**1.** Cho cây AVL sau:

Diagram

Description automatically generated

Thực hiện các thao tác:

INSERT 15

Cây AVL sau khi thực hiện thao tác được duyệt tiền thứ tự có kết quả là:

(Các số cách nhau bởi dấu phẩy, không có khoảng trắng).

|  |
| --- |
| 14,8,6,13,18,16,15,23,21,29 |

**2.** Cho cây AVL sau:

Diagram

Description automatically generated

Thực hiện các thao tác:

DELETE 6

Cây AVL sau khi thực hiện thao tác được duyệt tiền thứ tự có kết quả là:

(Các số cách nhau bởi dấu phẩy, không có khoảng trắng).

|  |
| --- |
| 14,8,13,18,16,23,21,29 |

**3.** Cho cây AVL sau:

Diagram

Description automatically generated

Thực hiện các thao tác:

DELETE 2

DELETE 10

Cây AVL sau khi thực hiện thao tác được duyệt tiền thứ tự có kết quả là:

(Các số cách nhau bởi dấu phẩy, không có khoảng trắng).

|  |
| --- |
| 29,11,6,15,13,20,42,41,44 |

**4.** Cho cây Splay sau:

Diagram

Description automatically generated

Lần lượt thực hiện các thao tác:  
INSERT 48  
INSERT 21  
Cây Splay sau khi thực hiện thao tác được duyệt tiền thứ tự có kết quả là:  
(Các số cách nhau bởi dấu phẩy, không có khoảng trắng).

|  |
| --- |
| 21,13,11,5,45,40,24,34,41,48 |

**5.** Cho cây Splay sau:

Diagram

Description automatically generated

Lần lượt thực hiện các thao tác:  
INSERT 48  
DELETE 24  
Cây Splay sau khi thực hiện thao tác được duyệt tiền thứ tự có kết quả là:  
(Các số cách nhau bởi dấu phẩy, không có khoảng trắng).

|  |
| --- |
| 13,11,5,45,40,34,41,48 |

**6.** Cho cây Splay sau:

A picture containing diagram

Description automatically generated

Lần lượt thực hiện các thao tác:

INSERT 38

SEARCH 35

Cây Splay sau khi thực hiện thao tác được duyệt tiền thứ tự có kết quả là:

(Các số cách nhau bởi dấu phẩy, không có khoảng trắng).

|  |
| --- |
| 35,4,33,21,37,38,48,44 |

**7.** Cho cây Splay sau:

A picture containing diagram

Description automatically generated

Lần lượt thực hiện các thao tác:

DELETE 48

SEARCH 44

Cây Splay sau khi thực hiện thao tác được duyệt tiền thứ tự có kết quả là:

(Các số cách nhau bởi dấu phẩy, không có khoảng trắng).

|  |
| --- |
| 44,37,35,4,33,21 |

**8.** 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  Node\* rotateLeft (Node\* root) {  Node\* tempPtr = root -> pRight;  root -> pRight = tempPtr -> pLeft;  tempPtr -> pLeft = root;  return tempPtr;  }    Node\* rotateRight (Node\* root) {  Node\* tempPtr = root -> pLeft;  root -> pLeft = tempPtr -> pRight;  tempPtr -> pRight = root;  return tempPtr;  }  Node\* leftBalance (Node\* root, bool& taller) {  Node\* leftTree = root -> pLeft;  if (leftTree -> balance == LH) {  root -> balance = EH;  root = rotateRight (root);  leftTree -> balance = EH;  /\*cout << '[' << root->value << ':' << root->Balance << ']' << endl;  cout << '[' << leftTree->value << ':' << leftTree->Balance << ']';\*/  taller = 0;  }  else if (leftTree -> pRight != NULL) {  Node\* rightTree = leftTree -> pRight;  if (rightTree -> balance == LH) {  root -> balance = RH;  leftTree -> balance = EH;  }  else if (rightTree -> balance == EH)  leftTree -> balance = EH;  else {  root -> balance = EH;  leftTree -> balance = LH;  }  rightTree -> balance = EH;  root -> pLeft = rotateLeft (leftTree);  root = rotateRight (root);  taller = 0;  }  return root;  }  Node\* rightBalance (Node\* root, bool& taller) {  Node\* rightTree = root -> pRight;  if (rightTree -> balance == RH) {  root -> balance = EH;  root = rotateLeft (root);  rightTree -> balance = EH;  taller = 0;  }  else if (rightTree -> pLeft != NULL) {  Node\* leftTree = rightTree -> pLeft;  if (leftTree -> balance == LH) {  root -> balance = EH;  rightTree -> balance = RH;  }  else if (leftTree -> balance == EH) {  rightTree -> balance = EH;  }  else {  root -> balance = LH;  rightTree -> balance = EH;  }  leftTree -> balance = EH;  root -> pRight = rotateRight (rightTree);  root = rotateLeft (root);  taller = 0;  }  return root;  }  Node\* insertAvlTree (Node\* root, T value, bool& taller) {  if (root == NULL) {  root = new Node (value);  taller = 1;  return root;  }  if (value < root -> data) {  root -> pLeft = insertAvlTree (root -> pLeft, value, taller);  if (taller) {  if (root -> balance == BalanceValue::LH)  root = leftBalance (root, taller);  else if (root -> balance == BalanceValue::EH)  root -> balance = BalanceValue::LH;  else if (root -> balance == BalanceValue::RH) {  root -> balance = EH;  taller = 0;  }  }  }  else {  root -> pRight = insertAvlTree (root -> pRight, value, taller);  if (taller) {  if (root -> balance == RH)  root = rightBalance (root, taller);  else if (root -> balance == EH)  root -> balance = RH;  else {  root -> balance = EH;  taller = 0;  }  }  }  return root;  }  void insert (const T& value) {  bool taller = 1;  root = insertAvlTree (root, value, taller);  } |

