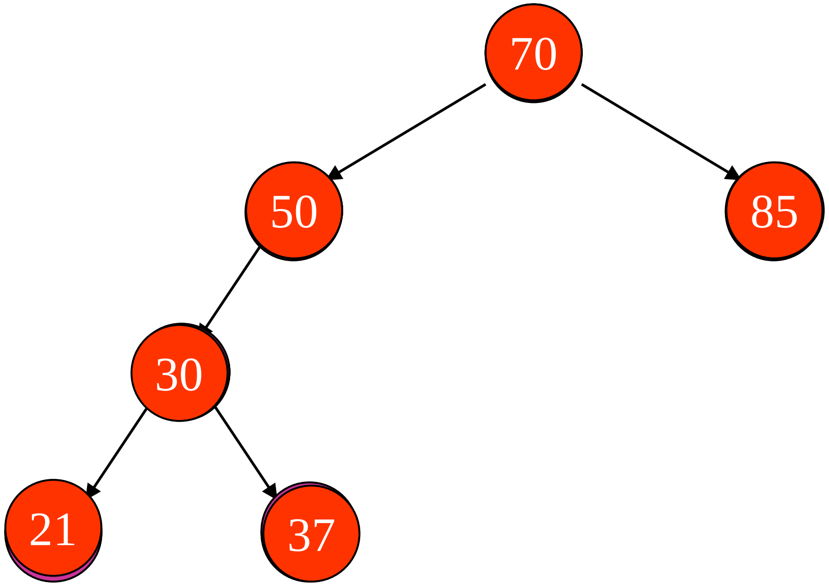
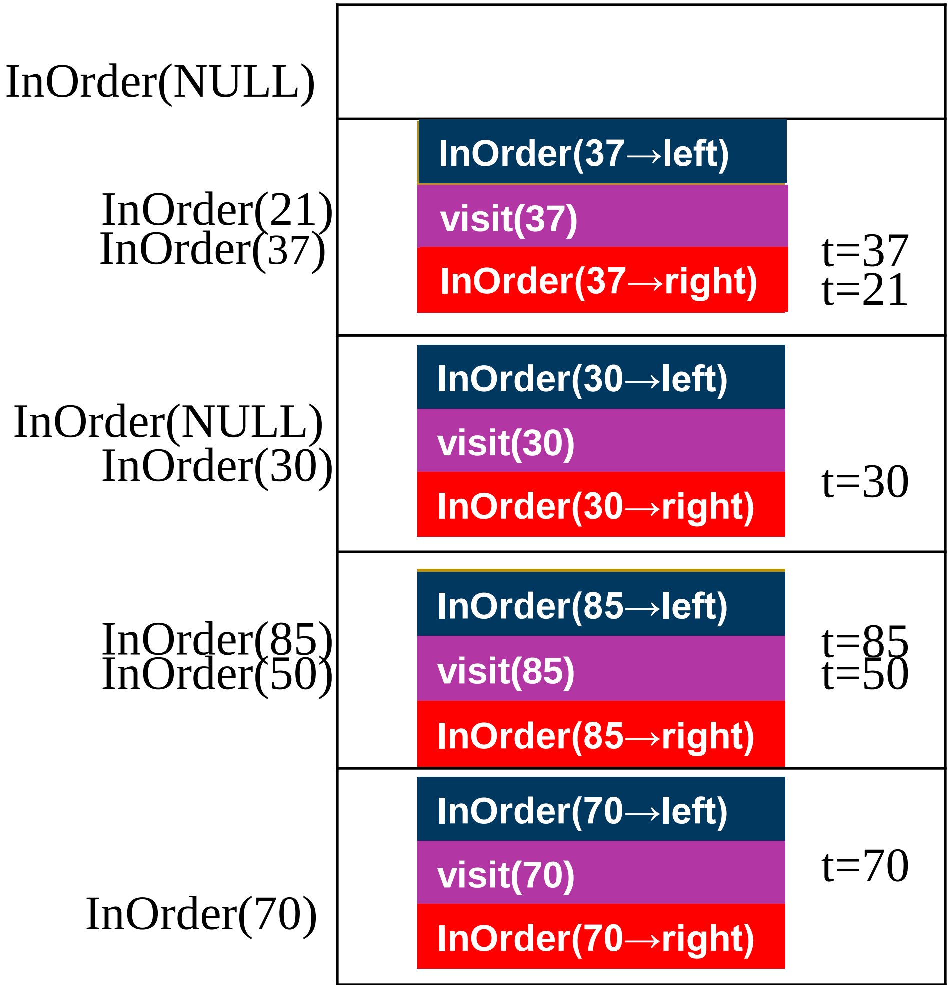
Binary Search Tree Traversal

In Order Traversal (LNR)

```
if (t) {
        InOrder(t→left);
        visit(t);
        InOrder(t→right);
}
```





InOrder(NULL)

InOrder(21)
InOrder(37)

InOrder(37→left)
visit(37)
InOrder(37→right)

t=37
t=21

InOrder(NULL)
InOrder(30)

InOrder(30→left)
visit(30)
InOrder(30→right)

t=30

InOrder(85)
InOrder(50)

