Programming Assignment 4 - Hash Table

Due: April 18

This is an individual assignment!

You are responsible for adhering to the Aggie Honor Code and student rules regarding original work. You must write and submit your own code to Mimir. Plagiarism checks will be used to check the code in addition to a manual review from the grader.

There is a provided `report.txt` where you will track all of the references you use. You may use references to help you get a high-level understanding of the problem, but remember: *Do not share code! Do not copy code!*

Overview

Hash Tables

We would like to have data structures that support fast search, insert, and delete operations, such as dictionaries and maps. Balanced trees, like 2-4 and AVL trees, have benefits when it comes to providing O(log(n)) time with a natural order that may be used to sort and find nearby elements.

However, if we don't care about sorting operations, there is another structure that provides much better performance for map and dictionary operations: the hash table.

In short, hash tables are just direct-indexed lists (i.e., arrays and vectors) where we use the key to get the index into the list. Unlike the direct addressing methods we saw with count sort, hash tables use hash-based indexing where we apply a *hash function* to the input. Hash functions map an arbitrary length input to a fixed-length, such as a number between 0 and the capacity of a list.

This Assignment

In this assignment, you will implement a polynomial rolling hash function and two types of hash tables capable of handling collisions: chaining and linear probing hash tables.

The hash tables will be used for a word counting application. Given a dictionary of 1 million words (in dictionary.txt), your hash table should be able to count the occurrences of each word; there are a little over 58,000 unique words in the file. The provided runner also generates timing info you should use to answer the report questions.

In the second part, you will practice another interesting application of hashing by utilizing your polynomial rolling hash to create a heavily simplified blockchain-based cryptocurrency.

Getting Started

1. Go to the assignment on Mimir and download the starter code.
    a. There is a `report.txt` file where you should track the references you use for this assignment, as well as provide answers for a couple questions.
    b. The following files are fully implemented and will not need to be edited:
        i. `Hash.h` - header file for the Hash namespace
        ii. `Block.h` - header for the basic blockchain you'll use in part B
        iii. `runner_*.cpp` - two runner programs are provided, one each for testing your chaining and probing hash tables
    c. The parent HashTable class is partly implemented, alongside the beginning of the ProbingHashTable. You will need to complete portions of these files for your code to compile
    d. You must complete the Hash, ProbingHashTable, and ChainingHashTable implementations for part A. Additionally, you will need to implement the Makefile and provide a new .cpp file for the cryptocurrency you name for part B.
2. I recommend implementing the hash function first since it is used for both parts and is a good first step.
3. Next, complete the partial implementations of HashTable.h and ProbingHashTable.h. This will allow you to complete the full Probing and Chaining hash tables.
4. You will need a Makefile for the Mimir program tests. You should also use this locally when you run the provided runner_*.cpp files to get timing info for your hash tables.
    a. The runners already make use of the provided dictionary.txt file for the word counting application.
    b. You should use dictionary.txt for your timing, but you are free to make your own runners and test files for evaluating word counting correctness.
    c. Keep in mind the Makefile is like giving a shortcut to a command line operation. All you need to do to compile one of your runners, such as the ProbingHashTable, is to pass all the source files into g++, as shown below. You should name the target *and the resulting executable* **probing** for the probing hash table and **chaining** for the other.

```
g++ -std=c++17 runner_probing.cpp Hash.cpp ProbingHashTable.cpp
```

5. Name and create your cryptocurrency program for part B.

a. The first step is to provide a **single word name** for your cryptocurrency in `coin_name.txt`. We will only read the first string up to a space. I used RevCoin, but you are welcome to use whatever you wish.

b. Create a <coin_name>.cpp file (e.g., mine was RevCoin.cpp) where you will write your mining program as described in part B.

c. Add a target to your Makefile that **matches** the coin name which yields an executable named the same. Mimir will call this target and executable to test input and output.