**9.** 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 printInorder (Node\* r) {  if (r == NULL) return;  printInorder (r -> pLeft);  cout << r -> data << " ";  printInorder (r -> pRight);  }  void printInorder() {  //TODO  printInorder (root);  }  bool search (Node\* r, T value) {  if (r == NULL) return 0;  if (value > r -> data) return search (r -> pRight, value);  else if (value < r -> data)return search (r -> pLeft, value);  else return 1;  }  bool search (const T& value) {  //TODO  return search (root, value);  } |

**10.** 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  int BalanceFactor (Node\* pNode) {  if (pNode == NULL) return 0;  return getHeightRec (pNode -> pRight) - getHeightRec (pNode -> pLeft);  }  Node\* LLRotation (Node\* pNode) {  Node\* plNode = pNode -> pLeft;  Node\* plrNode = plNode -> pRight;  plNode -> pRight = pNode;  pNode -> pLeft = plrNode;  if (root == pNode) {  root = plNode;  }  return plNode;  }  Node\* RRRotation (Node\* pNode) {  Node\* prNode = pNode -> pRight;  Node\* prlNode = prNode -> pLeft;  prNode -> pLeft = pNode;  pNode -> pRight = prlNode;  if (root == pNode) {  root = prNode;  }  return prNode;  }  Node\* LRRotation (Node\* pNode) {  pNode -> pLeft = RRRotation (pNode -> pLeft);  return LLRotation (pNode);  }  Node\* RLRotation (Node\* pNode) {  pNode -> pRight = LLRotation (pNode -> pRight);  return RRRotation (pNode);  }  Node\* minValueNode (Node\* pNode) {  Node\* curr = pNode;  while (curr && curr -> pLeft != NULL) {  curr = curr -> pLeft;  }  return curr;  }  Node\* maxValueNode (Node\* pNode) {  Node\* curr = pNode;  while (curr && curr -> pRight != NULL) {  curr = curr -> pRight;  }  return curr;  }  Node\* deleteNodeRec (Node\* pNode, T key) {  if (!pNode) {  return pNode;  }  if (pNode -> data < key) {  pNode -> pRight = deleteNodeRec (pNode -> pRight, key);  }  else if (pNode -> data > key) {  pNode -> pLeft = deleteNodeRec (pNode -> pLeft, key);  }  else if (pNode -> data == key && pNode -> pLeft == NULL) {  Node\* temp = pNode -> pRight;  free (pNode);  return temp;  }  else if (pNode -> data == key && pNode -> pRight == NULL) {  Node\* temp = pNode -> pLeft;  free (pNode);  return temp;  }  else if (pNode -> data == key && pNode -> pRight != NULL && pNode -> pLeft != NULL) {  Node\* temp = maxValueNode (pNode -> pLeft);  pNode -> data = temp -> data;  pNode -> pLeft = deleteNodeRec (pNode -> pLeft, temp -> data);  }  if (pNode == NULL) {  return pNode;  }  int bf = BalanceFactor (pNode);  //LL Rotation  if (bf < LH && BalanceFactor (root -> pLeft) <= EH) {  return LLRotation(pNode);  }  //RR Rotation  if (bf > RH && BalanceFactor (root -> pLeft) >= EH) {  return RRRotation(pNode);  }  //LR Rotation  if (bf < LH && BalanceFactor (root -> pLeft) < EH) {  return LRRotation(pNode);  }  //RL Rotation  if (bf > RH && BalanceFactor (root -> pLeft) > EH) {  return RLRotation(pNode);  }  return pNode;  }  void remove (const T & value) {  this -> root = deleteNodeRec (this -> root, value);  } |