Homework - Hash Tables based on Linked Lists (Open Hashing)

Brad Peterson – Weber State University

Goal:

- To implement a hash table with some standard methods. The hash table must support templated keys and values. The methods needed are an L-value create(), an R-value create(), exists(), retrieve(), operator[](), update(), and remove() methods.
- To practice using std::move()
- To complete an assignment that involves many interacting parts

Overall needs:

This assignment works by utilizing six interacting items. They are:

1) The STL hash function
2) The STL's pair class.
3) The STL's forward_list class (it manages its own nodes)
4) A HashTable class
5) An array of forward_list objects
6) The forward_list's iterators.

Details of each are given below.

STL Hash Function

C++ gives a build in hash function for you. For this assignment, you can use it one of two ways:

```
hash<string> myHashObject;
myHashObject(key);
```

or

```
hash<string>{}(key)
```

Both return a large integer. You need to mod that large integer by the number of buckets.

STL Pair class

This class has two template parameters. The first data member is called "first" and uses the first template parameter. The second data member is called "second" and uses the second template parameter. This assignment uses the pair's first to hold the key, and pair's second to hold the value. Thus, the first template argument should be string, the second should be T.

You can create objects with std::pair one of two ways:

```
pair<first_data_type, second_data_type> pairObject(data1, data2);
```

Or:

```
pair<first_data_type, second_data_type> pairObject;
pairObject.first = data1;
pairObject.second = data2;
```

(Note that if you are passing a new pair object into a method, you can create the object on-the-fly and you don't need to give your object a name.)

STL forward_list class

This class implements a singly linked list. You can use push_front() to insert items, iterators to iterate through its linked list or to set up a ranged-based for loop, and the remove_if() method which lets us provide our own node removal logic based on the key. (Recall that by using the forward_list class, you do not have direct access to nodes.)

HashTable class

A template class templated on one parameter T.

The private data members are:

- An unsigned integer to hold the number of buckets (the size of the array of linked lists).
- A pointer that will be used to point to an array of forward_list objects:
  ```
  forward_list< pair< string, T > >* arr{ nullptr };
  ```
  - Remember that if you had an int* arr, your constructor would have this->arr = new int[5]. For this assignment your array's data type isn't int, it's `forward_list< pair< string, T > >`.

The public members are:

- A constructor that accepts a number of buckets. The constructor should dynamically create an array of forward_lists of pair objects holding string and T data types. The array size is equal of the number of buckets passed in.
- A destructor which deletes the array (already completed)
- A move constructor (already completed)
- An L-value create() method. The key is a string, the value is a T. Use const and by-reference as appropriate. Hash the key, mod by the number of buckets. The return value of the hash gives you which linked list to use. Insert both the key and value into the linked list using push_back. Note that you must insert a pair object.
- An R-value create() method. The key is a string, the value is a T. Use const and by-reference as appropriate. The value should not be const, and should be &&. Hash the key, mod by the number of buckets. The return value of the hash gives you which linked list to use. Because this is an R-value, we desire avoiding copies. C++'s forward_list has a slick method called emplace_front(), which allows you to directly pass in the key and value without requiring a pair object. It will make the pair for you. Just note that you still have to use std::move() on the value.
- An exists() method. Accepts a key and returns a bool. Find the appropriate linked list. Use either forward_list iterators directly or a ranged-based for loop to iterate through all values in the specific linked list associated with that key. Check if the current key found via iteration matches the key passed in. If so, return true. If after all iteration no keys matched, return false.
- A retrieve() method. Accepts a key and returns T. Find the appropriate linked list. Use either forward_list iterators directly or a ranged-based for loop to iterate through all values in the specific linked list associated with that key. Check if the current key found via iteration matches the key passed in. If so, return the value. If after all iteration no keys matched, throw 1.
- An operator[]() method. Just like retrieve(), except it returns T&.
- An update() method. Accepts a key and a value. Returns nothing (void). Like retrieve, except where you would return the value, you instead update the value. If no keys matched, just inform the user and return.
- A remove() method. Accepts a key and returns void. Find the appropriate linked list. Use the forward_list's remove_if() method to make this method work. This remove_if() requires you to pass in a function, a functor, or a lambda. (The function cannot work for our needs because we need to also pass

in the key we want to match. The functor could, but that is too much unnecessary work.) We will use a lambda.
A lambda can be thought of has having the syntax []{}. The [] is the capture part, the () is the parameter part, and the {} is where code codes. Use the [] part to capture the key passed into remove() so that the key can be found within the upcoming code's scope. Use the () part to accept the node's data as a parameter (the pair object) that remove_if() will provide each iteration. Use the body of code to compare if the captured key and the parameter key matched. If so, return true, otherwise, return false. Note that remove_if() does all the iterating and node removal of you. You don't need to loop, and you don't have to manually remove nodes. In the lambda body of code, you just simply check if the pair's key matches the captured key. If so, return true, otherwise, return false.

Note that some of these hash table methods can be done in a single line of code. If you want to use multiple lines of code, just make sure you don't make copies of a linked list. For example, this is bad:

```
auto some_list = /*code which gets a particular linked list*/;
```

But this avoid the copy and is good:

```
auto& some_list = /*code which gets a particular linked list*/;
```

Tips for success:

- Comment in only the constructor usage prior to the first test. Get that to compile and work. Then comment in the first test. Get that to compile and work and use the debugger to verify.
- Watch the lecture video carefully. I gave many hints there. This assignment is tricky because there are so many interacting parts.
- Let each interacting part do its own job. Do NOT ignore the structure and lazily attempt to force things to work. For example, a HashTable has no ability to access a create a Node. Let the LinkedList do the inserting for you. Let the linked list's iterators do the iterating for you.
- Start early. This assignment is much more about proper object-oriented structure rather than debuggable logic. It is also often regarded as the hardest assignment of the semester.