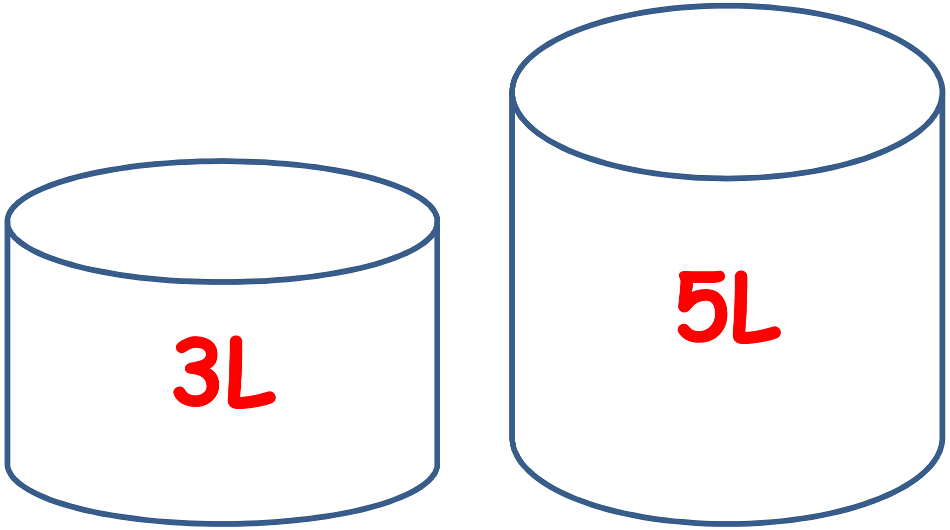
COMP108
Algorithmic Foundations

Graph Theory

Prudence Wong

http://www.csc.liv.ac.uk/~pwong/teaching/comp108/201617

How to Measure 4L?



a 3L container &
a 5L container
(without mark)

infinite supply of water

You can pour water from one container to another

How to measure 4L of water?

Learning outcomes

- Able to tell what an undirected graph is and what a directed graph is
  - Know how to represent a graph using matrix and list
- Understand what Euler circuit is and able to determine whether such circuit exists in an undirected graph
- Able to apply BFS and DFS to traverse a graph
- Able to tell what a tree is

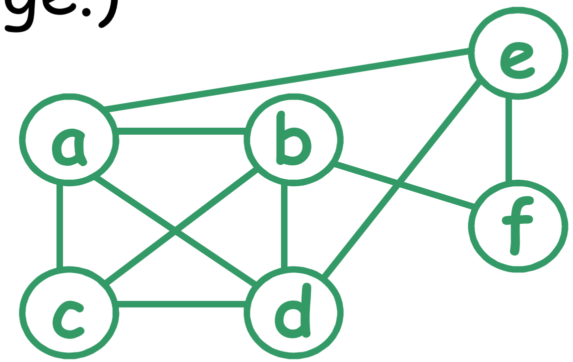
Graph …

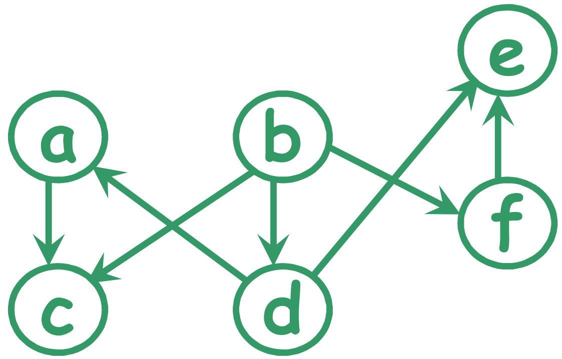
Graphs

introduced in the 18th century

Graph theory - an old subject with many modern applications.

An ***undirected*** graph G=(V,E) consists of a set of vertices V and a set of edges E. Each edge is an ***unordered*** pair of vertices. (E.g., {b,c} & {c,b} refer to the same edge.)



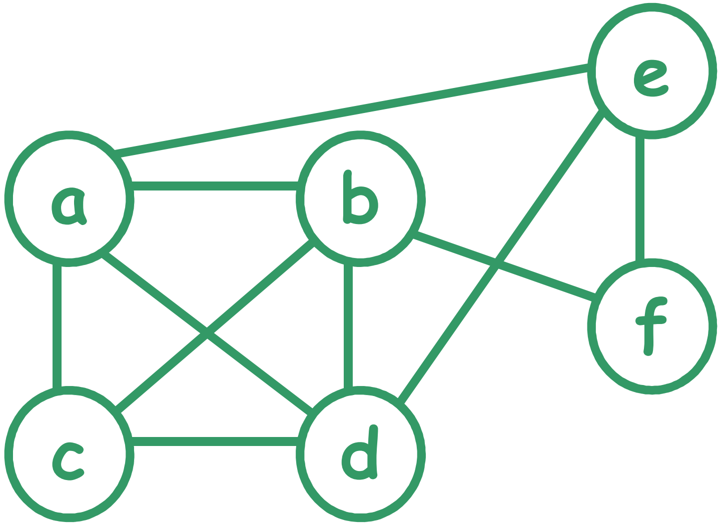


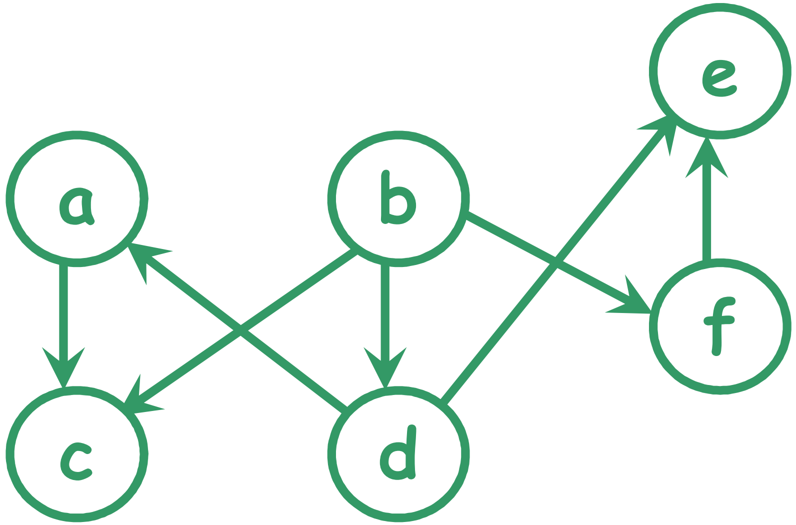
A ***directed*** graph G=(V,E) consists of … Each edge is an ***ordered*** pair of vertices. (E.g., (b,c) refer to an edge from b to c.)

Modeling Facebook & Twitter?

Graphs

represent a set of interconnected objects





undirected graph **directed graph**

Applications of graphs

In computer science, graphs are often used to model

- computer networks,
- precedence among processes,
- state space of playing chess (AI applications)
- resource conflicts, …

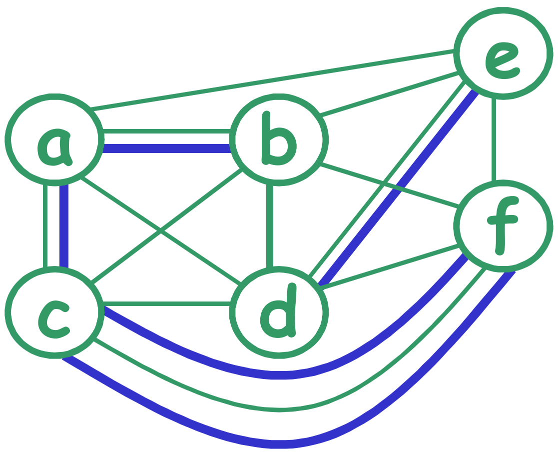
In other disciplines, graphs are also used to model the structure of objects. E.g.,

- biology - evolutionary relationship
- chemistry - structure of molecules

Undirected graphs

Undirected graphs:

- **simple graph:** at most one edge between two vertices, no self loop (i.e., an edge from a vertex to itself).
- **multigraph:** allows more than one edge between two vertices.

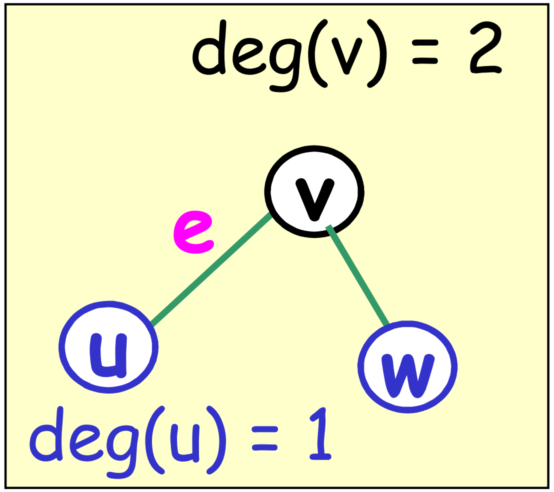


Reminder: An undirected graph G=(V,E) consists of a set of vertices V and a set of edges E. Each edge is an unordered pair of vertices.

Undirected graphs

In an undirected graph G, suppose that e = {u, v} is an edge of G

- u and v are said to be ***adjacent*** and called ***neighbors*** of each other.
- u and v are called ***endpoints*** of e.
- e is said to be ***incident*** with u and v.
- e is said to ***connect*** u and v.



- The ***degree*** of a vertex v, denoted by **deg(v)**, is the number of edges incident with it (a loop contributes twice to the degree); ***The degree of a graph is the maximum degree over all vertices***

Representation (of undirected graphs)

An undirected graph can be represented by **adjacency matrix**, **adjacency list**, **incidence matrix** or **incidence list**.

