Pure Binary Finger Search Trees*

Gerth Stølting Brodal† Casper Moldrup Rysgaard‡

Abstract

We present dynamic binary search trees where each node only stores a value and pointers to its parent and its children. We denote such binary search trees *pure* binary search trees. Our structure supports finger searches in worst-case $\mathcal{O}(\lg d)$ time, where d is the rank difference between the node given by the finger and the node found by the search. Inserting a new node with a successor or predecessor value for a node pointed to by a finger and deleting a node in the tree pointed to by a finger are supported in amortized $\mathcal{O}(1)$ time and worst-case $\mathcal{O}(\lg n)$ time, where n is the number of nodes in the tree. The temporary working space during the operations is $\mathcal{O}(1)$ words. The result is obtained by an alternative representation of the red-black trees by Guibas and Sedgewick [FOCS 1978] that encodes bits of information in the tree structure, generalizing the encoding of 2-3-trees by Brown [IPL 1979], and rearranging the nodes in a red-black tree ("folding" left and right paths) such that the predecessor and successor of a node can always be found in worst-case constant time. The same time bounds can easily be obtained by, say, red-black trees and AVL trees augmented with pointers to the predecessor and successor of each node. The novelty of our result is that we store no extra information than the binary tree structure. The structure can be represented by two pointers per value, i.e., the same representation as a doubly linked list.

Keywords: Binary search trees, finger searches, dynamic data structure, information encoding

1 Introduction

In this paper we present the first *pure* binary search trees supporting finger searches and finger updates in optimal time. By a pure binary search tree we denote a binary search tree (BST) where each node only stores a single value and pointers to the children and parent. No additional pointers or bits of information is allowed to be stored in a node. All previous finger search trees require more information to be stored for each node to achieve matching time bounds.

The overall goal of the presented research is to study how efficient operations one can achieve for BSTs if the representation should be as simple as a pure BST. The goal of the research presented in this paper became more general over time. Our final construction solves each of the following open problems. *i*) The initial open problem was the following: Some search trees, e.g., red-black trees [26], support the insertion of new values in a tree with n values in worst-case $\mathcal{O}(\lg n)$ time[1], but if the insertion point of the new value (empty leaf) is already known, the insertion can be performed in amortized constant time. But red-black trees store balance information at each node (a bit indication the color red or black). The first problem asked was if there exist balanced pure BSTs without any balance information at the nodes (like splay trees [41], scapegoat trees [3, 23], or encoded 2-3 trees [14, 15]) that could achieve matching time bounds for searching and adding a new leaf? *ii*) Adding a new leaf might be a canonical operation on a BST, but a more canonical black-box interface is to add a predecessor or successor value for an existing node in the search tree, like in a doubly linked list (see Figure 1). Can such insertions be supported in constant time in a balanced BST? In a balanced BST the initial step of an insertion (before some rebalancing is performed to ensure logarithmic height) is to insert the new value at the leftmost empty leaf in the right subtree of the node (see Figure 2). But finding this leaf can take worst-case $\Omega(\lg n)$ time, provided no shortcut pointers are maintained to the successor and predecessor of a node. *iii*) In addition to insertions, can deletions of arbitrary nodes be supported in constant time, like in a doubly linked list (see Figure 1)? In a balanced BST the deletion of a node with two children usually first swaps the node with its predecessor or successor, that takes worst-case $\Omega(\lg n)$ time to find, such that the deletion is reduced to the deletion of a node with at most one child. *iv*) Finally, can we in addition to $\mathcal{O}(\lg n)$ time top-down searches in a balanced pure BST

*Supported by Independent Research Fund Denmark, grant 9131-00113B.

†Aarhus University, Deparment of Computer Science, gerth@cs.au.dk. ORCID 0000-0001-9054-915X.

‡Aarhus University, Deparment of Computer Science, rysgaard@cs.au.dk. ORCID 0000-0002-3989-123X.

[1] $\lg n$ denotes the binary logarithm of n.

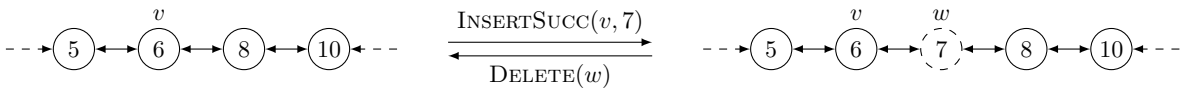


Figure 1: Inserting 7 as a new successor node w to an existing node v in a doubly linked list, and reversely the deletion of a node w from a doubly linked list.

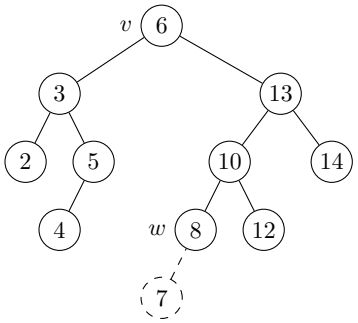


Figure 2: Performing InsertSucc$(v, 7)$ inserts 7 as a successor of 6 in a BST by inserting 7 as a predecessor (left child) of the successor w of v.

also support finger searches in $\mathcal{O}(\lg d)$ time (d being the rank difference between the start and end nodes of a search)? Previous finger search trees require additional pointers to achieve this goal, e.g., like [27, 12].

1.1 Search tree operations In this section we give a more formal definition of the BST operations we consider in this paper. We assume that a BST stores n ordered values $e_1 \leq \cdots \leq e_n$ in nodes $v_1, \ldots, v_n$, respectively. Note that we do not require the values stored to be distinct, i.e., the same value can be stored in multiple nodes. We assume that the n nodes form a BST, i.e., v_i is the ith node in an inorder traversal of the tree. We say that e_i and v_i have *rank* i, i.e., if there are multiple nodes with equal value, the rank of a value is defined by the index of the node it is stored in. We only consider "stable" BSTs, i.e., whenever a value is inserted into a node and a pointer to this node is returned, then the value resides in the node until deleted from the BST. See [7] for a discussion of stability in practical data structure libraries. The operations we consider supported are:

Pred(v) Takes a pointer v to a node v_i. Returns a pointer to v_{i-1}, if $i > 1$, and nil otherwise.

Succ(v) Takes a pointer v to a node v_i. Returns a pointer to v_{i+1}, if $i < n$, and nil otherwise.

InsertPred(v, e) Takes a pointer to a node v and a value e. Creates a new node v' storing e. If $v =$ nil, v' is a new tree by itself. Otherwise assume v points to v_i. It is a precondition that $e_{i-1} \leq e \leq e_i$ (assuming $e_0 = -\infty$). Inserts v' in the tree, such that v' is after v_{i-1} (if v_{i-1} exists) and before v_i in the new inorder of the tree. Returns a pointer to v'.

InsertSucc(v, e) Takes a pointer to a node v and a value e. Creates a new node v' storing e. If $v =$ nil, v' is a new tree by itself. Otherwise assume v points to v_i. It is a precondition that $e_i \leq e \leq e_{i+1}$ (assuming $e_{n+1} = \infty$). Inserts v' in the tree, such that v' is after v_i and before v_{i+1} (if v_{i+1} exists) in the new inorder of the tree. Returns a pointer to v'.

Delete(v) Takes a pointer to a node v. Removes the node v and its associated value from the tree containing v.

FingerSearch(v, e) Takes a pointer v to a node v_i and a value e. If $e < e_1$, returns a pointer to v_1. Otherwise, returns a pointer to the node v_j, such that $e_j \leq e < e_{j+1}$ (assuming $e_{n+1} = \infty$). For a finger search, we let $d = |j - i|$ denote the rank difference between the start node and the node returned by the search.

We denote the three operations InsertPred, InsertSucc and Delete as *finger updates*.

1.2 Binary search trees BSTs are one of earliest data structures in computer science, see, e.g., Windley [47] from 1960 for an early description of BSTs. Knuth [31, page 453] gives an account of the early history of BSTs, and Andersson et al. [4] of later developments on balanced BSTs.

The classic AVL trees of Adelson-Velsky and Landis [1] support insertions, deletions and searches in $\mathcal{O}(\lg n)$ time. Mehlhorn and Tsakalidis [35] showed that the amortized restructuring cost of AVL trees after an insertion is amortized $\mathcal{O}(1)$ time, i.e., an AVL tree supports finger insertions in amortized constant time if the AVL tree is augmented with successor/predecessor pointers. The red-black trees of Guibas and Sedgewick [26] achieve matching $\mathcal{O}(\lg n)$ worst-case bounds. Tarjan [44] proved that the rebalancing required by both insertions and deletions in a red-black tree is amortized constant time, i.e., a red-black tree supports finger updates in amortized constant time provided the red-black tree is augmented with successor/predecessor pointers. Insertions and deletions perform at most two and three rotations, respectively [43, 44].

The splay trees of Sleator and Tarjan [41] are arguably the simplest BST representation: Each node stores a value and two pointers to the children. They support insertions, deletions and searches in amortized $\mathcal{O}(\lg n)$ time. Even when a finger insertion can be done in worst-case constant time in a splay tree, as shown in Figure 3, the insertions must be charged logarithmic potential increase to support amortized logarithmic queries, i.e., the amortized insertion time is logarithmic.

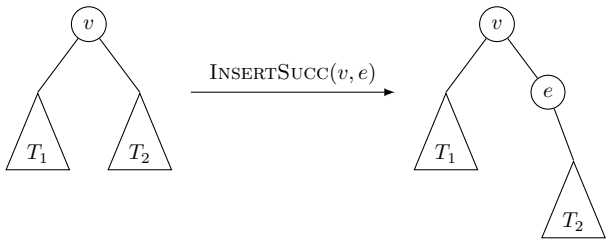


Figure 3: InsertSucc(v, e) in a splay tree.

Levcopoulos and Overmarks [33] describe BSTs supporting finger updates in worst-case $\mathcal{O}(1)$ time and searches in $\mathcal{O}(\lg n)$ time, essentially by maintaining bags of $\Theta\big(\lg^2 n\big)$ values.

Seidel and Aragon [40] presented simple randomized search trees denoted *treaps*, where a random priority is assigned to each value and the BSTs are heap ordered with respect to the priorities. Treaps support updates and searches in expected $\mathcal{O}(\lg n)$ time, and store expected $\mathcal{O}(1)$ bits at each node, and with high probability at most $\mathcal{O}(\lg n)$ bits. Martínez and Roura [34] considered a variant of treaps where each node instead of a random priority only stores the subtree size. Subsequently, Seidel [39] presented a variant of treaps where nodes store no balance information, i.e., they are pure BSTs, but the update time increases to expected $\mathcal{O}\big(\lg^2 n\big)$ time. Gila, Goodrich and Tarjan [24] presented zip-zip trees, that similar to treaps store random bits at the nodes of a BST, but they reduce the high probability bound on the number of random bits to $\mathcal{O}(\lg \lg n)$.

Andersson [3] and indenpendently Galperin and Rivest [23] presented the scapegoat trees, that are essentially pure BSTs, except for a global variable storing the tree size n. The invariant is that the tree has height $\mathcal{O}(\lg n)$, which is maintained by rebuilding subtrees whenever the height invariant is violated.

Brown [14, 15] presented a pure BST supporting updates and searches in $\mathcal{O}(\lg n)$ time, by encoding a 2-3 tree [2] in the structure of a pure BST, where pairs of nodes encode bits of balance information for a 2-3 tree node. It is crucial for the decoding that operations start at the root, i.e., finger operations cannot be supported.

1.3 Finger search data structures A finger search in a sorted array can be implemented by an exponential search as shown in Figure 4, consisting of a doubling phase followed by a binary search phase, in total performing $2 \lg d + \mathcal{O}(1)$ comparisons.

Bentley and Yao [8] studied the comparison complexity of finger searches. They proved that a finger search requires $\lg^{(1)} d + \lg^{(2)} d + \lg^{(3)} d + \cdots + \lg^{(\lg^* d)} d \pm \Theta(\lg^* d)$ comparisons, where $\lg^{(1)} d = \lg d$, $\lg^{(i+1)} d = \lg\big(\lg^{(i)} d\big)$, and $\lg^* d = \min\{i \in \mathbb{N} \mid \lg^{(i)} d \leq 1\}$, i.e., they proved that the number of comparisons performed by exponential search can be improved by essentially a factor two.

