CSC265 F18: Assignment 5

Due: November 21, by midnight

Guidelines: (read fully!!)

- Your assignment solution must be submitted as a *typed* PDF document. Scanned handwritten solutions, solutions in any other format, or unreadable solutions will **not** be accepted or marked. You are encouraged to learn the LaTeX typesetting system and use it to type your solution. See the course website for LaTeX resources. Solutions typed using LaTeX receive 2 bonus marks.
- Your submission should be no more than 6 pages long, in a single-column US Letter or A4 page format, using at least 9 pt font and 1 inch margins.
- To submit this assignment, use the MarkUs system, at URL `https://markus.teach.cs.toronto.edu/csc265-2018-09`
- This is a *group assignment*. This means that you can work on this assignment with *at most one other* student. You are *strongly encouraged* to work with a partner. Both partners in the group should work on and arrive at the solution together. Both partners receive the same mark on this assignment.
- Work on all problems together. For each problem, one of you should write the solution, and one should proof-read and revise it. The first page of your submission must list the *name*, *student ID*, and *UTOR email address* of both group members. It should also list, for each problem, which group member wrote the problem, and which group member proof-read and revised it.
- You **may not** consult any other resources except: your partner; your class notes; your textbook and assigned readings. *Consulting any other resource, or collaborating with students other than your group partner, is a violation of the academic integrity policy!*
- You may use any data structure, algorithm, or theorem previously studied in class, or in one of the prerequisites of this course, by just referring to it, and without describing it. This includes any data structure, algorithm, or theorem we covered in lecture, in a tutorial, or in any of the assigned readings. Be sure to give a *precise reference* for the data structure/algorithm/result you are using.
- Unless stated otherwise, you should justify all your answers using rigorous arguments. Your solution will be marked based both on its completeness and correctness, and also on the clarity and precision of your explanation.

Question 1. (15 marks)

Recall that during an AVL-INSERT(x) operation in an AVL tree, we first insert the key x into the tree as we do in any binary search tree (BST). Then we go up the path from the new node to the root and update the balance factor fields until we either reach a node which becomes perfectly balanced after the insertion, or we have to do a single or a double rotation. (Review the notes on AVL trees.)

This second part of an AVL-INSERT(x) – updating balance factors and doing rotations – can, in the worst case, double the time it takes to insert a new node. Your goal in this question is to show that this is not true in an amortized sense.

Use the potential function method to show that in any sequence of m consecutive AVL-INSERT(x) operations, starting from an empty AVL tree, the total number of balance factor updates is bounded by $O(m)$. I.e. if t_i is the number of nodes whose balance factor is changed during the i-th consecutive insert (not counting the new node), show that $\sum_{i=1}^{m} t_i = O(m)$. Be precise about the potential you are using. Show that the potential is 0 for an empty AVL tree, and is never negative. Derive the claimed bound on $\sum_{i=1}^{m} t_i$ by analyzing the potential differences.

[Solution]

Let T_i be the AVL tree after the i-th insert, and T_0 be the initial empty tree. We define the potential $\Phi(T_i)$ to equal the number of nodes of T_i which have balance factor 0. Clearly this potential starts at 0 and is never negative. We need to bound $a_i = t_i + \Phi(T_i) - \Phi(T_{i-1})$ by a constant.

Let us first consider the case in which no rotation is performed. The new node u inserted into the AVL tree is a leaf, and, therefore, has balance factor 0. This *increases* the potential by 1. Recall from the AVL-tree notes that we update balance factors of the nodes on the path from u to the root until we reach either the root, or a node which had nonzero balance factor before the insertion, and has balance factor 0 after the insertion. No further balance factors need to be updated. Therefore, of the t_i nodes whose balance factors are updated, at least $t_i - 1$ change their balance factor from 0 to ± 1, which decreases the potential by $t_i - 1$, and at most one changes its balance factor from ± 1 to 0, which increases the potential by 1. Then, the net increase in the potential is $\Phi(T_i) - \Phi(T_{i-1}) \leq 3 - t_i$.

