1 Search Trees (33 points)

We saw in class how to use binary search trees as dictionaries, and in particular how to use them to do insert and lookup operations. Suppose we want to modify the basic binary search tree to also let us perform the following operation (assuming that all keys are distinct): Given a key \( x \), do a version of \( \text{lookup}(x) \) that tells us how many keys are less than \( x \) (we call this the rank of \( x \)).

Note that if our data was fixed, this would be easy. We could simply use a sorted array \( A \). Then to find the rank of \( x \) we simply do a binary search to find \( x \) and then return its position minus 1.

But if we want to handle inserts as well, then one way to do this is through search trees. The goal of this problem is to modify normal binary search trees (nothing fancy like B-trees, red-black trees, AVL trees, etc.) so that the above operation can be done in \( O(\text{depth}) \) time, and inserts can also be done in \( O(\text{depth}) \) time. In particular, you should do the following:

(a) (17 points) Describe an extra piece of information that you will store at each node of the tree, and describe how to use this extra information to do the above rank operation in \( O(\text{depth}) \) time (prove correctness and running time).

Solution: We will store at each node \( v \) the size of the subtree rooted at \( v \).

When we lookup \( x \), if \( x \) is equal to the root then the size of the left subtree is the rank of \( x \). If \( x \) is smaller than the root, then we need to recurse down to the left subtree. If \( x \) is greater than the root, then we need to recurse down to the right subtree, and we further know that all the nodes on the left subtree are smaller than \( x \), hence we store the size of the left subtree each time we need to recurse down to the right. Let \( r \) denote the root of the tree, the lookup algorithm is as follows:

\[
\text{lookup}(x, r) \{
    \text{if } x == r.\text{key} \\
    \quad \text{return } r->\text{left}.\text{size}; \\
    \text{else if } (x < r.\text{key}) \\
    \quad \text{return } \text{lookup}(x, r->\text{left}); \\
    \text{else} \\
    \quad \text{return } \text{lookup}(x, r->\text{right})+r->\text{left}.\text{size+1};
\}
\]
We can prove the correctness by showing that every element which is less than \(x\) will be counted once in the algorithm, and every element which is greater than \(x\) will not be counted. We only prove the smaller part, the greater part is similar.

Consider an element \(y\) which is less than \(x\), and assume \(a\) is the lowest common ancestor of \(x\) and \(y\). From the property of binary search tree we know that \(y\) is in the left subtree of \(a\) and \(x\) is in the right subtree of \(a\) (or one of \(x, y\) is \(a\)), so \(y\) will be counted once the algorithm visits \(a\), because the algorithm adds the size of the left subtree of \(a\) to the result.

Because the algorithm follows the path from the root to \(x\), this algorithm must visit \(a\), which means \(y\) will be counted by the algorithm. In addition, \(y\) will not be counted twice because once it is counted, it is in the left subtree (or it is the root) and the algorithm goes to the right subtree, and \(y\) will not again be in any left subtree of the nodes which the algorithm visits. Therefore proved.

It takes at most \(O(\text{depth})\) of recursions to reach a leaf node, each recursion takes constant time. Hence the lookup operation takes \(O(\text{depth})\) time to get the rank of \(x\).

(b) (16 points) Describe how to maintain this information in \(O(\text{depth})\) time when a new node is inserted (note that there are no rotations on an insert – it’s just the regular binary search tree insert, but you need to update information appropriately). Prove correctness and running time.

**Solution:** Note that the inserted node must be a new leaf. When tracking down the tree, we increase the size by 1 for all the nodes that have been visited. The algorithm is as follows:

```plaintext
insert(x, r) {
    r.size = r.size + 1;
    if (x < r.key)
        if (r→left ≠ null)
            insert(x, r→left);
        else
            r→left := x;
            r→left.size := 1;
    else
        if (r→right ≠ null)
            insert(x, r→right);
        else
            r→right := x;
            r→right.size := 1;
}
```

To see the correctness, all the nodes which contains \(x\) in the subtree should increase their stored value by 1. Which means all the ancestors of \(x\) should increase their stored value by 1. This algorithm visited all the ancestors of \(x\) by following the path from root to the right position of \(x\), and increases their stored value by 1, which is what we want.

It takes at most \(O(\text{depth})\) of recursions to reach a leaf node, and the size information is already updated when we reach the leaf, hence the insert operation takes \(O(\text{depth})\) time.
For example, a bad way to do this would be for every node to store the rank of its key. This information would let us do the rank operation quickly, but maintaining it on an insert might require updating all of the other nodes (which might be much larger than the depth).

2 More counters (34 points)

We saw in class that if we have a binary counter which we increment \( n \) times the total cost (measured in terms of the number of bits that are flipped) is \( O(n) \), i.e. the amortized cost of an increment is \( O(1) \). What if we also want to be able to decrement the counter? Throughout this problem we will assume that the counter never goes negative – at every point in time the number of increments up to that point is at least as large as the number of decrements.

(a) (17 points) Show that it is possible for a sequence of \( n \) operations (increments and decrements) to have amortized cost of \( \Omega(\log n) \) per operation (so the total cost is \( \Omega(n \log n) \)). This should hold even if we start from 0 and the counter never goes negative.

Solution: Let \( m = \lfloor \log(n/2) \rfloor \). Note that \( n/4 \leq 2^m \leq n/2 \). We first do \( 2^m \) increments (starting from 0). After this, the counter has a 1 in the \((m+1)\)st bit, and a 0 in the other \( m \) bits. For the remaining \( n - 2^m \geq n/2 \) operations, we alternate decrements and increments.

