UNIT - II

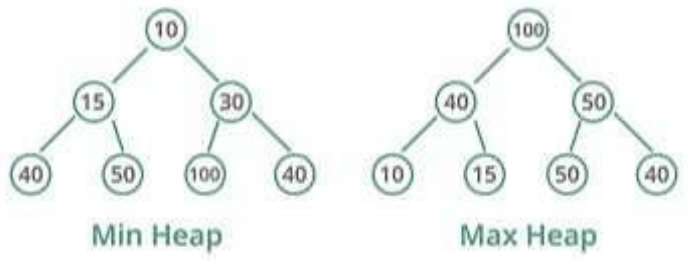
Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications
Graphs - Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications.

Heap trees: A heap tree is a specialized binary tree that satisfies the heap property. The heap property states that the value of every parent node is either less than or greater than its children nodes. Based on this heap property, there are two types of heaps:

1. **Max Heap and**
2. **Min Heap**

1. **Max Heap: It is a heap tree in which every parent node is greater than its children nodes.**
2. **Min Heap: It is a heap tree in which every parent node is less than its children nodes.**

Example:



A heap tree can be represented as an array with the following conditions:

- The root element is at index 0.
- For any node at given index i:
  - The left child is at index 2i+1.
  - The right child is at index 2i+2.
  - The parent is at index (i−1)/2 (using integer division).

Operations on Heap Trees:

1. **Insertion**: Add the new element at the end of the heap and then "heapify" up to restore the heap property.
2. **Deletion**: Generally, the root element is always removed from the heap tree. The last element is moved to the root, and then "heapify" down is performed to restore the heap property.
3. **Peek**: Access the root element without removing it.
4. **Heapify**: Adjust the tree to maintain the heap property. There are two types:

Algorithm for Insertion:

1. **Add the new element at the end of the heap array**.
2. **Heapify up** to restore the heap property.

```
void insert(int key)
{
   size = size + 1;
   heap[size] = key;
   heapifyUp();
}
```

```
void heapifyUp()
{     // last element
    int i = size;
      while(1)
      {
             int parent = i/2;
             if (parent > 0 && heap[parent] > heap[i])
             {
                     int t = heap[parent];
                     heap[parent] = heap[i];
                     heap[i]=t;
                     i = parent;
             }
             else
                   break;

      }
    }
```

Algorithm: Deletion in a Heap

1. Replace the root element with the last element.
2. Remove the last element.
3. Heapify down to restore the heap property.

```
int delete()
{
        int t = heap[1];
        heap[1] = heap[size];
        size = size - 1;
        heapifyDown();
        return t;
}

void heapifyDown()
{ // top element
        int i = 1;
        while(i<size)
        {
                int c1 = 2*i;
                int c2 = 2*i + 1;
                int t;
                if (c1 <= size)
                        t = c1;
                else
                      break;
        }
        if (c2 <= size && heap[c1] > heap[c2])
                t = c2;
        if(heap[i] >= heap[t]) break;

        int temp = heap[i];
        heap[i] = heap[t];
```

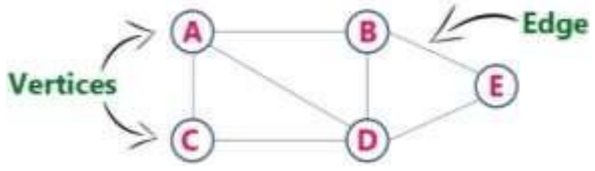
```
                heap[t] = temp;
                i = t;
        }
    }
```

Applications of Heaps

- Heaps are commonly used to implement priority queues.
- It is also used to sort the elements which is known as Heap sort.
- Used in graph algorithms like Dijkstra's shortest path and Prim's minimum spanning tree.

Introduction to Graphs:

- A graph is a non-linear data structure.
- It consists of finite set of nodes called vertices and finite set of links called edges.
- The edges are used to connect the vertices of graph.
- Hence a graph is defined as a collection of vertices and edges, i.e G= (V, E) where V is set of vertices and E is set of edges.



Basic Terminologies:
Vertex: It is an individual element of the graph. This vertex is also called as node. In the above example, A, B, C, D, E are called as vertices.
Edge: It is a connecting link between two vertices of graph. It is also called as arc. It is represented as pair of vertices. In the above example, there are seven edges i.e (A, B), (A, C), (A, D), (B, D), (B, E), (C, D), (D, E).
There are 3 types of edges.

