ESO207A Theoretical Assignment 2

Deepanshu Bansal(150219) Mukul Chaturvedi(150430)

March 9, 2017

Q1

Let elements are in sorted array a[] and it is filled upto n places and integer s to be found. Our algorithm return -1 if value is not present in array and its index if its present.
Pseudocode is as follows and we call FINDINDEX(a,s).

```
function BINSRCH(int a[],int start,int end,int s)
    if ($start \leq end$) then
        $mid \leftarrow (start + end)/2$
        if ($a[mid] == s$) then
            return mid
        end if
        if ($a[mid] < s$) then
            return BINSRCH(a,mid+1,end,s)
        else
            return BINSRCH(a,start,mid-1,s)
        end if
    end if
    return -1
end function

function FINDINDEX(int a[],int s)
    $index \leftarrow -1$
    $stindex \leftarrow 1$
    $endindex \leftarrow 1$
    $isfound \leftarrow 0$
    while ($a[endindex] \leq s$) do
        if ($a[endindex] == s$) then
            $isfound \leftarrow 1$
            $index \leftarrow endindex$
            break;
        else
            $stindex \leftarrow endindex$
            $endindex \leftarrow stindex$
        end if
    end while
    if ($isfound == 0$) then
        $index \leftarrow BINSRCH(a, stindex, endindex, s)$
    end if
    return index
end function
```

Proof of correctness is as follows.
Lets see proof of BINSRCH is s is mid element of array then it return index or of s is greater than mid element then clearly if present then it has to be on the right side of array thus our program correctly computing it or if it is less than mid element than it has to be on the left side if present . Now all this happens if starting address is less than or equal to end address otherwise we have seen all acrross the array and element is not present hence returing -1.

Now FINDINDEX function return the index of element if present in the array. Let's say element is not present then inside its while loop it never enters if condition hence index value is not updated and also in BINSRCH it will return -1 since it is not there thus our algo gives -1 if element is not present.
If element is present then either inside if condition of while loop or in BINSRCH function index value is updated to the index of element . Thus our algorithm gives the correct index of s.

1	2	...	2^i	...	n	...	2^{i+1}	...
a_1	a_2	...	a_{2^i}	...	a_n	...	$a_{2^{i+1}}$	...

Now **time complexity** of above algorithm is O(logn).
As in the worst case we have to travel all across upto index n to get endindex in while loop of FINDINDEX function and lets say that n lies between 2^i and 2^{i+1}.
So we have to make i iterations in the loop and n = 2^i so i = log(n). So upto while loop step we have taken atmax log(n) time now if still we hadn't find s then we do the binary search in array between indexes find in while loop which again takes atmost log(n) time. Thus we may say :
For BINSRCH lets say n = 2^k and T(1) = 1 and k = log(n)

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/2^2) + 2c \\ &= T(n/2^3) + 3c \\ &= .... \\ &= T(n/2^k) + kc \\ &= T(1) + clog(n) \\ T(n) &= O(logn) \end{aligned}$$

$$\begin{aligned} T(FINDINDEX) &= T(whileloop) + T(binarysearch) \\ &= c_1log(n) + c_2log(n) \\ &= clog(n) \end{aligned}$$

Thus time complexity of above algorithm is **O(logn)**.

