Layered Working-Set Trees

Prosenjit Bose[*]   Karim Douïeb[*]   Vida Dujmović[*]   John Howat[*]

Abstract

The *working-set bound* [Sleator and Tarjan, J. ACM, 1985] roughly states that searching for an element is fast if the element was accessed recently. Binary search trees, such as splay trees, can achieve this property in the amortized sense, while data structures that are not binary search trees are known to have this property in the worst case. We close this gap and present a binary search tree called a *layered working-set tree* that guarantees the working-set property in the worst case. The *unified bound* [Bădoiu et al., TCS, 2007] roughly states that searching for an element is fast if it is near (in terms of rank distance) to a recently accessed element. We show how layered working-set trees can be used to achieve the unified bound to within a small additive term in the amortized sense while maintaining in the worst case an access time that is both logarithmic and within a small multiplicative factor of the working-set bound.

1   Introduction

Let S be a set of keys from a totally ordered universe and let X be a sequence of elements from S. Typically, one is required to store elements of S in some data structure D such that accessing the elements of S using D in the order defined by X is "fast." Here, "fast" can be defined in many different ways, some focusing on worst case access times and others on amortized access times. For example, the search times of splay trees [8] can be stated in terms of the rank difference between the current and previous elements of X; this is the *dynamic finger property* [3, 4].

If x is the i-th element of X, we say that x is accessed at time i in X. The working-set number of x at time i, denoted $w_i(x)$, is the number of distinct elements accessed since the last time x was accessed or inserted, or $|D|$ if x is either not in D or has not been accessed by time i.

The *working-set property* states the time to access x at time i is $O(\lg w_i(x))$.[1]   Splay trees were shown by Sleator and Tarjan [8] to have the working-set property in the amortized sense. One drawback of splay trees, however, is that most of the access bounds hold only in an amortized sense. While the amortized cost of a query can be stated in terms of its rank difference between successive queries or the number of distinct queries since a query was last made, any particular operation could take $\Theta(n)$ time. In order to address this situation, attention has turned to finding data structures that maintain the distribution-sensitive properties of splay trees but guarantee good performance in the worst case.

The data structure of Bădoiu et al. [2], called the working-set structure, guarantees this property in the worst case. However, this data structure departs from the binary search tree model and is instead a collection of binary search trees and queues.

Bădoiu et al. [2] also describe a data structure called the unified structure that achieves the *unified property*, which states that searching for x at time i takes time $O(\min_{y \in S} \lg(w_i(y) + d(x, y)))$ where $d(x, y)$ is the rank difference between x and y. Again, this data structure is not a binary search tree. The skip-splay algorithm of Derryberry and Sleator [6] fits into the binary search tree model and comes within a small additive term of the unified bound in an amortized sense.

Our Results.   We present a binary search tree that is capable of searching for a query x in worst-case time $O(\lg w_i(x))$ and performs insertions and deletions in worst-case time $O(\lg n)$, where n is the number of keys

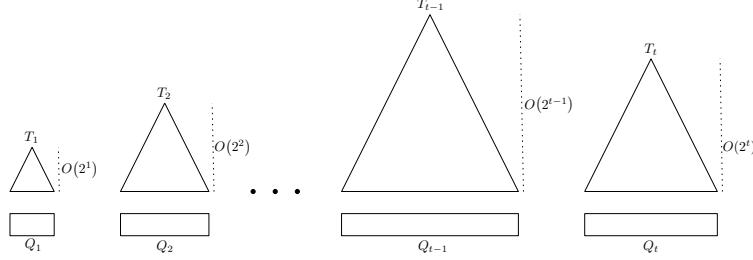[1]In this paper, $\lg x$ is defined to be $\log_2(x + 2)$.

Figure 1: The working-set structure of Bădoiu et al. [2]. The pointers between corresponding elements in T_j and Q_j are not shown.

stored by the tree at the time of the access. This fills in the gap between binary search trees that offer these query times in only an amortized sense and data structures which guarantee these query times in the worst-case but do not fit in the binary search tree model. We have also shown how to use this binary search tree to achieve the unified bound to within a small additive term in the amortized sense while maintaining in the worst case an access time that is both logarithmic and within a small multiplicative factor of the working-set bound.

Organization.   The rest of this paper is organized in the following way. We complete the introduction by summarizing the way the working-set structure of Bădoiu et al. [2] operates, since this will play a key role in our binary search tree. In Section 2, we describe our binary search tree and explain the way in which operations are performed. In Section 3, we show how to combine our results with those of Derryberry and Sleator [6] on the unified bound to achieve an improved worst-case search cost. We conclude with Section 4 which summarizes our results and explains possible directions for future research.

