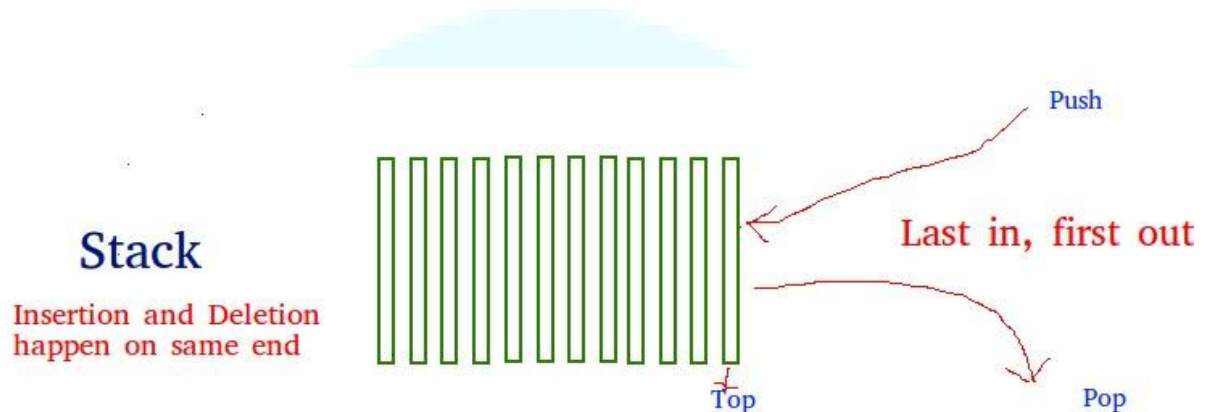


DATA STRUCTURE –PART 2

- **Stack-array and linked implementation. Application- evaluation and conversion of expressions.**
- **Queue – array and linked implementation – circular array queue, priority queue.**
- **Non-linear data structures – tree-basic definition, binary tree- array and linked representation, tree traversal (recursive and non recursive) threaded binary tree, binary search tree, AVL trees, B-trees, Red-black trees, decision and game trees.**
- **Searching – binary & sequential, sort, bubble, heap, insertion & selection.**
- **Representation of graphs – BFS & DFS algorithm Minimum cost spanning tree.**

STACK

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO (Last In First Out) or FILO (First In Last Out).



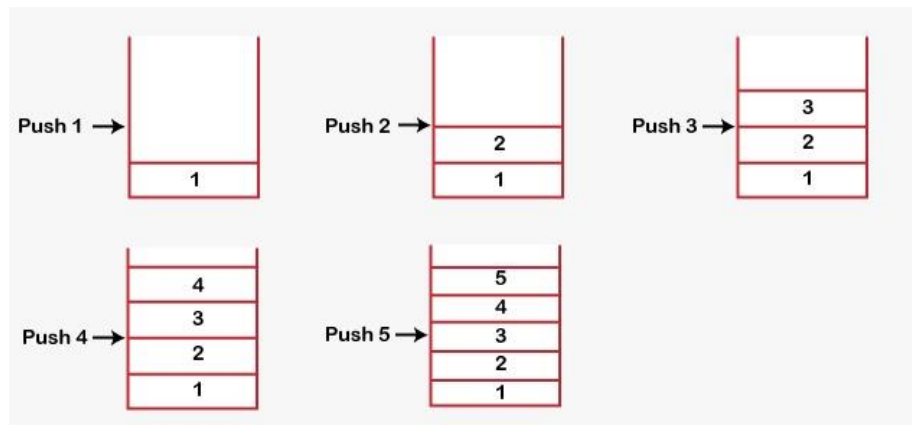
- There are many real-life examples of a stack.
- Consider an example of plates stacked over one another in the canteen.
- The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottom most position remains in the stack for the longest period of time.
- So, it can be simply seen to follow LIFO (Last In First Out)/FILO (First In Last Out) order.
- A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.
- Stack has one end, whereas the Queue has two ends (front and rear).
- It contains only one pointer top pointer pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
- In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

- Stack works on the LIFO pattern.
- As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty.
- We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



- Since our stack is full as the size of the stack is 5.
- In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack.
- The stack gets filled up from the bottom to the top.
- When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed.
- It follows the LIFO pattern, which means that the value entered first will be removed last.
- In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

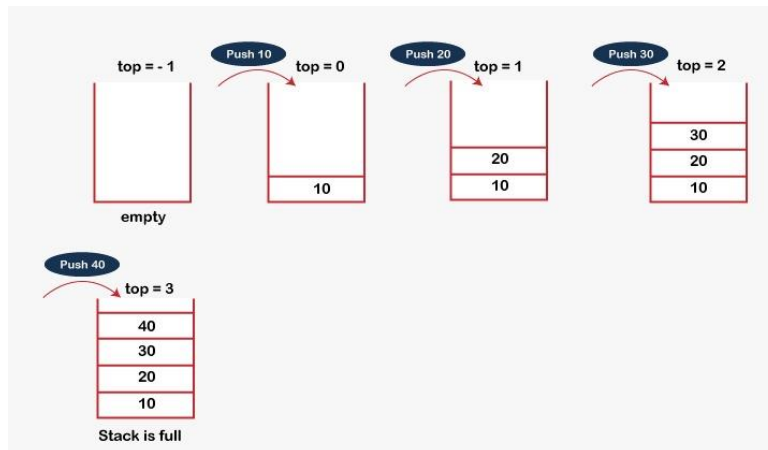
Standard Stack Operations

- The following are some common operations implemented on the stack:
- ✓ **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
 - ✓ **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
 - ✓ **isEmpty():** It determines whether the stack is empty or not.
 - ✓ **isFull():** It determines whether the stack is full or not.'
 - ✓ **peek():** It returns the element at the given position.
 - ✓ **count():** It returns the total number of elements available in a stack.
 - ✓ **change():** It changes the element at the given position.
 - ✓ **display():** It prints all the elements available in the stack.

PUSH operation

• The steps involved in the **PUSH** operation is given below:

- ✓ Before inserting an element in a stack, we check whether the stack is full.
- ✓ If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
- ✓ When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- ✓ When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., $top = top + 1$, and the element will be placed at the new position of the top.
- ✓ The elements will be inserted until we reach the max size of the stack.

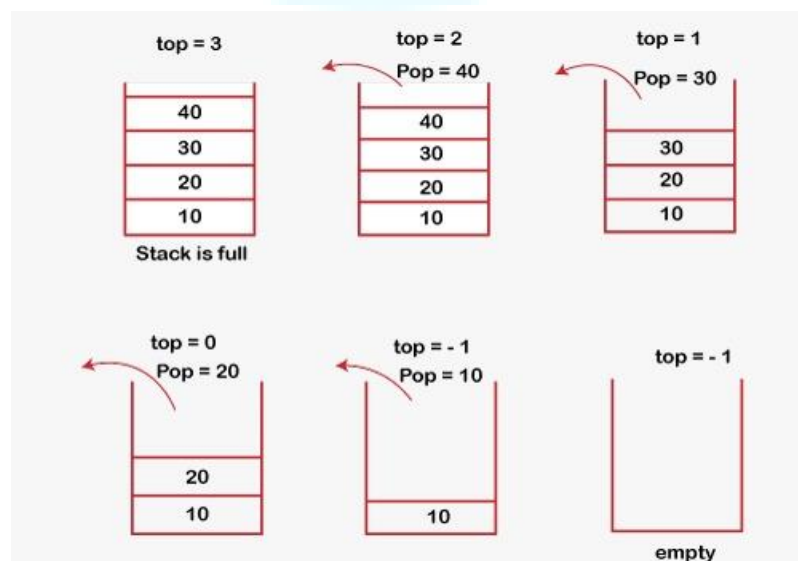


POP operation

• The steps involved in the **POP** operation is given below:

- ✓ Before deleting the element from the stack, we check whether the stack is empty.
- ✓ If we try to delete the element from the empty stack, then the underflow condition occurs.
- ✓ If the stack is not empty, we first access the element which is pointed by the top
- ✓ Once the pop operation is performed, the top is decremented by 1, i.e., $top = top - 1$.

Applications of



Stack

- The following are the applications of the stack:
- Balancing of symbols: Stack is used for balancing a symbol.
- For example, we have the following program:

```
int main()
{
    cout<<"Hello";
    cout<<"javaTpoint";
}
```

- As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack.
- Therefore, the net value comes out to be zero.
- If any symbol is left in the stack, it means that some syntax occurs in a program.

✓ **String reversal:**

- Stack is also used for reversing a string.
- For example, we want to reverse a "javaTpoint" string, so we can achieve this with the help of a stack.
- First, we push all the characters of the string in a stack until we reach the null character.
- After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

✓ **UNDO/REDO:**

- It can also be used for performing UNDO/REDO operations.
- For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc.
- So, there are three states, a, ab, and abc, which are stored in a stack.
- There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
- If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

✓ **Recursion:**

- The recursion means that the function is calling itself again.
- To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

✓ **DFS(Depth First Search):**

- This search is implemented on a Graph, and Graph uses the stack data structure.

✓ **Backtracking:**

- Suppose we have to create a path to solve a maze problem.
- If we are moving in a particular path, and we realize that we come on the wrong way.
- In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

✓ **Expression conversion:**

- Stack can also be used for expression conversion.
- This is one of the most important applications of stack.
- The list of the expression conversion is given below:

- **Infix to prefix**
- **Infix to postfix**

- Prefix to infix
- Prefix to postfix
- Postfix to infix

✓ **Memory management:**

- The stack manages the memory.
- The memory is assigned in the contiguous memory blocks.
- The memory is known as stack memory as all the variables are assigned in a function call stack memory.
- The memory size assigned to the program is known to the compiler.
- When the function is created, all its variables are assigned in the stack memory.
- When the function completed its execution, all the variables assigned in the stack are released.

ARRAY IMPLEMENTATION OF STACK

- In array implementation, the stack is formed by using the array.
- All the operations regarding the stack are performed using arrays.
- Let's see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

- Adding an element into the top of the stack is referred to as push operation.

Push operation involves following two steps.

- ✓ Increment the variable Top so that it can now refer to the next memory location.
- ✓ Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.
- Stack is overflow when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
begin
    if top = n then stack full
    top = top + 1
    stack (top) := item;
end
```

✓ **Time Complexity : $O(1)$**

Implementation of push algorithm in C language

```
void push (int val,int n) //n is size of the stack
{
    if (top == n )
        printf("\n Overflow");
    else
    {
        top = top +1;
        stack[top] = val;
    }
}
```

Deletion of an element from a stack (Pop operation)

- Deletion of an element from the top of the stack is called pop operation.

- The value of the variable top will be incremented by 1 whenever an item is deleted from the stack.
- The top most element of the stack is stored in another variable and then the top is decremented by 1.
- The operation returns the deleted value that was stored in another variable as the result.
- The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
  if top = 0 then stack empty;
  item := stack(top);
  top = top - 1;
end;
Time Complexity :  $O(1)$ 
```

Implementation of POP algorithm using C language

```
int pop ()
{
  if(top == -1)
  {
    printf("Underflow");
    return 0;
  }
  else
  {
    return stack[top - - ];
  }
}
```

Visiting each element of the stack (Peek operation)

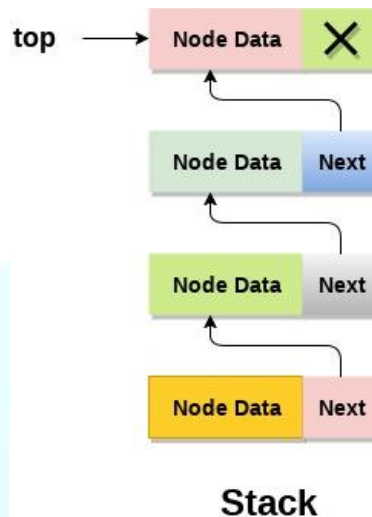
- Peek operation involves returning the element which is present at the top of the stack without deleting it.
- Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

```
PEEK (STACK, TOP)
Begin
  if top = -1 then stack empty
  item = stack[top]
  return item
End
✓ Time complexity:  $O(n)$ 
```

LINKED LIST IMPLEMENTATION OF STACK

- Instead of using array, we can also use linked list to implement stack.
- Linked list allocates the memory dynamically.
- However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.
- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.
- Stack is said to be overflow if the space left in the memory heap is not enough to create a node.

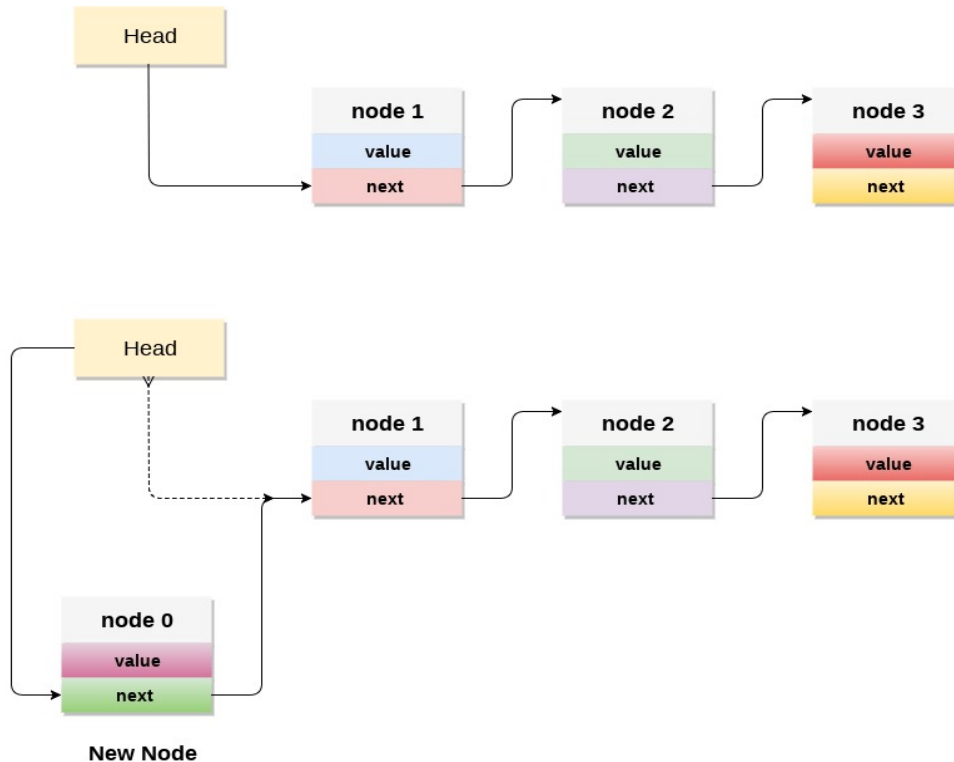


- The top most nodes in the stack always contain null in its address field.
- Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

- Adding a node to the stack is referred to as push operation.
- Pushing an element to a stack in linked list implementation is different from that of an array implementation.
- In order to push an element onto the stack, the following steps are involved.
- ✓ Create a node first and allocate memory to it.
- ✓ If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
- ✓ If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time complexity: $O(n)$



Deleting a node from the stack (POP operation)

- Deleting a node from the top of stack is referred to as pop operation.
- Deleting a node from the linked list implementation of stack is different from that in the array implementation.
- In order to pop an element from the stack, we need to follow the following steps :
 1. Check for the underflow condition: The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
 2. Adjust the head pointer accordingly: In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity: $O(n)$

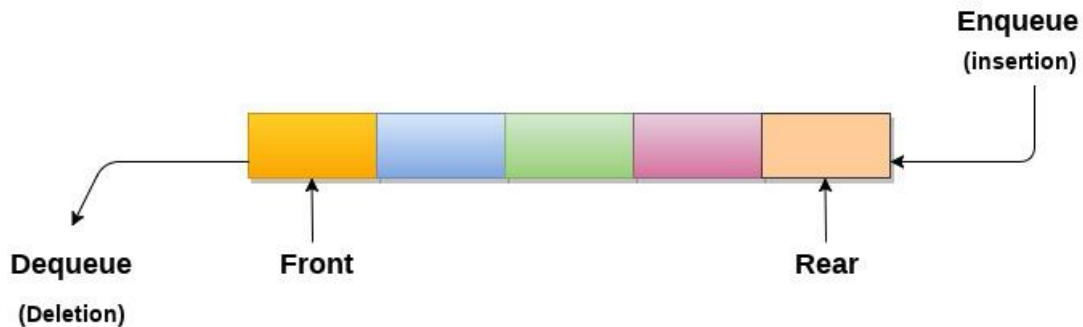
Display the nodes (Traversing)

- Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack.
- For this purpose, we need to follow the following steps.
 1. Copy the head pointer into a temporary pointer.
 2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity: $O(n)$

QUEUE

- A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.
- Queue is referred to be as First In First Out list.
- For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

- Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions.
 - There are various applications of queues discussed as below.
1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
 2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for e.g. Pipes, file IO, sockets.
 3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
 4. Queue is used to maintain the play list in media players in order to add and remove the songs from the play-list.
 5. Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	Time Complexity				Space Complexity				
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

Types of Queue

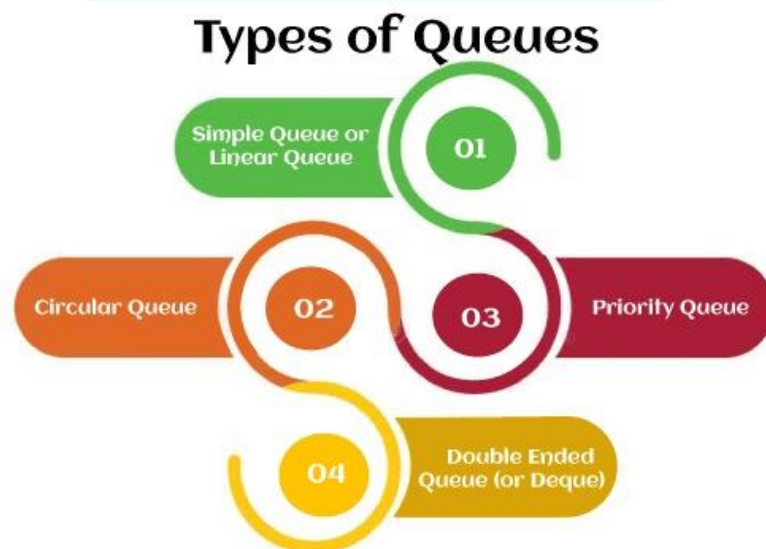
- Queue is the data structure that is similar to the queue in the real world.
- A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy.

- Queue can also be defined as the list or collection in which the insertion is done from one end known as the rear end or the tail of the queue, whereas the deletion is done from another end known as the front end or the head of the queue.
- The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last.
- Similar approach is followed in the queue in data structure.
- The representation of the queue is shown in the below image -



Types of Queue

There are four different types of queue that are listed as follows -



- ✓ Simple Queue or Linear Queue
- ✓ Circular Queue
- ✓ Priority Queue
- ✓ Double Ended Queue (or Deque)

Simple Queue or Linear Queue

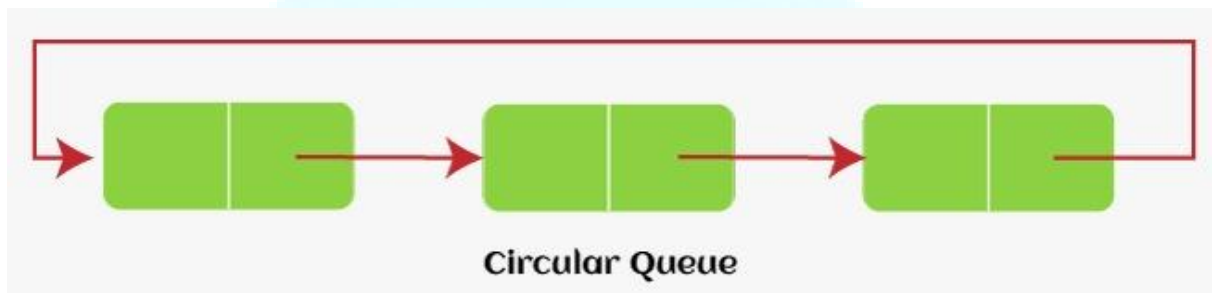
- In Linear Queue, an insertion takes place from one end while the deletion occurs from another end.
- The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end.
- It strictly follows the FIFO rule.



