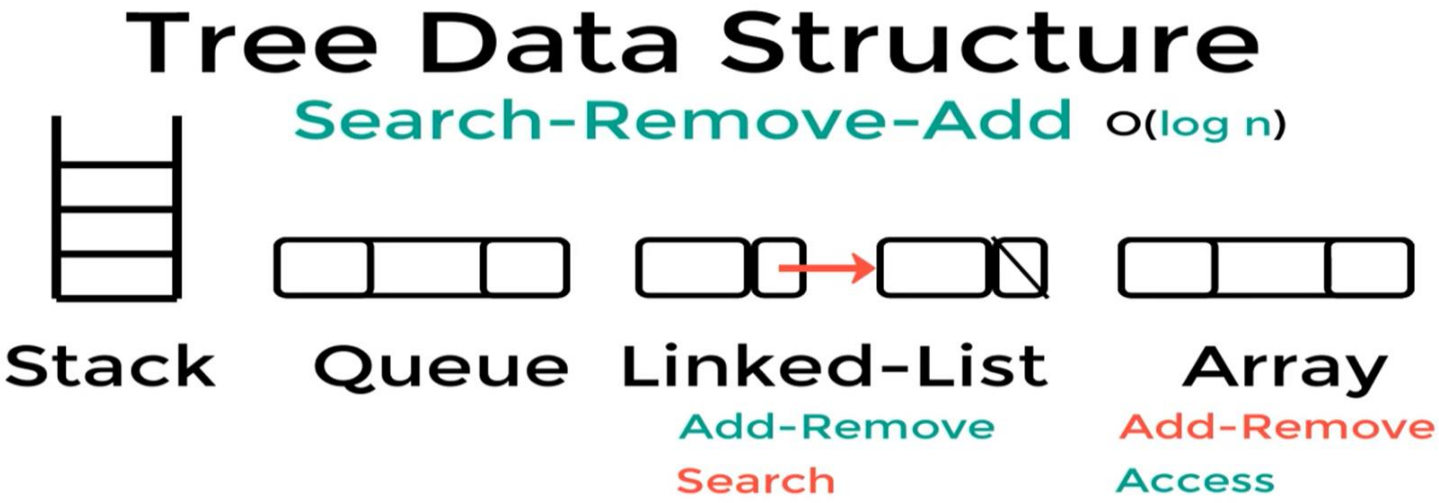
Data Structures and Algorithms

Lecture 7-8

© Fall 2025- **Dr. Hesham Sakr**
Building, office No. 12
Hesham.sakr@sut.edu.eg

WHAT ARE TREES?


Tree Data Structure
Search-Remove-Add
O(log n)
Stack
Queue
Linked-List
Array
Add-Remove
Search
Add-Remove
Access

Why Trees?

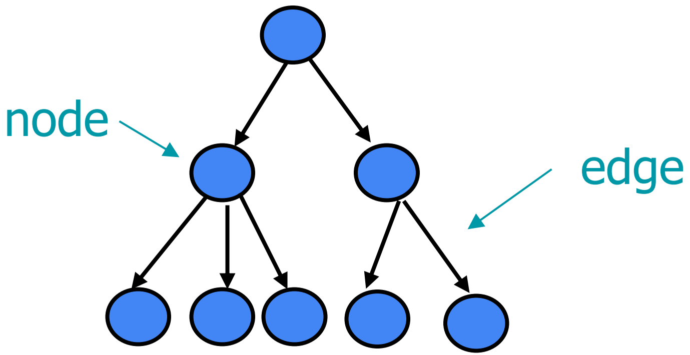
- - Efficient for searching, insertion, and deletion (e.g., BST).
- - Represent hierarchical data (e.g., file systems).
- - Useful in:
-   - Priority queues (binary heaps).
-   - Machine learning (decision trees).
-   - Compilers (Abstract Syntax Trees).

Applications of Trees

- - File systems (hierarchical structure).
- - Database indexing (B-Trees, B+ Trees).
- - Compilers (expression parsing).
- - Networking (optimal routing).
- - Machine learning (decision-making structures).

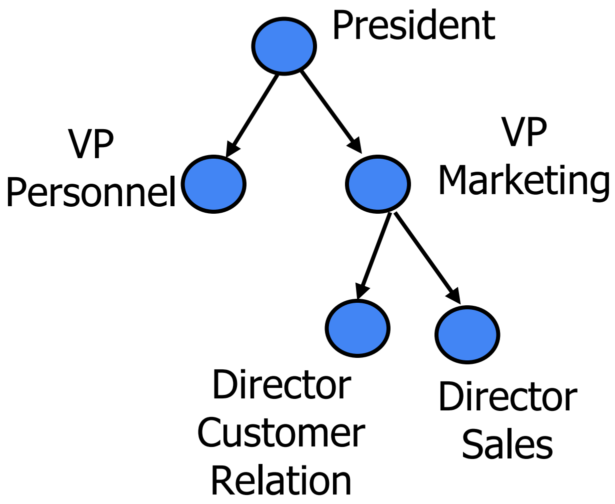
What is a tree?

- Trees are structures used to represent hierarchical relationship
- Each tree consists of nodes and edges
- Each node represents an object
- Each edge represents the relationship between two nodes.

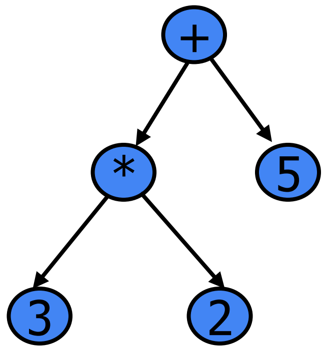


Some applications of Trees

Organization Chart

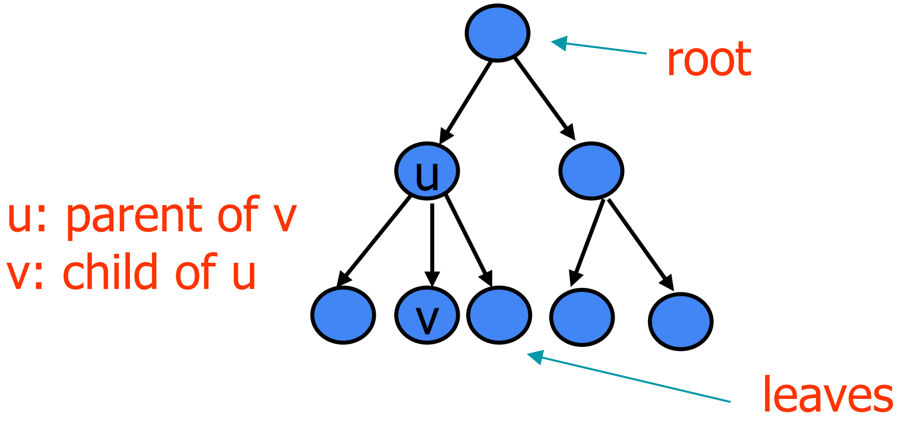


Expression Tree



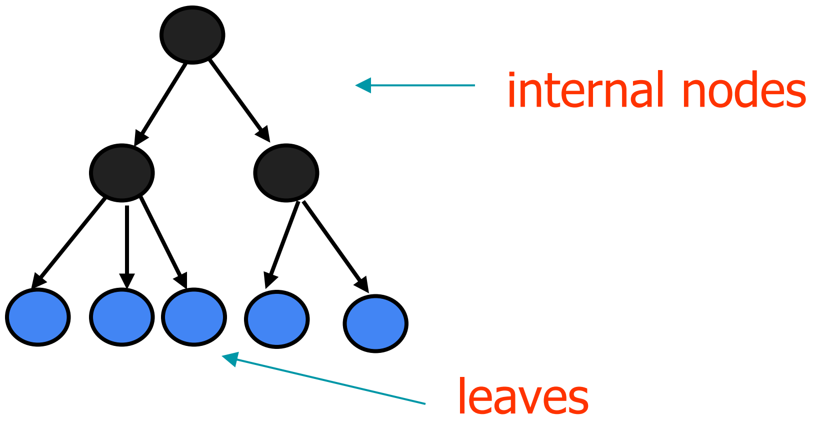
Terminology I

- For any two nodes u and v, if there is an edge pointing from u to v, u is called the parent of v while v is called the child of u. Such edge is denoted as (u, v).
- In a tree, there is exactly one node without parent, which is called the root. The nodes without children are called leaves.



