

Data Structures and Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant
at The American University in Cairo (AUC)

spring 2020



sw.eng.MohamedAyman@gmail.com



[linkedin.com/in/cs-MohamedAyman](https://www.linkedin.com/in/cs-MohamedAyman)



github.com/cs-MohamedAyman




codeforces.com/profile/Mohamed_Ayman



Lecture 12

Hash Table

Static Length



Course Roadmap



Part 2: Non-Linear Data Structures

Lecture 8: Binary Tree

Lecture 9: Binary Search Tree

Lecture 10: Self Balancing Binary Search Tree

Lecture 11: Binary Heap Tree

Lecture 12: Hash Table

Lecture 13: Graph

Lecture 14: STL in C++ (Non-Linear Data Structures)

Lecture Agenda

We will discuss in this lecture
the following topics

- 1- Introduction to Hash Tables
 - 2- Collision Resolution
 - 3- Separate Chaining
 - 4- Open Addressing
 - 5- Double Hashing
 - 6- Time Complexity & Space Complexity
-



Let's
STARTUP

Lecture Agenda



Section 1: Introduction to Hash Tables

Section 2: Collision Resolution

Section 3: Separate Chaining

Section 4: Open Addressing

Section 5: Double Hashing

Section 6: Time Complexity & Space Complexity



Introduction to Hash Tables



- **Many applications require** a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list $\Theta(n)$ time in the worst case - in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $\Theta(1)$.
- **A hash table generalizes** the simpler notion of an ordinary array. Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in $\Theta(1)$ time. We can take advantage of direct addressing when we can afford to allocate an array that has one position for every possible key.
- **When the number of keys** actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is computed from the key.

Introduction to Hash Tables



- **Hashing is a technique** that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:
 - In universities, each student is assigned a unique roll number that can be used to retrieve information about them. In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.
 - In both these examples the students and books were hashed to a unique number. Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use hashing.
- **In hashing, large keys are converted into small keys** by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in $O(1)$ time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Introduction to Hash Tables



➤ Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

$\text{index} = \text{hashfunc}(\text{key}) \% \text{array_size}$

- In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and $\text{array_size} - 1$) by using the modulo operator (%).

Index		Index	
0		-	
1		-	
-		-	
-		23	bcdefa
-		-	
11	defabc	-	
12		-	
13		38	abcdef
14	cdefab	-	

Introduction to Hash Tables



- **The ideal hash table data structure is merely** an array of some fixed size containing the items. Generally a search is performed on some part (that is data member) of the item. This is called the key. For instance, an item could consist of a string (that serves as the key) and additional data members (for instance, a name that is part of a large employee structure). We will refer to the table size as `TableSize`, with the understanding that this is part of a hash data structure and not merely some variable floating around globally.
- **The common convention** is to have the table run from 0 to `TableSize - 1` we will see why shortly. Each key is mapped into some number in the range 0 to `TableSize - 1` and placed in the appropriate cell. The mapping is called a hash function, which ideally should be simple to compute and should ensure that any two distinct keys get different cells. Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is clearly impossible, and thus we seek a hash function that distributes the keys evenly among the cells.

Hash Functions



- **A good hash function** satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. Unfortunately, we typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently.
- **Occasionally we do know the distribution.** For example, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$, then the hash function $h(k) = k \cdot m$ satisfies the condition of simple uniform hashing.
- **In practice**, we can often employ heuristic techniques to create a hash function that performs well. Qualitative information about the distribution of keys may be useful in this design process. A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data. Finally, we note that some applications of hash functions might require stronger properties than are provided by simple uniform hashing. For example, we might want keys that are 'close' in some sense to yield hash values that are far apart.
- **A hash function is any function** that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

Hash Functions



- **To achieve a good hashing mechanism**, with the following basic requirements:
 - Easy to compute: It should be easy to compute and must not become an algorithm in itself.
 - Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
 - Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.
- **Note:** Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.
- **Need for a good hash function:** Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab”, “defabc”}. To compute the index for storing the strings, use a hash function that states the following:
 - The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599. As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.
 - The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.

Hash Functions



➤ Hash Functions which convert Integer Number to index

```
// This function calculates the hash value of the given key with type int
int hash_function_int(int x) {
    return x % table_size;
}

// This function calculates the hash value of the given key with type int
int hash_function_int_complex(int x) {
    return (37LL * (x % table_size)) % table_size;
}
```

➤ Hash Functions which convert Long Number to index

```
// This function calculates the hash value of the given key with type long long
int hash_function_long(long long x) {
    return x % table_size;
}

// This function calculates the hash value of the given key with type long long
int hash_function_long_complex(long long x) {
    return (37LL * (x % table_size)) % table_size;
}
```



Hash-
Functions.cpp

Hash Functions



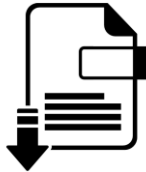
➤ Hash Functions which convert String to index

// This function calculates the hash value of the given key with type string

```
int hash_function_str(string x) {  
    int res = 0;  
    for (int i = 0 ; i < x.size() ; i++)  
        res = (res + x[i]) % table_size;  
    return res;  
}
```

// This function calculates the hash value of the given key with type string

```
int hash_function_str_complex(string x) {  
    int res = 0;  
    for (int i = 0 ; i < x.size() ; i++)  
        res = (res + 37LL * x[i]) % table_size;  
    return res;  
}
```



