Linear vs Non-Linear data structure

What is Data structure?

A data structure is a technique of storing and organizing the data in such a way that the data can be utilized in an efficient manner. In computer science, a data structure is designed in such a way that it can work with various algorithms. A data structure is classified into two categories:

- Linear data structure
- Non-linear data structure

Now let's have a brief look at both these data structures.

What is the Linear data structure?

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.

The types of linear data structures are Array, Queue, Stack, Linked List.

Let's discuss each linear data structure in detail.

- **Array:** An array consists of data elements of a same data type. For example, if we want to store the roll numbers of 10 students, so instead of creating 10 integer type variables, we will create an array having size 10. Therefore, we can say that an array saves a lot of memory and reduces the length of the code.
- **Stack:** It is linear data structure that uses the LIFO (Last In-First Out) rule in which the data added last will be removed first. The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.
- **Queue:** It is a data structure that uses the FIFO rule (First In-First Out). In this rule, the element which is added first will be removed first. There are two terms used in the queue **front** end and **rear** The insertion operation performed at the back end is known ad enqueue, and the deletion operation performed at the front end is known as dequeue.
- **Linked list:** It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.

What is a Non-linear data structure?

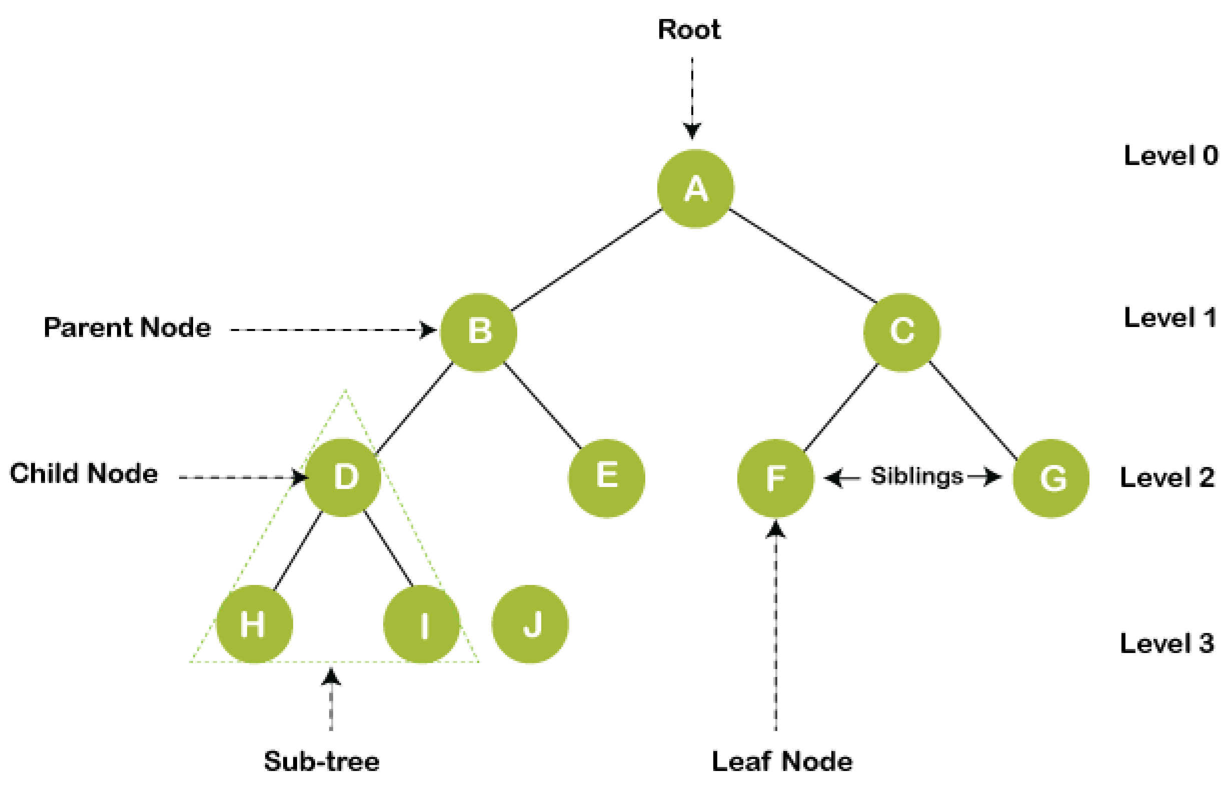
A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. As the arrangement is nonsequential, so the data elements cannot be traversed or accessed in a single run. In the case of linear data structure, element is connected to two elements (previous and the next element), whereas, in the non-linear data structure, an element can be connected to more than two elements.

Trees and **Graphs** are the types of non-linear data structure.

Let's discuss both the data structures in detail.

- **Tree**

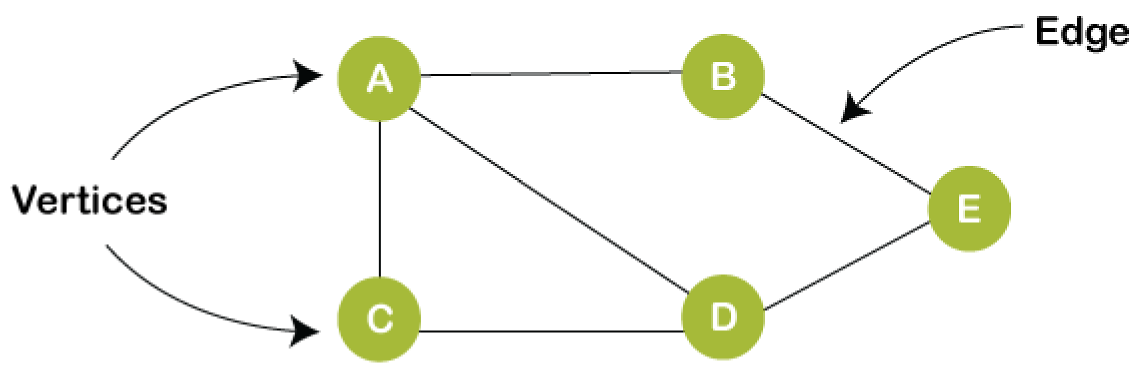
It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship. The diagrammatic representation of a **tree** data structure is shown below:



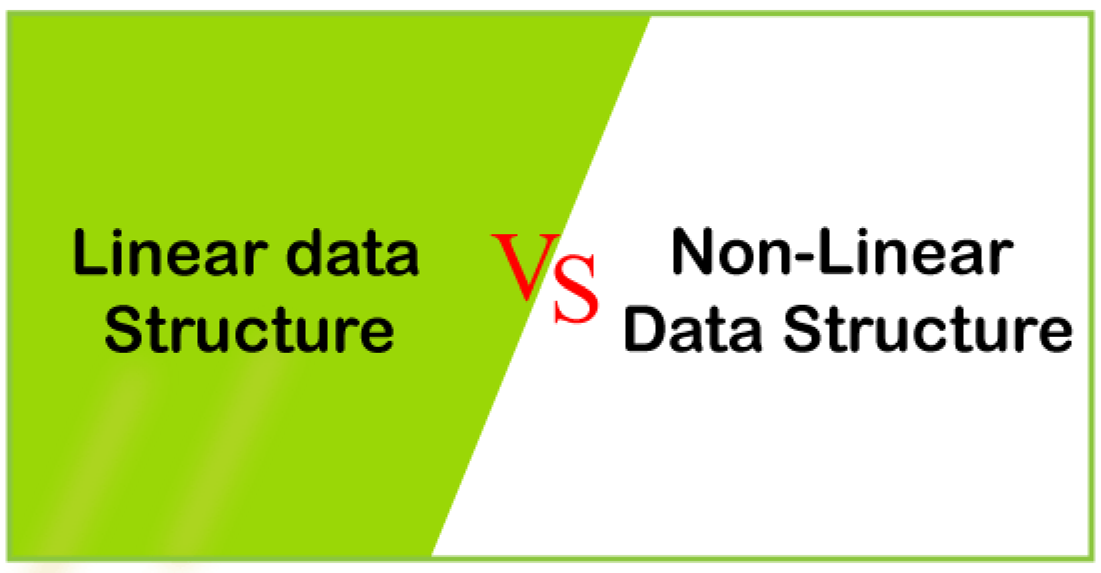
For example, the posts of employees are arranged in a tree data structure like managers, officers, clerk. In the above figure, **A** represents a manager, **B** and **C** represent the officers, and other nodes represent the clerks.

- **Graph**

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



Differences between the Linear data structure and non-linear data structure.



	Linear Data structure	Non-Linear Data structure
Basic	In this structure, the elements are arranged sequentially or linearly and attached to one another.	In this structure, the elements are arranged hierarchically or non-linear manner.

Types	Arrays, linked list, stack, queue are the types of a linear data structure.	Trees and graphs are the types of a non-linear data structure.
implementation	Due to the linear organization, they are easy to implement.	Due to the non-linear organization, they are difficult to implement.
Traversal	As linear data structure is a single level, so it requires a single run to traverse each data item.	The data items in a non-linear data structure cannot be accessed in a single run. It requires multiple runs to be traversed.
Arrangement	Each data item is attached to the previous and next items.	Each item is attached to many other items.
Levels	This data structure does not contain any hierarchy, and all the data elements are organized in a single level.	In this, the data elements are arranged in multiple levels.
Memory utilization	In this, the memory utilization is not efficient.	In this, memory is utilized in a very efficient manner.
Time complexity	The time complexity of linear data structure increases with the increase in the input size.	The time complexity of non-linear data structure often remains same with the increase in the input size.
Applications	Linear data structures are mainly used for developing the software.	Non-linear data structures are used in **image processing** and **Artificial Intelligence**.

Array vs Linked List

Array and **Linked list** are the two ways of organizing the data in the memory. Before understanding the differences between the **Array** and the **Linked List**, we first look **at an array** and **a linked list**.

What is an array?

An array is a data structure that contains the elements of the same type. A data structure is a way of organizing the data; an array is a data structure because it sequentially organizes the data. An array is a big chunk of memory in which memory is divided into small-small blocks, and each block is capable of storing some value.

Suppose we have created an array that consists of 10 values, then each block will store the value of an integer type. If we try to store the value in an array of different types, then it is not a correct array and will throw a compile-time error.

Declaration of array

An array can be declared as:

```
data_type name of the array[no of elements]
```

To declare an array, we first need to specify the type of the array and then the array's name. Inside the square brackets, we need to specify the number of elements that our array should contain.

Let's understand through an example.

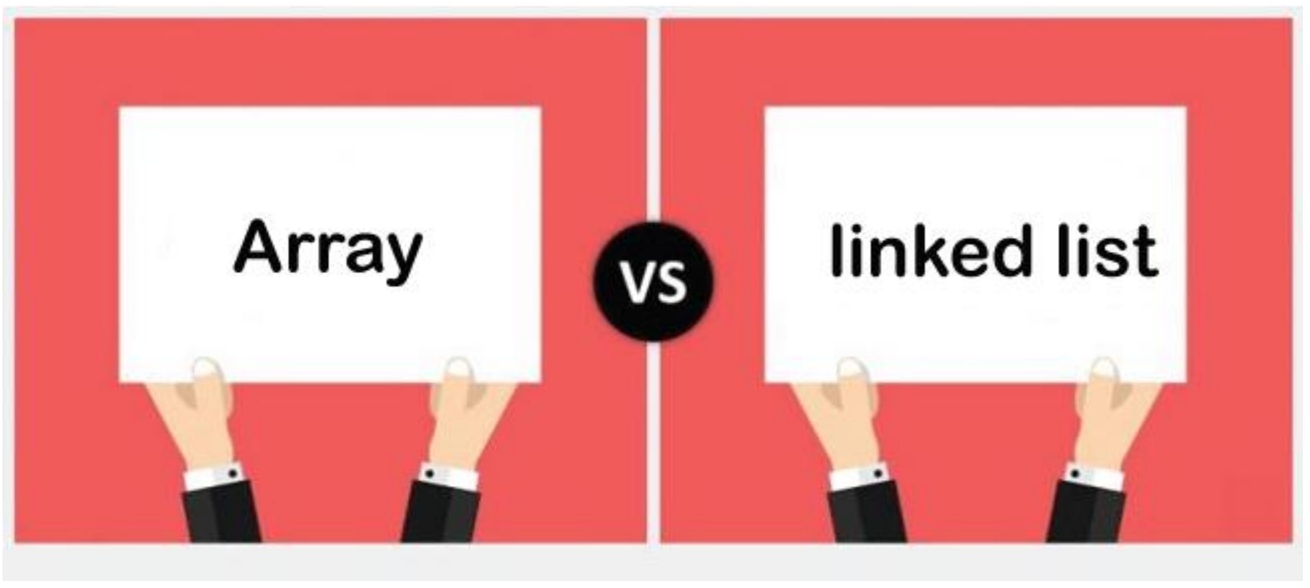
```
int a[5];
```

In the above case, we have declared an array of 5 elements with '**a**' name of an **integer** data type.

What is Linked list?

A linked list is the collection of nodes that are randomly stored. Each node consists of two fields, i.e., **data** and **link**. Here, data is the value stored at that particular node, and the link is the pointer that holds the address of the next node.

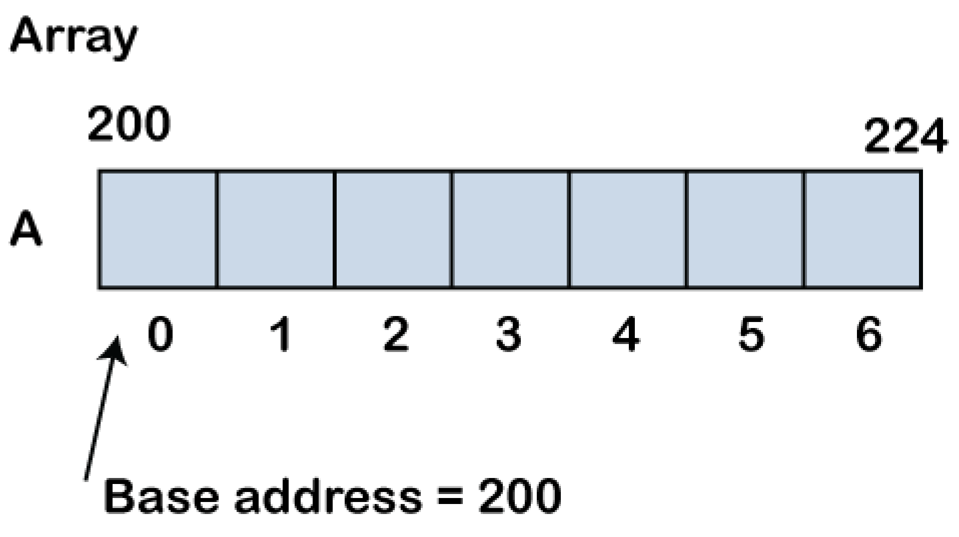
Differences between Array and Linked list



We cannot say which data structure is better, i.e., array or linked list. There can be a possibility that one data structure is better for one kind of requirement, while the other data structure is better for another kind of requirement. There are various factors like what are the frequent operations performed on the data structure or the size of the data, and other factors also on which basis the data structure is selected. Now we will see some differences between the array and the linked list based on some parameters.

1. Cost of accessing an element

In case of an array, irrespective of the size of an array, an array takes a constant time for accessing an element. In an array, the elements are stored in a contiguous manner, so if we know the base address of the element, then we can easily get the address of any element in an array. We need to perform a simple calculation to obtain the address of any element in an array. So, accessing the element in an array is **O(1)** in terms of time complexity.



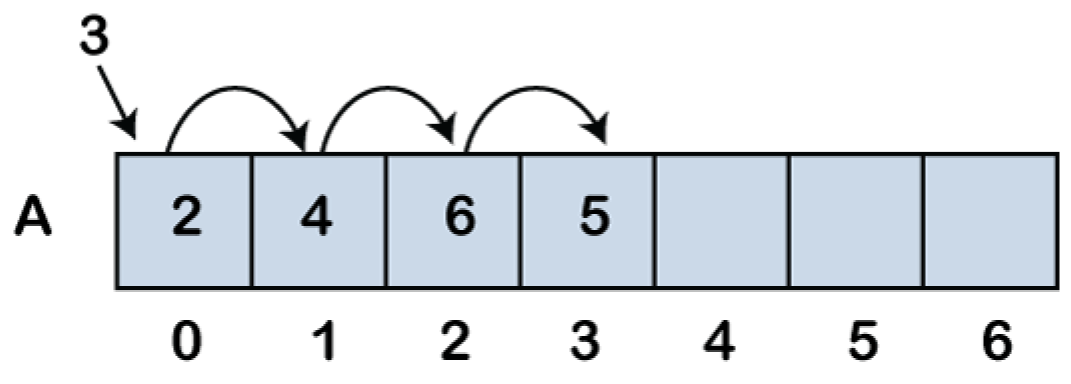
In the linked list, the elements are not stored in a contiguous manner. It consists of multiple blocks, and each block is represented as a node. Each node has two fields, i.e., one is for the data field, and another one stores the address of the next node. To find any node in the linked list, we first need to determine the first node known as the head node. If we have to find the second node in the list, then we need to traverse from the first node, and in the worst case, to find the last node, we will be traversing all the nodes. The average case for accessing the element is O(n).

We conclude that the cost of accessing an element in array is less than the linked list. Therefore, if we have any requirement for accessing the elements, then array is a better choice.

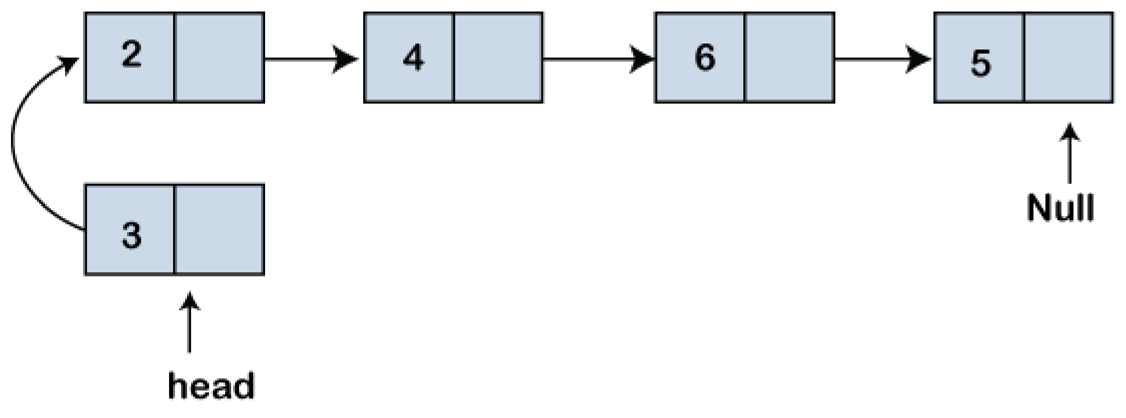
2. Cost of inserting an element

There can be three scenarios in the insertion:

- **Inserting the element at the beginning:** To insert the new element at the beginning, we first need to shift the element towards the right to create a space in the first position. So, the time complexity will be proportional to the size of the list. If n is the size of the array, the time complexity would be O(n).



In the case of a linked list, to insert an element at the starting of the linked list, we will create a new node, and the address of the first node is added to the new node. In this way, the new node becomes the first node. So, the time complexity is not proportional to the size of the list. The time complexity would be constant, i.e., O(1).



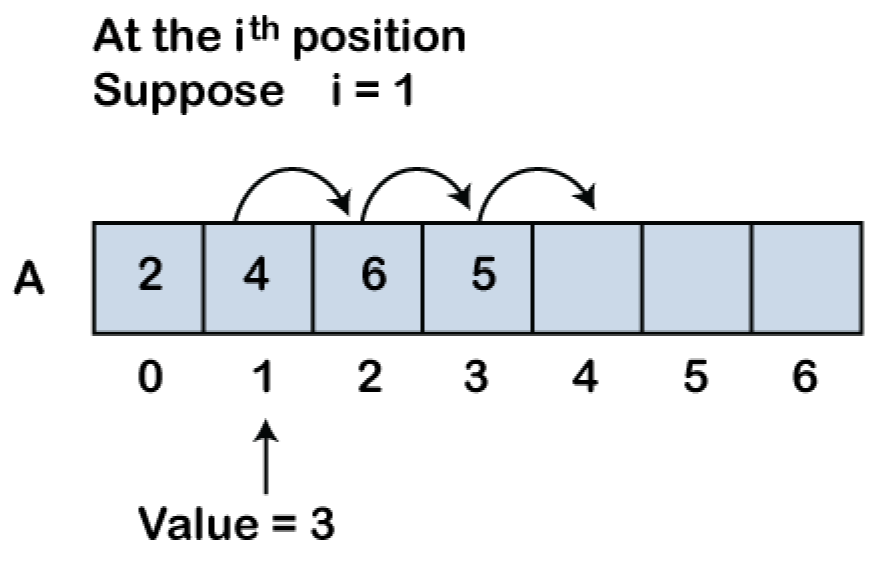
- **Inserting an element at the end**

If the array is not full, then we can directly add the new element through the index. In this case, the time complexity would be constant, i.e., O(1). If the array is full, we first need to copy the array into another array and add a new element. In this case, the time complexity would be O(n).

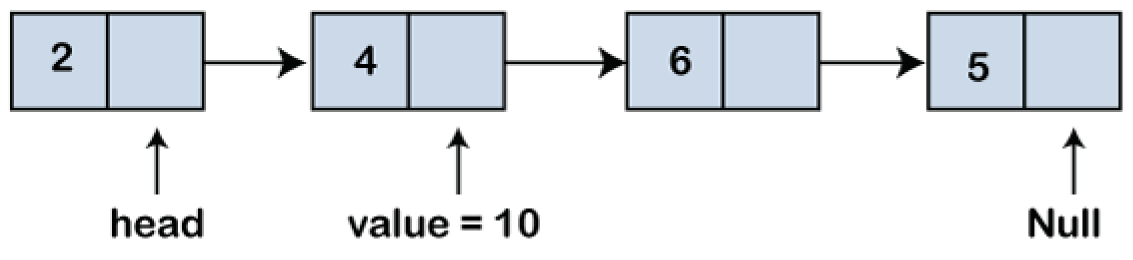
To insert an element at the end of the linked list, we have to traverse the whole list. If the linked list consists of n elements, then the time complexity would be O(n).

- **Inserting an element at the mid**

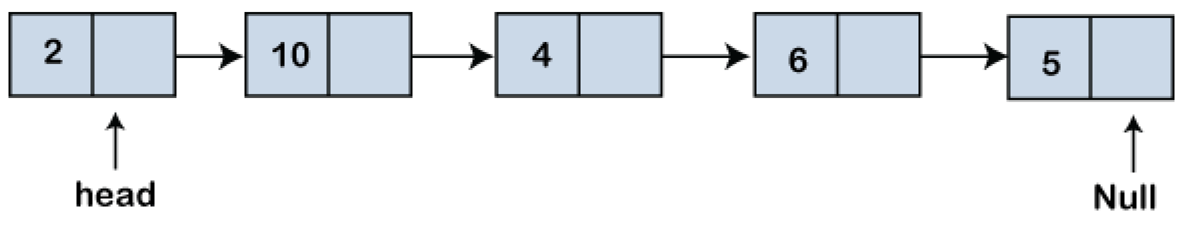
Suppose we want to insert the element at the ith position of the array; we need to shift the n/2 elements towards the right. Therefore, the time complexity is proportional to the number of the elements. The time complexity would be O(n) for the average case.



In the case of linked list, we have to traverse to that position where we have to insert the new element. Even though, we do not have to perform any kind of shifting, but we have to traverse to n/2 position. The time taken is proportional to the n number of elements, and the time complexity for the average case would be O(n).



The resultant linked list is:



- **Ease of use**

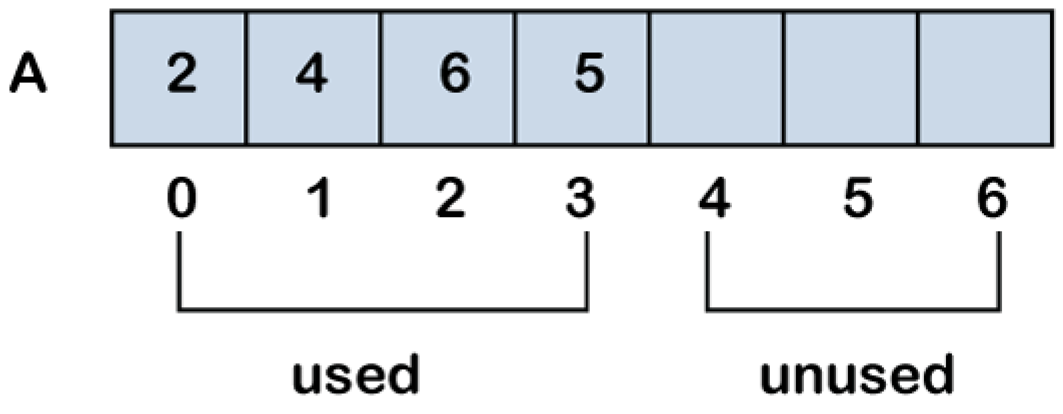
The implementation of an array is easy as compared to the linked list. While creating a program using a linked list, the program is more prone to errors like segmentation fault or memory leak. So, lots of care need to be taken while creating a program in the linked list.

- **Dynamic in size**

The linked list is dynamic in size whereas the array is static. Here, static doesn't mean that we cannot decide the size at the run time, but we cannot change it once the size is decided.

3. Memory requirements

As the elements in an array store in one contiguous block of memory, so array is of fixed size. Suppose we have an array of size 7, and the array consists of 4 elements then the rest of the space is unused. The memory occupied by the 7 elements:



Memory space = 7*4 = 28 bytes

Where 7 is the number of elements in an array and 4 is the number of bytes of an integer type.

In case of linked list, there is no unused memory but the extra memory is occupied by the pointer variables. If the data is of integer type, then total memory occupied by one node is 8 bytes, i.e., 4 bytes for data and 4 bytes for pointer variable. If the linked list consists of 4 elements, then the memory space occupied by the linked list would be:

Memory space = 8*4 = 32 bytes

The linked list would be a better choice if the data part is larger in size. Suppose the data is of 16 bytes. The memory space occupied by the array would be 16*7=112 bytes while the linked list occupies 20*4=80, here we have specified 20 bytes as 16 bytes for the size of the data plus 4 bytes for the pointer variable. If we are choosing the larger size of data, then the linked list would consume a less memory; otherwise, it depends on the factors that we are adopting to determine the size.

Let's look at the differences between the array and linked list in a tabular form.

Array	Linked list
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address.
Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory or randomly stored.
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at

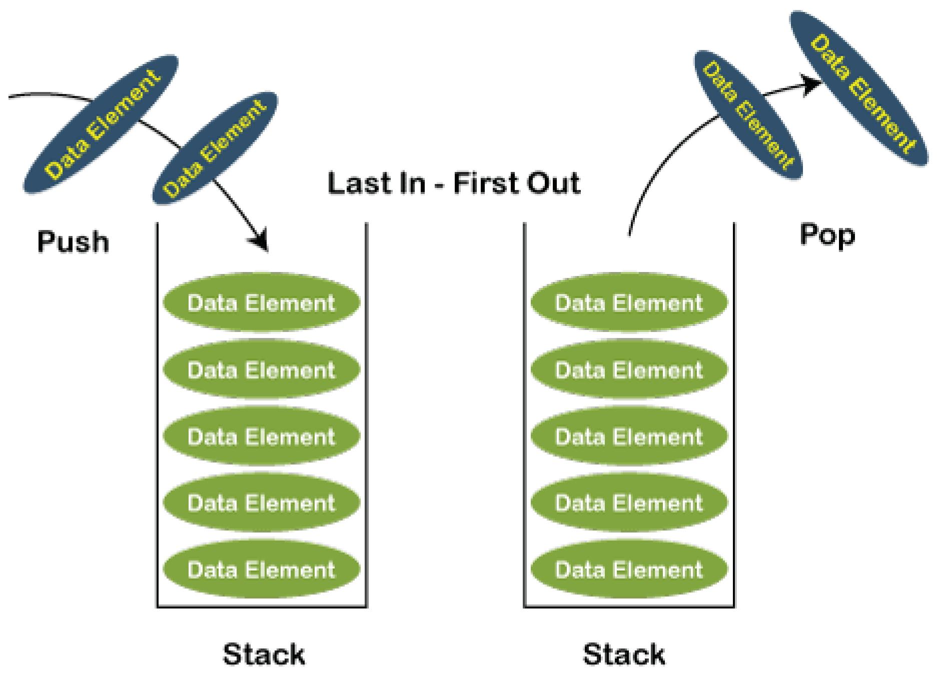
	the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performing any operation like insertion, deletion, etc.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement.

Stack vs. Queue

First, we will look at **what is stack** and **what is queue** individually, and then we will discuss the differences between stack and queue.

What is a Stack?

A Stack is a linear data structure. In case of an array, random access is possible, i.e., any element of an array can be accessed at any time, whereas in a stack, the sequential access is only possible. It is a container that follows the insertion and deletion rule. It follows the principle **LIFO (Last In First Out)** in which the insertion and deletion take place from one side known as a **top**. In stack, we can insert the elements of a similar data type, i.e., the different data type elements cannot be inserted in the same stack. The two operations are performed in LIFO, i.e., **push** and **pop** operation.



The following are the operations that can be performed on the stack:

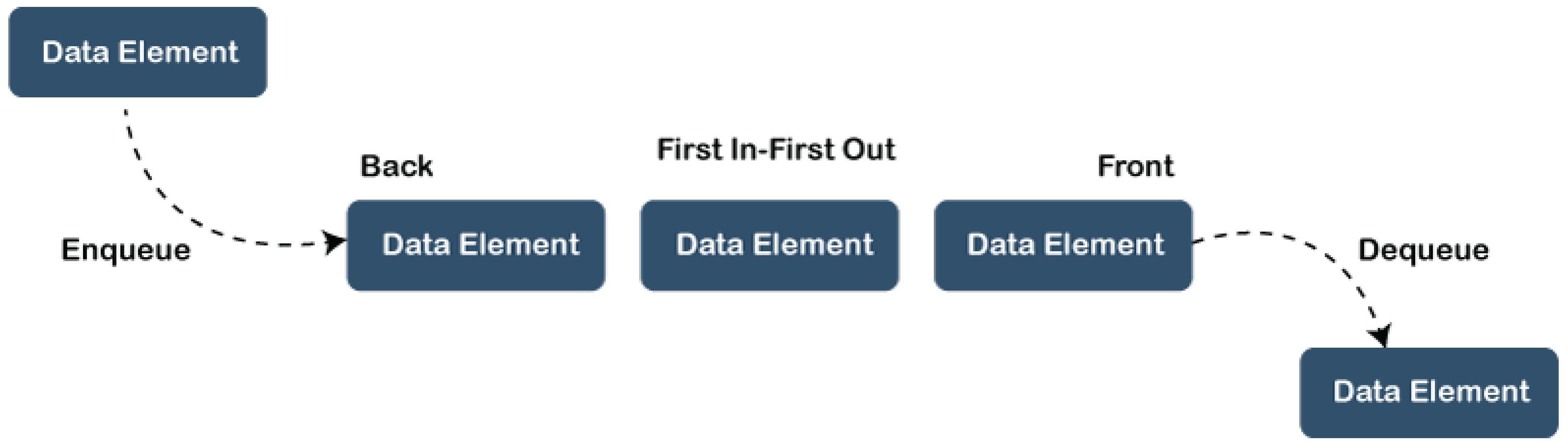
- **push(x):** It is an operation in which the elements are inserted at the top of the stack. In the **push** function, we need to pass an element which we want to insert in a stack.
- **pop():** It is an operation in which the elements are deleted from the top of the stack. In the **pop()** function, we do not have to pass any argument.
- **peek()/top():** This function returns the value of the topmost element available in the stack. Like pop(), it returns the value of the topmost element but does not remove that element from the stack.
- **isEmpty():** If the stack is empty, then this function will return a true value or else it will return a false value.
- **isFull():** If the stack is full, then this function will return a true value or else it will return a false value.

In stack, the **top** is a pointer which is used to keep track of the last inserted element. To implement the stack, we should know the size of the stack. We need to allocate the memory to get the size of the stack. There are two ways to implement the stack:

- **Static:** The static implementation of the stack can be done with the help of arrays.
- **Dynamic:** The dynamic implementation of the stack can be done with the help of a linked list.

What is the Queue?

A Queue is a linear data structure. It is an ordered list that follows the principle FIFO (First In -First Out). A Queue is a structure that follows some restrictions on insertion and deletion. In the case of Queue, insertion is performed from one end, and that end is known as a rear end. The deletion is performed from another end, and that end is known as a front end. In Queue, the technical words for insertion and deletion are **enqueue()** and **dequeue(),** respectively whereas, in the case of the stack, the technical words for insertion and deletion are push() and pop(), respectively. Its structure contains two pointers **front pointer** and **rear pointer**, where the front pointer is a pointer that points to the element that was first added in the queue and the rear pointer that points to the element inserted last in the queue.

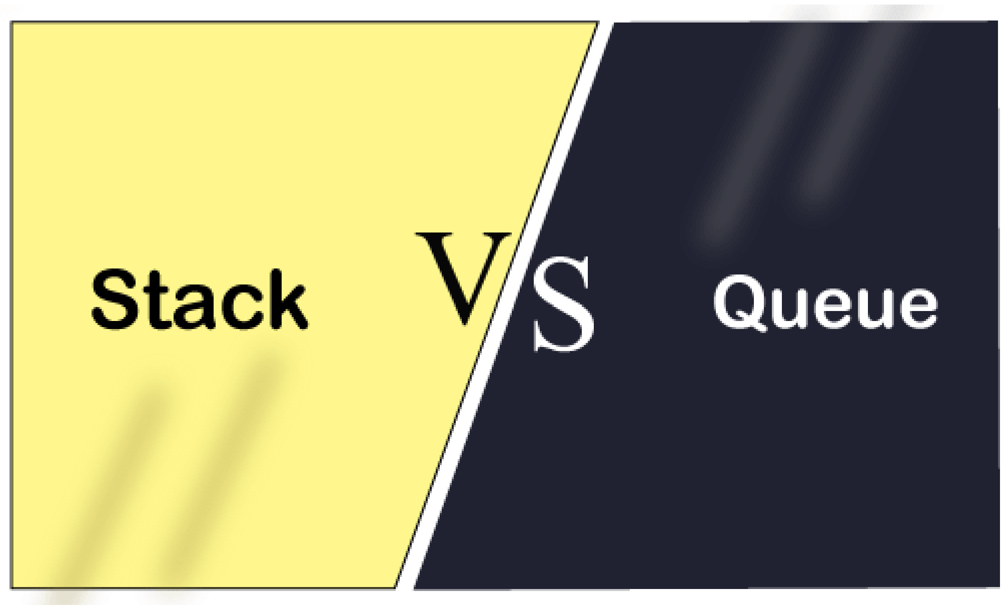


Similarities between stack and queue.

There are two similarities between the stack and queue:

- **Linear data structure**
  Both the stack and queue are the linear data structure, which means that the elements are stored sequentially and accessed in a single run.
- **Flexible in size**
  Both the stack and queue are flexible in size, which means they can grow and shrink according to the requirements at the run-time.

Differences between stack and queue



The following are the differences between the stack and queue:

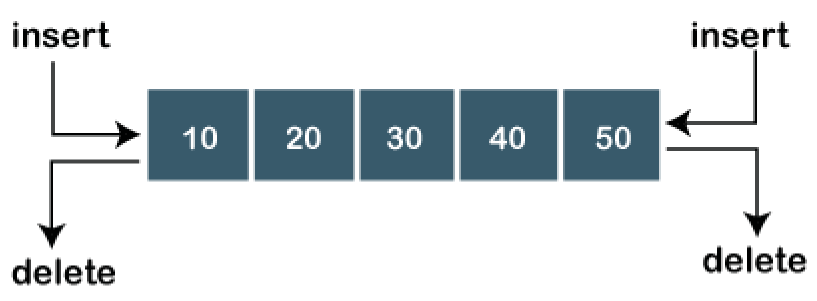
Basis for comparison	Stack	Queue
Principle	It follows the principle LIFO (Last In- First Out), which implies that the element	It follows the principle FIFO (First In -First Out), which implies that the element which

	which is inserted last would be the first one to be deleted.	is added first would be the first element to be removed from the list.
Structure	It has only one end from which both the insertion and deletion take place, and that end is known as a top.	It has two ends, i.e., front and rear end. The front end is used for the deletion while the rear end is used for the insertion.
Number of pointers used	It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack.	It contains two pointers front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue.
Operations performed	It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list.	It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue.
Examination of the empty condition	If top==-1, which means that the stack is empty.	If front== -1 or front = rear+1, which means that the queue is empty.
Examination of full condition	If top== max-1, this condition implies that the stack is full.	If rear==max-1, this condition implies that the stack is full.
Variants	It does not have any types.	It is of three types like priority queue, circular queue and double ended queue.
Implementation	It has a simpler implementation.	It has a comparatively complex implementation than a stack.
Visualization	A Stack is visualized as a vertical collection.	A Queue is visualized as a horizontal collection.

Linear vs Circular Queue

What is a Linear Queue?

A linear queue is a linear data structure that serves the request first, which has been arrived first. It consists of data elements which are connected in a linear fashion. It has two pointers, i.e., front and rear, where the insertion takes place from the front end, and deletion occurs from the front end.