Q2

Let heights and widths are given in arrays height[] and width[] and we have to find maximum banner area. So **pseudocode** is as follows:
Assuming that top function of stack gives n+1 if stack is empty .

```
function MAXAREA(int h[],int w[])
    Initialize wf[] array to store sum of widths till $i^{th}$ building.
    $wf[0] \leftarrow w[0]$
    for (i ← 1 to (n-1) ) do
        $wf[i] \leftarrow wf[i-1] + w[i]$
    end for
    $wf[n+1] \leftarrow 0$
    Create an empty Stack S
    $maxarea \leftarrow 0$
    $currarea \leftarrow 0$
    $top \leftarrow 0$
    $i \leftarrow 0$
    while $(i < n)$ do
        if (S isempty or h[S.top] ≤ h[i]) then
            $push(S, i)$
            $i++$
        else
            $top \leftarrow S.top$
            $S.pop()$
            $currarea \leftarrow h[top] * (wf[i] - wf[S.top])$
            **if** $(maxarea \leq currarea)$ **then**
                $maxarea \leftarrow currarea$
            **end if**
        **end if**
    **end while**
    **while** S.empty = false **do**
        $top \leftarrow S.top$
        $S.pop$
        $currarea \leftarrow h[top] * (wf[n-1] - wf[top])$
        **if** $(maxarea \leq currarea)$ **then**
            $maxarea \leftarrow currarea$
        **end if**
    **end while**
    return maxarea
**end function**
```

Proof of correctness is as follows.
Currarea corresponding to a index is maximum area that we can get of all the rectangles with index building included in it(because it is area of rectangle with all the buildings that we can go from index to left with height of buildings greater than height of index building and similarly for right side).Now, maxarea is maximum of all currarea which means it maximum rectangle of rectangle area possible. Now the for loop calculates the sum of widths of buildings upto ith place. Output returned by the program is correct is ensured by the if statement which ensures that max area is returned. Thus our algorithm gives correct output.

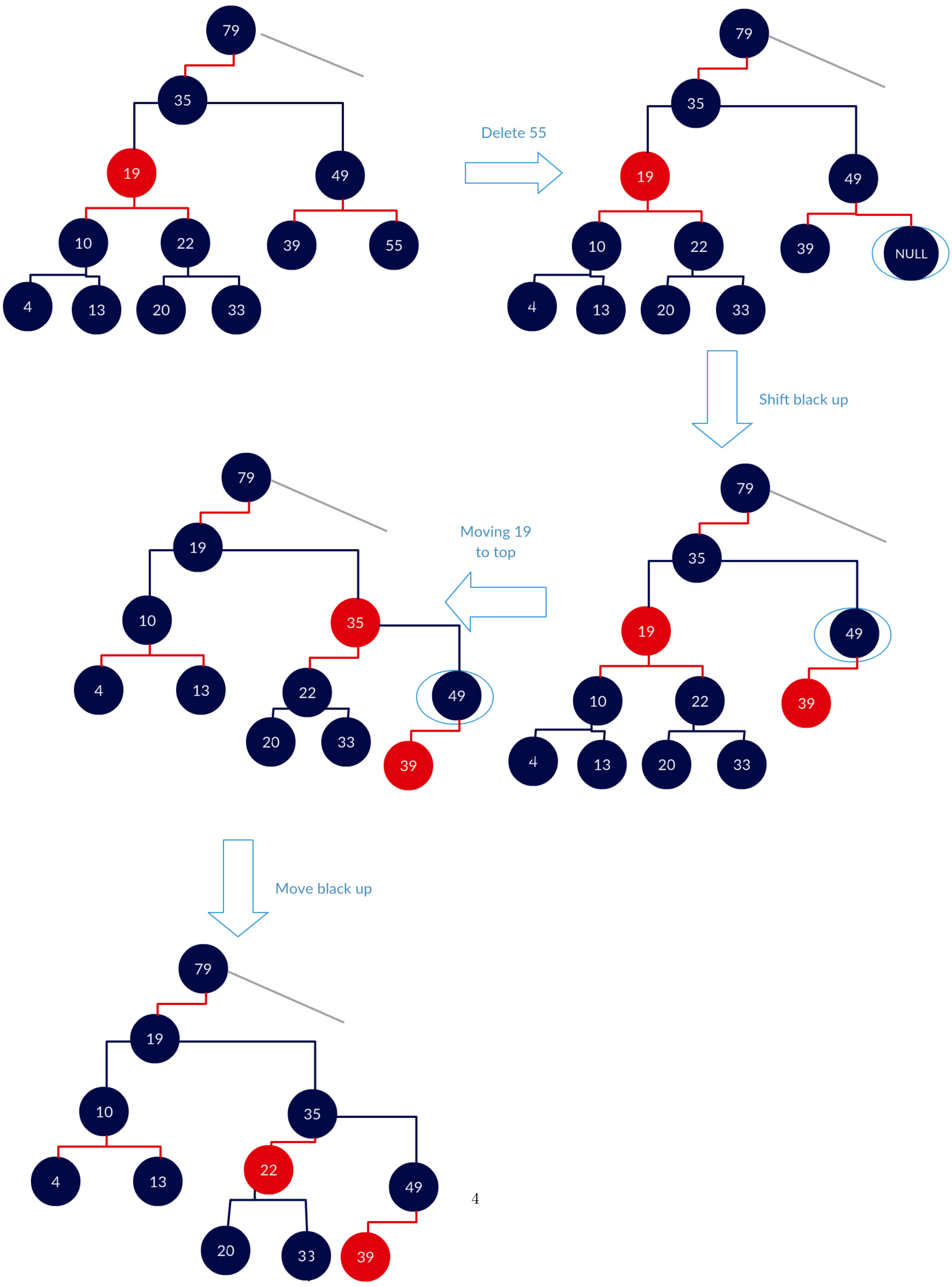
Now **time complexity** of above algorithm is O(n).
In the algorithm we first make wf array which takes constant time in each iteration so time taken to make it is c*n where c is some constant. Index corresponding to each building is pushed and popped into the stack one time only. Operation corresponding to index of a building is finding the currarea wrt it which takes constant time so finding all the currarea takes b*n time where b is a constant ie. while loop is iterated n times atmost with constant number of operations. Summing both these times is also takes order n times. So, time complexity of above algorithm is **O(n)**.

