be the right peer of w, as shown in Figure 4. Observe that the destination of f can be divided into three rank intervals: (i) $(w, y]$, (ii) $(y, z]$ and (iii) (z, ∞). The first two intervals are characterized by the right sub-tree of w together with y and the left sub-tree of z together with z. We can distinguish among these cases in $O(1)$ time by comparing a with y and z, both readily available from our hand on w.

To handle case (i), we first do an increment as in the in-order walk. This takes $O(1)$ time. Then we traverse the right-left spine of w bottom-up by scanning the Rps towards the back until we hit an element larger than a (or the bottom of Rps). Let x be the node in the *second to last* cell we scanned (or the bottom of Rps). Observe that if a is in the tree, then it must either be in x or its right sub-tree, where we will perform an additional binary tree search. In either case, it is straightforward to restore the two invariants of the hand on our destination. The whole process takes time proportional to the length of x's left spine minus one, which is logarithmic in the size of the left sub-tree skipped by the finger. (We note that the algorithm for this case is precisely the inverted spine technique used in heterogeneous finger search trees by Tarjan and Van Wyk [22].)

Case (ii) can be handled by first popping the Rps twice (removing w and y) and prepending the right-left spine prefix of y onto it (now z is at the top). We then start a binary tree search for a at z while restoring the invariants. The logarithmic time bound follows because the finger skipped the right sub-tree of w.

We handle case (iii) by reducing it to case (i) on a larger scale. We first locate the lowest node x_j on Rps whose key is no larger than our target by successive popping. (Note that as we scan down the Rps, the key gets larger.) Then we shift the hand over to x_j by completing its right-left spine. At this point we re-start the finger search at x_j and we know we will be in case (i). Note that both case (i) and case (ii) are just specializations of case (iii) and we can handle them using this

more general procedure. To analyze the running time, we seperate the rank difference into $\delta(w, x_j)$ and $\delta(x_j, a)$. The time it takes to obtain the hand on x_j is $O(\log \delta(w, x_j))$ because the size of the right sub-tree of x_{j+1} is at most $\delta(w, x_j)$. The subsequent finger search takes time $O(\log \delta(x_j, a))$, as we have already proved when analyzing case (i). The time bound follows from the inequality $\log(c) + \log(d) < 2\log(c + d)$.

We note that our algorithm can be similarly generalized to handle k-ary nodes as we described in Section 3.4 and the time bound remains the same. We also provide a more precise specification of our algorithm in Appendix A in the form of pseudo-code.

Theorem 4.1 (Forward Finger Search) *Using the hand, forward finger searches on a degree-balanced search tree can be performed in worst-case $O(\log d)$ time, where d is the difference in rank between successive search targets.*

5 Extensions

In this section we will outline how to extend the hand to support finger search in both directions and how to update the hand during insertions, deletions, splits and joins.

5.1 Left and right hands

We say that Invariants 3.1 and 3.2 specify the *right hand*. By flipping the left/right directions, we obtain the *left hand* which consists of the left parent stack Lps. For simplicity, we will use "the hands" to denote the left hand and the right hand collectively.

Consider the hands on a node x. By definition, each of the ancestors of x will appear on either Lps or Rps. In particular, the root node will be at the bottom of one of them. We now extend the stack cells to contain a triple (node, spine, cross), where cross is a pointer to another cell. Let Lps be $\langle (x_{lk}, s_{lk}, c_{lk}), \ldots, (x_{l1}, s_{l1}, c_{l1})\rangle_{\dashv}$ and Rps be $\langle (x_{rk}, s_{rk}, c_{rk}), \ldots, (x_{r1}, s_{r1}, c_{r1})\rangle_{\dashv}$. Note that in general $lk \neq rk$ but $x = x_{lk} = x_{rk}$. We require the hands to satisfy one additional invariant:

Invariant 5.1 (cross) *Starting at the cell containing the root, the path specified by chasing the* cross *pointers is the path from the root to x, with the encoding that if c_{lk} or c_{rk} is* nil*, then it points to the cell directly above the current cell. If x is a left child, then the path ends on* Lps*. Otherwise, it ends on* Rps*.*

5.2 Decrement

Instead of showing how to perform decrement, we will describe how to update the left hand in an increment. Decrement will follow by symmetry. This also serves as a demonstration of Invariant 5.1 and the cross pointers. As an aid to our description below, Figure 5 shows the hands on nodes 2 to 5 in a complete BST with 15 nodes, which was shown in Figure 3.

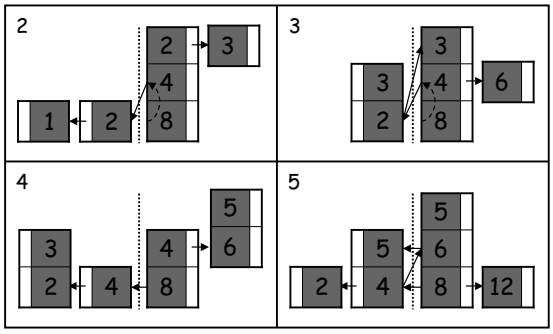


Figure 5: Example hands on 2 to 5, shown with cross pointers. (Dashed pointers are implicit.)

Before we pop the Rps, we check to see if the c_{lp} points to the top cell of Rps. If so, we set it to nil. (See $3 \rightarrow 4$.) Then we pop the Rps and extend the right-left spine of x_{rp} as usual. Let (x_{lj}, s_{lj}, c_{lj}) be the cell $cell_j$ pointed to by c_{rp}. (We can verify that this cell always exists.) If s_{lj} is non-empty, then we pop off its top cell to shorten the spine prefix by

one node and set s_{new} be nil. (See $2 \to 3$ and $4 \to 5$.) Otherwise, we set c_{rp} to nil and split Lpr at $cell_j$ to obtain $\langle (x_{lk}, s_{lk}, c_{lk}), \ldots, (x_{lj}, s_{lj}, c_{lj}) \rangle_{\dashv}$ as s_{new}. (See $3 \to 4$.) We preprend s_{curr} to Rps as usual. Finally we push $(x_{new}, s_{new}, \mathsf{nil})$ onto Lps, where x_{new} is the top node in Rps. (We can verify that s_{new} is the correct left-right spine prefix of x_{new}.)

While the above procedure may seem complicated, it can be derived from the maintainence required by the three invariants. We also note that the increment algorithm still takes $O(1)$ time. Since we showed the left hand can also be maintained in worst-case $O(1)$ time during an increment, by symmetry, the following theorem holds.

Theorem 5.1 (Backward In-order Walk) *An in-order walk in the backward direction takes worst-case $O(1)$ time per decrement.*

5.3 Backward finger search

The description in Section 4 can easily be adapted to update the left hand in a backward finger search. Here we show how to preserve Invariants 3.1 and 3.2 for the right hand as well. The maintenance of Invariant 5.1 is straightforward.

Recall that our finger search algorithm will first locate the left parent x containing the smallest key that is no smaller than the target. Let the last cell we popped from Lps be (z, s_z, c_z). We will pop the Rps and clean up the associated spine prefixes until the cell pointed to by c_z is removed. Note that we have enough time to do this because we have skipped the left sub-tree of z.

At this point, the top cell in Lps will be (x, s_x, c_x). We split Rps at c_x, push a new cell containing x into Rps and then associate the upper deque from the split to this cell as its right-left spine prefix. Finally, we extend the prefix to contain z unless the finger search initially started at z. This step only takes $O(1)$ time.

