Ch.1: Introduction to Algorithms

Part 4

Fundamental Data Structures

Fundamental data structures

- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue
- graph
- tree
- set and dictionary

Data structures

- A data structure can be defined as a **particular scheme of organizing related data items**.
- Data items can range from elementary data types (e.g., integers or characters) to data structures (e.g., a one-dimensional array for implementing matrices)
- **Deciding the proper data structure:**
  - The vast majority of algorithms of interest operate on **data**.
  - To design an algorithm, **we should use the proper data structure**.
  - Particular ways of organizing data **play a critical role** in the design and analysis of algorithms.

Some algorithms require some nontrivial organization of input data. Some algorithm design techniques are based on structuring and restructuring data (heap sort for example).

We will see how **different algorithms uses different data structures**.

Data structures

- There are two main categories of data structures: **Linear** and **non-linear** data structures

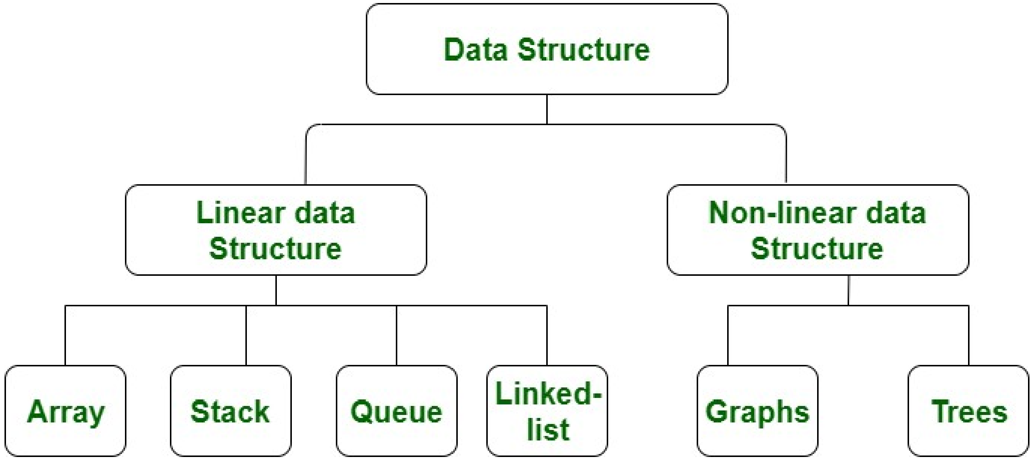
1) **Linear data structures:**

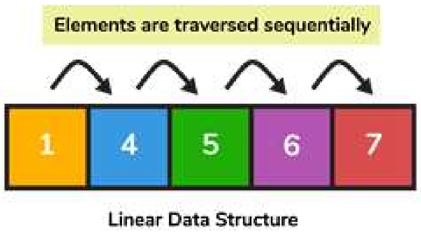
- A Linear data structure have data elements arranged in **sequential manner** and each member element is connected to its previous and next element.
- **Easy to implement** as computer memory is also sequential.
- Examples: **Array, Linked List, Stack, Queue**

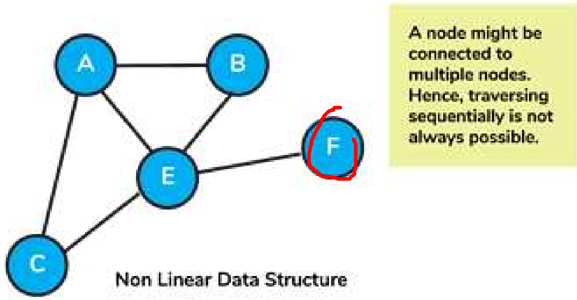
2) **Non-Linear Data structures:**

- The data elements are not arranged in a contiguous manner (the arrangement is nonsequential). And an element can be connected to more than two elements.
- are not easy to implement but are more efficient in utilizing computer memory.
- Examples: **Graphs, Trees**

Data structures







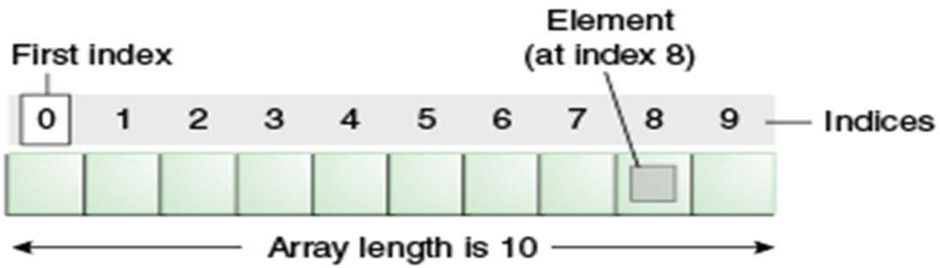
$O(1)$ $O(n)$

Arrays and Linked Lists

بعتمد عالحجم

Arrays

- **What is array?**
  - A (one-dimensional) array is a sequence of ***n*** items of the **<u>same data type</u>** that are **<u>stored contiguously</u>** in computer memory and made accessible by specifying a value of the array's **<u>index</u>**.



