Algorithms

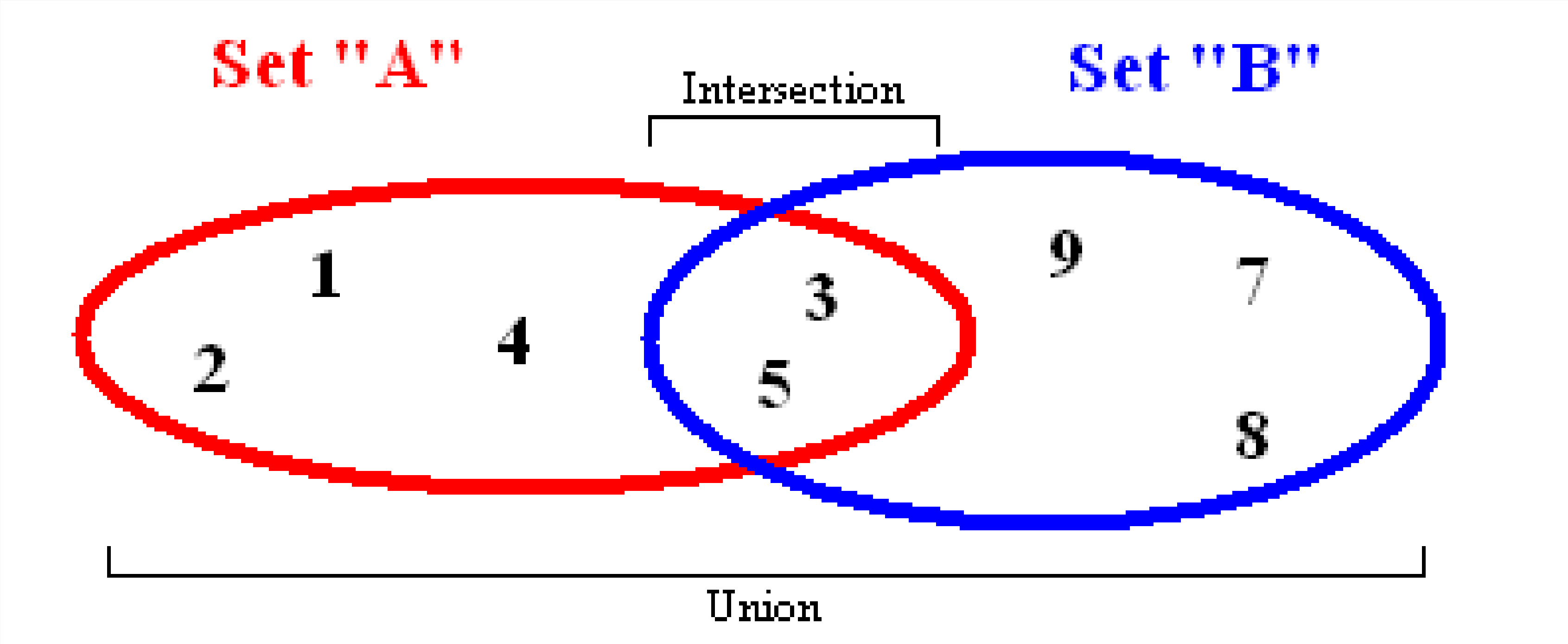
Sets & Hash Tables

Introduction

What is a Set?

From the point of view of **data structures**, a set is a **group of data without relationships between them**.

When working with a set, the important thing is the **membership or not** of an element in the set.



What can an ADT Set store?

- Elements of different types.

- Non-repeated elements.

Depending on the application, the data can be:

- **Atomic elements** (simple data), or.

- **Structured elements** (objects with attributes, where one of these attributes is a key).

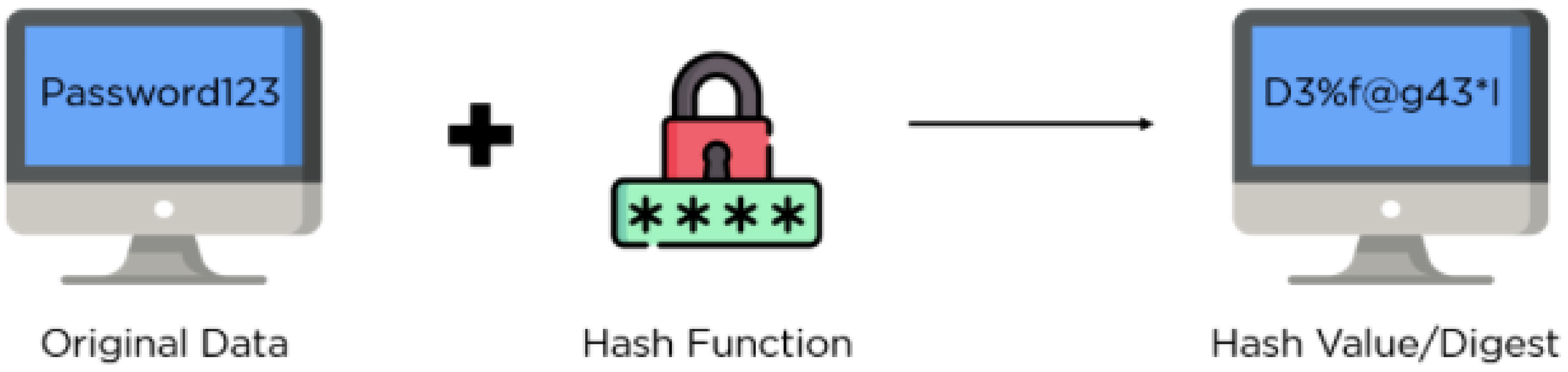
Hashing

What is Hashing?

Hashing is a technique of **mapping a large set of arbitrary data to tabular indexes** using a **mathematical function** that is **uniform**, **consistent**, and **one-way**. It is a method for representing **dictionaries** for large datasets.

It allows fast **lookups**, **updating** and **retrieval** operations.

The idea is to take a piece of information and passes it through a function that performs mathematical operations.



They are designed to be **irreversible**, which means your **hash value** should not provide you with any clue about the **original data**.

Hash functions also provide **the same output value if the input remains unchanged**, irrespective of the number of iterations.

Why is Hashing Needed?

After storing a large amount of data, we need to perform various operations on these data.

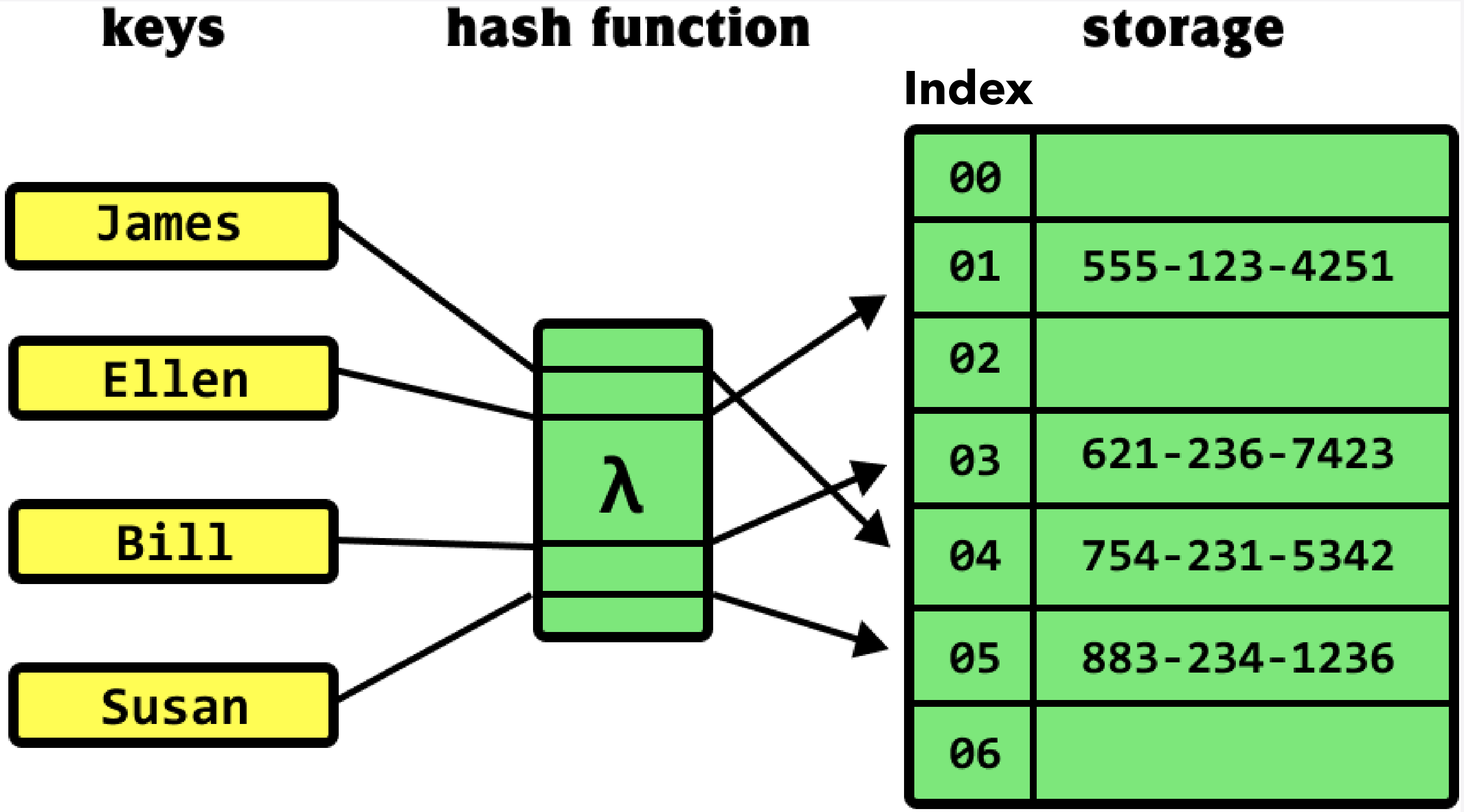
Lookups are inevitable in many tasks. **Linear search** and **binary search** perform lookups/search with time complexity of **O(n)** and **O(log n)** respectively.

As **the size of the dataset increases, these complexities also become significantly high** which is not acceptable.

We need a **technique that does not depend on the size of data**. Hashing allows **lookups** to occur in constant time i.e., **O(1)**.

What is a Hash Table?

A **hash table** is a **data structure** that you can use to store data in **key-value** format with direct access to its items in **constant time**.



Hash tables are said to be **associative**, which means that for each **key**, **ideally**, data **occurs at most once**. This let us store/access *data values* in the associated *index* (**key** value).

