## CSCI 230 PA 1 Submission

## Due Date: 03/01/2022 Late (date and time):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Name(s): Nero Li

Exercise 1 -- need to submit source code and I/O  
 -- check if completely done ✔️ ; otherwise, discuss issues below

Source code below:

**exercise\_1.cpp:**

/\* Program: PA\_1\_exercise\_1

Author: Nero Li

Class: CSCI 230

Date: 02/15/2022

Description:

Use C++ STL unordered\_map or Java HashMap to store the following

integer keys: 13 21 5 37 15 (reverse the key and use it as a

string for the value part so first entry would be <13, "31">).

Perform the following operations to make sure it is working

properly: search for 10 and 21, remove 20, 37, and then search

for 37.

Input data file small1k.txt, containing a list of 1,000 integer

values, to an array and then insert all the pairs <int, reverse

key as string> to a new hash map. Collect the time it took to

insert 1,000 pairs of values to the hash map and output the time

to the screen.

Input data file large100k.txt, containing a list of 100,000

integer values, to an array and then insert all the pairs

<int, reverse key as string> to another new hash map. Collect the

time it took to insert 100,000 pairs of values to the hash map

and output the time to the screen.

I certify that the code below is my own work.

Exception(s): N/A

\*/

#include <iostream>

#include <fstream>

#include <string>

#include <chrono>

#include <vector>

#include <unordered\_map>

using namespace std;

string changeIntToString(int n)

{

string s{"\0"};

while (n > 0)

{

s += '0' + n % 10;

n /= 10;

}

return s;

}

void printResult(unordered\_map<int, string> m, int n)

{

if (m.find(n) == m.end())

cout << "N/A\n";

else

cout << "(" << m.find(n)->first << "," << m.find(n)->second << ")\n";

}

void test()

{

unordered\_map<int, string> m;

int A[] = {13, 21, 5, 37, 15};

for (int a : A)

m[a] = changeIntToString(a);

printResult(m, 10);

printResult(m, 21);

m.erase(20);

m.erase(37);

printResult(m, 37);

}

void func(string str)

{

ifstream fin;

unordered\_map<int, string> m;

vector<int> v;

fin.open(str, ios::binary);

if (!fin)

return;

while (!fin.eof())

{

int n;

fin >> n;

v.push\_back(n);

}

auto start = chrono::high\_resolution\_clock::now();

for (int a : v)

{

m[a] = changeIntToString(a);

}

auto end = chrono::high\_resolution\_clock::now();

cout << (chrono::duration\_cast<chrono::nanoseconds>(end - start).count() \* (double)1e-6) << " ms" << endl;

}

int main()

{

test();

func("small1k.txt");

func("large100k.txt");

cout << "Author: Nero Li\n";

return 0;

}

Input/output below:

N/A

(21,12)

N/A

1.0006 ms

428.911 ms

Author: Nero Li

Exercise 2 (with extra credit) -- need to submit source code and I/O  
 -- check if completely done ✔️ ; otherwise, discuss issues below

Source code below:

**HashMap.h:**

#ifndef HM\_H

#define HM\_H

#include <list>

#include <vector>

#include <exception>

#include "Entry.h"

class NonexistentElement

{

public:

NonexistentElement(const std::string& err)

: errMsg(err) {}

std::string getError()

{ return errMsg; }

private:

std::string errMsg;

};

template <typename K, typename V>

class HashMap {

public: // public types

typedef Entry<const K,V> Entry; // a (key,value) pair

class Iterator; // a iterator/position

public: // public functions

HashMap(int capacity = 100); // constructor

int size() const; // number of entries

bool empty() const; // is the map empty?

Iterator find(const K& k); // find entry with key k

Iterator put(const K& k, const V& v); // insert/replace (k,v)

void erase(const K& k); // remove entry with key k

void erase(const Iterator& p); // erase entry at p

Iterator begin(); // iterator to first entry

Iterator end(); // iterator to end entry

protected: // protected types

typedef std::list<Entry> Bucket; // a bucket of entries

typedef std::vector<Bucket> BktArray; // a bucket array

Iterator finder(const K& k); // find utility

Iterator inserter(const Iterator& p, const Entry& e); // insert utility

void eraser(const Iterator& p); // remove utility

typedef typename BktArray::iterator BItor; // bucket iterator

typedef typename Bucket::iterator EItor; // entry iterator

static void nextEntry(Iterator& p) // bucket's next entry

{ ++p.ent; }

static bool endOfBkt(const Iterator& p) // end of bucket?

