## CSCI 230 PA 2 Submission

## Due Date: 03/08/2022 Late (date and time):\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Name(s): Nero Li

Answer for Question 1:

The most effective way to generate hash code for a key of the string is the cyclic shift hash code. For this method, we use the character’s ASCII numbers and add them up together. When we added one character, we do the n-bit shift operation onto our int number to change its value, and then we add the new character into this number. Finally, we successfully transferred the string into a hash code. Based on the experiment, if we do the bit shift around 4 to 14, we will get the least collisions for a separate chaining hash table.

The polynomial hash codes are also another effective way to generate hash codes for strings. It also adds up all the characters’ ASCII numbers, but instead of doing the shift operation, we multiplied a number to enlarge the current hash code before we add a new character, and finally get the result. It will consist of more collisions, but if we can choose a great number such as 33, 37, 39, or 41, we will get fewer collisions than expected.

Answer for Question 2:

We cannot create a hash table without capacity since it allows us to use indexes and access data in the vector. However, our hash code might return a number that is out of bound. To deal with this problem, we have the compression functions to compress our hash code into an index that can be found in our current hash table.

The most popular and easiest compression function is the division method, |k| mod N, where k is our key value and N is the capacity for our vector. This will make sure that we can get an index that won’t go out of the vector’s range. Another common compression function is the MAD method, |ak + b| mod N, where k and N are the same meaning as above, a and b are randomly generated positive integers when the hash map has been constructed. It will help reduce the collide situations and make our hash table more efficient than before.

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

Source code below:

**exercise\_1.cpp:**

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

Author: Nero Li

Class: CSCI 230

Date: 03/08/2022

Description:

Perform a comparative analysis that studies the collision rates

for various hash codes for character strings, such as various

polynomial hash codes for different values of the parameter a.

Use a hash table to determine collisions, but only count

collisions where different strings map to the same hash code

(not if they map to the same location in this hash table). Test

these hash codes on text files found on the Internet.

I certify that the code below is my own work.

Exception(s): N/A

\*/

#include <iostream>

#include <fstream>

#include <cmath>

#include <unordered\_map>

#include <unordered\_set>

using namespace std;

int cyclicShiftHash(string s, int a)

{

int len = s.size();

unsigned int h = 0;

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

{

h = (h << a) | (h >> (32 - a));

h += s[i];

}

return (int)h;

}

int polynomialHash(string s, int a)

{

int len = s.size();

int h = 0;

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

h = s[i] + a \* h;

return h;

}

void HashTest(string file, string msg, bool isPoly, int a)

{

int numOfCollisions = 0;

int numOfWords = 0;

unordered\_map<int, int> hashCollisionCount;

unordered\_set<string> wordCheck;

ifstream fin;

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

if (!fin)

return;

while (!fin.eof())

{

string cur;

fin >> cur;

if (wordCheck.find(cur) == wordCheck.end())

{

int hash = isPoly ? polynomialHash(cur, a) : cyclicShiftHash(cur, a);

if (hashCollisionCount.find(hash) == hashCollisionCount.end())

hashCollisionCount[hash] = 0;

else

++hashCollisionCount[hash];

wordCheck.insert(cur);

}

}

for (auto i : hashCollisionCount)

{

if (i.second)

numOfCollisions += i.second;

}

cout << msg << " Hash in Number " << a << " Results\n";

cout << "- Number of words:\t" << wordCheck.size() << endl;

cout << "- Number of collisions:\t" << numOfCollisions << endl;

cout << endl;

fin.close();

}

int main()

{

HashTest("usdeclarPC.txt", "Polynomial", true, 39);

HashTest("usdeclarPC.txt", "Polynomial", true, 40);

HashTest("usdeclarPC.txt", "Polynomial", true, 41);

HashTest("usdeclarPC.txt", "Cyclic Shift", false, 1);

HashTest("usdeclarPC.txt", "Cyclic Shift", false, 5);

HashTest("usdeclarPC.txt", "Cyclic Shift", false, 13);

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

return 0;

}

Input/output below:

Polynomial Hash in Number 39 Results

- Number of words: 623

- Number of collisions: 0

Polynomial Hash in Number 40 Results

- Number of words: 623

- Number of collisions: 0

Polynomial Hash in Number 41 Results

- Number of words: 623

- Number of collisions: 0

Cyclic Shift Hash in Number 1 Results

- Number of words: 623

- Number of collisions: 19

Cyclic Shift Hash in Number 5 Results

- Number of words: 623

- Number of collisions: 0

Cyclic Shift Hash in Number 13 Results

- Number of words: 623

- Number of collisions: 0

Author: Nero Li

Exercise 2 -- 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 <string>

#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

};

public:

double getAvgNum();

int getMaxNum();

protected:

std::vector<int> BktSize{0};