Why use Hash Tables?

The most valuable aspect of a **hash table** over other abstract data structures is its speed to perform **insertion**, **deletion**, and **search** operations. Hash tables can do them all in **constant time**, **theoretically**.

Algorithm	Average	Worst case
List	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

How do hash tables work?

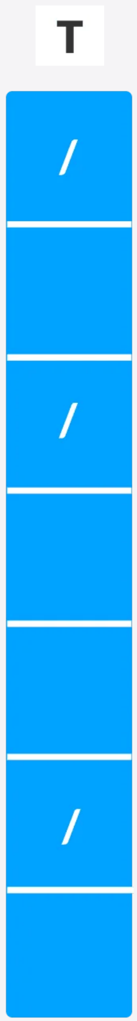
The Hash table data structure stores elements in **key-value pairs** where

- **Key** - unique integer that is used for indexing the values.
- **Value** - data that are associated with keys.

Hash Tables - Storage

A hash table is an abstract data type (ADT) that **relies on using a more primitive data type** (such as an **array** or an **object**) to store the data.

You can use either, but slight implementation implications occur depending on what you choose.



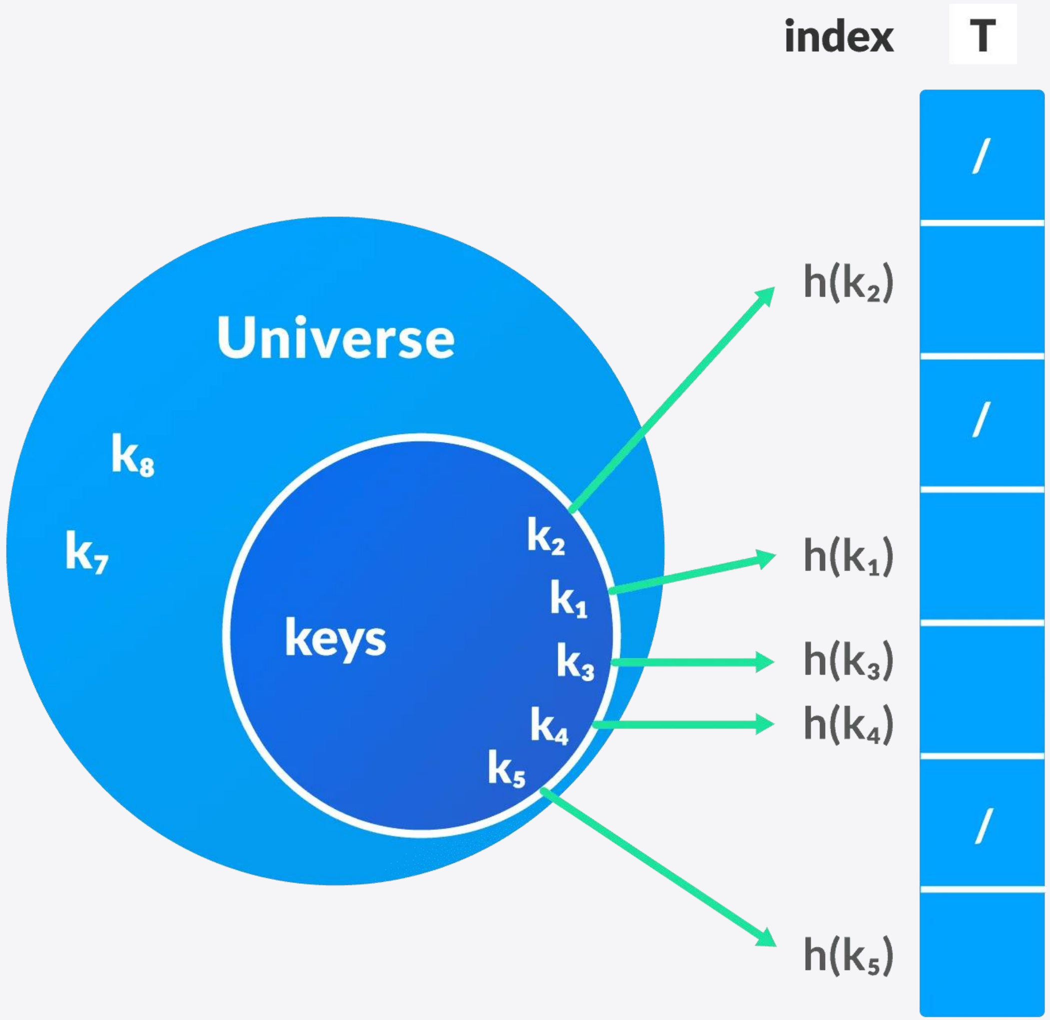
Hash Tables - (key, value) pairs

In a **hash table**, an **index** is generated using the **keys**. The **element** corresponding to that **key** is stored in the **index** given by the hash function.

Let **k** be a **key** and **h()** be a **hash function**. Here, **h(k)** will give us the **index** to store the **element** linked with **k**.

index = **h(k)**

element = **T[index]**



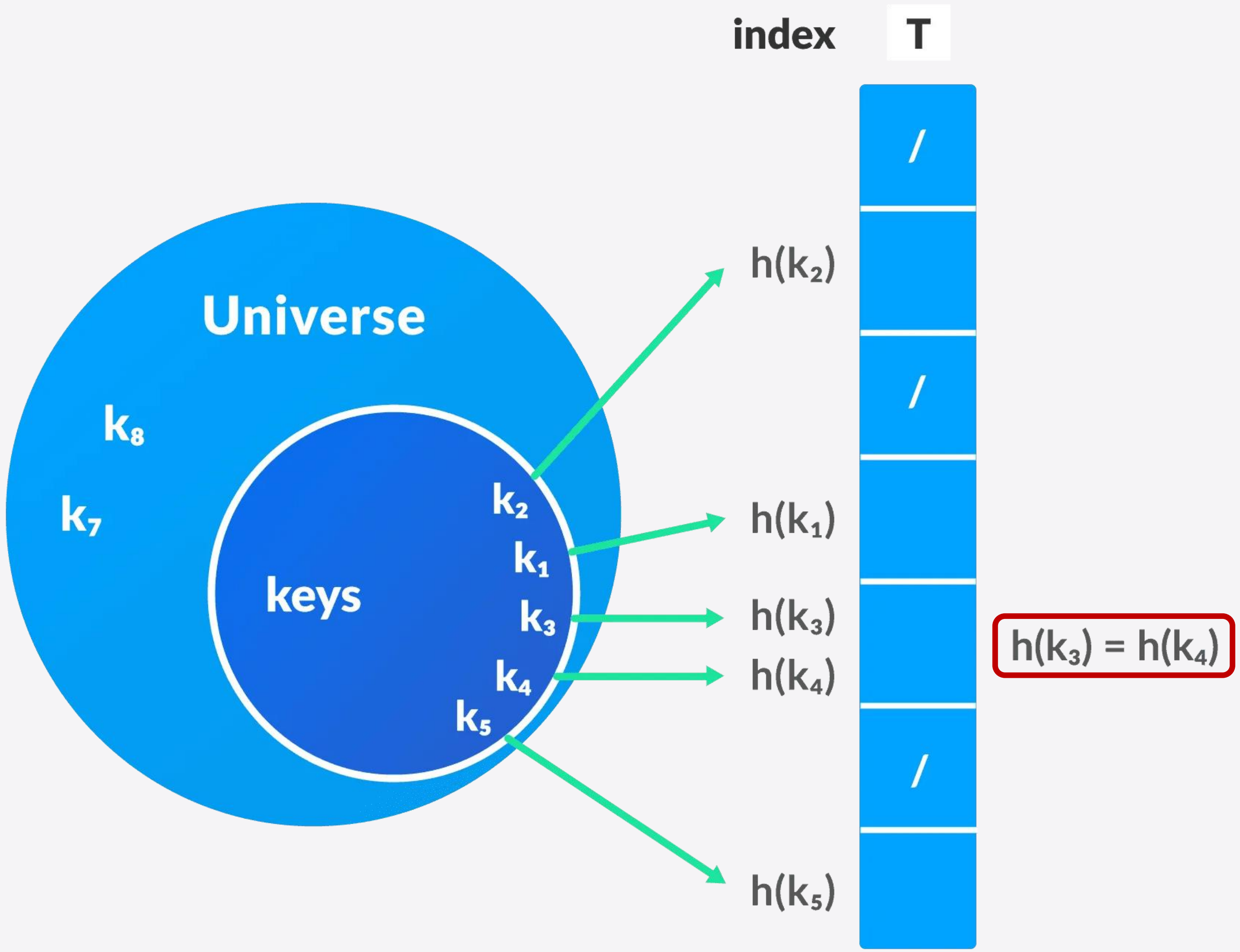
Hash Tables - (key, value) pairs

After deciding on an **index**, the hash function can **insert** or **merge** the value at the position specified by the **index**.

We can use this to **encounter duplicated data**.

However, In implementations where we are **not merging data**, **What happened if two different keys have the same index?**

This is generally referred to as a **collision**.



Hash Collision

When the **hash function generates the same index for multiple keys**, there will be a conflict (what value to be stored in that index). This is called a **hash collision**.

