Project 4 - Map ADT: HashTable

Learning Objectives

- Implement a data structure to meet given specifications
- Design, implement, and use an open addressing hash table data structure
- Use a hash table as a Map ADT

Overview

The Map ADT, sometimes called a Dictionary, allows access to item *values* based on a *key*. This is often denoted as: `<KeyType, ValueType>`. For example, a map could store customers' names and be able to look them up using their customer ID. If the customer ID were an integer and the customer's name a string, the *key* for the map would be a `size_t`, and the *value* would be a `std::string`.

Alternatively, we could store the customer IDs in the Map and use the customer's name to look up their ID. In this case, the *key* would be a `std::string`, and the *value* would be an `size_t`. For our Map, we will be using `std::strings` for *keys* and `size_t` for *values*. If we were using templates, our map would be declared like:

```
Map<std::string, size_t> myMap
```

However, to make things easier on us using C++, we're going to fix the types of the *key* and *value* to strings and integers.

Hash Table Representation

To keep things relatively simple, we are going to use a C++ `std::vector` to store our hash table data. This representation will make it easier when having to resize the table, as the `std::vector` can dynamically resize, where a regular array would need to be manually resized.

Because the `std::vector` needs to store *key-value* pairs, we will need a data structure to store both of them. In addition, we will need store more information about each bucket, such as whether is has a *value* or it is empty.

Since each index of our `std::vector` will store a single *key-value* pair, we will need a collision resolution policy. For our hash table, we will use pseudo-random probing.

The `HashTable` Class

You will implement class HashTable in a header and source file named `HashTable.h` and `HashTable.cpp`, respectively. The header file will contain the class declaration with all member variables and methods.

You need to implement, at a minimum, the following methods/functions:

Code Listing 1: Core methods of HashTable

```
/**
* Only a single constructor that takes an initial capacity for the table is
* necessary. If no capacity is given, it defaults to 8 initially
 */
HasTable::HashTable(size_t initCapacity = 8)
```

```
/**
* Insert a new key-value pair into the table. Duplicate keys are NOT allowed. The
* method should return true if the insertion was successful. If the insertion was
* unsucessful, such as when a duplicate is attempted to be inserted, the method
* should return false
 */
bool HashTable::insert(const std::string$ key, const size_t& value)

/**
* If the key is in the table, remove will "erase" the key-value pair from the
* table. This might just be marking a bucket as empty-after-remove
 */
bool HashTable::remove(const std::string& key)

/**
* contains returns true if the key is in the table and false if the key is not in
* the table.
 */
bool HashTable::contains(const string& key) const

/**
* If the key is found in the table, find will return the value associated with
* that key. If the key is not in the table, find will return something called
* nullopt, which is a special value in C++. The find method returns an
* optional<int>, which is a way to denote a method might not have a valid value
* to return. This approach is nicer than designating a special value, like -1, to
* signify the return value is invalid. It's also much better than throwing an
* exception if the key is not found.
 */
std::optional<size_t> HashTable::get(const string& key) const

/**
* The bracket operator lets us access values in the map using a familiar syntax,
* similar to C++ std::map or Python dictionaries. It behaves like get, returnin
* the value associated with a given key:

 int idNum = hashTable["James"];
* Unlike get, however, the bracker operator returns a reference to the value,
* which allows assignment:

 hashTable["James"] = 1234;

 If the key is not
* in the table, returning a valid reference is impossible. You may choose to
* throw an exception in this case, but for our implementation, the situation
* results in undefined behavior. Simply put, you do not need to address attempts
* to access keys not in the table inside the bracket operator method.
 */
size_t& HashTable::operator[](const string& key)
```

