

Week 14

Binary Search Tree

Data Structures CS218
Instructor: Anam Qureshi

Outline

This topic covers binary search trees:

- Background
- Definition and examples
- Implementation details of BST operations

Background

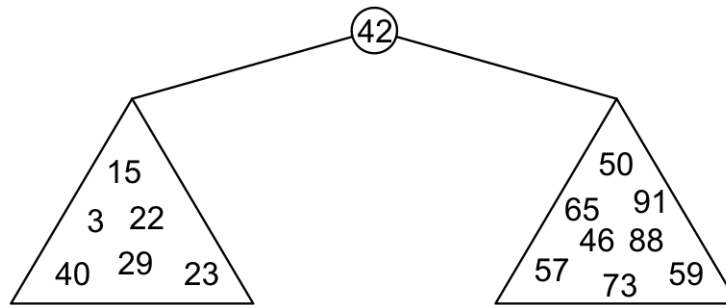
Recall that with a binary tree, we can dictate an order on the two children

We will exploit this order:

- Require all objects in the left sub-tree to be less than the object stored in the root node, and
- Require all objects in the right sub-tree to be greater than the object in the root object

Binary Search Trees

Graphically, we may relationship



- Each of the two sub-trees will themselves be binary search trees

Binary Search Trees

Notice that we have already used this structure for searching: examine the root node and if we have not found what we are looking for:

- If the object is less than what is stored in the root node, continue searching in the left sub-tree
- Otherwise, continue searching the right sub-tree

With a linear order, one of the following three must be true:

$$a < b \quad a = b \quad a > b$$

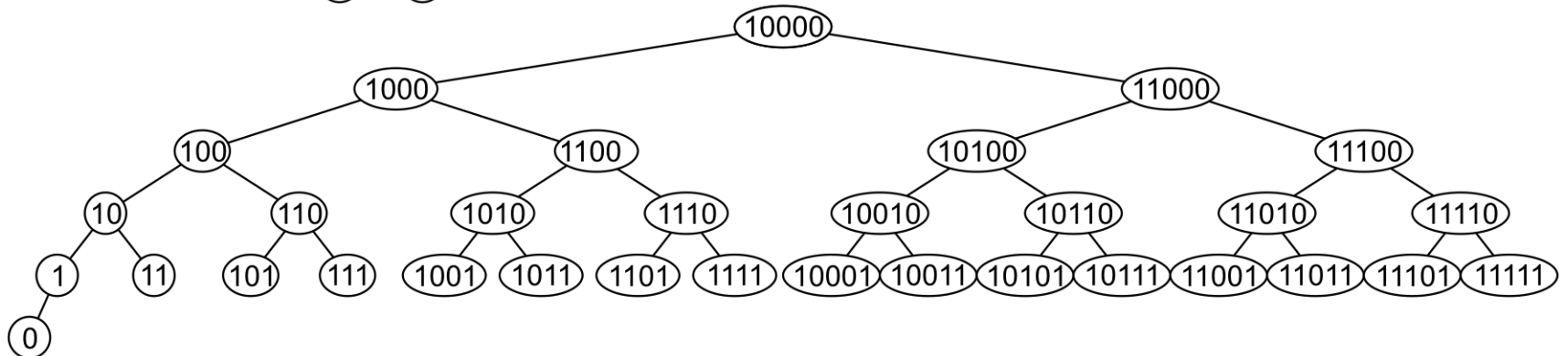
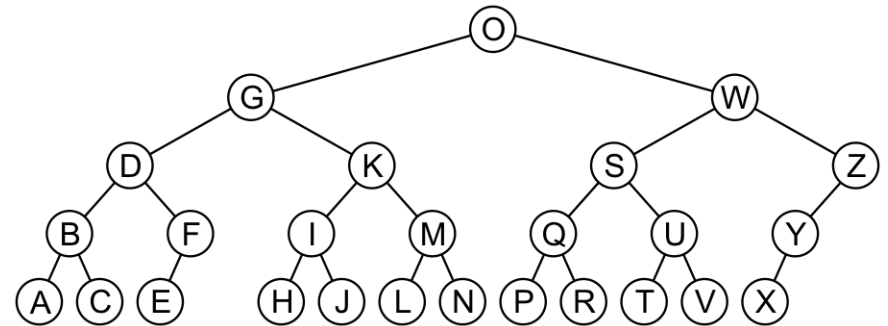
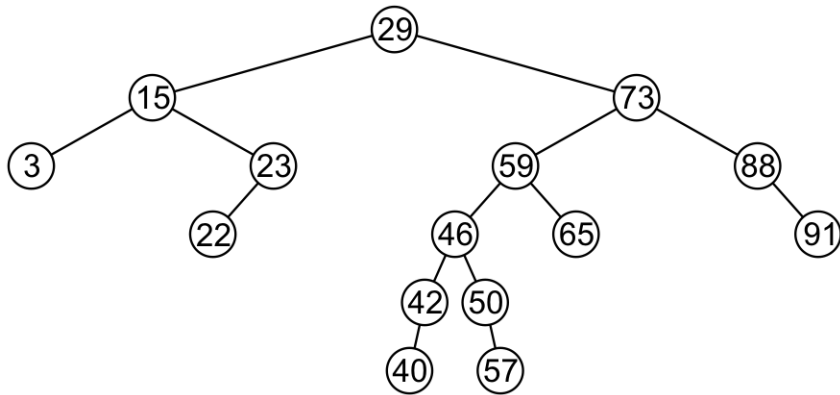
Definition

Thus, we define a non-empty binary search tree as a binary tree with the following properties:

- The left sub-tree (if any) is a binary search tree and all values are less than the root value, and
- The right sub-tree (if any) is a binary search tree and all values are greater than the root value

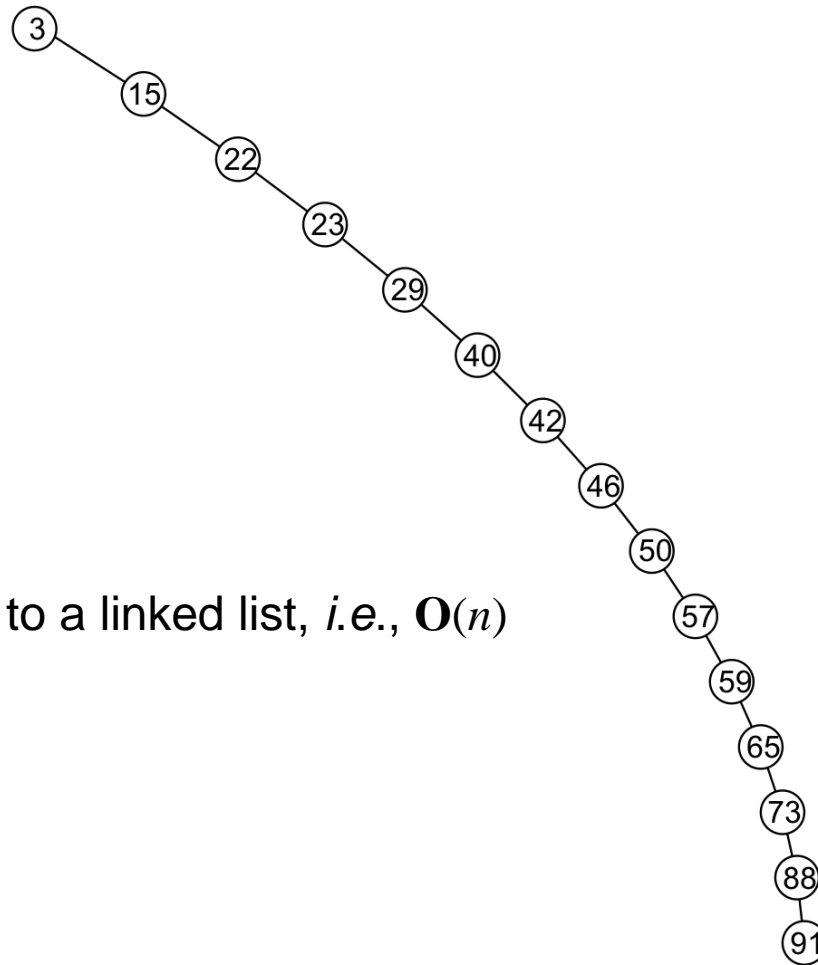
Examples

Here are other examples of binary search trees:



Examples

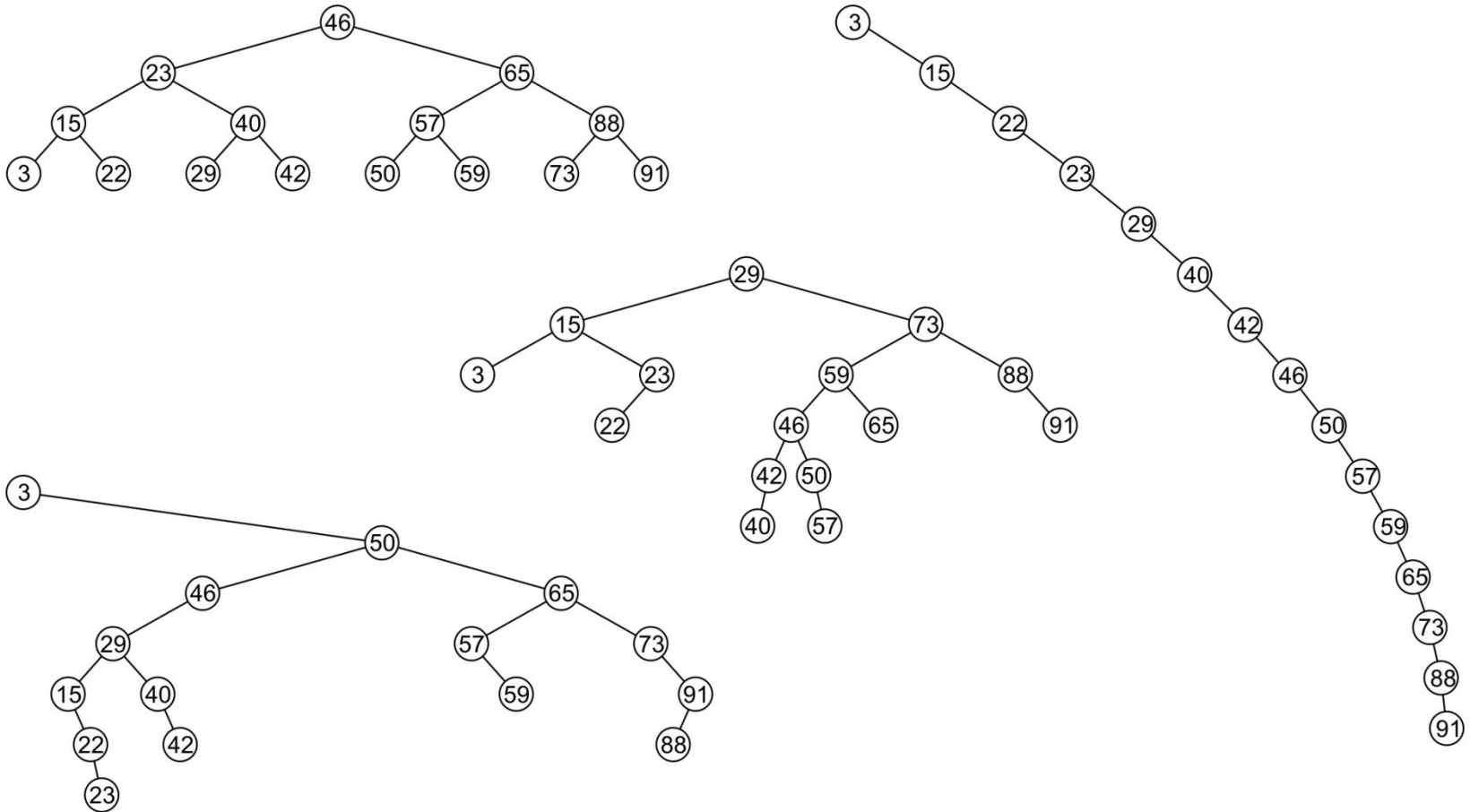
Unfortunately, it is possible to construct *degenerate* binary search trees



- This is equivalent to a linked list, *i.e.*, $O(n)$

Examples

All these binary search trees store the same data



Structure of BST Node

```
Struct node{  
    int data;  
    node * left;  
    node *right;  
};
```



Operations on a BST

- Insert
- Update
- Search
- Delete
- Traversal
 - BFT
 - DFS
 - In-order, post-order, and pre-order
- Finding minimum
- Finding maximum
- Finding predecessor
- Finding Successor
- Finding height

Insert

- Let's insert any random data into a BST
(Discussion)

Implementation Details

```
Node* insert (int data, node *head)
{
    if(head == NULL)
    {
        head= new node;
        head->data= data;
        head->left= NULL;
        head->right= NULL;
    }
    else if (data < head->data)
        head->left= insert(data, head->left);
    else if(data > head->data)
        head->right= insert(data, head->right);
    return head;
}
```

Search

```
Node* search(node* head, int data){  
    if(head == NULL)  
        return NULL;  
    else if( data < head->data)  
        return search(head->left, data);  
    else if (data > head->data)  
        return search(head->right, data);  
    else  
        return head;  
}
```

Finding minimum

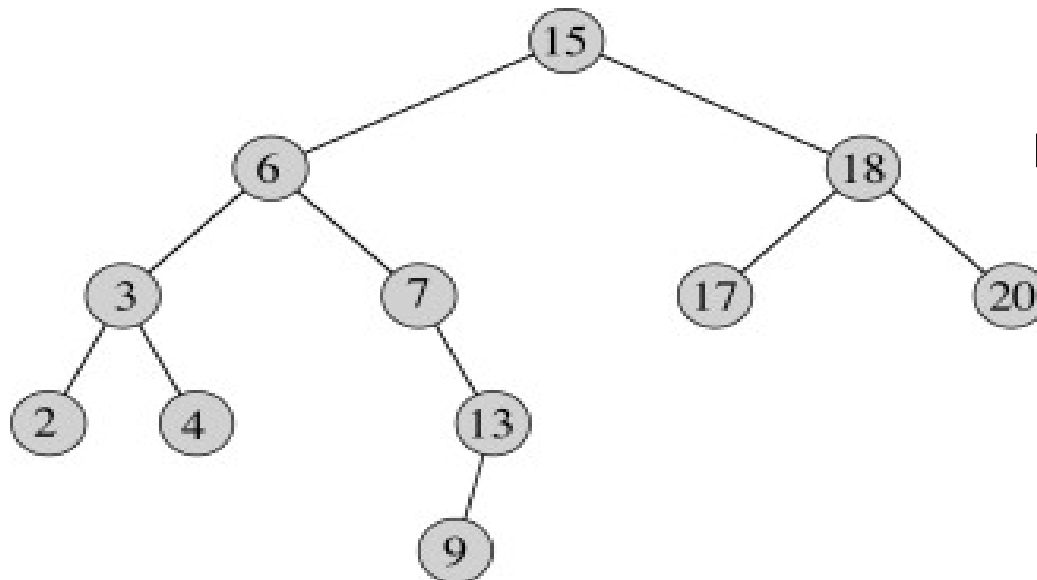
```
Node* min(node *head)
{
    if(head == NULL)
        return NULL;
    else if (head->left == NULL)
        return head;
    else
        return min(head->left);
}
```