Hash-
Functions.cpp

Hash Functions



➤ Hash Functions which convert Character to index

```
// This function calculates the hash value of the given key with type char
int hash_function_char(char x) {
    return int(x) % table_size;
}

// This function calculates the hash value of the given key with type char
int hash_function_char_complex(char x) {
    return (37LL * int(x)) % table_size;
}
```



Hash-
Functions.cpp

Lecture Agenda



✓ Section 1: Introduction to Hash Tables

Section 2: Collision Resolution

Section 3: Separate Chaining

Section 4: Open Addressing

Section 5: Double Hashing

Section 6: Time Complexity & Space Complexity



Collision Resolution



➤ **Collision Handling:** Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

1- Separate Chaining (Open Hashing): The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

2- Open Addressing (Close Hashing): In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Types: (Linear Probing - Quadratic Probing - Double Hashing)

Collision Resolution



- **The main ideas, focusing on “chaining”** as a way to handle “collisions,” in which more than one key maps to the same array index. We present and analyze several variations on the basic theme. “open addressing,” is another way to deal with collisions. The bottom line is that hashing is an extremely effective and practical technique: the basic dictionary operations require only $O(1)$ time on the average.
- **A hash table is a data structure that** is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is $O(1)$. An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NULL if no existing phone number has hash function value equal to the index for the entry.

Collision Resolution

➤ Separate Chaining (Open Hashing)

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				

➤ Open Addressing (Close Hashing)

Index			Index	
0			-	
1			-	
-			-	
-			23	bcdefa
-			-	
11	defabc		-	
12			-	
13			38	abcdef
14	cdefab		-	

Lecture Agenda



✓ Section 1: Introduction to Hash Tables

✓ Section 2: Collision Resolution

Section 3: Separate Chaining

Section 4: Open Addressing

Section 5: Double Hashing

Section 6: Time Complexity & Space Complexity



Separate Chaining (Open Hashing)



- **Separate chaining is one of the most commonly used** collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.
- **The cost of a lookup is that of scanning** the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries N is much higher than the number of slots.
- **For separate chaining, the worst-case scenario** is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number N of entries in the table.

Separate Chaining (Open Hashing)



➤ The idea is to keep a list of all elements that hash to the same value. The array elements are pointers to the first nodes of the lists. A new item is inserted to the front of the list.

- Advantages:

- Better space utilization for large items.
- Simple collision handling: searching linked list.
- Overflow: we can store more items than the hash table size.
- Deletion is quick and easy: deletion from the linked list.

- Disadvantages:

- Cache performance of chaining is not good as keys are stored using linked list.
- cache performance as everything is stored in same table.
- Wastage of Space (Some Parts of hash table are never used)
- If the chain becomes long, then search time can become $O(n)$ in worst case.
- Uses extra space for links.

Separate Chaining Algorithms



➤ Initialization:

All entries are set to NULL

➤ Search:

Locate the cell using hash function.

sequential search on the linked list in that cell.

➤ Insertion:

Locate the cell using hash function.

(If the item does not exist) insert it as the first item in the list.

➤ Deletion:

Locate the cell using hash function.

(If the item exists) Delete the item from the linked list.

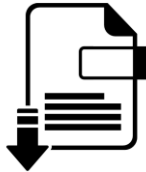
Linked List - Separate Chaining



- Initialize a global struct, single linked list node

```
#include <bits/stdc++.h>
using namespace std;

// A single linked list node
struct node {
    string data;
    node* next;
};
```



Separate-
Chaining.cpp

Insertion Operation - Linked List Separate Chaining



```
// This function inserts a node at the begin of the linked list
node* insert_begin(node* curr_head, string key) {
    // allocate new node and put it's data
    node* new_node = new node();
    new_node->data = key;
    // check if the linked list is empty
    if (curr_head == NULL) {
        curr_head = new_node;
    }
    // otherwise insert the new node in the begin of the linked list
    else {
        // set next of the new node to be the head
        new_node->next = curr_head;
        // set the new node as a head
        curr_head = new_node;
    }
    return curr_head;
}
```



Separate-
Chaining.cpp

Deletion Operation - Linked List Separate Chaining



```
// This function require a node to delete the node after it in the linked list
node* delete_node(node* curr_head, string key) {
    // check if the linked list is empty
    if (curr_head == NULL)
        return curr_head;
    // check if the first node in the list is the deleted node
    if (curr_head->data == key) {
        // get the node which it will be deleted
        node* temp_node = curr_head;
        // shift the head to be the next node
        curr_head = curr_head->next;
        // delete the temp node
        delete(temp_node);
    }
    // otherwise loop till reach the deleted node
    else {
```



Separate-
Chaining.cpp

Deletion Operation - Linked List Separate Chaining



```
// otherwise loop till reach the deleted node
else {
    // get the deleted node and the prev node of it in the linked list
    node* curr = curr_head;
    node* prev = NULL;
    while (curr != NULL && curr->data != key) {
        prev = curr;
        curr = curr->next;
    }
    // jump the deleted node
    prev->next = curr->next;
    // delete the node which selected
    delete(curr);
}
return curr_head;
}
```



Separate-
Chaining.cpp

Search Operation - Linked List Separate Chaining



```
// This function searches for a node in the linked list
bool search_node(node* curr_head, string key) {
    // iterate on the nodes till reach the last node in the linked list
    node* curr = curr_head;
    while (curr != NULL) {
        // check if the given key exists in the linked list
        if (curr->data == key)
            return true;
        curr = curr->next;
    }
    return false;
}
```