6. Be sure to answer the report and track your references!

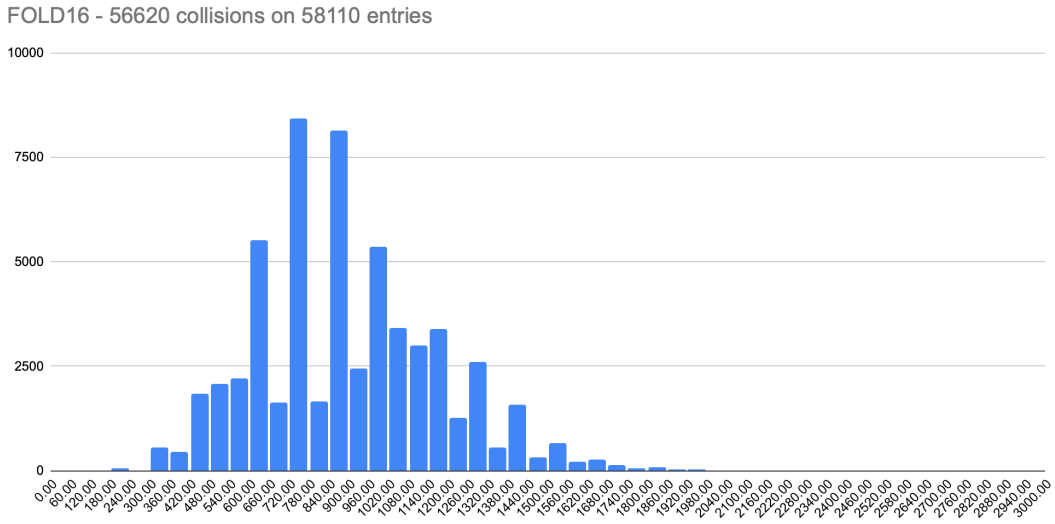
Part A: Hash Tables

Tasks

Implement probing and chaining hash tables that support insert, remove, and get operations. There are several sub-tasks associated with completing this part, which are detailed below.

Hash.cpp

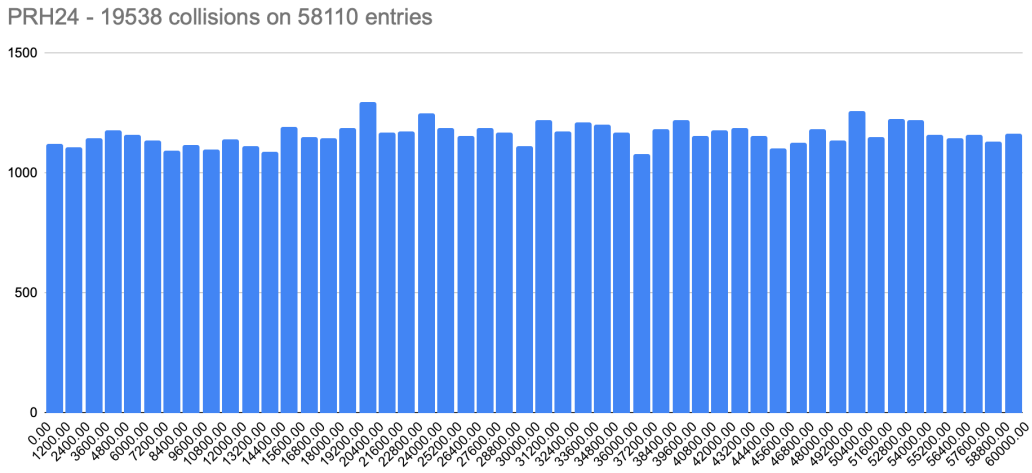
The first step is to write the hash function you will use for all subsequent code. In the lab, you implemented a basic string hash that summed each letter together, using the ASCII code as the numerical value, and applied mod to the final result. This is called a folding hash because you fold all the parts of the input together equally for the hash. Unfortunately, this function does not generate a good distribution of hashes. The image below shows the distribution as a histogram for the words in the dictionary.txt file.



