Randomized Search Trees

Cecilia R. Aragon* Raimund G. Seidel†
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 an update takes logarithmic time and requires 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. 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. Our balancing strategy and our algorithms are exceedingly simple and should be fast in practice.

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" 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 [AVL], (a,b)-trees [BMc], $BB(\alpha)$-trees [NR], red-black trees [GS], 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 [M1]). This fact is important for the parallel use of these search trees, and also for applications in computational geometry, where often 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 [M3]).

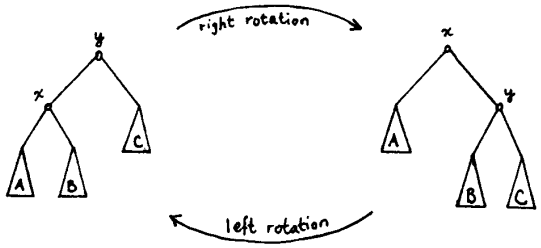

Figure 1

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 [BST] and D-trees [M1], 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 per-

*Supported by an AT&T graduate fellowship
†Supported by NSF grant CCR-8809040

formed 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. There is one method, though, that requires no such additional information to be maintained and that uses an extremely simple restructuring strategy, namely *splay trees* of Sleator and Tarjan [ST]. This method achieves all the access and update time bounds mentioned before 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.

In this abstract we present a strategy for balancing 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.

Sections 2 and 3 describe our approach and state all the results. In section 4 we give some proofs.

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.[1] "In-order" means that for any node x in the tree $y.key \leq x.key$ for all y in the left subtree of x and $x.key \leq y.key$ for y in the right subtree of x. "Heap-order" means that for any node x with parent z the relation $x.priority \leq z.priority$ 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.

Let T be the treap storing set X. Some item $x \in X$ can easily be accessed in T via the usual search tree algorithm given the key of x. 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.

Since an insertion operation can be viewed as a deletion operation performed backwards it suffices to analyze deletions. Their running time is determined by the time necessary to access the node x to be deleted and by the number of rotations. The first quantity is proportional to the depth of x in T, the second is easily seen to be the length of the right spine of the left subtree of x plus the length of the left spine of the right subtree of x. By *right spine* of a tree we mean the root of the tree together with the right spine of the right subtree of the root. The left spine is defined analogously.

Sometimes 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.[2]

[1] 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].

[2] In practice it will be preferable to approach these operations the other way round. Joins and splits of treaps can

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 1 *Let T be a randomized search tree for a set X of n items.*

(i) The expected time necessary to access an item $x \in X$ in the tree T is $O(\log n)$.

(ii) The expected time necessary to perform an insertion into T or a deletion from T is $O(\log n)$.

(iii) The expected number of rotations per insertion or deletion is less than 2.

(iv) If the cost of a rotation of a subtree of size s is assumed to be $O(s)$, then the expected time necessary to perform an insertion or deletion is still $O(\log n)$.

(v) If the cost of a rotation of a subtree of size s is assumed to be $O(s \log^k s)$, with $k \geq 0$, then the expected time necessary to perform an insertion or deletion is $O(\log^{k+1} n)$.

(vi) If the cost of a rotation of a subtree of size s is assumed to be $O(s^a)$, with $a > 1$, then the expected time necessary to perform an insertion or deletion is $O(n^{a-1})$.

(vii) The expected time necessary to join two randomized search trees with n and m nodes, respectively, is $O(\log n + \log m)$.

(viii) The expected time necessary to split T into two trees of size n and m, respectively, is $O(\log n + \log m)$.

(ix) For two items $x, y \in X$ that differ in key rank by d the expected length of the path connecting them in T is $O(\log d)$.

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 2 *Let T be a weighted randomized search tree for a set X of weighted items. Let W be the sum of the weights of the items in X.*

(i) The expected time necessary to access an item $x \in X$ with weight $w.x$ is $O(1 + \log(W/w.x))$.

