

Unit 6: Searching Algorithms and Hashing

Searching is an operation or a technique that helps find the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching techniques that are being followed in the data structure are listed below:

- Linear Search or Sequential Search
- Binary Search

1. Sequential Search

In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.

The list given below is the list of elements in an unsorted array. The array contains ten elements. Suppose the element to be searched is '46', so 46 is compared with all the elements starting from the 0th element, and the searching process ends where 46 is found, or the list ends.

The performance of the linear search can be measured by counting the comparisons done to find out an element. The number of comparisons is $O(n)$.

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop**, compare the search element with the current array element, and -
 - If the element matches, then return the index of the corresponding array element.
 - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **-1**.

Now, let's see the algorithm of linear search.

Algorithm

Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

Step 1: set pos = -1

Step 2: set i = 1

Step 3: repeat step 4 while i <= n

Step 4: if a[i] == val

set pos = i

print pos

go to step 6

[end of if]

set ii = i + 1

[end of loop]

Unit 6: Searching Algorithms and Hashing

```
Step 5: if pos = -1
print "value is not present in the array "
[end of if]
Step 6: exit
```

Working of Linear search

To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

	0	1	2	3	4	5	6	7	8
..	70	40	30	11	57	41	25	14	52

Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.

	0	1	2	3	4	5	6	7	8
	70	40	30	11	57	41	25	14	52

↑
K ≠ 70

The value of **K**, i.e., **41**, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
K ≠ 40

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
K ≠ 30

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
K ≠ 11

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
K ≠ 57

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
K = 41

Now, the element to be searched is found. So algorithm will return the index of the element matched.

Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

Case	Time Complexity
Best Case	$O(1)$
Average Case	$O(n)$
Worst Case	$O(n)$

- **Best Case Complexity** - In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **$O(1)$** .
- **Average Case Complexity** - The average case time complexity of linear search is **$O(n)$** .
- **Worst Case Complexity** - In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **$O(n)$** .

The time complexity of linear search is **$O(n)$** because every element in the array is compared only once.

2. Space Complexity

Space Complexity	$O(1)$
------------------	--------

- The space complexity of linear search is $O(1)$.

Implementation of Linear Search

```
#include <iostream>
using namespace std;
int linearSearch(int a[], int n, int val) {
    // Going through array linearly
    for (int i = 0; i < n; i++)
    {
        if (a[i] == val)
            return i+1;
    }
    return -1;
}
int main() {
    int a[] = {69, 39, 29, 10, 56, 40, 24, 13, 51}; // given array
    int val = 56; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
    int res = linearSearch(a, n, val); // Store result
    cout<<"The elements of the array are - ";
    for (int i = 0; i < n; i++)
        cout<<a[i]<<" ";
```

Unit 6: Searching Algorithms and Hashing

```
cout<<"\nElement to be searched is - "<<val;
if (res == -1)
cout<<"\nElement is not present in the array";
else
cout<<"\nElement is present at "<<res<<" position of array";
return 0;
}
```

Output

```
The elements of the array are - 56 30 20 41 67 31 22 14 52
Element to be searched is - 14
Element is present at 8 position of array
```

2. Binary Search

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

To do the binary search, first we have to sort the array elements. The logic behind this technique is given below.

1. First find the middle element of the array.
2. Compare the middle element with an item.
3. There are three cases.
 - a). If it is a desired element then search is successful,
 - b). If it is less than the desired item then search only in the first half of the array.
 - c). If it is greater than the desired item, search in the second half of the array.
4. Repeat the same steps until an element is found or search area is exhausted.

In this way, at each step we reduce the length of the list to be searched by half.

Requirements

- i. The list must be ordered
- ii. Rapid random access is required, so we cannot use binary search for a linked list

Unit 6: Searching Algorithms and Hashing

Algorithm

```
Binary_Search(a, lower_bound, upper_bound, val)
/* 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of
the last array element, 'val' is the value to search */
Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
Step 2: repeat steps 3 and 4 while beg <=end
Step 3: set mid = (beg + end)/2
Step 4: if a[mid] = val
    set pos = mid
    print pos
    go to step 6
else if a[mid] > val
    set end = mid - 1
else
    set beg = mid + 1
[end of if]
[end of loop]
Step 5: if pos = -1
    print "value is not present in the array"
[end of if]
Step 6: exit
```

Working of Binary search

To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

Unit 6: Searching Algorithms and Hashing

Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

1. $\text{mid} = (\text{beg} + \text{end})/2$

So, in the given array -

beg = 0

end = 8

mid = $(0 + 8)/2 = 4$. So, 4 is the mid of the array.

