A Constant Update Time Finger Search Tree

Rajeev Raman
Paul Dietz[1]

Department of Computer Science
University of Rochester
Rochester, NY 14627

December 1989

[1]This work was supported in part by the National Science Foundation under grant CCR-8909667

Abstract

Levcopolous and Overmars [LO88] described a search tree in which the time to insert or delete a key was $O(1)$ once the position of the key to be inserted or deleted was known. Their data structure did not support *fingers*, pointers to points of high access or update activity in the set such that access and update operations in the vicinity of a finger are particularly efficient [GMPR77, BT80, Kos81, HM82, Tsa85]. Levcopolous and Overmars left open the question of whether a data structure could be designed which allowed updates in constant time and supported fingers. We answer the question in the affirmative by giving an algorithm in the RAM with logarithmic word size.

CR Classification Number: [F.2.2 - Sorting and Searching]

Keywords: Real-Time Algorithm, Search Tree, Fingers.

1 Introduction

Define the *update time* of a search tree to be the cost of an insertion/deletion once the position of the key to be inserted or deleted is known, *i.e.*, the update time is how long it takes to insert/delete a key given a pointer to it. Levcopolous and Overmars [LO88] describe a search tree with constant worst-case update time. However, their data structure does not support *fingers*, which are pointers to particular keys in the sorted set. A *finger search tree* supports the following operations:

- *search(x,f)* : search for key x starting at finger f,
- *insert/delete(x,f)* : insert/delete key x starting at finger f,
- *create(x)* : create a new finger at key x,
- *destroy(f)* : destroy a finger f.

The operations *search*, *insert* and *delete* should take time $O(\log d)$, where d is the distance of x from f, and *create* and *destroy* should take $O(1)$ time. A data structure that supports finger searches and finger creation/deletion and has worst-case constant update time is superior to an ordinary finger tree, since it can clearly handle the full repertoire of finger operations, while a finger tree may not be able to guarantee constant update time. A good worst-case bound on update time also implies that the number of storage modifications made with each update to the data structure is small, which helps in making the data structure *persistent* [DSST89], *i.e.*, to permit access and update operations on old versions of the data structure. In this paper we show good worst-case bounds on the update time of finger search trees.

In [GMPR77, Kos81, Tsa85] finger tree schemes are described which have worst-case $O(\log d)$[1] search and update time (finger creation and deletion take $O(\log d)$ time in the latter two approaches). In [HM82] it is shown that 2-4 trees have constant amortized update time. By using links between all nodes at the same level (the so-called *level-links*[BT80]), a key can be found in $O(\log d)$ time, and thus the full set of finger operations can be realized in an amortized sense by level-linked 2-4 trees. Moreover, moving or creating a finger takes only constant time once the position is known, and deletion of fingers takes constant time. The best worst-case update time bound in a finger tree is by Harel [Har80, HL79], where a finger tree is described that allows updates in worst-case $O(\log^* n)$ time. In [DSST89] it is shown that red-black trees can be maintained with $O(1)$ worst-case modifications per update, with the consequence that finger search trees can be made persistent efficiently. The update time in this data structure is $O(\log n)$, however.

In the following section we turn to the problem of constructing a data structure that can support fingers, and still perform updates in constant worst-case time. We construct such a data structure that runs on a RAM with logarithmic word size. The algorithm has very simple implementations of finger searches, creation, movement and deletion, allows finger deletion in constant time, and finger creation and movement in constant time if the position is known.

[1] All logarithms in this paper are to the base 2 unless otherwise stated.

2 The Existing Algorithm

It is easy to devise a tree data structure for the sorted set problem that has $O(1)$ amortized time updates and has logarithmic search time [Ove82]. Take any data structure that can perform updates in time $O(\log n)$. Partition the elements of the set into contiguous buckets of size about $O(\log n)$ that are represented by a linear list, and let each bucket have a representative that is stored in the data structure. When a bucket doubles in size or reduces to half its size, we spend $O(\log n)$ time splitting it into two (or merging it with one of its neighbors) and a further $O(\log n)$ time inserting the new representative into (or deleting the old representative from) the top-level data structure. This is $O(1)$ work when amortized over the $O(\log n)$ updates that caused the bucket to change in size.

