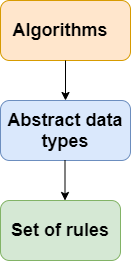
**DATA STRUCTURES**

**DS - TUTORIAL**

What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.



ADTs provide a way to abstract the complexity of data structures, allowing programmers to work with high-level specifications without worrying about the low-level details. The actual implementation of an ADT can vary, and different data structures can satisfy the requirements of the same ADT.

Top of Form

Types of Data Structures

There are two types of data structures:

* Primitive data structure
* Non-primitive data structure

**Primitive Data structure**

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

**Non-Primitive Data structure**

The non-primitive data structure is divided into two types:

* Linear data structure
* Non-linear data structure

**Compile Time:**

* **Definition:** Compile time, also known as compilation time, is the time during which a program's source code is translated into machine code or an intermediate code by a compiler.

**Run Time:**

* **Definition:** Run time, also known as execution time, is the time when a compiled program is actually running and performing its tasks.

**Linear Data Structure**

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

**When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.**

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures.

**Data structures can also be classified as:**

* **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
* **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

* **Searching:** We can search for any element in a data structure.
* **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
* **Insertion:** We can also insert the new element in a data structure.
* **Updation:** We can also update the element, i.e., we can replace the element with another element.
* **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Which Data Structure?

A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time. For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the implementation. Therefore, we conclude that we require some data structure to implement a particular ADT.

An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part. Now the question arises: how can one get to know which data structure to be used for a particular ADT?.

As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space. For example, the Stack ADT can be implemented by both Arrays and linked list. Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.

**--DS tutorial completed**

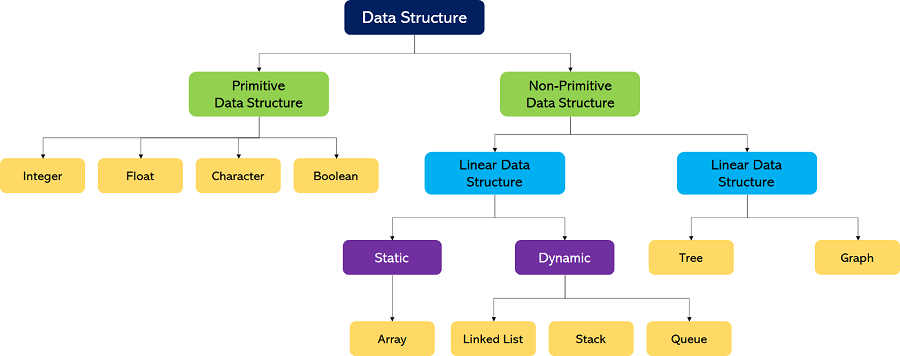
**DS – INTRODUCTION**

**Data** is a collection of facts and figures or a set of values or values of a specific format that refers to a single set of item values. The data items are then classified into sub-items, which is the group of items that are not known as the simple primary form of the item.

## **Classification of Data Structures**

1. Primitive Data Structure
2. Non-Primitive Data Structure

The following figure shows the different classifications of Data Structures.



### Primitive Data Structures

1. These data structures can be manipulated or operated directly by machine-level instructions.
2. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.

### Non-Primitive Data Structures

1. These data structures can't be manipulated or operated directly by machine-level instructions.
2. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
3. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
   1. Linear Data Structures
   2. Non-Linear Data Structures

### Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element

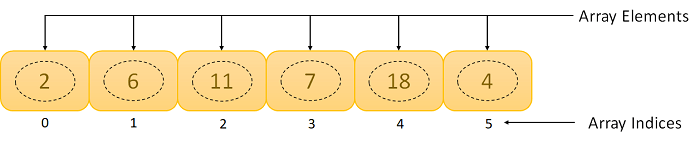
Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.  
   The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code.  
   **Linked Lists, Stacks**, and **Queues** are common examples of dynamic data structures

**1. Arrays**

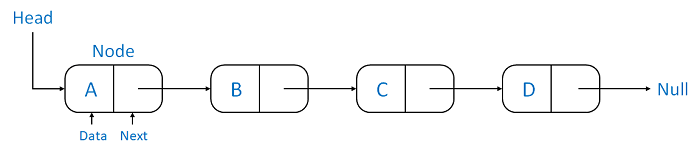
An **Array** is a data structure used to collect multiple data elements of the same data type into one variable. We can access any data element from the list with the help of its location in the list. Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.

