notes link

SECTION-A **2marks Questions**

1. How do you test for an empty queue?

Queue is said to be empty **when the value of front is at -1 or the value of front becomes greater than rear (front > rear).**

isEmpty()

2. Write down the steps to modify a node in linked lists. **CO1**

   1. First, we find the midpoint of that linked list and take a copy of that.
   2. Then we divide that linked list from the middle point.
   3. Then reverse the second part of that linked list.
   4. Then calculate the difference of the first node of the two linked list and put the value to the original linked list.
   5. Continue to find the difference until we go to the last node of the second element.

3. Difference between arrays and lists. **CO1**

S.No.	List	Array
1	List is used to collect items that usually consist of elements of multiple data types.	An array is also a vital component that collects several items of the same data type.
2	List cannot manage arithmetic operations.	Array can manage arithmetic operations.
3	It consists of elements that belong to the different data types.	It consists of elements that belong to the same data type.
4	When it comes to flexibility, the list is perfect as it allows easy modification of data.	When it comes to flexibility, the array is not suitable as it does not allow easy modification of data.
5	It consumes a larger memory.	It consumes less memory than a list.
6	In a list, the complete list can be accessed without any specific looping.	In an array, a loop is mandatory to access the components of the array.
7	It favors a shorter sequence of data.	It favors a longer sequence of data.

notes link

4. What are the various Operations performed on the Stack? **CO2**

push() to insert an element into the stack
pop() to remove an element from the stack
top() Returns the top element of the stack.
isEmpty() returns true is stack is empty else false.
size() returns the size of stack.

5. Define Circular Queue. **CO2**

   A Circular Queue is a special version of queue where the last element of the queue is connected to the first element of the queue forming a circle.

6. List out the steps involved in deleting a node from a binary search tree**.** **CO3**

- If the root is NULL, then return root (Base case)
- If the key is less than the root’s value, then set root->left = deleteNode(root->left, key)
- If the key is greater than the root’s value, then set root->right = deleteNode(root->right, key)
- Else check
    - If the root is a leaf node then return null
    - else if it has only the left child, then return the left child
    - else if it has only the right child, then return the right child
    - else set the value of root as of its inorder successor and recur to delete the node with the value of the inorder successor
- Return

7. List out few of the Application of tree data-structure. **CO3**

   File system:
   Layout of a webpage:
   Decision Tree – Machine learning Algorithm
   Working with Morse Code

8. When a graph said to be weakly connected? **CO4**

   A directed graph is called weakly connected if **replacing all of its directed edges with undirected edges produces a connected (undirected) graph**.

9. .What are the two traversal strategies used in traversing a graph? **CO4**

   The graph has two types of traversal algorithms. These are called the **Breadth First Search and Depth First Search**.

notes link

10.. State the logic of bubble sort algorithm. **CO5**

1. First Iteration (Compare and Swap)

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.

The above process goes on until the last element.

2. Remaining Iteration

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

11. Which sorting algorithm is easily adaptable to singly linked lists? Why**CO5**

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

SECTION-B **16 MARKS**

1. Explain the insertion operation in linked list. How nodes are inserted after a specified node**.** **CO1 5 MARKS**

- Traverse the Linked list upto *position-1* nodes.
- Once all the *position-1* nodes are traversed, allocate memory and the given data to the new node.
- Point the next pointer of the new node to the next of current node.
- Point the next pointer of current node to the new node.

```
void insertPos(Node** current, int pos, int data)
{
    // This condition to check whether the
    // position given is valid or not.
    if (pos < 1 || pos > size + 1)
        cout << "Invalid position!" << endl;
    else {
        // Keep looping until the pos is zero
        while (pos--) {

            if (pos == 0) {

                // adding Node at required position
                Node* temp = getNode(data);

                // Making the new Node to point to
                // the old Node at the same position
                temp->next = *current;

                // Changing the pointer of the Node previous
                // to the old Node to point to the new Node
                *current = temp;
            }
            else
              // Assign double pointer variable to point to the
              // pointer pointing to the address of next Node
              current = &(*current)->next;
        }
        size++;
    }
}
```

