DATA STRUCTURE

WEEK Ten

Contact Details:

Email: sobia.iftikhar@nu.edu.pk

GCR:

BST

insert

Delete

FINDING PREDECESSOR & SUCCESSOR

```
47    public static void main(String args[]) {
48        Node root = new Node(20);
49        root.left = new Node(10);
50        root.right = new Node(30);
51        root.left.left = new Node(5);
52        root.left.left.right = new Node(7);
53        root.left.right = new Node(15);
54        root.right.left = new Node(25);
55        root.right.right = new Node(35);
56        root.left.right.left = new Node(13);
57       // root.left.right.left.right = new Node(14);
58
59        root.left.right.right = new Node(18);
```

```
<terminated> InorderSuccessorPredecessor [Java Application] C:\Users\sobia\.p2\pool\plug
Inorder Successor of 35 is : 0 and predecessor is : 30
Inorder Successor of 30 is : 35 and predecessor is : 25
Inorder Successor of 15 is : 18 and predecessor is : 13
Inorder Successor of 7 is : 10 and predecessor is : 5
Inorder Successor of 18 is : 20 and predecessor is : 15
Inorder Successor of 13 is : 15 and predecessor is : 10
```

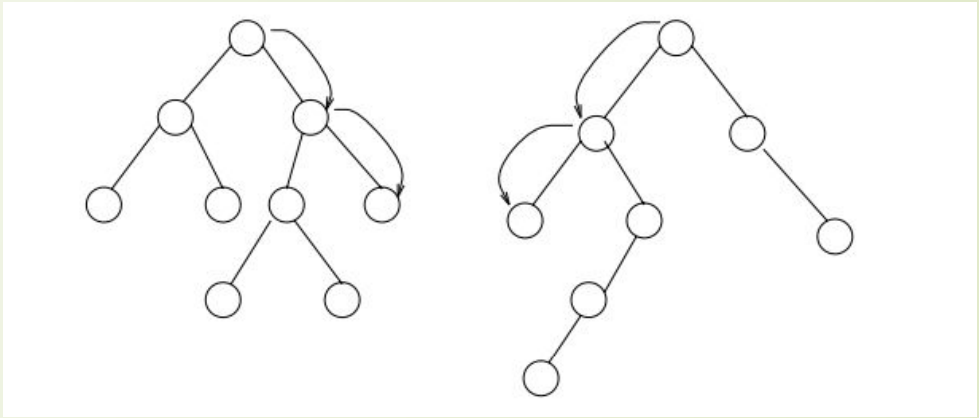
```
59        root.left.right.right = new Node(18);
60        InorderSuccessorPredecessor i = new InorderSuccessorPredecessor();
61        i.successorPredecessor(root, 35);
62        System.out.println("Inorder Successor of 35 is : " + successor
63                + " and predecessor is : " + predecessor);
64        i.successorPredecessor(root, 30);
65        System.out.println("Inorder Successor of 30 is : " + successor
66                + " and predecessor is : " + predecessor);
67        i.successorPredecessor(root, 15);
68        System.out.println("Inorder Successor of 15 is : " + successor
69                + " and predecessor is : " + predecessor);
70        i.successorPredecessor(root, 7);
71        System.out.println("Inorder Successor of 7 is : " + successor
72                + " and predecessor is : " + predecessor);
73        i.successorPredecessor(root, 18);
74        System.out.println("Inorder Successor of 18 is : " + successor
75                + " and predecessor is : " + predecessor);
76        i.successorPredecessor(root, 13);
77        System.out.println("Inorder Successor of 13 is : " + successor
78                + " and predecessor is : " + predecessor);
79  //      i.successorPredecessor(root, 14);
80  //      System.out.println("Inorder Successor of 14 is : " + successor
81  //              + " and predecessor is : " + predecessor);
    }
```

```
<terminated> InorderSuccessorPredecessor [Java Application] C:\Users\sobia\.p2\pool\plugi
Inorder Successor of 35 is : 0 and predecessor is : 30
Inorder Successor of 30 is : 35 and predecessor is : 25
Inorder Successor of 15 is : 18 and predecessor is : 14
Inorder Successor of 7 is : 10 and predecessor is : 5
Inorder Successor of 18 is : 20 and predecessor is : 15
Inorder Successor of 13 is : 14 and predecessor is : 10
Inorder Successor of 14 is : 15 and predecessor is : 13
```