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[\text{mid}] = 39$
 $A[\text{mid}] < K$ (or, $39 < 56$)
So, $\text{beg} = \text{mid} + 1 = 5$, $\text{end} = 8$
Now, $\text{mid} = (\text{beg} + \text{end})/2 = 13/2 = 6$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[\text{mid}] = 51$
 $A[\text{mid}] < K$ (or, $51 < 56$)
So, $\text{beg} = \text{mid} + 1 = 7$, $\text{end} = 8$
Now, $\text{mid} = (\text{beg} + \text{end})/2 = 15/2 = 7$

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

↑
 $A[\text{mid}] = 56$
 $A[\text{mid}] = K$ (or, $56 = 56$)
So, location = mid
Element found at 7th location of the array

Now, the element to search is found. So algorithm will return the index of the element matched.

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	$O(1)$
Average Case	$O(\log n)$
Worst Case	$O(\log n)$

Unit 6: Searching Algorithms and Hashing

- **Best Case Complexity** - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **$O(1)$** .
- **Average Case Complexity** - The average case time complexity of Binary search is **$O(\log n)$** .
- **Worst Case Complexity** - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **$O(\log n)$** .

2. Space Complexity

Space Complexity	$O(1)$
------------------	--------

- The space complexity of binary search is $O(1)$.

Implementation of Binary Search

```
#include <iostream>
using namespace std;
int binarySearch(int a[], int beg, int end, int val)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        /* if the item to be searched is present at middle */
        if(a[mid] == val)
        {
            return mid+1;
        }
        /* if the item to be searched is smaller than middle, then it can only be in left subarray */
        else if(a[mid] < val)
        {
            return binarySearch(a, mid+1, end, val);
        }
        /* if the item to be searched is greater than middle, then it can only be in right subarray */
        else
        {
            return binarySearch(a, beg, mid-1, val);
        }
    }
    return -1;
}

int main() {
    int a[] = {10, 12, 24, 29, 39, 40, 51, 56, 70}; // given array
    int val = 51; // value to be searched
    int n = sizeof(a) / sizeof(a[0]); // size of array
```

Unit 6: Searching Algorithms and Hashing

```
int res = binarySearch(a, 0, n-1, val); // Store result
cout<<"The elements of the array are - ";
for (int i = 0; i < n; i++)
    cout<<a[i]<<" ";
cout<<"\nElement to be searched is - "<<val;
if (res == -1)
    cout<<"\nElement is not present in the array";
else
    cout<<"\nElement is present at "<<res<<" position of array";
return 0;
}
```

Output

```
The elements of the array are - 10 12 24 29 39 40 51 56 70
Element to be searched is - 51
Element is present at 7 position of array
```

3. Hashing

Hashing in the data structure is a technique of mapping a large chunk of data into small tables using a hashing function. It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.

Hashing is an important data structure designed to solve the problem of efficiently finding and storing data in an array. For example, if you have a list of 20000 numbers, and you have given a number to search in that list- you will scan each number in the list until you find a match.

The hash function in the data structure verifies the file which has been imported from another source. A hash key for an item can be used to accelerate the process. It increases the efficiency of retrieval and optimises the search. This is how we can simply give hashing definition in data structure.

It requires a significant amount of your time to search in the entire list and locate that specific number. This manual process of scanning is not only time-consuming but inefficient too. With hashing in the data structure, you can narrow down the search and find the number within seconds.

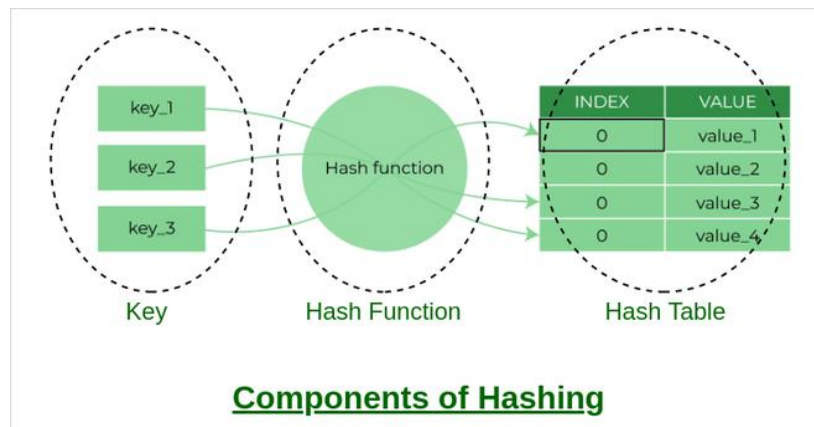
Components of Hashing

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**.