Adjacency matrix and adjacency list record the relationship between **vertex adjacency**, i.e., a vertex is adjacent to which other vertices

Incidence matrix and incidence list record the relationship between **edge incidence**, i.e., an edge is incident with which two vertices

Data Structure - Matrix

Rectangular / 2-dimensional array

- m-by-n matrix
  - m rows
  - n columns
- $a_{i,j}$
  - row i, column j

m-by-n matrix

$a_{i,j}$

n columns

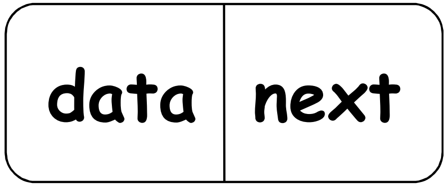
m rows

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,n} \end{pmatrix}$$

Data Structure - Linked List

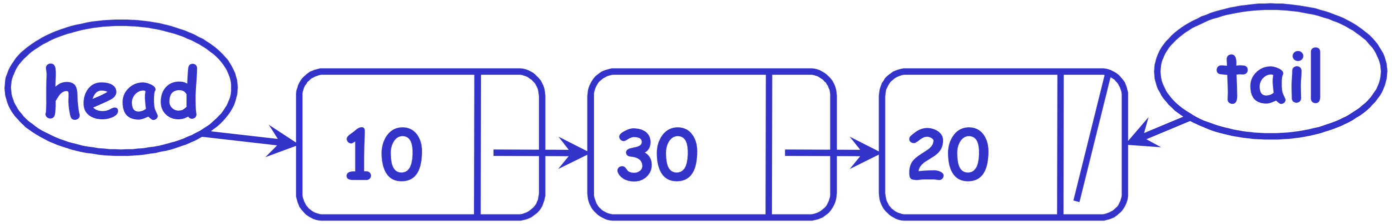
List of elements (nodes) connected together like a chain

Each node contains two fields:



- "data" field: stores whatever type of elements
- "next" field: pointer to link this node to the next node in the list

Head / Tail

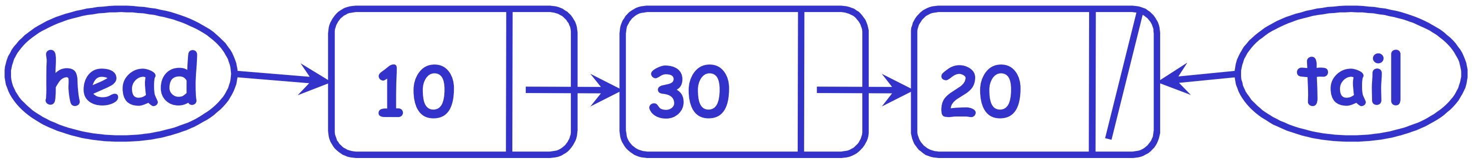


- pointer to the beginning & end of list

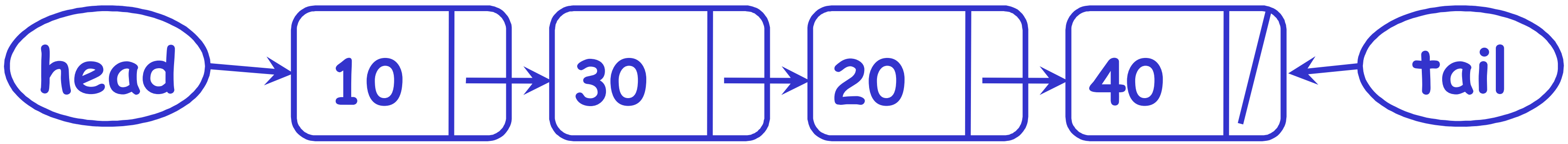
Data Structure - Linked List

Queue (FIFO: first-in-first-out)

- Insert element (enqueue) to tail
- Remove element (dequeue) from head

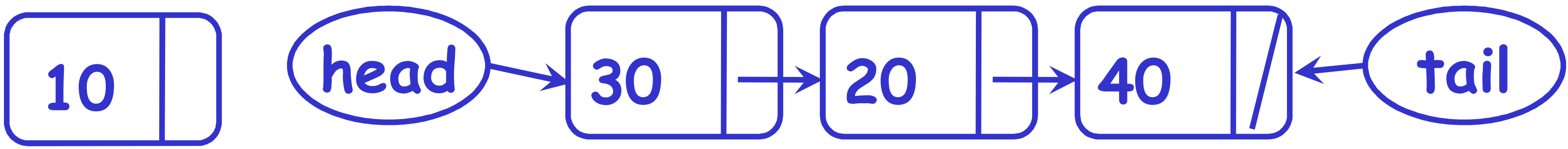


Insert 40



create newnode of 40; tail.next = newnode; tail = tail.next

Remove



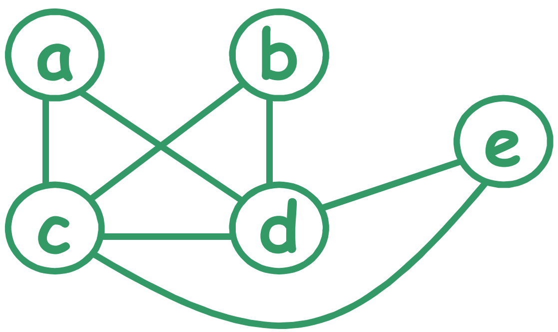
return whatever head points to; head = head.next

Adjacency matrix / list

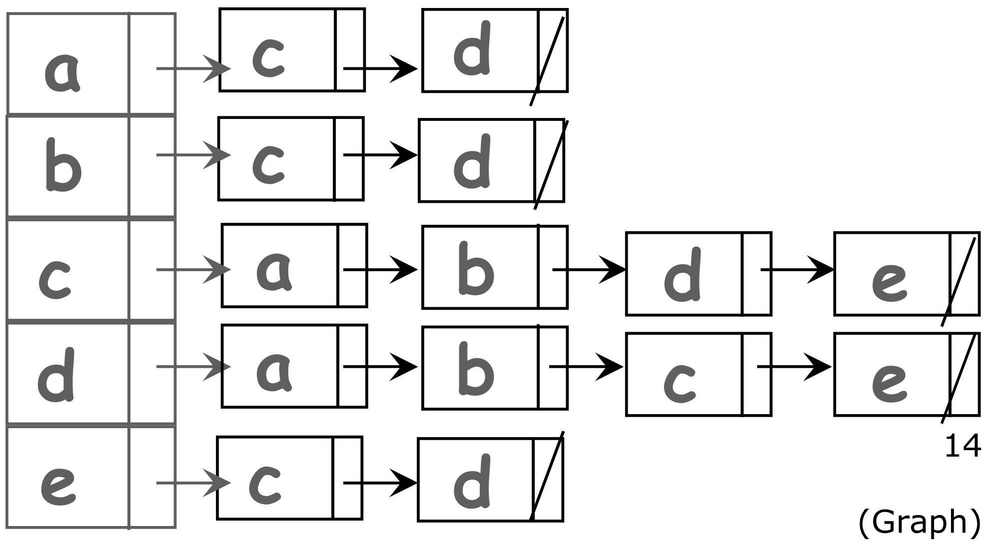
Adjacency matrix M for a simple **undirected** graph with n vertices is an **nxn** matrix

- M(i, j) = 1 if vertex i and vertex j are adjacent
- M(i, j) = 0 otherwise

Adjacency list: each vertex has a list of vertices to which it is adjacent



$$
\begin{array}{c|ccccc}
 & a & b & c & d & e \\
\hline
a & 0 & 0 & 1 & 1 & 0 \\
b & 0 & 0 & 1 & 1 & 0 \\
c & 1 & 1 & 0 & 1 & 1 \\
d & 1 & 1 & 1 & 0 & 1 \\
e & 0 & 0 & 1 & 1 & 0
\end{array}
$$



Representation (of undirected graphs)

An undirected graph can be represented by **adjacency matrix**, **adjacency list**, **incidence matrix** or **incidence list**.

Adjacency matrix and adjacency list record the relationship between **vertex adjacency**, i.e., a vertex is adjacent to which other vertices

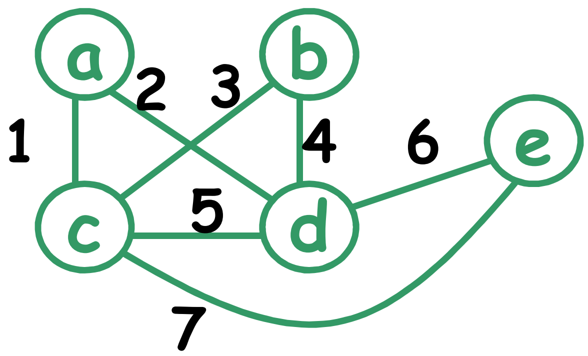
Incidence matrix and incidence list record the relationship between **edge incidence**, i.e., an edge is incident with which two vertices

Incidence matrix / list

Incidence matrix M for a simple **undirected** graph with n vertices and m edges is an **mxn** matrix

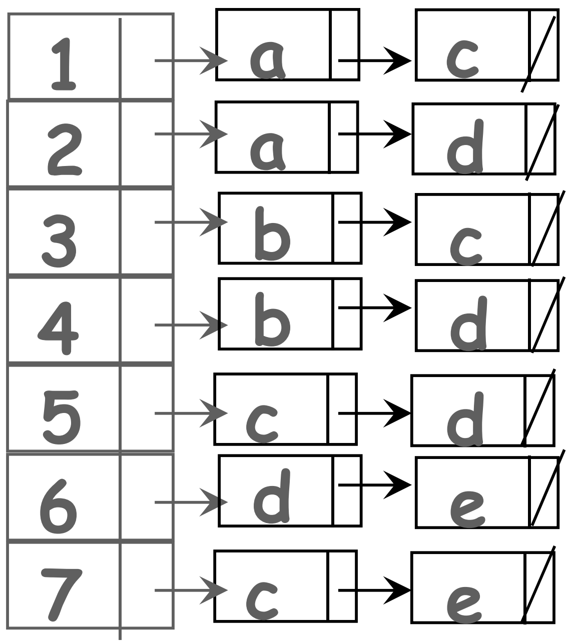
- M(i, j) = 1 if edge i and vertex j are incidence
- M(i, j) = 0 otherwise