The case of a single rotation is similar. Initially we insert the new node u which has balance factor of 0, *increasing* the potential by 1. Then we update the balance factors of a sequence of nodes whose balance factor before the insertion was 0. Let the last such node be v. The parent of v, let's call it w, becomes unbalanced after the insertion, and we perform a single rotation. You can verify, using the AVL Trees lectures notes, that after the rotation the balance factors of v and w change: the balance factor of v becomes 0 again, and the balance factor of w changes from ± 2 (after the insertion but before the rotation) to 0 after the rotation. Therefore, in total, out of the t_i nodes whose balance factor changes, $t_i - 1$ change their balance factor from 0 to ± 1, and one node, w, changes its balance factor from ± 1 to 0. Again we have that the total change in potential is $\Phi(T_i) - \Phi(T_{i-1}) \leq 3 - t_i$.

The case of a double rotation is analogous to the single rotation case: the new node increases the potential by 1; then, out of the t_i nodes whose balance factor changes, $t_i - 1$ change their potential from 0 to ± 1, and one changes its potential from ± 1 to 0. The total increase in potential is $3 - t_i$.

Because in all of the above cases the increase in potential is at most $3 - t_i$, we have that $a_i \leq t_i + 3 - t_i = 3$, and, the total number of balance factor updates performed is bounded by $\sum_{i=1}^{m} a_i \leq 3m$.

Question 2. (20 marks)

We define a simple mergeable heap data structure, called the TREE-HEAP. Unlike Binomial Heaps, a TREE-HEAP is implemented as a single (but *not necessarily binary*) tree. A TREE-HEAP is a tree which stores the keys of the heap elements at the nodes, so that the *min-heap property* is satisfied: the key stored at every node is at most the keys of its children. This is the only condition the TREE-HEAP needs to satisfy!

We will discuss three operations on TREE-HEAP's:

- INSERT(T, x), which inserts a new item with key x;
- UNION(T_1, T_2) which merges the heaps T_1 and T_2 into a new TREE-HEAP T;
- EXTRACT-MIN(T) which returns the heap element with the smallest key in T, and removes it from T.

As usual, we implement INSERT(T, x) by creating a new heap T' with a single node storing the key x, and calling UNION(T, T'). The operation UNION(T_1, T_2) is very simple: we compare the keys of the roots of T_1 and T_2, and make the root with the larger key a child of the root with the smaller key. You can assume that when T_1 becomes a subtree of T_2, it is placed as the leftmost subtree of the root, and vice versa. See Figure 1 for an example.

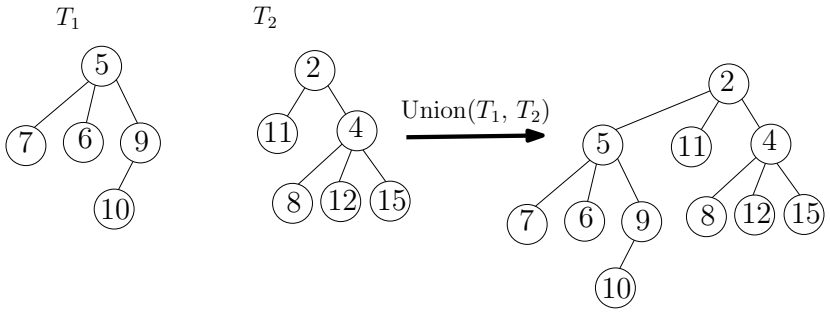

Figure 1: An example of a union of two TREE-HEAP's.

Since a TREE-HEAP is not a binary tree, we implement it using the leftmost child – right sibling representation, similarly to how binomial trees were implemented. In particular, each node u of a TREE-HEAP T has the following fields:

- $u.key$ stores the key of u;
- $u.lchild$ stores a pointer to the leftmost child of u, if u has any children, and NIL otherwise;
- $u.rsibl$ stores a pointer to the sibling of u directly to its right, if u has any siblings to the right, and NIL otherise.