1) **Undirected edge**: An undirected edge is bidirectional edge. If there is an undirectional edge between vertex A and vertex B then edge (A, B) is same as (B, A). In such case, pair of vertices is called as unordered pair.
2) **Directed edge:** A directed edge is unidirectional edge. If there is a directional edge between vertex a and vertex B then edge (A, B) is not same as (B, A). In such case, pair of vertices is called as ordered pair.
3) **Weighted edge:** It is an edge associated with a value (cost or weight)

Undirected Graph: It is a graph containing unordered pair of vertices or undirected edges.

Directed Graph: It is a graph with ordered pair of vertices or directed edges.

Connected Graph: It is a graph in which there exists a path from any vertex to any other vertex

Complete Graph: It is an undirected graph in which every vertex is connected to every other vertex of the graph. This is also called as fully connected.
Adjacent vertices: If there is an edge between vertex A and vertex B then they are said to be adjacent to each other.
Path: It is the sequence of adjacent vertices from source to destination vertices.

Self-loop: It is an edge that connects a vertex to itself.

Parallel and Multiple edges: If there are two undirected or directed edges connect the same vertices then that edges are called as Parallel or Multiple edges.

Simple Graph: It is a graph with no self-loop and parallel or multiple edges.

Degree: The total number of edges connected to a vertex is called as degree of that vertex.

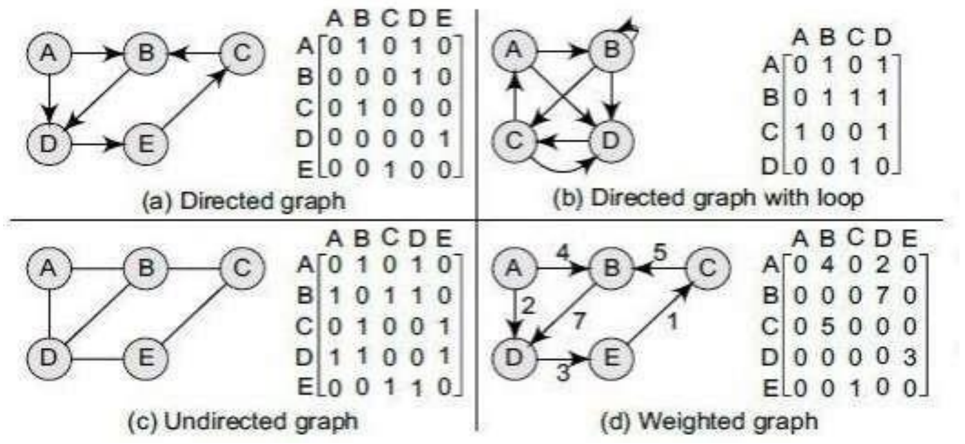
In-degree: The total number of edges leading into a vertex is called in-degree of that vertex.

Out-degree: The total number of edges leading away from a vertex is called out-degree of that vertex.

Representations of Graph: There are two common ways to represent a given graph, they are

1. **Sequential Representation** by using adjacency matrix.
2. **Linked Representation** by using adjacency list that stores the neighbours of a node using linked list.

1. **Adjacency Matrix:**
   - It is used to represent a graph using 2 dimensional matrix. For any given graph with n vertices then adjacency matrix will have the dimension nXn.
   - It is used to represent adjacent nodes of a given node.
   - In this matrix the rows and columns are labelled with vertices.
   - If the matrix is denoted by A then entry of A_{ij} = 1 or 0 for an undirected or directed graph and A_{ij} = cost for weighted graph.
   - Aij = 1 if there is an edge from V_i to V_j, otherwise A_{ij} =0

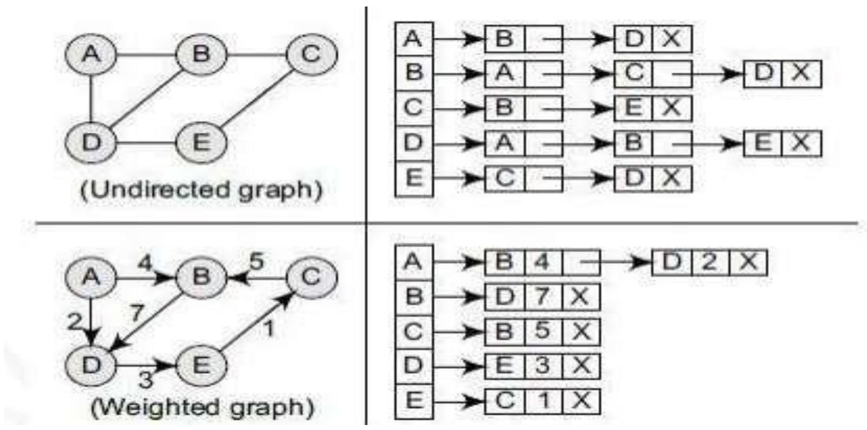


(a) Directed graph (b) Directed graph with loop (c) Undirected graph (d) Weighted graph

We can note the following things from an adjacency matrix

- For a simple graph, the adjacency matrix has 0s on the diagonal.
- The adjacency matrix for undirected graph is symmetric.
- Number of 1s in adjacency matrix is equal to the number of edges of graph.
- The adjacency matrix for a weighted graph contains the weights of edges connecting the nodes.

