Assignment 7. The Great Stanford Hash-Off

Due Friday, February 25 at 10:30 am

- Submissions received by the due date receive a small **on-time bonus**.
- All students are granted a pre-approved extension or "grace period" of 24 hours after the due date. Late submissions are accepted during the grace period with no penalty.
- The **grace period expires Sat, Feb 26 at 10:30 am**, after which we cannot accept further late submissions.
- All due dates and submission times are expressed in Pacific time.

You are permitted to work on this assignment in pairs.

Hash tables are one of the most ubiquitous data structures in software engineering, and a lot of effort has been placed into getting them to work quickly and efficiently. In lecture, we coded up chained hashing. We also talked about two other hashing approaches:

- ***Linear Probing:*** A hashing strategy where items can "leak out" of the slots they're supposed to stay in. This strategy is surprisingly fast in practice!
- ***Robin Hood Hashing:*** This slight modification on linear probing "smooths out" the cost of lookups in a linear probing table, which leads to some interesting consequences.

Your task in this assignment is to code up these two hashing strategies and evaluate their performance against chained hashing. How do they hold up in practice? Which hash table implementations tend to work well in which scenarios?

As usual, we recommend making slow and steady progress on this assignment throughout the week rather than trying to do everything here in one sitting. Here's our recommended timetable for this assignment:

- Aim to complete the enumeration warmup the day this assignment goes out.
- Aim to complete the Linear Probing Warmup the day this assignment goes out.
- Aim to complete the Linear Probing Coding component within three days.
- Aim to complete the Robin Hood Warmup within three days.
- Aim to complete the Robin Hood Coding component within six days.
- Aim to complete the Performance Analysis component within seven days.

Assignment Logistics

Starter Files

We provide a ZIP of the starter project. Download the zip, extract the files, and double-click the `.pro` file to open the project in Qt Creator.

 Starter code

Getting Help

Keep an eye on the Ed forum for an announcement of the Assignment 7 **YEAH** (YEAH = Your Early Assignment Help) group session where our veteran section leaders will answer your questions and share pro tips. We know it can be daunting to sit down and break the barrier of starting on a substantial programming assignment – come to YEAH for advice and confidence to get you on your way!

We also here to help if you get run into issues along the way! The Ed forum is open 24/7 for general discussion about the assignment, lecture topics, the C++ language, using Qt, and more. Always start by searching first to see if your question has already been asked and answered before making a new post.

To troubleshoot a problem with your specific code, your best bet is to bring it to the LaIR helper hours or office hours.

Part One: Enumerations Warmup

At a few points in this assignment, you'll be asked to work with ***enumerated types***, a form of custom C++ type representing one out of several different options. We haven't used enumerated types before this assignment, so in this section we figured we'd give you the rundown. 😃

To kick things off, consider the following C++ code:

```
enum class StudentType {
    FROSH,
    SOPHOMORE,
    JUNIOR,
    SENIOR,
    SUPER_SENIOR,
    COTERM,
    MS_STUDENT,
    PHD_STUDENT,
    SCPD_STUDENT
};
```

This defines a new type called **StudentType**. Each variable of type **StudentType** can take on one of the values listed above. For example, we can write code like this:

```
StudentType mia   = StudentType::FROSH;
StudentType chase = StudentType::COTERM;
StudentType erika = StudentType::PHD_STUDENT;
```

You can compare two **StudentType**s against one another using the == operator, reassign them using =, etc.

Enumerated types are useful for remembering which of many mutually exclusive options a particular object happens to be in. You'll make use of them in your linear probing hash table to remember whether a slot is empty, full, or a tombstone.

> There are no deliverables for this part of the assignment; just make sure you've read the above code and understand it. 😃

Part Two: Linear Probing Warmup

The linear probing hash strategy that we talked about in Friday's class is very different from the style of hash table (***chained hashing***) that we saw on Monday. Here's a few of the differences:

- Each slot in a chained hash table is a bucket that can store any number of elements. Each slot in a linear probing table is either empty or holds a single element.
- Every element in a chained hash table ends up in the slot corresponding to its hash code. Elements in a linear probing table can leak out of their initial slots and end up elsewhere in the table.
- Deletions in a linear probing table use tombstones; deletions in chained hashing don't require tombstones.

Before moving on, we'd like you to answer a few short answer questions to make sure that you're comfortable with linear probing as a collision resolution strategy.

