BINARY SEARCH TREE

SEARCH

- **Sorted Array: Binary Search**
  - O(log N)
- **Linked List: Linear Search**
  - O(N)
- **Can we Apply the Binary Search algorithm on a linked list?**
- **Why not?**

SORTED ARRAY

- Rigid Structure
  - Fixed Size
  - Need to know the size of the largest data set
  - Wastage
- Search: O (log n)
- Insertion: O (n)
- Deletion: O (n)

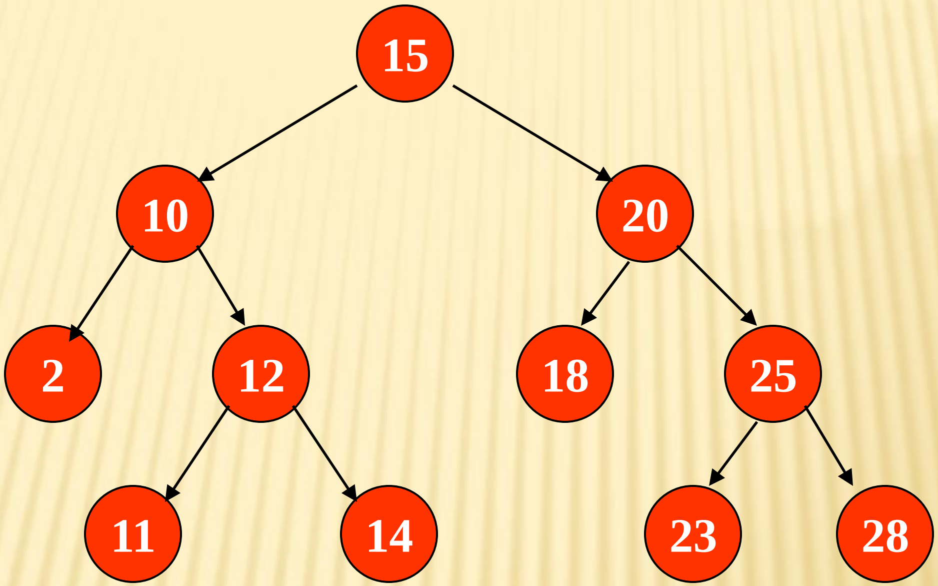
BINARY SEARCH TREE

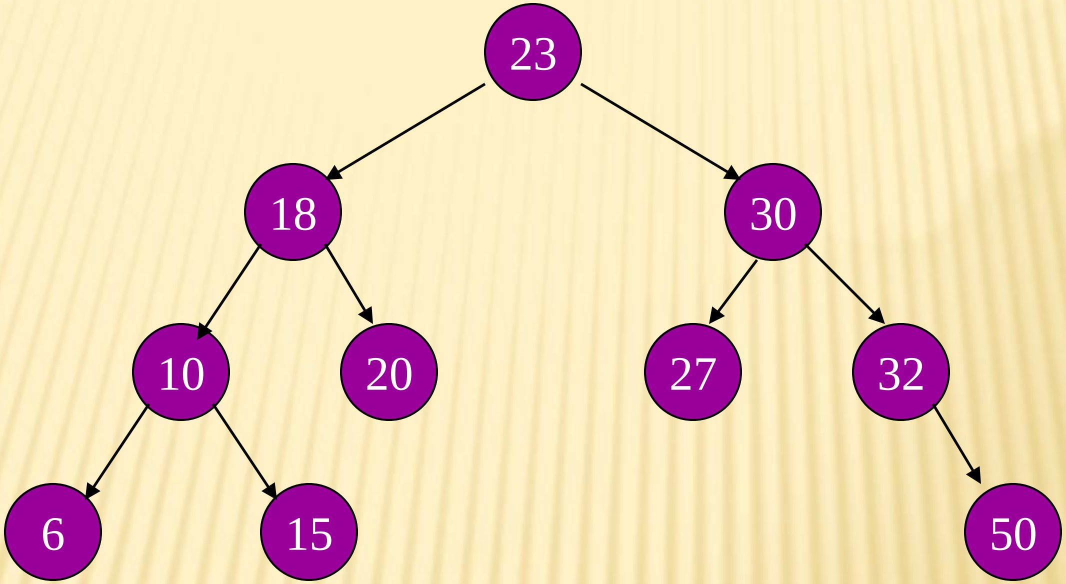
- Binary Tree
- Dynamic Structure (size is flexible)
- Data is stored in a sorted fashion
- A special class of BST has the following properties:
  - Search: O (log n)
  - Insertion: O (log n)
  - Deletion: O (log n)

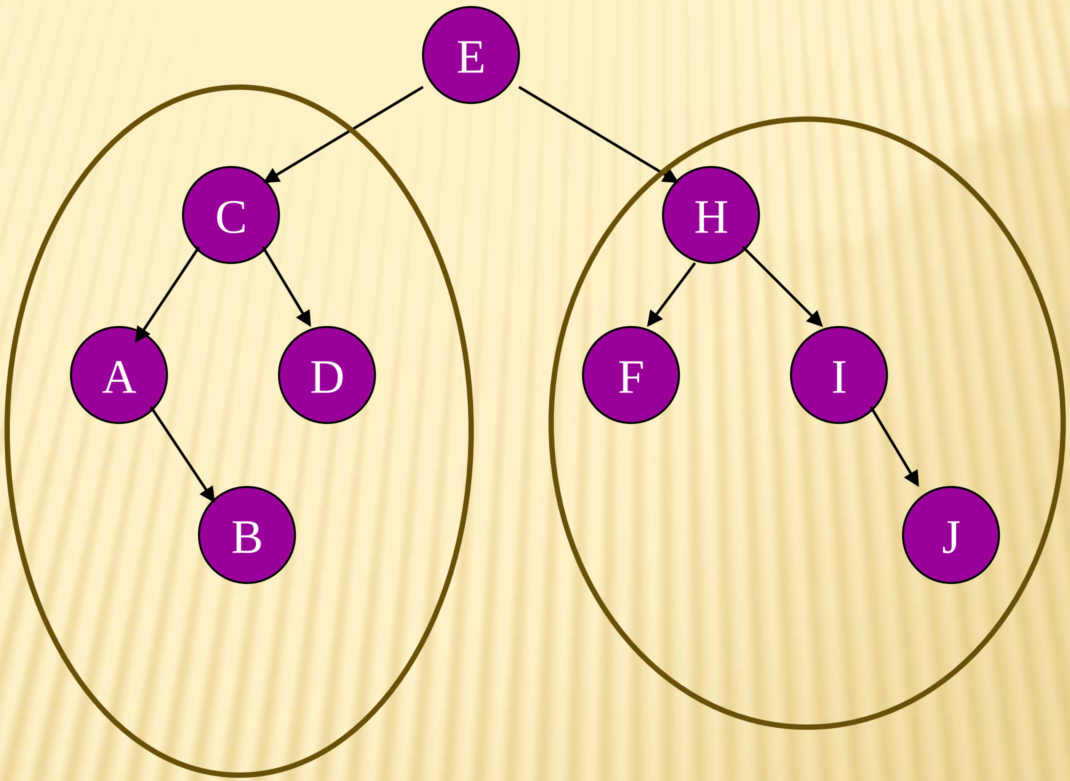
BINARY SEARCH TREE (BST)

