Ministry of Education and Science of the Russian Federation

State educational institution of higher education

<< Lobachevsky State University of Nizhny Novgorod >>

**FACULTY FOR INTERNATIONAL STUDENTS**

**Department: Computer Science**

**PRACTICE REPORT**

TOPIC OF THE PROJECT:

**Hash Table**

**Prepared by:**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Md. Tauhidul Islam, 2015BIT

**Supervised by:**

Assistant of Computer

Software Developer

**Signature:**

S. Lebedev

**NIZHNY NOVGOROD**

**2017**

**Table of contents**

* **Introduction and definition\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 1**
* **Problem statement\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2**
* **Algorithm and Data structure\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3**
* **Conclusion\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 4-5**
* **References\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 6**
* **Source code**

**Introduction and definition**

A hash table is a data structure for storing key-value pairs. Unlike a basic array, which uses index numbers for accessing elements, a hash table uses keys to look up table entries. This makes data management more manageable for the human user since its easier to catalog data entries by their attributes rather than their count in a giant list.



In C++, we implement a hash table as an array of linked lists. Its sort of like a multidimensional array. In a two-dimensional array, for instance, the elements consist of rows of a fixed length. In a hash table, however, the elements (a.k.a. buckets) can expand or shrink to accommodate a virtually infinite number of table entries.

In terms of efficiency, a hash table is a compromise between an array and a linked list. It uses both indexing and list traversal to store and retrieve data elements.

Looking up elements by index makes an array very efficient. No matter where an item is stored in the array, it always takes the same amount of time to retrieve it. In technical terms, getting an item from an array is an O(1) or “constant time operation.

Looking up elements in a linked list is a lot less efficient. You can’t just directly access any node in the list. Instead, you have to traverse down the list until you find the target item. If the item you are looking for happens to be at the front of the list, the retrieval is an O(1) operation since you only traversed down one node. If the item is at the end of the list, retrieving it would be an O(n) operation, where n is the total number of nodes in the list.

To summarize, as the number of elements increases in an array, the runtime to access an element by its index remains constant. With a linked list, the time it takes to access a particular element increases linearly with the number of elements.

**Task Statement**

Write a program that reads a file and prints top 10 common words from the file**.**

**Algorithm and Data structure**

Given a file (or stream) of words. Find the top k most frequent words in the document (or stream).

For example,

if stream = “aa bb cc bb bb cc bb dd dd ee ff ee dd ee aa ee”.

That is, {“dd”=3, “ee”=4, “ff”=1, “aa”=2, “bb”=4, “cc”=2}.

Then top 3 most frequent words are: {“bb”, “ee”, “dd”}.

One quick solution would be to create a pair object with the word and its frequency and then sort the pair array with respect to the frequency of the pair. Now, take the first k pairs from the sorted array of pairs. This is O(nlgn) solution.

O(nlgk) solution with O(n) space:

But we can improve this solution. Note that we are only concern about the top k elements. Sorting the array means we are sorting all the n elements which is unnecessary as we are only concerned for first k. Any idea popping in? Yeah, I am sure you have the same feeling that we could use a Min Heap of size k to keep top k most frequent words. That’s right. We also need to use a hashmap to keep frequency of each word.

Calculated frequency of all the words in a hashmap from the word to its frequency.

Start adding pair object of word and its frequency into a min heap where we use the frequency as the key for the min heap.

If the heap is full then remove the minimum element (top) form the heap and add add the new word-frequency pair only if the frequency of this word has frequency greater than the top word in the heap.

Once we scanned all the words in the map and the heap is properly updated then the elements contained in the min heap are the top k most frequents.