There are many different ways to write hash functions (rotation, rolling, etc.), which is a discussion beyond the scope of this class. However, we can easily write a much better hash function using polynomial rolling. This approach is called rolling because it is similar to applying a sliding window that steps along the data and applies a weighted operation to each character. Specifically, it is defined as below:

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

Where *s[i]* is the i^{th} character of the input string *s* and p^i is a constant raised to the power of *i* and modded by *m*. We prefer to select *p* and *m* such that they are coprime; that is, they are considered relatively prime to each other and share only 1 as a common factor. This has function generates a much nicer distribution with fewer collisions (when capped to a capacity of 60,000 as done in the provided runner files):



I call this function PRH24 because it generates a 24-bit polynomial rolling hash.

Hash.cpp	**Description**
unsigned int PRH24(std::string s);	This function takes in a string and returns an unsigned integer. The operation is as follows: - Use p = 137 and m = 16,777,215 (which is 2^{24} - 1, the largest unsigned 24-bit number) - Start with the hash = 0 - Iterate through each character in the string (recall: string provides iterators with .begin() and .end()!) - Add into the hash the ASCII value of the current character *times* the current power of *p*. Mod this by 16,777,215. - Increase *p* by multiplying it by the base power of 137 *and mod this by 16,777,215 as well.* If you don't mod this value, *p* can overflow and cause erroneous calculations.

HashTable.h

In this assignment, we will practice *inheritance* to implement the different hash tables.

Similar to templating, inheritance is a tool for the programmer that is designed to minimize repeated code. You can write a base class which provides functions and data members that will be used by all the derived classes.

In this case, our base HashTable class will track size and capacity; it also provides the empty() and subscript operator. The other functions are what we call "pure virtual functions" because they are not defined in the base class and will require being defined by the derived classes.

HashTable.h has several TODOs, including templating the class with two templated data types (as seen with the Cell object).

HashTable.h	**Description**
U operator[](T key)	The subscript operator is what enables vectors to index like an array, e.g., *v[10]*. For a hash table, we can use the subscript operator to access by key. For instance, a hash table with strings for keys could be accessed as *h["mystring"]* to get the value corresponding to the "mystring" key. If you recall dictionaries from Python or have used Javascript, this syntax will look very familiar, and it is trivial to allow your hash tables to support it: - Add the declaration shown - The definition is just to return get() using the *key*
virtual void insert(T key, U val)	The three dictionary operations of insert, remove, and get will be pure virtual functions. This means you should add their declarations to the HashTable.h file, but you set them equal to 0, as seen with the pure virtual "hash" function.
virtual U remove(T key)	
virtual U get(T key)	

ProbingHashTable.h

ProbingHashTable.h also has several TODOs. First, you need to modify the class to inherit from HashTable.h. You should use public inheritance, which will keep public members of the base class public in the derived class and allow protected access to the protected members.

This class **is not templated**, so while it will look similar to `class ProbingHashTable : public HashTable<T,U>`, you won't be passing templated types T and U. Instead, you should specify the data type T (the key) will be of type string, and U (the values) will be of type int.

ProbingHashTable.h	**Description**
"table" data member to hold Cells	You'll want to create a data member to hold the actual data of the HashTable. The data type will need to be a list that can hold

	Cell<string,int>. You can use a dynamic array that you allocate one-time in the constructor to the capacity and then delete in the destructor. You may also use a vector for this data, but you'll need to explicitly set the vector size in the constructor so you can index into specific locations during dictionary operations.
ProbingHashTable(); ProbingHashTable(int cap);	You should create the default constructor and one that takes an argument for the capacity. You'll see in the runner files that we use the parameterized constructor to create a hash table with capacity of 60,000
~ProbingHashTable();	The destructor should free any memory you might allocate, for instance if you use a dynamic array for the table data

unsigned int hash(std::string key);
void insert(std::string key, int val);
int remove(std::string key);
int get(std::string key);
void printAll(std::string filename);
	These are the five functions you should implement for both hash tables. They are discussed more below when we discuss the source files for the hash tables, but at this point, you should add the declarations to the header.