A balanced BST, like an AVL tree or a red-black tree, can support a finger search in $\mathcal{O}(\lg d)$ if the finger is at the smallest or largest value in the tree: Move up along the left or right spine of the tree, respectively, until a

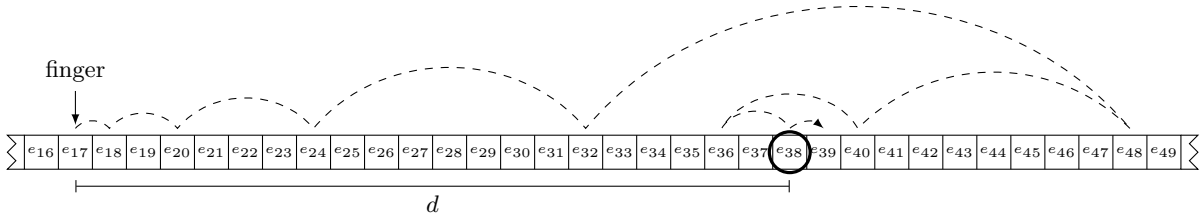


Figure 4: Exponential search for e in a sorted array, where $e_{38} \leq e < e_{39}$.

top-down search can be performed on a single subtree attached to the spine. Building on this observation, Guibas et al. [25] presented search trees based on B-trees supporting $\mathcal{O}(1)$ *moveable* fingers, finger updates in constant time, and finger searches and moving a finger in $\mathcal{O}(\lg d)$ time. Kosaraju [32] achieved a similar result based on 2-3-trees, Tsakalidis [46] a solution based on AVL trees, and Tarjan and van Wyk [45] a solution based on red-black trees. Huddleston and Mehlhorn [27] presented a solution based on *level-linked* (a, b)-trees, supporting finger updates in amortized constant time and finger searches in worst-case $\mathcal{O}(\lg d)$ time—without any restriction on the number of fingers. Their structure requires five pointers per node. Brodal et al. [12] presented an involved pointer based finger search structure supporting finger updates in worst-case constant time and finger searches in worst-case $\mathcal{O}(\lg d)$ time.

Treaps [40] support finger updates in expected constant time and finger searches in expected $\mathcal{O}(\lg d)$ time, essentially since the random structure of treaps implies that a finger is expected to be close to a leaf. Splay trees were analyzed in the context of finger searches by Cole et al. [16, 17], who proved that splay trees support searches, insertions, and deletions in amortized $\mathcal{O}(\lg d)$ time, where d is the rank distance from the last accessed node.

Deviating from pointer based structures, Anderson and Thorup [5] presented optimal finger search trees in the RAM model supporting finger updates in constant time and finger searches in $\mathcal{O}\big(\sqrt{\lg d / \lg \lg d}\big)$ time. Kaporis et al. [30] considered interpolation based finger search structures on the RAM model achieving $\mathcal{O}(\lg \lg d)$ expected search time for a large class of input distributions.

For more results on finger search data structures, see the survey by Brodal [10]. Table 1 summarizes the finger update bounds for various BSTs supporting an arbitrary number of fingers and the amount of balance information they store in each node.

1.4 Result Our main result is essentially an alternative representation of red-black trees achieving:

THEOREM 1.1. *There exist pure binary finger search trees supporting* PRED *and* SUCC *in worst-case* $\mathcal{O}(1)$ *time,* INSERTPRED, INSERTSUCC, *and* DELETE *in amortized* $\mathcal{O}(1)$ *time and worst-case* $\mathcal{O}(\lg n)$ *time, and* FINGERSEARCH *in worst-case* $\mathcal{O}(\lg d)$ *time, where* n *is the number of nodes in the tree before the operation and* d *is the rank difference between the node given by the finger and the node found by the search. The tree can be represented by* n *records, where each node stores a value and two pointers (a pointer to the left child, and a pointer to either the right sibling or the parent). The temporary working space during the operations is* $\mathcal{O}(1)$ *words.*

The result is achieved by combining four simple ideas: *i*) Red-black trees support finger searches in asymptotic optimal time if the successor and predecessor of a node can be accessed in constant time; *ii*) rearranging the nodes in a red-black tree by "folding" left and right paths (adding a single bit to the nodes but no additional pointers), successor and predecessor nodes can be located in $\mathcal{O}(1)$ time, while still allowing access to the parent and children of a red-black tree node in $\mathcal{O}(1)$ time; *iii*) by partioning the nodes of a pure BST into "metanodes", $\mathcal{O}(1)$ bits of information can be encoded by the tree structure of each metanode; and *iv*) by ensuring all unary nodes have a left child, two pointers are sufficient to represent each node.

1.5 Outline of paper The paper is organized top-down gradually introducing details to keep a focus of the contribution of the individual ideas to our construction. In Section 2 we first recall the definition and operations on red-black tree. In Section 3 we describe how red-black trees support optimal finger searches, provided that

Reference	Finger updates	Information stored
AVL trees [1]	$\mathcal{O}(\lg n)$	2 bits per node
Red-black trees [26]	$\mathcal{O}(\lg n)$	1 bit per node
Red-black trees + predecessor/successor links [44]	$\mathcal{O}_A(1)$	$\mathcal{O}(\lg n)$ bits per node
Scapegoat trees [3, 23]	$\mathcal{O}_A(\lg n)$	global tree size
Level-linked (a, b)-trees [27]	$\mathcal{O}_A(1)$	$\mathcal{O}(\lg n)$ bits per node
BST with $\mathcal{O}(\lg^2 n)$ sized bags [33]	$\mathcal{O}(1)$	$\mathcal{O}(\lg n)$ bits per node
Treaps [40]	$\mathcal{O}_E(1)$	$\mathcal{O}_E(1)$ bits per node
Seidel random BST [39]	$\mathcal{O}_E(\lg^2 n)$	none
Splay trees [41]	$\mathcal{O}_A(\lg n)$	none
Encoded 2-3 trees [14, 15]	$\mathcal{O}(\lg n)$	none
This paper	$\mathcal{O}_A(1)$	none

Table 1: Results for selected balanced BSTs. $\mathcal{O}_A$ and $\mathcal{O}_E$ denote amortized and expected bounds, respectively. All search trees have logarithmic height, and support searches in logarithmic time, except [39, 40] are randomized and bounds are expected, and bounds for [41] are amortized, with no bound on the tree height.

red-black trees are augmented to support finding the successor and predecessor of a node in worst-case constant time. In Section 4 we describe how paths in red-black trees can be rearranged such that the resulting tree is still a BST with two bits of information at each node, but no additional pointers, where we can navigate and update as in a red-black tree plus have access to the predecessor and successor of a node in constant time. In Section 5 we describe how we can group nodes of a pure BST into *metanodes*, where the metanodes form binary subtrees and encode bits of information through the tree structure, and in Section 6 we describe how to store the resulting pure BST with only two pointers per node. In Section 7 we combine the constructions of the preceding sections to achieve our main result (Theorem 1.1). In Section 8 we argue that it is not possible to achieve worst-case constant time finger updates and logarithmic searches if the structure is a pure BST, but in Section 9 we show that better bounds are possible for general pointer structures where each node only stores two pointers. Finally, in Section 10 we give some concluding remarks and mention some open problems.

2 Red-black trees

In this section we briefly recall the red-black trees of Guibas and Sedgewick [26], and how they support the operations described in Section 1.1. The amortized restructuring analysis of red-black trees is due to Tarjan [44]. The rebalancing operations in Figure 6 and 7 are from Cormen et al. [18].

Each node stores a value, pointers to its left child, right child, and parent, and a single bit indicating if the node is red or black. Red-black trees satisfy the invariants:

1. A red node has a black parent, and
2. the number of black nodes on all root-to-external-leaf paths are the same.

If k denotes the *black height* of the tree, i.e., the number of black nodes on any root-to-leaf path, then the tree contains between $2^k - 1$ and $2^{2k} - 1$ nodes, implying that the height of a red-black tree is $\mathcal{O}(\lg n)$. Figure 5 shows a red-black tree with black height 3.

A shortcoming of red-black trees, and most BSTs, is that the successor and predecessor of a node are not necessarily adjacent to the the node (e.g., in Figure 5 the nodes storing values 6 and 7 are not adjacent). In general there can be $\Theta(\lg n)$ nodes on the paths between a node and its successor and/or predecessor. In Section 3 we describe how a red-black tree can support finger searches in $\mathcal{O}(\lg d)$ time, provided the successor and predecessor of a node can be accessed in $\mathcal{O}(1)$ time, e.g., by adding additional links. In Section 4 we describe how to rearrange the nodes in the representation of a red-black tree such that we can both navigate and update the red-black tree, as if it was stored using the normal representation, and simultaneously have access to the successor and predecessor of a node in $\mathcal{O}(1)$ time, without additional links (but one more bit per node).

A $\textsc{Succ}(v)$ query is supported in $\mathcal{O}(\lg n)$ time as follows: If v has a right child, returns the node found by moving to the right child of v and then repeatedly moving to the left child until a node with no left child is reached. If v does not have a right child, repeatedly moves up to the parent until v is in the left subtree. If such

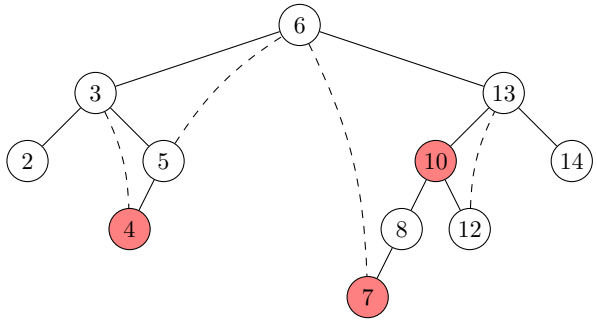


Figure 5: A Red-black tree (red nodes are shaded red; the remaining nodes are black). Dashed lines show where the successor or predecessor of a node is not an adjacent node.

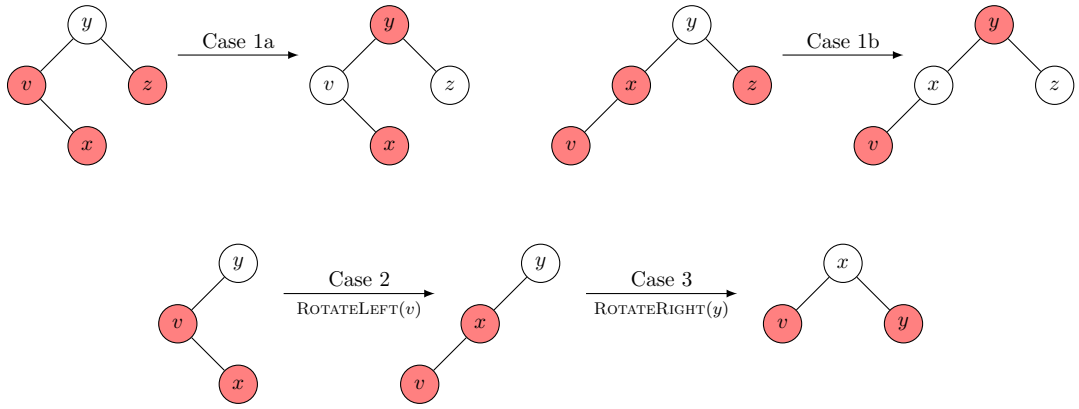


Figure 6: Insertion rebalancing cases on red-black trees, when the two adjacent red nodes are on the left. Omitted subtrees have a black root or are empty. Cases 1a and 1b perform recoloring and continues from node y, while Cases 2 and 3 perform rotations in addition to recoloring, before terminating.

