CSCI-GA.1170-001/002 Fundamental Algorithms October 16, 2014

Solutions to Problem 1 of Homework 6 (10 points)

Name: Keeyon Ebrahimi *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: In order to solve this solution we will first need create an *augmented* 2-3 tree. We repeat the next three steps n times.

1. Find Minimum - This process takes $O(log(n))$ time. We just keep going left at every branch, and once we hit the left most leaf, we have our min. *augmented* 2-3 trees are balanced meaning we know leaves are at height $log(n)$, so we know that we will be visiting $log(n)$ nodes.

2. Find m successors. By successors, I mean find the next greatest value. Also when I say m successors, I mean find the min's successor, then find that value's next min successor, and do that m times. This is at max $O(m \log(n))$ time. When finding successors, we first check if current leaf p has a right neighbor. If it does, then the successor is p's right neighbor. This case takes $O(1)$ to find. If p is the right most leaf, then we go up a tier. If this tier has a right neighbor, we check it's min. If this tier doesn't have a right neighbor, we go up another tier and try again. Also, if we hit the max, the max's successor is the tree's min.
   The worse case in finding the successor is if you have to go all the way back up the root and then go all the way back to the leaf, which only happens when you are at root nodes left branch max and right brancy max. This worst case takes time of $O(2log(n))$. So step two takes running time at worst of $O(2m\ log(n))$.

3. Now we have to print and delete this item. We have already found this item, now we have to delete, which still takes $O(log(n))$ time because we need to make sure the tree stays balanced after the deletion, which can be constant time, but mostly $O(log(n))$ time because we could potentially have to keep merging up branches all the way to the top.

   As for the total running time now, we have $O(creation)+n*(O(findingMin)+O(FindingMSuccessors)+O(deletion)$

   This equals $O(n\ log(n) + n * (log(n) + 2m\ log(n) + log(n)))$ which simplifies to

$$O(n\ log(n))$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 16, 2014

Solutions to Problem 2 of Homework 6 (18 Points)

Name: Keeyon Ebrahimi *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: We are only going to visit each node once, thanks to the PostOrder Tree Walk routine, and at each node visit, we are only going to do one simple comparison, which gives us the total running time of $O(n)$. For each node, we are also going to keep track of distinct $v.less$ values, and also a distinct $v.more$ values. We care about the $v.more$ values, because when moving in a PostOrder Tree Walk, a parent's $v.less$ will be the total of the left child's $v.less$ and also the total values greater than the left child that will be visited before getting back to the parent. So basically, $v.more$ contains all values below it that are greater than it self, or total distinct keys in its right branch. We care about this because a grandparent will add this to its $v.less$. Here is some pseudo code to represent this. We assume that the tree has all nodes with $v.less$ initially set to 0.

```
Node Fill-Less(Tree T,  Node v)
{
    //Base Case
    // We have no left children, meaning subtree containing
    // values less than self is 0

    if (v.leftChild == Null)
    {
        v.less = 0;
        return;
    }
    else
    {
        // if duplicate
        if (v.leftChild.key == v.key)
        {
            v.less = v.leftChild.less;
```

```
        }

        // if left child is not a duplicate and is less
        else if (v.leftChild.key < v.key)
        {
            v.less = v.leftChild.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: What we will be doing is a regular insert, but if we ever go to the left during the insert process, we must increase the parent's $v.less$ by one, and also all of the nodes in it's right tree will have their $v.less$ increased by one.

When calling this, we have the initial node be the root of the tree. Code on next page.

