Heap Trees (Priority Queues) – Min and Max Heaps, Operations and Applications
Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications
Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication, Convex Hull

Heap Trees

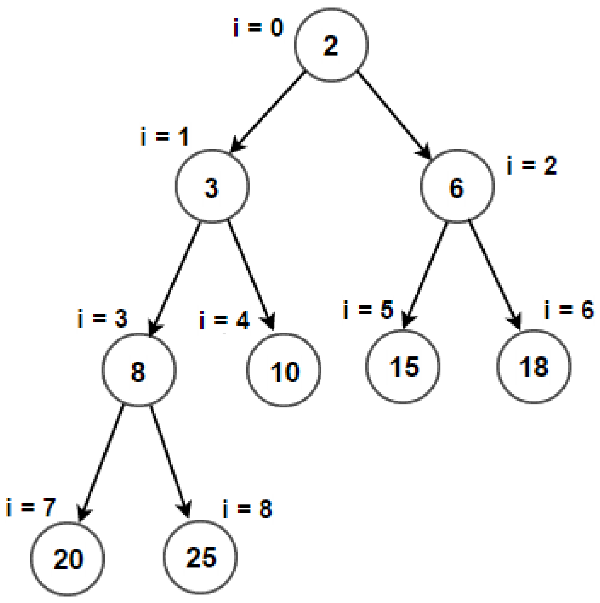
- **Complete binary tree** – A binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.

- **Heap** – A Heap is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is greater (or smaller) than or equal to its own value.

- A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. This implies that the height of a complete binary tree is $\lfloor \log N \rfloor$.

- A complete binary tress can be represented in an array and no links are necessary as shown in the figure.

i	0	1	2	3	4	5	6	7	8
	2	3	6	8	10	15	18	20	25

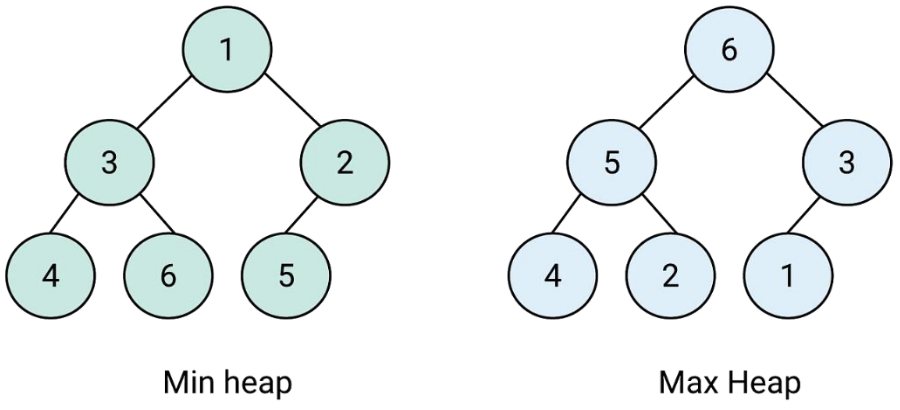


- In an array, where indexing starts from 0, the relationship between a parent node at index i and its children is given by the formulas:
  - left child at index $2i + 1$ and
  - right child at index $2i + 2$.
- Heaps are usually used to implement priority queues, where the smallest (or largest) element is always at the root of the tree.

- A **Heap** is a special Tree-based Data Structure that has the following properties.
  - It is a Complete Binary Tree.
  - It either follows max heap or min heap property.

- **Max-Heap**: The value of the root node must be the greatest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

- **Min-Heap**: The value of the root node must be the smallest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

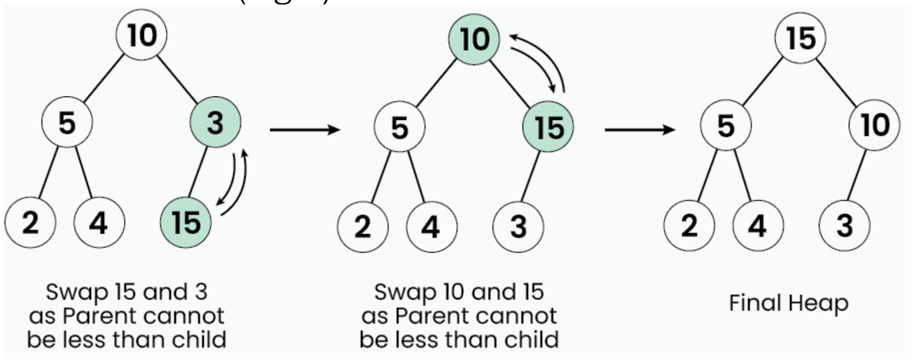


Heap Operations

Operations supported by min – heap and max – heap are same. The difference is just that min-heap contains minimum element at root of the tree and max – heap contains maximum element at the root of the tree.

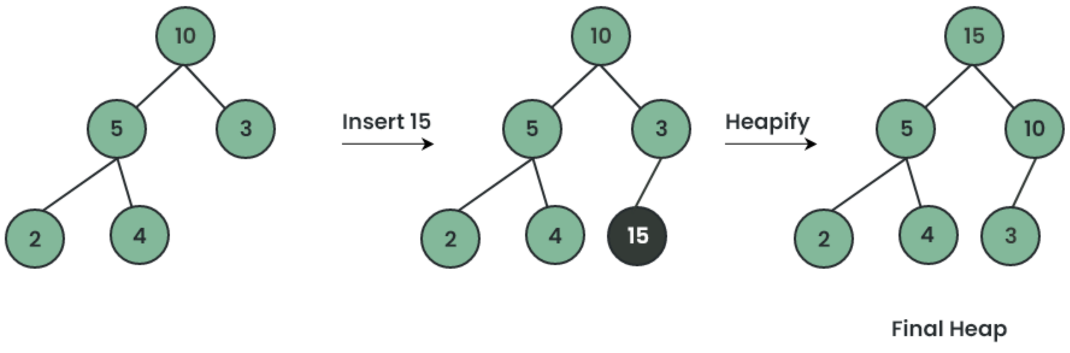
1. Heapify :

- It is the process to rearrange the elements to maintain the property of heap data structure.
- It is done when root is removed (we replace root with the last node and then call heapify to ensure that heap property is maintained) or heap is built (we call heapify from the last internal node to root) to make sure that the heap property is maintained.
- This operation also takes O(log n) time.


Swap 15 and 3 as Parent cannot be less than child
Swap 10 and 15 as Parent cannot be less than child
Final Heap

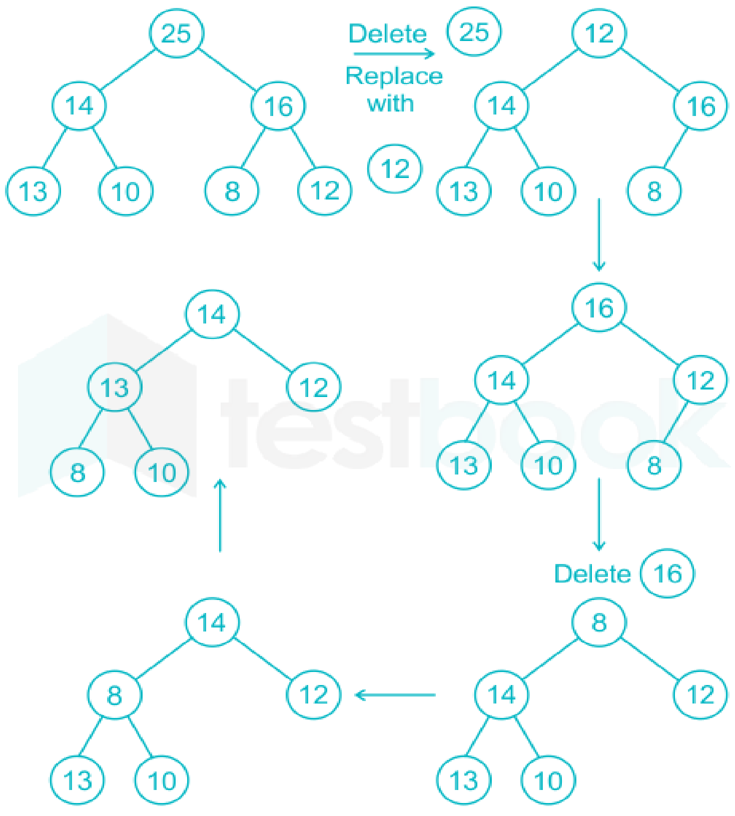
2. Insertion :

- When a new element is inserted into the heap, it can disrupt the heap's properties. To restore and maintain the heap structure, a heapify operation is performed.
- This operation ensures the heap properties are preserved and has a time complexity of O(log n).
- To insert an element X into the heap, we create a hole in the next available location, to preserve the complete binary tree property.
- If X is placed in the hole, violating the heap order, then slide the element in the parent node into the hole, thus bubbling the hole up toward the root. Continue this process until X can be placed in the hole.



