Amortized Analysis

Michiel Smid*

February 3, 2004

1 Introduction

In general, the running time of an algorithm depends on the efficiency of the data structure that is used to implement it. The algorithm performs, among other things, a sequence of operations on the data structure. In such cases, we are interested in the *total* time for a sequence of operations. In general, the time to perform a single operation may fluctuate—some operations may be "cheap", whereas others may be "expensive". What we want is that the total amount of time for all operations is not too large. Of course, this means that there should not be too many expensive operations. For example, if each expensive operation is followed by a large number of cheap ones, then on the average, one operation is still cheap.

We want to be able to analyze the time for a single operation, *averaged* over a sequence of operations. In these notes, we introduce the technique of *amortized analysis* as a tool for achieving this. We illustrate the technique for the implementation of a binary counter, and for a special class of balanced binary search trees, the so-called BB[α]-trees.

Definition 1 Suppose we have a data structure that supports certain operations. Let $T(n)$ be the worst-case time for performing any sequence of n such operations on this data structure. Then the *amortized time* per operation is defined as $T(n)/n$.

Equivalently, if the amortized time is $U(n)$, then *any* sequence of n operations takes at most $n \cdot U(n)$ time.

*School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6. E-mail: `michiel@scs.carleton.ca`.

2 A simple example: implementing a binary counter

We want to maintain a variable A, whose initial value is zero, under the following operation:

- *increment*, i.e., $A := A + 1$.

The data structure we use is a list

$$< \dots, a_3, a_2, a_1, a_0 >$$

containing the binary representation of the counter A. That is, $a_i \in \{0, 1\}$ for all $i \geq 0$, and $A = \sum_{i \geq 0} a_i 2^i$. Observe that it suffices to maintain a list of length $\lfloor \log A \rfloor + 1$, because $A = \sum_{i=0}^{\lfloor \log A \rfloor} a_i 2^i$. Hence, for all $i > \lfloor \log A \rfloor$, $a_i = 0$. The operation *increment* is implemented as follows.

$a_0 := a_0 + 1$;
$i := 0$;
while $a_i = 2$
do $a_i := 0$;
  $a_{i+1} := a_{i+1} + 1$;
  $i := i + 1$
endwhile

We define the time for this operation to be one plus the number of iterations of the while-loop. In other words, the time for this operation is equal to the number of bits that change.

Let n be a positive integer. We want to analyze the total time for a sequence of n *increment* operations. Hence, the counter A runs from zero to n.

Observe that the time complexity of *increment* fluctuates. To increment the counter from 47 to 48, five bits are changed and, hence, the operation takes five units of time. To increment the counter from 48 to 49, one bit is changed, and the operation takes only one time unit.

A	$=$	a_5	a_4	a_3	a_2	a_1	a_0
47	$=$	1	0	1	1	1	1
48	$=$	1	1	0	0	0	0
49	$=$	1	1	0	0	0	1

Here is a first upper bound on the time for a sequence of n *increment* operations. Let $0 \leq k \leq n-1$, and consider the k-th *increment* operation, i.e., A is incremented from k to $k+1$. During this operation, at most $\lfloor \log(k+1) \rfloor + 1$ bits are changed. Hence, the total time for the n *increment* operations is bounded from above by

$$\sum_{k=0}^{n-1} (\lfloor \log(k+1) \rfloor + 1) \leq n \log n + n. \tag{1}$$

Note that this is an *upper bound.* As it turns out, this upper bound is much too large.

We will give three methods to prove an $O(n)$ bound on the time complexity of the sequence of operations. Hence, according to Definition 1, the *amortized time* per *increment* operation is $O(n)/n = O(1)$, a constant, independent of the number of operations. That is, during any sequence of operations, we spend, on the average, a constant amount of time per operation.

