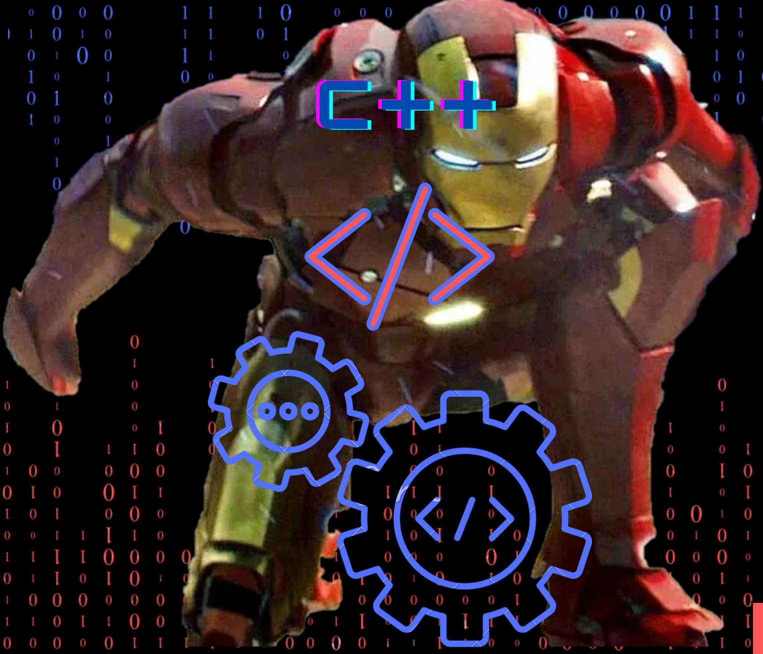
2023

Java

# DSA
# PART-6

# OBZTRON

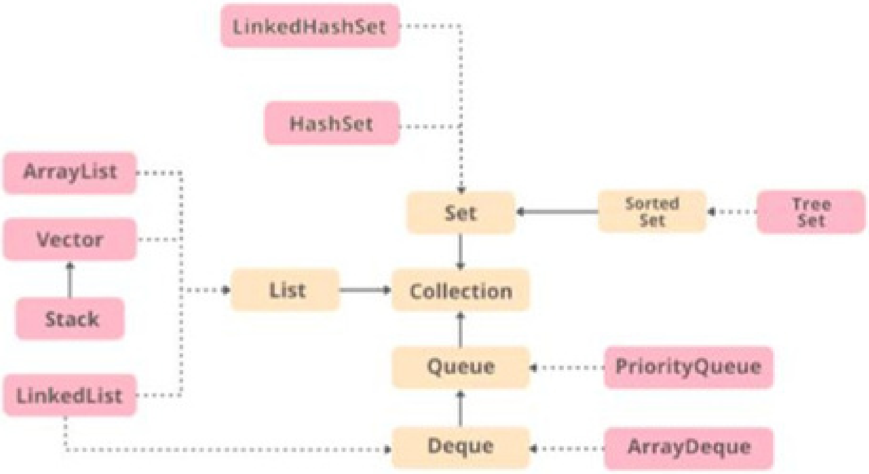


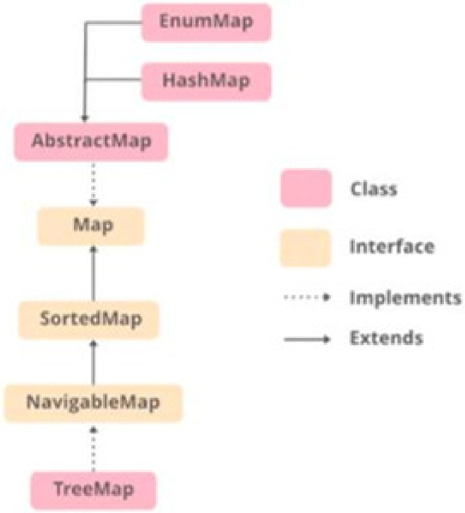
(SIXTH PART)