|  |  |
| --- | --- |
| Contiguous memory allocation allocates consecutive blocks of memory to a file/process. | Non-Contiguous memory allocation allocates separate blocks of memory to a file/process. |



**2. Linked Lists**

A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

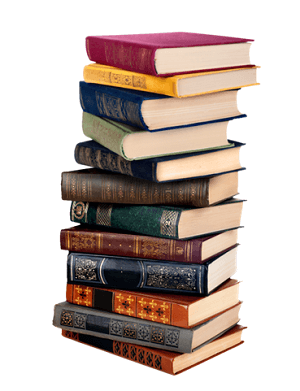


**Linked Lists can be classified into different types:**

1. **Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.
2. **Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).
3. **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

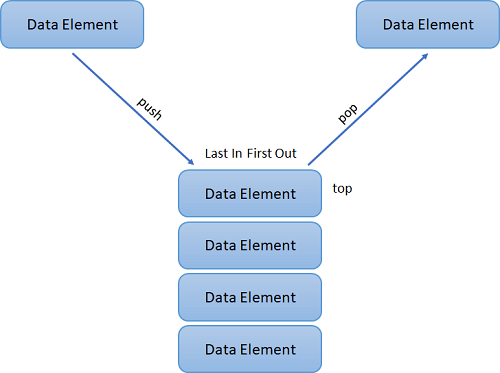
**3. Stacks**

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.



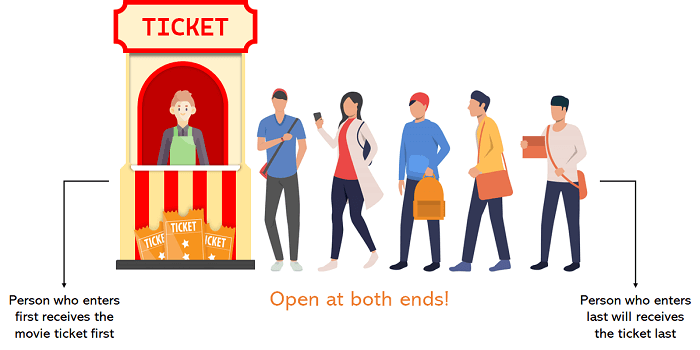
**The primary operations in the Stack are as follows:**

1. **Push:** Operation to insert a new element in the Stack is termed as Push Operation.
2. **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.



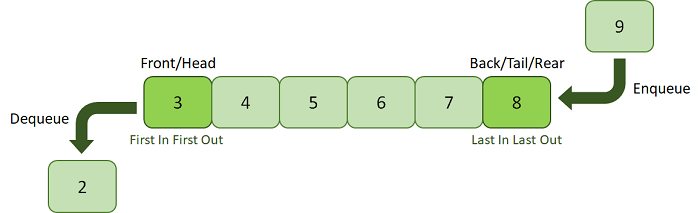
**4. Queues**

A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.



The following are the primary operations of the Queue:

1. **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.
2. **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.



Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

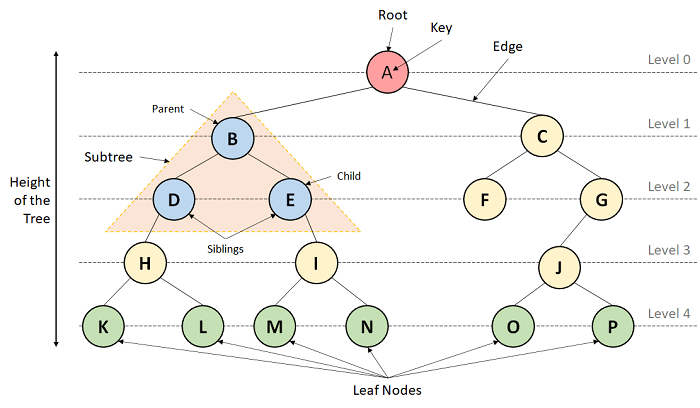
Types of Non-Linear Data Structures

The following is the list of Non-Linear Data Structures that we generally use:

**1. Trees**

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.



