Theoretical Assignment 2

Gupreet Singh - 150259 . Nikita Awasthi - 150453

March 9, 2017

Question 1

Assumptions

1. Rest of the elements (indexed from n) in the array are set to +infinity.
2. The first n elements in the array are finite, and in ascending order.

Pseudocode

Algorithm 1 Binary Search Algorithm

```
procedure binarySearch(A[0, 1...], key)        ▷ where A[0, 1 ...] is an array of infinite elements
    left = 0
    right = 1                                  ▷ where right defines an upper limit on n

    while A[right] ≠ inf do
        right = right * 2
    end while
    right = right − 1

    while A[left] ≠ key and left < right do
        mid = (left + right)/2                 ▷ where '/' is integer division

        if A[mid] == key then
            left = mid
            break
        else if A[mid] < key then
            left = mid + 1
        else
            right = mid − 1
        end if
    end while

    if A[left] ≠ key then
        return false
    else
        return true
    end if
end procedure
```

Proof of Correctness

The proof of correctness is similar to that of a normal Binary Search.

Since we assume the number of elements (n) to be finite, we can say that $\exists r$ s.t. $2^{r-1} \leq n \leq 2^r$. Clearly, here $r = \lceil log(n) \rceil$.

In the first loop of our algorithm, our motive is to find exactly $right = 2^r$, as this is a good upper limit on n. To do so, we set a variable right to 1, and keep on multiply two, until it is equal to r. We just check the elements on the index $right$, and if we encounter infinity, we know that $right \geq n$, and hence $right = 2^r$.

Now, we are left with '$right$' number of elements, which are in ascending order (as first n elements of array are finite), so we apply a simple binary search on this array for the first '$right$' elements.

Claim: If key exists in the array After the loop, in the array, then Array[left] = key
Proof (By Induction)

Induction Hypothesis: Assume it is true for all arrays with $length(array) < n$

Base Case: For $length(array) = 1$

$left = 0$
$right = 0$
$mid = (left + right)/2 = 0$

Case 1: $A[0] = key$, we set left to mid (0) and break
Case 2: $A[0] \neq key$, we set left to $mid + 1$ (1) and continue. Now, $left > right$, so loop breaks

Clearly, in the first case (i.e. key exists in the array), left is set to 0, and $Array[left] = key$

Inductive Step:

If the middle element is equal to the key, then we are done.

If the middle element is greater than key, then key exists (if at all) in the left half (i.e. all elements before middle one) of the array. Hence, we need only analyze the left half, which is of length $n/2$. Since from our induction hypothesis, our algorithm works for all array lengths less than n, we can say that we are done.

The case where middle element is less that key is symmetric to the upper one.

Hence, in all cases, we can say that our intended algorithm for binary search is correct.

In the second loop in our algorithm, we are finding the middle element and then comparing it to our key. After this, we are updating the $left$ or $right$ variables, according to the comparision result. Since our algorithm fits the above proof description, we can say that after the second loop, left contains the index of the key in the array, if it exists.

Hence, if $Array[left] = key$, we output 'true', else 'false'.

Complexity Analysis

Since the first loop is running until $right \neq 2^r$, and in each iteration, we are multiplying $right$ by 2, hence it will run r times, i.e. $log(n) + 1$ or $O(log(n))$ times, with time of each iteration $O(1)$.

Hence first loop requires $O(log(n))time$.

For the second loop, we are doing simple binary search, which takes $O(log(n))$ time.
Let the time for a binary search for n elements be T(n)
$\therefore T(n) = c + T(n/2)$ for some c

$\implies T(n) = clog(n)$
$\implies T(n) = O(log(n))$
Hence Total time is $O(log(n))$

Question 2

Reference: Largest Rectangle area in a Histogram

Pseudocode

Algorithm 2 Algorithm to get Maximum Banner Area

```
procedure getMaximumArea(widths[0, 1...n], heights[0, 1...n], n)
    for i in range(1, n − 1) do
        widths[i] = widths[i − 1] + widths[i]
    end for

    maxArea = 0
    stackbuildings                    ▷ Where we generate a stack of integers - buildings

    i = 0
    while i < n do
        if isEmpty(buildings) or heights[top(buildings)] < heights[i] then
            push(buildings, i)
            i = i + 1
        else
            top = pop(buildings)
            w = 0
            if isEmpty(buildings) then
                w = widths[i − 1]
            else
                w = widths[i − 1] − widths[pop(buildings)]
            end if
            maxArea = max(maxArea, buildings[top] ∗ w)
        end if
    end while

    while isEmpty(buildings) = false do
        top = pop(buildings)
        w = 0
        if isEmpty(buildings) then
            w = widths[i − 1]
        else
            w = widths[i − 1] − widths[pop(buildings)]
        end if
        maxArea = max(maxArea, buildings[top] ∗ w)
    end while

    return maxArea
end procedure
```