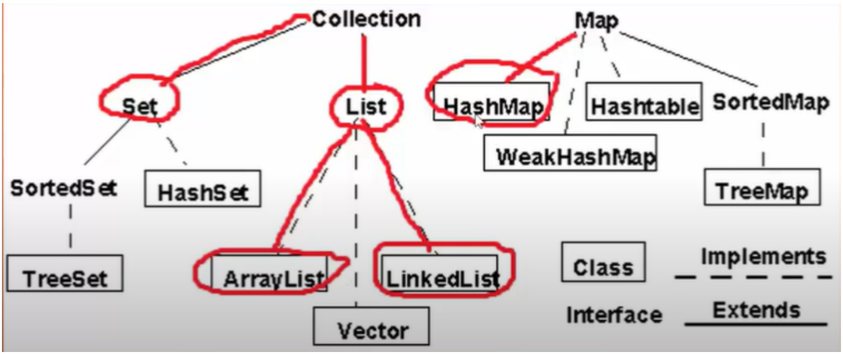
**T.I.M. HAMIM**

# Index: DSA < Part - 6 >

# 1. Collections Framework



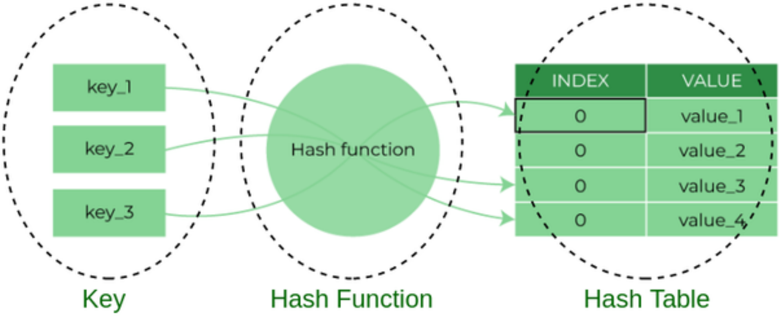



Collection
Map
Set
List
HashMap
Hashtable
SortedMap
WeakHashMap
SortedSet
HashSet
TreeMap
TreeSet
ArrayList
LinkedList
Class
Implements
Interface
Extends
Vector

## 2. Hashing Technique

### Definition of Hashing:

Hashing refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.



| INDEX | VALUE |
|---|---|
| 0 | value_1 |
| 0 | value_2 |
| 0 | value_3 |
| 0 | value_4 |

**Components of Hashing**

There are majorly three components of hashing:

1. **Key:** A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The hash function receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index.
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.

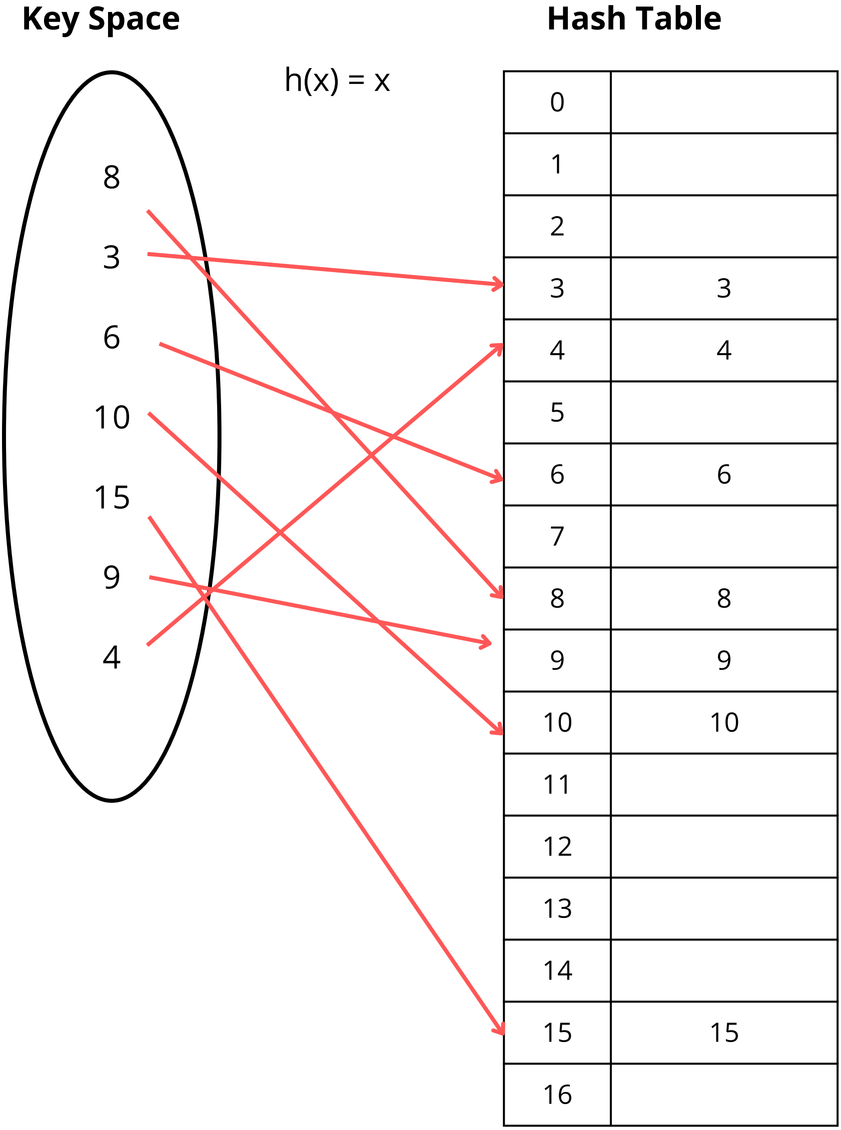
**Mappings:**

Functions:

- One-One
- One-many
- many-One
- many-many

- **Ideal function Hashing :**

One-One function.



