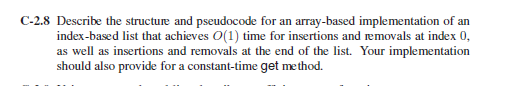
CS600 Homework 2

Atiq Patel CWID: 10432883



**Solution:**

One way to achieve O(1) time for insertions and deletions at index 0 or at the end of the list can be used using Circular Array.

Consider an Array “L” with max capacity “maxcap” filled with number “n” of elements “e”.

**Algorithm InsertBegin(e)**:

**Input:** An element e to be inserted at the beginning of index list L with max capacity “maxcap” and number of elements “n” filled in array.

**Output:** A display of list L with the element e to be inserted at the index.

**if** maxcap = 0 **then**

L[0]🡨e

**if** maxcap=n **then**

return ”Array is Full”

**else if** p>0 **then do**

p=p-1

**else**

p=maxcap-1

L[p] = e

n++

**Algorithm DelFirst(e)**:

**Input:** There are no inputs required

**Ouput:** The Array L with the removed element “e” at the beginning of the list

**If** maxcap=0 **then**

Return “the list is empty”

**else**

e=L[p]

p = (p+1) mod maxcap

n🡸n-1

**Algorithm addLast(e):**

**Input:** The element e to be added at the end of array.

**Output:** The Array L with the element e added at the last positon.

**If** n=maxcap **then**

**return** “The array is maxed out.”

**else do**

n=(n+1) mod maxcap

L[n]=e

**Algorithm DelLast()**:

**Input:** There is no input required.

**Output:** The Array L with deleted element “e”.

**if** n=0 **then**

return “The array is empty.”

**else do**

e = L[n]

n🡨n-1

return e

**Algorithm get(i):**

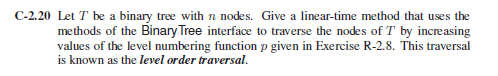
**Input:** The index position I in the array.

**Output:** The value present at the index “i” of the array L.

Pop=(i+p+1) mod maxcap

Return L[pop]

**All of these algorithms have their Time Complexities as O(1)**.



**Solution**:

**Algorithm levelordertraversal(T)**:

**Input:** The root node of the tree.

**Output:** The display of all the nodes of the tree in the order traversed.

Queue TQ

Traveresed🡨null

Cnode = root

**while**(cnode!=null) **do**

Display(TQ.value)

**if** (cnode.leftchild != NULL) **do**

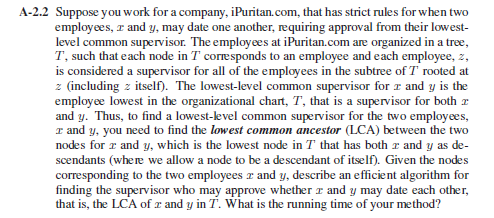
TQ.enqueue(cnode.leftchild)

**if** (cnode.rightchild != NULL) **do**

TQ.enqueue(cnode.rightchild)

TQ.Dequeue()

The time complexity for this algorithm is **O(n).**



**Solution:**

For this problem there can be two possible cases.

Case 1: When employee (node) x and (node) y are under the same supervisor i.e. same node.

Case 2: When employee x and y are in different subtrees.

**Algorithm LCA(node,x,y):**

**Input:** Root node and the employees x and y.

**Output:** A Least common ancestor (LCA) to the employees x and y.

**if** node = null **then**

Return “Tree isEmpty”

Case 1: when x & y are under the same node.

**if** node.content = x || node.content = y **then**

return node

LSubTree=LCA(node.left,x,y)

RSubTree=LCA(node.right,x,y)

Case 2: When both the employees are in different subtrees.

**If** LSubTree ! = NULL & RSubTree != NULL **then**

Return node

The Running time for this algorithm would be **O(h)** where h is the height of the tree.



**Solution:**

We know that the way Binary Search Tree is designed is that it has smaller element on the right of the parent and bigger element to the right of its parent. So to find the element with the smallest key or element. We just have to parse the left side of the tree.

**Algorithm ParseLeft(R):**

**Input:** The root node “R” of the BST.

**Output:** The element with smallest key.

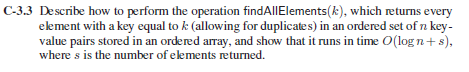
**if** (R.leftchild=NULL) **then**

return R

**else do**

ParseLeft(R.leftchild)

If R doesn’t have a left child which means that the R itself is the shortest node. In this case the algorithm provides **O(1).** If R has a Left Child then the algorithm will have **O(h)**, where h is the height of the tree.