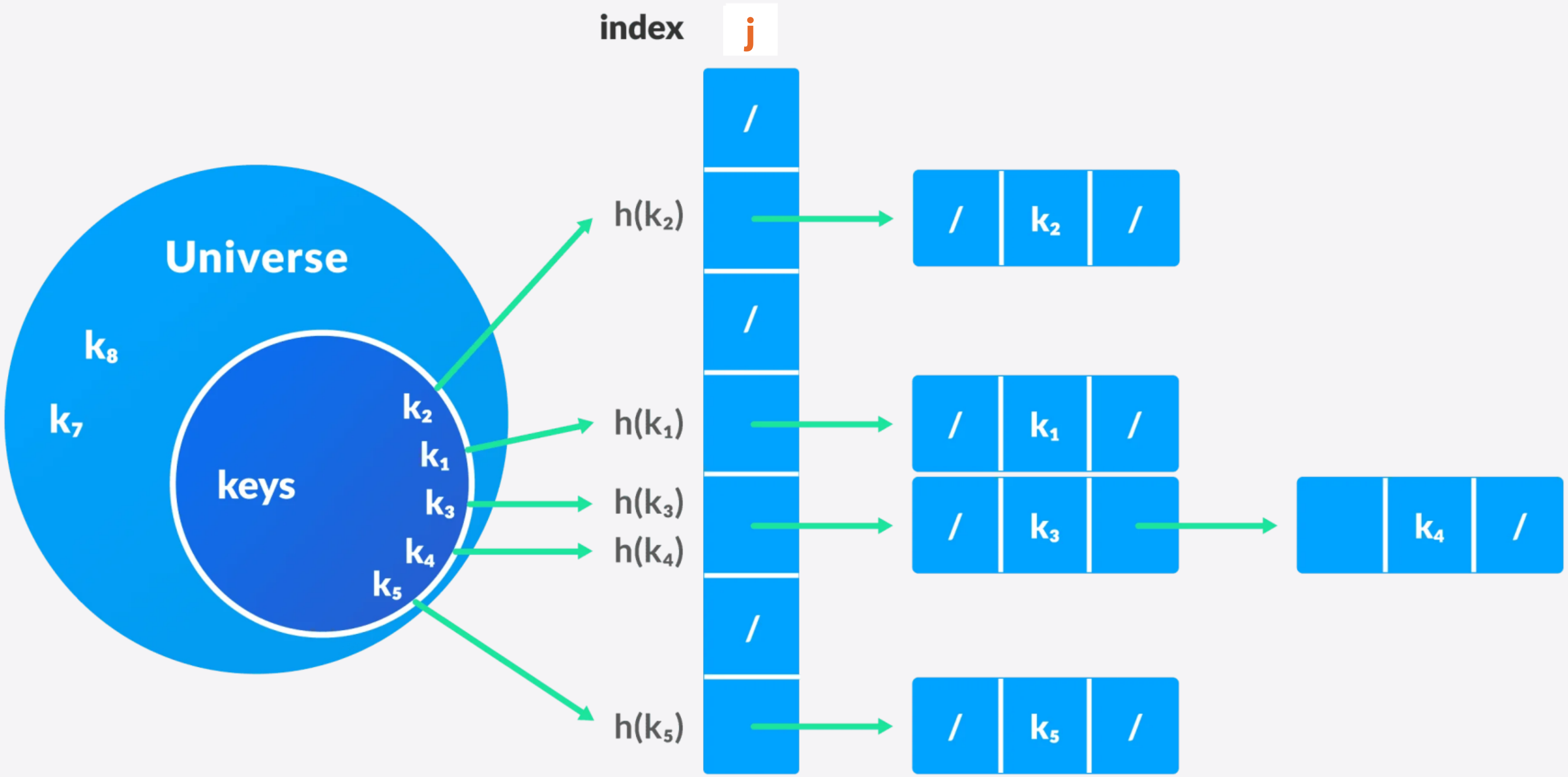
We can resolve the hash collision using one of the following techniques.

- **Collision resolution by chaining**
- **Open Addressing**: **Linear/Quadratic Probing** and **Double Hashing**

1. Collision resolution by chaining

The collided elements are stored in the same index by using a **doubly-linked list**.

Each **index** of the array contain a **pointer to the head of a list**, we append the collided elements to the list. If no element is present, the index contains ***NULL***.



Pseudocode

Pseudocode for operations: **Search**, **Insert**, **Delete**

```
chainedHashSearch(T, k)
  return T[h(k)]
chainedHashInsert(T, x)
  T[h(x.key)] = x //insert at the head
chainedHashDelete(T, x)
  T[h(x.key)] = NIL
```

2. Open Addressing

Unlike **chaining**, **open addressing doesn't store multiple elements into the same slot**. Here, each slot is either filled with a **single key** or left **NULL**.

Different techniques used in open addressing are:

I. Linear Probing
II. Quadratic Probing
III. Double Hashing

I. Linear Probing

In linear probing, collision is resolved by checking the next slot.

$$h'(k, i) = (h(k) + i) \bmod m$$

Where:

- $i = \{0, 1, 2, \ldots\}$
- **h(k)** is the original hash function
- **h'(k, i)** is the final hash function

If a collision occurs at the original position **h(k, i=0)**, the value of **i** increase linearly until you find the right place. Probing the next sequence:

h(k) mod m

(h(k) + 1) mod m

(h(k) + 2) mod m

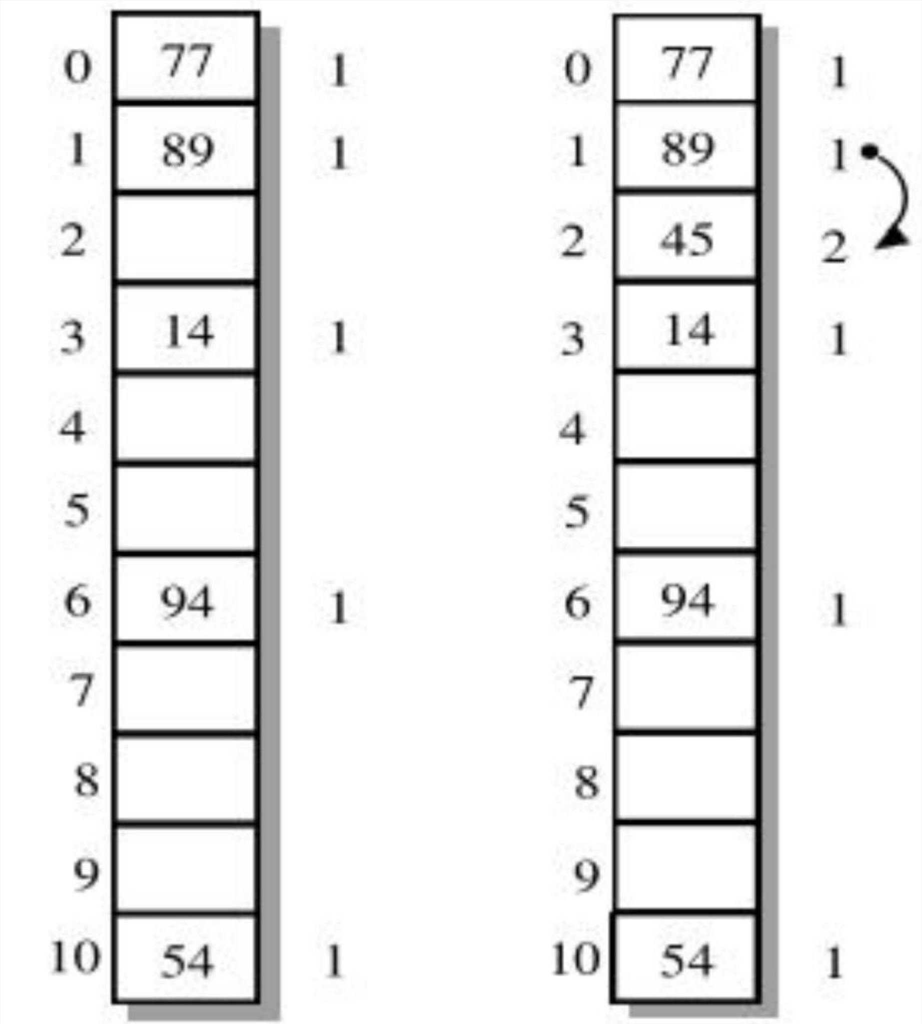
...