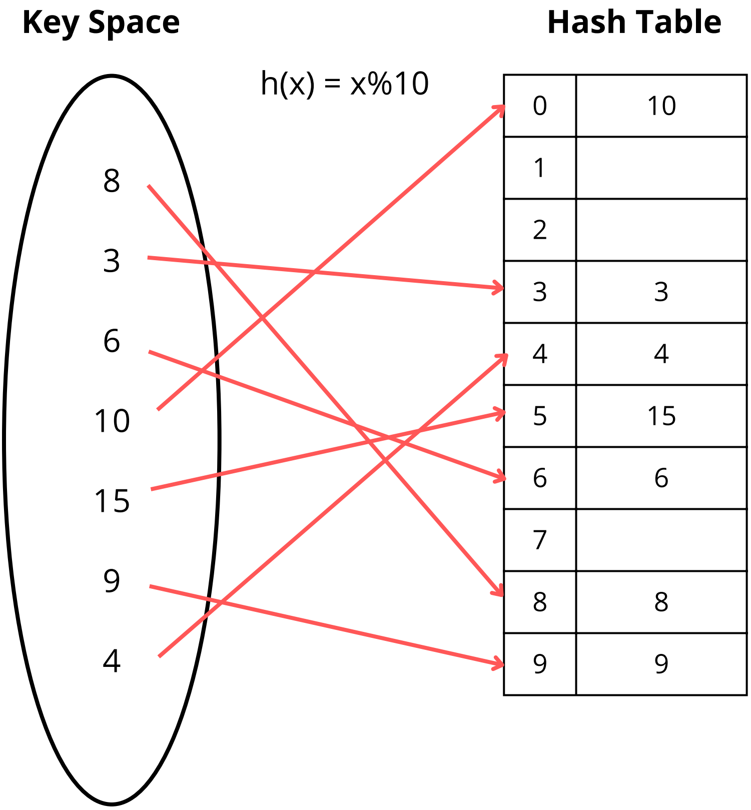
| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | 3 |
| 4 | 4 |
| 5 | |
| 6 | 6 |
| 7 | |
| 8 | 8 |
| 9 | 9 |
| 10 | 10 |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | 15 |
| 16 | |

**Drawback of Ideal function Hashing:**

It can takes huge space.

- **Modules function Hashing :**

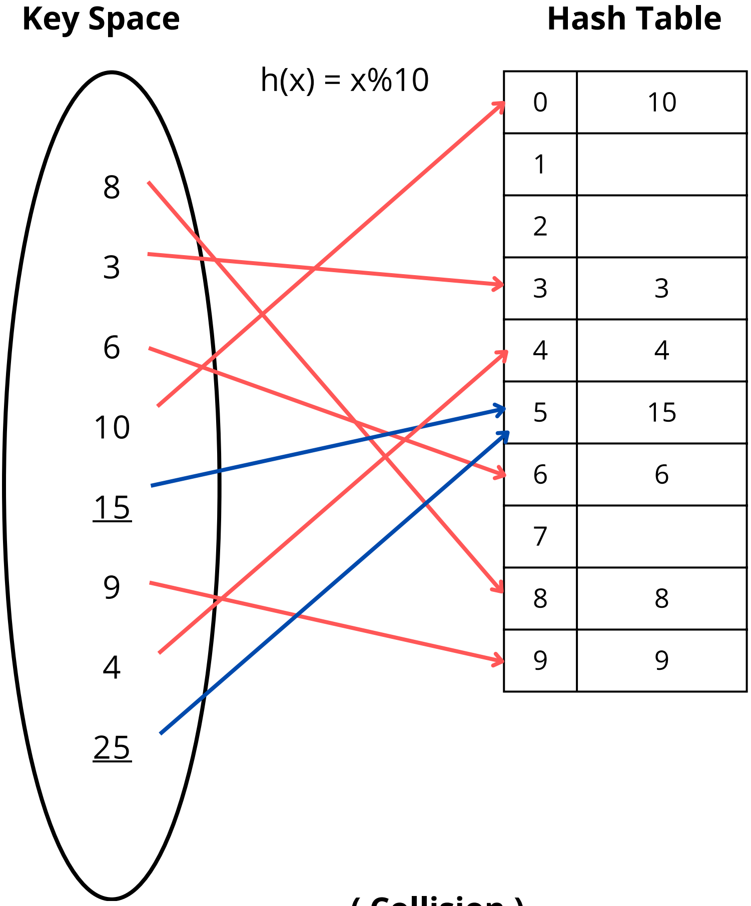
One-One / many-One   function.



| | |
|---|---|
| 0 | 10 |
| 1 | |
| 2 | |
| 3 | 3 |
| 4 | 4 |
| 5 | 15 |
| 6 | 6 |
| 7 | |
| 8 | 8 |
| 9 | 9 |

**Drawback of Modules function Hashing:**

It creates Collision.