**Solution:**

**Algorithm FindAllElements(K):**

**Input:** The Arraylist L stored in the form of “n” key ”k” value pairs.

**Output:** Returns every element with a key equal to k.

L🡨 0

H🡨 L.length -1

M🡨(L+H)/2

Start 🡨 0

End 🡨 0

**while** L<H **do**

M=(L+H)/2

**if** k<L[M] **then**

H🡨M-1

**else if** B=L[M] **then**

E=M

L=M+1

**else**

L=M+1

**for** i🡨S **to** E **do**

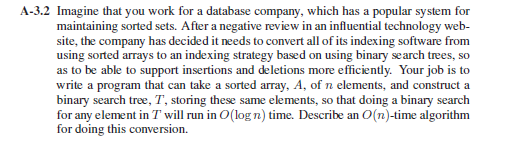
Display (i,L[i])

The running time for this algorithm is defined by two factors

1)The Binary Search algorithm which gives O(log n) &

2) Display the output of all the “k” keys. if there are s number of keys which gives O(s).

Hence the total running time for this algorithm is **O(logn + s)**



**Solution:**

We can convert the whole t

**Algorithm AtoBT(data,Start,End):**

**Input:** data which holds the content. Start specifies the start of the data and end specifies the end of the data.

**Output:** Successfully converting the array into a Binary Tree.

LeftChild = AtoBT(data,start,(end/2-1))

RightChild = AtoBT(data,(start/2-1),end)

**if** (data[start/2] !=NULL) **then**

root.value = data[end/2]

root.newleftchild = LeftChild

root.newrightchild = RightChild

data[end/2]= null

return root

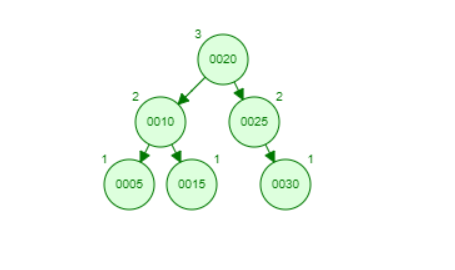
The Running time for this algorithm is **O(n)** because each node of the whole Tree is being visited.



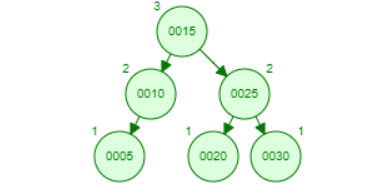
**Solution:**

Here is an example of Case 1 wherein all the elements are inserted in the order of 5,10,15,20,25,30

Case 2 where all the elements are inserted in the order of 30,25,20,15,10,5.



Case 1



Case 2

From the given diagram we can clearly see that Case 1!= Case 2. Thus we proved that Professor Amongus is wrong.



**Solution:**

By the concept of a red black tree,

*External-Node Property*: Every external node is black.

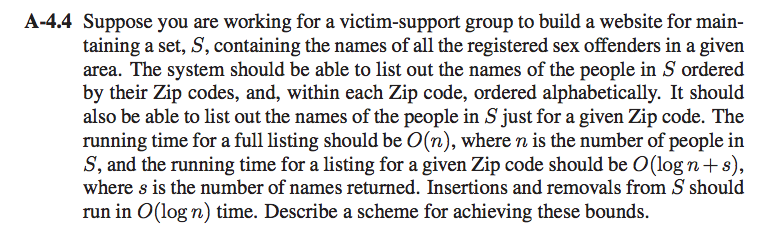
*Internal-node Property*: The children of a red node are black.

*Black-depth Property*: All the external nodes have the same *black depth*, that is,the same number of black nodes as proper ancestors. (given from the book Algorithm Design)

Thus, red nodes cannot have red children and black nodes cannot have black children either. The red and black nodes has to be alternating. So a maximum of 4 red nodes or 4 black nodes in the alternating way of 4. In this case the last node will be a red node with 2 null leaves.

Since the height is 8, the maximum number of height any branch can have is 5 wherein the 5th node of each path is a null leaf.

Thus, the minimum number of nodes with height 8 in a red black tree can have is **61**.



**Solution:**

On way to implement this by using Binary Search Tree. The root BST will have keys containing the zipp codes ret the children would have the keys containing all the names of the sex offenders.

**Traversal O(n):**

In this BST, to retrieve the list of all the sex offenders means to traversal and display of every node present in the tree. The parser will pass through all the elements present in the tree. So if there are n elements (people) the running time would be **O(n)**.

**Retrieval of a “s” names of given “n”0 zip codes O(log n+s):**

Retrieving of zip code means searching of a particular node in a BST which is an O(log n) operation.