2.1 The first method: counting carefully

In the upper bound (1) above, we charged $\lfloor \log(k+1) \rfloor + 1$ for the k-th *increment* operation. For certain values of k, this is indeed the time needed for the k-th operation. For example, if k has the form

$$k = 2^{j-1} + 2^{j-2} + \cdots + 2^2 + 2^1 + 2^0,$$

then $j + 1 = \lfloor \log(k+1) \rfloor + 1$ bits are changed during the k-th *increment* operation. (Observe that $k = 2^j - 1$.) For most values of k, however, the k-th *increment* operation takes much less than $\lfloor \log(k+1) \rfloor + 1$ units of time.

To be more precise, if k is even, then the k-th *increment* operation takes one time unit, because only one bit is changed. During n *increment* operations, this case occurs $\lceil n/2 \rceil$ times (because k runs from 0 to $n-1$).

If k is odd, but $k \equiv 1 \bmod 4$, then two bits are changed, because the counter is incremented from

$$< \ldots, a_3, a_2, 0, 1 >$$

to

$$< \ldots, a_3, a_2, 1, 0 > .$$

During n *increment* operations, this occurs at most $\lceil n/4 \rceil$ times.

In general, let $i \geq 1$. During n *increment* operations, it happens at most $\lceil n/2^i \rceil$ times that exactly i bits are changed. This happens if the counter is incremented from

$$< \ldots, a_{i+1}, a_i, 0, \underbrace{1, 1, \ldots, 1}_{i-1} >,$$

to

$$< \ldots, a_{i+1}, a_i, 1, \underbrace{0, 0, \ldots, 0}_{i-1} > .$$

It follows that the total time for the sequence of n *increment* operations is equal to

$$\sum_{k=0}^{n-1} (\text{number of bits changed during the } k\text{-th } \textit{increment} \text{ operation})$$

$$\begin{aligned}
&\leq \sum_{i=1}^{2+\lfloor \log(n-1) \rfloor} i \cdot \lceil n/2^i \rceil \\
&= O\left(\sum_{i=1}^{2+\lfloor \log(n-1) \rfloor} n \cdot \frac{i}{2^i} \right) \\
&= O\left(n \sum_{i=1}^{\infty} \frac{i}{2^i} \right) \\
&= O(n),
\end{aligned}$$

which is exactly what we wanted to show.

This concludes the first method. Basically, we just counted the total number of iterations in a more clever way than we did before.

2.2 The second method: using a potential function

For $0 \leq k \leq n-1$, let c_k be the number of bits changed during the k-th *increment* operation. Then the total time for the sequence of n *increment* operations is equal to

$$\sum_{k=0}^{n-1} c_k.$$

The problem is to estimate this summation.

The *potential method* is an "easy" method for finding such an estimate. The method is easy to understand, once you see it. It is not always easy, however, to apply it, i.e., to find a solution by yourself.

We start with a general description of the potential method. Then, we illustrate it for the binary counter algorithm.

Consider a data structure on which we perform a sequence of n operations. The initial data structure is denoted by D_0. For $k = 0, 1, \ldots, n-1$, let c_k be the cost of the k-th operation, and let D_{k+1} be the data structure that results by performing the k-th operation on D_k. Hence, the total cost for the n operations is $\sum_{k=0}^{n-1} c_k$.

Let Φ be a function that maps the data structure D_k to a real number $\Phi(D_k)$, $0 \leq k \leq n$. This function is called the *potential function*, and we call $\Phi(D_k)$ the *potential associated* with D_k.

Given this function Φ, we define

$$\hat{c}_k := c_k + \Phi(D_{k+1}) - \Phi(D_k),\ 0 \leq k \leq n-1.$$

That is, $\hat{c}_k$ is the sum of the *actual cost* of the k-th operation and the *increase in potential* due to this operation. We can rewrite the total cost for the n operations as follows:

$$\begin{aligned}\sum_{k=0}^{n-1} c_k &= \sum_{k=0}^{n-1} (\hat{c}_k - \Phi(D_{k+1}) + \Phi(D_k)) \\ &= \Phi(D_0) - \Phi(D_n) + \sum_{k=0}^{n-1} \hat{c}_k.\end{aligned}$$

Our goal is to estimate $\sum_{k=0}^{n-1} c_k$, which may be difficult, because the values c_k may be difficult to determine. Suppose that our potential function satisfies

(P.1) $\Phi(D_0) = 0$, and

(P.2) $\Phi(D_k) \geq 0$ for all $0 \leq k \leq n$.