Incidence list: each edge has a list of vertices to which it is incident with



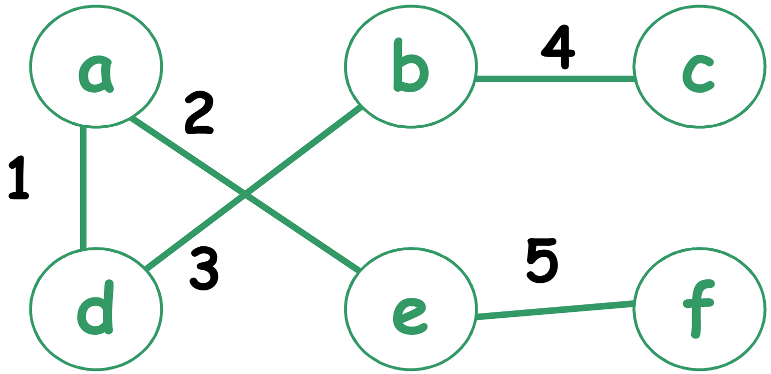
labels of edge are edge number

	a	b	c	d	e
1	1	0	1	0	0
2	1	0	0	1	0
3	0	1	1	0	0
4	0	1	0	1	0
5	0	0	1	1	0
6	0	0	0	1	1
7	0	0	1	0	1



Exercise

Give the adjacency matrix and incidence matrix of the following graph



labels of edge are edge number

	a	b	c	d	e	f
a						
b						
c						
d						
e						
f						

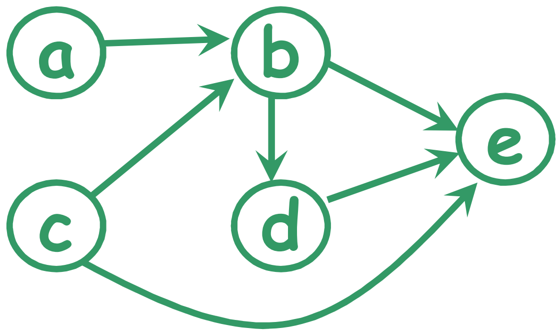
	a	b	c	d	e	f
1						
2						
3						
4						
5						

Directed graph ...

Directed graph

Given a directed graph G, a vertex ***a*** is said to be **connected to** a vertex ***b*** if there is a path from ***a*** to ***b***.

E.g., G represents the routes provided by a certain airline. That means, a vertex represents a city and an edge represents a flight from a city to another city. Then we may ask question like: Can we fly from one city to another?



E = { (a,b), (b,d), (b,e), (c,b), (c,e), (d,e) }

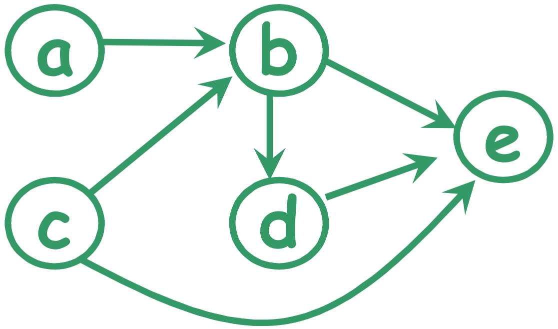
N.B. (a,b) is in E, but (b,a) is NOT

Reminder: A directed graph G=(V,E) consists of a set of vertices V and a set of edges E. Each edge is an ordered pair of vertices.

In/Out degree (in directed graphs)

The **in-degree** of a vertex *v* is the number of edges ***leading to*** the vertex *v*.

The **out-degree** of a vertex *v* is the number of edges ***leading away*** from the vertex *v*.



v	in-deg(v)	out-deg(v)
a	0	1
b	2	2
c	0	2
d	1	1
e	3	0
sum:	6	6

Always equal?

Representation (of directed graphs)

Similar to undirected graph, a directed graph can be represented by **adjacency matrix**, **adjacency list**, **incidence matrix** or **incidence list**.

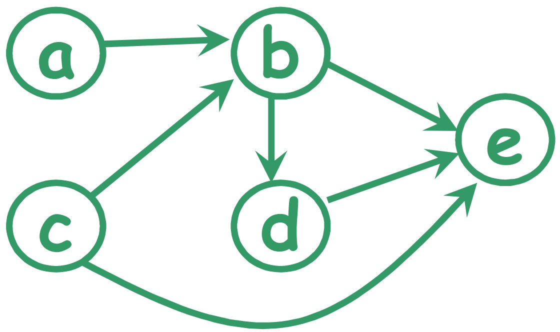
Adjacency matrix / list

Adjacency matrix M for a **directed** graph with n vertices is an **nxn** matrix

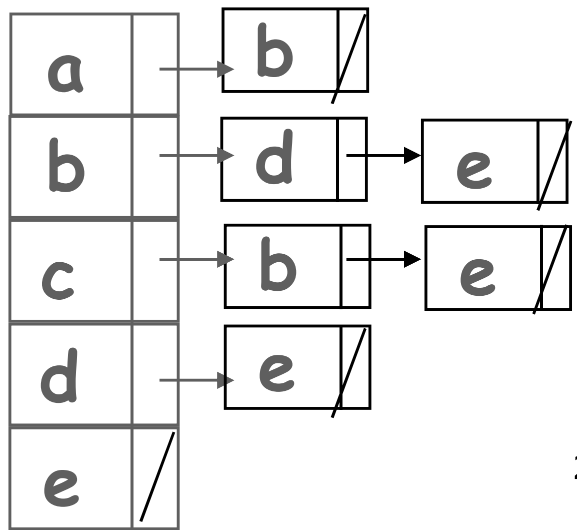
- M(i, j) = 1 if (i,j) is an edge
- M(i, j) = 0 otherwise

Adjacency list:

- each vertex *u* has a list of vertices pointed to by an edge leading away from *u*



	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	1	1
c	0	1	0	0	1
d	0	0	0	0	1
e	0	0	0	0	0

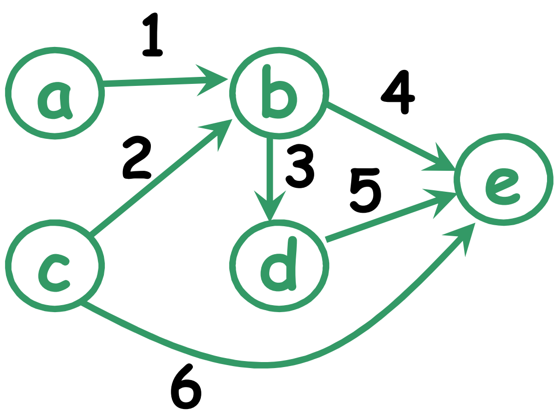


Incidence matrix / list

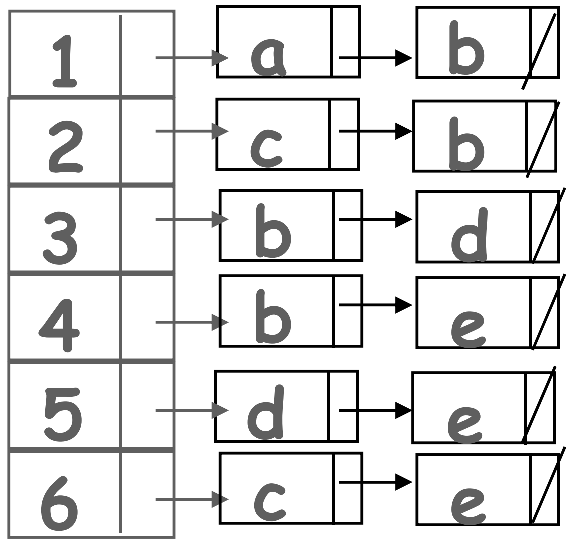
Incidence matrix M for a **directed** graph with n vertices and m edges is an **mxn** matrix

- M(i, j) = 1 if edge i is leading away from vertex j
- M(i, j) = -1 if edge i is leading to vertex j

Incidence list: each edge has a list of two vertices (leading away is 1st and leading to is 2nd)

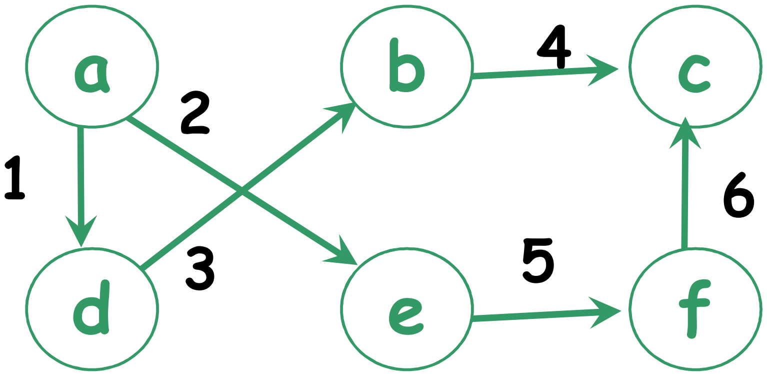


	a	b	c	d	e
1	1	-1	0	0	0
2	0	-1	1	0	0
3	0	1	0	-1	0
4	0	1	0	0	-1
5	0	0	0	1	-1
6	0	0	1	0	-1



Exercise

Give the adjacency matrix and incidence matrix of the following graph



labels of edge are edge number

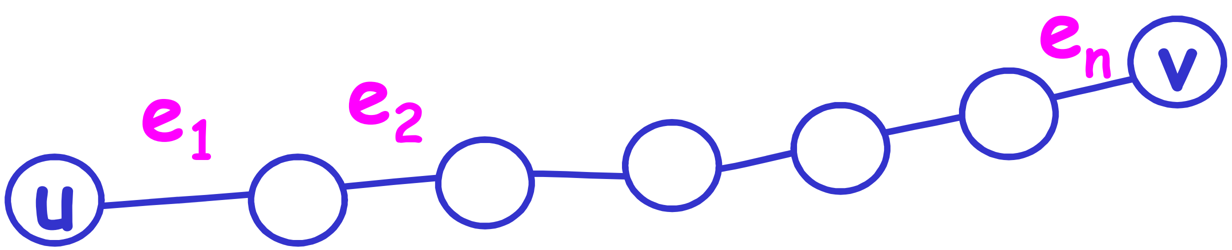
	a	b	c	d	e	f
a						
b						
c						
d						
e						
f						

	a	b	c	d	e	f
1						
2						
3						
4						
5						
6						

Euler circuit ...