Below is a simple implementation of the above idea using separate chaining of Linked Lists for each index within the Hash Table. When an item is hashed to a specific index, it is added to that index's Linked List. where hashTable(unsigned s = 650); is constructor assuming longest word/phrase length of 25 composed of the largest letter value 25, the maximum key value that the hash function will produce is 25 \* 25 + 25 = 650 .

void insert(string word); attempts to insert a word into the hash table.

bool find(string word); attempts to find a word in the hash table and returns a true if the word is found and a false otherwise.

void print(); Prints out words in the table.

void outputData(ofstream& outFile, int listSize); outputs data to a file by storing all valid data in an array and then sorting the data by frequency using selection sort. The amount of data outputted is determined by listSize.

int hashKey(string word); Uses the hash function to change word into index of hash table. Hashes by adding up the ASCII values of each letter and then adding the length of the word.

bool insertElem(string word, ListNode\* entryNode); Attempts to insert a word into the hash table given a word and position and returns a true upon a success and a false otherwise, Increments frequency if the word is already in the hash table. Increments count of total unique elements in the hash table when inserting a new word.

bool isEmpty(ListNode\* elem); returns a true if there is an open spot in the hash table and returns a false otherwise

void collisionFix(ListNode\*& position); resolves collisions using chaining.

bool format(string& word); Reformats the input word by removing punctuations and setting all letters to lower case returns a false if there was no word to retrieve and returns a true if it retrieved a word

bool filter(string word); returns a true if word is on a common word and returns a false otherwise

void sort(ListNode\*& list);Sorts the list of words in non-ascending order by frequency

void swap(ListNode& elemA, ListNode& elemB);Swaps the values of elemA and elemB

ListNode \*array;

int size; used to keep track of the size of the hash table

int count; used to keep track of total unique elements in the hash table,

**Conclusion**

Provides the functionality of a Hash Table. This includes constructing the Hash Table, deleting the Hash Table, inserting into the Hash Table , and finding if an element is in the Hash Table. The hashing function, and collision handling are private functions used by the insert function. The collsion handling is done by chaining. There is also functionality to print the hash table to the screen and to output a certain amount of data ordered in non-ascending order by frequency to a file.The copy constructor and the overloaded assignment operator are suppressed.

**Source code**

1. HashTable.h

#pragma once

using namespace std;

#include <string>

#include <iomanip>

#include <iostream>

#include <fstream>

class hashTable

{

public:

hashTable(unsigned s = 650);

~hashTable();

void insert(string word);

bool find(string word);

void print();

void outputData(ofstream& outFile, int listSize);

private:

struct ListNode

{

string data;

int freq;

ListNode \*next;

};

hashTable(const hashTable& h);

int hashKey(string word);

bool insertElem(string word, ListNode\* entryNode);

bool isEmpty(ListNode\* elem);

void collisionFix(ListNode\*& position);

bool format(string& word);

bool filter(string word);

void sort(ListNode\*& list);

void swap(ListNode& elemA, ListNode& elemB);

ListNode \*array;

int size;

int count;

};

2.HashTable.cpp

#include "stdafx.h"

#include "HashTable.h"

hashTable::hashTable(unsigned s)

{

size = s;

count = 0;

array = new ListNode[size];

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

array[i].freq = 0;

array[i].next = NULL;

}

}

hashTable::~hashTable()

{

ListNode \*head;

ListNode \*del;

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

head = array[i].next;

del = head;

while (head != NULL) {

head = head->next;

delete del;

del = head;

}

}

delete[] array;

}

void hashTable::insert(string word)

{

if (format(word)) {

if (!filter(word)) {

ListNode \*position = &array[hashKey(word)];

while (!insertElem(word, position))

collisionFix(position);

}

}

return;

}

bool hashTable::find(string word)

{

ListNode \*position = &array[hashKey(word)];

while (!isEmpty(position) && position->data != word)

collisionFix(position);

return !isEmpty(position);

}

void hashTable::print()

{

ListNode \*traverse;

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

traverse = &array[i];

if (!isEmpty(traverse)) {

while (traverse != NULL) {

cout << traverse->data << " " << traverse->freq

<< "\n";

traverse = traverse->next;

}

}

}

}

void hashTable::outputData(ofstream& outFile, int listSize)

{

ListNode \*traverse;

ListNode \*list = new ListNode[count];

int j = 0;

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

traverse = &array[i];

if (!isEmpty(traverse)) {

while (traverse != NULL) {

list[j] = \*traverse;

j++;

traverse = traverse->next;

