

Searching Algorithms

Searching algorithms are methods or procedures used to find a specific item or element within a collection of data. These algorithms are widely used in computer science and are crucial for tasks like searching for a particular record in a database, finding an element in a sorted list, or locating a file on a computer.

These are some commonly used searching algorithms:

1. **Linear Search:** In this simple algorithm, each element in the collection is sequentially checked until the desired item is found, or the entire list is traversed. It is suitable for small-sized or unsorted lists, but its time complexity is $O(n)$ in the worst case.
2. **Binary Search:** This algorithm is applicable only to sorted lists. It repeatedly compares the middle element of the list with the target element and narrows down the search range by half based on the comparison result. Binary search has a time complexity of $O(\log n)$, making it highly efficient for large sorted lists.
3. **Hashing:** Hashing algorithms use a hash function to convert the search key into an index or address of an array (known as a hash table). This allows for constant-time retrieval of the desired item if the hash function is well-distributed and collisions are handled appropriately. Common hashing techniques include direct addressing, separate chaining, and open addressing.
4. **Interpolation Search:** Similar to binary search, interpolation search works on sorted lists. Instead of always dividing the search range in half, interpolation search uses the value of the target element and the values of the endpoints to estimate its approximate position within the list. This estimation helps in quickly narrowing down the search space. The time complexity of interpolation search is typically $O(\log \log n)$ on average if the data is uniformly distributed.
5. **Tree-based Searching:** Various tree data structures, such as binary search trees (BST), AVL trees, or B-trees, can be used for efficient searching. These structures impose an ordering on the elements and provide fast search, insertion, and deletion operations. The time complexity of tree-based searching algorithms depends on the height of the tree and can range from $O(\log n)$ to $O(n)$ in the worst case.

6. **Ternary Search:** Ternary search is an algorithm that operates on sorted lists and repeatedly divides the search range into three parts instead of two, based on two splitting points. It is a divide-and-conquer approach and has a time complexity of $O(\log_3 n)$.
7. **Jump Search:** Jump search is an algorithm for sorted lists that works by jumping ahead a fixed number of steps and then performing linear search in the reduced subarray. It is useful for large sorted arrays and has a time complexity of $O(\sqrt{n})$, where n is the size of the array.
8. **Exponential Search:** Exponential search is a technique that combines elements of binary search and linear search. It begins with a small range and doubles the search range until the target element is within the range. It then performs a binary search within that range. Exponential search is advantageous when the target element is likely to be found near the beginning of the array and has a time complexity of $O(\log n)$.
9. **Fibonacci Search:** Fibonacci search is a searching algorithm that uses Fibonacci numbers to divide the search space. It works on sorted arrays and has a similar approach to binary search, but instead of dividing the array into halves, it divides it into two parts using Fibonacci numbers as indices. Fibonacci search has a time complexity of $O(\log n)$.
10. **Interpolation Search for Trees:** This algorithm is an extension of interpolation search designed for tree structures such as AVL trees or Red-Black trees. It combines interpolation search principles with tree traversal to efficiently locate elements in the tree based on their values. The time complexity depends on the tree structure and can range from $O(\log n)$ to $O(n)$ in the worst case.
11. **Hash-based Searching (e.g., Bloom Filter):** Hash-based searching algorithms utilize hash functions and data structures like Bloom filters to determine whether an element is present in a set or not. These algorithms provide probabilistic answers, meaning they can occasionally have false positives (indicating an element is present when it is not), but no false negatives (if an element is not present, it will never claim it is). Bloom filters have a constant-time complexity for search operations.

Linear Search Algorithm

Two popular search methods are Linear Search and Binary Search. So, here we will discuss the popular searching technique, i.e., Linear Search Algorithm.

Linear search is also called as **sequential search algorithm**. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **$O(n)$** .

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

First, we have to traverse the array elements using a **for** loop.

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

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

Algorithm

1. Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search
2. Step 1: set **pos** = -1
3. Step 2: set **i** = 1
4. Step 3: repeat step 4 while **i** <= n
5. Step 4: if **a[i]** == val
6. set **pos** = **i**
7. print pos
8. go to step 6
9. [end of if]
10. set **ii** = **i** + 1

11. [end of loop]
12. Step 5: if **pos** = -1
13. print "value is not present in the array "
14. [end of if]
15. Step 6: exit

Working of Linear search

Now, let's see the working of the linear search Algorithm.

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

Let the elements of array are -

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

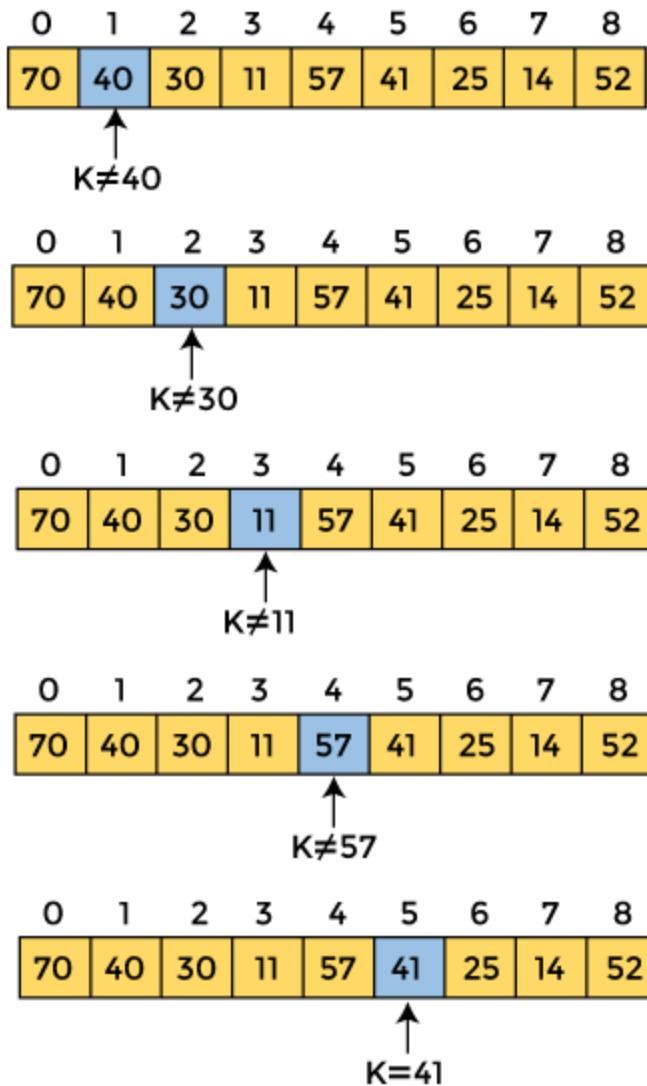
Let the element to be searched is **K = 41**

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

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

↑
K ≠ 70

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



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

Linear Search complexity

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

1. Time Complexity

Case

Time Complexity

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

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

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

2. Space Complexity

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

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

Advantages of Linear Search:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
- Does not require any additional memory.
- It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

- Linear search has a time complexity of $O(N)$, which in turn makes it slow for large datasets.
- Not suitable for large arrays.

Sequential Search

- 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 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.

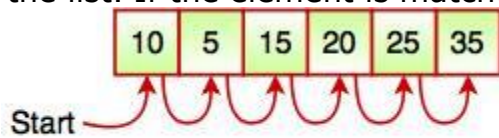


Fig. Sequential Search

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. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

Binary Search Algorithm

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

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

Algorithm

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

Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

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

There are two methods to implement the binary search algorithm -

- Iterative method
- Recursive method

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

Let the elements of array are -

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

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

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

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

So, in the given array -

$$\text{beg} = 0$$

$$\text{end} = 8$$

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

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



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

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



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

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



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

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

Binary Search complexity

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

1. Time Complexity

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

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

2. Space Complexity

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

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

For example, if searching an element 25 in the 7-element array, following figure shows how binary search works:

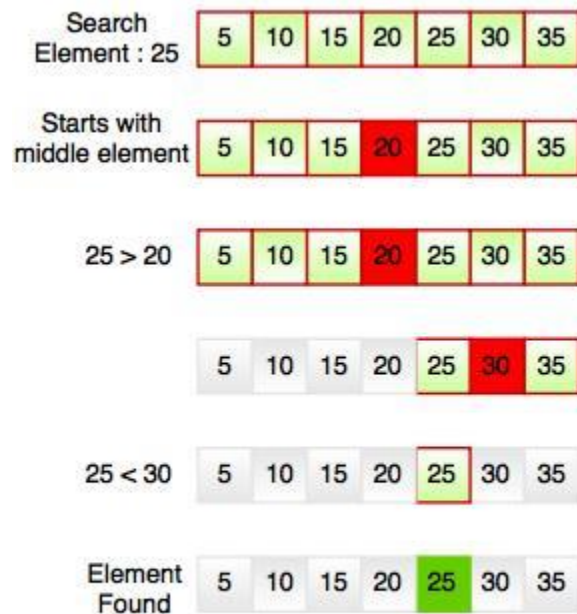


Fig. Working Structure of Binary Search

Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

Hashing in Data Structure

Introduction to Hashing in Data Structure:

Hashing is a popular technique in computer science that involves mapping large data sets to fixed-length values. It is a process of converting a data set of variable size into a data set of a fixed size. The ability to perform efficient lookup operations makes hashing an essential concept in data structures.

What is Hashing?

A hashing algorithm is used to convert an input (such as a string or integer) into a fixed-size output (referred to as a hash code or hash value). The data is then stored and retrieved using this hash value as an index in an array or hash table. The hash function must be deterministic, which guarantees that it will always yield the same result for a given input.

Hashing is commonly used to create a unique identifier for a piece of data, which can be used to quickly look up that data in a large dataset. For example, a web browser may use hashing to store website passwords securely. When a user enters their password, the browser converts it into a hash value and compares it to the stored hash value to authenticate the user.

What is a hash Key?

In the context of hashing, a hash key (also known as a hash value or hash code) is a fixed-size numerical or alphanumeric representation generated by a hashing algorithm. It is derived from the input data, such as a text string or a file, through a process known as hashing.

Hashing involves applying a specific mathematical function to the input data, which produces a unique hash key that is typically of fixed length, regardless of the size of the input. The resulting hash key is essentially a digital fingerprint of the original data.

The hash key serves several purposes. It is commonly used for data integrity checks, as even a small change in the input data will produce a significantly different hash key. Hash keys are also used for efficient data retrieval and storage in hash tables or data structures, as they allow quick look-up and comparison operations.

Examples of Hashing in Data Structure

The following are real-life examples of **hashing in the data structure** –

- In schools, the teacher assigns a unique roll number to each student. Later, the teacher uses that roll number to retrieve information about that student.
- A library has an infinite number of books. The librarian assigns a unique number to each book. This unique number helps in identifying the position of the books on the bookshelf.

How Hashing Works?

The process of hashing can be broken down into three steps:

- Input: The data to be hashed is input into the hashing algorithm.
- Hash Function: The hashing algorithm takes the input data and applies a mathematical function to generate a fixed-size hash value. The hash function should be designed so that different input values produce different hash values, and small changes in the input produce large changes in the output.
- Output: The hash value is returned, which is used as an index to store or retrieve data in a data structure.

How does Hashing in Data Structure Works?

In hashing, the hashing function maps strings or numbers to a small integer value. Hash tables retrieve the item from the list using a hashing function. The objective of hashing technique is to distribute the data evenly across an array. Hashing assigns all the elements a unique key. The hash table uses this key to access the data in the list.

Hash table stores the data in a key-value pair. The key acts as an input to the hashing function. Hashing function then generates a unique index number for each value stored. The index number keeps the value that corresponds to that key. The

hash function returns a small integer value as an output. The output of the hashing function is called the hash value.

Let us understand **hashing in a data structure** with an example. Imagine you need to store some items (arranged in a key-value pair) inside a hash table with 30 cells.

The values are: (3,21) (1,72) (40,36) (5,30) (11,44) (15,33) (18,12) (16,80) (38,99)

The hash table will look like the following:

Serial Number	Key	Hash	Array Index
1	3	$3\%30 = 3$	3
2	1	$1\%30 = 1$	1
3	40	$40\%30 = 10$	10
4	5	$5\%30 = 5$	5
5	11	$11\%30 = 11$	11

6	15	$15\%30 = 15$	15
7	18	$18\%30 = 18$	18
8	16	$16\%30 = 16$	16
9	38	$38\%30 = 8$	8

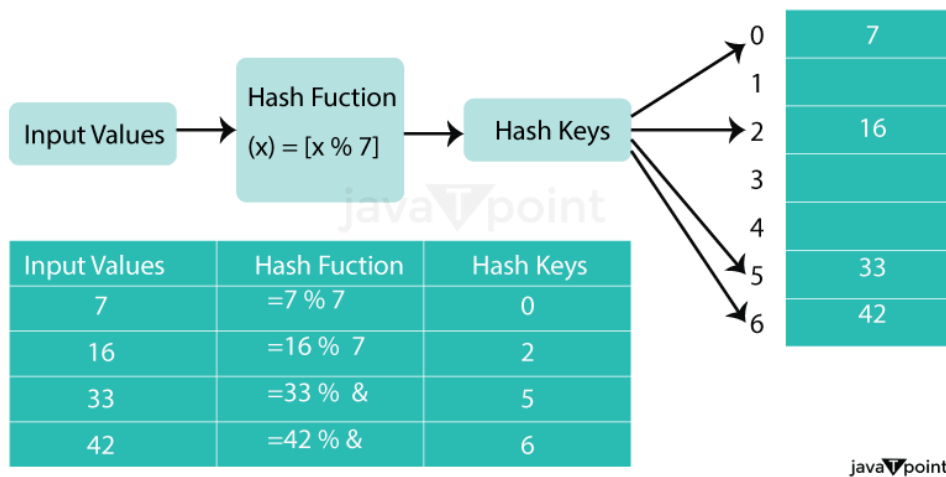
The process of taking any size of data and then converting that into smaller data value which can be named as hash value.

Hashing Algorithms:

There are numerous hashing algorithms, each with distinct advantages and disadvantages. The most popular algorithms include the following:

- MD5: A widely used hashing algorithm that produces a 128-bit hash value.
- SHA-1: A popular hashing algorithm that produces a 160-bit hash value.
- SHA-256: A more secure hashing algorithm that produces a 256-bit hash value.

