Randomized Search Trees

RAIMUND SEIDEL[1]

Computer Science Division
University of California Berkeley
Berkeley CA 94720

Fachberich Informatik
Universität des Saarlandes
D-66123 Saarbrücken, GERMANY

CECILIA R. ARAGON[2]

Computer Science Division
University of California Berkeley
Berkeley CA 94720

Abstract

We present a randomized strategy for maintaining balance in dynamically changing search trees that has optimal *expected* behavior. In particular, in the expected case a search or an update takes logarithmic time, with the update requiring fewer than two rotations. Moreover, the update time remains logarithmic, even if the cost of a rotation is taken to be proportional to the size of the rotated subtree. Finger searches and splits and joins can be performed in optimal expected time also. We show that these results continue to hold even if very little true randomness is available, i.e. if only a logarithmic number of truely random bits are available. Our approach generalizes naturally to weighted trees, where the expected time bounds for accesses and updates again match the worst case time bounds of the best deterministic methods.

We also discuss ways of implementing our randomized strategy so that no explicit balance information is maintained. Our balancing strategy and our algorithms are exceedingly simple and should be fast in practice.

This paper is dedicated to the memory of *Gene Lawler*.

1 Introduction

Storing sets of items so as to allow for fast access to an item given its key is a ubiquitous problem in computer science. When the keys are drawn from a large totally ordered set the method of choice for storing the items is usually some sort of search tree. The simplest form of such a tree is a binary search tree. Here a set X of n items is stored at the nodes of a rooted binary tree as follows: some item $y \in X$ is chosen to be stored at the root of the tree, and the left and right children of the root are binary search trees for the sets $X_< = \{x \in X \mid x.key < y.key\}$ and $X_> = \{x \in X \mid y.key > x.key\}$, respectively. The time necessary to access some item in such a tree is then essentially determined by the depth of the node at which the item is stored. Thus it is desirable that all nodes in the tree have small depth. This can easily be achieved if the set X is known in advance and the search tree can be constructed off-line. One only needs to "balance"

[1]Supported by NSF Presidential Young Investigator award CCR-9058440. Email: `seidel@cs.uni-sb.de`
[2]Supported by an AT&T graduate fellowship

the tree by enforcing that $X_<$ and $X_>$ differ in size by at most one. This ensures that no node has depth exceeding $\log_2(n+1)$.

When the set of items changes with time and items can be inserted and deleted unpredictably, ensuring small depth of all the nodes in the changing search tree is less straightforward. Nonetheless, a fair number of strategies have been developed for maintaining approximate balance in such changing search trees. Examples are AVL-trees [1], (a, b)-trees [3], $BB(\alpha)$-trees [13], red-black trees [11], and many others. All these classes of trees guarantee that accesses and updates can be performed in $O(\log n)$ worst case time. Some sort of balance information stored with the nodes is used for the restructuring during updates. All these trees can be implemented so that the restructuring can be done via small local changes known as "rotations" (see Fig. 1). Moreover, with the appropriate choice of parameters (a, b)-trees and $BB(\alpha)$-trees guarantee that the average number of rotations per update is constant, where the average is taken over a sequence of m updates. It can even be shown that "most" rotations occur "close" to the leaves; roughly speaking, for $BB(\alpha)$-trees this means that the number of times that some subtree of size s is rotated is $O(m/s)$ (see [17]). This fact is important for the parallel use of these search trees, and also for applications in computational geometry where the nodes of a primary tree have secondary search structures associated with them that have to be completely recomputed upon rotation in the primary tree (e.g. range trees and segment trees; see [18]).

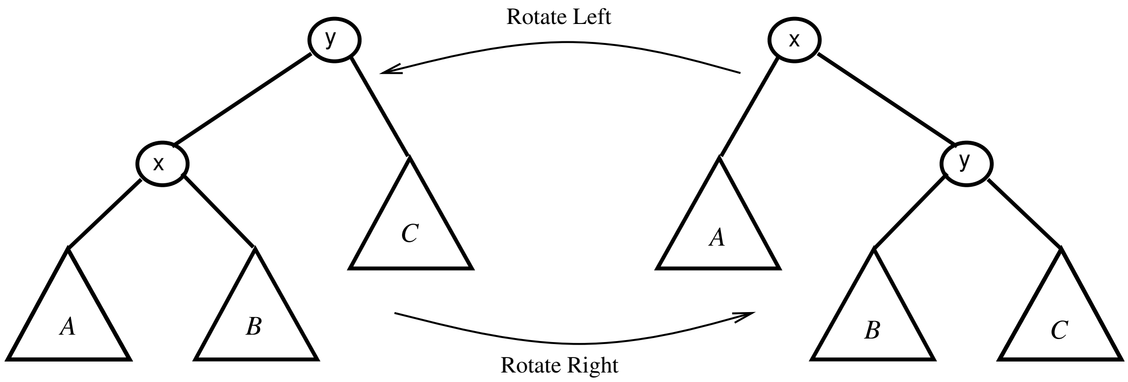

Figure 1.1:

Sometimes it is desirable that some items can be accessed more easily than others. For instance, if the access frequencies for the different items are known in advance, then these items should be stored in a search tree so that items with high access frequency are close to the root. For the static case an "optimal" tree of this kind can be constructed off-line by a dynamic programming technique. For the dynamic case strategies are known, such as biased 2-3 trees [4] and D-trees [17], that allow accessing an item of "weight" w in worst case time $O(\log(W/w))$, which is basically optimal. (Here W is the sum of the weights of all the items in the tree.) Updates can be performed in time $O(\log(W/\min\{w-, w, w+\})$, where $w-$ and $w+$ are the weights of the items that precede and succeed the inserted/deleted item (whose weight is w).

All the strategies discussed so far involve reasonably complex restructuring algorithms that require some balance information to be stored with the tree nodes. However, Brown [7] has pointed out that some of the unweighted schemes can be implemented without storing any balance information explicitly. This is best illustrated with schemes such as AVL-trees or red-black trees, which require only one bit to be stored with every node: this bit can be implicitly encoded by the order in which the two children pointers are stored. Since the identities of the children can be recovered from their keys in constant time, this leads to only constant overhead to the search and update times, which thus remain logarithmic.

There are methods that require absolutely no balance information to be maintained. A partiucarly attractive one was proposed by Sleator and Tarjan [25]. Their "splay trees" use an extremely simple restructuring strategy and still achieve all the access and update time bounds mentioned above both for the unweighted and for the weighted case (where the weights do not even need to be known to the algorithm). However, the time bounds are not to be taken as worst case bounds for individual operations, but as *amortized* bounds, i.e. bounds averaged over a (sufficiently long) sequence of operations. Since in many applications one performs long sequences of access and update operations, such amortized bounds are often satisfactory.

In spite of their elegant simplicity and their frugality in the use of storage space, splay trees do have some drawbacks. In particular, they require a substantial amount of restructuring not only during updates, but also during accesses. This makes them unusable for structures such as range trees and segment trees in which rotations are expensive. Moreover, this is undesirable in a caching or paging environment where the writes involved in the restructuring will dirty memory locations or pages that might otherwise stay clean.

Recently Galperin and Rivest [10] proposed a new scheme called "scapegoat trees," which also needs basically no balance information at all and achieves logarithmic search time even in the worst case. However logarithmic update time is achieved only in the amortized sense. Scapegoat trees also do not seem to lend themselves to applications such as range trees or segment trees.

In this paper we present a strategy for balancing unweighted or weighted search trees that is based on randomization. We achieve *expected case* bounds that are comparable to the deterministic worst case or amortized bounds mentioned above. Here the expectation is taken over all possible sequences of "coin flips" in the update algorithms. Thus our bounds do not rely on any assumptions about the input. Our strategy and algorithms are exceedingly simple and should be fast in practice. For unweighted trees our strategy can be implemented without storage space for balance information.

Section 2 of the paper describes *treaps*, the basic structure underlying randomized search trees. In section 3 unweighted and weighted randomized search trees are defined and all our main results about them are tabulated. Section 4 contains the analysis of various expected quantities in randomized search trees, such as expected depth of a node or expected subtree size. These results are then used in section 5, where the various operations on randomized search trees are described and their running times are analyzed. Section 6 discusses how randomized search trees can be implemented using only very few truly random bits. In section 7 we show how one can implement randomized search trees without maintaining explicit balance information. In section 8 we offer a short comparison of randomized search trees and skip lists, another randomized data structure.

2 Treaps

Let X be a set of n items each of which has associated with it a *key* and a *priority*. The keys are drawn from some totally ordered universe, and so are the priorities. The two ordered universes need not be the same. A *treap* for X is a rooted binary tree with node set X that is arranged in in-order with respect to the keys and in heap-order with respect to the priorities.[3] "In-order" means that for any node x in the tree $y.key \le x.key$ for all y in the left subtree of x and $x.key \le y.key$ for y in the right subtree of x. "Heap-order" means that for any node x with parent z the relation

[3]Herbert Edelsbrunner pointed out to us that Jean Vuillemin introduced the same data structure in 1980 and called it "Cartesian tree" [V]. The term "treap" was first used for a different data structure by Ed McCreight, who later abandoned it in favor of the more mundane "priority search tree" [Mc].

$x.priority \leq z.priority$ holds. It is easy to see that for any set X such a treap exists. With the assumption that all the priorities and all the keys of the items in X are distinct — a reasonable assumption for the purposes of this paper — the treap for X is unique: the item with largest priority becomes the root, and the allotment of the remaining items to the left and right subtree is then determined by their keys. Put differently, the treap for an item set X is exactly the binary search tree that results from successively inserting the items of X in order of decreasing priority into an initially empty tree using the usual leaf insertion algorithm for binary search trees.

Let T be the treap storing set X. Given the key of some item $x \in X$ the item can easily be located in T via the usual search tree algorithm. The time necessary to perform this access will be proportional to the depth of x in the tree T. How about updates? The insertion of a new item z into T can be achieved as follows: At first, using the key of z, attach z to T in the appropriate leaf position. At this point the keys of all the nodes in the modified tree are in in-order. However, the heap-order condition might not be satisfied, i.e. z's parent might have a smaller priority than z. To reestablish heap-order simply rotate z up as long as it has a parent with smaller priority (or until it becomes the root). Deletion of an item x from T can be achieved by "inverting" the insertion operation: First locate x, then rotate it down until it becomes a leaf (where the decision to rotate left or right is dictated by the relative order of the priorites of the children of x), and finally clip away the leaf.

At times it is desirable to be able to *split* a set X of items into the set $X_1 = \{x \in X \mid x.key < a\}$ and the set $X_2 = \{x \in X \mid x.key > a\}$, where a is some given element of the key universe. Conversely, one might want to *join* two sets X_1 and X_2 into one, where it is assumed that the keys of the items in X_1 are smaller than the keys from X_2. With treap representations of the sets these operations can be performed easily via the insertion and deletion operations. In order to *split* a treap storing X according to some a, simply insert an item with key a and "infinite" priority. By the heap-order property the newly inserted item will be at the root of the new treap. By the in-order property the left subtree of the root will be a treap for X_1 and the right subtree will be a treap for X_2. In order to *join* the treaps of two sets X_1 and X_2 as above, simply create a dummy root whose left subtree is a treap for X_1 and whose right subtree is a treap for X_2, and perform a delete operation on the dummy root.