Answer each of the following questions in the file **`ShortAnswers.txt`.**

We have a linear probing table containing ten slots, numbered 0, 1, 2, …, and 9. For the sake of simplicity, we'll assume that we're hashing integers and that our hash function works by taking the input integer and returning its last digit. (This is a *terrible* hash function, by the way, and no one would actually do this. It's just for the sake of exposition).

Q1. Draw the linear probing table formed by inserting 31, 41, 59, 26, 53, 58, 97, and 93, in that order, into an initially empty table with ten slots. Write out your table by writing out the contents of the slots, in order, marking empty slots with a period (.) character.

Q2. What is the load factor of the table you drew in Q1?

Q3. Draw a *different* linear probing table that could be formed by inserting the same elements given above into an empty, ten-slot table in a different order than the one given above, or tell us that it's not possible to do this. Assume that you're using the same hash function.

Q4. Which slots do you have to look at to see if the table from Q1 – the one formed by inserting the elements in the specific order we gave to you – contains the number 72?

Q5. Which slots do you have to look at to see if the table from Q1 contains the number 137?

Q6. Suppose you remove 41 and 53 from the linear probing table from Q1 using the tombstone deletion strategy described in class. Draw the resulting table, marking each tombstone slot with the letter T.

Q7. Draw the table formed by starting with the table you came up with in Q6 and the inserting the elements 106, 107, and 110, in that order. Don't forget to replace tombstones with newly-inserted values.

Want to check your answers? Sign up for help in the Conceptual LaIR (CLaIR) queue, which runs in parallel with the regular LaIR.

Part Three: Implementing Linear Probing

Your task in this part of the assignment is to implement the **`LinearProbingHashTable`** type. Here's the interface for that type, as given in the header file:

```
class LinearProbingHashTable {
public:
    LinearProbingHashTable(HashFunction<std::string> hashFn);
    ~LinearProbingHashTable();

    bool contains(const std::string& key) const;

    bool insert(const std::string& key);
    bool remove(const std::string& key);

    bool isEmpty() const;
    int  size() const;

    void printDebugInfo();

private:
    enum class SlotType {
        TOMBSTONE, EMPTY, FILLED
    };

    struct Slot {
        std::string value;
        SlotType type;
    };

    Slot* elems;

    /* The rest is up to you to decide; see below */
};
```

Endearing C++ Quirks, Part 1: `string` versus `std::string`

Inside header files, you have to refer to the string type as **`std::string`** rather than just string. Turns out that what we've been calling string is really named **`std::string`**. The line **`using namespace std;`** that you've placed at the top of all your .cpp files essentially says "I'd like to be able to refer to things like **`std::string`**, **`std::cout`**, etc. using their shorter names **`string`** and **`cout`**." The convention in C++ is to not include the using namespace line in header files, so in the header we have to use the full name **`std::string`**.

Think of it like being really polite. Imagine that the string type is a Supreme Court justice, a professor, a doctor, or some other job with a cool honorific, and she happens to be your sister. At home (in your **`.cpp`** file), you call just call her **`string`**, but in public (in the **`.h`** file), you're supposed to refer to her as **`std::string`**, the same way you'd call her Dr. **`string`**, Prof. **`string`**, Justice **`string`**, or whatever other title would be appropriate.

The **`LinearProbingHashTable`** type is analogous to **`Set<string>`** in that it stores a collection of strings with no duplicate elements allowed. However, it differs in a few key ways:

1. The **`LinearProbingHashTable`** type has its hash function provided to it when it's created. In practice, a hash table should choose a hash function internally. This is both to make it easier to test things (we can construct tests where we know exactly where each element is going to land).

2. The **`Set<string>`** type has no upper bound to how big it can get. For reasons that we'll discuss later, the

LinearProbingHashTable type you'll be implementing is built to have a fixed number of slots, which is specified in the constructor. (Specifically, you're given a particular **HashFunction<string>**, and that **HashFunction<string>** is built to work with a specific number of slots.) This means that there's a maximum number of elements that can be stored in the table, since a linear probing table can't store more than one element per slot.