Paths, circuits (in undirected graphs)

- In an undirected graph, a **path** from a vertex ***u*** to a vertex ***v*** is a sequence of edges $e_1 = \{u, x_1\}$, $e_2 = \{x_1, x_2\}$, … $e_n = \{x_{n-1}, v\}$, where $n \geq 1$.
- The **length** of this path is **n**.
- Note that a path from ***u*** to ***v*** implies a path from ***v*** to ***u***.
- If u = v, this path is called a **circuit** (cycle).

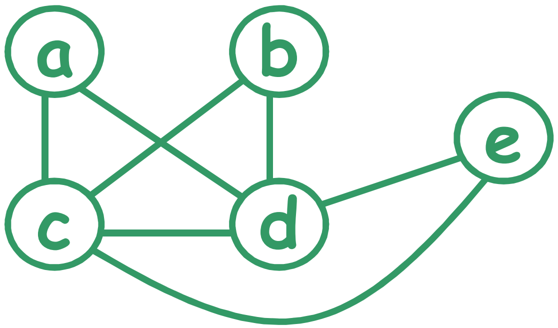


Euler circuit

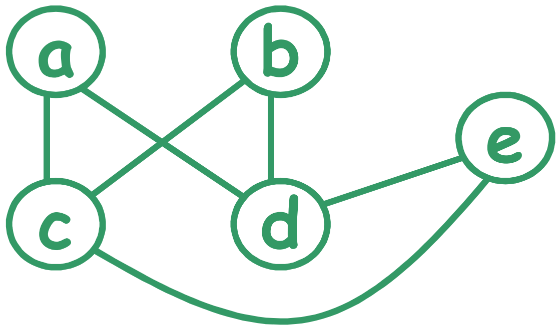
A simple circuit visits an edge **at most** once.

An **Euler** circuit in a graph G is a circuit visiting every edge of G **exactly** once.
(NB. A vertex can be repeated.)

Does every graph has an Euler circuit ?

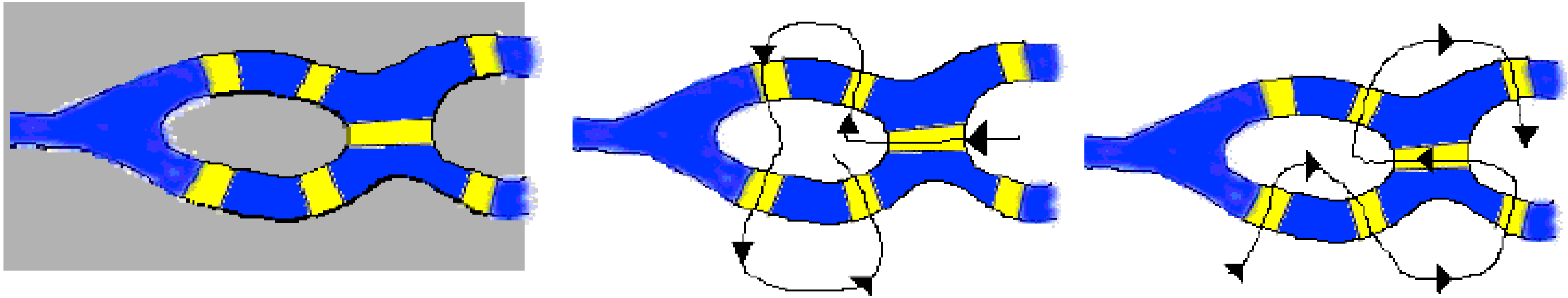


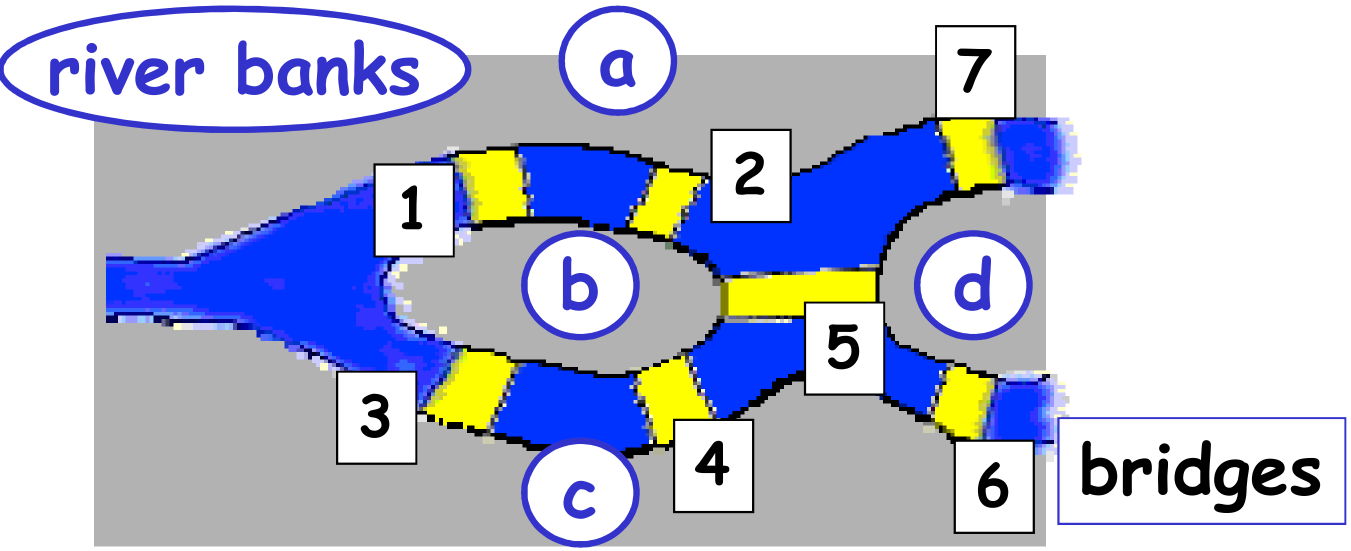
a c b d e c d a

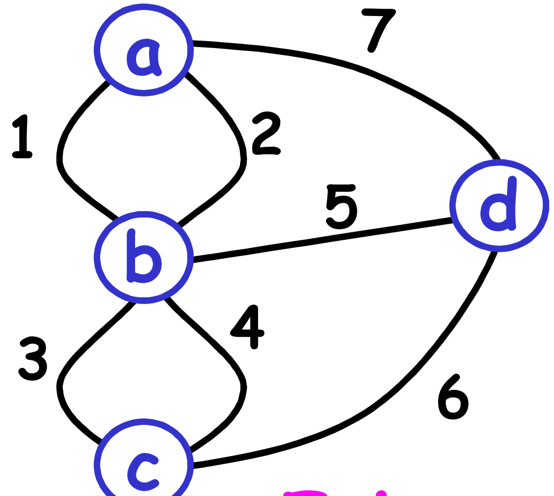


no Euler circuit

History: In Konigsberg, Germany, a river ran through the city and seven bridges were built. The people wondered whether or not one could go around the city in a way that would involve crossing each bridge exactly once.







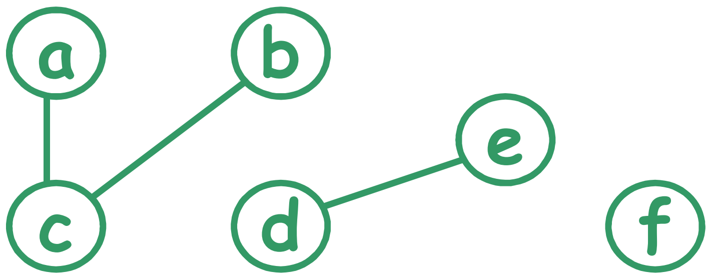
no Euler circuit

How to determine whether there is an Euler circuit in a graph?

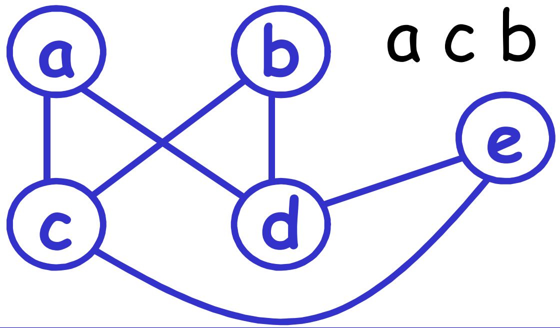
A trivial condition

An undirected graph G is said to be **connected** if there is a path between ***every pair*** of vertices.

If G is **not** connected, there is no single circuit to visit all edges or vertices.



Even if the graph is connected, there may be no Euler circuit either.

