Unit - 5
Searching

Searching in data structures refers to the process of locating a particular item or element within a given dataset efficiently. The efficiency of a search operation is crucial, especially when dealing with large volumes of data, as it directly impacts the performance of various algorithms and applications. Here are some characteristics of searching in data structures:

1. Time Complexity: The time complexity of a search operation determines how efficiently the search can be performed. Different data structures offer varying time complexities for search operations. For example, a binary search in a sorted array has a time complexity of O(log n), making it highly efficient compared to a linear search with O(n) time complexity.

2. Space Complexity: Space complexity refers to the amount of memory or space required by a search algorithm to execute. Some search algorithms may require additional space for auxiliary data structures or variables, while others may operate in constant space. Balancing time and space complexity is essential in designing efficient search algorithms.

3. Data Organization: The organization of data within a data structure significantly impacts search operations. For instance, in a sorted array, elements are arranged in a particular order, enabling efficient binary search. In contrast, in an unsorted array or a linked list, linear search may be the only option unless additional data structures like hash tables are used for optimization.

4. Search Criteria: The criteria for searching, such as equality, range, or pattern matching, influence the choice of search algorithm and data structure. For example, hash tables excel in exact match searches, while tree-based structures like binary search trees are suitable for range queries.

5. Search Flexibility: Some data structures offer flexibility in terms of search operations. For instance, a balanced search tree allows various types of searches, including exact match, range queries, and nearest neighbor searches, making it versatile for different applications.

6. Optimization Techniques: Advanced optimization techniques, such as pruning in tree-based searches or caching in hash-based searches, can further enhance the efficiency of search operations. These techniques aim to reduce the search space or eliminate redundant computations, improving overall performance.

In summary, searching in data structures involves finding specific elements efficiently, considering factors such as time and space complexity, data organization, search criteria, flexibility, and optimization techniques. The choice of data structure and search algorithm depends on the nature of the dataset and the requirements of the application.

Linear search is a simple searching algorithm that sequentially checks each element in a list or array until the target element is found or the end of the list is reached. It starts from the beginning of the data structure and compares each element with the target value. If the element matches the target, the search is successful; otherwise, it moves to the next element. Linear search has a time complexity of O(n), where n is the number of elements in the list.

Example: Suppose we have an array of integers [3, 8, 15, 20, 25] and we want to find the index of the element 15 using linear search.

Linear Search Algorithm:

1. Start from the first element of the array.
2. Iterate through each element of the array.
3. Compare the current element with the target element.

4. If the current element matches the target, return its index.
5. If the end of the array is reached without finding the target, return -1.

Binary search is a more efficient searching algorithm that works on sorted arrays or lists. It compares the target value with the middle element of the array. If the target value matches the middle element, the search is successful. If the target value is less than the middle element, the search continues on the lower half of the array; otherwise, it continues on the upper half. This process is repeated until the target element is found or the search space is empty. Binary search has a time complexity of O(log n), where n is the number of elements in the list.

Example: Suppose we have a sorted array of integers [5, 10, 15, 20, 25, 30, 35, 40] and we want to find the index of the element 20 using binary search.

Binary Search Algorithm:

1. Initialize variables to store the lower and upper bounds of the search space, set initially to the first and last indices of the array respectively.
2. Repeat the following steps until the lower bound is less than or equal to the upper bound:
   a. Calculate the middle index of the search space.
   b. Compare the middle element with the target element.
   c. If the middle element is equal to the target, return its index.
   d. If the middle element is greater than the target, update the upper bound to be one less than the middle index.
   e. If the middle element is less than the target, update the lower bound to be one more than the middle index.
3. If the target element is not found after the loop, return -1.

There are three cases used in the binary search:

Case 1: data<a[mid] then left = mid+1.
Case 2: data>a[mid] then right=mid-1

Case 3: data = a[mid] // element is found

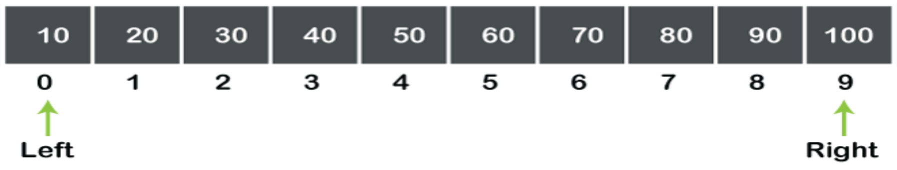
In the above case, '**a**' is the name of the array, **mid** is the index of the element calculated recursively, **data** is the element that is to be searched, **left** denotes the left element of the array and **right** denotes the element that occur on the right side of the array.

Let's understand the working of binary search through an example.

Suppose we have an array of 10 size which is indexed from 0 to 9 as shown in the below figure:

We want to search for 70 element from the above array.

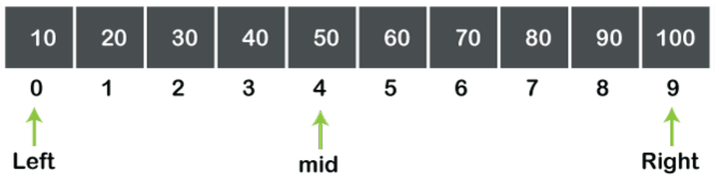
Step 1: First, we calculate the middle element of an array. We consider two variables, i.e., left and right. Initially, left =0 and right=9 as shown in the below figure:



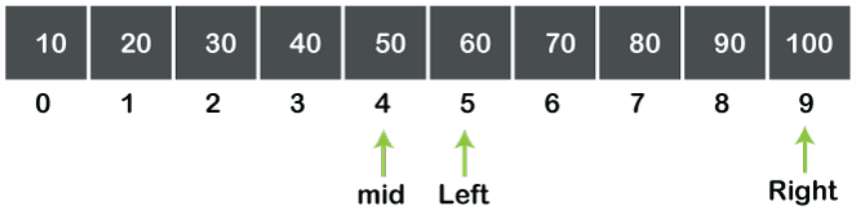
The middle element value can be calculated as:

$$mid = \frac{left + right}{2}$$

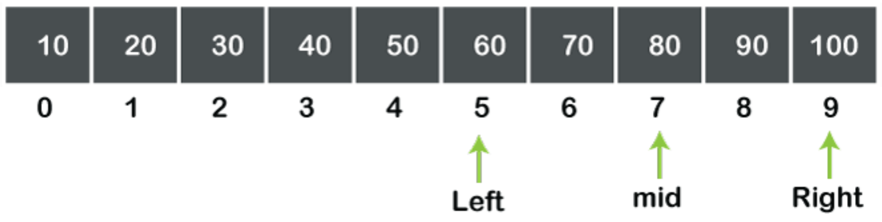
Therefore, mid = 4 and a[mid] = 50. The element to be searched is 70, so a[mid] is not equal to data. The case 2 is satisfied, i.e., data>a[mid].



Step 2: As data>a[mid], so the value of left is incremented by mid+1, i.e., left=mid+1. The value of mid is 4, so the value of left becomes 5. Now, we have got a subarray as shown in the below figure:

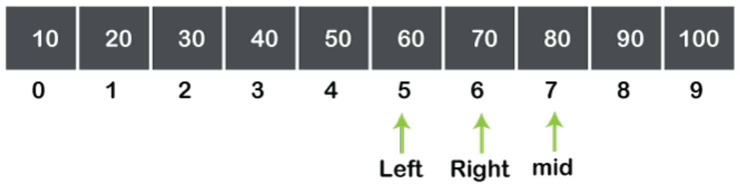


Now again, the mid-value is calculated by using the above formula, and the value of mid becomes 7. Now, the mid can be represented as:

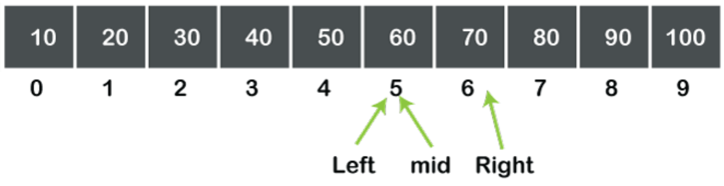


In the above figure, we can observe that a[mid]>data, so again, the value of mid will be calculated in the next step.

Step 3: As a[mid]>data, the value of right is decremented by mid-1. The value of mid is 7, so the value of right becomes 6. The array can be represented as:



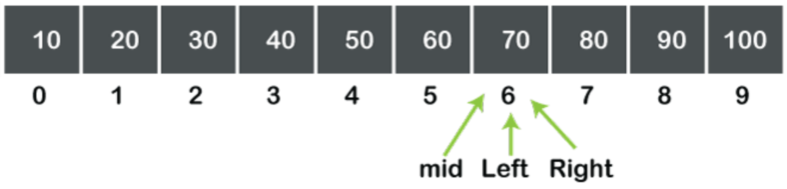
The value of mid will be calculated again. The values of left and right are 5 and 6, respectively. Therefore, the value of mid is 5. Now the mid can be represented in an array as shown below:



In the above figure, we can observe that a[mid]<data.

Step 4: As a[mid]<data, the left value is incremented by mid+1. The value of mid is 5, so the value of left becomes 6.

Now the value of mid is calculated again by using the formula which we have already discussed. The values of left and right are 6 and 6 respectively, so the value of mid becomes 6 as shown in the below figure:



We can observe in the above figure that a[mid]=data. Therefore, the search is completed, and the element is found successfully.

Here are examples illustrating both linear search and binary search:

Linear Search Example:

Let's consider an array of integers: [4, 2, 7, 1, 9, 5, 8, 3, 6]. We want to find the index of the element 7.

