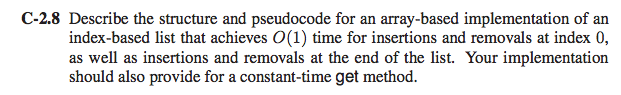
CS 600 Homework 2 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 09/11/2017



Solution:

We implement circular array to achieve 𝑂(1) time for each insertions and removals at index 0 and at the end of the list.

We must also keep in mind:

(1) starting element’s position (𝑓),

(2) The Total size of the array (𝑠𝑖𝑧𝑒)

(3) Elements Filled in the array (𝑛)

(4) Array A[ ]

**Algorithm:** 𝑎𝑑𝑑𝐹𝑖𝑟𝑠𝑡(𝑒)

**Input:** An element ‘e’

**Output:** The element ‘e’ added in first position in array

**if** 𝑠𝑖𝑧𝑒 = 𝑛 **then**

**return** “Array Full”

**else** if 𝑓>0 **then**

𝑓=𝑓−1

**else**

f = 𝑇𝑠𝑖𝑧𝑒−1

𝐴[𝑓] = 𝑒

𝑛=𝑛+1

**The Time Complexity is O(𝟏)**

**Algorithm**: 𝑟𝑒𝑚𝑜𝑣𝑒𝐹𝑖𝑟𝑠𝑡(𝑒)

**Input**: No Inputs

**Output**: 1st Element Removed, and returned

**if** 𝑛=0 **then**

return” empty list”

**else**

𝑒=𝐴[𝑓]

𝑓=(𝑓+1) 𝑚𝑜𝑑 s𝑖𝑧𝑒

𝑛=𝑛−1

**return** 𝑒

**The Time Complexity is 𝑶(𝟏)**

**Algorithm:** 𝑎𝑑𝑑𝐿𝑎𝑠𝑡(𝑒)

**Input**: element ‘e’ to be added in array

**Output**: New element added in the array at last position

**if** 𝑛=𝑠𝑖𝑧𝑒 **then**

return “List Full”

**else**

𝑛=(𝑛+1) 𝑚𝑜𝑑 𝑠𝑖𝑧𝑒

a[𝑛] = 𝑒

**The Time Complexity is 𝑶(𝟏)**

**Algorithm**: r𝑒𝑚𝑜𝑣𝑒𝐿𝑎𝑠𝑡(𝑒)

**Input**: No Input

**Output**: Last element from array removed and returned

**if** 𝑛=0 **then**

return “empty”

**else**

𝑒=𝑎[𝑛]

𝑛=𝑛−1

return 𝑒

**The Time Complexity is 𝑶(𝟏)**

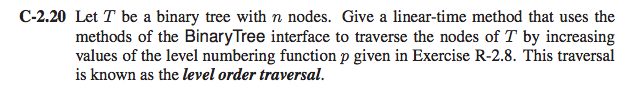
**Algorithm**: 𝑔𝑒𝑡(𝑖𝑛𝑑𝑒𝑥)

**Input**: Elements Index position in array

**Output**: returns the value of 𝐴[𝑖𝑛𝑑𝑒𝑥]

𝑔𝑒𝑡𝑓=(𝑖𝑛𝑑𝑒𝑥+𝑓+1) 𝑚𝑜𝑑 𝑠𝑖𝑧𝑒

**return** 𝐴[𝑔𝑒𝑡𝑓]



Solution:

To get a Linear time method for level order traversal we must use Queue.

As we are using Queue we traverse the tree in the form of increasing values of p.

We solve the problem using the following pseudo algorithm:

**Algorithm**: LevelOrderTraverse(root):

**Input**: The root node of the tree to be traversed.

**Output**: Nodes of the tree in the order required.

Queue treeQueue; //we set up a queue for the tree to be traversed

currentNode = root; // we start with the root node

while(currentNode ! = null)

{

print(currentNode->Content)

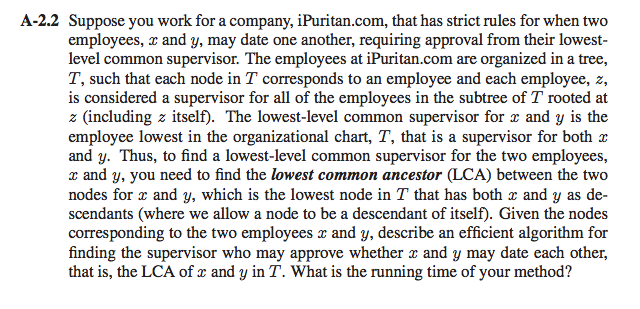
treeQueue.enqueue(currentNode.leftChild)

treeQueue.enqueue(currentNode.rightChild)

currentNode = treeQueue.Deque;

}

The traversal continues resulting in O(n) level order.