3. Deletion :

- If we delete the element from the heap, it always deletes the root element of the tree and replaces it with the last element of the tree.
- Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operation so that it maintains the property of the heap.



4. getMax (For max-heap) or getMin (For min-heap):

- It finds the maximum element or minimum element for max-heap and min-heap respectively and as we know minimum and maximum elements will always be the root node itself for min-heap and max-heap respectively.
- It takes O(1) time.

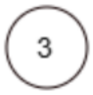
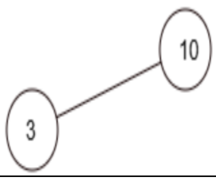
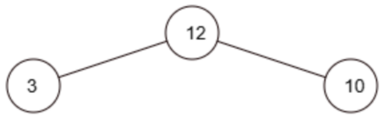
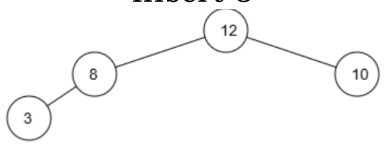
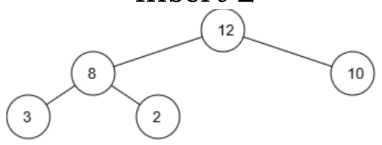
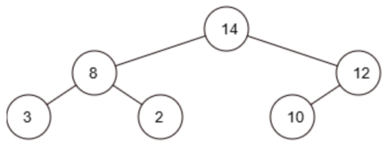
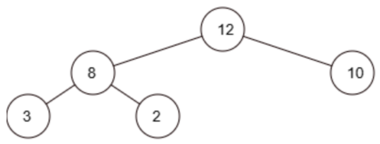
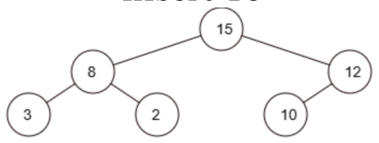
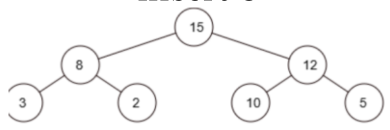
Implementation of Heap Data Structure:-

Let's understand the maxHeapify function in detail:-

maxHeapify is the function responsible for restoring the property of the Max Heap. It arranges the node i, and its subtrees accordingly so that the heap property is maintained.

1. Suppose we are given an array, $arr[]$ representing the complete binary tree. The left and the right child of i^{th} node are in indices $2 * i + 1$ and $2 * i + 2$.
2. We set the index of the current element, i, as the 'MAXIMUM'.
3. If $arr[2 * i + 1] > arr[i]$, i.e., the left child is larger than the current value, it is set as 'MAXIMUM'.
4. Similarly if $arr[2 * i + 2] > arr[i]$, i.e., the right child is larger than the current value, it is set as 'MAXIMUM'.
5. Swap the 'MAXIMUM' with the current element.
6. Repeat steps 2 to 5 till the property of the heap is restored.

Example - Construct a MaxHeap by inserting 3, 10, 12, 8, 2, 14, remove maximum, Insert 15, 16.

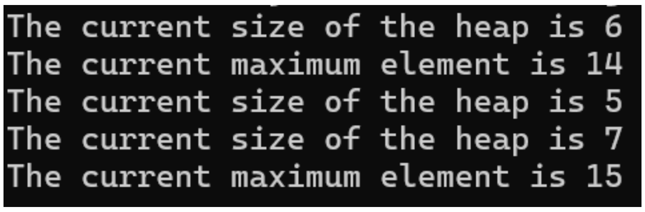
Insert 3 	Insert 10 	Insert 12 
Insert 8 	Insert 2 	Insert 14 
Remove Max 	Insert 15 	Insert 5 

Java Implementation of Max Heap

```
import java.util.Arrays;
import java.util.Scanner;

class MaxHeap {
        int[ ]   arr;
        int      maxSize, heapSize;

        MaxHeap(int maxSize){
                this.maxSize = maxSize;
                heapSize = 0;
                arr = new int[maxSize];
        }

        int parent(int i){
                return (i-1)/2;
        }

        int lChild(int i){
                return 2*i + 1;
        }

        int rChild(int i){
                return 2*i + 2;
        }

        int getMax(){
                return arr[0];
        }

        int curSize() {
                return heapSize;
        }

        void MaxHeapify(int i) {
                int l = lChild(i);
                int r = rChild(i);
                int largest = i;
                if (l < heapSize && arr[l] > arr[i])
                        largest = l;
                if (r < heapSize && arr[r] > arr[largest])
                        largest = r;
                if (largest != i) {
                        int temp = arr[i];   arr[i] = arr[largest];    arr[largest] = temp;
                        MaxHeapify(largest);
                }
        }
```

```
    void removeMax() {
        if (heapSize <= 0)
            System.out.println("Heap is empty");
        if (heapSize == 1)
            heapSize--;
        else{
            arr[0] = arr[heapSize - 1];
            heapSize--;
            MaxHeapify(0);
        }
    }

    void insertKey(int x) {
        if (heapSize == maxSize) {
            System.out.println("\nOverflow: Could not insertKey\n");
            return;
        }
        int i = heapSize;
        arr[i] = x;
        heapSize++;

        while (i != 0 && arr[parent(i)] < arr[i]) {
            int temp = arr[i];   arr[i] = arr[parent(i)];   arr[parent(i)] = temp;
            i = parent(i);
        }
    }

    public static void main(String[] args) {
        MaxHeap h = new MaxHeap(15);
        int elements[] = {3, 10, 12, 8, 2, 14};

        for (int e : elements)
            h.insertKey(e);

        System.out.println("The current size of the heap is " + h.curSize());
        System.out.println("The current maximum element is " + h.getMax());

        h.removeMax();

        System.out.println("The current size of the heap is " + h.curSize());

        // Inserting 2 new keys into the heap.
        h.insertKey(15);
        h.insertKey(5);
        System.out.println("The current size of the heap is " + h.curSize());
        System.out.println("The current maximum element is " + h.getMax());
    }
}
```

Output:



```
The current size of the heap is 6
The current maximum element is 14
The current size of the heap is 5
The current size of the heap is 7
The current maximum element is 15
```

Exercise – Design a Java Program to Implement MinHeap

Applications of Heaps:

- Priority Queues: Used to extract elements with the highest priority first, aiding in scheduling tasks, handling interruptions, and processing events.
- Sorting Algorithms: Heapsort sorts data efficiently with a time complexity of
- $O(n\, logn)$.
- Graph Algorithms: Utilized in Prim's, Dijkstra's, and A* algorithms for optimizing paths and spanning trees.
- Lossless Compression: Min-heaps help build Huffman trees in Huffman coding for data compression.
- Medical Applications: Manage patient data by priority (e.g., vital signs or treatments).
- Load Balancing: Distributes tasks to servers by processing the lowest-load element first.
- Resource Allocation: Allocates system resources like memory or CPU time based on priority.
- Job Scheduling: Schedules tasks by priority or deadline, ensuring efficient access to high-priority tasks.

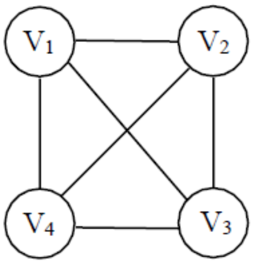
Time Complexities:

- Insert Operation: $O(\log N)$
- Delete Operation: $O(\log N)$
- Get Minimum (Min-Heap): $O(1)$
- Get Maximum (Max-Heap): $O(1)$

Graphs

Def: A graph $G = (V, E)$ consists of a set of vertices, V and set of edges E. V is a finite, non-empty set of vertices.

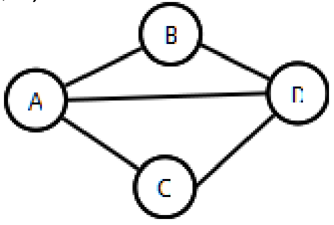
- Vertices are referred to as nodes and the arc between the nodes are referred to as Edges. Each edge is a pair (v, w) where $v, w \in V$.



- Here V_1, V_2, V_3, V_4 are the vertices and $(V_1, V_2), (V_2, V_3), (V_3, V_4), (V_4, V_1), (V_2, V_4), (V_1, V_3)$ are edges.
- Graphs can be classified by whether or their edges are having direction or not and also based on edge weights.

