Tutorial -5

Q1. What is difference between BFS and DFS ? Please write application of both algorithm.

BFS	DFS
• It uses queue data structure	• It uses stack data structure.
• It can be used to find single source shortest path in unweighted graph.	• It is used to find a path from source to destination node.
• No concept of backtracking	• It uses the concept of backtracking
• Requires more memory	• Less memory needed.
• It is better when target is closer to source	• It is better when target is far from source

→ Application of BFS - ① Used for detecting cycles.
② finding shortest and minimum path in unweighted graph.
③ finding a route from GPS.

→ Application of DFS - ① Used for detection of cycles.
② Used for finding path b/w two vertices.
③ Used for job scheduling process.

Q2. which data structure are used to implement BFS and DFS and why?

→ In DFS we have to traverse a whole branch of a tree. So for keep tracking on the current node it requires last in first out approach which can be implemented by stack, after it reaches depth of node then all the nodes will be popped out of stack. Next it searches for adjacent nodes.
If it was implemented with queue we could not reach the depth before that it would dequeue current node.

→ In BFS we have to go through the nodes which have minimum number of nodes in between so we don't have to look for all nodes. If we uses stack then it will go through all the adjacent nodes which consumes more time & do not find minimum path.

Q3. what do you mean by sparse and dense graphs? which representation of graph is better for sparse and dense graph?

→ Dense graph is a graph in which number of edges is close to the maximal number of edges.

→ Sparse graph is a graph in which number of edges is close to ~~minum~~ minimal number of edges.

→ For sparse graph adjacency list is used.
→ For dense graph adjacency matrix is used.

Q4. How can you detect cycle in a graph using BFS and DFS?

→ Using DFS →

① Create a graph using given number of edges & vertices.
② Create a recursive function that have current index visited array & parent node.
③ Mark current node as visited.
④ Find all vertices which are not visited & are adjacent to current node. Recursively call the function for those vertices. If the recursive function returns true return true.
⑤ If the adjacent node is not a parent & already visited then return true.
⑥ Calls the recursive function for all vertices and if any function returns true return true.
⑦ Else If for all vertices the function returns false return false.

→ Using BFS →

① compute number of incoming edges for each of vertex present in graph and initialize the count

of visited nodes as 0.

(2) Pick all vertices with status 0 and add them into a queue.

(3) Remove a vertex from queue & then
- Increment count by 1 for all its neighbouring nodes.
- Decrease status by 1 for all its neighbouring nodes.
- If status of neighbouring nodes is reduced to 0 then add it to queue.

(4) Repeat 3 until queue is empty.

(5) If count is not equal to number of nodes in a graph then cycle otherwise not.

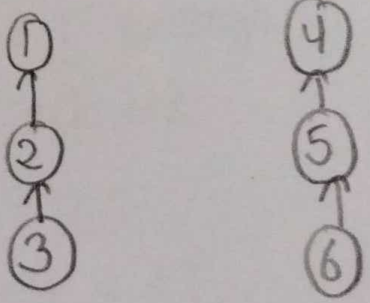
Qs. what do you mean by disjoint set data structure? Explain operations along with examples.

→ Disjoint set are similar to sets in mathematics but they are modified to be usage in algorithm.

→ Operations on disjoint data structure:

(1) Find → it tells the set to which an element belongs.

Ex →



$S_1 = \{1, 2, 3\}$

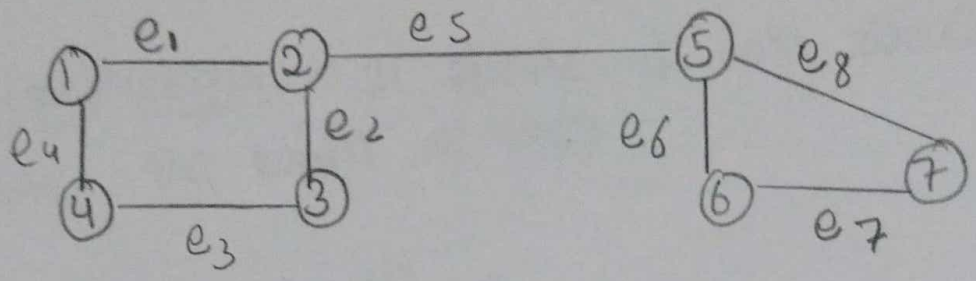
$S_2 = \{4, 5, 6\}$

Find(1) = S_1 , Find(5) = S_2

· It is used basically to find their parent.