2. Write an algorithm to insert a node at the beginning of list? **CO1 5 MARKS**

1. Insert at the beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

notes link

3. Write the algorithm for converting infix expression to postfix (polish) expression? **CO2 5 MARKS**

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

4. What is a DeQueue? Explain its operation with example**?** **CO2 5MARKS**

Dequeue in data structure : A dequeue is a linear data structure, which stands for Double Ended Queue. Unlike **queue**, where the data can be only inserted from one end and deleted from another, in a **dequeue**, the data can be inserted and deleted from both front and rear ends.

In a dequeue in data structure we can perform the following operations:

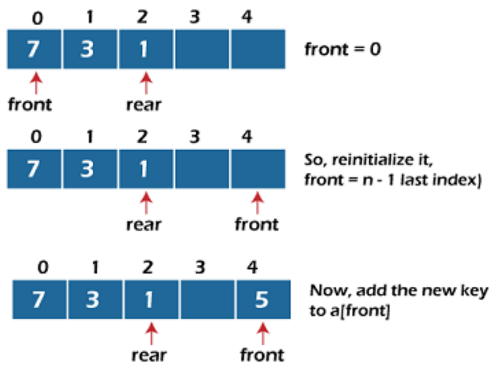
- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

Insertion at the front end

notes link

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.

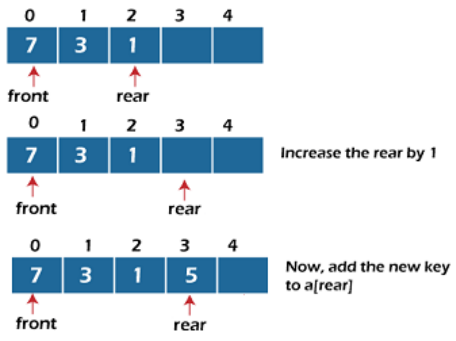


front = 0
So, reinitialize it, front = n - 1 last index)
Now, add the new key to a[front]
front
rear

Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

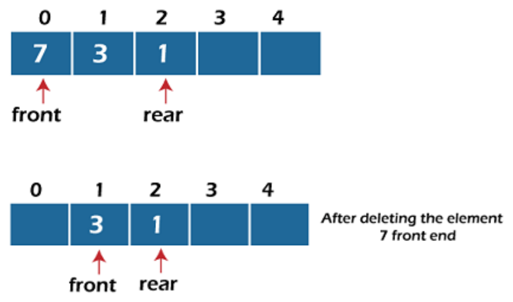
notes link

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

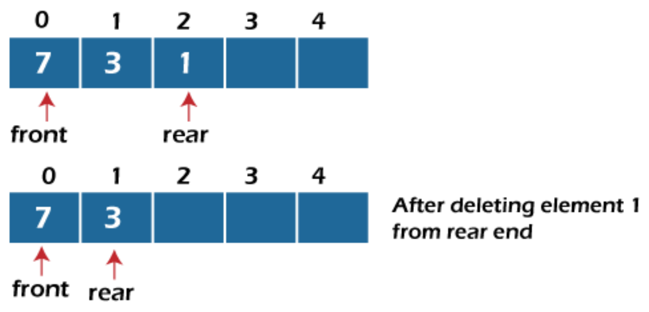
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

notes link

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

5. Discuss and explain in detail about the real-world applications based on Data Structure and algorithm**.** **CO6 6 Marks**

A data structure is a particular way of organizing data in a computer so that it can be used effectively. In this article, the real-time applications of all the data structures are discussed.

Some other applications of the **arrays** are:

1. Arrangement of the leader-board of a game can be done simply through arrays to store the score and arrange them in descending order to clearly make out the rank of each player in the game.
2. A simple question Paper is an array of numbered questions with each of them assigned some marks.