Then,

$$\sum_{k=0}^{n-1} c_k = \Phi(D_0) - \Phi(D_n) + \sum_{k=0}^{n-1} \hat{c}_k \leq \sum_{k=0}^{n-1} \hat{c}_k,$$

i.e., the summation $\sum \hat{c}_k$ is an upper bound on the total cost for the sequence of n operations.

Note that this is true for *any* function Φ that satisfies (P.1) and (P.2). The trick is to define a potential function Φ such that the corresponding summation $\sum \hat{c}_k$ can be computed, or at least estimated, easily.

Remark 1 The amortized cost per operation is equal to

$$\frac{1}{n} \sum_{k=0}^{n-1} c_k \leq \frac{1}{n} \sum_{k=0}^{n-1} \hat{c}_k.$$

In many applications, the values of c_k fluctuate, whereas the values of $\hat{c}_k$ are (roughly) equal for all k. In such cases, the amortized cost per operation is bounded from above by $\hat{c}_k$. Therefore, $\hat{c}_k$ is often called the amortized cost of the k-th operation.

Let us apply the potential method to our binary counter problem. In this case, D_k is the list containing the binary representation of the counter A. We saw already that c_k, the cost of the k-th *increment* operation, is equal to the number of bits that are changed when increasing the counter from k to $k+1$. If immediately before this operation, the counter has the form

$$< \ldots, a_{i+1}, a_i, 0, \underbrace{1, 1, \ldots, 1}_{i-1} >,$$

then $c_k = i$.

We define the potential function Φ as follows. Let

$$< \ldots, a_3, a_2, a_1, a_0 >$$

be the list of the data structure D_k. Then

$$\Phi(D_k) := \sum_{j \geq 0} a_j,$$

i.e., $\Phi(D_k)$ is the number of ones in the binary representation of the counter.

First observe that $\Phi(D_0) = 0$, because we start with the counter being zero. Also, it is clear that $\Phi(D_k) \geq 0$ for all $k \geq 0$. Hence, conditions (P.1) and (P.2) are satisfied.

What about the values $\hat{c}_k$? We have, by definition,

$$\hat{c}_k = c_k + \Phi(D_{k+1}) - \Phi(D_k).$$

Immediately before the k-th *increment* operation, the counter has the form

$$D_k = < \ldots, a_{i+1}, a_i, 0, \underbrace{1, 1, \ldots, 1}_{i-1} >,$$

for some $i \geq 1$. The k-th *increment* results in

$$D_{k+1} = < \ldots, a_{i+1}, a_i, 1, \underbrace{0, 0, \ldots, 0}_{i-1} > .$$

We saw already that $c_k = i$. Clearly, $\Phi(D_{k+1}) - \Phi(D_k) = 1 - (i-1) = 2 - i$. Therefore,

$$\hat{c}_k = i + (2 - i) = 2.$$

It follows that

$$\sum_{k=0}^{n-1} c_k \leq \sum_{k=0}^{n-1} \hat{c}_k = \sum_{k=0}^{n-1} 2 = 2n.$$

Hence, the total time for the sequence of n *increment* operations is equal to

$$\sum_{k=0}^{n-1} c_k \leq 2n,$$

or, equivalently, the amortized time per *increment* operation is bounded from above by two. Observe that we have used the "correct" potential function Φ, because the summation $\sum \hat{c}_k$ is easy to compute.

Remark 2 The difficulty of the potential method is finding the "correct" potential function Φ. Once the right function has been found, the method can often be applied easily.

2.3 The third method: how to pay for all operations

Consider again our binary counter problem. If we increment the counter from

$$< \ldots, a_{i+1}, a_i, 0, \underbrace{1, 1, \ldots, 1}_{i-1} >$$

to

$$< \ldots, a_{i+1}, a_i, 1, \underbrace{0, 0, \ldots, 0}_{i-1} >,$$

then we spend an amount of time that is equal to the number of bits that have changed; i in this case.