You should make sure that with this representation you know how to implement UNION(T_1, T_2) by changing a constant number of pointers.

The EXTRACT-MIN(T) operation is the trickiest one, and can be very slow if not implemented carefully. Below we explore two options.

Part a.

Suppose we implement EXTRACT-MIN(T) by removing the root of T (and storing its key to return once we are done); then we are left with the subtrees of the root of T: let's call them $T_1, \ldots, T_d$, listed from left to right. Here d denotes the degree of the root of T before it was deleted. Now we successively perform $T'_2 =$ UNION(T_1, T_2); $T'_3 =$ UNION(T'_2, T_3), …, $T'_d =$ UNION(T'_{d-1}, T_d), and return T'_d as the new heap.

For every m, show a sequence of m INSERT and EXTRACT-MIN operations, starting from an empty heap, so that the total time complexity of executing the sequence using the algorithms above is $\Omega(m^2)$. I.e. show that the amortized complexity is $\Omega(m)$ per operation. Justify your answer, and be specific about the exact sequence of operations.

[Solution]

First insert the keys 1 and then $2n, 2n-1, \ldots, n+1$ into the heap (for n to be determined shortly). Then perform INSERT$(T, 2)$, EXRACT-MIN(T), INSERT$(T, 3)$, EXTRACT-MIN(T), etc., until, finally, we end with INSERT(T, n) and EXTRACT-MIN(T) .

Intially the heap contains 1. After inserting $2n$, it contains 1 as the root and $2n$ as its only child. After inserting $2n-1$, it contains 1 as the root, and $2n-1$, $2n$ as its children. You can see (by an easy induction argument), that after inserting $n+1$, the heap contains 1 as the root, and $n+1, \ldots, 2n$ as its children. When we insert 2, the children of 1 become $2, n+1 \ldots, 2n$. Then after EXTRACT-MIN(T), we have a heap with 2 as the root, and children $2n, \ldots, n+1$. After INSERT($T, 3$), the children of 2 are $3, 2n, \ldots, n+1$, and after EXTRACT-MIN(T), the heap has 3 as the new root, with children $n+1 \ldots 2n$. By induction, we can show that after INSERT(T, i) and EXTRACT-MIN(T), the heap has i as root, and children $n+1, \ldots, 2n$ (in this order, or in the reverse order). Every EXTRACT-MIN(T) operation then involves $n-1$ unions, so the n EXTRACT-MIN operations we perform take time $\Omega(n)$ each, or $\Omega(n^2)$ total. The total number of operations in the sequence is $3n$, so we can just set $n = \lfloor m/3 \rfloor$, and we get that the total time complexity for the sequence of operations is $\Omega(m^2)$.

Part b.

Since the algorithm for EXTRACT-MIN(T) above turned out to be too slow, let us try another one. Again we remove the root of T (and store its key to return once we are done); then we are left with the subtrees of the root of T that we removed, denoted $T_1, \ldots, T_d$, which we need to merge into a single tree. We will now do this in two stage. First we merge them in pairs: we run $T'_1 = \text{UNION}(T_1, T_2)$, $T'_2 = \text{UNION}(T_3, T_4)$, ..., $T'_k = \text{UNION}(T_{2k-1}, T_{2k})$, where $k = \lfloor d/2 \rfloor$. If d is odd, then we set $T'_{k+1} = T_d$. This completes the first stage. In the second stage we merge the T'_i trees as we did before: we set $T''_2 = \text{UNION}(T''_1, T'_2)$; $T''_3 = \text{UNION}(T''_2, T'_3)$, ..., $T''_\ell = \text{UNION}(T''_{\ell-1}, T'_\ell)$, where $\ell = \lceil d/2 \rceil$. Finally we return T''_ℓ.

Show that in any sequence of m INSERT and EXTRACT-MIN operations, starting with the empty heap, the amortized cost of an INSERT is $O(1)$, and the amortized cost of an EXTRACT-MIN is $O(\sqrt{m})$. Prove this using the potential function method. Be precise about the potential you are using. Show that the potential is 0 for an empty TREE-HEAP, and is never negative. Derive the claimed bound on the amortized complexity by analyzing the potential differences.

