Data Structure

Queue

1. **Define queue with examples and applications.**

   Answer: A Queue is a linear list of elements in which deletions can take place only at one end, called the *front*, and insertions can take place only at the other end, called the *rear*. The terms “front” and “rear” are used in describing a linear list only when it implemented as a queue.

Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are Last-in First-out (LIFO) lists.

Examples:

- The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through.
- The people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on.

- An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed.
- Queue for printing purposes.

<u>Some common applications of Queue data structure/advantages :</u>

1. **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.

2. **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.

3. **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

4. **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

5. **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.

6. **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.

7. **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.

8. **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.

9. **Printer queues:** In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.

10. **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.

11. **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

2. Queue operations.

Answer:

Operations

1. Create queue.
2. Identify either queue is empty.
3. Add new item in queue.
4. Delete item from queue.
5. Call first item in queue.

3. Queue types in data structure.

Answer: In data structures, there are several types of queues that are commonly used to manage elements in a particular order. Here are some of the most commonly used queue types:

Linear Queue: Also known as a simple queue, it follows a First-In-First-Out (FIFO) order, where the element that has been in the queue the longest is processed first, and new elements are added at the rear and removed from the front.

Circular Queue: It is similar to a linear queue, but the rear and front pointers wrap around to the beginning of the queue when they reach the end. This allows for efficient space utilization and avoids the need to shift elements when the queue is full.

Priority Queue: Unlike linear and circular queues, a priority queue assigns a priority value to each element and processes elements based on their priority. Elements with higher priority are processed before elements with lower priority, regardless of the order in which they were added.

Double-Ended Queue (Deque): A deque allows insertion and deletion of elements from both ends. It can function as both a queue and a stack, providing operations like enqueue, dequeue, push, and pop.

Priority Deque: This is a combination of a priority queue and a deque. It allows elements to be inserted and removed from both ends, and the elements are processed based on their priority value.

These are some of the commonly used types of queues in data structures. Depending on the specific requirements and use cases, other variations or implementations of queues may also exist.

4. Define circular queue.

Answer: A circular queue, also known as a circular buffer, is a type of queue data structure that uses a fixed-size array or a circular linked list to efficiently manage elements in a circular manner. In a circular queue, the rear and front pointers wrap around to the beginning of the queue when they reach the end, creating a circular behavior.

The circular nature of the queue allows for efficient space utilization and avoids the need to shift elements when the queue becomes full. It also eliminates the problem of wasting space at the front end of the queue when elements are dequeued, as the front pointer wraps around to the beginning.

Here are some key features and operations of a circular queue:

Initialization: A circular queue is initialized by setting the front and rear pointers to -1 or 0 (depending on the implementation) to indicate an empty queue.

Enqueue: Adding an element to the rear of the circular queue. The rear pointer is incremented, and the element is placed at the rear index. If the rear pointer reaches the end of the queue, it wraps around to the beginning if there is space available.

Dequeue: Removing an element from the front of the circular queue. The front pointer is incremented, and the element at the front index is removed. If the front pointer reaches the end of the queue, it wraps around to the beginning if there are still elements in the queue.

IsFull: Checks if the circular queue is full by comparing the positions of the front and rear pointers. If the next position of the rear pointer is the same as the front pointer, the queue is full.

IsEmpty: Checks if the circular queue is empty by comparing the positions of the front and rear pointers. If they are the same, the queue is empty.

Peek: Returns the element at the front of the circular queue without removing it.

A circular queue provides efficient operations for both enqueue and dequeue, making it suitable for scenarios where elements are continuously added and removed in a cyclic manner.

5. Limitations of linear queue.

Answer: Linear queues, also known as simple queues, have certain limitations that can affect their performance and usage in specific scenarios. Here are some of the limitations of linear queues:

Fixed Size: Linear queues often have a fixed size determined at the time of creation. This limitation can lead to two main issues. First, if the queue becomes full, it cannot accommodate any additional elements, even if there is available memory. Second, if the queue becomes empty, the memory allocated for the fixed size queue remains unused, resulting in inefficient memory utilization.

Wasted Memory: In a linear queue, once an element is dequeued, the memory space previously occupied by that element remains unused until a new element is enqueued in that position. This can lead to inefficient memory usage, especially if the queue experiences frequent enqueue and dequeue operations.

Linear Time Complexity: The enqueue and dequeue operations in a linear queue have a linear time complexity of O(1) and O(n), respectively, where n is the number of elements in the queue. This means that as the number of elements increases, the dequeue operation takes longer. If the queue is large and elements are frequently dequeued, it can impact the overall performance.

Inefficient Removal of Middle Elements: Linear queues are designed to remove elements from the front. If there is a need to remove elements from the middle of the queue, it requires dequeuing and re-enqueuing the subsequent elements until the desired element is reached. This process can be time-consuming and inefficient, especially for large queues.

No Priority Handling: Linear queues follow a First-In-First-Out (FIFO) order, meaning the elements are processed in the order they were enqueued. There is no inherent support for prioritizing certain elements over others. If there is a need to handle elements based on priority, a linear queue alone may not be sufficient.

These limitations of linear queues highlight the need for alternative queue types, such as circular queues or priority queues, depending on the specific requirements and constraints of the application.

6. QINSERT Algorithm/routine to insert a element from a queue:

QINSERT(QUEUE,N,FRONT,REAR,ITEM)

This procedure inserts an element ITEM into a queue.

```
1. [Queue already fill]
If FRONT=1 and REAR=N, or if FRONT=REAR+1, then :
     Write: Overflow, and Return.
2. [Find new value of REAR]
     If FRONT:=NULL, then: [Queue initially empty]
            Set FRONT := 1 and REAR := 1
    Else if REAR =N then
        Set REAR:=1
  Else Set REAR:=REAR+1
  [End of if structure]
3. Set  QUEUE[REAR]:=ITEM [This inserts new element]
4. Return.
```

Explanation:

Check if the queue is already full.

If FRONT is equal to 1 and REAR is equal to N, or if FRONT is equal to REAR + 1, it means the queue is full.
Display "Overflow" (indicating that the queue is unable to accept more elements) and exit the procedure.
If the queue is initially empty (i.e., FRONT is uninitialized or set to NULL):
Set FRONT to 1 (indicating the queue starts from index 1) and REAR to 1 (as it's the only element in the queue).
Else, if REAR is equal to N (the maximum size of the queue):
Set REAR to 1, effectively wrapping around to the beginning of the queue.
If neither of the above conditions are met, increment REAR by 1 to move it to the next available position.Set the element at the position pointed to by REAR in the queue (QUEUE[REAR]) to the value of ITEM. This step effectively inserts the new element into the queue at the position specified by the updated REAR.
Return:
Exit the procedure after successfully inserting the element.
In summary, this algorithm outlines the process of inserting an element into a circular queue. It first checks if the queue is full, then updates the rear pointer to the appropriate position for insertion. Finally, it inserts the new element at the updated rear position and exits the procedure. The circular nature of the queue allows it to wrap around when the rear pointer reaches the maximum size of the queue.

7. QDELETE Algorithm:

QDELETE(QUEUE,N,FRONT,REAR,ITEM)

This procedure deletes an element from the queue and assigns it to the variable

```
1. [Queue already empty]
 If FRONT := NULL, then Write: Underflow, and Return.
2. Set ITEM := QUEUE[FRONT]
3. [Find new value of FRONT]
    If FRONT=REAR, then: [Queue  has only one element to start]
 Set FRONT := NULL and REAR := NULL
    Else if FRONT = N then
  Set FRONT := 1
    Else Set FRONT := FRONT +1
```

```
   [End of if structure]
4. Return.
```

Explanation:

1. The delete from queue() function takes the queue as an argument.
2. It first checks if the queue is empty using if not queue. If the queue is empty, it prints a message indicating that there are no elements to delete.
3. If the queue is not empty, it uses the pop(0) method to remove the element at the front of the queue and assigns it to the deleted element variable.
4. It then prints the deleted element.
5. Finally, it displays the updated queue after deletion.

8. What is a dequeue? Explain its operation with exapmle.

Answer: A deque, also known as a double-ended queue, is a linear data structure that allows elements to be inserted and removed from both ends. The name "double-ended queue" suggests that it is a queue with two ends, where elements can be added or removed from either the front or the rear. There are two types of deque in data structure:

1**. Input restricted queue:** In an input restricted queue, the data can be inserted from only one end, while it can be deleted from both the ends.

2. **Output restricted queue**: In an Output restricted queue, the data can be deleted from only one end, while it can be inserted from both the ends.

The main operations supported by a deque are as follows:

1. **EnqueueFront:** Adds an element to the front of the dequeue.
   Example:
   Deque[2,3,4]
   EnqueueFront(1)
   Deque after Enqueue [1,2,3,4]

2. **EnqueueRear:** Adds an element to the rear of the dequeue.
   Example:
   Deque:[1,2,3]
   EnqueueRear(4)

Deque after enqueue: [1,2,3,4]

3. **DequeueFront:** Removes and returns the element from the front of the dequeue.
   Example:
   Deque [1,2,3,4]
   DequeueFront()
   Deque after dequeue [2,3,4]
   Returned element 1

4. **DequeueRear:** Removes and returns the element from the rear of the dequeue.
   Example:
   Deque [1,2,3,4]
   DequeueRear()
   Deque after dequeue [1,2,3]
   Returned element 4
5. **Front:** Returns the element at the front of the dequeue without removing it.
6. **Rear:** Returns the element at the rear of the dequeue without removing it.
7. **IsEmpty:** Checks if the deque is empty.
8. **Size:** Returns the number of elements currently stored in the dequeue.

The deque combines the features of a stack (last-in, first-out) and a queue (first-in, first-out), making it a versatile data structure. It allows efficient insertion and deletion at both ends, making it useful in various applications where elements need to be added or removed from the front or rear.

9.Define priority queue.

Answer: A priority queue is a type of queue that arranges elements based on their priority values. An element of higher priority is processed before any element of lower priority. Two elements with the same priority are processed according to the order in which they were added in the queue. The hospital emergency queue is an ideal real-life example of a priority queue. In this queue of patients, the patient with the most critical situation is the first in a queue, and the patient who doesn't need immediate medical attention will be the last. In this queue, the priority depends on the medical condition of the patients.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first and programs with the same priority form a standard queue. There are various way to maintaining a priority queue in memory. One is a one-way list, and the other is multiple queues.