};

template <typename K, typename V> // constructor

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

++BktSize[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

}

// New Added Function

template <typename K, typename V>

double HashMap<K,V>::getAvgNum()

{

double total = 0;

double count = 0;

for (int i : BktSize)

{

if (i)

{

total += i;

++count;

}

}

return total / count;

}

template <typename K, typename V>

int HashMap<K,V>::getMaxNum()

{

int max = 0;

for (int i : BktSize)

{

max = (max > i) ? max : i;

}

return max;

}

#endif

**exercise\_2.cpp:**

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

Author: Nero Li

Class: CSCI 230

Date: MM/DD/2022

Description:

Create a program to collect some data about chain hashing that

you worked on in previous PA. You would be able to enter the

name of input data file and a load factor. Each input data file

will have N records and the first value in the file will tell

you how many records are in the file. You would need to use N

and the load factor to determine the size of the hash table. Do

not rehash the table like the version in the book (i.e., need to

modify the code from the book). The first value of each record

will be the key (county/state code as integer type) and

remaining items on each record will be the value (population and

county/state). After all the entries are inserted to the table,

print the table size, average number of probes, and maximum

number of probes for the worst case. It would take at least one

probe for each insertion (checking initial location). Therefore,

it would be two probes if second location is examined.

I certify that the code below is my own work.

Exception(s): N/A

\*/

#include <iostream>

#include <fstream>

#include "HashMap.h"

using namespace std;

struct County

{

int pop;

string county;

};

int findPrime(int n)

{

int ans = n;

while (true)

{

bool notPrime = false;

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

{

if (ans % i == 0)

{

++ans;

notPrime = true;

break;

}

}

if (!notPrime)

break;

}

return ans;

}

void hashTest(string str, double lf = -1)

{

ifstream fin;

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

if (!fin)

return;

int n;

fin >> n;

int N;

if (lf == -1)

N = findPrime(n);

else

N = findPrime(n / lf);

HashMap<int, County> countyMap(N);

while (!fin.eof())

{

County newData;

string countyData;

bool gotKey = false;

int code = -1;

getline(fin, countyData);

for (int i = 0; i < countyData.size(); ++i)

{

if (countyData[i] == ',')

{

gotKey = true;

}

else if (countyData[i] >= '0' && countyData[i] <= '9')

{

if (gotKey)

{

if (newData.pop == -1)

{

newData.pop = countyData[i] - '0';

}

else

{

newData.pop \*= 10;

newData.pop += countyData[i] - '0';

}

}

else

{

if (code == -1)

{

code = countyData[i] - '0';

}

else

{

code \*= 10;

code += countyData[i] - '0';

}

}

}

else if (countyData[i] == '\"')

{

++i;

while (countyData[i] != '\"')

{

newData.county += countyData[i];

++i;

}

}

}

countyMap.put(code, newData);

}

cout << "For " << str;

if (lf != -1)

cout << " with load factor " << lf;

cout << endl;

cout << "- Table size:\t\t\t" << N << endl;

cout << "- Average number of probes:\t" << countyMap.getAvgNum() << endl;

cout << "- Maximum number of probes:\t" << countyMap.getMaxNum() << endl;

cout << endl;

}

int main()

{

hashTest("popSmall.txt");

hashTest("popLarge.txt", 0.25);

hashTest("popLarge.txt", 0.5);

hashTest("popLarge.txt", 0.75);

hashTest("popLarge.txt", 0.9);

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

return 0;

}

Input/output below:

For popSmall.txt

- Table size: 17

- Average number of probes: 1.23077

- Maximum number of probes: 3

For popLarge.txt with load factor 0.25

- Table size: 12799

- Average number of probes: 1.0117

- Maximum number of probes: 3

For popLarge.txt with load factor 0.5

- Table size: 6397

- Average number of probes: 1.04371

- Maximum number of probes: 3

For popLarge.txt with load factor 0.75

- Table size: 4271

- Average number of probes: 1.27237

- Maximum number of probes: 3

For popLarge.txt with load factor 0.9

- Table size: 3557

- Average number of probes: 1.29397

- Maximum number of probes: 4

Modified by: Nero Li