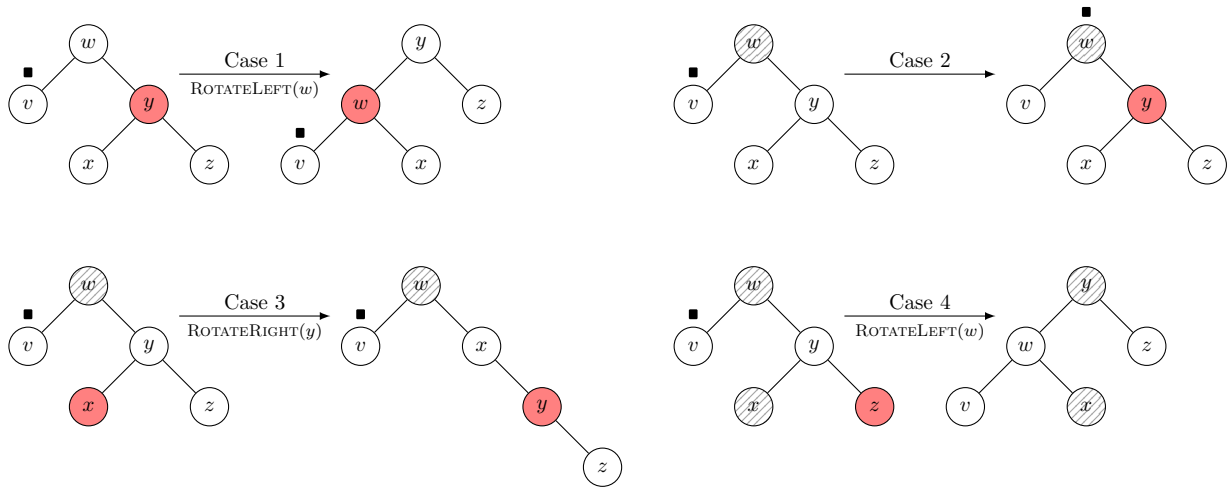


Figure 7: Deletion rebalancing cases on red-black trees, when the double black node (marked ■) is on the left. The dashed nodes denote a node which color can be either red or black. Omitted subtrees can be empty or have a black or red root. The node v can possibly be an empty subtree. Case 2 performs recoloring and continues from node w, while Cases 1, 3, and 4 perform rotations in addition to recoloring. In Case 4, the color of node y after rebalancing is the color of node w before rebalancing.

a node is found, it is returned. Otherwise, v is the rightmost node in the tree and NIL is returned. The query $\textsc{Pred}(v)$ is handled symmetrically.

The finger insertion $\textsc{InsertSucc}(v, e)$ creates a new node v' with value e. If v has no right child, v' becomes the right child of v. Otherwise, v' becomes the left child of $\textsc{Succ}(v)$. In both cases v' becomes a red leaf and the black height invariant is maintained, but potentially v' might have a red parent. This is fixed in amortized $\mathcal{O}(1)$ time by performing amortized $\mathcal{O}(1)$ local transformations on the tree, involving amortized $\mathcal{O}(1)$ recoloring at most two rotations. The transformations are recalled in Figure 6. $\textsc{InsertPred}(v, e)$ is handled symmetrically. A finger deletion $\textsc{Delete}(v)$ first checks if v has two children. If v has at most one child, we delete v from the tree, replacing it by its child if it had one. If v has two children, the values of v and $\textsc{Succ}(v)$ are swapped, and we delete $\textsc{Succ}(v)$, that has at most one child. If the deleted node was red, we are done. Otherwise the black height invariant might be violated, and a sequence of amortized $\mathcal{O}(1)$ transformations are applied where one node is treated having an extra layer of black color (initially the child that replaced the deleted node, possibly a NIL external leaf). The transformations are shown in Figure 7, where the "double black" node is marked with ■. If a red node becomes marked ■, it is made a black node, and the transformations terminate. The transformations perform amortized $\mathcal{O}(1)$ recoloring and at most three rotations. In total, the time for a finger update is amortized $\mathcal{O}(1)$ plus the time for a $\textsc{Succ}$ query (and worst-case $\mathcal{O}(\lg n)$ time).

3 Finger searches using predecessor and successor links

In this section we describe how a balanced binary tree, say a red-black tree or an AVL tree, can support a finger search in $\mathcal{O}(\lg d)$ time, if the predecessor $\textsc{Pred}(v)$ and successor $\textsc{Succ}(v)$ of a node v can be found in constant time. Whereas $\textsc{Pred}(v)$ and $\textsc{Succ}(v)$ can be supported in constant time trivially by augmenting a BST with these as pointers (such that the nodes are also maintained in a sorted doubly linked list), we in Section 4 show how the additional pointers can be avoided for the case of red-black trees by rearranging the nodes.

Let u denote the node to be returned by a finger search $\textsc{FingerSearch}(v, e)$, i.e., the rightmost node in the tree storing a value $\leq e$. In a normal top-down search starting at the root, we would go left whenever a node stores a value larger than the query value e and right otherwise, until we reach an empty leaf. The node u to be returned is the last visited node on the search path with value $\leq e$, i.e., the last node where the search continued to the right child. Our finger search algorithm makes repeated use of such top-down searches on various subtrees.

Our finger search algorithm uses a parameter h, bounding how many levels we are allowed to move up in the tree. Initially h is $\mathcal{O}(1)$. If a search fails because h is too small, we double h, and restart the search. For each choice of h the algorithm will spend $\mathcal{O}(h)$ time, i.e., the total running time will be a geometric sum asymptotically bounded by the final choice of h, that will be $\mathcal{O}(\lg d)$.

In the following we assume the value at v is $\leq e$. The case where the query value e is smaller than the value at v follows by a symmetric argument, where calls to $\textsc{Succ}$ are replaced by calls to $\textsc{Pred}$. A search $\textsc{FingerSearch}(v, e)$ proceeds as follows and as illustrated in Figure 8. If v has a right child, call $\textsc{Succ}(v)$ to find the leftmost node v_s in the right subtree of v. If the value at v_s is $> e$, return v as the answer. If v does not have a right child, let $v_s = v$. From v_s move up the ancestor path h levels to reach node r_ℓ (or until the root is reached). Perform a top-down search in the subtree rooted at r_ℓ. If the top-down search from r_ℓ finds a node with value $> e$, we return the answer u found in the subtree rooted at r_ℓ. Otherwise, we have reached the rightmost node ℓ in the subtree rooted at r_ℓ, and the value at ℓ is $\leq e$. We find $c = \textsc{Succ}(\ell)$. If c is NIL, then ℓ stores the largest value in the tree, and we return ℓ. Otherwise, c is an ancestor of both ℓ and r_ℓ, and ℓ is the rightmost node in the left subtree of c. If c stores a value $> e$, return ℓ. Otherwise, compute $r = \textsc{Succ}(c)$. If r is NIL or stores a value $> e$, return c. Otherwise, from r move up the ancestor path h levels to reach node r_r (or until the root is reached), and perform a top-down search in the subtree rooted at r_r. If the top-down search from r_r finds a node with value $> e$, we return the answer u found in the subtree rooted at r_r. Otherwise, we declare the search to have failed, and restart with a larger h.

For the analysis, we assume all root-to-leaf paths in a subtree are guaranteed to contain the same asymptotic number of nodes. E.g., for red-black trees and AVL trees the difference between the longest and shortest paths is a factor two. For fixed h, the total search time is $\mathcal{O}(h)$ plus the time for the top-down searches from r_ℓ and r_r. For r_ℓ, observe that we reach r_ℓ along an ancestor path of length h from v or $\textsc{Succ}(v)$, where either v has no right child or $\textsc{Succ}(v)$ has no left child, respectively. It follows that the subtree rooted at r_ℓ has height $\mathcal{O}(h)$. For r_r, observe that r either has no left child (r is the leftmost node in the right subtree of c), or c has no right child and c is the left child of r. Since we reach r_r from r along an ancestor of length h, the subtree rooted at r_r

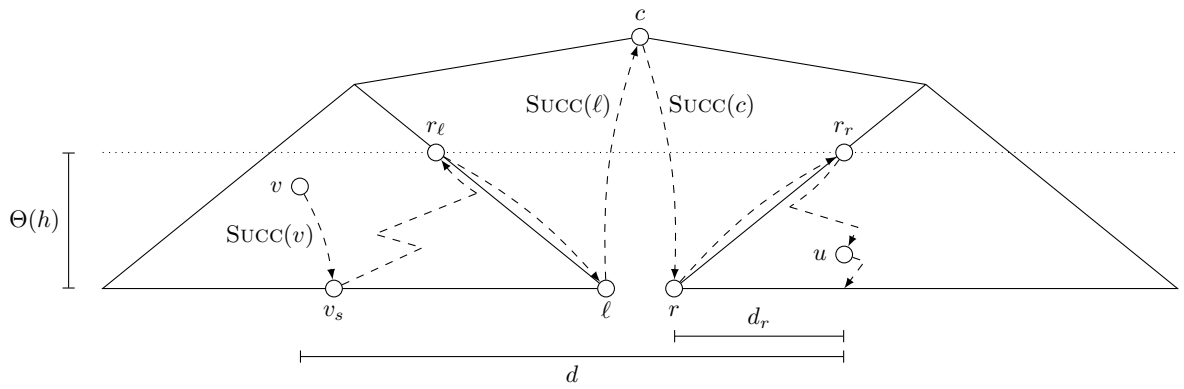


Figure 8: The steps of FingerSearch(v, e) for a fixed h in a BST with successor and predecessor links. The dashed path are the vertices visited and u the returned node.

has height $\mathcal{O}(h)$. For fixed h, this implies that the top-down searches at r_ℓ and r_r take $\mathcal{O}(h)$ time, i.e., $\mathcal{O}(h)$ total time. If a search fails, then the value at r_r is $\leq e$, and the final answer will be r_r or a node to the right of r_r. If r_r is in the right subtree of c, the rank distance d between v and the final node u is at least the rank difference d_r between r and r_r, and d_r equals the number of nodes in left subtree of r_r, where $d_r \geq 2^{\Omega(h)}$. If r_r is c or an ancestor of c (possibly to the left of v), and the search fails, then all $2^{\Omega(h)}$ nodes in the right subtree of c store values $\leq e$, i.e., $d \geq 2^{\Omega(h)}$. In both cases h is $\mathcal{O}(\lg d)$ when the search fails. Since h for the final successful search is twice the last failed h value, it follows that the total time for a finger search is $\mathcal{O}(\lg d)$.

4 Folded red-black trees

In this section we describe the *folded* representation of red-black trees, that we denote *folded red-black trees*, further abbreviated as *folded trees*. We let *unfolded red-black trees* denote regular red-black trees (Section 2), similarly abbreviated *unfolded trees*. The folded tree representation of an unfolded tree is a unique BST on the same set of nodes as the unfolded tree, but with an alternative arrangement. The goal of folded trees is to support access to the predecessor or successor of a node in constant time, access to the parent and children in the corresponding unfolded tree in constant time, and to support finger updates in amortized constant time (and worst-case logarithmic time), *without* adding additional pointers to the nodes of the tree (but with adding one additional bit to each node).

A folded tree consists of folding disjoint *left paths* and *right paths* in the unfolded tree. A left path (right path) is a maximal path of nodes that are left (right) children. The construction is applied recursively, such that the right subtrees of a left folded path are *right folded*, and the left subtrees of a right folded path are *left folded*. The initial fold is on the left path from the root.

Consider a left path in an unfolded tree (right paths are handled symmetrically). If the path contains only a single node, i.e., the node has no left child, the folded path is only this node and its right subtree recursively right folded. Otherwise, we split the unfolded left path in two parts; an *upper* part and a *lower* part. Denote nodes on the upper part as *upper nodes* and nodes on the lower part as *lower nodes*. The lower part is partitioned into connected *lower chunks*. The topmost lower chunk is denoted the *last lower chunk*. See Figure 9 (left) for an illustration of these definitions. The partitioning is created as follows to guarantee that *the number of upper nodes is equal to the number of lower chunks*: Initially, the topmost node of the path is the only upper node and all other nodes constitute the last lower chunk. Repeatedly, split off the lowest black node from the last lower chunk, and possibly its left red child, as a new lower chunk and split off the topmost node as a new upper node, provided the last lower chunk remains non-empty. Each iteration creates one new lower chunk and one new upper node, i.e., the final number of upper nodes equals the number of lower chunks. All lower chunks, except the last lower chunk, by definition consist of a lower black node, and additionally its left child if this node is red. Since in each iteration two or three nodes are removed from the last lower chunk, we can always split of an upper node and a lower chunk while the last lower chunk has four or more nodes, i.e., the final non-empty last lower chunk can at most have three nodes. Furthermore, if it contains three nodes, the lowest node must be red, since otherwise

