**HASH TABLES**

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 article, 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, search, and delete operations.
* A data structure to account for a collision of keys.

Choosing a Hash Function

The first step is to choose a reasonably good hash function that has a low chance of collision. However, for the purposes of this tutorial, a poor hash function will be applied to better illustrate hash collisions. This limited example will also only utilize strings (or character arrays in C).

#define CAPACITY 50000 // Size of the HashTable.

unsigned long hash\_function(char\* str)

{

unsigned long i = 0;

for (int j = 0; str[j]; j++)

i += str[j];

return i % CAPACITY;

}

// Tutorial //

How To Implement a Sample Hash Table in C/C++

Published on August 3, 2022 · Updated on January 13, 2023

C++

C Programming

Default avatar

By Vijaykrishna Ram

How To Implement a Sample Hash Table in C/C++

Introduction

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 article, 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, search, and delete operations.

A data structure to account for a collision of keys.

**Choosing a Hash Function**

The first step is to choose a reasonably good hash function that has a low chance of collision. However, for the purposes of this tutorial, a poor hash function will be applied to better illustratehash collisions. This limited example will also only utilize strings (or character arrays in C).

#define CAPACITY 50000 // Size of the HashTable.

unsigned long hash\_function(char\* str)

{

unsigned long i = 0;

for (int j = 0; str[j]; j++)

i += str[j];

return i % CAPACITY;

}

Run this code and test different strings for potential collisions. For example, the strings Hel and Cau will collide since they have the same ASCII value.

This code must return a number within the bounds of the capacity of the hash table. Otherwise, it may access an unbound memory location, leading to an error.

**Defining the Hash Table Data Structures**

A hash table is an array of items, which are { key: value } pairs.

First, define the item structure:

// Defines the HashTable item.

typedef struct Ht\_item

{

char\* key;

char\* value;} Ht\_item;

Now, the hash table has an array of pointers that point to Ht\_item, so it is a double-pointer.

// Defines the HashTable.

typedef struct HashTable

{

// Contains an array of pointers to items.

Ht\_item\*\* items;

int size;

int count;

} HashTable;

Your hash table will need to return the number of elements in the hash table using count and size of the hash table using size.

**Creating the Hash Table and Hash Table Items**

Next, create functions for allocating memory and creating items.

Create items by allocating memory for a key and value, and return a pointer to the item:

Ht\_item\* create\_item(char\* key, char\* value)

{

// Creates a pointer to a new HashTable item.

Ht\_item\* item = (Ht\_item\*) malloc(sizeof(Ht\_item));

item->key = (char\*) malloc(strlen(key) + 1);

item->value = (char\*) malloc(strlen(value) + 1);

strcpy(item->key, key);

strcpy(item->value, value);

return item;

}

***Create the table by allocating memory and setting size, count, and items*:**

HashTable\* create\_table(int size)

{

// Creates a new HashTable.

HashTable\* table = (HashTable\*) malloc(sizeof(HashTable));

table->size = size;

table->count = 0;

table->items = (Ht\_item\*\*) calloc(table->size, sizeof(Ht\_item\*));

for (int i = 0; i < table->size; i++)

table->items[i] = NULL;

return table;

}

The preceding example allocates memory for the wrapper structure HashTable and sets all the items to NULL.

Freeing up memory is a C/C++ best practice. Free up memory that you’ve allocated on the heap with *malloc() and calloc().*

***Write functions that free up a table item and the whole table.***

void free\_item(Ht\_item\* item)

{

// Frees an item.

free(item->key);

free(item->value);

free(item);

}

void free\_table(HashTable\* table)

{

// Frees the table.

for (int i = 0; i < table->size; i++)

{

Ht\_item\* item = table->items[i];

if (item != NULL)

free\_item(item);

}

free(table->items);

free(table);}

***Add a print\_table() to display the index, key, and value for each item:***

void print\_table(HashTable\* table)

{

printf(”\nHash Table\n-------------------\n”);

for (int i = 0; i < table->size; i++)

{

if (table->items[i])

{

printf(”Index:%d, Key:%s, Value:%s\n”, i, table->items[i] -> key, table->items[i]->value);

}

}

printf(”-------------------\n\n”);

}

This concludes the basic functionality of your custom hash table. You will now write insert, search, and delete functions.

**Inserting into the Hash Table**

Create a function, ht\_insert(), that performs insertions.

