BST and AVL implementation Assignment

Rishi Yadav

November 2020

1. Beginning with an empty binary search tree with complexity analysis, Construct binary search tree by inserting the values in the order given. After constructing a binary tree –

    (a) Insert new node

    (b) Find number of nodes in longest path

    (c) Minimum data value found in the tree

    (d) How to delete a node from BST with the help of all three cases

    (e) Search a value

```cpp


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


class Node{

  public:
    int data;
    Node *left;
    Node *right;
};

Node *root;

Node *getNode(int node_value)
{
  Node *temp = new Node;
  temp -> data = node_value;
  temp -> left = NULL;
  temp -> right = NULL;
  return temp;
}

void insert(int node_value)
{
  /*
  case 1 - when tree is empty
    simply create a node and insert to root;
  case 2 - when value is less or equal than root and left pointer        points to null
    create a node
    just insert to left
  case 3 - when value is greater than root and right pointer points to null
    create a node
    just insert to right
  case 4 - when both pointers are not null
    if value less than root
      traverse to left subtree
    if value greater than or equal than root
      traverse to right subtree
```

```
  */

  Node *temp = getNode(node_value);

  if (root == NULL)
  {
    root = temp;
    return;
  }
  Node *curr_ptr = root;
  while(curr_ptr!=NULL)
  {
    if(node_value <= curr_ptr->data && curr_ptr -> left == NULL)
    {
      curr_ptr->left = temp;
      return;
    }
    else if(node_value > curr_ptr->data && curr_ptr -> right == NULL)
    {
      curr_ptr -> right = temp;
      return;
    }
    else if(node_value <= curr_ptr->data && curr_ptr -> left != NULL)
      curr_ptr = curr_ptr -> left;
    else
      curr_ptr = curr_ptr -> right;
  }
}

void inorder(Node *local_root)
{
  if(local_root == NULL) return;
  inorder(local_root->left);
  cout<<local_root->data<<" ";
  inorder(local_root->right);
}

int maxLengthBST(Node *local_root)
{
  if(local_root == NULL) return 0;

  else
  {

    int leftNodes = maxLengthBST(local_root->left);
    int rightNodes = maxLengthBST(local_root->right);

    if (leftNodes > rightNodes) return leftNodes + 1;
    else return rightNodes + 1;
  }

}

int minimum(int x, int y, int z) {
```

```
  int min = x; /* assume x is the smallest */

  if (y < min) { /* if y is smaller than min, assign y to min */
    min = y;
  } /* end if */

  if (z < min) { /* if z is smaller than min, assign z to min */
    min = z;
  } /* end if */

  return min; /* max is the largest value */
}

int minData(Node *local_root)
{
  if(local_root == NULL) return INT_MAX;
  else
  {
    int minValue = local_root->data;

    minValue = minimum(minValue,
         minData(local_root->left),
         minData(local_root->right));

    return minValue;
  }
}

void search(Node *lRoot, int key)
{
  if(lRoot == NULL)
  {
    cout<<"ELement not found\n";
    return;
  }
  Node *curr_ptr = lRoot;
  while(curr_ptr!=NULL)
  {
    if(curr_ptr->data == key)
    {
      cout<<"\nElement found\n";
      return;
      //return curr_ptr;
    }
    else if(curr_ptr->data > key)
      curr_ptr = curr_ptr->left;
    else
      curr_ptr = curr_ptr->right;
  }
  cout<<"\nElement not found\n";
  return ;
}

Node *findMin(Node *ptr)
{
```

```
  Node *curr = ptr;
  while(curr && curr -> left !=NULL)
    curr = curr -> left;

  return curr;
}

Node* deleteNode(Node *lRoot,int value)
{
  if(lRoot == NULL) return lRoot;
  if(value < lRoot -> data)
    lRoot -> left = deleteNode(lRoot -> left, value);
  else if(value > lRoot -> data)
    lRoot -> right = deleteNode(lRoot -> right, value);
  else
  {
    if(lRoot -> left == NULL)
    {
      Node *temp = lRoot -> right;
      free(lRoot);
      return temp;
    }
    else if(lRoot -> right == NULL)
    {
      Node *temp = lRoot -> left;
      free(lRoot);
      return temp;
    }

    //Inorder successor (min value in right subtree)
    Node *temp = findMin(lRoot->right);

    lRoot -> data = temp -> data;

    lRoot -> right = deleteNode(lRoot -> right, temp -> data);
  }
  return lRoot;
}

int main() {

  int total_nodes; cin>>total_nodes;
  root = NULL;

  for(int i = 0;i<total_nodes; i++)
  {
    int node_val; cin>>node_val;
    insert(node_val);
    //insert(5);
    //insert(6);
    //insert(1);
  }

  cout<<"The inorder traversal of BST is :\n";
  inorder(root);

```

```
    cout<<"\nThe number of nodes in longest path of BST is :"<<
    maxLengthBST(root);

    cout<<"\nMinimum Data value found in the tree is :"<<
    minData(root);

    int value;
    cout<<"\nEnter which element you want to delete :";
    cin>> value;

    //search(root, value);

    deleteNode(root, value);
    cout<<"\nThe inorder traversal of BST after deletion is :\n";
    inorder(root);

    cout<<"\nEnter the element you want to search for?;";
    cin>>value;
    search(root, value);


    return 0;
  }