Application of Strings:

1. Spam email detection.
2. Plagiarism detection.
3. Search engine.

Some other applications of the linked list are:

1. Images are linked with each other. So, an image viewer software uses a linked list to view the previous and the next images using the previous and next buttons.
2. Web pages can be accessed using the previous and the next URL links which are linked using a linked list.

Some Applications of a stack are:

1. Converting infix to postfix expressions.
2. Undo/Redo button/operation in word processors.

Some applications of a queue are:

1. Operating System uses queues for job scheduling.
2. To handle congestion in the networking queue can be used.

Application of Sorting Algorithms

1. Order things by their value.
2. Backend Databases (Merge Sort).
3. Playing Cards with your friends (Insertion Sort).

Some applications of a graph are:

notes link

1. Facebook's Graph API uses the structure of Graphs.
2. Google's Knowledge Graph also has to do something with Graph.
3. Dijkstra algorithm or the shortest path first algorithm also uses graph structure to find the smallest path between the nodes of the graph.

Some applications of the trees are:

1. XML Parser uses tree algorithms.
2. The decision-based algorithm is used in machine learning which works upon the algorithm of the tree.
3. Databases also use tree data structures for indexing.

6. Discuss about latest research on efficient Data Structutres. **CO6**

- Succinct data structures for strings, trees, and graphs
- Probabilistic data structures
- Dynamic data structures
- Geometric data structures
- Distributed data structures
- Classic data structures
- Lower bounds
- Implementations

Also explain :-

a) Probabilistic Data Structures

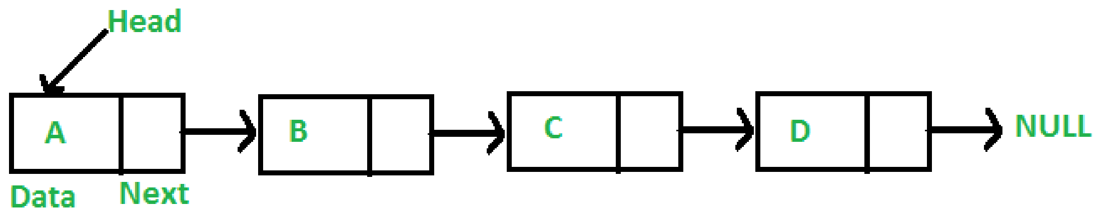
The *Probabilistic data structures and algorithms* (PDSA) are a family of advanced approaches that are optimized to use fixed or sublinear memory and constant execution time; they are often based on hashing and have many other useful features. However, they also have some disadvantages such as they cannot provide the exact answers and have some probability of error (that, actually, can be controlled). The trade-off between the error and the resources is another feature that distinguishes all algorithms and data structures of this family.

b) Dynamic Data Structures

In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures

notes link

are designed to facilitate change of data structures in the run time.



c) Distributed Data Structures

Distributed data structures (DDS): **a DDS has a strictly defined consistency model: all operations on its elements are atomic, in that any operation completes entirely, or not at all**. DDS's have one-copy equivalence, so although data elements in a DDS are replicated, clients see a single, logical data item.

notes link

SECTION C **8 MARKS**

1. Explain INORDER & POSTORDER traversals. Construct an expression tree for the expression (a+b*c) + ((d*e+f)*g). Give the outputs when you apply inorder, preorder and postorder traversals. **CO3**

- For **Inorder**, you traverse from the **left** subtree to the **root** then to the **right** subtree.
- For **Preorder**, you traverse from the **root** to the **left** subtree then to the **right** subtree.
- For **Post order**, you traverse from the **left** subtree to the **right** subtree then to the **root**.