Separate-
Chaining.cpp

Traversal Operation - Linked List Separate Chaining

```
// This function prints the contents of the linked list
void print_linked_list(node* curr_head) {
    // print the data nodes starting from head till reach the last node
    node* curr = curr_head;
    while (curr != NULL) {
        cout << curr->data << "\n";
        curr = curr->next;
    }
}
```



Separate-
Chaining.cpp

Hash Function - Separate Chaining



- Initialize a global hash table with static length

```
// Initialize a hash table with static length
const int hash_table_size = 1e3+9;
// Initialize a global array pointer for heads
node* head[hash_table_size];
```

- This function calculates the hash value of the given key with type string

```
// This function calculates the hash value of the given key with type string
int hash_function_str_complex(string key) {
    int res = 0;
    for (int i = 0 ; i < key.size() ; i++)
        res = (res + 37LL * key[i]) % hash_table_size;
    return res;
}
```



Separate-
Chaining.cpp

Search Operation - Separate Chaining

```
// This function searches for an element in the hash table
bool search_item(string key) {
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // return whether the current position in the hash table
    // contain the given key
    return search_node(head[idx], key);
}
```



Separate-
Chaining.cpp

Insertion Operation - Separate Chaining



```
// This function inserts an element in the hash table
void insert_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == true)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // insert the new element
    head[idx] = insert_begin(head[idx], key);
}
```



Separate-
Chaining.cpp

Deletion Operation - Separate Chaining



```
// This function deletes an element at the given index in the hash table
void delete_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == false)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // delete the given key from it's list
    head[idx] = delete_node(head[idx], key);
}
```



Separate-
Chaining.cpp

Traversal Operation - Separate Chaining



```
// This function prints the contents of the hash table
void print_hash_table() {
    // loop to print the elements in the hash table
    for (int i = 0 ; i < hash_table_size ; i++) {
        if (head[i] == NULL)
            continue;
        cout << "index " << i << ":\n";
        print_linked_list(head[i]);
    }
}
```



Separate-
Chaining.cpp

Functionality Testing - Separate Chaining



➤ In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {  
    string item = "";  
    item += i;  
    item += i;  
    item += i;  
    for (int j = 0 ; j < 75 ; j++) {  
        insert_item(item);  
        item += i;  
    }  
}  
  
print_hash_table();  
cout << "-----\n";
```



Separate-
Chaining.cpp

Functionality Testing - Separate Chaining



➤ In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {  
    string item = "";  
    item += i;  
    item += i;  
    item += i;  
    for (int j = 0 ; j < 75 ; j++) {  
        if (search_item(item) == false)  
            cout << item << " not found\n";  
        item += i;  
    }  
}  
  
print_hash_table();  
cout << "-----\n";
```



Separate-
Chaining.cpp

Functionality Testing - Separate Chaining



➤ In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {  
    string item = "";  
    item += i;  
    item += i;  
    item += i;  
    for (int j = 0 ; j < 75 ; j++) {  
        if (j < 10) {  
            item += i;  
            continue;  
        }  
        delete_item(item);  
        item += i;  
    }  
}  
  
print_hash_table();  
cout << "-----\n";
```



Separate-
Chaining.cpp

Functionality Testing - Separate Chaining



➤ In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {  
    string item = "";  
    item += i;  
    item += i;  
    item += i;  
    for (int j = 0 ; j < 10 ; j++) {  
        if (search_item(item) == false)  
            cout << item << " not found\n";  
        item += i;  
    }  
}  
  
print_hash_table();  
cout << "-----\n";
```



Separate-
Chaining.cpp

Lecture Agenda



✓ Section 1: Introduction to Hash Tables

✓ Section 2: Collision Resolution

✓ Section 3: Separate Chaining

Section 4: Open Addressing

Section 5: Double Hashing

Section 6: Time Complexity & Space Complexity



Open Addressing (Close Hashing)



- **In open addressing, instead of in linked lists**, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.
- **The probe sequence is the sequence that is followed** while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes. When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.
- **Separate chaining has a disadvantage** of using lists. Requires the implementation of a second data Structure. In an open addressing hashing system, all the data go inside the table. Generally the load factor should be below 0.5 If a collision occurs, alternative cells are tried until an empty cell is found, Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Open Addressing Algorithms



➤ Initialization:

All entries are set to NULL

➤ Search:

Locate the cell using hash function.

Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

➤ Insertion:

Locate the cell using hash function.

Keep probing until an empty slot or a deleted slot is found. Once an empty slot is found, insert k .

➤ Deletion:

Locate the cell using hash function.

Keep probing until k is found. Once the item is found, mark it as a deleted slot.

- If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as “deleted”.
- Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.
- Cells $h_0(x)$, $h_1(x)$, $h_2(x)$, ... are tried in succession where $h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{TableSize}$.

Open Addressing - Linear Probing



- **In linear probing**, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also. let $\text{hash}(x)$ be the slot index computed using hash function and S be the table size
- **Clustering**: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element. In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found. i.e. f is a linear function of i , typically $f(i) = i$.
- **Linear probing** is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is index . The probing sequence for linear probing will be:

$\text{index} = \text{index} \% \text{hashTableSize}$

$\text{index} = (\text{index} + 1) \% \text{hashTableSize}$

$\text{index} = (\text{index} + 2) \% \text{hashTableSize}$

$\text{index} = (\text{index} + 3) \% \text{hashTableSize}$

and so on...