In terms of the internal representation of the **LinearProbingHashTable**: as with the **HeapPQueue**, you also need to do all your own memory management, though we suspect this will be easier than the **HeapPQueue** because the size of your hash table never changes. You should represent the hash table as an array of objects of the type Slot, where Slot is the struct type defined in **LinearProbingHashTable**. Each slot consists of a string, along with a variable of the enumerated type **SlotType** indicating whether the slot is empty, full, or a tombstone. If the slot is empty or a tombstone, you should completely ignore the string value, since we're pretending the slot is empty in that case. If the slot is not empty, the string value tells you what's stored in that particular slot.

We've provided you with a fairly extensive set of automated tests you can use to validate that your implementation works correctly. To assist you with testing, we've also provided an "Interactive Linear Probing" environment akin to Assignment 6's "Interactive PQueue" button that lets you issue individual commands to a linear probing table to see what happens.

Milestone 3 Deliverables

Here's our recommendation for how to complete this assignment:

Implement the **LinearProbingHashTable** type in **LinearProbingHashTable.h/.cpp**. To do so:

1. Read over **LinearProbingHashTable.h** to make sure you understand what all the functions you'll be writing are supposed to do.
2. Add some member variables to **LinearProbingHashTable.h** so that you can, at a bare minimum, remember the hash function given to you in the constructor. (You'll may or many not need more member variables later; we're going to leave that up to you.) Remember that you need to do all your own memory management.
3. Implement the constructor, which should create a table filled with empty slots and store the hash function for later use. You can determine how many slots your table should have by calling the **HashFunction<T>::numSlots()** member function on **hashFn**, which returns the number of slots that the hash function was constructed to work with.
4. Implement the **size()** and **isEmpty()** functions, along with the destructor. Both functions should run in time O(1). The **size()** member function should return the number of elements currently in the table, rather than the number of slots in the table. (Do you see the distinction?) We also strongly recommend implementing **printDebugInfo()** in a way that prints out the contents of the table, marking which slots are empty and which are nonempty. This will help you later on.
5. Implement **contains()** and **insert()**. For now, don't worry about tombstones or removing elements. You should aim to get the basic linear probing algorithm working correctly.
6. Confirm that you pass all the tests that don't involve removing elements from the table, including the stress tests, which should take at most a couple of seconds each to complete. Don't forget that, if you aren't passing a test, you can set a breakpoint in the test and then run your code in the debugger to step through what's going on. You can also use the Interactive Linear Probing option to explore your table in an interactive environment – preferably with the debugger engaged.
7. Implement the **remove()** function. You should use the tombstone deletion algorithm described in lecture. This may require you to change the code you've written so far:
    1. You may – or may not – need to change your code for **contains()** to handle tombstones. It depends on how you implemented **contains()**.
    2. You may – or may not – need to change your code for **insert()** to handle tombstones. Remember to

place elements in the first empty or tombstone slot that you find. (And make sure not to insert an element into the table if it's already there!)

8. Confirm that you pass all the provided tests, including the stress tests.

Some notes on this problem:

- ***Do not implement contains, remove, or insert recursively.*** Some of the stress tests we'll be subjecting your code to in the time tests will involve working with extremely full tables, and you can easily get a stack overflow this way because the call stack won't be big enough to hold space for all the stack frames.
- Make sure you store the hash function that we provide you in the constructor and use that hash function throughout the table. Variables of type **`HashFunction<string>`** are like other types of variables; you can assign them by writing code to the effect of **`hashFn1 = hashFn2;`**. In the event that you have a **`HashFunction<string>`** as a data member (e.g. something in the **`private`** section of the class) that has the same name as a local variable or parameter to a function, write **`this->hashFn = hashFn;`**.
- If you have a variable of type **`HashFunction<string>`** named **`hashFn`** and an element to hash named **`elem`**, you can compute the hash code of **`elem`** by writing **`hashFn(elem)`**.
- Computing the hash code of a string is fast but not instantaneous. To avoid introducing inefficiencies into your code, try to avoid recomputing the same item's hash code multiple times in a loop when performing an insertion, deletion, etc.
- Remember that, like the **`Set<string>`** type, your **`LinearProbingHashTable`** should not allow for duplicate elements. If the user wants to insert a string that's already present, you should not insert a second copy.
- In Assignment 6, we added some code to the **`DataPoint`** type to make it a lot easier to spot when you had accidentally walked off the end of an array or otherwise used an uninitialized **`DataPoint`**. We have not provided the same safeguards here. This means that if you walk off the end of your array of slots, or try reading from a negative index, the behavior you see may vary from run to run. Your code might work correctly sometimes, crash other times, or produce test failures in other runs.
- A common mistake in implementing this hash table is to try to set the **`value`** field of a **`Slot`** to **`nullptr`** when removing a string from the hash table, with the idea of saying "this string doesn't exist any more." That, unfortunately, doesn't work. Remember that in C++, a string represents an honest-to-goodness string object, so you can't have a "null string" the same way you can't have a "null integer." Unfortunately, it is legal C++ code to assign **`nullptr`** to a string, which C++ interprets as "please crash my program as soon as I get here" rather than "make a null string." Instead, just change the type field to indicate that although there is technically a string in the slot, that string isn't meaningful.
- Make sure not to read the contents of a string in a **`Slot`** if that **`Slot`** is empty or a tombstone. If the slot isn't filled, it means that the string value there isn't meaningful. There are a lot of bugs that can arise if you accidentally read the string in a **`Slot`** when the slot doesn't have an element in it.
- Your table should be able to store any strings that the user wants to store, including the empty string.
- The **`contains`**, **`insert`**, and **`remove`** functions need to function correctly even if the table is full. Specifically, **`insert`** should return false because there is no more space, and **`remove`** and **`contains`** should operate as usual. You may need to special-case the logic here – do you see why?
- You are encouraged to add private helper functions, especially if you find yourself writing the same code over and over again. Just don't change the signatures of any of the existing functions. If you do define any helper functions, think about whether they should be marked **`const`**. Specifically, helper functions that don't change the hash table should be marked **`const`**, while helper functions that do make changes to the table should not be **`const`**.
- You are likely to run into some interesting bugs in the course of coding this one up, and when you do, don't forget