Example. In the next example, assume **h'(k) = k mod 11**

0	77	1
1	89	1
2		
3	14	1
4		
5		
6	94	1
7		
8		
9		
10	54	1

Insert: 54, 77, 94, 89, 14

Example. In the next example, assume **h'(k) = k mod 11**

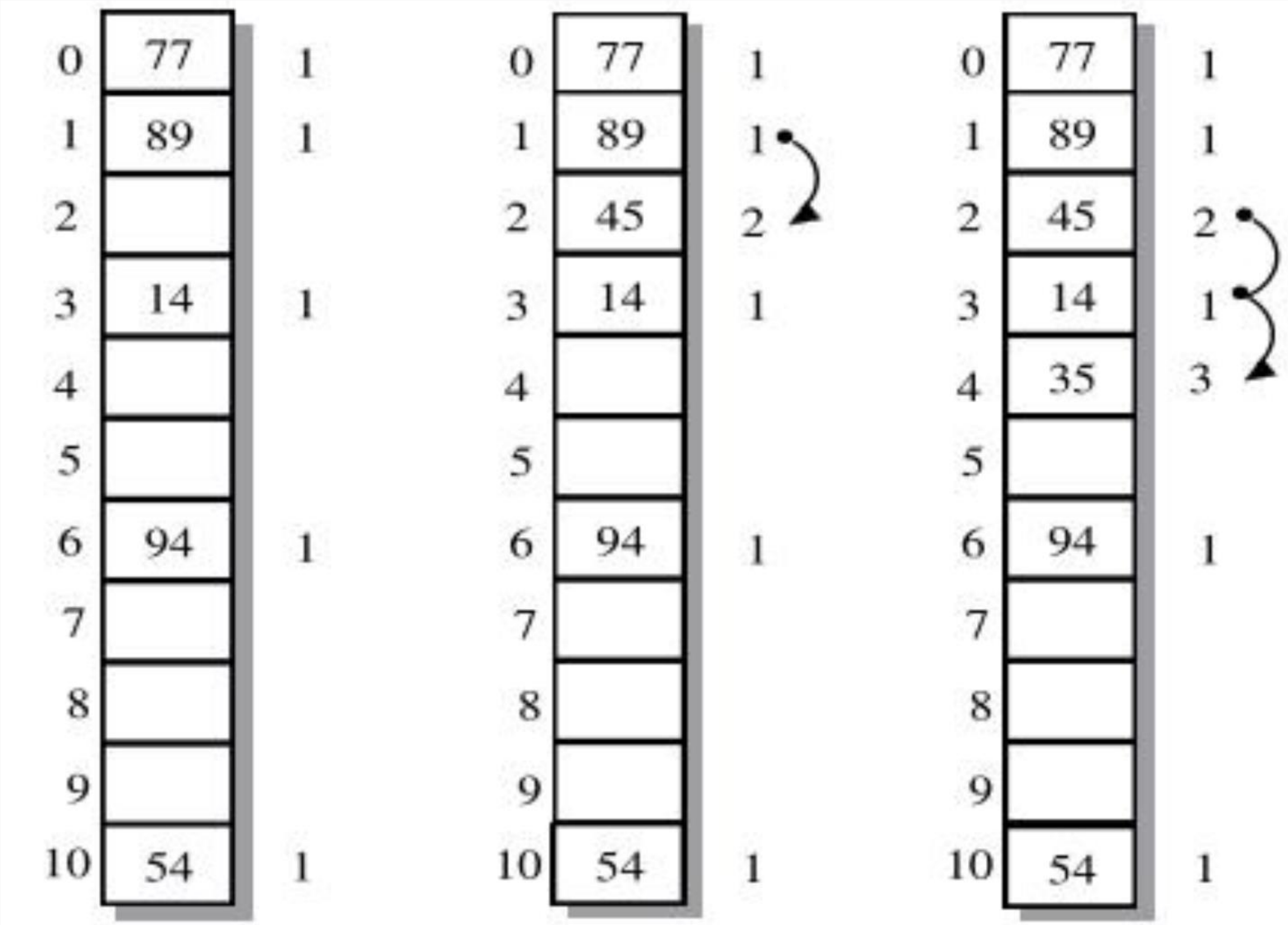


Insert: 54, 77, 94, 89, 14

Insert: 45

45 mod 11 = 1

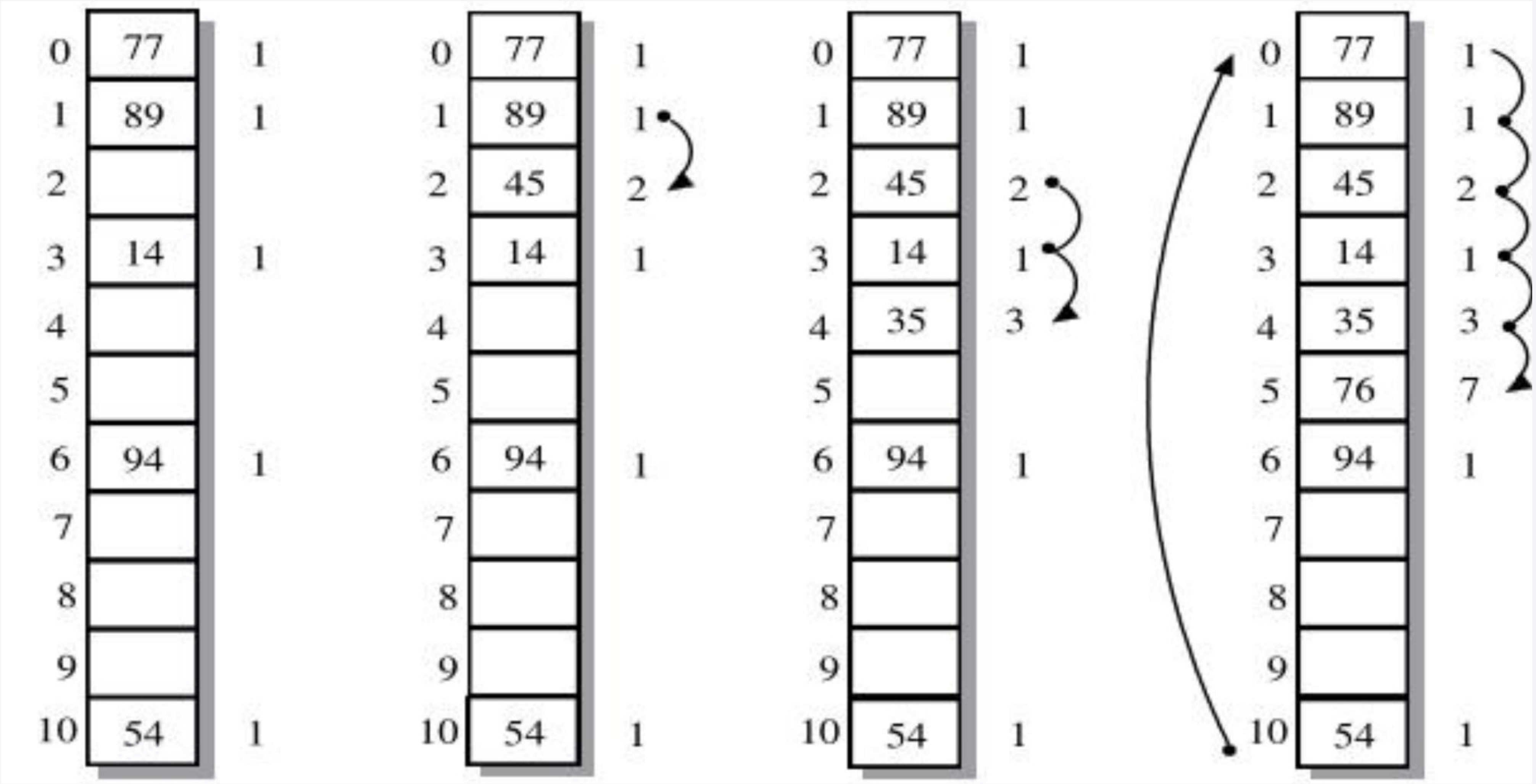
Example. In the next example, assume **h'(k) = k mod 11**



Insert: 54, 77, 94, 89, 14 **Insert:** 45 **Insert:** 35

45 mod 11 = 1 **35 mod 11** = 2

Example. In the next example, assume **h'(k) = k mod 11**



Insert: 54, 77, 94, 89, 14

Insert: 45

Insert: 35

Insert: 76

45 mod 11 = 1 **35 mod 11** = 2 **76 mod 11** = 10

The **problem with linear probing** is that a cluster of adjacent slots is filled. **When inserting a new element, the entire cluster must be traversed**. This adds to the time required to perform operations on the hash table.

II. Quadratic Probing

It works like linear probing but with a quadratic spacing between the slots:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Where,

- $i = \{0, 1, \ldots.\}$
- c_1 and c_2 are positive auxiliary constants,

If we assume a collision at the original position and c_1 **= 0** the value, the next sequence is followed:

$h(k) \bmod m$

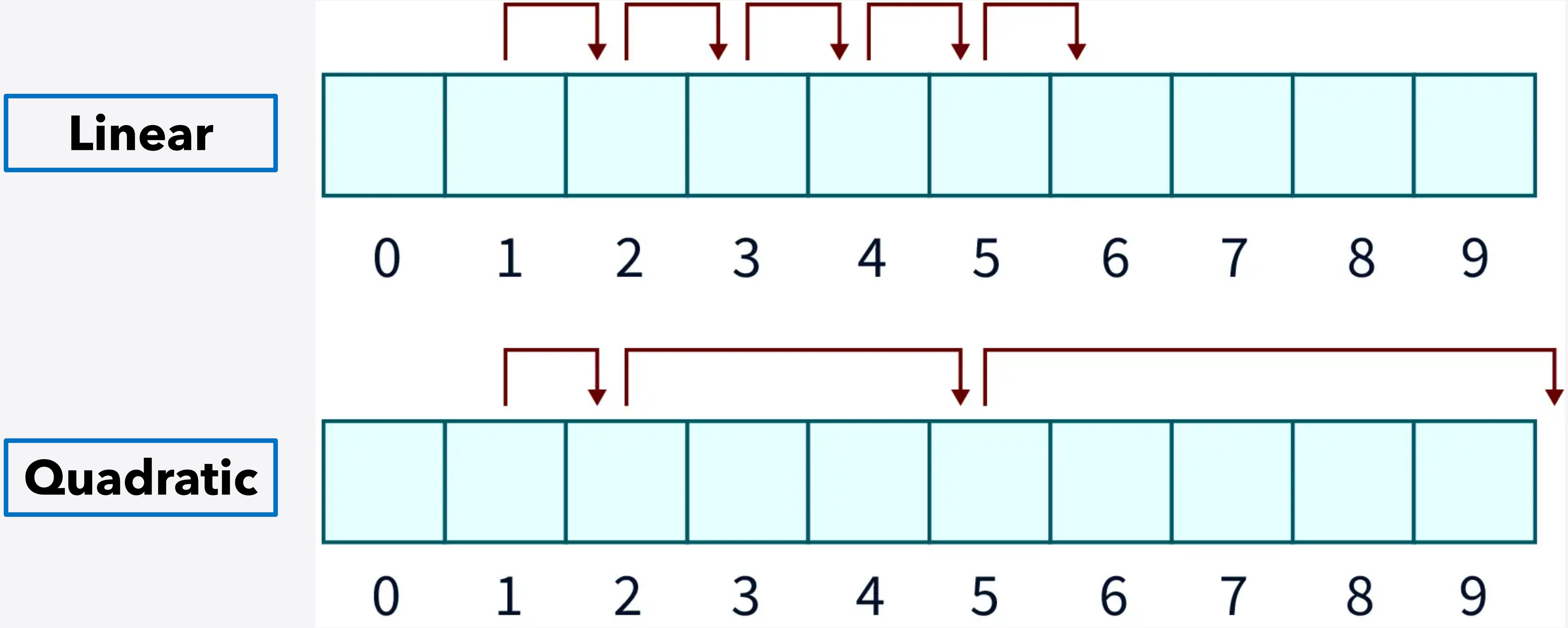
$(h(k) + 1) \bmod m$

$(h(k) + 4) \bmod m$

$(h(k) + 9) \bmod m$

...

Example. Difference between linear and quadratic probing



III. Double hashing

If a collision occurs after applying a hash function **h(k)**, then another hash function is calculated for finding the next slot.

$$\mathbf{h(k,\ i) = (h_1(k) + i * h_2(k))\ mod\ m}$$

Where,

- $\mathbf{h_1}$ and $\mathbf{h_2}$ are different hash functions,
- **i** = {0, 1, ….}

Summarizing

So far, we have already seen **how a hash table works** and some common **methods for collision resolution**.

Next, we will look at criteria and examples of good **hash functions**.

Good Hash Functions

A good hash function may not prevent the collisions completely however **it can reduce the number of collisions**.

There are different methods to find a good hash function, for example:

I. Division Method
II. Multiplication Method
III. Universal Hashing

I. Division Method

If **k** is a **key** and **m** is the **size of the hash table**, the hash function **h()** is calculated as:

$$h(k) = k \bmod m$$

Example.

If the size of a hash table is **10** and **k = 112** then **h(k) = 112 mod 10 = 2**.

In this cases, the value of **m** must not be a **power of 2**.

Why?

There are many numbers that are powers of two, this would increase the number of collisions.

II. Multiplication Method

Here, the hash function **h()** is calculated as:

$$h(k) = \lfloor m(k*A \bmod 1) \rfloor$$

Where,

- **(k*A) mod 1** gives the fractional part of **k*A**,
- **m** hash table size
- $\lfloor\ \rfloor$ gives the floor value
- **A** is any constant. The value of **A** lies between 0 and 1. But, an optimal choice will be **≈ $(\sqrt{5}-1)/2$** suggested by Knuth.

III. Universal hashing

In practice, the keys are not evenly distributed. **Universal hashing** refers to **selecting a hash function randomly from a family of hash functions**.