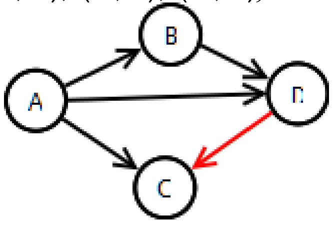
Undirected Graphs:

- each edge can be traversed in either direction.
- In an undirected graph, an edge is an unordered pair
- If the edge (A,B) ∈ E then (B,A) ∈ E



Directed Graphs:

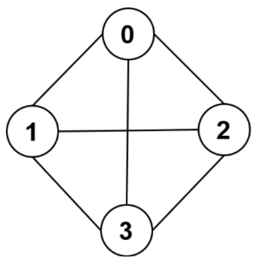
- each edge can be traversed only in a specified direction.
- Edge can only be traversed in direction of arrow.
- Edge is represented by a directed pair (u, v) - (u, v) and (v, u) are two different edges.
- Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}



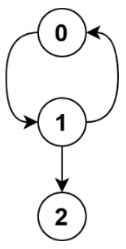
- In a digraph, an edge is an ordered pair Thus: (u,v) and (v,u) are not the same edge
- In the example, (D,C) ∈ E, (C,D) ∉ E

Degree:

- The degree of a vertex is the number of edges incident to that vertex.
- If G is a directed graph,
- In-degree – of a vertex v to be the number of edges for which v is head.
- Out-degree – is the number of edges for which v is the tail



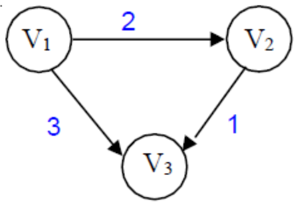
The degree of vertex **0** is 3.

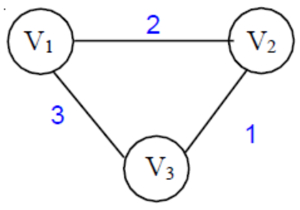


For vertex **1**,
Indegree – 1
Outdegree – 2
Degree - 3

Weighted Graph:

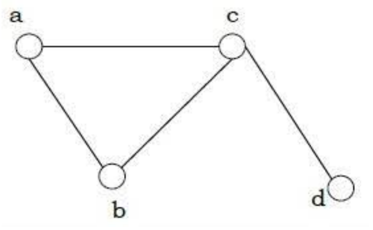
- A graph is said to be weighted graph if every edge in the graph is assigned a weight or value.
- It can be directed or undirected graph.





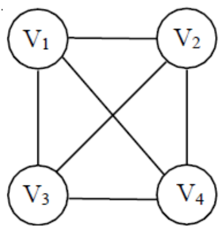
Connected Graph :

- An undirected graph is said to be connected iff, for every pair of distinct vertices u and v in $V(G)$ there is a path from u to v.



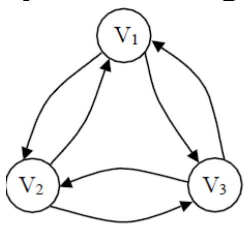
Complete or Strongly Connected Graph:

- A complete graph is a graph in which there is an edge between every pair of vertices.
- A complete graph with n vertices will have $n\,(n - 1)/2$ edges.



Strongly Connected diGraph

- If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph.
- Otherwise, it is said to be weakly connected graph.

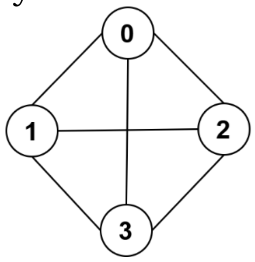


Path:

- A path from vertex u to v in graph G is a sequence of vertices $u, i_1, i_2, \dots i_k, v$ such that $(u, i_1), (i_1, i_2) \dots (i_k, v)$ are edges in $E(G)$.
- The length of the path is the number of edges on it.
- A simple path is a path in which all vertices except possibly the first and last are distinct.

Cycle: a path that starts and ends on the same vertex

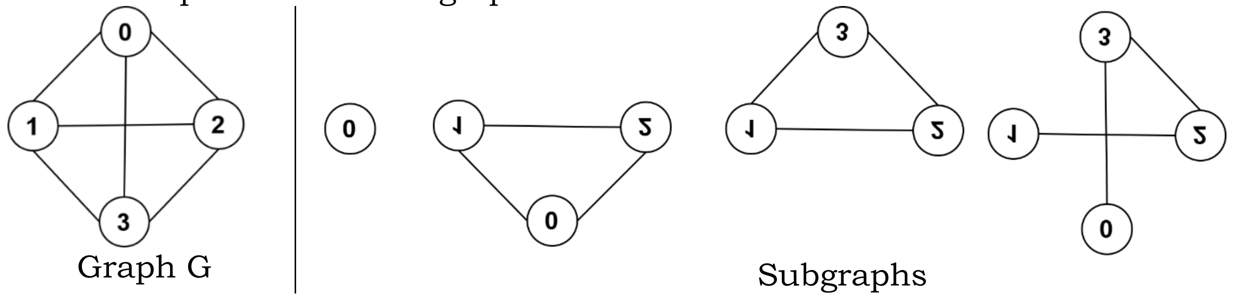
- A Cyclic graph contains cycles
- An acyclic graph contains no cycles



- In the above graph, the path, $0, 1, 2, 0$ is a cycle.

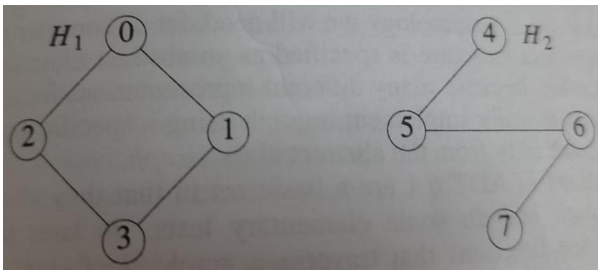
Subgraph:

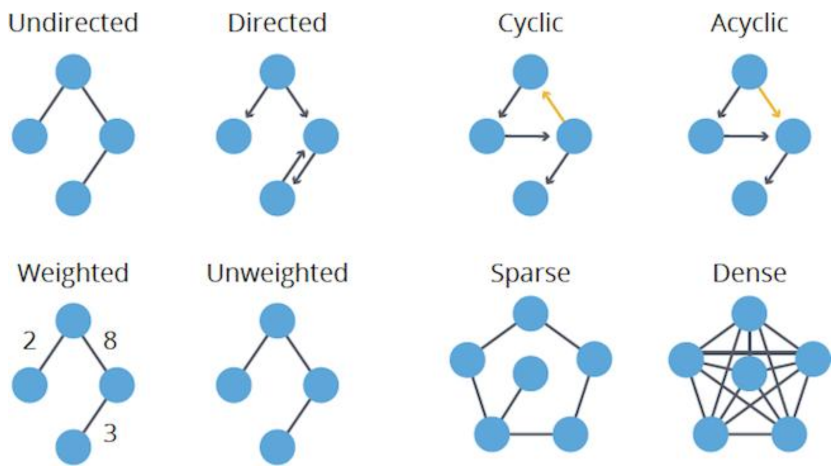
- A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$
- Example: the four subgraphs of G are



Connected Component

- A connected component H, of an undirected graph is a maximal connected subgraph.
- Maximal means, that graph G contains no other subgraph that is both connected and properly contains H.
- In the following graph G, there are two connected components H1, and H2.





Representation of Graphs

- Two common data structures are used for representing graphs:
    - Adjacency matrix
    - Adjacency lists

Adjacency Matrix Representation

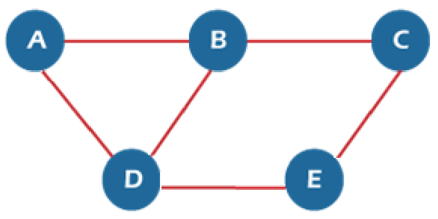
- This representation is the simple way to represent graphs.
- The adjacency matrix is one of the most common ways to represent a graph computationally.

- The adjacency Matrix A for a graph $G = (V, E)$ with n vertices is an $n \, X \, n$ matrix, such that
    - $A_{ij} = 1$, if there is an edge from $V_i \; to \; V_j$
    - $A_{ij} = 0$, if there is no edge.

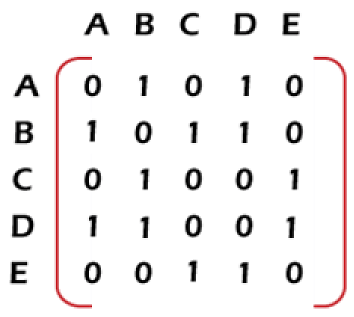
- From the adjacency matrix, it is easy to determine
    - If there is an edge connecting two vertices i and j.