ChainingHashTable.h

The ChainingHashTable header will mirror the ProbingHashTable almost exactly. The only difference will be the table structure. In the probing version, you just have a list of cells with keys and values. In the chaining version, each index of the list is actually its own list of cells with keys and values. This allows collisions to be resolved by having the ability to store multiple key,value pairs at the same index.

Effectively, this means only the table structure changes, so you may use a double pointer for a nested dynamic array, a vector of vectors, a dynamic array of vectors, an array of linked lists, or any other means of supporting nested objects like this.

ProbingHashTable.cpp and ChainingHashTable.cpp

When it comes to implementing the hash tables themselves, the overall activities are similar. It is the collision resolution that differs.

You are implementing a **linear probing** hash which just moves to the next index if it finds a collision has occurred. It will move until the next available index.

The chaining hash table just appends the key,value pair into the current index list.

There are other ways of resolving collisions, such as quadratic probing, which can be more efficient in some ways and less so in others. There is also *double hashing*, in which another hash function is applied if a collision has happened. It can perform somewhere between the two with a bit of an advantage when it comes to storage efficiency.

We will be discussing more about linear probing and chaining in class, but the following advice should help you get the core implementation for your hash tables:

- For “hash”, simply call your Hash::PRH24 with the key and mod the result by the capacity (this is to ensure it fits in the array since we are using 60,000 elements as seen in the runner, but the 24-bit hash can yield over 16 million values).
- “get” calls “hash” on the passed key (not PRH24 but the one you just implemented for this hash table that mods by capacity) and then checks if the data exists at the cell in that index. If data is there but not the expected key, the collision will have to be resolved as described above: either moving to the next index in linear probing or searching the sublist in chaining.
    - Because we are focused on a word-counting application, you should return 0 if not found.
- “insert” follows the behavior of “get” but either updates the value or adds a new entry.
- “remove” clears a cell and returns the value of the key; you should throw a runtime_error if the key is not found.
- “printAll” traverses the table structure and prints the key-value pairs.
    - You should use ofstream to open the passed filename
    - Do not print anything on blank cells
    - Write space-separated output to the file, e.g.,
      `file << table[i].key << " " << table[i].value << std::endl;`

Makefile

Once you have begun working on the .cpp files, you will want to implement the Makefile so you can easily test locally. **Mimir will use the Makefile for certain tests!** Additionally, you’ll want to be able to use the provided runners to get timing info for discussion in the report.

As mentioned in the “Getting Started” section, a line in the Makefile is essentially just a command line call. Consider using the provided example from that section to help start your Makefile.

- Create a target called “probing” which builds the runner_probing.cpp file and the ProbingHashTable.
    - Recall that you’ll need to pass Hash.cpp for the linker.
    - The output should also be an executable called “probing”; you can specify the output of g++ with the “-o” flag.
- Create a target called “chaining” which builds the runner_chaining.cpp file and the ChainingHashTable. The resulting executable should also be called “chaining”.

You may add a “clean” target to remove the executables for your own convenience, but Mimir will not use that target.

Point Distribution

This part comprises 72% of the overall assignment score. That is distributed as follows:

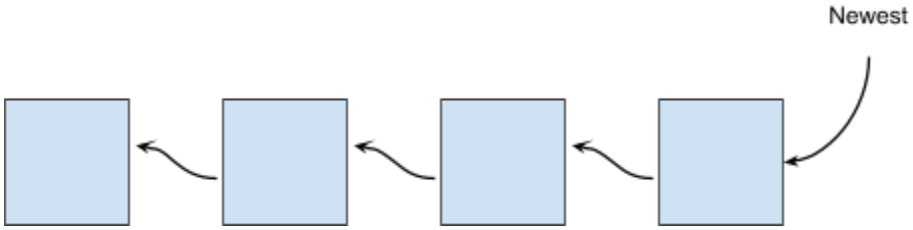
- 8% for Hash.cpp
- 4% for HashTable.h
- 30% for ProbingHashTable (includes word counting application)
- 30% for ChainingHashTable (includes word counting application)