By choosing a hash function randomly from a **set H**, the **probability of collision** between different keys is not greater than **1/m**, where **m** is the size of the hash table.

- Designing a universal class of hash functions

We start by choosing a **prime number p** large enough so that every possible key **k** is in the range **0** to **p-1**.

We define the hash function for any **a, b in [0, p-1]**, using a **linear transformation** followed by a **reduction modulo p** and then **modulo m**.

$$h(k) = ((a*k + b) \bmod p) \bmod m$$

Assuming that the size of the possible keys is greater than the number of spaces in the hash table, **p>m**.

Implementation

```cpp
class HashTable
{
    private:
        int capacity;
        list<int> *table;

    public:
        HashTable(int V)
        {
            int size = getPrime(V);
            this->capacity = size;
            table = new list<int>[capacity];
        }
        ~HashTable()
        {
            delete []table;
            cout << "\nDestructor: HashTable deleted.\n";
        }
        bool checkPrime(int);
        int getPrime(int);

        void insertItem(int);
        void deleteItem(int);
        int hashFunction(int);
        void displayHash();
};
```

```cpp
bool HashTable::checkPrime(int n)
{
    if (n == 1 || n == 0)
        return false;

    int sqr_root = sqrt(n);
    for (int i = 2; i <= sqr_root; i++)
    {
        if (n % i == 0)
            return false;
    }
    return true;
}

int HashTable::getPrime(int n)
{
    if (n % 2 == 0)
        n++;

    while (!checkPrime(n))
    {
        n += 2;
    }
    return n;
}
```

Insert - Implementation

The Insertion works as follows:

1. The function receives the value to store.
2. Get the hash index of the key i.e., value to store (The hash function may vary).
3. Insert at the given position, where a collision resolution method will be used if necessary.

```
int HashTable::hashFunction(int key)
{
    return (key % capacity);
}

void HashTable::insertItem(int data)
{
    int index = hashFunction(data);

    table[index].push_back(data);
}
```

Delete - Implementation

The deletion works as follows:

1. The function receives the value to delete.
2. Get the hash index of the key.
3. Find the key in (index)-th list. You will probably have to look in more than one place if there was a collision.
4. Delete the element if present.

```
void HashTable::deleteItem(int key)
{
    int index = hashFunction(key);

    table[index].remove(key);
}
```

Alternative:

```
void HashTable::deleteItem(int key)
{
    int index = hashFunction(key);

    list<int>::iterator i;
    for (i = table[index].begin(); i != table[index].end(); i++)
    {
        if (*i == key)
            break;
    }

    if (i != table[index].end())
        table[index].erase(i);
}
```

Print Table - Implementation

For each space in the hash table, we print its contents

```
void HashTable::displayHash()
{
    for (int i = 0; i < capacity; i++)
    {
        cout << "table[" << i << "]";

        for (auto x : table[i])
            cout << " --> " << x;

        cout << endl;
    }
}
```

Program body

```cpp
int main()
{
    int data[] = {231, 321, 212, 321, 433, 262};

    int size = sizeof(data) / sizeof(data[0]);

    HashTable h(size);

    for (int i = 0; i < size; i++)
        h.insertItem(data[i]);

    h.displayHash();

    cout <<"\nDelete element\n";
    h.deleteItem(231);
    h.displayHash();
}
```

Result:

```
table[0] --> 231
table[1]
table[2] --> 212
table[3] --> 262
table[4]
table[5]
table[6] --> 321 --> 321 --> 433

Delete element
table[0]
table[1]
table[2] --> 212
table[3] --> 262
table[4]
table[5]
table[6] --> 321 --> 321 --> 433
```

¿What is the time complexity?

Complexity of Hash tables

Search: O(1+(n/m)**)**

Delete: O(1+(n/m)**)**

where **n** = Total elements in hash table

m = Size of hash table

Here **n/m** is the **Load Factor**. The **load factor** (∝) must be as ***small as possible***. If load factor increases, then possibility of collision increases.

If we assume uniform distribution of keys ,

Expected chain length: O(∝)

Expected time to search: O(1 + ∝)

Expected time to insert / delete: O(1 + ∝)

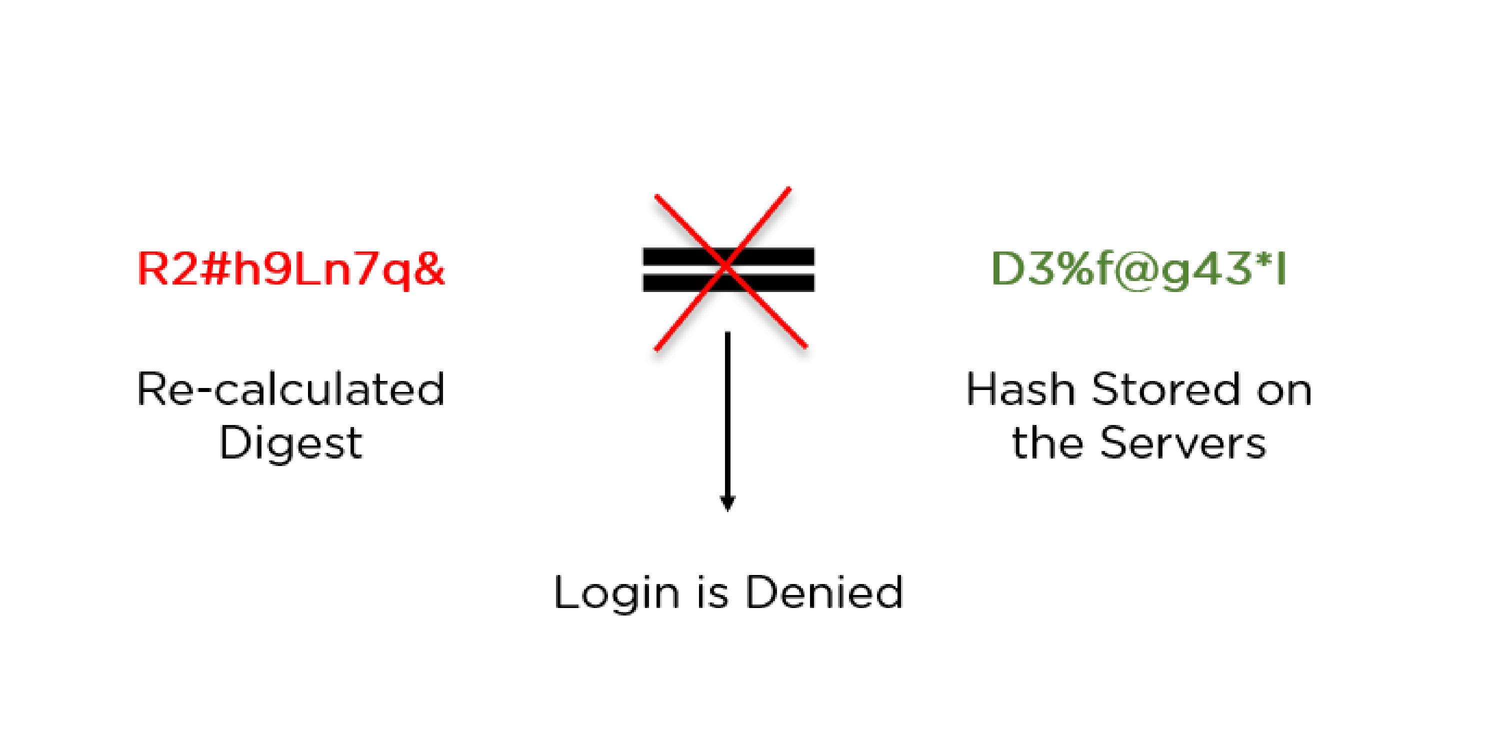
Applications of Hash Table

Hash tables are implemented where:

- Constant time lookup and insertion is required
- Cryptographic applications
- Indexing data is required

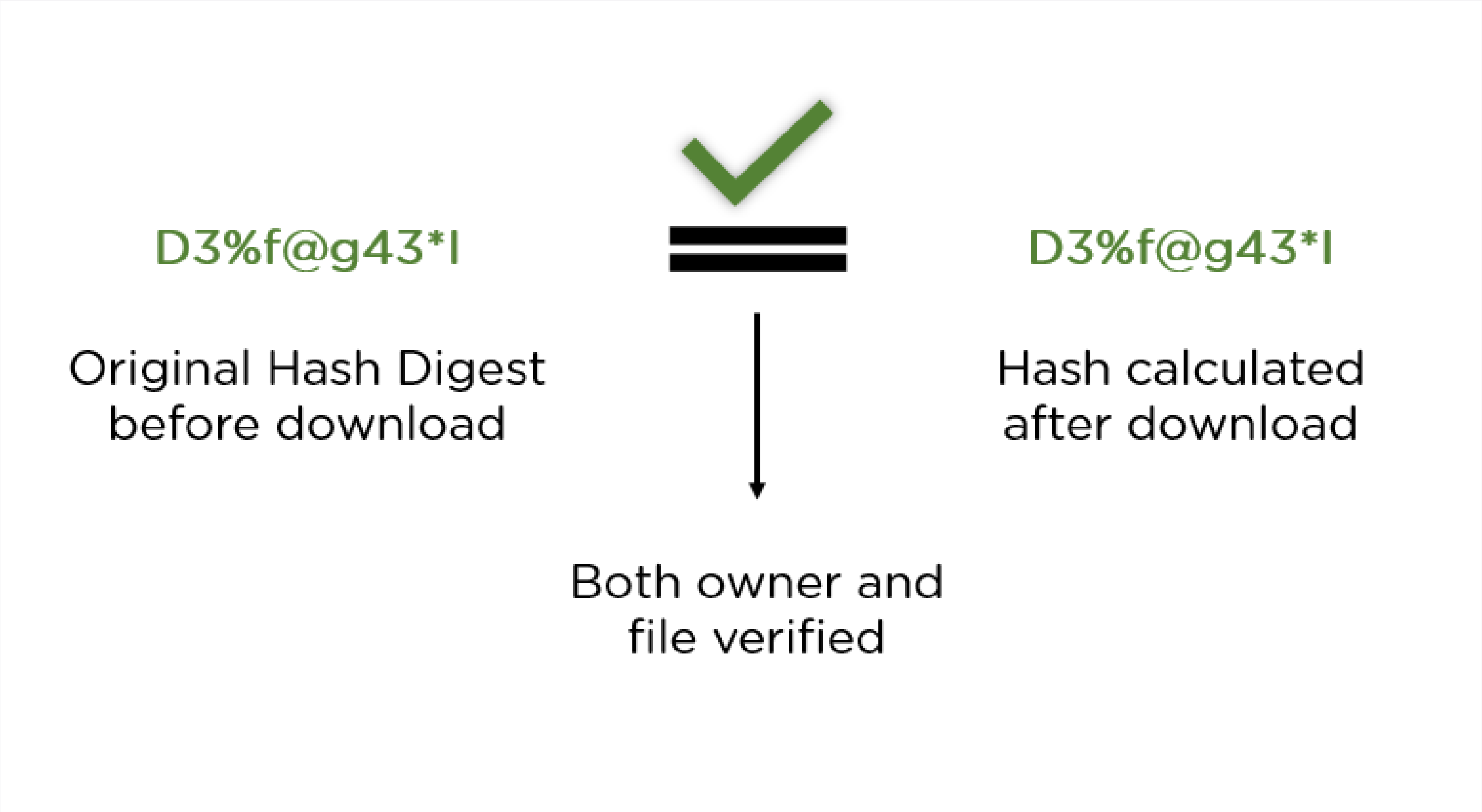
Application: Password Hashes

In most website servers, **it converts user passwords into a hash value before being stored on the server**. It compares the hash value re-calculated during login to the one stored in the database for validation.