<u>Adjacency Matrix for Undirected graph:</u>

- If there is an edge, 1 is used otherwise a 0 is used to representation.
- There is an edge between A and B in the figure, therefor $V_{AB} = V_{BA} = 1$
- For undirected graphs, the adjacency matrix is **symmetric**. This means $A_{ij} = A_{ji}$ for all i and j.



Undirected Graph



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

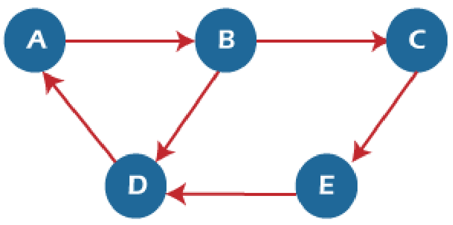
Adjacency Matrix

For an undirected graph the degree of any vertex i is its row sum:

$$\sum_{j=0}^{n-1} a[i][j]$$

Adjacency matrix for directed graph:

- If there is a directed edge from vertex $i \text{ } to \text{ } j$ then $V_{ij} = 1$, otherwise 0 is used for representation.
- There is an edge from A to B, therefore $V_{AB} = 1$, and as there is no edge from B to A, $V_{BA} = 0$.
- Adjacency matrix is not symmetric.



Directed Graph

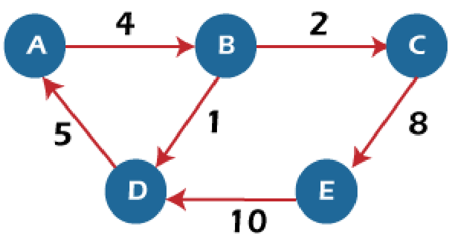
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Adjacency Matrix

- For a directed graph,
    - The row sum is the out-degree.
    - The column sum is the in-degree.

Adjacency matrix for weighted graph

- If an edge is associated with a weight, then the matrix contains its weight as the value in the matrix.



weighted Directed Graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

Adjacency Matrix

- By examining the entries of the adjacency matrix, one can determine whether the graph is connected or not.
- Degree of Vertices: The degree of a vertex in a graph is the number of edges incident to it. In an undirected graph, the degree of a vertex can be calculated by summing the entries in the corresponding row (or column) of the adjacency matrix. In a directed graph, the in-degree and out-degree of a vertex can be similarly determined.

Advantages of Adjacency Matrix:

- Simple and Easy to implement.
- Space efficient when the graph is dense as it requires $V * V$ space to represent the entire graph.
- Faster access to Edges: Adjacency Matrix allows constant look up to check whether there exists an edge between a pair of vertices.

Disadvantages of Adjacency Matrix:

- Space inefficient for Sparse Graphs: Takes up $O(V * V)$ space even if the graph is sparse.
- Costly Insertions and Deletions: Adding or deleting a vertex can be costly as it requires resizing the matrix.
- Slow Traversals: Graph traversals like DFS, BFS takes $O(V * V)$ time to visit all the vertices whereas Adjacency List takes only $O(V + E)$.

Adjacency List Representation

- In this representation, the graph is stored in a linked structure.
- We store all vertices in a list and then for each vertex, we have a linked list of its adjacency vertices.

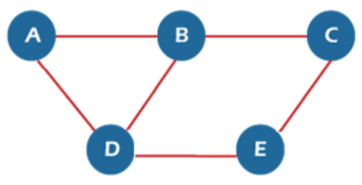
- In Adjacency list representation,
  - The n vertices are represented as n chains, one for each vertex in G.
  - The nodes in Chain i represent the vertices that are adjacent from vertex i.
  - The data field of a chain node stores the index of an adjacent vertex.
  - The vertices in each chain are not required to be ordered.

- In simple words, an adjacency list representation uses a list of lists. All adjacent nodes of a node are stored as a list. The size of the list is determined by the number of vertices in the graph.
- All adjacent of a vertex are easily available. To find all adjacent, we need only $O(n)$ time where is the number of adjacent vertices.

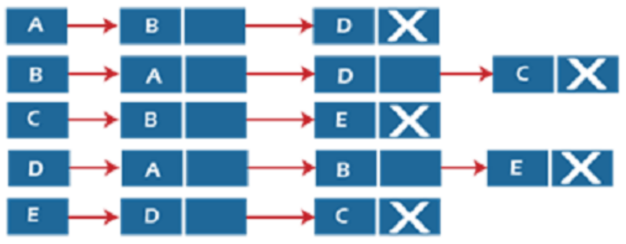
- For an undirected graph,
  - The number of list nodes is $2e$.
  - The degree of any vertex may be determined by counting the number of nodes on its adjacency list.
- For a directed graph,
  - The number of list nodes is e.
  - The in-degree of any vertex may be determined by counting the number of nodes on its adjacency list.
  - The out-degree calculation is complex.

- Graph algorithms: Many graph algorithms like Dijkstra's algorithm, Breadth First Search, and Depth First Search perform faster for adjacency lists to represent graphs.
- Adjacency List representation is the most commonly used representation of graph as it allows easy traversal of all edges.

Adjacency List representation of undirected graph

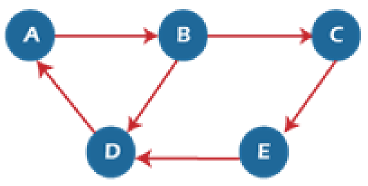


Undirected Graph

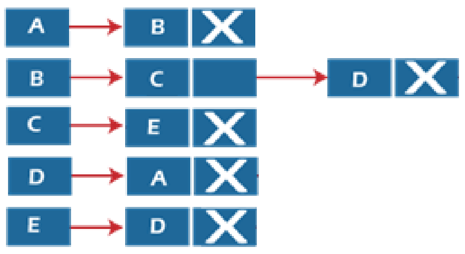


Adjacency List

Adjacency List representation of a directed graph

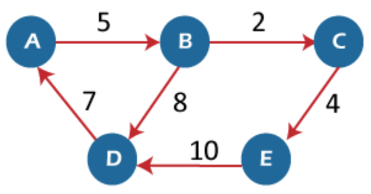


Directed Graph

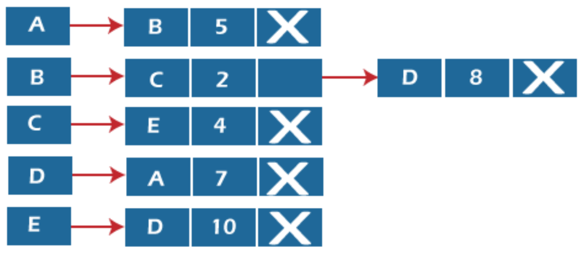


Adjacency List

Adjacency representation of a weighted directed graph



Weighted Directed Graph



Adjacency List

<u>Advantages of using an Adjacency list:</u>

- An adjacency list is simple and easy to understand.
- Requires less space compared to adjacency matrix for sparse graphs.
- Easy to traverse through all edges of a graph.
- Adding a vertex is easier compared to adjacency matrix representation.
- Most of the graph algorithms are implemented faster with this representation.

<u>Disadvantages of using an Adjacency list:</u>

- Checking if there is an edge between two vertices is costly as we have traversed the adjacency list.
- Not suitable for dense graphs.

Graph Traversals

- A graph traversal is a systematic way of visiting the nodes in a specific order. Given a graph $G = (V, E)$ and a vertex v in $V(G)$, traversal operations help to visit all vertices in G that are reachable from v.
- There are two ways of graph traversal namely,
    - Depth first search
    - Breadth first search

- These methods work on both directed and undirected graphs.

Breadth First Search or BFS for a Graph

- Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed.
- BFS is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level.
- BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

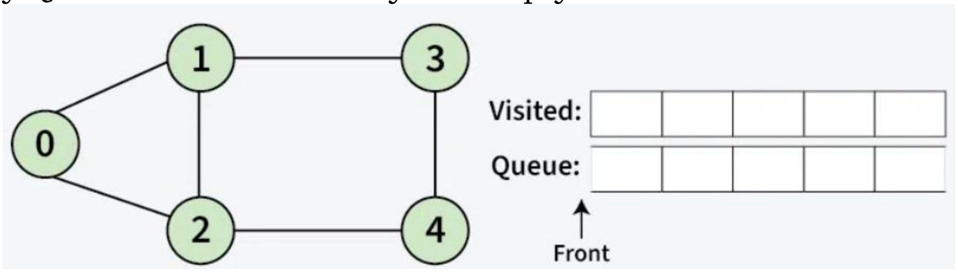
Algorithm

1. **Initialization**: Enqueue the given source vertex into a queue and mark it as visited.
2. **Exploration**: While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbour of the dequeued node:
        - Enqueue the neighbour into the queue.
        - Mark the neighbour as visited.
3. **Termination:** Repeat step 2 until the queue is empty.

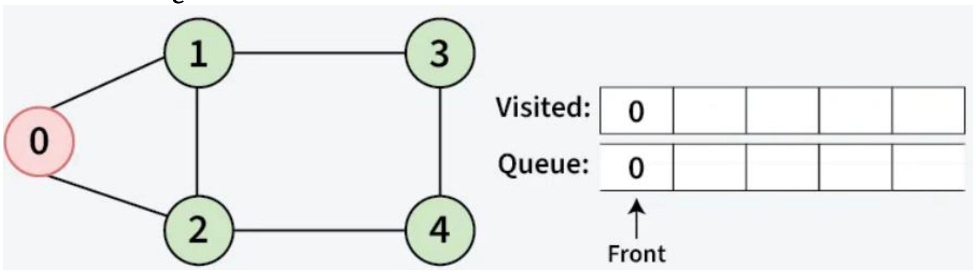
- This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node

Example:

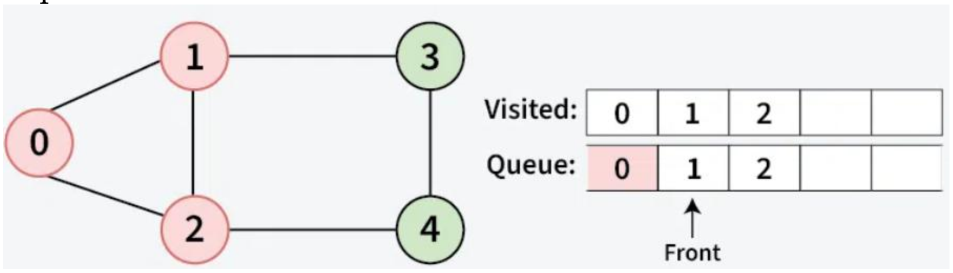
Step 1: Initially Queue and visited array are empty



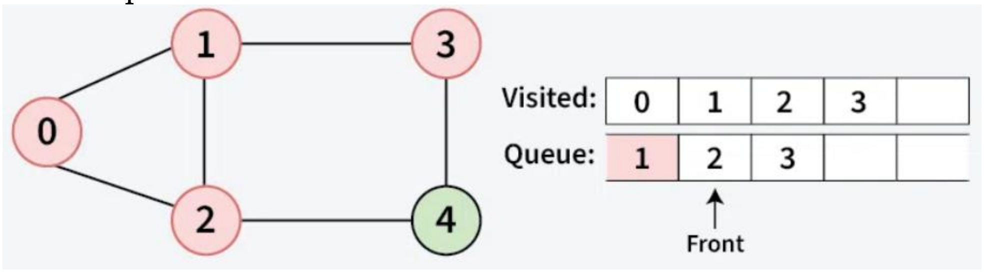
Step 2: Push 0 into the Queue and mark it as visited



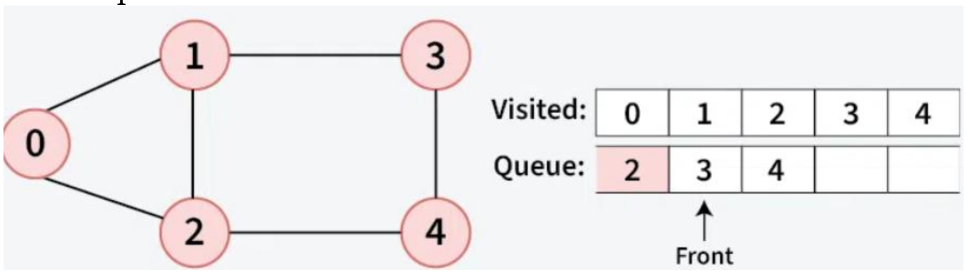
Step 3: Remove 0 from the front of the queue and visit the unvisited neighbours and push them into queue



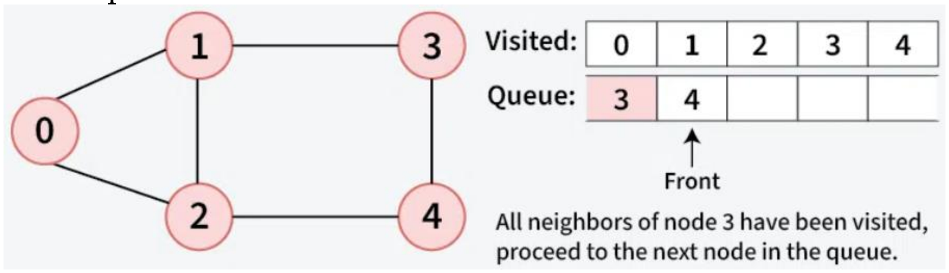
Step 4: Remove node 1 from the front of the queue, and visit the unvisited neighbours and push them into queue



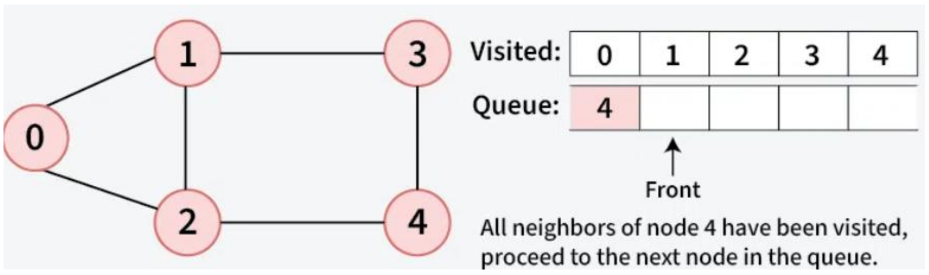
Step 5: Remove node 2 from the front of the queue and visit the unvisited neighbours and push them into queue.



Step 6: Remove node 3 from the front of the queue and visit the unvisited neighbours and push them into queue.


Visited: 0 1 2 3 4
Queue: 3 4
Front
All neighbors of node 3 have been visited, proceed to the next node in the queue.

Step 7: Remove node 3 from the front of the queue and visit the unvisited neighbors and push them into queue.


Visited: 0 1 2 3 4
Queue: 4
Front
All neighbors of node 4 have been visited, proceed to the next node in the queue.

Pseudocode for BFS

```
Algorithm BFS (int v)
//BFS starts at v, n is the no.of vertices
{
    visited = new bool[n];
    fill(visited, false);
    Queue q;
    q.enqueue(v);
    while ( !q.isEmpty( )){
        v = q.dequeue();
        for ( all vertices w adjacent to v)
            if (!visited[w]) {
                q.enqueue(w);
                visited(w) =true;
            }
    }
}
```

Analysis of BFS:

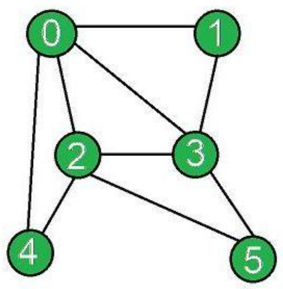
- Each visited vertex enters the queue exactly once.
- So, the while loop is iterated at most n times.
- If an adjacency matrix is used,
    - The loop takes $O(n)$ time for each vertex visited.
    - The total time is, therefore, $O(n^2)$.
- If adjacency lists are used,
    - The loop has a total cost of $d_0 + \cdots + d_{n-1} = O(e)$, where d_i is the degree of vertex i .