Open Addressing - Linear Probing

➤ Insert items with keys: 89, 18, 49, 58, 9 into an hash table. (Hash function is $\text{hash}(x) = x \bmod 10$)

Hash (89, 10) = 9

Hash (18, 10) = 8

Hash (49, 10) = 9

Hash (58, 10) = 8

Hash (9, 10) = 9

0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Hash Function - Linear Probing



- Initialize a global hash table with static length

```
#include <bits/stdc++.h>
using namespace std;

// Initialize a hash table with static length
const int hash_table_size = 1e3+9;
string arr[hash_table_size];
```

- This function calculates the hash value of the given key with type string

```
// This function calculates the hash value of the given key with type string
int hash_function_str_complex(string key) {
    int res = 0;
    for (int i = 0 ; i < key.size() ; i++)
        res = (res + 37LL * key[i]) % hash_table_size;
    return res;
}
```



Linear-
Probing.cpp

Search Operation - Linear Probing



```
// This function searches for an element in the hash table
bool search_item(string key) {
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i) % hash_table_size] != key &&
           arr[(idx + i) % hash_table_size] != "")
        i++;
    // return whether the current position in the hash table
    // contain the given key
    return arr[(idx + i) % hash_table_size] == key;
}
```

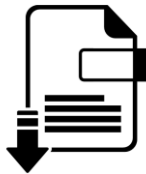


Linear-
Probing.cpp

Insertion Operation - Linear Probing



```
// This function inserts an element in the hash table
void insert_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == true)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // loop till find an empty position
    int i = 0;
    while (arr[(idx + i) % hash_table_size] != "" &&
           arr[(idx + i) % hash_table_size] != "#")
        i++;
    // insert the new element
    arr[(idx + i) % hash_table_size] = key;
}
```



Linear-
Probing.cpp

Deletion Operation - Linear Probing



```
// This function deletes an element at the given index in the hash table
void delete_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == false)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i) % hash_table_size] != key)
        i++;
    // set the position of the deleted element as a hash sign
    arr[(idx + i) % hash_table_size] = "#";
}
```



Linear-
Probing.cpp

Traversal Operation - Linear Probing



```
// This function prints the contents of the hash table
void print_hash_table() {
    // loop to print the elements in the hash table
    for (int i = 0 ; i < hash_table_size ; i++) {
        if (arr[i] == "" || arr[i] == "#")
            continue;
        cout << "index " << i << ": " << arr[i] << '\n';
    }
}
```



Linear-
Probing.cpp

Functionality Testing - Linear Probing



➤ In the Main function:

```
for (char c = 'a' ; c <= 'z' ; c++) {
    string item = "";
    for (int i = 0 ; i < 10 ; i++) {
        item += c;
        insert_item(item);
    }
}
for (char c = 'a' ; c <= 'z' ; c++) {
    string item = "";
    for (int i = 0 ; i < 10 ; i++) {
        if (search_item(item) == false)
            cout << item << " not found\n";
        item += c;
    }
}
print_hash_table();
cout << "-----\n";
```



Linear-
Probing.cpp

Functionality Testing - Linear Probing



➤ In the Main function:

```
for (char c = 'a' ; c <= 'z' ; c++) {
    string item = "";
    for (int j = 0 ; j < 10 ; j++) {
        item += c;
        if (j < 3)
            continue;
        delete_item(item);
    }
}

for (char c = 'a' ; c <= 'z' ; c++) {
    string item = "";
    for (int j = 0 ; j < 3 ; j++) {
        item += c;
        if (search_item(item) == false)
            cout << item << " not found\n";
    }
}

print_hash_table();
cout << "-----\n";
```



Linear-
Probing.cpp

Open Addressing - Quadratic Probing



- **Quadratic Probing eliminates** primary clustering problem of linear probing. Collision function is quadratic. The popular choice is $f(i) = i * i$. If the hash function evaluates to h and a search in cell h is inconclusive, we try cells $h + 1^2, h + 2^2, \dots, h + i^2$. Remember that subsequent probe points are a quadratic number of positions from the original probe point.
- **Quadratic probing is similar to linear probing** and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.
- The probing sequence for Quadratic probing will be:

$\text{index} = \text{index} \% \text{hashTableSize}$

$\text{index} = (\text{index} + 1^2) \% \text{hashTableSize}$

$\text{index} = (\text{index} + 2^2) \% \text{hashTableSize}$

$\text{index} = (\text{index} + 3^2) \% \text{hashTableSize}$

and so on...

