Introduction

Welcome to assignment 7, in the assignment you are tasked with implementing a `Hash Table`.

A hash table in C/C++ is a data structure that maps keys to values. A hash table uses a hash function to compute indexes for a key. You can store the value at the appropriate location based on the hash table index.

The benefit of using a hash table is its very fast access time. Typically, the time complexity (amortized time complexity) is a constant `O(1)` access time.

If two different keys get the same index, you will need to use other data structures (buckets) to account for these collisions. If you choose a very good hash function, the likelihood of a collision can be negligible.
The C++ STL (Standard Template Library) has the `std::unordered_map()` data structure.

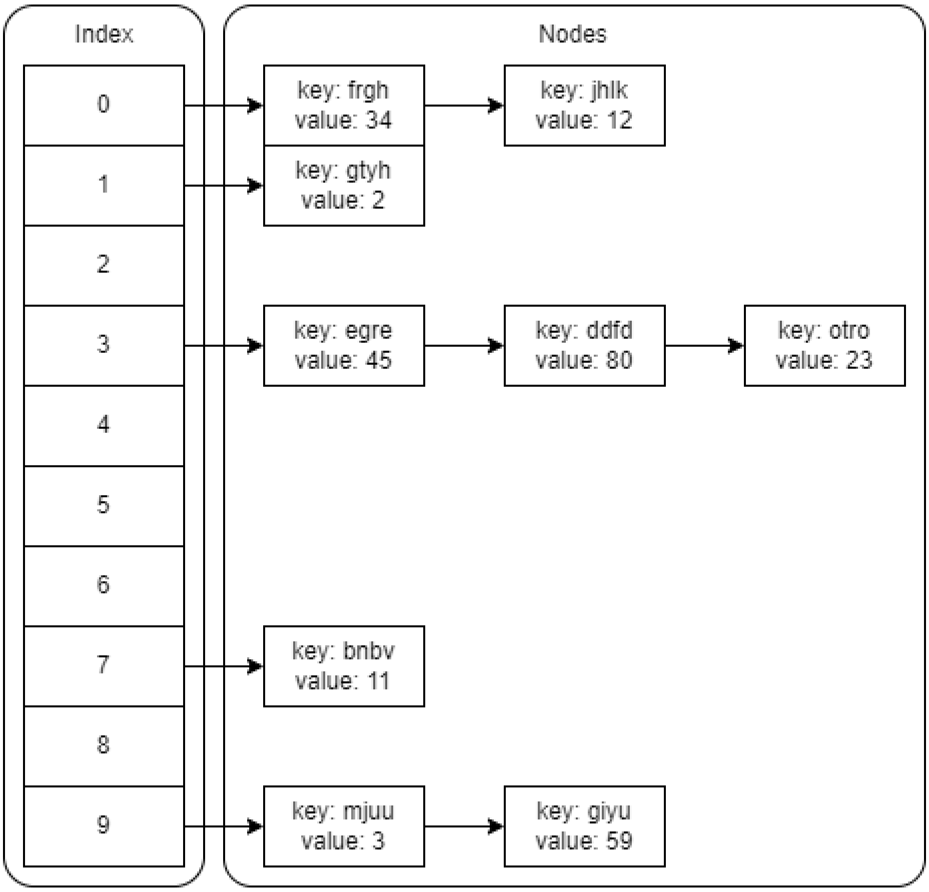
In this assignment, you will construct a hash table from scratch comprised of:

- A hash function to map keys to values.
- A hash table data structure that supports `Insert`, `Get`, and `Delete` operations.
- A data structure to account for a collision of keys.

Hash Tables

Hash Tables are a type of data container. The way they store data is by holding an array of linked lists (also commonly referred to as buckets).

- Here is an example of the memory layout in a Hash Table:



- Hash Tables rely on hash functions to sort the data into their buckets.
  Hash functions work by taking in a key and applying a user-defined algorithm to generate an index corresponding to the bucket to store that value.
  `key` -> `HashFunction()` -> `index`

Hash Table Structure Layouts

HashTable Structure

```
// Structure for the hash table
typedef struct _HashTable
{
    Node* table[TABLE_SIZE];
} HashTable;
```

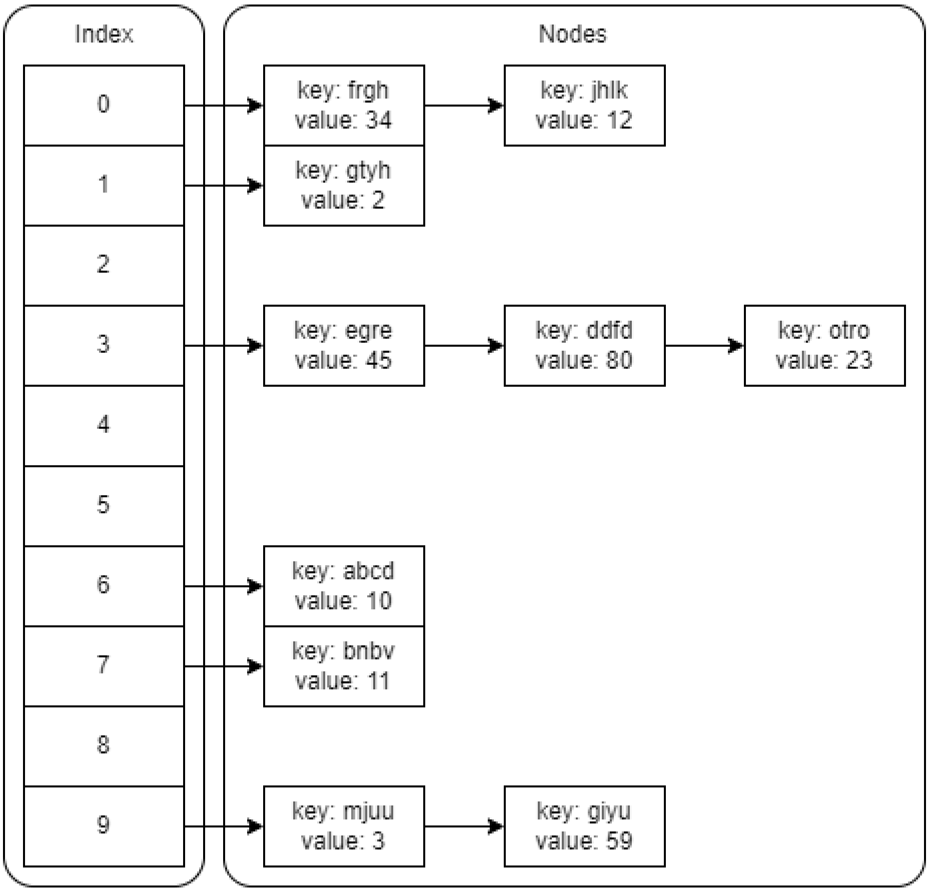
Node Structure

```
// Structure for a node in the linked list
typedef struct _Node
{
    char key[50];
    int value;
    struct _Node* next;
} Node;
```