Recursive pseudocode implementations[4] of these elementary treap update algorithms are shown in Figure 2.

Sometimes "handles" or "fingers" are available that point to certain nodes in a treap. Such handles permit accelerated operations on treaps. For instance, if a handle on a node x is available, then deleting x reduces just to rotating it down into leaf position and clipping it; no search is necessary. Similarly the insertion of an item x into a treap can be accelerated if a handle to the successor (or predecessor) s of x in the resulting treap is known: start the search for the correct leaf position of x at node s instead of at the root of the treap. So-called "finger searches" are also possible where one is to locate a node y in a treap but the search starts at some (hopefully "nearby") node x that has a handle pointing to it; essentially one only needs to traverse the unique path between x and y. Finally splits and joins of treaps can be performed faster if handles to the minimum and maximum key items in the treaps are available. These operations are discussed in detail in sections 5.6 and 5.7.

[4]In practice it will be preferable to approach these operations the other way round. Joins and splits of treaps can be implemented as iterative top-down procedures; insertions and deletions can then be implemented as accesses followed by splits or joins. These implementations are operationally equivalent to the ones given here.

```
function Empty-Treap() : treap
    tnull→[priority,lchild,rchild]  ←  [−∞,tnull,tnull]
    return( tnull )


procedure Treap-Insert( (k,p) : item, T : treap )
    if T = tnull then T  ← newnode()
                      T→[key,priority,lchild,rchild]  ← [k,p,tnull,tnull]
    else if k < T→key then Treap-Insert( (k,p) , T→lchild )
                           if T→lchild →priority > T→priority then Rotate-Right( T )
    else if k > T→key then Treap-Insert( (k,p) , T→rchild )
                           if T→rchild →priority > T→priority then Rotate-Left( T )
    else (* key k already in treap T *)


procedure Treap-Delete ( k : key, T : treap )
    tnull→key  ← k
    Rec-Treap-Delete( k,T )
procedure Rec-Treap-Delete ( k : key, T : treap )
    if k <  T→key then Rec-Treap-Delete( k,T→lchild )
    else if k >  T→key then Rec-Treap-Delete( k,T→rchild )
    else Root-Delete( T )
procedure Root-Delete( T : treap )
    if Is-Leaf-or-Null( T ) then T ← tnull
    else if T→lchild→priority > T→rchild→priority then Rotate-Right( T )
                                                       Root-Delete( T→rchild )
                                                  else Rotate-Left( T )
                                                       Root-Delete( T→lchild )


procedure Treap-Split( T : treap, k : key, T1, T2 : treap )
    Treap-Insert( (k,∞), T )
    [T1,T2]  ←  T→[lchild,rchild]

procedure Treap-Join( T1,T2, T : treap )
    T  ← Newnode()
    T→[lchild,rchild]  ← [T1,T2]
    Root-Delete( T )


procedure Rotate-Left( T : treap )
    [T,T→rchild,T→rchild→lchild]  ← [T→rchild,T→rchild→lchild,T]
procedure Rotate-Right( T : treap )
    [T,T→lchild,T→lchild→rchild]  ← [T→lchild,T→lchild→rchild,T]
function Is-Leaf-or-Null( T : treap ) : Boolean
    return( T→lchild = T→rchild )
```

Figure 2.2: Simple routines for the elementary treap operations of creation, insertion, deletion, splitting, and joining. We assume call-by-reference semantics. A treap node has fields *key, priority, lchild, rchild.* The global variable *tnull* points to a sentinel node whose existence is assumed. $[\ldots] \leftarrow [\ldots]$ denotes parallel assignment.

3 Randomized Search Trees

Let X be a set of items, each of which is uniquely identified by a key that is drawn from a totally ordered set. We define a *randomized search tree* for X to be a treap for X where the priorities of the items are independent, identically distributed continuous random variables.

Theorem 3.1 *A randomized search tree storing n items has the expected performance characteristics listed in the table listed below:*

Performance measure	**Bound on expectation**
access time	$O(\log n)$
insertion time	$O(\log n)$
*insertion time for element with handle on predecessor or successor	$O(1)$
deletion time	$O(\log n)$
*deletion time for element with handle	$O(1)$
number of rotations per update	2
†time for finger search over distance d	$O(\log d)$
time for joining two trees of size m and n	$O(\log \max\{m,n\})$
time for splitting a tree into trees of size m and n	$O(\log \max\{m,n\})$
‡time for fast joining or fast splitting	$O(\log \min\{m,n\})$
update time if cost of rotating a subtree of size s is $O(s)$	$O(\log n)$
update time if cost of rotating a subtree of size s is $O(s \log^k s)$, $k \geq 0$	$O(\log^{k+1} n)$
update time if cost of rotating a subtree of size s is $O(s^a)$ with $a > 1$	$O(n^{a-1})$
update time if rotation cost is $f(s)$, with $f(s)$ non-decreasing	$O(\frac{f(n)}{n} + \sum_{0<i<n} \frac{f(i)}{i^2})$

*requires parent pointers
†requires parent pointers and some additional information, see section 5.4
‡requires some parent pointers, see section 5.6 and 5.7

Now let X be a set of items identifiable by keys drawn from a totally ordered universe. Moreover, assume that every item $x \in X$ has associated with it an integer weight $w(x) > 0$. We define the *weighted randomized search tree* for X as a treap for X where the priorities are independent continuous random variables defined as follows: Let F be a fixed continuous probability distribution. The priority of element x is the maximum of $w(x)$ independent random variables, each with distribution F.

Theorem 3.2 *Let T be a weighted randomized search tree for a set X of weighted items. The following table lists the expected performance characteristics of T. Here W denotes the sum of the weights of the items in X; for an item x, the predecessor and the successor (by key rank) in X are denoted by x^- and x^+, respectively; Tmin and Tmax denote the items of minimal and maximal key rank in X.*

Performance measure	**Bound on expectation**
time to acces item x	$O(1+\log\frac{W}{w(x)})$
time to insert item x	$O\left(1+\log\frac{W+w(x)}{\min\{w(x^-),w(x),w(x^+)\}}\right)$
time to delete item x	$O\left(1+\log\frac{W}{\min\{w(x^-),w(x),w(x^+)\}}\right)$
*insertion time for item x with handle on predecessor	$O(1+\log(1+\frac{w(x)}{w(x^-)}+\frac{w(x)}{w(x^+)}+\frac{w(x^-)}{w(x^+)}))$
*time to delete item x with handle	$O(1+\log(1+\frac{w(x)}{w(x^-)}+\frac{w(x)}{w(x^+)}))$
number of rotations per update on item x	$O(1+\log(1+\frac{w(x)}{w(x^-)}+\frac{w(x)}{w(x^+)}))$
†time for finger search between x and y, where V is total weight of items between and including x and y	$O(\log\frac{V}{\min\{w(x),w(y)\}})$
time to join trees T_1 and T_2 of weight W_1 and W_2	$O(1+\log\frac{W_1}{w(T_1max)}+\log\frac{W_2}{w(T_2min)})$
time to split T into trees T_1, T_2 of weight W_1, W_2	
‡time for fast joining or fast splitting	$O(1+\log\min\{\frac{W_1}{w(T_1max)},\frac{W_2}{w(T_2min)}\})$
*time for increasing the weight of item x by Δ	$O(1+\log\frac{w(x)+\Delta}{w(x)})$
*time for decreasing the weight of itme x by Δ	$O(1+\log\frac{w(x)}{w(x)-\Delta})$

*requires parent pointers
†requires parent pointers and some additional information, see section 5.4
‡requires some parent pointers, see section 5.6 and 5.7

Several remarks are in order. First of all, note that no assumptions are made about the key distribution in X. All expectations are with respect to randomness that is "controlled" by the update algorithms. For the time being we assume that the priorities are kept hidden from the "user." This is necessary to ensure that a randomized search tree is transformed into a randomized search tree by any update. If the user knew the actual priorities it would a simple matter to create a very "non-random" and unbalanced tree by a polynomial number of updates.

The requirement that the random variables used as priorities be continuous is not really necessary. We make this requirement only to ensure that with probability one all priorities are distinct. Our results continue to hold for i.i.d. random variables for which the probability that too many of them are equal is sufficiently small. This means that in practice using integer random numbers drawn uniformly from a sufficiently large range (such as 0 to 2^{31}) will be adequate for most applications.

Comparing with other balanced tree schemes some of which require only one bit of balance information per node, it might seem disappointing that randomized search trees require to store such "extensive" balance information at each node. However, it is possible to avoid this. In section 7 we discuss various strategies of maintaining randomized search trees that obviate the explicit storage of the random priorities.

Next note that weighted randomized search trees can be made to adapt naturally to observed access frequencies. Consider the following strategy: whenever an item x is accessed a new random number r is generated (according to distribution F); if r is bigger than the current priority of x, then make r the new priority of x, and, if necessary, rotate x up in the tree to reestablish the heap-property. After x has been accessed k times its priority will be the maximum of k i.i.d. random variables. Thus the expected depth of x in the tree will be $O(\log(1/p))$, where p is the access frequency of x, i.e. $p = k/A$, with A being the total number of accesses to the tree.

How would one insert an item x into a weighted randomized search tree with fixed weight k? This can most easily be done if the distribution function F is the identity, i.e. we start with random variables uniformly distributed in the interval $[0,1]$. The distribution function G_k for the maximum of k such random variables has the form $G_k(z) = z^k$. From this it follows that x should be inserted into the tree with priority $r^{1/k}$, where r is a random number chosen uniformly from the interval $[0,1]$. Since the only operations involving priorities are comparisons and the logarithm function is monotonic, storing $(\log r)/k$ is also possible.

Finally there is the question of how much "randomness" is required for our method. How many random bits does one need to implement randomized search trees? There is a rather simple minded strategy that shows that an expected *constant* number of random bits per update can suffice in the unweighted case. This can be achieved as follows: Let the priorities be real random numbers drawn uniformly from the interval $[0,1]$. Such numbers can be generated piece-meal by adding more and more random bits as digits to their binary representations. The idea is, of course, to generate only as much of the binary representation as needed. The only priority operation in the update algorithms are comparisons between priorities. In many cases the outcome of such a comparison will already be determined by the existing partial binary representations. When this is not the case, i.e. one representation happens to be a prefix of the other, one simply refines the representations by appending random bits in the obvious way. It is easy to see that the expected number of additional random bits needed to resolve the comparison is not greater than 4. Since in our update algorithms priority comparisons happen only in connection with rotations, and since the expected number of rotations per update is less than 2, it follows that the expected number of random bits needed is less than 12 for insertions and less than 8 for deletions.

Surprisingly, one can do much better than that. Only $O(\log n)$ truly random bits suffice overall. This can be achieved by letting a pseudo random number generator supply the priorities, where this generator needs $O(\log n)$ truly random seed bits. One possible such generator works as follows. Let $U \geq n^3$ be a prime number. Randomly choose 8 integers in the range between 0 and $U - 1$ and make them the coefficients of a degree-7 polynomial q (this requires the $O(\log n)$ random bits). As i-th priority produce $q(i) \bmod U$. Other generators are possible. The essential property they need to have is that they produce "random" numbers that are 8-wise independent. (Actually a somewhat weaker property suffices; see section 6.)

