CS106B
Winter 2020

Handout 24
February 21, 2020

Assignment 6: The Great Stanford Hash-Off

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 can "smooth out" the cost of lookups in a linear probing table, allowing the table to function better at higher load factors.

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?

Due Friday, February 28th at 11:30AM.

You are welcome to work in pairs on this assignment.

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

Problem One: 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 Wednesday. 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 `res/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. 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.

Q3. 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?

Q4. Which slots do you have to look at to see if the table from Q1 contains the number 137?

Q5. 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`.

Q6. Draw the table formed by starting with the table you came up with in Q5 and the inserting the elements 106, 107, and 110, in that order. Don't forget to replace tombstones with newly-inserted values.

Problem Two: 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:
    struct Slot {
        std::string value;
        bool isEmpty;
    };

    /* 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 `HashSet<string>` in that it stores a collection of strings with no duplicate elements allowed. However, it differs in a few key ways:

1. The `HashSet<string>` type does all the work to choose its hash function internally. The `LinearProbingHashTable` type has its hash function provided to it when it's created. 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 `HashSet<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 `bool` indicating whether the slot is empty. If the slot is empty, 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.

You may have noticed that the `Slot` type doesn't have a way of marking a slot as being a tombstone. For now, just ignore that detail; we'll introduce tombstones later.

We've provided you with a fairly extensive set of automated tests you can use to validate that your implementation works correctly, but they aren't exhaustive and there are some cases we aren't checking for. To assist you with testing, we've also provided an "Interactive Linear Probing" environment akin to Assignment 5's "Interactive PQueue" button that lets you issue individual commands to a linear probing table to see what happens.

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 and maintain an array of slots. (You'll likely need more member variables later.) Remember that you need to do all your own memory management.
3. Implement the constructor, which should create an empty table 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. 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 update the `Slot struct` type to have some way of telling you whether a slot contains a tombstone.
    2. You may – or may not – need to change your code for `contains()` to handle tombstones. It depends on how you implemented `contains()`.
    3. 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:

- Make sure you store the hash function that we provide you in the constructor and use that hash function throughout the table. Otherwise, you'll likely fail the provided tests, which assume they know which hash function your table uses.
- Remember that, like the `HashSet<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.
- You might be wondering why `Slot` has an `isEmpty` `bool` rather than using the strategy of just assigning `nullptr` to the string when the slot is empty. 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."
- Make sure not to read the contents of a string in a `Slot` if that `Slot` is empty. If the slot is empty, 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 is empty.
- Your table should be able to store any strings that the user wants to store, including the empty string. In particular, you shouldn't pick some string value to mean "this is a tombstone," since if the user were to try to insert that string, your table might have trouble differentiating the user's string from an actual tombstone.
- 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?
- Our Interactive Linear Probing demo uses the same hash function every time. If you want to add a custom test and use that hash function, you can do so by initializing your hash table as

  ```
  LinearProbingHashTable table(Hash::consistentRandom(numSlots));
  ```
- 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`, `HashSet`, 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.

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:

```
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!
}
```

Problem Three: 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. Before coding it up, we'd like you to take a few minutes to work through some quick exercises.

Answer each of the following questions in the file `res/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.)

Q7. 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.

Q8. 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.

Q9. Which slots do you have to look at to see if the table from Q7 – 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 at is closer to home than the element you're searching for.

Q10. Which slots do you have to look at to see if the table from Q7 contains the number 300?

Q11. Draw the Robin Hood table formed by removing 151 from the table you drew in Q7. Use the backward-shift deletion algorithm from lecture rather than tombstones.

Q12. Draw the Robin Hood table formed by removing 145 from the table you drew in Q11.

Problem Four: 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. The interface for the `RobinHoodHashTable` is essentially 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 three core differences between the Robin Hood hashing strategy and linear probing. These differences, however, may result in some large changes to the code.

1. When looking up an element in a Robin Hood hashing table, you can sometimes stop your search earlier by noting that you're further away from home than the current element is away from its home.
2. When inserting into a Robin Hood hash table, you may need to shift existing elements further down the table, incrementing their distances from home.
3. When removing from a Robin Hood hash table, instead of deleting by using tombstones, you use the backward shift deletion algorithm described in lecture.

You may find that parts of your `LinearProbingHashTable` will be useful starting points for your design of the `RobinHoodHashTable`. You will certainly find that the changes given above will require you to abandon or heavily modify other parts.

Here's what you need to do:

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

1. Define a custom `struct` type inside `RobinHoodHashTable` representing a slot in the table. There are many ways to do this. You'll need to be able to store the contents of the slot, whether the slot is empty, and some information that lets you determine how far away from home the element in that slot (if any) is.
2. Add some member variables to `RobinHoodHashTable.h` to remember the hash function given to you in the constructor and maintain an array of slots. Remember that you need to do all your own memory management.
3. Implement the constructor, which should create an empty table and store the hash function for later use.
4. Implement the `size()` and `isEmpty()` functions, along with the destructor. 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.
5. Implement `contains()` and `insert()`. Remember that `insert` may move elements.
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. Feel free to use the "Interactive Robin Hood" option to test your code.
7. Implement `remove()`. You should use backward shift deletion to accomplish this, which should not require you to change the implementations of `contains` or `insert`.
8. Confirm that you pass all the provided tests, including the stress tests.

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.
```

Problem Five: 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, measuring the average cost of each successful insertion.
- Insert those words again in random order, measuring the cost of each unsuccessful insertion. (These insertions will fail because these hash tables don't support duplicates.)
- 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, reporting the times back to you.

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 minute or two, 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 `res/ShortAnswers.txt`.

Q13. Look at the numbers for chained hashing. Compare the cost of successful and unsuccessful insertions across a variety of load factors. What trends do you see? Offer the best explanation you can for why those numbers are the way they are. Then, repeat this for lookup times and removal times.

Q14. Repeat the above exercise for linear probing.

Q15. Repeat the above exercise for Robin Hood hashing.

Q16. If you create a chained hash table, a linear probing table, and a Robin Hood table that each have the same load factor, which will use the most memory? Which will use the least? Why?

Q17. Based on your answers to the above questions, if you had to pick a single hash table / load factor combination that offered a good balance of performance and memory usage, which one would you pick, and why?

(Optional) Problem Six: 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. 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!
- 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?
- 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 `AllHashTables` in the file `Demos/PerformanceGUI.cpp`.

Submission Instructions

Once you've autoindented your code so that it looks beautiful and worked through the Assignment Submission Checklist, submit the following files on Paperless, plus any other files you modified when writing up extensions:

- `res/ShortAnswers.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!)***

And that's it! You're done! You've just explored how to engineer a hash table and are now an expert at dynamic arrays.

Good luck, and have fun!