Levcopolous and Overmars [LO88] describe a worst-case analog of this method (a nearly identical approach was taken in [DS87] to solve a different problem). Their solution is to perform both the bucket splitting and the insertion of representative into the top level tree (which must now be able to support updates in *worst-case* $O(\log n)$ time) incrementally. They show how, given a data structure on N keys that is "freshly built", it is possible to perform only insertions in constant time while the size of the data structure is in the range $[N/2, 2N]$ (their approach toward handling deletions is described later). We will assume that when "freshly built" all buckets have size at most $\frac{1}{2}\log^2 N$. As updates progress, the bucket of largest size is repeatedly picked, and over the next $O(\log N)$ insertions into the data structure, this bucket is split and the new representative is inserted into the data structure. Since $O(\log N)$ insertions are done to the buckets between splits, it is possible that the buckets get too large to split concurrently with the top-level insertion. As a consequence of a combinatorial lemma proved in [LO88] picking the largest bucket each time guarantees that no bucket will have size more than $O(\log^2 N)$.

In [DS87] a slightly stronger version of the same lemma is proved, and is formulated as the following game on an array of size n:

Lemma 1 *(Theorem 5 in [DS87]) Let $x_1, \ldots, x_n$ be n real valued variables, all initially zero. Repeatedly perform the following moves:*

1. *Choose an x_i such that $x_i = max_j\{x_j\}$, and set it to zero.*
2. *Choose n nonnegative reals $a_1, \ldots, a_n$ such that $\sum_{i=1}^{n} a_i = 1$, and for all i, $1 \leq i \leq n$, set x_i to $x_i + a_i$.*

Then each x_i will always be less than $H_{n-1} + 1$.

Here $H_k = \sum_{i=1}^{k} 1/i$ is the kth harmonic number (it can be verified that $H_k - 1 < \ln k < H_k - 1/k$).

In the case above, $k = c \log N$ insertions occur between bucket splitting operations. Initially the data structure on N keys is constructed so that all the buckets have size less than $\frac{1}{2}\log^2 N$. Since the data structure will be maintained only while the size is less than $2N$, there will be at most $2N$ buckets. Now we apply lemma 1 to show that the buckets will never have more than $\log^2 N$ elements in them. Let the excess of the bucket size over

$\frac{1}{2}\log^2 N$ (if there is no excess we define the excess to be 0) be represented by the variables x_i. Splitting the bucket will reduce the size of any bucket to below $\frac{1}{2}\log^2 N$ if no bucket ever exceeds size $\log^2 N$. By lemma 1 if $c \leq 0.5/\ln 2$, the x_i's will never grow beyond $k(H_{2N-1}+1) \leq \frac{1}{2}\log^2 N$. Each bucket of size $O(\log^2 N)$ is represented by a 2-level tree with a branching factor of $O(\log N)$. Insertions necessitate controlling the branching factor at the internal nodes, which can be done in $O(1)$ time incrementally. Splitting a bucket takes $O(\log N)$ time (to partition the $O(\log N)$ children of the root) and can be done concurrently with the insertion of its representative being performed in the top level data structure.

To make the data structure valid for all values of n, their algorithm makes use of a technique called "global rebuilding" due to Overmars [Ove83]. The idea involves maintaining two different copies of the data structure with different values of N, and "freshly building" one concurrently with updates on the other, in such a way that the new data structure is ready to take over before the old one gets out of its "operating range". Deletions can also be handled by this technique, by merely marking elements as deleted, and discarding marked elements while rebuilding the structure. While this approach is conceptually easy to describe, it leads to large increases in the running time (the particular global rebuilding algorithm described in [LO88] results in multiplying the run time by up to a factor of 13) and also complicates the implementation.

