Hash Map Project

100 pts, ***(Part 1 due Fri, April 14, Part 2 due Fri,Apr 21)***

You may work with a partner or you may work alone. If you choose to work with a partner, make sure:

- you both turn in the project
- you include both names on the project.
- Equally, note your partner's name in canvas.

Please be aware that if your partner flakes on you, you are still responsible for completing the mini project and turning it in on time.

Time: This project may take about 16 hours, and should be worked on in a spread-out manner, as opposed to one sitting.

★★★★☆In terms of difficulty, I consider this to be a 4 out of 5

This miniproject is closely tied with the week 9 videos and pptx on my web site (https://www.eecis.udel.edu/~yarringt/CISC220/)

Technically this lab is worth 103 points. So. 3 pts extra credit already built in for completing it.

I hope you have fun with this project!

Part 1 (15 pts), due Fri, Apr 14)

You must come up with your own testing plan.

Note that in this case, I CANNOT give you expected output for your methods since you will each be using different hashing methods and different collision handling methods.

For this lab you will be writing the following methods:

In the HashNode Class:

- *void hashNode::addValue(string v); /* – this will need to be written early on! */*
- *void hashNode::dblArray(); /* – probably don't need to write early on */*
- *string hashNode::getRandValue(); /* – can be written close to last */*

In the HashMap Class:

- *hashMap(bool hash1, bool coll1); /* – needs to be written early */*
- *void addKeyValue(string k, string v); /* – has to be written early */*
- *int getIndex(string k); /* – has to be written early */*
- *int calcHash1(string k); /* s – one of the hashing functions needs to be written early */*
- *int calcHash1(string k); /* – again – one of the hashing funcions needs to be written early */*
- *void getClosestPrime(); /* – this can be written later on because it's used when you're rehashing your entire table */*

- *void rehash(); /* – again – this can be written later on, especially if you set your map to be relatively large to start with */*
- *int coll1(int h, int i, string k); /* – like the hashing functions, one of the collision functions needs to be written relatively early on */*
- *int coll2(int h, int i, string k); /* – see above collision function */*

The first thing you should do is come up with a plan of attack. What is the order in which you intend to write these methods?

Next, given the following input (it's a subset of the Seuss text file), what would you expect to be the output for various methods when you test them?

Yes, this is vague. Yes, this seems like a weird thing to ask you. But what I want to see is that you have a plan for how you are going to approach writing this project, and for how you are going to test the code you write for this project.

At the absolute minimum, you should draw out what a possible hash map would look like with the test text below. What would be the keys, and what would be the values associated with each key? How full is the hashmap you created (what is the load factor)? Is it too full or too empty?

AND – hand generate some random numbers and then generate potential output. IF you use a random number generator online (or write your own), what would a small set of your output look like?

Testing input file:

WOULD YOU LIKE GREEN EGGS AND HAM?

I DO NOT LIKE THEM, SAM I AM.
I DO NOT LIKE GREEN EGGS AND HAM.

WOULD YOU LIKE THEM HERE OR THERE?

I WOULD NOT LIKE THEM HERE OR THERE.
I WOULD NOT LIKE THEM ANYWHERE.
I DO NOT LIKE GREEN EGGS AND HAM.
I DO NOT LIKE THEM, SAM I AM.

WOULD YOU LIKE THEM IN A HOUSE?
WOULD YOU LIKE THEN WITH A MOUSE?

I DO NOT LIKE THEM IN A HOUSE.
I DO NOT LIKE THEM WITH A MOUSE.
I DO NOT LIKE THEM HERE OR THERE.
I DO NOT LIKE THEM ANYWHERE.
I DO NOT LIKE GREEN EGGS AND HAM.
I DO NOT LIKE THEM, SAM I AM.