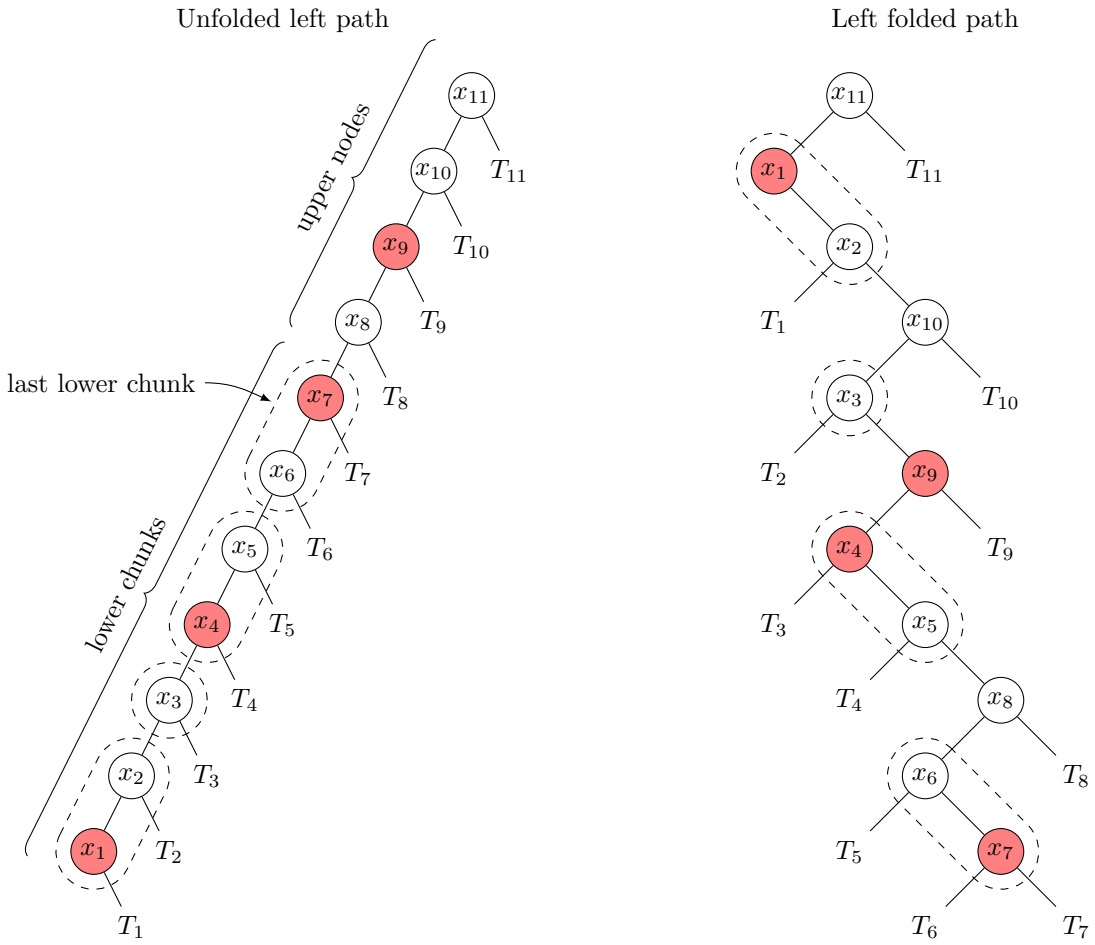


Figure 9: (left) unfolded left path and (right) the folded rearrangement of the path. The dashed areas denote the lower chunks, and the nodes in non-dashed areas denote the upper nodes, with (left) their unfolded positions and (right) their folded positions. T_i is (left) the right subtree of node x_i, and (right) the recursively folded subtree at the new location in the folded path.

we can split off an upper node and a single black node as a lower chunk. We conclude that the last lower chunk contains 1–3 nodes, and only three nodes if the lowest node in the last lower chunk is red. The possible last lower chunks are illustrated in Figure 10.

The left folded path is constructed by alternating upper nodes and lower chunks, starting with the top upper node and ending with the last lower chunk, such that lower chunks are left children of upper nodes, and upper nodes are right children of lower chunks. The top-down order of the lower nodes are reversed in the left folded path, such that if in an unfolded tree a lower chunk contains v and its left child $u = v.\ell$, then in the folded tree v is the right child of u. See Figure 9 for an illustration of an unfolded left path, and the folded representation of that path. Let $x_1, x_2, \ldots$ denote the nodes bottom-up on the unfolded left path. Each node x_i on the left path in the unfolded tree has a right subtree T_i, possibly empty. For each upper node x_i in the folded tree, the right subtree is T_i. Similarly, the right subtree of the node at the end of the last lower chunk is placed as the right child (x_7 and T_7 in Figure 9). Otherwise, the right subtree T_i of a lower node x_i is placed as the left subtree of x_{i+1} (since T_i contains values between x_i and x_{i+1}). For each subtree T_i, we recursively fold the right path.

In the folded tree, when given a node, it cannot be detected locally if the node is on a right or left path. We therefore add one *additional bit* to each node, to mark the *orientation*, i.e., if the node is a left or right child in the unfolded. Using this information, it can be detected in constant time if a node is an upper or lower node: for a left folded path, only upper nodes have left children, which are also on the left path, i.e., have the same

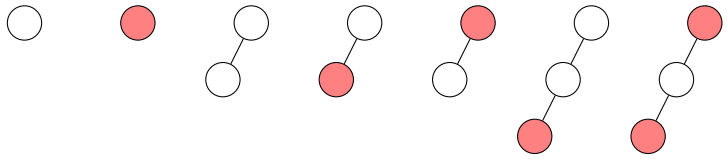


Figure 10: Possible colorings of the last lower chunk on an unfolded left path.

orientation, and symmetrically for the right folded path. If a node defines a path by itself, it can similarly be detected in constant time. Using the information left/right fold and upper/lower, the local lower chunks can be detected locally in constant time.

4.1 Properties of folded trees Having defined the folded representation of an unfolded tree, we summarize the properties of the folded representation and how it supports navigation in the unfolded tree:

- The inorder of the nodes of the unfolded and folded trees are identical,
- left and right paths in the unfolded tree become paths in the folded tree with identical topmost nodes,
- the topmost and bottommost nodes on a left (right) path in the unfolded tree are adjacent in the folded tree, i.e., given the topmost node in a left (right) path, the minimum (maximum) node in the subtree rooted at the node in the unfolded tree can be found in constant time,
- given a node in the folded tree, the left child, right child, and parent of the node in the unfolded tree can be found in constant time, and
- the folded tree is unique for a given unfolded tree,
- the height of the folded tree is logarithmic.

To see that the folded trees preserve inorder, observe the layout of the folded path in Figure 9 (right) satisfies inorder, and apply the argument recursively. By definition a left/right path becomes a folded path with the same topmost node and the topmost and bottommost nodes on left and right paths are adjacent.

Let v be a node in the unfolded tree, which is the left child of some node u, where u is either part of a left or right path. We will argue that the path between u and v in the folded tree contains at most two additional nodes, i.e., we can navigate from v to u and from u to v in constant time. If u is on a left path, then v is on the same left path, and will in the folded path be separated by at most two nodes (two upper nodes are separated by a lower chunk containing one or two nodes; two lower nodes can be adjacent or are separated by one upper node; and the adjacent upper node and lower node by the at most two other nodes in the last lower chunk). If u is on a right path, then v is the root of the left subtree of u and the topmost node on a left path, and will in the folded right path be separated by at most two nodes (a lower and an upper node; see the distance between x_i and the root of T_i in Figure 9 (right)). Symmetrically, this applies to right children.

The folding definition is deterministic. The root of the folded tree is the same node as the root of the unfolded tree. Given a node in the folded tree, the left and right children in the unfolded tree can be found. Therefore there exists a deterministic unfolding algorithm of a folded tree. The folded tree is therefore unique for a given unfolded tree.

To bound the height of a folded tree, let v be a deepest node in the folded tree. Since the root in the folded and unfolded trees are identical, the above argument implies that if the simple path in the unfolded tree from v to the root contains k nodes, there exists a path (not necessarily simple) from v to the root containing at most $k + 2(k-1) = 3k - 2$ nodes. Therefore, the simple path in the folded tree from v to the root contains at most $3k - 2$ nodes. Since unfolded trees have logarithmic depth (Section 2), folded trees also have logarithmic depth.

4.2 Finding the predecessor and successor In this section we consider how to find the predecessor and successor of a node in a folded tree in constant time, even if they do not exist in the corresponding subtree in the unfolded tree, but as an ancestor. We only describe how to find the successor of a node. Finding the predecessor is symmetric.

Denote the starting node as v. If the right subtree of v in the unfolded tree is non-empty, then the successor is the minimum value in this subtree. Denote the right child of v as r, and the left subtree of r as T_r. Note that r must exist, as the right subtree of v is non-empty, and r can be found in constant time in the folded tree (Section 4.1). If T_r is empty, then the successor of v is r. Otherwise the successor is the minimum value in T_r. Independent if v is on a left or right folded path, T_r must be left folded, as illustrated on Figure 11. As T_r is left folded, the minimum in T_r can be accessed in constant time from v (Section 4.1). If the right subtree of v is empty, then the successor of v must be an ancestor u of v, where v is the maximum in the left subtree of u. Note that v is the predecessor of u. By symmetric argument from above for finding the predecessor in a non-empty left subtree, we from u can access v in constant time. As the pointers of the tree can be traversed in both directions, we from v can access its successor u in constant time. This concludes that the successor of any node can be found in constant time.

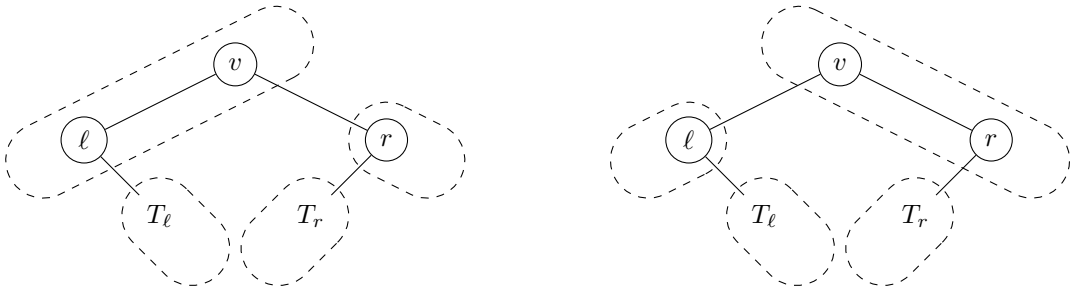


Figure 11: Folding orientations covering node v in an unfolded tree. The dashed areas show if the underlying path is left or right folded, with (left) the path containing v being left folded and (right) right folded. The subtrees T_ℓ and T_r may be empty.

4.3 Translating rotations from unfolded red-black trees An unfolded tree is maintained balanced by performing rotations. When a rotation occurs while rebalancing the unfolded tree, the folded tree must be updated accordingly. How to do so depends on the orientation of the paths in the unfolded tree. Figure 12 shows the four cases how rotations are performed in unfolded trees and how the orientation of the paths may be.

Consider the right rotation on Figure 12 (top). Define the *main path* to be the path containing the node rotated, in this case v, and the *height* of the rotation to be the height of the node rotated in the unfolded tree. In this case the main path loses node v. If v is an upper node or a node in the last lower chunk, then the path needs to be rebuilt, which takes time proportional to the height of the rotation, as the upper nodes are in the top half of the path. If v is a red lower node not in the last lower chunk, then the number of lower chunks is unaltered, and updating the main path takes constant time. The right path starting at T_4 gains a new upper node, the right path starting at w loses an upper node and the left path starting at T_2 gains an upper node, which all need to be rebuilt in time proportional to the height of the rotation. All other paths are unaltered.

Symmetric arguments exist for the other three cases of rotations. The total time spent to perform a rotation is proportional to the height of the rotation in the unfolded tree, provided that the number of lower chunks is unaltered in the case when the node rotated is a lower node.

4.4 Insertions in folded red-black trees In this section we describe how to perform InsertSucc on a folded tree in amortized constant time. Performing InsertPred is symmetric.