This data structure does not permit finger searches, however. Due to the $O(\log n)$ branching factor in the buckets, searches take $O(\log n)$ time even when fingers are provided. Another problem is the presence of the marked elements, which could increase the distance between two adjacent keys. In what follows, we will show how to overcome these problems. First we describe how to handle deletions explicitly in the above algorithm (in addition, we will avoid global rebuilding), and then we describe how to organize the buckets so that finger searches are possible.

3 An Improved Algorithm

3.1 Explicit Deletions

One problem with handling deletions explicitly is that we cannot allow the buckets to get too small. Otherwise, the deletion of the last key in a bucket would require an immediate deletion in the top-level tree, which cannot be done in constant time. We will handle deletions by operations that involve fusing small buckets. An argument similar to the one that is used to show that buckets do not grow too large can show that if the smallest underflowing bucket is chosen to rebalance each time, then the buckets do not get too small. At any point in the operation of the data structure, let $\hat{n}$ be the maximum number of keys present in the data structure at any previous time. Our buckets will always be of size $\Theta(\log^2 \hat{n})$. More precisely, if the *fullness* of a bucket b is defined to be

$$\phi(b) = size(b)/\log^2 \hat{n},$$

we will ensure that $0.5 \leq \phi(b) \leq 2$ unless b is the only bucket present. Define the *criticality* of a bucket b to be

$$\rho(b,\hat{n}) = \frac{1}{\alpha \log \hat{n}} \max\{0, 0.7\log^2 \hat{n} - size(b), size(b) - 1.8\log^2 \hat{n}\}$$

for an appropriately chosen constant α. A bucket is called *critical* if $\rho(b, \hat{n}) > 0$.

To maintain the sizes of the buckets in the appropriate range, we do the following: repeatedly pick the bucket b with maximum criticality and perform the appropriate rebalancing transformations, provided b is critical:

- *Split:* If $\phi(b) > 1.8$ split the bucket into two parts of approximately equal size.
- *Transfer:* If $\phi(b) < 0.7$ and one of its adjacent buckets b' has $\phi(b') \geq 1$ then transfer elements from b' to b till the sizes are approximately equal.
- *Fuse:* If $\phi(b) < 0.7$ and transferring is not possible, then fuse with an adjacent bucket b', *i.e.*, delete b and place all all the elements in b in b'.

We will show later how to organize the buckets so that all the above operations can be done in $O(\log \hat{n})$ time in such a way that for splitting and transferring, the resulting buckets have ϕ values which are equal to within an additive factor of $O(\log^{-1} \hat{n})$. Thus it is clear that when a critical bucket is rebalanced, it becomes non-critical. In addition to the time required to split/fuse buckets, a bucket rebalancing step may require $O(\log \hat{n})$ time to insert/delete a bucket representative from the top-level tree. Since the total work to rebalance a bucket is $O(\log \hat{n})$, we can perform it with $O(1)$ work per update spread over no more than $\alpha \log \hat{n}$ updates.

Now we show that the buckets never get too big. Updates cause the criticalities of buckets to increase, and every $\alpha \log \hat{n}$ updates a bucket with maximum criticality is made non-critical. The criticalities of buckets can increase in two ways: by changes in bucket size and by increases in the value of $\hat{n}$ (underflowing buckets only). If $\hat{n}$ increases by 1, the criticality of an underflowing bucket b will increase by δ_b, where:

$$\delta_b \leq \frac{d\,\rho(B, \hat{n})}{d\,\hat{n}} = \frac{c_1}{\hat{n}} + \frac{c_2\, size(b)}{\hat{n} \log^2 \hat{n}} \leq \frac{c_3}{\hat{n}},$$

for some constant c_3. So the change in the criticalities of all $\Theta(\hat{n}/\log^2 \hat{n})$ buckets due to a increase in 1 in the value of $\hat{n}$ adds up to an amount that is $O(\log^{-2} \hat{n})$. Since $\hat{n}$ may change by upto $O(\log \hat{n})$ between bucket rebalancings, we see that the changes in criticalities due to the change in $\hat{n}$ between rebalancings is at most $O(\log^{-1} \hat{n})$, taken over all buckets. Updates cause the criticalities to change by values adding up to at most α between rebalancings, and thus the total change to the criticalities between bucket rebalancings is $\alpha + o(1)$. By lemma 1 at no stage can the criticality of a bucket exceed $\alpha H_{\hat{n}-1} + O(1)$, and so if α is sufficiently small, this means that all buckets will have $\rho \in [0.5, 2.0]$.