**Trees can be classified into different types:**

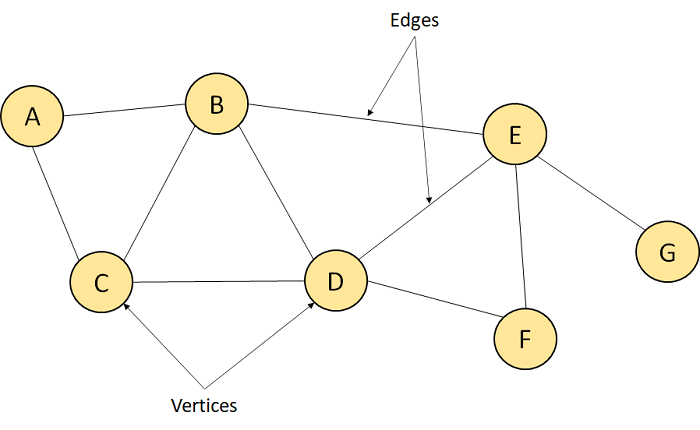
1. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.
2. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.

**2. Graphs**

A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below:

G = (V,E)



## **Basic Operations of Data Structures**

In the following section, we will discuss the different types of operations that we can perform to manipulate data in every data structure:

1. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.
2. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
3. **Insertion:** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.
4. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.
5. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.
6. **Merge:** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.
7. **Create:** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:
   1. Compile-time
   2. Run-time  
      For example, the **malloc()** function is used in C Language to create data structure.
8. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.
9. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.
10. **Splitting:** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

## **Understanding the Abstract Data Type**

we can conclude that the operations in data structure include:

1. A high level of abstractions like addition or deletion of an item from a list.
2. Searching and sorting an item in a list.
3. Accessing the highest priority item in a list.

Whenever the data structure does such operations, it is known as an **Abstract Data Type (ADT)**.

We can define it as a set of data elements along with the operations on the data. The term "abstract" refers to the fact that the data and the fundamental operations defined on it are being studied independently of their implementation. It includes what we can do with the data, not how we can do it.

An ADI implementation contains a storage structure in order to store the data elements and algorithms for fundamental operation. All the data structures, like an array, linked list, queue, stack, etc., are examples of ADT

**DS –- INTRODUCTION COMPLETED**

# **DS Algorithm**

## **What is an Algorithm?**

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Approaches of Algorithm

**The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:**

* **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:
  1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
  2. **Sacrificing:** As soon as the best solution is found, then it will stop.
* **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.
* **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But, there are very rare cases in which it provides the optimal solution.
* **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
  1. It breaks down the problem into a subproblem to find the optimal solution.
  2. After breaking down the problem, it finds the optimal solution out of these subproblems.
  3. Stores the result of the subproblems is known as memorization.
  4. Reuse the result so that it cannot be recomputed for the same subproblems.
  5. Finally, it computes the result of the complex program.
* **Branch and Bound Algorithm:** The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.
* **Randomized Algorithm:** As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm?. In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.
* **Backtracking:** Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

The major categories of algorithms are given below:

* **Sort:** Algorithm developed for sorting the items in a certain order.
* **Search:** Algorithm developed for searching the items inside a data structure.
* **Delete:** Algorithm developed for deleting the existing element from the data structure.
* **Insert:** Algorithm developed for inserting an item inside a data structure.
* **Update:** Algorithm developed for updating the existing element inside a data structure.

### Algorithm Analysis

* **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution
* **Space complexity:** An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.
5. Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

**Space complexity = Auxiliary space + Input size.**

Types of Algorithms

**The following are the types of algorithm:**

* **Search Algorithm**
* **Sort Algorithm**

**Search Algorithm**

* **Linear search**
* **Binary search**

**Linear Search**

Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found. It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1. This algorithm can be implemented on the unsorted list.

**Binary Search**

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

### Sorting Algorithms

Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.