Hashing Data Structure



Collision Resolution

One of the main challenges in hashing is handling collisions, which occur when two or more input values produce the same hash value. There are various techniques used to resolve collisions, including:

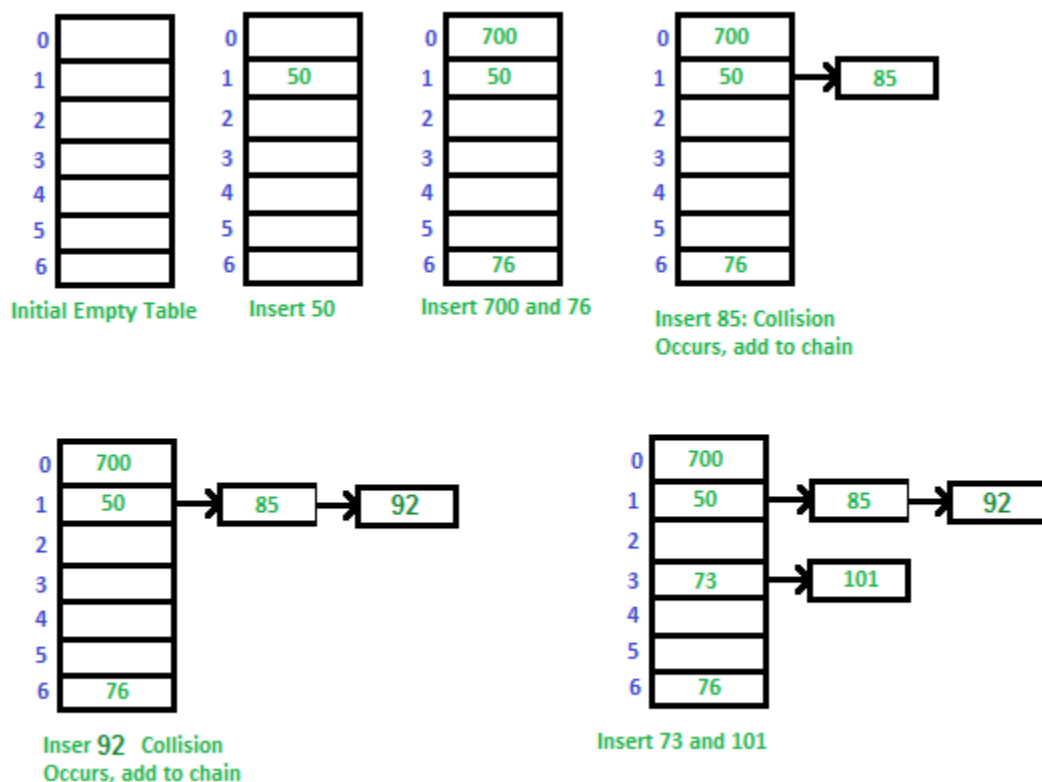
- Chaining: In this technique, each hash table slot contains a linked list of all the values that have the same hash value. This technique is simple and easy to implement, but it can lead to poor performance when the linked lists become too long.
- Open addressing: In this technique, when a collision occurs, the algorithm searches for an empty slot in the hash table by probing successive slots until an empty slot is found. This technique can be more efficient than chaining when the load factor is low, but it can lead to clustering and poor performance when the load factor is high.
- Double hashing: This is a variation of open addressing that uses a second hash function to determine the next slot to probe when a collision occurs. This technique can help to reduce clustering and improve performance.

Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

*The **linked list** data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain. Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.*

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



You can refer to the following link in order to understand how to implement separate chaining with C++.

Advantages:

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

Disadvantages:

Wastage of Space (Some Parts of the hash table are never used)

- If the chain becomes long, then search time can become $O(n)$ in the worst case
- Uses extra space for links

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: $\text{rehash}(\text{key}) = (n+1) \% \text{table-size}$.

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 $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1) \% S$

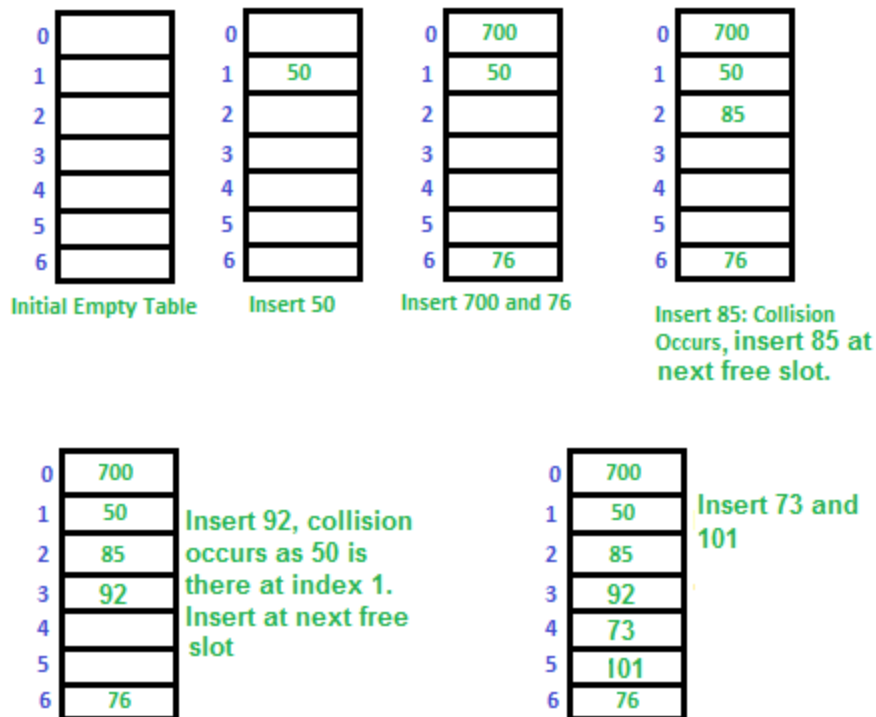
If $(\text{hash}(x) + 1) \% S$ is also full, then we try $(\text{hash}(x) + 2) \% S$

If $(\text{hash}(x) + 2) \% S$ is also full, then we try $(\text{hash}(x) + 3) \% S$

.....
.....

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 $\text{hash}(\text{key}) = \text{key} \% S$, here $S = \text{size of the table} = 7$, 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.



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:

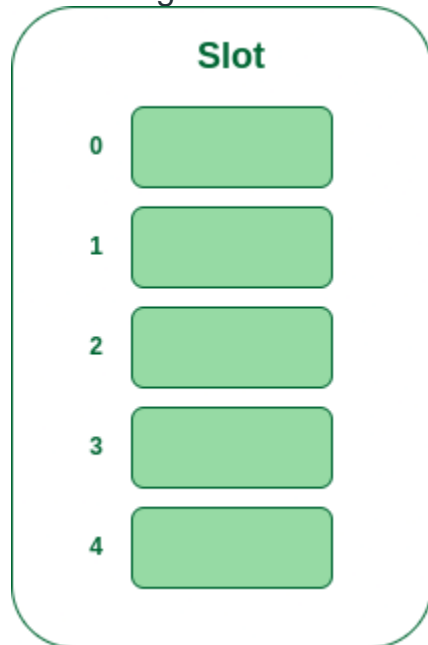
Caching: Linear probing can be used in caching systems to store frequently accessed data in memory. When a cache miss occurs, the data can be loaded into the cache using linear probing, and when a collision occurs, the next available slot in the cache can be used to store the data.

- **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.

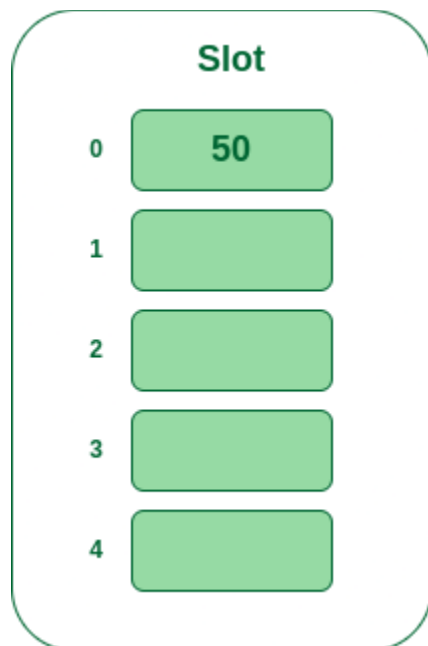
Example: Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 93.

- **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.



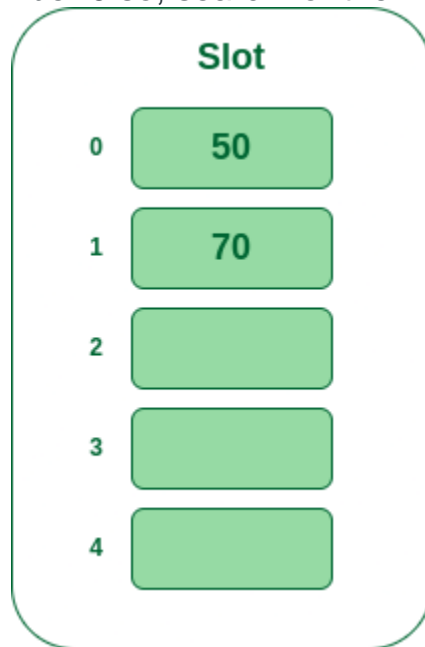
Hash table

- **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.



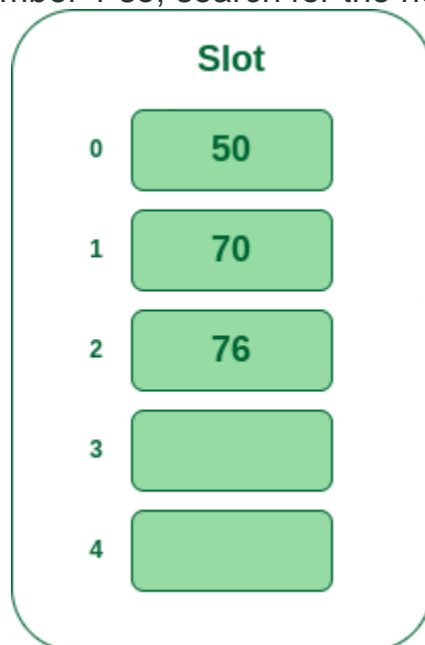
Insert 50 into hash table

- **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.



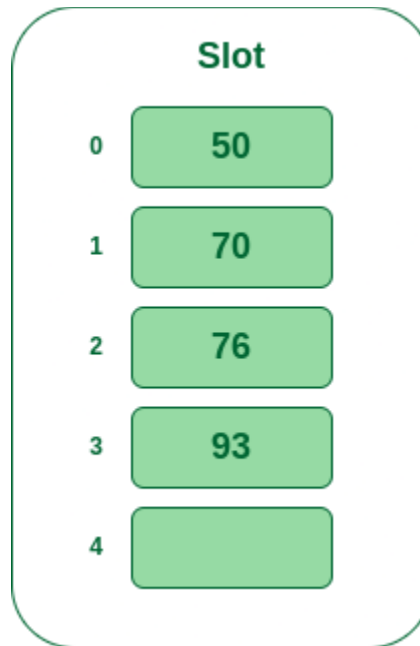
Insert 70 into hash table

- **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^{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 $\text{hash}(x)$ be the slot index computed using hash function.

*If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1*1) \% S$*

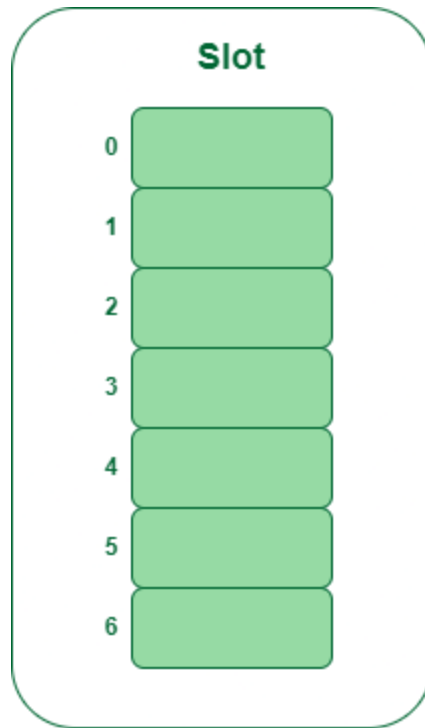
*If $(\text{hash}(x) + 1*1) \% S$ is also full, then we try $(\text{hash}(x) + 2*2) \% S$*

*If $(\text{hash}(x) + 2*2) \% S$ is also full, then we try $(\text{hash}(x) + 3*3) \% S$*

.....

