CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2014

Solutions to Problem 1 of Homework 6 (10 points)

Name: Sahil Goel *Due: Wednesday, October 15*

According to Josephus' account of the siege of Yodfat, he and his n comrade soldiers were trapped in a cave, the exit of which was blocked by Romans. They chose suicide over capture and decided that they would form a circle and start killing themselves using "step" of size m; i.e., starting the count with some fixed person (called "number 1"), every m-th person is killed, after which the suicides continue with the remaining people on the (now smaller) circle. For example for $n = 8$ and $m = 3$, the order in which the people are killed is $(3, 6, 1, 5, 2, 8, 4, 7)$. We want an $O(n \log n)$ algorithm to find the order in which the soldiers were killed. (**Hint**: Use *augmented* 2-3 trees.)

Solution: This problem can be solved using Augmented 2-3 trees by storing the v.num for every node which stores number of nodes in v's subtree.

Since we know, if we store v.num with every node, we can find out any kth rank element in time $O(\log n)$. We will use this property in our algorithm, to find any kth rank element.

We will also use a ptr root which will always point at root and it can check whether the tree is empty or not and also number of elements present in the tree at any given time.

Overall Idea: Initialize rank=k(First element to be deleted or first person to die). After the first element is deleted, the next element that should be deleted should have rank = previous element rank + k, but when the previous element is deleted, the new rank will be =previous element rank + k - 1(because of deletion of that node). Now we need to check if this rank is lower than the total number of nodes in tree or not. If it is lower then continue, else take rank = mod(rank, root.num) and delete the element with rank=rank. Continue until root.num is not equal to zero or tree is not empty.

Assume create2-3Augmentedtree is a procedure which creates an 2-3 Augmented tree for elements of Arr[ ] which contains all the numbers from 1 to n. This procedure also stores v.num value for every node. This procedure returns the root of the tree

Also assuming that whenever a node is deleted, all the v.num values are updated.

```
suicideorder(Arr[],k)
{
root = create2-3Augmentedtree(Arr[]);
"rank"=k;
while(root is not equal to NULL)
Print element having rank "rank"
Delete element having rank "rank"
"rank" = mod("rank" + $k - 1$, root.num)
if(rank==0) rank=root.num
end while
}
```

This will print the order in which the persons will die

Example
Suppose n=8 and k=3
Initially k=3, elements in tree are 1,2,3,4,5,6,7,8
print element with rank 3 : = {3}
now element with rank 3 will be deleted

Now, root.num=7 and tree = {1,2,4,5,6,7,8}
new rank=mod(rank + k-1,7)
rank = mod(3 + 3 - 1 , 7)
rank = 5
print element with rank 5 := {6}
Now element with rank 5 will be deleted

Now, root.num=6 and tree = {1,2,4,5,7,8}
new rank = mod(5 + 3 - 1,6) = 1
rank=1
print element with rank 1 := {1}
Now element with rank 1 will be deleted

Now, root.num=5 and tree = {2,4,5,7,8}
new rank = mod(1 + 3 - 1, 5) = 3
print element with rank 3 := {5}
Now element with rank 3 will be deleted

Now, root.num=4 and tree = {2,4,7,8}
new rank = mod(3 + 3 - 1, 4) = 1
print element with rank 1 := {2}
Now element with rank 1 will be deleted

Now, root.num=3 and tree = {4,7,8}
new rank = mod(1 + 3 - 1, 3) = 0
rank = root.num = 3
print element with rank 3 := {8}
Now element with rank 3 will be deleted

Now, root.num=2 and tree = {4,7}
new rank = mod(3 + 3 - 1, 2) = 1
print element with rank 1 := {4}
Now element with rank 1 will be deleted

Now, root.num=1 and tree = {7}
new rank = mod(1+3-1,1) = 0
rank = root.num = 1
print element with rank 1 := {7}
Now element with rank 7 will be deleted