Assume that we have to pay one dollar for changing one bit. Then the total time for n *increment* operations is equal to the total amount of dollars we pay for all bit-changes. Suppose that we pay according to the following scheme:

1. If a bit is changed from 0 to 1, then we pay *two* dollars. The receiver takes one dollar for the bit-change, and puts the other dollar into an account.

2. If a bit is changed from 1 to 0, then we do not pay *anything*; the receiver takes one dollar from the account.

Claim 1 *If we pay according to this scheme, then we pay for every bit-change that occurs during the sequence of increment operations.*

Proof. Consider a bit-change in which a 0 is changed into a 1. By our payment scheme, we pay the one dollar for this bit-change. In fact, we pay two dollars, but the other one is put into the account.

Consider a bit-change in which a 1 is changed into a 0. By our payment scheme, we do not pay the one dollar for this bit-change at this moment. Why not? Well, just before this bit-change, the bit is equal to 1. Consider the moment when this bit was set to 1. At that moment, it was changed from 0 to 1, and we have put one dollar into the account at that moment. So we have payed for the current bit-change from 1 to 0 already. (Observe that the one dollar needed to pay for the current bit-change is in the account.) ∎

In other words, the idea of our payment scheme is as follows.

- If we change a bit from 0 to 1, then we pay one dollar for this bit-change, and we pay one extra dollar *in advance* for the bit-change later on in which this bit is changed from 1 to 0 again.

How many dollars do we pay during n *increment* operations if we pay according to our scheme? During one *increment* operation, only one bit is changed from 0 to 1. Hence, we pay only two dollars per operation and, therefore, we pay overall $2n$ dollars. This proves that the total time for performing n *increment* operations is bounded from above by $2n$.

Let us compare this payment scheme with the potential method.

- Just before the k-th *increment* operation, the amount of dollars in the account is exactly equal to the number of ones in the binary representation of k, which is the value of $\Phi(D_k)$.
- Initially, the counter is zero and the account is empty. This corresponds to the fact that $\Phi(D_0) = 0$.
- The account is never empty, except just before the 0-th *increment* operation. So for each bit-change from 1 to 0, the receiver can indeed take one dollar from the account.
- We pay exactly two dollars per *increment* operation, which is the value of $\hat{c}_k$.

3 BB[α]-trees

In the course 95.384, you have seen red-black trees. These form a class of balanced binary search trees that can be maintained under insertions and deletions, in logarithmic worst-case time per operation. In this section, we introduce another class of binary search trees that are also balanced, i.e., they have logarithmic height and, hence, we can search for an arbitrary element (or its successor/predecessor) in logarithmic time. To keep these trees balanced, the insertion and deletion algorithms use a sort of brute-force technique. As a result, a single update requires linear time in the worst case. We will see, however, that the amortized times per insertion and deletion are $O(\log n)$.

We will use binary trees as *leaf search trees*. Let S be a subset of $\mathbb{R} \cup \{-\infty, +\infty\}$ of size n. We assume that $-\infty$ and $+\infty$ are both contained in S, and that these elements are never deleted. (This assumption guarantees that searches are always well-defined.) A leaf search tree for S is a binary tree storing the elements of S in its leaves, sorted in increasing order from left to right. Each node of the tree has either two children (in which case it is an *internal node*) or no children (in which case it is a *leaf*). Internal nodes contain information to guide searches. That is, each internal node u contains the following *search information*:

1. $maxl(u)$, which is the largest element of S that is stored in the left subtree of u, and

2. $minr(u)$, which is the smallest element of S that is stored in the right subtree of u.

Note that $maxl(u)$ is the element stored in the rightmost leaf of the left subtree of u. Similarly, $minr(u)$ is the element stored in the leftmost leaf of the right subtree of u.