For an InsertSucc(v, e) operation, Succ(v) can be found in $\mathcal{O}(1)$ time (Section 4.2). Remaining is to show that the time spent maintaining the folded tree to accommodate the new node in the folded tree is proportional to the time spent on rebalancing the unfolded tree. The rebalancing steps on the folded tree is identical to the rebalancing on unfolded trees (Section 2), utilizing that the local nodes needed in the rebalancing can be found in constant time (Section 4.1). First the successor leaf of v is located to create the new node containing e. The leaf is $v.r$ if $v.r = \text{NIL}$ and otherwise it is Succ$(v).\ell$. Initially the new node is colored red. If it is placed in an empty path, it becomes the root of the path. Otherwise, it is inserted into the top of the topmost lower chunk in the folded tree, i.e., the location of the lowest node on the unfolded path in the folded tree (see node x_1 on

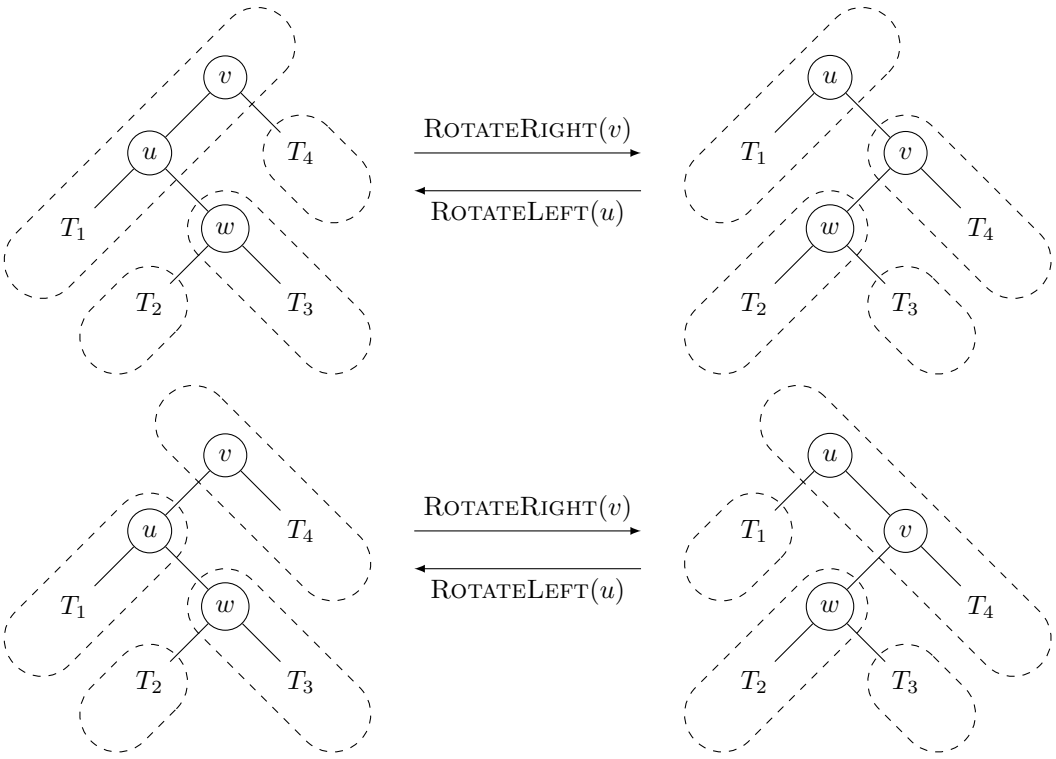


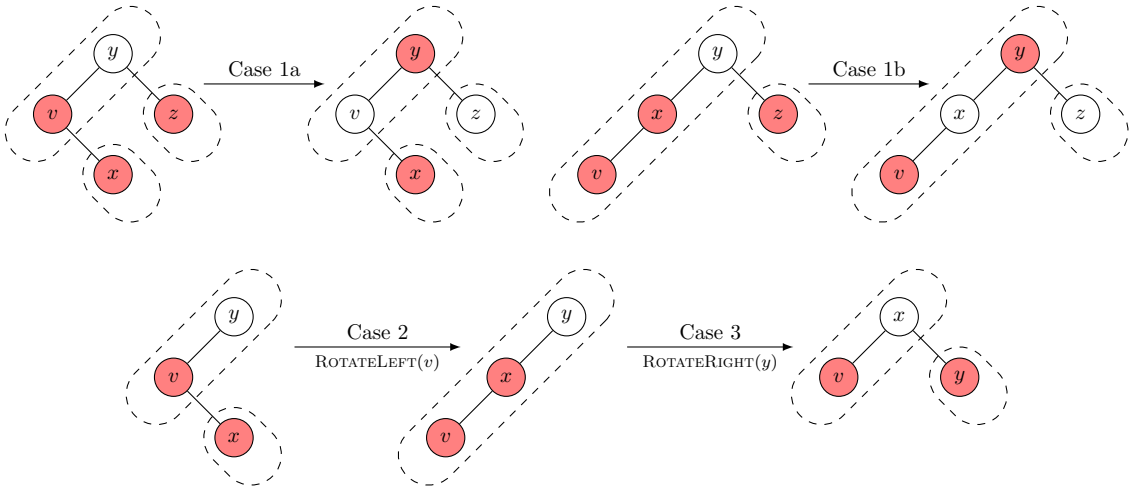
Figure 12: Rotations in an unfolded binary tree. The dashed areas show if the underlying path is left or right folded, with (top) the main path being left folded and (bottom) right folded.

Figure 9). The folded tree may during rebalancing contain a lower chunk with three nodes, with one black node at the bottom and two red nodes above it. It is an invariant during the rebalancing that each non-last lower chunk, contains exactly one black node and the red nodes below it from the unfolded tree. Since during insertions, at most one location in the unfolded tree contains two consecutive red nodes, all non-last lower chunks contain 1–3 nodes with exactly one black node, with at most one chunk being of size three. Further, in each non-last lower chunk the black node is at the bottom. Rebalancing proceeds by first performing a phase of recoloring nodes, and then terminating either at the root of the tree or by performing at most two rotations with associated recoloring of the rotated nodes. In Figure 6 these cases are shown for the unfolded tree, with Case 1a and 1b being the recoloring cases, and Case 2 and 3 being the terminating cases, which perform rotations. For the possible orientations of the paths, see Figure 13.

During the recoloring phase, three nodes are recolored in each iteration, and the rebalancing continues two nodes higher up in the unfolded tree. If an upper node is recolored, the tree remains correctly folded, as upper nodes may be of any color. If a lower node in a non-last lower chunk is recolored red, and it is the bottom node in the chunk in the folded tree, it is moved to the top of the lower chunk below in the folded tree, to ensure that the bottom node in the non-last lower chunk is black, which according to the recoloring cases it must, as the now bottom node has been recolored black. If this moves the node to the last lower chunk, it is further checked if the last lower chunk can split to introduce a new extra lower chunk and upper node. When a recoloring appears in the last lower chunk, this may similarly allow for splitting the last lower chunk. Both checks on the last lower chunk can be done similarly to how the folded tree was initially constructed. Note that during the recoloring, it is not possible that two nodes in the same lower chunk become black, and each lower chunk always contains exactly one black node, which therefore lead to only checks on nodes colored red in lower chunks being needed. Each of these operations correspond to a single recoloring step in the unfolded tree, and as each operation can be executed in constant time, the total time for the recoloring phase on the folded tree is asymptotically equal to the time for recoloring the unfolded tree.

If the recoloring phase does not terminate at the root of the tree, then at most two rotations are performed,

Left folded



Right folded

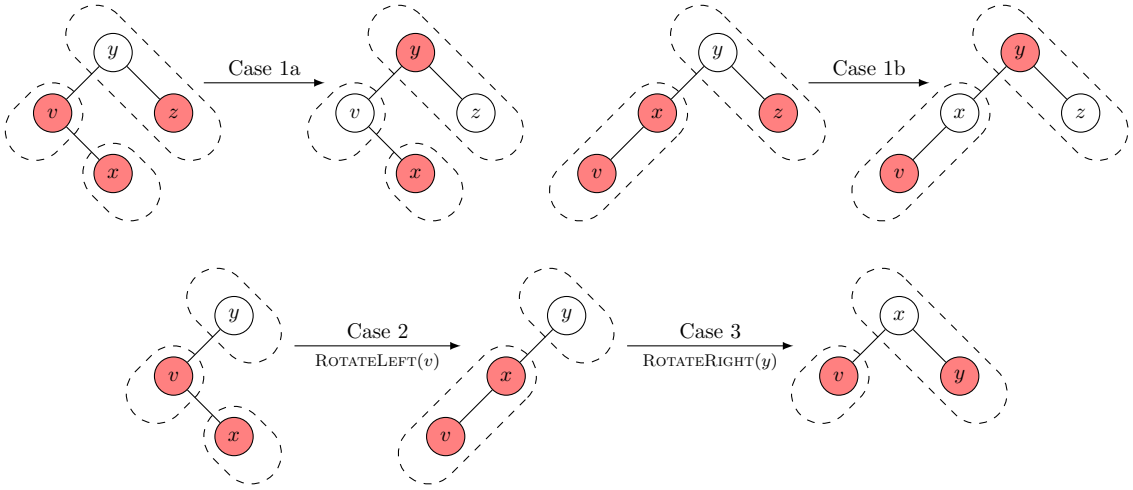


Figure 13: Insertion rebalancing cases on unfolded trees, when the two adjacent red nodes are on the left, with the underlying possible folding of the paths shown with dashed areas. Omitted subtrees have a black root or are empty. Cases 1a and 1b perform recoloring and continue from node y, while Cases 2 and 3 perform rotations in addition to recoloring, before terminating.

along with constant recoloring. These rotations are performed on the unfolded tree, and are therefore to be performed as in described in Section 4.3 on the folded tree. The rotations take time proportional to the height of the rotated node in the unfolded tree, assuming that the number of lower chunks in the main path remains the same if the rotated node is a lower node, without counting updates to the number of lower chunks from splitting or joining the last lower chunk. In each case of rotations performed in Case 2 and 3, the assumption that the number of lower chunks is unaltered is satisfied, utilizing in Case 3 that nodes are recolored, and therefore the rotations take time proportional to the height of the rotated node. As the height increases by a constant for each recoloring iteration, then the total time does not asymptotically exceed the time spent on the recoloring phase.

It therefore holds that the folded tree can be maintained under the rebalancing following an insertion, with the same asymptotic running time as the unfolded tree. The InsertSucc operation on an unfolded tree takes amortized constant time for the rebalancing plus the time for a Succ operation (Section 2). The Succ operation on folded trees take $\mathcal{O}(1)$ time, resulting in the InsertSucc and symmetrically InsertPred operations running in amortized constant time on the folded tree.

4.5 Deletions in folded red-black trees In this section we show how to perform the Delete operation on a folded tree in amortized constant time. Similar to insertions, the procedure uses the same rebalancing rules as unfolded trees, and it must be shown that the time spent on maintaining the folded tree to remove the desired node is proportional to the time spent on rebalancing the unfolded tree.

When performing a deletion on an unfolded tree, if the node to be removed has two children, it is swapped with its successor, including swapping their colors, and then remove the node. This guarantees that the deleted node has at most one child (if it has a child, it must be a single red right child), and let the rebalancing operation proceed from this node. In the folded tree, the successor can be found in $\mathcal{O}(1)$ time (Section 4.2). If the removed node is red, then the unfolded tree remains a valid red-black tree. In the folded tree, this node can be removed in $\mathcal{O}(1)$ time, as it must either be a red lower node, which does not decrease the number of lower chunks, or it is an upper node, on a path consisting of only that node. Otherwise, the removed node is black and the number of black nodes on the path in the unfolded tree decreases, which breaks the black height invariant. A rebalancing procedure proceeds from the child replacing the removed node (possibly a nil pointer).

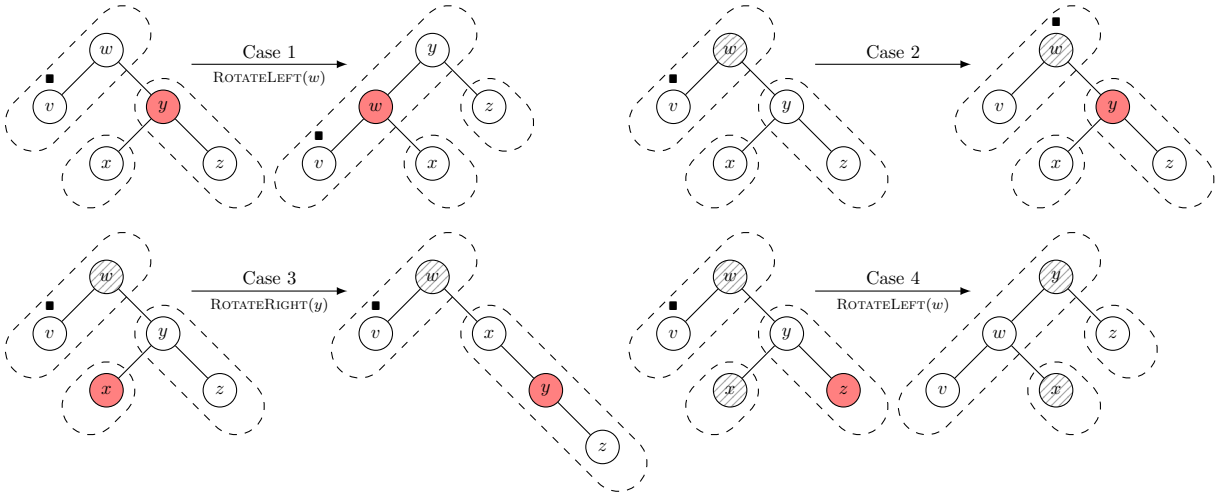
In the folded tree, if the double black node is nil, it remains a leaf, and therefore does not give rise to any complications. However, in the folded tree this location may exist a non-leaf, and therefore the situation must be handled with care, e.g., by introducing a temporary black dummy node.