how powerful a tool the debugger is! Feel free to set breakpoints in the different test cases so that you can see exactly what your code is doing when it works and when it doesn't work. Inspect the contents of your array of slots and make sure that it's consistent with what you expect to see. Once you've identified the bug – and no sooner – edit your code to fix the underlying problem.

- You ***must not*** use any of the container types (e.g. **`Vector`**, **`Set`**, etc.) when solving this problem. Part of the purpose of this assignment is to let you see how you'd build all the containers up from scratch.
- We encourage you to write your own **`STUDENT_TEST`**s here to help debug your solution as you go. However, you are not required to do so.

Endearing C++ Quirks, Part 2: Returning Nested Types

There's another charming personality trait of C++ that pops up when implementing member functions that return nested types. For example, suppose that you want to write a helper function in your **`LinearProbingHashTable`** that returns a pointer to a **`Slot`**, like this:

```
private:
    Slot* hsAreCute();
```

In the **`.cpp`** file, when you're implementing this function, you need to give the full name of the **`Slot`** type when specifying the return type:

from .h file's name

```
LinearProbingHashTable::Slot* LinearProbingHashTable::hsAreCute() {
    // Wow, this pun lost a lot in translation.
}
```

While you need to use the full name **`LinearProbingHashTable::Slot`** in the return type of an implementation of a helper function, you don't need to do this anywhere else. For example, this code is perfectly legal:

```
LinearProbingHashTable::Slot* LinearProbingHashTable::hsAreCute() {
    Slot* h = new Slot[137]; // Totally fine!
    return h;
}
```

Similarly, you don't need to do this if the function takes a **`Slot`** as a parameter. For example, imagine you have this member function:

```
private:
    void iLostMoneyToA(Slot* machine);
```

You could implement this function without issue as

```
void LinearProbingHashTable::iLostMoneyToA(Slot* machine) {
    // Don't make the same mistake as me!
}
```

Part Four: Robin Hood Warmup

Robin Hood hashing is a clever variation on linear probing that reduces the variance in the costs of insertions, deletions, and lookups. As a reminder, Robin Hood hashing is based on linear probing, but differs in the following ways:

1. Each element in a Robin Hood hash table is annotated with the distance it is from its home slot. This distance is measured by the number of steps backwards you have to take, starting at that element, to get the index of its home slot. (As in linear probing, we wrap around the ends of the table if we reach that point.)
2. Lookups in a Robin Hood hash table can stop early. Specifically, if the element we're looking for is further from home than the currently-scanned table element, we know that the element we're looking for isn't in the table and can stop our search. (Do you see why the element can't be there?)
3. When inserting an element into a Robin Hood hash table, if the element being inserted is further from home than the element in the table slot being scanned, we displace the element in the table at that index, place the element we wanted to insert there, then continue onward as if we were inserting the displaced element all along. (We do not do anything if the distances are tied.)
4. There are no tombstones in a Robin Hood hash table. Instead, when deleting an element, we use ***backwards-shift deletion***: we shift elements back one spot in the table until we either (1) find an empty slot or (2) find an element in its natural home spot.