Open Addressing - Quadratic Probing

➤ Insert items with keys: 89, 18, 49, 58, 9 into an hash table. (Hash function is $\text{hash}(x) = x \bmod 10$)

Hash (89, 10) = 9

Hash (18, 10) = 8

Hash (49, 10) = 9

Hash (58, 10) = 8

Hash (9, 10) = 9

0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Hash Function - Quadratic Probing



- Initialize a global hash table with static length

```
#include <bits/stdc++.h>
using namespace std;

// Initialize a hash table with static length
const int hash_table_size = 1e3+9;
string arr[hash_table_size];
```

- This function calculates the hash value of the given key with type string

```
// This function calculates the hash value of the given key with type string
int hash_function_str_complex(string key) {
    int res = 0;
    for (int i = 0 ; i < key.size() ; i++)
        res = (res + 37LL * key[i]) % hash_table_size;
    return res;
}
```



Quadratic-
Probing.cpp

Search Operation - Quadratic Probing



```
// This function searches for an element in the hash table
bool search_item(string key) {
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i*i) % hash_table_size] != key &&
           arr[(idx + i*i) % hash_table_size] != "")
        i++;
    // return whether the current position in the hash table
    // contain the given key
    return arr[(idx + i*i) % hash_table_size] == key;
}
```



Quadratic-
Probing.cpp

Insertion Operation - Quadratic Probing



```
// This function inserts an element in the hash table
void insert_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == true)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // loop till find an empty position
    int i = 0;
    while (arr[(idx + i*i) % hash_table_size] != "" &&
           arr[(idx + i*i) % hash_table_size] != "#")
        i++;
    // insert the new element
    arr[(idx + i*i) % hash_table_size] = key;
}
```



Quadratic-
Probing.cpp

Deletion Operation - Quadratic Probing



```
// This function deletes an element at the given index in the hash table
void delete_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == false)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i*i) % hash_table_size] != key)
        i++;
    // set the position of the deleted element as a hash sign
    arr[(idx + i*i) % hash_table_size] = "#";
}
```



Quadratic-
Probing.cpp

Traversal Operation - Quadratic Probing



```
// This function prints the contents of the hash table
void print_hash_table() {
    // loop to print the elements in the hash table
    for (int i = 0 ; i < hash_table_size ; i++) {
        if (arr[i] == "" || arr[i] == "#")
            continue;
        cout << "index " << i << ": " << arr[i] << '\n';
    }
}
```



Quadratic-
Probing.cpp

Functionality Testing - Quadratic Probing



➤ In the Main function:

```
for (char c = 'a' ; c <= 'z' ; c++) {  
    string item = "";  
    for (int i = 0 ; i < 10 ; i++) {  
        item += c;  
        insert_item(item);  
    }  
}  
for (char c = 'a' ; c <= 'z' ; c++) {  
    string item = "";  
    for (int i = 0 ; i < 10 ; i++) {  
        if (search_item(item) == false)  
            cout << item << " not found\n";  
        item += c;  
    }  
}  
print_hash_table();  
cout << "-----\n";
```



Quadratic-
Probing.cpp

Functionality Testing - Quadratic Probing



➤ In the Main function:

```
for (char c = 'a' ; c <= 'z' ; c++) {
    string item = "";
    for (int j = 0 ; j < 10 ; j++) {
        item += c;
        if (j < 3)
            continue;
        delete_item(item);
    }
}

for (char c = 'a' ; c <= 'z' ; c++) {
    string item = "";
    for (int j = 0 ; j < 3 ; j++) {
        item += c;
        if (search_item(item) == false)
            cout << item << " not found\n";
    }
}

print_hash_table();
cout << "-----\n";
```



Quadratic-
Probing.cpp

Lecture Agenda



✓ Section 1: Introduction to Hash Tables

✓ Section 2: Collision Resolution

✓ Section 3: Separate Chaining

✓ Section 4: Open Addressing

Section 5: Double Hashing

Section 6: Time Complexity & Space Complexity



Double Hashing



- **Double hashing** is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.
- **Let us say that the hashed index** for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

$\text{index} = (\text{index} + 1 * \text{indexH}) \% \text{hashTableSize}$

$\text{index} = (\text{index} + 2 * \text{indexH}) \% \text{hashTableSize}$

and so on... **Here**, indexH is the hash value that is computed by another hash function.

- A second hash function is used to drive the collision resolution. $f(i) = i * \text{hash}''(x)$, We apply a second hash function to x and probe at a distance $\text{hash}''(x)$, $2 * \text{hash}''(x)$, $3 * \text{hash}''(x)$, ... and so on.
- The function $\text{hash}''(x)$ must never evaluate to zero. e.g. Let $\text{hash}''(x) = x \bmod 9$ and try to insert 99 in the previous example.
- A function such as $\text{hash}''(x) = R - (x \bmod R)$ with R a prime smaller than TableSize will work well. e.g. try $R = 7$ for the previous example. $(7 - x \bmod 7)$.

Double Hashing

➤ Insert items with keys: 76, 93, 40, 47, 10, 55 into an hash table. (Hash function is $\text{hash}(x) = x \bmod 7$)

Prime Number: 5

insert(76) insert(93) insert(40) insert(47) insert(10) insert(55)
 $76\%7 = 6$ $93\%7 = 2$ $40\%7 = 5$ $47\%7 = 5$ $10\%7 = 3$ $55\%7 = 6$
 $5 - (47\%5) = 3$ $5 - (55\%5) = 5$

Hash (76, 7) = 6

Hash (93, 7) = 2

Hash (40, 7) = 5

Hash (47, 7) = 5

Hash (10, 7) = 3

Hash (55, 7) = 6

0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	2	1	2

Double Hashing Algorithms



➤ Initialization:

All entries are set to NULL

➤ Search:

Locate the cell using hash function.

Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

➤ Insertion:

Locate the cell using hash function.

Keep probing until an empty slot or a deleted slot is found. Once an empty slot is found, insert k .

➤ Deletion:

Locate the cell using hash function.

Keep probing until k is found. Once the item is found, mark it as a deleted slot.

