1. Min-heap k-th element smaller than x?

* Algorithm Description

Perform an inorder traversal of the min-heap. Count how many keys are found to be less than x. If the number of keys found is equal to k then stop traversing and return true. If after the traversal you don't find k keys less than x then return false.

* Pseudocode

public boolean is\_smaller(List<T> A, T k, T x) {

if (A.size == 0) return null;

if (A[1] > x) return false;

return inorder\_traversal(A, k, x, 0, 0);

}

private boolean inorder\_traversal(List<T> A, T k, T x int i, int count) {

if (count == k) return true;

if (i == (2 \*\* k) - 1) return false;

if (i < (2 \*\* k) - 1) {

return inorder\_traversal(A, k, x, 2i, count);

if (A[i] < x) count++;

return inorder\_traversal(A, k, x, 2i+1, count);

}

}

* Correctness

A min-heap k-th smallest element can be at most on the kth level of the heap. Therefore a bounds on where the k-th smallest is can be found. Then it's just a matter of searching through all the possibilities.

* Time Analysis

My algorithm will search k values to determine if there is a k-th smallest element in A that is smaller than x. The time complexity for this is O(k).

1. BST get successor of x

* Algorithm Description

First we get the max key of the tree and check x with it. Then we need to figure out if there is a node equal to x. Once that has been done we can determine what the successor of x is.

* Pseudocode

public T successor(T x) {

T max = max(root);

if (max == null) return null;

if (x > max) return null;

if (x == max) return x;

return successor(null, root, x);

}

private T max(Node<T> node) {

if (node == null) return null;

while (node.right != null) {

node = node.right;

}

return node.key;

}

private T successor(Node<T> parent, Node<T> node, T x) {

if (node == null && parent == null) return x;

if (node == null && parent != null) return parent.key;

if (x > node.key) return successor(node, node.right, x);

if (x < node.key) return successor(node, node.left, x);

return x;

}

* Correctness

I have tested my code with real examples

* Time Analysis

First I check if x is larger than the largest key, that takes O(log(n)). Then I traverse down one of the branches in the tree to find what the successor of x is and that also takes O(log(n)). Thus the overall complexity would be O(log(n)) + O(log(n)) = O(log(n))

1. BST get rank of x

* Algorithm Description

How to augment?

I would augment the BST by adding a size attribute to the node class. Thus each node would keep track of how many nodes there are in its subtree.

How to keep size updated?

Like we talked in class, you can update the size of a node by using the following methodology:

Insert: every node you compare to find the spot for the new node, increment the size

Delete: every node you compare to find the node to delete, decrement the size

This type of augmentation would not affect the time complexity of insert, delete, or search

Layout of the algorithm?

The algorithm will use recursion to find the rank of an item x. Each recursive call will find out more information about what the rank of x should be.

* Pseudocode

public int rank(T x) {

return get\_rank(root, x, 0);

}

private int rank(Node<T> node, T x, int c) {

if (node == null) return c + 1;

m = 0;

if (node.left != null) m = node.left.size;

if (x < node.key) return rank(node.left, x, c);

if (x > node.key) return rank(node.right, x, c + m + 1);

return c + m + 1;

}

* Correctness

I have tested my code with real examples

* Time Analysis

The algorithm has a time complexity of O(log(n)) this is because I only follow one branch down the tree.

1. Range Query

* Algorithm Description

My algorithm uses recursion and the heap to build a list of keys in the range. First the algorithm finds the first node that is within the range. After the first node is found several questions need to be asked to figure out when a node should be added to the list to keep the list in sorted order.

* Pseudocode

public ArrayList<T> range\_query(T[] range) {

List<T> list = new ArrayList<T>();

range\_query(root, list, range[0], range[1]);

return list;

}

private ArrayList<T> range\_query(Node<T> node, ArrayList<T> list, T l, T r) {

if (node == null) return;

if (node.key < l) {

range\_query(node.right, list, l, r);

}

else if (node.key > r) {

range\_query(node.left, list, l, r);

}

else {

if (node.key == l && node.key == r) {

list.add(node.key);

}

else if (node.key == l) {

list.add(node.key);

range\_query(node.right, list, l, r);

}

else if (node.key == r) {

range\_query(node.left, list, l, r);

list.add(node.key);

}

else {

range\_query(node.left, list, l, r);

list.add(node.key);

range\_query(node.right, list, l, r);

}

}

}

* Correctness

The algorithm is correct because for one I have gone over many examples on the white board but also because the algorithm makes sense. When spoken in plain english the logic seems to flow.

* Time Analysis

My algorithm finds each node that is within the range and adds it to the list in sorted order.

The time complexity for this is O(k+log(n)) because the algorithm has to find each node k times.

1. Range Sum

* Algorithm Description

Augmentation:

Similar to the augmentation done in class. I will have each subtree root keep track of the sum of the keys in that subtree. Keeping this value updated would be easy and would not affect insert, delete, or search. This is because you can update the sum of the subtrees as you traverse the tree.

My algorithm is a modification of the algorithm talked about in class when we talked about finding the minimum value in a tree given a range in O(log(n)).

My algorithm starts with finding a node that is within the range. After the node is found the left side of its subtree is traversed with a loop. The loop ends when it either finds a null value or it finds a node that is equal to the left boundary. During the loop it will add the sum of keys the nodes stored in the right child.

A similar idea is used for traversing down the right side of the node.

After both sides have been traversed the sum of keys should be known with a time complexity of O(log(n)).

* Pseudocode

public int range\_sum(int[] range) {

return range\_sum(root, range[0], range[1]);

}

private int range\_sum(Node<int> node, int l, int r) {

if (node == null) return 0;

int sum = 0;

if (node.key < l) {

sum += range\_sum(node.right, l, r);

}

else if (node.key > r) {

sum += range\_sum(node.left, l, r);

}

else {

sum += node.key;

Node<int> temp = node.left.copy();

while (temp != null) {

if (temp.key == l) {

sum += temp.key;

if (temp.right != null) sum += temp.right.key\_sum;

break;

}

else if (temp.key > l) {

if (temp.right != null) sum += temp.right.key\_sum;

temp = temp.left;

}

else if (temp.key < l) {

temp = temp.right;

}

}

temp = node.right.copy();

while (temp != null) {

if (temp.key == r) {

sum += temp.key;

if (temp.left != null) sum += temp.left.key\_sum;

break;

}

else if (temp.key < r) {

if (temp.left != null) sum += temp.left.key\_sum;

temp = temp.right;

}

else if (temp.key > r) {

temp = temp.left;

}

}

}

return sum;

}

* Correctness

As stated before my algorithm is a modification of the algorithm talked about in class.

* Time Analysis

My algorithm first traverses the tree until a node is found that is within range this has a time complexity of O(log(n)). Then it traverses the left side of the node which also has a time complexity of O(log(n)). Then it traverses the right side of the node which as well takes O(log(n)).

Thus the overall time complexity of my algorithm would be

O(log(n)) + O(log(n)) + O(log(n)) = O(log(n)).