Alternatives to splay trees with $O(\log n)$ worst-case access times

John Iacono*
iacono@cs.rutgers.edu

Department of Computer Science
Hill Center
Rutgers University – New Brunswick
Piscataway NJ 08854

Abstract

Splay trees are a self adjusting form of search tree that supports access operations in $O(\log n)$ amortized time. Splay trees also have several amazing distribution sensitive properties, the strongest two of which are the working set theorem and the dynamic finger theorem. However, these two theorems are shown to poorly bound the performance of splay trees on some simple access sequences. The unified conjecture is presented, which subsumes the working set theorem and dynamic finger theorem, and accurately bounds the performance of splay trees over some classes of sequences where the existing theorems' bounds are not tight. While the unified conjecture for splay trees is unproven, a new data structure, the unified structure, is presented where the unified conjecture does hold. This structure also has a worst case of $O(\log n)$ per operation, in contrast to the $O(n)$ worst case runtime of splay trees. A second data structure, the working set structure, is introduced. The working set structure has the same performance attributed to splay trees through the working set theorem, except the runtime is worst case per operation rather than amortized.

1 Introduction

Splay trees are a form of self adjusting search tree introduced by Sleator and Tarjan [12]. By using a simple restructuring heuristic, splay trees are able to achieve a $O(\log n)$ amortized time per operation, without storing any balance information at each node. The amortized times of a set of operations are defined such that the runtime of any sequence of operations can be no greater than the sum of the amortized times of those operations. An overview of amortized analysis may be found in [15]. Amortized times are useful when the runtime of a sequence of operations, and not the runtimes of individual operations is of prime importance. Moreover, data structures designed to achieve a certain amortized performance are often simpler than their worst-case counterparts. However, if worst case per operation performance is important, data structures designed for amortized analysis, such as splay trees, make poor choices. Splay trees have a $O(n)$ worst case performance per operation. There are many search tree data structures with $O(\log n)$ worst case times, such as AVL trees [1], (2-3)-trees [2], B-trees [3] and red-black trees [8]. However, splay trees are able to take advantage of access patterns that have certain distributional qualities, resulting in runtimes better than $O(\log n)$ per operation. This ability, which we term *distribution sensitivity,* manifests itself in a variety of theorems regarding splay trees. Data structures that have reasonable worst-case runtimes, and moreover, that share several of the types of distribution sensitivity that splay trees exhibit, are not known to exist. The primary goal of this paper is to introduce new data structures that have the same distribution sensitive behavior attributed to splay trees, while having reasonable, $O(\log n)$ or better, worst case runtimes. In the process we present a unified view of the variety of theorems that establish distribution sensitive performance in splay trees.

In the discussion that follows, we limit ourselves to analyzing a series of n insertions, $Y = y_1, \ldots y_n$, in a splay tree at an amortized cost of $O(n \log n)$, followed by a series of m accesses, $X = x_1, x_2, \ldots x_m$, on items found in the splay tree. These accesses have an amortized cost of $\Theta(\log n)$ in the worst case [17], and $\Theta(1)$ in the best case, depending on the exact access pattern X. By limiting the analysis to this sequence of operations, and by assuming the n insertions take $O(n \log n)$ amortized time, allows us to ignore any distribution sensitive effects of insertion and deletion, and of short (smaller than $n \log n$) sequences of operations. While accurate analysis of insertions, deletions and short sequences are

*Research supported by NSF grant CCR-9732689

of interest, and there have been several results for such sequences of operations [12, 6, 5, 16, 13], the focus here is to determine how certain distributional qualities of long access sequences can be utilized by splay trees and other structures.

Let $f(x_i)$ be the frequency of accesses to x_i in the access sequence X. It was shown in [11, 9] that it is possible to achieve the entropy bound of $O(-\log f(x_i))$ for each access, provided the values of $f(y)$ are known in advance for each $y \in Y$. Such search trees, known as optimal search trees, have been improved over the years to allow insertion and deletion, but their fundamental flaw of requiring that the access distribution be known, remains. Splay trees have the property that they share the same $O(-\log f(y_i))$ runtime as optimal search trees, only the bound is amortized, rather than worst case. Amazingly, they achieve this bound without any prior knowledge of the input distribution. This property of splay trees was proven as the static optimality theorem in [12].

