Lab 5: HashMaps

Creating text in an author's voice

100 pts**, Due Nov 8 midnight**

NOTES: This lab can be challenging conceptually (more so than in terms of programming). Please leave time to ponder exactly what you are doing and why. DO NOT start this lab at the last moment.

You may, of course, work with a partner, or you may choose to work alone.

HashMaps

For this lab you will be writing your own hash maps. You will be responsible for:

- *controlling the array size of the map,*
- *the hashing function,*
- *how to handle collisions, and*
- *rehashing when the array becomes too full*

The hashMap implements the ADT sets. In our case, the hashmap we will be working with is storing and quickly accessing a ***word, and the set of words that follow that particular word*** *in text documents.*

- *The* ***key*** *we will be using is the* ***word****.*
- *The* ***values*** *associated with that word* ***are all the words that directly follow that word*** *in a document.*
- *The Node to be stored in the hashMap will be the word, an array of words that follow that word in the text, and the number of words that follow that word.*

There are a number of reasons why we would want to keep track of the set of potential words that follow a particular word in text documents. One reason is for **word prediction**. While we can predict the word you're currently typing based purely on the letters you've typed so far, successful prediction increases when you keep track of the set of words that are likely to follow a particular word and limit your predictions to that set of word. Another reason is for speech recognition – again, if we know what word an individual has just said, we have a better chance of successfully predicting the current word if we can limit (or at least favor) words we know are likely to follow the previous word.

For this lab we'll be using this information (i.e., a word and set of following words) to **generate text in a particular "voice".** By voice, I mean the patterns with which different authors use words. Every author (and speaker) has their own unique word patterns. For this example, we'll be emulating Dr. Seuss' voice, although you could easily attempt to emulate Poe, Dickens, or Shakespeare (I've included some Dickens' text as well, and when you get this working, feel free to download some Shakespeare texts and try to generate brand new Shakespeare novels).

More specifically, for this assignment, you will be reading in a text file of Dr. Seuss stories. Each word read in will be a key, and the values associated with each key will be an array of Strings, or the set of words that follow a word.

So, for instance, if you have the **word "I" as your key**, the **array of values** might be a list that would look like **{"do","see","have","am","do","need"}**, etc. The value that is associated with the key "I" is every word in the Dr. Seuss text that follows the word "I".

So Part A would be reading in the Dr. Seuss text into a HashMap, with the keys being the words in the file, and the values associated with each key are the words that follow a particular word throughout the text. More specifically, as you read in the file, the current word you are reading in is first added to the value set of the previous word you read in, which is the key. The current word then becomes the key and you read in the next word, which is the value of that word.

You are responsible for adding values to your key/values node, and, if necessary, increasing the size of the array of values and copying them over. The hashNode class also has a method that will return a random word from the array of values if there are values, and, if not, returns an empty string.

You are responsible for creating the hash map as well – you will be choosing the array size.

In addition, you will be responsible for **writing 2 separate hash function methods** that take a key (in this case a string), and uses a hash function to change that key to a particular index. You can make up 2 hashing functions, as long as they're not ridiculously bad (e.g., hf("anywordatall") = 1). Make sure you CLEARLY document and explain the hash functions you wrote. You will be comparing the two methods by using a field that keeps track of the original collisions (i.e., when there is a collision the very first time you try to insert a key into the array. Note that this is separate from the collisions that occur based on the method of handling collisions you use).

You are required to write 2 hash functions so you can compare their efficiency. Hence the count of the number of collisions that occur with each hashing function you wrote

You will also be responsible for dealing with collisions. For this you will be writing two methods that finds the new index if the original hash function returns an index that is already occupied by a node with a key that isn't the one you are inserting. You will be comparing these two methods by using a separate field to keep track of the secondary collisions (those that happen as a result of the probing, as opposed to those that resulted from the original hash function). For these methods you can use chaining, linear probing, quadratic probing, pseudo-random probing, double-hashing, or any other method you come up with. Make sure to CLEARLY document and explain the methods you chose.

You are required to write 2 collision functions so that you can compare how efficient each of the collision functions is. Hence the count of collisions that occur using each of your collision functions.

You will also be responsible for adding to the array of values if the node does contain the key you are attempting to insert. And you will be responsible for rehashing if the map array becomes over 70% full

Once you have created the array, you should be able to run the writeFile() method. The writeFIle method will take a key word, choose a random word from the array of values that follows that word, print that word to the file, and make that word be the new keyword. Continue this for a count of maybe 500 or until the value returned is ""

Now read your new document. Did you create a new Dr. Seuss book?