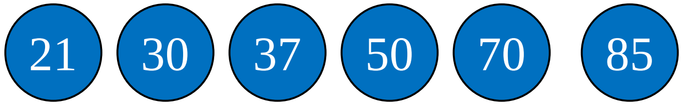
InOrder(85→left)
visit(85)
InOrder(85→right)

t=85
t=50

InOrder(70)

InOrder(70→left)
visit(70)
InOrder(70→right)

t=70



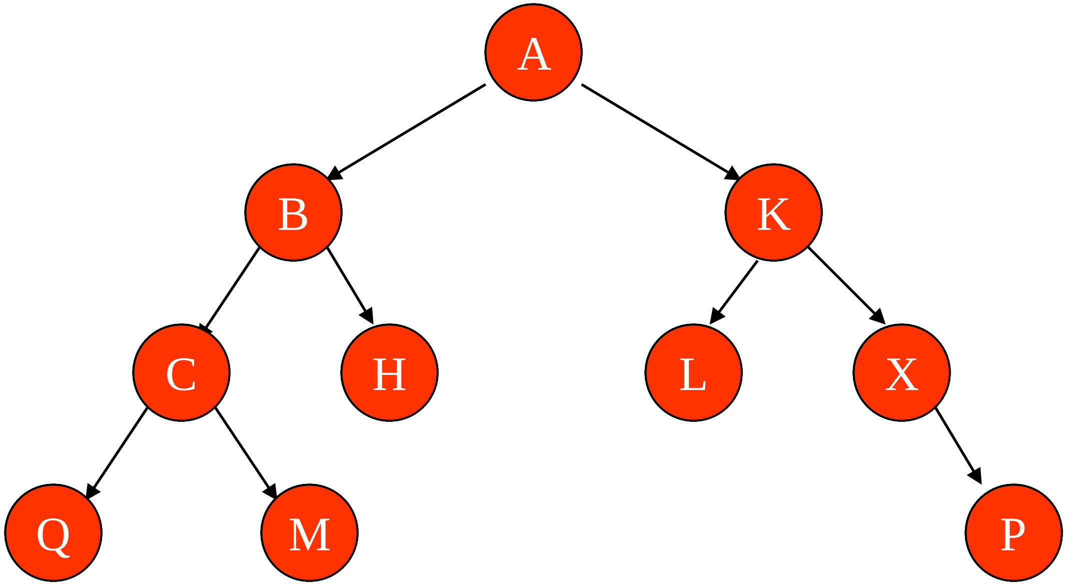
BST Print (Inorder traversal)

```
template <class type>
void tree<type>::Print() {
        InOrder(root);
}
```

```
template <class type>
void tree<type>::InOrder(Node<type> *t)
{
    if (t) {
        InOrder(t->left);
        visit(t);
        InOrder(t->right);
    }
}
```

```
template <class type>
void tree<type>:: ::visit(Node<type> *t){
        cout << t->data;
}
```

Binary Tree Traversal
In Order Traversal (LNR)

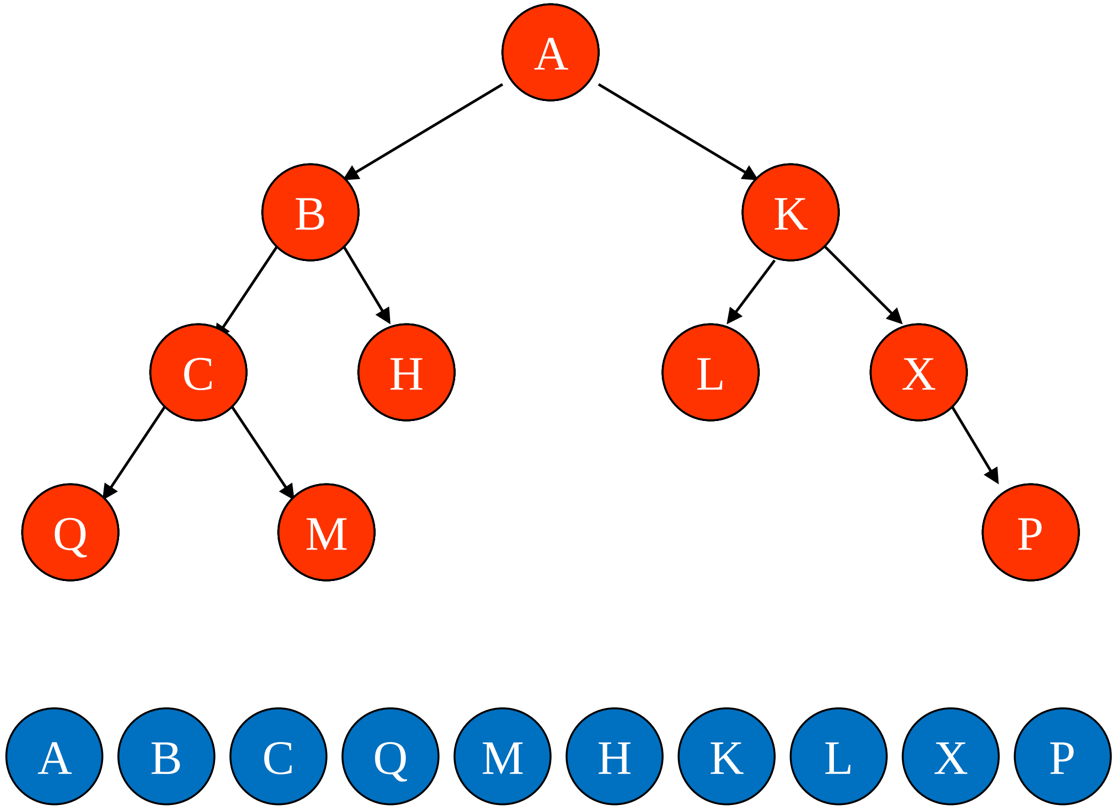


```
if (t) {
        InOrder(t→left);
        visit(t);
        InOrder(t→right);
}
```