Operations: Priority queues generally support the following operations:

Insertion: Adding an element along with its priority value to the queue.

Deletion: Removing the element with the highest (or lowest) priority from the queue.

Peek/Top: Accessing the element with the highest (or lowest) priority without removing it.

Update: Modifying the priority of an existing element.

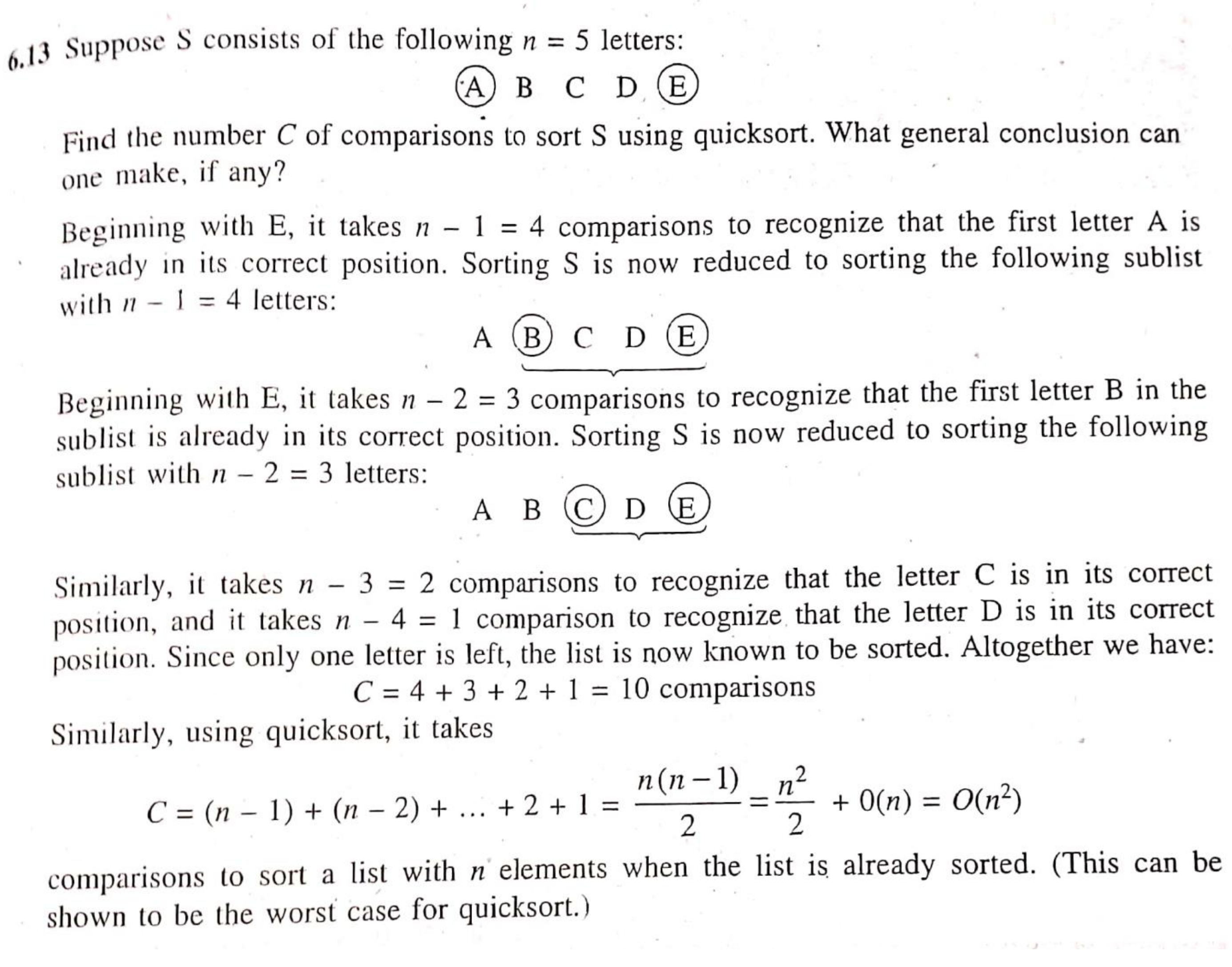
Quicksort

1. **Quicksort Algorithm:**

   This algorithm sorts an array A with N elements

```
1.  [Initialize] TOP:=NULL.
2.  [Push boundary values of A onto stack when A has 2 or more elements]
    If N>1, then TOP := TOP+1, LOWER[1]:=1 and UPPER[1]:=N.
3.  Repeat Steps 4 to 7  while TOP!= NULL.
4.  [Pop sub list from stacks.]
     Set BEG:=LOWER[TOP], END:=UPPER[TOP]
     TOP:=TOP-1.
5.  Call QUICK(A,N,BEG,END,LOC). [Procedure 6.5]
6.  [Push left sub list onto stacks when it has 2 or more elements]
    If BEG<LOC-1, then:
      TOP:=TOP+1, LOWER[TOP]:=BEG,
    UPPER[TOP]=LOC-1
     [End of If structure].
    7.  [Push right sub list onto stacks when it has 2 or more elements]
     If LOC+1 < END then:
     TOP:=TOP+1, LOWER[TOP]:= LOC+1,
     UPPER[TOP]:= END
    [End of If structure]
      [End of  Step 3 loop].
    8.  Exit
```

2.



6.13 Suppose S consists of the following $n = 5$ letters:

$$\textcircled{A}\ \ B\ \ C\ \ D\ \ \textcircled{E}$$

Find the number C of comparisons to sort S using quicksort. What general conclusion can one make, if any?

Beginning with E, it takes $n - 1 = 4$ comparisons to recognize that the first letter A is already in its correct position. Sorting S is now reduced to sorting the following sublist with $n - 1 = 4$ letters:

$$A\ \ \underbrace{\textcircled{B}\ \ C\ \ D\ \ \textcircled{E}}$$

Beginning with E, it takes $n - 2 = 3$ comparisons to recognize that the first letter B in the sublist is already in its correct position. Sorting S is now reduced to sorting the following sublist with $n - 2 = 3$ letters:

$$A\ \ B\ \ \underbrace{\textcircled{C}\ \ D\ \ \textcircled{E}}$$

Similarly, it takes $n - 3 = 2$ comparisons to recognize that the letter C is in its correct position, and it takes $n - 4 = 1$ comparison to recognize that the letter D is in its correct position. Since only one letter is left, the list is now known to be sorted. Altogether we have:

$$C = 4 + 3 + 2 + 1 = 10 \text{ comparisons}$$

Similarly, using quicksort, it takes

$$C = (n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + 0(n) = O(n^2)$$

comparisons to sort a list with n elements when the list is already sorted. (This can be shown to be the worst case for quicksort.)

3. Shows every possible steps to sort the given values using quicksort.
   65 70 75 80 85 60 55 50 45
   44 33 11 55 77 90 40 60 99 22 88 66

Linked List

- **Define Linked list.**
  Answer: The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node". This structure allows for dynamic memory allocation and efficient insertion and deletion operations. There are various types of linked lists, including singly linked lists, doubly linked lists, and circular linked lists, each with its own characteristics and use cases.

1. **Singly Linked List:** Single linked list is a sequence of elements in which every element has link to its next element in the sequence.In a singly linked list, each node contains two main components:
   Data: The value or element that the node holds.
   Next Pointer: A reference to the next node in the list.
   The last node in the list typically has its next pointer set to NULL to indicate the end of the list.

   Operations on Single Linked List

   - Insertion
   - Deletion
   - Display

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode→next = NULL and head = newNode.

Step 4 - If it is Not Empty then, set newNode→next = head and head = newNode.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1 - Create a newNode with given value and newNode → next as NULL.

Step 2 - Check whether list is Empty (head == NULL).

Step 3 - If it is Empty then, set head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6 - Set temp → next = newNode.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode → next = NULL and head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Check whether list is having only one node (temp → next == NULL)

Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)

Step 6 - If it is FALSE then set head = temp → next, and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 - Check whether list has only one Node (temp1 → next == NULL)

Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)

Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)

Step 7 - Finally, Set temp2 → next = NULL and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).

Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).

Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.

Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).

Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Advantages of Singly linked list:

Dynamic Memory Allocation: Singly linked lists allow for dynamic memory allocation, which means we can allocate memory for nodes as needed, without requiring a fixed amount of memory upfront.

Efficient Insertion and Deletion: Inserting or deleting a node in the middle of a singly linked list is relatively efficient. We can modify the next pointers of neighboring nodes to insert or delete a node, which takes constant time O(1) after you locate the insertion or deletion point.

Constant-Time Insertions at the Head: Adding a new node at the beginning of the list (insertion at the head) can be done in constant time O(1), as you only need to update the next pointer of the new node to point to the current head and then update the head pointer.

Memory Efficiency: Singly linked lists consume less memory per node compared to doubly linked lists, as they only require one pointer (the next pointer) per node instead of two.

Sequential Access: Singly linked lists support sequential access, meaning you can easily traverse the list from the beginning to the end. This is useful when you need to process each element in order.

Simple Implementation: Singly linked lists have a simpler implementation compared to doubly linked lists because they only involve one pointer per node. This can make them easier to work with and debug in some cases.

Dynamic Data Structures: Singly linked lists are commonly used in cases where the data needs to be frequently inserted or removed, such as task scheduling, memory management, and certain algorithms like graph traversal.

Reduced Overhead: Since singly linked lists have only one pointer per node, they have less overhead compared to doubly linked lists, which have two pointers per node.

Disadvantages of singly Linked list:

Lack of Bidirectional Traversal: Singly linked lists only allow traversal in one direction, from the head to the end of the list. This limitation can be problematic when we need to traverse the list in reverse order or perform operations that require accessing both previous and next elements.

Inefficient Reverse Traversal: Traversing a singly linked list in reverse order requires extra time and space compared to doubly linked lists. To traverse in reverse, we need to either modify the structure of the list or use a stack or recursion, which can increase overhead.

Inefficient Deletion of Last Element: Deleting the last element in a singly linked list requires traversing the entire list to locate the second-to-last element, which can result in linear time complexity O(n) for deletion at the end.

Limited Access to Previous Element: Without a reference to the previous node, operations like inserting a node at a specific position (without traversing from the head) or deleting a node efficiently require additional steps or traversal.

Extra Memory Overhead for Next Pointers: While singly linked lists have less memory overhead compared to doubly linked lists, they still require an extra memory location (the next pointer) for each node, which can be significant when dealing with large lists.