Operations on Linear Queue

There are two operations that can be performed on a linear queue:

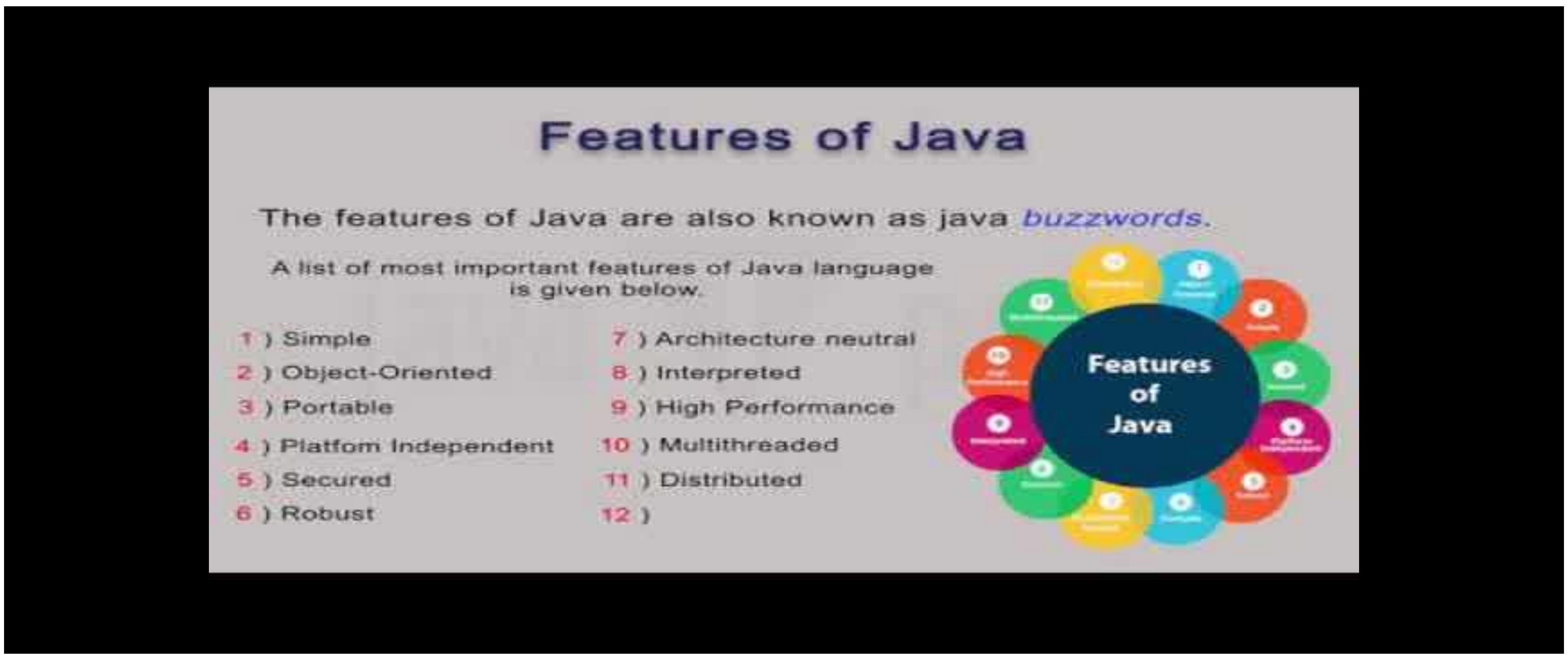
- **Enqueue:** The enqueue operation inserts the new element from the rear end.

- **Dequeue:** The dequeue operation is used to delete the existing element from the front end of the queue.

What is a Circular Queue?

As we know that in a queue, the front pointer points to the first element while the rear pointer points to the last element of the queue. The problem that arises with the linear queue is that if some empty cells occur at the beginning of the queue then we cannot insert new element at the empty space as the rear cannot be further incremented.

A circular queue is also a linear data structure like a normal queue that follows the FIFO principle but it does not end the queue; it connects the last position of the queue to the first position of the queue. If we want to insert new elements at the beginning of the queue, we can insert it using the circular queue data structure.



Features of Java

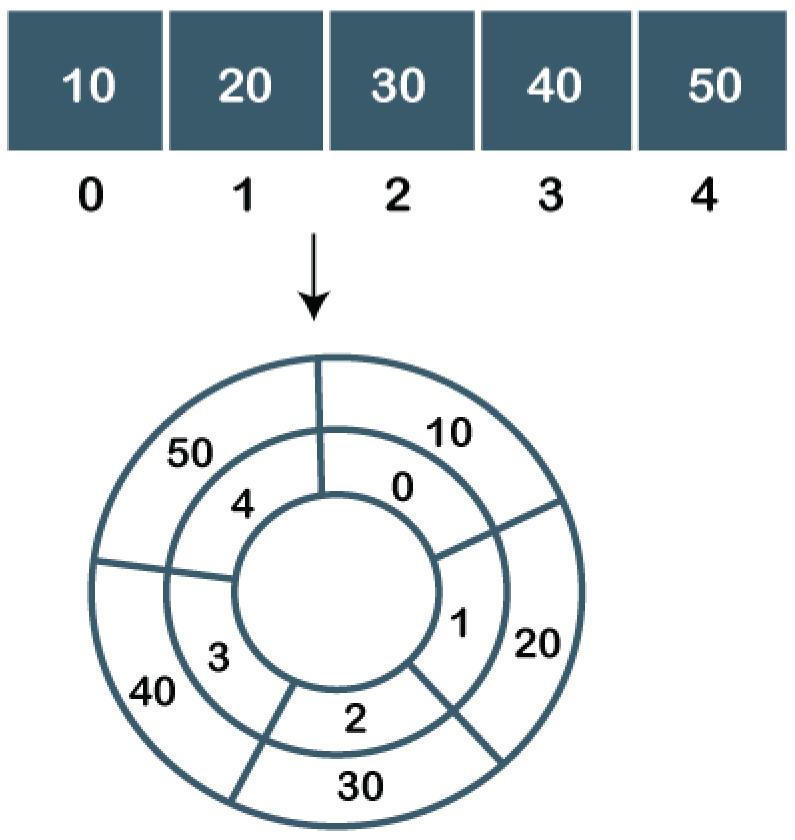
The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.

1) Simple
2) Object-Oriented
3) Portable
4) Platfom Independent
5) Secured
6) Robust
7) Architecture neutral
8) Interpreted
9) High Performance
10) Multithreaded
11) Distributed
12)

Features of Java

In the circular queue, when the rear reaches the end of the queue, then rear is reset to zero. It helps in refilling all the free spaces. The problem of managing the circular queue is overcome if the first position of the queue comes after the last position of the queue.



Conditions for the queue to be a circular queue

- Front ==0 and rear=n-1
- Front=rear+1

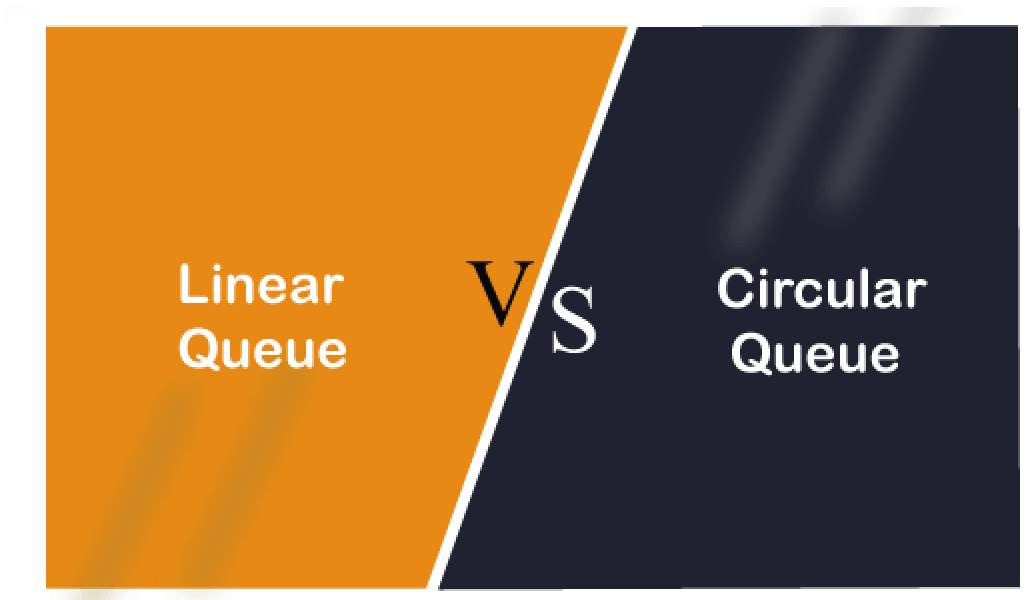
If either of the above conditions is satisfied means that the queue is a circular queue.

Operations on Circular Queue

The following are the two operations that can be performed on a circular queue are:

- **Enqueue:** It inserts an element in a queue. The given below are the scenarios that can be considered while inserting an element:
  1. If the queue is empty, then the front and rear are set to 0 to insert a new element.
  2. If queue is not empty, then the value of the rear gets incremented.
  3. If queue is not empty and rear is equal to n-1, then rear is set to 0.
- **Dequeue:** It performs a deletion operation in the Queue. The following are the points or cases that can be considered while deleting an element:
  1. If there is only one element in a queue, after the dequeue operation is performed on the queue, the queue will become empty. In this case, the front and rear values are set to -1.
  2. If the value of the front is equal to n-1, after the dequeue operation is performed, the value of the front variable is set to 0.
  3. If either of the above conditions is not fulfilled, then the front value is incremented.

Differences between linear Queue and Circular Queue



Basis of comparison	Linear Queue	Circular Queue
Meaning	The linear queue is a type of linear data structure that contains the elements in a sequential manner.	The circular queue is also a linear data structure in which the last element of the Queue is connected to the first element, thus creating a circle.
Insertion and Deletion	In linear queue, insertion is done from the rear end, and deletion is done from the front end.	In circular queue, the insertion and deletion can take place from any end.
Memory space	The memory space occupied by the linear queue is more than the circular queue.	It requires less memory as compared to linear queue.
Memory utilization	The usage of memory is inefficient.	The memory can be more efficiently utilized.
Order of execution	It follows the FIFO principle in order to perform the tasks.	It has no specific order for execution.

Linear Search vs Binary Search

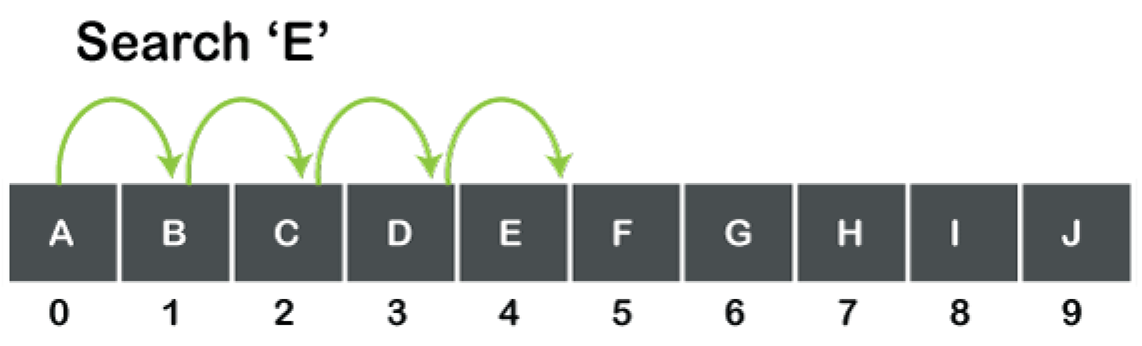
Before understanding the differences between the linear and binary search, we should first know the linear search and binary search separately.

What is a linear search?

A linear search is also known as a sequential search that simply scans each element at a time. Suppose we want to search an element in an array or list; we simply calculate its length and do not jump at any item.

Let's consider a simple example.

Suppose we have an array of 10 elements as shown in the below figure:



The above figure shows an array of character type having 10 values. If we want to search 'E', then the searching begins from the 0th element and scans each element until the element, i.e., 'E' is not found. We cannot directly jump from the 0th element to the 4th element, i.e., each element is scanned one by one till the element is not found.

Complexity of Linear search

As linear search scans each element one by one until the element is not found. If the number of elements increases, the number of elements to be scanned is also increased. We can say that the ***time taken to search the elements is proportional to the number of elements***. Therefore, the worst-case complexity is O(n)

What is a Binary search?

A binary search is a search in which the middle element is calculated to check whether it is smaller or larger than the element which is to be searched. The main advantage of using binary search is that it does not scan each element in the list. Instead of scanning each element, it performs the searching to the half of the list. So, the binary search takes less time to search an element as compared to a linear search.

The one ***pre-requisite of binary search*** is that an array should be in sorted order, whereas the linear search works on both sorted and unsorted array. The binary search algorithm is based on the divide and conquer technique, which means that it will divide the array recursively.

There are three cases used in the binary search:

Case 1: data<a[mid] then left = mid+1.

Case 2: data>a[mid] then right=mid-1

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

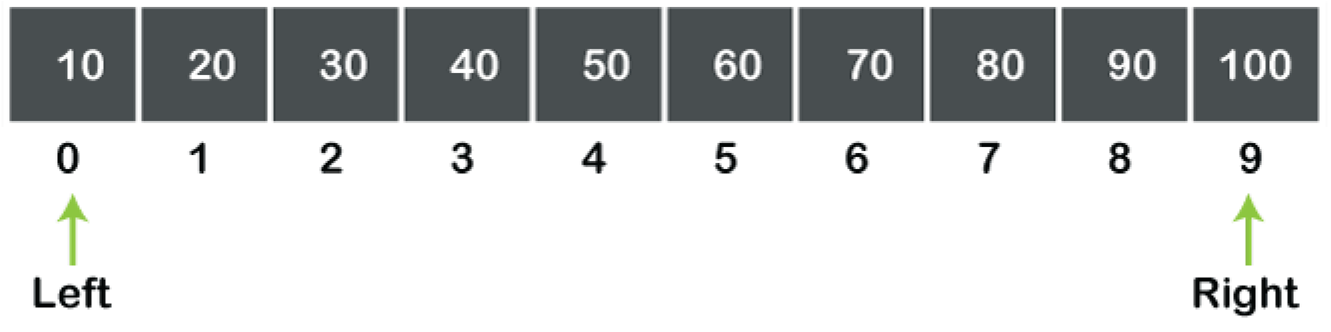
In the above case, '**a**' is the name of the array, **mid** is the index of the element calculated recursively, **data** is the element that is to be searched, **left** denotes the left element of the array and **right** denotes the element that occur on the right side of the array.

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

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

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

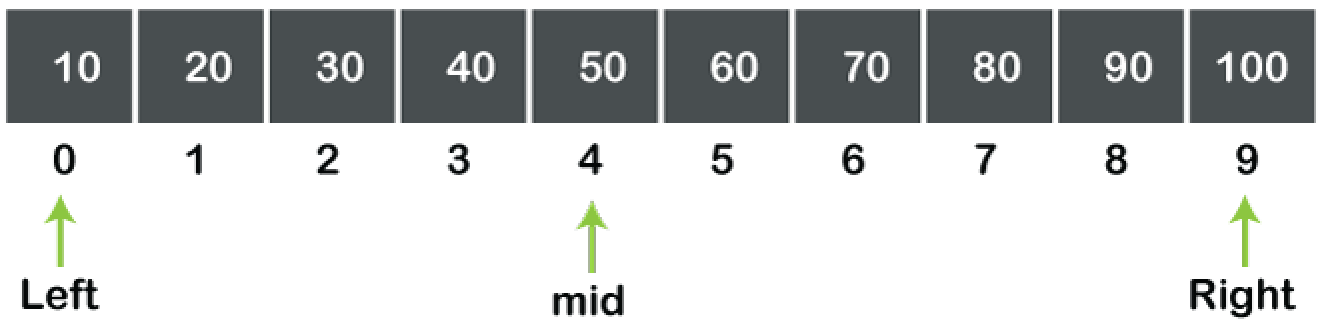
Step 1: First, we calculate the middle element of an array. We consider two variables, i.e., left and right. Initially, left =0 and right=9 as shown in the below figure:



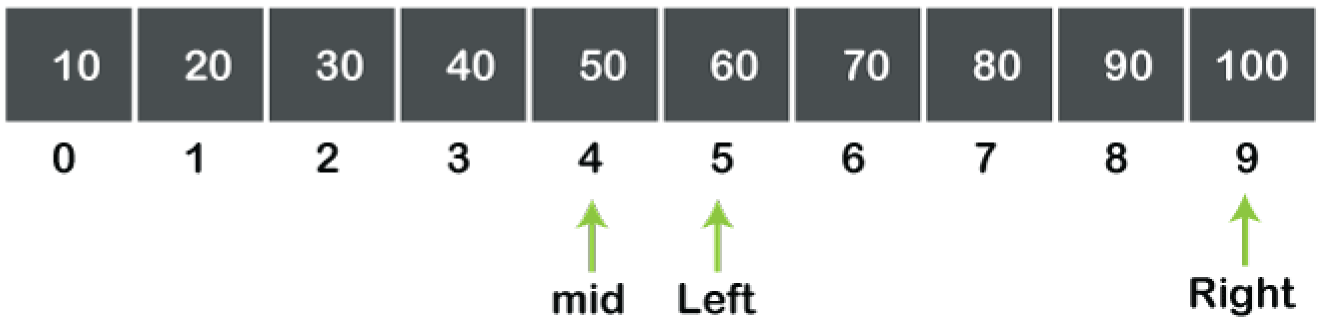
The middle element value can be calculated as:

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

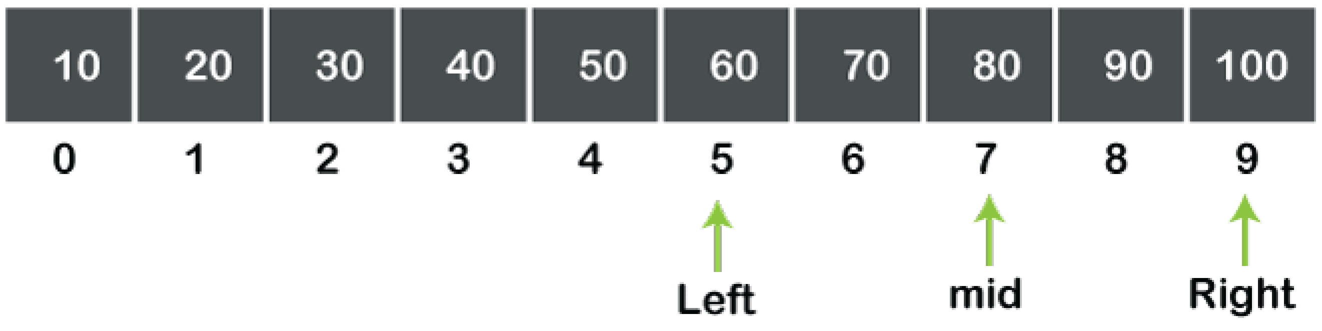
Therefore, mid = 4 and a[mid] = 50. The element to be searched is 70, so a[mid] is not equal to data. The case 2 is satisfied, i.e., data>a[mid].



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

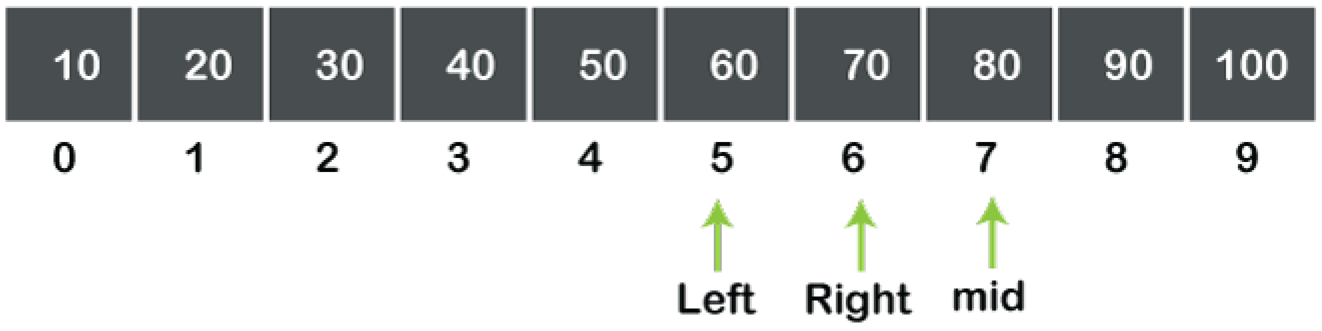


Now again, the mid-value is calculated by using the above formula, and the value of mid becomes 7. Now, the mid can be represented as:

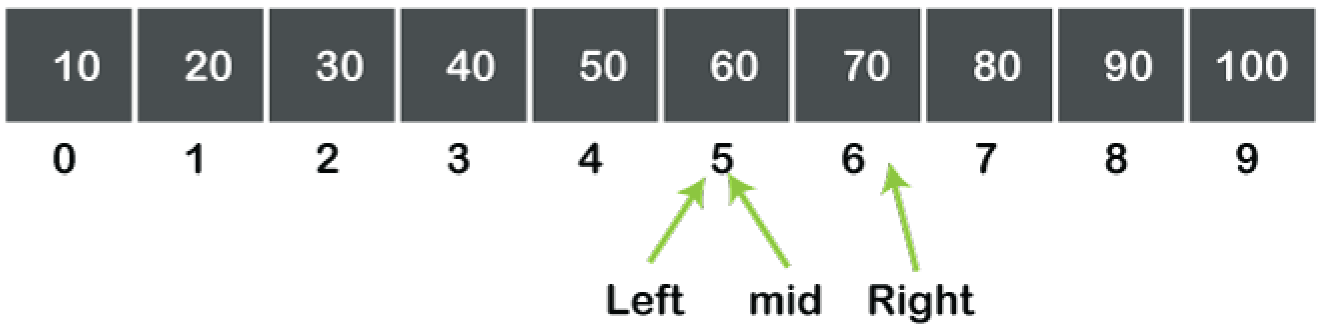


In the above figure, we can observe that a[mid]>data, so again, the value of mid will be calculated in the next step.

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



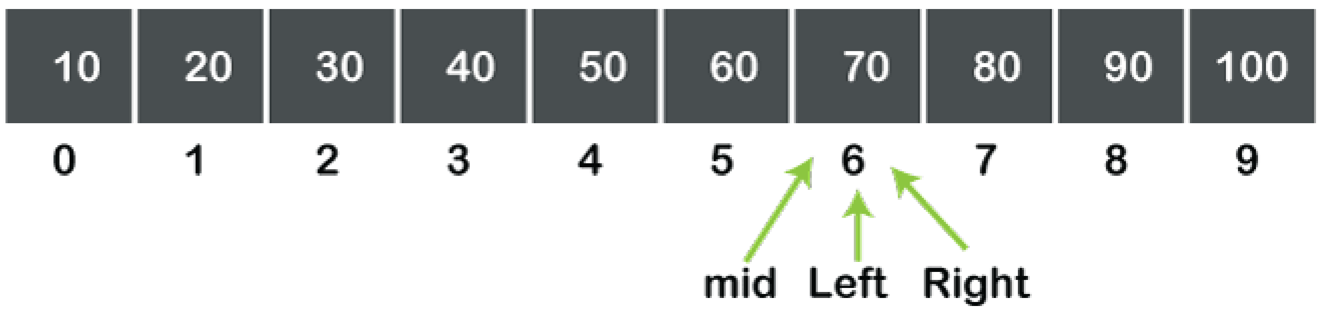
The value of mid will be calculated again. The values of left and right are 5 and 6, respectively. Therefore, the value of mid is 5. Now the mid can be represented in an array as shown below:



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

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

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



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

Differences between Linear search and Binary search

The following are the differences between linear search and binary search:

Description

Linear search is a search that finds an element in the list by searching the element sequentially until the element is found in the list. On the other hand, a binary search is a search that finds the middle element in the list recursively until the middle element is matched with a searched element.

Working of both the searches

The linear search starts searching from the first element and scans one element at a time without jumping to the next element. On the other hand, binary search divides the array into half by calculating an array's middle element.

Implementation

The linear search can be implemented on any linear data structure such as vector, singly linked list, double linked list. In contrast, the binary search can be implemented on those data structures with two-way traversal, i.e., forward and backward traversal.

Complexity

The linear search is easy to use, or we can say that it is less complex as the elements for a linear search can be arranged in any order, whereas in a binary search, the elements must be arranged in a particular order.

Sorted elements

The elements for a linear search can be arranged in random order. It is not mandatory in linear search that the elements are arranged in a sorted order. On the other hand, in a binary search, the elements must be arranged in sorted order. It can be arranged either in an increasing or in decreasing order, and accordingly, the algorithm will be changed. As binary search uses a sorted array, it is necessary to insert the element at the proper place. In contrast, the linear search does not need a sorted array, so that the new element can be easily inserted at the end of the array.

Approach

The linear search uses an iterative approach to find the element, so it is also known as a sequential approach. In contrast, the binary search calculates the middle element of the array, so it uses the divide and conquer approach.

Data set

Linear search is not suitable for the large data set. If we want to search the element, which is the last element of the array, a linear search will start searching from the first element and goes on till the last element, so the time taken to search the element would be large. On the other hand, binary search is suitable for a large data set as it takes less time.

Speed

If the data set is large in linear search, then the computational cost would be high, and speed becomes slow. If the data set is large in binary search, then the computational cost would be less compared to a linear search, and speed becomes fast.

Dimensions

Linear search can be used on both single and multidimensional array, whereas the binary search can be implemented only on the one-dimensional array.

Efficiency

Linear search is less efficient when we consider the large data sets. Binary search is more efficient than the linear search in the case of large data sets.

Let's look at the differences in a tabular form.

Basis of comparison	Linear search	Binary search
Definition	The linear search starts searching from the first element and compares each element with a searched element till the element is not found.	It finds the position of the searched element by finding the middle element of the array.
Sorted data	In a linear search, the elements don't need to be arranged in sorted order.	The pre-condition for the binary search is that the elements must be arranged in a sorted order.
Implementation	The linear search can be implemented on any linear data structure such as an array, linked list, etc.	The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal.
Approach	It is based on the sequential approach.	It is based on the divide and conquer approach.
Size	It is preferrable for the small-sized data sets.	It is preferrable for the large-size data sets.
Efficiency	It is less efficient in the case of large-size data sets.	It is more efficient in the case of large-size data sets.
Worst-case scenario	In a linear search, the worst- case scenario for finding the element is O(n).	In a binary search, the worst-case scenario for finding the element is $O(\log_2 n)$.
Best-case scenario	In a linear search, the best-case scenario for finding the first element in the list is O(1).	In a binary search, the best-case scenario for finding the first element in the list is O(1).

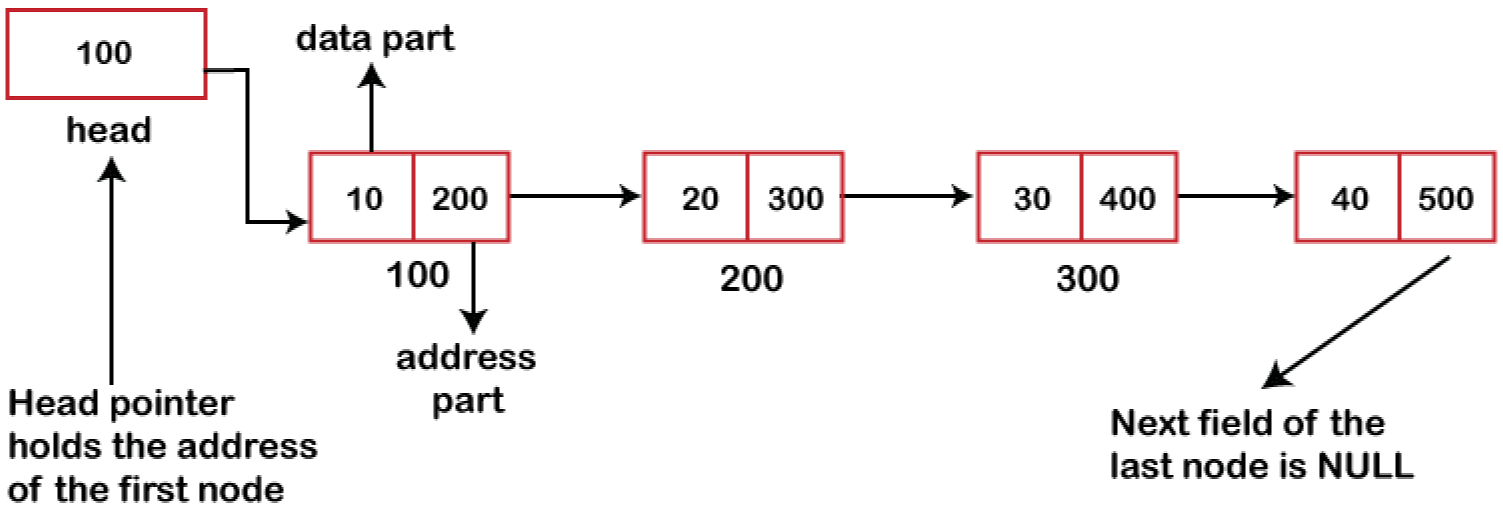
Dimensional array	It can be implemented on both a single and multidimensional array.	It can be implemented only on a multidimensional array.

Singly Linked List vs Doubly Linked List

Before looking at the differences between the singly linked list and doubly linked list, we first understand what is singly linked list and doubly linked list separately.

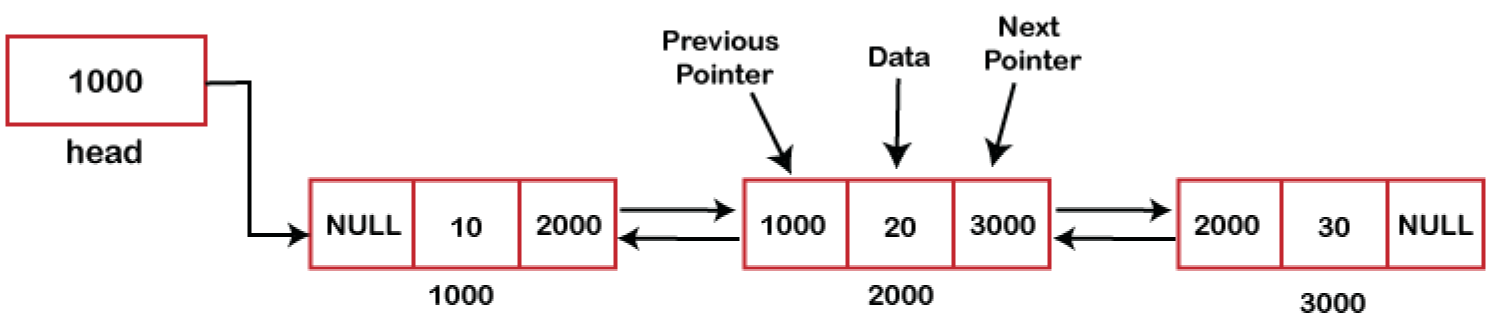
What is a singly linked list?

A singly linked list can be simply called a linked list. A singly linked list is a list that consists of a collection of nodes, and each node has two parts; one part is the data part, and another part is the address. The singly linked can also be called a chain as each node refers to another node through its address part. We can perform various operations on a singly linked list like insertion, deletion, and traversing.


100
head
data part
10
200
100
20
300
200
30
400
300
40
500
address part
Head pointer holds the address of the first node
Next field of the last node is NULL

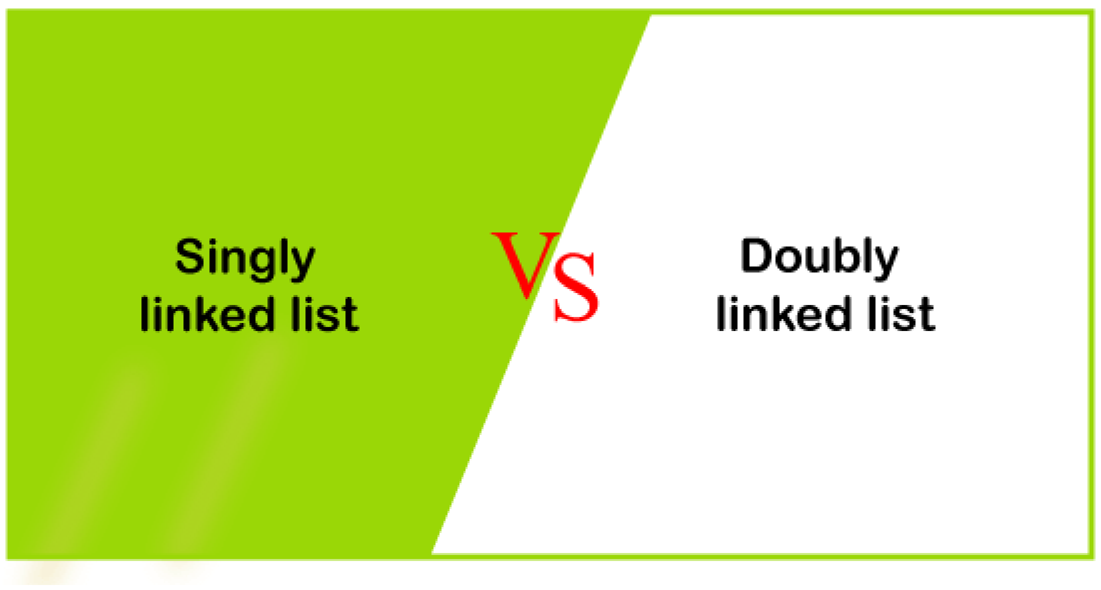
What is a doubly-linked list?

A doubly linked list is another type of the linked list. It is called a doubly linked list because it contains two addresses while a singly linked list contains a single address. It is a list that has total three parts, one is a data part, and others two are the pointers, i.e., previous and next. The previous pointer holds the address of the previous node, and the next pointer holds the address of the next node. Therefore, we can say that list has two references, i.e., forward and backward reference to traverse in either direction.



We can also perform various operations on a doubly-linked list like insertion, deletion, and traversing.

Differences between the singly-linked list and doubly linked list.



The differences between the singly-linked list and doubly linked list are given below:

- **Definition**

The singly-linked is a linear data structure that consists of a collection of nodes in which one node consists of two parts, i.e., one is the data part, and another one is the address part. In contrast, a doubly-linked list is also a linear data structure in which the node consists of three parts, i.e., one is the data part, and the other two are the address parts.

- **Direction**

As we know that in a singly linked list, a node contains the address of the next node, so the elements can be traversed in only one direction, i.e., forward direction. In contrast, in a doubly-linked list, the node contains two pointers (previous pointer and next pointer) that hold the ***address of the next node*** and the ***address of the previous node, respectively*** so elements can be traversed in both directions.

- **Memory space**

The singly linked list occupies less memory space as it contains a single address. We know that the pointer variable stores the address, and the pointer variable occupies 4 bytes; therefore, the memory space occupied by the pointer variable in the singly linked list is also 4 bytes. The doubly linked list holds two addresses in a node, one is of the next node and the other one is of the previous node; therefore, the space occupied by the two pointer variables is 8 bytes.

- **Insertion and Deletion**

The insertion and deletion in a singly-linked list are less complex than a doubly linked list. If we insert an element in a singly linked list then we need to update the address of only next node. On the other hand, in the doubly linked list, we need to update the address of both the next and the previous node.

Let's look at the differences in a tabular form.

Basis of comparison	Singly linked list	Doubly linked list
Definition	A single linked list is a list of nodes in which node has two parts, the first part is the data part, and the next part is the pointer pointing to the next node in the sequence of nodes.	A doubly linked list is also a collection of nodes in which node has three fields, the first field is the pointer containing the address of the previous node, the second is the data field, and the third is the pointer containing the address of the next node.
Access	The singly linked list can be traversed only in the forward direction.	The doubly linked list can be accessed in both directions.
List pointer	It requires only one list pointer variable, i.e., the head pointer pointing to the first node.	It requires two list pointer variables, **head** and **last**. The head pointer points to the first node, and the last pointer points to the last node of the list.
Memory space	It utilizes less memory space.	It utilizes more memory space.
Efficiency	It is less efficient as compared to a doubly-linked list.	It is more efficient.
Implementation	It can be implemented on the stack.	It can be implemented on stack, heap and binary tree.

Complexity	In a singly linked list, the time complexity for inserting and deleting an element from the list is **O(n)**.	In a doubly-linked list, the time complexity for inserting and deleting an element is **O(1)**.

Binary tree vs Binary Search tree

First, we will understand the ***binary tree*** and ***binary search tree*** separately, and then we will look at the differences between a binary tree and a binary search tree.

What is a Binary tree?

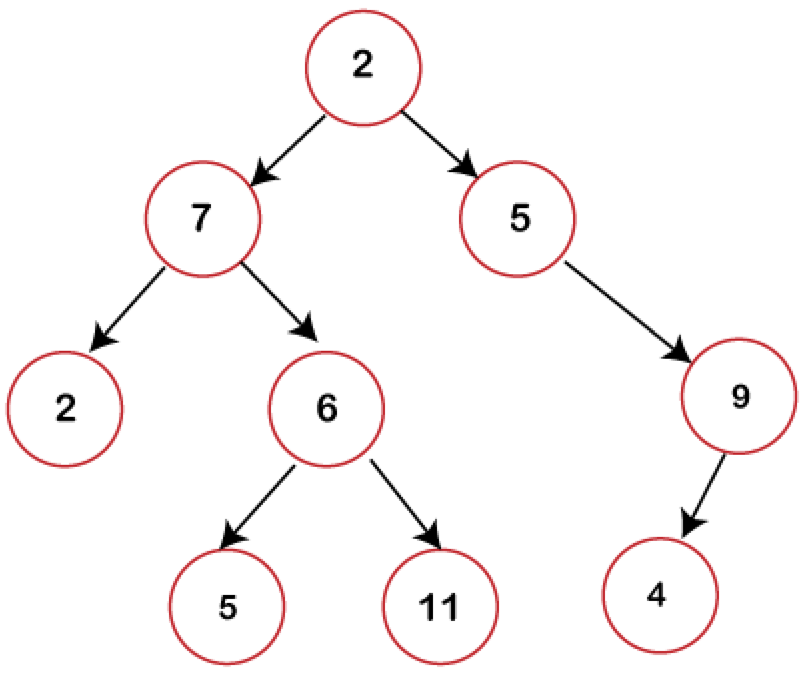
A Binary tree is a non-linear data structure in which a node can have either **0, 1** or **maximum 2 nodes**. Each node in a binary tree is represented either as a parent node or a child node. There can be two children of the parent node, i.e., **left child** and **right child**.