- Use one hash function to determine the first slot and use a second hash function to determine the increment for the probe sequence. $H(k, i) = (h'(k) + i * h''(k)) \bmod \text{TableSize}$, $i = 0, 1, 2, \dots$
- Initial probe: $h'(k)$, Second probe is offset by $h''(k) \bmod \text{TableSize}$, so on ...

Hash Function - Double Hashing



- Initialize a global hash table with static length

```
#include <bits/stdc++.h>
using namespace std;
// Initialize a hash table with static length
const int hash_table_size = 1e3+13;
const int hash_table_prime = 1e3+9;
string arr[hash_table_size];
```

- These functions calculates the hash value of the given key with type string

```
// This function calculates the hash value of the given key with type string
int hash_function_str_complex(string key) {
    int res = 0;
    for (int i = 0 ; i < key.size() ; i++)
        res = (res + 37LL * key[i]) % hash_table_size;
    return res;
}
// This function calculates the hash value of the given hash value
int hash_function_secondary(int hash_idx) {
    return hash_table_prime - hash_idx % hash_table_prime;
}
```



Double-
Hashing.cpp

Search Operation - Double Hashing



```
// This function searches for an element in the hash table
bool search_item(string key) {
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // calculate the second hash value of the given hash value
    int step = hash_function_secondary(idx);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i*step) % hash_table_size] != key &&
           arr[(idx + i*step) % hash_table_size] != "")
        i++;
    // return whether the current position in the hash table
    // contain the given key
    return arr[(idx + i*step) % hash_table_size] == key;
}
```



Double-
Hashing.cpp

Insertion Operation - Double Hashing



```
// This function inserts an element in the hash table
void insert_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == true)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // calculate the second hash value of the given hash value
    int step = hash_function_secondary(idx);
    // loop till find an empty position
    int i = 0;
    while (arr[(idx + i*step) % hash_table_size] != "" &&
           arr[(idx + i*step) % hash_table_size] != "#")
        i++;
    // insert the new element
    arr[(idx + i*step) % hash_table_size] = key;
}
```



Double-
Hashing.cpp

Deletion Operation - Double Hashing



```
// This function deletes an element at the given index in the hash table
void delete_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == false)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // calculate the second hash value of the given hash value
    int step = hash_function_secondary(idx);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i*step) % hash_table_size] != key)
        i++;
    // set the position of the deleted element as a hash sign
    arr[(idx + i*step) % hash_table_size] = "#";
}
```



Double-
Hashing.cpp

Traversal Operation - Double Hashing



```
// This function prints the contents of the hash table
void print_hash_table() {
    // loop to print the elements in the hash table
    for (int i = 0 ; i < hash_table_size ; i++) {
        if (arr[i] == "" || arr[i] == "#")
            continue;
        cout << "index " << i << ": " << arr[i] << '\n';
    }
}
```



Double-
Hashing.cpp

Functionality Testing - Double Hashing



➤ In the Main function:

```
for (char c = 'a' ; c <= 'z' ; c++) {  
    string item = "";  
    for (int i = 0 ; i < 10 ; i++) {  
        item += c;  
        insert_item(item);  
    }  
}  
for (char c = 'a' ; c <= 'z' ; c++) {  
    string item = "";  
    for (int i = 0 ; i < 10 ; i++) {  
        if (search_item(item) == false)  
            cout << item << " not found\n";  
        item += c;  
    }  
}  
print_hash_table();  
cout << "-----\n";
```



Double-Hashing.cpp

Functionality Testing - Double Hashing



➤ In the Main function:

```
for (char c = 'a' ; c <= 'z' ; c++) {  
    string item = "";  
    for (int j = 0 ; j < 10 ; j++) {  
        item += c;  
        if (j < 3)  
            continue;  
        delete_item(item);  
    }  
}  
for (char c = 'a' ; c <= 'z' ; c++) {  
    string item = "";  
    for (int j = 0 ; j < 3 ; j++) {  
        item += c;  
        if (search_item(item) == false)  
            cout << item << " not found\n";  
    }  
}  
print_hash_table();  
cout << "-----\n";
```



Double-
Hashing.cpp

Lecture Agenda



- ✓ Section 1: Introduction to Hash Tables
- ✓ Section 2: Collision Resolution
- ✓ Section 3: Separate Chaining
- ✓ Section 4: Open Addressing
- ✓ Section 5: Double Hashing
- Section 6: Time Complexity & Space Complexity**



Time Complexity & Space Complexity



➤ **Load factor λ** of a hash table T is defined as follows:

- N = number of elements in T (**current size**)
- M = size of hash table T (**total size**)
- $\lambda = N/M$ (**load factor**) λ is the average length of chain
- **Unsuccessful** search time: $O(\lambda)$, **Successful** search time: $O(\lambda/2)$
- **Worst case** to insert or delete or search is $O(\lambda)$. Ideally, want $\lambda \leq 1$ (not a function of N)

Time Complexity & Space Complexity



- **Linear probing** has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.
- **Quadratic probing** lies between the two in terms of cache performance and clustering.
- **Double hashing** has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

LOAD FACTOR	SUCCESSFUL			UNSUCCESSFUL		
	<i>LINEAR</i>	<i>i + bkey</i>	<i>DOUBLE</i>	<i>LINEAR</i>	<i>i + bkey</i>	<i>DOUBLE</i>
25%	1.17	1.16	1.15	1.39	1.37	1.33
50%	1.50	1.44	1.39	2.50	2.19	2.00
75%	2.50	2.01	1.85	8.50	4.64	4.00
90%	5.50	2.85	2.56	50.50	11.40	10.00
95%	10.50	3.52	3.15	200.50	22.04	20.00

Separate Chaining vs. Open Addressing



Separate Chaining

- 1) Chaining is Simpler to implement.
- 2) In chaining, Hash table never fills up, we can always add more elements to chain.
- 3) Chaining is Less sensitive to the hash function or load factors.
- 4) Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- 5) Cache performance of chaining is not good as keys are stored using linked list.
- 6) Wastage of Space (Some Parts of hash table in chaining are never used).
- 7) Chaining uses extra space for links.