Complexity of Insertions/Deletions in the Middle: While inserting or deleting a node at the head or tail of the list is efficient, performing such operations in the middle of the list requires locating the previous node, which can lead to more complex and less efficient code compared to other data structures like arrays or linked lists with more features.

Not Ideal for Searching and Accessing Arbitrary Elements: Singly linked lists are not the best choice when we need to frequently search for or access elements at arbitrary positions in the list. This is because we need to traverse from the head to the desired position, resulting in linear time complexity.

2. **Doubly Linked list:** Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In a doubly linked list, each node contains three main components:
Data: The value or element that the node holds.
Next Pointer: A reference to the next node in the list.
Previous Pointer: A reference to the previous node in the list.

Operations on Double Linked List

In a double linked list, we perform the following operations...

- Insertion

- Deletion
- Display

Insertion:

In a double linked list, the insertion operation can be performed in three ways as follows...

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

Step 1 - Create a newNode with given value and newNode → previous as NULL.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, assign NULL to newNode → next and newNode to head.

Step 4 - If it is not Empty then, assign head to newNode → next and newNode to head.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

Step 1 - Create a newNode with given value and newNode → next as NULL.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty, then assign NULL to newNode → previous and newNode to head.

Step 4 - If it is not Empty, then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6 - Assign newNode to temp → next and temp to newNode → previous.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, assign NULL to both newNode → previous & newNode → next and set newNode to head.

Step 4 - If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.

Step 5 - Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.

Step 7 - Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Check whether list is having only one node (temp → previous is equal to temp → next)

Step 5 - If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)

Step 6 - If it is FALSE, then assign temp → next to head, NULL to head → previous and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Check whether list has only one Node (temp → previous and temp → next both are NULL)

Step 5 - If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)

Step 6 - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

Step 7 - Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4 - Keep moving the temp until it reaches to the exact node to be deleted or to the last node.

Step 5 - If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free(temp)).

Step 8 - If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).

Step 9 - If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.

Step 10 - If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).

Step 11 - If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).

Step 12 - If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next = temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty, then display 'List is Empty!!!' and terminate the function.

Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4 - Display 'NULL <--- '.

Step 5 - Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node

Step 6 - Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

The advantages of a doubly linked list:

Bidirectional Traversal: Doubly linked lists allow for traversal in both directions, from the head to the tail and vice versa. This bidirectional traversal is a significant advantage over singly linked lists and makes it easier to perform operations that require accessing elements in reverse order or navigating both forward and backward through the list.

Efficient Insertions and Deletions: Inserting or deleting a node in the middle of a doubly linked list is more efficient compared to a singly linked list. Since each node has references to both the previous and next nodes, inserting or deleting a node only requires adjusting the adjacent nodes' pointers, resulting in constant time complexity O(1) after locating the insertion or deletion point.

Efficient Deletion of Last Element: Deleting the last element in a doubly linked list is efficient, as we have direct access to the previous node. This allows for constant-time deletion at the end of the list without traversing the entire list.

Flexibility in Operations: Doubly linked lists are more flexible than singly linked lists for various operations like inserting or deleting nodes at arbitrary positions, which may be required in certain algorithms or applications.

Efficient Searching from Both Ends: Searching for an element can be more efficient when we have the option to start searching from either end of the list. This can be advantageous in certain algorithms or use cases.

Backward Iteration: The bidirectional traversal of a doubly linked list makes it convenient for iterating backward through the list. This can be useful in scenarios where you need to process elements in reverse order.

Easier Reversal: Reversing a doubly linked list is more straightforward than reversing a singly linked list, as we can simply swap the next and prev pointers of each node.

Smoother Implementation of Certain Algorithms: Doubly linked lists are often used in algorithms that require traversal in both directions, such as cache management, text editing, and simulations.

Managing Circular Structures: When dealing with circular structures (like circular queues), doubly linked lists can be used to efficiently manage the circular relationships and enable bidirectional traversal.

Disadvantages of doubly linked list:

Higher Memory Consumption: Doubly linked lists consume more memory compared to singly linked lists due to the additional memory required for storing both the next and previous pointers in each node.

Increased Complexity: The presence of both next and previous pointers in each node increases the complexity of implementing and managing doubly linked lists, making them more error-prone and harder to debug compared to singly linked lists.

Slower Insertion and Deletion at Both Ends: While doubly linked lists excel at insertion and deletion in the middle of the list, they are less efficient than singly linked lists for inserting or deleting nodes at the beginning or end. This is because both the next and previous pointers of neighboring nodes need to be adjusted.

Extra Pointer Manipulation: Inserting or deleting nodes in a doubly linked list requires adjusting both next and previous pointers of the affected nodes. This extra pointer manipulation can lead to more complex code and potential bugs.

Greater Complexity for Circular Lists: Implementing and maintaining circular doubly linked lists can be more intricate compared to circular singly linked lists due to the need to update both next and previous pointers.

Cache Locality Concerns: Doubly linked lists can result in even poorer cache utilization compared to singly linked lists, as the additional pointers lead to more scattered memory access patterns.

Less Memory Locality: In doubly linked lists, each node points to both its predecessor and successor, potentially leading to worse memory locality compared to singly linked lists.

Inefficiency for Simple Structures: For simple use cases where only one-directional traversal is needed, using a doubly linked list introduces unnecessary complexity and memory overhead.

Less Suitable for Some Algorithms: Algorithms that do not require bidirectional traversal or do not benefit from it might be less efficient when implemented using a doubly linked list.

Not Always Ideal for Reversal: While reversing a doubly linked list might be simpler compared to reversing a singly linked list, there are still other data structures (like arrays) that offer more straightforward methods for reversing data.

Limited Hardware Support: Some modern hardware architectures are optimized for contiguous memory access, which can make doubly linked lists less efficient in certain computing environments.

Inefficient Memory Usage for Fixed-Size Data: For fixed-size data structures where the memory overhead of pointers is significant, using doubly linked lists can be wasteful.

3. **Circular Linked list:** A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element. This means that the elements in a circular linked list form a loop or circle, and there is no distinct "end" node. Circular linked lists come in both singly circular linked list and doubly circular linked list variations.

4. **Advantages of Linked list:**

   Dynamic Memory Allocation: Linked lists allow for dynamic memory allocation, making them suitable for scenarios where the size of the data structure is not known in advance or needs to change frequently during runtime.

   Efficient Insertions and Deletions: Linked lists excel in insertions and deletions, especially when performed at the beginning or middle of the list. These operations involve changing a few pointers, resulting in constant-time complexity O(1) after locating the insertion or deletion point.

   Constant-Time Insertion at Head: Adding a new node at the beginning of a linked list (insertion at the head) is a constant-time operation, requiring only pointer adjustments.

   Efficient Memory Usage for Inserts/Deletes: Unlike arrays, where resizing might involve copying elements to a new memory location, linked lists can insert or delete nodes without needing to move the entire list. This can be beneficial for scenarios requiring frequent insertions and deletions.

   Dynamic Data Structures: Linked lists are commonly used in scenarios where data needs to be frequently inserted, removed, or modified, such as task scheduling, memory allocation, and certain algorithms like graph traversal.

   Linked Structures for Advanced Data Structures: Linked lists serve as building blocks for more complex data structures like stacks, queues, hash tables, and graphs, enabling the implementation of efficient algorithms and data organization.

Memory Efficiency for Large Datasets: For very large datasets where memory allocation for the entire structure isn't feasible, linked lists can be used to manage data dynamically.

Simpler Implementation for Some Use Cases: In certain cases, linked lists can offer a simpler implementation compared to more complex data structures like trees or graphs, which require more intricate algorithms.

Circular Data Structures: Linked lists can be adapted to create circular structures, like circular queues, which are challenging to implement using arrays.

Separate Allocation for Nodes: Linked lists allow separate memory allocation for each node, which can be useful for managing different-sized elements or objects.

Indefinite Length: Linked lists can handle indefinite or varying lengths of data, accommodating growth or reduction as needed.

Ordered Data Storage: Linked lists can store data in a specific order, which might be advantageous for algorithms that rely on sorted or sequential data.

5. Limitations of linked list:

Memory Overhead: Linked lists require additional memory for storing pointers or references in each node. This overhead can become significant, especially when dealing with a large number of nodes, leading to higher memory consumption compared to arrays.

Slower Access Time: Accessing elements in a linked list is generally slower compared to arrays due to the need to traverse the list to find the desired element. This traversal results in linear time complexity O(n) for access operations.

Limited Random Access: Linked lists do not support efficient random access by index, as you need to traverse the list from the beginning or another known point to find the desired element.

Complexity of Implementation: Implementing linked lists requires careful management of pointers or references to ensure the structure remains consistent and doesn't lead to memory leaks or dangling pointers. This complexity can increase the likelihood of programming errors.

Fragmentation: Frequent insertions and deletions in linked lists can lead to memory fragmentation, where memory becomes scattered. This can negatively impact memory allocation efficiency.

Lack of Cache Locality: Linked lists can result in poor cache utilization due to the scattered memory locations of nodes, leading to slower traversal speeds compared to arrays.

Limited Hardware Support: Some modern hardware architectures are optimized for contiguous memory access, which can make linked lists less efficient in certain computing environments.

Inefficient for Sorting: Linked lists can be less efficient for sorting algorithms compared to arrays, as swapping elements requires adjusting pointers or references rather than simply swapping values.

No Built-in Search Index: Unlike some data structures like arrays or hash tables, linked lists do not inherently offer efficient ways to search for specific values or maintain an index.

Limited Memory Usage Awareness: Linked lists do not have built-in knowledge of memory constraints or memory usage patterns, which can lead to inefficient memory allocation in situations where memory is a critical resource.

Inefficient for Large Data Access: If large amounts of data need to be accessed together, linked lists can be less efficient due to the overhead of traversing the list.

Less Intuitive for Certain Operations: For some operations that naturally involve bidirectional traversal, such as undo/redo in text editors, a singly linked list might not be the most intuitive choice.