Exercise 1 (1) Prove that any leaf search tree for S contains exactly $2n-1$ nodes.
(2) Let $x \in \mathbb{R}$. Give an algorithm that finds the smallest element of S that is greater than or equal to x. Similarly, show how to find the largest element of S that is less than or equal to x. The running time of both algorithms should be proportional to the height of the tree.

Remark 3 Storing the elements in the leaves has some advantages. For example, in so-called *multi-level data structures* that are used in computational geometry, leaf search trees are more appropriate.

Clearly, the best performance is obtained if the binary tree is *perfectly balanced*, i.e., for each internal node u, the number of leaves in the left and right subtrees of u differ by at most one. It is easy to see that such a tree has height $O(\log n)$.

Exercise 2 Give an algorithm that constructs a perfectly balanced leaf search tree for S in $O(n \log n)$ time. If the elements of S are sorted already, the running time should be $O(n)$.

3.1 Keeping a binary tree balanced using the partial rebuilding technique

Let us consider the insertion and deletion algorithms for leaf search trees. Let T be such a tree storing a set S of n elements. (Recall that $-\infty, +\infty \in S$.)

Let x be a real number that is not contained in S, and assume that we want to insert x into S. We do the following. First, we search in T for the leftmost leaf that stores an element that is larger than x. Let w be the leaf in which this search ends, and let y be the element of S stored in w. We make w an internal node by giving it two children. The left child is a leaf storing x, whereas the right child is a leaf storing y. Finally, we walk back the path to the root, and update the $maxl(v)$ and $minr(v)$ values of all nodes v on the search path.

The deletion algorithm is similar. Let $x \in S \setminus \{-\infty, +\infty\}$. To delete x from S, we search for the leaf that contains x. Let w be this leaf, let u be the parent of w, and let v be the other child of u. Then we replace the subtree of u by the leaf v, walk from v back to the root and update the search information of the nodes on the search path.

Of course, the problem is how to keep the tree T balanced. We will take the binary tree from the class of BB[α]-trees, which are defined as follows.

Definition 2 Let α be a constant, $0 < \alpha \leq 1/3$, and let T be a binary tree. For any node v of T, let n_v denote the number of leaves in its subtree. The tree T is called a BB[α]-*tree*, if for all nodes u and v such that u is a child of v, we have

$$\alpha \leq n_u/n_v \leq 1 - \alpha.$$

Hence, each of the two subtrees of v contains at least αn_v and at most $(1-\alpha)n_v$ leaves.

Exercise 3 Prove that for each $n \geq 1$ there is a BB[α]-tree with n leaves. Why do we require that $\alpha \leq 1/3$? Prove that the height of a BB[α]-tree with n leaves is less than or equal to $c_\alpha \cdot \log n$, and determine the "best" constant c_α.

This exercise shows that we can search for an element in a BB[α]-tree in $O(\log n)$ time. How do we rebalance such a tree after an element has been inserted or deleted? The method we use is called the *partial rebuilding technique*.

Let T be a BB[α]-tree storing a set S of n elements in its leaves. With each node v of T, we store, besides the values $maxl(v)$ and $minr(v)$, the number n_v of leaves in its subtree. Here is the algorithm to insert or delete an element x.

Step 1: Search for the leftmost leaf storing a value that is greater than or equal to x.

Step 2: Insert or delete x, as described above, and update the $maxl(v)$, $minr(v)$ and n_v values of all nodes v on the search path.

Step 3: At this moment, we have a leaf search tree, which may not be a BB[α]-tree. In order to rebalance this tree, we again walk back from the leaf where the update took place to the root, and find the *highest* node v that is out of balance, i.e., does not satisfy the BB[α]-property. If there is no such v, then the tree is already a BB[α]-tree and the algorithm terminates. Otherwise, if v exists, let S_v be the set of elements of S that are stored in the subtree of v. Then we replace the subtree of v by a perfectly balanced binary tree for the set S_v.

Exercise 4 Convince yourself that the rebalancing step results in a BB[α]-tree.