Part B: Simplified Cryptocurrency

Overview

Many modern security and crypto applications utilize hashing in some form. As we discussed in lecture, cryptocurrency is one of the most prominent examples of hashing today. Many forms of this currency exist, each with their own rules and specifications, but at their core, they rely on cryptographic hashes being difficult to reverse to ensure transactional veracity and assign value to the effort of signing blocks in the blockchain.

A blockchain can be thought of as a reversed singly-linked list where we keep only a tail pointer indicating the most recently added block to the list, and every block has a pointer to the previous block in the chain.



Tasks

In this part of the assignment, you will create a small application that is able to read a sequence of transactions from the command line and create a signed blockchain. It is not a true blockchain nor a true currency, and the amount of mining "work" will be quite small, but it will practice the concept.

Make edits for your custom crypto coin

Fill in coin_name.txt
There is a blank file coin_name.txt in the starter code. You must update this to contain the **single-word** name of the coin you are creating. The Mimir test will use this as the name for the makefile target and executable.

Create a new source file, matching the coin's name
Using the same name as specified in coin_name.txt, create a source file for your program that matches. For example, I put `RevCoin` in `coin_name.txt` and created `RevCoin.cpp`.

Add target to Makefile
Add a new target to the Makefile that uses the same name as well. Mimir will call this target to build your program. The target should build your new source file and generate an executable that matches in name; my executable was called `RevCoin`.

Implement the program

Your program will not be unit-tested but rather evaluated as a complete executable. It should match the following criteria:

- Include Block.h and Hash.h so you can make use of the provided Block algorithms.
- In the main program, repeatedly read a line from std::cin until a blank line is given.

    - Each line should be viewed as a transaction and added to a new block. Note that you should pass the previous block pointer into the block along with the transaction since these are set at the time of block creation; see the constructor in Block.h for more information.
    - Write a mining function that searches for a nonce yielding a hash beginning with '2' for this block. You will want to hash data in the same way the sign_block() function does; that is:
        - to_hash = transaction + get_prev_hash() + nonce;
- After a blank line has been read, start from the most recent block and traverse backwards through the chain.
    - Every step should print the block to std::cout.
    - You should use the overloaded insertion operator, which outputs comma-separated members of the block on a line.

In short, your executable will read a series of transactions from stdin and return a valid blockchain on stdout. Mimir will check if the proposed hashes form a valid blockchain. For C++ memory management, you should also make sure to free the memory by the end by deleting the allocated blocks.

Below is an example of input and output from RevCoin. There is a blank line at the end of the input. The output simply calls the operator<< from Block.h by couting the block. Your output may look different depending on your mining function, but all that is required is the hash begins with '2'.

Input	Output
A gives B 10 B gives C 5	B gives C 5,2158731,5,2418096 A gives B 10,0,:,2158731

You can implement any mining method that you wish to find a valid nonce. I recommend just trying different characters from 0 to z in ASCII. The constraint of beginning with '2' makes it very straightforward to find a match, but you can imagine how quickly it becomes a challenge. Try changing the validate_hash function in Block.h to require "221" and see how long it takes! (if it terminates at all, depending on your mining function)

Point Distribution

This part comprises 28% of the overall assignment score. That is distributed as follows:

- 6% - Report
    - Add your references as you go and answer the questions
- 8% - Manual Code Review (includes both parts)
    - Note that autograded points can be revoked if attempts to circumvent the spirit of the assignment's autograding are discovered.
- 2% - Providing the name and target to build the executable
- 12% - Your executable that outputs the blockchain