Q C M B H A L K X P

Binary Tree Traversal

Pre Order Traversal (NLR)



```
if (t) {
        visit(t);
        PreOrder(t→left);
        PreOrder(t→right);
}
```

Binary Tree Traversal

Pre Order Traversal (NLR)

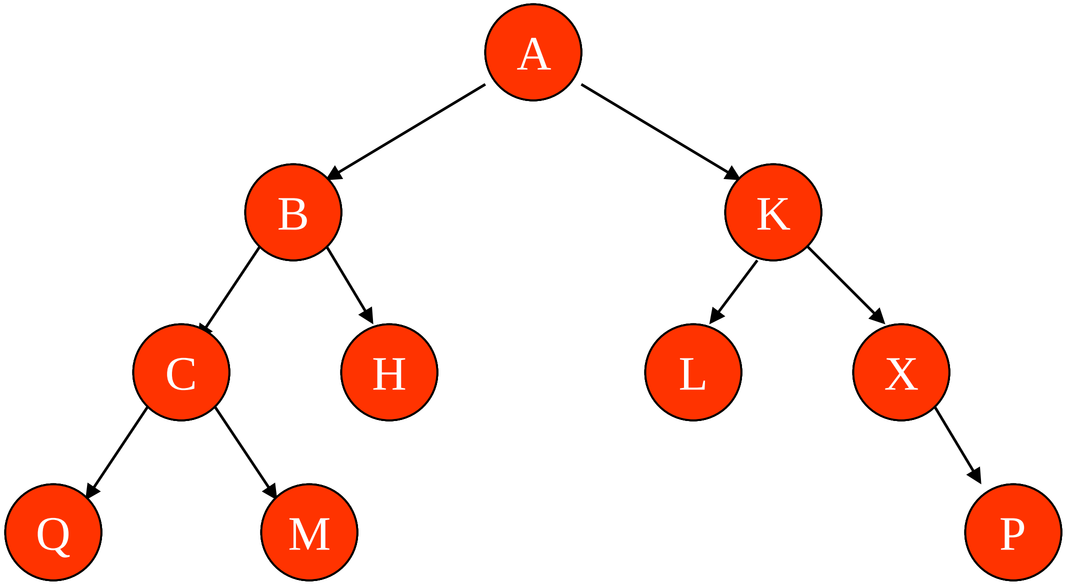
```
void BinaryTree::PreOrder()
{
        PreOrder(root);
}
```

```
void BinaryTree::PreOrder(TreeNode *t)
{
    if (t) {
        visit(t);
        PreOrder(t→left);
        PreOrder(t→right);
    }
}
```

```
void BinaryTree::visit(TreeNode *t) {
        cout << t→data;
}
```

Binary Tree Traversal

Post Order Traversal (LRN)



Q M C H B L P X K A

```
if (t) {
    PostOrder(t→left);
    PostOrder(t→right);
    visit(t);
}
```

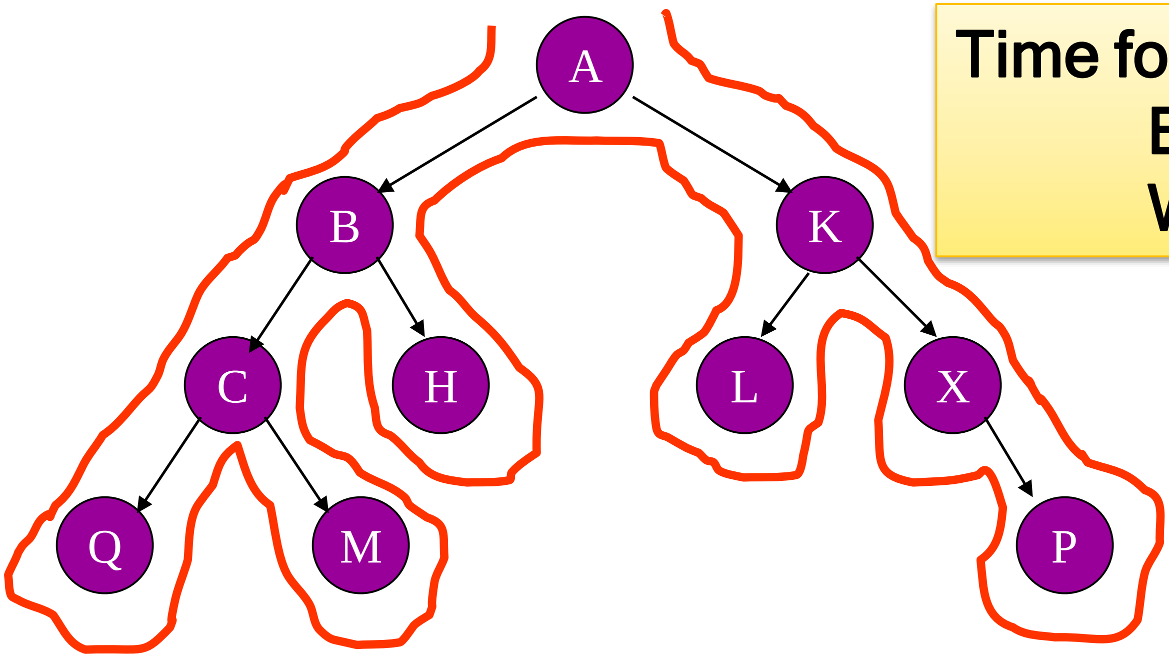
Binary Tree Traversal
Post Order Traversal (NLR)

```
void BinaryTree::PreOrder()
{
        PostOrder(root);
}
```

```
void BinaryTree::PostOrder(TreeNode *t)
{
    if (t) {
        PostOrder(t→left);
        PostOrder(t→right);
        visit(t);
    }
}
```

```
void BinaryTree::visit(TreeNode *t) {
        cout << t→data;
}
```

