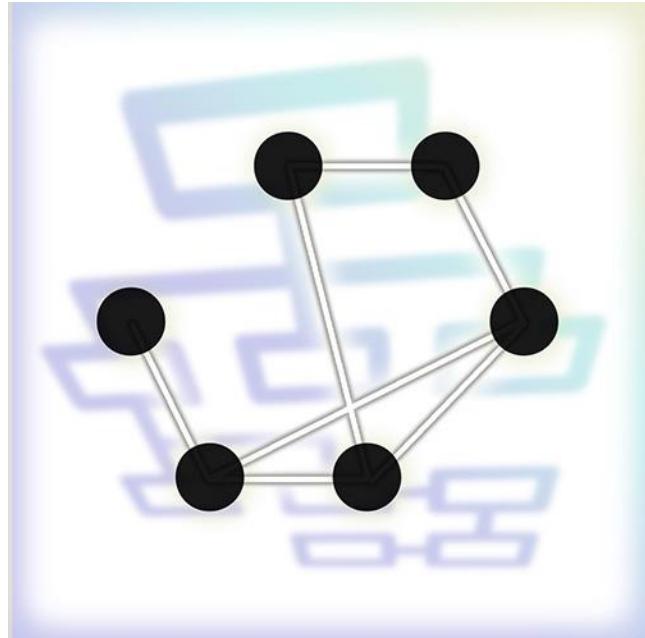


DATA STRUCTURES & ALGORITHMS



UNIT NO. 5 GRAPH

Topic Name
Graph : Introduction

Graph: Introduction

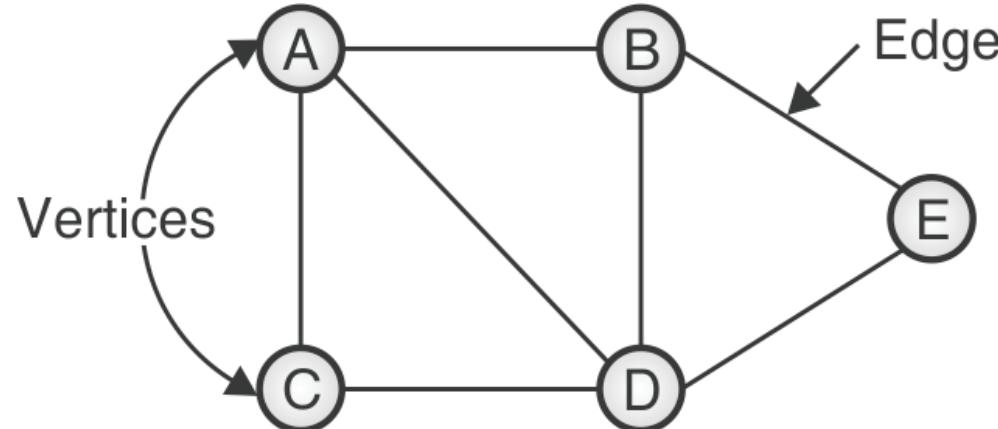
- Data structures are mainly categorized as linear and non-linear data structures.
- We have seen various linear data structures like Stack, Queue and Linked list while one non-linear data structure : Tree.
- Now we are going to learn another non-linear data structure that is Graph.

Graph: Introduction

- Graphs are very powerful and versatile data structures that easily allow us to represent real life relationships between different types of data (nodes).
- There are two main parts of a graph:
 1. The vertices (nodes) where the data is stored
 2. The edges (connections) which connect the nodes

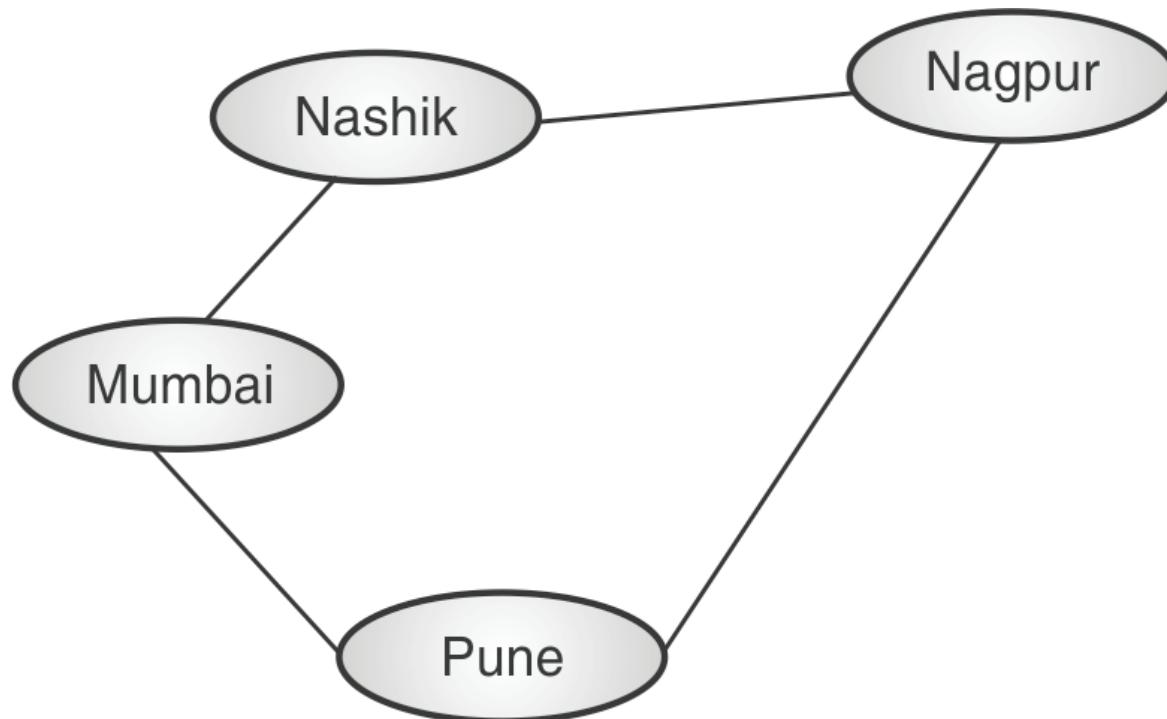
Graph: Definition

- A graph with 5 vertices and 6 edges.
- This graph G can be defined as $G = (V, E)$
Where $V = \{A, B, C, D, E\}$ and
 $E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$.



Graph: Definition

- the vertices represents the cities while edges represents the road between two cities.



Graph: Example

There are number of real life examples of graph.

- Representation of road network of cities.
- Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
- Using GPS/Google Maps/Yahoo Maps, to find a route based on shortest route.
- Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks; each page is a vertex and the link between two pages is an edge.
- On eCommerce websites, relationship graphs are used to show recommendations.

Terminologies of Graph

- Node (Vertex)
- Arc (Edge)
- Directed Edge
- Undirected Graph
- Directed Graph
- Degree
- In-degree
- Out-degree
- Adjacent
- Successor
- Predecessor

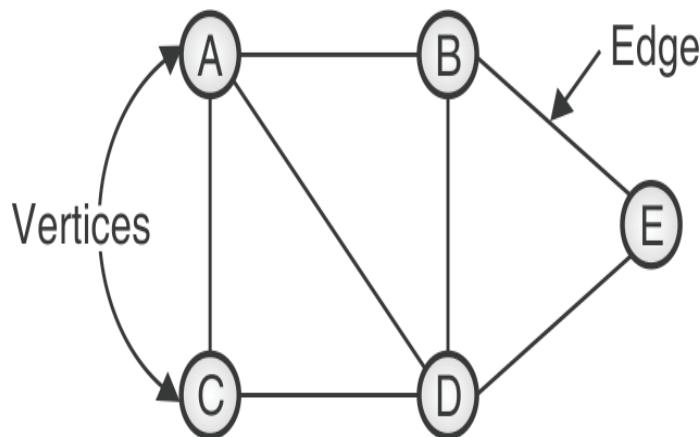
Terminologies of Graph

- Relation
- Weight
- Weighted Graph
- Path
- Length
- Linear Path
- Subgraph
- Source
- Isolated Node
- Sink

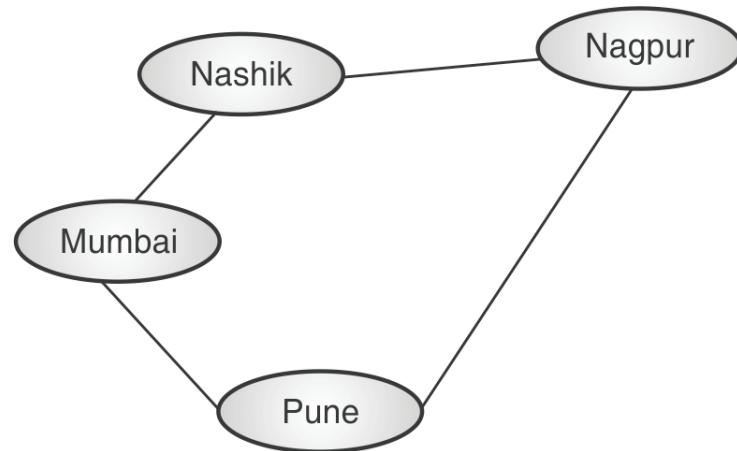
Terminologies of Graph

Node (Vertex)

- Every individual element in a graph is known as Vertex. Vertex is also called as a Node.



A, B, C, D and E are vertices.

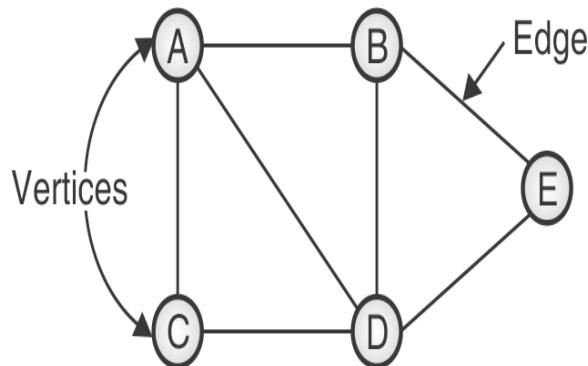


Pune, Mumbai, Nasik and Nagpur are vertices.

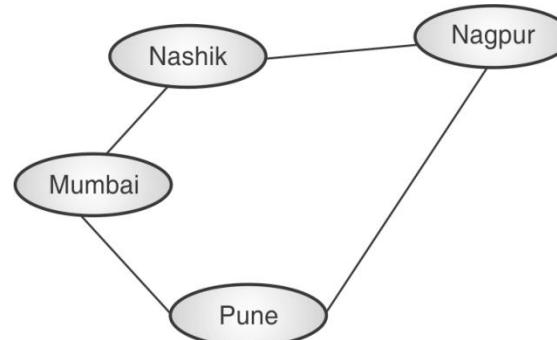
Terminologies of Graph

Arc (Edge)

- An Edge is the connecting link between two vertices. Edge is also called as Arc.
- An edge is represented as (startingVertex, endingVertex).



There are 7 edges : (A,B), (A,C),
(A, D), (B, D), (B, E), (C, D) and (D, E).

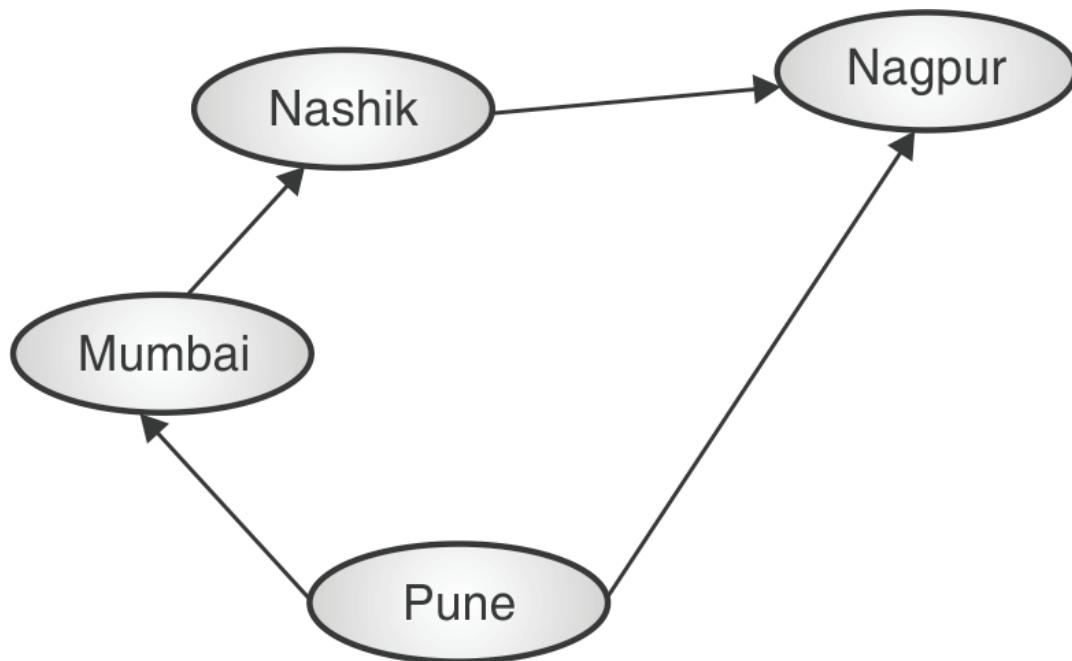


There are 4 edges : (Pune, Mumbai),
(Pune, Nagpur), (Mumbai, Nasik) and (Nasik,
Nagpur).

Terminologies of Graph

Directed Edge

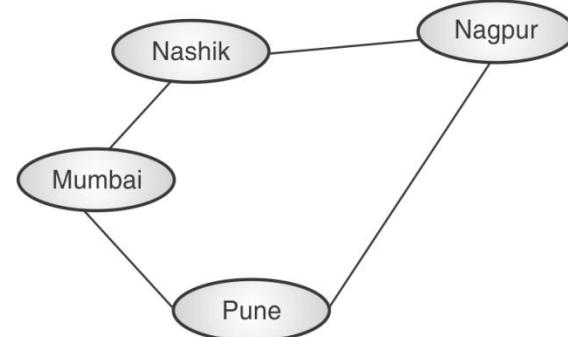
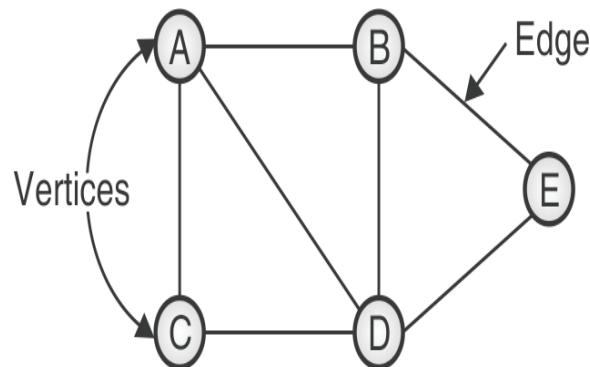
- Definition : The edge which has specific directions, is called as directed edge.



Terminologies of Graph

Undirected Graph

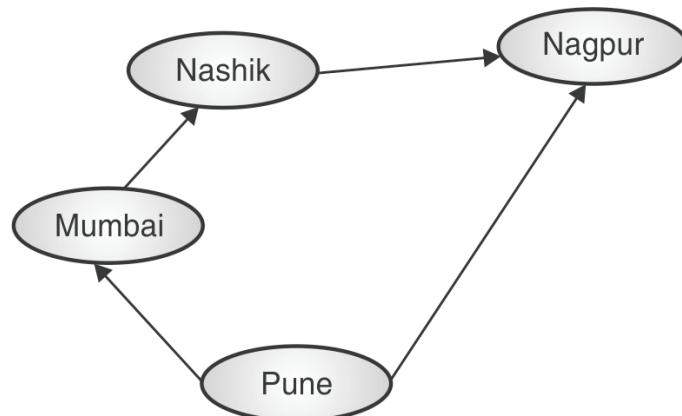
- The graph in which the edges do not show any direction is called as Undirected Graph.
- The edges which do not have directions are called as undirected edges.
- In undirected graph every pair of vertices make two edges.
- For example, in the Fig. 3.1.2 vertices Pune and Mumbai represents two edges as (Pune, Mumbai) and (Mumbai, Pune). Both the edges are considered as same.



Terminologies of Graph

Directed Graph

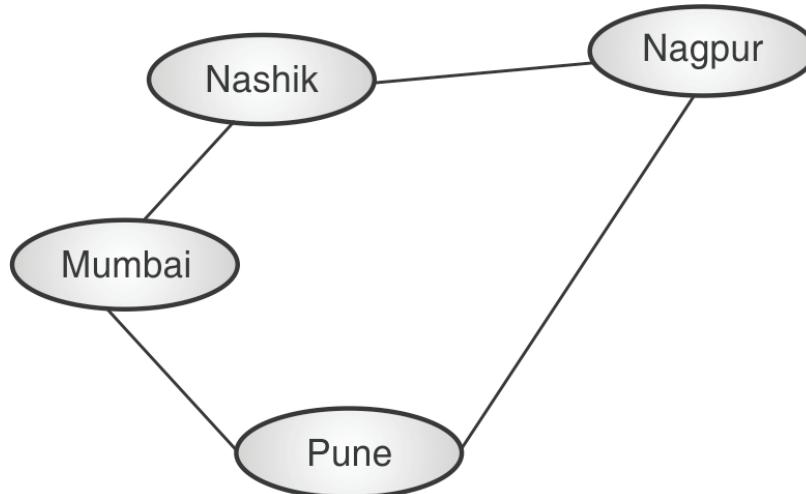
- There is need to show directions to the edges. we may want to show directions to indicate whether the travelling is from Pune to Mumbai or Mumbai to Pune.
- The graph in which all the edges have specific directions is called as directed graph.
- The edges which have directions are called as directed edges.



Terminologies of Graph

Degree

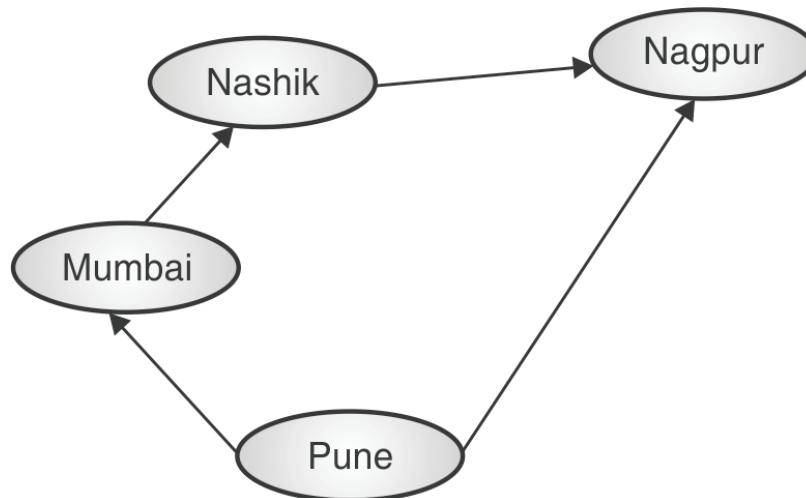
- Total number of edges connected to a vertex is said to be degree of that vertex.
- In undirected graph, every vertex has degree 2.



Terminologies of Graph

In-Degree

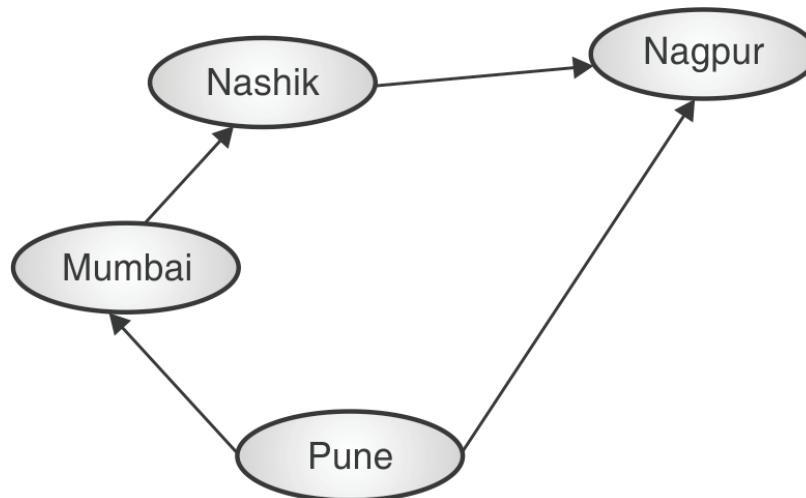
- Total number of incoming edges connected to a vertex is said to be in-degree of that vertex.
- In directed graph, the in-degree of vertex pune is 0 while in-degree of vertex Nagpur is 2.



Terminologies of Graph

Out-Degree

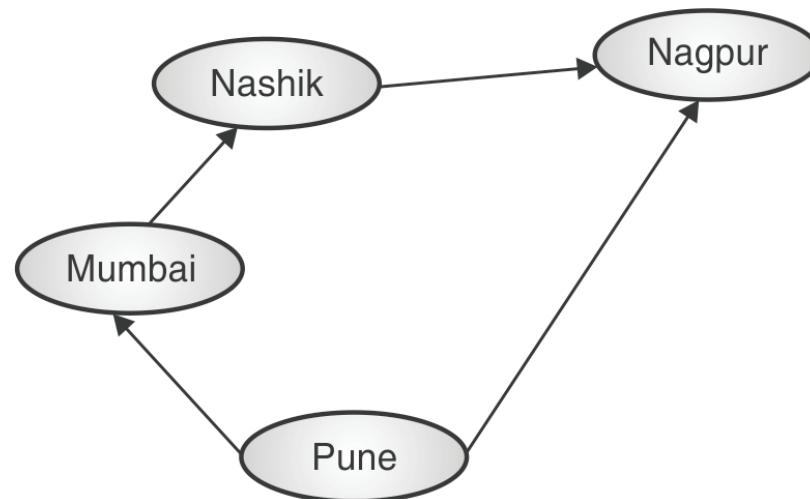
- Total number of outgoing edges connected to a vertex is said to be out-degree of that vertex.
- In directed graph, the out-degree of vertex pune is 2 while in-degree of vertex Nagpur is 0.



Terminologies of Graph

Adjacent

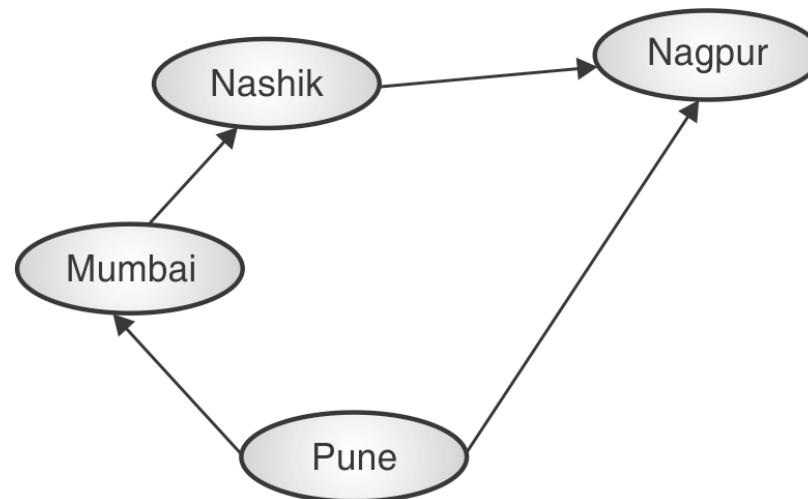
- If there is an edge between vertices A and B then both A and B are said to be adjacent.
- In directed graph, the vertices Pune and Mumbai are adjacent.



Terminologies of Graph

Successor

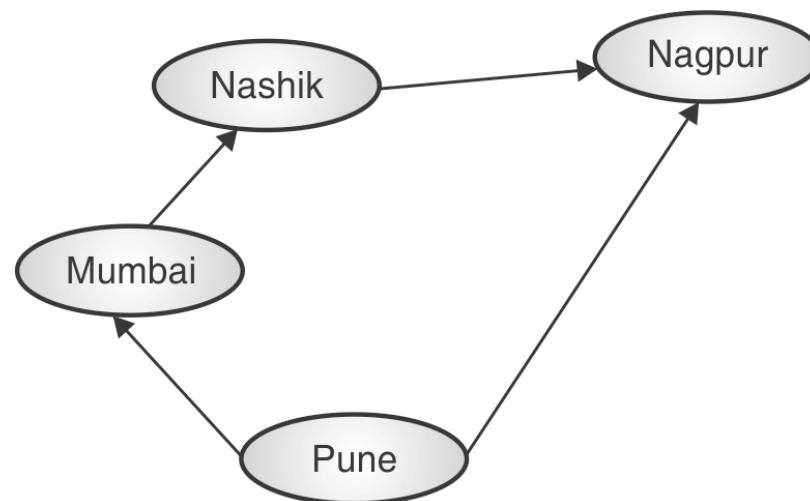
- A vertex coming after a given vertex in a directed path is called as successor.
- In directed graph, the vertex Mumbai is successor of vertex Pune.



Terminologies of Graph

Predecessor

- A vertex coming before a given vertex in a directed path is called as predecessor.
- In directed graph, the vertex Pune is predecessor of vertex Mumbai.



Terminologies of Graph

Relation

- Relation is set of ordered pairs.

Terminologies of Graph

Weight

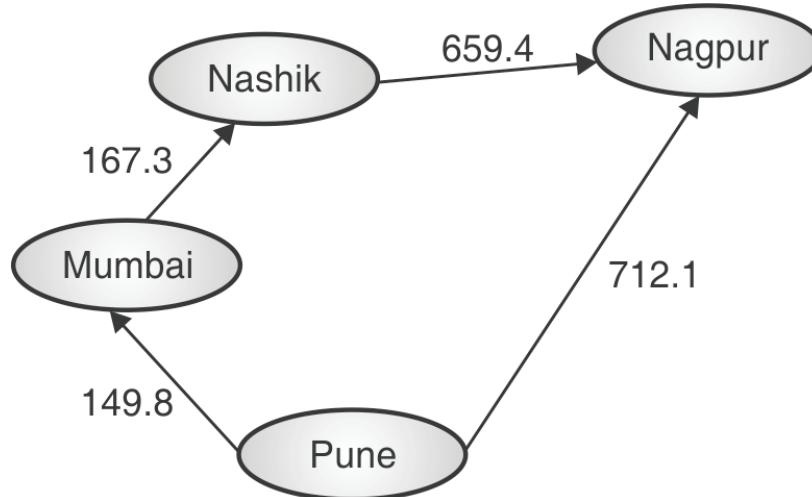
- A numerical value, assigned as a label to a vertex or edge of a graph is called as weight.

- The weight may indicate distance between two cities or any other resources such cost, time etc.

Terminologies of Graph

Weighted Graph

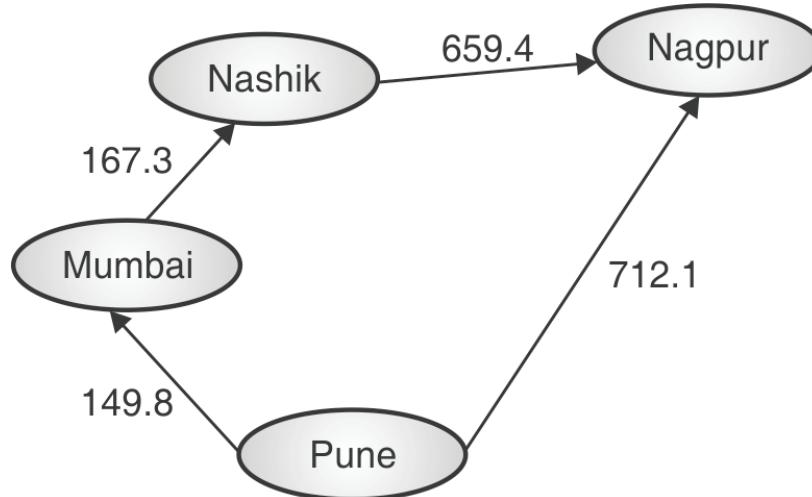
- When weight is assigned to each and every edge of a graph, then such graph is called Weighted Graph.
- the weights (numbers associated to edges) represent distance between two cities.



Terminologies of Graph

Path

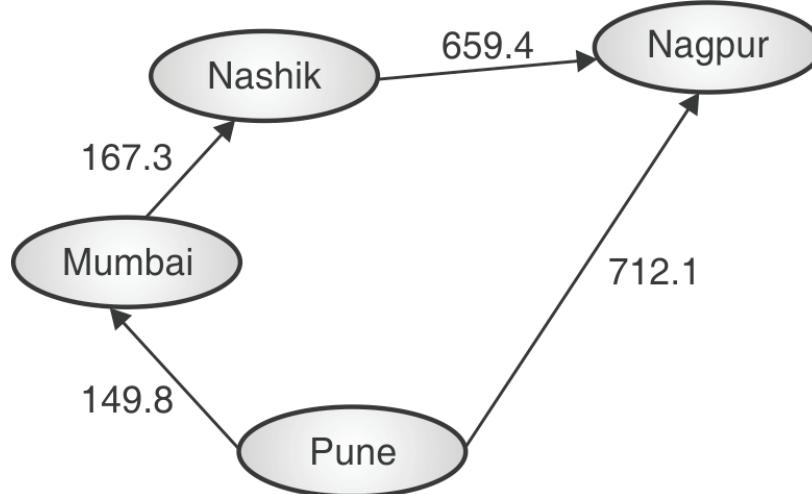
- A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.



Terminologies of Graph

Length

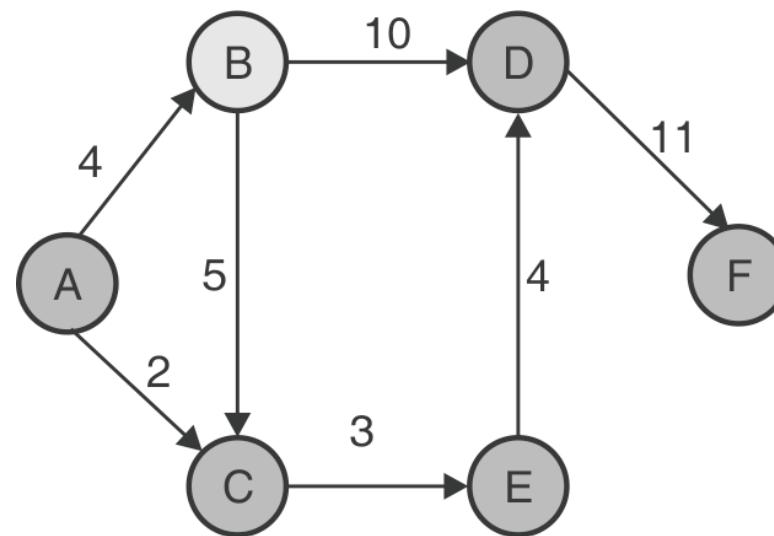
- Length of path is the total number of edges included in that path.
- The length of path (Pune-Nasik) is 2.



Terminologies of Graph

Linear Path

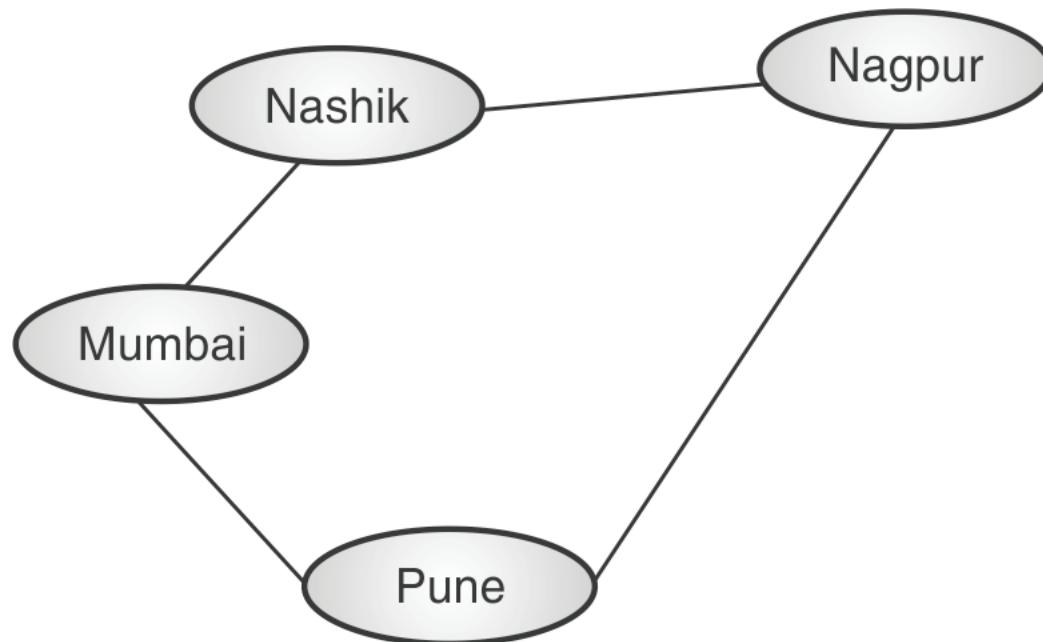
- The path in which the starting and ending vertices are different, is called as Linear Path.



Terminologies of Graph

Cycle

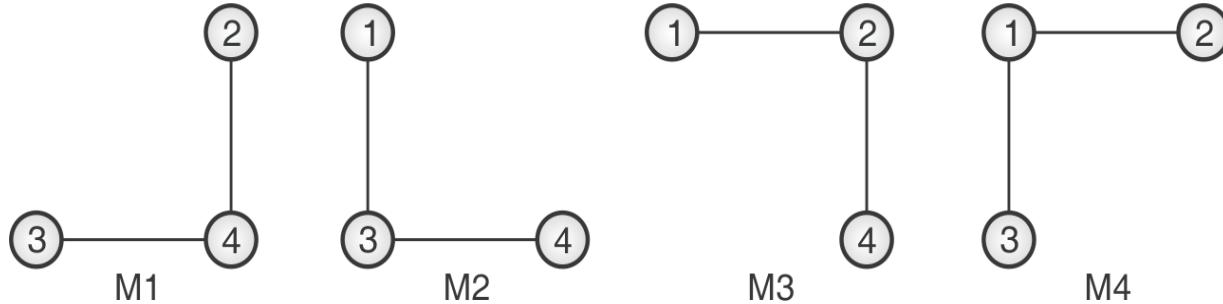
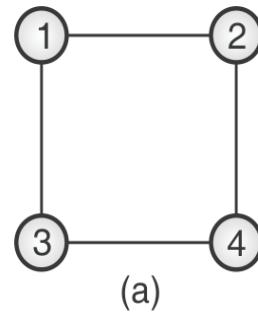
- The path, in which the starting and ending vertex is same, is called as Cycle.



Terminologies of Graph

Sub Graph

- A subgraph of a graph G is another graph formed from a subset of the vertices and edges of G.
- M1, M2, M3 and M4 are sub-graphs of Graph - (a)



Terminologies of Graph

Source

- A source, in a directed graph, is a vertex with no incoming edges (in-degree equals to 0).

Terminologies of Graph

Isolated Node

- An isolated vertex of a graph is a vertex having degree as zero, that is, a vertex with no incident edges.

Terminologies of Graph

Sink

- A sink, in a directed graph, is a vertex with no outgoing edges (out-degree equals 0).

Terminologies of Graph

Articulation Point

- A vertex in a connected graph removal of which would disconnect the graph, is called as Articulation Point.

GRAPH REPRESENTATION

- A graph can be represented in two ways :
 1. Sequential representation of graphs
 2. Linked representation of graphs

Sequential Representation of Graph :Adjacency Matrix

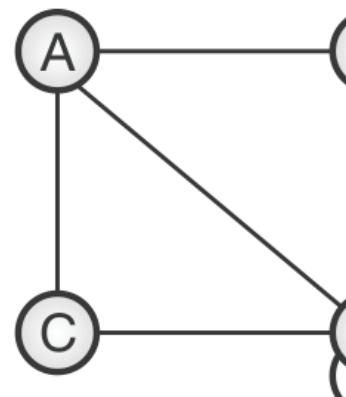
- This is a simple way to represent a graph. 2D array is used for this representation.
- In Adjacency Matrix Graph is represented using a matrix of size total number of vertices by total number of vertices.
- That means if a graph has 5 vertices, then a matrix of $5 * 5$ will be used.
- In this matrix, rows and columns both represent vertices.
- All the values of this matrix are either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

Sequential Representation of Graph :Adjacency Matrix

- Let the 2D array be arr[][], a slot $\text{arr}[i][j] = 1$ indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric : the row i, column j entry is 1 if and only if the row j , column i entry is 1.
- With an adjacency matrix, we can find out whether an edge is present in constant time, by just looking up the corresponding entry in the matrix.

Sequential Representation of Graph :Adjacency Matrix

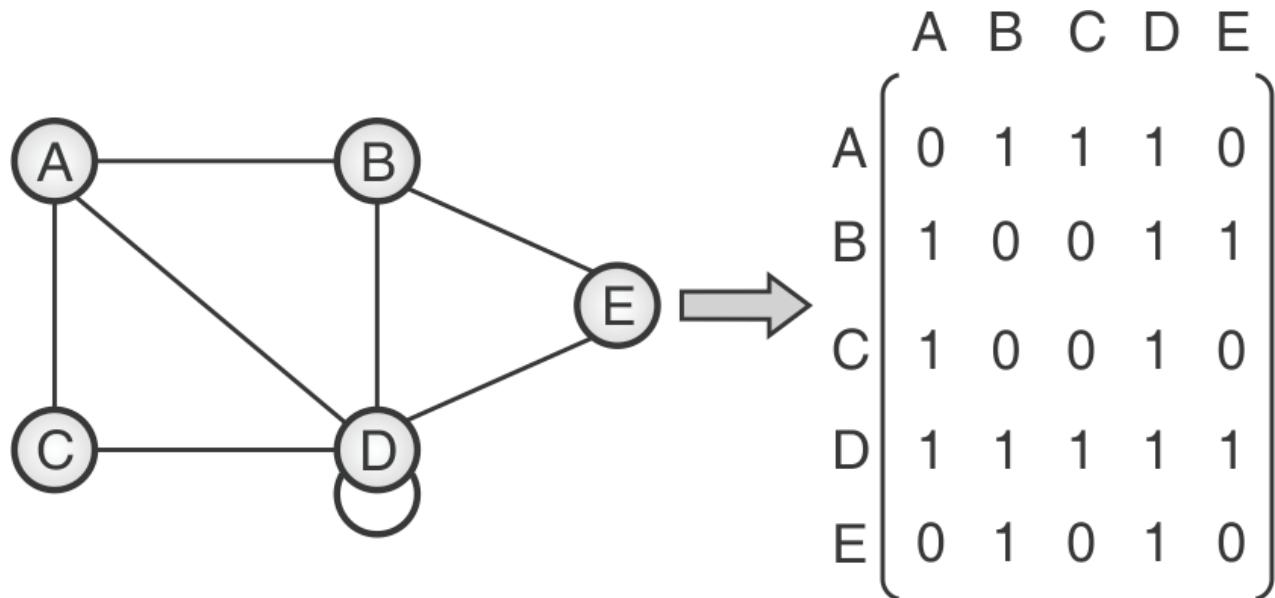
- For example, if the adjacency matrix is named graph, then we can query whether edge (i, j) is in the graph by looking at graph[i][j].
- In the adjacency matrix, the calculation of degree is very simple task.
- In undirected graph, the degree of any vertex is the number of 1s in the row of that vertex.



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Sequential Representation of Graph :Adjacency Matrix

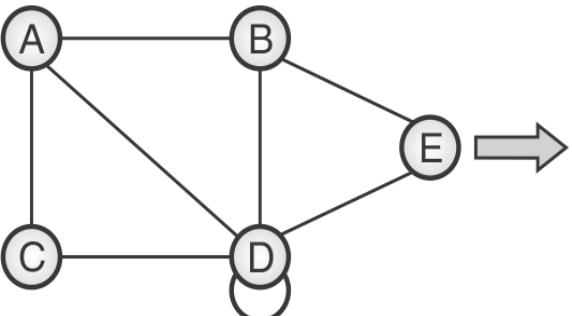
Degree of vertices in undirected graph



Vertex	Degree
A	3
B	3
C	2
D	5
E	2

Sequential Representation of Graph :Adjacency Matrix

Array representation



$$\begin{array}{c} \begin{matrix} & A & B & C & D & E \\ \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} & \left(\begin{matrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{matrix} \right) \end{matrix} \end{array}$$

Vertex	Degree
A	3
B	3
C	2
D	5
E	2

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

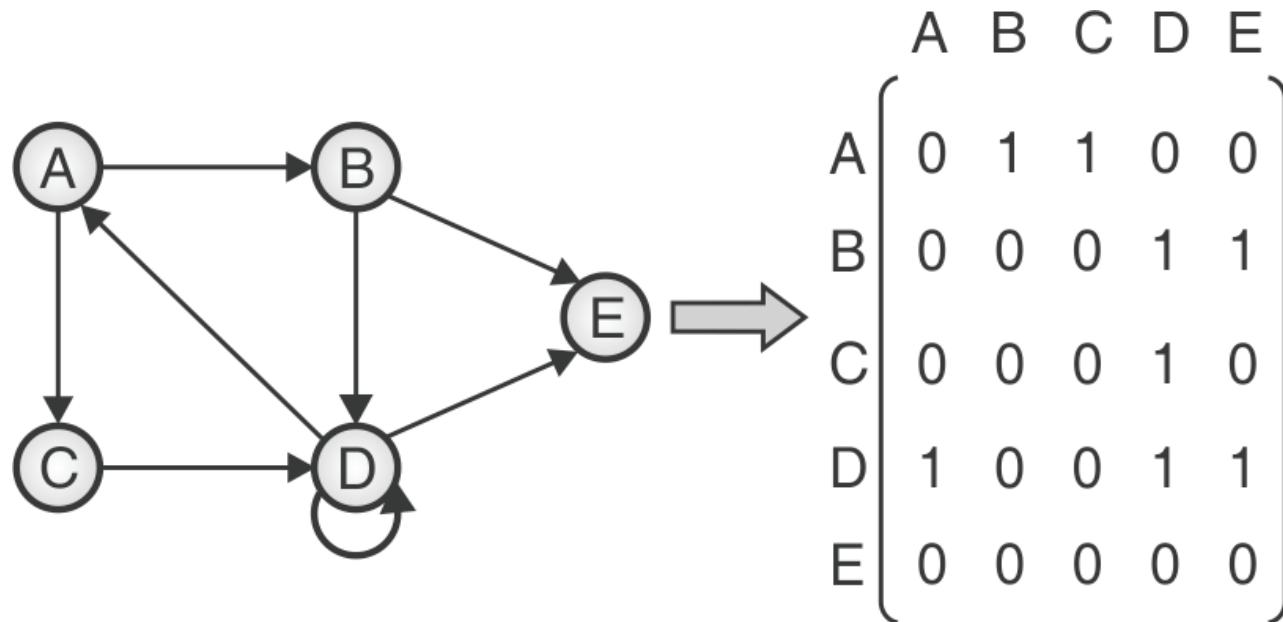
arr[i][j]

arr[0][0] = 1

arr[2][4] = 0

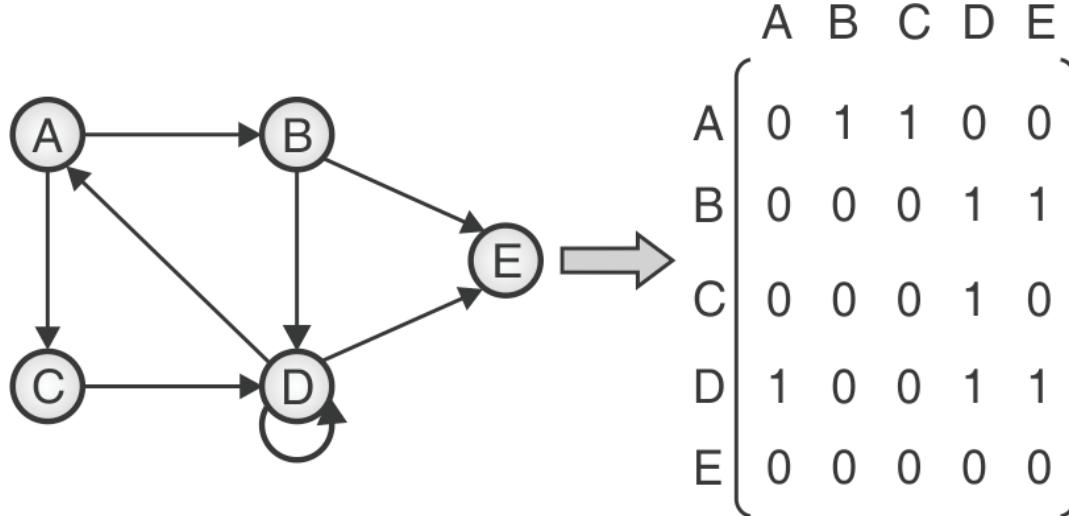
Sequential Representation of Graph :Adjacency Matrix

Adjacency Matrix of directed graph



Sequential Representation of Graph :Adjacency Matrix

Adjacency Matrix of directed graph



In directed graph, two types of degrees can be calculated : In-degree and Out-degree.

- **In-degree**

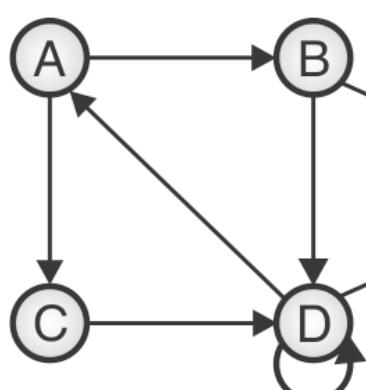
It is the total number of 1s in column of the vertex.

- **Out-degree**

It is the total number of 1s in row of the vertex.

Sequential Representation of Graph :Adjacency Matrix

Degree of vertices in directed graph



$$\begin{array}{ccccc} & \text{A} & \text{B} & \text{C} & \text{D} & \text{E} \\ \text{A} & 0 & 1 & 1 & 0 & 0 \\ \text{B} & 0 & 0 & 0 & 1 & 1 \\ \text{C} & 0 & 0 & 0 & 1 & 0 \\ \text{D} & 1 & 0 & 0 & 1 & 1 \\ \text{E} & 0 & 0 & 0 & 0 & 0 \end{array}$$

Vertex	In-degree	Out-degree
A	1	2
B	1	2
C	1	1
D	3	3
E	2	0

Sequential Representation of Graph :Adjacency Matrix

Pros of Adjacency Matrix

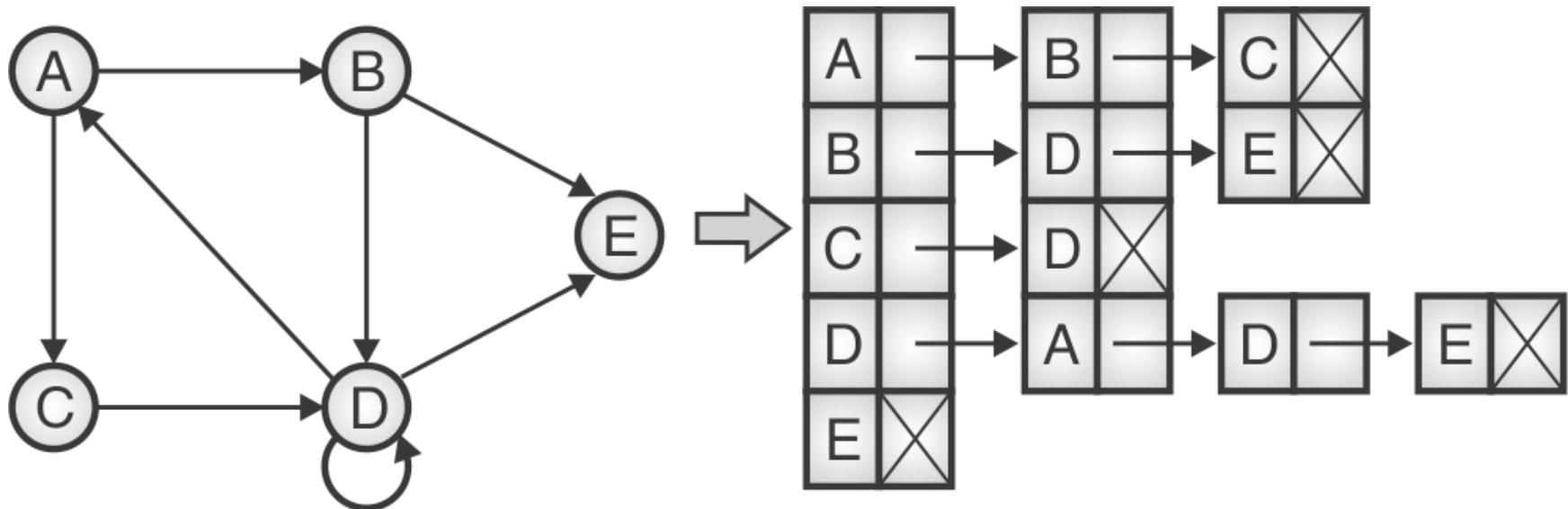
- Representation is easier to implement and follow.
- Removing an edge takes less time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient.

Cons of Adjacency Matrix

- Consumes more space. Even if the graph is sparse (contains less number of edges), it consumes the same space.

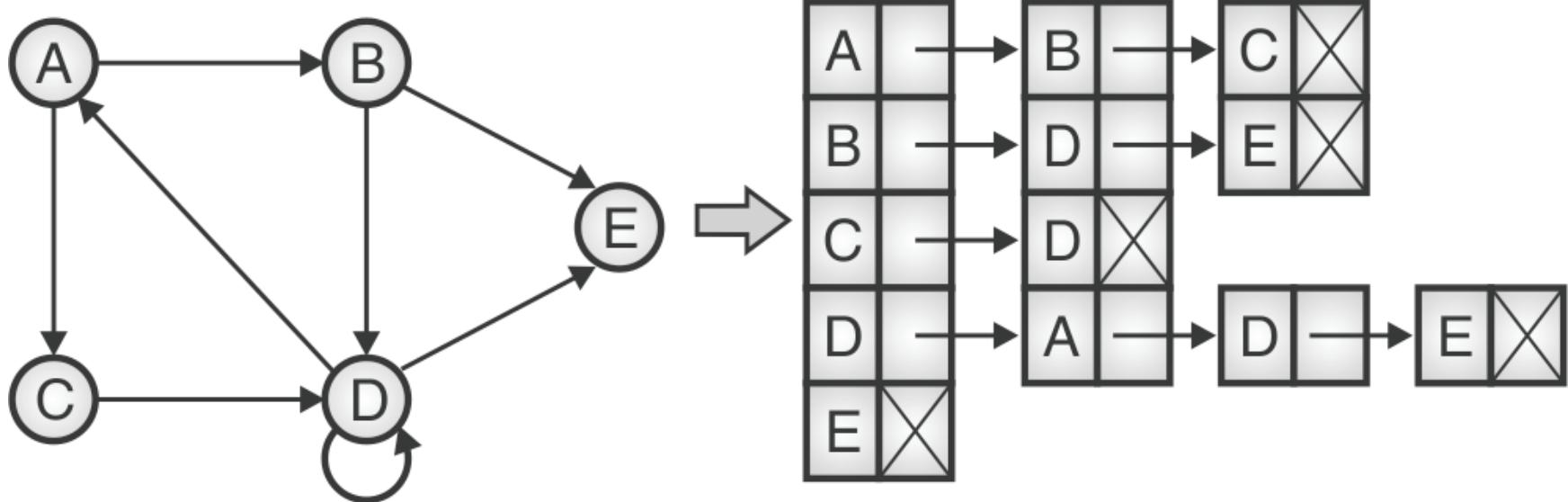
Linked Representation of Graph: Adjacency List

- This representation is an alternative to Adjacency Matrix. The memory required by this representation is very less.
- In this representation, every vertex of graph contains list of its adjacent vertices.
- directed graph representation implemented using linked list



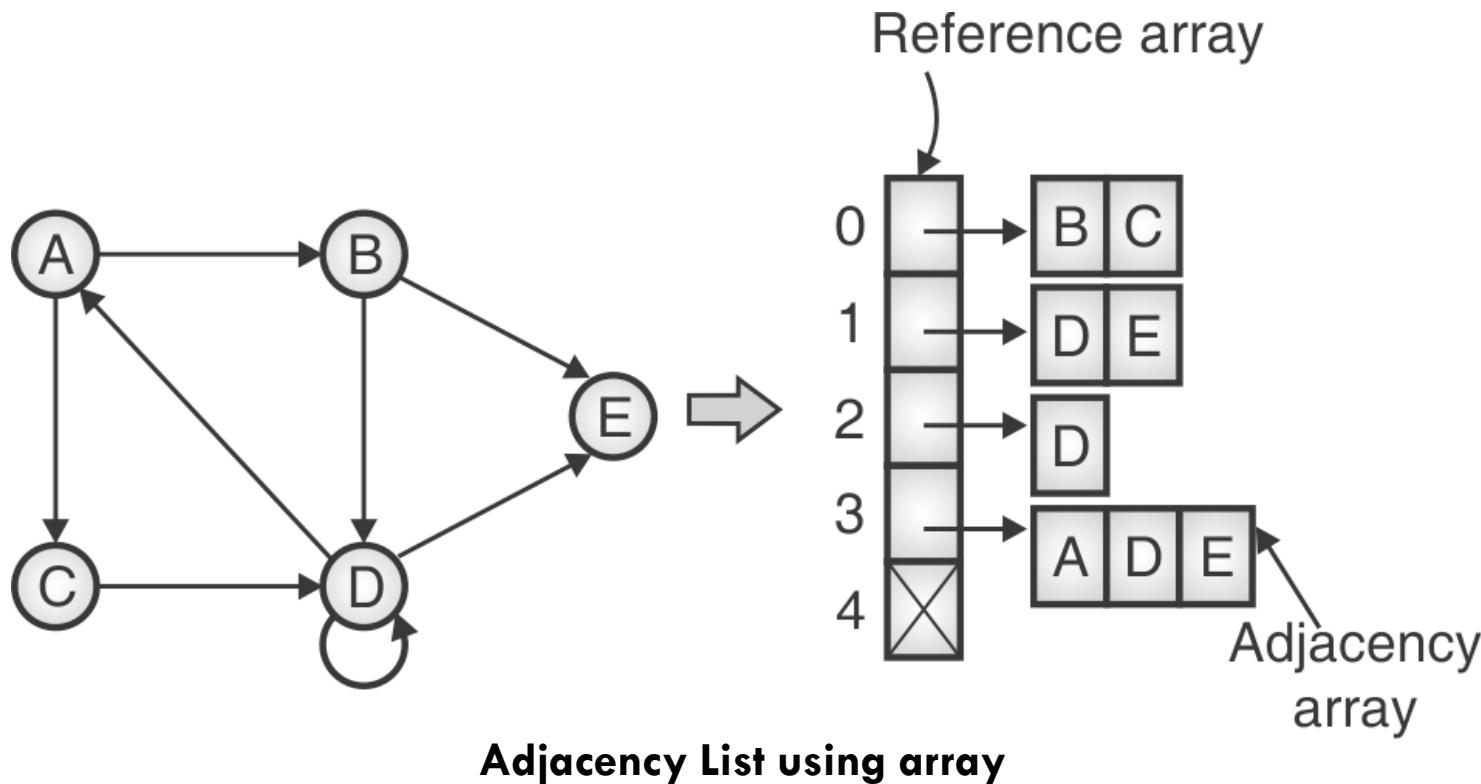
Linked Representation of Graph: Adjacency List

- In this representation an array of linked lists is generally used.
- Size of this array is exactly equal to number of vertices in the graph.
- Let the array be $\text{arr}[]$. An entry $\text{arr}[i]$ represents the linked list of vertices adjacent to the i th vertex.



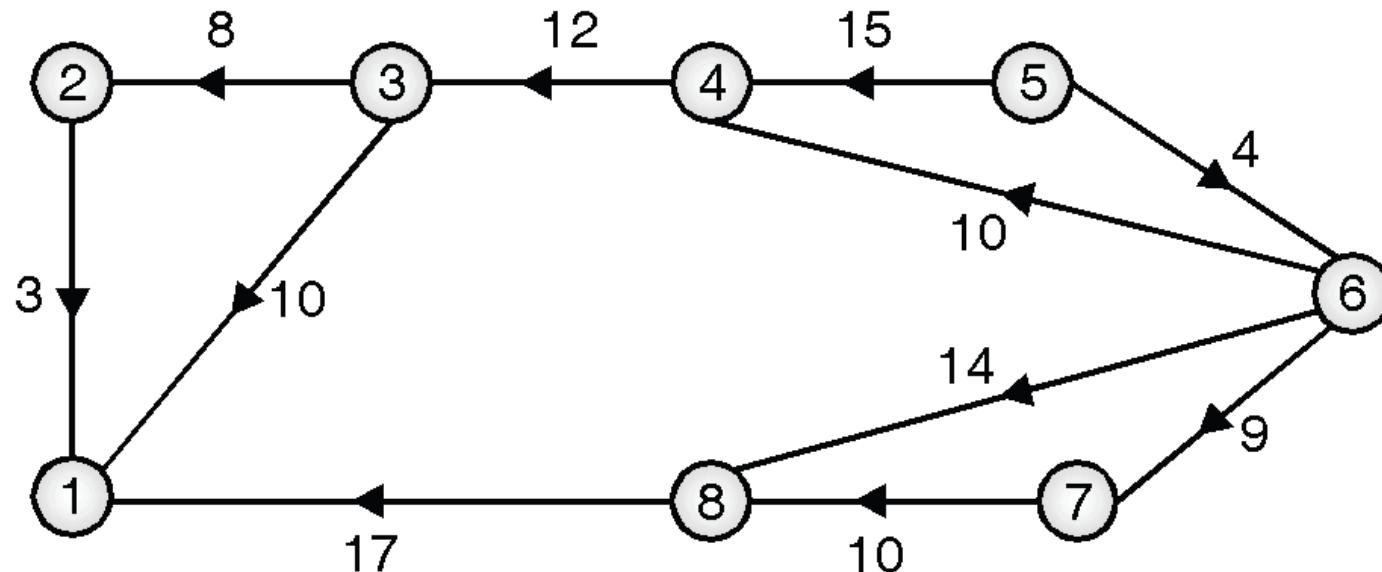
Linked Representation of Graph: Adjacency List

- This representation can also be used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.



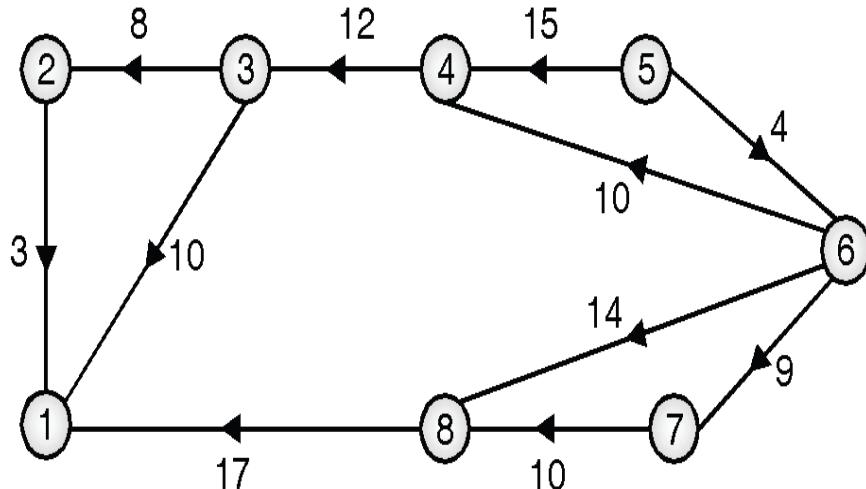
Example #1

- For the graph shown below:
 - (i) The in degree and out degree of each vertex,
 - (ii) Its adjacency matrix
 - (iii) Its adjacency list representation.



Example #1

- For the graph shown below:
 - The in degree and out degree of each vertex,

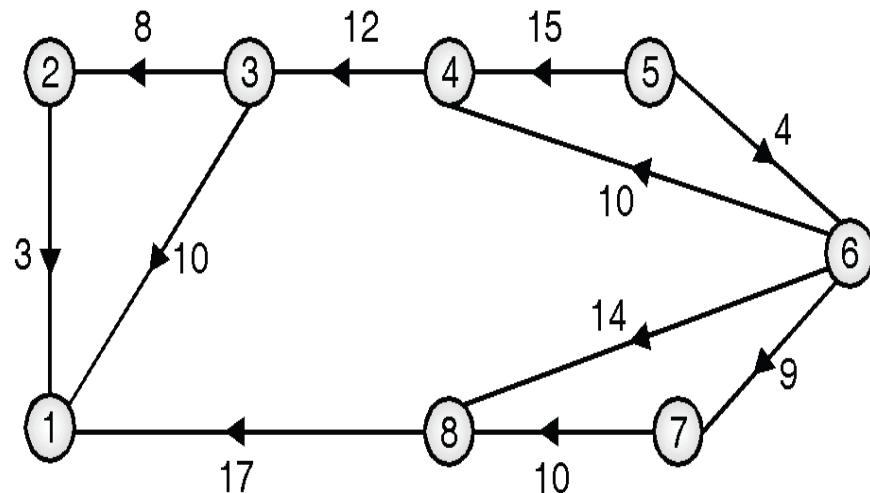


Vertex No.	Indegree	Outdegree
1	3	0
2	1	1
3	1	2
4	2	1
5	1	1

Vertex No.	Indegree	Outdegree
6	1	3
7	1	1
8	2	1

Example #1

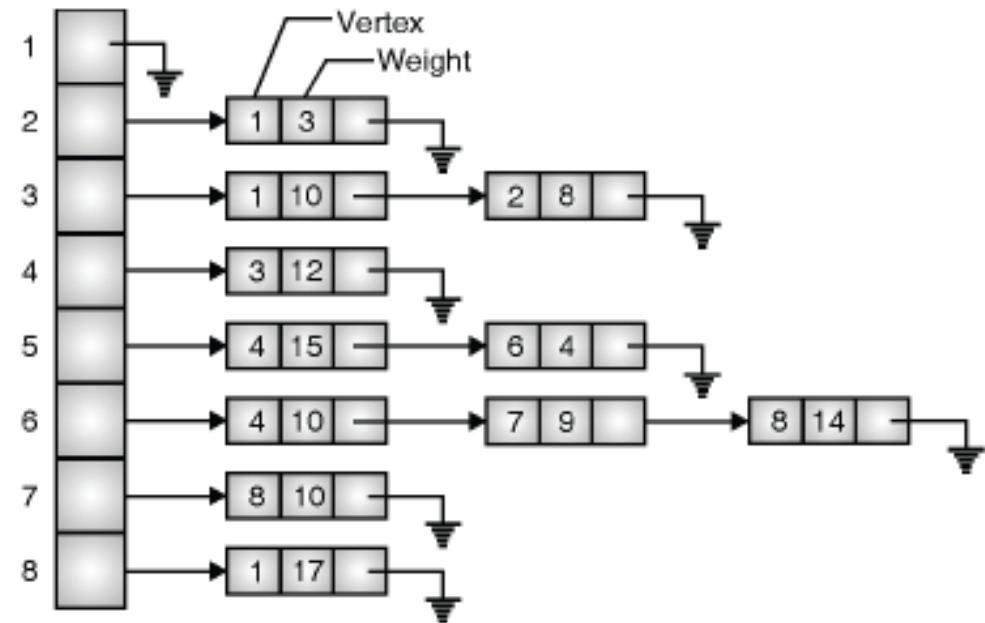
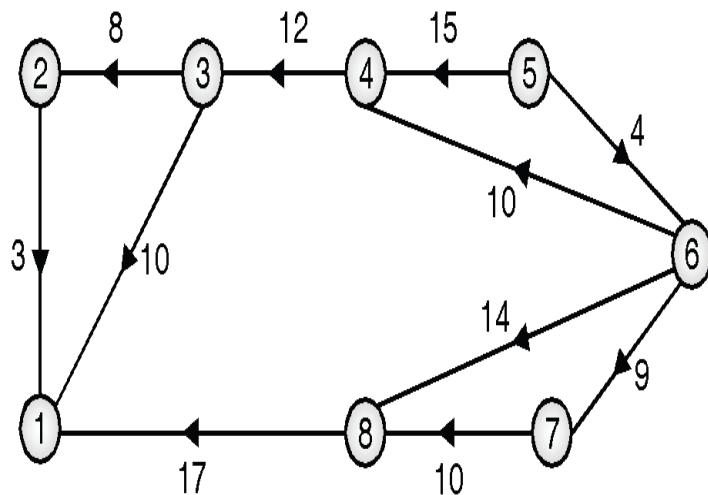
- For the graph shown below:
 - (ii) Its adjacency matrix



	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0
3	10	8	0	0	0	0	0	0
4	0	0	12	0	0	0	0	0
5	0	0	0	15	0	4	0	0
6	0	0	0	10	0	0	9	14
7	0	0	0	0	0	0	0	10
8	17	0	0	0	0	0	0	0

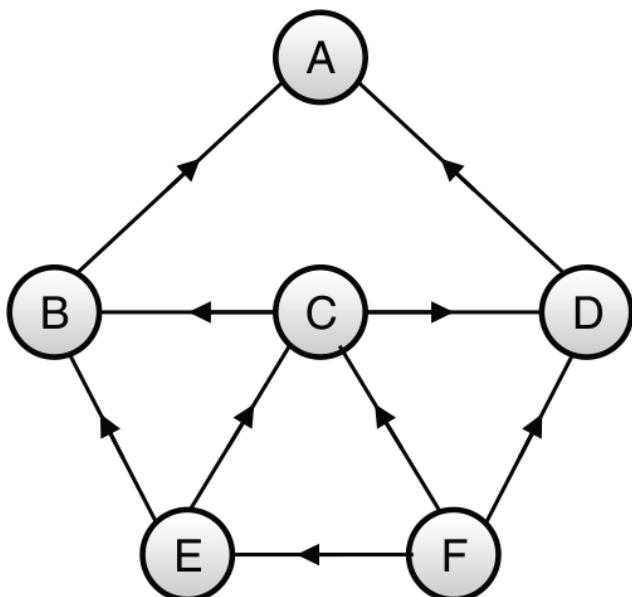
Example #1

- For the graph shown below:
 - (iii) Its adjacency list representation.