- The major drawback of using a linear Queue is that insertion is done only from the rear end.
- If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue.
- In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Circular Queue

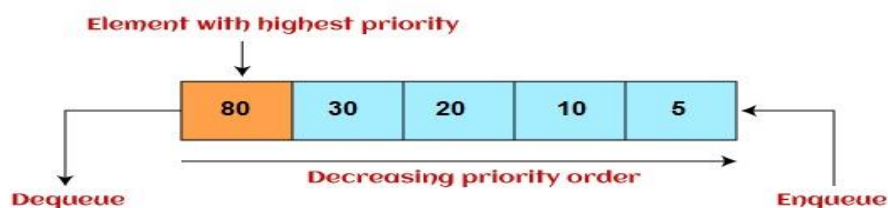
- In Circular Queue, all the nodes are represented as circular.
- It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- It is also known as Ring Buffer, as all the ends are connected to another end.
- The representation of circular queue is shown in the below image -



- The drawback that occurs in a linear queue is overcome by using the circular queue.
- If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.
- The main advantage of using the circular queue is better memory utilization.

Priority Queue

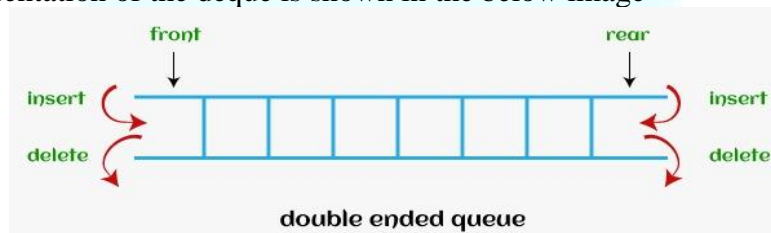
- It is a special type of queue in which the elements are arranged based on the priority.
- It is a special type of queue data structure in which every element has a priority associated with it.
- Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle.
- The representation of priority queue is shown in the below image -



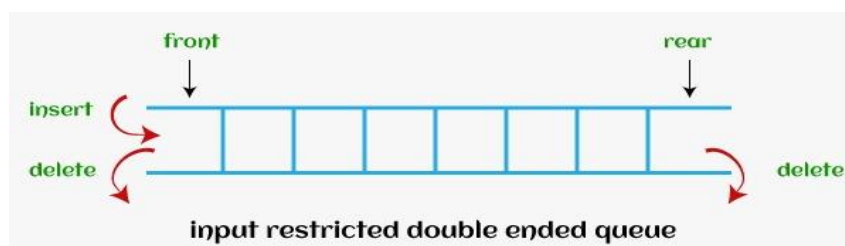
- Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority.
- Priority queue is mainly used to implement the CPU scheduling algorithms.
- There are two types of priority queue that are discussed as follows -
- ✓ **Ascending priority queue** - In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- ✓ **Descending priority queue** - In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or, Double Ended Queue)

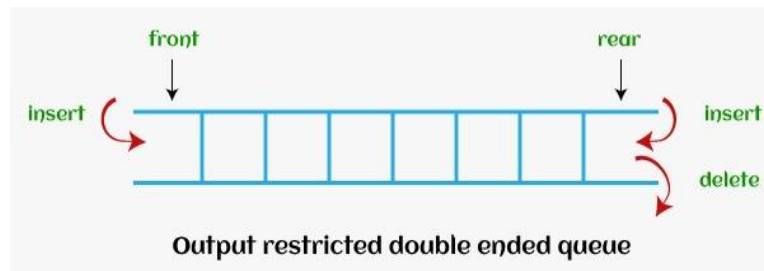
- In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear.
- It means that we can insert and delete elements from both front and rear ends of the queue.
- Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends.
- Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end.
- And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.
- The representation of the deque is shown in the below image -



- There are two types of deque that are discussed as follows -
- ✓ **Input restricted deque** - As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



- ✓ **Output restricted deque** - As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Operations performed on queue

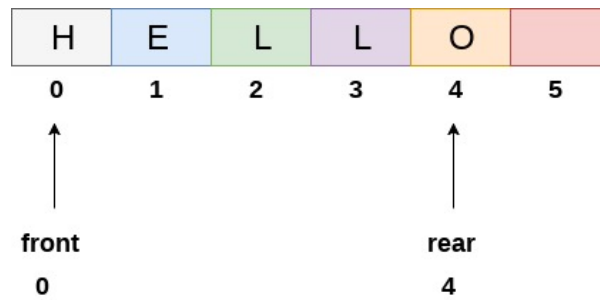
- The fundamental operations that can be performed on queue are listed as follows -
- ✓ **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- ✓ **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- ✓ **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- ✓ **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- ✓ **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Ways to implement the queue

- There are two ways of implementing the Queue:
- ✓ **Implementation using array:** The sequential allocation in a Queue can be implemented using an array.
- ✓ **Implementation using linked list:** The linked list allocation in a Queue can be implemented using a linked list.

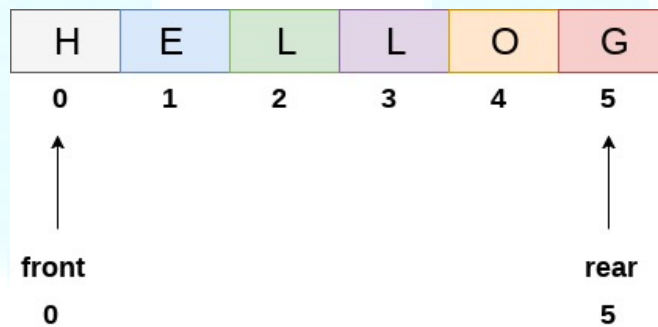
ARRAY REPRESENTATION OF QUEUE

- We can easily represent queue by using linear arrays.
- There are two variables i.e. front and rear that are implemented in the case of every queue.
- Front and rear variables point to the position from where insertions and deletions are performed in a queue.
- Initially, the value of front and queue is -1 which represents an empty queue.
- Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



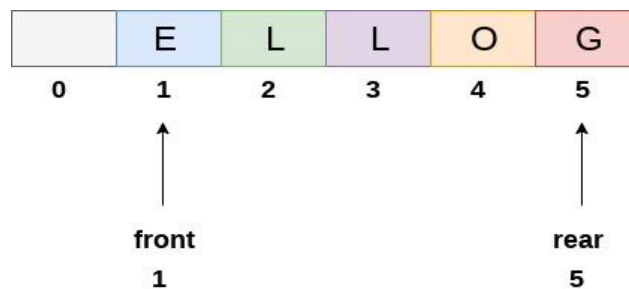
Queue

- The above figure shows the queue of characters forming the English word "HELLO".
- Since, No deletion is performed in the queue till now, therefore the value of front remains -1 .
- However, the value of rear increases by one every time an insertion is performed in the queue.
- After inserting an element into the queue shown in the above figure, the queue will look something like following.
- The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

- After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

- Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
- If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
- Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

Step 1: IF REAR = MAX - 1

Write OVERFLOW

Go to step

[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: Set QUEUE[REAR] = NUM

Step 4: EXIT

Algorithm to delete an element from the queue

- If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
- Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

Step 1: IF FRONT = -1 or FRONT > REAR

Write UNDERFLOW

ELSE

SET VAL = QUEUE[FRONT]

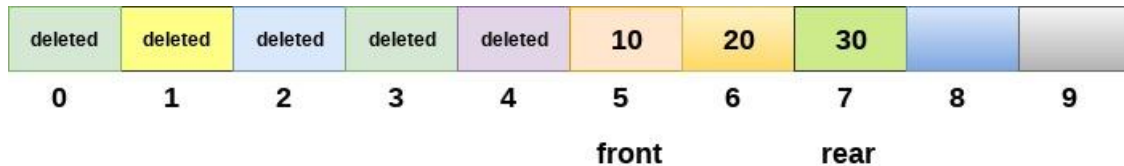
SET FRONT = FRONT + 1

[END OF IF]

Step 2: EXIT

Drawback of array implementation

- Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.
- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.

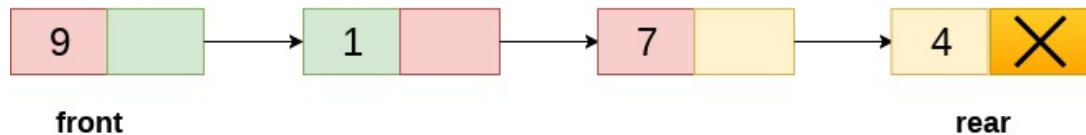


limitation of array representation of queue

- The above figure shows how the memory space is wasted in the array representation of queue.
- In the above figure, a queue of size 10 having 3 elements is shown.
- The value of the front variable is 5, therefore, we cannot reinsert the values in the place of already deleted element before the position of front.
- That much space of the array is wasted and cannot be used in the future (for this queue).
- **Deciding the array size**
- One of the most common problems with array implementation is the size of the array which requires to be declared in advance.
- Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place.
- Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused.
- It will again lead to memory wastage.

LINKED LIST IMPLEMENTATION OF QUEUE

- The array implementation cannot be used for the large scale applications where the queues are implemented.
- One of the alternatives of array implementation is linked list implementation of queue.
- The storage requirement of linked representation of a queue with n elements is $O(n)$ while the time requirement for operations is $O(1)$.
- In a linked queue, each node of the queue consists of two parts i.e. **data part and the link part**.
- Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer.
- The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion and deletions are performed at rear and front end respectively.
- If front and rear both are NULL, it indicates that the queue is empty.
- The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

- There are two basic operations which can be implemented on the linked queues.
- The operations are Insertion and Deletion.

Insert operation

- The insert operation appends the queue by adding an element to the end of the queue.
- The new element will be the last element of the queue.
- Firstly, allocate the memory for the new node ptr by using the following statement.

Ptr = (struct node *) malloc (sizeof(struct node));

- There can be the two scenario of inserting this new node ptr into the linked queue.
- In the first scenario, we insert element into an empty queue.
- In this case, the condition front = NULL becomes true.
- Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```

ptr -> data = item;
if(front == NULL)
{
    front = ptr;
    rear = ptr;
    front -> next = NULL;
    rear -> next = NULL;
}

```

- In the second case, the queue contains more than one element.
- The condition front = NULL becomes false.
- In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr.
- Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node ptr.
- We also need to make the next pointer of rear point to NULL.

```

rear -> next = ptr;
rear = ptr;
rear->next = NULL;

```

- In this way, the element is inserted into the queue. The algorithm is given as follows.

Algorithm

Step 1: Allocate the space for the new node PTR

Step 2: SET PTR -> DATA = VAL

Step 3: IF FRONT = NULL

```
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
Step 4: END
```

Deletion

- Deletion operation removes the element that is first inserted among all the queue elements.
- Firstly, we need to check either the list is empty or not. The condition `front == NULL` becomes true if the list is empty, in this case, we simply write underflow on the console and make exit.
- Otherwise, we will delete the element that is pointed by the pointer front.
- For this purpose, copy the node pointed by the front pointer into the pointer ptr.
- Now, shift the front pointer, point to its next node and free the node pointed by the node ptr.
- This is done by using the following statements.