Theorem 3.3 *All the results of Theorem 3.1 continue to hold, up to constant factors, if the priorities are 8-wise independent random variables from a sufficiently large range, as for instance produced by the pseudo random number generator outlined above.*

In order to achieve logarithmic expected access, insertion, and deletion time, 5-wise independence suffices.

4 Analysis of Randomized Search Trees

In this section we analyze a number of interesting quanitities in randomized search trees and derive their expected values. These quantities are:

$D(x)$, the *depth* of node x in a tree, in other words the number of nodes on the path from x to the root;

$S(x)$, the *size* of the subtree rooted at node x, i.e. the number of nodes contained in that subtree;

$P(x,y)$, the length of the unique *path* between nodes x and y in the tree, i.e. the number of nodes on that path;

$SL(x)$ **and** $SR(x)$, the length of the right *S*pine of the *L*eft subtree of x and the length of the left *S*pine of the *R*ight subtree of x; by *left spine* of a tree we mean the root together with the left spine of the root's left subtree; the *right spine* is defined analogously.

Throughout this section we will deal with the treap T for a set $X = \{x_i = (k_i, p_i) | 1 \leq i \leq n\}$ of n items, numbered by increasing key rank, i.e. $k_i < k_{i+1}$ for $1 \leq i < n$. The priorities p_i will be random variables. Since for a fixed set of keys k_i the shape of the treap T for X depends on the priorities, the quantities of our interest will be random variables also, for which it makes sense to analyze expectations, distributions, etc.

Our analysis is greatly simplified by the fact that each of our quantities can be represented readily by two types of indicator random variables:

$A_{i,j}$ is 1 if x_i is an ancestor of x_j in T and 0 otherwise

$C_{i;\ell,m}$ is 1 if x_i is a common ancestor of x_ℓ and x_m in T and 0 otherwise

We consider each node an ancestor of itself. In particular we have:

Theorem 4.1 *Let* $1 \leq \ell, m \leq n$*, and let* $\ell < m$*.*

(i) $D(x_\ell) = \sum_{1 \leq i \leq n} A_{i,\ell}$

(ii) $S(x_\ell) = \sum_{1 \leq j \leq n} A_{\ell,j}$

(iii) $P(x_\ell, x_m) = 1 + \sum_{1 \leq i < m} (A_{i,\ell} - C_{i;\ell,m}) + \sum_{\ell < i \leq n} (A_{i,m} - C_{i;\ell,m})$

(iv) $SL(x_\ell) = \sum_{1 \leq i < \ell} (A_{i,\ell-1} - C_{i;\ell-1,\ell})$
$SR(x_\ell) = \sum_{\ell < i \leq n} (A_{i,\ell+1} - C_{i;\ell,\ell+1})$

Proof. (i) and (ii) are clear, since the depth of a node is nothing but the number of its ancestors, and the size of its subtree is nothing but the number of nodes for which it is ancestor.

For (iii) note that the path from x_ℓ to x_m is divided by their lowest common ancestor v into two parts. The part from x_ℓ to v, which consists exactly of the ancestors of x_ℓ minus the common ancestors of x_ℓ and x_m, is accounted for by the first sum. (This would be clear if the index range of that sum were between 1 and n. The smaller index range is justifed by the fact that for $\ell < m \leq i$ the node x_i is an ancestor of x_ℓ iff it is a common ancestor of x_ℓ and x_m, i.e. $A_{i,\ell} = C_{i;\ell,m}$ in that range.) Similarly the second sum accounts for the path between x_m and v. Since v is not counted in either of those two parts, the $+1$ correction factor is needed.

For (iv) it suffices to consider $SL(x_\ell)$, by symmetry. If x_ℓ has a left subtree, then the lowest node on its right spine must be $x_{\ell-1}$. It is clear that in that case the spine consists exactly of the ancestors of $x_{\ell-1}$ minus the ancestors of x_ℓ; but the latter are the common ancestor of $x_{\ell-1}$ and x_ℓ. Also, no x_i with $i \geq \ell$ can lie on that spine.

If x_ℓ has no left subtree, then either $\ell = 1$, in which case the formula correctly evaluates to 0, or $x_{\ell-1}$ is an ancestor of x_ℓ, in which case every ancestor of $x_{\ell-1}$ is a common ancestor of $x_{\ell-1}$ and x_ℓ, and the formula also correctly evaluates to 0. $\square$

If we let $a_{i,j} = \mathrm{Ex}[A_{i,j}]$ and $c_{i;\ell,m} = \mathrm{Ex}[C_{i;\ell,m}]$, then by the linearity of expectations we immediately get the following:

Corollary 4.2 *Let $1 \le \ell, m \le n$, and let $\ell < m$.*

(i) $\mathrm{Ex}[D(x_\ell)] = \sum_{1\le i\le n} a_{i,\ell}$

(ii) $\mathrm{Ex}[S(x_\ell)] = \sum_{1\le j\le n} a_{\ell,j}$

(iii) $\mathrm{Ex}[P(x_\ell, x_m)] = 1 + \sum_{1\le i<m}(a_{i,\ell} - c_{i;\ell,m}) + \sum_{\ell<i\le n}(a_{i,m} - c_{i;\ell,m})$

(iv) $\mathrm{Ex}[SL(x_\ell)] = \sum_{1\le i<\ell}(a_{i,\ell-1} - c_{i;\ell-1,\ell})$
$\mathrm{Ex}[SR(x_\ell)] = \sum_{\ell<i\le n}(a_{i,\ell+1} - c_{i;\ell,\ell+1})$

In essence our whole analysis has now been reduced to determining the expectations $a_{i,j}$ and $c_{i;\ell,m}$. Note that since since we are dealing with indicator 0-1 random variables, we have

$$a_{i,j} = \mathrm{Ex}[A_{i,j}] = \Pr[A_{i,j} = 1] = \Pr[x_i \text{ is ancestor of } x_j], \quad \text{and}$$

$$c_{i;\ell,m} = \mathrm{Ex}[C_{i;\ell,m}] = \Pr[C_{i;\ell,m} = 1] = \Pr[x_i \text{ is common ancestor of } x_\ell \text{ and } x_m].$$

Determining the probabilities and hence the expectations is made possible by the following *ancestor lemma*, which completely characterizes the ancestor relation in treaps.

Lemma 4.3 *Let T be the treap for X, and let $1 \le i, j \le n$.*
Then, assuming that all priorities are distinct, x_i is an ancestor of x_j in T iff among all x_h, with h between and including i and j, the item x_i has the largest priority.

Proof. Let x_m be the item with the largest priority in T, and let $X' = \{x_\nu | 1 \le \nu < m\}$ and $X'' = \{x_\mu | m < \mu \le n\}$. By the definition of a treap, x_m will be the root of T and its left and right subtrees will be treaps for X' and X'' respectively.

Clearly our ancestor characterization is correct for all pairs of nodes involving the root x_m. It is also correct for all pairs $x_\nu \in X'$ and $x_\mu \in X''$: they lie in different subtrees, and hence are not ancestor-related. But indeed the largest priority in the range between ν and μ is realized by x_m and not by x_ν or x_μ.

Finally, by induction (or recursion, if you will) the characterization is correct for all pairs of nodes in the treap for X' and for all pairs of nodes in the treap for X''. ◻

As an immediate consequence of this ancestor lemma we obtain an analogous *common ancestor lemma.*

Lemma 4.4 *Let T be the treap for X, and let $1 \le \ell, m, i \le n$ with $\ell < m$. Let p_ν denote the priority of item x_ν.*
Then, assuming that all priorities are distinct, x_i is a common ancestor of x_ℓ and x_m in T iff

$$\begin{array}{ll} p_i = \min\{p_\nu | i \le \nu \le m\} & \text{if } 1 \le i \le \ell \\ p_i = \min\{p_\nu | \ell \le \nu \le m\} & \text{if } \ell \le i \le m \\ p_i = \min\{p_\nu | \ell \le \nu \le i\} & \text{if } m \le i \le n, \end{array}$$

which can be summarized as

$$p_i = \min\{p_\nu |\ \min\{i, \ell, m\} \le \nu \le \max\{i, \ell.m\}\},$$

Now we have all tools ready to analyze our quantities of interest. We will deal with the case of unweighted and weighted randomized search trees separately.

4.1 The unweighted case

The last two lemmas make it easy to derive the ancestor probabilities $a_{i,j}$ and $c_{i;\ell,m}$.

Corollary 4.5 *In an (unweighted) randomized search tree x_i is an ancestor of x_j with probability $1/(|i-j|+1)$, in other word we have*

$$a_{i,j} = 1/(|i-j|+1)\,.$$

Proof. According to the ancestor lemma we need the priority of x_i to be the largest among the priorities of the $|i-j|+1$ items between x_i and x_j. Since in an unweighted randomized search tree these priorities are independent identically distributed continuous random variables this happens with probability $1/(|i-j|+1)$. ◻

Corollary 4.6 *Let $1 \le \ell < m \le n$.*
In the case of unweighted randomized search trees the expectation for common ancestorship is given by

$$c_{i;\ell,m} = \begin{cases} 1/(m-i+1) & \text{if } 1 \le i \le \ell \\ 1/(m-\ell+1) & \text{if } \ell \le i \le m \\ 1/(i-\ell+1) & \text{if } m \le i \le n, \end{cases}$$

which can be summarized as

$$c_{i;\ell,m} = 1/(\max\{i,\ell,m\} - \min\{i,\ell,m\} + 1)\,.$$

Proof. Analogous to the previous proof. ◻

Now we can put everything together to obtain exact expressions and closed form upper bounds for the expectations of the quantities of interest. The expressions involve harmonic numbers, defined as $H_n = \sum_{1\le i\le n} 1/i$. Note the standard bounds $\ln n < H_n < 1 + \ln n$ for $n > 1$.

Theorem 4.7 *Let $1 \le \ell, m \le n$, and let $\ell < m$. In an unweighted randomized search tree of n nodes the following expectations hold:*

(i) $$\begin{aligned} \mathrm{Ex}[D(x_\ell)] &= H_\ell + H_{n+1-\ell} - 1 \\ &< 1 + 2\cdot \ln n \end{aligned}$$

(ii) $$\begin{aligned} \mathrm{Ex}[S(x_\ell)] &= H_\ell + H_{n+1-\ell} - 1 \\ &< 1 + 2\cdot \ln n \end{aligned}$$

(iii) $$\begin{aligned} \mathrm{Ex}[P(x_\ell, x_m)] &= 4H_{m-\ell+1} - (H_m - H_\ell) - (H_{n+1-\ell} - H_{n+1-m}) - 3 \\ &< 1 + 4\cdot \ln(m-\ell+1) \end{aligned}$$

(iv) $$\begin{aligned} &\mathrm{Ex}[SL(x_\ell)] = 1 - 1/\ell \\ &\mathrm{Ex}[SR(x_\ell)] = 1 - 1/(n+1-\ell) \end{aligned}$$

Proof. Just use Corollary 4.2 and plug in the values from Corollaries 4.5 and 4.6. ◻

It is striking that in an unweighted randomized search tree the expected number of ancestors of a node x exactly equals the expected number of its descendants. However, it is reminiscent, although apparently unrelated, to the following easily provable fact: in any rooted binary tree T the average node depth equals the average size of a subtree.