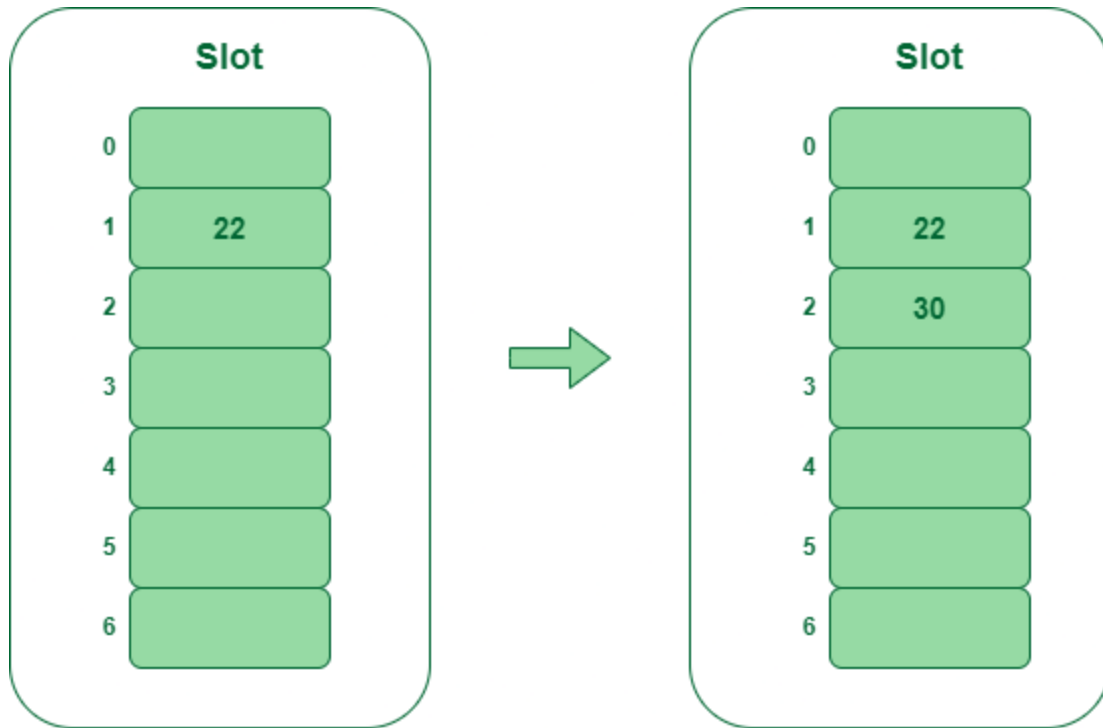
Example: Let us consider table Size = 7, hash function as $\text{Hash}(x) = x \% 7$ and collision resolution strategy to be $f(i) = i^2$. Insert = 22, 30, and 50.

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



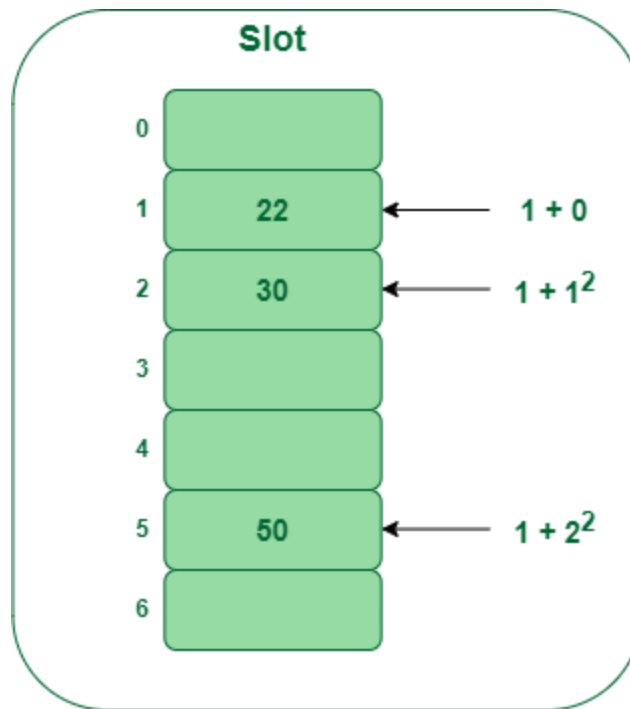
Hash table

- **Step 2** – Insert 22 and 30
 - $\text{Hash}(22) = 22 \% 7 = 1$, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
 - $\text{Hash}(30) = 30 \% 7 = 2$, 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
 - $\text{Hash}(50) = 50 \% 7 = 1$
 - 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

3. Double Hashing

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 $\text{hash2}(x)$ and look for the $i \cdot \text{hash2}(x)$ slot in the i^{th} rotation.

let $\text{hash}(x)$ be the slot index computed using hash function.

If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1 \cdot \text{hash2}(x)) \% S$

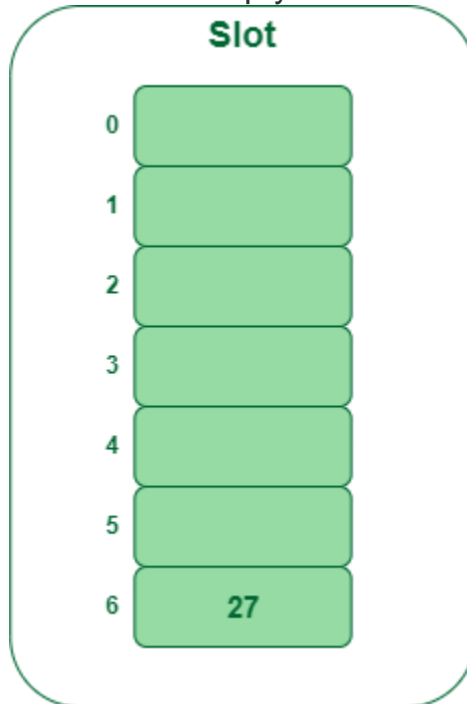
If $(\text{hash}(x) + 1 \cdot \text{hash2}(x)) \% S$ is also full, then we try $(\text{hash}(x) + 2 \cdot \text{hash2}(x)) \% S$

If $(\text{hash}(x) + 2 \cdot \text{hash2}(x)) \% S$ is also full, then we try $(\text{hash}(x) + 3 \cdot \text{hash2}(x)) \% S$

.....

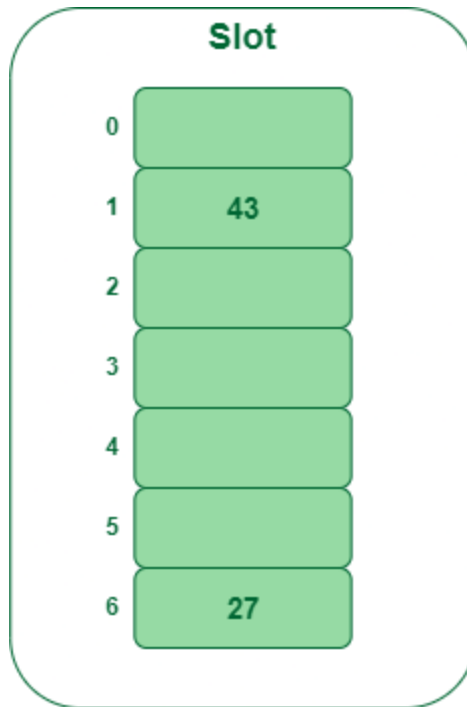
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is $h1(k) = k \bmod 7$ and second hash-function is $h2(k) = 1 + (k \bmod 5)$

- **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.

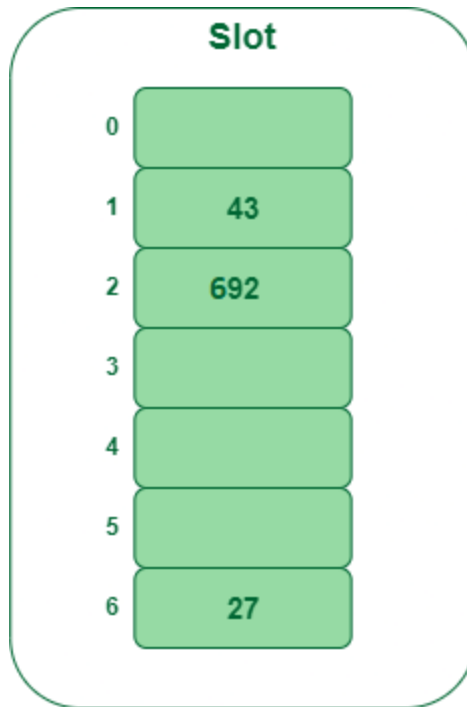


Insert key 43 in the hash table

- **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.

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

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

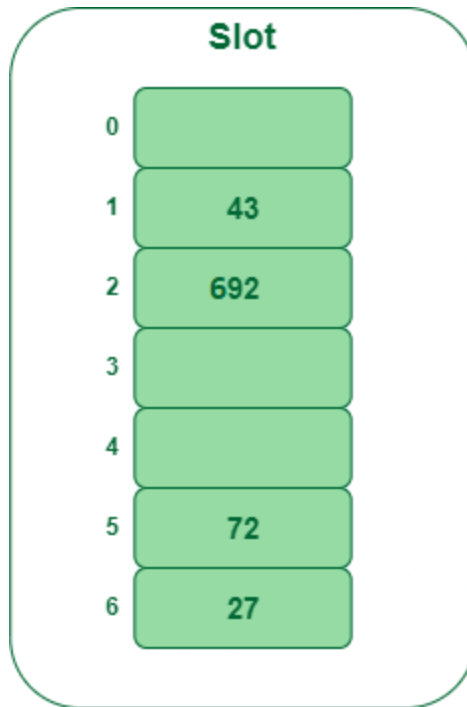


Insert key 692 in the hash table

- **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.

$$\begin{aligned}h_{new} &= [h1(72) + i * (h2(72))] \% 7 \\&= [2 + 1 * (1 + 72 \% 5)] \% 7 \\&= 5 \% 7 \\&= 5,\end{aligned}$$

*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.

Hash Table:

A hash table is a data structure that stores data in an array. Typically, a size for the array is selected that is greater than the number of elements that can fit in the hash table. A key is mapped to an index in the array using the hash function.

The hash function is used to locate the index where an element needs to be inserted in the hash table in order to add a new element. The element gets added to that index if there isn't a collision. If there is a collision, the collision resolution method is used to find the next available slot in the array.

The hash function is used to locate the index that the element is stored in order to retrieve it from the hash table. If the element is not found at that index, the collision resolution method is used to search for the element in the linked list (if chaining is used) or in the next available slot (if open addressing is used).

Hash Table Operations

There are several operations that can be performed on a hash table, including:

- Insertion: Inserting a new key-value pair into the hash table.
- Deletion: Removing a key-value pair from the hash table.
- Search: Searching for a key-value pair in the hash table.

Applications of Hashing

Hashing has many applications in computer science, including:

- Databases: Hashing is used to index and search large databases efficiently.
- Cryptography: Hash functions are used to generate message digests, which are used to verify the integrity of data and protect against tampering.
- Caching: Hash tables are used in caching systems to store frequently accessed data and improve performance.
- Spell checking: Hashing is used in spell checkers to quickly search for words in a dictionary.
- Network routing: Hashing is used in load balancing and routing algorithms to distribute network traffic across multiple servers.

Advantages of Hashing:

- Fast Access: Hashing provides constant time access to data, making it faster than other data structures like linked lists and arrays.
- Efficient Search: Hashing allows for quick search operations, making it an ideal data structure for applications that require frequent search operations.
- Space-Efficient: Hashing can be more space-efficient than other data structures, as it only requires a fixed amount of memory to store the hash table.

Introduction to Sorting Techniques – Data Structure and Algorithm

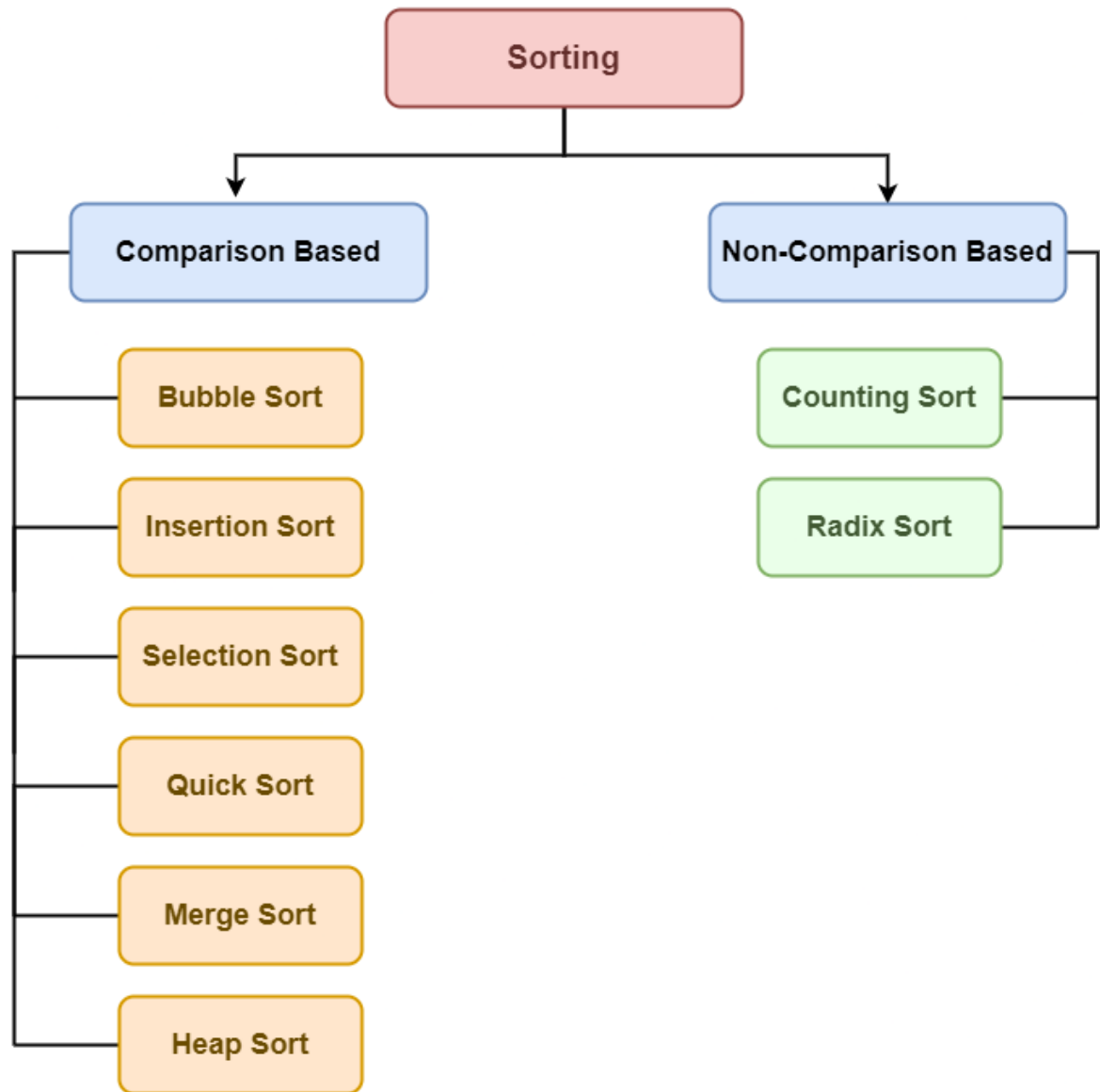
Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

When we have a large amount of data, it can be difficult to deal with it, especially when it is arranged randomly. When this happens, sorting that data becomes crucial. It is necessary to sort data in order to make searching easier.

Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)



Sorting algorithm

Why Sorting Algorithms are Important

The sorting algorithm is important in Computer Science because it reduces the complexity of a problem. There is a wide range of applications for these algorithms, including searching algorithms, database algorithms, divide and conquer methods, and data structure algorithms.

In the following sections, we list some important scientific applications where sorting algorithms are used

- When you have hundreds of datasets you want to print, you might want to arrange them in some way.
- Sorting algorithm is used to arrange the elements of a list in a certain order (either ascending or descending).
- Searching any element in a huge data set becomes easy. We can use Binary search method for search if we have sorted data. So, Sorting become important here.
- They can be used in software and in conceptual problems to solve more advanced problems.

Some of the most common sorting algorithms are:

Below are some of the most common sorting algorithms:

1. [Selection sort](#)

Selection sort is another sorting technique in which we find the minimum element in every iteration and place it in the array beginning from the first index. Thus, a selection sort also gets divided into a sorted and unsorted subarray.

Working of Selection Sort algorithm:

Lets consider the following array as an example: `arr[] = {64, 25, 12, 22, 11}`

First pass:

*For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing whole array it is clear that **11** is the lowest value.*

64	25	12	22	11
-----------	----	----	----	----

Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

11	25	12	22	64
-----------	----	----	----	----

Second Pass:

For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

11	25	12	22	64
----	-----------	----	----	----

After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

11	12	25	22	64
----	----	-----------	----	----

Third Pass:

Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.

11	12	25	22	64
----	----	-----------	----	----

While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.

11	12	22	25	64
----	----	-----------	----	----

Fourth pass:

Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array

As 25 is the 4th lowest value hence, it will place at the fourth position.

11	12	22	25	64
----	----	----	-----------	----

Fifth Pass:

At last the largest value present in the array automatically get placed at the last position in the array

The resulted array is the sorted array.

11	12	22	25	64
----	----	----	----	----

2. [Bubble sort](#)

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Working of Bubble Sort algorithm:

Lets consider the following array as an example: $arr[] = \{5, 1, 4, 2, 8\}$

First Pass:

Bubble sort starts with very first two elements, comparing them to check which one is greater.

(5 1 4 2 8) \rightarrow (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since $5 > 4$

(1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since $5 > 2$

(1 4 2 5 8) \rightarrow (1 4 2 5 8), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass:

Now, during second iteration it should look like this:

(1 4 2 5 8) \rightarrow (1 4 2 5 8)

(1 4 2 5 8) \rightarrow (1 2 4 5 8), Swap since $4 > 2$

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Third Pass:

Now, the array is already sorted, but our algorithm does not know if it is completed.

The algorithm needs one whole pass without any swap to know it is sorted.

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

3. [Insertion Sort](#)

Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part.

Values from the unsorted part are picked and placed at the correct position in the sorted part.

Working of Insertion Sort algorithm:

Consider an example: $arr[]: \{12, 11, 13, 5, 6\}$

12	11	13	5	6
-----------	-----------	-----------	----------	----------

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6
-----------	-----------	----	---	---

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6
-----------	-----------	----	---	---

Second Pass:

- Now, move to the next two elements and compare them

11	12	13	5	6
----	-----------	-----------	---	---

- Here, 13 is greater than 12, thus both elements seem to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

Third Pass:

- Now, two elements are present in the sorted sub-array which are 11 and 12
- Moving forward to the next two elements which are 13 and 5

11	12	13	5	6
----	----	-----------	----------	---

- Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6
----	----	----------	-----------	---

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6
----	---	----	----	---

- Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6
---	----	----	----	---

- here, it is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are 5, 11 and 12
- Moving to the next two elements 13 and 6

5	11	12	13	6
---	----	----	----	---

- Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13
---	----	----	---	----

- Now, 6 is smaller than 12, hence, swap again

5	11	6	12	13
---	----	---	----	----

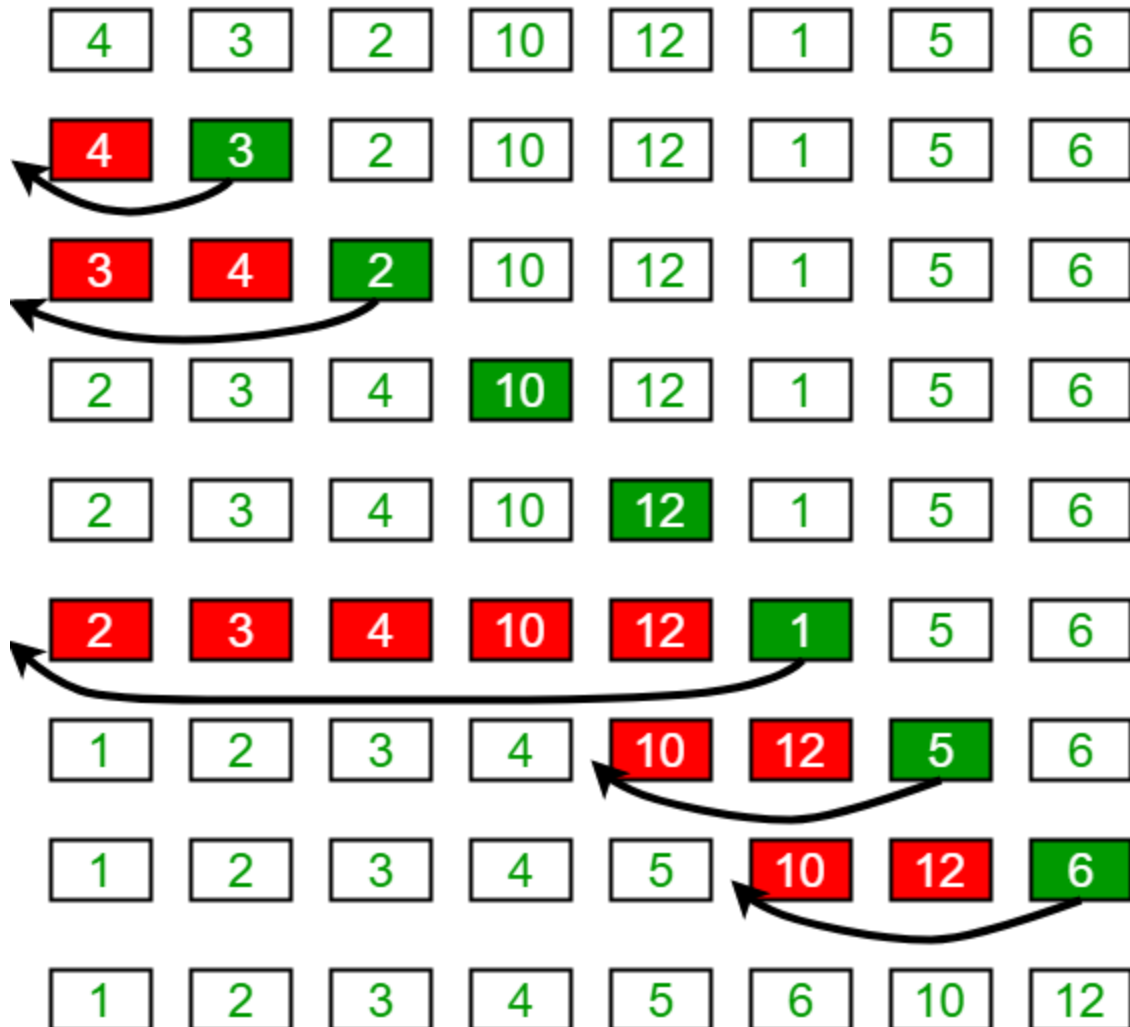
- Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13
---	---	----	----	----

- Finally, the array is completely sorted.

Illustrations:

Insertion Sort Execution Example



4. [Merge Sort](#)

The Merge Sort algorithm is a sorting algorithm that is based on the **Divide and Conquers** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Let's see how Merge Sort uses Divide and Conquer:

The merge sort algorithm is an implementation of the divide and conquers technique. Thus, it gets completed in three steps:

1. **Divide:** In this step, the array/list divides itself recursively into sub-arrays until the base case is reached.
2. **Conquer:** Here, the sub-arrays are sorted using recursion.
3. **Combine:** This step makes use of the `merge()` function to combine the sub-arrays into the final sorted array.

Working of Merge Sort algorithm:

To know the functioning of merge sort, lets consider an array `arr[] = {38, 27, 43, 3, 9, 82, 10}`

At first, check if the left index of array is less than the right index, if yes then calculate its mid point

l = Left Index

r = Right Index



**Is $l < r$
Yes
 $m = l + (r - 1) / 2$**

Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.

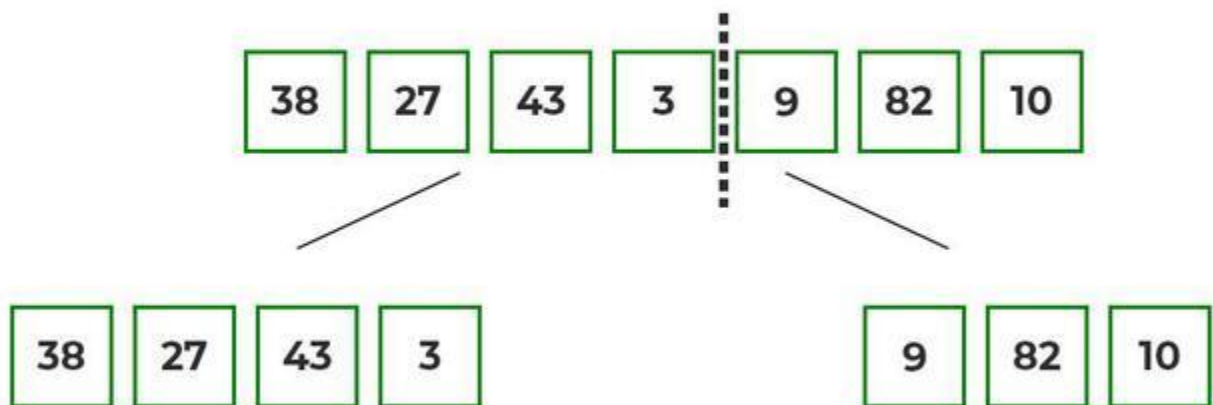
Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

l = Left Index

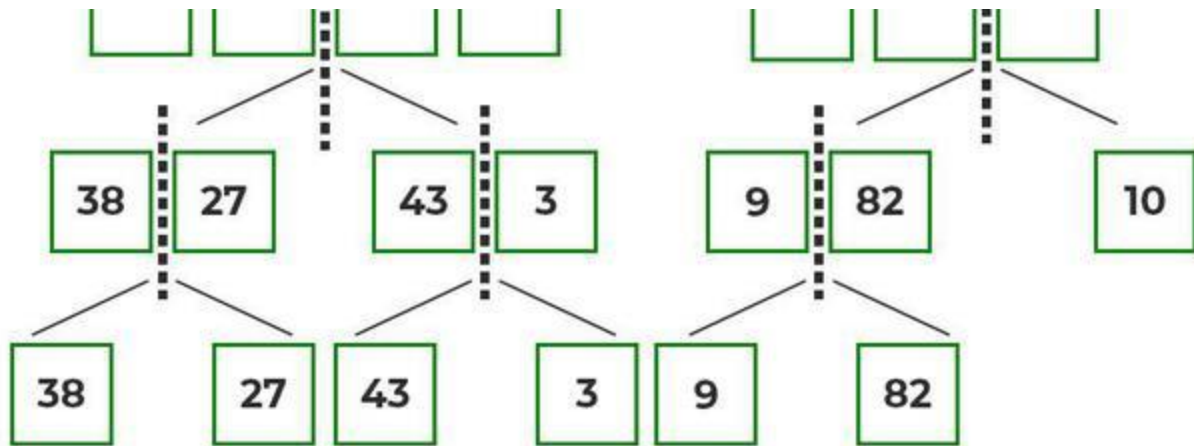
r = Right Index



Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



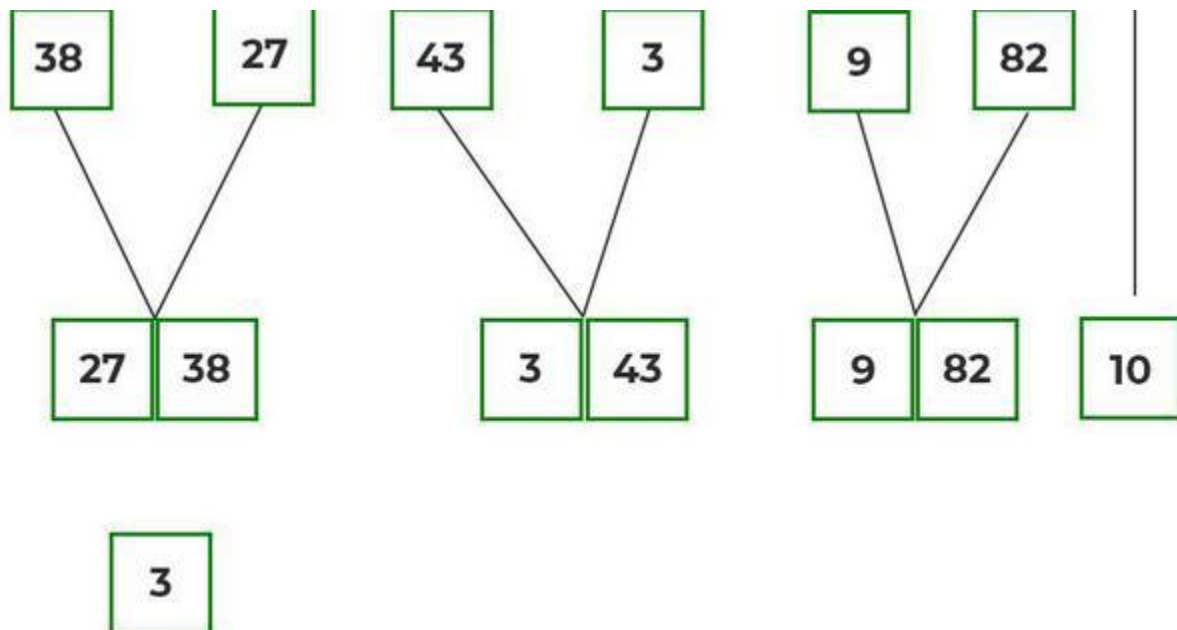
Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