```
ptr = front;
front = front -> next;
free(ptr);
```

The algorithm is given as follows.

Algorithm

Step 1: IF FRONT = NULL

Write "Underflow"

Go to Step 5

[END OF IF]

Step 2: SET PTR = FRONT

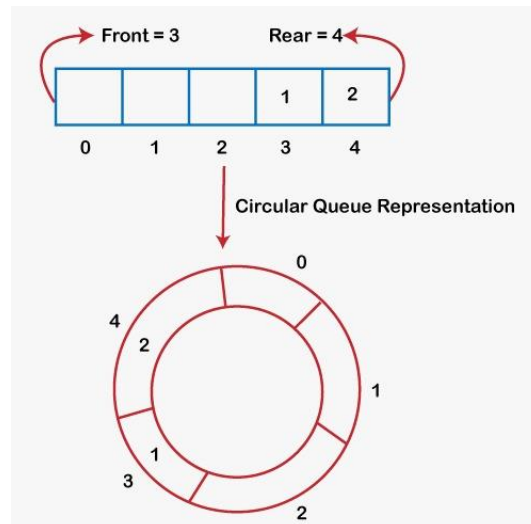
Step 3: SET FRONT = FRONT -> NEXT

Step 4: FREE PTR

Step 5: END

CIRCULAR QUEUE

- There was one limitation in the array implementation of Queue.
- If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized.
- So, to overcome such limitations, the concept of the circular queue was introduced.



- As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position.
- In the above array, there are only two elements and other three positions are empty.
- The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue.
- There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly.
- It is not a practically good approach because shifting all the elements will consume lots of time.
- The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

- A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle.
- It is also known as a Ring Buffer.

Operations on Circular Queue

- The following are the operations that can be performed on a circular queue:
 - ✓ **Front:** It is used to get the front element from the Queue.
 - ✓ **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):**
 - ✓ This function is used to insert the new value in the Queue.
 - ✓ The new element is always inserted from the rear end.
- **deQueue():**
 - ✓ This function deletes an element from the Queue.
 - ✓ The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

- The circular Queue can be used in the following scenarios:
 - ✓ **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But

in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

- ✓ **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- ✓ **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every interval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

- The steps of enqueue operation are given below:
 1. First, we will check whether the Queue is full or not.
 2. Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
 3. When we insert a new element, the rear gets incremented, i.e., $\text{rear} = \text{rear} + 1$.

Scenarios for inserting an element

- There are two scenarios in which queue is not full:
 1. If $\text{rear} \neq \text{max} - 1$, then rear will be incremented to $\text{mod}(\text{maxsize})$ and the new value will be inserted at the rear end of the queue.
 2. If $\text{front} \neq 0$ and $\text{rear} = \text{max} - 1$, it means that queue is not full, then set the value of rear to 0 and insert the new element there.
- There are two cases in which the element cannot be inserted:
 1. When $\text{front} == 0$ && $\text{rear} = \text{max} - 1$, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
 2. $\text{front} == \text{rear} + 1$;

Algorithm to insert an element in a circular queue

Step 1: IF $(\text{REAR} + 1) \% \text{MAX} = \text{FRONT}$

Write " OVERFLOW "

Goto step 4

[End OF IF]

Step 2: IF $\text{FRONT} = -1$ and $\text{REAR} = -1$

SET $\text{FRONT} = \text{REAR} = 0$

ELSE IF $\text{REAR} = \text{MAX} - 1$ and $\text{FRONT} \neq 0$

SET $\text{REAR} = 0$

ELSE

SET $\text{REAR} = (\text{REAR} + 1) \% \text{MAX}$

[END OF IF]

Step 3: SET $\text{QUEUE}[\text{REAR}] = \text{VAL}$

Step 4: EXIT

Deque Operation

- The steps of dequeue operation are given below:

1. First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
2. When the element is deleted, the value of front gets decremented by 1.
3. If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

Step 1: IF FRONT = -1

Write " UNDERFLOW "

Goto Step 4

[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR

SET FRONT = REAR = -1

ELSE

IF FRONT = MAX -1

SET FRONT = 0

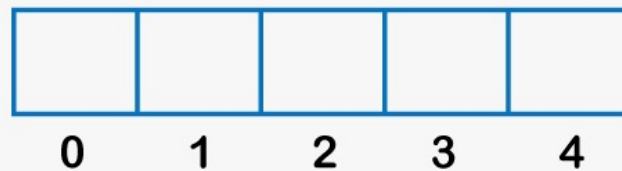
ELSE

SET FRONT = FRONT + 1

[END of IF]

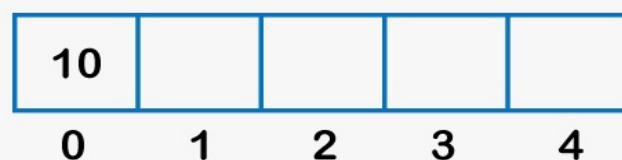
[END OF IF]

Step 4: EXIT



Front = -1

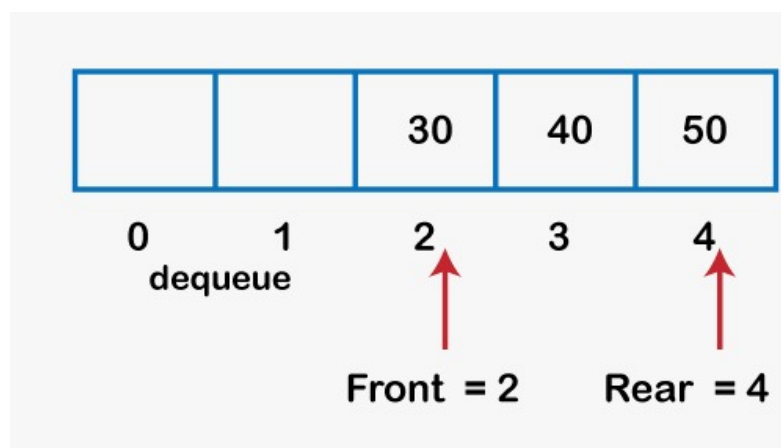
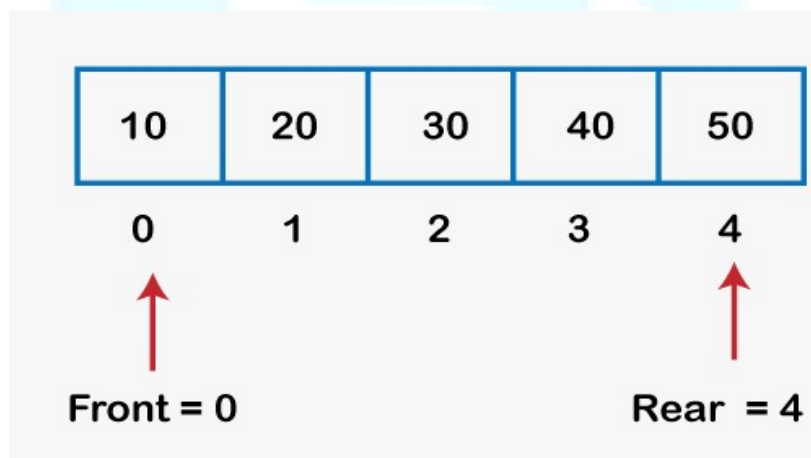
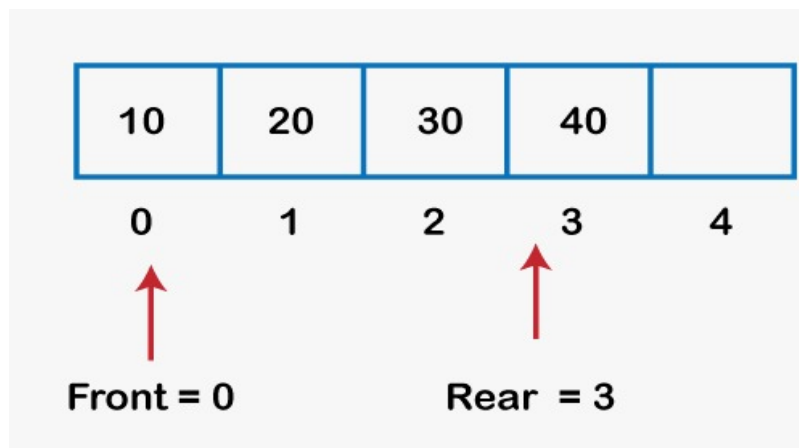
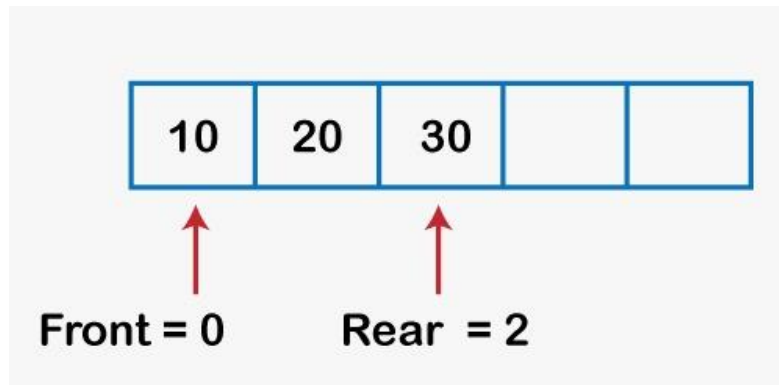
Rear = -1

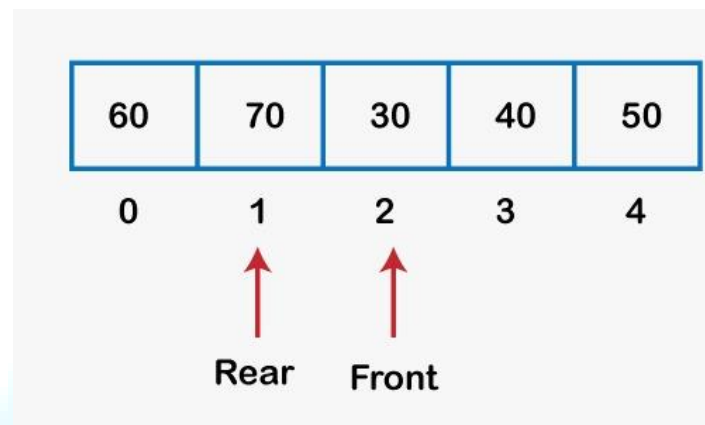
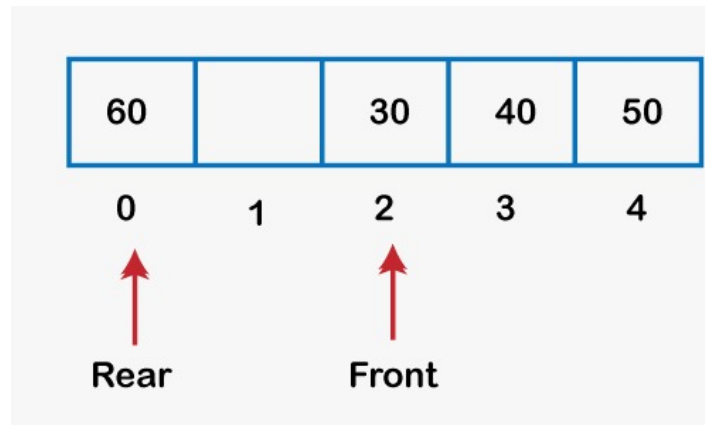


Front = 0

Rear = 0

E ▶ ENTRI





PRIORITY QUEUE

- A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue.
- The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.
- The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.
- For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest.
- Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

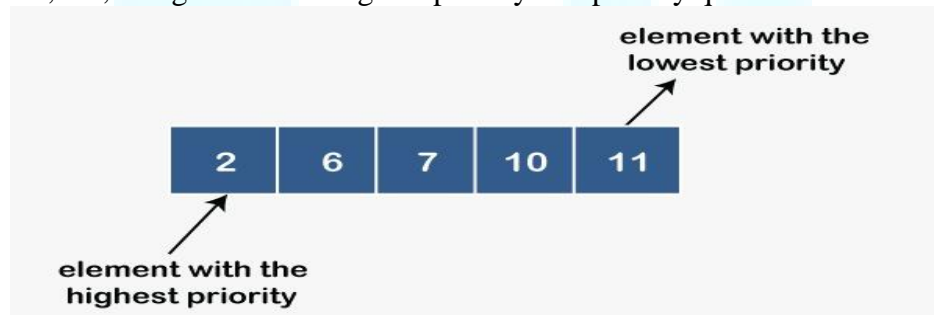
Characteristics of a Priority queue

- A priority queue is an extension of a queue that contains the following characteristics:
 - ✓ Every element in a priority queue has some priority associated with it.
 - ✓ An element with the higher priority will be deleted before the deletion of the lesser priority.
 - ✓ If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.
- Let's understand the priority queue through an example.

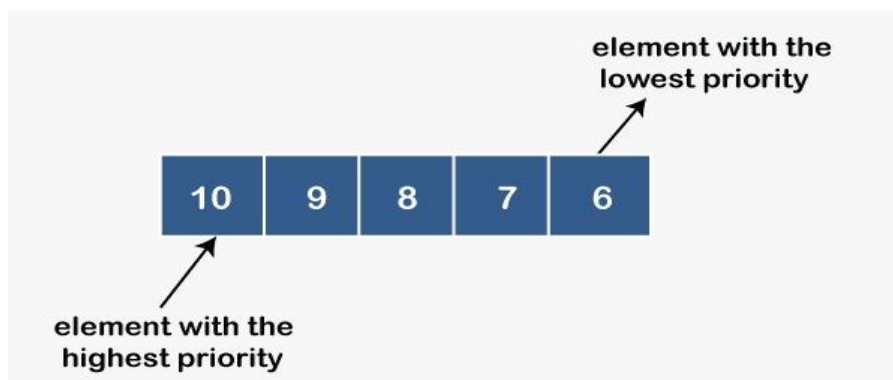
- We have a priority queue that contains the following values:
1, 3, 4, 8, 14, 22
 - All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:
- ✓ **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
 - ✓ **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
 - ✓ **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
 - ✓ **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

- There are two types of priority queue:
- ✓ **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- ✓ **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.

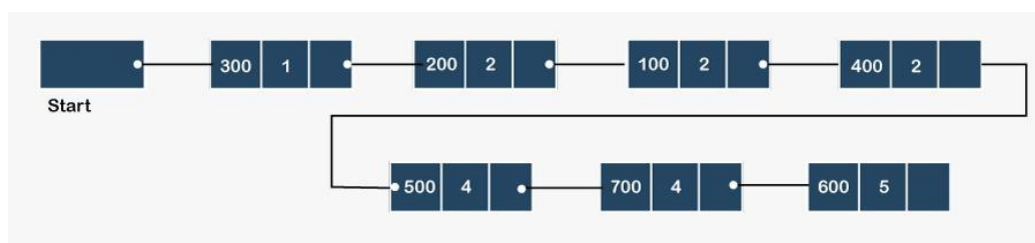


Representation of priority queue

- Now, we will see how to represent the priority queue through a one-way list.
- We will create the priority queue by using the list given below in which INFO list contains the data elements, PRN list contains the priority numbers of each data element available in the INFO list, and LINK basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

- Let's create the priority queue step by step.
 - In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.
- ✓ **Step 1:** In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:
- ✓ **Step 2:** After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.
- ✓ **Step 3:** After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.
- ✓ **Step 4:** After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

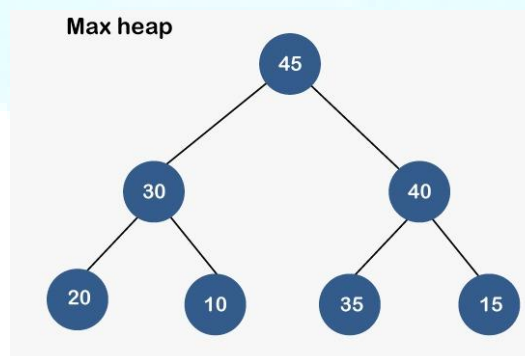
- The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree.
- The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic.
- Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Analysis of complexities using different implementations

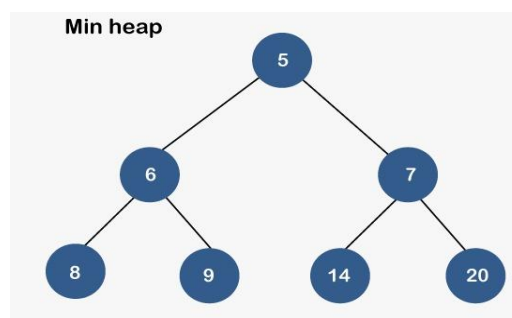
Implementation	add	Remove	peek
Linked list	$O(1)$	$O(n)$	$O(n)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(1)$
Binary search tree	$O(\log n)$	$O(\log n)$	$O(1)$

HEAP

- A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property.
- If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap.
- It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node.
- Therefore, we can say that there are two types of heaps:
- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



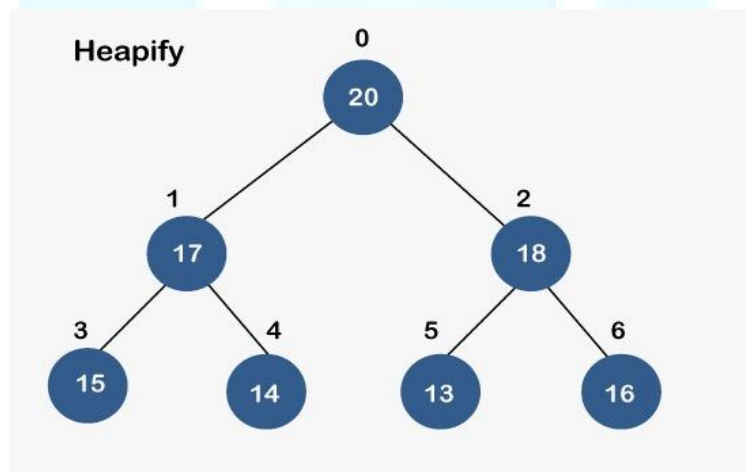
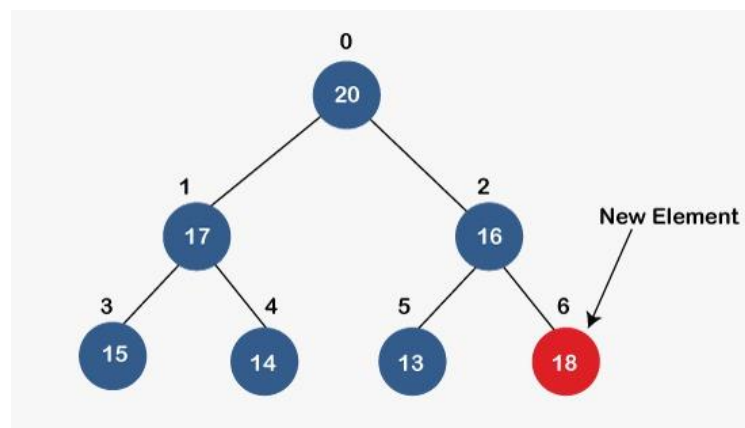
- Both the heaps are the binary heap, as each has exactly two child nodes.

Priority Queue Operations

- The common operations that we can perform on a priority queue are insertion, deletion and peek.

✓ Inserting the element in a priority queue (max heap)

- If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.
- If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped.
- This process continues until the element is placed in a correct position.



Removing the minimum element from the priority queue

- As we know that in a max heap, the maximum element is the root node.
- When we remove the root node, it creates an empty slot.
- The last inserted element will be added in this empty slot.
- Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two.
- It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

- **The following are the applications of the priority queue:**
- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

NON-LINEAR DATA STRUCTURE

- A non-linear data structure is another important type in which data elements are not arranged sequentially; mainly, data elements are arranged in random order without forming a linear structure.
- Data elements are present at the multilevel, for example, tree.
- In trees, the data elements are arranged in the hierarchical form, whereas in graphs, the data elements are arranged in random order, using the edges and vertex.
- Multiple runs are required to traverse through all the elements completely.
- Traversing in a single run is impossible to traverse the whole data structure.
- Each element can have multiple paths to reach another element.
- The data structure where data items are not organized sequentially is called a non-linear data structure. In other words, data elements of the non-linear data structure could be connected to more than one element to reflect a special relationship among them.

Types:

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

TREE

- The tree is a non-linear data structure that is comprised of various nodes.
- The nodes in the tree data structure are arranged in hierarchical order.
- It consists of a root node corresponding to its various child nodes, present at the next level.
- The tree grows on a level basis, and root nodes have limited child nodes depending on the order of the tree.
- For example, in the binary tree, the order of the root node is 2, which means it can have at most 2 children per node, not more than it.
- The non-linear data structure cannot be implemented directly, and it is implemented using the linear data structure like an array and linked list.
- The tree itself is a very broad data structure and is divided into various categories like **Binary tree, Binary search tree, AVL trees, Heap, max Heap, min-heap**, etc.
- All the types of trees mentioned above differ based on their properties.

GRAPH

- A graph is a non-linear data structure with a finite number of vertices and edges, and these edges are used to connect the vertices.
- The graph itself is categorized based on some properties; if we talk about a complete graph, it consists of the vertex set, and each vertex is connected to the other vertexes having an edge between them.

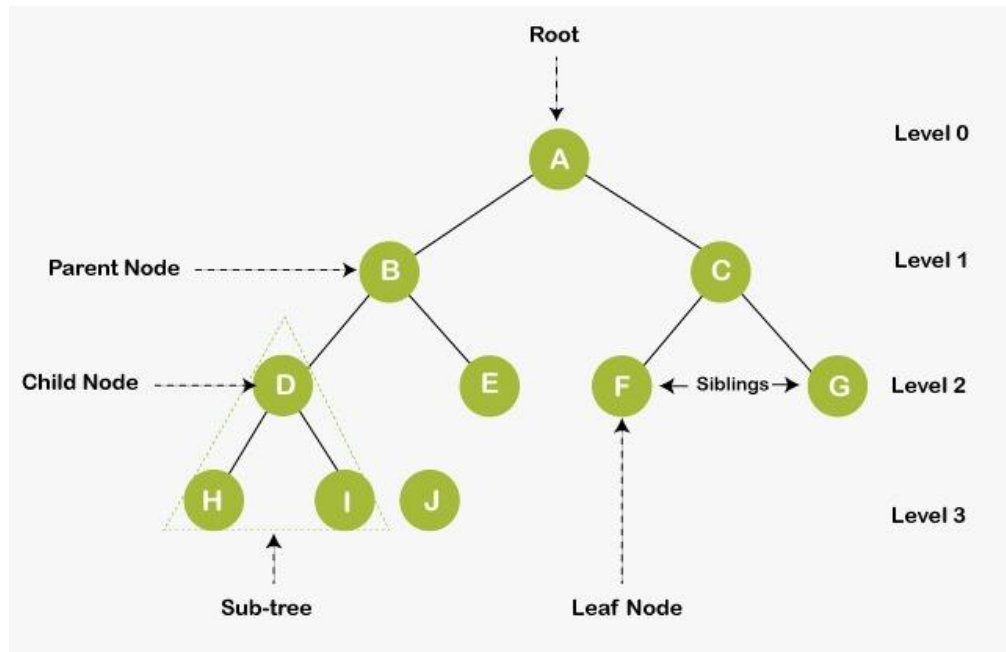
- The vertices store the data elements, while the edges represent the relationship between the vertices.
- A graph is very important in various fields; the network system is represented using the graph theory and its principles in computer networks.
- Even in Maps, we consider every location a vertex, and the path derived between two locations is considered edges.
- The graph representation's main motive is to find the minimum distance between two vertexes via a minimum edge weight.

Properties of Non-linear data structures

- It is used to store the data elements combined whenever they are not present in the contiguous memory locations.
- It is an efficient way of organizing and properly holding the data.
- It reduces the wastage of memory space by providing sufficient memory to every data element.
- Unlike in an array, we have to define the size of the array, and subsequent memory space is allocated to that array; if we don't want to store the elements till the range of the array, then the remaining memory gets wasted.
- So to overcome this factor, we will use the non-linear data structure and have multiple options to traverse from one node to another.
- Data is stored randomly in memory.
- It is comparatively difficult to implement.
- Multiple levels are involved.
- Memory utilization is effective.

TREE DATA STRUCTURE

- Trees: Unlike Arrays, Stack, Linked Lists, and queues, which are linear data structures, trees are hierarchical.
- A tree data structure is a collection of objects or entities known as nodes linked together to represent or simulate hierarchy.
- This data is not arranged in a sequential contiguous location like as we have observed in an array, the homogeneous data elements are placed at the contiguous memory location so that the retrieval of data elements is simpler.
- A tree data structure is non-linear because it does not store sequentially.
- It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- The topmost node in the Tree data structure is known as a root node.
- Each node contains some data, and data can be of any type.
- The node contains the employee's name in the tree structure, so the data type would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.



Terminologies used in tree:

- ✓ **Root node:** The tree consists of a root node; it is the starting node of the tree from which any tree grows. The root node is present at level 0 in any tree. Depending on the order of the tree, it can hold that much of child nodes. For example, if in a tree the order is 3, then it can have at most three child nodes, and minimum, it can have 0 child nodes.
- ✓ **Child node:** The child node is the node that comes after the root node, which has a parent node and has some ancestors. It is the node present at the next level of its parent node. For example, if any node is present at level 5, it is sure that its parent node is present at level 4, just above its level.
- ✓ **Edge:** Edges are nothing but the link between the parent node, and the children node is called the link or edge connectivity between two nodes.
- ✓ **Siblings:** The nodes having the same parent node is called siblings of each other. Same ancestors are not siblings, and only the parent node must be the same, defined as siblings. For example, if node A has two child nodes, B and C, B and C are called siblings
- ✓ **Leaf node:** The leaf node is the node that does not have any child nodes. It is the termination point of any tree. It is also known as External nodes.
- ✓ **Internal nodes:** Internal nodes are non-leaf nodes, having at least one child node with child nodes.
- ✓ **Degree of a node:** The degree of a node is defined as the number of children nodes. The degree of the leaf node is always 0 because it does not have any children, and the internal node always has atleast one degree, as it contains atleast one child node.