**DS ALGORITHM – completed**

### Asymptotic AnalysisAsymptotic Notations

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

* Big oh Notation (?)
* Omega Notation (Ω)
* Theta Notation (θ)

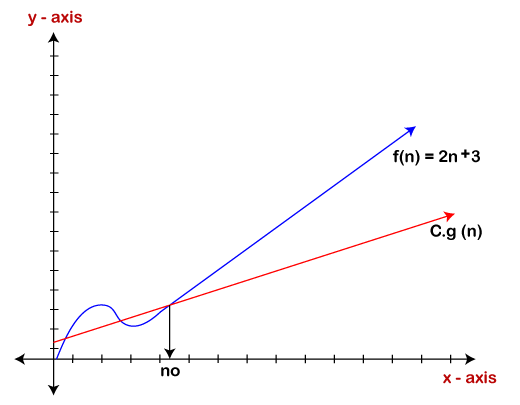
Big oh Notation (O)

* Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
* This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

# Asymptotic Analysis

Omega Notation (Ω)

* It basically describes the best-case scenario which is opposite to the big o notation.
* It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
* It determines what is the fastest time that an algorithm can run.



Theta Notation (θ)

* The theta notation mainly describes the average case scenarios.
* It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
* Big theta is mainly used when the value of worst-case and the best-case is same.
* It is the formal way to express both the upper bound and lower bound of an algorithm running time.



### Why we have three different asymptotic analysis?

As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm.

Suppose we have an array of n numbers, and we want to find the particular element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose, if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst case would be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.

Common Asymptotic Notations

|  |  |  |
| --- | --- | --- |
| constant | - | ?(1) |
| linear | - | ?(n) |
| logarithmic | - | ?(log n) |
| n log n | - | ?(n log n) |
| exponential | - | 2?(n) |
| cubic | - | ?(n3) |
| polynomial | - | n?(1) |
| quadratic | - | ?(n2) |

Asymptotic Notations – completed

# **Pointer**

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

* Traversing String
* Lookup Tables
* Control Tables
* Tree Structures

## **Pointer Details**

* **Pointer arithmetic:** There are four arithmetic operators that can be used in pointers: ++, --, +, -
* **Array of pointers:** You can define arrays to hold a number of pointers.
* **Pointer to pointer:** C allows you to have pointer on a pointer and so on.
* **Passing pointers to functions in C:** Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
* **Return pointer from functions in C:** C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.



**POINTER – COMPLETED**

# **Array in Data Structure**

Why are arrays required?

Arrays are useful because -

* Sorting and searching a value in an array is easier.
* Arrays are best to process multiple values quickly and easily.
* **Arrays are good for storing multiple values in a single variable -** In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

## **Basic operations**

Now, let's discuss the basic operations supported in the array -

* Traversal - This operation is used to print the elements of the array.
* Insertion - It is used to add an element at a particular index.
* Deletion - It is used to delete an element from a particular index.
* Search - It is used to search an element using the given index or by the value.
* Update - It updates an element at a particular index.
* **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Average Case** | **Worst Case** |
| Access | O(1) | O(1) |
| Search | O(n) | O(n) |
| Insertion | O(n) | O(n) |
| Deletion | O(n) | O(n) |

**DS 1D ARRAY – COMPLETED**

**Linked list in Python**

# Python Linked List

In this article, we will learn about the implementation of a linked list in [Python](https://www.geeksforgeeks.org/python-programming-language/). To implement the linked list in Python, we will use [classes in Python](https://www.geeksforgeeks.org/python-classes-and-objects/). Now, we know that a linked list consists of nodes and nodes have two elements i.e. data and a reference to another node. Let’s implement the node first.

## What is Linked List in Python

[A linked list](https://www.geeksforgeeks.org/what-is-linked-list/) is a type of linear data structure similar to arrays. It is a collection of nodes that are linked with each other. A node contains two things first is data and second is a link that connects it with another node. Below is an example of a linked list with four nodes and each node contains character data and a link to another node. Our first node is where **head** points and we can access all the elements of the linked list using the **head.**



*Linked List*

## Creating a linked list in Python

In this LinkedList class, we will use the Node class to create a linked list. In this class, we have an **\_\_init\_\_**method that initializes the linked list with an empty head. Next, we have created an **insertAtBegin()** method to insert a node at the beginning of the linked list, an **insertAtIndex()** method to insert a node at the given index of the linked list, and **insertAtEnd()** method inserts a node at the end of the linked list. After that, we have the **remove\_node()**method which takes the data as an argument to delete that node. In the **remove\_node()** method we traverse the linked list if a node is present equal to data then we delete that node from the linked list. Then we have the **sizeOfLL()**method to get the current size of the linked list and the last method of the LinkedList class is **printLL()** which traverses the linked list and prints the data of each node.