In the folded tree, if the removed black node is a lower node, then this removes a lower chunk. Therefore the node marked double black must count as the black node in an additional lower chunk. To handle this, during the rebalancing, if the node marked double black is a lower node, and there exists a lower chunk below it in the unfolded tree, then this non-last lower chunk is *broken*, meaning that it contains no nodes, or only a single red node. During rebalancing, this broken lower chunk must be fixed, when progressing up the unfolded tree, such that the node marked double black still counts as the needed black node in the neighboring lower chunk.

The cases of the rebalancing on unfolded trees can be seen in Figure 7, where the node the rebalancing is executed from is node v which is marked double black. For the possible orientations of the paths, see Figure 14. The procedure on unfolded trees runs in amortized $\mathcal{O}(1)$ time (Section 2). Only in Case 2 the rebalancing proceeds upwards by a single node in the unfolded tree. In all other cases, a constant number of rotations and recoloring is performed, before terminating. At the point of the rotations, the time spent on rebalancing is equal to the height of the double black node and therefore also asymptotically equal to the height of the rotations. Note that in all cases, the node marked double black can be assumed to be black, since if the marked node is red, it may freely be colored black, thereby fixing the black height invariant, and the procedure terminates. In the folded tree, there may be a broken lower chunk, which in this case of termination, by recoloring the red node marked double black into a black node, introduces a new black node without removing a lower chunk, and the broken chunk can be fixed in constant time.

In Case 2, the sibling y is colored red. If y is a lower node, then this breaks the lower chunk of y. However, in this case v must be the root of an unfolded path, and therefore it is an an upper node in the folded path, and there is therefore no broken lower chunk below v, and as the procedure proceeds from node w, then y is exactly the allowed broken lower chunk below w. If v is the last node in the last lower chunk, then this chunk must be fixed when moving to the upper node w on the same path. This can be fixed by merging the last lower chunk with the neighboring upper node and lower chunk in the unfolded tree to create a new last lower chunk, which

Left folded



Right folded

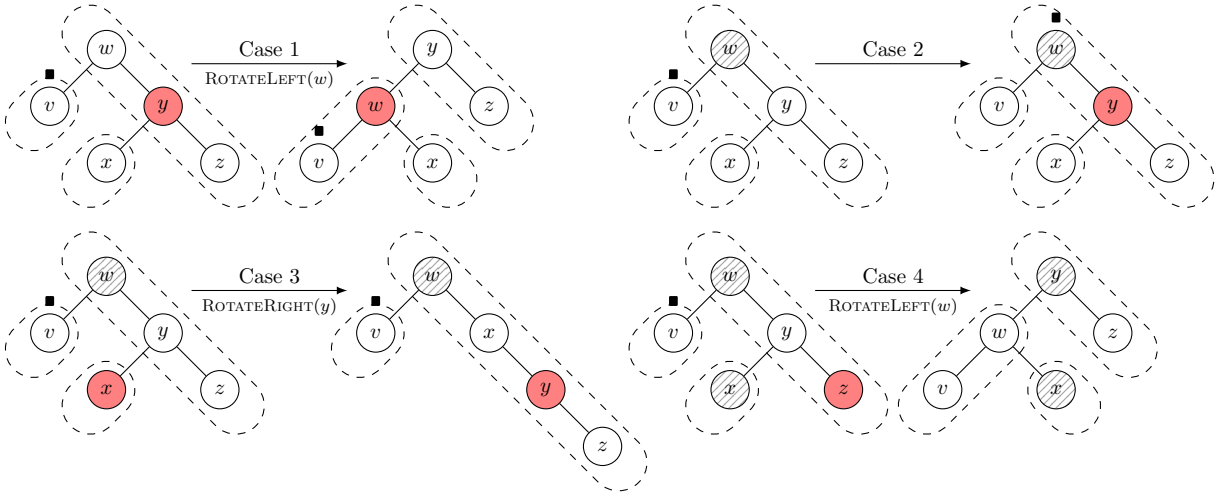


Figure 14: Deletion rebalancing cases on unfolded trees, when the double black node (marked ■) is on the left, with the underlying possible folding of the paths shown with dashed areas. The dashed nodes denote a node which color can be either red or black. Omitted subtrees can be empty or have a black or red root. The node v can possibly be an empty subtree. Case 2 performs recoloring and continues from node w, while Cases 1, 3, and 4 perform rotations in addition to recoloring. In Case 4, the color of node y after rebalancing is the color of node w before rebalancing.

then potentially splits. The broken lower chunk then ceases to exist. The total time for Case 2 on the folded tree is $\mathcal{O}(1)$.

In Case 1, a rotation is performed around w. According to Section 4.3, this rotation takes time proportional to the height of node w, if w is either an upper node, or if the number of lower chunks in the path of w is unaltered. If the path around w is right folded, then removing a red lower node does not alter the number of lower chunks. If the path is left folded, then a red node is added to the path, which does not alter the number of lower chunks. Both cases use that nodes are recolored alongside the rotation. The potentially broken lower chunk below v remains. The rebalancing then continues to Case 2, 3, or 4. If it continues in Case 2, then the procedure terminates afterwards, as w then is red.

In Case 3, a rotation is performed around y, and by a similar argument to above, this rotation runs in time proportional to the height of y, and the potentially broken lower chunk below v is preserved, with no new broken lower chunks being created. This runs in time proportional to the height of y. The rebalancing then continues to Case 4.

In Case 4, a rotation is performed around w. If the path of w is right folded, then the number of lower chunks is preserved. If the path is left folded and w is a lower node, then a new lower chunk consisting of a single black node is created. However, there must then be a broken lower chunk below v, which then can be filled, to allow the number of lower chunks to be preserved. It is used that nodes are recolored. The time is therefore proportional to the height of node w. The rebalancing then terminates.

In Case 2, the time for the rebalancing on the folded tree runs in $\mathcal{O}(1)$ time, which does not increase the asymptotic time, from the unfolded tree. In Case 1, 3, and 4, the time for the rebalancing on the folded tree runs in time proportional to the height of the rotation. Each of these operations can only be run once. As the height of the rotations increases by a constant for each iteration of Case 2, the folded tree can be maintained under rebalancing with the same asymptotic running time as the unfolded tree. The DELETE operation on an unfolded tree takes amortized $\mathcal{O}(1)$ time for the rebalancing plus the time for a SUCC operation (Section 2). The SUCC operation on folded trees take $\mathcal{O}(1)$ time, resulting in the DELETE operation running in amortized constant time on the folded tree.

5 Metanodes

In this section we describe how a BST T_{bits}, where each node stores a constant number of m bits, can be represented by a pure BST T with no information at the nodes and without increasing the running times by more than a constant factor. Each node in T_{bits} is represented by $\mathcal{O}(1)$ nodes in T. The basic idea is to partition a sorted sequence of n values into groups of $4 + 2m$ to $7 + 4m$ consecutive values, where each group is stored as a small pure BST, denoted a *metanode*. The metanodes become the binary nodes of a T_{bits} tree, where the tree structure of a metanode encodes m bits. For the special case where the tree contains less than $4 + 2m$ nodes, we have no metanode and the tree is just a single left path.

A metanode consists of $3 + 2m + s$ pure binary nodes, $1 \leq s \leq 4 + 2m$, each node storing a value, and in left-to-right inorder are the nodes

$$x, y, w_1, \ldots, w_s, b_1^1, b_1^2, \ldots, b_m^1, b_m^2, z\ .$$

The root of the metanode is y with left child x and right child z storing the smallest and largest value in the metanode, respectively. If all m bits $b_1, \ldots, b_m$ are equal to zero, the remaining nodes are stored as a left path in the left subtree of z. The nodes $w_1, \ldots, w_s$ are buffer nodes, and b_i^1 and b_i^2 encode bit b_i. If bit b_i is set to 1, node b_i^2 is rotated right to become the right child of b_i^1. Flipping bit b_i from 0 to 1 is done by ROTATERIGHT(b_i^2), and reversely flipping bit b_i from 1 to 0 is done by ROTATELEFT(b_i^1). Finally the left child of x is either NIL or the root L of another metanode (storing values smaller than or equal to the value at x), and similarly the right child R of z is possibly a metanode (with values larger than or equal to the value at z). The "value" of a metanode is the *value interval* between the values in x and z. Figure 15 shows a metanode encoding two bits.

The resulting pure BST T will be composed of multiple metanodes, and the search tree operations should be performed on the tree T. It is crucial that given a node v, to determine if v is the root of a metanode, since then we can decode the bits and access the left and right metanode children L and R as $v.\ell.\ell$ and $v.r.r$, respectively. The following lemma states that a metanote root can be uniquely determined by computing (5.1).

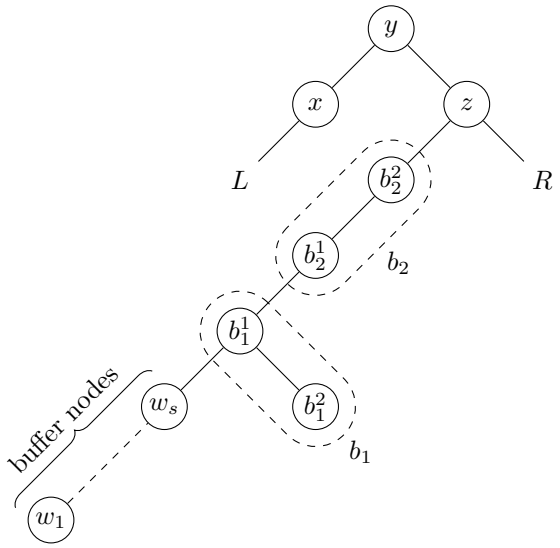


Figure 15: A metanode with root y, encoding two bits $b_1 = 1$ and $b_2 = 0$, and L and R are the roots of two metanodes (possibly NIL).

LEMMA 5.1. *A node v is the root of a metanode if and only if*

$$v.\ell \neq \text{NIL} \wedge v.r \neq \text{NIL} \wedge v.r.\ell \neq \text{NIL} \wedge (v.\ell.\ell = \text{NIL} \vee (v.\ell.\ell.r \neq \text{NIL} \wedge v.\ell.\ell.r.\ell \neq \text{NIL})) \tag{5.1}$$

Proof. If v is the root y of a metanode, it by definition has two children $v.\ell = x$ and $v.r = z$, and its right child $v.r$ has a left child $v.r.\ell$ that is either b_m^1 or b_m^2. Either the metanode with root v has no left child metanode, i.e., $v.\ell.\ell = L = \text{NIL}$, or the left child metanode exists and $v.\ell.\ell.r.\ell$ exists and is a node in the encoding of the b_m bit of the metanode with root L. This completes the "⇒" part of the proof. To prove "⇐", observe that the only nodes in a metanode with both a left and right child are the root y and possibly z and all b_i^1, but only y and z can have a right child with a left child, i.e., only y and z can satisfy $v.\ell \neq \text{NIL} \wedge v.r \neq \text{NIL} \wedge v.r.\ell \neq \text{NIL}$. Finally, if $v.\ell.\ell = \text{NIL}$, then $v \neq z$, since $z.\ell.\ell$ is a b_i^j or w_s ($z.\ell.\ell = w_s$ only if $m = 1$ and bit $b_1 = 1$), i.e., $v = y$. Otherwise, $v.\ell.\ell \neq \text{NIL}$, and by (5.1) we must have $v.\ell.\ell.r \neq \text{NIL}$ and $v.\ell.\ell.r.\ell \neq \text{NIL}$. If $z.\ell.\ell.r \neq \text{NIL}$ then $z.\ell.\ell = b_{m-1}^1$, $z.\ell.\ell.r = b_{m-1}^2$, and $z.\ell.\ell.r.\ell = \text{NIL}$. We have $v \neq z$, and the only possible value for v is the metanode root y. □

Note that all nodes with a right child (y and possibly z and all b_i^1) also have a left child. This fact is used in Section 6 to represent each node in a pure BST using two pointers only.