Algorithm:
1. Start from the first element of the array.
2. Compare each element with the target value (7 in this case) sequentially.
3. If the element matches the target value, return the index.
4. If the end of the array is reached without finding the target value, return "Not found".

Solution:
1. Start from index 0: Element at index 0 is 4, not the target.
2. Move to index 1: Element at index 1 is 2, not the target.
3. Move to index 2: Element at index 2 is 7, which matches the target. Return index 2.

Binary Search Example:

Let's consider a sorted array of integers: [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]. We want to find the index of the element 15.

Algorithm:
1. Compare the target value with the middle element of the array.
2. If they match, return the index.
3. If the target value is less than the middle element, search the left half of the array.

4. If the target value is greater than the middle element, search the right half of the array.
5. Repeat the process until the target value is found or the search space is empty.

Solution:
1. Start with the entire array: The middle element is 15, which matches the target. Return index 4.

In summary, linear search sequentially checks each element until the target is found, while binary search divides the search space in half with each comparison, making it more efficient for sorted arrays.

C program that implements both linear search and binary search through a menu-driven approach:

```
#include <stdio.h>

// Function prototypes
int linearSearch(int arr[], int n, int target);
int binarySearch(int arr[], int low, int high, int target);

int main() {
   int choice, target;
   int arr[] = {3, 6, 9, 12, 15, 18, 21, 24, 27, 30};
   int n = sizeof(arr) / sizeof(arr[0]);

   printf("Menu:\n");
   printf("1. Linear Search\n");
   printf("2. Binary Search\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);

   switch (choice) {
      case 1:
         printf("Enter the target element to search: ");
         scanf("%d", &target);
         int linearResult = linearSearch(arr, n, target);
         if (linearResult == -1)
            printf("Element not found.\n");
         else
            printf("Element found at index %d.\n", linearResult);
         break;
      case 2:
         printf("Enter the target element to search: ");
         scanf("%d", &target);
         int binaryResult = binarySearch(arr, 0, n - 1, target);
         if (binaryResult == -1)
            printf("Element not found.\n");
         else
            printf("Element found at index %d.\n", binaryResult);
         break;
      default:
         printf("Invalid choice.\n");
   }

   return 0;
}

// Linear search function
```

```
int linearSearch(int arr[], int n, int target) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == target)
         return i; // Element found, return index
   }
   return -1; // Element not found
}

// Binary search function
int binarySearch(int arr[], int low, int high, int target) {
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (arr[mid] == target)
         return mid; // Element found, return index
      else if (arr[mid] < target)
         low = mid + 1; // Search right half
      else
         high = mid - 1; // Search left half
   }
   return -1; // Element not found
}
```

This program provides a menu to choose between linear search and binary search. Depending on the choice made by the user, it prompts for the target element to search and then displays the result.

Difference Between Linear and Binary Search: Linear vs Binary Search

Parameters	Linear Search	Binary Search
Definition	Linear Search sequentially checks each element in the list until it finds a match or exhausts the list.	Binary Search continuously divides the sorted list, comparing the middle element with the target value.
Time Complexity	The time complexity is O(n), where n is the number of elements in the list.	The time complexity is O(log n), making it faster for larger datasets.
Efficiency	Less efficient, especially for large datasets.	More efficient, especially for large datasets.
Data Requirement	Does not require the list to be sorted.	Requires the list to be sorted.
Implementation	Easier to implement.	Requires a more complex implementation.
Search Space	Examines each element sequentially.	Eliminates half of the search space with each comparison.
Use Case	Suitable for small and unsorted datasets.	Ideal for large and sorted datasets.

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.

Need for Hash data structure

Every day, the data on the internet is increasing multifold and it is always a struggle to store this data efficiently. In day-to-day programming, this amount of data might not be that big, but still, it needs to be stored, accessed, and processed easily and efficiently. A very common data structure that is used for such a purpose is the Array data structure.

Now the question arises if Array was already there, what was the need for a new data structure! The answer to this is in the word "**efficiency**". Though storing in Array takes O(1) time, searching in it takes at least O(log n) time. This time appears to be small, but for a large data set, it can cause a lot of problems and this, in turn, makes the Array data structure inefficient.

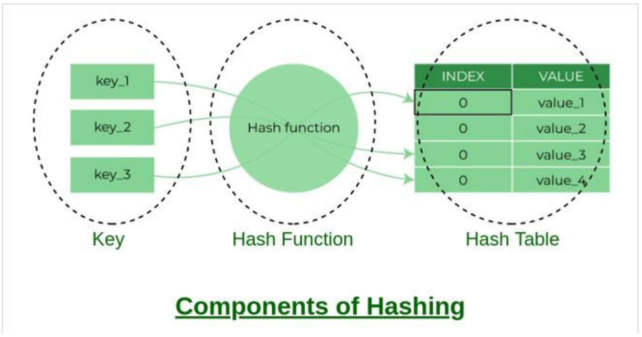
So now we are looking for a data structure that can store the data and search in it in constant time, i.e. in O(1) time. This is how Hashing data structure came into play. With the introduction of the Hash data structure, it is now possible to easily store data in constant time and retrieve them in constant time as well.

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.

How does Hashing work?



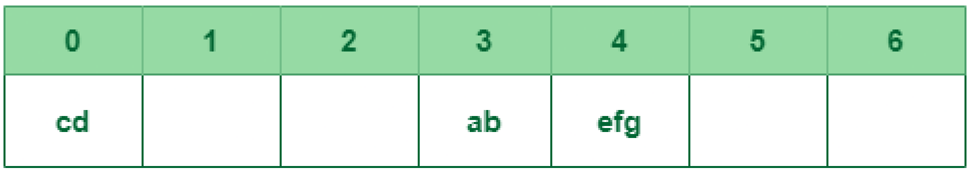
Components of Hashing

Suppose we have a set of strings {"ab", "cd", "efg"} and we would like to store it in a table.

Our main objective here is to search or update the values stored in the table quickly in O(1) time and we are not concerned about the ordering of strings in the table. So the given set of strings can act as a key and the string itself will act as the value of the string but how to store the value corresponding to the key?

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let's assign
  - "a" = 1,
  - "b"=2, .. etc, to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:
  - "ab" = 1 + 2 = 3,
  - "cd" = 3 + 4 = 7 ,
  - "efg" = 5 + 6 + 7 = 18

- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size**. We can compute the location of the string in the array by taking the **sum(string) mod 7**.
- **Step 5:** So we will then store
  - "ab" in 3 mod 7 = 3,
  - "cd" in 7 mod 7 = 0, and
  - "efg" in 18 mod 7 = 4.



0	1	2	3	4	5	6
cd			ab	efg		

Mapping key with indices of array

The above technique enables us to calculate the location of a given string by using a simple hash function and rapidly find the value that is stored in that location. Therefore the idea of hashing seems like a great way to store (key, value) pairs of the data in a table.

What is a 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:

There are many hash functions that use numeric or alphanumeric keys. Different:

1. Division Method.
2. Mid Square Method
3. Folding Method.
4. Multiplication Method

Properties of a Good hash function

A hash function that maps every item into its own unique slot is known as a perfect hash function. We can construct a perfect hash function if we know the items and the collection will never change but the problem is that there is no systematic way to construct a perfect hash function given an arbitrary collection of items. Fortunately, we will still gain performance efficiency even if the hash function isn't perfect. We can achieve a perfect hash function by increasing the size of the hash table so that every possible value can be accommodated. As a result, each item will have a unique slot. Although this approach is feasible for a small number of items, it is not practical when the number of possibilities is large.

So, We can construct our hash function to do the same but the things that we must be careful about while constructing our own hash function.

A good hash function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position is equally likely for each.
3. Should minimize collisions.
4. Should have a low load factor(number of items in the table divided by the size of the table).

Complexity of calculating hash value using the hash function

- Time complexity: O(n)
- Space complexity: O(1)

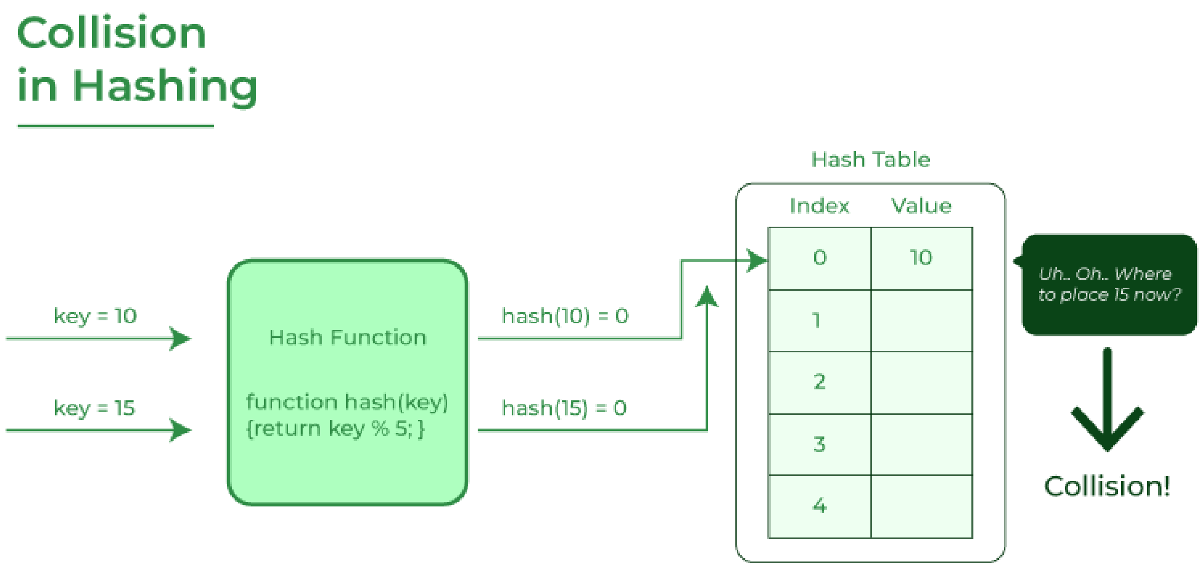
Problem with Hashing

If we consider the above example, the hash function we used is the sum of the letters, but if we examined the hash function closely then the problem can be easily visualized that for different strings same hash value is begin generated by the hash function.