### Creating a Node Class

We have created a Node class in which we have defined a **\_\_init\_\_** function to initialize the node with the data passed as an argument and a reference with None because if we have only one node then there is nothing in its reference.

## Insertion in Linked List

### Insertion at Beginning in Linked List

This method inserts the node at the beginning of the linked list. In this method, we create a **new\_node** with the given **data**and check if the head is an empty node or not if the head is empty then we make the **new\_node** as **head** and **return**else we insert the head at the next **new\_node** and make the **head** equal to **new\_node.**

### Insert a Node at a Specific Position in a Linked List

This method inserts the node at the given index in the linked list. In this method, we create a **new\_node**with given data , a current\_node that equals to the head, and a counter **‘position’**initializes with **0.**Now, if the index is equal to zero it means the node is to be inserted at begin so we called **insertAtBegin() method**else we run a while loop until the **current\_node** is not equal to **None**or**(position+1)** is not equal to the index we have to at the one position back to insert at a given position to make the linking of nodes and in each iteration, we increment the position by 1 and make the **current\_node** next of it. When the loop breaks and if **current\_node** is not equal to **None**we insert new\_node at after to the **current\_node.**If **current\_node** is equal to **None** it means that the index is not present in the list and we print **“Index not present”.**

### Insertion in Linked List at End

This method inserts the node at the end of the linked list. In this method, we create a **new\_node**with the given data and check if the**head** is an empty node or not if the**head** is empty then we make the **new\_node** as **head** andreturn **else** we make a**current\_node equal**to**the head** traverse to the last **node** of the linked list and when we get **None**after the current\_node the while loop breaks and insert the **new\_node**in the next of **current\_node**which is the last node of linked list.

# **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).**

Applications of Linked list

The applications of the Linked list are given as follows -

* With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
* A linked list can be used to represent the sparse matrix.
* The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
* Using linked list, we can implement stack, queue, tree, and other various data structures.
* The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
* A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list

The basic operations that are supported by a list are mentioned as follows -

* **Insertion -** This operation is performed to add an element into the list.
* **Deletion -** It is performed to delete an operation from the list.
* **Display -** It is performed to display the elements of the list.
* **Search -** It is performed to search an element from the list using the given key.

Complexity of Linked list

Now, let's see the time and space complexity of the linked list for the operations search, insert, and delete.

1. Time Complexity

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average case time complexity** | **Worst-case time complexity** |
| **Insertion** | O(1) | O(1) |
| **Deletion** | O(1) | O(1) |
| **Search** | O(n) | O(n) |

## **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).**

# **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.

. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(1) |
| **Average Case** | O(logn) |
| **Worst Case** | O(logn) |

* **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(logn).**
* **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(logn).**

**Searching algorithms -- completed**

# **Bubble sort Algorithm**

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is **O(n2),** where **n** is a number of items.

## **Algorithm**

In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

1. begin BubbleSort(arr)
2. **for** all array elements
3. **if** arr[i] > arr[i+1]
4. swap(arr[i], arr[i+1])
5. end **if**
6. end **for**
7. **return** arr
8. end BubbleSort

1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **O(n2).**
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **O(n2).**

## **Optimized Bubble sort Algorithm**

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable ***swapped.*** It is set to **true** if swapping requires; otherwise, it is set to **false.**

It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable **swapped** will be **false.** It means that the elements are already sorted, and no further iterations are required.

This method will reduce the execution time and also optimizes the bubble sort.

Algorithm for optimized bubble sort

1. bubbleSort(array)
2. n = length(array)
3. repeat
4. swapped = **false**
5. **for** i = 1 to n - 1
6. **if** array[i - 1] > array[i], then
7. swap(array[i - 1], array[i])
8. swapped = **true**
9. end **if**
10. end **for**
11. n = n - 1
12. until not swapped
13. end bubbleSort

# **Heap Sort Algorithm**

Heap sort basically recursively performs two main operations -

* Build a heap H, using the elements of array.
* Repeatedly delete the root element of the heap formed in 1st phase.

Before knowing more about the heap sort, let's first see a brief description of **Heap.**

What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Complete binary tree – all levels should have 2 nodes except the last level.