Another theorem proved in [12] is the static finger theorem. It states that for any fixed key, f in the splay tree, the amortized time to access an item x is proportional to the logarithm of distance between the "finger" f and x. The distance between two items, x and y, $d(x, y)$, in a data structure storing a set S is defined to be to be the number of keys stored in S whose cardinality is between x and y. Thus the static finger theorem states that given a fixed key f, which does not change over the course of the access sequence X, the amortized time to access x_i is $O(\log(d(x_i, f) + 2))$. The $+2$ is to assure the logarithm is always positive. If f is known, a data structure of Guibas, McCreight, Pass and Roberts [7] has a $O(\log(d(x_i, f) + 2))$ worst cast runtime. However, splay trees achieve this runtime, in the amortized sense, without any knowledge of f.

The working set theorem, which was introduced in [12] is based upon the following idea: If a sequence of accesses only access a subset of size n' of the n items, the amortized time to access items in this subset should be $O(\log n')$, instead of $O(\log n)$. The actual theorem is stated using the stronger idea that the items that have been accessed recently should take less time to access than items that have not been accessed in a long time. Let $t_i(z)$ be the number of distinct items accessed since the last time an access was performed on z before the execution of the access on x_i. For the purpose of computing the t values, insertions count as accesses. Note that $t_i(z)$ is at most n. It was shown that in splay trees the amortized time to access x_i is $O(\log(t_i(x_i) + 2))$ [12]. We have shown in [10], that the working set theorem implies the static optimality theorem for sufficiently long access sequences. It can also be seen from the way weights are assigned to items in the proofs of the static optimality theorem and the static finger theorem in [12] that the static optimality theorem implies the static finger theorem.

As the strongest of the above three theorems regarding splay trees, the working set theorem is the property that our first data structure will be designed around. This data structure, which we call the working set structure, has the same performance has the same $O(\log(t_i(x_i) + 2))$ performance attributed to splay trees through the working set theorem, except the performance is worst case instead of amortized. Since the working set theorem implies the static finger theorem which, in turn, implies the static optimality theorem, these two theorems hold for the working set structure, albeit in the amortized sense only. It can easily be shown that the static finger theorem and the static optimality theorem can not hold in the worst case in any data structure without knowledge of the finger, or the access distribution, respectively. The working set structure has a worst-case access time of $O(\log n)$, a significant improvement over the $O(n)$ of splay trees. The working set structure is relatively simple to implement, and a naive implementation will have a worst-case performance slightly over four times that of AVL trees. By performing some optimizations, this factor of four can be reduced so that the working set structure performs competitively in practice, especially on non-uniform access sequences.

One distribution sensitive property that splay trees have that is not implied by the working set theorem is that if the access pattern X simply consists of accessing every item in the data structure in sorted order repeatedly, the amortized cost per operation is $O(1)$. This is known as the sequential access lemma and was proven by Tarjan [16], with an alternative proof appearing in Sundar's thesis [14].

The dynamic finger theorem, proven by Cole [5], states that an access should be fast if it is close, in terms of distance, to the previous access. In splay trees, the amortized cost to access x_i is $O(\log(d(x_i, x_{i-1}) + 2))$. The $+2$ is to ensure that the logarithm is defined and positive. A non-self adjusting data structure that has this performance predates splay trees: The level linked trees of Brown and Tarjan [4] support accesses in $O(\log(d(x_i, x_{i-1})+2))$ worst case time, while supporting insertions and deletions in $O(\log n)$ worst-case time, and even faster under certain conditions.

