A Simple Balanced Search Tree With $O(1)$ Worst-Case Update Time [1]

by

Rudolf Fleischer [2]

ABSTRACT In this paper we show how a slight modification of (a, b)-trees allows us to perform member and neighbor queries in $O(\log n)$ time and updates in $O(1)$ worst-case time (once the position of the inserted or deleted key is known). Our data structure is quite natural and much simpler than previous worst-case optimal solutions. It is based on two techniques : 1) *bucketing*, i.e. storing an ordered list of $2 \log n$ keys in each leaf of an (a, b) tree, and 2) *lazy splitting*, i.e. postponing necessary splits of big nodes until we have time to handle them. It can also be used as a finger tree with $O(\log^* n)$ worst-case update time.

1. Introduction

One of the most common (and most important) data structures used in efficient algorithms is the balanced search tree. Hence there exists a great variety of them in literature. Basically, they all store a set of n keys such that location, insertion and deletion of keys can be accomplished in $O(\log n)$ worst-case time.

In general, updates (insertions or deletions) are done in the following way : First, locate the place in the tree where the change has to be made; second, perform the actual update; and third, rebalance the tree to guarantee that future query times are still in $O(\log n)$. The second step usually takes only $O(1)$ time, whereas steps 1 and 3 both need $O(\log n)$ time. But there are applications which do not need the first step because it is already known where the key has to be inserted or deleted in the tree. In these cases we would like to have a data structure which can do the rebalancing step as fast as the actual update, i.e. in constant time.

One such example are dynamic planar triangulations. In [M91] Mulmuley examined (among others) point location in dynamic planar Delauney triangulations. The graph of the triangulation is stored such that at each node of the planar graph the adjacent triangles are stored (sorted in clockwise radial order) in a balanced search tree. But now, whenever a point v is deleted or inserted, all points in the neighborhood of v can be affected by the retriangulation because their sequence of adjacent triangles might have changed. However, these changes are only local in the sense that one triangle (which is known at that time

[1] This work was supported by the ESPRIT II program of the EC under contract No. 3075 (project ALCOM)

[2] Max-Planck-Institut für Informatik, W-6600 Saarbrücken, Germany

and has not to be searched in the radial tree) must be deleted or get some new neighbors (see [M91], 3.2 for details). To guarantee a worst-case update time for the triangulated point set which is proportional to the structural change (i.e. the number of deleted or newly created triangles) one needs search trees at the nodes which can handle updates in constant worst-case time.

It has been well known for a long time that some of the standard balanced search trees can achieve $O(1)$ amortized update time once the position of the key is known ([GMPR],[HM],[O82]). But for the worst-case update time the best known method had been a complicated $O(\log^* n)$ algorithm by Harel ([H79],[HL]). It has also been known that updates can be done with $O(1)$ structural changes (e.g. rotations) but the node to be changed has to be searched in $\Omega(\log n)$ time ([T83]). Levcopoulos and Overmars ([LO]) have only recently come up with an algorithm achieving optimal $O(1)$ update time (a similar result had been obtained by [vE]). They use the *bucketing technique* of [O82] : Rather than storing single keys in the leaves of the search tree, each leaf (*bucket*) can store a list of several keys. Unfortunately, the buckets in [LO] have size $O(\log^2 n)$; so they need a 2-level hierarchy of lists to guarantee $O(\log n)$ query time within the buckets. They show that this bucket size is sufficient if after every $\log n$ insertions the biggest bucket is split into two halves and then the rebalancing of the search tree is distributed over the next $\log n$ insertions (for which no split occurs).

Our paper simplifies this approach considerably : We, too, distribute the rebalancing over the next $\log n$ insertions into the bucket which was split, but allow many buckets to be split at consecutive insertions (into different buckets). This seems fatal for internal nodes of the search tree : they may grow arbitrarily big because of postponed (but necessary) splits. But we show that internal nodes will never have more than twice the allowed number of children; hence queries can be done in $O(\log n)$ time. Furthermore, our buckets can grow only up to size $2\log n$, which means that we only need an ordered list to store the keys in a bucket. Also, the analysis of our algorithm seems simpler and more natural than in [LO].