Example #2

- Consider the graph shown in Fig :
 - (i) Give adjacency matrix representation.
 - (ii) Give adjacency list representation of graph..

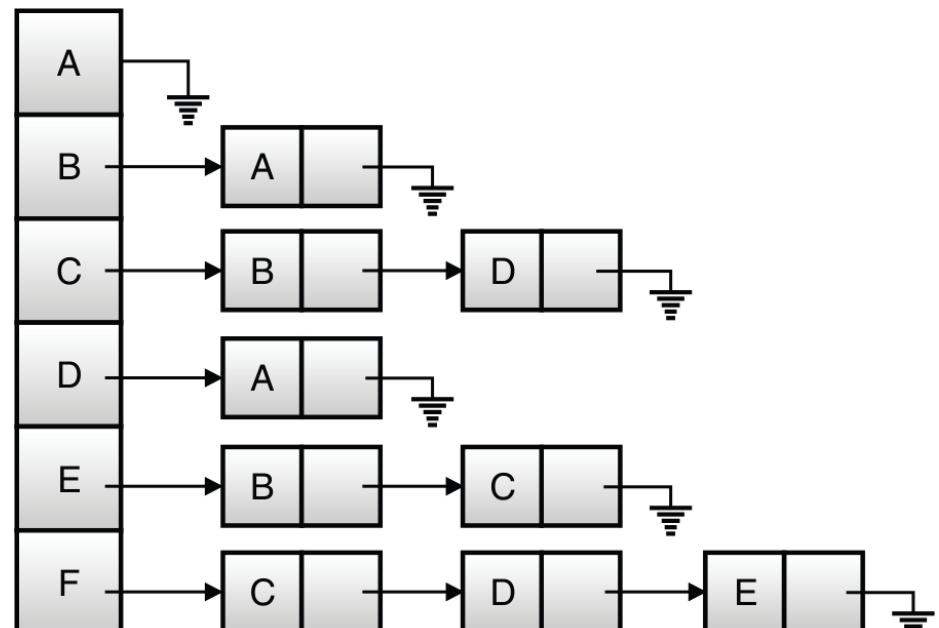
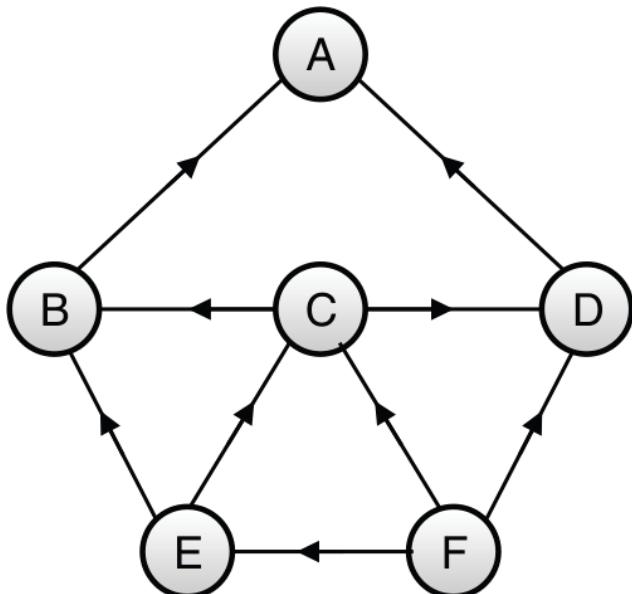


	A	B	C	D	E	F
A	0	0	0	0	0	0
B	1	0	0	0	0	0
C	0	1	0	1	0	0
D	1	0	0	0	0	0
E	0	1	1	0	0	0
F	0	0	1	1	1	0

Adjacency matrix

Example #2

- Consider the graph shown in Fig :
 - (i) Give adjacency matrix representation.
 - (ii) Give adjacency list representation of graph..



Adjacency list

TRAVERSAL : DEPTH FIRST AND BREADTH FIRST

- Graph traversal refers to the process of visiting each vertex in a graph at least once.
- There are various reasons for which we may have to traverse a graph :
 1. Finding all reachable nodes.
 2. Finding the best reachable.
 3. Finding the best path through a graph.
 4. Topologically sorting a graph.

TRAVERSAL : DEPTH FIRST AND BREADTH FIRST

- Graph traversal is classified by the order in which the vertices are visited.
- There are two most frequently used graph traversal methods :

Graph Traversal Methods

1. Depth First Search (DFS)

2. Breadth First Search (BFS)

DEPTH FIRST SEARCH (DFS)

- A depth-first search (DFS) is an algorithm for traversing a finite graph.
- DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth.
- The algorithm begins with a chosen "root" vertex; it then iteratively transitions from the current vertex to an adjacent, unvisited vertex, until it can no longer find an unexplored vertex to transition to from its current location.

DEPTH FIRST SEARCH (DFS)

- The algorithm then backtracks along previously visited vertices, until it finds a vertex connected to..
- It will then proceed down the new path as it had before, backtracking as it encounters dead-ends, and ending only when the algorithm has backtracked past the original "root" vertex from the very first step.

Algorithm of Non-Recursive DFS

In non- recursive traversal, we use concept of stack to hold some elements.

- **Step 1 :** Initially define a stack with size same as total number of vertices in the graph.
- **Step 2 :** Select the root vertex as starting node for traversal. Visit that vertex and push it in the Stack.
- **Step 3 :** Visit any nearest (adjacent) vertex of the vertex which is located at top of the stack and which is not visited and push it in the stack.
- **Step 4 :** Repeat step 3 until there are no any adjacent vertex to visit from the vertex on top of the stack

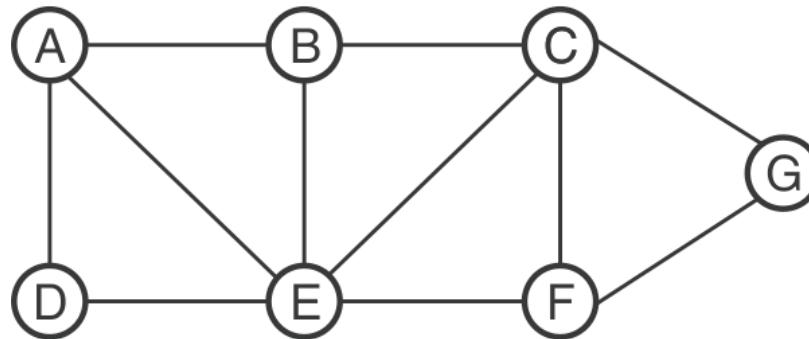
Algorithm of Non-Recursive DFS

In non- recursive traversal, we use concept of stack to hold some elements.

- **Step 5 :** When there is no any vertex remain to visit then use the method of back tracking and pop one vertex from the stack.
- **Step 6 :** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 :** When stack becomes completely empty, then generate final spanning tree by removing unused edges from the graph.

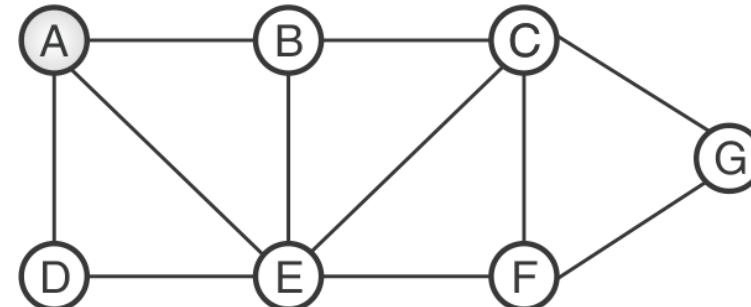
DFS: Example

Consider following graph to perform DFS traversal.



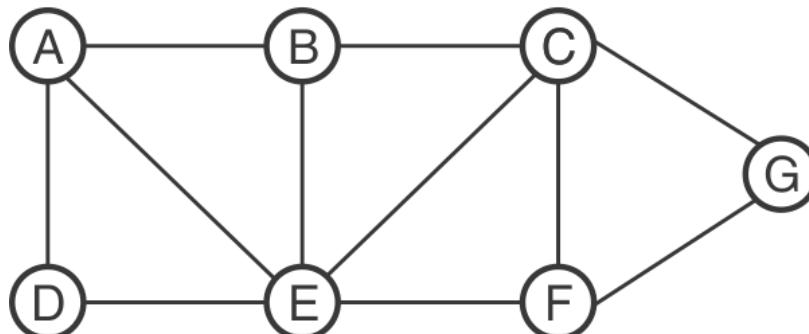
Step I:

- Select the vertex A as starting point (visit A)
- Push A in the Stack.



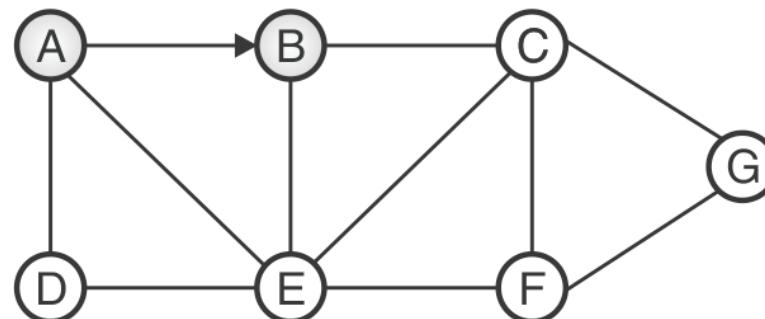
DFS: Example

Consider following graph to perform DFS traversal.



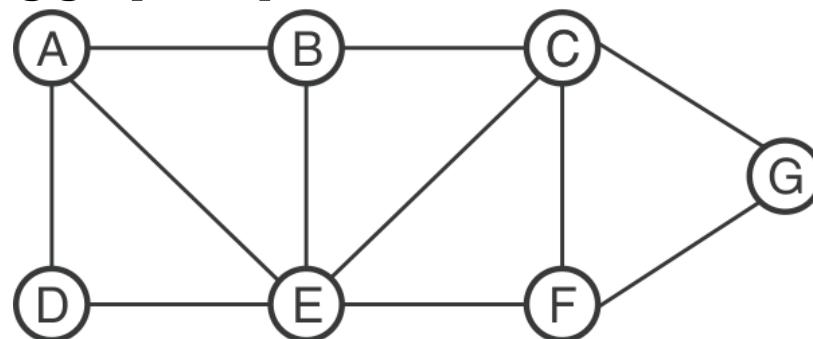
Step II:

- Visit any adjacent unvisited vertex of A. Here we visit B.
- Push B in the Stack.



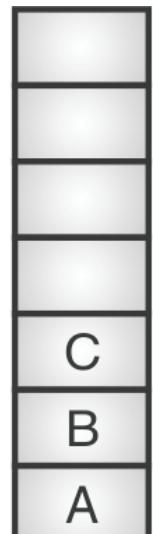
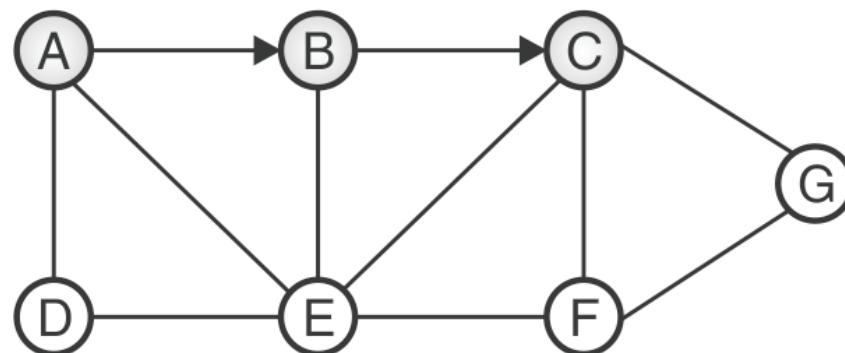
DFS: Example

Consider following graph to perform DFS traversal.



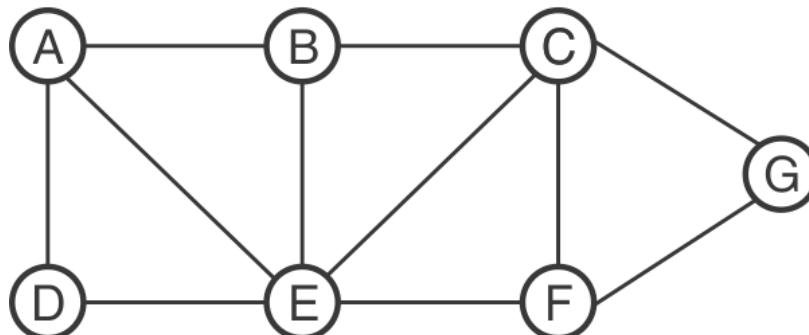
Step III:

- Visit any adjacent unvisited vertex of B. Here we visit C.
- Push C in the Stack.



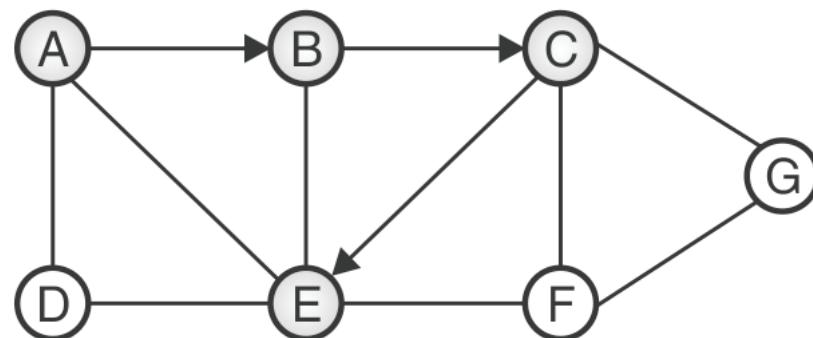
DFS: Example

Consider following graph to perform DFS traversal.



Step IV:

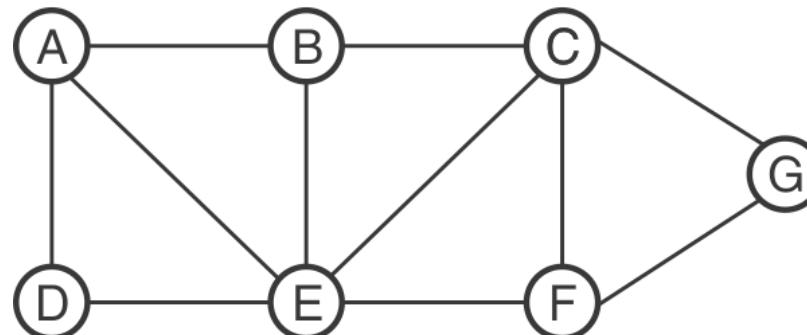
- Visit any adjacent unvisited vertex of C. Here we visit E.
- Push E in the Stack.



Stack

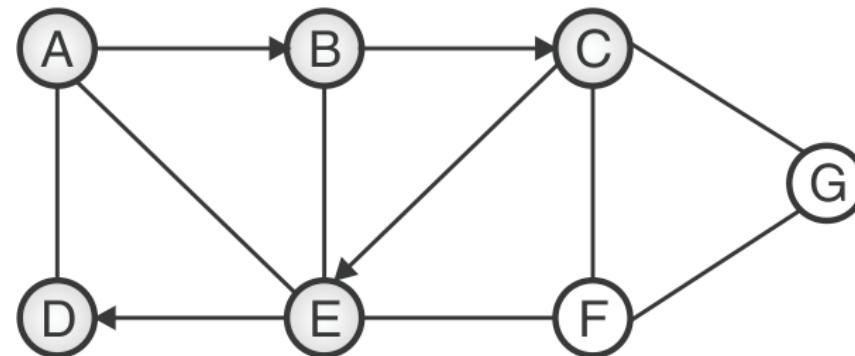
DFS: Example

Consider following graph to perform DFS traversal.



Step V:

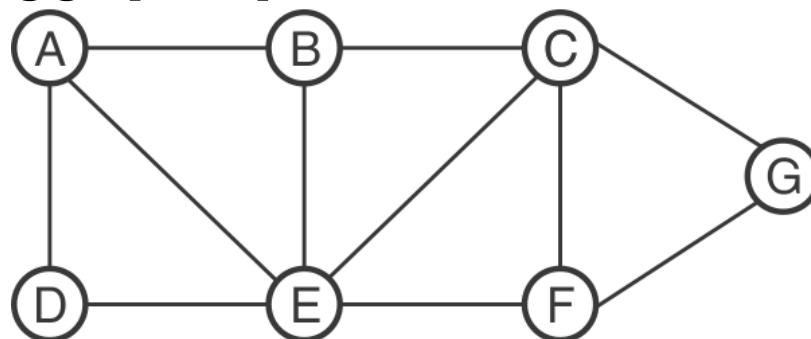
- Visit any adjacent unvisited vertex of E. Here we visit D.
- Push D in the Stack.



Stack

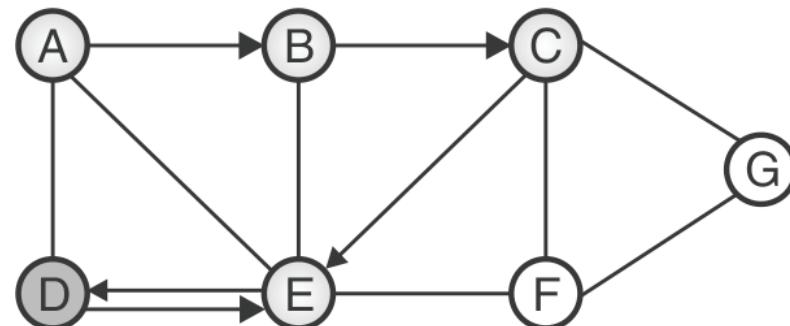
DFS: Example

Consider following graph to perform DFS traversal.



Step VI:

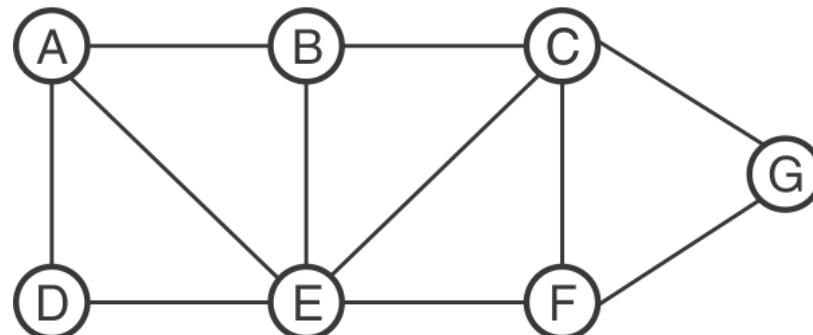
- From D, there is no any unvisited vertex. Hence use Backtrack.
- Pop D from Stack..



Stack

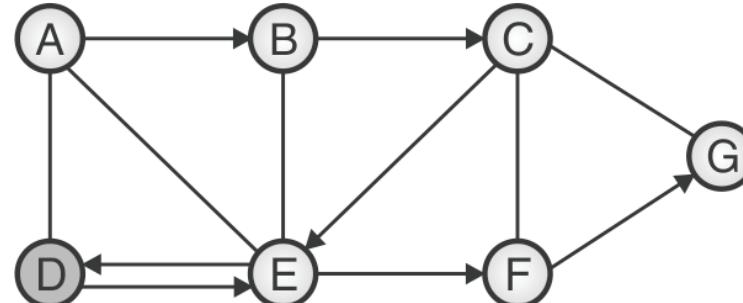
DFS: Example

Consider following graph to perform DFS traversal.



Step VII:

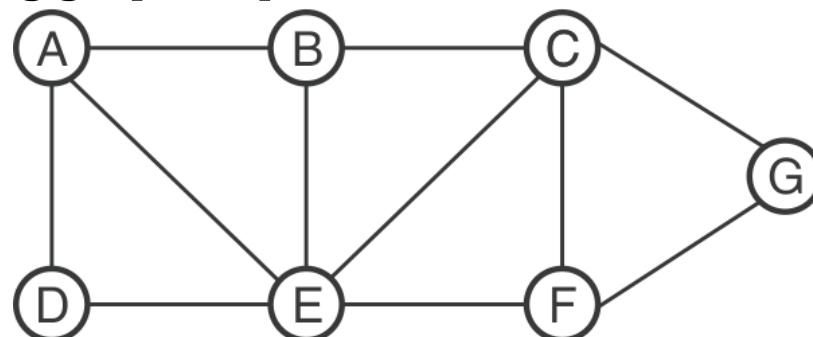
- Visit any adjacent unvisited vertex of E. Here we visit F.
- Push F in the Stack.



Stack

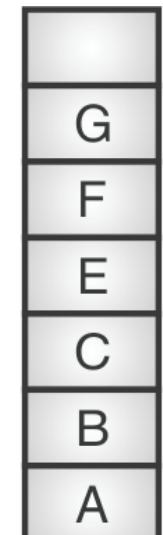
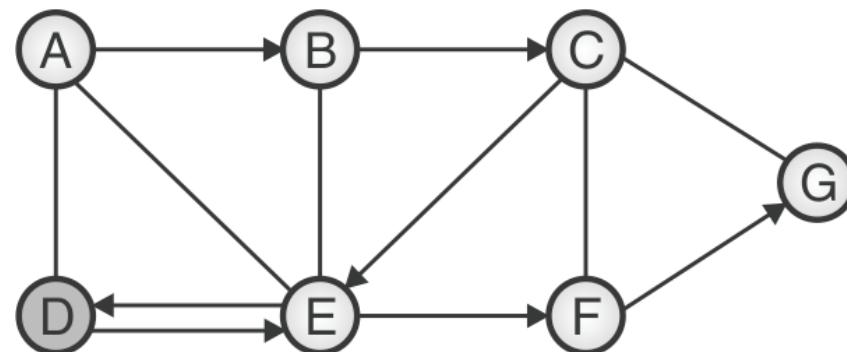
DFS: Example

Consider following graph to perform DFS traversal.



Step VIII:

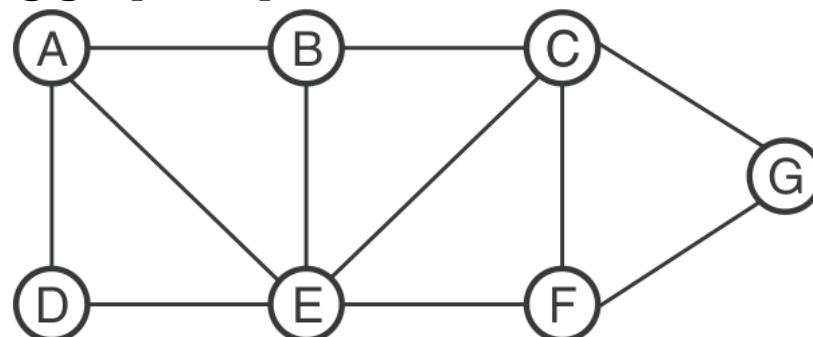
- Visit any adjacent unvisited vertex of F. Here we visit G.
- Push G in the Stack.



Stack

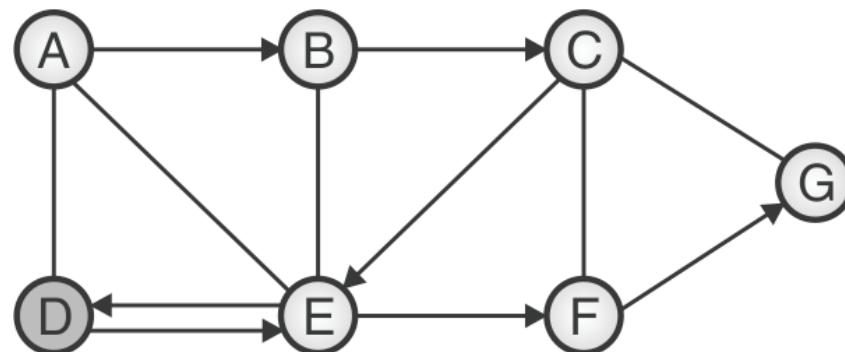
DFS: Example

Consider following graph to perform DFS traversal.



Step VIII:

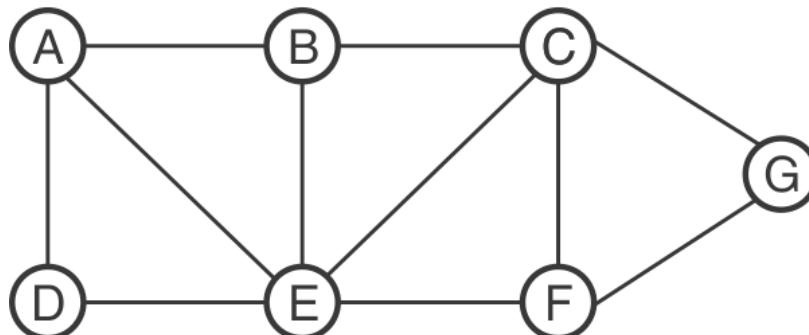
- Visit any adjacent unvisited vertex of F. Here we visit G.
- Push G in the Stack.



Stack

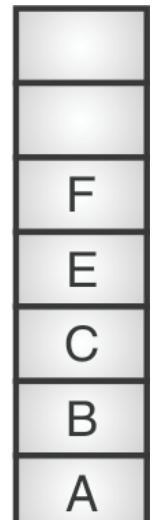
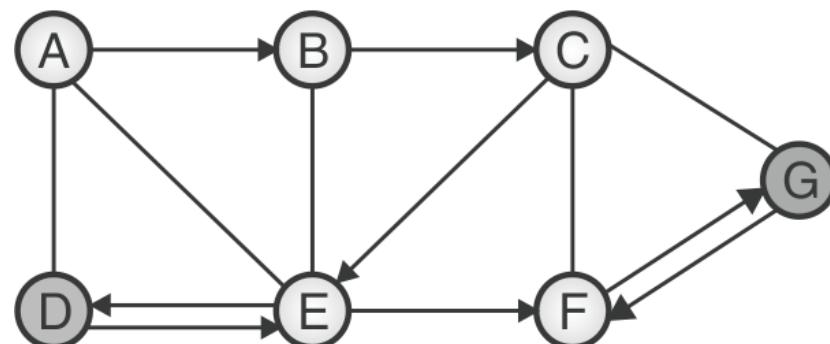
DFS: Example

Consider following graph to perform DFS traversal.



Step IX:

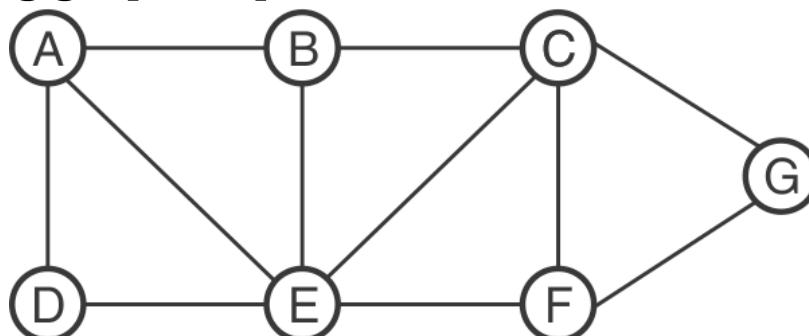
- From G, there is no any unvisited vertex. Hence use Backtrack.
- Pop G from Stack



Stack

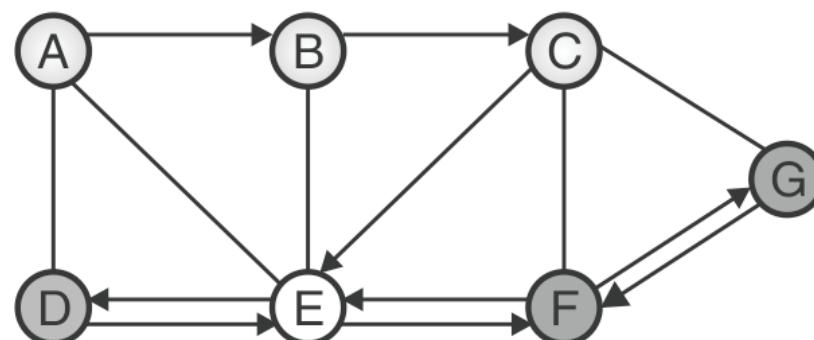
DFS: Example

Consider following graph to perform DFS traversal.



Step X:

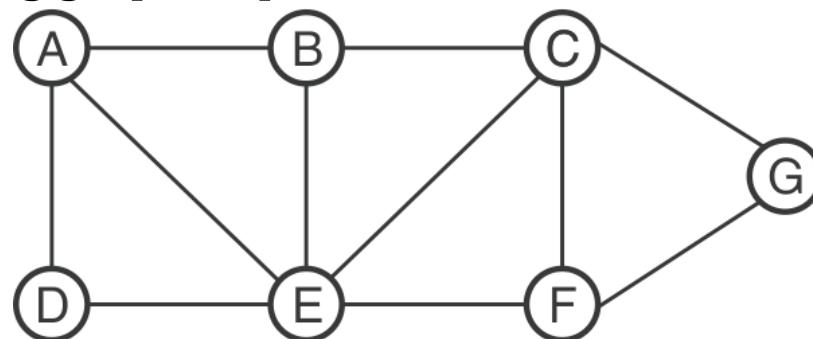
- From F, there is no any unvisited vertex. Hence use Backtrack.
- Pop F from Stack.



Stack

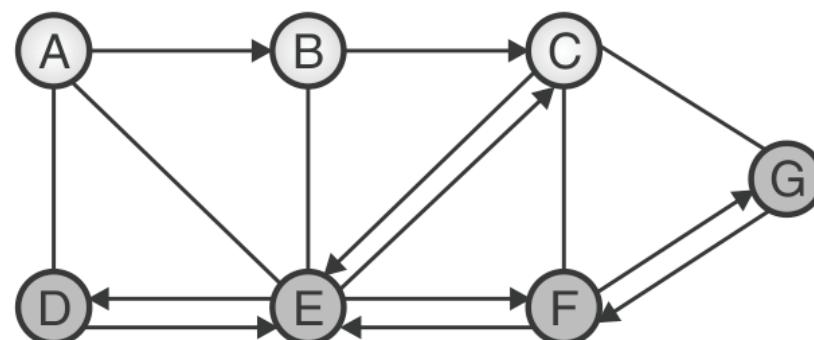
DFS: Example

Consider following graph to perform DFS traversal.



Step XI:

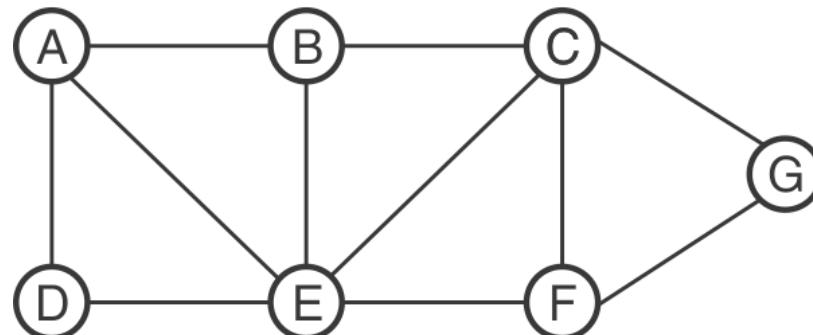
- From E, there is no any unvisited vertex. Hence use Backtrack.
- Pop E from Stack.



Stack

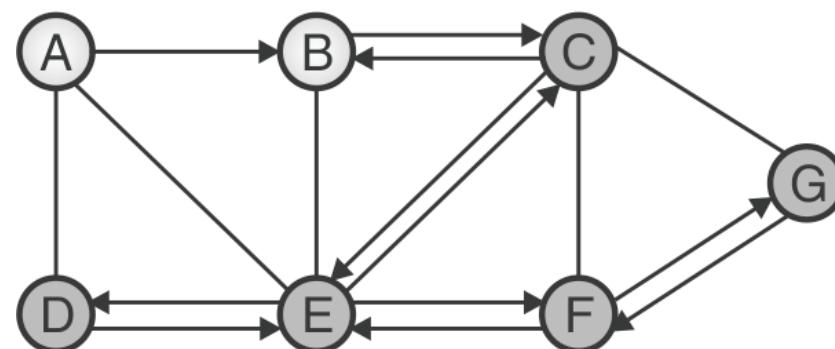
DFS: Example

Consider following graph to perform DFS traversal.



Step XII:

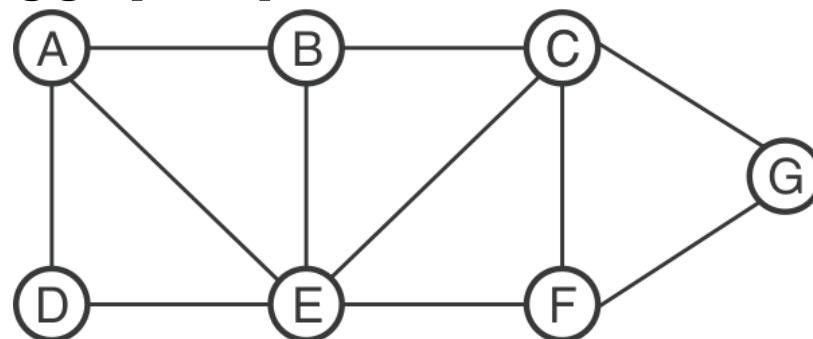
- From C, there is no any unvisited vertex. Hence use Backtrack.
- Pop C from Stack.



Stack

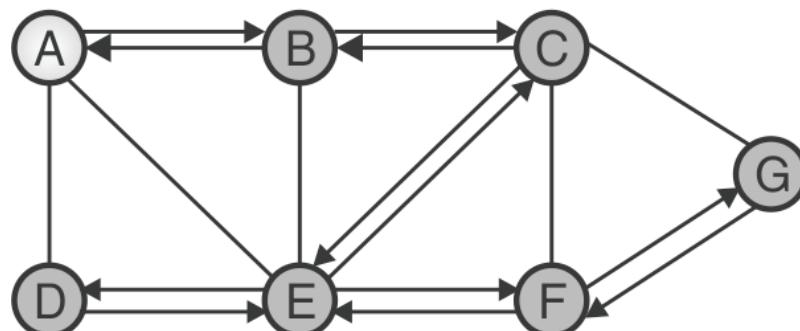
DFS: Example

Consider following graph to perform DFS traversal.



Step XIII:

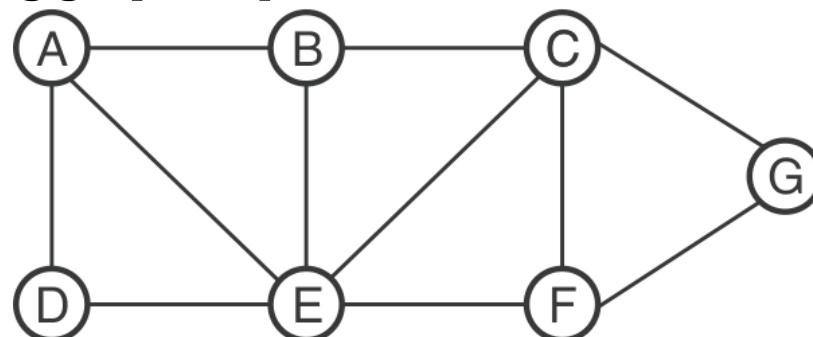
- From B, there is no any unvisited vertex. Hence use Backtrack.
- Pop B from Stack.



Stack

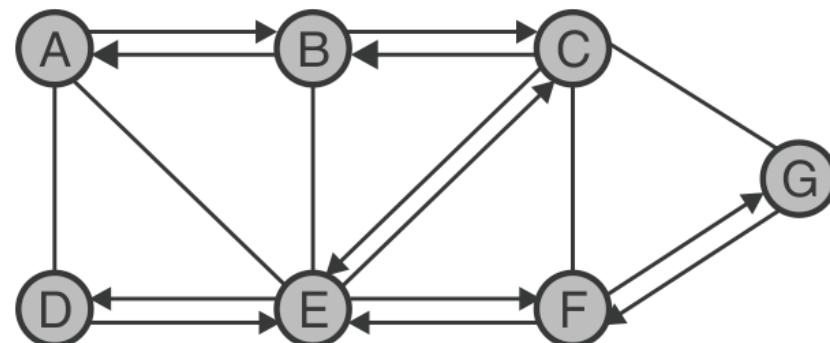
DFS: Example

Consider following graph to perform DFS traversal.



Step XIV:

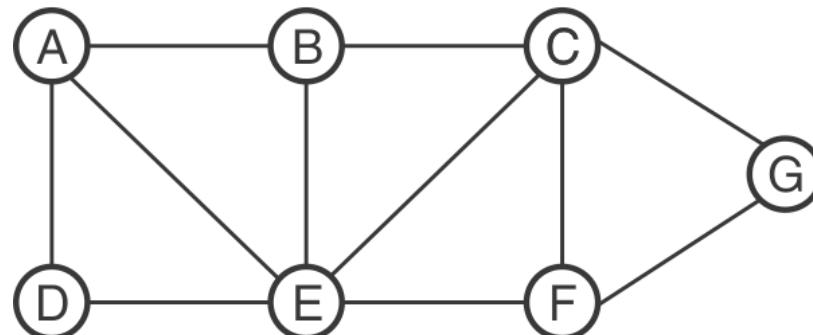
- From A, there is no any unvisited vertex. Hence use Backtrack.
- Pop A from Stack.



Stack

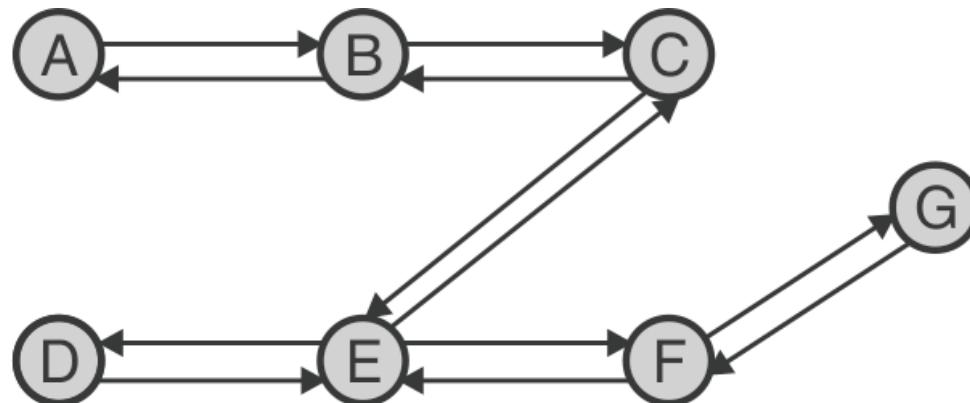
DFS: Example

Consider following graph to perform DFS traversal.



Step XIV:

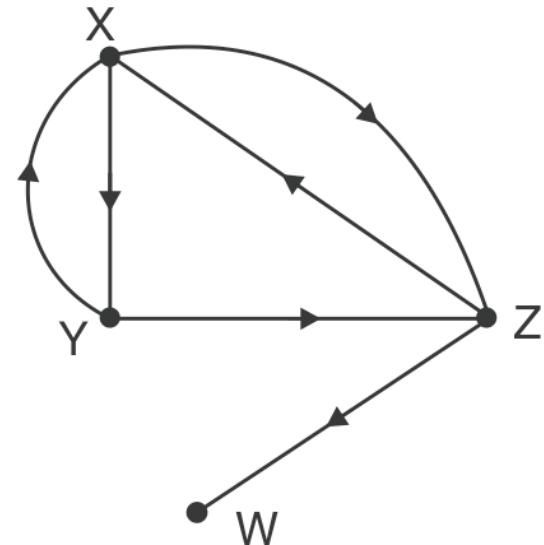
- Now Stack becomes empty, hence stop traversal.
- Final result of DFS is spanning tree :



The output of DFS traversal is : A, B, C, E, D, F, G

DFS: Example #2

- Consider the graph G.
 - (i) Write Adjacency matrix representation.
 - (ii) Depth first traversal sequence.
 - (iii) Find all simple path from X to W.
 - (iv) Find indegree (X) and outdegree (W).



Adjacency matrix representation

	W	X	Y	Z
W	0	0	0	0
X	0	0	1	1
Y	0	1	0	1
Z	1	1	0	0

DFS: Example #2

- Consider the graph G.
 - (i) Write Adjacency matrix representation.
 - (ii) Depth first traversal sequence.
 - (iii) Find all simple path from X to W.
 - (iv) Find indegree (X) and outdegree (W).

(ii) Depth first traversal sequence

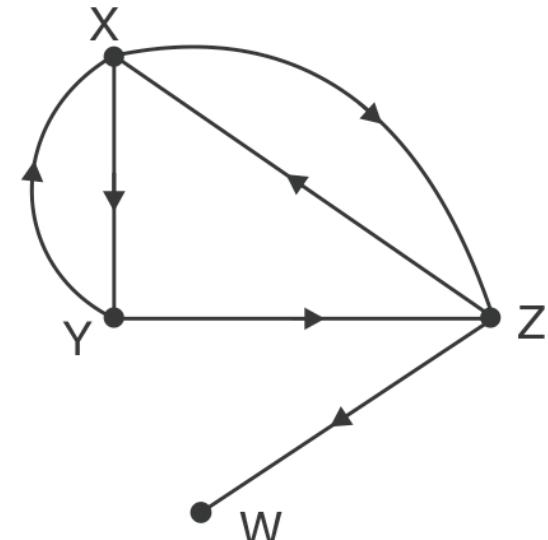
X, Y, Z, W

(iii) Find all simple path from X to W.

- (a) X → Z → W (b) X → Y → Z → W

(iv) Find indegree (X) and outdegree (W).

- indegree (X) - 2
- outdegree (W) - 0



	W	X	Y	Z
W	0	0	0	0
X	0	0	1	1
Y	0	1	0	1
Z	1	1	0	0

Advantages: DFS

1. DFS consumes very less memory space.
2. It will reach at the goal node in a less time than BFS if it traverses in a right path.
3. It may find a solution without examining much of search because we may get the desired solution in the very first go.

Disadvantages :DFS

1. It is possible that states may keep reoccurring. There is no guarantee of finding the expected node.
2. Sometimes the states may also enter into infinite loops.

BREADTH FIRST SEARCH (BFS)

- A breadth-first search (BFS) is another technique for traversing a finite graph.
- BFS visits the neighbour vertices before visiting the child vertices, and a queue is used in the search process.
- This algorithm is often used to find the shortest path from one vertex to another.
- BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without any loops.
- We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

Algorithm of BFS Traversal of Graph

We use the following steps to implement BFS Traversal:

Step 1 : Define a Queue of size equal to total number of vertices in the graph.

Step 2 : Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3 : Visit all the nearest (adjacent) vertices of the vertex which is at front of the Queue which is not visited and insert these vertices in the Queue.

Algorithm of BFS Traversal of Graph

We use the following steps to implement BFS Traversal:

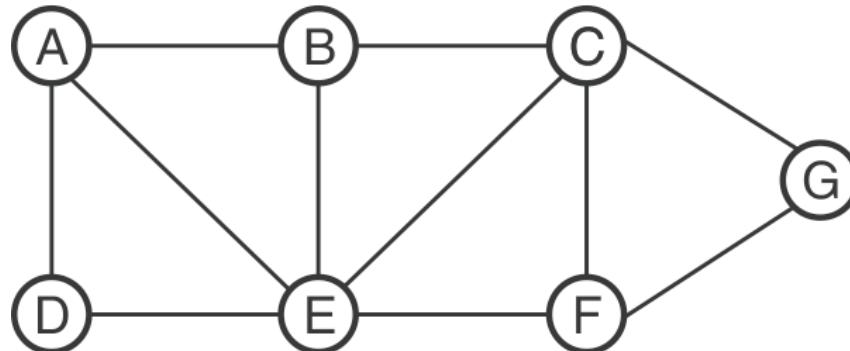
Step 4 : When there is no any unvisited vertex remaining from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5 : Repeat step 3 and 4 until queue becomes completely empty.

Step 6 : When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph.

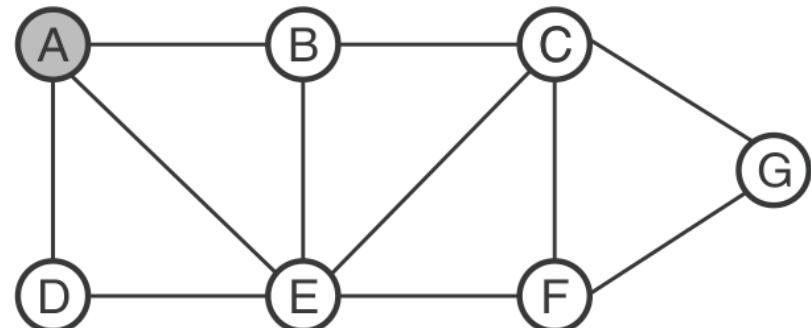
BFS: Example

Consider following graph.



Step I:

- Select the vertex A as starting vertex.
- Insert A into Queue

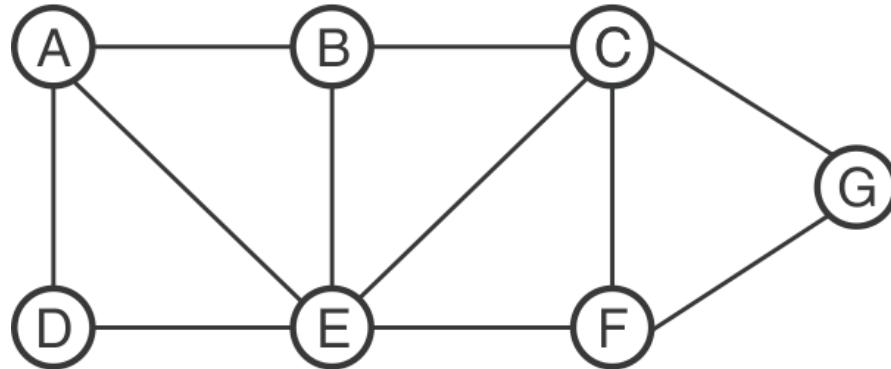


Queue



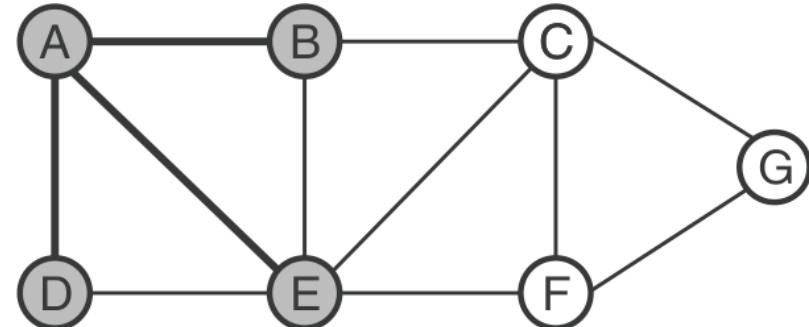
BFS: Example

Consider following graph.



Step II:

- Visit all the adjacent vertices of Vertex A which are unvisited (D, E, B).
- Insert the vertices D, E, B into Queue and remove vertex A from the queue.

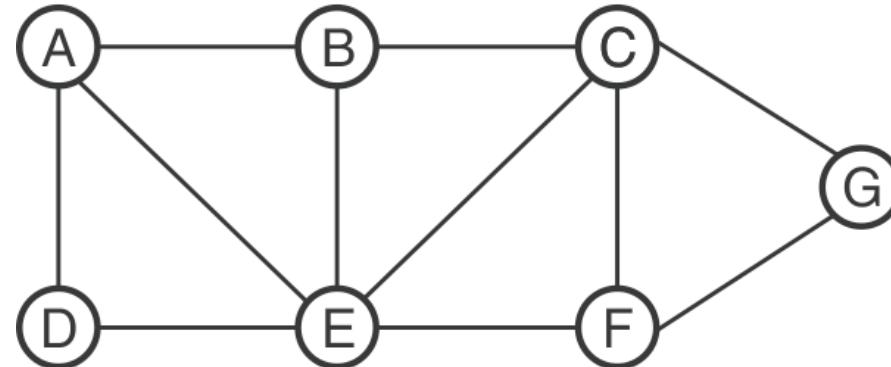


Queue



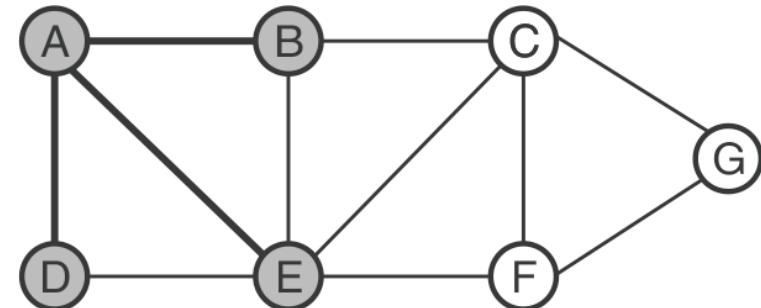
BFS: Example

Consider following graph.



Step III:

- Visit all the adjacent vertices of Vertex D which are unvisited (No any vertex).
- Remove D from queue.

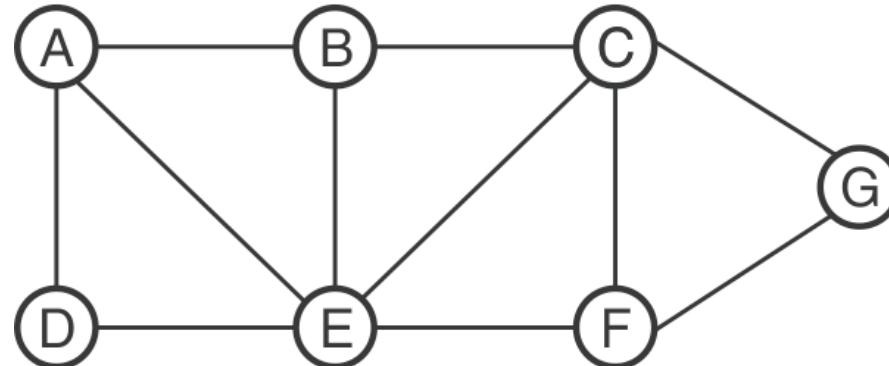


Queue



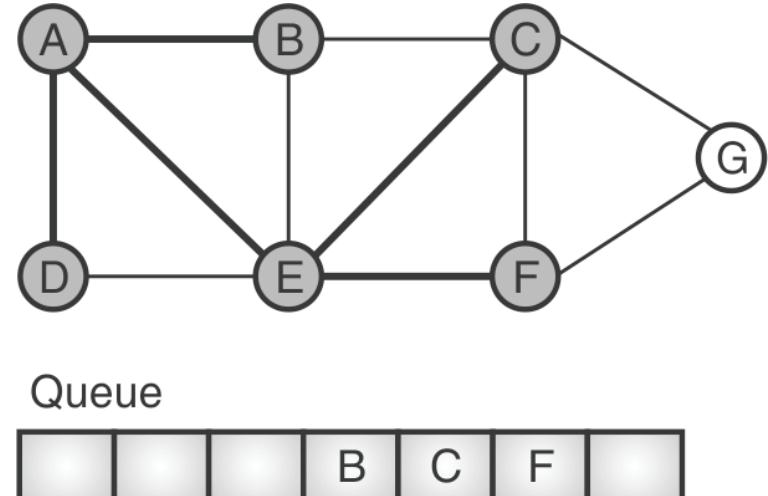
BFS: Example

Consider following graph.



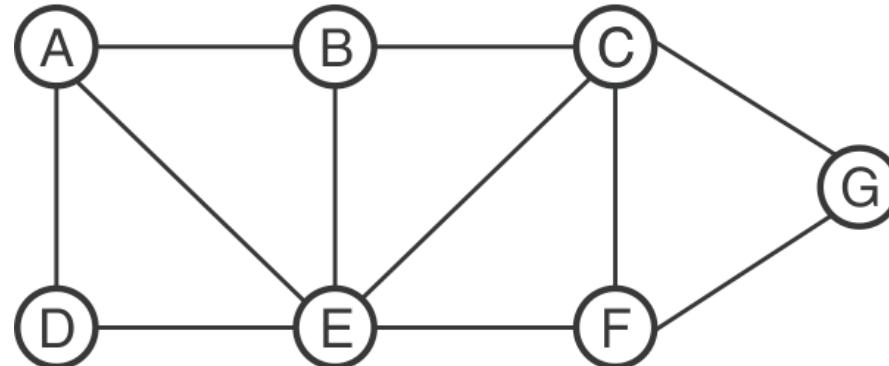
Step IV:

- Visit all the adjacent vertices of Vertex E which are unvisited (C, F).
- Insert the vertices C and F into Queue and remove vertex E from the queue.



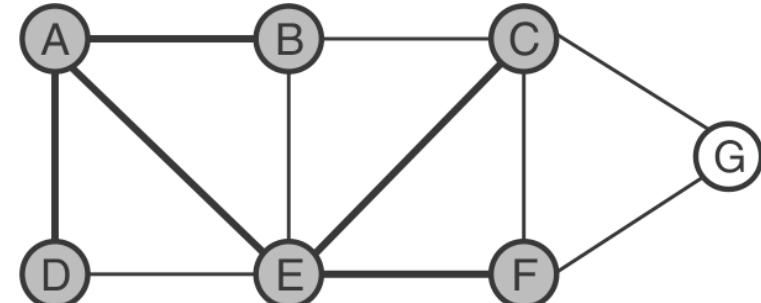
BFS: Example

Consider following graph.



Step V:

- Visit all the adjacent vertices of Vertex B which are unvisited (No any vertex).
- Remove B from queue.

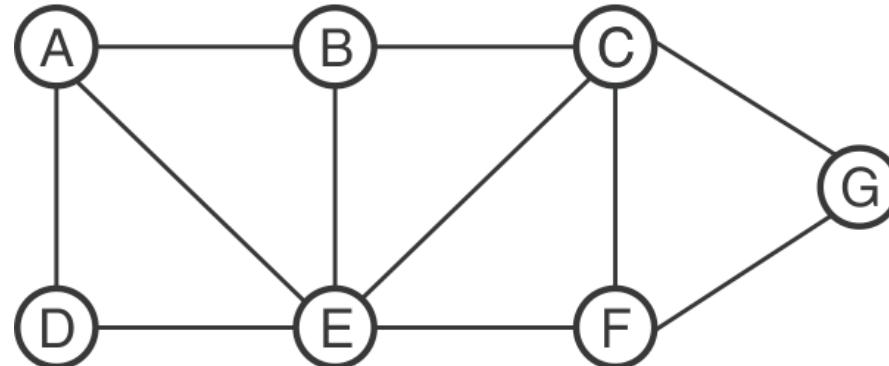


Queue



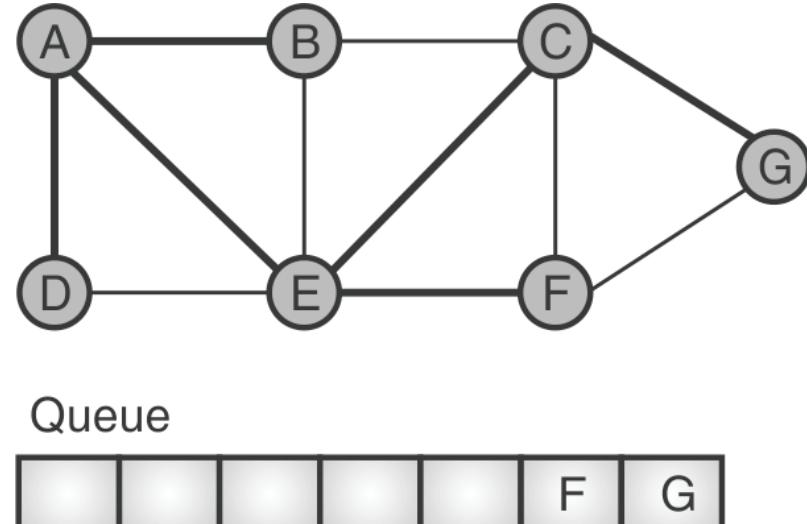
BFS: Example

Consider following graph.



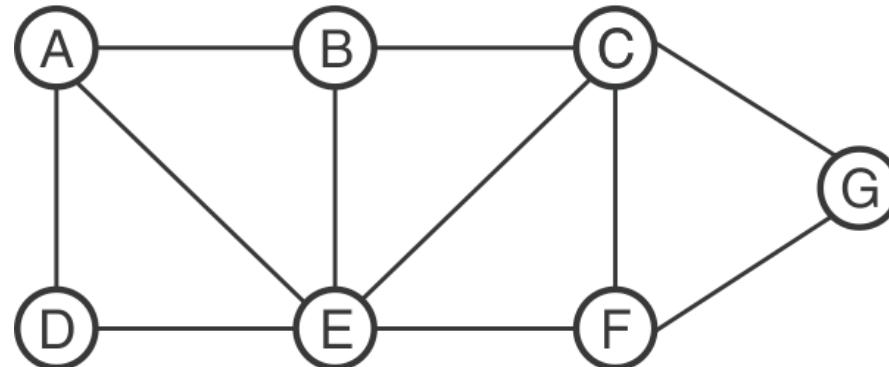
Step VI:

- Visit all the adjacent vertices of Vertex C which are unvisited (G).
- Insert the vertex G into Queue and remove vertex C from the queue.



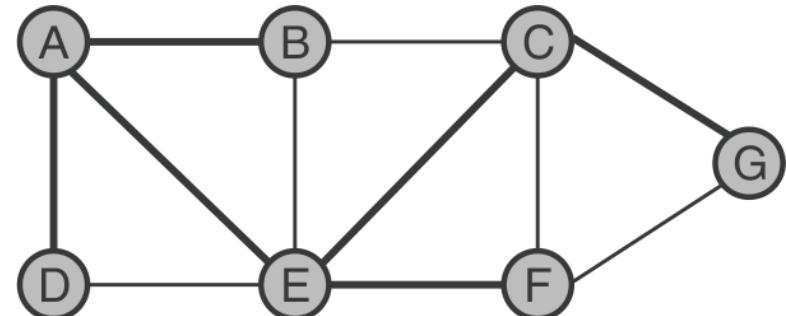
BFS: Example

Consider following graph.



Step VII:

- Visit all the adjacent vertices of Vertex F which are unvisited (No any vertex).
- Remove F from queue.

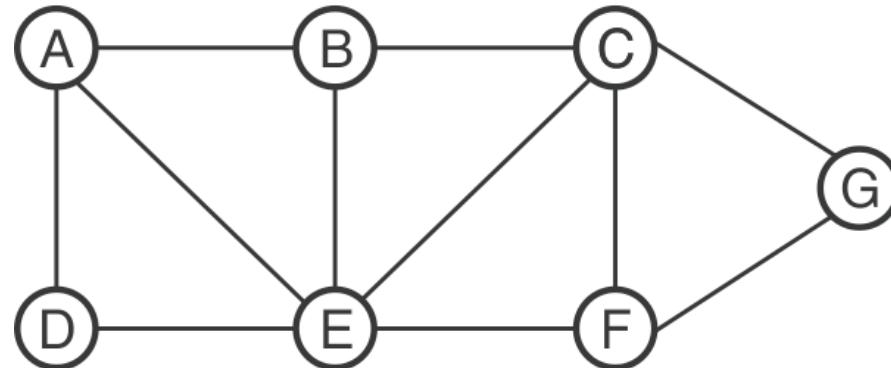


Queue



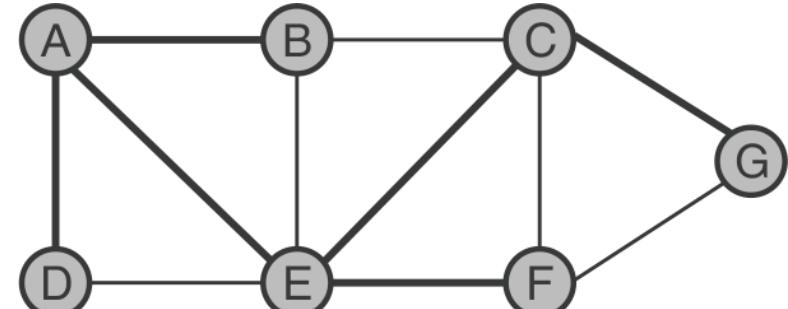
BFS: Example

Consider following graph.



Step VII:

- Visit all the adjacent vertices of Vertex G which are unvisited (No any vertex).
- Remove G from queue.
- As now queue is completely empty, stop the process.

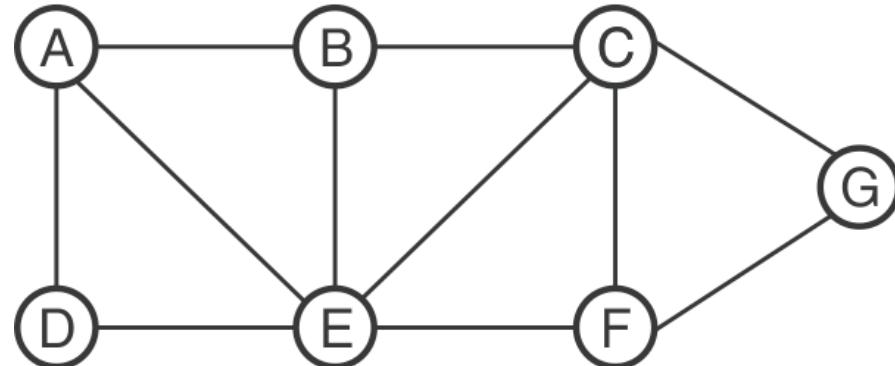


Queue

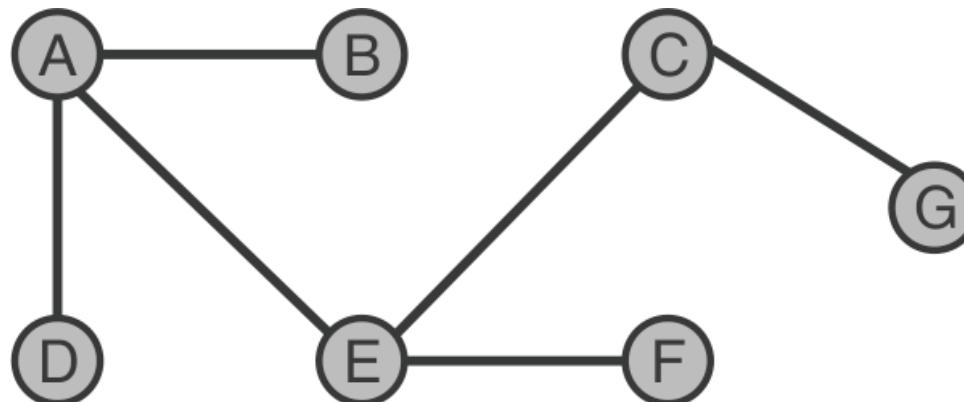


BFS: Example

Consider following graph.



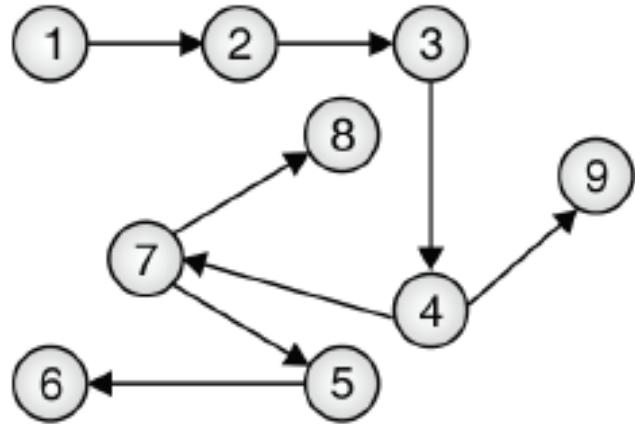
Final result of BFS is spanning tree :



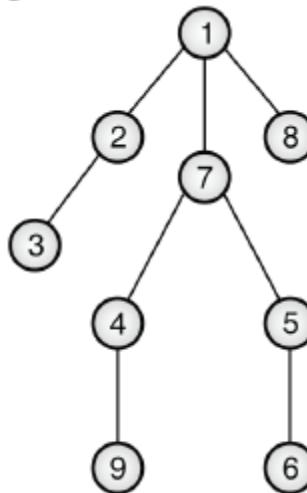
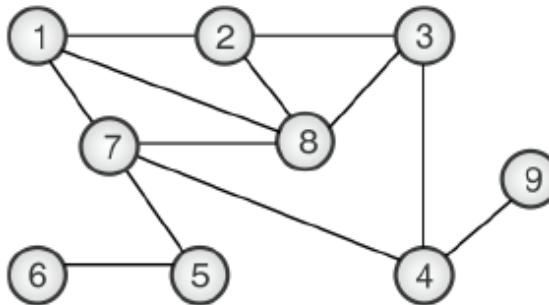
The output of BFS traversal is :A, D, E, B, C, F, G

Example #4

- Define DFS and BFS for a graph. Show BFS and DFS for the following graph with starting vertex.



DFS traversal - 1, 2, 3, 4, 7, 5, 6, 8, 9



BFS Traversal

1
2 7 8
3 4 5
9 6

Advantages :BFS

- In this procedure at any way it will find the goal.
- It does not follow a single unfruitful path for a long time. It finds the minimal solution in case of multiple paths.

Disadvantages :BFS

- BFS consumes large memory space. Its time complexity is more.
- It has long pathways, when all paths to a destination are on approximately the same search depth.