Then our algorithm will complete the left-right spine of x to obtain the hands on it. We update the right hand by completing its right-left spine prefix on Rps. Since the prefix already reaches z, the time it actually takes to complete the spine is logarithmic in the size of the left sub-tree of z, which we skipped.

If the target is not x, then it is in the left sub-tree. Our algorithm will preform a decrement and then start searching for the target by scanning the left-right spine of x upward until we hit the smallest key that is no smaller than the target. Every time we go up a node, we also update the right hand by shortening the right-left spine prefix of x in Rps. Finally, our algorithm will finish with a descent while restoring the invariants. The updates to the right hand in this part are straightforward and take the same amount of time as updating the left hand.

Theorem 5.2 (Finger Search) *The hands can be maintained in any sequence of finger searches in $O(\log d)$ time per search, where d is the difference in rank between successive search targets.*

5.4 Insertions and Deletions

In a search structure that supports finger search, insertions and deletions are typically implemented by first placing the finger at the target element followed by the actual update. Huddleston and Mehlhorn [12] have shown that in a sequence of updates, the amount of structural changes in an (a, b) tree is exponentially decreasing with the height of the propagation from the leaves and that each update takes amortized $O(1)$ time, both assuming an initially empty tree and discounting the time spent to shift the finger.

In this section, we will show that the hands can be updated to reflect each structural change in worst-case $O(1)$ time. Therefore, *any* result on the distribution of structural changes can be carried over to the hands directly. In particular, both of the above results by Huddleston and Mehlhorn will continue to hold even when we have to maintain the hands.

In the following discussion, we assumed familiarity with the insertion and deletion algorithms for degree-balanced search trees. (See Huddleston and Mehlhorn [12] for more information.) Let the target element of the update be t. We will consider the hands for k-ary nodes. To simplify our presentation, we will only update the right hand w.r.t. the k-ary adaption of Invariants 3.1 and 3.2. The left hand can be updated by symmetry and it is also easy to maintain Invariant 5.1 throughout. We adopt the convention that the hands will be placed on t after its insertion, or t^{++} for deletion.

We will start with an observation. In a degree-balanced search tree, the structural change due to insertions and deletions propagates up from a leaf to the root. All the nodes involved must be in either Lps or Rps. Let Rps be $\langle (x_k, s_k), \ldots, (x_1, s_1) \rangle_{\dashv}$. We will update the hands by considering one depth at a time in a bottom-up fashion, provided that the hands are placed on a leaf first.

There are three kinds of possible structural changes at a depth: node fusion, children sharing and node split. (The last one is not to be confused with the splitting of a tree.) We will first analyze them and then return to insertions and deletions.

Fusion. Consider a node x, with the finger under it. Suppose x has a right brother y that will be fused into x and p is the right parent with key c. Note that c is the key being demoted. Let z be the right parent of p, if it exists. If c is p^k, then first extend the spine prefix of z and remove the cell of p from Rps. No further change is needed if x is in Rps. Otherwise, create a cell in Rps above that of p (or z if p is not in Rps anymore) and let it contain x with key c. Also eject the bottom cell from the spine prefix of p and re-associate the result with x instead.

Now suppose x has a left brother w and x is being fused into w. No change is needed if x is not in Rps. Otherwise, update the cell of x to contain w instead and adjust the offset in the cell accordingly.

Sharing. Consider a node x, with the finger under it. Suppose x is sharing from its right brother y and p is the right parent with key c. We only need to update Rps if x is not originally in it. First create a new cell above that of p and let it contain x with key c. Then shorten the spine prefix of p by ejecting the bottom cell and re-associate the result with x instead.

Now suppose x has a left brother w where x is sharing from. There are four possible cases depending on whether x is in Rps and similarly for p. In each of these cases, the structure of Rps does not change except that the offset in the cell of x needs to be updated to reflect the new key(s) in x.

Split. Consider a overfull node x, with the finger under it and c as its median key. Suppose after c is promoted to the parent p, a new right brother y of x is created. Let the finger be under $x[j]$ and z be the right parent of p, if it exists. We break down the analysis into three cases. In the first two, if p is the new root, then inject an empty cell at the bottom of Rps and let it contain p.

Suppose x^j is smaller than c. If p is on Rps, then no change is needed. Otherwise, shorten the spine prefix of z, create a new cell under that of x and let it contain p with key c.

Suppose x^j is c and let d be x^{j+1}. If p is not on Rps, then shorten the spine prefix of z, create a new cell under that of x and let it contain p with key c. Now remove the cell of x from Rps and create a new cell containing y with key d. Finally, inject the new cell at the bottom of the spine prefix of x and re-associate the result with p.

Suppose x^j is larger than c. If p is on Rps, then increment the offset in its cell. Also, if x is on Rps, then update its cell to contain y instead and adjust the offset accordingly.

Insertion. As we assumed the list maintains unique elements, t must be absent and the hands are on either t^- or t^+. Observe that at least one of t^- and t^+ is in a leaf. Here we assume t^+ is in the leaf x with the hands placed on it. If t^+ is an internal node instead, then perform a decrement

to obtain the hands on t^- and the rest is the same.

To begin the insertion, first increment the offset of the top cell of Rps. Notice that Rps is a valid hand on t, but x may have too many keys and a split or sharing will be needed. After we have handled x, its parent p may have one more key and another split or sharing may occur at its depth. We will repeat until the propagation stops. It should be clear that at each depth involved in the propagation, we spent only $O(1)$ time.

Deletion. Here we assume we have the hands on the leaf x containing t. Observe that if t is not in a leaf, then t^{++} is. By Invariant 3.2, t^{++} will be at the top of the spine stack associated with x. We replace t with t^{++} in x, perform an increment to obtain the hands on the tail x', which contains the original t^{++}. Now we will consider deleting t^{++} from x' instead. A further decrement will put the hands back on t^{++} in $O(1)$ time.

To begin the deletion, consider the leaf x. If t is x^k, then first extend the spine prefix of the right parent of x and remove the cell of x from Rps. If t is not x^k, then update the top cell of Rps to contain t^{++} instead of t. In both cases, notice that Rps is still a valid right hand on t (it is on t^+ now), but x may have too few keys and a fusion or sharing will be need. After we have handled x, its parent p may have one less key and another fusion or sharing may occur at its depth. We will repeat until the propagation stops. At the end, if the root is empty and it is on the Rps, then we can simply eject the bottom cell. It should be clear that at each depth involved in the propagation, we spent only $O(1)$ time.

Theorem 5.3 (Insertion and Deletion) *The hands can be updated synchronously during an insertion or a deletion in time proportional to the total number of structural changes in the tree.*

5.5 Splits and Joins

Again we assume familiarity with the split and join algorithms on degree-balanced search trees. (See Cormen et al.[9, p. 278, p. 399] for their coverage on Red-Black Trees and 2-3-4 Trees.) Our focus will be on analyzing the maintainence of the hands during these operations.

Join. Consider joining two trees T_1 and T_2 with b as the splitting key where $a < b < c$ for any $a \in T_1$ and $c \in T_2$. Let h_1 and h_2 be the heights of T_1 and T_2 respectively and by symmetry assume $h_1 \leq h_2$. The join operation should produce a new tree $T = T_1 \cup \{b\} \cup T_2$ in $O(h_2 - h_1)$ time, assuming the two trees maintain their own heights. For the purpose of our analysis, we will assume that we have our hands holding any key in T_2.

The first step in a typical join algorithm is to identify the node z on the left spine of T_2 such that the sub-tree rooted at z has the same height as T_1. When the algorithm is scanning for z down from the root, we can easily locate the cells in the hands that correspond to that height. (There can be one cell at this depth in both Lps or Rps and so there are exactly either one or two.) Notice that the cells located may not correspond to z, but they are the starting points for us to update the hands one depth at a time.