Terminology II

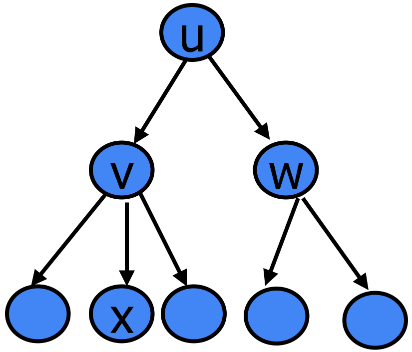
- In a tree, the nodes without children are called leaves. Otherwise, they are called internal nodes.



Terminology III

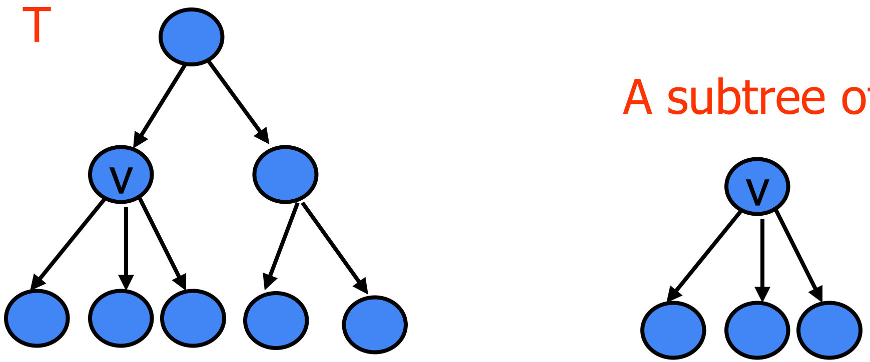
- If two nodes have the same parent, they are siblings.
- A node u is an ancestor of v if u is parent of v or parent of parent of v or …
- A node v is a descendent of u if v is child of v or child of child of v or …

v and w are siblings
u and v are ancestors of x
v and x are descendents of u



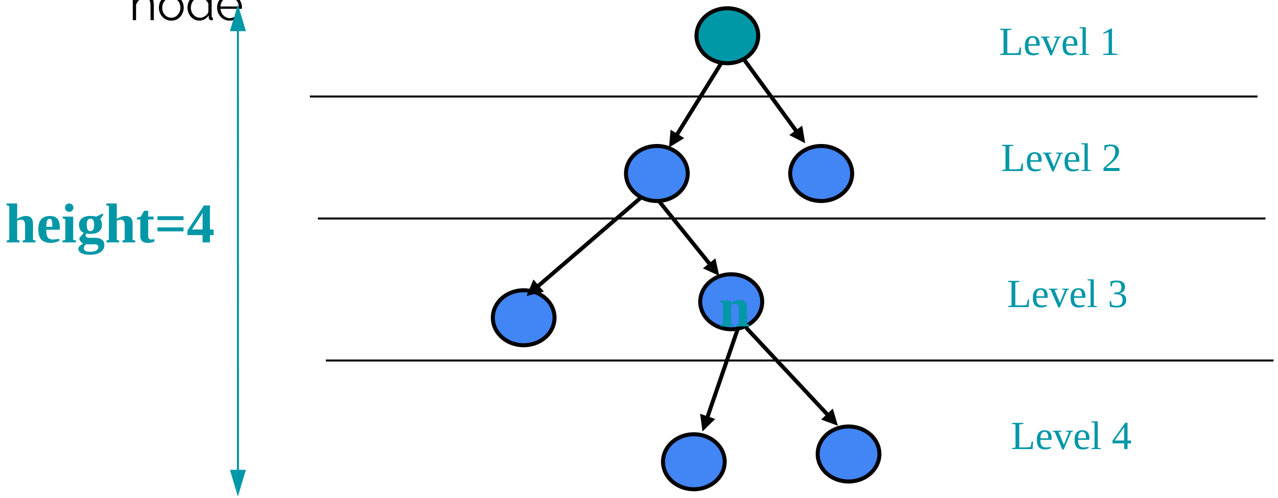
Terminology IV

- A subtree is any node together with all its descendants.

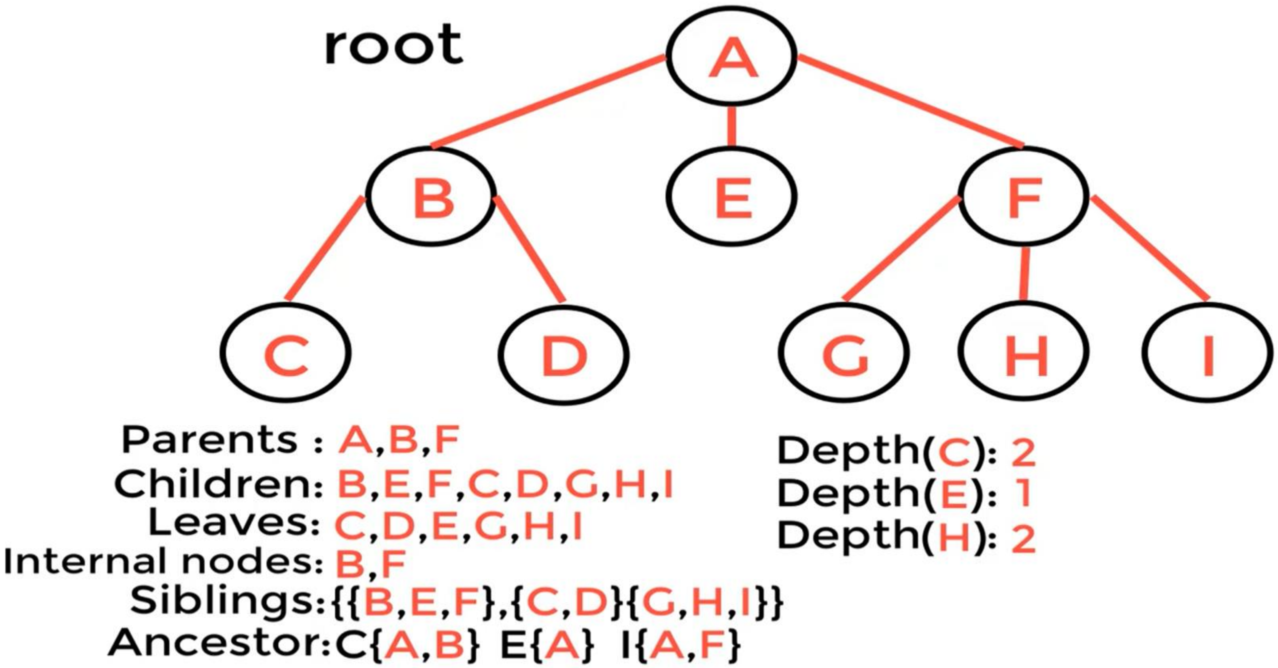


Terminology V

- Level of a node n: number of nodes on the path from root to node n
- Height of a tree: maximum level among all of its node