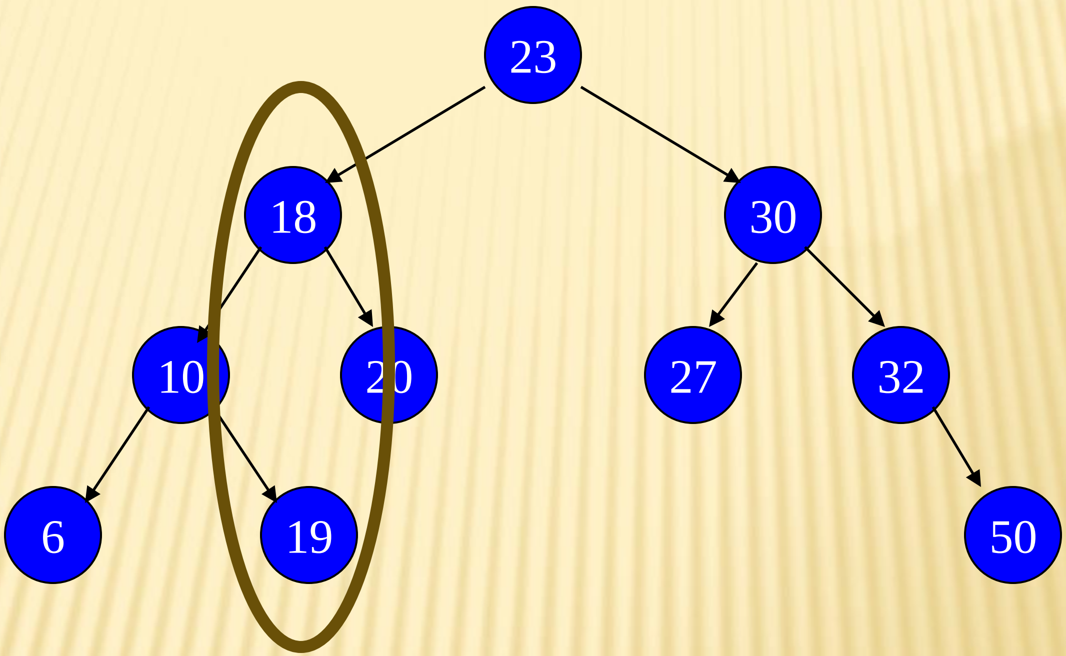
- A BST is a binary tree with the following properties:

1. Data value in the root node is greater than all the data values stored in the left subtree and is less than or equal to all the values stored in the right subtree.
2. Both the left subtree and right subtree are BSTs.


15
10
20
2
12
18
25
11
14
23
28


23
18
30
10
20
27
32
6
15
50


E
C
H
A
D
F
I
B
J



Violating the condition for BST

NODE

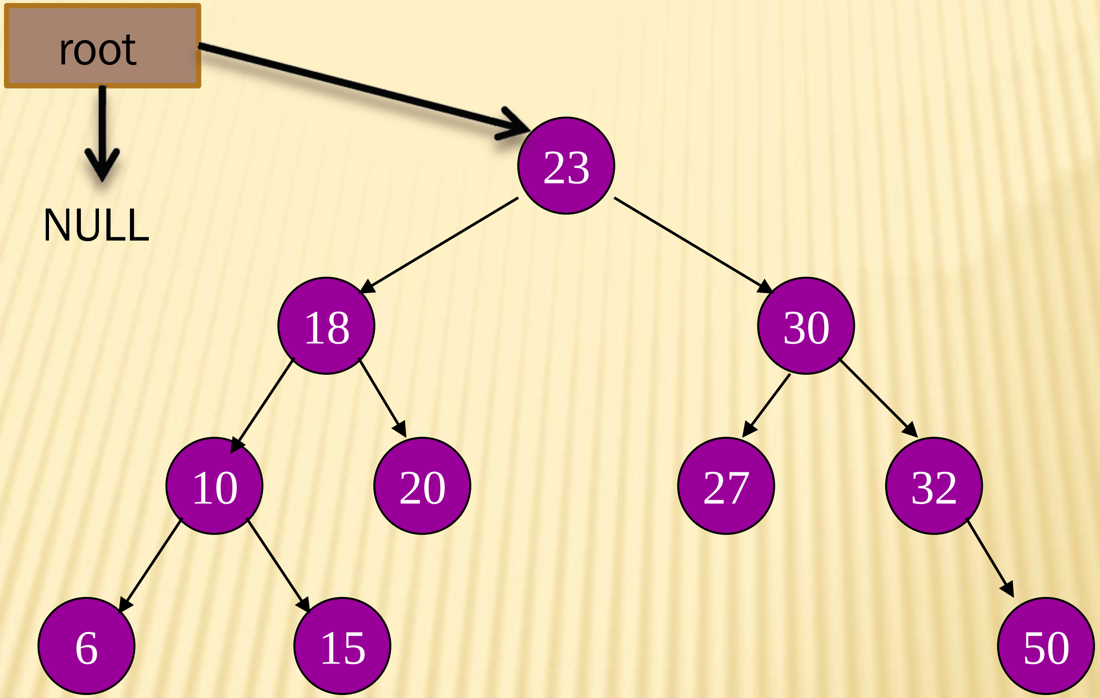
```
template <class type>
class Node{
public:
        type data;
        Node * left;
        Node * right;
        Node (type d = 0);
};

template <class type>
Node <type>::Node(type d){
        data = d;
        left = NULL;
        right = NULL;
}
```

TREE

```
template <class type>
class tree{
private:
		Node<type> * root;
public:
		tree();
		void inOrder(Node<type> * iterator);
		void inOrder(){inOrder(root);}
		void insertR(type d, Node <type> *& node );
		void insertR(type d){ insertR(d,root);}
		void insertI(type d);
		void visit(Node<type> * ptr){cout<<ptr->data<<"        ";}
		bool searchR(Node<type> * node, type d);
		bool searchR(type d){ return searchR(root,d);}
		bool searchI(type);
		void deleteR(type d){ deleteR(d,root);}
		void deleteR(type d, Node<type> *& node);
		void deleteNode(Node <type> *& node);
		void getPredecessor(Node <type> * node,type & data);
		void deleteI(type d);
		void Destroy(Node<type> *& node);
		~tree(){Destroy(root);}
};
```

```
template <class type>
tree <type>:: tree(){
		root = NULL;
}
```