Before coding up a Robin Hood hash table, few minutes to work through some quick short answer questions.

Answer each of the following questions in the file **`ShortAnswers.txt`**.

We have a Robin Hood hash table containing ten slots, numbered 0, 1, 2, …, and 9. For the sake of simplicity, we'll assume that we're hashing integers and that our hash function works by taking the input integer and returning its last digit. (As before, this is a *terrible* hash function, and we're doing this just for the sake of simplicity.)

Q8. Draw the Robin Hood table formed by inserting 106, 107, 246, 145, 151, 103, 245, 108, and 221, in that order, into an initially empty table with ten slots. Write out your table by writing out the contents of the slots, in order, marking empty slots with a period (.) character. Below each table slot, indicate the distance it is from its home position, marking empty slots distances with a dash (-) character.

Q9. Draw a *different* Robin Hood table that could be formed by inserting the same elements given above into an empty, ten-slot table in a different order than the one given above, or tell us that it's not possible to do this. Assume that you're using the same hash function.

Q10. Which slots do you have to look at to see if the table from Q8 – the one formed by inserting the elements in the specific order we gave to you – contains the number 345? Remember that, with Robin Hood hashing, you can cut off a search for an element if the element you're currently looking for is further from home than the currently-scanned element.

Q11. Which slots do you have to look at to see if the table from Q8 contains the number 300?

Q12. Draw the Robin Hood table formed by removing 151 from the table you drew in Q8. Use backward-shift deletion.

Q13. Draw the Robin Hood table formed by removing 145 from the table you drew in Q12.

Want to check your answers? Sign up for help in the Conceptual LaIR (CLaIR) queue, which runs in parallel with the regular LaIR.

Part Five: Robin Hood Hashing

Your next task is to implement the **`RobinHoodHashTable`** type, which represents a hash table implemented using Robin

Hood hashing, as described in lecture. Here's what we've provided you:

```
class RobinHoodHashTable {
public:
    RobinHoodHashTable(HashFunction<std::string> hashFn);
    ~RobinHoodHashTable();

    bool contains(const std::string& key) const;

    bool insert(const std::string& key);
    bool remove(const std::string& key);

    bool isEmpty() const;
    int  size() const;

    void printDebugInfo();

private:
    static const int EMPTY_SLOT = /* something */;

    struct Slot {
        std::string value;
        int distance;
    };

    Slot* elems;

    /* The rest is up to you to decide; see below */
};
```

In many ways, what we've given you for **RobinHoodHashTable** is the same as that for the **LinearProbingHashTable**: the constructor takes in a **HashFunction<string>** that lets you know how many slots to use, you need to support insert, contains, and remove, the table never grows, etc.

There are, however, some notable differences. First, note that there's no longer a **SlotType** enumerated type. That's because there are only two options for each table slot – either the slot is empty, or the slot is filled and the item there is some distance from home. ***You should mark slots empty by setting their distance value to the constant EMPTY_SLOT.*** Any slot whose distance is not the value **EMPTY_SLOT** is assumed to be full and to be the indicated number of spots away from home.

Second, as the name suggests, this class should be implemented using Robin Hood hashing rather that linear probing. You may find parts of your **LinearProbingHashTable** useful as starting points for your design of the **RobinHoodHashTable**, but the differences between the two table types (tracking elements' distances from their home, displacing elements on insertions, cutting off searches early, backward-shift deletion, the lack of tombstones, and the different **Slot** representation) will require you to rewrite each of the core functions (**contains**, **insert**, and **remove**) to some degree.

Here's what you need to do:

Part Five Deliverables

Implement the **RobinHoodHashTable** type in **RobinHoodHashTable.h/.cpp**. To do so:

1. Add a member variable to **RobinHoodHashTable.h** to remember the hash function given to you in the

constructor. You may need to add more data members later. Remember that you need to do all your own memory management.

