# CS-218
# DATA STRUCTURE

Dr. Hashim Yasin

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

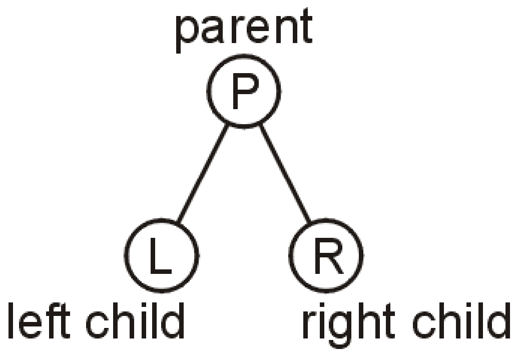
# BINARY TREE

# Binary Trees

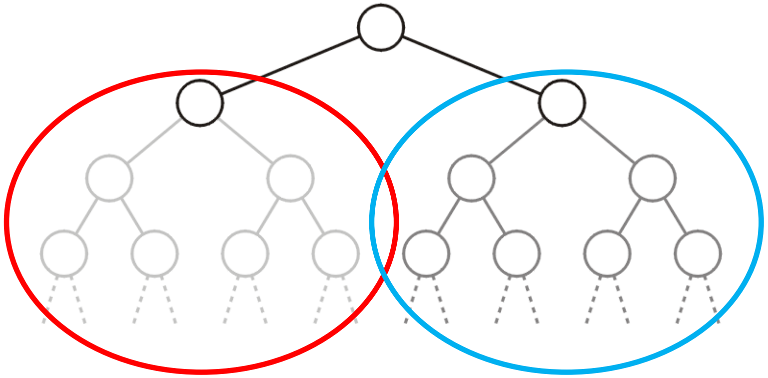


## Binary Tree

- In a binary tree, each node has at most two children
  - Allows to label the children as left and right




- Likewise, the two sub-trees are referred as
  - Left sub-tree
  - Right sub-tree



# Binary Tree … Implementation



- Binary Tree … Implementation
  - Array based implementation
  - Linked List based implementation

# Binary Tree … Implementation



- Objects: any type of objects can be stored in a tree data structure

- Accessor methods
  - root() – return the root of the tree
  - parent(p) – return the parent of a node
  - children(p) – returns the children of a node

# Binary Tree … Implementation



- query methods
  - size() – returns the number of nodes in the tree
  - isEmpty() - returns true if the tree is empty
  - elements() – returns all elements
  - isRoot(p), isInternal(p), isExternal(p)
- other methods
  - Tree traversal, Node addition/deletion, create/destroy

# Binary Tree … Implementation

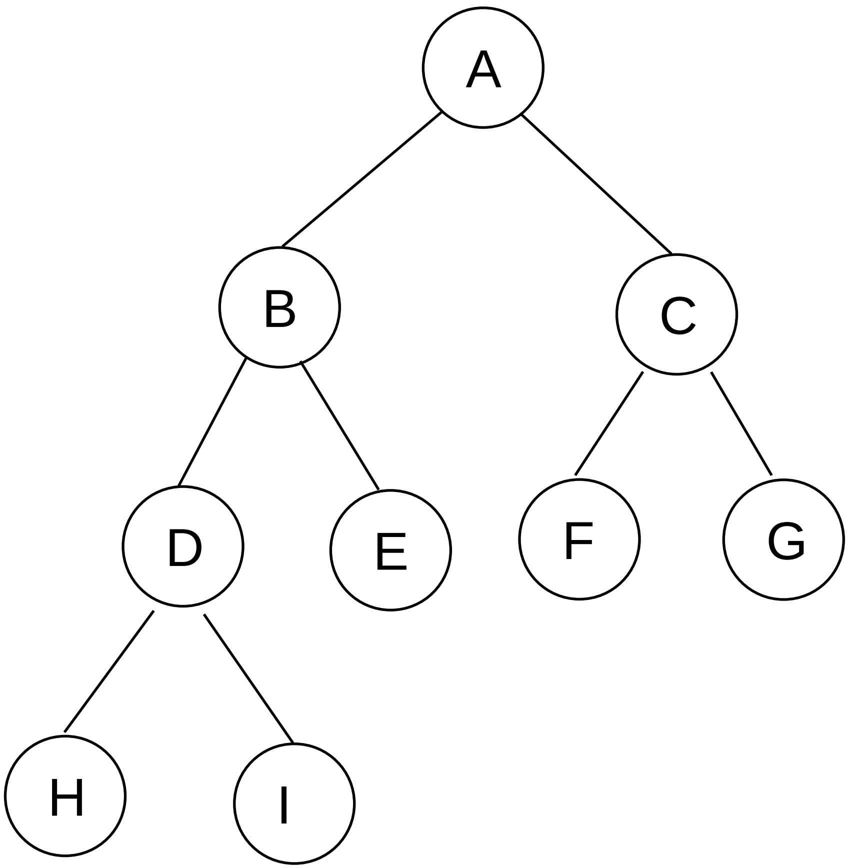


**Array-based Implementation:**

- Value in root node stored first, followed by left child, then right child
- Each successive level in the tree ***stored left to right***; unused nodes in tree represented by a bit pattern to indicate nothing stored there
- Children of any given node *n* is stored in cells $2n$ and $2n + 1$ (If array index starts at 1)
- *Storage allocated as for full tree, even if many nodes empty*
- For a tree of depth *h* we need array of $2^{h+1}-1$ cells

# Binary Tree … Implementation

**Array-based Implementation:**

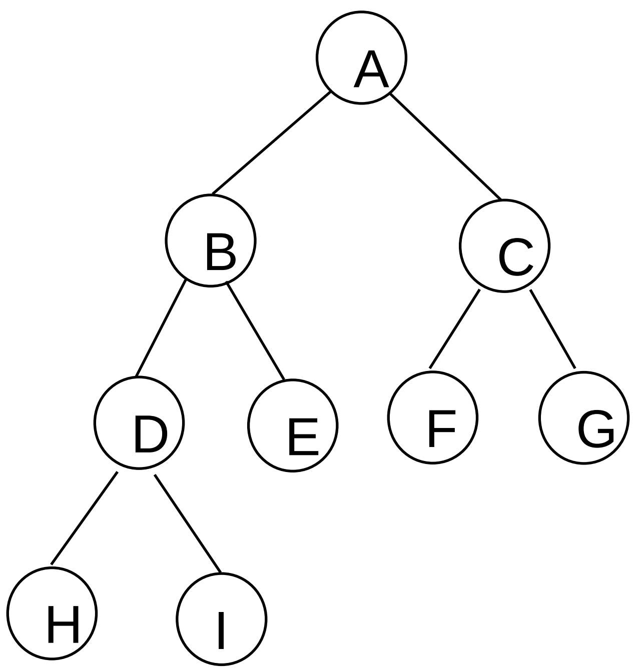


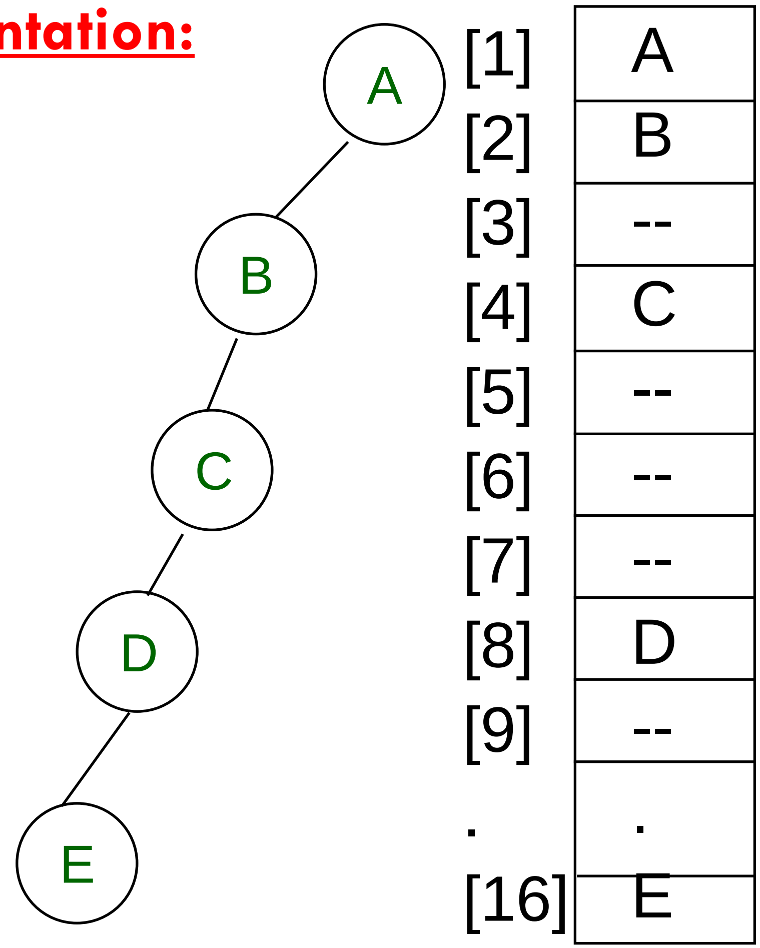

| | |
|---|---|
| [1] | A |
| [2] | B |
| [3] | C |
| [4] | D |
| [5] | E |
| [6] | F |
| [7] | G |
| [8] | H |
| [9] | I |

# Binary Tree … Implementation



**Array-based Implementation:**



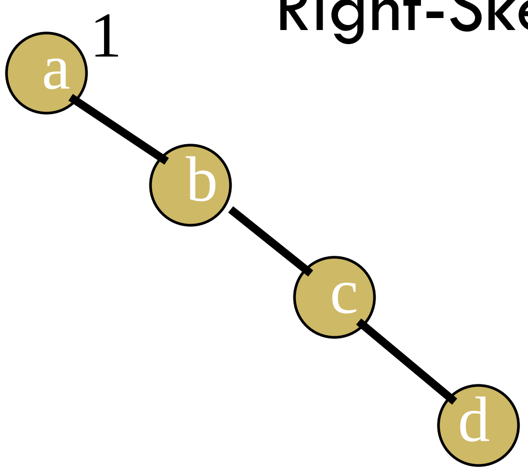



# Binary Tree … Implementation



**Array-based Implementation:**

Right-Skewed Binary Tree




tree[]

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | - | b | - | - | - | c | - | - | - | - | - | - | - | d |

# Binary Tree … Implementation



## Linked List-based Implementation:

- Implementation of Binary tree can be done using dynamic creation of node having data and two pointers.

