A balanced search tree with O(1) worst-case update time

Christos Levcopoulos and Mark H. Overmars

RUU-CS-87-8
May 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

amortized update time once the position of the key is known. In [4] this result is based on some properties of 2-4 trees. In [5] a bucketing technique is used. Rather than storing single keys in the leaves of a tree, each leaf is allowed to store $O(\log n)$ keys. When such a bucket becomes too large, it is split in two buckets of half the size. This means that a new bucket has to be inserted in the tree. Dividing the cost over updates at which no bucket is split this results in an amortized O(1) method.

The best known worst-case bound is $O(\log^* n)$. This is achieved using a very complicted method by Harel (see [2,3]). In this paper we will describe a quite simple method, based on the bucketing technique of [5] that achieves a worst-case update time of O(1) only. (The same result has recently also been achieved in a completely different, and much more complicated way by van der Erf[8].)

The technique is based on a combinatorial result on piles that is interesting in its own right and might have a number of other applications. Assume we are given an infinite number of piles on which we place stones, and, after each k stones, we split the highest pile into two piles each of half the size. We will prove that after placing kn stones, the highest pile has size at most $O(k \log n)$. We will use this result to show that, with the appropriate update scheme, buckets in the tree will never grow too big.

The paper is organized as follows. In Section 2 we describe the combinatorial result on piles. In Section 3 we describe the search structure with O(1) worst-case update time once the position of the key is known. The query time of the new structure will remain bounded by $O(\log n)$. In Section 4 we give some concluding remarks and directions for further research.

2 Splitting piles.

We consider the following problem. Let S be an infinite array $S(1), S(2), ...$, whose entries are all initially 0. Let $P_1(k,n)$, for any integers k and n greater than 1, be the non-deterministic procedure operating on S as described in figure 1. You can consider $S(1), S(2), ...$ as being a row of piles. We put stones on arbitrary piles and every k steps we take the largest pile and cut it in two piles, both of half the size.

Let $A_1(k,n)$ be the largest integer, such that at some moment during some execution of $P_1(k,n)$, some entry in S can be set to $A_1(k,n)$. The problem is to estimate $A_1(k,n)$ in terms of k and n.

It appears to be a complicated task to estimate $A_1(k,n)$ exactly. Moreover, it is not necessary for our purposes. It is enough to estimate the growth of $A_1(k,n)$ in terms of k and n, preferably within a not too large approximation constant. In this section we show that $A_1(k,n) = O(k \log n)$. More precisely, we show that $A_1(k,n) \leq 4k \times \lceil \log n \rceil$. (It can also be shown that $0.5k \times \lfloor \log n \rfloor \leq A_1(k,n)$. Hence, the bound is tight except for constants. For our purposes the lowerbound

is not important and, hence, it will not be shown here and is left as an exercise to the reader.)

To prove the bound on $A_1(k,n)$ we consider two variations of $P_1(k,n)$, the deterministic version $P_2(k,n)$ shown in figure 2 and another non-deterministic version $P_3(k,n)$ as shown in figure 3.

Let $A_2(k,n)$ be the largest integer, such that at some moment during the execution of $P_2(k,n)$, some entry in S can be set to $A_2(k,n)$. Similar let $A_3(k,n)$ be the largest such integer for $P_3(k,n)$. We will first prove an upperbound for $A_2(k,n)$. Next we prove that $A_2(k,n) \geq A_3(k,n)$ and finally, we prove that $A_1(k,n) \leq 2A_3(k,n) \leq 2A_2(k,n)$.

Lemma 2.1 *The following relation holds:* $A_2(k,n) \leq 2k\lceil \log n \rceil$.

Proof. For convenience, we assume in this proof that n is a power of 2. The results can then be extended in a straight-forward way to any value of n.

Let $G(i)$, $1 \leq i \leq n$, be the contents of the i-th entry of S just before it is depleted by $P_2(k,n)$. In the proof we use the following simple fact:

Fact 1: For $1 \leq i \leq n-1$, the inequality $G(i) \leq G(i+1)$ holds.

If $n \leq 4$, the lemma follows because $P_2(k,n)$ performs only kn incrementation steps, and $kn \leq 2k \times \lceil \log n \rceil$ if $n \leq 4$. So we may assume that $n \geq 8$ (remember we made the assumption that n is a power of 2). We prove by induction that $G(n+1-2^j) \leq 2 \times (\log n - j) \times k$, for $0 \leq j \leq \log n - 1$. The induction is performed inversely, i.e. we start from $j = \log n - 1$ and proceed down to the case $j = 0$.