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.

Max heap – max element will always be at the root. (will be sorted in ascending order)

Min heap – will be sorted in descending order

## **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.

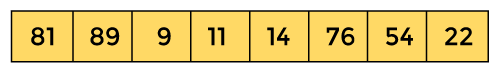
. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n logn) |
| **Average Case** | O(n log n) |
| **Worst Case** | O(n log n) |

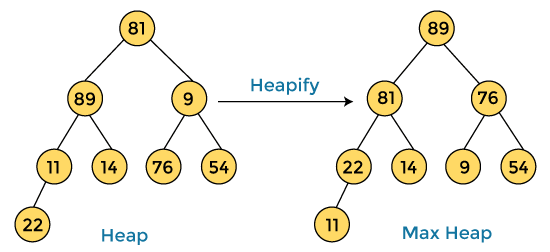
* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

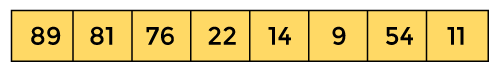
Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.



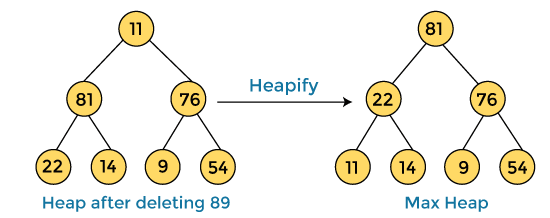
First, we have to construct a heap from the given array and convert it into max heap.



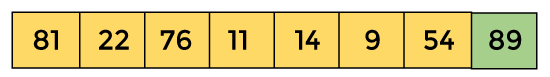
After converting the given heap into max heap, the array elements are -



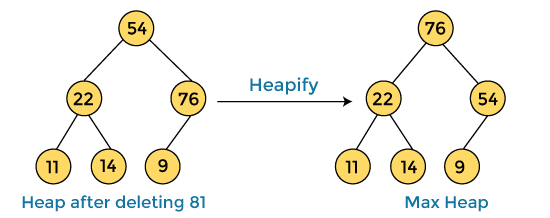
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



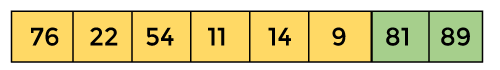
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -



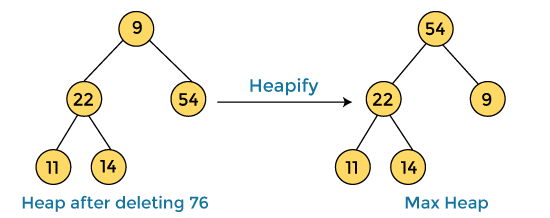
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



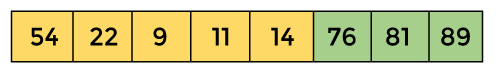
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -



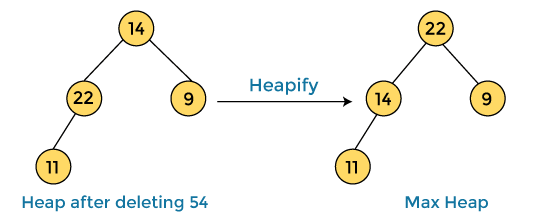
In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



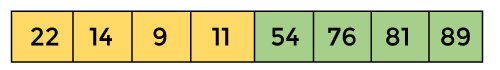
After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -



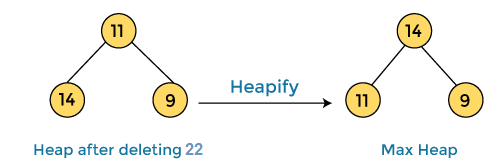
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



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



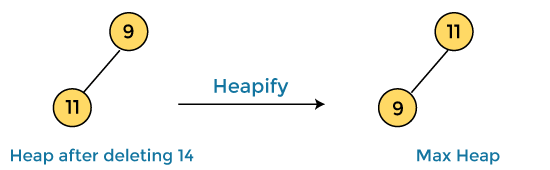
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



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



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.



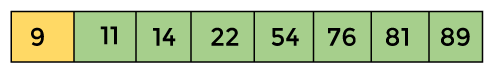
After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



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.



