AVL tree

#include <iostream>

#include <math.h>

#include <queue>

using namespace std;

#define SEPARATOR "#<ab@17943918#@>#"

enum BalanceValue

{

    LH = -1,

    EH = 0,

    RH = 1

};

void printNSpace(int n)

{

    for (int i = 0; i < n - 1; i++)

        cout << " ";

}

void printInteger(int &n)

{

    cout << n << " ";

}

template<class T>

class AVLTree

{

public:

    class Node;

private:

    Node \*root;

protected:

    int getHeightRec(Node \*node)

    {

        if (node == NULL)

            return 0;

        int lh = this->getHeightRec(node->pLeft);

        int rh = this->getHeightRec(node->pRight);

        return (lh > rh ? lh : rh) + 1;

    }

public:

    AVLTree() : root(nullptr) {}

    ~AVLTree(){}

    int getHeight()

    {

        return this->getHeightRec(this->root);

    }

    void printTreeStructure()

    {

        int height = this->getHeight();

        if (this->root == NULL)

        {

            cout << "NULL\n";

            return;

        }

        queue<Node \*> q;

        q.push(root);

        Node \*temp;

        int count = 0;

        int maxNode = 1;

        int level = 0;

        int space = pow(2, height);

        printNSpace(space / 2);

        while (!q.empty())

        {

            temp = q.front();

            q.pop();

            if (temp == NULL)

            {

                cout << " ";

                q.push(NULL);

                q.push(NULL);

            }

            else

            {

                cout << temp->data;

                q.push(temp->pLeft);

                q.push(temp->pRight);

            }

            printNSpace(space);

            count++;

            if (count == maxNode)

            {

                cout << endl;

                count = 0;

                maxNode \*= 2;

                level++;

                space /= 2;

                printNSpace(space / 2);

            }

            if (level == height)

                return;

        }

    }

      Node \*rotateRight(Node \*node)

{

    Node \*temp=node->pLeft;

    if(temp==nullptr)

        return node;

    node->pLeft=temp->pRight;

    temp->pRight=node;

    return temp;

}

Node \*rotateLeft(Node \*node)

{

    Node \*temp=node->pRight;

    if(temp==nullptr)

        return node;

    node->pRight=temp->pLeft;

    temp->pLeft=node;

    return temp;

}

Node \*rightBalance(Node \*node, bool &taller)

{

   Node \*rightTree=node->pRight;

   if(rightTree==nullptr)

   {

       return node;

   }

   if(rightTree->balance==RH)

   {

       node->balance=EH;

     node=rotateLeft(node);

     rightTree->balance=EH;

     taller=false;

   }

   else

   {

     Node \*leftTree=rightTree->pLeft;

     if(leftTree==nullptr)

     {

         return node;

     }

     if(leftTree->balance==RH)

     {

       node->balance=LH;

       rightTree->balance=EH;

     }

     else if(leftTree->balance==EH)

     {

         node->balance=EH;

       rightTree->balance=EH;

     }

     else

     {

       node->balance=EH;

       rightTree->balance=RH;

     }

     leftTree->balance=EH;

     node->pRight=rotateRight(rightTree);

     node=rotateLeft(node);

     taller=false;

   }

   return node;

}

Node \*leftBalance(Node \*node, bool &taller)

{

    Node \*leftTree=node->pLeft;

    if(leftTree==nullptr)

    {

        return node;

    }

   if(leftTree->balance==LH)

   {

       node->balance=EH;

     node=rotateRight(node);

     leftTree->balance=EH;

     taller=false;

   }

   else

   {

     Node \*rightTree=leftTree->pRight;

     if(rightTree==nullptr)

     {

         return node;

     }

     if(rightTree->balance==LH)

     {

       node->balance=RH;

       leftTree->balance=EH;

     }

     else if(rightTree->balance==EH)

     {

         node->balance=EH;

       leftTree->balance=EH;

     }

     else

     {

       node->balance=EH;

       leftTree->balance=LH;

     }

     rightTree->balance=EH;

     node->pLeft=rotateLeft(leftTree);

     node=rotateRight(node);

     taller=false;

   }

   return node;

}

Node \*insertRec(Node \*node, const int &value, bool &taller)

{

    if(node==nullptr)

  {

      node=new Node(value);

      taller=true;

      return node;

  }

  if(value<node->data)

  {

      node->pLeft=insertRec(node->pLeft,value,taller);

      if(taller)

      {

        if(node->balance==LH)

        {

          node=leftBalance(node,taller);

        }

        else if(node->balance==EH)

        {

          node->balance=LH;

        }

        else

        {

          node->balance=EH;

          taller=false;

        }

      }

  }

  else

  {

    node->pRight=insertRec(node->pRight,value,taller);

    if(taller)

    {

      if(node->balance==LH)

      {

        node->balance=EH;

        taller=false;