For the basis of the induction, we show that the inequality $G(n+1-2^{\log n-1}) \leq 2k$ holds, which is equivalent to the inequality $G(1+\frac{n}{2}) \leq 2k$.

Suppose that $G(1+\frac{n}{2}) > 2k$. Then, by Fact 1, it holds that $\sum_{n/2 \leq i \leq n} G(i) > 2k \times \frac{n}{2} = kn$. which would imply that more than kn incrementation steps are performed by $P_2(k,n)$, a contradiction.

For the induction step, we show that for $0 \leq j \leq \log n - 1$, the inequality $G(n+1-2^j) < G(n+1-2^{j+1}) + 2k$ holds.

At the time the entry $n+1-2^{j+1}$ is depleted, no entry has contents greater than $G(n+1-2^{j+1})$. So, because only $k \times (2^{j+1}-1)$ incrementation steps are performed after this depletion, we obtain the chain of inequalities $\sum_{(n-2^{j+1}) \leq i \leq n}(G(i) - G(n+1-2^{j+1}) \leq k \times (2^{j+1}-1) \leq 2k \times 2^j - k$. But if the inequality $G(n+1-2^j) \geq G(n+1-2^{j+1}) + 2k$ would hold, then, by Fact 1, we would obtain $\sum_{(n+1-2^{j+1}) \leq i \leq n}(G(i) - G(n+1-2^{j+1}) \geq \sum_{(n+1-2^j) \leq i \leq n}(G(i) - G(n+1-2^{j+1}) \geq 2k \times 2^j$, which contradicts the previous chain of inequalities. □

Lemma 2.2 *It holds that* $A_2(k,n) \geq A_3(k,n)$.

1. **REPEAT at most n times**
2. **REPEAT at most k times**
3. Let i be some integer;
4. $S(i) := S(i) + 1$
5. **Let j be an integer such that $S(j) \geq S(i)$ for all $i \geq 1$;**
6. **Let l be an integer such that $S(l) = 0$;**
7. **Set $S(l)$ to at most $\lfloor S(j)/2 \rfloor$;**
8. **Set $S(j)$ to at most $\lceil S(j)/2 \rceil$**

Figure 1: The non-deterministic procedure $P_1(k, n)$.

1. $i := 0; j := 1;$
2. **REPEAT n times**
3. **REPEAT k times**
4. **IF $i = n$ THEN $i := j$ ELSE $i := i + 1$;**
5. $S(i) := S(i) + 1$
6. $S(j) := 0;$
7. $j := j + 1$

Figure 2: The deterministic procedure $P_2(k, n)$.

1. **REPEAT at most n times**
2. **REPEAT at most k times**
3. Let i be some integer;
4. $S(i) := S(i) + 1$
5. **Let j be an integer such that $S(j) \geq S(i)$ for all $i \geq 1$;**
6. $S(j) := 0$

Figure 3: The non-deterministic procedure $P_3(k, n)$.

Proof. Let $P_3'(k,n)$ be some deterministic version of $P_3(k,n)$ which achieves to set some entry of S to $A_3(k,n)$. Without loss of generality, we may assume that $P_3'(k,n)$ operates according to the following rules:

(i) It executes line 4 (the incrementation step) a minimum number of times.

(ii) It executes line 6 (the depletion step) m times, where m is the minimum number necessary according to rule (i) to achieve to set some entry to $A_3(k,n)$.

(iii) Once it sets some entry of S to zero in line 6, in the continuation it does not operate on this entry any more.

(iv) The m entries which are set to zero in line 6, (see rule ii) are the entries with index $1, 2, ..., m$ in this order.

By rules (i) and (iv) above, the procedure $P_3(k,n)$ does not change the contents of any entry with index larger than m. By the rules (i), (ii) and (iii), the entry $S(m)$ is equal to $A_3(k,n)$ just before it is set to zero.