We want to point out a major difference between the random variables $D(x_\ell)$ and $S(x_\ell)$. Although both have the same expectation, they have very different distributions. $D(x_\ell)$ is very tightly concentrated around its expectation, whereas $S(x_\ell)$ is not. For instance, one can easily see that in an n-node tree $\Pr[D(x_1) = n] = 1/n!$, whereas $\Pr[S(x_1) = n] = 1/n$. Using the ancestor lemma it is not too hard to derive the exact distribution of $S(x_\ell)$: one get $\Pr[S(x_\ell) = n] = 1/n$ and $\Pr[S(x_\ell) = s] = O(1/s^2)$ for $1 \le s < n$. We leave the details as an exercise to the reader. Here we briefly prove a concentration result for $D(x_\ell)$.

Lemma 4.8 *In an unweighed randomized search tree with $n > 1$ nodes we have for index $1 \le \ell \le n$ and any $c > 1$*

$$\Pr[D(x_\ell) \ge 1 + 2c \ln n] < 2(n/e)^{-c\ln(c/e)}\,.$$

Proof. Recall that $D(x_\ell) = \sum_{1 \le i \le n} A_{i,\ell}$, where $A_{i,\ell}$ indicates whether x_i is an ancestor of x_ℓ. Let $L = \sum_{1 \le i < \ell} A_{i,\ell}$ count the "left ancestors" of x_ℓ and let $R = \sum_{\ell < i \le n} A_{i,\ell}$ count the "right ancestors." Since $D(x_\ell) = 1 + L + R$, in order for $D(x_\ell)$ to exceed $1 + 2c \ln n$ at least one of L and R has to exceed $c \ln n$. It now suffices to prove that the probability of either of those events is bounded by $(n/e)^{-c\ln(c/e)}$. We will just consider the case of R. The other case is symmetric.

The crucial observation is that for $i > \ell$ the 0-1 random variables $A_{i,\ell}$ are independent and one can therefore apply so-called Chernoff bounds [12, 8] which, in one form, state that if random variable Z is the sum of independent 0-1 variables, then $\Pr[Z \ge c \cdot \mathrm{Ex}[Z]] < e^{-c\ln(c/e)\mathrm{Ex}[Z]}$.

In order to make the bound independent of ℓ we consider random variable $R' = \sum_{\ell < i < \ell+n} A_{i,\ell}$, where we use additional independent 0-1 variables $A_{i,\ell}$ for $i > n$, with $\mathrm{Ex}[A_{i,\ell}] = 1/(i - \ell + 1)$. Obviously for any $k > 0$ we have $\Pr[R \ge k] \le \Pr[R' \ge k]$. Using $\mathrm{Ex}[R'] = H_n - 1$, and using $\ln n - 1 < H_n - 1 < \ln n$ and applying the Chernoff bound we get the desired

$$\begin{aligned} \Pr[R \ge c \ln n] &\le \Pr[R' \ge c \ln n] < \Pr[R' \ge c(H_n - 1)] \\ &< e^{-c\ln(c/e)(H_n-1)} < e^{-c\ln(c/e)(\ln n - 1)} = (n/e)^{-c\ln(c/e)}\,. \end{aligned}$$

□

Note that this lemma implies that the probability of least one of the n nodes in an unweighted randomized search tree having depth greater than $2c \ln n$ is bounded by $n(n/e)^{-c\ln(c/e)}$. In other words, the probability that the height of a randomized search tree is more than logarithmic is exceedingly small, and hence the tree's expected height is logarithmic also. In contrast to the random variables studied in this section the random variable h_n, the height of an n-node unweighted randomized search tree, is quite difficult to analyze exactly. Devroye [9], though, has shown that $h_n / \ln n \to \gamma$ almost surely, as $n \to \infty$, where $\gamma = 4.31107...$ is the unique solution of $\gamma \ln(2e/\gamma) = 1$ with $\gamma \ge 2$.

4.2 The weighted case

Recall that in the weighted case every item x_i has associated with it a positive integer weight w_i, and the weighted randomized search tree for a set of items uses as priority for x_i the maximum of w_i independent continuous random variables, each with the same distribution.

For $i \le j$ let $w_{i:j}$ denote $\sum_{i \le h \le j} w_h$, and for $i > j$ define $w_{i:j} = w_{j:i}$. Let $W = w_{1:n}$ denote the total weight.

Corollary 4.9 *In an weighted randomized search tree x_i is an ancestor of x_j with probability $w_i / w_{i:j}$, in other word we have*

$$a_{i,j} = w_i / w_{i:j}\,.$$

Proof. According to the ancestor lemma we need the priority of x_i to be the largest among the priorities of the items between x_i and x_j. This means one of the w_i random variables "drawn" for x_i has to be the largest of the $w_{i:j}$ random variables "drawn" for the items between x_i and x_j. But since these random variables are identically distributed this happens with the indicated probability. □

Corollary 4.10 *Let $1 \leq \ell < m \leq n$.*
In the case of unweighted randomized search trees the expectation for common ancestorship is given by

$$c_{i;\ell,m} = \begin{cases} w_i/w_{i:m} & \text{if } 1 \leq i \leq \ell \\ w_i/w_{\ell:m} & \text{if } \ell \leq i \leq m \\ w_i/w_{\ell:i} & \text{if } m \leq i \leq n \end{cases}$$

Proof. Analogous to the previous proof, but based on the common ancestor lemma. □

Now we just need to plug our values into Corollary 4.2 to get the following:

Theorem 4.11 *Let $1 \leq \ell, m \leq n$, and let $\ell < m$. In an weighted randomized search tree with n nodes of total weight W the following expectations hold:*

$$\begin{array}{llll}
(i) & \mathrm{Ex}[D(x_\ell)] & = & \sum_{1\leq i\leq n} w_i/w_{i:\ell} \\
 & & < & 1 + 2\cdot\ln(W/w_\ell) \\
(ii) & \mathrm{Ex}[S(x_\ell)] & = & \sum_{1\leq i\leq n} w_\ell/w_{i:\ell} \\
(iii) & \mathrm{Ex}[P(x_\ell, x_m)] & = & 1 + \sum_{1\leq i<\ell}(w_i/w_{i:\ell} - w_i/w_{i:m}) \\
 & & & \quad + \sum_{\ell\leq i\leq m}(w_i/w_{\ell:i} + w_i/w_{i:m} - 2w_i/w_{\ell:m}) \\
 & & & \quad + \sum_{m<i\leq n}(w_i/w_{m:i} - w_i/w_{\ell:i}) \\
 & & < & 1 + 2\cdot\ln(w_{\ell:m}/w_\ell) + 2\cdot\ln(w_{\ell:m}/w_m) \\
(iv) & \mathrm{Ex}[SL(x_\ell)] & = & \sum_{1\leq i<\ell}\left(w_i/w_{i:\ell-1} - w_i/w_{i:\ell}\right) \\
 & & < & 1 + \ln(1 + w_\ell/w_{\ell-1}) \\
 & \mathrm{Ex}[SR(x_\ell)] & = & \sum_{\ell<i\leq n}\left(w_i/w_{\ell+1:i} - w_i/w_{\ell:i}\right) \\
 & & < & 1 + \ln(1 + w_\ell/w_{\ell+1})
\end{array}$$

Proof. The exact expressions follow from Corollaries 4.9 and 4.10.

The quoted upper bounds are consequences of the following two inequalities that can easily be verified considering the area underneath the curve $f(x) = 1/x$:

$$\alpha/A \leq \ln A - \ln(A-\alpha) \quad \text{for } 0 \leq \alpha < A \tag{1}$$

$$\alpha/A - \alpha/B \leq (\ln A - \ln(A-\alpha)) - (\ln B - \ln(B-\alpha)) \quad \text{for } 0 \leq \alpha < A \leq B \tag{2}$$

For instance, to prove (i) we can apply inequality (1) and use the principle of telescoping sums to derive

$$\sum_{1\leq i<\ell} w_i/w_{i:\ell} < \sum_{1\leq i<\ell} (\ln w_{i:\ell} - \ln w_{i+1:\ell}) = \ln w_{1:\ell} - \ln w_\ell = \ln(w_{1:\ell}/w_\ell) < \ln(W/w_\ell)$$

and analogously $\sum_{\ell<i\leq n} w_i/w_\ell < \ln(w_{\ell:n}/w_\ell) \leq \ln(W/w_\ell)$, which, adding in the 1 stemming from $i = \ell$, yields the stated bound.

Similarly we can use inequality (2) to derive (iv); we just show the case of $\mathrm{Ex}[SL(x_\ell)]$:

$$\begin{aligned}\textstyle\sum_{1\le i<\ell}\Big(w_i/w_{i:\ell-1} - w_i/w_{i:\ell}\Big) &< 1 + \textstyle\sum_{1\le i<\ell-1}\Big((\ln w_{i:\ell-1} - \ln w_{i+1:\ell-1}) - (\ln w_{i:\ell} - \ln w_{i+1:\ell})\Big)\\ &= 1 + (\ln w_{1:\ell-1} - \ln w_{\ell-1:\ell-1}) - (\ln w_{1:\ell} - \ln w_{\ell-1:\ell})\\ &< 1 + \ln w_{\ell-1:\ell} - \ln w_{\ell-1:\ell-1}\\ &= 1 + \ln(1 + w_\ell/w_{\ell-1})\end{aligned}$$

For proving (iii) one uses inequality (2) and telescoping to bound the first sum by $\ln(w_{\ell:m}/w_\ell)$ and the third sum by $\ln(w_{\ell:m}/w_m)$. The middle sum can be broken into three pieces. The third piece evaluates to -2, and using inequality (1) the first piece is bounded by $1+\ln(w_{\ell:m}/w_\ell)$ and the second piece by $1+\ln(w_{\ell:m}/w_m)$. Together this yields the indicated bound. $\square$

With sufficiently uneven weight assignments the bounds listed in this theorem can become arbitrarily bad, in particular they can exceed n, the number of nodes in the tree, which is certainly an upper bound for every one of the discussed quantities. One can obtain somewhat stronger bounds by optimizing over all possible weight assignments while leaving the total weight W and the weight of x_ℓ (and possibly x_m) fixed. Although this is possible in principle it seems technically quite challenging in general. We just illustrate the case $D(x_1)$.

Which choice of w_i maximizes $\mathrm{Ex}[D(x_1)] = \sum_{1\le i\le n} w_i/w_{1:i}$ while leaving w_1 and $W = w_{1:n}$ fixed? Rewrite the sum as

$$1 + \sum_{1<i\le n}\frac{w_{1:i} - w_{1:i-1}}{w_{1:i}} = 1 + \sum_{1<i\le n}(1 - w_{1:i-1}/w_{1:i}) = n - \sum_{1<i\le n} w_{1:i-1}/w_{1:i}\,.$$

A little bit of calculus shows that the last sum is minimized when all its summands are the same, which, using our boundary conditions, implies that each of them is $(w_1/W)^{1/(n-1)}$. Thus it follows that $S = \mathrm{Ex}[D(x_1)]$ is bounded by

$$n - (n-1)(w_1/W)^{1/(n-1)} = 1 + (n-1)\Big(1 - (w_1/W)^{1/(n-1)}\Big)\,.$$

which, however, is not a particularly illuminating expression, except that it is easily seen never to exceed n.

5 Analysis of Operations on Randomized Search Trees

In this section we discuss the various operations on unweighted and weighted randomized search trees and derive their expected efficiency, thus proving all bounds claimed in our Theorems 3.1 and 3.2.

