Practical 3

# Student Details

* Name : Tushar Harsora
* Roll : 19BCE509

# Aim

Implementing the Scapegoat trees. Implement the insertion, deletion, view operations. With no extra space per node.

# Code

* Scapegoat (Header File)

#ifndef SCAPEGOAT\_H

#define SCAPEGOAT\_H

#include <iostream>

#include <cassert>

#include <vector>

#include <tuple>

#include <queue>

#include <cstdint>

template<typename Key, typename Value>

struct Node {

public:

Node() = default;

Node(Key key, Value value) : key(key), value(value),

left(nullptr), right(nullptr) { }

Key key;

Value value;

Node<Key, Value>\* left;

Node<Key, Value>\* right;

size\_t size() const {

int count = 1;

std::queue<Node<Key, Value>\*> q;

q.push(left);

q.push(right);

while (!q.empty()) {

auto f = q.front();

q.pop();

if (f != nullptr) {

q.push(f->left);

q.push(f->right);

count++;

}

}

return count;

}

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

using std::swap;

swap(lhs.key, rhs.key);

swap(lhs.value, rhs.value);

}

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

if (node == nullptr)

return out;

// 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 << "\n";

out << node->right;

return out;

}

};

template<typename Key, typename Value>

struct scapegoat\_tree {

public:

scapegoat\_tree() : m\_root(nullptr), num\_nodes(0) { }

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

auto new\_node = new Node<Key, Value>(key, value);

size\_t depth = 1;

m\_root = \_insert(m\_root, new\_node, depth);

}

void erase(const Key& key) {

m\_root = \_delete(m\_root, key);

}

friend std::ostream& operator<<(std::ostream& out, const scapegoat\_tree<Key, Value>& tree) {

out << tree.m\_root;

return out;

}

private:

Node<Key, Value>\* \_insert(Node<Key, Value>\* root, Node<Key, Value>\* new\_node, size\_t& depth) {

if (root == nullptr) {

num\_nodes++;

return new\_node;

}

bool inserted\_left = true; // for checking if i am inserting to right or left

// Useful when unrolling from recursion to check which branch

// was taken

if (new\_node->key < root->key) {

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

}

else if (new\_node->key > root->key) {

root->right = \_insert(root->right, new\_node, ++depth);

inserted\_left = false;

}

else {

assert(false);

}

bool finding\_scapegoat = depth > log3by2(num\_nodes);

if (!finding\_scapegoat)

return root;

// TODO: Can Further Optimize calls to size using formula

// size(parent) = size(inserted\_subtree) + size(not\_inserted\_subtree) + 1

// This would require large restructuring of code.

// Result Is From Lemma 4.5 From "Scapegoat Trees" Igal Galperin, Ronald L. Rivest

size\_t my\_size = root->size();

bool i\_am\_scapegoat = my\_size < ceill((inserted\_left ? root->left->size() : root->right->size())\* factor);

if (!i\_am\_scapegoat)

return root;

size\_t ind = 0;

std::vector<Node<Key, Value>\*> store(my\_size);

to\_array(root, store, ind);

ind = 0;

return create\_balanced\_tree(store, ind, store.size());

}

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

if (root == nullptr) {

return nullptr;

}

bool deleted\_left = true; // for checking if i am inserting to right or left

// Useful when unrolling from recursion to check which branch

// was taken

if (key < root->key) {

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

}

else if (key > root->key) {

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

deleted\_left = false;

}

else {

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;

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

while (inorder\_successor->left) {

inorder\_successor\_parent = inorder\_successor;

inorder\_successor = inorder\_successor->left;

}

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

if (inorder\_successor\_parent->right == inorder\_successor)

inorder\_successor\_parent->right = inorder\_successor->right;

else

inorder\_successor\_parent->left = inorder\_successor->right;

delete inorder\_successor;

}

}

if (deleted\_left && root->left == nullptr)

return root;

if (!deleted\_left && root->right == nullptr)

return root;

size\_t my\_size = root->size();

bool i\_am\_scapegoat = my\_size < ceill((deleted\_left ? root->left->size() : root->right->size()) \* factor);

if (!i\_am\_scapegoat)

return root;

size\_t ind = 0;

std::vector<Node<Key, Value>\*> store(my\_size);

to\_array(root, store, ind);

ind = 0;

return create\_balanced\_tree(store, ind, store.size());

}

static Node<Key, Value>\* create\_balanced\_tree(std::vector<Node<Key, Value>\*> nodes, uint64\_t index, uint64\_t num\_elements) {

if (index >= nodes.size() || index < 0) {

//assert(false);

return nullptr;

}

if (num\_elements == 0)

return nullptr;

uint64\_t midpoint = num\_elements / 2;

nodes[index + midpoint]->left = create\_balanced\_tree(nodes, index, midpoint);

nodes[index + midpoint]->right = create\_balanced\_tree(nodes, index + midpoint + 1, num\_elements - midpoint - 1);

return nodes[index + midpoint];

}

static size\_t log3by2(size\_t value) {

static long double log2offactor = log2l(factor);

return static\_cast<size\_t>(std::ceill(log2l((double)value) / log2offactor));

}

static void to\_array(Node<Key, Value>\* r, std::vector<Node<Key, Value>\*>& store, size\_t& ind) {

if (r == nullptr)

return;

to\_array(r->left, store, ind);

store[ind] = r;

ind++;

to\_array(r->right, store, ind);

return;

}

Node<Key, Value>\* m\_root;

size\_t num\_nodes;

constexpr static double factor = 3.0 / 2.0;

};

#endif // !SCAPEGOAT\_H

* Scapegoat Test (main function)

#include <iostream>

#include "Scapegoat.h"

int main() {

scapegoat\_tree<int, int> tree;

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

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

}

std::cout << tree << std::endl;

for (int i = 1; i <= 11; i++) {

tree.erase(i);

std::cout << "After Deleting : " << i << "\n" << tree << std::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 Deleting : 1

2 : 20

3 : 30

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Deleting : 2

3 : 30

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Deleting : 3

4 : 40

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Deleting : 4

5 : 50

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Deleting : 5

6 : 60

7 : 70

8 : 80

9 : 90

10 : 100

After Deleting : 6

7 : 70

8 : 80

9 : 90

10 : 100

After Deleting : 7

8 : 80

9 : 90

10 : 100

After Deleting : 8

9 : 90

10 : 100

After Deleting : 9

10 : 100

After Deleting : 10

After Deleting : 11

# Conclusion

Here we learned about the various techniques for rebalancing the weight balanced tree. The α weight balanced tree is also α height balanced tree. So the Height Of scapegoat tree would be approximately .