IMPLEMENTATION OF HASH TABLE

A MTE Project report
Submitted in the partial fulfillment of the requirements
for the Award of Degree
OF

Master of Technology
In
INFORMATION TECHNOLOGY

Submitted by:
RAUNAK PODDAR
2K21/ISY/20
KARAN YADAV
2K21/ISY/11

Under the supervision of
Dr. Jasraj Meena

DEPARTMENT OF INFORMATION TECHNOLOGY
DELHI TECHNOLOGY UNIVERSITY

(Formerly Delhi College of Engineering)
Bawana Road, Delhi – 110042
NOVEMBER 2021

INTRODUCTION

A hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key. Such collisions are typically accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key–value pairs, at constant average cost per operation.

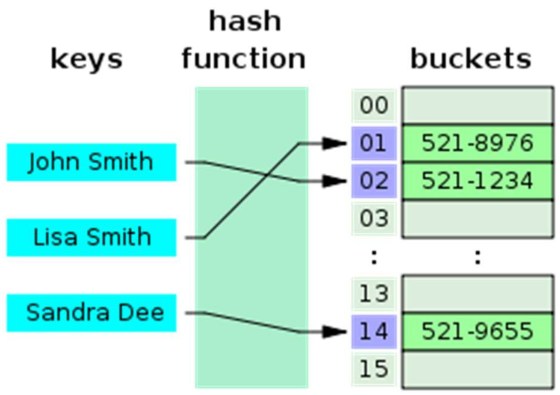
In many situations, hash tables turn out to be on average more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

IMPLEMENTATION OF PROJECT

A Hash Table in C/C++ (*Associative array*) is a data structure that maps keys to values. This uses a *hash function* to compute indexes for a key.

Based on the Hash Table index, we can store the value at the appropriate location.

If two different keys get the same index, we need to use other data structures (buckets) to account for these collisions.



The whole benefit of using a Hash Table is due to its very fast access time. While there can be a collision, if we choose a very good hash function, this chance is almost zero.

So, *on average*, the time complexity is a constant **O(1)** access time.

The **C++ STL** (Standard Template Library) has the std::unordered_map() data structure which implements all these hash table functions.

However, knowing how to construct a hash table from scratch is a crucial skill, and that is indeed what we aim to show in this project.

Let us understand more about the implementation details.

Any Hash Table implementation has the following three components:

- A good 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 collision of keys.

Choose a Hash Function

The first step is to choose a reasonably good hash function that has a low chance of collision

We will be working only with **strings** (or **character arrays** in C) in this project.

I'll be using a very simple hash function, that simply sums the ASCII value of the string. I am using this to show you how we will handle collision cases.

```
#define CAPACITY 50000 // Size of the Hash Table


unsigned long hash_function(char* str) {
   unsigned long i = 0;
   for (int j=0; str[j]; j++)
      i += str[j];
   return i % CAPACITY;
}
```

We can test this for different strings and check if they collide or not. For example, the strings "**Hel**" and "**Cau**" will collide, since they have the same ASCII value.

NOTE: We must return a number within the capacity of the hash table. Otherwise, we may access an unbound memory location, leading to an error.

Define the Hash Table data structures

A Hash table is an array of items, which themselves are a {**key**: **value**} pair.

Let's define our item structure first.

```
typedef struct Ht_item Ht_item;


// Define the Hash Table Item here
struct Ht_item {
   char* key;
```

```
    char* value;
};
```

Now, the Hash table has an array of pointers which themselves point to Ht_item, so it is a double-pointer.
Other than that, we will also keep track of the number of elements in the Hash table using count, and store the size of the table in size.

```
typedef struct HashTable HashTable;
 // Define the Hash Table here
struct HashTable {
    // Contains an array of pointers
    // to items
    Ht_item** items;
    int size;
    int count;
};
```

Create the Hash Table and its items

We need functions to create a new Hash table into memory and also create its items.

Let's create the item first. This is very simple since we only need to allocate memory for its key and value and return a pointer to the item.

```
Ht_item* create_item(char* key, char* value) {
    // Creates a pointer to a new hash table 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;
}
```

Now, let's write the code for creating the table. This allocates memory for the wrapper structure HashTable, and sets all it's items to NULL (Since they aren't used).

```
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;
}
```

Now, we are almost done with this part. As a C/C++ programmer, it is our responsibility to free up memory that you've allocated on the heap using malloc(), calloc().

