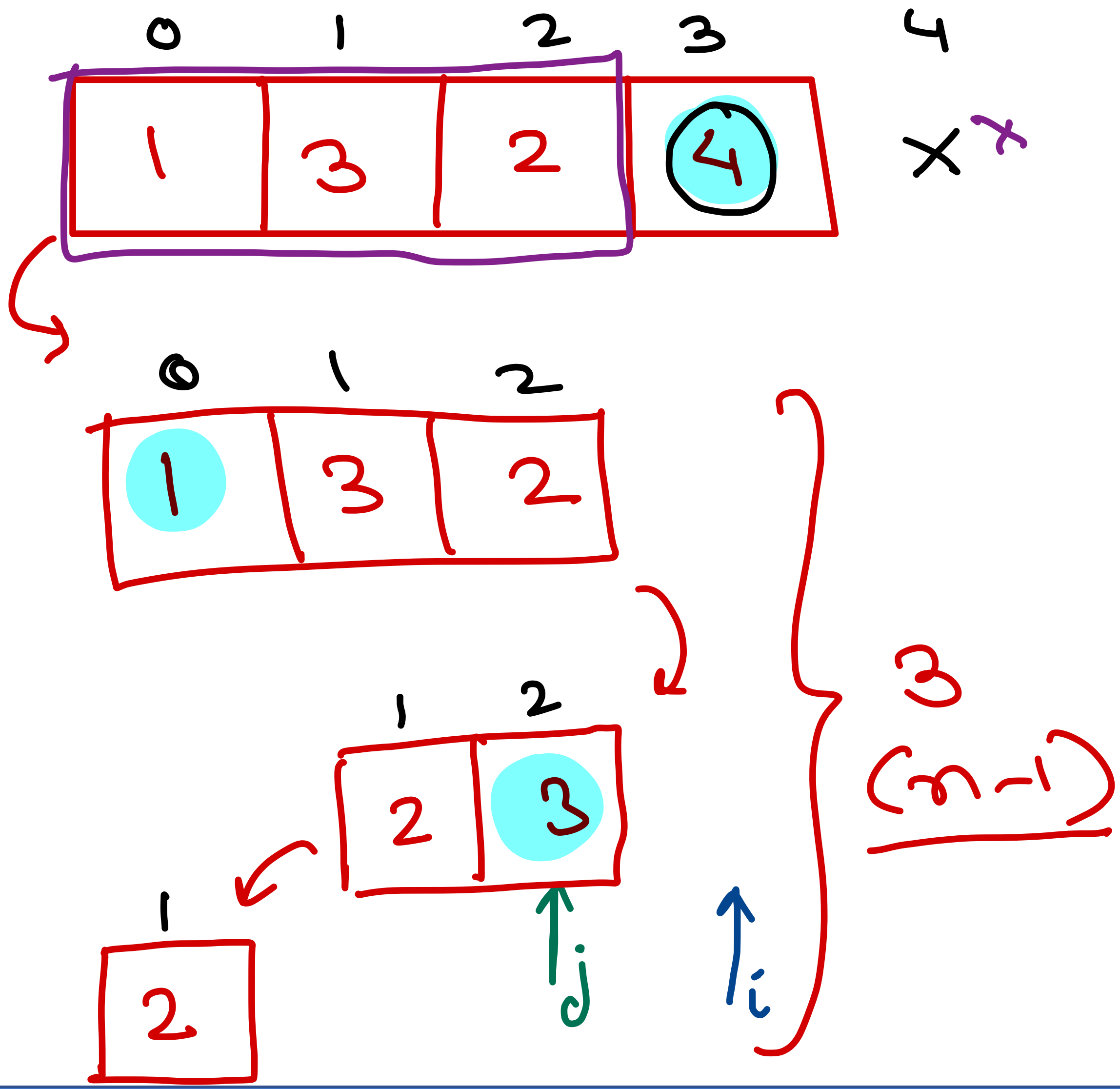
Data Structure & Algorithms

Sunbeam Infotech

Quick Sort



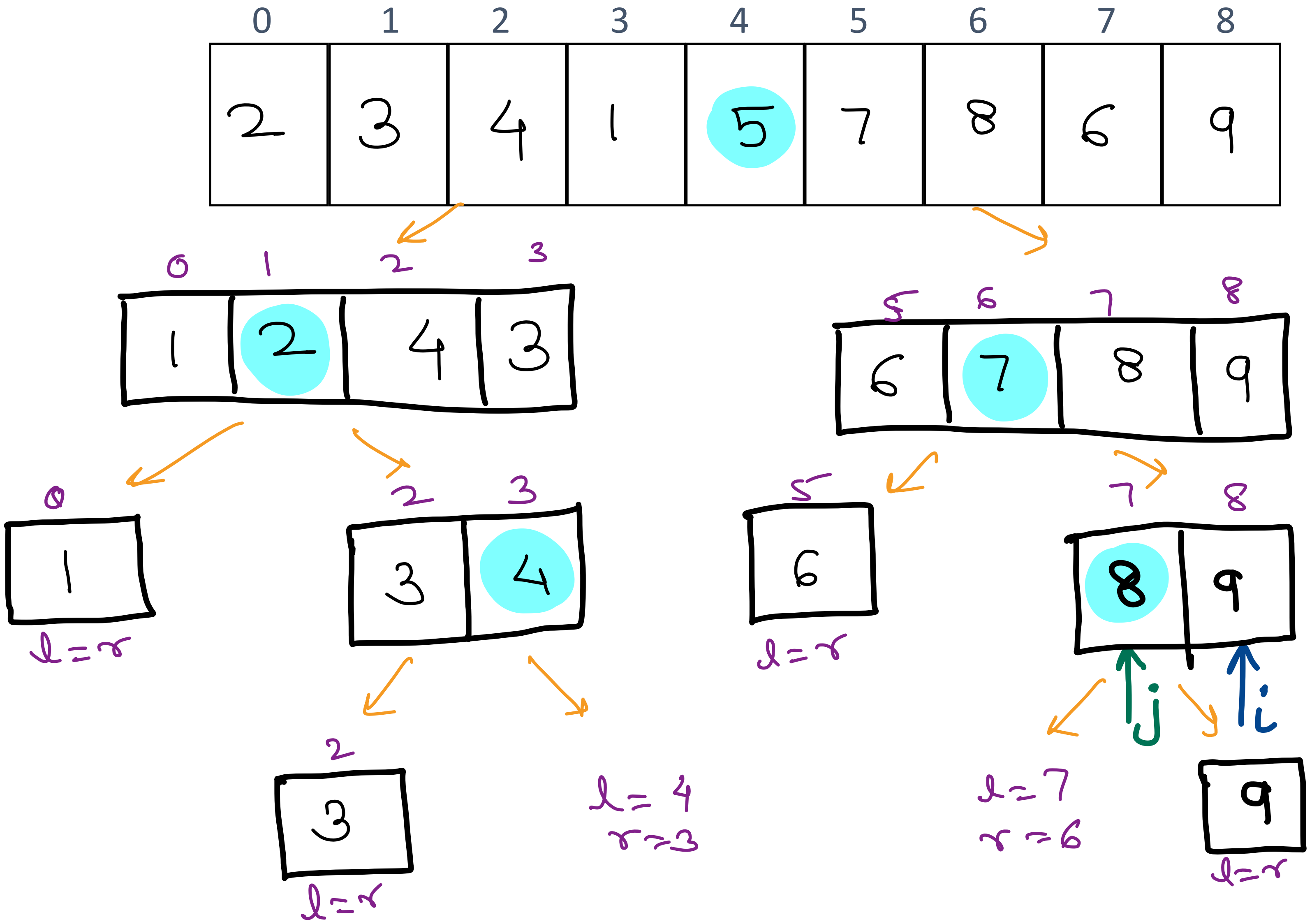
5 3 9 1 8 7 2 6 4

0	1	2	3	4	5	6	7	8
2	3	4	1	5	7	8	6	9

Quick Sort

Quick Sort
↳ O(n log n)

worst case
↳ O(n²)



Quick Sort – Time complexity

- Quick sort pivot element can be
  - First element or Last element
  - Random element
  - Median of the array → middle element in sorted array.
- Quick sort time
  - Time to partition as per pivot – T(n)
  - Time to sort left partition – T(k)
  - Time to sort left partition – T(n-k-1)
- Worst case
  - T(n) = T(0) + T(n-1) + O(n) => O(n^2)
- Best case
  - T(n) = T(n/2) + T(n/2) + O(n) => O(n log n)
- Average case
  - T(n) = T(n/9) + T(9n/10) + O(n) => O(n log n)

Merge Sort

$h \simeq \log n$

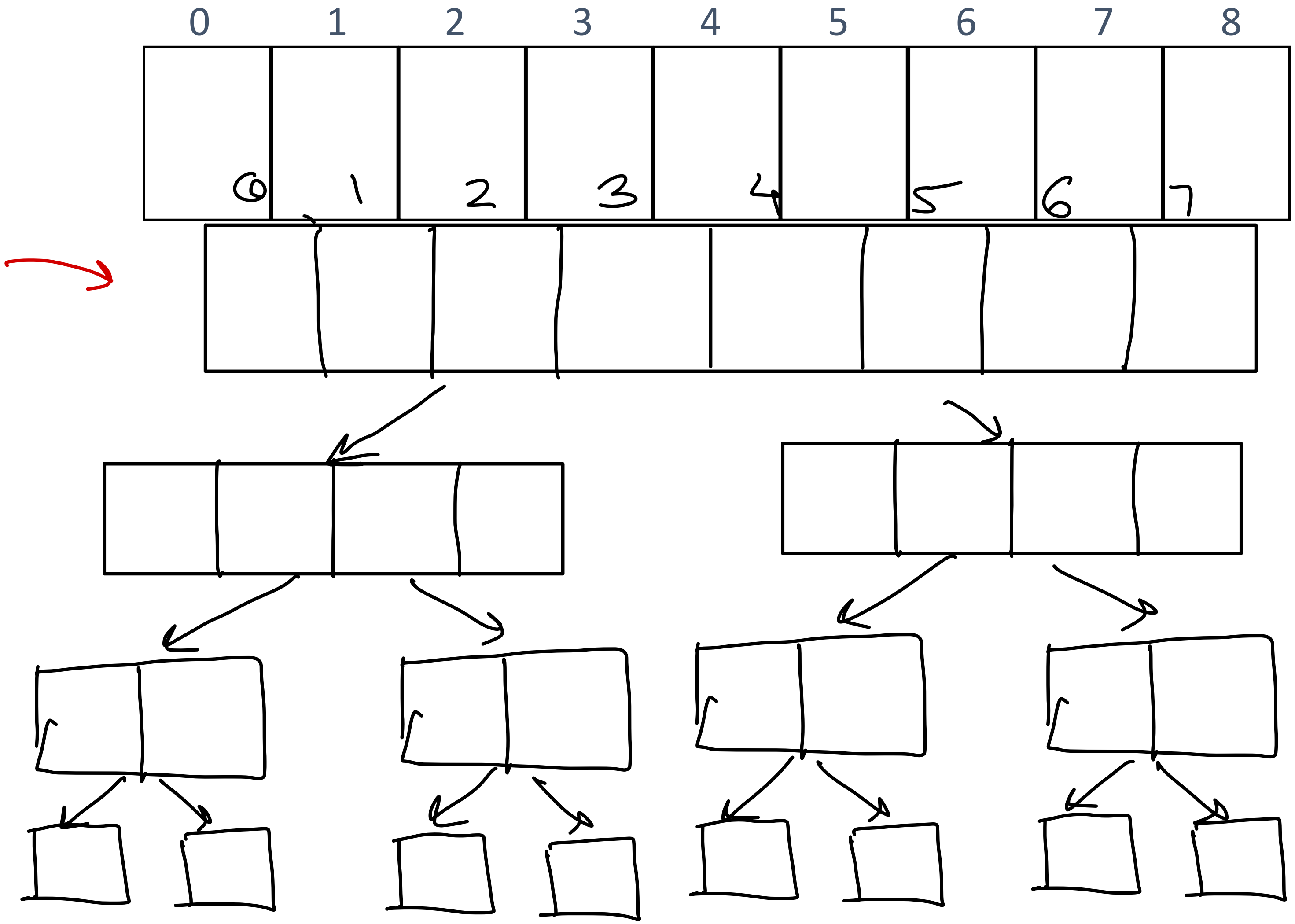
* <u>partitioning</u>
  ↳ $\log n$

* <u>merging</u>
  ↳ n

* <u>time complexity</u>
  $O(n \log n)$

h

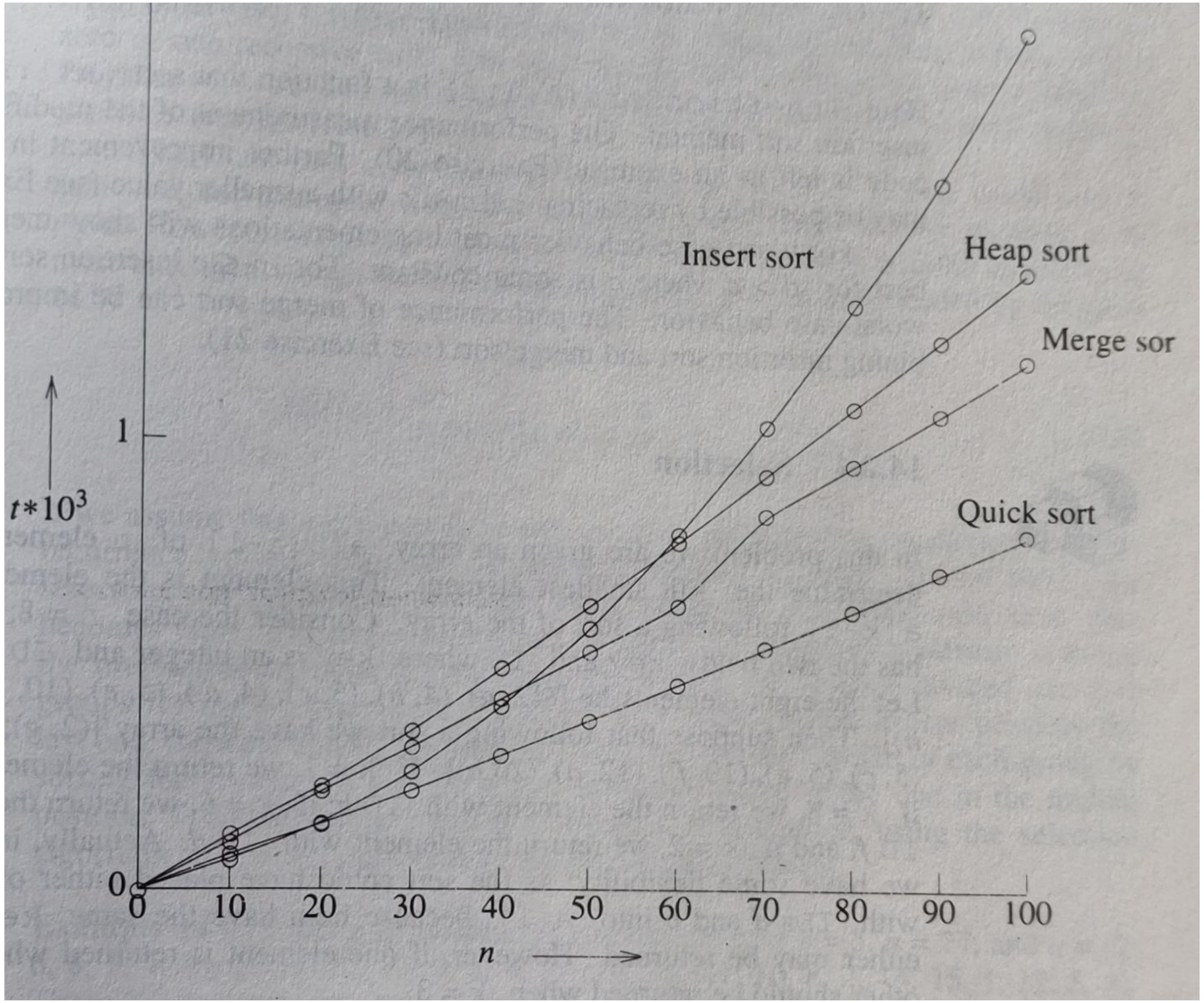
recursion tree



Sorting Algorithm Comparison

Shell Sort → Partitioning + insertion sort
↳ O(n log n)

- Selection sort algorithm is too simple, but performs poor and no optimization possible.
  ↳ no caching usability
- Bubble sort can be improved to reduce number of iterations.
- Insertion sort performs well if number of elements are too less. Good if adding elements and resorting.
- Quick sort is stable if number of elements increase. However worst case performance is poor.
- Merge sort also perform good, but need extra auxiliary space. O(n)



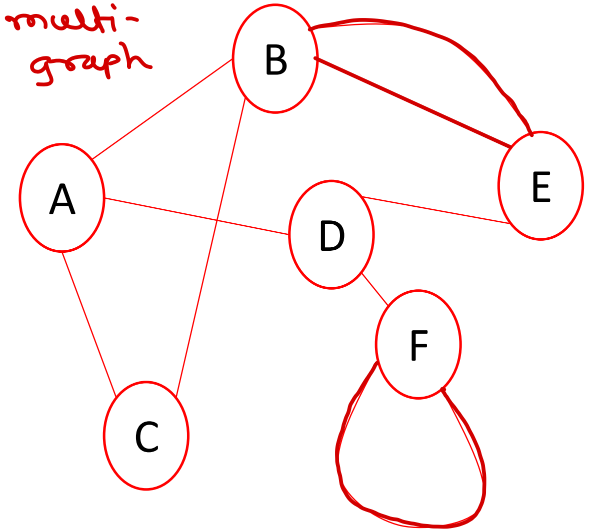
Graph Data Structure & Algorithms

Sunbeam Infotech

Graph

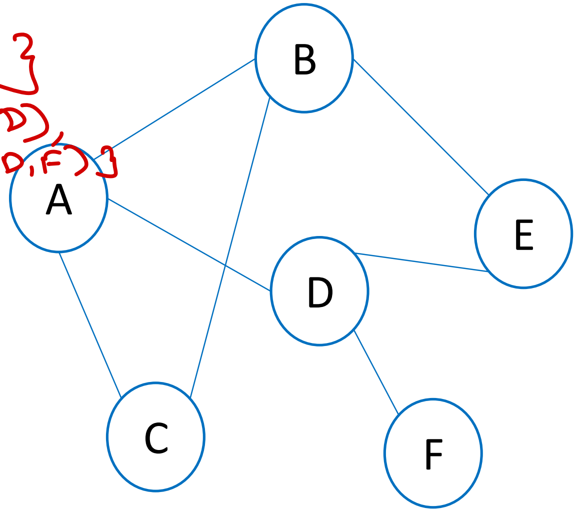
Graph Theory - Math

- Graph is a non-linear data structure.
- Graph is defined as set of vertices and edges. Vertices (also called as nodes) hold data, while edges connect vertices and represent relations between them.
  - G = { V, E }
- Vertices hold the data and Edges represents relation between vertices.
- When there is an edge from vertex P to vertex Q, P is said to be adjacent to Q.
- Multi-graph
  - Contains multiple edges in adjacent vertices or loops (edge connecting a vertex to it-self).
- Simple graph
  - Doesn't contain multiple edges in adjacent vertices or loops.



V = { A, B, C, D, E, F }

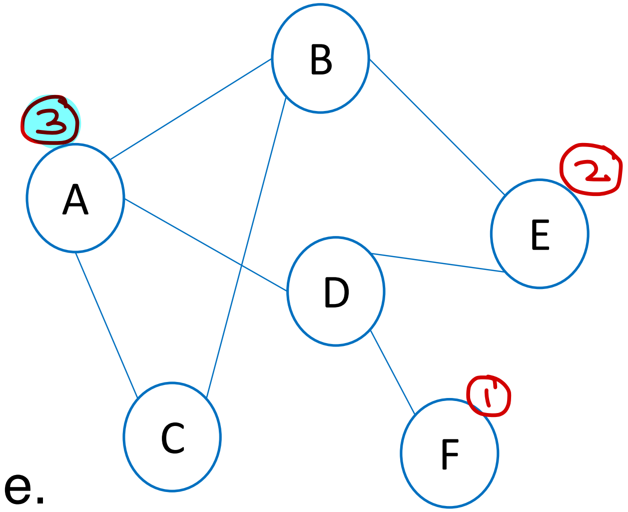
E = { (A,B), (A,C), (A,D), (B,C), (B,E), (D,E), (D,F) }



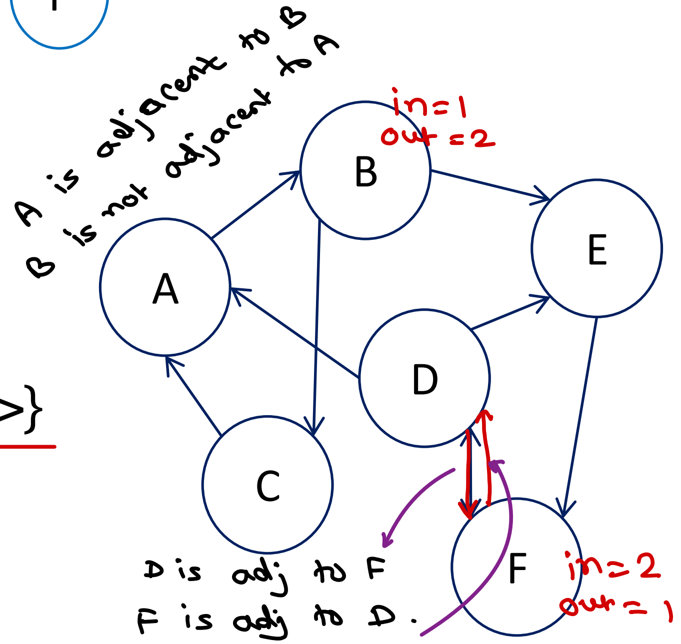
Graph

- Graph edges may or may not have directions.

- Undirected Graph: G = { V, E }
  - V = { A, B, C, D, E, F}
  - E = { (A,B), (A,C), (A,D), (B,C), (B,E), (D,E), (D,F) }
  - If P is adjacent to Q, then Q is also adjacent to P.
  - Degree of node: Number of nodes adjacent to the node.
  - Degree of graph: Maximum degree of any node in graph.



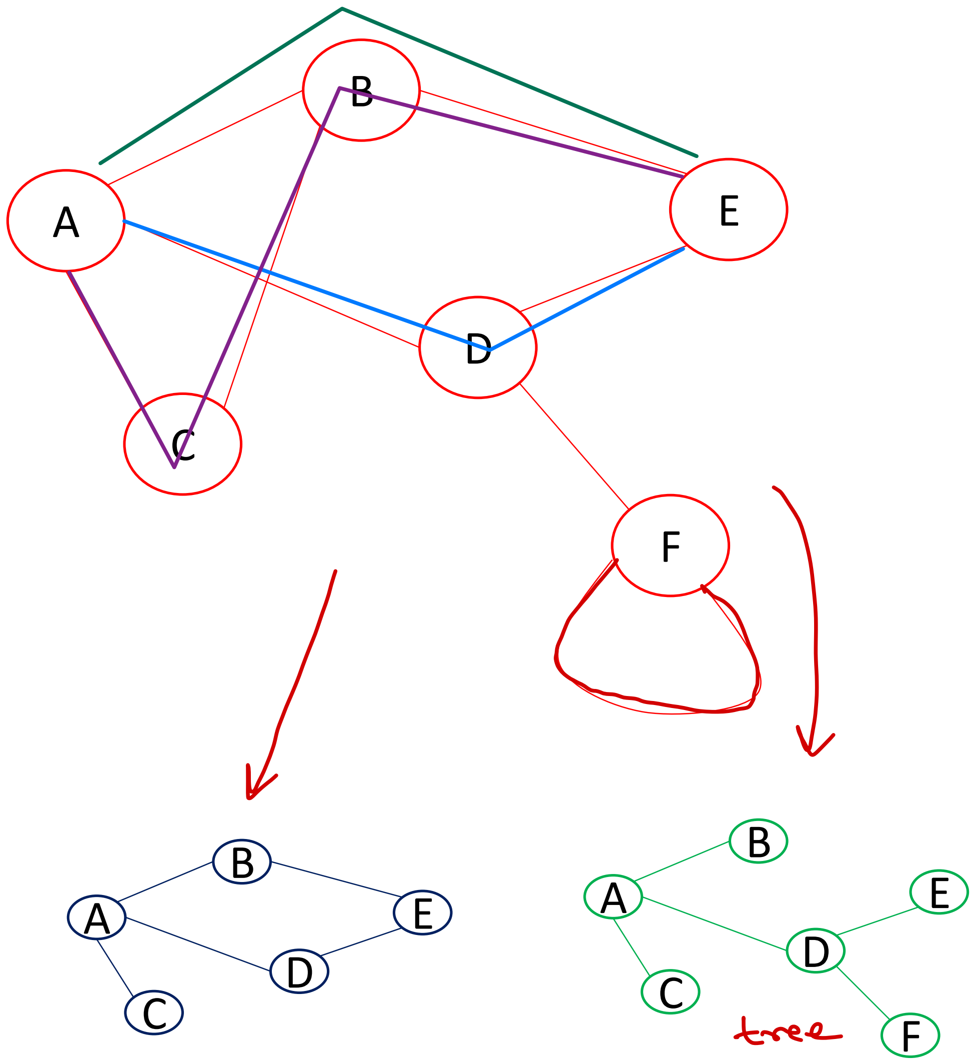
- Directed Graph: G = { V, E }
  - V = { A, B, C, D, E, F}
  - E = {<A,B>, <B,C>, <B,E>, <C,A>, <D,A>, <D,E>, <D,F>, <E,F>, <F,D>}
  - If P is adjacent to Q, then Q is may or may not be adjacent to P.
  - Out-degree: Number of edges originated from the node
  - In-degree: Number of edges terminated on the node



Graph

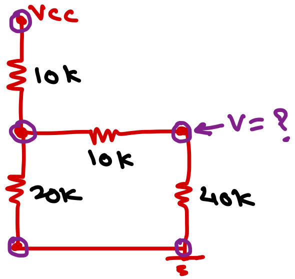
tree is graph without cycle.

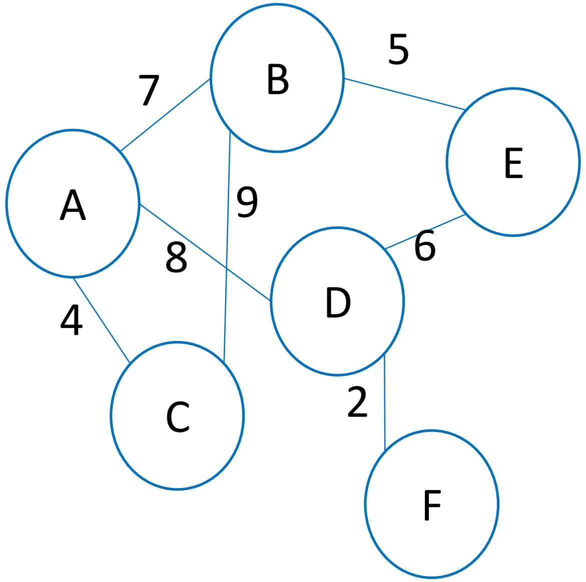
- Path: Set of edges between two vertices. There can be multiple paths between two vertices.
  - A – D – E
  - A – B – E
  - A – C – B – E
- Cycle: Path whose start and end vertex is same.
  - A – B – C – A
  - A – B – E – D – A
- Loop: Edge connecting vertex to itself. It is smallest cycle.
  - F – F
- Sub-Graph: A graph having few vertices and few edges in the given graph, is said to be sub-graph of given graph.



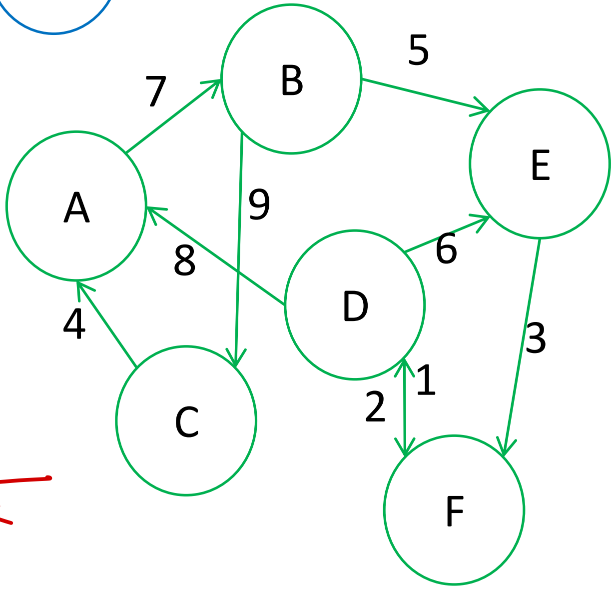
Graph

- Weighted graph
  - Graph edges have weight associated with them.
  - Weight represent some value e.g. distance, resistance.
- Directed Weighted graph (Network)
  - Graph edges have directions as well as weights.
- Applications of graph
  - Electronic circuits
  - Social media
  - Communication network
  - Road network
  - Flight/Train/Bus services
  - Bio-logical & Chemical experiments
  - Deep learning (Neural network, Tensor flow)
  - Graph databases (Neo4j)





undirected weighted graph

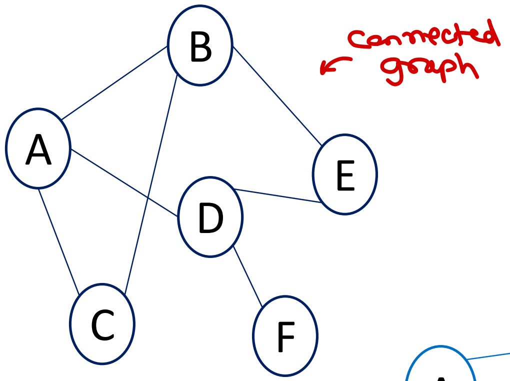


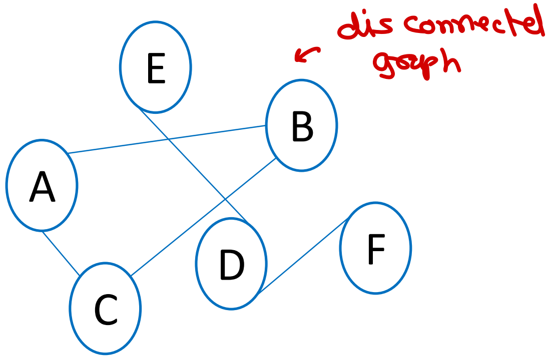
directed weighted graph

network

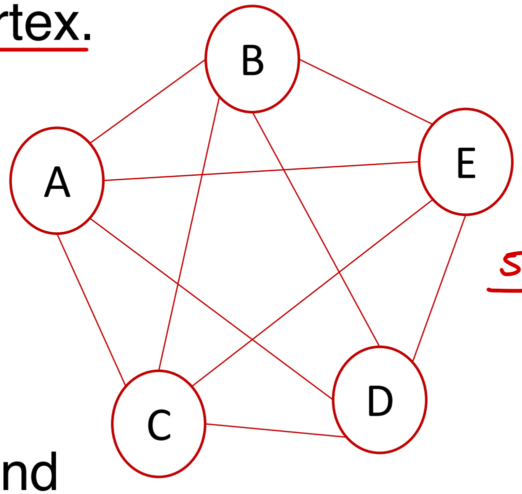
Graph

- Connected graph
  - From each vertex some path exists for every other vertex.
  - Can traverse the entire graph starting from any vertex.



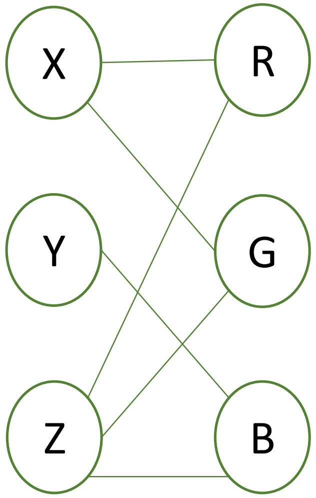


- Complete graph
  - Each vertex of a graph is adjacent to every other vertex.
  - Un-directed graph: Number of edges = n (n-1) / 2
  - Directed graph: Number of edges = n (n-1)



$$\frac{5 \times 4}{2} = 10$$

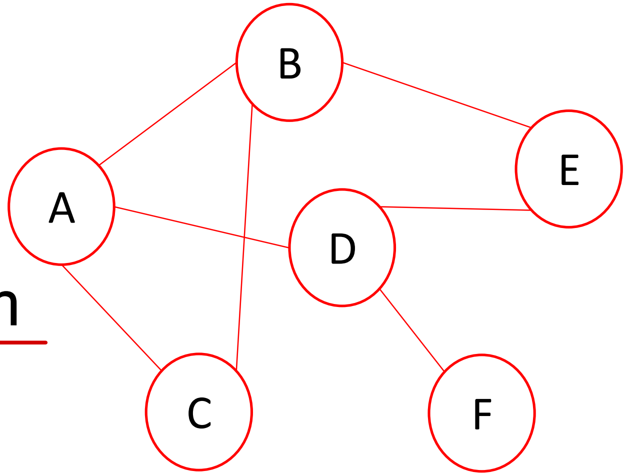
- Bi-partite graph
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.

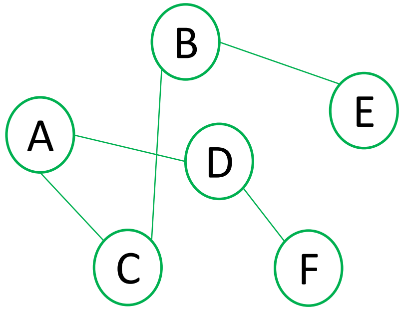


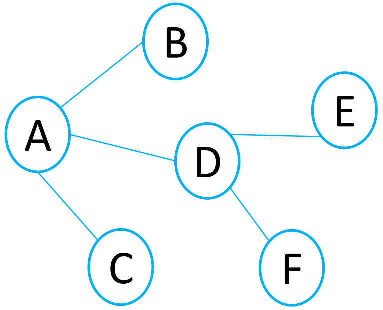
Spanning Tree

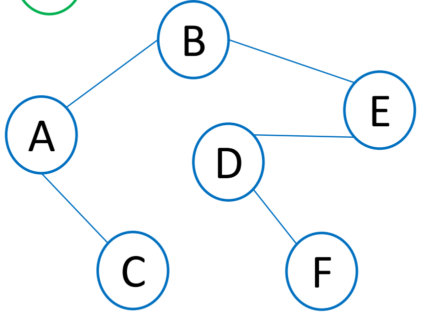
- Tree is a graph without cycles.
- Spanning tree is connected sub-graph of the given graph that contains all the vertices and sub-set of edges (V-1).
- Spanning tree can be created by removing few edges from the graph which are causing cycles to form.
- One graph can have multiple different spanning trees.
- In weighted graph, spanning tree can be made who has minimum weight (sum of weights of edges). Such spanning tree is called as Minimum Spanning Tree.
- Spanning tree can be made by various algorithms.
  - BFS Spanning tree
  - DFS Spanning tree
  - Prim's MST
  - Kruskal's MST

MST



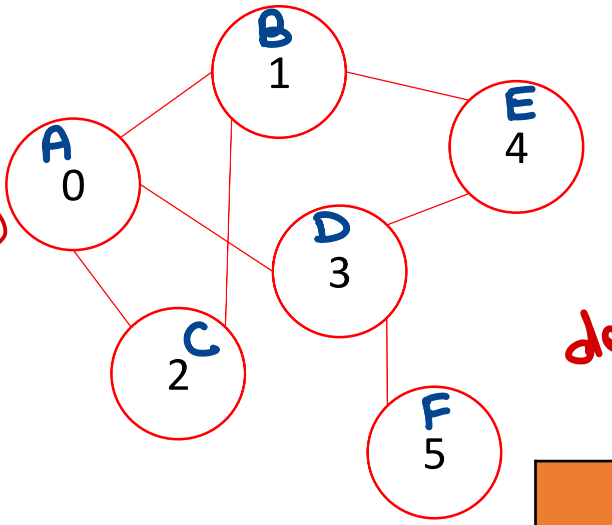






Graph Implementation – Adjacency Matrix

- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence (1) or (0) absence of the edge between vertices.
- For *non-weighted graph*, 1 indicate edge and 0 indicate no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is $O(V^2)$.



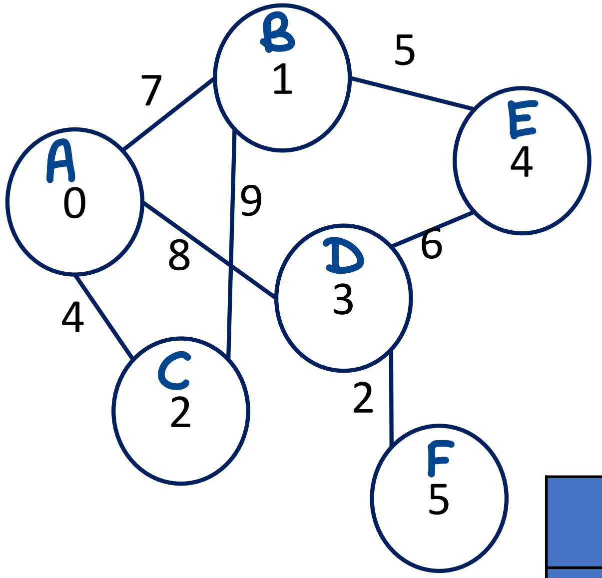
dest →

src ↓

		0	1	2	3	4	5
		A	B	C	D	E	F
0	A	0	1	1	1	0	0
1	B	1	0	1	0	1	0
2	C	1	1	0	0	0	0
3	D	1	0	0	0	1	1
4	E	0	1	0	1	0	0
5	F	0	0	0	1	0	0

Graph Implementation – Adjacency Matrix

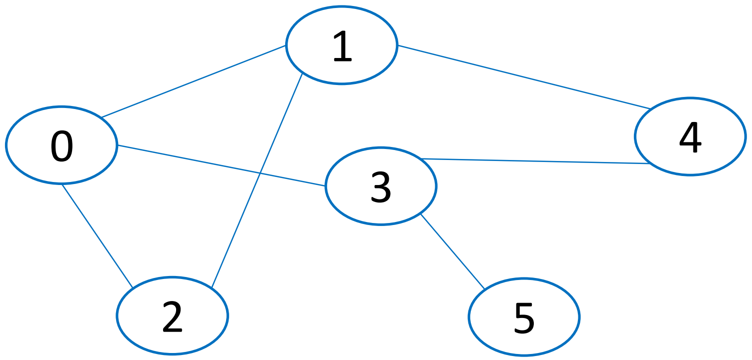
- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For *weighted graph*, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is $O(V^2)$.

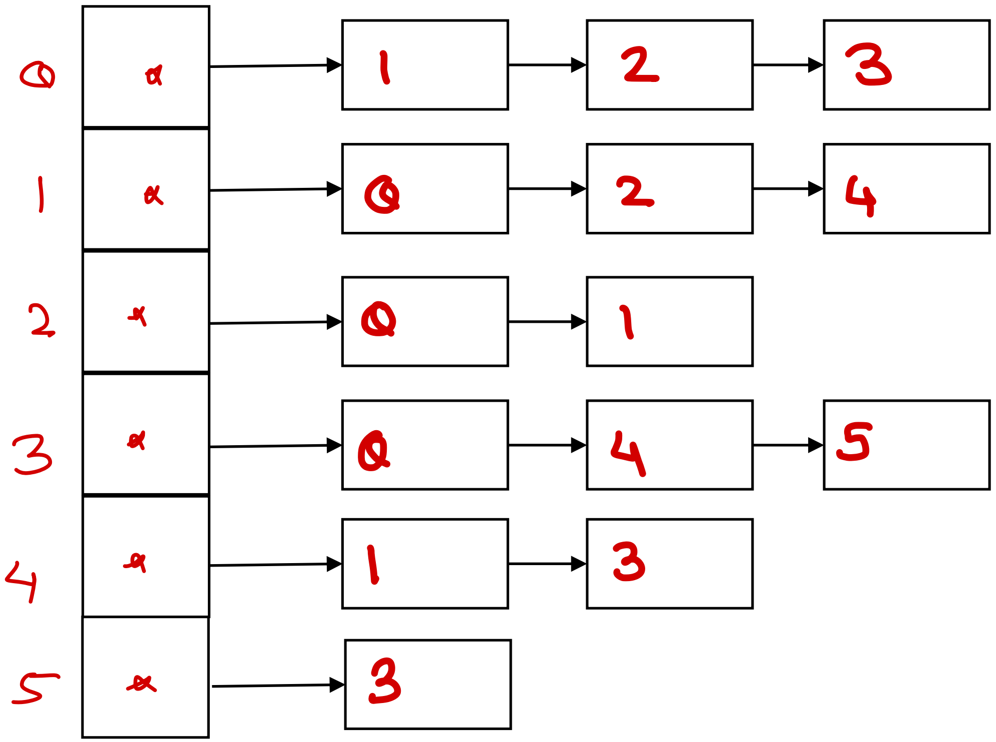


	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

Graph Implementation – Adjacency List

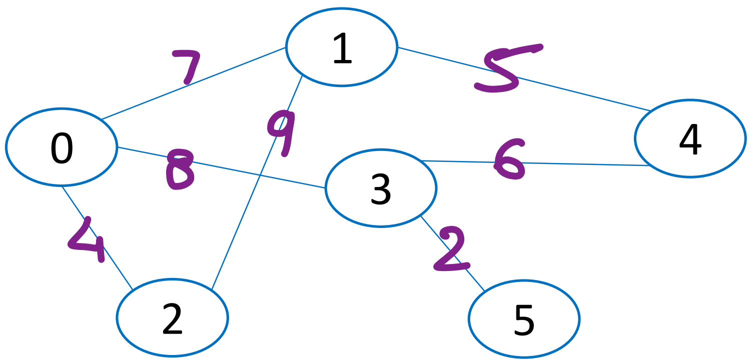
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbour vertices are stored.
- For weighted graph, neighbour vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V*E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).

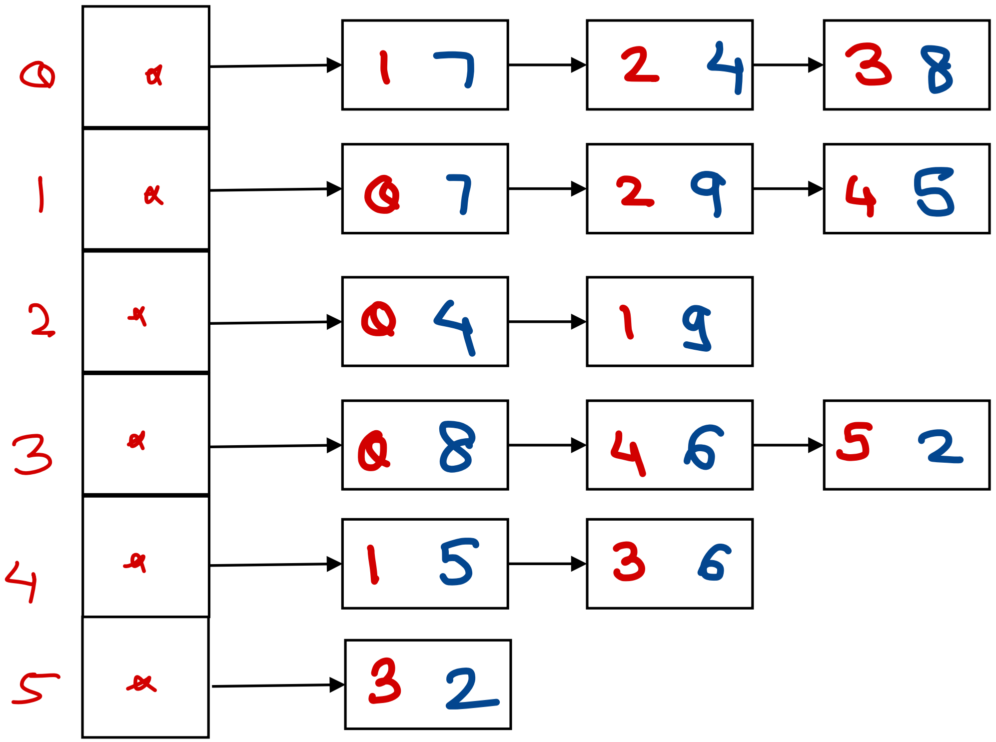




Graph Implementation – Adjacency List

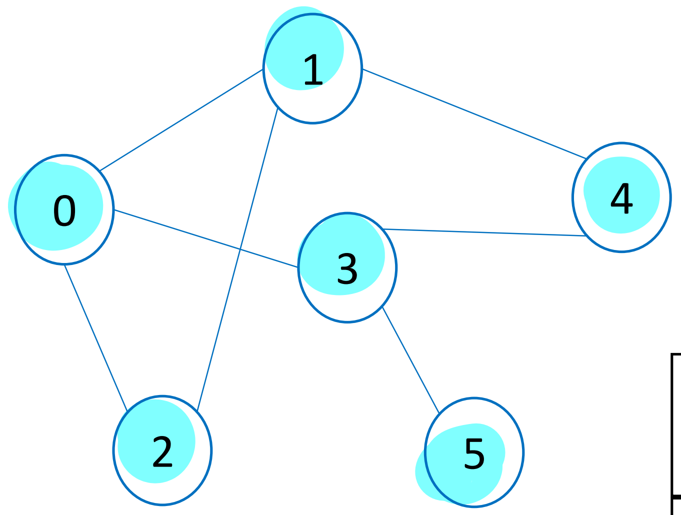
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbour vertices are stored.
- For weighted graph, neighbour vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).



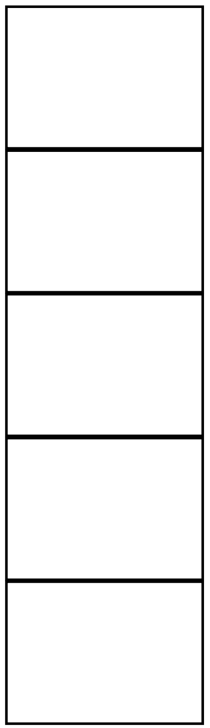


Graph Traversal – DFS Algorithm

1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Visit (Print) the vertex.
5. Put all non-~~visited~~ marked neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.



0 3 5 4 2 1



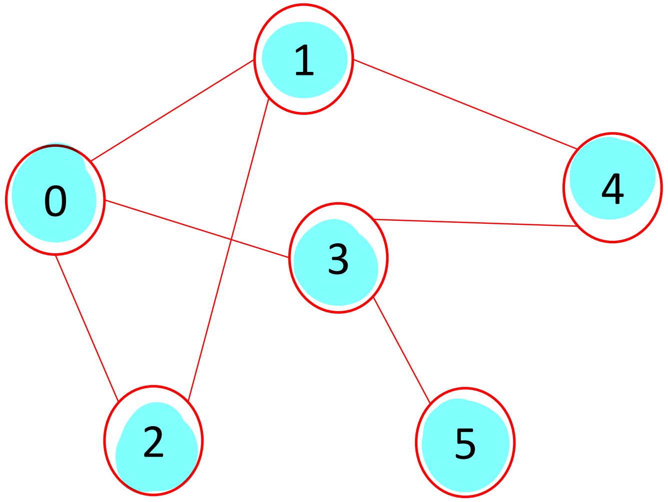
Graph Traversal – BFS Algorithm

1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it.
3. Pop vertex from queue.
4. Visit (Print) the vertex.
5. Put all non-~~visited~~ marked neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.

- BFS is also referred as level-wise search algorithm.

boolean marked[]

0	✓
1	✓
2	✓
3	✓
4	✓
5	✓



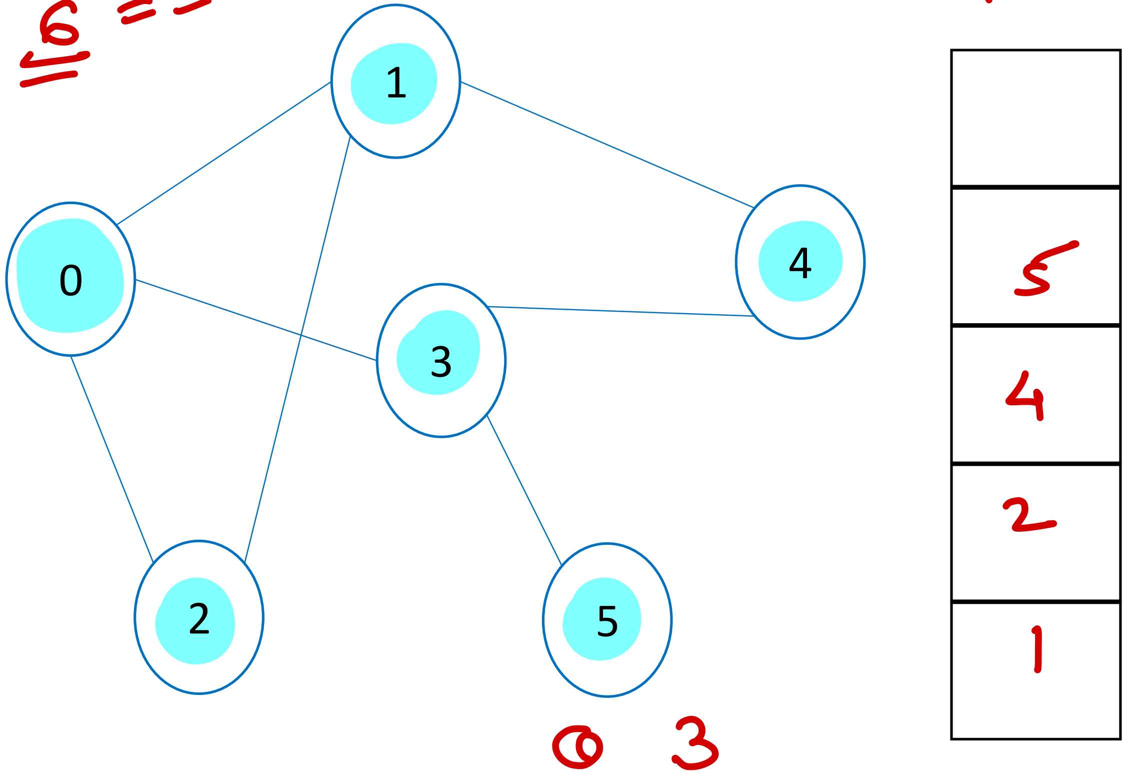
0 1 2 3 4 5

		0	1	2	3	4	5
		A	B	C	D	E	F
0	A	0	1	1	1	0	0
1	B	1	0	1	0	1	0
2	C	1	1	0	0	0	0
3	D	1	0	0	0	1	1
4	E	0	1	0	1	0	0
5	F	0	0	0	1	0	0

Check Connected-ness

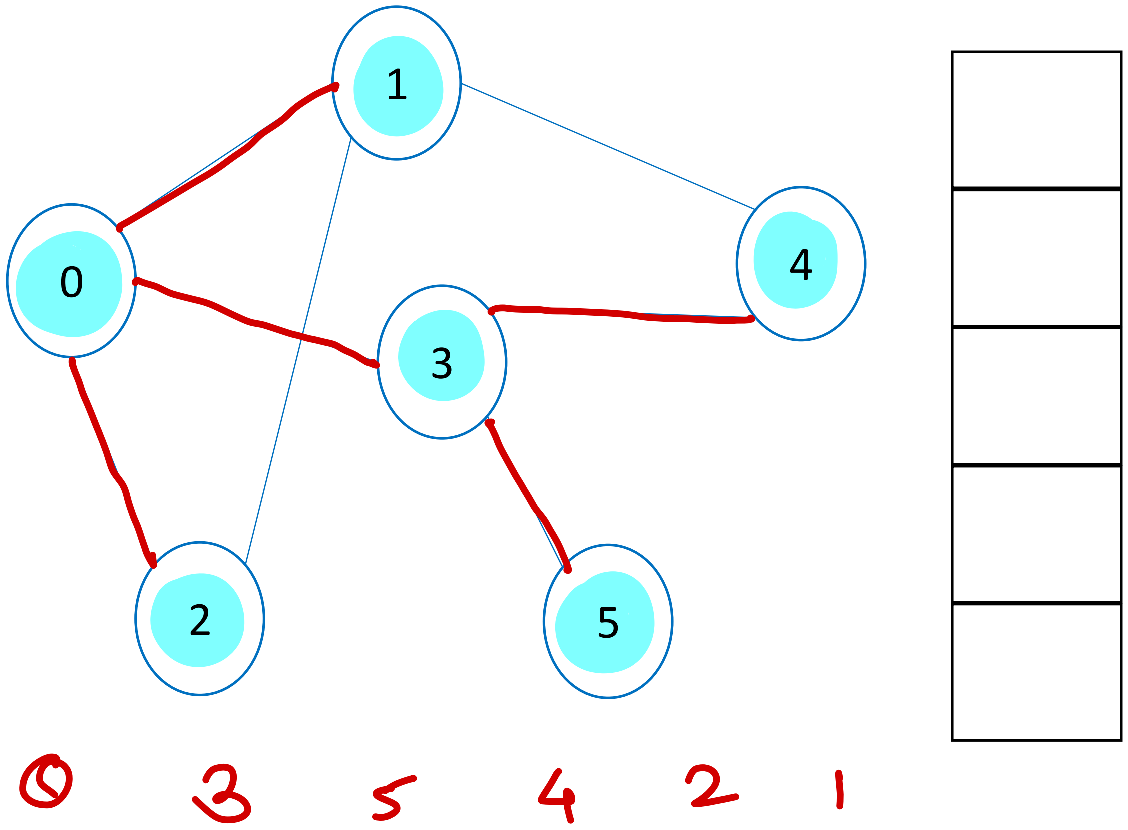
1. push starting vertex on stack & mark it.
2. begin counting marked vertices from 1.
3. pop a vertex from stack.
4. push all its non-marked neighbors on the stack, mark them and increment count.
5. if count is same as number of vertices, graph is connected (return).
6. repeat steps 3-5 until stack is empty.
7. graph is not connected (return)

6 == vertCount → Connected.



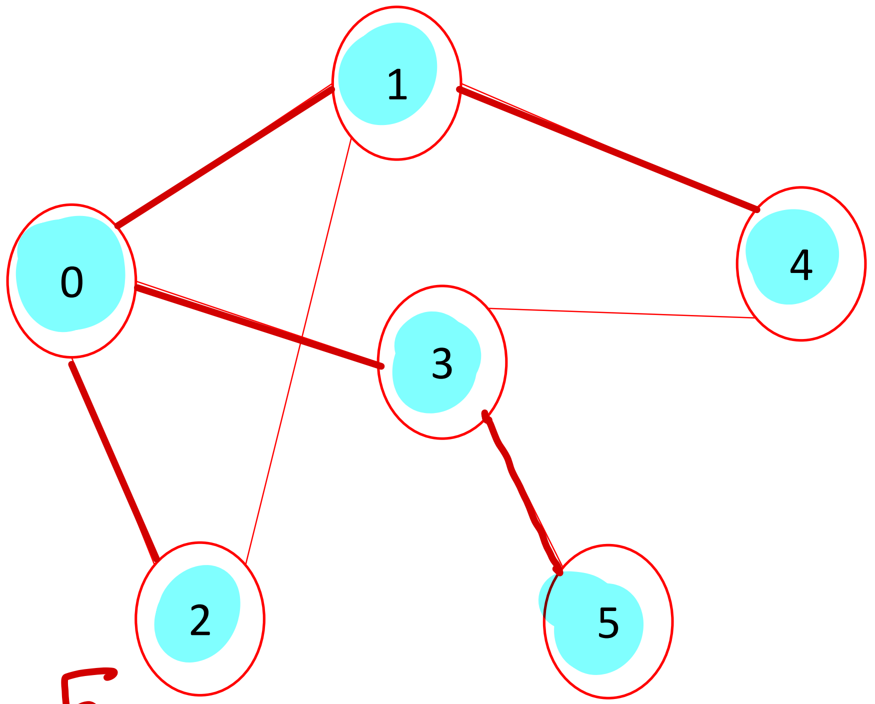
DFS Spanning Tree

1. push starting vertex on stack & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the stack, mark them. Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until stack is empty.



BFS Spanning Tree

1. push starting vertex on queue & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the queue, mark them. Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until queue is empty.



0 1 2 3 4 5

Thank you!

Nilesh Ghule <Nilesh@sunbeaminfo.com>