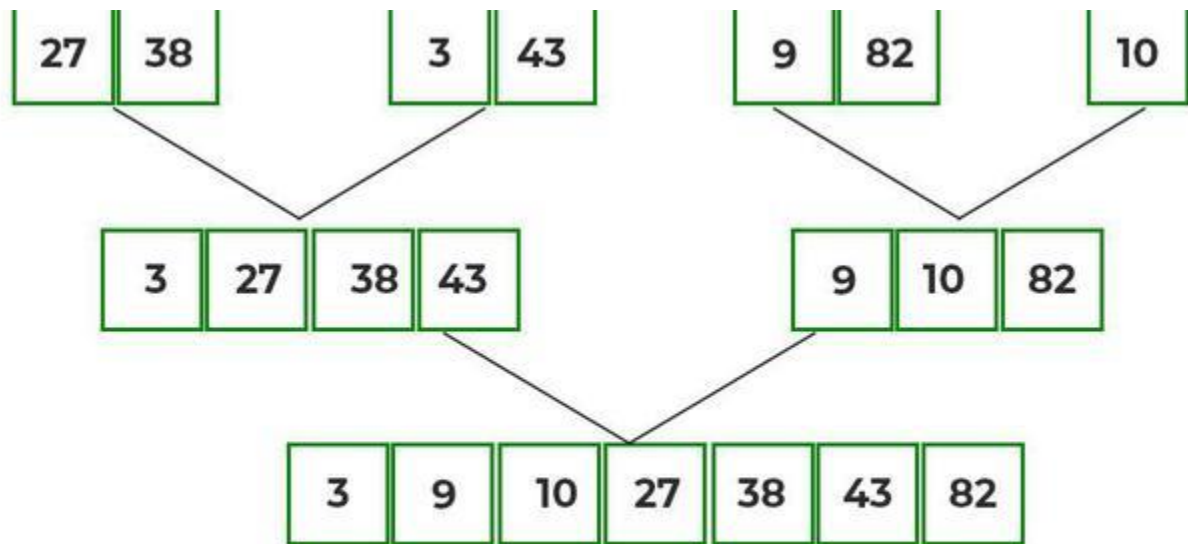
After dividing the array into smallest units merging starts, based on comparison of elements.

After dividing the array into smallest units, start merging the elements again based on comparison of size of elements

Firstly, compare the element for each list and then combine them into another list in a sorted manner.



After the final merging, the list looks like this:



5. [Quick sort](#)

Quicksort is a sorting algorithm based on the divide and conquer approach where an array is divided into subarrays by selecting a pivot element (element selected from the array).

1. While dividing the array, the pivot element should be positioned in such a way that elements less than the pivot are kept on the left side, and elements greater than the pivot is 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 Quick Sort algorithm:

To know the functioning of Quick sort, let's consider an array $arr[] = \{10, 80, 30, 90, 40, 50, 70\}$

- *Indexes: 0 1 2 3 4 5 6*
- *low = 0, high = 6, pivot = $arr[h] = 70$*
- *Initialize index of smaller element, $i = -1$*

Partition

10	80	30	90	40	50	70
----	----	----	----	----	----	----

↑
Pivot

Counter variables

I: Index of smaller element

J: Loop variable

We start the loop with initial values

Test Condition	Actions	Value of variables
$arr[J] \leq pivot$		$i = -1$ $J = 0$

Step1

- *Traverse elements from $j = low$ to $high-1$*
- *$j = 0$: Since $arr[j] \leq pivot$, do $i++$ and $swap(arr[i], arr[j])$*
- *$i = 0$*
- *$arr[] = \{10, 80, 30, 90, 40, 50, 70\}$ // No change as i and j are same*
- *$j = 1$: Since $arr[j] > pivot$, do nothing*

Partition



↑
Pivot

Counter variables

I: Index of smaller element

J: Loop variable

Pass 2

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 0 J = 1
80 < 70 false	No Action	

Step2

- $j = 2$: Since $arr[j] <= pivot$, do $i++$ and $swap(arr[i], arr[j])$
- $i = 1$
- $arr[] = \{10, 30, 80, 90, 40, 50, 70\}$ // We swap 80 and 30

Partition



↑
Pivot

Counter variables

I: Index of smaller element

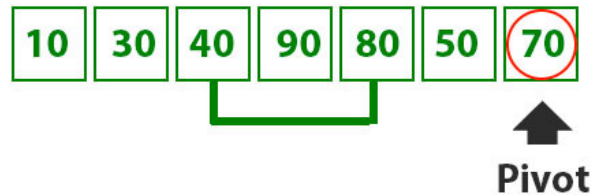
J: Loop variable

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 1 J = 2
30 < 70 true	$i++$ $Swap(arr[i], arr[j])$	

Step3

- $j = 3$: Since $arr[j] > pivot$, do nothing // No change in i and $arr[j]$
- $j = 4$: Since $arr[j] \leq pivot$, do $i++$ and $swap(arr[i], arr[j])$
- $i = 2$
- $arr[] = \{10, 30, 40, 90, 80, 50, 70\}$ // 80 and 40 Swapped

Partition



Counter variables

I: Index of smaller element

J: Loop variable

Pass 5

Test Condition	Actions	Value of variables
$arr[J] \leq pivot$ <div>40 < 70 true</div>	<div>$i++$ Swap($arr[i], arr[j]$)</div>	$i = 2$ $J = 4$

Step 4

- $j = 5$: Since $arr[j] \leq pivot$, do $i++$ and swap $arr[i]$ with $arr[j]$
- $i = 3$
- $arr[] = \{10, 30, 40, 50, 80, 90, 70\}$ // 90 and 50 Swapped

Partition



↑
Pivot

Counter variables

I: Index of smaller element

J: Loop variable

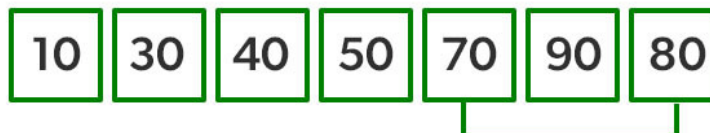
Before Pass 7, J becomes 6
so we come out of the loop

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 3 J = 6

Step 5

- We come out of loop because j is now equal to high-1.
- Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
- arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Partition



Counter Variable

I : Index of smaller element

J : Loop variable

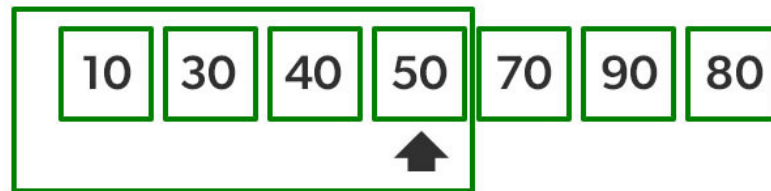
We know swap arr[i+1] and pivot

I = 3

Step 6

- Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.
- Since quick sort is a recursive function, we call the partition function again at left and right partitions

Quick sort left



Since quick sort is a recursion function, we call the Partition function again

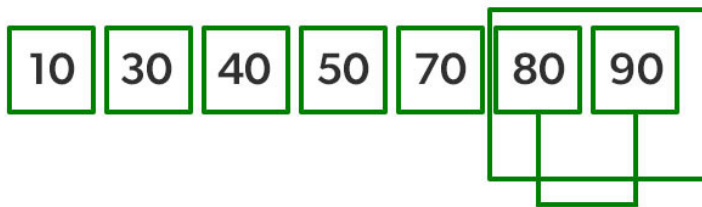
First 50 is the pivot.

As it is already at its correct position we call the quicksort function again on the left part.

Step 7

- Again call function at right part and swap 80 and 90

Quick sort Right



80 is the Pivot

80 and 90 are swapped to bring pivot to correct position

Step 8

What is heap sort?

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

Working of Heap sort Algorithm

Now, let's see the working of the Heapsort Algorithm.

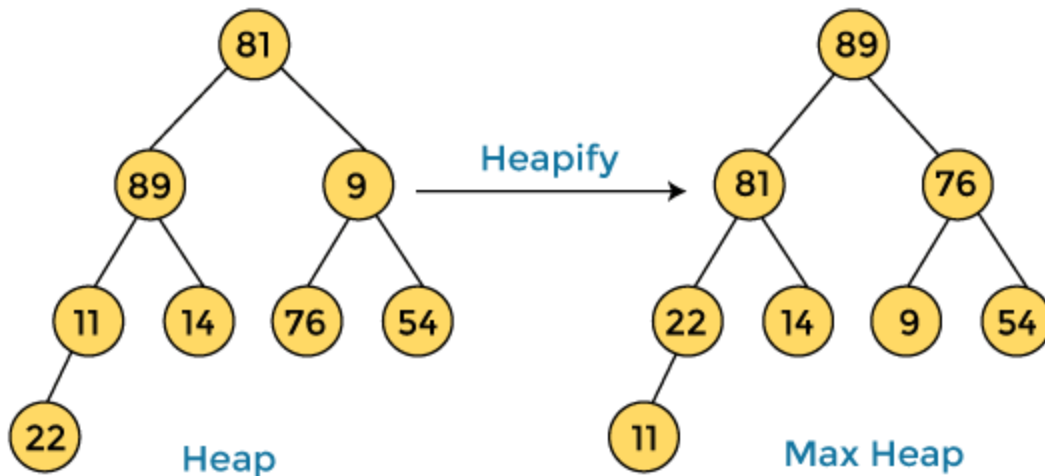
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

81	89	9	11	14	76	54	22
----	----	---	----	----	----	----	----

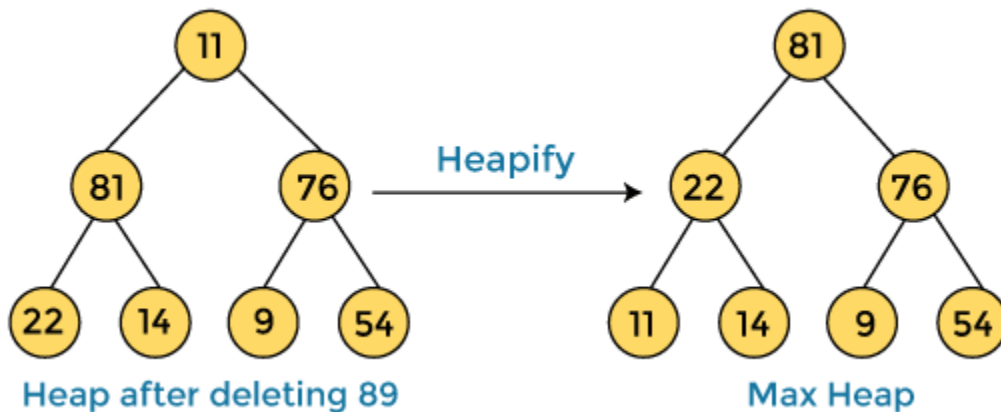
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11
----	----	----	----	----	---	----	----

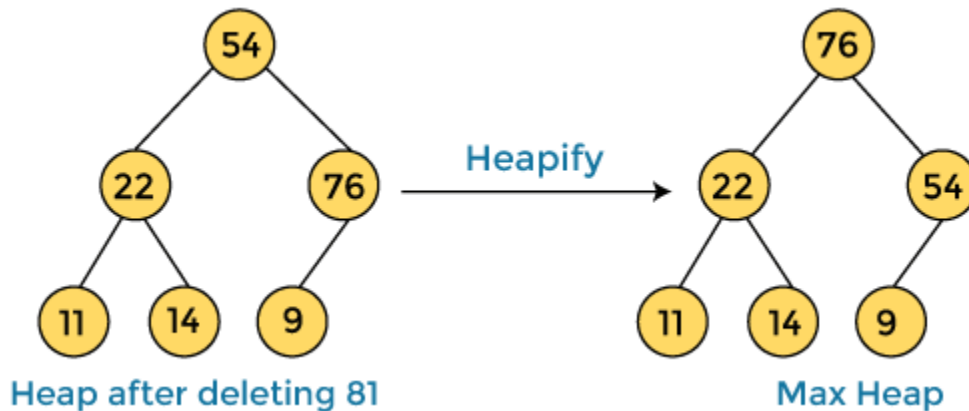
Next, we have to delete the root element (**89**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**11**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **89** with **11**, and converting the heap into max-heap, the elements of array are -

81	22	76	11	14	9	54	89
----	----	----	----	----	---	----	----

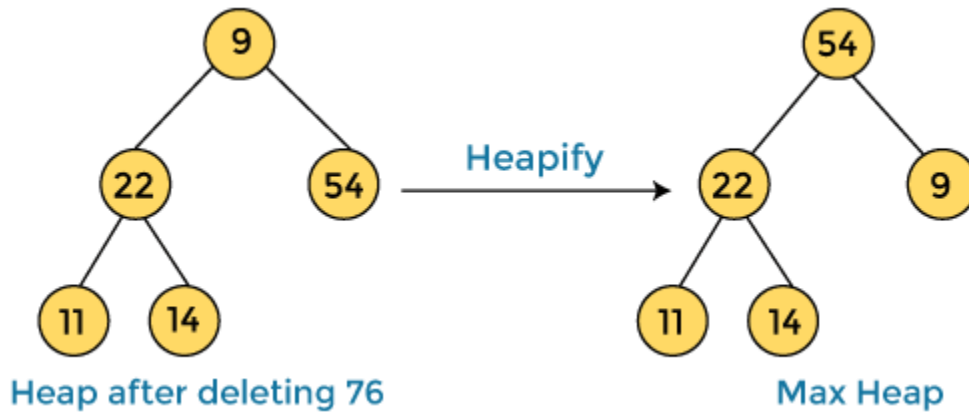
In the next step, again, we have to delete the root element (**81**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**54**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