Note : Leave punctuation in. So, in other words, "end" and "end." are two different strings that should be added separately to a value set. This makes the resulting newly created file marginally more readable because it will include some punctuation. If you want, you can modify this function so that a capitalized version of a word and a lower-case version would be considered the same word.

Favorite quotes from mine (there's true wisdom in some):

- the Whos lay asnooze when he snarled with me.
- Today is too, too slow.

- They trembled. They shook. But those wild screaming beaches, just go right along.
- bump! thump! bump! down in a goat!
- Life's a mistletoe wreath.
- “ALL the sound sounded merry! It could not mind at all.”
- I'm the Thneed I do with Thing Two fish Red fish to people as bees
- “Then he whiffed. He lurked in this tree up! Then he spoke with a house.”

At the bottom of this document, I’ve included an output from my code, but since we’re using random values, yours will not look exactly like mine (same general idea, but if it’s identical, the random number generator is just sad).

Outline:

hashNode.hpp: for the nodes to be inserted into the hashmap. The key used for the hash function is the keyword. Everything else in the node is what we want stored with the keyword.

```
class hashNode {
        friend class hashMap;
        string keyword;   // this is used as the key – each key is only inserted once!
        string *values;   // the dynamically allocated array (not linked list!) of words that follow the key in the Dr. Seuss text.
If this set of words gets to be larger than the valuesSize of the array, then you must re-allocate the array double in size and copy
over all the old values
        int valuesSize;  // the size of the values array
        int currSize;  // the number of words in the values array so far
public:
        hashNode(); //constructor-initializes keyword to “” , valuesSize to 0, and currSize to 0 and values to NULL
        hashNode(string s);  // initializes keyword to s, the valuesSize to 100 (or whatever you like for starting), the currSize to
0, and the values to be a dynamically allocated array of valuesSize
        hashNode(string s, string v);  // in addition, puts a value in the values array and initializes currSize to 1
        void addValue(string v);  // adds a new value to the end of the values array, increases currSize, checks to make sure
there’s more space, and, if not, calls dblArray()
        void dblArray(); //creates a new array, double the length, and copies over the values.  Sets the values array to be the
newly allocated array.
        string getRandValue();  // returns a random string from the values array.  If there’s no values in the value array, then it
returns an empty string.
};
/*************************************************************************************/
```

hashMap.hpp: This is everything associated with storing and retrieving from a hash map. Our hashMap will store and retrieve hashNodes, based on the key being the keyword in the hashNode. We’re using a hashMap because when we store words, we want to be able to tell quickly whether the word is already in the map and, if so, add to the values, and, if not add to the array. We also want to be able to quickly find a particular keyword in the hashMap to find the words that follow it, so we can get a random one, print it out into a text file, and then quickly find that word in the hashMap.

```
#ifndef HASHMAP_HPP_
#define HASHMAP_HPP_

#include "hashNode.hpp"

class hashMap {
        friend class makeSeuss;
        hashNode **map;  //a single dimensional dynamically allocated array of pointers to hashNodes
        /****************************************************************************************
```

```
	/* NOTE HERE: the map is a dynamically allocated array of nodes.  Meaning it's a pointer to an array of
	/* pointers to nodes.  In this case it is not a matrix.  It is just a pointer to an array, and the array happens
	/* to be of pointers (aka addresses).
	/*
	/* I did this so that I could move the addresses around without having to recreate
	/* nodes each time I rehashed.  It looks complicated, but it's not.  We've seen this.
	/*
	/* To make the map, you'll do something to the effect of:
	/*		map = new hashNode*[mapSize];
	/* and then you can either set map[i] = NULL or map[i] = new hashNode(k,v);
	/*
	/* Make sure you originally set every address in the map array to NULL when you first create the array
	/* on the heap.  Make sure you re-set the array to NULL when you rehash.
	/*******************************************************************************
	string first; // for first keyword for printing to a file
	int numKeys;
	int mapSize;
	bool h1; // if true, first hash function used, otherwise second hash function is used
	bool c1; //if true, first collision method used, otherwise second collision method is used.
	int collisionct1;  //count of original collisions (caused by the hashing function used)
	int collisionct2; //count of secondary collisions (caused by the collision handling method used)
public:
	hashMap(bool hash1, bool coll1);  // when creating the map, make sure you initialize the values to NULL
so you know whether that index has a key in it or not already.  The Boolean values initialize the h1 and the c1
boolean values, making it easier to control which hash and which collision methods you use.
	void addKeyValue(string k, string v);
	// adds a node  to the map at the correct index based on the key string, and then inserts the value into the
value field of the hashNode
	// Must check to see whether there's a node at that location.  If there's nothing there (it's NULL), add the
hashNode with the keyword and value.
	// If the node has the same keyword, just add the value to the list of values.
	//If the node has a different keyword, keep calculating a new hash index until either the keyword matches
the node at that index's keyword, or until the
	// map at that index is NULL, in which case you'll add the node there.
	//This method also checks for load, and if the load is over 70%, it calls the reHash method to create a new
longer map array and rehash the values
	//if h1 is true, the first hash function is used, and if it's false, the second is used.
	//if c1 is true, the first collision method is used, and if it's false, the second is used
	int getIndex(string k); // uses calcHash and reHash to calculate and return the index of where the keyword
k should be inserted into the map array
	int calcHash(string k);  // hash function
	int calcHash2(string k);  // hash function 2


	void getClosestPrime();  // I used a binary search on an array of prime numbers to find the closest prime
to double the map Size, and then set mapSize to that new prime.  You can include as one of the fields an array of
prime numbers, or you can write a function that calculates the next prime number.  Whichever you prefer.
	void reHash();  // when size of array is at 70%, double array size and rehash keys
	int collHash1(int h, int i, string k);  // getting index with collision method 1 (note – you may modify the
parameters if you don't need some/need more)
	int collHash2(int h, int i, string k);  // getting index with collision method 2 (note – you may modify the
parameters if you don't need some/need more)
```

```
                    int findKey(string k);  //finds the key in the array and returns its index.  If it's not in the array,
returns -1
        void printMap();  //I wrote this solely to check if everything was working.  Its only purpose is for testing.

};

/*******************************************************************************/
```

