

Homework 3 Solution Guide

1)

Idea

The important thing to note is that we need to find the k th smallest number in $O(k \log k)$ not $O(k \log n)$ time. Because of this, we can never perform a remove-min operation on the original binary search tree, because this would be an $O(\log n)$ time operation.

However, the heap property can be exploited to access the k smallest elements in the heap. Notice that in a min heap, the smallest element is at the top of the heap, and the next smallest element must be one of the children of this node. This is because the heap property requires that any child of a node be greater than its parent.

To solve this problem, we will create another min heap. The entries in this heap will be stored based on the same key, however an additional value will be stored that indicates the index in the original array where the key was stored. Note that the children of any node in an array based heap can be found in $O(1)$ time from section 4.3.2 of Manber on heaps.

The following procedure can be used. Take the first item in the original heap and add it to the new heap. This initializes the new heap. Next, remove the top item from the new heap. This is the minimum element not yet considered. Take the index into the original array, and use this to calculate the children of this node in constant time. Add those children to the new heap, there will only be two of them. Removing the next item from the new heap finds the next smallest node. Repeat this process until the k th item has been reached.

The second heap used is the same as a normal heap, except a second item is stored with each key. Specifically each item contains two integers $\{\text{key}, \text{index}\}$. The second heap maintains the heap property on the key, and the index is the position in the original array for the original heap where that key came from.

It is assumed that $K \leq N$.

PseudoCode

```
find_kth()
{
    int K = desired value;
    int original_heap[N];

    Heap second_heap; //heap per pg 76 of Manber,

    int count = 1;
    Item last_value = original_heap[0];
    //lowest is stored at original_heap[0]
    add_children(0);

    while(count < K)
    {
```

```

        //remove 1 item from second_heap
        {next_value, next_index} = second_heap.remove_min();
        //add 2 items to second_heap
        add_children(next_index);
        count++;
    }

return last_value;
}

void add_children(int location)
{
    //note that the left child is at 2i and the right at 2i+1
    //if the index of the array is 1 based
    int left = 2i+1//formula for left child
    int right = 2i+2//formula for right child
    if(left>=0 && left<N)
        second_heap.add({original_heap[left],left});
    if(right>=0 && left<N)
        second_heap.add({original_heap[right],right});
}

```

Complexity

Notice that every time this procedure is used, the new heaps size increases by one. Running this procedure k times finds the k th smallest node, with the new heap size being at most $k+1$ (it starts at size 1 and is increased by one with each procedure) Each run of this procedure requires two $O(\log k)$ time insertions to the new heap. It also requires one $O(\log k)$ remove-min operations, this happens k times so the running time is at worst $O(k \log k)$ time.

2)

This data structure can be achieved by using a modification of the binary search tree data structure. Note that in order for an insert, delete or search in a binary search tree to be $O(\log n)$ time the binary search tree used must be a balanced binary search tree, such as an AVL or red-black tree.

Modify the binary search tree such that in addition to the key, each node holds a value indicating the total number of nodes in the left and right subtrees of that node, including itself.

Modifying the insert method is simple. Assume initially the modified binary search tree is empty. A new item i is added, for every new item, initialize the count to one. Simply add this to the tree. If the tree is not empty, the tree is traversed to find the location of the new item. At every node that is visited in the traversal, increase the count at that node by one.

Modifying the delete method is similar, decreasing the count as the tree is traversed. Some care must be taken that if the node removed is not a leaf that the counts of the nodes involved are updated properly when a new node is swapped into the position of the deleted node. Basically, keep decrementing counts until the leaf node that is going to be used is found, then swap that in with the old node's counter (decremented as it was touched initially).

The `find_smallest(k)` requires a little bit more thought. Basically, the idea is to do a traversal of the tree and use the structure of the tree to determine the position of the current node. If the left child of the tree has more elements in it than k , then this node certainly is not the k th smallest, and also the k th smallest element must be in the left subtree of the node. If the left subtree has a sum less than k , then the k th smallest must either be this node or in the right subtree of this node. Update k by subtracting the total number of nodes in the left subtree, and subtracting one more for this node. When k has reached 0, the resulting node is found. It is important to check for the node before moving on to the right subtree – otherwise the $k+1$ th element is returned.

Starting at the first node, repeatedly perform this procedure. If the current node's count is equal to k , a solution is found. Return this node. Look at the left subtree of this node. If the total count of the left subtree is greater than or equal to k , then the solution is in the left subtree. Repeat this procedure starting at the root of the left subtree. If the total count in the left subtree is less than k , subtract the count of the left subtree plus one (for the current node) from k and repeat the procedure on the right node.