② Union → It is used to merge two sets when an edge is added. If both the component belong to same set then it forms a cycle.

Ex-



$U = \{1,2,3,4,5,6,7\}$

① Add $e_1 = \{1,2\}$

1,2 both belongs to U

$\therefore S_1 = \{1,2\}$

② Add $e_2 = \{2,3\}$ [As 2 is in S_1 & 3 is in universal set]

$\therefore S_1 = \{1,2,3\}$

③ Add $e_3 = \{3,4\}$

$\therefore S_1 = \{1,2,3,4\}$

④ Add $e_4 = \{4,1\}$

Both 4 and 1 belongs to S_1

$\therefore$ Cycle detected

⑤ Add $e_5 = \{2,5\}$

$\therefore S_1 = \{1,2,3,4,5\}$

⑥ Add $e_6 = \{5, 6\}$

$S_1 = \{1, 2, 3, 4, 5, 6\}$

⑦ Add $e_7 = \{6, 7\}$

$S_1 = \{1, 2, 3, 4, 5, 6, 7\}$

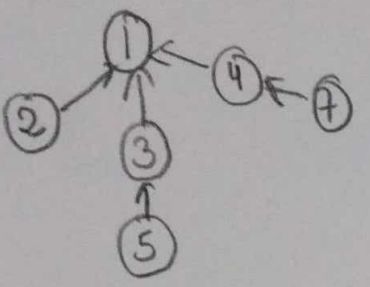
⑧ Add $e_8 = \{5, 7\}$

Both 5 & 7 $\in S_1$

∴ Cycle detected.

③ Path compression → It speeds up data structure by compressing the height of trees.

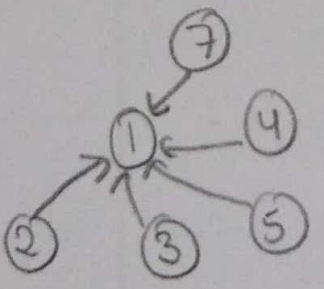
Ex→



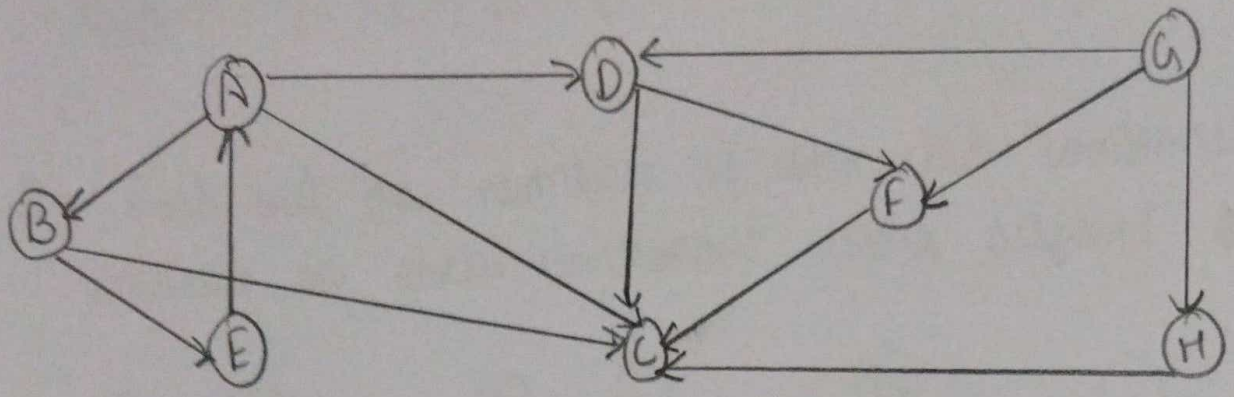
Now parent of 5 = Find(5) = 3 → 1

and parent of 7 = Find(7) = 4 → 1

To speed up data we can directly make path of 7 and 5 to 1.