The function takes a HashTable pointer, a key, and a value as parameters:

void ht\_insert(HashTable\* table, char\* key, char\* value)

{

...

}

Now, there are certain steps involved in the ht\_insert() function.

* Create the item based on the { key: value } pair.
* Compute the index based on the hash function.
* Check if the index is already occupied or not, by comparing the key.

*If it is not occupied, you can directly insert it into index. Otherwise, it is a collision, and you will need to handle it.*

This tutorial will address handling collisions after the initial model has been created.

**First, create the item:**

create\_item(key, value)

**Then, compute the index:**

int index = hash\_function(key);

**When inserting the key for the first time, the item must be a NULL:**

// Creates the item.

Ht\_item\* item = create\_item(key, value);

// Computes the index.

int index = hash\_function(key);

Ht\_item\* current\_item = table->items[index];

if (current\_item == NULL)

{

// Key does not exist.

if (table->count == table->size)

{

// HashTable is full.

printf(”Insert Error: Hash Table is full\n”);

free\_item(item);

return;

}

// Insert directly.

table->items[index] = item;

table->count++;

}

Consider the scenario where the { key: value } pair already exists because the same item has been inserted into the hash table. To address this, the code must update the item value to the new one:

if (current\_item == NULL)

{

...

}

else {

// Scenario 1: Update the value.

if (strcmp(current\_item->key, key) == 0)

{

strcpy(table->items[index] -> value, value);

return;

}

}

Consider the scenario where a collision has to be handled. To address this, a placeholder has been added:

void handle\_collision(HashTable\* table, Ht\_item\* item)

{

}

void ht\_insert(HashTable\* table, char\* key, char\* value)

{

...

if (current\_item == NULL)

{

...

}

else {

// Scenario 1: Update the value.

if (strcmp(current\_item->key, key) == 0)

{

...

}

else {

// Scenario 2: Handle the collision.

handle\_collision(table, item);

return;

}

}

}

Now, your ht\_insert() function is complete.

**Searching for Items in the Hash Table**

Create a function, ht\_search(), that checks if the key exists, and returns the corresponding value if it does.

The function takes a HashTable pointer and a key as parameters:

char\* ht\_search(HashTable\* table, char\* key)

{

...

}

**Search for an item with the key in the HashTable. If the item cannot be found in the HashTable, NULL is returned.**

char\* ht\_search(HashTable\* table, char\* key)

{

// Searches for the key in the HashTable.

// Returns NULL if it doesn’t exist.

int index = hash\_function(key);

Ht\_item\* item = table->items[index];

// Provide only non-NULL values.

if (item != NULL)

{

if (strcmp(item->key, key) == 0)

return item->value;

}

return NULL;

}

***Add a print\_search() to display the item that matches the key:***

void print\_search(HashTable\* table, char\* key)

{

char\* val;

if ((val = ht\_search(table, key)) == NULL)

{

printf(”Key:%s does not exist\n”, key);

return;

}

else {

printf(”Key:%s, Value:%s\n”, key, val);

}

}

Now, your ht\_search() function is complete.

**Handling Collisions**

There are different ways to resolve a collision. This tutorial will rely upon a method called Separate Chaining, which aims to create independent chains for all items that have the same hash index. The implementation in this tutorial will create these chains using linked lists.

Whenever there is a collision, additional items that collide on the same index are added to an overflow bucket list. Thus, you will not have to delete any existing records on the hash table.

Due to linked lists having O(n) time complexity for insertion, searching, and deletion, in case of a collision, you will have a worst-case access time of O(n) as well. The advantage of this method is that it is a good choice if your hash table has a low capacity.

**Implement the overflow bucket list:**

// Defines the LinkedList.

typedef struct LinkedList {

Ht\_item\* item;

struct LinkedList\* next;

} LinkedList;;

LinkedList\* allocate\_list()

{

// Allocates memory for a LinkedList pointer.

LinkedList\* list = (LinkedList\*) malloc(sizeof(LinkedList));

return list;

}

LinkedList\* linkedlist\_insert(LinkedList\* list, Ht\_item\* item)

{

// Inserts the item onto the LinkedList.

if (!list)

{

LinkedList\* head = allocate\_list();

head->item = item;

head->next = NULL;

list = head;

return list;

}

else if (list->next == NULL)

{

LinkedList\* node = allocate\_list();

node->item = item;

node->next = NULL;

list->next = node;

return list;

}

LinkedList\* temp = list;

while (temp->next->next)

{

temp = temp->next;

}

LinkedList\* node = allocate\_list();

node->item = item;

node->next = NULL;

temp->next = node;

return list;

}

Ht\_item\* linkedlist\_remove(LinkedList\* list)

{

// Removes the head from the LinkedList.