makeSeuss.hpp: This file is largely responsible for reading in the text file into a hash map, and then writing out a new file of computer-generated Dr. Seuss text.

```
#ifndef MAKESEUSS_HPP_
#define MAKESEUSS_HPP_

#include "hashMap.hpp"
#include <iostream>
using namespace std;

class makeSeuss {
        hashMap *ht;
        string fn;  // file name for input (“DrSeuss.txt”)
        string newfile;  // name of output file
public:
        makeSeuss(string file,string newf,bool h1, bool c1);
        void readFile();
        void writeFile();

};
/***************************************/
```

makeSeuss.cpp: This is the definitions for all the methods in the makeSeuss.hpp.

The **constructor (written below)** constructs a makeSeuss object by setting the name of the file to be read in from (e.g., DrSeuss.txt), setting the output file’s name (maybe Seussout.txt), setting true for using hash1 function, false for using hash2 function, and then true for using collision1 function and false for using collision2 function

The **readFile** (written below) reads text from the file specified as f1 into the hashMap

The **writeFIle** (written below) uses the hashMap to write out new text into a file specified as f2.

```
/******************************/
#include "makeSeuss.hpp"
#include "hashMap.hpp"

#include <iostream>
#include <stdlib.h>
#include <string>
#include <fstream>

using namespace std;

makeSeuss::makeSeuss(string f1,string f2,bool h1, bool c1) {
        ht = new hashMap(h1,c1);
```

```
		newfile = f2;
		fn = f1;
		readFile();
		writeFile();
}
void makeSeuss::readFile() {
		ifstream infile(fn.c_str(),ios::in);	// open file
		string key = "";
		string value= "";
		infile>> key;
		ht->first = key;
		while (infile >> value) {		// loop getting single characters
			cout << key <<": " << value << endl;
			ht->addKeyValue(key,value);
			key = value;
			value = "";
		}
		ht->addKeyValue(key,value);
		cout << endl;
		infile.close();
}
void makeSeuss::writeFile() {
		ofstream outfile(newfile.c_str(),ios::out);

		outfile << ht->first << " ";
		string key = "";
		string value = ht->map[ht->getIndex(ht->first)]->getRandValue();
		int ct = 0;
		int len = 0;
		while (ct < 500 &&  value != "") {
			key = value;
			outfile << key << " ";
			if (len == 11) {
				outfile << "\n";
				len = 0;
			}
			else len++;
			value = ht->map[ht->getIndex(key)]->getRandValue();
			ct ++;
		}
		outfile.close();
}


#endif /* MAKESEUSS_HPP_ */

/*********************************************************************************/
```

Lab7main.cpp: - you'll have to make the main. Make sure you **seed your random number generator** or your output file will look the same every time you run it. To test this, run the code more than once and make sure you don't end up with the same output file each time. They should start with the same word, but then be notably different each time. **Make sure your main makes and writes out 4 different files** – one that uses the first hash function and the first collision method, One with the first hash function and the second collision method, one with the second hash function and the first collision method, and one with the second hash function and the second collision method.