For the above, you can assume that the actual cost of an EXTRACT-MIN(T) equals the number of children of the root of T, and that the actual cost of INSERT(T, x) equals 1.

HINT: Consider assigning every node u of T a potential $\phi(u) = C(1 - \min\{d(u), n^\alpha\})$ for some constants $C > 0$ and $0 < \alpha \leq 1$. Here $d(u)$ equals the number of children of u, and n equals the number of nodes in T. Then define the potential of T as the sum of the potentials of its nodes. Specify precise values for C and α and prove all the necessary properties of the potential. You can use the inequality $(a+b)^\alpha \leq a^\alpha + b^\alpha$, valid for all $a, b \geq 0$ and $0 < \alpha \leq 1$.

[Solution]

We use the potential in the hint with $\alpha = 1/2$ and $C = 2$. The potential is 0 for an empty heap, because in that case the sum is over an empty set. Let's prove that it's always nonnegative for any non-empty heap. We have that $\phi(u) \geq 2(1 - d(u))$, so the potential is

$$\Phi(T) = \sum_{u \in T} \phi(u) \geq 2n - 2 \sum_{u \in T} d(u).$$

Now note that, for $n > 0$, $\sum_{u \in T} d(u)$ equals exactly $n-1$ since each non-root node v contributes exactly one in the sum, as it is counted towards $d(u)$ of its parent u. Then the potential is at least $2n - 2n + 2 = 2$ for $n > 1$, and equals 0 for $n = 0$.

First we claim that an insert can increase the potential by at most 2. We insert a new node, whose potential is either 2, if it becomes a leaf, or at most 0 if it becomes the new root of a heap with at least two elements. The potential of any node u which was previously in T stays the same or decreases, because $d(u)$ stays the same or increases, and $\sqrt{n}$ increases. Therefore, the amortized cost of an insert is at most 3: actual cost of 1 plus potential increase of at most 2.

We analyze EXTRACT-MIN(T) next. Let n be the size of the heap before we extract the minimum. First, removing the root r of T can increase the potential by at most $O(\sqrt{n})$. This is is because $\phi(r) \geq 2(1 - \sqrt{n})$ is removed from the potential, and also the potential of any node u with at least $\sqrt{n}$ children can increase by

at most $2(\sqrt{n} - \sqrt{n-1}) \leq 2$. There are less than $\sqrt{n}$ such nodes, as the sum of all $d(u)$ is $n-1$. Therefore, just removing the root of T can increase the potential by at most $4\sqrt{n} - 2 = O(\sqrt{n})$ as claimed.

Suppose that the root of T before it was removed had d children. Let us focus on the case in which d is even; the case of d odd is analogous. Then every union in the first stage decreases the potential by 2 if the new root u of the merged tree had degree $d(u) \leq \sqrt{n} - 1$ initially. Otherwise the potential is not changed by the union. Let t be the number of roots u_i of $T_1, \ldots, T_d$, respectively, such that $d(u_i) > \sqrt{n} - 1$ before the EXTRACT-MIN(T) was called. Then the decrease in potential after the first stage of unions is at least $2(\lfloor d/2 \rfloor - t) \geq d - 1 - t$, because there $\lfloor d/2 \rfloor$ many unions in the first stage. Using the inequalities

$$t(\sqrt{n} - 1) < \sum_{i=1}^{d} d(u_i) \leq n - 1$$

we have that $t = O(\sqrt{n})$. Then, the second stage of unions can only decrease the potential further. It follows that the potential during an EXTRACT-MIN(T) decreases by at least $d - O(\sqrt{n})$, so the amortized cost is at most $d - d + O(\sqrt{n}) = O(\sqrt{n})$. Finally, since the number of elements in T cannot be bigger than the total number of operations performed, we have $m \leq n$, so the amortized complexity of EXTRACT-MIN(T) is at most $O(\sqrt{m})$ as well.