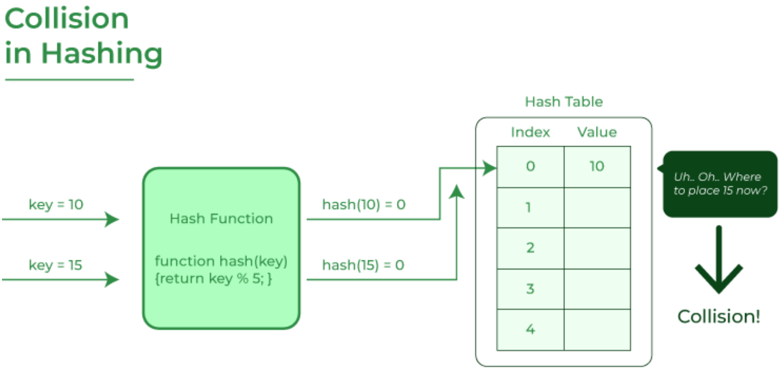


| | |
|---|---|
| 0 | 10 |
| 1 | |
| 2 | |
| 3 | 3 |
| 4 | 4 |
| 5 | 15 |
| 6 | 6 |
| 7 | |
| 8 | 8 |
| 9 | 9 |

**( Collision )**

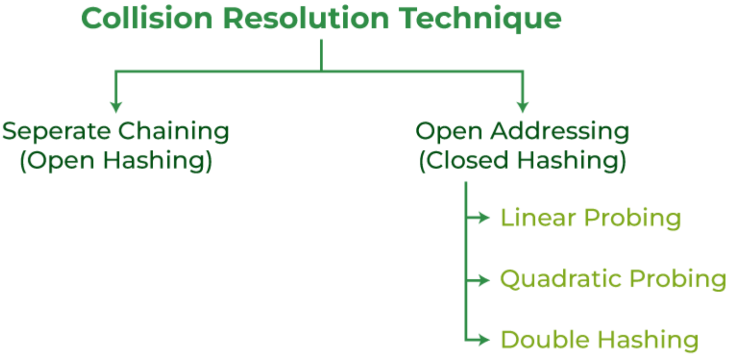
**collision:**

The hashing process generates a small number for a big key, so there is a possibility that two keys could produce the same value. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



There are mainly two methods to handle collision:

1. Open Hashing / Separate Chaining
2. Closed Hashing / Open Addressing

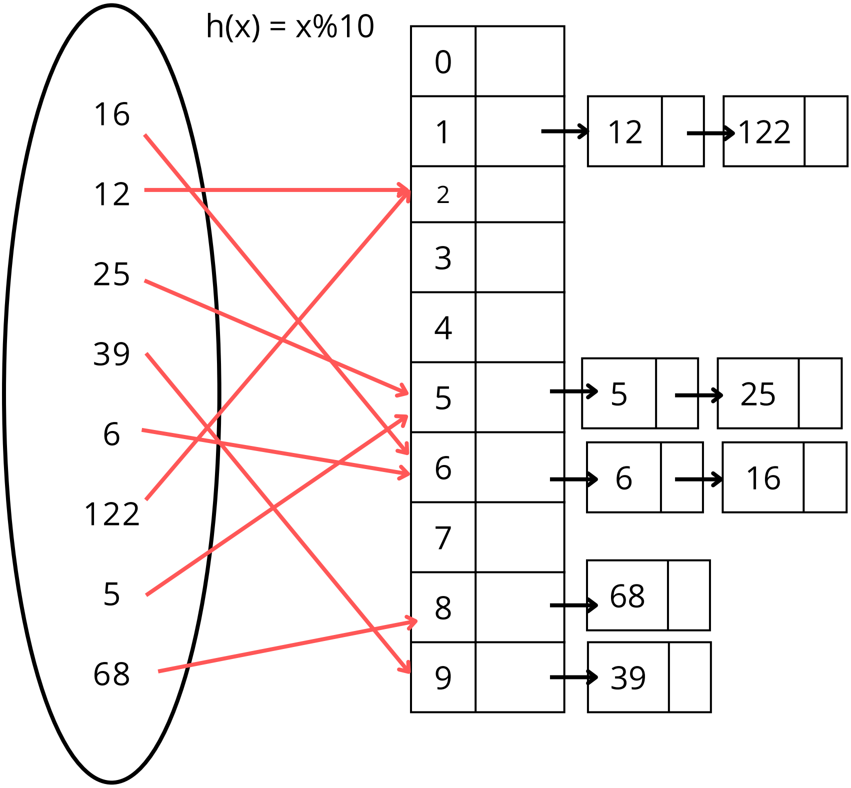

Collision Resolution Technique
Seperate Chaining (Open Hashing)
Open Addressing (Closed Hashing)
Linear Probing
Quadratic Probing
Double Hashing

## 1. Open Hashing / Separate Chaining:

The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

**Key Space** **Hash Table**

Number of keys = **n** Hash Table Size = **m**



**Loading factor, λ = n / m**

If $n = 100$, $m = 10$

$\lambda = 100 / 10 = 10$

**Average Successful Search time :**

$t = 1 + \lambda/2$

**Average Unsuccessful Seach time :**

$t = 1 + \lambda$

**Drawback of Open Hashing:**

The loading factor will be broken if all the key modules are the same.