For example: {“ab”, “ba”} both have the same hash value, and string {“cd”,”be”} also generate the same hash value, etc. This is known as **collision** and it creates problem in searching, insertion, deletion, and updating of value.

What is 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.



What is Collision in Hashing

How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining:
2. Open Addressing:

Separate Chaining

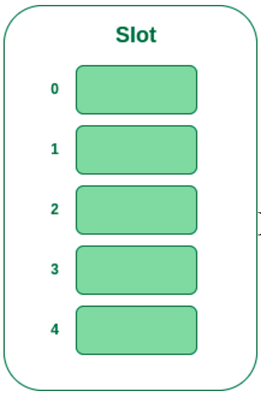
The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple but requires additional memory outside the table.

Example: We have given a hash function and we have to insert some elements in the hash table using a separate chaining method for collision resolution technique.

Hash function = key % 5,
Elements = 12, 15, 22, 25 and 37.

Let's see step by step approach to how to solve the above problem:

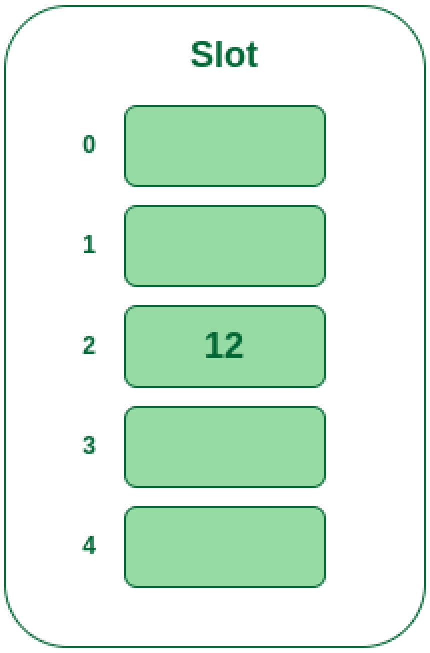
- **Step 1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



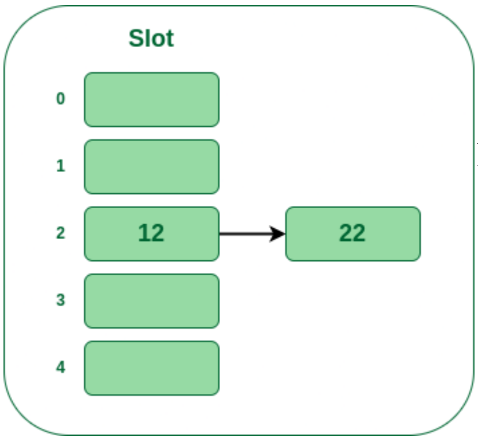
Hash table

- **Step 2:** Now insert all the keys in the hash table one by one. The first key to be inserted is 12 which is mapped to bucket number 2 which is calculated by using the hash function 12%5=2.

Insert 12

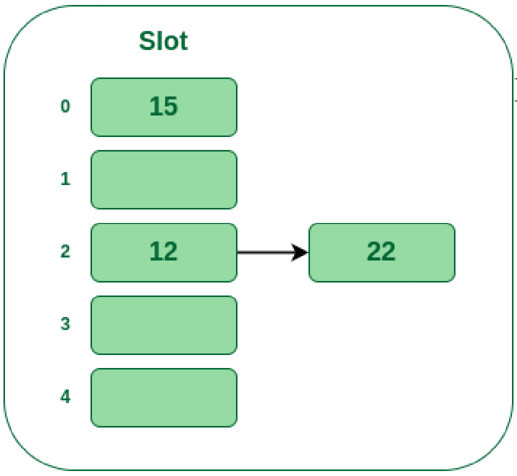


- **Step 3:** Now the next key is 22. It will map to bucket number 2 because 22%5=2. But bucket 2 is already occupied by key 12.



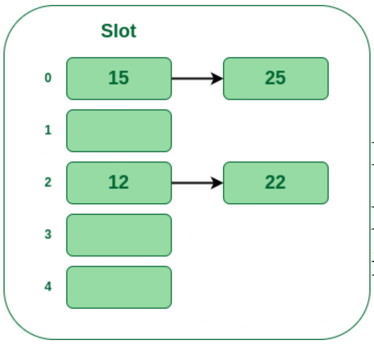
Insert 22

- **Step 4:** The next key is 15. It will map to slot number 0 because 15%5=0.



Insert 15

- **Step 5:** Now the next key is 25. Its bucket number will be 25%5=0. But bucket 0 is already occupied by key 25. So separate chaining method will again handle the collision by creating a linked list to bucket 0.



Insert 25

Hence In this way, the separate chaining method is used as the collision resolution technique.

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:

- ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
- ***Search(k):*** *Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.*
- ***Delete(k)****:* ***Delete operation is interesting****. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".*
  The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Different ways of Open Addressing:

1. Linear Probing:

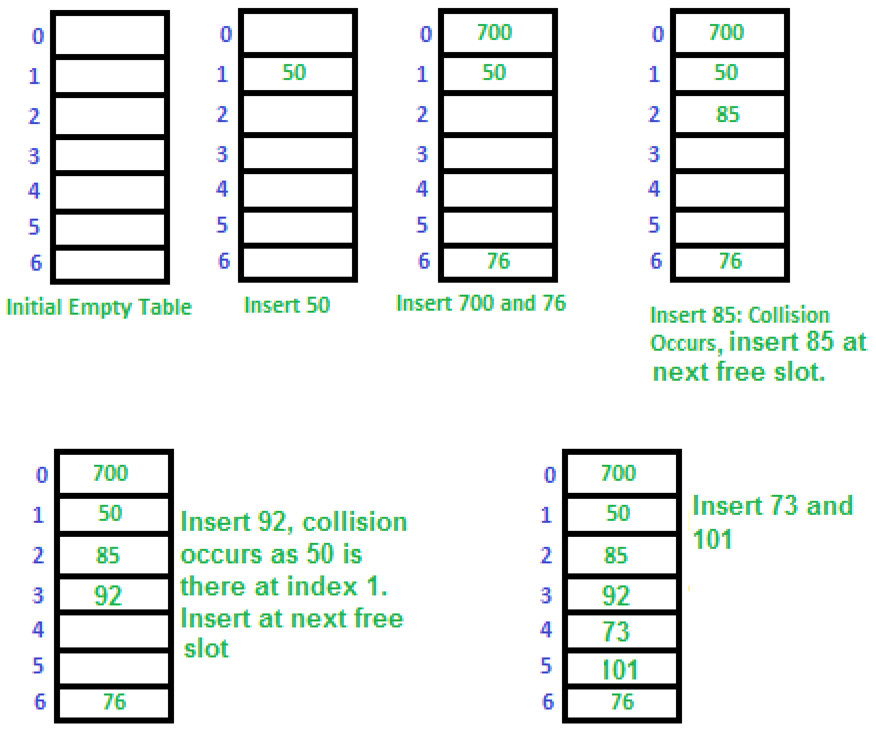
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.

For example, The typical gap between two probes is 1 as seen in the example below:
*Let **hash(x)** be the slot index computed using a hash function and **S** be the table size*
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
……………………………………………..
……………………………………………..

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

which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.We can define the hash function as per our choice if we want to create a hash table,although it is fixed internally with a pre-defined formula.



Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, insert 85 at next free slot.
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
Insert 73 and 101

Applications of linear probing:

Linear probing is a collision handling technique used in hashing, where the algorithm looks for the next available slot in the hash table to store the collided key. Some of the applications of linear probing include:

- **Symbol tables**: Linear probing is commonly used in symbol tables, which are used in compilers and interpreters to store variables and their associated values. Since symbol tables can grow dynamically, linear probing can be used to handle collisions and ensure that variables are stored efficiently.
- **Caching**: Linear probing can be used in caching systems to store frequently accessed data in memory. When a cache miss occurs, the data can be loaded into the cache using linear probing, and when a collision occurs, the next available slot in the cache can be used to store the data.

- **Databases**: Linear probing can be used in databases to store records and their associated keys. When a collision occurs, linear probing can be used to find the next available slot to store the record.
- **Compiler design**: Linear probing can be used in compiler design to implement symbol tables, error recovery mechanisms, and syntax analysis.
- **Spell checking:** Linear probing can be used in spell-checking software to store the dictionary of words and their associated frequency counts. When a collision occurs, linear probing can be used to store the word in the next available slot.

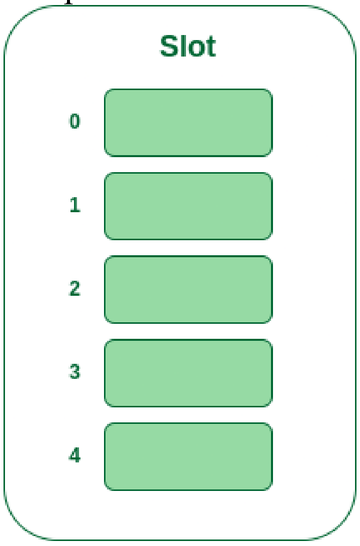
Overall, linear probing is a simple and efficient method for handling collisions in hash tables, and it can be used in a variety of applications that require efficient storage and retrieval of data.