Less Suitable for Fixed-Size Structures: If the size of your data structure is known and won't change frequently, other data structures like arrays might be more efficient in terms of memory usage and access time**.**

6. Why linked list called one way list?

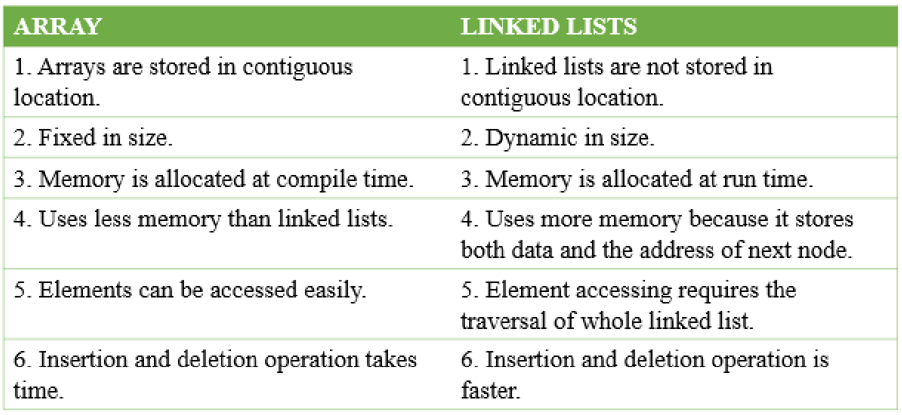
Answer: A linked list is often referred to as a "one-way list" or a "singly linked list" because it allows traversal in only one direction, typically from the head (the starting point) to the end of the list. This is in contrast to a "two-way list" or a "doubly linked list," which allows bidirectional traversal, meaning you can traverse in both directions—from the head to the tail and from the tail to the head.

In a singly linked list, each node contains a reference (pointer) to the next node in the sequence, forming a chain-like structure. This reference points to the next element in the list, allowing you to move from one element to the next, but you cannot directly move backward to the previous element without traversing the entire list from the beginning again.

On the other hand, a doubly linked list has each node containing references to both the next and the previous nodes, enabling bidirectional traversal. This makes it possible to move both forward and backward through the list.

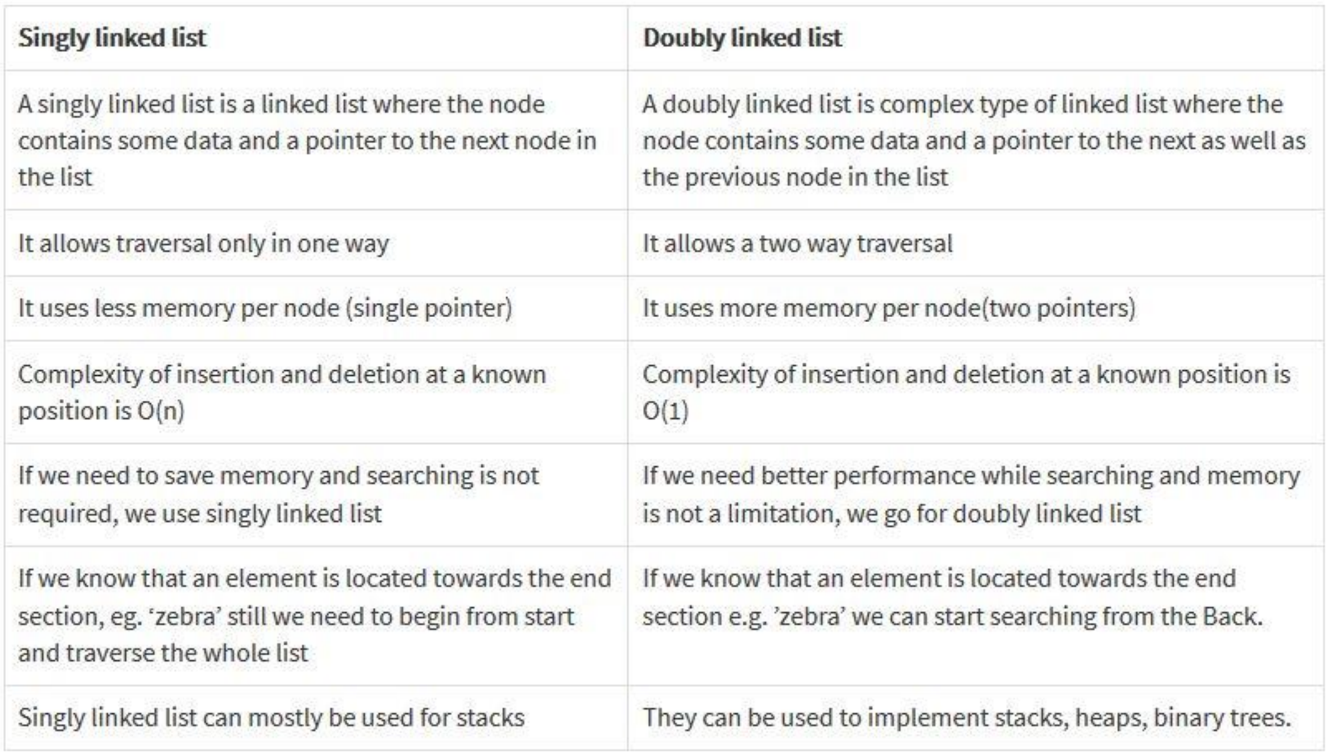
So, the term "one-way list" is a way of describing the unidirectional traversal characteristic of a singly linked list, whereas "two-way list" or "doubly linked list" refers to the bidirectional traversal capability of a linked list with both next and previous pointers.

7. Difference between array and linked list:



ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

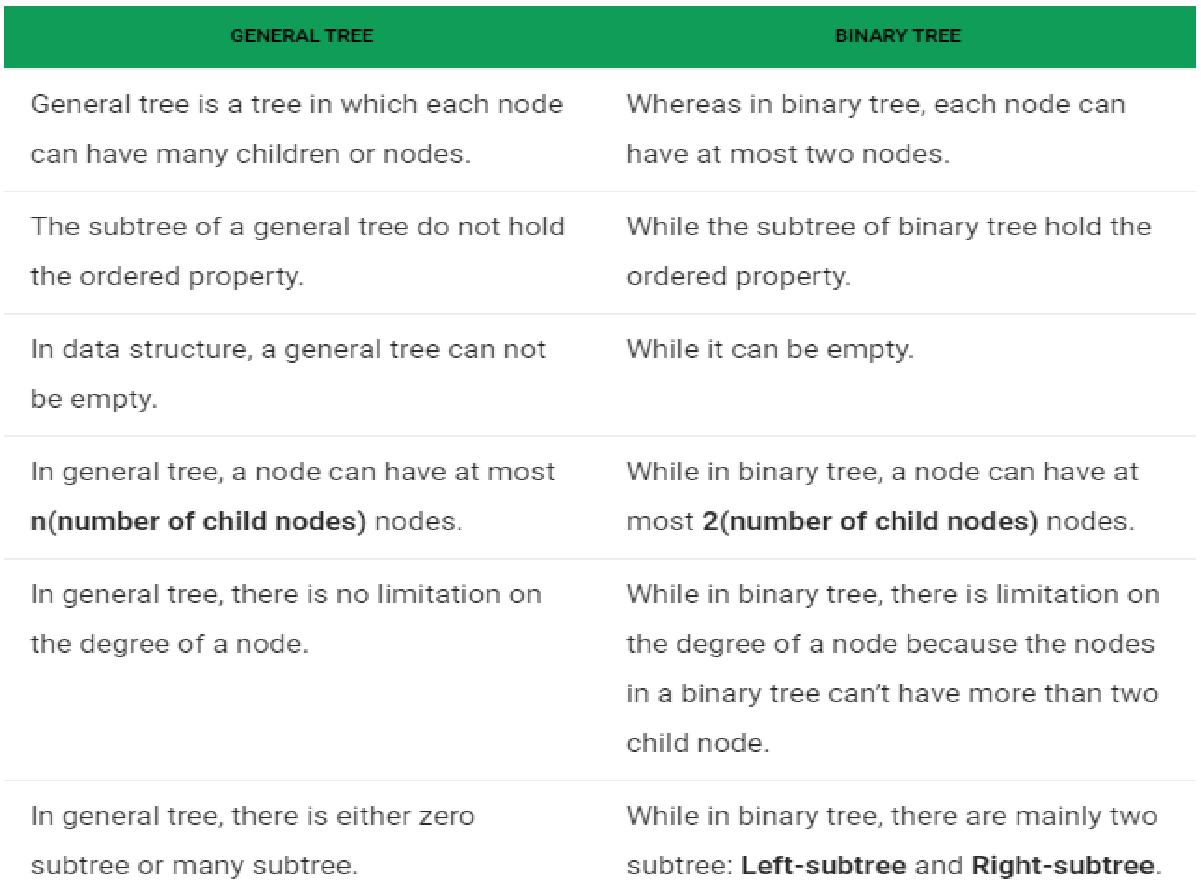
8. Difference between singly and doubly linked list:



Singly linked list	Doubly linked list
A singly linked list is a linked list where the node contains some data and a pointer to the next node in the list	A doubly linked list is complex type of linked list where the node contains some data and a pointer to the next as well as the previous node in the list
It allows traversal only in one way	It allows a two way traversal
It uses less memory per node (single pointer)	It uses more memory per node(two pointers)
Complexity of insertion and deletion at a known position is O(n)	Complexity of insertion and deletion at a known position is O(1)
If we need to save memory and searching is not required, we use singly linked list	If we need better performance while searching and memory is not a limitation, we go for doubly linked list
If we know that an element is located towards the end section, eg. 'zebra' still we need to begin from start and traverse the whole list	If we know that an element is located towards the end section e.g. 'zebra' we can start searching from the Back.
Singly linked list can mostly be used for stacks	They can be used to implement stacks, heaps, binary trees.

Trees

1. Difference between Tree and Binary tree:



GENERAL TREE	BINARY TREE
General tree is a tree in which each node can have many children or nodes.	Whereas in binary tree, each node can have at most two nodes.
The subtree of a general tree do not hold the ordered property.	While the subtree of binary tree hold the ordered property.
In data structure, a general tree can not be empty.	While it can be empty.
In general tree, a node can have at most **n(number of child nodes)** nodes.	While in binary tree, a node can have at most **2(number of child nodes)** nodes.
In general tree, there is no limitation on the degree of a node.	While in binary tree, there is limitation on the degree of a node because the nodes in a binary tree can't have more than two child node.
In general tree, there is either zero subtree or many subtree.	While in binary tree, there are mainly two subtree: **Left-subtree** and **Right-subtree**.