INSERTION

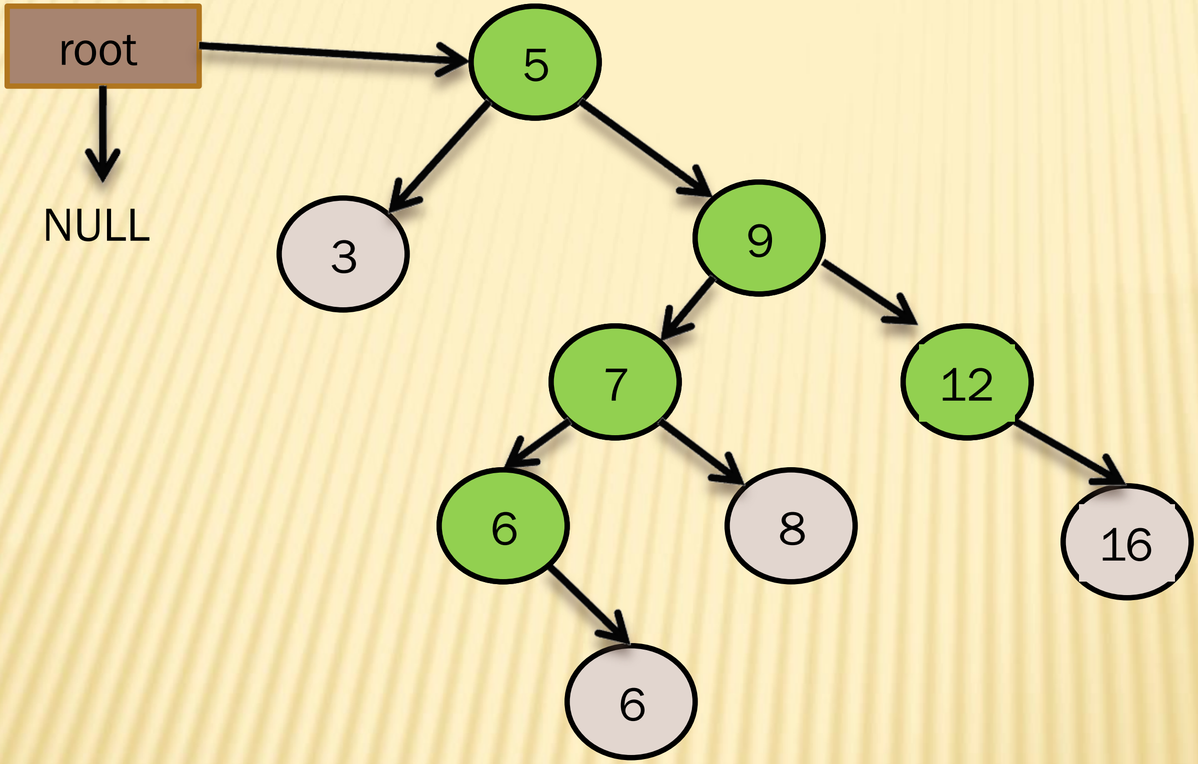


1. Insert 23
2. Insert 18
3. Insert 10
4. Insert 30
5. Insert 20
6. Insert 27
7. Insert 32
8. Insert 50
9. Insert 6
10. Insert 15

RECUSIVE INSERTION

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

            if(node==NULL){
                        node = new Node<type>(d);
            }
            else if(node->data > d)
                                    insertR(d,node->left);
            else
                        insertR(d,node->right);
}


 void insertR(type d){

             insertR(d,root);
 }
```

INSERTION

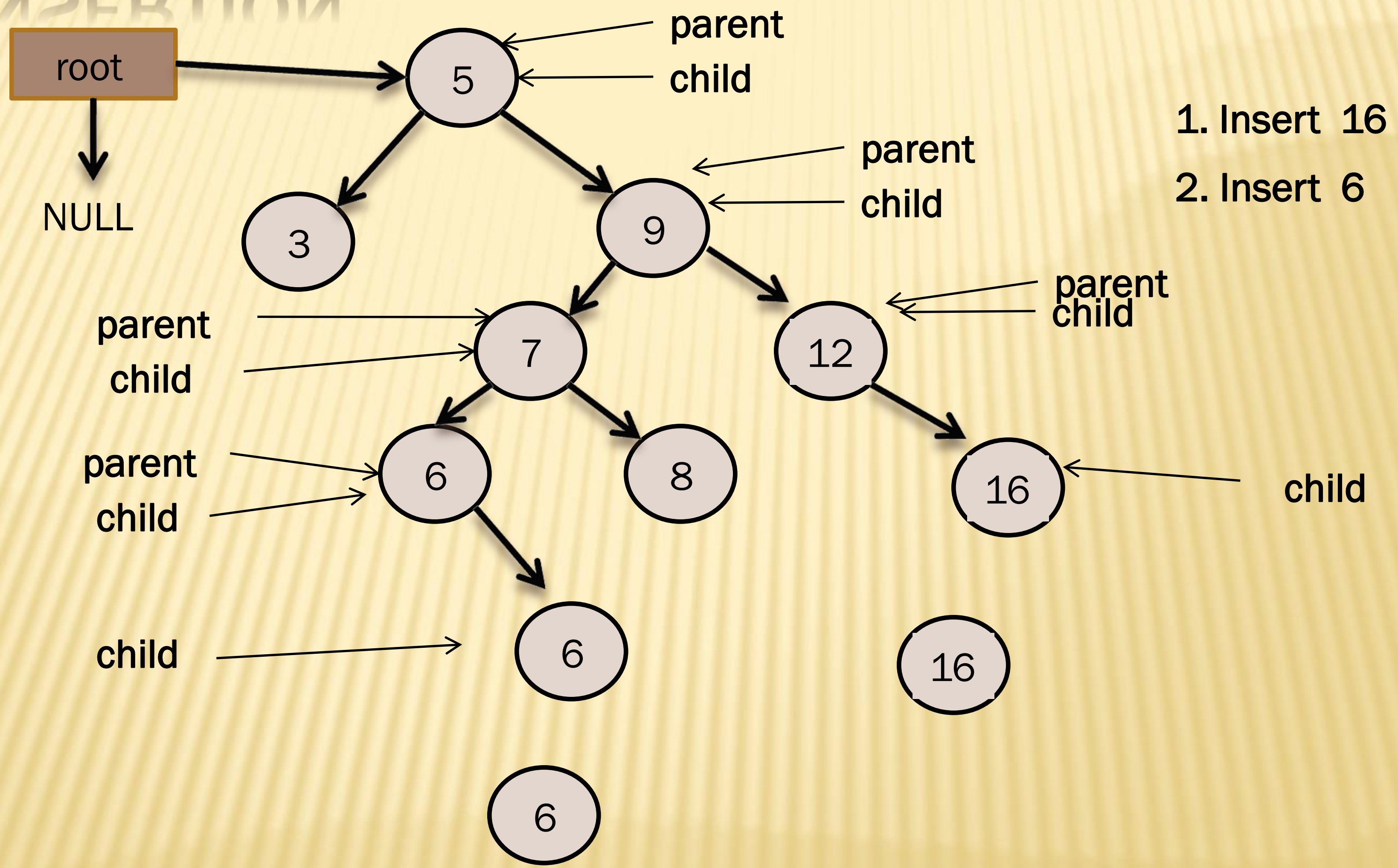


1. Insert 5
2. Insert 9
3. Insert 7
4. Insert 3
5. Insert 8
6. Insert 12
7. Insert 6
8. Insert 6
9. Insert 16

ITERATIVE INSERTION

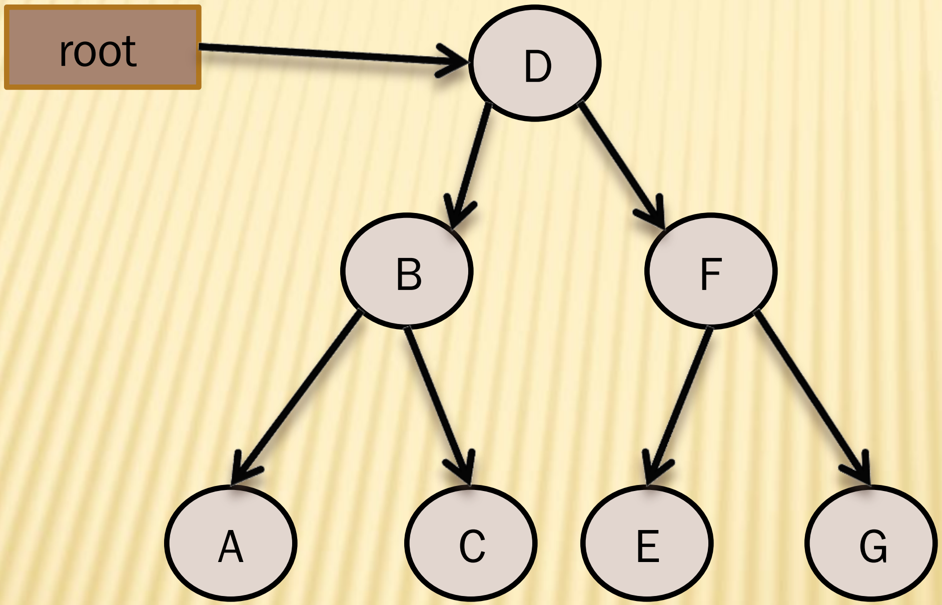
```
template <class type>
void tree<type>::insertI(type d){
        Node <type> * newNode = new Node<type>(d);
        Node <type> * parent = root;
        Node <type> * child = root;