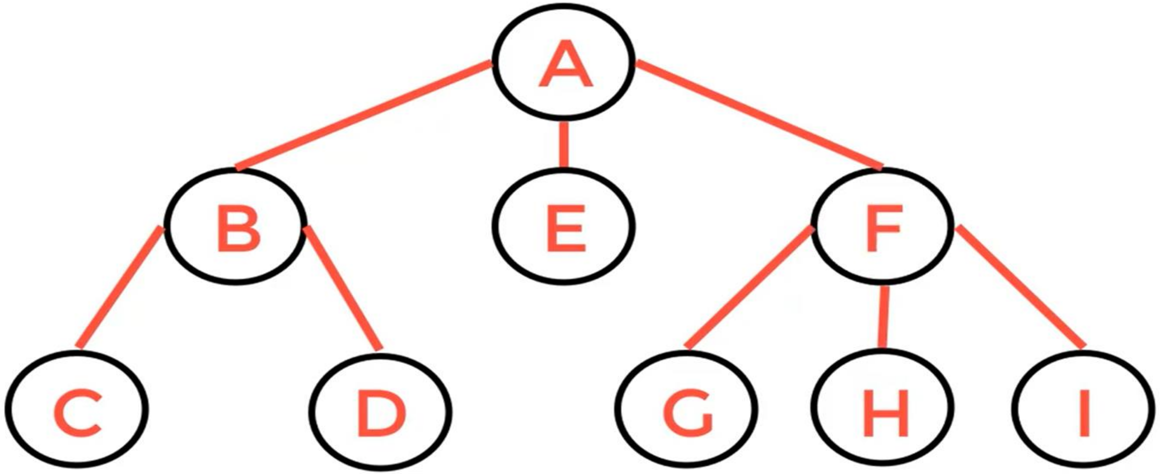


Example



root
A
B
E
F
C
D
G
H
I
Parents : A,B,F
Children: B,E,F,C,D,G,H,I
Leaves: C,D,E,G,H,I
Internal nodes: B,F
Siblings:{{B,E,F},{C,D}{G,H,I}}
Ancestor:C{A,B} E{A} I{A,F}
Depth(C): 2
Depth(E): 1
Depth(H): 2

important



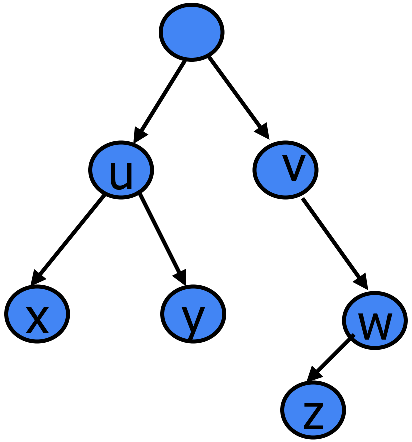
Parenthetical notation: A(B(C D)E F(G H I))

Tree types

- Binary tree Binary-search tree AVL tree Red-black tree.....
- B tree
- Heaps
- Trees
- Multiway tree
- Application specific tree

Binary Tree

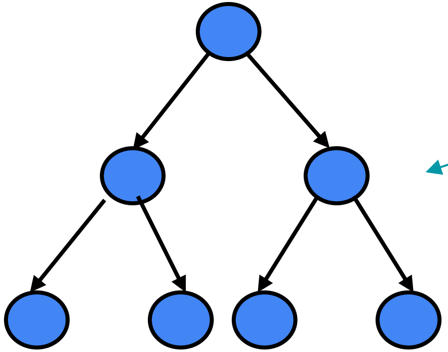
- Binary Tree: Tree in which every node has at most 2 children
- Left child of u: the child on the left of u
- Right child of u: the child on the right of u



x: left child of u
y: right child of u
w: right child of v
z: left child of w

Full binary tree

- If T is empty, T is a full binary tree of height 0.
- If T is not empty and of height h >0, T is a full binary tree if both subtrees of the root of T are full binary trees of height h-1.



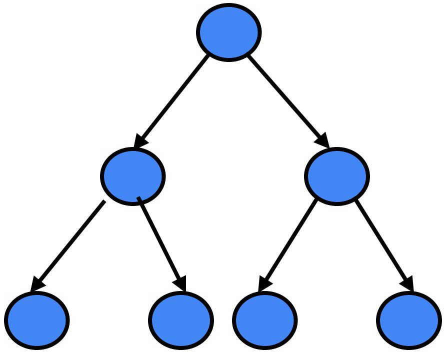
Full binary tree of height 3

Property of binary tree (I)

- A full binary tree of height h has 2^h-1 nodes

No. of nodes $= 2^0 + 2^1 + \ldots + 2^{(h-1)}$

$= 2^h - 1$



Level 1: 2^0 nodes

Level 2: 2^1 nodes

Level 3: 2^2 nodes

Property of binary tree (II)

- Consider a binary tree T of height h. The number of nodes of T $\leq 2^h-1$

Reason: you cannot have more nodes than a full binary tree of height h.

- The minimum height of a binary tree with n nodes is log(n+1)

By property (II), $n \leq 2^h-1$

Thus, $2^h \geq n+1$

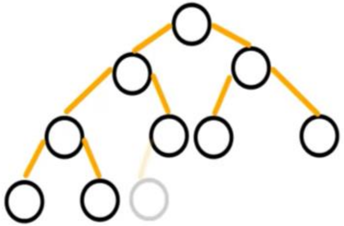
That is, $h \geq \log_2 (n+1)$

Other binary tree types

Complete Binary Tree

1.All levels is comletely filled except the last level.

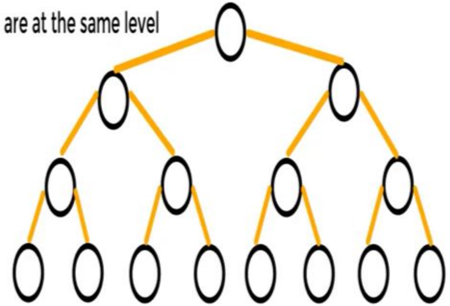
2.All Nodes as left as possible in last level



Perfect Binary Tree (All levels in completely filled)

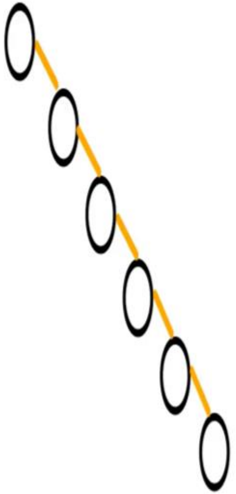
1.Every nodes has two children.

2.All leaves are at the same level



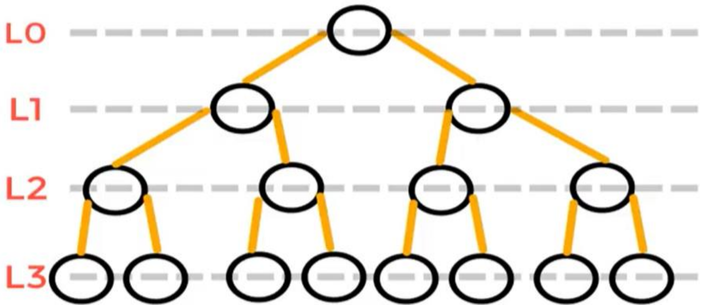
A degenerate (or pathological) Tree

- Every parent node has only one child either left or right.
- Such trees are performance-wise same as linked list.