Proof of correctness

The basic idea is that we try to compute the area taking maximal sequences with every building as the shortest building and then find the maximum of all such areas. For this, we need to find the closest buildings which are shorter than the given building on the left as well as right and then find the area. The $maxArea$ variable dynamically stores the maximum area computed at every point. So, we need to find the left and right limit for each building.

Invariant: The stack at any point has an increasing order of the height of the buildings.
This can be seen as building is pushed only if the its height is greater than the height of the building at the top of the stack

Claim: For every case of a maximal sequence, with maximal area, there would be at least one building such that the banner covers the entire height of this building
Proof: Let us assume that this is not so, i.e. we have a maximal area sequence of buildings, for which no building is fully covered by the banner
For this sequence of buildings used to compute the area, there will exist a building of minimum height. By our assumption, this building is not totally covered by the banner. Now, we had to consider the case of maximum area. Taking the entire height of this building increases the areas, without over-covering any building as the height of this building is minimum, which contradicts with our assumption.
Therefore, for area to be maximum, there would be at least one building such that its entire height will be covered.

Claim: The leftmost index for computation of an area with a given building as the shortest building would be one greater than the preceding element in the stack, with the shortest building in that sequence being the building at the top of the stack.
Proof: As proved earlier that the stack at any point has an increasing order of height. Therefore the left index or the building no to be included would be the once preceding the popped building.

The algorithm finds the limits for the computation of the area with the building as the smallest building in the group of buildings. The $maxArea$ dynamically stores the maximum of all such areas.

Claim: During area computation for a sequence, the rightmost index for computation of an area would be the one less than the current index, with the shortest building in that sequence being the building at the top of the stack.
Proof: The buildings are pushed into the stack only if their height is greater than the top of the stack. Finally the top of the stack is such that the current index is the first smaller element after it.
In case, there is no element which is smaller than the given building and the traversal has been complete then each element can be popped such that now all elements to the right of the element being popped are included because their height is greater than recently popped element.

Complexity Analysis

As can be seen in the pseudocode, each building is pushed and popped only once and some constant time operations are performed for each stack. Therefore, it requires maximum two traversals of the each building representing the buildings.

$T(n) = 2c \times n$

$\therefore T(n) = O(n)$

Question 3

Deletion

The steps involved in the deletion are as follows:

- Simple deletion of the node 55. After deletion we can identify that the keys q, r, s are 49, 35 and 19 respectively. The tree would look like as shown in the figure 1

