Hash Map Project

100 pts, ***(due Thurs, Dec 10)***

Time: I expect this project to take about 8 hours. The hardest thing about this project is wrapping your mind around how hashmaps work, but the programming shouldn't be horrible.

This project should be done in multiple sittings over the course of the assigned 2-week period.

I hope you have fun with this project!

Contents

HashMaps....1
Key-Values....2
Voice Generation: 2
Examples: 2
Favorite quotes from mine (there's true wisdom in some): 3
Grading: 3
NOTE THAT IF YOUR FINDKEY METHOD HAS A LOOP IN IT THAT GOES THROUGH EVERY VALUE IN THE ARRAY starting at 0 to find a KEY, YOU WILL LOSE 50%. That negates the whole value of hash maps! 3
In hashNode.cpp, you're writing: 4
void hashNode::addValue(string v); /*(2 pts) */ 4
void hashNode::dblArray(); /* 4 pts) */ 4
string hashNode::getRandValue(); /*(2 pts)*/....4
In HashMap.cpp, you're writing: 4
hashMap(bool hash1, bool coll1); // (3 pts) 4
void addKeyValue(string k, string v);/* (6 pts) */ 4
int getIndex(string k); // (6 pts)....5
int calcHash1(string k); // (7 pts) 5
int calcHash2(string k); // (7 pts) 5
void getClosestPrime(); // (3 pts)....5
void reHash(); // (6 pts)....5
int coll1(int h, int i, string k); // (7 pts) 5
int coll2(int h, int i, string k); // (7 pts) 5
int findKey(string k); // (8 pts)....5
Four Output Files (24 pts, 6 pts each): 5
A word or text document with some chosen favorite quotes from your output: (8 pts) 5
To turn in: 5
Extra Credit (15 pts): 5

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*

Key-Values

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 goes up 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.

Voice Generation:

For this lab we'll be using this 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 has their own unique word patterns. For this example, we'll be emulating Dr. Seuss' voice and Charles Dickens' voice, although you could easily attempt to emulate Poe, Shakespeare, or even J.K. Rowling (when you get this working, feel free to download some Shakespeare texts and try to generate brand new Shakespeare novels).

Examples:

More specifically, for this assignment, you will be reading in a text file of Dr. Seuss stories first and the first 2 chapters of Great Expectations. 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 you'll be reading in the Dr. Seuss text and Charles Dickins 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 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 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 ""

You will be writing 4 separate files – one with hash function A and collision function A, one with hash function A and collision function B, one with hash function B and collision function A, and one with hash function B and collision function B. I've used Boolean values to indicate whether to use Hash function A or B and a separate Boolean value to indicate whether to use Collision function A or B.

Now read your new documents. Did you create a new Dr. Seuss book? A new Charles Dickens novel?

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):

- My name is no fear. have no fear, little car.
- I do not be about.
- Today is too, too slow.
- We were all got terribly mad.
- But down long as Yertle, the toy ship, sank it with the morning, he said,
- Life's a mistletoe wreath.
- I will stuff up the waiting for three ninety-eight
- My first fancies regarding what broken bits of the achievement of laying it was possible
- it in mortal terror of mincemeat
- I was in the unconscious Joe.

I've included in the zip file my output files from the four tests, 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).

Grading:

NOTE THAT IF YOUR FINDKEY METHOD HAS A LOOP IN IT THAT GOES THROUGH EVERY VALUE IN THE ARRAY starting at 0 to find a KEY, YOU WILL LOSE 50%. That negates the whole value of hash maps!

In hashNode.cpp, you're writing:

void hashNode::addValue(string v); /*(2 pts) */

// adding a value to the end of the value array associated
// with a key

void hashNode::dblArray(); /* 4 pts) */

// when the value array gets full, you need to make a new
// array twice the size of the old one (just double, no
//going to next prime) and then copy over the old values
//to the new values, then de-allocate the old array.
//Again, just copying over, no hash functiosn involved
//here.

string hashNode::getRandValue(); /*(2 pts)*/

//Every key has a values array - an array of words that
// follow that key in the text document. You're going to
//randomly select one of those words and return it. That
//will be the word that follows your key in your output
//function, and it will also be the next key.

In HashMap.cpp, you're writing:

hashMap(bool hash1, bool coll1); // (3 pts)

//when creating the
//map, make sure you initialize the values to
//NULL so you know whether that index has a key
//in it or is set to NULL

void addKeyValue(string k, string v);/* (6 pts) */

// 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 already 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

```
int getIndex(string k); // (6 pts) uses calcHash and reHash to
                        // calculate and return the index of where
                        // the keyword k should be inserted into the map
                        // array
int calcHash1(string k);  // (7 pts) hash function
int calcHash2(string k);  // (7 pts) hash function 2
void getClosestPrime();  // (3 pts) I used a binary search and an
                        //array of primes to find the closest prime to
                        //double the map Size, and then set mapSize to
                        //that new prime - you can find the prime in
                        //another way if you choose
void reHash();  // (6 pts)  when size of array is at 70%, double
                        //array size and rehash keys
int coll1(int h, int i, string k);  // (7 pts) a probing method
                        //for collisions (when index is already full)
int coll2(int h, int i, string k);  // (7 pts) a different method
                        //for dealing with collisions
int findKey(string k);  // (8 pts)  finds the key in the array and
                        //returns its index.  If it's not in the
                        // array, returns -1
                        //NOTE: THIS MAY NOT LOOP IN THE SENSE THAT YOU
                        //START AT INDEX 0 AND LOOP THROUGH EVERY INDEX
                        //LOOKING FOR THE KEY.  IF YOU DO THAT YOU'RE
                        // MISSING THE WHOLE ENTIRE POINT OF HASH
                        // FUNCTIONS!!!!!
```

Four Output Files (24 pts, 6 pts each):

There should be 4 output files – for all combinations of the hash function/collision handling techniques.

A word or text document with some chosen favorite quotes from your output: (8 pts)

At least 2 or 3 quotes from each of the generated text documents

To turn in:

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

- HashNode.hpp and cpp,
- HashMap.hpp and cpp,
- MakeSeuss.hpp and cpp, (I wrote these) and
- mainHash.cpp
- the 4 output files: Seussout.txt, Seussout2.txt, GeChap1out.txt,and GEChap1out2.txt
- the word or text file with favorite examples from your generated files

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.