2. **Adjacency List:**
   - It is another representation used to represent a given graph in the computer memory.
   - It consists of list of all nodes in the graph.
   - However, every node is connected to its own list of nodes that are adjacent to it.



Graph Traversals:

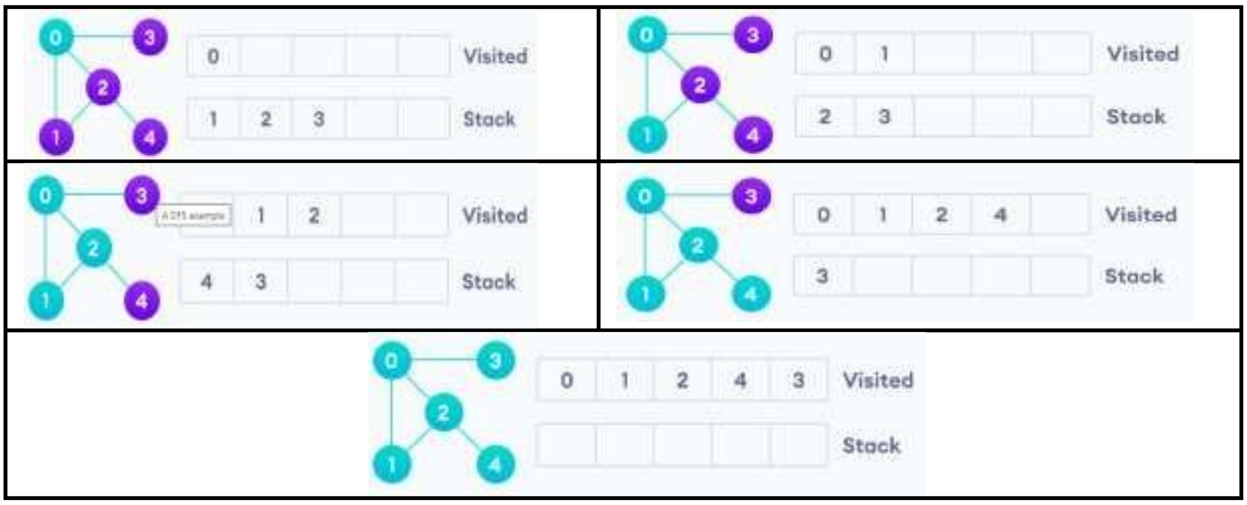
- Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
- The order in which the vertices are visited is important and may depend upon the given algorithm.
- Graph traversal is a technique also used for a searching vertex in a graph.
- In this traversal we will visit or access all the vertices of the graph.
- There are two graph traversal techniques and they are given as

   1. **DFS (Depth First Search)**
   2. **BFS (Breadth First Search)**

1. **DFS:**
   - This is used to traverse a graph. It is also called Depth First Traversal.
   - The basic idea of this process is to start with any arbitrary vertex and mark it as visited.
   - Then move to an adjacent unvisited node of it and mark it visited.
   - This process continues until there is no unvisited adjacent node.
   - Then backtrack and check for any other unvisited nodes and traverse them.
   - This process continues until all the nodes of graph can be marked as visited.

   Algorithm:

   Step 1: Define a stack with size of vertices in graph
   Step 2: Select any one starting vertex for traversal. Push it on to the stack.
   Step 3: Visit that vertex and push all non-visited adjacent vertices of that vertex into the stack.
   Step 4: When there no new vertex to be visited then use backtrack and pop.
   Step 5: Repeat above Step 3 and Step 4 until stack becomes empty.
   Step 6: Then the order of visit of the vertices becomes DFS of the graph.



2. **BFS:**

- This is another graph traversal method. It is also called Breadth First Traversal.
- The basic idea of this process is to start with any arbitrary vertex and mark it as visited.
- Then move to all adjacent unvisited nodes of it and mark them as visited.
- Check for any other unvisited adjacent nodes to visited node and traverse them.
- This process continues until all the nodes of graph can be marked as visited.

