Lab 12: Binary Search Trees and Heaps

Due: Thursday, March 16, 2023, before class (2 Python modules and 1 Google doc)

1. Twenty Questions

(7 pts) The game Twenty Questions is a two-player guessing game where one player thinks of something and the other player tries to figure out what it is by asking at most 20 yes/no questions.

1. Can this game be implemented using a binary search tree? If so, explain what the nodes represent, the relationship between a parent node and its children, and how it satisfies the binary search tree property. If not, explain why not and propose modifications to binary search trees to make it work.
   Yes! We would need to implement some kind of comparability trait for each of the questions, or simply map them to numbers, but otherwise it would work! This is because BSTs don't actually care about the exact difference between one node and another; they just need to compare the two. Each node would represent a question, with all numbers smaller than it being on the "no" side of the question, and all numbers larger than it being on the "yes" side of the question. You could construct this binary tree from the questions by assigning every question/guess the value 0, then making your way down the tree from the root. For each node, you would add 2^{-h} to all values in the right subtree and subtract 2^{-h} from all values in the left subtree. This would ensure that the BST property holds, giving a nice way to abstract away the funny tree of questions into a simple numeric BST.

2. AVL Tree

An AVL tree is a binary search tree that is height-balanced: for each node x, the heights of the left and right subtrees of x differ by at most 1 (the height of a subtree is the maximum distance from the root to any leaf). To implement an AVL tree, we need to consider how new nodes are inserted into the tree to maintain the height-balanced property.

Part A

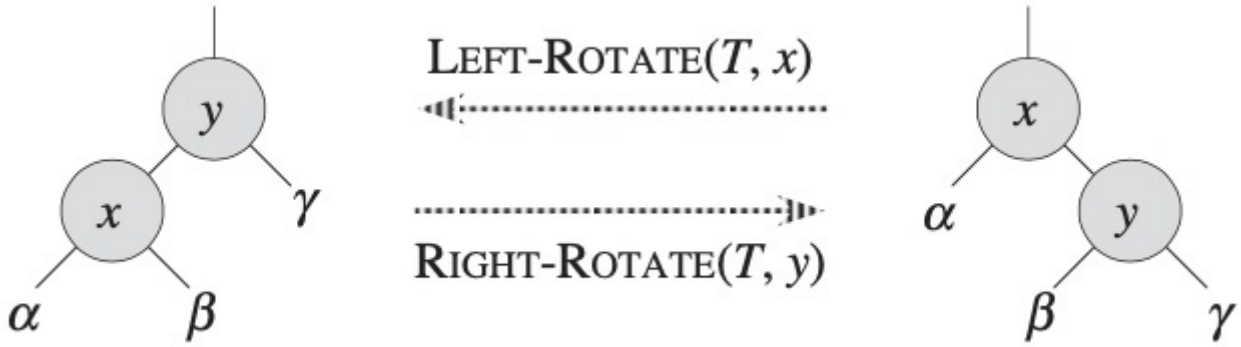
To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height-balanced. Specifically, the heights of the left and right children of a node might differ by 2. Describe a procedure called balance(x) that takes a subtree rooted at x whose left and right children have heights that differ by at most 2 and uses rotations to alter the subtree rooted at x to be height-balanced.

Here's my pseudocode. `x.left` and `x.right` are the left and right subtrees of node `x`, and `x.h` is the height of the tree rooted at x.

```
fn BALANCE_AVL(x) {
        if x.left.h == x.right.h { return; } // no imbalance detected

        if x.left.h > x.right.h {
                // case 1: heavier side is left
                if x.left.left.h < x.left.right.h {
                        LEFT_ROTATE(x.left);
                }
                RIGHT_ROTATE(x);
        } else {
                // case 2: heavier side is right
                if x.right.right.h < x.right.left.h {
                        RIGHT_ROTATE(x.right);
                }
                LEFT_ROTATE(x);
        }
}
```

The motivation for this comes from the way rotation works. As shown in the diagram, rotations only affect the positions of the nodes x and y and their subtrees α, β, and γ.