1.1   The Working-Set Structure

We now describe the working-set structure of Bădoiu et al. [2]. The structure maintains a dynamic set under the operations INSERT, DELETE and SEARCH. Denote by $S_i \subseteq S$ the set of keys stored in the data structure at time i.

The structure is composed of $t = O(\lg \lg |S_i|)$ balanced binary search trees $T_1, T_2, \ldots, T_t$ and the same number of doubly linked lists $Q_1, Q_2, \ldots, Q_t$. For any $1 \le j \le t$, the contents of T_j and Q_j are identical, and pointers (in both directions) are maintained between their common elements. Every element in the set S_i is contained in exactly one tree and in its corresponding list. For $j < t$, the size of T_j and Q_j is 2^{2^j}, whereas the size of T_t and Q_t is $|S_i| - \sum_{j=1}^{t-1} 2^{2^j} \le 2^{2^t}$. Figure 1 shows a schematic of the structure.

The working-set structure achieves its stated query time of $O(\lg w_i(x))$ by ensuring that an element x with working-set number $w_i(x)$ is stored in a tree T_j with $j \le \lceil \lg \lg w_i(x) \rceil$. Every list Q_j orders the elements of T_j by the time of their last access, starting with the youngest (most recently accessed) and ending with the oldest (least recently accessed).

Operations in the working-set structure are facilitated by an operation called a *shift*. A shift is performed between two trees T_j and T_k. Assume $j < k$, since the other case is symmetric. To perform a shift, we begin at T_j. We look in Q_j to determine the oldest element and remove it from Q_j and delete it from T_j. We then insert it into T_{j+1} and Q_{j+1} (as the youngest element) and repeat the process by shifting from $j + 1$ to k. This process continues until we attempt to shift from one tree to itself. Observe that a shift causes the size of T_j to decrease by one and the size of T_k to increase by one. All of the trees between T_j and T_k will end up with the same size, but the elements contained in them change, since the oldest element from the previous tree is always added as the youngest element of the next tree.

We are now ready to describe how to make queries in the working-set structure. To search for an element x, we search sequentially in $T_1, T_2, \ldots$ until we find x or search all of the trees and fail to find x. If $x \notin T_j$ for any j, then we will search every tree at a total cost of $O(\lg |S_i|)$ and then report that x is not in the structure. Otherwise, assume $x \in T_j$. We delete x from T_j and Q_j and insert it in T_1 and place it at the front of Q_1.

2

We now have that the size of T_1 and Q_1 has increased by one and the size of T_j and Q_j has decreased by one. We therefore perform a shift from 1 to j to restore the sizes of the trees and lists. The time required for a search is dominated by the search time in T_j. Observe that if $x \in T_j$ and $j > 1$, then it must have been removed as the oldest element from Q_{j-1}, at which point at least $2^{2^{j-1}}$ distinct queries had been made. Therefore, $w_i(x) \geq 2^{2^{j-1}}$ and so the search time is $O\left(\lg 2^{2^j}\right) = O\left(\lg 2^{2^{j-1}}\right) = O(\lg w_i(x))$.

Insertions are performed by inserting the element into T_1 and Q_1 (as the youngest element). Again, this causes T_1 and Q_1 to be too large. Since no other tree has space for one more element, we must shift to the last tree T_t. Thus, a shift from 1 to t is performed at total cost $O(\lg |S_i|)$. Note that it is possible that a new tree may need to be created if the size of T_t grows past 2^{2^t}. Deletions are performed by first searching for the element to be deleted. Once found, say in T_j, it is removed from T_j and Q_j. To restore these sizes, we perform a shift from t to j at total cost $O(\lg |S_i|)$. If the last tree becomes empty, it can be removed.

2 The Binary Search Tree

In this section, we describe a binary search tree that has the working-set property in the worst case.

2.1 Model

Recall the binary search tree model of Wilber [10]. Each node of the tree stores the key associated with it and has a pointer to its left and right children and its parent. The keys stored in the tree are from a totally ordered universe and are stored such that at any node, all of the keys in the left subtree are less than that stored in the node and all of the keys in the right subtree are greater than that stored at the node. Furthermore, each node may keep a constant[2] amount of additional information called *fields*, but no additional pointers may be stored.

To perform an access to a key, we are given a pointer initialized to the root of the tree. An access consists of moving this pointer from a node to one of its adjacent nodes (through the parent pointer or one of the children pointers) until the pointer reaches the desired key. Along the way, we are allowed to update the fields and pointers in any nodes that the pointer reached. The access cost is the number of nodes reached by the pointer.

2.2 Tree Decomposition

Our binary search tree will adapt the working-set structure described in the previous section to the binary search tree model. Let T denote the binary search tree as a whole. At a high level, our binary search tree layers the trees $T_1, T_2, \ldots, T_t$ of the working-set structure together to form T, and then augments nodes with enough information to recover which is the oldest in each tree at any given time.