video link

2. Write shorts notes on:- **CO3**

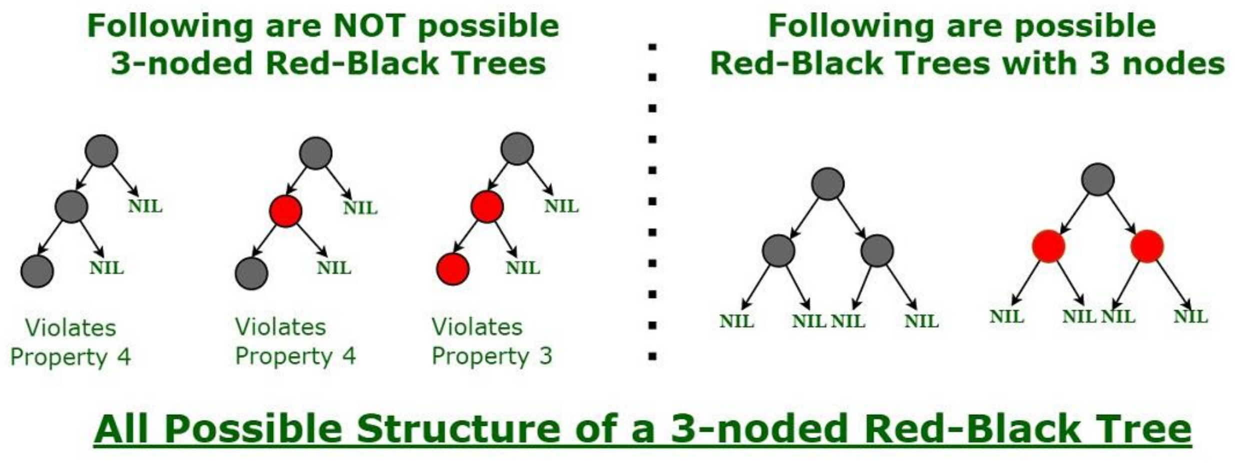
a) Red-Black Tree

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.

It must be noted that as each node requires only 1 bit of space to store the color information, these types of trees show identical memory footprints to the classic (uncolored) binary search tree.

Rules That Every Red-Black Tree Follows:

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. All leaf nodes are black nodes.


Following are NOT possible 3-noded Red-Black Trees
Following are possible Red-Black Trees with 3 nodes
NIL
Violates Property 4
Violates Property 4
Violates Property 3
All Possible Structure of a 3-noded Red-Black Tree

notes link

b) AVL Tree

AVL Trees:

*AVL tree is a self-balancing binary search tree in which each node maintain an extra factor which is called **balance factor** whose value is either -1, 0 or 1.*

c) B+ Tree

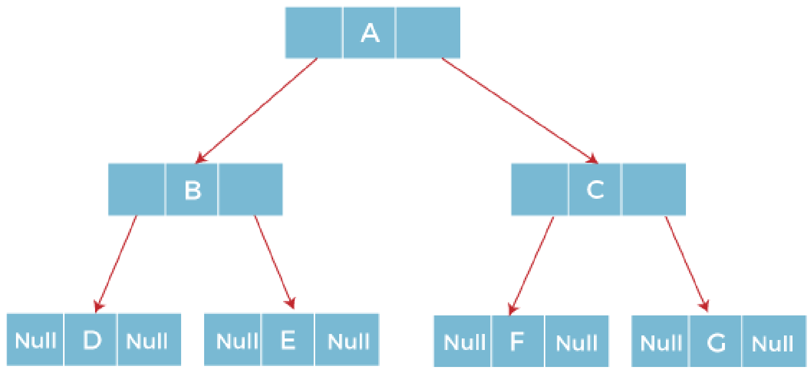
A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.

Properties of a B+ Tree

1. All leaves are at the same level.
2. The root has at least two children.
3. Each node except root can have a maximum of `m` children and at least `m/2` children.
4. Each node can contain a maximum of `m` `- 1` keys and a minimum of `⌈m/2⌉` `-` `1` keys.

d) 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.



Threaded binary tree

notes link

3. Explain Breadth First Search algorithm with example? **CO4**

In this article, we will discuss the BFS algorithm in the data structure. 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.