Finding Maximum

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


Finding Predecessor

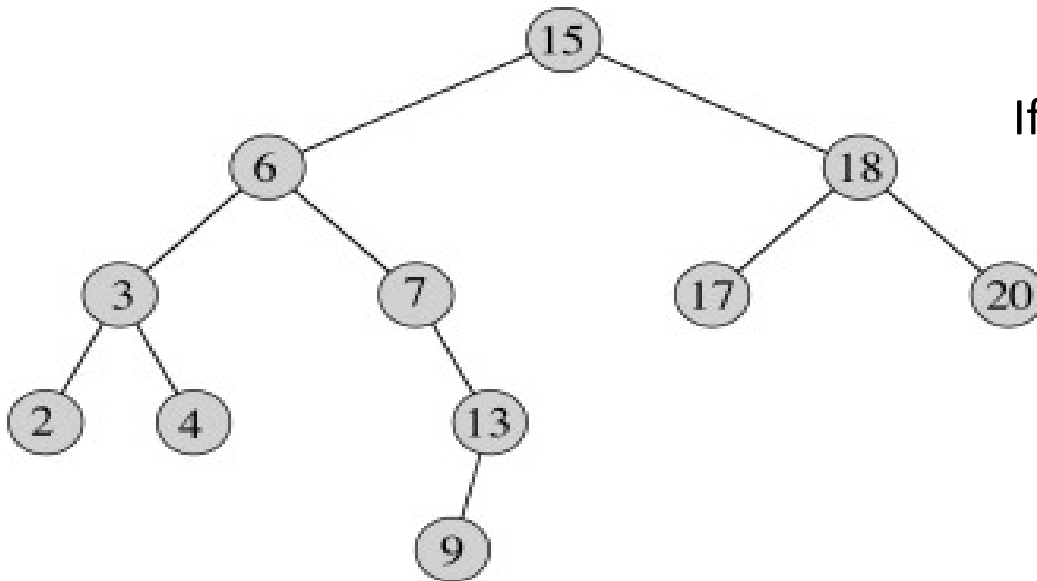
- A predecessor is a maximum value in a left sub tree
- A predecessor of a node is a right most element in a left sub tree.



If(head->left!=NULL)
return max(head->left)

Finding Successor

- A successor is a minimum value in a right sub tree
- A successor of a node is a left most element in a right sub tree.



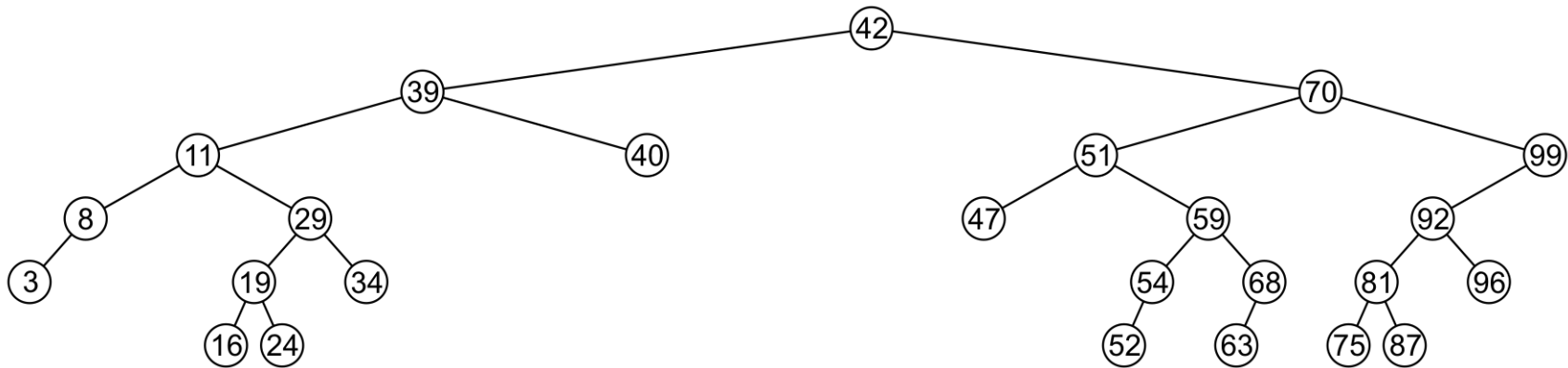
If(head->right!=NULL)
return min(head->right)

Delete

A node being erased is not always going to be a leaf node

There are three possible scenarios:

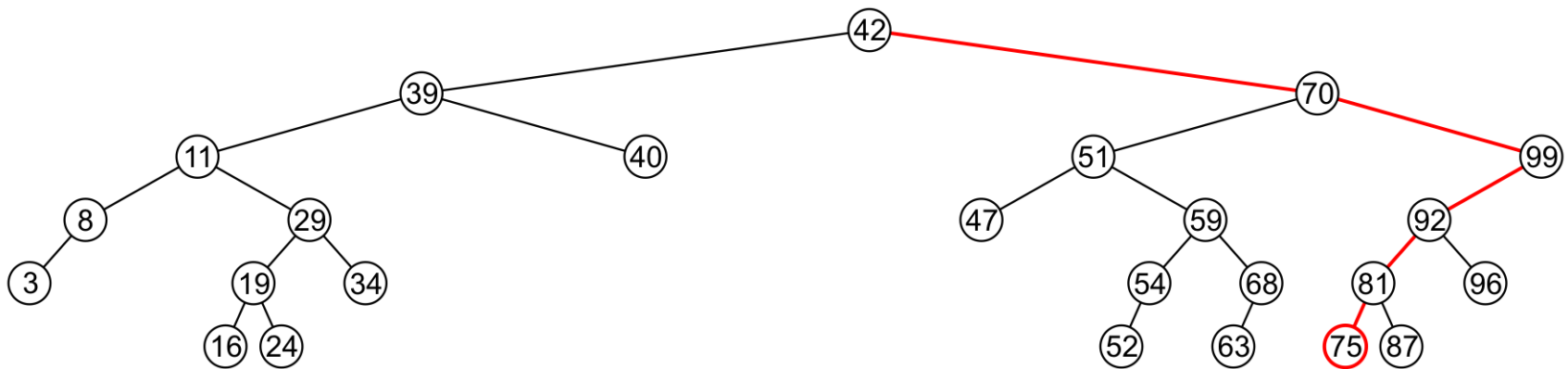
- The node is a leaf node,
- It has exactly one child, or
- It has two children (it is a full node)



Delete

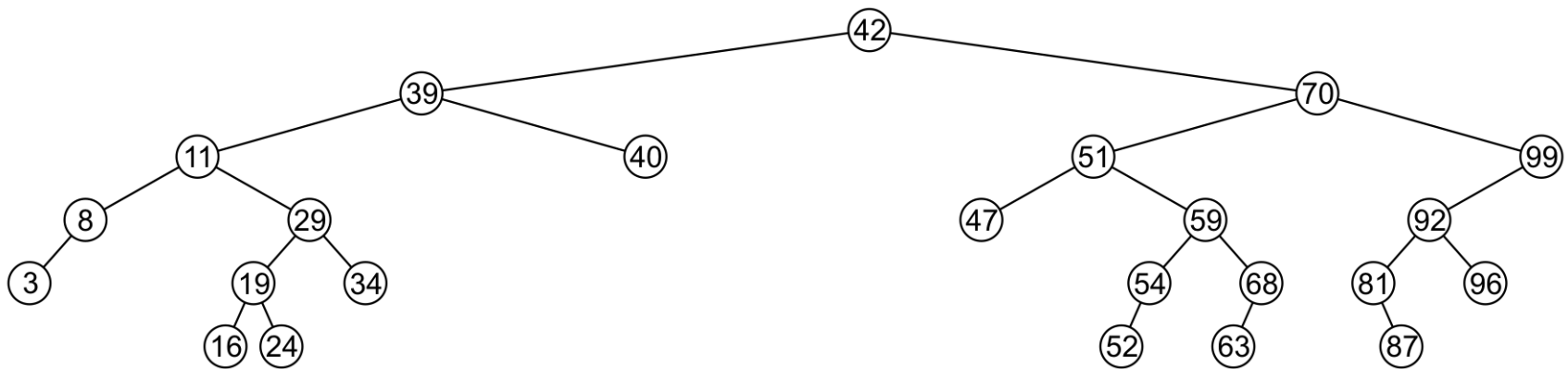
A leaf node simply must be removed and the appropriate member variable of the parent is set to nullptr