3.2 Organizing Buckets for Efficient Finger Searches

In the case of a RAM, a reasonable measure of the complexity of an algorithm is the *uniform cost with logarithmic word size* [AHU74]: if the data structure has size at most M, then the RAM can in unit time perform simple manipulations (*e.g.*, arithmetic and indexing) on $c \log M$-bit words, for some small constant c. This model is motivated by the operation of computers encountered in practice: these can perform manipulations on

word-sized operands in a small (one or two) number of machine cycles, and one or two machine words usually suffice to address the entire memory into which the data structure must fit. With this stronger model, we can represent the buckets in a way that permits efficient finger searches. In this case, the permutation that represents the sorted order in sets of a sufficiently small size may be represented by bit strings that fit into a single word. Operations on these bit strings can be performed in $O(1)$ time by using the bit string to index into a precomputed table that contains the result of the operation. This is the fact that we use in:

Lemma 2 *For some sufficiently small constant c, if S is a set of at most $k \leq c \log M / \log\log M$ keys, there is a data structure that permits:*

- *Updates in constant time if a pointer to the point of insertion or deletion is known.*
- *Given a pointer to an key, search for an key a distance d away in $O(\log d)$ time.*

provided certain small tables are available precomputed.

PROOF.

Let $k \leq c \log M / \log\log M$ for some sufficiently small constant $c > 0$. The data structure which will contain no more than k elements consists of a record r with 2 fields:

- An array $A[1..k]$ that contains pointers to keys.
- An array $Perm[1..k]$, which is a *representation* of the permutation of the $A[i]$'s. (The following representation is just one of many possibilities, chosen for some slight advantages it has.) $Perm[i]$, $1 \leq i \leq k$, contains an integer in the range $[0, i]$, which is 0 if $A[i]$ is null, and if not, the rank of the item stored in $A[i]$ in the subset $\{key(A[1]), \ldots, key(A[i])\}$. That is:

$$Perm[i] = \begin{cases} 0 & \text{if } A[i]\text{=null,} \\ |\{key(A[j]) : j \leq i \wedge key(A[j]) \leq key(A[i])\}| & \text{otherwise.} \end{cases}$$

We will assume that the keys are actually stored in a doubly linked list L of records. Each record k in L will store in addition to a key, a pointer to a record r, as well as a number i such that $r.A[i]$ contains a pointer to k. Since the keys are stored in L, the positions for insertion/deletion are actually pointers to the appropriate place in L.

To perform an insertion, we determine j such that $A[j]$ points to the key after which the new element is to be inserted (this information is stored in L) and the rank of the new key (from our knowledge of j and $Perm$), insert the new key in L, set $A[|S|+1]$ to point to this new key, increment the size of S and update the $Perm$ array. To perform a deletion, we determine j such that $A[j]$ points to the deleted key and the rank of the deleted key, swap $A[j]$ and $A[|S|]$, set $A[|S|]$ to null, delete the key from L, and alter the record pointed formerly by $A[|S|]$ to record the fact that $A[j]$ now has a pointer to it, decrement the size of S and update the $Perm$ array.

The following functions are useful:

UpdatePerm(Perm, r, op): This updates the array *Perm* when a key with rank r is inserted or deleted (this is specified by *op*). Returns the updated *Perm* array.

GetRank(Perm, i): Returns the rank of the key pointed to by $A[i]$.

GetIndex(Perm, i): Returns a number j such that the key pointed to by $A[j]$ has rank i.