There is only one way to reach from one node to its next node in a binary tree.

A node in a binary tree has three fields:

- **Pointer to the left child:** It stores the reference of the left-child node.
- **Pointer to the right child:** It stores the reference of the right-child node.
- **Data element:** The data element is the value of the data which is stored by the node.

The binary tree can be represented as:



In the above figure, we can observe that each node contains utmost 2 children. If any node does not contain left or right child then the value of the pointer with respect to that child would be NULL.

Basic terminologies used in a Binary tree are:

- **Root node:** The root node is the first or the topmost node in a binary tree.
- **Parent node:** When a node is connected to another node through edges, then that node is known as a parent node. In a binary tree, parent node can have a maximum of 2 children.
- **Child node:** If a node has its predecessor, then that node is known as a ***child node***.
- **Leaf node:** The node which does not contain any child known as a ***leaf node***.
- **Internal node:** The node that has atleast 2 children known as an ***internal node***.
- **Depth of a node:** The distance from the root node to the given node is known as a ***depth of a node***. We provide labels to all the nodes like root node is labeled with 0 as it has no depth, children of the root nodes are labeled with 1, children of the root child are labeled with 2.
- **Height:** The longest distance from the root node to the leaf node is the ***height of the node***.

In a binary tree, there is one tree known as a ***perfect binary tree***. It is a tree in which all the internal nodes must contain two nodes, and all the leaf nodes must be at the same depth. In the case of a perfect binary tree, the total number of nodes exist in a binary tree can be calculated by using the following equation:

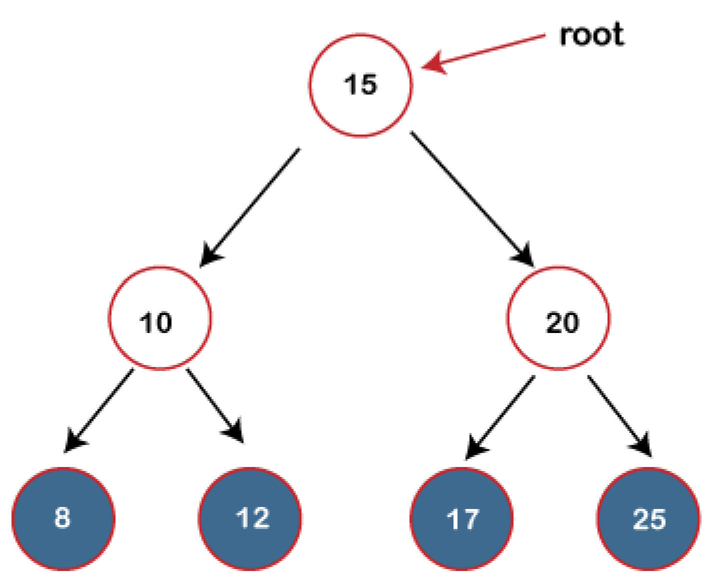
$n = 2^{m+1}-1$

where n is the number of nodes, m is the depth of a node.

What is a Binary Search tree?

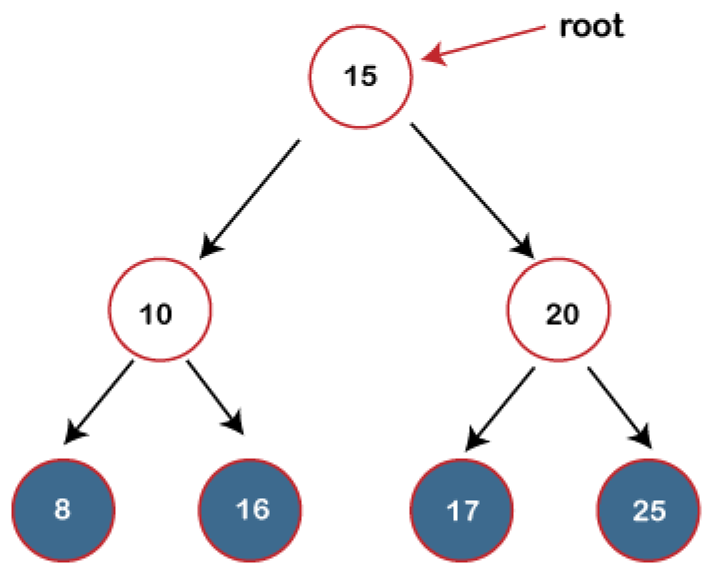
A Binary search tree is a tree that follows some order to arrange the elements, whereas the binary tree does not follow any order. In a Binary search tree, the value of the left node must be smaller than the parent node, and the value of the right node must be greater than the parent node.

Let's understand the concept of a binary search tree through examples.



In the above figure, we can observe that the value of the root node is 15, which is greater than the value of all the nodes in the left subtree. The value of root node is less than the values of all the nodes in a right-subtree. Now, we move to the left-child of the root node. 10 is greater than 8 and lesser than 12; it also satisfies the property of the Binary search tree. Now, we move to the right-child of the root node; the value 20 is greater than 17 and lesser than 25; it also satisfies the property of binary search tree. Therefore, we can say that the tree shown above is the binary search tree.

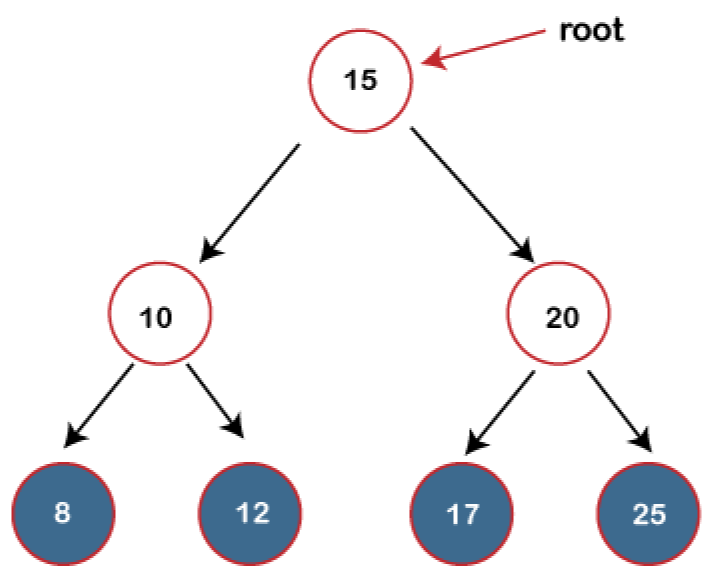
Now, if we change the value of 12 to 16 in the above binary tree, we have to find whether it is still a binary search tree or not.



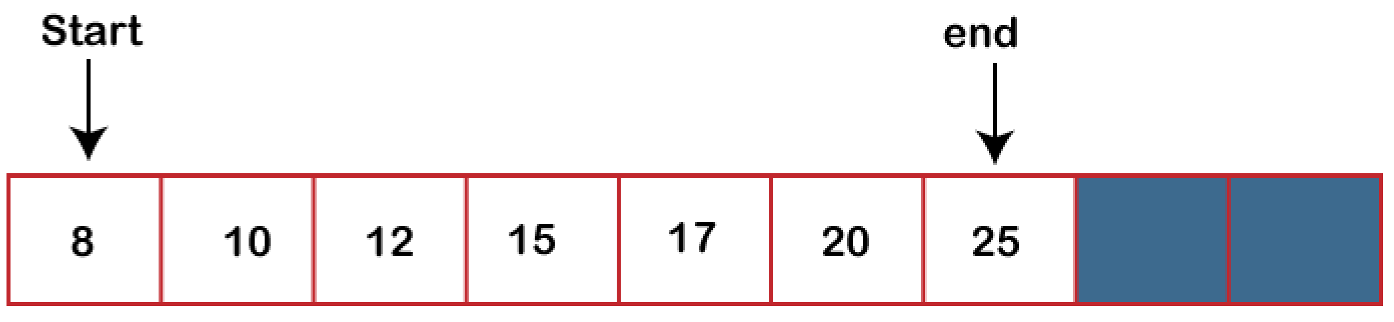
The value of the root node is 15 which is greater than 10 but lesser than 16, so it does not satisfy the property of the Binary search tree. Therefore, it is not a binary search tree.

Operations on Binary search tree

We can perform insert, delete and search operations on the binary search tree. Let's understand how a search is performed on a binary search. The binary tree is shown below on which we have to perform the search operation:



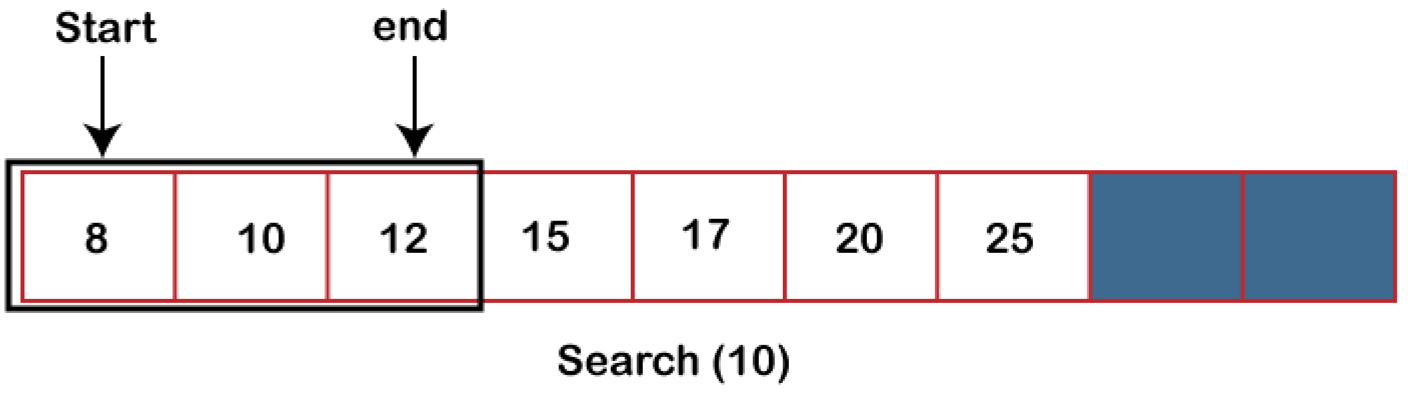
Suppose we have to search 10 in the above binary tree. To perform the binary search, we will consider all the integers in a sorted array. First, we create a complete list in a search space, and all the numbers will exist in the search space. The search space is marked by two pointers, i.e., start and end. The array of the above binary tree can be represented as



First, we will calculate the middle element and compare the middle element with the element, which is to be searched. The middle element is calculated by using n/2. The value of n is 7; therefore, the middle element is 15. The middle element is not equal to the searched element, i.e., 10.

Note: If the element is being searched is lesser than the mid element, then the searching will be performed in the left half; else, searching will be done on the right half. In the case of equality, the element is found.

As the element to be searched is lesser than the mid element, so searching will be performed on the left array. Now the search is reduced to half, as shown below:

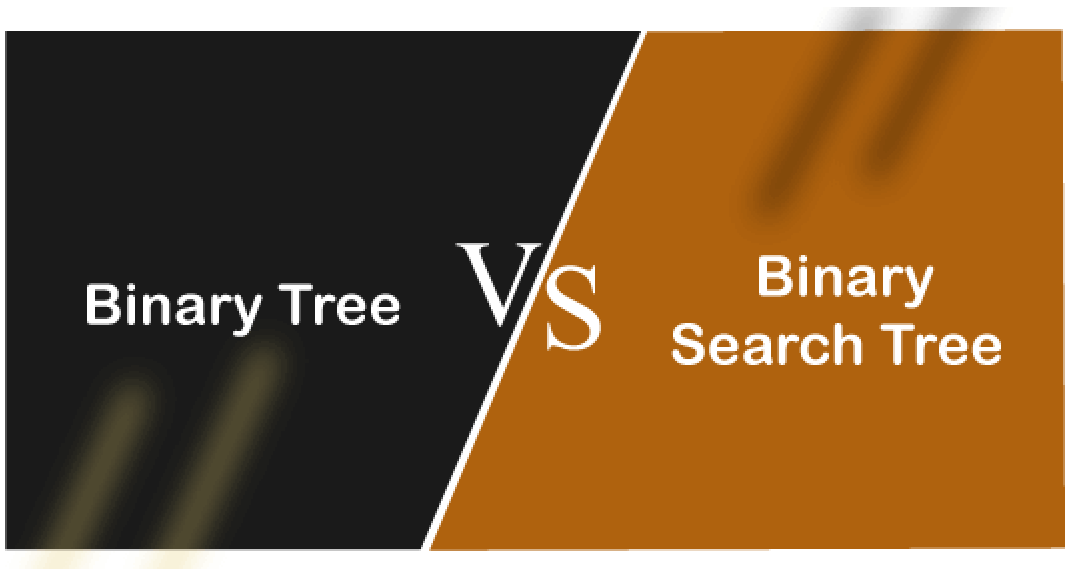


The mid element in the left array is 10, which is equal to the searched element.

Time complexity

In a binary search, there are n elements. If the middle element is not equal to the searched element, then the search space is reduced to n/2, and we will keep on reducing the search space by n/2 until we found the element. In the whole reduction, if we move from n to n/2 to n/4 and so on, then it will take $\log_2 n$ steps.

Differences between Binary tree and Binary search tree

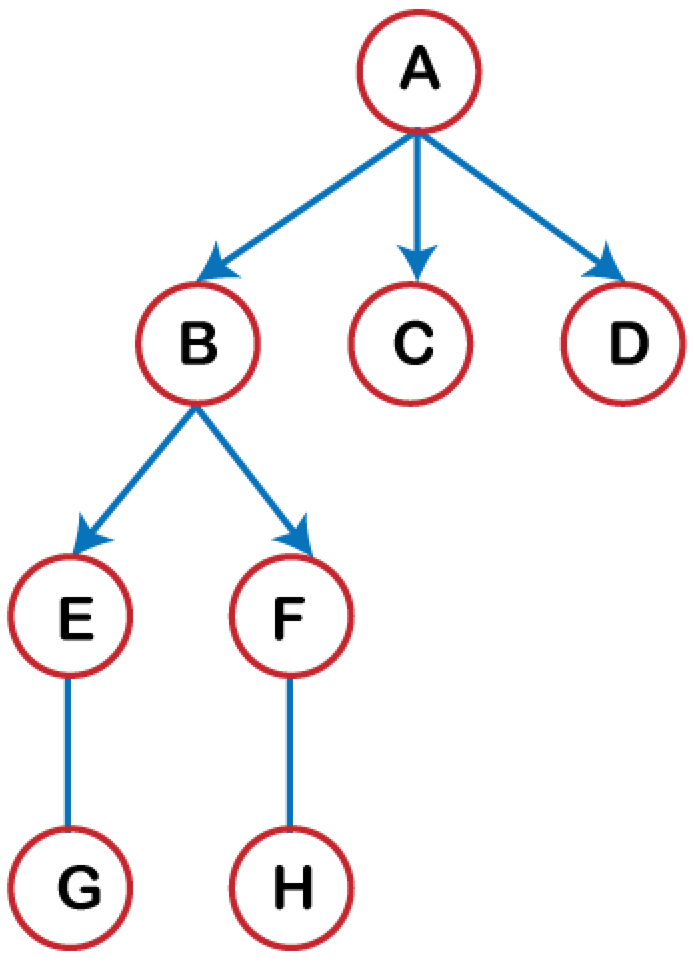


Basis for comparison	Binary tree	Binary search tree
Definition	A binary tree is a non-linear data structure in which a node can have utmost two children, i.e., a node can have 0, 1 or maximum two children. A binary search tree is an ordered binary tree in which some order is followed to organize the nodes in a tree.	
Structure	The structure of the binary tree is that the first node or the topmost node is known as the root node. Each node in a binary tree contains the left pointer and the right pointer. The left pointer contains the address of the left subtree, whereas right pointer contains the address of right subtree.	The binary search tree is one of the types of binary tree that has the value of all the nodes in the left subtree lesser or equal to the root node, and the value of all the nodes in a right subtree are greater than or equal to the value of the root node.
Operations	The operations that can be implemented on a binary tree are insertion, deletion, and traversal.	Binary search trees are the sorted binary trees that provide fast insertion, deletion and search. Lookups mainly implement binary search as all the keys are arranged in sorted order.
types	Four types of binary trees are Full Binary Tree, Complete Binary Tree, Perfect Binary Tree, and Extended Binary Tree.	There are different types of binary search trees such as AVL trees, Splay tree, Tango trees, etc.

Tree vs Graph data structure

Before knowing about the tree and graph data structure, we should know the linear and non-linear data structures. Linear data structure is a structure in which all the elements are stored sequentially and have only single level. In contrast, a non-linear data structure is a structure that follows a hierarchy, i.e., elements are arranged in multiple levels.

Let's understand the structure that forms the hierarchy.



In the above figure, we can assume the company hierarchy where A represents the CEO of the company, B, C and D represent the managers of the company, E and F represent the team leaders, and G and H represent the team members. This type of structure has more than one level, so it is known as a non-linear data structure.

What is Tree?

A tree is a non-linear data structure that represents the hierarchy. A tree is a collection of nodes that are linked together to form a hierarchy.

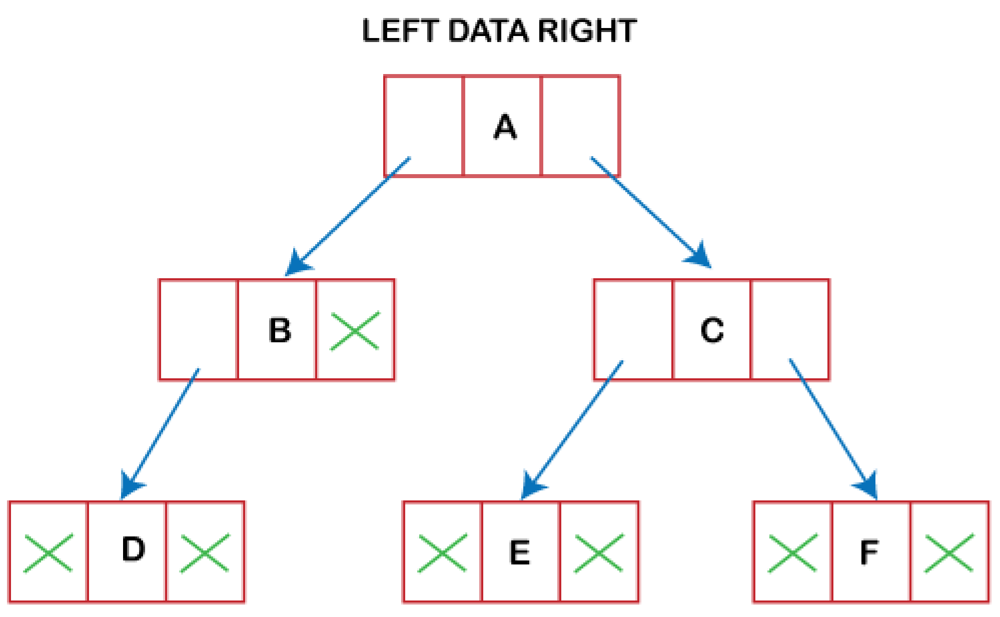
Let's look at some terminologies used in a ***tree*** data structure.

- **Root node:** The topmost node in a tree data structure is known as a root node. A root node is a node that does not have any parent.
- **Parent of a node:** The immediate predecessor of a node is known as a parent of a node. Here predecessor means the previous node of that particular node.
- **Child of a node:** The immediate successor of a node is known as a ***child of a node***.
- **Leaf node:** The leaf node is a node that does not have any child node. It is also known as an external node.
- **Non-leaf node:** The non-leaf node is a node that has atleast one child node. It is also known as an ***internal node***.
- **Path:** It is a sequence of the consecutive edges from a source node to the destination node. Here edge is a link between two nodes.
- **Ancestor:** The predecessor nodes that occur in the path from the root to that node is known as an ancestor.
- **Descendant:** The successor nodes that exist in the path from that node to the leaf node.
- **Sibling:** All the children that have the same parent node are known as siblings.
- **Degree:** The number of children of a particular node is known as a degree.
- **Depth of node:** The length of the path from the root to that node is known as a depth of a node.
- **Height of a node:** The number of edges that occur in the longest path from that node to the leaf node is known as the height of a node.
- **Level of node:** The number of edges that exist from the root node to the given node is known as a level of a node.

Note: If there are n number of nodes in the tree, then there would be (n-1) number of edges.

How is a tree represented in the memory?

Each node will contain three parts, data part, address of the left subtree, and address of the right subtree. If any node does not have the child, then both link parts will have NULL values.



What is a Graph?

A graph is like a tree data structure is a collection of objects or entities known as nodes that are connected to each other through a set of edges. A tree follows some rule that determines the relationship between the nodes, whereas graph does not follow any rule that defines the relationship among the nodes. A graph contains a set of edges and nodes, and edges can connect the nodes in any possible way.

Mathematically, it can be defined as an ordered pair of a set of vertices, and a set of nodes where vertices are represented by 'V' and edges are represented by 'E'.

G= (V , E)

Here we are referring to an ordered pair because the first object must be the set of vertices, and the second object must be a set of edges.

In Graph, each node has a different name or index to uniquely identify each node in the graph. The graph shown below has eight vertices named as v1, v2, v3, v4, v5, v6, v7, and v8. There is no first node, a second node, a third node and so on. There is no ordering of the nodes. Now, we will see how can we represent the edges in a graph?. An edge can be represented by the two endpoints in the graph. We can write the name of the two endpoints as a pair, that represents the edge in a graph.

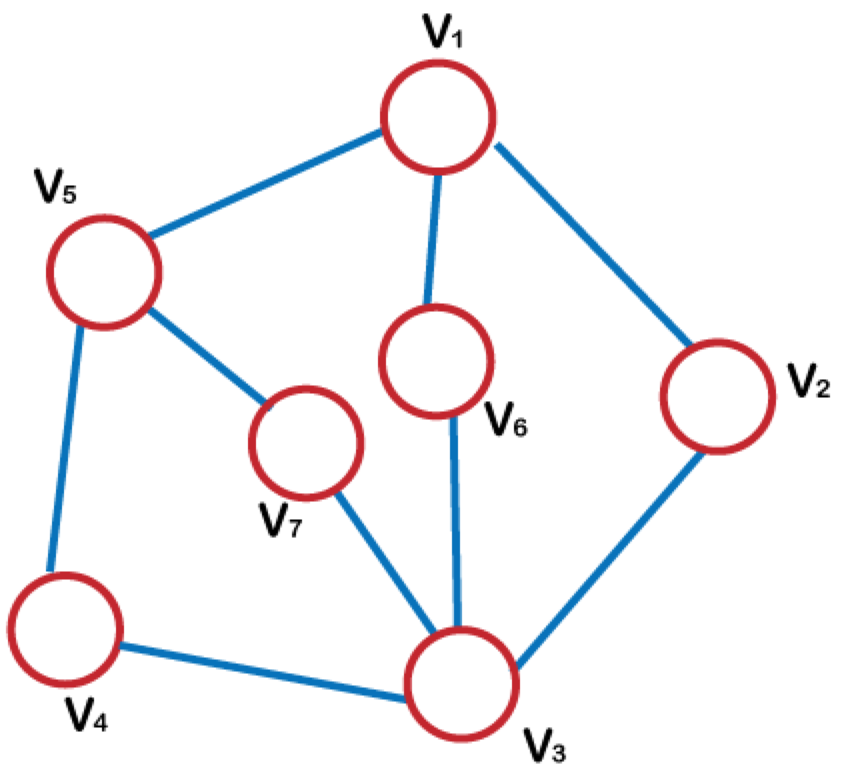
There are two types of edges:

- **Directed edge:** The directed edge represents one endpoint as an origin and another point as a destination. The directed edge is one-way. For example, there are two vertices U and V; then directed edge would represent the link or path from U to V, but no path exists from V to U. If we want to create a path from V to U, then we need to have one more directed edge from V to U. The directed edge can be represented as an ordered pair in which the first element is the origin, whereas the second element is the destination.
- **Undirected edge:** The undirected edge is two-way means that there is no ***origin*** and ***destination***. For example, there are two vertices U and V, then undirected would represent two paths, i.e., from U to V as well as from V to U. An undirected edge can be represented as an unordered pair because the edge is ***bi-directional***.

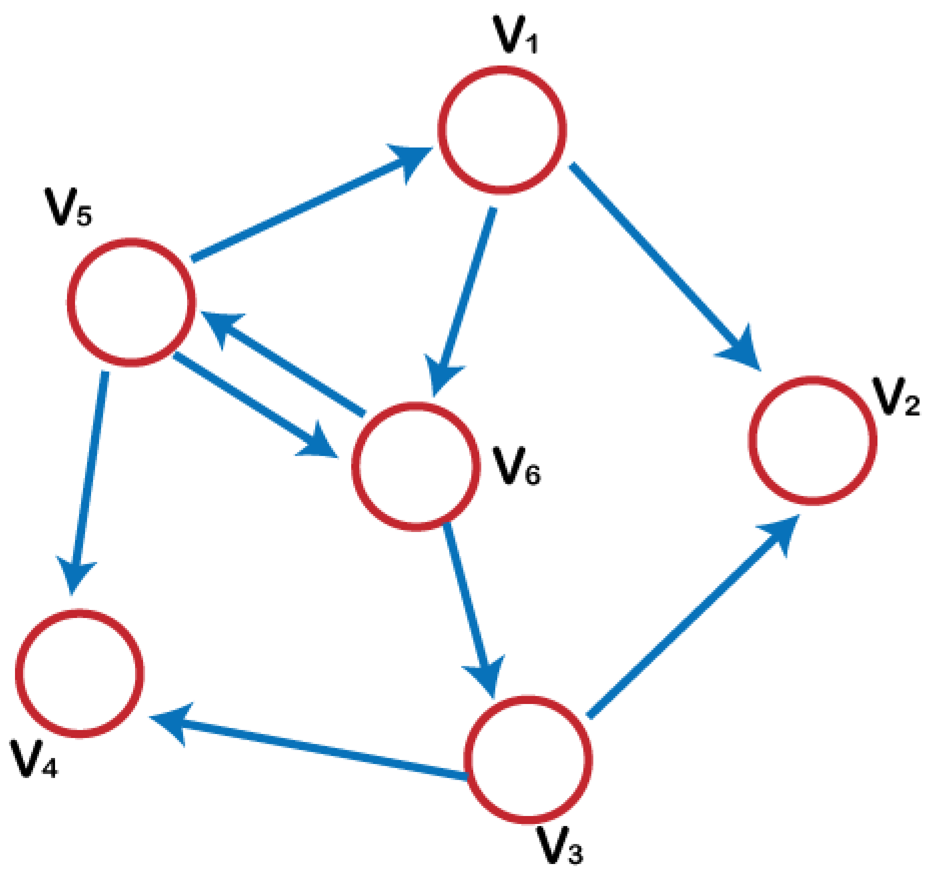
The tree data structure contains only directed edges, whereas the graph can have both types of edges, i.e., ***directed as well as undirected***. But, we consider the graph in which all the edges are either directed edges or undirected edges.

There are two types of graphs:

Directed graph: The graph with the directed edges known as a ***directed graph***.



Undirected graph: The graph with the undirected edges known as a ***undirected graph***. The directed graph is a graph in which all the edges are uni-directional, whereas the undirected graph is a graph in which all the edges are bi-directional.



Differences between tree and graph data structure.

Basis for comparison	Tree	Graph

Definition	Tree is a non-linear data structure in which elements are arranged in multiple levels.	A Graph is also a non-linear data structure.
Structure	It is a collection of edges and nodes. For example, node is represented by N and edge is represented as E, so it can be written as: T = {N,E}	It is a collection of vertices and edges. For example, vertices are represented by V, and edge is represented as 'E', so it can be written as: T = {V, E}
Root node	In tree data structure, there is a unique node known as a parent node. It represents the topmost node in the tree data structure.	In graph data structure, there is no unique node.
Loop formation	It does not create any loop or cycle.	In graph, loop or cycle can be formed.
Model type	It is a hierarchical model because nodes are arranged in multiple level, and that creates a hierarchy. For example, any organization will have a hierarchical model.	It is a network model. For example, facebook is a social network that uses the graph data structure.
Edges	If there are n nodes then there would be n-1 number of edges.	The number of edges depends on the graph.
Type of edge	Tree data structure will always have directed edges.	In graph data structure, all the edges can either be directed edges, undirected edges, or both.
Applications	It is used for inserting, deleting or searching any element in tree.	It is mainly used for finding the shortest path in the network.

Binary Search tree vs AVL tree

Before knowing about the Binary search tree and AVL tree differences, we should know about the binary search tree and AVL tree separately.

What is a Binary Search Tree?

The binary search tree is a tree data structure that follows the condition of the binary tree. As we know, that tree can have 'n' number of children, whereas; the binary tree can contain the utmost two children. So, the constraint of a binary tree is also followed by the binary search tree. Each node in a binary search tree should have the utmost two children; in other words, we can say that the binary search tree is a variant of the binary tree.

The binary search tree also follows the properties of the binary search. In binary search, all the elements in an array must be in sorted order. We calculate the middle element in the binary search in which the left part of the middle element contains the value lesser than the middle value, and the right part of the middle element contains the values greater than the middle value.

In Binary Search Tree, the middle element becomes the root node, the right part becomes the right subtree, and the left part becomes the left subtree. Therefore, we can say that the binary search tree is a combination of a ***binary tree*** and ***binary search***.

Note: Binary Search Tree can be defined as the binary tree in which all the elements of the left subtree are less than the root node, and all the elements of the right subtree are greater than the root node.

Time Complexity in Binary Search Tree

If the binary search tree is almost a balanced tree then all the operations will have a time complexity of **O(logn)** because the search is divided either to the left or the right subtree.

If the binary search tree is either left or right-skewed, then all the operations will have the time complexity of **O(n)** because we need to traverse till the n elements.

What is the AVL Tree?

An AVL tree is a self-balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one. This difference is known as a balance factor. In the AVL tree, the values of balance factor could be either -1, 0 or 1.

How does the self-balancing of the binary search tree happen?

As we know that AVL tree is a self-balancing binary search tree. If the binary search tree is not balanced, it can be self-balanced with some re-arrangements. These re-arrangements can be done using some rotations.

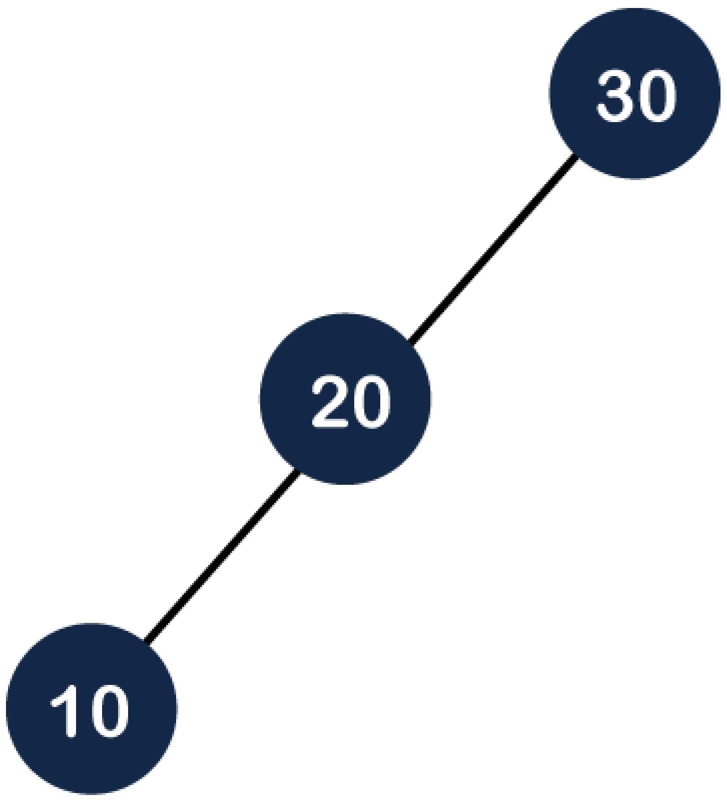
Let's understand self-balancing through an example.

Suppose we want to insert **10, 20, 30** in an AVL tree.

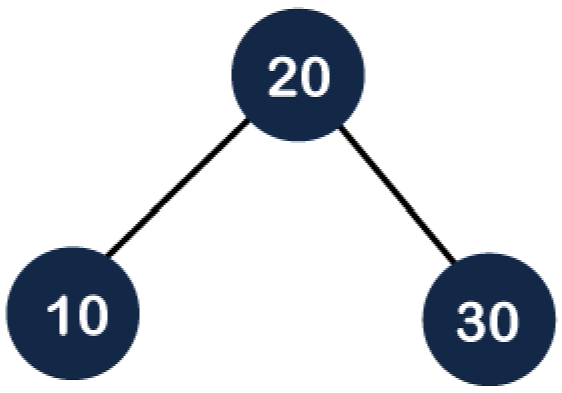
The following are the ways of inserting 10, 20, 30 in an AVL tree:

- **If the order of insertion is 30, 20, 10.**

Step 1: First, we create a Binary search tree, as shown below:



Step 2: In the above figure, we can observe that tree is unbalanced because the balance factor of node 30 is 2. In order to make it an AVL tree, we need to perform some rotations. If we perform the right rotation on node 20 then the node 30 will move downwards, whereas the node 20 will move upwards, as shown below:

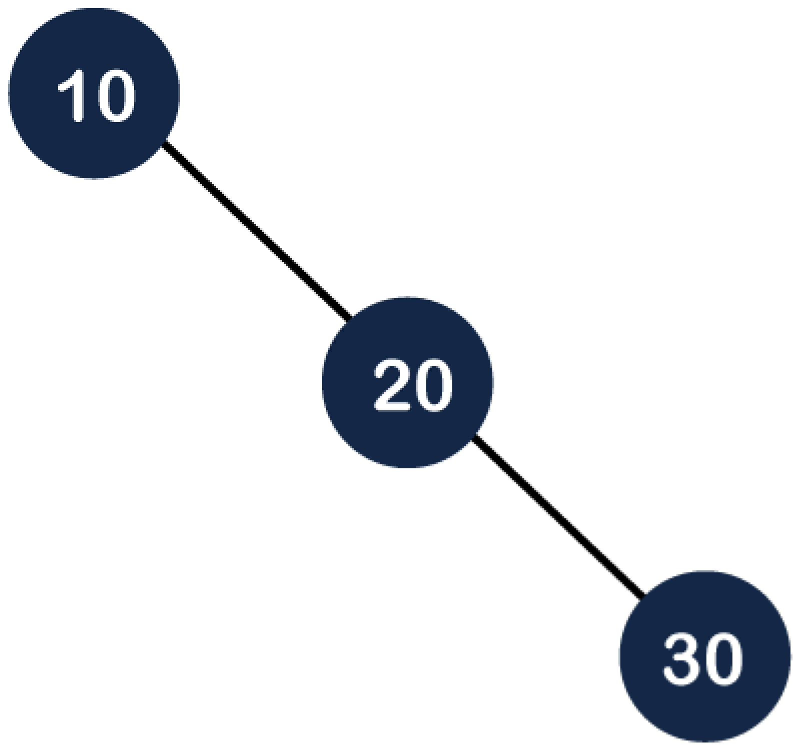


As we can observe, the final tree follows the property of the Binary Search tree and a balanced tree; therefore, it is an AVL tree.

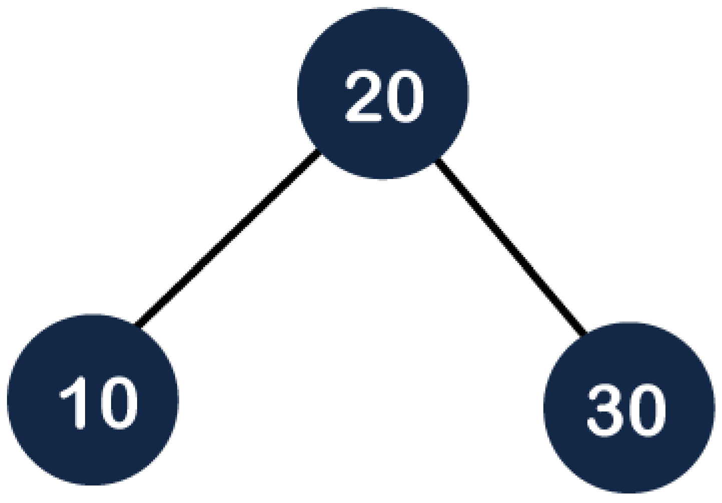
In the case, the tree was ***left unbalanced tree,*** so we perform the right rotation on the node.