Implementation

```
8⊖    public void successorPredecessor(Node root, int val) {
9         if (root != null) {
10            if (root.data == val) {
11                if (root.left != null) {
12                    Node t = root.left;
13                    while (t.right != null) {
14                        t = t.right;
15                    }
16                    predecessor = t.data;
17                }
18                if (root.right != null) {
19                    Node t = root.right;
20                    while (t.left != null) {
21                        t = t.left;
22                    }
23                    successor = t.data;
24                }
25            } else if (root.data > val) {
26                successor = root.data;
27                successorPredecessor(root.left, val);
28
29
30            } else if (root.data < val) {
31                predecessor = root.data;
32                successorPredecessor(root.right, val);
33            }
34        }
35    }
```

FINDING MAXIMUM



```
Node max(node head)
{
        if(head == NULL)
                return NULL;
        else if (head->right == NULL)
                return head;
        else
                return max(head->right);
}
```

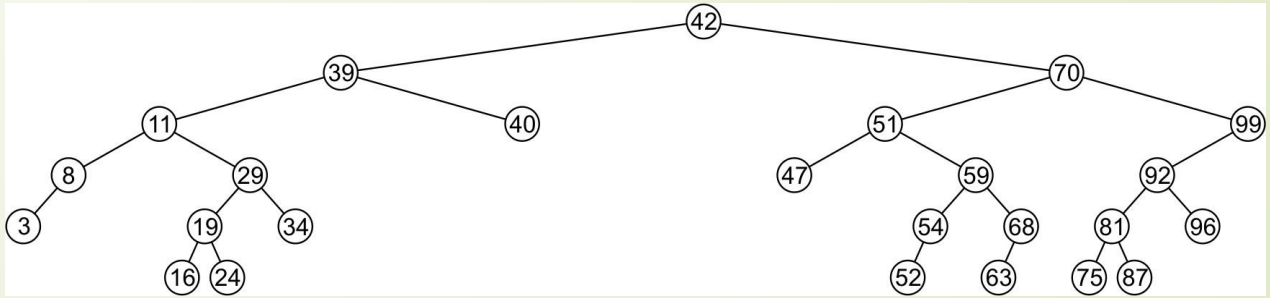
DELETE

Deleting a node from Binary search tree is little complicated compare to inserting a node. It includes two steps:

1. Search the node with given value.
2. Delete the node.

The algorithm has 3 cases while deleting node:

1. Node to be deleted has is a leaf node (no children).
2. Node to be deleted has one child (eight left or right child node).
3. Node to be deleted has two nodes.



Delete

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

Case III

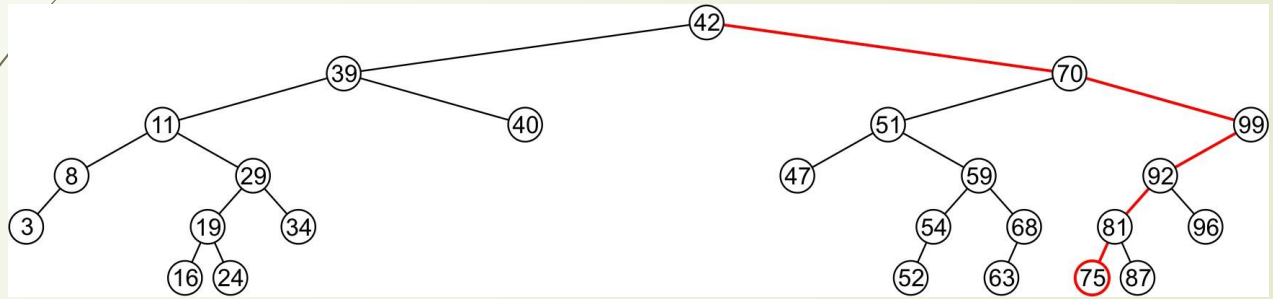
In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.