5.1 Searches

A succesful search for item x in a treap T commences at the root of T and, using the key of x traces out the path from the root to x. Clearly the time taken by such a search is proportional to the length of the path or, in other words, the number of ancestors of x. Thus the expected time for a search for x is propotional to the expected number of its ancestors, which is $O(\log n)$ in the unweighted and $O(1+\log(W/w(x)))$ in the weighted case by Theorems 4.7 and 4.11.

An unsuccessful search for a key that falls between the keys of successive items x^- and x^+ takes expected time $O(\log n)$ in the unweighted and $O(\log(W/\min\{w(x^-), w(x^+)\}))$ in the weighted case, since such a search must terminate either at x^- or x^+ (ignoring the two cases where x^- or x^+ does not exist).

5.2 Insertions and Deletions

An item is inserted into a treap by first attaching it at the proper leaf position which is located by an (unsuccessful) search using the item's key; then the item is rotated up the treap, as long as it has a parent with smaller priority. Clearly the number of those rotations can be at most the length of the search path that was just traversed. Thus the insertion time is proportional to the time needed for an unsuccessful search, which in expectation is $O(\log n)$ in the unweighted and $O(\log(W/\min\{w(x^-), w(x^+)\}))$ in the weighted case, where x^- and x^+ are the predecessor and successor of x in the resulting tree.

The deletion operation requires essentially the same time since it basically reverses an insertion operation: first the desired item is located in the tree using its key; then it is rotated down until it is in leaf position, at which point it is clipped away; during those downward rotations the decision whether to rotate right or left is dictated by the priorities of the two current children, as the one with larger priority must be rotated up.

What is still interesting is the expected number of rotations that occur during an update. Since in terms of rotations a deletion is an exact reversal of an insertion it suffices to analyze the number of rotations that occur during a deletion.

So let x be an item in a treap T to be deleted from T. Although we cannot tell just from the tree structure of T which downwards rotations will be performed, we can tell how many there will be: their number is the sum of the lengths of the right spine of the left subtree of x and the left spine of the right subtree of x. The correctness of this fact can be seen by observing that a left-rotation of x has the effect of shortening the left spine of its right subtree by one; a right-rotation of x shortens the right spine of the left subtree.

Thus we know from Theorems 4.7 and 4.11 that in expectation the number of rotations per update is less than 2 in the unweighted case and less than $2+\ln(1+w(x)/w(x^-))+\ln(1+w(x)/w(x^+)) = O\Big(1 + \log(1 + w(x)/w(x^-) + w(x)/w(x^+))\Big)$ in the weighted case.

5.3 Insertion and Deletions using Handles

Having a handle, i.e. a pointer, to a node to be deleted from a treap of course obviates the need to search for that node. Only the downward rotations to leaf position, and the clipping away needs to be performed, which results in expected time bound of $O(1)$ in the unweighted and $O\Big(1 + \log(1 + w(x)/w(x^-) + w(x)/w(x^+))\Big)$ in the weighted case.

Similarly, if we know the leaf position at which a new node x is to be inserted (which is always either the resulting predecessor x^- or successor x^+ of x), then, after attaching the new leaf, only the upward rotations need to be performed, which also results in the expected time bounds just stated. If just a pointer to x^- is available, and x cannot be attached as a leaf to x^- since it has a nonempty right subtree, then the additional cost of locating x^+ is incurred. Note that in this case x^+ is the bottom element of the left spine of that right subtree of x^- and locating it amounts to traversing the spine. The expeced cost for this is therefore $O(1)$ in the unweighted and $O\Big(1 + \log(1 + w(x^-)/w(x^+))\Big)$, which results in a total expected cost of $O(1)$ and $O\Big(1 + \log(1 + w(x)/w(x^-) + w(x)/w(x^+) + w(x^-)/w(x^+))\Big)$ in the respective cases.

Note that in contrast to ordinary insertions, where the sequence of ancestors can be computed during the search for the leaf position, insertions using handles require that ancestor information is stored explicitly in the tree, i.e. parent pointers must be available, so that the upward rotations can be performed. Deletions based on handles also require parent pointers, since when the node x to be deleted is rotated down one needs to know its parent y so that the appropriate child pointer of y can be reset.

5.4 Finger searches

In a finger search we assume that we have a "finger" or "handle" or "pointer" on item x in a tree and using this information we quickly want to locate the item y with given key k. Without loss of generality we assume that $x = x_\ell$ and $y = x_m$ with $\ell < m$. The most natural way to perform such a search is to follow the unique path between x_ℓ and x_m in the tree. The expected length of this path is given in Theorems 4.7 and 4.11 for weighted and unweighted randomized search trees, which immediately would yield the bounds of Theorems 3.1 and 3.2. However, things are not quite that easy since it is not clear that one can traverse the path between x_ℓ and x_m in time proportional to its length. Such a traversal would, in general, entail going up the tree, starting at x_ℓ, until the lowest common ancestor u of x_ℓ and x_m is reached, and then going down to x_m. But how can one tell that u has been reached (For all we kmow, x_ℓ may be u already.)?

If one reaches during the ascent towards the root a node z whose key is larger than the "search key" k, then one has gone too far (similarly, if one reaches the root of the entire tree via its right child). The desired u was the last node on the traversed path that was reached via its left child, or, if no such node exists, the start node x. This *excess path* between u and z may be much longer than the path between x and y and traversing this path may dominate the running time of the finger search.

There are several ways of dealing with this excess path problem. One is to try to argue that in expectation such an excess path has only constant length. However, this turns out to be true for the unweighted case only. Other methods store extra information in the tree that either allows to shortcut such excess paths or obviates the need to traverse them.

Extra pointers

Let us first discuss a method of adding extra pointers. Define the *left parent* of v is the first node on the path from v to the root whose key is smaller than the key of v, and the *right parent* of v is the first node on the path with larger key. Put differently, a node is the right parent of all nodes on the right spine of its left subtree and the left parent of all nodes on the left spine of its right subtree. We store with every node v in the tree besides the two children pointers also two pointers set to the left parent and right parent of v, respectively, or to nil if such an parent does not exist. Such parent pointers have been used before in [2]. Please note that during a rotation or when adding or clipping a leaf these parent pointers can be maintained at constant cost. Thus no significant increase in update times is incurred.

A finger search now proceeds as follows: starting at x chase right parent pointers until either a nil pointer is reached or the next node has key larger than k (using a sentinel with key $+\infty$ obviates this case distinction). At this point one has reached the lowest common ancestor u of x and y, and using the key k one can find the path from u to y following children pointers in the usual fashion. Note that the number of links traversed is at most one more than the length of the path between x and y, and that it can be much smaller because of the shortcuts provided by the right parent pointers.

Extra keys

With every node u in the tree one stores min(u) and max(u), the smallest and largest key of any item in the subtree rooted at u. With this information one can tell in constant time whether an item can possibly be in the subtree rooted at u: its key has to lie in the range between min(u) and

$\max(u)$. The common ancestor of x_ℓ and x_m is then the first node z on the path from x_ℓ to the root with $\min(z) \le k \le \max(z)$.

Maintaining this min()/max() information clearly only takes constant time during rotations. However, when a leaf is added during an insertion or is clipped away at the end of a deletion the min()/max() information needs to be updated on an initial portion of the path toward the root. To be specific, consider the change when leaf x is removed. The nodes for which the min() information changes are exactly the nodes on the left spine of the right subtree of the *predecessor* of x. The max() information needs to be adjusted for all nodes on the right spine of the left subtree of the *successor* of x. The expected lengths of those spines are given in Theorems 4.7 and 4.11. They need to be added to the expected update times, which asymptotically is irrelevant in the case of unweighted randomized search trees, but can be the dominant factor in the case of weighted trees.

Finally, to make this argument applicable to the items with smallest and largest keys in the tree, one needs to maintain treaps with two sentinel nodes that are never removed: one with key $-\infty$ and one with key $+\infty$ (both with random priorities).

Relying on short excess paths

Finally let us consider the method, suggested to us by Mehlhorn and Raman [22], that traverses the excess path and relies on the fact that in expectation this path is short, at least for unweighted trees. (In the weighted case one can concoct examples where this expectation can become arbitrarily large.) The advantage of this method is that only the one usual parent pointer needs to be stored per node, and not two parent pointers. However, as we shall see soon, one also need to store one additional bit per node.

As before let u be the lowest common ancestor between $x = x_\ell$ and $y = x_m$ and let the right parent of u be z, the *endnode* of the excess path. Call the nodes on the path between, but not including u and z the *excess nodes.* Note that $u = x_\nu$ for some ν with $\ell \le \nu \le m$ (namely the one with largest priority in that range), and $z = x_j$ for some j with $m < j \le n$, and any excess node must be some x_i with $1 \le i < \ell$. We estimate the expected number of excess nodes by introducing for each i with $1 \le i < \ell$ a 0-1 random variable X_i indicating whether or not x_i is an excess node and summing the expectations. Now $\mathrm{Ex}[X_i]$ is the probability that x_i is an excess node, which we represent as the sum of the probabilities of the disjoint events E_{ij} that x_i is an excess node and z is x_j. Now, for E_{ij} to happen x_j must be the right parent of x_i and of u, or in other words, x_i and u must both be on the right spine of the left subtree of x_j. This happens if in the range between i and j, item x_j has the largest priority and x_i has the second largest, and if u has the largest priority in the range from ℓ to $j-1$. By the definition of u this last condition can be reformulated that the largest priority in the range ℓ to $j-1$ occur in the range ℓ to m. By the assumption that all priorities are independently, identically distributed, it follows that

$$\Pr[E_{ij}] = \frac{1}{(j-i+1)} \cdot \frac{1}{(j-i)} \cdot \frac{m-\ell+1}{(j-\ell)}$$

and the expected number of excess nodes is

$$\sum_{1\le i<\ell} \mathrm{Ex}[X_i] = \sum_{1\le i<\ell} \sum_{m<j\le n} \Pr[E_{ij}] = \sum_{1\le i<\ell} \sum_{m<j\le n} \frac{1}{(j-i+1)} \cdot \frac{1}{(j-i)} \cdot \frac{m-\ell+1}{(j-\ell)} .$$

Summing first over i and applying the bound $\sum_{a\le k<b} 1/k(k+1) < 1/a$ one can see that the double sum is upper bounded by

$$\sum_{m<j\le n} \frac{m-\ell+1}{(j-\ell)(j-\ell+1)} ,$$

which in turn by applying the same bound can be seen to be upper bounded by 1. Thus we have proved that in the case of unweighted trees the expected number of excess nodes is always less than one.

The above analysis relies on the existence of z, i.e. the lowest common ancestor u must have a right parent. This need not be the case: u can lie on the right spine of the entire tree. In this case a naive finger search would continue going up past u along the spine until the root is reached. This could be very wasteful. Consider for instance a finger search from x_{n-1} to x_n which with probability $1/2$ would this way traverse the entire spine, which has $O(\log n)$ expected length. To prevent this we require that every node has stored with it information indicating whether it is on the right or left spine of the entire tree. With this information a finger search can stop going up as soon as the right spine of the entire tree is reached, and thus in effect only the path between x and y is traversed.