- **If the order of insertion is 10, 20, 30.**

Step 1: First we create a Binary search tree as shown below:



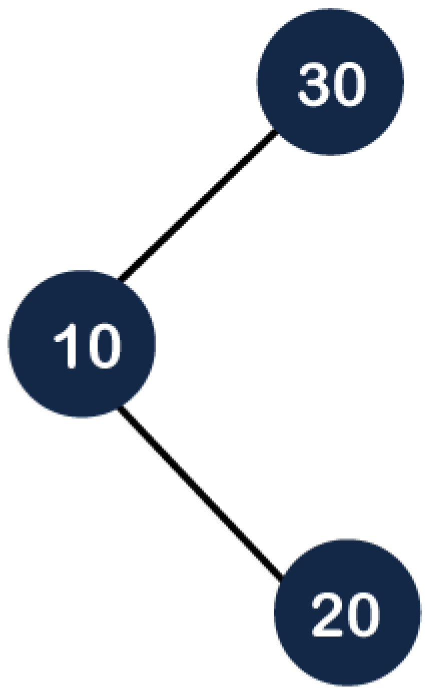
Step 2: In the above figure, we can observe that the tree is unbalanced because the balance factor of node 10 is -2. In order to make it an AVL tree, we need to perform some rotations. It is a right unbalanced tree, so we will perform left rotation. If we perform left rotation on node 20, then the node 20 will move upwards, and node 10 will move downwards, as shown below:



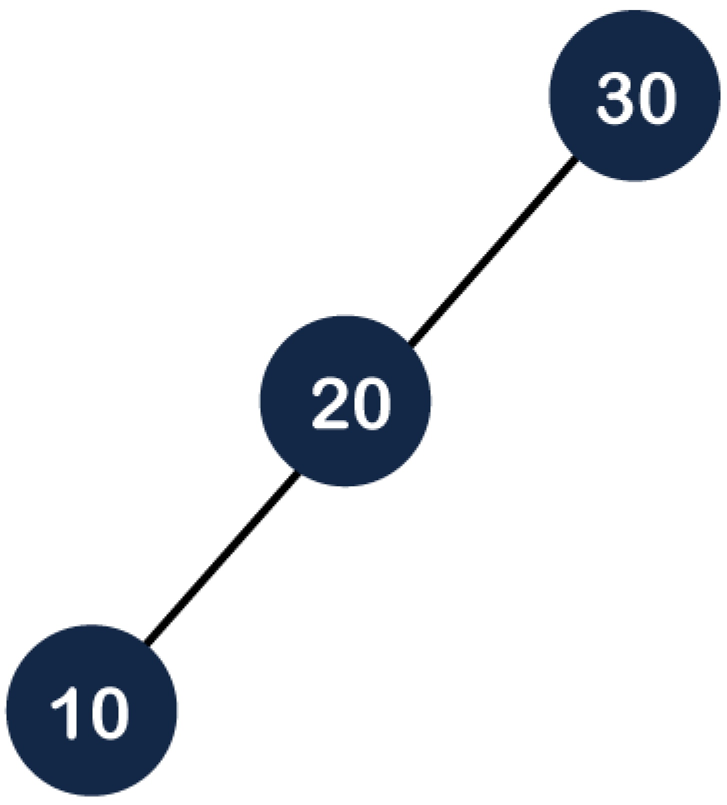
As we can observe, the final tree follows the property of the ***Binary Search tree*** and a ***balanced tree***; therefore, it is an AVL tree.

- **If the order of insertion is 30, 10, 20**

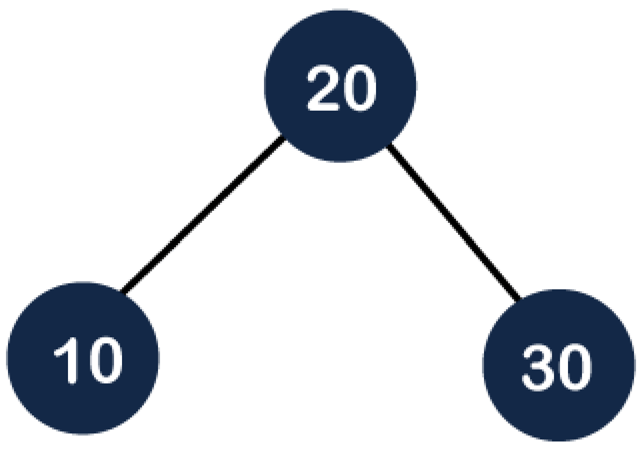
Step 1: First we create the Binary Search tree as shown below:



Step 2: In the above figure, we can observe that the tree is unbalanced because the balance factor of the root node is 2. In order to make it an AVL tree, we need to perform some rotations. The above scenario is left-right unbalanced as one node is the left of its parent node and another is the right of its parent node. First, we will perform the left rotation, and rotation happens between nodes 10 and 20. After left rotation, 20 will move upwards, and 10 will move downwards as shown below:



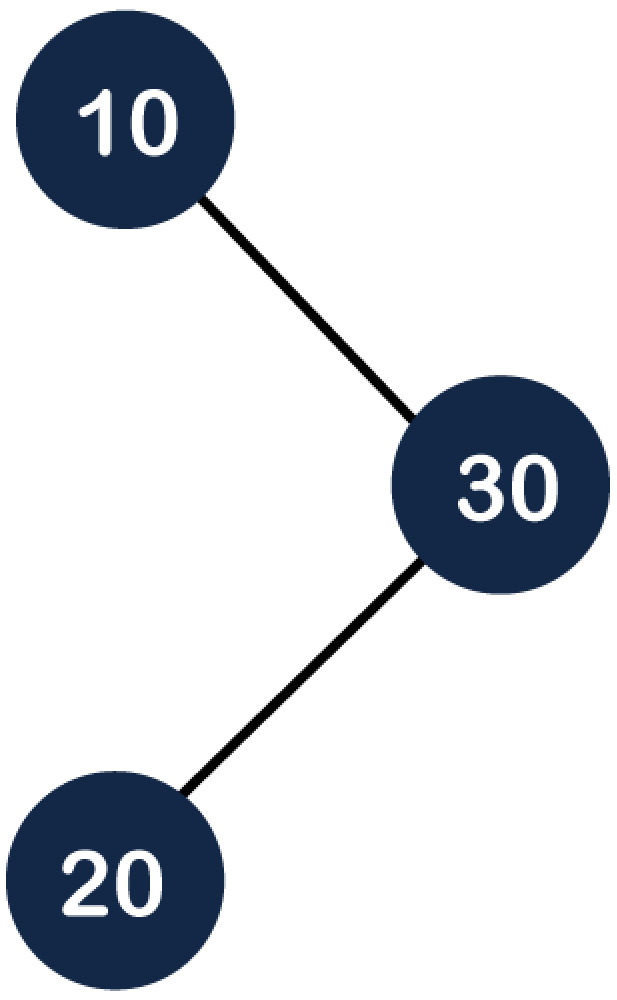
Still, the tree is unbalanced, so we perform the right rotation on the tree. Once the right rotation is performed on the tree, then the tree would like, as shown below:



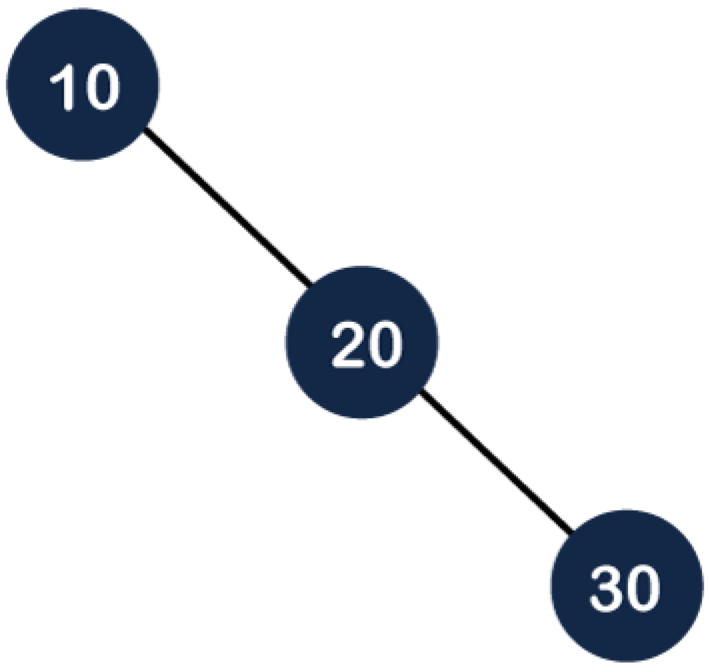
As we can observe in the above tree, the tree follows the property of the Binary Search tree and a balanced tree; therefore, it is an AVL tree.

- **If the order of the insertion is 10, 30, 20**

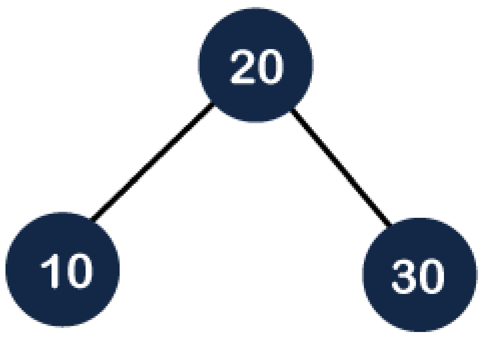
Step 1: First, we create the Binary Search tree, as shown below:



Step 2: In the above figure, we can observe that tree is unbalanced because the balance factor of the root node is 2. In order to make it an AVL tree, we need to perform some rotations. The above scenario is right-left unbalanced as one node is right of its parent node, and another node is left of its parent node. First, we will perform the right rotation that happens between nodes 30 and 20. After right rotation, 20 will move upwards, and 30 will move downwards as shown below:



Still, the above tree is unbalanced, so we need to perform left rotation on the node. Once the left rotation is performed, the node 20 will move upwards, and node 10 will move downwards as shown below:



As we can observe in the above tree, the tree follows the property of the Binary Search tree and a balanced tree; therefore, it is an AVL tree.

Differences between Binary Search tree and AVL tree

Binary Search tree	AVL tree
Every binary search tree is a binary tree because both the trees contain the utmost two children.	Every AVL tree is also a binary tree because AVL tree also has the utmost two children.
In BST, there is no term exists, such as balance factor.	In the AVL tree, each node contains a balance factor, and the value of the balance factor must be either -1, 0, or 1.

Every Binary Search tree is not an AVL tree because BST could be either a balanced or an unbalanced tree.	Every AVL tree is a binary search tree because the AVL tree follows the property of the BST.
Each node in the Binary Search tree consists of three fields, i.e., left subtree, node value, and the right subtree.	Each node in the AVL tree consists of four fields, i.e., left subtree, node value, right subtree, and the balance factor.
In the case of Binary Search tree, if we want to insert any node in the tree then we compare the node value with the root value; if the value of node is greater than the root node value then the node is inserted to the right subtree otherwise the node is inserted to the left subtree. Once the node is inserted, there is no need of checking the height balance factor for the insertion to be completed.	In the case of AVL tree, first, we will find the suitable place to insert the node. Once the node is inserted, we will calculate the balance factor of each node. If the balance factor of each node is satisfied, the insertion is completed. If the balance factor is greater than 1, then we need to perform some rotations to balance the tree.
In Binary Search tree, the height or depth of the tree is O(n) where n is the number of nodes in the Binary Search tree.	In AVL tree, the height or depth of the tree is O(logn).
It is simple to implement as we have to follow the Binary Search properties to insert the node.	It is complex to implement because in AVL tree, we have to first construct the AVL tree, and then we need to check height balance. If the height is imbalance then we need to perform some rotations to balance the tree.
BST is not a balanced tree because it does not follow the concept of the balance factor.	AVL tree is a height balanced tree because it follows the concept of the balance factor.
Searching is inefficient in BST when there are large number of nodes available in the tree because the height is not balanced.	Searching is efficient in AVL tree even when there are large number of nodes in the tree because the height is balanced.

Red Black Tree vs AVL tree

Before understanding the ***Red-Black tree and AVL tree*** differences, we should know about the Red-Black tree and AVL tree separately.

What is a Red-Black tree?

The Red-black tree is a self-balanced binary search tree in which each node contains one extra bit of information that denotes the color of the node. The color of the node could be either Red or Black, depending on the bit information stored by the node.

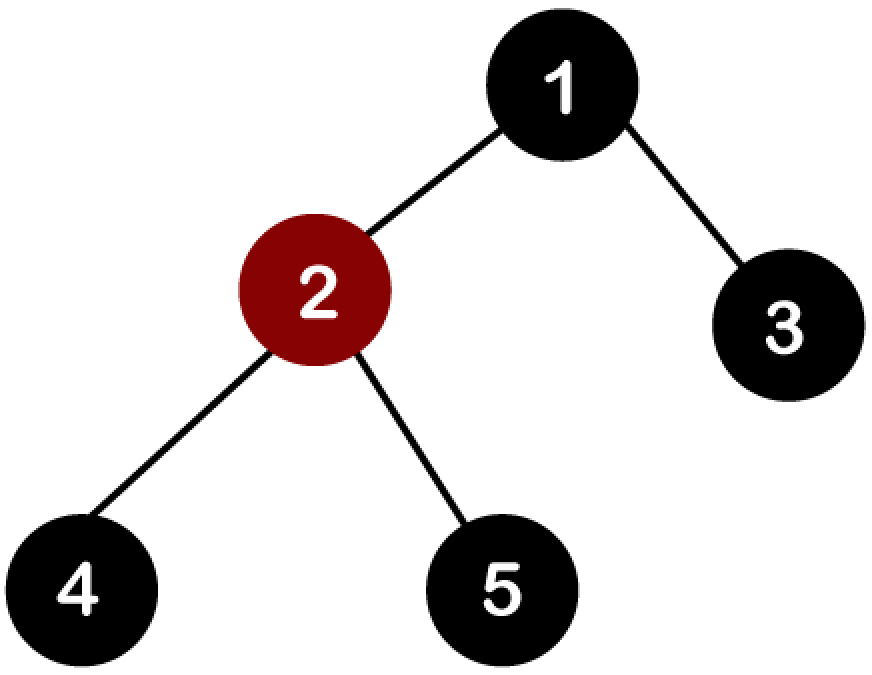
Properties

The following are the properties associated with a Red-Black tree:

- The root node of the tree should be Black.
- A red node can have only black children. Or, we can say that two adjacent nodes cannot be Red in color.
- If the node does not have a left or right child, then that node is said to be a leaf node. So, we put the left and right children below the leaf node known as **nil**

The black depth or black height of a leaf node can be defined as the number of black that we encounter when we move from the leaf node to the root node. One of the key properties of the Red-Black tree is that every leaf node should have the same black depth.

Let's understand this property through an example.



In the above tree, there are five nodes, in which one is a Red and the other four nodes are Black. There are three leaf nodes in the above tree. Now we calculate the black depth of each leaf node. As we can observe that the black depth of all the three leaf nodes is 2; therefore, it is a Red-Black tree.

If the tree does not obey any of the above three properties, then it is not a Red-Black tree.

Height of a red-black tree

Consider h as the height of the tree having n nodes. What would be the smallest value of n?. The value of n is the smallest when all the nodes are black. If the tree contains all the black nodes, then the tree would be a complete binary tree. If the height of a complete binary tree is h, then the number of nodes in a tree is:

n = 2h -1

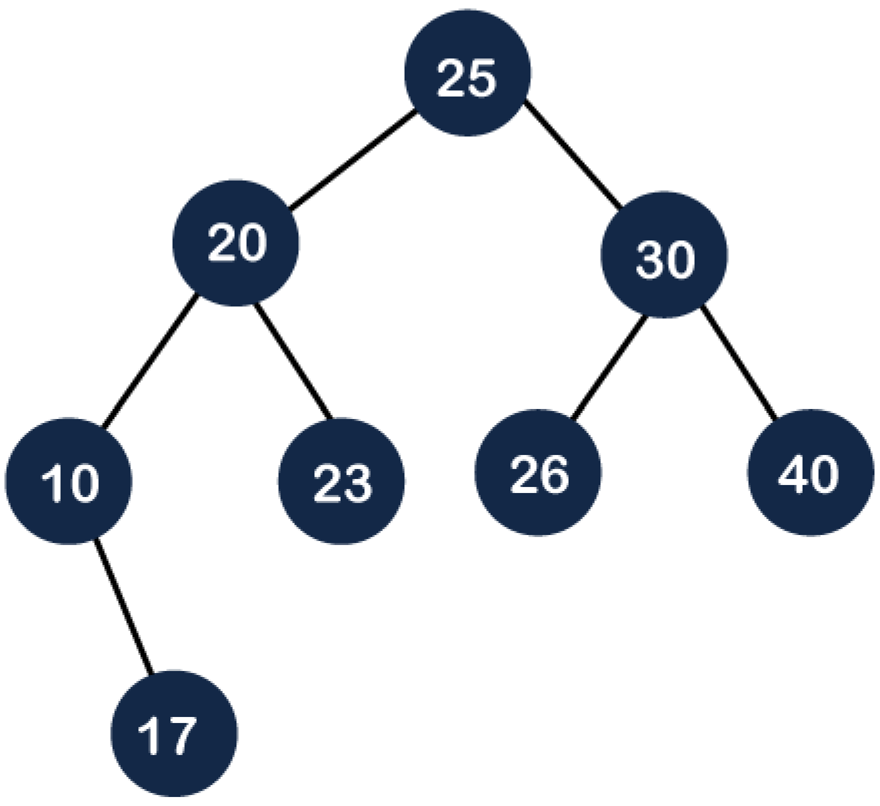
What would be the largest value of n?

The value of n is largest when every black node has two red children, but the red node cannot have red children, so that it will have black children. In this way, there are alternate layers of black and red children. So, if the number of black layers is h, then the number of red layers is also h; therefore, the tree's total height becomes 2h. The total number of nodes is:

n = 2*2h-1

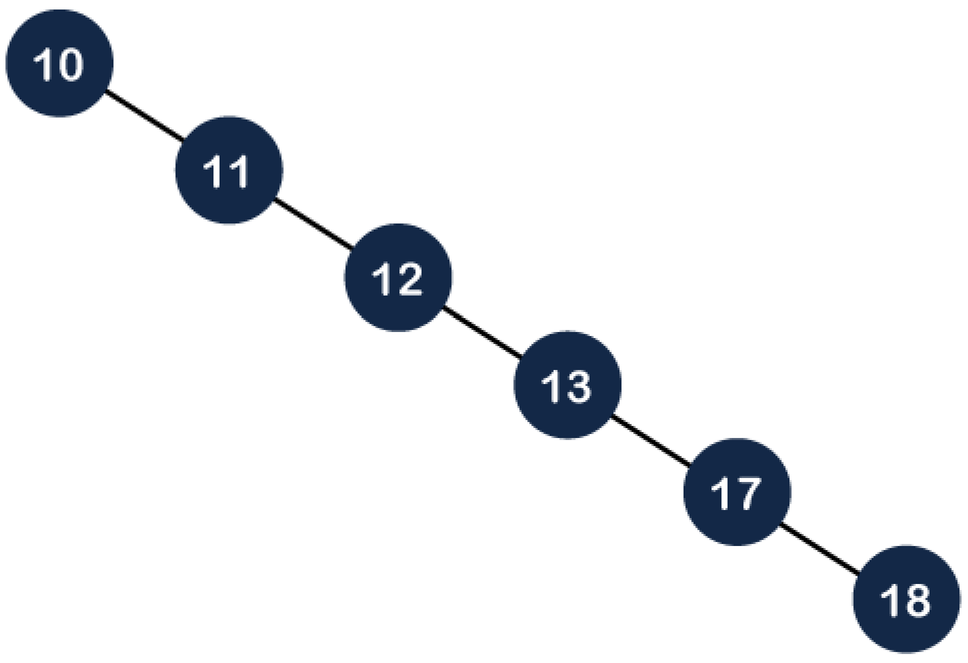
What is the AVL tree?

An AVL tree is a self-balancing binary search tree if we observe the below case, which is a binary search tree and a balanced tree.



In the above tree, the worst-case time complexity for searching an element is O(h), i.e., the height of the tree. In the above case, the number of comparisons made to search 17 element is 4, and the height of the tree is also 4.

If we consider the skewed binary tree, as shown below:



In the above right skewed tree, the number of comparisons made to search 17 element is 5, and the number of elements present in the tree is also 5. Therefore, we can say that if the tree is a skewed binary tree then the time complexity would be O(n).

Now, we have to balance these skewed trees so that they will have the time complexity O(h). There is a term known as a ***balance factor***, which is used to self -balance the binary search tree. The balance factor can be computed as:

```
Balance factor = height of left subtree-height of right subtree
```

The value of the balance factor could be either 1, 0 or -1. If each node in the binary tree is having a value of either 1, 0, or -1, then that tree is said to be a balanced binary tree or AVL tree.

The tree is known as a perfectly balanced tree if the balance factor of each node is 0. The AVL tree is an almost balanced tree because each node in the tree has the value of balance factor either 1, 0 or -1.

Differences between the Red-Black tree and AVL tree.

The following are the differences between the Red-Black tree and AVL tree:

- **Binary search tree**

The Red-Black tree is a binary search tree, and the AVL tree is also a binary search tree.

- **Rules**

The following rules are applied in a Red-Black Tree:

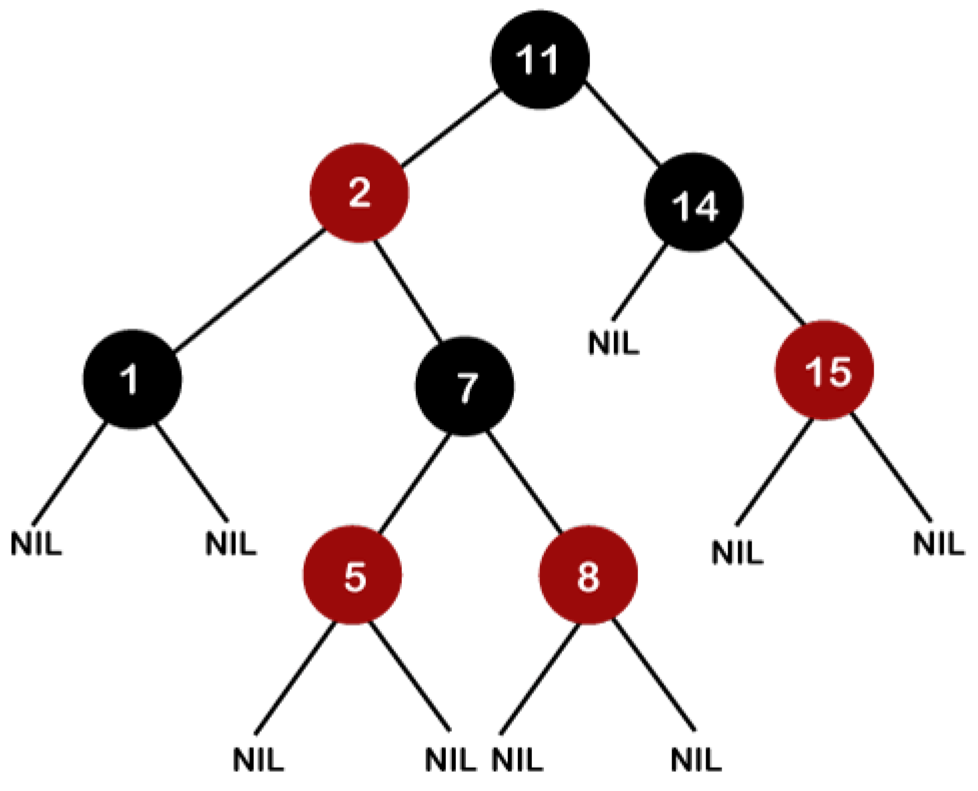
1. The node in a Red-Black tree is either red or black in color.
2. The color of the root node should be black.
3. The adjacent nodes should not be red. In other words, we can say that the red node cannot have red children, but the black node can have black children.
4. There should be the same number of black nodes in every path; then, only a tree can be considered a Red-Black tree.

5. The external nodes are the nil nodes, which are always black in color.

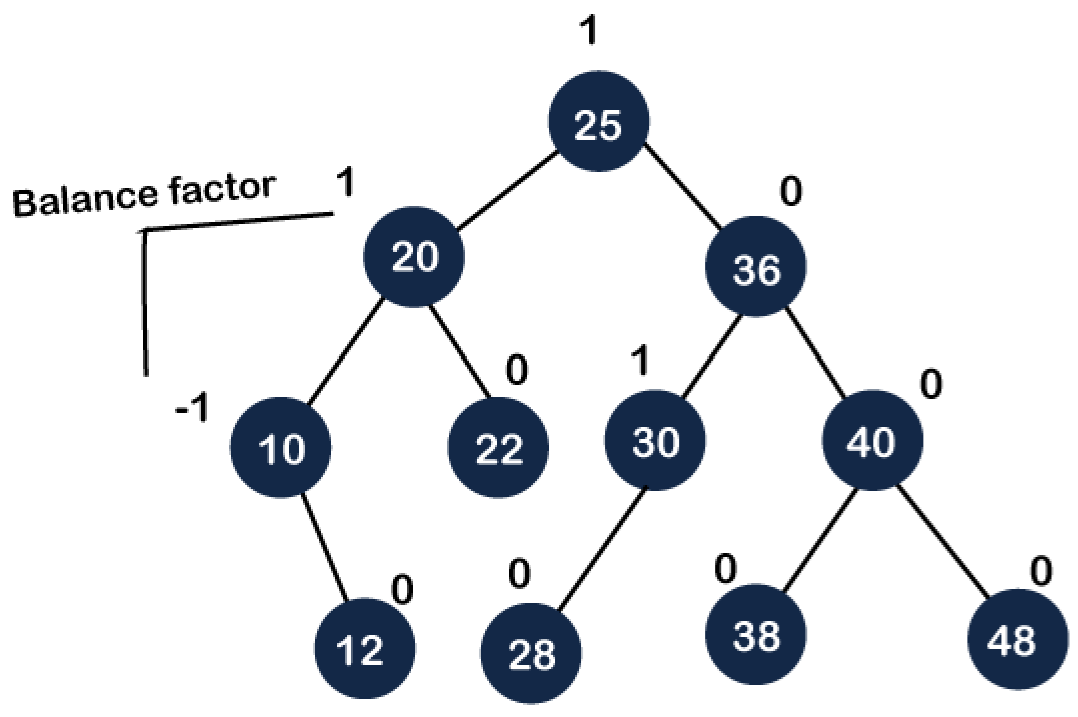
Rule of the AVL tree:

Every node should have the balance factor either as -1, 0 or 1.

- **Example**



In the above figure, we need to check whether the tree is a Red-Black tree or not. In order to check this, first, we need to check whether the tree is a binary search tree or not. As we can observe in the above figure that it satisfies all the properties of the binary search tree; therefore, it is a binary search tree. Secondly, we have to verify whether it satisfies the above-said rules or not. The above tree satisfies all the above five rules; therefore, it concludes that the above tree is a Red-Black tree.



In the above figure, we need to check whether the tree is an AVL tree or not. As each node has a value of balance factor either as -1, 0, or 1, so it is an AVL tree.

- **How can the tree be considered as a balanced tree or not?**

In the case of a Red-Black tree, if all the above rules are satisfied, provided that a tree is a binary search tree, then the tree is said to be a Red-black tree.

In the case of the AVL tree, if the balance factor is -1, 0, or 1, then the above tree is said to be an AVL tree.

- **Tools used for balancing**

If the tree is not balanced, then two tools are used for balancing the tree in a Red-Black tree:

1. **Recoloring**

2. **Rotation**

If the tree is not balanced, then one tool is used for balancing the tree in the AVL tree:

1. **Rotation**

- **Efficient for which operation**

In the case of the Red-Black tree, the insertion and deletion operations are efficient. If the tree gets balanced through the recoloring, then insertion and deletion operations are efficient in the Red-Black tree.

In the case of the AVL tree, the searching operation is efficient as it requires only one tool to balance the tree.

- **Time complexity**

In the Red-Black tree case, the time complexity for all the operations, i.e., insertion, deletion, and searching is O(logn).

In the case of AVL tree, the time complexity for all the operations, i.e., insertion, deletion, and searching is O(logn).

Let's understand the differences in the tabular form.

Parameter	Red Black Tree	AVL Tree
Searching	Red Black tree does not provide efficient searching as Red Black Trees are roughly balanced.	AVL trees provide efficient searching as it is strictly balanced tree.
Insertion and Deletion	Insertion and Deletion are easier in Red Black tree as it requires fewer rotations to balance the tree.	Insertion and Deletion are complex in AVL tree as it requires multiple rotations to balance the tree.
Color of the node	In the Red-Black tree, the color of the node is either Red or Black.	In the case of AVL trees, there is no color of the node.
Balance factor	It does not contain any balance factor. It stores only one bit of information that denotes either Red or Black color of the node.	Each node has a balance factor in AVL tree whose value can be 1, 0, or -1. It requires extra space to store the balance factor per node.
Strictly balanced	Red-black trees are not strictly balanced.	AVL trees are strictly balanced, i.e., the left subtree's height and the height of the right subtree differ by at most 1.

B tree vs B+ tree

Before understanding **B tree** and **B+ tree** differences, we should know the B tree and B+ tree separately.

What is the B tree?

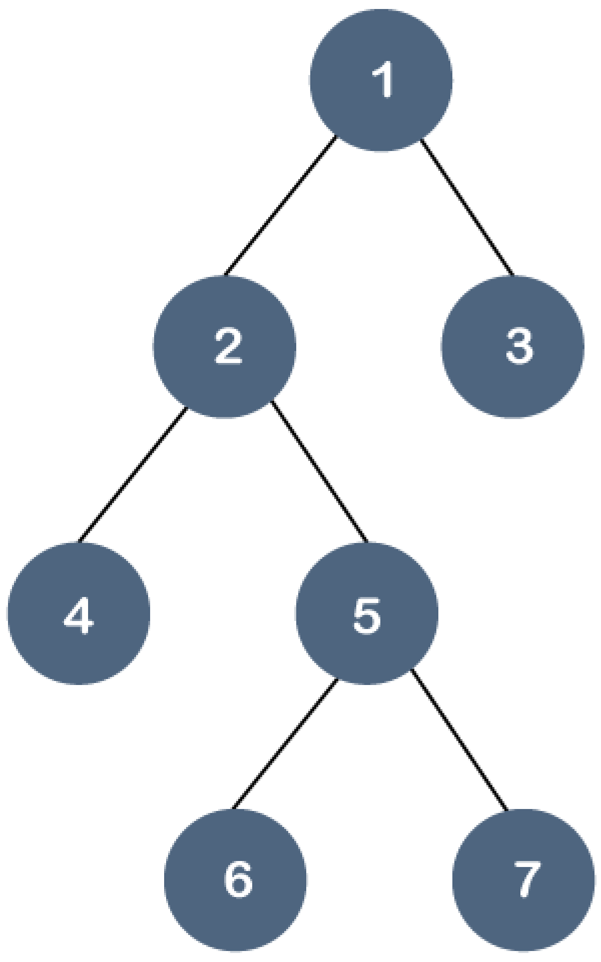
B tree is a self-balancing tree, and it is a m-way tree where m defines the order of the tree. **Btree** is a generalization of the Binary Search tree in which a node can have more than one key and more than two children depending upon the value of **m**. In the B tree, the data is specified in a sorted order having lower values on the left subtree and higher values in the right subtree.

Properties of B tree

The following are the properties of the B tree:

- In the B tree, all the leaf nodes must be at the same level, whereas, in the case of a binary tree, the leaf nodes can be at different levels.

Let's understand this property through an example.



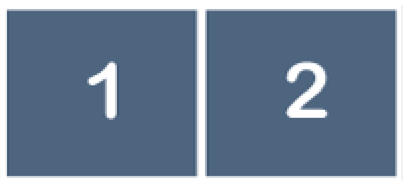
In the above tree, all the leaf nodes are not at the same level, but they have the utmost two children. Therefore, we can say that the above tree is a binary tree but not a B tree.

- If the Btree has an order of m, then each node can have a maximum of **m** In the case of minimum children, the leaf nodes have zero children, the root node has two children, and the internal nodes have a ceiling of m/2.
- Each node can have maximum (m-1) keys. For example, if the value of m is 5 then the maximum value of keys is 4.
- The root node has minimum one key, whereas all the other nodes except the root node have (ceiling of m/2 minus - 1) minimum keys.
- If we perform insertion in the B tree, then the node is always inserted in the leaf node.

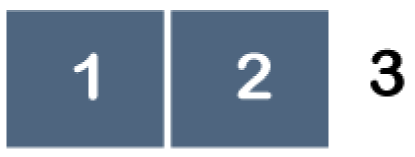
Suppose we want to create a B tree of order 3 by inserting values from 1 to 10.

Step 1: First, we create a node with 1 value as shown below:

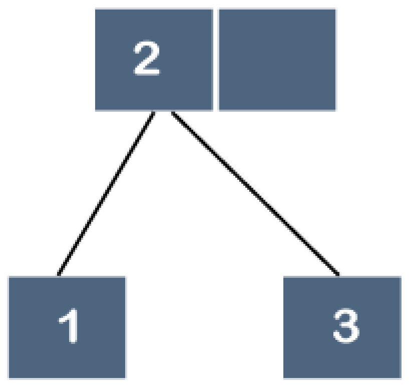
Step 2: The next element is 2, which comes after 1 as shown below:



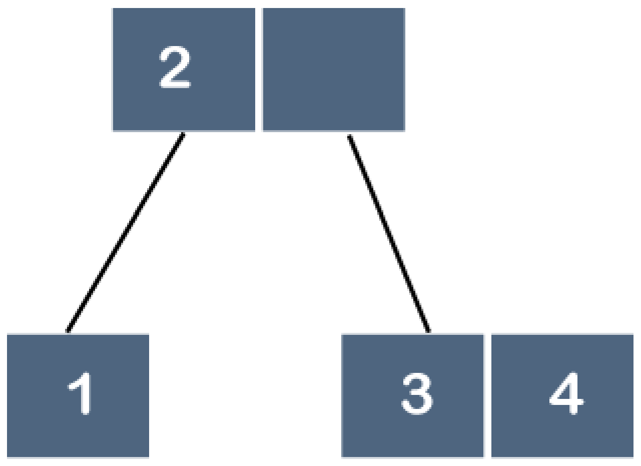
Step 3: The next element is 3, and it is inserted after 2 as shown below:



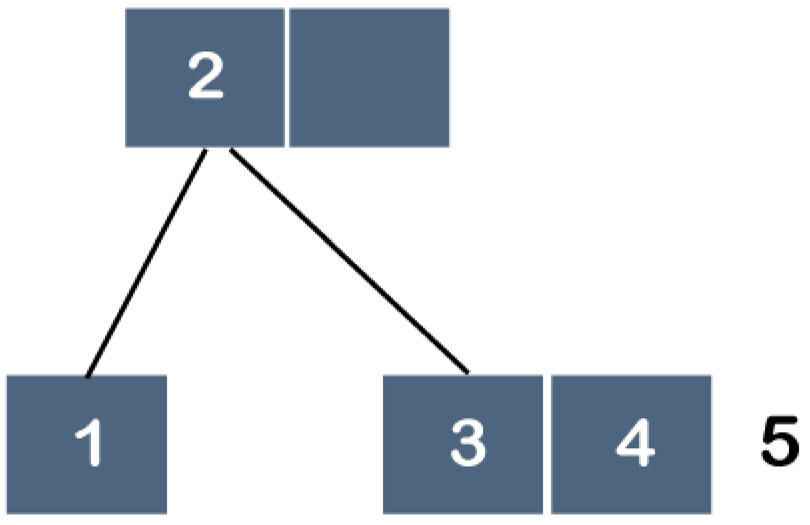
As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 2, so it moves to its parent. The node 2 does not have any parent, so it will become the root node as shown below:



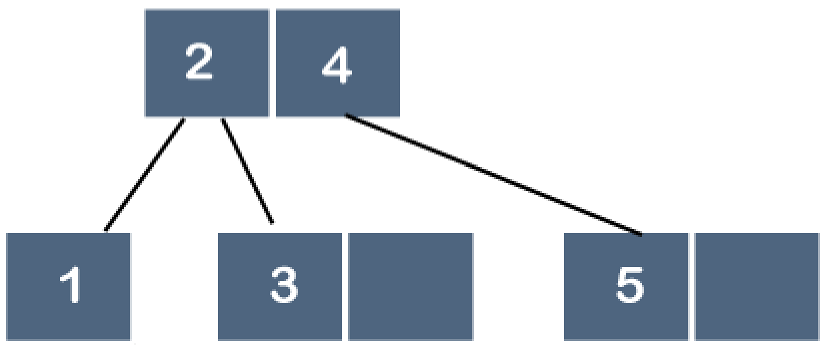
Step 4: The next element is 4. Since 4 is greater than 2 and 3, so it will be added after the 3 as shown below:



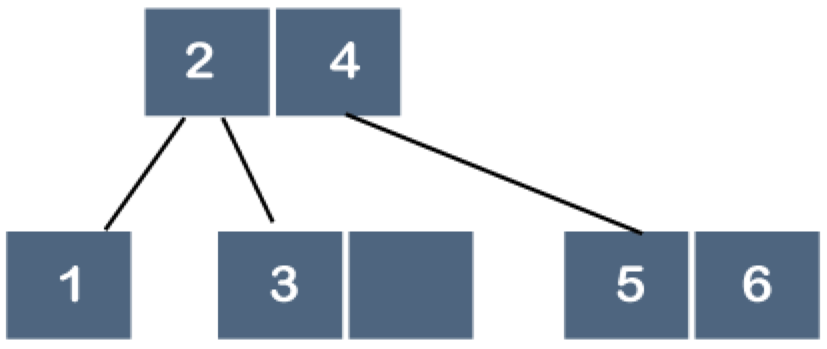
Step 5: The next element is 5. Since 5 is greater than 2, 3 and 4 so it will be added after 4 as shown below:



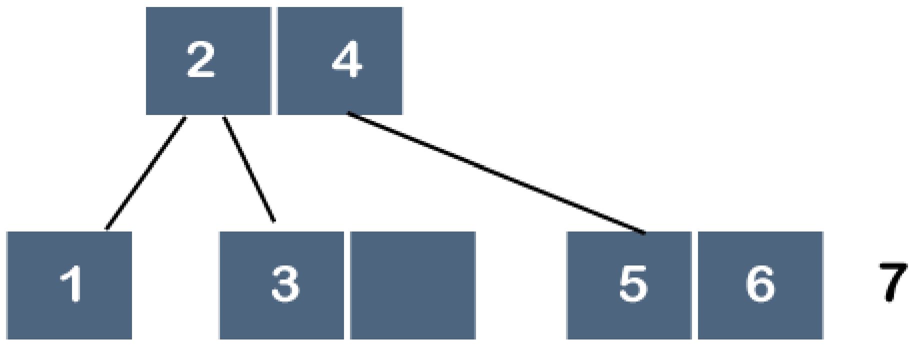
As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 4, so it moves to its parent. The parent is node 2; therefore, 4 will be added after 2 as shown below:



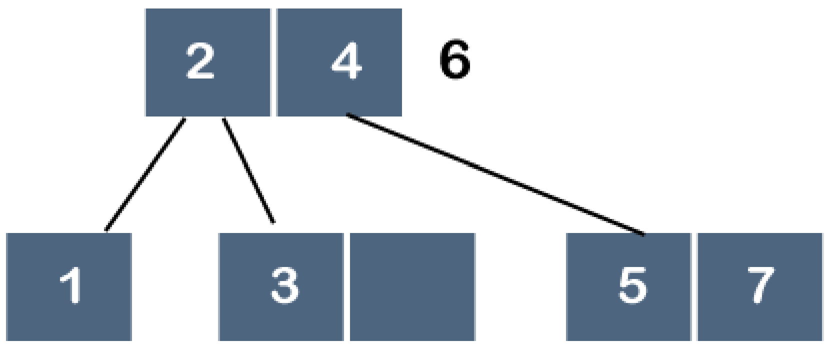
Step 6: The next element is 6. Since 6 is greater than 2, 4 and 5, so 6 will come after 5 as shown below:



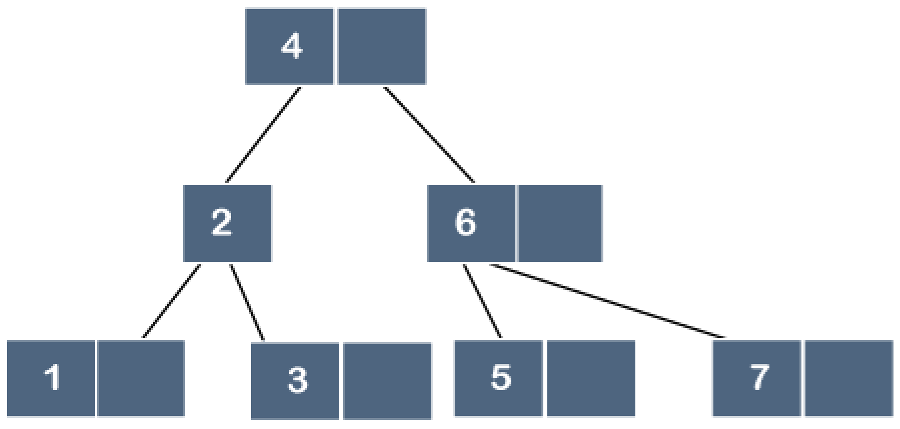
Step 7: The next element is 7. Since 7 is greater than 2, 4, 5 and 6, so 7 will come after 6 as shown below:



As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 6, so it moves to its parent as shown below:



But, 6 cannot be added after 4 because the node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 4, so it moves to its parent. As node 4 does not have any parent, node 4 will become a root node as shown below:



What is a B+ tree?