Given a T_{bits} binary finger search tree, a pure binary finger search tree is created as follows. The sorted list of n values is partitioned left-to-right in a sequence of metanodes. The metanodes become the nodes of a T_{bits} tree. To perform FINGERSEARCH(v, e), where v is a node in a metanode, we find the root of the metanode by repeatedly moving up along the path until (5.1) is true or we reach the root of the tree. To verify that the tree actually contains sufficiently many nodes to encode the first metanode, we check if the tree contains at least $4 + 2m$ nodes, by traversing the tree until $4 + 2m$ nodes have been identified. If the tree is smaller than a single metanode, the answer can be computed by comparing e with the values at all nodes. Otherwise, the search is performed on T_{bits} using $v.\ell.\ell$, $v.r.r$ and $v.p.p$ to navigate to the left child, right child and parent, respectively. To compare e with a metanode, we compare e with the value e_x and e_z at the nodes x and z of the metanode. If $e_x \leq e < e_z$, the answer to the search is inside the metanode and we return the result of a top-down search in the metanode. Otherwise, $e < e_x$ or $e_z \leq e$ and we answer that e is less than or greater than or equal to the value of the metanode, respectively, and we continue the search in T_{bits}.

To perform a finger update at a node v in a metanode, we find the root of the metanode and identify left-to-right the remaining nodes of the metanode in sorted order. Insertions create a new node next to v in the list, whereas deletions remove v. If the resulting list contains between $4 + 2m$ and $7 + 4m$ nodes, the nodes are rearranged to represent a metanode representing the same bits, possibly updating pointers to changed x, y and z,

i.e., the T_{bits} tree is unchanged. If the update is an insertion that causes the metanode to overflow, i.e., to contain $8 + 4m$ nodes, the mentanode is split into two metanodes with $4 + 2m$ values each, and we perform InsertSucc on T_{bits} to insert the new metanode. If the update is a deletion that causes the metanode to underflow, i.e., contain $\leq 3 + 2m$ nodes, we try to transfer a node from the successor or predecessor metanode, provided one of them exists and would not underflow. If the successor or predecessor metanode would underflow (i.e., before the deletion stores $4 + 2m$ nodes), we remove the successor or predecessor metanode from T_{bits} by performing Delete, and fusion the two metanodes to one metanode with $4m + 7$ nodes. Overall a finger search or update takes worst-case $\mathcal{O}(1)$ time plus the time for searching or updating T_{bits} with a constant factor overhead for decoding metanodes.

6 Representation of pure binary search trees

The canonical representation of a pure BST is to store each node as a record containing a value and three pointers ℓ, r and p to the left child, right child and parent of the node, respectively. See Figure 16 (left). The pure BSTs resulting from the metanode construction in Section 5 satisfy that *all nodes with at least one child have a left child*. Therefore, the nodes can be represented only using two pointers per node: a pointer ℓ to the left child and a pointer r to either the right sibling, if it exists, or the parent of the node. For the root, the r pointer is NIL. See Figure 16 (right).

Figure 16: Representation of a node v in a pure BST with parent c and two children a and b and no right sibling: (left) left-right-parent representation using three pointers; (right) left-right sibling/parent representation using two pointers.

In the two pointer representation, the right child and parent of a node v can be determined as follows: The right child of v is $v.\ell.r$, if $v.\ell \neq \text{NIL}$ and $v.\ell.r \neq v$. Otherwise, v has no right child. If $v.r = \text{NIL}$, then v is the root and has no parent (and no sibling). The parent of v is $v.r.r$, if $v.r.r \neq \text{NIL}$ and $v.r.r.\ell = v$ (v is the left sibling of $v.r$). Otherwise, $v.r$ is the parent of v.

The leftmost child plus right sibling representation is traditionally used to represent multiway trees—essentially storing the children of a node as a singly linked list. Iverson [28] denoted this representation a filial-heir chain; Sussenguth [42] denoted it a doubly-chained tree; Knuth [31, page 477] denoted it the binary tree representation; and Fredman et al. [22] denote it the child, sibling representation. Fredman et al. [22] in their paper on pairing heaps discuss a representation of binary trees only using two pointers, storing a pointer to the leftmost child and a combined right sibling/parent pointer. They store a bit per node to distinguish if a node with only one child has a left or right child. We avoid this by our invariant that all nodes with at least one child have a left child. Note that if the tree is a single left path then the representation is exactly a doubly linked list.

7 Pure binary finger search trees

We now combine the results of Sections 2–6 to achieve our main result (Theorem 1.1): A pure BST supporting finger searches in $\mathcal{O}(\lg d)$ time and finger updates in amortized constant time. The tree can be represented by n records, each storing a value and two pointers.

The pure BST is simply a folded-red black tree (Section 4) represented using metanodes (Section 5). This is possible since the nodes in a folded red-black tree only store left child, right child, and parent pointers and two bits (color and orientation). It can be represented by records storing a value and two pointers (Section 6), since each node in a metanode cannot have a right child without a left child.

Since a folded red-black tree allows navigation like in a red-black tree plus predecessor and successor queries in worst-case constant time (Section 4), they allow finger searches in $\mathcal{O}(\lg d)$ time (Section 3), and the metanode encoding (Section 5) only causes a constant factor overhead. By the same argument (combining Sections 2, 4, 5),

finger updates will take amortized constant time, since red-black trees support amortized constant time finger updates, provided Succ can be supported in constant time (Section 2).

In the preceding sections we did not discuss the temporary working space during the operations. But searches and updates in a red-black tree only requires constant temporary space to remember the current state during a search and rebalancing. Since our pure binary trees support both navigation to the parent and children of a node, we can with constant working space move between adjacent nodes in the pure binary tree, and therefore also in the folded and unfolded trees represented by the pure binary tree, using additional constant space to decode the local cases in the encoded metanodes and folded trees. In total $\mathcal{O}(1)$ temporary working space is sufficient for the operations listed in Section 1.1.

8 Worst-case lower bound for pure binary search trees

The finger update times achieved in Section 7 are inherently *amortized* constant. In this section we argue that this is a crucial requirement for *pure* BSTs, i.e., BSTs without any additional pointers than pointers to the parent and children of a node (the following arguments even apply if nodes can store any additional balance information). We show that *worst-case* constant finger updates in a pure BST can result in trees with asymptotic linear height, causing searches to take worst-case linear time.

Lemma 8.1. *Assume that an* InsertPred *or* InsertSucc *operation can at most access (and possibly modify) a connected component of c nodes in a pure BST containing the node pointed to by the finger. Then there exists a sequence of n insertions causing the the tree to have height at least $\lfloor n/2^c \rfloor$.*

Proof. We first make an observation, that we will use repeatedly: Consider a binary tree with at least 2^c nodes. In such a tree there must exist at least one node v with depth $c+1$ (the root having depth one), since the first c levels of a binary tree can at most contain $\sum_{i=0}^{c-1} 2^i = 2^c - 1$ nodes. By performing InsertPred or InsertSucc at v, the changes caused by the insertion cannot reach the root of the tree, and the root remains unchanged.

After the first 2^c insertions, we consider the root v_1 fixed. We then repeatedly insert 2^c additional nodes as predecessors or successors to a deepest node (in the subtree rooted at v_1). None of these updates can reach v_1. The subtree rooted at v_1 now contains 2^{c+1} nodes. The left or right subtree of v_1 must contain at least 2^c nodes (otherwise the subtree rooted at v_1 would contain at most $1 + 2 \cdot (2^c - 1) = 2^{c+1} - 1$ nodes). We let v_2 be a child of v_1 with a largest subtree. We now repeat inserting 2^c nodes in the subtree rooted at v_2, and let v_3 be a child of v_2 with largest resulting subtree, etc. After $k \cdot 2^c$ insertions, we will have constructed a path $v_1, v_2, \ldots, v_k$ starting at the root v_1, i.e., the tree has height at least k. It follows that the tree after n insertions has height at least $\lfloor n/2^c \rfloor$. ∎

From Lemma 8.1 we have the following two corollaries.

Corollary 8.1. *If* InsertPred *or* InsertSucc *takes worst-case $\mathcal{O}(c)$ time in a pure BST, then searches take worst-case $n/2^{\mathcal{O}(c)}$ time.*

Corollary 8.2. *Pure BSTs with height $\mathcal{O}(\lg n)$ can only be achieved if* InsertSucc *and* InsertPred *take worst-case $\Omega(\lg n)$ time.*

Proof. If the height of a pure BST is at most $\alpha \lg n$ after n InsertPred insertions, for some constant α, and the maximum number of nodes accessed by InsertPred is c, then by Lemma 8.1 the height can be at least $\lfloor n/2^c \rfloor$, i.e., we have the constraint $\lfloor n/2^c \rfloor \leq \alpha \lg n$, implying $c \geq \lg \frac{n}{1+\alpha \lg n} = \Omega(\lg n)$. Similarly for InsertSucc. ∎

Interestingly, in the comparison model, if arbitrary insertions and deletions (not given by a finger) are allowed to make $\mathcal{O}(c)$ comparisons, queries are known to require $n/2^{\mathcal{O}(c)}$ comparisons in both the worst-case and amortized sense [9, 11, 29]. For finger updates, like studied in this paper, total order is provided for free to the algorithm. Only the $\Omega(\lg n)$ lower bound for binary searches applies together with the lower bound of Bentley and Yao for finger searches [8].

It is important to note that the above lower bounds do not rule out general pointer based finger search structures with worst-case constant time finger updates. Such a data structure was described by Brodal et al. [12]. It remains an open problem if a pointer based data structure exists, supporting worst-case $\mathcal{O}(\lg d)$ time finger searches and $\mathcal{O}(1)$ time finger updates, where each node only stores a value and two pointers. In the next section we describe a general transformation that potentially could lead to this result. In particular, it is an open problem if the transformation can be applied to the construction in [12].

9 Linked list metanodes

In the previous sections the underlying assumption is that the final finger search data structure should be a pure BST, where each node can be represented using two or three pointers. In this section we briefly deviate from the pure binary tree assumption and present alternative data structures, where each record only stores a single value and two pointers. The main goal is to show that the lower bounds on the worst-case update time in Section 8 do not apply under this relaxation.

Assume the goal is to store a sorted list of n values. The basic idea is to maintain the values in a sorted singly linked list, where each node stores a value and has a *next* pointer to the next node in the list, where the *next* pointer of the last node is NIL. Finally, each node has a secondary pointer *info*. The linked list is partitioned into *metanodes* consisting of $\Theta(1)$ consecutive nodes. Each metanode can represent a record in a pointer based structure storing t pointers to other records and m bits. The nodes of a metanode left-to-right are: a head marker, with *info* pointing to the node itself; a node for each of the t pointers, with *info* pointing to the head of the corresponding metanode; a node for each of the m bits, with *info* being NIL if the bit is zero, or *info* equal to *next* otherwise, i.e., non-NIL; s buffer nodes, where $0 \leq s \leq 1+t+m$, all with *info* equal to NIL; and a terminating tail node with *info* pointing to the head of the metanode. The "value" of a metanode is the *value interval* between the values in head and tail. Figure 17 shows a metanode representing a red node in a red-black tree with parent, left child, right child pointers and links to the predecessor and successor metanodes, and a single color bit set to one. Note that the successor pointer in this representation points to the node following tail, and therefore could be omitted by only introducing a constant overhead to find the successor of a metanode.

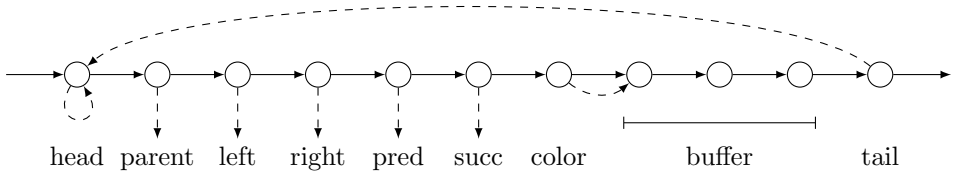


Figure 17: A linked-list metanode representing a node in a red-black tree with predecessor and successor links. Horizontal arrows are *next* pointers, and dashed arrows are *info* pointers.

Head nodes are uniquely determined by *info* pointing to the node itself. Tail nodes are uniquely determined by *next* being NIL or pointing to the head node of the next metanode in the linked list (with *info* pointing to itself). Given a finger to a node v in a metanode, we can find the head of the metanode by repeatedly following *next* pointers until we reach a tail node, where *info* points to the head.