Application: Integrity verification

When it uploads a file to a website, it also shared its hash as a bundle. When a user downloads it, **it can recalculate the hash and compare it to establish data integrity**.



Act 5.1
- Individual implementation of operations on sets

All functionalities must be documented. As part of the documentation, the complexity of each of them must be included.

What do I have to do?

Individually, create a **Hash table** class with the variables **table size**, fixed-size **lists** or **int array** (as applicable).

You must propose a hash function (*that avoids collisions as best as possible*) using chaining (list array) and an open address collision resolution: quadratic probing (int array).

Test each test case for each collision resolution method through the **insert()** function.

- **Insert ()**

Input: Integer value

Inserts the element at the calculated position (hash value)

Print the total number of collisions in each test case.

Act 5.2
- Comprehensive activity on the use of hash codes (Competence evidence)

All functionalities must be documented. Carry out an individual investigation and reflection on the importance and efficiency of the use of hash tables for such purposes.

What do we have to do?

In Teams of three, create a **Hash table** class:

- Open the input file "bitacora.txt" and add all the entries in a hash table structure where the **key** will be the **port number** and the **values to save on the hash table** are the **total number of accesses** and the **bitacora entries** that access in that port, similar to the activity 3.4.

In this activity the following is expected:

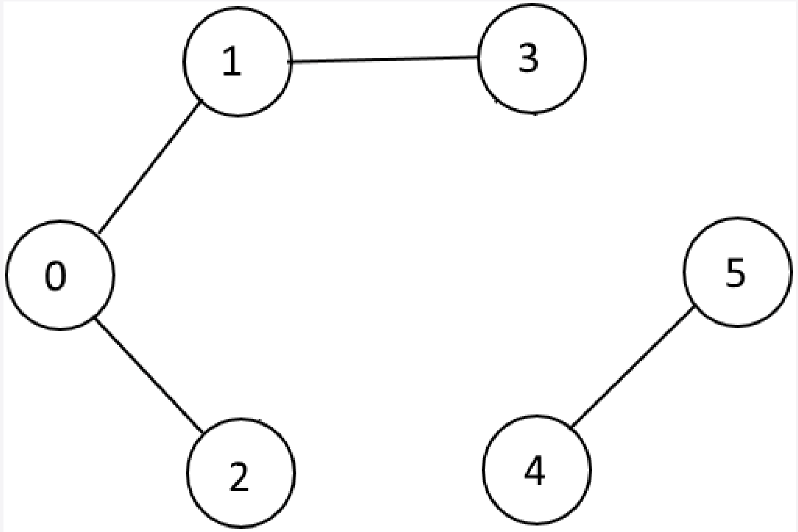
- Investigate and implement a hash function that avoids as many collisions as possible.
- Implement a suitable collision handling method (chaining, open addressing).
- Print the 5 port numbers with the most accesses (already obtained in activity 3.4) in the hash table. The results should match.

***You are not allowed to use C++ STL hash tables**

Disjoint sets

What is a Disjoint set Data structure?

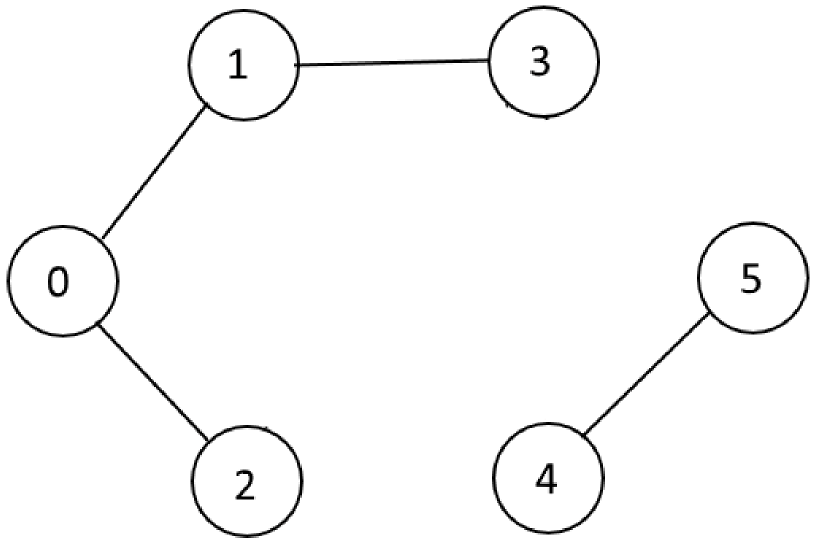
Two sets are called **disjoint sets** if they **don't have any element in common**, the **intersection of sets** is a **null** set.



The **data structure** stores non overlapping or disjoint subset of elements, called **disjoint set data structure**.

How Disjoint Set is constructed?

- A **disjoint-set forest** consists of elements each of which stores an **id**, a **parent pointer**.
- The **parent pointers** of elements are arranged to form **one or more trees**, each representing a **set**.
- If an **element's parent pointer points to no other element**, then the element is **root of the tree** and is **representative member of its set**.
- If the element has a **parent**, the **representative member of the set** is identified by following the chain of parents until an **element without a parent** (root of the tree) is reached. A **set may consist of only a single element**.



Disjoint Sets

Set Id: 0, elements: [0, 1, 2, 3]
Set Id: 4, elements: [4, 5]

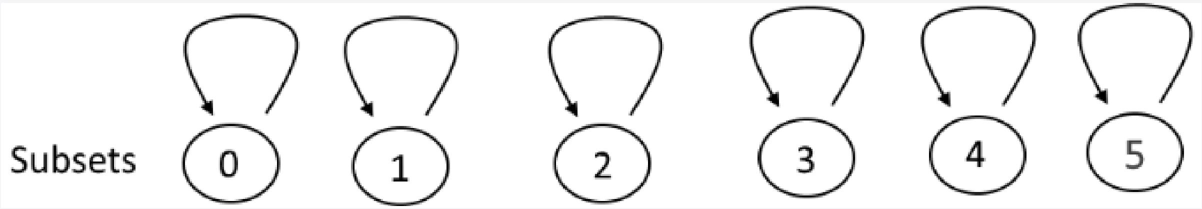
Disjoint set Operations

- **MakeSet(X)**: It creates a new element with a parent pointer to itself. The **parent pointer** **pointing to itself** indicates that the element is the **representative member** of its own set.

- **Find(X)**: follows the chain of **parent pointers** from **x** upwards until an element whose parent is itself. This element is the **representative member** of the set to which **x** belongs and may be **x** itself.

- **Union(x,y)**: Uses **Find()** to determine the **roots of the trees** to which **x** and **y** belong. If the **roots are different**, the **trees are combined** by joining the **root** of one to the **root** of the other.

How Disjoint set works?

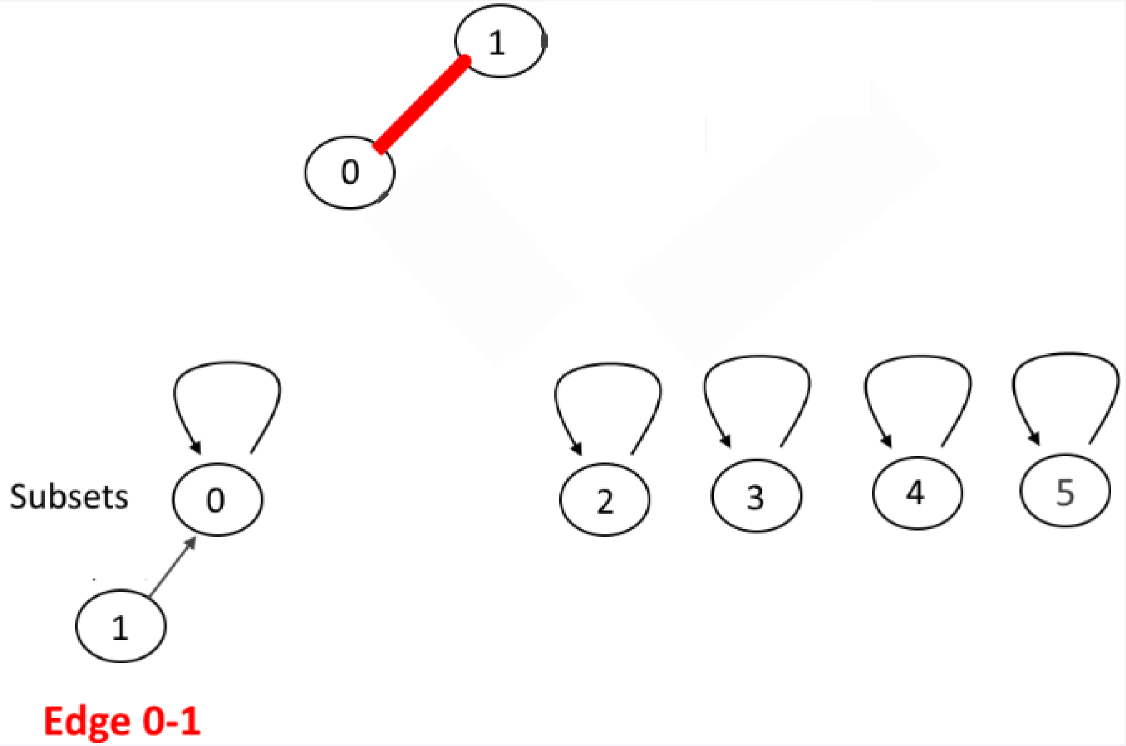


Initially all parent pointers are pointing to self means only one element in each subset

How Disjoint set works?