The B+ tree is also known as an advanced self-balanced tree because every path from the root of the tree to the leaf of the tree has the same length. Here, the same length means that all the leaf nodes occur at the same level. It will not happen that some of the leaf nodes occur at the third level and some of them at the second level.

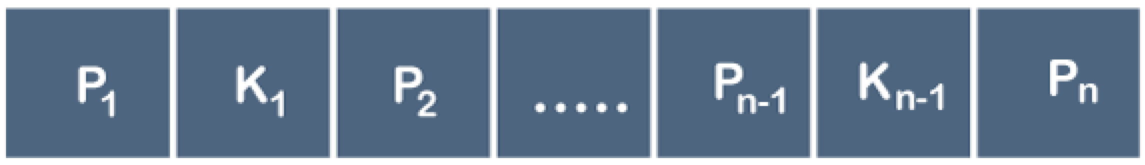
A B+ tree index is considered a multi-level index, but the B+ tree structure is not similar to the multi-level index sequential files.

Why is the B+ tree used?

A B+ tree is used to store the records very efficiently by storing the records in an indexed manner using the B+ tree indexed structure. Due to the multi-level indexing, the data accessing becomes faster and easier.

B+ tree Node Structure

The node structure of the B+ tree contains pointers and key values shown in the below figure:



As we can observe in the above B+ tree node structure that it contains n-1 key values (k_1 to k_{n-1}) and n pointers (p_1 to p_n).

The search key values which are placed in the node are kept in sorted order. Thus, if i<j then $k_i<k_j$.

Constraint on various types of nodes

Let 'b' be the order of the B+ tree.

Non-Leaf node

Let 'm' represents the number of children of a node, then the relation between the order of the tree and the number of children can be represented as:

$$\left\lceil \frac{b}{2} \right\rceil \leq m \leq b$$

Let k represents the search key values. The relation between the order of the tree and search key can be represented as:

As we know that the number of pointers is equal to the search key values plus 1, so mathematically, it can be written as:

Number of Pointers (or children) = Number of Search keys + 1

Therefore, the maximum number of pointers would be 'b', and the minimum number of pointers would be the ceiling function of b/2.

Leaf Node

A leaf node is a node that occurs at the last level of the B+ tree, and each leaf node uses only one pointer to connect with each other to provide the sequential access at the leaf level.

In leaf node, the maximum number of children is:

$$\left\lceil \frac{b}{2} \right\rceil -1 \leq k \leq b-1$$

The maximum number of search keys is:

$$\left\lceil \frac{b}{2} \right\rceil -1 \leq m \leq b-1$$

Root Node

The maximum number of children in the case of the root node is: b

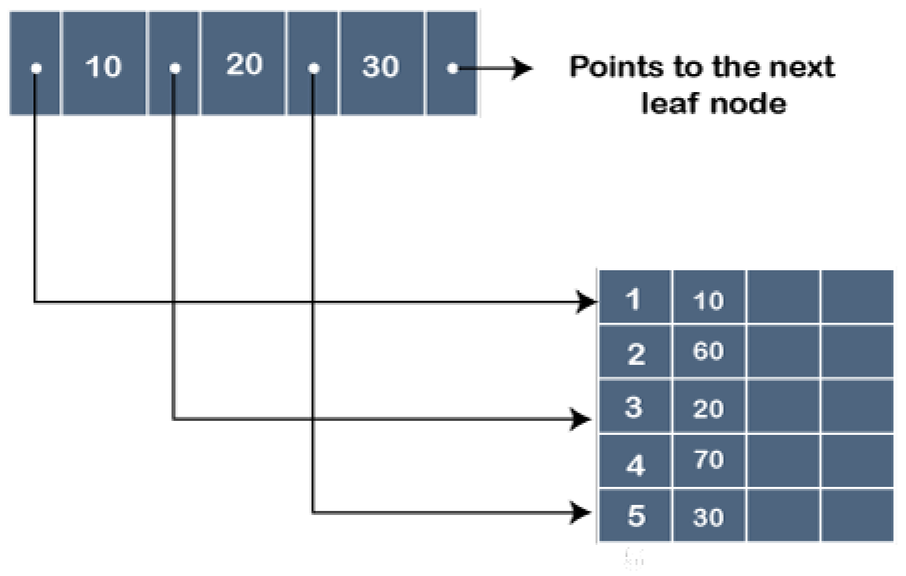
The minimum number of children is: 2

Special cases in B+ tree

Case 1: If the root node is the only node in the tree. In this case, the root node becomes the leaf node.

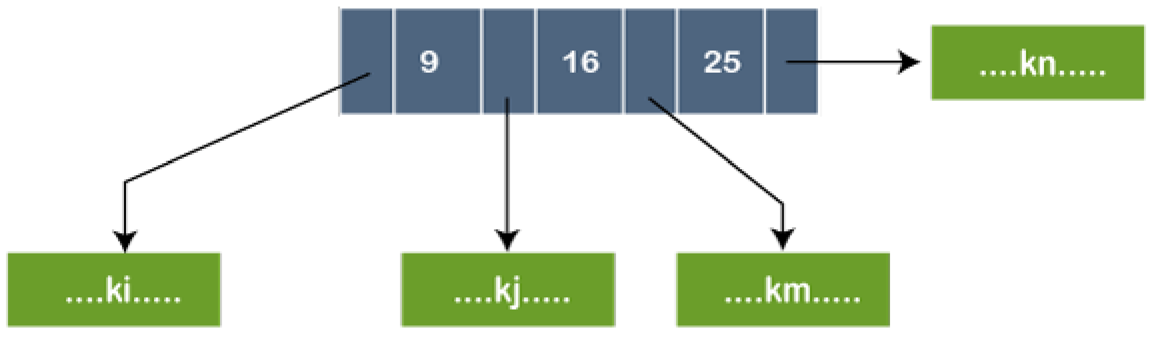
In this case, the maximum number of children is 1, i.e., the root node itself, whereas, the minimum number of children is b-1, which is the same as that of a leaf node.

Representation of a leaf node in B+ tree



In the above figure, '.' represents the pointer, whereas the 10, 20 and 30 are the key values. The pointer contains the address at which the key value is stored, as shown in the above figure.

Example of B+ tree



In the above figure, the node contains three key values, i.e., 9, 16, and 25. The pointer that appears before 9, contains the key values less than 9 represented by k_i. The pointer that appears before 16, contains the key values greater than or equal to 9 but less than 16 represented by kj. The pointer that appears before 25, contains the key values greater than or equal to 16 but less than 25 represented by k_n.

Differences between B tree and B+ tree

The following are the differences between the B tree and B+ tree:

B tree	B+ tree
In the B tree, all the keys and records are stored in both internal as well as leaf nodes.	In the B+ tree, keys are the indexes stored in the internal nodes and records are stored in the leaf nodes.
In B tree, keys cannot be repeatedly stored, which means that there is no duplication of keys or records.	In the B+ tree, there can be redundancy in the occurrence of the keys. In this case, the records are stored in the leaf nodes, whereas the keys are stored in the internal nodes, so redundant keys can be present in the internal nodes.

In the Btree, leaf nodes are not linked to each other.	In B+ tree, the leaf nodes are linked to each other to provide the sequential access.
In Btree, searching is not very efficient because the records are either stored in leaf or internal nodes.	In B+ tree, searching is very efficient or quicker because all the records are stored in the leaf nodes.
Deletion of internal nodes is very slow and a time-consuming process as we need to consider the child of the deleted key also.	Deletion in B+ tree is very fast because all the records are stored in the leaf nodes so we do not have to consider the child of the node.
In Btree, sequential access is not possible.	In the B+ tree, all the leaf nodes are connected to each other through a pointer, so sequential access is possible.
In Btree, the more number of splitting operations are performed due to which height increases compared to width,	B+ tree has more width as compared to height.
In Btree, each node has atleast two branches and each node contains some records, so we do not need to traverse till the leaf nodes to get the data.	In B+ tree, internal nodes contain only pointers and leaf nodes contain records. All the leaf nodes are at the same level, so we need to traverse till the leaf nodes to get the data.
The root node contains atleast 2 to m children where m is the order of the tree.	The root node contains atleast 2 to m children where m is the order of the tree.

Difference Between Quick Sort and Merge Sort

A **sorting** is the arrangement of collectively data in particular format like ascending or descending order. Generally, it is used to arrange the homogeneous data in sorted manner. Using the **sorting** algorithms, we can arrange the data in a sequence order and search an element easily and faster. **Sorting techniques** depends on two situation such as total time and total space required to execute a program. In this section, we will discuss **quick sort** and **merge sort** and also compare them each other.

Quick Sort

Quick sort is a comparison based sorting algorithm that follows the divide and conquer technique to sort the arrays. In quick sort, we usually use a **pivot (key)** element to compare and interchange the position of the element based on some condition. When a pivot element gets its fixed position in the array that shows the termination of comparison & interchange procedure. After that, the array divides into the two sub arrays. Where the first partition contains all those elements that are less than pivot (key) element and the other parts contains all those elements that are greater than pivot element. After that, it again selects a pivot element on each of the sub arrays and repeats the same process until all the elements in the arrays are sorted into an array.

Algorithm of Quick Sort

Partition (A, p, r)

1. X <- A[r]

```
2. I <- p-1
3. For j <- p to r -1
4. Do if A[j] <= x
5. Then I <- I + 1
6. Exchange A[i] <-> A[j]
7. Exchange A[I + 1] <--> A[r]
8. Return I + 1
```

Quicksort (A, p, r)

```
1. While (p < r)
2. Do q <- Partition (A, p, r)
3. R <- q-1
4. While (p < r)
5. Do q <- Partition (A, p, r)
6. P <- q + 1
```

Steps to sort an array using the quick sort algorithm

Suppose, we have an array X having the elements X[1], X[2], X[3],...., X[n] that are to be sort. Let's follow the below steps to sort an array using the quick sort.

Step 1: Set the first element of the array as the **pivot** or key element. Here, we assume pivot as X[0], **left** pointer is placed at the first element and the **last** index of the array element as **right**.

Step 2: Now we starts the scanning of the array elements from right side index, then

If X[key] is less than X[right] or if X[key] < X[Right],

1. Continuously decreases the right end pointer variable until it becomes equal to the key.
2. If X[key] > X[right], interchange the position of the key element to the X[right] element.
3. Set, key = right and increment the left index by 1.

Step 3: Now we again start the scanning of the element from left side and compare each element with the key element. X[key] > X[left] or X[key] is greater than X[left], then it performs the following actions:

1. Continuously compare the left element with the X[key] and increment the left index by 1 until key becomes equal to the left.
2. If X[key] < X[left], interchange the position of the X[key] with X[left] and go to step 2.

Step 4: Repeat Step 2 and 3 until the X[left] becomes equal to X[key]. So, we can say that if X[left] = X[key], it shows the termination of the procedures.

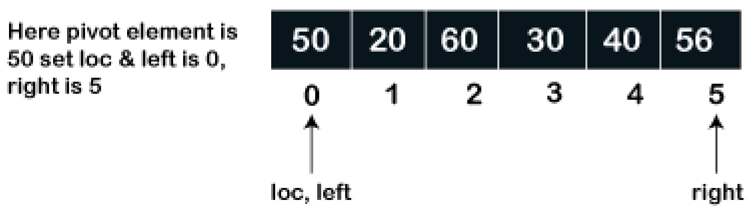
Step 5: After that, all the elements at the left side will be smaller than the key element and the rest element of the right side will be larger than the key element. Thus indicating the array needs to partitioned into two sub arrays.

Step 6: Similarly, we need to repeatedly follow the above procedure to the sub arrays until the entire array becomes sorted.

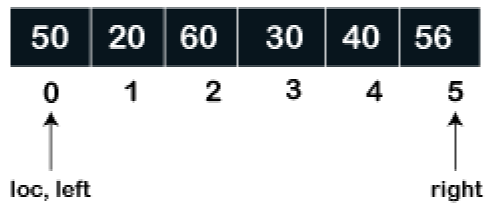
Let's see an example of quick sort.

Example: Consider an array of 6 elements. Sort the array using the quick sort.

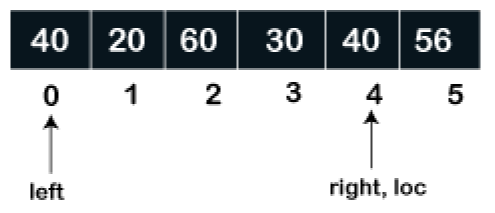
arr[] = {50, 20, 60, 30, 40, 56}



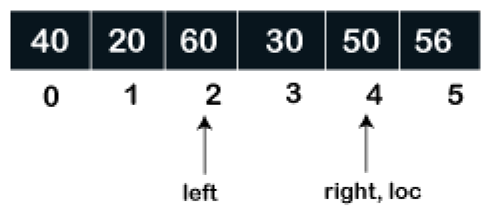
Scanning of the element from right and decrease the variable index,if a[loc] <a[right], we get



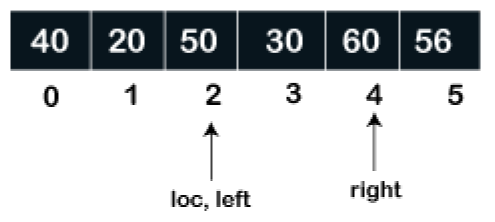
Here, a[loc]> aright, interchange the elements with pivot:



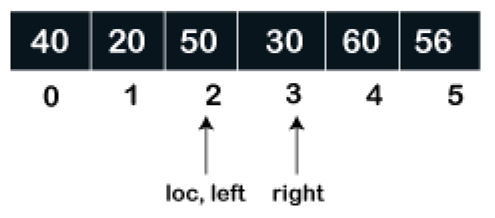
Start scanning of the elements from left and increment the left pointer, until a[loc]> a[left]:



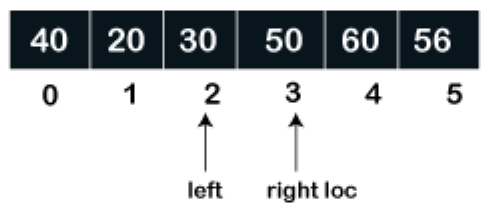
When a[loc] <a[left], interchange the element:



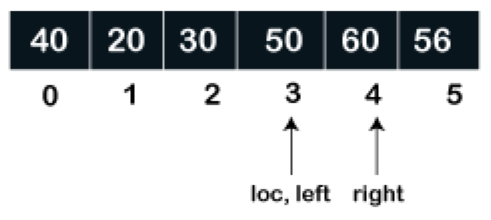
Start scanning of the element from right side:



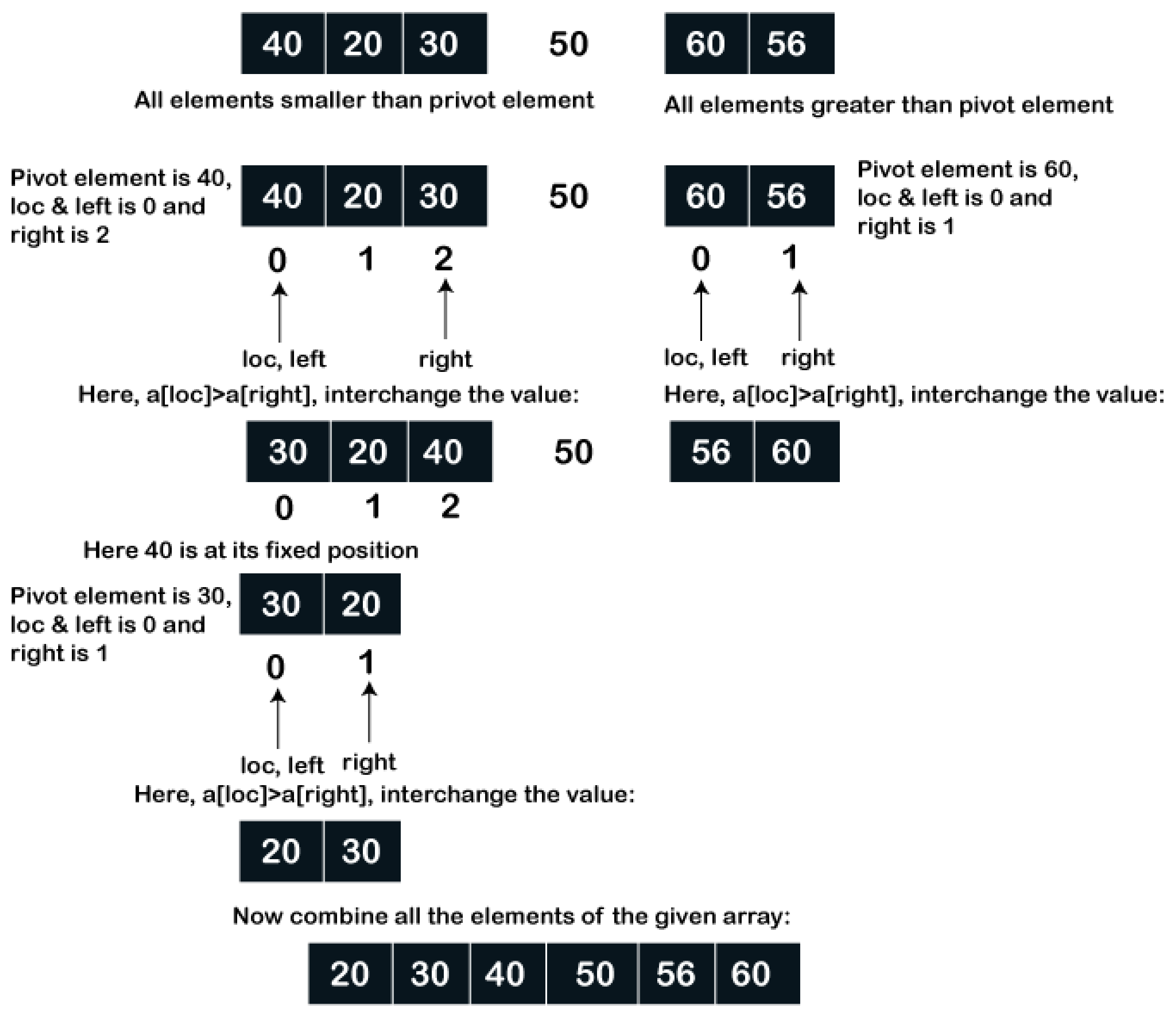
Here, a[loc]> a[right], interchage the element:



Now start Scanning of the element from left until a[loc] > a[left] then increment the left pointer:



In the above array, 50 is in its right place. So, we divided the elements that are less than pivot in one sub array and the elements that are larger than the pivot element in another sub array.



40 20 30 50 60 56
All elements smaller than privot element
All elements greater than pivot element
Pivot element is 40, loc & left is 0 and right is 2
Pivot element is 60, loc & left is 0 and right is 1
loc, left
right
Here, a[loc]>a[right], interchange the value:
30 20 40 50
56 60
Here 40 is at its fixed position
Pivot element is 30, loc & left is 0 and right is 1
Here, a[loc]>a[right], interchange the value:
20 30
Now combine all the elements of the given array:
20 30 40 50 56 60

Hence, we get the sorted array.

Let's implement the above logic in C program.

Quick.c

```c
#include <stdio.h>
int division_of_array(int x[], int st, int lt); // declaration of functions
void quick_sort(int x[], int st, int lt);
void main()
{
  int i;
   int ar[6] = {50, 20, 60, 30, 40, 56}; // given array elements
  quick_sort(ar, 0, 5);  // it contains array, starting index and last index as an parameters.
   print(" Sorted Array is : \n");
  for(i=0; i<6; i++)
  {
     printf("Array = %d\t", ar[i]);
  }
  int division_of_array(int x[], int st, int lt)
  {
     int left, right, temp, key, flag;
      key = left = st; // initially these variables are in same place.
     right = lt;
      flag = 0;
     while(flag != 1)
      {
        while((x[key] <= x[right]) && (key != right))
         right--;
        if(key == right) // if pivot element is equal to the last index.
         flag = 1;
        else if(x[key] > x[right]) // if pivot is greater than last index.
         {
           temp = x[key];
```

```
            x[key] = x[right];
            x[right] = temp;
            key = right;
        }
        if (flag != 1)
        {       /* Repeat the while loop until pivot is greater than left and equal to the left. */
            while((x[key] >= x[left]) && (key != left))
            left++;
            if(key == left)
            flag = 1;
            else if(x[key] < x[left])
            {
                temp = x[key];
                x[key] = x[left];
                x[left] = temp;
                key = left;
            }
        }
    }
    return key;
  }
}void quick_sort(int x[], int st, int lt)
{
   int key;
   if(st<lt) // left index value is greater than right index.
   {
      key = division_of_array(x, st, lt);
      quick_sort(x, st, key-1);
      quick_sort(x, key+1, lt);
   }

}
```

Output:

```
Sorted Array is:
Array = 50  20  60  30  40  56
```

Merge sort

Merge sort is a most important sorting techniques that work on the divide and conquer strategies. It is the most popular sorting techniques used to sort data that is externally available in a file. The merge sort algorithm divides the given array into two halves (N/2). And then, it recursively divides the set of two halves array elements into the single or individual elements or we can say that until no more division can take place. After that, it compares the corresponding element to sort the element and finally, all sub elements are combined to form the final sorted elements.

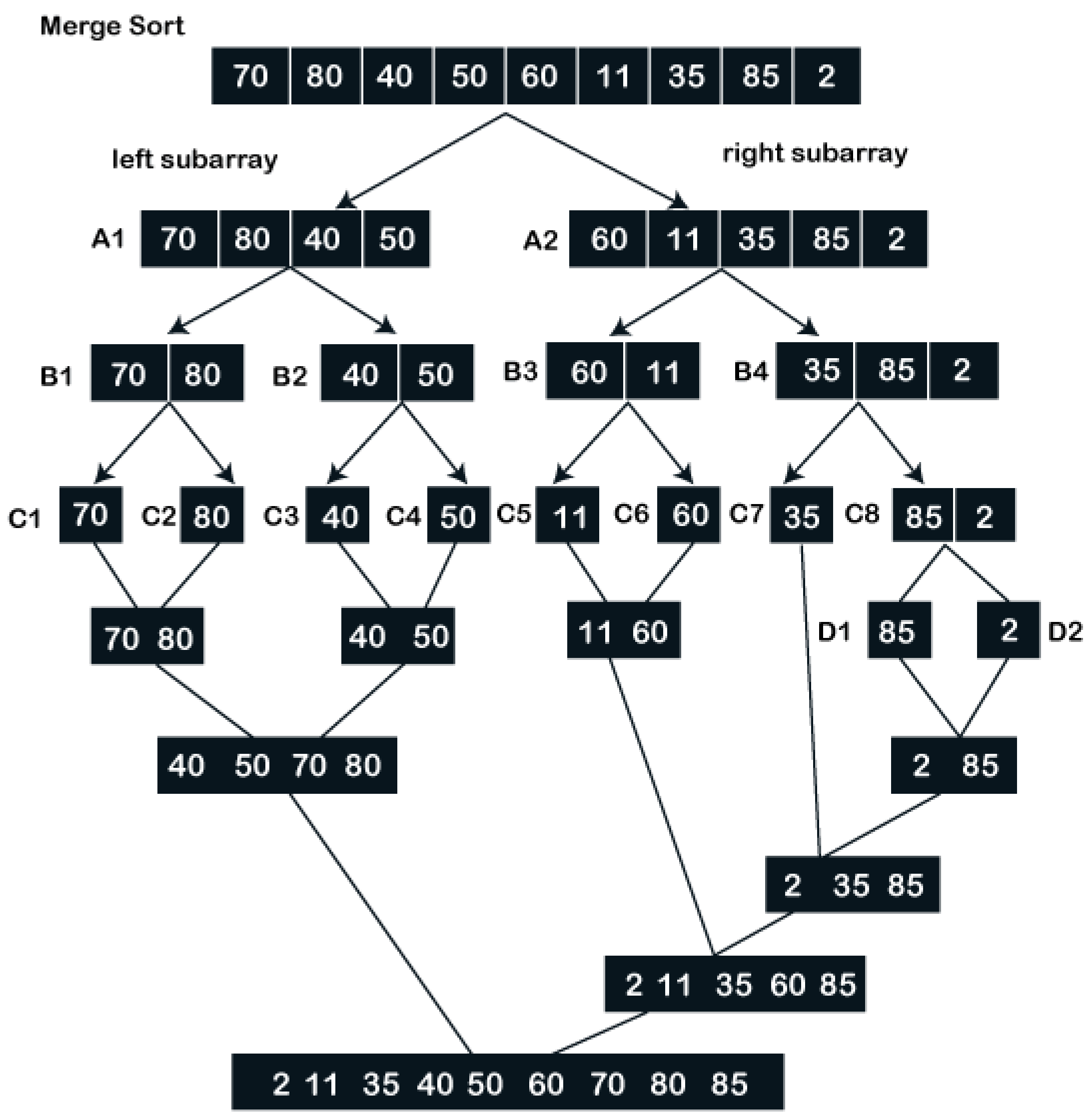
Steps to sort an array using the Merge sort algorithm

1. Suppose we have a given array, then first we need to divide the array into sub array. Each sub array can store 5 elements.
2. Here we gave the first sub array name as A1 and divide into next two subarray as B1 and B2.
3. Similarly, the right sub array name as A2 and divide it into next two sub array as B3 and B4.
4. This process is repeated continuously until the sub array is divided into a single element and no more partitions may be possible.
5. After that, compare each element with the corresponding one and then start the process of merging to arrange each element in such a way that they are placed in ascending order.
6. The merging process continues until all the elements are merged in ascending order.

Let's see an example of merge sort.

Example: Consider an array of 9 elements. Sort the array using the merge sort.

arr[] = {70, 80, 40, 50, 60, 11, 35, 85, 2}



Hence, we get the sorted array using the merge sort.

Let's implement the above logic in a C program.

Merge.c

```
#include <stdio.h>
//#include <conio.h>
#include <stdlib.h>

void merge(int arr[], int l, int m, int end)
{
  int i, j, k;
  int a1 = m - l + 1;
  int a2 = end - m;

  // create temp subarray
  int sub1[a1], sub2[a2];

  // Store data to temp subarray subArr1[] and subArr2[]
  for (i = 0; i < a1; i++)
     sub1[i] = arr[l + i];
  for (j = 0; j < a2; j++)
     sub2[j] = arr[m + 1 + j];

  // Merge the temp array into the original array arr[]
  i = 0;
  j = 0;
  k = l;
  while (i < a1 && j < a2)
  {
     if (sub1[i] <= sub2[j])
     {
        arr[k] = sub1[i];
         i++;
```

```c
        }
        else
        {
            arr[k] = sub2[j];
           j++;
        }
        k++;
    }

    // copry the rest element of subArr1[], if some element is left;
while (i < a1)
{
    arr[k] = sub1[i];
    i++;
    k++;
}

    // copry the rest element of subArr2[], if some element is left;
while (j < a2)
{
    arr[k] = sub2[j];
    j++;
    k++;
  }
}

/* st represents the first index and end represents the right index of the subarray of arr[] to be sorted using merge sort. */
void mergeSort(int arr[], int l, int end)
{
    if (l < end)
    {
        int m = l + (end - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, end);

        merge(arr, l, m, end);
    }
}

// Function to print the merge sort.
void mergePrint(int Ar[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d \t", Ar[i]);
    printf("\n");
}

int main()
{
    int arr[] = { 70, 80, 40, 50, 60, 11, 35, 85, 2 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf(" Predefined Array is \n");
    mergePrint(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\n Sorted array using the Merge Sort algorithm \n");
    mergePrint(arr, arr_size);
    return 0;
}
```

Output:

```
Predefined Array is
70      80      40      50      60      11      35      85      2

 Sorted array using the Merge Sort algorithm
2       11      35      40      50      60      70      80      85
```

Quick Sort vs. Merge Sort

S.N.	Parameter	Quick Sort	Merge Sort
1.	**Definition**	It is a quick sort algorithm that arranges the given elements into ascending order by comparing and interchanging the position of the elements.	It is a merge sort algorithm that arranges the given sets of elements in ascending order using the divide and conquer technique, and then compare with corresponding elements to sort the array.
2.	**Principle**	It works on divide and conquer techniques.	It works on divide and conquer techniques.
3.	**Partition of elements**	In quick sort, the array can be divide into any ratio.	Merge sort partition an array into two sub array (N/2).
4.	**Efficiency**	It is more efficient and work faster in smaller size array, as compared to the merge sort.	It is more efficient and work faster in larger data sets or array, as compare to the quick sort.
5	**Sorting method**	It is an internal sorting method that sort the array or data available on main memory.	It is an external sorting method that sort the array or data sets available on external file.
6	**Time complexity**	Its worst time complexity is O (n2).	Whereas, it's worst time complexity is O (n log n).
7	**Preferred**	It is a sorting algorithm that is applicable for large unsorted arrays.	Whereas, the merge sort algorithm that is preferred to sort the linked lists.
8	**Stability**	Quick sort is an unstable sort algorithm. But we can made it stable by using some changes in programming code.	Merge sort is a stable sort algorithm that contains two equal elements with same values in sorted output.
9	**Requires Space**	It does not require any additional space to perform the quick sort.	It requires the additional space as temporary array to merge two sub arrays.
10.	**Functionality**	Compare each element with the pivot until all elements are arranged in ascending order.	Whereas, the merge sort splits the array into two parts (N/2) and it continuously divides the array until an element is left.

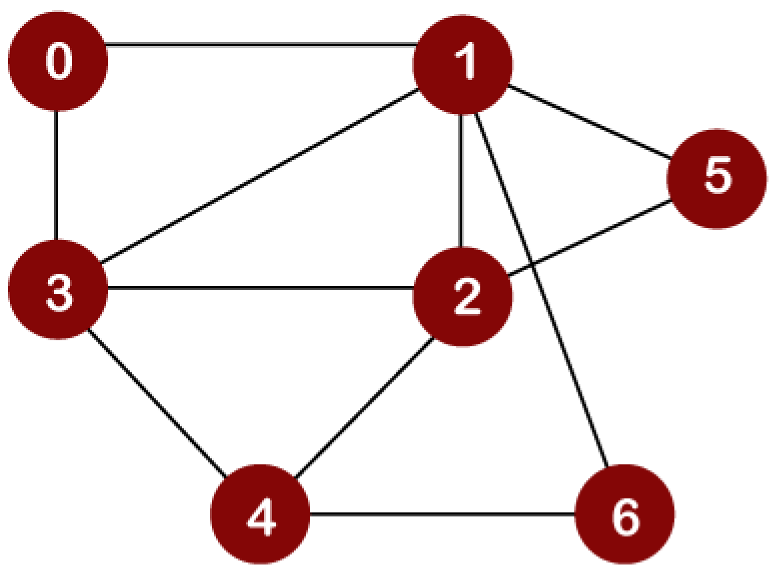
BFS vs. DFS

Before looking at the differences between BFS and DFS, we first should know about BFS and DFS separately.

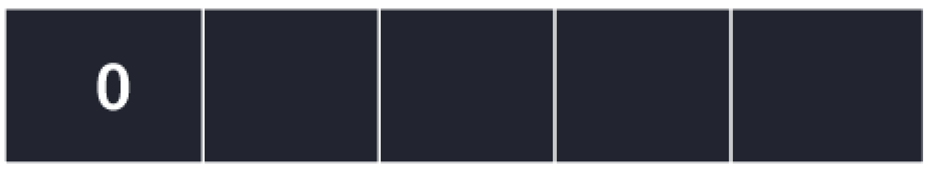
What is BFS?

BFS stands for ***Breadth First Search***. It is also known as **level order traversal**. The Queue data structure is used for the Breadth First Search traversal. When we use the BFS algorithm for the traversal in a graph, we can consider any node as a root node.

Let's consider the below graph for the breadth first search traversal.

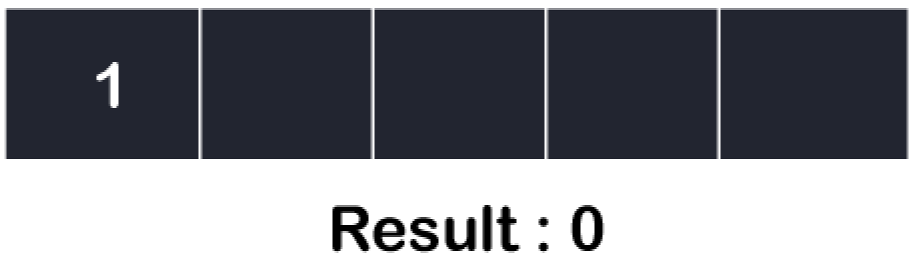


Suppose we consider node 0 as a root node. Therefore, the traversing would be started from node 0.