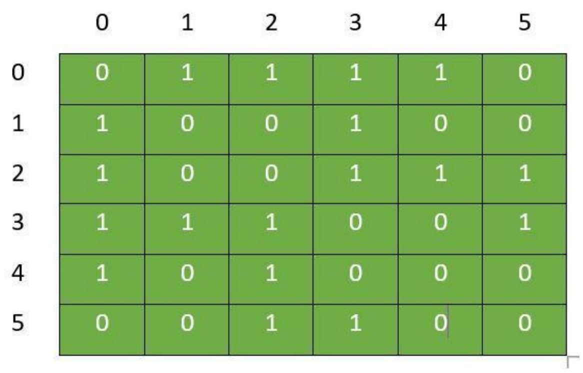
Java Implementation of BFS using Adjacency Matrix

```
import java.util.*;

class BFSAdjMatrix {

  private int v;
  private int[ ][ ]  adj;

  public BFSAdjMatrix(int v) {
    this.v = v;
    adj = new int[v][v];
  }
```

```
    public void BFS(int start) {
        boolean[] visited = new boolean[v];
        Arrays.fill(visited, false);

        Queue<Integer> queue = new LinkedList<>();
        queue.add(start);

        visited[start] = true;

        while (!queue.isEmpty()) {
            int vis = queue.poll();
            System.out.print(vis + " ");
            for (int i = 0; i < v; i++)  // Explore all adjacent vertices
                if (adj[vis][i] == 1 && !visited[i]) {
                    queue.add(i);
                    visited[i] = true;
                }
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int v = scanner.nextInt();

        BFSAdjMatrix graph = new BFSAdjMatrix(v);

        System.out.println("Enter the adjacency matrix (" + v + "x" + v + "): ");
        for (int i = 0; i < v; i++)
            for (int j = 0; j < v; j++)
                graph.adj[i][j] = scanner.nextInt();

        System.out.print("Enter the starting vertex for BFS: ");
        int start = scanner.nextInt();
        System.out.println("BFS traversal starting from vertex " + start + ":");
        graph.BFS(start);

        scanner.close();
    }
}
```

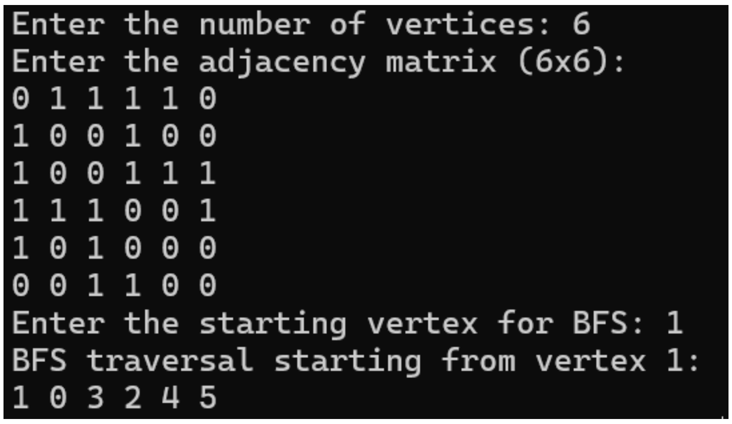
Input Graph





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

Execution

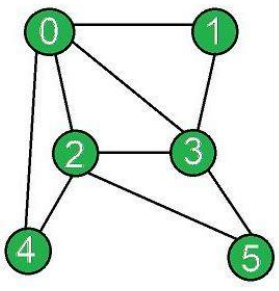


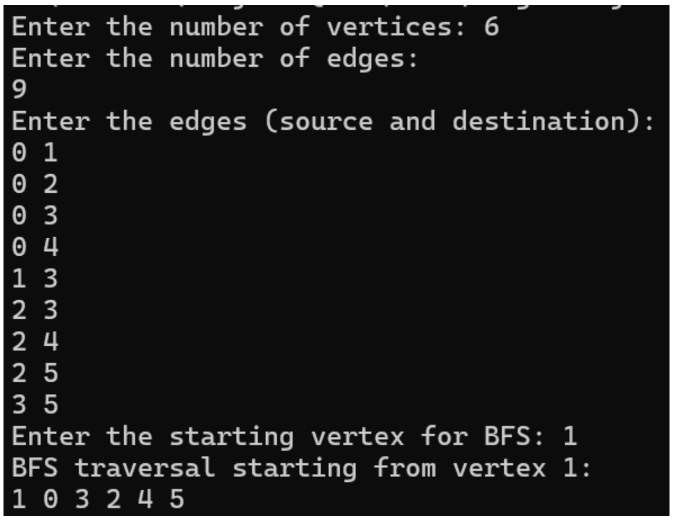
```
Enter the number of vertices: 6
Enter the adjacency matrix (6x6):
0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 1 1 1
1 1 1 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0
Enter the starting vertex for BFS: 1
BFS traversal starting from vertex 1:
1 0 3 2 4 5
```

Java Implementation of BFS using Adjacency list

```
import java.util.*;

class BFSAdjList {
    private int v;
    private LinkedList<Integer>[] adj;
    public BFSAdjList(int v) {
        this.v = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++)
            adj[i] = new LinkedList<>();
    }

    public void addEdge(int src, int dest) {
        adj[src].add(dest);
        adj[dest].add(src); // for undirected graph
    }

    public void BFS(int start) {
        boolean[] visited = new boolean[v];
        Arrays.fill(visited, false);

        Queue<Integer> queue = new LinkedList<>();
        queue.add(start);
        visited[start] = true;

        while (!queue.isEmpty()) {
            int vis = queue.poll();
            System.out.print(vis + " ");

            for (int neighbor : adj[vis])    // Explore all adjacent vertices
                if (!visited[neighbor]) {
                    queue.add(neighbor);
                    visited[neighbor] = true;
                }
        }
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of vertices: ");
        int v = scanner.nextInt();

        BFSAdjList graph = new BFSAdjList(v);

        System.out.println("Enter the number of edges: ");
        int e = scanner.nextInt();

        System.out.println("Enter the edges (source and destination): ");
        for (int i = 0; i < e; i++) {
            int src = scanner.nextInt();
            int dest = scanner.nextInt();
            graph.addEdge(src, dest);
        }

        System.out.print("Enter the starting vertex for BFS: ");
        int start = scanner.nextInt();
        System.out.println("BFS traversal starting from vertex " + start + ":");
        graph.BFS(start);

        scanner.close();
    }
}
```

Output:





```
Enter the number of vertices: 6
Enter the number of edges:
9
Enter the edges (source and destination):
0 1
0 2
0 3
0 4
1 3
2 3
2 4
2 5
3 5
Enter the starting vertex for BFS: 1
BFS traversal starting from vertex 1:
1 0 3 2 4 5
```

Depth First Search or DFS of a graph

- The algorithm starts from a given source and explores all reachable vertices from the given source.
- We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated.
- When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w.

- The search terminates when no unvisited vertex can be reached from any of the visited vertices.
- This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

Algorithm: Depth First Search (DFS)

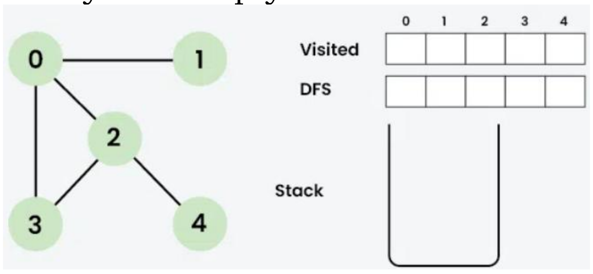
1. **Initialization**:
    - Create a stack (or use recursion for implicit stack).
    - Push the source vertex onto the stack and mark it as visited.
2. **Exploration**:
    - While the stack is not empty:
        - Pop a node from the stack and visit it (e.g., print its value).
        - For each unvisited neighbor of the popped node:
            - Push the neighbor onto the stack.
            - Mark the neighbor as visited.
3. **Termination**:
    - Repeat step 2 until the stack is empty.

A Recursive Pseudocode for DFS is

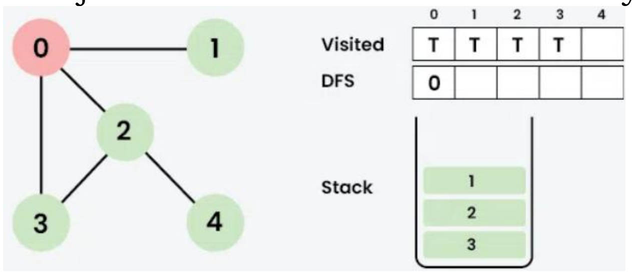
```
Algorithm DFS_start()
{
        visited = new bool[n];
        fill = visited(false);
        DFS(0);
}
Algorithm DFS(int v)
{
        visited[v] = true;
        for ( each vertex w adjacent to v)
                if (!visited[w])
                        DFS(w)'
}
```

Example:

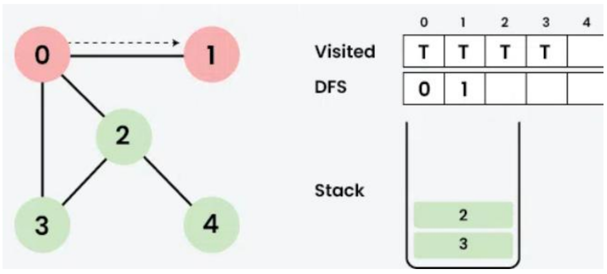
Initially, Stack and visited arrays are empty



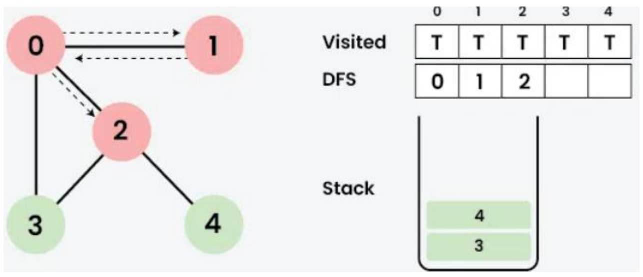
Step 1 : Visit 0 and put its adjacent nodes which are not visited yet into the stack



