CMSC 341 Data Structures — All Sections — Spring 2018

Project 5: Hash Table Strategies & Performance

Due: Tuesday, May 15, 8:59:59pm

Addenda

- **[Mon 5/07/18 18:05]**
  - Fixed declaration of data member `hashFunc` in HashTable.h, to declare that it takes a `const` param--this matches what it already said in the project spec.
  - Changed declaration of `HashTable::~HashTable()` from pure virtual function to just empty virtual function.
  - Mentioned that you need to overload operator<<() in order to print out interesting things in dump()
  - Provided `getCurrentCPU.{h,cpp}`
  - Provided test code.
- **[Sat 5/05/18 14:00]**
  - Fixed typo in return type of hashFunc() to `unsigned int` (was "bool")
  - Intro in Spec originally said hash function is template param; changed that to "constructor parameter"; was correct in rest of document.
  - Changed type of container for buckets in ChainHashTable, from `forward_list` to `list`

Additions and changes are highlighted in orange throughout.

Objectives

The objective of this programming assignment is for you to gain experience working with hash tables and measuring execution times

Background

Okay, no more silly, condescending made-up stories about dumb managers that you have to fool. Prof. Park is your manager now, and he's no dummy. He decides to immediately set about evaluating your ability to do clean implementations and rigorous empirical evaluations of your latest course topic: hash tables. No more joking around--coding is a serious matter. Well, okay, one joke, but that's it--make it last:

A man walks into a bar. He asks the bartender for three beers, all at the same time. He proceeds to take turns drinking some from each glass, until they're all empty. He then gets up and leaves. The bartender thinks this is weird, but he's seen weirder. Some UMBC students came in the week before...

The next week, the man comes in again and does the exact same thing: always three beers, always drinking all of them simultaneously, then quietly leaving. This goes on week after week for months. Finally, one night the man comes in and orders only two beers. The bartender can't contain his curiosity and asks: why the multiple beers? The man explains that he has two brothers who have moved far away, but they agreed they would all stay spiritually connected by "sharing a drink" this way every Saturday night at 9, wherever they may each be. The bartender immediately gets the significance of the two beers and says: "I'm

so sorry for your loss." The man asks "why??". The bartender answers: "Oh, I'm assuming one of your brothers must have died." "Ah, no", the man replies; "my brothers are still alive--I've just given up drinking."

Okay, back to business.

Your assignment is to implement and compare the performance of two different types of hash tables. You must implement one version that uses hash buckets with chaining, and a second version that uses open addressing with linear probing. Recall that these two kinds of hash tables have fundamentally different approaches to dealing with collisions: Chaining uses a more elaborate data structure for the hash buckets that allows us to store multiple items in each bucket. This leads to additional implementation complexity, depending on what data structure you use to implement each bucket. What it loses in terms of complexity it makes up for by the fact that a handling a collision amounts to searching through a list of exactly those items that hashed to the identical bucket, but no others. An additional benefit is that you can actually have a load factor greater than one: in other words, you can store >N items in an N-bucket hash table.

Open addressing, on the other hand, stores only a single item in each bucket, and deals with collisions by trying alternative buckets in an ordered fashion until it finds one that is empty. Data structure-wise, it is much simpler to implement. However, it has some performance issues when dealing with collisions: first, it might "collide" with earlier values that initially hashed to a different initial index but wound up landing in the same bucket that the current value wants to be in. Then, when we probe other alternatives, it again might collide with any number of other values that ended where they are through various means. Then, for the more simplistic probing strategies, there is the matter of clustering, which leads to an increased number of probe attempts. Then, there is the matter of having to implement lazy deletions. To make life simpler for you, for this project, the open addressing implementation will use a very simple linear probing strategy for handling collisions.

Since theoretical analyses of the performance of these hash tables is complex, you are going to compare how the two above designs perform by doing an empirical evaluation, adding code to your implmentation to measure runtimes for various operations. We will show you how to make time measurements; it is your responsibility to decide how to use these tools to actually profile the performance of your implementation.