2. Difference between binary tree and binary search tree:

S. No.	Basis of Comparison	BINARY TREE	BINARY SEARCH TREE
1.	Definition	BINARY TREE is a nonlinear data structure where each node can have at most two child nodes.	BINARY SEARCH TREE is a node based binary tree that further has right and left subtree that too are binary search tree.
2.	Types	• Full binary tree • Complete binary tree • Extended Binary tree and more	• AVL tree • Splay Tree • T-trees and more
3.	Structure	In BINARY TREE there is no ordering in terms of how the nodes are arranged	In BINARY SEARCH TREE the left subtree has elements less than the nodes element and the right subtree has elements greater than the nodes element.
4.	Data Representation	Data Representation is carried out in a hierarchical format.	Data Representation is carried out in the ordered format.
5.	Duplicate Values	Binary trees allow duplicate values.	Binary Search Tree does not allow duplicate values.

S. No.	Basis of Comparison	BINARY TREE	BINARY SEARCH TREE
6.	Speed	The speed of deletion, insertion, and searching operations in Binary Tree is slower as compared to Binary Search Tree because it is unordered.	Because the Binary Search Tree has ordered properties, it conducts element deletion, insertion, and searching faster.
7.	Complexity	Time complexity is usually O(n).	Time complexity is usually O(logn).
8.	Application	It is used for retrieval of fast and quick information and data lookup.	It works well at element deletion, insertion, and searching.
9.	Usage	It serves as the foundation for implementing Full Binary Tree, BSTs, Perfect Binary Tree, and others.	It is utilized in the implementation of Balanced Binary Search Trees such as AVL Trees, Red Black Trees, and so on.

3. Trees are a fundamental data structure in computer science that are used to organize and store hierarchical data. There are several types of trees, each with its own structure and characteristics. Here are some of the most common types of trees:

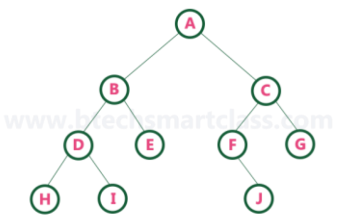
- Binary Tree: A binary tree is a tree in which each node has at most two children, referred to as the left child and the right child. Binary trees are widely used for their simplicity and efficiency in search, insertion, and deletion operations.

- Binary Search Tree (BST): A binary search tree is a type of binary tree where the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree contains only nodes with keys greater than the node's key. This ordering property allows for efficient searching and sorting.

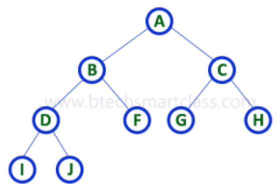
- Balanced Tree: A balanced tree is a type of tree in which the height of the left and right subtrees of any node differs by at most one. Balanced trees help maintain efficient search, insertion, and deletion operations.

- AVL Tree: An AVL tree is a type of balanced binary search tree in which the heights of the left and right subtrees of any node differ by at most one. AVL trees automatically balance themselves through rotations to ensure logarithmic time complexity for operations.

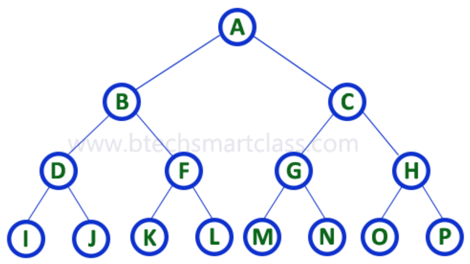
4. **Binary tree:** A tree in which every node can have a maximum of two children is called Binary Tree. In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.



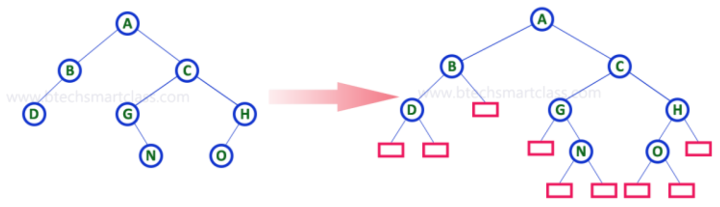
- **Full binary Tree:** A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. All nodes except leaf nodes have either 0 or 2 children. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.



- **Complete Binary Tree:** A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree. All nodes except for the level before the last must have 2 children. All nodes in the last level are as far left as possible.



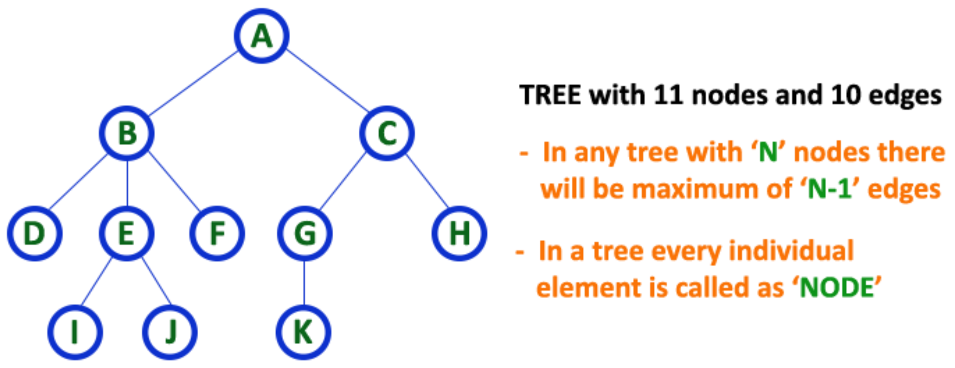
- **Extended Binary tree:** The full binary tree obtained by adding NULL nodes to a binary tree is called as Extended Binary Tree.



5. Terminology of trees:

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition. In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure. In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

Example

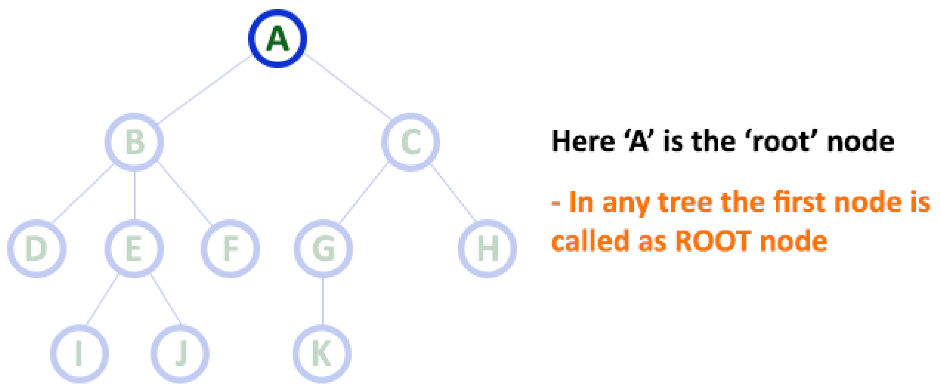

A
B
C
D
E
F
G
H
I
J
K
TREE with 11 nodes and 10 edges
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology...

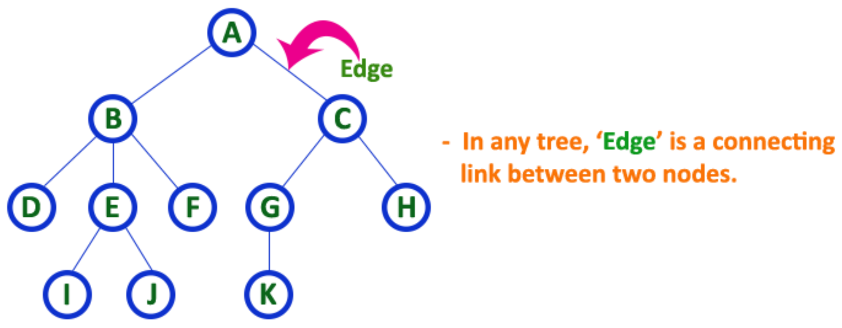
1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.


A
B
C
D
E
F
G
H
I
J
K
Here 'A' is the 'root' node
- In any tree the first node is called as ROOT node

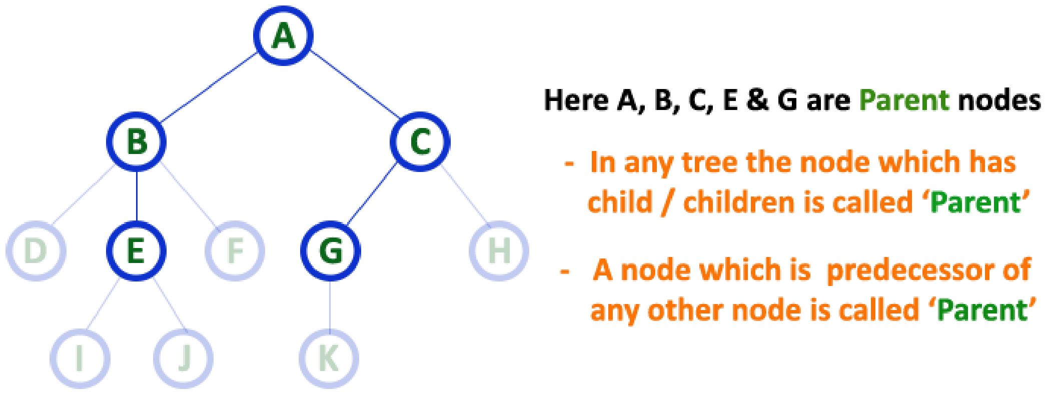
2. Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.


A
B
C
D
E
F
G
H
I
J
K
Edge
- In any tree, 'Edge' is a connecting link between two nodes.

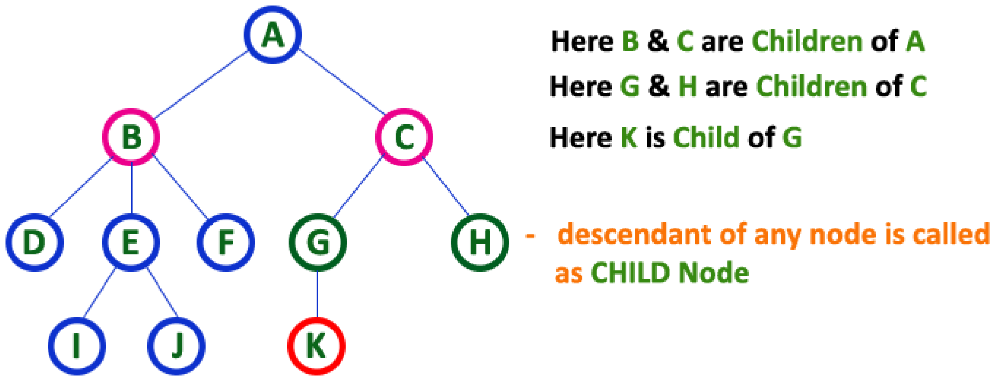
3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".