Binary Tree Traversal



Time for Traversal

Best case

Worst case

NLR – visit when at the left of the Node **A B C Q M H K LX P**

LNR – visit when under the Node **Q C M B H A L KX P**

LRN – visit when at the right of the Node **Q M C H B L P XK A**

Binary Tree Destructor

Which Algorithm?

Delete both the left child and right child before deleting itself

LRN (post order)

Destructor

```
void tree<type>::~tree() {
        Destroy(root);
}
```

POST ORDER traversal

```
template <class type>
void tree<type>::Destroy(Node<type> *& node) {
    if (node) {
        Destroy(node->left);
        Destroy(node->right);
        delete node;
    }
}
```

Duplicate Tree -Copy constructor

PRE ORDER traversal

```
template <class type>
void tree<type>::Duplicate(Node<type>*curr_tree,Node<type>*& dup_tree){
    if (curr_tree) {
    // set the value from curr_tree
        dup_tree = new Bnode(curr_tree->data);
        Duplicate(curr_tree->left, dup_tree->left);
        Duplicate(curr_tree->right, dup_tree->right);
    }
}
```

Tree traversal

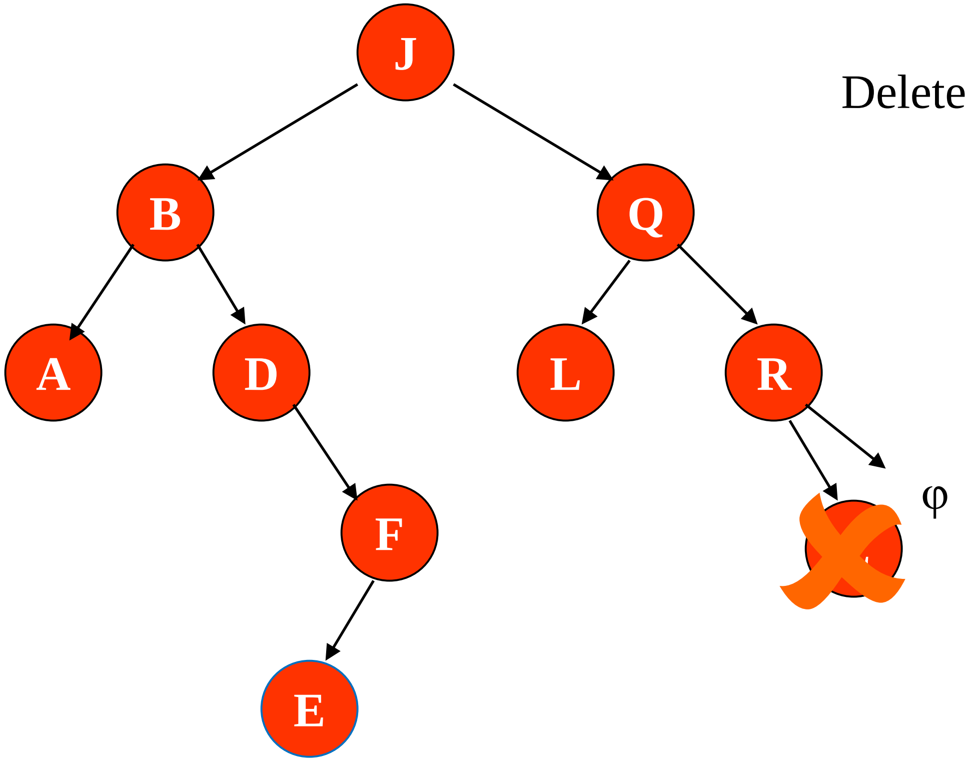
- In-order
  - Sorted data printing
- Post Order
  - Destructor
- Pre Order
  - Create a Tree copy (duplicate the current tree)