APPLICATIONS OF GRAPH

- Graphs can be used to model many types of relations and processes in physical, biological, social and information systems.
- Many practical problems can be represented by graphs.
 - In computer science, graphs are used to represent networks of communication, data organization, and computational devices.
 - Graph theory is also used to study molecules in chemistry and physics.
 - In mathematics, graphs are useful in geometry.

APPLICATIONS OF GRAPH

- Graphs can be used to model many types of relations and processes in physical, biological, social and information systems.
- Many practical problems can be represented by graphs.
 - Weighted graphs are used to represent structures in which pairwise connections have some numerical values. Ex: Road Network.
 - Graph algorithms are useful for calculating the shortest path in Routing.
 - Maps – finding the shortest / cheapest path for a car from one city to another, by using given roads.

Differentiate between Tree and Graph

Parameter	Tree	Graph
Path	Tree is special form of graph i.e. Minimally connected graph and having only one path between any two vertices.	In graph there can be more than one path i.e. graph can have uni-directional or bi-directional paths (edges) between nodes
Loops	Tree is a special case of graph having no loops, no circuits and no self loops.	Graph can have loops, circuits as well as can have self-loops.
Root Node	In tree there is exactly one root node and every child have only one parent.	In graph there is no such concept of root node.
Parent Child relationship	In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa. In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa.	In Graph there is no such parent child relationship.

Differentiate between Tree and Graph

Parameter	Tree	Graph
Complexity	Trees are less complex than graphs as having no cycles, no self loops and still connected.	Graphs are more complex in comparison with trees as it can have cycles, loops etc
Types of Traversal	Tree traversal is a kind of special case of traversal of graph. Tree is traversed in Pre-Order, In-Order and Post-Order (all three in DFS or in BFS algorithm)	Graph is traversed by DFS: Depth First Search and in BFS : Breadth First Search algorithm
Types	Different types of trees are : Binary Tree, Binary Search Tree, AVL tree, Heaps.	There are mainly two types of Graphs : Directed and Undirected graphs.
Applications	Sorting and searching like Tree Traversal & Binary Search.	Coloring of maps, algorithms, Graph coloring, job scheduling, etc.

Greedy Algorithms: Introduction

- When the problem has many feasible solutions with different cost or benefit, finding the best solution is known as an **optimization problem**. And best solution is known as the **optimal solution**.
- Examples:
 - Knapsack problem
 - Shortest Path
- Decisions are completely locally optimal. This method constructs the solution simply by looking at current benefit without exploring future possibilities and hence known as greedy.

Greedy Technique Definition

- The choice made under greedy solution procedure are irrevocable, means once we have selected the local best solution, it cannot be backtracked.
- Thus, the choice made at each step in the greedy method should be:
 - ✓ **Feasible:** Choice should satisfy problem constraints.
 - ✓ **Locally optimal:** Best solution from all feasible solution at current stage should be selected.
 - ✓ **Irrevocable:** Once the choice is made, it cannot be altered. i.e. if a feasible solution is selected(rejected) in step i, it cannot be rejected (selected) in subsequent stages.

Advantages of Greedy Algorithms

- It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.
- Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and Conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size gets smaller and the number of sub-problems increases.

Disadvantages of Greedy Algorithms

- The difficult part is that for greedy algorithms we have to work much harder to understand correctness issues.
- Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Greedy Algorithms: Generic Algorithm

```
Algorithm Greedy_Approach(L,n)
```

```
//Description: Solve the given problem using greedy approach  
//Input: L - List of possible choices, n-size of solution of given  
problem.  
//Output: Set solution containing solution of given problem.
```

```
Solution  $\leftarrow \Phi$   
for i  $\leftarrow 1$  to n do  
    Choice  $\leftarrow$  Select(L)  
    if(feasible(Choice U Solution)) then  
        Solution  $\leftarrow$  Choice U Solution  
    end  
end  
return Solution
```

Applications of the Greedy Strategy

- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Graph - Vertex Cover
- Knapsack Problem
- Dijkstra's Minimal Spanning Tree Algorithm
- Job Scheduling Problem

The time complexity of the Greedy algorithm is $O(n)$.

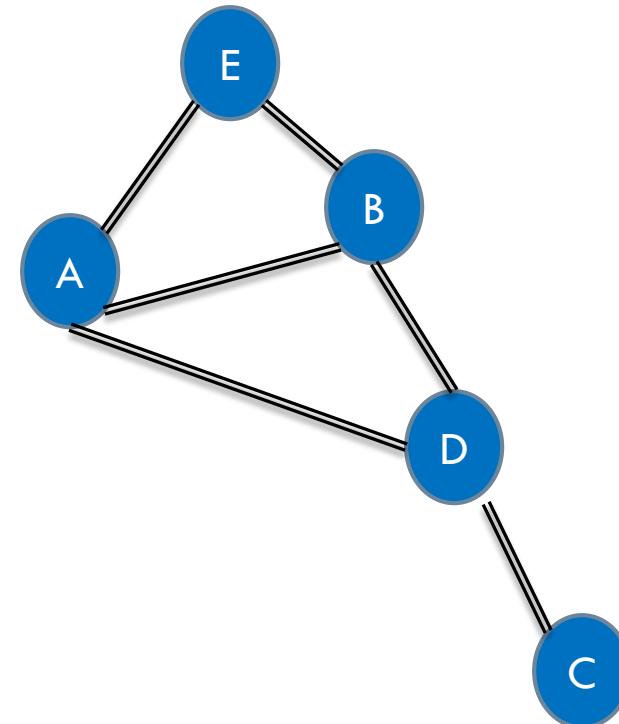
Minimum Spanning Tree(MST)

- A tree is a connected graph with no cycle.
- A spanning tree is a sub graph of G that has all vertices of G and is a tree.
- A MST of weighted graph G is the spanning tree of G whose edges sum to minimum weight.

Minimum Spanning Tree(MST)

Graph

- $G=(V,E)$ is defined by a set of V and set of E joining those vertices.
- $V=\{A, B, C, D, E\}$
- $E=\{AB,AD,AE,BD, BE, DC\}$

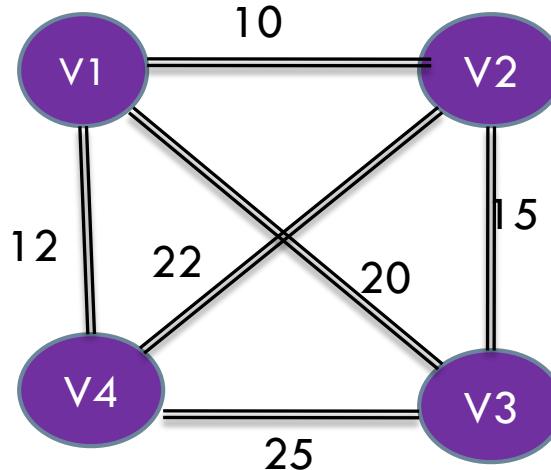


Minimum Spanning Tree(MST)

Weighted Graph

- $G=(V, E, W)$ is defined if some weight and cost is associated with each of its edges.
- $W \rightarrow$ Weight

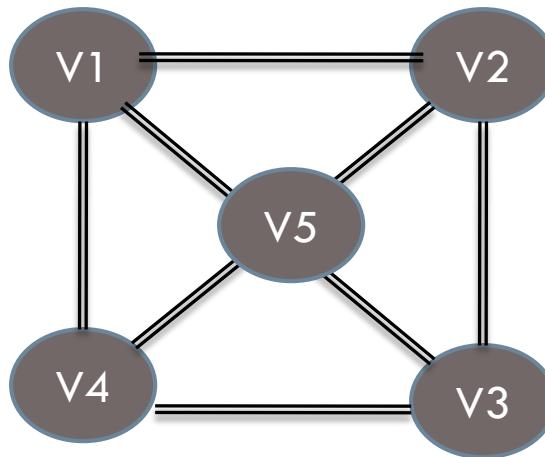
- $V=\{V_1, V_2, V_3, V_4\}$
- $E=\{V_1V_2, V_1V_3, \dots\}$
- $W=\{10, 20, \dots\}$



Minimum Spanning Tree(MST)

Tree

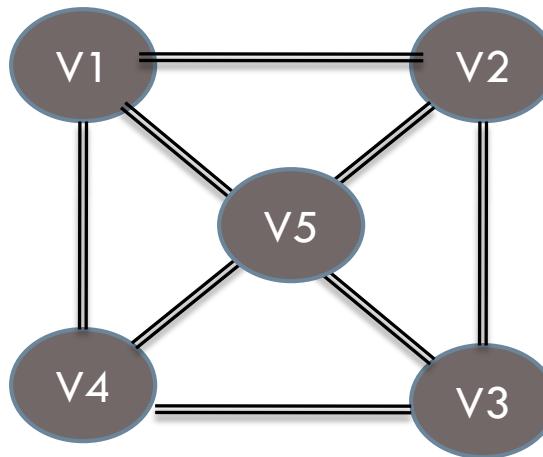
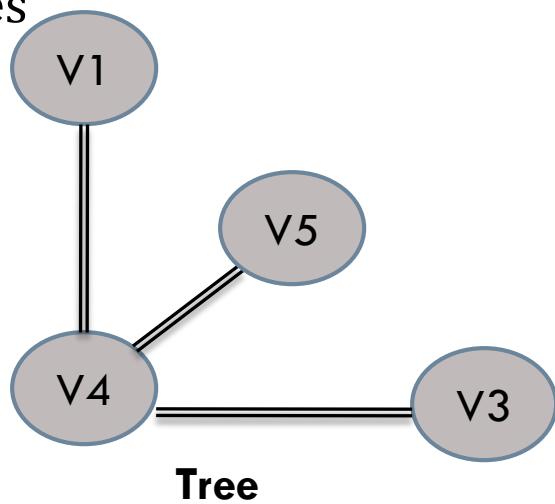
- $T(V', E')$ is a subset of graph $G=(V, E)$
- $V' \rightarrow$ Subset of Vertices V
- $E' \rightarrow$ Subset of Edges E
- Tree does not contain a cycle.
- Where graph, sub graph can have cycles



Minimum Spanning Tree(MST)

Tree

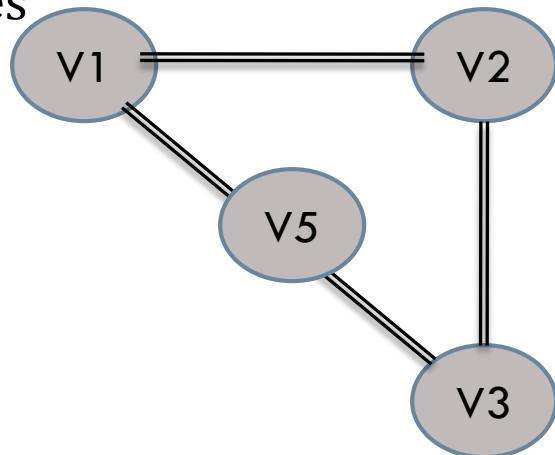
- $T(V', E')$ is a subset of graph $G=(V, E)$
- $V' \rightarrow$ Subset of Vertices V
- $E' \rightarrow$ Subset of Edges E
- Tree does not contain a cycle.
- Where graph, sub graph can have cycles



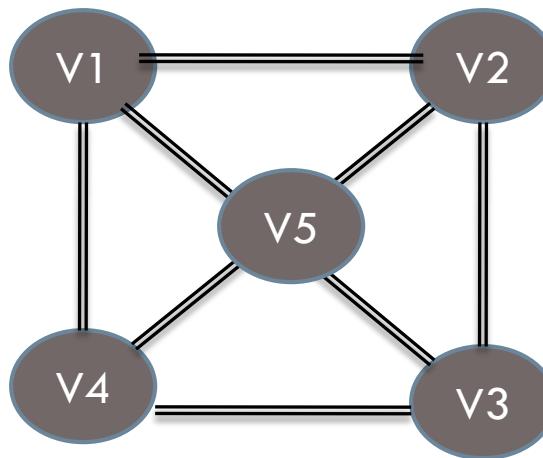
Minimum Spanning Tree(MST)

Sub Graph

- $T(V', E')$ is a subset of graph $G=(V, E)$
- $V' \rightarrow$ Subset of Vertices V
- $E' \rightarrow$ Subset of Edges E
- Tree does not contain a cycle.
- Where graph, sub graph can have cycles



Sub graph of G

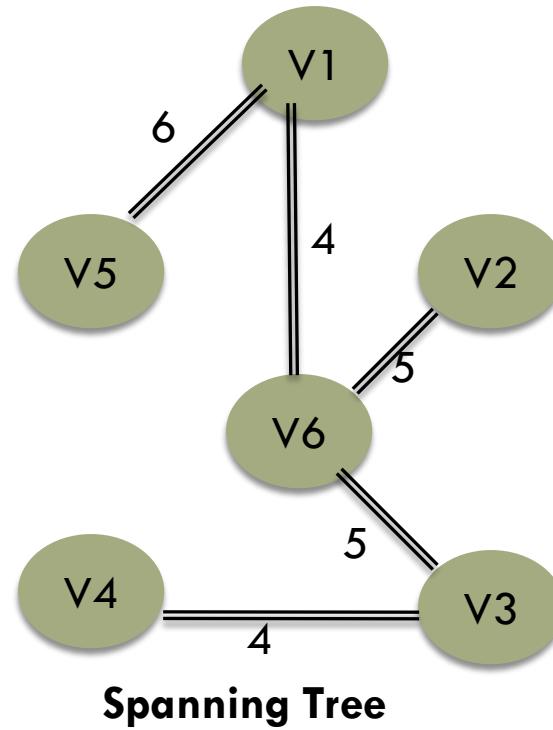
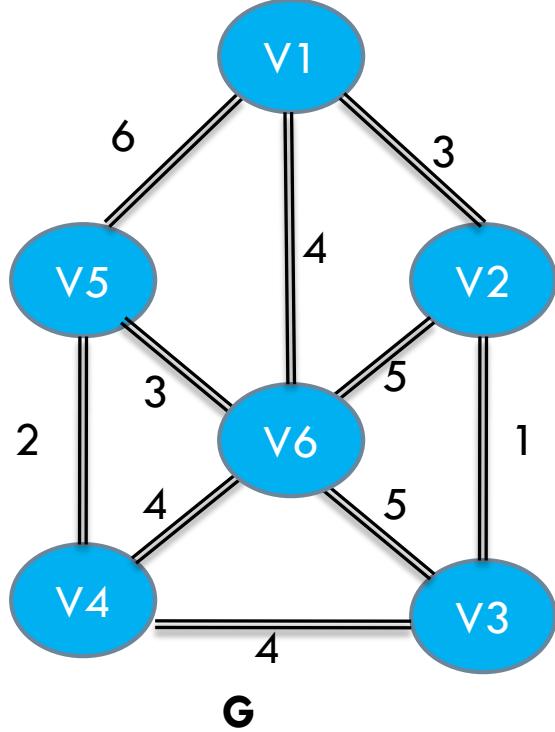


Minimum Spanning Tree(MST)

Spanning Tree

- $T(V', E')$ is a tree of connected, undirected, weighted graph $G=(V, E, W)$, which contains all the vertices of G & some or all edges of G so,

$$V'=V \text{ and } E' \subset E$$

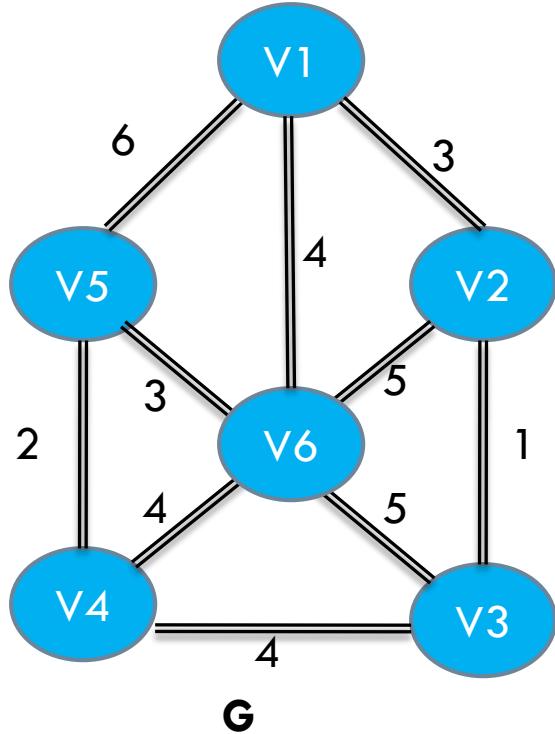


Minimum Spanning Tree(MST)

Minimum Spanning Tree

- $T(V', E')$ is a tree of connected, undirected, weighted graph $G=(V, E, W)$, which contains all the vertices of G & some or all edges of G so,

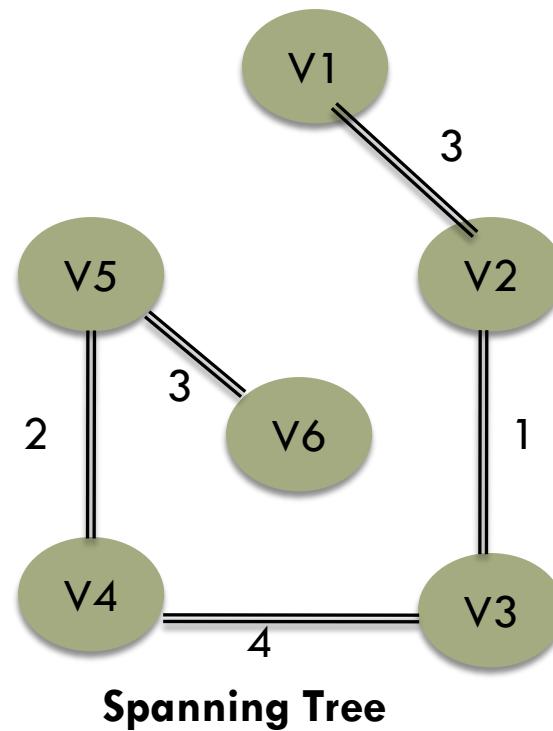
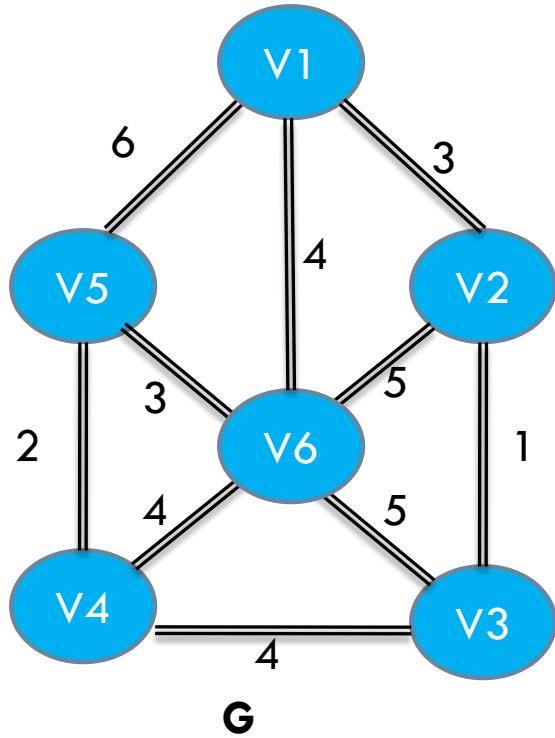
$$V'=V \text{ and } E' \subset E$$



Minimum Spanning Tree(MST)

Minimum Spanning Tree

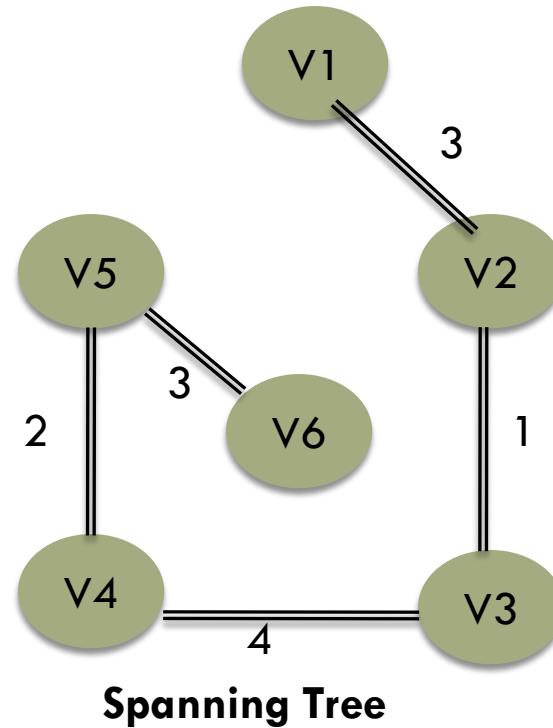
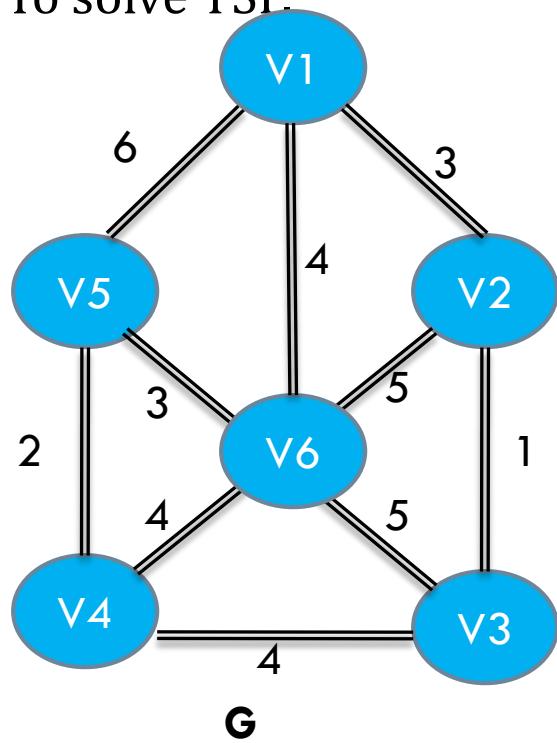
- Graph G can have many spanning trees with a different cost. Minimum Spanning Tree(MST) is Spanning Tree(ST) with minimum cost.



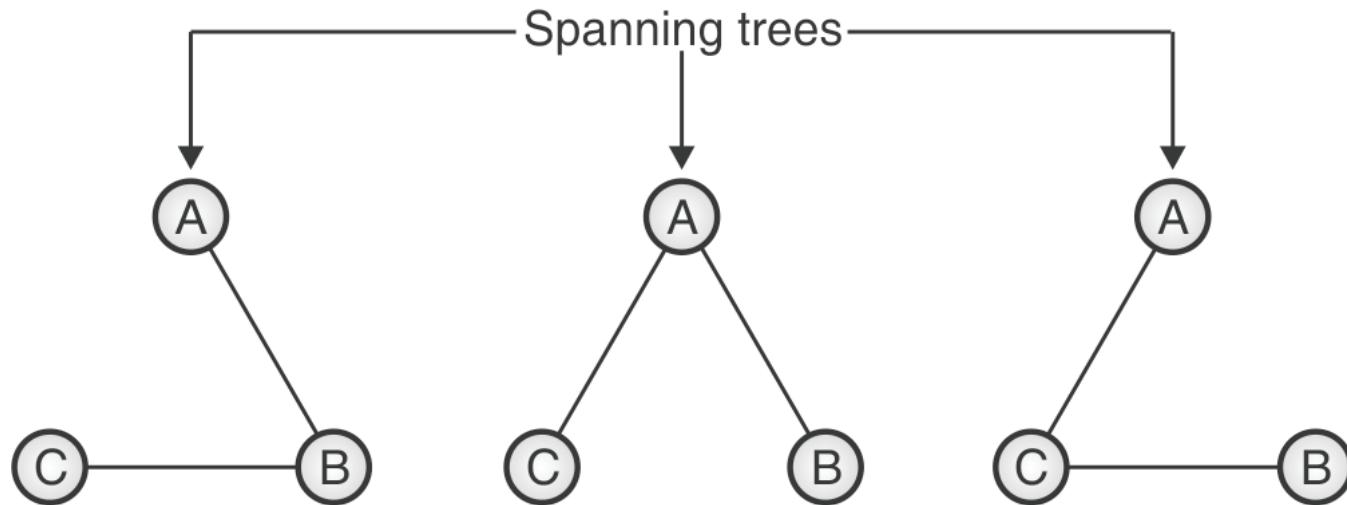
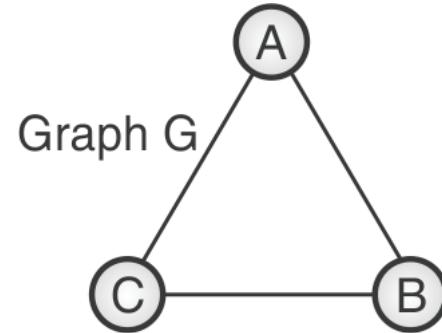
Minimum Spanning Tree(MST):

Minimum Spanning Tree: Applications

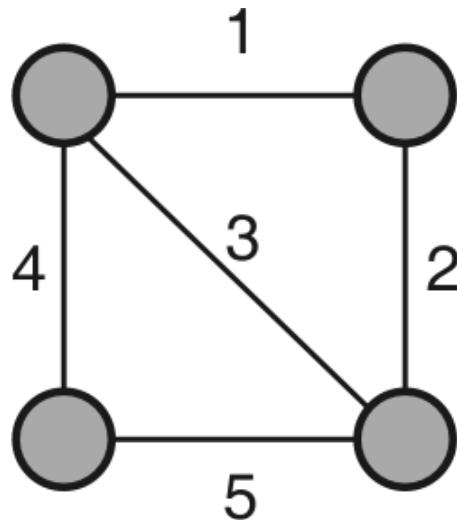
- Network design
- To implement efficient routing algorithm
- To solve TSP.



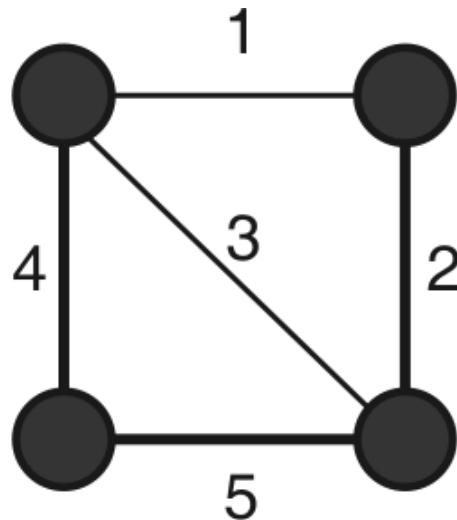
Minimum Spanning Tree(MST)



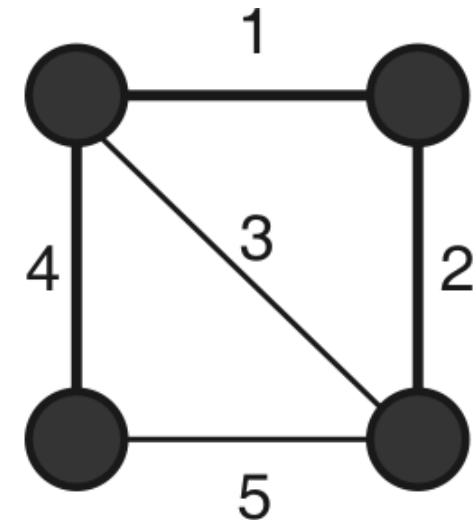
Minimum Spanning Tree(MST)



Undirected
Graph



Spanning
Tree
 $\text{Cost} = 11(=4+5+2)$



Minimum Spanning
Tree
 $\text{Cost} = 7(=4+1+2)$

Prim's Algorithm

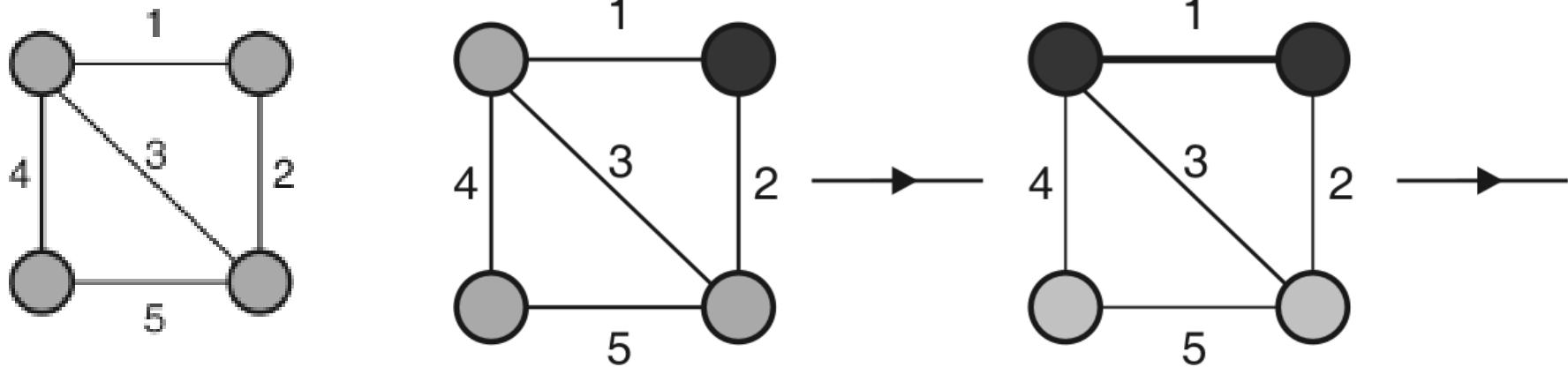
- Prim's Algorithm also uses Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position.
- Unlike an **edge in Kruskal's**, we add vertex to the growing spanning tree in Prim's.
- It is a variation of Dijkstra's algorithm.

Prim's Algorithm: Steps

- **Step 1 :** Maintain two disjoint sets of vertices.
 - One containing vertices that are in the growing spanning tree and
 - other that are not in the growing spanning tree.
- **Step 2:** Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and
 - add it into the growing spanning tree.
 - This can be done using Priority Queues.
 - Insert the vertices that are connected to growing spanning tree, into the Priority Queue.
- **Step 3:** Check for cycles.
 - To do that, mark the nodes which have been already selected and
 - insert only those nodes in the Priority Queue that are not marked.

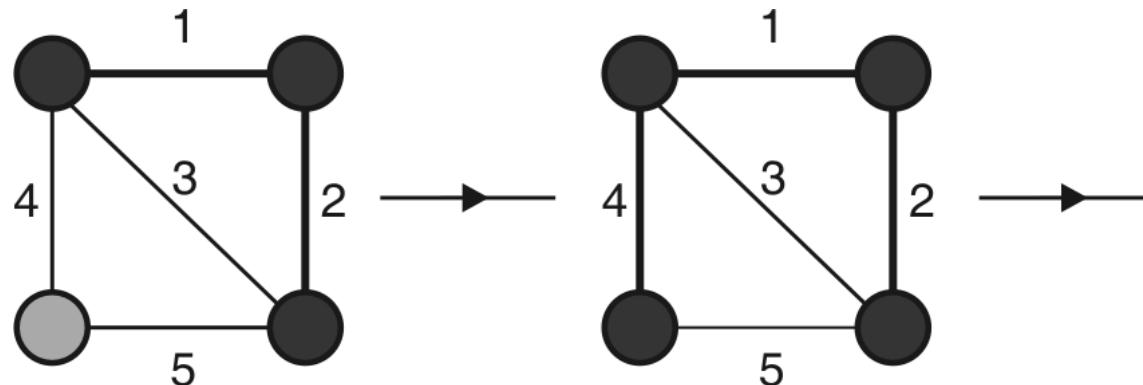
Prim's Algorithm: Example #1

- In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it.
- In each iteration we will mark a new vertex that is adjacent to the one that we have already marked.
- As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1.



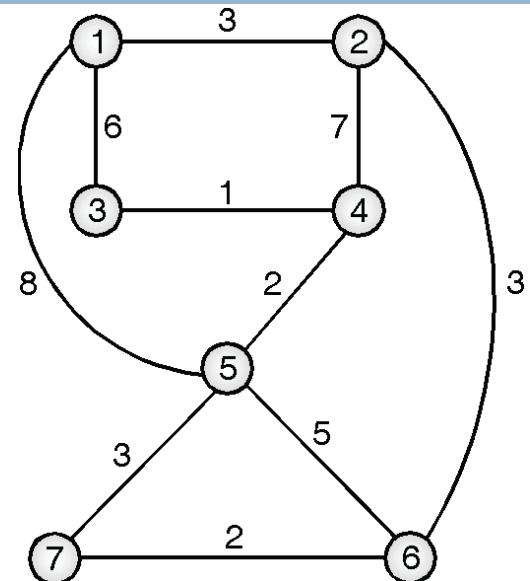
Prim's Algorithm: Example #1

- In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex.
- Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 ($= 1 + 2 + 4$).



Prim's Algorithm: Example #2

- Find minimum spanning tree for following graph using Prim's algorithm. Show various steps.



Step 1 :

Edge	Nodes	1	2	3	4	5	6	7
Distance	-	3	6	∞	8	∞	∞	
Distance from	-	1	1	1	1	1	1	

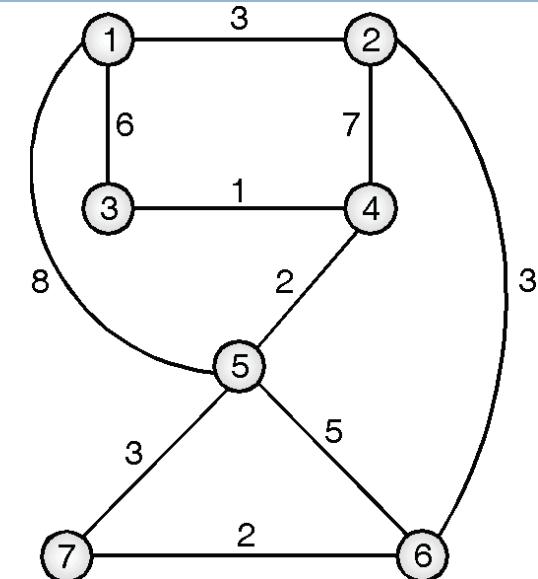
↑
Nearest Node is selected.

Prim's Algorithm: Example #2

Step 1 :

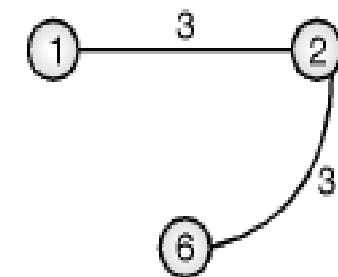
Edge	Nodes	1	2	3	4	5	6	7	1	2
Distance	-	3	6	∞	8	∞	∞			
Distance from	-	1	1	1	1	1	1			
				↑						

Nearest Node is selected.

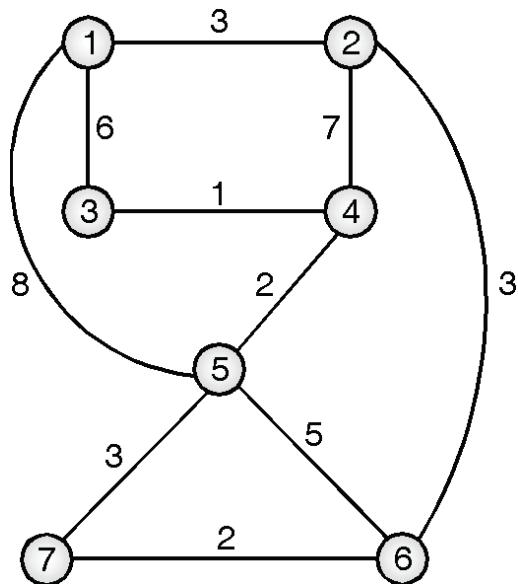


Step 2

Edge	Nodes	1	2	3	4	5	6	7	1	2
Distance	-	3	6	7	8	3	∞			
Distance from	-	1	1	2	1	2	1			
				↑						



Prim's Algorithm: Example #2



Step 2

Edge

Nodes	1	2	3	4	5	6	7	
Distance	-	3	6	7	8	3	∞	
Distance from	-	1	1	2	1	2	1	

Diagram showing the state of the algorithm after Step 2. Nodes 1 and 2 are highlighted in orange. The shortest distance to node 1 is 1, and to node 2 is 3. A purple arrow points to the edge between node 5 and node 6, indicating it is being considered for selection.

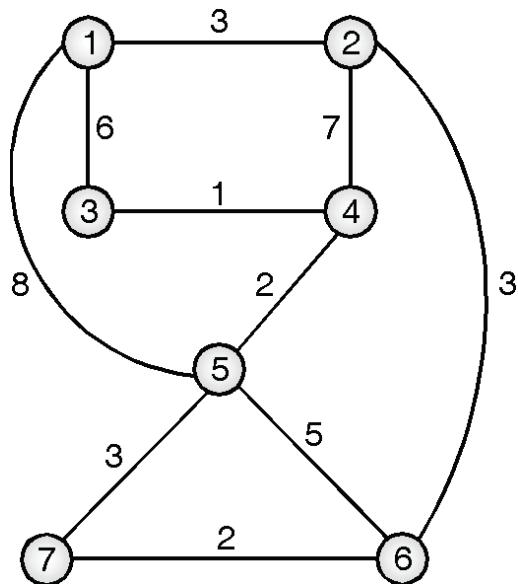
Step 3

edge

Nodes	1	2	3	4	5	6	7	
Distance	-	3	6	7	5	3	2	
Distance from	-	1	1	2	6	2	6	

Diagram showing the state of the algorithm after Step 3. Nodes 1 and 2 are highlighted in orange. The shortest distance to node 1 is 1, and to node 2 is 3. A blue arrow points to the edge between node 7 and node 6, indicating it is being considered for selection.

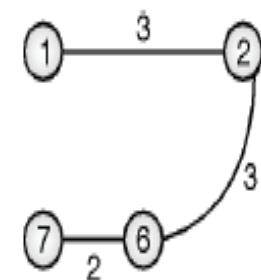
Prim's Algorithm: Example #2



Step 3

edge

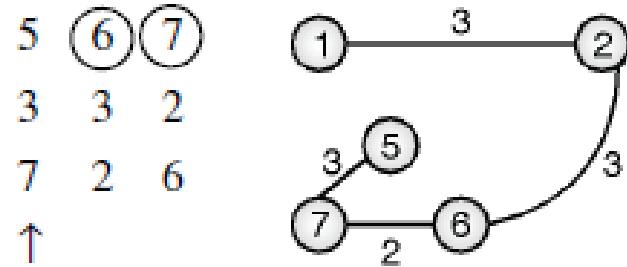
Nodes	1	2	3	4	5	6	7
Distance	-	3	6	7	5	3	2
Distance from	-	1	1	2	6	2	6



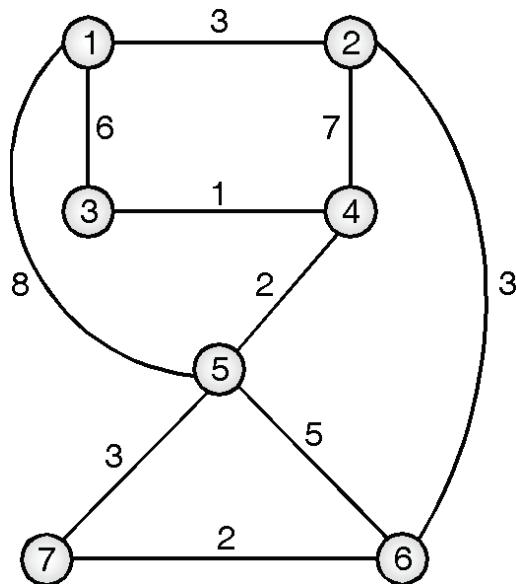
Step 4

Edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	7	3	3	2
Distance from	-	1	1	2	7	2	6



Prim's Algorithm: Example #2



Step 4

Edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	7	3	3	2
Distance from	-	1	1	2	7	2	6

↑

A partial spanning tree is shown with nodes 1, 2, 3, 4, 5, and 6. Node 7 is not yet connected. The edges included are (1,2), (1,3), (2,4), (3,5), (4,5), (5,6), and (6,2). The edge (5,7) is highlighted with a red arrow pointing to it.

Step 5

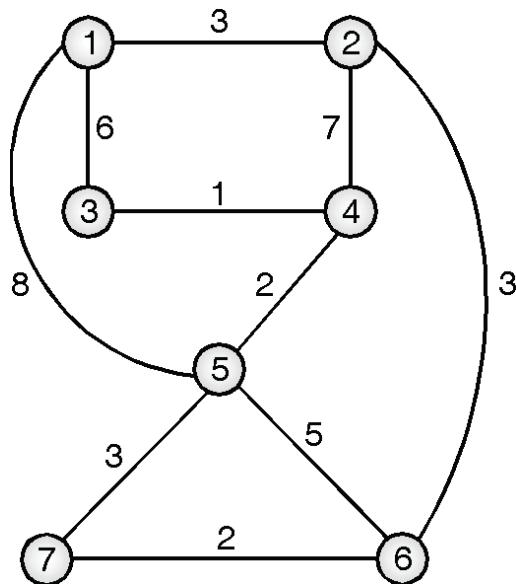
edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	2	3	3	2
Distance from	-	1	1	5	7	2	6

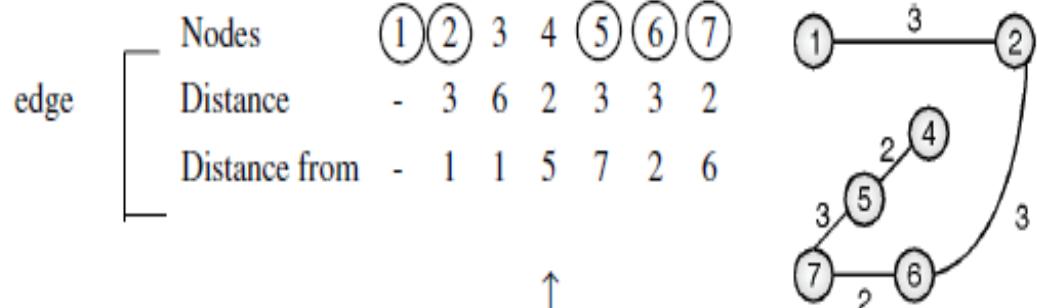
↑

The partial spanning tree now includes node 7. The edges are (1,2), (1,3), (2,4), (3,5), (4,5), (5,6), (6,2), and (5,7). The edge (6,7) is highlighted with a red arrow pointing to it.

Prim's Algorithm: Example #2



Step 5

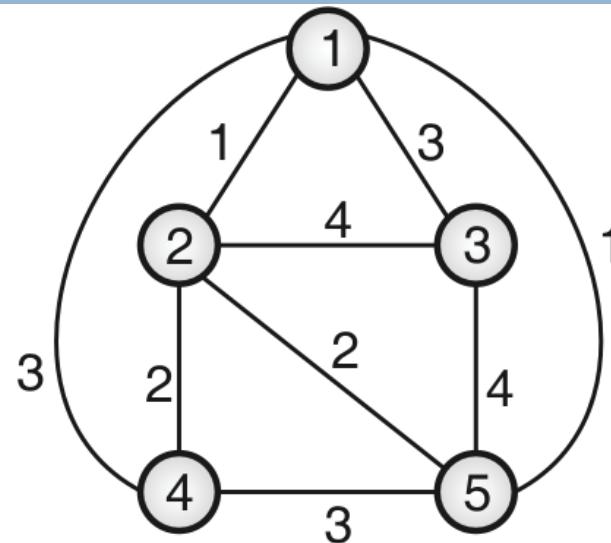


Step 6 :

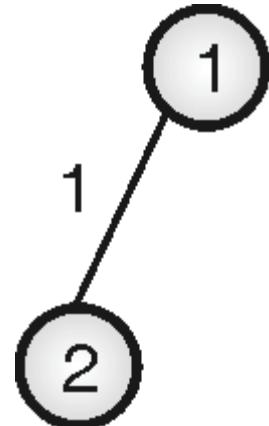


Prim's Algorithm: Example #3

Draw the MST using prim's Algorithm



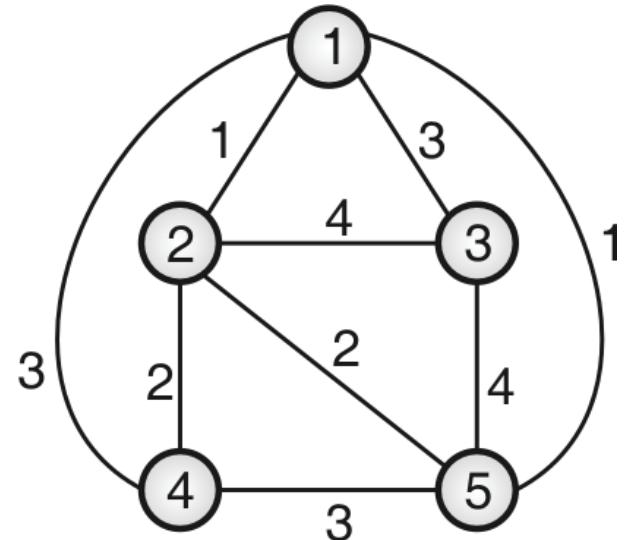
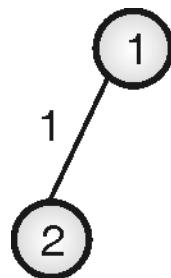
Initially the vertex 1 with its nearest vertex 2 is added to the spanning tree



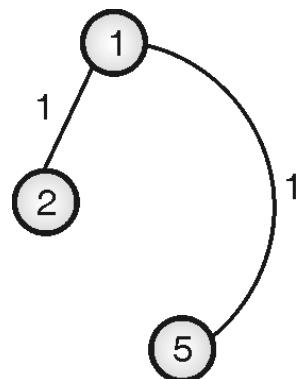
Prim's Algorithm: Example #3

Draw the MST using prim's Algorithm

Initially the vertex 1 with its nearest vertex 2 is added to the spanning tree



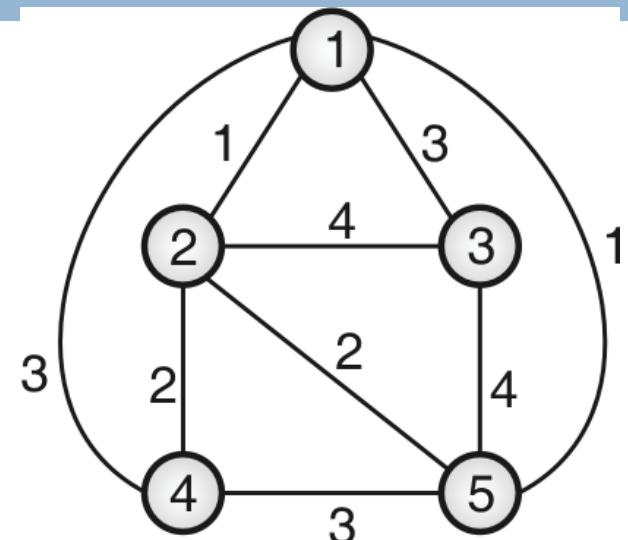
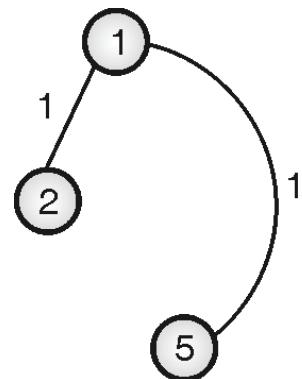
The nearest vertex of the vertices (1, 2) is 5. The edge (1, 5) is added.



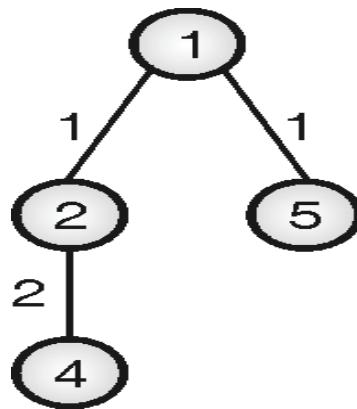
Prim's Algorithm: Example #3

Draw the MST using prim's Algorithm

The nearest vertex of the vertices (1, 2) is 5. The edge (1, 5) is added.



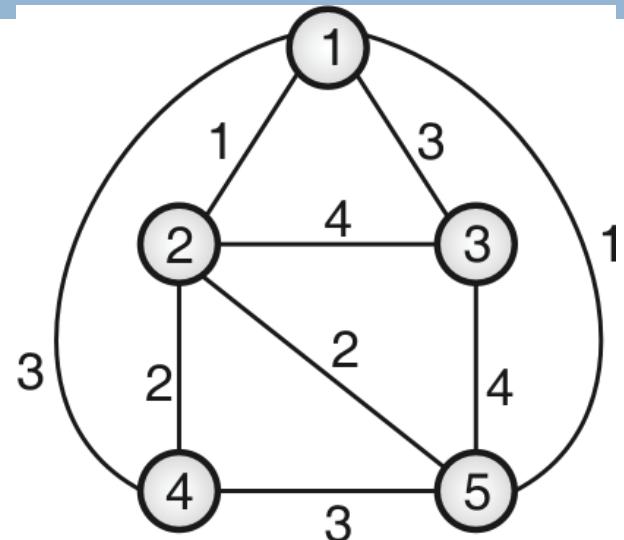
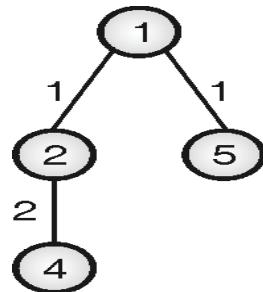
The nearest vertex of the vertices (1, 2, 5) is the vertex, The edge (2, 4) is added



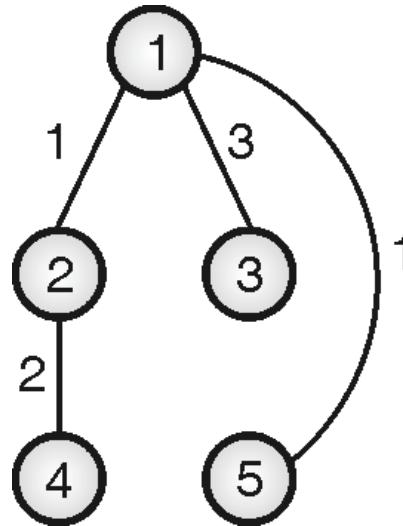
Prim's Algorithm: Example #3

Draw the MST using prim's Algorithm

The nearest vertex of the vertices (1, 2, 5) is the vertex, The edge (2, 4) is added



The nearest vertex 3 is added through the edge (1, 3).



Kruskal's Algorithm

- Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.
- Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and adds it to the growing spanning tree.

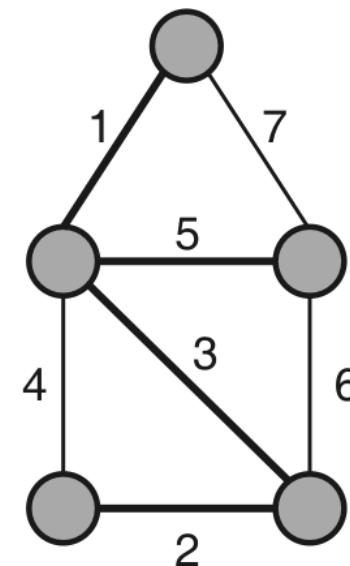
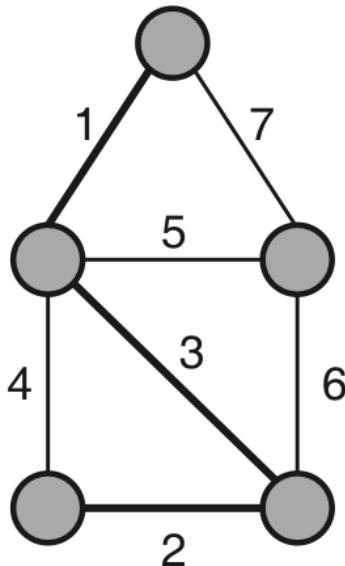
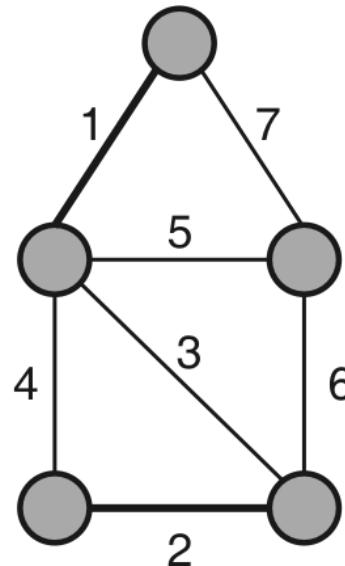
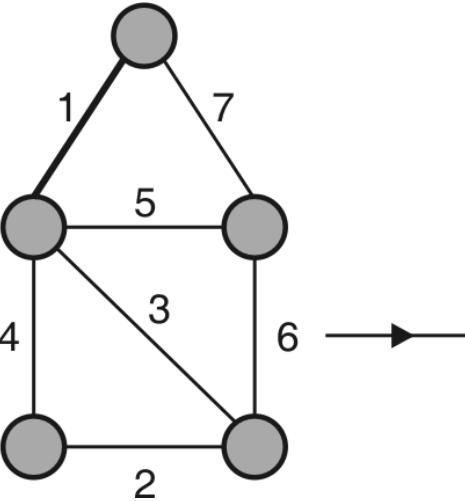
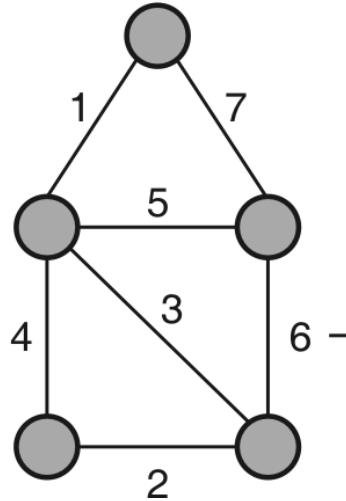
Kruskal's Algorithm: Steps

- **Step 1 :** Sort the graph edges with respect to their weights.
- **Step 2 :** Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- **Step 3 :** Only add edges which doesn't form a cycle, edges which connect only disconnected components.

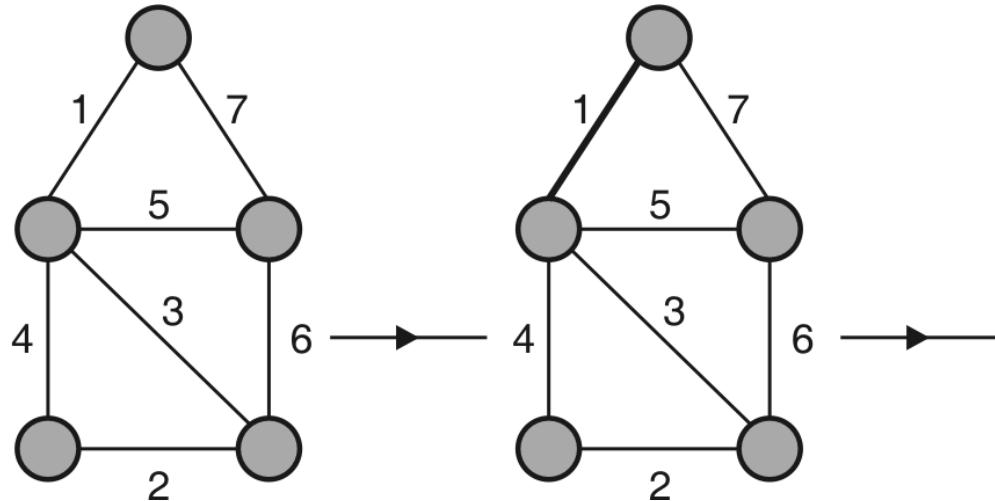
Kruskal's Algorithm

- How to check if 2 vertices are connected or not ?
- This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not.
- But DFS will make time complexity large as it has an order of $O(V+E)$ where V is the number of vertices, E is the number of edges.
- So the best solution is “Disjoint Sets”.
- Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Kruskal's Algorithm

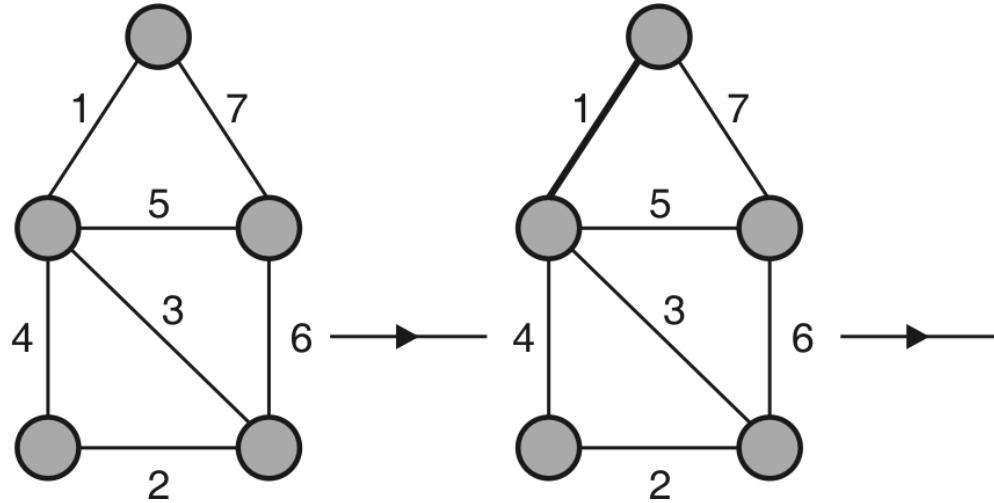


Kruskal's Algorithm

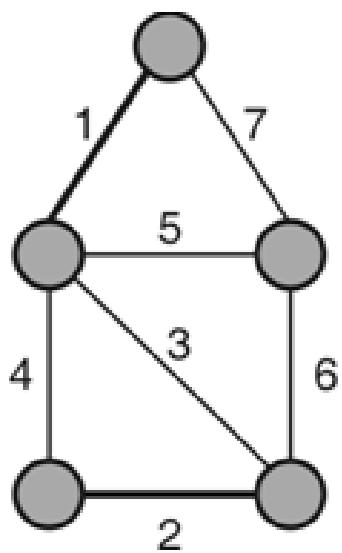


- Kruskal's algorithm, at each iteration we will select the edge with the lowest weight.
- So, we will start with the lowest weighted edge first i.e., the edges with weight 1.

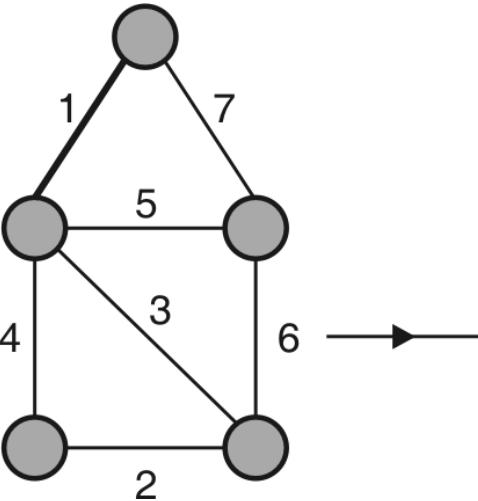
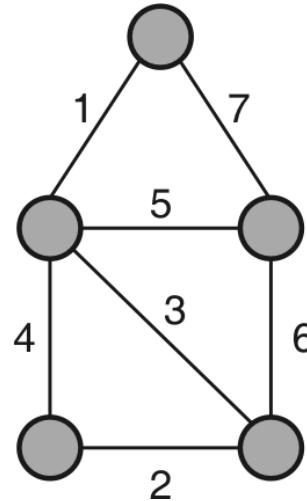
Kruskal's Algorithm



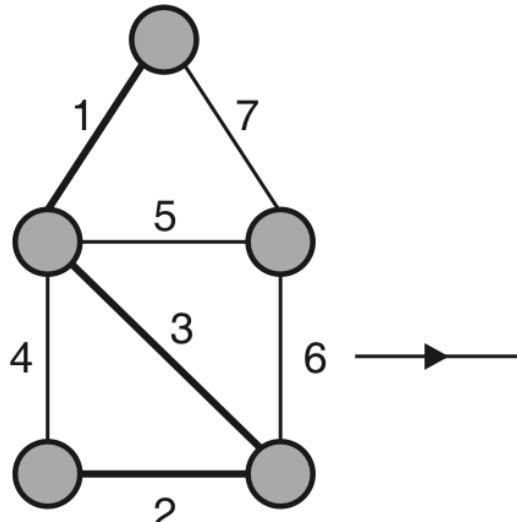
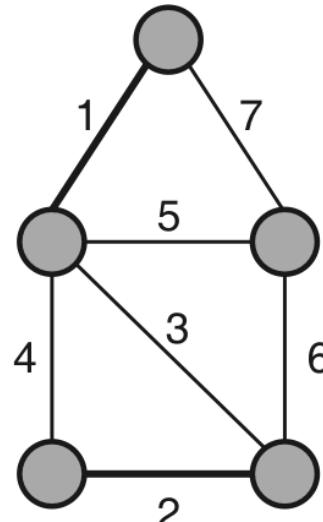
- After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint.



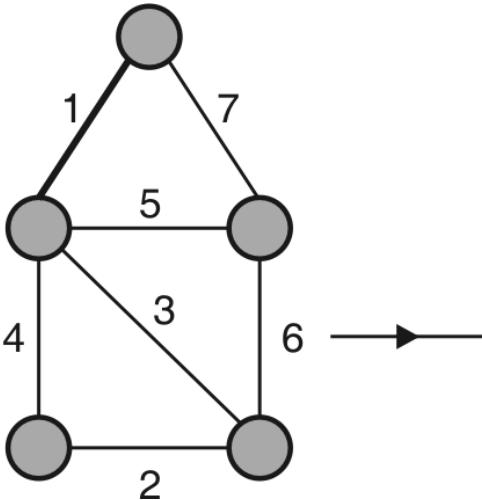
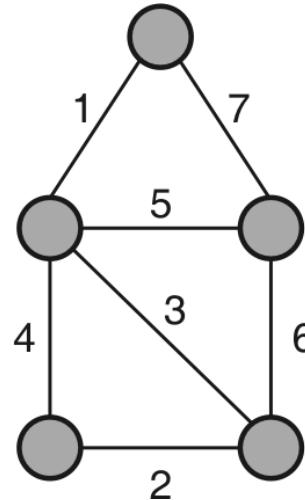
Kruskal's Algorithm



- Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph.

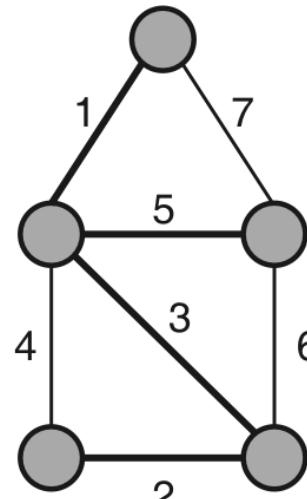
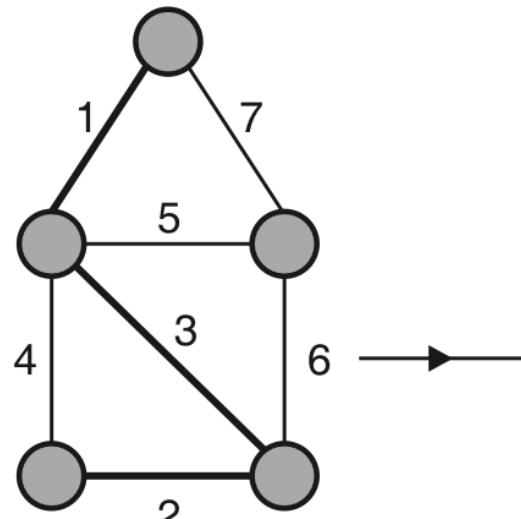
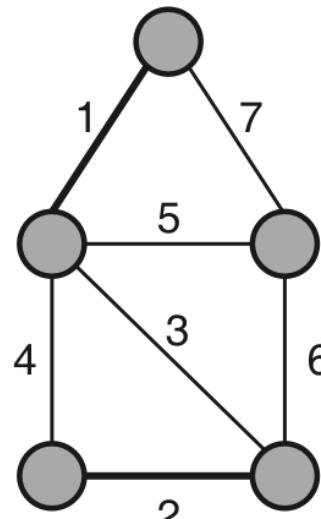


Kruskal's Algorithm

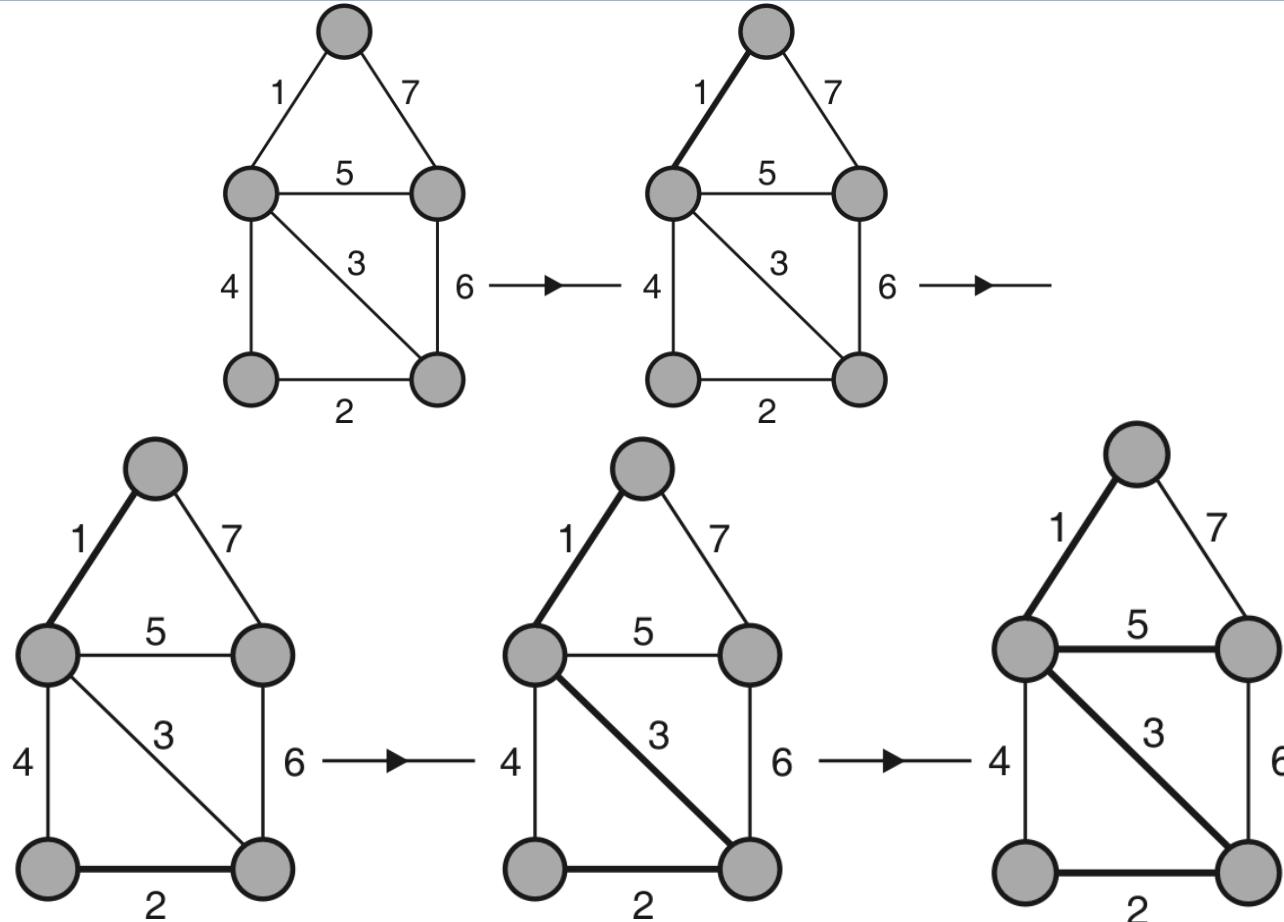


- Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles.