- ✓ **Height of the tree:** The tree's height is defined as the distance of the root node to the leaf node present at the last level is called the height of the tree. In other words, height is the maximum level upto which the tree is extended.

Types of trees

- Trees are divided into various categories as follows mentioned below:

- ✓ **Simple Tree**
- ✓ **Binary tree**
- ✓ **Complete Binary tree**
- ✓ **Full Binary tree**
- ✓ **Binary search tree**
- ✓ **AVL Trees**
- ✓ **B -Tree**
- ✓ **B+ Tree**

And many more.

- Let us discuss a few of its types:

1. Simple tree –

- Simple tree is nothing but an N - array tree consisting of a root node and having multiple child nodes.
- The child nodes are present on the next level root node, and in this, the tree grows.
- We do not have any restrictions on the root node's children; we can call it a simple tree.
- Any tree with a hierarchical structure can be described as a general tree.
- A general tree is characterized by the lack of any configuration or limitations on the number of children a node can have.
- A node can have any number of children, and the tree's orientation can be any combination of these.
- The degree of the nodes can range from 0 to n.

2. Binary tree –

- It is a very important subcategory of simple trees.
- As the name suggests, we can easily predict that the binary tree consists of two children only.
- A binary tree comprises nodes that can have two children, as described by the word "binary," which means "two numbers."
- Any node can have a maximum of 0, 1, or 2 nodes in a binary tree.
- Data structures' binary trees are highly functional ADTs that can be subdivided into various types.
- A tree whose elements have at most 2 children is called a binary tree.
- Since each element in a binary tree can have only 2 children, we typically name them the left and right children.
- They are most commonly used in data structures for two reasons:
- For obtaining nodes and categorizing them, as observed in Binary Search Trees.

- For representing data through a bifurcating structure.

3. A Binary Search Tree (BST)

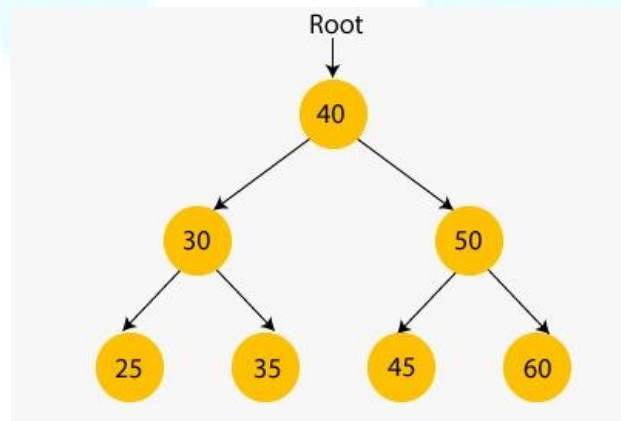
- It is a subtype of a binary tree organized in such a way that the left subnode of the parent node is always less than the parent node, and the right subnode of the parent node is always greater than the parent node.
- Similarly in this way the whole tree grows, it is the modified version of the binary tree.
- In binary search tree, we have more restrictions as compared to the binary; in a binary tree, we have to maintain the 0, 1, or 2 children of every node, but here in this it is a binary tree, along with it, it must have some order in which insertion of child nodes is present.
- In BST, the right child is always less than the root node, and similarly, in this way, the left child is larger than the root node. In this, we will categorize the Binary search tree.

4. Complete Binary Tree:

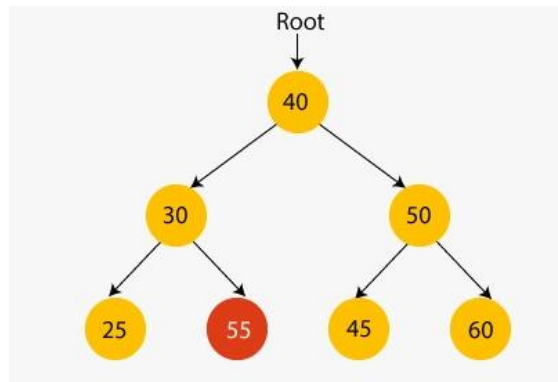
- The complete binary tree is another form of the binary tree that consists of at most two children; it contains 0 or two children.
- The insertion of the nodes in the complete binary tree is always done from the left to right side.

BINARY SEARCH TREE

- A binary search tree follows some order to arrange the elements.
- In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.
- This rule is applied recursively to the left and right subtrees of the root.
- Let's understand the concept of Binary search tree with an example.



- In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.
- Similarly, we can see the left child of root node is greater than its left child and smaller than its right child.
- So, it also satisfies the property of binary search tree.
- Therefore, we can say that the tree in the above image is a binary search tree.
- Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



- In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55.
- So, the above tree does not satisfy the property of Binary search tree.
- Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

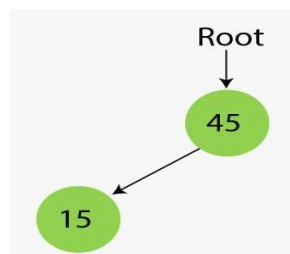
- Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50
- First, we have to insert 45 into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.
- Now, let's see the process of creating the Binary search tree using the given data element.
- The process of creating the BST is shown below -

Step 1 - Insert 45.

Binary Search tree

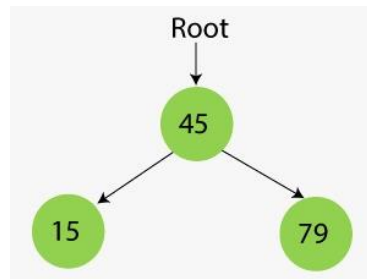
Step 2 - Insert 15.

- As 15 is smaller than 45, so insert it as the root node of the left subtree.

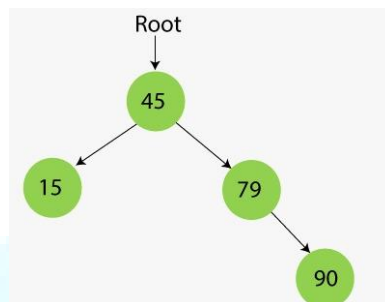


Step 3 - Insert 79.

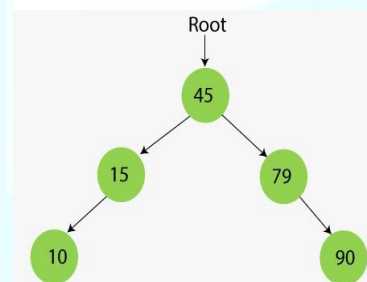
- As 79 is greater than 45, so insert it as the root node of the right subtree.

**Step 4 - Insert 90.**

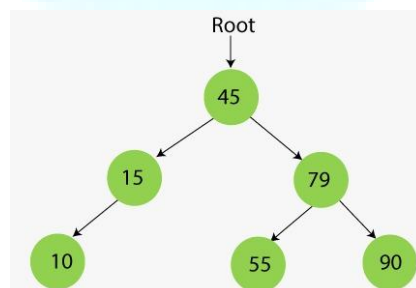
- 90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.

**Step 5 - Insert 10.**

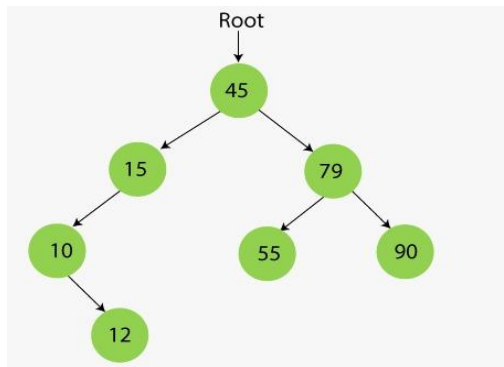
- 10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.

**Step 6 - Insert 55.**

- 55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.

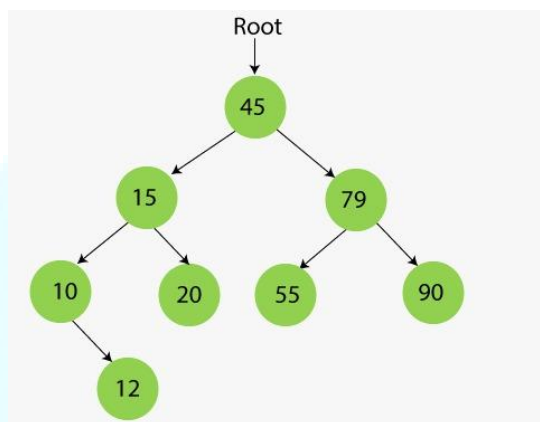
**Step 7 - Insert 12.**

- 12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



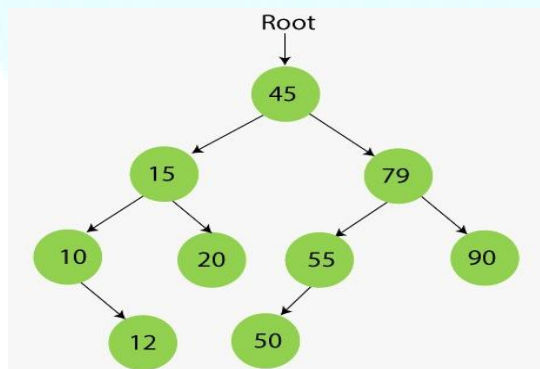
Step 8 - Insert 20.

- 20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

- 50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.

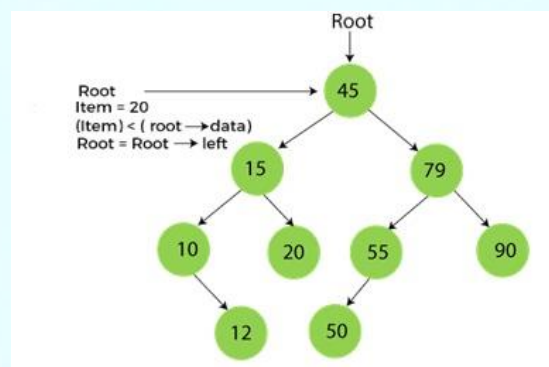


- Now, the creation of binary search tree is completed.
- After that, let's move towards the operations that can be performed on Binary search tree.
- We can perform insert, delete and search operations on the binary search tree.
- Let's understand how a search is performed on a binary search tree.

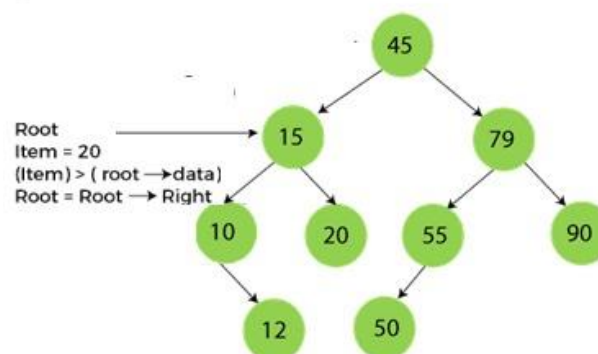
Searching in Binary search tree

- Searching means to find or locate a specific element or node in a data structure.
- In Binary search tree, searching a node is easy because elements in BST are stored in a specific order.
- **The steps of searching a node in Binary Search tree are listed as follows –**
 - ✓ First, compare the element to be searched with the root element of the tree.
 - ✓ If root is matched with the target element, then return the node's location.
 - ✓ If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
 - ✓ If it is larger than the root element, then move to the right subtree.
 - ✓ Repeat the above procedure recursively until the match is found.
 - ✓ If the element is not found or not present in the tree, then return NULL.
- Now, let's understand the searching in binary tree using an example.
- We are taking the binary search tree formed above.
- Suppose we have to find node 20 from the below tree.

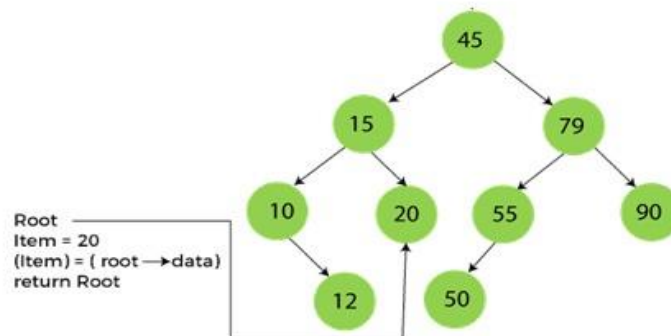
Step1:



Step 2:

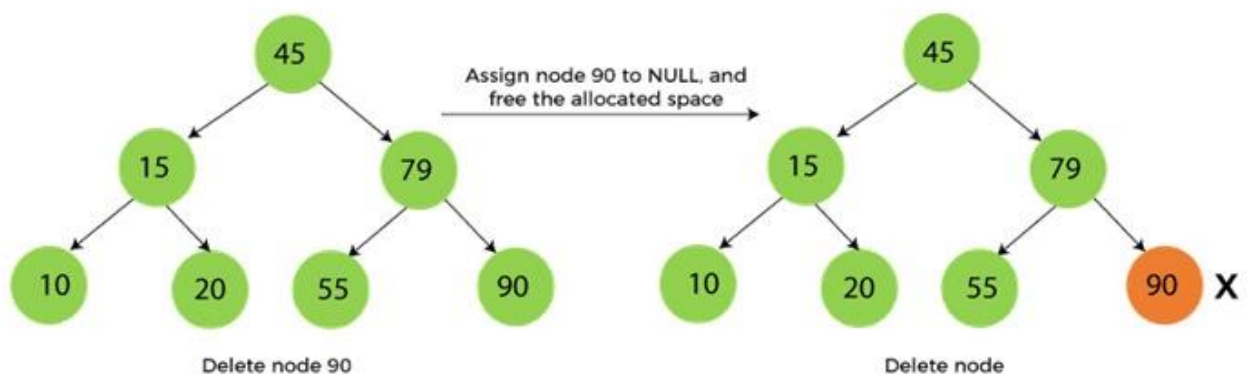


Step 3:



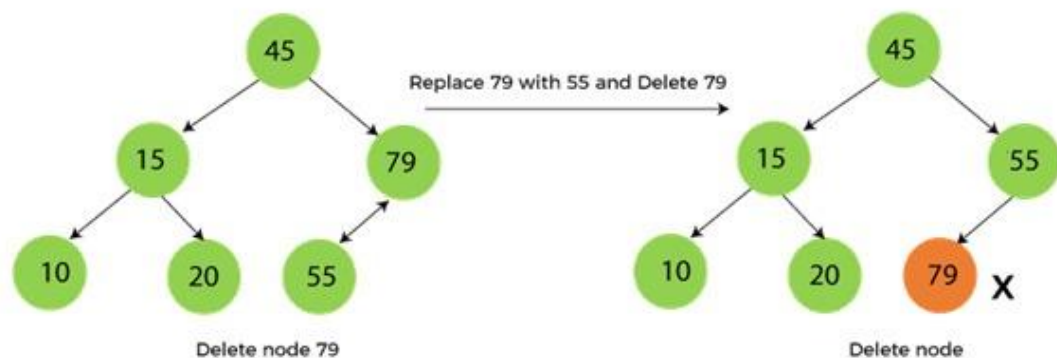
Deletion in Binary Search tree

- In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated.
- To delete a node from BST, there are three possible situations occur -
 - The node to be deleted is the leaf node, or,
 - The node to be deleted has only one child, and,
 - The node to be deleted has two children
- **When the node to be deleted is the leaf node**
 - It is the simplest case to delete a node in BST.
 - Here, we have to replace the leaf node with NULL and simply free the allocated space.
 - We can see the process to delete a leaf node from BST in the below image.
 - In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.

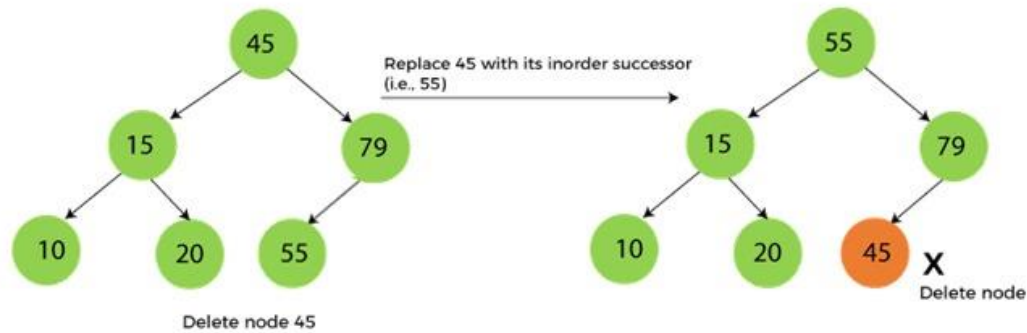


- **When the node to be deleted has only one child**

- In this case, we have to replace the target node with its child, and then delete the child node.
- It means that after replacing the target node with its child node, the child node will now contain the value to be deleted.
- So, we simply have to replace the child node with NULL and free up the allocated space.
- We can see the process of deleting a node with one child from BST in the below image.
- In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.
- So, the replaced node 79 will now be a leaf node that can be easily deleted.

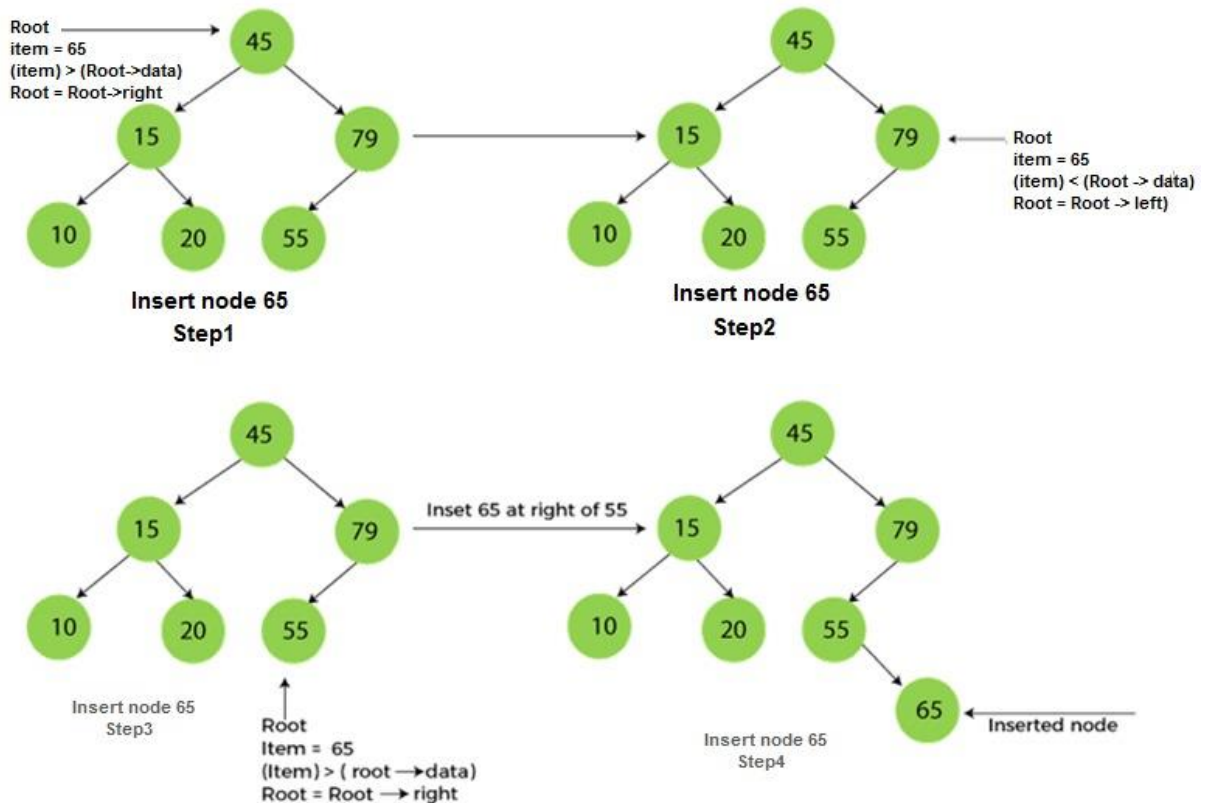


- **When the node to be deleted has two children**
- This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows –
 - ✓ First, find the inorder successor of the node to be deleted.
 - ✓ After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
 - ✓ And at last, replace the node with NULL and free up the allocated space.
 - ✓ The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.
- We can see the process of deleting a node with two children from BST in the below image.
- In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor.
- Now, node 45 will be at the leaf of the tree so that it can be deleted easily.



Insertion in Binary Search tree

- A new key in BST is always inserted at the leaf.
- To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree.
- Else, search for the empty location in the right subtree and insert the data.
- Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.
- Now, let's see the process of inserting a node into BST using an example.



The complexity of the Binary Search tree

- Let's see the time and space complexity of the Binary search tree.
- We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	$O(\log n)$	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$	$O(n)$

Where 'n' is the number of nodes in the given tree.

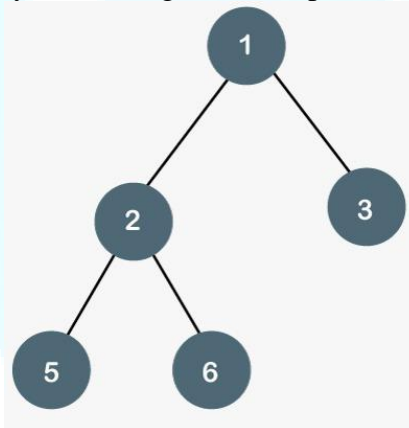
2. Space Complexity

Operations	Space complexity
Insertion	$O(n)$
Deletion	$O(n)$
Search	$O(n)$

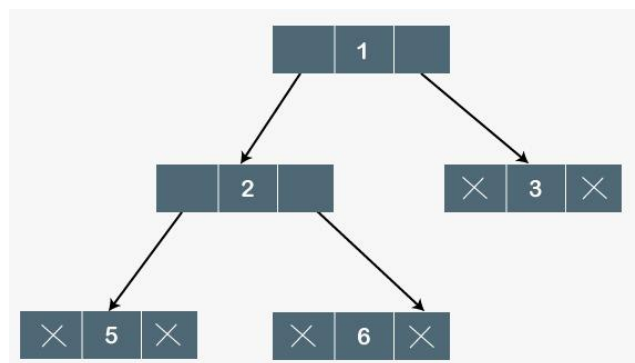
The space complexity of all operations of Binary search tree is $O(n)$.

BINARY TREE

- The Binary tree means that the node can have maximum two children.
- Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.
- Let's understand the binary tree through an example.



- The above tree is a binary tree because each node contains the utmost two children.
- The logical representation of the above tree is given below:



- In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively.
- The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right).
- The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain NULL pointer on both left and right parts.

Properties of Binary Tree

- At each level of i , the maximum number of nodes is 2^i .
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3.
- Therefore, the maximum number of nodes at height 3 is equal to $(1+2+4+8) = 15$.
- In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \dots + 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to $h+1$.
- If the number of nodes is minimum, then the height of the tree would be maximum.
- Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.
- If there are 'n' number of nodes in the binary tree.
- The minimum height can be computed as:

As we know that,

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

Taking log on both the sides,

$$\log_2(n+1) = \log_2(2^{h+1})$$
$$\log_2(n+1) = h+1$$
$$h = \log_2(n+1) - 1$$

- The maximum height can be computed as:

As we know that,

$$n = h+1$$
$$h = n-1$$

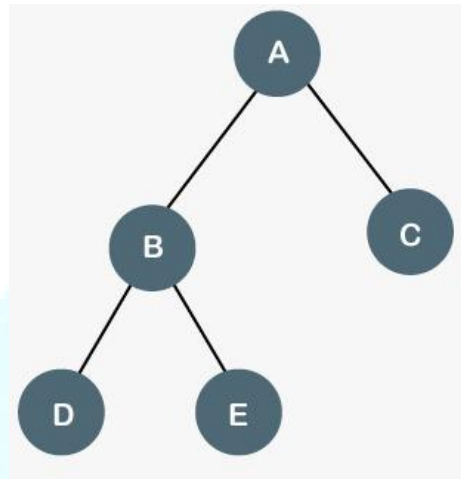
Types of Binary Tree

- There are four types of Binary tree:

- ✓ **Full/ proper/ strict Binary tree**
- ✓ **Complete Binary tree**
- ✓ **Perfect Binary tree**
- ✓ **Degenerate Binary tree**
- ✓ **Balanced Binary tree**

1. Full/ proper/ strict Binary tree

- The full binary tree is also known as a strict binary tree.
- The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children.
- The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.
- Let's look at the simple example of the Full Binary tree.



Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1.
- In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., $2^{h+1} - 1$.
- The minimum number of nodes in the full binary tree is $2^h - 1$.
- The minimum height of the full binary tree is $\log_2(n+1) - 1$.
- The maximum height of the full binary tree can be computed as:

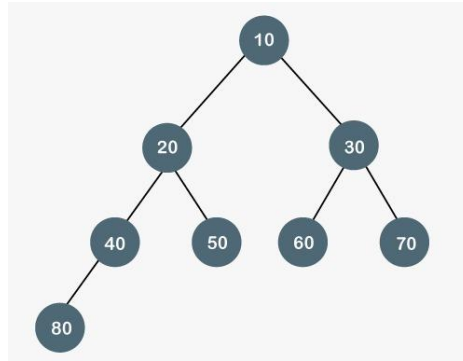
$$n = 2^h - 1$$

$$n+1 = 2^h$$

$$h = \log_2(n+1)$$

Complete Binary Tree

- The complete binary tree is a tree in which all the nodes are completely filled except the last level.
- In the last level, all the nodes must be as left as possible.
- In a complete binary tree, the nodes should be added from the left.
- Let's create a complete binary tree.



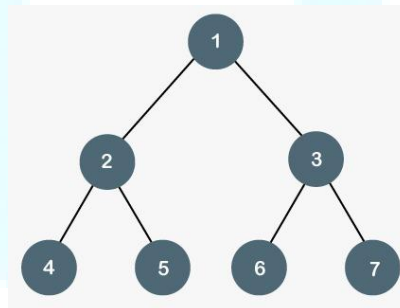
- The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

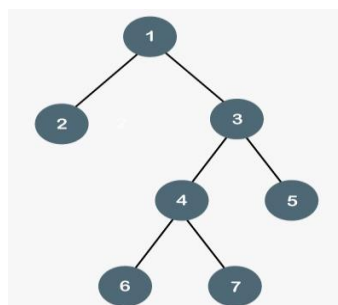
- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^h .
- The minimum height of a complete binary tree is $\log_2(n+1) - 1$.
- The maximum height of a complete binary tree is

Perfect Binary Tree

- A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.

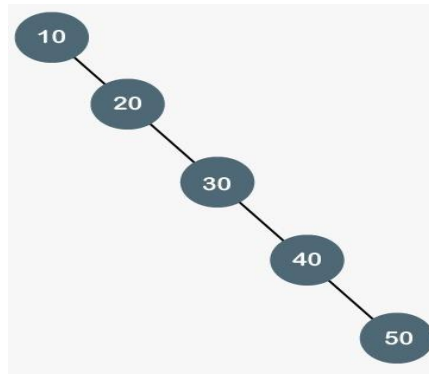


- Let's look at a simple example of a perfect binary tree.
- The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

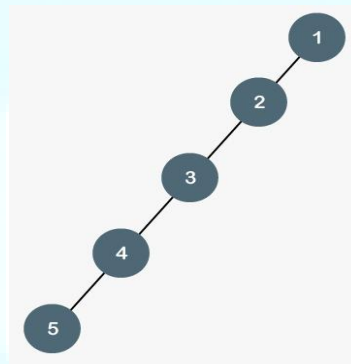


Degenerate Binary Tree

- The degenerate binary tree is a tree in which all the internal nodes have only one children.
- Let's understand the Degenerate binary tree through examples.



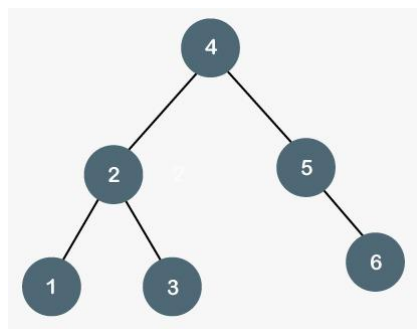
- The above tree is a degenerate binary tree because all the nodes have only one child.
- It is also known as a right-skewed tree as all the nodes have a right child only.



- The above tree is also a degenerate binary tree because all the nodes have only one child.
- It is also known as a left-skewed tree as all the nodes have a left child only.

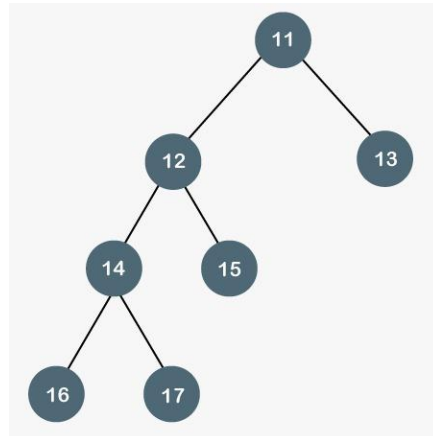
Balanced Binary Tree

- The balanced binary tree is a tree in which both the left and right trees differ by atmost 1.
- For example, AVL and Red-Black trees are balanced binary tree.
- Let's understand the balanced binary tree through examples.



- The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.

- The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.



Binary Tree

- A Binary tree is implemented with the help of pointers.
- The first node in the tree is represented by the root pointer.
- Each node in the tree consists of three parts, i.e., data, left pointer and right pointer.
- To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```

struct node
{
    int data,
    struct node *left, *right;
}
  