The working set theorem and the dynamic finger theorem are the best known analyses of access sequences in splay trees but yet each is easily seen to be flawed. Consider the following three access sequences:

$X_1 : 1, 2, \ldots, n, 1, 2, \ldots n, 1, 2 \ldots$

$X_2 : 1, 2, \ldots, n, 1, n, 1, n, 1, n, \ldots$

$X_3 : 1, \frac{n}{2}+1, 2, \frac{n}{2}+2, 3, \frac{n}{2}+3, \ldots \frac{n}{2}, n, 1, \frac{n}{2}+1, \ldots$

The sequences have length m, where $m \geq n \log n$ and n is even. In X_1 the dynamic finger theorem would tightly bound this sequence as taking $O(m)$ to execute on a splay tree, while the working set theorem could only say the runtime was $O(m \log n)$. The situation is reversed in X_2, with the working set theorem tightly bounds the execution time as $O(m)$ while the dynamic finger theorem only yields an $O(m \log n)$ bound. More troubling is X_3. Both the working set theorem and the dynamic finger theorem say this sequence takes time $O(m \log n)$, however, this is not tight, as this sequence executes in time $O(m)$. This can be seen by proving a new theorem, based on the sequential access lemma. However, introducing new theorems that prove the runtimes of highly specific classes sequences such as X_3 will only contribute to our fragmented understanding of splay trees. In an attempt to more accurately characterize the runtime of accesses on splay trees, we provide the following conjecture:

Unified Conjecture[1]: The time to access x_i in a splay tree storing static set S is

$$O(\log \min_{y \in S}(t_i(y) + d(x_i, y) + 1))$$

This conjecture, while implying the working set theorem and the dynamic finger theorem, is also strong enough to predict that X_3, along with many possible variants, takes time $O(m)$. Informally, it says a access is fast if the access is close to, in terms of distance, some item that has been accessed recently. In the case if X_3 the majority of the accesses are to items that are at a distance of one away from the item accessed two accesses ago, thus the amortized cost per access is $O(\log 1+2+2)$. We offer no proof of this conjecture.

However, we present a data structure, which we call the unified structure, which has the same performance attributed to splay trees by the unified conjecture. This structure serves to demonstrate the plausibility of the unified conjecture for splay trees. It also has a worst case runtime of $O(\log n)$, which is significantly better then the worst case runtime of $O(n)$ for splay trees. This data structure is the best comparison based static data structure known, in terms of the strength of theorems proved about it. The analysis of unified trees is relatively simple, especially compared to the proof of the dynamic finger theorem found in [6, 5]. The unified structure is prohibitively complicated for practical use, requiring 21 pointers per node. This structure is static, although we are currently working on a dynamic version. Finally, while the unified theorem holds for unified trees, and we are unable to demonstrate that it holds for splay trees, this says nothing about their actual relative performance. Although we know of none, it may be entirely possible that certain classes of sequences exist that will execute asymptotically faster on splay trees than on unified trees or vice versa.

The dynamic optimality conjecture of Sleator and Tarjan [12], states that splay trees can execute any sufficiently long access sequence as fast as any rotation based binary search tree, within constant factors. As the unified structure is composed of a collection of search trees, not one search tree, we are unable to derive any dynamic optimality based statements from the results presented here. In particular, assuming the dynamic optimality conjecture holds does not imply the unified conjecture holds for splay trees. Conversely, a disproof of the unified theorem for splay trees would not disprove the dynamic optimality theorem.

2 The Working Set Structure

The working set structure is a comparison based dictionary. It supports insertion, deletion and access operations to maintain a dynamic set S of n keys. The data structure consists of a series of AVL, or other balanced binary, trees $t_1, t_2, \ldots t_k$, and a series of queues $q_i, q_2, \ldots q_k$. For any i, the contents of q_i and t_i are identical, and pointers are maintained between the corresponding elements of t_i and q_i. The size of t_i and q_i, where $i < k$ is $2^{(2^i)}$. The size of t_k and q_k is $n - \sum_{i=1}^{k-1} 2^{(2^i)}$. Thus the sums of the sizes of the trees is n. The queues support arbitrary deletion with a pointer. Every key in the set S may be found in exactly one tree and exactly one queue. If there are n items in the set S, the number of queues and trees, k, is $O(\log \log n)$.