// Returns the item of the popped element.

if (!list)

return NULL;

if (!list->next)

return NULL;

LinkedList\* node = list->next;

LinkedList\* temp = list;

temp->next = NULL;

list = node;

Ht\_item\* it = NULL;

memcpy(temp->item, it, sizeof(Ht\_item));

free(temp->item->key);

free(temp->item->value);

free(temp->item);

free(temp);

return it;

}

void free\_linkedlist(LinkedList\* list)

{

LinkedList\* temp = list;

while (list)

{

temp = list;

list = list->next;

free(temp->item->key);

free(temp->item->value);

free(temp->item);

free(temp);

}

}

// Tutorial //

How To Implement a Sample Hash Table in C/C++

Published on August 3, 2022 · Updated on January 13, 2023

C++

C Programming

Default avatar

By Vijaykrishna Ram

How To Implement a Sample Hash Table in C/C++

Introduction

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 article, 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, search, and delete operations.

A data structure to account for a collision of keys.

Choosing a Hash Function

The first step is to choose a reasonably good hash function that has a low chance of collision. However, for the purposes of this tutorial, a poor hash function will be applied to better illustrate hash collisions. This limited example will also only utilize strings (or character arrays in C).

HashTable.cpp

#define CAPACITY 50000 // Size of the HashTable.

unsigned long hash\_function(char\* str)

{

unsigned long i = 0;

for (int j = 0; str[j]; j++)

i += str[j];

return i % CAPACITY;

}

Run this code and test different strings for potential collisions. For example, the strings Hel and Cau will collide since they have the same ASCII value.

This code must return a number within the bounds of the capacity of the hash table. Otherwise, it may access an unbound memory location, leading to an error.

Defining the Hash Table Data Structures

A hash table is an array of items, which are { key: value } pairs.

First, define the item structure:

HashTable.cpp

// Defines the HashTable item.

typedef struct Ht\_item

{

char\* key;

char\* value;

} Ht\_item;

Now, the hash table has an array of pointers that point to Ht\_item, so it is a double-pointer.

HashTable.cpp

// Defines the HashTable.

typedef struct HashTable

{

// Contains an array of pointers to items.

Ht\_item\*\* items;

int size;

int count;

} HashTable;

Your hash table will need to return the number of elements in the hash table using count and size of the hash table using size.

Creating the Hash Table and Hash Table Items

Next, create functions for allocating memory and creating items.

Create items by allocating memory for a key and value, and return a pointer to the item:

HashTable.cpp

Ht\_item\* create\_item(char\* key, char\* value)

{

// Creates a pointer to a new HashTable item.

Ht\_item\* item = (Ht\_item\*) malloc(sizeof(Ht\_item));

item->key = (char\*) malloc(strlen(key) + 1);

item->value = (char\*) malloc(strlen(value) + 1);

strcpy(item->key, key);

strcpy(item->value, value);

return item;

}

Create the table by allocating memory and setting size, count, and items:

HashTable.cpp

HashTable\* create\_table(int size)

{

// Creates a new HashTable.

HashTable\* table = (HashTable\*) malloc(sizeof(HashTable));

table->size = size;

table->count = 0;

table->items = (Ht\_item\*\*) calloc(table->size, sizeof(Ht\_item\*));

for (int i = 0; i < table->size; i++)

table->items[i] = NULL;

return table;

}

The preceding example allocates memory for the wrapper structure HashTable and sets all the items to NULL.

Freeing up memory is a C/C++ best practice. Free up memory that you’ve allocated on the heap with malloc() and calloc().

Write functions that free up a table item and the whole table.