Let b_i, respectively b_i', for $1 \leq i \leq m$ be the sum $\sum_{i \leq j \leq m} S(j)$ just before the i-th entry is set to zero by procedure $P_3'(k,n)$, repectively by procedure $P_2(k,n)$. We observe that $b_m = A_3(k,n)$. Hence, to show the proposition, it suffices to show, by induction on i, that $b_i \leq b_i'$ holds, for $1 \leq i \leq m$. For the induction basis, we note that $b_1 \leq b_1' = k$. Now assuming inductively that for some $i \leq m$ we have $b_{i-1} \leq b_{i-1}'$, it suffices to show that $b_i \leq b_i'$. According to the line 5, at the time $S(i-1)$ is depleted by $P_3'(k,n)$, it is not smaller than the average contents of the entries $i-1, i, ..., m$, i.e. it holds $S(i-1) \geq \lceil b_{i-1}/(m-i+1) \rceil$. This gives us a way to estimate an upper bound on b_i in terms of b_{i-1} for $1 \leq i \leq m-1$. We obtain $b_i \leq b_{i-1} + k - \lceil b_{i-1}/(m-i+1) \rceil$. We observe that the right-hand side of the latter inequality does never decrease when b_i increases. Hence, by the induction hypothesis, we obtain $b_i \leq b_{i-1}' + k - \lceil b_{i-1}'/(m-i+1) \rceil$. On the other hand, by the definition of $P_2(k,n)$ we derive that $b_i' = b_{i-1}' + k - \lceil b_{i-1}'/(m-i+1) \rceil$. By combining this equality with the latter inequality, we get $b_i \leq b_i'$. □

Lemma 2.3 *It holds that* $A_1(k,n) \leq 2 \times A_3(k,n)$.

Proof. The proof is by contradiction. By supposing that for some k and n the inequality $A_1(k,n) > 2 \times A_3(k,n)$ holds, we show how $P_3(k,n)$ can "imitate" $P_1(k,n)$ to set some variable to an integer greater than $A_3(k,n)$.

To be able to distinguish the values of the entries of the array depending on which of the two procedures is operating, we let $P_3(k,n)$ use another array, which we call S_3. The procedure $P_3(k,n)$ imitates an execution of $P_1(k,n)$ which sets some entry to a value greater than $2 \times A_3(k,n)$ as follows.

For any integer i, when $P_1(k,n)$ increments the i-th entry of S, and the resulting value of the entry is $> A_3(k,n)$, then $P_3(k,n)$ increments the i-th entry of S_3, otherwise it leaves the array S_3 unchanged for this step. When $P_1(k,n)$ decreases the value of the i-th entry, then $P_3(k,n)$ sets the i-th entry of S_3 to zero, if it is not already zero. The procedure $P_3(k,n)$ continues in this way until $P_1(k,n)$ sets some entry, say the j-th entry of S, to $1 + 2 \times A_3(k,n)$, in which case $P_3(k,n)$ increments the j-th entry of S_3 and stops.

We claim that when $P_3(k,n)$ stops, the equality $S_3(j) = 1 + A_3(k,n)$ holds. To see this, we can conclude by straight-forward induction that after every step of $P_1(k,n)$ and the corresponding step by $P_3(k,n)$, it holds that $S_3(i) = \max(0, S(i) - A_3(k,n))$ for all $i \geq 1$. (Note that before $S(j)$ is set to $2 \times A_3(k,n)$, no entry in S has value greater than $2 \times A_3(k,n)$ and, hence, when $P_1(k,n)$ sets an entry to some value at line 7 or 8, this value is never greater than $A_3(k,n)$. Hence, $P_3(k,n)$ correctly sets in this case the corresponding entries in S_3 to zero, if they are not already zero.) □

Theorem 2.4 *It holds that* $A_1(k,n) \leq 4k \times \lceil \log n \rceil$.

Proof. This follows immediately from the above lemmas. According to Lemma 2.3 $A_1(k,n) \leq 2A_3(k,n)$. According to Lemma 2.2 $A_3(k,n) \leq A_2(k,n)$ and according to Lemma 2.1 $A_2(k,n) \leq 2k \times \lceil \log n \rceil$. The bound follows. □

3 The data structure.

In this section we will describe the data structure with O(1) worst-case update time once the position of the inserted or deleted key is known. The query we will treat is the so-called *neighbor query*: given a key K, if it is in V report it, otherwise, report one of the two neighbors in V according to the given order. We will first describe a partial result.

Lemma 3.1 *Let V be a set of at most n' keys, there exists a structure to store V such that insertions can be carried out in $O(1)$ worst-case time when the position of the key is known and neighbor queries can be carried out in time $O(\sqrt{n'})$.*

Proof. We store V as a list of lists. We split V, ordered, in subsets $V_0, \ldots, V_j$ such that

$$\begin{aligned} |V_i| &< \lceil 2\sqrt{n'} \rceil \quad 0 \leq i \leq j \\ |V_i| &\geq \lceil \sqrt{n'} \rceil \quad 1 \leq i \leq j-1 \end{aligned}$$

Initialy we take care that no V_i has size larger than $\lceil \sqrt{n'} \rceil$. Each subset V_i we store in a doubly linked list L_i. With each list we keep a header, containing its size and a pointer f_i to its first element. Each element in L_i has a pointer to the header of the list it is in. (In fact, this is not completely true. At some stages the keys will point to a list that is under construction but, while this list is under construction it will point to the old list. Hence, the keys will have a pointer to a pointer to the list they are in.) We also make a doubly linked list L' that stores the lists $L_0, \ldots, L_j$ in the right order. Clearly L' has at most $O(\sqrt{n'})$ entries.