Challenges in Linear Probing :

- **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
- **Secondary Clustering*:*** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

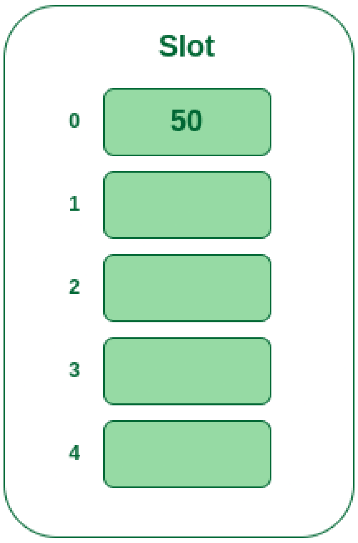
Example: Let us consider a simple hash function as "key mod 5" and a sequence of keys that are to be inserted are 50, 70, 76, 93.

- **Step1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



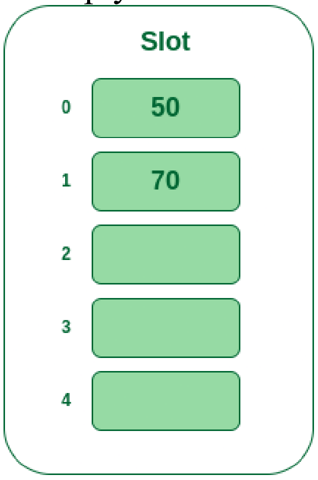
Hash table

- **Step 2:** Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.



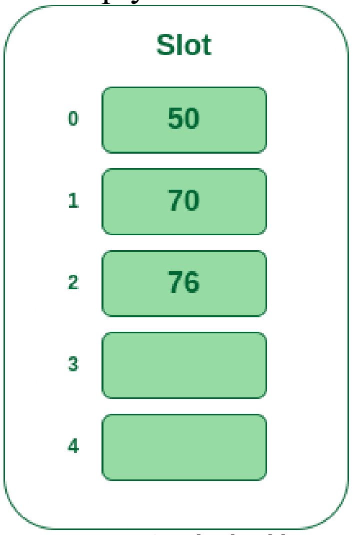
Insert 50 into hash table

- **Step 3:** The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.



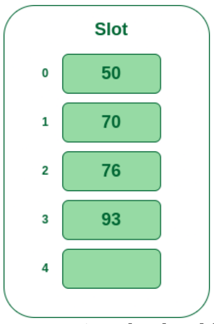
Insert 70 into hash table

- **Step 4:** The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.



Insert 76 into hash table

- **Step 5:** The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.



Insert 93 into hash table

2. Quadratic Probing

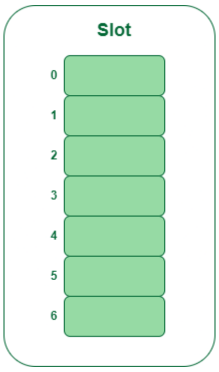
If you observe carefully, then you will understand that the interval between probes will increase proportionally to the hash value. Quadratic probing is a method with the help of which we can solve the problem of clustering that was discussed above. This method is also known as the **mid-square** method. In this method, we look for the **i^2'th** slot in the **i^{th}** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

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*
……………………………………………………..
……………………………………………………..

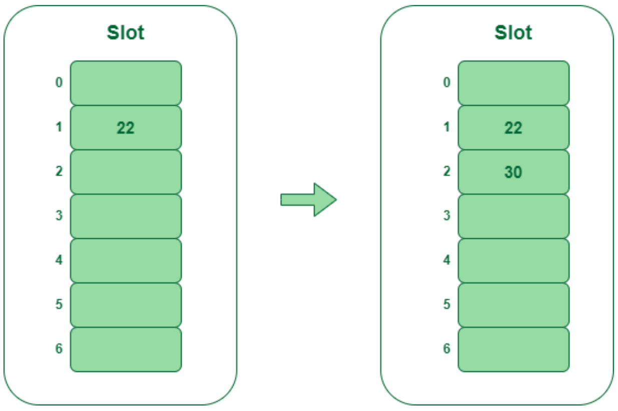
Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50.

- **Step 1:** Create a table of size 7.



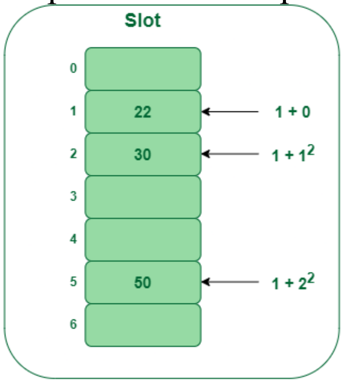
Hash table

- **Step 2** – Insert 22 and 30
  - Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
  - Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert keys 22 and 30 in the hash table

- **Step 3:** Inserting 50
  - Hash(50) = 50 % 7 = 1
  - In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
  - Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
  - Now, cell 5 is not occupied so we will place 50 in slot 5.



Insert key 50 in the hash table

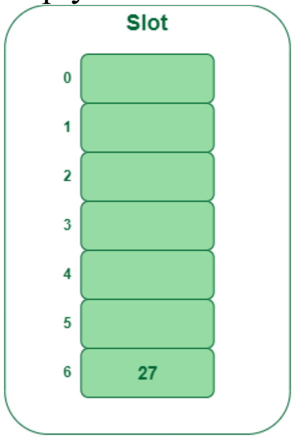
<u>Double Hashing</u>

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **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*
…………………………………………………..
…………………………………………………..

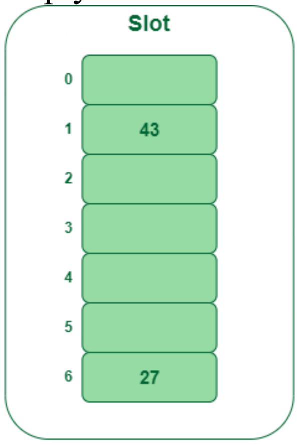
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
    - 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Insert key 27 in the hash table

- **Step 2:** Insert 43
    - 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.

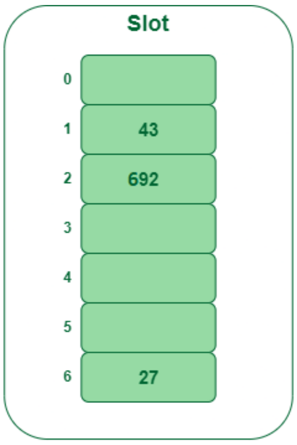


Insert key 43 in the hash table

- **Step 3:** Insert 692
    - 692 % 7 = 6, but location 6 is already being occupied and this is a collision
    - So we need to resolve this collision using double hashing.

$h_{new} = [h1(692) + i * (h2(692)] \% 7$
$= [6 + 1 * (1 + 692 \% 5)] \% 7$
$= 9 \% 7$
$= 2$

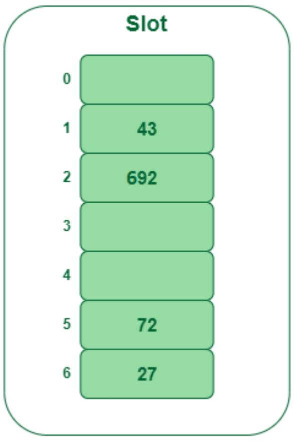
Now, as 2 is an empty slot,
so we can insert 692 into 2nd slot.



Insert key 692 in the hash table

- **Step 4:** Insert 72
    - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
    - So we need to resolve this collision using double hashing.

*h_{new} = [h1(72) + i * (h2(72)] % 7*
*= [2 + 1 * (1 + 72 % 5)] % 7*
= 5 % 7
= 5,
Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.



Insert key 72 in the hash table

See this for step-by-step diagrams:

Comparison of the above three:

Open addressing is a collision handling technique used in hashing where, when a collision occurs (i.e., when two or more keys map to the same slot), the algorithm looks for another empty slot in the hash table to store the collided key.

- In **linear probing**, the algorithm simply looks for the next available slot in the hash table and places the collided key there. If that slot is also occupied, the algorithm continues searching for the next available slot until an empty slot is found. This process is repeated until all collided keys have been stored. Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
- In **quadratic probing**, the algorithm searches for slots in a more spaced-out manner. When a collision occurs, the algorithm looks for the next slot using an equation that involves the original hash value and a quadratic function. If that slot is also occupied, the algorithm increments the value of the quadratic function and tries again. This process is repeated until an

empty slot is found. Quadratic probing lies between the two in terms of cache performance and clustering.

- In **double hashing**, the algorithm uses a second hash function to determine the next slot to check when a collision occurs. The algorithm calculates a hash value using the original hash function, then uses the second hash function to calculate an offset. The algorithm then checks the slot that is the sum of the original hash value and the offset. If that slot is occupied, the algorithm increments the offset and tries again. This process is repeated until an empty slot is found. Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

The choice of collision handling technique can have a significant impact on the performance of a hash table. Linear probing is simple and fast, but it can lead to clustering (i.e., a situation where keys are stored in long contiguous runs) and can degrade performance. Quadratic probing is more spaced out, but it can also lead to clustering and can result in a situation where some slots are never checked. Double hashing is more complex, but it can lead to more even distribution of keys and can provide better performance in some cases.

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 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

Note: Cache performance of chaining is not good because when we traverse a Linked List, we are basically jumping from one node to another, all across the computer’s memory. For this reason, the CPU cannot cache the nodes which aren’t visited yet, this doesn’t help us. But with Open Addressing, data isn’t spread, so if the CPU detects that a segment of memory is constantly being accessed, it gets cached for quick access.