What is the running time of the update algorithm? Steps 1 and 2 take $O(\log n)$ time, because at the start of the algorithm, the tree T is a BB[α]-tree. The time for Step 3 depends on the position of node v in the tree. Clearly, if this node is close to the root, rebalancing will take much time. In particular, if v is the root, then by Exercise 2, we spend $O(n)$ time. We will show, however, that the amortized update time is $O(\log n)$. We will give two different proofs of this bound. The first proof follows the "count carefully" method of Section 2.1, whereas the second proof uses the potential method of Section 2.2.

3.2 The amortized update time of BB[α]-trees

3.2.1 The first proof: counting carefully

In order to prove an $O(\log n)$ bound on the amortized update time, we have to guarantee that expensive rebalancing operations do not occur too often. The following lemma makes this precise.

Lemma 1 *Let v be a node in a BB[α]-tree that is in perfect balance. Assume we make a sequence of insertions and deletions in this tree. Let n_v be the number of leaves in the subtree of v at the moment when v gets out of balance for the first time. Then there have been at least $(1-2\alpha)n_v - 1$ updates in the subtree of v.*

Proof. Let n'_v, n'_{lv}, and n'_{rv} be the number of leaves in the subtrees of v, the left child of v, and the right child of v, respectively, at the moment when v is in perfect balance. Assume without loss of generality that $n'_{lv} \leq n'_{rv}$. Then $n'_{lv} = \lfloor n'_v/2 \rfloor$. The fastest way for node v to get out of balance is by deleting elements from its left subtree and inserting elements into its right subtree.

Suppose that, at the moment when v gets out of balance, N_i insertions have taken place in the right subtree of v, and N_d deletions have taken place in its left subtree. Let n_{lv} be the number of leaves in the subtree of the left child of v at the moment when v gets out of balance. Then $n_v = n'_v + N_i - N_d$ and $n_{lv} = n'_{lv} - N_d = \lfloor n'_v/2 \rfloor - N_d$. Since node v is out of balance at this moment, we have $n_{lv}/n_v < \alpha$. It follows that

$$\begin{aligned} \alpha n_v &> n_{lv} \\ &= \lfloor n'_v/2 \rfloor - N_d \\ &\geq n'_v/2 - 1/2 - N_d \\ &= \frac{n_v - N_i + N_d}{2} - 1/2 - N_d \\ &= \frac{n_v - (N_i + N_d)}{2} - 1/2. \end{aligned}$$

Thus $N_i + N_d > (1 - 2\alpha)n_v - 1$, i.e., there have been at least $(1 - 2\alpha)n_v - 1$ updates in the subtree of v. ∎

Using this lemma, we can analyze the amortized running time of the update algorithm for BB[α]-trees. We saw already that Steps 1 and 2 take $O(\log n)$ time, even in the worst case. Hence, it remains to bound the amortized time for Step 3, i.e., the time for rebalancing. We will show that the amortized rebalancing time is $O(\log n)$. This will prove that the entire update algorithm has amortized update time $O(\log n)$.

Consider a fixed node v of the tree, and assume we rebuild the subtree rooted at v. Let S_v be the set of elements that are stored in the subtree of v. Note that $n_v = |S_v|$. Using the old subtree of v, we can, in $O(n_v)$ time, obtain the elements of S_v in sorted order. Then we know from Exercise 2 that we can build a perfectly balanced binary tree for these elements in another $O(n_v)$ time. Hence, the entire rebuilding operation takes $O(n_v)$ time. We say that this visit to node v has *rebalancing cost* $O(n_v)$.

By Lemma 1, there have been at least $(1 - 2\alpha)n_v - 2$ updates in the subtree of v during which this subtree was not rebuild. That is, each of the previous $(1-2\alpha)n_v-2$ visits to node v had rebalancing cost zero. This proves that the $(1-2\alpha)n_v-1$ most recent visits to node v have total rebalancing cost $O(n_v)$. Averaged over these visits, we get an upper bound of

$$\frac{O(n_v)}{(1-2\alpha)n_v-1} = O(1)$$

on the rebalancing cost per visit to node v.