```
Insert(Node, Value)
{
    if (Node.key > value)
    {
        Node.vless = Node.vless + 1;
        // Increase everything in the right tree by one
        PostOrderTraversalIncreaseVLess(Node)

        //Check if left child exists for insert
        if(Node.leftChild != null)
        {
          Insert(Node.Left, Value);
        }
        else
        {
            // Duplicate
         if (Node.key == value)
         {
             Node.leftChild == new Node(value, Node.vless);
         }
         else
         {
                Node.leftChild == new Node(value, Node.vless - 1);
           }
        }
    }
    else (Node.key < value)
    {
        if (Node.rightChild != null)
        {
           Insert(Node.Right, Value);
        }
        else
        {
            // Duplicate
            if (Node.key == value)
            {
                Node.rightChild == new Node(value, Node.vless + 1);
            }
            else
            {
                Node.rightChild == new Node(value, Node.vless + 1);
            }
        }
    }
```

```
}

PostOrderTraversalIncreaseVLess(Node)
{
   If (Node.leftChild != Null)
   {
       PostOrderTraversalIncreaseVLess(Node.leftChild);
   }
   If (Node.rightChild != Null)
   {
       PostOrderTraversalIncreaseVLess(Node.rightChild);
   }
   Node.vless = Node.vless + 1;
}
```

□

(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: With a maintained $v.less$, we find the rank in $O(h)$ this way.

If node v has a left child, then $v.rank = v.leftChild.less + 1 + 1$ We add 1 to find the rank of the left child, and the second 1 to find the rank of the node v.

If however, v doesn't have a left child, then we have to check in its parent is less than v. If it is less than v, then we do $v.rank = v.parent.less + 1 + 1$, once again adding the two 1's to get the correct rank.
If the parent is greater than v, we keep on going up the tree until we find the parent that is less than v. Potentially going all the way up to the root. If we go all the way up to the root and still don't find any parents that are less than v, which only happens when v is the min, then we just return 1 as the rank, for we know we are at the min and that the min has a rank of 1.

The worst case for this is when we are at the min and we have to go up all levels the three, making us go up h stages, making the running time of this algorithm $O(h)$.

□

(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: Unfortunately we cannot do this in time of $O(h)$. This is because with insertions and deletions, we have to update the rank of everything that is greater than the value of what is being inserted or deleted. For example, lets say we insert a new element that is going to be the new min of the entire tree, giving it a rank of 1. If this is the case, we have to update the rank of every element that comes after the current node which is every other element in the tree, giving it a running time of $O(n)$ which is much greater than the desired $O(h)$. This goes for both inserting and deleting. □

CSCI-GA.1170-001/002 Fundamental Algorithms October 16, 2014

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

Name: Keeyon Ebrahimi *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: 3

The resulting height of this three will be 3. □

(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: It is not possible to get a better height than 3 in this case. With a two - three three, the only way a height can be bettered is if we have multiple levels of nodes that all only have 2 children. If this was the case, we could do some merging and make them have 3 children each and a smaller height. However, the reason this is not possible is because when we were doing inserts, we were always inserting to size three, and then splitting to compensate when we hit above three. This means that with our insertions happening like explained above, we will be filled with three child nodes, making the above merging of multiple 2 child levels impossible. □

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

Solution: You sure can. From the root, if you go left once, you can then have A, E, I grouped under there with a height of 2. From the root, if we go right then left, under this we can have grouping L, N, O, which will once again be at height three. Last but not least, if from the root we go right and then right, we can have R, S, T under this node, once again at height 3. □

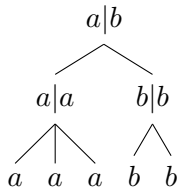
(d) (2 points) **Extra credit:** Can you achieve $h_{\min}$ for part (b) using an English word that is a permutation of $R, E, L, A, T, I, O, N, S$?

Solution: Yes you can. Take the word Orientals. If we insert $O, R, I, E, N, T, A, L, S$, we will get the exact same 2 - 3 three that we had in part b, which of course has the same height as part b. □

