# Q1 – lam lai, so voi reference code

In this question, you have to perform **delete on AVL tree**. Note that:

- Provided **insert** function already.

Your task is to implement two functions: **remove**. You could define one or more functions to achieve this task.

#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;

}

}

void remove(const T &value)

{

//TODO

}

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() {}

};

};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| 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(); | 52  32 92  13 40 68 98  42 63 |
| AVLTree<int> avl;  int arr[] = {10,52,98,32,68,92,40,13,42,63,99,100};  for (int i = 0; i < 12; i++){  avl.insert(arr[i]);  }  avl.remove(13);  avl.printTreeStructure(); | 52  32 92  10 40 68 99  42 63 98 100 |

//Helping functions

void remove(const T &value){

//TODO

}

}

//--------------------TODO--------------------

    //Helping functions

*Node*\* LLRotation(*Node* \**p*) {

*Node*\* pl = *p*->pLeft;

*Node*\* plr = pl->pRight;

        pl->pRight = *p*;

*p*->pLeft = plr;

        // Update root

        if (root == *p*){

            root = pl;

        }

        return pl;

    }

*Node*\* RRRotation(*Node* \**p*) {

*Node*\* pr = *p*->pRight;

*Node*\* prl = pr->pLeft;

        pr->pLeft = *p*;

*p*->pRight = prl;

        // Update root

        if (root == *p*){

            root = pr;

        }

        return pr;

    }

*Node*\* LRRotation(*Node* \**p*) {

*Node*\* pl = *p*->pLeft;

*Node*\* plr = pl->pRight;

        pl->pRight = plr->pLeft;

*p*->pLeft = plr->pRight;

        plr->pLeft = pl;

        plr->pRight = *p*;

        // Update root

        if (root == *p*){

            root = plr;

        }

        return plr;

    }

*Node*\* RLRotation(*Node* \**p*) {

*Node*\* pr = *p*->pRight;

*Node*\* prl = pr->pLeft;

        pr->pLeft = prl->pRight;

*p*->pRight = prl->pLeft;

        prl->pRight = pr;

        prl->pLeft = *p*;

        // Update root

        if (root == *p*){

            root = prl;

        }

        return prl;

    }

    int BalanceFactor(*Node* \**p*) {

        int hl;

        int hr;

        hl = (*p* && *p*->pLeft) ? getHeightRec(*p*->pLeft) : 0;

        hr = (*p* && *p*->pRight) ? getHeightRec(*p*->pRight) : 0;

        return hl - hr;

    }

*Node*\* InPre(*Node* \**p*) {

        while (*p* && *p*->pRight != nullptr){

*p* = *p*->pRight;

        }

        return *p*;

    }

*Node*\* InSucc(*Node* \**p*) {

        while (*p* && *p*->pLeft != nullptr){

*p* = *p*->pLeft;

        }

        return *p*;

    }

*Node*\* Delete(*Node* \**p*, int *key*){

        if (*p* == nullptr){

            return nullptr;

        }

        if (*p*->pLeft == nullptr && *p*->pRight == nullptr){

            if (*p* == root){

                root = nullptr;

            }

            delete *p*;

            return nullptr;

        }

        if (*key* < *p*->data){

*p*->pLeft = Delete(*p*->pLeft, *key*); }

        else if (*key* > *p*->data){

*p*->pRight = Delete(*p*->pRight, *key*); }

        else {

*Node*\* q;

            if (getHeightRec(*p*->pLeft) > getHeightRec(*p*->pRight)){

                q = InPre(*p*->pLeft);

*p*->data = q->data;

*p*->pLeft = Delete(*p*->pLeft, q->data);

            } else {

                q = InSucc(*p*->pRight);

*p*->data = q->data;

*p*->pRight = Delete(*p*->pRight, q->data);

            }

        }

        // Balance Factor and Rotation

        if (BalanceFactor(*p*) == 2 && BalanceFactor(*p*->pLeft) == 1) {  // L1 Rotation

            return LLRotation(*p*);

        } else if (BalanceFactor(*p*) == 2 && BalanceFactor(*p*->pLeft) == -1){  // L-1 Rotation

            return LRRotation(*p*);