Find: 0 belongs to subset **0** and **1** belongs to subset **1** so they are in different subsets.

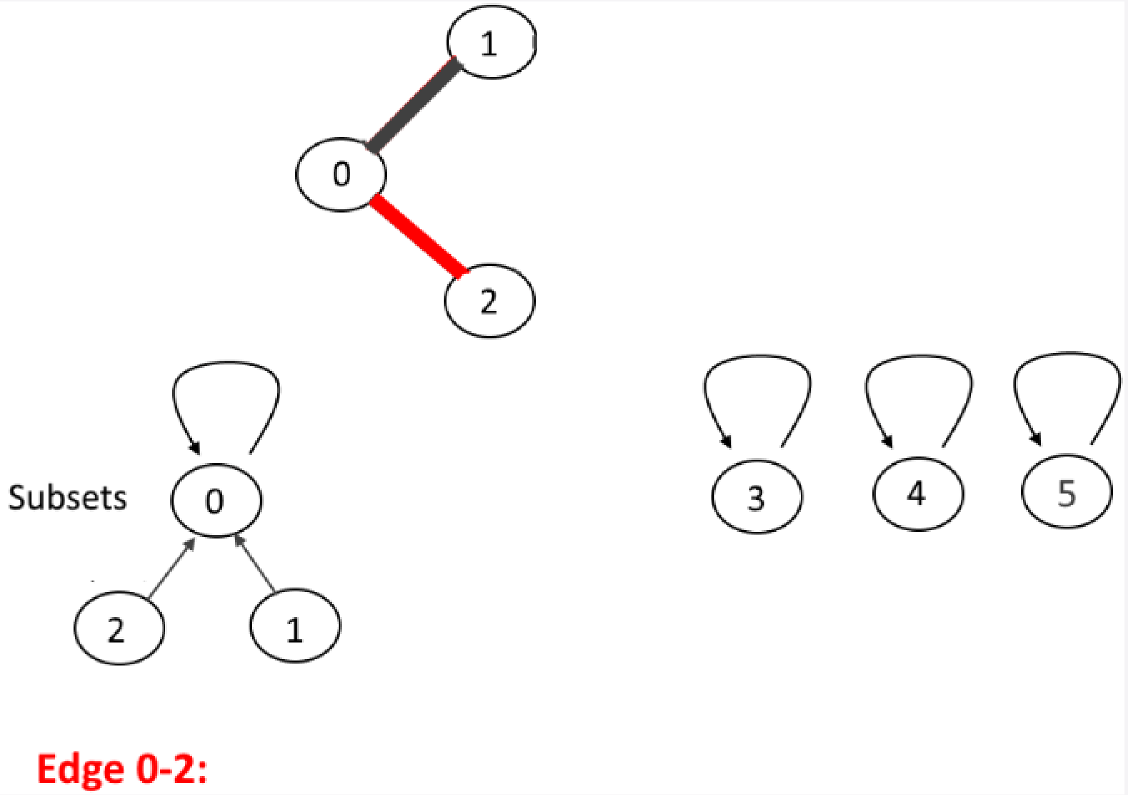
Union: Make **0** as the parent of **1**. Updated set is **{0, 1}**. **0** is the set representative since **0** is parent for itself.



How Disjoint set works?

Find: 0 belongs to subset **0** and **2** belongs to subset **2** so they are in different subsets.

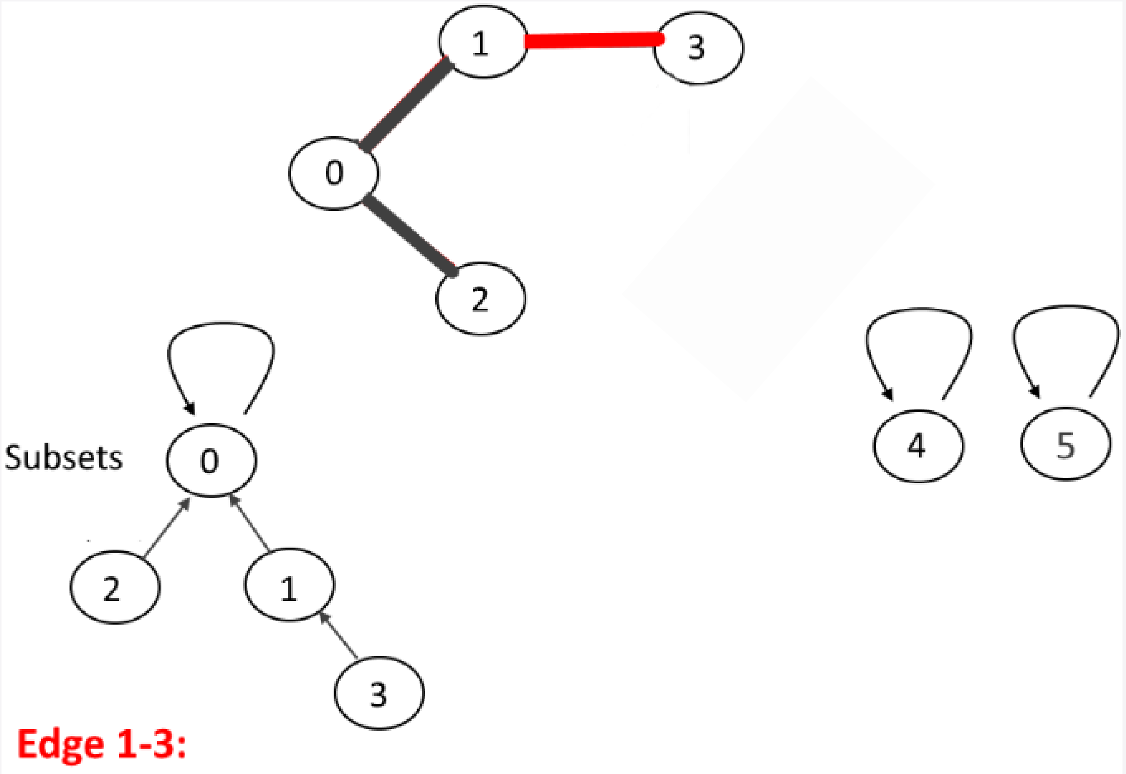
Union: Make **0** as the parent of **2**. Updated set is **{0, 1, 2}**. **0** is the set representative since **0** is parent for itself.



How Disjoint set works?

Find: **1** belongs to subset **0** and **3** belongs to subset **3** so they are in different subsets.

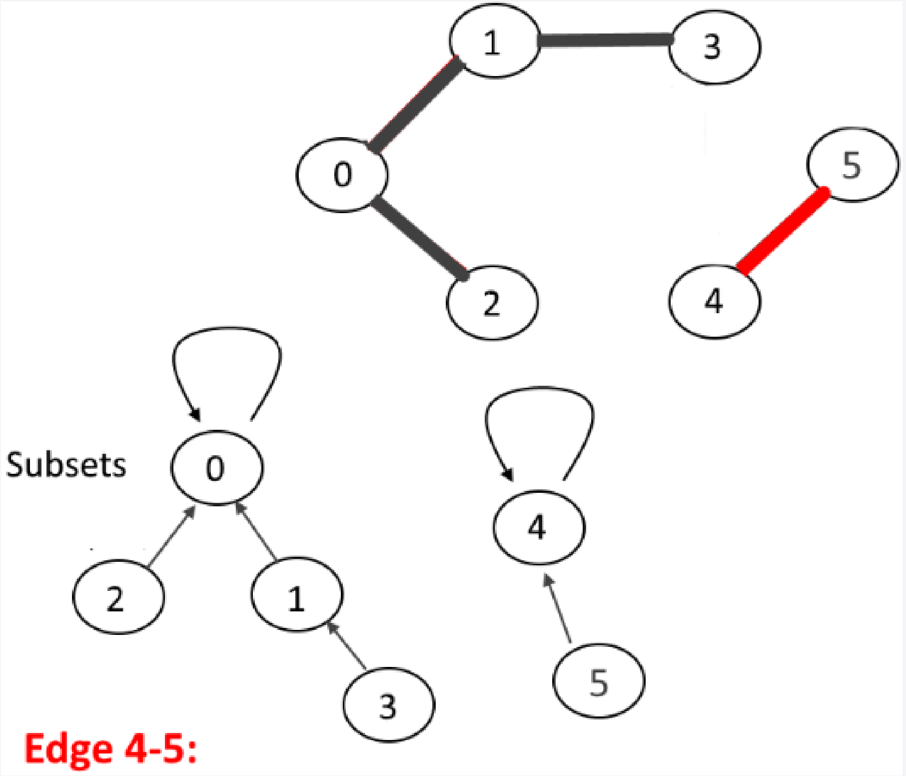
Union: Make **1** as the parent of **3**. Updated set is **{0, 1, 2, 3}**. **0** is the set representative since **0** is parent for itself.



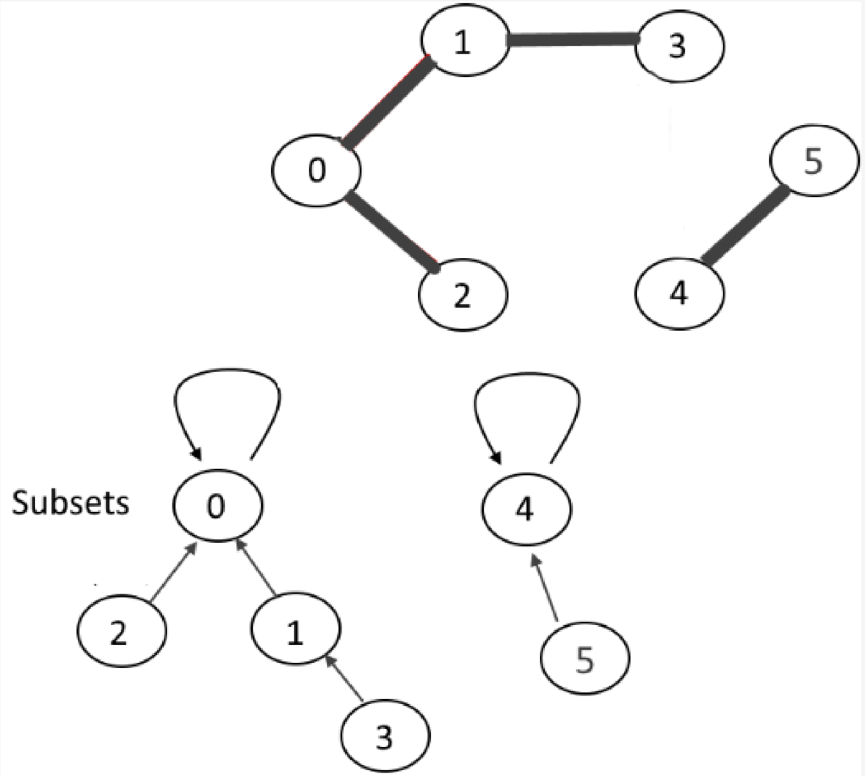
How Disjoint set works?