```

- In the above structure, data is the value, left pointer contains the address of the left node, and right pointer contains the address of the right node.

Binary Tree program in C

```

#include<stdio.h>
struct node
{
    int data;
    struct node *left, *right;
}
void main()
{
    struct node *root;
    root = create();
}
struct node *create()
{
    struct node *temp;
    int data;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Press 0 to exit");
    printf("\nPress 1 for new node");
    printf("Enter your choice : ");
    scanf("%d", &choice);
    if(choice==0)
    {
  
```

```
    return 0;
}
else
{
    printf("Enter the data:");
    scanf("%d", &data);
    temp->data = data;
    printf("Enter the left child of %d", data);
    temp->left = create();
    printf("Enter the right child of %d", data);
    temp->right = create();
    return temp;
}
```

- The above code is calling the create() function recursively and creating new node on each recursive call.
- When all the nodes are created, then it forms a binary tree structure.
- The process of visiting the nodes is known as tree traversal.
- There are three types traversals used to visit a node:

✓ **Inorder traversal**

✓ **Preorder traversal**

✓ **Postorder traversal**

BINARY TREE TRAVERSAL IN DATA STRUCTURE

- The tree can be defined as a non-linear data structure that stores data in the form of nodes, and nodes are connected to each other with the help of edges.
- Among all the nodes, there is one main node called the root node, and all other nodes are the children of these nodes.
- In any data structure, traversal is an important operation.
- In the traversal operation, we walk through the data structure visiting each element of the data structure at least once.
- The traversal operation plays a very important role while doing various other operations on the data structure like some of the operations are searching, in which we need to visit each element of the data structure at least once so that we can compare each incoming element from the data structure to the key that we want to find in the data structure.
- So like any other data structure, the tree data also needs to be traversed to access each element, also known as a node of the tree data structure.
- There are different ways of traversing a tree depending upon the order in which the tree's nodes are visited and the types of data structure used for traversing the tree.
- There are various data structures involved in traversing a tree, as traversing a tree involves iterating over all nodes in some manner.
- As from a given node, there could be more than one way to traverse or visit the next node of the tree, so it becomes important to store one of the nodes traverses further and store the rest of the nodes having a possible path for backtracking the tree if needed.
- Backtracking is not a linear approach, so we need different data structures for traversing through the whole tree.
- The stack and queue are the major data structure that is used for traversing a tree.
- Traversal is a technique for visiting all of a tree's nodes and printing their values.

- Traversing a tree involves iterating over all nodes in some manner.
- We always start from the root (head) node since all nodes are connected by edges (links).
- As the tree is not a linear data structure, there can be more than one possible next node from a given node, so some nodes must be deferred, i.e., stored in some way for later visiting.

Types of Traversal of Binary Tree

- There are three types of traversal of a binary tree.

- ✓ **Inorder tree traversal**
- ✓ **Preorder tree traversal**
- ✓ **Postorder tree traversal**

Inorder Tree Traversal

- The left subtree is visited first, followed by the root, and finally the right subtree in this traversal strategy.
- Always keep in mind that any node might be a subtree in and of itself.
- The output of a binary tree traversal in order produces sorted key values in ascending order.

Preorder Tree Traversal

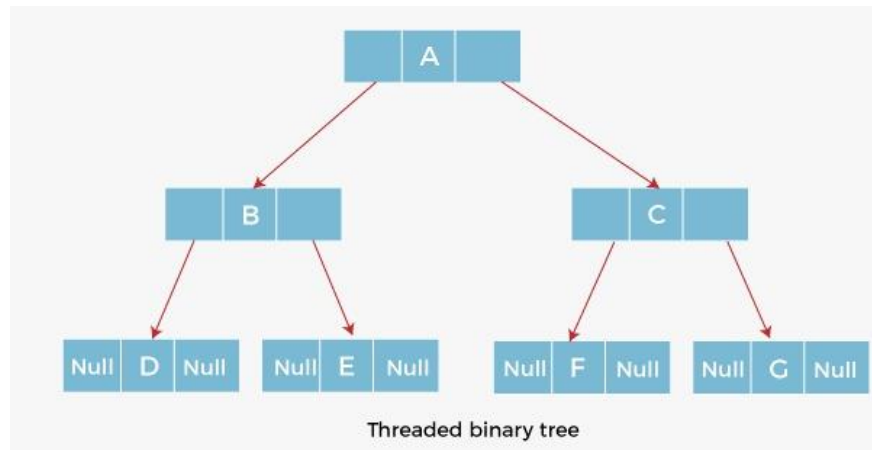
- In this traversal method, the root node is visited first, then the left subtree, and finally the right subtree.

Postorder Tree Traversal

- The root node is visited last in this traversal method, hence the name. First, we traverse the left subtree, then the right subtree, and finally the root node.

THREADED BINARY TREE

- In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space.
- If a binary tree consists of n nodes then $n+1$ link fields contain NULL values.
- So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads.
- Such binary trees with threads are known as threaded binary trees.
- Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

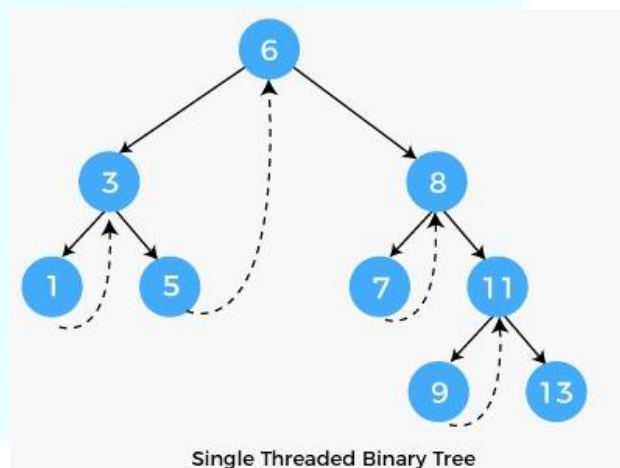


Types of Threaded Binary Tree

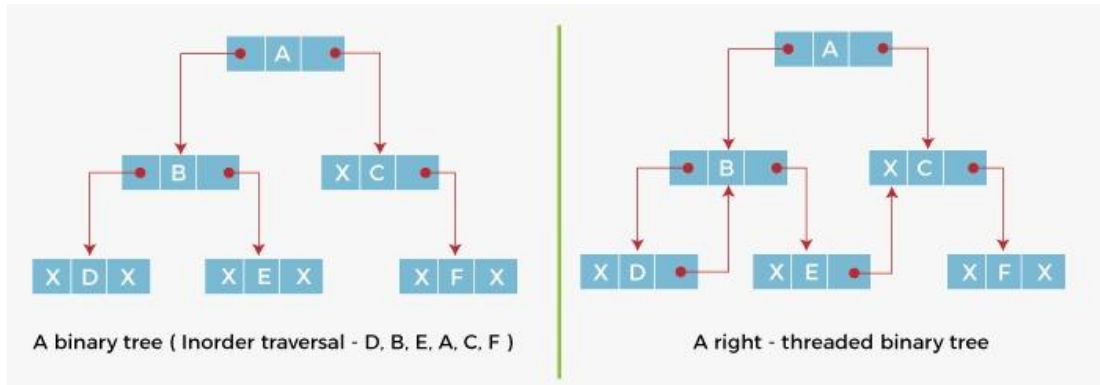
- There are two types of threaded Binary Tree:

- ✓ **One-way threaded Binary Tree**
- ✓ **Two-way threaded Binary Tree**

One-way threaded Binary trees:

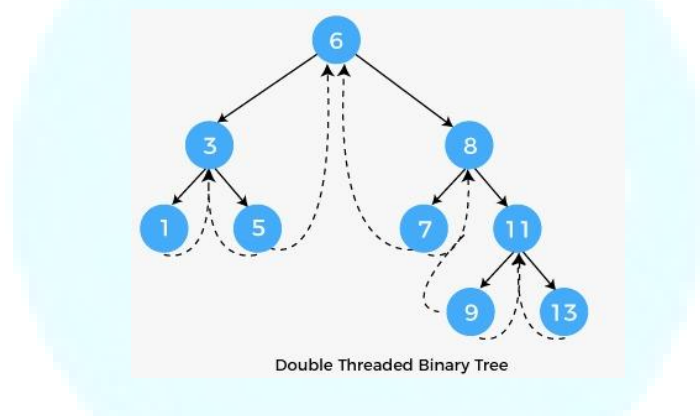


- In one-way threaded binary trees, a thread will appear either in the right or left link field of a node.
- If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal.
- Such trees are called Right threaded binary trees.
- If thread appears in the left field of a node then it will point to the nodes inorder predecessor.
- Such trees are called Left threaded binary trees.
- Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees.
- In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL.
- In order to distinguish threads from normal links they are represented by dotted lines.

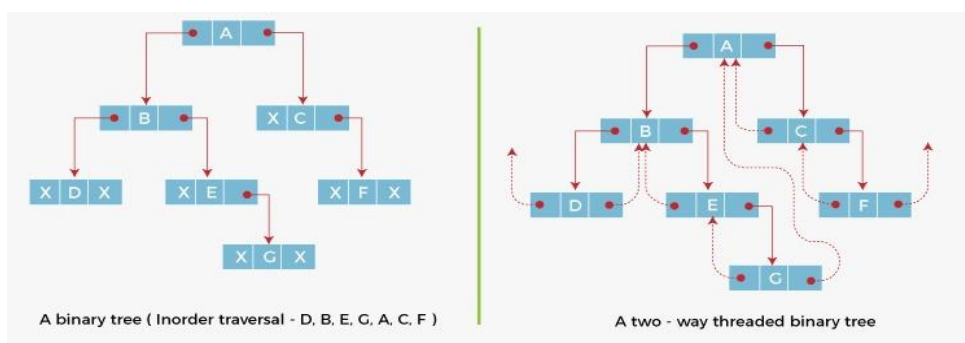


- The above figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F.
- When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the inorder successor of a node D.
- In the same way other nodes containing values in the right link field will contain NULL value.

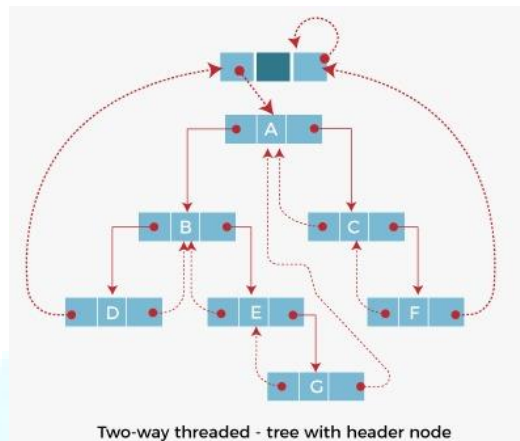
Two-way threaded Binary Trees:



- In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.



- The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F.
- If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B.
- Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor.
- In the same way, other nodes containing NULL values in their link fields are filled with threads.



- In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node.
- This is because they don't have any inorder predecessor and successor respectively.
- This is indicated by threads pointing nowhere.
- So in order to maintain the uniformity of threads, we maintain a special node called the header node.
- The header node does not contain any data part and its left link field points to the root node and its right link field points to itself.
- If this header node is included in the two-way threaded Binary tree then this node becomes the inorder predecessor of the first node and inorder successor of the last node.
- Now threads of left link fields of the first node and right link fields of the last node will point to the header node.

Advantages of Threaded Binary Tree:

- In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack.
- If the stack is used then it consumes a lot of memory and time.
- It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links.
- It almost behaves like a circular linked list.

Disadvantages of Threaded Binary Tree:

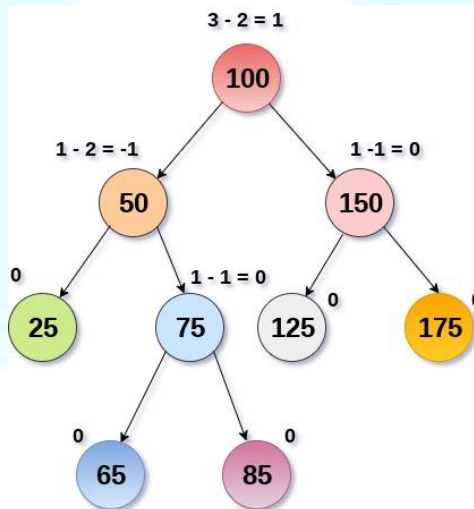
- When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
- Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

AVL TREE

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962.
- The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

$$\text{Balance Factor (k)} = \text{height (left(k))} - \text{height (right(k))}$$

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
- An AVL tree is given in the following figure.
- We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

COMPLEXITY

Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Operations on AVL tree

- Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree.

- Searching and traversing do not lead to the violation in property of AVL tree.
- However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

- AVL tree controls the height of the binary search tree by not letting it to be skewed.
- The time taken for all operations in a binary search tree of height h is $O(h)$.
- However, it can be extended to $O(n)$ if the BST becomes skewed (i.e. worst case).
- By limiting this height to $\log n$, AVL tree imposes an upper bound on each operation to be $O(\log n)$ where n is the number of nodes.

