Ashish Upadhyay
CST SPL 1
Roll no. 16

Tutorial (5)

Ans. (1)

BFS	DFS
(1) ~~Stack~~, Queue data structure is used.	(1) ~~Queue~~, Stack data structure is used.
(2) Useful when searching for vertices closer to source node.	(2) Useful when destination node is farther from source node.
(3) Not suitable for decision making trees used in games or puzzles.	(3) More suitable for game or puzzle problems.
(4) Siblings are visited before the children.	(4) Children are visited before the siblings.
(5) Requires ~~a~~ more memory.	(5) Requires lesser memory.
Application: In peer-to-peer networks, used to find all neighbours.	Application: We can detect cycles in a graph.

Ans. (2) BFS is implemented using ~~stack~~ queue because BFS explores the closest vertices first and then moves away from the source. So, we need a data structure that gives us the oldest element based on the order they were inserted, that is why we use queue for BFS.

DFS is used to detect cycles, so we have to backtrack, backtracking is possible by using stack, hence we ~~se~~ use stack for DFS.

Ans. (3) In sparse graph, the no. of edges is closer to the minimal no. of edges.

In ~~dese~~ dense graphs, the no. of edges is closer to the maximal no. of edges.

A sparse graph should be stored as a list of edges and a dense graph should be stored as an adjacency matrix.

Ans. (4) In BFS we maintain a "visited" nodes list, when traversing a level, if we encounter a node that was already in the list, we come across a cycle.

In DFS, while ~~traversing~~ traversing if we find a node that is already in the recursion stack, a cycle is detected.

Ans. (5) A disjoint set data structure is also known as union-find data structure and merge-find set. It contains ~~set of~~ collection of disjoint or non-overlapping sets.

Operations that can be performed on this data structure are —

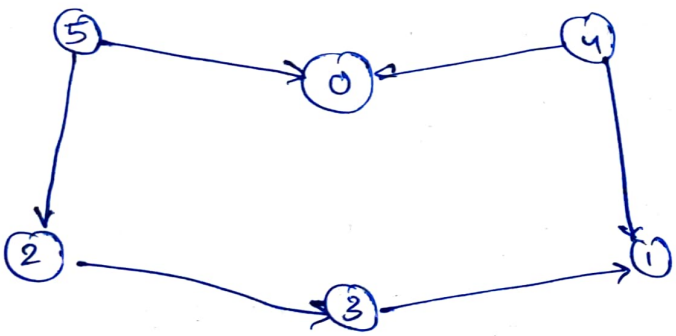
(1) intersection:

(2) Union:

(3) delete an element:

~~(4) get an~~

Ans. (8)



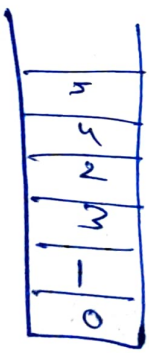
T.s (0)
no neighbour

T.s (1)
no weight

T.s (2)
↓
T.s (3)
↓
~~T.s (4)~~
no weight

T.s (4)

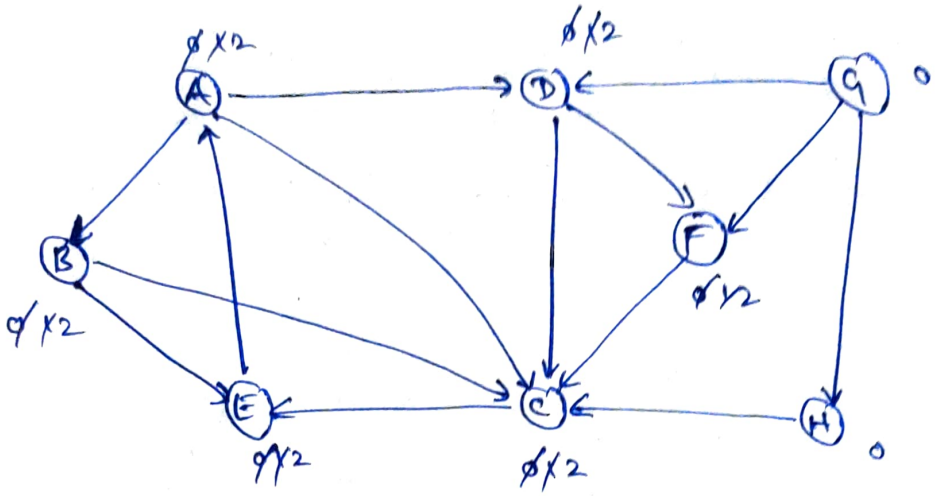
T.s (5)



5, 4, 2, 3, 1, 0

Ans.

Ans. (6)



BFS: Node B E C A D F

Parent B B E A D

Unvisited nodes G and H —

path = B → E → A → D → F

DFS :

node processed: B B C E A D F

Stack : B CF EE AE DE FE E

Path: B → C → E → A → D → F

Ans. (8)

V = {a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

E = {a,b} {a,c} {b,c} {b,d} {e,f} {e,g} {h,i} {j}

(a, b)	{a, b} {c} {d} {e} {f} {g} {h} {i} {j}
(a, c)	{a, b, c} {d} {e} {f} {g} {h} {i} {j}
(b, c)	{a, b, c} {d} {e} {f} {g} {h} {i} {j}
(b, d)	{a, b, c, d} {e} {f} {g} {h} {i} {j}
(e, f)	{a, b, c, d} {e, f} {g} {h} {i} {j}
(e, g)	{a, b, c, d} {e, f, g} {h} {i} {j}
(h, i)	{a, b, c, d} {e, f, g} {h, i} {j}

no. of connected components = 3

Ans. (9) A priority queue is a special type of queue in which each element is associated with a priority value.

Basic operations on this queue are — inserting, removing and peeking elements.

Using Heap data structure:

(1) Insert: insert new element at the end of heap then heapify the tree.

(2) Deleting: Select element, swap with last element, remove last element.

(3) Peeking: return root node.

Applications:
Dijkstra's algorithm
To implement stack
Data compression in Huffman Code.

Ans. (10)

Min heap	Max heap
(1) key at root node is less than or equal to its children.	(1) key at root node is greater than or equal to its children.
(2) Minimum key is at root.	(2) Maximum key is at root.
(3) Smallest element is first to be popped from heap.	(3) largest element is first to be popped from heap.