HashTable.cpp

void free\_item(Ht\_item\* item)

{

// Frees an item.

free(item->key);

free(item->value);

free(item);

}

void free\_table(HashTable\* table)

{

// Frees the table.

for (int i = 0; i < table->size; i++)

{

Ht\_item\* item = table->items[i];

if (item != NULL)

free\_item(item);

}

free(table->items);

free(table);

}

Add a print\_table() to display the index, key, and value for each item:

HashTable.cpp

void print\_table(HashTable\* table)

{

printf(”\nHash Table\n-------------------\n”);

for (int i = 0; i < table->size; i++)

{

if (table->items[i])

{

printf(”Index:%d, Key:%s, Value:%s\n”, i, table->items[i] -> key, table->items[i]->value);

}

}

printf(”-------------------\n\n”);

}

This concludes the basic functionality of your custom hash table. You will now write insert, search, and delete functions.

Inserting into the Hash Table

Create a function, ht\_insert(), that performs insertions.

The function takes a HashTable pointer, a key, and a value as parameters:

void ht\_insert(HashTable\* table, char\* key, char\* value)

{

...

}

Now, there are certain steps involved in the ht\_insert() function.

Create the item based on the { key: value } pair.

Compute the index based on the hash function.

Check if the index is already occupied or not, by comparing the key.

If it is not occupied, you can directly insert it into index.

Otherwise, it is a collision, and you will need to handle it.

This tutorial will address handling collisions after the initial model has been created.

First, create the item:

create\_item(key, value)

Then, compute the index:

int index = hash\_function(key);

When inserting the key for the first time, the item must be a NULL:

HashTable.cpp

// Creates the item.

Ht\_item\* item = create\_item(key, value);

// Computes the index.

int index = hash\_function(key);

Ht\_item\* current\_item = table->items[index];

if (current\_item == NULL)

{

// Key does not exist.

if (table->count == table->size)

{

// HashTable is full.

printf(”Insert Error: Hash Table is full\n”);

free\_item(item);

return;

}

// Insert directly.

table->items[index] = item;

table->count++;

}

Consider the scenario where the { key: value } pair already exists because the same item has been inserted into the hash table. To address this, the code must update the item value to the new one:

HashTable.cpp

if (current\_item == NULL)

{

...

}

else {

// Scenario 1: Update the value.

if (strcmp(current\_item->key, key) == 0)

{

strcpy(table->items[index] -> value, value);

return;

}

}

Consider the scenario where a collision has to be handled. To address this, a placeholder has been added:

HashTable.cpp

void handle\_collision(HashTable\* table, Ht\_item\* item)

{

}

void ht\_insert(HashTable\* table, char\* key, char\* value)

{

...

if (current\_item == NULL)

{

...

}

else {

// Scenario 1: Update the value.

if (strcmp(current\_item->key, key) == 0)

{

...

}

else {

// Scenario 2: Handle the collision.

handle\_collision(table, item);

return;

}

}

}

Now, your ht\_insert() function is complete.

Searching for Items in the Hash Table

Create a function, ht\_search(), that checks if the key exists, and returns the corresponding value if it does.

The function takes a HashTable pointer and a key as parameters:

char\* ht\_search(HashTable\* table, char\* key)

{

...

}

Search for an item with the key in the HashTable. If the item cannot be found in the HashTable, NULL is returned.

HashTable.cpp

char\* ht\_search(HashTable\* table, char\* key)

{

// Searches for the key in the HashTable.

// Returns NULL if it doesn’t exist.

int index = hash\_function(key);

Ht\_item\* item = table->items[index];

// Provide only non-NULL values.

if (item != NULL)

{

if (strcmp(item->key, key) == 0)

return item->value;

}

return NULL;

}

Add a print\_search() to display the item that matches the key:

HashTable.cpp

void print\_search(HashTable\* table, char\* key)

{

char\* val;

if ((val = ht\_search(table, key)) == NULL)

{

printf(”Key:%s does not exist\n”, key);

return;

}

else {

printf(”Key:%s, Value:%s\n”, key, val);

}

}

Now, your ht\_search() function is complete.

Handling Collisions

There are different ways to resolve a collision. This tutorial will rely upon a method called Separate Chaining, which aims to create independent chains for all items that have the same hash index. The implementation in this tutorial will create these chains using linked lists.

Whenever there is a collision, additional items that collide on the same index are added to an overflow bucket list. Thus, you will not have to delete any existing records on the hash table.

Due to linked lists having O(n) time complexity for insertion, searching, and deletion, in case of a collision, you will have a worst-case access time of O(n) as well. The advantage of this method is that it is a good choice if your hash table has a low capacity.