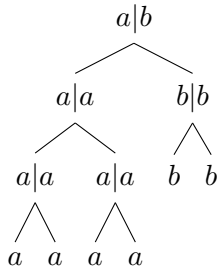
(e) (4 points) Give an example of a 2-3-tree T with exactly 5 leaves and two distinct values a, b, such that one gets different final trees if one first inserts a into T and then deletes b, instead of first deleting b from T and then inserting a. Explain what happened.

Solution: If we start with this three, we will have a difference depending on if we delete or insert first.

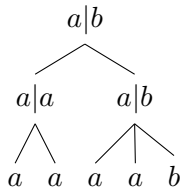
Starting Tree



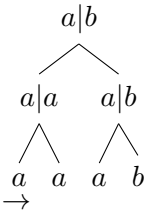
Tree with Insert A First, then Delete B



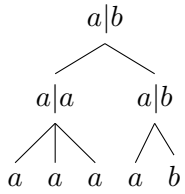
→



Tree with Delete B, then Insert A



→



There is a difference based on the order of the merge. After the first step, we will have all nodes with 2 children. When the second step is the Deletion of B, we know merge the

remaining b with the greatest instance of a, which is at the very right. Now the subtree at the right will have 3 children, as drawn above. In the later example when we insert A last, we know want to merge with the earliest and least value of A, so we put this to the very left, now making the very leftmost sub tree have three children. When we merge, we are merging either to the greatest value or to the lowest value based on the order, and that is what gives us the different trees. □

CSCI-GA.1170-001/002 Fundamental Algorithms October 16, 2014

Solutions to Problem 4 of Homework 6 (14 points)

Name: Keeyon Ebrahimi *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:
We first check if current leaf p has a right neighbor. If it does, then the successor is p's right neighbor. This case takes $O(1)$ to find. If p is the right most leaf, then we go up a tier. If this tier has a right neighbor, we check it's min. If this tier doesn't have a right neighbor, we go up another tier and try to find that tier's min again. We keep going up tiers for as long as we have to. The worse case in finding the successor is if you are at the roots left branches max. Then you have to go all the way back up the root and then go all the way back to a leaf located at the the roots right branches min. This worst case takes time of $O(2log(n))$. □

(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: As explained above, for $\frac{2}{3}$ of the cases, left most and middle child in a three grouping of 2 - 3 tree, we will be able to find the successor just right next to itself, so the time to find that value is Constant. Even when this is not the case, we have to only go up one tier, go to the right once and find it's min, which is its left most child. This then takes only three steps, which is still considered $O(1)$ time. There is only one case where we do not fit in these two constant categories, and that is when we are at the maximum of the left branch of the root. In this case, the running time is $O(log\ n)$, but because every other case is constant and there is only one exception, when chosen *uniformly at random*, it is safe to say that the expected time is $O(1)$. □

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: We will have to do some work with finding the predecessor, which is the exact opposite of the successor. The predecessor is the value that comes right before the current node. To find this, we just copy the steps of finding the successor, but instead of looking to its right, or going up tiers and looking at its right min, we instead look at its left neighbor, or we go to up tiers, looking at a left subbranch and finding it's max. Finding the predecessor is

the same process as finding the successor, but we look in the opposite direction. This being the exact same process as the finding successor, this process takes running time of $O(log\ n)$.

With the deletion, what we will be doing is we will first find the deleted node's predecessor. We will then change that predecessor's $v.succ$ to the soon to be deleted node's $v.succ$. After the predecessor's $v.succ$ has been updated, we can now safely delete the node. This keeps the $O(log\ n)$ running time of finding the predecessor, plus a little work on the top which is just constant. So with the deletion time added to the finding predecessor time we have $O(2\ log(n))$, which is just $O(log\ n)$.

With the insertion, what we will be doing is we will first insert the value normally. At this point we will find the new inserted value's predecessor and setting the predecessor's $v.succ$ to this newly inserted node. At this point, we will then set the newly inserted node's $v.succ$ to whatever the predecessor's $v.succ$ was before changed it just barely. Just like the deletion, the insertion takes $O(log\ n)$ time and the finding predecessor takes $O(log\ n)$ time with some constant time $v.succ$ updating, meaning that just like the deletion, the insertion also takes a total of $O(log\ n)$ time.

□