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

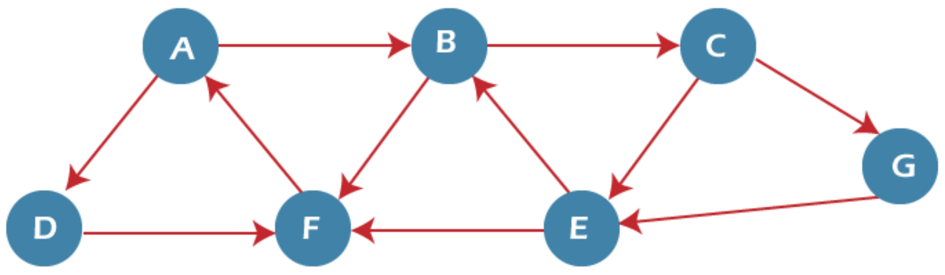
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

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]



Adjacency Lists

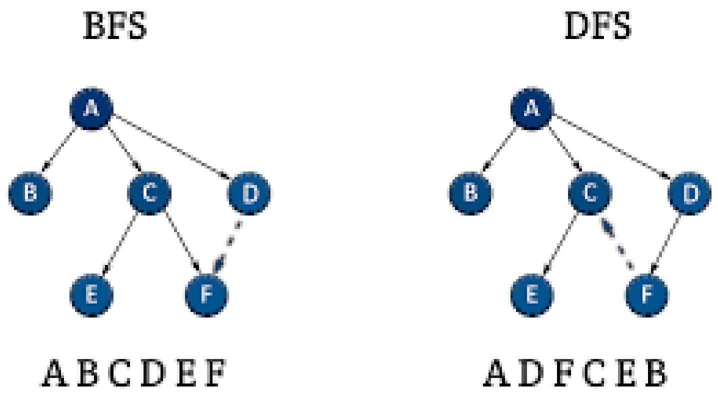
A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

notes link

4. Explain Depth first and breadth first traversal? **CO4**

S. No.	***Parameters***	***BFS***	***DFS***
1.	Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	Definition	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.
4.	Technique	BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
5.	Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
6.	Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
7.	Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
8.	Suitable for Decision Treestheirwinning	BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop.
9.	Time Complexity	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.
10.	Visiting of Siblings/	Here, siblings are visited	Here, children are visited

	Children	before the children.	before the siblings.
11.	Removal of Traversed Nodes	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.
12.	Backtracking	In BFS there is no concept of backtracking.	DFS algorithm is a recursive algorithm that uses the idea of backtracking
13.	Applications	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.
14.	Memory	BFS requires more memory.	DFS requires less memory.
15.	Optimality	BFS is optimal for finding the shortest path.	DFS is not optimal for finding the shortest path.
16.	Space complexity	In BFS, the space complexity is more critical as compared to time complexity.	DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node.
17.	Speed	BFS is slow as compared to DFS.	DFS is fast as compared to BFS.
18.	When to use?	When the target is close to the source, BFS performs better.	When the target is far from the source, DFS is preferable.



notes link

5. Write an algorithm to implement Bubble sort with suitable example. **CO5**

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

How does Bubble Sort Work?

Input: *arr[] = {5, 1, 4, 2, 8}*

First Pass:

- *Bubble sort starts with very first two elements, comparing them to check which one is greater.*
    - *(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.*
    - *(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4*
    - *(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2*
    - *(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.*

Second Pass:

- *Now, during second iteration it should look like this:*
    - *(**1 4** 2 5 8) –> (**1 4** 2 5 8)*
    - *(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2*
    - *(1 2 **4 5** 8) –> (1 2 **4 5** 8)*
    - *(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

Third Pass:

- *Now, the array is already sorted, but our algorithm does not know if it is completed.*
- *The algorithm needs one **whole** pass without **any** swap to know it is sorted.*
    - *(**1 2** 4 5 8) –> (**1 2** 4 5 8)*
    - *(1 **2 4** 5 8) –> (1 **2 4** 5 8)*
    - *(1 2 **4 5** 8) –> (1 2 **4 5** 8)*
    - *(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
```