Introduction to Hashing

In all search techniques, the time required to search an element depends on the total number of elements.

Hashing is another approach in which time required to search an element doesn't depend on the total number of elements.

Using hashing data structure, a given element is searched with constant time complexity.

Definition

Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key.

Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.

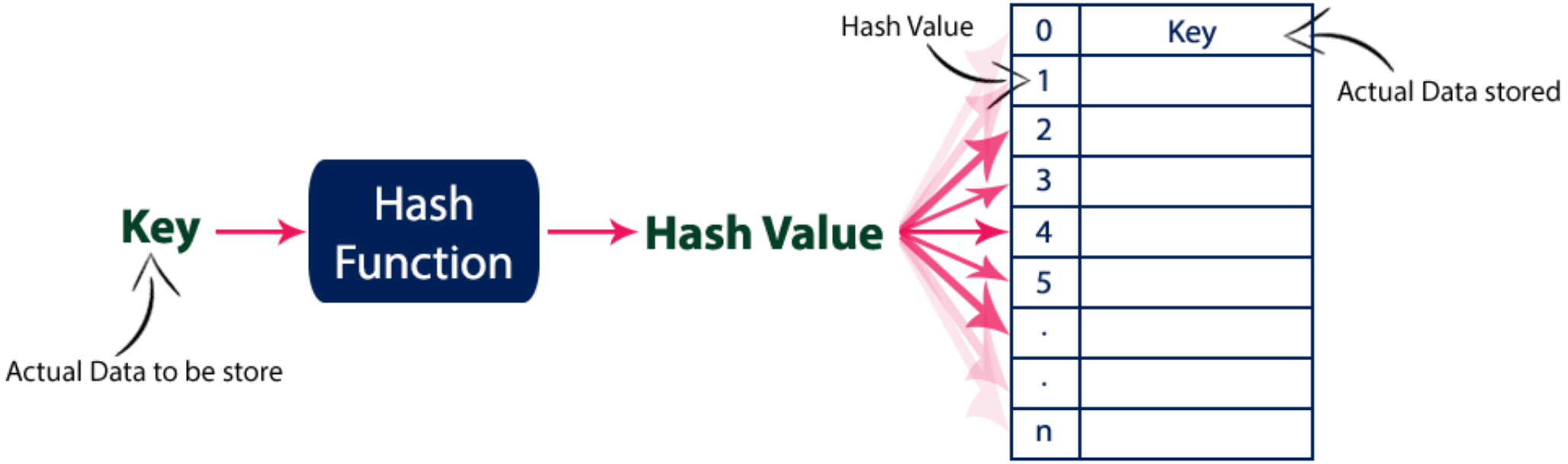
Hash Table

Hash Table is a data structure which stores data in an associative manner.

- All the data values are inserted into the hash table based on the hash key value.
- The hash key value is used to map the data with an index in the hash table.
- And the hash key is generated for every data using a hash function.

Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table



Hashing

hash key = key % number of slots in the table

Assume a table with 8 slots:

Hash key = key % table size

4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

6 = 6 % 8

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Hashing – Creating Hash table

```
int[] arr;
int capacity;

/** constructor **/
public HashTable(int capacity)    {
    this.capacity = capacity;
    arr = new int[this.capacity];
}
```

Hashing - Insert

```
/** function to insert **/
public void insert(int ele)
{
    arr[ele % capacity] = ele;
}

/** function to clear **/
public void clear()
{
    arr = new int[capacity];
}
```

Hashing - Remove

```
 /** function contains **/
public boolean contains(int ele)    {
    return arr[ele % capacity] == ele;
}

/** function to delete **/
public void delete(int ele)
{
    if (arr[ele % capacity] == ele)
        arr[ele % capacity] = 0;
    else
        System.out.println("\nError : Element not found\n");
}
```

Hashing – Display Hash table

```
/** function to print hash table **/
public void printTable()
{
    System.out.print("\nHash Table = ");
    for (int i = 0; i < capacity; i++)
        System.out.print(arr[i] +" ");
    System.out.println();
}
```

Hashing

The previous method is simple, but it is flawed if the table size is large. For example, assume a table size of 10007 and that all keys are eight or fewer characters long.

