**Hash Table Project: Code Documentation**

Brandon Michelsen

**Introduction:**

Hash tables are one of the most elegant data structures in the programmer’s toolbox. The hash table’s method of data insertion and retrieval makes both these operations very fast in most cases. By using a key and hashing method, as will be seen later, hash tables are capable of storing data very efficiently.

A hash table combines three separate elements of programming: arrays, a hashing function, and key-value pairs. Arrays, being a more primitive data structure, are relatively easy to understand. The array starts with a pointer to the first element in memory, and all other memory locations are stored in a contiguous block. Each element has a specific index value associated with it. This is what indicates where the element is stored in the array (i.e. its position).

Key-value pairs are relatively easy to understand, as well. When given a value, there is a certain key associated with it that identifies that value. The key can be used to access the value. An example would be an apartment mailbox. There are many mailboxes, each with a different number. However, each mailbox number is associated with a different resident. For example, mailbox 31 might be associated with Mrs. Weller. In this case, the key would be 31 and the value would be Mrs. Weller.

Finally, the hashing function needs to be explained. A hash function is simply a way of taking some data and giving it a numeric identifier. For example, running a hashing algorithm on a string of characters will result in some numeric output that identifies the string. A hash function that could be used in this case would be to count the number of characters in the string. As an example, feeding in the value “Bob” will produce a numeric output of 3. This is the basics of what a hash function does.

Now, these three elements need to be combined to create the hash table. The array makes up the table itself. It stores key-value pairs at specific indexes given by the key. The key-value pair and the hash function are used for insertion, retrieval, and removal. In the case of insertion, the key is put through a hash function in order to get the index of where the data needs to be stored in the table. Then, the key-value pair is stored at that index. That is all that needs to happen – in the simplest case – for a hash table to store data. It is a very elegant solution.

Retrieval works in a similar manner. The key of the data the user would like to retrieve is passed to the retrieval function and is put through the hash function again to get the index of where the data is stored. Then, the retrieval function returns the data at that index.

Although a hash table is very elegant in how it handles data storage, there is an issue with this simple method: what happens when two keys evaluate to the same index? This scenario is known as a collision. Two data elements are vying for the same location in the table, and this needs to be resolved. The simplest way to resolve this is to replace the old data at that index with the incoming data. But this results in data loss, which is never good for a data structure. Another method to handle collisions would be to store a linked list at each index in the table. Every data element whose key evaluates to a certain index will be added to the list of elements at that index. This is great for keeping data under a single index, but it can become cumbersome as the list grows.

Another option, and the final one to be analyzed here, is open addressing. There are several different methods of open addressing collision handling, but they all involve taking the colliding data element and giving it a new index in the table. The simplest form of open addressing collision handling is linear probing, where the data the colliding data element simply progresses through contiguous locations in the table until an empty location is found. The other two most common methods of open addressing are quadratic probing, where the data index is squared until a new location is found, and double hashing, where the key is put through a secondary hashing function to get a new index.

Removal is very simple to handle with a hash table. Simply add a flag to as part of data elements that indicates whether that element is “removed” or not. This ensures that collision handling operates properly – specifically linear probing – while also ensuring that the program recognizes the data as being removed, and that the index associated with the data is free to store new data.

Those are the three basic functions of a hash table. By using key-value pairs and hash functions, hash tables are able to keep insertion and retrieval operations relatively simple. The main complexity comes in when collisions between data elements needs to be handled, and even this can be handled in a very simple manner. Altogether, hash tables are a very elegant data structure.

**Analysis:**

The best-case runtime for hash table insertion, retrieval, and removal is O(1). This is because, in the best-case scenario, all that needs to happen for the data to be stored is for the index to be found and that location in the table to be set to store the data. Both these operations run in constant time, and so the amount of time it takes to run these operations does not increase with the amount of data. The average-case runtime is also O(1).

The worst-case runtime for the above operations is O(n). This is due to the collision handling algorithms involved with the table. As an example, we can analyze linear probing. If no collisions happen with insertion, then the insertion time is O(1). However, if a collision happens, then the data needs to go through each contiguous memory space until an empty location is found. In the worst-case scenario, all the contiguous memory locations will need to be searched. This leads to a worst-case runtime of O(n).

Below is a chart displaying all the measured runtimes of the hash table, using both 1 element and 10 elements (note: the resizing algorithm’s runtime was also measured):

|  |  |  |
| --- | --- | --- |
| **Operation** | **10 Elements Runtime** | **1 Element Runtime** |
| Insert | 0.000166 | 2.473e-05 |
| Retrieval | 0.000113 | 3.055e-05 |
| Resize & Insert | 0.000208 | Calculated altogether |
| Resize & Retrieval | 0.000154 | 2.667e-05 |

As can be seen, insertion on 10 elements actually took the longest time. Compared to the insertion time of a single element, we can see that, with the data I was using, the insertion time increases with the number of elements, which gives a Big-O of about O(n), as we would expect.