{ return p.ent == p.bkt->end(); }

int hash(const K& k)

{

return k;

}

private:

int n; // number of entries

BktArray B; // bucket array

public: // public types

class Iterator { // an iterator (& position)

private:

EItor ent; // which entry

BItor bkt; // which bucket

const BktArray\* ba; // which bucket array

public:

Iterator(const BktArray& a, const BItor& b, const EItor& q = EItor())

: ent(q), bkt(b), ba(&a) { }

Entry& operator\*() const; // get entry

bool operator==(const Iterator& p) const; // are iterators equal?

Iterator& operator++(); // advance to next entry

friend class HashMap; // give HashMap access

};

};

template <typename K, typename V> // constructor

HashMap<K,V>::HashMap(int capacity) : n(0), B(capacity) { }

template <typename K, typename V> // number of entries

int HashMap<K,V>::size() const { return n; }

template <typename K, typename V> // is the map empty?

bool HashMap<K,V>::empty() const { return size() == 0; }

template <typename K, typename V> // find utility

typename HashMap<K,V>::Iterator HashMap<K,V>::finder(const K& k) {

int i = hash(k) % B.size(); // get hash index i

BItor bkt = B.begin() + i; // the ith bucket

Iterator p(B, bkt, bkt->begin()); // start of ith bucket

while (!endOfBkt(p) && (\*p).key() != k) // search for k

nextEntry(p);

return p; // return final position

}

template <typename K, typename V> // find key

typename HashMap<K,V>::Iterator HashMap<K,V>::find(const K& k) {

Iterator p = finder(k); // look for k

if (endOfBkt(p)) // didn't find it?

return end(); // return end iterator

else

return p; // return its position

}

template <typename K, typename V> // insert utility

typename HashMap<K,V>::Iterator HashMap<K,V>::inserter(const Iterator& p, const Entry& e) {

EItor ins = p.bkt->insert(p.ent, e); // insert before p

n++; // one more entry

return Iterator(B, p.bkt, ins); // return this position

}

template <typename K, typename V> // insert/replace (v,k)

typename HashMap<K,V>::Iterator HashMap<K,V>::put(const K& k, const V& v) {

Iterator p = finder(k); // search for k

if (endOfBkt(p)) { // k not found?

return inserter(p, Entry(k, v)); // insert at end of bucket

}

else { // found it?

p.ent->setValue(v); // replace value with v

return p; // return this position

}

}

template <typename K, typename V> // remove utility

void HashMap<K,V>::eraser(const Iterator& p) {

p.bkt->erase(p.ent); // remove entry from bucket

n--; // one fewer entry

}

template <typename K, typename V> // remove entry at p

void HashMap<K,V>::erase(const Iterator& p)

{ eraser(p); }

template <typename K, typename V> // remove entry with key k

void HashMap<K,V>::erase(const K& k) {

Iterator p = finder(k); // find k

if (endOfBkt(p)) // not found?

throw NonexistentElement("Erase of nonexistent"); // ...error

eraser(p); // remove it

}

template <typename K, typename V> // iterator to end

typename HashMap<K,V>::Iterator HashMap<K,V>::end()

{ return Iterator(B, B.end()); }

template <typename K, typename V> // iterator to front

typename HashMap<K,V>::Iterator HashMap<K,V>::begin() {

if (empty()) return end(); // emtpty - return end

BItor bkt = B.begin(); // else search for an entry

while (bkt->empty()) ++bkt; // find nonempty bucket

return Iterator(B, bkt, bkt->begin()); // return first of bucket

}

template <typename K, typename V> // get entry

typename HashMap<K,V>::Entry&

HashMap<K,V>::Iterator::operator\*() const

{ return \*ent; }

template <typename K, typename V> // advance to next entry

typename HashMap<K,V>::Iterator& HashMap<K,V>::Iterator::operator++() {

++ent; // next entry in bucket

if (endOfBkt(\*this)) { // at end of bucket?