**Implement the overflow bucket list:**

// Defines the LinkedList.

typedef struct LinkedList {

Ht\_item\* item;

struct LinkedList\* next;

} LinkedList;;

LinkedList\* allocate\_list()

{

// Allocates memory for a LinkedList pointer.

LinkedList\* list = (LinkedList\*) malloc(sizeof(LinkedList));

return list;

}

LinkedList\* linkedlist\_insert(LinkedList\* list, Ht\_item\* item)

{

// Inserts the item onto the LinkedList.

if (!list)

{

LinkedList\* head = allocate\_list();

head->item = item;

head->next = NULL;

list = head;

return list;

}

else if (list->next == NULL)

{

LinkedList\* node = allocate\_list();

node->item = item;

node->next = NULL;

list->next = node;

return list;

}

LinkedList\* temp = list;

while (temp->next->next)

{

temp = temp->next;

}

LinkedList\* node = allocate\_list();

node->item = item;

node->next = NULL;

temp->next = node;

return list;

}

Ht\_item\* linkedlist\_remove(LinkedList\* list)

{

// Removes the head from the LinkedList.

// Returns the item of the popped element.

if (!list)

return NULL;

if (!list->next)

return NULL;

LinkedList\* node = list->next;

LinkedList\* temp = list;

temp->next = NULL;

list = node;

Ht\_item\* it = NULL;

memcpy(temp->item, it, sizeof(Ht\_item));

free(temp->item->key);

free(temp->item->value);

free(temp->item);

free(temp);

return it;

}

void free\_linkedlist(LinkedList\* list)

{

LinkedList\* temp = list;

while (list)

{

temp = list;

list = list->next;

free(temp->item->key);

free(temp->item->value);

free(temp->item);

free(temp);

}

}

Now, add these overflow bucket lists to your HashTable. Every item will have a chain, so the whole table is an array of LinkedList poi

typedef struct HashTable HashTable;

// Defines the HashTable.

struct HashTable

{

// Contains an array of pointers to items.

Ht\_item\*\* items;

LinkedList\*\* overflow\_buckets;

int size;

int count;

};

Now that overflow\_buckets have been defined, add functions to create and delete them. You will also need to account for them in the create\_table() and free\_table() functions.

HashTable.cpp

LinkedList\*\* create\_overflow\_buckets(HashTable\* table)

{

// Create the overflow buckets; an array of LinkedLists.

LinkedList\*\* buckets = (LinkedList\*\*) calloc(table->size, sizeof(LinkedList\*));

for (int i = 0; i < table->size; i++)

buckets[i] = NULL;

return buckets;

}

void free\_overflow\_buckets(HashTable\* table)

{

// Free all the overflow bucket lists.

LinkedList\*\* buckets = table->overflow\_buckets;

for (int i = 0; i < table->size; i++)

free\_linkedlist(buckets[i]);

free(buckets);

}

HashTable\* create\_table(int size)

{

// Creates a new HashTable.

HashTable\* table = (HashTable\*) malloc(sizeof(HashTable));

table->size = size;

table->count = 0;

table->items = (Ht\_item\*\*) calloc(table->size, sizeof(Ht\_item\*));

for (int i = 0; i < table->size; i++)

table->items[i] = NULL;

table->overflow\_buckets = create\_overflow\_buckets(table);

return table;

}

void free\_table(HashTable\* table)

{

// Frees the table.

for (int i = 0; i < table->size; i++)

{

Ht\_item\* item = table->items[i];

if (item != NULL)

free\_item(item);

}

// Free the overflow bucket lists and its items.

free\_overflow\_buckets(table);

free(table->items);

free(table);

}

If the overflow bucket list for the item does not exist, create a list and add the item to it.

**Update handle\_collision() for insertions**:

void handle\_collision(HashTable\* table, unsigned long index, Ht\_item\* item)

{

LinkedList\* head = table->overflow\_buckets[index];

if (head == NULL)

{

// Creates the list.

head = allocate\_list();

head->item = item;

table->overflow\_buckets[index] = head;

return;

}

else {

// Insert to the list.

table->overflow\_buckets[index] = linkedlist\_insert(head, item);

return;

}

}

And the call:

HashTable.cpp

void ht\_insert(HashTable\* table, char\* key, char\* value)

{

...

if (current\_item == NULL)

{

...

}

else {

// Scenario 1: Update the value.

if (strcmp(current\_item->key, key) == 0)

{

...