AVL Rotations

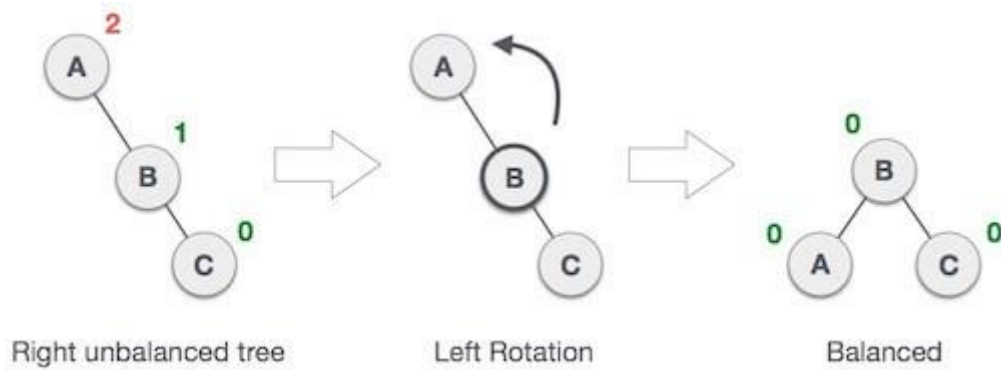
- We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1.
- There are basically four types of rotations which are as follows:
 - ✓ **L L rotation**: Inserted node is in the left subtree of left subtree of A
 - ✓ **R R rotation** : Inserted node is in the right subtree of right subtree of A
 - ✓ **L R rotation** : Inserted node is in the right subtree of left subtree of A
 - ✓ **R L rotation** : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations.
- For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

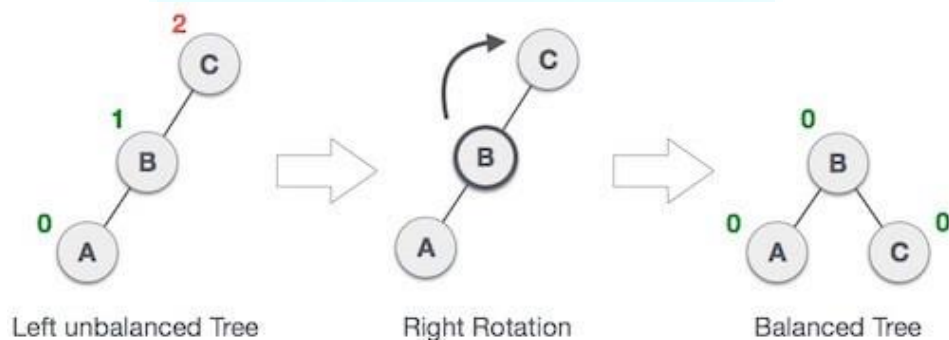
- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation.
- RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2.



- In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree.
- We perform the RR rotation on the edge below A.

2. LL Rotation

- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation,.
- LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

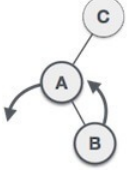
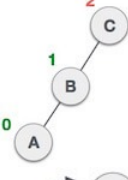
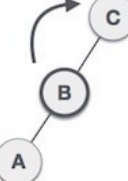
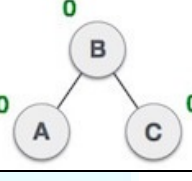


- In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree.
- We perform the LL rotation on the edge below A.

3. LR Rotation

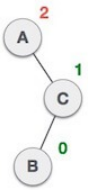
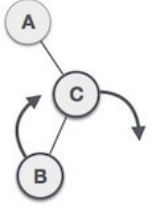
- Double rotations are bit tougher than single rotation which has already explained above.
- LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.
- Let us understand each and every step very clearly:

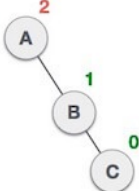
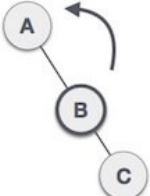
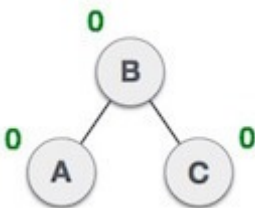
State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted

	node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

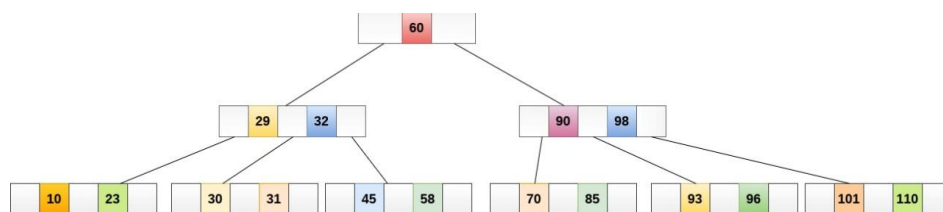
- As already discussed, that double rotations are bit tougher than single rotation which has already explained above.
- R L rotation= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.

	<p>After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.</p>
	<p>Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.</p>

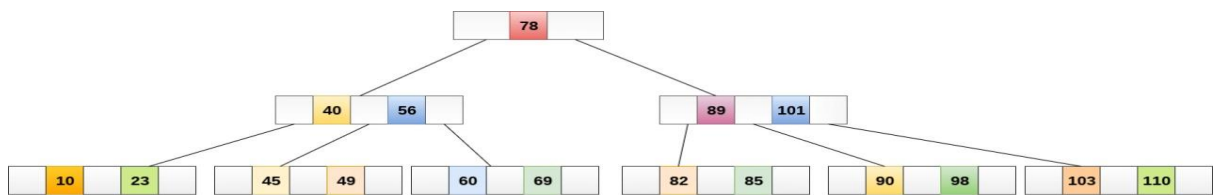
B TREE

- B Tree is a specialized m-way tree that can be widely used for disk access.
- A B-Tree of order m can have at most m-1 keys and m children.
- One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.
- A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.
- Every node in a B-Tree contains at most m children.
- Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.
- The root nodes must have at least 2 nodes.
- All leaf nodes must be at the same level.
- It is not necessary that, all the nodes contain the same number of children but, each node must have $m/2$ number of nodes.
- A B tree of order 4 is shown in the following image.



Searching:

- Searching in B Trees is similar to that in Binary search tree.
- For example, if we search for an item 49 in the following B Tree.
- The process will something like following :
 - ✓ Compare item 49 with root node 78. since $49 < 78$ hence, move to its left sub-tree.
 - ✓ Since, $40 < 49 < 56$, traverse right sub-tree of 40.
 - ✓ $49 > 45$, move to right. Compare 49.
 - ✓ Match found, return.
- Searching in a B tree depends upon the height of the tree. The search algorithm takes $O(\log n)$ time to search any element in a B tree.



Inserting

- Insertions are done at the leaf node level.
- The following algorithm needs to be followed in order to insert an item into B Tree.
 - ✓ Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
 - ✓ If the leaf node contain less than $m-1$ keys then insert the element in the increasing order.
 - ✓ Else, if the leaf node contains $m-1$ keys, then follow the following steps.
 - ✓ Insert the new element in the increasing order of elements.
 - ✓ Split the node into the two nodes at the median.
 - ✓ Push the median element upto its parent node.
 - ✓ If the parent node also contain $m-1$ number of keys, then split it too by following the same steps.

Deletion

- Deletion is also performed at the leaf nodes.
- The node which is to be deleted can either be a leaf node or an internal node.
- Following algorithm needs to be followed in order to delete a node from a B tree.
 - ✓ Locate the leaf node.
 - ✓ If there are more than $m/2$ keys in the leaf node then delete the desired key from the node.
 - ✓ If the leaf node doesn't contain $m/2$ keys then complete the keys by taking the element from right or left sibling.

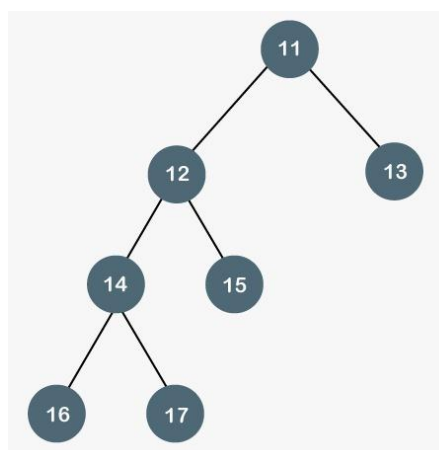
- ✓ If the left sibling contains more than $m/2$ elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
- ✓ If the right sibling contains more than $m/2$ elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
- ✓ If neither of the sibling contain more than $m/2$ elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
- ✓ If parent is left with less than $m/2$ nodes then, apply the above process on the parent too.
- ✓ If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Application of B tree

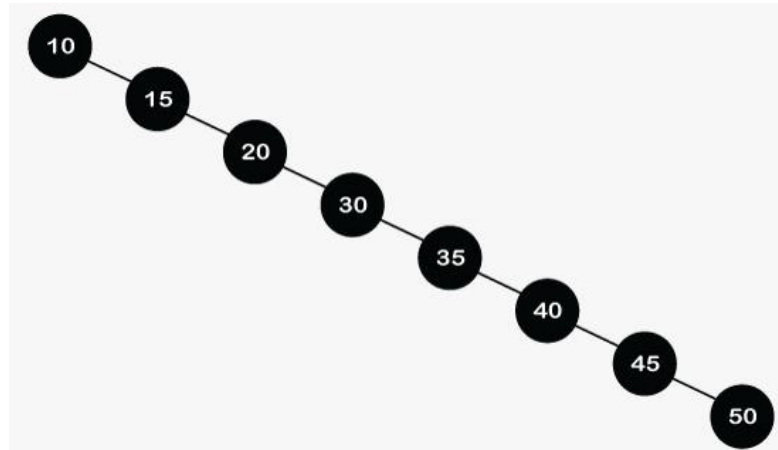
- B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.
- Searching an un-indexed and unsorted database containing n key values needs $O(n)$ running time in worst case.
- However, if we use B Tree to index this database, it will be searched in $O(\log n)$ time in worst case.

RED-BLACK TREE IN DATA STRUCTURE

- The red-Black tree is a binary search tree.
- The prerequisite of the red-black tree is that we should know about the binary search tree. In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.
- Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc.
- In a binary search tree, the searching, insertion and deletion take $O(\log_2 n)$ time in the average case, $O(1)$ in the best case and $O(n)$ in the worst case.
- Let's understand the different scenarios of a binary search tree.



- In the above tree, if we want to search the 80.
- We will first compare 80 with the root node. 80 is greater than the root node, i.e., 10, so searching will be performed on the right subtree.
- Again, 80 is compared with 15; 80 is greater than 15, so we move to the right of the 15, i.e., 20.
- Now, we reach the leaf node 20, and 20 is not equal to 80.
- Therefore, it will show that the element is not found in the tree.
- After each operation, the search is divided into half.
- The above BST will take $O(\log n)$ time to search the element.



- The above tree shows the right-skewed BST.
- If we want to search the 80 in the tree, we will compare 80 with all the nodes until we find the element or reach the leaf node.
- So, the above right-skewed BST will take $O(N)$ time to search the element.
- In the above BST, the first one is the balanced BST, whereas the second one is the unbalanced BST.
- We conclude from the above two binary search trees that a balanced tree takes less time than an unbalanced tree for performing any operation on the tree.
- Therefore, we need a balanced tree, and the Red-Black tree is a self-balanced binary search tree.
- Now, the question arises that why do we require a Red-Black tree if AVL is also a height-balanced tree.
- The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree.
- The main difference between the AVL tree and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced.
- So, the AVL tree is more balanced than the Red-Black tree, but the Red-Black tree guarantees $O(\log 2n)$ time for all operations like insertion, deletion, and searching.
- Insertion is easier in the AVL tree as the AVL tree is strictly balanced, whereas deletion and searching are easier in the Red-Black tree as the Red-Black tree requires fewer rotations.
- As the name suggests that the node is either colored in Red or Black color.
- Sometimes no rotation is required, and only recoloring is needed to balance the tree.

Properties of Red-Black tree

- It is a self-balancing Binary Search tree.
- Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
- This tree data structure is named as a Red-Black tree as each node is either Red or Black in color.
- Every node stores one extra information known as a bit that represents the color of the node.
- For example, 0 bit denotes the black color while 1 bit denotes the red color of the node.
- Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
- In the Red-Black tree, the root node is always black in color.
- In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
- If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
- Every path from a node to any of its descendant's NIL node should have same number of black nodes.

Is every AVL tree can be a Red-Black tree?

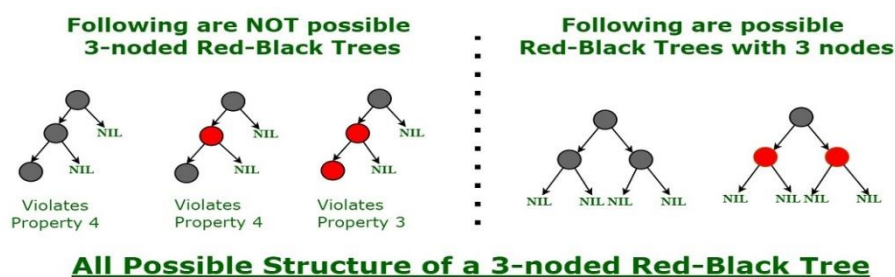
- Yes, every AVL tree can be a Red-Black tree if we color each node either by Red or Black color.
- But every Red-Black tree is not an AVL because the AVL tree is strictly height-balanced while the Red-Black tree is not completely height-balanced.

Insertion in Red Black tree

- The following are some rules used to create the Red-Black tree:
 - ✓ If the tree is empty, then we create a new node as a root node with the color black.
 - ✓ If the tree is not empty, then we create a new node as a leaf node with a color red.
 - ✓ If the parent of a new node is black, then exit.
 - ✓ If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.

4a) If the color is Black, then we perform rotations and recoloring.

4b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node.



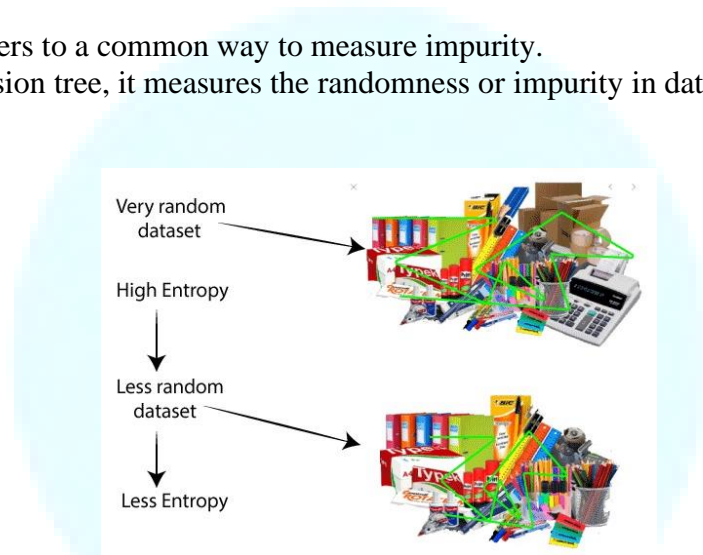
DECISION TREE INDUCTION

- Decision Tree is a supervised learning method used in data mining for classification and regression methods.
- It is a tree that helps us in decision-making purposes.
- The decision tree creates classification or regression models as a tree structure.
- It separates a data set into smaller subsets, and at the same time, the decision tree is steadily developed.
- The final tree is a tree with the decision nodes and leaf nodes.
- A decision node has at least two branches.
- The leaf nodes show a classification or decision.
- We can't accomplish more split on leaf nodes-The uppermost decision node in a tree that relates to the best predictor called the root node.
- Decision trees can deal with both categorical and numerical data.

Key factors:

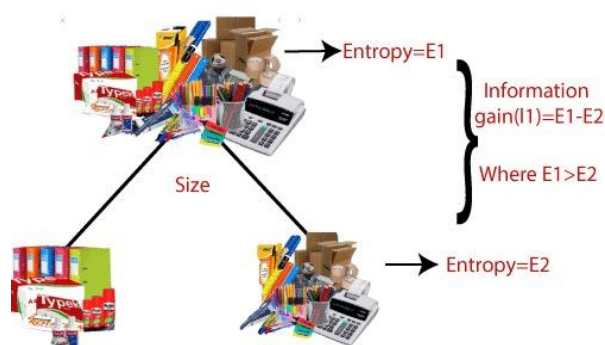
Entropy:

- Entropy refers to a common way to measure impurity.
- In the decision tree, it measures the randomness or impurity in data sets.



Information Gain:

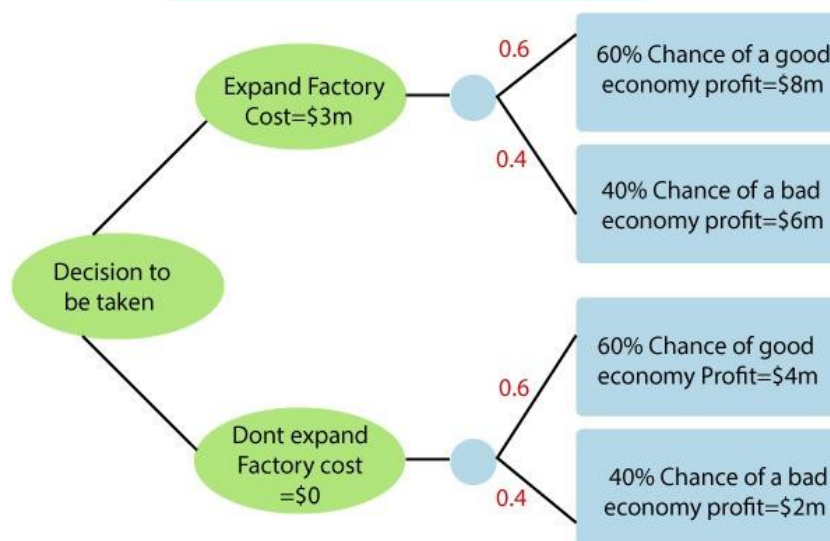
- Information Gain refers to the decline in entropy after the dataset is split. It is also called Entropy Reduction.
- Building a decision tree is all about discovering attributes that return the highest data gain.



- In short, a decision tree is just like a flow chart diagram with the terminal nodes showing decisions.
- Starting with the dataset, we can measure the entropy to find a way to segment the set until the data belongs to the same class.

Why are decision trees useful?

- It enables us to analyze the possible consequences of a decision thoroughly.
- It provides us a framework to measure the values of outcomes and the probability of accomplishing them.
- It helps us to make the best decisions based on existing data and best speculations.
- In other words, we can say that a decision tree is a hierarchical tree structure that can be used to split an extensive collection of records into smaller sets of the class by implementing a sequence of simple decision rules.
- A decision tree model comprises a set of rules for portioning a huge heterogeneous population into smaller, more homogeneous, or mutually exclusive classes.
- The attributes of the classes can be any variables from nominal, ordinal, binary, and quantitative values, in contrast, the classes must be a qualitative type, such as categorical or ordinal or binary.
- In brief, the given data of attributes together with its class, a decision tree creates a set of rules that can be used to identify the class.
- One rule is implemented after another, resulting in a hierarchy of segments within a segment.
- The hierarchy is known as the tree, and each segment is called a node.
- With each progressive division, the members from the subsequent sets become more and more similar to each other.
- Hence, the algorithm used to build a decision tree is referred to as recursive partitioning.
- The algorithm is known as CART (Classification and Regression Trees)
- Consider the given example of a factory where



- Expanding factor costs \$3 million, the probability of a good economy is 0.6 (60%), which leads to \$8 million profit, and the probability of a bad economy is 0.4 (40%), which leads to \$6 million profit.
- Not expanding factor with 0\$ cost, the probability of a good economy is 0.6(60%), which leads to \$4 million profit, and the probability of a bad economy is 0.4, which leads to \$2 million profit.
- The management teams need to take a data-driven decision to expand or not based on the given data.
$$\text{Net Expand} = (0.6 * 8 + 0.4 * 6) - 3 = \$4.2\text{M}$$
$$\text{Net Not Expand} = (0.6 * 4 + 0.4 * 2) - 0 = \$3\text{M}$$
$$\$4.2\text{M} > \$3\text{M}, \text{ therefore the factory should be expanded.}$$

Decision tree Algorithm:

- The decision tree algorithm may appear long, but it is quite simply the basis algorithm techniques is as follows:
- The algorithm is based on three parameters: **D**, **attribute_list**, and **Attribute_selection_method**.
- Generally, we refer to D as a data partition.
- Initially, D is the entire set of training tuples and their related class levels (input training data).
- The parameter **attribute_list** is a set of attributes defining the tuples.
- **Attribute_selection_method** specifies a heuristic process for choosing the attribute that "best" discriminates the given tuples according to class.
- Attribute_selection_method process applies an attribute selection measure.