76	22	54	11	14	9	81	89
----	----	----	----	----	---	----	----

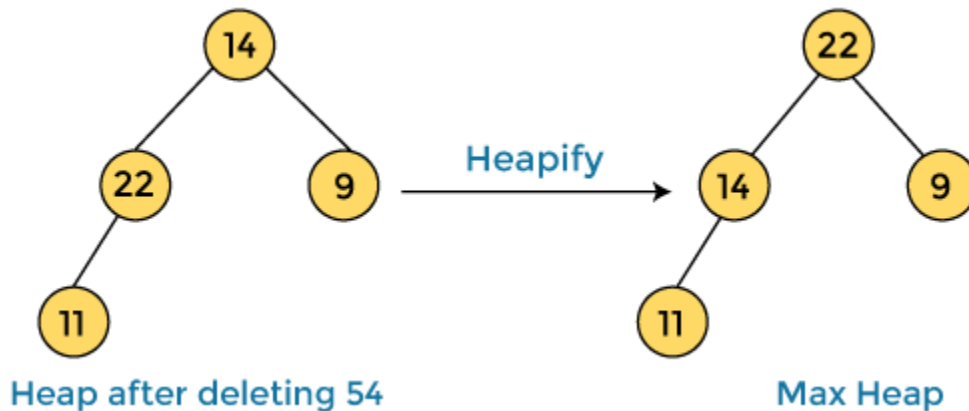
In the next step, we have to delete the root element (**76**) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

54	22	9	11	14	76	81	89
----	----	---	----	----	----	----	----

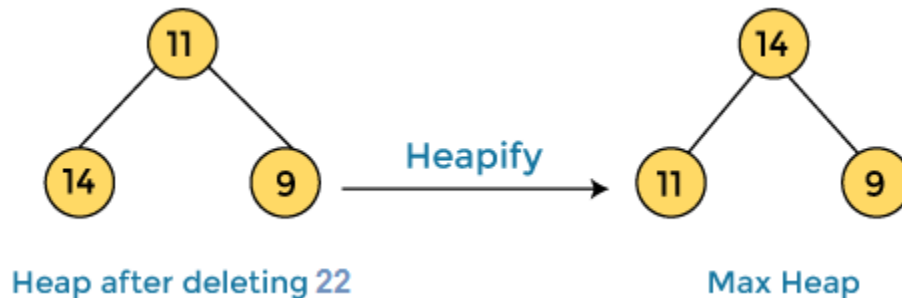
In the next step, again we have to delete the root element (**54**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**14**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

22	14	9	11	54	76	81	89
----	----	---	----	----	----	----	----

In the next step, again we have to delete the root element (**22**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**11**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89
----	----	---	----	----	----	----	----

In the next step, again we have to delete the root element (**14**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89
----	---	----	----	----	----	----	----

In the next step, again we have to delete the root element (**11**) from the max heap. To delete this node, we have to swap it with the last node, i.e. (**9**). After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **11** with **9**, the elements of array are -

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are -

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

Now, the array is completely sorted.

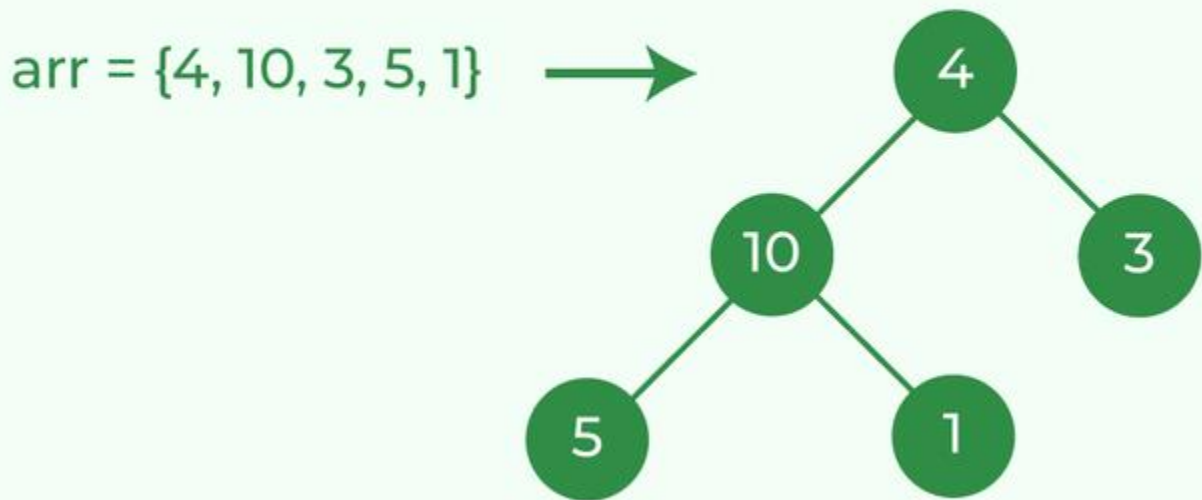
EXAMPLE 2

To understand heap sort more clearly, let's take an unsorted array and try to sort it using heap sort. Consider the array: $\text{arr}[] = \{4, 10, 3, 5, 1\}$.

Build Complete Binary Tree: Build a complete binary tree from the array.

Step 1

Build complete Binary Tree



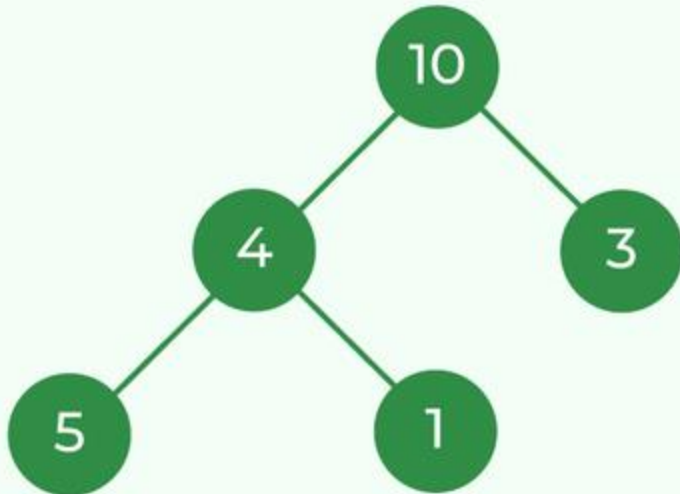
Build complete binary tree from the array

Transform into max heap: After that, the task is to construct a tree from that unsorted array and try to convert it into [max heap](#).

- To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes
 - Here, in this example, as the parent node 4 is smaller than the child node 10, thus, swap them to build a max-heap.
- Transform it into a max heap

Step 2

Make it a max heap (4 less than 5 & 10 is greater between the two children)

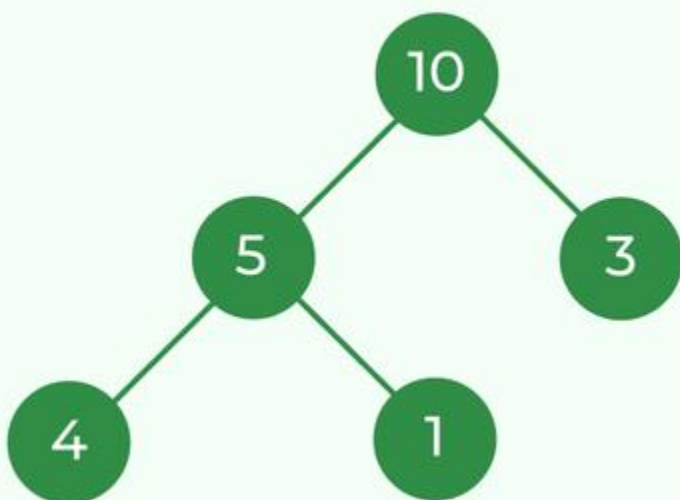


arr = {10, 4, 3, 5, 1}

- Now, as seen, 4 as a parent is smaller than the child 5, thus swap both of these again and the resulted heap and array should be like this:

Step 3

Make it a max heap (4 less than 5 & 5 is greater between the two children)

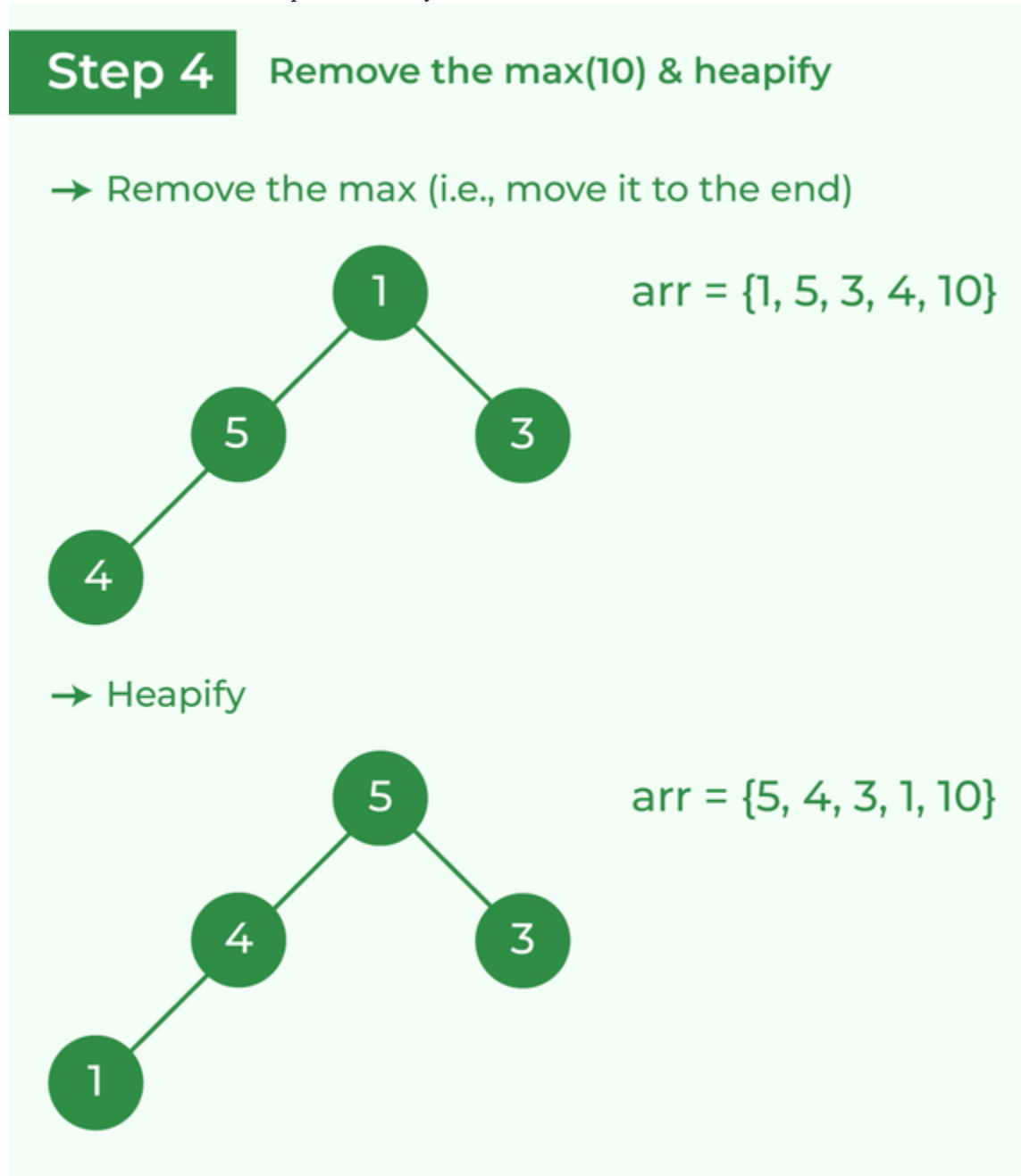


arr = {10, 5, 3, 4, 1}

Make the tree a max heap

Perform heap sort: Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.

- Delete the root element (10) from the max heap. In order to delete this node, try to swap it with the last node, i.e. (1). After removing the root element, again heapify it to convert it into max heap.
 - Resulted heap and array should look like this:



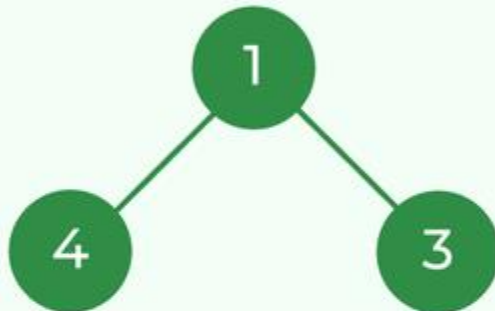
Remove 10 and perform heapify

- Repeat the above steps and it will look like the following:

Step 5

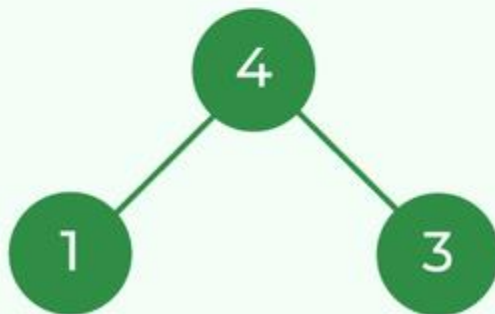
Remove the current max(5) & heapify

→ Remove the max (i.e., move it to the end)



arr = {1, 4, 3, 5, 10}

→ Heapify



arr = {4, 1, 3, 5, 10}

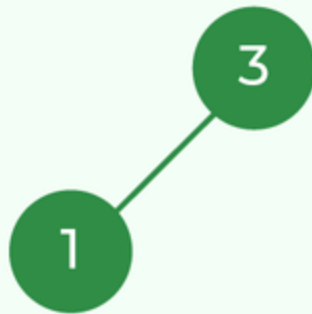
Remove 5 and perform heapify

- Now remove the root (i.e. 3) again and perform heapify.

Step 6

Remove the current max(4) & heapify

→ Remove the max (i.e., move it to the end)



$arr = \{3, 1, 4, 5, 10\}$

It is already in max heap form

Remove 4 and perform heapify

- Now when the root is removed once again it is sorted. and the sorted array will be like $arr[] = \{1, 3, 4, 5, 10\}$.