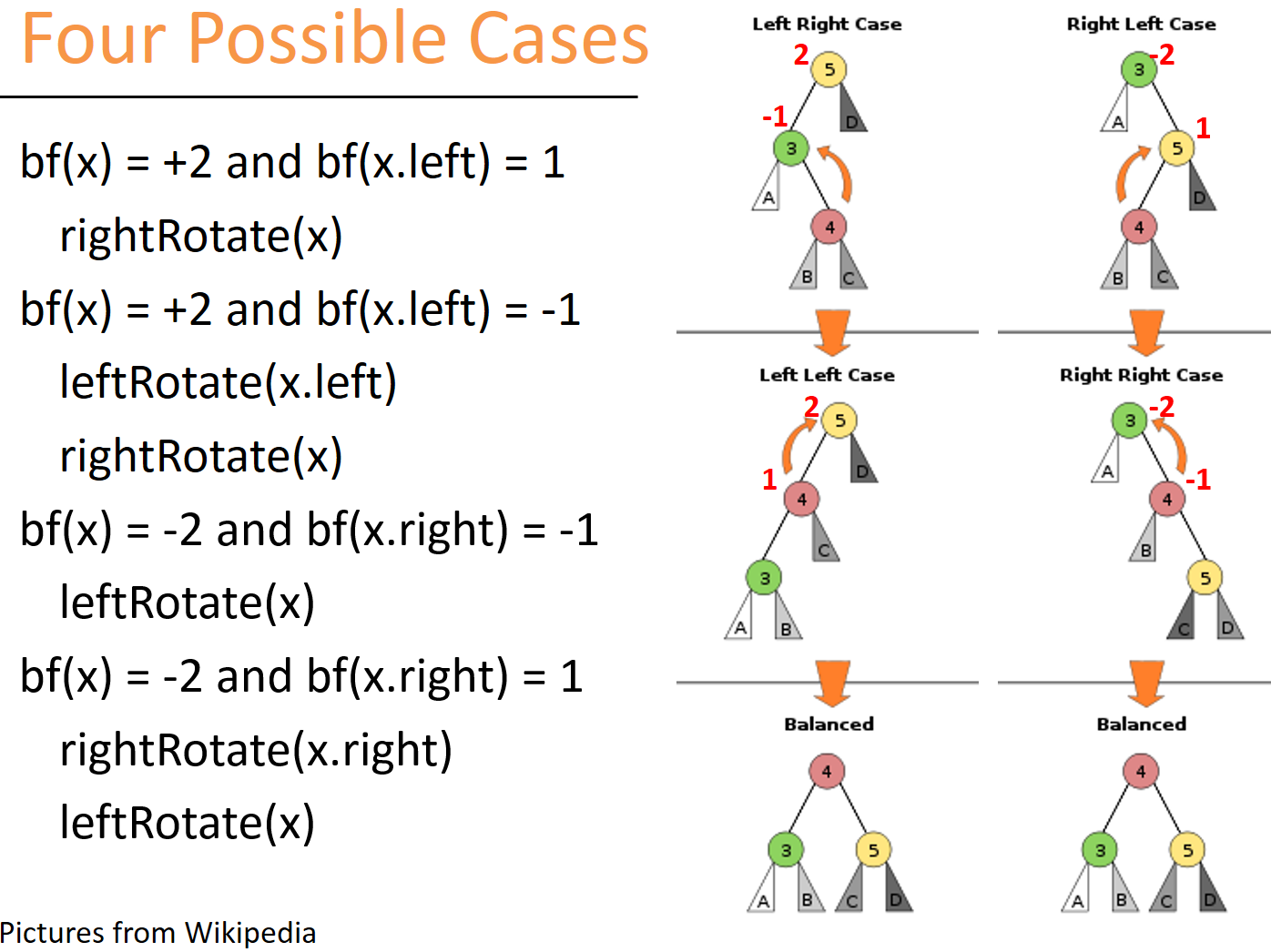
Suppose we are told to retrieve s number of names from the n number of zip codes, we will first have to the zip code and then search inside for the s number of names of people, then we will have **O(log n+s)**

**Insertion from S :**

Step 1: Perform insertion inside the BST for the node like any other standard BST insertion operation.

Step 2: Once insertion is done, check for the balance. If the tree doesn’t require balancing then insertion is done, else check for the first unbalanced node by traveling upwards. Let q be the first unbalanced node, v be the child of q that comes on the path p to q and x be the grandchild of q that comes on the path from p to q.

Step 3: Try to re balance the tree by performing set of appropriate rotations on the subtree rooted with node q. There are four possible cases that has to be handled as p, q and x can be arranged in 4 ways. Given below in the picture.



The running time for this algorithm is **O(log n)**.

**Deletion form S:**

Let node n be the node that we want to delete.

Step 1: Perform deletion inside the BST for the node like any other standard BST insertion operation.

Step 2: Once deletion is done, check for the balance. If the tree doesn’t require balancing then deletion is done, else check for the first unbalanced node by traveling upwards. Let q be the first unbalanced node, v be the child of q that comes on the path p to q and x be the grandchild of q that comes on the path from p to q.

Step 3: Try to re balance the tree by performing set of appropriate rotations on the subtree rooted with node q. There are four possible cases that must be handled as p, q and x can be arranged in 4 ways. Given in the picture above.

Both insertion and deletion operations in the BST is **O(log n)**

The running time for this algorithm is **O(log n)**.