Code Listing 2: Additional HashTable methods

```
/**
* keys returns a std::vector (C++ version of ArrayList, or simply list/array)
* with all of the keys currently in the table. The length of the vector should be
* the same as the size of the hash table.
 */
std::vector<std::string> HashTable::keys() const

/**
* alpha returns the current load factor of the table, or size/capacity. Since
* alpha returns a double,make sure to properly cast the size and capacity, which
* are size_t, to avoid size_teger division. You can cast a size_t num to a double
* in C++ like:
                  static_cast<double>(num)
 The time complexity
* for this method must be O(1).
 */
double HashTable::alpha() const


/**
* capacity returns how many buckets in total are in the hash table. The time
* complexity for this algorithm must be O(1).
 */
size_t HashTable::capacity() const

/**
* The size method returns how many key-value pairs are in the hash table. The
* time complexity for this method must be O(1)
 */
size_t HashTable::size() const
```

Table Data

To store the hash table data, you will use the C++ `std::vector` class. For reference, https://cppreference.com/w/cpp/container/vector lists the methods available for `std::vector`. The ones of most interest to us will be `at` and `operator[]` for accessing elements, `size` to retrieve the number of elements in the `vector` (this is not the number of non-empty buckets, but how many buckets are in the `vector`), and `resize` to change the number of elements.

The data type stored in the `std::vector` should be `HashTableBucket`, another class you will implement. Thus, you can declare the table data member of HashTable like:

```
std::vector<HashTableBucket> tableData;
```

`operator<<`

In addition to the methods of `HashTable`, you will also implement an operator to easily print out the hash table. This is not a method of the `HashTable` class, and as such does not have access to the private data of `HashTable`.

```
/**
* operator<< is another example of operator overloading in C++, similar to
* operator[]. The friend keyword only needs to appear in the class declaration,
* but not the definition. In addition, operator<< is not a method of HashTable,
* so do not put HashTable:: before it when defining it. operator<< will allow us
* to print the contents of our hash table using the normal syntax:
 cout <<
* myHashTable << endl;
   You should only print the buckets which are occupied,
* and along with each item you will print which bucket (the index of the bucket)
* the item is in. To make it easy, I suggest creating a helper method called
* something like printMe() that returns a string of everything in the table. An
* example which uses open addressing for collision resolution could print
* something like:

 Bucket 5: <James, 4815>
 Bucket 2: <Juliet, 1623>
 Bucket
* 11: <Hugo, 42108>
 */
friend ostream& operator<<(ostream& os, const HashTable& hashTable)
```

The `HashTableBucket` Class

To store each *key-value* pair, you will need to implement **`HashTableBucket`**. This is an internal implementation detail, so there are no specific requirements for the class except it needs to store a `std::string` for the *key* and an `size_t` for the *value*. In addition, you will need to have a way to indicate whether the bucket is:

- **Normal** – the bucket is non-empty and currently storing a *key-value* pair
- Empty Since Start (**ESS**) – the bucket has never had a *key-value* pair
- Empty After Remove (**EAR**) – the bucket previously stored a *key-value* pair, but that pair was later removed from the table.

A simple way to accomplish this would be to create a C++ enum, which you can define like:

```
enum class BucketType {NORMAL, ESS, EAR};
```

Inside `HashTableBucket`, you would then have a member variable of type `BucketType` to keep track of the bucket state. As for methods, you may implement any member functions you feel necessary, but here are some recommended ones to consider:

```
/**
* The default constructor can simply set the bucket type to ESS.
  *
 */
HashTableBucket::HashTableBucket()


/**
* A parameterized constructor could initialize the key and value, as
* well as set the bucket type to NORMAL.
 */
HashTableBucket::HashTableBucket(const std::string& key, const size_t& value)
```

```
/**
* A load method could load the key-value pair into the bucket, which
* should then also mark the bucket as NORMAL.
 */
void HashTableBucket::load(const std::string& key, const size_t& value)

/**
* This method would return whether the bucket is empty, regardless of
* if it has had data placed in it or not.
 */
bool HashTableBucket::isEmpty() const

/**
* The stream insertion operator could be overloaded to print the
* bucket's contents. Or if preferred, you could write a print method
* instead.
 */
friend ostream& operator<<(ostream& os, const HashTableBucket& bucket)
```

In addition to these suggestions, you could also provide accessors (getters) and mutators (setters) for the member data. You could also have methods to change the bucket type, such as `makeNormal()`, `makeESS()`, and `makeEAR()`. Other methods that could be useful might be `isEmptySinceStart()` and `isEmptyAfterRemove()` which could check for each of those conditions, and an `isEmpty()` which just checks whether the bucket is normal or empty.

