Chapter 8: Searching

Searching

Searching is an operation or a technique that helps finds 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. If the data is kept properly in sorted order, then searching becomes very easy and efficient. Some of the standard searching technique that is being followed in the data structure is listed below:

1. Linear Search or Sequential Search
2. Binary Search

Sequential Search

Sequential or Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found. The time complexity of Linear search algorithm is **O(n).**

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

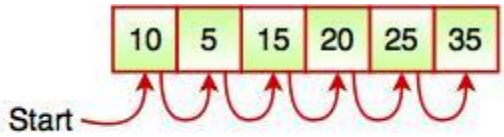


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order.

Algorithm

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
```

```
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Implementing Linear Search

Following are the steps of implementation that we will be following:

1. Traverse the array using a `for` loop.
2. In every iteration, compare the `item` value with the current value of the array.
    - If the values match, return the current index of the array.
    - If the values do not match, move on to the next array element.
3. If no match is found, return `-1`.

Implementation of linear search in C

```
 /*
    - values[] => array with all the values
    - item => value to be found
    - n => total number of elements in the array
*/
int linearSearch(int values[], int item, int n)
{
   for(int i = 0; i < n; i++)
   {
      if (values[i] == item)
      {
         return i;
      }
   }
   return -1;
}
```

Binary Search

Binary Search is a fast and efficient search algorithm applied on the sorted array or list of large size. It's time complexity of **O(log n)**. The only limitation is that the array or list of elements must

be sorted for the binary search algorithm to work on it. If the elements are not sorted already, we need to sort them first.

This search algorithm works on the principle of divide and conquer. Binary search looks for a particular item by comparing the middle item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

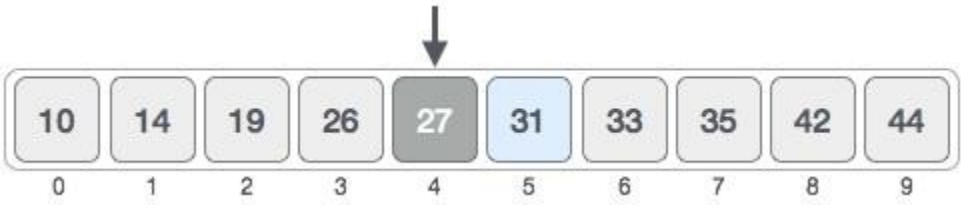
How Binary Search Works?

For a binary search to work, it is mandatory for the item array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the item value must be in the upper portion of the array.

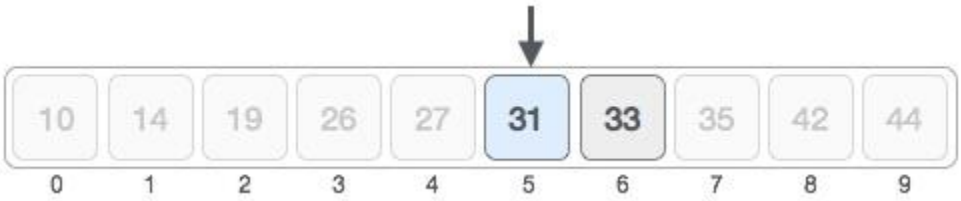
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our item value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our item value. We find that it is a match.

We conclude that the item value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Implementing Binary Search Algorithm

Following are the steps of implementation that we will be following:

1. Start with the middle element:
    - If the **item** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the **item** value,
        - If the item value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.

- If the item value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.

2. When a match is found, return the index of the element matched.
3. If no match is found, then return `-1`

Implementing Iterative Binary Search in C++

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

int binarySearch(int arr[], int l, int r, int x)
{
   while (l <= r) {
      int m = l + (r - l) / 2;

      // Check if x is present at mid
      if (arr[m] == x)
         return m;

      // If x greater, ignore left half
      if (arr[m] < x)
         l = m + 1;