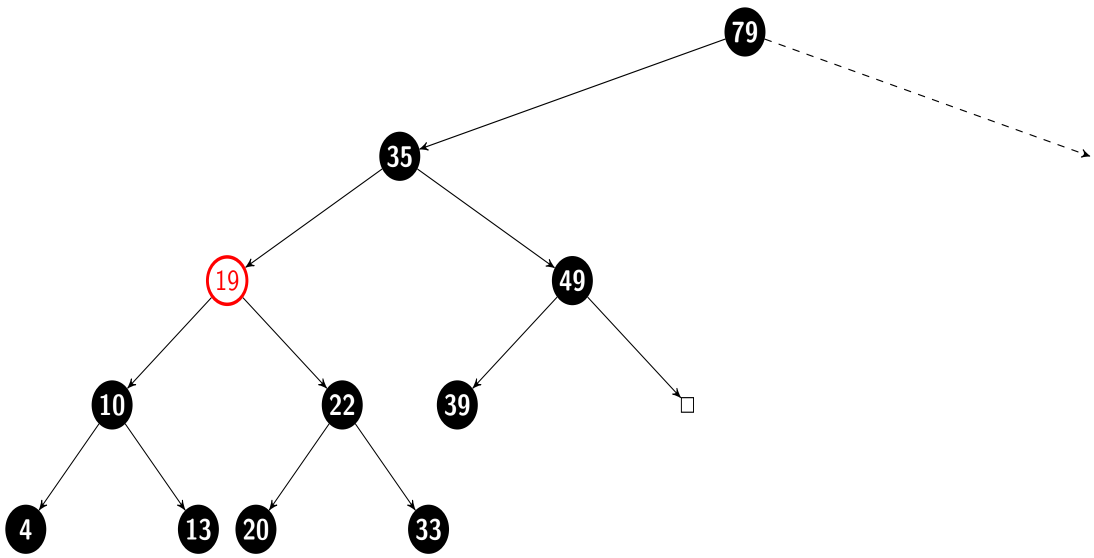

Figure 1: Step 1

- Since s is red, we right rotate about r and swap the colours of s and r. Our updated s is the node with value 22 which is now black. The tree would look like as shown in figure 2

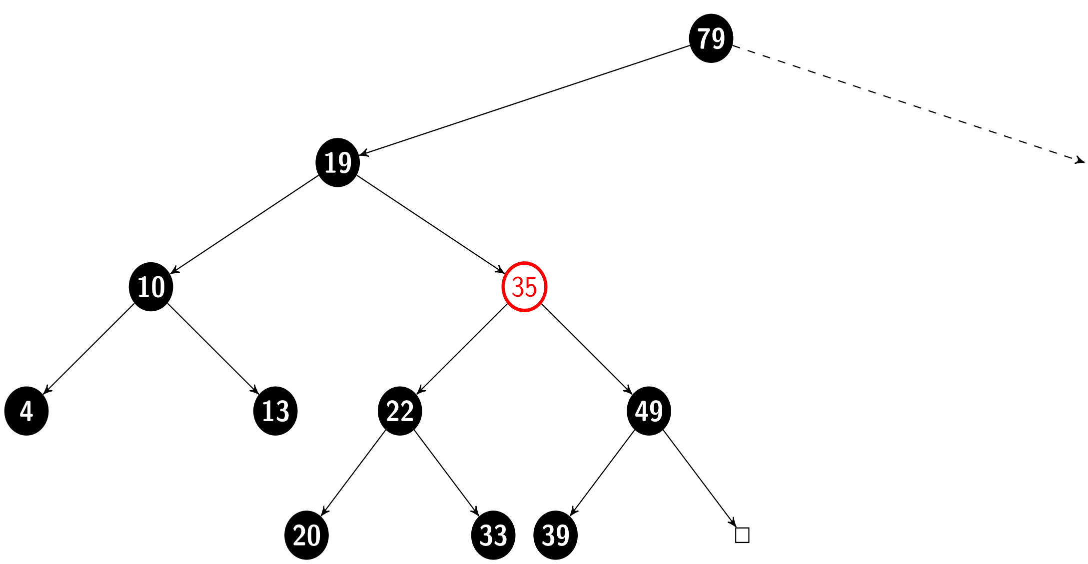

Figure 2: Step 2

- Swap colours of the nodes s and r and color the node with value 39 as red to maintain the black height of the tree. The final tree can be seen in figure 3