Note that in each of the final \( n - 2^m \) operations we have to flip all \( m + 1 \) bits. Thus the total cost for all \( n \) operations is at least

\[
\sum_{i=1}^{n-2^m} (m + 1) \geq \sum_{i=1}^{n/2} \lfloor \log(n/2) \rfloor \geq \Omega(n \log n)
\]

(b) (17 points) To decrease this cost, let’s consider a new way of representing numbers: a redundant ternary number system. A number is represented as a sequence of trits (as opposed to the more usual bits or digits), each of which is 0, −1, or +1. The value of the number represented by \( t_{k-1}, \ldots, t_0 \) (where each \( t_i \) is a trit) is defined to be \( \sum_{i=0}^{k-1} t_i 2^i \).

Note that the same number might have multiple representations. This is why this system is a redundant ternary system. For example, 1 0 1 and 1 1 −1 both represent the number 5.

Incrementing and decrementing work as you would expect. When incrementing, we add 1 to the low order trit. If the result is 2, then we change it to 0 and propagate a carry to the next trit. This is repeated until no carry results. Similarly, when we decrement we subtract 1 from the low order trit. If the result is −2, we set it to 0 and propagate a borrow (i.e. subtract 1 from the next lowest order trit). Again, we repeat this until no borrow is necessary.

The cost of an increment or decrement is the number of trits that change in the process. Suppose that we perform a sequence of \( n \) increments and decrements, starting from 0. Prove that the amortized cost of each operation is \( O(1) \), i.e. the total cost is \( O(n) \). Hint: think about a “potential function” or “bank account” argument.
**Solution:** Let $\Phi$ be a potential function equal to the number of nonzero trits. Note that $\Phi$ is initially 0 and is nonnegative, and hence the sum of amortized costs is an upper bound on the sum of true costs.

Note that each increment changes at most one trit from 0 to a nonzero (in this case 1), and similarly each decrement changes at most one trit from 0 to a nonzero (in this case $-1$).

Consider an operation (increment or decrement) which changes $k$ trits, and so has actual cost $k$. Then since at most one trit becomes nonzero, at least $k-1$ of these changes are from a nonzero to a 0 (since by construction a 1 never changes to a $-1$ and a $-1$ never changes to a 1). Hence $\Delta \Phi \leq 1 - (k-1)$, and thus the amortized cost is at most $k + \Delta \Phi \leq k + 1 - (k-1) = 2 = O(1)$.

### 3 k-th smallest elements (33 points)

Given an array $a_1, a_2, \ldots, a_n$ and an integer $k \in [n]$. For each $i \in [n+1-k]$, let $b_i$ be the $k$-th smallest element in $a_1, \ldots, a_{k+i-1}$.

(a) (17 points) Design an algorithm which outputs the array $b_1, \ldots, b_{n+1-k}$. The running time should be $O(n \log k)$.

**Solution:** We use a max-heap to maintain the smallest $k$ items in $a_1, \ldots, a_{k+i-1}$.

We first use $a_1, \ldots, a_k$ to build a max-heap $H$. Then output the top item as $b_1$. For $i = 2, \ldots, n+1-k$, if $a_{k+i-1}$ is smaller than the top item in the heap, we push $a_{k+i-1}$ into the heap, and then pop the top item. We output the current top as $b_i$.

The pseudocode is as follows:

```plaintext
H := build_max_heap([a_1, \ldots, a_k]);
b_1 := H.top();
for i := 2 \ldots n+1-k
  if (a_{k+i-1} < H.top())
    H.pop_top();
    H.push(a_{k+i-1});
    b_i := H.top();
output([b_1, \ldots, b_{n+1-k}])
```

(b) (16 points) Prove the correctness and the running time for your algorithm.

**Solution:** We use induction on $i$ to prove that: $H$ is holding the smallest $k$ items in $a_1, \ldots, a_{k+i-1}$ when $b_i$ is set. (When there is a tie on value, the one with smaller index is defined smaller.)

The base case is $i = 1$, the first $k$ items are all in the heap, which is exactly the smallest $k$ items in $a_1, \ldots, a_k$.

Assume case $i = i'$ hold, we want to prove that case $i = i' + 1$ holds.

If $a_{k+i-1}$ is greater or equal to the top of heap $H$. Then the smallest $k$ items in $a_1, \ldots, a_{k+i-2}$ is still the smallest $k$ items in $a_1, \ldots, a_{k+i-1}$, so there is no need to make changes on $H$. 

4
If $a_{k+i-1}$ is less than the top of heap $H$, then there are exactly $k$ items in $a_1, \ldots, a_{k+i-1}$ which are smaller than $H.top()$ (Because from induction hypothesis there are exactly $k - 1$ items in $a_1, \ldots, a_{k+i-2}$ which is smaller than $H.top()$, and $a_{k+i-1}$ is also smaller than $H.top()$). By removing $H.top()$ and pushing in $a_{k+i-1}$, $H$ now holds the smallest $k$ items in $a_1, \ldots, a_{k+i-1}$.

Therefore $H$ is holding the smallest $k$ items in $a_1, \ldots, a_{k+i-1}$ when $b_i$ is set. Since we set $b_i$ as the top item of $H$, it is exactly the $k$-th smallest element in $a_1, \ldots, a_{k+i-1}$. So the correctness is proven.

For the running time, the first step of building heap takes $O(k)$ time, and finding the top takes $O(1)$ time. In each iteration, there is at most one push and one pop_top. Since the heap has either $k$ or $k - 1$ items. It takes $O(\log k)$ time in each iteration. There are $n - k$ iterations, and the total running time is:

$$O(k) + O(1) + (n - k) \cdot (O(\log k) + O(1)) = O(n \log k)$$