Once z has been identified on the spine, the root node of T_1 will be fused into z, with b as the key in-between. z may now get overfull and a sequence of node splits and children sharings may follow. The key observation is that this sequence of structural changes all occur on the left spine of T_2, starting at the depth of z. The situaton is very similar to an insertion, except the structural change propagates from z instead of from a leaf. Therefore, the hands can be updated in a way similar to during insertions and stop as soon as the propagation stops. The actual details are straightforward.

Theorem 5.4 (Join) *The hands in the larger tree can be updated synchronously during a join in time proportional to the total number of structural changes in that tree.*

However, we note that if there is a pair of hands on the smaller tree (T_1 in our case), then discarding the hands alone will take $O(h_1)$ time. Fortunately, there is no pointers from the nodes to the hands. Therefore, we may choose to discard the hands at a later time.

Split. When we split a tree T of height h using a key b, we obtain two trees T_1 and T_2 such that $a < b < c$ for any $a \in T_1$ and $c \in T_2$ in $O(\log n)$ time. In this analysis, we assume we have already placed the hands on b (see below if we do not have our hands on b yet) and by symmetry we assume b is in the left sub-tree of root. At the end of the split, we should have two pairs of hands, one at b^{--} and the other at b^{++}.

A typical split algorithm will decompose T into four groups: the left ancestors of b, their left sub-trees and symmetrically for the right ancestors. By symmetry, we will consider only the right ancestors and their right sub-trees. Let Rps be $\langle (x_{rk}, s_{rk}), \ldots, (x_{r1}, s_{r1}) \rangle_{\dashv}$, i.e., x_{rk} contains b and x_{r1} is the root, and let T_{ri} be the right sub-tree of x_{ri}. To obtain T_2, split will be joining T_{ri} to the left of $T_{r(i-1)}$ using $x_{r(i-1)}$ as the splitting key for $i = k$ down to 2 (or from $k-1$ down to 2 if b is a leaf). We now show that the hands actually facilitate these joins in an intuitive way.

Consider joining T_{ri} with $T_{r(i-1)}$. Observe that, using the notation in our analysis of join, we don't have to scan for z any more. By Invariant 3.2, z will be the left-left grandchild of the node contained in the top cell of $s_{r(i-1)}$. The key $x_{r(i-1)}$ is also readily available on the Rps. Therefore, we can immediately start the join at z. Furthermore, notice that s_{ri} actually contains a fragment of the left spine of T_2. Therefore, the assembling s_{ri}'s into the Rps for T_2 is can be done straightforwardly. The Lps for T_2 will be a single cell containing b^{++}. The hands for T_1 can also be obtained symmetrically.

Theorem 5.5 (Split) *When splitting a tree T into T_1 and T_2 using the current key b held in the hands, the two new hands at b^{--} and b^{++} can be generated in time proportional to the total amount of work done by the joins to assemble the right spine of T_1 and the left spine of T_2.*

The above analysis assumes that we have already placed the hands on the splitting key b. Notice that the total time of the split is proportional to the depth of b in T. Therefore, if b is deep and the hands are not positioned close-by, then it will be simpler to discard the hands and split T as usual. Rebuilding the two hands can then be done in logarithmic time.

6 Discussion

Hands and inverted spines. In this paper, we have demonstrated that our view of finger search as a property, rather than an operation, allows us a much bigger design space. In fact, there are other previous works that do not use an element pointer. A recent exception to this is the purely-functional catenable sorted list designed by Kaplan and Tarjan [13] in 1996. Instead of an element pointer, their structure allows splitting the list at the d-th position in worst-case $O(\log d)$ time and catenating in time doubly logarithmic in the size of the smaller list. Finger searches can thus be realized by splitting and catenating between two instances of their structure, with the finger pointing at the element at the break.

Although it was not mentioned explicitly, the modified 2-3 finger search tree representation in their paper actually uses only $O(\log n)$ extra storage for an n-element list. The key to their design is to carefully relax the degree constraint on the spines to allow a suitable storage redundancy, which can in turn be used to absorb the propagation of structural changes due to splits and catenations. We can view their design as an improvement upon the heterogeneous finger search trees [22] in which splits and joins have an amortized time bound. (See Booth [4, Ch. 2], Mehlhorn [17] and Kosaraju [16] for the analysis.) As we have pointed out in Section 4, the power of the hands also comes from the inverted spine technique used in the heterogeneous finger search trees. However,

instead of relaxing the degree constraint on the spines, we showed that it is possible to avoid the splits and joins if we view the "inverted spine" by the way of the hands. This connection should be relatively straightforward if we review our discussion of split in Section 5.5.

Remarks on practice. We would like to conclude with several remarks on issues that may arise when the hands are used in practice. First, notice that the simultaneous maintainance of the forward and backward hands is often not necessary. In applications like merging, only the forward hand is needed and therefore we do not need to maintain Lps and the cross points on Rps. Furthermore, even when an application requires the ability to perform finger searches in both directions, we note that a new hand can be built on any node in only logarithmic time. Hence if the number of direction change is small and we must maintain exactly one hand, it may be more desirable to simply rebuild the hand at each direction change. Finally, we have included a brief discussion in Appendix C on how the hands can be used to facilitate pre-fectching in databases. While our in-order walk result is only a theoretical improvement, it will be interesting to see if the space savings in the tree also translates to a cache performance gain that outweights the overhead to maintain the hands.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics*, 3:1259–1263, 1962. 1

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974. 1, 2

[3] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. 1, 2

[4] H. D. Booth. *Some Fast Algorithms on Graphs and Trees*. PhD thesis, Princeton University, 1990. 6

[5] G. S. Brodal. Finger search trees with constant insertion time. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 540–549, 1998. 1

[6] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979. 1, 3.1

[7] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9(3):594–614, 1980. 1, 1, 1, 3.2

[8] R. Cole. On the dynamic finger conjecture for splay trees part II: The proof. Technical Report TR1995-701, Courant Institute, New York University, 1995. 1

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1989. 5.5

[10] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977. 1

[11] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978. 2

[12] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. 1, 1, 2, 5.4

[13] H. Kaplan and R. E. Tarjan. Purely functional representations of catenable sorted lists. In *Proc. 28th Annual ACM Symposium on the Theory of Computing*, pages 202–211, 1996. 1, 1, 6

[14] D. E. Knuth. *The Art of Computer Programming, Volume 1*. Prentice Hall PTR, 3 edition, 1997. 2

[15] S. R. Kosaraju. Localized search in sorted lists. In *Proc. 13th Annual ACM Symposium on Theory of Computing*, pages 62–69, 1981. 1

[16] S. R. Kosaraju. An optimal RAM implementation of catenable min doubled-ended queues. In *Proc. 5th Annual ACM Symposium on Discrete Algorithms*, pages 195–203, 1994. 6

[17] K. Mehlhorn. *Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching*, pages 214–216. Springer-Verlag, 1984. 6

[18] M. H. Overmars. *The Design of Dynamic Data Structures*, pages 34–35. Number 156 in LNCS. Springer-Verlag, 1983. 1

[19] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland, College Park, 1990. 1

[20] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996. 1

[21] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985. 1

[22] R. E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal of Computing*, 17(1):143–178, 1988. 1, 1, 4, 6

[23] A. K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67:173–194, 1985. 1