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

# Binary Tree … Implementation



- A binary tree is composed of zero or more nodes
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be null), and
  - A reference or pointer to a right child (may be null)
- A binary tree may be *empty* (contain no nodes)

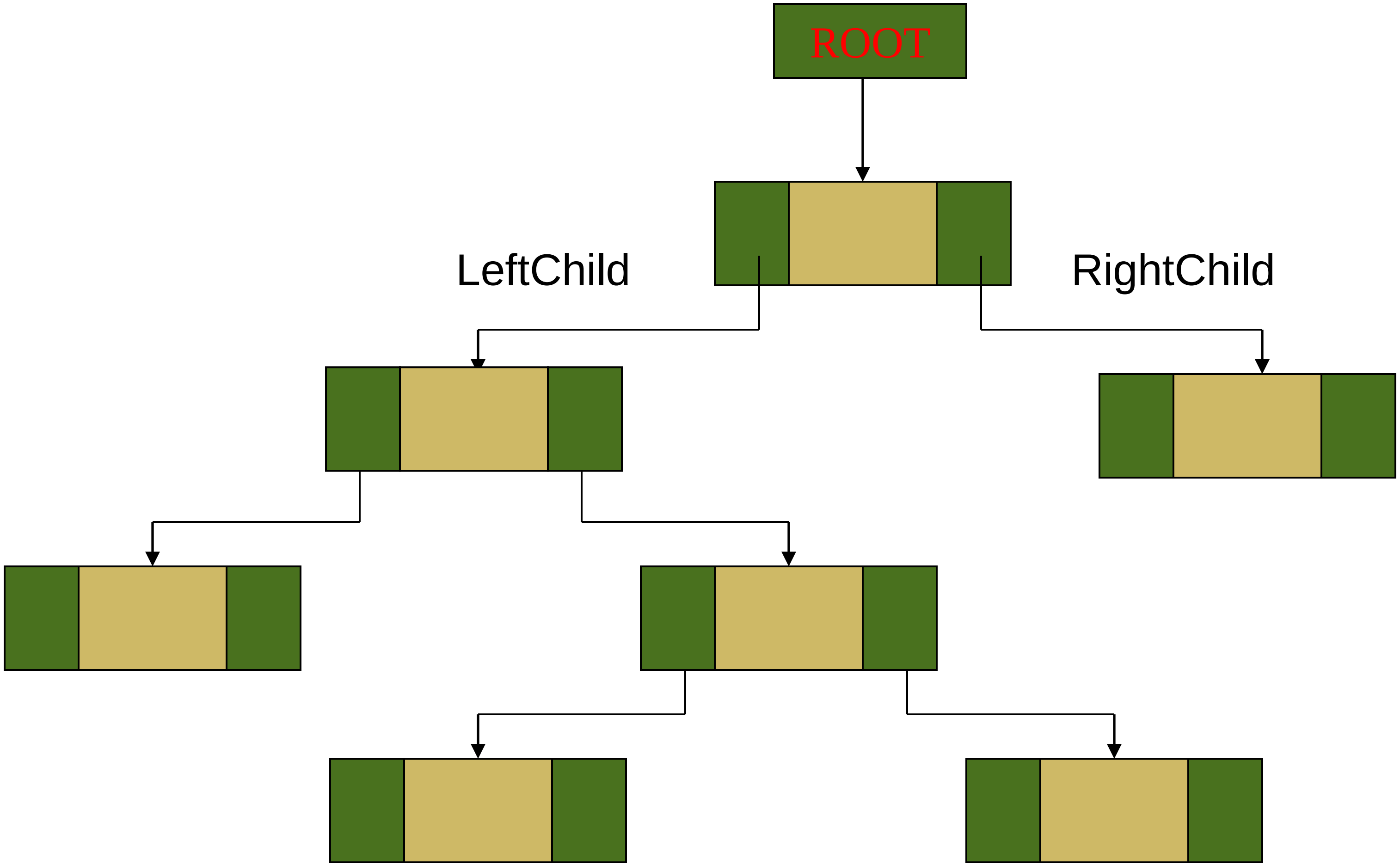
# Binary Tree … Implementation



- If not empty, a binary tree has a root node
  - Every node in the binary tree is reachable from the root node by a ***unique*** path
- A node with neither a left child nor a right child is called a **leaf**
  - In some binary trees, only the leaves contain a value