HW

Do Iterative versions of the traversals

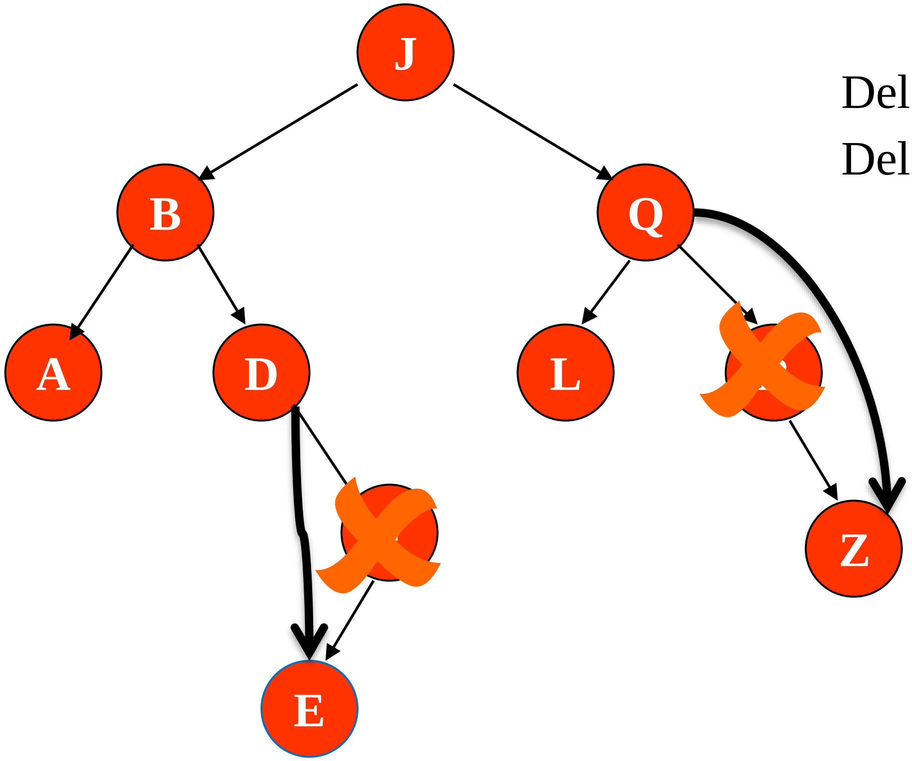
BST DELETION

Deleting a leaf node



Delete Z

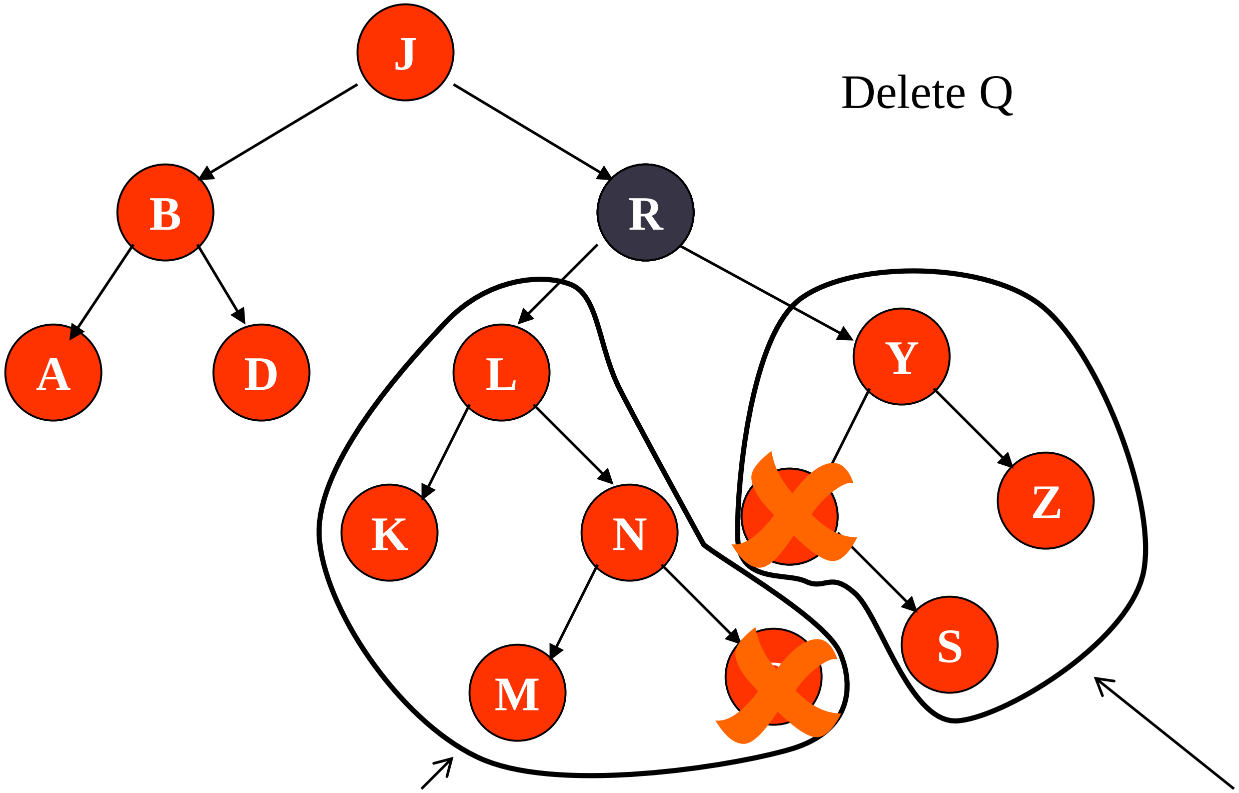
Deleting a node with only one child



Delete R

Delete F

Deleting a node with 2 children
(without duplication)