PseudoCode

This pseudocode will be based on a balanced binary search tree. However, Manber does not include pseudocode for a balanced binary search tree, but only for standard binary search trees. For notes on the balancing operations, see Chapter 13 of Corman. The insert and delete methods from Manber will be a suitable basis for the other functions, assuming some balancing occurs. Note that for any balanced binary search tree, there may be rotations required to maintain the balanced property. Therefore, those movements must also be accounted for in this code. This is simply enough done any time there is a rotation, update the counts of each node. Each subtree that is rotated simply carries the counts for that subtree along. Then, at the root of the rotation, note that the

total number of nodes below that node is the same, just in a different order. The details of this will vary depending on the type of balanced binary search tree used.

Structure

Each node will now have an additional counter for the total number of children below it. Node = {value, count}. Count is initialized to one whenever a new node is created.

Insert(x)

This proceeds exactly according to the standard tree, except for the counter.

Every time any node is visited, increment count at that node.

Delete(x)

Every time any node is visited, decrement count at that node. If the node deleted is not a leaf, then switch the values of the node per the normal delete procedure, and decrement any node on the path from the root to the leaf that was switched in.

Find_Smallest(k)

```
{
    return find_by_node(root, k);
}
```

find_by_node(Node current, int k)

```
{
    if(current->left!=NULL && current->left->count>=k)
        return find_by_node(current->left, k);
    //not here, not in the left subtree, it must be in the right subtree
    int new_k = k-1;
    if(current->left!=NULL)
    {
        new_k-=current->left->count;
    }
    if(current->right==NULL)
        exit(0);//no such element
    if(new_k==0)
        return current;
    find_by_node(current->right, new_k);
}
```

Complexity

For insert and delete, an $O(1)$ increment or decrement is added to the standard procedure. The find_smallest operation only requires a simple traversal of a tree. Since these are all just traversals of a balanced binary search tree, the total time is $O(\log n)$.

3)

This problem requires another modification of a balanced binary search tree.

Create a binary search tree that has a normal key for the binary search tree property, but includes a pointer to another standard binary search tree at each node. This modified binary search tree will be the “block tree”. The individual trees pointed to by each node are the “item trees”.

The modified binary search tree will correspond to the block numbers, and each individual tree will be the stored values within that block.

When inserting new items, first call $b_i = \text{which_block}(s_i)$ on the new item. Run the normal binary search insertion procedure on b_i . If no node has a key of b_i insert a new node, which will point to a new binary search tree into which s_i will be inserted. If some node has the key b_i then insert the item s_i into the item tree at that node. In the worst case the block tree has one node for each item, and the original insertion will take $O(\log n)$ time, and similarly in the worst case there is only one node and the search tree pointed to by that node includes all n items the second insertion will take at worst $O(\log n)$ time, so at worst this procedure takes $2 * O(\log n) = O(\log n)$ time.

Deleting item s_i takes the same approach, first search for b_i ; if no node b_i is found, there is no item s_i and this procedure is done. If there is a node b_i then just use the normal binary search tree deletion method on the item tree pointed to by node b_i . Again by the same argument this will take at worst $O(\log n)$ time. Note that it is important to delete an empty node in the block tree if all items have been removed from its item tree, in order to preserve the $O(\log n)$ running time – otherwise the number of empty block nodes is potentially unbounded (inserting and deleting new block codes over and over expands the tree indefinitely).

Deleting an entire block is just the same as a deletion in a normal binary search tree, except the memory used by the subordinate item tree will be freed as well. Clearly this is an $O(\log n)$ time procedure.

Pseudocode

Structure

BlockTree = a balanced binary search tree of

BlockNode = {block_ID, ItemTree}

ItemNode = {ItemValue}

BlockTree = a balanced binary search tree of block nodes

ItemTree = a balanced binary search tree of ItemNodes (standard BST)

Insert(s_i)

{

$b_i = \text{which_block}(s_i)$;

 BlockNode found = BlockTree.search(b_i);

 if(found==null){

 BlockNode new_node = new BlockNode(b_i , new ItemTree(s_i));

 found.insert(new_node); //insert on b_i w/ standard BST insertion

 }

```

        found->ItemTree.insert(si); //standard insertion into the subtree
    }

Delete(si)
{
    bi = which_block(si);
    BlockNode found = BlockTree.search(bi);
    if(found==null){
        return; //nothing found
    }
    found->ItemTree.delete(si); //standard BST deletion
    if(found->ItemTree->size==0)
        BlockTree.delete(bi); //standard deletion again
}

Delete_Block(j)
{
    BlockNode found = BlockTree.search(j);
    if(found==null){
        return; //nothing found
    }
    delete found->ItemTree; //free the memory from the itemtree
    //this can be done in a O(1) operation, just free the memory
    BlockTree.delete(j); //delete the node
}

