Practical 4

# Student Details

* Name : Tushar Harsora
* Roll : 19BCE509

# Aim

Implementing The skip list data structure. The skip list should have the insert, delete and find operation. The data structure is probabilistic in nature and implement up to 3rd Level

# Code

* skiplist (Header File)

#ifndef SKIPLIST\_H

#define SKIPLIST\_H

#include <vector>

#include <cassert>

#include <random>

#include <iostream>

template<typename Key, typename Value>

struct skip\_node {

Key m\_key;

Value m\_value;

size\_t m\_level;

std::vector<skip\_node\*> forward;

skip\_node(Key key, Value value, size\_t level) : m\_key(key), m\_value(value), m\_level(level) {

forward.resize(level + 1);

for (size\_t i = 0; i <= level; i++)

forward[i] = nullptr;

}

};

template<typename Key,

typename Value, double Probability = 0.5, size\_t max\_level = 3,

typename = typename std::enable\_if<std::is\_arithmetic<Key>::value, Key>::type>

struct Skiplist {

private:

skip\_node<Key, Value>\* head;

skip\_node<Key, Value>\* NIL;

mutable std::random\_device rd;

mutable std::uniform\_real\_distribution<double> distrib;

size\_t get\_random\_level() const {

size\_t lev = 1;

while (distrib(rd) < Probability && lev < max\_level)

lev++;

return lev;

}

static size\_t node\_level(const std::vector<skip\_node<Key, Value>\*>& arr) {

int curr = 1;

Key nil\_key = std::numeric\_limits<Key>::max();

for (size\_t i = 0; i < arr.size(); i++) {

if (arr[i] && arr[i]->m\_key != nil\_key) {

curr++;

}

else {

break;

}

}

return curr;

}

public:

Skiplist() {

distrib = std::uniform\_real\_distribution(0.0, 1.0);

Key min\_value = std::numeric\_limits<Key>::min();

//Value v; // Default Constructed Value. we will not use it anyways

head = new skip\_node<Key, Value>(min\_value, Value(), max\_level);

Key max\_value = std::numeric\_limits<Key>::max();

NIL = new skip\_node<Key, Value>(max\_value, Value(), max\_level);

for (size\_t i = 0; i < head->forward.size(); i++)

head->forward[i] = NIL;

}

const skip\_node<Key, Value>\* find(const Key& key) const {

skip\_node<Key, Value>\* ret = head;

size\_t currMax = node\_level(head->forward);

for (size\_t i = currMax ; i-- > 0;) {

while (ret->forward[i] && ret->forward[i]->m\_key < key) {

ret = ret->forward[i];

}

}

ret = ret->forward[0];

if (ret->m\_key == key)

return ret;

return nullptr;

}

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

std::vector<skip\_node<Key, Value>\*> to\_update(head->forward);

size\_t currMax = node\_level(head->forward);

skip\_node<Key, Value>\* node = head;

for (size\_t i = currMax; i-- > 0;) {

while (node->forward[i] && node->forward[i]->m\_key < key) {

node = node->forward[i];

}

to\_update[i] = node;

}

node = node->forward[0];

if (node->m\_key != key) {

size\_t new\_node\_level = get\_random\_level();

size\_t curr\_level = node\_level(to\_update);

if (new\_node\_level > curr\_level) {

for (size\_t i = curr\_level + 1; i < new\_node\_level; i++) {

to\_update[i] = node;

}

}

node = new skip\_node<Key, Value>(key, value, new\_node\_level);

for (size\_t i = 0; i < new\_node\_level; i++) {

node->forward[i] = to\_update[i]->forward[i];

to\_update[i]->forward[i] = node;

}

}

else {

assert(false);

node->m\_value = value;

return;

}

}

void erase(Key key) {

skip\_node<Key, Value>\* node = head;

std::vector<skip\_node<Key, Value>\*> to\_update(head->forward);

size\_t currMax = node\_level(to\_update);

for (size\_t i = currMax; i-- > 0;) {

while (node->forward[i] && node->forward[i]->m\_key < key) {

node = node->forward[i];

}

to\_update[i] = node;

}

node = node->forward[0];

if (node->m\_key == key) { // Found Key

for (size\_t i = 0; i < to\_update.size(); i++) {

if (to\_update[i]->forward[i] != node)

break;

to\_update[i]->forward[i] = node->forward[i];

}

delete node;

}

else { // Key Not Found Then No Problem

}

}

~Skiplist() {

delete head;

delete NIL;

}

};

#endif // !SKIPLIST\_H

* Skiplist Test (main function)

#include "skiplist.h"

#include <string>

#include <iostream>

int main() {

Skiplist<int, std::string> sl;

sl.insert(1, "Tushar");

sl.insert(2, "Vishwa");

sl.insert(3, "Sakshi");

sl.insert(4, "Rahul");

std::cout << "After Insert Operation\n";

for (int i = 1; i <= 4; i++) {

auto res = sl.find(i);

if (res)

std::cout << res->m\_key << " : " << res->m\_value << std::endl;

}

for (int i = 1; i <= 2; i++)

sl.erase(i);

std::cout << "\nAfter Erasing Key 1 and 2\n";

for (int i = 0; i <= 4; i++) {

auto res = sl.find(i);

if (res)

std::cout << res->m\_key << " : " << res->m\_value << std::endl;

}

return 0;

}

# Inputs

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

# Screenshots of output

After Insert Operation

1 : Tushar

2 : Vishwa

3 : Sakshi

4 : Rahul

After Erasing Key 1 and 2

3 : Sakshi

4 : Rahul

# Conclusion

Here we learned about probabilistic nature of the skiplist. The probability is configurable in the implementation and same for height. So it doesn’t incur high memory footprint for small applications and become slow for large inputs. The skiplist has its own rng object. The key is constrained to arithmetic type for ease of code and the Value type can be any type such as objects, structs, arrays, primitive datatypes etc.