The paper is organized as follows. In Section 2 we define the data structure and give the algorithms for find and insert. In Section 3 we prove their efficiency. Then we conclude with some remarks in Section 4.

2. The Data Structure

In this Section we want to describe a simple data structure which maintains a set S of ordered keys and allows for operations *query*, *insert* and *delete*. *Queries* are the so-called *neighbor queries* : given a key K, if it is in the current set S report it, otherwise, report one of the two neighbors in S according to the given order. *Insert* and *delete* assume that we have previously located the key (to be deleted) or one of its neighbors (if we insert a new key) in the data structure. As was illustrated by the triangulation example in Section 1 where we have two nested data structures, this does not necessarily mean that we must perform a query in our data structure to locate this key. Our data structure is basically a balanced search tree, a variant of an (a, b)-tree ($4 \leq 2a \leq b$ and b even).

The main problem with update operations in a balanced search tree (and all other query-efficient search structures) is that it is not enough only to insert or delete a node, but it is also necessary to take care of the balanced structure of the tree if future queries are still to be efficient. This means that we should rebalance the tree after each update. Unfortunately, this rebalancing can affect the whole path from the node to the root of the tree, which can be of length $\Omega(\log n)$. However, we will show in the next Section that in our search tree the rebalancing does not need to be performed immediately but can be distributed, step by step, over following updates which do not need a costly rebalancing. Thus we can guarantee constant worst-case update time.

In [OvL] and [O83], Overmars presented a very general method of handling deletions efficiently, the *global rebuilding technique*. We only give a short outline of this method here and refer to the original papers for details.

The idea is that a delete operation only deletes the node without doing any other operations, especially no rebalancing. This does not increase the query or insert time, but it does not decrease it either as it should. But since the optimal query time for $\frac{n}{c}$ keys is still $\log n - c$, we can afford being lazy for quite a long time before running into trouble (this is the reason why this method works only for deletions and not for insertions: $\frac{n}{c}$ insertions, all into the same position, can result in a path of length $\Omega(n)$ which would be disastrous for queries).

If there are too many deletions and the number of keys in the tree sinks below $\frac{n}{c}$ (here $c \geq 2$ is some constant), we start rebuilding the whole tree from scratch. Since this takes linear time, we distribute it over the next $\frac{n}{3c}$ operations, i.e. we still use and update our original (but meanwhile rather unbalanced) tree, but in parallel we also build a new tree, and we do this three times as fast. If the new tree is completed, we must still perform the updates which occured after we started the rebuilding. But again, we can do this in parallel during the next $\frac{n}{9c}$ operations. This continues until we finally reach a state where both trees store the same set set of (at least $(\frac{1}{3} + \frac{1}{9} + \cdots)\frac{n}{c} = \frac{n}{2c}$) keys. Now we can dismiss the old tree and use the new tree instead. Since $c \geq 2$, we are always busy constructing at most one new tree. Hence the query and update time can only increase by a factor of 4.

This allows us to focus on a data structure which can only handle insertions; deletions can then be done using the global rebuilding technique. Now we give the details of our data structure. Assume that we initially have a set S_0 of n_0 keys (n_0 could be zero).

The Tree : Let $4 \leq 2a \leq b$ and b even. Then our tree T can be viewed as an $(a, 2b)$-tree, i.e. its internal nodes have between a and $2b$ children. However, each leaf does not store a single key but contains a doubly-linked ordered list of several keys; so we call the leaves *buckets*. Furthermore, each bucket B has a pointer r_B which points to some node within T (usually on the path from the root to B).

We remark that it is not really important that the keys in a bucket are stored in sorted order, but our algorithm automatically inserts new keys at the correct position of the list. For a node v of T let $size(v)$ denote the number of its children. We call v *small* if $size(v) \leq b$, otherwise *big*. We want to split big nodes into two small nodes whenever we encounter one. This makes our tree similar to an (a, b)-tree; the main difference is that we cannot afford to split big nodes immediately when they are created, but instead have to wait for a later insertion which can do the job. Insertions are done as follows.