- In the majority of cases, the **index** is an integer either between **0 and n − 1** (as shown in the Figure) or between **1 and n.**

- Each and every element of an array can be accessed in **the same constant amount of time** regardless of where in the array the element in question is located.

Linked List

❑ What is a linked list?

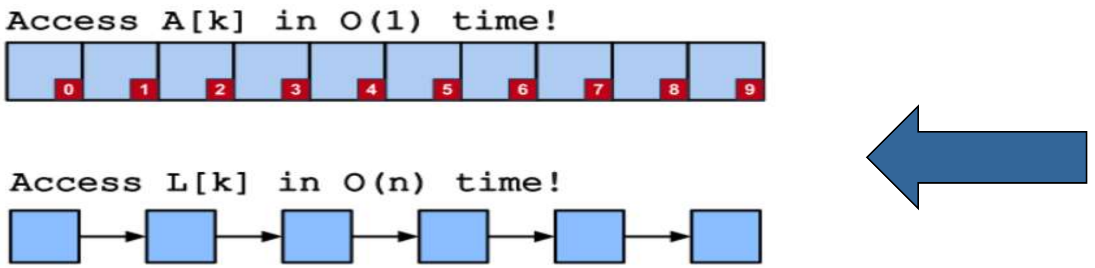
- A linked list is a sequence of zero or more elements called nodes, each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.

Singly linked list (SLL)	Doubly linked list (DLL)
SLL nodes contains 2 field -**data field and next link field**.	DLL nodes contains 3 fields -**data** field, a **previous** link field and a **next** link field.
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in **one direction only**.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in **both directions** (forward and backward).
The SLL occupies **less memory** than DLL as it has only 2 fields.	The DLL occupies **more memory** than SLL as it has 3 fields.

Array vs Linked List

Arrays and Linked Lists are both linear data structures, but both have some advantages and disadvantages over each other

Arrays	Linked Lists
Fixed length (need preliminary reservation of memory)	**Dynamic length (S**ize of a Linked list grows/shrinks as and when new elements are inserted/deleted**)**
contiguous memory locations	**arbitrary memory locations** (New elements can be stored anywhere and a reference is created for the new element using pointers.)
Direct access (Array elements can be accessed randomly using the array **index**.)	To access a particular node of a linked list, one starts with the list's first node and traverses the pointer chain until the particular node is reached (The elements will have to be accessed sequentially.)
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	**Insertion and Deletion** operations are fast and easy in a linked list.



Array vs Linked List

❑ Advantages of Linked Lists

- **Size** of linked lists is **not fixed**, they can expand and shrink during run time.
- Insertion and Deletion Operations are **fast and easier** in Linked Lists.
- Memory allocation is done during run-time (no need to allocate any fixed memory).
- Data Structures like Stacks, Queues, and trees can be easily implemented using Linked list.

❑ Disadvantages of Linked Lists

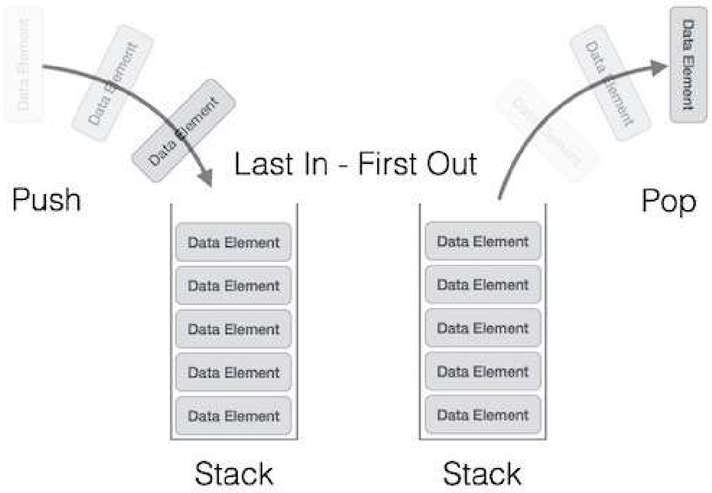
- **Memory consumption is more** in Linked Lists when compared to arrays.
- **Elements cannot be accessed at random** in linked lists.
- Traversing from reverse is not possible in singly linked lists.