A Finger search pseudo-code

We provide the pseudo-code of the forward finger search algorithm on a complete BST we presented in Sections 3.3 and 4 for reference.

```
ExtendRightLeftSpine(x, s)
1  if |s| = 0   (* atlas vs. the rest *)
2     then y ← x.right
3     else  y ← s.Front().node.left
4  if y ≠ nil
5     then s.Push((y, MakeDeque()))
```

```
CompleteRightLeftSpine(x, s)
1  if |s| = 0   (* atlas vs. the rest *)
2     then y ← x.right
3     else  y ← s.Front().node.left
4  while y ≠ nil
5       do s.Push((y, MakeDeque()))
6          y ← y.left
```

```
Increment()
1  (x_curr, s_curr) ← Rps.Pop()
2  if |Rps| > 0
3     then (x_rp, s_rp) ← Rps.Front()
4          ExtendRightLeftSpine(x_rp, s_rp)
5  Rps.Prepend(s_curr)
```

```
RootedSearch(b)
1  (x_curr, s_curr) ← Rps.Pop()
2  if |Rps| > 0
3     then (x_rp, s_rp) ← Rps.Front()
4     else  (x_rp, s_rp) ← (x_curr, MakeDeque())
```

```
 5  while x_curr ≠ nil
 6     do (* restore invariants while descending *)
 7        if b ≤ x_curr.Key
 8           then (x_rp, s_rp) ← (x_curr, MakeDeque())
 9                Rps.Push((x_rp, s_rp))
10                x_curr ← x_curr.left
11           else ExtendRightLeftSpine(x_rp, s_rp)
12                x_curr ← x_curr.right
```

```
ForwardSubTreeSearch(b)
1  (x_curr, s_curr) ← Rps.Pop()
2  (* ascend along the inverted spine *)
3  while |Rps| > 0 ∧ Rps.Front().node.key ≤ b
4     do (x_curr, s_curr) ← Rps.Pop()
5  Rps.Push((x_curr, s_curr))
6  (* descend as in a binary tree search *)
7  RootedSearch(b)
```

```
BuildHand(T, b)
1  Rps ← MakeDeque()
2  Rps.Push((T.root, MakeDeque()))
3  RootedSearch(b)
```

```
ObtainInitFinger(T)
1  BuildHand(T, −∞)
```

```
ForwardFingerSearch(b)
 1  (* assumes hand is not at ∞ and x_curr.key < b *)
 2  x_curr ← Rps.Front().node
 3  if |Rps| ≥ 2
 4     then (x_rp, s_rp) ← Rps.Front().next   (* 2^nd cell *)
 5  if b ≤ x_rp.key   (* case (i) *)
 6     then Increment()
 7          ForwardSubTreeSearch(b)
 8          return
 9  (x_rp, s_rp) ← Rps.Pop()   (* case (ii) and case (iii) *)
10  while |Rps| > 0 ∧ Rps.Front().node.key ≤ b
11     do (x_rp, s_rp) ← Rps.Pop()
12  Rps.Push((x_rp, s_rp))
13  CompleteRightLeftSpine(x_rp, s_rp)
14  Increment()
15  ForwardSubTreeSearch(b)
```

B Handling k-ary nodes

We only require a slight adjustment to the cells when we extend the hands to handle k-ary nodes. In particular, instead of storing a pointer to a k-ary node x, we now also store the offset, which indicates the sub-tree that contains the finger. For example, if the finger is under $x[j]$, then x will

appear on the Rps as (x, j) instead of just x. This is to reflect the fact that x^j is the right parent key. For concision, we will simply say x^j in our discussion and a cell will be written as (x^j, s). Here we present the increment algorithm that has been adapted to handle k-ary nodes as an example of how we can adapt our algorithms.

```
INCREMENT()
1  (x^j_curr, s_curr) ← Rps.POP()
2  if j < k − 1
3     then Rps.PUSH(x^{j+1}_curr, nil)
4     else if |Rps| > 0
5             then (x^j_rp, s_rp) ← Rps.FRONT()
6                  EXTENDRIGHTLEFTSPINE(x^j_rp, s_rp)
7  Rps.PREPEND(s_curr)
```

C In-order walk and databases

Typically we will not index every column in a database table. When a query involves columns that are not indexed, we may need to scan the whole column. This corresponds to an in-order walk in the B-tree that contains the actual table. The in-order walk algorithm in Section 3 is already an improvement over the straightforward recursive solution asymptotically, since its $O(1)$ time bound is worst-case instead of amortized. However, it is not clear that in practice our algorithm will be faster if we just use the simple implementation outlined in our pseudo-code.

We do want to briefly describe an observation about the hands and the idea of pre-fetching. In particular, our in-order walk algorithm suggests a pre-fetching schedule that seems implementatable. We can lock the upper portion of the hands and all the nodes referenced by those cells in the data cache (and as the height of the hand shrinks, we will also use pre-fetching to bring the lower cells and their associated spine lists into the cache). This guarantees that accessing any cells in that portion as well as the nodes referenced will not generate a cache miss. Also, we can use pre-fetching in EXTENDRIGHTLEFTSPINE since the right-left spine s_{rp} will not be needed immediately. Further, it is possible to do the eager walk "over-eagerly" by extending Invariant 3.2 to mandate the full right-left spines to be stored for the top few cells. Finally, we can associate second level spines to the cells in the spine lists. This corresponds to a very aggressive pre-fetching schedule that guarantees all the nodes required in the near future are pre-fetched by the hands.

We note that, however, in practice more sophisticated trees are used in databases. For example, B*-trees is designed with fast scanning in mind. Since these trees use more space than a B-tree, their cache performance may be worse than a simple B-tree with the hands. It will be interesting to implement and benchmark the performance of using the hands and see what our theoretical result would translate to in the real world.

Bibliography

[Aca05] Umut A. Acar. "Self-Adjusting Computation". CMU-CS-05-129. PhD thesis. Carnegie Mellon University, 2005. See p. 88.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974. ISBN: 0-201-00029-6. See pp. 16, 59, 93.

[AHU82] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1982. ISBN: 0-201-00023-7. See p. 16.

[And93] Arne Andersson. "Balanced Search Trees Made Simple". In: *Proceedings of the 3rd Workshop on Algorithms and Data Structures*. 1993, pp. 60–71. DOI: 10.1007/3-540-57155-8_236. See p. 17.

[And96] Arne Andersson. "Faster Deterministic Sorting and Searching in Linear Space". In: *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*. 1996, pp. 135–141. DOI: 10.1109/SFCS.1996.548472. Cited as [AT07].

[AS89] Cecilia R. Aragon and Raimund Seidel. "Randomized Search Trees". In: *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*. 1989, pp. 540–545. DOI: 10.1109/SFCS.1989.63531. Later [SA96]. See p. 62.

[AT00] Arne Andersson and Mikkel Thorup. "Tight(er) Worst-Case Bounds on Dynamic Searching and Priority Queues". In: *Proceedings of the 32nd ACM Symposium on Theory of Computing*. 2000, pp. 335–342. DOI: 10.1145/335305.335344. Cited as [AT07].

[AT01] Arne Andersson and Mikkel Thorup. "Dynamic String Searching". In: *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*. 2001, pp. 307–308. Cited as [AT07].

[AT07] Arne Andersson and Mikkel Thorup. "Dynamic Ordered Sets with Exponential Search Trees". In: *Journal of the ACM* 54.3 (2007). 40 pages, Article 13. DOI: 10.1145/1236457.1236460. From [And96; AT00; AT01]. See pp. 3, 63.

[Aur87] Franz Aurenhammer. "Jordan Sorting via Convex Hulls of Certain Non-Simple Polygons". In: *Proceedings of the 3rd ACM Symposium on Computational Geometry*. 1987, pp. 21–29. DOI: 10.1145/41958.41961. See p. 41.