A
B
C
D
E
F
G
H
I
J
K
Here A, B, C, E & G are Parent nodes
- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

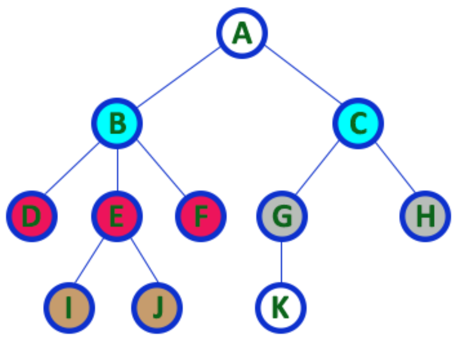
4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.


A
B
C
D
E
F
G
H
I
J
K
Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G
- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



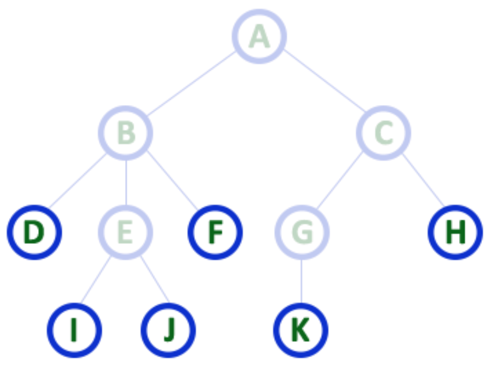
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



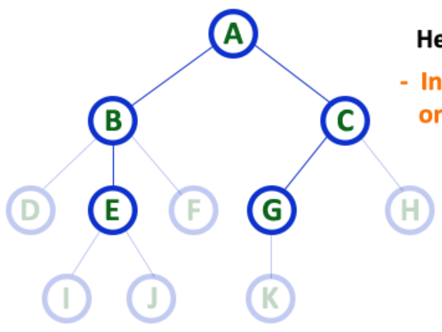
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.

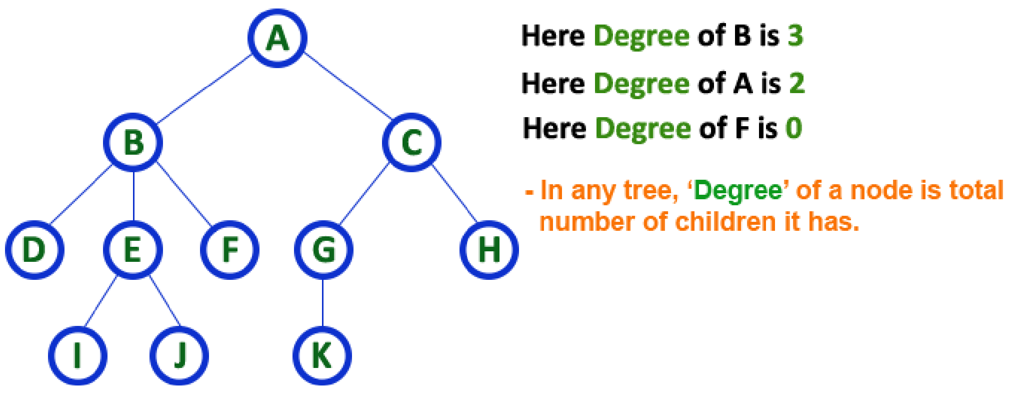


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

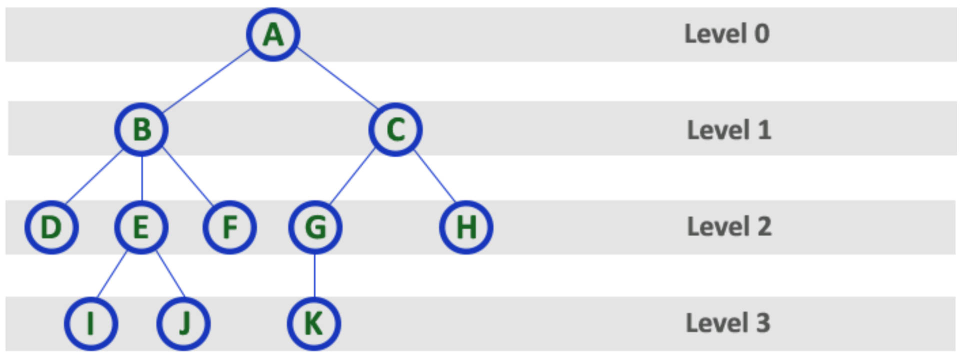
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



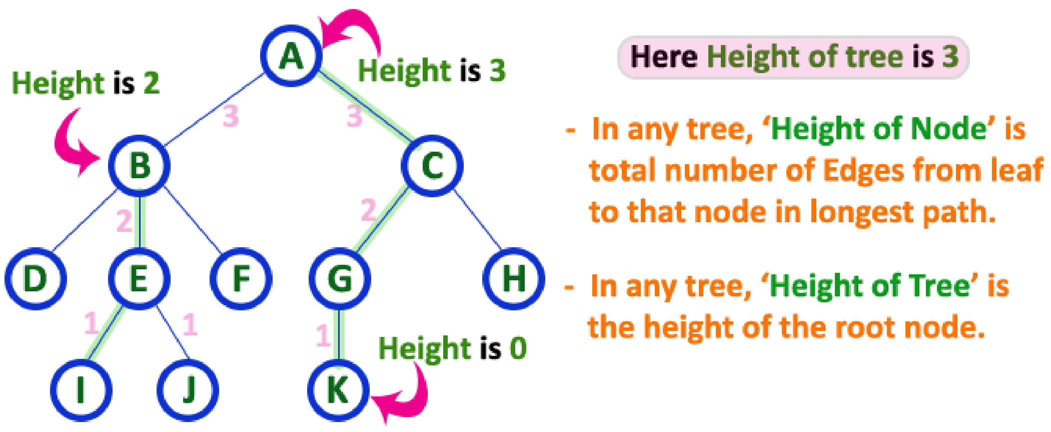
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

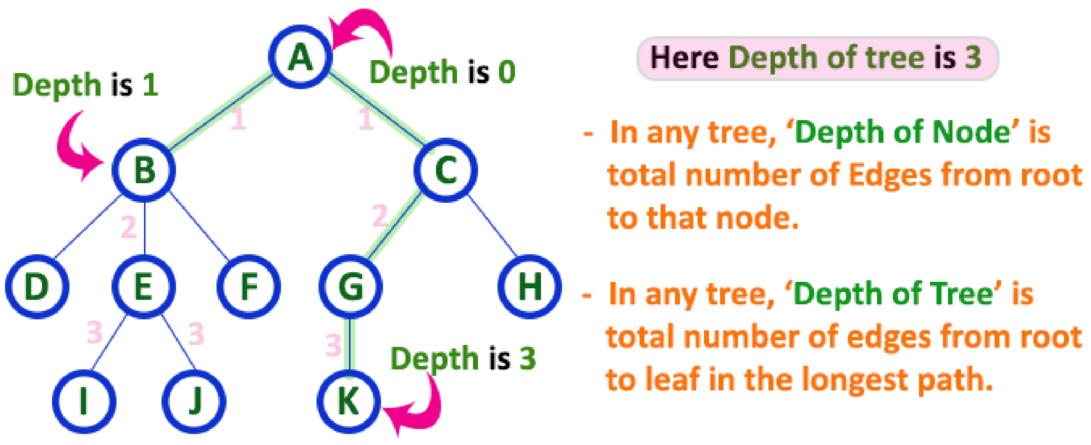
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. <u>In a tree, height of the root node is said to be **height of the tree**</u>. In a tree, <u>**height of all leaf nodes is '0'.**</u>


Height is 2
Height is 3
Height is 0
A
B
C
D
E
F
G
H
I
J
K
3
3
2
2
1
1
1
Here Height of tree is 3
- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

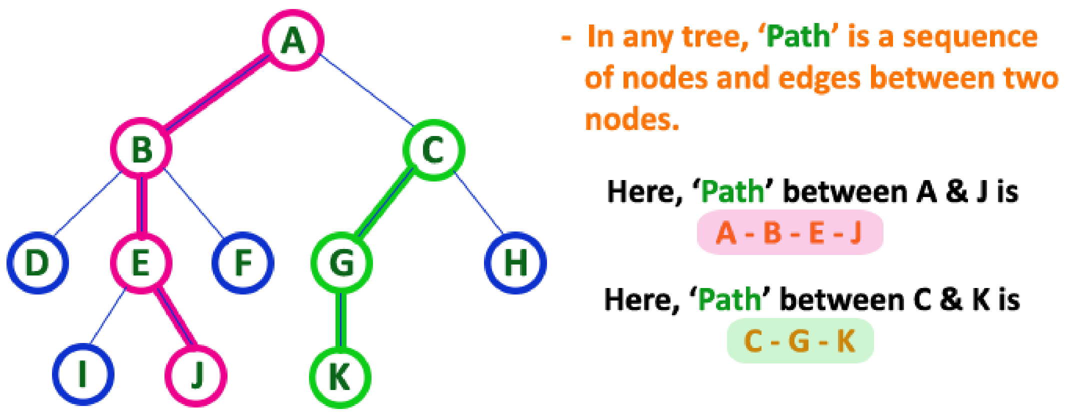
In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. <u>In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth**</u>

of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**


Depth is 1
A
Depth is 0
1
1
B
C
2
2
D
E
F
G
H
3
3
3
I
J
K
Depth is 3
Here Depth of tree is 3
- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

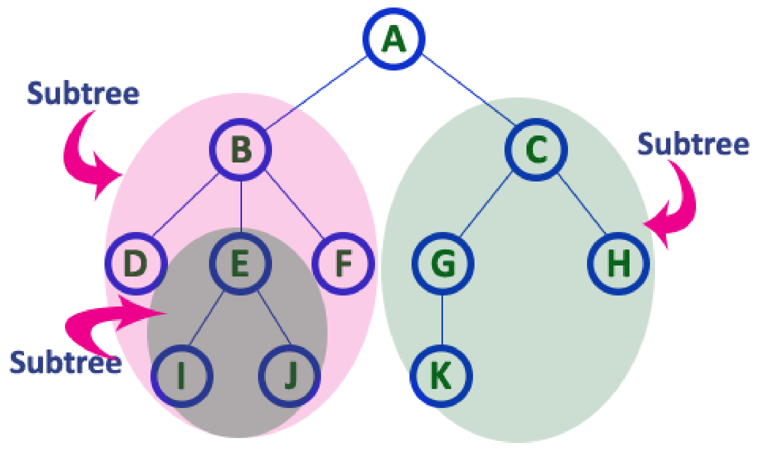
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.