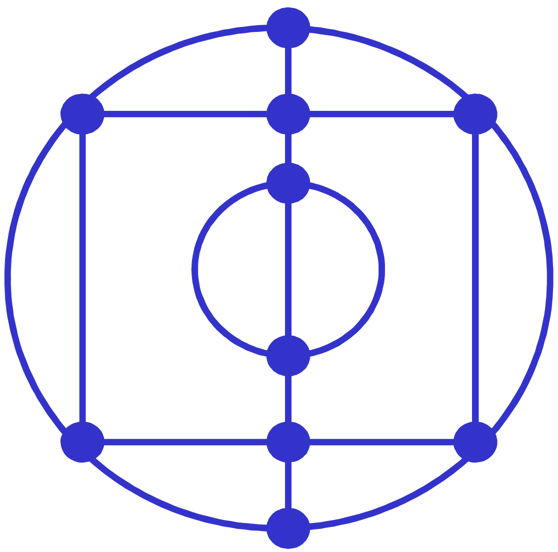


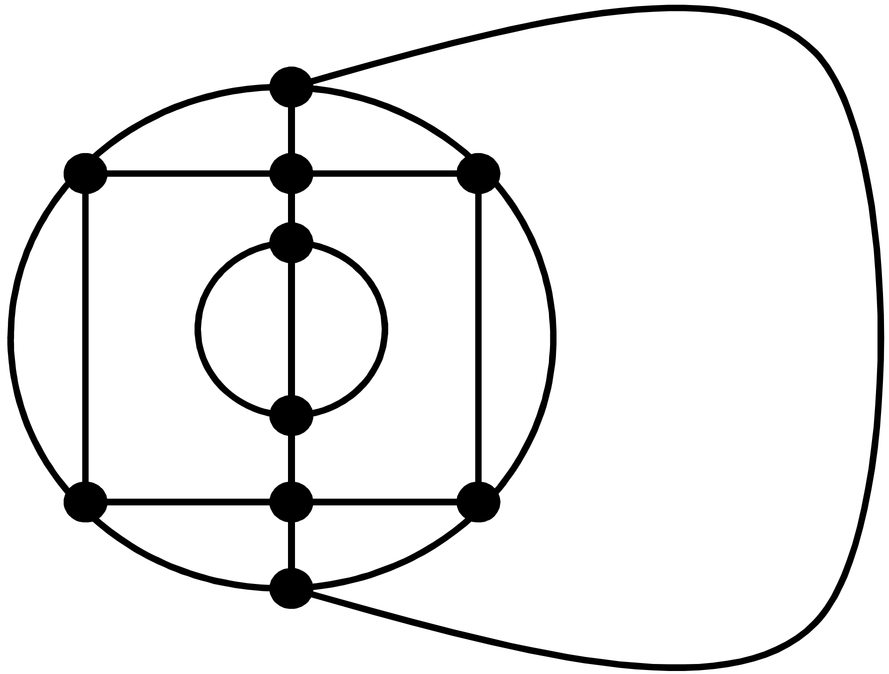
a c b d e c b d a

Necessary and sufficient condition

Let G be a connected graph.

Lemma: G contains an Euler circuit if and only if degree of every vertex is **even**.



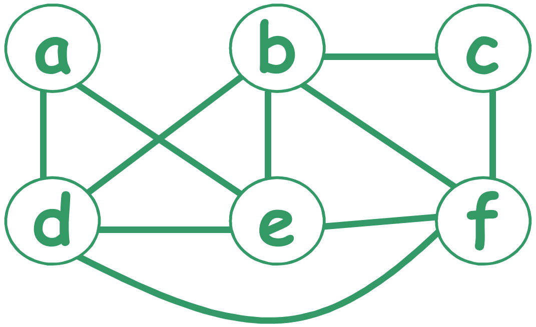


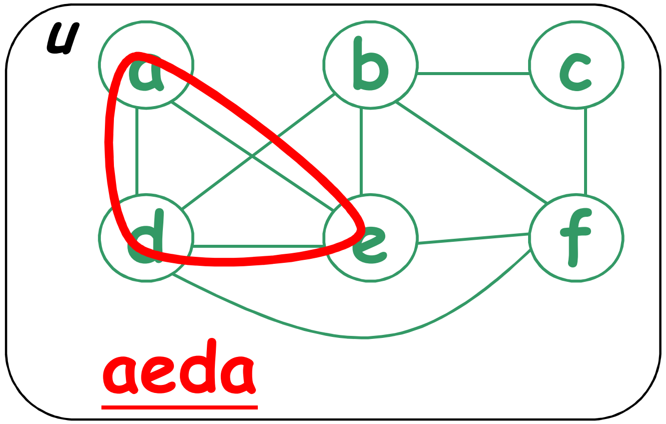
Necessary and sufficient condition

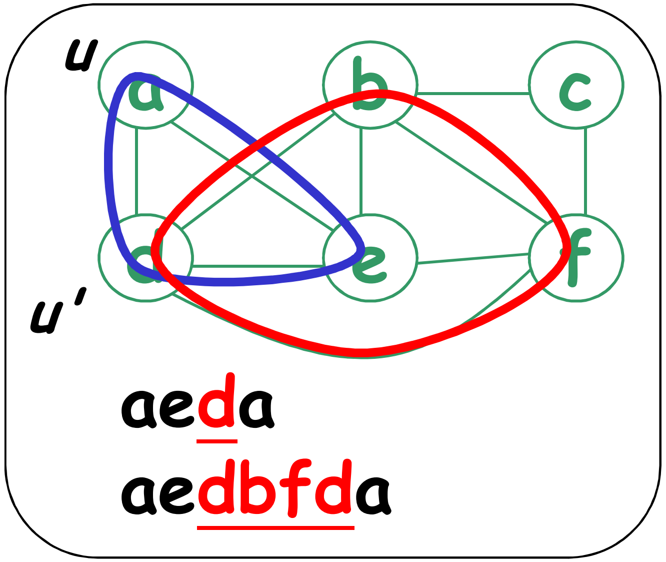
Let G be a connected graph.

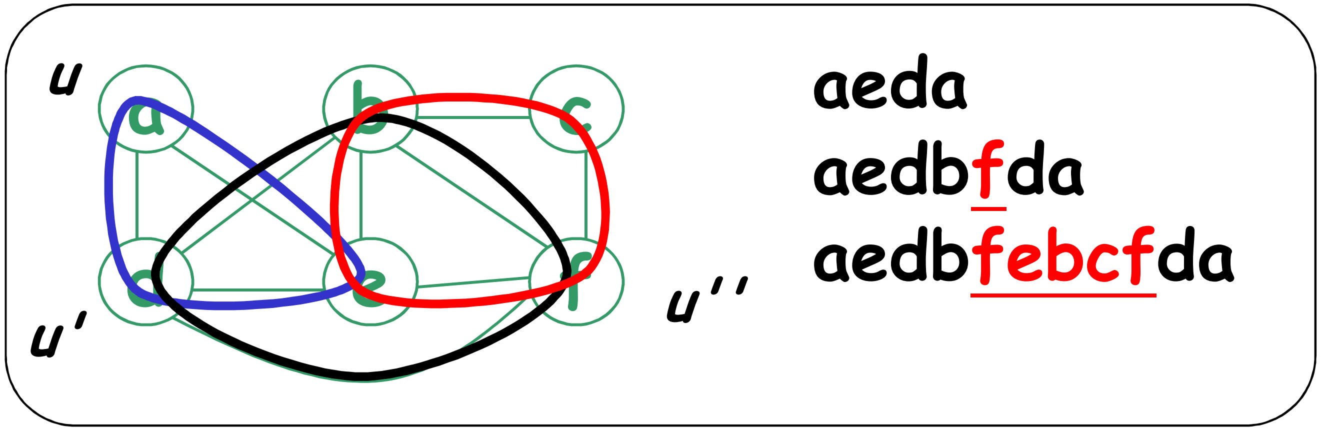
How to find it?

Lemma: G contains an Euler circuit if and only if degree of every vertex is **even**.









Hamiltonian circuit

Let G be an undirected graph.

A **Hamiltonian circuit** is a circuit containing **every vertex** of G **exactly once**.

Note that a Hamiltonian circuit may NOT visit all edges.

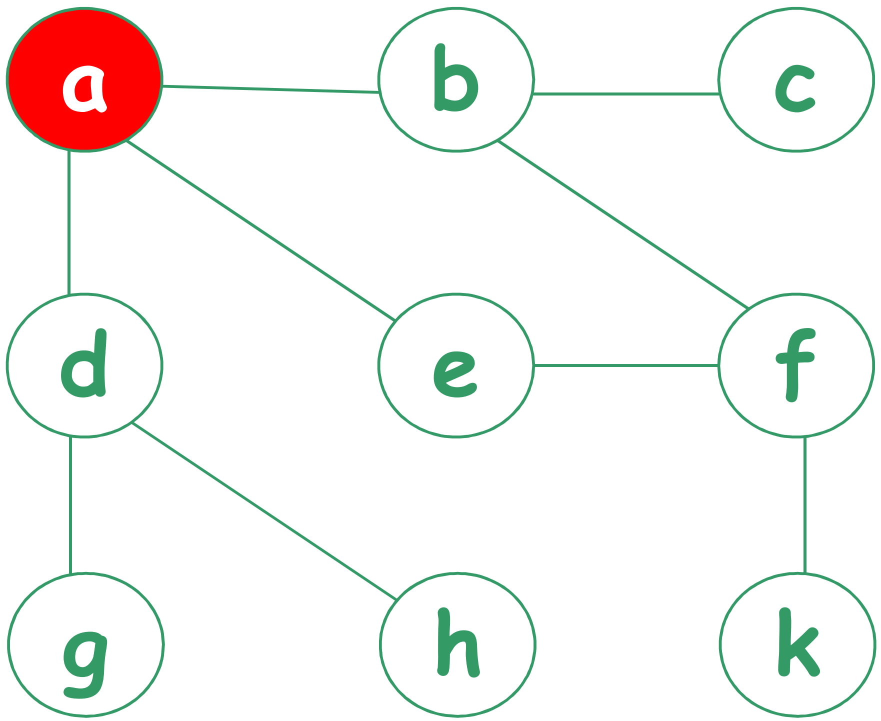
Unlike the case of Euler circuits, determining whether a graph contains a Hamiltonian circuit is a very ***difficult*** problem. (NP-hard)

Breadth First Search BFS …

Breadth First Search (BFS)

All vertices at distance k from s are explored before any vertices at distance k+1.

The source is a.