}

else {

// Scenario 2: Handle the collision.

handle\_collision(table, index, item);

return;

}

}

}

Now, update the search method to use overflow bucke

char\* ht\_search(HashTable\* table, char\* key)

{

// Searches for the key in the HashTable.

// Returns NULL if it doesn’t exist.

int index = hash\_function(key);

Ht\_item\* item = table->items[index];

LinkedList\* head = table->overflow\_buckets[index];

// Provide only non-NULL values.

if (item != NULL)

{

if (strcmp(item->key, key) == 0)

return item->value;

if (head == NULL)

return NULL;

item = head->item;

head = head->next;

}

return NULL;

}

Finally, collisions are now handled in insert() and search()!

Deleting from the Hash Table

Let’s now finally look at the delete function:

void ht\_delete(HashTable\* table, char\* key)

{

...

}

Again, the method is similar to insertion.

Compute the hash index and get the item.

***If it is NULL, don’t do anything.***

Otherwise, after comparing keys, if there is no collision chain for that index, remove the item from the table.

***If a collision chain exists, remove that element and shift the links accordingly***.

void ht\_delete(HashTable\* table, char\* key)

{

// Deletes an item from the table.

int index = hash\_function(key);

Ht\_item\* item = table->items[index];

LinkedList\* head = table->overflow\_buckets[index];

if (item == NULL)

{

// Does not exist.

return;

}

else {

if (head == NULL && strcmp(item->key, key) == 0)

{

// Collision chain does not exist.

// Remove the item.

// Set table index to NULL.

table->items[index] = NULL;

free\_item(item);

table->count--;

return;

}

else if (head != NULL)

{

// Collision chain exists.

if (strcmp(item->key, key) == 0)

{

// Remove this item.

// Set the head of the list as the new item.

free\_item(item);

LinkedList\* node = head;

head = head->next;

node->next = NULL;

table->items[index] = create\_item(node->item->key, node->item->value);

free\_linkedlist(node);

table->overflow\_buckets[index] = head;

return;

}

LinkedList\* curr = head;

LinkedList\* prev = NULL;

while (curr)

{

if (strcmp(curr->item->key, key) == 0)

{

if (prev == NULL)

{

// First element of the chain.

// Remove the chain.

free\_linkedlist(head);

table->overflow\_buckets[index] = NULL;

return;

}

else

{

// This is somewhere in the chain.

prev->next = curr->next;

curr->next = NULL;

free\_linkedlist(curr);

table->overflow\_buckets[index] = head;

return;

}

}

curr = curr->next;

prev = curr;

}

}

}

}

**Complete implementation of the explanation**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#define CAPACITY 50000 // Size of the HashTable.**