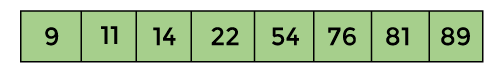
After swapping the array element **11** with **9,** the elements of array are -



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



After completion of sorting, the array elements are -



# **Insertion Sort Algorithm**

The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **O(n2)**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

* Simple implementation
* Efficient for small data sets
* Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Let the elements of array are -

Insertion Sort Algorithm

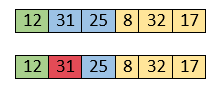
Initially, the first two elements are compared in insertion sort.

Insertion Sort Algorithm

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

Insertion Sort Algorithm

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

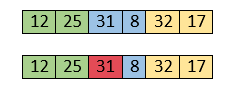


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

Insertion Sort Algorithm

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



Both 31 and 8 are not sorted. So, swap them.

Insertion Sort Algorithm

After swapping, elements 25 and 8 are unsorted.

Insertion Sort Algorithm

So, swap them.

Insertion Sort Algorithm

Now, elements 12 and 8 are unsorted.

Insertion Sort Algorithm

So, swap them too.

Insertion Sort Algorithm

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

Insertion Sort Algorithm

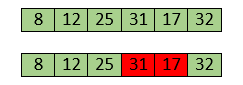
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Insertion Sort Algorithm

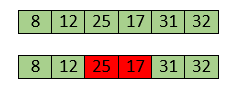
Move to the next elements that are 32 and 17.

Insertion Sort Algorithm

17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

Insertion Sort Algorithm

Now, the array is completely sorted.

## **Insertion sort complexity**

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

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **O(n2)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **O(n2)**.

# **Merge Sort Algorithm**

it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

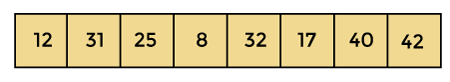
1. MERGE\_SORT(arr, beg, end)
3. **if** beg < end
4. set mid = (beg + end)/2
5. MERGE\_SORT(arr, beg, mid)
6. MERGE\_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of **if**
9. END MERGE\_SORT

## **Working of Merge sort Algorithm**

Now, let's see the working of merge sort Algorithm.

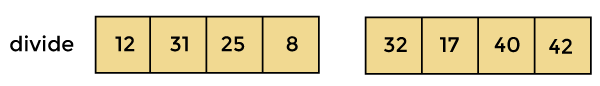
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

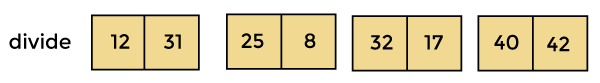


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

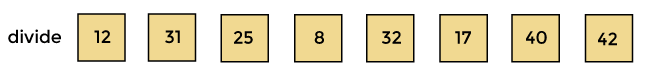
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



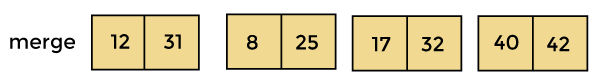
Now, again divide these arrays to get the atomic value that cannot be further divided.



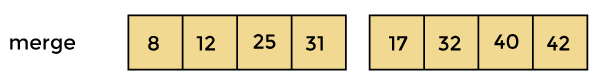
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

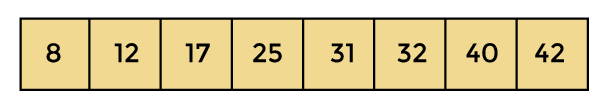
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

## **Merge sort complexity**

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

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n\*logn) |
| **Average Case** | O(n\*logn) |
| **Worst Case** | O(n\*logn) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n\*logn)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n\*logn)**.

# **Selection Sort Algorithm**

## **Working of Selection sort Algorithm**

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

selection Sort Algorithm

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

selection Sort Algorithm

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

selection Sort Algorithm

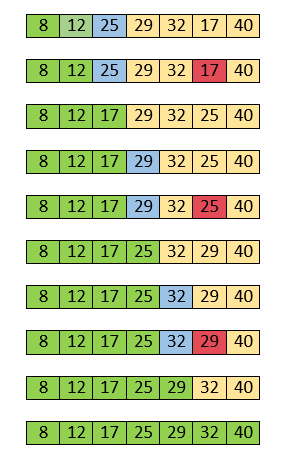
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

selection Sort Algorithm

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

selection Sort Algorithm

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

## **Selection sort complexity**

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

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n2) |
| **Average Case** | O(n2) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **O(n2)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **O(n2)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **O(n2)**.