DELETE

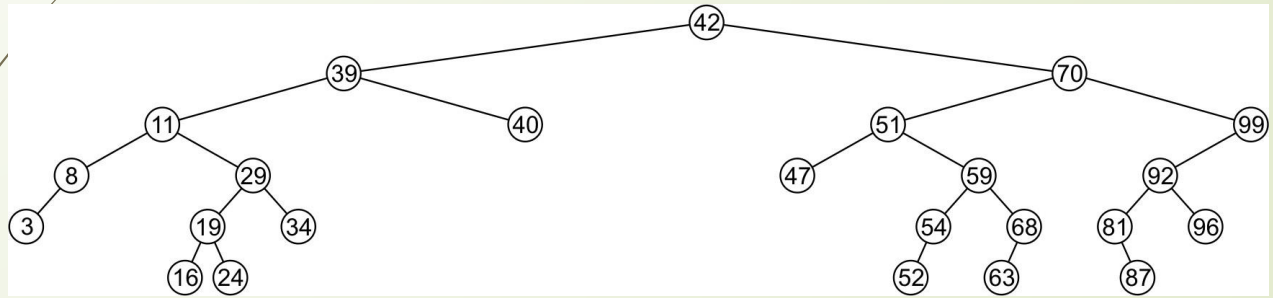
A leaf node simply must be removed and the appropriate member variable of the parent is set to nullptr

- Consider removing 75



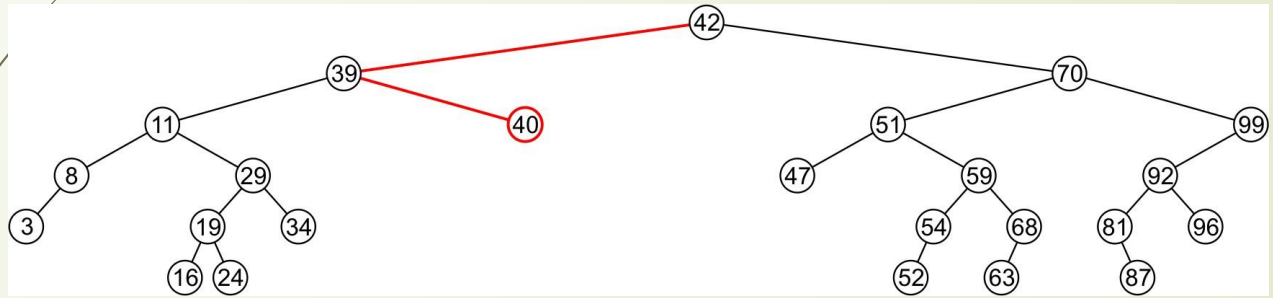
DELETE

The node is deleted and left_tree of 81 is set to nullptr



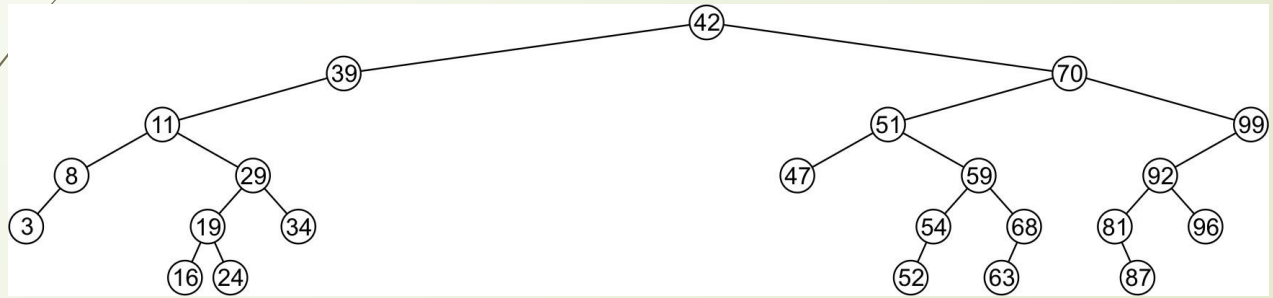
DELETE

Erasing the node containing 40 is similar



DELETE

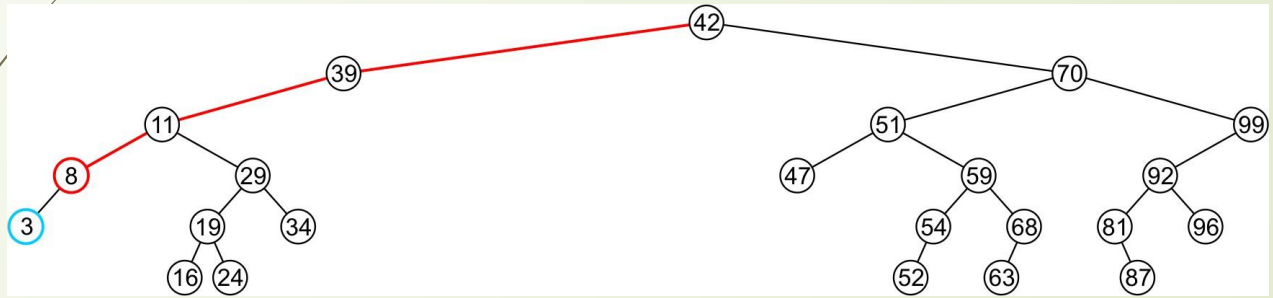
The node is deleted and right_tree of 39 is set to nullptr