## 2. Closed Hashing / Open Addressing:

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 examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

Three types of open addressing:

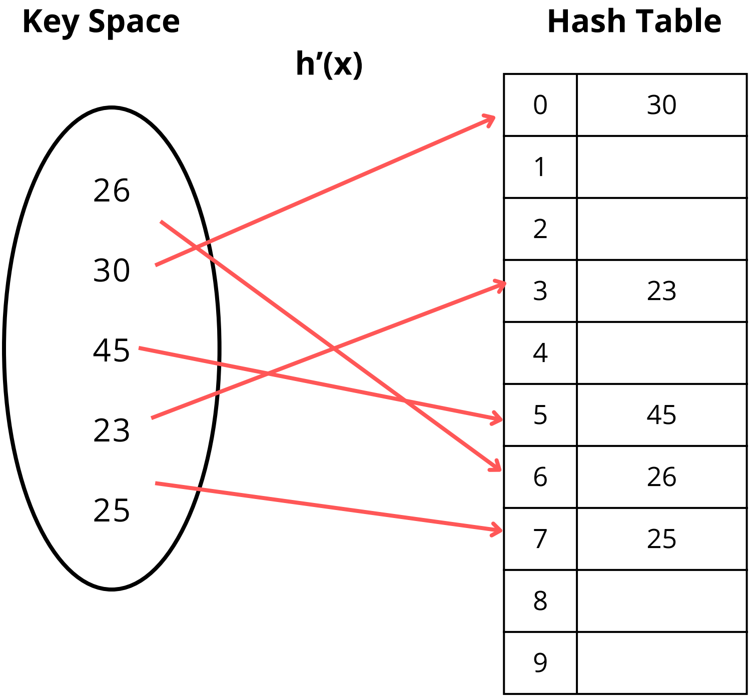
- a) Linear Probing
- b) Quadratic Probing
- c) Double Hashing

### a) Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

**h'(x)** = (h(x) + f(i)) % 10

where h(x) = x%10, f(i) = i ( i = 0, 1, 2, 3, .....)



| | |
|---|---|
| 0 | 30 |
| 1 | |
| 2 | |
| 3 | 23 |
| 4 | |
| 5 | 45 |
| 6 | 26 |
| 7 | 25 |
| 8 | |
| 9 | |

Loading factor has to be **λ<=0.5**

Here,
h'(x) = (h(x) + f(i)) % 10

h'(25) = (h(25) + f(0)) % 10 ; ( i = 0)
= (5 + 0) % 10 = 5
h'(25) = (h(25) + f(0)) % 10 ; ( i = 1)
= (5 + 1) % 10 = 6
h'(25) = (h(25) + f(0)) % 10 ; ( i = 2)
= (5 + 2) % 10 = 7

In this method, at the time of searching, we will stop searching when we will find an empty space.

The loading factor has to be **λ<=0.5**

**Average successful search time:**
t = (1/ λ) . ln(1 / (1-λ))

**Average unsuccessful search time:**
t = 1 / (1-λ)

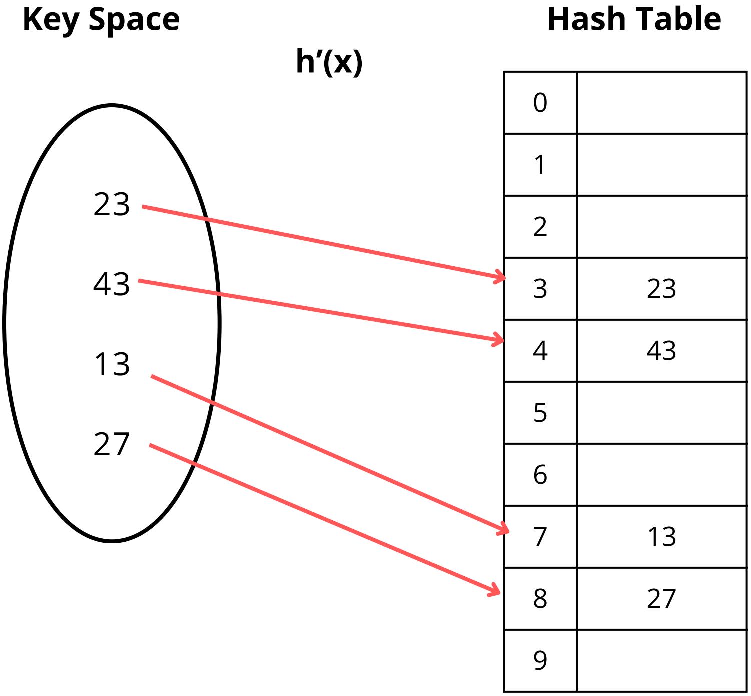
**Drawback of Linear Probing:**
The loading factor has to be λ<=0.5 for this reason we need extra space and sometimes it may create cluster (A group of elements were forming a single block) .

### b) Quadratic Probing:

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

$$\mathbf{h'(x)} = (h(x) + f(i)) \% 10$$

where $h(x) = x\%10$, $f(i) = i\text{^}2$ ( $i = 0, 1, 2, 3, \ldots..$)



| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | 23 |
| 4 | 43 |
| 5 | |
| 6 | |
| 7 | 13 |
| 8 | 27 |
| 9 | |

Loading factor has to be **λ<=0.5**

Here,
$h'(x) = (h(x) + f(i)) \% 10$

$h'(23) = (h(23) + f(0)) \% 10 \quad ; ( i = 0)$
$\quad = (3 + 0) \% 10 = 3 \quad ; f(i) = 0^2 = 0$