# **Quick Sort Algorithm**

Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

**Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

**Conquer:** Recursively, sort two subarrays with Quicksort.

**Combine:** Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

**Choosing the pivot**

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

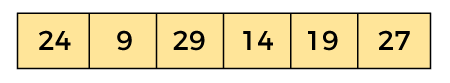
* Pivot can be random, i.e. select the random pivot from the given array.
* Pivot can either be the rightmost element of the leftmost element of the given array.
* Select median as the pivot element.

## **Working of Quick Sort Algorithm**

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

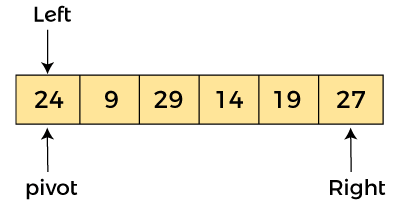
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

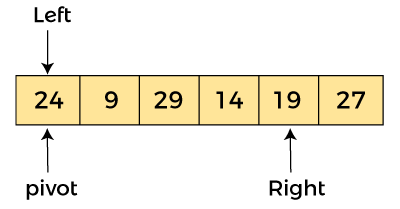


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

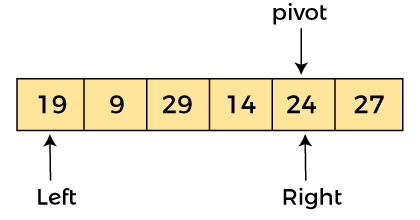


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



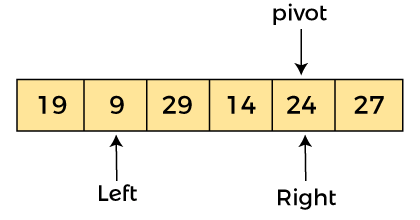
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

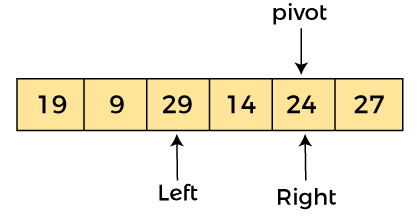


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

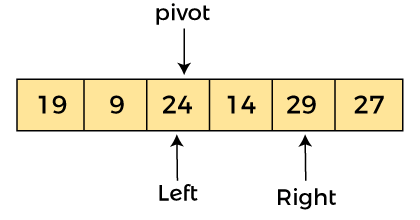
As a[pivot] > a[left], so algorithm moves one position to right as -



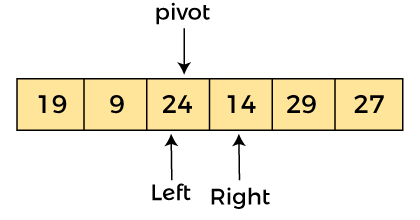
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



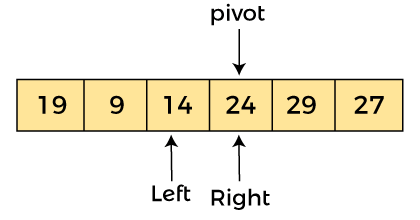
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



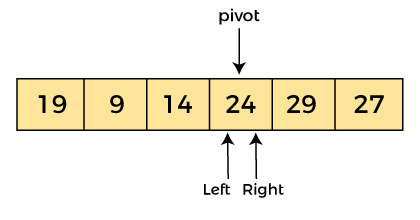
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



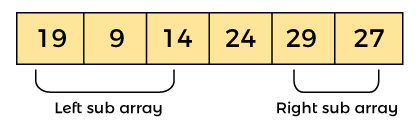
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



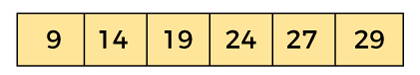
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now,in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



## **Quicksort complexity**

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

### 1. Time Complexity

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| **Best Case** | O(n\*logn) |
| **Average Case** | O(n\*logn) |
| **Worst Case** | O(n2) |

* **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n\*logn)**.
* **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n2)**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element. **– Sorting algorithms completed**