- Consider removing 75



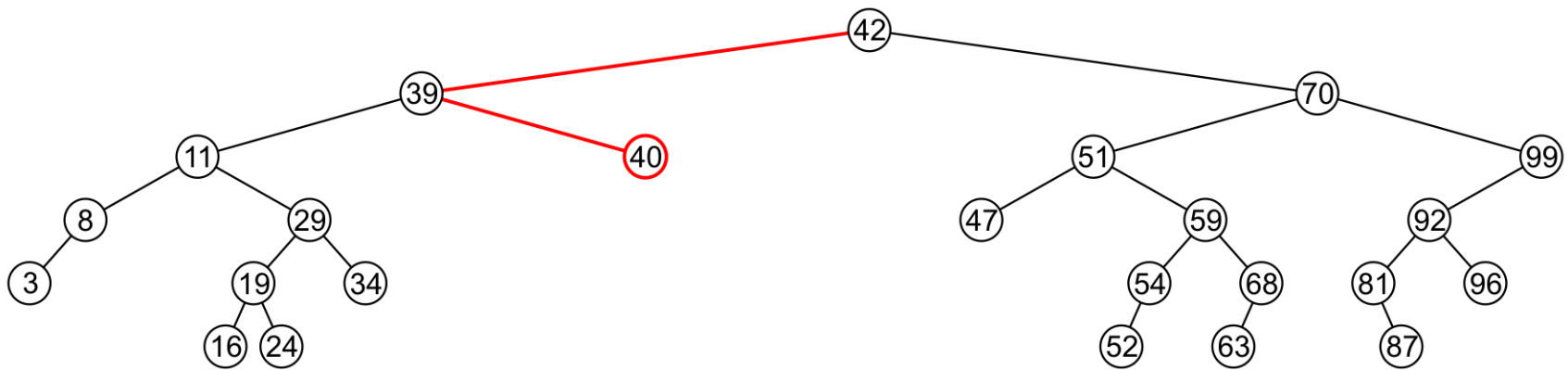
Delete

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



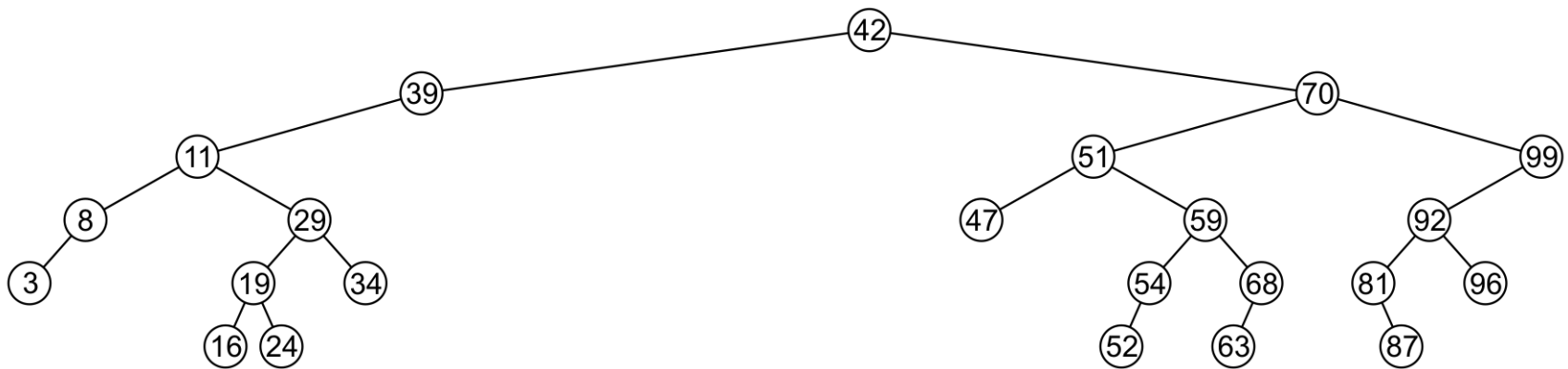
Delete

Erasing the node containing 40 is similar



Delete

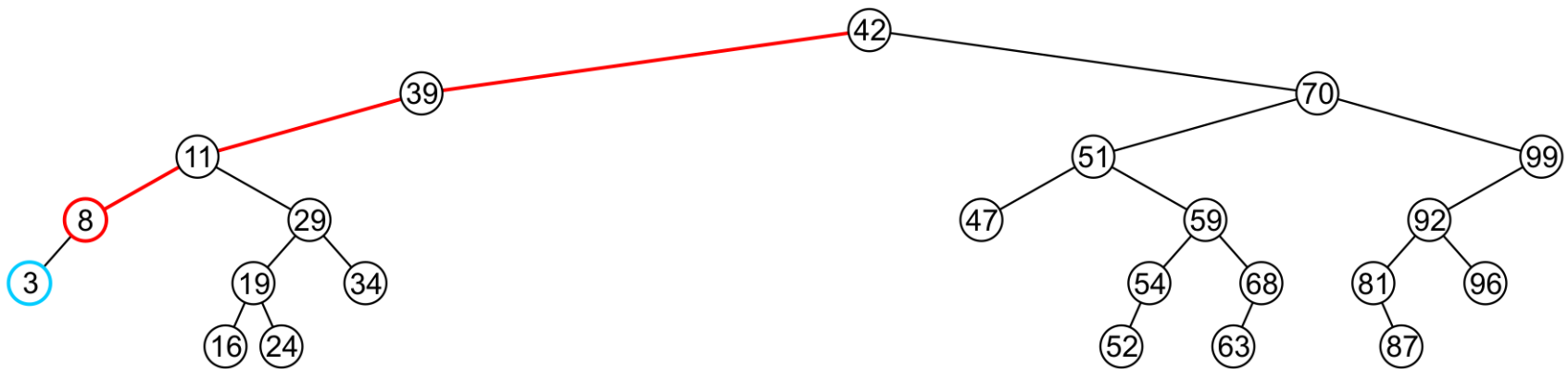
The node is deleted and `right_tree` of 39 is set to `nullptr`



Delete

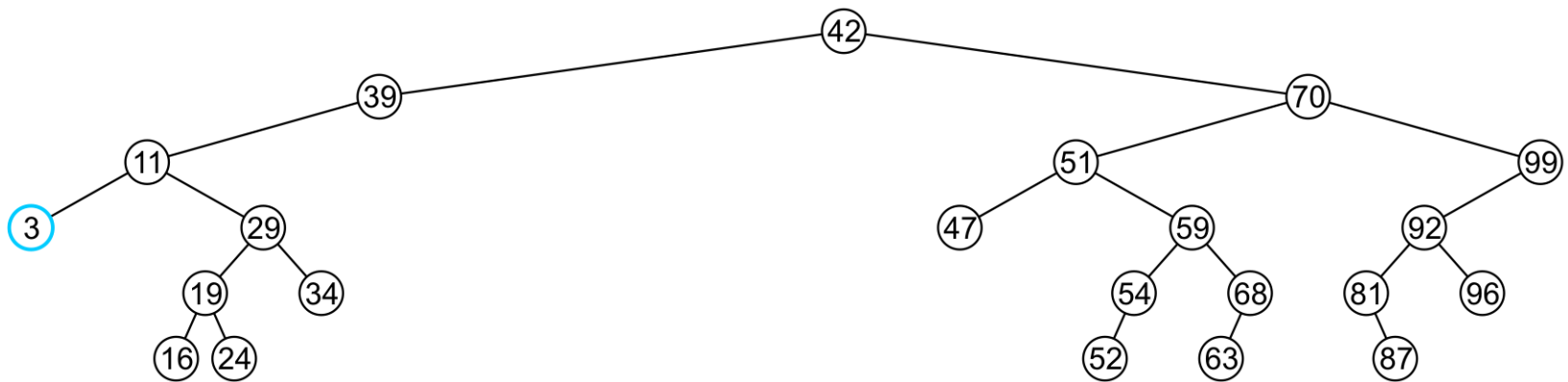
If a node has only one child, we can simply promote the sub-tree associated with the child

