1. Find K’th smallest and K’th largest element in BST

#include <iostream>

#include <climits>

using namespace std;

struct Node

{

int data;

Node \*left, \*right;

};

Node \*newNode(int key)

{

Node \*node = new Node;

node->data = key;

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

return node;

}

Node \*insert(Node \*root, int key)

{

if (root == NULL)

return newNode(key);

if (key < root->data)

root->left = insert(root->left, key);

else

root->right = insert(root->right, key);

return root;

}

void inorder(Node \*root)

{

if (root == NULL)

return;

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

int kthSmallest(Node \*root, int \*i, int k)

{

if (root == NULL)

return INT\_MAX;

int left = kthSmallest(root->left, i, k);

if (left != INT\_MAX)

return left;

if (++\*i == k)

return root->data;

return kthSmallest(root->right, i, k);

}

int kthSmallest(Node \*root, int k)

{

int i = 0;

return kthSmallest(root, &i, k);

}

int kthLargest(Node \*root, int \*i, int k)

{

if (root == NULL)

return INT\_MAX;

int left = kthLargest(root->right, i, k);

if (left != INT\_MAX)

return left;

if (++\*i == k)

return root->data;

return kthLargest(root->left, i, k);

}

int kthLargest(Node \*root, int k)

{

int i = 0;

return kthLargest(root, &i, k);

}

int main()

{

struct Node \*root = NULL;

root = insert(root, 50);

root = insert(root, 13);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 3);

root = insert(root, 9);

root = insert(root, 80);

root = insert(root, 4);

root = insert(root, 60);

root = insert(root, 111);

int k = 2;

int smallest = kthSmallest(root, k);

if (smallest != INT\_MAX)

cout << "k'th smallest: " << smallest << endl;

else

cout << "Invalid Input";

int largest = kthLargest(root, k);

if (largest != INT\_MAX)

cout << "k'th largest: " << largest;

else

cout << "Invalid Input";

return 0;

}

# 2. Remove nodes from BST that have keys outside the valid range (range will be given by user)

#include <iostream>

using namespace std;

// Data structure to store a Binary Search Tree node

struct Node

{

int data;

Node \*left, \*right;

};

// Function to create a new binary tree node having given key

Node \*newNode(int key)

{

Node \*node = new Node;

node->data = key;

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

return node;

}

// Function to perform inorder traversal of the tree

void inorder(Node \*root)

{

if (root == NULL)

return;

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

// Recursive function to insert a key into BST

Node \*insert(Node \*root, int key)

{

// if the root is null, create a new node and return it

if (root == NULL)

return newNode(key);

// if given key is less than the root node, recur for left subtree

if (key < root->data)

root->left = insert(root->left, key);

// if given key is more than the root node, recur for right subtree

else

root->right = insert(root->right, key);

return root;

}

// Function to truncate the BST and remove nodes having keys

// outside valid range

Node \*truncate(Node \*root, int min, int max)

{

// base case

if (root == NULL)

return root;

// recursively truncate left and right subtree first

root->left = truncate(root->left, min, max);

root->right = truncate(root->right, min, max);

Node \*curr = root;

// if root's key is smaller than the minimum allowed, delete it

if (root->data < min)

{

root = root->right;

delete curr;

}

// if root's key is larger than the maximum allowed, delete it

else if (root->data > max)

{

root = root->left;

delete curr;

}

return root;

}

// Remove nodes from BST that have keys outside the valid range

int main()

{

Node \*root = NULL;

root = insert(root, 15);

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 8);

root = insert(root, 12);

root = insert(root, 16);

root = insert(root, 25);

// [9, 12] is valid range

root = truncate(root, 9, 20);

inorder(root);

return 0;

}

# 3. Find a value either exist in BST or not

#include<iostream>

#include<stdlib.h>

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;

}

//printing the whole tree

void PrintTree(struct node \*root)

{

if (root == NULL)

return;

cout<<root->key<<" ";

PrintTree(root->left);

PrintTree(root->right);

}

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

{

if (node == NULL)

return newNode(key);

if (key < node->key)

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

else

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

return node;

}

struct node \* minValueNode(struct node\* node)

{

struct node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

struct node\* deleteNode(struct node\* root, int key)

{

//case 1: No Child

if (root == NULL)

return root;

// find left or right

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

//case 2:

else

{

// if only right child present

if (root->left == NULL)

{

struct node \*temp = root->right;

free(root);

return temp;

}

// if only left child present

else if (root->right == NULL)

{

struct node \*temp = root->left;

free(root);

return temp;

}

// if two child present

struct node\* temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

return root;

}

bool valueExist (node \*n, int val)

{

if (n==NULL)

{

return false;;

}

if (n->key== val)

{

return true;

}

if (n->key > val)

valueExist(n->left,val);

else

valueExist(n->right,val);

}

int main()

{

struct node \*root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

cout<<"Printing Tree: ";

PrintTree(root);

cout<<"\nDelete 20";

root = deleteNode(root, 20);

cout<<"\nDelete 30";

root = deleteNode(root, 30);

cout<<"\nPrinting Tree: ";

PrintTree(root);

cout<<endl;

if (valueExist(root, 20))

{

cout <<"\nExist "<< endl;

}

else

{

cout <<"\nNot Exist"<< endl;

}

if (valueExist(root, 70))

{

cout <<"\nExist "<< endl;

}

else

{

cout <<"\nNot Exist"<< endl;

}

return 0;

}

# 4. Replace every element of an array with the least greater element on its right

#include <iostream>

using namespace std;

// Data structure for binary search tree node

struct Node

{

int key;

Node \*left, \*right;

};

// Utility function to create a new binary search tree node

Node \*allocateNode(int key)

{

Node \*node = new Node;

node->key = key;

node->left = NULL;

node->right = NULL;

return node;

}

// Function to insert a specified key in the binary search tree rooted at

// specified node and also find its successor

void insert(Node \*&root, int key, int &successor)

{

// base case: empty tree

if (root == NULL)

{

root = allocateNode(key);

return;

}

// if the key is less than root

if (key < root->key)

{

// set successor as current node

successor = root->key;

// traverse the left subtree

insert(root->left, key, successor);

}

// if the key is more than root

else if (key > root->key)

{

// traverse the right subtree

insert(root->right, key, successor);

}

}

// Replace each element of the specified array with the

// least greater element on its right

void findInorderSuccessor(int arr[], int n)

{

// root of the binary search tree

Node \*root = NULL;

// traverse the array from the end

for (int i = n - 1; i >= 0; i--)

{

// insert the current element in the binary search tree

// and replace it with its in-order successor

int successor = -1;

insert(root, arr[i], successor);

arr[i] = successor;

}

// print the resultant array

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

}

int main()

{

int arr[] = {7, 95, 16, 21, 90, 99, 6, 55, 15, 6};

int n = sizeof(arr) / sizeof(arr[0]);

findInorderSuccessor(arr, n);

return 0;

}