As you can observe the order of deletion is ”3, 6 ,1, 5 , 2, 8, 4, 7” which is the correct order

Lets calculate the complexity of solution
1. create2-3Augmentedtree will take time $O(n \log n)$ as to insert one element it takes time $O(\log n)$ and hence to insert n elements it will be $O(n \log n)$
2. deleting a single node takes time $O(\log n)$, and since this delete is in while loop which will be called for n times, total time in the loop is $O(n \log n)$

Total time T = T(step 1) + T(step 2) = $O(n \log n) + O(n \log n)$
T = $O(n \log n)$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2014

Solutions to Problem 2 of Homework 6 (18 Points)

Name: Sahil Goel *Due: Wednesday, October 15*

Assume you are given a binary search tree T on n elements, whose height is h. As usual, let $v.key$ denote the key value of node v, and recall than the left sub-tree of every node v only contains elements less than *or equal to* $v.key$. (In particular, our tree might contain several elements having the same key.)

(a) (5 points) Using a slight modification of the PostOrder-Tree-Walk procedure, argue that in time $\Theta(n)$ you can compute, for every node v, the number of nodes (call it $v.less$) *in v's sub-tree* which are *strictly* less than $v.key$. Write a pseudocode for the resulting procedure Fill-Less(T), which will fill in the values $v.less$ for all the nodes of T in time $O(n)$.
(**Hint**: Remember than some elements in the left sub-tree of v might be equal to $v.key$, and should not be included in the count for $v.less$. Also do not forget to cover the base case.)

Solution:
Assuming T points to the root of the Tree initially. This problem can be solved by augmenting BST tree nodes with these values
(a). v.num : It stores number of nodes in its subtree
(b). v.max and count : It stored the maximum value in its subtree and its count
Now, these values can be calculated in O(n) time

To calculate v.num in time O(n)

```
PTWnum(T)
{
if(T==NULL)
return -1;       % -1 because we counted this node so we will have to subtract
PTWnum(T.left);
PTWnum(T.right);
T.num = T.left.num + T.right.num + 2;  %2 to add left and right child
return T.num; }
```

To calculate v.maxc in time O(n). There can be 3 cases for each node
(a). If node v has a right subtree, then the maximum value for subtree under this node will lie in its right subtree. The maximum value then can be its right child or T.right.max, which is maximum of right child's subtree. If the maximum is equal to the right child, then count=1 as it will have occurred first time else the count will be what it was before i.e. T.right.count

(b). If node v doesn't have a right subtree and Its left child is equal to the maximum of left child's subtree, then we need to assign T.max as its left child and T.count = T.left.count+1

(c). If node v doesn't have a right subtree and its left child is smaller or greater than the maximum then we will assign T.max=T.left.max and T.count=T.left.count

Initially max=-infinity and count=0

```
PTWmaxnc(T)
{
if(T==NULL)
return
PTWmaxnc(T.left);
PTWmaxnc(T.right);
if(T.right!=NULL)        % Case (a)
T.max=maximum(T.right.max,T.right);
if(T.right.max¿=T.right) then
T.count = T.right.count
else T.count=1
else if(max=T.left)        % Case (b)
T.count=T.left.count+1
T.max=T.left
else        % Case (c)
T.max=T.left.max;
T.count=T.left.count;
}
```

Now We have calculated all these values and now we will use these to calculate the less values for every node. Notice, for every node
there will be 2 cases now
if v.key = v.left.max v.less = v.left.less - v.left.count
else v.less = v.left.less
Also we have to add "1" if v.left.ket is not equal to v.key
v.less = v.less + 1

```
Fill-less(T)
{
% Base case :- When T is a leaf node or when tree is empty or when only right subtree exists%
if($T == NULL$)
return

if ($T.left == NULL$ and $T.right == NULL$)     Leaf node
T.less = 0
return

if ($T.left == NULL$)        Only right subtree exists
T.less = 0
Fill-less($T.right$)
return
else
Fill-less($T.left$)
Fill-less($T.right$)

if($T.key = T.left.max$ )
T.less = T.$left.less - T.left.count$      Left subtree has "T.count" equal elements to T.key
else
T.less = T.$left.less$


% We still have to add 1 for left child if it not equal to T.key
if(T.key is not equal to T.left.key) T.less=T.less+1;
return
}
```

