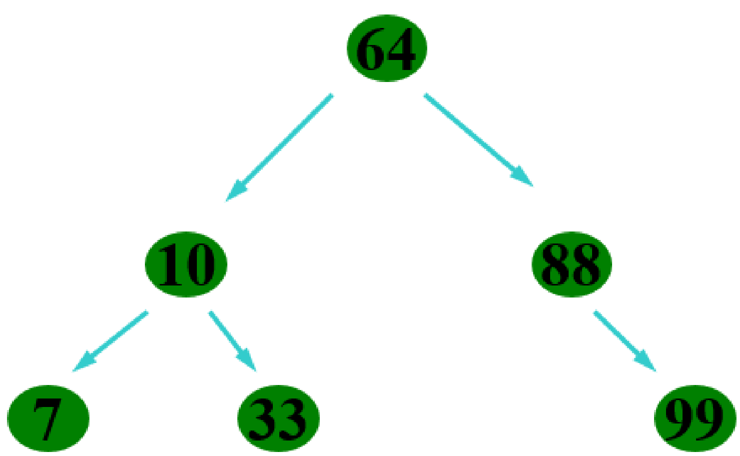
The School Electrical Engineering and Information Technology
Computer Science Department

CS223 Lab 6
Binary Search Tree

Definition:

- BST : A tree in which each node has 0,1 or 2 children and data in each node is:
- **Larger** than the data in its **left** child
- **Smaller** than the data in its **right** child



A structure, which contains a data element and a pointer to the left and right child , is created as follows:

```
struct node {
int key;
struct node *left, *right;
};
```

Main Operations

- **Insert Operation**

Add a node onto the tree and return the root node

- **Delete Operation**

Remove a node from the tree and return the root node

- **Traverse Operation (inorder, preorder or postorder)**

Visiting each node in the tree

- **Search Operation**

Check whether a given key is exists or not

Lab Work

Consider the following C++ code, which contains the following operations for Binary search tree:

- Define a structure node
- A function to create a new node and return a pointer to that node
- Insert function, which takes a root and a key as parameters, and return a pointer to the root
- Delete function which call a minValue function
- In Order Traverse function

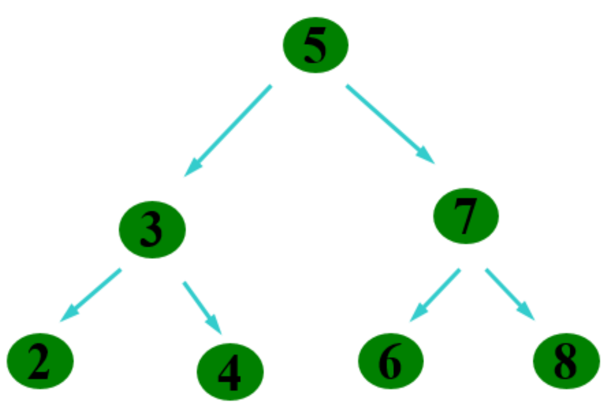
```
#include <iostream>
using namespace std;

struct node {
        int key;
        struct node *left, *right;
};
struct node* newNode(int item)
{
        struct node* temp=new node;
        temp->key = item;
        temp->left = temp->right = NULL;
        return temp;
}
void inorder(struct node* root)
{
        if (root != NULL) {
                inorder(root->left);
                cout << root->key;
                inorder(root->right);
        }
}
```

```
struct node* insert(struct node* node, int data){

 /* If the tree is empty, return a new node */
	if (node == NULL)
		return newnode(key);

	/* Otherwise, recur down the tree */
	if (key < node->key)
		node->left = insert(node->left, key);
	else
		node->right = insert(node->right, key);

	/* return the (unchanged) node pointer */
	return node;
 }


struct node*  minValueNode(struct node* root) {


}


bool search(struct node* root, int key){

  }

struct node* deleteNode(struct node* root, int key)
{
	// base case
	if (root == NULL)
		return root;

	// If the key to be deleted is
	// smaller than the root's
	// key, then it lies in left subtree
	if (key < root->key)
		root->left = deleteNode(root->left, key);

	// If the key to be deleted is
	// greater than the root's
	// key, then it lies in right subtree
	else if (key > root->key)
		root->right = deleteNode(root->right, key);

	// if key is same as root's key, then This is the node
	// to be deleted
	else {
```

```
        // node has no child
        if (root->left==NULL && root->right==NULL)
            return NULL;

        // node with only one child or no child
     else if (root->left == NULL) {
            struct node* temp = root->right;
            delete(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor
        // (smallest in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

Lab Exercises:

1. Call the insert function to build the following tree:



2. Add a Minimum function, which takes a root and return a pointer to the node with the minimum key.

3. Add a search Function to check if a given key exist or not.