Q6. Run DFS and BFS on following graph:



→ BFS →

Node	A	D	B	C	F	E	
Parent	–	A	A	A	D	B	

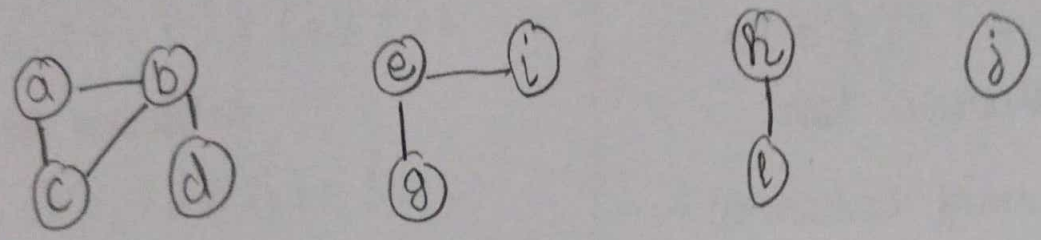
Path :→ A → B → E

→ DFS →

Node	stack (←)
	A
A	BDC
C	BDE
E	BD
D	BF
F	B
B	G
G	H
H	

Path → A → C → E → D → F → B → G → H

Q7. Find out the number of connected components & vertices in each component using disjoint set.



→ $S_1 = \{a, b, c, d\}$
$S_2 = \{e, i, g\}$
$S_3 = \{h, f\}$
$S_4 = \{j\}$

→ connection from a :- Find(a) = S_1

- (a, b) = Find(b) = S_1
  connected.
- (a, c) = Find(c) = S_1
  connected
- (a, d) = Find(d) = S_1
  connected
- (a, e) = Find(e) = S_2
  Not connected
- (a, i) = Find(i) = S_2
  Not connected
- (a, g) = Find(g) = S_2
  Not connected
- (a, h) = Find(h) = S_3
  Not connected
- (a, f) = Find(f) = S_3
  Not connected
- (a, j) = Find(j) = S_4
  Not connected.

- connected from a :
  b, c, d.

→ connection from b: Find (b) = S_1

- (b,a) = Find (a) = S_1
  connected
- (b,c) = Find (c) = S_1
  connected.
- (b,d) = Find (d) = S_1
  connected
- (b,e) = Find (e) = S_2
  Not connected
- (b,i) = Find (i) = S_2
  Not connected
- (b,g) = Find (g) = S_2
  Not connected
- (b,h) = Find (h) = S_3
  Not connected.
- (b,l) = Find (l) = S_3
  Not connected.
- (b,j) = Find (j) = S_4
  Not connected

→ connected from b:
a, c, d

→ connection from c: Find (c) = S_1

- (c,a) = Find (a) = S_1
  connected
- (c,b) = Find (b) = S_1
  connected
- (c,d) = Find (d) = S_1
  connected
- (c,e) = Find (e) = S_2
  Not connected
- (c,i) = Find (i) = S_2
  Not connected
- (c,g) = Find (g) = S_2
  Not connected.
- (c,h) = Find (h) = S_3
  Not connected.
- (c,l) = Find (l) = S_3
  Not connected
- (c,j) = Find (j) = S_4
  Not connected

→ connected from c:
a, b, d

→ connection from d: Find (d) = S_1

- (d,a) = Find (a) = S_1
  connected
- (d,b) = Find (b) = S_1
  connected
- (d,g) = Find (g) = S_2
  Not connected.
- (d,h) = Find (h) = S_3
  Not connected.

- (d, c) = Find(c) = S_1
  connected
- (d, e) = Find(e) = S_2
  Not connected.
- (d, i) = Find(i) = S_2
  Not connected.
- (d, l) = Find(l) = S_3
  Not connected
- (d, j) = Find(j) = S_4
  Not connected.