These functions can easily be computed in $O(k)$ time. Since we want to do better than this, we note that the contents of the *Perm* array can be specified by $k\lceil \log k \rceil$ bits. This is less than $\log M$ if $c < 1$, and the array can be represented by the contents of a single word of memory. We will precompute all the above functions for all possible values of their arguments, and use the arguments to index into a table to obtain the values of the function. Since k is very small, the size of this table will be small in comparison to the maximum size of the data structure. (*E.g.*, the arguments to *UpdatePerm* can take at most $2k \cdot 2^{k\lceil \log k \rceil}$ different values, which is $O(M^{c_2})$, $c_2 < 1$, if c is small enough.)

It should be clear that by using the *Index* function, in time $O(\log d)$ we can search for a key at a distance d away. □

Now we represent each bucket of size $\Theta(\log^2 \hat{n})$ by a tree of height 3, with a branching factor of $\Theta(\log^{2/3} \hat{n})$. At each internal node we will store a set that consists of the maximum elements stored in the subtrees rooted at each of its children. The sets stored at the internal nodes are represented in the manner described in lemma 2. In a manner similar to [DS87], we will maintain the bucket so that a node at height h has between $\Theta(\log^{2/3} \hat{n})^h$ leaves under it. In the case $h = 3$, the bucket rebalancing operations described previously ensure that the buckets always have size $\Theta(\log^2 \hat{n})$. Let $\beta = \log^{2/3} \hat{n}$. We will ensure the above for nodes at height $h = 1, 2$ by ensuring they have at least $\beta/2 - 1$ and at most $2\beta + 1$ children. If after an insertion occurs beneath a node r, we detect that it has more than 2β children, we create an additional node s that is linked to r but not to r's parent. If after a deletion we detect that a node s has less than $\beta/2$ children, we take a child of an adjacent sibling node and make it s's child, provided the sibling has a little over $\beta/2$ children. If both adjacent siblings have $\beta/2$ children then we delete s from its parent and attach it as an additional node to one of its ajdacent siblings, say r (this causes a deletion in s's parent). Define the *weight* of a node r, $w(r)$, to be $nchildren(r) + nchildren(s)/2$ if it has an additional node s, and $nchildren(r)$ otherwise. Any node r with an additional node s satisfies the following invariants:

$$3\beta/4 \leq w(r) \leq 2\beta \tag{1}$$
$$0 < nchildren(s) < nchildren(r)$$

If an insertion is made under either r or s children are moved from r to s or vice versa in order to maintain these invariants. If this is not possible, then either $size(r) = size(s)$ or $size(s) = 0$: in the first case, we insert s as a sibling of r (causing an insertion in r's parent) and in the second case, we delete s.

Violations of (1) can be caused by either a change in $\hat{n}$ or an update. If the weight of a node r is at most $O(1)$ away from satisfying (1) then in $O(1)$ time it can be rectified, by transferring $O(1)$ children. An update can cause the weight to go at most $O(1)$ 'out of balance' and so we can rebalance after an update in $O(1)$ time. In order to handle violations

caused by changes in $\hat{n}$, we start a process that sweeps through all the nodes in the entire data structure at height 1 and 2 (there are always $o(\hat{n})$ such nodes) and rectifies them if they do not satisfy (1). This process is woken up and made to run for $O(1)$ more steps whenever an update occurs. Since the process spends only $O(1)$ time at each node rectifying it, the value of $\hat{n}$ can increase only by a factor of $(1 + o(1))$ between visits, and so changes in $\hat{n}$ cause weights to go out of balance by at most $O(1)$.

To perform finger searches, we let the top-level data structure be a 2-4 tree with level links and parent pointers [HM82]. We will assume that every node inside a bucket also has parent pointers. We can search within a bucket for a key a distance d away in several ways: for example, by finding the node that is the least common ancestor of the finger key f and the search key s in $O(1)$ time and then searching downward from this node. Let the path down to s in the bucket be s_1, s_2, s_3, and let d_1, d_2, d_3, respectively be the ranks of s_2 in s_1, s_3 in s_2 and s in s_3. The distance d between s and f must be at least $d_3 + d_2 \log^{2/3} \hat{n} + d_3 \log^{4/3} \hat{n}$. The time taken for the search is $c(\log d_1 + \log d_2 + \log d_3)$, which is no more than $c(\log(d_3 \log^{4/3} \hat{n})) \leq c \log d$, since $d_1, d_2 \leq \log^{2/3} \hat{n}$. Since we can find the bucket containing s in $O(\log b)$ time, where b is the number of buckets between the ones containing s and f (using the level-links), it is easy to show that the search time in case s and f are in different buckets is $O(\log d)$ as well.