      // If x is smaller, ignore right half
      else
         r = m - 1;
   }

   // if we reach here, then element was
   // not present
   return -1;
}

int main()
{
   int arr[] = { 2, 3, 4, 10, 40 };
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
   int result = binarySearch(arr, 0, n - 1, x);
   if(result == -1)
            cout << "Element is not present in array";
   else
            cout << "Element is present at index " << result;
   return 0;
}
```

Recursive implementation of Binary Search

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

C++ program to implement recursive Binary Search

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

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;

      // If the element is present at the middle
      // itself
      if (arr[mid] == x)
         return mid;

      // If element is smaller than mid, then
      // it can only be present in left subarray
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);

      // Else the element can only be present in right subarray
      return binarySearch(arr, mid + 1, r, x);
   }

   // We reach here when element is not present in array
   return -1;
}

int main()
{
   int arr[] = { 2, 3, 4, 10, 40 };
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
   int result = binarySearch(arr, 0, n - 1, x);
   if(result == -1)
         cout << "Element is not present in array";
   else
         cout << "Element is present at index " << result;
   return 0;
}
```

Hashing

The problem at hands is to speed up searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have a function that would tell us the index for a given value. With this function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a **hash function**. A hash function is a function which when given a key, generates an address in the table.

Hashing is a technique used for storing and retrieving information as quickly as possible. It allows performing **optimal searches** and is useful in implementing hash tables.

As part of Hashing a string is transformed into a shorter, fixed-length value(or key) which represents the original string. In general, when we store strings in a database and then have to search those strings, character by character comparison is carried out on each string until a match is found, this is a very cumbersome process.

But what if we generate a unique 4 digit number for each string using some conversion function and then whenever we are asked to search for any string, we can convert the string to be searched into the 4 digit number and then look for that number. This way, the search will be very fast.

Hashing provides a way to perform insert, delete and search operation in O(1) average time, which although can reach O(n) in the worst cases. When we use hashing, we use a function to generate a hash code for the data to be inserted and then whenever we have to search that data, we again use the same function to generate the unique hash code and can easily find the data.

Let a hash function H(x) maps the value x at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.

Why Hashing?

Let's take an example to understand the need for hashing. Let's say we have a table in our database in which we store names of students like:

- John Wick
- John Rambo
- Tony Stark
- Steve Martin
- Nicki Lauda
- John Travolta, etc.

If the table has 1 million such name entries, then whenever we have to search for any name, names are searched by matching characters one by one which is time-consuming. But what if we create a function which can produce a unique 7 digit number for every name. For example,

- John Wick - 0089761
- John Rambo - 0089651
- Tony Stark - 0912321, etc

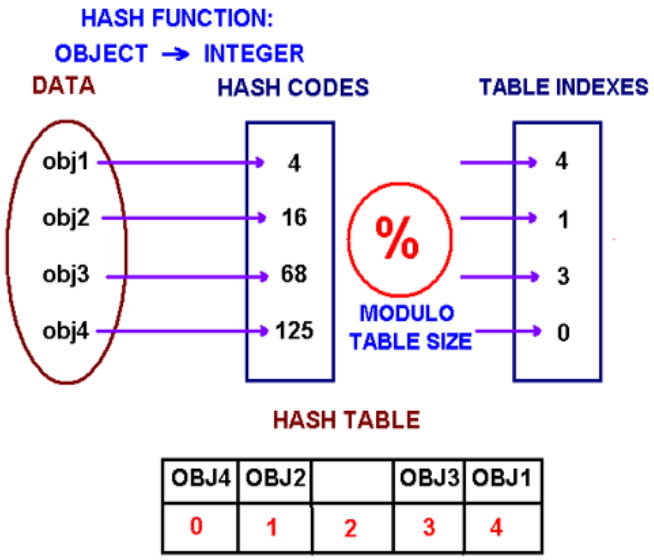
Now, if someone has to search for 'Tony Stark', rather than searching strings, we can convert the name into numeric 7 digit code and can then look for that code which would be multifold faster than string search.

The table which stores the keys and their respective hashed value is termed as a **hash table**, and the function which is used to generate the hash keys is called a **hash function**.

Hashing is not only useful for searching and indexing but this technique is also used for encryption.

Hash Table

Create an array of size *M*. Choose a hash function *h*, that is a mapping from objects into integers *0, 1, ..., M-1*. Put these objects into an array at indexes computed via the hash function *index = h(object)*. Such array is called a **hash table**.