Step 7

Remove the max(3)

$arr = \{1, 3, 4, 5, 10\}$

The array is now sorted

The sorted array

What is Counting Sort?

Counting Sort is a **non-comparison-based** sorting algorithm that works well when there is limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind **Counting Sort** is to count the **frequency** of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

How does Counting Sort Algorithm work?

Step1 :

- Find out the maximum element from the given array.

Step 1:

	0	1	2	3	4	5	6	7	max
inputArray	2	5	3	0	2	3	0	3	5

Counting Sort



Step 2:

- Initialize a `countArray[]` of length `max+1` with all elements as 0. This array will be used for storing the occurrences of the elements of the input array.

Step 2:

	0	1	2	3	4	5
countArray	0	0	0	0	0	0

Counting Sort



Step 3:

- In the **countArray[]**, store the count of each unique element of the input array at their respective indices.
- **For Example:** The count of element **2** in the input array is **2**. So, store **2** at index **2** in the **countArray[]**. Similarly, the count of element **5** in the input array is **1**, hence store **1** at index **5** in the **countArray[]**.

Step 3:

	0	1	2	3	4	5
countArray	2	0	2	3	0	1

Counting Sort



Step 4:

- Store the **cumulative sum** or **prefix sum** of the elements of the **countArray[]** by doing **countArray[i] = countArray[i - 1] + countArray[i]**. This will help in placing the elements of the input array at the correct index in the output array.

Step 4 :

	0	1	2	3	4	5
countArray	2	2	4	7	7	8

Counting Sort



Step 5:

- Iterate from end of the input array and because traversing input array from end preserves the order of equal elements, which eventually makes this sorting algorithm **stable**.
- Update **outputArray[countArray[inputArray[i]] - 1] = inputArray[i]**.
- Also, update **countArray[inputArray[i]] = countArray[inputArray[i]] - 1**.

Step 5 :

inputArray	0	1	2	3	4	5	6	7
	2	5	3	0	2	3	0	3

countArray	0	1	2	3	4	5
	2	2	4	7	7	8

outputArray	0	1	2	3	4	5	6	7
							3	

Diagram illustrating the state of the arrays during Step 5. The inputArray has 8 elements. The countArray has 6 elements. The outputArray has 8 elements. Arrows indicate the mapping of elements from inputArray to outputArray. Specifically, the element 3 at index 7 of inputArray is being placed at index 6 of outputArray (7-1=6).

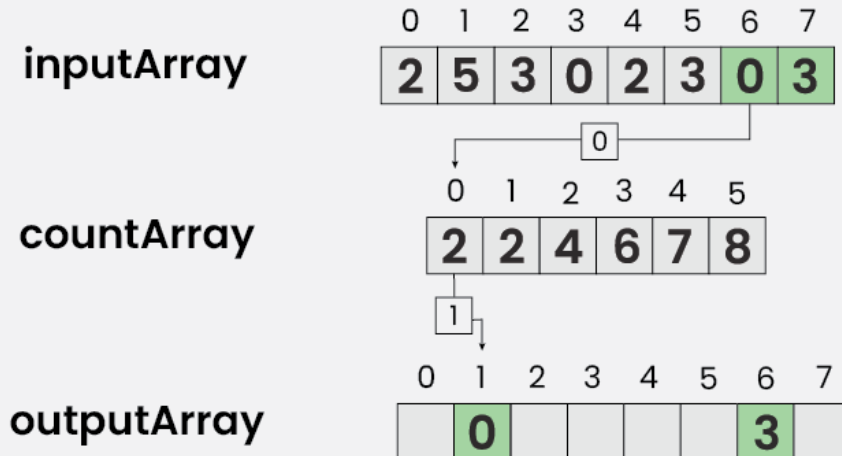
Counting Sort



Step 6: For $i = 6$,

Update $\text{outputArray}[\text{countArray}[\text{inputArray}[6]] - 1] = \text{inputArray}[6]$
 Also, update $\text{countArray}[\text{inputArray}[6]] = \text{countArray}[\text{inputArray}[6]] - 1$

Step 6 :

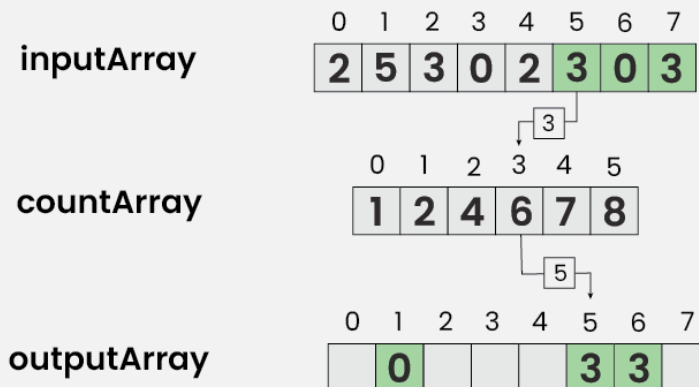


Counting Sort



Step 7: For $i = 5$,
 Update $\text{outputArray}[\text{countArray}[\text{inputArray}[5]] - 1] = \text{inputArray}[5]$
 Also, update $\text{countArray}[\text{inputArray}[5]] = \text{countArray}[\text{inputArray}[5]] - 1$

Step 7 :

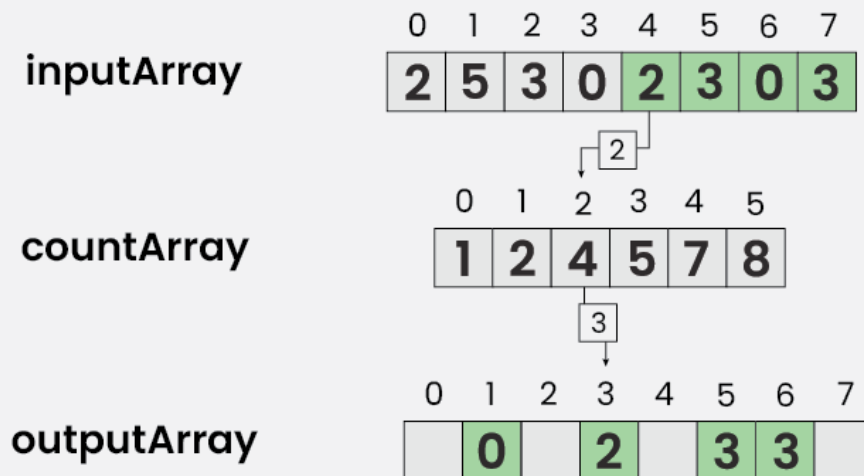


Counting Sort



Step 8: For $i = 4$,
 Update $\text{outputArray}[\text{countArray}[\text{inputArray}[4]] - 1] = \text{inputArray}[4]$
 Also, update $\text{countArray}[\text{inputArray}[4]] = \text{countArray}[\text{inputArray}[4]] - 1$

Step 8:



Counting Sort

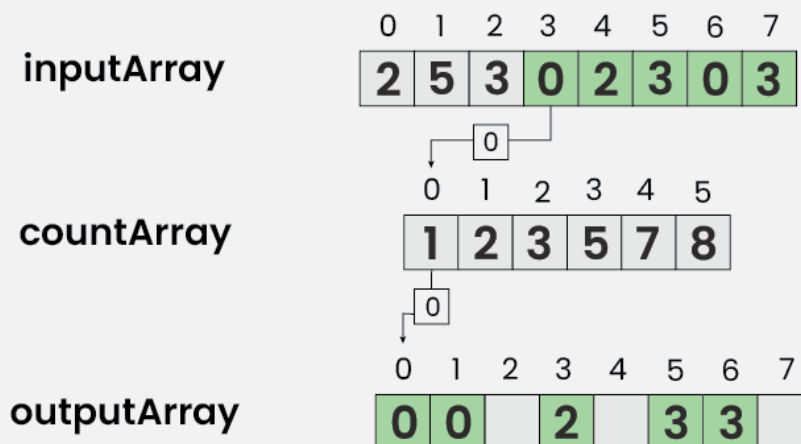


Step 9: For $i = 3$,

Update $\text{outputArray}[\text{countArray}[\text{inputArray}[3]] - 1] = \text{inputArray}[3]$

Also, update $\text{countArray}[\text{inputArray}[3]] = \text{countArray}[\text{inputArray}[3]] - 1$

Step 9:



Counting Sort

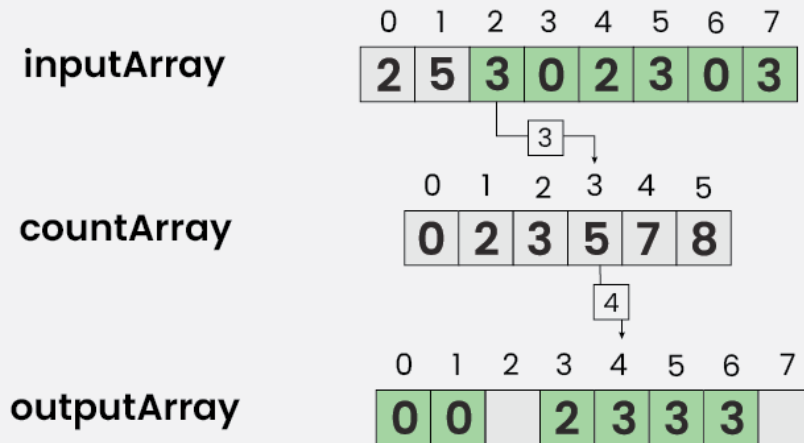


Step 10: For $i = 2$,

Update $\text{outputArray}[\text{countArray}[\text{inputArray}[2]] - 1] = \text{inputArray}[2]$

Also, update $\text{countArray}[\text{inputArray}[2]] = \text{countArray}[\text{inputArray}[2]] - 1$

Step 10 :



Counting Sort

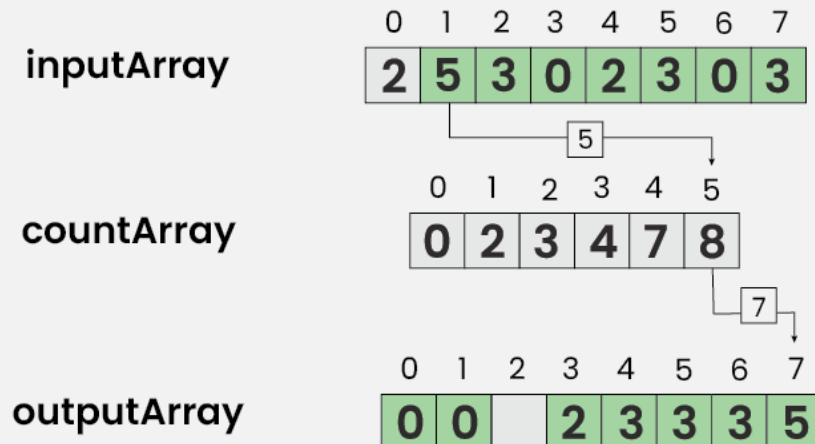


Step 11: For $i = 1$,

Update $outputArray[countArray[inputArray[1]] - 1] = inputArray[1]$

Also, update $countArray[inputArray[1]] = countArray[inputArray[1]] - 1$

Step 11 :



Counting Sort



Step 12: For $i = 0$,

Update $outputArray[countArray[inputArray[0]] - 1] = inputArray[0]$

Also, update $countArray[inputArray[0]] = countArray[inputArray[0]] - 1$

Step 12 :

inputArray

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

countArray

0	1	2	3	4	5
0	2	3	4	7	7

outputArray

0	1	2	3	4	5	6	7
0	0	2	2	3	3	3	5

Counting Sort



Radix Sort Algorithm

The key idea behind Radix Sort is to exploit the concept of place value. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list. Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

How does Radix Sort Algorithm work?


To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:

Consider this input

Array

170	45	75	90	802	24	2	66
-----	----	----	----	-----	----	---	----

Unsorted

Radix Sort 

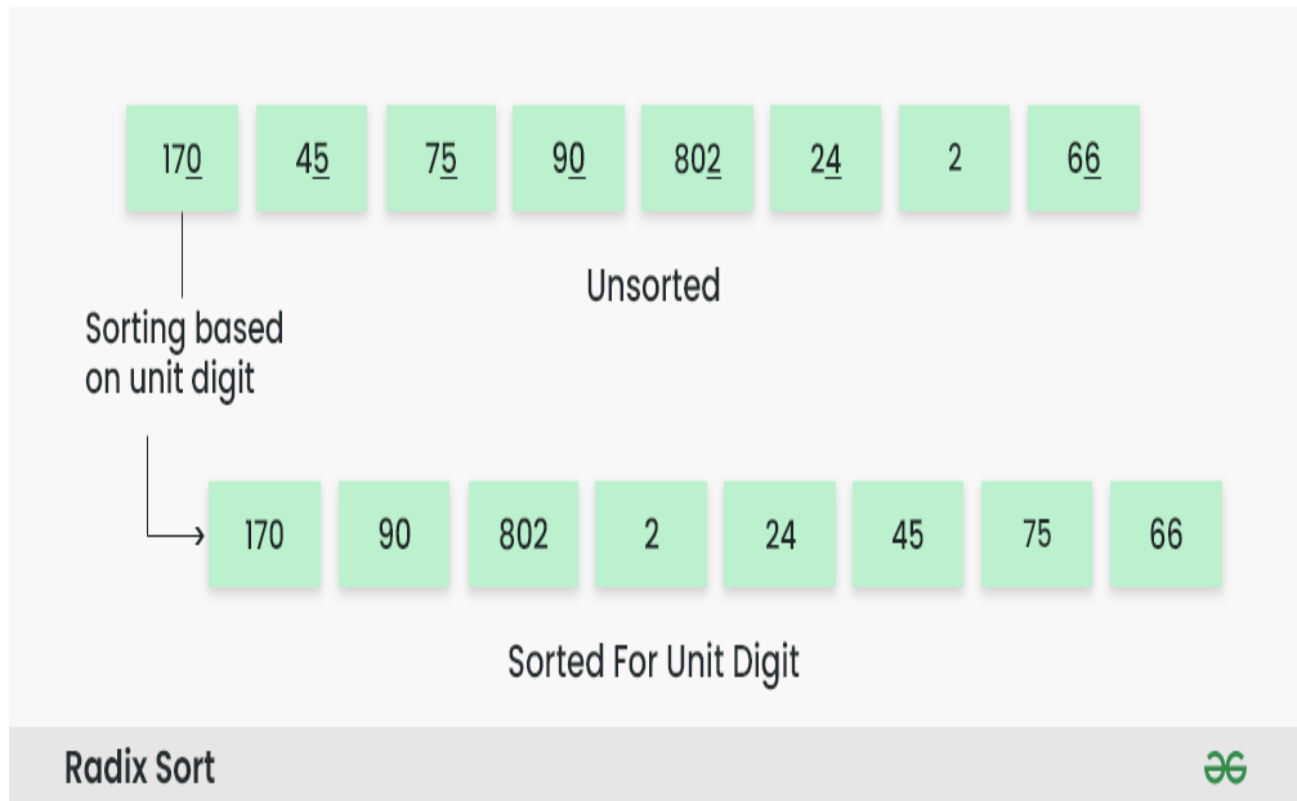
How does Radix Sort Algorithm work | Step 1

Step 1: Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.