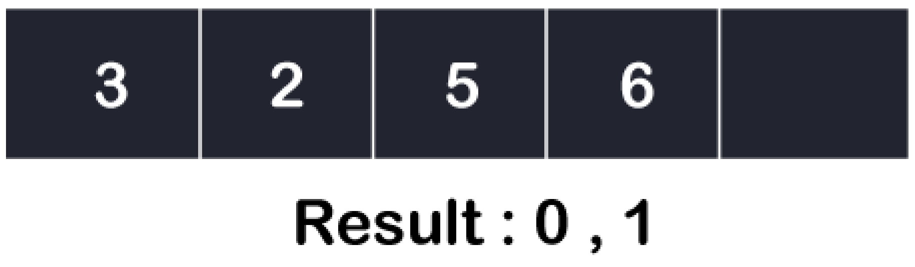
Once node 0 is removed from the Queue, it gets printed and marked as a ***visited node.***

Once node 0 gets removed from the Queue, then the adjacent nodes of node 0 would be inserted in a Queue as shown below:

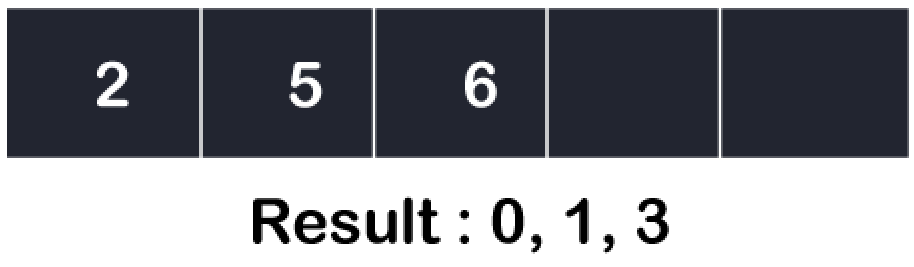


Now the node 1 will be removed from the Queue; it gets printed and marked as a visited node

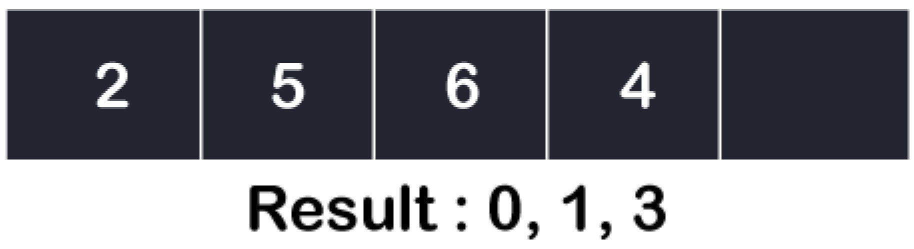
Once node 1 gets removed from the Queue, then all the adjacent nodes of a node 1 will be added in a Queue. The adjacent nodes of node 1 are 0, 3, 2, 6, and 5. But we have to insert only unvisited nodes in a Queue. Since nodes 3, 2, 6, and 5 are unvisited; therefore, these nodes will be added in a Queue as shown below:



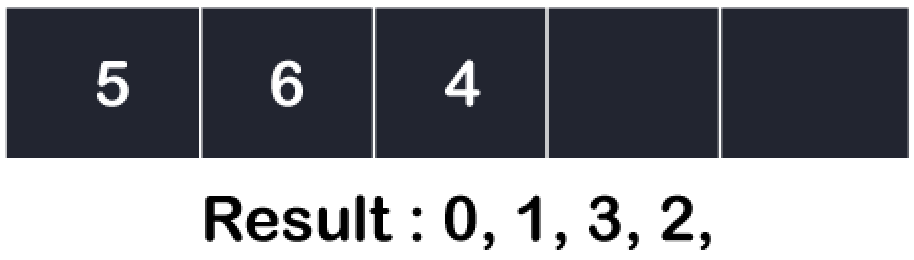
The next node is 3 in a Queue. So, node 3 will be removed from the Queue, it gets printed and marked as visited as shown below:



Once node 3 gets removed from the Queue, then all the adjacent nodes of node 3 except the visited nodes will be added in a Queue. The adjacent nodes of node 3 are 0, 1, 2, and 4. Since nodes 0, 1 are already visited, and node 2 is present in a Queue; therefore, we need to insert only node 4 in a Queue.

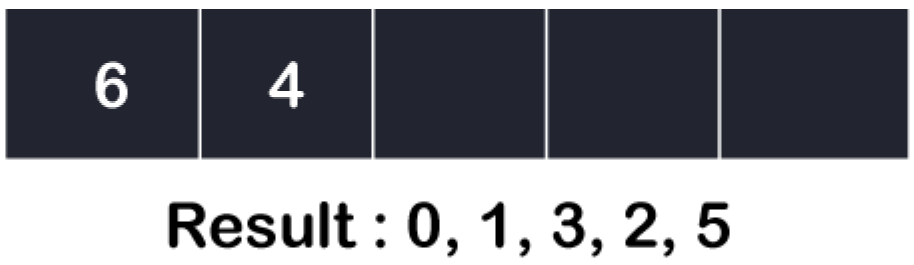


Now, the next node in the Queue is 2. So, 2 would be deleted from the Queue. It gets printed and marked as visited as shown below:



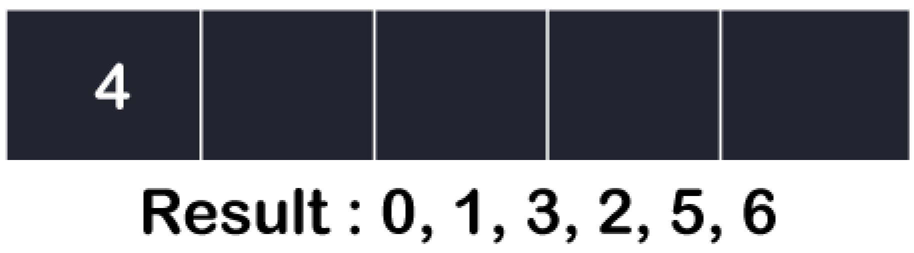
Once node 2 gets removed from the Queue, then all the adjacent nodes of node 2 except the visited nodes will be added in a Queue. The adjacent nodes of node 2 are 1, 3, 5, 6, and 4. Since the nodes 1 and 3 have already been visited, and 4, 5, 6 are already added in the Queue; therefore, we do not need to insert any node in the Queue.

The next element is 5. So, 5 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Once node 5 gets removed from the Queue, then all the adjacent nodes of node 5 except the visited nodes will be added in the Queue. The adjacent nodes of node 5 are 1 and 2. Since both the nodes have already been visited; therefore, there is no vertex to be inserted in a Queue.

The next node is 6. So, 6 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Once the node 6 gets removed from the Queue, then all the adjacent nodes of node 6 except the visited nodes will be added in the Queue. The adjacent nodes of node 6 are 1 and 4. Since the node 1 has already been visited and node 4 is already added in the Queue; therefore, there is not vertex to be inserted in the Queue.

The next element in the Queue is 4. So, 4 would be deleted from the Queue. It gets printed and marked as visited.

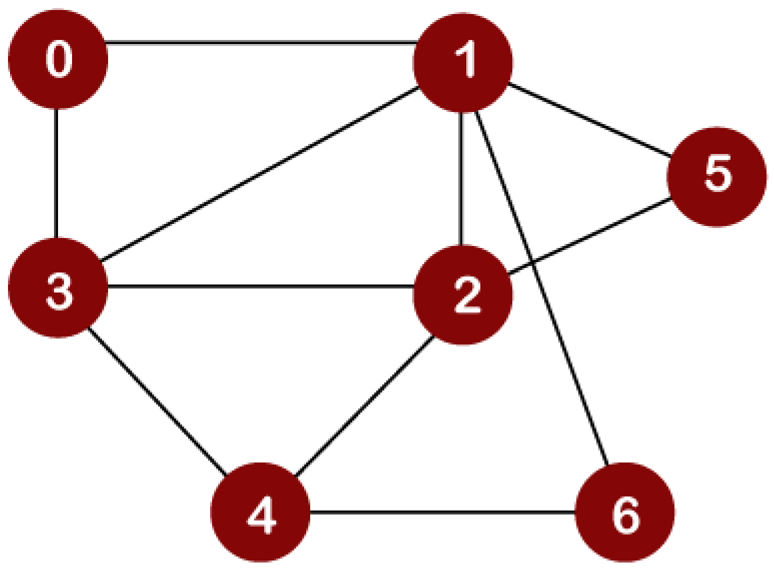
Once the node 4 gets removed from the Queue, then all the adjacent nodes of node 4 except the visited nodes will be added in the Queue. The adjacent nodes of node 4 are 3, 2, and 6. Since all the adjacent nodes have already been visited; so, there is no vertex to be inserted in the Queue.

What is DFS?

DFS stands for Depth First Search. In DFS traversal, the stack data structure is used, which works on the LIFO (Last In First Out) principle. In DFS, traversing can be started from any node, or we can say that any node can be considered as a root node until the root node is not mentioned in the problem.

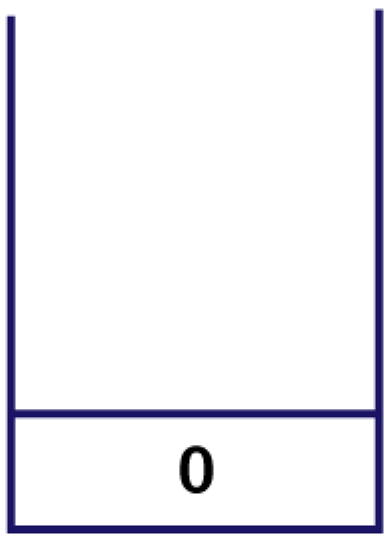
In the case of BFS, the element which is deleted from the Queue, the adjacent nodes of the deleted node are added to the Queue. In contrast, in DFS, the element which is removed from the stack, then only one adjacent node of a deleted node is added in the stack.

Let's consider the below graph for the Depth First Search traversal.

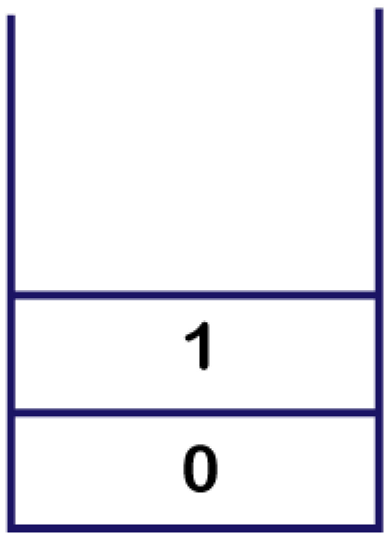


Consider node 0 as a root node.

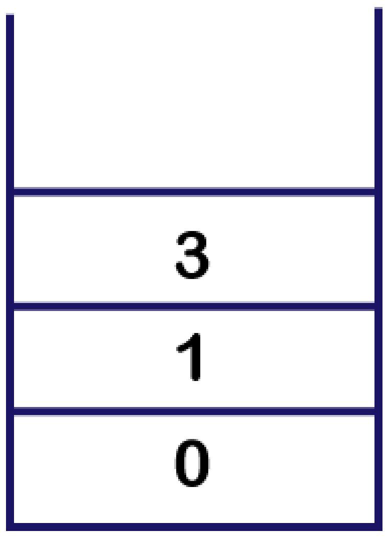
First, we insert the element 0 in the stack as shown below:



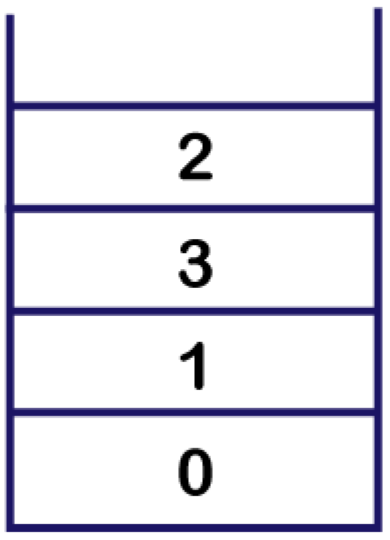
The node 0 has two adjacent nodes, i.e., 1 and 3. Now we can take only one adjacent node, either 1 or 3, for traversing. Suppose we consider node 1; therefore, 1 is inserted in a stack and gets printed as shown below:



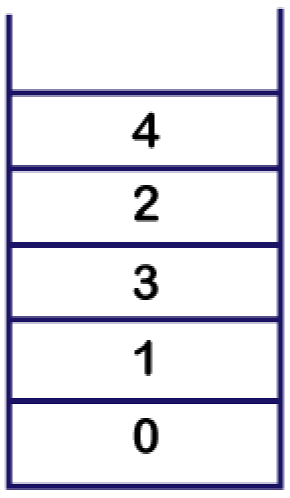
Now we will look at the adjacent vertices of node 1. The unvisited adjacent vertices of node 1 are 3, 2, 5 and 6. We can consider any of these four vertices. Suppose we take node 3 and insert it in the stack as shown below:



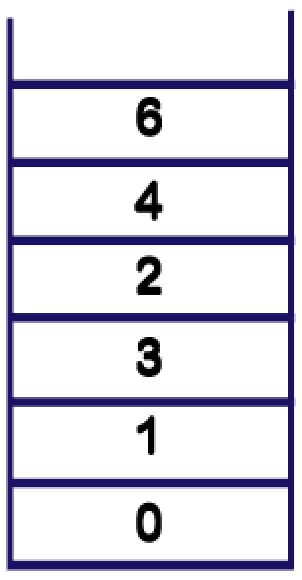
Consider the unvisited adjacent vertices of node 3. The unvisited adjacent vertices of node 3 are 2 and 4. We can take either of the vertices, i.e., 2 or 4. Suppose we take vertex 2 and insert it in the stack as shown below:



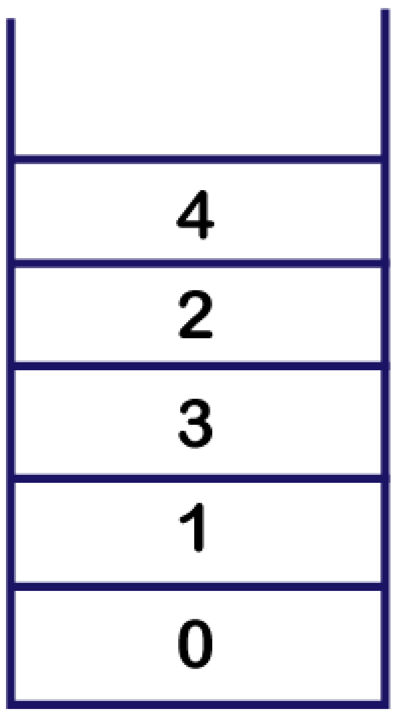
The unvisited adjacent vertices of node 2 are 5 and 4. We can choose either of the vertices, i.e., 5 or 4. Suppose we take vertex 4 and insert in the stack as shown below:

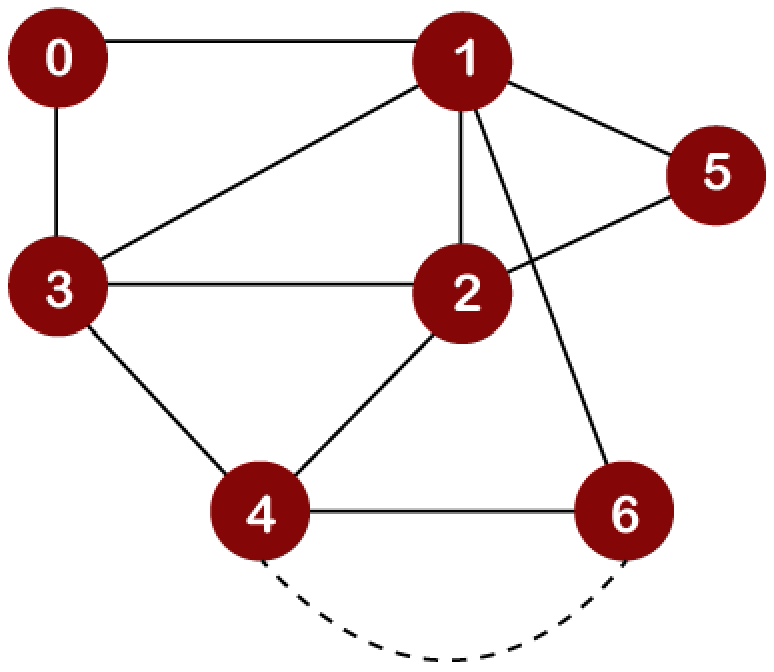


Now we will consider the unvisited adjacent vertices of node 4. The unvisited adjacent vertex of node 4 is node 6. Therefore, element 6 is inserted into the stack as shown below:

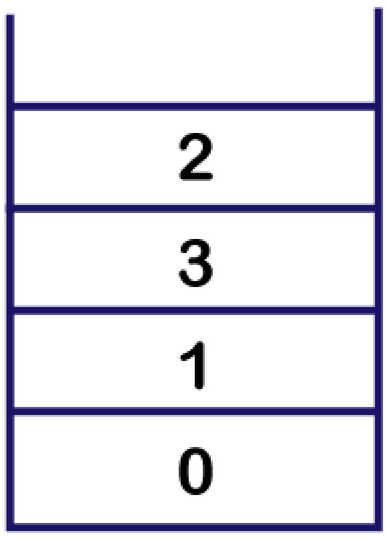


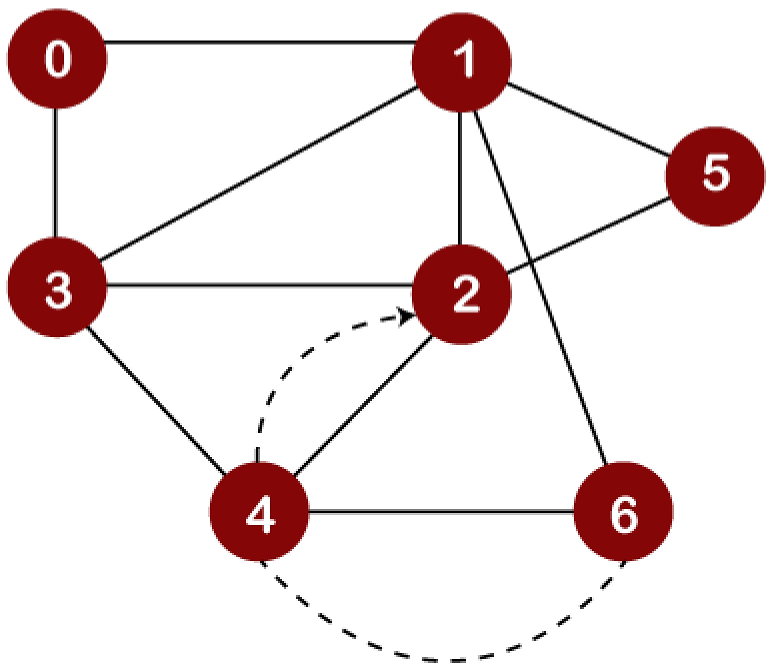
After inserting element 6 in the stack, we will look at the unvisited adjacent vertices of node 6. As there is no unvisited adjacent vertices of node 6, so we cannot move beyond node 6. In this case, we will perform **backtracking**. The topmost element, i.e., 6 would be popped out from the stack as shown below:



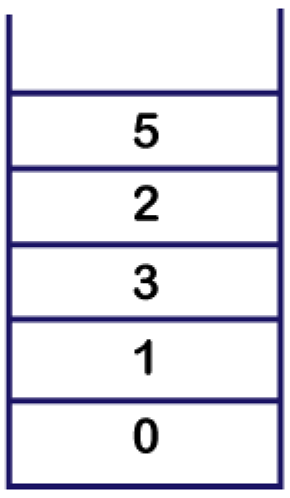


The topmost element in the stack is 4. Since there are no unvisited adjacent vertices left of node 4; therefore, node 4 is popped out from the stack as shown below:

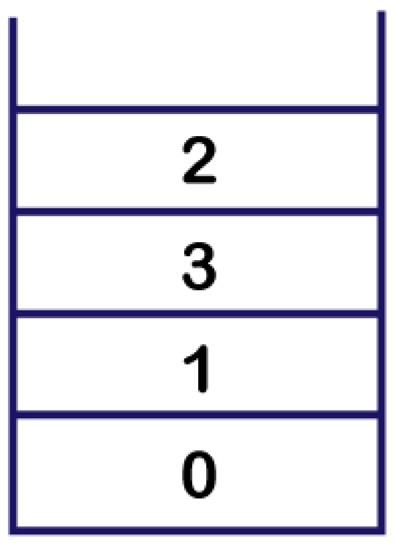




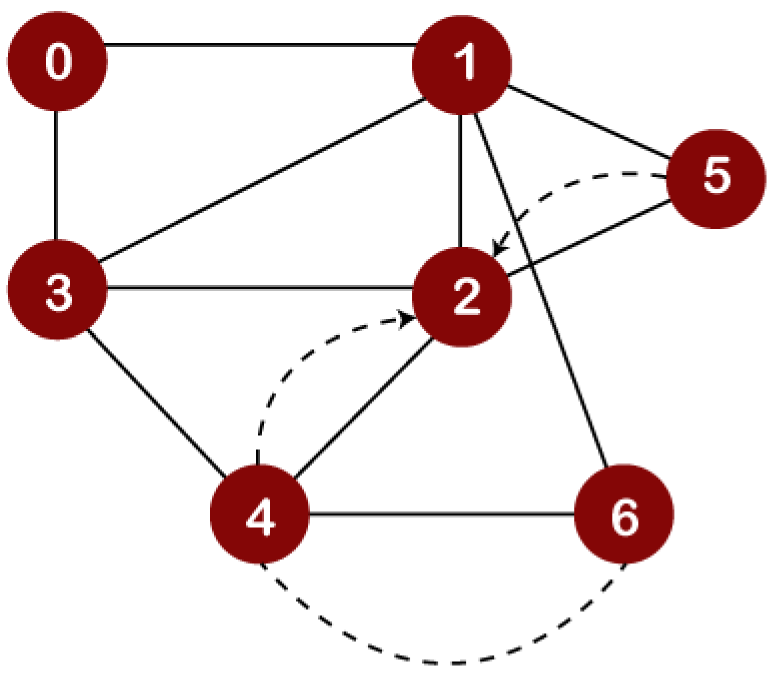
The next topmost element in the stack is 2. Now, we will look at the unvisited adjacent vertices of node 2. Since only one unvisited node, i.e., 5 is left, so node 5 would be pushed into the stack above 2 and gets printed as shown below:



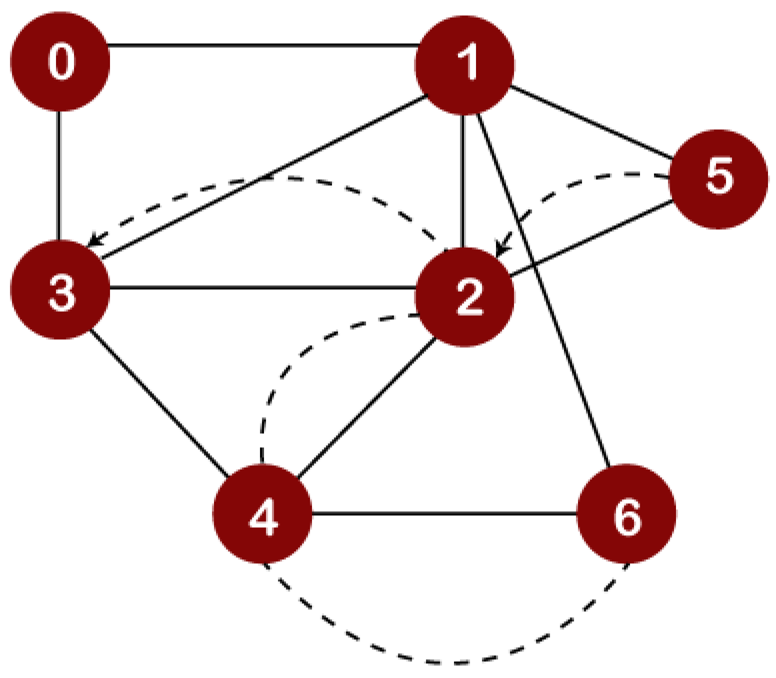
Now we will check the adjacent vertices of node 5, which are still unvisited. Since there is no vertex left to be visited, so we pop the element 5 from the stack as shown below:

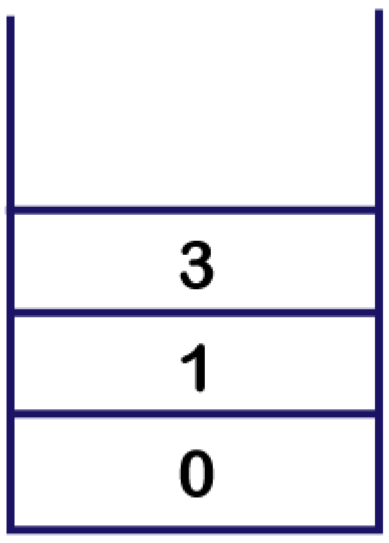


We cannot move further 5, so we need to perform backtracking. In backtracking, the topmost element would be popped out from the stack. The topmost element is 5 that would be popped out from the stack, and we move back to node 2 as shown below:

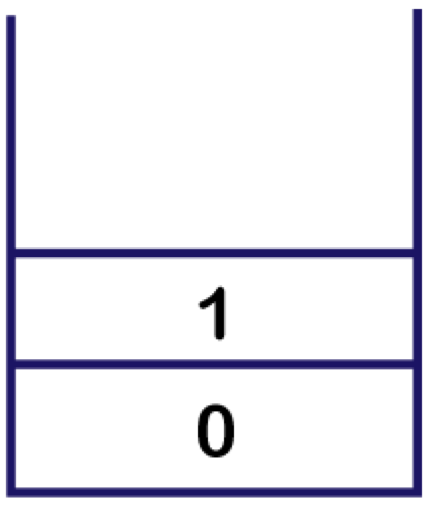


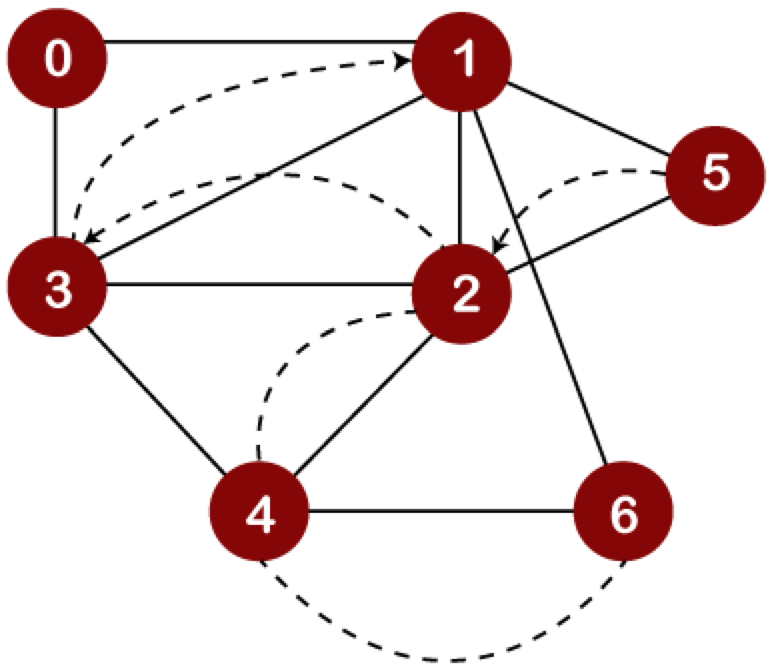
Now we will check the unvisited adjacent vertices of node 2. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 2 would be popped out from the stack, and we move back to the node 3 as shown below:



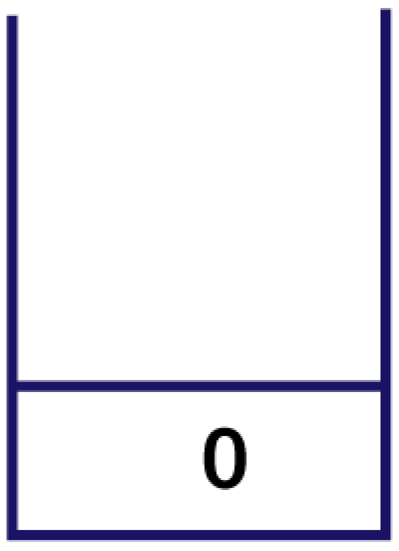


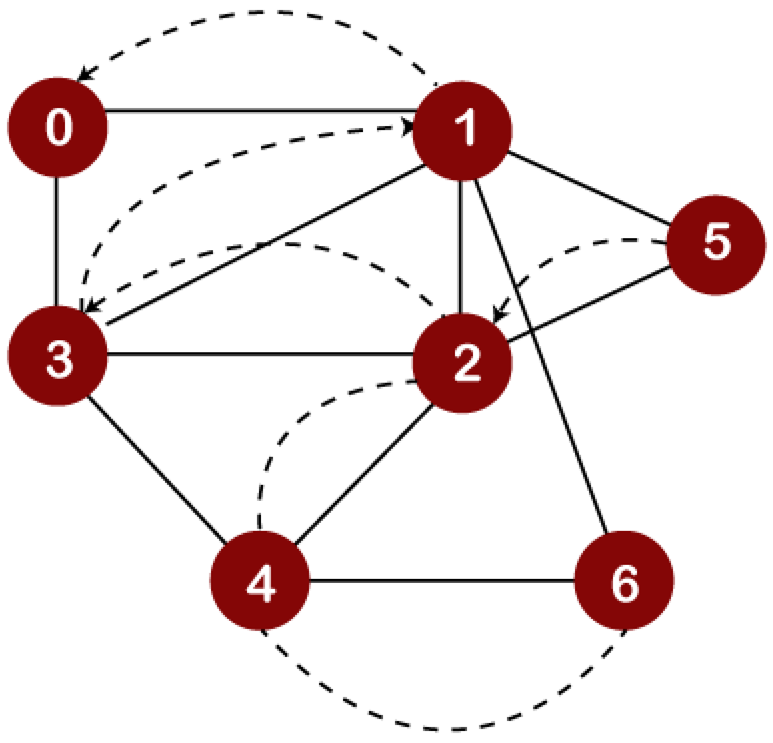
Now we will check the unvisited adjacent vertices of node 3. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 3 would be popped out from the stack and we move back to node 1 as shown below:



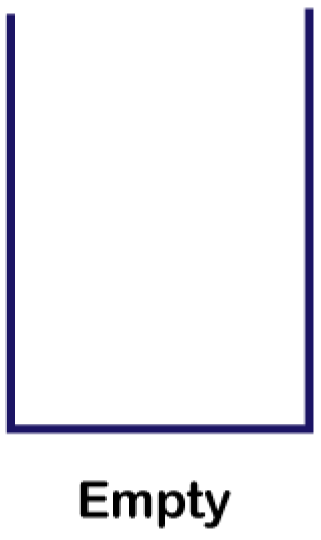


After popping out element 3, we will check the unvisited adjacent vertices of node 1. Since there is no vertex left to be visited; therefore, the backtracking will be performed. In backtracking, the topmost element, i.e., 1 would be popped out from the stack, and we move back to node 0 as shown below:





We will check the adjacent vertices of node 0, which are still unvisited. As there is no adjacent vertex left to be visited, so we perform backtracking. In this, only one element, i.e., 0 left in the stack, would be popped out from the stack as shown below:



As we can observe in the above figure that the stack is empty. So, we have to stop the DFS traversal here, and the elements which are printed is the result of the DFS traversal.

Differences between BFS and DFS

The following are the differences between the BFS and DFS:

	BFS	DFS
Full form	BFS stands for Breadth First Search.	DFS stands for Depth First Search.

Technique	It a vertex-based technique to find the shortest path in a graph.	It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node.
Definition	BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level.	DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes.
Data Structure	Queue data structure is used for the BFS traversal.	Stack data structure is used for the BFS traversal.
Backtracking	BFS does not use the backtracking concept.	DFS uses backtracking to traverse all the unvisited nodes.
Number of edges	BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex.	In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex.
Optimality	BFS traversal is optimal for those vertices which are to be searched closer to the source vertex.	DFS traversal is optimal for those graphs in which solutions are away from the source vertex.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Suitability for decision tree	It is not suitable for the decision tree because it requires exploring all the neighboring nodes first.	It is suitable for the decision tree. Based on the decision, it explores all the paths. When the goal is found, it stops its traversal.
Memory efficient	It is not memory efficient as it requires more memory than DFS.	It is memory efficient as it requires less memory than BFS.

Stack vs Heap

Before understanding the differences between the Stack and Heap data structure, we should know about the stack and heap data structure separately.

What is Stack?

A Stack is a data structure that is used for organizing the data. Stack is similar to the stack a way organizing the objects in the real world. Some examples of stack of real world are stack of dinner plates, mathematical puzzle known as Tower of Hanoi containing three rods having multiple discs and stack of tennis balls. Stack is a collection with a property that an item or an object must be removed from one end known as the ***top of the stack***.

It cannot be considered as the property rather it is considered as the constraint or restriction applied to the stack. In other words, we can say that only top of the stack is accessible and any item can be removed or inserted from the top of the stack. It follows LIFO (Last In First Out) principle in which recently added element must be removed from the stack first.

A stack is a list or a collection with a restriction that the insertion and deletion will take place from one end known as the top of the stack.

Operation

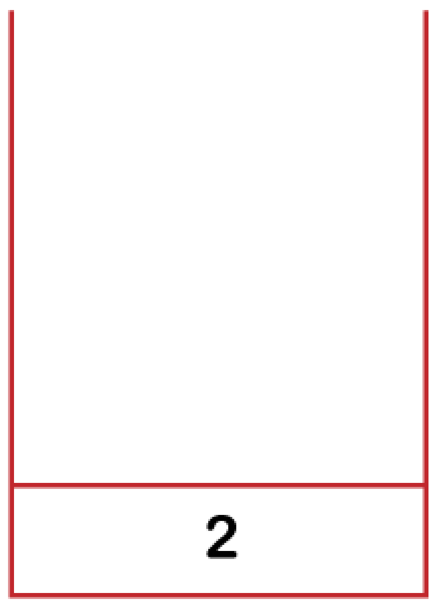
The following are the two operations available with a stack ADT:

- **Push:** The process of inserting an element in a stack is known as push operation. The posh operation can be written as:
  Push(x): It inserts an element x into the stack.
- **Pop:** The process of deleting an element from the stack is known as a pop operation. The pop operation can be represented as:
  Pop(): It removes the most recent element from the stack.

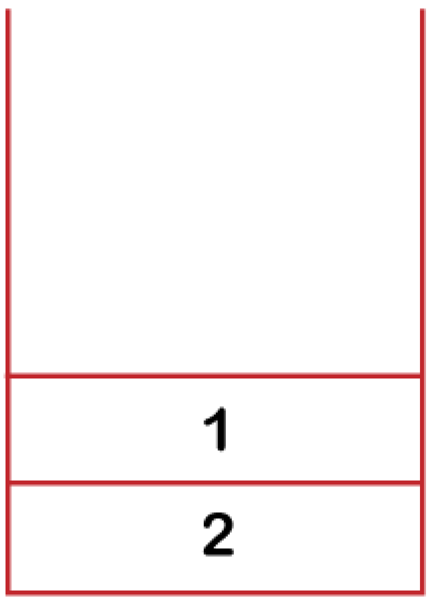
Representation of stack

Stack is shown below:

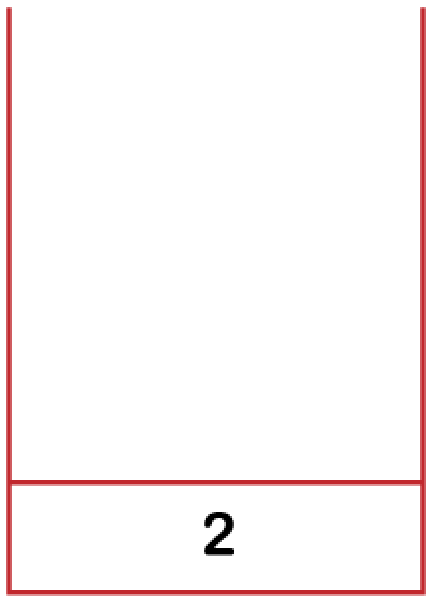
We can observe in the above figure that stack looks like a container which is opened from one side. It can be represented logically as a three-sided figure as a container open from one side. The above representation is an empty stack and let's assume that stack is 's'. It is a stack of integers. Now we will perform push and pop operations on the stack. Suppose we want to insert 2 element in the stack. After push operation, the stack would look like:



Since there is only one element in the stack, so 2 element would be at the top of the stack. If we want to insert 1 element in the stack, then the stack would like:



Since element 1 comes above the element 2, so element 1 would be considered the top of the stack. If we perform pop operation, then the topmost element, i.e., 1 would be removed from the stack as shown below:



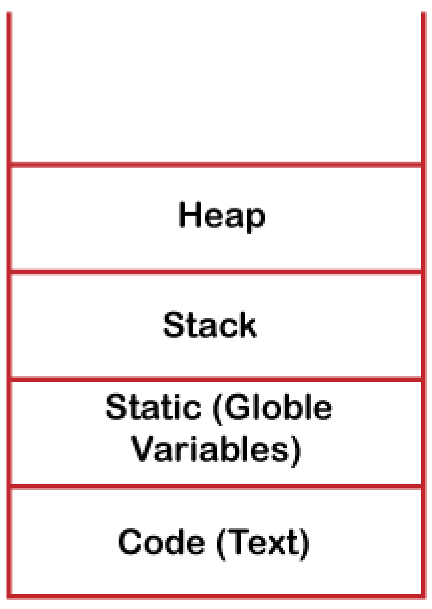
What is Heap?

Heap is also a data structure or memory used to store the global variables. By default, all the global variables are stored in the heap memory. It allows dynamic memory allocation. The heap memory is not managed by CPU. Heap data structure can be implemented either using arrays or trees.

It is a complete binary tree that satisfies the condition of the heap property where complete binary tree is a tree in which all the levels are completely filled except the last level. In the last level, all the nodes are far as left as possible.