❑ Applications of Linked Lists

- Linked Lists can be used to implement Stacks, Queues and Trees.
- Linked Lists can be also used to implement **Graphs**. (Adjacency list representation of Graph).

Stack and Queue

Stack



- **Stack** is a linear data structure
- Follows the **LIFO** (Last-In-First-Out) principle or FILO(First In Last Out)
- **insertion/deletion can be done only at the top**
- **basic operations**
  - **Push**: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
  - **Pop**: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
  - **Peek** or **Top**: Returns the top element of the stack without removing it.
  - **isEmpty**: Returns true if the stack is empty, else false.
- Implementation:
  - Using array
  - Using linked list

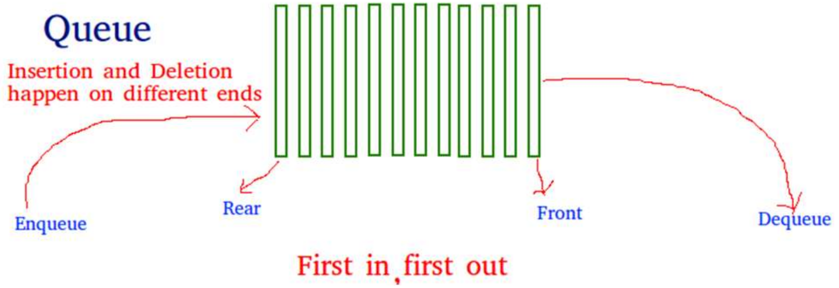
Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take **O(1) time**. We do not run any loop in any of these operations.

Queue

- ❑ **A queue of customers waiting for services**
- ❑ Follows the **FIFI** (First-In-First-Out) principle
- ❑ **Insertion** (enqueue) from the **rear** , **deletion** (dequeue) from the **front**.
- ❑ **basic operations**
  - **enqueue** – add (store) an item to the queue.
  - **dequeue** – remove (access) an item from the queue.
  - **Front**: Get the front item from queue.

    Rear: Get the last item from queue.
  - **isEmpty**: Checks if the queue is empty.



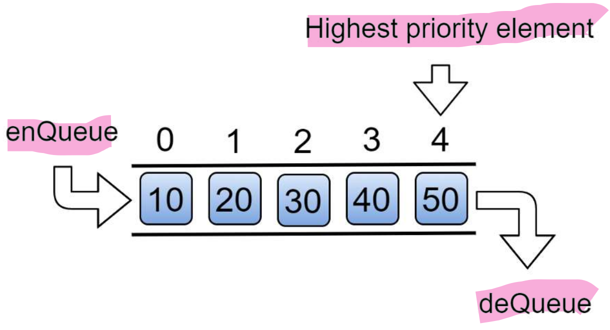
- ❑ Implementation:
  - Using array
  - Using linked list

we need to keep track of two indices, front and rear

Priority Queue

Queue

- **Priority queue** is a data structure similar to a normal queue except that each element has a certain priority.
- The priority of the elements in the priority queue determine the order in which elements are removed from the PQ.
- **Note**: The data inserted into the PQ **must be able to be ordered** in some way (from least to greatest or greatest to least)

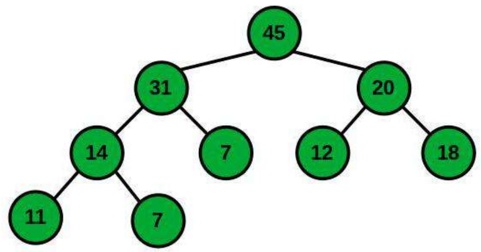


Dequeue highest priority element

Queue

- **Principle Operations**:
  - Finding the element with the highest priority.
  - Deleting the element with the highest priority.
  - Inserting a new element.
- **Implementation of PQ:**
- A better implementation of a priority queue is based on an ingenious data structure called the **heap (We Will discuss it Later)**
- **Some applications of PQ:**
  - Scheduling jobs on computer
  - Heap sort
  - Prim's algorithm
  - Load balancing

