Hash Table Documentation

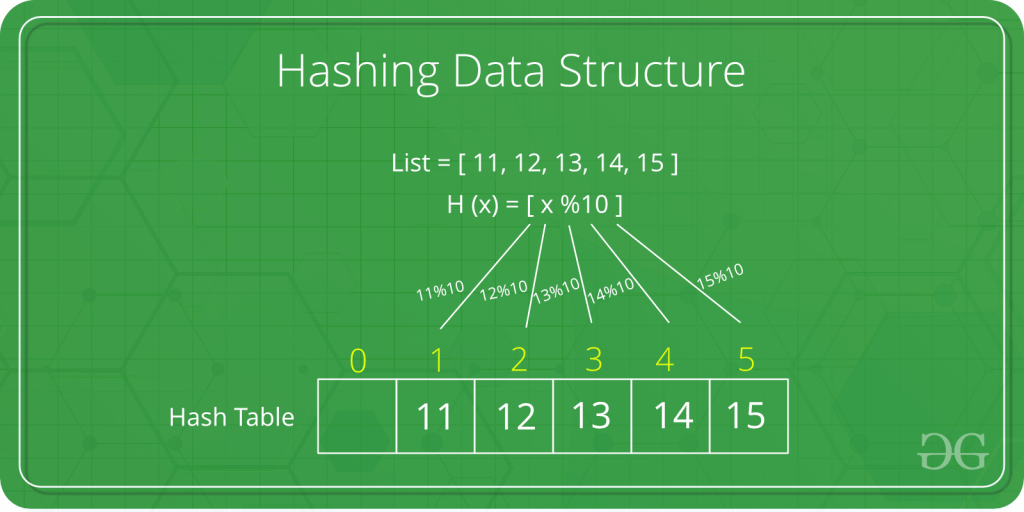
Samael Newgate

University of Advancing Technology

Documentation

**Hash Table:**

Hash table also known as hash map is a data structure that uses a function called hashing. This hashing function will map any given value with a key in order to have faster accessing of the element. The hash table will use the hashing function to create an index into an array of slots. The goal is for the hashing function to assign each key a unique slot. But because of the hash table structure, sometimes this slot will have multiple keys within it, this is called collision. There are multiple ways to bypass collisions, they are separate chaining and open addressing. In separate chaining you use linked lists inside the slots. This will make the table size issue nonexistent but will defeat the main purpose of the hash table, which is fast accessing. Open addressing is what I went for in my code. To simplify this method, all it does is find an empty slot within the table.



**Hash Table Class Structure:**

template<typename K, typename V>

class HashTable

{

private:

struct element

{

K key;

V value;

};

element\*\* hash\_array;

int size;

int current\_size;

void resize(K key, V value)

{

int old\_size = size;

size += 5;

current\_size = 0;

element\*\* new\_table = new element\*[size];

element\*\* old\_array = hash\_array;

for (int i = 0; i < size; i++)

{

new\_table[i] = nullptr;

}

for(int i=0; i<old\_size; i++)

{

if(hash\_array[i] != nullptr)

{

insert\_private(new\_table, hash\_array[i]->key, hash\_array[i]->value);

delete hash\_array[i];

}

}

insert\_private(new\_table, key, value);

hash\_array = new\_table;

delete[] old\_array;

}

void insert\_private(element\*\* new\_table, K key, V new\_value)

{

int start\_index = -1;

int index = hash(key);

while (index != start\_index && new\_table[index] != nullptr && new\_table[index]->key != key)

{

if (start\_index == -1)

{

start\_index = index;

}

index = (index + 1) % size;

}

if (new\_table[index] == nullptr)

{

new\_table[index] = new element;

new\_table[index]->key = key;

new\_table[index]->value = new\_value;

current\_size++;

}

else if (new\_table[index]->key == key)

{

new\_table[index]->value = new\_value;

}

else

{

return;

}

}

public:

HashTable()

{

size = 10;

current\_size = 0;

hash\_array = new element\*[size];

for (int i = 0; i < 10; i++)

{

hash\_array[i] = nullptr;

}

}

~HashTable() {}

int hash(K key)

{

int hash\_value = std::hash<K>()(key);

if (hash\_value < 0)

{

hash\_value \*= -1;

}

return hash\_value % size;

}

void insert(K key, V new\_value)

{

int start\_index = -1;

int index = hash(key);

while (index != start\_index && hash\_array[index] != nullptr && hash\_array[index]->key != key)

{

if (start\_index == -1)

{

start\_index = index;

}

index = (index + 1) % size;

}

if(index == start\_index && current\_size == size)

{

resize(key, new\_value);

}

else if (hash\_array[index] == nullptr)

{

hash\_array[index] = new element;

hash\_array[index]->key = key;

hash\_array[index]->value = new\_value;

current\_size++;

}

else if (hash\_array[index]->key == key)

{

hash\_array[index]->value = new\_value;

}

else

{

return;

}

}

V retrieve(K key)

{

int start\_index = -1;

int index = hash(key);

while (index != start\_index && hash\_array[index] != nullptr && hash\_array[index]->key != key)

{

if (start\_index == -1)

{

start\_index = index;

}

index = (index + 1) % size;

}

if (hash\_array[index] == nullptr||hash\_array[index]->key != key)

{

return NULL;

}

return hash\_array[index]->value;

}

int table\_size()

{

return size;

}

int get\_current\_size()

{

return current\_size;

}

};

**Member Functions:**

|  |  |
| --- | --- |
| Struct element | Struct use for our key and value elements |
| Void resize(K key, V value) | Resizes the overall table size if the hash table gets full. |
| Void insert\_private(element\*\* new\_table, K key, V value) | Second insert function used when the resize function is called. It will insert the values from the old table to the new table. |
| HashTable() | Hash table constructor. |
| ~HashTable() | Hash table deconstructor. |
| Int hash(K key) | Hash function used to hash the key inserted. |
| Void insert(K key, V new\_value) | Main insert function inserts a key and value. |
| V retrieve(K key) | Retrieve function used to retrieve a specific key within the hash table. |
| Int table\_size() | Function used to find the overall table size. |
| Int get\_current\_size() | Function used to get the used table size. |

**Usage:**

Hash tables are great for when you need to create an index. This index can be for anything. An index of files, phone numbers, patches, anything. Another usage example would apply to a college. One way to find students would be to take advantage of the usage of pairs within the hash table data structure. The college could use the students name and ID within a hash table to make the searching for specific students extremely fast. Because the hashing function takes an entry and reduces it to a hash key(number), this makes accessing that element faster.