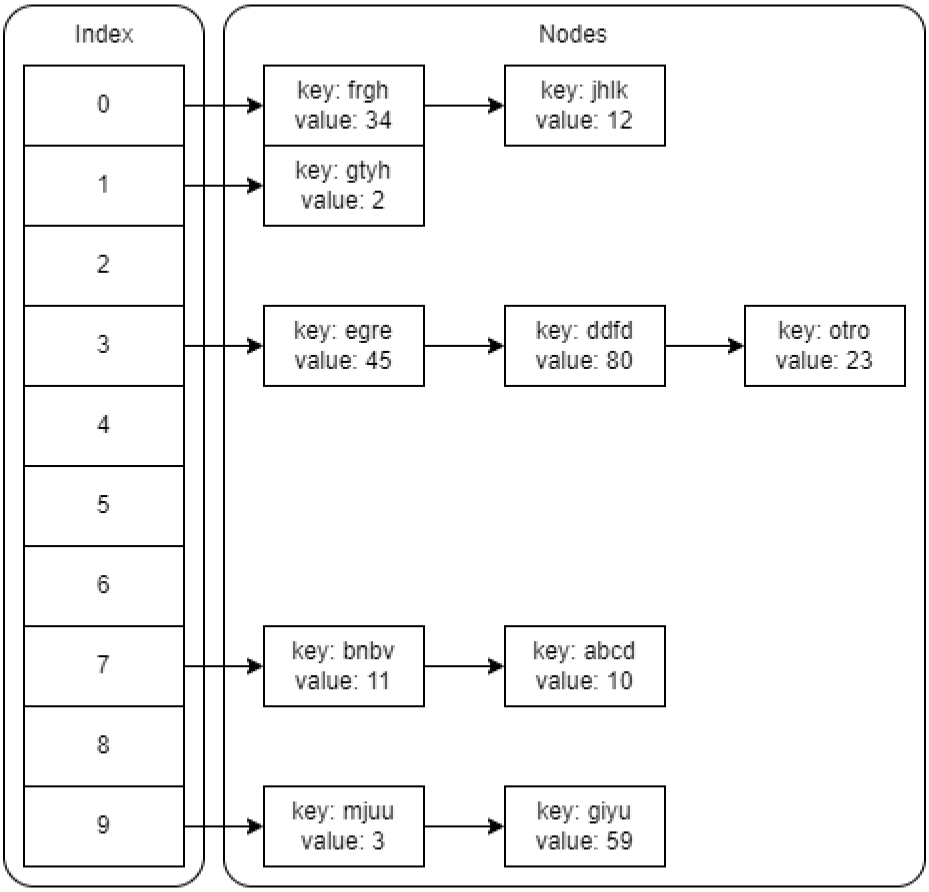
- **Note:** *TABLE_SIZE should be defined as a constant `10`*

Hash Table Insertion

- To insert a new item `{ key: "abcd", value: 10 }` into the Hash Table, first we run the key through the hash function to generate an index from `0 - (TABLE_SIZE-1)`. Let's say the index generated for `abcd` is `6`, so then the value `10` is stored in the bucket at the 6th index of the Hash Table.



- In the event that the index genrated was `7` instead, the value `10` would be added as the next element in the bucket at the 7th index of the Hash Table.



- **Note:** *Each `Node` is dynamically allocated and added to the bucket for every value inserted.*

Hash Table Accessing

The process to retrieve values from the Hash Table is similar to insertion. Lets say we want to retrieve value with the key `abcd`.

- Run the key through the hash function to get the index of the data's bucket, in this case `7`.
- Iterate through the bucket till you reach the `Node` with the desired key and return the value.

Hash Table Printing

- Iterate through the array of buckets.
- Iterate through each `Node` in each bucket, printing out it's value.

```
[0]->34->12
[1]->2
[2]
[3]->45->80->23
[4]
[5]
[6]
[7]->11->10
[8]
[9]->3->59
```

Hash Table Deletion

- Iterate through the array of buckets.
- Iterate through each `Node` in each bucket, deleting the nodes.

Requirements

1. You are tasked with creating and implementing the `hash-table.h` and `hash-table.c` files which declares and defines the Hash Table logic described in the above section.
2. You are allowed to use any techniques/programming patterns previously discussed in our lessons in your implementation.
3. Run the `!run.bat` file to run the suite of tests to test your Hash Table implementation.
4. You are to ensure that there are no memory leaks in your final program submission.