# Tutorial - 5

① 

| BFS | DFS |
|---|---|
| 1. Uses queue data structure. | 1. Uses stack data structure. |
| 2. Stands for Breadth First Search. | 2. Stands for Depth First Search. |
| 3. Can be used to find single source shortest path in an unweighted graph & we reach a vertex with minimum no. of edges from a source vertex. | 3. We might traverse through more edges to reach a destination vertex from a source. |
| 4. Siblings are visited before the children. | 4. Children are visited before siblings. |
| Applications: | Applications: |
| i) Shortest path & minimum Spanning Tree for unweighted graph. | i) Detecting cycle in a graph. |
| ii) Peer to Peer networks. | ii) Path finding. |
| iii) Social Networking websites | iii) Topological sorting. |
| iv) GPS navigation system. | iv) Solving puzzles with only one solution. |

② An BFS we use Queue data structure as queue is used when things have to be processed immediately, but have to be processed in FIFO order like BFS.

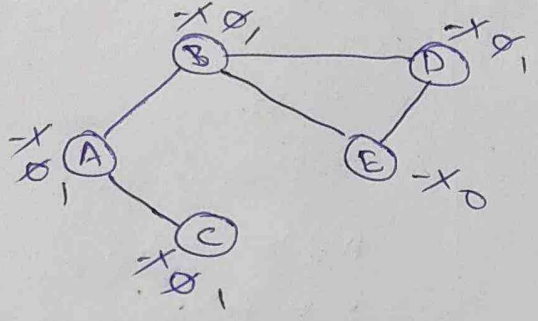
In DFS, stack is used as DFS uses backtracking. For DFS, we retrieve it from root to the farthest node as much as possible, this is the same idea as LIFO (used by stack).

(3) Dense graph is a graph in which the no. of edges is close to the maximum no. of edges.

Sparse graph is a graph in which the no. of edges is close to the minimal no. of edges.
It can be disconnected graph.

* Adjacency list are preferred for sparse graph & Adjacency matrix for dense graph.

(4) Cycle detection in Undirected Graph (BFS)




-1 = Unvisited
0 = into the queue (node)
1 = traversed.

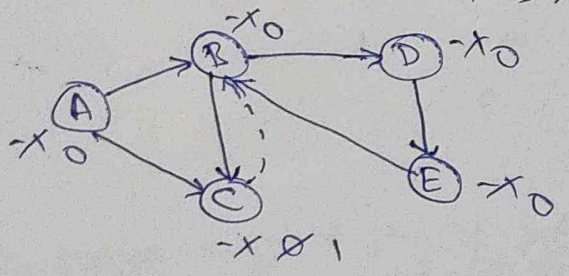
Queue : | A | B | C | D | E |

Visited set : { A | B | C | D }

When D checks it's adjacent vertices it finds E with 0.

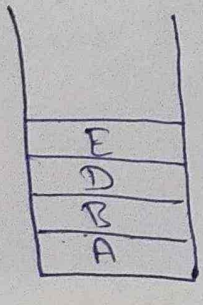
=) If any vertex finds the adjacent vertex with flag 0, then it contains a cycle.

Cycle Detection in Directed Graph (DFS)




-1 = Unvisited
0 = Visited & in stack
1 = Visited & popped out of stack.

Stack :




Visited set:
A B C D E

=) B → D → E → B

Parent Map

| Vertex | Parent |
|---|---|
| A | — |
| B | A |
| C | B |
| D | B |
| E | D |

Here E finds B (adjacent vertex of E) with O.

=) It contains a cycle.

⑤ The disjoint set data structure is also known as union-find data structure & merge-find set. It is a data structure that contains a collection of disjoint or non-overlapping sets.

The disjoint set means that when the set is partitioned into the disjoint subsets, various operation can be performed on it.

In this case, we can add new sets, we can merge the sets & we can also finds the representative member of a set. It also allows to find out whether the two elements are in the same set or not efficiently.

Operations on disjoint sets

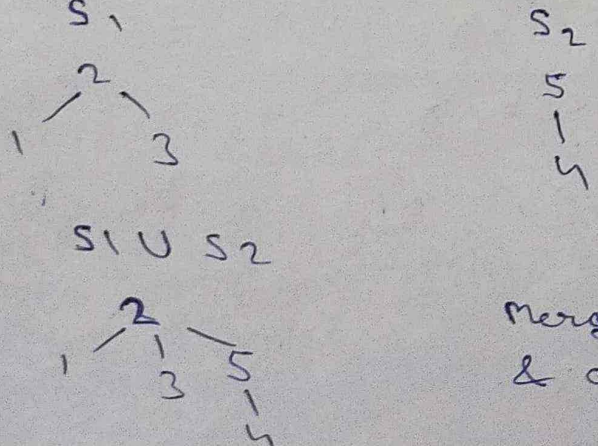
1. Union

a) If S1 & S2 are two disjoint sets, their union S1 ∪ S2 is a set of all elements x such that x is in either S1 or S2.

b) As the sets should be disjoint S1 ∪ S2 replaces S1 & S2 which no ~~longer~~ longer exists.

c) Union is achieved by simply making one of the trees as a subtree of other i.e. to set parent field of one of the roots of the trees to other root

Ex.




Merge the sets containing x & containing y into one.

② Find

Given an element x, to find the set containing it.

