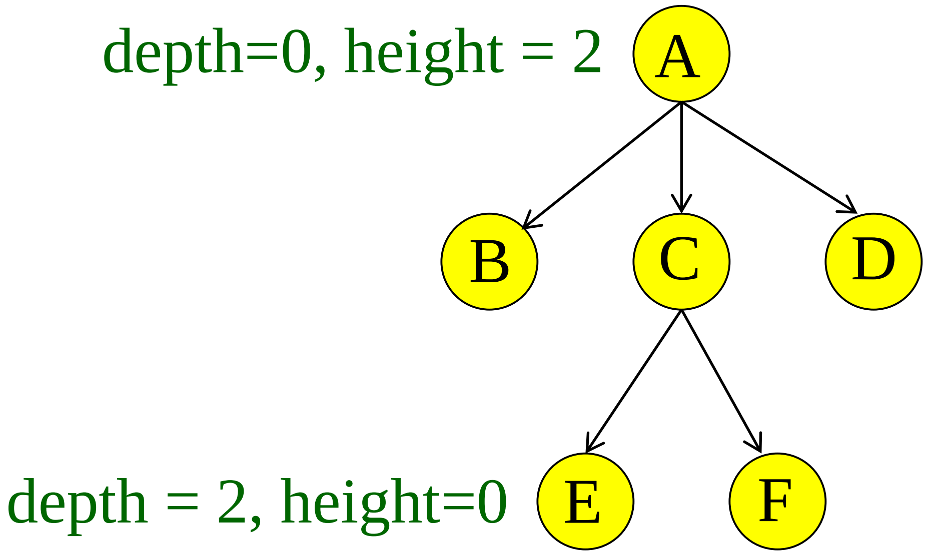
Introduction to Binary Search Trees

Tree Jargon

- Length of a path = number of edges
- Depth of a node N = length of path from root to N
- Height of node N = length of longest path from N to a leaf
- Depth and height of tree = height of root

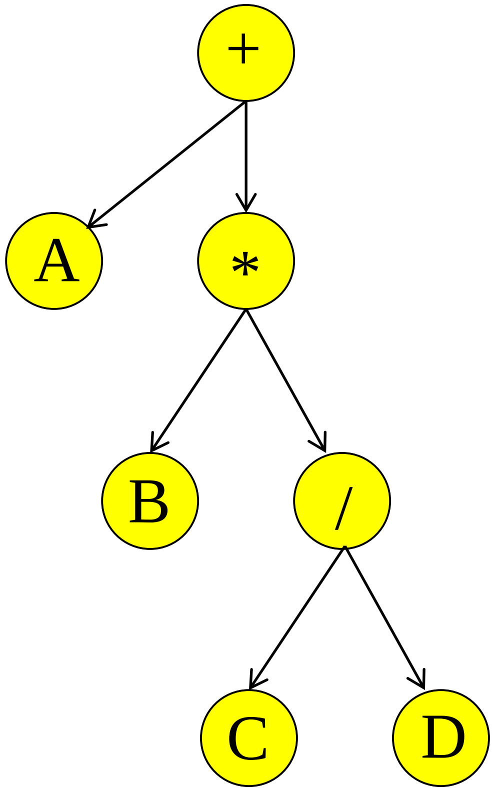


Application: Arithmetic Expression Trees

Example Arithmetic Expression:

A + (B * (C / D))

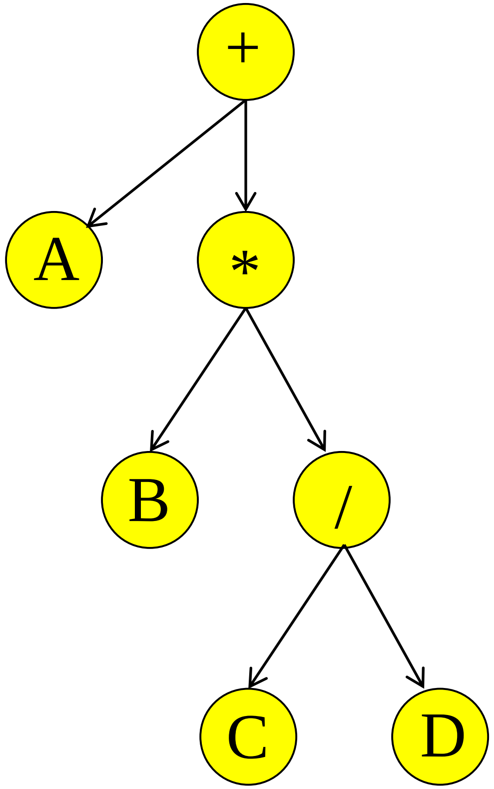
Tree for the above expression:



- Used in most compilers
- No parenthesis need – use tree structure
- Can speed up calculations e.g. replace / node with C/D if C and D are known
- Calculate by traversing tree (how?)