A
B
C
D
E
F
G
H
I
J
K
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.
Here, 'Path' between A & J is
A - B - E - J
Here, 'Path' between C & K is
C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

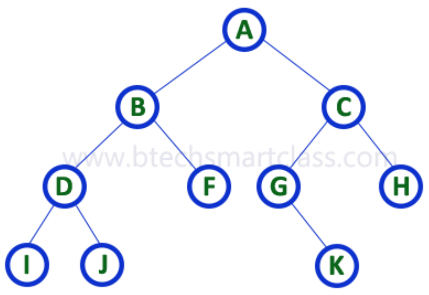


6. Binary tree representation:

A binary tree data structure is represented using two methods. Those methods are as follows...

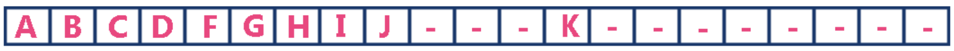
1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree. Consider the above example of a binary tree and it is represented as follows...

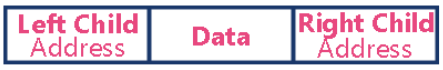


A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

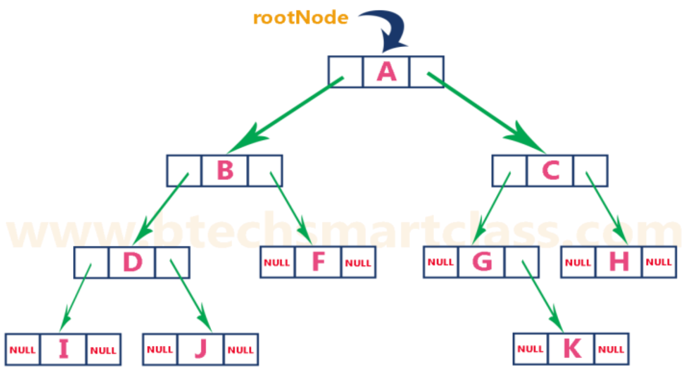
To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

2. Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...



The above example of the binary tree represented using Linked list representation is shown as follows...



7. Why Linked representation of binary tree best and most popular way?

Answer:

The linked representation of a binary tree is considered one of the best and most popular ways to represent binary trees for several reasons:

Memory Efficiency: Linked representation only uses memory proportional to the number of nodes in the tree, making it more memory-efficient compared to array-based representations that might allocate space for unused nodes.

Dynamic Structure: Linked representations are more dynamic and flexible, allowing for easy insertions and deletions of nodes without requiring resizing or rearranging memory, as might be needed in array-based representations.

Natural Representation: Binary trees are inherently hierarchical structures, and linked representation closely reflects this hierarchy. Each node directly points to its children, making traversal, manipulation, and understanding of the structure more intuitive.

Easier to Implement and Maintain: Linked representation requires less complicated logic and implementation compared to array-based representations, especially for tree operations like insertion and deletion.

Adaptability to Various Tree Types: Linked representation easily extends to various types of binary trees, including binary search trees, AVL trees, red-black trees, and more. Each node can store additional information specific to the type of binary tree.

Simplicity in Traversal: In linked representation, binary tree traversal (in-order, pre-order, post-order) becomes straightforward and intuitive, as you simply follow the pointers to the children.

Balanced Trees: Linked representation can naturally handle balanced trees without worrying about the potential wastage of space that can occur in an array-based representation when the tree isn't completely full.

Efficient Memory Allocation: Memory allocation for nodes in a linked representation can be dynamically managed using heap memory allocation, making it more efficient in terms of space usage.

Support for Arbitrary Trees: Linked representation can be extended to represent more general tree structures, such as n-ary trees, by allowing nodes to have references to multiple children.

8. **The advantages and disadvantages of the linear representation of a binary tree using an array:**

   Advantages:
   Space Efficiency: The linear representation can be more space-efficient compared to linked representations, especially when dealing with sparse trees where some nodes are missing.

   Compact Serialization: Linearized binary trees are easily serializable into a continuous sequence of values, making them suitable for storage, transmission, or communication.

   Predictable Memory Access: Array-based representation allows for predictable memory access patterns, which can lead to better cache performance during traversal.

   Simpler Implementation for Complete Binary Trees: For complete binary trees, where nodes are filled from left to right, the linear representation is particularly compact and easy to work with.

   Disadvantages:
   Memory Wastage: In the case of sparse trees or unbalanced trees, the array might contain many empty or unused positions, leading to memory wastage.

   Inefficient for Dynamic Trees: Insertions and deletions of nodes can be complex and inefficient in the linearized representation, especially when dealing with partially filled arrays.

   Limited Flexibility: The linear representation is less flexible compared to linked representations, as it doesn't allow for easy dynamic modifications of the tree structure.

   Complex Indexing: Calculating indices for child nodes and parent nodes can be error-prone and more difficult to manage compared to linked representations.

   Not Suitable for All Trees: The linear representation is most suitable for complete binary trees or trees that are relatively balanced. For unbalanced trees or trees with varying degrees of branching, the array might be inefficiently utilized.

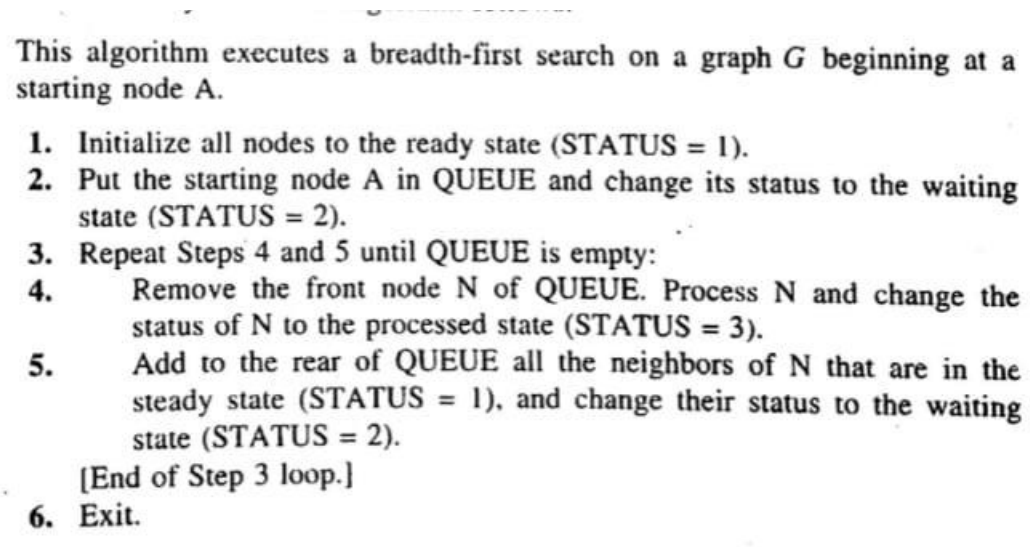
Inefficient for Tree Traversal: While array-based traversal is efficient for level-order traversal (breadth-first), it can be less intuitive and less efficient for other types of traversal like in-order or post-order.

9. **Difference between Tree and Graph:**

Comparison	Tree	Graph
Relationship of node	Only one root node. Parent-Child relationship exists.	No root node. No Parent-Child relationship exists.
Path	Only one path between two nodes	One or more paths exist between two nodes
Edge	N – 1 (N = Number of nodes)	Can not defined
Loop	Loop is not allowed	Loop is allowed
Traversal	Preorder, Inorder, Postorder	BFS, DFS
Model type	Hierarchical	Network

Graph

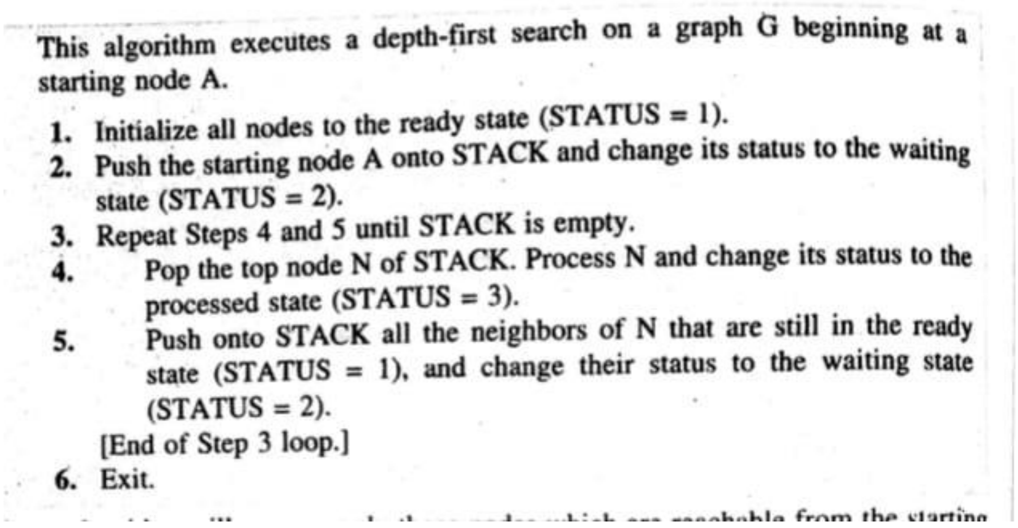
1. **BFS Algorithm:**



This algorithm executes a breadth-first search on a graph G beginning at a starting node A.

```
1. Initialize all nodes to the ready state (STATUS = 1).
2. Put the starting node A in QUEUE and change its status to the waiting
   state (STATUS = 2).
3. Repeat Steps 4 and 5 until QUEUE is empty:
4.       Remove the front node N of QUEUE. Process N and change the
         status of N to the processed state (STATUS = 3).
5.       Add to the rear of QUEUE all the neighbors of N that are in the
         steady state (STATUS = 1), and change their status to the waiting
         state (STATUS = 2).
   [End of Step 3 loop.]
6. Exit.
```