$h'(43) = (h(43) + f(0)) \% 10 \quad ; ( i = 0)$
$\quad = (3 + 0) \% 10 = 3 \quad ; f(i) = 0^2 = 0$
$h'(43) = (h(43) + f(0)) \% 10 \quad ; ( i = 1)$
$\quad = (3 + 1) \% 10 = 4 \quad ; f(i) = 1^2 = 1$

$h'(13) = (h(13) + f(0)) \% 10 \quad ; ( i = 1)$
$\quad = (3 + 0) \% 10 = 3 \quad ; f(i) = 0^2 = 0$
$h'(13) = (h(13) + f(0)) \% 10 \quad ; ( i = 1)$
$\quad = (3 + 1) \% 10 = 4 \quad ; f(i) = 1^2 = 1$
$h'(13) = (h(13) + f(0)) \% 10 \quad ; ( i = 1)$
$\quad = (3 + 4) \% 10 = 7 \quad ; f(i) = 2^2 = 4$

$h'(27) = (h(27) + f(0)) \% 10 \quad ; ( i = 0)$
$\quad = (7 + 0) \% 10 = 7 \quad ; f(i) = 0^2 = 0$
$h'(27) = (h(27) + f(0)) \% 10 \quad ; ( i = 1)$
$\quad = (7 + 1) \% 10 = 8 \quad ; f(i) = 1^2 = 1$

In this method, at the time of searching, we will stop searching when we will find an empty space. The loading factor has to be **$\lambda <= 0.5$**

**Average successful search time:**
$t = - \ln(1-\lambda) / \lambda$

**Average unsuccessful search time:**
$t = 1 / (1-\lambda)$

### c) Double Hashing:

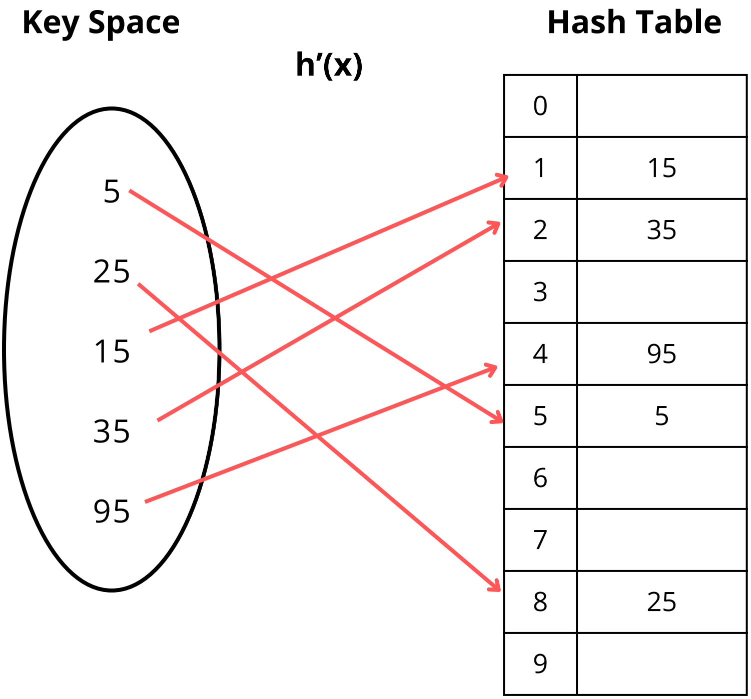
Double hashing is a collision resolving technique in <u>Open Addressed</u> Hash tables. Double hashing make use of two hash function

$h1(x) = x\%10$

$h2(x) = p - (x\%p)$ ; $p$ = hash table size closest prime number

**$h'(x)$** $= (h1(x) + i * h2(x)) \% 10$

where $i = 0, 1, 2, 3, \ldots..m$



| | |
|---|---|
| 0 | |
| 1 | 15 |
| 2 | 35 |
| 3 | |
| 4 | 95 |
| 5 | 5 |
| 6 | |
| 7 | |
| 8 | 25 |
| 9 | |

Loading factor has to be **λ<=0.5**

Here,

h1(x) = x%10
h2(x) = 7 - (x%7)
h’(x) = ( h1(x) + i*h2(x) ) % 10

h’(5) = ( h1(5) + 0*h2(5) ) % 10
= (5 + 0*2) % 10 = 5

h’(25) = ( h1(25) + 1*h2(25) ) % 10
= (5 + 1*3) % 10 = 8

h’(15) = ( h1(15) + 1*h2(15) ) % 10
= (5 + 1*6) % 10 = 1

h’(35) = ( h1(35) + 1*h2(35) ) % 10
= (5 + 1*7) % 10 = 2

h’(95) = ( h1(95) + 3*h2(95) ) % 10
= (5 + 3*3) % 10 = 4

**Hash Functions:**

- Mod
- Midsquare
- Folding

- **Mod:**

h(x) = (x % size of the Table(m) ) + 1 = index
where size of the Table m has to be prime number.