We give two applications of the above metanode encodings. The first application is a simplified dynamic finger structure achieving time bounds matching those in Section 7. Consider a red-black tree with explicit successor and predecessor pointers. Such a tree supports amortized constant time finger updates (Section 2) and worst-case $\mathcal{O}(\lg d)$ finger searches (Section 3). Since the resulting data structure consists of records storing a value, five pointers (parent, left child, right child, predecessor and successor), and one bit (node color) we can apply the above metanode construction and achieve a finger search data structure where each node only stores a value and two pointers. To support FINGERSEARCH(v, e), where v is a node inside a metanode, we first find the head of the metanode containing v in $\mathcal{O}(1)$ time. We then perform the search in the red-black tree of metanodes as described in (Section 3), with the following twist: When comparing e with a metanode we compare e with the values e_{head} and e_{tail} at head and tail. If $e_{\text{head}} \leq e < e_{\text{tail}}$, the answer to the finger search is inside the metanode and the query can be finalized by a scan through the metanode in $\mathcal{O}(1)$ time. Otherwise, $e < e_{\text{head}}$ or $e_{\text{tail}} \leq e$ and the search proceeds left or right in the red-black tree, respectively. To perform a finger insertion INSERTPRED(v, e) or INSERTSUCC(v, e) for a node v in a metanode, we add a new node v' storing the value e to the singly linked list before or after v, respectively, such that v' becomes a new node in the metanode containing v. The information in the *info* pointers of the metanode is updated to reflect the same metanode information (pointers to other metanodes and bits). Note that to insert e to the left of head we need to update the *next* pointer in a node in the predecessor metanode that can be accessed using the predecessor pointer. Also if head changes, then all pointers to this metanode should be updated, but since these are the parent and the children of the metanode in the red-black tree, this can be done in $\mathcal{O}(1)$ time. Finally, if the buffer overflows, i.e., contains $2+t+m$ buffer nodes,

the metanode is split to create a new metanode containing the last $2+t+m$ nodes, by updating the *info* pointers, where each of the two metanodes have empty buffers. The new metanode is then inserted as the successor of the old metanode in the red-black tree in amortized $\mathcal{O}(1)$ time (Section 2). A DELETE(v) operation similarly deletes a node from a metanode, and requires similar updates. If a metanode underflows, i.e., has $1+t+m$ nodes, the metanode either gets a node from the successor or predecessor metanode or fusions with either the predecessor or successor metanode with an empty buffer (like a deletion in a B-tree [6]) such that the resulting metanode has a buffer of size $1+t+m$. This requires deleting a metanode from the red-black tree in amortized $\mathcal{O}(1)$ time (Section 2). It follows that finger updates can be performed in amortized constant time and finger searches in $\mathcal{O}(\lg d)$ time. The same result could also have been achieved by combining the metanode encoding idea with the level-linked (a,b)-trees of Huddleston and Mehlhorn [27].

Our second application is the existence of a search tree encoding supporting *worst-case* constant time insertions. Levcopolous and Overmars [33] presented a BST supporting finger updates in worst-case constant time and searches in worst-case $\mathcal{O}(\lg n)$ time. We sketch below how this structure can be adapted such that we can apply the metanodes of this section, and support INSERTSUCC in worst-case $\mathcal{O}(1)$ time and searches in $\mathcal{O}(\lg n)$ time, i.e., surparsing the lower bound for pure BSTs (Corollary 8.1). The structure in [33] is pointer based, but uses integers and records not storing values, two requirements falling outside the requirements for the metanode encoding. We next discuss how to circumvent these restrictions, such that we can apply the metanode encoding and achieve the same bounds for INSERTSUCC and searches using only two pointers per node. Essentially, [33] partitions the sorted list of values into *bags* of $\mathcal{O}(\lg^2 n)$ values, and stores the bags as the nodes of a balanced BST, say a red-black tree. During a sequence of insertions, a bag of maximum size is split into two bags and the new bag is inserted into the search tree incrementally over the next $\mathcal{O}(\lg n)$ operations. This ensures that all bags will have size $\mathcal{O}(\lg^2 n)$ [33, Lemma 3.3]. Bags are represented as lists-of-lists, such that all lists have length $\mathcal{O}(\lg n)$. Bags of equal size are again stored in a linked list, and all these lists-of-bags are maintained in a linked list by increasing bag size. Finally, to achieve worst-case bounds bag splitting is performed incrementally while inserting into the bags, and incremental global rebuilding [38] is applied to handle changing values of $\lg n$. By letting a record storing a value contain multiple pointers for each of the involved linked lists, pointers to head of the lists etc., we can manage with one record for each value. To be able to store bag sizes, and compare bag sizes for equality, we can let the record for each value represent an integer $1,\ldots,n$, and maintain these in increasing order. To store an integer we have a pointer to this list. Crucially, for this to work we disallow deletions, such that whenever a new value is inserted it can just be added as the next value $n+1$ to this list. We omit further details, since the main goal of this section is just to highlight that the lower bounds in Section 8 only apply to pure BSTs.

We leave it as an open problem if the finger search data structure by Brodal et al. [12] supporting worst-case constant time finger updates and $\mathcal{O}(\lg d)$ finger searches can be encoded using linked-list metanodes.

10 Conclusion and open problems

We have presented optimal pure binary finger search trees. Whereas each of the ideas is conceptually simple, the resulting data structure has a quite large constant factor overhead. It would be interesting to find simpler pure BSTs with a smaller overhead.

In this paper we have only considered pointer based data structures. One could also consider the problem in the more general RAM model, e.g., allowing bit operations on words and aiming at more succinct representations. Recently, Bender et al. [7] considered a randomized scheme to encode BSTs in this model using *tiny pointers*. They showed how to encode dynamic balanced BST using only $\mathcal{O}(n)$ bits for storing the tree structure, while each tree modification takes $\mathcal{O}(\lg^* n)$ time. An open problem is what the best possible space and time bounds are for finger search trees in this model.

For the extreme case, where no additional space is allowed, Munro and Suwanda [37] introduced the notion of *implicit data structures*, where only an array of the values is stored, without any additional information. All additional information must be encoded in the permutation of the values. Munro and Suwanda presented an implicit dynamic dictionary with $\mathcal{O}(n^{1/3}\lg n)$ time bounds. Frederickson [21] improved the update time bounds to $\mathcal{O}(n^{\sqrt{2/\lg n}}\lg^{3/2} n)$ with $\mathcal{O}(\lg n)$ time searches. Munro [36] achieved $\mathcal{O}(\lg^2 n)$ time bounds for queries and updates by encoding AVL-trees, Franceschini et al. [20] achieved $\mathcal{O}(\lg^2 n/\lg\lg n)$ time bounds for queries and updates by encoding implicit B-trees. Finally, Franceschini and Grossi [19] achieved searches and updates in worst-

case optimal $\mathcal{O}(\lg n)$ time. Dictionaries supporting finger searches in this very restricted model were considered by Brodal, Nielsen and Truelsen [13], who proved trade-offs between the time for finger queries and the time to move a single finger.

Generally, in the RAM model, it is an interesting open problem how efficient dynamic finger search structures can be obtained with limited space in addition to the stored values, e.g., are there trade-offs between update time, query time and space usage? In particular, what are the best possible bounds for *stable* dictionaries, i.e., values are never moved once inserted? Stability appears crucial to be able to support an arbitrary number of fingers.

Acknowledgments The authors wish to thank the anonymous reviewers for detailed feedback on the presentation.

References

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Arne Andersson. Improving partial rebuilding by using simple balance criteria. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989. doi:10.1007/3-540-51542-9_33.

[4] Arne Andersson, Rolf Fagerberg, and Kim S. Larsen. Balanced binary search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 10. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.CH10.

[5] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):13, 2007. doi:10.1145/1236457.1236460.

[6] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.

[7] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. Tiny pointers. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 477–508. SIAM, 2023. doi:10.1137/1.9781611977554.CH21.

[8] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976. doi:10.1016/0020-0190(76)90071-5.

[9] Allan Borodin, Leonidas J. Guibas, Nancy A. Lynch, and Andrew Chi-Chih Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12(2):71–75, 1981. doi:10.1016/0020-0190(81)90005-3.

[10] Gerth Stølting Brodal. Finger search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.CH11.

[11] Gerth Stølting Brodal, Shiva Chaudhuri, and Jaikumar Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic Journal of Computing*, 3(4):337–351, 1996.

[12] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios K. Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003. doi:10.1016/S0022-0000(03)00013-8.

[13] Gerth Stølting Brodal, Jesper Sindahl Nielsen, and Jakob Truelsen. Finger search in the implicit model. In Kun-Mao Chao, Tsan-sheng Hsu, and Der-Tsai Lee, editors, *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*, volume 7676 of *Lecture Notes in Computer Science*, pages 527–536. Springer, 2012. doi:10.1007/978-3-642-35261-4_55.

[14] Mark R. Brown. A storage scheme for height-balanced trees. *Information Processing Letters*, 7(5):231–232, 1978. doi:10.1016/0020-0190(78)90005-4.

[15] Mark R. Brown. Addendum to "a storage scheme for height-balanced trees". *Information Processing Letters*, 8(3):154–156, 1979. doi:10.1016/0020-0190(79)90009-7.

[16] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: the proof. *SIAM Journal of Computing*, 30(1):44–85, 2000. doi:10.1137/S009753979732699X.

[17] 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. *SIAM Journal of Computing*, 30(1):1–43, 2000. doi:10.1137/S0097539797326988.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022.

[19] Gianni Franceschini and Roberto Grossi. Optimal implicit dictionaries over unbounded universes. *Theory of Computing Systems*, 39(2):321–345, 2006. doi:10.1007/S00224-005-1167-9.

[20] Gianni Franceschini, Roberto Grossi, J. Ian Munro, and Linda Pagli. Implicit B-trees: a new data structure for the dictionary problem. *Journal of Computer and System Sciences*, 68(4):788–807, 2004. doi:10.1016/J.JCSS.2003.11.003.

[21] Greg N. Frederickson. Implicit data structures for the dictionary problem. *Journal of the ACM*, 30(1):80–94, 1983. doi:10.1145/322358.322364.

[22] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. doi:10.1007/BF01840439.

[23] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 165–174. ACM/SIAM, 1993. URL: https://dl.acm.org/doi/10.5555/313559.313676.

[24] Ofek Gila, Michael T. Goodrich, and Robert E. Tarjan. Zip-zip trees: Making zip trees more balanced, biased, compact, or persistent. In Pat Morin and Subhash Suri, editors, *Algorithms and Data Structures - 18th International Symposium, WADS 2023, Montreal, QC, Canada, July 31 - August 2, 2023, Proceedings*, volume 14079 of *Lecture Notes in Computer Science*, pages 474–492. Springer, 2023. doi:10.1007/978-3-031-38906-1_31.

[25] Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 49–60. ACM, 1977. doi:10.1145/800105.803395.

[26] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.

[27] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. doi:10.1007/BF00288968.

[28] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[29] Haim Kaplan, Or Zamir, and Uri Zwick. The amortized cost of finding the minimum. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 757–768. SIAM, 2015. doi:10.1137/1.9781611973730.51.

[30] Alexis C. Kaporis, Christos Makris, Spyros Sioutas, Athanasios K. Tsakalidis, Kostas Tsichlas, and Christos D. Zaroliagis. Improved bounds for finger search on a RAM. *Algorithmica*, 66(2):249–286, 2013. doi:10.1007/S00453-012-9636-4.

[31] Donald Ervin Knuth. *The art of computer programming, Volume III: Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998.

[32] S. Rao Kosaraju. Localized search in sorted lists. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 62–69. ACM, 1981. doi:10.1145/800076.802458.

[33] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26(3):269–277, 1988. doi:10.1007/BF00299635.

[34] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998. doi:10.1145/274787.274812.

[35] Kurt Mehlhorn and Athanasios K. Tsakalidis. An amortized analysis of insertions into AVL-trees. *SIAM Journal of Computing*, 15(1):22–33, 1986. doi:10.1137/0215002.

[36] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33(1):66–74, 1986. doi:10.1016/0022-0000(86)90043-7.

[37] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21(2):236–250, 1980. doi:10.1016/0022-0000(80)90037-9.

[38] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. doi:10.1007/BFB0014927.

[39] Raimund Seidel. Maintaining ideally distributed random search trees without extra space. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 134–142. Springer, 2009. doi:10.1007/978-3-642-03456-5_9.

[40] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. doi:10.1007/BF01940876.

[41] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.

[42] Edward H. Sussenguth, Jr. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963. doi:10.1145/366552.366600.

[43] Robert Endre Tarjan. Updating a balanced search tree in $O(1)$ rotations. *Inf. Process. Lett.*, 16(5):253–257, 1983. doi:10.1016/0020-0190(83)90099-6.

[44] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031.

[45] Robert Endre Tarjan and Christopher J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17(1):143–178, 1988. doi:10.1137/0217010.

[46] Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67(1-3):173–194, 1985. doi:10.1016/S0019-9958(85)80034-6.

[47] P. F. Windley. Trees, forests and rearranging. *The Computer Journal*, 3(2):84–88, 1960. doi:10.1093/COMJNL/3.2.84.