Insert key 72 in the hash table

What is meant by Load Factor in Hashing?

The load factor of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. Load factor is the decisive parameter that is used when we want to rehash the previous hash function or want to add more elements to the existing hash table.

It helps us in determining the efficiency of the hash function i.e. it tells whether the hash function which we are using is distributing the keys uniformly or not in the hash table.

Load Factor = Total elements in hash table/ Size of hash table

What is Rehashing?

As the name suggests, rehashing means hashing again. Basically, when the load factor increases to more than its predefined value (the default value of the load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double-sized array to maintain a low load factor and low complexity.

Applications of Hash Data structure

- Hash is used in databases for indexing.
- Hash is used in disk-based data structures.
- In some programming languages like Python, JavaScript hash is used to implement objects.

Real-Time Applications of Hash Data structure

- Hash is used for cache mapping for fast access to the data.
- Hash can be used for password verification.
- Hash is used in cryptography as a message digest.
- Rabin-Karp algorithm for pattern matching in a string.
- Calculating the number of different substrings of a string.

Advantages of Hash Data structure

- Hash provides better synchronization than other data structures.
- Hash tables are more efficient than search trees or other data structures
- Hash provides constant time for searching, insertion, and deletion operations on average.

Disadvantages of Hash Data structure

- Hash is inefficient when there are many collisions.
- Hash collisions are practically not avoided for a large set of possible keys.
- Hash does not allow null values.

Hash Functions and list

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.

A **Hash Function** is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function **maps** a significant number or string to a small integer that can be used as the **index** in the hash table.

The pair is of the form **(key, value)**, where for a given key, one can find a value using some kind of a "function" that maps keys to values. The key for a given object can be calculated using a function called a hash function. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i].

Types of Hash functions

There are many hash functions that use numeric or alphanumeric keys. This article focuses on discussing different hash functions:

1. **Division Method.**
2. **Mid Square Method.**
3. **Folding Method.**
4. **Multiplication Method.**

Let's begin discussing these methods in detail.

1. **Division Method:**This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

h(K) = k mod M

Here,
k is the key value, and
M is the size of the hash table.

It is best suited that **M** is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

k = 12345
M = 95
h(12345) = 12345 mod 95
= 90

k = 1276
M = 11
h(1276) = 1276 mod 11
= 0

Pros:

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:

1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

2. Mid Square Method:

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k^2
2. Extract the middle **r** digits as the hash value.

Formula:

h(K) = h(k x k)

Here,
k is the key value.

The value of **r** can be decided based on the size of the table.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60
k x k = 60 x 60
      = 3600
h(60) = 60

The hash value obtained is 60

Pros:

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

1. The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

3. Digit Folding Method:

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:

k = k1, k2, k3, k4, ….., kn
s = k1+ k2 + k3 + k4 +….+ kn
h(K)= s

Here,
s is obtained by adding the parts of the key **k**

Example:

k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3
  = 12 + 34 + 5
  = 51
h(K) = 51

Note:
The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

4. Multiplication Method

This method involves the following steps:

1. Choose a constant value A such that 0 < A < 1.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:

h(K) = floor (M (kA mod 1))

Here,
M is the size of the hash table.
k is the key value.
A is a constant value.

Example:

k = 12345
A = 0.357840
M = 100

h(12345) = floor[100 (12345*0.357840 mod 1)]
  = floor[100 (4417.5348 mod 1)]
  = floor[100 (0.5348)]
  = floor[53.48]
  = 53

Pros:

The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

Cons:

The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

Sorting

Sorting is a fundamental operation in data structures. It enables efficient data storage and retrieval, making it an important part of many applications. In this article, we will discuss different sorting techniques in data structures and how they differ from each other. We will look at the challenges of sorting in data structure and explore the various sorting algorithms that can be used for efficient data structure manipulation. By the end of the article, you will have a better understanding of sorting techniques in data structures.

What is Sorting in Data Structure?

Sorting techniques in data structures is a process of rearranging data elements in an array or list in order to make it easier to search and retrieve. By sorting in data structure, the complexity of searching for a particular item is reduced. For instance, searching the entire list would take too long if you have an unsorted list of 10 items. However, searching for an item would be much faster if the same list is sorted. Various types of Sorting in data structure can also be used to compare two items and determine which one should come first in a sequence.

For example, consider an array A = {A1, A2, A3, A4, ?? An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > ? > An .

Consider an array;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by using which, sorting can be performed. In this section of the tutorial, we will discuss each method in detail.

Sorting techniques in data structure can also help produce information quickly by arranging the data elements according to certain criteria. This helps to identify relevant data from the dataset quickly.

Sorting in data structure comes under the Business Analytics and Data Science domain and helps to derive meaningful insights from the data.

Types of Sorting Techniques in Data Structure

Several sorting techniques in data structure can be used to sort data elements in an array or list. The most common types of sorting in data structure are insertion sort, selection sort, bubble sort, quick sort, heap sort, and merge sort.

- ### Quick sort

Quick sort is a sorting algorithm using the divide and conquer approach. It works by selecting a pivot element from the array or list, then partitioning the elements around the pivot into two subsets. This reduces the size of each sub-set and makes it easier to find the next pivot element. The process is repeated until all elements are sorted.

- ### Bubble Sort

Bubble Sort is among the types of sorting in data structure algorithms that helps in comparing adjacent elements and swaps them if and when they are in the wrong order. This process is repeated until all elements are sorted.

- **Merge sort**

Merge sort is a sorting algorithm based on the divide and conquer approach. It works by splitting the array into two halves, sorting in data structure each half in ascending or descending order, and then merging them. When this process is repeated for each sub-array, the entire array becomes sorted.

- **Insertion Sort**

Insertion Sort is among the various types of sorting in data structure sorting algorithms that work by inserting each element of the array or list into its correct position. This sorting technique starts at the beginning of the array, takes the second element, and then compares it with the first element. If it is smaller than the first element, then they are swapped. The same process is repeated for all other elements until the array is sorted.

- **Selection Sort**

Selection Sort is a sorting algorithm that selects the smallest or largest element from an unsorted array and places it at the beginning of the array. This process continues until all elements in the array are sorted. Although similar to Bubble Sort, Selection Sort is more efficient for sorting large datasets.

- **Heap sort**

Heapsort is a sorting in data structure algorithm based on the heap data structure. It works by creating a max or min heap from an unsorted array, then removing the root element and placing it at the end of the sorted list.

- **Radix Sort**

Radix Sort is a sorting algorithm that groups elements into buckets based on their numerical value. It starts with the least significant digit and then moves to the most significant digit. This process is repeated until all digits are sorted, resulting in an array of sorted elements. Radix Sort is an efficient sorting technique for large datasets.

- **Bucket Sort**

Bucket Sort is a sorting algorithm that divides the elements into buckets and then sorts each bucket. It uses a hash function to determine which bucket an element belongs to, then sorts each bucket with another sorting algorithm. This process is repeated until all elements in the array are sorted.

Moreover, sorting techniques in data structure and IoT play an important role in the Industry 4.0 era, as they help to process large volumes of data quickly and accurately. When you know what is IoT and its role in sorting in data structure, you can easily identify the advantages of incorporating it into your business.

Scope of Sorting technique in Data Structure

Sorting techniques in data structures are widely used in data structures and algorithms, providing an efficient way to store and retrieve data. Different types of sorting in data structure can be implemented depending on the dataset size and the type of data being sorted. Each sorting technique has advantages and disadvantages, so choosing the one that best fits the task is important.

Sorting Technique	**Scope**
Quick Sort	Efficient sorting in data structure algorithm for large datasets
Bubble Sort	Simple to understand and implement but unsuitable for large datasets

Merge Sort	Divide-and-Conquer approach, efficient for larger datasets
Insertion Sort	Works by inserting each element into its correct position, suitable for small datasets
Selection Sort	Selects the smallest or largest element from an unsorted array and places it at the beginning of the array. Suitable for large datasets
Heap Sort	Based on the heap data structure, efficient for large datasets
Radix Sort	Groups elements into buckets based on their numerical value. Efficient sorting in data structure technique for large datasets
Bucket Sort	Divides elements into buckets and then sorts each bucket with another sorting algorithm. Efficient sorting in data structure and the technique for large datasets.

Overall, sorting algorithms are important for data structures and can be used to store and retrieve data efficiently. Different algorithms have different scopes and suitability, so choosing the one that best fits the task is important. Data can be efficiently sorted and stored using the right sorting technique in data structure.

Moreover, queue-data-structures are also used for sorting techniques in data structure. Queues are a linear data structure that follows the First-in-First-Out (FIFO) principle.

Challenges Faced: Sorting In Data Structures

Sorting techniques in data structures can be challenging, especially when dealing with large datasets. Several challenges can arise when sorting in data structure:

Challenge	**Description**
Time complexity	Complexity of sorting algorithms depends on the number of elements in the dataset and how they are organized. Sorting algorithms can take a long time to complete when dealing with large datasets
Memory complexity	Sorting algorithms may require a lot of memory, depending on the size of the dataset. Large datasets may require more memory for sorting than is available
Computational complexity	Some sorting algorithms may be complex and difficult to understand, making them difficult to implement and debug
Data representation	The data being sorted must be represented in a way that makes sorting efficient. Different types of data may require different sorting algorithms for efficient sorting
Performance	Depending on the size of the dataset, some sorting algorithms may perform better than others. It is important to choose the appropriate sorting algorithm to ensure efficient sorting.

Bubble Sort

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.