Algorithm A : (A.1) Insert the new key into the bucket. Let this bucket be B with r_B pointing to node v.

(A.2) If v is big then split v into two small nodes.

(A.3) If v is the root of T then split B into two halves and let the r-pointer of both new buckets point to their common father. Otherwise, set r_B to $father(v)$, i.e. move it up one level in the tree.

Initially, we start with an (a, b)-tree T_0 (which is also an $(a, 2b)$-tree) for the set S_0 such that each bucket contains exactly one key of S_0. Also, all pointers r_B point to their own bucket B. However, it is not clear at this point that Algorithm A really preserves the $(a, 2b)$-property of T; one could think that some nodes could grow arbitrarily big because we could split the children of a node too often before testing and splitting the node itself in (A.2). But we will show in the next Section that this can not happen. Therefore it is reasonable to speak of splitting a big node into only two small nodes in (A.2). Nevertheless, this splitting can not be done arbitrarily but must follow some easy rules which will be given in the proof of Lemma 3.1 and in Lemma 3.3.

Furthermore, in (A.3), we want to split a bucket into two halves, and this should be done in constant time. Therefore we must design the buckets a little bit more complicated than just using a list. It is not really difficult, but it is technical : Each bucket has two headers (with five entries) to control the list (see Fig. 2.1). Initially, or after a split, *rightheader* controls an empty sublist. At each insertion, one key is added to the sublist of *rightheader* (either the inserted key or the rightmost key of the sublist of *leftheader*). Then we can easily split a bucket in (A.3) in constant time by changing only the header information and creating two new empty headers. This does not split the bucket exactly into two halves, but it is sufficient for our purpose as the next Lemma shows.

Lemma 2.1 Let n be the number of keys currently stored in T. Then $size(B) \leq 2 \log n$ for all buckets B.

Proof : Easy by induction once we have proven that T is always an $(a, 2b)$-tree of height at most $\log n$ (see Theorem 3.2). □

The r-pointers of the buckets always move up the tree from the leaves to the root, then starting again at the leaf-level. They usually follow the path from their own bucket to the root but if a node v is split (by an insertion into another bucket) then some r-pointers may point to the wrong sibling (if we create one new node as a sibling for v then all buckets in the subtree below the new node have their r-pointer still pointing to v which is not on their path to the root). But the analysis in Lemma 3.1 shows that this is not a problem (only Invariant (I.6) deals with r-pointers, and in case of a split both siblings must belong to the same red tree). Hence it is not necessary to care about the r-pointers pointing to v when v is split.