Consider a labelling of T where each node $x \in T$ has a label from $\{1, 2, \ldots, t\}$ such that no node has an ancestor with a label greater than its own label. This labelling partitions the nodes of T. We say that the nodes with label $j \in \{1, 2, \ldots, t\}$ form a *layer* L_j. A layer L_j will play the same role as T_j in the working-set structure. Like T_j, L_j contains exactly 2^{2^j} elements for $j < t$, and L_t contains the remaining elements. Unlike T_j, L_j is typically a collection of subtrees of T. We refer to a subtree of a layer L_j as a *layer-subtree*. Figure 2 shows this decomposition. Every node $x \in T$ stores as a field the value j such that $x \in L_j$ which we denote by $\mathsf{layer}[x]$. We also record the total number of layers t and the size of L_t at the root as fields of each node.

Each layer-subtree $T'_j \in L_j$ is maintained independently as a tree that guarantees that each node of T'_j has depth in T'_j at most $O\left(\lg |T'_j|\right) = O(\lg |L_j|)$. This can be done using, *e.g.*, a red-black tree [1, 7]. By "independently", we mean that balance criteria are applied only to the elements within one layer-subtree.

Our first observation concerns the depth of a node in a given layer.

Lemma 1. *The depth of a node $x \in L_j$ is $O\left(2^j\right)$.*

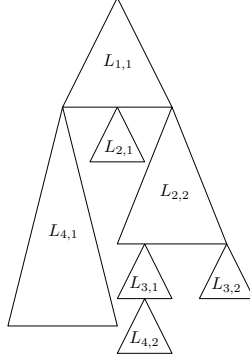[2]By standard convention, $O(\lg |S_i|)$ bits are considered to be "constant."

Figure 2: The decomposition of the tree T into layers. Here, the layer-subtrees of L_j are denoted $L_{j,1}, L_{j,2}, \ldots$. Observe that layer L_j can be connected to any layer L_k with $k > j$. In this case, all of the elements of the layer-subtree $L_{4,1}$ are less than the elements in $L_{1,1}$, and so the layer-subtree $L_{4,1}$ must be connected to a leaf of $L_{1,1}$.

Proof. In the worst case, we must traverse a layer-subtree of each of $L_1, L_2, \ldots, L_{j-1}$ to reach L_j and then locate x in L_j. Each layer L_k has size 2^{2^k} and thus each layer-subtree we pass through has size at most 2^{2^k}. Since each layer-subtree guarantees depth logarithmic in the size of the layer-subtree and thus the layer, the total depth is $\sum_{k=1}^{j} O(2^k) = O(2^j)$. □

The main obstacle in creating our tree comes from the fact that the core operations are performed on subtrees rather than trees, as is the case for the working-set structure. Consequently, standard red-black tree operations can not be used for the operations spanning more than one layer as described in Section 2.4. We break the operations into those restricted to one layer, those spanning two neighbouring layers, and finally those performed on the tree as a whole. These operations are described in the following sections.

Another difficulty arises from the having to implement the queues of the working-set structure in the binary search tree model. The queues are needed in order to determine the oldest element in a layer at any given time.

We encode the linked lists in our tree as follows. Each node $x \in L_j$ stores the key of the node inserted into L_j directly before and after it. This information is stored in the fields older[x] and younger[x], respectively. We also store a key value in the field nextlayer[x]. If x is the oldest element in layer L_j, then no element was inserted before it and so we set older[x] = nil. In this case, we use nextlayer[x] to store the key of the oldest element in layer L_{j+1}. Similarly, if x is the youngest element in layer L_j, then no element was inserted after it and so we set younger[x] = nil and use nextlayer[x] to store the key of the youngest element in layer L_{j+1}. If x is neither the youngest nor the oldest element in L_j, then we have nextlayer[x] = nil.

Before we describe how operations are performed on this binary search tree, we must make a brief note on storage. By the above description, each node x stores three pointers (parent and children) and a key, as per the usual binary search tree model. The root also maintains the number of trees t and the size of L_t. In addition, we must store balance information (one bit for red-black trees) and three additional key values (exactly one of which is nil): older[x], younger[x] and nextlayer[x]. If keys are assumed to be of size $O(\lg n)$, then it is clear our binary search tree fits the model of Section 2.1. Note that we are storing *key values*, not pointers. Given a key value stored at a node, we do not have a pointer to it, so we must instead search for it by traversing to the root and performing a standard search in a binary search tree. If keys have size $\omega(\lg n)$, it is true that we use more than $O(\lg n)$ additional space per node. However, since any node would then store a key of size $\omega(\lg n)$, we are only increasing the size of a node by a constant factor.