        while(child){
                parent = child;
                if(parent->data > d)
                        child = child ->left;
                else if(parent->data <= d)
                        child = child ->right;
        }
        if(parent == NULL)
                root = newNode;
        else if(parent->data > d)
                parent->left = newNode;
        else if(parent->data <= d)
                parent->right = newNode;
}
```

INSERTION


root
NULL
5
parent
child
3
9
parent
child
7
12
parent
child
parent
child
6
8
16
child
parent
child
6
child
16
6
1. Insert 16
2. Insert 6

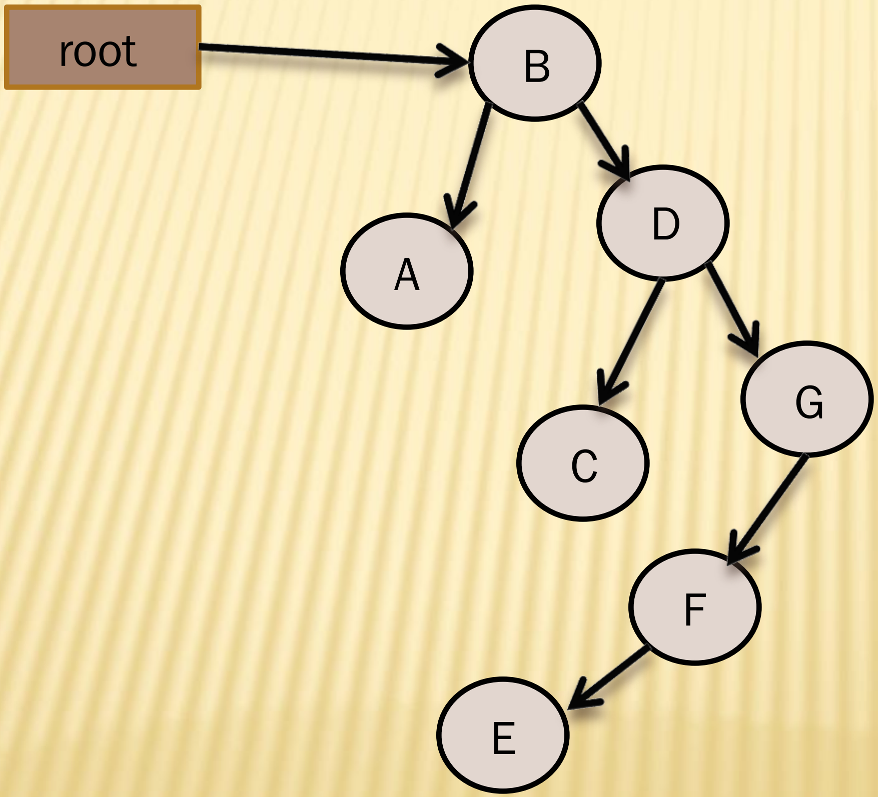
INSERTION ORDER

- Input: D B F A C E G



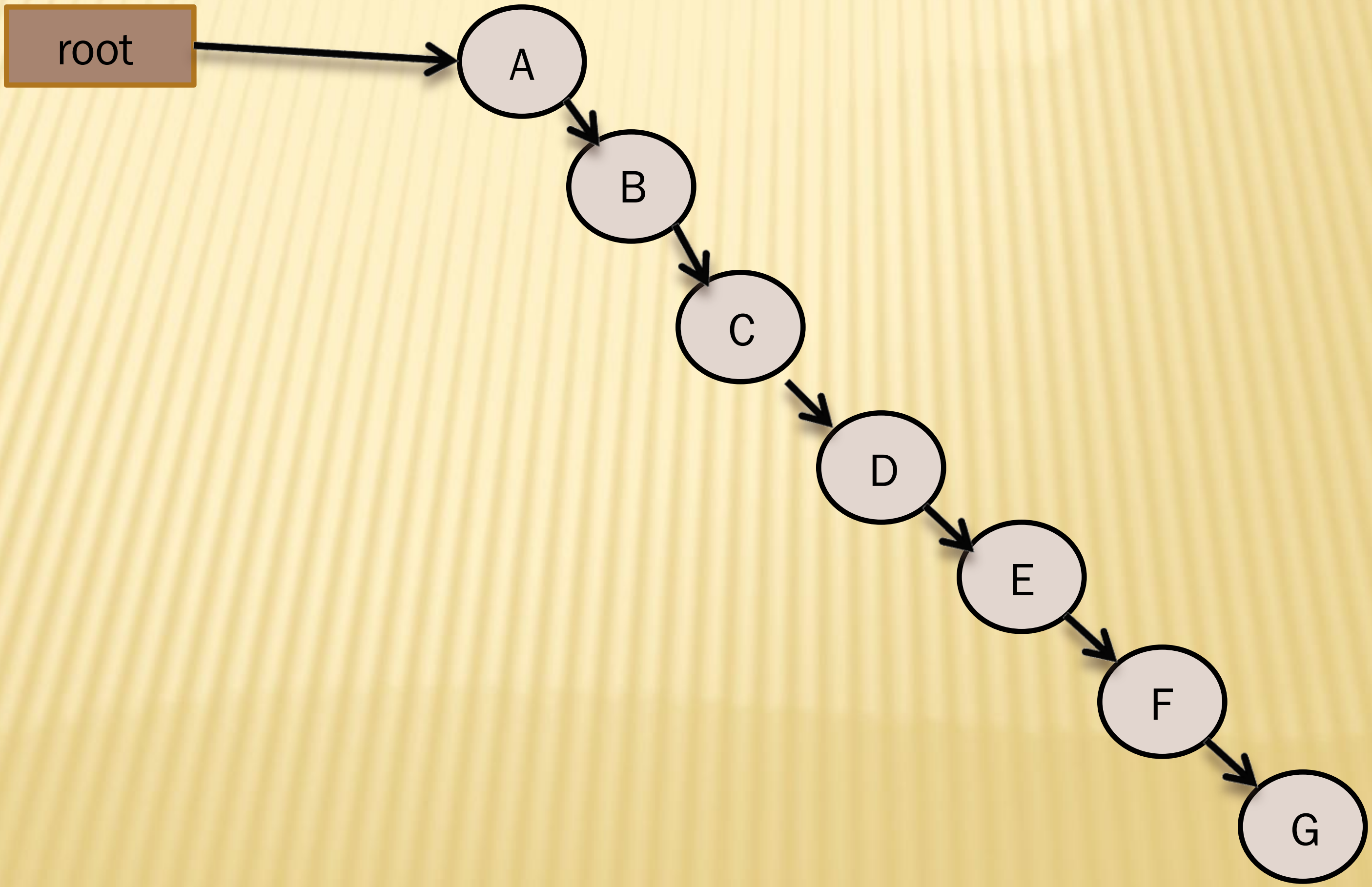
INSERTION ORDER

- Input: B A D C G F E



INSERTION ORDER

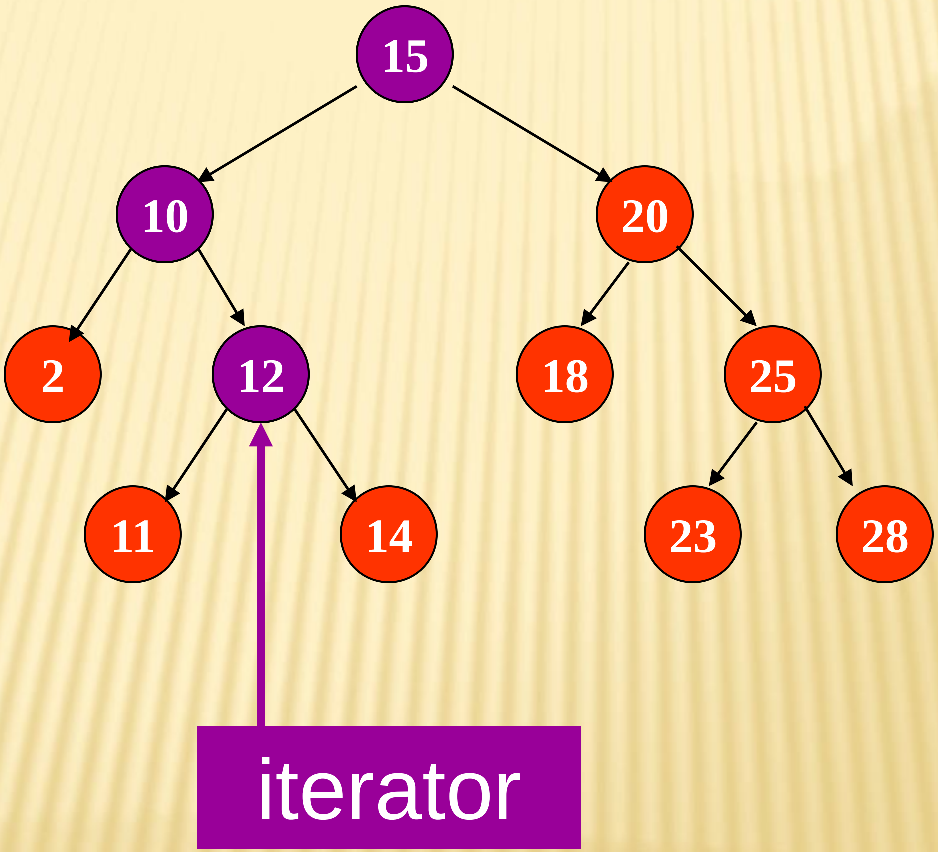
- Input: A B C D E F G



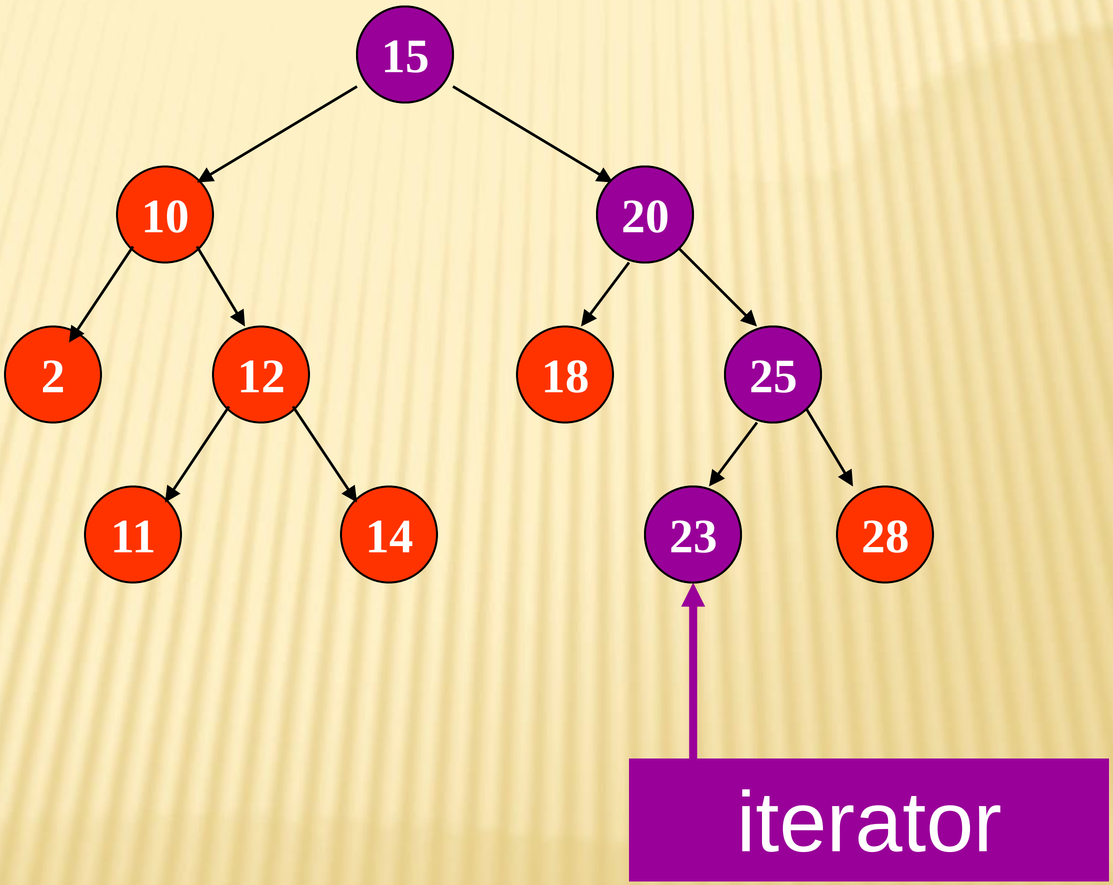
ITERATIVE SEARCH