- So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them.



Kruskal's Algorithm



At the end, we end up with a minimum spanning tree with total cost $11 (= 1 + 2 + 3 + 5)$.

Kruskal's Algorithm: Example #1

- Find minimum spanning tree for following graph using Kruskal's algorithm. Show various steps.
- Step I : First we sort the edges on weight in ascending order.

Edges : E1 : 3 4 1

E2 : 4 5 2

E3 : 6 7 2

E4 : 1 2 3

E5 : 2 7 3

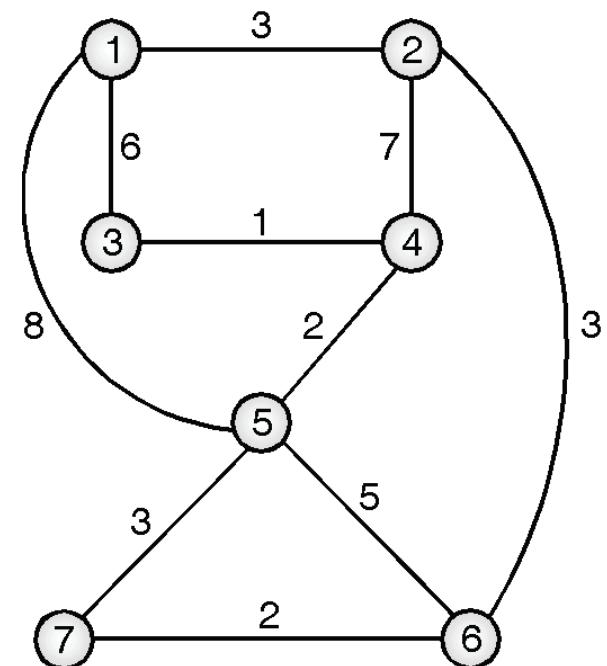
E6 : 5 6 3

E7 : 5 7 5

E8 : 1 3 6

E9 : 2 4 7

E10 : 1 5 8



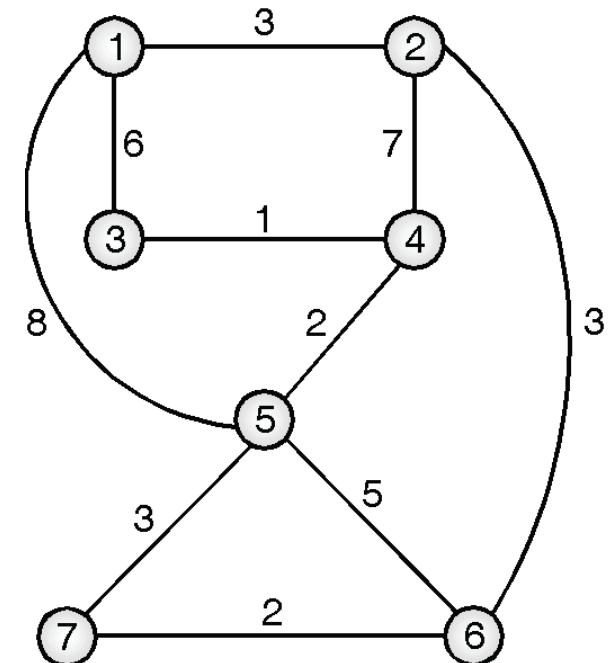
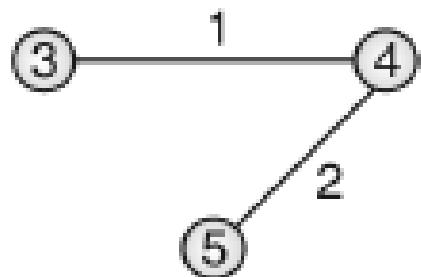
Kruskal's Algorithm: Example #1

- **Step-II:** In the spanning tree, the edges from E1 to E10 are added sequentially to spanning tree. When cycle is formed by an edge, it is discarded.

Add E1 (3, 4, 1)



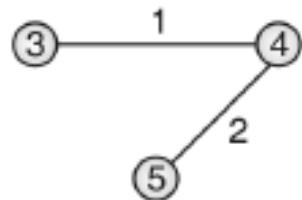
Add E2 (4, 5, 2)



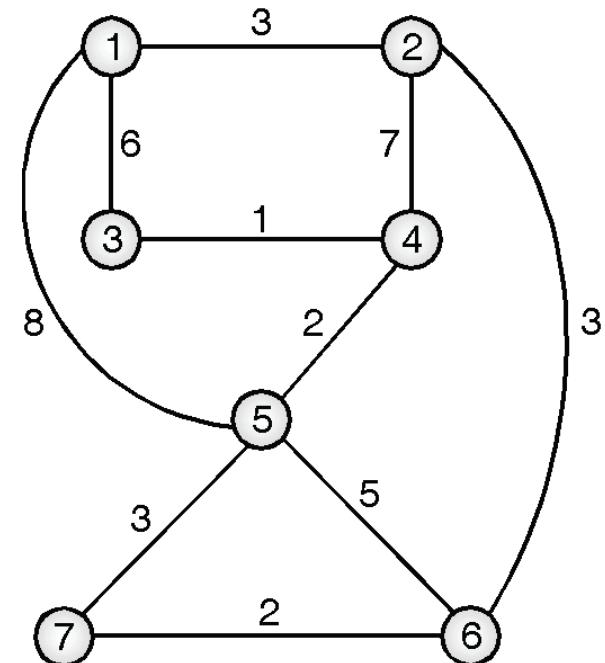
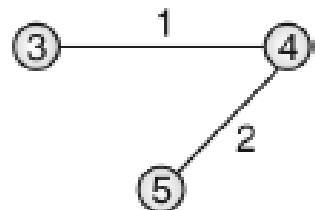
Kruskal's Algorithm: Example #1

- **Step-II:** In the spanning tree, the edges from E1 to E10 are added sequentially to spanning tree. When cycle is formed by an edge, it is discarded.

Add E2 (4, 5, 2)



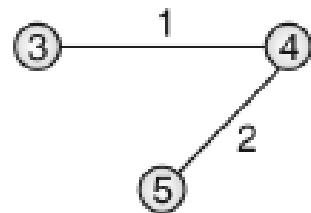
Add E3 (7, 6, 2)



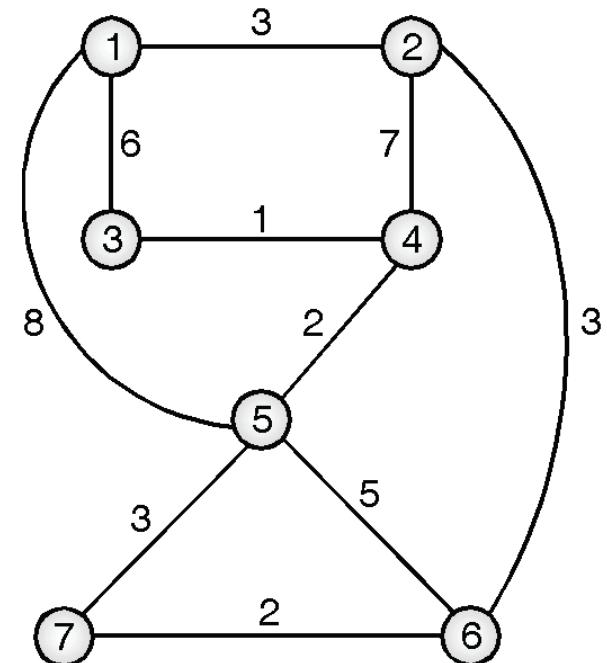
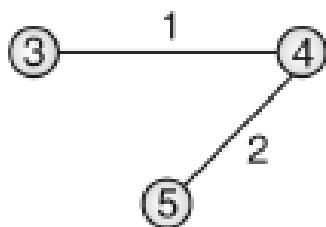
Kruskal's Algorithm: Example #1

- **Step-II:** In the spanning tree, the edges from E1 to E10 are added sequentially to spanning tree. When cycle is formed by an edge, it is discarded.

Add E3 (7, 6, 2)

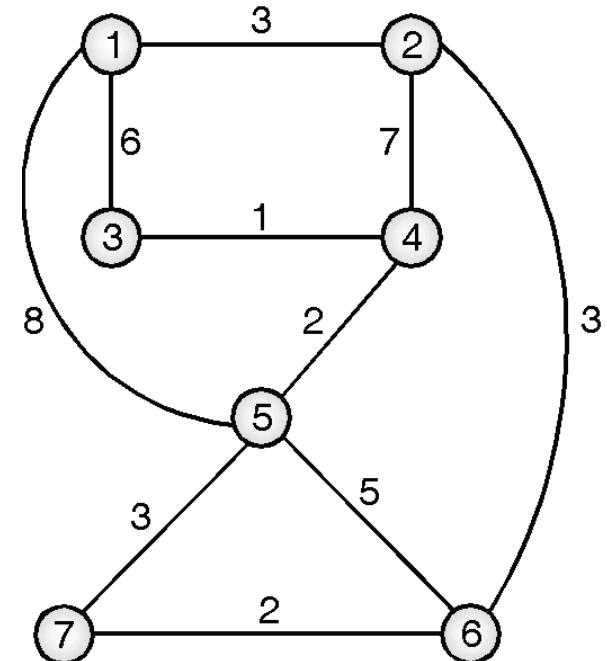
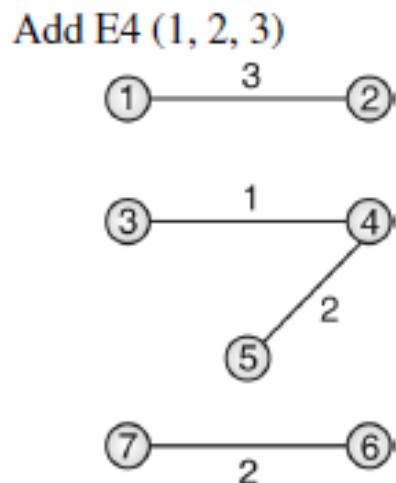
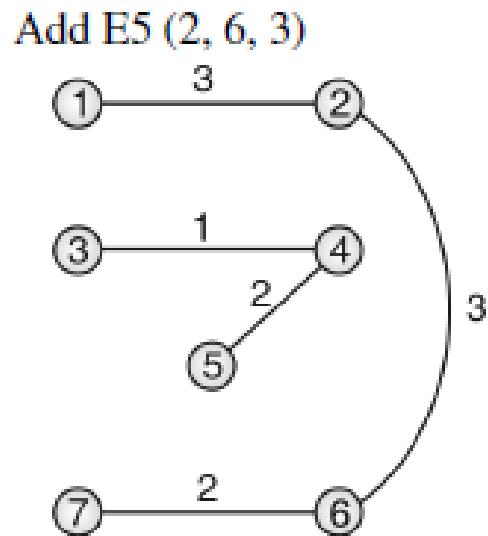


Add E4 (1, 2, 3)



Kruskal's Algorithm: Example #1

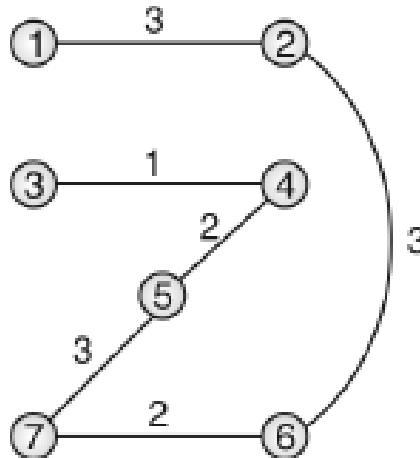
- **Step-II:** In the spanning tree, the edges from E1 to E10 are added sequentially to spanning tree. When cycle is formed by an edge, it is discarded.



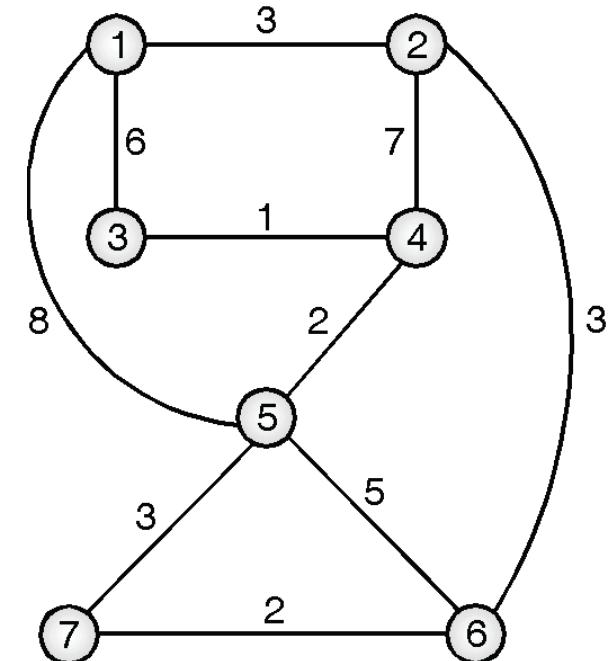
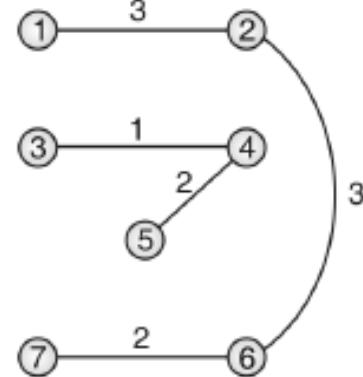
Kruskal's Algorithm: Example #1

- **Step-II:** In the spanning tree, the edges from E1 to E10 are added sequentially to spanning tree. When cycle is formed by an edge, it is discarded.

Add E6 (5, 7, 3)

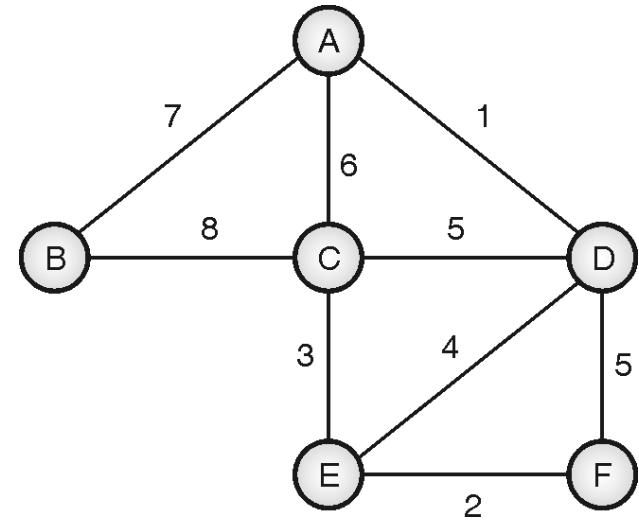
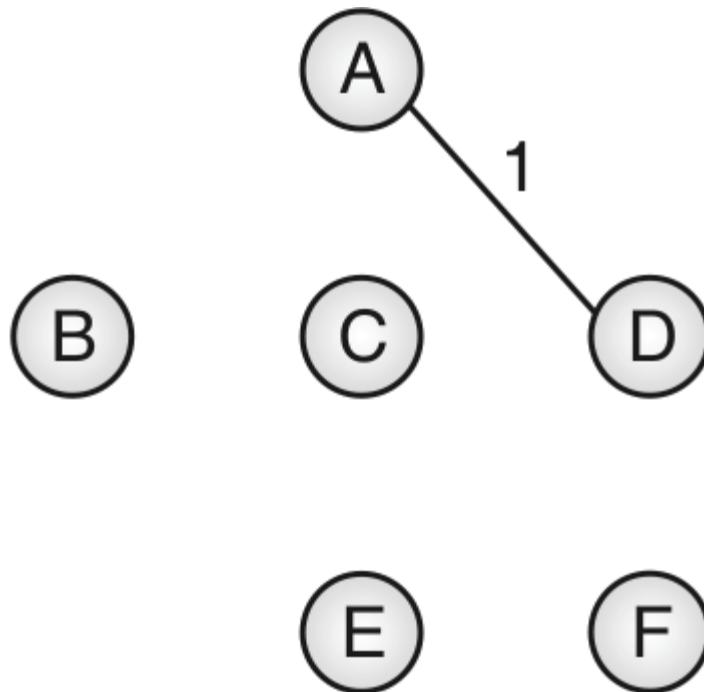


Add E5 (2, 6, 3)



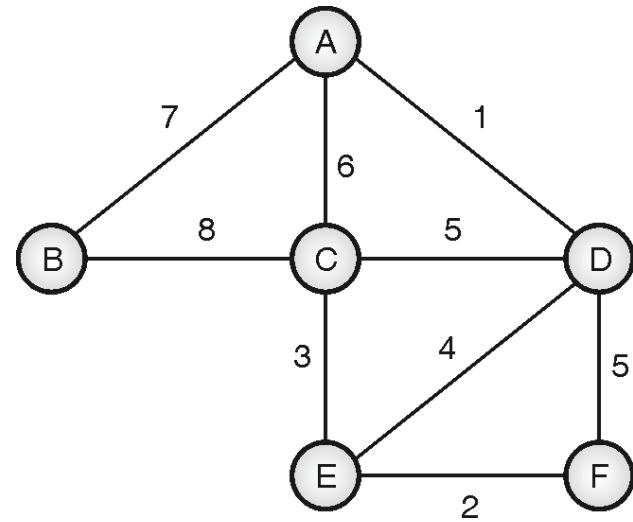
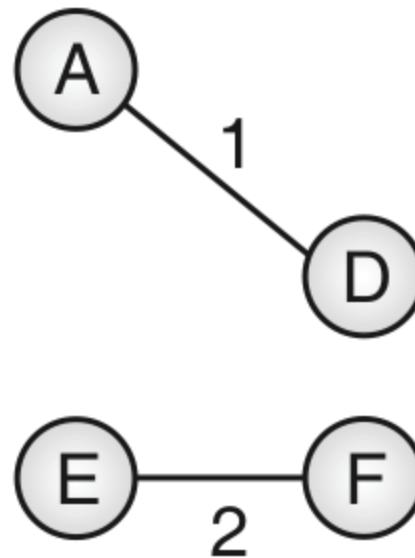
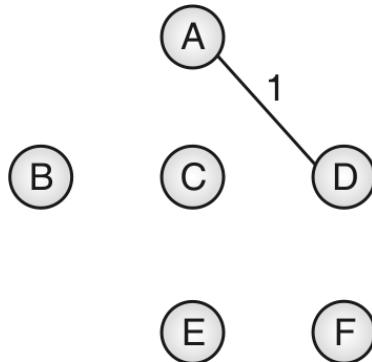
Kruskal's Algorithm: Example #2

- Find the minimum spanning tree for the given graph using Kruskal's algorithm.



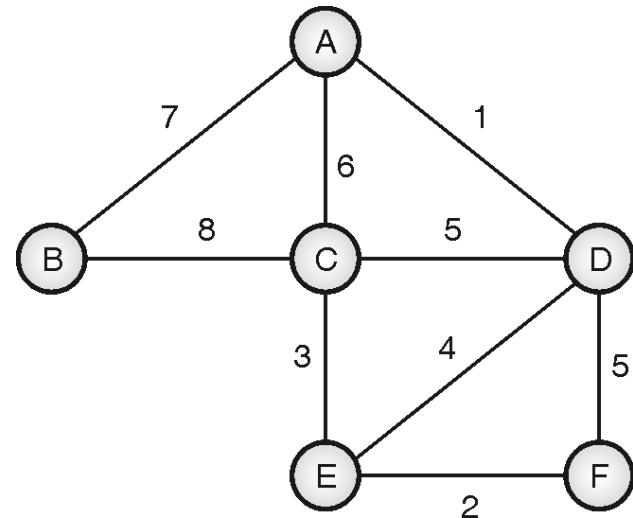
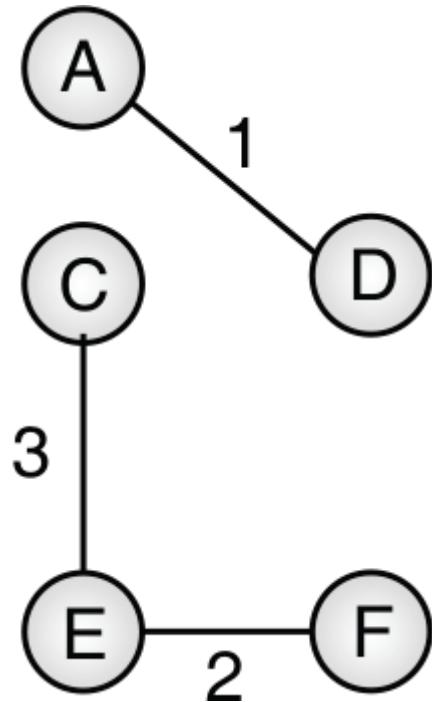
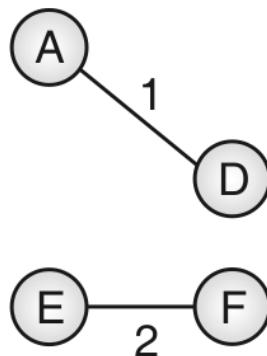
Kruskal's Algorithm: Example #2

- Find the minimum spanning tree for the given graph using Kruskal's algorithm.



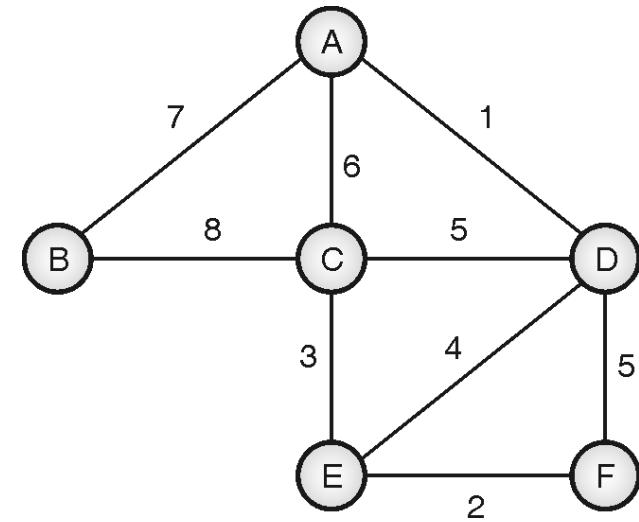
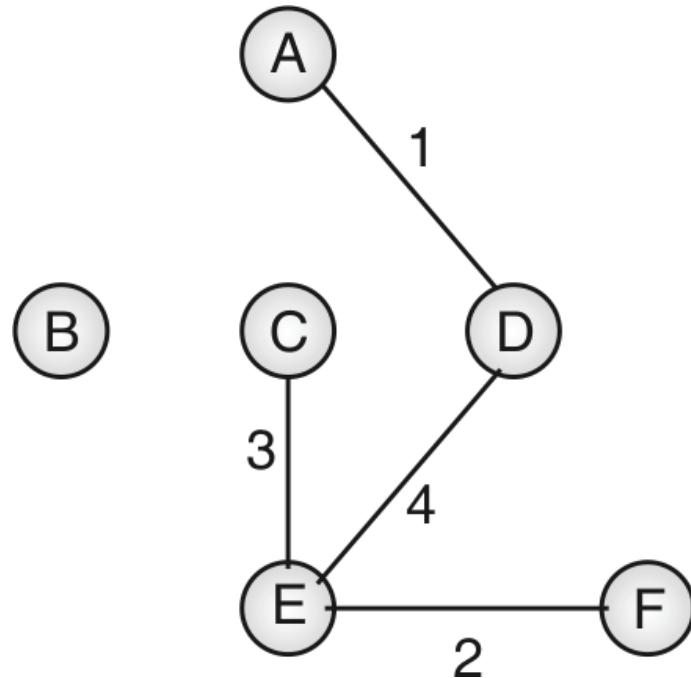
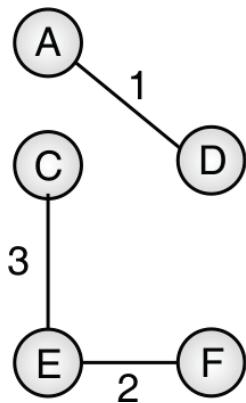
Kruskal's Algorithm: Example #2

- Find the minimum spanning tree for the given graph using Kruskal's algorithm.



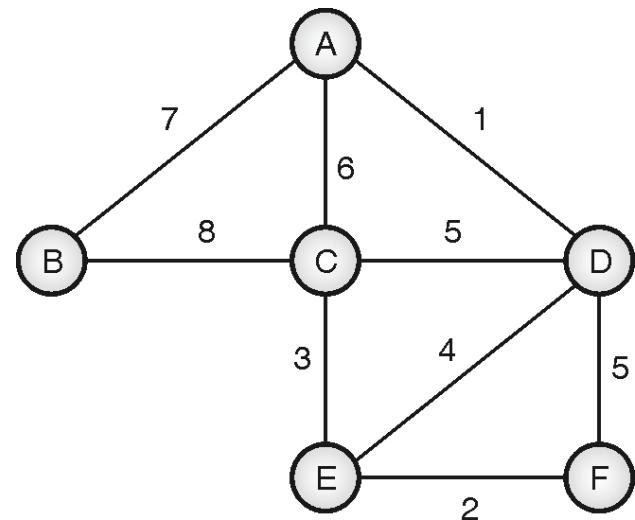
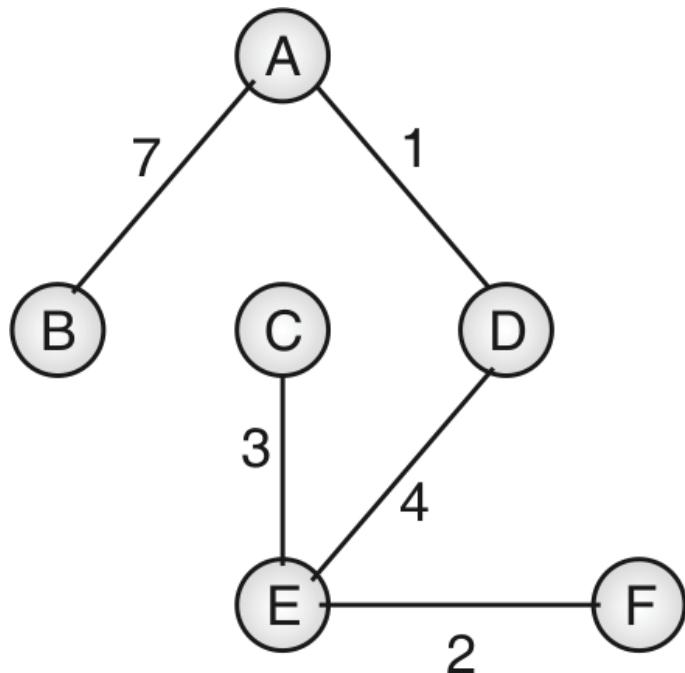
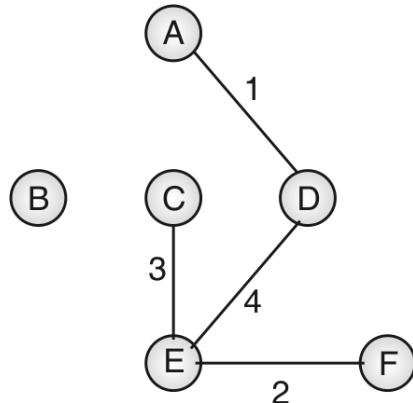
Kruskal's Algorithm: Example #2

- Find the minimum spanning tree for the given graph using Kruskal's algorithm.



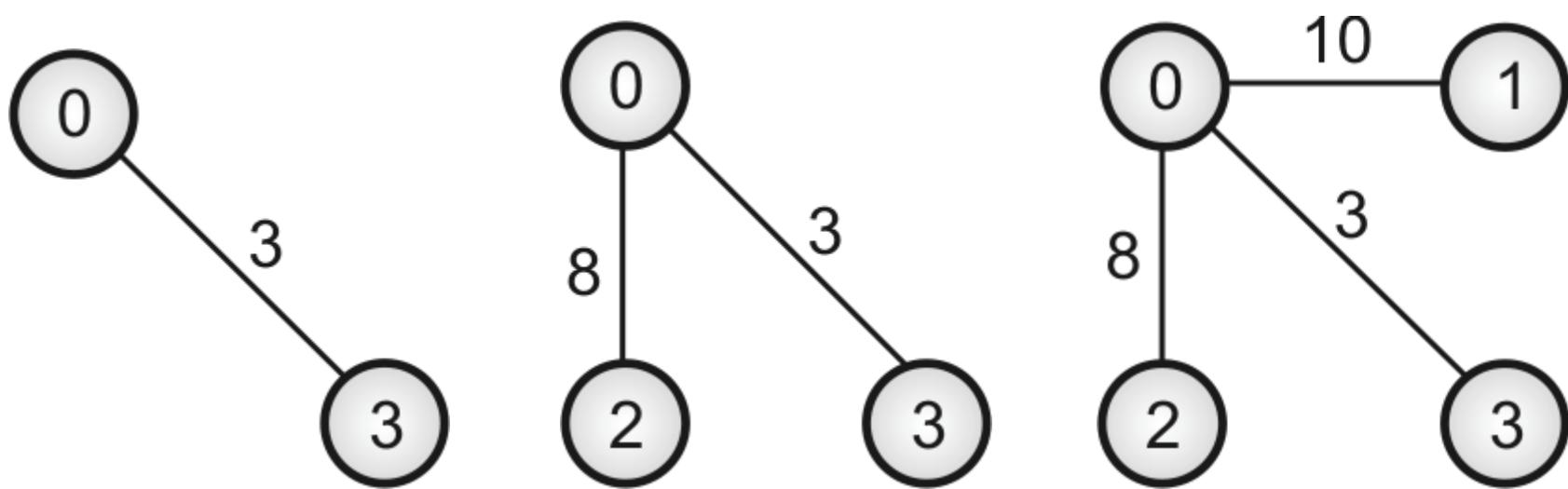
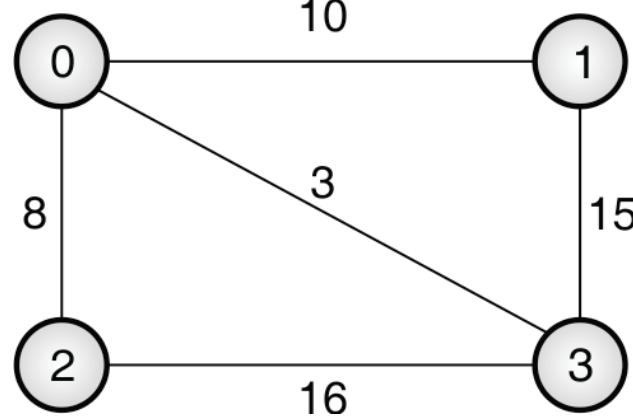
Kruskal's Algorithm: Example #2

- Find the minimum spanning tree for the given graph using Kruskal's algorithm.



Kruskal's Algorithm: Example #2

- Find minimal spanning tree of the following graph using Kruskal's algorithm.



Difference between Prim's and Kruskal's Algorithm

Parameter	Prim's Algorithm	Kruskal's Algorithm
Purpose	This algorithm is for obtaining minimum spanning tree by selecting the adjacent vertices of already selected vertices.	This algorithm is for obtaining minimum spanning tree but it is not necessary to choose adjacent vertices of already selected vertices.
Initiation	Prim's algorithm initializes with a node.	Kruskal's algorithm initiates with an edge.
Spanning	Prim's algorithms span from one node to another.	Kruskal's algorithm select the edges in a way that the position of the edge is not based on the last step.

Difference between Prim's and Kruskal's Algorithm

Parameter	Prim's Algorithm	Kruskal's Algorithm
Graph	In prim's algorithm, graph must be a connected graph.	Kruskal's algorithm's can function on disconnected graphs too.
Time complexity	Prim's algorithm has a time complexity of $O(V^2)$.	Kruskal's algorithm's time complexity is $O(\log V)$.

DIJKSTRA'S SINGLE SOURCE SHORTEST PATH

- Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.
- A single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.
- For a given source node in the graph, the algorithm finds the shortest path between that node and every other node.

DIJKSTRA'S SINGLE SOURCE SHORTEST PATH

- It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.
- For example, if the nodes of the graph represent cities and edges represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.
- As a result, the shortest path algorithm is widely used in network routing protocols,

Dijkstra's algorithm

- Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y.
- Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

Dijkstra's algorithm: Steps

- **Step 1 :** Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
- **Step 2 :** Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- **Step 3 :** For the current node, consider all of its neighbours and calculate their tentative distances.
- **Step 4 :** Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

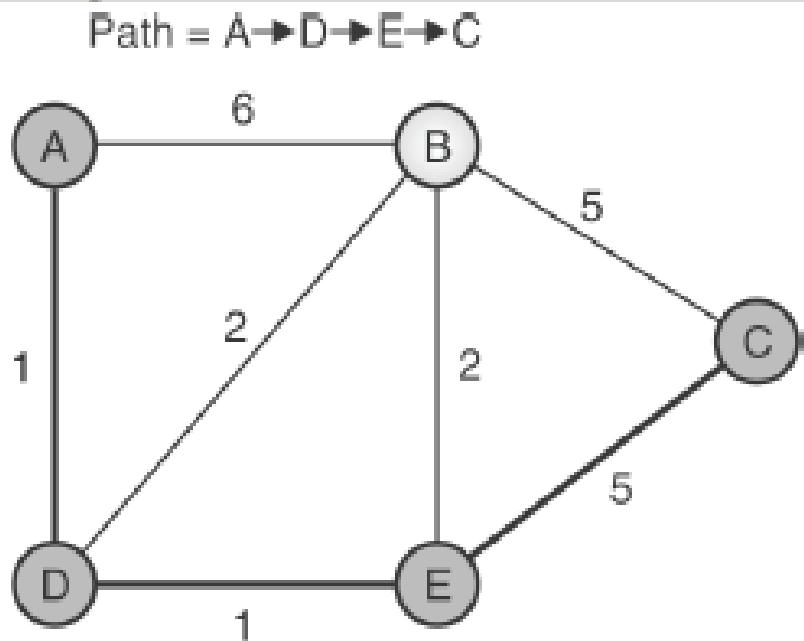
Dijkstra's algorithm: Steps

- **Step 4 :** Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
 - For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has length 2, then the distance to B (through A) will be $6 + 2 = 8$.
 - If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
- **Step 5:** When we are done considering all of the neighbours of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.

Dijkstra's algorithm: Steps

- **Step 7:** If the destination node has been marked visited ,Then algorithm has finished.
- **Step 8:** Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

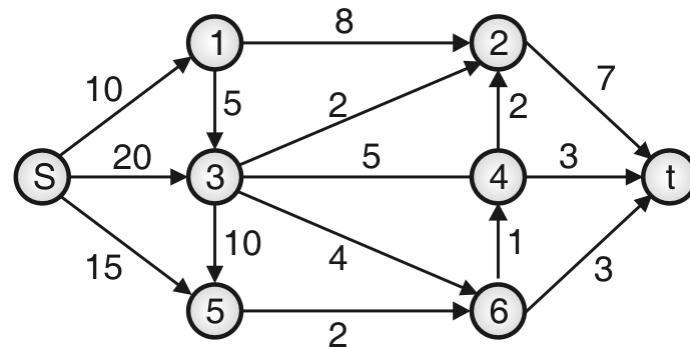
Dijkstra's algorithm: Steps



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Dijkstra's Algorithm: Example

- Find the shortest path using Dijkstra's Algorithm.



Visited minimum distance from S

(S)	1	2	3	4	5	6	t
0	10	∞	20	∞	15	∞	∞

↑
Node taken for expansion

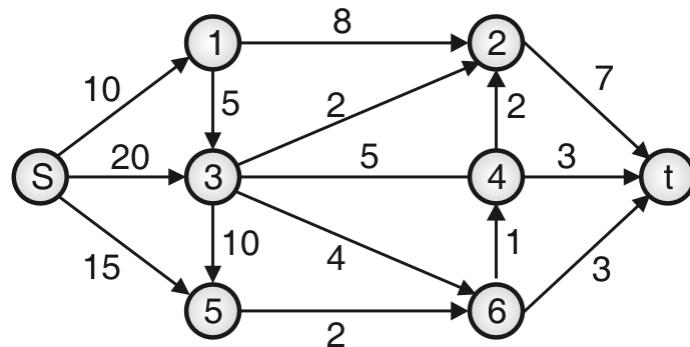
Visited minimum distance from S

(S)	(1)	2	3	4	5	6	t
0	10	18	15	∞	15	∞	∞

↑
Node taken for expansion

Dijkstra's Algorithm: Example

- Find the shortest path using Dijkstra's Algorithm.



Visited minimum distance from S

(S) 1 2 3 4 5 6 t

0 10 18 15 ∞ 15 ∞ ∞



Node taken for expansion

Visited minimum distance from S

(S) 1 2 3 4 5 6 t

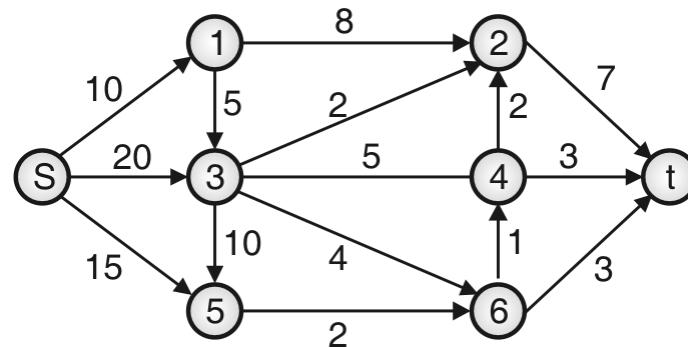
0 10 17 15 20 15 19 ∞



Node taken for expansion

Dijkstra's Algorithm: Example

- Find the shortest path using Dijkstra's Algorithm.



Visited minimum distance from S

(S)	(1)	2	(3)	4	5	6	t
0	10	17	15	20	15	19	∞

↑
Node taken for expansion

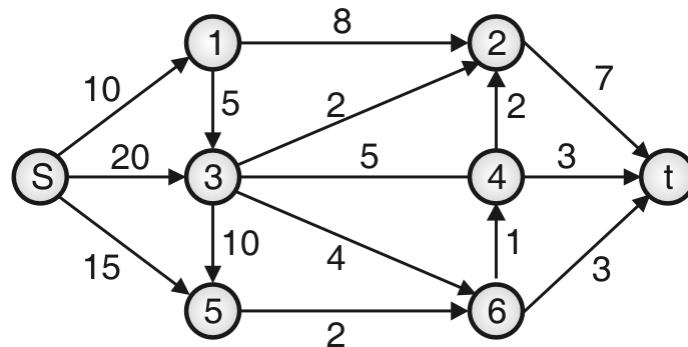
Visited minimum distance from S

(S)	(1)	2	(3)	4	(5)	6	t
0	10	17	15	20	15	17	∞

↑
Node taken for expansion

Dijkstra's Algorithm: Example

- Find the shortest path using Dijkstra's Algorithm.



Visited minimum distance from S



0 10 15 20 15 17 ∞

↑
Node taken for expansion

Visited minimum distance from S

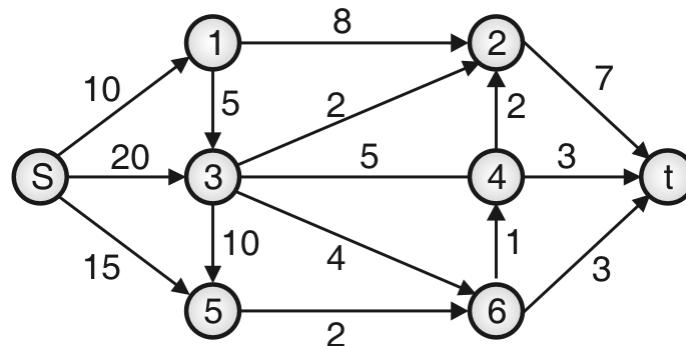


0 10 17 15 20 15 17 24

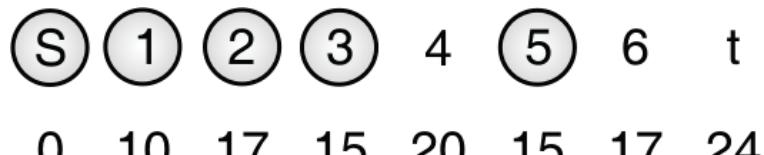
↑
Node taken for expansion

Dijkstra's Algorithm: Example

- Find the shortest path using Dijkstra's Algorithm.

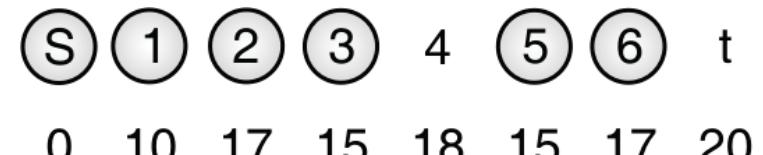


Visited minimum distance from S



Node taken for expansion

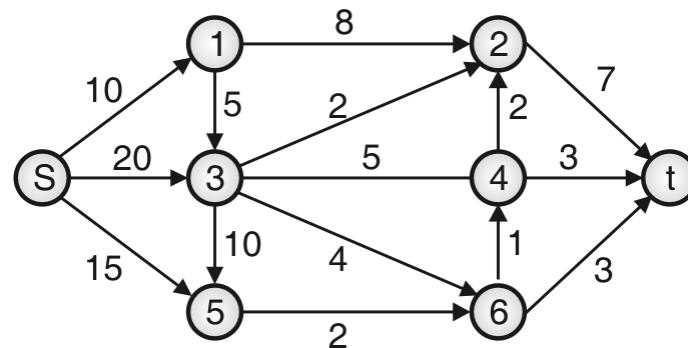
Visited minimum distance from S



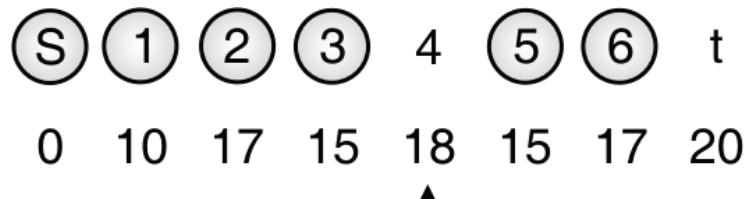
Node taken for expansion

Dijkstra's Algorithm: Example

- Find the shortest path using Dijkstra's Algorithm.

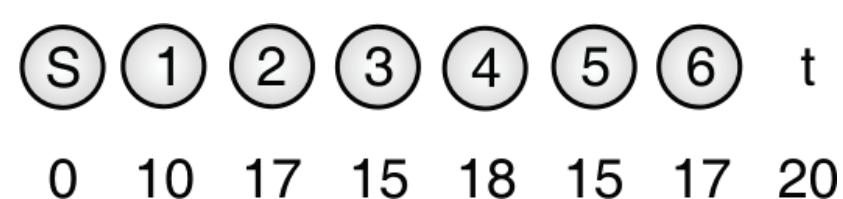


Visited minimum distance from S



↑
Node taken for expansion

Visited minimum distance from S



Shortest path from S to t is given by $S \rightarrow 5 \rightarrow 6 \rightarrow t$ with length = 20.

ALL PAIRS SHORTEST PATHS – FLYOD-WARSHALL ALGORITHM

- It is used to solve All Pairs Shortest Path Problem.
- It computes the shortest path between every pair of vertices of the given graph.
- Floyd Warshall Algorithm is an example of dynamic programming approach.

ALL PAIRS SHORTEST PATHS – FLYOD-WARSHALL ALGORITHM

- Floyd Warshall Algorithm has the following main advantages:
 - It is extremely simple.
 - It is easy to implement.

ALL PAIRS SHORTEST PATHS – FLYOD-WARSHALL ALGORITHM

- Floyd Warshall Algorithm:

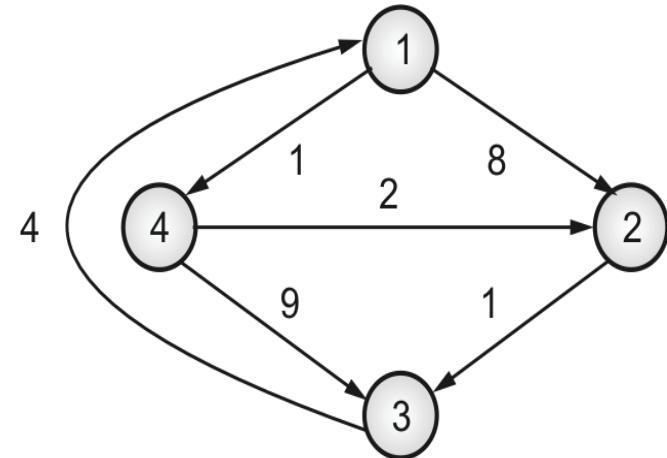
Let dist be a $|V| \times |V|$ array of minimum distances initialized to ∞ (infinity)

```
for each edge  $(u, v)$  do
     $\text{dist}[u][v] \leftarrow w(u, v)$  // The weight of the edge  $(u, v)$ 
for each vertex  $v$  do
     $\text{dist}[v][v] \leftarrow 0$ 
for  $k$  from 1 to  $|V|$ 
    for  $i$  from 1 to  $|V|$ 
        for  $j$  from 1 to  $|V|$ 
            if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$ 
                 $\text{dist}[i][j] \leftarrow \text{dist}[i][k] + \text{dist}[k][j]$ 
            end if
```

FLYOD-WARSHALL ALGORITHM: Example

- Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.
- Step 1:

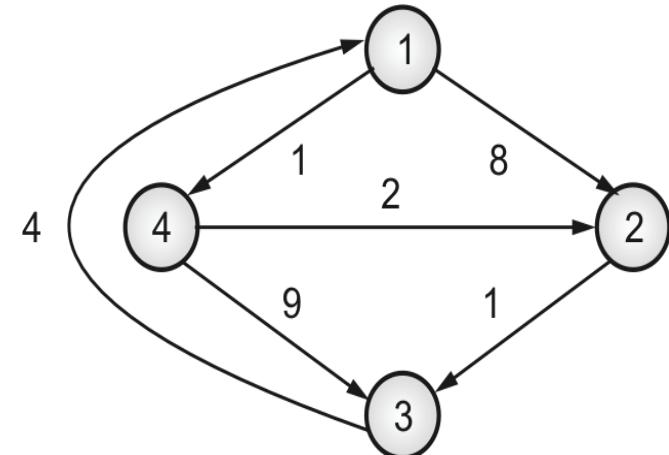
In the first step, Remove all the self loops as well as parallel edges (keeping the lowest weight edge) from the graph. In the given graph, there are neither self edges nor parallel edges.



FLYOD-WARSHALL ALGORITHM: Example

Write the starting distance matrix.

- It denotes the distance among each pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For the vertices which have a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞ .
- Initial distance matrix for the given graph is-



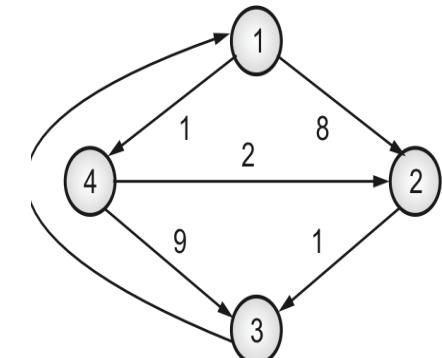
$$D_0 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix}$$

FLYOD-WARSHALL ALGORITHM: Example

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_0 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix}$$

$$D_1 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix}$$



FLYOD-WARSHALL ALGORITHM: Example

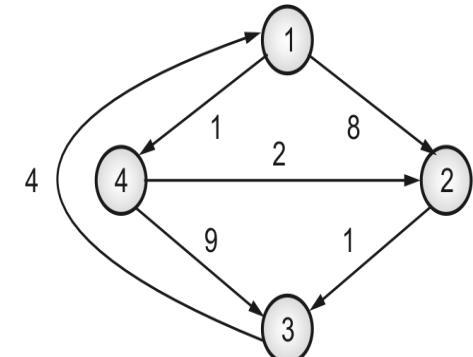
Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix}$$

$$D_3 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{bmatrix}$$

$$D_2 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{bmatrix}$$

$$D_4 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{bmatrix}$$



FLYOD-WARSHALL ALGORITHM: Example

In the given graph, there are 4 vertices.

- So, there will be total 4 matrices of order 4×4 in the solution excluding the initial distance matrix.
- Diagonal elements of each matrix will always be 0.

$$D_1 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & \boxed{0} & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix}$$

$$D_3 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & \boxed{0} & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{bmatrix}$$

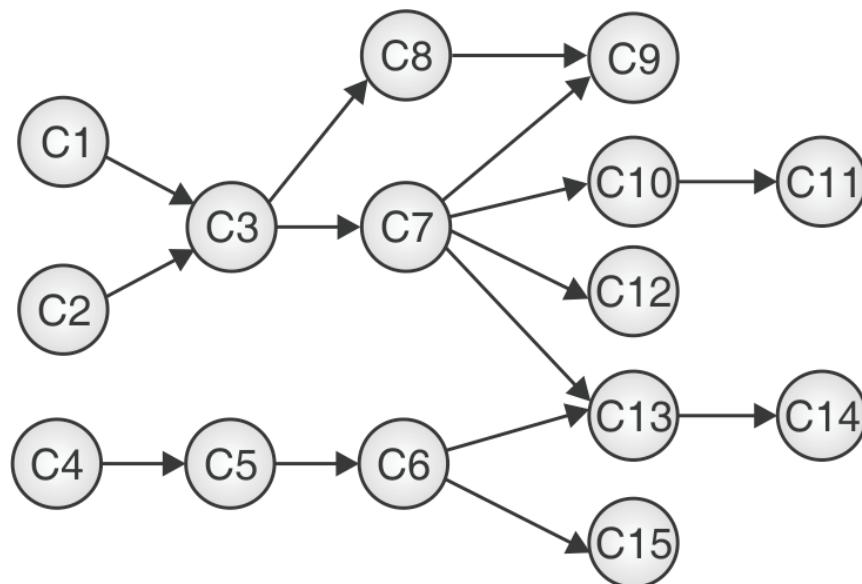
$$D_2 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & \boxed{0} & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{bmatrix}$$

$$D_4 = \begin{bmatrix} & 1 & 2 & 3 & 4 \\ 1 & \boxed{0} & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{bmatrix}$$

TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

AOV Network

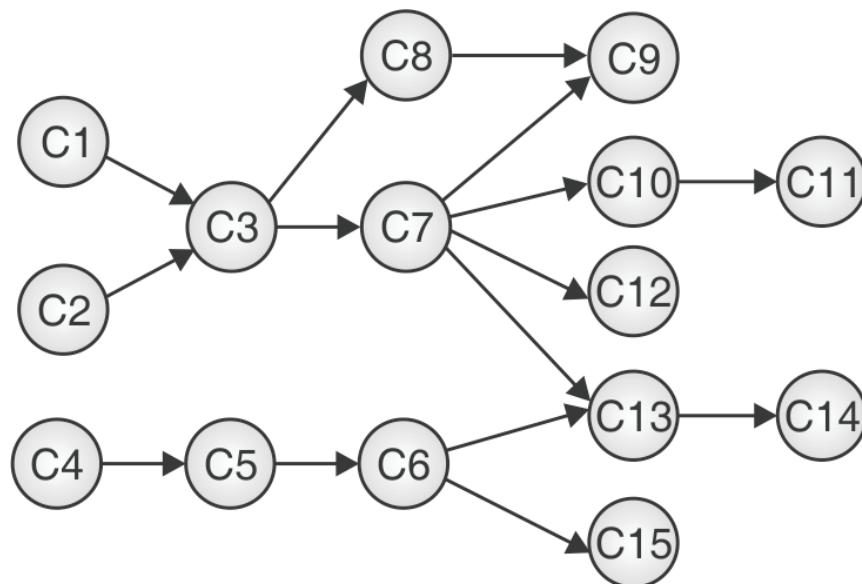
- An activity on vertex, or AOV network, is a directed graph G in which the vertices represent tasks or activities and the edges represent the precedence relation between tasks.
- Example : C3 is C1's successor, C1 is C3's predecessor.



TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

Topological Sorting

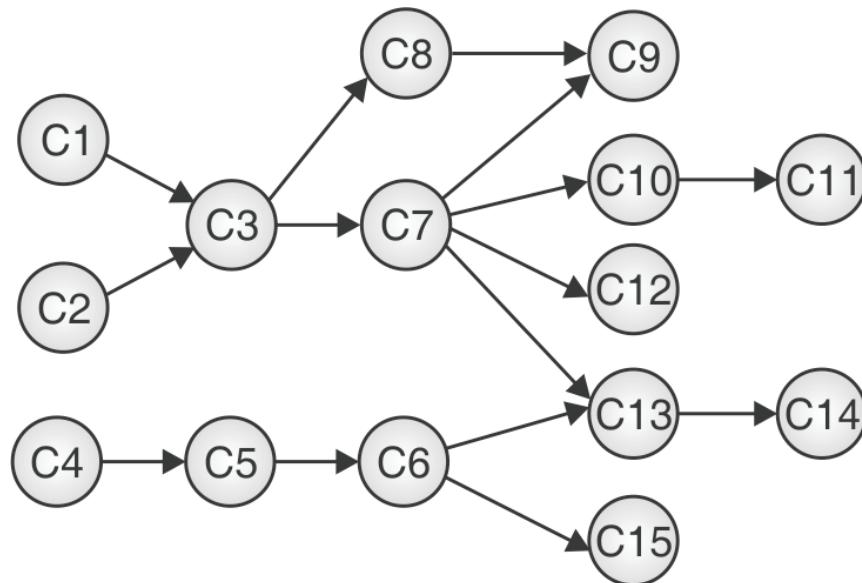
- A topological order is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network then i precedes j in the ordering.
- Example :



TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

Topological Sorting

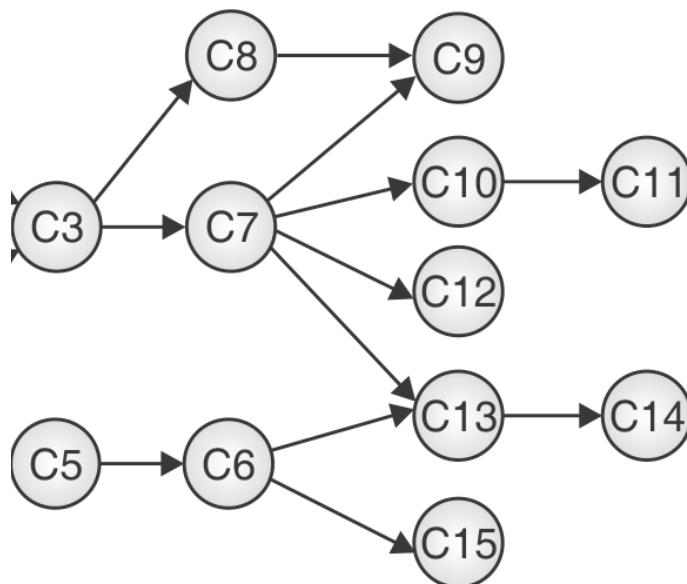
- A topological order is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network then i precedes j in the ordering.
- Example : C1 C2 C4



TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

Topological Sorting

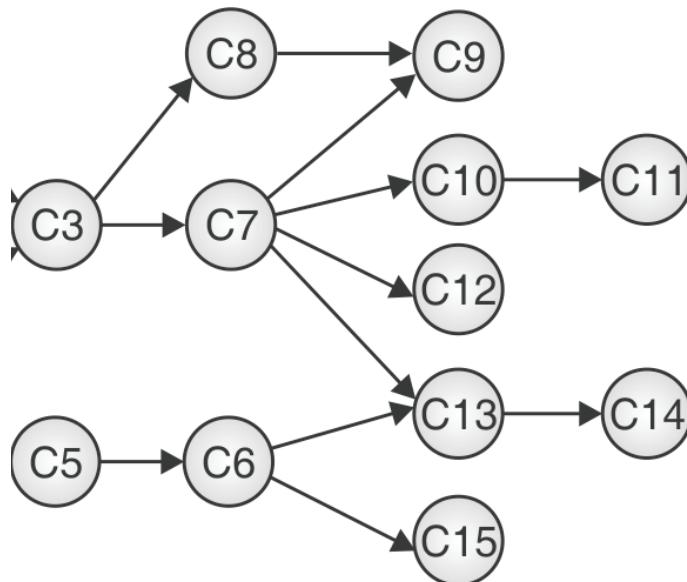
- A topological order is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network then i precedes j in the ordering.
- Example : C1 C2 C4



TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

Topological Sorting

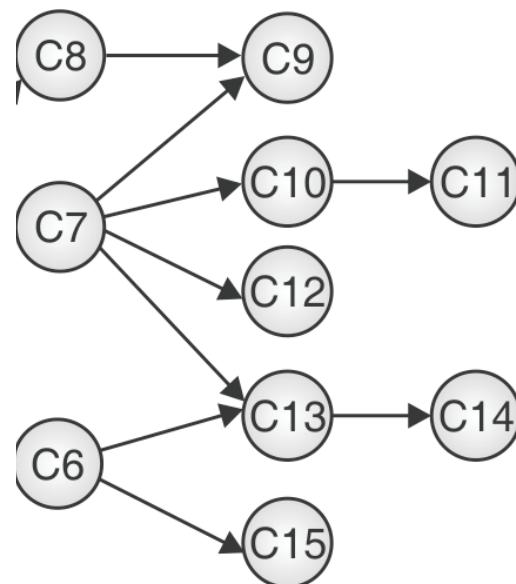
- A topological order is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network then i precedes j in the ordering.
- Example : C1 C2 C4 C5 C3



TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

Topological Sorting

- A topological order is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network then i precedes j in the ordering.
- Example : C1 C2 C4 C5 C3



TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

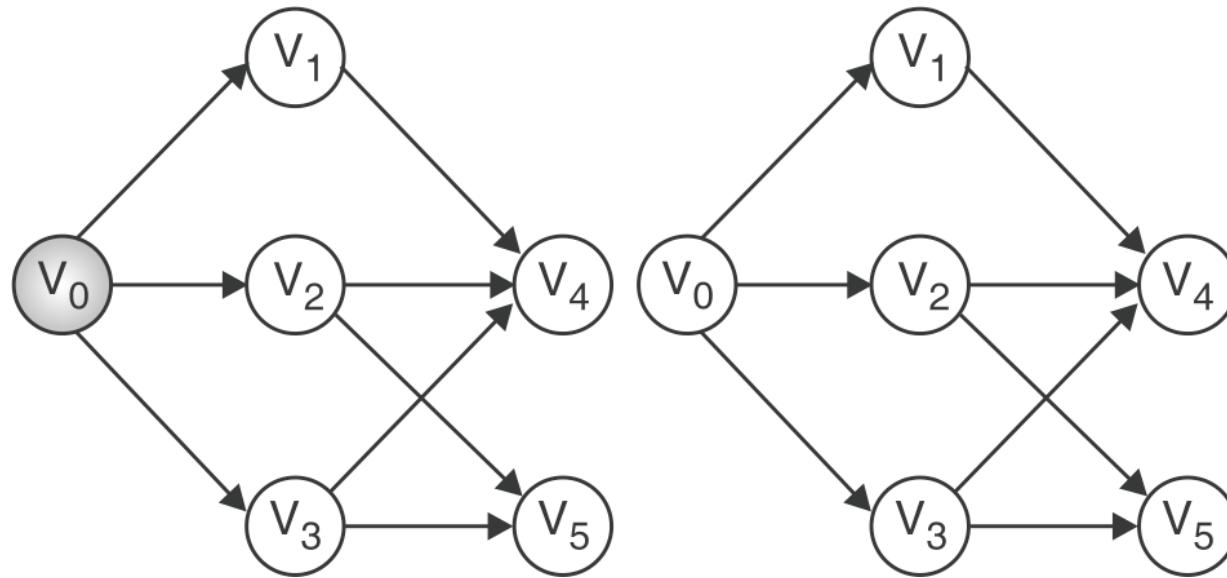
Topological Sorting

- Step 1 : Find a vertex v such that v has no predecessor, output it. Then delete it from network

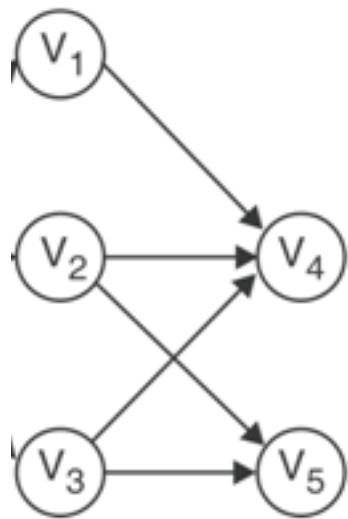
- Step 2 : Repeat this step until all vertices are deleted, Time complexity: $O(|V| + |E|)$

```
for (i = 0; i <n; i++)  
{  
    if every vertex has a predecessor  
    {  
        fprintf(stderr, "Network has a  
        cycle.\n ");  
        exit(1);  
    }  
    pick a vertex v that has no  
    predecessors;  
    output v;  
    delete v and all edges leading out  
    of v from the network;  
}
```

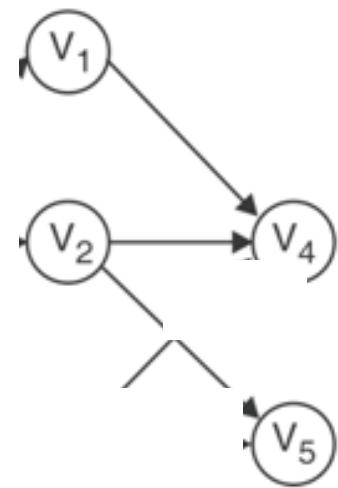
- v_0 has no predecessor, output it. Then delete it and three edges.



- v_0 has no predecessor, output it. Then delete it and three edges.

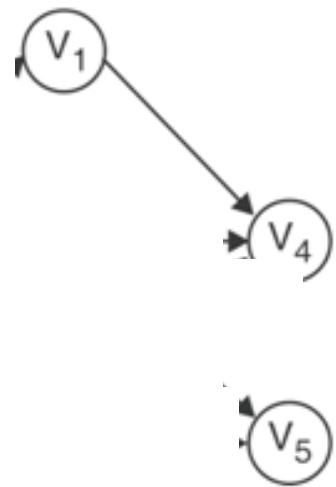
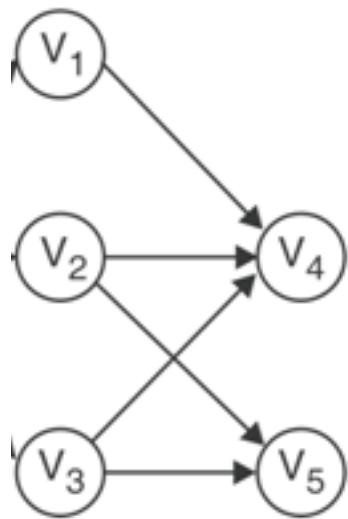


v_0



$v_0, v3$

- v_0 has no predecessor, output it. Then delete it and three edges.



$v_0, v_3, v_2,$

- v_0 has no predecessor, output it. Then delete it and three edges.



v_0, v_3, v_2, v_1

- v_0 has no predecessor, output it. Then delete it and three edges.