- **Midsquare:**

key = 11 = 11^2 = 121 = 2 no. index
key = 13 = 13^2 = 169 = 6 no. index

- **Folding:**

key = 123347

```
   12
   33
 + 47
  92 no. index
```

## 3. HashSet

Java HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the hash sets which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

HashSet Time complexity:

- Insert/ Add --> O(1)
- Search/ Contains --> O(1)
- Delete/ Remove --> O(1)

- **Example of basic HashSet functions:**

```
import java.util.HashSet;
import java.util.Iterator;

public class Test {
  public static void main(String[] args) {
    HashSet<Integer> set = new HashSet<>();

    set.add(1);
    set.add(2);
    set.add(3);
    set.add(1); // set does not add duplicate element

    System.out.println("size of set is : " + set.size());

    System.out.println("set elements : "+set);

    // Iterator
    Iterator it = set.iterator();

    while ((it.hasNext())) {
      System.out.println(it.next());
    }

    if(set.contains(1)){
      System.out.println("set contains 1");
    }
    if(!set.contains(6)){
      System.out.println("set does not contain 6");
    }
    set.remove(1);
    if(!set.contains(1)){
      System.out.println("set does not contain 1 - we deleted 1");
    }
  }
}
```

**Output:**

```
size of set is : 3
set elements : [1, 2, 3]
1
2
3
set contains 1
set does not contain 6
set does not contain 1 - we deleted 1
```

## 4. HashMap

In Java, HashMap is a part of Java's collection since Java 1.2. This class is found in java.util package. It provides the basic implementation of the Map interface of Java. HashMap in Java stores the data in (Key, Value) pairs, and you can access them by an index of another type (e.g. an Integer). One object is used as a key (index) to another object (value). If you try to insert the duplicate key in HashMap, it will replace the element of the corresponding key.

Java HashMap is similar to HashTable, but it is unsynchronized. It allows to store the null keys as well, but there should be only one null key object and there can be any number of null values. This class makes no guarantees as to the order of the map. To use this class and its methods, you need to import java.util.HashMap package or its superclass.

**Parameters:**
It takes pairs of two parameters namely as follows:

1. **Key:** The type of keys maintained by this map and keys must be unique.
2. **Value:** The type of mapped values.

| Key | Value |
|---|---|
| **roll no** | **name** |
| 64 | Hamim |
| 65 | Jim |
| 66 | Hridi |
| 78 | Surovi |
| 74 | Mim |

- **Example of basic HashMap functions:**

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class Test {
    public static void main(String[] args) {
        // contry(key), population(value)
        HashMap<String, Integer> map = new
HashMap<>();

        map.put("Bangladesh", 120);
        map.put("US", 30);
        map.put("China", 150);

        // map prints its elements unorderly
because it's elements are in unorderd
        System.out.println(map);

        System.out.println(map.size());

        //foreach loop
        for(Map.Entry<String, Integer> e :
map.entrySet()){ // entrySet() --> convert Map
into Set
            System.out.println(e.getKey() + " " +
e.getValue());
        }

        map.put("China", 180); //update value
automatically
        System.out.println(map);
        if(map.containsKey("China")){
            System.out.println("China key is present
in the map");
        } else {
            System.out.println("China key is not
present in the map");
        }

        System.out.println(map.get("China"));
        System.out.println(map.get("India"));

        Set<String> keys = map.keySet();
        // printing keys
        for(String key : keys){
            System.out.println(key+ " "+
map.get(key));
        }

        map.remove("China");
        System.out.println("After removing China
= "+map);
    }
}
```

**Output:**

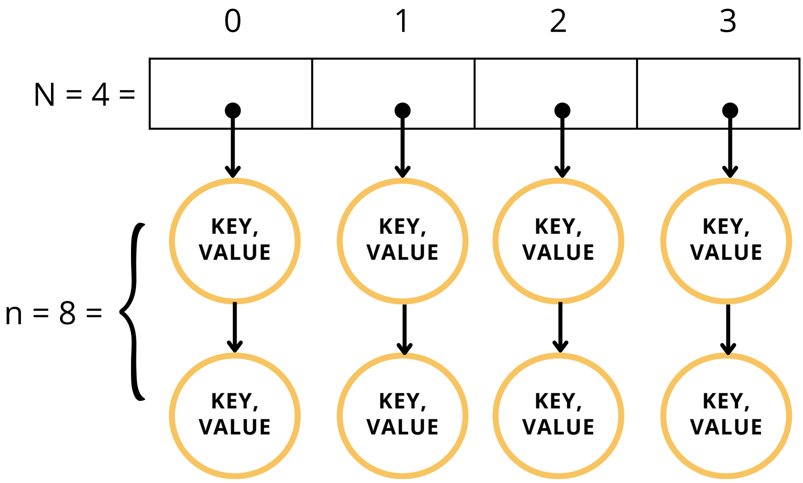
```
{Bangladesh=120, China=150, US=30}
3
Bangladesh 120
China 150
US 30
{Bangladesh=120, China=180, US=30}
China key is present in the map
180
null
Bangladesh 120
China 180
US 30
After removing China = {Bangladesh=120, US=30}
```

- **Implementation of HashMap :**

HashMap functions: O(1)

- put()
- get()
- containsKey()
- remove()
- size()
- keySet()

<u>Array of LinkedList:</u>



**$n/m = \lambda <= K$ (constant/ trash-hold value)**

If, $K = 2$

$4/8 = 2 = \lambda <= k\ (2)$

If, $\lambda > k$ then double the array size.