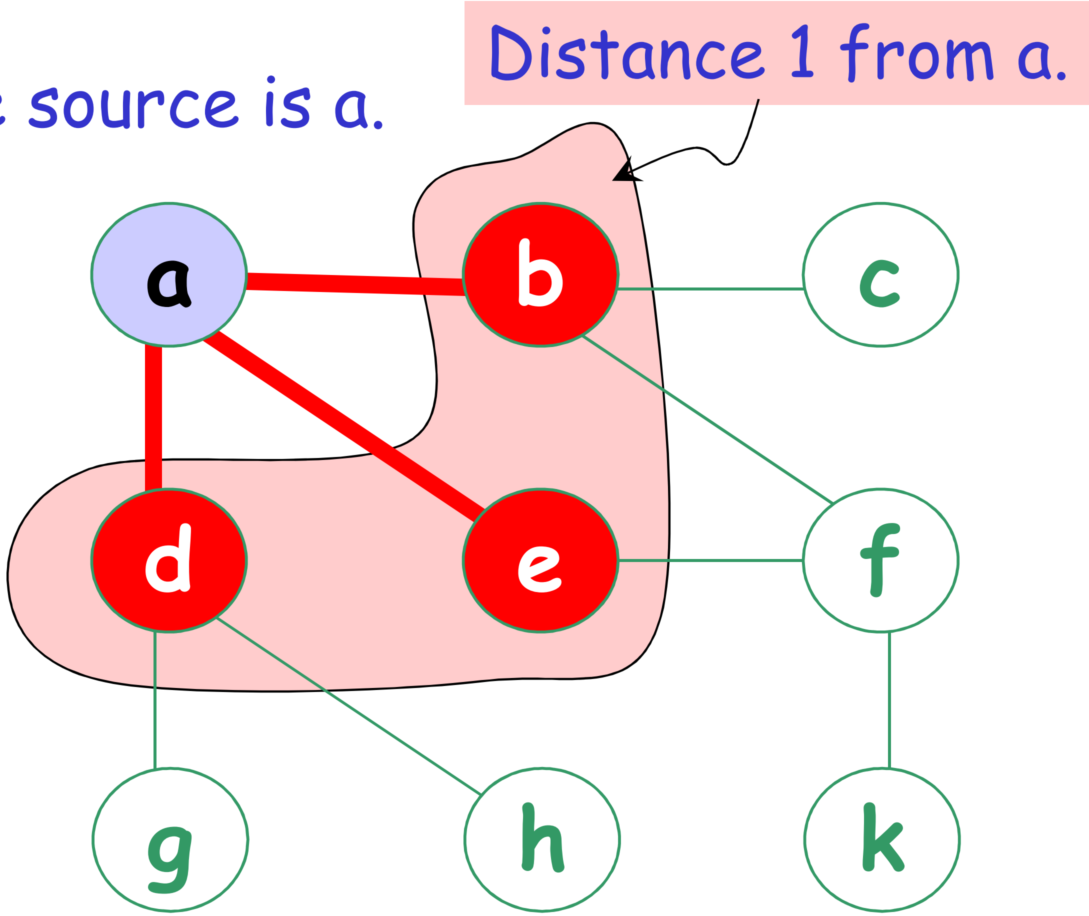
Order of exploration

a,

Breadth First Search (BFS)

All vertices at distance k from s are explored before any vertices at distance k+1.

The source is a.

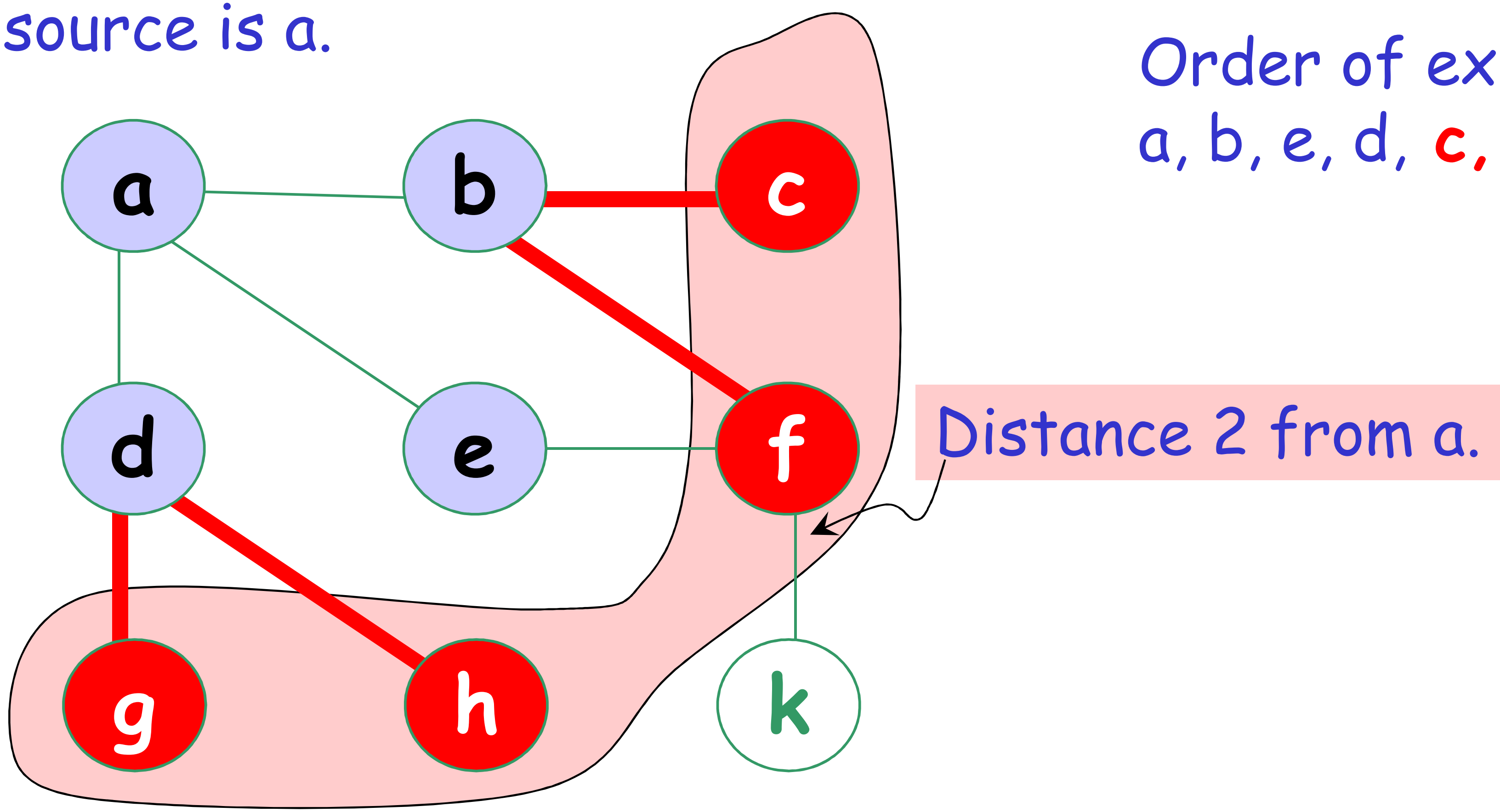


Order of exploration
a, b, e, d

Breadth First Search (BFS)

All vertices at distance k from s are explored before any vertices at distance k+1.

The source is a.



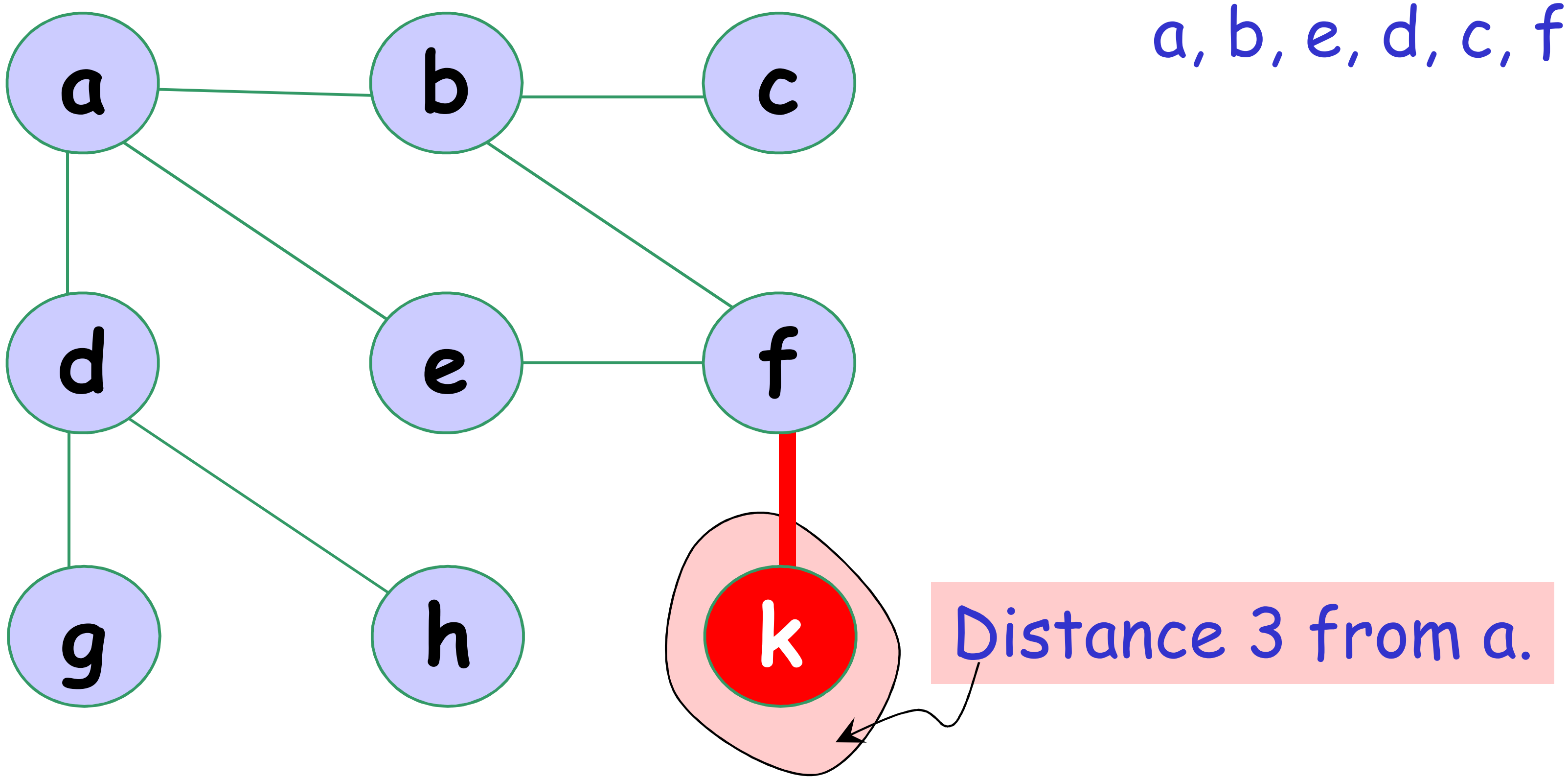
Order of exploration
a, b, e, d, c, f, h, g

Breadth First Search (BFS)

All vertices at distance k from s are explored before any vertices at distance k+1.