Max heap



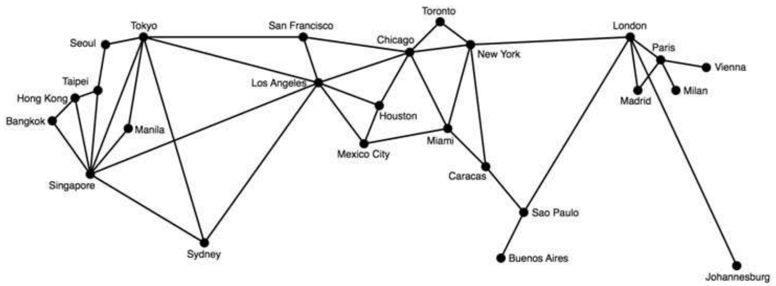
0	1	2	3	4	5	6	7	8
45	31	20	14	7	12	18	11	7

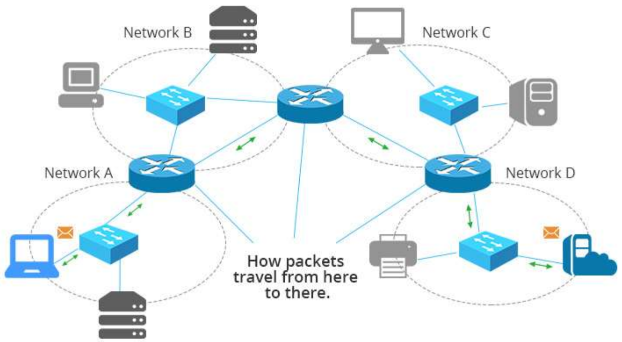
Basics of Graphs

Graphs and Trees

Graphs

- Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm.
- *A typical **example** of a graph is a network of roads and cities in a country*

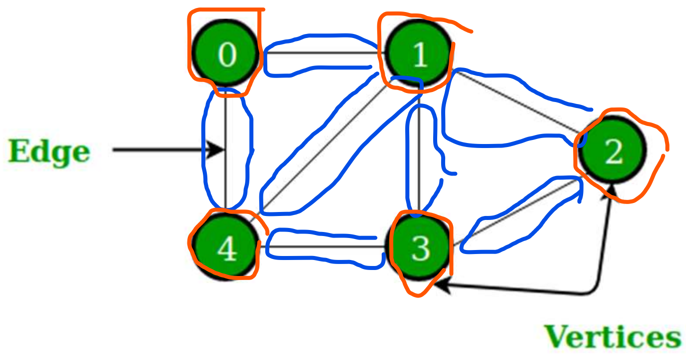




Graph terminology

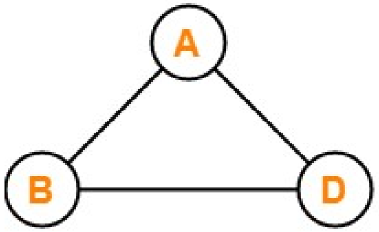
- A graph consists of **nodes** and **edges**.
- Informally, graph is thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs."
- *Formally*, a graph **G = (V, E)** is defined by a pair of two sets:
  - a finite nonempty set **V** of items called **vertices**
  - and a set **E** of pairs of these items called **edges**

Any graph is denoted as **G = {V, E}.**

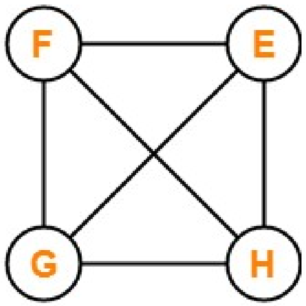


Graph terminology

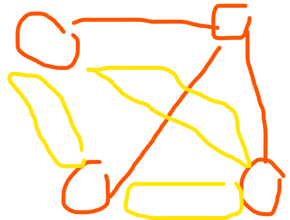
- **Complete graph:**
  - A graph with every pair of its vertices connected by an edge is called complete.
  - A standard notation for the complete graph with |V| vertices is $\mathbf{K_{|V|}}$



K_3



K_4



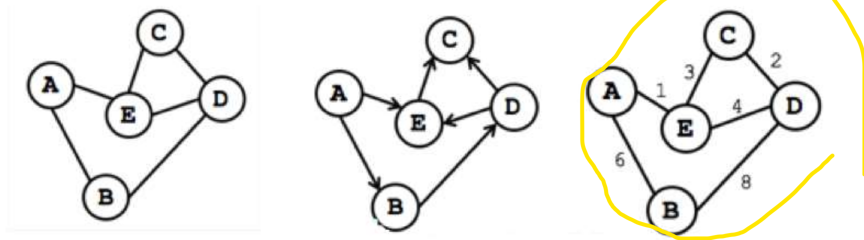
هاي مش complete

لانه مش كل زوج من

النود بينه إدج

Graph terminology

- **Undirected graph:**
  - A graph **G** is called undirected if **every edge in it is undirected**.
  - a pair of vertices (u, v) is the same as the pair (v, u) when they are connected by un undirected edge.
- **Directed graph (digraph):**
  - A graph whose **every edge is directed** is called directed.
  - a pair of vertices (u, v) is not the same as the pair (v, u)
  - we say that the edge (u, v) is directed from the vertex u, to the vertex v
- **Weighted graph:**
  - a graph (or digraph) with numbers assigned to its edges
  - These numbers are called ***weights*** or ***costs***.



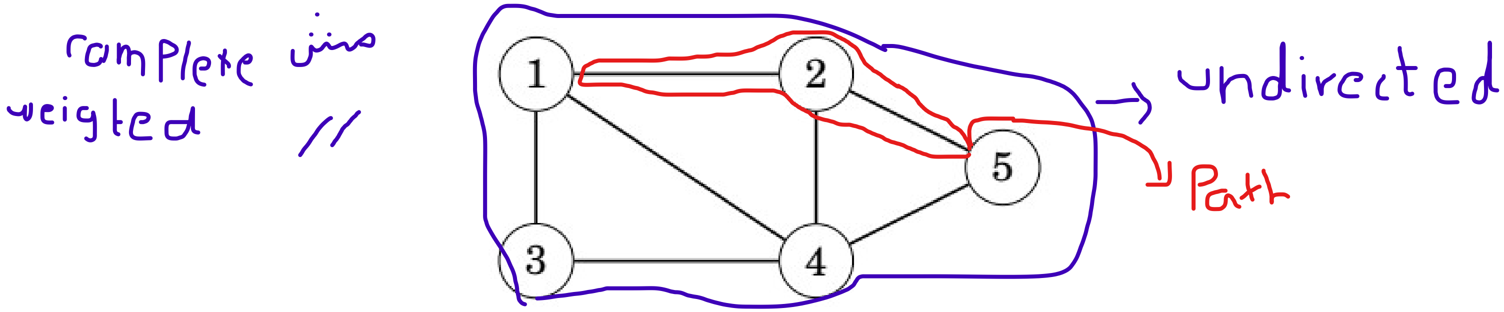
(A) Undirected Graph (B) Directed Graph (C) Weighted Graph

هاي الأرقام

يدل على معلومات معينة مثل مسافة سعر ....

Graph terminology

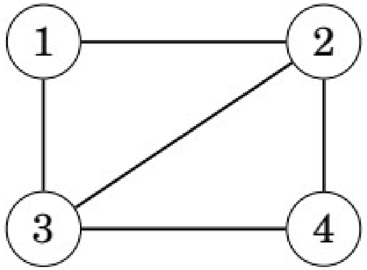
- For example, the following graph consists of 5 nodes and 7 edges:



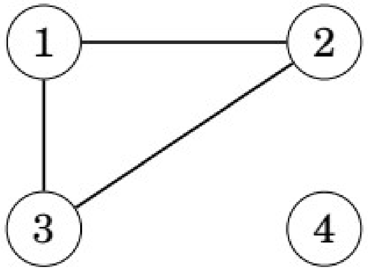
- A **path** leads from node ***a*** to node ***b*** through edges of the graph. The length of a path is the number of edges in it. For example, the above graph contains a path 1→ 3 → 4 → 5 of length 3 from node 1 to node 5.
- A path is a cycle if the first and last node is the same. For example, the above graph contains a cycle 1 → 3 → 4 → 1

Graph Connectivity

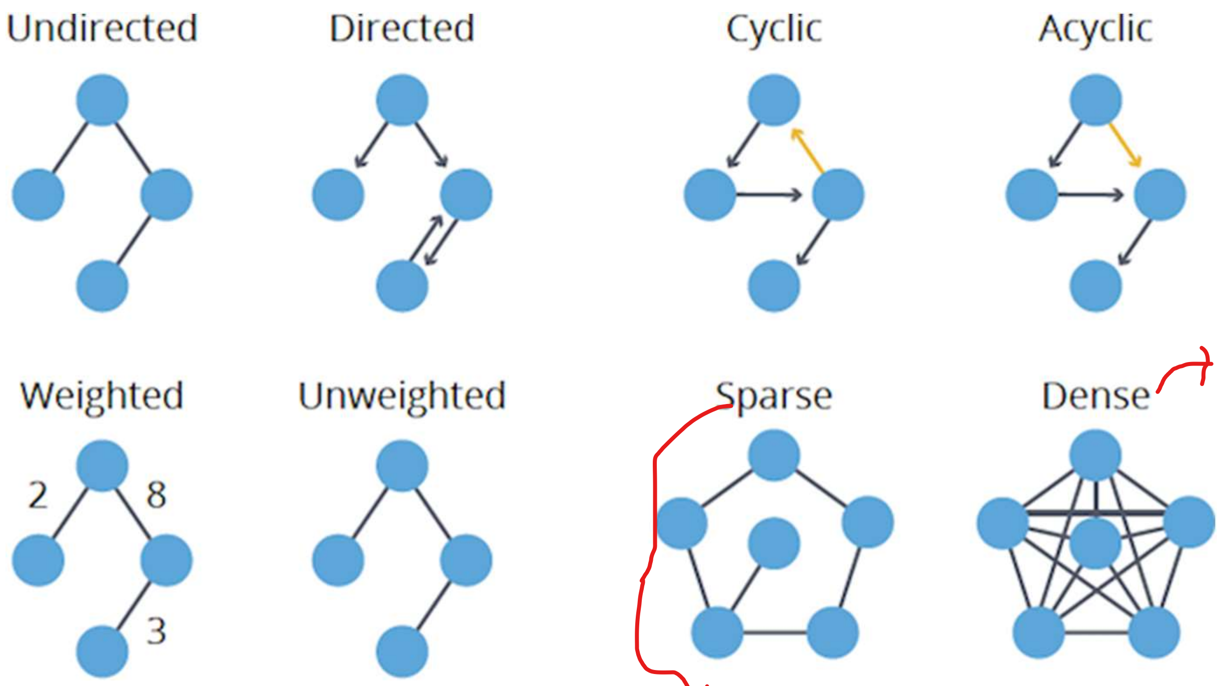
- A graph is **connected** if there is a path between any two nodes. For example, the following graph is connected.



- The following graph is **not connected (disconnected)**, because it is not possible to get from node 4 to any other node:



Graph Properties



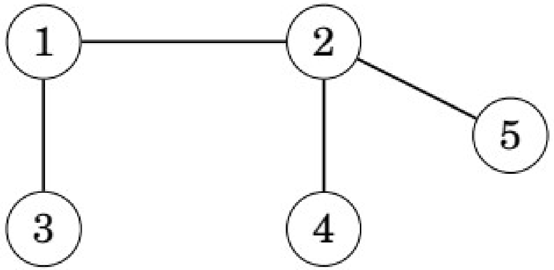
بكون فيها كثير edge لدرجة توصل لـ complete

ما بكون فيها كثير edge

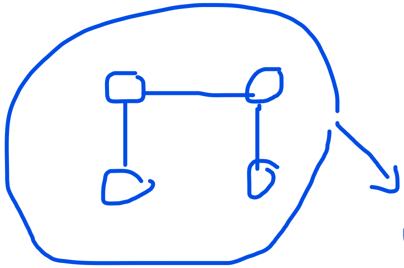
- **Cyclic vs Acyclic graphs????**

Tree

- A **tree** is a **connected graph** that consists of **n nodes** and ***n-1*** edges.
- There is a unique path between any two nodes of a tree.
- For example, the following graph is a tree:



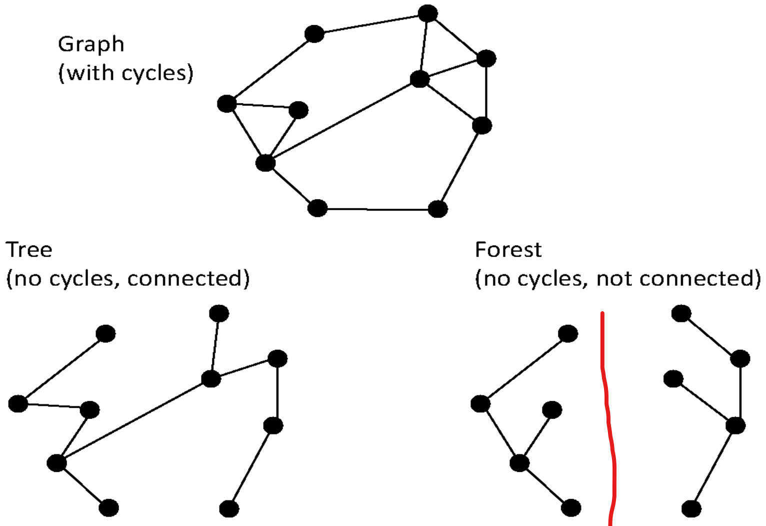
ليس كل graph هي tree
لكن العكس ✓



ليست tree لانه في ال tree لازم
يكون بين كل 2node edge

Trees

- A **tree** is a **connected acyclic** graph. $E = n-1$
  - **connected**: if for every pair of its vertices u and v there is a path from u to v.
  - **Acyclic**: A graph with **no cycles**.
- A **forest**, A graph that has **no cycles** but is not necessarily connected (**disconnected**).



في فاصل

tree مش بينهن.

Trees

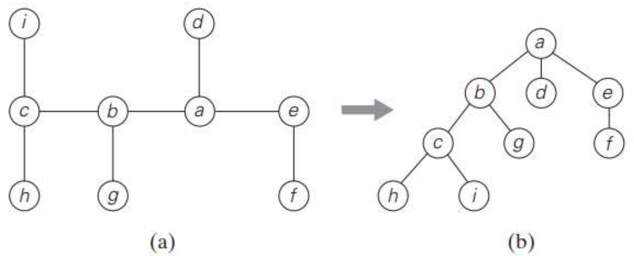
- Trees have several important properties other graphs do not have
- **Properties of trees:**
  - The **number of edges** in a tree is always one less than the number of its vertices

$$|E| = |V| - 1$$

  - **Rooted trees**: for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other (connected) . This property makes it possible to **select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree**.
- **Transformation from free tree to rooted tree:**

A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on

Free tree



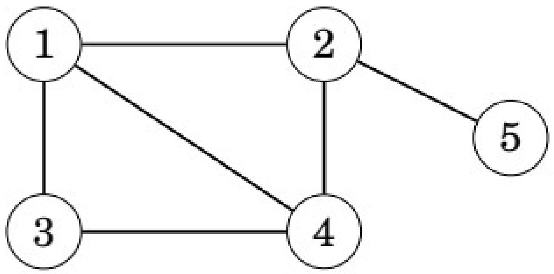
Rooted tree

Similarities and differences of graph and tree

No.	Graph	Tree
1	Graph is a non-linear data structure.	Tree is a non-linear data structure.
2	It is a collection of vertices/nodes and edges.	It is a collection of nodes and edges.
3	Each node can have any number of edges.	General trees consist of the nodes having any number of child nodes. But in case of **binary trees** every node can have at the most two child nodes.
4	**There is no unique node called root in graph.**	**There is a unique node called root in trees.**
5	**A cycle can be formed.**	**There will not be any cycle.**
6	Applications: For finding shortest path in networking, Google maps …… etc.	Applications: Binary search tree, .Heap, Spanning tree, B+ tree, Decision trees … etc

Neighbors and degrees

- Two nodes are ***neighbors*** or ***adjacent*** if there is an edge between them.
- The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.

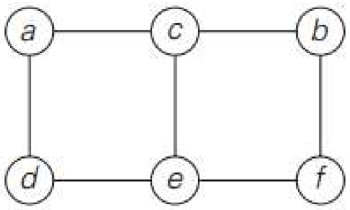


- Exercise: What is the degree of every node in a complete graph?

$n-1$

Graph Representation

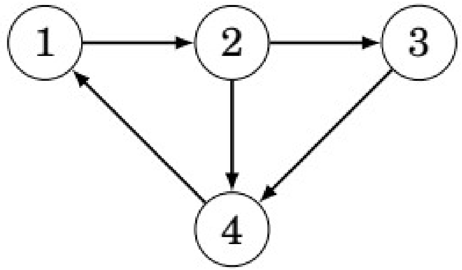
- Graphs for computer algorithms are usually represented in one of two ways: the ***adjacency matrix*** and ***adjacency lists***.
- **1) Adjacency matrix:**
  - The adjacency matrix of a graph with ***n*** vertices is an **n × n Boolean matrix** with one row and one column for each of the graph's vertices.
  - The element in the i^{th} row and the j^{th} column is equal to **1** if there is an edge from the i^{th} vertex to the j^{th} vertex, and equal to **0** if there is no such edge.
- Example:



$$
\begin{array}{c c}
 & \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} &
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

Symmetric for undirected graph. Why?