- Consider removing 8 which has one left child



Delete

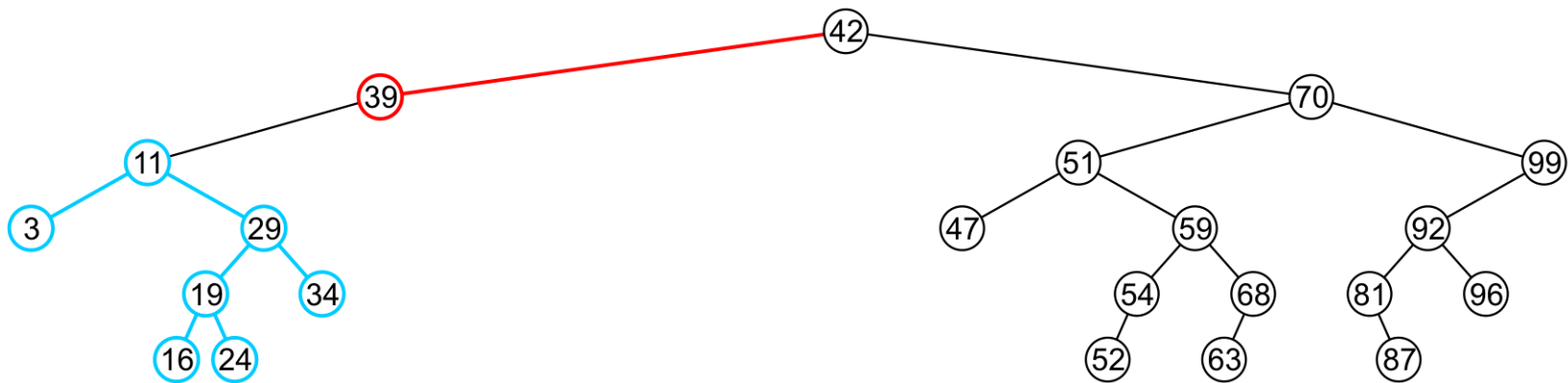
The node 8 is deleted and the left_tree of 11 is updated to point to 3



Delete

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

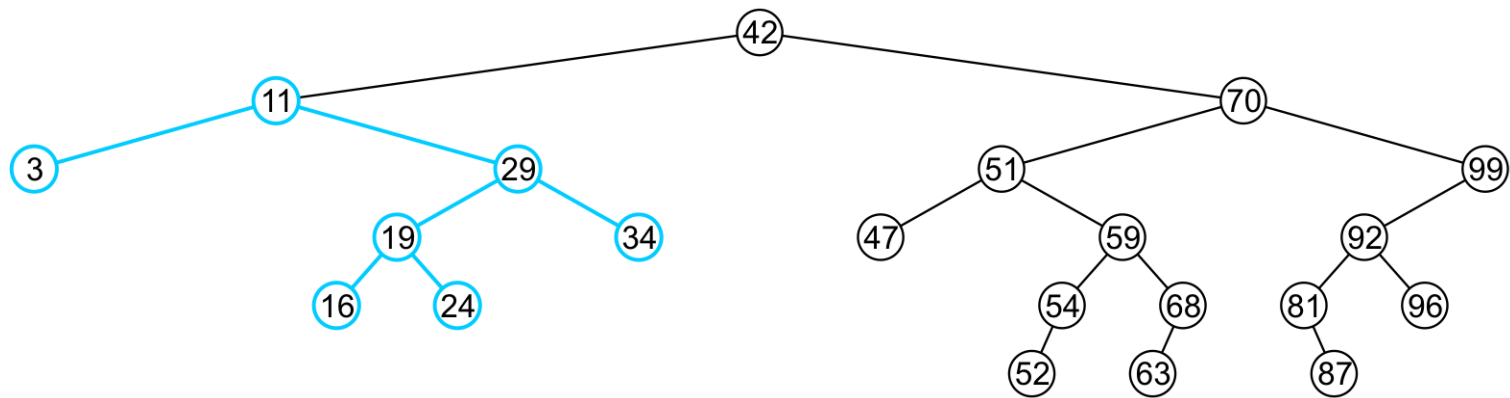
- To remove 39, it has a single child 11



Delete

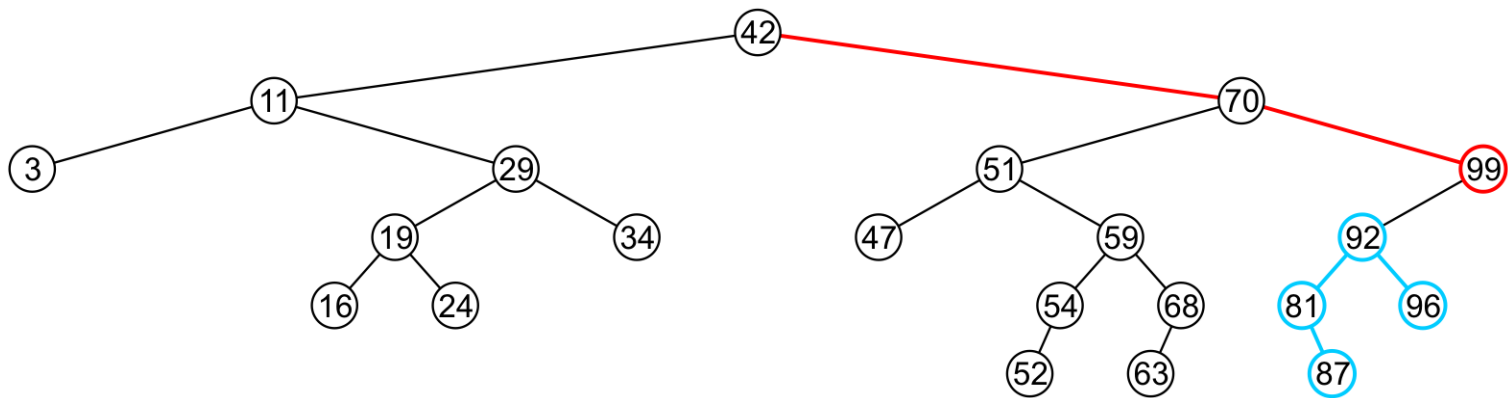
The node containing 39 is deleted and left_node of 42 is updated to point to 11