Unit 6: Searching Algorithms and Hashing

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.



3.1 Hash Function

The hash function creates a mapping between key and value, this is done through the use of mathematical formulas known as hash functions. The result of the hash function is referred to as a hash value or hash. The hash value is a representation of the original string of characters but usually smaller than the original.

For example: Consider an array as a Map where the key is the index and the value is the value at that index. So for an array A if we have index i which will be treated as the key then we can find the value by simply looking at the value at $A[i]$.
simply looking up $A[i]$.

Types of Hash functions

- Division Method.
- Mid Square Method.
- Folding Method.
- Multiplication Method.

1. Division Method (Modulo-Division Hashing)

The division method is the simplest and easiest method used to generate a hash value. In this hash function, the value of k is divided by M and uses the remainder as obtained.

Formula - $h(K) = k \bmod M$

(where k = key value and M = the size of the hash table)

Advantages -

- This method works well for any value of M
- The division approach is extremely quick because it only calls for one operation.

Disadvantages -

- This may lead to poor performance as consecutive keys are mapped to consecutive hash values in the hash table
- There are situations when choosing the value of M requires particular caution.

Example -

- $k = 1320$
- $M = 11$
- $h(1320) = 1320 \bmod 11$
- $= 0$

2. Mid Square Method

The steps involved in computing this hash method include the following -

1. Squaring the value of k (like $k*k$)
2. Extract the hash value from the middle r digits.

Formula - $h(K) = h(k \times k)$

(where k = key value)

Advantages -

- Since most or all of the key value's digits contribute to the outcome, this strategy performs well. The middle digits of the squared result are produced by a process in which all of the essential digits participate.
- The top or bottom digits of the original key value do not predominate in the outcome.

Disadvantages -

- One of this method's constraints is the size of the key; if the key is large, its square will have twice as many digits.
- Chance of repeated collisions.

Example -

Let's take the hash table with 200 memory locations and $r = 2$, as decided on the size of the mapping in the table.

- $k = 50$
- Therefore,
- $k = k \times k$
- $= 50 \times 50$
- $= 2500$
- Thus,
- $h(50) = 50$

3. Folding Method

There are two steps in this method -

1. The key-value k should be divided into a specific number of parts, such as $k_1, k_2, k_3, \dots, k_n$, each having the very same number of digits aside from the final component, which may have fewer digits than the remaining parts.
2. Add each component separately. The last carry, if any, is disregarded to determine the hash value.

Formula - $k = k_1, k_2, k_3, k_4, \dots, k_n$

$$s = k_1 + k_2 + k_3 + k_4 + \dots + k_n$$

$$h(K) = s$$

(Where, s = addition of the parts of key k)

Advantages -

- Breaks up the key value into precise equal-sized segments for an easy hash value
- Independent of distribution in a hash table

Disadvantages -

- Sometimes inefficient if there are too many collisions

Example -

- $k = 54321$
- $k_1 = 54 ; k_2 = 32 ; k_3 = 1$
- Therefore,
- $s = k_1 + k_2 + k_3$
- $= 54 + 32 + 1$
- $= 87$
- Thus,
- $h(k) = 87$

4. Multiplication Method

Steps to follow -

1. Pick up a constant value A (where $0 < A < 1$)
2. Multiply A with the key value
3. Take the fractional part of kA
4. Take the result of the previous step and multiply it by the size of the hash table, M .

Formula - $h(K) = \text{floor}(M(kA \bmod 1))$

Where,

M = size of the hash table,

k = key value

A = constant value)

Advantages -

- It may be applied to any number between 0 and 1, albeit some numbers tend to produce better results than others.

Disadvantages -

- When the table size is a power of two, the multiplication technique is typically appropriate since multiplication hashing makes it possible to compute the index by key quickly.