[AVL62] G. M. Adel'son-Vel'skii and E. M. Landis. "An Algorithm for the Organization of Information". In: *Soviet Mathematics Doklady* 3 (1962), pp. 1259–1263. See pp. 15, 28, 60, 87, 92.

[BA95] Amir M. Ben-Amram. "What is a "Pointer Machine"?" In: *ACM SIGACT News* 26.2 (1995), pp. 88–95. DOI: 10.1145/202840.202846. See p. 63.

[Bay71] Rudolf Bayer. "Binary B-Trees for Virtual Memory". In: *ACM Workshop on Data Description, Access and Control*. 1971, pp. 219–235. See p. 17.

[Bay72] Rudolf Bayer. "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms". In: *Acta Informatica* 1 (1972), pp. 290–306. DOI: 10.1007/BF00289509. See p. 17.

[BBG02] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. "Biased Skip Lists". In: *Proceedings of the 13th International Symposium on Algorithms and Computation*. 2002, pp. 1–13. DOI: 10.1007/3-540-36136-7_1. Cited as [BBG05].

[BBG05] Amitabha Bagchi, Adam L. Buchsbaum, and Michael T. Goodrich. "Biased Skip Lists". In: *Algorithmica* 42.1 (2005), pp. 31–48. DOI: 10.1007/s00453-004-1138-6. From [BBG02]. See pp. 62, 63.

[BCDI07] Mihai Badoiu, Richard Cole, Erik D. Demaine, and John Iacono. "A Unified Access Bound on Comparison-Based Dynamic Dictionaries". In: *Theoretical Computer Science* 382.2 (2007), pp. 86–96. DOI: 10.1016/j.tcs.2007.03.002. From [BD04]. See pp. 31, 64.

[BD04] Mihai Badoiu and Erik D. Demaine. "A Simplified, Dynamic Unified Structure". In: *Proceedings of the 4th Latin American Theoretical Informatics Symposium*. 2004, pp. 466–473. Cited as [BCDI07].

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. "Cache-Oblivious B-Trees". In: *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*. 2000, pp. 399–409. DOI: 10.1109/SFCS.2000.892128. Cited as [BDFC05].

[BDFC05] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. "Cache-Oblivious B-Trees". In: *SIAM Journal on Computing* 35.2 (2005), pp. 341–358. DOI: 10.1137/S0097539701389956. From [BDFC00]. See p. 17.

[BF02] Paul Beame and Faith Ellen Fich. "Optimal Bounds for the Predecessor Problem and Related Problems". In: *Journal of Computer and System Sciences* 65.1 (2002), pp. 38–72. DOI: 10.1006/jcss.2002.1822. From [BF99]. See p. 63.

[BF99] Paul Beame and Faith Ellen Fich. "Optimal Bounds for the Predecessor Problem". In: *Proceedings of the 31st ACM Symposium on Theory of Computing*. 1999, pp. 295–304. DOI: 10.1145/301250.301323. Cited as [BF02].

[BLMTT02] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios K. Tsakalidis, and Kostas Tsichlas. "Optimal Finger Search Trees in the Pointer Machine". In: *Proceedings of the 34th ACM Symposium on Theory of Computing*. 2002, pp. 583–591. DOI: 10.1145/509907.509991. Cited as [BLMTT03].

[BLMTT03] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios K. Tsakalidis, and Kostas Tsichlas. "Optimal Finger Search Trees in the Pointer Machine". In: *Journal of Computer and System Sciences* 67.2 (2003), pp. 381–418. DOI: 10.1016/S0022-0000(03)00013-8. From [BLMTT02]. See pp. 27, 61, 63.

[BM72] Rudolf Bayer and Edward M. McCreight. "Organization and Maintenance of Large Ordered Indexes". In: *Acta Informatica* 1.3 (1972), pp. 173–189. DOI: 10.1007/BF00288683. See pp. 16, 17.

[BMW02] Guy E. Blelloch, Bruce M. Maggs, and Shan Leung Maverick Woo. *Space-Efficient Finger Search on Degree-Balanced Search Trees*. Tech. rep. CMU-CS-02-184. Carnegie Mellon University, 2002. See pp. 95, 153.

[BMW03] Guy E. Blelloch, Bruce M. Maggs, and Shan Leung Maverick Woo. "Space-Efficient Finger Search on Degree-Balanced Search Trees". In: *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*. 2003, pp. 374–383. See pp. 91, 95, 97, 153.

[Boo90] Heather Donnell Booth. "Some Fast Algorithms on Graphs and Trees". PhD thesis. Princeton University, 1990. See pp. 34, 37, 93.

[Bro04] Gerth Stølting Brodal. "Finger Search Trees". In: *Handbook of Data Structures and Applications*. Ed. by Dinesh P. Mehta and Sartaj Sahni. Chapman & Hall/CRC, 2004. Chap. 11. ISBN: 1584884355. See p. 62.

[Bro96] Gerth Stølting Brodal. "Worst-Case Efficient Priority Queues". In: *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*. 1996, pp. 52–58. See p. 58.

[Bro98] Gerth Stølting Brodal. "Finger Search Trees with Constant Insertion Time". In: *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*. 1998, pp. 540–549. See pp. 3, 61.

[BT78] Mark R. Brown and Robert E. Tarjan. "A Representation for Linear Lists with Movable Fingers". In: *Proceedings of the 10th ACM Symposium on Theory of Computing*. 1978, pp. 19–29. DOI: 10.1145/800133.804328. Later [BT80]. See p. 59.

[BT79] Mark R. Brown and Robert E. Tarjan. "A Fast Merging Algorithm". In: *Journal of the ACM* 26.2 (1979), pp. 211–226. DOI: 10.1145/322123.322127. See pp. 28, 92–94.

[BT80] Mark R. Brown and Robert E. Tarjan. "Design and Analysis of a Data Structure for Representing Sorted Lists". In: *SIAM Journal on Computing* 9.3 (1980), pp. 594–614. DOI: 10.1137/0209045. From [BT78]. See pp. 23, 28, 29, 59, 63, 93, 106.

[BY76] Jon Louis Bentley and Andrew Chi-Chih Yao. "An Almost Optimal Algorithm for Unbounded Searching". In: *Information Processing Letters* 5.3 (1976), pp. 82–87. DOI: 10.1016/0020-0190(76)90071-5. See pp. 23, 24.

[CH93] Ranjan Chaudhuri and Hartmut Höft. "Splaying a Search Tree in Preorder Takes Linear Time". In: *ACM SIGACT News* 24.2 (1993), pp. 88–93. DOI: 10.1145/156063.156067. See p. 54.

[CK77] Michael J. Clancy and Donald E. Knuth. *A Programming and Problem-Solving Seminar*. Tech. rep. CS-TR-77-606. Stanford University, 1977. See p. 58.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2nd ed. MIT Press, 2001. ISBN: 0-07-013151-1. See pp. 6, 7, 10, 15, 34, 66, 108.

[CMSS00] Richard Cole, Bud Mishra, Jeanette P. Schmidt, and Alan Siegel. "On the Dynamic Finger Conjecture for Splay Trees, Part I: Splay Sorting log n-Block Sequences". In: *SIAM Journal on Computing* 30.1 (2000), pp. 1–43. DOI: 10.1137/S0097539797326988. Cited as [Col90; CMSS95].

[CMSS95] Richard Cole, Bud Mishra, Jeanette P. Schmidt, and Alan Siegel. *On the Dynamic Finger Conjecture for Splay Trees Part I: Splay Sorting log n-Block Sequences*. Tech. rep. TR1995-700. Courant Institute, New York University, 1995. Cited as [CMSS00].