Solution:

Here we use the basic technique that is used for searching a key in a binary search tree. we directly look from data (𝑛𝑜𝑑𝑒.𝑑𝑎𝑡𝑎 which consists of values of both leftChild nodes and rightChild nodes) of the node rather that searching the leftChild 𝑜𝑟 rightChild

CASE:

1) When both 𝑥 and 𝑦 are under same node

2) When 𝑥 and 𝑦 are in different subtree

**Algorithm**: find𝐿𝐶𝐴(𝑛𝑜𝑑𝑒,𝑥,𝑦)

**Input**: variables x and y. Also pass 𝑟𝑜𝑜𝑡 𝑛𝑜𝑑𝑒 initially.

**Output**: A least common ancestor to the variables x and y

if 𝑛𝑜𝑑𝑒 = 𝑛𝑢𝑙𝑙 then

Return 𝑛𝑢𝑙𝑙

//Case 1

if 𝑛𝑜𝑑𝑒.𝑑𝑎𝑡𝑎 = 𝑥 𝑜𝑟 𝑛𝑜𝑑𝑒.𝑑𝑎𝑡𝑎 = 𝑦 then // x is root or y is root

Return 𝑛𝑜𝑑𝑒

𝑙𝑒𝑓𝑡𝑆𝑢𝑏𝑇𝑟𝑒𝑒=𝑓𝑖𝑛𝑑𝐿𝐶𝐴(𝑛𝑜𝑑𝑒.𝑙𝑒𝑓𝑡,𝑥,𝑦)

𝑟𝑖𝑔ℎ𝑡𝑆𝑢𝑏𝑇𝑟𝑒𝑒=𝑓𝑖𝑛𝑑𝐿𝐶𝐴(𝑛𝑜𝑑𝑒.𝑟𝑖𝑔ℎ𝑡,𝑥,𝑦)

//Case 2

if 𝑙𝑒𝑓𝑡𝑆𝑢𝑏𝑇𝑟𝑒𝑒 !=𝑛𝑢𝑙𝑙 𝑎𝑛𝑑 𝑟𝑖𝑔ℎ𝑡𝑆𝑢𝑏𝑇𝑟𝑒𝑒 !=𝑛𝑢𝑙𝑙 then

Return 𝑛𝑜𝑑𝑒

The Time Complexity is **𝑶(h)** since h is the height of the **entire tree**.

../../../../../Desktop/R-3.6.png

Solution:

In a Binary Search Tree, the Leftmost node will always have the smallest key & Rightmost node will always have the greatest key. (Principal of BST)

**Point to be noted**:

1] If the left link of the root is null, the smallest key in a BST is the key at the root.

2] If the left link is not null, the smallest key in the BST is in the subtree rooted at that particular node on the left.

PseudoCode:

**Algorithm**: SearchSmallest(root):

**Input**: The root node of the Binary Search Tree denoted by ‘root’

**Output**: The desired smallest node in the tree having root node by key.

if(root.LeftLeaf == null) then

{

return root;

}

else

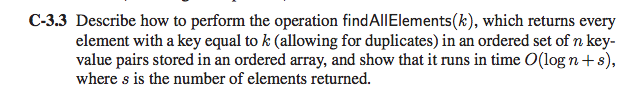
{

return SearchSmallest(root.leftLeaf)

}

If root doesn’t have a leftChild, then the root itself is the smallest key. Hence the Best Time Complexity would be 𝑂(1)

If root has the left Child, **then the Time Complexity would be 𝑶(𝒉),** where h is the height of the tree.



**Solution**:

One solution to this is to use Binary Search technique to find the number which has a time complexity of 𝑂(𝑙𝑜𝑔 𝑛).

We should find all nodes having the key ‘k’. To do this we must have 2 variables which will give us the starting and ending positon of key k in the ordered array i.e. start and end.

In the algorithm, the 1st while loop finds the start of the key ‘k’ and the 2nd while loop finds the end of the key ‘k’.