Open Addressing

- 1) Open Addressing requires more computation.
- 2) In open addressing, table may become full.
- 3) Open addressing requires extra care for to avoid clustering and load factor.
- 4) Open addressing is used when the frequency and number of keys is known.
- 5) Open addressing provides better cache performance as everything is stored in same table.
- 6) In Open addressing, a slot can be used even if an input doesn't map to it.
- 7) No links in Open addressing.

Lecture Agenda



- ✓ Section 1: Introduction to Hash Tables
- ✓ Section 2: Collision Resolution
- ✓ Section 3: Separate Chaining
- ✓ Section 4: Open Addressing
- ✓ Section 5: Double Hashing
- ✓ Section 6: Time Complexity & Space Complexity



Practice



Practice



- 1- Find if an array is subset of another array
- 2- Union and Intersection of two linked lists
- 3- Check for pair in array with sum as number x
- 4- Minimum delete operations to make all elements of array same
- 5- Minimum operation to make all elements equal in array
- 6- Count maximum points on same line
- 7- Check if a given array contains duplicate elements within k distance from each other
- 8- Find duplicates in a given array when elements are not limited to a range
- 9- Find top k (or most frequent) numbers in a stream
- 10- Find most frequent element in an array
- 11- Find smallest subarray with all occurrences of a most frequent element
- 12- First element occurring k times in an array
- 13- Given an array of pairs, find all symmetric pairs in it
- 14- Find the only repetitive element between 1 to n-1
- 15- Find any one of the multiple repeating elements in read only array

Practice



- 16- Find top three repeated in array
- 17- Group multiple occurrence of array elements ordered by first occurrence
- 18- Check if two given sets are disjoint
- 19- Sum of non-overlapping sum of two sets
- 20- Find elements which are present in first array and not in second
- 21- Check if two arrays are equal or not
- 22- Find pair with given sum and maximum shortest distance from end
- 23- Find pair with given product
- 24- Find missing elements of a range
- 25- Find k-th missing element in increasing sequence which is not present
- 26- Find pair with greatest product in array
- 27- Minimum number of subsets with distinct elements
- 28- Remove minimum number of elements such that no common element exist in both array
- 29- Count items common to both the lists but with different prices
- 30- Minimum index sum for common elements of two lists

Practice



- 31- Find pairs with given sum such that elements of pair are in different rows
- 32- Common elements in all rows of a given matrix
- 33- Find distinct elements common to all rows of a matrix
- 34- Find all permuted rows of a given row in a matrix
- 35- Change the array into a permutation of numbers from 1 to n
- 36- Count pairs from two sorted arrays whose sum is equal to a given value x
- 37- Count pairs from two linked lists whose sum is equal to a given value
- 38- Count quadruples from four sorted arrays whose sum is equal to a given value x
- 39- Calculate number of subarrays having sum exactly equal to k
- 40- Count pairs whose products exist in array
- 41- Find all pairs whose sum is x in two unsorted arrays
- 42- Cumulative frequency of count of each element in an unsorted array
- 43- Numbers with prime frequencies greater than or equal to k
- 44- Find pairs in array whose sums already exist in array
- 45- Find all pairs (a, b) in an array such that $a \% b = k$

Practice



- 46- Convert an array to reduced form
- 47- Return maximum occurring character in an input string
- 48- Second most repeated word in a sequence
- 49- Smallest element repeated exactly 'k' times (not limited to small range)
- 50- Find k numbers with most occurrences in the given array
- 51- Find the first repeating element in an array of integers
- 52- Find sum of non-repeating (distinct) elements in an array
- 53- Find non-repeating element
- 54- Find k-th distinct (or non-repeating) element in an array
- 55- Print all distinct elements of a given integer array
- 56- Find only integer with positive value in positive negative value in array
- 57- Pairs of positive negative values in an array
- 58- Count pairs with given sum
- 59- Group words with same set of characters
- 60- Maximum distance between two occurrences of same element in array

Assignment



Implement STL Unordered Set



- Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.
- In an `unordered_set`, the value of an element is at the same time its key, that identifies it uniquely. Keys are immutable, therefore, the elements in an `unordered_set` cannot be modified once in the container - they can be inserted and removed, though.
- Internally, the elements in the `unordered_set` are not sorted in any particular order, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their values (with a constant average time complexity on average).
- `unordered_set` containers are faster than set containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.
- An `unordered_set` is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized.
- The `unordered_set` can contain key of any type - predefined or user-defined data structure but when we define key of type user define the type, we need to specify our comparison function according to which keys will be compared.