Pointers and Dynamic memory

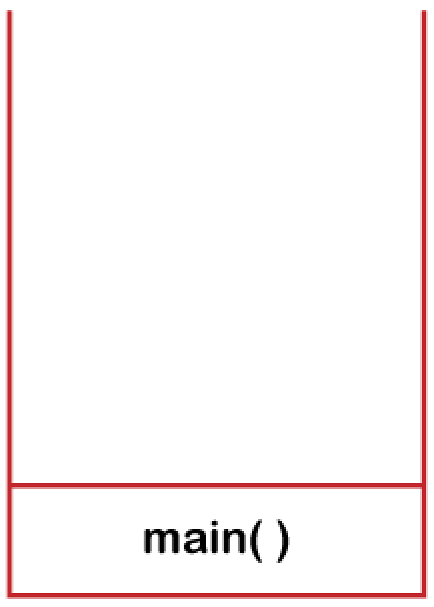
Here, we will see the architecture of memory. It is very crucial to know that how system manages the memory and accessible to us as programmers. The memory which is assigned to the program or application in a typical architecture is divided into four segments. One segment of the memory stores the instructions which are to be executed. Another segment of the memory stores the global or the static variables. The global variables are the variables which are declared outside the function and the lifetime is throughout the program. The third segment, i.e., stack is used to store all the function calls and the local variables. When any function is called then it occupies some space in the memory known as a stack memory.



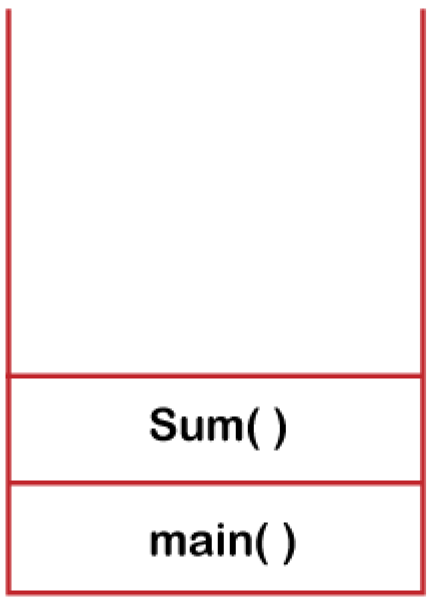
Let's understand the stack memory through an example.

```c
#include <stdio.h>
int square(int s)
{
    return s*s;
}
int sum(int x,int y)
{
    int z= x+y;
    int result = square(z);
    return result;
}
int main()
{
  int a=10;
  int b=20;
  int x = sum(a,b);
  printf("Output is : %d", x);
  return 0;
}
```

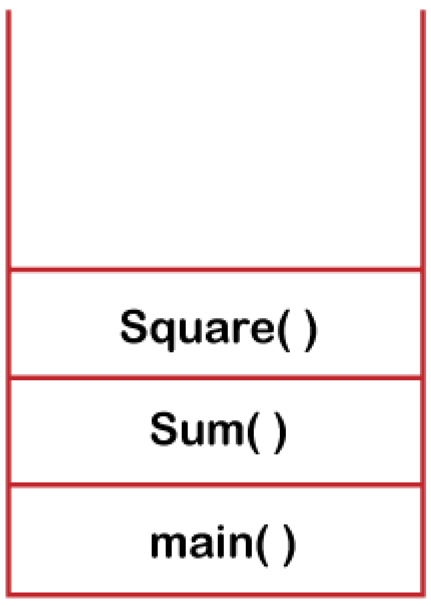
In the above code, execution starts from the main() method. So main() method would be given a memory in the stack as shown below:



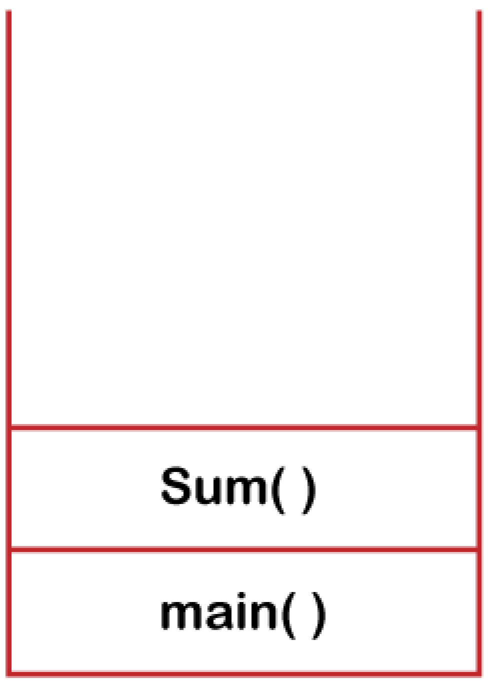
When the sum() method is called, the control moves to the **sum()** function.



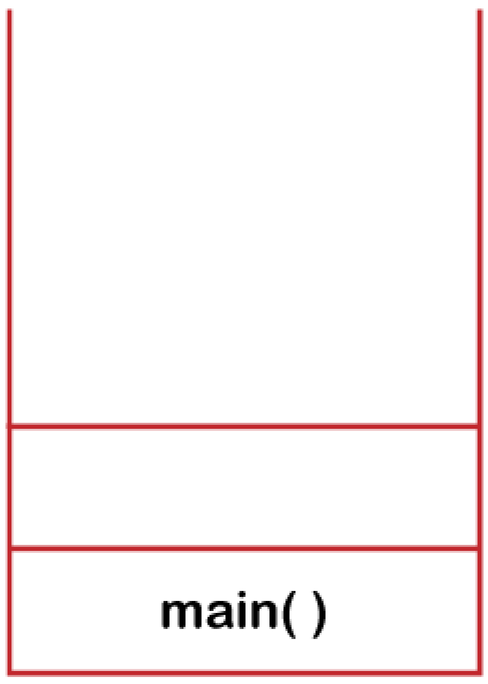
The sum() method calls the square() method; therefore, the square method would be given a memory in the stack as shown below:



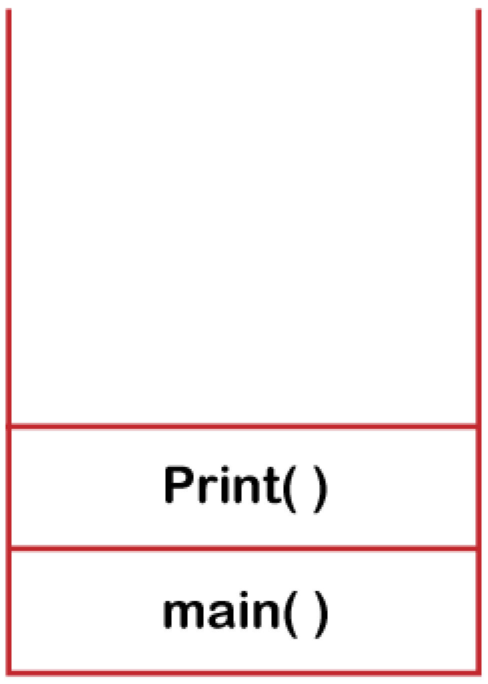
Once the square() method return statement is executed, the control moves back to the sum() method and the square() method gets removed from the stack as shown below:



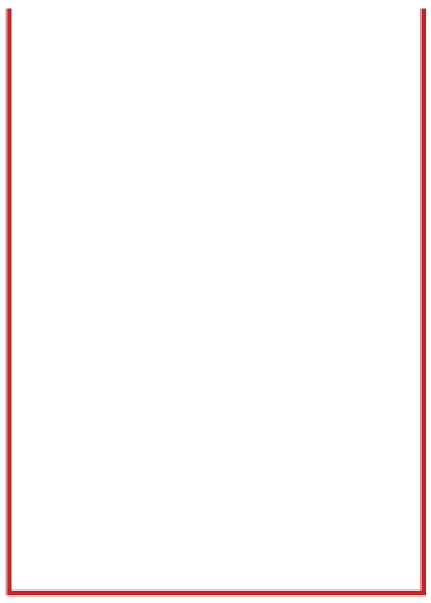
When the return statement of the sum() method is executed, the control moves to the main() method and sum() method gets removed from the stack as shown below:



In the main() method, printf() function is called so it gets memory in the stack as shown below:



Once the execution of printf() statement is completed, the printf() and main() methods are removed from the stack memory as shown below:



There are some limitations of using a stack memory. Suppose operating system reserves 1MB stack memory for the program. If program keeps calling the functions again then stack memory would not be sufficient and it leads to the stack overflow condition. The stack overflow causes a program to crash. So, we can say that the stack memory does not grow runtime.

Another limitation of stack is that the scope of the variable cannot be manipulated. The allocation and deallocation of memory onto the stack are set by the rule, i.e., when the function is called then it is pushed onto the top of the stack and when pop() operation is called then the element is removed from the top of the stack.

The third limitation of the stack is that if we define the large data type such as array and the size of the array is not defined at the compile time. We want to define the size of the array based on some parameters, and defining the size of the array at the runtime is not possible with the stack.

So, to allocate the large chunks of memory and keep the memory aside till the time we want, we can use a **heap** data structure. Unlike stack data structure, the size of the heap memory can vary and it is not fixed throughout the lifetime of the application. In heap memory, there is no set rule for the allocation and deallocation of memory. A programmer can itself manually handle the memory. The abstracted way for the programmer of looking at the heap as a large free of memory available to use and we can use it as per our needs.

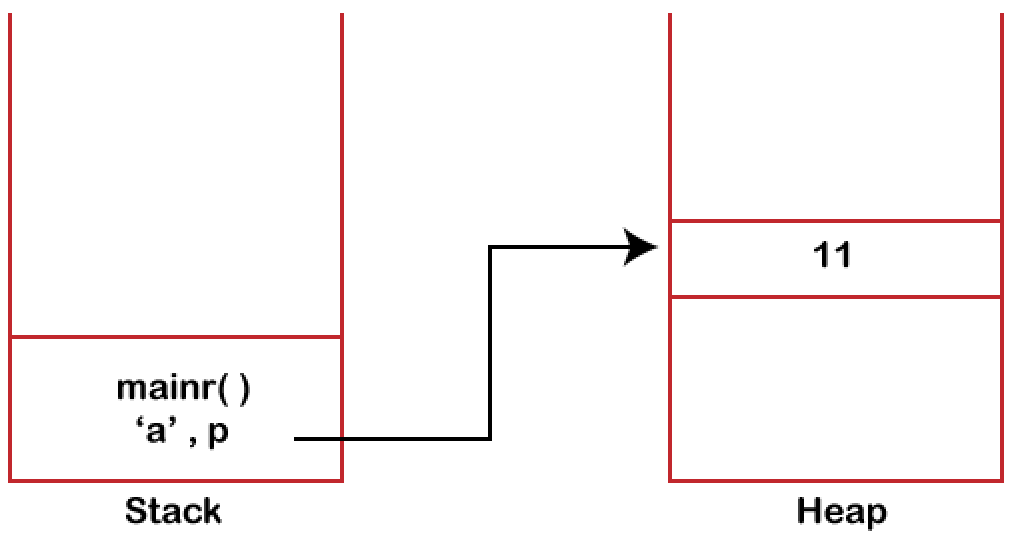
Heap is also known as a dynamic memory and using heap, can be considered as the dynamic memory allocation. To use dynamic memory, we need to use some functions. In C language, we can use malloc() and calloc() to allocate the memory and free() function to deallocate the memory, whereas, in C++ language, we use new operator for the allocation and delete operator for the deallocation.

Let's understand the dynamic memory through an example.

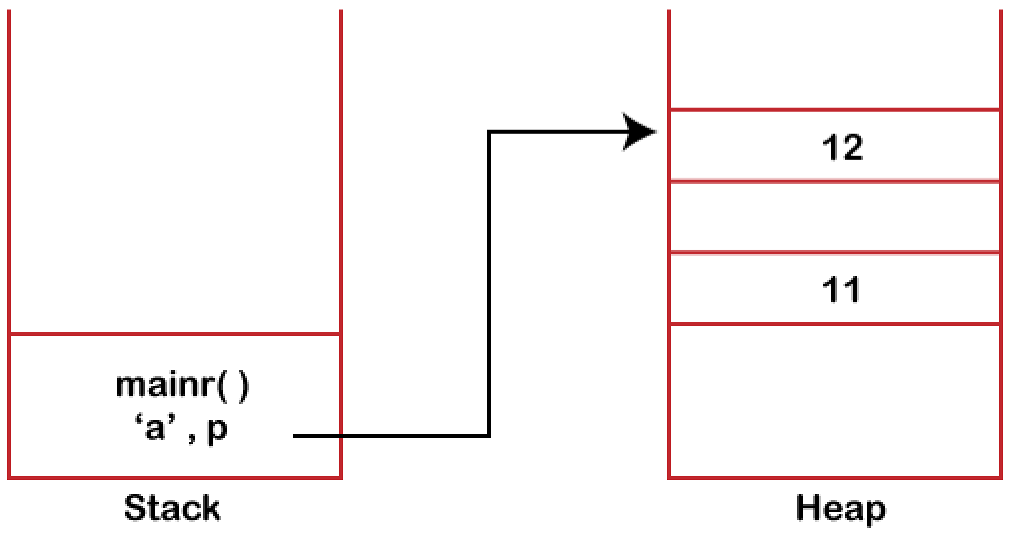
```
#include<stdio.h>
int main()
{
   int a; // goes onto stack.
   int *p;
   p = (int*)malloc(sizeof(int));
   *p = 11;
    free(p);
     p = (int*)malloc(sizeof(int));
     *p = 12;
}
```

Explanation of the above code.

First, we declared the 'a' variable and it gets allocated within the stack frame of the ***main()*** method in the stack as shown below:

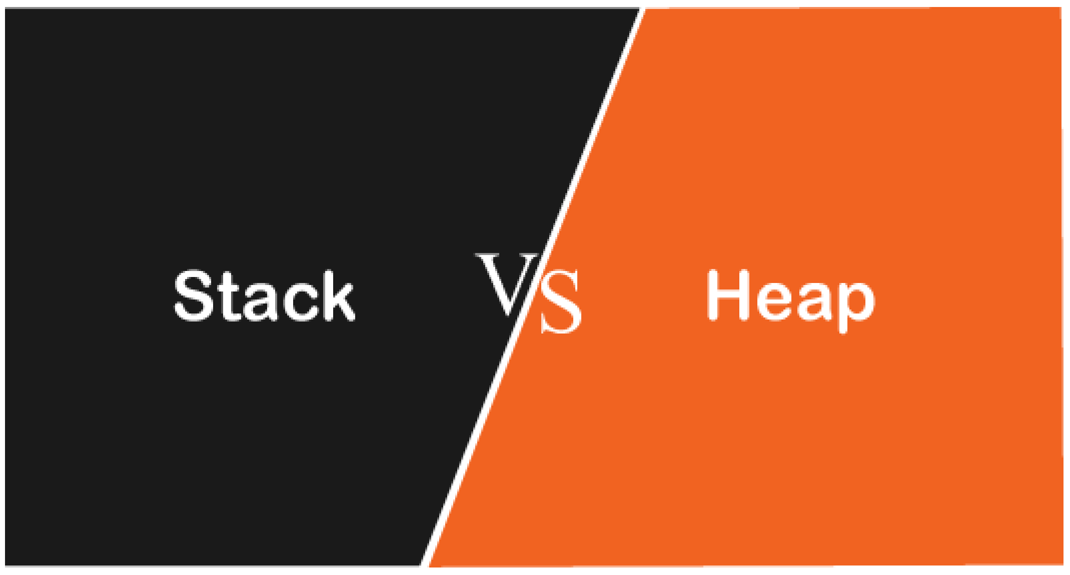


To allocate something in the **heap memory**, we need to use the malloc() function. We have used the malloc() function in the above code in which we pass the sizeof(int) defines that 4bytes of block is allocated in the heap memory. This function returns the void pointer that contains the starting address of the block. The 'p' is a pointer variable local to the function so it gets stored in the stack as shown below:



Again, we have allocated new block of memory pointed by the 'p' variable, so 'p' does not hold the address of the previous block. The block having value 11 is an unnecessary consumption of memory. In heap, the memory is not automatically deallocated, we have to release the memory manually. So, in the above code, we have used the free(p) function in which we pass the 'p' to deallocate the memory pointed by 'p'.

Differences between Stack and Heap



Stack	Heap
Stack provides static memory allocation, i.e., it is used to store the temporary variables.	Heap provides dynamic memory allocation. By default, all the global variables are stored in the heap.
It is a linear data structure means that elements are stored in the linear manner, i.e., one data after another.	It is hierarchical data structure means that the elements are stored in the form of tree.
It is used to access the local variables.	It is used to access the global variables by default.
The size of the stack memory is limited which is dependent on the OS.	The size of the memory is not limited.
As it is a linear data structure, so data is stored in the contiguous blocks.	As it is hierarchical data structure, so elements are stored in the random manner.
In stack, the allocation and deallocation are automatically managed.	In heap, the memory is manually managed.
The implementation of stack can be done in three forms using array, linked list and dynamic memory.	The implementation of heap can be done in two forms using arrays and trees.
The main issue that occurs with a stack is the shortage of memory because the memory size cannot be changed at the runtime. The size of the stack is determined at the compile time.	The main issue that occurs with a heap is the memory fragmentation. Here, memory fragmentation means that the memory gets wasted.

It is of fixed size.	It is flexible to use as the size of the heap can vary as per our needs.
The access time in stack is faster.	The access time in heap is slower.
The size of the stack memory is decided by the operating system.	The size of the heap memory is decided by the programmers.
The scope of the variable cannot be changed.	The scope of the variable can be changed.

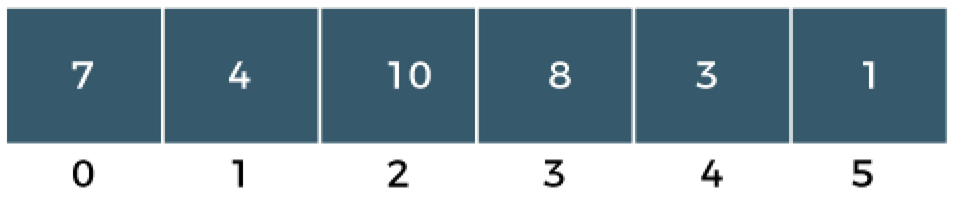
Bubble sort vs. Selection sort

Here we will look at the differences between the selection sort and bubble sort. Before understanding the differences, we should know about the selection sort and bubble sort individually.

What is selection sort?

Sorting means arranging the elements of an array in ascending order. Selection sort is one sorting technique used for sorting the array. In selection sort, an array is divided into two sub- arrays, i.e., one is an unsorted sub-array, and the other is sorted sub-array. Initially, we assume that the sorted subarray is empty. First, we will find the minimum element from the unsorted subarray, and we will swap the minimum element with an element which is at the beginning position of the array. This algorithm is named as selection sort because it is selecting the minimum element and then performs swapping.

Let's understand the selection sort through an example.

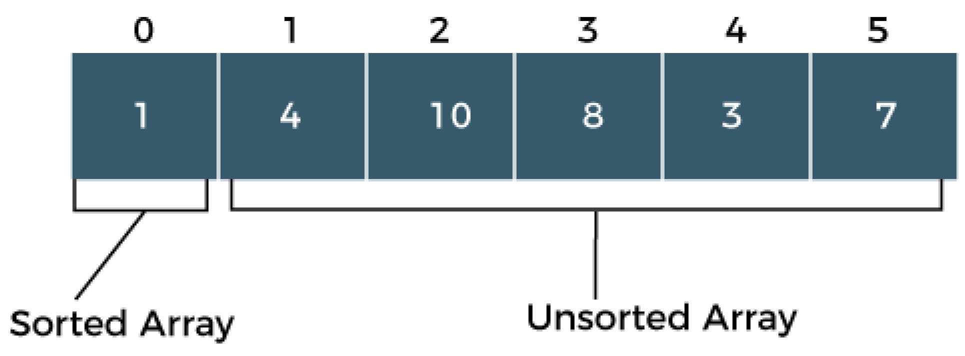


As we can observe in the above array that it contains 6 elements. The above array is an unsorted array whose indexing starts from 0 and ends at 5. The following are the steps used to sort the array:

Step 1: In the above array, the minimum element is 1; swap element 1 with an element 7.

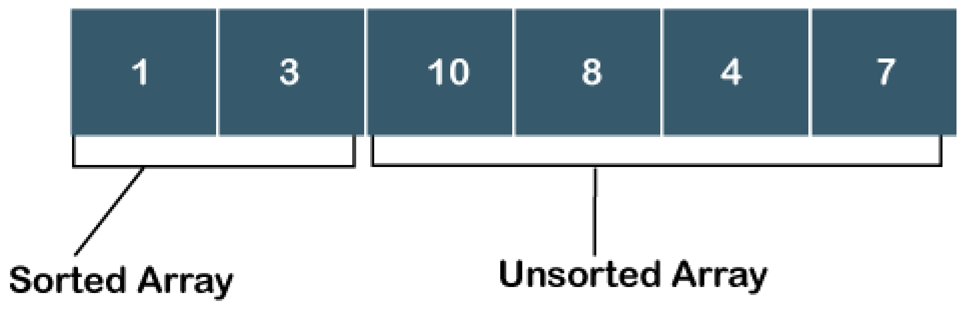
Now, the sorted array contains only one element, i.e., 1, while the unsorted array contains 5

elements, i.e., 4, 10, 8, 3, 7.



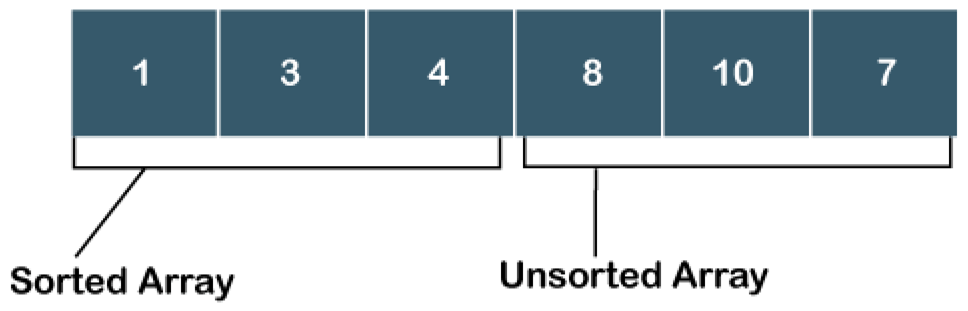
Step 2: In the unsorted sub-array, the minimum element is 3, so swap element 3 with an element 4, which is at the beginning of the unsorted sub-array.

Now the sorted array contains two elements, i.e., 1 and 3, while the unsorted array has four elements, i.e., 10, 8, 4, 7, as shown in the above figure.

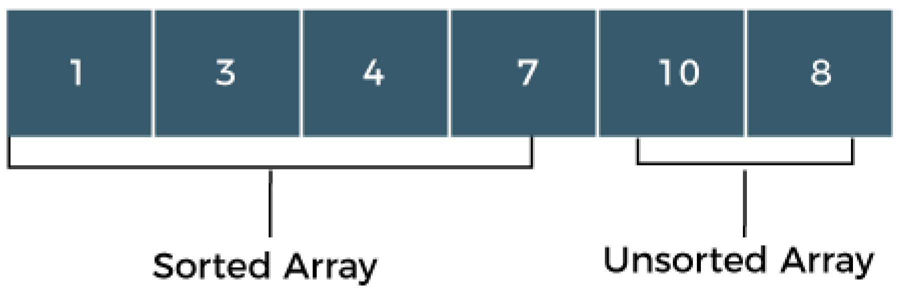


Step 3: Search the minimum element in the unsorted sub-array, and the minimum element is 4. Swap the element 4 with an element 10, which is at the beginning of the unsorted sub- array.

Now, the sorted array contains three elements, i.e., 1, 3, 4, while the unsorted array has three elements, i.e., 10, 8, 7

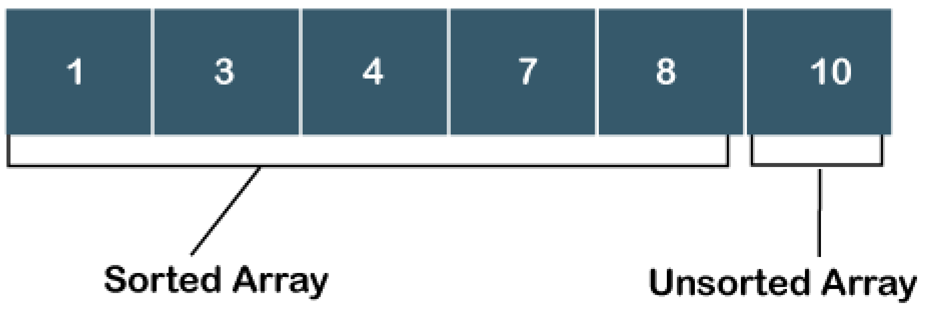


Step 4: Search the minimum element in the unsorted array, and the minimum element is 7. Swap element 7 with an element 10, which is at the beginning of the unsorted sub-array.



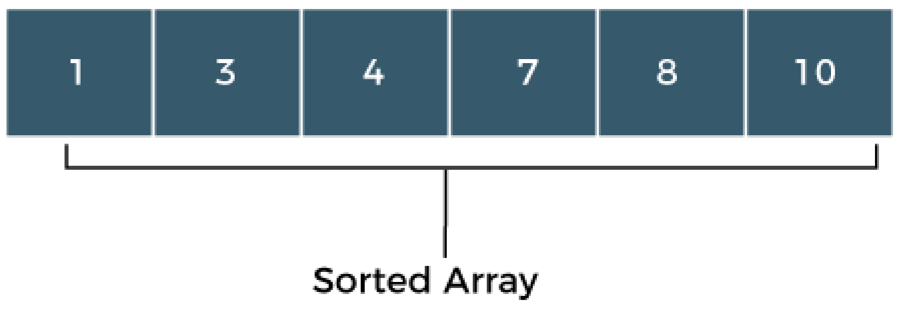
Now, the sorted array contains four elements, i.e., 1, 3, 4, 7, while the unsorted array has two elements, i.e., 10, 8.

Step 5: Search the minimum element in the unsorted array and the minimum element is 8. Swap the element 8 with an element 10 which is at the beginning of the unsorted sub-array.



Now, the sorted array contains the elements, i.e., 1, 3, 4, 7, 8.

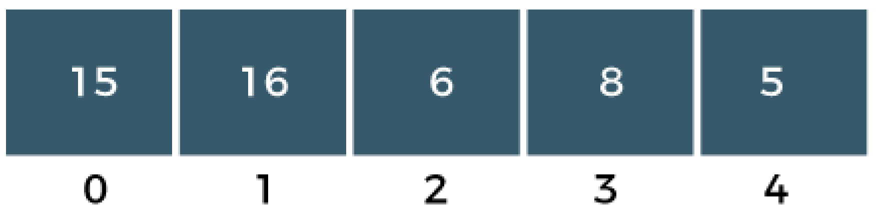
Step 6: The last element is left in the unsorted sub-array. Move the last element to the sorted sub array shown as below:



What is Bubble sort?

The bubble sort is also one of the sorting techniques used for sorting the elements of an array. The basic principle behind the bubble sort is that the two adjacent elements are to be compared; if those elements are in correct order, then we move to the next iteration. Otherwise, we swap those two elements. Let's understand the bubble sort through an example.

Consider the below array:

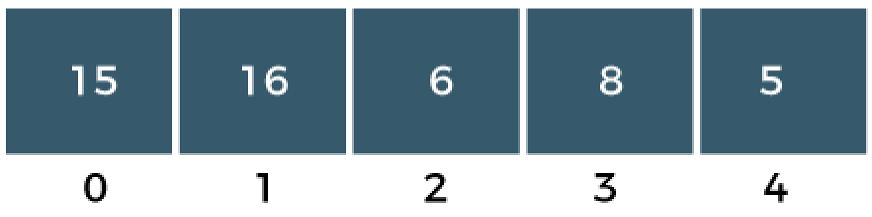


The above array is an unsorted array. An array consists of 5 integers, i.e., 15, 16, 6, 8, 5.

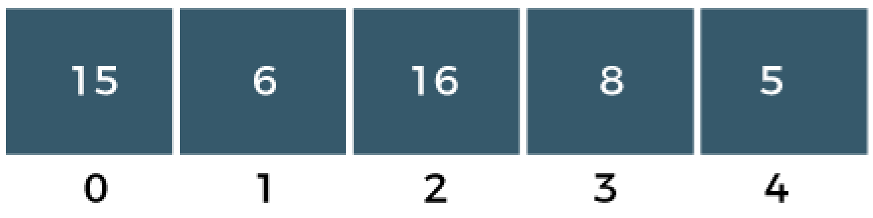
The following are the steps required used to sort the array:

PASS 1

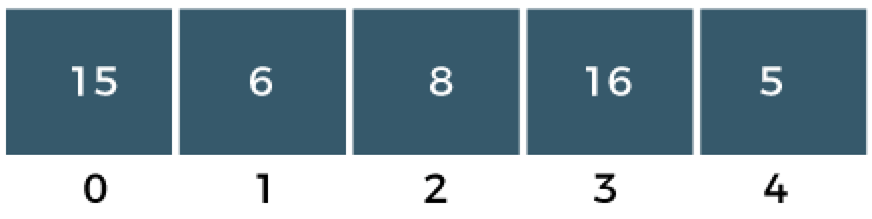
Step 1: The a[0] element is compared with a a[1] element. The a[0] is less than a[1], i.e., 15<16, so no swapping would be done as shown in the below figure:



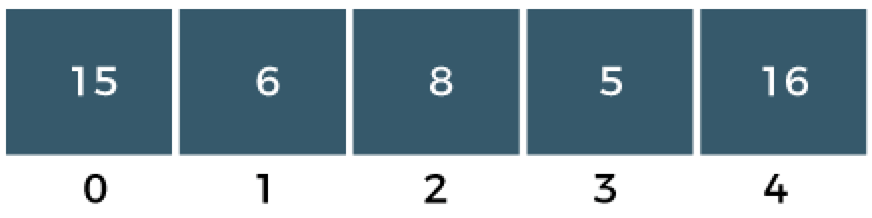
Step 2: Now, a[1] would be compared with a a[2] element. Since a[1] is greater than the a[0] element, i.e., 16>6, so swap 16 and 6 as shown in the below figure:



Step 3: The a[2] would be compared with a a[3] element. Since a[2] is greater than the a[3] element, i.e., 16>8, so swap 16 and 8 elements as shown in the below figure:



Step 4: The a[3] would be compared with a[4] element. Since a[3] is greater than the a[4], i.e., 16 > 5, so swap 16 and 5 elements as shown in the below figure:

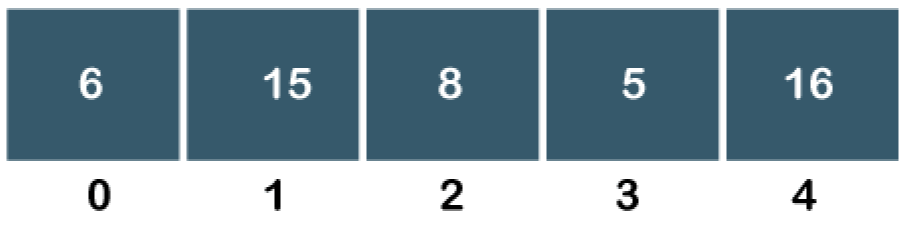


As we can observe in the above array that the element which is the largest has been bubbled up to its correct position. In other words, we can say that the largest element has been placed at the last position of the array. The above steps are included in PASS 1in which the largest element is at its correct position.

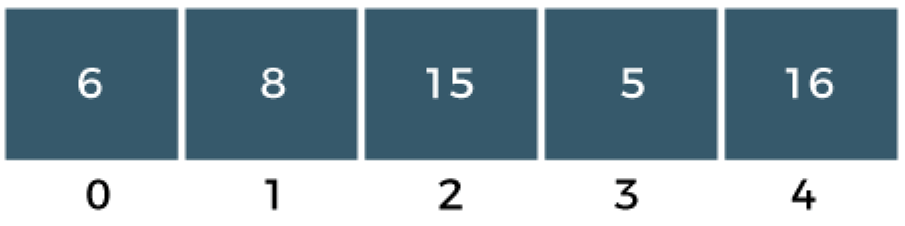
We will again start comparing the elements from the first position in PASS 2.

PASS 2:

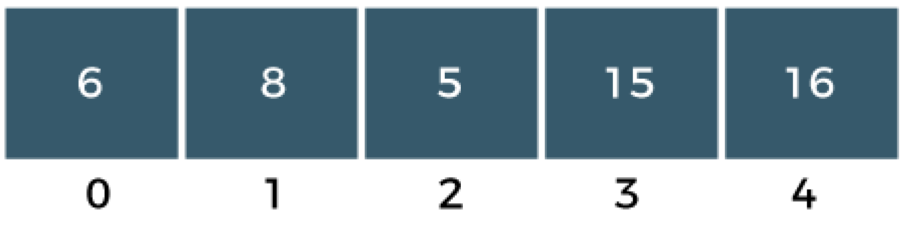
Step 1: First, we compare a[0] with a[1] element. Since a[0] element is greater than the a[1] element, i.e., 15 > 6, swap a[0] element with a[1] as shown in the below figure:



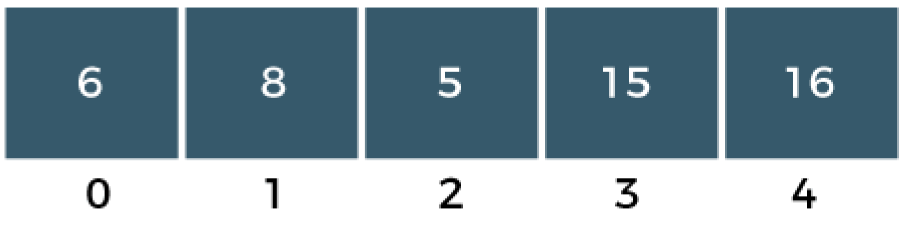
Step 2: The a[1] element would be compared with a[2] element. The a[1] is greater than the a[2], i.e., 15 > 8, so swap a[1] with element a[2] as shown in the below figure:



Step 3: The a[2] element would be compared with a a[3] element. Since a[2] is greater than the a[3] element, i.e., 15 > 5, swap element 15 with element 5 as shown in the below figure:



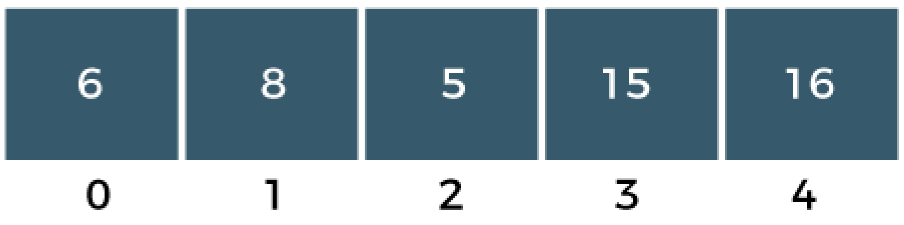
Step 4: Now, a[3] is compared to a[4]. Since a[3] is less than a[4], so no swapping would be done as shown in the below figure:



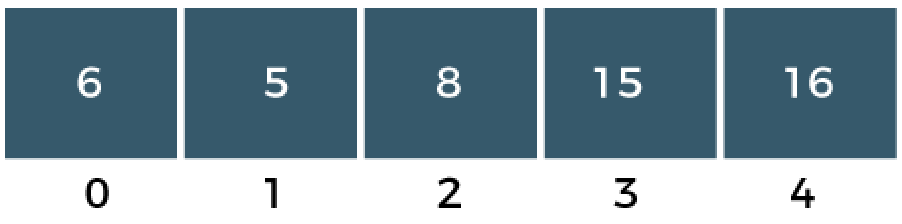
As we can observe above that the two elements are at the right position, largest (16) and the second largest element (15). In an array, three elements are unsorted, so again we will follow the same steps in PASS 3.

PASS 3:

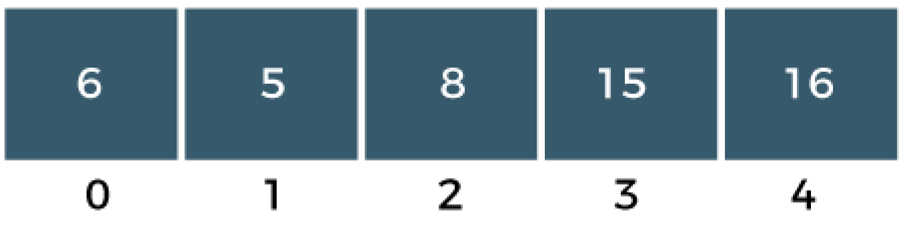
Step 1: First, we compare a[0] with a[1]. Since a[0] is less than a[1], i.e., 6 < 8, so no swapping would be done as shown in the below figure:



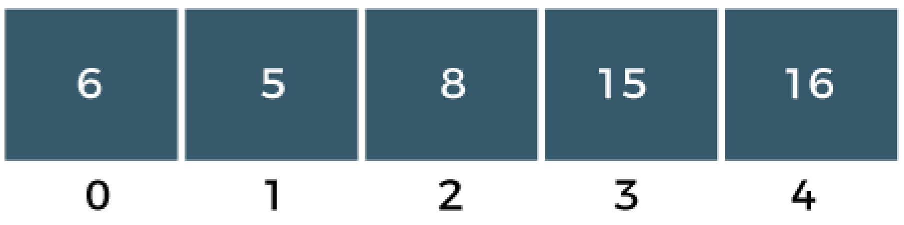
Step 2: Now, a[1] would be compared with a[2]. As a[1] is greater than a[2], so swap the element 8 with the element 5 as shown in the below figure:



Step 3: The a[2] would be compared with a[3]. Since a[2] is less than a[3], i.e., 8 < 15, so no swapping would be done as shown in the below figure:



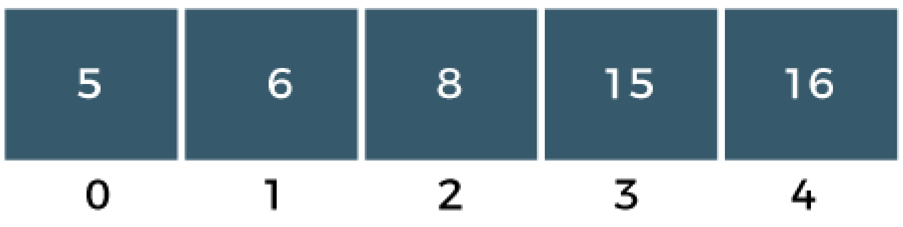
Step 4: The a[3] element would be compared with a[4]. Since a[3] is less than a[4], i.e., 15 < 16, so no swapping would be done as shown in the below figure:



In PASS 3, three elements are at the right positions, largest, second largest and third largest. In an array, two elements are not sorted, so again we will follow the same steps in PASS 4.