Graph Representation



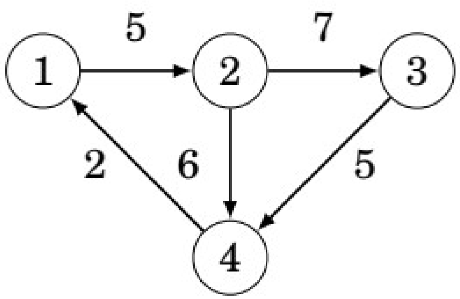
- can be represented as follows:

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

مش symmetric لإن
directed graph

Graph Representation

- If the graph is **weighted**, the adjacency matrix representation can be extended so that the matrix contains the weight of the edge if the edge exists. Using this representation, the graph



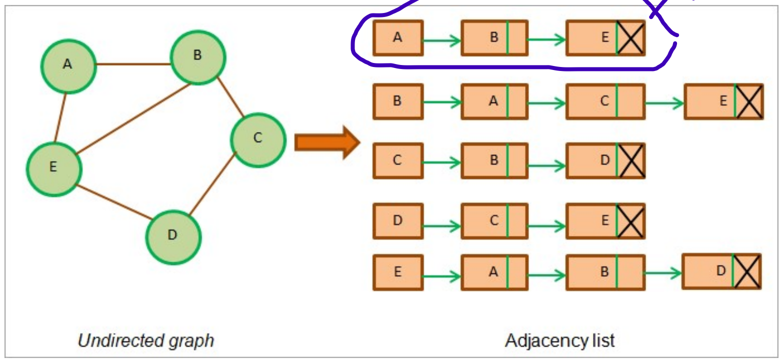
	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

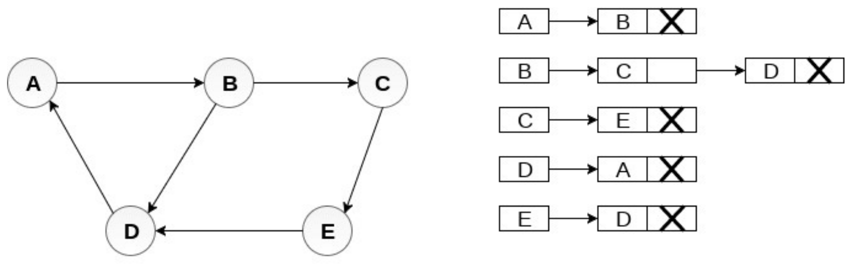
The drawback of the adjacency matrix representation is that the matrix contains n2 elements, and usually most of them are zero. For this reason, the representation cannot be used if the graph is large.

Graph Representation

- **2) Adjacency linked lists:**
  - The adjacency lists of a graph or a digraph is **a collection of linked lists**, one for each vertex, that contain all the vertices adjacent to the list's vertex
  - Examples:

لكل نود بحجز لنكد لست
خاص للنود A و B و E) مرتبطين معها بـ edge



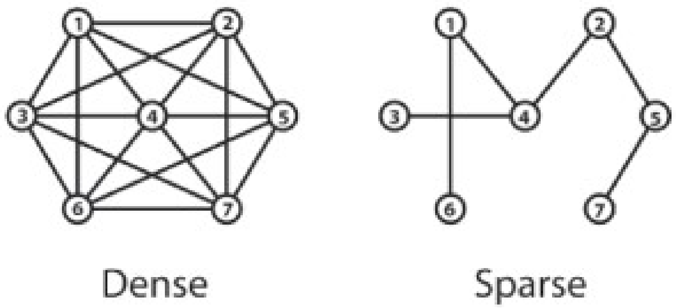


Directed Graph

Adjacency List

Graph Representation

- **Which data structure is better? Adjacency matrix or adjacency linked lists?**
  - In general, which of the two representations is more convenient **depends** on the nature of the problem, on the algorithm used for solving it, and, possibly, on the type of input graph (**sparse** or **dense**).
  - If a graph is ***sparse***, the **adjacency list representation may use less space** than the corresponding adjacency matrix.
  - If a graph is ***dense***, the **adjacency matrix representation may use less space** than the corresponding **adjacency list** . (Why???)



Exercise

Considering the following adjacency matrix representation of the graph G

	v1	v2	v3	v4	v5
v1	0	1	0	1	1
v2	0	0	0	1	0
v3	0	0	0	0	1
v4	0	0	0	0	0
v5	0	1	0	0	0

1. Is the graph is directed or undirected? Explain why?
2. Draw the graph based on the above adjacency matrix.
3. Show the adjacency linked list representation of the graph.