(ii) The expected time necessary to insert an item y with weight $w.y$ into the tree T is

$$O(1 + \log((W + w.y)/\min\{w.x, w.y, w.z\})),$$

where x and z are the predecessor and successor of y in X (with respect to the order of the keys).

(iii) The expected time necessary to delete from T an item y with predecessor x and successor z is

$$O(1 + \log(W/\min\{w.x, w.y, w.z\})).$$

(iv) The expected number of rotations necessary for the insertion or deletion of y is

$$O(1 + \log(1 + w.y/w.x) + \log(1 + w.y/w.z)).$$

(v) The expected time necessary to split T into two randomized search trees T_1 and T_2 (or to join T_1 and T_2) is

$$O(1 + \log(W_1/w.x) + \log(W_2/w.z)),$$

where W_i is the total weight of the items in T_i, x is the maximal element in T_1 and z is the minimal element in T_2.

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. It is assumed 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.

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

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.

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 $[0,1]$.

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? Surprisingly, an expected *constant* number of random bits per update suffices 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. Related issues, such as the expected size of the partial binary representations of the priorities, and the analogous questions for weighted randomized search trees are currently being investigated. Preliminary experiments suggest that in the unweighted case the expected number of random bits needed per node is about 4.34. So far we have no theoretical explanation for this particular constant.

Recently, Bob Tarjan related to us a suggestion of Dan Sleator that would make it unnecessary to store explicit priorities with the nodes in the case of unweighted randomized search trees. The idea is to assign priorities to items via a universal hash function and to (re)compute priorities only when needed. Besides saving space this would also yield a unique tree representation for any fixed set of items, which seems to be of interest in some applications.

4 Analysis

Before we analyze randomized search trees let us consider a few "games." Similar games have been considered by Mulmuley in the context of the analysis of randomized algorithms in computational geometry [Mu]. Assume we have a set P of p "players," a set B of b "bystanders," a set T of t "triggers," and a set S of s "stoppers." Assume that these four sets are pairwise disjoint. The sets B and T can be arbitrary. However, assume the sets P and S to be totally ordered, with the elements of P being smaller than the elements of S. Consider the following four "games:"

Game A involves only players and bystanders. Repeatedly choose and remove some element from $P \cup B$ at random, until no elements of P remain. We are interested in A^p, the expected number of times that an element from P is chosen that is larger than all previously chosen elements from P.

Game C involves only players, stoppers, and bystanders. Repeatedly choose and remove some element from $P \cup S \cup B$ at random, until some stopper from S is chosen, at which point the game ends. We are interested in C_s^p, the expected number of times that an element from $P \cup S$ is chosen that is larger than all previously chosen elements from $P \cup S$.

Game D involves only players, triggers, and bystanders. Repeatedly choose and remove some element from $P \cup T \cup B$ at random, until no elements of P remain. We are interested in D_t^p, the expected number of times that, after some trigger from T has been chosen, an element from P is chosen that is larger than all previously chosen elements from P.

Game E involves all four sets. Repeatedly choose and remove some element from $P \cup T \cup S \cup B$ at random, until some stopper from S is chosen, at which point the game ends. Here we are interested in $E_{t,s}^p$, the expected number of times that, after some trigger from T has been chosen, an element from $P \cup S$ is chosen that is larger than all previously chosen elements from $P \cup S$.

In the following let H_k denote $\sum_{1 \le i \le k} 1/i$.