++bkt; // go to next bucket

while (bkt != ba->end() && bkt->empty()) // find nonempty bucket

++bkt;

if (bkt == ba->end()) return \*this; // end of bucket array?

ent = bkt->begin(); // first nonempty entry

}

return \*this; // return self

}

template <typename K, typename V> // are iterators equal?

bool HashMap<K,V>::Iterator::operator==(const Iterator& p) const {

if (ba != p.ba || bkt != p.bkt) return false; // ba or bkt differ?

else if (bkt == ba->end()) return true; // both at the end?

else return (ent == p.ent); // else use entry to decide

}

#endif

**exercise\_2.cpp:**

/\* Program: PA\_1\_exercise\_2

Author: Nero Li

Class: CSCI 230

Date: 08/24/2021

Description:

Put together the C++ HashMap in the book (Chain Hashing; may

want to eliminate the third template parameter and add a hash

function) or Java ChainHashing (Java book) with N = 11 and test

it out with the same data and test cases from above. You might

want to come up with all relevant test cases to confirm that C++

HashMap or Java ChainHashing is working correctly.

I certify that the code below is my own work.

Exception(s): N/A

\*/

#include <iostream>

#include <fstream>

#include <string>

#include <chrono>

#include <vector>

#include "HashMap.h"

using namespace std;

string changeIntToString(int n)

{

string s{"\0"};

while (n > 0)

{

s += '0' + n % 10;

n /= 10;

}

return s;

}

void printResult(HashMap<int, string> m, int n)

{

if (m.find(n) == m.end())

cout << "N/A\n";

else

cout << "(" << (\*(m.find(n))).key() << "," << (\*(m.find(n))).value() << ")\n";

}

void eraseNumber(HashMap<int, string> &m, int n)

{

try

{

m.erase(n);

cout << "Erased " << n << endl;

}

catch(NonexistentElement& e)

{

cout << e.getError() << endl;

}

}

void test()

{

HashMap<int, string> m(11);

int A[] = {13, 21, 5, 37, 15};

for (int a : A)

m.put(a, changeIntToString(a));

printResult(m, 10);

printResult(m, 21);

eraseNumber(m, 20);

eraseNumber(m, 37);

printResult(m, 37);

}

void func(string str, int size)

{

ifstream fin;

HashMap<int, string> m(size);

vector<int> v;

fin.open(str, ios::binary);

if(!fin)

return;

while (!fin.eof())

{

int n;

fin >> n;

v.push\_back(n);

}

auto start = chrono::high\_resolution\_clock::now();

for (int a : v)

{

m.put(a, changeIntToString(a));

}

auto end = chrono::high\_resolution\_clock::now();

cout << (chrono::duration\_cast<chrono::nanoseconds>(end - start).count() \* (double)1e-6) << " ms" << endl;

}

int main()

{

test();

func("small1k.txt", 2000);

func("large100k.txt", 200000);

cout << "Modified by: Nero Li\n";

return 0;

}

Input/output below:

N/A

(21,12)

Erase of nonexistent

Erased 37

N/A

1.0015 ms

199.183 ms

Modified by: Nero Li

Answer for Question 1:

The final running time result doesn’t seem reasonable since if we got 1 millisecond with 1000 data, we should get 100 milliseconds with 100,000 data since the Big-O notation for hash map insert should be O(n). However, for the result we have seen in the terminal, it has changed much bigger than. Since I didn’t find a method to modify or change the hash table inside the unordered map and the bucket that our map has was dynamically added during the process, I think the reason for the result is due to the bigger data brought more collisions and the work for adding buckets cause the problem. It might also be due to the high-resolution clock since when I change the code to directly pass an integer as a value, the final running time is increased.

Answer for Question 2:

Both separate chaining and open addressing are popular ways to solve the hash collision issues.

For separate chaining, we will create a linked list called bucket and an array that stored a bunch of buckets. When we do the insert, we find the index by key mod array capacity, and then add that value to the index targeted linked list. This method will be easier for frequently doing insert and erase work and the situation when we cannot know the quantity of total data, but the running time will count on the key since the situation might happen when all the keys point to the same index.

For open addressing, we won’t have a linked list as the bucket. When we see the collision index, we will put the index into a function that makes it become another index until we found the index that doesn’t collide. Since we just need an array to store data, we can save more space by using open addressing, but the erasing function will become more complicated and the capacity cannot be dynamically expanded easily.