2.3 Intra-Layer Operations

The operations we perform within a single layer are essentially the same as those we perform on any balanced binary search tree. We need notions of restoring balance after insertions and deletions and of splitting

4

and joining. As mentioned before, we are not necessarily restricting ourselves to using any particular implementation of layer-subtrees. Instead, we will state the intra-layer operations and the required time bounds, and then show how red-black trees [1, 7] can be used to fulfill this role. Other binary search trees that meet the requirements of each operation could also be used. Layer-subtrees must also ensure that their operations do not leave the layer-subtree; this can be done by checking the layer number of a node before visiting it.

Intra-layer operations rearrange layer-subtrees in some way. Observe that layer-subtrees hanging off a given node are maintained even after rearranging the layer-subtree, since the roots of such layer-subtrees can be viewed as the results of unsuccessful searches. Therefore, when describing these operations, we need not concern ourselves with explicitly maintaining layer-subtrees below the current one.

In our binary tree T, for each node x in a layer-subtree T'_j of L_j, we define the following operations. They are straightforward, but mentioned here for completeness and as a basis for the operations performed between layers.

INSERT-FIXUP(x) This operation is responsible for ensuring that each node of T'_j has depth $O\big(\lg |T'_j|\big)$ after the node x has been inserted into the layer-subtree. For red-black trees, this operation is precisely the RB-INSERT-FIXUP operation presented by Cormen et al. [5, Section 13.3]. Although the version presented there does not handle colouring x, it is straightforward to modify it to do so.

DELETE-FIXUP(x) This operation is responsible for ensuring that each node of T'_j has depth $O\big(\lg |T'_j|\big)$ after a deletion in the layer-subtree. The exact node x given to the operation is implementation dependent. For red-black trees, this operation is precisely the RB-DELETE-FIXUP operation presented by Cormen et al. [5, Section 13.4]. In this case, the node x is the child of the node spliced out by the deletion algorithm; we will elaborate on this when describing the layer operations in Section 2.4.

SPLIT(x) This operation will cause the node $x \in T'_j$ to be moved to the root of T'_j. The rest of the layer-subtree will be split between the left and right side of x such that each side is independently balanced and thus guarantee depth $O\big(\lg |T'_j|\big)$ of their respective nodes; this may mean that the layer-subtree is no longer balanced as a whole. For red-black trees, this operation is described by Tarjan [9, Chapter 4], except we do not destroy the original trees, but rather stop when x is the root of the layer-subtree.

JOIN(x) This operation is the inverse of SPLIT(x): given a node $x \in T'_j$, we will restructure T'_j to consist of x at the root of the T'_j and the remaining elements in subtrees rooted at the children of x such that all nodes in the layer-subtree have depth $O\big(\lg |T'_j|\big)$. For red-black trees, this operation is described by Cormen et al. [5, Problem 13-2].

Lemma 2. *The operations* INSERT-FIXUP(x), DELETE-FIXUP(x), SPLIT(x) *and* JOIN(x) *on a node* $x \in L_j$ *can be implemented to take worst-case time* $O\big(2^j\big)$ *when red-black trees are used as layer-subtrees.*

Proof. Immediate from the operations given by Cormen et al. [5] and Tarjan [9].   □

2.4   Inter-Layer Operations

The operations performed on layers correspond to the queue and shift operations of the working-set structure. The four operations performed on layers are YOUNGESTINLAYER(L_j) and OLDESTINLAYER(L_j) for a layer L_j and MOVEUP(x) and MOVEDOWN(x) for a node x.

As we did with the intra-layer operations, we will describe the requirements of the operations independently of the actual layer-subtree implementation. In fact, only the operation MOVEDOWN(x) will require knowledge of the implementation of the layer-subtrees; the remaining operations simply make use of the operations defined in Section 2.3.

YOUNGESTINLAYER(L_j)   This operation returns the key of the youngest node in layer L_j. We first examine all elements in L_1 (of which there are $O(1)$). Once we find the element that is the youngest (by looking for the element for which younger[x] = nil), say x_1, we go back to the root and search for nextlayer[x_1], which will bring us to the youngest element in L_2, say x_2. We then go back to the root and search for nextlayer[x_2], and so on. This repeats until we find the youngest element in L_j, as desired. The process for OLDESTINLAYER(L_j) is the same, except our initial search in L_1 is for the oldest element, *i.e.*, the element for which older[x] = nil.

MOVEUP(x)   This operation will move x from its current layer L_j to the next higher layer L_{j-1}. To accomplish this, we first split x to the root of its layer-subtree using SPLIT(x). We remove x from L_j by setting layer[x] = $j-1$. We now must restore balance properties. Observe that, by the definition of split, both of the layer-subtrees rooted at the children of x are balanced. Therefore, we only need to ensure the balance properties L_{j-1}. Since we have just inserted x into the layer L_{j-1}, this can be done by performing the intra-layer operation INSERT-FIXUP(x). Finally, we must remove x from the implicit queue structure of L_j and place it in the implicit queue structure of L_{j-1}.