As presented above, the algorithm requires the tables to be precomputed completely. However, the precomputation can be done incrementally, *i.e.*, the table can be extended as and when necessary. If at some time the largest set we need to store has size k, and the corresponding value of $\hat{n}$ is m, then we will never have to store sets of size $k + 1$ while $\hat{n}$ has value at most $2m$. Extending the table to handle sets of size $k + 1$ will take $o(m)$ time, and can therefore be done with $O(1)$ work over each update that increases the value of $\hat{n}$. If *Perm* is stored in a word in such a way that *Perm*[1] is the least significant bit, then the new tables can be extensions of the old ones (*i.e.*, the new tables can be obtained by adding values to the ends of the old ones).

4 Conclusions and Open Problems

We have described a finger search tree that achieves constant update time when implemented on a RAM with logarithmic word size. This data structure is an improvement over the one of [LO88]: besides allowing for finger searches, it is also simpler, as it avoids having to periodically rebuild the entire data structure. We leave open the question of whether a constant update time finger search tree can be constructed on a weaker model (*e.g.*, the pointer machine). If only ***expected*** time bounds are required then the randomized search trees of [AS89] satisfy the above requirements.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AS89] C. Aragon and R. Seidel. Randomized search trees. In *Proc. 30th IEEE FOCS*, pages 540–545, 1989.

[BT80] Brown and Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 1980.

[DS87] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM STOC*, pages 365–372, 1987.

[DSST89] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Science*, 38:86–124, 1989.

[GMPR77] L. Guibas, E. McCreight, M. Plass, and J. Roberts. A new representation for sorted lists. In *Proc. 9th ACM STOC*, pages 49–60, 1977.

[Har80] D. Harel. Fast updates with a guaranteed time bound per update. Technical Report 154, University of California, Irvine, 1980.

[HL79] D. Harel and G. Lueker. A data structure with movable fingers and deletions. Technical Report 145, University of California, Irvine, 1979.

[HM82] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[Kos81] S. R. Kosaraju. Localized search in sorted lists. In *Proc. 13th ACM STOC*, pages 62–69, 1981.

[LO88] C. Levcopolous and M. H. Overmars. A balanced search tree with O(1) worst-case update time. *Acta Informatica*, 26:269–278, 1988.

[Ove82] M.H. Overmars. A $O(1)$ average time update scheme for balanced binary search trees. *Bull. EATCS*, pages 27–29, 1982.

[Ove83] M. H. Overmars. *The design of dynamic data structures, LNCS 156.* Springer-Verlag, 1983.

[Tsa85] A. K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67:173–194, 1985.

A Details of the Algorithm

We will not describe the implementation of 2-4 trees, with level links. Nodes in the buckets have the following fields:

parent: Contains a pointer to the parent of the node. For the root of a bucket this will contain a pointer to its position in the top-level tree.

$A[1..k]$: An array of size k (k sufficiently small for lemma 2 to apply) — contains pointers to leaves or children.

perm: A description of the permutation that represents the sorted order within A.

index: This field is 0 for the root of a bucket. For leaves and non-root nodes. contains an integer i, such that the ith position in the array A in their parent contains a pointer to them.

nchildren: The number of children of the node. Field absent for leaves.

size: The number of leaves in the subtree rooted at that node. Field absent for leaves.

extra: A pointer to an 'overflow' node.

key: For leaves, this contains the key associated with them and for internal nodes it contains the largest element in the subtree rooted at the node.

There is a priority queue PQ of buckets, ordered by size. The implementation must do the following operations all in $O(1)$ time (this is possible since priorities change only by ± 1 at a time):

$PQInsert(b, s)$: Insert a new bucket of given size s into PQ, given that s is not a critical size.