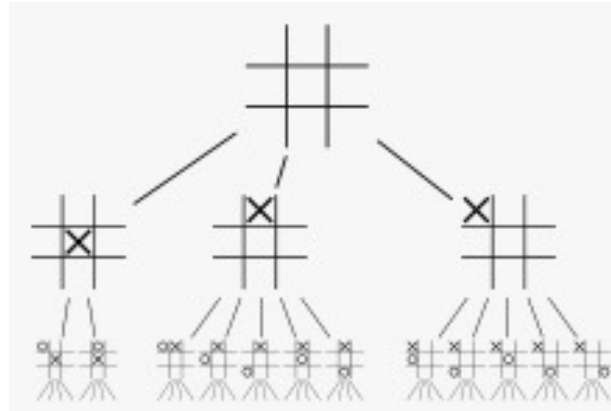
Advantages of using decision trees:

- A decision tree does not need scaling of information.
- Missing values in data also do not influence the process of building a choice tree to any considerable extent.
- A decision tree model is automatic and simple to explain to the technical team as well as stakeholders.
- Compared to other algorithms, decision trees need less exertion for data preparation during pre-processing.
- A decision tree does not require a standardization of data.

GAME TREE

- Game trees are generally used in board games to determine the best possible move.
- For the purpose of this article, Tic-Tac-Toe will be used as an example.
- The idea is to start at the current board position, and check all the possible moves the computer can make.
- Then, from each of those possible moves, to look at what moves the opponent may make.
- Then to look back at the computer's.
- Ideally, the computer will flip back and forth, making moves for itself and its opponent, until the game's completion.
- It will do this for every possible outcome (see image below), effectively playing thousands (often more) of games.

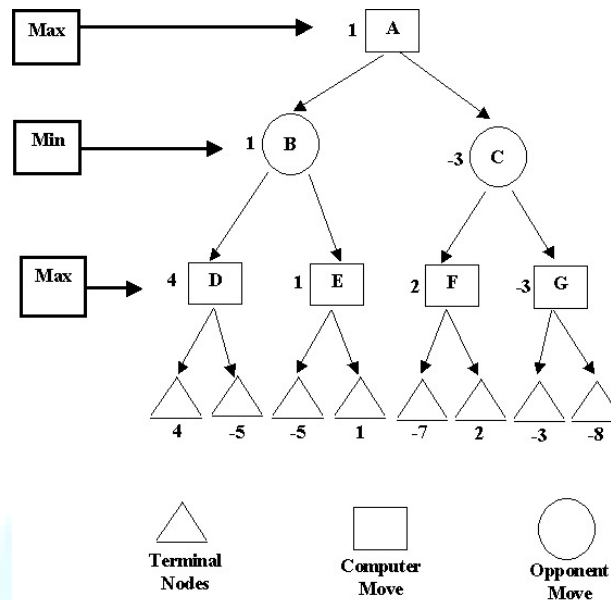
- From the winners and losers of these games, it tries to determine the outcome that gives it the best chance of success.
- This can be likened to how people think during a board game - for example "if I make this move, they could make this one, and I could counter with this" etc.
- Game trees do just that. They examine every possible move and every opponent move, and then try to see if they are winning after as many moves as they can think through.



- As one could imagine, there are an extremely large number of possible games.
- In Tic-Tac-Toe, there are 9 possible first moves (ignoring rotation optimization). There are 8 on the second turn, then 7, 6, etc.
- In total, there are 255,168 possible games.
- The goal of a game tree is to look at them all (on the first move) and try to choose a move that will, at worst, make losing impossible, and, at best, win the game.
- The computer begins evaluating a move by calculating all possible moves.
- In Tic-Tac-Toe, it is filling any empty square on the grid, thus the number of possible moves will be equal to the number of empty squares.
- Once it has found these moves, it loops over each of the possible moves, and tries to find out whether the move will result in a win for the computer or not.
- It does this by running this same algorithm (recursion) over the position (obtained by performing one of the computer's original possible moves) but trying to calculate the opponent's best move.
- To find if a move is "good" or "bad" the game tree is extended and continues to branch out until the game ends (a "terminal node").
- Terminal nodes are then assigned values based on the result of the game; the higher the value, the better the result is for the computer.
- Because there are only two possible results (a win or a loss), the values of "-1" and "1" can be used to represent who has won (the example below offers a greater variety of values than "-1" and "1").
- Now that the terminal nodes' values have been determined, the node above them (nodes D, E, F, and G in the picture below).
- If the node represents the computer's choice of moves it is a "max node", if it represents the player's choice of moves it is a "min" node.
- The value of a max node becomes that of the highest node beneath it.
- The value of a min node becomes that of the lowest node beneath it.
- The value D in the example below is 4 and the value of E is 1. The value of B becomes 1 (the lowest of 4 and 1).

- By continuing to move up, each node is given a value.
- The top node (A) then assumes the highest value of the nodes beneath it; the node with the highest value is the move that the computer should make.

Note: If the position is that of a finished game, and the winner is the opponent, the position is regarded as a winning position for the opponent (i.e. a "bad" move for the computer), and vice-versa.



Components

- All game trees require the same script concepts.
 - The names given to these scripts in this article are not official, but rather represent the functions of the scripts.
- ✓ **A movement script** — This returns an Array of the positions possible to be obtained by a move made by a given player.
 - ✓ **A make move script** — This selects the best of the moves returned by the above script, with the logic described in the above section.
 - ✓ **A winner script** — This takes in a position and outputs if it is a completed game, and if so, who won.

SEARCHING

- Searching is the process of finding some particular element in the list.
- If the element is present in the list, then the process is called successful, and the process returns the location of that element.
- Otherwise, the search is called unsuccessful.
- **Linear Search and Binary Search are the two popular searching techniques.**

BINARY SEARCH

- Binary search is the search technique that works efficiently on sorted lists.

- Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.
- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.
- If the match is found then, the location of the middle element is returned.
- Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm

Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is the index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <= end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val

set pos = mid

print pos

go to step 6

else if a[mid] > val

set end = mid - 1

else

set beg = mid + 1

[end of if]

[end of loop]

Step 5: if pos = -1

print "value is not present in the array"

[end of if]

Step 6: exit

Working of Binary search

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

- There are two methods to implement the binary search algorithm -

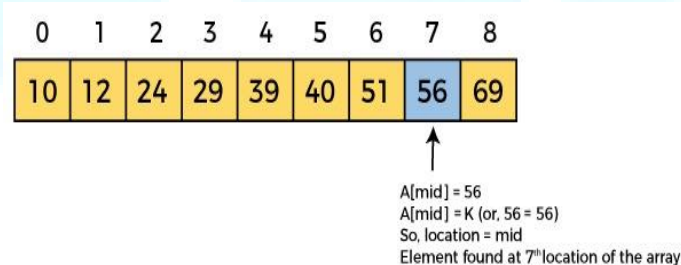
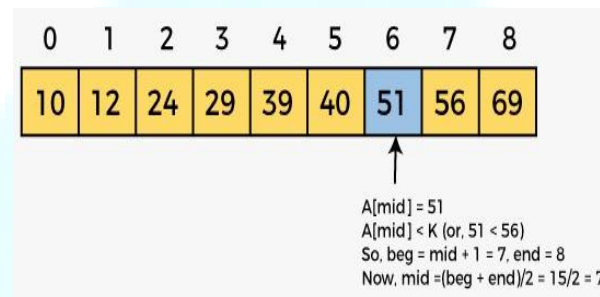
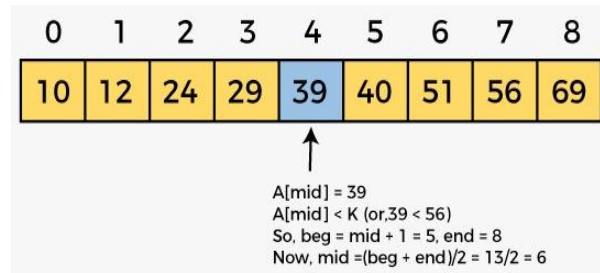
- **Iterative method**
- **Recursive method**

- The recursive method of binary search follows the divide and conquers approach.
- Let the elements of array are -

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

- Let the element to search is, K = 56

- We have to use the below formula to calculate the mid of the array -
 $\text{mid} = (\text{beg} + \text{end})/2$
 So, in the given array -
 $\text{beg} = 0$
 $\text{end} = 8$
 $\text{mid} = (0 + 8)/2 = 4$. So, 4 is the mid of the array.



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

Binary Search complexity

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

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

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

3. Space Complexity

Space Complexity $O(1)$

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

LINEAR SEARCH ALGORITHM

- **Linear search is also called as sequential search algorithm.**
- It is the simplest searching algorithm.
- In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.
- If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is $O(n)$.
- The steps used in the implementation of Linear Search are listed as follows -
 - ✓ First, we have to traverse the array elements using a for loop.
 - ✓ In each iteration of for loop, compare the search element with the current array element, and -
 - ✓ If the element matches, then return the index of the corresponding array element.
 - ✓ If the element does not match, then move to the next element.
 - ✓ If there is no match or the search element is not present in the given array, return -1.

Algorithm

Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

Step 1: set pos = -1

Step 2: set i = 1

Step 3: repeat step 4 while i <= n

Step 4: if a[i] == val

set pos = i

print pos

go to step 6

[end of if]

set ii = i + 1

```
[end of loop]
Step 5: if pos = -1
print "value is not present in the array "
[end of if]
Step 6: exit
```

Working of Linear search

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

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

- Let the element to be searched is $K = 41$
- Now, start from the first element and compare K with each element of the array.

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

↑
 $K \neq 70$

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

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

↑
 $K \neq 40$

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

↑
 $K \neq 30$

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

↑
 $K \neq 11$

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

↑
 $K \neq 57$

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

↑
 $K = 41$

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

Linear Search complexity

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

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

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

3. Space Complexity

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

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

SORTING ALGORITHMS

- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.
- For example, consider an array $A = \{A_1, A_2, A_3, A_4, \dots, A_n\}$, the array is called to be in ascending order if element of A are arranged like $A_1 < A_2 < A_3 < A_4 < A_5 < \dots < A_n$.
- Consider an array;
 $\text{int } A[10] = \{ 5, 4, 10, 2, 30, 45, 34, 14, 18, 9 \}$
- The Array sorted in ascending order will be given as;
 $A[] = \{ 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 \}$
- There are many techniques by using which, sorting can be performed.
- In this section of the tutorial, we will discuss each method in detail.

Sorting Algorithms

- Sorting algorithms are described in the following table along with the description.

SN	Sorting Algorithms	Description
1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
2	Bucket Sort	Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. In this sorting algorithms, Buckets are sorted individually by using different sorting algorithm.
3	Comb Sort	Comb Sort is the advanced form of Bubble Sort. Bubble Sort compares all the adjacent values while comb sort removes all the turtle values or small values near the end of the list.
4	Counting Sort	It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of

		objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.
5	Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements depending upon the choice and the elements are sorted by deleting the root element of the heap.
6	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
7	Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.
8	Quick Sort	Quick sort is the most optimized sort algorithms which performs sorting in $O(n \log n)$ comparisons. Like Merge sort, quick sort also work by

		using divide and conquer approach.
9	Radix Sort	In Radix sort, the sorting is done as we do sort the names according to their alphabetical order. It is the linear sorting algorithm used for Integers.
10	Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time $O(n^2)$ which is worst than insertion sort.
11	Shell Sort	Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

BUBBLE SORT ALGORITHM

- The working procedure of bubble sort is simplest.
- Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order.
- It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water.

- Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.
- Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world.
- It is not suitable for large data sets.
- The average and worst-case complexity of Bubble sort is $O(n^2)$, where n is a number of items.
- Bubble sort is majorly used where -
 - complexity does not matter
 - simple and short code is preferred

Algorithm

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

```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```

Working of Bubble sort Algorithm

- To understand the working of bubble sort algorithm, let's take an unsorted array.
- We are taking a short and accurate array, as we know the complexity of bubble sort is $O(n^2)$.
- Let the elements of array are -

13	32	26	35	10
----	----	----	----	----

First Pass

- Sorting will start from the initial two elements. Let compare them to check which is greater.

13	32	26	35	10
----	----	----	----	----

- Here, 32 is greater than 13 ($32 > 13$), so it is already sorted. Now, compare 32 with 26.

13	32	26	35	10
----	----	----	----	----

- Here, 26 is smaller than 32. So, swapping is required.
- After swapping new array will look like –

13	26	32	35	10
----	----	----	----	----

- Now, compare 32 and 35.

13	26	32	35	10
----	----	----	----	----

- Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
- Now, the comparison will be in between 35 and 10.

13	26	32	35	10
----	----	----	----	----

- Here, 10 is smaller than 35 that are not sorted. So, swapping is required.
- Now, we reach at the end of the array. After first pass, the array will be -

13	26	32	10	35
----	----	----	----	----

- Now, move to the second iteration.

Second Pass

- The same process will be followed for second iteration.

13	26	32	10	35
----	----	----	----	----

13	26	32	10	35
----	----	----	----	----

13	26	32	10	35
----	----	----	----	----

- Here, 10 is smaller than 32. So, swapping is required.
- After swapping, the array will be –

13	26	10	32	35
----	----	----	----	----

13	26	10	32	35
----	----	----	----	----

- Now, move to the third iteration.

Third Pass

- The same process will be followed for third iteration.

13	26	10	32	35
----	----	----	----	----

13	26	10	32	35
----	----	----	----	----

- Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be –

13	10	26	32	35
----	----	----	----	----

13	10	26	32	35
----	----	----	----	----

13	10	26	32	35
----	----	----	----	----

- Now, move to the fourth iteration.

Fourth pass

- Similarly, after the fourth iteration, the array will be –
- Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

- Now, let's see the time complexity of bubble sort in the best case, average case, and worst case.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is $O(n)$.
- Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is $O(n^2)$.
- Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is $O(n^2)$.

2. Space Complexity

Space Complexity	$O(1)$
Stable	YES

- The space complexity of bubble sort is $O(1)$.
- It is because, in bubble sort, an extra variable is required for swapping.

- The space complexity of optimized bubble sort is $O(2)$.
- It is because two extra variables are required in optimized bubble sort.

HEAP SORT ALGORITHM

- Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array.
- Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.
- Heap sort basically recursively performs two main operations -
 - ✓ Build a heap H, using the elements of array.
 - ✓ Repeatedly delete the root element of the heap formed in 1st phase.

What is a heap?

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

What is heap sort?

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

Algorithm

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
    heap_size[arr] = heap_size[arr] - 1
    MaxHeapify(arr,1)
End
```

BuildMaxHeap(arr)

```
BuildMaxHeap(arr)
    heap_size(arr) = length(arr)
    for i = length(arr)/2 to 1
        MaxHeapify(arr,i)
    End
```

MaxHeapify(arr,i)

```
MaxHeapify(arr,i)
L = left(i)
R = right(i)
if L > heap_size[arr] and arr[L] > arr[i]
    largest = L
else
```

```

largest = i
if R > heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr, largest)
End

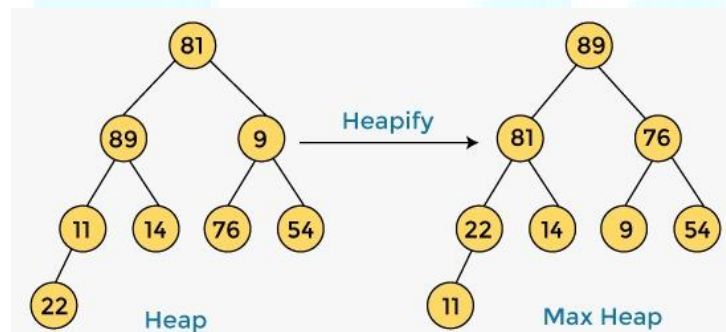
```

Working of Heap sort Algorithm

- In heap sort, basically, there are two phases involved in the sorting of elements.
- By using the heap sort algorithm, they are as follows -
- ✓ The first step includes the creation of a heap by adjusting the elements of the array.
- ✓ After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.
- Now let's see the working of heap sort in detail by using an example.
- To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

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

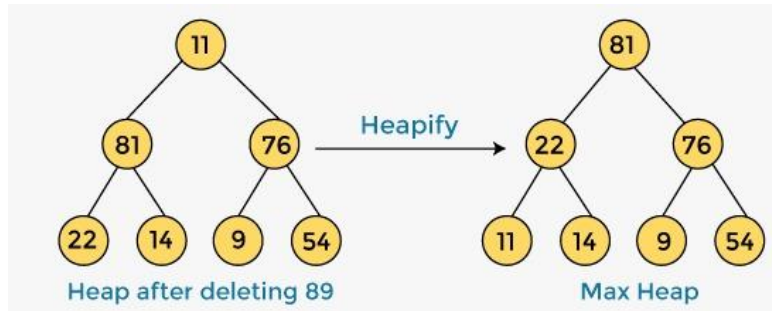
- First, we have to construct a heap from the given array and convert it into max heap.



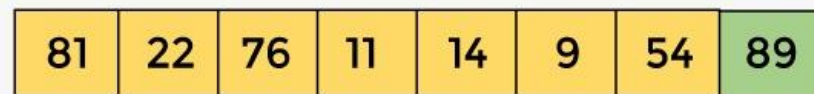
- After converting the given heap into max heap, the array elements are –

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

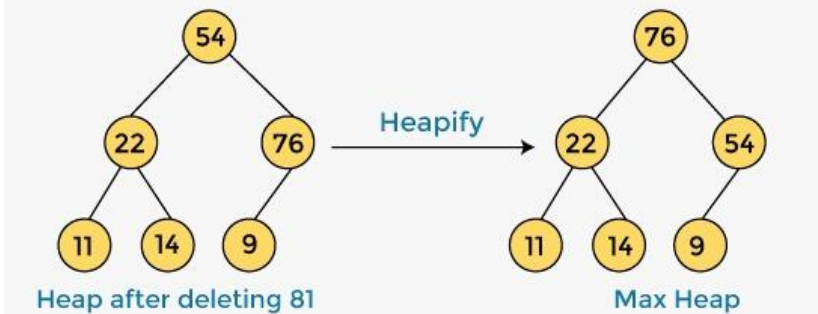
- Next, we have to delete the root element (89) from the max heap.
- To delete this node, we have to swap it with the last node, i.e. (11).
- After deleting the root element, we again have to heapify it to convert it into max heap.



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



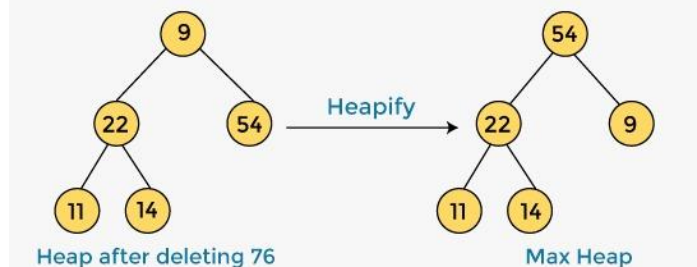
- In the next step, again, we have to delete the root element (81) from the max heap.
- To delete this node, we have to swap it with the last node, i.e. (54).
- After deleting the root element, we again have to heapify it to convert it into max heap.



