DM582 Solutions

Mads Anker Nielsen
madsn20@student.sdu.dk

April 14, 2024

This document contains written solution to exercise problems from the course DM582 (spring 2024). The solutions given here might differ from the solutions discussed in class. In class, we place more emphasis on the intuition leading to the correct answer. Please do not consider reading these solutions an alternative to attending the exercise classes.

References to CLRS refer to the book *Introduction to Algorithms, 4th edition* by Cormen, Leiserson, Rivest, and Stein.

References to KT refer to the book *Algorithm Design, 1st edition* by J. Kleinberg and E. Tardos.

This document will inevitably contain mistakes. If you find some, please report them to me (Mads) so that I can correct them.

Sheet 7

Exercise from course website

Exercise

You know that red-black tree has many good properties. Now we'll add another: We'll show that rebalancing is amortized constant. So, yesterday you just knew that carrying out n operations would lead to at most $O(n \log n)$ rebalancing work. After today, you'll know that it's actually at most $O(n)$ rebalancing work.

Please see the sheet with a more compact representation of the red-black tree rebalancing operations. We refer to the operations in Figure 4 as (Ra) through (Rd) and the ones in Figure 5 as (Ba) through (Be) ('R' for "red" and 'B' for "black"). Half-colored nodes are either red or black. You already know that an insertion may create a red conflict (two consecutive red nodes) and that a deletion may create a black conflict (a doubly-black node - indicated by "-"). No operation creates additional problems but the red or black conflicts may be moved up in the tree until they disappear.

1. Call an operation *finishing* if it removes the conflict. Determine which operations are finishing. For (Ra), consider two cases, depending on whether the parent of the operation (that we don't see in the figure) is red or black.

2. Though (Bb) does not appear to be finishing, I would like to consider it finishing. Why?

3. Refer to a configuration in the tree where a black node has two black children as BBB and a configuration where a black node has two red children as BRR. We define the potential function $\Phi(T) = \#BBB + 2\#BRR$, where $\#BBB$ is the number of BBB configurations in T.

4. If there are k updates, how much can the insertions and deletions themselves plus all the finishing operations increase the potential? Conclude that if we can show that all non-finishing operations decrease the potential by at least one, then rebalancing is amortized constant.

5. Show that (Ba) decreases the potential - be careful with the sur-

roundings to make sure no new configurations in the potential function appears.

6. Show that the non-finishing case of (Ra) decreases the potential. Here you have to look quite a bit at what the surroundings must be.

Suggested solution

1. Looking at the figures, we observe that all operations except (Ba), (Bb), and (Ra) when the parent is red are finishing since they remove the conflict.

2. (Bb) is not finishing but it transforms the tree into a configuration where the conflict is resolved in the next iteration by case (Bc), (Bd), or (Be).

3. Nothing to answer here :).

4. The potential is a function of how many connected subtrees of size 3 have certain properties. Since every insertion, deletion or finishing operation modifies only a constant number of connected subtrees of size 3, the potential can increase by at most a constant amount. Note that one could explicitly calculate exactly how much the potential changes for each operation, but it is not necessary to obtain the desired result that the amortized cost of rebalancing is constant.

   Now, assuming that one unit of potential is enough to pay for any non-finishing operation and any non-finishing operation decreases the potential by at least one, then all non-finishing operations pay for themselves. That is, all non-finishing operations have non-positive amortized cost. If all non-finishing operations have non-positive amortized cost, and the insertion, deletion and finishing operations have amortized cost $O(1)$, then the amortized cost of rebalancing is $O(1)$.

5. We calculate the contribution to the potential from the subtree affected by the operation before and after performing the operation. Figure 1 highlights the relevant configurations.

6. We take the same approach as in the previous part. See Figure 2.

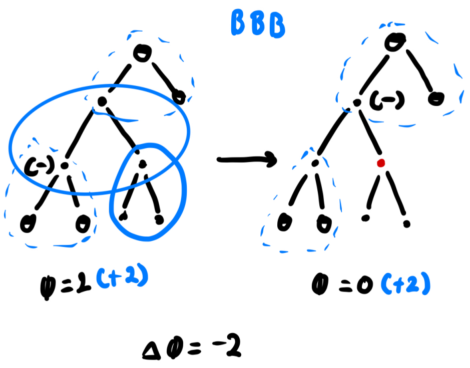

Figure 1

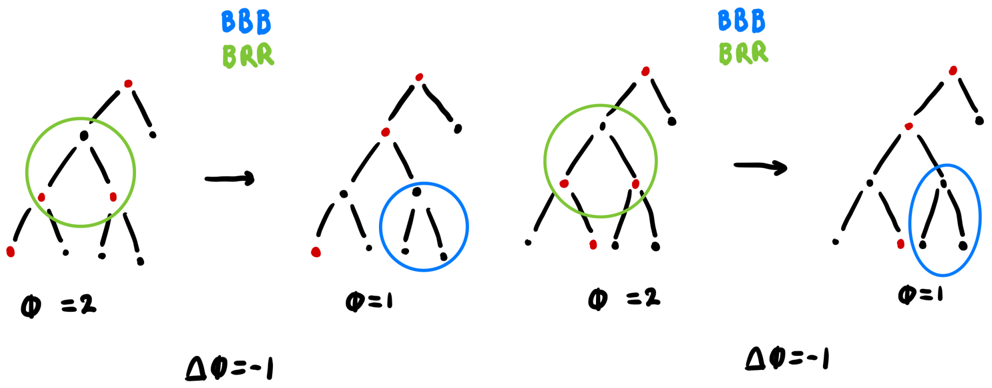

Figure 2

CLRS, Problem 16-2

Exercise

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. You can improve the time for insertion by keeping several sorted arrays. Specifically, suppose that you wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \log_2(n+1) \rceil$, and let the binary representation of n be $(n_{k-1}, n_{k-2}, \dots, n_0)$. Maintain k sorted arrays $A_0, A_1, \dots, A_{k-1}$, where for $i = 0, 1, \dots, k-1$, the length of array A, is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the `SEARCH` operation for this data structure. Analyze its worst-case running time.

b. Describe how to perform the `INSERT` operation. Analyze its worst-case and amortized running times, assuming that the only operations are `INSERT` and `SEARCH`.