Algorithm:

Step 1: Define a Queue with size of vertices in the graph.
Step 2: Select any vertex as **starting vertex** for traversal. Visit that vertex and insert it into the Queue.
Step 3: Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
Step 4: When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
Step 5: Repeat steps 3 and 4 until queue becomes empty.
Step 6: Then the order of vertices deleted from the queue becomes BFS.

Time and Space Complexity of BFS and DFS:

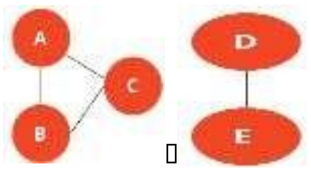
Here the time and space required by BFS and DFS on an n-vertices and e-edges graph are O(n+e) and O(n) respectively if adjacency list is used to represent it, if adjacency matrix is used to represent it then they are O(n^2) and O(n) respectively

Connected Components of graph:

- In a graph, a connected component or simply component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the rest of the graph.
- In other words, a connected component is a maximal subset of vertices such that there is a path between any pair of vertices within this subset. Maximal means that we can't add another vertex from the graph to this subset without breaking the condition that all vertices must be reachable from each other.

Example: Consider a graph with the following vertices and edges:

- **Vertices:** A, B, C, D, E
- **Edges:** (A-B), (B-C), (C-A), (D-E)
-



In this graph:

- One connected component consists of the vertices {A, B, C} because each of these vertices is connected to the others.
- Another connected component consists of the vertices {D, E} because they are connected to each other but not to any of the vertices {A, B, C}.

So, the graph has two connected components: {A, B, C} and {D, E}.

Biconnected Components:

A **biconnected component** of an undirected graph is a maximal subgraph in which any two vertices are connected by at least two independent paths. This means that the subgraph remains connected even if any single vertex is removed.

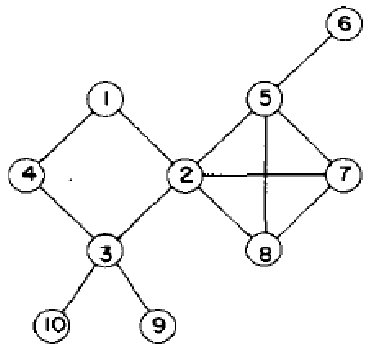
Biconnected Graph:

A graph is known as biconnected if it is connected and there are no vertices whose removal would disconnect the graph. In other words, there are no "articulation points" in that graph. i.e an articulation point is a vertex whose removal increases the number of connected components.

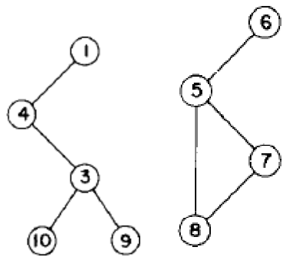
Articulation Point:

- An articulation point also known as cut vertex is a vertex in a graph whose removal would result in the graph to be disconnected in to 2 or more non empty components.
- A graph can have multiple articulation points, and each such point can separate the graph into different biconnected components.

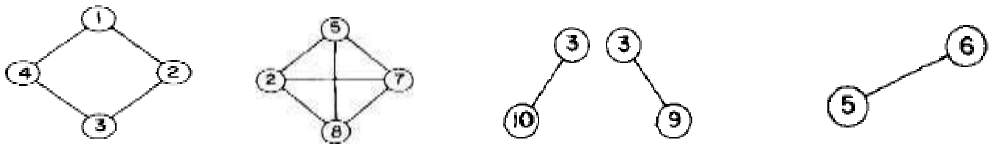
Consider the following graph



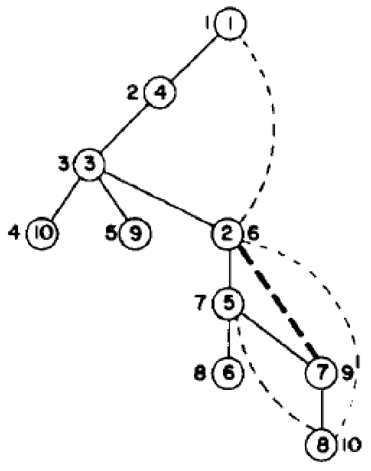
It is not biconnected because the deletion of vertex 2 and all its edges disconnects the graph into two non empty components as



Moreover it is also having two another articulation points. They are vertex 3 and vertex 5. So the biconnected component for the above graph are



So in order to identify articulation points and biconnected components DFS spanning tree can be used. The DFS spanning tree of the above graph is as follows:



The dotted line indicates back edge. There is a number outside of each vertex that corresponds to an order in which the depth first search visits the vertices. This is usually referred as Depth First Number (dfn) of vertices. So dfn(1) = 1, dfn(2)=6, dfn(3)=3, dfn(4)=2, dfn(5)=7, dfn(6)=8, dfn(7)=9, dfn(8)=10, dfn(9)=5, dfn(10)=4.