**unsigned long hash\_function(char \*str)**

**{**

**unsigned long i = 0;**

**for (int j = 0; str[j]; j++)**

**i += str[j];**

**return i % CAPACITY;**

**}**

**// Defines the HashTable item.**

**typedef struct Ht\_item**

**{**

**char \*key;**

**char \*value;**

**} Ht\_item;**

**// Defines the LinkedList.**

**typedef struct LinkedList**

**{**

**Ht\_item \*item;**

**LinkedList \*next;**

**} LinkedList;**

**// Defines the HashTable.**

**typedef struct HashTable**

**{**

**// Contains an array of pointers to items.**

**Ht\_item \*\*items;**

**LinkedList \*\*overflow\_buckets;**

**int size;**

**int count;**

**} HashTable;**

**LinkedList \*allocate\_list()**

**{**

**// Allocates memory for a LinkedList pointer.**

**LinkedList \*list = (LinkedList \*)malloc(sizeof(LinkedList));**

**return list;**

**}**

**LinkedList \*linkedlist\_insert(LinkedList \*list, Ht\_item \*item)**

**{**

**// Inserts the item onto the LinkedList.**

**if (!list)**

**{**

**LinkedList \*head = allocate\_list();**

**head->item = item;**

**head->next = NULL;**

**list = head;**

**return list;**

**}**

**else if (list->next == NULL)**

**{**

**LinkedList \*node = allocate\_list();**

**node->item = item;**

**node->next = NULL;**

**list->next = node;**

**return list;**

**}**

**LinkedList \*temp = list;**

**while (temp->next->next)**

**{**

**temp = temp->next;**

**}**

**LinkedList \*node = allocate\_list();**

**node->item = item;**

**node->next = NULL;**

**temp->next = node;**

**return list;**

**}**

**Ht\_item \*linkedlist\_remove(LinkedList \*list)**

**{**

**// Removes the head from the LinkedList.**

**// Returns the item of the popped element.**

**if (!list)**

**return NULL;**

**if (!list->next)**

**return NULL;**

**LinkedList \*node = list->next;**

**LinkedList \*temp = list;**

**temp->next = NULL;**

**list = node;**

**Ht\_item \*it = NULL;**

**memcpy(temp->item, it, sizeof(Ht\_item));**

**free(temp->item->key);**

**free(temp->item->value);**

**free(temp->item);**

**free(temp);**

**return it;**

**}**

**void free\_linkedlist(LinkedList \*list)**

**{**

**LinkedList \*temp = list;**

**while (list)**

**{**

**temp = list;**

**list = list->next;**

**free(temp->item->key);**

**free(temp->item->value);**

**free(temp->item);**

**free(temp);**

**}**

**}**

**LinkedList \*\*create\_overflow\_buckets(HashTable \*table)**

**{**

**// Create the overflow buckets; an array of LinkedLists.**

**LinkedList \*\*buckets = (LinkedList \*\*)calloc(table->size, sizeof(LinkedList \*));**

**for (int i = 0; i < table->size; i++)**

**buckets[i] = NULL;**

**return buckets;**

**}**

**void free\_overflow\_buckets(HashTable \*table)**

**{**

**// Free all the overflow bucket lists.**

**LinkedList \*\*buckets = table->overflow\_buckets;**

**for (int i = 0; i < table->size; i++)**

**free\_linkedlist(buckets[i]);**

**free(buckets);**

**}**

**Ht\_item \*create\_item(char \*key, char \*value)**

**{**

**// Creates a pointer to a new HashTable item.**

**Ht\_item \*item = (Ht\_item \*)malloc(sizeof(Ht\_item));**

**item->key = (char \*)malloc(strlen(key) + 1);**

**item->value = (char \*)malloc(strlen(value) + 1);**

**strcpy(item->key, key);**

**strcpy(item->value, value);**

**return item;**

**}**

**HashTable \*create\_table(int size)**

**{**

**// Creates a new HashTable.**

**HashTable \*table = (HashTable \*)malloc(sizeof(HashTable));**

**table->size = size;**

**table->count = 0;**

**table->items = (Ht\_item \*\*)calloc(table->size, sizeof(Ht\_item \*));**

**for (int i = 0; i < table->size; i++)**

**table->items[i] = NULL;**

**table->overflow\_buckets = create\_overflow\_buckets(table);**

**return table;**

**}**

**void free\_item(Ht\_item \*item)**

**{**

**// Frees an item.**

**free(item->key);**

**free(item->value);**

**free(item);**

**}**

**void free\_table(HashTable \*table)**

**{**

**// Frees the table.**

**for (int i = 0; i < table->size; i++)**

**{**

**Ht\_item \*item = table->items[i];**

**if (item != NULL)**

**free\_item(item);**

**}**

**// Free the overflow bucket lists and its items.**

**free\_overflow\_buckets(table);**

**free(table->items);**

**free(table);**

**}**

**void handle\_collision(HashTable \*table, unsigned long index, Ht\_item \*item)**

**{**

**LinkedList \*head = table->overflow\_buckets[index];**

**if (head == NULL)**

**{**

**// Creates the list.**

**head = allocate\_list();**

**head->item = item;**

**table->overflow\_buckets[index] = head;**

**return;**

**}**

**else**

**{**

**// Insert to the list.