The basic operation is called the shift. In a shift from a to b, if $a < b$, for every c from a to $b - 1$, an item is dequeued from q_c and enqueued into q_{c+1}. The corresponding item in t_c is deleted and inserted into t_{c+1}. The runtime is $O(\sum_{i=a}^{b} \log 2^{(2^i)}) = O(2^b)$. If $a > b$, for every c from a to $b + 1$, an item is dequeued from q_c and enqueued into q_{c-1}. The corresponding item in t_c is deleted and inserted into q_{c-1}. The runtime is $O(\sum_{i=b}^{a} \log 2^{(2^i)}) = O(2^a)$. In either case the net effect is that the size of q_a and t_a has decreased by one, and q_b and t_b have increased by one. If $a = b$ nothing is done.

Inserting a new key x is implemented by inserting x into t_1 and enqueuing it into q_1. If the size of t_k is $2^{(2^k)}$, k is incremented and a new empty t_k and q_k is created. A shift from 1 to k is performed. The runtime

[1]Note that a unified theorem for splay trees is presented in [12], which is a linear combination of the static optimality theorem, the static finger theorem and the working set theorem. This theorem is distinct from the one presented here.

is $O(\log 2^{(2^k)}) = O(\log 2^{(2^{\log\log n})}) = O(\log n)$.

Deleting an element x is performed by searching for x in $t_1, t_2, \dots t_k$ until it is either found in some tree, t_l, or the deletion ends unsuccessfully if a node with key x is not found in any of the trees. If a node with key x is found, it is deleted from t_l and q_l and a shift is performed from k to l. If t_k and q_k are empty, they are removed and k is decremented. The runtime is $O(\log 2^{(2^k)}) = O(\log 2^{(2^{\log\log n})}) = O(\log n)$.

Accessing an element x is performed by searching for x in $t_1, t_2, \dots t_k$ until it is either found in t_l, for some l, or the search ends unsuccessfully. If x is found, it is deleted from t_l and q_l and inserted into t_1 and enqueued into q_1. A shift is performed from 1 to l, which restores the size of the trees and queues to their size at the beginning of the operation. The runtime is $O(\log 2^{(2^l)})$ unless the search was unsuccessful, in which case the runtime is $O(\log 2^{(2^k)}) = O(\log 2^{(2^{\log\log n})}) = O(\log n)$.

If x was found in t_1 the access time is $O(1)$. If $l > 1$ we proceed with the following analysis: Since x was originally inserted into t_1, and every time x was accessed it was moved to t_1, and since items which are not being accessed, may only move to the next larger or smaller tree, it may be concluded that, at some point since x's most recent access, or insertion if it was never accessed, x was in t_{l-1} and was deleted and moved to t_l. Furthermore, x was selected for deletion from t_{l-1} by being dequeued from q_{l-1}. At the time that x's was deleted from q_{l-1}, $l-1$ was not k and thus the size of q_{l-1} was exactly $2^{(2^{l-1})}$. Since at most one enqueue, and one dequeue, is performed on each queue, per insertion, deletion, or find, one can conclude that x was in q_{l-1} and t_{l-1} for a period of time between x's most recent access, or its insertion if it was never accessed, during which time $2^{(2^{l-1})}$ operations were performed. Thus $t(x) \geq 2^{(2^{l-1})}$. The runtime is $O(\log 2^{(2^l)}) = O(\log((2^{(2^{l-1})})^2)) = O(\log t(x))$.

The working set structure, while composed of search trees, is not a search tree and thus does not support tree operations, such as traversals. Of particular importance is the inorder traversal, which may be implemented my merging the inorder traversals of the $O(\log\log n)$ AVL trees. If the traversals are merged incrementally, starting with the smallest tree, the total time to merge them is $O(n)$.