5.5 Joins and splits

Two treaps T_1 and T_2 with all keys in T_1 smaller than all keys in T_2 can be joined by creating a new tree T whose root r has as left child T_1 and right child T_2, and then deleting r fromt T. Creating T takes constant time. Deleting r, as we observed before, takes the time proportional to the length of the right spine of T_1 plus the length of the left spine of T_2, i.e. the depth of the largest key item of T_1 plus the depth of the smallest key item of T_2. By Theorem 4.7 this is in expectation $O(\log m + \log n) = O(\log \max\{m, n\})$ if T_1 and T_2 are unweighted randomized search trees with m and n items, respectively. In the weighted case Theorem 4.11 implies an expected bound of $O(1 + \log \frac{W_1}{w(T_1max)} + \log \frac{W_2}{w(T_2min)})$, if we let W_i denote the total weight of T_i.

Splitting a treap T into a treap T_1 with all keys smaller than k and treap T_2 with all keys larger than k is achieved by essentially reversing the join operation: An item x with key k and infinite priority is inserted into T; because of its large priority the new item will end up as the root; the left and right subtrees of the root will be T_1 and T_2, respectively. This insertion works by first doing a search starting at the root to locate the correct leaf position for key k, adding item x as a leaf, and then rotating x up until it becomes the root. Note that the length of the search path is the same as the number of rotations performed, which in turn is the sum of the lengths of the right spine of T_1 and the left spine of T_2. Thus the time necessary to perform this split is proportional to the sum of the lengths of those two spines, and therefore the same expected time bounds hold as for joining T_1 and T_2.

5.6 Fast splits

How can one speed up the split operation described and analyzed above? One needs to shorten the search time for the correct leaf position and one needs to reduce the number of subsequent rotations. Let us assume the size (or total weight) of the eventual T_1 is small. Then it makes sense to determine the correct leaf position for x in T by a finger search starting at $Tmin$, the item in T with minimum key. Let z be the lowest common ancestor of $Tmin$ and the leaf added for x. Note that z is part of the path between $Tmin$ and x and that z must be on the left spine of T. Now, when rotating x up the tree one can stop as soon as x becomes part of the left spine of the current tree, or, in other words, as soon as z becomes the left child of x: The current left subtree of x will then contain exactly all items of T with key smaller than the splitting key k and forms the desired T_1. The tree T_2 is obtained by simply replacing the subtree rooted at x by the right subtree of x.

The time to do all this is proportional to the length L of the path in the original T between $Tmin$ and the leaf added for x: as argued before, the finger search takes this much time, and the number of subsequent rotations is exactly the number of edges on this path between x and z. In the unweighted case the expected value of L is by Theorem 4.7 $O(\log m)$ where m is the number of nodes that end up in T_1. This is not particularly good if T_1 is almost all of T. In this case it would be much better to proceed symmetrically: start the finger search at $Tmax$, the item in T with largest priority, and identify the path form $Tmax$ to x. The expectation of the length R of that path, and hence of the splitting time would then be $O(\log n)$, where n is the size of T_2. Of course in general one does not know in advance whether T_1 or T_2 is smaller. But one can circumvent this problem by starting both searches in lockstep in parallel, terminating both as soon as one succeeds. This would take $O(\min\{L, R\})$ time. And since $\mathrm{Ex}[\min\{L, R\}] \leq \min\{\mathrm{Ex}[L], \mathrm{Ex}[R]\}$, the total expected time for the fast split would be $O(\min\{\log m, \log n\}) = O(\log \min\{m, n\})$.

For the weighted case it now would seem to suffice to observe that by Theorem 4.11 the expectation of L is $O(1 + \log(W_1/w(T_1min) + W_1/w(T_1max)))$, where T_1max is the item with largest key in T_1, which is of course nothing but x^-, the predecessor of x in T. However, this is not quite true, since the new leaf x is not necessarily a child of x^-, but could be a child of x^+, the successor of x. In that case the path in T from x^- to x^+ (in other words the right spine of the left subtree of x^-) that needs to be traversed additionally after x^- has been reached could be quite long, namely $O(1 + \log(1 + w(x^-)/w(x^+))$. This additional traversal and cost can be avoided if one uses more space and stores with every item in the tree a predecessor and succesor pointer. (Note that those additional pointers could be maintained with constant overhead during all update operations.)

What kind of information needs to be maintained so that such a split procedure can be implemented? One needs $Tmin$ and $Tmax$; since the finger searches start at $Tmin$ and $Tmax$ and the upward parts of the searches only proceed along the spines of T, one needs to maintain parent pointers only for spine nodes and not for all nodes.

5.7 Fast joins

In a fast join of T_1 and T_2 one tries to reverse the operations of a fast split. Starting at T_1max and T_2min one traverses the right spine RS of T_1 and the left spine LS of T_2 in a merge like fashion until either a node x of RS is found whose priority is larger than the one of the root of T_2 or, symmetrically, a node y of LS is found whose priority exceeds the one of the root of T_1. In the first case the subtree T_x of T_1 rooted at x is replaced by the join of T_x and T_2, computed using the normal join algorithm, and the root of T_1 becomes the root of the result. In the other case one proceeds symmetrically and the root of T_2 becomes the root of the result.

This in effect reverses the operation of a fast split, except that no finger searches had to be made. Therefore the time required for fast joining T_1 and T_2 into T is upper bounded by the time necessary to fast split T into T_1 and T_2.

5.8 Expensive rotations

How expensive is it to maintain unweighted randomized search trees under insertions and deletions when the cost of a rotation is not constant but depends on the size of the subtree that is being rotated?

Since an insertion operation is just the reverse of a deletion operation it suffices to analyze just deletions. Recall that in order to delete a node x it first has to be located and then it has to be rotated down into leaf position, where it is then clipped away. Since the search time and the clipping away is unaffected by the cost of rotations we only need to analyze the expected cost $R(x)$ of rotating x into leaf position. This cost clearly depends on $f(s)$, the assumed cost of rotating a subtree of size s, but also on $S(x)$, the size of the subtree rooted at x, which of course is a random variable depending on the randomly chosen priorities.

Now let $F(m)$ denote the expected cost of rotating the root of an m node randomized search tree into leaf position, assuming rotating a subtree of size s costs $f(s)$. Then we have

$$\mathrm{Ex}[R(x)] = \sum_{1\le s\le n} \Pr[S(x)=s]\cdot F(s)\,.$$

It is not hard to show that $F(n) = O(f(n))$, provided $f(n)/n$ is non-decreasing, and that in an n-node tree $\Pr[S(x)=s] = O(1/s^2)$ for $1 \le s < n$ and $\Pr[S(x)=n] = 1/n$. From this one immediately gets

$$\mathrm{Ex}[R(x)] = O\Big(f(n)/n + \sum_{1\le s<n} f(s)/s^2\Big)$$

and the bounds of Theorem 3.1 follow immediately.

Instead of fleshing out the details we offer an entirely different proof that has the advantage of applying to a more general class of functions f, yielding somewhat more exact bounds, and also applying when the random priorities are not completely independent (see the following section).

First of all we will slightly change how the cost of a rotation is measured. If node x is rotated, then the cost will be $f(\ell)+f(r)$, where ℓ and r are the sizes of the left and right subtrees of x. This cost model is asymptotically indistinguishable from the previous one as long as there exists a constant c so that $f(\ell+r+1) \le c(f(\ell)+f(r))$ for all $\ell, r \ge 0$. This is the case for all non-decreasing functions that grow at most polynomially fast. We will distribute this cost of a rotation at x to the individual nodes of the subtrees as follows: a node y that differs in key rank from x by j is charged $\Delta f(j) = f(j) - f(j-1)$, with the convention that $f(0)=0$. Since the right subtree of x contains the first r nodes that succeed x in key rank, the charge distributed in the right subtree is thus $\sum_{1\le j\le r} \Delta f(j)$ which evaluates to $f(r)$ as desired. Symmetrically, the total charge in the left subtree adds up to $f(\ell)$.

Now let $x = x_k$ be the node to be deleted and let $y = x_{k+j}$ be some other node. The node y may participate in several rotations during the deletion of x. What are the roots $z = x_{k+i}$ of he (maximal) subtrees that are moved up during those rotations? It is not hard to see that before the deletion began both y and z must have been descendants of x and after completion of the deletion y must be a descendant of z. This charaterizes the z's exactly and corresponds to the following condition on the priorities: In the index range between the minimum and the maximum of $\{k, k+i, k+j\}$ the node $x = x_k$ must have the largest priority and $z = x_{k+i}$ must have the second largest. Note that with uniform and independent random priorities this condition hold with probability $1/s^{\overline{2}} = 1/s(s+1)$ if the size of the range is $s+1$.

If $D_{i,j}$ denotes the event that $y = x_{k+j}$ participated in a down rotation of $x = x_k$ against $z = x_{k+i}$, then the expected cost of the rotations $R_f(k,n)$ incurred when x_k is deleted from an n-node tree using cost function f can be written as

$$R_f(k,n) = \sum_{\substack{-k\le j\le n-k\\ j\ne 0}} \sum_{\substack{-k\le i\le n-k\\ i\ne 0}} \Pr[D_{i,j}]\Delta f(|j|)$$

Since $\Pr[D_{i,j}] = 1/(\max\{k+i, k+j, k\} - \min\{k+i,k+j,k\})^{\overline{2}}$, the inner sum evaluates for a fixed $j > 0$ to

$$\sum_{-k\le i<0} 1/(j-i)^{\overline{2}} + \sum_{0<i\le j} 1/j^{\overline{2}} + \sum_{j<i\le n-k} 1/i^{\overline{2}}\,,$$

which using $\sum_{a\le \nu<b} 1/\nu^{\overline{2}} = 1/a - 1/b$ evaluates to

$$\Big[1/(j+1) - 1/(j+k)\Big] + \Big[1/(j+1)\Big] + \Big[1/(j+1) - 1/(n-k+1)\Big] < 3/(j+1) < 3/j\,.$$

For $y = x_{k-j}$ a symmetric argument shows that the inner sum is also upper bounded by $3/j$. When f is non decreasing, i.e. Δf is non-negative, we therefore get

$$R_f(k,n) \le \sum_{1\le j<k} \frac{3\Delta f(j)}{j} + \sum_{1\le j<n+1-k} \frac{3\Delta f(j)}{j} < 6 \sum_{1\le j\le n} \frac{\Delta f(j)}{j} = 6\Big(\frac{f(n)}{n} + \sum_{1\le j<n} \frac{f(j)}{j^2}\Big)$$

from which again the bounds stated in Theorem 3.1 follow. Note that this method actually allows the *exact* computation of $R_f(k,n)$ for any arbitrary real valued function f. For instance with $f(x) = x \bmod 2$ one could determine the expected number of odd-sized subtrees that participate in rotations.

5.9 Changing weights

If the priority p of a single item x in a treap is changed to a new priority p', then the heap property of the treap can be reestablished by simply rotating x up or down the treap as is done during the insertion and deletion operations. The cost of this will be proportional to the number of rotations performed, which is $|D(x) - D'(x)|$, where $D(x)$ and $D'(x)$ are the depth of x in the old and new tree, respectively.