Here is an alternative way to show this. If $n_v < 4/(1-2\alpha)$, i.e., n_v is bounded from above by a constant, then rebalancing at v takes $O(1)$ time, even in the worst case. So we may assume that $n_v \geq 4/(1-2\alpha)$. Assume each visit to node v costs nothing, in case we do not rebalance at v. Otherwise, if we rebuild the subtree rooted at v, we have to pay n_v dollars. Then the total amount of dollars we pay for our visits to v during a sequence of updates is proportional to the total rebalancing cost "caused" by node v.

We pay according to the following scheme. If we visit node v, and do not rebuild v's subtree, then we pay $2/(1-2\alpha)$ dollars. Since this visit is for free, we put this amount into an account. If we rebuild the subtree of v, then we do not pay anything, but take the required n_v dollars out of the account. Does the account contain this amount? By Lemma 1, we did not rebuild v's subtree during the previous $(1-2\alpha)n_v-2$ visits to v. Hence, the account contains at least

$$((1-2\alpha)n_v-2)\cdot\frac{2}{1-2\alpha} \geq \frac{1-2\alpha}{2}n_v\cdot\frac{2}{1-2\alpha} = n_v$$

dollars, where the inequality follows from the fact that $n_v \geq 4/(1-2\alpha)$. The amount of dollars payed by us per visit to node v, which is a constant, is proportional to the amortized rebalancing cost caused by v.

We have proved the following. During an update operation, each node on the search path causes $O(1)$ amortized rebalancing costs. Since the search path contains $O(\log n)$ nodes, we get an $O(\log n)$ upper bound on the amortized rebalancing cost per update. Hence, we have proved the following result.

Theorem 1 *Using the partial rebuilding technique, a BB[α]-tree can be maintained under insertions and deletions in $O(\log n)$ amortized time per operation.*

3.2.2 A proof based on the potential method

In this section we give a cleaner proof of Theorem 1 that uses the potential method of Section 2.2. Of course, the problem is to define the "correct" potential function Φ.

Recall that for each node v of our tree, n_v denotes the number of leaves in its subtree. For v an internal node, let v_l and v_r be its left an right child, respectively. Let

$$\Delta_v := |n_{v_l} - n_{v_r}|,$$

if v is an internal node, and $\Delta_v := 0$, if v is a leaf. We define the potential $\Phi(T)$ of a binary tree T as

$$\Phi(T) := \gamma \sum_{v \in T : \Delta_v \geq 2} \Delta_v,$$

where γ is a positive constant to be determined later.

Suppose we start with a BB[α]-tree T_0 that only stores the elements $-\infty$ and $+\infty$. We perform a sequence of n insert and delete operations, as described in Section 3.1. Let $T_0, T_1, T_2, \ldots, T_n$ be the sequence of BB[α]-trees obtained in this way.

First observe that $\Phi(T_0) = 0$ and $\Phi(T_k) \geq 0$ for all $0 \leq k \leq n$. Hence, conditions (P.1) and (P.2) of Section 2.2 are satisfied. Also, observe that a perfectly balanced binary tree has potential zero.

Let $0 \leq k \leq n-1$, and consider the k-th update operation. Let T'_k be the tree obtained by performing Steps 1 and 2 of the update algorithm on T_k. Then, T_{k+1} is obtained by applying Step 3, the rebalancing step, to T'_k. Assume that T'_k is not a BB[α]-tree, and let v be the highest node of T'_k that is out of balance. Then in Step 3, we rebuild the subtree rooted at v.

We will determine the values c_k and $\hat{c}_k$ defined in Section 2.2. Since the tree T_k stores at most $n+2$ elements, Steps 1 and 2 take $O(\log n)$ time. Step 3 takes $O(n_v)$ time. Hence, there is a constant γ' such that the time c_k for the k-th update operation satisfies

$$c_k \leq \gamma'(\log n + n_v).$$

To determine $\hat{c}_k$, we have to estimate the increase in potential. We do this by considering $\Phi(T_{k+1}) - \Phi(T'_k)$ and $\Phi(T'_k) - \Phi(T_k)$ separately.