$ChangePriority(b, d)$: Change the priority of b by $d \in \{-1, 1\}$.

$ResetPriority(b, s)$: Reset the priority of b to s. Again, we are given that s is not a critical size.

$MostCriticalBucket()$: Return the bucket with maximum criticality (return if no buckets are critical).

The implementation that we choose is the following. At any time let m be the maximum possible bucket priority. We will have an array $P[1..m]$, such that $P[i]$ points to a doubly linked list of all buckets with priority i. In addition, let l and u denote the lower and upper limits for the sizes of noncritical buckets (*i.e.*, a bucket b is critical iff $size(b) \notin [l, u]$). Note that l and u may change by at most 1 per update (due to changes in $\hat{n}$),

and our implementation must be able to take care of this. The indices of all non-empty elements of $P[1..l-1]$ are placed in a list L_1 in ascending order. Similarly all non-empty indices in $P[u+1..m]$ are placed in a list L_2 in descending order. Since the bucket of maximum criticality must be a bucket of maximum or minimum size, and the heads of L_1 and L_2 point to the smallest and largest critical bucket sizes, we can perform the operation $MostCriticalBucket()$ in constant time. To insert a non-critical bucket with size s into the priority queue, we merely insert it at the front of the list pointed to by $P[s]$. To delete a bucket with size s, we delete it from $P[s]$. If this causes $P[s]$ to become empty and s is a critical size, then we delete s from L_1 or L_2, as appropriate. To change the priority of a bucket (with old priority s) to $t = s \pm 1$, we remove the bucket from the list pointed to by $P[s]$, and add it onto $P[t]$. If $P[t]$ was formerly empty and t is a critical size then we insert t next to s in L_1 or L_2, as appropriate. If $P[s]$ becomes empty as a result of this movement and s is a critical size we delete s from L_1 or L_2. Finally, if u or l changes (by ± 1), we can modify L_1 or L_2 appropriately (by adding/deleting to their tails).

```
(* Insert y immediately after x in the search tree *)
proc Insert(x, y)
   insert y after x in L;
   y↑.parent ← x↑.parent;
   UpdateMax(x, del);
   UpdateSizes(x, ins);
   InsertIntoBucket(x, y);
   Do O(1) steps of RebalanceBucket
end Insert


(* Delete x from the search tree *)
proc Insert(x, y)
   delete x from L;
   UpdateMax(x, del);
   UpdateSizes(x, del);
   DeleteFromBucket(x);
   Do O(1) steps of RebalanceBucket
end Delete
```

Figure 1: Insertion/deletion algorithms

```
(* Change the size values in x's bucket and in the entire data structure due to the update
that occured *)
proc UpdateSizes(x, type)
   if type = ins then delta ← 1 else delta ← −1;
   for i ← 1 to 3 do
      x ← x↑.parent;
      x↑.size ← x↑.size + delta;
   end for;
   (* x is now the root of the bucket *)
   ChangePriority(x, delta);
   n ← n + delta;
   if n > n̂ then
      n̂ ← n;
      Do O(1) steps of ConstructTables;
   end if
end UpdateSizes

(* Change the search info in x's bucket due to the update that occured *)
proc UpdateMax(x, type)
   val ← x↑.key;
   if type = ins then
      for i ← 1 to 3 do
         x ← x↑.parent;
         if (val > x↑.max) then x↑.max ← val
      end for
   else
      (* newval is the second largest value in x's parent *)
      with x↑.parent do
         newval ← A[GetIndex(perm, nchildren − 1)]↑.key;
      for i ← 1 to 3 do
         x ← x↑.parent;
         if (val = x↑.max) then x↑.max ← newval
      end for
   end if
end UpdateMax
```