2. Implement the constructor, which should create an empty table and store the hash function for later use.
3. Implement the **`size()`** and **`isEmpty()`** functions, along with the destructor. The **`size()`** and **`isEmpty()`** functions should run in time O(1). The **`size()`** member function should return the number of elements currently the table, rather than the number slots in the table. We recommend implementing **`printDebugInfo()`** in a way that prints out the contents of the table, which slots are empty, and how far away from home each element is.
4. Implement **`contains()`** and **`insert()`**. Remember that **`insert`** may move elements and that **`contains`** should cut off searches early when the element in question is too far from home.
5. Confirm that you pass all the tests that don't involve removing elements from the table, including the stress tests, which should take at most a couple of seconds each to complete. Feel free to use the "Interactive Robin Hood" option to test your code.
6. Implement **`remove()`**. You should use backward shift deletion to accomplish this, which should not require you to change the implementations of contains or insert.
7. Confirm that you pass all the provided tests, including the stress tests.

Some notes on this problem:

- Do not implement **`contains`**, **`insert`**, or **`remove`** recursively; this may cause stack overflows when working with large tables.
- You might find the **`swap`** function, defined in **`<algorithm>`**, useful here. It takes in two arguments and swaps them with one another. For example:

```
int x = 137, y = 42;
swap(x, y); // Now x = 42, y = 137.
```

- As with linear probing, make sure not to read the **`value`** field of a slot until you've checked that the slot isn't empty. Otherwise, you'll read a string that may or may not be in the table.
- When inserting an element, you should only displace an element if it is *strictly* closer to its home than the inserted element is to its home. For example, if the element you're inserting is two steps from home and the currently-scanned element is two steps from home as well, you should not displace that element.
- When removing an item from the table, you'll need some way to refer to "the slot after the current one, wrapping around." It's easy to remember the "slot after the current one" bit, and easy to forget the "wrapping around" part. 😃
- Similarly, watch for bugs that arise when trying to talk about "the slot before the current one, wrapping around."
- You ***must not*** use any of the container types (e.g. **`Vector`**, **`Set`**, etc.) when solving this problem. Part of the purpose of this assignment is to let you see how you'd build all the containers up from scratch.
- We encourage you to write your own **`STUDENT_TEST`**s here to help debug your solution as you go. However, you are not required to do so.

Part Six: Performance Analysis

We implemented chained hashing in class, and we've included a **`ChainedHashTable`** type in the **`Demos/`** directory along

the lines of the two hash tables you built here. You just implemented a linear probing table and a Robin Hood hash table. The question then is – how do they stack up against one another?

Choose the "Performance Analysis" button from the main menu. This option will run the following workflow on each of the three table types:

- Insert all words in the file **`EnglishWords.txt`** into an empty hash table. Each item is inserted twice, the first time to measure the speed of a successful insertion, and the second time to measure the speed of an unsuccessful insertion when the item is already present.
- Look up each word in **`EnglishWords.txt`** in that hash table, measuring the average cost of each successful lookup.
- Look up the capitalized version of each word in that hash table. Since all the words in **`EnglishWords.txt`** are stored in lower-case, this measures the average cost of each unsuccessful lookup.
- Remove the capitalized versions of each word in the hash table. This measures the average cost of unsuccessful deletions, since none of those words are present.
- Remove the lower-case versions of each word in the hash table. This measures the average cost of successful deletions.

The provided starter code will run this workflow across a variety of different load factors α (ratios of numbers of elements in the table to the number of slots in the table), reporting the times back to you. ***This may take a while to complete;*** it's okay if it takes about five or ten minutes to finish running.

As a note, ***the timing numbers you get will be sensitive to what else is running on your computer.*** If you leave your program running time tests in the background while, say, watching a YouTube video, the overhead of your computer switching back and forth between different processes can skew the numbers you'll get back. We recommend that once you click the "Performance Analysis" button, you walk away from your computer for a while, stretch a bit, and return once all the time trials have finished.

Once you have the data, review the numbers that you're seeing. Look vertically to see how the times for a particular hash table compare across load factors, and horizontally to see how the different tables compare against one another.

Answer each of the following questions in the file **`HashTableAnalysis.txt`**.

Q1. Look at the costs of ***successful lookups*** in both Robin Hood hashing and linear probing hashing across a range of load factors. Which approach is faster? Qualitatively, does the size of the gap stay the same, grow slowly, or grow quickly as the load factor increases? Explain how the similarities / differences between linear probing and Robin Hood lookups algorithms would give rise to these trends. (Your answer should be 2 - 4 sentences in length.)