Find: **4** belongs to subset **4** and **5** belongs to subset **5** so they are in different subsets.

Union: Make **4** as the parent of **5**. Updated set is **{4, 5}**. **4** is the set representative since **4** is parent for itself.



How Disjoint set works?



Set Id: 0**, elements:** [0,1,2,3]

Set Id: 4**, elements:** [4,5]

Implementation

Implement the constructor to create and initialize sets of **n** items

```
void DisjSet::display()
{
    cout << "Parent: ";
    for (int i = 0; i < n; i++)
        cout << parent[i] << " ";
    cout << endl;
}
```

```
class DisjSet
{
    private:
        int *parent, n;

    public:
        DisjSet(int n)
        {
            this->n = n;

            parent = new int[n];

            for (int i = 0; i < n; i++)
                parent[i] = i;
        }
        ~DisjSet()
        {
            delete []parent;
        }

        //Functions
        int find(int);
        void Union(int, int);
        void display();
};
```

Find - Implementation

Finds the representative of the set of the given item **x**.

If **x** is not the **parent of itself** then **x** is **not the representative of his set**, so we recursively call **Find** on its parent and move **x's node** directly under the representative of this set.

```
int DisjSet::find(int x)
{
    if (parent[x] == x)
        return x;

    int xset = find(parent[x]);

    parent[x] = xset;

    return xset;
}
```

Union- Implementation

- **Find** the representatives (or the root nodes) for the set that includes **x**, and do the same for the set that includes **y**.

- Make the parent of **x's** representative be **y's** representative effectively moving all of **x's** set into **y's** set.

- The height of the trees can grow as **O(n)**.

```
void DisjSet::Union(int x, int y)
{
    int xset = find(x);
    int yset = find(y);

    parent[xset] = yset;
}
```

Program body

```
int main()
{
    DisjSet DS(5);
    DS.display();

    DS.Union(0, 2);
    DS.display();

    DS.Union(4, 2);
    DS.display();

    DS.Union(3, 1);
    DS.display();

    if (DS.find(4) == DS.find(0))
        cout << "Yes\n";
    else
        cout << "No\n";

    if (DS.find(1) == DS.find(0))
        cout << "Yes\n";
    else
        cout << "No\n";

    return 0;
}
```

Result:

```
Parent: 0 1 2 3 4
Parent: 2 1 2 3 4
Parent: 2 1 2 3 2
Parent: 2 1 2 1 2
Yes
No
```

¿What is the time complexity?

Applications of Disjoint set Union

Hash tables are implemented where:

- Kruska's Minimum Spanning Tree Algorithm
- Cycle Detection
- Number of pairs

Using sets and hash tables in C++

Hashing in C++ STL

unordered_map is an associated container that stores elements formed by the combination of a **key value** and a **mapped value**. Both **key** and **value** can be of **any type predefined** or **user-defined**.

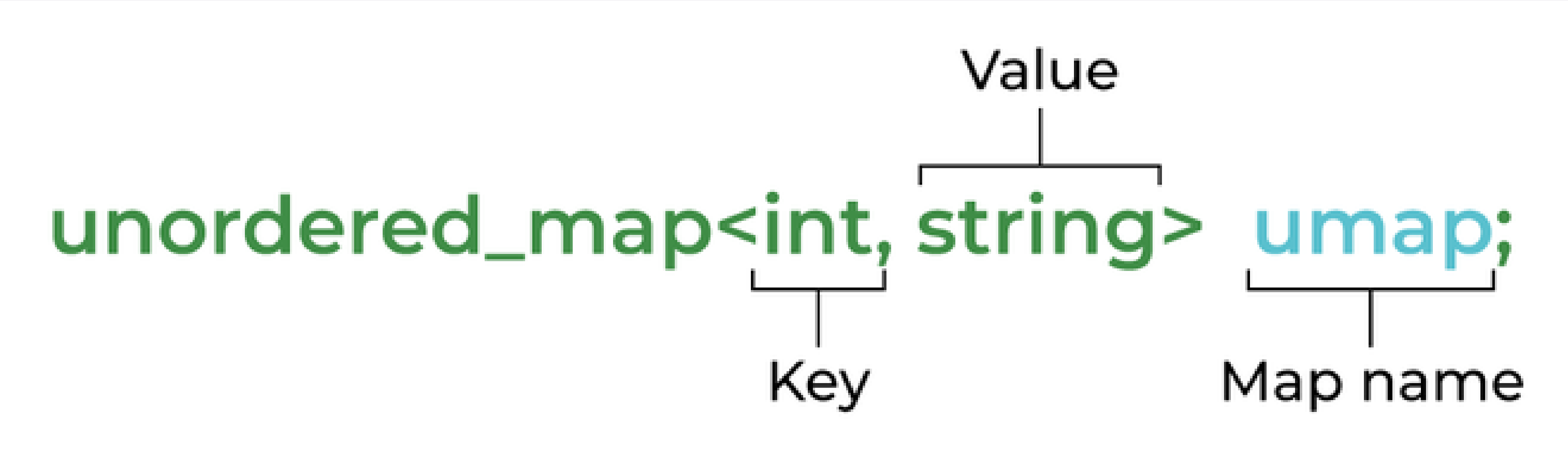
In simple terms, an **unordered_map** is like a data structure of **dictionary type** that stores elements in itself. It contains successive pairs **(key, value)**, which allows fast retrieval of an individual element based on its unique key.

Hashing in C++ STL

Internally **unordered_map** is implemented using **Hash Table**, the key provided to map is hashed into indices of a hash table which is why the performance of data structure depends on the hash function a lot but on average, the cost of **search**, **insert**, and **delete** from the hash table is **O(1)**.

Note: *In the worst case, its time complexity can go from **O(1)** to **O(n)**, especially for big prime numbers. In this situation, it is highly advisable to use a **map** instead to avoid getting a **TLE**(Time Limit Exceeded) **error**.*

Syntax - unordered map



The key value is used to uniquely identify the element and the mapped value is the content associated with the key.

Below is the C++ program to demonstrate an unordered map:

Program body:

```
#include <iostream>
#include <unordered_map>
using namespace std;

// Driver code
int main()
{
    unordered_map<string, int> umap;

    umap["Hashing"] = 10;
    umap["Practice"] = 20;
    umap["Exercise"] = 30;

    for (auto x: umap)
        cout << x.first << " " << x.second << endl;
}
```

Output:

```
Practice 20
Exercise 30
Hashing 10
```

Explanation

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).

```
Practice 20
Exercise 30
Hashing 10
```

unordered_map VS map

Unordered_map	Map
The unordered_map key can be stored in any order.	The map is an ordered sequence of unique keys
Unordered_Map implements an unbalanced tree structure due to which it is not possible to maintain order between the elements	Map implements a balanced tree structure which is why it is possible to maintain order between the elements (by specific tree traversal)
The time complexity of unordered_map operations is O(1) on average.	The time complexity of map operations is O(log n)

unordered_map VS map

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 Sets in C++ STL

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**.

All operations on the **unordered_set** takes constant time **O(1)** on an average which can go up to linear time **O(n)** in worst case which depends on the internally used **hash function**, but practically they perform very well and generally provide a **constant time lookup operation**.

Unordered Sets in C++ STL

The **unordered_set** can contain key of any type – predefined or user-defined data structure.

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.

unordered_set VS set

Set is an **ordered sequence of unique keys** whereas **unordered_set** is a **set** where the **key** can be stored in any order, so unordered.

Internally, the elements in the **unordered_map** 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 key values.

unordered_set VS set

Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal).

The time complexity of **set** operations is **O(log n)** while for **unordered_set**, it is **O(1)**.

Program body:

```cpp
#include <iostream>
#include <unordered_set>
using namespace std;

int main()
{
    unordered_set <string> stringSet ;

    stringSet.insert("code") ;
    stringSet.insert("in") ;
    stringSet.insert("c++") ;
    stringSet.insert("is") ;
    stringSet.insert("fast") ;

    string key = "slow" ;

    if (stringSet.find(key) == stringSet.end())
        cout << key << " not found" << endl << endl ;
    else
        cout << "Found " << key << endl << endl ;

    key = "c++";
    if (stringSet.find(key) == stringSet.end())
        cout << key << " not found\n" ;
    else
        cout << "Found " << key << endl ;

    cout << "\nAll elements : ";
    unordered_set<string> :: iterator itr;
    for (itr = stringSet.begin(); itr != stringSet.end(); itr++)
        cout << (*itr) << endl;
}
```

Output:

```
slow not found

Found c++

All elements : fast
is
c++
in
code
```

unordered_map VS unordered_set

Unordered_map	Unordered_set
Unordered_map contains elements only in the form of (key-value) pairs.	Unordered_set does not necessarily contain elements in the form of key-value pairs, these are mainly used to see the presence/absence of a set.
Operator '**[]**' to extract the corresponding value of a key that is present in the map.	The searching for an element is done using a **find**() function. So no need for an operator[].

Note: For example, consider the problem of counting the frequencies of individual words. We can't use **unordered_set** (or **set**) as we can't store counts while we can use **unordered_map**.

A practical problem

Based on **unordered_set** – Given an **array** of **integers**, find all the duplicates among them.

Program body:

```cpp
void printDuplicates(int arr[], int n)
{
    unordered_set<int> intSet;
    unordered_set<int> duplicate;

    for (int i = 0; i < n; i++)
    {
        if (intSet.find(arr[i]) == intSet.end())
            intSet.insert(arr[i]);
        else
            duplicate.insert(arr[i]);
    }

    cout << "Duplicate item are : ";
    unordered_set<int> :: iterator itr;

    for (itr = duplicate.begin(); itr != duplicate.end(); itr++)
        cout << *itr << " ";
}
```

```cpp
int main()
{
    int arr[] = {1, 5, 2, 1, 4, 3, 1, 7, 2, 8, 9, 5};
    int n = sizeof(arr) / sizeof(int);

    printDuplicates(arr, n);
    return 0;
}
```

Output:

```
Duplicate item are : 5 2 1
```