3 The Unified Structure

The unified structure stores a static set S of n keys. The data structure is a compound one. The framework of the data structure is a set of $k = \lceil \log\log n \rceil$ AVL trees $t_1, t_2, \dots t_k$. The size of all trees t_i, $i < k$ is $2^{(2^i)}$. The size of t_k is n. The set of key values stored in t_k is the set S, while the key values of the elements of all of the other trees are a subset of S, with possible repetitions of items allowed. Every element x of t_i, $i < k$, has a pointer to an element in t_{i+1}, which we refer to as the left pointer of x, $lp(x)$. We also define the right pointer set of an item x, in tree t_i, to be the set of items in t_{i-1} that have x as their left pointer. For each t_i, $i < k$, we maintain a queue that supports enqueue, dequeue, and deletion with a pointer. We denote the queue associated with t_i as q_i. The queue will have in it every item x in t_i with an empty right pointer set. Thus the size of the q_i will be at least $2^{(2^i)} - 2^{(2^{i-1})}$.

Since S is a static totally ordered set, we presort this set and store it in an array, A. Every element x of a tree has an integer $index(x)$, which corresponds to the index i such that $A[i]$ is the key stored in x.

The right pointer set of node x in t_i is restricted to only contain items whose $index$ values lie in the range $index(x) - 2^{(2^i)}$ to $index(x) + 2^{(2^i)} - 1$. The right pointer set is actually stored in four subsets, which we denote as $rp1(x)$, $rp2(x)$, $rp3(x)$, and $rp4(x)$, where $rp1(x)$ stores links to items in the right pointer set that have $index$ values in the range $index(x) - 2^{(2^i)}$ to $index(x) - \frac{2^{(2^i)}}{2} - 1$. The sets $rp2(x)$, $rp3(x)$ and $rp4(x)$ correspond to the ranges of index values $index(x) - \frac{2^{(2^i)}}{2}$ to $index(x) - 1$, $index(x)$ to $index(x) + \frac{2^{(2^i)}}{2} - 1$, and $index(x) + \frac{2^{(2^i)}}{2}$ to $index(x) + 2^{(2^i)} - 1$, respectively. The total space used by this data structure is $O(n)$.

The search set of a node x, $s(x)$ is defined to be the set of keys located in the array at positions $A[index(x) - 2 * 2^{(2^i)}]$ to $A[index(x) + 2 * 2^{(2^i)}]$. The search set of a tree i, $s(t_i)$, is the union of the search sets of all of the nodes in the tree. It is possible to determine whether a key a is in the search set of t_i in time $O(2^i)$ by using the following procedure: Search the AVL tree t_i for a. If a is not in the tree, let x be the largest node with key smaller than a in the tree and let y be the smallest node with key larger than a in the tree. Binary search the array for a in the ranges $A[index(x) - 2 * 2^{(2^i)}]$ to $A[index(x) + 2 * 2^{(2^i)}]$ and $A[index(y) - 2 * 2^{(2^i)}]$ to $A[index(y) + 2 * 2^{(2^i)}]$. If a is in the search set of t_i, it will be found using this procedure. The three AVL tree operations take time $O(2^i)$ since tree t_i has at most $2^{(2^i)}$ items, and the two binary searches over a range of $4 * 2^{(2^i)}$ each take time $O(2^i)$.

An access operation on an key a is carried out as follows:

Search the search set of $t_1, t_2, \dots$ until a is found in the search set of tree t_l. Let y be the element in t_l that is closest in the array to a. Using the procedure outlined in the next paragraph, a new node with key value a is inserted into t_l, with the same left pointer as y. Using this same procedure, new nodes with key values a are also inserted into $t_{l-1}, t_{l-2}, \ldots t_1$, with the left pointer for these insertions being the item that was just inserted into the previous tree.