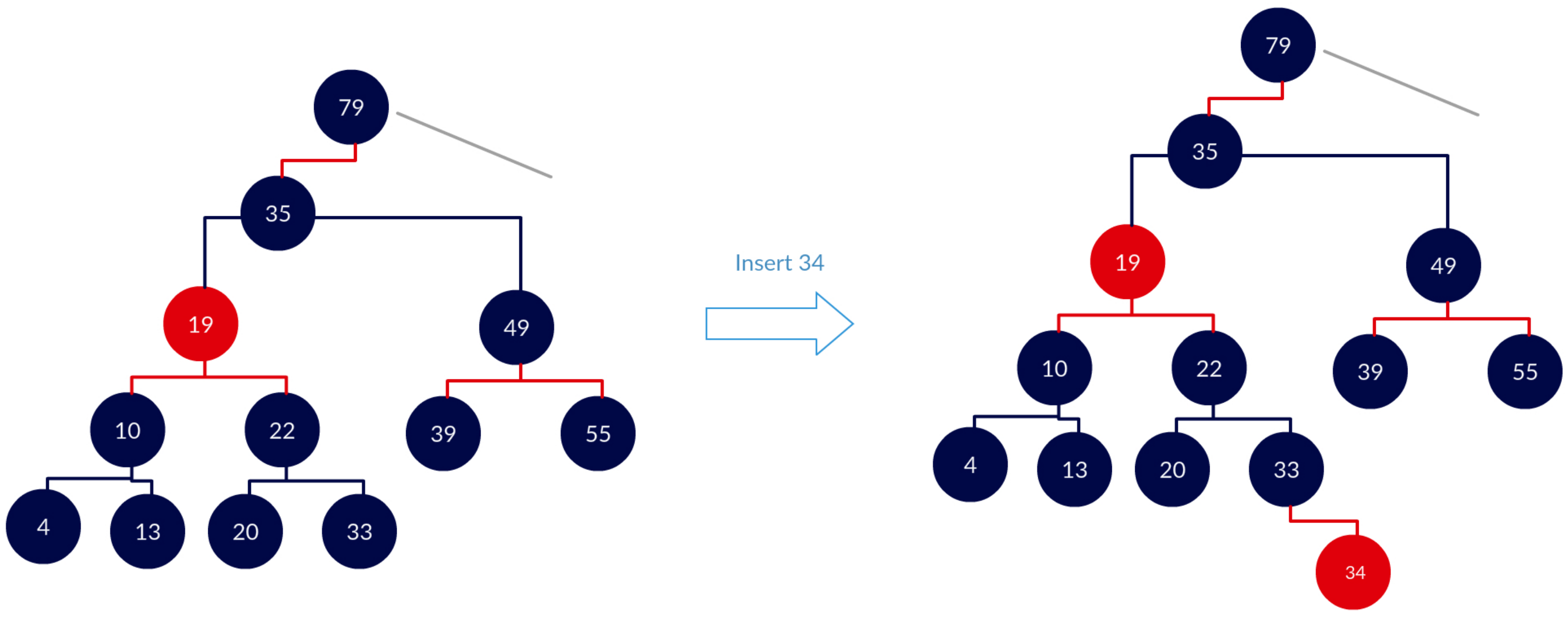
$$\begin{aligned} T(n) &= T(wfarray) + T(whileloop) \\ &= b * n + c * n \\ &= d * n \\ T(n) &= O(n) \end{aligned}$$