Example -

- k = 1234
- A = 0.35784
- M = 100
- So,
- $h(1234) = \text{floor} [100(1234 \times 0.35784 \bmod 1)]$
- $= \text{floor} [100 (441.57456 \bmod 1)]$
- $= \text{floor} [100 (0. 57456)]$
- $= \text{floor} [57.456]$
- $= 57$
- Thus,
- $h(1234) = 57$

Choosing a good hash function

- Creating an effective hash function that distributes the added item's index value evenly across the database is important.
- Quick and easier to compute according to the requirements.
- An approach to successfully resolve collisions in hash tables is essential for generating an index for a key whose hash index corresponds to an existing spot.

3.2 Hash Table

A hash table is simply an array that is address via a hash function. It is a data structure made up of

- A table of some fixed size to hold a collection of records each uniquely identified by some key
- A function called hash function that is used to generate index values in the table.

Unit 6: Searching Algorithms and Hashing

Calculating a Hash Table:

Let's take a simple example by taking each number mod 10, and putting it into a hash table that has 10 slots.

Numbers to hash: 22, 3, 18, 29

We take each value, apply the hash function to it, and the result tells us what slot to put that value in, with the left column denoting the slot, and the right column denoting what value is in that slot, if any.

Our hash function here is to take each value mod 10. The table to the right shows the resulting hash table. We hash a series of values as we get them, so the first value we hash is the first value in the string of values, and the last value we hash is the last value in the string of values.

0	
1	
2	22
3	3
4	
5	
6	
7	
8	18
9	29

$22 \bmod 10 = 2$, so it goes in slot 2.

$3 \bmod 10 = 3$, so it goes in slot 3.

$18 \bmod 10 = 8$, so it goes in slot 8.

$29 \bmod 10 = 9$, so it goes in slot 9.

3.3 Hashing as a Data Structure and a Search Technique

Hashing is a data structure and a search technique that allows for efficient storage and retrieval of data. It involves mapping data elements, called keys, to corresponding positions in a data structure called a hash table.

A hash table is an array-based data structure that uses a hashing function to determine the index or address where a key-value pair should be stored. The hashing function takes a key as input and computes a hash code, which is typically an integer. This hash code is used to calculate the index in the hash table where the data should be stored.

When storing data in a hash table, the key-value pair is hashed using the hashing function, and the resulting hash code is used to determine the index in the array. If multiple key-value pairs produce the same hash code (known as a hash collision), different techniques can be used to handle collisions, such as chaining or open addressing.

Chaining involves creating a linked list at each index of the hash table. If two or more key-value pairs have the same hash code, they are stored as nodes in the linked list at the corresponding index. When searching for a key, the hash code is computed, and the linked list at that index is traversed to find the desired key-value pair.

Open addressing, on the other hand, aims to resolve collisions by finding an alternative position within the hash table to store the key-value pair. There are different open addressing techniques, such as linear probing, quadratic probing, or double hashing, which determine the sequence of probe locations to find an empty slot for storage or to locate the desired key during search.

Unit 6: Searching Algorithms and Hashing

The search process in a hash table is typically very efficient. When searching for a key, the hash code is computed using the same hashing function as during insertion. The computed hash code is used to locate the index in the hash table where the key-value pair might reside. If the key is found at the expected index, the search is successful. If a collision occurred, the appropriate collision resolution technique is employed to find the correct key-value pair.

Hashing provides constant-time average-case complexity for insertion, deletion, and search operations when collisions are minimal. However, in the worst-case scenario with many collisions, the performance may degrade to linear time complexity, making it less efficient. The quality of the hashing function and the capacity of the hash table are important factors in maintaining good performance.

Hashing is widely used in various applications, including databases, caches, symbol tables, and cryptographic algorithms, due to its ability to provide efficient retrieval and storage of data.

4. Collision in Hash Table

A collision occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure (hash table) being generated by the hash function.

Example Hash Table With Collisions:

Let's take the exact same hash function from before: take the value to be hashed mod 10, and place it in that slot in the hash table.


Numbers to hash: 22, 9, 14, 17, 42

As before, the hash table is shown to the right.

As before, we hash each value as it appears in the string of values to hash, starting with the first value. The first four values can be entered into the hash table without any issues. It is the last value, 42, however, that causes a problem. $42 \bmod 10 = 2$, but there is already a value in slot 2 of the hash table, namely 22. This is a collision.