In order to insert a new node x into tree t_i with a left pointer to y in tree t_{i+1}, it is required that the key stored in x is either equal to the key stored in y or the key stored in x is in the search set of one item in the right pointer set of y. There are two cases, depending on whether $index(y) - 2^{(2^{i+1})} \leq index(x) \leq index(y) + 2^{(2^{i+1})} - 1$. If $index(x)$ is in this range, it is allowed to be in the right pointer set of y, and the procedure is relatively simple. In this case, which we call a basic insertion, x is inserted into AVL tree t_i. Its left pointer it set to y. A link to x is added to one of the right pointer sets of y, depending on the exact relationship between $index(x)$ and $index(y)$. If y is contained in q_{i+1}, it will be deleted from the queue using a pointer stored in y for this purpose, as its right pointer sets are no longer empty. As the new item, x, has an empty right pointer set, it is enqueued into q_i and a pointer for deleting it from the queue is stored in x. In order to maintain the size of t_i one item must be deleted, to compensate for the insertion of x. This is done by dequeuing an item, call it z, from q_i. z is deleted from t_i. Let w be the item in t_{i+1} that z's left pointer points to. The link to z is removed from one of w's right pointer sets. If this causes all of w's right pointer sets to become empty, w is then enqueued into q_{i+1}. Other than the AVL tree insertion and deletion, which take $O(2^i)$ time, all of the operations take constant time.

The second case, which we call an extended insertion, is where $index(x)$ is outside of the range from $index(y) - 2^{(2^{i+1})}$ to $index(y) + 2^{(2^{i+1})} - 1$. In this case x can not be a element of the right pointer set of y, so a new node must be inserted into t_{i+1} to be the object of x's left pointer. This case is more complicated, but we will show that it is rare, compared to basic insertions. The details are provided where $index(x) < index(y)$, and the case where $index(x) > index(y)$ is handled symmetricly. Since x and y must not have the same key value, it can be concluded that x is in the search set of one item, call it z, in the right pointer set of y. Since the size of the search set of z is small, $4 * 2^{(2^i)}$, compared to the number of distinct keys that could appear in the right pointer set of y, $2 * 2^{(2^{i+1})}$, it may be concluded that z is in the first right pointer set of y. A new item, w, is recursively inserted with key value $A[index(y) - 2^{(2^{i+1})}]$ and with the same left pointer as y into t_{i+1}. Then, the first right pointer set of y, which will not be empty, is moved to the third right pointer set of w. This can be done in constant time with a suitable implementation of this structure. One such implementation is described in detail in the next section. As the right pointer set of w is nonempty, w must then be removed from q_{i+1}. Should the removal of the first right pointer set of y cause y's right pointer set to become empty, y must be enqueued into q_{i+1}. Next, x is inserted with left pointer w using the basic insertion algorithm of the previous paragraph.

Worst case runtime: In the worst case, a fixed number of AVL tree operations, plus a constant number of other $O(1)$ operations are done on each of the AVL trees. Thus since the size of each of each tree is $2^{(2^i)}$, and the total number of trees is $\lceil \log\log n \rceil$, the runtime is $O(\Sigma_{i=1}^{\lceil \log\log n \rceil}(\log 2^{(2^i)})) = O(\log n)$.

The potential method is used to calculate the amortized runtime of the access operation. The amortized time of access x_i, $\hat{a}_i$, is defined to be the actual time of the access, a_i, plus the change in potential, $\Phi_i - \Phi_{i-1}$. Summing over the sequence S and rearranging yields: $\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \hat{a}_i + \Phi_0 - \Phi_m$. Thus, the actual runtime of a sequence of accesses is equal to the sum of the amortized time of the accesses plus the net loss of potential. Since the potential function we will define is initially zero, and is always nonnegative, the sum of the amortized times is an upper bound on the actual runtime of any sequence. For a full discussion of the potential method, and amortized analysis in general, see [15].

In order to define the potential function two counters are associated with each node x, right counter 1, and right counter 4. These counters are initialized to zero for new elements, and are incremented by one whenever a new item is added to the first or fourth right pointer set of an item. These counters are reset to zero whenever the right pointer set associated with a counter is transferred to a different node. The potential of the data structure is the sum of the counters, multiplied by a constant c, to be determined below. The counters are used for the analysis only, they need not be actually maintained.

