Practical 2

# Student Details

* Name : Tushar Harsora
* Roll : 19BCE509

# Aim

Implement a balanced binary search tree ( AVL) using model 2 ( Node tree) structure. The search tree operations create, insert, delete, display should be included.

# Code

* AVL (Header file)

#pragma once

#include <algorithm>

#include <iostream>

#include <cstdint>

using namespace std;

template<typename Key, typename Value>

struct Node {

public:

Node() = default;

Node(Key key, Value value, int64\_t height) : key(key), value(value), height(height),

left(nullptr), right(nullptr) { }

Key key;

Value value;

int64\_t height;

Node<Key, Value>\* left;

Node<Key, Value>\* right;

friend ostream& operator<<(ostream& out, const Node<Key, Value>\* const node) {

if (node == nullptr)

return out << "\n";

// Or Instead You Can Prematurely Check if left or right child exists

// if yes then call operator<< recursively

out << node->left;

out << node->key << " : " << node->value;

out << node->right;

return out;

}

friend void swap(Node<Key, Value>& lhs, Node<Key, Value>& rhs) {

using std::swap;

swap(lhs.key, rhs.key);

swap(lhs.value, rhs.value);

}

};

template<typename Key, typename Value>

struct AVL {

private:

Node<Key, Value>\* root;

static int64\_t getHeight(Node<Key, Value>\* node) {

return node == nullptr ? 0 : node->height;

}

Node<Key, Value>\* \_right\_rotate(Node<Key, Value>\* x) {

/\*

x y

/ \ / \

y xr ==> / \

/ \ z x

z yr / \ / \

/ \ zl zr yr xr

zl zr

\*/

auto y = x->left;

auto yr = x->left->right;

y->right = x;

x->left = yr;

x->height = 1 + std::max(getHeight(x->left), getHeight(x->right));

y->height = 1 + std::max(getHeight(y->left), getHeight(y->right));

return y;

}

Node<Key, Value>\* \_left\_rotate(Node<Key, Value>\* x) {

/\*

x y

/ \ / \

xl y ==> / \

/ \ x z

yl z / \ / \

/ \ xl yl zl zr

zl zr

\*/

auto y = x->right;

auto yl = x->right->left;

y->left = x;

x->right = yl;

x->height = 1 + std::max(getHeight(x->left), getHeight(x->right));

y->height = 1 + std::max(getHeight(y->left), getHeight(y->right));

return y;

}

Node<Key, Value>\* \_insert(Node<Key, Value>\* root, Node<Key, Value>\* new\_node) {

if (root == nullptr)

return new\_node;

if (new\_node->key < root->key) // If Key Compares Less than then Left Subtree

root->left = \_insert(root->left, new\_node);

else

root->right = \_insert(root->right, new\_node); // If Key Compares greater equal then Right Subtree

root->height = 1 + std::max(getHeight(root->left), getHeight(root->right));

auto balance = getHeight(root->left) - getHeight(root->right);

/\*

\* There Are 4 cases That are possible

\* 1) Height of left Subtree is Greater

\* a) key of grand child is less than key of child (Right Rotate)

\* b) key of grand child is greater equal to key of child (Left Right Rotate)

\* 2) Height of right Subtree is Greater

\* a) key of grand child is less than key of child (Right Left Rotate)

\* b) key of grand child is greater equal to key of child (Left Rotate)

\*/

if (balance > 1) { // Height Of left subtree is greater

if (new\_node->key < root->left->key) { // Right Rotate

return \_right\_rotate(root);

}

else { // Left Right Rotate

root->left = \_left\_rotate(root->left);

return \_right\_rotate(root);

}

}

else if (balance < -1) { // Height Of right subtree is greater

if (new\_node->key < root->right->key) { // Right Left Rotate

root->right = \_right\_rotate(root->right);

return \_left\_rotate(root);

}

else { // Left Rotate

return \_left\_rotate(root);

}

}

// There Was No Balancing Issues return this node

return root;

}

Node<Key, Value>\* \_delete(Node<Key, Value>\* root, const Key& key) {

if (root == nullptr)

return root;

if (key < root->key) {

root->left = \_delete(root->left, key);

}

else if (key > root->key) {

root->right = \_delete(root->right, key);

}

else { // Delete This

auto exactly\_one\_child = (root->left == nullptr || root->right == nullptr) &&

!(root->left == nullptr && root->right == nullptr); // A exor B

if (exactly\_one\_child) {

auto child\_node = root->left ? root->left : root->right;

delete root;

return child\_node;

}

else if (root->left == nullptr && root->right == nullptr) {

delete root;

return nullptr;

}

else {

Node<Key, Value>\* inorder\_successor = root->right;

while (inorder\_successor->left)

inorder\_successor = inorder\_successor->left;

swap(\*inorder\_successor, \*root);

delete inorder\_successor;

}

}

auto leftHeight = getHeight(root->left);

auto rightHeight = getHeight(root->right);

root->height = max(leftHeight, rightHeight) + 1;

auto balance = get\_balance(root);

if (balance > 1) {

auto left\_balance = get\_balance(root->left);

if (left\_balance >= 0) {

return \_right\_rotate(root);

}else{

root->left = \_left\_rotate(root->left);

return \_right\_rotate(root);

}

}

else if (balance < -1) {

auto right\_balance = get\_balance(root->right);

if (right\_balance <= 0) {

return \_left\_rotate(root);

}

else {

root->right = \_right\_rotate(root->right);

return \_left\_rotate(root);

}

}

return root;

}

auto get\_balance(Node<Key, Value>\* root) {

return getHeight(root->left) - getHeight(root->right);

}

public:

AVL() : root(nullptr) { }

void insert(const Key& key, const Value& value) {

Node<Key, Value>\* node = new Node<Key, Value>(key, value, 1);

root = \_insert(root, node);

}

void erase(const Key& key) {

root = \_delete(root, key);

}

friend ostream& operator<<(ostream& out, const AVL<Key, Value>& avl) {

out << avl.root;

return out;

}

};

* AVL Test (Header file)

#include <iostream>

#include "avl.h"

int main() {

AVL<int, int> avl;

for (int i = 1; i <= 10; i++) {

avl.insert(i, i \* 10);

}

std::cout << avl << "\n";

for (int i = 0; i < 15; i++) {

avl.erase(i);

cout << "After Removing : " << i << " :: " << avl << endl;

}

return 0;

}

# Inputs

This Program Doesn’t take any input the output is attached below

# Screenshots of output

## Screenshot Of Stack Program

1 : 10

2 : 20

3 : 30

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 0 ::

1 : 10

2 : 20

3 : 30

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 1 ::

2 : 20

3 : 30

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 2 ::

3 : 30

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 3 ::

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 4 ::

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 5 ::

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 6 ::

7 : 70

8 : 80

9 : 90

10 : 100

After Removing : 7 ::

8 : 80

9 : 90

10 : 100

After Removing : 8 ::

9 : 90

10 : 100

After Removing : 9 ::

10 : 100

After Removing : 10 ::

After Removing : 11 ::

After Removing : 12 ::

After Removing : 13 ::

After Removing : 14 ::

# Conclusion

Here we learned about the various techniques for rebalancing the tree. And multiple rotations and how to incorporate those into program.