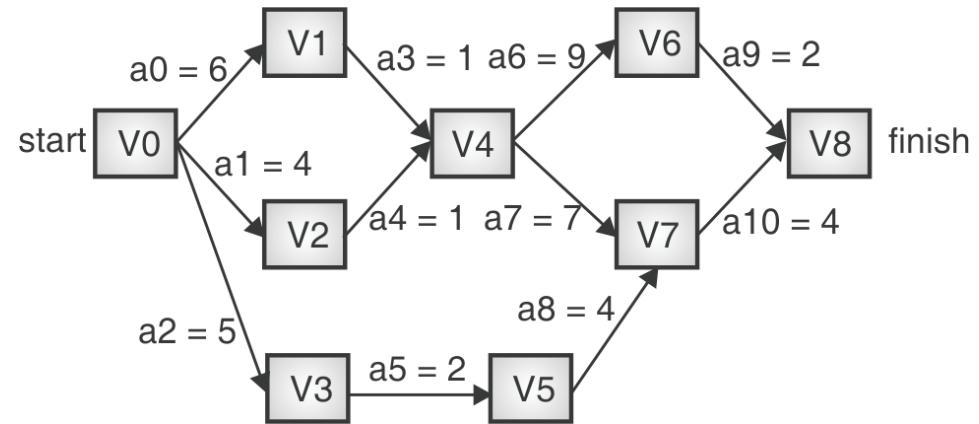


$v_0, v_3, v_2, v_1, v_4, v_5$

TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

AOE Network

- AOE network is an activity network closely related to the AOV network. The directed edges in the graph represent tasks or activities to be performed on a project.
- Directed edge : tasks or activities to be performed.
- Vertex : events which signal the completion of certain activities.
- Number : time required to perform the activity.
- Example : Activities: a_0, a_1, \dots Events : v_0, v_1, \dots

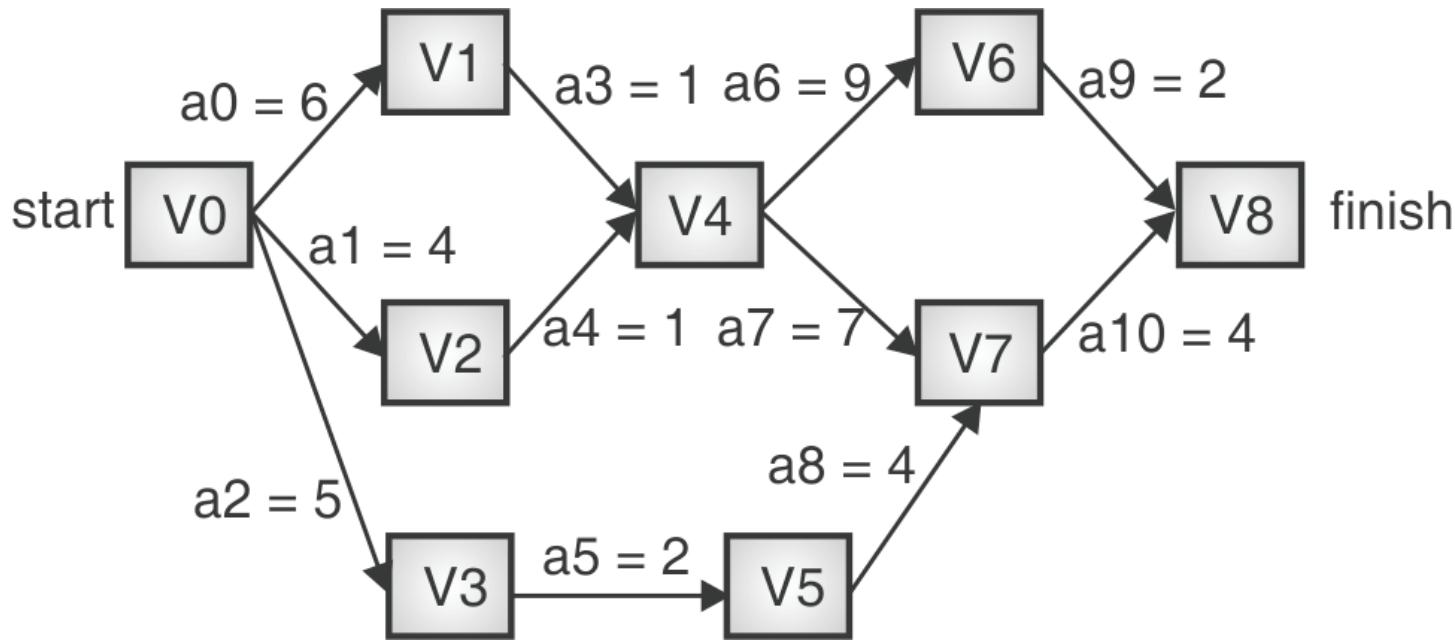


TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

Critical Path

- A critical path is a path that has the longest length.

$(v_0, v_1, v_4, v_7, v_8)$



TOPOLOGICAL SORT / TOPOLOGICAL ORDERING

Slack Time

- Slack is the amount of time that an activity can be delayed past its earliest start or earliest finish without delaying the project..

Symbol Table

- Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- Symbol table is used by both the analysis and the synthesis parts of a compiler.

Symbol Table

- Following attributes are used to store the information of data in the symbol table.
 - Symbol name
 - Symbol type
 - Symbol size
 - Symbol Scope
 - Symbol lifetime
 - Symbol address
 - Symbol value

Classification of Symbol Table

- Symbol tables are categorized as follows:
 - **Simple Symbol Table**
 - These tables have single scope i.e. all variables are global.
 - **Scoped Symbol Table**
 - These types of tables have many scopes.

Operations of Symbol Table

- A symbol table, may linear or hash, should offer the operations given below:
 - 1. Insert()
 - 2. lookup()

Operations of Symbol Table

- A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbol name, type>

- For example, if a symbol table has to store information about the following variable declaration:

int amount;

- then it should store the entry such as:

<amount, int>

Insert (amount, int)

- The attribute clause contains the entries related to the name.

Operations of Symbol Table

- A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:
<symbol name, type, attribute>
- For example, if a symbol table has to store information about the following variable declaration:
static int interest;
- then it should store the entry such as:
<interest, int, static>
- The attribute clause contains the entries related to the name.

Operations of Symbol Table

- Basic operations on Symbol table:
 - **Insert:** Insert a new entry for string s and attribute t.

Insert (s, t)

For example: **int a;** should be processed by the compiler as:

insert(a, int);

- **Lookup:** Returns the entry of index of string s;

lookup (s)

This method returns 0 zero if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Symbol Table: Example

- Consider the following program:

```
extern int total(int a)
void main( )
{
    double count(int b)
    {
        int b=5;
        float sum=0.0;
        for(int i=1; i<=b; i++)
            sum=sum + total((int) i);
        return sum;
    }
}
```

Symbol Table: Example

- Consider the following program:

```
extern int total(int a)
void main( )
{
    double count(int b)
    {
        int b=5;
        float sum=0.0;
        for(int i=1; i<=b; i++)
            sum=sum + total((int) i);
        return sum;
    }
}
```

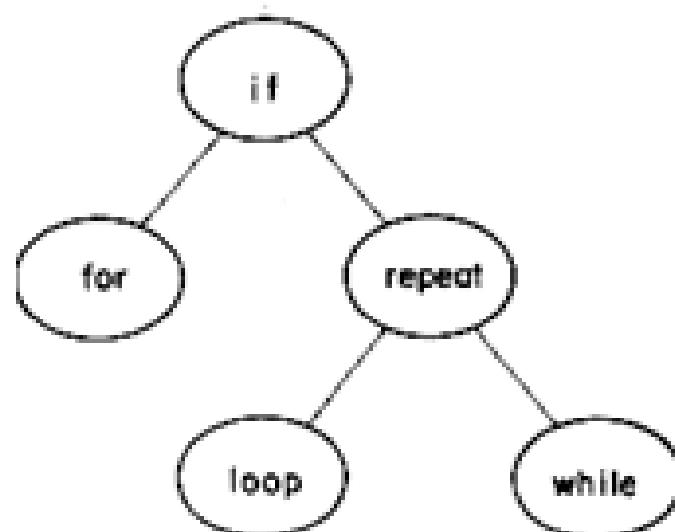
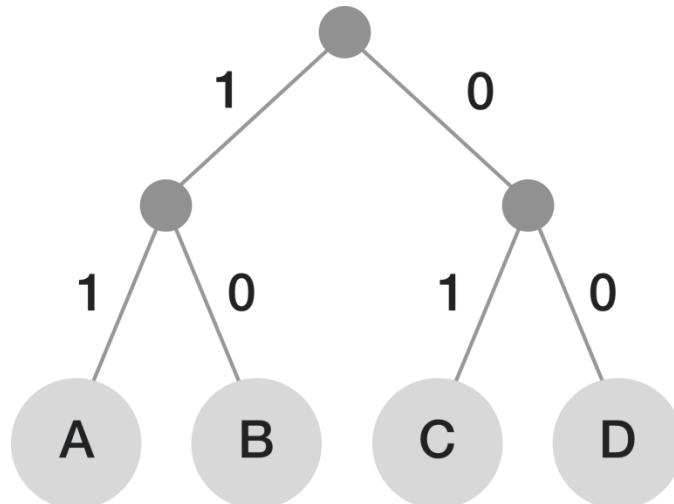
Symbol name	Type	Scope
total	Function int	extern
a	int	parameter
sum	float	Local scope
b	int	parameter
i	int	For loop stmt
count	function double	global

Symbol Table : Representation

- If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only.
- A symbol table can be implemented in one of the following ways:
 - Linear sorted or unsorted list / Array
 - Binary Search Tree
 - Optimal Binary Search Tree (OBST)
 - Hash table
- Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

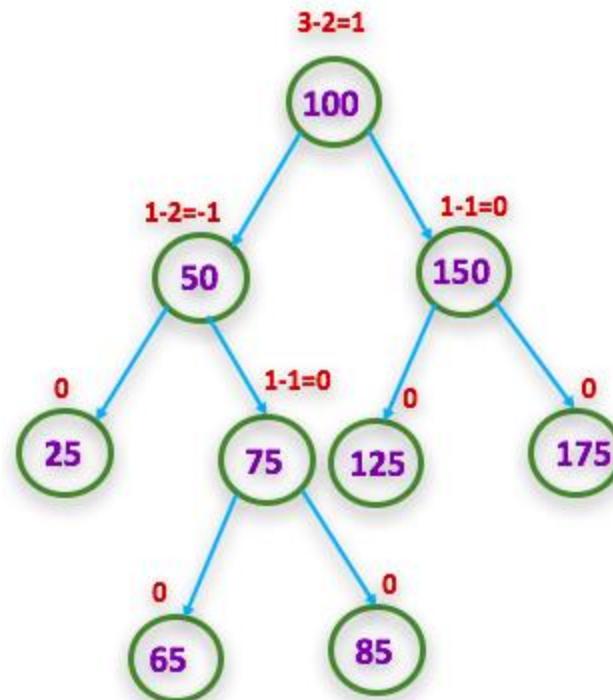
Static Tree Table

- Here, symbols are known in advance and no insertion and deletion are allowed.
- Example: Huffman Tree, OBST.
- Cost of searching a symbol occurring with higher frequency should be small.



Dynamic Tree Table

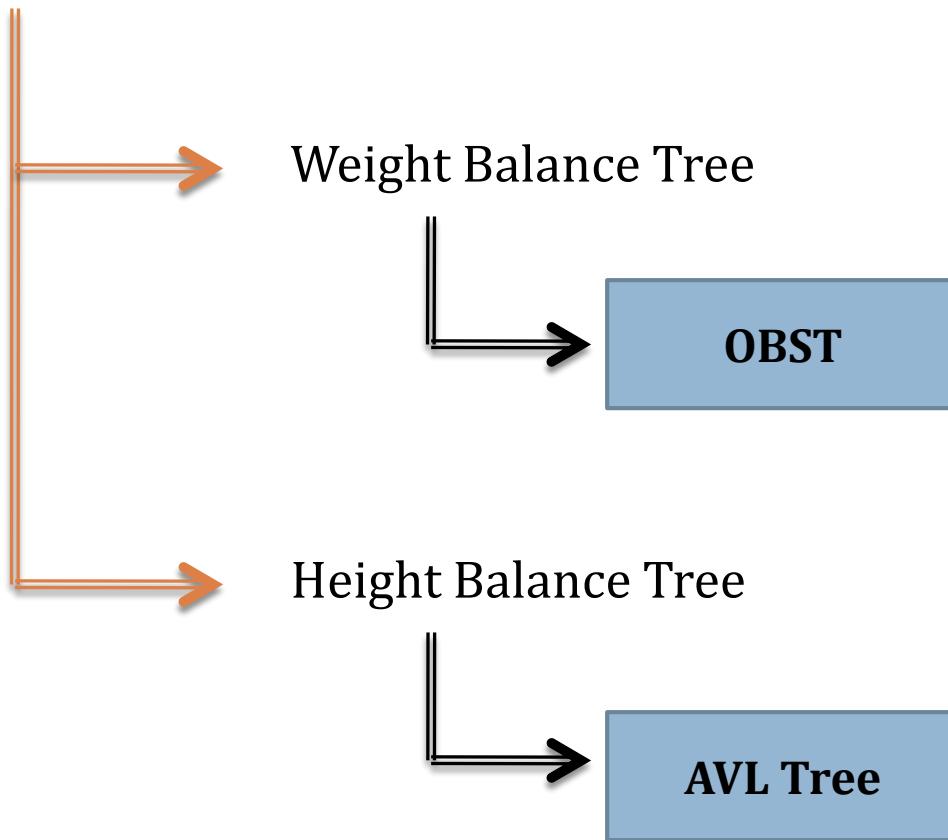
- Here, symbols are inserted a and when they come. They are deleted when they are no longer required.
- Example: AVL Tree



AVL Tree

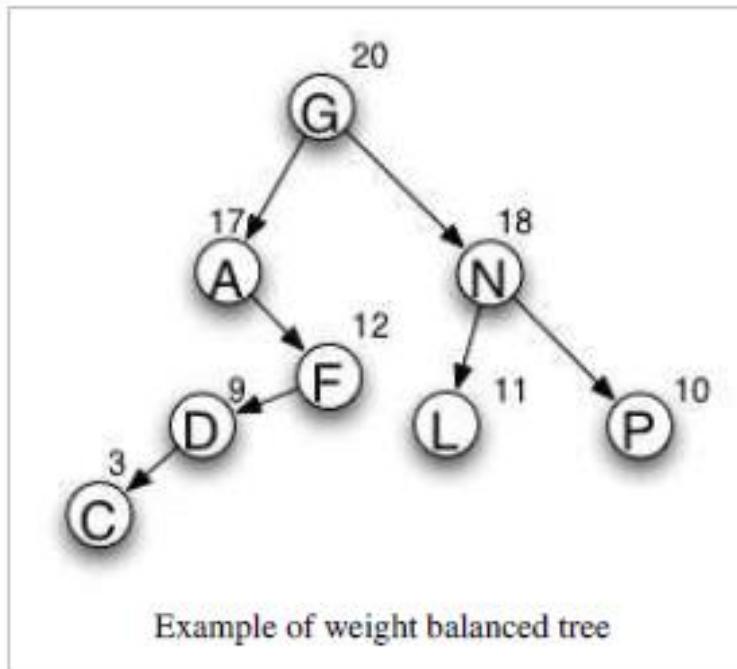
Balanced Trees

- Balanced Trees



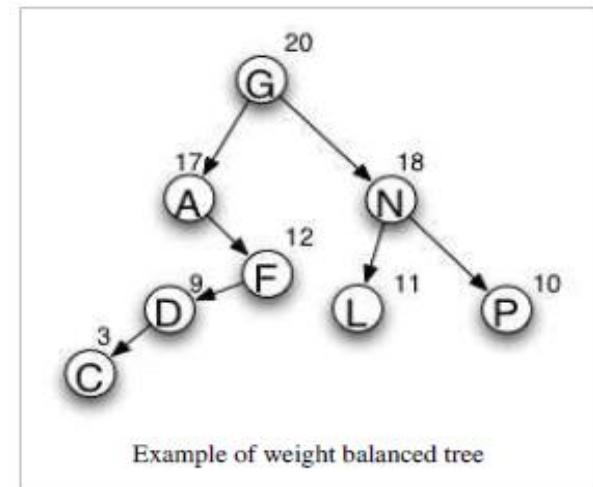
Weight-balanced tree

- A weight-balanced binary tree is a binary tree which is balanced based on knowledge of the probabilities of searching for each individual node.
- Within each subtree, the node with the highest weight appears at the root. This can result in more efficient searching performance.



Weight-balanced tree

- In the diagram to the right, the letters represent node values and the numbers represent node weights.
- Values are used to order the tree, as in a general binary search tree. The weight may be thought of as a probability or activity count associated with the node.
- In the diagram, the root is G because its weight is the greatest in the tree.
- The left subtree begins with A because, out of all nodes with values that come before G, A has the highest weight. Similarly, N is the highest-weighted node that comes after G.



Optimal Binary Search Tree

- In computer science, an optimal binary search tree , sometimes called a weight-balanced binary tree, is a binary search tree which provides the smallest possible search time for a given sequence of accesses.
- Optimal BSTs are generally divided into two types: static and dynamic.

Optimal Binary Search Tree

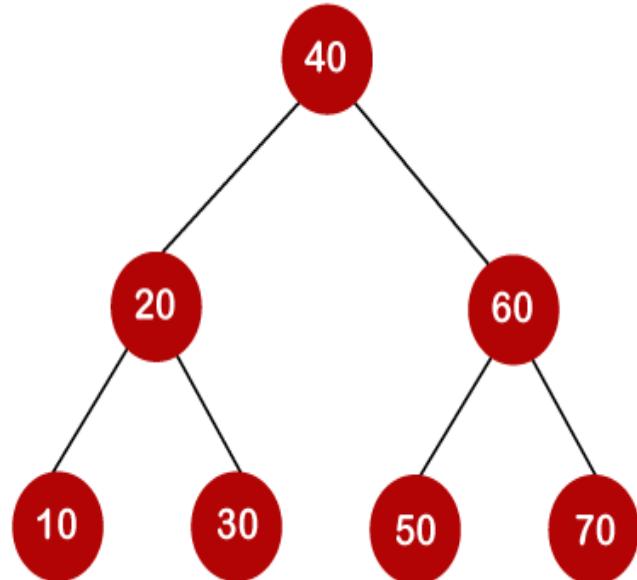
- As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.
- We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node.
- The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications.

Optimal Binary Search Tree

- The overall cost of searching a node should be less.
- The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST.
- There is one way that can reduce the cost of a binary search tree is known as an optimal binary search tree.

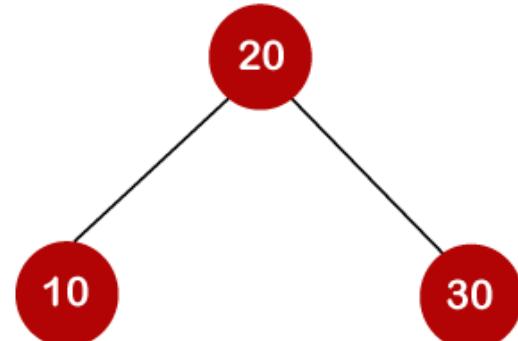
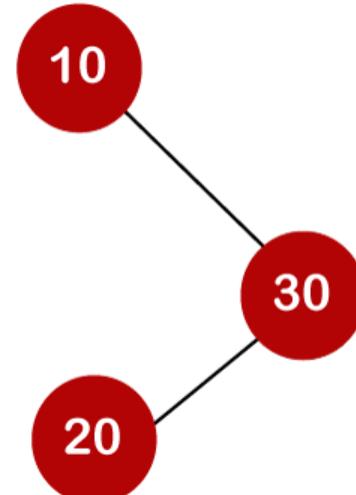
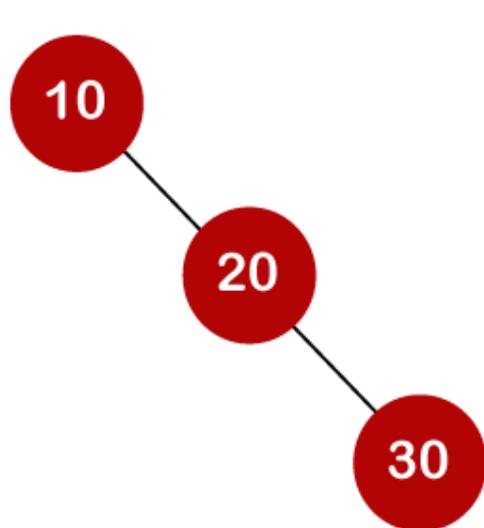
Optimal Binary Search Tree

- If the keys are 10, 20, 30, 40, 50, 60, 70
- In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node.
- The maximum time required to search a node is equal to the minimum height of the tree, equal to $\log n$.



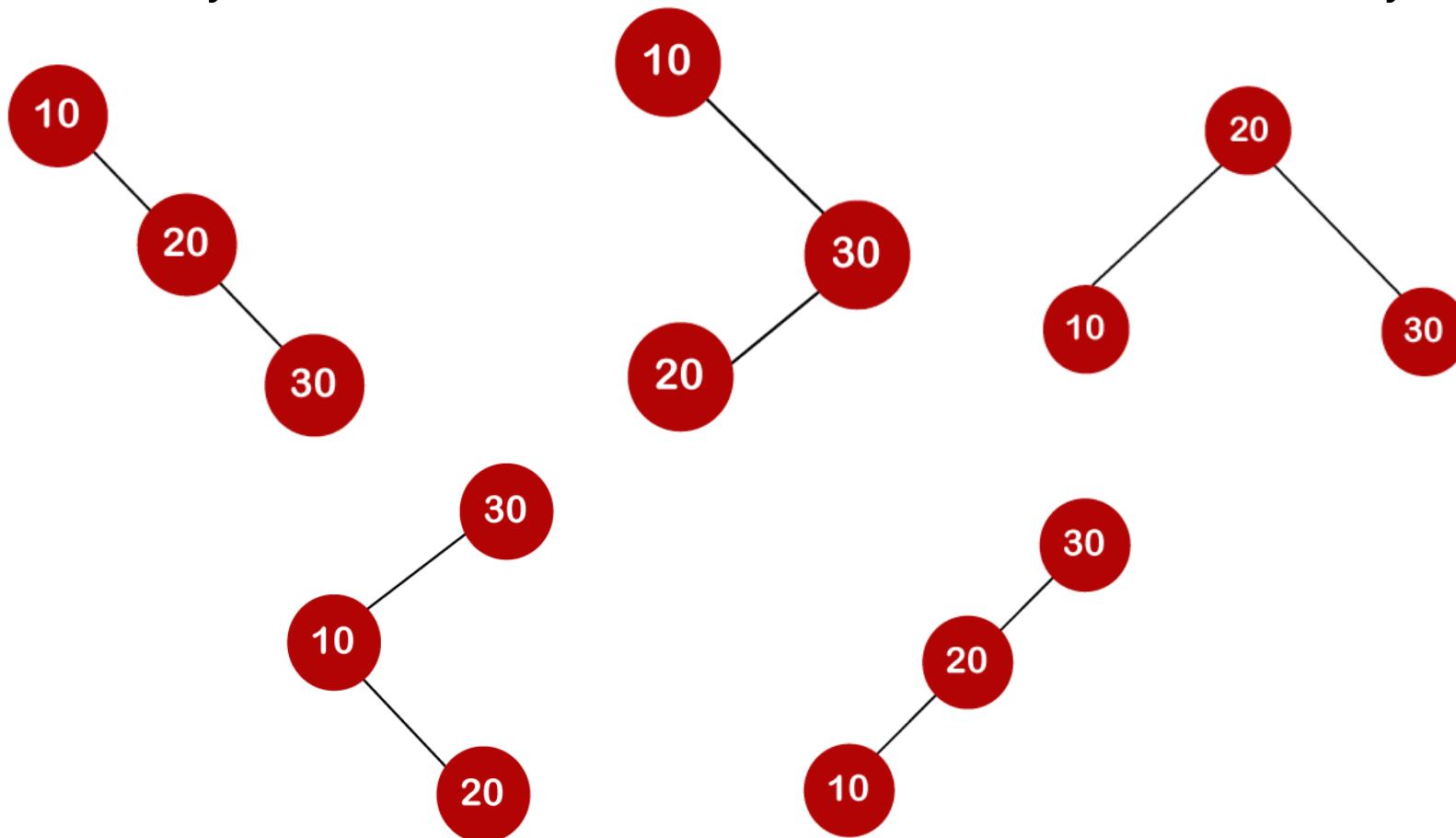
Optimal Binary Search Tree

- Now we will see how many binary search trees can be made from the given number of keys.
- For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

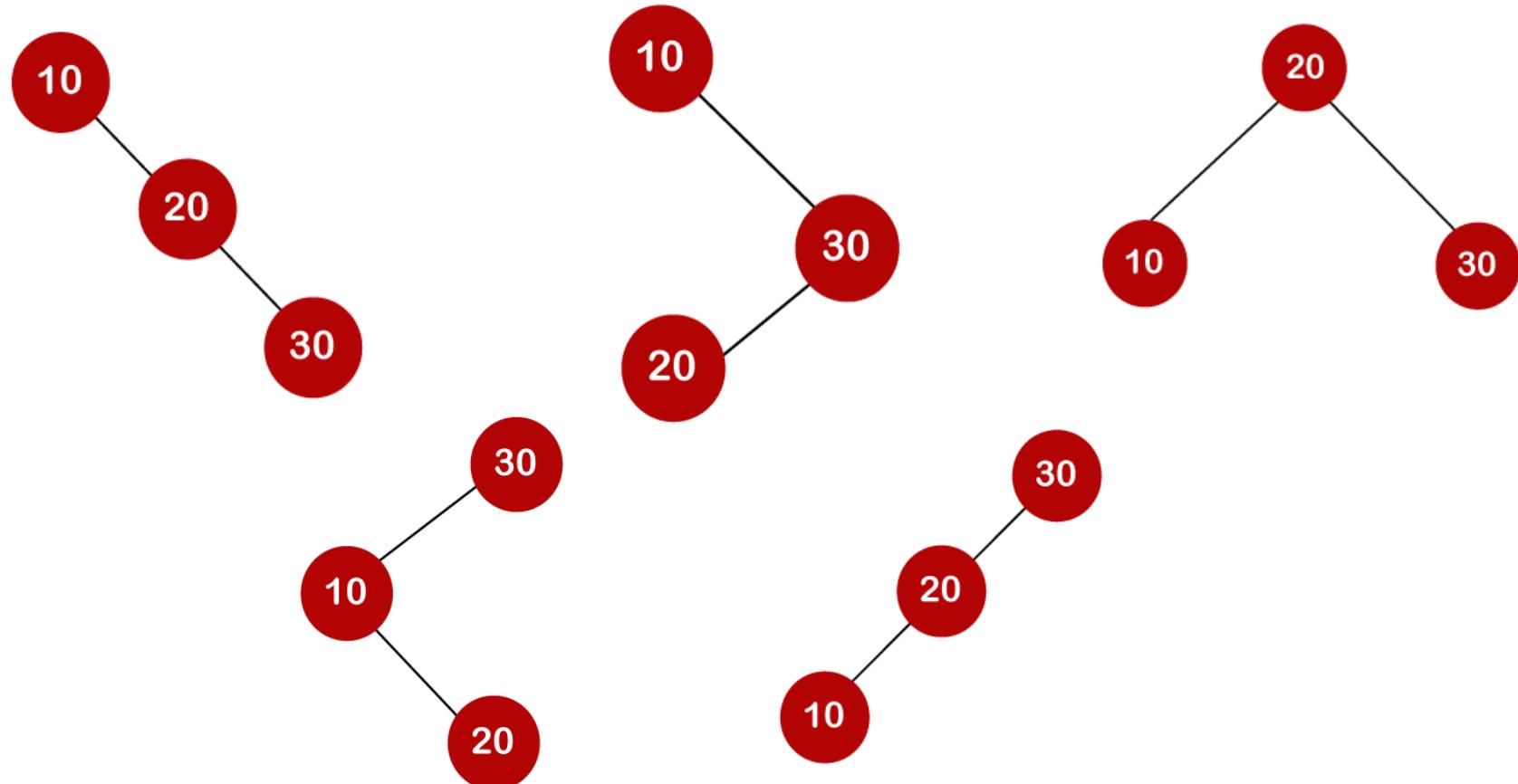


Optimal Binary Search Tree

- For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.



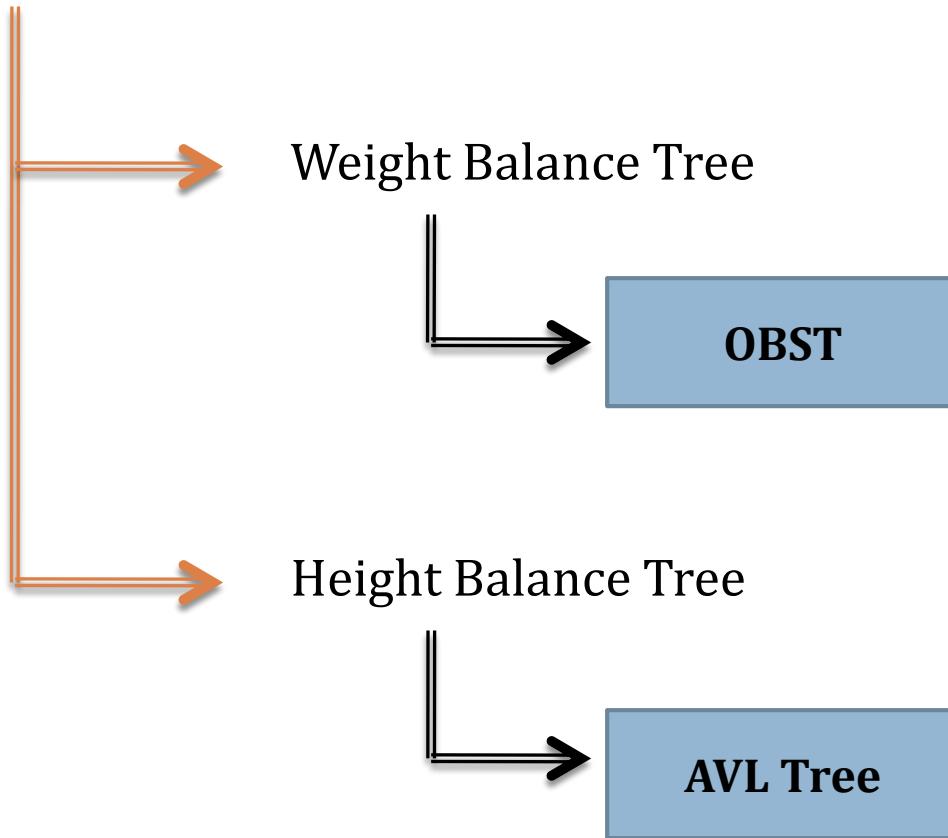
Optimal Binary Search Tree



The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree.

Balanced Trees

- Balanced Trees

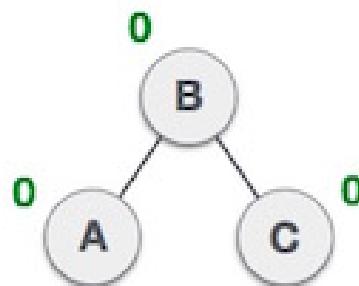


Height Balanced Tree: AVL Tree

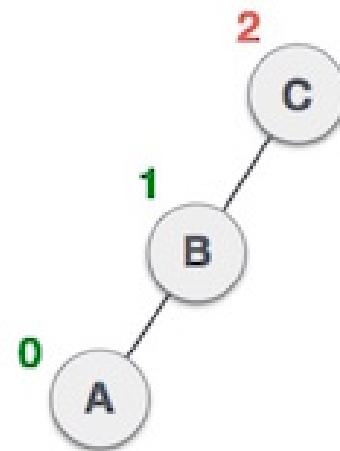
- Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree.
- AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1.
- This difference is called the Balance Factor.

AVL Tree

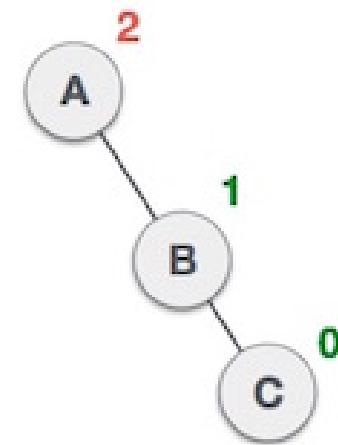
- Here we see that the first tree is balanced and the next two trees are not balanced –



Balanced



Not balanced



Not balanced

BalanceFactor = height(left-subtree) – height(right-subtree)

Operations on AVL Tree

- Operations on AVL tree-
 - Insertion
 - Deletion

Operations on AVL Tree

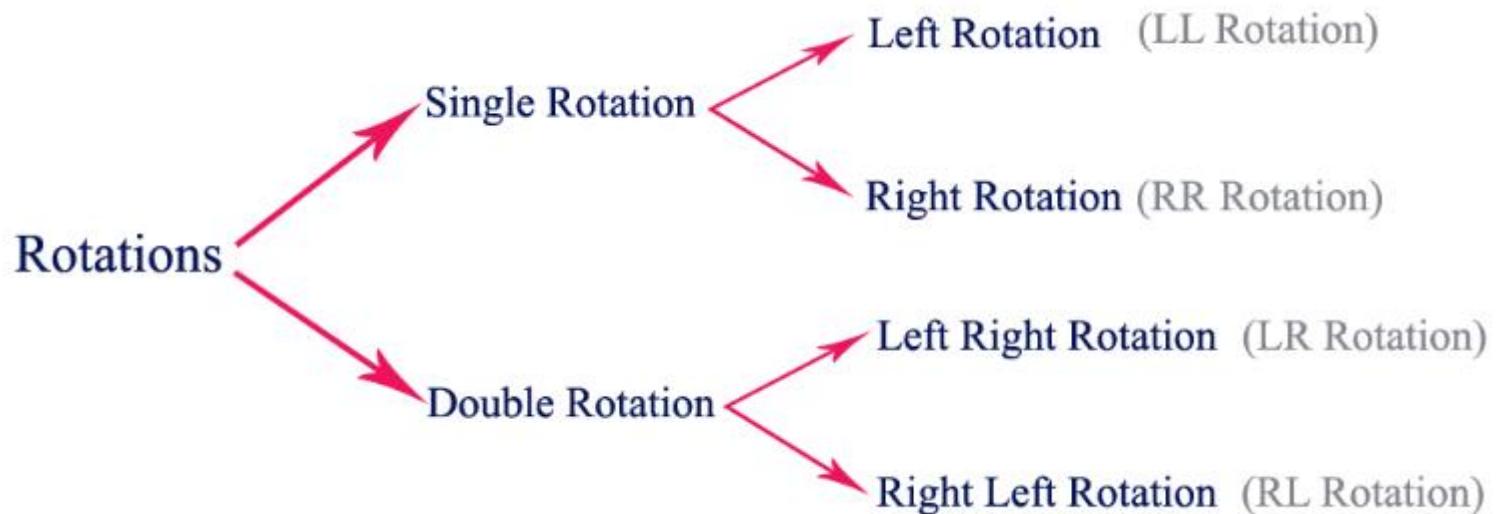
Insertion

- **Step 1:** First, insert a new element into the tree using BST's insertion logic.
- **Step 2:** Check the balance factor of each node.
- **Step 3:** if balance factor is 0 or 1 or -1 then the algorithm will proceed for next operation.
- **Step 4:** If balance factor is other than above then it is said to be imbalanced.

AVL Rotations

- If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.
- To balance itself, an AVL tree may perform the following four kinds of rotations –
 - Left rotation (LL Rotation)
 - Right rotation(RR Rotation)
 - Left-Right rotation (LR Rotation)
 - Right-Left rotation (RL Rotation)
- The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2.

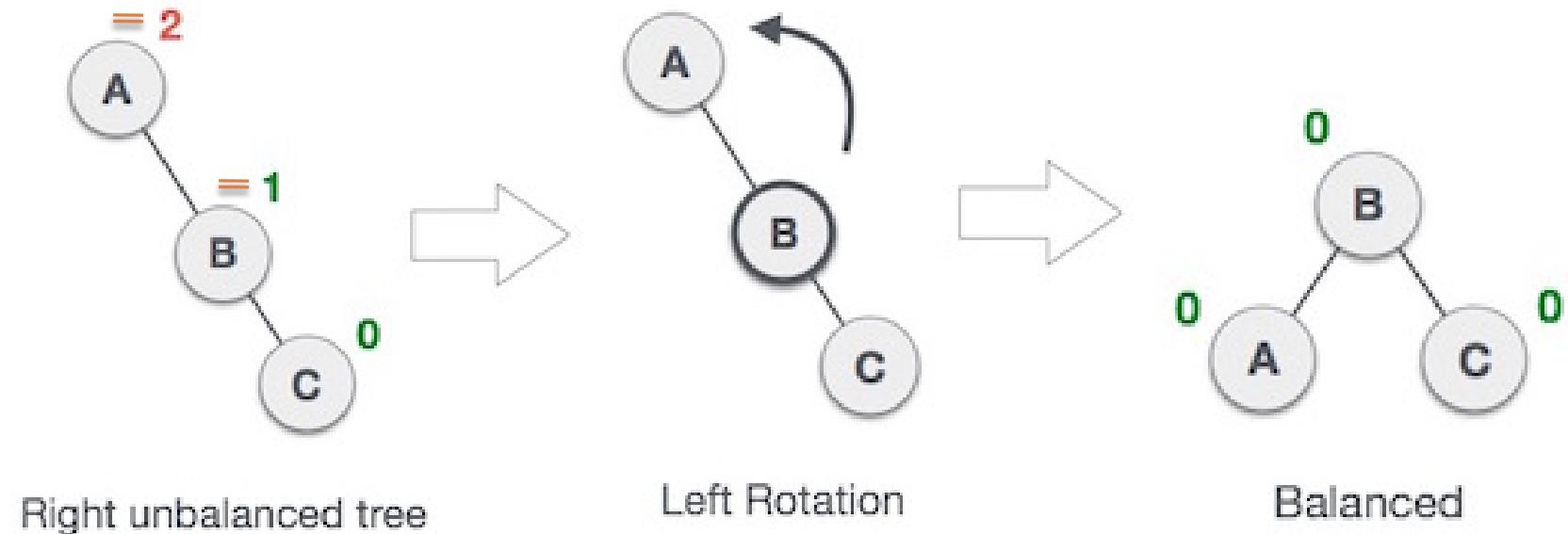
AVL Rotations



AVL Tree

Left rotation

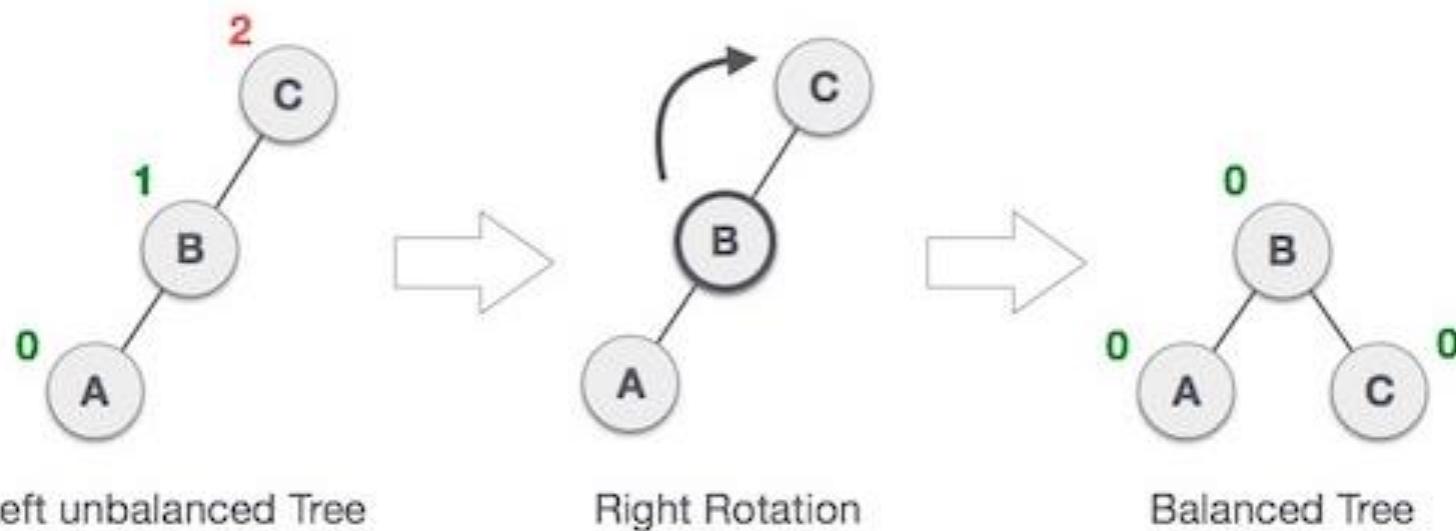
- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.
- Node A has become unbalanced as a node is inserted in the right subtree of A's right subtree.
- We perform the left rotation by making A the left-subtree of B.



AVL Tree

Right Rotation

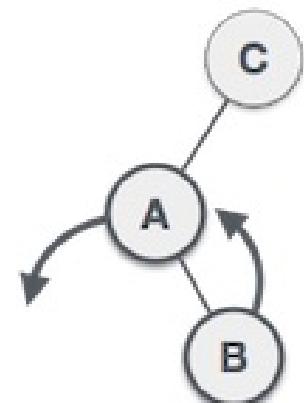
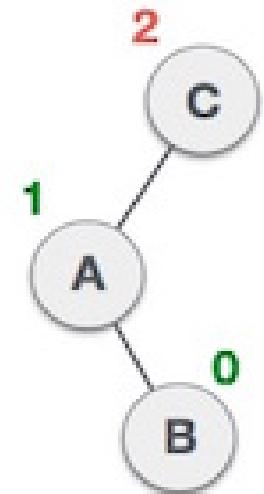
- AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.
- The unbalanced node becomes the right child of its left child by performing a right rotation.



AVL Tree

Left Right Rotation

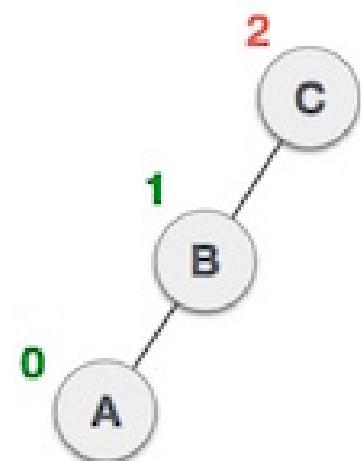
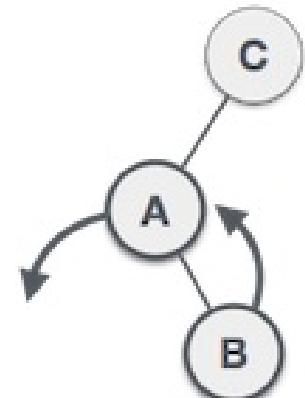
- A left-right rotation is a combination of left rotation followed by right rotation.
- A node has been inserted into the right subtree of the left subtree.
- This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
- We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.



AVL Tree

Left Right Rotation

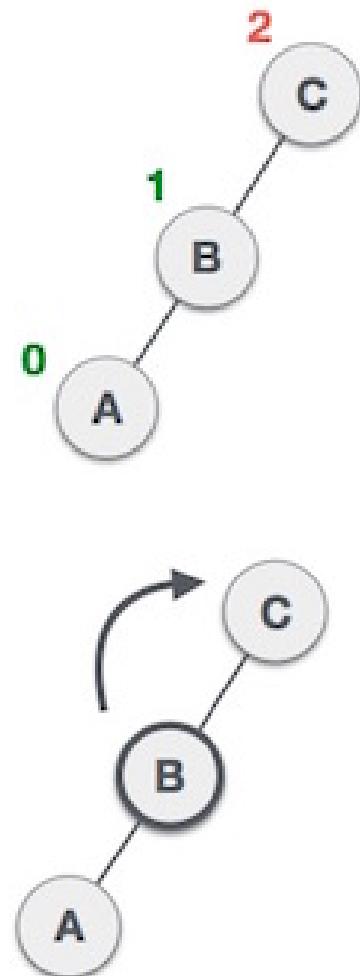
- A left-right rotation is a combination of left rotation followed by right rotation.
- Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.



AVL Tree

Left Right Rotation

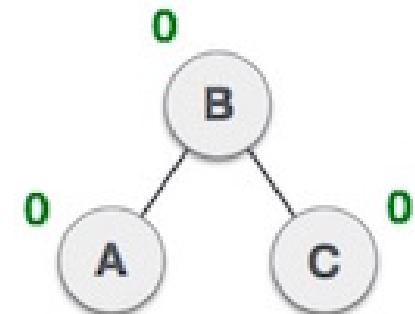
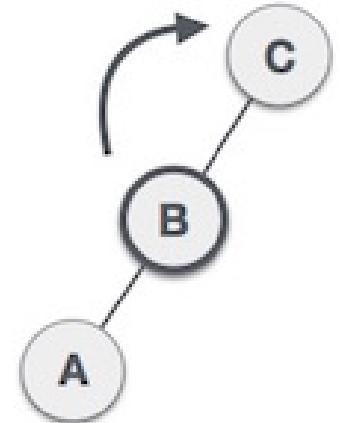
- A left-right rotation is a combination of left rotation followed by right rotation.
- Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
- We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.



AVL Tree

Left Right Rotation

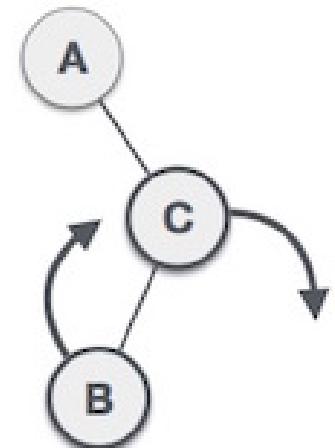
- A left-right rotation is a combination of left rotation followed by right rotation.
- Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
- We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.
- The tree is now balanced.



AVL Tree

Right-Left Rotation

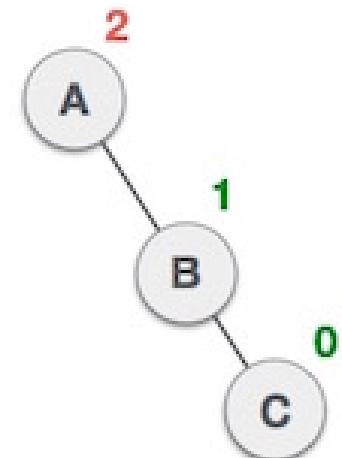
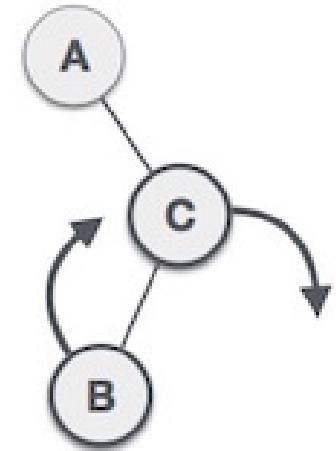
- It is a combination of right rotation followed by left rotation.
 - A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.
 - First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.



AVL Tree

Right-Left Rotation

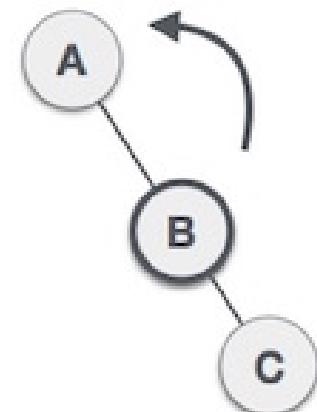
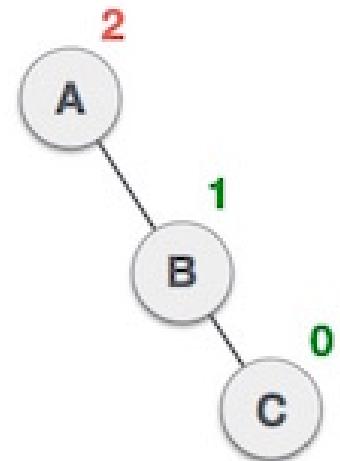
- It is a combination of right rotation followed by left rotation.
 - First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.
 - Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.



AVL Tree

Right-Left Rotation

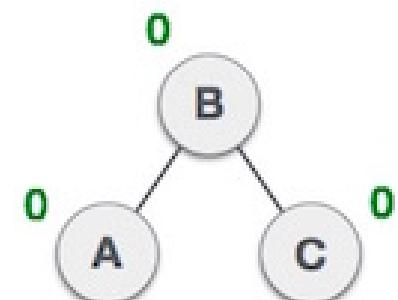
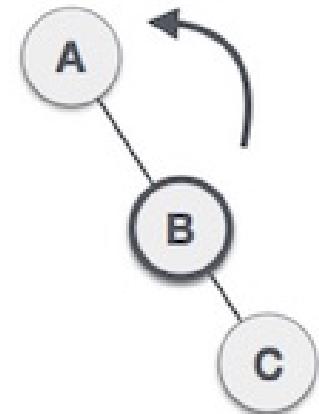
- It is a combination of right rotation followed by left rotation.
 - Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
 - A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.



AVL Tree

Right-Left Rotation

- It is a combination of right rotation followed by left rotation.
 - Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
 - A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.
 - The tree is now balanced.



AVL Tree: Example

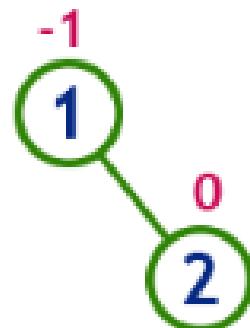
Construct an AVL Tree by inserting numbers from 1 to 8.

insert 1



Tree is balanced

insert 2

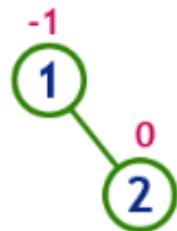


Tree is balanced

AVL Tree: Example

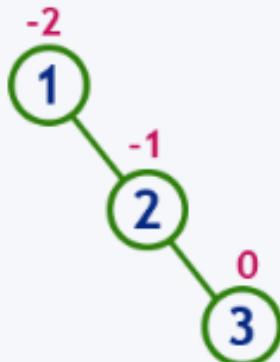
Construct an AVL Tree by inserting numbers from 1 to 8.

insert 2

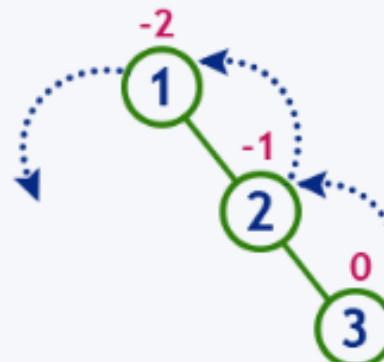


Tree is balanced

insert 3

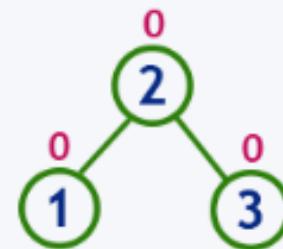


Tree is imbalanced



LL Rotation

After LL Rotation

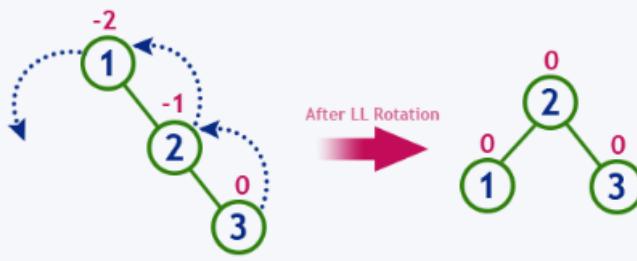
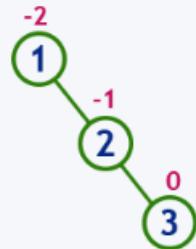


Tree is balanced

AVL Tree: Example

Construct an AVL Tree by inserting numbers from 1 to 8.

insert 3

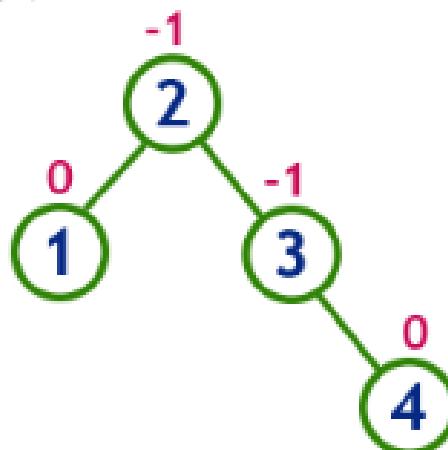


After LL Rotation

Tree is balanced

Tree is imbalanced

insert 4

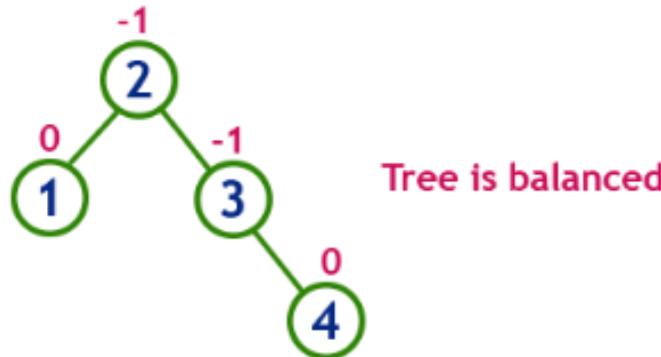


Tree is balanced

AVL Tree: Example

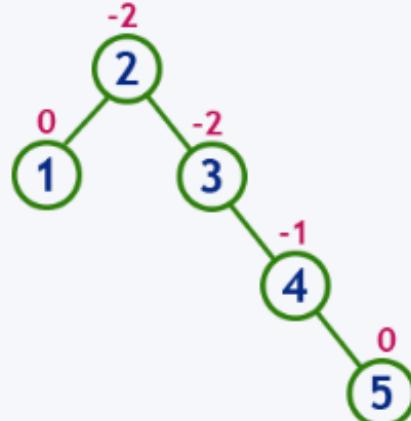
Construct an AVL Tree by inserting numbers from 1 to 8.

insert 4

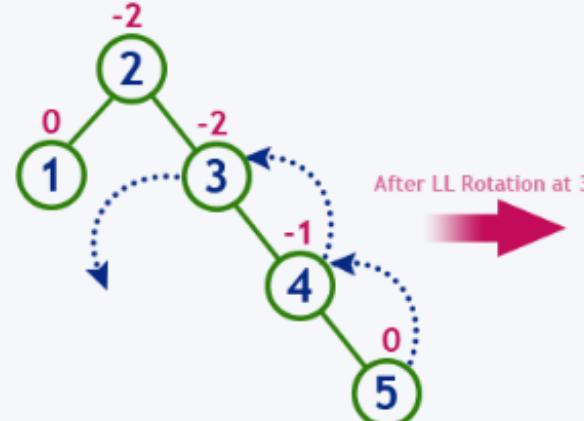


Tree is balanced

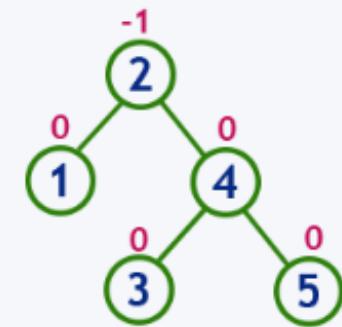
insert 5



Tree is imbalanced



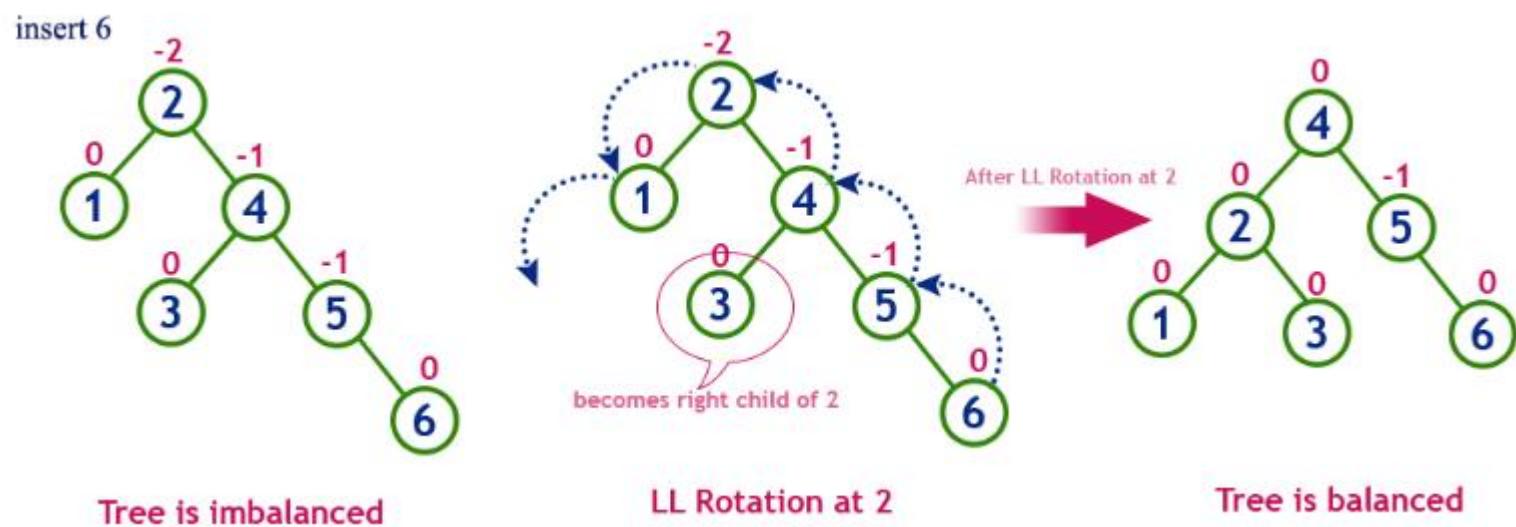
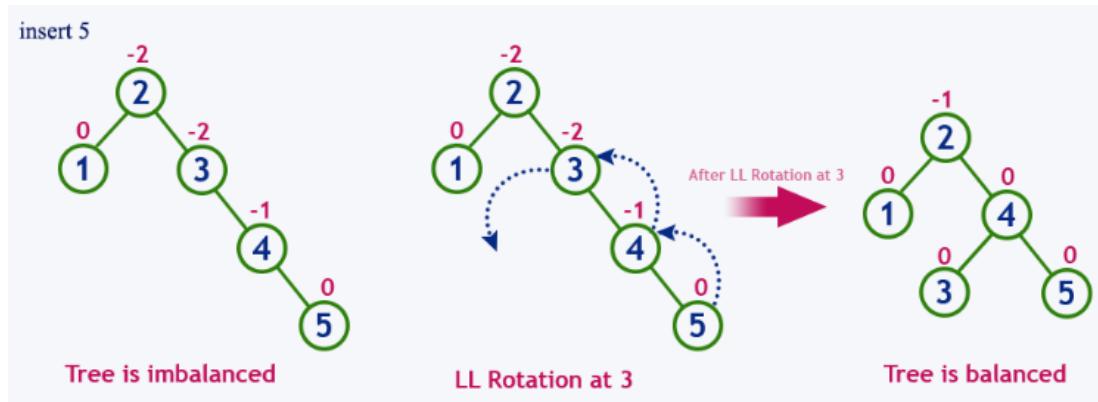
LL Rotation at 3



Tree is balanced

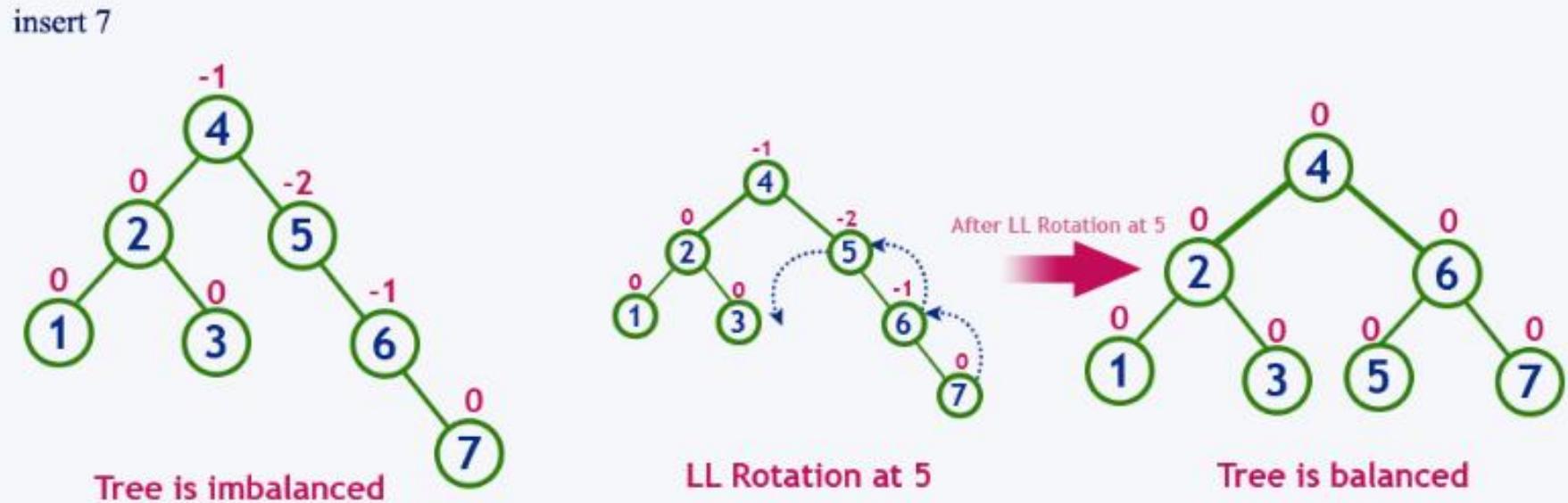
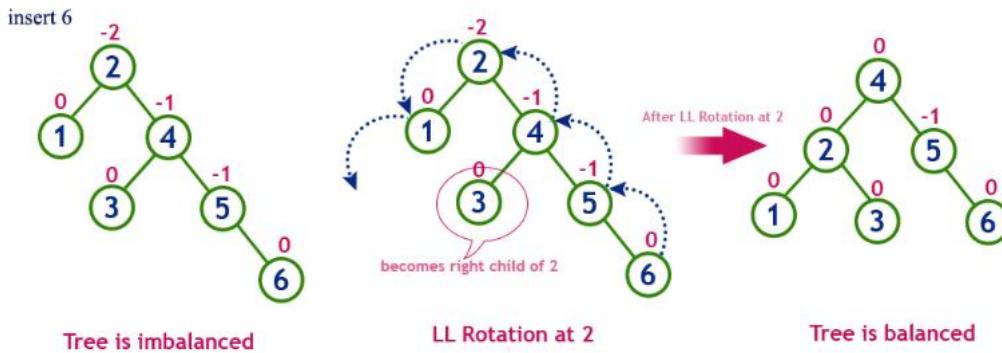
AVL Tree: Example

Construct an AVL Tree by inserting numbers from 1 to 8.



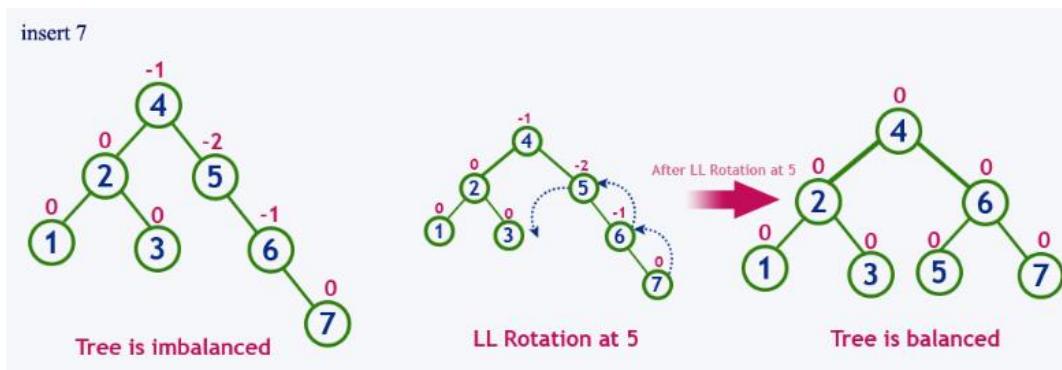
AVL Tree: Example

Construct an AVL Tree by inserting numbers from 1 to 8.

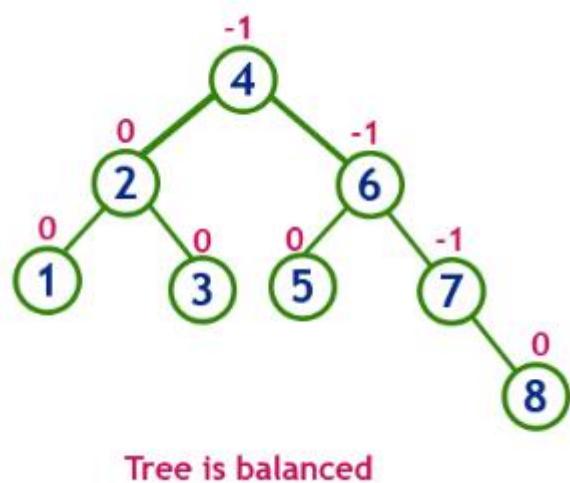


AVL Tree: Example

Construct an AVL Tree by inserting numbers from 1 to 8.

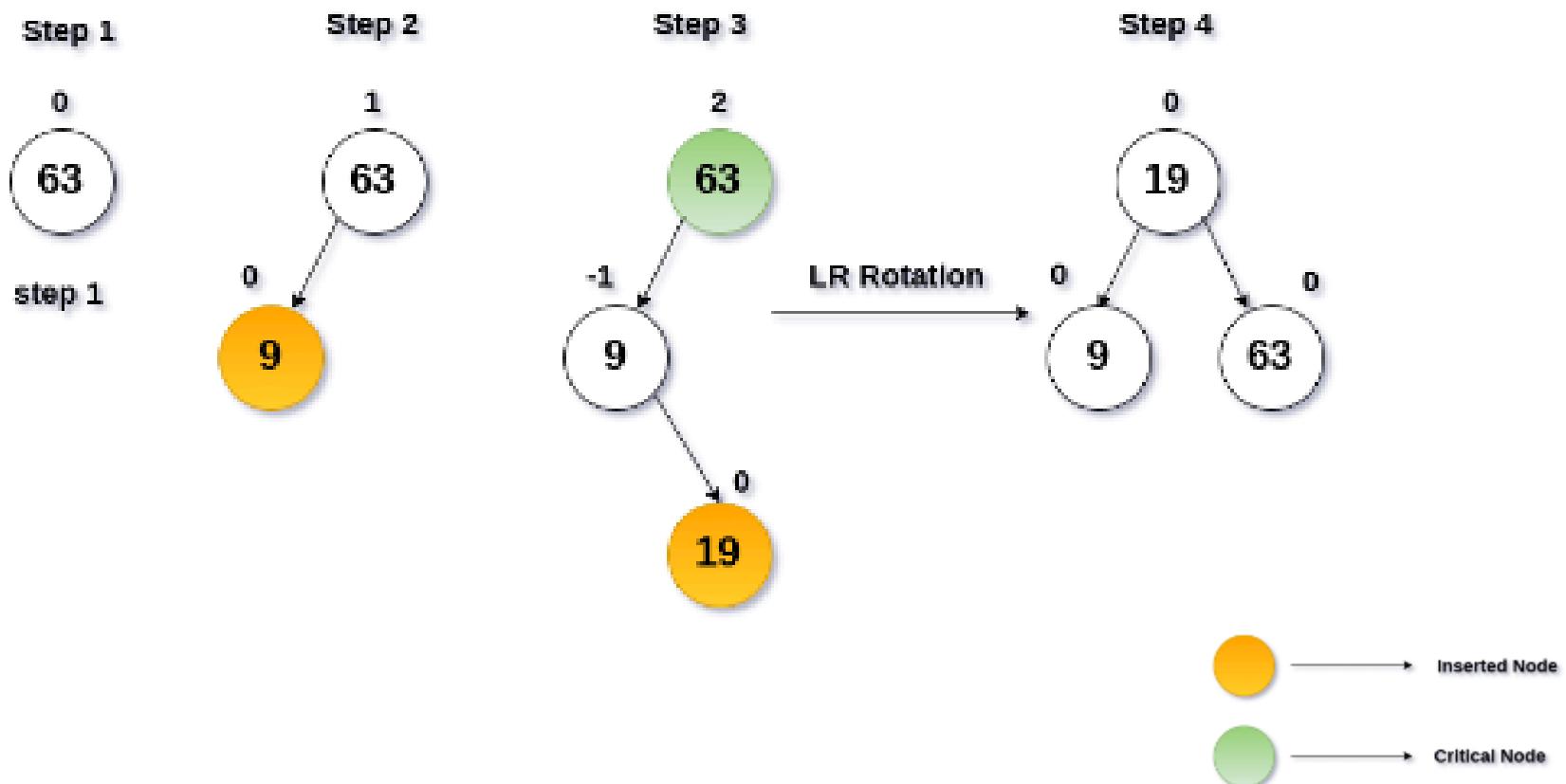


insert 8



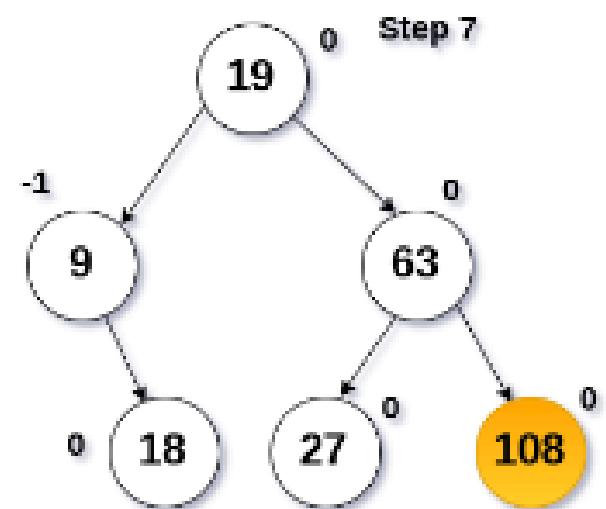
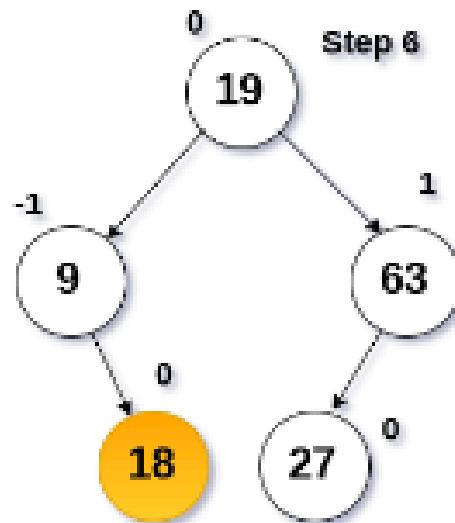
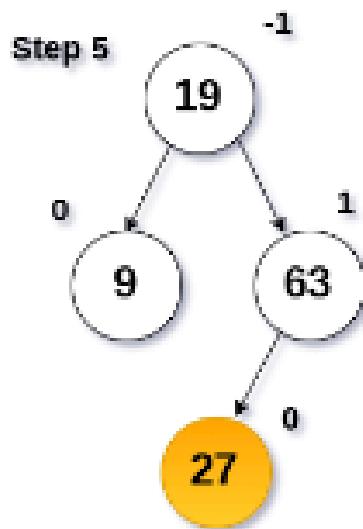
AVL Tree: Example

- Construct an AVL tree by inserting the following elements in the given order. **63, 9, 19, 27, 18, 108, 99, 81**



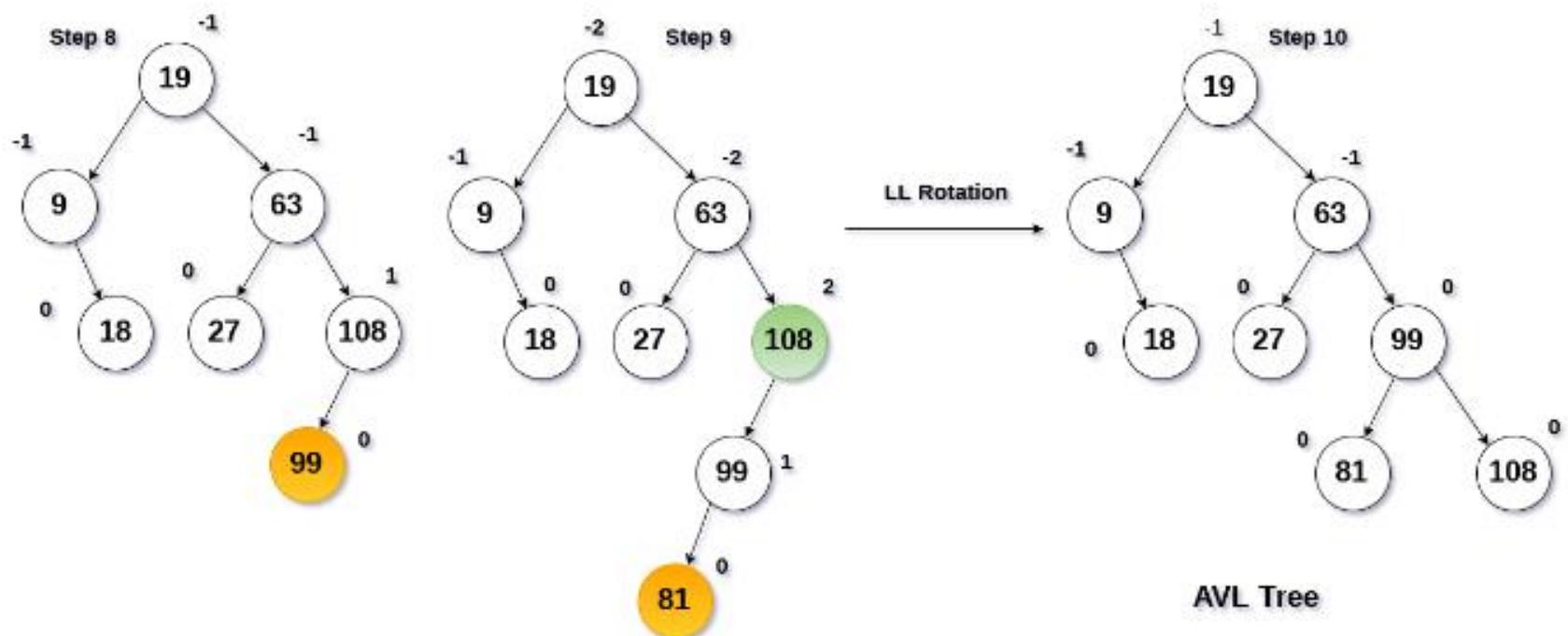
AVL Tree: Example

- Construct an AVL tree by inserting the following elements in the given order. **63, 9, 19, 27, 18, 108, 99, 81**



AVL Tree: Example

- Construct an AVL tree by inserting the following elements in the given order. **63, 9, 19, 27, 18, 108, 99, 81**



AVL Tree: Example

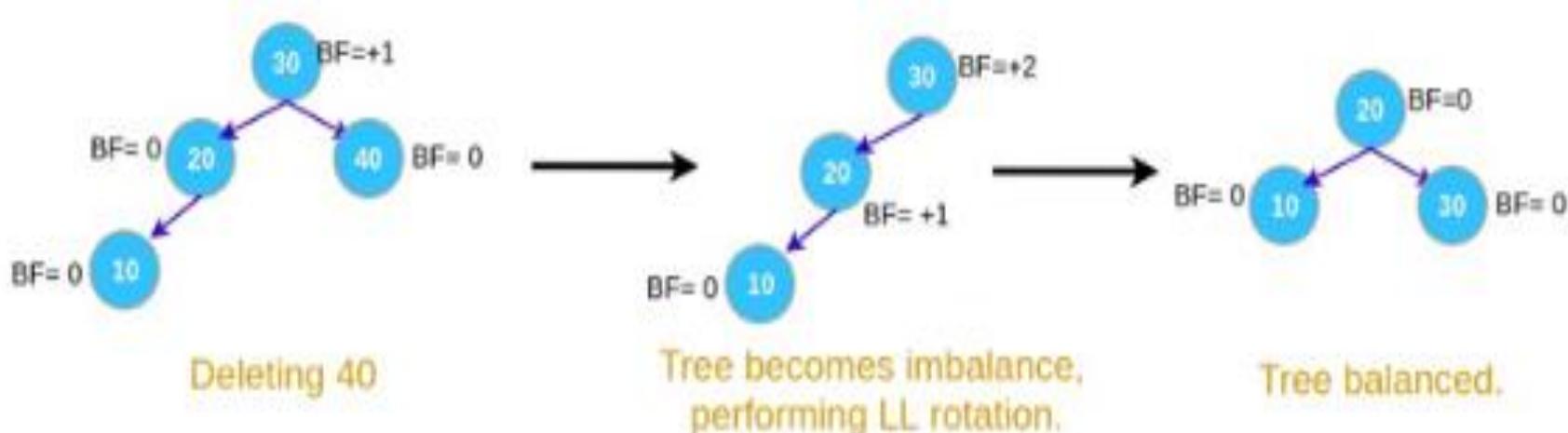
Deletion in AVL Trees

- Step 1: Find the element in the tree.
- Step 2: Delete the node, as per the BST Deletion.
- Step 3: Two cases are possible:-
 - Case 1: Deleting from the right subtree.
 - Case 2: Deleting from left subtree.

AVL Tree: Example

Deleting from the right subtree.

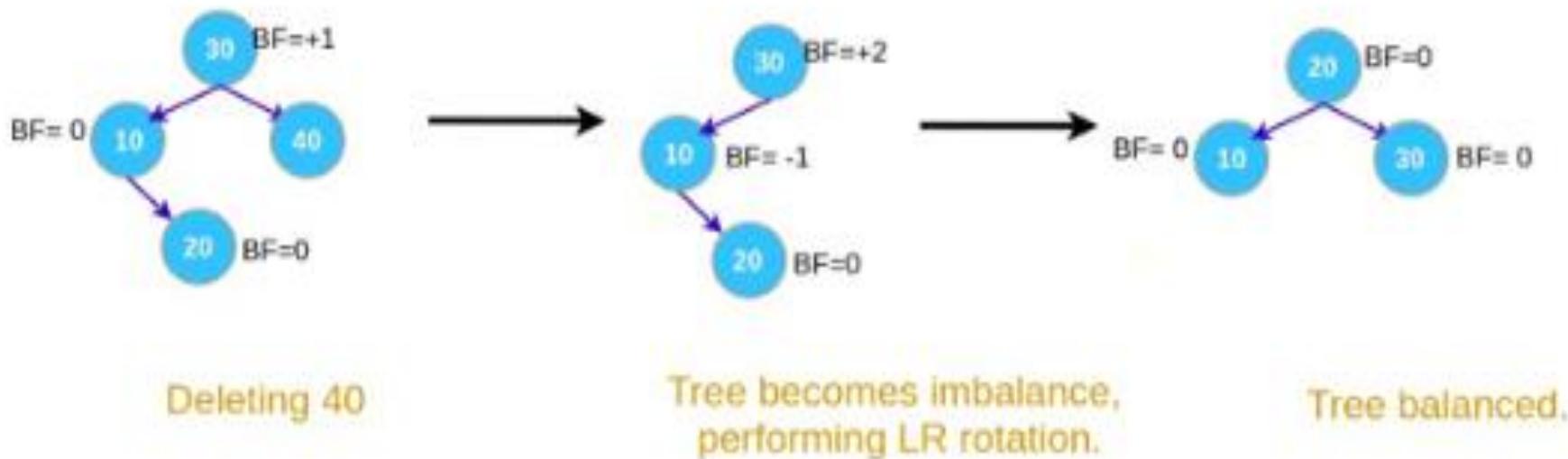
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = +1$, perform LL rotation.
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = -1$, perform LR rotation.
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = 0$, perform LL rotation.



AVL Tree: Example

Deleting from the right subtree.

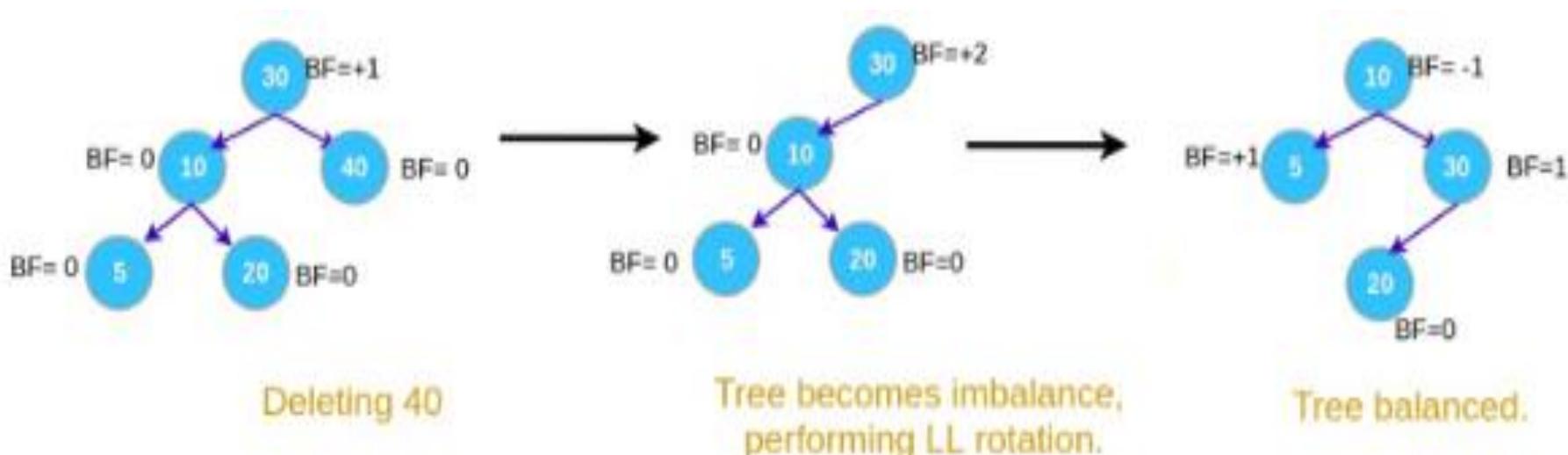
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = +1$, perform LL rotation.
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = -1$, perform LR rotation.
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = 0$, perform LL rotation.



AVL Tree: Example

Deleting from the right subtree.

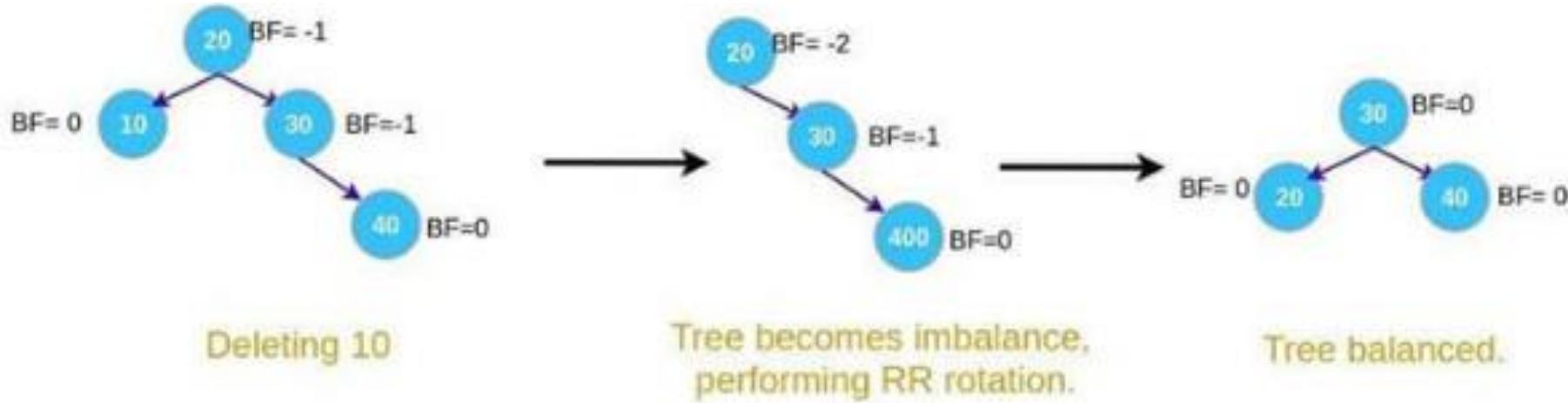
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = +1$, perform LL rotation.
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = -1$, perform LR rotation.
- If $\text{BF}(\text{node}) = +2$ and $\text{BF}(\text{node} \rightarrow \text{left-child}) = 0$, perform LL rotation.



AVL Tree: Example

Deleting from the left subtree.

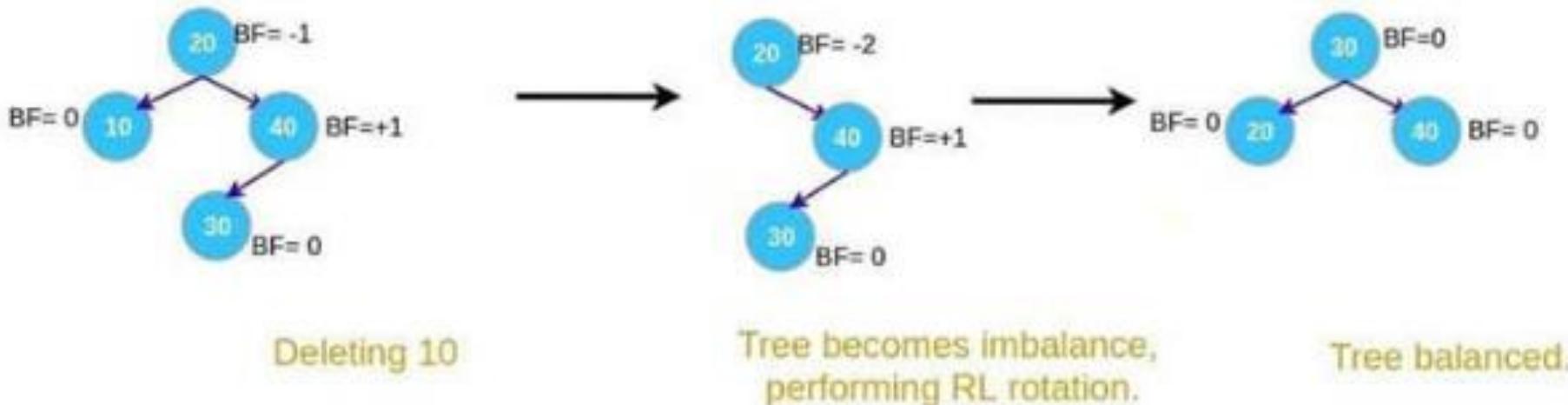
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = -1$, perform RR rotation.
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = +1$, perform RL rotation.
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = 0$, perform RR rotation.



AVL Tree: Example

Deleting from the left subtree.

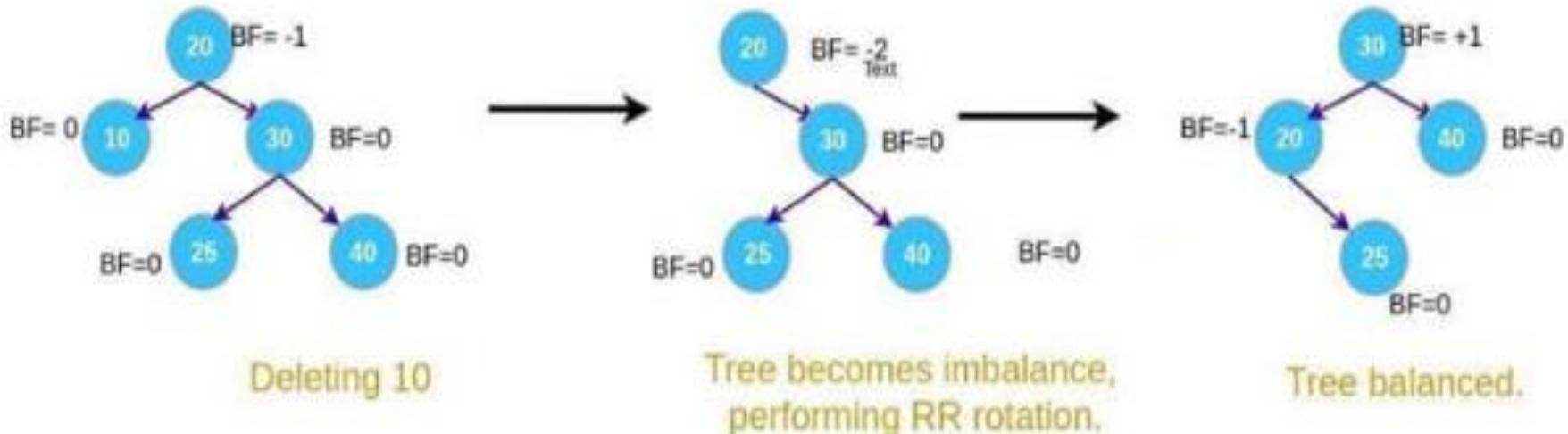
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = -1$, perform RR rotation.
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = +1$, perform RL rotation.
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = 0$, perform RR rotation.



AVL Tree: Example

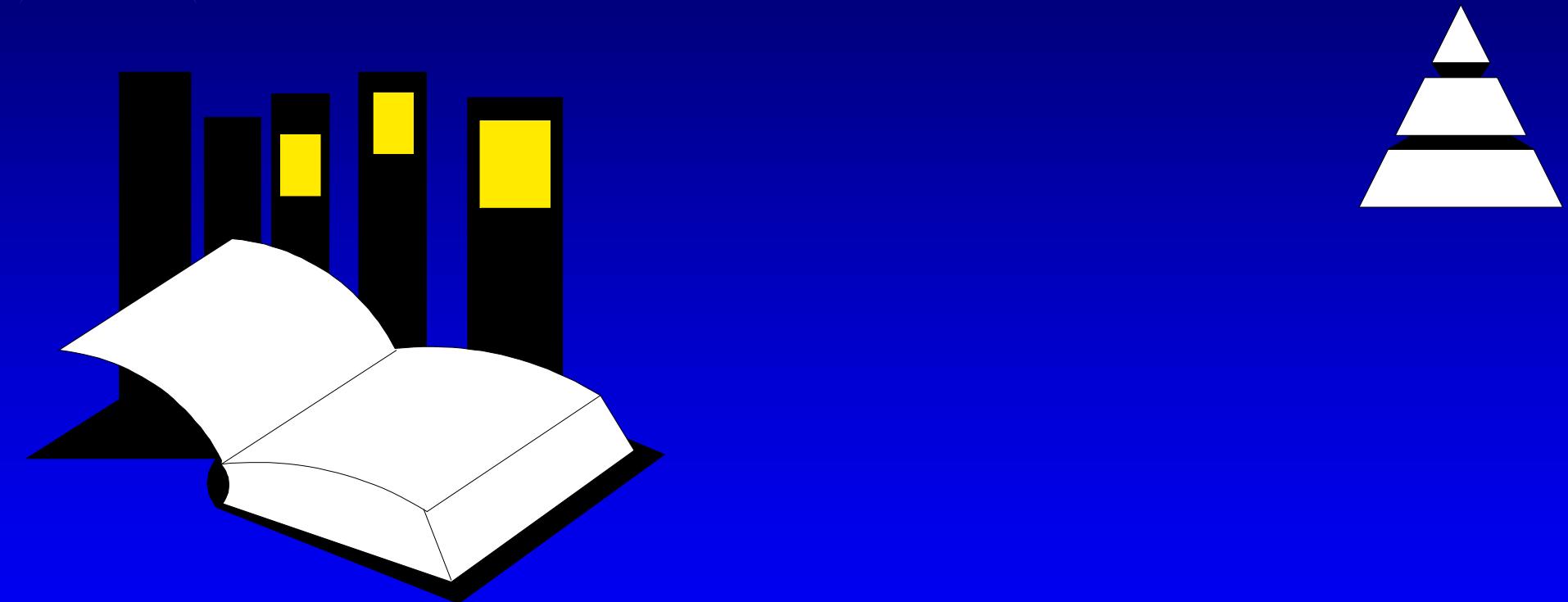
Deleting from the left subtree.

- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = -1$, perform RR rotation.
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = +1$, perform RL rotation.
- If $\text{BF}(\text{node}) = -2$ and $\text{BF}(\text{node} \rightarrow \text{right-child}) = 0$, perform RR rotation.





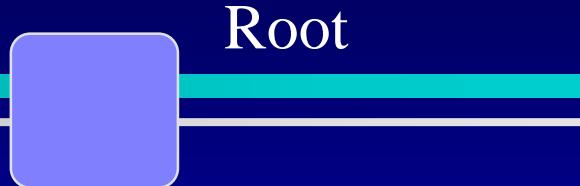
Heaps



Heaps

A heap is a
certain kind of
complete
binary tree.

Heaps



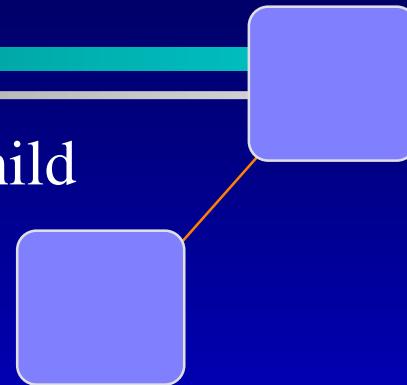
A **heap** is a certain kind of complete binary tree.

When a complete binary tree is built, its first node must be the root.

Heaps

Complete
binary tree.

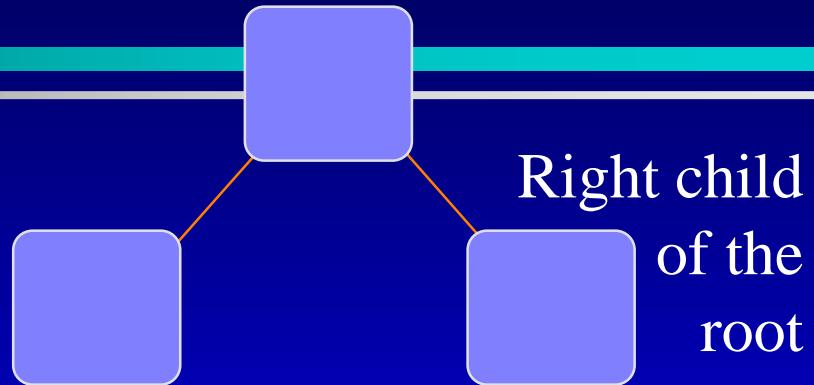
Left child
of the
root



The second node is
always the left child
of the root.

Heaps

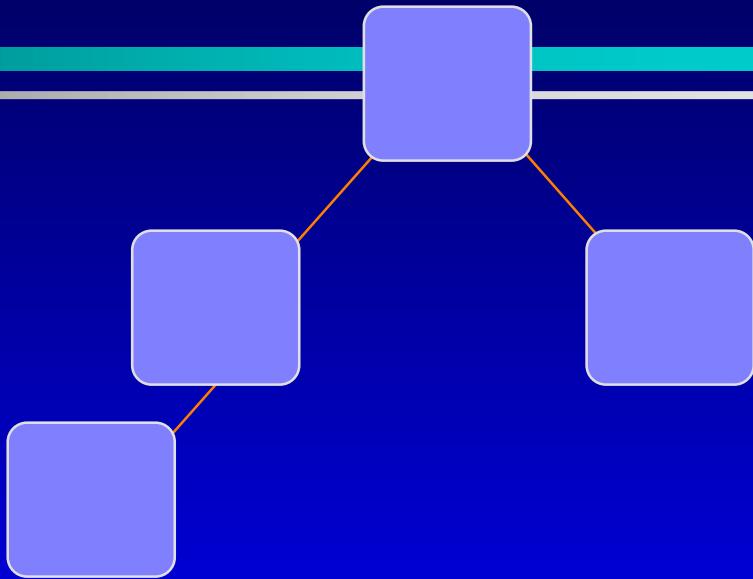
Complete
binary tree.



The third node is
always the right child
of the root.

Heaps

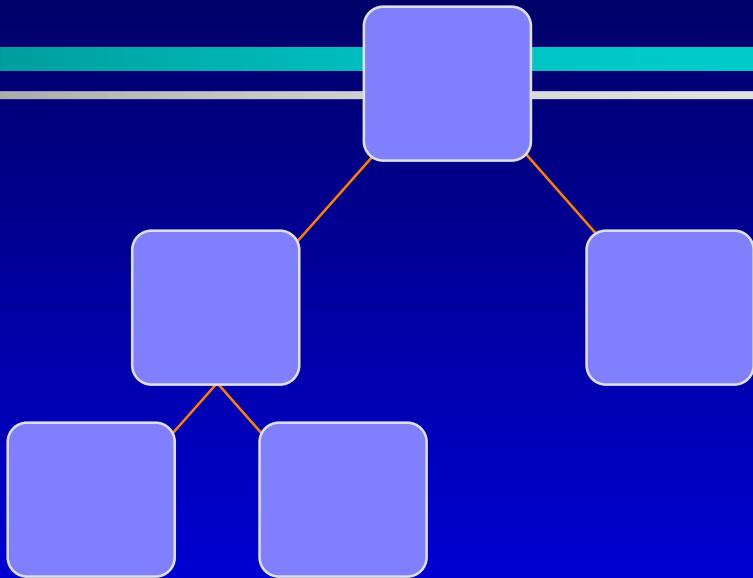
Complete
binary tree.



The next nodes
always fill the next
level from left-to-right.

Heaps

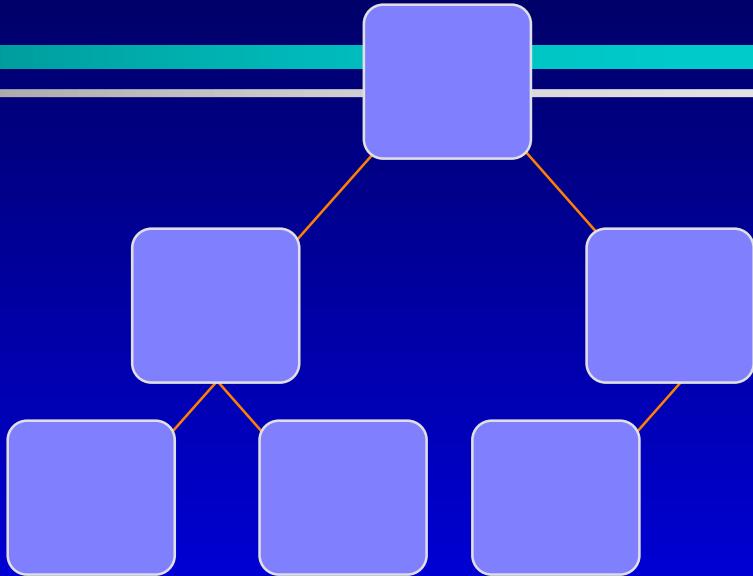
Complete
binary tree.



The next nodes
always fill the next
level from left-to-right.

Heaps

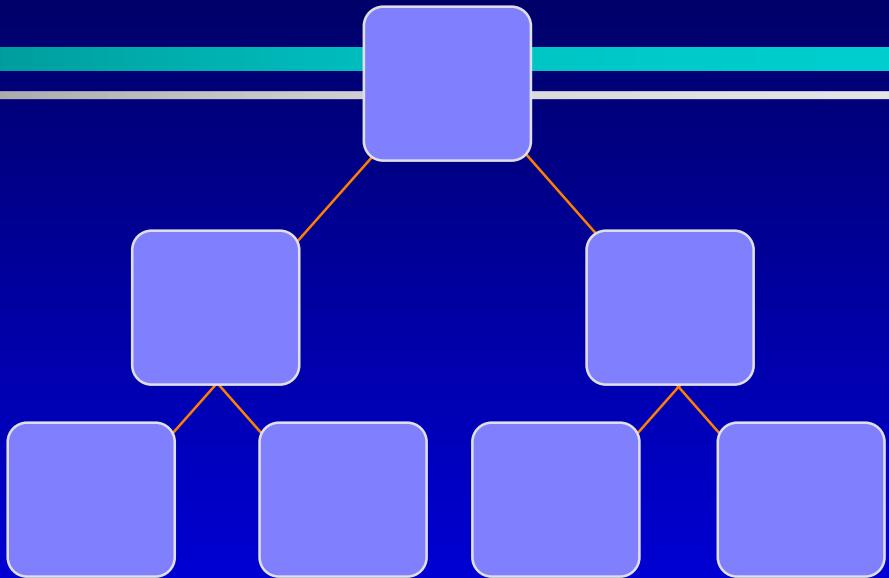
Complete
binary tree.



The next nodes
always fill the next
level from left-to-right.

Heaps

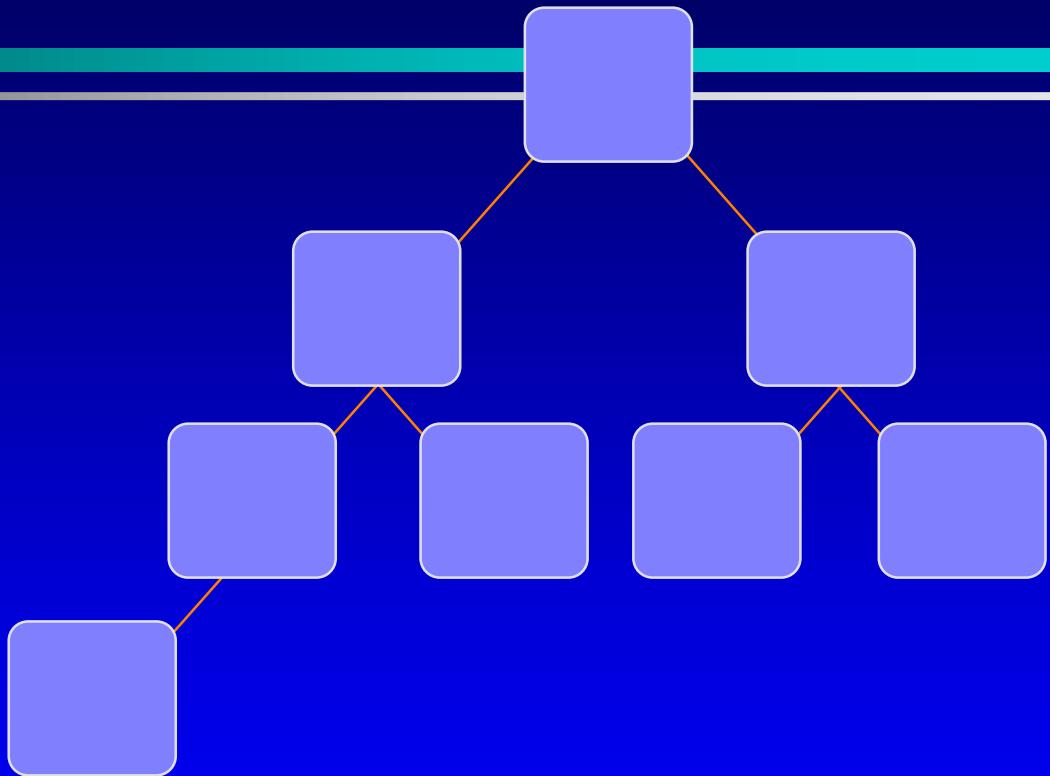
Complete
binary tree.



The next nodes
always fill the next
level from left-to-right.

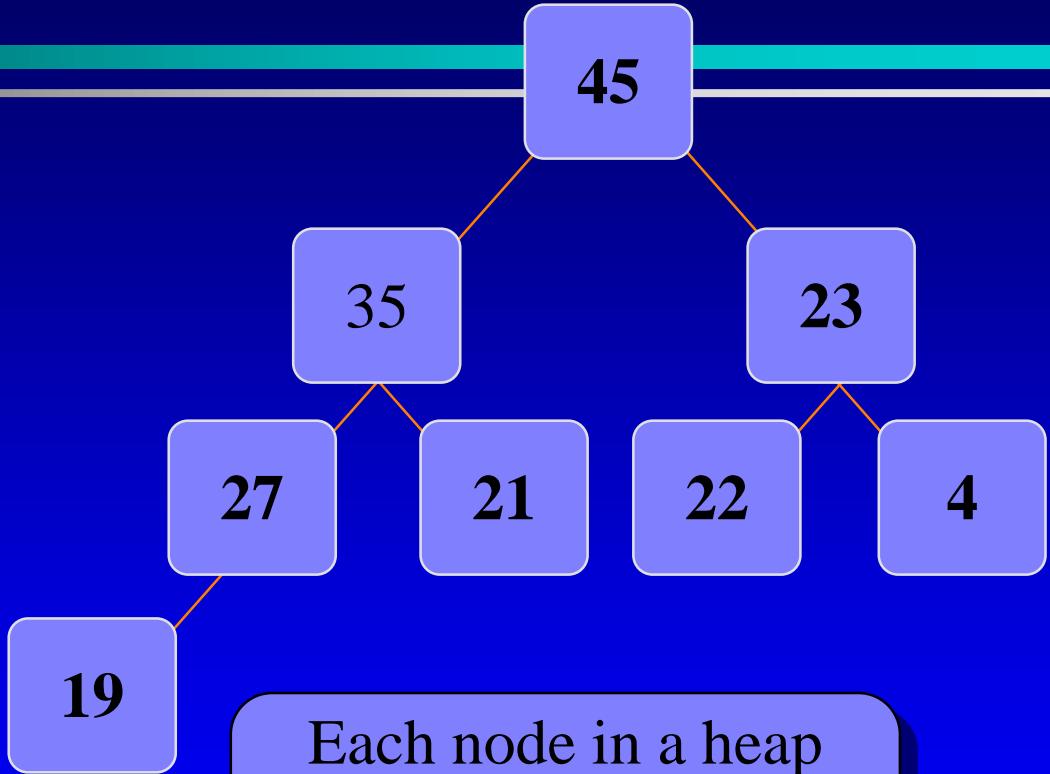
Heaps

Complete
binary tree.



Heaps

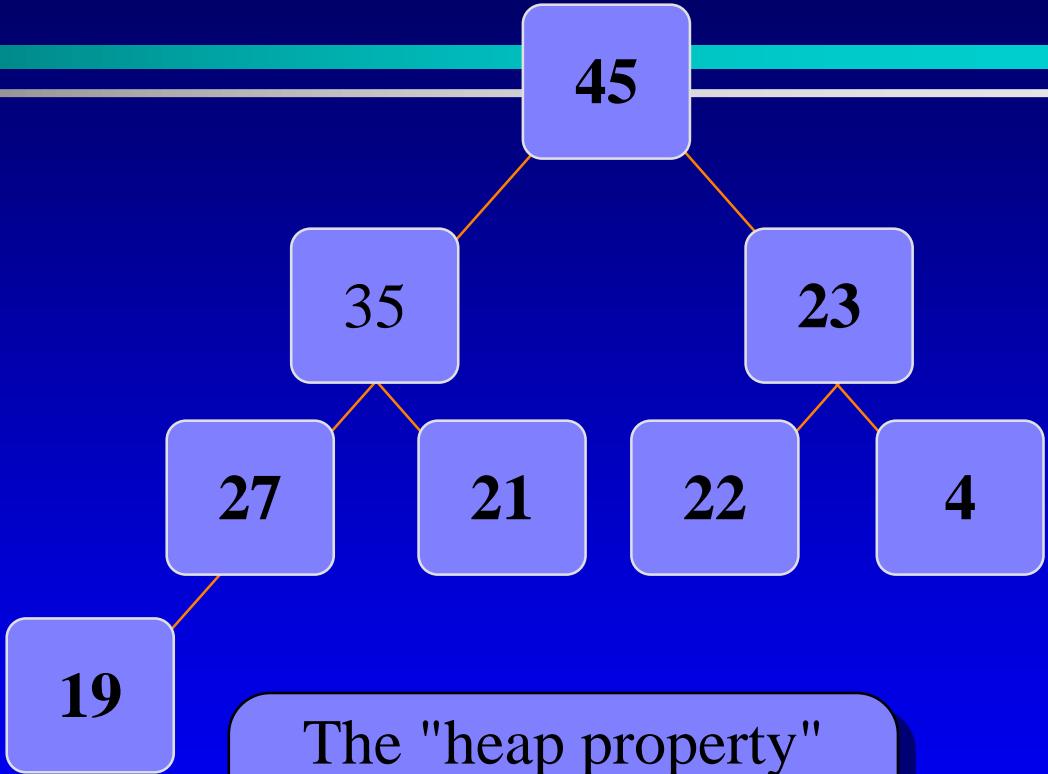
A heap is a **certain** kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

Heaps

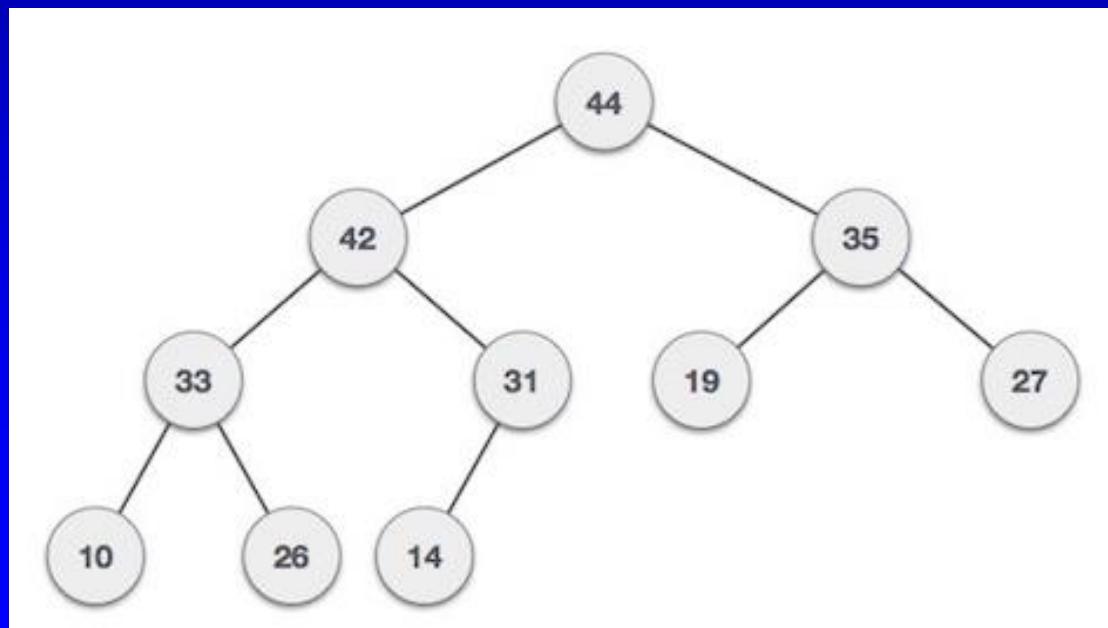
A heap is a **certain** kind of complete binary tree.



The "heap property" requires that each node's key is \geq the keys of its children

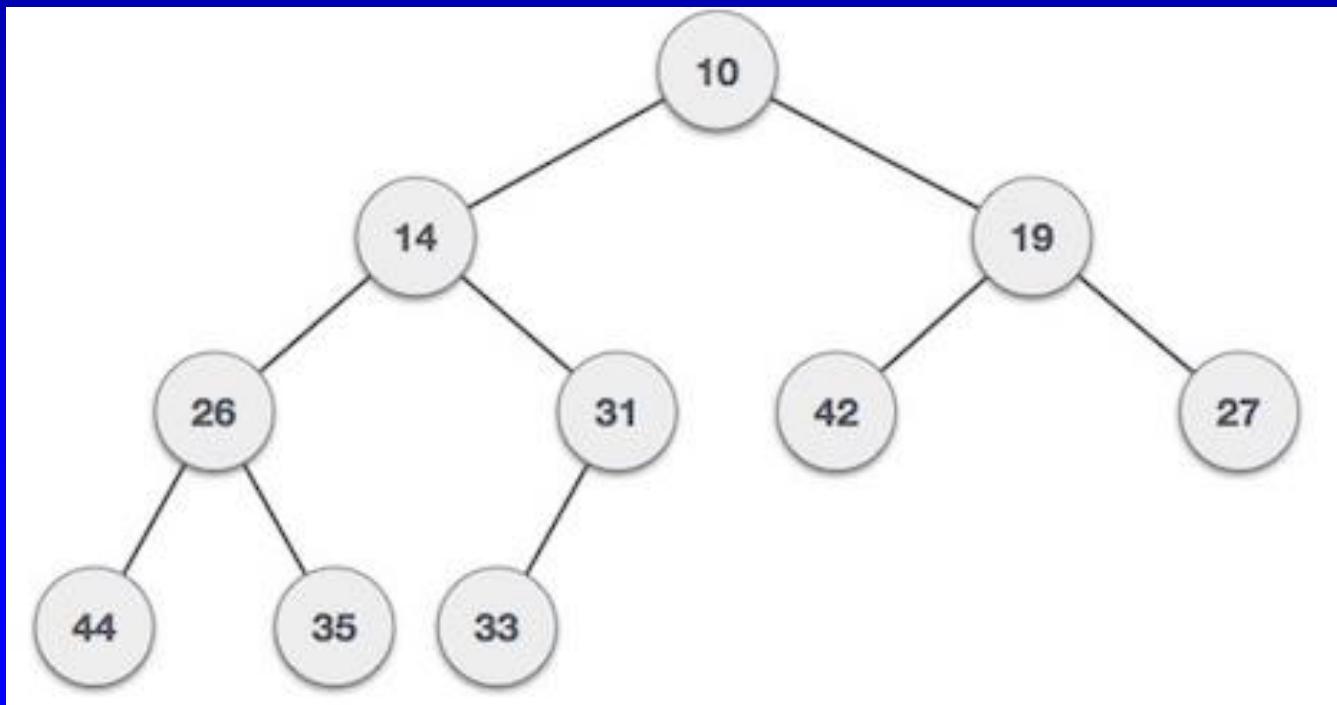
Max Heap and Min Heap

- **Max-Heap** – Where the value of the root node is greater than or equal to either of its children.



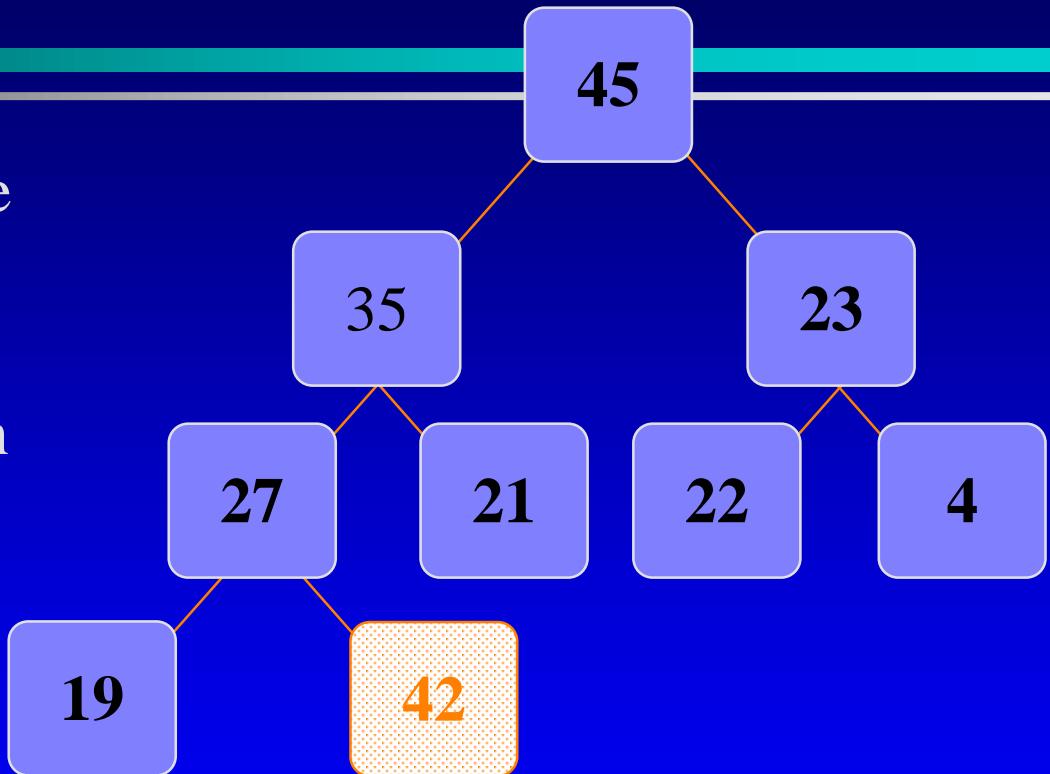
Max Heap and Min Heap

- Min-Heap – Where the value of the root node is less than or equal to either of its children.



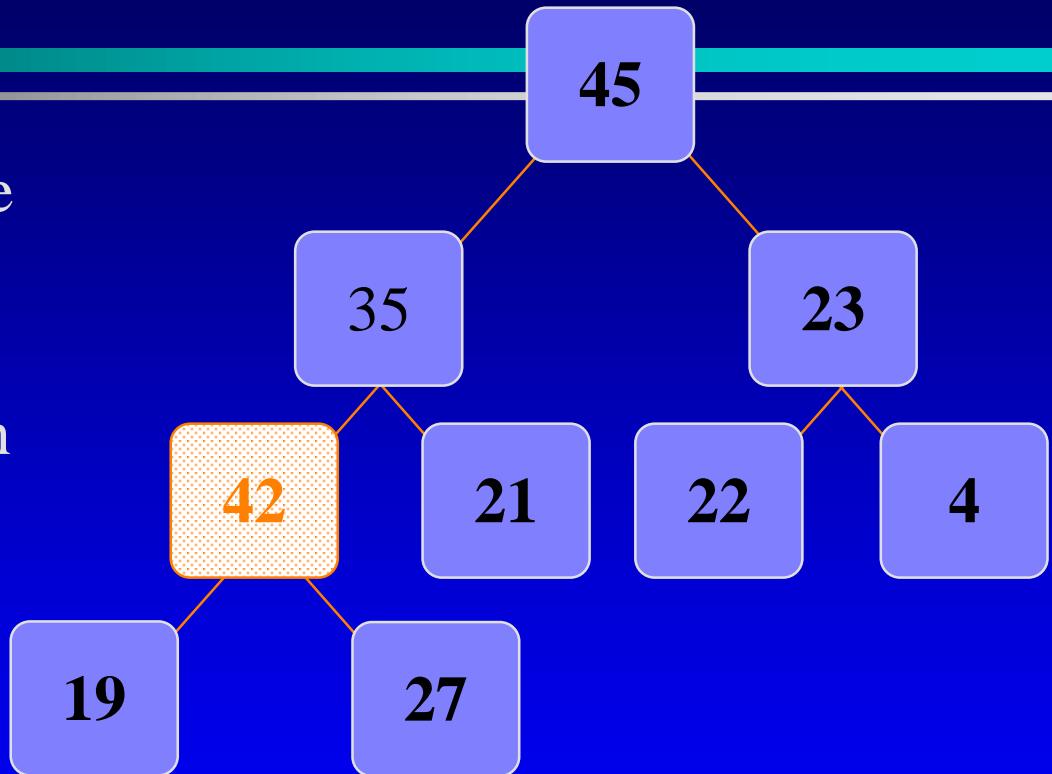
Adding a Node to a Heap

- ❑ Put the new node in the next available spot.
- ❑ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



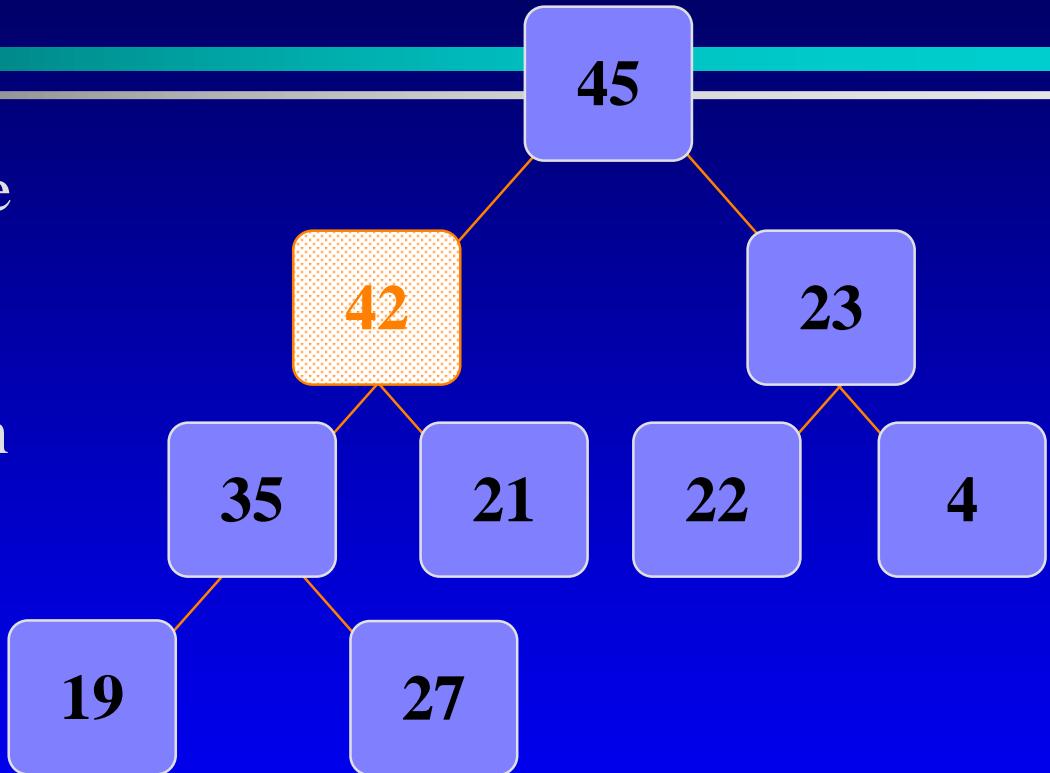
Adding a Node to a Heap

- ❑ Put the new node in the next available spot.
- ❑ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



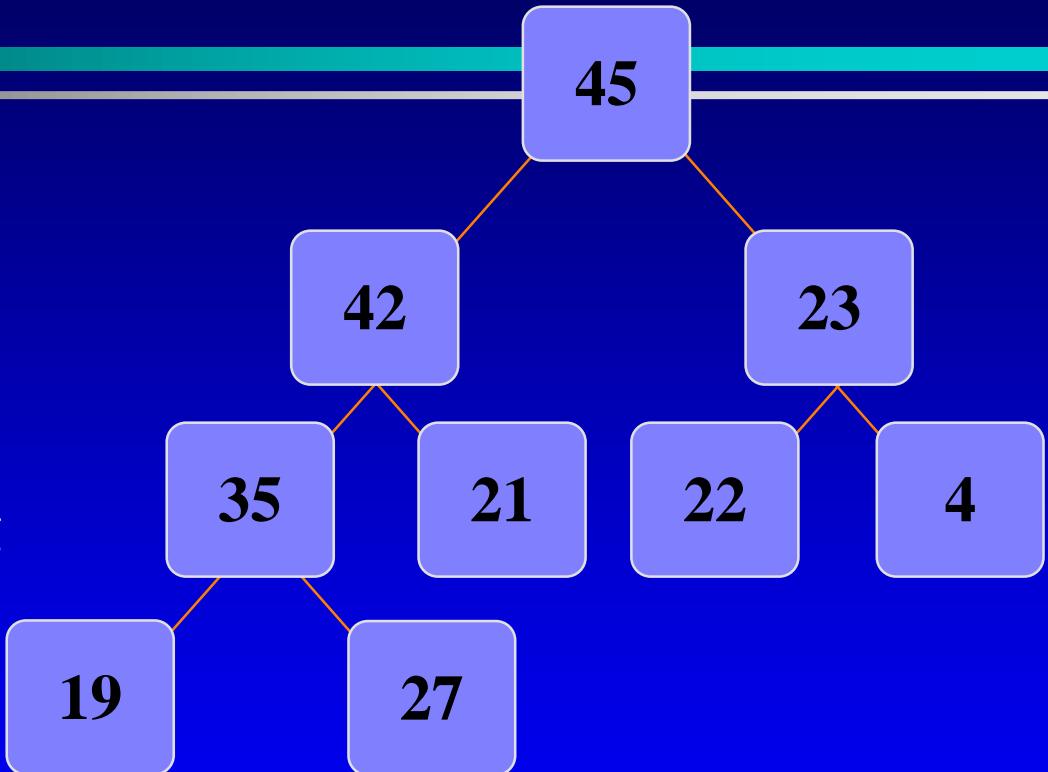
Adding a Node to a Heap

- ❑ Put the new node in the next available spot.
- ❑ Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



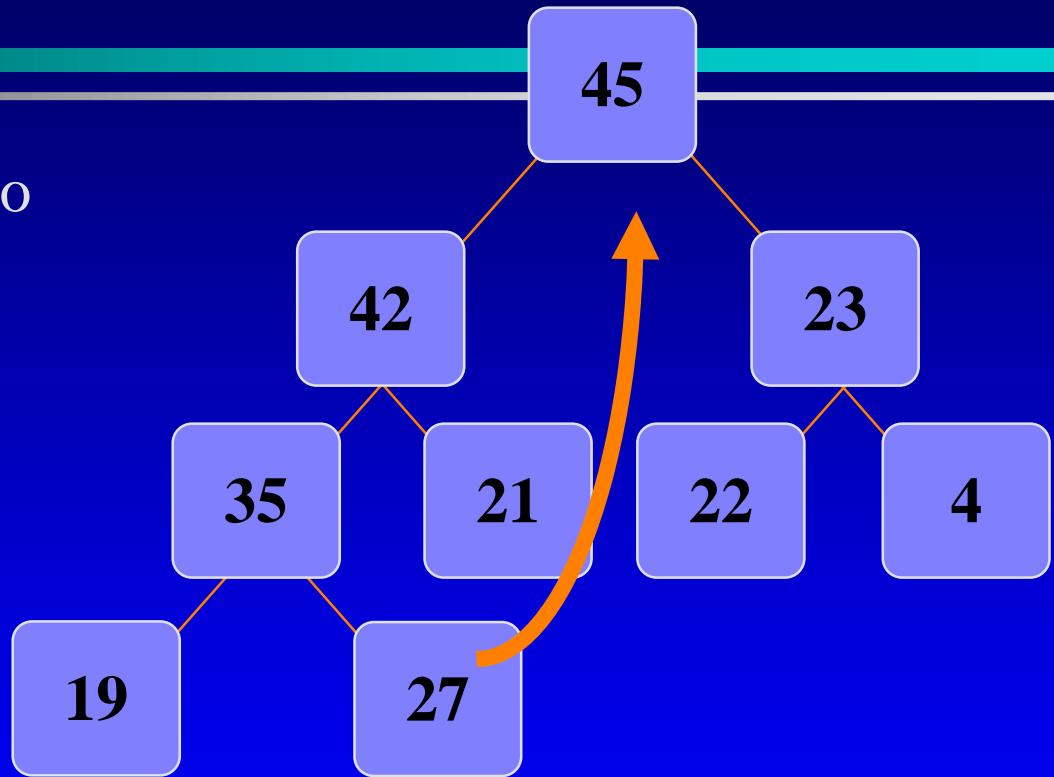
Adding a Node to a Heap

- ❑ The parent has a key that is \geq new node, or
- ❑ The node reaches the root.
- ❑ The process of pushing the new node upward is called **reheapification upward**.



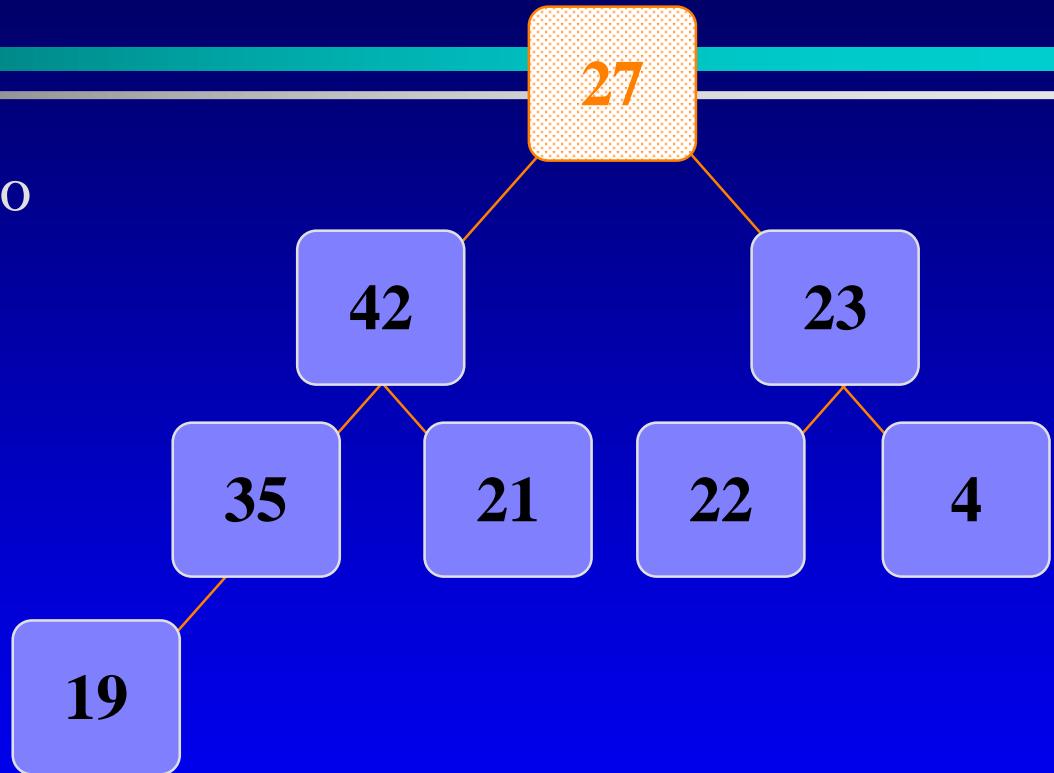
Removing the Top of a Heap

- Move the last node onto the root.



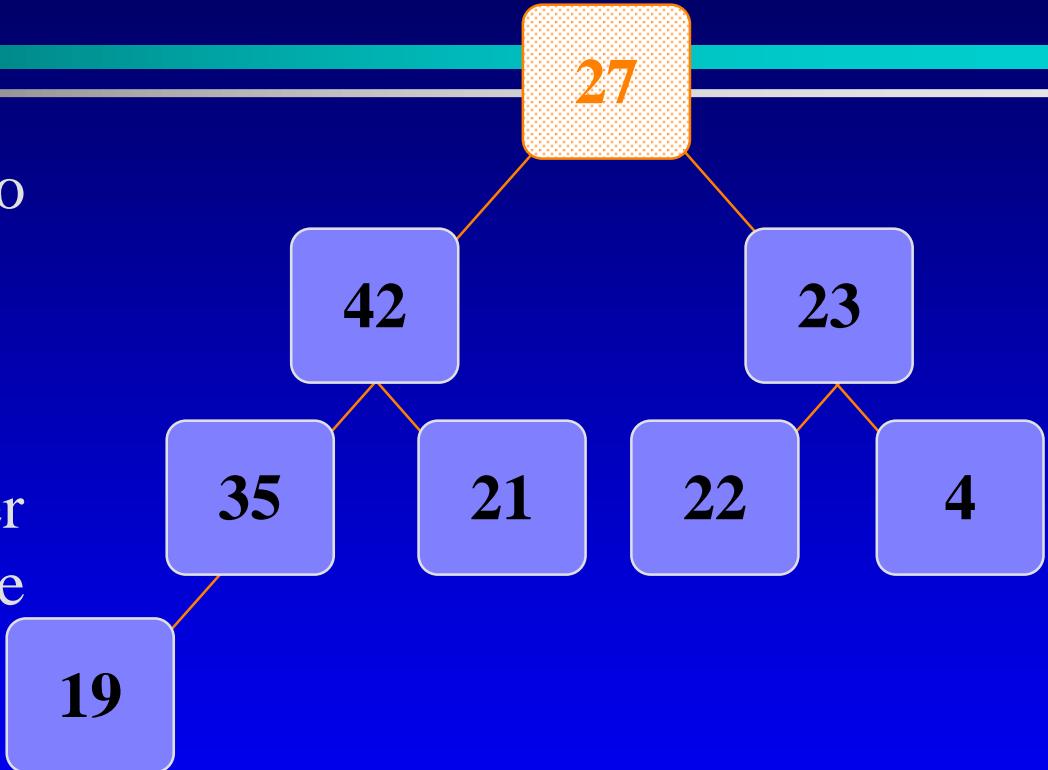
Removing the Top of a Heap

- Move the last node onto the root.



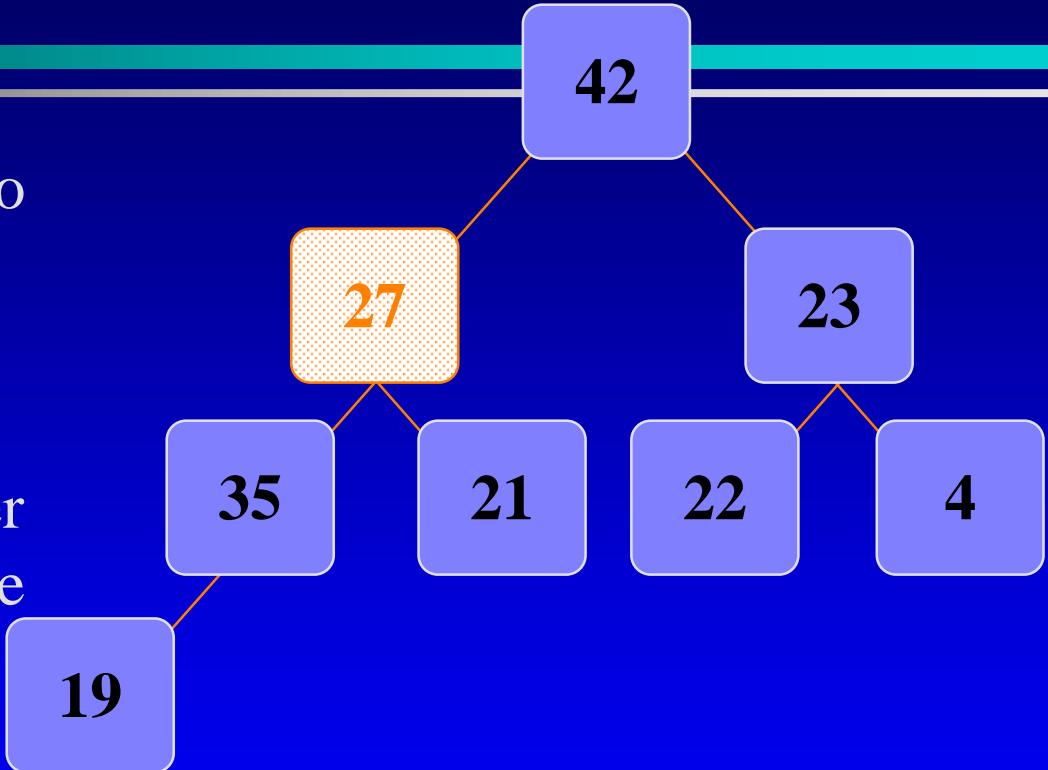
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



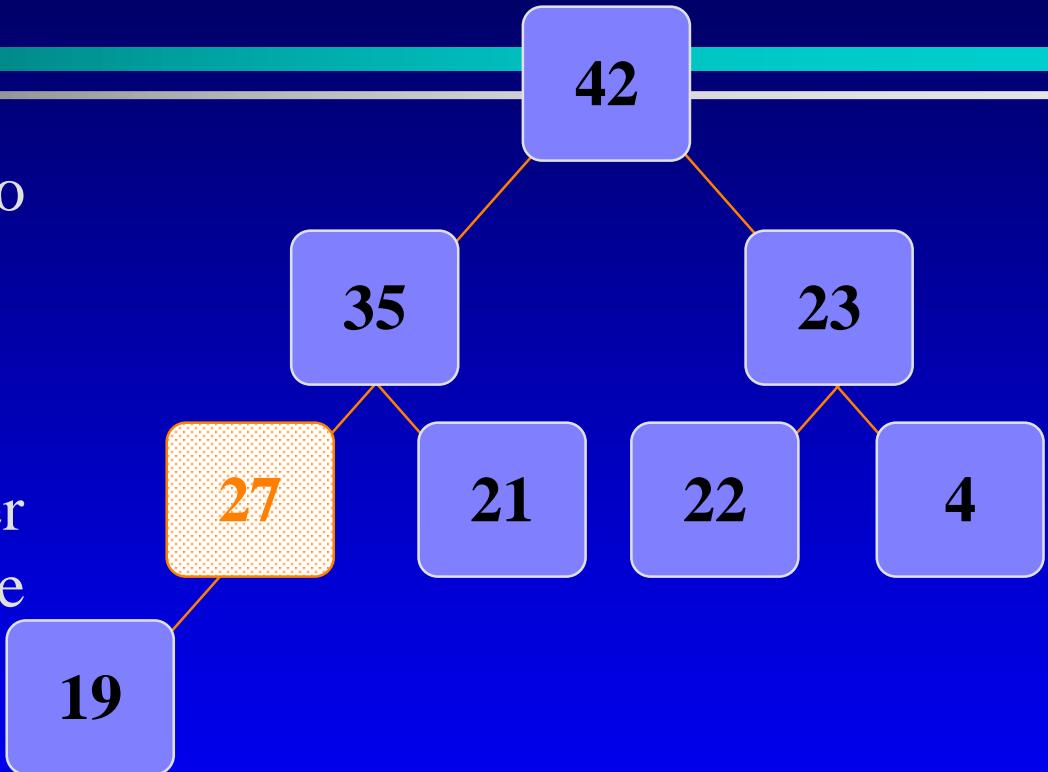
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



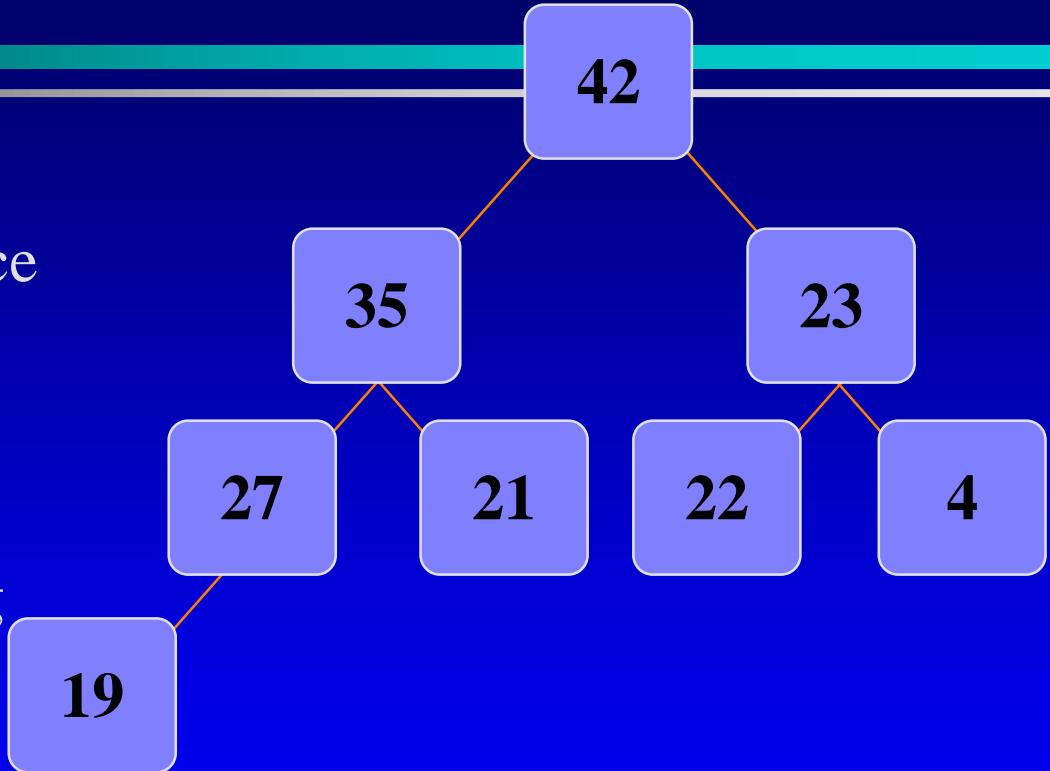
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



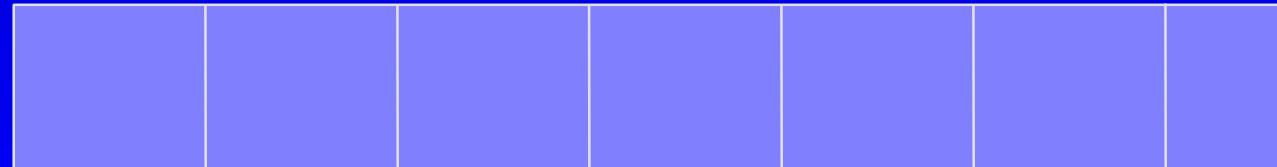
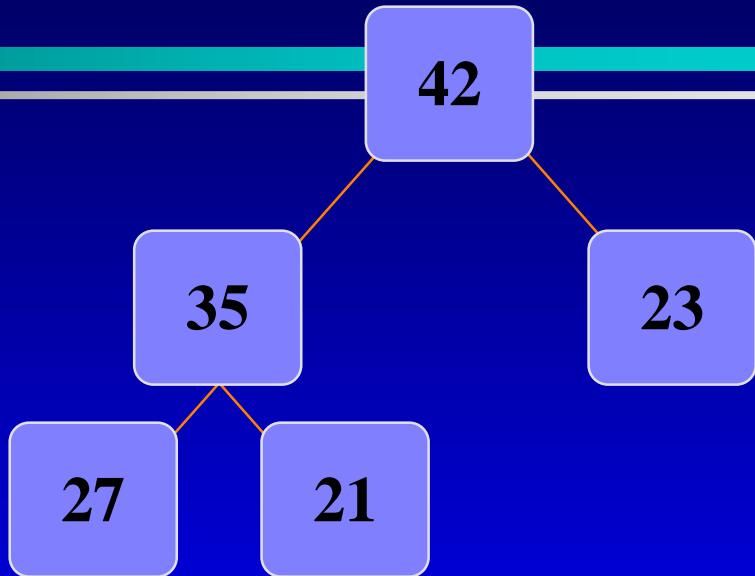
Removing the Top of a Heap

- ❑ The children all have keys \leq the out-of-place node, or
- ❑ The node reaches the leaf.
- ❑ The process of pushing the new node downward is called **reheapification downward**.



Implementing a Heap

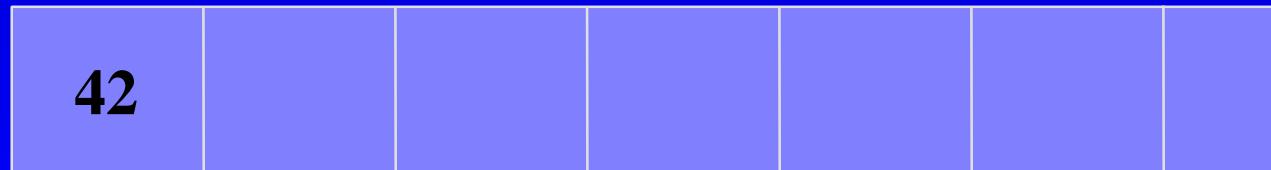
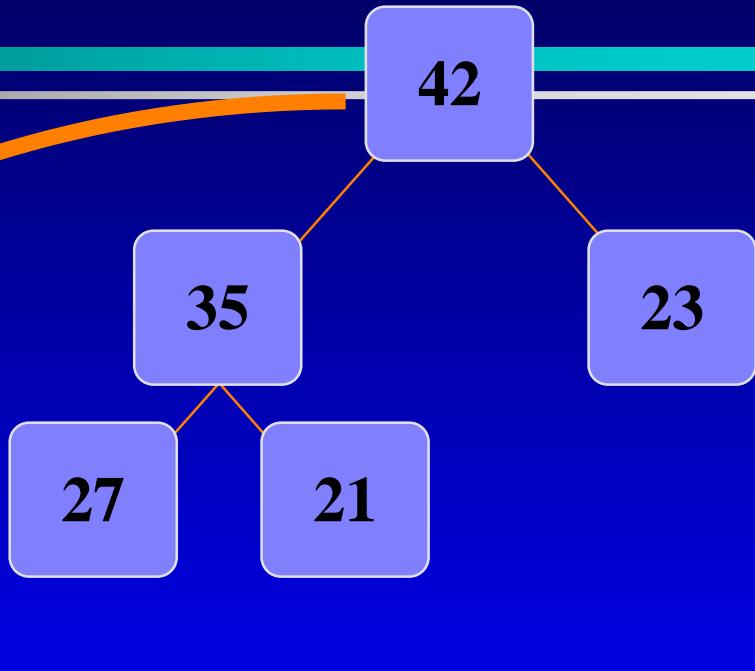
- We will store the data from the nodes in a partially-filled array.



An array of data

Implementing a Heap

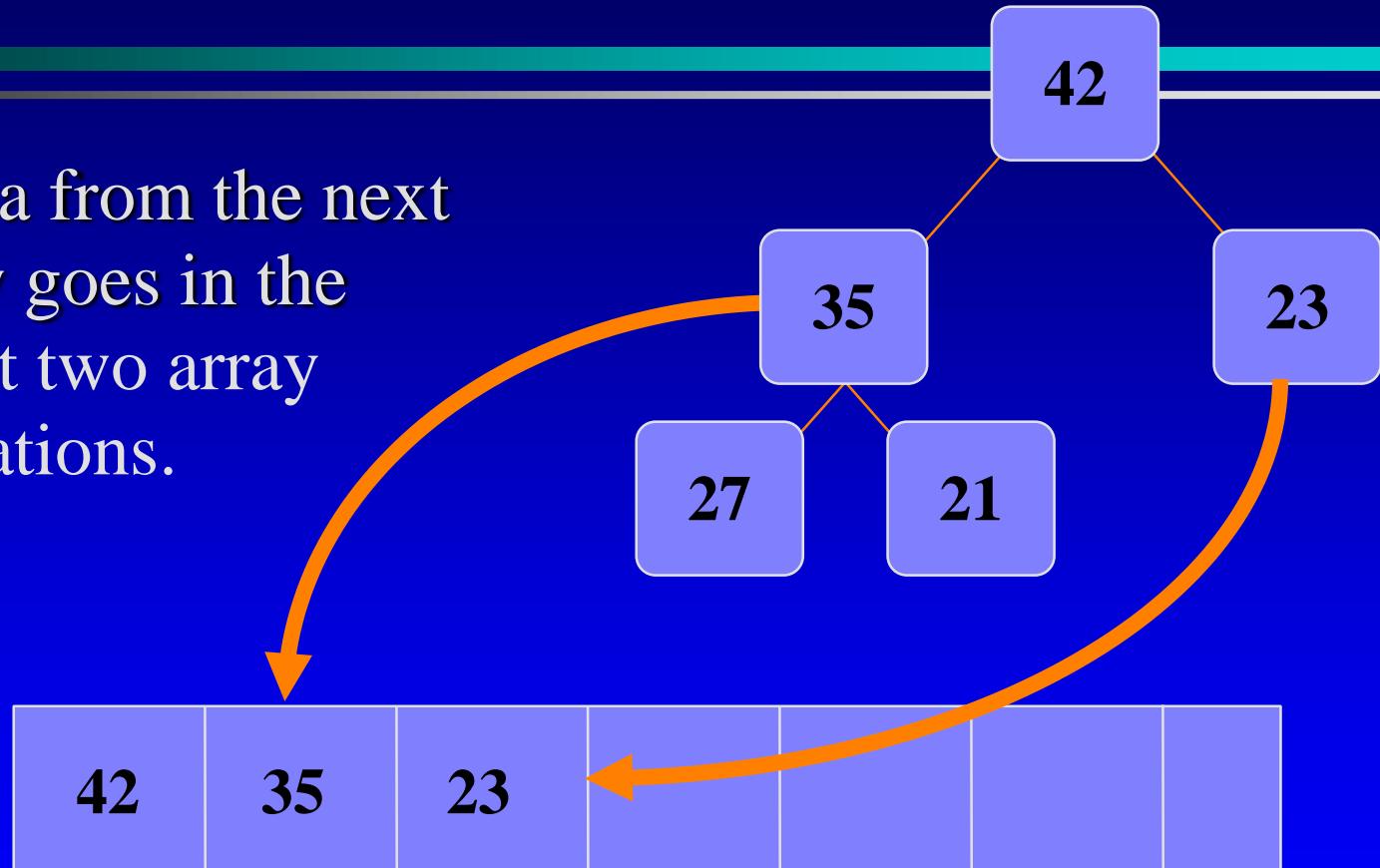
- Data from the root goes in the first location of the array.



An array of data

Implementing a Heap

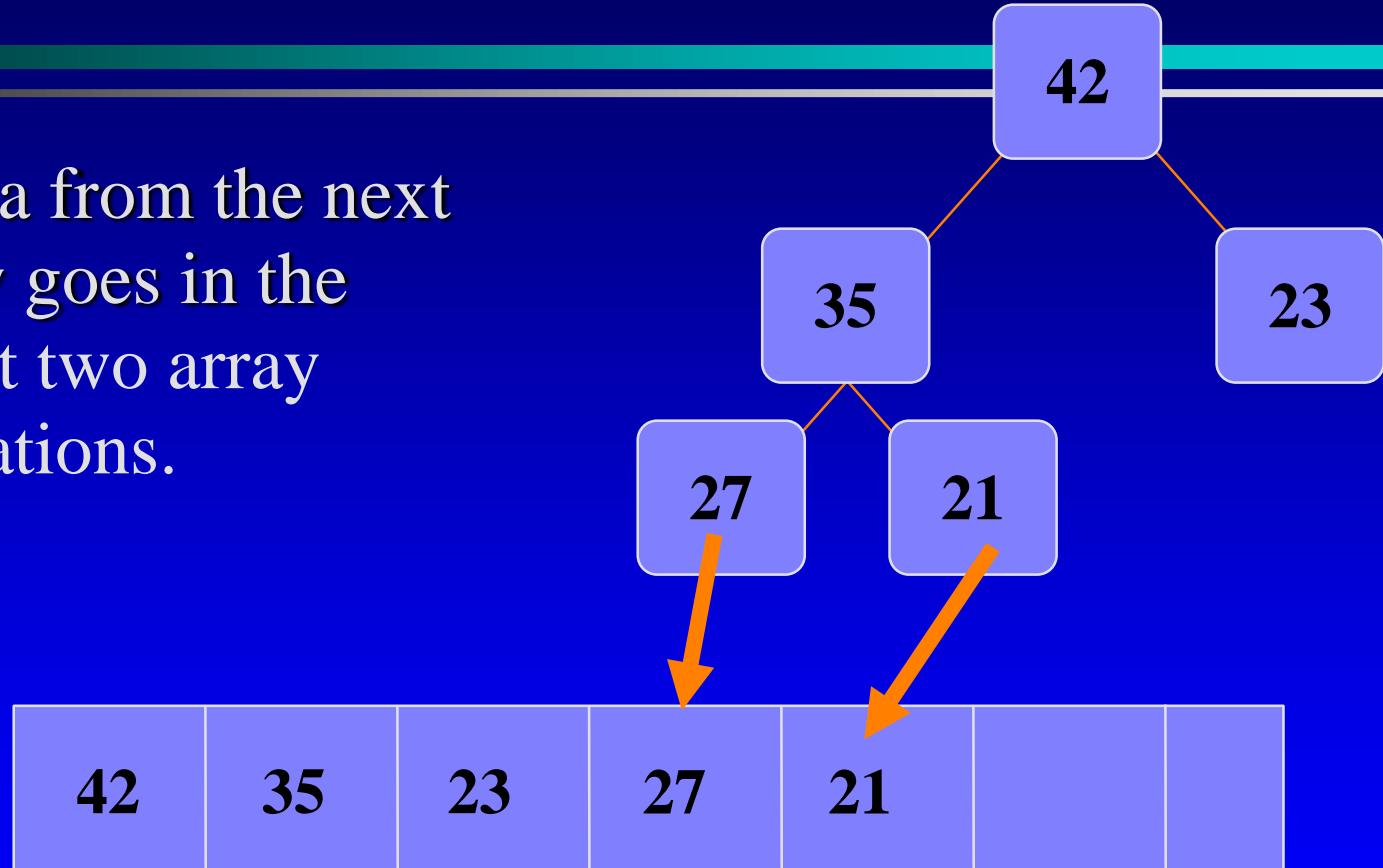
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

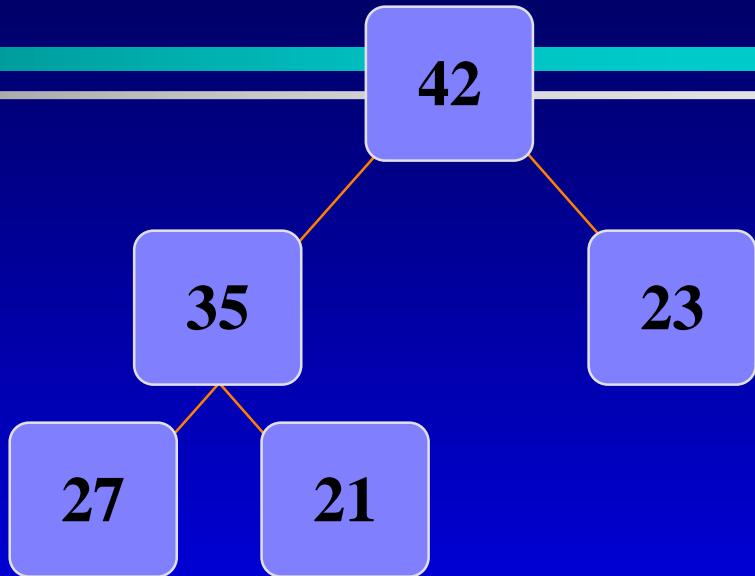
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

- Data from the next row goes in the next two array locations.

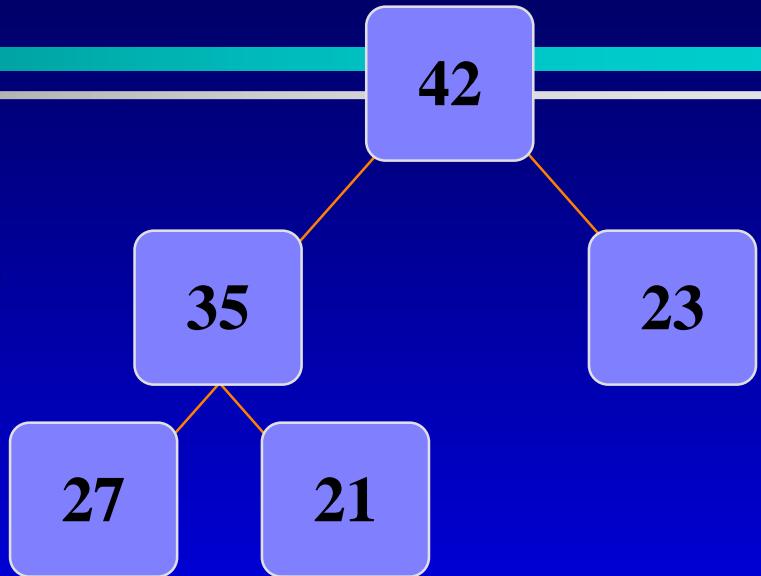


An array of data

We don't care what's in
this part of the array.

Important Points about the Implementation

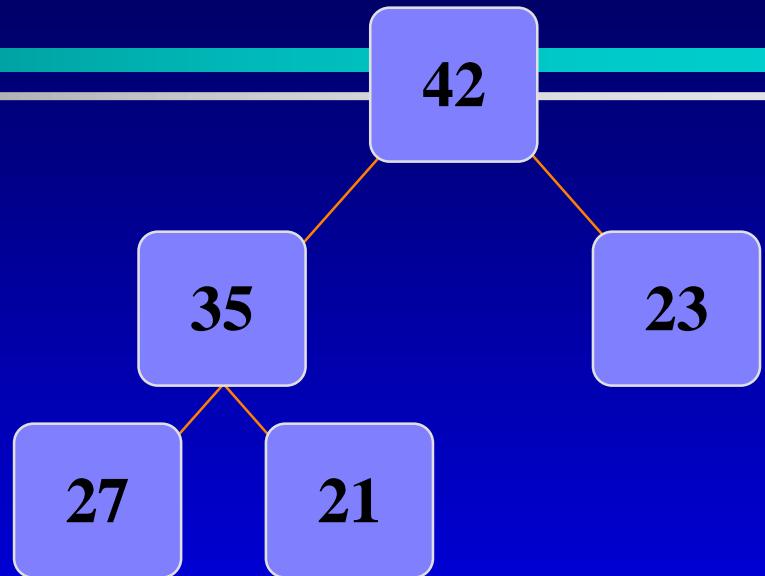
- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.



An array of data

Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given in the book.



42	35	23	27	21		
[1]	[2]	[3]	[4]	[5]		

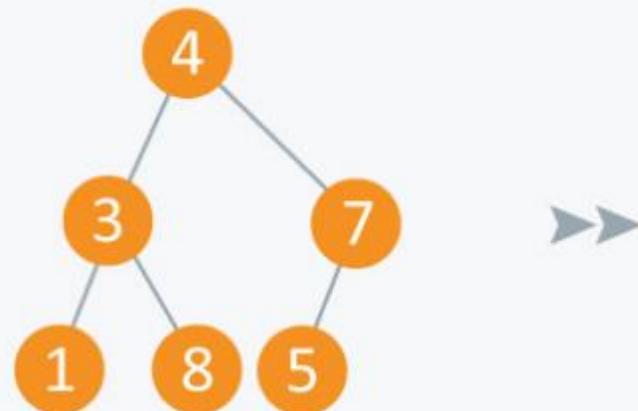
Insert

- Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:
 - First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
 - Secondly, the value of the parent node should be greater than the either of its child.

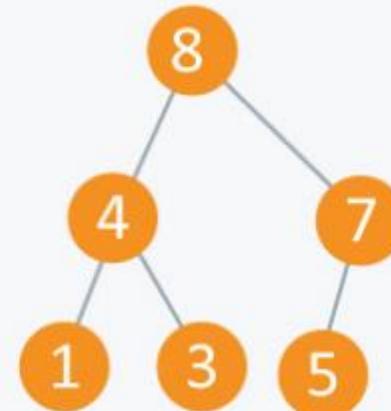
Max Heap: Example

Arr	4	3	7	1	8	5
	0	1	2	3	4	5

Initial Elements



Max Heap



Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- Step 1: First we add the 44 element in the tree as shown below:

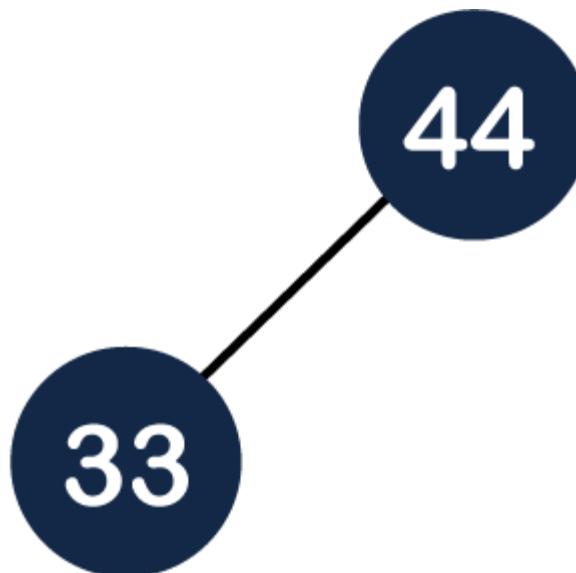


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:

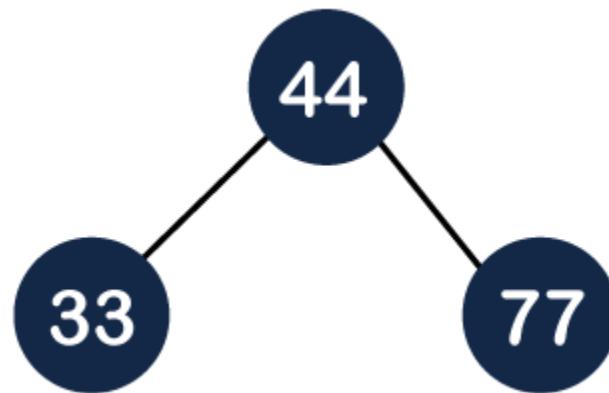


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:

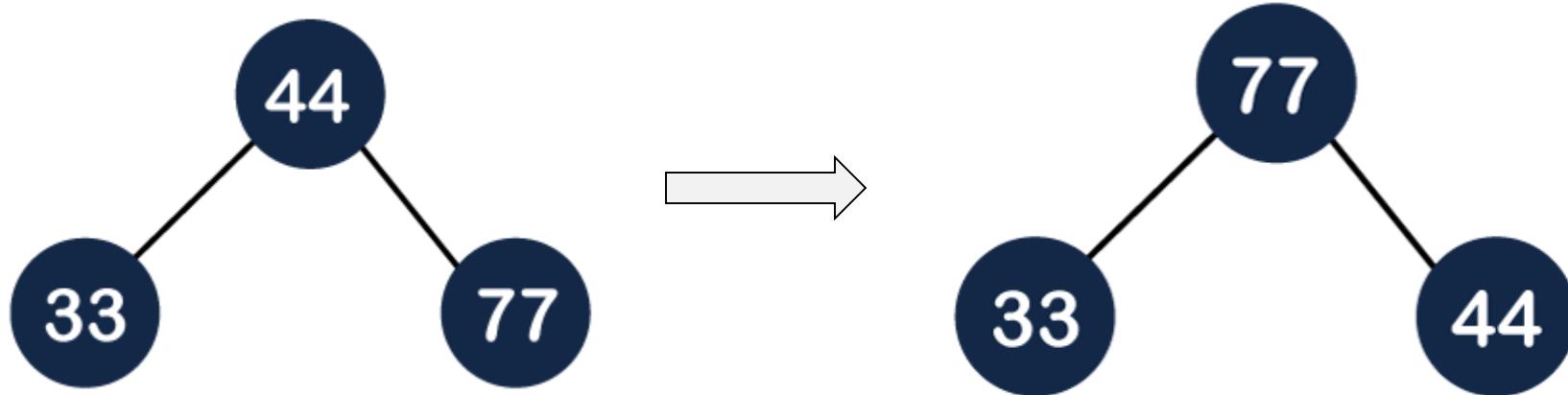


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- Tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:

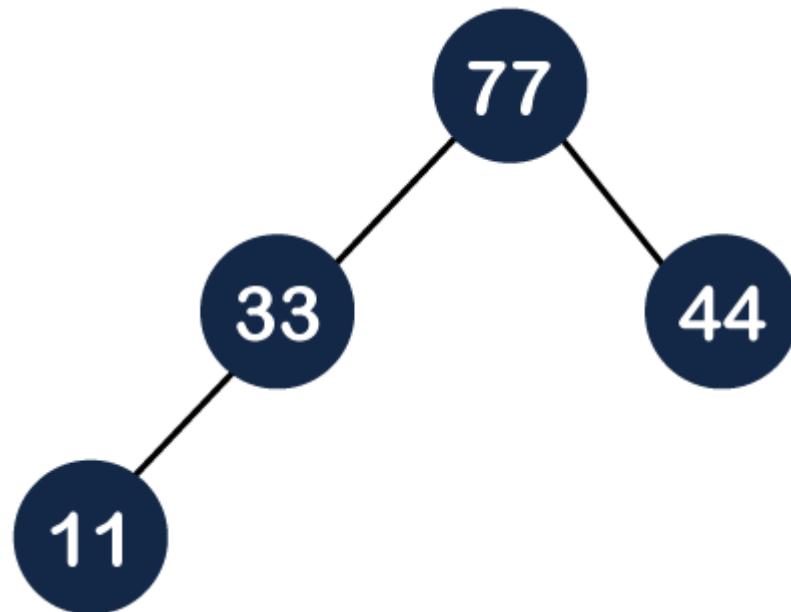


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:

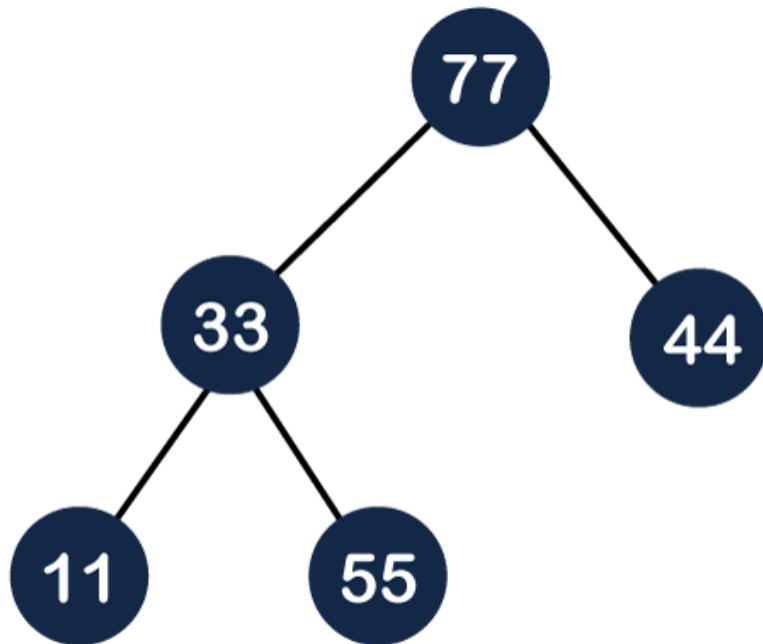


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:

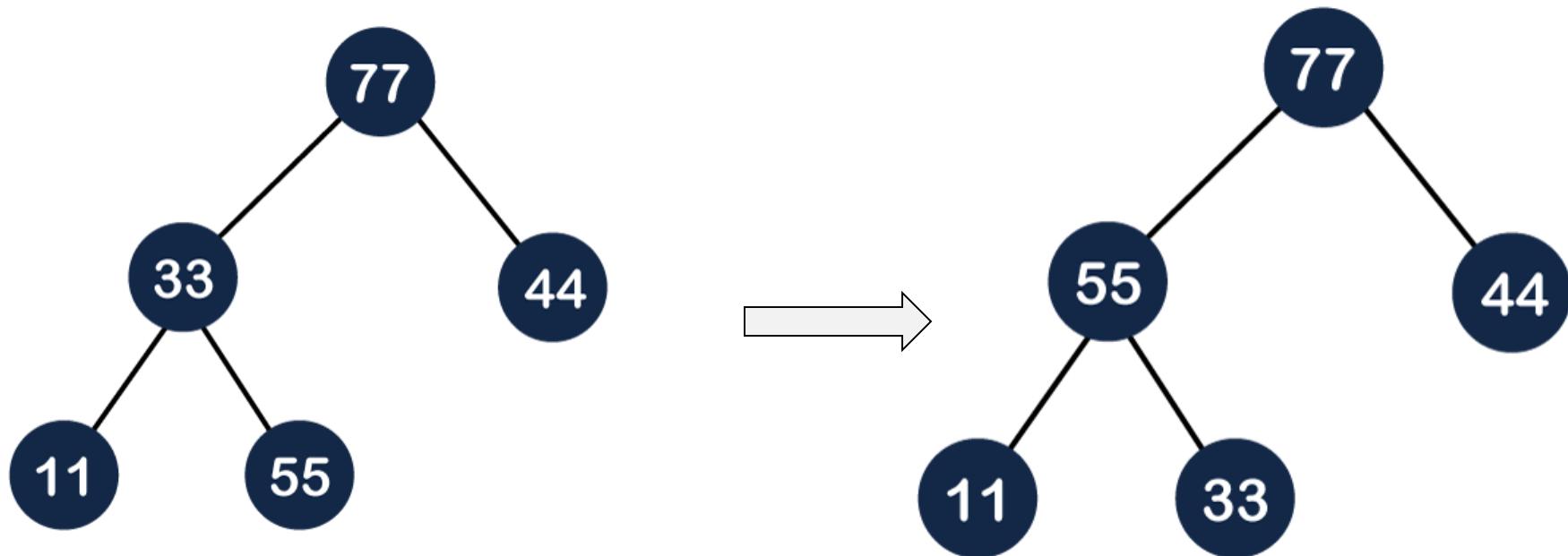


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- it does not satisfy the property of the max heap because $33 < 55$, so we will swap these two values as shown below:

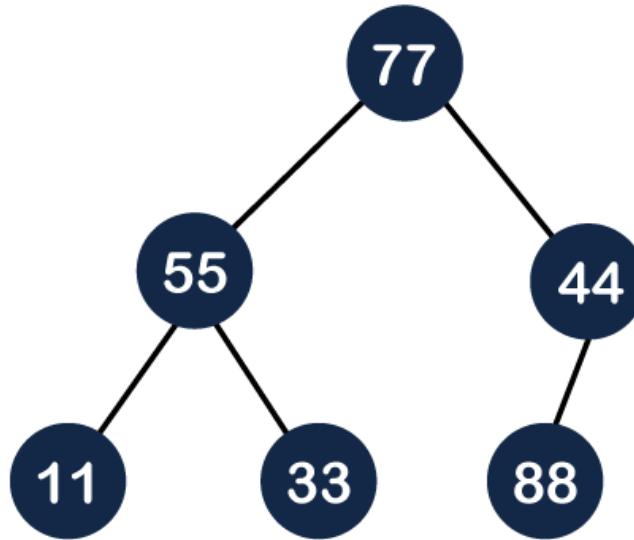


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:

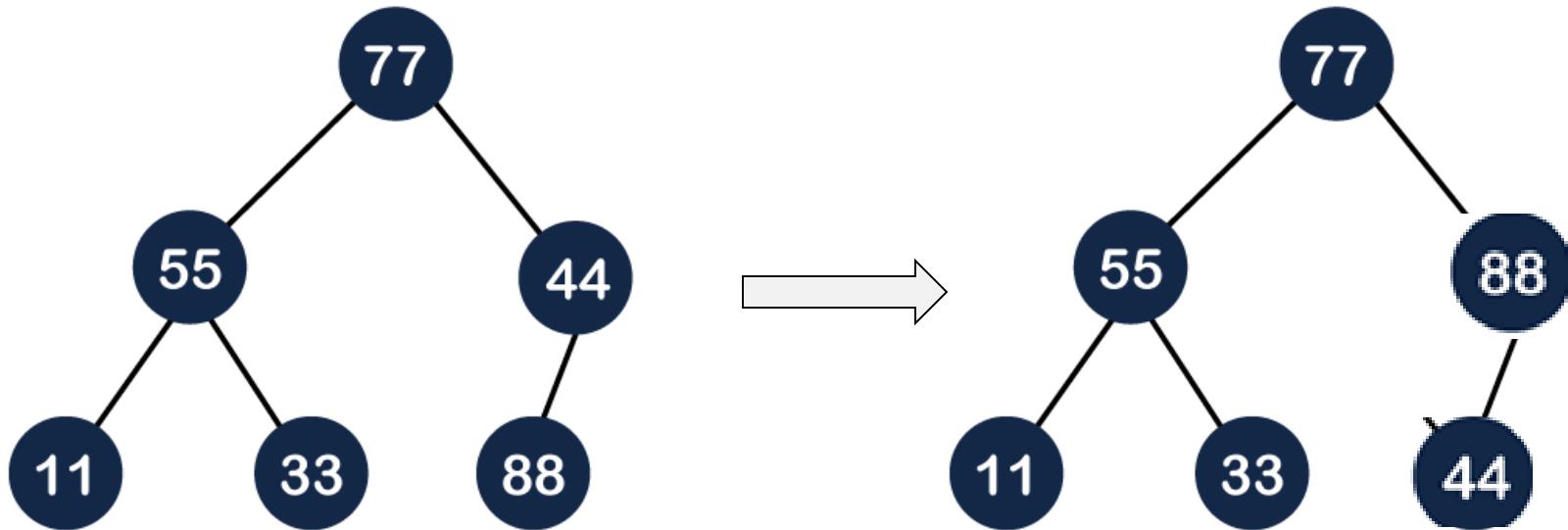


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

- it does not satisfy the property of the max heap because $44 < 88$, so we will swap these two values as shown below:

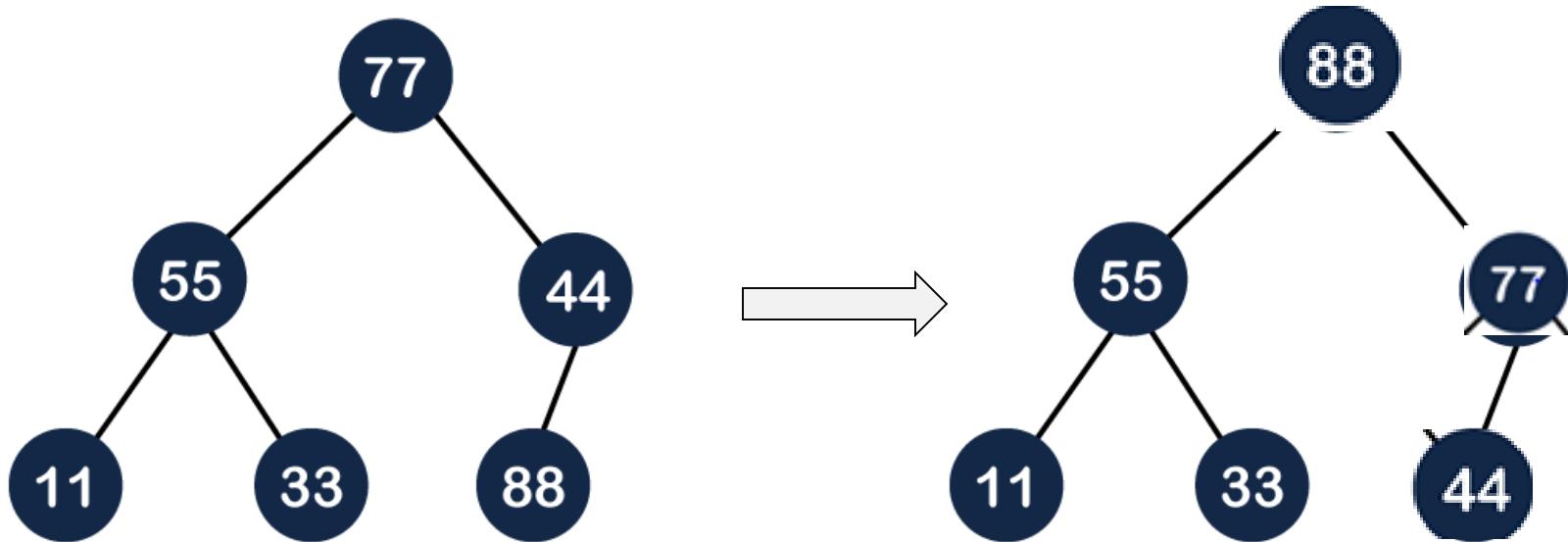


Insert: Example

Insertion in the Heap tree

44, 33, 77, 11, 55, 88

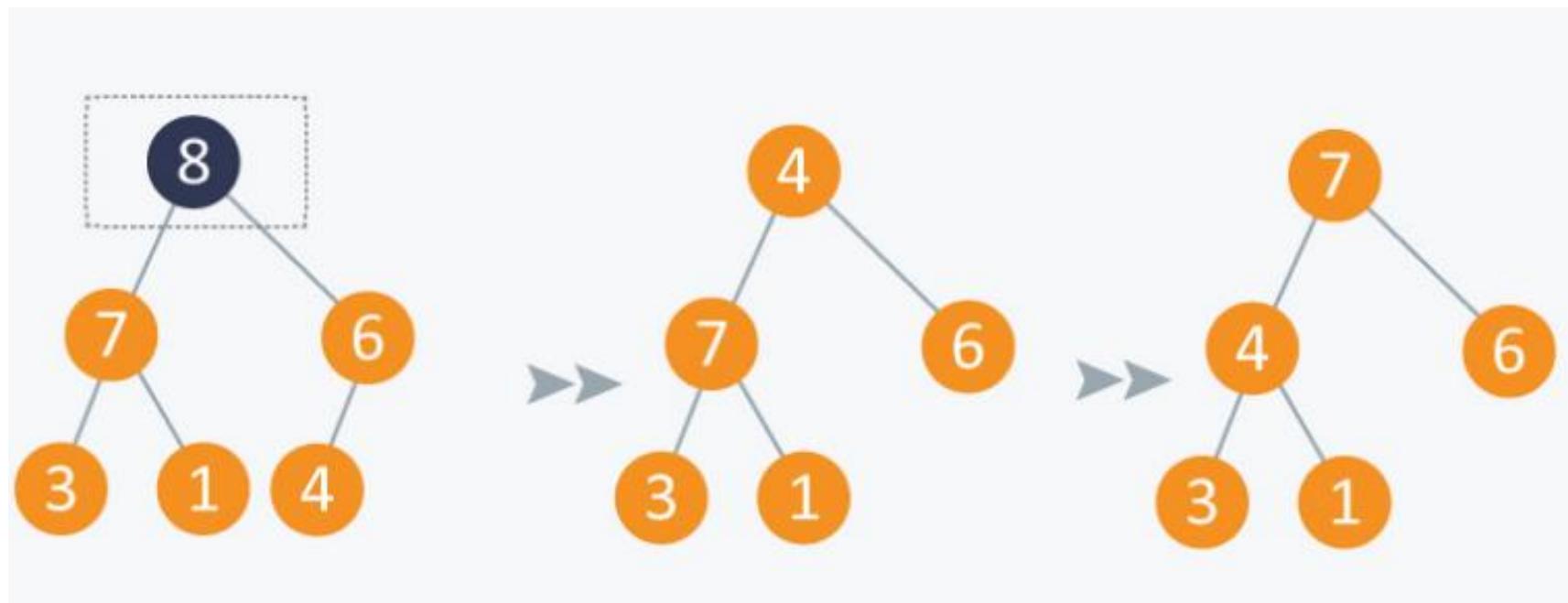
- Again, it is violating the max heap property because $88 > 77$ so we will swap these two values as shown below:



Delete Operation: Heap

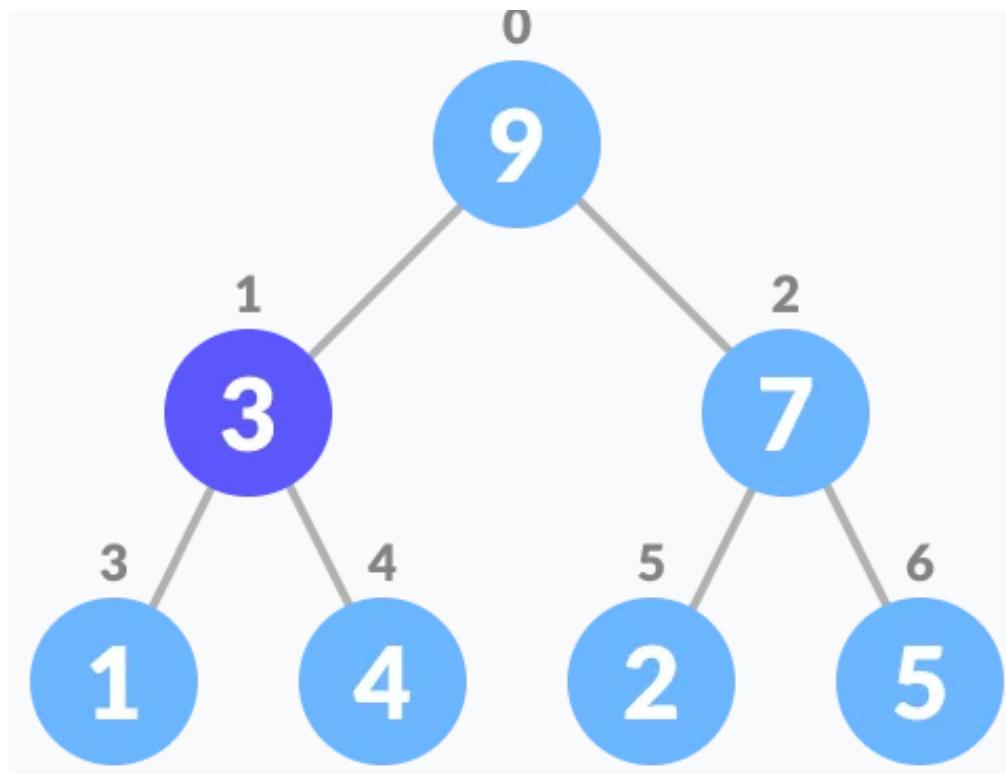
- Step 1 – Remove root node.
- Step 2 – Move the last element of last level to root.
- Step 3 – Compare the value of this child node with its parent.
- Step 4 – If value of parent is less than child, then swap them.
- Step 5 – Repeat step 3 & 4 until Heap property holds.

Delete Operation: Heap



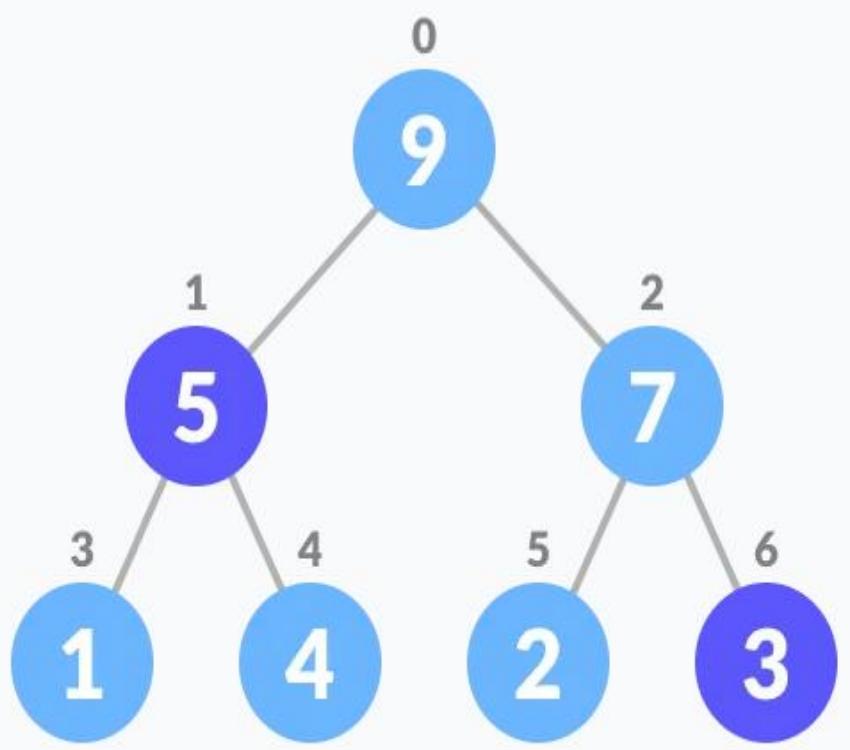
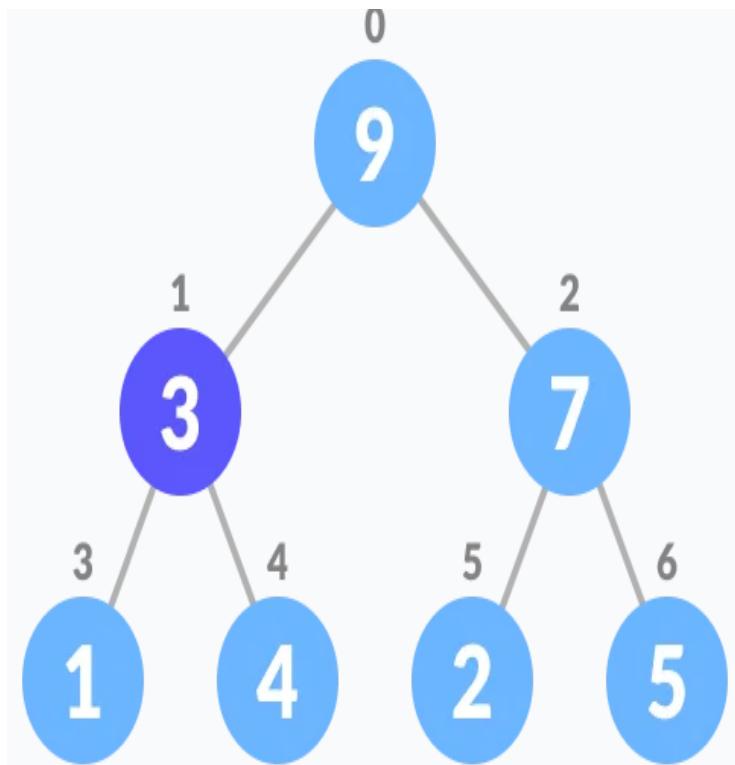
Delete Operation: Heap

Select the element to be deleted.



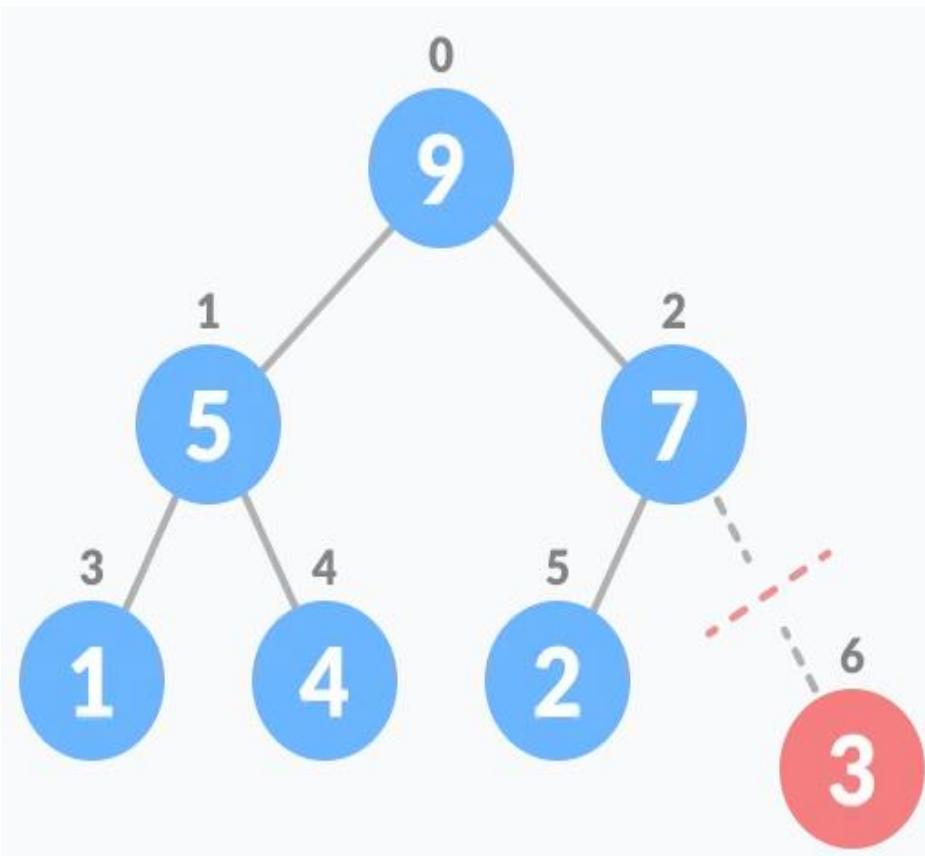
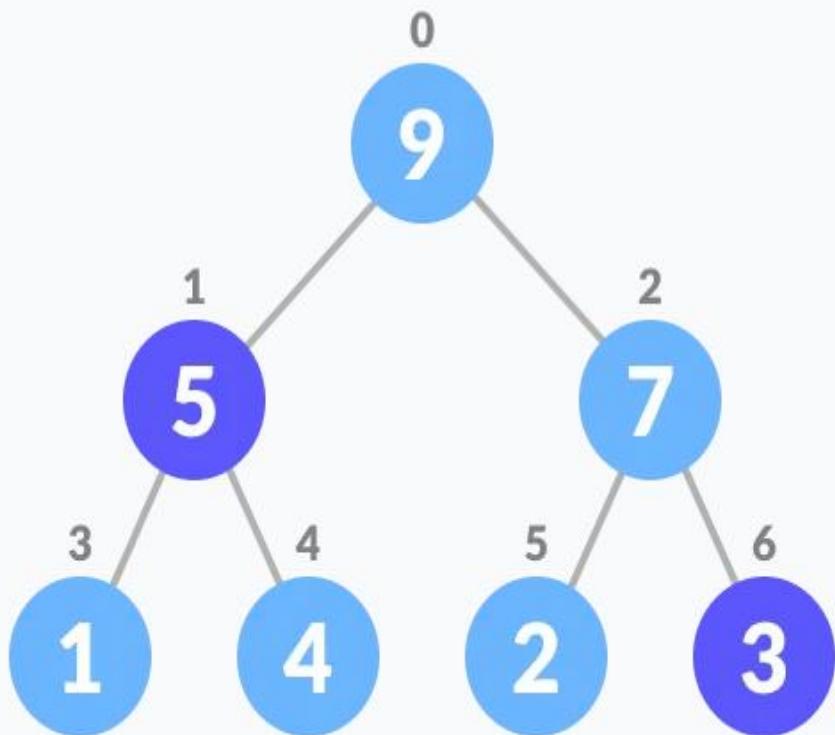
Delete Operation: Heap

Swap it with the last element.



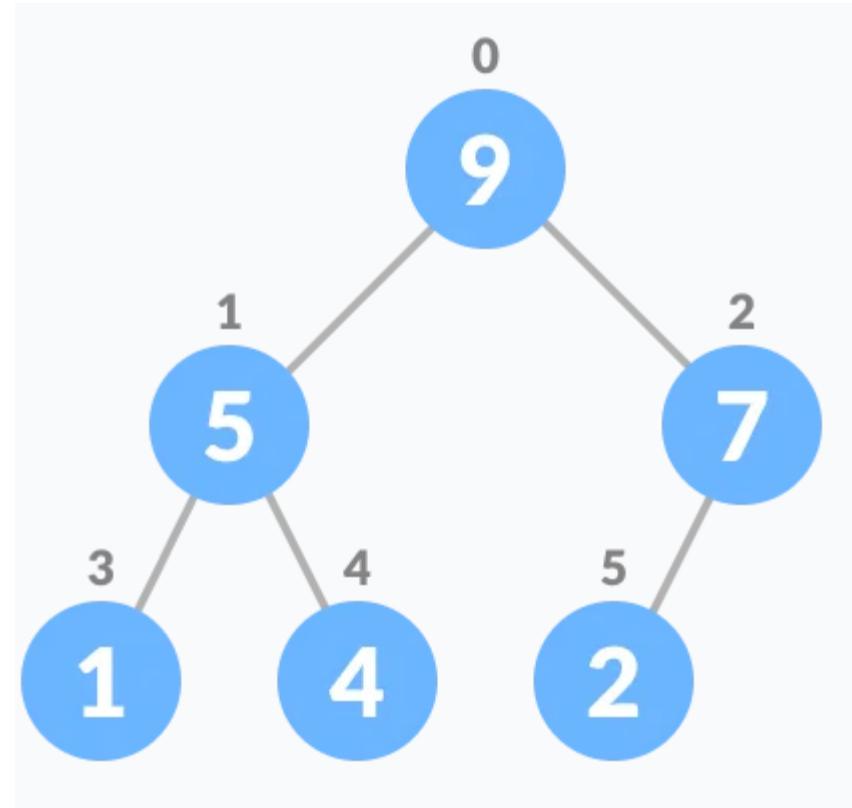
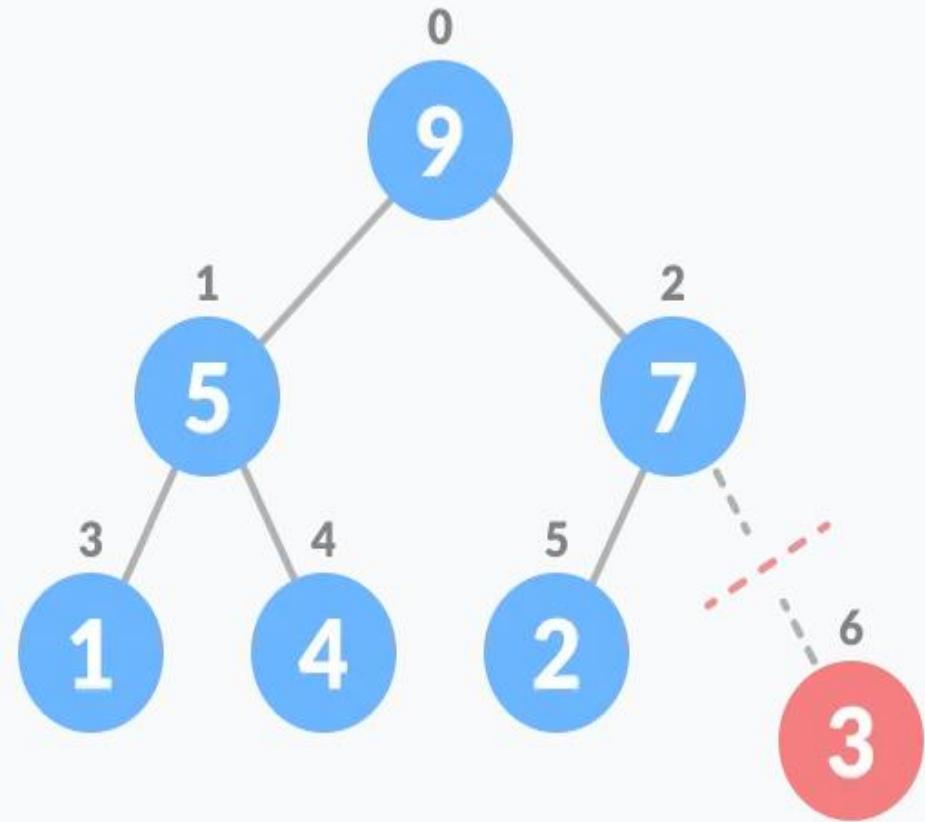
Delete Operation: Heap

Remove the last element



Delete Operation: Heap

Heapify the tree.





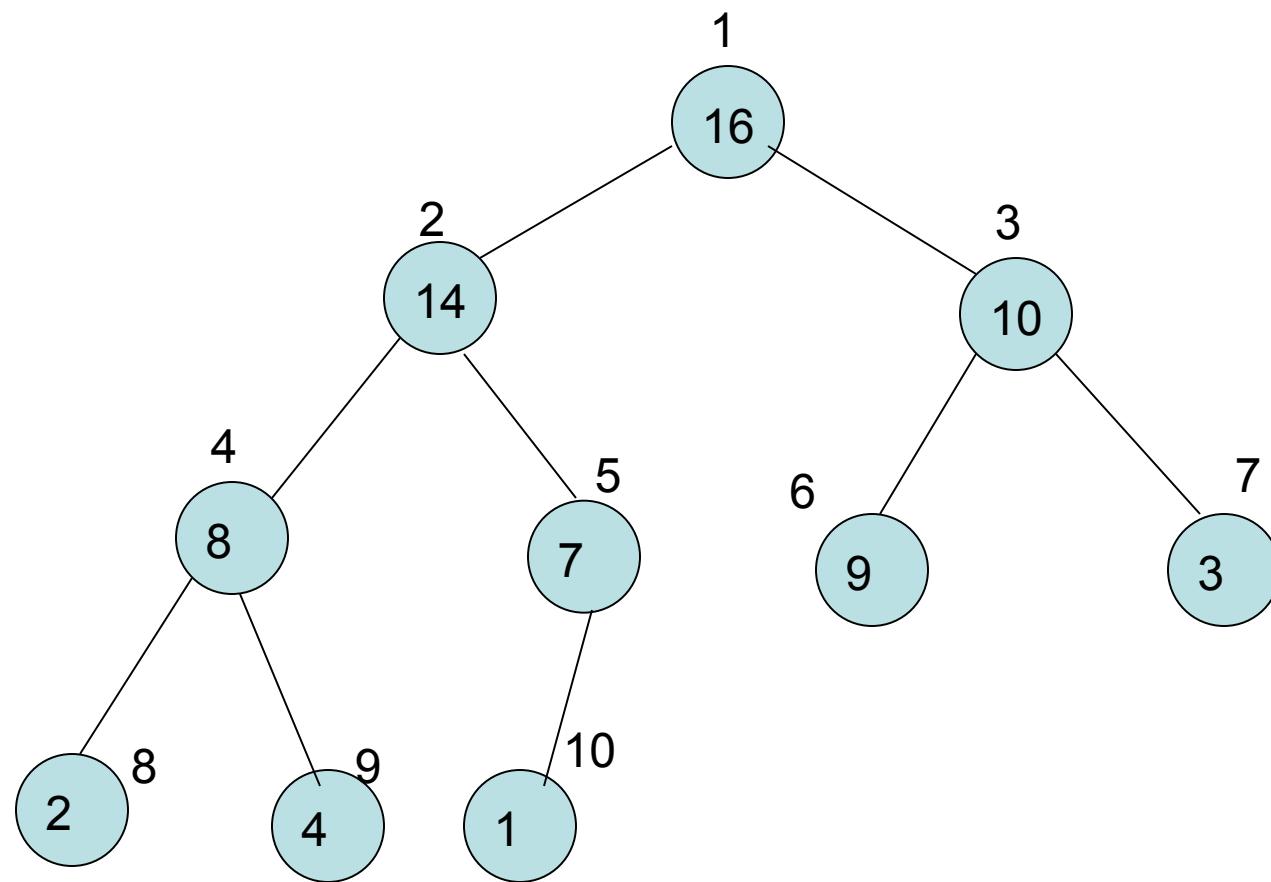
Summary

- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.