```

Complexity

Note that there are two balanced binary search trees with n total items. There are two worst case scenarios. In one, each node in the block tree contains only one item, so the BlockTree has n nodes, and all operations on it are $O(\log n)$ time. In the other worst case, the BlockTree has only one node, whose item tree contains all n items, therefore any operation is just an $O(\log n)$ time operation on this node. Each operation requires at most two operations, one at each level, since each level has at most n nodes, this is at worst $O(\log n) + O(\log n) = O(\log n)$ time for each operation.

4)

This problem is somewhat similar to problem 4.16.

The important thing to note with this problem is that the partial sum requires the sum to be made by **value** where the addition operation is done by **key**. In order to reconcile these two methods, two binary search trees must be used. One will store the elements by value and one will store the elements by key. It is important to keep track of pointers between the nodes in the two different trees, so that each key node corresponds to a value node in the value tree. Also note that the elements in the value tree need not be unique – and so multiple copies at one node value can be used.

In order to add a node, simply create two nodes one with the key and one with the value, and each with a pointer to the other node. Add these nodes to both trees.

In order to delete or modify from these trees, the simplest method is to search for the desired key in the key tree and remove that node from the key tree. Then go to the node pointed to in the value tree by the node in the key tree and remove that node from the value tree. These operations both take $O(\log n)$ time. Then, make any modifications required and add the new node to the key and value tree.

In the value tree, additional information must be stored. At each node, store the sum of the values in the children of that node (including the value at the root node itself). In order to calculate the partial sum, simply perform a traversal of the tree searching for the value y . If a visited node has a value less than y , add the value of that node to the partial sum. If the traversal enters the right subtree, add the sum of the left subtree to the partial sum (as y is greater than the current node to enter the right subtree, add the sum of any values in the left subtree) and continue the traversal. If y is less than the current node, the right subtree is larger and need not be added.

The node format for the key tree is {key, pointer to value node, # of keys in this nodes subtree, sum of values in subtree including this node}.

The node format for the value tree is {value, pointer to value node, # of copies of value, partial sum}.

```
keyNode = {key, value_pointer,key_count,partial_sum}
valueNode = {value, key_pointer,copies_count, partial_sum}
```

```
BalancedBinaryTree keyTree;
BalancedBinaryTree valueTree;
```

```
keyTree insert()
{
    //Perform a standard balanced BST insert
    //add the value to each partial sum along the way, and
    //increment the counters on node visited in the traversal
}
valueTree insert()
{
```

```

        //Perform a standard balanced BST insert
        //add the value to each partial sum along the way
    }

insert(key, value)
{
    KeyNode = {key, NULL, 1, value};
    ValueNode = {value, KeyNode, 1, value};
    KeyNode->Pointer = ValueNode;

    keyTree.insert(KeyNode);//O(log n)
    valueTree.insert(ValueNode);//O(log n);
}

delete(key)
{
    KeyNode = keyTree.search(key);
    keyTree.delete(KeyNode);//O(log n)
    valueTree.delete(KeyNode->Pointer);//delete the whole node unless there
    //are copies
}

//find the sum of values of the first k nodes
find_smallest_by_key(int k)
{
    Node current = keyTree->root;
    int count = keyTree->root->subtree_count;
    while(count!=k)
    {
        if(count==k)//found
            return current->partial_sum;
        if(current->left!=null && current->left->count>=k)
        {
            current = current->left;
        }else if(current->right !=null){
            k-=current->left->count +1;//+1 for this node
            current = currnet->right;
        }else{
            break;//error
        }
    }
}

partial_sum(y)
{

```

```

Node current = valueTree->root;
int count = valueTree->root->subtree_count;
partial_sum = 0;
while(true)
{
    if(current->value<=y)
        partial_sum+=current->value;
    if(current->value==y || (current->left==NULL && current-
>right==NULL)//done
        break;
    if(y<current->value)
    {
        current = value->left;
    }else if(y>current->value)
    {
        partial_sum+=current->left->partial_sum;
        current = value->right;
    }
}
return partial_sum;
}

```

Complexity

Each operation is a traversal in a binary search tree, and so is $O(\log n)$ time so long as the search tree is balanced. Some care must be taken to update the balancing algorithms, carrying along the partial sums properly. Even though the insert and delete methods modify two binary search trees, no search in the value tree (which has multiple items) is ever required.