Lemma 1

(a) $A^p = H_p$

(b) $C_s^p = 1 + H_{s+p} - H_s$

(c) $D_t^p = H_p + H_t - H_{p+t}$

(d) $E_{t,s}^p = \frac{t}{t+s} + (H_{s+p} - H_s) - (H_{t+s+p} - H_{t+s})$

Proof: First observe that the bystanders are really irrelevant to the problem. Next note, that during these games, whenever an element from P is chosen all smaller elements from P are in effect relegated to bystander status. Thus we get the following recurrence relations:

$$A^p = \tfrac{1}{p} + \textstyle\sum_{1<i\le p} \tfrac{1}{p}(1 + A^{i-1})$$

$$C_s^p = \tfrac{s}{s+p} + \textstyle\sum_{0\le i<p} \tfrac{1}{s+p}(1 + C_s^i)$$

$$D_t^p = \tfrac{t}{t+p} A^p + \textstyle\sum_{0\le i<p} \tfrac{1}{t+p} D_t^i$$

$$E_{t,s}^p = \tfrac{t}{t+s+p} C_s^p + \textstyle\sum_{0\le i<p} \tfrac{1}{t+s+p} E_{t,s}^i$$

Using the fact that $\sum_{0\le i<k} H_i = kH_k - k$ it is then routine to check that the expressions given in the lemma satisfy these recurrence relations.[3] ∎

[3] The reader may also want to check that the relations $A^p = C_1^{p-1} = D_\infty^p = E_{\infty,1}^{p-1}$ hold, as they should, as well as $C_s^p = E_{\infty,s}^p$ and $D_t^p = E_{t,1}^{p-1}$.

What is the relevance of these games to randomized search trees? In section 2 we showed how to perform splits and joins of treaps via insertions and deletions. Moreover we argued that insertions are just "inverse" operations of deletions. Thus it suffices to analyze deletions. We argued that the time necessary to delete a node depends on the time necessary to access it, plus the time needed to rotate it down to a leaf. The first quantity is proportional to the depth of the node in the tree, the second is proportional to the length of the right spine of the left subtree of the node plus the length of the left spine of its right subtree. Thus it suffices to bound the expectations of these quantities for randomized search trees.

We say that the *key rank* of an item x in a set X is k iff there are exactly k elements in X with key not greater than the key of x.

Lemma 2 *Let X be a set of n items with distinct keys, and let $x \in X$ be the item with key rank k.*

(i) The expected depth of x in a randomized search tree for X is exactly $H_k + H_{n-k+1} - 1$.

(ii) In a randomized search tree storing X the expected length of the right spine of the left subtree of x is $1 - 1/k$, and the expected length of the left spine of the right subtree of x is $1 - 1/(n-k+1)$.

Proof: Let X' be the items in X with key not greater than the key of x. In order to prove (i) it suffices to show that on the path from the root to x the expected number of nodes from X' is H_k.

Arrange the items in X by decreasing priority. Since the priorities are i.i.d. random variables every permutation of X is equally likely to occur. Imagine one constructed the treap for X according to these priorities by inserting nodes by decreasing priority. It should be clear that this way the tree will grow only at the leaves and that no rotations can occur. Let us concentrate our attention on X'. It is not difficult to see that the items in X' that lie on the access path to x are exactly those that, when added to the tree, are larger (in key) than the previously inserted members of X'. Thus, in essence, game A is played with k players X' and $n-k$ bystanders (or equivalently, game C is played with $k-1$ players and 1 stopper, namely x) and thus the expected number of items from X' on the access path to x is H_k.

In order to prove (ii) it suffices by symmetry to show that the expected length of the right spine of the left subtree of x is $1 - 1/k$. Using similar arguments as in the previous paragraph it is clear that determining the expected length of this spine amounts to determining D_1^{k-1} of game D with $k-1$ players $X' \setminus \{x\}$, trigger x, and bystanders $X \setminus X'$. But $D_1^{k-1} = H_{k-1} + H_1 - H_k = 1 - 1/k$. ∎