Hash Function: A function that converts a given object to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
One way to create such a perfect hash function is to have the size of the hash table so large that it can accommodate all the hash keys in it. But then there will be no point of calling it hashing, and memory will be wasted too as most of the keys aren't required frequently.

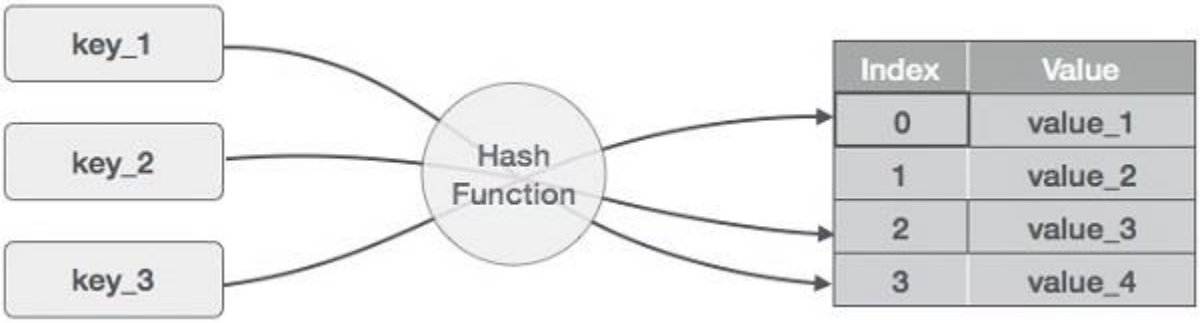
Our goal is to choose a hash function that quickly calculates a hash value, evenly distributes the keys into the hash table and causes a minimum number of collisions.

A good hash function should have following properties

1) Efficiently computable.
2) Should uniformly distribute the keys (Each table position equally likely for each key)

Example

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

SN	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12

6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.
- **Insert** − inserts an element in a hash table.
- **delete** − Deletes an element from a hash table.

Collisions

When we put objects into a hash table, it is possible that different objects might have the same hash key (hash code). This is called a **collision**.

How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions.

Collision Resolution Techniques

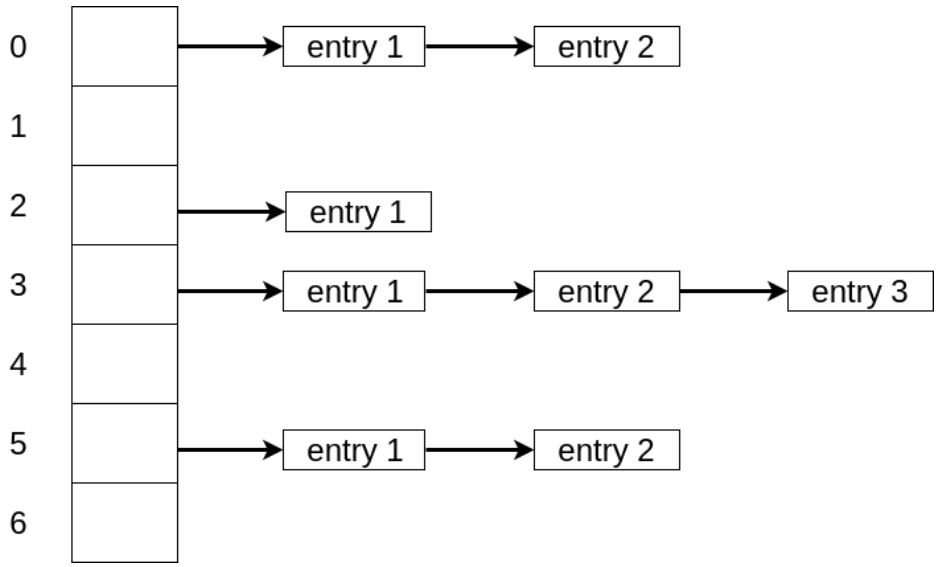
Finding an alternate location for the hashed key is called **collision resolution**. There are a number of such techniques:

1. **Direct Chaining**: Array of linked list implementation.
   - Separate chaining
2. **Open Addressing**: Array-based implementation.
   - Linear Probing
   - Quadratic Probing
   - Double Hashing