**

**table->overflow\_buckets[index] = linkedlist\_insert(head, item);**

**return;**

**}**

**}**

**void ht\_insert(HashTable \*table, char \*key, char \*value)**

**{**

**// Creates the item.**

**Ht\_item \*item = create\_item(key, value);**

**// Computes the index.**

**int index = hash\_function(key);**

**Ht\_item \*current\_item = table->items[index];**

**if (current\_item == NULL)**

**{**

**// Key does not exist.**

**if (table->count == table->size)**

**{**

**// HashTable is full.**

**printf(”Insert Error: Hash Table is full\n”);**

**free\_item(item);**

**return;**

**}**

**// Insert directly.**

**table->items[index] = item;**

**table->count++;**

**}**

**else**

**{**

**// Scenario 1: Update the value.**

**if (strcmp(current\_item->key, key) == 0)**

**{**

**strcpy(table->items[index]->value, value);**

**return;**

**}**

**else**

**{**

**// Scenario 2: Handle the collision.**

**handle\_collision(table, index, item);**

**return;**

**}**

**}**

**}**

**char \*ht\_search(HashTable \*table, char \*key)**

**{**

**// Searches for the key in the HashTable.**

**// Returns NULL if it doesn’t exist.**

**int index = hash\_function(key);**

**Ht\_item \*item = table->items[index];**

**LinkedList \*head = table->overflow\_buckets[index];**

**// Provide only non-NULL values.**

**if (item != NULL)**

**{**

**if (strcmp(item->key, key) == 0)**

**return item->value;**

**if (head == NULL)**

**return NULL;**

**item = head->item;**

**head = head->next;**

**}**

**return NULL;**

**}**

**void ht\_delete(HashTable \*table, char \*key)**

**{**

**// Deletes an item from the table.**

**int index = hash\_function(key);**

**Ht\_item \*item = table->items[index];**

**LinkedList \*head = table->overflow\_buckets[index];**

**if (item == NULL)**

**{**

**// Does not exist.**

**return;**

**}**

**else**

**{**

**if (head == NULL && strcmp(item->key, key) == 0)**

**{**

**// Collision chain does not exist.**

**// Remove the item.**

**// Set table index to NULL.**

**table->items[index] = NULL;**

**free\_item(item);**

**table->count--;**

**return;**

**}**

**else if (head != NULL)**

**{**

**// Collision chain exists.**

**if (strcmp(item->key, key) == 0)**

**{**

**// Remove this item.**

**// Set the head of the list as the new item.**

**free\_item(item);**

**LinkedList \*node = head;**

**head = head->next;**

**node->next = NULL;**

**table->items[index] = create\_item(node->item->key, node->item->value);**

**free\_linkedlist(node);**

**table->overflow\_buckets[index] = head;**

**return;**

**}**

**LinkedList \*curr = head;**

**LinkedList \*prev = NULL;**

**while (curr)**

**{**

**if (strcmp(curr->item->key, key) == 0)**

**{**

**if (prev == NULL)**

**{**

**// First element of the chain.**

**// Remove the chain.**

**free\_linkedlist(head);**

**table->overflow\_buckets[index] = NULL;**

**return;**

**}**

**else**

**{**

**// This is somewhere in the chain.**

**prev->next = curr->next;**

**curr->next = NULL;**

**free\_linkedlist(curr);**

**table->overflow\_buckets[index] = head;**

**return;**

**}**

**}**

**curr = curr->next;**

**prev = curr;**

**}**

**}**

**}**

**}**

**void print\_search(HashTable \*table, char \*key)**

**{**

**char \*val;**

**if ((val = ht\_search(table, key)) == NULL)**

**{**

**printf(”Key:%s does not exist\n”, key);**

**return;**

**}**

**else**

**{**

**printf(”Key:%s, Value:%s\n”, key, val);**

**}**

**}**

**void print\_table(HashTable \*table)**

**{**

**printf(”\nHash Table\n-------------------\n”);**

**for (int i = 0; i < table -> size; i++)**

**{**

**if (table -> items[i])**

**{**

**printf(”Index:%d, Key:%s, Value:%s\n”, i, table -> items[i] -> key, table -> items[i] -> value);**

**}**

**}**

**printf(”-------------------\n\n”);**

**}**

**int main()**

**{**

**HashTable \*ht = create\_table(CAPACITY);**

**ht\_insert(ht, (char \*)”1”, (char \*)”First address”);**

**ht\_insert(ht, (char \*)”2”, (char \*)”Second address”);**

**ht\_insert(ht, (char \*)”Hel”, (char \*)”Third address”);**

**ht\_insert(ht, (char \*)”Cau”, (char \*)”Fourth address”);**

**print\_search(ht, (char \*)”1”);**

**print\_search(ht, (char \*)”2”);**

**print\_search(ht, (char \*)”3”);**

**print\_search(ht, (char \*)”Hel”);**

**print\_search(ht, (char \*)”Cau”); // Collision!**

**print\_table(ht);**

**ht\_delete(ht, (char \*)”1”);**

**ht\_delete(ht, (char \*)”Cau”);**

**print\_table(ht);**

**free\_table(ht);**

**return 0;**

**}**