The value 42 must end up in one of the hash table's slots, but arbitrarily assigning it a slot at random would make accessing data in a hash table much more time consuming, as we obviously want to retain the constant time growth of accessing our hash table. There are two common ways to deal with collisions: chaining, and open addressing.

0	
1	
2	22
3	
4	14
5	
6	
7	17
8	
9	9

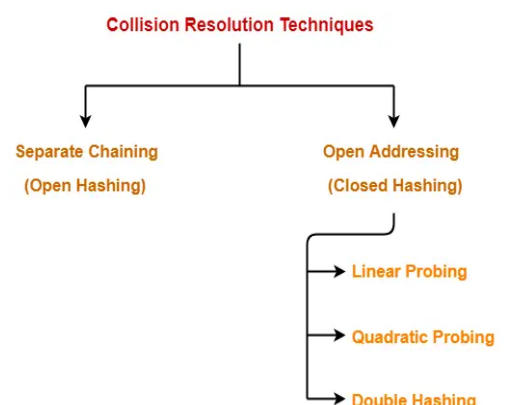


5. Collision Resolution Techniques

Finding an alternate location for the hashed key is called **collision resolution**. Collision resolution techniques are used in hashing to handle situations where two or more keys produce the same hash code, resulting in a collision.

There are several commonly used collision resolution techniques:

1. Direct Chaining: Array of linked list implementation.



Unit 6: Searching Algorithms and Hashing

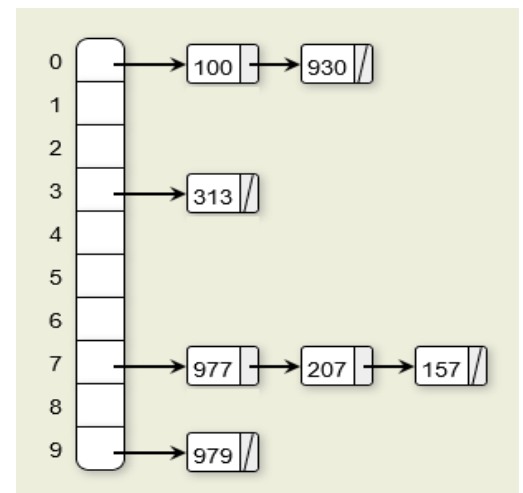
- Separate chaining
- 2. Open Addressing: Array-based implementation.
 - Linear Probing
 - Quadratic Probing
 - Double Hashing Chaining

5.1 Open Hashing

Open Hashing is popularly known as Separate Chaining. This is a technique which is used to implement an array as a linked list known as a chain. It is one of the most used techniques by programmers to handle collisions.

5.1.1 Separate Chaining

When two or more string hash to the same location (hash key), we can append them into a singly linked list (called chain) pointed by the hash key. A hash table then is an array of lists. It is called open hashing because it uses extra memory to resolve collision.



Advantages of Open Hashing:

1. Simple to implement
2. Hash table never fills up; we can always add more elements to the chain.
3. Less sensitive to the function or load factors.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages of Open Hashing:

1. The cache performance of the Separate Chaining method is poor as the keys are stored using a singly linked list.
2. A lot of storage space is wasted as some parts of the hash table are never used.
3. In the worst case, the search time can become " $O(n)$ ". This happens only if the chain becomes too lengthy.
4. Uses extra space for links.

5.2 Closed Hashing

Similar to separate chaining, open addressing is a technique for dealing with collisions. In Open Addressing, the hash table alone houses all of the elements. The size of the table must therefore always be more than or equal to the total number of keys at all times (Note that we can increase table size by copying old data if needed). This strategy is often referred to as **closed hashing**.

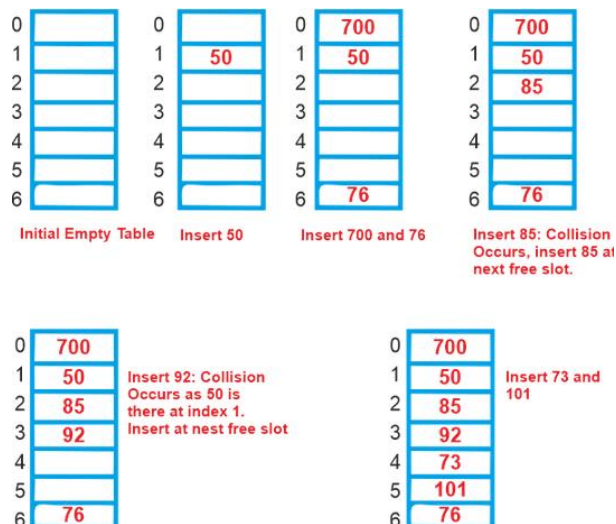
5.2.1 Linear Probing

Defination: insert K_i at first free location from $(u+i)\%m$ where $i = 0$ to $(m-1)$