Now assume the weight w of an item x in a weighted randomized search tree of total weight W is changed to w' and after the required change in the random priorities the tree is restructured as outlined above. In the old tree x had expected depth $O(1 + \log(W/w))$, in the new tree it has expected depth $O(1 + \log(W'/w'))$, where $W' = W - w + w'$ is the total weight of the new tree. One is now tempted to claim that the expected depth difference and hence the expected number of rotations to achieve the change is $O(|\log(W/w) - \log(W'/w')|)$, which is about $O(|\log(w'/w)|)$ if the weight change is small relative to W. There are two problems with such a quick claim: (a) in general it is not true that $\mathrm{Ex}[|X - Y|] = |\mathrm{Ex}[X] - \mathrm{Ex}[Y]|$; (b) one cannot upper bound a difference $A - B$ using only upper bounds for A and B.

For the sake of definiteness let us deal with the case of a weight increase, i.e. $w' > w$. The case of a decrease can be dealt with analogously. Let us first address problem (a). In section 3 we briefly outlined how to realize weighted randomized search trees. As priority p for an item x of weight w use $u^{1/w}$ (or equivalently $(\log u)/w$), where u is a random number uniformly distributed in $[0,1]$. This simulates generating the maximum of w random numbers drawn independently and uniformly from $[0,1]$. If the new weight of x is w' and one chooses as new priority $p' = v^{1/w'}$, where v is a new random number drawn from $[0,1]$, then p' is not necessarily larger than p. This means that $D'(x)$, the new depth of x, could be larger than the old depth $D(x)$, inspite of the weight increase, which is expected to make the depth of x smaller. Thus, since the relative order of $D(x)$ and $D'(x)$ is unknown, $\mathrm{Ex}[|D(x) - D'(x)|]$ becomes difficult to evaluate.

This difficulty does not arise if one chooses as new priority $p' = u^{1/w'}$ (or equivalently $(\log u)/w'$), where u is the random number originally drawn from $[0,1]$. Note that although the random variable p and p' are highly dependent, each has the correct distribution, and this is all that is required. Since $w' > w$ we have $u^{1/w'} > u^{1/w}$, i.e. $p' > p$. Thus we have $D(x) \ge D'(x)$, and therefore $\mathrm{Ex}[|D(x) - D'(x)|] = \mathrm{Ex}[D(x) - D'(x)] = \mathrm{Ex}[D(x)] - \mathrm{Ex}[D'(x)]$.

Addressing problem (b) pointed out above, we can bound the difference of those two expectations using the fact that we know exact expressions for each of them. Assume that x has key rank ℓ, i.e. $x = x_\ell$, and let $\Delta = w' - w$ be the weight increase. From Theorem 4.11 we get (using the notation from there)

$$\mathrm{Ex}[D(x_\ell)] - \mathrm{Ex}[D'(x_\ell)] \;=\; \Big(1 + \sum_{\substack{1\le i\le n\\ i\ne\ell}} \frac{w_i}{w_{i:\ell}}\Big) - \Big(1 + \sum_{\substack{1\le i\le n\\ i\ne\ell}} \frac{w_i}{w_{i:\ell} + \Delta}\Big)$$

$$= \sum_{1\le i<\ell}\left(\frac{w_i}{w_{i:\ell}} - \frac{w_i}{w_{i:\ell}+\Delta}\right) + \sum_{\ell<i\le n}\left(\frac{w_i}{w_{\ell:i}} - \frac{w_i}{w_{\ell:i}+\Delta}\right).$$

Applying the methods used in the proof of part (iv) of Theorem 4.11 it is easy to show that each of the last two sums is bounded from above by $\ln\frac{w_\ell+\Delta}{w_\ell} = \ln\frac{w'}{w}$. From this bound on the expected difference of old and new depth of x it follows that the expected time to adjust the tree after the weight of x has been increased from w to w' is $O(1+\log(w'/w))$. Using the same methods one can show that decreasing w to w' can be dealt with in time $O(1+\log(w/w'))$.

When implementing the method outlined here, one needs to store for every item the priority implicitly in two pieces (w,u), where integer w is the weight and u is a random number from $[0,1]$. When two priorities (w,u) and $(\overline{w},\overline{u})$ are to be compared one has to compare $u^{1/w}$ with $\overline{u}^{1/\overline{w}}$. Alternatively one could store the pieces $(w,\log u)$ and use $(\log u)/w$ for the explicit comparison.

This raises the issue of the cost of arithmetic. We can postulate a model of computation where the evaluation of an expression like $u^{1/w}$ or $\log u$ takes constant time and thus dealing with priorities does not become a significant overhead to the tree operations. We would like to argue that such a model is not that unrealistic. This seems clear in practice, since there one would definitely use a floating point implementation. (This is not to say that weighted trees are necessarily practical.) From the theoretical point of view, the assumption of constant time evaluation of those functions is not that unrealistic since Brent [6] has shown that, when measured in the bit model, evaluating such functions up to a relative error of 2^{-m} is only slightly more costly than multyplying two m bit numbers.

Thus we assume a *word model* where each of the four basic arithmetic operations and evaluating functions such as $\log u$ using operands specified by logarithmically many bits costs constant time. It seems natural to assume here that "logarithmically" means $O(\log W)$. We now need to deal with one issue: it is not clear that a word size of $O(\log W)$ suffices to represent our random priorities so that their relative order can be determined.

Here is a simple argument why $O(\log W)$ bits per word should suffice for our purposes. Following the definition of a weighted randomized search tree the priorities of an n node tree of total weight W can be viewed as follows: W random numbers are drawn independently and uniformly from the interval $[0,1]$ and certain n of those chosen numbers are selected to be the priorities. Now basic arguments show that with probability at most $1/W^{k-2}$ the difference of any two of the W chosen numbers is smaller than $1/W^k$. This means that with probability at least $1-1/W^{k-2}$ all comparisons between priorities can be resolved if the numbers are represented using more than $k\log_2 W$ bits, i.e. a word size of $O(\log W)$ suffices with high probability.

6 Limited Randomness

The analyses of the previous sections crucially relied on the availability of perfect randomness and on the complete independence of the random priorities. In this section we briefly discuss how one can do with much less, thus proving Theorem 3.3. We will show that for unweighted trees all the asymptotic results about expectations proved in the previous sections continue to hold, if the priorities in the tree are integer random variables that are of only limited independence and are uniformly distributed over a sufficiently large range. In particular we will show that 8-wise independence and range size $U \ge n^3$ suffice. The standard example of a family of random variables satisfying these properties is $X_i = q(i) \bmod U$ for $1 \le i \le n$, where $U > n^3$ is a prime number and q is a degree 7 polynomial whose coefficients are drawn uniformly and independently from $\{0,\ldots,U-1\}$. Thus q acts as a pseudo random number generator that needs $O(\log n)$ truely random bits as seeds to specify its eight coefficients.

It is quite easy to see why one would want $U \geq n^3$. Ideally all priorites occuring in a randomized search tree should be distinct. Our algorithms on treaps can easily be made to handle the case of equal priorities. However for the analysis and for providing guarantees on the expectations it is preferably that all priorities be distinct. Because of the pairwise independence implied by 8-wise independence, for any two distinct random variables X_i, X_j we have $\Pr[X_i = X_j] = 1/U$. Thus the probability that any two of the n random variables happen to agree is upper bounded by $\binom{n}{2}/U$ and, as the birthday paradox illustrates, not much smaller than that. With $U \geq n^3$ the probability of some two priorities agreeing thus becomes less than $1/n$. We can now safely ignore the case of agreeing priorities since in that event we could even afford to rebuild the entire tree which would incur only $O(\log n)$ expected cost.

Why 8-wise independence? Let $\mathcal{X}$ be a finite set of random variables and let d be some integer constant. We say that $\mathcal{X}$ has the *d-max property* iff there is some constant c so that for any enumeration $X_1, X_2, \ldots X_m$ of the elements of any subset of $\mathcal{X}$ we have

$$\Pr[X_1 > X_2 > \cdots > X_d > \{X_{d+1}, X_{d+2}, \ldots, X_m\}] \leq c/m^{\underline{d}}.$$

Note that identically distributed independent continous random variables have the d-max property for any d with a constant $c = 1$.

It turns out that all our results about expected properties of randomized search trees and of their update operations can be proved by relying only on the 2-max property of the random priorities. Moreover, the 2-max property is implied by 8-wise independence because of the following remarkable lemma that is essentially due to Mulmuley [15].

Lemma 6.1 *Let $\mathcal{X}$ be a set of n random variables, each uniformly distributed over a common integer range of size at least n.*
$\mathcal{X}$ has the d-max property if its random variables are $(3d+2)$-wise independent.

A proof of this lemma (or rather, of a related version) can be found in Mulmuley's book [15, section 10.1].

We now need to show that results of section 4 about unweighted trees continue to hold up to a constant factor if the random priorities satisfy the 2-max property. This is clear for the central Corollaries 4.5 and 4.6. As a consequence of the ancestor and common ancestor lemma they essentially just give the probability that a certain one in a set of priorities achieves the maximum. For those two corollaries the 1-max property would actually suffice. From the continued validity of Corollary 4.5 the asymptotic versions of points (i) and (ii) of Theorem 4.7 about expected depth of a node and size of its subtree follow immediately, also relying only on the 1-max property. Note that this means that if one is only interested in a basic version of randomized search trees where the expected search and update times are logarithmic (although more than an expected constant number of rotations may occur per update), then 5-wise independence of the random priorities provably suffices.

Points (iii) and (iv) of Theroem 4.7 do not follow immediately. They consider expectations of random variables that were expressed in Theorem 4.1 as the sum of differences of indicator variables $(A_{i,\ell} - C_{i;\ell,m})$ and upper bounds for the expectations of ($A_{i,\ell}$ and $C_{i;\ell,m}$) yield no upper bound for the expectation of their difference. Now $(A_{i,\ell} - C_{i;\ell,m})$ really indicates the event $E_{i;\ell,m}$ that x_i is an ancestor of x_ℓ but not an ancestor of x_m. We need to show that if the priorities have the 2-max property, $\Pr[E_{i;\ell,m}]$ is essentially the same as if the priorities were completely independent.

Without loss of generality let us assume $\ell < m$. In the case $i \leq \ell$ the event $E_{i;\ell,m}$ happens exactly when the following constellation occurs among the priorities: x_i has the largest priority among the items with index between and including i and ℓ, but not the largest priority among the items with index between i and m. For $\ell < i < m$ event $E_{i;\ell,m}$ occurs iff x_i has the largest priority among the items with indices in the range between ℓ and i, but not the largest in the range ℓ to m. (For the case $i > m$ the event is empty.)

Thus in both cases we are dealing with an event $E_{X,\mathcal{Y},\mathcal{Z}}$ of the following form: For a set $\mathcal{Z}$ of random variables and $X \in \mathcal{Y} \subset \mathcal{Z}$ the random variable X is largest among the ones in $\mathcal{Y}$ but not the largest among the ones in $\mathcal{Z}$. In the case of identically distributed, independent random variables clearly we get $\Pr[E_{X,\mathcal{Y},\mathcal{Z}}] = 1/|\mathcal{Y}| - 1/|\mathcal{Z}|$. The following claim shows that essentially the same is true if $\mathcal{Z}$ has the 2-max property.

Claim 1 *If $\mathcal{Z}$ has the 2-max property, then* $\Pr[E_{X,\mathcal{Y},\mathcal{Z}}] = O(1/|\mathcal{Y}| - 1/|\mathcal{Z}|)$.