Delete Q

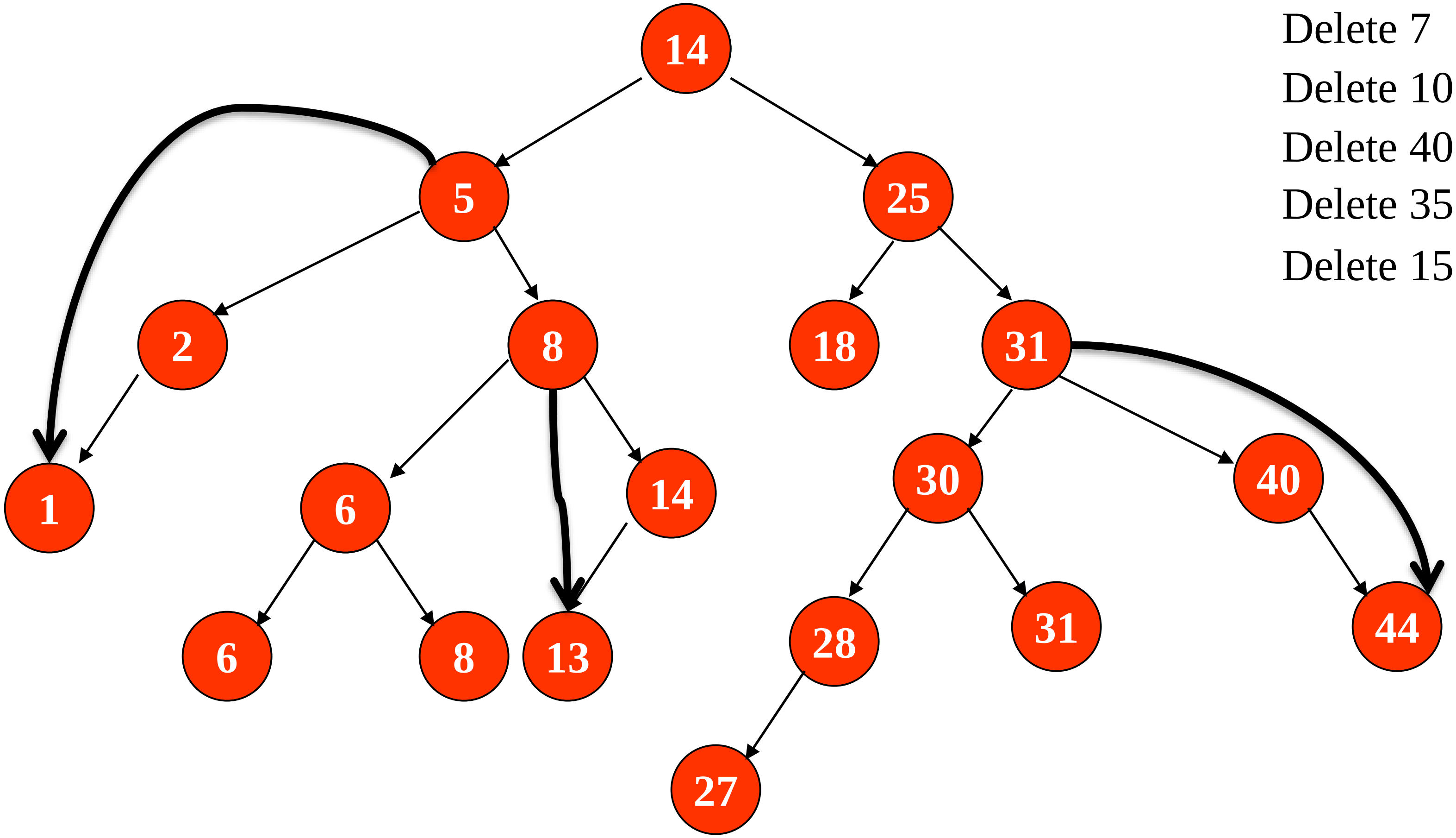
Largest from left subtree
(Predecessor)

smallest from right subtree
(Successor)

Delete a node from a BST (without duplication)

1. Search the node to be deleted; call it **t**
2. If t is a leaf,
   - disconnect it from its parent and set the parent pointer to NULL
3. If t has **only one child,**
   - remove t from the tree by making **t's** parent point to its child.
4. If t has **two children**
   1. Find the largest/smallest among **t's** LST/RST; call it **p**.
   2. Copy **p's** information into t.
   3. Delete **p.**

Delete(without duplication)