        } else if (BalanceFactor(*p*) == -2 && BalanceFactor(*p*->pRight) == -1){  // R-1 Rotation

            return RRRotation(*p*);

        } else if (BalanceFactor(*p*) == -2 && BalanceFactor(*p*->pRight) == 1){  // R1 Rotation

            return RLRotation(*p*);

        } else if (BalanceFactor(*p*) == 2 && BalanceFactor(*p*->pLeft) == 0){  // L0 Rotation

            return LLRotation(*p*);

        } else if (BalanceFactor(*p*) == -2 && BalanceFactor(*p*->pRight) == 0){  // R0 Rotation

            return RRRotation(*p*);

        }

        return *p*;

    }

    void remove(const *T* &*value*)

    {

        //TODO

        Delete(root, *value*);

    }

    //-------------------------------------------------------

Node\* rotL(Node\* root) {

Node\* temp = root->pRight;

root->pRight = temp->pLeft;

temp->pLeft = root;

return temp;

}

Node\* rotR(Node\* root) {

Node\* temp = root->pLeft;

root->pLeft = temp->pRight;

temp->pRight = root;

return temp;

}

int balance(Node\* root) {

int leftheight = getHeightRec(root->pLeft);

int rightheight = getHeightRec(root->pRight);

return rightheight - leftheight;

}

int getmax(Node\* root) {

while (root->pRight != nullptr) {

root = root->pRight;

}

return root->data;

}

Node\* remove(Node\* root, T value) {

if (root == nullptr) return nullptr;

else if (value < root->data) root->pLeft = remove(root->pLeft, value);

else if (root->data < value) root->pRight = remove(root->pRight, value);

else {

if (root->pLeft == nullptr && root->pRight == nullptr) { delete root; return nullptr; }

else if (root->pLeft != nullptr && root->pRight != nullptr) {

int temp = getmax(root->pLeft);

root->data = temp;

root->pLeft = remove(root->pLeft, temp);

}

else {

Node\* ok = root->pLeft ? root->pLeft : root->pRight;

return ok;

}

};

int b\_fact = getHeightRec(root->pRight) - getHeightRec(root->pLeft);

if (b\_fact > 1) {

int b\_fact2 = getHeightRec(root->pRight->pRight) - getHeightRec(root->pRight->pLeft);

if (b\_fact2 >= 0) {

root = rotL(root);

}

else {

root->pRight = rotR(root->pRight);

root = rotL(root);

}

}

else if (b\_fact < -1) {

int b\_fact2 = getHeightRec(root->pLeft->pRight) - getHeightRec(root->pLeft->pLeft);

if (b\_fact2 <= 0) {

root = rotR(root);

}

else {

root->pLeft = rotL(root->pLeft);

root = rotR(root);

}

}

return root;

}

void remove(const T& value) {

//TODO

root = remove(root, value);

}

# Q2

In this question, you have to perform **add** on AVL tree. Note that:

- When adding a node which has the same value as parent node, add it in the**right sub tree**.

Your task is to implement function: **insert**. You could define one or more functions to achieve this task.

#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;

}

}

void insert(const T &value)

{

//TODO

}

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() {}

};

};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| AVLTree<int> avl;  for (int i = 0; i < 9; i++){  avl.insert(i);  }  avl.printTreeStructure(); | 3  1 5  0 2 4 7  6 8 |
| AVLTree<int> avl;  for (int i = 10; i >= 0; i--){  avl.insert(i);  }  avl.printTreeStructure(); | 7  3 9  1 5 8 10  0 2 4 6 |

//Helping functions

void insert(const T &value){

//TODO

}

//Helping functions

*Node*\* LLRotation(*Node* \**p*) {

*Node*\* pl = *p*->pLeft;

*Node*\* plr = pl->pRight;

        pl->pRight = *p*;

*p*->pLeft = plr;

        // Update root

        if (root == *p*){

            root = pl;

        }

        return pl;

    }

*Node*\* RRRotation(*Node* \**p*) {

*Node*\* pr = *p*->pRight;

*Node*\* prl = pr->pLeft;

        pr->pLeft = *p*;