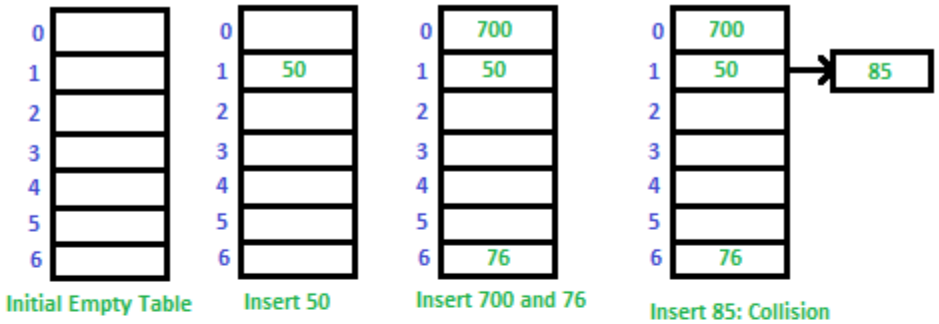
Separate Chaining (Open Hashing)

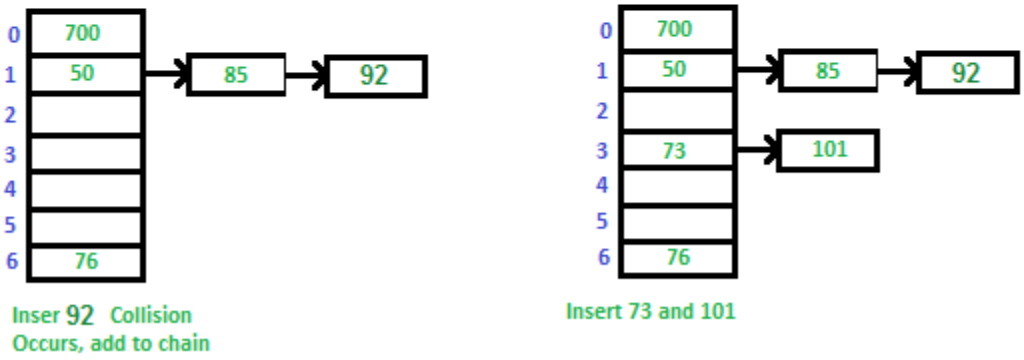
When two or more string hash to the same location (hash key), we can append them into a singly linked list (called chain) pointed by the hash key. A hash table then is an array of lists

It is called open hashing because it uses extra memory to resolve collision.



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





Advantages:

1) Simple to implement.
2) Hash table never fills up, we can always add more elements to the chain.

3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

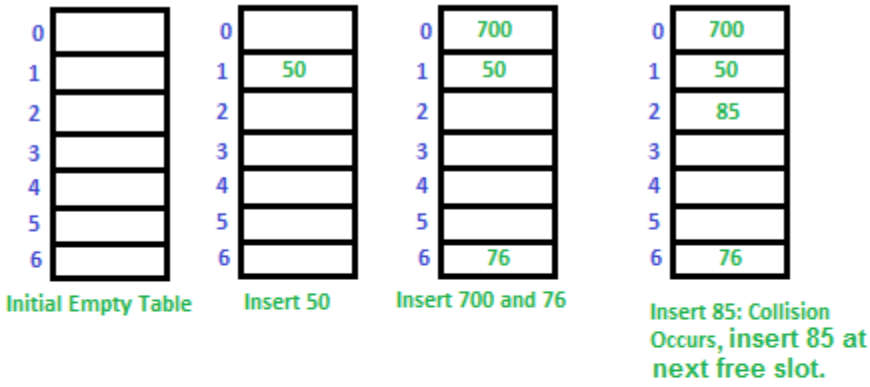
Disadvantages:
1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
2) Wastage of Space (Some Parts of hash table are never used)
3) If the chain becomes long, then search time can become O(n) in the worst case.
4) Uses extra space for links.

Linear Probing

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.

b) Quadratic Probing: We look for i^2'th slot in i'th iteration.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S

If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S

If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

.....

.....

c) Double Hashing: We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

.....

.....

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.

S.No.	**Separate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.

2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing