Sunbeam Institute of Information Technology

Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Priority Queue

- Queue where high priority data is always peeked or deleted.
- Priority can be implemented using array, linked list and heap data structures.
- An array is used to store a value and its associated priority. In some simpler implementations, the value itself might represent the priority (e.g., lower value means higher priority).
- Array and linked list implementation of priority queue often leads to less efficient performance compared to heap-based implementations for insertion and deletion operations.

- **Ordered vs. Unordered Array**
  - **Ordered Array** / linked list
    - Elements are kept sorted by priority. Insertion requires shifting existing elements to maintain order, leading to O(n) time complexity for insertion.
    - Deletion of the highest priority element is O(1) as it's typically at the beginning or end of the array.
  - **Unordered Array:**/ linked list
    - Elements are inserted without regard to order, making insertion O(1).
    - Deletion of the highest priority element requires searching the entire array to find it, resulting in O(n) time complexity for deletion.

Priority Queue Implementation

- Priority : number associated with value
- Priority range is defined by programmer

  e.g. Priority range : 1 to 10

Priority Queue

Higher value Higher priority (Max priority Queue)	Lower value Higher priority (Min priority Queue)

- Every element of priority queue will have two parts :

value : any data type
priority : integer

```
struct item {
    int value;
    int priority;
};
```

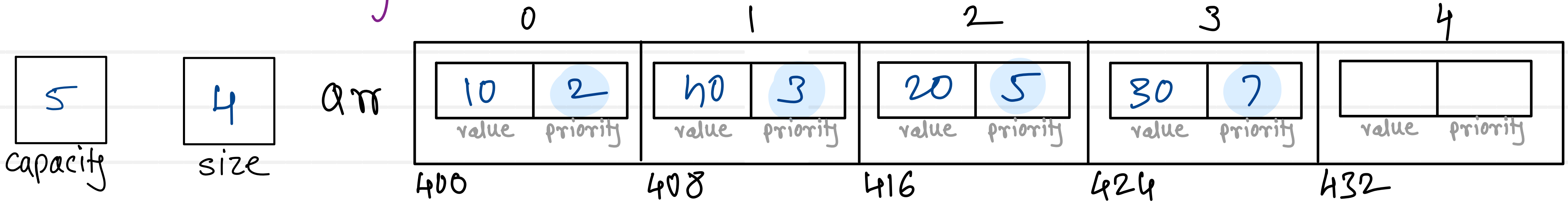
```
struct priorityQueue {
    struct item arr[5];
    int capacity;      ( maxSize)
    int size;
};
```

Operations :

1) Enqueue
2) Dequeue
3) Peek
4) isEmpty → size == 0
5) isFull → size == capacity

Priority Queue Implementation

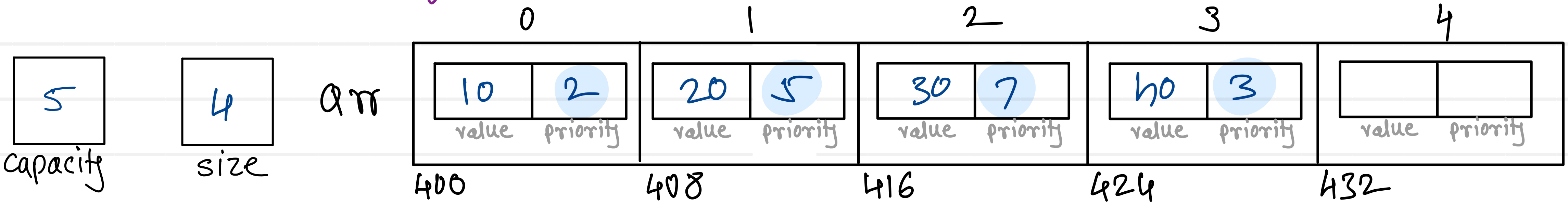
Ordered array:



insert - O(n)   delete - O(1)   peek - O(1)

for shifting - O(n)

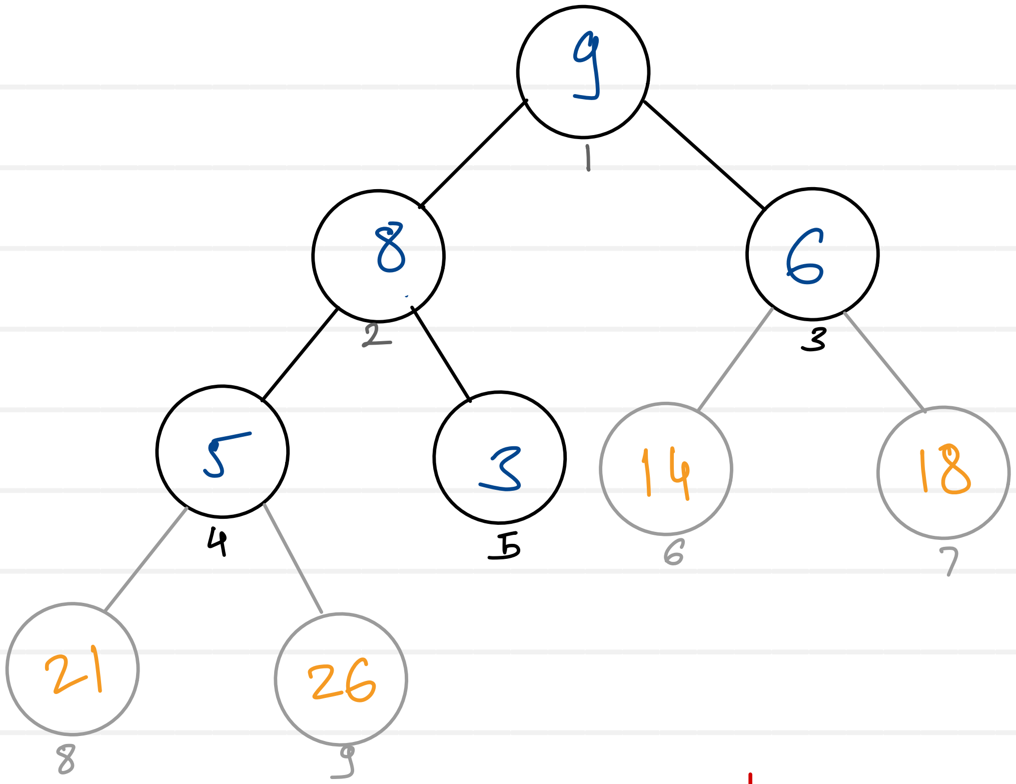
Unordered array:



insert - O(1)   delete - O(n)   peek - O(n)

for shifting - O(n)

Heap sort



9	8	6	5	3	14	18	21	26
1	2	3	4	5	6	7	8	9

arr

6	14	3	26	8	18	21	9	5
1	2	3	4	5	6	7	8	9

Algorithm:

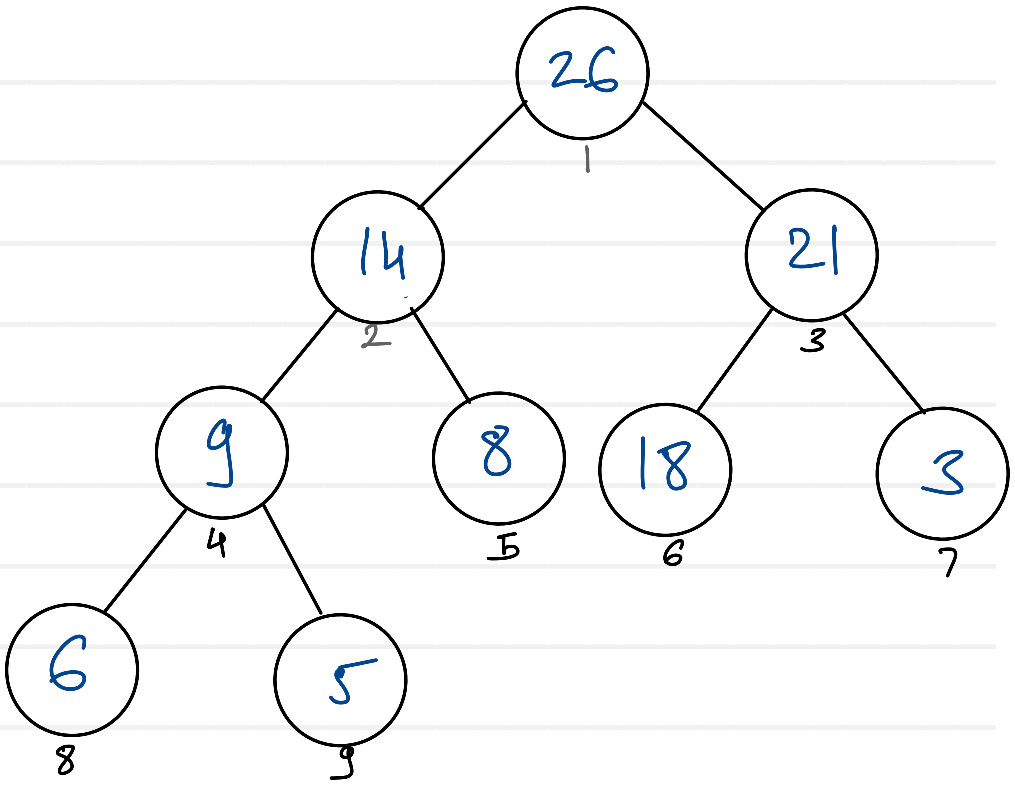
1. convert given array into either min heap / max heap
   (descending sort) (ascending sort)
2. delete elements from heap one by one and keep them into empty places of array from right side

to create heap $= n \log n$

to delete heap $= n \log n$

$2n \log n$

$\boxed{S(n) = O(1)}$ $\boxed{T(n) = O(n \log n)}$



26	14	21	9	8	18	3	6	5
1	2	3	4	5	6	7	8	9

size = 9

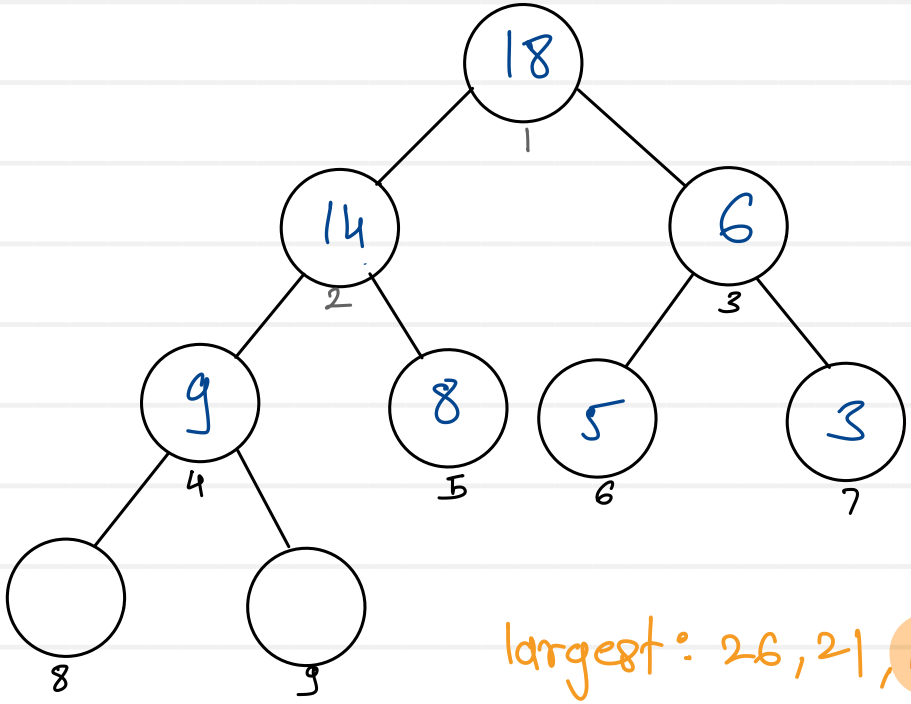
last parent index = $9/2 = 4$

last parent = $\frac{size}{2}$

k^{th} Largest

$k = 3$

arr	6	14	3	26	8	18	21	9	5
	1	2	3	4	5	6	7	8	9

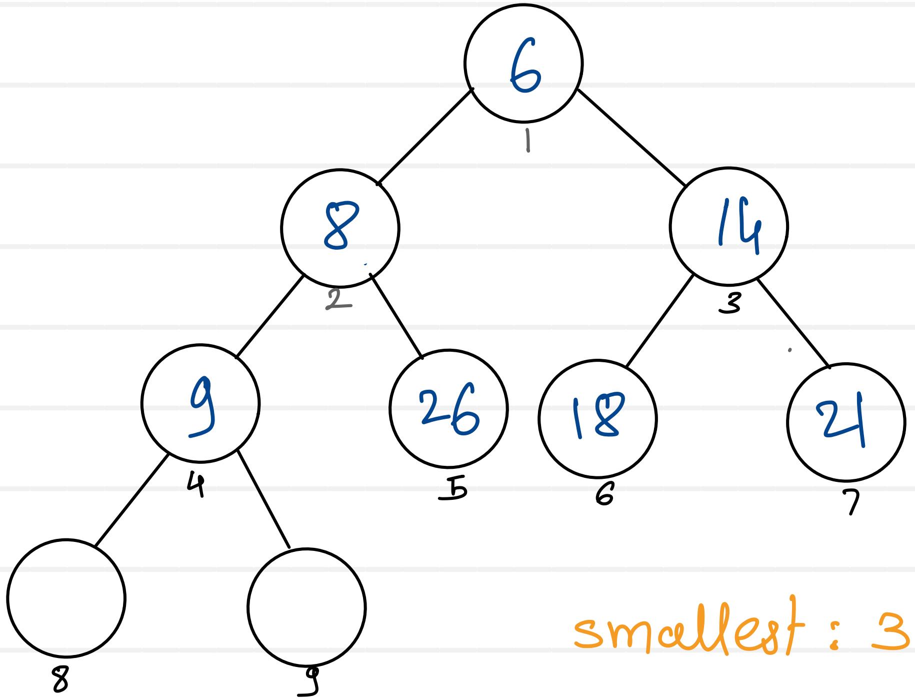


largest : 26, 21, 18

k^{th} smallest

$k = 3$

arr	6	14	3	26	8	18	21	9	5
	1	2	3	4	5	6	7	8	9



smallest : 3, 5, 6

Sliding window Technique

- involve moving a fixed or variable-size window through a data structure, to solve problems efficiently.
- This technique is used to find subarrays or substrings according to a given set of conditions.
- This method used to efficiently solve problems that involve defining a **window** or **range** in the input data and then moving that window across the data to perform some operation within the window.
- This technique is commonly used in algorithms like
  - **finding subarrays** with a specific sum
  - **finding the longest substring** with unique characters
  - solving problems that require a fixed-size window to process elements efficiently.

- There are two types of sliding window
  - **Fixed size sliding window**
    - Find the size of the window required
    - Compute the result for 1st window
    - Then use a loop to slide the window by 1 and keep computing the result
  - **Variable size sliding window**
    - increase right pointer one by one till our condition is true.
    - At any step if condition does not match, shrink the size of window by increasing left pointer.
    - Again, when condition satisfies, start increasing the right pointer
    - follow these steps until reach to the end of the array

Maximum Average Subarray

You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose length is equal to k that has the maximum average value and return this value. Any answer with a calculation error less than 10-5 will be accepted.

maxSum = ~~2~~, 51

windowSum

2

51

42

Example 1:
Input: nums = [1,12,-5,-6,50,3], k = 4
Output: 12.75000
Explanation: Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75

Example 2:
Input: nums = [5], k = 1
Output: 5.00000

1	12	-5	-6	50	3
0	1	2	3	4	5

k=4

1. decide window size : size = 4
2. compute result of first window

```
windowSum = 0;
for( i=0; i<k ; i++)
   windowSum += nums[i];
maxSum = windowSum;
```

3. slide window by 1 till last index

```
for (i=k ; i<n ; i++) {
   windowSum = windowSum + nums[i]
                         - nums[i-k];
   if(windowSum > maxSum)
      maxSum = windowSum;
}
```

4. return max Avg.

```
return maxSum/k;
```

Maximum Length Substring With Two Occurrences

Given a string s, return the maximum length of a substring such that it contains at most two occurrences of each character.

Example 1:
Input: s = "bcbbbcba"
Output: 4
Explanation: The following substring has a length of 4 and contains at most two occurrences of each character: "bcbbbcba".

Example 2:
Input: s = "aaaa"
Output: 2
Explanation: The following substring has a length of 2 and contains at most two occurrences of each character: "aaaa".

maxLength=4

arr	1	2	1			
	a	b	c	-	---	

S	b	c	b	b	b	c	b	a
	0	1	2	3	4	5	6	7
					start			end

```
int maxLength=0;
int start=0, end=0;
int[] arr = new int[26];
for( ; end < s.length() ; end++){
    arr[s.charAt(end) - 'a']++;
    while(arr[s.charAt(end) - 'a']==3){
        arr[s.charAt(start)-'a']--;
        start++;
    }
    maxLength = Math.max(maxLength, end-start+1);
}
return maxLength;
}
```

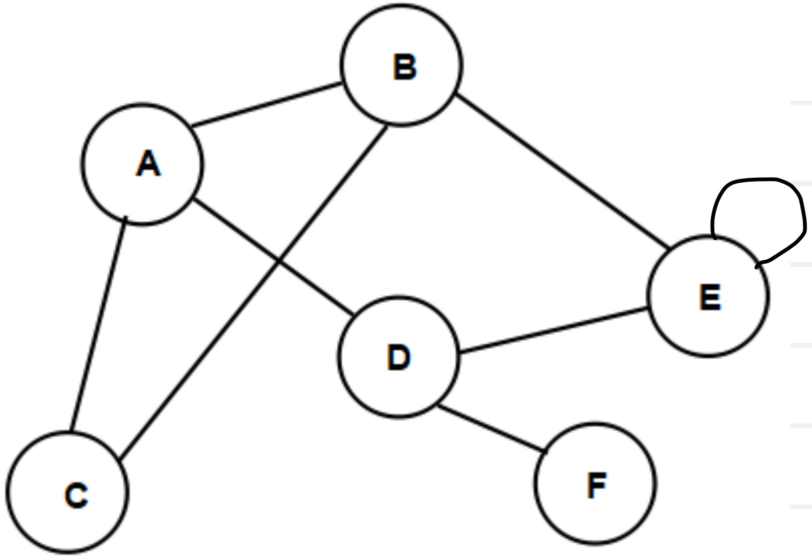
Graph : Terminologies

- **Graph** is a non linear data structure having set of vertices (nodes) and set of edges (arcs).
  - G = {V, E}

    Where V is a set of vertices and E is a set of edges
  - **Vertex (node)** is an element in the graph

    V = {A, B, C, D, E, F}
  - **Edge (arc)** is a line connecting two vertices

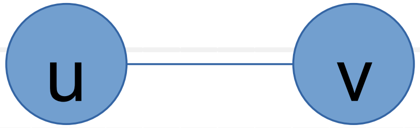
    E = {(A,B), (A,C), (B,C), (B,E), (D, E), (D,F),(A,D)}



- Vertex A is set be **adjacent** to B, if and only if there is an edge from A to B.
- **Degree of vertex :-** Number of vertices adjacent to given vertex
- **Path :-** Set of edges connecting any two vertices is called as path between those two vertices.
  - Path between A to D = {(A, B), (B, E), (E, D)}
- **Cycle :-** Set of edges connecting to a node itself is called as cycle.
  - {(A, B), (B, E), (E, D), (D, A)}
- **Loop :-** An edge connecting a node to itself is called as loop. Loop is smallest cycle.

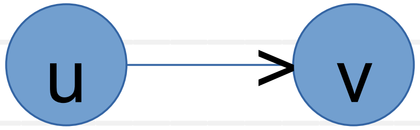
Graph : Types

- **Undirected graph.**
  - If we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**
  - **graph which contains undirected edges referred as undirected graph.**



(u, v) == (v, u)

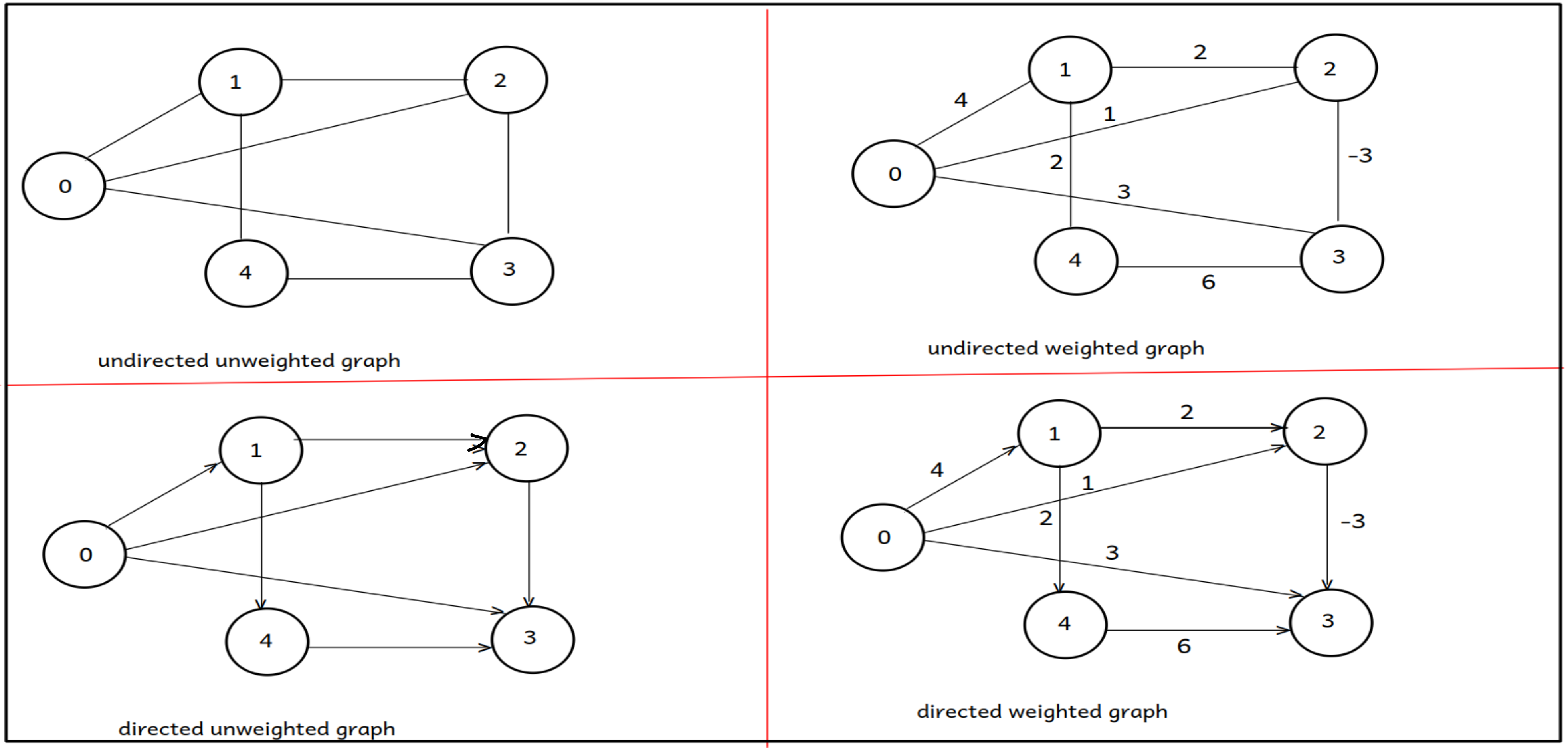
- **Directed Graph (Di-graph)**
  - If we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **ordered pair of vertices** i.e. directed edge.
  - **graph which contains set of directed edges referred as directed graph (di-graph).**
  - graph in which each edge has some direction



(u, v) != (v, u)

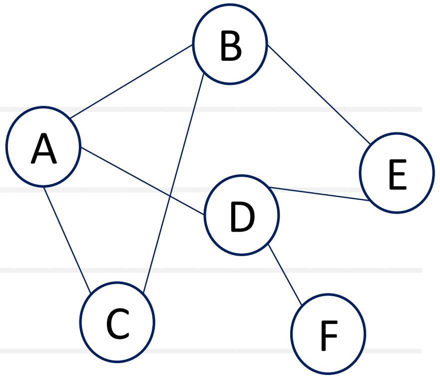
Graph : Types

- **Weighted Graph**
  - A graph in which edge is associated with a number (ie weight)



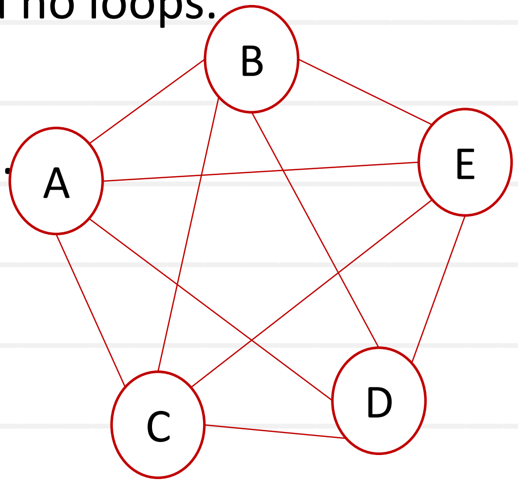
Graph : Types

- **Simple Graph**
  - Graph not having multiple edges between adjacent nodes and no loops.

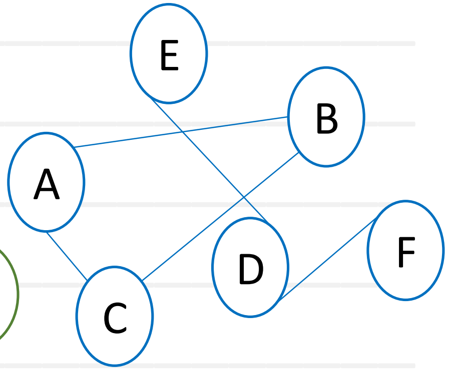


- **Complete Graph**
  - Simple graph in which node is adjacent with every other node.
  - Un-Directed graph: Number of Edges = n (n -1) / 2

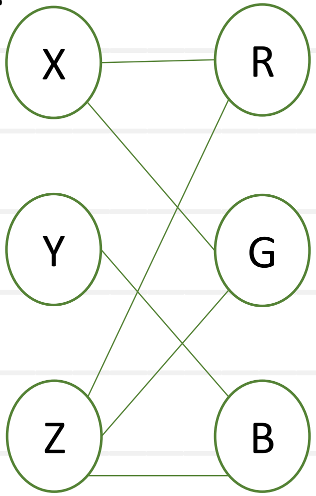
    where, n – number of vertices
  - Directed graph: Number of edges = n (n-1)



- **Connected Graph**
  - Simple graph in which there is some path exist between any two vertices.
  - Can traverse the entire graph starting from any vertex.

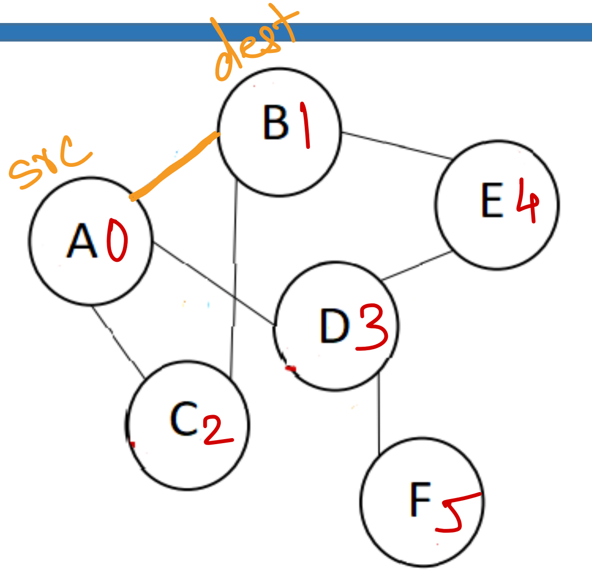


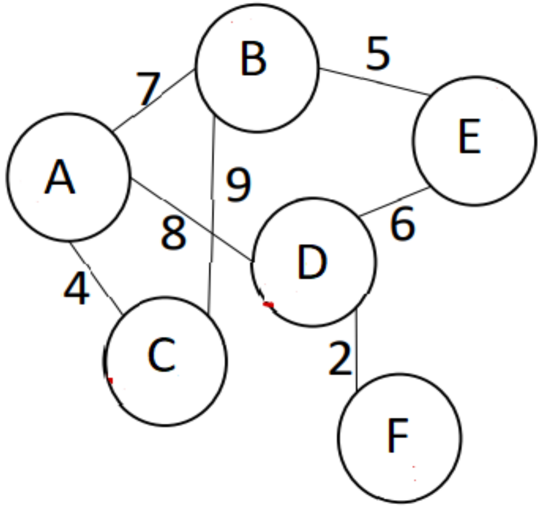
- **Bi-partite graph**
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.



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 non-weighted graph, 1 indicate edge and 0 indicate no edge.
- 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).



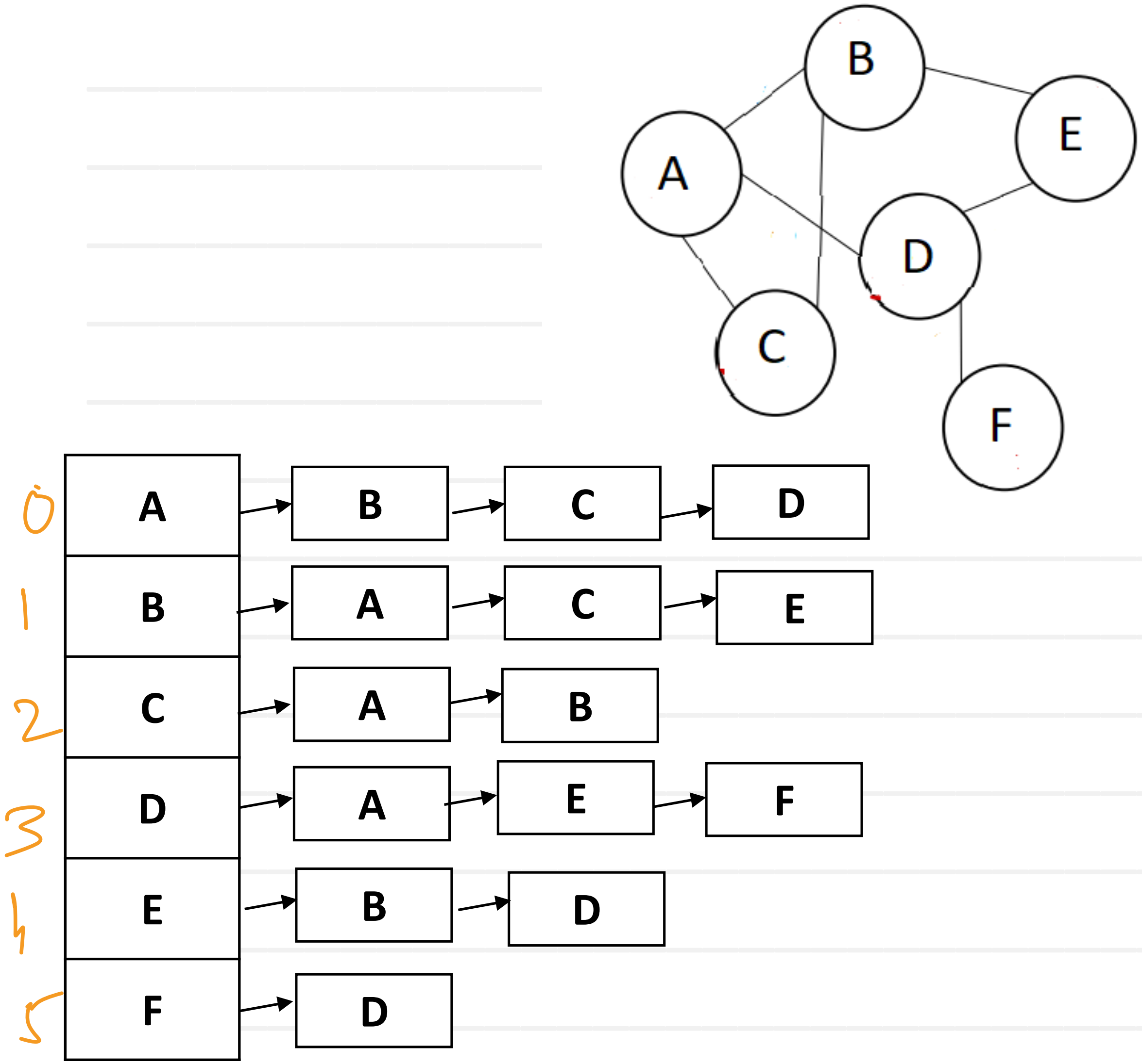


	A 0	B 1	C 2	D 3	E 4	F 5
A 0	0	1	1	1	0	0
B 1	1	0	1	0	1	0
C 2	1	1	0	0	0	0
D 3	1	0	0	0	1	1
E 4	0	1	0	1	0	0
F 5	0	0	0	1	0	0

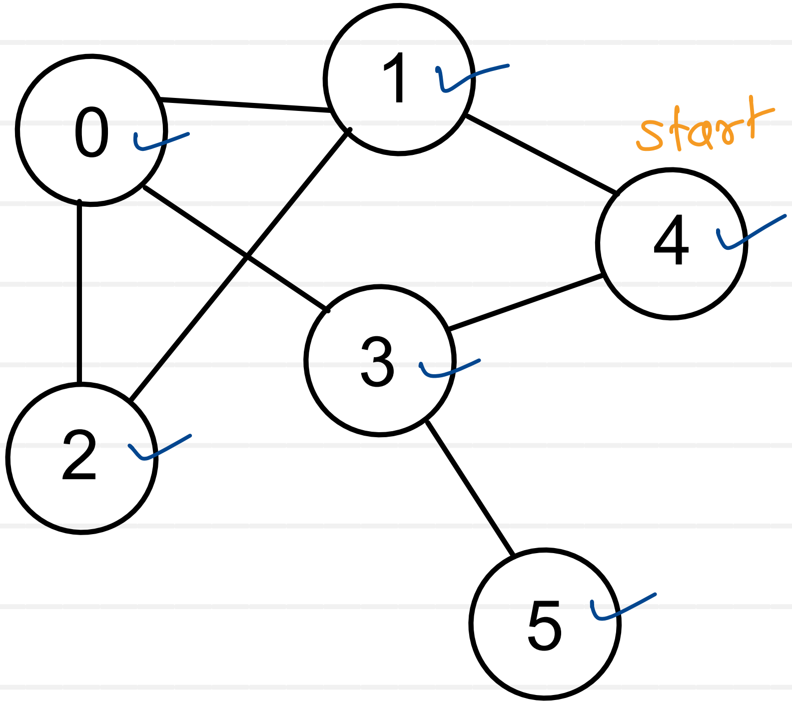
	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

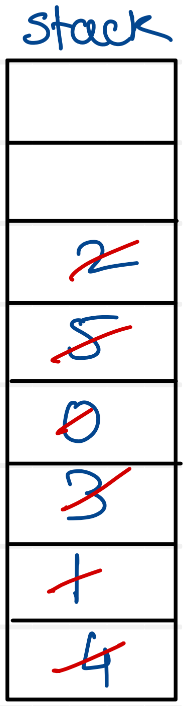
Graph Implementation – Adjacency List

- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbor vertices are stored.
- For weighted graph, neighbor 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).



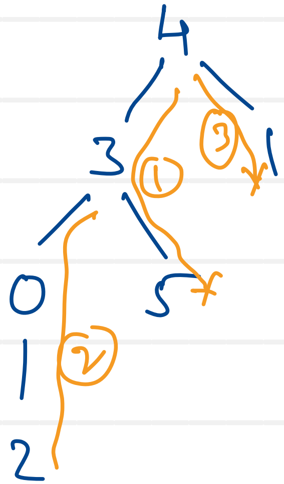
DFS Traversal



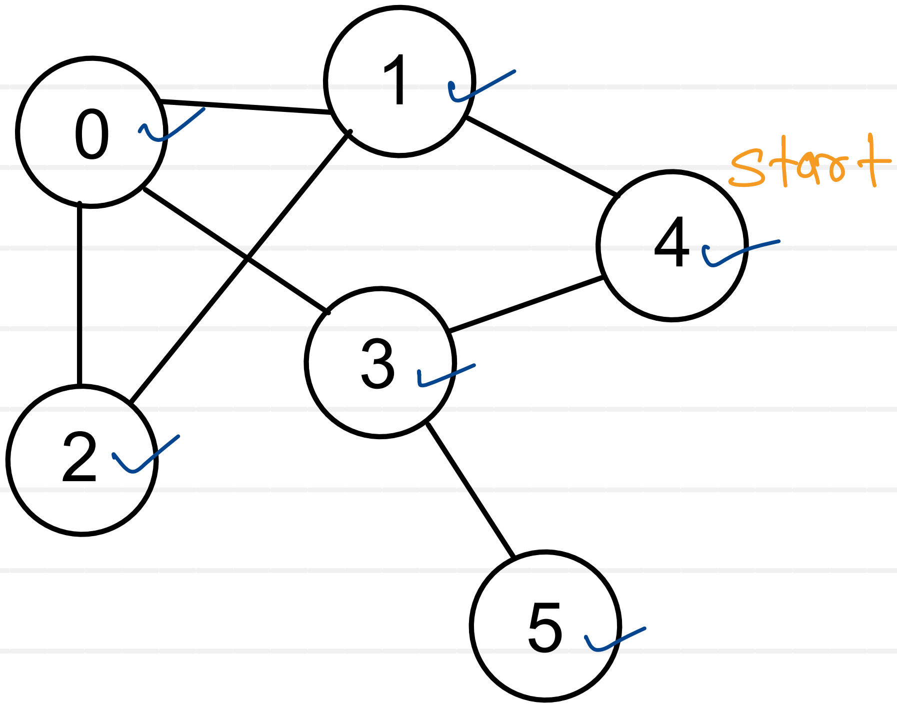


1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.

Traversal : 4, 3, 5, 0, 2, 1



BFS Traversal

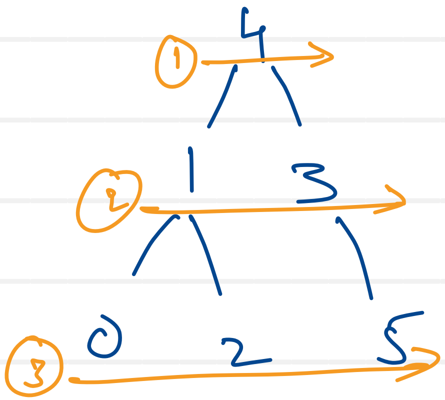


Queue

~~4~~	~~1~~	~~3~~	~~0~~	~~2~~	~~5~~		

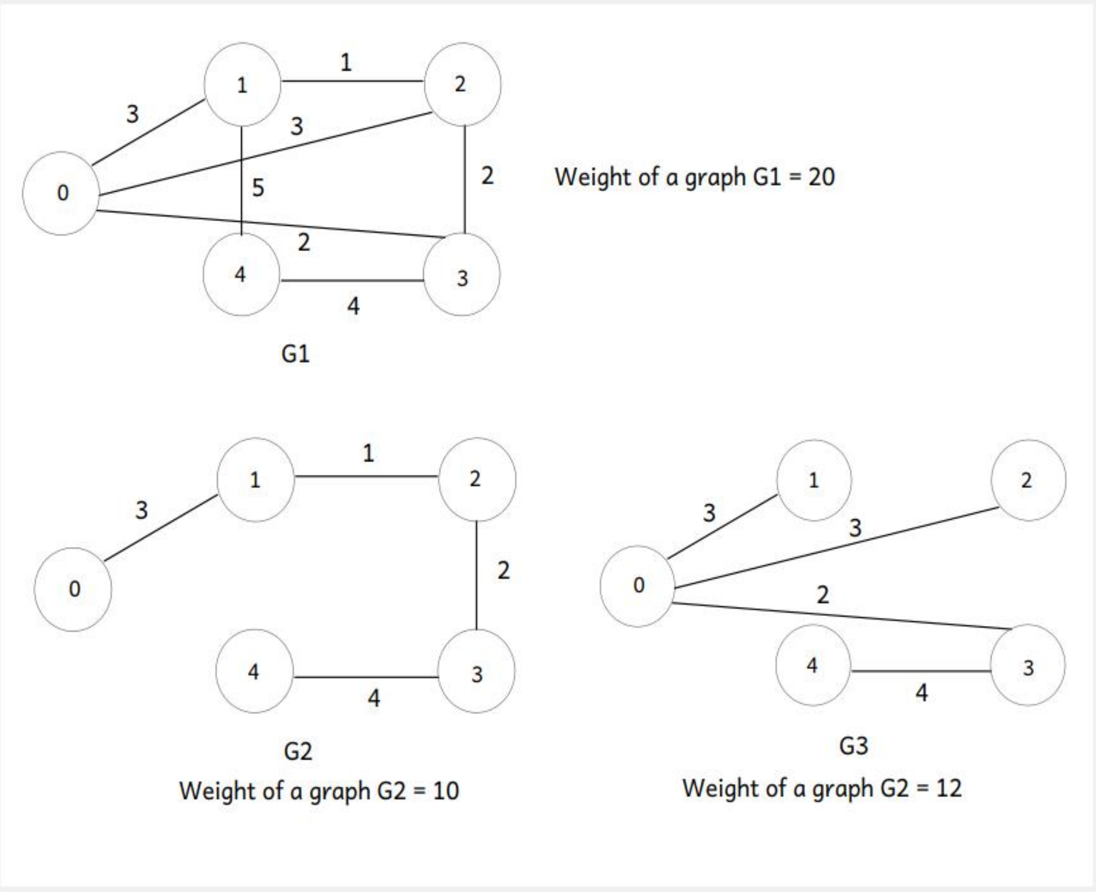
Traversal : 4, 1, 3, 0, 2, 5

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

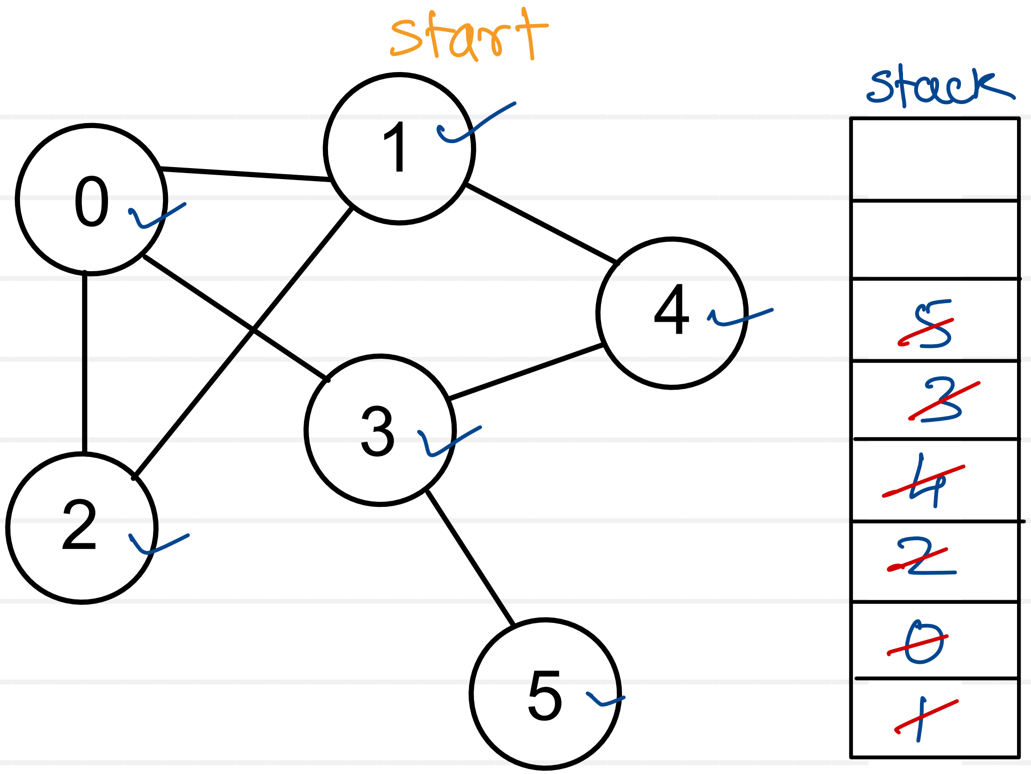


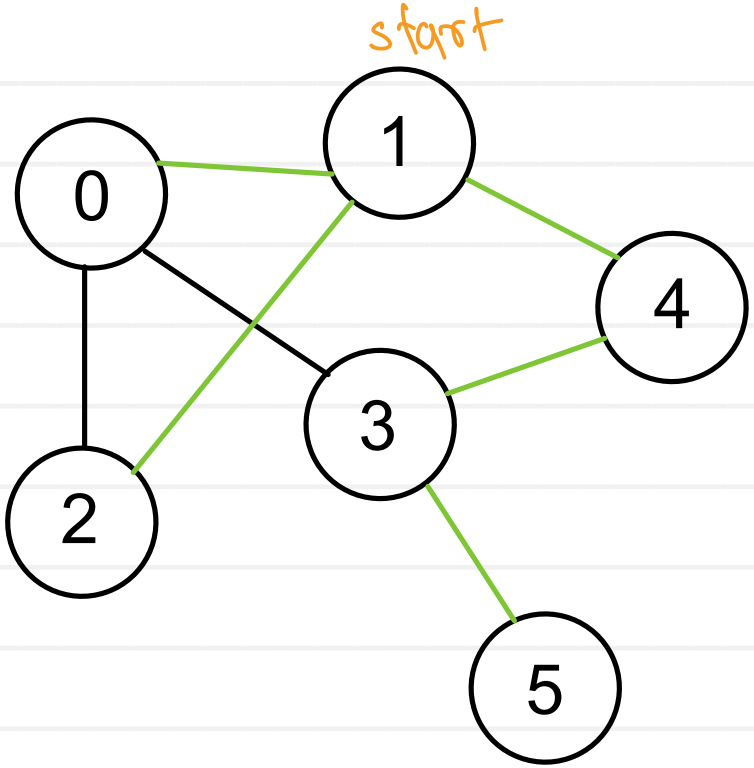
Spanning Tree

- Tree is a graph without cycles. Includes all V vertices and V-1 edges.
- Spanning tree is connected sub-graph of the given graph that contains all the vertices and sub-set of edges.
- 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



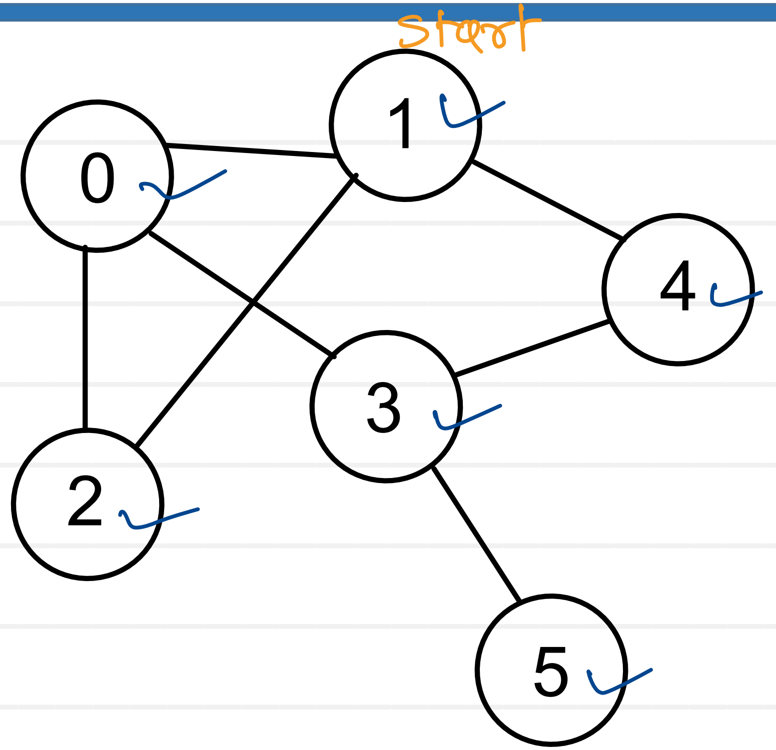
DFS spanning tree

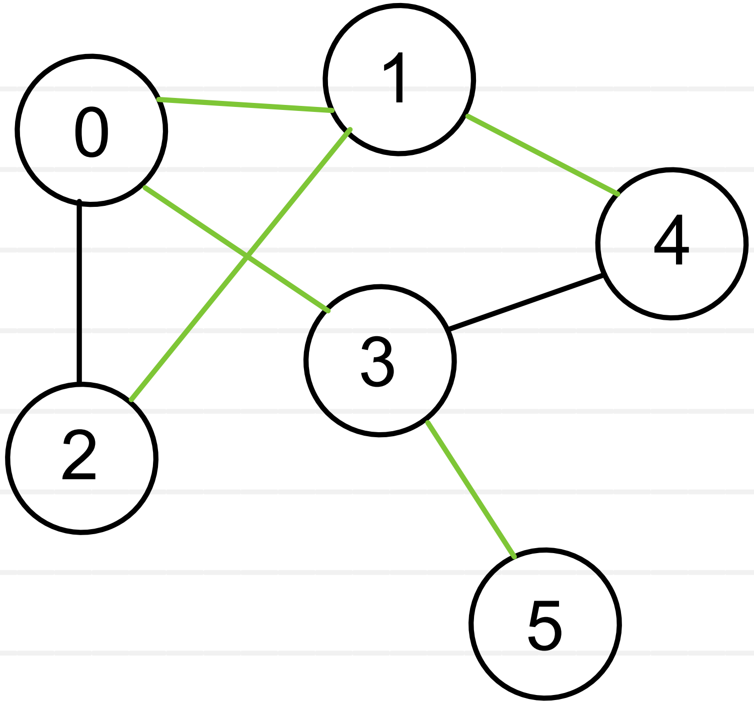




Spanning tree: (1-0), (1-2) (1-4), (4-3) (3-5)

BFS spanning tree





Queue	~~1~~	~~0~~	~~2~~	~~4~~	~~3~~	~~5~~		

spanning tree : (1-0), (1-2), (1-4) (0-3) (3-5)

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com