}

}

}

sort(list);

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

outFile << list[i].data << " " << list[i].freq << "\n";

}

int hashTable::hashKey(string word)

{

const char ZERO = 'a';

int num = 0;

for (unsigned i = 0; i < word.length(); i++)

if (word.at(i) > ZERO)

num += (int)word.at(i) - (int)ZERO;

num += word.length();

num %= size;

return num;

}

bool hashTable::insertElem(string word, ListNode\* entryNode)

{

if (isEmpty(entryNode)) {

entryNode->data = word;

entryNode->freq = 1;

entryNode->next = NULL;

count++;

return true;

}

else if (entryNode->data == word) {

entryNode->freq++;

return true;

}

return false;

}

bool hashTable::isEmpty(ListNode\* elem)

{

return elem->freq == 0;

}

void hashTable::collisionFix(ListNode\*& position)

{

if (position->next != NULL)

position = position->next;

else {

position->next = new ListNode;

position = position->next;

position->freq = 0;

}

}

bool hashTable::format(string& word)

{

const char LOWERBOUND = 'A';

const char UPPERBOUND = 'z';

const char MIDLOWER = 'Z';

const char MIDUPPER = 'a';

char \*tmp;

int j = 0;

int start = 0;

int end = word.length() - 1;

while (start < (int)word.length() &&

(word.at(start) < LOWERBOUND || word.at(start) > UPPERBOUND ||

(word.at(start) > MIDLOWER && word.at(start) < MIDUPPER)))

start++;

while (end >= 0 &&

(word.at(end) < LOWERBOUND || word.at(end) > UPPERBOUND ||

(word.at(end) > MIDLOWER && word.at(end) < MIDUPPER)))

end--;

if (start > end)

return false;

tmp = new char[end - start + 2];

for (int i = start; i <= end; i++) {

tmp[j] = tolower(word.at(i));

j++;

}

tmp[end - start + 1] = '\0';

word = tmp;

delete[] tmp;

tmp = NULL;

return true;

}

bool hashTable::filter(string word)

{

const string FILTERWORDS = "filter.txt";

bool check = false;

string common;

ifstream inFile;

inFile.open(FILTERWORDS.c\_str());

while (!check && inFile >> common) {

if (word == common)

check = true;

}

inFile.close();

return check;

}

void hashTable::sort(ListNode\*& list)

{

int max;

int replace;

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

max = list[i].freq;

replace = i;

for (int j = i; j < count; j++) {

if (list[j].freq > max) {

max = list[j].freq;

replace = j;

}

}

swap(list[i], list[replace]);

}

}

void hashTable::swap(ListNode& elemA, ListNode& elemB)

{

ListNode tmp = elemA;

elemA = elemB;

elemB = tmp;

}

3. main.cpp\*

#include "stdafx.h"

#include <iostream>

#include "hashTable.h"

#include <cstring>

#include <iomanip>

#include <fstream>

using namespace std;

const string inF = "C:\\Users\\nobodypunk\\Desktop\\C++\\HashTable\\a.txt";

const string outF = "C:\\Users\\nobodypunk\\Desktop\\C++\\HashTable\\b.txt";

const int LISTSIZE = 5;

bool fileRead(hashTable& table)

{

int count = 0;

string data;

ifstream inFile;

inFile.open(inF.c\_str());

while (inFile >> data) {

table.insert(data);

count++;

}

inFile.close();

return (count > 0);

}

void fileWrite(hashTable& table)

{

ofstream outFile;

outFile.open(outF.c\_str());

table.outputData(outFile, LISTSIZE);

outFile.close();

}

int main()

{

hashTable testData;

if (fileRead(testData)) {

cout << "The data has been read from the file." << endl;

cout << "Press <Enter>";

cin.get();

fileWrite(testData);

cout << "The data has been analyzed and put " << LISTSIZE << " most common words into a file named " << outF << endl;

}

else

cout << "Error: No Data was Read! \n";

return 0;

}

**References**

<https://www.google.ru/>

<http://pumpkinprogrammer.com/2014/06/21/c-tutorial-intro-to-hash-tables/>