DELETE

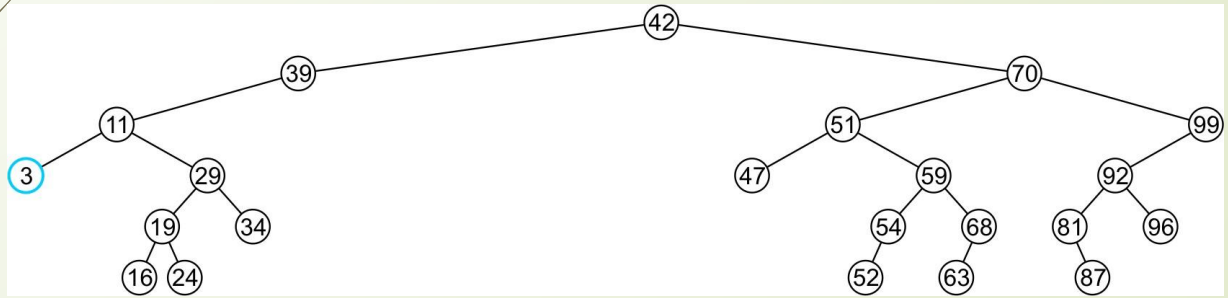
If a node has only one child, we can simply promote the sub-tree associated with the child

- Consider removing 8 which has one left child



DELETE

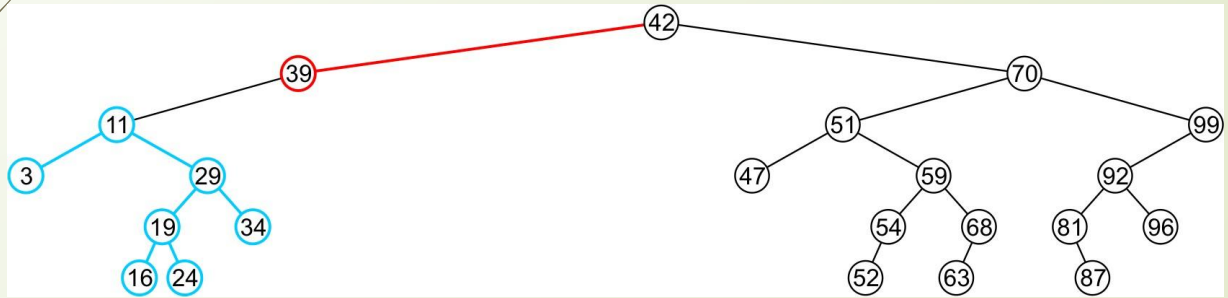
The node 8 is deleted and the left_tree of 11 is updated to point to 3



DELETE

There is no difference in promoting a single node or a sub-tree

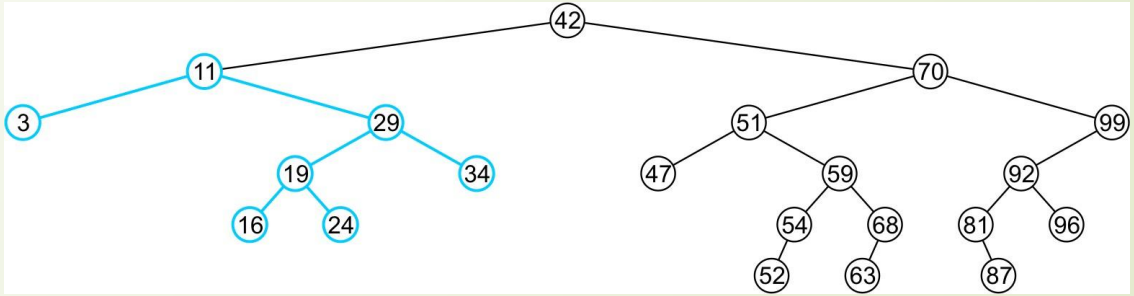
- To remove 39, it has a single child 11