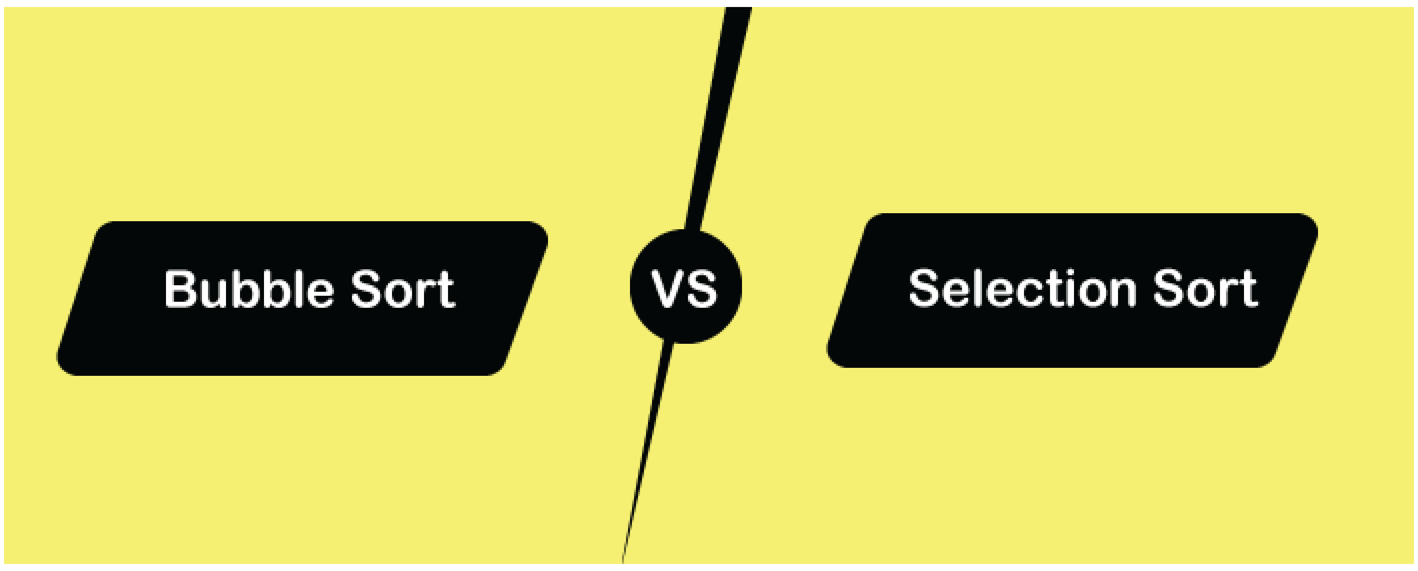
PASS 4:

Step 1: First, we will compare a[0] and a[1]. Swap the a[0] with a[1] as a[0] is greater than a[1].



The above array is a sorted array as all the elements are at the correct positions.

Differences between Bubble sort and Selection sort



Bubble sort	**Selection sort**
In bubble sort, two adjacent elements are compared. If the adjacent elements are not at the correct position, swapping would be performed.	In selection sort, the minimum element is selected from the array and swap with an element which is at the beginning of the unsorted sub array.
The time complexities in best case and worst case are O(n) and O(n2) respectively.	The time complexity in both best and worst cases is O(n 2).
It is not an efficient sorting technique.	It is an efficient sorting technique as compared to Bubble sort.
It uses an exchanging method.	It uses a selection method.
It is slower than the selection sort as a greater number of comparisons is required.	It is faster than the bubble sort as a lesser number of comparisons is required.

Stack vs Array

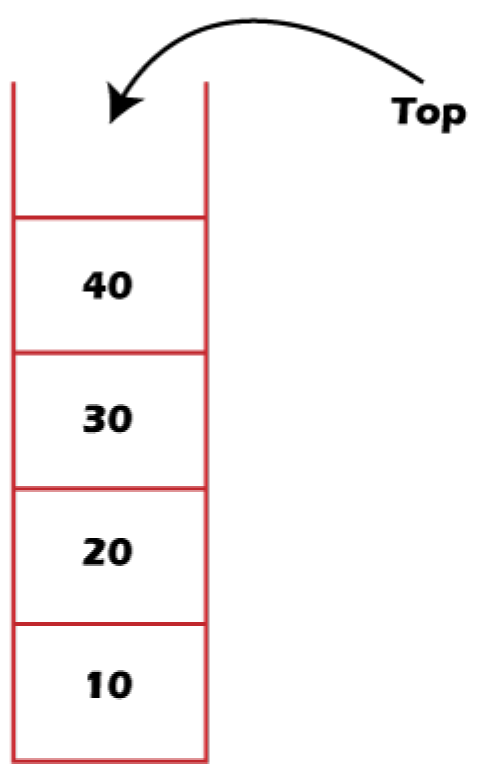
Data structure is a way of organizing the data and storing the data in a prescribed format so that the data can be accessed and modified in an efficient manner. Data structure basically provides the logical representation to store the data so that the various operations can be performed on the data. There are multiple ways of storing and retrieving the data, but the stack and array are the two most common ways of storing the data. Although the stack can be implemented with the help of array but there is a difference between these two. The major difference is access.

Let's first understand both the data structures separately, then we will look at the differences between these two data structures.

What is Stack?

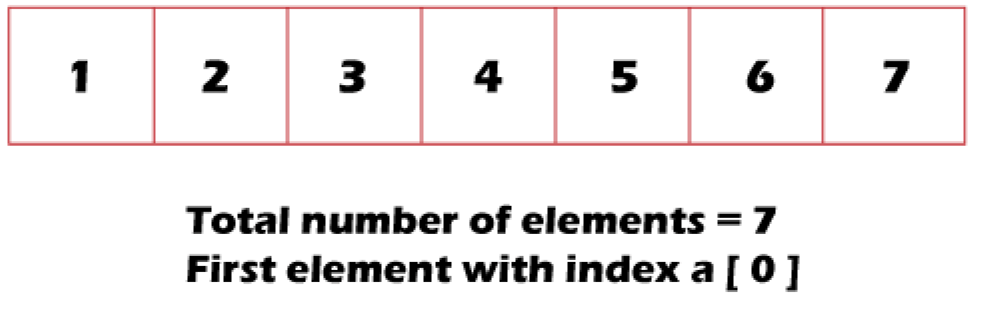
A stack is a linear list data structure having a sequential collection of elements in which elements are added on the top of the stack like piles of books, etc. The new item can be added, or existing item can be removed only from the top of the stack. Stack is considered as a dynamic data structure because the size of the stack can be changed at the run time. The two operations can be performed on the stack, i.e., push and pop. Here, push operation means that a new item is inserted into the stack, whereas the pop operation means that the existing item is removed from the stack. It strictly follows the ***LIFO*** (Last In First Out) principle in which the recently added item will be removed first, and the item which is added first will be removed last.

The representation of the stack is given below:



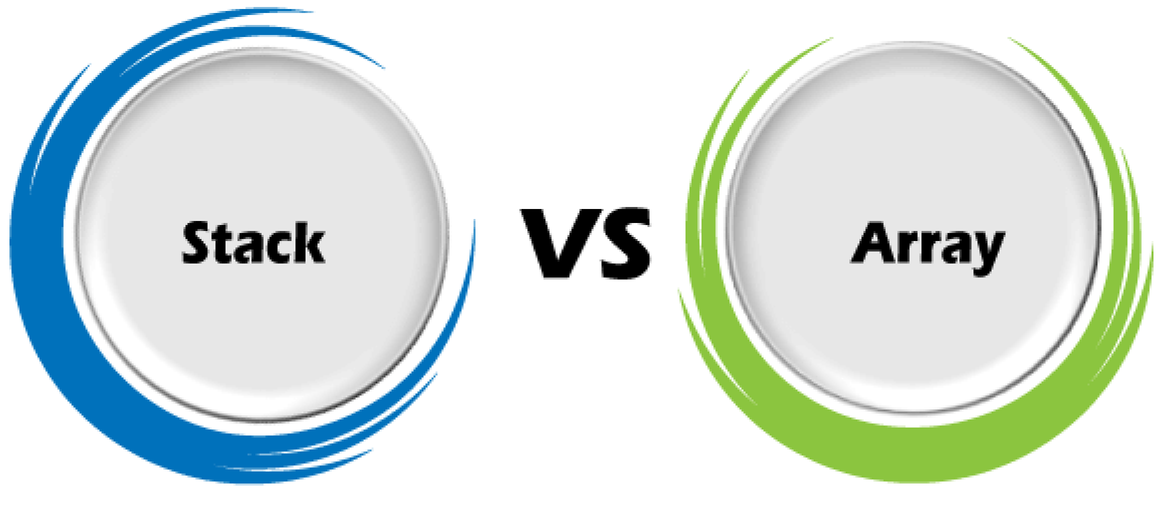
What is Array?

An array is a linear data structure that has the collection of elements of a similar data type. The size of the array is pre-decided and the location at which the values are stored is known as the index of value. It is a static data structure because the memory is allocated at the compile-time, and it is fixed throughout the program. It is one of the efficient ways to store multiple elements belonging to the same data type. It is used to store multiple values of the same type, and we can access them through indices. It provides random access where all the elements are stored linearly and accessed directly through the index.



1
2
3
4
5
6
7
Total number of elements = 7
First element with index a [0]

Differences between Stack and Array



The following are the differences between the stack and array:

- **Definition**
  Stack is a linear data structure that can be defined as a collection of items arranged in the form of piles of books. It is a sequential collection of elements and arranged in such a way that the elements can be added and removed only from one end, i.e., **top of the stack**. In contrast, an array is a random-access data structure used to store the multiple elements of similar data types to reduce the complexity of a program. In array, we can access any element directly through an index, and all the elements are stored one after another for efficient memory management.

- **Data type**
  Stack is an abstract data type which is a sequential collection of objects that can store heterogeneous data. Here, heterogeneous data means that the data of various types can be stored in the stack. It is a limited-access data structure means that the element can be inserted or removed in a specific order. In other words, we can say that only top element of the stack can be accessed. In contrast, Array is a linear data structure that store homogeneous data. Homogeneous data means that data of a similar type can only be stored in the array. In array, access is not limited as we can access any element through indices.
- **Working Principle**
  Stack is a linear data structure that follows **LIFO** (Last In First Out). The name LIFO itself suggests that the element which is added last will be removed first, or in other words, we can say that the element which is inserted first will be removed last. In contrast, in an array, any element can be accessed at any time irrespective of the order of elements.
- **Size**
  The stack is a dynamic data structure means that size of the stack can grow or shrink at run time. In contrast, the size of the array is fixed, and it cannot be modified at run time.

Let's look at the differences between the Stack and Array in a tabular form.

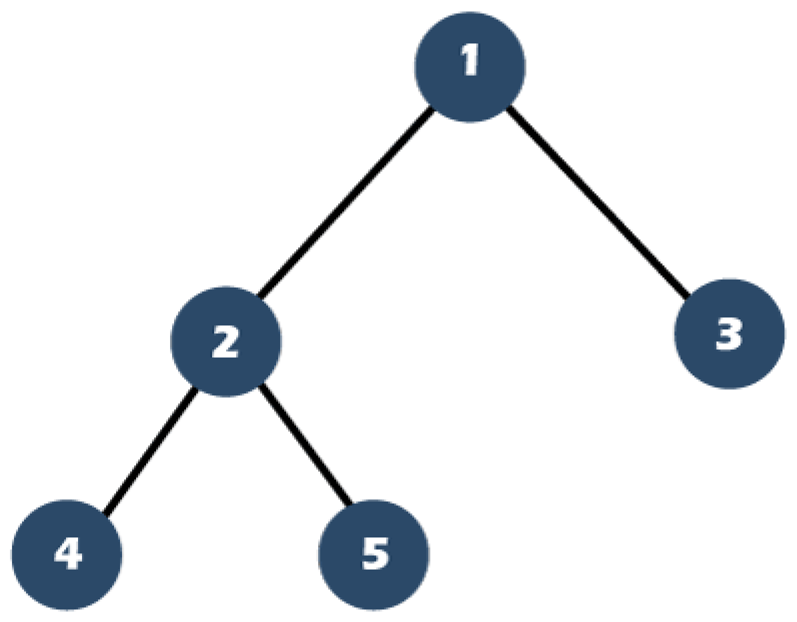
Array	Stack
It is a data structure that consists of a collection of elements that are identified by their indexes, where the first index is available at index 0.	It is an abstract data type that consists of a collection of elements that implements the LIFO (Last In First Out) principle.
It is a collection of elements of the same data type.	It is a collection of elements of different data types.
It provides random-access, i.e., read and write operations would be performed to any element at any position through their indexes.	As it implements LIFO so it has limited-access. We can access the only top element of the stack using push and pop operations.
It is rich in methods or operations like sorting, traversing, reverse, push, pop, etc.	The limited operations can be performed on a stack like push, pop, peek, etc.
It is a data type. It is an abstract data type.	
It is used when we know all the data to be processed and require constant changes at any element.	It is good to use when there are dynamic processes. It is useful when we do not know how much data would be required.

Full Binary Tree vs. Complete Binary Tree

What is a Full binary tree?

A full binary tree can be defined as a binary tree in which all the nodes have 0 or two children. In other words, the full binary tree can be defined as a binary tree in which all the nodes have two children except the leaf nodes.

The below tree is a full binary tree:



The above tree is a full binary tree as all the nodes except the leaf nodes have two children.

Full Binary tree theorem:

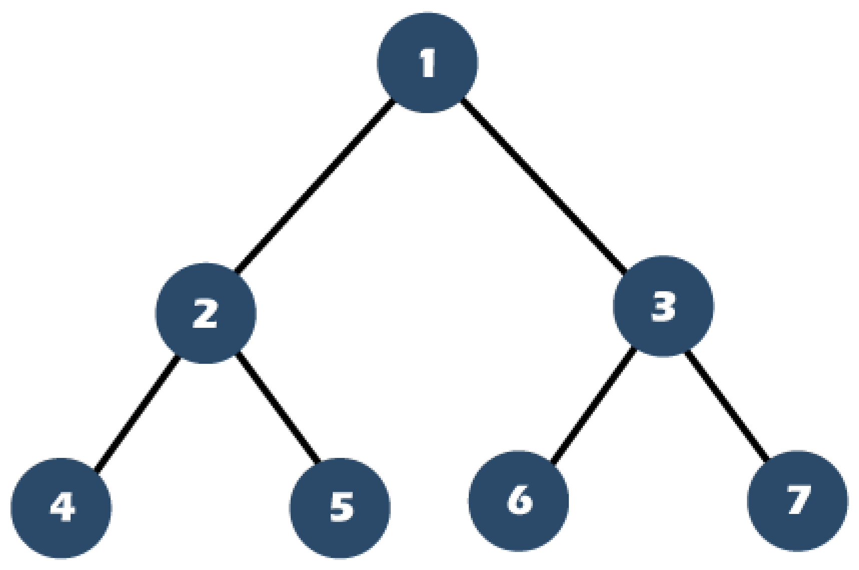
Consider a Binary tree T to be a nonempty tree then:

- Let I be internal nodes in a tree and L to be a leaf node in a tree, then the number of leaf nodes would be equal to: L = I + 1
- If T has, I number of internal nodes and N to be the total number of nodes, then the total number of nodes would be equal to: N = 2I + 1
- If T contains 'N' total number of nodes and 'I' to be number of internal nodes, then the number of internal nodes would be equal to: I = (N-1)/2
- If 'T' has 'N' total number of nodes, and 'L' to be a number of leaf nodes, then the number of leaf nodes would be equal to: L = (N+1)/2
- If 'T' contains 'L' number of leaf nodes, then the total number of nodes would be equal to: N = 2L - 1
- If 'T' has 'L' number of leaf nodes, and 'I' to be a number of internal nodes, then the number of internal nodes would be equal to: I = L - 1

What is a complete binary tree?

A binary tree is said to be a complete binary tree when all the levels are completely filled except the last level, which is filled from the left.

The below tree is a complete binary tree:

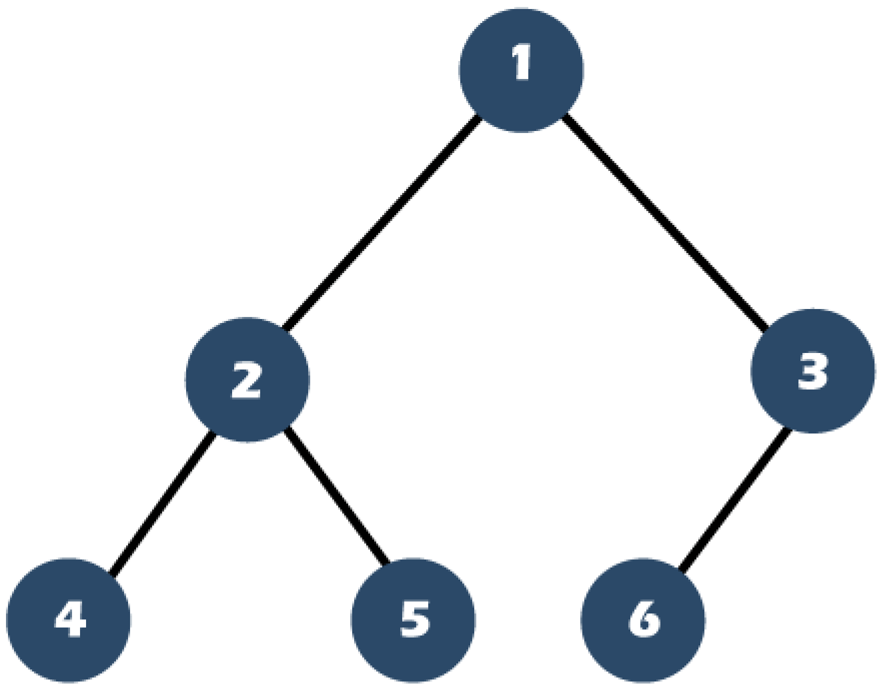


The complete binary tree is similar to the full binary tree except for the two differences which are given below:

- The filling of the leaf node must start from the leftmost side.
- It is not mandatory that the last leaf node must have the right sibling.

Let's understand the above points through an example:

Consider the below tree:



The above tree is a complete binary tree, but not a full binary tree as node 6 does not have its right sibling.

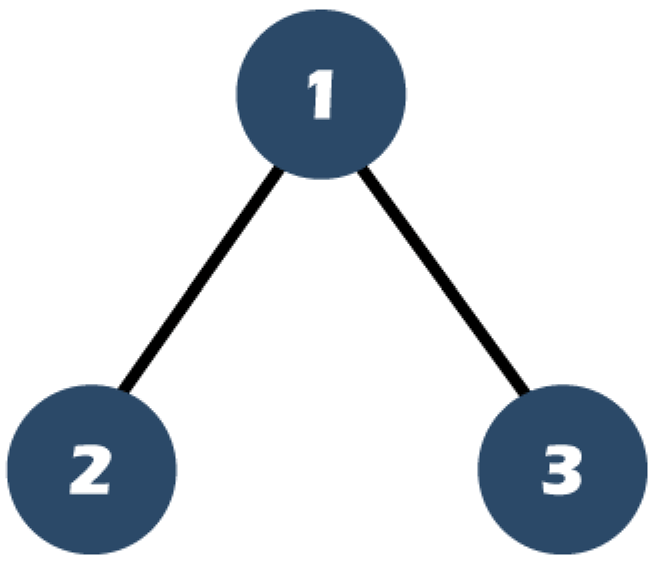
Creation of Complete Binary Tree

Suppose we have an array of 6 elements shown as below:

The above array contains 6 elements, i.e., 1, 2, 3, 4, 5, 6. The following are the steps to be used to create a complete binary tree:

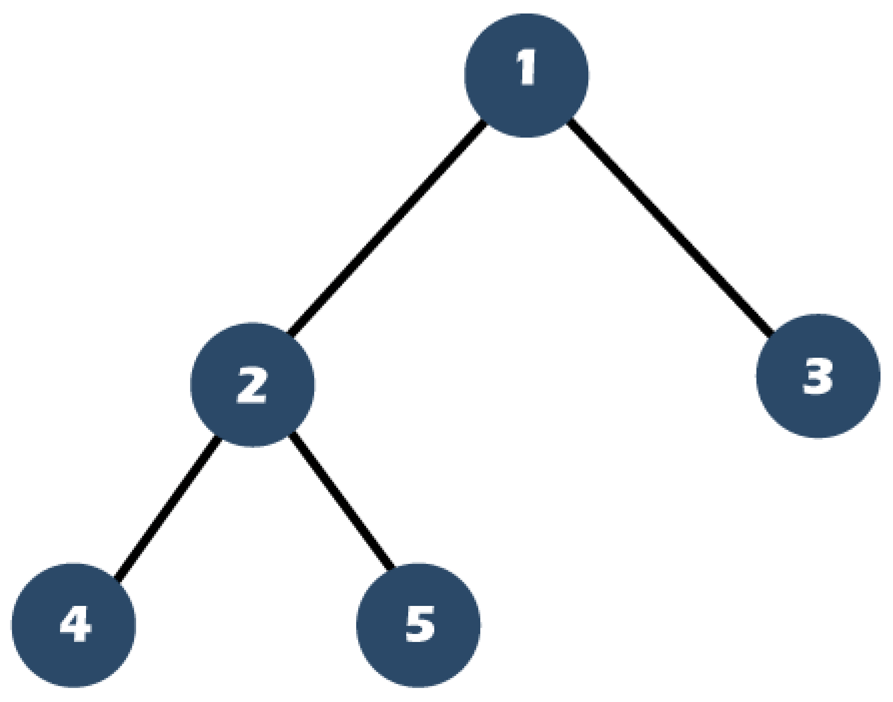
Step 1: First, we will select the first element of the array, i.e., 1, and make a root node of the tree. The number of elements available in the first level is 1.

Step 2: Now, we will select the second and third elements of the array. Keep the second element and third element of the array as the left and right child of the root node respectively shown as below:



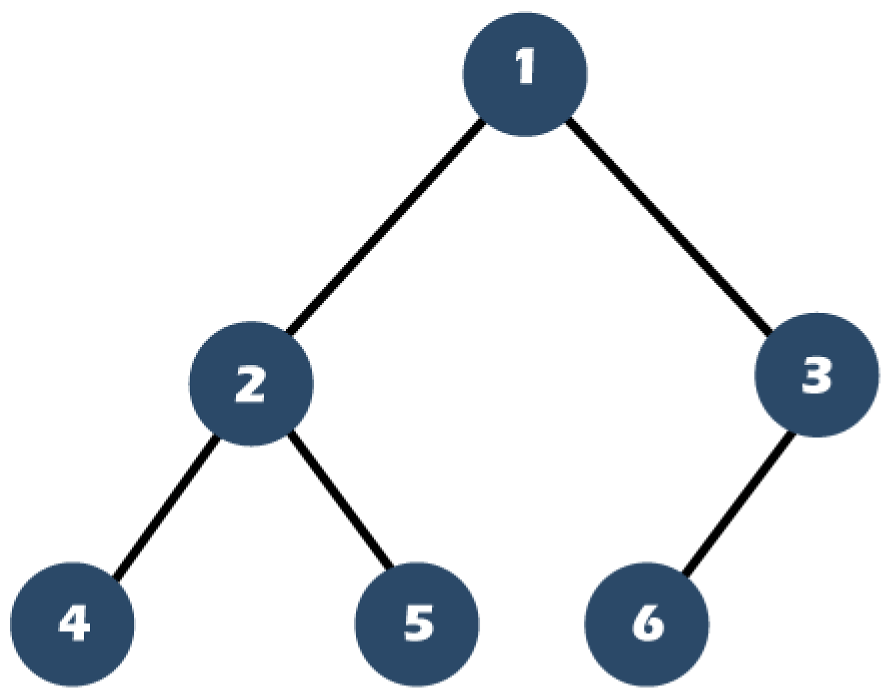
As we can observe above that the number of elements available in the second level is 2.

Step 3: Now, we will select the next two elements from the array, i.e., 4 and 5. Keep these two elements on the left and right of node 2 shown as below:



As we can observe above that nodes 4 and 5 are the left and right child of node 2 respectively.

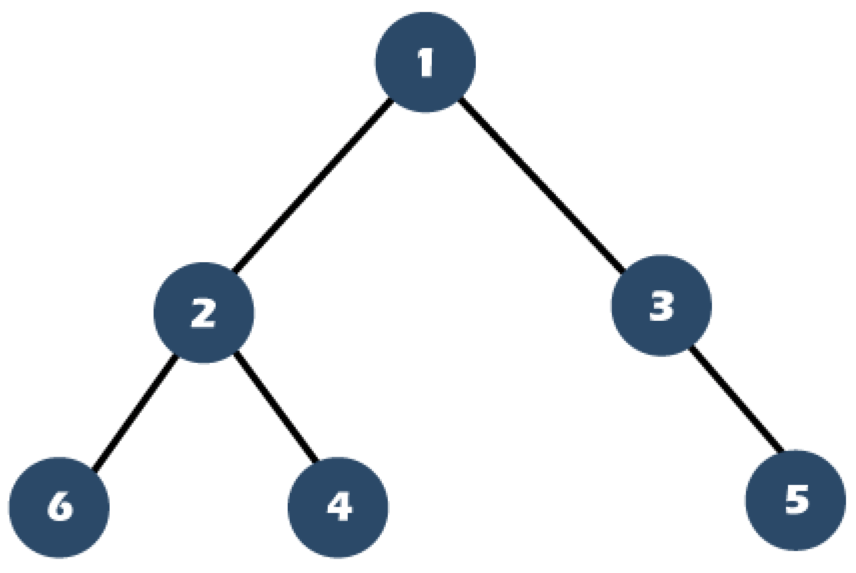
Step 4: Now, we will select the last element of the array, i.e., 6, and keep it as left child of the node 3 as we know that in a complete binary tree, the nodes are filled from the left side shown as below:



As we can observe that the second level contains 3 elements.

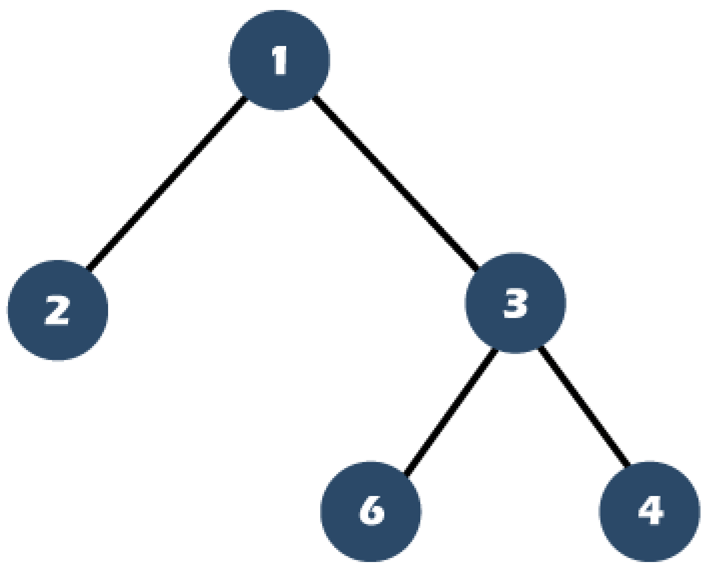
Let's understand the differences between complete and full binary tree through the images.

1. The binary tree which is shown below is neither a complete nor a full binary tree. It is not a full binary tree because node 3 has only one child. It is also not a complete binary tree as the nodes should be filled from the left side, but node 3 has a right child and does not have a left child.

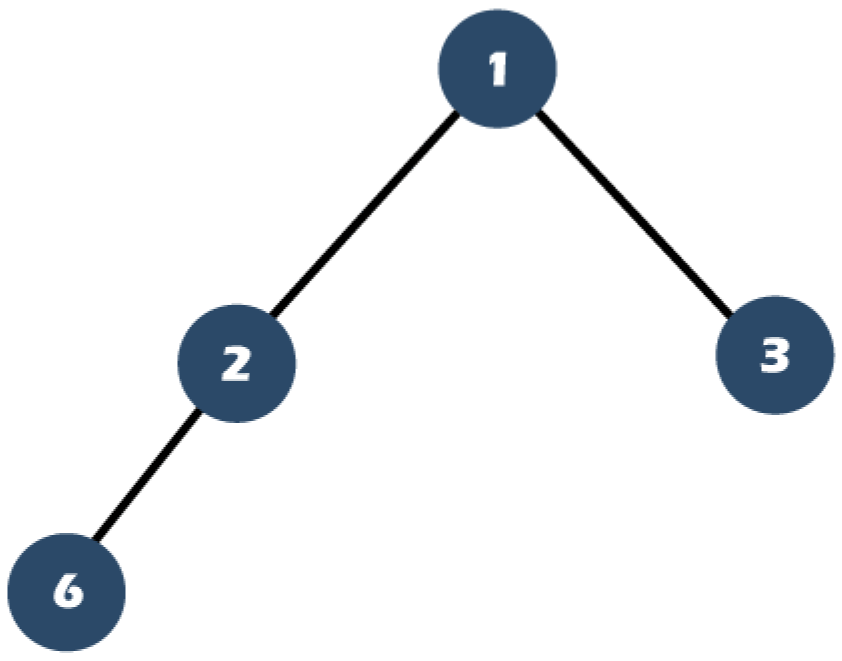


2. The binary tree, which is shown below, is a full binary tree but not a complete binary tree. It is a full binary tree because all the nodes have either 0 or 2 children. It is not a complete binary tree because node 3 does not have any children while node 2 has its children and we know that the nodes should be filled from the left side in a complete

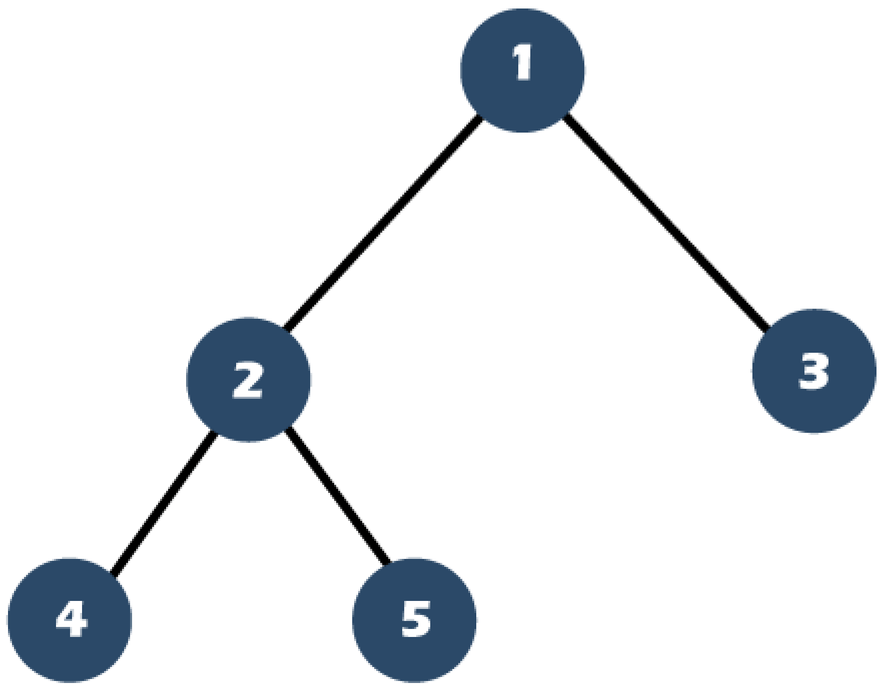
binary tree.



3. The binary tree which is shown below is a complete binary tree but not a full binary tree. It is a complete binary tree as all the nodes are left filled. It is not a full binary tree as node 2 has only one child.



4. The binary tree which is shown below is a complete as well as a full binary tree. It is a complete binary tree as all the nodes are left filled. It is a full binary tree as all the nodes have either 0 or 2 children.



Binary Tree vs B Tree

Before understanding the differences between the binary tree and btree, we should know about **binary tree** and **btree** separately.

What is Binary tree?

A binary tree is a tree that contains at most two child nodes. The binary tree has one limitation on the degree of the node as node in a binary tree cannot contain more than two child nodes. The topmost node in a binary tree is known as a root node and node mainly consists of two sub trees known as left subtree and right subtree. If the node in a binary tree does not contain any children, then it is known as a leaf node. Thus, node can have either 0, 1 or 2 children. The operations that can be performed on a binary tree are insertion, deletion, and traversal.

Binary tree can be categorized into following types:

- Full binary tree: Each node in a binary tree can have zero or two child nodes.
- Perfect binary tree: A perfect binary tree is a full binary tree except one condition that all the leaf nodes exist at the same level of depth.
- Complete binary tree: A complete binary tree is a tree in which all the leaf nodes are left aligned as possible.
- Balanced binary tree: A binary tree is said to be balanced if the height of the tree is as small as possible.
- Binary search tree: A binary search tree is a tree in which all the keys are sorted to provide the faster search.

It can be used to implement the expression evaluation, parsers, data compression algorithms, storing router-tables, cryptographic applications, etc.

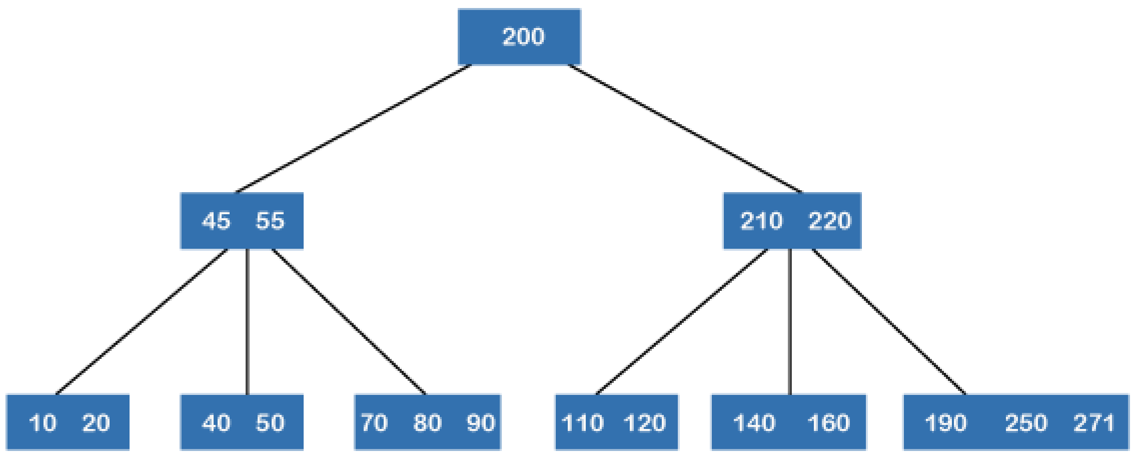
What is BTree?

A btree is a self-balancing tree because its nodes are sorted in an inorder traversal. In contrast to binary tree, node in a btree can have more than two children. The height of btree is logMN where M is the order of tree and N is the number of nodes. The height of a btree adjusts automatically, and the height in a btree is sorted in a specific order having the lowest value on the left and the highest value on the right.

Btree is mainly used to store huge amount of data that cannot fit in the main memory. When the number of keys increases then the data is read from the disk in the form of blocks. As we know that disk access time is more than the memory access time. The main idea behind using the btree is to reduce the number of disk accesses. Most of the operations that are implemented on btree like search, delete, insert, max, min, etc have O(h) disk accesses where h is the height of the tree. Btree is a very wide tree. The idea behind constructing the btree by keeping the height of the tree as low as possible by attaching the maximum number of keys in a btree node. The size of the btree node is mainly made equal to the disk block size. Since the height of the tree is quite low so disk accesses are reduced significantly as compared to the balanced binary search tree like AVL tree, Red Black tree, etc.

Some important facts related to btree are given below:

- All the leaf nodes in a btree must be at the same level.
- There should not be exists any empty subtree above the leaf nodes.
- The height of the btree should be kept as low as possible.



In the above figure, we can observe that all the leaf nodes exist at the same level, and all the non-leaf nodes are non-empty sub trees having keys one less than the number of their children.

Let's understand the differences between binary tree and btree in a tabular form.

B-tree	Binary tree
A node in b tree can have maximum "M' number of child nodes where M is the order of the tree.	Unlike btree, binary tree can have maximum two children or two subtrees.
A btree is a sorted tree because its nodes are sorted in an inorder traversal.	A Binary tree is not a sorted tree A tree can be sorted either in inorder, preorder or postorder traversal.
The height of btree is logMN where M is the order of tree and N is the number of nodes.	The height of binary tree is log2N where N is the number of nodes.
Btree is implemented when the data is stored in the disk.	Binary tree is implemented when the data is stored in RAM.
It is used in DBMS.	It is used in Huffman coding and code optimization.
Insertion of data or a key in a btree is more complex than the binary tree.	Insertion of data in a binary tree is easier as compared to btree.