Max no. of nodes at level $\rightarrow$ 2^L

Max no. of nodes in a binary tree $\rightarrow$ $2^{h+1} - 1$



$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

Find height of tree?!

$$n = 2^{h+1} - 1$$
$$2^{h+1} = (n + 1)$$
$$h = \log_2(n + 1) - 1$$
$$\log_2(16) \quad - 1$$

Reference Based Representation

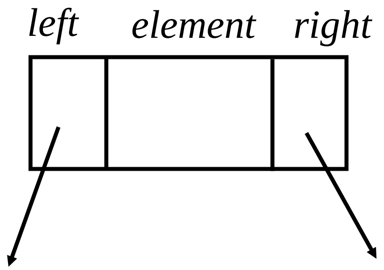
NULL: empty tree

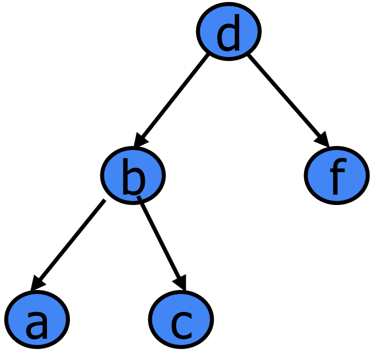
You can code this with a class of three fields:

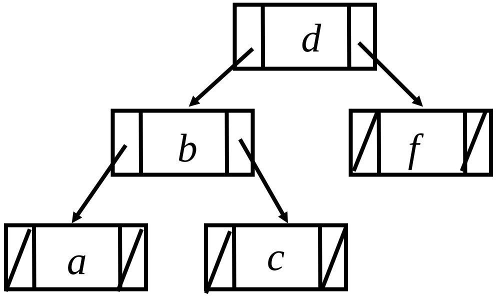
Object element;

BinaryNode left;

BinaryNode right;

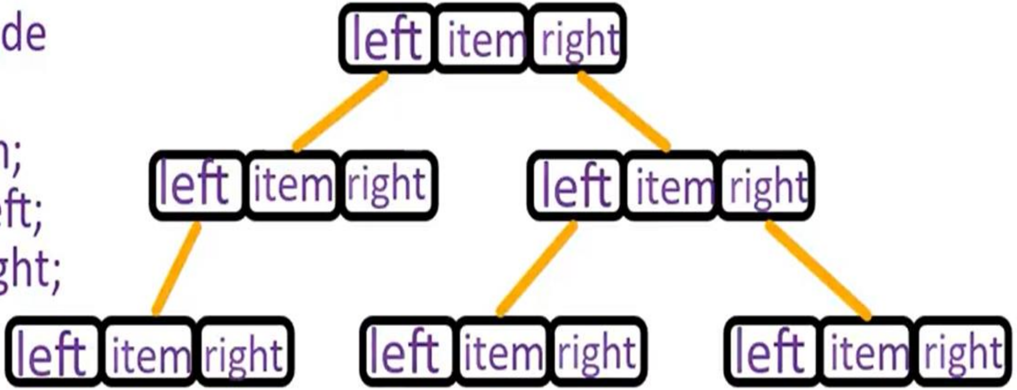






Coding

```
struct Node
{
Type item;
Node* Left;
Node* right;
}
```




```
class Node
{
Type item;
Node Left;
Node right;
}
```

Tree Traversal

- Given a binary tree, we may like to do some operations on all nodes in a binary tree. For example, we may want to double the value in every node in a binary tree.
- To do this, we need a traversal algorithm which visits every node in the binary tree.

Ways to traverse a tree

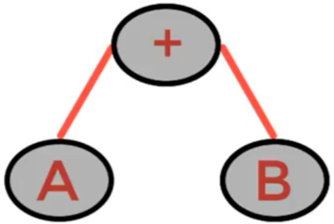
- There are three main ways to traverse a tree:
  - Pre-order:
    - (1) visit node, (2) recursively visit left subtree, (3) recursively visit right subtree
  - In-order:
    - (1) recursively visit left subtree, (2) visit node, (3) recursively right subtree
  - Post-order:
    - (1) recursively visit left subtree, (2) recursively visit right subtree, (3) visit node
  - Level-order:
    - Traverse the nodes level by level
- In different situations, we use different traversal algorithm.

Example

Binary Tree Traversal

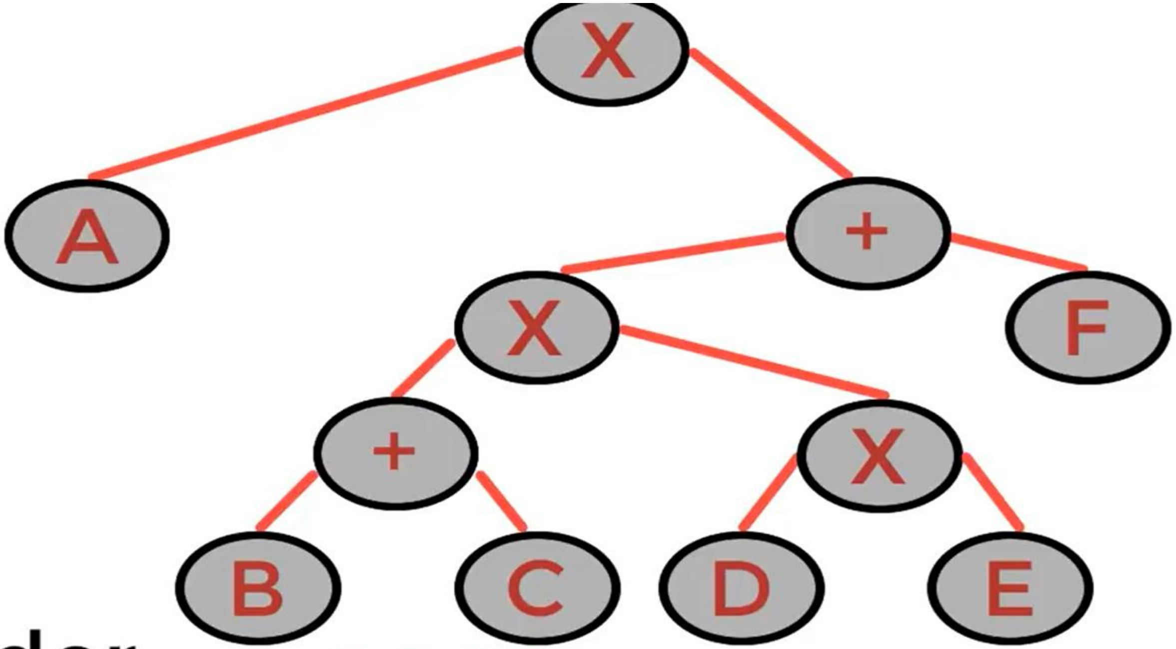
Pre-order : root left right
+ A B

In-order : left root right
A + B



Post-order : left right root
A B +

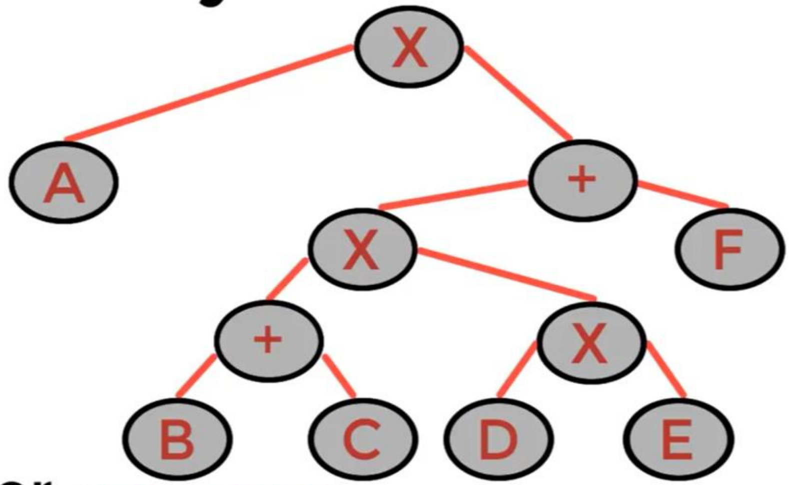
Examples for expression tree



Pre-order : root left right

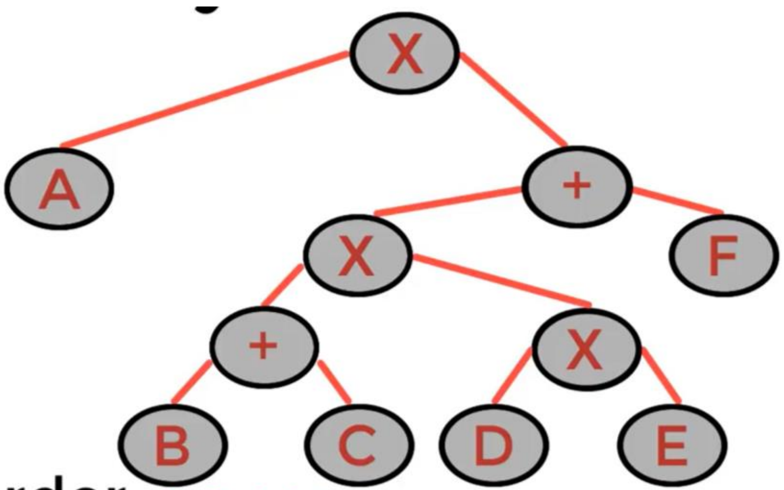
XA+X+BCXDEF

Binary Tree Traversal



In-order : left root right

AXB+CXDXE+F



Post-order : left right root

ABC+DEXXF+X

Traversal method types

Binary Tree Traversal

Breadth-first traversal:

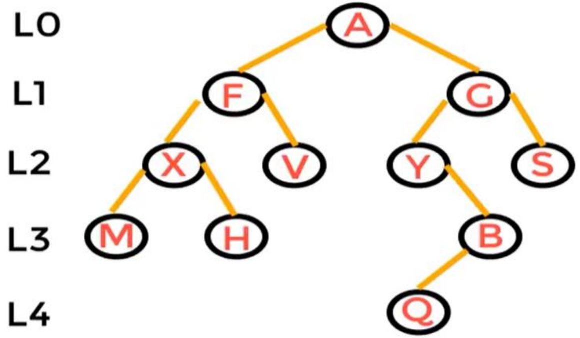
Level-order

A F G X V Y S M H B Q

Depth-first traversal:

- -Pre-order :root left right
- -In-order :left root right
- -Post-order :left right root

:root right left
:right root left
:right left root



WHAT ARE GRAPHS?

Why Graphs?

زوجية

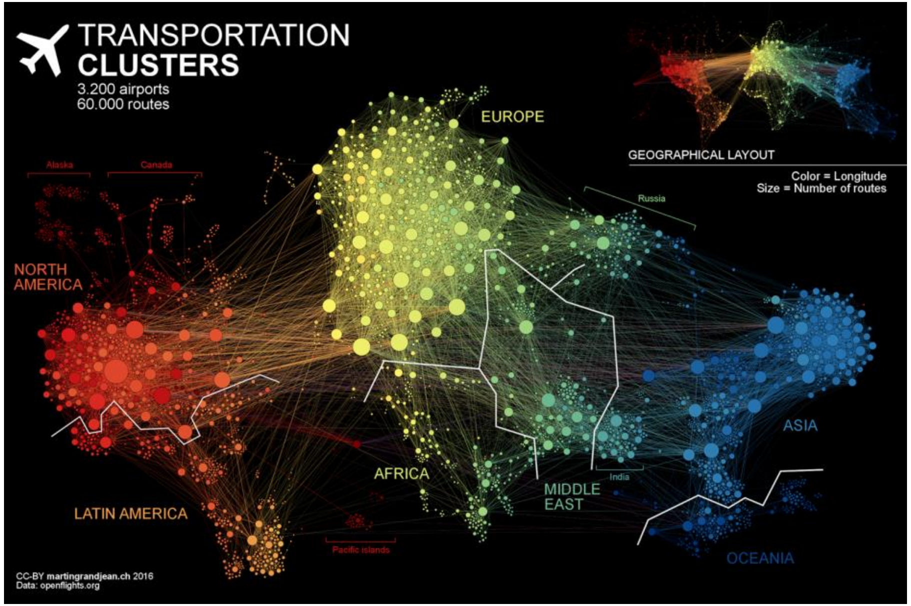
- - Represent pairwise relationships flexibly.
- - Useful in modeling real-world networks:
- - Social relationships.
- - Transportation systems.
- - Communication networks.
- - Enable powerful algorithms (e.g., Dijkstra, A*).

Applications of Graphs

- - Social networks (e.g., Facebook, LinkedIn).
- - GPS routing and internet protocols.
- - Search engines (e.g., Google's PageRank).
- - Electrical circuits representation.
- - Artificial Intelligence (state-space problems).

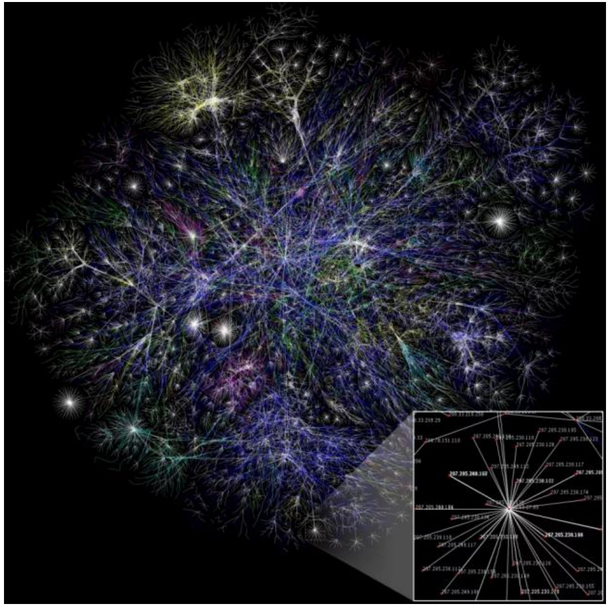
GRAPH EXAMPLES

Each "node" is an airport, and flight routes are represented by the "edge" in between them



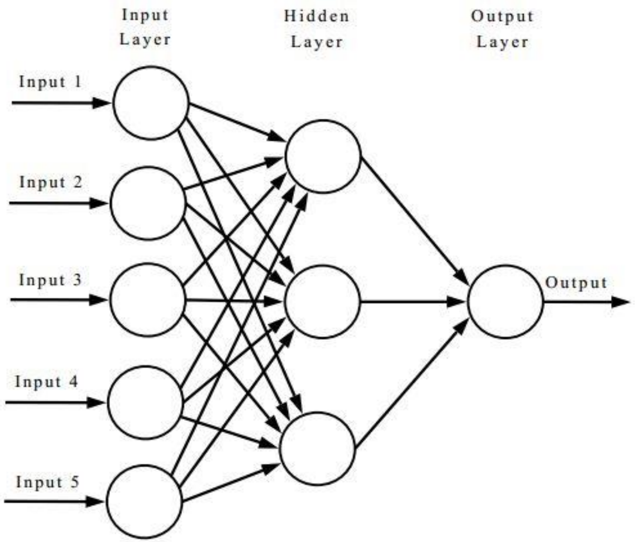
GRAPH EXAMPLES

Partial graph of the Internet (in 2005), where each “node” is an IP address, and the “edges” between them reveal connectivity delays (shorter lines = closer IP addresses)



GRAPH EXAMPLES

Neural networks! Each "node" represents a module of the neural network, and "edge" represent output/input relationships



GRAPH EXAMPLES

Graph of characters in the third book of Game of Thrones, where each “node” is a character, and “edge” reveal frequency of interaction (i.e. 2 names appearing within 15 words of one another).