During an access on an item found in $s(t_l)$, all of the insertions into trees $t_1, \ldots t_{l-1}$ insert nodes with key values that equal the key values of their left pointer, thus these insertions are basic insertions. The insertion on t_l may be either a basic insertion or an extended insertion. The cost of each these basic insertions into a tree t_i is comprised of a fixed number of AVL operations at a cost of $O(2^i)$, $O(1)$ maximum potential gain, and $O(1)$ other work for queue operations and other overhead. Thus the total amortized cost of all l of all basic insertions is at most $\Sigma_{i=1}^{l}(O(2^i) + O(1) + O(1)) = O(2^l)$.

Any insertions in the trees t_i in the range t_{l+1} to t_k are also done at an actual cost of 2^i. However, these insertions are extended insertions that perform a copying of the contents of the first or fourth right pointer set of an existing node to the third or second right pointer set of a new node, and thus have a corresponding potential loss. The insertion into t_l may be an extended insertion as well. When a new node is inserted, its first and fourth right pointer sets are empty. Any additions to the right pointer set of a node in t_i has an *index* value that differs from an existing element of the right pointer set by $2*2^{(2^{i-1})}$. Extended insertions in t_i remove one of the right pointer sets of a node x in t_i when an new node is being inserted into t_{i-1} that has an index value within $2*2^{(2^{i-1})}$ of a node in the first or fourth right pointer pointer set of x, but yet has an index value outside of the allowable range for elements of the right pointer set of x, $index(x) - 2^{(2^i)}$ to $index(x) + 2^{(2^i)} - 1$. This indicates that there is a node in the right pointer set to be moved that has an index value within $2*2^{(2^{i-1})}$ of the boundary of allowable values for the right pointer set. Thus the right pointer set to be moved must have had at least $\frac{2^{(2^i)}/2}{2*2^{(2^{i-1})}} = \frac{(2^{(2^{i-1})})^2}{4*2^{(2^{i-1})}} = \frac{2^{(2^{i-1})}}{4}$ items inserted. Thus, an extended insertion in tree t_i, which has actual cost $O(2^i)$, incurs a potential loss of at least $\frac{c2^{(2^{i-1})}}{4}$. By choosing c to be sufficiently large, the potential loss will always dominate the actual cost of $O(2^i)$ and thus extended insertions have an amortized cost of at most 0. Thus the amortized cost of an access operation on an item that is in found in the search set of tree t_l is simply the amortized cost of the basic insertions, $O(2^l)$.

Suppose at time j, an access was made to an element with key value x_j. A new item with x_j as the key value is inserted into t_1. Call this element $p(x_j, 1)$. Let the item in tree t_2, that the left pointer of $p(x_j, 1)$ points to after the execution of the access on x_j be known as $p(x_j, 2)$. In general, recursively define define $p(x_j, i)$, $1 < i \leq k$ to be the node in t_i that the left pointer of $p(x_j, i-1)$ points to. It can be observed that, at time j, for all i between 1 and k, x_j is in the search set of $p(x_j, i)$. Moreover, x_j is in the center half of the search set of $p(x_j, i)$, so that any item within a distance of $2^{(2^i)}$ from x_j is in the search set of $p(x_j, i)$. This implies that any item it a distance less than $2^{(2^i)}$ from x_j is in the search set of t_i as long as $p(x_j, i)$ remains in t_i. Three facts about the unified structure allow establishing a minimum amount of time $p(x_j, i)$ remains in t_i. First, items are only removed from t_i after being dequeued from q_i. Second, at most 2 dequeue, or queue delete operations are permitted on each queue during every access. Third, all queues q_i have a size of at least $2^{(2^i)} - 2^{(2^{i-1})}$. It can be concluded that at any time, any item whose distance lies within $2^{(2^i)}$ of any of the most recently accessed $\frac{2^{(2^i)} - 2^{(2^{i-1})}}{2}$ lies in the search set of t_i.