*p*->pRight = prl;

        // Update root

        if (root == *p*){

            root = pr;

        }

        return pr;

    }

*Node*\* LRRotation(*Node* \**p*) {

*Node*\* pl = *p*->pLeft;

*Node*\* plr = pl->pRight;

        pl->pRight = plr->pLeft;

*p*->pLeft = plr->pRight;

        plr->pLeft = pl;

        plr->pRight = *p*;

        // Update root

        if (root == *p*){

            root = plr;

        }

        return plr;

    }

*Node*\* RLRotation(*Node* \**p*) {

*Node*\* pr = *p*->pRight;

*Node*\* prl = pr->pLeft;

        pr->pLeft = prl->pRight;

*p*->pRight = prl->pLeft;

        prl->pRight = pr;

        prl->pLeft = *p*;

        // Update root

        if (root == *p*){

            root = prl;

        }

        return prl;

    }

    int BalanceFactor(*Node* \**p*) {

        int hl;

        int hr;

        hl = (*p* && *p*->pLeft) ? getHeightRec(*p*->pLeft) : 0;

        hr = (*p* && *p*->pRight) ? getHeightRec(*p*->pRight) : 0;

        return hl - hr;

    }

*Node*\* rInsert(*Node* \**p*, *T* *key*) {

        if (*p* == nullptr){

*Node*\* t = **new** *Node*(*key*);

            return t;

        }

        if (*key* < *p*->data){

*p*->pLeft = rInsert(*p*->pLeft, *key*);

        } else {

*p*->pRight = rInsert(*p*->pRight, *key*);

        }

        // Balance Factor and Rotation

        if (BalanceFactor(*p*) == 2 && BalanceFactor(*p*->pLeft) == 1) {

            return LLRotation(*p*);

        } else if (BalanceFactor(*p*) == 2 && BalanceFactor(*p*->pLeft) == -1){

            return LRRotation(*p*);

        } else if (BalanceFactor(*p*) == -2 && BalanceFactor(*p*->pRight) == -1){

            return RRRotation(*p*);

        } else if (BalanceFactor(*p*) == -2 && BalanceFactor(*p*->pRight) == 1){

            return RLRotation(*p*);

        }

        return *p*;

    }

    void insert(const *T* &*value*){

        //TODO

        root = rInsert(root, *value*);

    }

    //----------------------------------------------------------------

# Q3

In this question, you have to search and print inorder on **AVL tree**. You have o implement functions: **search** and **printInorder** to complete the task. Note that:

- When the tree is null, don't print anything.

- There's a whitespace at the end when print the tree inorder in case the tree is not null.

- When tree contains value, search return true.

#include <iostream>

#include <queue>

using namespace std;

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

enum BalanceValue

{

LH = -1,

EH = 0,

RH = 1

};

template<class T>

class AVLTree

{

public:

class Node;

private:

Node \*root;

public:

AVLTree() : root(nullptr) {}

~AVLTree(){}

void printInorder(){

//TODO

}

bool search(const T &value){

//TODO

}

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() {}

};

};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| AVLTree<int> avl;  int arr[] = {10,52,98,32,68,92,40,13,42,63,99,100};  for (int i = 0; i < 12; i++){  avl.insert(arr[i]);  }  avl.printInorder();  cout << endl;  cout << avl.search(10); | 10 13 32 40 42 52 63 68 92 98 99 100  1 |

void Inorder(*Node* \**p*) {

        if (*p*){

            Inorder(*p*->pLeft);

            cout << *p*->data << " ";

            Inorder(*p*->pRight);

        }

    }

    void printInorder(){

        //TODO

        Inorder(root);

    }

    bool Rsearch(*Node*\* *root*,const *T* &*value*){

        if (*root* == nullptr){

            return false;

        }

        if (*root*->data == *value*){

            return true;

        }

        if (*value* < *root*->data){

            return Rsearch(*root*->pLeft, *value*);

        } else {

            return Rsearch(*root*->pRight, *value*);

        }

    }

    bool search(const *T* &*value*){

        //TODO

        return Rsearch(root, *value*);

    }