To do this, we look at both older[x] and younger[x]. If they are both non-nil, then we go to the root and perform searches for older[x] and younger[x], setting younger[older[x]] = younger[x] and older[younger[x]] = older[x]. Otherwise, if only younger[x] is nil, then we conclude that x is the youngest in its former layer. After removing it from that layer, older[x] will be the new youngest element in that layer, so we go to the root search for older[x] and set younger[older[x]] = nil. Since older[x] is the youngest element in that layer, we also copy nextlayer[x] into nextlayer[older[x]]. We must also update the key stored by the youngest element in the next higher layer. In order to do this, we run YOUNGESTINLAYER(L_{j-1}) to find this element, say y, and set nextlayer[y] = older[x]. The case for when only older[x] is nil is symmetric: the new oldest element in the layer is younger[x], so we update older[younger[x]] = nil, we copy nextlayer[x] into nextlayer[younger[x]], and update the pointer to the oldest element in this layer that is stored in L_{j-1} in the same was as we did for the youngest.

We now must insert x into the implicit queue structure of layer L_{j-1}. To do this, we search for the youngest node in L_{j-1}, say y. We then set older[x] = y, younger[x] = nil and younger[y] = x. We then go to the next layer L_{j-2} and update its pointer to the youngest element in this layer the same way we did before.

MOVEDOWN(x)   This operation will move x from its current layer L_j to the next lower layer L_{j+1}. We describe how to perform this operation for red-black trees; other implementations of the layer-subtrees will need to define different implementations but must respect the stated worst-case time bound of $O(2^j)$. Let p denote the predecessor of x in L_j. If x does not have a predecessor in L_j, set $p = x$. Similarly, let s denote the successor of x in L_j, and if x does not have a successor in L_j, set $s = x$. Our first goal is to move x such that it becomes a leaf of its layer-subtree. If x is not already a leaf in L_j, then x has at least one child in its layer-subtree. To make it a leaf of it layer-subtree, we *splice* out the node s by making the parent of s point to the right child of s instead of s itself. Note that this is well-defined since s has no left child in L_j as it is the smallest element greater than x. We then move s to the location of x. Finally, we make x a child of p and make the new children of x the old children of p and s. Figure 3 explains this process.

Observe that we now have that x is a leaf of its layer-subtree. The layer-subtree is configured exactly as if we had deleted x using the deletion operation described by Cormen et al. [5, Section 13.4]. Therefore, we can perform DELETE-FIXUP(s'), where s' is the (only) child of s, to restore the balance properties of the nodes of the layer-subtree. Thus, s' is exactly the child of the node spliced out by the deletion (s), as required by the operation of Cormen et al. [5, Section 13.4].

To complete the movement to the next layer, we change the layer number of x and execute JOIN(x) to create a single balanced layer-subtree from x and its children.[3] We then update the implicit queue structure as we did before. Observe that once x has been removed from its original layer-subtree, layer-subtree balance has been restored because no node on that path was changed.

Lemma 3. *The operations* YOUNGESTINLAYER(L_j) *and* OLDESTINLAYER(L_j), MOVEUP(x) *and* MOVEDOWN(x) *for a layer L_j or a node $x \in L_j$ each take worst-case time $O(2^j)$.*

---

[3]Note that if these children have larger layer numbers than the new layer number for x, nothing is performed and x becomes the lone element in its (new) layer-subtree; this follows from the fact that JOIN(x) only joins nodes that are in the same layer.
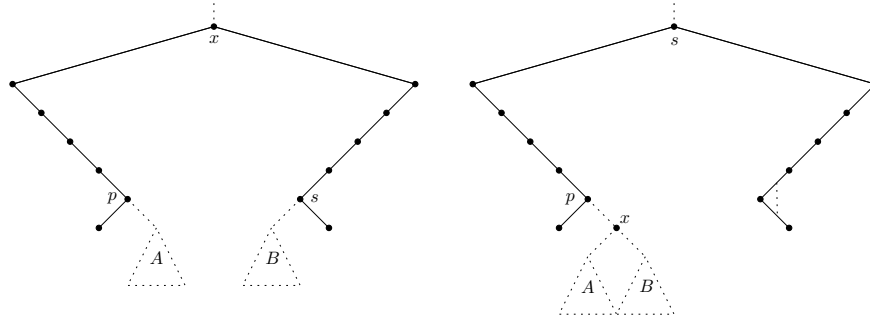
Figure 3: The first part of the MoveDown(x) operation. On the left is the initial layer-subtree and the on the right is the layer-subtree after the nodes have been moved and layers changed but before the Delete-FixUp(s'). The dotted lines to nodes and subtrees indicate layer boundaries and the dotted line over the old node s indicates a splice.