No matter what the hash function, there is the possibility that two keys could resolve to the same hash key. This situation is known as a collision.

When this occurs, there are two simple solutions:

- chaining
- linear probe (aka linear open addressing)

And two slightly more difficult solutions

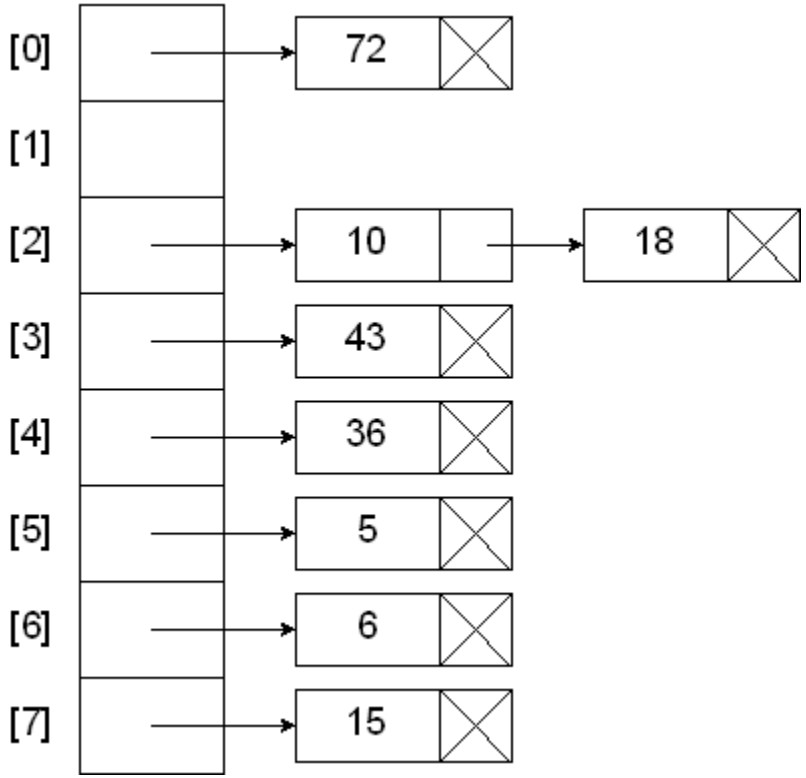
- Quadratic Probe
- Double Hashing

Hashing with Chains (Separate Chaining)

When a collision occurs, elements with the same hash key will be chained together. A chain is simply a linked list of all the elements with the same hash key.

Hash key = key % table size

4 = 36 % 8
2 = 18 % 8
0 = 72 % 8
3 = 43 % 8
6 = 6 % 8
2 = 10 % 8
5 = 5 % 8
7 = 15 % 8



Separate Chaining – Creating hash object

```
/* Class LinkedHashEntry */
class LinkedHashEntry  {
        String key;
        int value;
        LinkedHashEntry next;
        /* Constructor */
        LinkedHashEntry(String key, int value)     {
                this.key = key;
                this.value = value;
                this.next = null;
        }
}
```

Separate Chaining – Creating hash table

```
private int TABLE_SIZE;
private int size;
private LinkedHashEntry[] table;

/* Constructor */
public HashTable(int ts)  {
    size = 0;
    TABLE_SIZE = ts;
    table = new LinkedHashEntry[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++)
        table[i] = null;
}
```

Separate Chaining – Insert

```
/* Function to insert a key value pair */
public void insert(String key, int value)    {
    int hash = (myhash( key ) % TABLE_SIZE);
    if (table[hash] == null)
          table[hash] = new LinkedHashEntry(key, value);
    else {
          LinkedHashEntry entry = table[hash];
        while (entry.next != null && !entry.key.equals(key))
                        entry = entry.next;
        if (entry.key.equals(key))
                        entry.value = value;
        else
             entry.next = new LinkedHashEntry(key, value);
    }
    size++;
}
```