The argument in the proof of this lemma can be adapted to weighted randomized search trees as follows: For each $y \in X$ let $\overline{y}$ be the multiset consisting of $w.y$ copies of y. Let $\overline{X}$ be the multiset formed by the union of the $\overline{y}$'s. Assign i.i.d. random variables as priorities to the elements in $\overline{X}$ and arrange them in decreasing order. Again, each permutation must occur with equal likelihood. Next the items of $\overline{X}$ are inserted into a treap in decreasing order of their priorities. However, whenever an itemcopy is to be inserted for which a copy already exists in the tree, this itemcopy is discarded. It should be clear that upon completion the priority of every item y in the tree will be the maximum of $w.y$ i.i.d. random variables, as desired. It should also be clear that the number of nodes on the access path to y with keys less than the key of y is bounded from above by $C^m_{w.y}$ of the game C played with the m players $\bigcup_{u.key<y.key} \overline{u}$ and the $w.y$ stoppers $\overline{y}$. Similarly, the expected length of the right spine of the left subtree of y will be bounded by $E^m_{w.y,w.x}$ of the game E played with the m players $\bigcup_{u.key<x.key} \overline{u}$, $w.y$ triggers $\overline{y}$, and $w.x$ stoppers $\overline{x}$, where x is the item in X that immediately precedes y in key. With these observations the following lemma can be proved easily.

Lemma 3 *Let X be a set of weighted items. Let x, y, z be three items consecutive in key. Let i, j, k be the weights of x, y, z, respectively, and let α be the sum of the weights of the items preceding x, and β be the sum of the weights of the items succeeding z.*

(i) The expected depth of x in a weighted randomized search tree for X is not greater than

$$1 + H_{\alpha+i+j} + H_{\beta+k+j} - 2H_j.$$

(ii) The expected length of the right spine of the left subtree of x is not greater than

$$\tfrac{j}{i+j} + (H_{\alpha+i} - H_i) - (H_{\alpha+i+j} - H_{i+j}).$$

The expected length of the left spine of the right subtree of x is not greater than

$$\tfrac{j}{k+j} + (H_{\beta+k} - H_k) - (H_{\beta+k+j} - H_{k+j}).$$

These last two lemmas together with the standard approximation of H_k by the natural logarithm of k imply all the results stated in Theorems 1 and 2, except for the statements about costly rotations. Consider, for instance, claim (iv) of Theorem 1, which stated that the expected update time is still $O(\log n)$ if the cost of a rotation of a subtree of size s is assumed to be $O(s)$. In a nutshell, this claim can be proved as follows: first one shows by a simple counting argument that the probability p_s for a node x in a randomized search tree to be the root of a subtree of size s is $O(1/s^2)$ (except for the case $s = n$ where the probability is $1/n$). Next, assuming that the cost of rotating a tree of size t is $O(t)$, one computes the expected total cost R_s of rotating the root of a randomized search tree of size s all the way down to a leaf. R_s turns out to be $O(s)$. Summing $p_s R_s$ over all s then yields that the expected total costs of rotations in course of a deletion is $O(H_n)$ which is $O(\log n)$, as claimed. The other claims can be proved analogously.

5 Conclusion

Randomized search trees appear to have a large number of attractive properties that should make them very useful in practice. We have not had opportunity to compare them experimentally with other kinds of search trees. But of course our hope is that performance-wise randomized search trees relate to other kinds of search trees as *Quicksort* relates to other sorting methods. This hope is based on the obvious close relationship between *Quicksort* and randomized search trees.

6 References

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

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

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

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

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

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

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

[M1] Mehlhorn, K. *Sorting and Searching.* Springer (1984).

[M3] Mehlhorn, K. *Multi-dimensional Searching and Computational Geometry.* Springer (1984).

[P] Pugh, B. Skip Lists: A Probabilistic Alternative to Balanced Trees. Univ. of Maryland, Computer Science Dept. Tech. Rep. CS-TR-2190 (1989).

[ST] Sleator, D.D., and Tarjan, T.E. Self-adjusting binary search trees. *JACM 32* (1985), 652-686.

[V] Vuillemin, J. A Unifying Look at Data Structures. *CACM 23* (1980), 229-239.