Proof. The operations YoungestInLayer(L_j) and OldestInLayer(L_j) find the youngest (respectively oldest) element in layers $L_1, L_2, \ldots, L_j$. Given the youngest (respectively oldest) element in layer L_k, we can determine the youngest (respectively oldest) element in layer L_{k+1} in constant time since such an element maintains the key of the youngest (respectively oldest) element in the next layer. We then need to traverse from the root to that element. By Lemma 1, the total time is $\sum_{k=1}^{j} O(2^k) = O(2^j)$.

The MoveUp(x) and MoveDown(x) operations, where $x \in L_j$, consist of searching for x, performing a constant number of intra-layer operations and then making series of queries for the youngest elements in several layers and updating the queue structures. The search can be done is $O(2^j)$ time by Lemma 1 and the intra-layer operations each take $O(2^j)$ time by Lemma 2 for a total of $BigOh2^j$. Finally, the queries for the youngest elements and the cost of updating the queues is dominated by the cost of the query in the deepest layer since each layer is twice the size of the previous one. Since $x \in L_j$, this cost is $O(2^j)$ by the above argument. The total cost of MoveUp(x) and MoveDown(x) is thus $O(2^j)$. □

2.5 Tree Operations

We are now ready to describe how to perform the operations Search(x), Insert(x) and Delete(x) on the tree as a whole. Such operations are independent of the layer-subtree implementation given the inter-layer and intra-layer operations defined in the previous sections.

Search(x) To perform a search for x, we begin by performing the usual method of searching in a binary search tree. Once we have found $x \in L_j$, we execute MoveUp(x) a total of $j-1$ times to bring x into L_1. We then restore the sizes of the layers as was done in the working-set structure. We run OldestInLayer(L_1) to find the oldest element y_1 in layer L_1 and then run MoveDown(y_1). We then perform the same operation in L_2 by running OldestInLayer(L_2) to find the oldest element y_2 in layer L_2, then run MoveDown(y_2). This process of moving elements down layer-by-layer continues until we reach a layer L_k such that $|L_k| < 2^{2^k}$.[4] Note that efficiency can be improved by remembering the oldest elements of previous layers instead of finding the oldest element in each of $L_1, \ldots, L_j$ when running OldestInLayer(L_j). Such an improvement does not alter the asymptotic running time, however.

Insert(x) To insert x into the tree, we first examine the index t and size $|L_t|$ of the deepest layer, which we have stored at the root. If $|L_t| = 2^{2^t}$, then we increment t and set $|L_t| = 1$. Otherwise, if $|L_t| < 2^{2^t}$, we simply increment $|L_t|$. We now insert x into the tree (ignoring layers for now) using the usual algorithm where x is placed in the tree as a leaf. We set layer$[x] = t + 1$ (*i.e.*, a temporary layer larger than any other)

---

[4]Note that for an ordinary search, we have $k = j$. However, thinking of the algorithm this way gives us a clean way to describe insertions.

and update the implicit queue structure for L_t (and the youngest and oldest elements of L_{t-1}) as we did before. Finally, we run SEARCH(x) to bring x to L_1. Note that since SEARCH(x) stops moving down elements once the first non-full layer is reached, we do not place another element in layer $t+1$. Thus, this layer is now empty and we update the youngest and oldest elements in layer t to indicate that there is no layer below.

DELETE(x)   To delete x from the tree, we look at the total number t of layers in the tree that is stored at the root. We then locate $x \in T_j$ and perform MOVEDOWN(x) a total of $t - j + 1$ times. This will cause x to be moved to a new (temporary) layer that is guaranteed to have no other nodes in it. Therefore, x must be a leaf of the tree, and we can simply remove it by setting the corresponding child pointer of its parent to nil. As was the case for insertion, this temporary layer is now empty and so we update the youngest and oldest elements in layer t to indicate that there is no layer below. We then perform $t - j + 1$ MOVEUP(y) operations for the youngest element y of each layer from t to j to restore the sizes of the layers. At this point, it could be the case that $|L_t| = 0$. If this happens, we decrement the number of layers t which is stored at the root, and update the youngest and oldest elements in the new deepest layer to indicate that there is no layer below.

