Warm-up:

A. List the red black tree properties.

Solution:

• Every node is black or red
• The root is black
• If a node is red, both it’s children must be black
• Null nodes are black
• Every path from a node to a NULL contains the same number of black nodes. The black-height of a node \( x \) is the number of nodes along the path from \( x \) to a null leaf, not including \( x \), but including the null leaf (for the edge case where \( x \) is a null leaf, the black height is 0).

Q1: Staircase

Let \( P \) be a set of \( n \) points in the plane. The staircase of \( P \) is the set of all points in the plane that have no points in \( P \) both above and to the right. Describe an algorithm to compute the staircase of a set of \( n \) points in \( O(n \log n) \) time.

Solution: Sort the points by decreasing \( x \) coordinates, put these in a list \( X \). To compute the staircase, move through \( X \) in order. The first point is always part of the staircase. Keeping track of the previous point added to the staircase—call it \( p = (x, y) \)—then for any other point \( p' = (x', y') \), \( p' \) is part of the staircase if \( p'.y > p.y \). Sorting takes \( O(n \log n) \) and iterating through \( X \) takes \( O(n) \) so the algorithm is \( O(n \log n) \).

Q2: Smallest Subtree

Given a binary tree and an integer \( d \), find the smallest subtree with a root at depth \( d \).

Solution: We can perform a recursive algorithm: if \( d = 0 \) then we just compute the size of the entire tree. Otherwise, we recursively find the root of the smallest subtree at depth \( d - 1 \) in the left subtree, call it \( r_L \), and we recursively find the root of the smallest subtree at depth \( d - 1 \) in the right subtree, call it \( r_R \). Then we simply return whichever has the smallest size.

Now, if we recompute the size of each subtree from scratch at every step, this algorithm becomes very inefficient. So as a preprocessing step, we can find the size of every subtree recursively: \( \text{size}(\text{root}) = \text{size}(\text{root.left}) + \text{size}(\text{root.right}) + 1 \) where \( \text{size}(\text{NULL}) = 0 \). We can then store the intermediate results (the size of all the subtrees) so when we run the above algorithm, we can find the size in \( O(1) \) time.

The pseudocode for the \text{size} method would be:

```python
ComputeSize(x):
    if x == NIL:
```
return 0
else:
    x.size = ComputeSize(x.left) + ComputeSize(x.right) + 1
    return x.size

And the pseudocode for the overall function (assuming the preprocessing has already occurred):

SmallestTree(x, d):
    if d == 0:
        return x
    else:
        left = SmallestTree(x.left, d-1)
        leftSize = left.size
        right = SmallestTree(x.right, d-1)
        rightSize = right.size
        if leftSize < rightSize:
            return left
        else:
            return right

With this preprocessing step, the above algorithm runs in $O(n)$ time.

Q3: Augmenting Trees  We want to maintain a data structure $D$ representing an infinite array of integers under the following operations:

- **INIT($D$)**: Create a data structure for an infinite array with all entries being zero.
- **LOOKUP($D$, $x$)**: Return the value of integer with index $x$.
- **UPDATE($D$, $x$, $k$)**: Change the value of integer with index $x$ to $k$.
- **MAX($D$)**: Return the maximal index for which the corresponding integer is non-zero.
- **SUM($D$)**: Return the sum of all integers in the array.

**Solution:** Let $n$ be the number of indices with non-zero elements. The data structure will be a RB Tree with the keys defined as the indices of the non-zero elements in $D$ and the values defined as the elements at that index. **INIT($D$)** just involves creating an empty RB Tree, which is $O(1)$.

**LOOKUP($D$, $x$)** is just a matter of searching the RB tree for the index $x$: if it’s not found, return 0. Otherwise return the value associated with that key. This takes $O(\lg n)$ time.

**UPDATE($D$, $x$, $k$)** just involves searching for the index $x$: if it’s present and $k \neq 0$, change the value in the node to $k$. If $k = 0$, then delete the node with key $x$ from the RB tree. If $x$ is not present, insert into the RB Tree the key-value pair $(x, k)$. This takes $O(\lg n)$ time.
Max(D): We can call the normal Max function for BSTs discussed in recitation (return the right-most node) which will run in $O(\lg n)$ time.