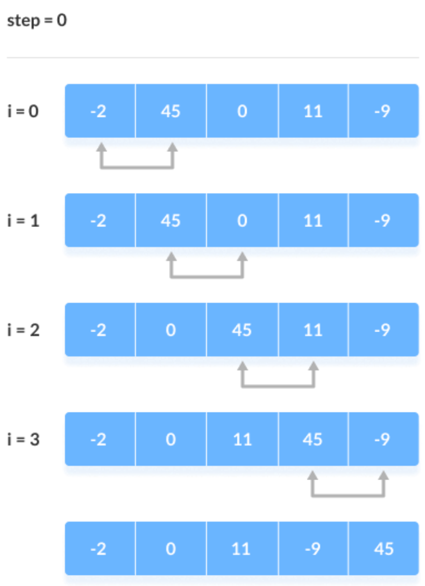
Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration. Therefore, it is called a bubble sort.

Working of Bubble Sort

Suppose we are trying to sort the elements in **ascending order**.

1. First Iteration (Compare and Swap)

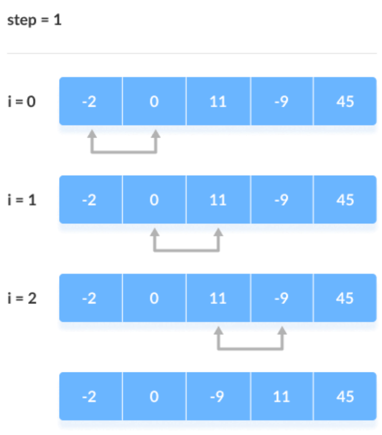
1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.



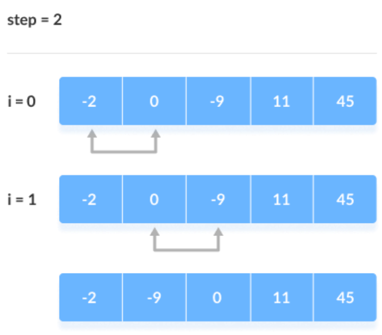
2. Remaining Iteration

The same process goes on for the remaining iterations.

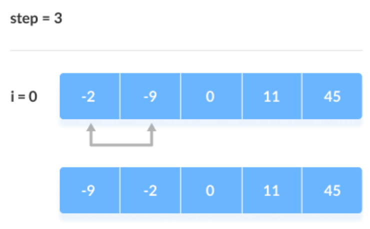
After each iteration, the largest element among the unsorted elements is placed at the end.



In each iteration, the comparison takes place up to the last unsorted element.



The array is sorted when all the unsorted elements are placed at their correct positions.



Bubble Sort Algorithm

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Bubble Sort Code in C

```
#include <stdio.h>

// perform the bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size - 1; ++step) {

    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping occurs if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find the array's length
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  printf("Sorted Array in Ascending Order:\n");
  printArray(data, size);
}
```

Optimized Bubble Sort Algorithm

In the above algorithm, all the comparisons are made even if the array is already sorted.

This increases the execution time.

To solve this, we can introduce an extra variable *swapped*. The value of *swapped* is set true if there occurs swapping of elements. Otherwise, it is set **false**.

After an iteration, if there is no swapping, the value of *swapped* will be **false**. This means elements are already sorted and there is no need to perform further iterations.

This will reduce the execution time and helps to optimize the bubble sort.

Algorithm for optimized bubble sort is

```
bubbleSort(array)
  swapped <- false
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
      swapped <- true
end bubbleSort
```

Optimized Bubble Sort in C

```
#include

// perform the bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size - 1; ++step) {

    // check if swapping occurs
    int swapped = 0;

    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two array elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {
        // swapping occurs if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;

        swapped = 1;
      }
    }

    // no swapping means the array is already sorted
    // so no need for further comparison
    if (swapped == 0) {
      break;
    }

  }
}

// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

// main method
int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find the array's length
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  printf("Sorted Array in Ascending Order:\n");
  printArray(data, size);
}
```

Bubble Sort Complexity

Time Complexity	
Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)
Stability	Yes

Complexity in Detail

Bubble Sort compares the adjacent elements.

Cycle	**Number of Comparisons**
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
....	......
Last	1

Hence, the number of comparisons is

(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2

nearly equals to n^2

Hence, **Complexity:** *$O(n^2)$*

Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is n*n = n^2

1. Time Complexities

- **Worst Case Complexity:** $O(n^2)$
  If we want to sort in ascending order and the array is in descending order then the worst case occurs.
- **Best Case Complexity:** O(n)
  If the array is already sorted, then there is no need for sorting.
- **Average Case Complexity:** $O(n^2)$
  It occurs when the elements of the array are in jumbled ord

- er (neither ascending nor descending).

2. Space Complexity

- Space complexity is O(1) because an extra variable is used for swapping.
- In the **optimized bubble sort algorithm**, two extra variables are used. Hence, the space complexity will be O(2).

Bubble Sort Applications

Bubble sort is used if

- complexity does not matter
- short and simple code is preferred

Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

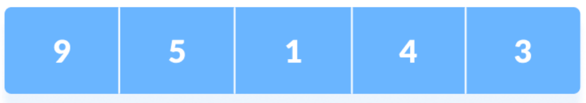
Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

Working of Insertion Sort

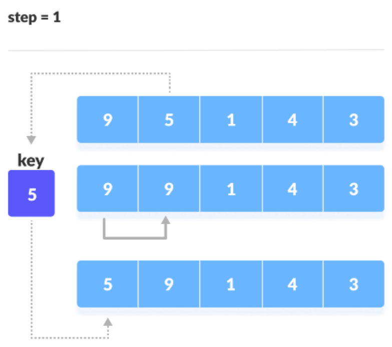
Suppose we need to sort the following array.



Initial array

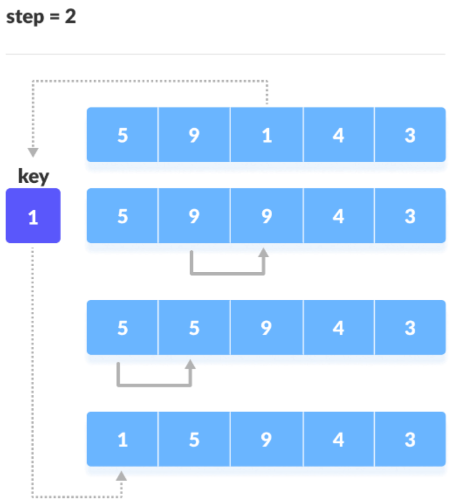
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

   Compare key with the first element. If the first element is greater than key, then key is

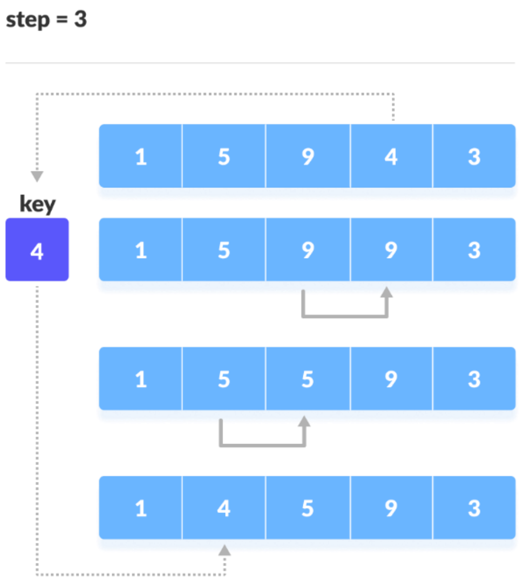
   placed in front of the first element. If the first element is greater than key, then key is placed in front of the first element.
2. Now, the first two elements are sorted.

Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

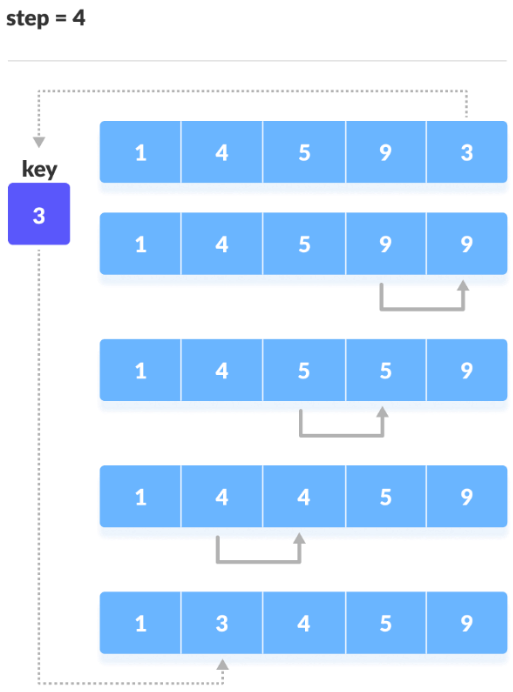


Place 1 at the beginning

3. Similarly, place every unsorted element at its correct position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

Insertion Sort Algorithm

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

Insertion Sort in C

```c
#include <stdio.h>

// Function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", array[i]);
  }
  printf("\n");
}

void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;

    // Compare key with each element on the left of it until an element smaller than
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

// Driver code
int main() {
  int data[] = {9, 5, 1, 4, 3};
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  printf("Sorted array in ascending order:\n");
  printArray(data, size);
}
```

Insertion Sort Complexity

Time Complexity	
Best	O(n)
Worst	O(n^2)

Average	O(n^2)
Space Complexity	O(1)
Stability	Yes

Time Complexities

- **Worst Case Complexity:** O(n^2)
  Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst-case complexity occurs.

  Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

  Thus, the total number of comparisons = $n*(n-1) \sim n^2$
- **Best Case Complexity:** O(n)
  When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.
- **Average Case Complexity:** O(n^2)
  It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Space Complexity

Space complexity is O(1) because an extra variable key is used.

Insertion Sort Applications

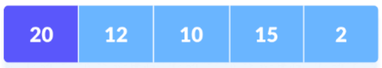
The insertion sort is used when:

- the array is has a small number of elements
- there are only a few elements left to be sorted

<u>Selection Sort Algorithm</u>

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

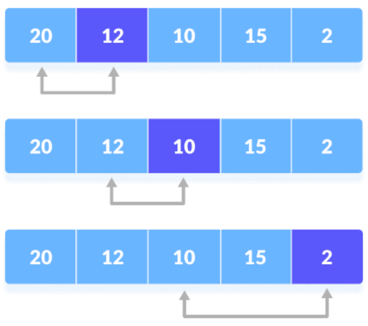
Working of Selection Sort

1. Set the first element as minimum. 

Select first element as minimum

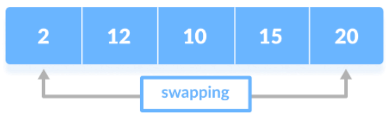
2. Compare `minimum` with the second element. If the second element is smaller than `minimum`, assign the second element as `minimum`.

   Compare `minimum` with the third element. Again, if the third element is smaller, then assign `minimum` to the third element otherwise do nothing. The process goes on until the last element.



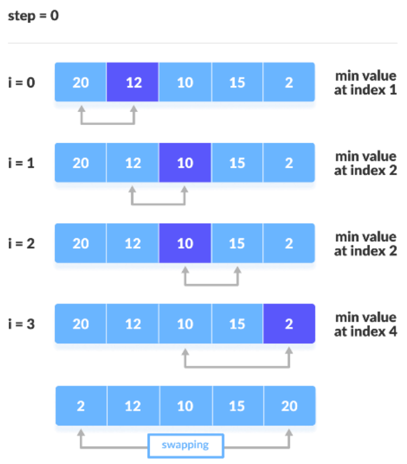
Compare minimum with the remaining elements

3. After each iteration, `minimum` is placed in the front of the unsorted list.

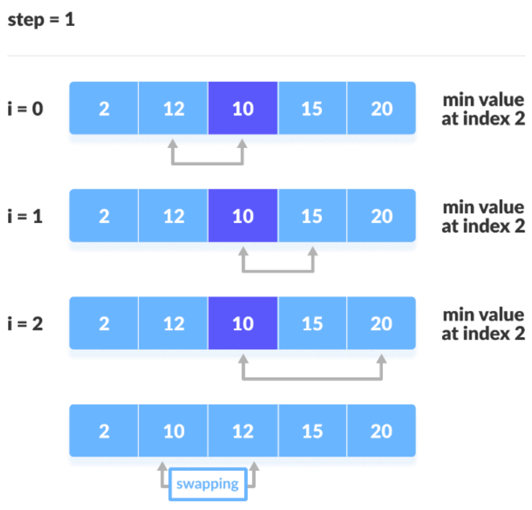


Swap the first with minimum

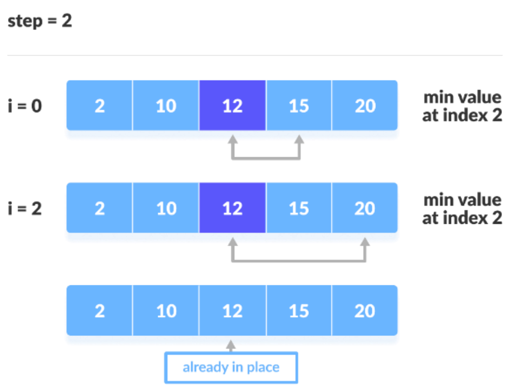
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



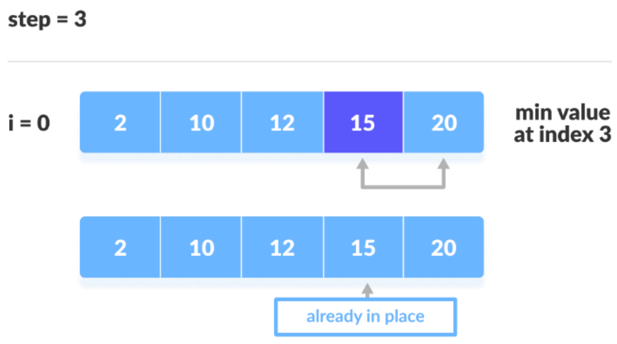
The first iteration



The second iteration



The third iteration



The fourth iteration

Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

```c
#include <stdio.h>
int main() {
   int arr[10]={6,12,0,18,11,99,55,45,34,2};
   int n=10;
   int i, j, position, swap;
   for (i = 0; i < (n - 1); i++) {
      position = i;
      for (j = i + 1; j < n; j++) {
         if (arr[position] > arr[j])
            position = j;
      }
      if (position != i) {
         swap = arr[i];
         arr[i] = arr[position];
         arr[position] = swap;
      }
   }
   for (i = 0; i < n; i++)
      printf("%d\t", arr[i]);
   return 0;
}
```

Output

```
0 2 6 11 12 18 34 45 55 99
```

Selection Sort Complexity

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	$O(1)$
Stability	No

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
Last	1

Number of comparisons: $(n - 1) + (n - 2) + (n - 3) + ... + 1 = n(n - 1) / 2$ nearly equals to n^2.

Complexity = $O(n^2)$

Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is $n*n = n^2$.

Time Complexities:

- **Worst Case Complexity:** $O(n^2)$

  If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

- **Best Case Complexity:** $O(n^2)$

  It occurs when the array is already sorted

- **Average Case Complexity:** $O(n^2)$

  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

  The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.

  Space Complexity:

  Space complexity is $O(1)$ because an extra variable `temp` is used.

Selection Sort Applications

The selection sort is used when

- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is $O(n)$ as compared to $O(n^2)$ of bubble sort)

Quicksort Algorithm

Quicksort is a sorting algorithm based on the **divide and conquer approach** where

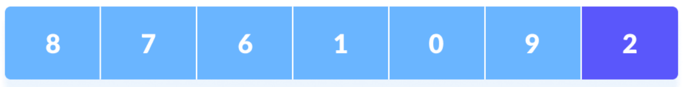
1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

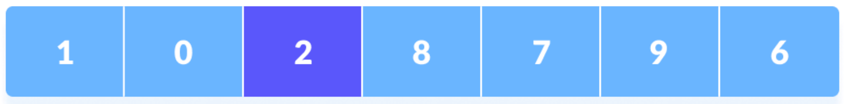
There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.



Select a pivot element

2. Rearrange the Array

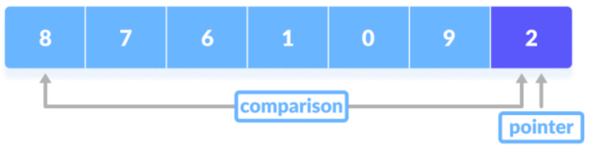
Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.



Put all the smaller elements on the left and greater on the right of pivot element

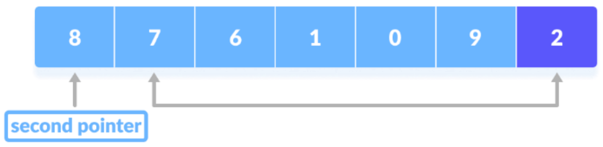
Here's how we rearrange the array:

1. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



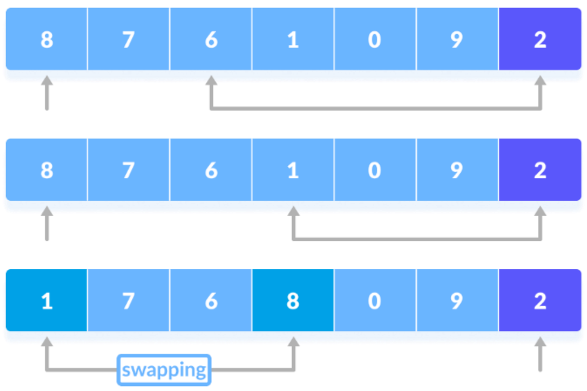
Comparison of pivot element with element beginning from the first index

2. If the element is greater than the pivot element, a second pointer is set for that element.



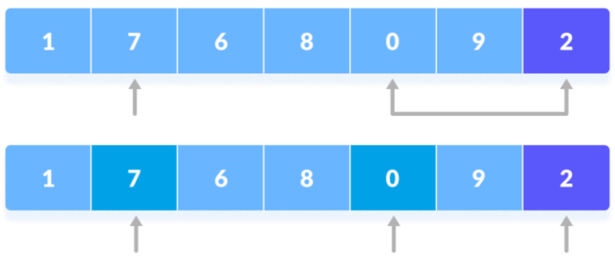
If the element is greater than the pivot element, a second pointer is set for that element.

3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



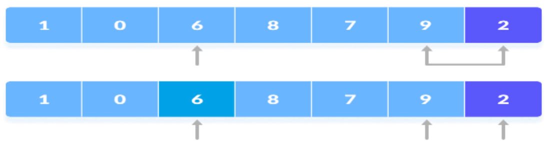
Pivot is compared with other elements.

4. Again, the process is repeated to set the next greater element as the second pointer. And,



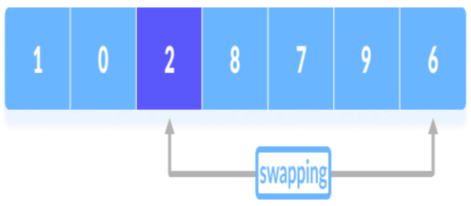
swap it with another smaller element. The process is repeated to set the next greater element as the second pointer.