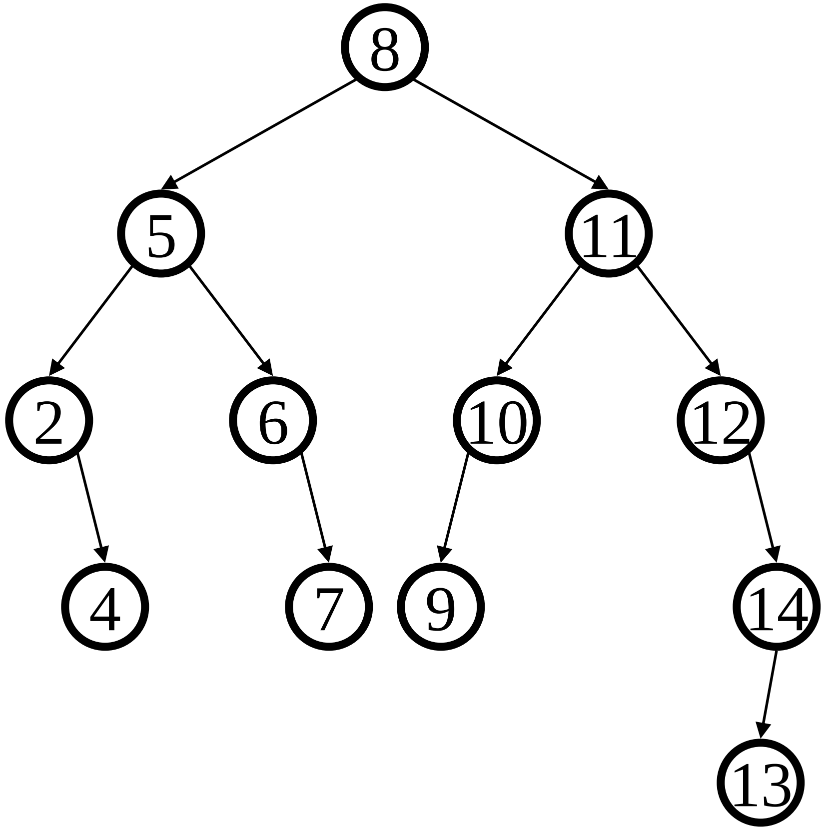
Traversing Trees

- Preorder: Root, then Children
  - + A * B / C D
- Postorder: Children, then Root
  - A B C D / * +
- Inorder: Left child, Root, Right child
  - A + B * C / D

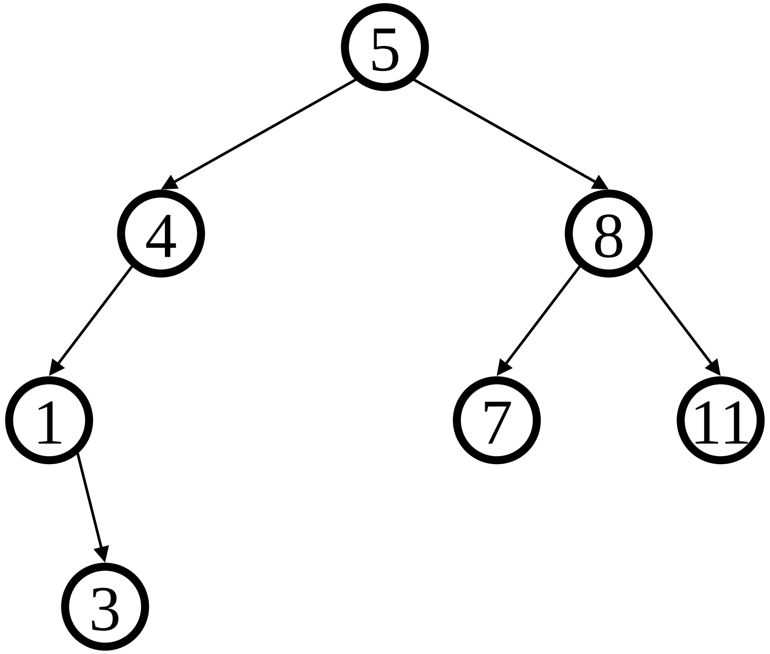


Binary Search Tree :

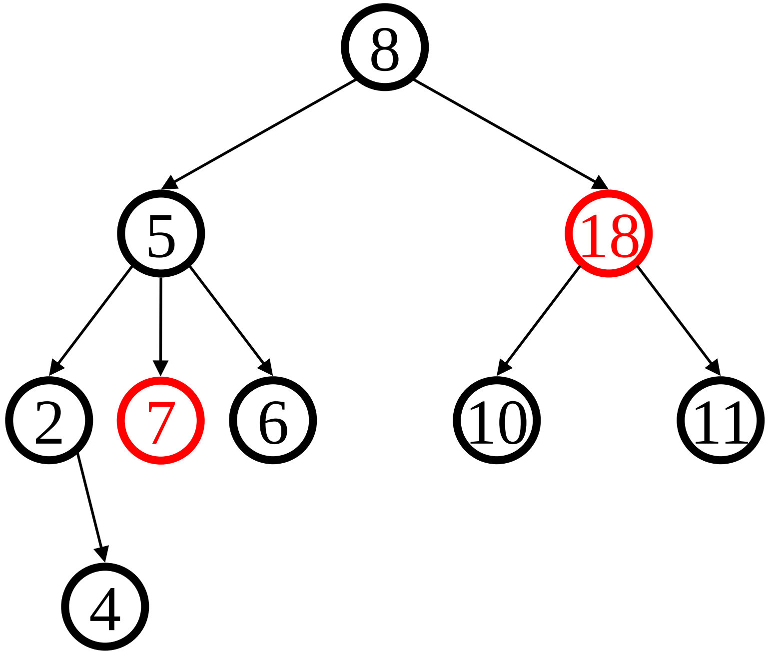
- Search tree property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result:
    - easy to find any given key
    - inserts/deletes by changing links



Example and Counter-Example



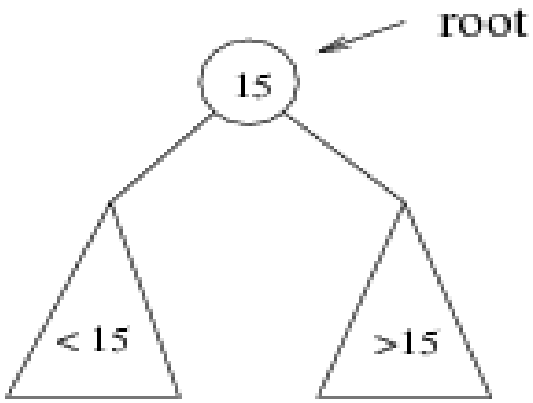
BINARY SEARCH TREE



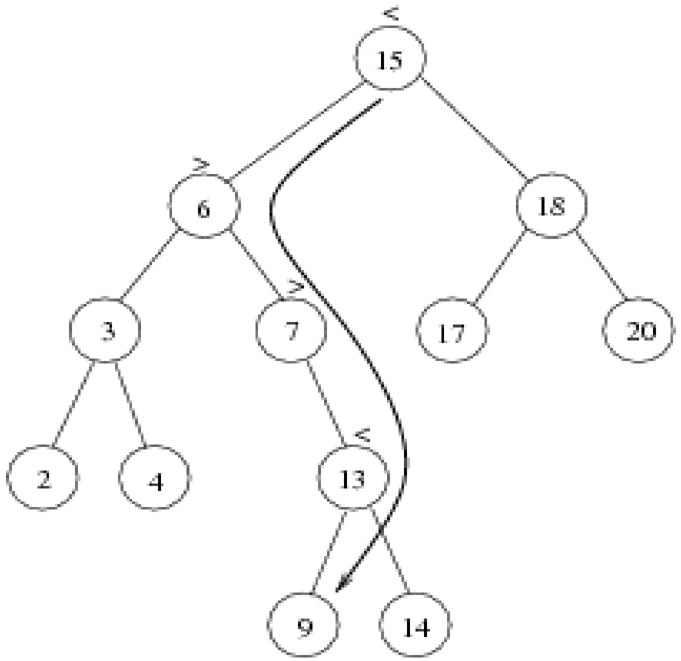
NOT A
BINARY SEARCH TREE

Searching BST:

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Example: Search for 9 ...

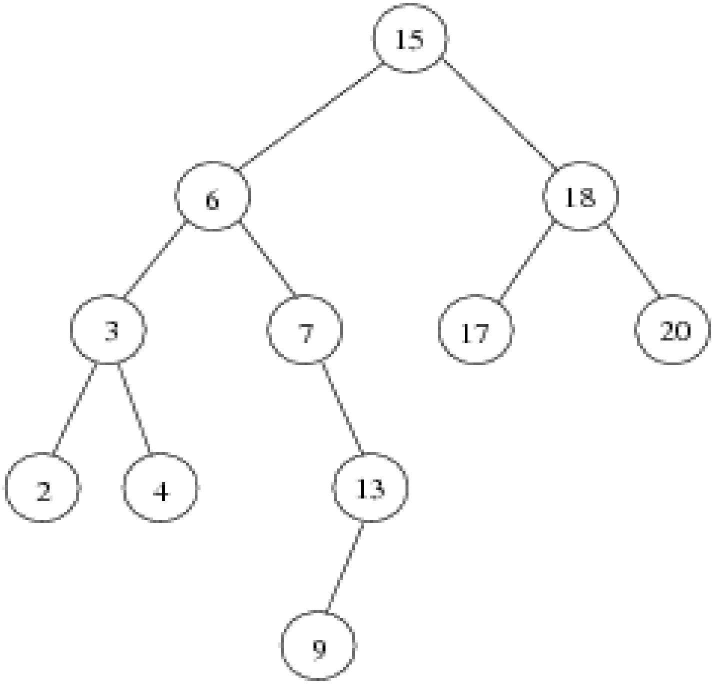


Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Inorder traversal of BST

- Print out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

BST Construction :

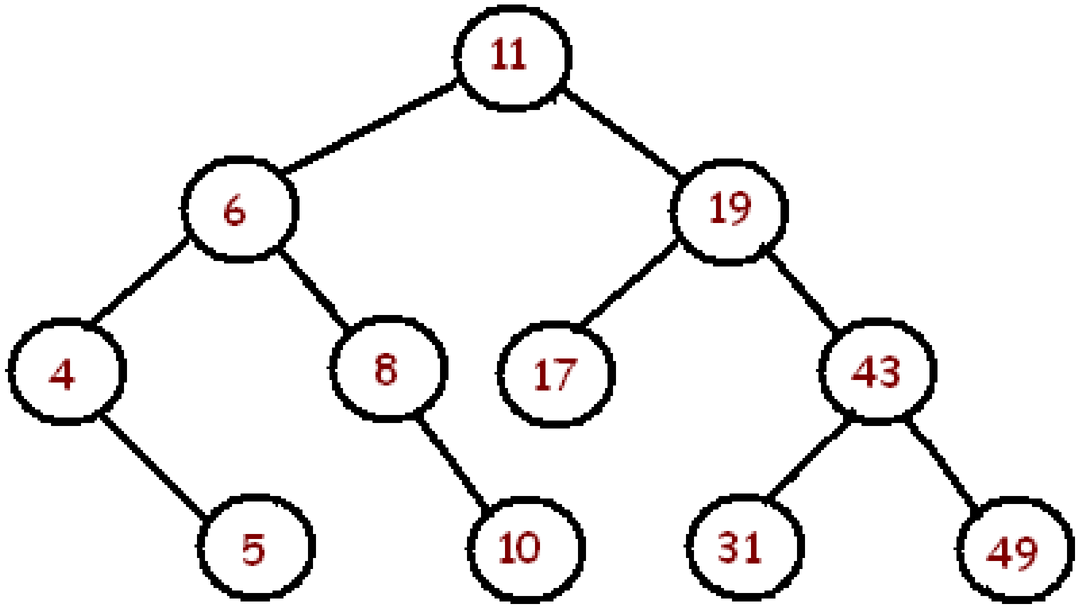
- A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

Given a sequence of numbers:

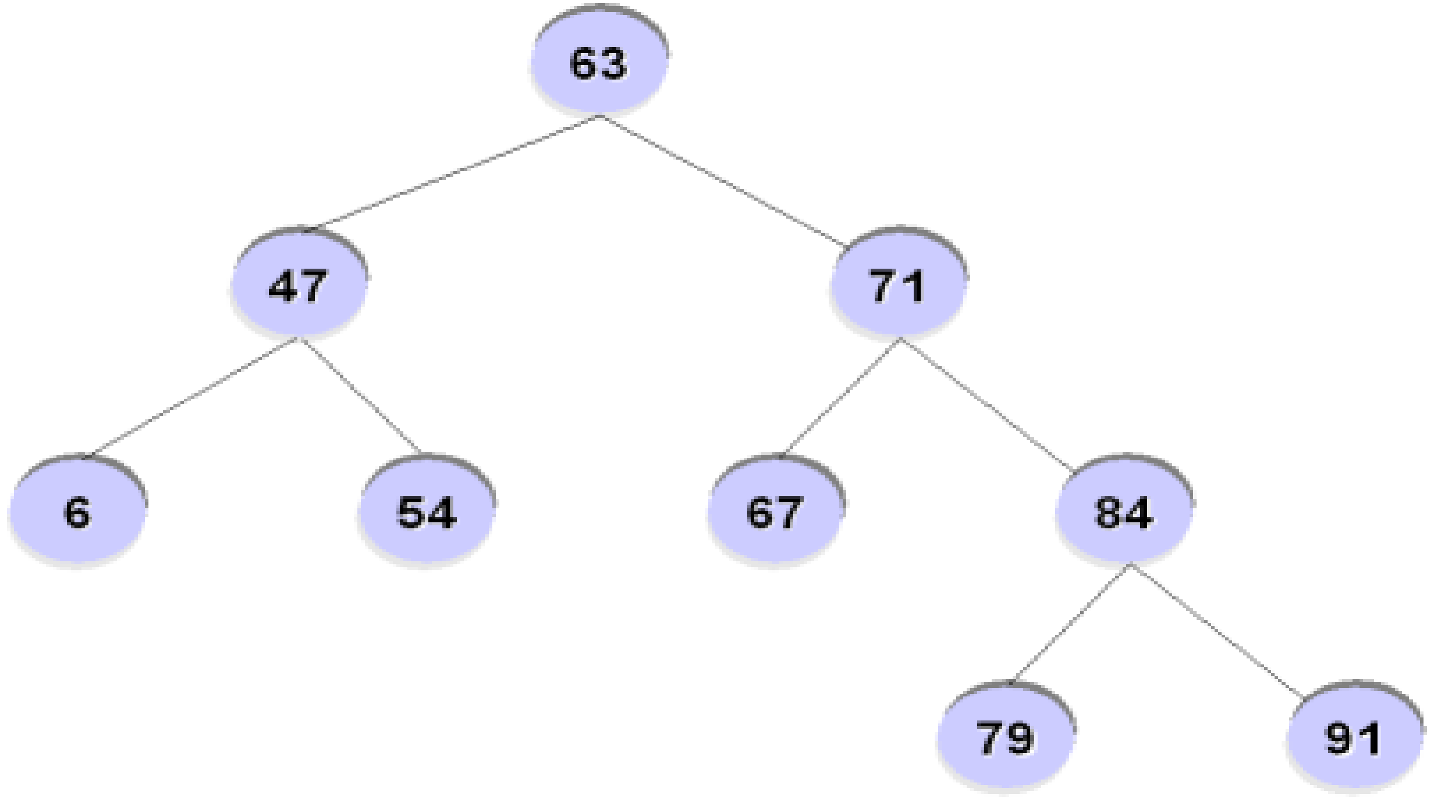
```
11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31
```

- Draw a binary search tree by inserting the above numbers from left to right.

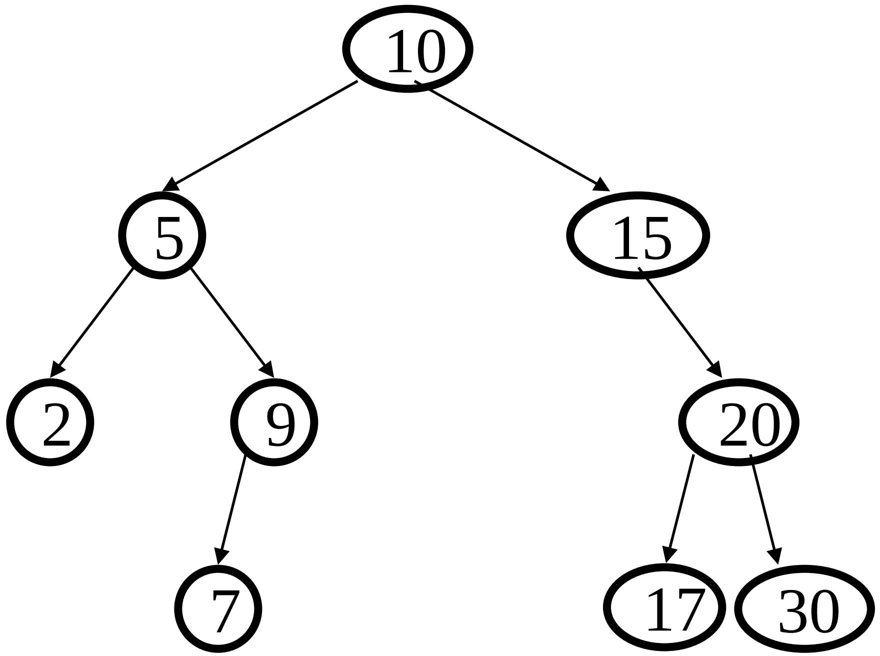


Binary Search Tree : 63,47,6,54,71,67,84,79,91

Example:



Finding a Node




```
Node find(x, Node root)
{
  if (root == NULL)
    return root;
  else if (x < root.key)
    return find(x,root.left);
  else if (x > root.key)
    return find(x, root.right);
  else
    return root;
}
```

Insert :

Concept: proceed down tree as in Find; if new key not found, then insert a new node at last spot traversed

```
void insert(x,  Node root) {
  // Does not work for empty tree – when root is NULL
  if (x < root.key){
      if (root.left == NULL)
            root.left = new Node(x);
      else insert( x, root.left ); }
  else if (x > root.key){
      if (root.right == NULL)
            root.right = new Node(x);
      else insert( x, root.right ); } }
```

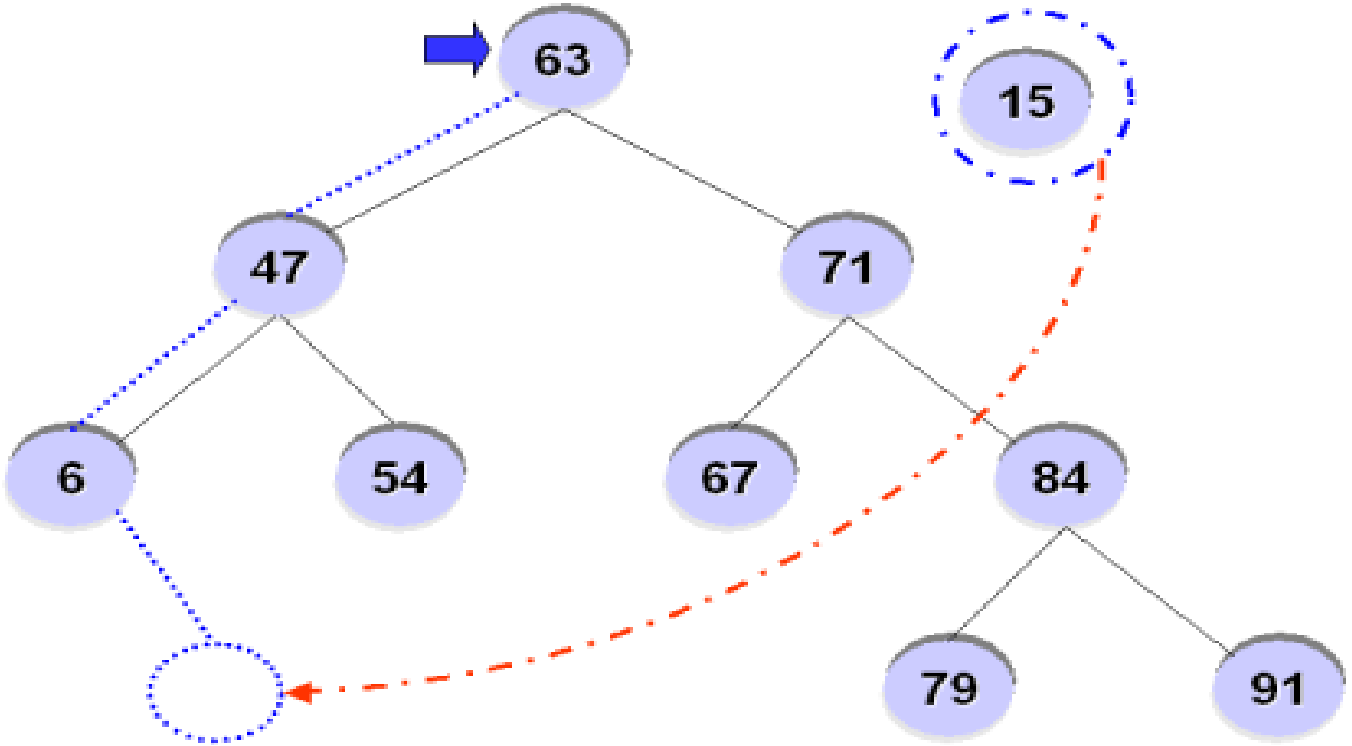
Binary Search Tree :

Insert Algorithm

- If value we want to insert < key of current node,>we have to go to the left subtree
- Otherwise we have to go to the right subtree
- If the current node is empty (not existing) create a node with the value we are inserting and place it here.

Binary Search Tree

For example, inserting '15' into the BST?



Binary Search Tree

Delete Algorithm

How do we delete a node form BST?

Similar to the insert function, after deletion of a node, the property of the BST must be maintained.