To perform a neighbor query with a key K, we walk along the list L'. For each list L_i we check f_i to see whether the first element in L_i is smaller or larger (or equal) to K. In this way we can locate in time $O(\sqrt{n'})$ the list L_i K should be in

when present. Next we walk along L_i to locate K or one of its neighbors. This again takes time $O(\sqrt{n'})$.

Assume we want to insert a key K. We have a pointer to the key in some list L_i where K has to be inserted after. Performing the insertion in the doubly linked list is no problem and clealy takes time O(1). Next we increase the size of L_i by one. (We can reach this size field because we have a pointer to the list or a pointer to a pointer to the list.)

When the size of L_i gets larger than $\lceil 1.5\sqrt{n'} \rceil$ we start to find the middle of the list (such that it can be split when it becomes too large.) To this end we initiate a new list header that will later contain the new half list and we set a pointer m_i that points to f_i. The new list header gets a pointer to the old header. With each next insertion in L_i we do the following. We check whether the insertion is before, or after m_i in the list. When it is before m_i we move m_i one key to the right. Otherwise we move it two keys to the right. When m_i passes a key we make its pointer point to the new list header. It is easy to see that in this way, at the moment the size of L_i becomes $\lceil 2\sqrt{n'} \rceil$ m_i points to the middle element of L_i and all elements before m_i point to the new list header. At this moment we split the list before m_i in two, adapt the size fields and add the new list header to L'. This takes time O(1). □

To improve the query time, we will split the set in bags of size at most $O(\log^2 n)$ and store each bag in the way described above. The bags we store in a balanced search tree. We will guarantee that bags never become too large by splitting every $\log n$ updates the largest bag at the moment. To this end we must be able to maintain the largest bag. The following lemma shows that this is possible.

Lemma 3.2 *Let V be a set of at most n' objects and assume each object has been assigned a value between 0 and $\log^2 n'$. We can maintain V such that insertions of objects (with value) take time $O(\log n')$, a value can be incremented by 1 in time O(1), and the object with maximal value can be determined and deleted in time O(1).*

Proof. We again use a list of lists (of lists). We split the set V in subsets V_i in which V_i contains all objects with value between $(i-1)\log n'$ and $i \log n'$. L is a doubly linked list that contains the non-empty sets V_i in order. Each V_i is structured in the following way. We construct a doubly linked list L_i that contains all different values of elements in V_i in order. For each such value v we construct a doubly linked list L_v of objects with value v. For each L_i we have pointer to the smallest and largest element. Clearly L and each L_i has at most $\log n'$ entries. The different operations are performed in the following way:

To perform an insertion with an object p with value v we search for the appropriate entry $L_{\lceil v/\log n' \rceil}$ in L. If it does not exist we create it. Next we search for the entry v in this list. If it does not exist we create it. Finally we add p to L_v. This obviously takes time $O(\log n')$ because we have walk along two lists of size at most $\log n'$.

To increment the value v of an object p we first remove p from L_v. If L_v becomes empty we remove it from its list L_i. If L_{v+1} exitsts we add p to it. Otherwise we first create L_{v+1} either in L_i or L_{i+1}. This might lead to the creation of L_{i+1}. If L_i is now empty we remove it. Because L_{v+1} can be found in time O(1) an increment takes time O(1). (The same result applies for a decrement.)

Finding an object with maximal value can be done by taking the last list in L and in it finding the last entry. Obviously this can be done in O(1) and the element can be deleted in O(1) as well. □

Lemma 3.3 *Let V be a set of between $n_0/2$ and $2n_0$ keys, there exists a structure to store V, using $O(n_0)$ storage such that insertions can be carried out in $O(1)$ worst-case time when the position of the key is known and neighbor queries can be carried out in time $O(\log n_0)$.*

Proof. As stated above we split the set in a number of bags. Initially, each bag contains one key. We structure the bags as a list of lists, as described in the proof of Lemma 3.1. The bags we store in the leaves of a balanced binary search tree T.