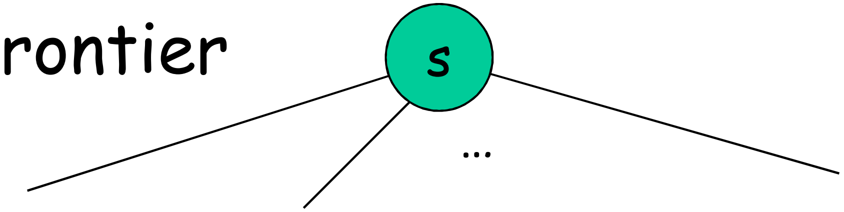
The source is a.

Order of exploration
a, b, e, d, c, f, h, g, k



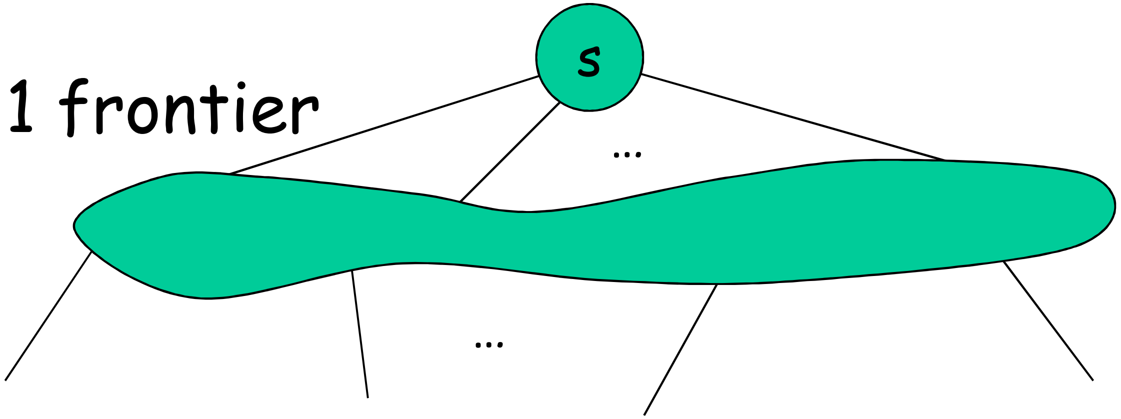
In general (BFS)

Explore dist 0 frontier



distance 0

In general (BFS)

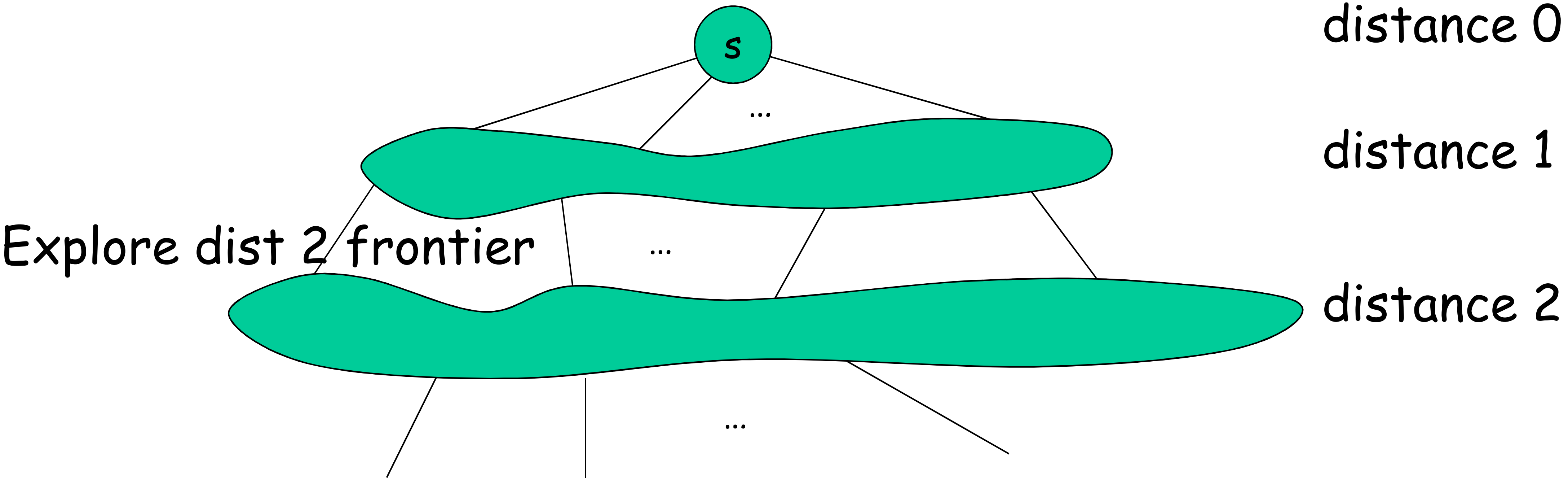


Explore dist 1 frontier

distance 0

distance 1

In general (BFS)



Breadth First Search (BFS)

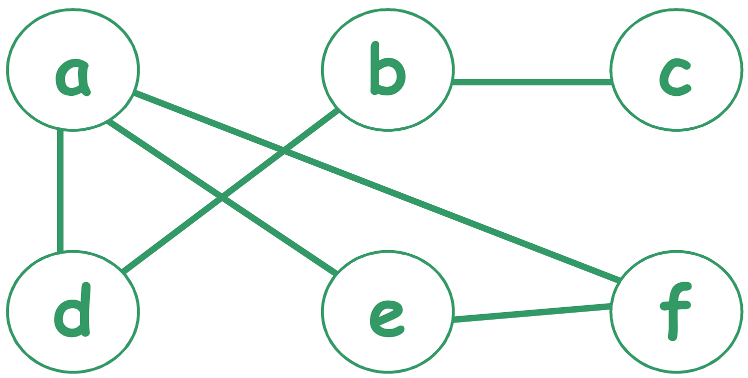
A simple algorithm for searching a graph.

Given G=(V, E), and a distinguished source vertex ***s***, BFS systematically explores the edges of G such that

- all vertices at ***distance k*** from s are explored ***before*** any vertices at ***distance k+1***.

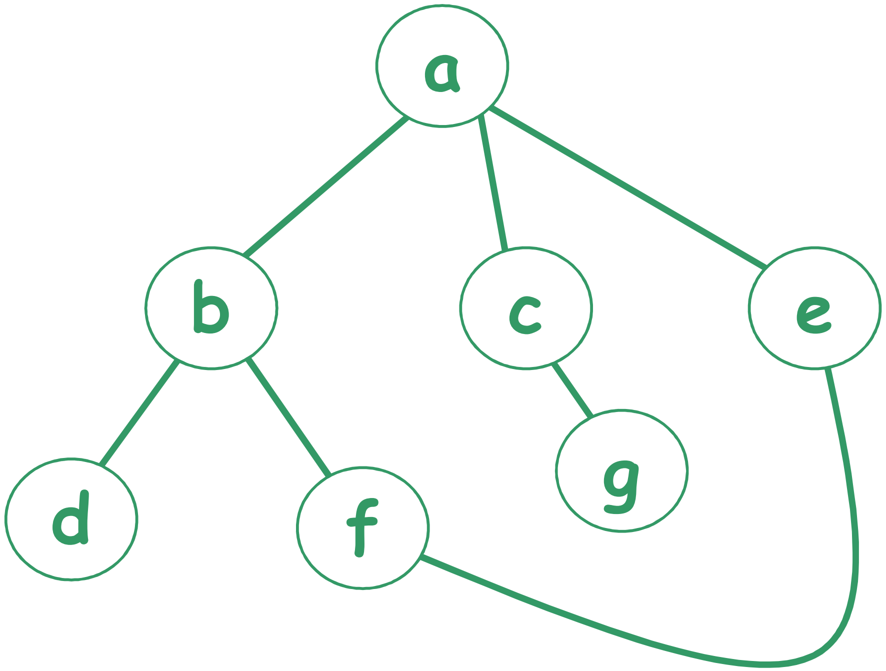
Exercise – BFS

Apply **BFS** to the following graph starting from vertex **a** and list the order of exploration



Exercise (2) – BFS

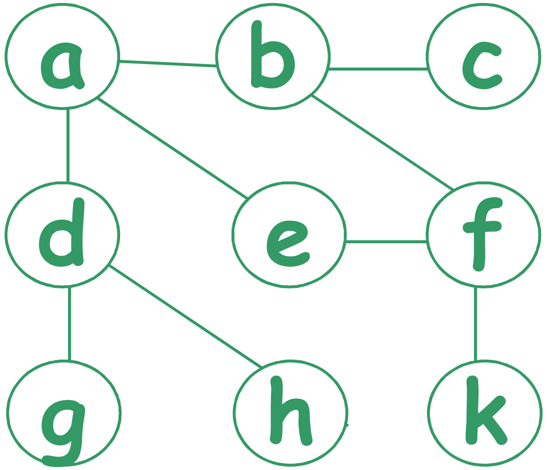
Apply **BFS** to the following graph starting from vertex **a** and list the order of exploration