/***/

Part 2:

Now you will be writing the following methods (with description of what each is below this list of methods):

In the HashNode Class:

- *void hashNode::addValue(string v); /* (2 pts) */*
- *void hashNode::dblArray(); /*3 pts */*
- *string hashNode::getRandValue(); /* (2 pts) */*

In the HashMap Class:

- *hashMap(bool hash1, bool coll1); /* (3 pts)*/*
- *void addKeyValue(string k, string v); /*(7 pts) */*
- *int getIndex(string k); /* (7 pts) */*
- *int calcHash1(string k); /* (7 pts) */*
- *int calcHash1(string k); /* (7 pts) */*
- *void getClosestPrime(); /*(3 pts) */*
- *void rehash(); /* (8 pts) */*
- *int coll1(int h, int i, string k); /* (7 pts) */*
- *int coll2(int h, int i, string k); /*(7 pts) */*

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.*

Why words as key-value pairs?

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 (our project):

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 or Charles Dickens' voice, although you could easily attempt to emulate Poe, Shakespeare, J.R.R Tolkien, or any of your other favorite authors.

Examples:

More specifically, for this assignment, you will be reading in a text file of Dr. Seuss stories (or the first 2 chapters of Great Expectations – Great Expectations is LONG!!!). 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".

- Key – "I"
- Values: array of strings ON THE HEAP: ["do","see","have","am","do","need"]

So you'll be reading in the Dr. Seuss text or Charles Dickins text into a HashMap:

Every time you read a word, that becomes 2 things:

- one of the strings in the value array for the word right before it in the text,
- and then it becomes a key itself

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.

Hash Functions:

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.

Why 2 separate hashing functions?

The point is that different data will work better with different hash functions. You are writing two hash functions so **you can compare their efficiency**.

You can make up 2 hashing functions, as long as they're not ridiculously bad (e.g., hf("any_word_at_all") = 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).

Collision Functions:

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.

Why two collision functions?

Again, the reason you are writing two separate collision functions is so you can compare how efficient they are with your data set. 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, double-hashing, or any other method you come up with. Make sure to CLEARLY document and explain the methods you chose.

Resizing Hash Map:

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 are creating 4 separate files:

You will be writing 4 separate files –

1. one with hash function A and collision function A,
2. one with hash function A and collision function B,
3. one with hash function B and collision function A,
4. 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 is 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 STARTS AT 0 AND 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!

Yes, for probing, you will use a systematic loop to find a value if there's a collision. My point is that finding values by starting at 0 and looping through the array until you either find the key or until you reach the end of the array is extremely inefficient and the opposite of the goal of using a hashmap in the first place.

In hashNode.cpp, you're writing:

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

/* In the hashNodes, the data includes a key string and a pointer to an array of strings, which is the
values. It also includes the current size of that array of strings, and the number of strings currently in
that array. This is done so that we know when the array gets full, and when that happens we'll allocate
a longer array on the heap and copy over the values. For me this didn't happen very often. THIS PART IS
NOT THE HASH MAP. I literally just started at index 0 and copied the string values over.

Also these hashNodes DO NOT have a prev or next (or parent, or left, or right) pointer. These are just
nodes that the hashMap points to.*/

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

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

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

void hashNode::dblArray(); /* 3 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
// ****here you're just looping through the map of hashNodes (a pointer to an array of
//pointers to hashNodes) and initializing the array to NULL
```

void addKeyValue(string k, string v);/* (7 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); // (7 pts) uses calcHash and reHash to

```
// calculate and return the index of where
// the keyword k should be inserted into the map
// array.  For me this just got the index of the key word
// and returned that, or, if the key wasn't found, returned
// -1.
//addKeyValue is the method that actually added the key
// and associated value to the hashmap if the key wasn't
//already in the method, or, just added the value if it was,
//etc.
```

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(); // (8pts) 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

Four Output Files (20 pts, 5 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: (5 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: Either Seussout1.txt, Seussout2.txt, Seussout3.txt, Seussout4.txt
  OR Geout1.txt,GEout2.txt, Geout3.txt,GEout4.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 either modify the Seuss text file to include even more Dr. Seuss books, or add some extra Charles Dickens (which is easy because the guy was seriously verbose!)