```
template <class type>
bool tree<type>::searchI(type key){
        Node <type>* iterator= root;
        bool flag = false;
        while (iterator && !flag) {

                if (iterator->data == key)
                        flag = true;
                else if (iterator->data > key)
                        iterator = iterator->left;
                else
                        iterator = iterator->right;
        }
        return flag;

}
```

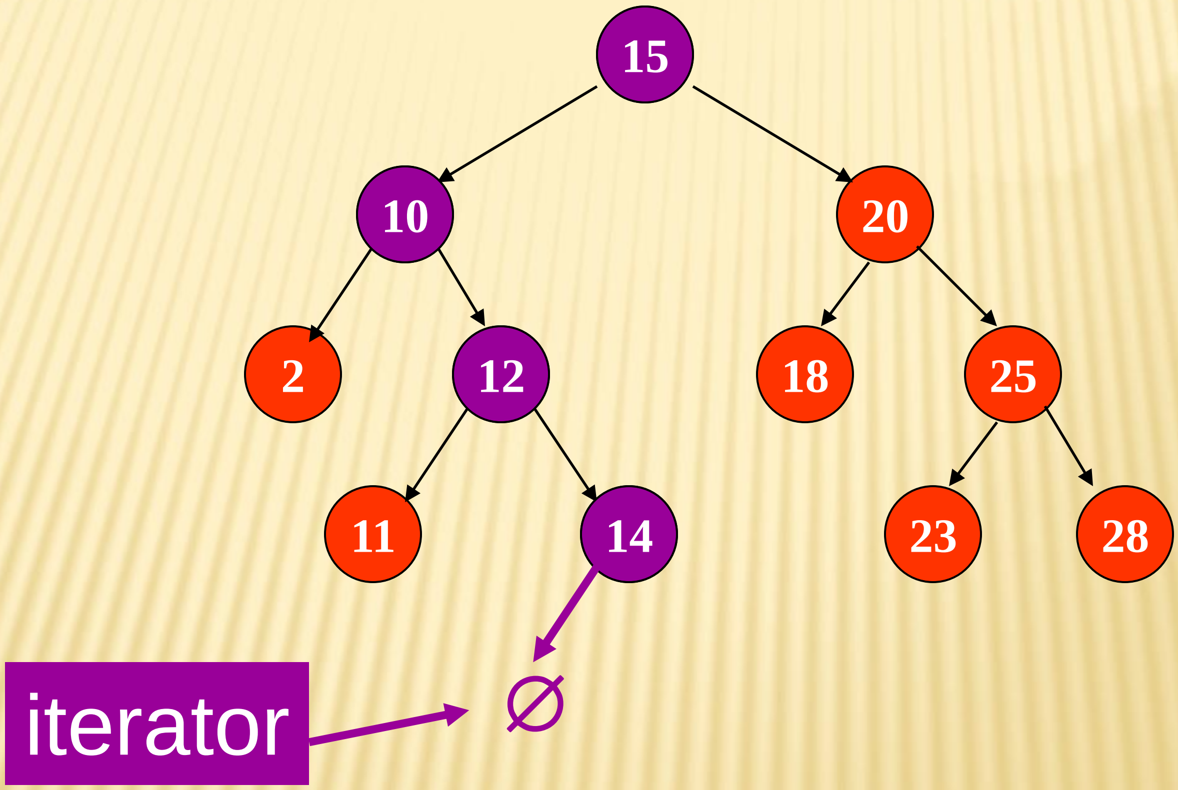
SEARCH(12)



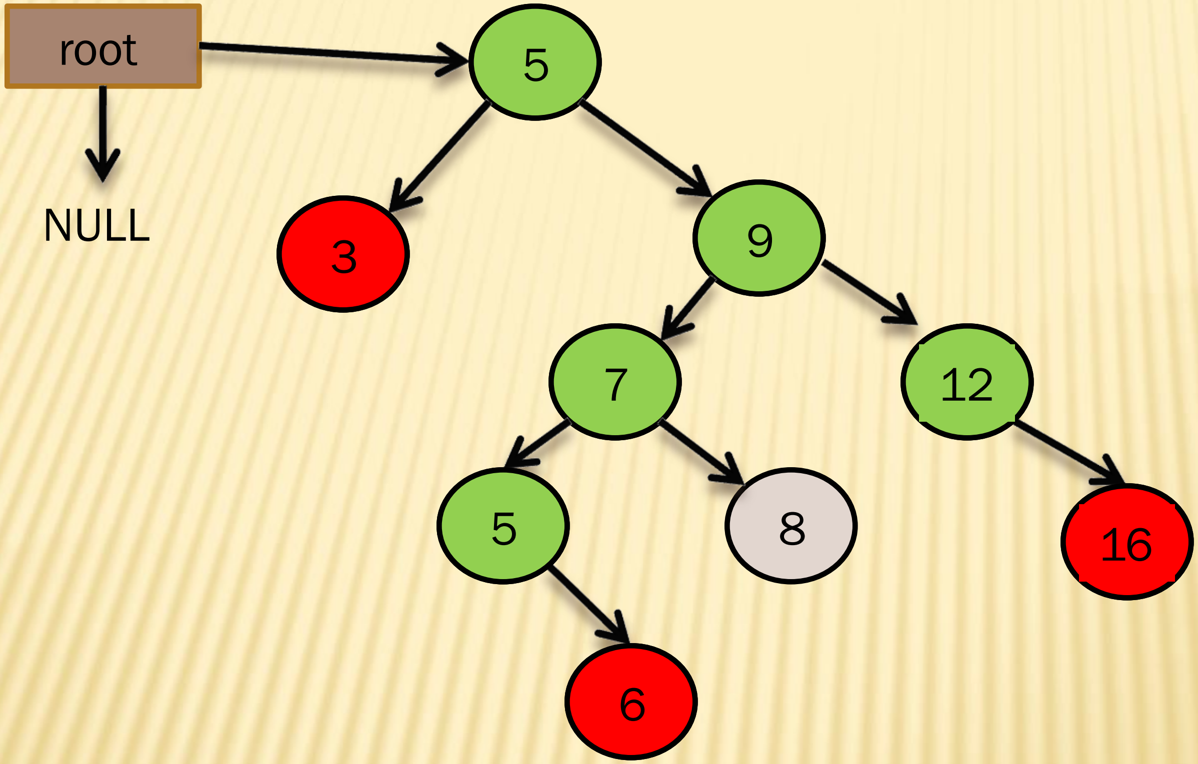
SEARCH(23)



SEARCH(13)



SEARCH



1. Search 3
2. Search 6
3. Search 16

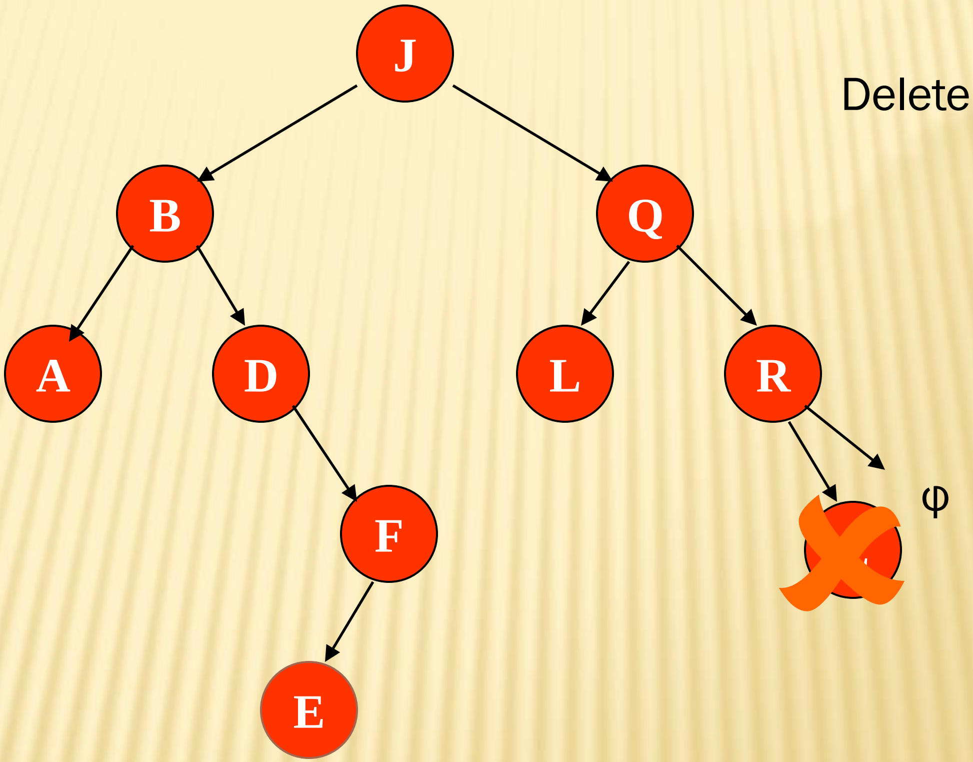
Write recursive version of search function

RECURSIVE SEARCH