Separate Chaining – Remove

```
public void remove(String key)   {
        int hash = (myhash( key ) % TABLE_SIZE);
        if (table[hash] != null)    {
            LinkedHashEntry prevEntry = null;
            LinkedHashEntry entry = table[hash];
            while (entry.next != null && !entry.key.equals(key))  {
                prevEntry = entry;
                entry = entry.next;
            }
            if (entry.key.equals(key))  {
                if (prevEntry == null)
                    table[hash] = entry.next;
                else
                    prevEntry.next = entry.next;
                size--;
            }       }     }
```

Hashing with Linear Probe

When using a linear probe, the item will be stored in the next available slot in the table, assuming that the table is not already full.

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

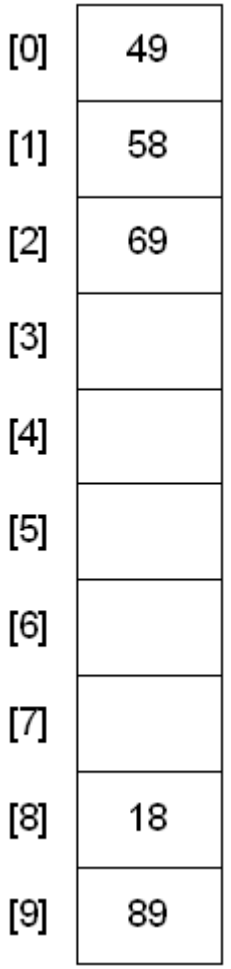
[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

Hashing with Linear Probe

A problem with the linear probe method is that it is possible for blocks of data to form when collisions are resolved. This is known as primary clustering.

This means that any key that hashes into the cluster will require several attempts to resolve the collision.

For example, insert the nodes 89, 18, 49, 58, and 69 into a hash table that holds 10 items using the division method:

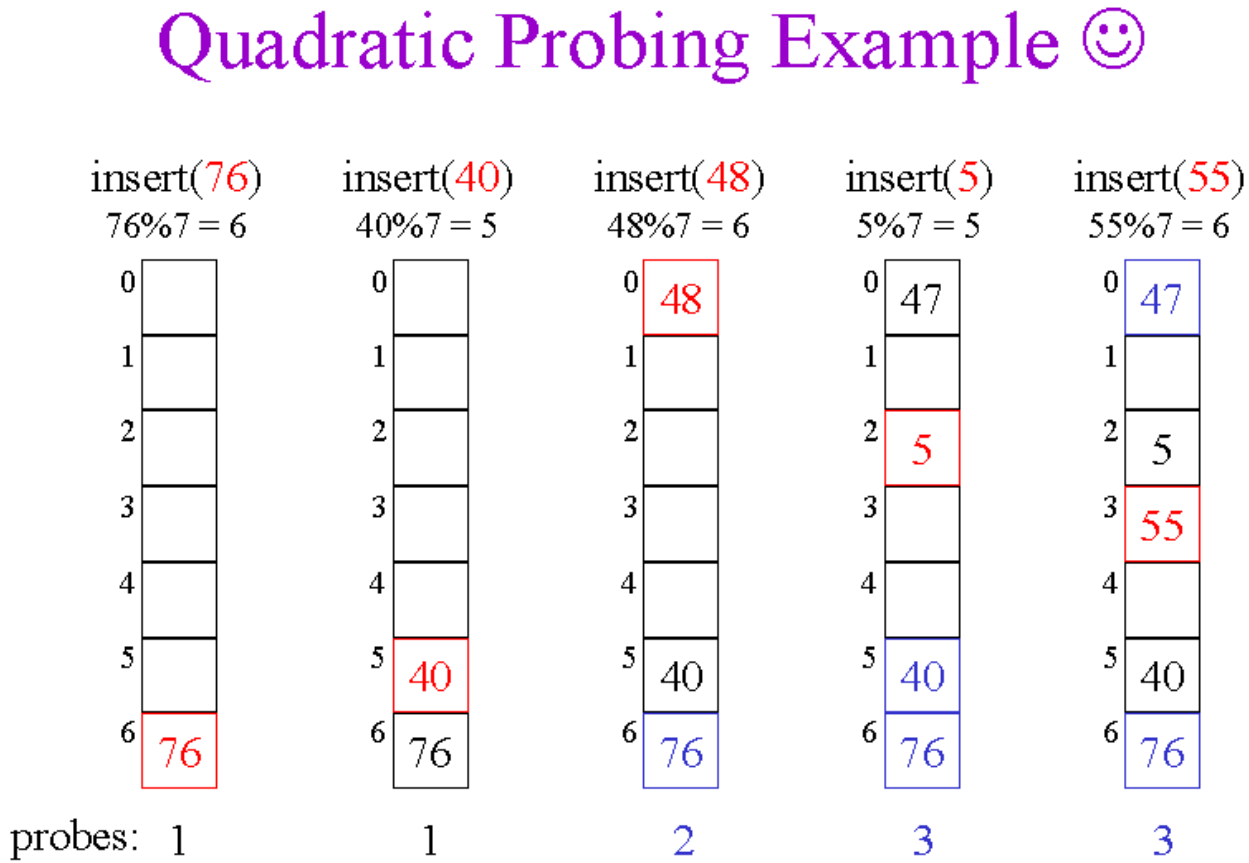


Hashing with Quadratic Probe

To resolve the primary clustering problem, quadratic probing can be used.

With quadratic probing, rather than always moving one spot, move i2 spots from the point of collision, where i is the number of attempts to resolve the collision.

Hashing with Quadratic Probe



Quadratic Probe – Creating Hash table

```
private int currentSize, maxSize;
private String[] keys;
private String[] vals;

/** Constructor **/
public QuadraticProbingHashTable(int capacity)     {
    currentSize = 0;
    maxSize = capacity;
    keys = new String[maxSize];
    vals = new String[maxSize];
}
```

Quadratic Probe – Insert

```
/** Function to insert key-value pair **/
public void insert(String key, String val)     {
    int tmp = hash(key);
    int i = tmp, h = 1;
    do {
        if (keys[i] == null)  {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }
```

Quadratic Probe – Insert

```
            if (keys[i].equals(key))
            {
                vals[i] = val;
                return;
            }
            i = (i + h * h++) % maxSize;
        } while (i != tmp);
    }
```

Quadratic Probe – Remove

```
/** Function to remove key and its value **/
public void remove(String key)
{
    if (!contains(key))
        return;

    /** find position key and delete **/
    int i = hash(key), h = 1;
    while (!key.equals(keys[i]))
                i = (i + h * h++) % maxSize;
    keys[i] = vals[i] = null;
```

Quadratic Probe – Remove

```
    /** rehash all keys **/
    for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)
    {
        String tmp1 = keys[i], tmp2 = vals[i];
        keys[i] = vals[i] = null;
        currentSize--;
        insert(tmp1, tmp2);
    }
    currentSize--;
}
```

Quadratic Probe – Display Hash table

```
/** Function to print HashTable **/
public void printHashTable()
{
    System.out.println("\nHash Table: ");
    for (int i = 0; i < maxSize; i++)
        if (keys[i] != null)
            System.out.println(keys[i] +" "+ vals[i]);
    System.out.println();
}
```

Quadratic Probe – Find Value

```
/** Function to get value for a given key **/
public String get(String key)    {
    int i = hash(key), h = 1;
    while (keys[i] != null)    {
        if (keys[i].equals(key))
            return vals[i];
        i = (i + h * h++) % maxSize;
        System.out.println("i "+ i);
    }
    return null;
}
```

Hashing with Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.

There are a couple of requirements for the second function:

it must never evaluate to 0
must make sure that all cells can be probed

A popular second hash function is: Hash2(key) = R - (key % R) where R is a prime number that is smaller than the size of the table.