Implement STL Unordered Set



- **Sets vs. Unordered Sets:** Set is an ordered sequence of unique keys whereas `unordered_set` is a set in which key can be stored in any order, so unordered. Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal).
- **Methods on Unordered Sets:** For `unordered_set` many functions are defined among which most users are the `size` and `empty` for capacity, `find` for searching a key, `insert` and `erase` for modification. The `Unordered_set` allows only unique keys, for duplicate keys `unordered_multiset` should be used.

More Info: cplusplus.com/reference/unordered_set/unordered_set/

More Info: en.cppreference.com/w/cpp/container/unordered_set

More Info: [geeksforgeeks.org/unordered_set-in-cpp-stl/](https://www.geeksforgeeks.org/unordered_set-in-cpp-stl/)

More Info: [geeksforgeeks.org/set-in-cpp-stl/](https://www.geeksforgeeks.org/set-in-cpp-stl/)

Implement STL Unordered Set



- Member functions: **(constructor)** Construct vector (public member function)
 (destructor) Vector destructor (public member function)
 (operator=) Assign content (public member function)
- Iterators: **(begin)** Return iterator to beginning (public member function)
 (end) Return iterator to end (public member function)
 (cbegin) Return const_iterator to beginning (public member function)
 (cend) Return const_iterator to end (public member function)
- Capacity: **(empty)** Test whether vector is empty (public member function)
 (size) Return size (public member function)
 (max_size) Return maximum size (public member function)
- Operations: **(find)** Get iterator to element (public member function)
 (count) Count elements with a specific value (public member function)
 (equal_range) Get range of equal elements (public member function)

Implement STL Unordered Set



- **Modifiers:**
 - (**insert**) Insert element (public member function)
 - (**erase**) Erase elements (public member function)
 - (**swap**) Swap content (public member function)
 - (**clear**) Clear content (public member function)
- **Buckets:**
 - (**bucket_count**) Return number of buckets (public member function)
 - (**max_bucket_count**) Return maximum number of buckets (public member function)
 - (**bucket_size**) Return bucket size (public member type)
 - (**bucket**) Locate element's bucket (public member function)
- **Hash policy:**
 - (**load_factor**) Return load factor (public member function)
 - (**max_load_factor**) Get or set maximum load factor (public member function)
 - (**rehash**) Set number of buckets (public member function)
 - (**reserve**) Request a capacity change (public member function)

More Info: cplusplus.com/reference/unordered_set/unordered_set/

More Info: en.cppreference.com/w/cpp/container/unordered_set

Implement STL Unordered Map



- Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.
- In an `unordered_map`, the key value is generally used to uniquely identify the element, while the mapped value is an object with the content associated to this key. Types of key and mapped value may differ.
- Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).
- `unordered_map` containers are faster than `map` containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.
- Unordered maps implement the direct access operator (`operator[]`) which allows for direct access of the mapped value using its key value as argument.

Implement STL Unordered Map



- `unordered_map` is an associated container that stores elements formed by combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.
- Internally `unordered_map` is implemented using Hash Table, the key provided to map are hashed into indices of hash table.
- `unordered_map` vs. `unordered_set`: In `unordered_set`, we have only key, no value, these are mainly used to see presence/absence in a set.
- `unordered_map` vs. `map`: `map` (like set) is an ordered sequence of unique keys whereas in `unordered_map` key can be stored in any order, so unordered. `Map` is implemented as balanced tree structure that is why it is possible to maintain an order between the elements (by specific tree traversal).

More Info: [cplusplus.com/reference/unordered_map/unordered_map/](https://en.cppreference.com/reference/unordered_map/unordered_map/)

More Info: en.cppreference.com/w/cpp/container/unordered_map

More Info: [geeksforgeeks.org/unordered-map-in-cpp-stl/](https://www.geeksforgeeks.org/unordered-map-in-cpp-stl/)

More Info: [geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/](https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)

Implement STL Unordered Map



- Member functions: **(constructor)** Construct vector (public member function)
(destructor) Vector destructor (public member function)
(operator=) Assign content (public member function)
- Iterators: **(begin)** Return iterator to beginning (public member function)
(end) Return iterator to end (public member function)
(cbegin) Return const_iterator to beginning (public member function)
(cend) Return const_iterator to end (public member function)
- Capacity: **(empty)** Test whether vector is empty (public member function)
(size) Return size (public member function)
(max_size) Return maximum size (public member function)
- Modifiers: **(insert)** Insert element (public member function)
(erase) Erase elements (public member function)
(swap) Swap content (public member function)
(clear) Clear content (public member function)

Implement STL Unordered Map



- Element access: `(operator[])` Access element (public member function)
`(at)` Access element (public member function)
- Operations: `(find)` Get iterator to element (public member function)
`(count)` Count elements with a specific value (public member function)
`(equal_range)` Get range of equal elements (public member function)
- Buckets: `(bucket_count)` Return number of buckets (public member function)
`(max_bucket_count)` Return maximum number of buckets (public member function)
`(bucket_size)` Return bucket size (public member type)
`(bucket)` Locate element's bucket (public member function)

Implement STL Unordered Map



- Hash policy:
 - (load_factor) Return load factor (public member function)
 - (max_load_factor) Get or set maximum load factor (public member function)
 - (rehash) Set number of buckets (public member function)
 - (reserve) Request a capacity change (public member function)

More Info: cplusplus.com/reference/unordered_map/unordered_map/

More Info: en.cppreference.com/w/cpp/container/unordered_map



DO
MORE.