2. DFS Algorithm:



This algorithm executes a depth-first search on a graph G beginning at a starting node A.

```
1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A onto STACK and change its status to the waiting
   state (STATUS = 2).
3. Repeat Steps 4 and 5 until STACK is empty.
4.     Pop the top node N of STACK. Process N and change its status to the
       processed state (STATUS = 3).
5.     Push onto STACK all the neighbors of N that are still in the ready
       state (STATUS = 1), and change their status to the waiting state
       (STATUS = 2).
   [End of Step 3 loop.]
6. Exit.
```

3. Define graph:

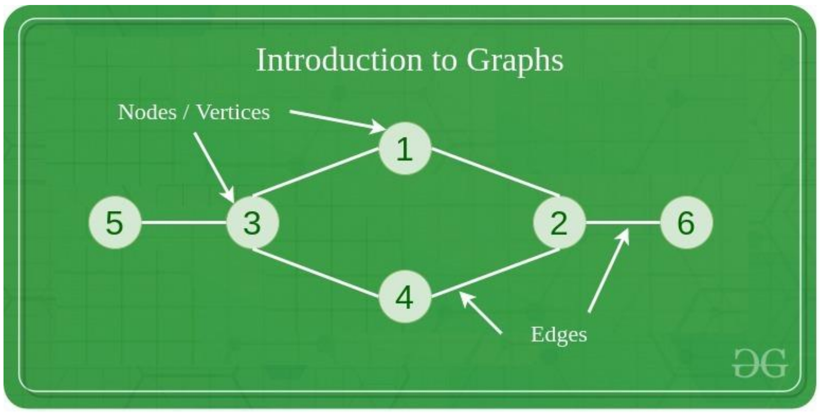
A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).

Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. They are particularly useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.

Components of a Graph

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.



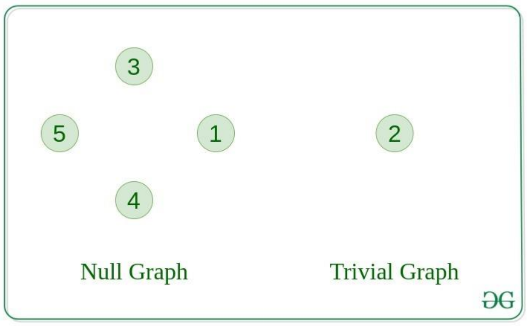
Types Of Graph

1. Null Graph

A graph is known as a null graph if there are no edges in the graph.

2. Trivial Graph

Graph having only a single vertex, it is also the smallest graph possible.

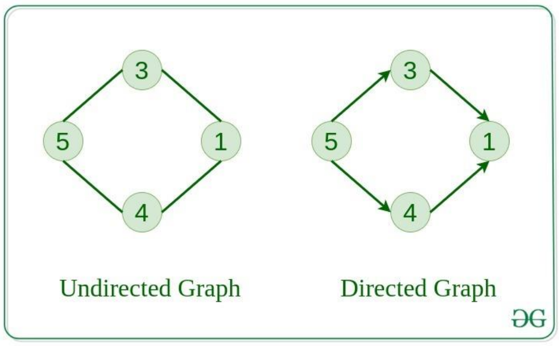


3. Undirected Graph

A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

4. Directed Graph

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

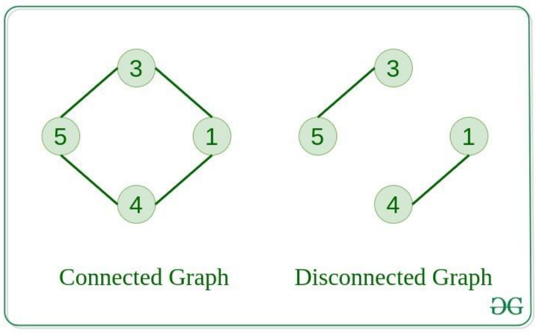


5. Connected Graph

The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. Disconnected Graph

The graph in which at least one node is not reachable from a node is known as a disconnected graph.

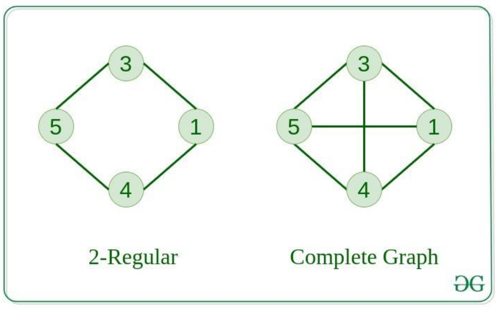


7. Regular Graph

The graph in which the degree of every vertex is equal to K is called K regular graph.

8. Complete Graph

The graph in which from each node there is an edge to each other node.



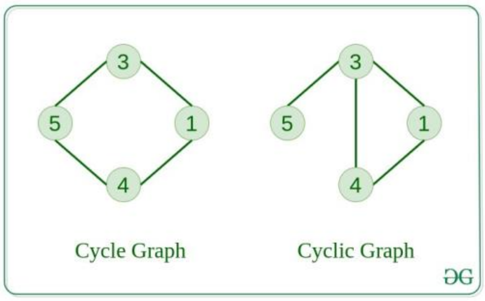
.

9. Cycle Graph

The graph in which the graph is a cycle in itself, the degree of each vertex is 2.

10. Cyclic Graph

A graph containing at least one cycle is known as a Cyclic graph.

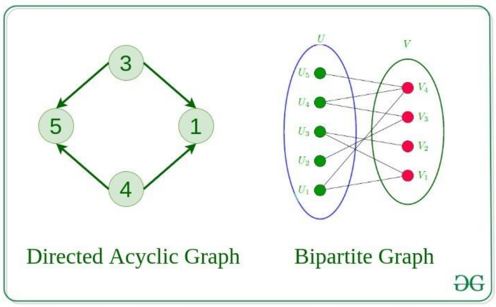


11. Directed Acyclic Graph

A Directed Graph that does not contain any cycle.

12. Bipartite Graph

A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



13. Weighted Graph

- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

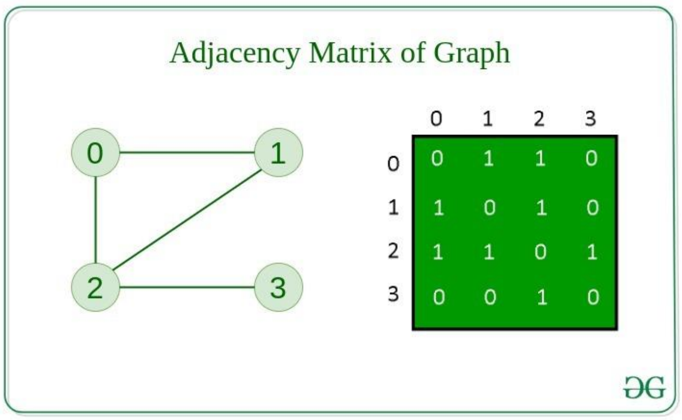
Representation of Graphs

There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List

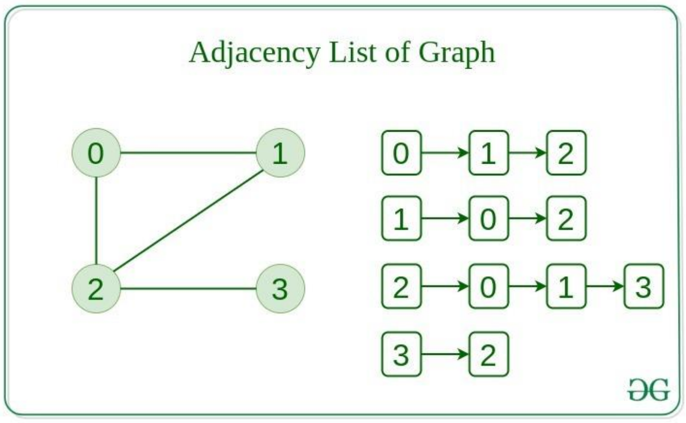
Adjacency Matrix

In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.



Adjacency List

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Basic Operations on Graphs

- Insertion of Nodes/Edges in the graph – Insert a node into the graph.
- Deletion of Nodes/Edges in the graph – Delete a node from the graph.
- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph.

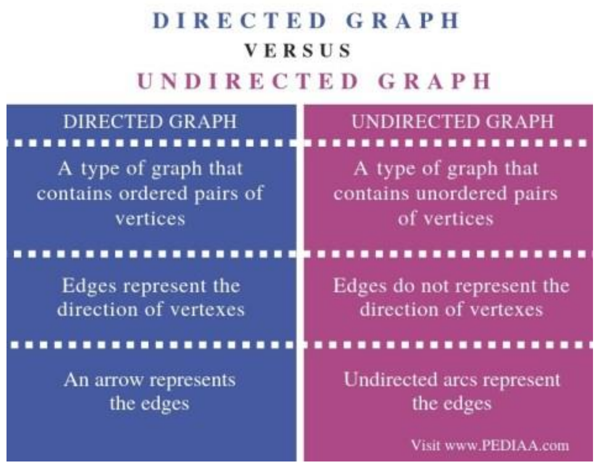
Usage of graphs

- Maps can be represented using graphs and then can be used by computers to provide various services like the shortest path between two cities.
- When various tasks depend on each other then this situation can be represented using a Directed Acyclic graph and we can find the order in which tasks can be performed using topological sort.
- State Transition Diagram represents what can be the legal moves from current states. In-game of tic tac toe this can be used.

4. BFS vs DFS

BFS	DFS
BFS stands for Breadth First Search.	DFS stands for Depth First Search.
BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Here, siblings are visited before the children.	Here, children are visited before the siblings.
Nodes that are traversed several times are deleted from the queue.	The visited nodes are added to the stack and then removed when there are no more nodes to visit.
BFS is used in various applications such as bipartite graphs, shortest paths, etc.	DFS is used in various applications such as acyclic graphs and topological order etc.

5. Directed vs Undirected graph:



DIRECTED GRAPH
VERSUS
UNDIRECTED GRAPH

DIRECTED GRAPH	UNDIRECTED GRAPH
A type of graph that contains ordered pairs of vertices	A type of graph that contains unordered pairs of vertices
Edges represent the direction of vertexes	Edges do not represent the direction of vertexes
An arrow represents the edges	Undirected arcs represent the edges

Visit www.PEDIAA.com