Q2. Repeat the above exercise for ***unsuccessful lookups***.

Q3. Repeat the above exercise for ***successful insertions***.

Q4. Repeat the above exercise for ***unsuccessful insertions***.

Q5. Repeat the above exercise for ***successful deletions***.

Q6. Repeat the above exercise for ***unsuccessful deletions***.

Q7. Explain why it would *not* be a good idea to use a linear probing hash table or Robin Hood hash table with a load factor of 0.01 even though it would be much faster than with a higher load factor.

Q8. In practice, someone has to make a judgment call about which type of hash table to use and with which load factors. Review the numbers you've seen for chained hashing, linear probing hashing, and Robin Hood hashing. Propose a single choice of hash table and load factor that you believe is the "best," then justify your decision. (Your answer should be 2 - 4 sentences in length.)

(Optional) Part Seven: Extensions

You've just implemented two hash tables! If that isn't enough for you, or if you want to take things a step further, you're welcome to build on the base assignment and do whatever cool and exciting things seem most interesting to you!

If you'd like to implement a more complex hash table than the ones you did here, please edit the file **`MyOptionalHashTable.h`** and **`MyOptionalHashTable.cpp`** with your implementations, while leaving your **`LinearProbingHashTable`** and **`RobinHoodHashTable`** types unmodified. You can then integrate your type into the performance analyzer by editing the file **`Demos/TimeTestConfig.h`**. There are instructions there about how to edit that file.

Here are some suggestions of things to try out:

- The hash tables we've defined here are fixed-sized and don't grow when they start to fill up. In practice, you'd pick some load factor and rehash the tables whenever that load factor was reached. Write code that lets you rehash the tables once they exceed some load factor of your choosing. Tinker around to see what the optimal load factor appears to be!
- Tombstone deletion has a major drawback: if you fill a linear probing table up, then delete most of its elements, the cost of doing a lookup will be way higher than if you had just built a brand new table and filled it in with just those elements. (Do you see why?) Some implementations of these hash tables will keep track of how many deleted elements there are, and when that exceeds some threshold, they'll rebuild the table from scratch to clean out the tombstones. Experiment with this and see what you can come up with!
- There are many other hashing strategies you can use. Quadratic probing and double hashing, for example, are variations on linear probing that cap the number of elements that can be in a slot at one, but choose a different set of follow-up slots to then look at when finding the next place to look. Cuckoo hashing is based on a totally different idea: it uses two separate hash functions and places each element into one of two tables, always ensuring that the elements are either at the spot in the first table given by the first hash function or the spot in the second table given

by the second hash function. Hopscotch hashing is like linear probing, but ensures that elements are never "too far" away from their home location. FKS hashing is like chained hashing, but uses a two-layer hashing scheme to ensure that each element can be found in at most two probes. Read up on one of these strategies – or another of your choosing – and code them up. How quickly do they run? You can add your own hash table type to the performance analysis by editing **`Demos/TimeTestConfig.h`**.

- We provide the hash functions in this assignment, but there's no reason to suspect that our hash functions are the "best" hash functions and you can change which hash function to use. Research other hash functions, code them up, and update the performance test (it's the **`timeTest`** function in the files **`Demos/PerformanceGUI.cpp`**) to use your new hash function. How does your new hash function compare with ours?

Submission Instructions

Before you call it done, run through our submit checklist to be sure all your **`t`**s are crossed and **`i`**s are dotted. Make sure your code follows our style guide. Then upload your completed files to Paperless for grading.

Partner Submissions:

- If you forget to list your partner you can resubmit to add one
- Either person can list the other, and the submissions (both past and future) will be combined
- Partners are listed per-assignment
- You can't change/remove a partner on an individual submission

Please submit only the files you edited; for this assignment, these files will be:

- **`ShortAnswers.txt`**. ***(Don't forget this one, even though there's no code in it!)***
- **`HashTableAnalysis.txt`**. ***(Don't forget this one, even though there's no code in it!)***
- **`LinearProbingHashTable.h/.cpp`**. ***(Remember to submit both of these files!)***
- **`RobinHoodHashTable.h/.cpp`**. ***(Remember to submit both of these files!)***

You don't need to submit any of the other files in the project folder.

🏁 Submit to Paperless

If you modified any other files that you modified in the course of coding up your solutions, submit those as well. And that's it! You're done!

Good luck, and have fun!