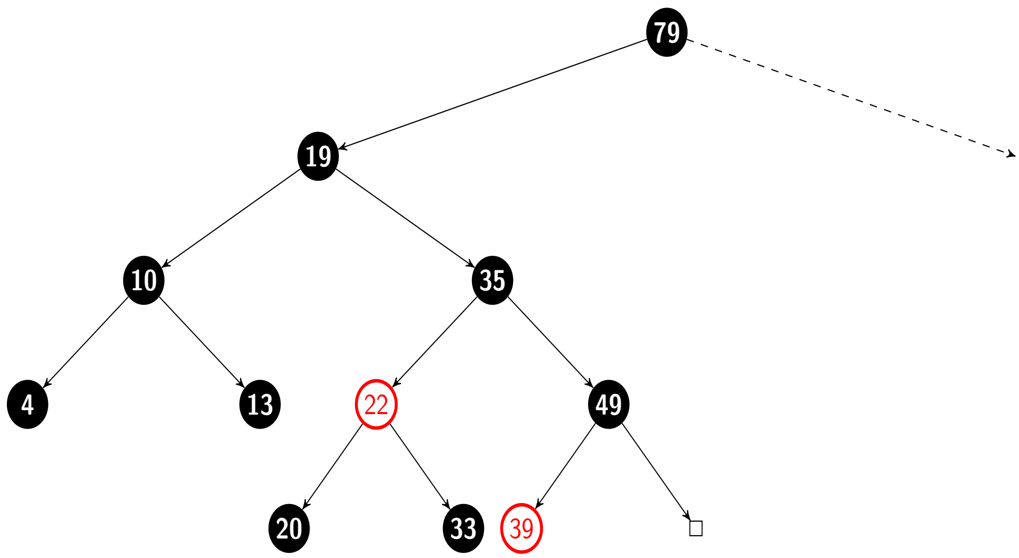

Figure 3: Red Black Tree after deletion

Insertion

The insertion of 34 is a trivial case with the addition of a red node with 33 as parent. The addition of the new red node does not violate any of the requirements of a Red-Black Tree. The final tree can be seen in the figure 4

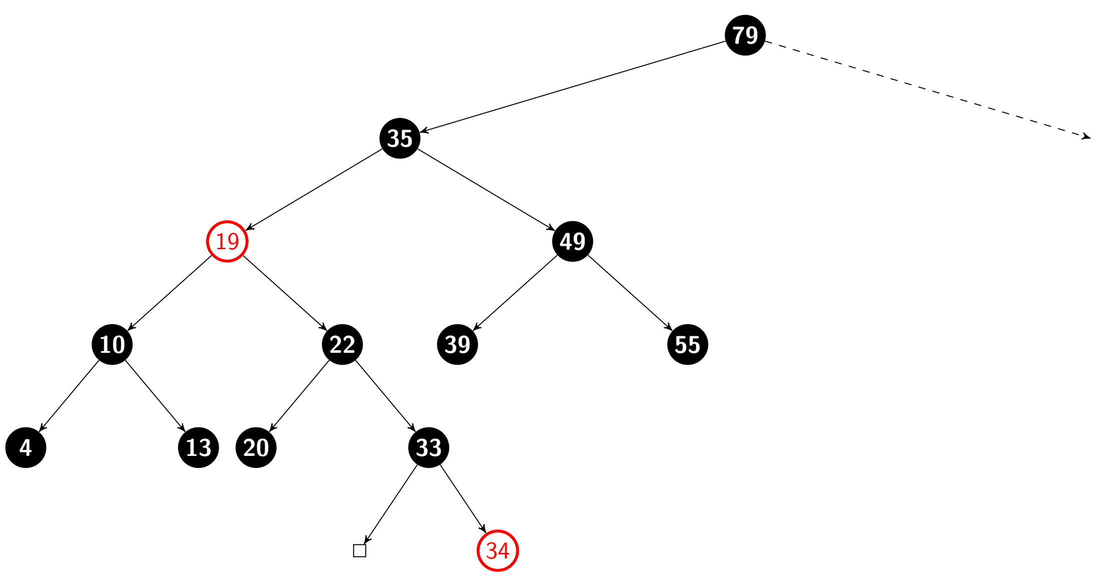

Figure 4: Red Black Tree after insertion

Question 4

New structure

We augment our red-black tree data structure such that our new structure includes the following:

- key
- pointer to left and right child
- colour
- predecessor
- parent

Pseudocodes

Algorithm 3 Algorithm to find k-predecessors

```
procedure findPredecessors(key, root, k)    ▷ where key contains the value of the node whose
predecessors we need
    node = root
    while true do
        if node ⟶ val = key then
            break
        else if node ⟶ val < key then
            node = node ⟶ right
        else if node ⟶ val > key then
            node = node ⟶ left
        end if
    end while

    preds[k] = NULL    ▷ where preds is a array of length k with all entries NULL
    count = 0
    while count < k and node ≠ NULL do
        node = node ⟶ predecessor
        preds[count] = node
        count = count + 1
    end while

    return preds
end procedure
```

Algorithm 4 Algorithm to find direct predecessor

```
procedure findPredecessor(node)        ▷ where node is the node whose predecessor we need
    pred = node
    if node ⟶ left ≠ NULL then
        pred = node ⟶ left
        while pred ⟶ right ≠ NULL do
            pred = pred ⟶ right
        end while

    else
        while pred ≠ NULL and pred ⟶ parent ⟶ left = pred do
            pred = pred ⟶ parent
        end while
        if pred ≠ NULL then
            pred = pred ⟶ parent
        end if
    end if

    return pred
end procedure
```