Would the following work: “Just keep track of the largest index added so far, and whenever Max is called, return that. This runs in $O(1)$ time.” (No: what if you update the index currently stored at Max to be 0?)

Sum(D): Just maintain a Sum variable that you update whenever a call to Update is made. If call Update(D,x,k), let $k'$ be the original value stored at index x, then let $\text{Sum} = \text{Sum} - k' + k$

Q4: Classic Heap Problem  
Given an infinite stream of numbers, show how to maintain the $k$ smallest elements at any given time.

Solution: Use a Max-Heap $H$. For each new element $x$ seen in the stream, if $H\.\text{size} < k$, insert $x$ into $H$. If $x < H\.\text{Max}()$, call $H\.\text{ExtractMax}()$ and insert $x$ into $H$. Then $H$ keeps track of the $k$ smallest integers at any given time.

Q5: Heap Practice
A. True or False: Every BST is a Heap
   Solution: False, the BST property is different from the Heap-order property.

B. True or False: A sorted array is a Heap
   Solution: True: it is a min-heap. Each element is smaller than all the elements that come after it in the array (i.e. smaller than all it’s descendants in the heap).

C. Recall that you can sort elements by inserting them into a BST and then doing an inorder traversal. True or False: there exists an $O(n)$ in-order traversal function for Heaps that allows you to print out the elements in a heap in order in $O(n)$ time.
   Solution: False: If this were true, then you could sort in $O(n)$ time: given an unsorted array, call BuildHeap on the array and then do the in-order traversal. Since comparison based sorting algorithms are all $\Omega(n \lg n)$, this would be impossible.

D. Consider the tree $T$ and array $A$ that define a heap $H$. True or False: a level order traversal of $T$ outputs the elements in $H$ in the same order they appear in $A$.
   Solution: True: draw a picture to convince yourself.

Q6: Heap Operations  
Consider the following heap (drawn as a tree):
A. Is this a Min heap or a Max heap?
   **Solution:** It is a max heap.

B. Draw the array representation of this heap.
   **Solution:** The array would just be $A = [-, 10, 8, 3, 1, 6]$

C. Show the resulting tree after calling **ExtractMax**
   **Solution** We just replace 10 with 6 and percolate down. The result is:

Q7: **Splay Tree Basic Operations** Insert the following into an initially empty splay tree: 3, 6, 2, 7, 4. Show the resulting tree.

**Solution:** After inserting 2, there is a zig-zig splay operation.

After inserting 7 there is another zig-zig operation, followed by a single rotation.
After inserting 4, there are two zig-zags.

The sequence of trees is:

Q8: RB Tree Operations  Insert the following into a RB tree: 3,1,6,2,9. Show the resulting tree. Then, delete 1 and show the resulting tree.

Solution: The first time something interesting happens is when 2 is inserted: in this case, 1 and 6 are colored black and 3 is colored red, then we recurse up the tree and color 3 black since it is the root.

The sequence of trees that results is (double circle = black, single circle = red):
If we were to delete 1 from the tree, we’d replace it with 2, then since 2 was red, we can just color it black to absorb the extra unit of blackness. The resulting tree is:
Some Useful Facts

These will be stapled to the back of your exam for reference (if needed).
Case 1:

Case 2:

Case 3:

Case 4:

new $x = \text{root}[T]$
1. \( \lg n = \log_2 n \)
\( \ln n = \log_e n \)

2. Below are some formulas that may come handy.
   
   - \( \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \)
   
   - \( \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \)
   
   - \( (a + b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i} \)

3. Log rules:
   
   - \( a^{\log_a b} = b \)
   
   - \( \log ab = \log a + \log b \)
   
   - \( \log \frac{a}{b} = \log a - \log b \)
   
   - \( \log a^b = b \log a \)
   
   - \( \log_a a^b = b \)
   
   - \( \log_a a = 1 \)
   
   - \( \log 1 = 0 \)

4. Expected value:
   
   - \( \mathbb{E}[X] = \sum_{x} x P(X = x) \)
   
   - Linearity of Expectation: for any finite collections of random variables \( X_1, X_2, ..., X_n \), \( \mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] \)