DELETE

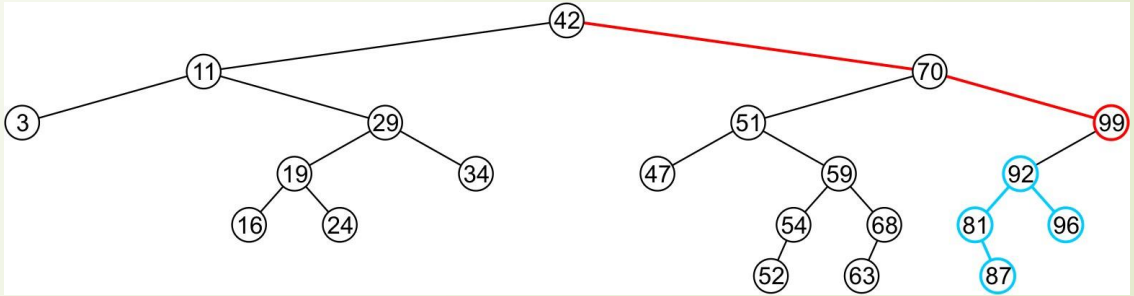
The node containing 39 is deleted and left_node of 42 is updated to point to 11

- Notice that order is still maintained



DELETE

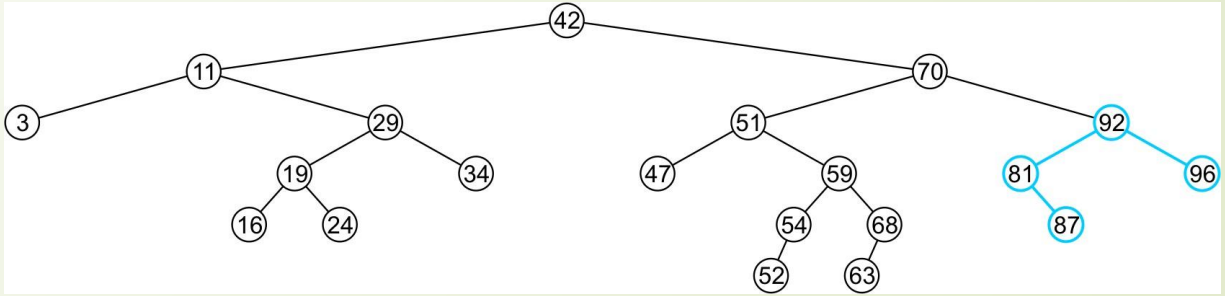
Consider erasing the node containing 99



DELETE

The node is deleted and the left sub-tree is promoted:

- The member variable right_tree of 70 is set to point to 92
- Again, the order of the tree is maintained

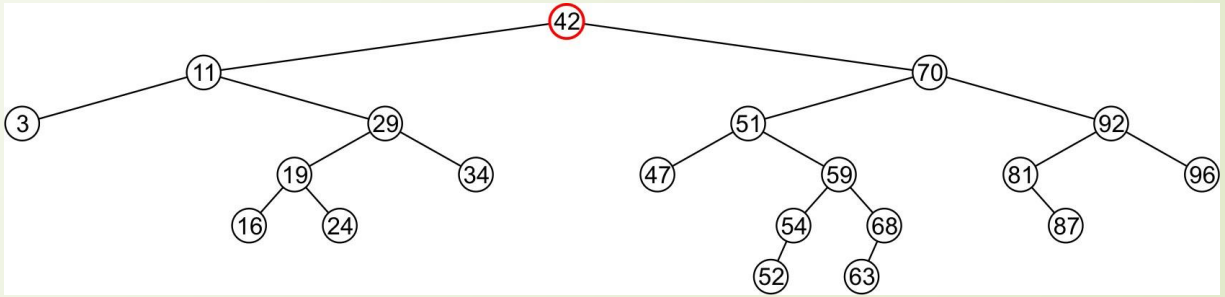


DELETE

Finally, we will consider the problem of erasing a full node, *e.g.*, 42

We will perform two operations:

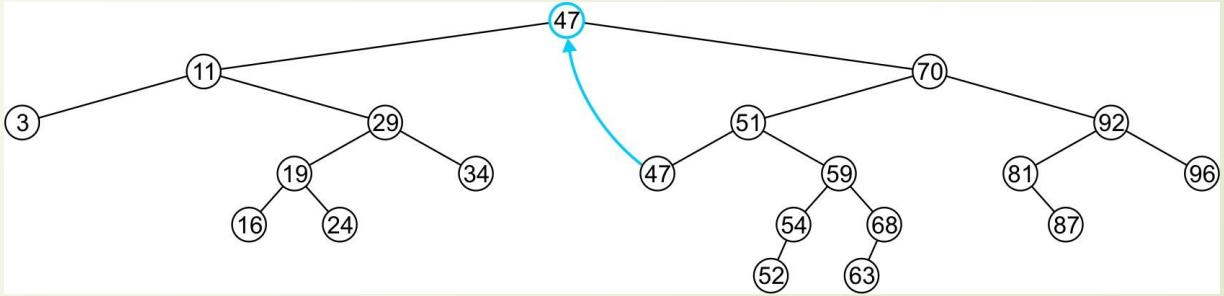
- Replace 42 with the minimum object in the right sub-tree
- Erase that object from the right sub-tree



DELETE

In this case, we replace 42 with 47

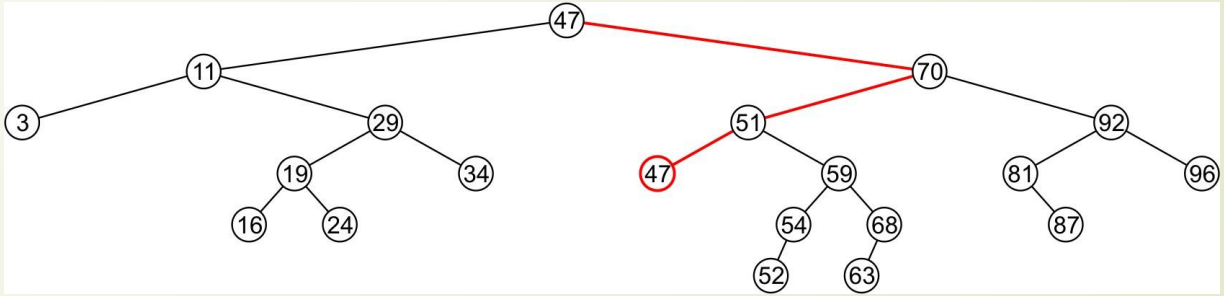
– We temporarily have two copies of 47 in the tree



DELETE

We now recursively erase 47 from the right sub-tree

– We note that 47 is a leaf node in the right sub-tree

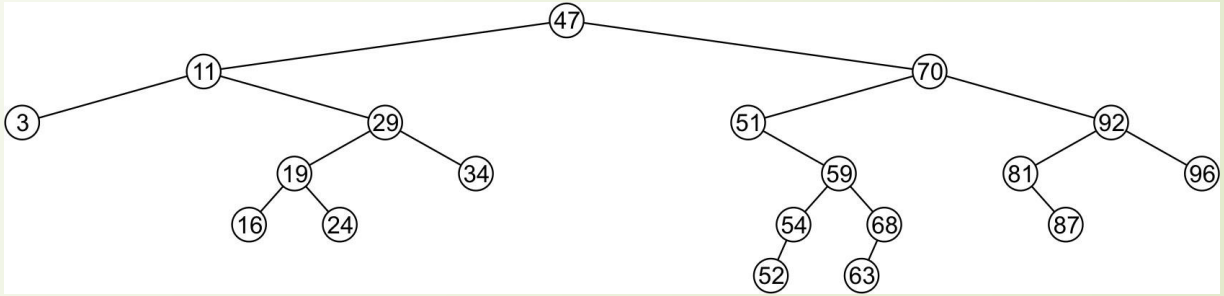


DELETE

Leaf nodes are simply removed and left_tree of 51 is set to nullptr

– Notice that the tree is still sorted:

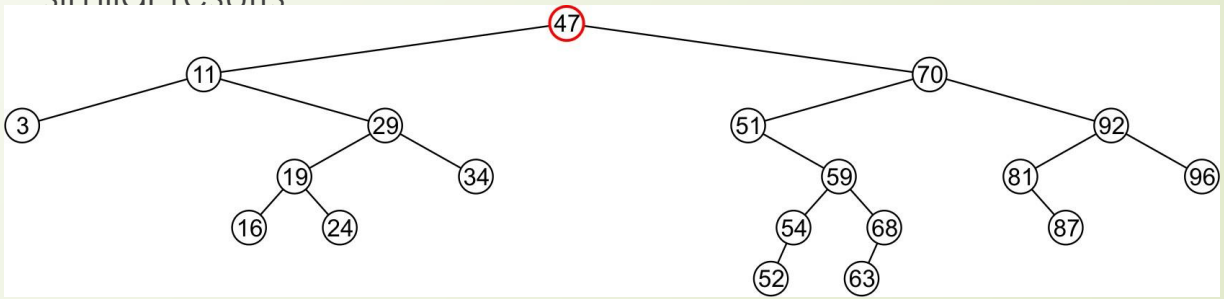
47 was the least object in the right sub-tree