Thus time complexity of above algorithm is **O(n)**.

Q3

The red-black tree after deleting key 55 and the tree after inserting 34 in the original tree are as follows :


Delete 55
NULL
Shift black up
Moving 19 to top
Move black up


79
35
19
49
10
22
39
55
4
13
20
33
Insert 34
79
35
19
49
10
22
39
55
4
13
20
33
34

Q4

To find the predecessor of x we have following algo:

```
function PREDECESSOR(int x)
    if ($left[x] \neq NULL$) then
        return MaxElement(left[x]) // maximum element in left subtree.
    end if
    $y \leftarrow p[x]$
    while ($y \neq NULL$ and $x == left[y]$) do
        $x \leftarrow y$
        $y \leftarrow p[y]$
    end while
    return y
end function
```

Now **time complexity** of above algorithm is O(logn).
The running time of Predecessor on a tree of height h is O(h), since we either follow a path up the tree or follow a path down the tree. And we know that for red black h is $\leq$ logn.
Thus T(Predecessor) = O(h) = O(logn).

```
function KPREDECESSOR(int x,int k)
    $curr \leftarrow predecessor(x)$
    $i \leftarrow 1$
    while $i \leq k$ do
        $print(curr)$
        $curr \leftarrow predecessor(start)$
        $i++$
    end while
end function
```

Now for k predecessors time taken is O(logn + k). Let's say S is the first predecessor L is the last predecessor. When S and L are in the same path (S is the ancestor of L or L is the ancestor of S), the conclusion is obvious as we just move along the path and we get predecessors. Now any edge can be visited atmost two times. When S and L are not in the same path, let A be the smallest common ancestor of S and L. Consider the node S to A, now to move up can take atmost 2h steps as each edge is visited atmost twice. Thus A to L the extra cost of up to only 2h, The other is the normal back to the point takes atmost 2k steps.
Thus the number of steps or time taken for N is T(n) $\leq$ (2h +2 k) = 2(h+k)
ie. **T(n) = O(h+k) = O(logn + k)** as for red black tree h = log(n)

Proof of correctness is as follows.
If algorithm of predecessor algorithm is correct then algorithm of Kpredecessor is automatically correct predecessor of element is the one whose value is just smaller than x . Thus k^{th} predecessor of x is the predecessor of $(k-1)^{th}$ predecessor of x and so on.
Now for the correctness of predecessor algoritm we see if the left subtree is not NULL then value just smaller than x have to be present in it as it is also a BST and value in left subtree are smaller(or equal to) than the parent and if it is not present then the predecessor of x must be upto the way up till we move in left direction as for that node x it the successor as it is the smallest value greater than that present in tree thus our algorithm correctly gives the predecessor of x.

Deepanshu Bansal(150219)