```
import java.util.*;
import java.util.ArrayList;

public class Test {
    static class HashMap<K,V> {
        private class Node {
            K key;
            V value;

            public Node(K key, V value){
                this.key = key;
                this.value = value;
            }
        }

        private int n; // n - nodes
        private int N; // N - buckets

        // Arrays of LinkedList
        private LinkedList<Node> buckets[]; // N =
buckets.length

        @SuppressWarnings("unchecked")
        public HashMap(){
            this.N = 4;
            this.buckets = new LinkedList[4];
            for(int i=0; i<4; i++){
                this.buckets[i] = new LinkedList<>();
            }
        }

        private int hashFunction(K key){ // bi =
buckets index, di = data/node index
```

```
            int bi = key.hashCode();  // hashCode()
could return + or - hashcode value
            return Math.abs(bi) % N;  // 0 to N-1
        }

        private int searchInLL(K key, int bi){
            LinkedList<Node> LL = buckets[bi];

            for(int i=0; i<LL.size(); i++){
                if(LL.get(i).key == key){
                    return i; // di
                }
            }

            return -1;
        }

        @SuppressWarnings("unchecked")
        private void rehash(){
            LinkedList<Node> oldBucket[] =
buckets;
            buckets = new LinkedList[N*2];

            for(int i=0; i<N*2; i++){
                buckets[i] = new LinkedList<>();
            }

            for(int i=0; i<oldBucket.length; i++){
                LinkedList<Node> LL = oldBucket[i];
                for(int j=0; j<LL.size(); j++){
                    Node node = LL.get(j);
                    put(node.key, node.value);
                }
            }
        }
        public void put(K key, V value){
            int bi = hashFunction(key);
            int di = searchInLL(key, bi); //di = -1

            if(di == -1){ // key doesn't exist
                buckets[bi].add(new Node(key,
value));
                n++;
            } else { //key exists
                Node node = buckets[bi].get(di);
                node.value = value;
            }

            double lamda = (double)n/N;
            if(lamda > 2.0){
                rehash();
            }
        }

        public V get(K key){
            int bi = hashFunction(key);
            int di = searchInLL(key, bi); //di = -1

            if(di == -1){ // key doesn't exist
                return null;
            } else { //key exists
                Node node = buckets[bi].get(di);
                return node.value;
            }
        }

        public boolean containsKey(K key){
            int bi = hashFunction(key);
            int di = searchInLL(key, bi); //di = -1
            if(di == -1){ // key doesn't exist
                return false;
            } else { //key exists
                return true;
            }
        }

        public V remove(K key){
            int bi = hashFunction(key);
            int di = searchInLL(key, bi); //di = -1

            if(di == -1){ // key doesn't exist
                return null;
            } else { //key exists
                Node node = buckets[bi].remove(di);
                n--;
                return node.value;
            }
        }

        public boolean  isEmpty(){
            return n == 0;
        }

        public ArrayList<K> keySet(){
            ArrayList<K> keys = new ArrayList<>();

            for(int i=0; i<buckets.length; i++){
                LinkedList<Node> LL = buckets[i];
                for(int j=0; j<LL.size(); j++){
                    Node node = LL.get(j);
                    keys.add(node.key);
                }
            }
```

```
            return keys;
        }
    }

    public static void main(String[] args) {
        HashMap<String, Integer> map = new
HashMap<>();
        map.put("Bangladesh", 190);
        map.put("China", 200);
        map.put("US", 50);

        ArrayList<String> keys = map.keySet();
        for(int i=0; i<keys.size(); i++){
            System.out.println(keys.get(i)+"
"+map.get(keys.get(i)));
        }

        map.remove("Bangladesh");
        System.out.println(map.get("Bangladesh"));
    }
}
```

**Output:**

```
Bangladesh 190
US 50
China 200
null
```

## Operator Precedence and Associativity:

| | | |
|---|---|---|
| | | |
| | | |
| | | |

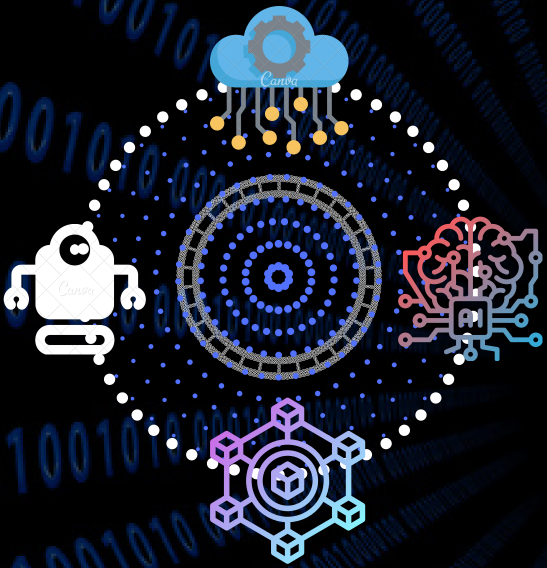
Hamim

Jim

✓ ☒

→

- **What is the output of the following C program fragment:**



C++ LANGUAGE
(FIRST PART)
T.I.M. HA-MEAM

ABC
PROKASHONI