Binary Search Tree

There are 3 possible cases

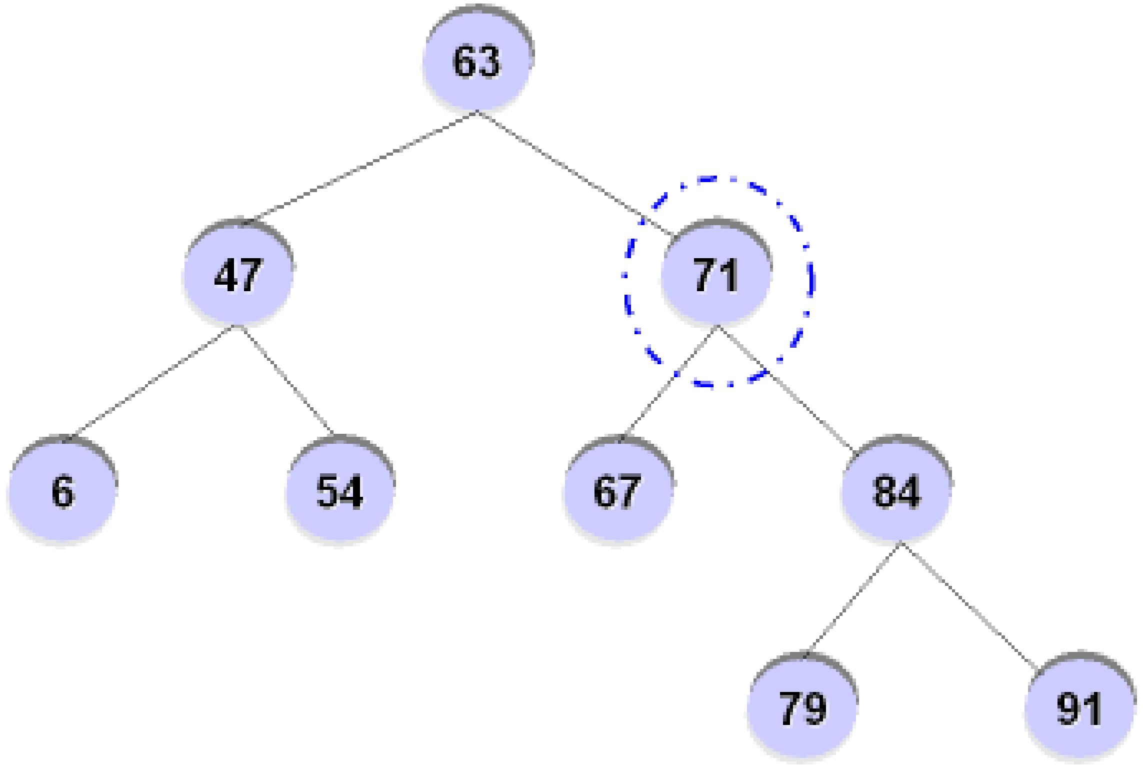
- Node to be deleted has no children

  → **We just delete the node.**
- Node to be deleted has only one child

  → **Replace the node with its child and make the parent of the deleted node to be a parent of the child of the deleted node**
- Node to be deleted has two children
- Next page

Binary Search Tree

Node to be deleted has two children



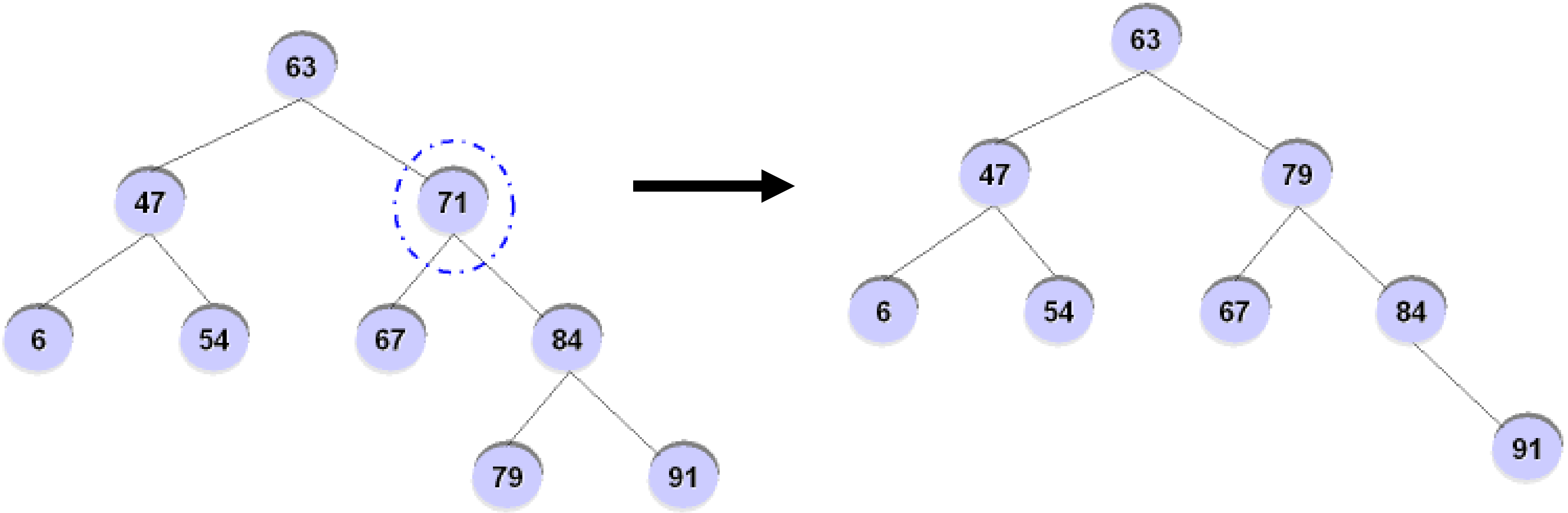
Binary Search Tree

Node to be deleted has two children

Steps:

- **Find minimum value of right subtree**
- **Delete minimum node of right subtree but keep its value**
- **Replace the value of the node to be deleted by the minimum value whose node was deleted earlier.**

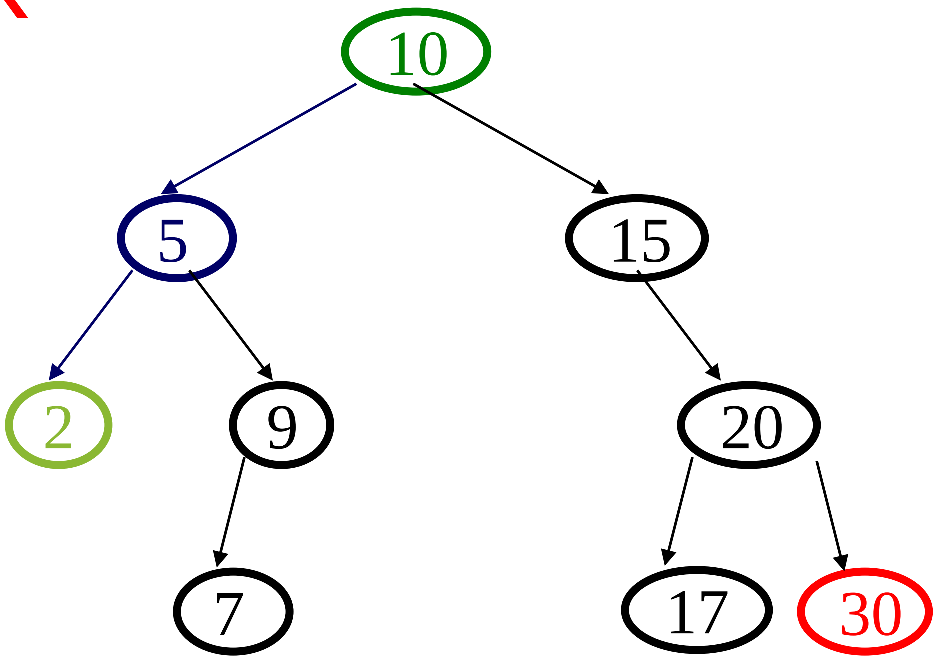
Binary Search Tree



FindMin/FindMax

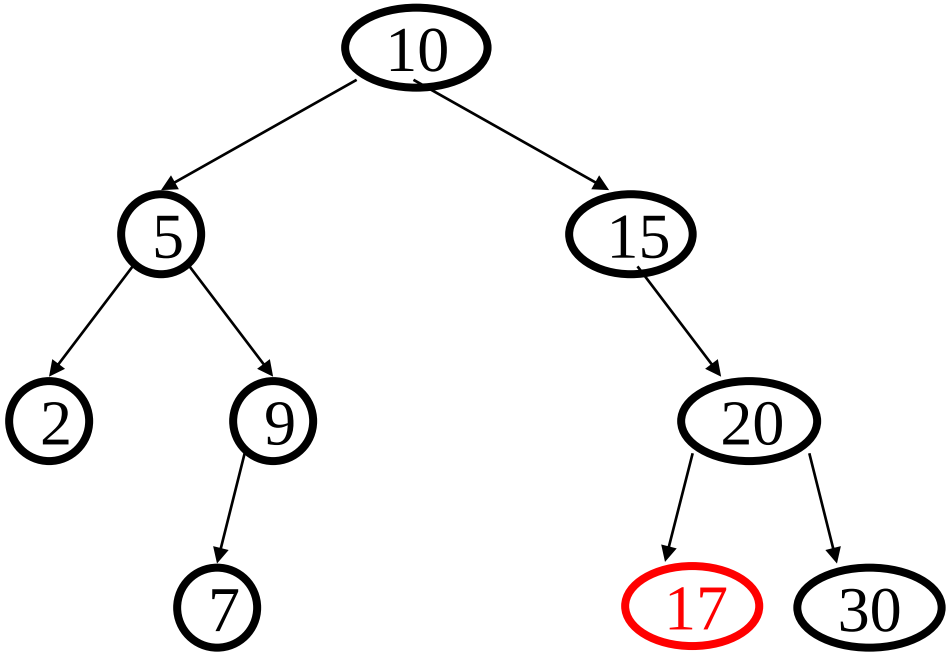
- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```
Node min(Node root)
{
  if (root.left == NULL)
    return root;
  else
    return min(root.left);
}
```



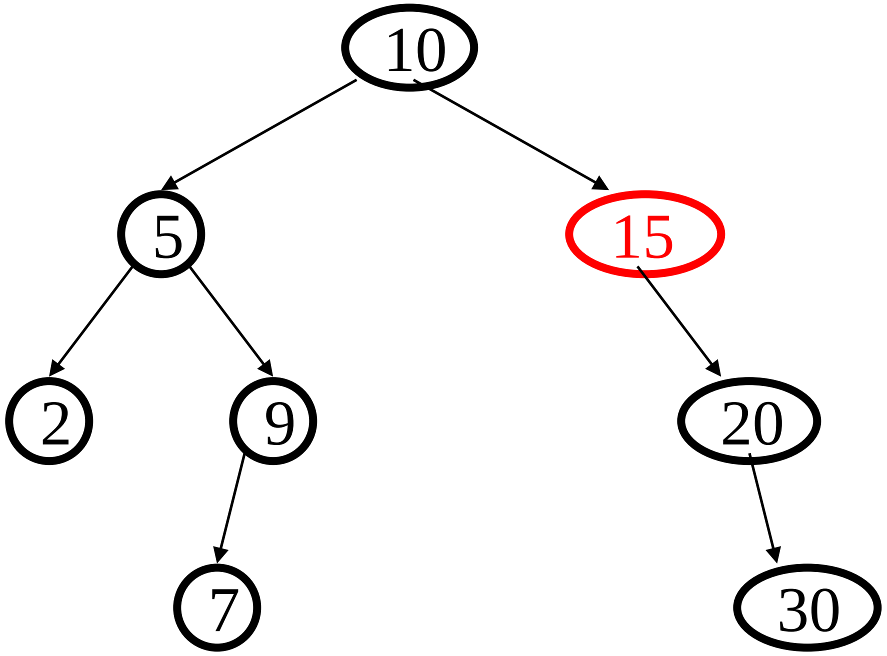
Deletion - Leaf Case

Delete(17)



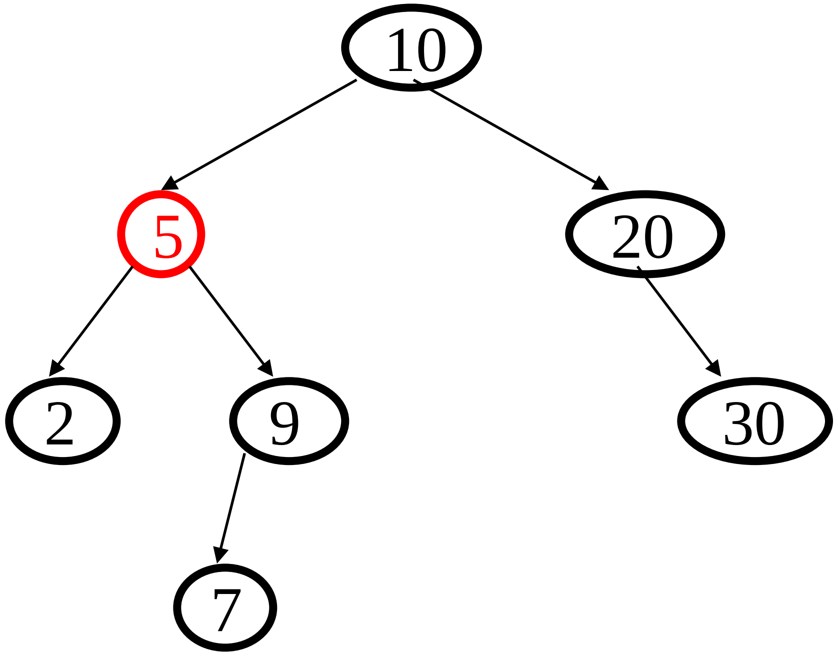
Deletion - One Child Case

Delete(15)



Deletion - Two Child Case

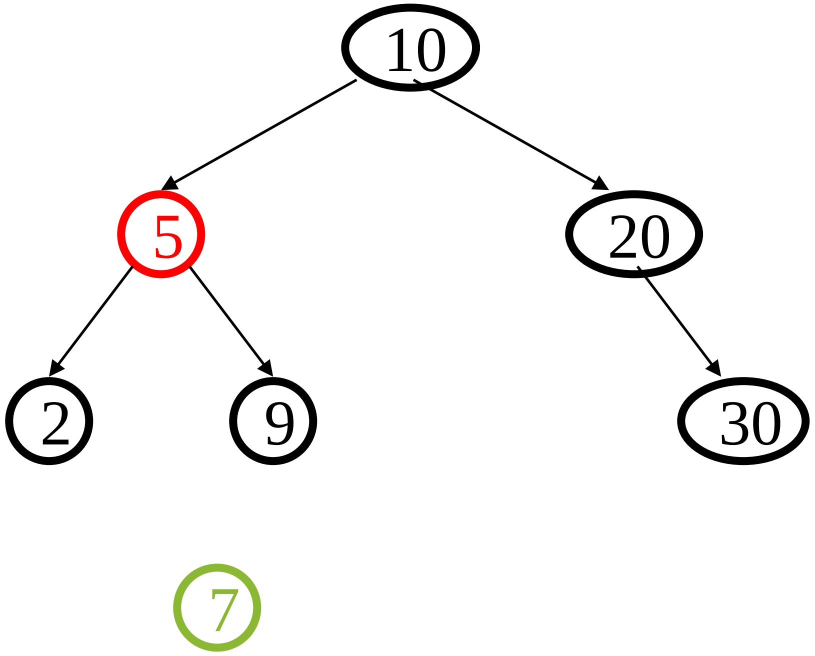
Delete(5)



replace node with value guaranteed to be between the left and right subtrees: the successor

Deletion - Two Child Case

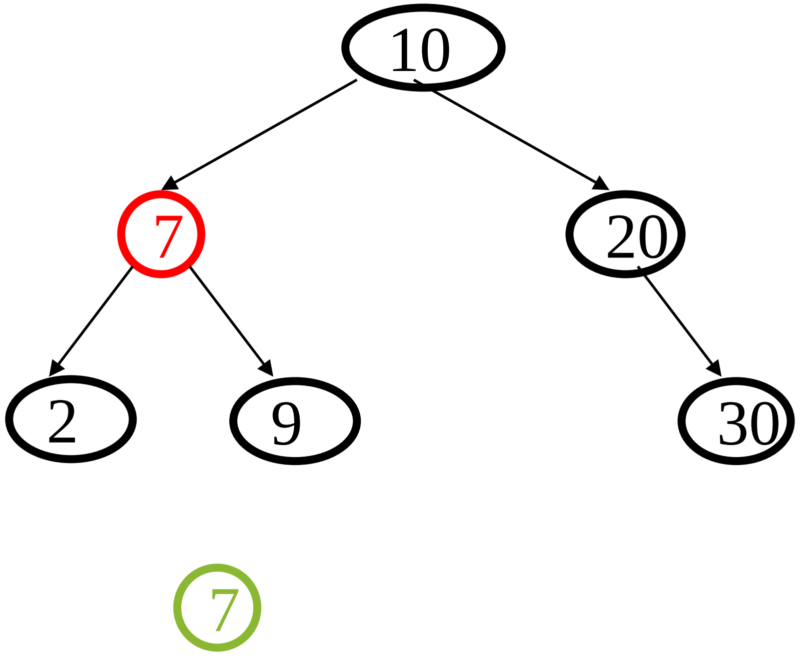
Delete(5)



always easy to delete the successor – always has either 0 or 1 children!

Deletion - Two Child Case

Delete(5)



Finally copy data value from deleted successor into original node

Method to search in BST:

```
static boolean search(node r,int key)
{
    if(r==null)
        return false;
    else if(key==r.data)
        return true;
    else if(key<r.data)
        return search(r.left,key);
    else
        return search(r.right,key);
}
```

Insert in BST:

```
static node insert(node r,int key)
{
	if(r==null)
	{
		r=new node();
		r.data=key;
		r.left=r.right=null;
		return r;
	}
	else if(key<r.data)
		r.left=insert(r.left,key);
	else
		r.right=insert(r.right,key);
	return r;
}
```

Maximum element in BST:

```
static int max(node r)
{
    while(r.right!=null)
        r=r.right;
    return r.data;
}
```

Preorder Traversal in BST:

```
static void preorder(node root)
{
    if(root!=null)
    {

        System.out.print(root.data+"  ");
        preorder(root.left);
        preorder(root.right);
    }
}
```

Inorder Traversal of BST:

```
static void inorder(node root)
{
    if(root!=null)
    {
        inorder(root.left);
        System.out.print(root.data+"  ");
        inorder(root.right);
    }
}
```

Postorder Traversal in BST:

```
static void postorder(node root)
{
    if(root!=null)
    {
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.data+"  ");
    }
}
```

Delete in BST:

```
static node Delete(node r,int key)
{
    if(r==null)
        return null;
    else if(key<r.data)
        r.left=Delete(r.left,key);
    else if(key>r.data)
        r.right=Delete(r.right,key);
    else
    {
        if(r.left==null)
            return r.right;
        else if(r.right==null)
            return r.left;
        else
        {
            r.data=max(r.left);
            r.left=Delete(r.left,r.data);
        }
    }
    return r;
}
```

Main applications of trees include:

1. Manipulate hierarchical data.

2. Make information easy to search (see tree traversal).

3. Manipulate sorted lists of data.

4. As a workflow for compositing digital images for visual effects.

5. Router algorithms

6. Form of a multi-stage decision-making (see business chess).