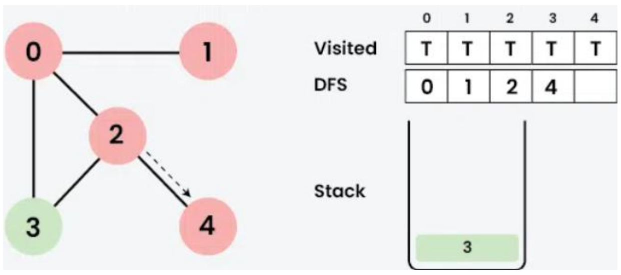
Step 2: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



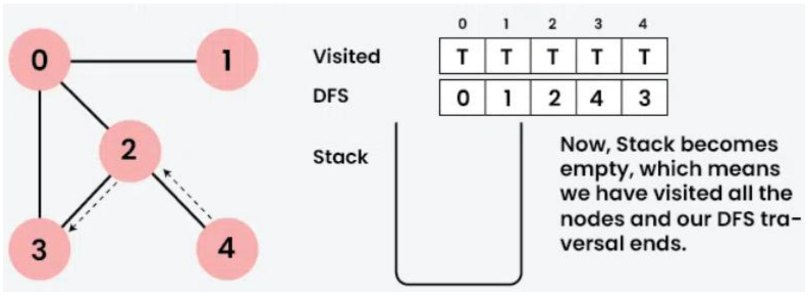
Step 3: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Step 4: Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Step 5: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.


Visited
T T T T T
DFS
0 1 2 4 3
Stack
Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

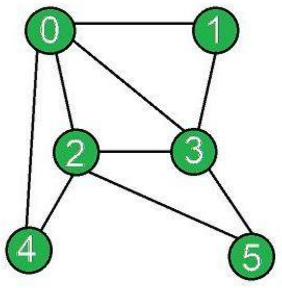
Analysis of DFS:

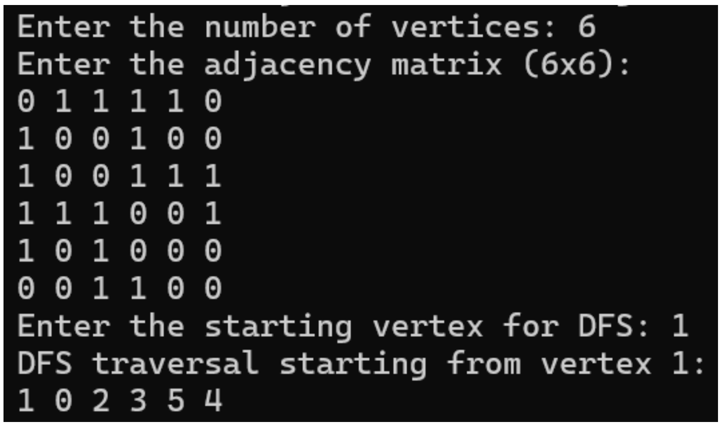
- If Graph G, is represented by using adjacency lists,
    - The vertices w adjacent to v can be determined by following a chain of links.
    - Since DFS, examines each node in the adjacency list at most once, and there are $2e$ list nodes, the time to complete the search is $O(e)$.
- If Graph G, is represented by using adjacency matrix,
    - Then the time to determine all vertices adjacent to v is $O(n)$.
    - Since, at most n vertices are visited, the total time is $O(n^2)$.

Java Implementation of DFS (recursive) using Adjacency Matrices

```
import java.util.*;

class DFSAdjMatrix {

    private int v; // Number of vertices
    private int[ ][ ] adj; // Adjacency matrix
    boolean[ ] visited;

    public DFSAdjMatrix(int v) {
        this.v = v;
        adj = new int[v][v];
        visited = new boolean[v];
        Arrays.fill(visited, false);
    }

    public void DFS(int node) {
        visited[node] = true;
        System.out.print(node + " ");

        for (int i = 0; i < v; i++) {
            if (adj[node][i] == 1 && !visited[i])
                DFS(i); // Recursive call for the unvisited neighbor
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        int v = scanner.nextInt();
        DFSAdjMatrix graph = new DFSAdjMatrix(v);
        System.out.println("Enter the adjacency matrix (" + v + "x" + v + "): ");
        for (int i = 0; i < v; i++)
            for (int j = 0; j < v; j++)
                graph.adj[i][j] = scanner.nextInt();
        System.out.print("Enter the starting vertex for DFS: ");
        int start = scanner.nextInt();
        System.out.println("DFS traversal starting from vertex " + start + ":");
        graph.DFS(start);
        scanner.close();
    }
}
```

Output:





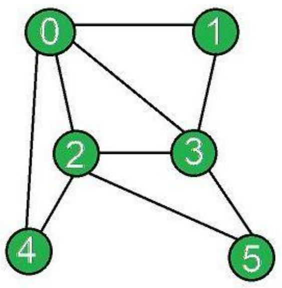
```
Enter the number of vertices: 6
Enter the adjacency matrix (6x6):
0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 1 1 1
1 1 1 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0
Enter the starting vertex for DFS: 1
DFS traversal starting from vertex 1:
1 0 2 3 5 4
```

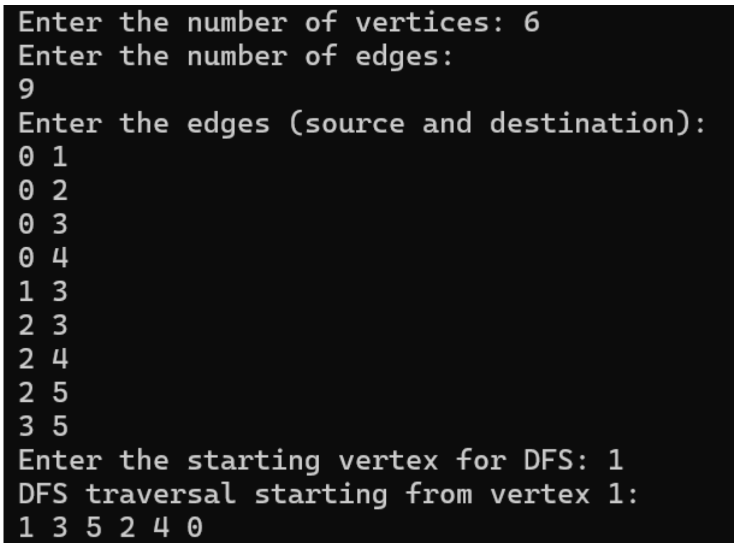
Java implementation of DFS using Adjacency list

```
import java.util.*;
class DFSAdjList {
    private int v;
    private LinkedList<Integer>[] adj;
    @SuppressWarnings("unchecked")
    public DFSAdjList(int v) {
        this.v = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++)
            adj[i] = new LinkedList<>();
    }
    public void addEdge(int src, int dest) {
        adj[src].add(dest);
        adj[dest].add(src); // for undirected graph
    }
    public void DFS(int start) {
        boolean[] visited = new boolean[v];
        Arrays.fill(visited, false);
        java.util.Stack<Integer> stack = new java.util.Stack<>();
        stack.push(start);
        visited[start] = true;
        while (!stack.isEmpty()) {
            int vis = stack.pop();
            System.out.print(vis + " ");
            // Explore all adjacent vertices
            for (int neighbor : adj[vis])
```

```
                if (!visited[neighbor]) {
                    stack.push(neighbor);
                    visited[neighbor] = true;
                }
            }
        }

    public static void main(String[] args) {
        //refer BFS AdjList implementation and write.
    }
}
```

Output:





```
Enter the number of vertices: 6
Enter the number of edges:
9
Enter the edges (source and destination):
0 1
0 2
0 3
0 4
1 3
2 3
2 4
2 5
3 5
Enter the starting vertex for DFS: 1
DFS traversal starting from vertex 1:
1 3 5 2 4 0
```

Connected Components

Graph

- A graph G=(V,E)G = (V, E)G=(V,E) consists of:
    - **V**: Set of vertices (nodes).
    - **E**: Set of edges connecting pairs of vertices.

Connected Graph

- A graph is connected if there is a path between every pair of vertices.

Component

- A **component** of a graph is a maximal connected subgraph.

