# Project #1 – Map ADT: Hash Table

## Learning Objectives

- Implement a data structure to meet given specifications
- Design, implement, and use a close 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, we 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, our key would be an integer, and the value would be a 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 string, and the value would be an integer. For our Map, we will be using strings for keys and integers for values. If we were using templates, our map would be declared like:

```
Map<string, int> 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

As discussed in class, there are two main ways of handling collision resolution: chaining and probing. Both approaches involve utilizing an array, but the process for handling collisions is different. Additionally, the storage required for each collision resolution approach differs. For chaining, you would use an array where every index of the array contained a list of table entries. For probing, the array would contain buckets that held one entry each.

For this project, you are to decide whether to utilize chaining or probing. Depending on your choice, you will need to add the appropriate data structure into your HashTable class (discussed below).

Additionally, if you choose probing, you have a further choice to make. For this project, you must decide between pseudo-random probing and double hashing.

## HashTableBucket Class

Provided for you along with the HashTableClass, our buckets will store string as the key and int as the value. There is also an enum BucketType to distinguish between empty since start, empty after removal, and normal (occupied) buckets. zyBooks has a different method for distinguishing bucket types, and you may use that one instead, but I think the enum approach is easier.

## The HashTable class

You will be provided with two files: HashTable.h and HashTable.cpp. You should not modify any of the code given to you, you will only be adding code. HashTable.h, also known as the header file, has the declaration of all the methods. The header file is where you will decide on the appropriate data structure for your hash table depending on your collision resolution approach. You will need to add any member data, placing them in the appropriate visibility sections.

HashTable.cpp has stubs for all the methods for the HashTable and HashTableBucket (discussed below) classes. The HashTableBucket methods have been written for you. Leave the existing methods as they are, but if you feel you need more functionality you may add member data or functions to HashTableBucket.

The majority of what you will write are the bodies of the HashTable methods.

**`HashTable::HashTable(size_t initCapacity)`**
There is one constructor for the HashTable that takes the initial capacity for the table. Depending on your collision resolution strategy, this value will mean different things.

**`bool HashTable::insert(const string& key, int value)`**
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 unsuccessful, such as when a duplicate is attempted to be inserted, the method should return false.

**`bool HashTable::remove(const string& key)`**
If the key is in the table, remove will "delete" the key-value pair from the table. Depending on your collision resolution, this might either be marking a bucket as empty-after-remove or removing an element from a list.

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

**`optional<int> HashTable::get(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.

**`int& operator[](const string& key)`**
The bracket operator will allow us to use our map the same way various programming languages such as C++ and Python allow us to access values. The bracket operator will work like `get` in so that it will return the value stored in the bucket with the given key. We can get the value associated with a key by saying:

```
int idNum = hashTable["James"];
```

However, the bracket operator returns a reference to that value. This means we can update the value associated with a key like:

```
hashTable["James"] = 1234;
```

The one issue comes when the key is not in the table. For that case, there are options like the ones we had with get. However, returning a reference to a value not in the table is impossible, since it's not in the table. So, you may opt to throw an exception if you want, or you can return a reference to a value that is not associated with the key. This is called undefined behavior, and for us it is just something we will have.

**`vector<string> HashTable::keys() const`**
The keys method will return a 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.

**`double HashTable::alpha() const`**
The alpha method returns the current load factor of the table, or size/capacity. Since alpha returns a double, make sure to properly cast the values to avoid integer division. The time complexity for this method must be O(1).

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

**`size_t HashTable::size() 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)

In addition to these 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.

**`ostream& operator<<(ostream& os, const HashTable& hashTable)`**
Operator<< is another example of operator overloading in C++, similar to the bracket operator. This 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>
```

If your hash table uses chaining, there may be multiple key-value pairs located in the same bucket.

## Table Data

You will have the choice between using chaining or open addressing to store the data. There are a few caveats for each choice:

### Chaining

If you decide to implement chaining, you **must** implement the bucket lists as a **linked list** of your own, and not utilize one of the C++ containers such as list or vector. However, you are allowed to use the vector class to hold all the lists. Your data table might look like

```
vector<MyGreatLinkedList> dataTable;
```

And the nodes in `MyGreatLinkedList` would store a bucket and a pointer to the next node.

### Open Addressing

If you opt to implement open addressing, you **must** implement the storage array manually. You are not allowed to use a C++ container, but instead you will manually manage an array of `HashTableBucket` objects. Your data table will look like:

```
HashTableBucket* dataTable;
```

*Failure to comply with these requirements will result in a zero for the assignment.*

## Probe Function (Open Addressing)

If your hash table utilizes open addressing, you must decide on a probe function to handle collisions. For this project, you must choose between **pseudo-random probing** and **double hashing**. Both approaches will need to handle when the table is resized.

## 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 array/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 array when necessary. You will choose which metric to use to decide when to increase the size of the table among the ones discussed in class:

- Load factor goes above 0.5
- Number of collisions during an insert is more than N/3
- Length of a bucket's list is greater than 3

The first two approaches pertain to open addressing implementations, the last one would apply to a hash table with chaining.

## Time Complexity Analysis

Based your choice of data structure for your data table, you will report the time complexity of: **`insert, remove, contains, get,`** and **`operator[].`** Give a short justification for why each method is your stated time complexity. You may have these as comments for each of the stated methods.

## Turn in and Grading

You will submit the HashTable.h and HashTable.cpp with your implementation. Additionally, you may have additional files for your linked list or array class. Zip all of your source files into a zip file named project1.zip and upload it to the Pilot Dropbox for the assignment.

This project is worth 50 points, distributed as follows:

| **Task** | **Points** |
|---|---|
| `HashTable::insert` stores key/index 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 | 6 |
| `HashTable::remove` correctly finds and deletes records from the table without interfering with subsequent search and insert operations | 6 |
| `HashTable::contains` correctly returns true if the key is in the table and false otherwise | 6 |
| `HashTable::get` correctly finds keys using the appropriate hashing and collision resolution, and returns the value associated if the key is found. Returns `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 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) | 3 |
| `HashTable` dynamically grows in accordance with constraints described above | 6 |
| `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 |