When a rotation is performed, the tree is redistributed. On a left rotation, γ is lifted by one and α lowered by one, while on a right rotation, α is lifted by one while γ is lowered by one. We can use this to our advantage: if we can ensure that the longest subtree is either α or γ, we know that a rotation about the root node can balance the entire tree. We know that if either α or γ are the longest subtree, then the other must be either 1 or 2 nodes shorter, because it is given that there is a difference in length of 1 or 2 (0 can be elimitated with an initial check) between the subtrees of the main root node.
Making sure either α or γ is the longest subtree is a simple process. Take the left graph in the diagram, for example. If γ is the longest sub-tree, we're done. If not, we figure out if α or β is longer. If α is longer, we're also done. If β is longer, we have to rotate left about x. For graph on the right side of the diagram, the process is identical, except everything is mirrored, including the direction of the rotation.

Part B

Using your balance(x) procedure, describe how to add a new node z to an AVL tree while maintaining the properties of an AVL tree.
Keep track of the max height of the tree rooted at each node with an additional height counter field in each node.

1. insert z into AVL tree T using the normal `TREE_INSERT` method for BSTs. This inserts it as a leaf.
2. Update all parent nodes' height counters. Do this by going up the tree, making sure that each node's height counter is one greater than its child with the largest height counter.
3. Move up the tree (ie, z, $z.\text{parent}$, $z.\text{parent.parent}$, etc.) until you find a node that is unbalanced (its child nodes' height counters differ by 2).
4. Balance that node using the `BALANCE_AVL` as described above.

Part C

Recall that the point of maintaining a balanced tree is to guarantee that methods like inserting a new node run in O(lg n) time. Explain how the procedure you described in part (b) takes O(lg n) time and performs O(1) rotations.

Insertion takes $\lg(n)$ time, as we've shown in class, because the tree is balanced.
Updating the height counters takes $lg(n)$ time, because you only have to update the inserted node's parents, which is a single path up the tree of length $\lg(n)$.
Balancing around the unbalanced ancestor takes constant time because it does a constant number of rotations (it can only call rotate a maximum of twice) and finding the heights of the subtrees is constant time (because we have kept track of them). Rotate is constant time because it is just moving a few pointers around the root nodes - it doesn't need to touch anything far down from the node it is called on.
Insertion is thus $\lg(n)$ time because initial insertion is $\lg(n)$, updating height counters is also $\lg(n)$, and everything else is constant time.

4. Heapsort Correctness Proof

3. (3 pts) Prove the correctness of HEAPSORT (as shown in CLRS on page 170) using the loop invariant:

> At the start of each iteration of the for loop of lines 2–5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$ in sorted order.

Code (copied from CLRS):

```
// indexing: assume [1,2,3,4,5,6,7,8,9][1..4] = [1,2,3,4]
HEAPSORT(A, n) // n = len(A)
1 BUILD-MAX-HEAP(A, n)
```

```
2 for i = n downto 2
3     exchange A[1] with A[i]
4     A.heap-size = A.heap-size − 1
5     MAX-HEAPIFY(A, 1)
```

> ***NOTE:***
> $A.\text{heap-size}$ starts at n and is reduced by one for each iteration of the loop. This is the exact same behavior as i from the perspective of `MAX-HEAPIFY`, which is the only place $A.\text{heap-size}$ is used. For this reason, it will be assumed that $A.\text{heap-size} = i$.

Initialization

At init, $i = n$, so $A[1..i] = A[1..n] = A$. We know A is a max heap because all that has been done in the algorithm so far is call `BUILD-MAX-HEAP(A, n)`, which makes A into a max heap. We know it contains the i smallest elements of A because $i = n$ and the n smallest elements of A are all elements of A. The subarray $A[i+1..n]$ contains the $n - i = n - n = 0$ largest values of A in sorted order because $A[i+1..n] = A[n+1..n] = [\,]$, which indeed has zero elements. The fact that $n + 1 > n$ seems to indicate something wonky with the indexes, but it's just an artifact of the inclusive indexing style.