DF spanning tree properties:

1. If (u,v) is any tree edge then either u is ancestor of v or v is ancestor of u.
2. If u is any other vertex other than the root then it is not an articulation point if and only if from every child w of u it is possible to reach an ancestor of u using only a path made up of the descendents of w and a back edge. That means if it can"t be donefor a child w of u then the deletion of u is atleast two non empty components. This rule is identified an articulation point. For example each vertex u, L(u) is defined as

$$L(u) = \min\{\mathrm{DFN}(u), \min\{L(w) \mid w \text{ is a child of } u\}, \min\{\mathrm{DFN}(w) \mid (u, w) \text{ is a back edge }\}\}$$

Where L(u) is lowest depth first number that can be reached from u using a path of descendants followed by alteast one back edge. If u is an articulation point if and only if u has a child w such that L[w] ≥ dfn(u). For example for the above spanning tree the L values are computed by applying postorder traversal of the tree as

L[10] = min{ 4, _, _} = 4
L[9] = min{ 5, _, _} = 5
L[6] = min{ 8, _, _} = 8
L[8] = min{ 10, _,6}= 6
L[7] = min{ 9, 6, 6} = 6
L[5] = min{ 7, 6, _} = 6
L[2] = min{ 6, 6, 1} = 1
L[3] = min{ 3, 1, 1} = 1
L[4] = min{ 2, 1, _} = 1
L[1] = min{ 1, _, _} = 1

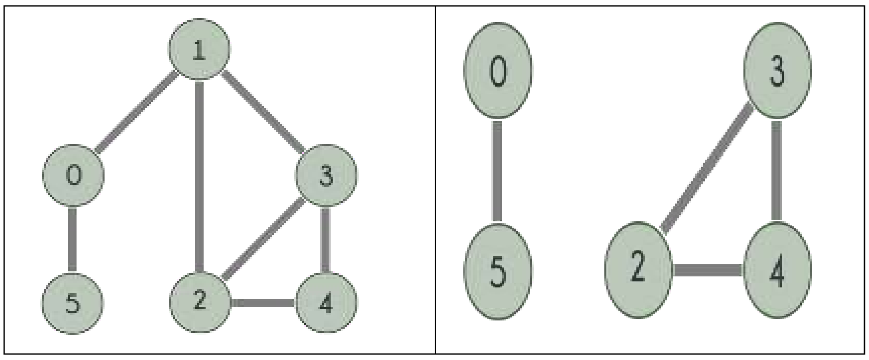
The articulation points are identified by using the condition L[w] ≥ dfn(u).

For vertex 2, L[5] = 6 and dfn(2) = 6 i.e 6 ≥ 6 is true. So vertex 2 is articulation point.For vertex 3, L[10] ≥ dfn(3) i.e 4 ≥ 3, so vertex 3 is also one articulation point. Similarly vertex 5 is also one articulation point because L[6] ≥ dfn(5) i.e 8 ≥ 7.

Algorithm Bicomp(u, v)

```
{
 Dfn[u]:= num; L[u]:= num; num:= num+1;
 for each vertex w adjacent from u do
{
    if((u ≠ w) and (dfn[w] < dnf[u])) then
        Add (u, w) to top of stack s;
    if(dfn[w] = 0) then
     {
      if(L[w] ≥ L[u]) then
       Write (“New icomponent”);
     repeat
     {
         Delete an edge from top of stack, let this edge be (x, y);
         Write(x, y);
       }until ((x,y) = (u, w) or (x, y) = (w, u);
    }
   Bicomp(w, u);
   L[u]:= min{ L[w], L[u]}
  }
  else if( w≠v) then
   L[u]:= min{L[u], dnf[w]};
  }
 }
```

Example: Consider a graph and its biconnected components



Applications:

- Graphs are used to define the **flow of computation**.
- Graphs are used to represent **networks of communication**.
- Graphs are used to represent **data organization**.
- Graph is used to find **shortest path in road** or a network.
- In **Google Maps**, various locations are represented as vertices or nodes and the roads are represented as edges and graph is used to find the shortest path between two nodes.