DELETE

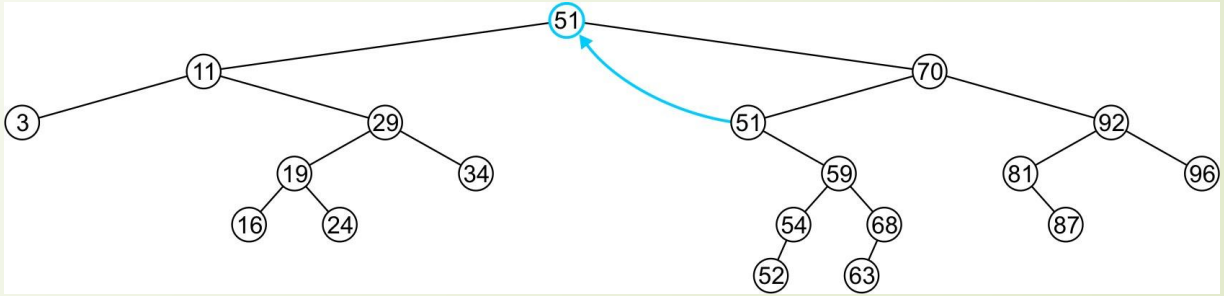
?Suppose we want to erase the root 47 again:

- We must copy the minimum of the right sub-tree
- We could promote the maximum object in the left sub-tree and achieve similar results