Hashing with Double Hashing

Double Hashing – Creating Hash object

```
    /* Class LinkedHashEntry */
    class HashEntry  {
    String key;
    int value;

    /* Constructor */
    HashEntry(String key, int value)  {
        this.key = key;
        this.value = value;
    }
}
```

Double Hashing – Creating Hash table

```
private int TABLE_SIZE;
private int size, primeSize;
private HashEntry[] table;

 /* Constructor */
public HashTable(int ts)     {
    size = 0;        TABLE_SIZE = ts;
    table = new HashEntry[TABLE_SIZE];
    for (int i = 0; i < TABLE_SIZE; i++)
        table[i] = null;
    primeSize = getPrime();
}
```

Double Hashing – Find Prime

```
/* Function to get prime number less than table size for myhash2 function */
public int getPrime()  {
    for (int i = TABLE_SIZE - 1; i >= 1; i--)   {
        int fact = 0;
        for (int j = 2; j <= (int) Math.sqrt(i); j++)
            if (i % j == 0)
                fact++;
        if (fact == 0)
            return i;
    }
    /* Return a prime number */
    return 3;    }
```

Double Hashing – Insert

```
/* Function to insert a key value pair */
public void insert(String key, int value)  {
    if (size == TABLE_SIZE)  {
        System.out.println("Table full"); return;        }
    int hash1 = myhash1( key );
    int hash2 = myhash2( key );
    while (table[hash1] != null)    {
        hash1 += hash2;
        hash1 %= TABLE_SIZE;        }
    table[hash1] = new HashEntry(key, value);
    size++;
}
```

Double Hashing – First Hash function

```
/* Function myhash which gives a hash value for a given string */
 private int myhash1(String x )
 {
     int hashVal = x.hashCode( );
     hashVal %= TABLE_SIZE;
     if (hashVal < 0)
         hashVal += TABLE_SIZE;
     return hashVal;
 }
```

Double Hashing – Second Hash function

```
/* Function myhash function for double hashing */
private int myhash2(String x )
{
    int hashVal = x.hashCode( );
    hashVal %= TABLE_SIZE;
    if (hashVal < 0)
        hashVal += TABLE_SIZE;
    return primeSize - hashVal % primeSize;
}
```

Double Hashing – Remove

```
/* Function to remove a key */
public void remove(String key)  {
    int hash1 = myhash1( key );
    int hash2 = myhash2( key );
    while (table[hash1] != null && !table[hash1].key.equals(key))    {
        hash1 += hash2;
        hash1 %= TABLE_SIZE;
    }
    table[hash1] = null;
    size--;
}
```

Double Hashing – Display Hash table

```
/* Function to print hash table */
public void printHashTable()
{
    System.out.println("\nHash Table");
    for (int i = 0; i < TABLE_SIZE; i++)
        if (table[i] != null)
            System.out.println(table[i].key +" "+table[i].value);
}
```

Double Hashing – Find value

```
 /* Function to get value of a key */
public int get(String key)  {
    int hash1 = myhash1( key );
    int hash2 = myhash2( key );
     while (table[hash1] != null && !table[hash1].key.equals(key))   {
        hash1 += hash2;
        hash1 %= TABLE_SIZE;
    }
    return table[hash1].value;
}
```

Hashing with Rehashing

Once the hash table gets too full, the running time for operations will start to take too long and may fail.

To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

The new size of the hash table:

- should also be prime
- will be used to calculate the new insertion spot (hence the name rehashing)

This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is ok though since it doesn't happen that often.

When should the rehashing be applied?

- Once the table becomes half full
- Once an insertion fails
- Once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size

When should the rehashing be applied?

- Separate Chaining - Implement hash table as LinkedHashTable
- Linear Probing – Next Possible position
- Quadratic Probing - $(h(k)+ i^2)$ mod N
- Double Hashing - R - (key % R) where R is a prime number that is smaller than the size of the table.
- Re hashing – Multiply hash table size by 2 and find prime number. Then continue hashing from first