1. **Lowest Common Ancestor in Binary Tree**

**Algorithm :**

* Create a recursive function that takes a node and the two values n1 and n2.
* If the value of the current node is less than both n1 and n2, then LCA lies in the right subtree. Call the recursive function for the right subtree.
* If the value of the current node is greater than both n1 and n2, then LCA lies in the left subtree. Call the recursive function for the left subtree.
* If both the above cases are false, then return the current node as LCA.

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node \*left, \*right;

};

struct node \*lca (struct node \*root, int n1, int n2)

{

while (root != NULL)

{

if (root->data > n1 && root->data > n2)

root = root->left;

else if (root->data < n1 && root->data < n2)

root = root->right;

else

break;

}

return root;

}

struct node \*newNode (int data)

{

struct node \*node = (struct node \*) malloc (sizeof (struct node));

node->data = data;

node->left = node->right = NULL;

return (node);

}

int main ()

{

struct node \*root = newNode (20);

root->left = newNode (8);

root->right = newNode (22);

root->left->left = newNode (4);

root->left->right = newNode (12);

root->left->right->left = newNode (10);

root->left->right->right = newNode (14);

int n1 = 10, n2 = 14;

struct node \*t = lca (root, n1, n2);

printf ("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 14, n2 = 8;

t = lca (root, n1, n2);

printf ("LCA of %d and %d is %d \n", n1, n2, t->data);

n1 = 10, n2 = 22;

t = lca (root, n1, n2);

printf ("LCA of %d and %d is %d \n", n1, n2, t->data);

getchar ();

return 0;

}

**OUTPUT :** LCA of 10 and 14 is 12

LCA of 14 and 8 is 8

LCA of 10 and 22 is 20

1. **Height of a Binary Tree**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node \*left;

struct node \*right;

};

int height (struct node \*node)

{

if (node == NULL)

return 0;

else

{

int leftHeight = height (node->left);

int rightHeight = height (node->right);

if (leftHeight > rightHeight)

return (leftHeight + 1);

else

return (rightHeight + 1);

}

}

struct node \*newNode (int data)

{

struct node \*node = (struct node \*) malloc (sizeof (struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

int main ()

{

struct node \*root = newNode (10);

root->left = newNode (20);

root->right = newNode (30);

root->left->left = newNode (40);

root->left->right = newNode (50);

printf ("Height of tree is %d", height (root));

return 0;

}

OUTPUT : Height of tree is 3

**3. Program to perform insertion in Binary Search Tree**

#include<stdio.h>

#include<stdlib.h>

// Basic struct of Tree

struct node

{

int val;

struct node \*left, \*right;

};

// Function to create a new Node

struct node\* newNode(int item)

{

struct node\* temp = (struct node \*)malloc(sizeof(struct node));

temp->val = item;

temp->left = temp->right = NULL;

return temp;

}

// Function print the node in inorder format, when insertion is complete

void inorder(struct node\* root)

{

if (root != NULL)

{

inorder(root->left);

printf("%d \n", root->val);

inorder(root->right);

}

}

// Here we are finding where to insert the new node so BST is followed

struct node\* insert(struct node\* node, int val)

{

/\* If the tree(subtree) is empty, return a new node by calling newNode func.\*/

if (node == NULL) return newNode(val);

/\* Else, we will do recursion down the tree to further subtrees \*/

if (val < node->val)

node->left = insert(node->left, val);

else if (val > node->val)

node->right = insert(node->right, val);

/\* (Safety) return the node's pointer which is unchanged \*/

return node;

}

int main()

{

/\* Our BST will look like this

100

/ \

40 140

/ \ / \

40 80 120 160 \*/

struct node\* root = NULL;

root = insert(root, 100);

insert(root, 60);

insert(root, 40);

insert(root, 80);

insert(root, 140);

insert(root, 120);

insert(root, 160);

// Finally printing the tree using inorder

inorder(root);

return 0;

}

OUTPUT : **40**

**60**

**80**

**100**

**120**

**140**

**160**