**Algorithm** 𝑓𝑖𝑛𝑑𝐴𝑙𝑙𝐸𝑙𝑒𝑚𝑒𝑛𝑡𝑠(𝑘)

**Input**: The array A with all numbers sorted and a key ‘k’

**Output**: The list of position-key pairs ‘I-k’ returned from the ordered array where I is the position of the key in the array.

𝑙𝑜𝑤← 0

high← 𝐴.𝑙𝑒𝑛𝑔𝑡h −1

𝑠𝑡𝑎𝑟𝑡← 0

𝑒𝑛𝑑← 0

//To Find Start

while 𝑙𝑜𝑤<high do

𝑚𝑖𝑑= (high+𝑙𝑜𝑤)/2

if 𝑘<𝐴[𝑚𝑖𝑑] then

high=𝑚𝑖𝑑−1

else if 𝑘 = 𝐴[𝑚𝑖𝑑]

𝑠𝑡𝑎𝑟𝑡=𝑚𝑖𝑑

high=𝑚𝑖𝑑+1

else

𝑙𝑜𝑤=𝑚𝑖𝑑+1

//To find End

while 𝑙𝑜𝑤<ℎ𝑖𝑔ℎ do

𝑚𝑖𝑑= (ℎ𝑖𝑔ℎ+𝑙𝑜𝑤)/2

if 𝑘<𝐴[𝑚𝑖𝑑] then

ℎ𝑖𝑔ℎ=𝑚𝑖𝑑−1

else if 𝑘 = 𝐴[𝑚𝑖𝑑] then

𝑒𝑛𝑑 =𝑚𝑖𝑑

𝑙𝑜𝑤=𝑚𝑖𝑑+1

else

𝑙𝑜𝑤=𝑚𝑖𝑑+1

𝒇𝒐𝒓 𝑖=𝑠𝑡𝑎𝑟𝑡 𝒕𝒐 𝑒𝑛𝑑 𝒅𝒐

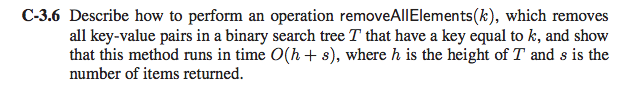
print 𝑖, 𝐴[𝑖]

Therefore the Time Complexity is

1) To find the start and end of the duplicate number of key ‘k’ is 𝑂(𝑙𝑜𝑔 𝑛)

2) To print all the key ‘k’. suppose there are ‘s’ number of keys 𝑂(𝑠)

**Hence, 𝑻𝒊𝒎𝒆 𝑪𝒐𝒎𝒑𝒍𝒆𝒙𝒊𝒕𝒚 𝒊𝒔 𝑶((𝒍𝒐𝒈 𝒏) +𝒔)**

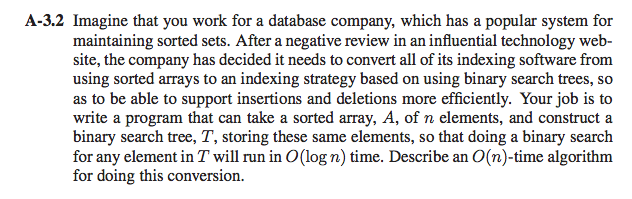


Solution:

Just like the previous problem, we find the key k. An O(log n) tree search whose max runtime is h (h being the height of the tree) is used to perform this. The first value of k leads us to all the other values of k because they exist in the same sub-tree of the node k.

We start with removing all the left-child leaves of the initial node k and rebalance the tree at every iteration, until the right child’s value is no longer k. Once the left and right children are not left with a value of key k, we remove the current node and rebalance the tree.

It takes a maximum of h searches to find node with key value is k. It takes s operations to remove s values having key k. And therefore, this operation runs for **O(h + s) time complexity**.



**Solution**: ArrayToBinaryTree(x, start, end):

**Input**: x which holds the data. start specifying the start of the content. end specifying the end of the content.

**Output**: Completed binary tree with root node.

Leftleaf2 = ArrayToBinaryTree (x, start, ((end/2)-1));

Rightleaf2 = ArrayToBinaryTree (x, ((start/2)+1), ((end/2)-1));

if(content[start/2]!=null) then

{

root.value = content[end/2];

root.Leftleaf = Leftleaf2;

root.Rightleaf = Rightleaf2;

content[end/2] = null;

return root;

}

**Time Complexity**:

Since, each node is traversed at least once, the time complexity is 𝑶(𝒏)