Proof. Let $a = |\mathcal{Y}|$ and $b = |\mathcal{Z}|$ and let $X_1, X_2, \ldots, X_b$ be an enumeration of $\mathcal{Z}$ so that $X_1 = X$ and $\mathcal{Y} = \{X_1, \ldots, X_a\}$. For $a < i \le b$ let F_i denote the event that X_i is largest among $\{X_1, \ldots, X_i\}$ and X_1 is second largest. Because of the 2-max property of $\mathcal{Z}$ we have $\Pr[F_i] = O(1/i^2)$. Since $E_{X,\mathcal{Y},\mathcal{Z}} = \bigcup_{a<i\le b} F_i$ we therefore get

$$\Pr[E_{X,\mathcal{Y},\mathcal{Z}}] \le \sum_{a<i\le b} \Pr[F_i] = O\Big(\sum_{a<i\le b} 1/i^2\Big) = O(1/a - 1/b)\,,$$

as desired. $\square$

Thus points (iii) and (iv) of Theorem 4.7 hold, up to constant factors, if priorities have the 2-max property. This immediately means that all results listed in Theorem 3.1, except for the ones on expensive rotations, continue to hold, up to constant factors, if random priorities satisfying the 2-max property are used. The results on expensive rotations also continue to hold since their proof in section 5.8 quite explicitly only relies on the 2-max property of the random priorities.

The only result for unweighted trees that seems to require something more than the 2-max property is the one on short excess paths for finger searches in section 5.4. This is not too serious since other methods for implementing finger searches are available. We leave it as an open problem to determine weak conditions on priorities under which excess paths remain short in expectation.

7 Implicit Priorities

In this section we show that it is possible to implement unweighted randomized search trees so that no priorities are stored explicitly. We offer three different methods. One uses hash functions to generate or regenerate priorities on demand. The other stores the nodes of the tree in a random permutation and uses node addresses as priorities. The last method recomputes priorities from subtree sizes.

7.1 Priorities from hash functions

This method is based on an initial idea of Danny Sleator [24]. He suggested to choose and associate with a randomized search tree a hash function h. For every item in the tree the priority is then declared to be $h(k)$, where k is the key of the item. This priority need not be stored since it can be recomputed whenever it is needed. The hope is, of course, that a good hash function is "random enough" so that the generated priorities behave like random numbers.

Initially it was not clear what sort of hash function would actually exhibit enough random behaviour. However, the results of the previous section show that choosing for h the randomly selected degree-7 polynomial q mentioned at the beginning of the previous section does the trick.

If one is only interested in normal search and update times, then a randomly selected degree-4 polynomial suffices, as discussed in the previous section. In order to make this scheme applicable for any sort of key type we apply q not to the key but to the address of the node where the respective item is stored.

One may argue that from a practical point of view it is too expensive to evaluate a degree-7 polynomial whenever a priority needs to be looked at. Note, however, that priorities are compared only during updates, and that priority comparisons are coupled with rotations. This means that the expected number of priorities looked at during a deletion is less than 3 and during an insertion it is less than 4.

7.2 Locations as priorities

Here we store the nodes of the tree in an array $L[]$ in random order. Now the addresses of the nodes can serve as priorities, i.e. the node $L[i]$ has priority i. We will assume here that the underlying treap has the min-heap property, and not the max-heap property. Thus $L[1]$ will the root of the tree.

How does one insert into or delete from an n-node tree stored in this fashion? Basically one needs to update a random permutation. In order to insert an item x some i with $1 \leq i \leq n+1$ is chosen uniformly at random. If $i = n+1$ then x is stored in location $L[n+1]$, i.e. it is assigned priority $n+1$, and it is then inserted in the treap. If $i \leq n$ the node stored at $L[i]$ is moved to $L[n+1]$, i.e. its priority is changed to $n+1$. This means in the tree it has to be rotated down into leaf position. The new item x is placed into location $L[i]$ and it is inserted in the treap with priority i.

When item $x = L[i]$ is to be removed from the tree, it is first deleted in the usual fashion. Since this vacates location $L[i]$ the node stored at $L[n]$ is moved there. This means its priority was changed from n to i and it has to be rotated up the tree accordingly.

This scheme is relatively simple, but it does have some drawbacks. Per update some extra node changes location and has to be rotated up or down the tree. This is not a problem timewise since the expected number of those rotations is constant. However, changing the location of a node y means that one needs to access its parent so that the relevant child pointer can be reset. For accessing the parent one either needs to maintain explicit parent pointers, which is costly in space or one needs to find it via a search for y, which is costly in time. Explicit parent pointers definitely are necessary if a family of trees is to be maintained under joins and splits. One can keep the nodes of all the trees in one array. However, when an extra node is moved during an update, one does not know which tree it belongs to and hence one cannot find its parent via search. Also note the book keeping problem when the root of a tree is moved.

Finally, there is the question what size the array $L[]$ should have. This is no problem if the maximum size of the tree is known a priori. If this is not the case, one can adjust the size dynamically by, say, doubling it whenever the array becomes filled, and halving it whenever it is only 1/3 full. With a strategy of this sort the copying cost incurred through the size changes is easy to be seen constant in the amortized sense.

7.3 Computing priorities from subtree sizes

This method was suggested by Bent and Driscoll [5]. It assumes that for every node x in the tree the size $S(x)$ of its subtree is known. In a number of applications of search trees this information is stored with every node in any case.

During the deletion of a node y for every down rotation one needs to decide whether to rotate left or right. This decision is normally dictated by the priorities of the two children x and z of y: the one with larger priority is rotated up. The priority of x is the largest of the $S(x)$ priorities stored in its subtree. The priority of z is the largest of the $S(z)$ priorities in its subtree. Thus the probability that the priority of x is larger than the priority of z is $p = S(x)/(S(x) + S(z))$. This means that p should also be the probability that x is rotated up. Thus the decision which way to rotate x can be probabilistically correctly simulated by flipping a coin with bias $S(x)/(S(x)+S(z))$. It is amusing that one can actually do this without storing the sizes. Before the rotation one could determine $S(x)$ and $S(z)$ in linear time by traversing the two subtrees. This would make the cost of the rotation linear in the size of the subtree rotated, and our results about costly rotations imply that the expected deletion time is still logarithmic.

Unfortunately this trick does not work for insertions. How does one perform them? Note that when a node x is to be inserted into a a tree rooted at y it becomes the root of the new tree with probability $1/(S(y) + 1)$. This suggests the following strategy: Flip a coin with bias $1/(S(y) + 1)$. In case of success insert x into the tree by finding the correct leaf position and rotating it all the way back up to the root. In case of failure apply this strategy recursively to the appropriate child of y.

We leave the implementation of joins and splits via this method as an exercise.

8 Randomized search trees and Skip lists

Skip lists are a probabilistic search structure that were proposed and popularized by Pugh [23]. They can be viewed as a hierarchy of coarser and coarser lists constructed over an initial linked list, with a coarser list in the hierarchy guiding the search in the next finer list in the hierarchy. Which list items are used in the coarses lists is determined via random choice. Alternatively, skip lists can also be viewed as a randomized version of (a, b)-trees.

The performance characteristics of skip lists are virtually identical to the ones of unweighted randomized search trees. The bounds listed in Theorem 3.1 all hold if the notion of rotation is changed to the notion of pointer change. The only possible exception are the results about costly rotations: here, it seems, only partial results are known for skip lists (see [15, section 8.1.1]).

Just as with randomized search trees it is possible to generalize skip lists to a weighted version [19]. This is done by appropriately biasing the random choice that determines how far up the hierachy a list item is to go. Apparently again the expected performance characteristics of weighted skip lists match the ones of weighted randomized search trees listed in Theorem 3.2.

Apparently it is not known whether an analogue of Theorem 3.3 about limited randomness also holds for skip lists.

Comparing skip lists and randomized search trees seems a fruitless exercise. They are both conceptually reasonably simple and both are reasonably simply to implement. Both have been implemented and are for instance availabe as part of LEDA [20, 21]. They seem to be almost identical in their important performance characteristics. Differences such as randomized search trees can be implemented using only exactly n pointers, whereas this appears to be impossible for skip lists, are not of particularly great practical significance.

There seems to be ample room for both skip lists and randomized search trees. We would like to refer the reader to chapter 1 of Mulmuley's book [15], where the two structures are used and discussed as two prototypical randomization strategies.

References

[1] G.M. Adel'son-Velskii and Y.M. Landis, An algorithm for the organization of information. *Soviet Math. Dokl. 3* (1962) 1259–1262.

[2] H. Baumgarten, H. Jung, and K. Mehlhorn, Dynamic point location in general subdivision. *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms (SODA)* (1992) 250–258.

[3] R. Bayer and E. McCreight, Organization and maintenance of large ordered indices. *Act. Inf. 1* (1972) 173–189.

[4] S.W. Bent, D.D. Sleator, and R.E. Tarjan, Biased search trees. *SIAM J. Comput. 14* (1985) 545–568.

[5] S.W. Bent and J.R. Driscoll, Randomly balanced search trees. *Manuscript* (1991).

[6] R.P. Brent, Fast Multiple Precision Evaluation of Elementary Functions. *J. of the ACM 23* (1976) 242–251.

[7] M. Brown, Addendum to "A Storage Scheme for Height-Balanced Trees." *Inf. Proc. Letters 8* (1979).

[8] K.L. Clarkson, K. Mehlhorn, and R. Seidel, Four results on randomized incremental construction. *Comp. Geometry: Theory and Applications 3* (1993) 185–212.

[9] L. Devroye, A note on the height of binary search trees. *J. of the ACM 33* (1986) 489–498.

[10] I. Galperin and R.L. Rivest, Scapegoat Trees. *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms (SODA)* (1993) 165–174.

[11] L.J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees. *Proc. 19th FOCS* (1978) 8–21.

[12] T. Hagerup and C. Rüb, A guided tour of Chernoff bounds. *Inf. Proc. Letters 33* (1989/90) 305–308.

[13] J. Nievergelt and E.M. Reingold, Binary search trees of bounded balance. *SIAM J. Comput. 2* (1973) 33–43.

[14] K. Mulmuley, A fast planar partition algorithm, I. *Proc. 29th FOCS* (1988) 580–589.

[15] K. Mulmuley, **Computational Geometry: An Introduction through Randomized Algorithms.** Prentice Hall (1994).

[16] E. McCreight, Priority search trees. *SIAM J. Comput. 14* (1985) 257–276.

[17] K. Mehlhorn, **Sorting and Searching.** Springer (1984).

[18] K. Mehlhorn, **Multi-dimensional Searching and Computational Geometry.** Springer (1984).

[19] K. Mehlhorn (private communication).

[20] K. Mehlhorn and S. Näher, LEDA, a Platform for Combinatorial and Geometric Computing. To appear in *Commun. ACM*, January 1995.

[21] S. Näher, LEDA User Manual Version 3.0. Tech. Report MPI-I-93-109, Max-Planck-Institut für Informatik, Saarbrücken (1993).

[22] K. Mehlhorn and R. Raman (private communication).

[23] W. Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM 33* (1990) 668–676.

[24] D.D. Sleator (private communication).

[25] D.D. Sleator and R.E. Tarjan, Self-adjusting binary search trees. *J. of the ACM 32* (1985) 652–686.

[26] J. Vuillemin, A Unifying Look at Data Structures. *Commun. ACM 23* (1980) 229–239.