Another technique of collision resolution is a linear probing. If we cannot insert at index k , we try the next slot $k+1$. If that one is occupied, we go to $k+2$, and so on. This is quite simple approach but it requires new thinking about hash tables. Do you always find an empty slot? What do you do when you reach the end of the table?

Search the hash table sequentially starting from the original hash location. If we reach the end of the table, wrap up from last to the first location. The major disadvantage of linear probing is the clustering of keys together in a consecutive pattern. This results in one part being very dense, while other parts having few items. It causes long probe searches.

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



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.

5.2.2 Quadratic Probing

Defination: insert K_i at first free location from $(u+i^2)\%m$ where $i=0$ to $(m-1)$. This method makes an attempt to correct the problem of clustering with linear probing. It forces the problem key to move quickly a considerable distance from the initial collision.

When a key value hashes and collision occurs for key, this method probes the table location at:

$(h(k) + 1^2) \% \text{table-size}$,

$(h(k) + 2^2) \% \text{table-size}$,

$(h(k) + 3^2) \% \text{table-size}$ and so on.

Unit 6: Searching Algorithms and Hashing

That is, the first rehash adds 1 to the hash value. The second rehash adds 4, the third adds 9 and so on. It reduces primary clustering but suffers from secondary clustering : hash that hash to some initial slot will probe the same alternative cells.

There is no guarantee to finding an empty location once the table gets more than half full.

9		9		9	89	9	89
8		8		8		8	18
7		7		7		7	
6		6		6		6	
5		5	15	5	15	5	15
4		4		4		4	
3		3		3		3	
2		2		2		2	
1		1		1		1	
0	20	0	20	0	20	0	20
	$20\%10=0$		$15\%10=5$		$89\%10=9$		$18\%10=8$

9	$89((49+1)\%10)$	9	$89((48+4)\%10)$	9	$89((79+1)\%10)$	9	89
8	18	8	$18((48+1)\%10)$	8	$18((79+16)\%10)$	8	18
7		7		7		7	
6		6		6		6	
5	15	5	15	5	$15((79+25)\%10)$	5	15
4		4		4	79	4	79
3	49	3	49	3	$49((79+9)\%10)$	3	49
2		2	48	2	48	2	48
1		1		1		1	21
0	$20((49+4)\%10)$	0	20	0	$20((79+4)\%10)$	0	20
	$49\%10=9$		$48\%10=8$		$79\%10=9$		$21\%10=1$

5.2.3 Double Hashing

Defination: insert K_i at first free place from $(u+v*i)\%m$ where $i=0$ to $(m-1)$ & $u=h_1k\%m$ and $v=h_2k\%m$

Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence.

Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing.

Advantages of Double hashing

- The advantage of Double hashing is that it is one of the best forms of probing, producing a uniform distribution of records throughout a hash table.
- This technique does not yield any clusters.
- It is one of the effective methods for resolving collisions.

Double hashing can be done using :

$(\text{hash1}(\text{key}) + i * \text{hash2}(\text{key})) \% \text{TABLE_SIZE}$

Unit 6: Searching Algorithms and Hashing

Here $\text{hash1}()$ and $\text{hash2}()$ are hash functions and TABLE_SIZE is size of hash table.
(We repeat by increasing i when collision occurs)

Example:

Insert the keys 79, 69, 98, 72, 14, 50 into the **Hash Table of size 13**. Resolve all collisions using Double Hashing where first hash-function is $h_1(k) = k \bmod 13$ and second hash-function is $h_2(k) = 1 + (k \bmod 11)$

Solution:

Initially, all the hash table locations are empty. We pick our **first key = 79** and apply $h_1(k)$ over it,

$h_1(79) = 79 \% 13 = 1$, hence key 79 hashes to 1st location of the hash table. But before placing the key, we need to make sure the 1st location of the hash table is empty. In our case it is empty and we can easily place key 79 in there.