Theorem 4. *Searching for x at time i takes worst-case time $O(\lg w_i(x))$ and insertion and deletion each take worst-case time $O(\lg n)$.*

Proof. A search consists of a regular search in a binary search tree followed by several layer operations. Suppose $x \in L_j$ at time i. By Lemma 1, we can find x in time $O(2^j)$. We then perform MOVEUP(x) in time $O(2^j)$ by Lemma 3. We then run OLDESTINLAYER and MOVEDOWN operations for every layer from 1 to j. By Lemma 3, this has total cost $\sum_{k=1}^{j} O(2^k) = O(2^j)$. The total time is therefore $O(2^j)$. Observe that, by the same analysis as that of the working-set structure of Bădoiu et al. [2], we have that $w_i(x) \geq 2^{2^{j-1}}$, and so $O(2^j) = O(\lg w_i(x))$.

An insertion consists of traversing through all layers. By Lemma 1, this takes time $\sum_{k=1}^{t} O(2^k) = O(2^t) = O(2^{\lg \lg n}) = O(\lg n)$. We then perform a search at cost $O(\lg n)$ by the above argument, since the element searched for is in the deepest layer. The total cost is thus $O(\lg n)$.

A deletion consists of traversing the tree to find $x \in L_j$ and then performing MOVEDOWN and MOVEUP at most once per layer. The traversal takes time $O(2^j)$ by Lemma 1 and the MOVEDOWN and MOVEUP operations each cost $O(2^k)$ for L_k by Lemma 3. The total cost is thus $O(2^j) + \sum_{k=1}^{t} O(2^k) = O(2^t) = O(2^{\lg \lg n}) = O(\lg n)$. $\qquad\square$

3   Skip-Splay and the Unified Bound

In this section, we show how to use layered working-set trees in the skip-splay structure of Derryberry and Sleator [6] in order to achieve the unified bound to within a small multiplicative factor. The unified bound [2] requires that the time to search an element x at time i is

$$\text{UB}(x) = O\left(\min_{y \in S_i} \lg(w_i(y) + d(x,y))\right)$$

where $w_i(y)$ is the working-set number of y at time i (as in Section 1) and $d(x,y)$ is defined as the rank distance between x and y. This property implies the working-set and the dynamic finger properties. Informally, the unified bound states that an access is fast if the current access is close in term of rank distance to some element that has been accessed recently. Bădoiu et al. [2] introduced a data structure achieving the unified bound in the amortized sense. This structure does not fit into the binary search tree model, but the splay tree [8], which does fit into this model, is conjectured to achieve the unified bound [2]

Recently, Derryberry and Sleator [6] developed the first binary search tree that guarantees an access time close to the unified bound. Their algorithm, called *skip-splay*, performs an access to the element x in $O(\text{UB}(x) + \lg \lg n)$ amortized time. Insertions and deletions are not supported. In the remainder of this section, we briefly describe skip-splay and then show how to modify it using the layered working-set tree presented in Section 2 in order to achieve a new bound in the binary search tree model.

The skip-splay algorithm works in the following way. Assume for simplicity that the tree T stores the set $\{1, 2, \ldots, n\}$ where $n = 2^{2^{k-1}} - 1$ for some integer $k \geq 0$ and that T is initially perfectly balanced. Nodes of

height 2^i (where the leaves of T have height 1) for $i \in \{0, 1, \ldots, k-1\}$ are marked as the root of a subtree. Such nodes partition T into a set of splay trees called *auxiliary trees*. Each auxiliary tree is maintained as an independent splay tree. Observe that the i-th auxiliary tree encountered on a path from the root to a leaf in T has size $2^{\lg_2 n/2^i} = n^{1/2^i}$. Define $\text{aux}[x]$ to be the auxiliary tree containing the node x.

To access an element x, we perform a standard binary search in T to locate x. We then perform a series of splay operations on some of the auxiliary trees of T. We begin by splaying x to the root of $\text{aux}[x]$ using the usual splay algorithm. If x is now the root of T, the operation is complete. Otherwise, we *skip* to the new parent of x, say y, and splay y to the root of $\text{aux}[y]$. This process is repeated until we reach the root of T.

By using layered working-set trees as auxiliary trees in place of splay trees, we can get the following result.

Theorem 5. *There exists a binary search tree that performs an access to the element x_i in $O(\lg n)$ worst-case time and in $O(\text{UB}(x_i) + \lg \lg n)$ amortized time.*

Proof. As suggested by Derryberry and Sleator [6], instead of using splay trees to maintain the auxiliary trees, we could use any data structure that satisfies the working-set property. Thus, by maintaining the auxiliary trees as layered working-set, we straightforwardly guarantee an amortized time of $O(\text{UB}(x_i) + \lg \lg n)$ to search for an element x_i. Note that the splay in the auxiliary tree corresponds to the $\text{SEARCH}(x)$ operation in our structure.

