Practical 10

# Student Details

* Name : Tushar Harsora
* Roll : 19BCE509

# Aim

The aim is to implement the HashTables. In this program I’ve used the array based hashtables and which can dynamically resize based on the “max load factor”. When the load factor reaches the “max load factor”. The HashTable resizes and the load factor is rebalanced. This helps when table size grows and not initially making large sized HashTables. The HashTable supports the Insert and Erase methods.

# Code

* HashTable.h

#pragma once

#include <iostream>

#include <vector>

#include <cassert>

using namespace std;

template<typename T>

struct Node {

T value;

Node\* next;

Node(const T& value) : value(value) {

next = nullptr;

}

};

template<typename T, class Hasher>

struct HashTable {

private:

std::vector<Node<T>\*> buckets;

uint16\_t num\_elems;

constexpr static long double max\_load\_factor = 2.0;

Hasher m\_hash{};

void resize\_and\_rehash() {

std::vector<Node<T>\*> new\_buckets;

new\_buckets.resize(2 \* buckets.size());

// insert every items from buckets to new\_buckets

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

auto ptr = buckets[i];

while (ptr) {

const T& key = ptr->value;

auto hash = m\_hash(key) % new\_buckets.size();

Node<T>\* curr = ptr;

ptr = ptr->next;

curr->next = nullptr;

if (new\_buckets[hash]) {

auto first = new\_buckets[hash];

while (first->next)

first = first->next;

first->next = curr; // No need to construct new node

}

else {

new\_buckets[hash] = curr;

}

}

}

buckets = std::move(new\_buckets);

return;

}

public:

HashTable() : num\_elems(0) {

buckets.resize(5);

for (auto& bucket : buckets)

bucket = nullptr;

}

void insert(const T& key) {

uint64\_t hash = m\_hash(key);

uint64\_t position = hash % buckets.size();

if (buckets[position] == nullptr) {

buckets[position] = new Node(key);

}

else {

// go to last node whose next pointer is nullptr

Node<T>\* last = buckets[position];

while (last->next)

last = last->next;

last->next = new Node(key);

}

num\_elems++;

if (load\_factor() >= max\_load\_factor) {

resize\_and\_rehash();

}

}

void erase(const T& key) {

uint64\_t hash = m\_hash(key) % buckets.size();

Node<T>\* prev = buckets[hash];

// if we find element with specified with key then delete else do nothing

if (prev) {

auto target = buckets[hash]->next;

if (prev->value == key) {

buckets[hash] = prev->next;

delete prev;

return;

}

if (target && target->value == key) {

prev->next = target->next;

delete target;

return;

}

while (target && target->value != key) {

prev = target;

target = target->next;

}

if (target == nullptr) { // we didn't find key

return;

}

assert(target->value == key);

Node<T>\* temp = target;

prev->next = target->next;

delete temp;

}

}

long double load\_factor() const {

return static\_cast<long double>(num\_elems) / buckets.size();

}

friend ostream& operator<<(ostream& out, HashTable<T, Hasher>& tab) {

for (Node<T>\* bucket : tab.buckets) {

if (!bucket)

continue;

while (bucket) {

out << bucket->value << " ";

bucket = bucket->next;

}

}

return out;

}

};

# HashTableTest.cpp

#include <iostream>

#include "HashTable.h"

#include <unordered\_set>

using namespace std;

struct Hasher {

uint64\_t operator()(int data) {

uint64\_t val = data;

val = (val ^ (val >> 30)) \* UINT64\_C(0xbf58476d1ce4e5b9);

val = (val ^ (val >> 27)) \* UINT64\_C(0x94d049bb133111eb);

val = val ^ (val >> 31);

return val;

}

};

int main() {

HashTable<int, Hasher> s;

for (int i = 0; i < 10; i++)

s.insert(i);

cout << "After inserting [0-10) = " << s << endl;

for (int i = 5; i < 10; i++)

s.erase(i);

s.erase(1337);

cout << "After Removing [5,10) and 1337 = " << s << endl;;

return 0;

}

# Inputs

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

The Elements 1-10 are added to HashTable and some values are then removed from the hashtable.

# Screenshots of output



# Conclusion

Here we learned about the HashTables and how not to waste space when low number of elements and how to resize hashtable based on load factor. And how to implement Fixed Bucket hashtables and use chaining for collision resolution.