      }

      else if(node->balance==EH)

      {

        node->balance=RH;

      }

      else

      {

        node=rightBalance(node,taller);

      }

    }

  }

  return node;

}

void insert(const T &value)

{

    bool taller=false;

    this->root=insertRec(this->root,value,taller);

}

Node \*deleteLeftBalance(Node \*goc,bool &shorter)

{

    if(goc->balance== RH)

        goc->balance = EH;

    else if (goc->balance== EH)

    {

        goc->balance = LH;

        shorter = false;

    }

    else

    {

        Node \*leftTree = goc->pLeft;

        if (leftTree->balance== RH)

        {

            Node \*rightTree = leftTree->pRight;

            if (rightTree->balance== RH)

            {

                leftTree->balance = LH;

                goc->balance = EH;

            }

            else if (rightTree->balance== EH)

            {

                goc->balance = RH;

                leftTree->balance = EH;

            }

            else

            {

                goc->balance = EH;

                leftTree->balance = RH;

            }

            rightTree->balance = EH;

            goc->pLeft = rotateLeft(leftTree);

            goc = rotateRight(goc);

        }

        else

        {

                if (leftTree->balance!= EH)

                {

                    goc->balance = EH;

                    leftTree->balance = EH;

                }

                else

                {

                    goc->balance = LH;

                    leftTree->balance = RH;

                    shorter = false;

                }

                goc = rotateRight(goc);

        }

    }

    return goc;

}

Node \*deleteRightBalance(Node \*goc,bool &shorter)

{

    if(goc==nullptr)

    {

        return goc;

    }

    if (goc->balance== LH)

    {

        goc->balance = EH;

    }

    else if (goc->balance== EH)

        {

            goc->balance = RH;

            shorter = false;

        }

    else

    {

        Node \*rightTree = goc->pRight;

        if (rightTree->balance== LH)

        {

            Node \*leftTree = rightTree->pLeft;

            if (leftTree->balance==LH)

            {

                rightTree->balance =RH;

                goc->balance = EH;

            }

            else if (leftTree->balance==EH)

            {

                goc->balance = LH;

                rightTree->balance = EH;

            }

            else

            {

                goc->balance = EH;

                rightTree->balance = LH;

            }

            leftTree->balance = EH;

            goc->pRight = rotateRight(rightTree);

            goc = rotateLeft(goc);

        }

        else

        {

            if (rightTree->balance!=EH)

            {

                goc->balance = EH;

                rightTree->balance = EH;

            }

            else

            {

                goc->balance = RH;

                rightTree->balance = LH;

                shorter = false;

            }

            goc = rotateLeft(goc);

        }

    }

    return goc;

}

Node \*removeKey(Node \*goc,const T &value,bool &shorter, bool &suss)

{

    if(goc==nullptr)

    {

        shorter=false;

        suss=false;

        return goc;

    }

    if(value<goc->data)

    {

        goc->pLeft=removeKey(goc->pLeft,value,shorter,suss);

        if(shorter)

        {

            goc=deleteRightBalance(goc,shorter);

        }

    }

    else if( value>goc->data)

    {

        goc->pRight=removeKey(goc->pRight,value,shorter,suss);

        if(shorter)

        {

            goc=deleteLeftBalance(goc,shorter);

        }

    }

    else

    {

        Node \*deleteNode=goc;

        if(deleteNode->pRight==nullptr)

        {

            Node \*newroot=deleteNode->pLeft;

            suss=true;

            shorter=true;

            delete deleteNode;

            return newroot;

        }

        else if(deleteNode->pLeft==nullptr)

        {

            Node \*newroot=deleteNode->pRight;

            suss=true;

            shorter=true;

            delete deleteNode;

            return newroot;

        }

        else

        {

                Node \*exchPtr = goc->pLeft;

                while( exchPtr->pRight!=nullptr)

                        exchPtr = exchPtr->pRight;

                goc->data = exchPtr->data;

                goc->pLeft = removeKey(goc->pLeft,exchPtr->data, shorter, suss);

                if (shorter)

                goc= deleteRightBalance(goc,shorter);

        }

    }

    return goc;

}

void remove(const T &value){

    //TODO

    bool shorter=false;

    bool suss=false;

    this->root=this->removeKey(this->root,value,shorter,suss);

}

    class Node

    {

    private:

        T data;

        Node \*pLeft, \*pRight;

        BalanceValue balance;

        friend class AVLTree<T>;

    public:

        Node(T value) : data(value), pLeft(NULL), pRight(NULL), balance(EH) {}

        ~Node() {}

    };

};

int main()

{

    AVLTree<int> avl;

int arr[] = {10,52,98,32,68,92,40,13,42,63};

for (int i = 0; i < 10; i++){

    avl.insert(arr[i]);

}

avl.remove(10);

avl.printTreeStructure();

system("pause");

}