□

(b) (5 points) Now that each node v contains the value $v.less$, show that you keep maintaining this value for each successive INSERT operation. Namely, write the pseudocode for the INSERT$(T, value)$ procedure, which will insert a new (leaf) node w into T with $w.key = value$. The running time of your procedure should be $O(h)$, and it should correctly maintain all the $v.less$ values.
(**Hint**: When going down the tree from a node v, when does the insertion of w increases the value $v.less$?)

Solution:
Notice to insert an element in a BST, we initially search for element and find the exact position where it should be inserted. If value is less than or equal to any node v.key, then we go to the left subtree and if its greater than node v.key, than we go to right subtree. In this case there will be a slight modification. Whenever value is less than node v.key, we will increment v.less by 1 and go to left, whereas if it is equal to node v, we will still go left but not increment v.less. There is no change if we go to right

```
Insert(T,value)
{
if(T==NULL)          Found correct place to insert
T=new v;
T.key=value;
T.less=0;

if(value < T.key)
T.less=T.less+1;
Insert(T.left, value);
if(value = T.key)
Insert(T.left,value);

if(value > T.key)
Insert(T.right, value)

}
```

□

(c) (5 points) Assume now that we successfully maintain the $v.less$ field for both the INSERT and the DELETE Operation, and also that all the key values are distinct. Show how to implement an $O(h)$-time procedure MYRANK(v), which will return the rank of a node v in the BST (i.e., if $v.key$ is the minimum of all the values it will return 1, or if it is the median it will return $n/2$). Why can't we simply return $v.less + 1$ in time $O(1)$?

Solution:
We know, that if we go to left child of node all the nodes are less than the node and if we go to right child of node all the nodes are greater than node. Now in this case we will have to consider the reverse of traversing. Suppose A is parent of B.
1. If B is left child of A, then $A.key > B.key$, therefore rank of B will not be affected by A's right subtree because all the elements in A's right subtree will be greater than B.key

2. If B is right child of A, then $A.key < B.key$ and we also know that all the elements in A's left subtree are also less than A.key, therefore all those elements will be less than B.key also. Hence rank = rank + 1 + number of elements less than A.key(A.less). We had to add one to consider A.key itself.

```
MyRank(v)
{
temp = new v;
% Initialize rank with 1 and temp with v.parent
rank = 1; temp=v.parent;
while(temp is not NULL)
{
if(v is left child of temp)      %No computation required just traverse up
{
v=temp;
temp=v.parent;
}

if(v is right child of temp)      %It is greater than all elements in left subtree of parent
{
rank = rank + temp.less + 1;      %+1 because of parent itself
v=temp;
temp=v.parent;
}
return rank;
}
```

Since we are traversing up all the time and at each level we are performing work O(1), therefore total time complexity is O(1*h)= O(h)

We can't simply return v.less + 1, because it only gives the number of elements which are less than the key in its subtree, it doesn't consider smaller elements in other subtrees which might be present. Example:- if we do this, all the leaf nodes will have rank 1 which is clearly wrong.

□

(d) (3 points) Instead of running the procedure MyRank as in part (c), is it possible to simply maintain — in time $O(h)$ — the field $v.rank$ which will correctly compute the rank of v after each insertion and deletion? If yes, show how, if not, explain why not.

Solution: No, it is not possible to maintain the field v.rank in time O(h) because, deletion of one element might change the rank of all the elements in the tree. For example, suppose if the minimum of tree is deleted, then all the other nodes rank should be decremented by 1 which will take time O(n). Also if suppose an element is added which is minimum of all, then every node rank should be incremented by 1 which is also O(n). To conclude, whenever we delete or insert a node, ranks of all the nodes might get affected(not just the in its subtree or its parents)., hence it is not possible to maintain rank in time O(h).