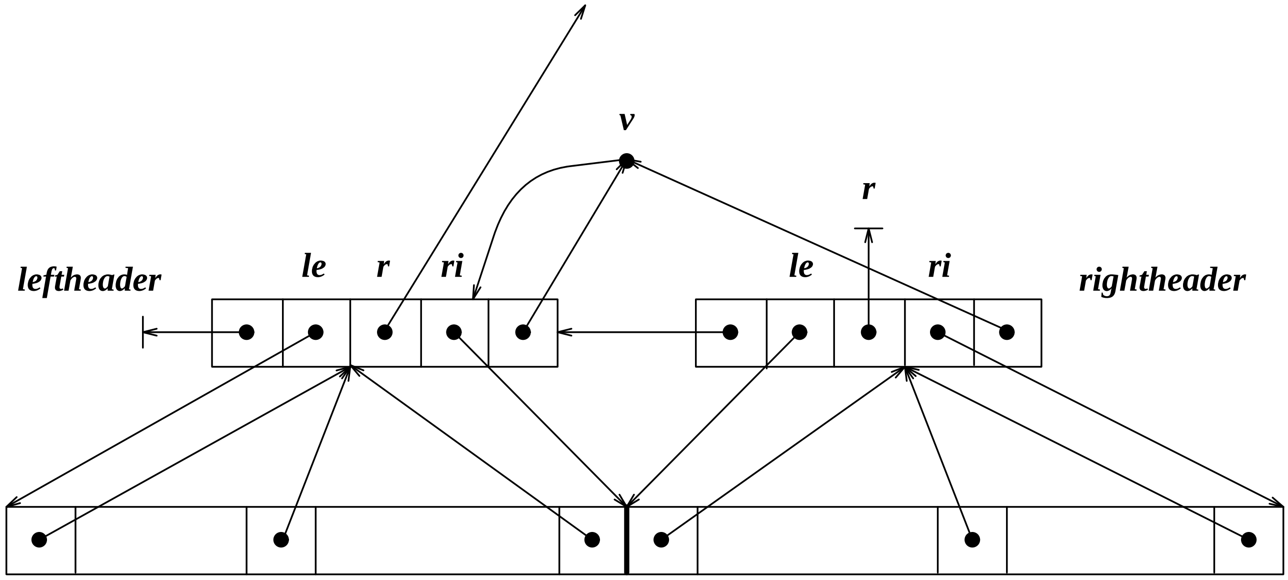

Fig. 2.1 A bucket

3 . The Analysis

In this Section we prove that Algorithm A will never blow up an internal node of T, i.e. T actually always remains an $(a, 2b)$-tree. The idea is to show that the algorithm always splits a big node into only two small nodes, and that none of these two nodes can be split again before their common father has been tested and found to be small (or made small by a split). Since we start with an (a, b)-tree T_0, this proves that T is always an $(a, 2b)$-tree.

In order to prove this we extend Algorithm A to an Algorithm B which computes some additional attributes for the nodes and edges of the tree. These attributes allow us to conclude easily the desired properties of the tree T' constructed by Algorithm B. But since T and T' are identical (the additional attributes in no way influence the structure of the tree) the claim also holds for the tree T of Algorithm A.

First, we define the attributes which are just colours : Nodes and edges are coloured *blue* or *red*. A node is red if one of its incident edges is red, and blue otherwise. The red nodes together with the red edges form a forest of subtrees of T; we call the trees of this forest *red trees*. Leaves (buckets) are always red. Initially (in T_0), all other nodes and all edges are blue; this means that each bucket is the root of a red tree which only consists of the bucket itself. We call these red trees *trivial*. As we will see in the proof of Lemma 3.1, red edges can only be created by splitting a blue edge into two red edges; hence red edges always appear as pairs of two red edges. The red colour shall indicate that these edges can not be split (as long as they are red).

The edge connecting a node to its father is called *f-edge*, whereas the edges going to its children are called *c-edges*. We define the (non-existent) f-edge of $root(T)$ as always being blue. For a node v, let d_v (e_v) denote the number of its red (blue) c-edges. Obviously, $size(v) = d_v + e_v$. If $d_v + 2e_v \leq b$ then v is called *blocking*. In particular, blocking nodes are small and can not grow big (because red edges are never split and blue edges are always split into two red edges); hence they can not propagate a split to their father. For a leaf (bucket) B we define $d_B = 0$ and $e_B = 0$, i.e. leaves are blocking.

We now give the extended algorithm.

Algorithm B : (B.1) Insert the new key into the bucket. Let this bucket be B with r_B pointing to node v.

(B.2) If v is big then split v into small nodes and colour these nodes, their f-edges and their common father red (Lemma 3.1 shows that we always obtain only two small nodes). Otherwise, if v is the root of a nontrivial red tree (i.e. v is a red inner node of T with a blue f-edge) then $colour\_blue(v)$.

(B.3) If v is the root of T then split B into two halves and let the r-pointer of both new buckets point to their common father; also, colour both buckets, their f-edges and their common father red. Otherwise, set r_B to $father(v)$, i.e. move it up one level in the tree.

$colour\_blue(v)$ is a subroutine which recursively recolours (blue) a whole red tree with root v.

$colour\_blue(v)$: Colour v blue;
For all red c-edges $e = (v, w)$ colour e blue, and if w is not a leaf then $colour\_blue(w)$;

Now we define invariant (I) which we will show to be true before and after every execution of Algorithm B.