/***/

To turn in:

Your code: zip together 7 separate code files, 4 output files, and one comment file

- HashNode.hpp and cpp,
- HashMap.hpp and cpp,
- MakeSeuss.hpp and cpp, (I wrote these) and
- Lab7main.cpp

Grading:

NOTE THAT IF YOUR FINDKEY METHOD HAS A LOOP IN IT THAT GOES THROUGH EVERY VALUE IN THE ARRAY LOOKING FOR A KEY, YOU WILL LOSE 50%. That negates the whole value of hash maps!

- (25%) Reading in keys and values into hashmap (printed out using printMap()

In order to compare hash functions and collision handling methods, you should have:

- (32%) 16% for each of the 2 hash functions run to create separate output files (clearly label with: A) which of the hashing functions was used, and B) which of the collision methods was used
    - *(Note that you can use the same collision handling method for both hash functions in this case – in other words, if you get the program to work to completion, you've got at least one hash function and one collision function working)*
- (32%) 16% for each of the collision handling methods run to create separate output files (clearly label with: A) which of the hashing functions was used, and B) which of the collision methods was used
    - *(Note that you can use the same hashing function for both collision handling methods in this case – in other words, if you get the program to work to completion, you've got at least one hash function and one collision function working)*
- (11%) In addition, you must turn in a separate file containing the number of original collisions for each of the two hash methods, and the number of post collisions for each of the two collision dealing methods. In this file, also list at least 6 of your favorite resulting phrases created.

Extra Credit (15 pts):

In this hashmap, the keys are single words. Word prediction on your phone uses both the keys you are typing in and the previous word to predict the current word. However, word prediction can improve significantly if we take into account the 2 previous words.

For extra credit, modify your code (save your old code, and make a new version) so that the key is 2 words instead of one, and the value(s) are the words in the document that follow a 2 word pair. Be aware that 2-word keys will often result in only 1 possible value. The more text you start with, the more likely it is that there will be more than one value associated with a 2-word key. Thus, you may want to modify the Seuss text file to include even more Dr. Seuss books.

My Output:

```
Congratulations! Today is perfectly true. But McBean Invited them here to boil, or
```

two. They run for the South! To the house? Would brag, We're
all Sala-ma-Sond, Yertle the cat. 'take a hole in the smog you've
paid him step in your money was wrong. For, just at all!'
and he hated! The Plain-Belly sort! And the house and with a
look!' and worried and he said, "Listen here! Here's a quick call.
I do not like them, Sam-I-am. I do not like them, Sam-I-am.
I now rule! For he cried. "Oh, the Mud. That day, on
the Who girls and a car! You will they will take them
out of a groan from here and glad and Thing One fish
said, 'look! look!' then he sent her to us why. No more
the ball. with his Grinch simply MUST hear!" So be sure when
they want to the tall and early. They'd stand hand-in-hand. And in
the last of brains in the Turtle was what to come with
a box. a month... or there. I rushed 'cross the clouds! Over
land! Over land! Over land! Over sea! There's nothing, no, NOTHING, that's
how they should not like to us jump! we two. They were
too slow. Some are starving!" groaned Mack. "You stay in the Turtle
King, lifted his head of a dish! and he makes his grinch-feet
ice-cold in the fish in a goat. And why was two Things
really so good, so could be, with a Yes or waiting, perhaps,
that thing one was shouting, he was up.) "Let me be! I
rule from that case, of all. But, now, how much can you
choose. You're off to shove, When the air and BIGGERING and he
climbed to go away. What a mind-maker-upper to eat. The ribbons! It
could see them with a glove, It's a pat. 'they are for
their future is not a car. Not in the things they paraded
about it? now, what you up with great speedy speed, I now
was quite wrong. For, just waiting. Waiting for miles cross weirdish wild
space, headed, I do! And, what's more, beyond that. I'm sending them
with a Sneetch! But that scare you have fun.' then, out of
all hung in a tree. I would she will stuff up his
teeth sounding gray, "how the Whos, still wet day.' now, what I'll
do not like them away. Never let out of joy in the
fish said, 'how do NOT wish I would sit there that cat
in the strangest of the top of his cave with his head!
he grunts, "I speak for a doubt. The wrappings! The Truffula Trees
at his hand to order and i can you need 'bout five
thousand, six feet And the worst. But, now, how in the waiting
around for much greater I'd be scored. There are the hat you
eat them with sally. we can do! And, under the day. but
your problems whatever they may. Good grief! groaned