[Col00] Richard Cole. "On the Dynamic Finger Conjecture for Splay Trees, Part II: The Proof". In: *SIAM Journal on Computing* 30.1 (2000), pp. 44–85. DOI: 10.1137/S009753979732699X. From [Col90; Col95]. See pp. ii, 1, 19, 21, 28, 54, 61, 106.

[Col90] Richard Cole. "On the Dynamic Finger Conjecture for Splay Trees (Extended Abstract)". In: *Proceedings of the 22nd ACM Symposium on Theory of Computing*. 1990, pp. 8–17. DOI: 10.1145/100216.100218. Later [CMSS00; Col00]. See p. 138.

[Col95] Richard Cole. *On the Dynamic Finger Conjecture for Splay Trees Part II: The Proof*. Tech. rep. TR1995-701. Courant Institute, New York University, 1995. Cited as [Col00].

[Com79] Douglas Comer. "The Ubiquitous B-Tree". In: *ACM Computing Surveys* 11.2 (1979), pp. 121–137. DOI: 10.1145/356770.356776. See pp. 16, 17.

[Cra72] Clark A. Crane. "Linear Lists and Priority Queues as Balanced Binary Trees". PhD thesis. Stanford University, 1972. See p. 15.

[CW82] Karel Culik II and Derick Wood. "A Note on Some Tree Similarity Measures". In: *Information Processing Letters* 15.1 (1982), pp. 39–42. DOI: 10.1016/0020-0190(82)90083-7. See p. 43.

[Den05] Peter J. Denning. "The Locality Principle". In: *Communications of the ACM* 48.7 (2005), pp. 19–24. DOI: 10.1145/1070838.1070856. See p. 33.

[Der08] Jonathan C. Derryberry. "Adaptive Search Data Structures". Thesis Proposal, Carnegie Mellon University. 2008. See p. 64.

[DHIKP09] Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Pătraşcu. "The Geometry of Binary Search Trees". In: *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms*. 2009, pp. 496–505. See pp. 38, 44, 46, 55, 138, 143.

[DHIP04] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. "Dynamic Optimality—Almost". In: *Proceedings of the 45th IEEE Symposium on Foundations of Computer Science*. 2004, pp. 484–490. DOI: 10.1109/FOCS.2004.23. Later [DHIP07]. See pp. 42, 44–46, 48, 54, 56, 135.

[DHIP07] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. "Dynamic Optimality—Almost". In: *SIAM Journal on Computing* 37.1 (2007), pp. 240–251. DOI: 10.1137/S0097539705447347. From [DHIP04]. See pp. 54, 56, 107, 135, 138, 152.

[DN04] Luc Devroye and Ralph Neininger. "Distances and Finger Search in Random Binary Search Trees". In: *SIAM Journal on Computing* 33.3 (2004), pp. 647–658. DOI: 10.1137/S0097539703424521. See p. 62.

[DR90] Paul F. Dietz and Rajeev Raman. "A Constant Update Time Finger Search Tree". In: *International Conference on Computing and Information*. 1990, pp. 100–109. DOI: 10.1007/3-540-53504-7_66. Cited as [DR94].

[DR94] Paul F. Dietz and Rajeev Raman. "A Constant Update Time Finger Search Tree". In: *Information Processing Letters* 52.3 (1994), pp. 147–154. DOI: 10.1016/0020-0190(94)00115-4. From [DR90]. See p. 63.

[DS87] Paul F. Dietz and Daniel D. Sleator. "Two Algorithms for Maintaining Order in a List". In: *Proceedings of the 19th ACM Symposium on Theory of Computing*. 1987, pp. 365–372. DOI: 10.1145/28395.28434. See p. 63.

[DSW05] Jonathan Derryberry, Daniel D. Sleator, and Chengwen Chris Wang. *A Lower Bound Framework for Binary Search Trees with Rotations*. Tech. rep. CMU-CS-05-18. Carnegie Mellon University, 2005. See p. 46.

[ECW92] Vladmir Estivill-Castro and Derick Wood. "A Survey of Adaptive Sorting Algorithms". In: *ACM Computing Surveys* 24.4 (1992), pp. 441–476. DOI: 10.1145/146370.146381. See p. 31.

[EF03] Amr Elmasry and Michael L. Fredman. "Adaptive Sorting and the Information Theoretic Lower Bound". In: *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*. 2003, pp. 654–662. DOI: 10.1007/3-540-36494-3_57. Cited as [EF08].

[EF08] Amr Elmasry and Michael L. Fredman. "Adaptive Sorting: An Information Theoretic Perspective". In: *Acta Informatica* 45.1 (2008), pp. 33–42. DOI: 10.1007/s00236-007-0061-0. From [EF03]. See p. 31.

[Eli75] Peter Elias. "Universal Codeword Sets and Representations of Integers". In: *IEEE Transaction on Information Theory* 21.2 (1975), pp. 194–203. See p. 31.

[Elm04] Amr Elmasry. "On the Sequential Access Theorem and Deque Conjecture for Splay Trees". In: *Theoretical Computer Science* 314.3 (2004), pp. 459–466. DOI: 10.1016/j.tcs.2004.01.019. See pp. 28, 61.

[Fle93] Rudolf Fleischer. "A Simple Balanced Search Tree with O(1) Worst-Case Update Time". In: *Proceedings of the 4th International Symposium on Algorithms and Computation*. 1993, pp. 138–146. DOI: 10.1007/3-540-57568-5_243. Cited as [Fle96].

[Fle96] Rudolf Fleischer. "A Simple Balanced Search Tree with O(1) Worst-Case Update Time". In: *International Journal of Foundations of Computer Science* 7.2 (1996), pp. 137–149. DOI: 10.1142/S0129054196000117. From [Fle93]. See p. 61.

[FNTVW90] Khun Yee Fung, Tina M. Nicholl, Robert E. Tarjan, and Christopher J. Van Wyk. "Simplified Linear-Time Jordan Sorting and Polygon Clipping". In: *Information Processing Letters* 35.2 (1990), pp. 85–92. DOI: 10.1016/0020-0190(90)90111-A. See p. 41.

[Geo05] George F. Georgakopoulos. "How to Splay for loglog$N$-Competitiveness". In: *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms*. 2005, pp. 570–579. Later [Geo08]. See p. 56.

[Geo08] George F. Georgakopoulos. "Chain-splay Trees, or, How to Achieve and Prove loglog$N$-Competitiveness by Splaying". In: *Information Processing Letters* 106.1 (2008), pp. 37–43. DOI: 10.1016/j.ipl.2007.10.001. From [Geo05]. See p. 56.

[GMPR77] Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. "A New Representation for Linear Lists". In: *Proceedings of the 9th ACM Symposium on Theory of Computing*. 1977, pp. 49–60. DOI: 10.1145/800105.803395. See pp. 18, 30, 58–61, 106.

[Gol08] Daniel Golovin. "Uniquely Represented Data Structures with Applications to Privacy". CMU-CS-08-135. PhD thesis. Carnegie Mellon University, 2008. See p. 88.

[GS78] Leonidas J. Guibas and Robert Sedgewick. "A Dichromatic Framework for Balanced Trees". In: *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*. 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3. See pp. 17, 27, 54, 55, 87.

[Har06] Dion Harmon. "New Bounds on Optimal Binary Search Trees". PhD thesis. Massachusetts Institute of Technology, 2006. See pp. 43, 46, 54, 138.