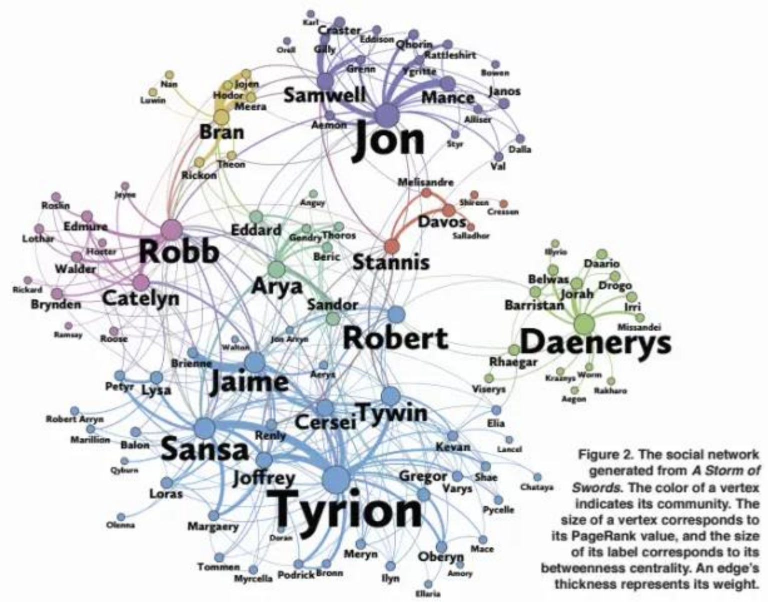
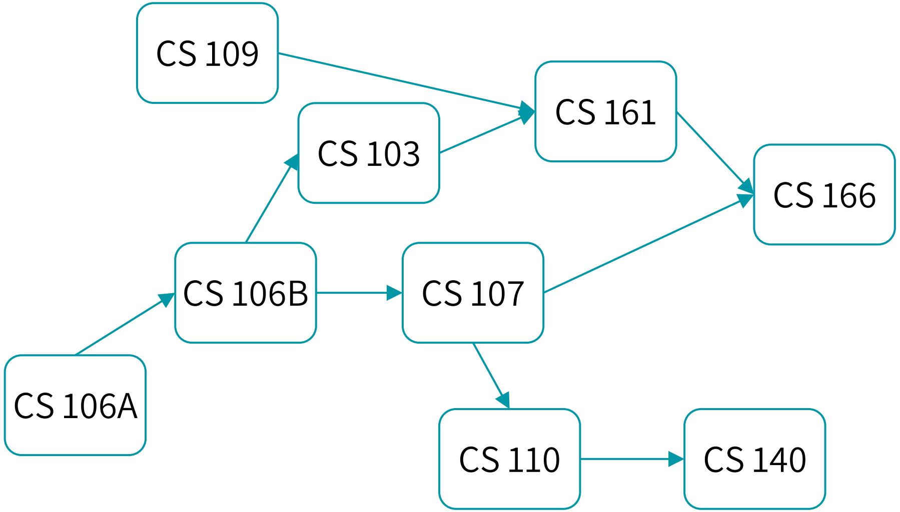


Figure 2. The social network generated from *A Storm of Swords*. The color of a vertex indicates its community. The size of a vertex corresponds to its PageRank value, and the size of its label corresponds to its betweenness centrality. An edge's thickness represents its weight.

GRAPH EXAMPLES

CS prerequisites!
“nodes” are classes and an “edge” from class A to class B means “class B depends on class A”



GRAPH EXAMPLES

- **To an algorithms person,** a graph means a representation of the relationships between pairs of Objects.
- **Road networks**. When your smartphone's software computes driving directions, it searches through a graph that represents the road network, with **vertices** corresponding to **intersections** and **edges** corresponding to **individual road segments**.

GRAPH EXAMPLES

- **The World Wide Web**. The Web can be modeled as a directed graph, with the **vertices** corresponding to **individual Web pages**, and the **edges** corresponding to **hyperlinks**, directed from the page containing the hyperlink to the destination page.
- **Precedence constraints**. Graphs are also useful in problems that lack an obvious network stucture. For example, imagine that you a first-year university student, **planning which courses to take and in which order**.

GRAPH EXAMPLES

- **Social networks.** A social network can be represented as a graph whose **vertices** correspond to **individuals** and **edges** to **some type of relationship**. For example, an edge could indicate a friendship between its endpoints, or that one of its endpoints is a follower of the other. Among the currently popular social networks, which ones are most naturally modeled as an **undirected** graph, and which ones as a **directed** graph?

WHAT ARE GRAPHS USED FOR?

- There are a lot of diverse problems that can be represented as graphs, and we want to answer questions about them

- For example:
    - How do we most efficiently route packets across the internet?
    - Are there natural “clusters” or “communities” in a graph?
    - Which character(s) are least related with _____?
    - How should I sign up for classes without violating pre-req constraints?

But first off, some terminology!

Differences: Trees vs. Graphs

Comparison	Tree	Graph
Relationship of node	Only one root node. Parent-Child relationship exists.	No root node. No Parent-Child relationship exists.
Path	Only one path between two nodes	One or more paths exist between two nodes
Edge	N – 1 (N = Number of nodes)	Can not defined
Loop	Loop is not allowed	Loop is allowed
Traversal	Preorder, Inorder, Postorder	BFS, DFS
Model type	Hierarchical	Network

KINDS OF GRAPHS

TWO KINDS OF GRAPHS:

1. Undirected Graph

2. Directed Graph

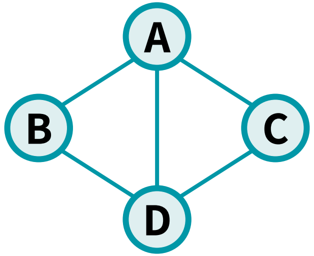
Two KINDS OF GRAPHS:

- In an undirected graph, each edge corresponds to an unordered pair {v,w} of vertices, which are called the endpoints of the edge. In an undirected graph, there is no difference between an edge (v,w) and an edge (w, v).

UNDIRECTED GRAPHS

An undirected graph has
a set of vertices (V) & a set of edges (E)

Formally,
G = (V, E)



V = {A, B, C, D}
E = { {A, B}, {A, C}, {A, D}, {B, D}, {C, D}}

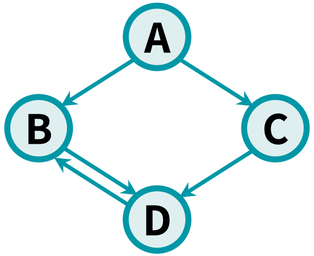
Two KINDS OF GRAPHS:

- In a directed graph, each edge (v,w) is an ordered pair, with the edge traveling from the first vertex v (called the tail) to the second w (the head);

DIRECTED GRAPHS

A directed graph has
a set of vertices (V) & a set of **DIRECTED** edges (E)

Formally,
G = (V, E)



The **in-degree** of vertex D is 2. The **out-degree** of vertex D is 1.
Vertex D's **incoming neighbors** are A, B, & C
Vertex D's **outgoing neighbor** is B

ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

GRAPH REPRESENTATIONS OPTIONS

GRAPH REPRESENTATIONS OPTIONS:

OPTION 1: ADJACENCY MATRIX

OPTION 2: ADJACENCY LIST

GRAPH REPRESENTATIONS

OPTION 1: ADJACENCY MATRIX

- The adjacency matrix representation of G is a square n x n matrix A equivalently, a two-dimensional array—with only zeroes and ones as entries. Each entry A_{ij} is defined as:

$$A_{ij} = \begin{cases} 1 & \text{if edge } (i, j) \text{ belongs to } E \\ 0 & \text{otherwise.} \end{cases}$$

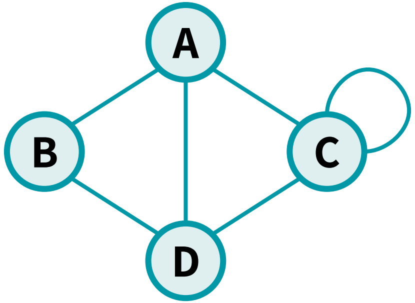
GRAPH REPRESENTATIONS

OPTION 2: ADJACENCY LIST

- Ingredients for Adjacency Lists
    1. An array containing the graph’s vertices.
    2. An array containing the graph’s edges.
    3. For each edge, a pointer to each of its two endpoints.
    4. For each vertex, a pointer to each of the incident edges.