```

Output

```
(base) rishi@rishi-X541UAK:~/Programs$ ./bst
5
7 2 3 1 9
The inorder traversal of BST is :
1 2 3 7 9
The number of nodes in longest path of BST is :3
Minimum Data value found in the tree is :1
Enter which element you want to delete :1

The inorder traversal of BST after deletion is :
2 3 7 9
Enter the element you want to search for?;3

Element found
(base) rishi@rishi-X541UAK:~/Programs$
```

2. Write a program to implement AVL Tree with various operations of Insertion, deletion and searching.

```

#include<bits/stdc++.h>

using namespace std;
struct Node {
    int key;
    Node * left;
    Node * right;
    int height;
};
int max(int a, int b);
int height(Node * N) {
```

```
    if (N == NULL)
        return 0;
    return N -> height;
}
int max(int a, int b) {
    return (a > b) ? a : b;
}
Node * newNode(int key) {
    Node * node = new Node();
    node -> key = key;
    node -> left = NULL;
    node -> right = NULL;
    node -> height = 1;
    return (node);
}
Node * rightRotate(Node * y) {
    Node * x = y -> left;
    Node * T2 = x -> right;
    x -> right = y;
    y -> left = T2;
    y -> height = max(height(y -> left), height(y -> right)) + 1;
    x -> height = max(height(x -> left), height(x -> right)) + 1;
    return x;
}
Node * leftRotate(Node * x) {
    Node * y = x -> right;
    Node * T2 = y -> left;
    y -> left = x;
    x -> right = T2;
    x -> height = max(height(x -> left), height(x -> right)) + 1;
    y -> height = max(height(y -> left), height(y -> right)) + 1;
    return y;
}
int getBalance(Node * N) {
    if (N == NULL)
        return 0;
    return height(N -> left) - height(N -> right);
}
Node * insert(Node * node, int key) {
    if (node == NULL)
        return (newNode(key));
    if (key < node -> key)
        node -> left = insert(node -> left, key);
    else if (key > node -> key)
        node -> right = insert(node -> right, key);
    else
        return node;
    node -> height = 1 + max(height(node -> left), height(node -> right));
    int balance = getBalance(node);
    if (balance > 1 && key < node -> left -> key)
        return rightRotate(node);
    if (balance < -1 && key > node -> right -> key)
```

```
        return leftRotate(node);
    if (balance > 1 && key > node -> left -> key) {
        node -> left = leftRotate(node -> left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node -> right -> key) {
        node -> right = rightRotate(node -> right);
        return leftRotate(node);
    }
    return node;
}
Node * minValueNode(Node * node) {
    Node * current = node;
    while (current -> left != NULL)
        current = current -> left;
    return current;
}
Node * deleteNode(Node * root, int key) {
    if (root == NULL)
        return root;
    if (key < root -> key)
        root -> left = deleteNode(root -> left, key);
    else if (key > root -> key)
        root -> right = deleteNode(root -> right, key);
    else {
        if ((root -> left == NULL) || (root -> right == NULL)) {
            Node * temp = root -> left ? root -> left : root -> right;
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = * temp;
            free(temp);
        } else {
            Node * temp = minValueNode(root -> right);
            root -> key = temp -> key;
            root -> right = deleteNode(root -> right,
                temp -> key);
        }
    }
    if (root == NULL)
        return root;
    root -> height = 1 + max(height(root -> left),
        height(root -> right));
    int balance = getBalance(root);
    if (balance > 1 && getBalance(root -> left) >= 0)
        return rightRotate(root);
    if (balance > 1 && getBalance(root -> left) < 0) {
        root -> left = leftRotate(root -> left);
        return rightRotate(root);
    }
    if (balance < -1 && getBalance(root -> right) <= 0)
        return leftRotate(root);
    if (balance < -1 && getBalance(root -> right) > 0) {
        root -> right = rightRotate(root -> right);
```

```
        return leftRotate(root);
    }
    return root;
}
struct Node * search(struct Node * root, int key) {
    if (root == NULL) {
        cout << "Element not found";
        return root;
    }
    if (root -> key == key) {
        cout << "\nElement found";
        return root;
    }
    if (root -> key < key)
        return search(root -> right, key);
    return search(root -> left, key);
}
void inorder(Node * root) {
    if (root != NULL) {
        inorder(root -> left);
        cout << root -> key << " ";
        inorder(root -> right);
    }
}
int main() {
    Node * root = NULL;
    char ch;
    int x;
    do {
        cout << "Enter value to be inserted\t\t";
        cin >> x;
        root = insert(root, x);
        cout << "\nInorder traversal of the AVL tree is \n";
        inorder(root);
        cout << "\nEnter y to add more elements\t";
        cin >> ch;
    } while (ch == 'y');
    do {
        cout << "Enter 1 to delete\n2 to search ";
        cin >> x;
        switch (x) {
        case 1:
            cout << "\nEnter value to be deleted :";
            cin >> x;
            root = deleteNode(root, x);
            cout << "\nInorder traversal after deletion :";
            inorder(root);
            break;
        case 2:
            cout << "\nEnter value to be searched :";
            cin >> x;
            search(root, x);
            break;
        default:
            cout << "\nEnter correct option";
        }
        cout << "\nEnter y to repeat\t";
```

```
        cin >> ch;
    } while (ch == 'y');
    return 0;
}

```

Output