One more design detail: Since the two types of hash tables are closely related, and have the exact same external interface, you will be implementing them as derived classes of a common base class. In other words, you will be using polymorphism to allow us to use the same driver code to test out both implementations, by using a common base class pointer but having them point to hash table objects of different subclasses. Additionally, we want your hash tables to be able to store arbitrary data types, which means you must define them as templated classes. Then, since it is not possible to have an effective hash function that works universally across different types, one of the ~~template~~constructor parameters will be a hash function appropriate for the type, computing a hash code that is an unsigned int. You will then convert this into a hash table index using the division method.

Your Assignment

Your assignment is to implement a hash table class hierarchy that consists of a base class that defines the basic interface for a hash table, as well as two derived classes: one that implements a hash table with chaining, and another version that uses open addressing. The base class definition declares the function prototypes for the various member functions a hash table needs, like insert(), find() and remove(). Of course, these will not be defined in the base class, but instead in the specialized subclasses you will be creating. This allows polymorphism to work. (Review the concept from your Comp Sci II notes if you forgot.) Note that other than the interface definitions, there is little else we can put in the base class, since almost every other part, even the underlying hash table array, depends on what type of collision strategy we're using.

We are providing the base class definition to you. You should not change any part of those files, nor should you submit it. Here is the definition of the class from HashTable.h:

```
#ifndef HASHTABLE_H
#define HASHTABLE_H

#include <vector>

template <typename T>
```

```
class HashTable {
public:
    virtual ~HashTable() {};

    // Functions in a standard hash table interface,
    // independent of implementation:
    virtual bool insert(const T &data) = 0;
    virtual bool find(const T &data) = 0;
    virtual T remove(const T &data, bool &found) = 0;

    // Functions for debugging and grading:
    virtual void dump() = 0;
    virtual int at(int index, std::vector<T> &contents) = 0;

    // Place to store pointer to hash code generating function
    unsigned int (*hashFunc)(const T &data);

    // There are no other data members that the different derived classes
    // would have in common
};

#endif
```

Note that the base class--and also the derived classes--must be templated. This is because we want the flexibility to be able to store any type of data in our hash tables, not just some fixed type like ints or strings. Unfortunately, this makes it difficult to have a generic hash function defined inside the class, since we don't even know the type of the key. We resolve this by having the user provide an appropriate hash function at the time the hash table is constructed, which we then store a pointer to in our class (in the "(*hashFunc)(T &data)" member as seen above).

Another complication of being type-agnostic is that the user of our hash tables must make sure the type being stored has an overloaded operator== (and operator!=) so that we can tell when we've found the desired item in the hash table. Note that this is especially true if the user wants to store pointers-to-objects in our hash table. Then, the user-provided overloaded comparison function would dereference the pointer first and compare the actual data being pointed to. The standard pointer equality operator would merely test whether the pointed-to addresses are the same. You can see an example of this in the first test program we provide, `test1.cpp`. Note that we simply want to know if/when we've found the specific item in the table, nothing fancier, so we do not need any of the other comparison operators (like '<', and '>') to be overloaded.

The ChainHashTable class

The first concrete subclass of the HashTable class that you will be defining is the `ChainHashTable` class. This is a version that uses chaining to handle collisions. That means that the hash table is a bucket array where every bucket can hold a collection of values. As discussed in class, any of a number of different data structures could be used to store the items for each bucket. For this project, we are requiring that you use the STL ~~singly-linked list: the `forward_list` class. The forward_list class only allows simple insertion at the head of the list, but this is fine for our needs; when you add new elements to a bucket, make sure you add it to the head of the list. To search the forward_list to look for an item, and to subsequently remove the item if desired, you must use the forward_list's iterator.~~ `std::list` class. When you add new elements to a bucket, you must add it to the head of the list, using the `push_front()` member function. To search the list to look for an item, and to subsequently remove the item if desired, you must use the std::list's iterator. You should be very familiar with iterators by this point.

There isn't much else to the ChainHashTable class. At construction time, the user provides a pointer to a hash function appropriate to generate a hash code for the data type. Your hash table operations use this to generate a hash code, and then uses the division method (simple modulo division by the size of the hash table) for compression to generate an index into the hash table. It then adds the item to the bucket (i.e., ~~forward_~~ list) at the index. Note that the hash table should be initialized to an array of empty ~~forward_~~ list's at construction time, so that you can start adding items immediately after construction.

The ProbeHashTable class

The second subclass of the HashTable class that you will be defining is the `ProbeHashTable` class. This is a version that uses open addressing to handle collisions. That means that the table is an array where each member only holds at most a single value. You might be tempted to therefore declare the hash table as a simple array of elements of type T (the templated type parameter), but this has a problem. First, you need a way to mark empty buckets. Also, a hash table that uses open addressing cannot just free the bucket up by marking it as empty when an item is deleted: it must mark that slot as a special lazy-deleted bucket; otherwise linear probing won't work for later searches. (Review your notes on how lazy deletion works.) So, it must have two special unique values that it can use to mark empty buckets and lazy-deleted buckets, respectively, and these must be distinct from any actual data value the user might insert. However, since you don't know the type of T or what the user is using it for, you cannot know what two values are free for you to use. So, the cleaner, but slightly space-intensive, solution you will use is to define a helper class called `HashTableEntry` that has two data members:`m_flag` and `m_data`. The `m_flag` member is of type int, and has one of 3 possible values at any given time: 0 if that bucket is empty, -1 if that bucket is marked as lazy-deleted, and 1 if that bucket currently holds a value. (Obviously, you should use named constants instead of these magic numbers.) The `m_data` member is of type T, and holds the value the user inserted, if m_flag == 1; else, it is garbage. The bucket array is then declared as an array of elements of type HashTableEntry. You should define this HashTableEntry helper class as a nested class of ProbeHashTable, and you should make all of its data members public. It does not need any member functions: you are using it just as a struct.

Like the ChainHashTable, the ProbeHashTable should use the hash function and the division method to generate an starting index into the hash table. In the case of a collision, it should use simple linear probing to look for the first empty or lazy-deleted bucket and insert the data item there. Note that the hash table should be initialized to an array of HashTableEntry objects, each with its m_flag member initialized to 0, for "empty", so that you can start adding items immediately after construction.

Measuring Execution Times

An important part of this project is computing average execution times for the various operations on the hash table. To do that, we are providing a function called `getCurrentCPU()`, in the files `getCurrentCPU.{h,cpp}` which returns the cummulative amount of CPU time (measured in some defined units) that your program has used so far. By calling this function immediately before, and then just after a series of hash table operations, you can get a measure of how much CPU time those operations took, and you can use that to compute the average time per insertion, deletion, etc. We give you an example of its usage in test2.cpp; you can just adapt that to do your own performance measurements.

Requirements

The following are some firm requirements, i.e., necessary for grading. (When we refer to "both hash table classes" here, we are stating the requirement applies to both the `ChainHashTable` and `ProbeHashTable` classes.)

- We are providing the definition of the abstract parent class `HashTable`. This class will serve as the base class for your two other actual functional hash table classes. You must not modify any part of the file HashTable.h.
- You must define a concrete templated class named `ChainHashTable`, as described in the previous section. It must be directly derived from the HashTable class. The class definition and function definitions must be in the files `ChainHashTable.h` and `ChainHashTable.cpp`
- You must define a concrete templated class named `ProbeHashTable`, as described in the previous section. It must be directly derived from the HashTable class. The class definition and function definitions must be in the files `ProbeHashTable.h` and `ProbeHashTable.cpp`
- Both hash table classes must be templated, with a template parameter defining the type of data stored. Look at the definition of HashTable for hints.
- Both hash table classes must define a constructor that accepts a parameter passing a pointer to the hash code generation function, and one specifying the size of the underlying hash table.
- Both hash table classes must override the virtual destructor to free up the dynamic memory allocated for the hash table, so that there are no memory leaks.

- Both hash table classes must use the division method, where the hash index is: the value returned by `hashCode()` modulo-divided by the table size.
- The `ChainHashTable` must have a data member called `m_table`, which points to a dynamically allocated array of `std::list` objects (part of the STL). This array must have the number of elements specified by the constructor argument. (Do ***not*** use the vector class!).
- The ChainHashTable class must insert new data at the *head* of the list for the appropriate bucket, using `list::push_front()`
- The `ProbeHashTable` must have a data member called `m_table`, which points to a dynamically allocated array of `HashTableEntry` objects (described above, and defined as a nested class of your ProbeHashTable class). This array must have the number of elements specified by the constructor argument.
- Other than the `list` class, and the `vector` we use for the `at()` grading helper function, you may not use any other STL class (or a container class from any other library), not even `string`.
- Your program must not leak memory. Design your destructors carefully. (Note that the base class HashTable declares the destructor to be virtual, meaning it will call the appropriate hash table type-specific destructor even through base class pointers.)

Additional Specifications

Here are some additional specifications for the `ChainHashTable` and `ProbeHashTable` class member functions that you have to implement. You will definitely want to add other data members and member functions to your derived hash table classes, but the prototypes for the member functions listed here should not be changed. You may create additional classes if you prefer, but all class declarations should be placed in one of your submitted .h files (not `HashTable.h`!) and all implementations should be placed in the corresponding .cpp file.

- ```
  ChainHashTable(unsigned int (*hashFunc)(const T&), int n=101);
  ```

- ```
  ProbeHashTable(unsigned int (*hashFunc)(const T&), int n=101);
  ```

  These are the default constructors for your two hash table classes. The parameter `hashFunc` is a pointer to the function that should be used to generate hash codes for the data type T. The parameter `n` is the requested size of the hash table. We chose 101 as the default value because it is a nice prime number, but it is not your responsibility to enforce any constraints on the size if the user provides an argument. The `ChainHashTable` or `ProbeHashTable` object created by the constructor should be ready for insertion, search and deletion without any additional initialization.

- ```
  virtual ~ChainHashTable();
  ```
  ```

- ```
  virtual ~ProbeHashTable();
  ```

  These are the destructors. Make sure you deallocate all memory for this object. You do not have to individually delete the items stored in the hash table buckets.

- ```
  ChainHashTable (ChainHashTable& other);
  ```

- ```
  ProbeHashTable (ProbeHashTable& other);
  ```

  These are the copy constructors for your two hash table classes. Do not call the assignment operator from the copy constructor. If you do not want to have duplicate code, then create a third function that handles the common parts of the both functions.

- ```
  const ChainHashTable& operator= (ChainHashTable& rhs);
  ```

- ```
  const ProbeHashTable& operator= (ProbeHashTable& rhs);
  ```

  These are the overloaded assignment operators for your hash table classes.

  For the following member functions, we list each only once, but each should be implemented twice in specific versions for your `ChainHashTable` and `ProbeHashTable` classes. Also, all three functions need to be able to test if the data matches an item currently in the hash table. It is guaranteed that the template type `T` provides an appropriate overloaded operator== and operator!= to allow you to test for matches. (For your own test files, you would be responsible for this when you try other types.)

- ```
    virtual bool insert(const T &data);
  ```

  This function inserts `data` into the hash table. It returns true if data was new and inserted. if data is already in the hash table, it is left as-is, and the function returns false. You should not add a duplicate to the table, or even replace the current entry!

  For the ProbeHashTable class, if the table is full, an exception is thrown.

- ```
    virtual bool find(const T &data);
  ```

  The `find()` function looks for `data` in the hash table. The function returns `true` if found, `false` otherwise.

- ```
    virtual T remove(const T &data, bool &found);
  ```

The `remove()` function removes `data` from the hash table if it is found, and returns it, setting `found` to true. If `data` is not in the hash table, `remove()` returns with `found` set to false.

It is the responsibility of the code that calls `remove()` to deallocate the returned data if necessary.

When an item is removed, the slot it occupied should be marked as lazy-deleted, by setting the m_flag field to -1, so as not to break linear probing (see above).

- ```
  virtual void dump();
  virtual int at(int index, std::vector<T> &contents);
  ```

These functions are used for grading purposes, so we can examine your hash table(s). The `dump()` function does the usual: print some vital statistics and the contents of the hash table(s) to `stdout`. You should include the table size and number of items in the hash table(s). `dump()` for ChainHashTables should print out all the items in each bucket as a list, while the version for ProbeHashTables should print out the value in each hash slot, and include the item's hash index in parentheses. Look at our sample output for the exact required format. Note that you have to exactly match the format of our sample output in order to be graded correctly!

The `at()` function returns the number of items stored at the `index` slot of the hash table. For ProbeHashTables, that would obviously be either 0 or 1; for ChainHashTables, it would be 0 or more. In either case, the `contents` parameter is an out parameter, in which at() should store the item(s) found in the specified bucket. The items should be appended one at a time to the end of the vector. If the `index` is invalid (i.e., less than 0 or greater than or equal to table size), then `at()` should throw an `out_of_range` exception (already defined in `stdexcept`).

Implementation Notes

- Remember to mod out by the table size when you are working with hash table indices.
- Again, in hash tables, the indices wrap around to 0 at the bottom of the hash table. You must take this into account when you use linear probing for insert, find and remove. Note that `for` loops do not work very well in this situation, because

  ```
  for (int index = start ; index <= stop ; index++)
  ```

  doesn't do the right thing when wraparound occurs.
- You should periodically run your program under `valgrind` during development. This is so you can catch memory leaks as soon as possible. Also, if `valgrind` complains about memory read or memory write errors, this means **you have a bug in your program**. You should not ignore these errors. You should fix the bug as soon as possible because memory errors tend to manifest themselves in other places and are difficult to debug. You want to know if your program has any memory errors as soon as possible, because it is likely that the bug is in the last few modifications you have made in your program.
- Note that the usual drill in our class projects for classes is that you declare the class in a .h file, with appropriate guard lines, and define the member functions in the associated .cpp file. You would typically then compile the class's .cpp file. However, with templated classes, you actually need to include the templated function definitions into your source when you compile it, and you do not compile the templated function definitions directly. So, as with the previous project, you should #include the .h file (ProbeHashTable.h, for instance) in your Driver.cpp, and ProbeHashTable.h should in turn #include ProbeHashTable.cpp at the end (you only do this for templates!). Then, when you compile, you do ***not*** actually compile ProbeHashTable.cpp: you just compile Driver.cpp (or test1.cpp, etc.), and this brings in all the templated class definitions via the #include directives.
  ```

Test Programs

Note: this whole section is new.

The provided sources (HashTable.h, getCurrentCPU.{h,cpp}, words.h, test programs, sample output) are all available in the GL directory `/afs/umbc.edu/users/p/a/park/pub/www/cs341.s18/projects/proj5files`. You can use the "cp" command to copy these to your own directory.

The following programs should be compiled using

```
g++ -g testX.cpp getCurrentCPU.cpp -o testX.out
```

where `testX.cpp` is one of `test1.cpp`, `test2.cpp`, `test3.cpp`, etc.. Run these programs under `valgrind` to check for memory leaks and memory read/write errors.

- Basic test of both ChainHashTable and ProbeHashTable classes, instantiated for simple `int` type. Exercises default constructors, `insert()`, `find()` and `remove()`. Has CPU timing code, but tests are too simple to register any significant CPU usage. Code (`test1.cpp`) and sample output (`test1.txt`).
- A more complex version of earlier test, instantiated for C-string data, again for both ChainHashTable and ProbeHashTable classes. Again, timing not really significant. Code (`test2.cpp`) and sample output (`test2.txt`).
- Version that throws a real stress test on the hash tables, throwing 100,000 data values at it. Timing finally registers. You should see how fast your program runs, and possibly raise the number of turns of each loop up to a million or more if reasonable. Code (`test3.cpp`) and sample output (`test3.txt`).
- Additional tests might be released in the future.

What to Submit

You must submit the following files:

- `ChainHashTable.h`
- `ChainHashTable.cpp`
- `ProbeHashTable.h`
- `ProbeHashTable.cpp`
- `Driver.cpp`

The main function in `Driver.cpp` should exercise your `HashTable` functions and show what your program has implemented successfully. (I.e., if your code in `Driver.cpp` does not produce any output or it seg faults, then we will assume that you have not implemented very much.)

Do ***not*** submit the provided `HashTable.h` or `HashTable.cpp`. We will just discard them and use our own copies. They were not supposed to be modified.