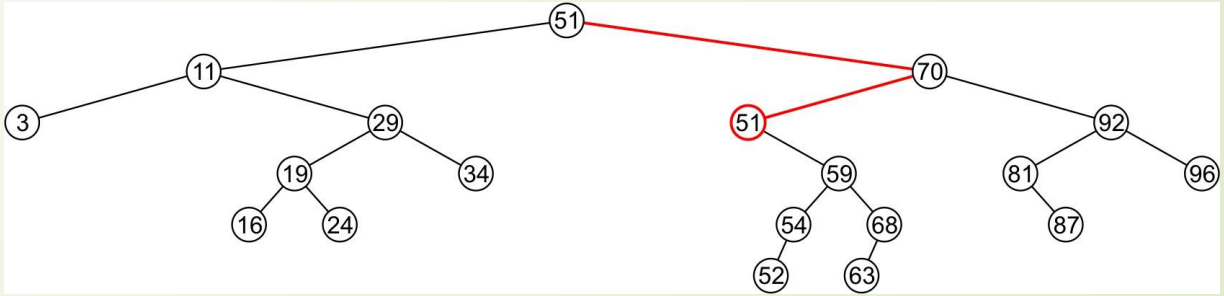
DELETE

We copy 51 from the right sub-tree



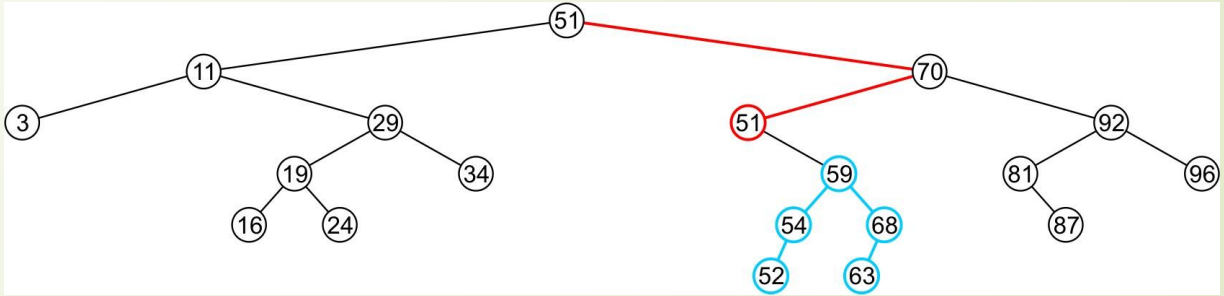
DELETE

We must proceed by delete 51 from the right sub-tree



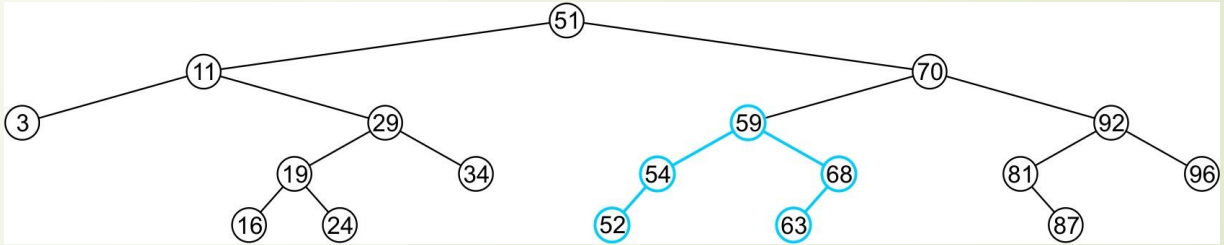
DELETE

In this case, the node storing 51 has just a single child



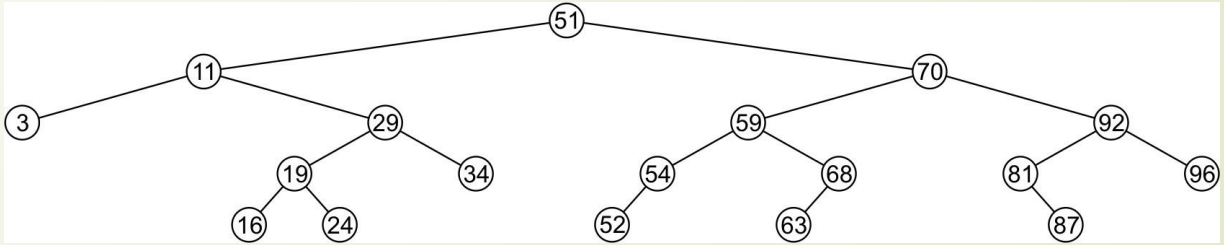
DELETE

We delete the node containing 51 and assign the member variable left_tree of 70 to point to 59



DELETE

Note that after seven removals, the remaining tree is still correctly sorted



Implementation

```
void deleteKey(int key) { root = deleteRec(root, key); }
        Node deleteRec(Node root, int key)
        {   /* Base Case: If the tree is empty */
                if (root == null)
                        return root;
                /* Otherwise, recur down the tree */
                if (key < root.key)
                        root.left = deleteRec(root.left, key);
                else if (key > root.key)
                        root.right = deleteRec(root.right, key);
                // if key is same as root's // key, then This is the // node to be deleted
                else {
                        if (root.left == null)
                                return root.right;
                        else if (root.right == null)
                                return root.left;
                        // node with two children: Get the inorder
                        // successor (smallest in the right subtree)
                        root.key = minValue(root.right);

                        // Delete the inorder successor
                        root.right = deleteRec(root.right, root.key);       }
                return root;
        }
```

Thinking

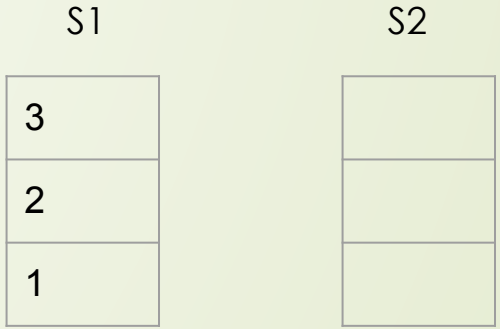
Implementing Queue using stack?

Implementing stacks using queue?

Stacks with Queue

Using two stacks implement QUEUE functionality

```
Class Queue{
Stack s1;
Stack s2;
Void push(int x){
        s1.push(s);
}
int Qpop(){
while(!s1. empty()){
        S2.push(s1.pop());
}
Int ans= s2.pop();// deleted
while(!s2. empty()){
        S1.push(s2.pop());
}
Return ans;
}
}
```



Binary Search Tree Complexities

Time Complexity

Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

TRAVERSAL IN A BST

There are two types of traversals

- Breadth First Search Traversal (BFS)

- Depth First Search Traversal (DFS)

  Make use of Stack ADT

  - Inorder
  - Preorder
  - Postorder