During the transformation from T_k into T'_k, the Δ_u values of all nodes u on the search path increase by at most one. All other Δ_u values remain unchanged. It follows that

$$\Phi(T'_k) - \Phi(T_k) \leq h_\alpha \gamma \log n,$$

where h_α is the constant in the $O(\log n)$ bound on the height of a BB[α]-tree.

To estimate $\Phi(T_{k+1}) - \Phi(T'_k)$, consider the node v in T'_k. Because v is out of balance, we have (i) $n_{v_l}/n_v < \alpha$ and $n_{v_r}/n_v > 1 - \alpha$, or (ii) $n_{v_r}/n_v < \alpha$ and $n_{v_l}/n_v > 1 - \alpha$. Assume w.l.o.g. that (i) holds. Then,

$$n_{v_r} - n_{v_l} > (1 - \alpha)n_v - \alpha n_v = (1 - 2\alpha)n_v.$$

If T_v and T'_v denote the subtrees of T_{k+1} and T'_k rooted at v, respectively, then

$$\Phi(T_{k+1}) - \Phi(T'_k) = \Phi(T_v) - \Phi(T'_v).$$

Since T_v is perfectly balanced, its potential is zero. Moreover,

$$\Phi(T'_v) \geq \gamma \Delta_v = \gamma |n_{v_l} - n_{v_r}| = \gamma(n_{v_r} - n_{v_l}) > \gamma(1 - 2\alpha)n_v.$$

Hence,

$$\Phi(T_{k+1}) - \Phi(T'_k) < -\gamma(1 - 2\alpha)n_v.$$

Putting everything together, we have shown that

$$\begin{aligned}\Phi(T_{k+1}) - \Phi(T_k) &= (\Phi(T_{k+1}) - \Phi(T'_k)) + (\Phi(T'_k) - \Phi(T_k)) \\ &< -\gamma(1 - 2\alpha)n_v + h_\alpha \gamma \log n.\end{aligned}$$

This implies that

$$\begin{aligned}\hat{c}_k &= c_k + \Phi(T_{k+1}) - \Phi(T_k) \\ &< \gamma'(\log n + n_v) - \gamma(1 - 2\alpha)n_v + h_\alpha \gamma \log n \\ &= (\gamma' + h_\alpha \gamma) \log n + (\gamma' - \gamma(1 - 2\alpha))n_v.\end{aligned}$$

Observe that we still have to choose the constant γ in the definition of Φ. We take $\gamma := \gamma'/(1 - 2\alpha)$. Then $\gamma' - \gamma(1 - 2\alpha) = 0$ and

$$\hat{c}_k < \gamma' \left(1 + \frac{h_\alpha}{1 - 2\alpha}\right) \log n.$$

Using the results of Section 2.2, it follows that the total time for the n update operations is equal to

$$\sum_{k=0}^{n-1} c_k \leq \sum_{k=0}^{n-1} \hat{c}_k < \sum_{k=0}^{n-1} \gamma' \left(1 + \frac{h_\alpha}{1 - 2\alpha}\right) \log n = O(n \log n).$$

That is, the amortized time per update operation is $O(\log n)$. This gives an alternative proof of Theorem 1.

You may wonder why we introduced BB[α]-trees. Red-black trees can be updated in $O(\log n)$ *worst-case* time using rotations, they are not too hard to implement and fast in practice. In geometric applications, we often use *multi-level data structures*. Such a data structure consists of a binary tree (called the main tree), in which each node contains, besides the two pointers to its children, a pointer to another data structure (called a secondary structure), e.g., a binary tree or a linked list. Examples are range trees, segment trees, and interval trees. For such a data structure, a rotation may be expensive, because some secondary structures have to be rebuilt. Blum and Mehlhorn, however, have shown that BB[α]-trees yield multi-level structures that can be updated efficiently using rotations, in the amortized sense. Moreover, by taking BB[α]-trees and applying the partial rebuilding technique, we get simple update algorithms for multi-level structures that have good amortized time bounds.