5. The process goes on until the second last element is reached.



The process goes on until the second last element is reached.

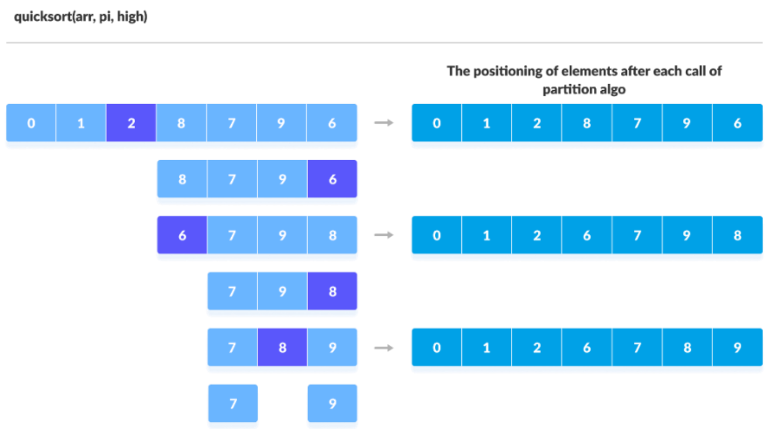
6. Finally, the pivot element is swapped with the second pointer.



Finally, the pivot element is swapped with the second pointer.

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.


quicksort(arr, pi, high)
The positioning of elements after each call of partition algo

Select pivot element of in each half and put at correct place using recursion

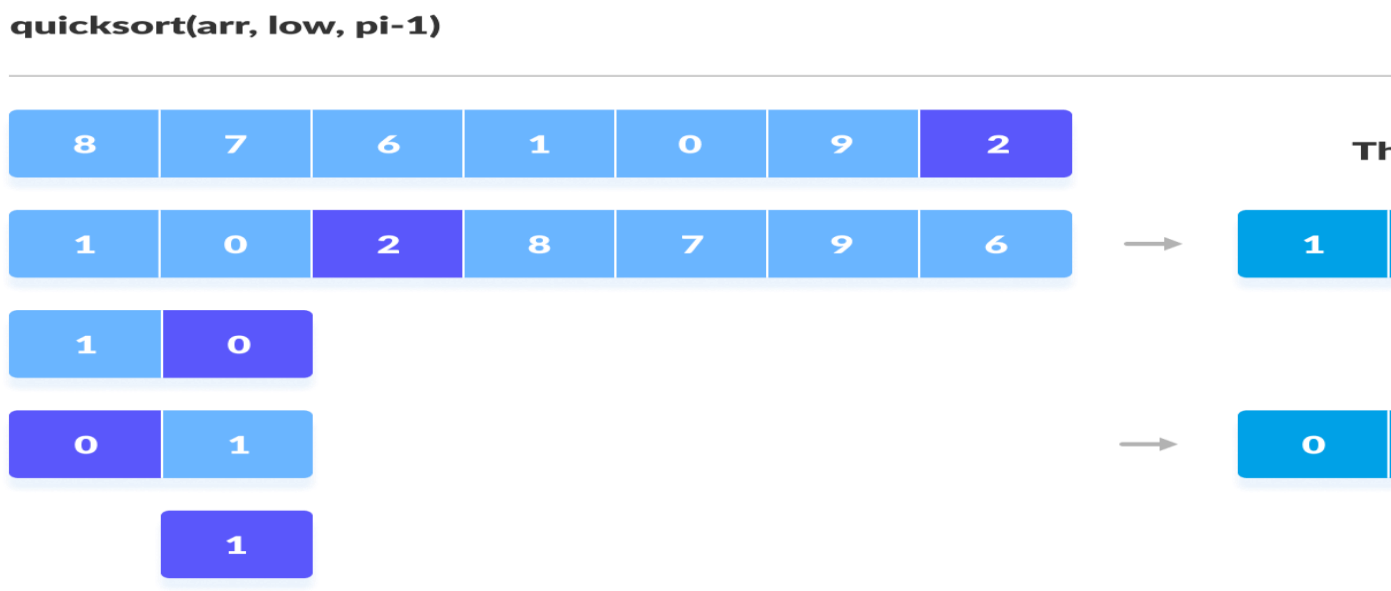
The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Quick Sort Algorithm

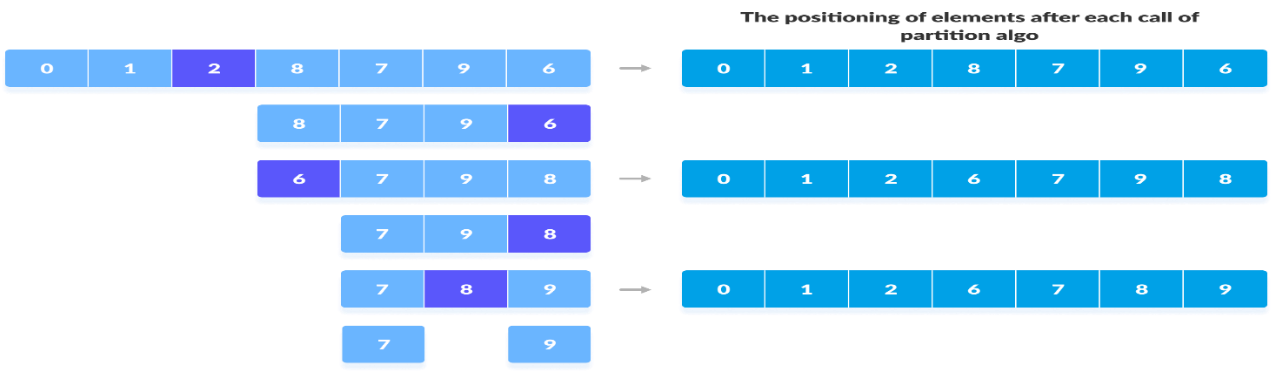
```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Visual Illustration of Quicksort Algorithm

You can understand the working of quicksort algorithm with the help of the illustrations below.



Sorting the elements on the left of pivot using recursion quicksort(arr, pi+1, high)



Sorting the elements on the right of pivot using recursion

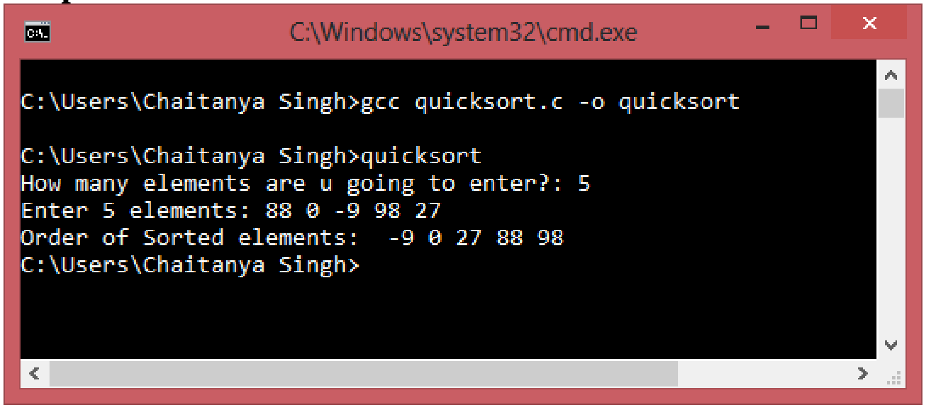
Quicksort program in C

Quicksort is a divide and conquer algorithm. The steps are: 1) Pick an element from the array, this element is called as pivot element. 2) Divide the unsorted array of elements in two arrays with values less than the pivot come in the first sub array, while all elements with values greater than the pivot come in the second sub-array (equal values can go either way). This step is called the partition operation. 3) Recursively repeat the step 2(until the sub-arrays are sorted) to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values. The same logic we have implemented in the following C program.

C Program – Quicksort algorithm implementation

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
  int i, j, pivot, temp;

  if(first<last){
    pivot=first;
    i=first;
    j=last;

    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
        i++;
      while(number[j]>number[pivot])
        j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }

    temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);

  }
}

int main(){
  int i, count, number[25];

  printf("How many elements are u going to enter?: ");
  scanf("%d",&count);

  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);

  quicksort(number,0,count-1);

  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);

  return 0;
}
```

Output:



```
C:\Users\Chaitanya Singh>gcc quicksort.c -o quicksort

C:\Users\Chaitanya Singh>quicksort
How many elements are u going to enter?: 5
Enter 5 elements: 88 0 -9 98 27
Order of Sorted elements:  -9 0 27 88 98
C:\Users\Chaitanya Singh>
```

Quicksort Complexity

Time Complexity	
Best	O(n*log n)
Worst	O(n^2)
Average	O(n*log n)
Space Complexity	O(log n)
Stability	No

1. Time Complexities

- **Worst Case Complexity [Big-O]**: O(n^2)

It occurs when the pivot element picked is either the greatest or the smallest element.

This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains `n - 1` elements. Thus, quicksort is called only on this sub-array.

However, the quicksort algorithm has better performance for scattered pivots.

- **Best Case Complexity [Big-omega]**: `O(n*log n)`

  It occurs when the pivot element is always the middle element or near to the middle element.
- **Average Case Complexity [Big-theta]**: `O(n*log n)`

  It occurs when the above conditions do not occur.

2. Space Complexity

The space complexity for quicksort is `O(log n)`.

Quicksort Applications

Quicksort algorithm is used when

- the programming language is good for recursion
- time complexity matters
- space complexity matters