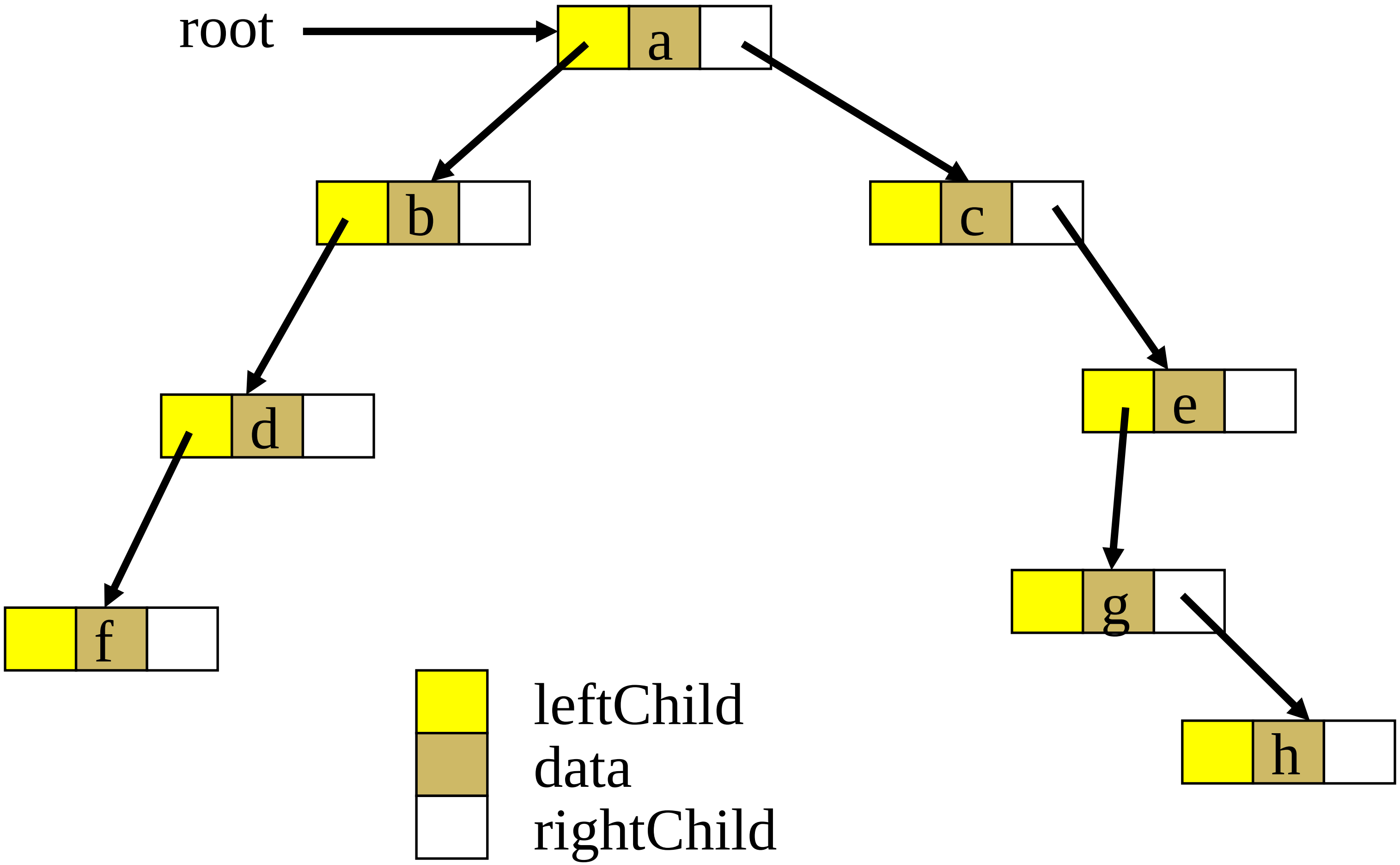
# Binary Tree … Implementation





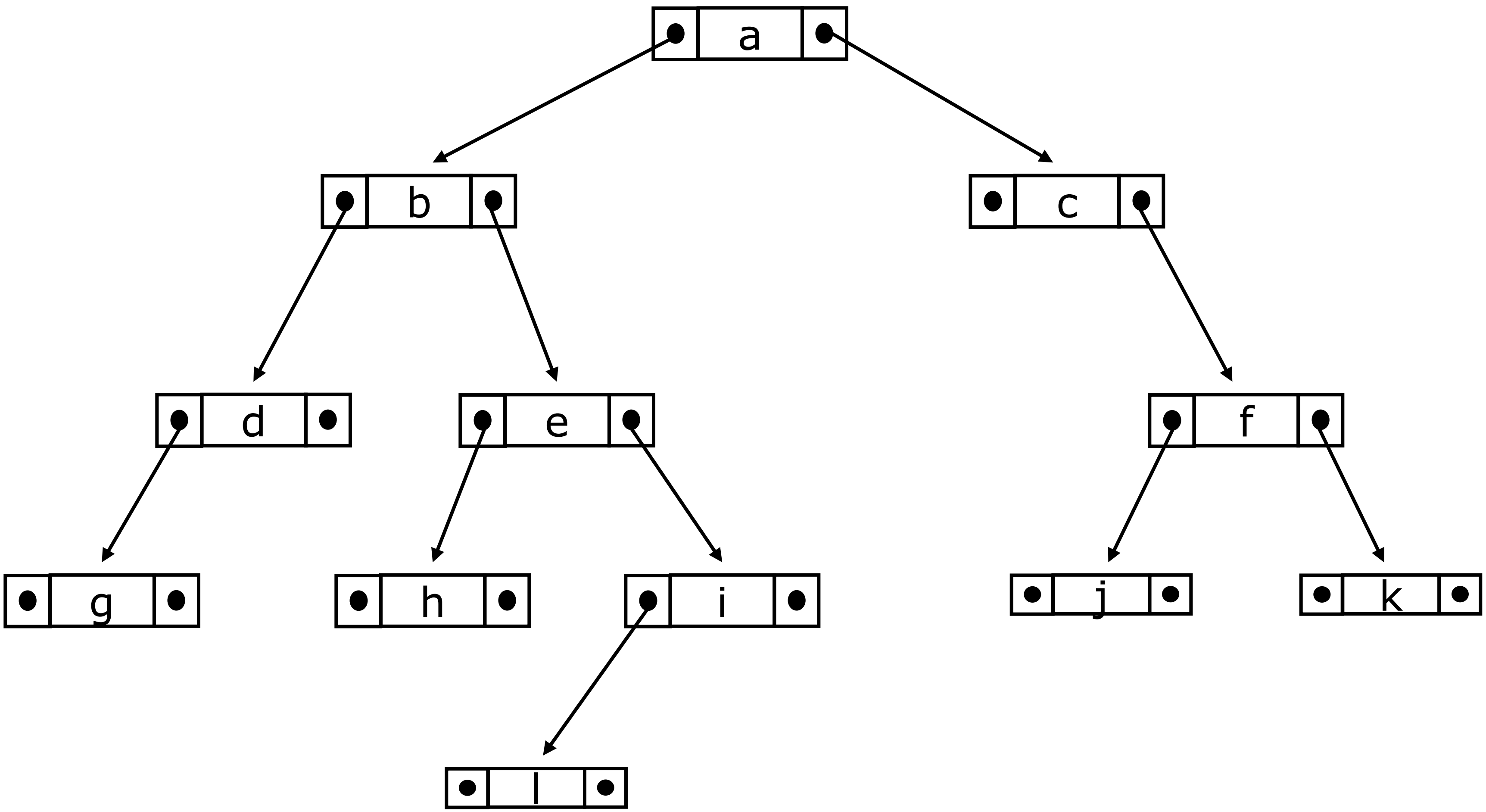
# Binary Tree ... Implementation





# Binary Tree … Implementation





# Binary Tree ... Implementation



## Implementation: with Linked List

- **Each node** in the tree consists of:
  - The data, or value contained in the element
  - A left child pointer (pointer to first child)
  - A right child pointer (pointer to second child)
- A **root pointer** points to the root node
  - Follow pointers to find every other element in the tree
- *Add and remove nodes by manipulating pointers*
- **Leaf nodes** have child pointers set to null

# Binary Tree … Implementation



- **Left(node):** Gives index/pointer of left child
- **Right(node):** Gives index/pointer of right child
- **Parent(node):** Returns index/pointer of parent
- **Brother(node):** Returns index/pointer of brother
- **Root:** Gives index/pointer of root node
- **Info(Node):** Data/Info stored at node
- **IsLeft(node):** Is node left child? Yes/No
- **IsRight(node):** Is node right child? Yes/No

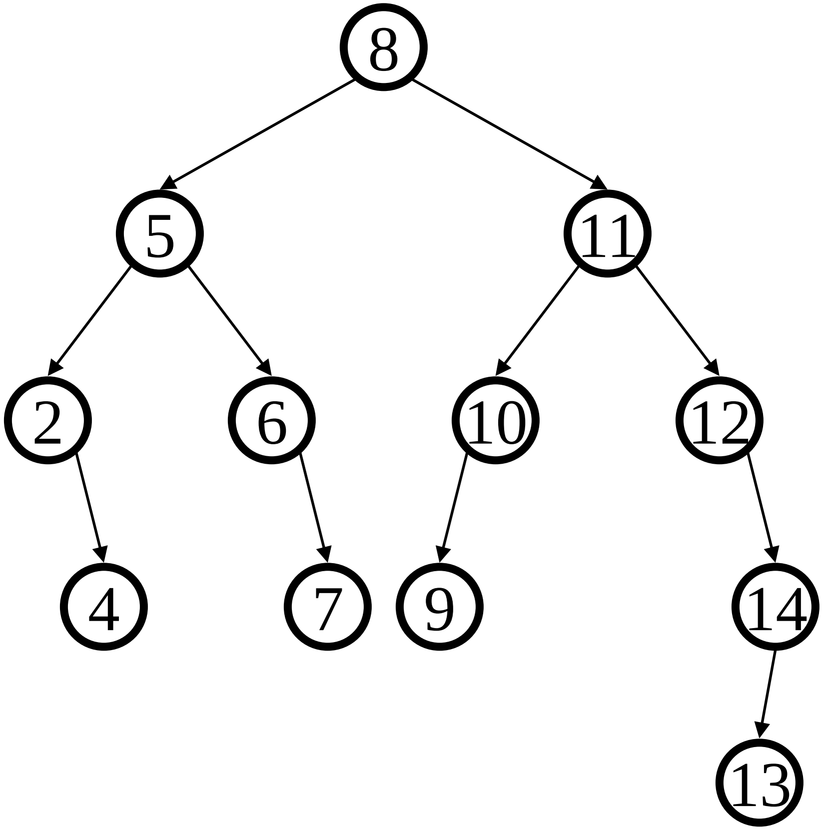
# BINARY SEARCH TREE

# Binary Search Trees (BSTs)



- **Binary Search Tree** **Property:**

  The value stored at a node is *greater than* the value stored at its left child and *less than* the value stored at its right child

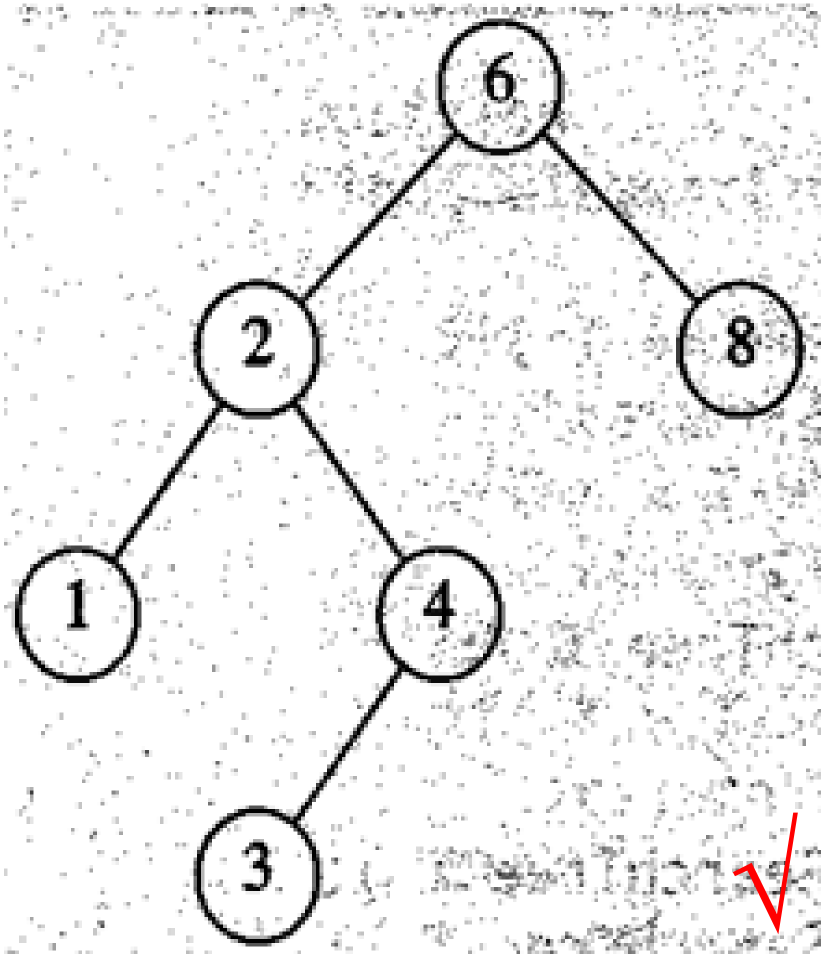


# Binary Search Trees (BSTs)



- **Binary Search Tree** **Property:**

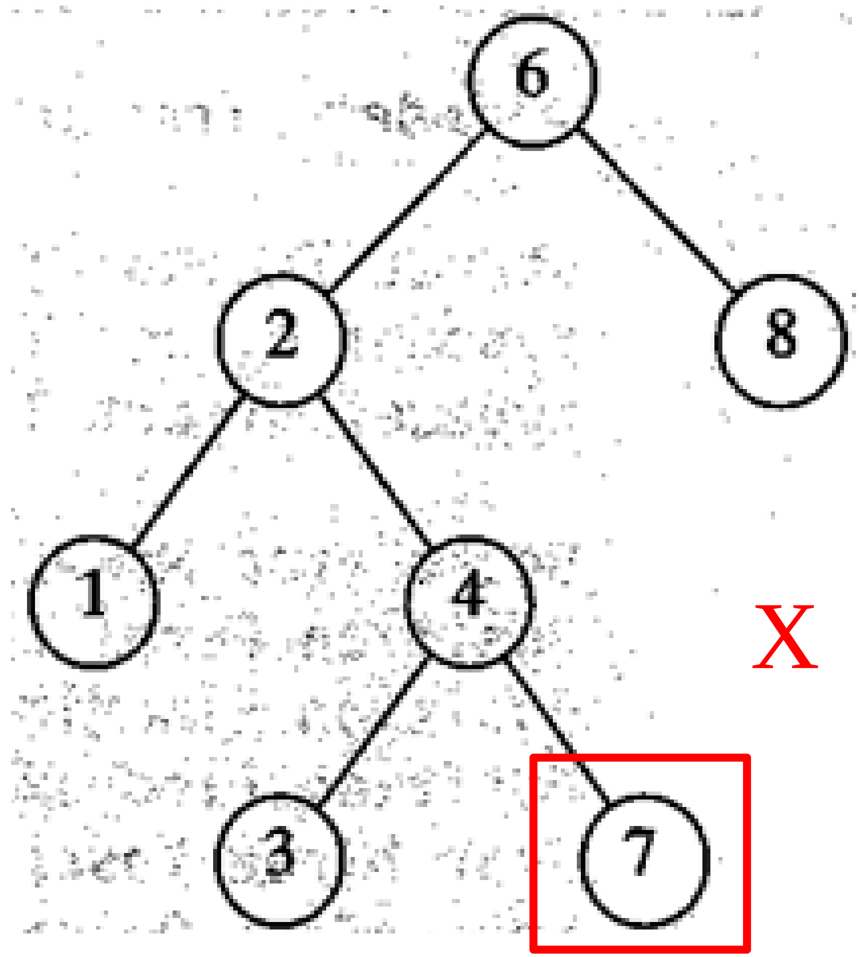
  The value stored at a node is *greater than* the value stored at its left child and *less than* the value stored at its right child



# Binary Search Trees (BSTs)



- **Binary Search Tree** **Property:**
  The value stored at a node is *greater than* the value stored at its left child and *less than* the value stored at its right child

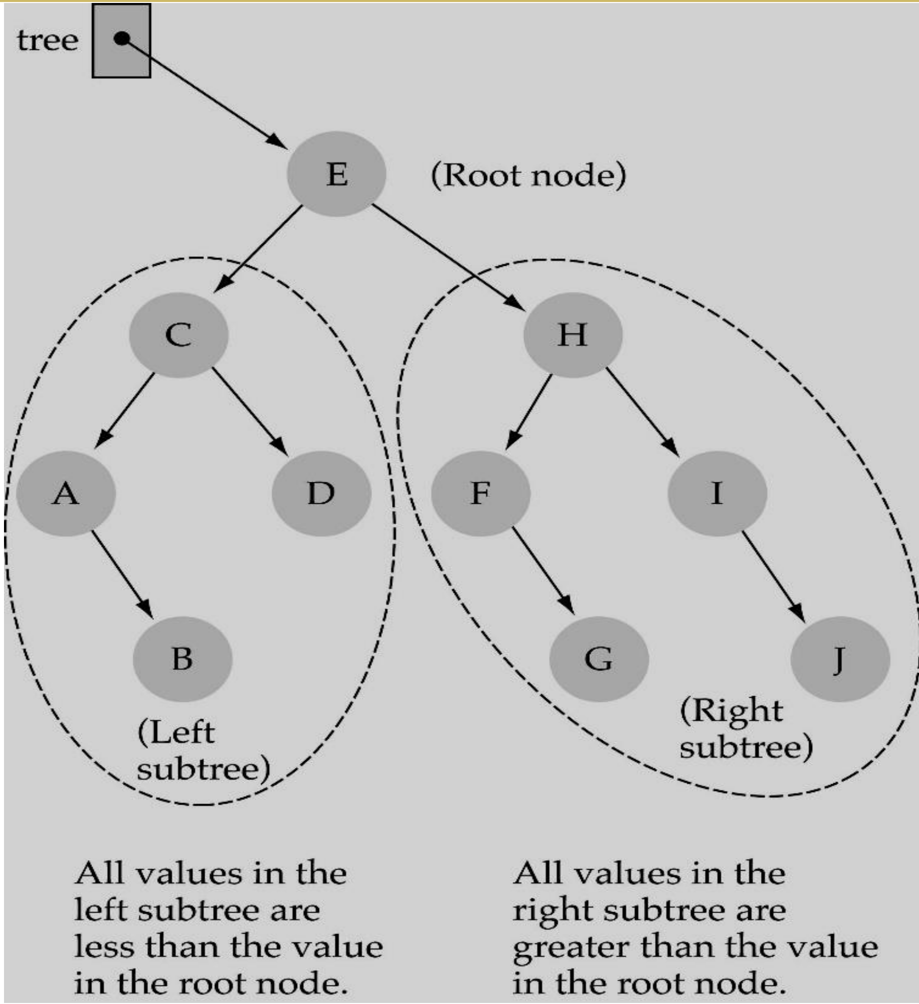


# Binary Search Trees (BSTs)



- **Binary Search Tree Property:**

  The value stored at a node is *greater than* the value stored at its left child and *less than* the value stored at its right child



# Binary Search Trees (BSTs)



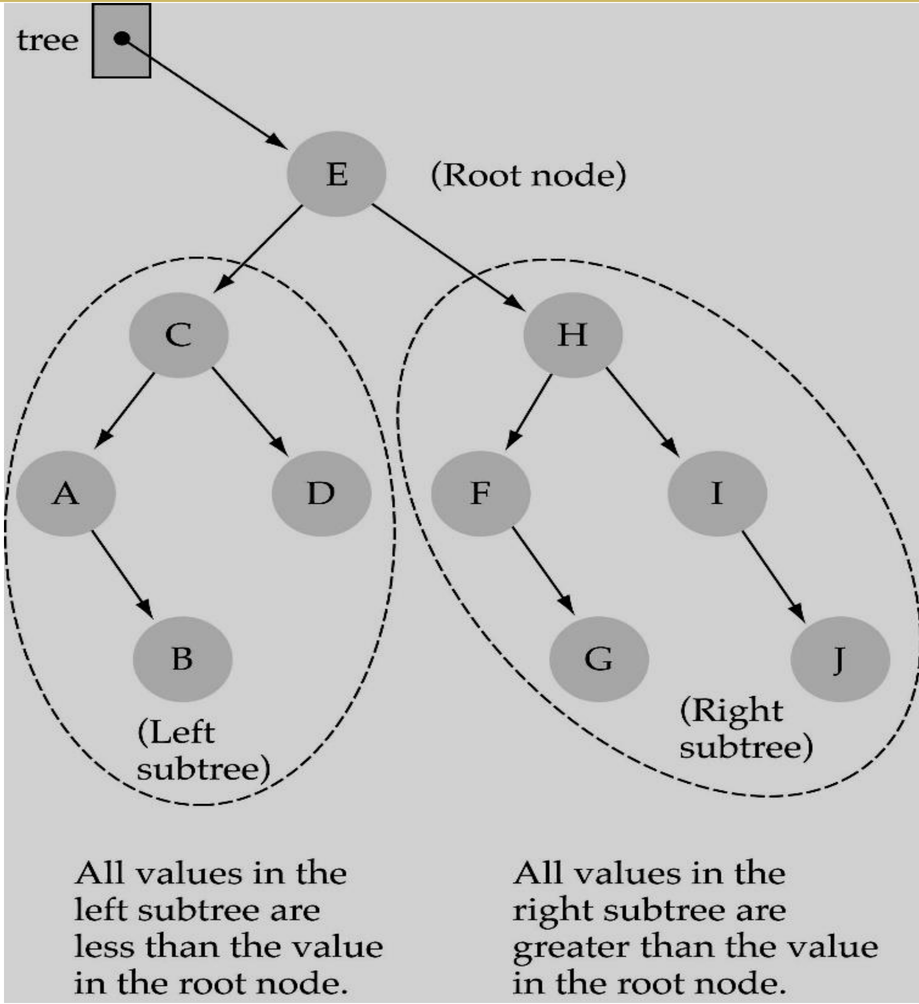
- **Binary Search Tree** **Property:**

Where is the smallest element?

Ans: leftmost element

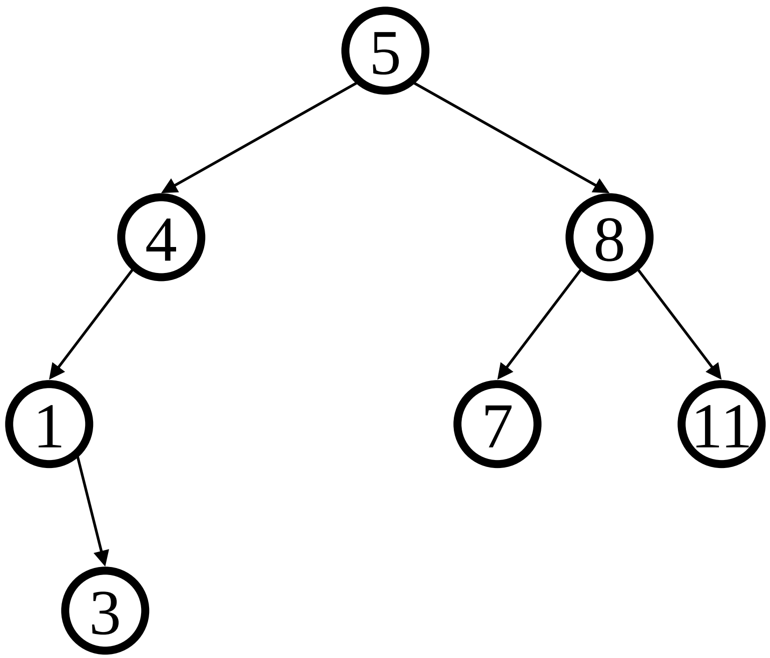
Where is the largest element?

Ans: rightmost element

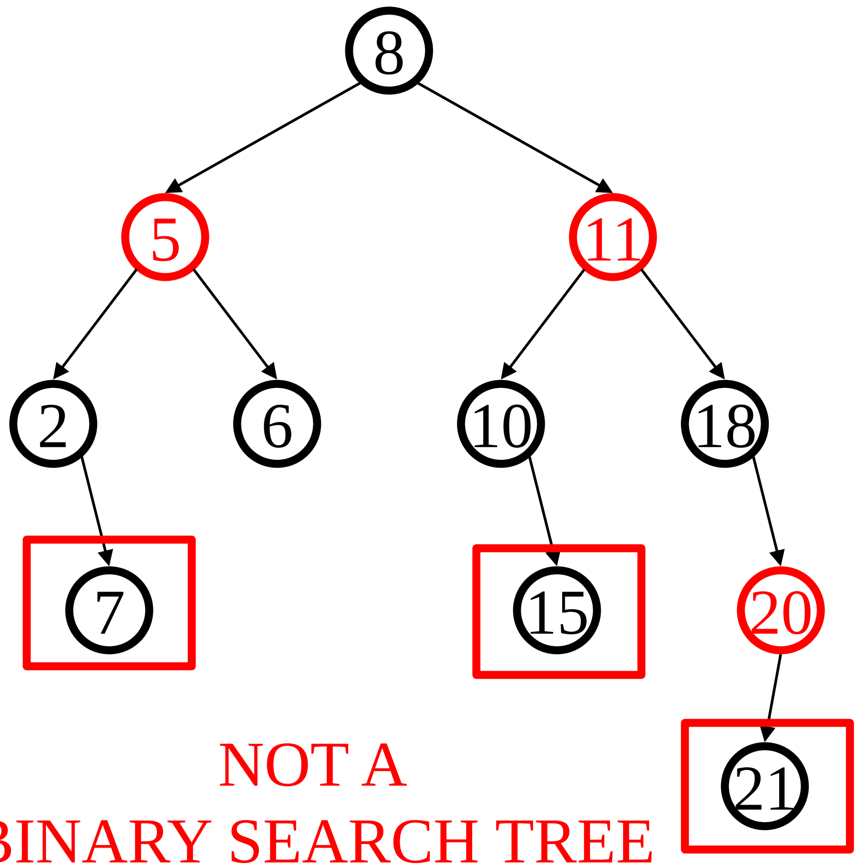


# Binary Search Trees (BSTs)






BINARY SEARCH TREE




NOT A
BINARY SEARCH TREE

# Binary Search Trees (BSTs)



Binary tree property

- each node has $\leq$ 2 children
- result:
  - storage is small
  - operations are simple
  - average depth is small

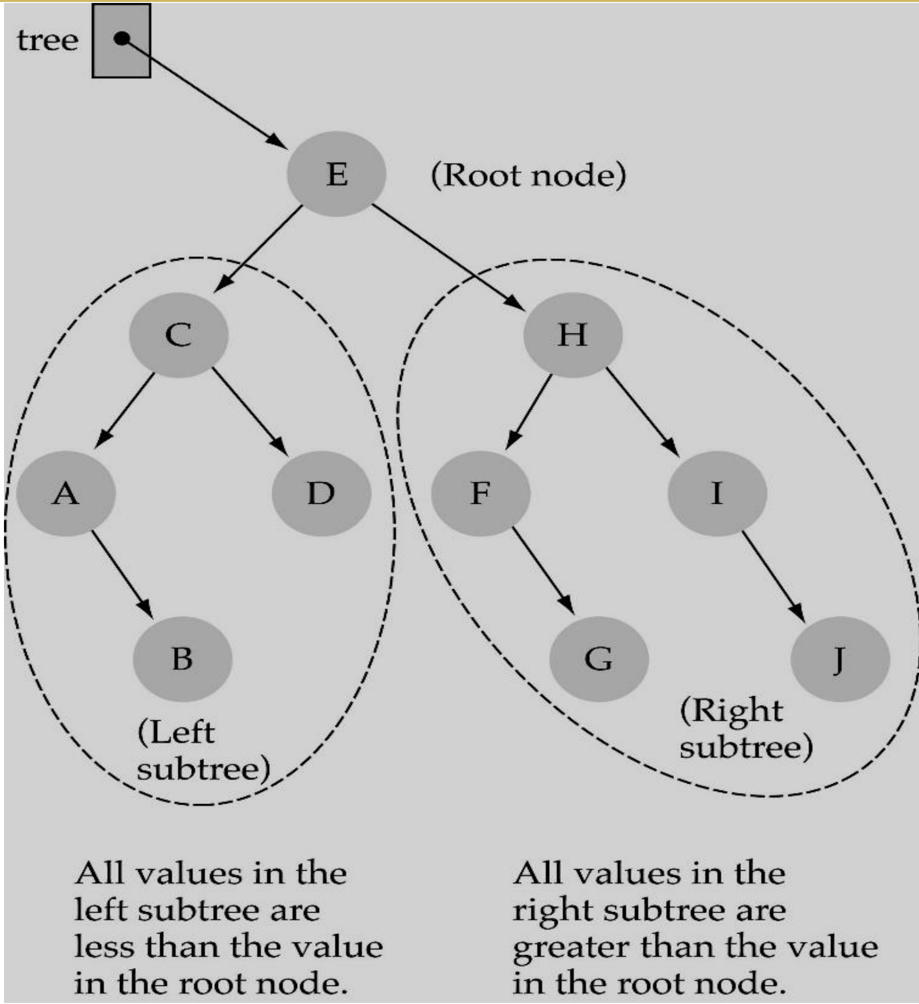
Search tree property

- 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
  - Insert/delete by changing links

# Searching in BST



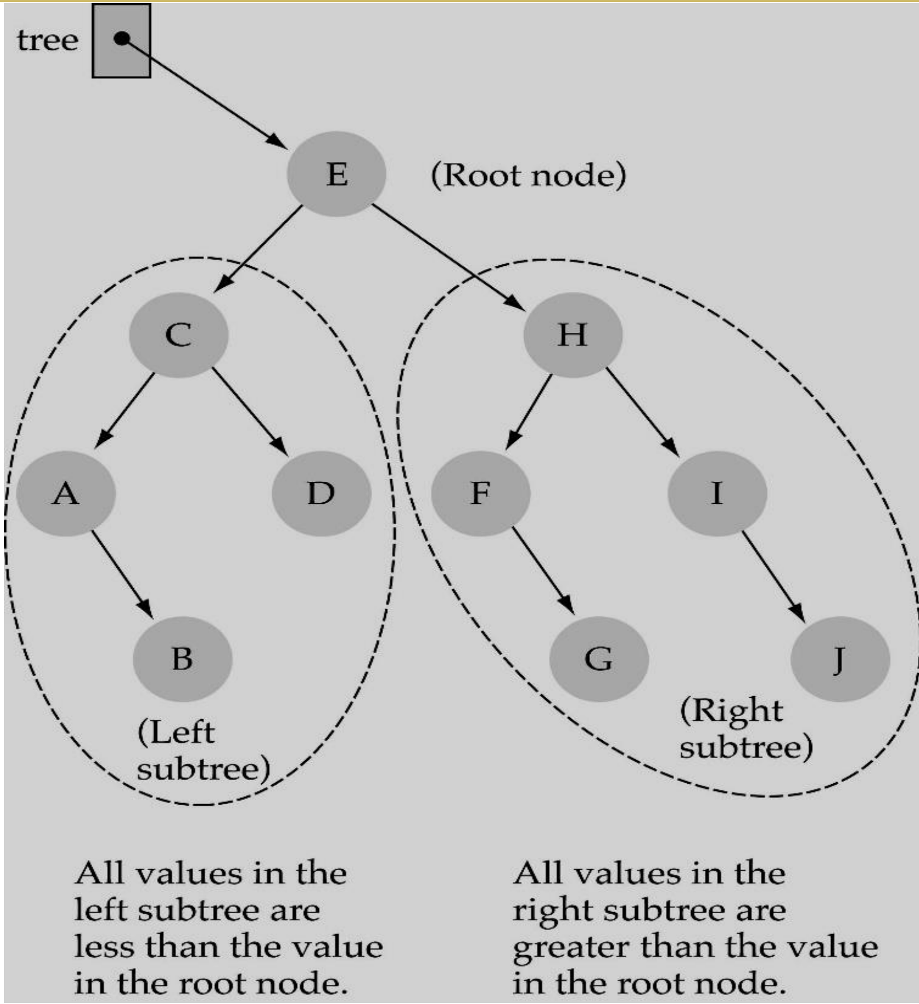
1. Start at the root
2. Compare the value of the item you are searching for with the value stored at the root
3. If the values are equal, then *item found*; otherwise, if it is a leaf node, then *not found*



# Searching in BST



4. If it is less than the value stored at the root, then search the left subtree
5. If it is greater than the value stored at the root, then search the right subtree
6. Repeat steps 2-6 for the root of the subtree chosen in the previous step 4 or 5

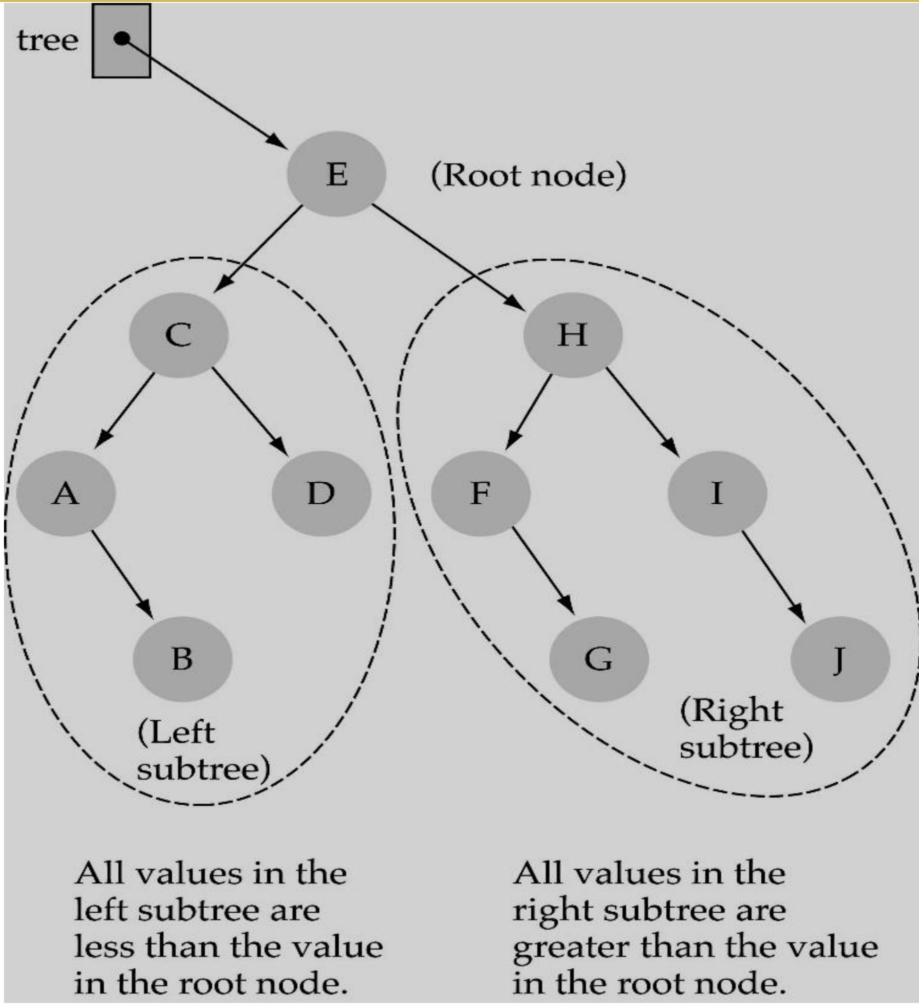


# Searching in BST



Is this better than searching a linked list?

Yes !! ---> O(logN)



# Why BST



## Array

- Searching in the Array  O(n)
- Insertion O(1)
- Remove O(n)

## Linked List

- Searching in the Linked List  O(n)
- Insertion O(1)
- Remove O(n)

## BST

- Searching in the BST  O(log n)
- Insertion O(log n)
- Remove O(log n)

# BST Operations



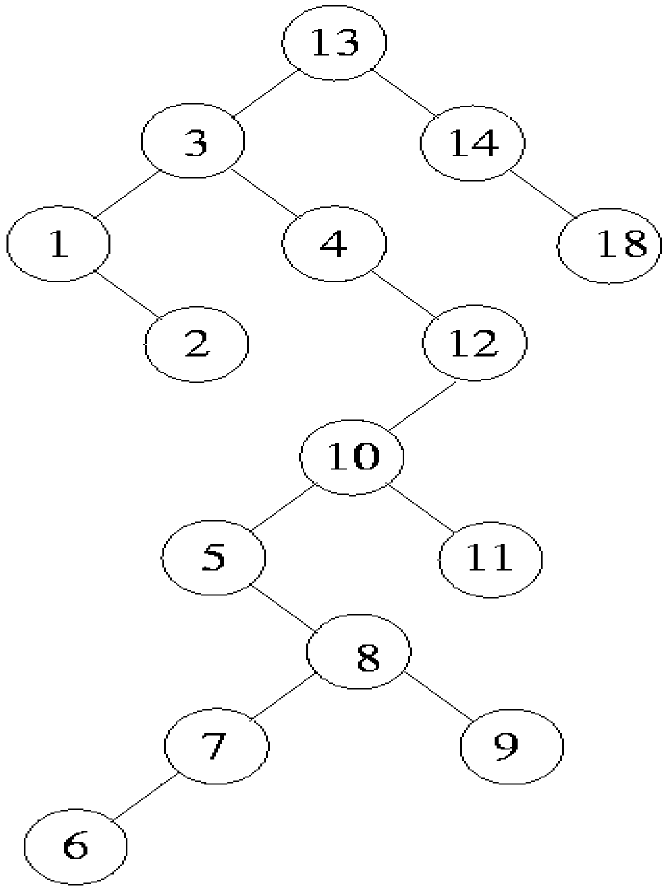
- There are <u>many operations,</u> one can perform on a *binary search tree.*

- **Creating** a binary search tree
- **Inserting** a node into a binary search tree
- **Finding** a node in a binary search tree
- **Deleting** a node in a binary search tree.

# BST



13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



# Implementation



- The basis of our binary tree node is the following `struct` declaration:

```
struct TreeNode
   {
        int value;
        TreeNode *left;
        TreeNode *right;
   };
```

```
class IntBinaryTree{
private:
	struct TreeNode{
		int value;
		TreeNode *left;
		TreeNode *right;
	};

	TreeNode *root;
	void destroySubTree(TreeNode *);
	void deleteNode(int, TreeNode *&);
	void makeDeletion(TreeNode *&);
	void displayInOrder(TreeNode *);
	void displayPreOrder(TreeNode *);
	void displayPostOrder(TreeNode *);

 public:

	IntBinaryTree() { root = NULL; }		// Constructor
	~IntBinaryTree() { destroySubTree(root); }// Destructor
	void insertNode(int);
	bool searchNode(int);
	void remove(int);
	void showNodesInOrder(){	displayInOrder(root); }
	void showNodesPreOrder(){	displayPreOrder(root); }
	void showNodesPostOrder(){	displayPostOrder(root); }
};
```

# Implementation



- The *root pointer* is the pointer to the binary tree. This is similar to the *head* pointer in a linked list.

- The *root pointer* will point to the first node in the tree, or to NULL (if the tree is empty).

- It is initialized in the constructor.

- The destructor calls ***destroySubTree***, a private member function, that **recursively** deletes **all** the nodes in the tree.

# Insertion



- The code to insert a new value in the tree is fairly straightforward.

- First, a new node is allocated, and its *value* member is initialized with the new value.

# Insertion



## Insert a Node

- First, a new node is allocated and its value member is initialized with the new value.
- The left and right child pointers are set to NULL, because all nodes must be inserted as leaf nodes.
- Next, we determine if the tree is empty. If so, we simply make root point to it, and there is nothing else to be done.
- But, if there are nodes in the tree, we must find the new node's proper insertion point.

# Insertion



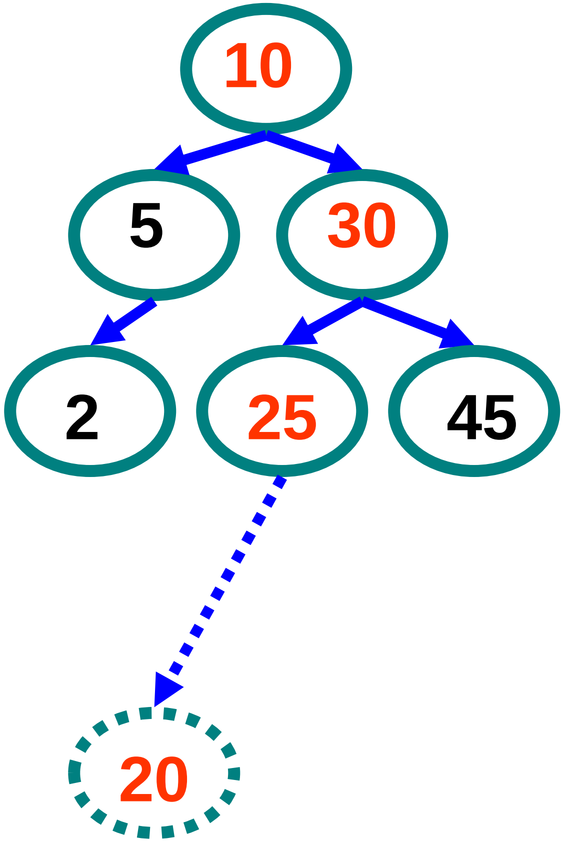
## Insert a Node

- If the new value is **less** than the root node's value, we know it will be inserted somewhere in the left subtree. Otherwise, the value will be inserted into the right subtree.
- We simply traverse the subtree, comparing each node along the way with the new node's value, and deciding if we should continue to the left or the right.
- *When we reach a child pointer that is set to NULL, we have found out insertion point.*

# Insertion



**Insert ( 20 )**




10 < 20, right

30 > 20, left

25 > 20, left

Insert 20 on left

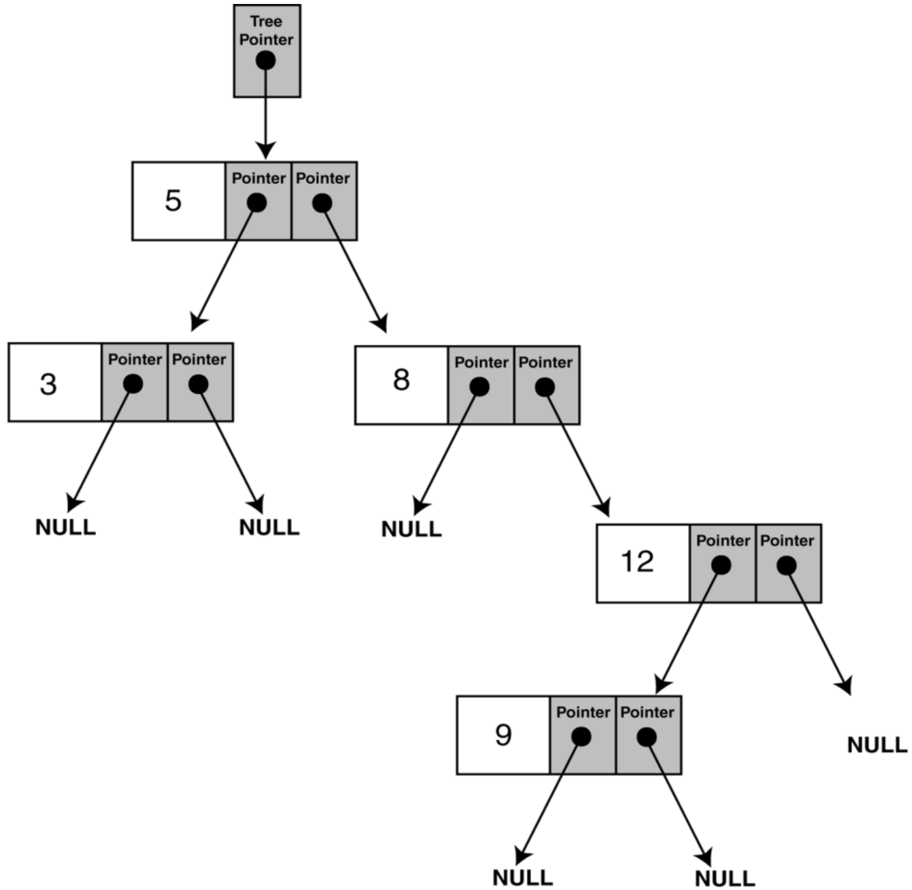
```
void IntBinaryTree::insertNode(int num){
        TreeNode *newNode,       // Pointer to a new node
                 *nodePtr;       // Pointer to traverse the tree
        newNode = new TreeNode; // Create a new node
        newNode->value = num;
        newNode->left = newNode->right = NULL;
        if (!root){root = newNode;}      // Is the tree empty?
        else{
                nodePtr = root;
                while (nodePtr != NULL){
                        if (num < nodePtr->value){
                                if (nodePtr->left)
                                        nodePtr = nodePtr->left;
                                else {
                                     nodePtr->left = newNode;   break;
                                }
                        }
                        else if (num > nodePtr->value) {
                                if (nodePtr->right)
                                        nodePtr = nodePtr->right;
                                else {
                                      nodePtr->right = newNode; break;
                                }
                        }
                        else {
                           cout << "Duplicate value found.\n";     break;
                        }
                }
        }
}
```

```
// This program builds a binary tree with 5 nodes.

#include <iostream.h>
#include "IntBinaryTree.h"

int main(void)
{
        IntBinaryTree tree;

        cout << "Inserting nodes. ";
        tree.insertNode(5);
        tree.insertNode(8);
        tree.insertNode(3);
        tree.insertNode(12);
        tree.insertNode(9);
        cout << "Done.\n";
        return 0;
}
```




***<u>Note:</u>*** *The <u>shape</u> of the tree is determined by the **order** in which the values are <u>inserted</u>. The root node in the diagram above holds the value 5 because that was the <u>first value inserted</u>.*

# Traversal



- The IntBinaryTree class can *display* **all** the values in the tree using all 3 of these algorithms.

- The algorithms are initiated by the following inline public member functions -

```
void showNodesInOrder(void)
      {     displayInOrder(root); }
void showNodesPreOrder()
      {     displayPreOrder(root); }
void showNodesPostOrder()
      {     displayPostOrder(root); }
```

# Traversal



- Each of these public member functions calls a *recursive* private member function and passes the root pointer as argument.

```
void showNodesInOrder(void)
        {       displayInOrder(root); }
void showNodesPreOrder()
        {       displayPreOrder(root); }
void showNodesPostOrder()
        {       displayPostOrder(root); }
```

# Traversal



```
void IntBinaryTree::displayInOrder(TreeNode *nodePtr)
{
       if (nodePtr)
       {
              displayInOrder(nodePtr->left);
              cout << nodePtr->value << endl;
              displayInOrder(nodePtr->right);
       }
}
```

# Traversal



```
void IntBinaryTree::displayPreOrder(TreeNode *nodePtr)
{
	if (nodePtr)
	{
		cout << nodePtr->value << endl;
		displayPreOrder(nodePtr->left);
		displayPreOrder(nodePtr->right);
	}
}
```