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



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

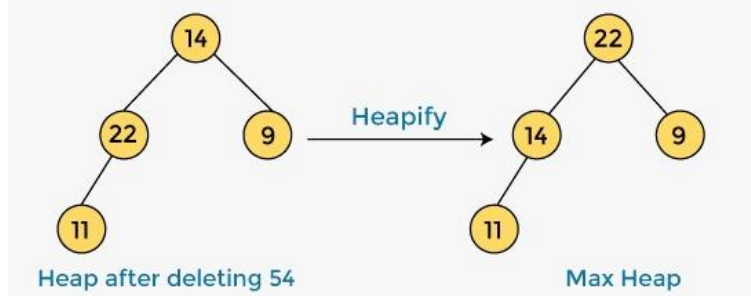


- After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are –



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

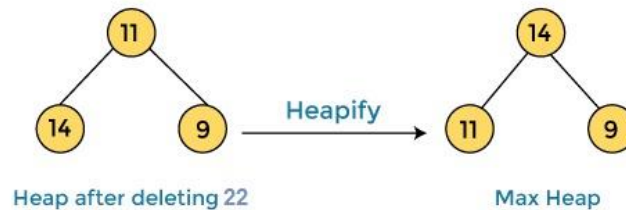
- After deleting the root element, we again have to heapify it to convert it into max heap.



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

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

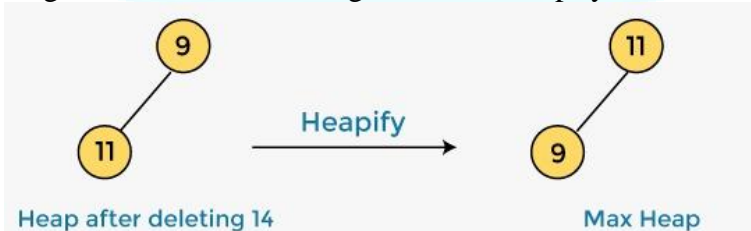
- In the next step, again we have to delete the root element (22) from the max heap.
- To delete this node, we have to swap it with the last node, i.e. (11).
- After deleting the root element, we again have to heapify it to convert it into max heap.



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

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

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



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

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

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



- After completion of sorting, the array elements are –

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

- Now, the array is completely sorted.

Heap sort complexity

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

1. Time Complexity

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

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

2.Space Complexity

Space Complexity	$O(1)$
Stable	No

The space complexity of Heap sort is $O(1)$.

GRAPH REPRESENTATIONS

- In graph theory, a graph representation is a technique to store graph into the memory of computer.
- To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge).
- If it is a weighted graph, then the weight will be associated with each edge.
- There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

1. Adjacency Matrix

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a $n \times n$ matrix A .
- If there is any edge from a vertex i to vertex j , then the corresponding element of A , $a^{ij} = 1$, otherwise $a^{ij} = 0$.

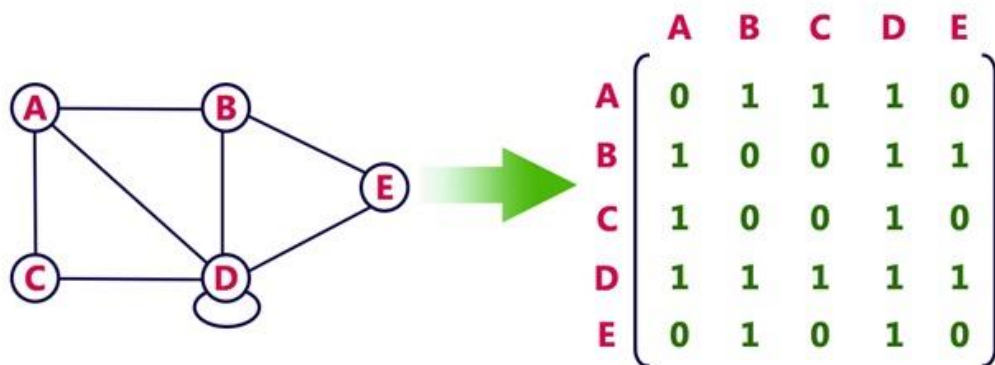
Note, even if the graph on 100 vertices contains only 1 edge, we still have to have a 100x100 matrix with lots of zeroes.

- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

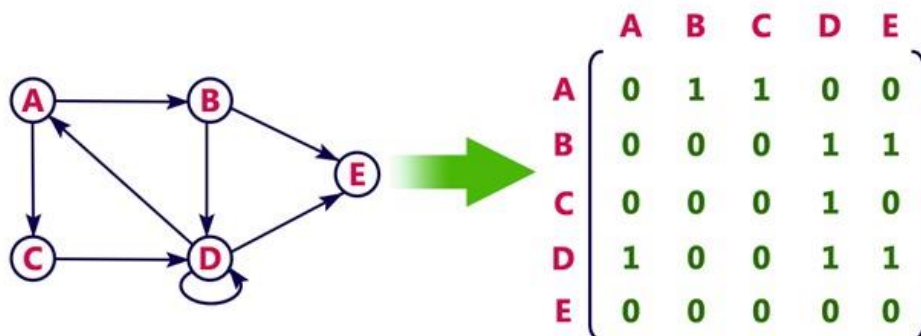
Example

Consider the following undirected graph representation:

Undirected graph representation

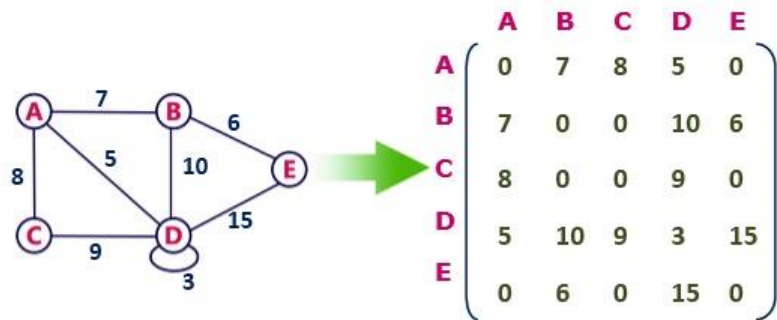


Directed graph representation



- In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.

Undirected weighted graph representation



Pros: Representation is easier to implement and follow.

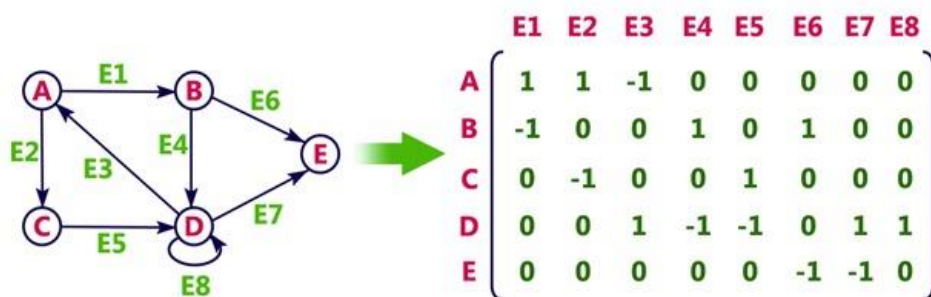
Cons: It takes a lot of space and time to visit all the neighbors of a vertex, we have to traverse all the vertices in the graph, which takes quite some time.

3. Incidence Matrix

- Total number of vertices by total number of edges.
- It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class.
- In this matrix, columns represent edges and rows represent vertices.
- This matrix is filled with either 0 or 1 or -1. Where,
 - ✓ 0 is used to represent row edge which is not connected to column vertex.
 - ✓ 1 is used to represent row edge which is connected as outgoing edge to column vertex.
 - ✓ -1 is used to represent row edge which is connected as incoming edge to column vertex.

Example

Consider the following directed graph representation.

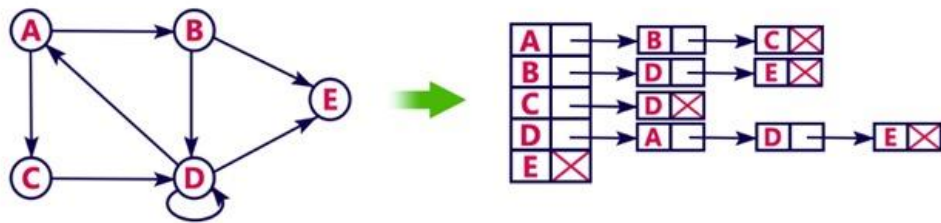


3. Adjacency List

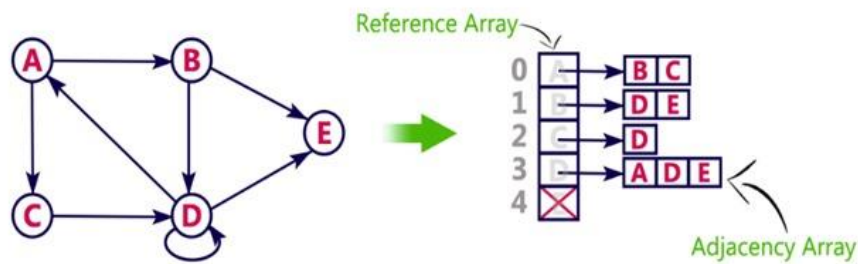
- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors.
- It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a singly linked list of neighbors of v.

Example

- Let's see the following directed graph representation implemented using linked list:



- We can also implement this representation using array as follows:



Pros:

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

Cons:

- The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

BFS ALGORITHM

- Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes.
- Then, it selects the nearest node and explores all the unexplored nodes.
- While using BFS for traversal, any node in the graph can be considered as the root node.
- There are many ways to traverse the graph, but among them, BFS is the most commonly used approach.
- It is a recursive algorithm to search all the vertices of a tree or graph data structure.
- BFS puts every vertex of the graph into two categories - visited and non-visited.
- It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Applications of BFS algorithm

- The applications of breadth-first-algorithm are given as follows -
- ✓ BFS can be used to find the neighboring locations from a given source location.
- ✓ In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.

- ✓ BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
- ✓ BFS is used to determine the shortest path and minimum spanning tree.
- ✓ BFS is also used in Cheney's technique to duplicate the garbage collection.
- ✓ It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Algorithm

- The steps involved in the BFS algorithm to explore a graph are given as follows -

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

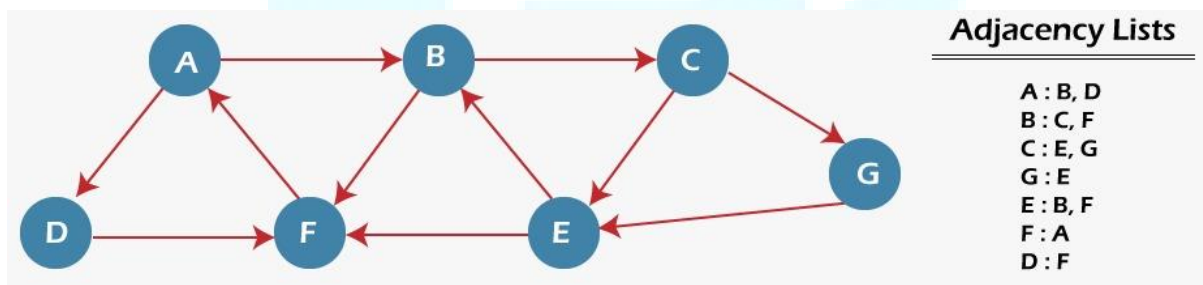
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example of BFS algorithm

- Now, let's understand the working of BFS algorithm by using an example.
- In the example given below, there is a directed graph having 7 vertices.



- In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E.
- The algorithm uses two queues, namely QUEUE1 and QUEUE2.
- QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.
- Now, let's start examining the graph starting from Node A.

Step 1 - First, add A to queue1 and NULL to queue2.

QUEUE1 = {A}

QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

QUEUE1 = {B, D}

QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

QUEUE1 = {D, C, F}
QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

QUEUE1 = {C, F}
QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

QUEUE1 = {F, E, G}
QUEUE2 = {A, B, D, C}

Step 5 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

QUEUE1 = {E, G}
QUEUE2 = {A, B, D, C, F}

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

QUEUE1 = {G}
QUEUE2 = {A, B, D, C, F, E}

Complexity of BFS algorithm

- Time complexity of BFS depends upon the data structure used to represent the graph.
- The time complexity of BFS algorithm is $O(V+E)$, since in the worst case, BFS algorithm explores every node and edge.
- In a graph, the number of vertices is $O(V)$, whereas the number of edges is $O(E)$.
- The space complexity of BFS can be expressed as $O(V)$, where V is the number of vertices.

DFS (DEPTH FIRST SEARCH) ALGORITHM

- It is a recursive algorithm to search all the vertices of a tree data structure or a graph.
- The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.
- Because of the recursive nature, stack data structure can be used to implement the DFS algorithm.
- The process of implementing the DFS is similar to the BFS algorithm.

- The step by step process to implement the DFS traversal is given as follows -
 - ✓ First, create a stack with the total number of vertices in the graph.
 - ✓ Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
 - ✓ After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
 - ✓ Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
 - ✓ If no vertex is left, go back and pop a vertex from the stack.
 - ✓ Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

- The applications of using the DFS algorithm are given as follows -
 - ✓ DFS algorithm can be used to implement the topological sorting.
 - ✓ It can be used to find the paths between two vertices.
 - ✓ It can also be used to detect cycles in the graph.
 - ✓ DFS algorithm is also used for one solution puzzles.
 - ✓ DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Pseudocode

DFS(G,v) (v is the vertex where the search starts)

Stack S := { }; (start with an empty stack)

for each vertex u, set visited[u] := false;

push S, v;

while (S is not empty) do

u := pop S;

if (not visited[u]) then

visited[u] := true;

for each unvisited neighbour w of u

push S, w;

end if

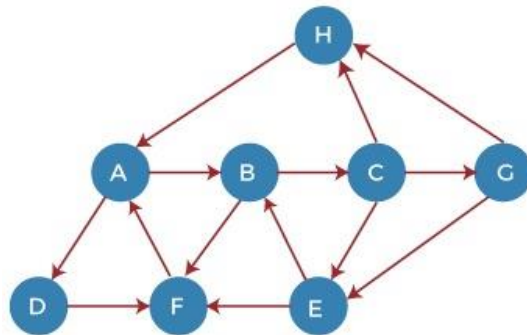
end while

END DFS()

Example of DFS algorithm

- Now, let's understand the working of the DFS algorithm by using an example.

- In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

```
A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A
```

- Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.
STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.
Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.
Print: A
STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.
Print: D
STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.
Print: F
STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.
Print: B
STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.
Print: C
STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.
Print: G

STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

Print: E

STACK:

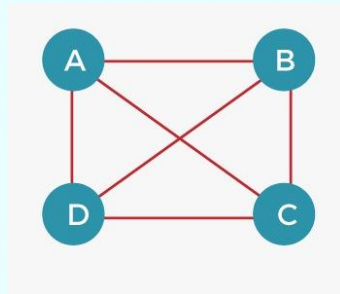
Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

- The time complexity of the DFS algorithm is $O(V+E)$, where V is the number of vertices and E is the number of edges in the graph.
- The space complexity of the DFS algorithm is $O(V)$.

MINIMUM SPANNING TREE

- Before knowing about the minimum spanning tree, we should know about the spanning tree.
- To understand the concept of spanning tree, consider the below graph:



- The above graph can be represented as $G(V, E)$, where ' V ' is the number of vertices, and ' E ' is the number of edges.
- The spanning tree of the above graph would be represented as $G'(V', E')$.
- In this case, $V' = V$ means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph, but the number of edges would be different.
- The number of edges in the spanning tree is the subset of the number of edges in the original graph. Therefore, the number of edges can be written as:

$$E' \subseteq E$$

It can also be written as:

$$E' = |V| - 1$$

- Two conditions exist in the spanning tree, which is as follows:

- ✓ The number of vertices in the spanning tree would be the same as the number of vertices in the original graph.

$$V' = V$$

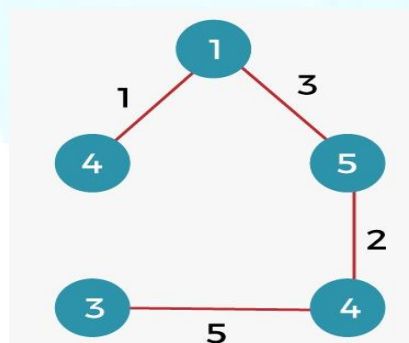
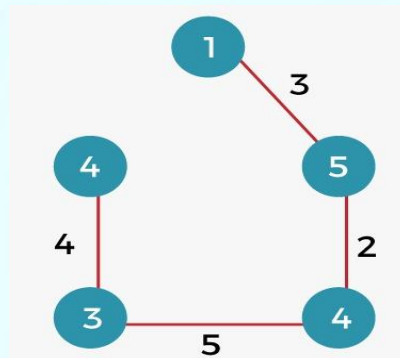
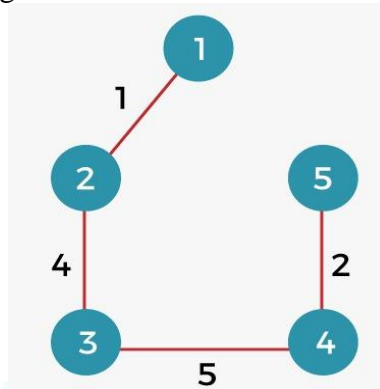
- ✓ The number of edges in the spanning tree would be equal to the number of edges minus 1.

$$E' = |V| - 1$$

- The spanning tree should not contain any cycle.
- The spanning tree should not be disconnected.

Note: A graph can have more than one spanning tree.

- Consider the below graph:
- The above graph contains 5 vertices.
- As we know, the vertices in the spanning tree would be the same as the graph; therefore, V is equal 5.
- The number of edges in the spanning tree would be equal to $(5 - 1)$, i.e., 4. The following are the possible spanning trees:



What is a minimum spanning tree?

- The minimum spanning tree is a spanning tree whose sum of the edges is minimum.
- Consider the below graph that contains the edge weight:
- The following are the spanning trees that we can make from the above graph.
- ✓ The first spanning tree is a tree in which we have removed the edge between the vertices 1 and 5 shown as below:
The sum of the edges of the above tree is $(1 + 4 + 5 + 2)$: 12

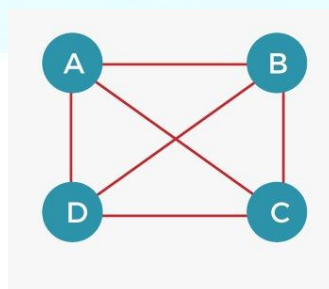
- ✓ The second spanning tree is a tree in which we have removed the edge between the vertices 1 and 2 shown as below:
The sum of the edges of the above tree is $(3 + 2 + 5 + 4) : 14$
- ✓ The third spanning tree is a tree in which we have removed the edge between the vertices 2 and 3 shown as below:
The sum of the edges of the above tree is $(1 + 3 + 2 + 5) : 11$
- ✓ The fourth spanning tree is a tree in which we have removed the edge between the vertices 3 and 4 shown as below:
The sum of the edges of the above tree is $(1 + 3 + 2 + 4) : 10$.
The edge cost 10 is minimum so it is a minimum spanning tree.

General properties of minimum spanning tree:

- ✓ If we remove any edge from the spanning tree, then it becomes disconnected. Therefore, we cannot remove any edge from the spanning tree.
- ✓ If we add an edge to the spanning tree then it creates a loop. Therefore, we cannot add any edge to the spanning tree.
- ✓ In a graph, each edge has a distinct weight, then there exists only a single and unique minimum spanning tree. If the edge weight is not distinct, then there can be more than one minimum spanning tree.
- ✓ A complete undirected graph can have an $n(n-2)$ number of spanning trees.
- ✓ Every connected and undirected graph contains atleast one spanning tree.
- ✓ The disconnected graph does not have any spanning tree.
- ✓ In a complete graph, we can remove maximum $(e-n+1)$ edges to construct a spanning tree.

Example

- Consider the complete graph which is given below:



- The number of spanning trees that can be made from the above complete graph equals to $n(n-2) = 4(4-2) = 8$.
- Therefore, 8 spanning trees can be created from the above graph.
- The maximum number of edges that can be removed to construct a spanning tree equals to $e-n+1 = 6 - 4 + 1 = 3$.