To perform a query we first search in T to locate the appropriate bag. This clearly takes $O(\log n_0)$ time because there are $O(n_0)$ bags. Next we search in the bag. This will take time $O(\sqrt{n'})$ where n' is the number of points in the bag. As we will guarantee that the bag size is bounded by $O(\log^2 n_0)$ the query time will be bounded by $O(\log n_0)$.

To guarantee that bags do not get too large, we use the result from the previous section. Every $\log n_0$ insertions we split the largest bag in two equal halves. Hence, the bag size will be bounded by $O((\log n_0) \times (\log n)) = O(\log^2 n_0)$. (Note that $\log n_0$ is constant and, hence, the results from the previous section are applicable.) Splitting a bag can easily be done in $O(\log n_0)$ time. The new bag has to be inserted in the tree T which takes another $O(\log n_0)$. This work we can divide over the next $\log n_0$ insertions at which no bag is split. This makes for O(1) work per insertion.

There is a problem left. To be able to determine which bag to split we have to maintain sizes of bags. To this end, every key has to know in which bag it is. We cannot maintain this information for each key because it would take too much time to split the bag. To solve this problem we keep with each list L_i in the bag a pointer to the bag. As each key can know in which list it is in O(1) time, it can find this pointer in O(1) time and, hence, knows its bag in O(1) time. These bags with their sizes we maintain in a structure as described in Lemma 3.2. Updating this structure takes O(1) per insertion of a key. Determining the largest bag now takes time O(1). Inserting the two new bags of half the size takes time $O(\log n_0)$ which can be spread over the next insertions.

So the insertion procedure looks as follows: First we insert the key in its bag. Next we update the size of the bag. When it is time to start a split ($\log n_0$ insertions

have taken place) we locate the largest bag and initiate the split. Otherwise we spend O(1) work on the splitting that is taking place. It is easy to see that in this way insertions take O(1) time and bags will never become larger than $O(\log^2 n_0)$.

The storge bound follows in a trivial way. □

It remains to show that we can maintain the structure when n grows arbitrary and when we perform deletions. To this end we use a general technique called *global rebuilding* by Overmars[6] (see also [7]). We won't describe the method in detail. See [6] for more information.

To delete a key we just remove it from the list L_i it is in. When the list L_i is empty we remove it from the bag. When the bag is empty, we remove the corresponding leaf from the tree T. We do not change any information about sizes and do not rebalance the tree when removing a leaf. Hence, the insertion procedure is in no way disturbed. Such a deletion takes O(1) time and does not increase the query time or insertion time. Hence, it is *weak* (see [6]). To restore the balance we will sometimes rebuilt the whole structure.

Assume we construct the whole structure for $|V| = n_0$. Now we insert or delete keys in the ways described above, in time O(1) each, as long as $(1 - 1/3)n_0 \leq n \leq (1 + 1/3)n_0$. At the moment this relation no longer holds, we start constructing a new structure with $n_0 = n$. We will spread the work for rebuilding over the next $n/6$ updates. This will guarantee that the structure will be ready before $n < n_0/2$ or $n > 2n_0$. Unfortunately, these $n/6$ updates will have to be performed on the new structure as well. To this end, we speed up the construction of the new structure and take care that it is ready within $n/12$ updates. With the next $n/12$ updates, we perform two updates on the new structure. In this way, the new structure will be up to date after $n/6$ updates and can take over from the old structure. Because on the new structure at most $n/6$ updates have been performed by the time it takes over we do not have to start constructing a new new structure earlier. Hence, we are always busy constructing at most one new structure. (For details of this method see [6].)

Theorem 3.4 *Let V be a set of keys, there exists a structure to store V, using $O(n)$ storage, such that insertions and deletions can be carried out in $O(1)$ worst-case time when the position of the key is known and neighbor queries can be carried out in time $O(\log n)$, where n is the current number of keys in V.*

Proof. In our new method, an update consists of an update on the current structure and some work on the new structure. The update in the current structure takes time O(1). The work in the new structure either consists of two updates which clearly takes O(1) as well, or some work on the construction. It is easy to see that the structure can be constructed in time $O(n)$ when the old structure is available (i.e., when the keys are ordered). As this work is spread over $n/12$ updates, it makes for O(1) per update.

The storage bound follows from the previous lemma and the fact that we always have at most one current structure and one structure under construction. □

[8] van der Erf, J., *Een datastructuur met zoektijd $O(\log n)$ en constante updatetijd* (in Dutch), Techn. Rep. RUU-CS-87-19, Dept of Computer Science, University of Utrecht, 1987.