Second key = 69, we again apply $h_1(k)$ over it, $h_1(69) = 69 \% 13 = 4$, since the 4th location is empty we can place 69 there.

Third key = 98, we apply $h_1(k)$ over it, $h_1(98) = 98 \% 13 = 7$, 7th location is empty so 98 placed there.

Fourth key = 72, $h_1(72) = 72 \% 13 = 7$, now this is a collision because the **7th** location is already occupied, we need to resolve this collision using double hashing.

$$\begin{aligned} h_{\text{new}} &= [h_1(72) + i * h_2(72)] \% 13 \\ &= [7 + 1 * (1 + 72 \% 11)] \% 13 \\ &= 1 \end{aligned}$$

Location **1st** is already occupied in the hash-table, hence we again had a collision. Now we again recalculate with $i = 2$

$$\begin{aligned} h_{\text{new}} &= [h_1(72) + i * h_2(72)] \% 13 \\ &= [7 + 2 * (1 + 72 \% 11)] \% 13 \\ &= 8 \end{aligned}$$

Location 8th is empty in hash-table and now we can place key 72 in there.

Fifth key = 14, $h_1(14) = 14 \% 13 = 1$, now this is again a collision because 1st location is already occupied, we now need to resolve this collision using double hashing.

$$\begin{aligned} h_{\text{new}} &= [h_1(14) + i * h_2(14)] \% 13 \\ &= [1 + 1 * (1 + 14 \% 11)] \% 13 \\ &= 5 \end{aligned}$$

Location 5th is empty and we can easily place key 14 there.

Sixth key = 50, $h_1(50) = 50 \% 13 = 11$, 11th location is already empty and we can place our key 50 there.

Unit 6: Searching Algorithms and Hashing

Since all the keys are placed in our hash table the double hashing procedure is completed. Finally, our hash table looks like the following,

KEYS : 79, 69, 98, 72, 14, 50

0	
1	79
2	
3	
4	69
5	14
6	
7	98
8	72
9	
10	
11	50
12	

Hash table

6. Load Factor and Rehashing

In terms of hashing, the load factor refers to the measure of how full a hash table is, or the ratio of the number of elements stored in the table to the total number of slots or buckets available in the hash table.

Load factor (α) is defined as-

$$\text{Load factor } (\alpha) = \frac{\text{Number of elements present in Hash Table}}{\text{Total size of Hash Table}}$$

The load factor helps determine the capacity utilization of the hash table. It provides valuable insights into how efficiently the hash table is being utilized and can impact the performance of operations such as insertion, deletion, and search.

The load factor value in open addressing is always between 0 and 1. This is due to

- In open addressing, the hash table contains all of the keys.
- As a result, the table's size is always more than or at least equal to the number of keys it stores.

Rehashing

Rehashing is the process of increasing the size of a hashmap and redistributing the elements to new buckets based on their new hash values. It is done to improve the performance of the hashmap and to prevent collisions caused by a high load factor.

When a hashmap becomes full, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. As the load factor increases, the number of collisions also increases, which can lead to poor performance. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions.

During rehashing, all elements of the hashmap are iterated and their new bucket positions are calculated using the new hash function that corresponds to the new size of the hashmap. This process can be time-consuming but it is necessary to maintain the efficiency of the hashmap.

Unit 6: Searching Algorithms and Hashing

Why rehashing?

Rehashing is needed in a hashmap to prevent collision and to maintain the efficiency of the data structure.

As elements are inserted into a hashmap, the load factor (i.e., the ratio of the number of elements to the number of buckets) increases. If the load factor exceeds a certain threshold (often set to 0.75), the hashmap becomes inefficient as the number of collisions increases. To avoid this, the hashmap can be resized and the elements can be rehashed to new buckets, which decreases the load factor and reduces the number of collisions. This process is known as rehashing.

Rehashing can be costly in terms of time and space, but it is necessary to maintain the efficiency of the hashmap.

How Rehashing is done?

Rehashing can be done as follows:

- For each addition of a new entry to the map, check the load factor.
- If it's greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucketarray.
- Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

Example:

- Given
 $h(k) = x \bmod m$
Where $x = \text{key}$ and $m = \text{size of bucket array}$
- Keys = 33, 34, 35
[33|34|35]
Here $n=3; N=3; \lambda=1$
- Now, Rehashing
 $N' = 7$ (Closest Prime Number to $2N$)
 $h'(x) = x \bmod 7$
- Here's the new array:
[35| | | | |33|34]

Comparison of above three

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.