→ connected from d : a, b, c

→ connection from e : Find(e) = S_2

- (e, a) = Find(a) = S_1
  Not connected
- (e, b) = Find(b) = S_1
  Not connected
- (e, c) = Find(c) = S_1
  Not connected
- (e, d) = Find(d) = S_1
  Not connected
- (e, i) = Find(i) = S_2
  connected
- (e, g) = Find(g) = S_2 connected
- (e, h) = Find(h) = S_3
  Not connected
- (e, l) = Find(l) = S_3
  Not connected
- (e, j) = Find(j) = S_4
  Not connected

conned from e : i, g
Similarly i is connected from : e, g
and g is conneted from : e, i

→ connection from h : Find(h) = S_3

- (h, a) = Find(a) = S_1
  Not connected
- (h, b) = Find(b) = S_1
  Not connected
- (h, c) = Find(c) = S_1
  Not connected
- (h, i) = Find(i) = S_2
  Not connected
- (h, g) = Find(g) = S_2
  Not connected
- (h, l) = Find(l) = S_3
  connected

- (h, d) = Find(d) = S_1
  Not connected.
- (h, e) = Find(e) = S_2
  Not connected

- (h, j) = Find(j) = S_4
  Not connected

→ connected from h : f
similarly connected from f : R

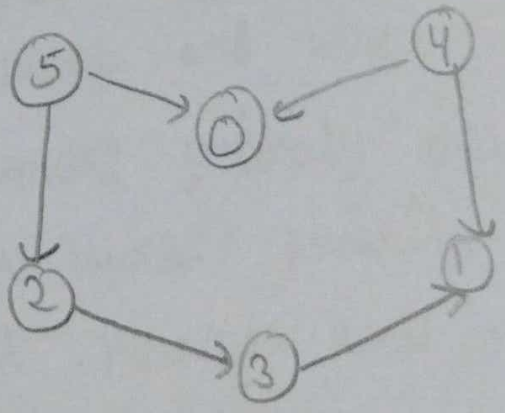
→ connection from j : Find(j) = S_4

- (j, a) = Find(a) = S_1
  Not connected.
- (j, b) = Find(b) = S_1
  Not connected.
- (j, c) = Find(c) = S_1
  Not connected
- (j, d) = Find(d) = S_1
  Not connected
- (j, e) = Find(e) = S_2
  Not connected

- (j, i) = Find(i) = S_2
  Not connected
- (j, g) = Find(g) = S_2
  Not connected
- (j, h) = Find(h) = S_3
  Not connected
- (j, f) = Find(f) = S_3
  Not connected

→ connected from j = ϕ

Q 8: Apply topological sort & DFS on following graph:



DFS →

Node
5
2
3
1
0
4

Stack(e)
5
02
03
01
0
4

Path: 5 → 2 → 3 → 1 → 0 → 4

Topological -

Stack (←)	Node
—	5
0	50
0	52
0	523
0	5231
01	523
013	52
0132	5
01325	4
013254	

Path : 4 → 5 → 2 → 3 → 1 → 0

Q9. Heap data structure can be used to implement priority queue? Name few graph algorithm where you need to use priority queue and why.

→ Heap can be used to implement priority queue because in heap the highest (or lowest) priority element is always stored at the root. However heap is not sorted it can be regarded as partially ordered. It is useful when to remove highest or lowest priority.

→ Use of priority queue :-

① Dijkstra's shortest path - Priority queue is used to extract minimum efficiently node when implementing Dijkstra algorithm.

② Prim's algorithm - Priority queue is used to store keys of nodes and extract minimum key node at every step.

③ Huffman algorithm - Priority queue uses data to compress data.

④ A* search algorithm - Priority queue is used to keep track of unexplored routes.

Q10. what is difference b/w min and max heap?

Min heap	Max heap
• The key present at root must be less than or equal to other nodes.	The key present at root must be greater or equal to other nodes.
• Minimum key element is present at root.	Maximum key element present at root.
• Uses ascending priority.	Uses descending priority.
• Smallest element has priority.	largest element has priority.
• Smallest element is first to be popped.	largest element is first to be popped.