GRAPH REPRESENTATIONS

OPTION 1: **ADJACENCY MATRIX**



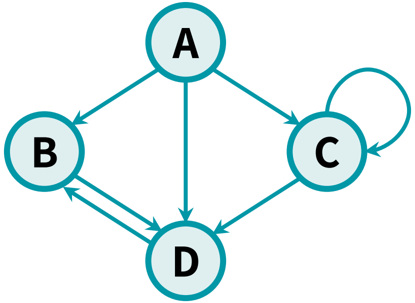
(An undirected graph)

(destination)

$$\text{(source)}\quad\begin{array}{c|c} & \begin{matrix} \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \end{matrix} \\ \begin{matrix} \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \\ \mathbf{D} \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

GRAPH REPRESENTATIONS

OPTION 1: **ADJACENCY MATRIX**



(A directed graph)

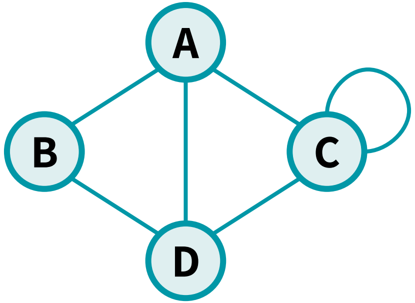
(destination)

(source)

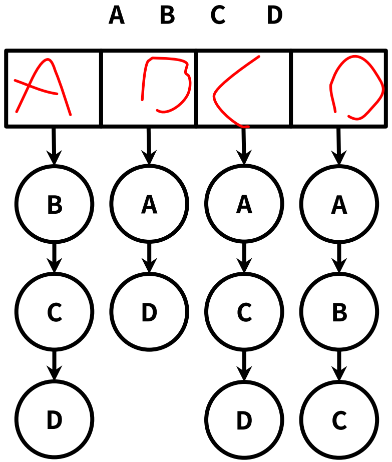
$$\begin{array}{cc} & \begin{array}{cccc} \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \end{array} \\ \begin{array}{c} \mathbf{A} \\ \mathbf{B} \\ \mathbf{C} \\ \mathbf{D} \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

GRAPH REPRESENTATIONS

OPTION 2: **ADJACENCY LISTS**



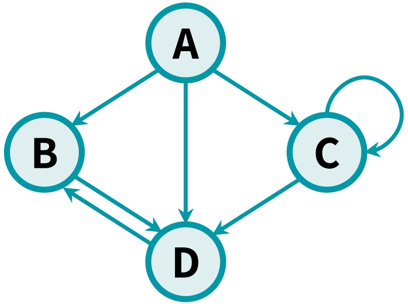
(An undirected graph)



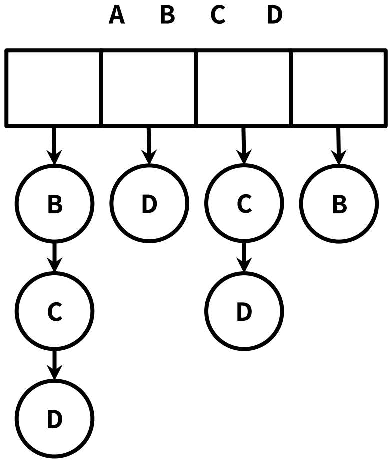
Each list stores a node's neighbors

GRAPH REPRESENTATIONS

OPTION 2: **ADJACENCY LISTS**



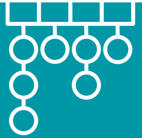
(A directed graph)



Tracks *outgoing neighbors*.

(You could also do the same for incoming neighbors as well)

GRAPH REPRESENTATIONS

GRAPH REPRESENTATIONS TECKNIQUES	ADJACENCY MATRIX	ADJACENCY LISTS
For a graph G = (V, E) where $\lvert V \rvert$ = **n**, and $\lvert E \rvert$ = **m**	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
SPACE REQUIREMENTS	$O(n^2)$	$O(n + m)$

Generally, better for sparse graphs (where m << n^2).

We'll assume this representation, unless otherwise stated.

GRAPH REPRESENTATIONS

Generally Adjacency Lists representation, better than Adjacency Matrix representation for sparse graphs (where $m < n^2$).

We’ll assume Adjacency Lists representation, unless otherwise stated.

Sparse Graphs vs. Dense Graphs

Sparse Graphs vs. Dense Graphs

- A graph is **sparse** if the number of edges is relatively close to **linear** in the **number of vertices**.
- A graph is **dense** if the number of edges is closer to **quadratic** in the **number of vertices**.

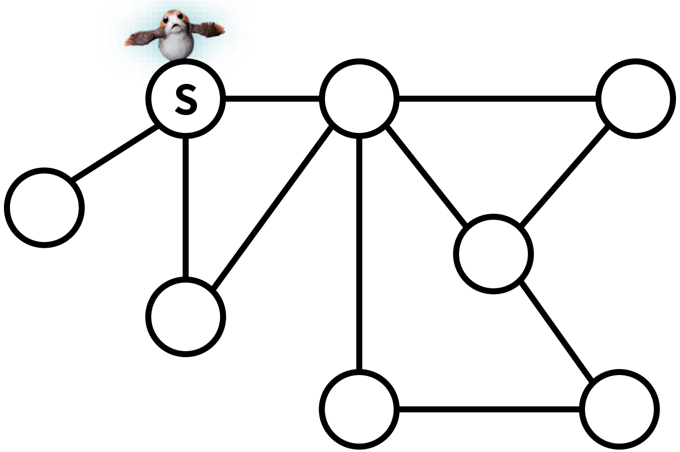
BREADTH-FIRST SEARCH (BFS)

One way to explore a graph!

BREADTH-FIRST SEARCH

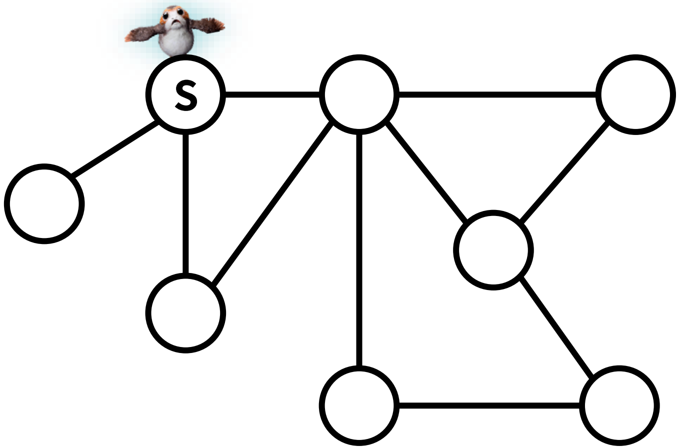
- Breadth-first search explores the vertices of a graph in layers, in order of increasing distance from the starting vertex.

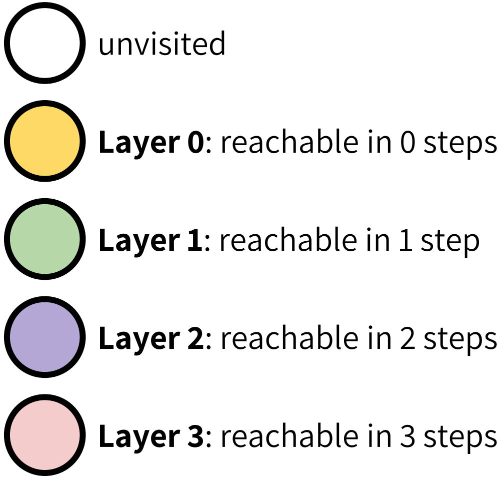
An analogy:
A bird is exploring a maze from above (with a bird's eye view)



BREADTH-FIRST SEARCH

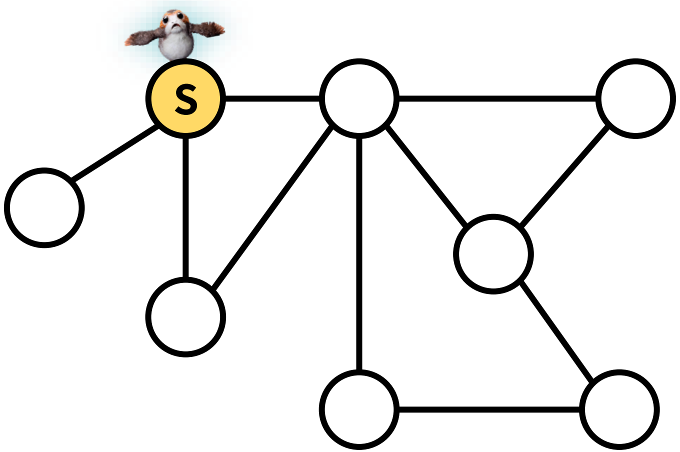
An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)





BREADTH-FIRST SEARCH

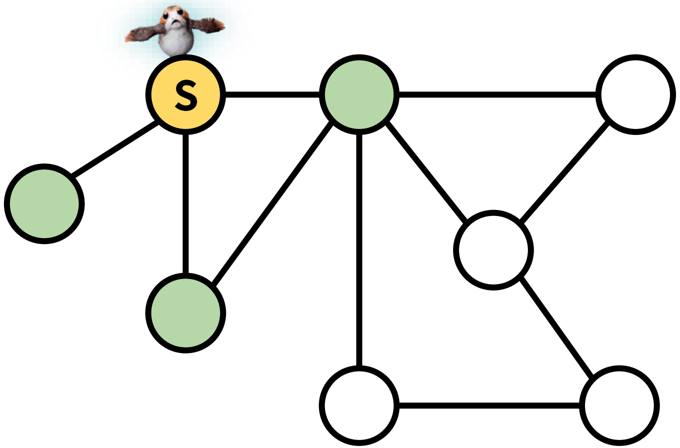
An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)



- unvisited
- **Layer 0**: reachable in 0 steps
- **Layer 1**: reachable in 1 step
- **Layer 2**: reachable in 2 steps
- **Layer 3**: reachable in 3 steps

BREADTH-FIRST SEARCH

An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)



unvisited

Layer 0: reachable in 0 steps

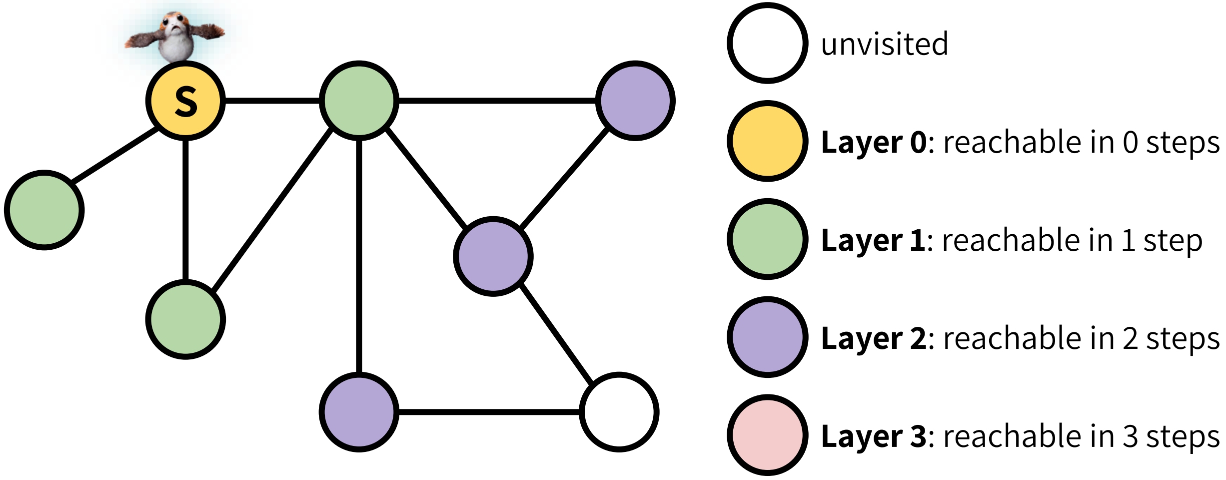
Layer 1: reachable in 1 step

Layer 2: reachable in 2 steps

Layer 3: reachable in 3 steps

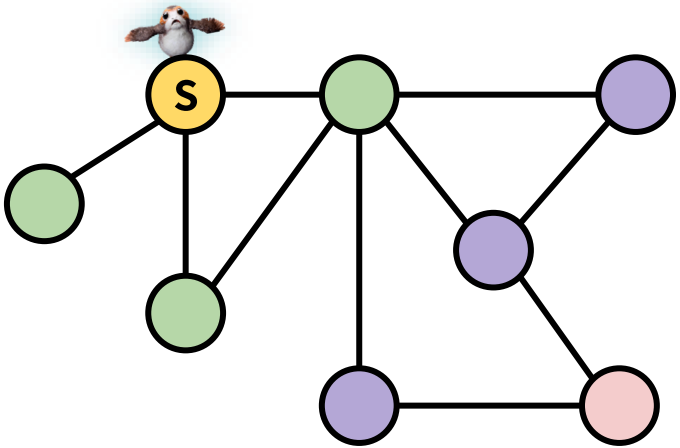
BREADTH-FIRST SEARCH

An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)



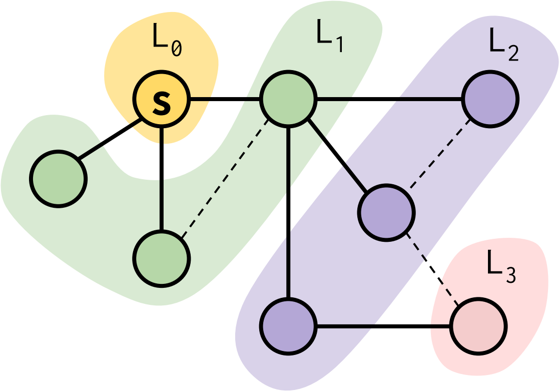
BREADTH-FIRST SEARCH

An analogy:
A bird is exploring a labyrinth from above (with a bird's eye view)



- unvisited
- **Layer 0**: reachable in 0 steps
- **Layer 1**: reachable in 1 step
- **Layer 2**: reachable in 2 steps
- **Layer 3**: reachable in 3 steps

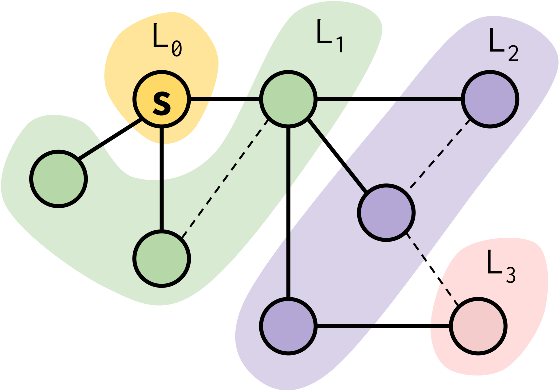
BREADTH-FIRST SEARCH



L_i = The set of nodes we can reach in i steps from s

```
BFS(s):
    Set L_i = [] for i = 0, ..., n-1
    L_0 = s
    for i = 0, ..., n-1:
        for u in L_i:
            for v in u.neighbors:
                if v not yet visited:
                    mark v as visited
                    add v to L_{i+1}
```

BREADTH-FIRST SEARCH



L_i = The set of nodes we can reach in i steps from s

```
BFS(s):
  Set Li = [] for i = 0, ..., n-1
  L0 = s
  for i = 0, ..., n-1:
    for u in Li:
      for v in u.neighbors:
        if v not yet visited:
          mark v as visited
          add v to Li+1
```

Go through all nodes in L_i and add their unvisited neighbors to L_{i+1}

ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

Thank You