(I) : (I.1) Blue nodes are small.

(I.2) Red edges always appear as pairs.

(I.3) T is an $(a, 2b)$-tree.

Let rT be any red subtree of T with root v.

(I.4) All nodes w of rT, $w \neq v$, are blocking.

(I.5) rT contains at least one bucket of T.

(I.6) If rT is a nontrivial red tree then, for all buckets B of rT, r_B points into rT.

Lemma 3.1 Invariant (I) holds true after each step of Algorithm B. In particular, all nodes of T always have size at most $2b$, and we always split big nodes into only two small nodes.

Proof : By induction.

Initially, (I) is true because in T_0 all edges are blue and all nodes are small. So suppose that, after some time, we have a tree T and want to insert a new key into bucket B. Let r_B point to node v.

(B.1) This step does not affect (I). So (I) is true after (B.1) iff it was true before.

(B.2) We have to consider two cases.

v is small : If v is the root of a red tree rT then rT is coloured blue in (B.2). This destroys one red tree and creates many trivial red trees (the buckets of rT). Therefore (I.3)-(I.6) hold after (B.2). (I.1) is true because we only change the colour of nodes of rT; but v was small by assumption and the other nodes were small by (I.4). And (I.2) is true because both edges of a pair of red edges must belong to the same red tree and hence are both unaffected or both coloured blue.

And if v is not the root of a red tree then nothing happens in (B.2).

v is big : Then (B.2) does not affect (I.1). From (I.1) and (I.4) we conclude that v must be the root of a red tree rT. But then v has at most $2b$ children by (I.3). Hence we can split v into only two small nodes; this, together with the fact that the f-edge of v must be blue, proves (I.2). However, we have to be careful about how to split v. Both new nodes must get at least a children to satisfy (I.3); on the other hand, both nodes must not get too many children because they must become blocking to satisfy (I.4). Lemma 3.3 shows that it is always possible to split v such that both (I.3) and (I.4) are satisfied.

The red tree rT grows by the split; either $father(v)$ becomes its new root, or, if it was a red node of a red tree rT' before, rT becomes part of the bigger tree rT'. In any case, (I.5) and (I.6) hold after the split.

(B.3) If v is the root of T then B is split. This can only occur if v was not split in (B.2). But then, B must be a trivial red tree now : If B was not a trivial red tree before (B.2) then, by (I.6), v must have been a node of the red tree rT which contained B; but then it must have been the root of rT and therefore rT was coloured blue in (B.2). Hence (I) holds after splitting B.

If v is not the root of T then r_B is moved up to $father(v)$. But this can only affect (I.6). With a similar argument as above we conclude that (I.6) is still true after (B.3) : either B was a trivial red tree before (B.3) (and therefore still is afterwards), or v was a node of the red tree rT containing B; but in this case, since v was not recoloured in (B.2), $father(v)$ must also be in rT. □

From this follows immediately

Theorem 3.2 *Algorithm A always maintains an $(a, 2b)$-tree T. It supports neighbor queries in time $(\lceil \log b \rceil + 3) \cdot \log n$ and insertions in time $O(1)$, once the* *position where the key is to be inserted in the tree is known. Also, deletions can be done in time $O(1)$ using the global rebuilding technique.*

Proof : We start with an (a, b)-tree T_0 and always maintain an $(a, 2b)$-tree T by Lemma 3.1. Hence the height of T is always bounded by $\log n$, and, doing a query, we can decide in time $\lceil \log b \rceil + 1$ at which child of an internal node the search must be continued. And in the leaves, we can locate each key in time $2 \log n$ by Lemma 2.1. $\square$

It remains to prove that we can always split a big node into two small nodes satisfying (I.3) and (I.4). This is an easy consequence from the following combinatorial Lemma (if blue edges are coded as $c_i = 1$ whereas pairs of red edges are coded as $c_i = 2$).