Maintainence

The only thing that happens in the loop is that $A[1]$ and $A[i]$ are swapped, the heap is shrunk so it does not include $A[i]$, and then it is (magically) fixed using `MAX-HEAPIFY` so that $A[1..A.\text{heap-size}]$ is once again a max-heap. $A[1]$ is always the maximum value in the max-heap, and by swapping it with $A[i]$ and shortening the max-heap, it is removed from the max-heap. Thus the first part of the LI - that the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$ - is true: that subarray is a max-heap, and it contains the i smallest elements of A because $A[1..i+1]$ contains the $i+1$ smallest elements of A, and we removed the largest value from it.

We know that the subarray $A[i+1..n]$ contains the $n - i$ largest elements of $A[1..n]$ in sorted order because before this loop, it contained the $n - (i+1)$ largest elements of A, in sorted order, and we added in the next largest element of A to the front of it.

Termination

After the algorithm terminates, we can assume it goes back to the top and decrements i one more time before seeing that $i < 2$, at which point it terminates.
Now, $i = 1$, so $A[2..n]$ is in sorted order. The rest of the array, $A[1..1]$, contains the single smallest value of the entire array, so the entire array is in sorted order.

5. Min-Priority Queue K Largest Values

Min-priority queues have many useful applications as we've already seen. The following is another example. Suppose you have N values that you read in one at a time, and you want to maintain a collection of the k largest values you've seen so far, where k is much smaller than N. It would be inefficient to repeatedly sort the array of values every time you receive a new one just to pick out the top k, so you need to find another strategy.

Part A

Describe a better way to accomplish this task using a min-priority queue. Specifically, address the counterintuitive notion of using a min-priority queue when you're trying to maintain a collection of maximum values. What is the runtime of your algorithm in big-O notation? Don't forget to justify it!

Pseudocode:

```
1 fn K_LARGESET_VALUES(A: MinPriorityQueue, val: Comparable) {
2         if A.heap_size < A.length {
3                 A.insert(val);
4         } else if val > A[0] {
5                 A[0] = val;
6                 MIN_HEAPIFY(A, 0);
7         }
8 }
```

The min-priority queue holds the k largest values. This way, it is easy to add new elements, and it is very easy to remove the smallest value, which is always the one we will have to replace with any new values.
There are two cases here. First, `A` is not full, we just insert the new value, which takes at most $\lg(k)$ time. Second, if it is full but `val` is greater than the minimum of the MPQ (meaning that it should be a new largest value), then we replace the minimum of the MPQ with `val` (taking constant time) and call `MIN_HEAPIFY` (taking $\lg(k)$ time) to make it back into a valid min priority queue.
Because the function takes at most $\lg(k)$ time and must run once for each new value, the overall runtime of the algorithm is $O(N \lg(k))$. This can be further simplified because N is much greater than k (ie, $\frac{k}{N} \approx 0$), $lg(k)$ is insignificant compared to N, and can be assumed to be $O(1)$. This gives a final runtime of $O(N)$.

Part C

Describe an application where the program you wrote in part (b) would be useful. Please be clear about how each piece of your application fits into the problem statement (e.g. what do the N values represent, what does k represent, why would you want to find the largest values, etc.).

One application would be a leaderboard for a video game. Because it's a video game, tons of people can play, but that also means tracking every single score that has ever existed would be

very slow and memory intensive. However, people would still expect a relatively lengthy (by human standards) leaderboard - enough to slow down inefficient algorithms. k represents the length of leaderboard you want to store and N represents the total number of players. The amount of time it takes to update the leaderboard after each game would be constant, regardless of how many people are playing/have played. Even with larger values of k, the worst case would only be $\lg(k)$, so this would not slow down the game.