Unit 6: Searching Algorithms and Hashing

Open Addressing	Chaining
Also known as Closed Hashing	Also known as Open Hashing/ Closed Addressing
All elements would be stored in the Hash table itself. No additional data structure is needed.	Additional Data structure needs to be used to accommodate collision data.
In cases of collisions, a unique hash key must be obtained.	Simple and effective approach to collision resolution. Key may or may not be unique.
Determining size of the hash table, adequate enough for storing all the data is difficult.	Performance deterioration of closed addressing much slower as compared to Open addressing.
State needs be maintained for the data (additional work)	Performance deterioration of closed addressing much slower as compared to Open addressing.
Uses space efficiently	Expensive on space

Procedures of some Hash Resolutions:

An array is declared and each cell is assigned -1 to denote free cell.

```
int hash[10];
for (int i=0; i<10; i++)
hash[i] = -1;
```

void linear_probing(int hash[], int key)

```
{
    int rem;
    rem = key%10;
    if(hash[rem] == -1;
        hash[rem] = key;
    else
    {
        while(hash[rem] != -1)
        {
            If(rem >= 10)
                rem = 0;
            rem++;
        }
        hash[rem] = key;
    }
}
```

void double_hashing(int hash[], int key)

```
{
```

```
int rem;
rem = key%10;
if(hash[rem] == -1)
    hash[rem] = key;
else
{
    Rem = 7 - (key%7)
    while(hash[rem] != -1)
    {
        if(rem >= 10)
            rem = 0;
        rem++;
    }
    hash[rem]=key;
}
}
```

void quadratic_probing(int hash[], int key)

```
{
    int rem;
    rem = key%10;
    if(hash[rem] == -1)
        hash[rem] = key;
    else
    {
        int x = 0;
        int h = rem;
        while(hash[rem] != -1)
        {
            x++;
            int rem1 = h + pow(x,2);
            rem = rem1%10;
        }
        hash[rem] = key;
    }
}
```

Procedure for chaining:

```
struct chain
{
    int data;
    struct chain *next;
};
```

Unit 6: Searching Algorithms and Hashing

```
typedef struct chain node;
```

node *table[10]; Here, initially all the node pointers of the table are initialized to NULL.

```
void chaining(int key)
```

```
{
    int rem;
    rem = key%10;
    node *p;
    p = (node *)malloc(sizeof(node));
    p->data = key;
    p->next = NULL;
    if(table[rem] == NULL)
        table[rem] = p;
    else
    {
        node *temp;
        temp = table[rem];
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = p;
    }
}
```

Program of Implement Hash system using linear probing for collision

```
#include <iostream>
```

```
using namespace std;
```

```
const int TABLE_SIZE = 7;
```

```
class HashTable {
```

```
private:
```

```
    int table[TABLE_SIZE];
```

```
    int hash(int key) {
```

```
        return key % TABLE_SIZE;
```

```
    }
```

```
public:
```

```
    HashTable() {
```

```
        for (int i = 0; i < TABLE_SIZE; i++) {
```

```
            table[i] = -1;
```

```
        }
```

```
}

void put(int key) {
    int index = hash(key);
    while (table[index] != -1) {
        index = (index + 1) % TABLE_SIZE;
    }
    table[index] = key;
}

void contains(int key) {
    int index = hash(key);
    while (table[index] != -1) {
        if (table[index] == key) {
            cout << key << ": Key Found" << endl;
            return;
        }
        index = (index + 1) % TABLE_SIZE;
    }
    cout << key << ": Key Not Found" << endl;
}

void display() {
    cout << "Hash Table:" << endl;
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (table[i] != -1) {
            cout << table[i] << " ";
        } else {
            cout << "- ";
        }
    }
    cout << endl;
}

};

int main() {
    HashTable hashTable;
    hashTable.put(15);
    hashTable.put(11);
    hashTable.put(27);
    hashTable.put(8);
    hashTable.put(12);
```


Unit 6: Searching Algorithms and Hashing

```
hashTable.display();
cout<<"\nSearching Results:"<<endl;
hashTable.contains(12);
hashTable.contains(10);
hashTable.contains(8);

return 0;
}
```