CS 15 Lab 10: Hash Tables

Introduction

Hash tables are often a very fast way to store and retrieve data. The basic idea of hashing is to use an array of buckets — places to put the information based on an integer value computed from the key. The key to efficient hashing (pun!) is a good function to tell you which bucket to use for a given key. For this lab, we will explore a bad and a good hash function, and we'll also implement two strategies to handle when a bucket is already in use (collision).

Review

Collisions

Recall that the input value to a hash function is a key to hash, and the output of the hash function (after compression) is an index of your array. What happens when something is already present at that index? This is known as a **collision**. We will explore two methods of collision resolution in this lab - **linear probing** and **chaining**.

Linear Probing

For linear probing, you simply search for the next open index. So, if your hash function outputs the index 3, and something is occupying 3, then you search at index 4, then 5, and so on. Make sure to wrap around if you go over the end of the table! In other words:

```
1 indexForKey(x) = (hashFunction(x) + attempt) % table_size
```

where we start with attempt 0, and keep going until we find an empty slot. This strategy requires that there **always** must be an empty slot in the array!

Therefore, we must keep track of whether a slot is empty or not.

Chaining

Handle collisions by putting storing a linked list in each slot of the array. If there is a collision, simply add the element to the back of the linked list at that slot.

The Lab

Introduction

Hector Hash and his friends are out at a disco. Hector wants to remember his night out by storing his favorite songs played that night in a hash table. He wants to be able to quickly retrieve information associated with these songs to request them at future discos. In particular, he want to associate with each song name the position that song has on the BillBoard top 30 chart, and also the tempo of the song in beats per minute (so a DJ can do a good job blending them).

While discussing hash functions with his friends, Hector and the group come up with two hash function ideas:

1. Use the length of the song name as the hash value.
2. Use the `C++ std::hash` facility.

The friends also disagree on which way of handling collisions is best. They consider probing and chaining as options. Hector has started writing a class which implements both ideas for the hash function, as well as both collision resolution techniques, but he needs your help to finish the job!

Representing Songs

For the table, Hector is using `C++ string`s to represent the song titles (the keys), and the following `struct` to represent the values:

```
struct SongInfo {
    int chartPosition;
    int bpm;

    // Constructors for struct
    SongInfo()
    {
        chartPosition = bpm = -1;
    }

    SongInfo(int position, int tempo)
    {
        chartPosition = position;
        bpm           = tempo;
    }
};
```

FunkeyTable

Hector is writing a `FunkeyTable` class, which effectively maintains two hash tables - inside the class, there are two private member arrays - one will be used for the chaining technique, and the other for linear probing.

You'll find that these arrays are of different types - `TableEntry` objects will be used for the slots in the array in the linear probing method, whereas (pointers to) `ChainNode` objects will be used for the chaining method. These `structs` are both defined in `FunkeyTable.h`.

Funkey Details

You'll notice a few details in the `Funkey` class. For instance,

```
typedef std::string   KeyType;
typedef SongInfo      ValueType;
```

These type definitions allow us to abstract the FunkeyTable class over the details of the key and value type. In the implementation, the only code that depends on knowing the types is the print function, which could be removed after debugging. Then we could make this a template! Similarly,

```
enum HashFunction {BAD_HASH_FUNCTION, GOOD_HASH_FUNCTION};
```

This allows us to use variables of type `HashFunction`, whose values are either `GOOD_HASH_FUNCTION` or `BAD_HASH_FUNCTION`. This is simply a convenience to make our code more readable.

Functions to Write

Your job will be to write the destructor and the two insertion functions for the class (along with any helper functions you'd like to write). If you have time, you can also write the expansion function. The functions you will write for this lab are:

```
FunkeyTable::~FunkeyTable();
```

The destructor for the class. Be sure to reclaim all the space associated with the tables used for linear probing and chaining, inluding all the nodes in the chained table. You may find it useful to write `FunkeyTable::deleteList`.

```
FunkeyTable::insertProbing(KeyType key, ValueType value,
                           HashFunction hashFunction);
```

Adds a song and its information to the hash table using using either the `GOOD_HASH_FUNCTION` or the `BAD_HASH_FUNCTION`. This function handles collisions by using the linear probing method.

```
FunkeyTable::insertChaining(KeyType key, ValueType value,
                            HashFunction hashFunction);
```

Adds a song and its information to the hash table using either the `GOOD_HASH_FUNCTION` or the `BAD_HASH_FUNCTION`. This function handles collisions by using the chaining method. You may find it helpful to write a `listLength` function to help with this one.

```
FunkeyTable::expand();
```

Expand both tables to accommodate more values. You should expand when the load factor, which is the number or items in the table divided by the table capacity, exceeds something like `0.7`.

Other Notes

Make a directory for this lab and get the files from the usual place. You can use the `make` command to compile and link the files. The executable generated is named `funkey`. Submit your work as discussed in class. Well done!