- Notice that order is still maintained



Delete

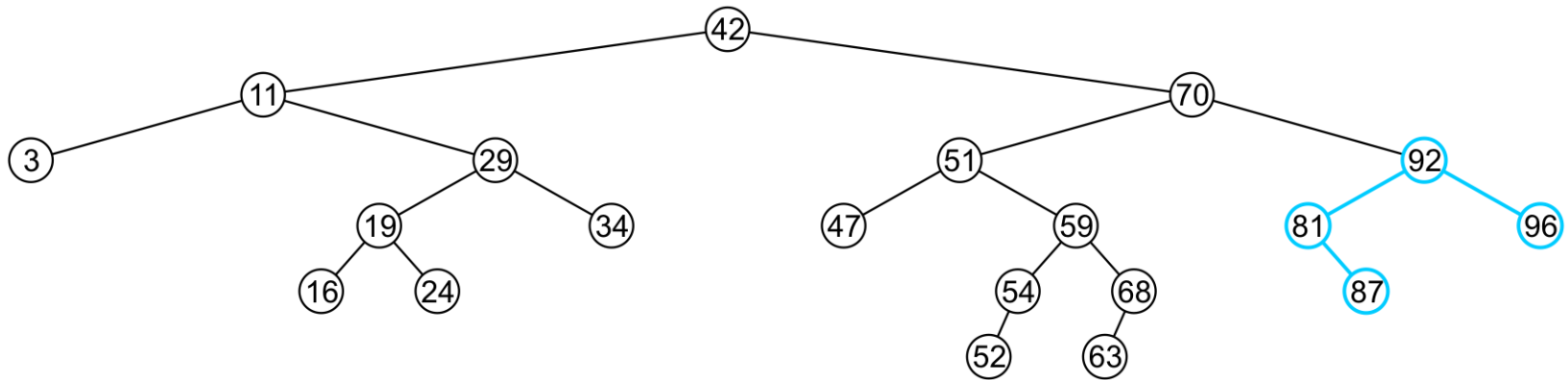
Consider erasing the node containing 99



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

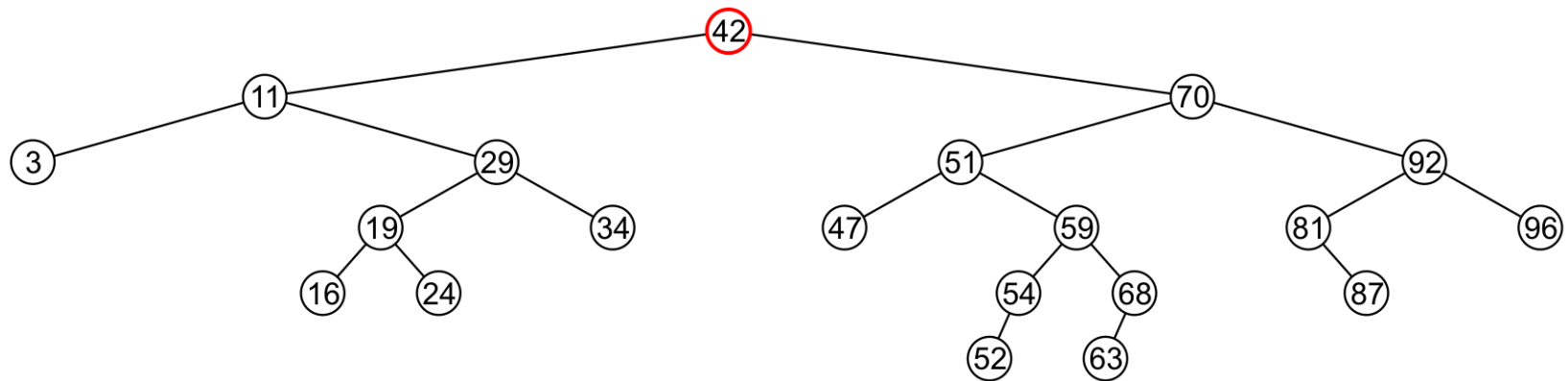


Delete

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

We will perform two operations:

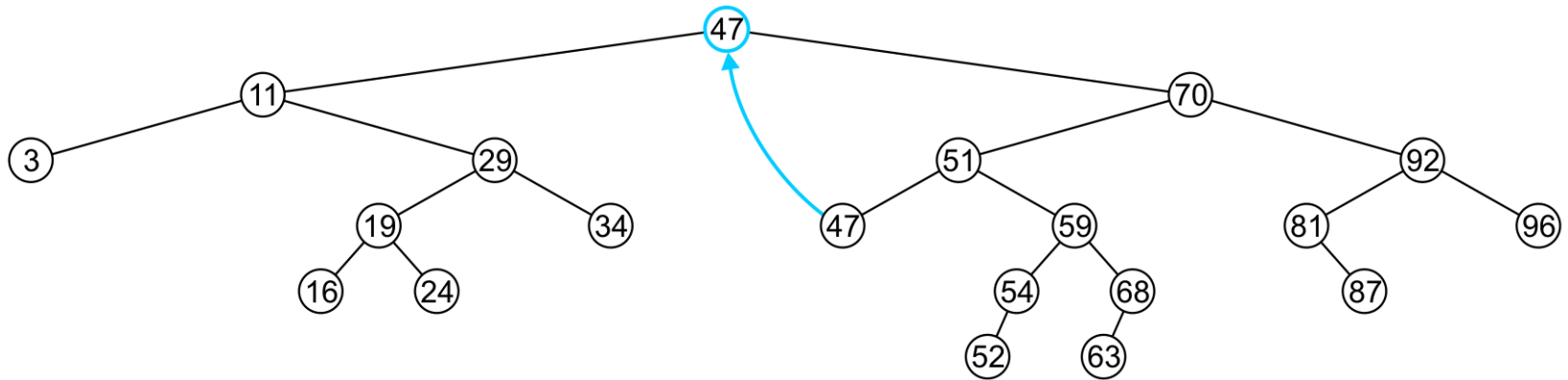
- Replace 42 with the minimum object in the right sub-tree
- Erase that object from the right sub-tree



Delete

In this case, we replace 42 with 47

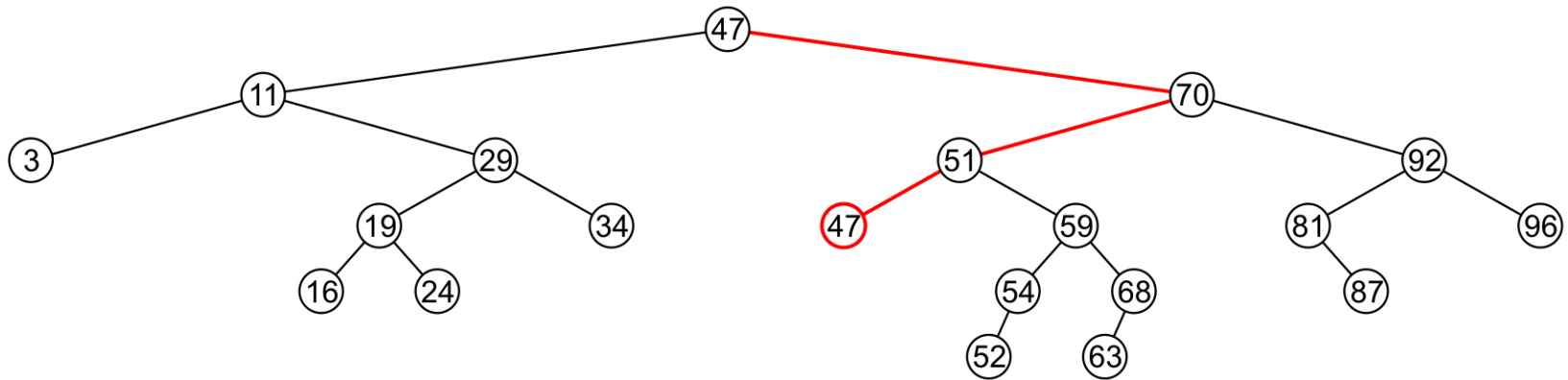
- We temporarily have two copies of 47 in the tree



