Homework Assignment 3 (Binary Search Trees and Amortized Analysis)

Solution

Problem 1.

Each node in a sorted binary search tree not only carries information of what element it contains, but also pointers to its parent and its children. To efficiently search for the kth smallest element, we add the size (i.e., the number of elements) of its left subtree to each node’s information. Let $s_i$ denote the size of the left subtree of node $i$.

The procedure to find the kth smallest element is as follows: Define the auxiliary variable $x$ with initial value $x = 0$; $x$ counts the number of nodes that are "cut off" whenever the algorithm proceeds with a right child (see below). Starting at the root, you go through the tree according to the following rules.

- If $s_i + x = k - 1$, the node $i$ contains the kth smallest element and you are done.
- If $s_i + x > k - 1$, you need to find a smaller element and therefore proceed to the left child of node $i$.
- If $s_i + x < k - 1$, you need to find a larger element and therefore proceed to the right child of node $i$. In addition, you increase the value of $x$ by $s_i + 1$.

With this algorithm, you immediately notice when entering the node containing the kth smallest element. Furthermore, you never have to move upside (i.e. backwards) in the tree. In the worst case, the procedure halts when ending up in a leaf. In that case, the kth smallest element indeed is the leaf, i.e. $x = k$ (thus, we have not missed it on the way down the tree). Therefore, the depth of the kth smallest element is exactly the number of steps that have to be taken and thus, the time to find the kth smallest item is linear in tree depth.

A possible implementation in pseudocode reads:

```plaintext
Node *KSMALLEST(Tree T, int k)
    case
        k = (T → s) + 1: return T;
        k < (T → s) + 1: return KSMALLEST(T → l, k);
        k > (T → s) + 1: return KSMALLEST(T → r, k - (T → s) - 1)
    endcase
```
When inserting an element, you always insert it as a leaf. To do so, follow the path down the tree as if you were looking for the element until you end up in a leaf. Make this leaf a node and add the new element as the corresponding child (left/right) to it. This has also been described in the lecture — the more interesting part is how to adjust the $s_i$ accordingly. To achieve this, follow the path from your newly inserted element back up the tree until you hit the root. When arriving at node $i$, do the following:

- If the node you were in before is a right child of node $i$, do nothing.
- If the node you were in before is a left child of node $i$, increase $s_i$ by +1.

Alternatively, we could also update the $s_i$ during the search procedure: Every time we process to the left child of a node, we increment its $s_i$ value by 1.

Deleting elements is the most tricky part. Say that the item that is to be deleted is contained in node $\nu$. Like in the lecture, consider three possibilities:

(i) $\nu$ has no children. Then, just remove it and adjust the $s_i$ similar to the procedure above: Starting out from where $\nu$ was positioned initially, move the tree up to its root. On the way, when approaching any node $i$ from the left, decrease $s_i$ by one.

(ii) $\nu$ has exactly one child. Replace $\nu$ by its child and run the procedure above to get the $s_i$ right.

(iii) $\nu$ has two children. Find the largest element of $\nu$’s left subtree (call it $\mu$), remove it using point (i) or (ii) (it has no right child, since it is the largest element of the left subtree). With this procedure, we already adjust all the $s_i$ correctly; in particular, $s_\nu$ (i.e., the s.value of $\nu$) is updated to $s_\nu^* = s_\nu - 1$. Finally, substitute $\mu$ for $\nu$ and update the value $s_\mu$ to $s_\nu^*$. 
Problem 2.

To implement a queue out of two stacks, I use one stack ($A$) to put items upon and the second one ($B$) to remove items from. Thus:

- To enqueue an item, push it to stack $A$.
- To dequeue an item, there are two possibilities:
  - Stack $B$ is non-empty: To dequeue an item, simply pop it from stack $B$.
  - Stack $B$ is empty: Reshuffle stack $A$ to stack $B$, i.e., pop and push items one by one until stack $A$ is empty. Then, pop the first item of stack $B$.

This procedure guarantees that stack $B$ holds items in reverse order than put upon stack $A$, hence popping from the top gives the "oldest" item from the queue.

The accounting method. By accounting, it is relatively easy to figure out the amortized cost of enqueuing and dequeuing. Suppose that moving an element costs one unit each. Then, each element first has to be placed on stack $A$, then moved from $A$ to $B$ and finally be picked up from $B$. These are a maximum of four steps per item (2×push and 2×pop). If an item is never dequeued, you push it at most twice and pop it once. Therefore, enqueuing has amortized cost 3 and dequeuing has amortized cost 1.

In other words, for each element coming in onto stack $A$ we have to charge 3$: One for the actual cost of pushing and two to be able to pay for popping and pushing the element to stack $B$ if we need to at a later point. Thus, we keep a total of $2|A_n|$ in reserve ($|A_n|$ is the number of elements on stack $A$ after the $n$th operation) to come up for later costs.

A potential function. The above analysis indicates that $\phi_n = 2|A_n|$ is a potential function for the problem:

- If the $n$th operation is enqueue, we have
  $$a_n = c_n + \phi_n - \phi_{n-1} = 1 + 2|A_n| - 2(|A_n| - 1) = 1 + 2 = 3$$
  Thus, as above, the amortized cost of enqueuing is 3.

- If the $n$th operation is dequeue, there are two possibilities:
  - Stack $B$ is not empty and therefore there is no need for reshuffling. Then
    $$a_n = 1 + \phi_n - \phi_{n-1} = 1 + 2|A_n| - 2|A_n| = 1.$$
  - Stack $B$ is empty and therefore we have to move the items from stack $A$ to stack $B$, and then pop the upmost element. Thus
    $$a_n = \underbrace{1 + 2|A_n| + \phi_n - \phi_{n-1}}_{c_n} = 1 + 2|A_n| + 0 - 2|A_n| = 1.$$
  In both cases, the amortized cost of dequeuing is 1.