Figure 2: Updating size/search information

```
(* Inserting y after x in their common bucket *)
proc InsertIntoBucket(x, y)
   p ← x↑.parent; cascade ← true ;
   while cascade do
      InsertChild(p, x, y);
      q ← Balance(p):
      if q ≠ null then
         x ← p; y ← q; p ← p↑.parent
      else
         cascade ← false
      end if
   end while
end InsertIntoBucket

(* Deleting x from its bucket *)
proc DeleteFromBucket(x)
   p ← x↑.parent; cascade ← true ;
   while cascade do
      DeleteChild(p, x);
      q ← Balance(p);
      if q ≠ null then
         x ← q; p ← p↑.parent
      else
         cascade ← false
      end if
   end while
end DeleteFromBucket

(* Insert y after x as a child of p *)
proc InsertChild(p, x, y)
   with p↑ do
      nchildren ← nchildren + 1; A[nchildren] ← y; y↑.index ← nchildren;
      perm ← UpdatePerm(perm, GetRank(perm, y↑.index), ins);
   end with
end InsertChild

(* Remove x as a child of p *)
proc DeleteChild(p, x)
   with p↑ do
      A[nchildren]↑.index ← x↑.index; A[nchildren] ↔ A[x↑.index];
      A[nchildren] ← null; nchildren ← nchildren − 1;
      perm ← UpdatePerm(perm, x↑.index, del);
   end with
end DeleteChild
```

Figure 3: Updates in buckets

```
(* The process that periodically picks the most critical bucket and rebalances it. Next()
is a function that returns the adjacent bucket of larger size or with the larger number of
children *)
process RebalanceBucket
   while true do
      b ← MostCriticalBucket();
      if b↑.size > 1.8 log² n̂ then (* split *)
         Insert a new node b' after b in the 2-4 tree;
         while b↑.size ≥ b'↑.size do TransferChild(b, b');
         InsertPQ(b, b↑.size); InsertPQ(b', b'↑.size);
      elsif Next(b)↑.size > log² n̂ then (* transfer *)
         b' ← Next(b);
         while b↑.size ≥ b'↑.size do TransferChild(b, b');
         InsertPQ(b, b↑.size); ResetPriority(b', b'↑.size);
      else (* fuse *)
         b' ← Next(b);
         while b↑.size > 0 do TransferChild(b, b');
         Delete b from the 2-4 tree;
         ResetPriority(b', b'↑.size);
      end if
   end while
end RebalanceBucket

(* Move one child from one bucket to another *)
proc TransferChild(b, b')
   Let x be the rightmost child of b;
   Delete x from b and make it a child of b';
   b↑.size ← b↑.size − x↑.size;
   b'↑.size ← b'↑.size + x↑.size;
end TransferChild
```

Figure 4: Routines for periodically rebalancing buckets

```
(* Called to rebalance nodes at height 1,2 or 3, returning a sibling node that is required
to be inserted or deleted, if necessary. Does not attempt to rebalance nodes at height 3.
β = log^{2/3} n̂ *)
proc Balance(p)
   if p↑.root then return null;
   if p↑.extra = null then
      if p↑.nchildren > 2β then
         Create a new node q;
         p↑.extra ← q;
      elsif p↑.nchildren < β/2 then
         Let q be the adjacent sibling of p with most children;
         if (q↑.nchildren > β/2 + 5) then
            while p↑.nchildren < β/2 do TransferChild(q, p);
            return null;
         elsif w(q) > 0 then
            while p↑.nchildren < β/2 do TransferChild(q, p);
            Balance(q) (* This call will always return null *); return null;
         else
            If q is to the right of p, exchange p and q; p↑.extra ← q; retval ← q;
         end if
      end if
   else
      q ← p↑.extra;
      if w(p) < 3β/4 then
         while (q↑.nchildren > 0) ∧ (w(p) < 3β/4)do TransferChild(q, p);
         if (q↑.nchildren = 0) then p↑.extra ← null;
      elsif w(p) > 2β then
         while (q↑.nchildren < p↑.nchildren) ∧ (w(p) < 2β) do
            TransferChild(p, q);
         if (q↑.nchildren ≥ p↑.nchildren) then
            p↑.extra ← null;
            retval ← q;
         end if
      end if
   return retval;
end Balance
```

Figure 5: Balancing nodes at height 1 or 2