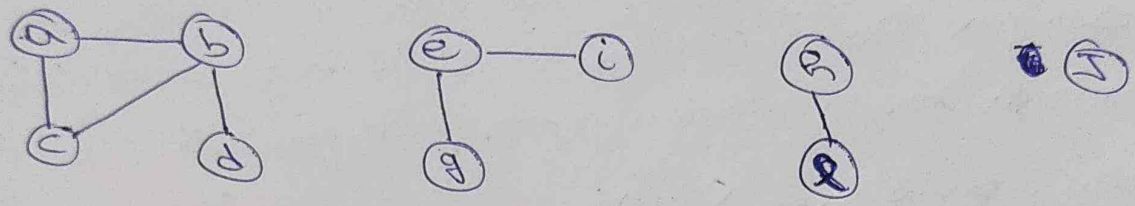
Ex.


find(3) ⇒ S1 return in which set x belongs

find(5) ⇒ S2

③ Make-Set(x) : Create a set containing x.

⑦




V = { a, b, c, d, e, g, h, i, J, l }

E = { (a,b), (a,c), (b,c), (b,d), (e,i), (e,g), (h,l), (J) }

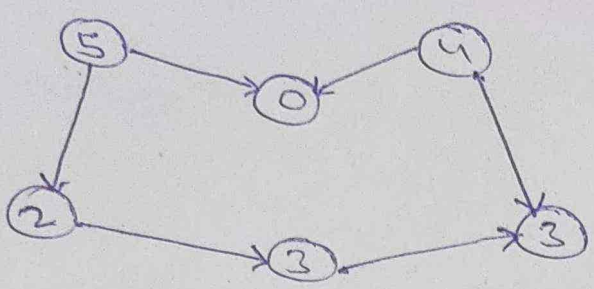
| | {a} | {b} | {c} | {d} | {e} | {g} | {h} | {i} | {J} | {l} |
|---|---|---|---|---|---|---|---|---|---|---|
| (a,b) | {a,b} | | {c} | {d} | {e} | {g} | {h} | {i} | {J} | {l} |
| (a,c) | {a,b,c} | | | {d} | {e} | {g} | {h} | {i} | {J} | {l} |
| (b,c) | {a,b,c} | | | {d} | {e} | {g} | {h} | {i} | {J} | {l} |
| (b,d) | {a,b,c,d} | | | | {e} | {g} | {h} | {i} | {J} | {l} |
| (e,i) | {a,b,c,d} | | | | {e,i} | {g} | {h} | {J} | {l} | |
| (e,g) | {a,b,c,d} | | | | {e,i,g} | {h} | {J} | {l} | | |
| (h,l) | {a,b,c,d} | | | | {e,i,g} | {h,l} | {J} | | | |
| (J) | {a,b,c,d} | | | | {e,i,g} | {h,l} | {J} | | | |

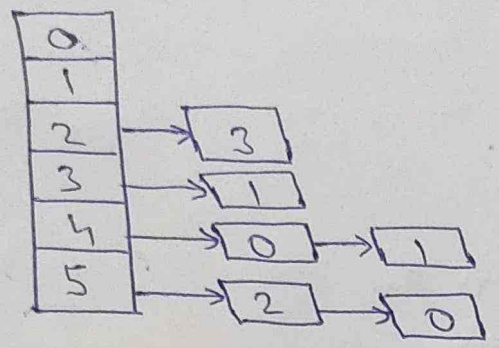
We have,

{a,b,c,d} {h,l}

{e,i,g} {J}

(8)

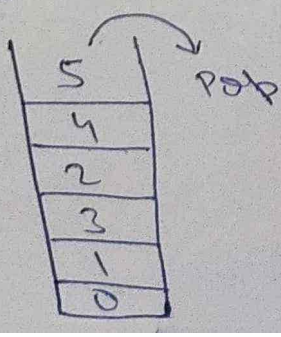



Algorithm -

1. Go to node 0, it has no outgoing edges so push node 0 into the stack & mark it visited.
2. Go to node 1, again it has no outgoing edges, so push node 1 into the stack & mark it visited.
3. Go to node 2, process all the adjacent nodes & mark node 2 visited.
4. Node 3 is already visited so continue with the next node.
5. Go to node 4, all its adjacent nodes are already visited so push node 4 into the stack & mark it visited.
6. Go to node 5, all its adjacent nodes are already visited so push node 5 into the stack & mark it visited.

=>




5 4 2 3 1 0

(output)

(9) Heap is generally preferred for priority queue implementation because heaps provide better performance compared to arrays or linked list.

Algorithms where priority queue is used :-

1. Dijkstra's shortest Path Algorithm : When the graph is stored in the form of adjacency list or matrix, priority is used to extract minimum efficiency when implementing Dijkstra's algorithm.

2. Prim's Algorithm : To store keys of nodes & extract minimum keys node at every step.

(10)

| Min Heap | Max Heap |
|---|---|
| 1. For every pair of parent & descendant child node, the parent node always has lower value than descendant child node. | 1. For every pair of the parent & descendant child node, the parent node has greater value than descendant child node. |
| 2. The value of nodes increases as we traverse from root to leaf node. | 2. The value of nodes decreases as we traverse from root to leaf node. |
| 3. Root node has lowest value. | 3. Root node has greatest value. |