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 14, 2014

Solutions to Problem 3 of Homework 6 (14 (+2) points)

Name: Sahil Goel *Due: Wednesday, October 15*

Assume we insert sequences of English letters into an empty 2-3-tree, following the standard alphabetic ordering when making comparisons: $A < B < C < \ldots < Z$.

(a) (3 points) Assume you insert the letters $R, E, L, A, T, I, O, N, S$ into a fresh 2-3-tree T in that exact order. What is the resulting height of T?

Solution: ***************** INSERT PROBLEM 3a SOLUTION HERE *************** □

(b) (4 points) Let h be your answer to part (a). Is it possible to get less than h? If not, prove it. If yes, give the best ordering you can. What is the depth $h_{\min}$ you get?

Solution: ***************** INSERT PROBLEM 3b SOLUTION HERE *************** □

(c) (3 points) Can you achieve $h_{\min}$ with lexicographic order $A, E, I, L, N, O, R, S, T$?

Solution: ***************** INSERT PROBLEM 3c SOLUTION HERE *************** □

(d) (2 points) **Extra credit:** Can you achieve $h_{\min}$ for part (b) using an English word that is a permutation of $R, E, L, A, T, I, O, N, S$?

Solution: ***************** INSERT PROBLEM 3d SOLUTION HERE *************** □

(e) (4 points) Give an example of a 2-3-tree T with exactly 5 leaves and two distinct values a, b, such that one gets different final trees if one first inserts a into T and then deletes b, instead of first deleting b from T and then inserting a. Explain what happened.

Solution: ***************** INSERT PROBLEM 3e SOLUTION HERE *************** □

CSCI-GA.1170-001/002 Fundamental Algorithms	October 14, 2014

Solutions to Problem 4 of Homework 6 (14 points)

Name: Sahil Goel *Due: Wednesday, October 15*

Assume that you are given a 2-3 tree T containing n distinct elements.

(a) (4 points) Show how to find the successor of a given element $x \in T$ in time $O(\log n)$.

Solution:
Initially to search for element x and find a pointer to it

```
search(T,x)
{
while(T.key not equal to x)
if(T.key less than x)
return element doesn't exist
if(T.child is leaf node and T.child[1] <> x and T.child[2] <> x and T.child[3] <>x) return
-1;
for i in 1..3 loop          if(T.child[i] = x and T.child[i] is leaf node)
return T.child[i];
end for;
for i in 1..3 loop
if(T.child[i].key ≥ x)
break;
search(T.child[i],x)

}

Successor(T,x) {
pos = search (T,x);
if(pos has a right sibling)
return right sibling
else
{
pos = pos.parent;
while(pos is rightmost child of its parent and pos != root)
pos=pos.parent;
end while
if(pos==root)
return "No successor element"
else
posright= right sibling of post;
while(posright is not equal to leaf node)
posright=postright.left;
end while
return posright
}
```
$\square$

(b) (4 points) Show that if the input element x is chosen *uniformly at random* from T, then your procedure from part (a) runs in *expected* time $O(1)$.

Solution:
Observe, There will be one element for which we will have to traverse the whole tree till root and then down to leaf(Rightmost element of leftsubtree to Leftmost of element of right subtree).
There will be two Elements for which we will have to traverse till height h-1 and then down to leaf.
Similarly

1	2*(h-0)
2	2*(h-1)
4	2*(h-2)
.	
.	
2^{h-1}	2*(h-(h-1))
2^h	2*(h-h)