Heap Data Structure

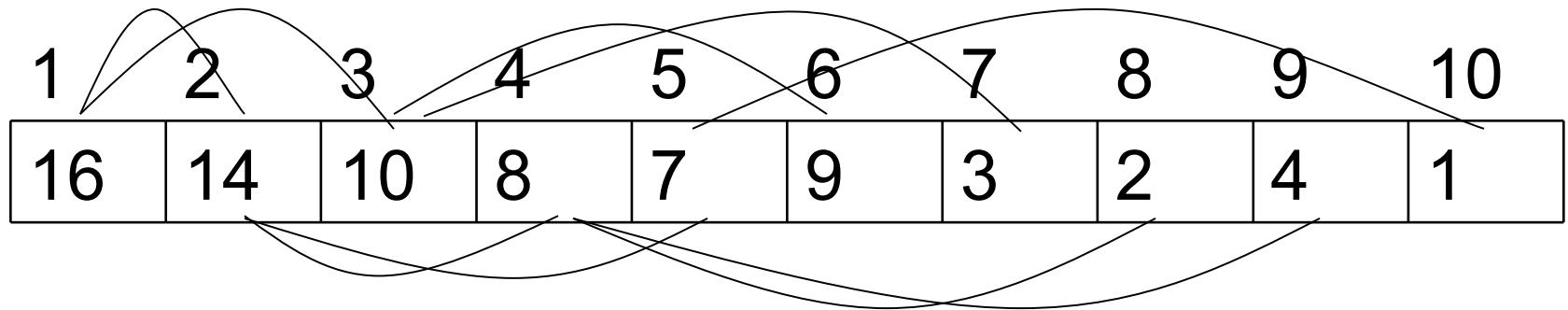
The heap data structure is an array object
that can be viewed as a nearly complete
binary tree.

Heap



(a)

Heap



(b)

Heap

There are two kinds of binary heaps

- max-heaps
- min heaps

Maintaining the Heap Property

The function of MAX-HEAPIFY is to let the value at $A[i]$ “float down” in the max-heap so that the sub-tree rooted at index i becomes a max-heap

Maintaining the Heap Property

MAX-HEAPIFY(A, i)

$l \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$

then $\text{largest} \leftarrow l$

else $\text{largest} \leftarrow i$

Maintaining the Heap Property

If $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$

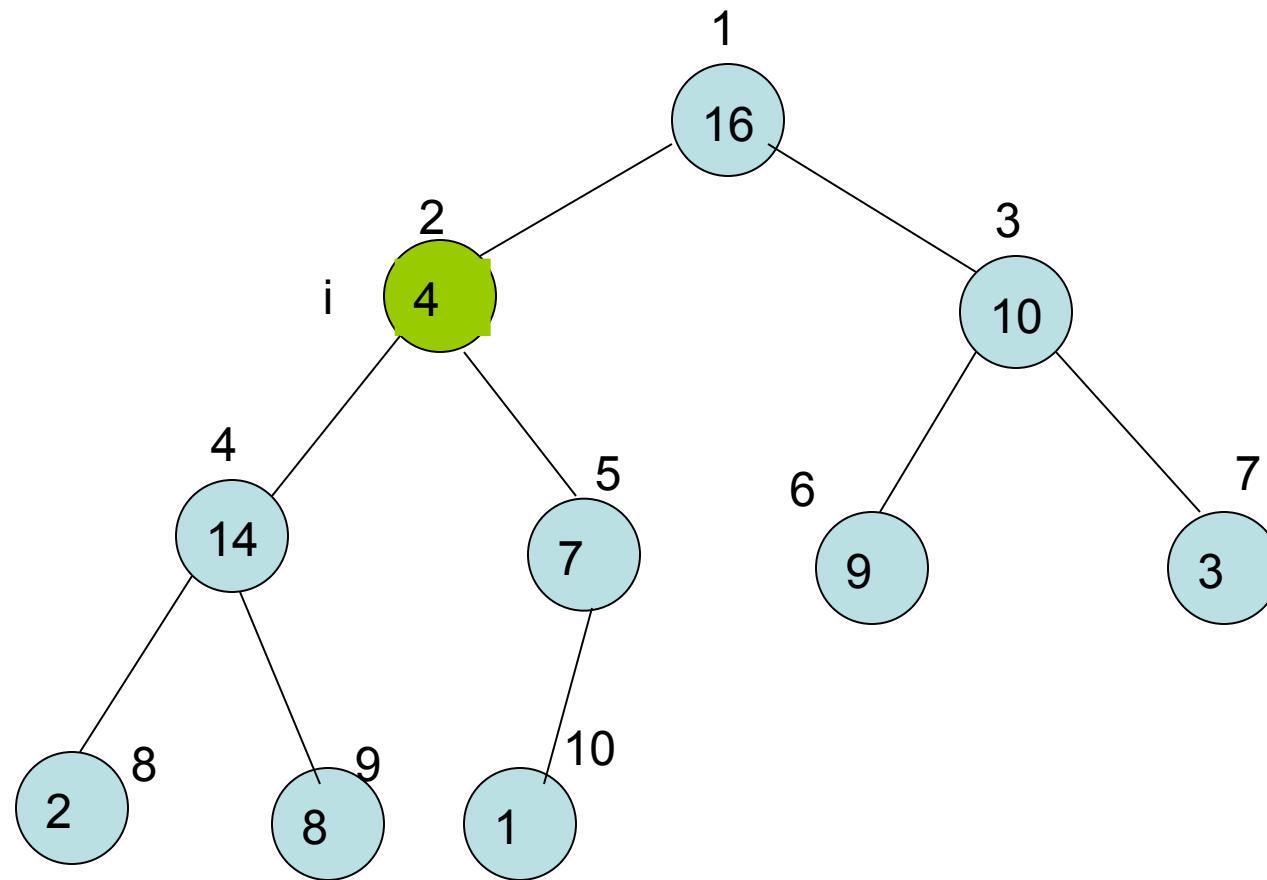
then $\text{largest} \leftarrow r$

If $\text{largest} \neq i$

then exchange $A[i] \leftrightarrow A[\text{largest}]$

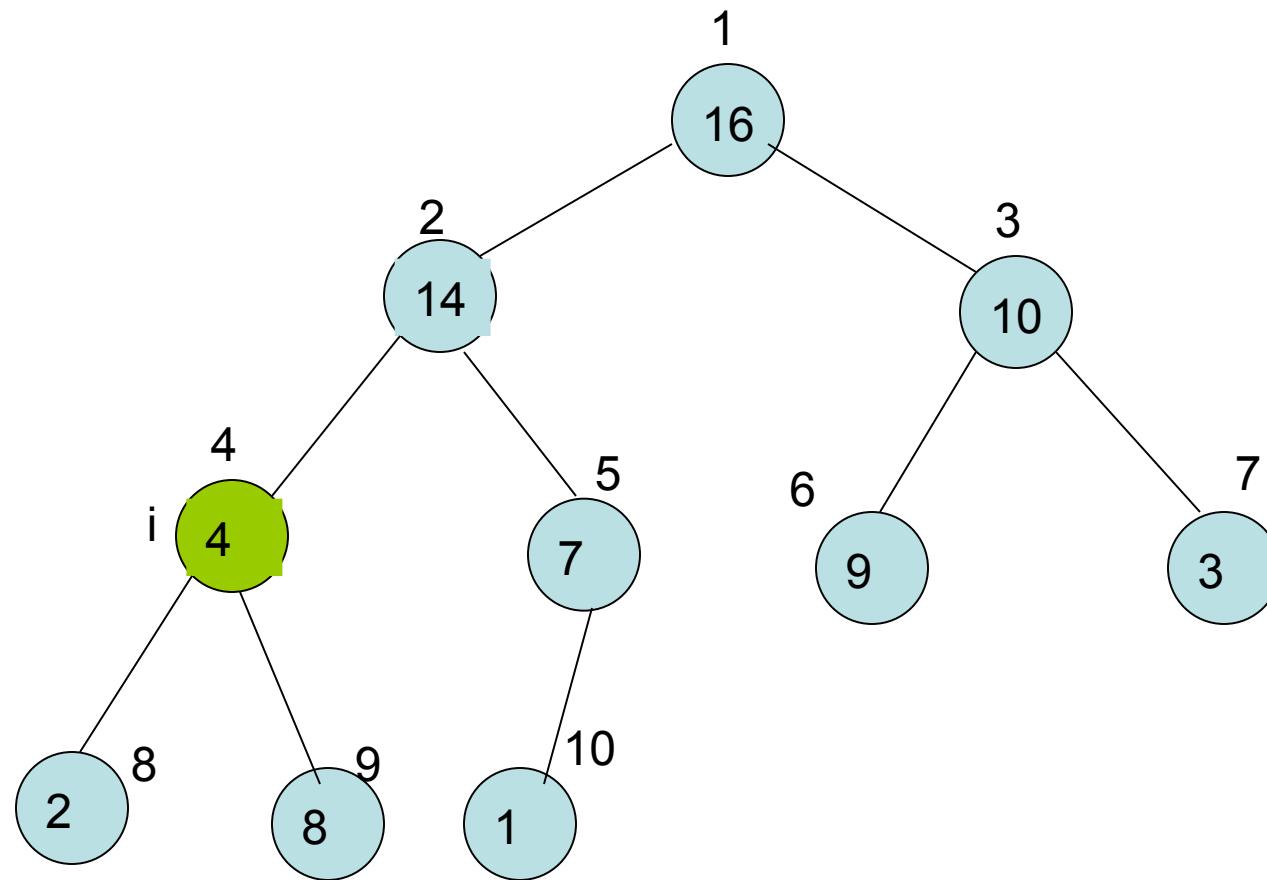
MAX-HEAPIFY($A, \text{largest}$)

Maintaining the Heap Property



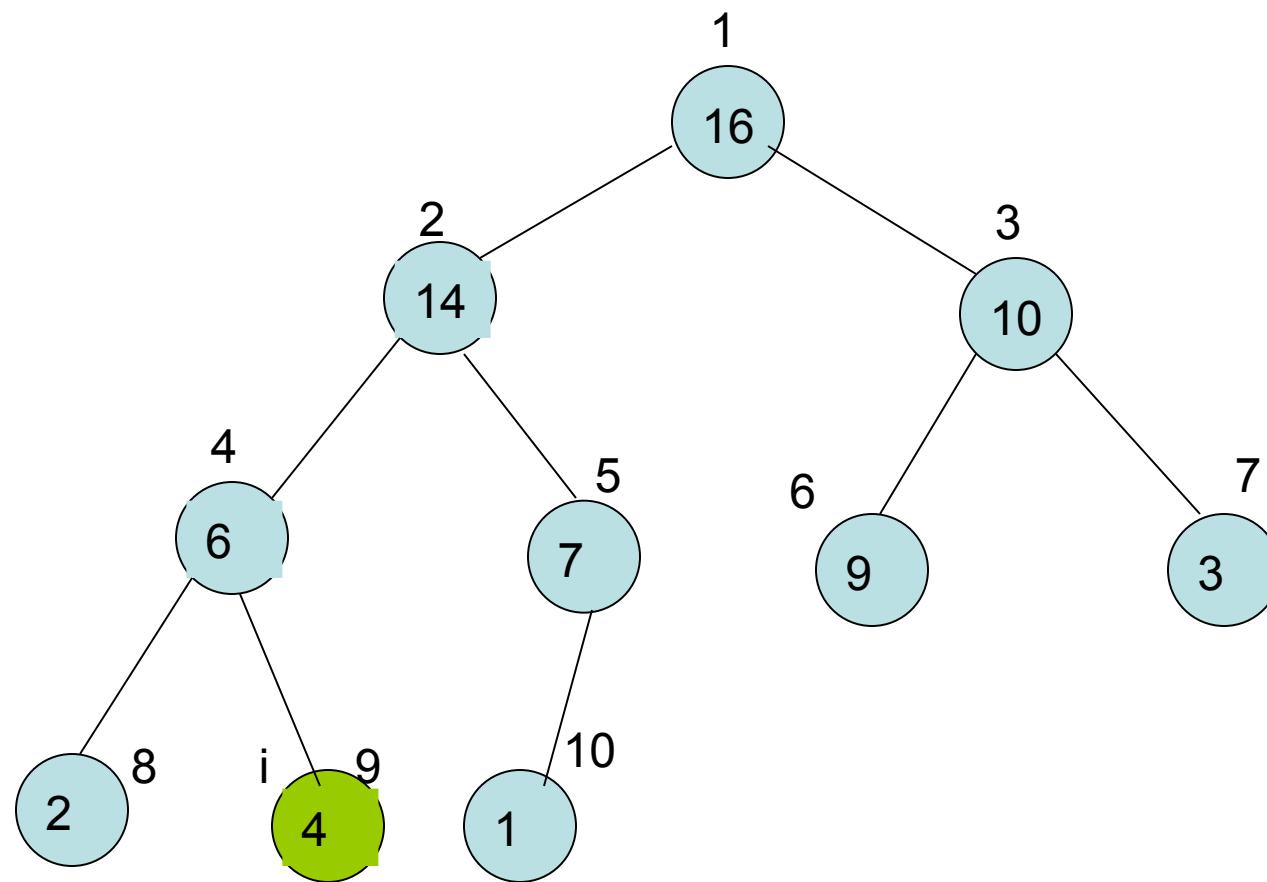
(a)

Maintaining the Heap Property



(b)

Maintaining the Heap Property



(c)

Building a Heap

BUILD-MAX-HEAP(A)

$\text{heap-size}[A] \leftarrow \text{length}[A]$

 for $i \leftarrow \text{length}[A]/2$ downto 1

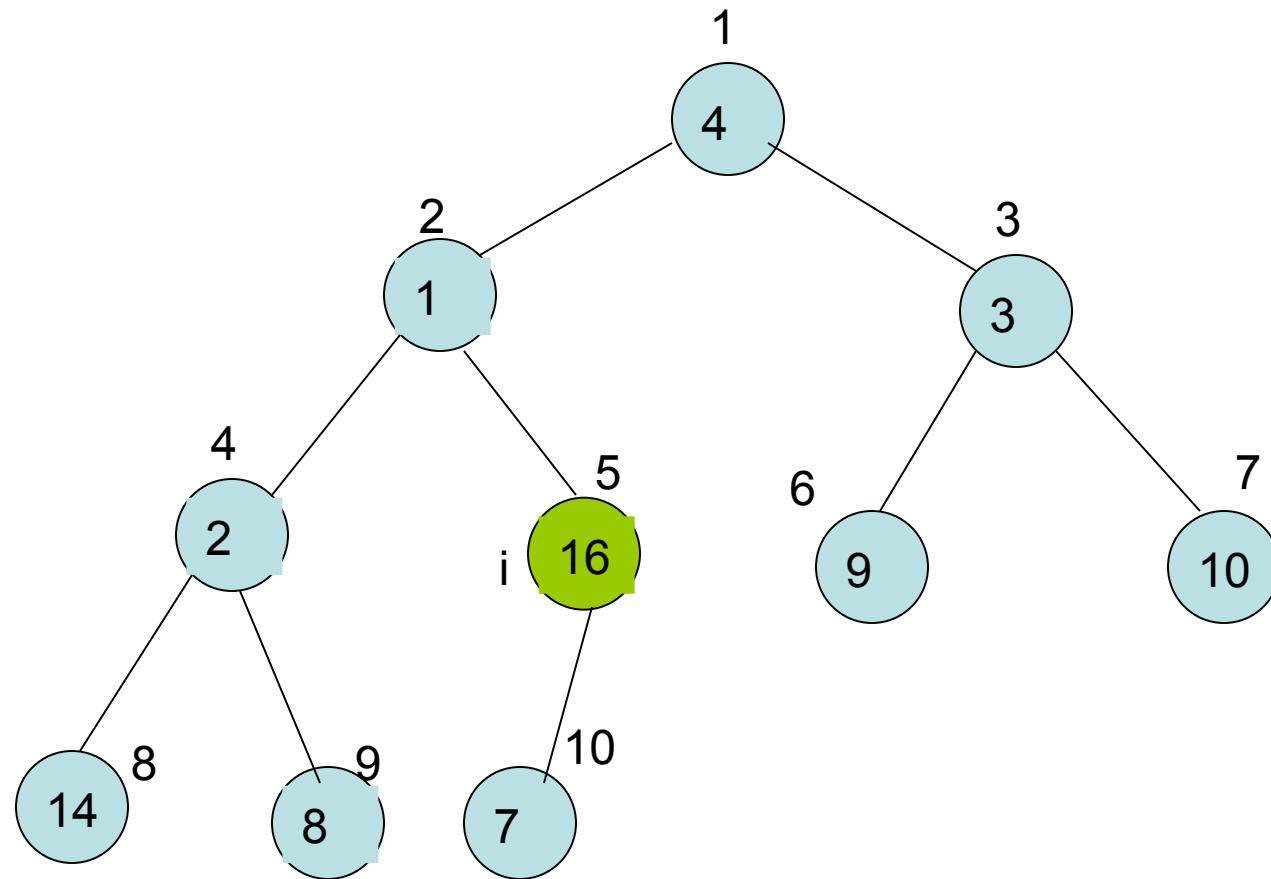
 do MAX-HEAPIFY(A, i)

Building a Heap

A

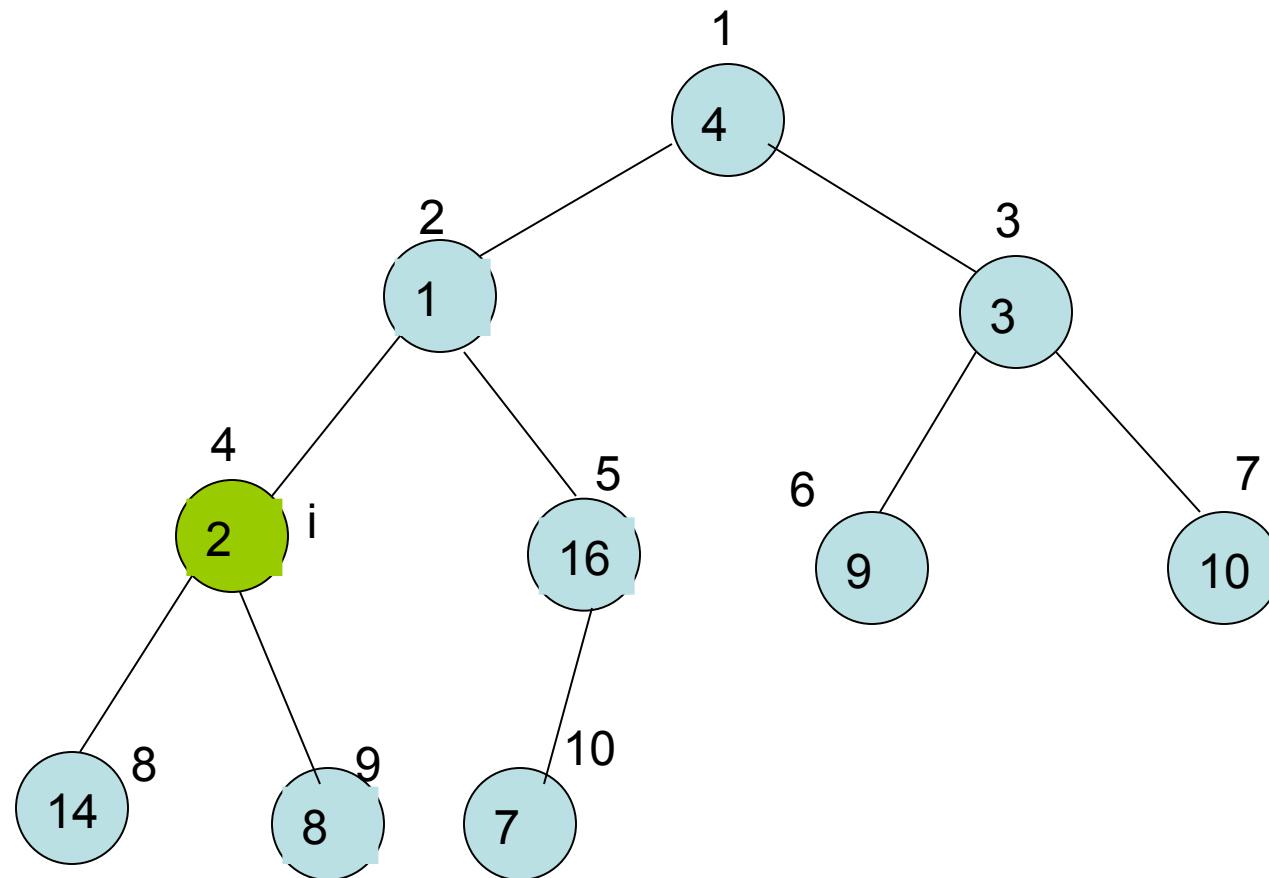
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

Building a Heap



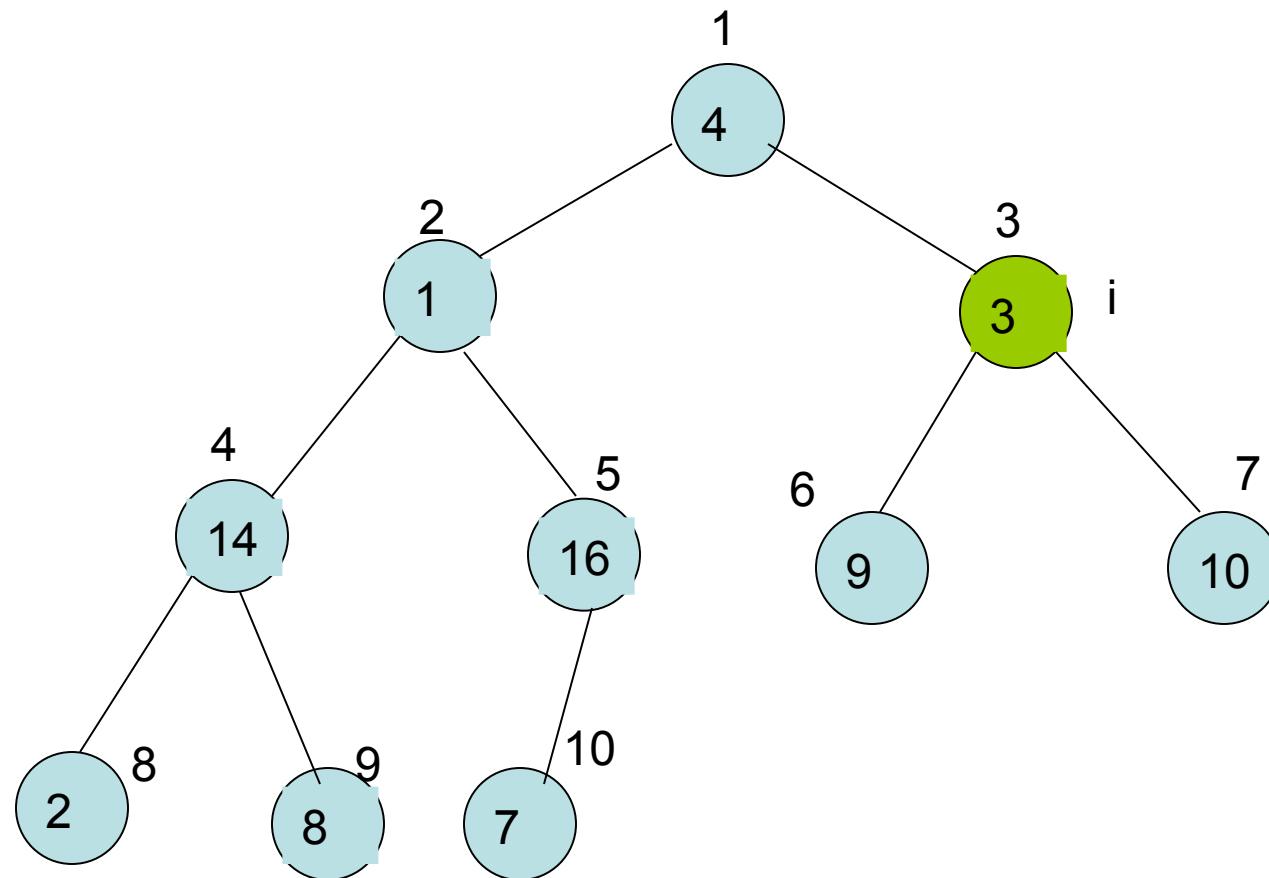
(a)

Building a Heap



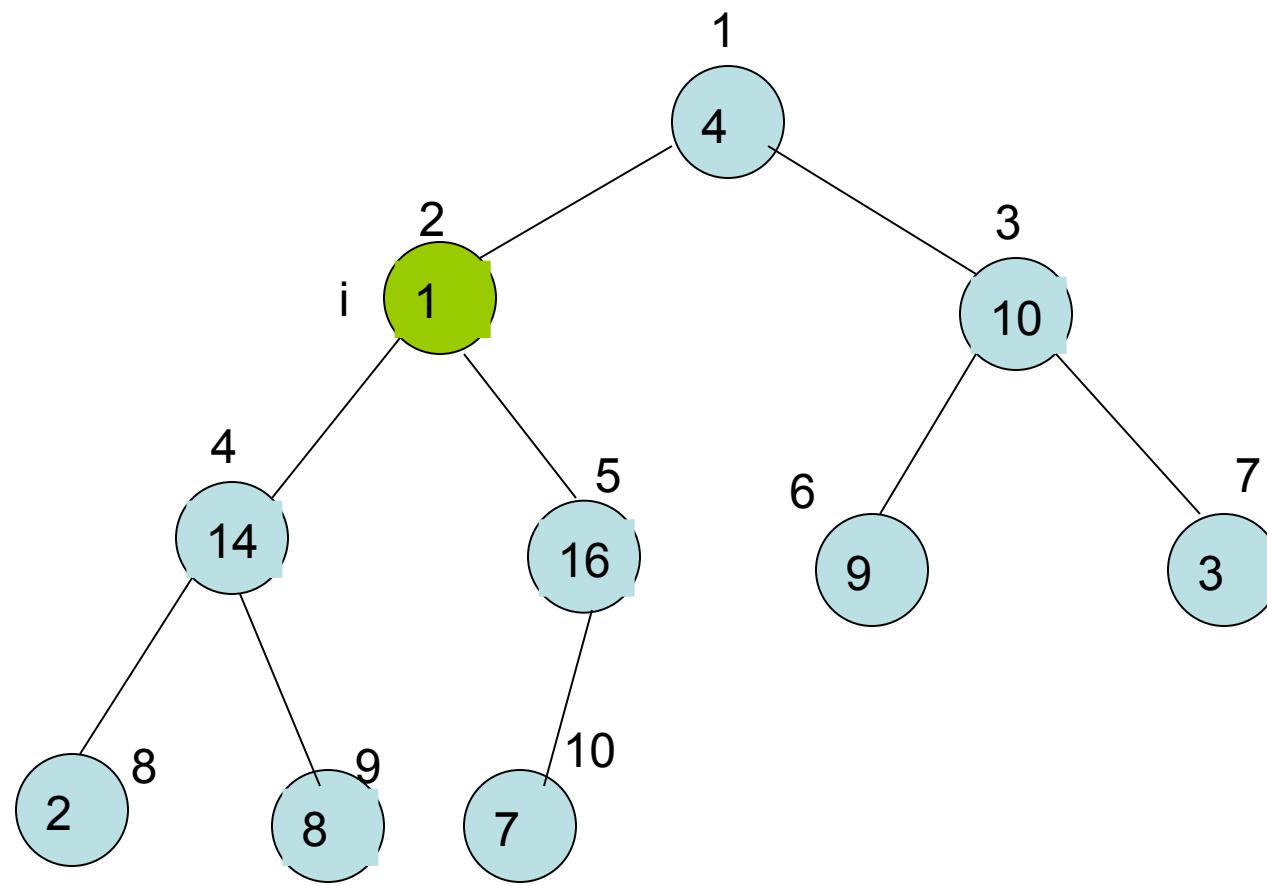
(b)

Building a Heap



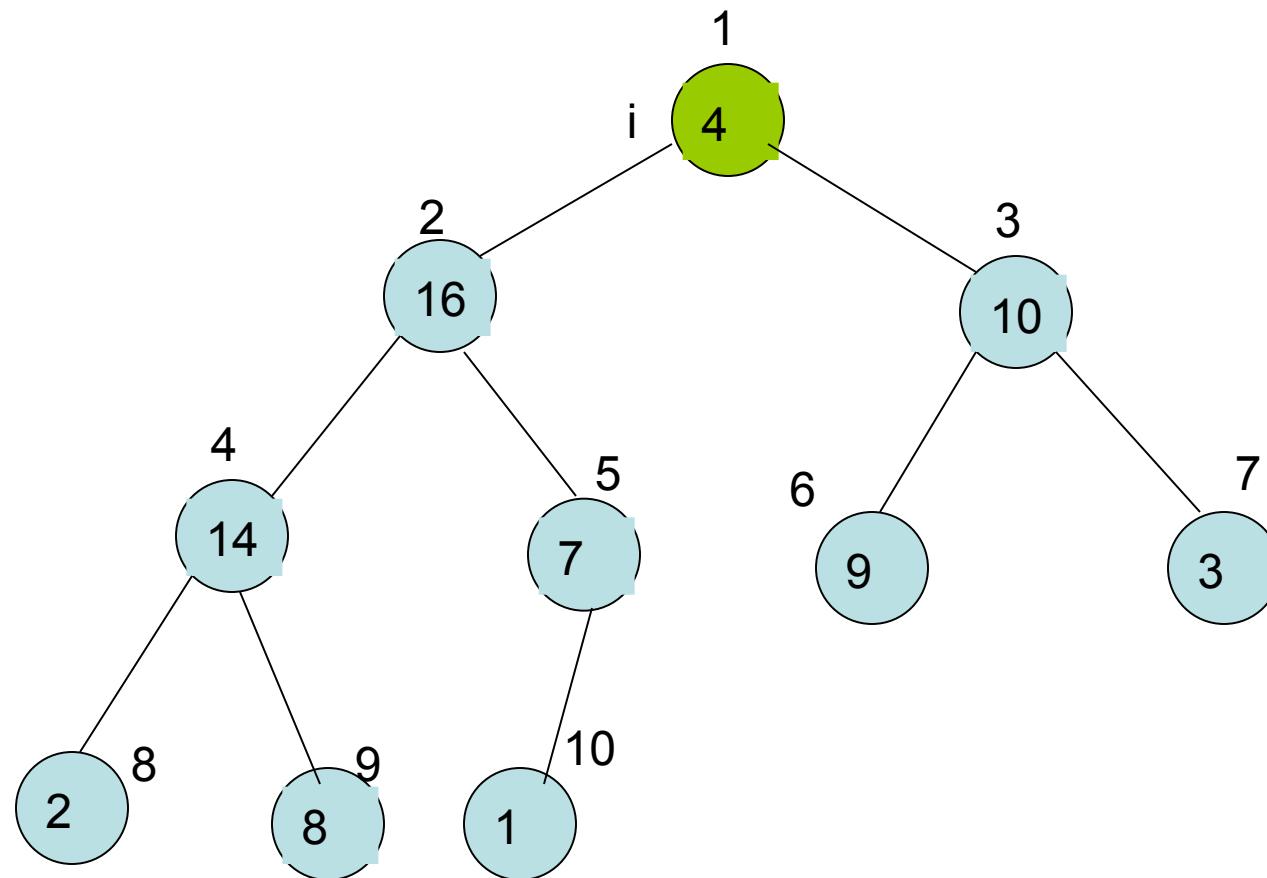
(c)

Building a Heap



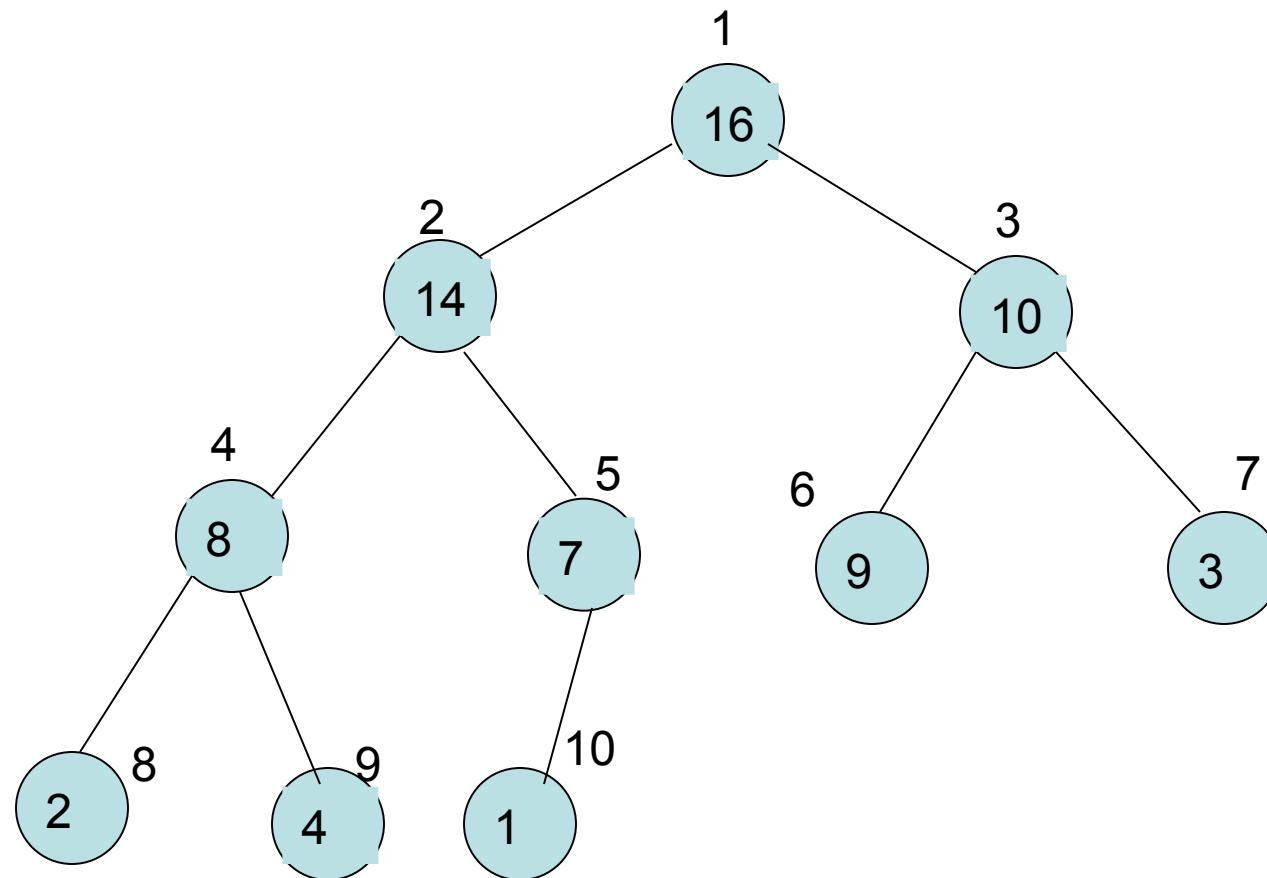
(d)

Building a Heap



(e)

Building a Heap



(f)

Heapsort Algorithm

HEAPSORT(A)

BUILD-MAX-HEAP(A)

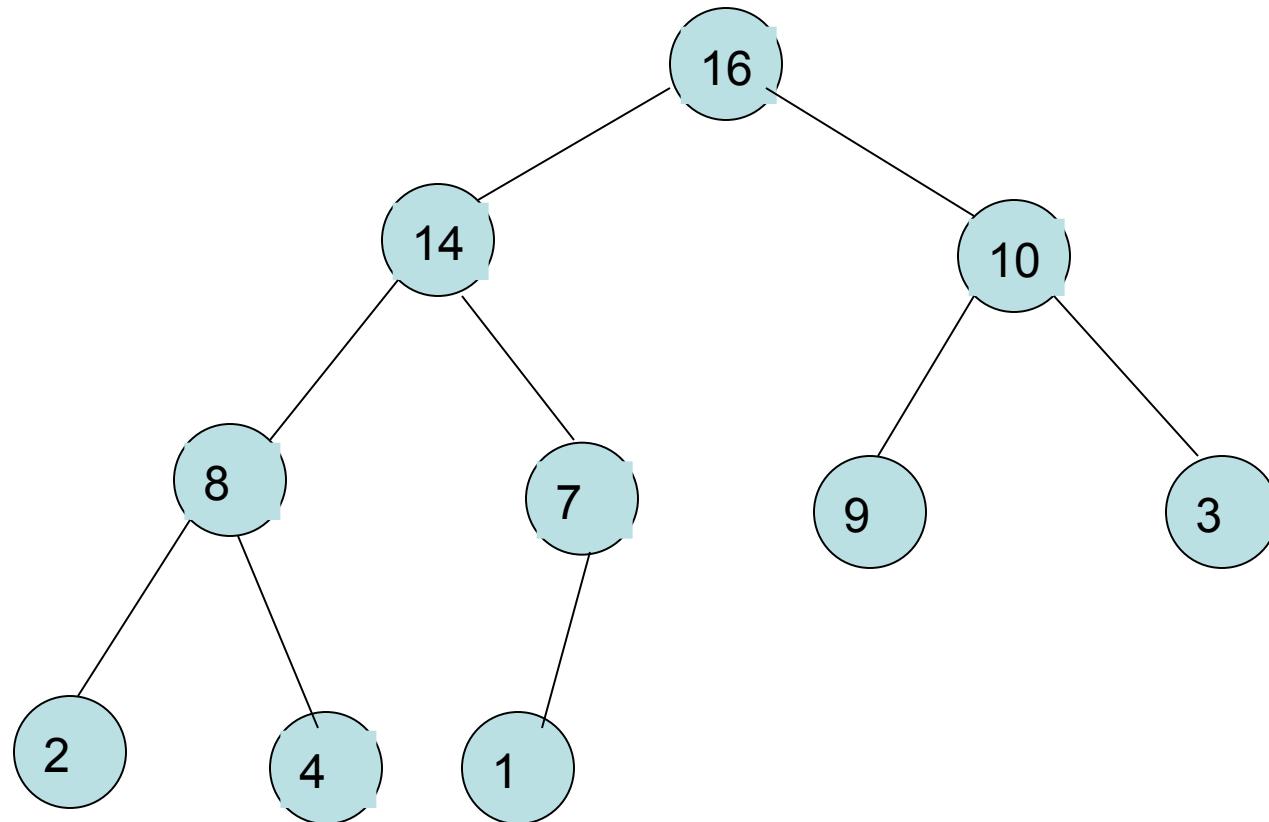
for $i \leftarrow \text{length}[A]$ downto 2

do exchange $A[1] \leftarrow \rightarrow A[i]$

$\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$

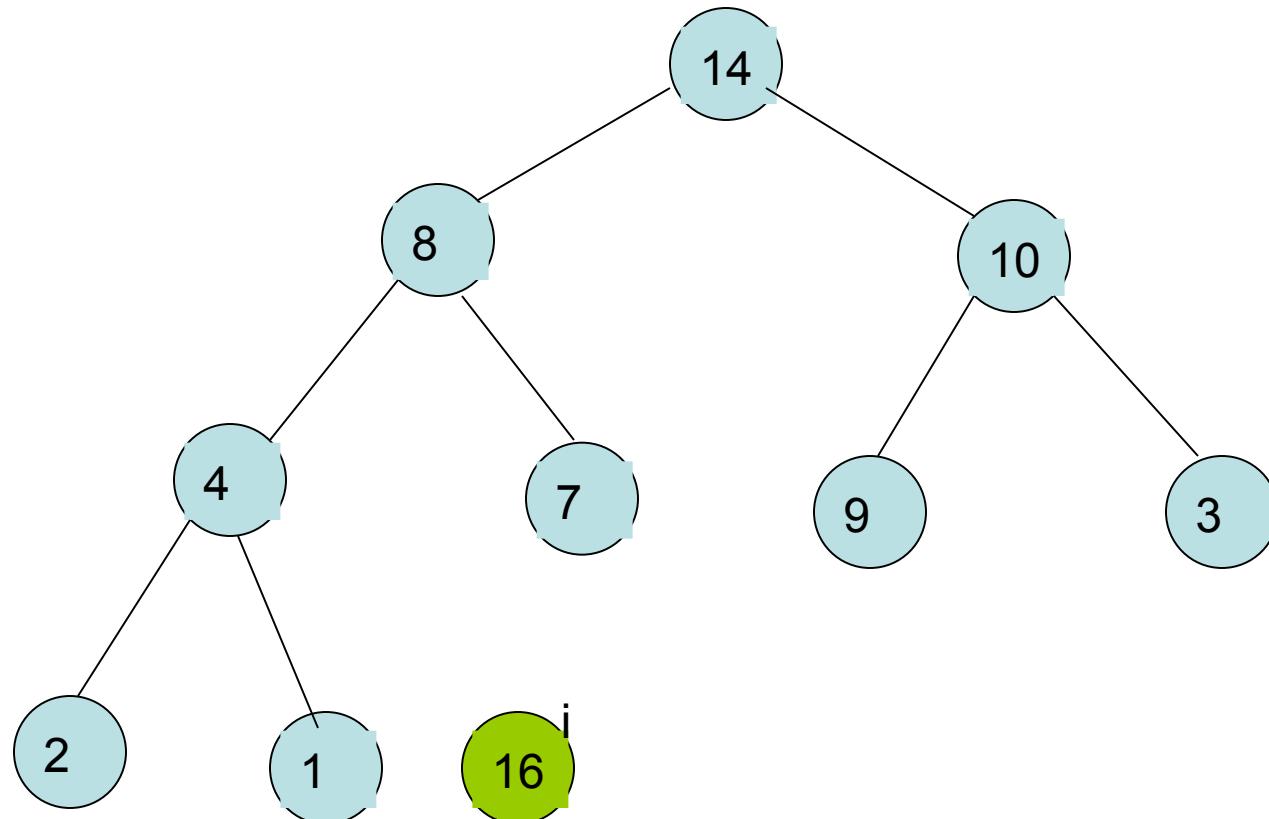
MAX-HEAPIFY(A,1)

Heapsort Algorithm



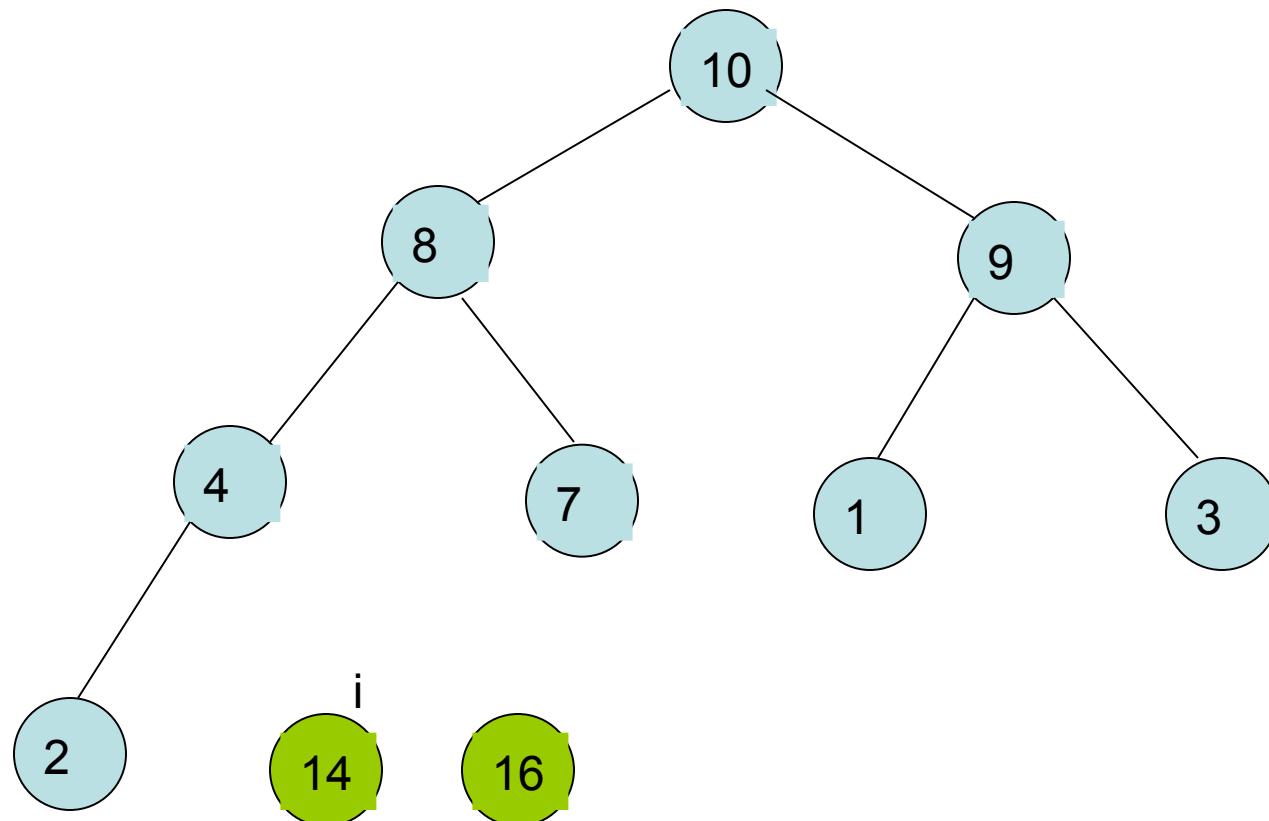
(a)

Heapsort Algorithm



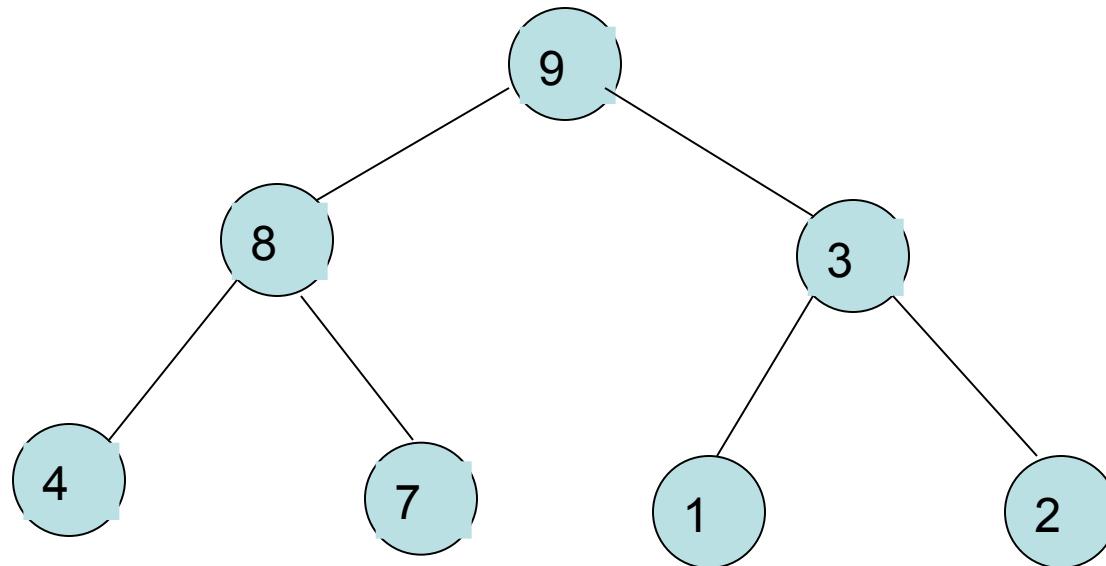
(b)

Heapsort Algorithm



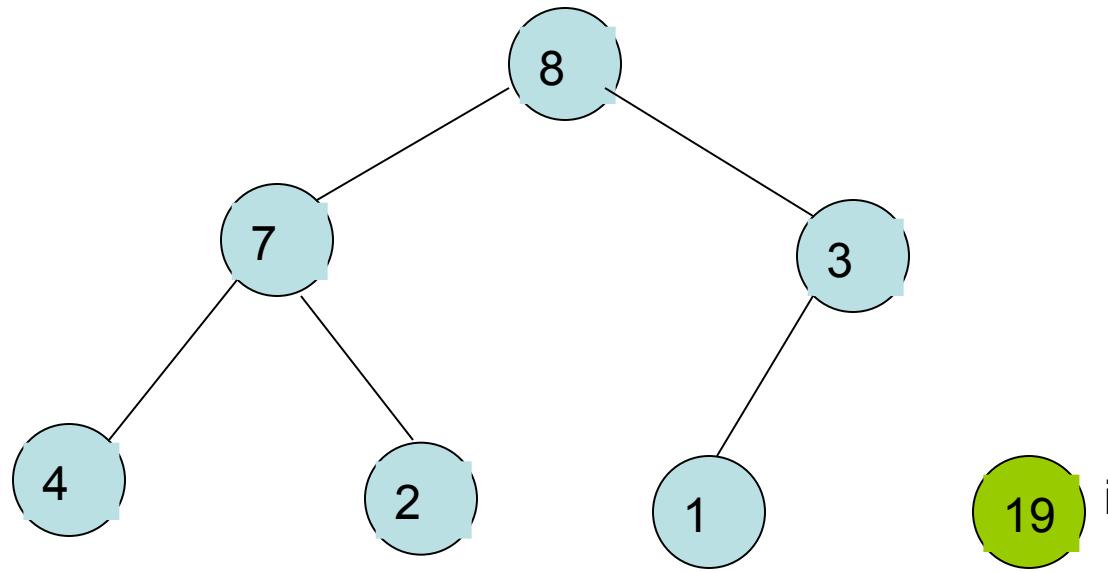
(c)

Heapsort Algorithm



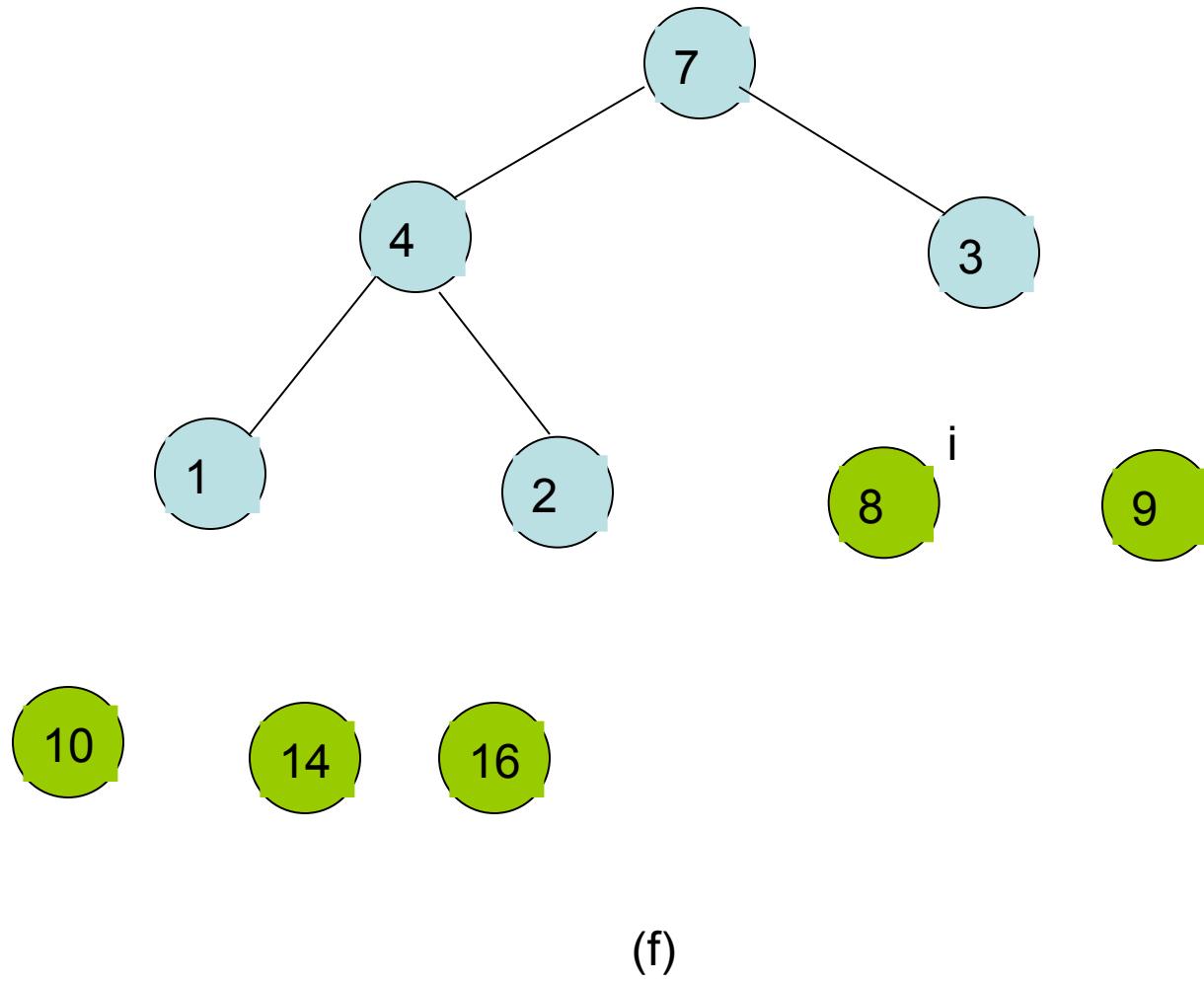
(d)

Heapsort Algorithm

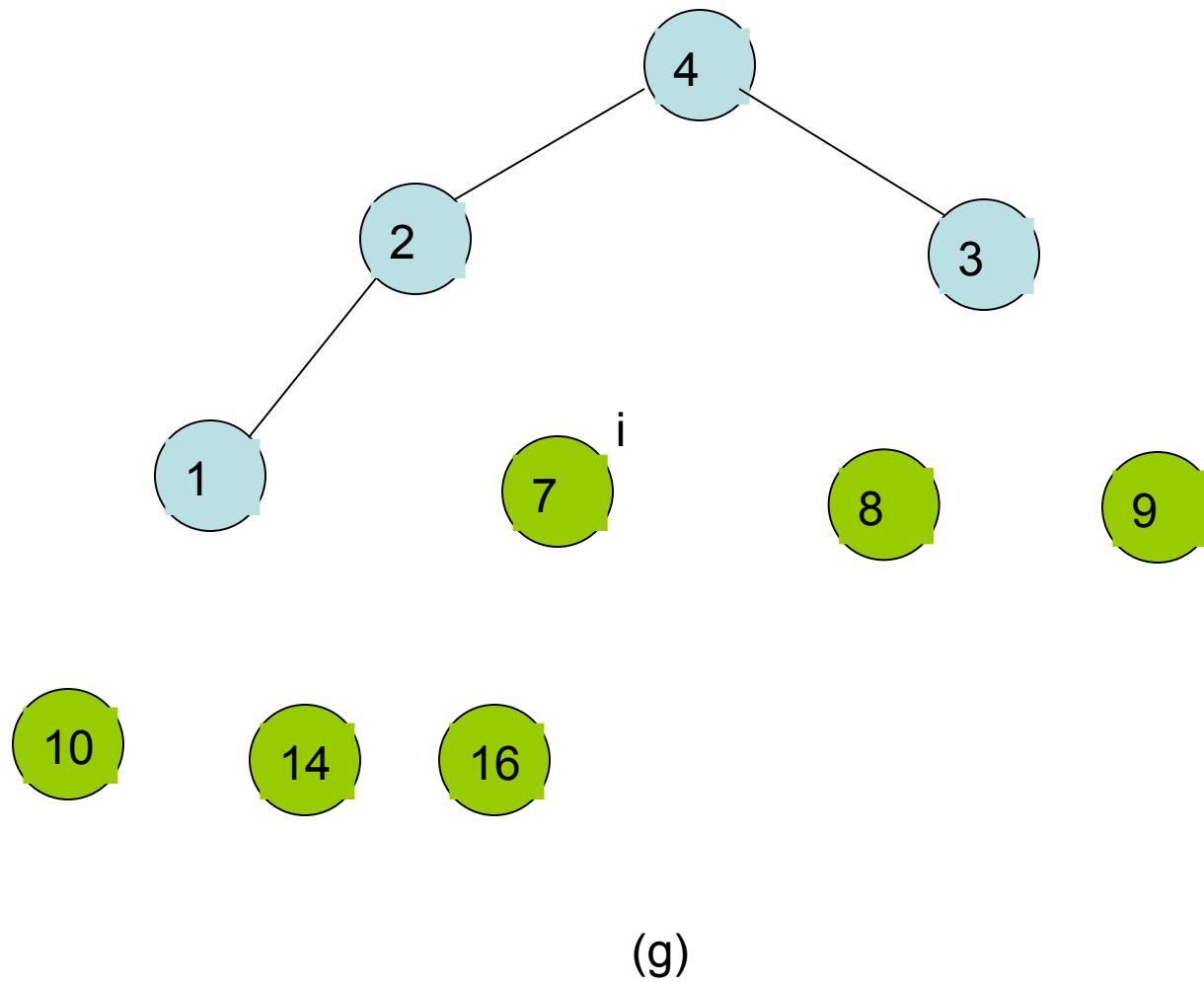


(e)

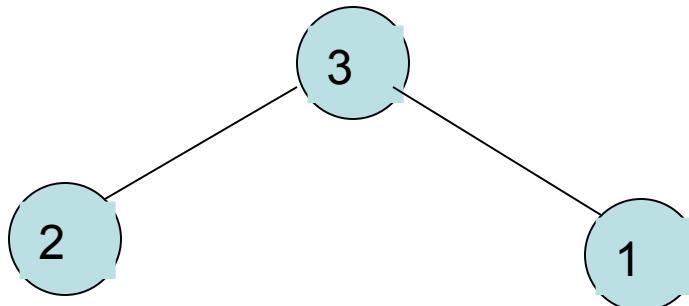
Heapsort Algorithm



Heapsort Algorithm

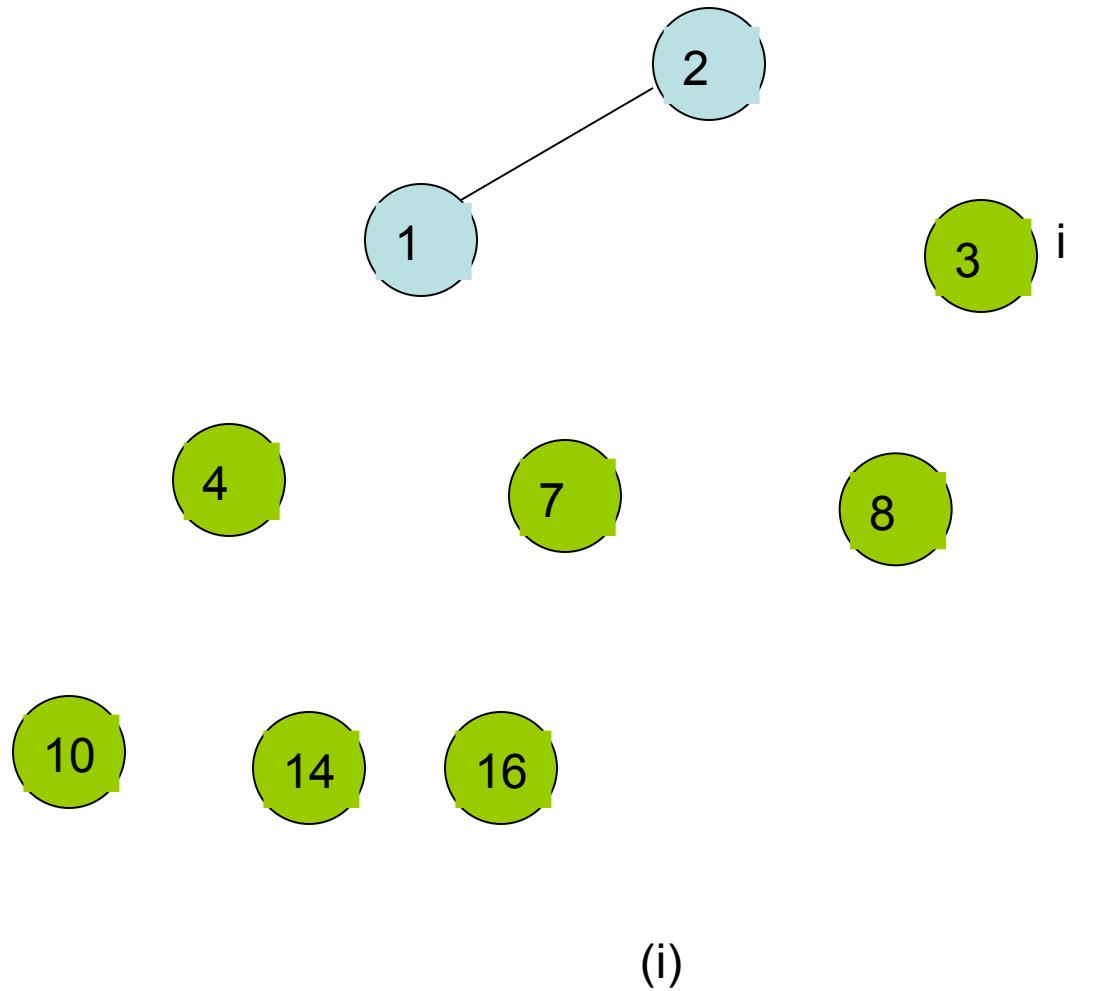


Heapsort Algorithm

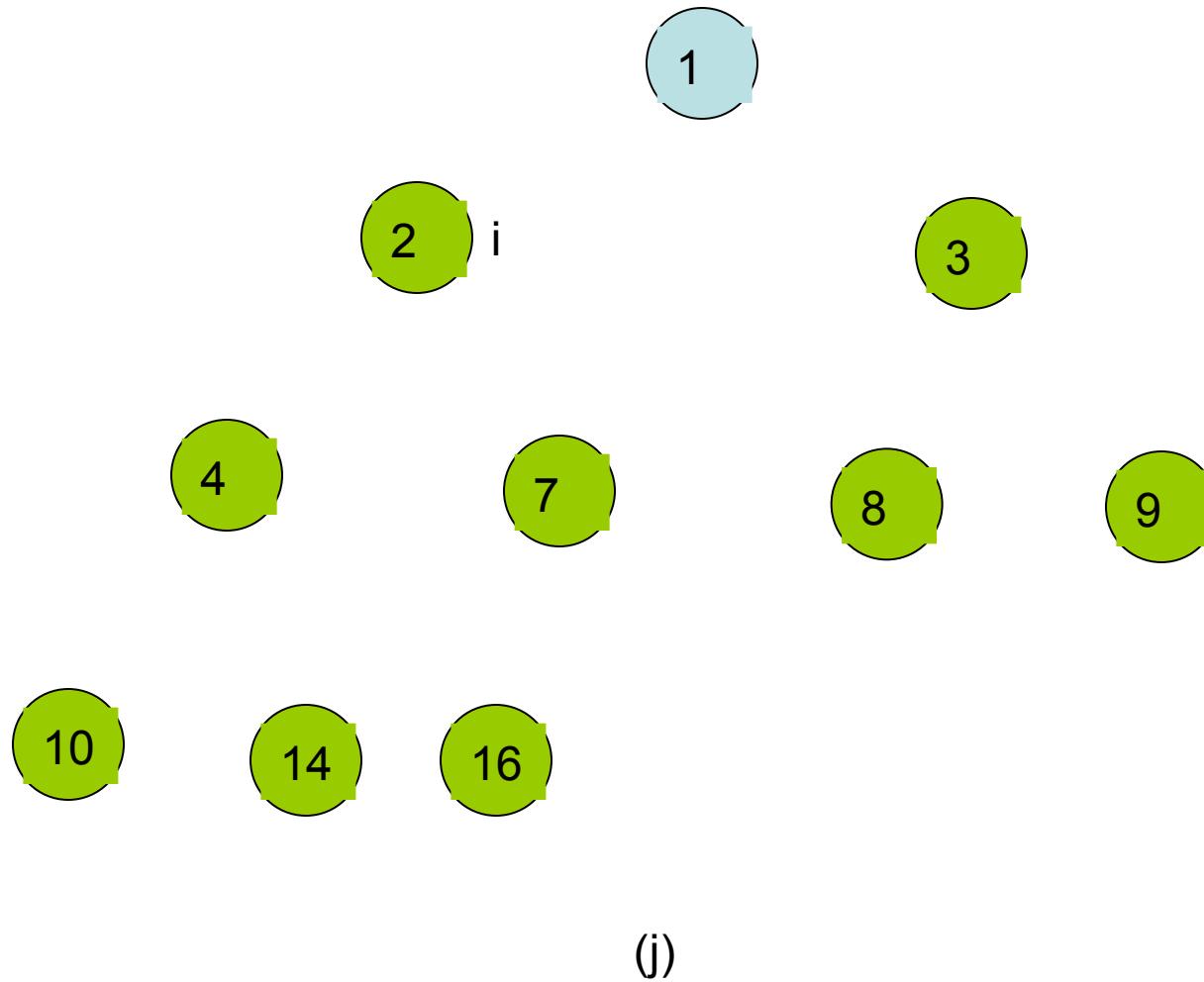


(h)

Heapsort Algorithm



Heapsort Algorithm



Heapsort Algorithm

A

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

k