Now we show that this modified version of the skip-splay has the additional property that the worst case search time is $O(\lg n)$. A search consists of traversing a maximum of k auxiliary trees where the size of the i-th encountered auxiliary tree is $n^{1/2^i}$. In the worst case, the amount of work performed in an auxiliary tree A is $O(\lg |A|)$. Since the auxiliary trees are maintained independently from each other, the total worst-case search cost in the tree T is $O\left(\sum_{i=1}^{k} \lg n/2^i\right) = O(\lg n)$. $\qquad\square$

By doubling the access to an element, we also obtain the following result.

Theorem 6. *The binary search tree described in Theorem 5 performs an access to the element x_i in worst-case time $O(\lg \lg n \lg w_i(x_i))$.*

Proof. Doubling the access to an element increases by at most twice its worst-case access time. Thus, the asymptotic performance of the structure still holds for both the worst-case access time and amortized access time.

In order to reach an element in the tree, we have to traverse several auxiliary trees. Let $A_1, A_2, \ldots, A_k$ be the ordered sequence of trees traversed during an access to the element x_i (note that $k \leq \lg \lg n$). The number of accesses performed independently in each of those trees is bounded above by $w_i(x_i)$.

For $j = 1, 2, \ldots, k-1$, define $d_i(A_j, A_{j+1})$ to be the distance between the root node of A_j and the root node of A_{j+1} in the structure at time i. More generally, define $d_i(A_j, y)$ as the distance between the root node of A_j and the element y where y is a descendent of the root of A_j. Let $p(A_j)$ (and $s(A_j)$) be the greatest (smallest) element of A_{j-1} that is smaller (greater) than any element in A_j. Thus the cost of accessing x_i is $\sum_{j=1}^{k-1} d_i(A_j, A_{j+1}) + d_i(A_k, x_i)$.

By the definition of a search tree we know that the parent of the root node of A_j is either $p(A_j)$ or $s(A_j)$. Thus

$$d_i(A_j, A_{j+1}) = \max\{d_i(A_j, p(A_{j+1})), d_i(A_j, s(A_{j+1}))\} + 1. \tag{1}$$

When we access x_i twice, we independently access both $p(A_j)$ and $s(A_j)$ in each traversed auxiliary tree A_j. By Theorem 4, we have

$$\left. \begin{array}{l} d_i(A_j, p(A_{j+1})) \\ d_i(A_j, s(A_{j+1})) \end{array} \right\} = O(\lg w_i(x_i))) \quad \text{for} \quad j = 1, 2, \ldots, k-1.$$

And we also have $d_i(A_k, x_i) = O(\lg w_i(x_i))$. Hence, by applying equation (1), the result follows. $\qquad\square$

Note that this last property is not satisfied by the original unified structure [2]. Theorems 5 and 6 thus show

Corollary 7. *There exists a binary search tree that performs an access to the element x_i in worst-case time $O(\min\{\lg n, (\lg \lg n) \lg w_i(x_i)\})$ and in $O(\text{UB}(x_i) + \lg \lg n)$ amortized time.*

4 Conclusion and Open Problems

We have given the first binary search tree that guarantees the working-set property in the worst-case. We have also shown how to combine this binary search tree with the skip-splay algorithm of Derryberry and Sleator [6] to achieve the unified bound to within a small additive term in the amortized sense while maintaining in the worst case an access time that is both logarithmic and within a small multiplicative factor of the working-set bound. Several directions remain for future research.

For layered working-set trees, it seems that by forcing the working-set property to hold in the worst case, we sacrifice good performance on some other access sequences. Is it the case that a binary search tree that has the working-set property in the worst case cannot achieve other properties of splay trees? For example, what kind of scanning bound can we achieve if we require the working-set property in the worst case? It would also be interesting to bound the number of rotations performed per access. Can we guarantee at most $O(\lg \lg w_i(x_i))$ rotations to access x_i? Red-black trees guarantee $O(1)$ rotations per update, for instance.

For the results on the unified bound, the most obvious improvement would be to remove the $\lg \lg n$ term from the amortized access cost, as posed by Derryberry and Sleator [6]. Another improvement would be to remove the $\lg \lg n$ factor from the worst-case access cost.

References

[1] R. Bayer. Symmetric binary b-trees: Data structures and maintenance algorithms. *Acta Informatica*, 1: 290–306, 1972.

[2] Mihai Bădoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007.

[3] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM J. Comput.*, 30 (1):44–85, 2000.

[4] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM J. Comput.*, 30(1):1–43, 2000.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[6] Jonathan C. Derryberry and Daniel D. Sleator. Skip-splay: Toward achieving the unified bound in the BST model. In *WADS '09: Proceedings of the 16th Annual International Workshop on Algorithms and Data Structures*, 2009.

[7] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *FOCS '78: Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[8] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3): 652–686, 1985.

[9] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[10] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.