So let's write functions which free up a table item, and the whole table too.

```
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);
}
```

We've now completed our groundwork to building a functional Hash Table. Let's now start writing our insert(), search() and delete() methods.

Insert into the Hash table

We will create a function ht_insert() that performs this task for us.
This takes in 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 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 **key**.
  - If it is not occupied. we can directly insert it into index
  - Otherwise, it is a collision, and we need to handle it

I will explain how we will handle collisions after we create the initial model.

The first step is simple. We directly call create_item(key, value).

```
int index = hash_function(key);
```

The second and third steps use hash_function(key) to get the index. If we are inserting the key for the first time, the item must be a NULL. Otherwise, the exact **key : value** pair already exists, or it is a collision.
In this case, we'll define another function handle_collision(), which as the name suggests, will handle that potential collision for us.

```
// Create the item

Ht_item* item = create_item(key, value);


// Compute 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) {
        // Hash Table Full
        printf("Insert Error: Hash Table is full\n");
        free_item(item);
        return;
    }

    // Insert directly
    table->items[index] = item;
    table->count++;
}
```

Let's consider the first scenario where the **key : value** pair already exists (i.e the same item was already inserted before). In this case, we must update the item value only to the new one.

```
if (current_item == NULL) {
    ....
    ....
}
else {
        // Scenario 1: We only need to update value
        if (strcmp(current_item->key, key) == 0) {
            strcpy(table->items[index]->value, value);
            return;
        }
    else {
        // Scenario 2: Collision
        // We will handle case this a bit later
        handle_collision(table, item);
        return;
    }
}
```

Okay, so our insert function (without collisions) now looks a bit like this:

```
void handle_collision(HashTable* table, Ht_item* item) {
}


void ht_insert(HashTable* table, char* key, char* value) {
    // Create the item
    Ht_item* item = create_item(key, value);

    Ht_item* current_item = table->items[index];

    if (current_item == NULL) {
        // Key does not exist.
        if (table->count == table->size) {
            // Hash Table Full
            printf("Insert Error: Hash Table is full\n");
            return;
        }

        // Insert directly
        table->items[index] = item;
        table->count++;
    }

    else {
            // Scenario 1: We only need to update value
            if (strcmp(current_item->key, key) == 0) {
                strcpy(table->items[index]->value, value);
                return;
            }

        else {
            // Scenario 2: Collision
            // We will handle case this a bit later
            handle_collision(table, item);
            return;
        }
    }
}
```

Search Items in the Hash Table

If we want to check if the insertion was done correctly, we also need to define a search function, that checks if the key exists or not, and returns the corresponding value if it does.

```
char* ht_search(HastTable* table, char* key);
```

The logic is very simple. It simply moves to non-**NULL** items and compares the key. Otherwise, we will return **NULL**

```
char* ht_search(HashTable* table, char* key) {
    // Searches the key in the hashtable
    // and returns NULL if it doesn't exist
    int index = hash_function(key);
    Ht_item* item = table->items[index];

    // Ensure that we move to a non NULL item
    if (item != NULL) {
        if (strcmp(item->key, key) == 0)
            return item->value;
    }
    return NULL;
}
```