**Programmer’s Guide**

This section describes how the class for the hash table is meant to be used.

**Hash Table Class:**

|  |
| --- |
| **Class: HashTable** |
| template<typename K, typename V, typename H> |
| **Member Variables:** |
| int TABLE\_SIZE  int currentSize  TableEntry\*\* table |
| **Member Structs:** |
| struct TableEntry |
| **Member Functions (Public):** |
| HashTable()  ~HashTable()  int getTotalSize()  int getCurrentSize()  TableEntry\*\* getTable()  int hash(K key)  void insert(K key, V val)  void remove(K key)  TableEntry\* retrieve(K key) |
| **Member Functions (Private):** |
| void resize(K key, V val)  void insertPrivate(TableEntry\*\* tempTable, int newSize, K key, V val) |

The HashTable class is a templated data structure that stores key-value pairs of types K and V respectively (K and V are generic data types). The table is comprised of an array of pointers to data types of TableEntry structs. Each TableEntry struct contains the key and value to be stored, as well as a flag indicating if the entry has been removed. Insertion, retrieval, and removal are all accomplished by passing the given key through a generic hashing function, given by the template parameter H. The hashing function gives a numeric representation of the data, which is then used with the modulo operator and the table size to get the index of the data. The index can then be used the insert the data, remove the data, or return the data for further use.

There is also a private resizing function that can resize the table. The table will only increase in size after the previous table is completely filled and more data needs to be entered. The table size is multiplied by 3 every time the table grows. There is a private insertion helper function for the resize function.

**Member Variables**

|  |  |
| --- | --- |
| Variable | Description |
| TABLE\_SIZE | Indicates the total size of the table (of type int) |
| currentSize | Indicates how many elements are currently stored in the table (of type int) |
| table | The table where the data is stored (an array of TableEntry pointers) |

**Public Member Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Description | Parameters | Return Type |
| (constructor) | Constructs the class and initializes the variables and table | None | None |
| (destructor) | Clears the table from memory and destroys the class instance | None | None |
| getTotalSize | Returns the current total capacity of the table | None | int |
| getCurrentSize | Returns the current number of elements in the table | None | int |
| getTable | Returns the pointer to the table | None | TableEntry\*\* |
| hash | Hashes the key using a generic hashing function given by template argument H | K key | int |
| insert | Inserts a node at a location in the table given by the key | K key, V val | void |
| remove | Removes an entry from the table by setting the entry’s flag (entry given by the entered key) | K key | void |
| retrieve | Searches for and retrieves a value from the table based on the given key | K key | TableEntry\* |

**Private Member Functions**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Description | Parameters | Return Type |
| resize | Resizes the table to a new size that is 3 times the last size; accepts the key and value for a new entry | K key, V val | void |
| insertPrivate | Helper function for resize function: inserts data properly into the new table | TableEntry\*\* tempTable, int newSize, K key, V val | void |

**Member Structs**

|  |
| --- |
| **Struct: TableEntry** |
| **Member Variables:** |
| K key  V data  bool deleted |

The TableEntry member struct is used to store the data required for an entry in the hash table. This includes the key-value pair for that entry and a flag indicating if the entry has been removed from the table

**TableEntry Member Variables**

|  |  |
| --- | --- |
| Variable | Description |
| key | The key associated with the data entry (of generic type K) |
| data | The data associated with the entry (of generic type V) |
| deleted | A flag indicating if the entry has been removed from the table (of type bool) |

**Implementation Details:**

The retrieve and getTable functions both return pointer data types. Both functions are capable of returning null pointers. It is up to the programmer to handle null pointers returned from these functions.

The generic hash function is passed to the HashTable object as a template argument. To be able to pass a hashing function to the table, it must be encapsulated in a struct. This allows it to be seen as a data type and can therefore be passed in as a template argument. It is recommended that the hash function used be the Standard Library hash function (std::hash), although it is possible for the programmer to create their own hashing algorithm and pass it to the class.

**Usage Examples:**

Below are the usage examples for the public member functions of the HashTable class. For each example, make sure you are using the proper directory where the HashTable.h file is stored.

**Getter for Table Size:**



**Getter for Current Size:**



**Getter for Table:**



**Insertion Function:**



**Retrieval Function:**



**Removal Function:**



**Hash Function:**