Lemma 3.3 Let $2a \leq b$, b even, $k \leq b$ and $c_1, \ldots, c_k \in \{1, 2\}$ with $b < \sum_{i=1}^{k} c_i$. Then an

$$j < k \text{ exists such that } a \leq \sum_{i=1}^{j} c_i\,,\; a \leq \sum_{i=j+1}^{k} c_i\,,\; 2j \leq b \text{ and } 2(k-j) \leq b.$$

Proof : Let $j_1 := \min_j \left(\sum_{i=1}^{j} c_i \geq a\right)$ and $j_2 := \max_j \left(\sum_{i=j+1}^{k} c_i \geq a\right)$. From $\sum_{i=1}^{k} c_i \geq 2a + 1$ and $c_i \in \{1, 2\}$ follows $j_1 \leq j_2$, $j_1 \leq \frac{b}{2}$ and $k - j_2 \leq \frac{b}{2}$. Let $j := \min(j_2, \frac{b}{2})$. Then $j_1 \leq j \leq j_2$, $2j \leq b$ and $2(k-j) \leq b$ (because $k \leq b$). $\square$

4 . Conclusions

We have seen how to implement a simple data structure which supports neighbor queries in time $O(\log n)$ and updates in time $O(1)$. However, as in [LO], our data structure can not be used as a finger tree. Hence, it remains an open problem to obtain a finger tree with only $O(1)$ worst-case update time (which would have many more useful applications). However, using our tree recursively in the buckets (instead of the ordered lists), it is possible to obtain a data structure with $O(\log^* n)$ worst-case update time which also allows finger searches. This matches the best previous bounds ([H79], [HL]).

Another open problem is the question whether the query time can be reduced to $\log n$ (with factor 1); Andersson and Lai recently addressed this problem but could only find an optimal amortized solution ([AL]).

Our data structure seems to depend heavily on special properties of (a, b)-trees. It is not clear how to apply our techniques to other kinds of balanced search trees (e.g. BB[α], AVL,...). Furthermore, as R. Tamassia pointed out, it may be difficult to make our search tree persistent. But we hope that it can be used in all kinds of efficient dynamic data structures (as in the triangulation example in Section 1).

Acknowledgements

We would like to thank Rajeev Raman for stimulating this research and Simon Kahan for reading a preliminary version of this paper.

References

[AL] A. Andersson, T.W. Lai
"Comparison-efficient and write-optimal searching and sorting"
Proc. 2nd ISA 1991, 273–282

[GMPR] L. Guibas, E. McCreight, M. Plass, J. Roberts
"A new representation for linear lists"
Proc. 9th ACM STOC 1977, 49–60

[H79] D. Harel
"Fast updates with a guaranteed time bound per update"
Technical Report, Dept. of ICS, University of California at Irvine, 1979

[HL] D. Harel, G. Lueker
"A data structure with movable fingers and deletions"
Technical Report 145, Dept. of ICS, University of Califonia at Irvine, 1979

[HM] S. Huddleston, K. Mehlhorn
"A new data structure for representing sorted lists"
Acta Informatica **17** (1982), 157–184

[LO] C. Levcopoulos, M.H. Overmars
"A balanced search tree with $O(1)$ worst-case update time"
Acta Informatica **26** (1988), 269–277

[M91] K. Mulmuley
"Randomized multidimensional search trees : Dynamic sampling"
Proc. 7th Symposium on Computational Geometry 1991, 121–131

[O82] M.H. Overmars
"A $O(1)$ average time update scheme for balanced search trees"
Bull. EATCS **18** (1982), 27–29

[O83] M.H. Overmars
"The design of dynamic data structures"
Lecture Notes in Computer Science, Vol. 156, Springer 1983

[OvL] M.H. Overmars, J. van Leeuwen
"Worst-case optimal insertion and deletion methods for decomposable searching methods"
Information Processing Letters **12** (1981), 168–173

[T83] R.E. Tarjan
"Updating a balanced search tree in $O(1)$ rotations"
Information Processing Letters **16** (1983), 253–257

[vE] J. van der Erf
"Een datastructuur met zoektijd $O(\log n)$ en constante update-tijd (in Dutch)"
Technical Report RUU-CS-87-19, Dept. of Computer Science, University of Utrecht 1987