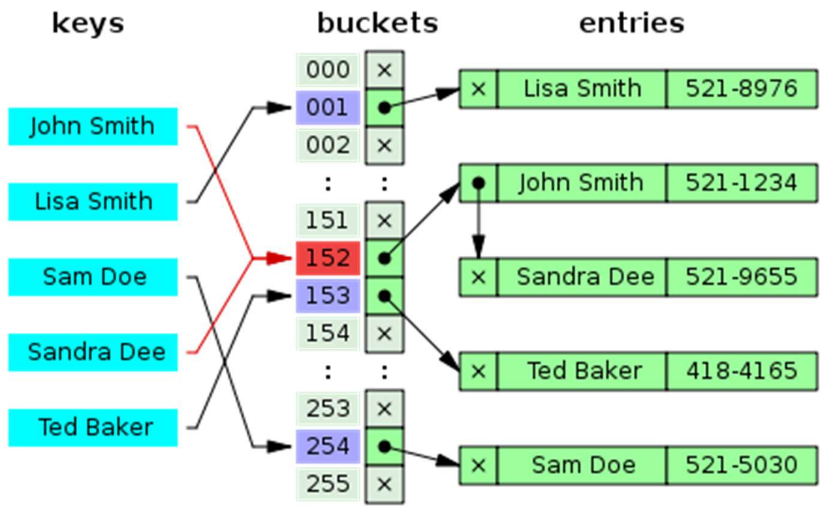
Handle Collisions

There are different ways through which a collision can be resolved. We will look at a method called Separate Chaining, which aims to create independent chains for all items that have the same hash index.

We will create these chains using Linked Lists.

Whenever there is a collision, we will add further items that collide on the same index on an *Overflow Bucket List*. Thus, we will not have to delete any existing records on our hash table.

Due to linked lists having **O(n)** time complexity for insertion, searching and deletion, in case of a collision, we 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.



With that covered, let's start implementing our good old Linked List!

```
typedef struct LinkedList LinkedList;


// Define the Linkedlist here
struct LinkedList {
  Ht_item* item;
  LinkedList* next;
```

```
};

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 Linked List
  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 linked list
    // and 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, we need to add these “Overflow Bucket” lists to our Hash Table. We want every item to have one such chain, so for the whole table, it is an array of **LinkedList** pointers.

```
typedef struct HashTable HashTable;

// Define the Hash Table here
struct HashTable {
    // Contains an array of pointers
    // to items
    Ht_item** items;
    LinkedList** overflow_buckets;
    int size;
    int count;
};
```

Now that we’ve defined our `overflow_buckets`, we will add functions to create and delete them. We also need to account for them in our old `create_table()` and `free_table()` functions.

```
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 linked linkedlist and it's items
    free_overflow_buckets(table);
    free(table->items);
    free(table);
}
```

Let’s now go to the `handle_collision()` function.

There are two scenarios here. If the overflow bucket list for the item does not exist, we need to create one such list and add the item to it.

Otherwise, we can simply insert the item to the list

```
void handle_collision(HashTable* table, unsigned long index, Ht_item* item) {
   LinkedList* head = table->overflow_buckets[index];

   if (head == NULL) {
      // We need to create 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;
   }}
```

So we are done with insertion, but now, we need to update our search function as well, since we may need to look at the overflow buckets as well.

```
char* ht_search(HashTable* table, char* key) {
   // Searches the key in the hashtable
   // and returns NULL if it doesn't exist
   int index = hash_function(key);
   Ht_item* item = table->items[index];
   LinkedList* head = table->overflow_buckets[index];

   // Ensure that we move to items which are not NULL
   while (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, we have accounted for collisions in insert() and search()!

Delete 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.

1. Compute the hash index and get the item
2. If it is **NULL**, we don’t need to do anything
3. Otherwise, after comparing keys, it there is no collision chain for that index, simply remove the item from the table
4. If a collision chain exists, we must 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
        return;
    }
    else {
        if (head == NULL && strcmp(item->key, key) == 0) {
            // No collision chain. Remove the item
            // and 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 and 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;
        }

     }
  }
}
```

CONCLUSION

In this project we have successfully implemented the Hash Table. We have defined the following functions and methodologies

- Choose a Hash Function
- Define the Hash Table data structures
- Create the Hash Table and its items
- Insert into the Hash Table
- Search items in the Hash Table
- Handle Collision
- Delete from Hash Table

We have tried to implement the Hash Table in such a way so that we can handle the collision in more effective way.

REFERENCES

[1] Teschner, Matthias, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H. Gross. "Optimized spatial hashing for collision detection of deformable objects." In *Vmv*, vol. 3, pp. 47-54. 2003.

[2] Nimbe, Peter, Samuel Ofori Frimpong, and Michael Opoku. "An efficient strategy for collision resolution in hash tables." *International Journal of Computer Applications* 99, no. 10 (2014): 35-41.

[3] https://www.geeksforgeeks.org/

[4] The Joys of Hashing: Hash Table Programming Using C By Thomas Mailund

[5] Let Us C: Authentic Guide To C Programming Language By Yashwant Karnetkar