Suggested solution

a. One way would be to simply perform a binary search on each of the at most k non-empty arrays. If all arrays are full, the time complexity is $O(k \log n) = O(\log^2 n)$.

b. We start by merging the arrays $A_0, A_1, \dots, A_m$ and the element to be inserted into a single array A_m, where m is the smallest index such that $n_m = 0$. That is, m is the index of the least significant 0 in the binary representation of n. Notice that this simply inserts the new element into the array A_0 if $m = 0$.

   Merging two arrays of size 2^i can be done in 2^{i+1} time using the standard merge algorithm as in merge sort. Thus, first merging the new element and A_0, then the resulting array with A_1, and so on, we can merge all the arrays in time $2^1 + 2^2 + \cdots + 2^{m+1} = \Theta(2^{m+2})$ which is $\Theta(n)$ in the worst case.

   However, we can show that the amortized cost of an `INSERT` is actually $O(\log n)$. We can show this using the aggregate method. Consider any sequence of n insertion operations. A_0 is merged with the new element every operation at cost 2, A_1 is merged with A_0 every second operation at cost 4, in general, A_i is merged with A_{i-1} every 2^i-th operation at cost 2^{i+1}. Let $m = \lfloor \log_2(n) \rfloor$. The total cost of the n insertions is thus

$$\sum_{i=0}^{m} \frac{n}{2^i} \cdot 2^{i+1} = \sum_{i=0}^{m} 2n = 2n(m+1)$$

and hence the amortized cost per operation is $2(m+1) \in O(\log n)$.

Exercise

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$, which gives the number of keys stored in the subtree rooted at x. Let a be a constant in the range $1/2 \leq \alpha < 1$. We say that a given node x is α-balanced if $x.left.size \leq \alpha \cdot x.size$ and $x.right.size \leq \alpha \cdot x.size$. The tree as a whole is α-balanced if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A 1/2-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes 1/2-balanced. Your algorithm should run in $\Theta(x.size)$ time, and it can use $O(x.size)$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced binary search tree takes $\Theta(\log n)$ worst-case time.

c. For the remainder of this problem, assume that the constant a is strictly greater than 1/2. Suppose that you implement INSERT and DELETE as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then you "rebuild“ the subtree rooted at the highest such node in the tree so that it becomes 1/2-balanced. We'll analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, define $\Delta(x) = |x.left.size - x.right.size|$. Define the potential of T as

$$\phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x)$$

where c is a sufficiently large constant that depends on α Argue that any binary search tree has nonnegative potential and also that a 1/2-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $\Theta(n)$ amortized time.

Suggested solution

a. One way to do this is by first obtaining a sorted array of the elements in the subtree rooted at x by doing an in-order traversal of the subtree. Then, we can build a 1/2-balanced subtree by selecting the middle element of the array and then recursively building the left and right subtrees from the elements to the left and right of the middle element.

b. Suppose we take an arbitrary path $P = n_1 n_2 \dots n_k$ form the root to a leaf. Since $n_i.size \le 1/2 \cdot n_i.size$, we have $n_k.size = 1 \le (1/2)^k \cdot n_1.size$ which implies $k \le \log_2 n$. Since P was arbitrary, we conclude that the height of the search tree is $O(\log n)$ and thus searching takes $O(\log n)$ time.

c. Assuming c is nonnegative, the potential of a tree is the sum of positive values multiplied by a positive constant, and is thus nonnegative. A 1/2-balanced tree has potential 0 since $\Delta(x) \le 1$ for all x in the tree, and thus the sum in the definition of $\phi(T)$ is 0.

d. Suppose that the i-th operation is a rebalancing operation and that the actual cost of rebuilding an m-node subtree is m, the amortized cost of rebuilding an m-node subtree is

$$\hat{c_i} = m + \phi(T_i) - \phi(T_{i-1}).$$

By c., the potential of a 1/2-balanced tree is 0, and hence we have $\hat{c_i} = m - \phi(T_{i-1})$. Thus, picking c such that $m \le \phi(T_{i-1})$ we have $\hat{c_i} \le 0$. Let x denote the root of the subtree being rebuilt and let l and r denote the size of the left and right subtrees of x respectively. Without loss of generality, assume that the right subtree is larger or switch the roles of l and r. $\phi(T_{i-1})$ is the sum of $c\Delta(n)$ over all nodes n in the tree such that $\Delta(n) \ge 2$. x is one such particular node, and thus $\phi(T_{i-1}) \ge c\Delta(x) = c|r - l| = c(r - l)$. Now, solving for c such that $c(r - l) \ge m$ we get

$$\begin{aligned} m &\le c(r - l) \\ &= c(r - (m - r - 1)) \\ &= c(2r - m + 1)) \\ &\le c(2\alpha m - m + 1) \end{aligned}$$

which implies $c \ge \frac{m}{2\alpha m - m + 1}$, which is bounded from above by $\frac{1}{2\alpha - 1}$. Thus, setting $c \ge \frac{1}{2\alpha - 1} \ge \frac{m}{2\alpha m - m + 1}$ will make the amortized cost of rebuilding a subtree $O(1)$.