Delete 2

Delete 7

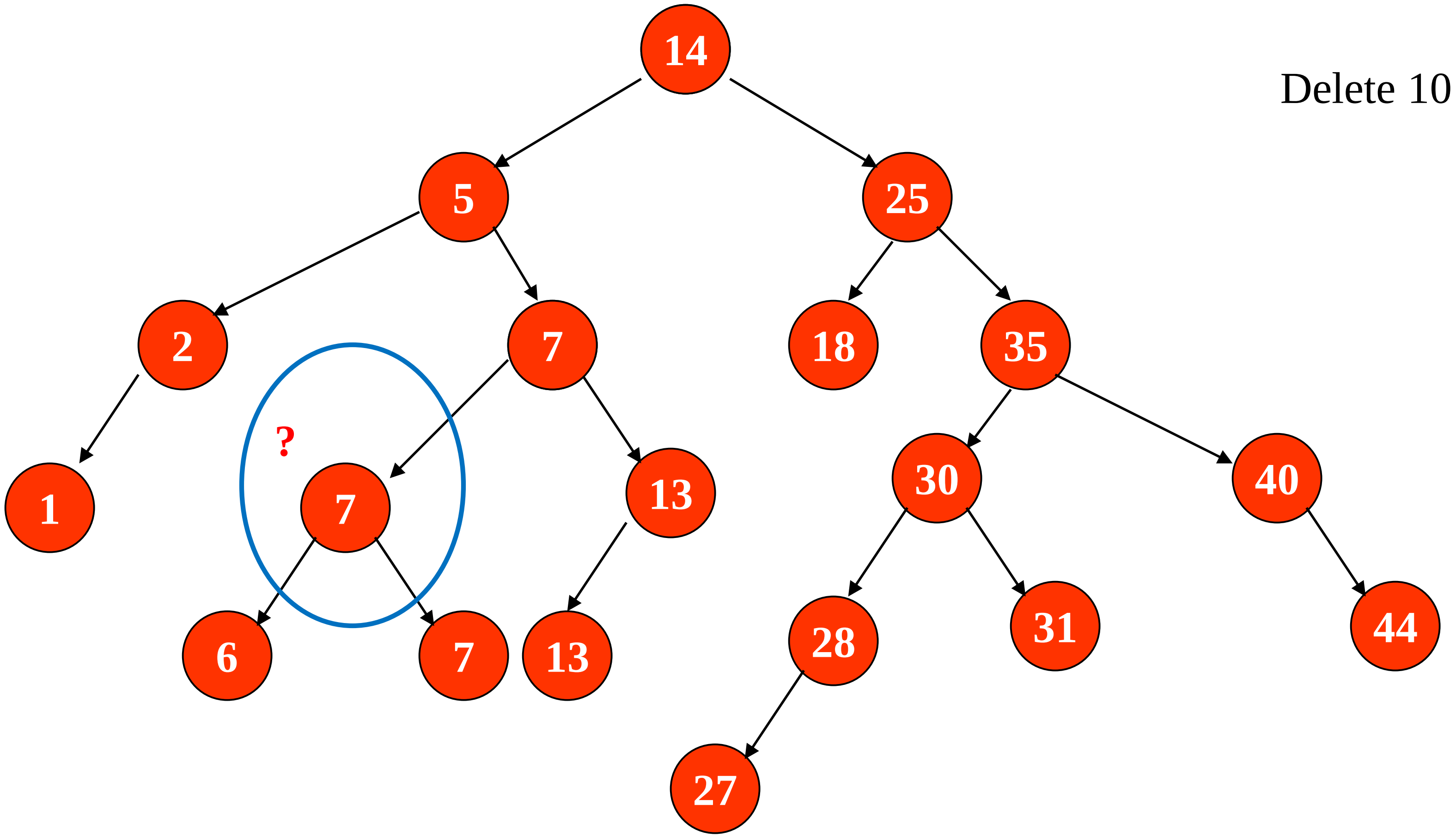
Delete 10

Delete 40

Delete 35

Delete 15

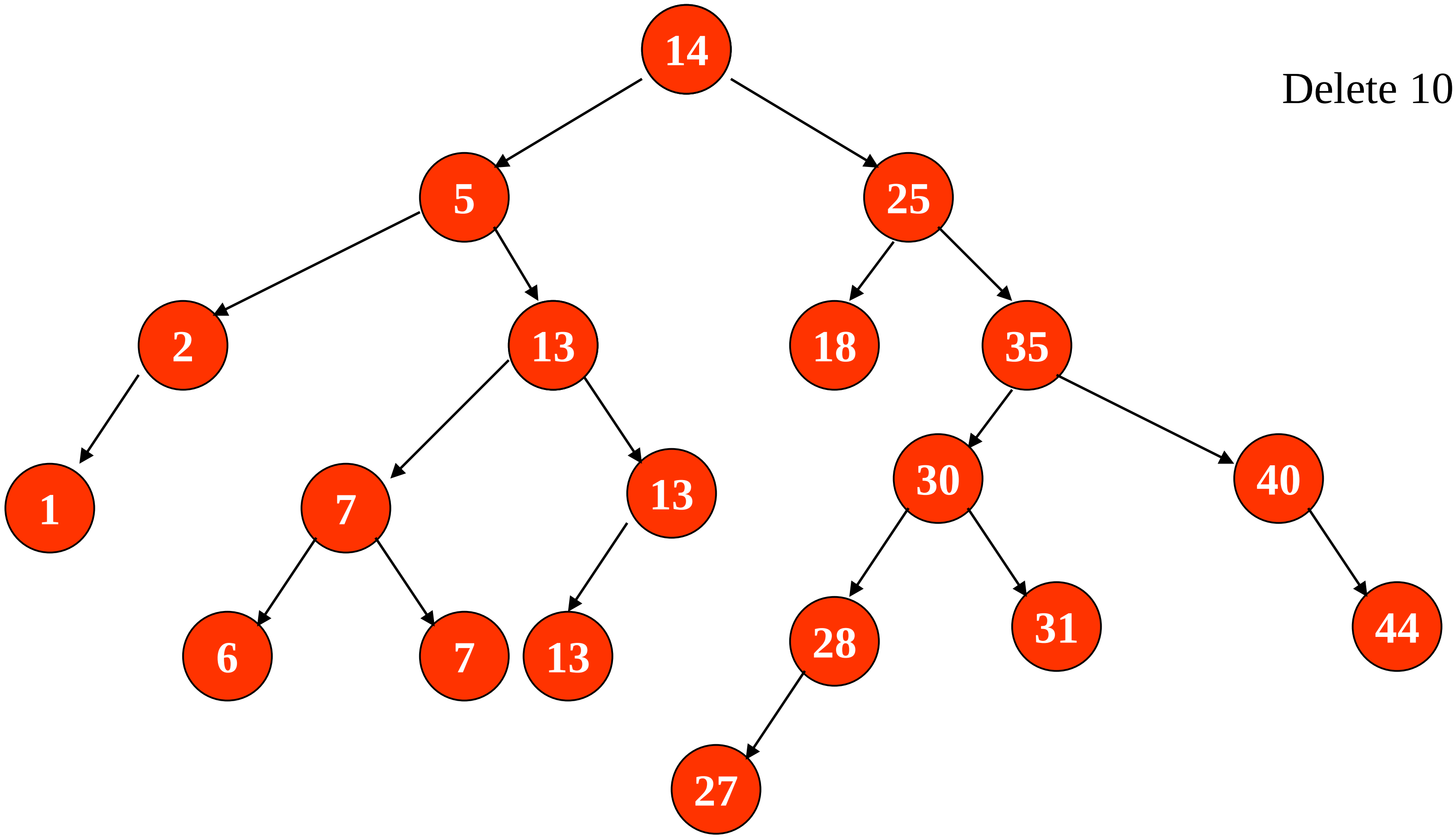
Delete (with Duplication)



Delete 10

Solution :Use Successor instead of Predecessor

Delete (with Duplication)



Delete 10

Recursive Delete

```
void tree<type>::deleteR(type d)
{
       deleteR(d, root);
}
```

```
template <class type>
void tree<type>::deleteR(type d, Node<type> *& node)
{
    if (node) {
        if (d > node->data)
               deleteR(d, node->right);
        else if (d < node->data)
               deleteR(d, node->left);
        else
               deleteNode(node);
    }
}
```

Recursive Delete

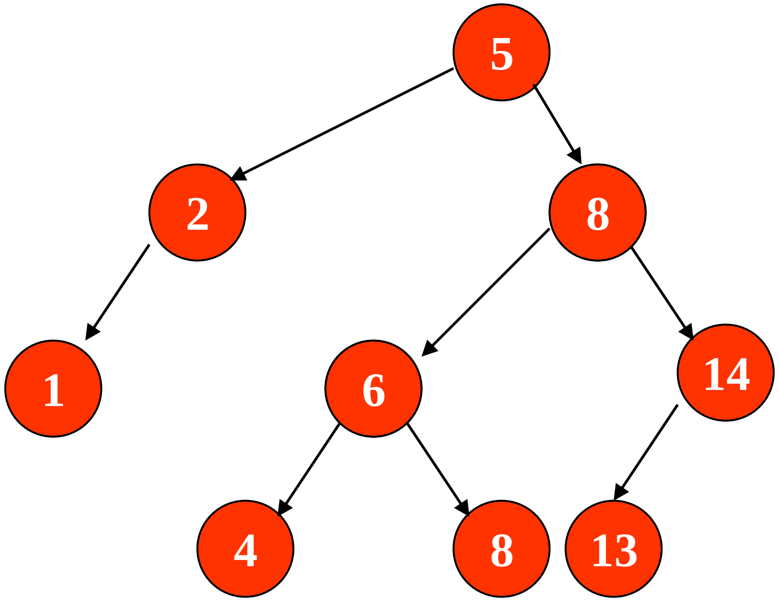
```
template <class type>
void tree<type>::deleteNode(Node <type> *& node) {

    Node <type> * temp = node;
    if (node->left == NULL) {
        node = node->right;
        delete temp;
    }
    else if(node->right == NULL) {
        node = node->left;
        delete temp;
    }
    else{
        type d = getPredecessor(node->left);
        node->data = d;
        deleteR(d, node->left);
    }
}
```

```
template <class T>
T tree<T>::getPredecessor(Node<T>*n)
{
    while (n->right != NULL)
        n = n->right;
    return n->data;
}
```

Iterative delete

```
template <class type>
void tree<type>::deleteI(type d) {
    Node <type> * parent = root;
    Node <type> * child = root;
    while (child && child→data != d) {
        parent = child;
        if (parent→data > d)
               child = child →left;
        else if (parent→data < d)
               child = child →right;
    }
    if (child) { //if data is found
        if (child == root)
               deleteNode(root);
        else if (parent→left == child)
               deleteNode(parent→left);
        else
               deleteNode(parent→right);
    }
}
```


5
2
8
1
6
14
4
8
13