Theorem The amortized time to access x_i in the unified structure is $O(\log \min_{y \in S} t_i(y) + d(x_i, y) + 2)$.

Proof Suppose $\frac{2^{(2^{l-1})}}{2} \leq \min_{y \in S} t_i(y) + d(x_i, y) + 2) < \frac{2^{(2^l)}}{2}$. This indicates that x_i is in the search set of tree t_l. The access then takes amortized time $O(2^l)$. Since $\frac{2^l}{2} - 1 \leq \log \min_{y \in S} t_i(y) + d(x_i, y) + 2) \leq 2^l - 1$, it may be concluded that the amortized time to access x_i is $O(\log \min_{y \in S} t_i(y) + d(x_i, y) + 2)$.

4 Implementing the Unified Structure

In this section the details are described as to the exact pointer setup required in the unified structure. The most complicated aspect of the implementation is the implementation of the left pointer and the right pointer sets. The following manipulations of the left and right pointer sets are carried out by the access operation: Set the left pointer of a node; access the left pointer of a node; add a node to a right pointer set of a node, while receiving a pointer to use to delete that node from the right pointer set later; deleting a node from a right pointer set, using a pointer; determining whether a right pointer set is empty; moving a right pointer set of one node, x, to another, y, such that subsequent left pointer query to items in the right pointer set return y. All of these operations must be performed in constant time. Each of the right pointer sets is implemented as a doubly linked list. Thus unordered insertion, deletion with pointer, testing for emptiness and moving the set to another node take constant time. Each node x stores a left deletion pointer, which is used to delete it from whatever right pointer set it belongs to whenever x itself is being deleted. Every node has four handles, meaning a pointer to a pointer, to itself. Each of these handles is associated with one of the right pointer sets. To implement the "left pointer", a node x that logically has left pointer y, has a left handle pointer to the same pointer that the handle associated with the right pointer set of y that x is an element of points. Thus in order to determine the the what the logical left pointer of x points to, one needs only to dereference the left handle pointer twice. When moving one of the right pointer sets from one node, x, to another y, the right handle should be copied as well, and the node the right handle points to should be changed from x to y. This will have the desired effect of whenever the left handle pointer of an element in this moved right pointer set is dereferenced twice, the result will now be y. This implementation of the left pointer and right pointer sets, while it gives the desired runtimes is pointer-intensive, and will require 17 pointers per node. Add in the two pointers to maintain each node as part of a queue, the integer index into the array, and the two pointers plus balance field required of every AVL tree nodes, for a total of 21 pointers, one integer, and a balance field for every node.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet. Math.*, 3:1259–1262, 1962.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullmann. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[3] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inform.*, pages 290–306, 1972.

[4] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM J. Comput.*, 9:594–614, 1980.

[5] R. Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. Technical Report Computer Science TR1995-701, New York Univerity, 1995.

[6] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. Technical Report Computer Science TR1995-700, New York Univerity, 1995.

[7] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing*, pages 49–60, 1977.

[8] L. J. Guibas and R. Sedgewick. A dichomatic framework for balanced trees. In *Proc. 19th Ann. IEEE Symp. on Theory of Computing*, pages 8–21, 1978.

[9] T. C. Hu and A. C. Tucker. Optimal computer-search trees and varible-legth alphabetic codes. *SIAM J. Appl. Math.*, 37:246–256, 1979.

[10] J. Iacono. New upper bounds for pairing heaps. In *Scandanavian Workshop on Algorithm Theory*, 2000. To appear.

[11] D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.

[12] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *JACM*, 32:652–686, 1985.

[13] A. Subramanian. An explanation of splaying. *Journal of Algorithms*, 20:512–525, 1996.

[14] R. Sundar. *Amoritzed Complexity of Data Structures.* PhD thesis, New York University, 1991.

[15] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[16] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985.

[17] R. Wilbur. Lower bounds for accessing binary search trees with rotations. In *Proc. 27th Symp. on Foundations of Computer Science*, pages 61–69, 1986.