Step 2: Sort the elements based on the unit place digits ($X=0$). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place.

Sorting based on the unit place:

- Perform counting sort on the array based on the unit place digits.
- The sorted array based on the unit place is [170, 90, 802, 2, 24, 45, 75, 66].

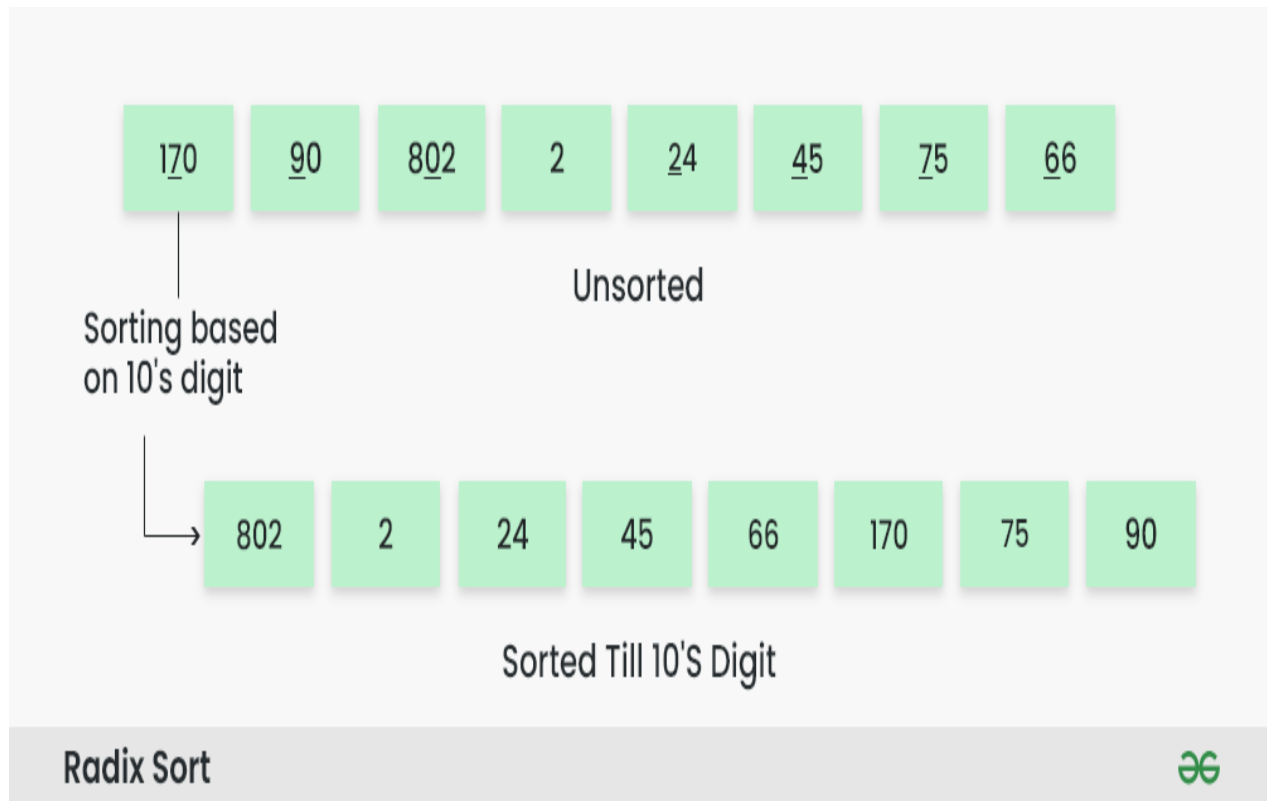


How does Radix Sort Algorithm work | Step 2

Step 3: Sort the elements based on the tens place digits.

Sorting based on the tens place:

- Perform counting sort on the array based on the tens place digits.
- The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90].

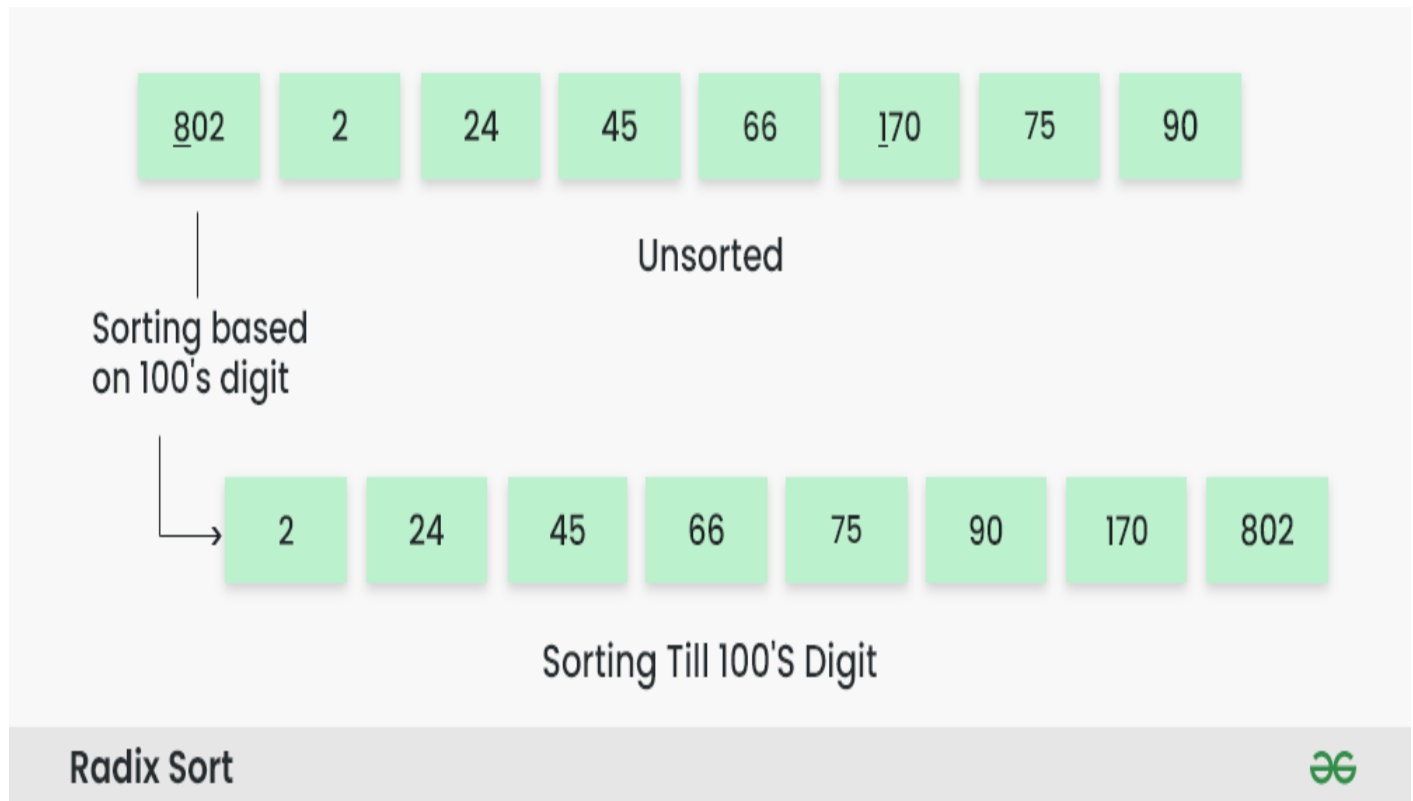


How does Radix Sort Algorithm work | Step 3

Step 4: Sort the elements based on the hundreds place digits.

Sorting based on the hundreds place:

- Perform counting sort on the array based on the hundreds place digits.
- The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].



How does Radix Sort Algorithm work | Step 4

Step 5: The array is now sorted in ascending order.
The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802].

Array after performing **Radix Sort** for all digits

2	24	45	66	75	90	170	802
---	----	----	----	----	----	-----	-----

Radix Sort



How does Radix Sort Algorithm work | Step 5

EXAMPLE 2

Now let's see the working of radix sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using radix sort. It will make the explanation clearer and easier.

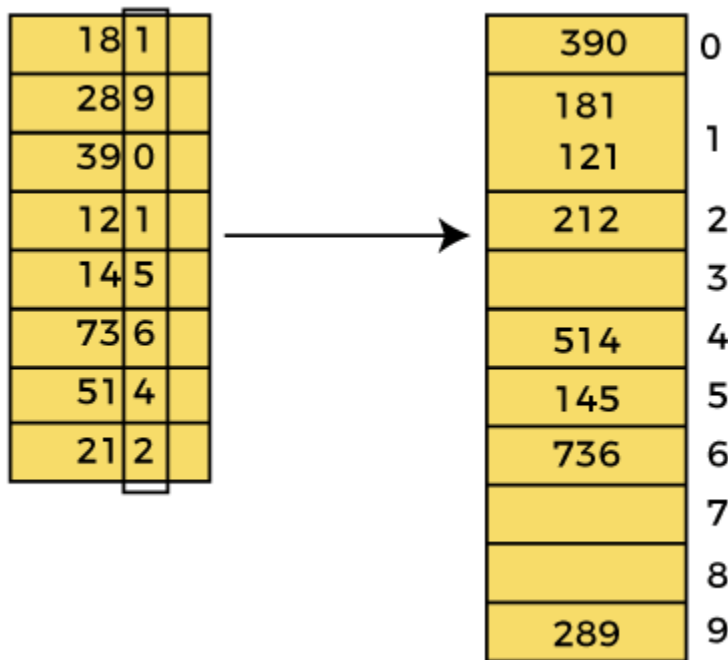
181	289	390	121	145	736	514	212
-----	-----	-----	-----	-----	-----	-----	-----

In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up to three times (i.e., to the **hundreds place**). That means three passes are required to sort the array.

Now, first sort the elements on the basis of unit place digits (i.e., **x = 0**). Here, we are using the counting sort algorithm to sort the elements.

Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.

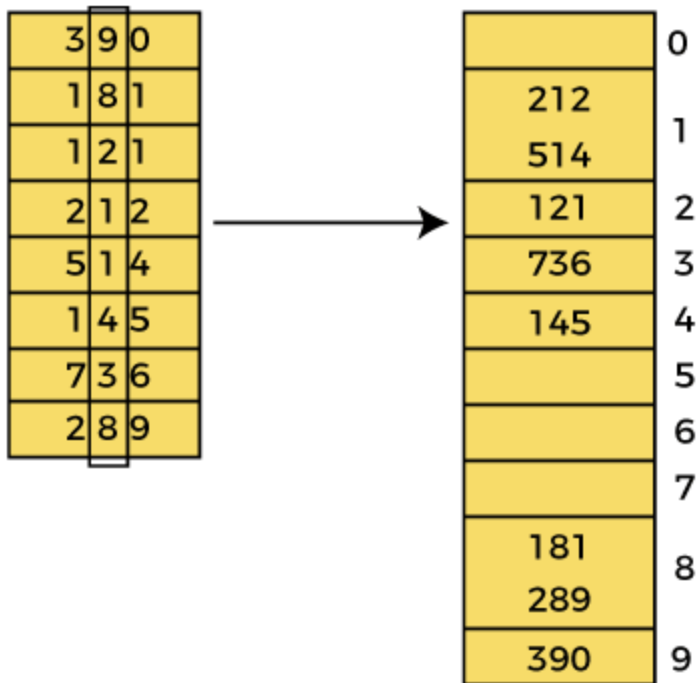


After the first pass, the array elements are -

390	181	121	212	514	145	736	289
-----	-----	-----	-----	-----	-----	-----	-----

Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).

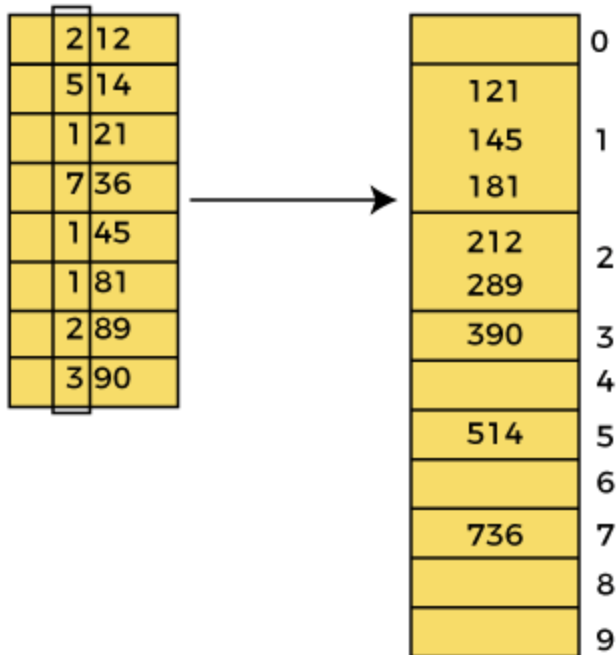


After the second pass, the array elements are -

212	514	121	736	145	181	289	390
-----	-----	-----	-----	-----	-----	-----	-----

Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).



After the third pass, the array elements are -

121	145	181	212	289	390	514	736
-----	-----	-----	-----	-----	-----	-----	-----