

THEORETICAL ASSIGNMENT № 2

Harsh Sinha(14265), Deepak Gangwar(14208)

09/03/2017

Problem 1

The pseudo code is as follows.

Listing 1: Pseudo code – Search in an Infinite Array.

```
1  BOOL infinite_search{
2  isfound = FALSE;
3  i = 1;
4  s = s;
5  while (isfound == 0)
6  {
7      if (A[i] == s)
8          isfound = TRUE;
9      else if (A[i] < s)
10         i = i * 2;
11     else if (A[i] > s OR A[i] == empty)
12         modif_bin_search (i);
13 }
14 return isfound;
15 }
```

Listing 2: modif_bin_search.

```
1  void modif_bin_search (int i)
2  {
3      L = i / 2;
4      R = i;
5      mid = (L + R) / 2
6      while (mid != R)
7      {
8          if (A[mid] == s)
9              return;
10         else if (A[mid] < s)
11             L = mid + 1;
12         else if (A[mid] > s OR A[mid] == empty)
13             R = mid -1;
14     }
15 }
```

Proof of correctness

TODO fill here.

Time Complexity

The function `modif_bin_search` takes $\log(i)$ steps to give us a solution (Similar to a binary search). Now the original function in the worst case takes $\log(n)$ steps. Thus in the worst possible case our algorithm shall consume $2 * \log(n)$. Hence, the order of the given algorithm is $\log(n)$.

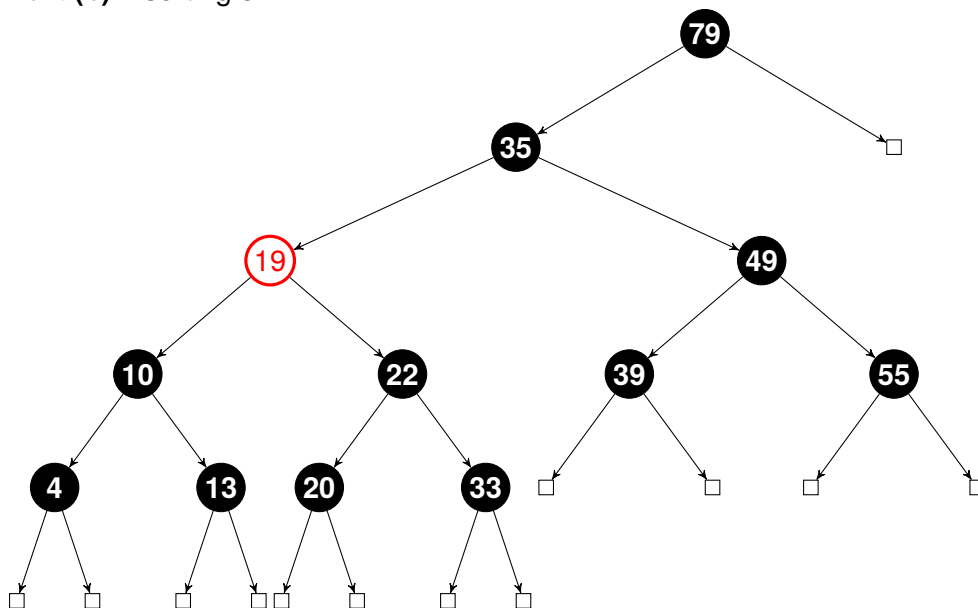
This can also be proposed by saying that the number of elements that this algorithm skips keeps on doubling every loop and so the order should be $\log(n)$.

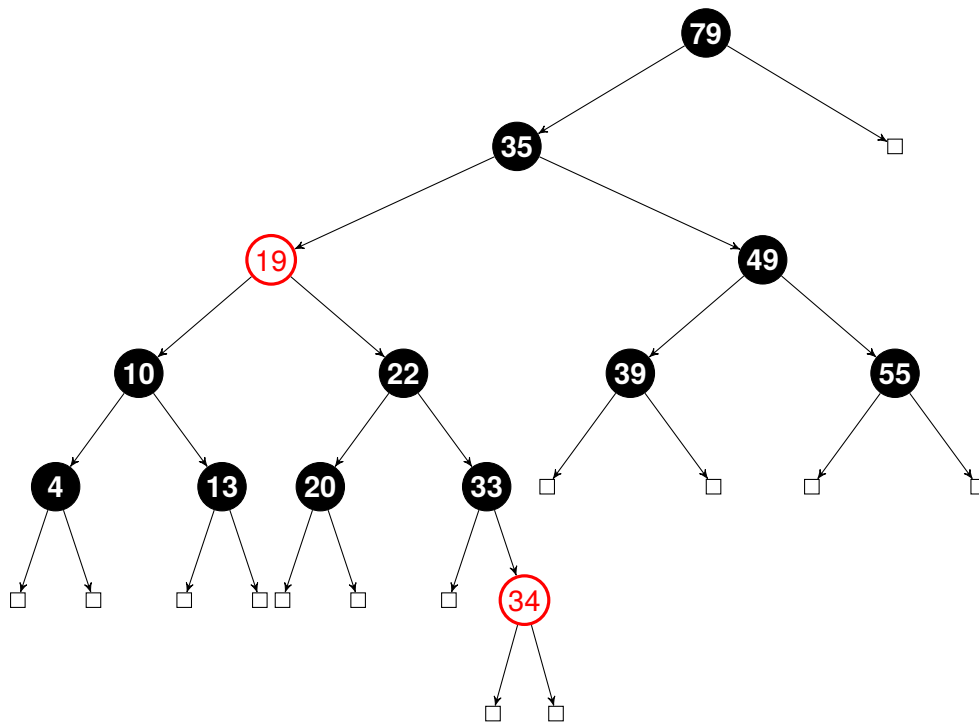
Notice that the base of $\log(n)$ is decided based on our choice of the multiplier in every step.

Problem 3

Part (a) Deleting 55

Part (b) Inserting 34





Problem 4

We propose a data structure where we augment to every node of the existing tree another value which is the predecessor of the node. Now for this data structure we will be able to maintain the $\log(n)$ time complexity for Insert, Delete and Query operations and yet find the required k predecessor in $O(k + \log(n))$. Although for this we shall have to change the Insert and Delete operations a bit. The details for the same are as follows.

The tree

A normal RB tree where each node, in addition to its own value also stores the value of its predecessor. So the structure of each node is like

Listing 3: Node Structure.

```

1 struct Node{
2     int val;
3     int pre; // Value of predecessor
4     struct Node *left,*right;
5 };

```

Now the algorithms

Algorithm for k predecessor

Listing 4: k predecessor

```
1 void k_pred (int num_predecessor, int value_to_search, Node * root)
2 {
3     key ← value_to_search;
4     ptr ← search (key, root);
5     if(ptr == NULL)
6         return;
7     k ← num_predecessor;
8     print(ptr.pre);
9     k_pred ( k - 1, ptr.pre, root);
10 }
11
12 Node * search (int key, Node * root)
13 {
14     p ← root;
15     while(p != NULL)
16     {
17         if (p.val == key)
18             return p;
19         else if (p.val > key)
20             p ← p.left;
21         else p ← p.right;
22     }
23 }
```

Proof of correctness for k predecessor

Since this is a recursive algorithm, after every call the value that it prints is basically the immediate predecessor and the function sets up for $k - 1$ sequential calls. Thus, as after every call we have one predecessor and we have the next call then. This continues till we have exhausted all our calls which was equal to the number of the predecessors to print. Hence, we can conclude that this algorithm gives the correct outputs at the end of recursive calls.

Time Complexity for the above algorithm

For each query the search effectively does Binary Search and then returns the pointer to it. Thus its time complexity will be $\log(n)$ as is known. Now, each call of k_pred takes $T(k)$ time, such that $T(k) = 4 * T(1) + T(k - 1) + O(\log(n))$. Hence, this algorithm is of time $O(k + k * \log(n))$.

Changes to Insert function

The only change we need to make in Insert is to search for the predecessor of the inserted value and augment this into the newly inserted node. Now, search for the predecessor shall be

of order $\log(n)$, insert itself is of order $\log(n)$. Hence, insert is still done in $O(\log(n))$.

Changes to Delete function

In deletion we need to store the predecessor of the deleted node in a temporary variable and then after deletion travel to the successor of the deleted node and change its predecessor from the previous value (equal to the deleted node) to the value stored in the temporary variable. This too similar to the above case will be done in $O(\log(n))$.