# Traversal



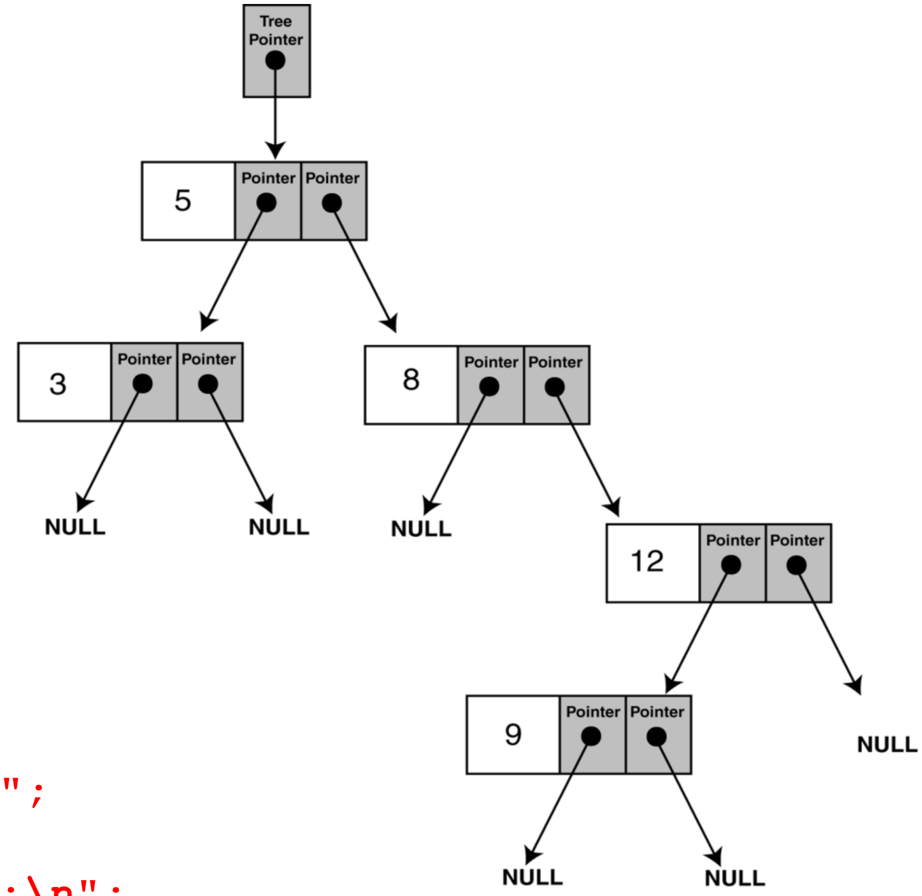
```
void IntBinaryTree::displayPostOrder(TreeNode *nodePtr)
{
       if (nodePtr)
       {
              displayPostOrder(nodePtr->left);
              displayPostOrder(nodePtr->right);
              cout << nodePtr->value << endl;
       }
}
```

```
// This program builds a binary tree with 5 nodes.

#include <iostream.h>
#include "IntBinaryTree.h"

int main(void)
{
	IntBinaryTree tree;

	cout << "Inserting nodes. ";
	tree.insertNode(5);
	tree.insertNode(8);
	tree.insertNode(3);
	tree.insertNode(12);
	tree.insertNode(9);
	cout << "Done.\n";
	cout << "Inorder traversal:\n";
	tree.showNodesInOrder();
	cout << "\nPreorder traversal:\n";
	tree.showNodesPreOrder();
	cout << "\nPostorder traversal:\n";
	tree.showNodesPostOrder();
	return 0;
}
```

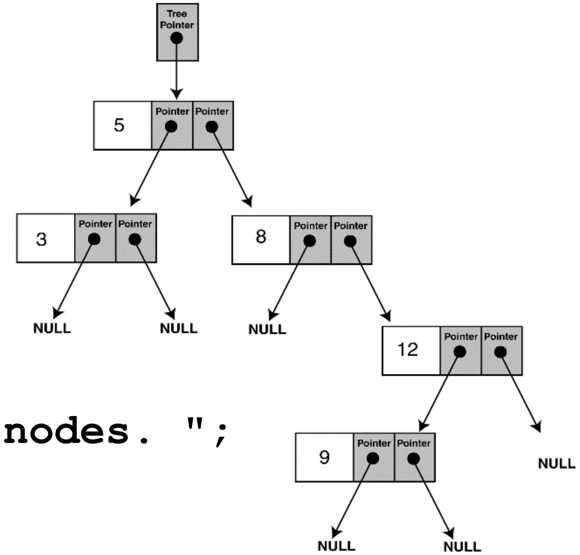


```
// This program builds a binary tree with 5 nodes.

#include <iostream.h>
#include "IntBinaryTree.h"

int main(void)
{
	IntBinaryTree tree;

	cout << "Inserting nodes. ";
	tree.insertNode(5);
	tree.insertNode(8);
	tree.insertNode(3);
	tree.insertNode(12);
	tree.insertNode(9);
	cout << "Done.\n";
	cout << "Inorder traversal:\n";
	tree.showNodesInOrder();
	cout << "\nPreorder traversal:\n";
	tree.showNodesPreOrder();
	cout << "\nPostorder traversal:\n";
	tree.showNodesPostOrder();
	return 0;
}
```




## Program Output

```
Inserting nodes.
Inorder traversal:
3
5
8
9
12
Preorder traversal:
5
3
8
12
9

Postorder traversal:
3
9
12
8
5
```

# Recursive Search of Binary Tree

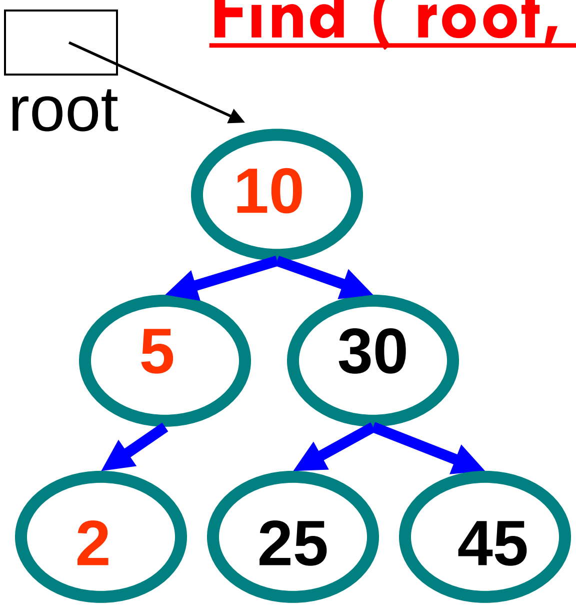


```
Node *Find( Node *n, int key) {
      if (n == NULL)                    // Not found
            return( n );
      else if (n->data == key)          // Found it
            return( n );
      else if (n->data > key)           // In left subtree
            return Find( n->left, key );
      else                              // In right subtree
            return Find( n->right, key );
}
Node * n = Find( root, 5);
```

# Examples



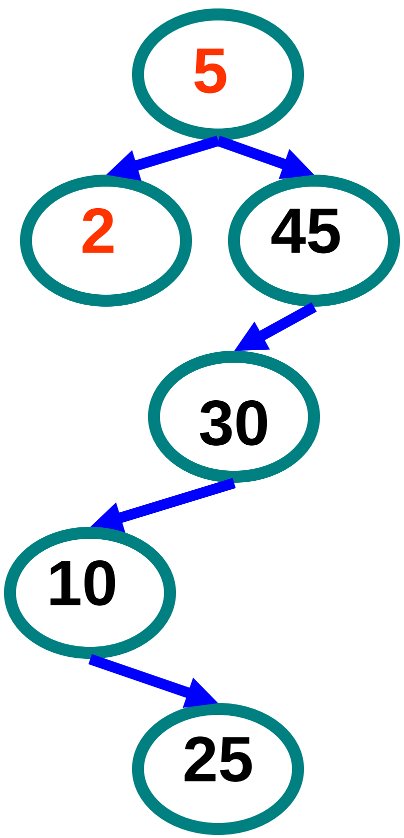
**Find ( root, 2 )**




**10 > 2, left**

**5 > 2, left**

**2 = 2, found**



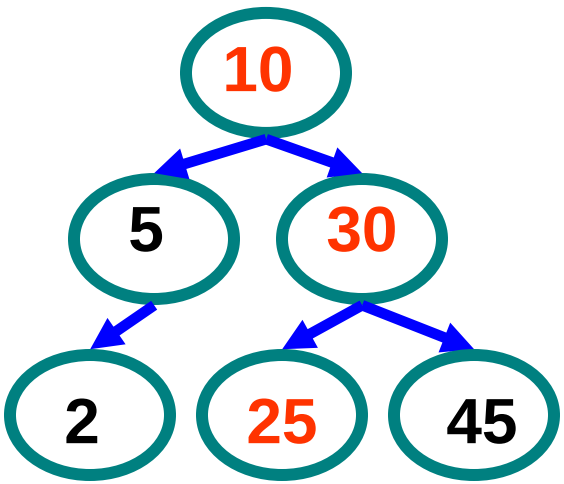

**5 > 2, left**

**2 = 2, found**

# Examples



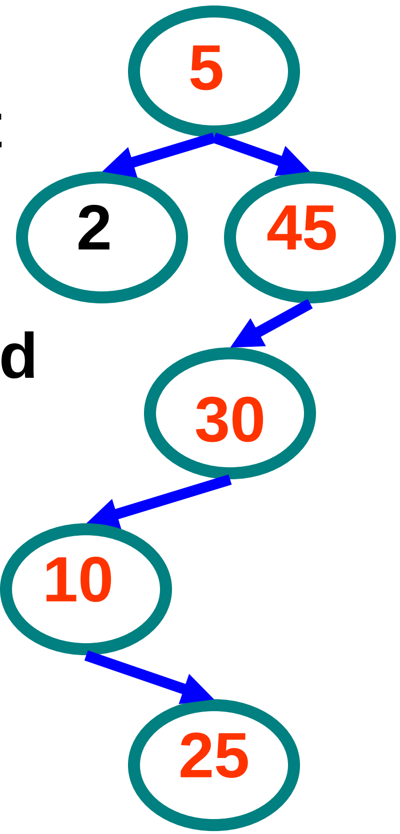
**Find ( root, 25 )**




10 < 25, right

30 > 25, left

25 = 25, found




5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

25 = 25, found

# Reading Materials



- Schaum's Outlines: Chapter # 7
- D. S. Malik: Chapter # 11
- Nell Dale: Chapter # 8
- Allen Weiss: Chapter # 4
- Tenebaum: Chapter # 5