```
template <class type>
bool tree<type>::searchR(Node<type> * node, type d){
        if(node){
                if(node->data>d)
                        searchR(node->left, d);
                else if(node->data<d)
                        searchR(node->right,d);
                else
                        return true;
        }
        else
                return false;
}
```

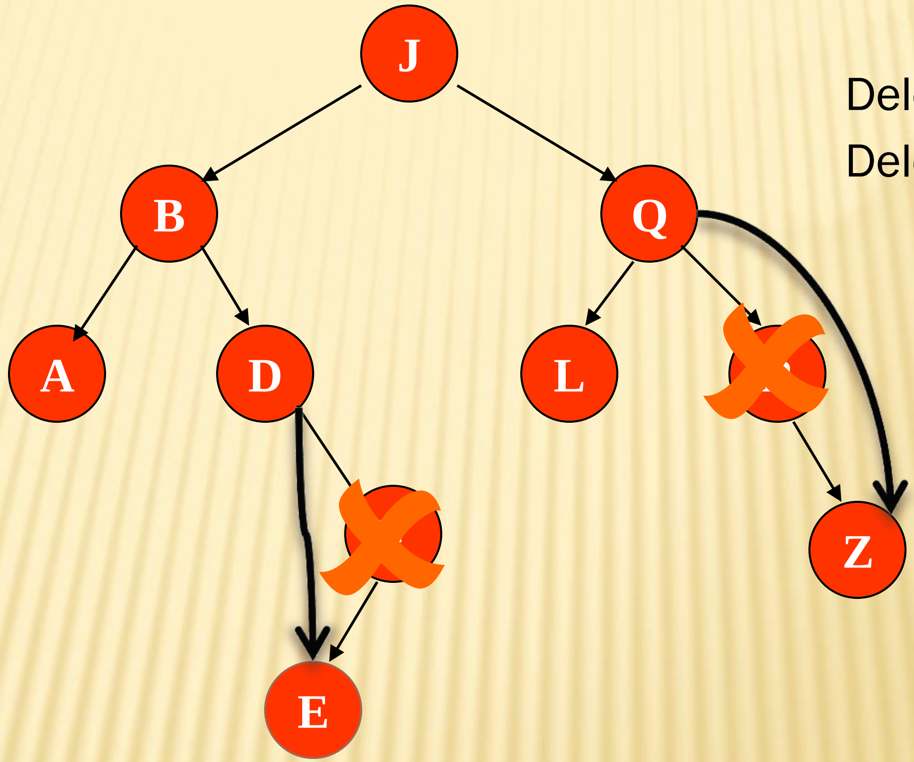
```
bool searchR(type d){

        return searchR(root,d);

}
```

DELETING A LEAF NODE



Delete Z

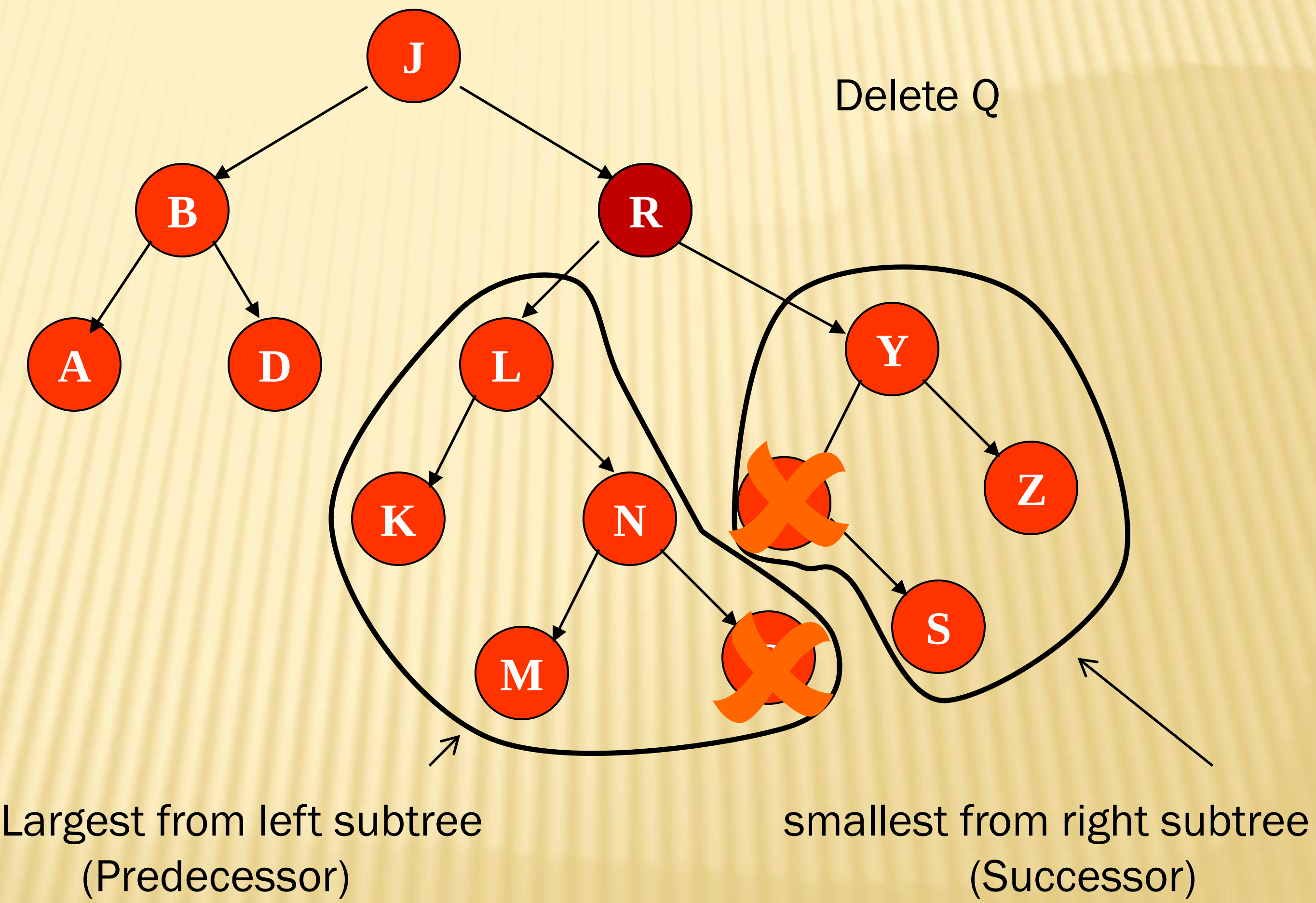
DELETING A NODE WITH ONLY ONE CHILD



Delete R

Delete F

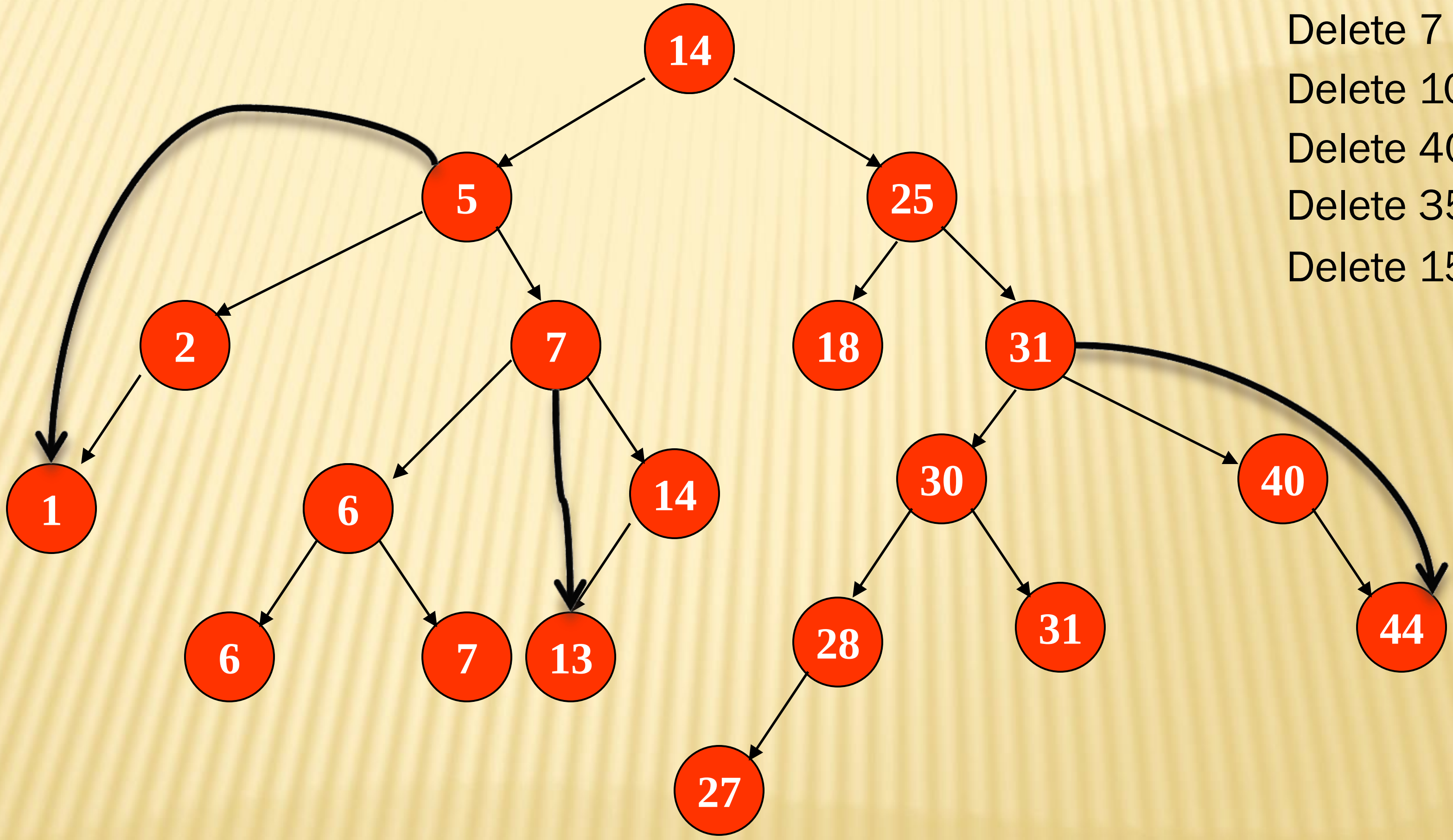
DELETING A NODE WITH 2 CHILDREN



DELETE A NODE FROM A BST

1. Locate the desired node by search; call it **t**
2. If **t** is a leaf, disconnect it from its parent and set the pointer in the parent node equal to NULL
3. If it has only one child then remove t from the tree by making **t's** parent point to its child.
4. Otherwise, find the largest/smallest among **t's** LST/RST; call it **p**. Copy **p's** information into t. Delete **p.**

DELETE



Delete 2
Delete 7
Delete 10
Delete 40
Delete 35
Delete 15

RECURSIVE DELETE

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


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

RECURSIVE DELETE