[Har80] Dov Harel. "Efficient Algorithms with Threaded Balanced Trees". PhD thesis. University of California, Irvine, 1980. See p. 61.

[Hib62] Thomas N. Hibbard. "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting". In: *Journal of the ACM* 9.1 (1962), pp. 13–28. DOI: 10.1145/321105.321108. See p. 13.

[HL72] Frank K. Hwang and Shen Lin. "A Simple Algorithm for Merging Two Disjoint Linearly Ordered Sets". In: *SIAM Journal on Computing* 1.1 (1972), pp. 31–39. DOI: 10.1137/0201004. See p. 92.

[HM81] Scott Huddleston and Kurt Mehlhorn. "Robust Balancing in B-Trees". In: *The 5th GI-Conference on Theoretical Computer Science*. 1981, pp. 234–244. DOI: 10.1007/BFb0017315. See p. 59.

[HM82] Scott Huddleston and Kurt Mehlhorn. "A New Data Structure for Representing Sorted Lists". In: *Acta Informatica* 17 (1982), pp. 157–184. DOI: 10.1007/BF00288968. See pp. 16, 28, 29, 59, 106.

[HM84] Kurt Hoffman and Kurt Mehlhorn. "Intersecting a Line and a Simple Polygon". In: *Bulletin of the European Association for Theoretical Computer Science* 22 (1984), pp. 120–121. Cited as [HMRT86].

[HMRT86] Kurt Hoffman, Kurt Mehlhorn, Pierre Rosenstiehl, and Robert E. Tarjan. "Sorting Jordan Sequences in Linear Time Using Level-Linked Search Trees". In: *Information and Control* 68.1–3 (1986), pp. 170–184. DOI: 10.1016/S0019-9958(86)80033-X. Merge. From [HM84]. See p. 41.

[HP06] Ralf Hinze and Ross Paterson. "Finger Trees: A Simple General-Purpose Data Structure". In: *Journal of Functional Programming* 16.2 (2006), pp. 197–217. DOI: 10.1017/S0956796805005769. See p. 61.

[HT71] John E. Hopcroft and Robert E. Tarjan. "Planarity Testing in $V \log V$ Steps: Extended Abstract". In: *Information Processing 71*. Proceedings of IFIP Congress, Volume 1-Foundations and Systems. 1971, pp. 85–90. See p. 41.

[HT74] John E. Hopcroft and Robert E. Tarjan. "Efficient Planarity Testing". In: *Journal of the ACM* 21.4 (1974), pp. 549–568. DOI: 10.1145/321850.321852. See p. 41.

[Hud81] Scott Huddleston. "Robust Balancing in B-Trees". PhD thesis. University of Washington, Seattle, 1981. See p. 16.

[Iac01] John Iacono. "Alternatives to Splay Trees with $O(\log n)$ Worst-Case Access Times". In: *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*. 2001, pp. 516–522. See p. 64.

[Iac02] John Iacono. "Key Independent Optimality". In: *Proceedings of the 13th International Symposium on Algorithms and Computation*. 2002, pp. 223–237. DOI: 10.1007/3-540-36136-7_3. Later [Iac05]. See p. 54.

[Iac05] John Iacono. "Key-Independent Optimality". In: *Algorithmica* 42.1 (2005), pp. 3–10. DOI: 10.1007/s00453-004-1136-8. Cited as [Iac02].

[Kap97] Haim Kaplan. "Purely Functional Lists". PhD thesis. Princeton University, 1997. See pp. 58, 60.

[KE06] Jussi Kujala and Tapio Elomaa. "Poketree: A Dynamically Competitive Data Structure with Good Worst-Case Performance". In: *Proceedings of the 17th International Symposium on Algorithms and Computation*. 2006, pp. 277–288. DOI: 10.1007/11940128_29. See p. 57.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*. 3rd ed. Vol. 3. Prentice Hall, 1998. ISBN: 0-201-89683-4. See p. 16.

[Kos81] S. Rao Kosaraju. "Localized Search in Sorted Lists". In: *Proceedings of the 13th ACM Symposium on Theory of Computing*. 1981, pp. 62–69. DOI: 10.1145/800076.802458. See pp. 22, 59, 60, 107.

[KP07] Markus Kuba and Alois Panholzer. "The Left-Right-Imbalance of Binary Search Trees". In: *Theoretical Computer Science* 370.1–3 (2007), pp. 265–278. DOI: 10.1016/j.tcs.2006.10.033. See p. 62.

[KT96] Haim Kaplan and Robert E. Tarjan. "Purely Functional Representations of Catenable Sorted Lists". In: *Proceedings of the 28th ACM Symposium on Theory of Computing*. 1996, pp. 202–211. DOI: 10.1145/237814.237865. See pp. 58, 60.

[LO88] Christos Levcopoulos and Mark H. Overmars. "A Balanced Search Tree with $O(1)$ Worst-Case Update Time". In: *Acta Informatica* 26.3 (1988), pp. 269–277. DOI: 10.1007/BF00299635. See p. 63.

[Luc88a] Joan M. Lucas. *Arbitrary Splitting in Splay Trees*. Tech. rep. DSC-TR-234. Rutgers, The State University of New Jersey, 1988. See pp. 54, 138.

[Luc88b] Joan M. Lucas. *Canonical Forms for Competitive Binary Search Tree Algorithms*. Tech. rep. DCS-TR-250. Rutgers, The State University of New Jersey, 1988. See pp. 42, 43.

[Man84] Heikki Mannila. "Measures of Presortedness and Optimal Sorting Algorithms". In: *Proceedings of the 11th International Colloquium on Automata, Languages and Programming*. 1984, pp. 324–336. DOI: 10.1007/3-540-13345-3_29. Cited as [Man85].

[Man85] Heikki Mannila. "Measures of Presortedness and Optimal Sorting Algorithms". In: *IEEE Transactions on Computers* 34.4 (1985), pp. 318–325. From [Man84]. See pp. 30, 31.

[MR98] Conrado Martínez and Salvador Roura. "Randomized Binary Search Trees". In: *Journal of the ACM* 45.2 (1998), pp. 288–323. DOI: 10.1145/274787.274812. From [RM96]. See p. 62.

[MS81] David Maier and Sharon C. Salveter. "Hysterical B-Trees". In: *Information Processing Letters* 12.4 (1981), pp. 199–202. DOI: 10.1016/0020-0190(81)90101-0. See p. 17.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. ISBN: 0262631814. See pp. 105, 108.

[NW73] Jürg Nievergelt and C. K. Wong. "Upper Bounds for the Total Path Length of Binary Trees". In: *Journal of the ACM* 20.1 (1973), pp. 1–6. DOI: 10.1145/321738.321739. See p. 6.

[Oka96] Chris Okasaki. "Purely Functional Data Structures". PhD thesis. Carnegie Mellon University, 1996. See p. 58.

[Oli80] Henk J. Olivié. "A Study of Balanced Binary Trees and Balanced One-Two Trees". PhD thesis. University of Antwerp, U.I.A., 1980. See p. 17.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*. Lecture Notes in Computer Science 156. Springer-Verlag, 1983. DOI: 10.1007/BFb0014927. See p. 2.

[OW90] Thomas Ottmann and Derick Wood. "How to Update a Balanced Binary Tree with a Constant Number of Rotations". In: *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory*. 1990, pp. 122–131. DOI: 10.1007/3-540-52846-6_83. Cited as [OW92].

[OW92] Thomas Ottmann and Derick Wood. "Updating Binary Trees with Constant Linkage Cost". In: *International Journal of Foundations of Computer Science* 3.4 (1992), pp. 479–501. DOI: 10.1142/S0129054192000243. From [OW90]. See p. 17.