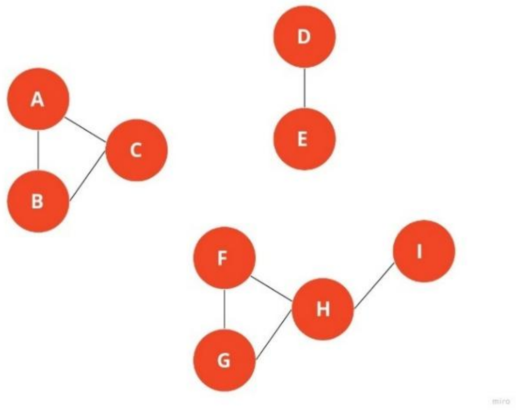
Connected Components

- In an **undirected graph**, a connected component is a set of vertices such that there exists a path between any two vertices in the set, and the set is not a subset of a larger connected set.
- A connected component in a graph is a subgraph that is connected and is not connected to any other subgraph. In other words, a connected component is a maximal subgraph that is connected. A graph can have one or more connected components.
- A graph can have multiple connected components.

Formal Definition:

A connected component is a subgraph C of a graph G = (V, E) if:

- C is a subgraph of G
- C is connected
- There is no other subgraph C' of G such that C is a subgraph of C' and C' is connected



- The number of connected components in the above graph are three
- There are several algorithms to identify Connected Components in a graph. The most popular ones are:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
  - Union-Find Algorithm (also known as Disjoint Set Union)

Finding connected components of a given undirected graph

Pseudocode

```
Algorithm CComponents()
{
    visited = new bool[n];
    fill(visited, false);
    for ( i = 0; i < n; i++ )
        if ( !visited[i] ) {
        DFS(i); //find a component
        OutputNewComponent();
        }
}
```

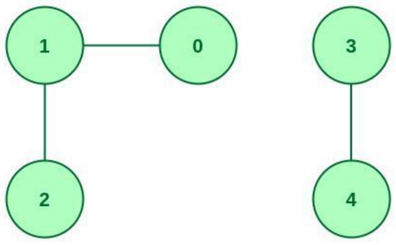
Analysis of Components

- If G is represented by its adjacency lists,
  - Then the total time taken by DFS is $O(e)$.
  - The output can be completed in time $O(e)$ if DFS keeps a list of all newly visited vertices.
  - Since the for loops take $O(n)$ time, the total time to generate all connected components is $O(n + e)$.
- If adjacency matrix is used,
  - The time required is $O(n^2)$.

Java Method

```
void DFS(int v, boolean[] visited)
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v + " ");

        for (int x : adjListArray.get(v))
            if (!visited[x])
                DFS(x, visited);
    }

    void connectedComponents()
    {
        // Mark all the vertices as not visited
        boolean[] visited = new boolean[V];
        for (int v = 0; v < V; ++v)
            if (!visited[v]) {
                // print all reachable vertices from v
                DFS(v, visited);
                System.out.println();
            }
    }
```

Input :



Output:

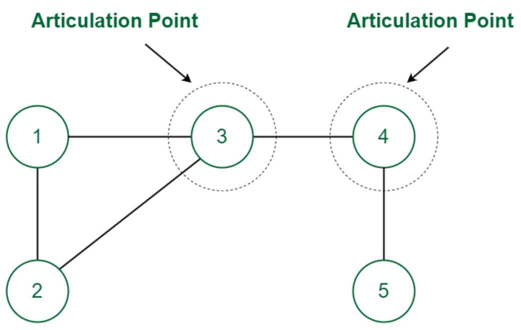
```
        0 1 2
        3 4
```

Applications of Connected Component:

- Graph Theory: It is used to find subgraphs or clusters of nodes that are connected to each other.
- Computer Networks: It is used to discover clusters of nodes or devices that are linked and have similar qualities, such as bandwidth.
- Image Processing: Connected components also have usage in image processing.

Biconnected Component

Let G is an undirected, connected graph.
Articulation Point – A vertex v of G is an articulation point, iff the deletion of v, together with the deletion of all edges incident to v, leaves behind a graph that has atleast two connected components.

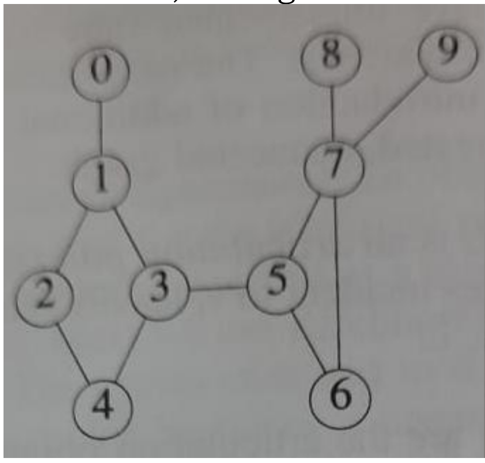


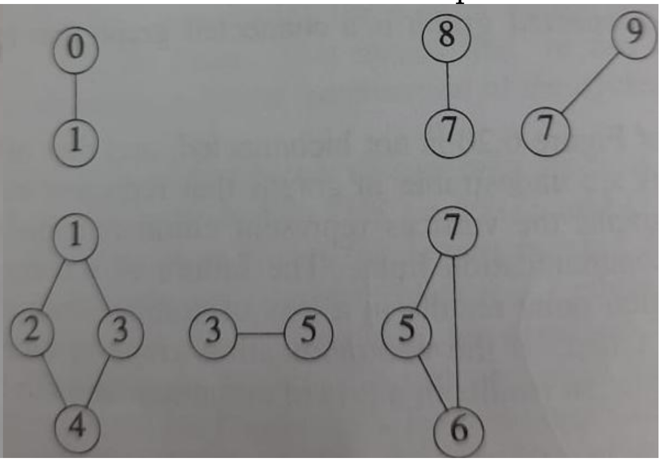
- In the above graph vertex 3 and 4 are Articulation Points since the removal of vertex 3 (or 4) along with its associated edges makes the graph disconnected.

Articulation points are undesirable in graphs that represent communication networks.

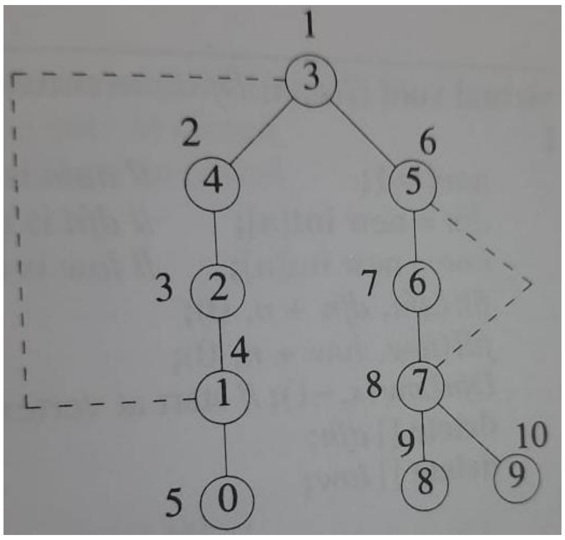
- **Biconnected Graph -** It is a connected graph that has no articulation points.
- **Biconnected Component** – A Biconnected component of a connected graph G is a maximal biconnected subgraph H of G.
- By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H.

- **A Biconnected Graph** has just one biconnected component, the whole graph.
- Two biconnected components of the same graph can have atmost one vertex in common. i.e., no edge can be in two or more biconnected components.


A Connected Graph


Its biconnected components

- Therefore, biconnected components of G partition the edges of G.
- Biconnected components can be found by Depth-First spanning tree of G.
- For the graph of figure, a depth first spanning tree with root 3 is shown. The sequence in which vertices are visited is also mentioned in the diagram. This is called depth first number (dfn) of the vertex.



- The non-tree edge (3,1) and (5,7) are called backedges.
- The root of the depth first spanning tree is an articulation point iff, it has atleast two children.
- Any other vertex u is an articulation point iff, it has atleast one child w, such that it is not possible to reach an ancestor of u using a path composed solely of w, and descendents of w, and a single back edge.
- $low(w)$ is the lowest dfn that can be reached from w using a path of descendents followed by, atmost one backedge.

$$low(w) = \left\{ \begin{matrix} dfn(w) \\ \min\{low(x) \mid x \text{ is child of } w\} \\ min\{dfn(x) \mid (w,x) \text{is a backedge}\} \end{matrix} \right\}$$

Vertex	0	**1**	2	**3**	4	**5**	6	**7**	8	9
dfn	5	4	3	1	2	6	7	8	9	10
low	5	1	1	1	1	6	6	6	10	9

- The vertex u is an articulation point iff u is either the root of the spanning tree and has two or more children or u is not the root and u has a child w such that $low(w) \geq dfn(u)$. The dfn and low values are listed in the table.