Pseudo-random Probing Function

For this project, you must implement pseudo-random probing as the probe function. Pseudo-random probing means the sequence of probe indices is generated from a deterministic pseudo-random sequence rather than a simple linear or quadratic sequence. You will store the probe offsets in a `std::vector` of `size_t`, which could look like:

```
std::vector<size_t> offsets;
```

For a table with $N = 10$, we want the vector to store the values 1 to $N - 1 = 9$ shuffled in a random order. For instance, it could look like this:

4	6	9	7	3	5	1	8	2

When probing, if the home position of a key is 8, the first bucket probed in the sequence would be:

```
probe_0 = (home + offsetArray[0]) % N
        = (8 + 4) % 10
        = 2
```

The next bucket in the probe sequence would be `(8 + 6) % 10 = 4`.

Table Resizing

In addition to the above methods, your hash table will need to be able to dynamically grow as the user inserts data. Your initial table capacity should be 8, meaning the size of your `std::vector` should start with 8 empty buckets (for chaining it will be 8 empty lists of buckets). To keep it simple, you may double the size of your `std::vector` when necessary. The table will need to be resized once the load factor (alpha) reaches or exceeds **0.5**.

You will also need to generate a resized offsets array. When the table grows, you need to have enough offsets to accommodate probing the resized `std::vector`. There is no need to shrink the table.

Time Complexity Analysis

Based your choice of data structure for your data table, you will report the time complexity of:

- `insert`
- `remove`
- `contains`
- `get`
- `operator[]`

Give a short justification for why each method is your stated time complexity. Put your analysis in the README.md file of your git repository

Turn in and Grading

You will commit `HashTable.h` and `HashTable.cpp`, along with any additional files you create necessary to compile your HashTable, and push those commits to the remote branch of your GitHub Classroom repository for this assignment. You should make a habit of committing your code **regularly** and **frequently**. If your repository shows one large commit and few others, that will flag your submission for potential academic integrity violation.

You should include your time-complexity analysis in the README.md of your repository.

This project is worth 50 points, distributed as follows: Task Points

Test	Points
`HashTable::insert` stores *key-value* pairs in the hash table using appropriate hashing and collision resolution, and correctly rejects duplicate keys. `insert` correctly re-uses space from previously deleted records	5
`HashTable::remove` correctly finds and deletes records from the table without interfering with subsequent search and insert operations	5
`HashTable::contains` correctly returns `true` if the *key* is in the table and `false` otherwise	5
`HashTable::get` correctly finds keys using the appropriate hashing and collision resolution, and returns the *value* associated if the *key* is found. Returns `std::nullopt` when the *key* is not in the table	3
`HashTable::operator[]` correctly returns a reference to the *value* associated with the given *key*. If *key* is not in the table, operator use results in undefined behavior.	3
`HashTable::keys` correctly returns a `std::vector` with all of the keys currently stored in the table	2
`HashTable::alpha` correctly calculates the load factor of the table. Operation is $O(1)$	2
`HashTable::capacity` correctly returns the total number of buckets (both empty and non-empty) in the table, and size correctly returns how many *key-value* pairs are currently stored in the table. Both operations are $O(1)$	2
`HashTable` dynamically grows in accordance with constraints described above	5

Test	**Points**
`operator<<` is correctly overloaded as described above	5
Analysis correctly identifies time complexities of your methods.	8
Code is well organized, documented, and formatted according to the course Coding Guidelines.	5
Total	**50**