```
template <class type>
void tree<type>::getPredecessor(Node <type> * node,type & data){
        while(node->right!=NULL)
                node = node->right;
        data = node ->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(child == root)
                        deleteNode(root);
                else if(parent->left == child)
                        deleteNode(     parent->left);
                else
                        deleteNode(     parent->right);
        }
}
```

DESTRUCTOR

```
~tree(){
        Destroy(root);
}

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

NUMBER OF NODES IN A TREE

```
int tree<type>:: NumberOfNodes()const {
        return CountNodes(root);
}

template <class type>
int tree<type>::CountNodes(Node <type> * node){
        if(node == NULL)
                return 0;
        else
                return CountNodes(node->left)+ CountNodes(node->right) +1;
}
```

insert(15)

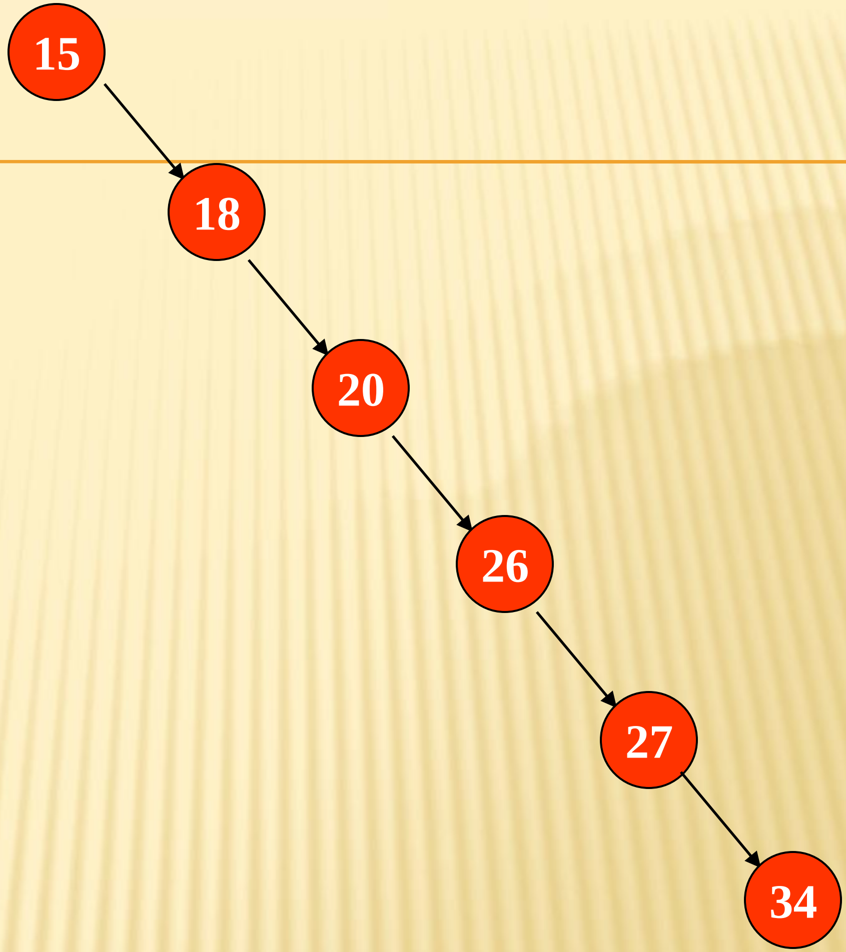
insert(18)

insert(20)

insert(26)

insert(27)

insert(34)



TIME COMPLEXITY

O(k) where k is the height

Height Balanced Trees

$k = \log(n)$

COMPARISON OF LINK LIST & BST

Operation	BST	Link List
Constructor	O(1)	O(1)
Destructor	O(N)	O(N)
Search	$O(\log_2 N)$	O(N)
Insert	$O(\log_2 N)$	O(N)
Delete	$O(\log_2 N)$	O(N)