$h = \log n$
Average time will be
T(n) $= (2^{h-1} * 2 + 2^{h-2} * 2 * 2 + \ldots + 2 * 2 * (h-1) + 1 * 2 * h)/n$
T(n) $= (2/n)(1 * 2^{h-1} + 2 * 2^{h-2} + \ldots + 2 * (h-1) + 1 * (h))$
T(n) $= (2/n)(1 * h + 2 * (h-1) + \ldots + 2^{h-2} * (h-(h-2)) + 2^(h-1) * (h-(h-1))$
T(n) $= (2 * 2^h/n)(h/2^h + (h-1)/2^{h-1} + \ldots + 2/2^2 + 1/2^1)$
Since $2^h = n$
T(n) $= 2 * (h/2^h + (h-1)/2^{h-1} + \ldots + 2/2^2 + 1/2^1)$
T(n) $= 2 * \sum_{j=1}^{h} j/2^j$
Lets find this by small trick, for $x < 1$
$\sum_{j=1}^{\infty} x^j = \frac{1}{1-x}$
Take derivative on both sides
$\sum_{j=1}^{\infty} j * x^{j-1} = \frac{1}{(1-x)^2}$
Multiply both sides with x and put x=1/2
$\sum_{j=1}^{\infty} j * (1/2)^j = \frac{1/2}{(1/2)^2}$
$\sum_{j=1}^{\infty} \frac{j}{2^j} = 2$
Since $\sum_{j=1}^{\infty} \frac{j}{2^j} \leq \sum_{j=1}^{h} \frac{j}{2^j}$

Therefore, substituting this value in our result
T(n) = 2*2=4=O(1)

□

Assume that we wish to augment our 2-3 tree data structure so that that each node v maintains a pointer $v.succ$ to the successor of v, so that queries for the successor of an element can be answered in $O(1)$ time *worst-case*.

(c) (6 points) Show that the 2-3 trees can be augmented while maintaining $v.succ$, such that the Insert and Delete operations can still be performed in $O(\log n)$ time. (**Hint**: Think of a linked list.)

Solution: This problem can be solved using the linked list data structure. . The smallest(first) node's successor element can be stored in say top of linked list. The largest element successor link will currently point to NULL. Each leaf node will have a link to its successor. This way, we can find out successor of any element in time O(1).

Now, whenever an element is inserted, we will need to make two or update depending upon the inserted element. Suppose x is inserted into the 2-3 tree.
First of all, we will compare x with the smallest element by going to the left most of the tree, which will take O($\log n$) time. We can directly compare x with the largest element in O(1) time because the root will have the maximum value.
Intitally we will insert x normally and then
There will be 3 cases

1. If x is smallest
In this case we will make new node successor's link to top and then assign top to new node. It will take O($\log n$) time because we have to find the smallest element first.

2. if x is largest
In this case we will find the right most element first and we will add a link to it to point to the new node. Then we will add a link to the last node and point it to NULL. This procedure will also take time O($\log n$)

3. If x has a predecessor as well as successor
In this case, we will have to find the position where x is inserted first. Then we will have to find the successor and predecessor of x which can be done in O($\log n$) time. Then we will have to make the following changes
(a). New node link now points to the successor of it.
(a). Make predecessor link now points to the new node
This whole procedure will also take O($\log n$) time

Now suppose we have to **delete an element x** from the 2-3 tree
There will again be 3 cases
1. If x is smallest i.e. it is at the top of the linked list

We will make it successor as top and delete the node x. Time taken will be O(1) in this case.

2. If x is largest i.e. it is equal to the root of the tree
We will find its predecessor and make its link point to NULL and delete x. This will take time $\mathrm{O}(\log n)$, since to find predecessor it takes time $\mathrm{O}(\log n)$

3. If x has a predecessor as well as a successor
We will find its predecessor and successor and then will change predecessor's link to point to successor. Then we will delete x. This will also take time $\mathrm{O}(\log n)$ since finding predecessor and successor takes $\mathrm{O}(\log n)$ time.

Hence for each new element insertion and deletion, these operations can be performed in $\mathrm{O}(\log n)$ time and each node successor can be maintained.

$\square$