Algorithm 5 Algorithm to find direct successor

```
procedure findSuccessor(node)        ▷ where node is the node whose successor we need
    succ = node
    if node ⟶ right ≠ NULL then
        succ = node ⟶ right
        while succ ⟶ left ≠ NULL do
            succ = succ ⟶ left
        end while

    else
        while succ ≠ NULL and succ ⟶ parent ⟶ right = succ do
            succ = succ ⟶ parent
        end while
        if succ ≠ NULL then
            succ = succ ⟶ parent
        end if
    end if

    return succ
end procedure
```

Algorithm 6 Modified Insertion Algorithm

```
procedure modInsertRBT(key)                ▷ where key contains the value to be inserted
    node = insertRBT(key)                  ▷ insertRBT is the normal insert into RBT function

    succ = findSuccessor(node)
    pred = NULL
    if succ ≠ NULL then
        pred = succ ⟶ predecessor
        succ ⟶ predecessor = node
    else
        pred =findPredecessor(node)
    end if

    node ⟶ predecessor = pred

    return node
end procedure
```

Algorithm 7 Modified Deletion Algorithm

```
procedure modDeleteBST(key)                ▷ where key contains the value to be deleted
    node = searchRBT(key)                  ▷ searchRBT is the normal search in RBT function

    succ = findSuccessor(node)
    if succ ≠ NULL then
        succ ⟶ predecessor = node ⟶ predecessor
    end if

    deleteRBT(key)                         ▷ deleteRBT is the normal delete in RBT function

    return node
end procedure
```

Proofs of correctness

Key ideas

- *FindPredecessor* procedure gives the direct predecessor of the given node
- *FindSuccessor* procedure gives the direct successor of the given node

k-predecessors algorithm

The augmented red-black tree data structure includes the predecessor for each node. Our basic ideas is to keep a count of the predecessors and find the predecessor of the previous node computed till the count reaches k.

Claim: If k-predecessors present, then algorithm finds the k-predecessors
Proof (By Induction):

Induction Hypothesis: Assume that the algorithm finds $k-1$ predecessors

Base Case: Considering the case when $k = 1$

The case is trivial because our augmented data structure already has the predecessor as one of its attributes.

Inductive step:

According to the definition of predecessor, the predecessor of the predecessor of a node is the 2nd predecessor of the node. Similarly, the kth predecessor of a node is the predecessor of the $(k-1)$th predecessor of that node.

Since we already have the $(k-1)$th predecessor, its predecessor will be the kth predecessor of our given node. The predecessor of the $(k-1)$th node is already present in its structure so we are done. In case any of the predecessor of the the sequence is not defined, we store NULL and also stop the operation. Thus our sequence of predecessors is defined and valid.

Addition of node

From our discussion in class, we know that addition of a node in RBT has an $O(logn)$ complexity.

Claim: Augmentation of RBT does not change the time complexity for addition of node.
Proof: The modifications to the insertion algorithm can be seen in the pseudocode 6. As discussed in class, finding successor and predecessor have a time complexity of $O(logn)$.

The extra computations required include finding successor, finding predecessor and some constant complexity operations.

$$T(n) = 3c \times log(n) + an$$

$$\therefore T(n) = O(logn)$$

Therefore, the modifications preserve the time complexity for the addition of the node.

Deletion of node

We know that deletion of a node in RBT has an $O(logn)$ complexity.

Claim: Augmentation of RBT does not change the time complexity for deletion of node.

Proof: The modifications to the delete operation can be seen in the pesudocode 7. The changes to the operations includes finding the successor and some constant operations.

$$T(n) = 2c \times log(n) + an$$
$$\therefore T(n) = O(logn)$$

Therefore, the modifications preserve the time complexity for deletion of the node.

Query operation

There are no changes in the query operation so the time complexity is preserved.

Complexity Analysis

Our algorithm to find k-predecessors for a given value can be divided into two segments:

- Searching for the node with the given value
- Finding k-predecessors of the node

The search operation has a time complexity of $O(logn)$ as mentioned in the question and in the discussion in class.
Also, for finding the k predecessors, we require traversal through k elements each of which has the value of the predecessor stored in its structure. The while loop is run such that *count* varies from 0 to $k - 1$. Each iteration requires constant operation. Thus this segment has time complexity $O(k)$.

$$\therefore T(n) = c \times log(n) + ak$$

$$\therefore T(n) = O(logn + k)$$