Delete

We now recursively erase 47 from the right sub-tree

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

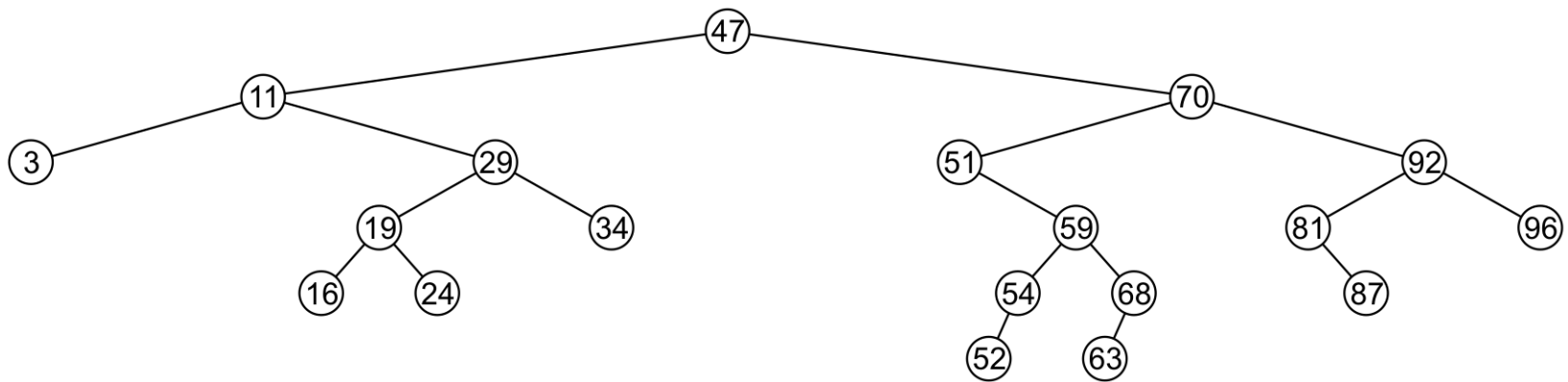


Delete

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

- Notice that the tree is still sorted:

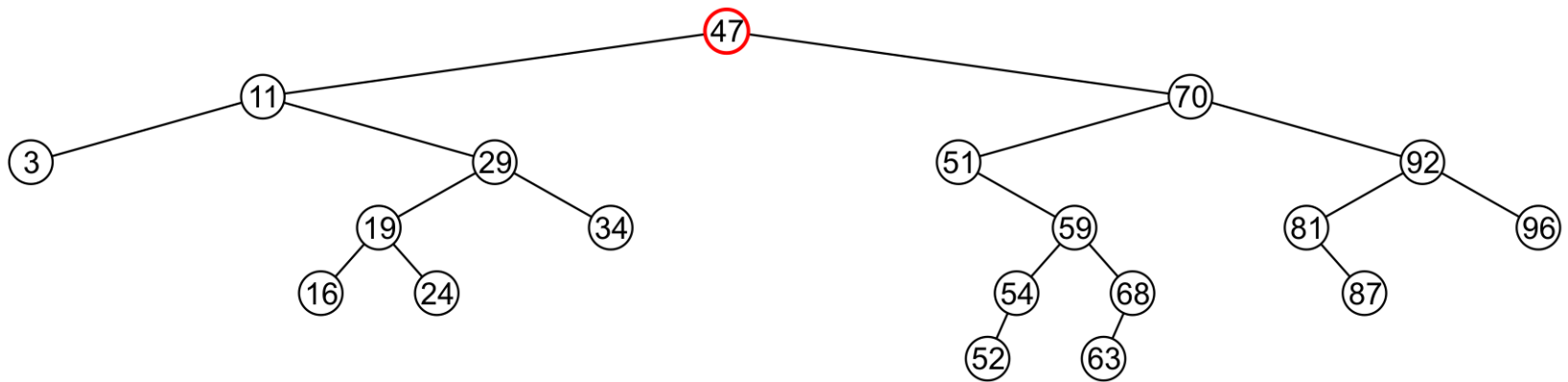
47 was the least object in the right sub-tree



Delete

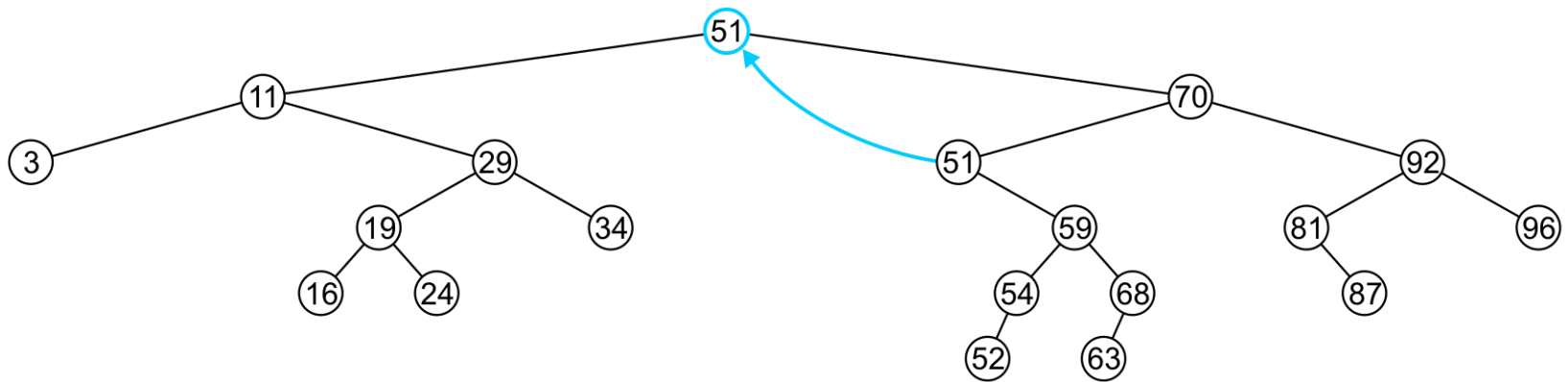
Suppose we want to erase the root 47 again:

- 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



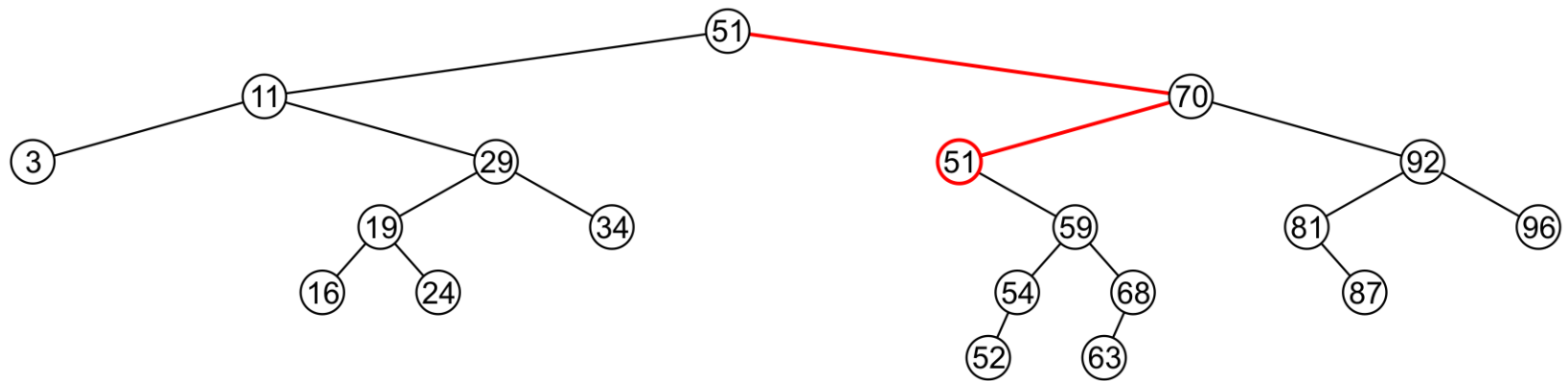
Delete

We copy 51 from the right sub-tree



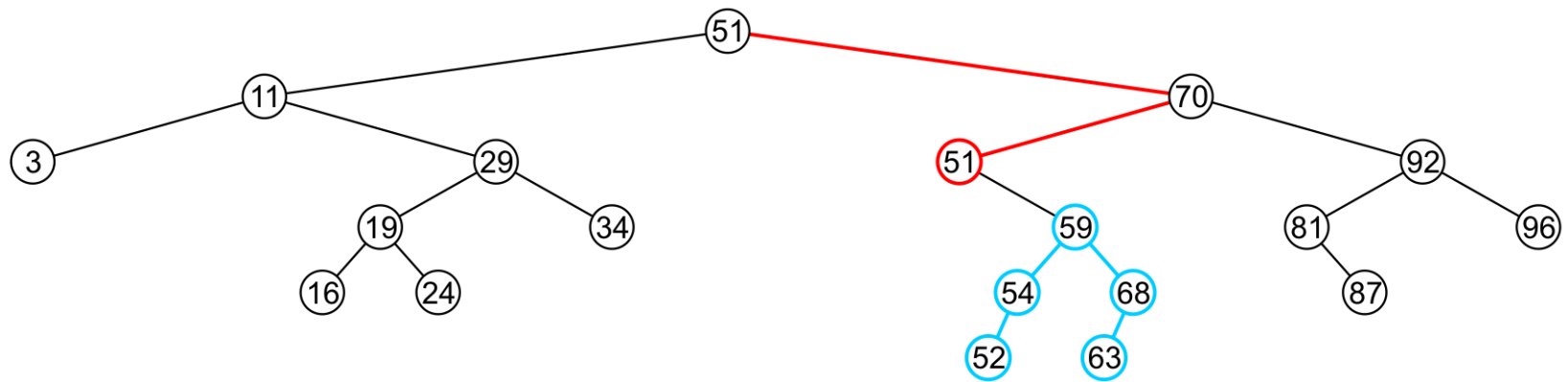
Delete

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



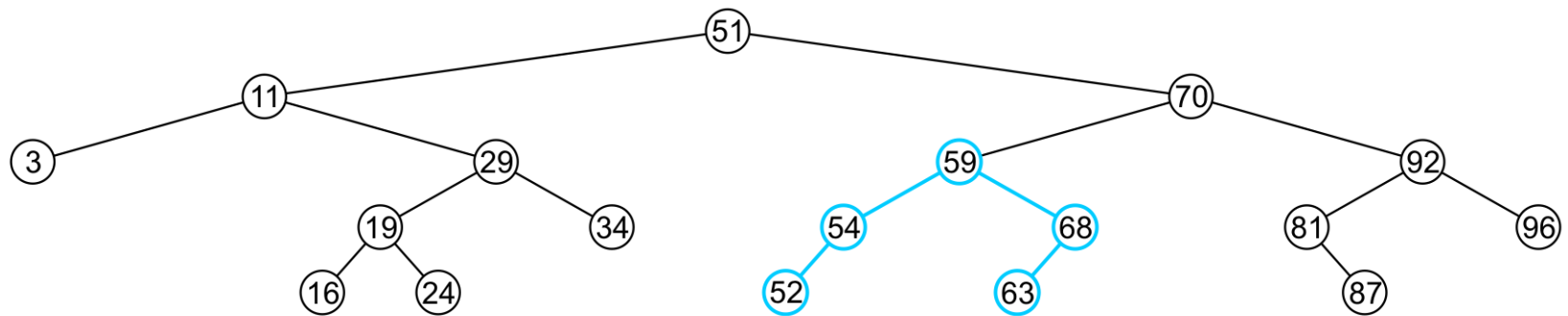
Delete

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



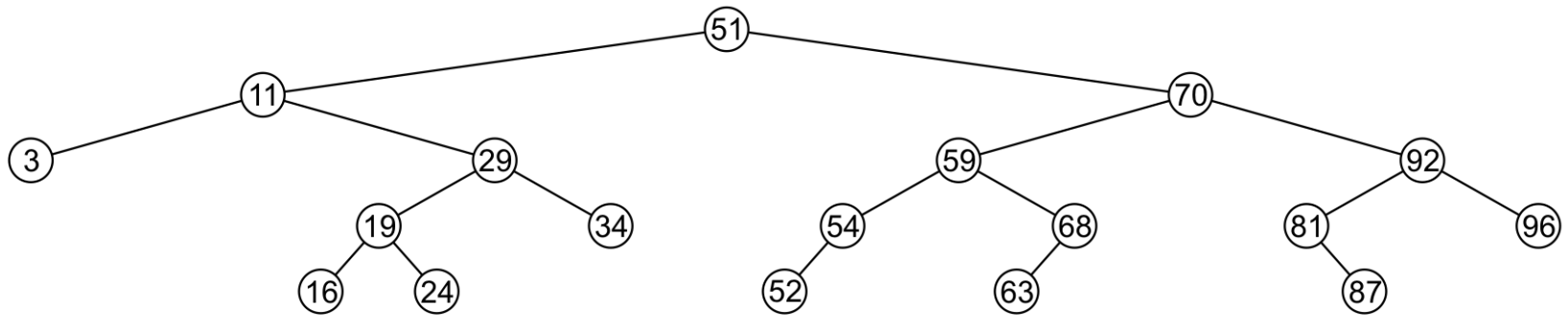
Delete

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 (Delete)

```
Node * delete(int x, node* head)
{
    node *temp;
    if( head == NULL)
        return NULL;
    else if (data < head->data) //left leaf node
        head->left = delete (data, head->left);
    else if (data > head->data) //left right node
        head->right = delete (data, head->right);
    else if( head->left && head->right ) //full node
    {
        temp= min(head->right);
        head->data= temp->data;
        head->right= delete (head->data, head->right);
    }
}
```


Implementation (Delete) cont..

```
else
{
temp=head;
if(head->left == NULL) //neither node
    head=head->right;
else if (head->right ==NULL) //neither node
    head=head->left;
delete temp;
}
return head;
```

```
}
```

Traversal in a BST

- There are two types of traversals
 - Breadth First Search Traversal (BFS)
 - Make use of Queue ADT (Already covered)
 - Depth First Search Traversal (DFS)
 - Make use of Stack ADT
 - Inorder
 - Preorder
 - Postorder

In-order Traversal

- It is a type of depth first search traversal.
- Following is the traversing order
 - Left node, root node, right node
- Example discussion

```
Void inorder (node *head)
{
    if (head == NULL)
        return;
    inorder (head->left);
    cout << head-data << " ";
    inorder (head->right);
}
```

Pre-order Traversal

- It is a type of depth first search traversal.
- Following is the traversing order
 - root node, left node, right node
- Example discussion

```
Void preorder (node *head)
{
    if (head == NULL)
        return;
    cout << head->data << " ";
    preorder (head->left);
    preorder(head->right);
}
```

Post-order Traversal

- It is a type of depth first search traversal.
- Following is the traversing order
 - Left node, right node, root node
- Example discussion

```
Void postorder (node *head)
{
    if (head == NULL)
        return;
    postorder (head->left);
    postorder(head->right);
    cout << head-data << " ";
}
```

Finding Height of a BST

```
Int height (node * head)
{
    if(head == NULL)
        return NULL;
    else
    {
        int h_left= height(head->left);
        int h_right = height (head ->right);
        if (h_left > h_right)
            return (h_left + 1);
        else return ( h_right + 1);
    }
}
```

Summary

In this topic, we covered binary search trees

- Described Abstract Sorted Lists
- Problems using arrays and linked lists
- Definition of a binary search tree
- Looked at the implementation of:
 - Insert
 - Delete
 - Search
 - Traversals
 - Height