[Pet08] Seth Pettie. "Splay trees, Davenport-Schinzel sequences, and the Deque Conjecture". In: *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*. 2008, pp. 1115–1124. See pp. 54, 61.

[PM92] Ola Petersson and Alistair Moffat. "A Framework for Adaptive Sorting". In: *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory*. 1992, pp. 422–433. DOI: 10.1007/3-540-55706-7_38. Cited as [PM95].

[PM95] Ola Petersson and Alistair Moffat. "A Framework for Adaptive Sorting". In: *Discrete Applied Mathematics* 59 (1995), pp. 153–179. DOI: 10.1016/0166-218X(93)E0160-Z. From [PM92]. See p. 31.

[Pug89] William Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Proceedings of the 1st Workshop on Algorithms and Data Structures*. 1989, pp. 437–449. Cited as [Pug90b].

[Pug90a] William Pugh. *A Skip List Cookbook*. Tech. rep. CS-TR-2286.1. University of Maryland, College Park, 1990. See p. 62.

[Pug90b] William Pugh. "Skip Lists: a Probabilistic Alternative to Balanced Trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676. DOI: 10.1145/78973.78977. From [Pug89]. See p. 62.

[RM96] Salvador Roura and Conrado Martínez. "Randomization of Search Trees by Subtree Size". In: *Proceedings of the 4th Annual European Symposium on Algorithms*. 1996, pp. 91–106. DOI: 10.1007/3-540-61680-2_49. Later [MR98]. See p. 62.

[SA96] Raimund Seidel and Cecilia R. Aragon. "Randomized Search Trees". In: *Algorithmica* 16 (1996), pp. 464–497. DOI: 10.1007/s004539900061. From [AS89]. See pp. 62, 108.

[Sed08] Robert Sedgewick. "Left-Leaning Red-Black Trees". In: *Dagstuhl Workshop on Data Structures*. Wadern, Germany 2008. See p. 17.

[Sed98] Robert Sedgewick. *Algorithms in C++ Parts 1–4*. 3rd ed. Addison Wesley, 1998. ISBN: 0-201-35088-2. See p. 17.

[ST83] Daniel D. Sleator and Robert E. Tarjan. "Self-Adjusting Binary Trees". In: *Proceedings of the 15th ACM Symposium on Theory of Computing*. 1983, pp. 235–245. DOI: 10.1145/800061.808752. Later [ST85b]. See p. 42.

[ST85a] Daniel D. Sleator and Robert E. Tarjan. "Amortized Efficiency of List Update and Paging Rules". In: *Communications of the ACM* 28.2 (1985), pp. 202–208. DOI: 10.1145/2786.2793. See pp. 18, 42.

[ST85b] Daniel D. Sleator and Robert E. Tarjan. "Self-Adjusting Binary Search Trees". In: *Journal of the ACM* 32 (1985), pp. 652–686. DOI: 10.1145/3828.3835. From [ST83]. See pp. ii, 1, 19, 21, 28, 33, 42, 44, 54, 55, 61, 106, 107.

[STT86] Daniel D. Sleator, Robert E. Tarjan, and William P. Thurston. "Rotation Distance, Triangulations, and Hyperbolic Geometry". In: *Proceedings of the 18th ACM Symposium on Theory of Computing*. 1986, pp. 122–135. DOI: 10.1145/12130.12143. See p. 43.

[Sun89] Rajamani Sundar. "Twists, Turns, Cascades, Deque Conjecture, and Scanning Theorem". In: *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*. 1989, pp. 555–559. DOI: 10.1109/SFCS.1989.63534. Cited as [Sun92].

[Sun92] Rajamani Sundar. "On the Deque Conjecture for the Splay Algorithm". In: *Combinatorica* 12.1 (1992), pp. 95–124. DOI: 10.1007/BF01191208. From [Sun89]. See pp. 28, 54, 61.

[Tar71] Robert E. Tarjan. "An Efficient Planarity Algorithm". STAN-CS-244-71. PhD thesis. Stanford University, 1971. See p. 41.

[Tar83a] Robert E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics 44. SIAM, 1983. ISBN: 0-89871-187-8. See pp. 6, 17.

[Tar83b] Robert E. Tarjan. "Updating a Balanced Search Tree in $O(1)$ Rotations". In: *Information Processing Letters* 16.5 (1983), pp. 253–257. DOI: 10.1016/0020-0190(83)90099-6. See p. 17.

[Tar85] Robert E. Tarjan. "Sequential Access in Splay Trees Takes Linear Time". In: *Combinatorica* 5 (1985), pp. 367–378. DOI: 10.1007/BF02579253. See pp. 28, 54, 55, 61.

[Tsa84] Athanasios K. Tsakalidis. "AVL-Trees for Localized Search". In: *Proceedings of the 11th International Colloquium on Automata, Languages and Programming*. 1984, pp. 473–485. DOI: 10.1007/3-540-13345-3_44. Cited as [Tsa85].

[Tsa85] Athanasios K. Tsakalidis. "AVL-Trees for Localized Search". In: *Information and Control* 67 (1985), pp. 173–194. DOI: 10.1016/S0019-9958(85)80034-6. From [Tsa84]. See pp. 34, 60, 87.

[TVW86] Robert E. Tarjan and Christopher J. Van Wyk. "A Linear-Time Algorithm for Triangulating Simple Polygons". In: *Proceedings of the 18th ACM Symposium on Theory of Computing*. 1986, pp. 380–388. DOI: 10.1145/12130.12170. Cited as [TVW88].

[TVW88] Robert E. Tarjan and Christopher J. Van Wyk. "An $O(n \log\log n)$-Time Algorithm for Triangulating a Simple Polygon". In: *SIAM Journal on Computing* 17.1 (1988), pp. 143–178. DOI: 10.1137/0217010. From [TVW86]. See pp. 34, 35, 37, 40, 41, 60, 65, 87, 93.

[Vit08] Jeffrey S. Vitter. *Algorithms and Data Structures for External Memory*. now Publishers Inc., 2008. ISBN: 978-1601981066. See p. 17.

[Vui80] Jean Vuillemin. "A Unifying Look at Data Structures". In: *Communications of the ACM* 23.4 (1980), pp. 229–239. DOI: 10.1145/358841.358852. See p. 62.

[Wan06] Chengwen Chris Wang. "Multi-Splay Trees". PhD thesis. Carnegie Mellon University, 2006. See pp. 21, 55, 56, 149.

[WDS06] Chengwen Chris Wang, Jonathan Derryberry, and Daniel D. Sleator. "$O(\log\log n)$-Competitive Dynamic Binary Search Trees". In: *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*. 2006, pp. 374–383. DOI: 10.1145/1109557.1109600. See pp. ii, 1, 55, 56.

[Wei99] Mark Allen Weiss. *Data Structures & Algorithm Analysis in C++*. 2nd ed. Addison Wesley, 1999. ISBN: 0-201-36122-1. See p. 17.

[Wil86] Robert E. Wilber. "Lower Bounds for Accessing Binary Search Trees with Rotations". In: *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*. 1986, pp. 61–70. DOI: 10.1109/SFCS.1986.28. Later [Wil89]. See pp. 42, 44, 46.

[Wil89] Robert E. Wilber. "Lower Bounds for Accessing Binary Search Trees with Rotations". In: *SIAM Journal on Computing* 18.1 (1989), pp. 56–67. DOI: 10.1137/0218004. From [Wil86]. See p. 45.

(135 citations)

☺

Author Index

Concept Index