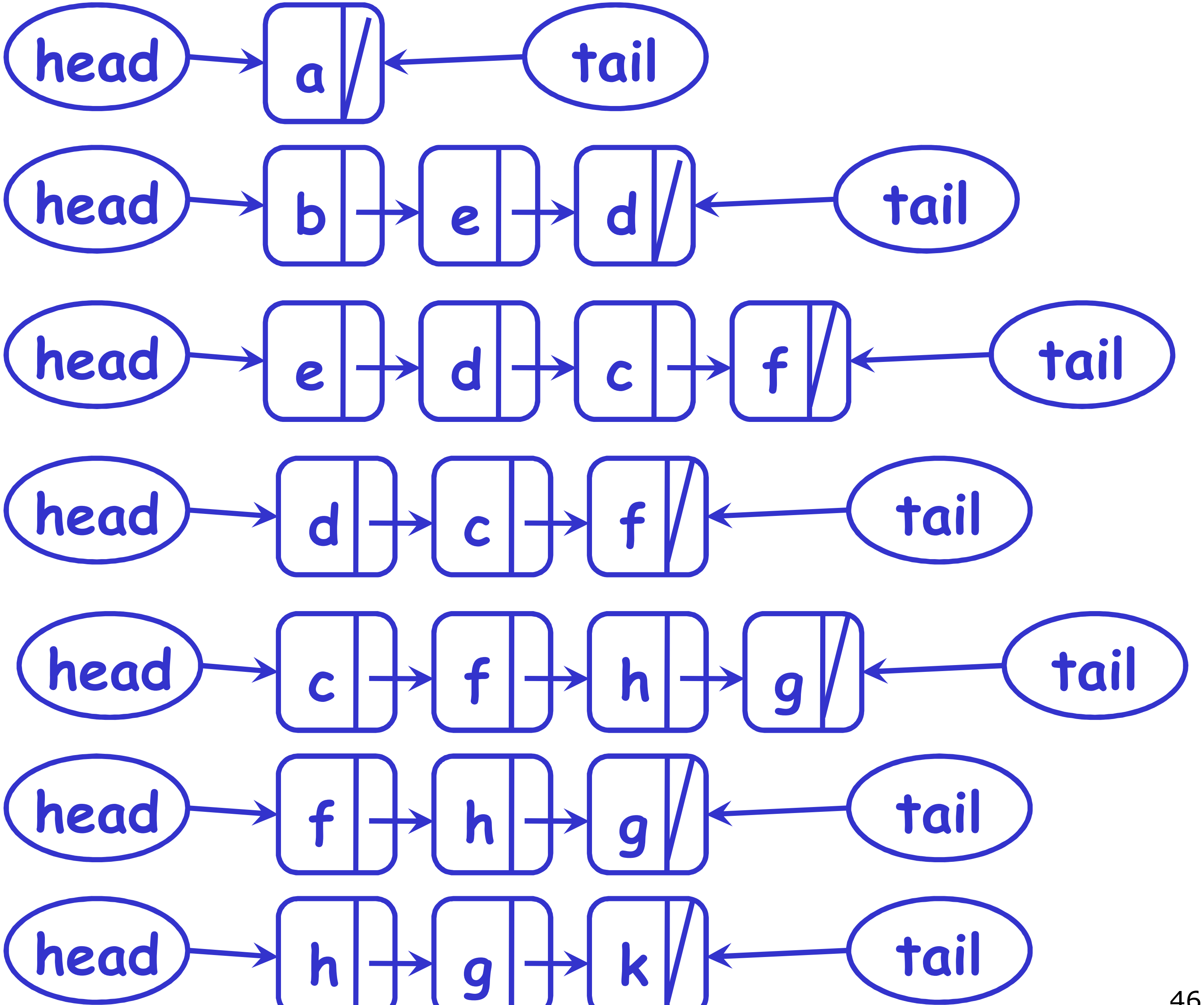
BFS – Pseudo code

```
unmark all vertices
choose some starting vertex s
mark s and insert s into tail of list L
while L is nonempty do
begin
   remove a vertex v from front of L
   visit v
   for each unmarked neighbor w of v do
        mark w and insert w into tail of list L
end
```

BFS using linked list



a, b, e, d, c, f, h, g, k



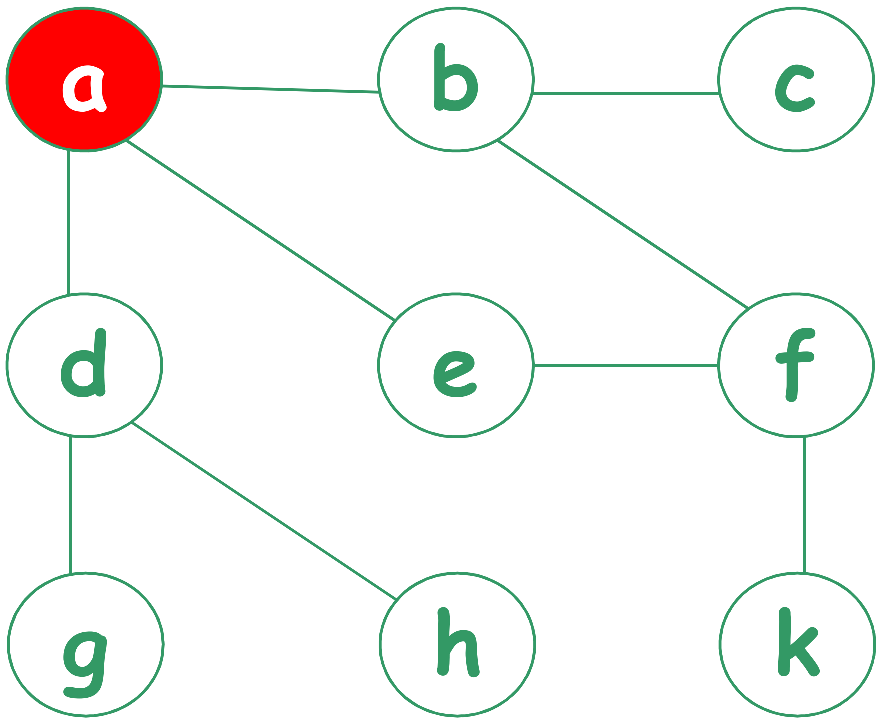
& so on …

Depth First Search DFS …

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



Order of exploration

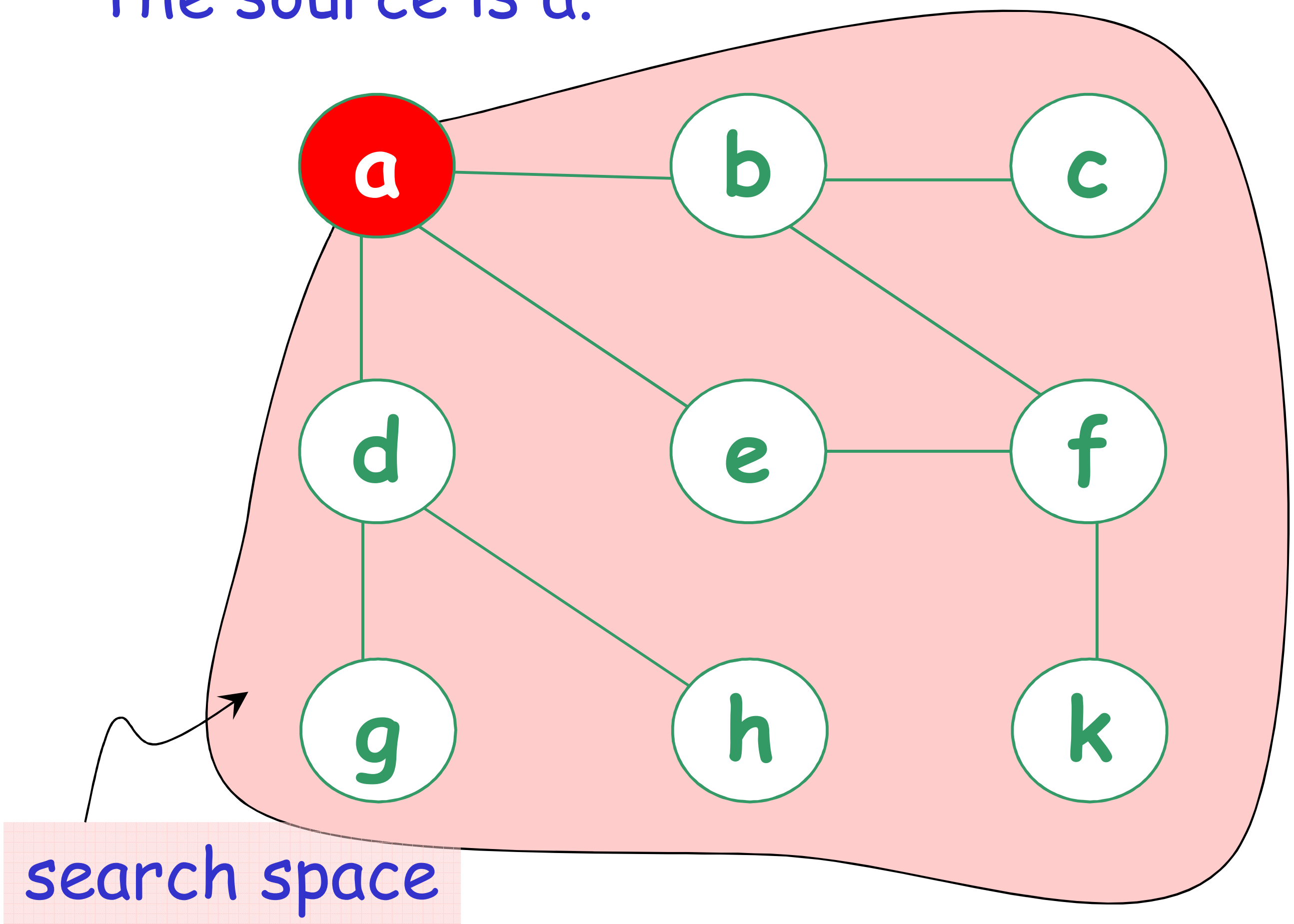
a,

DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



Order of exploration

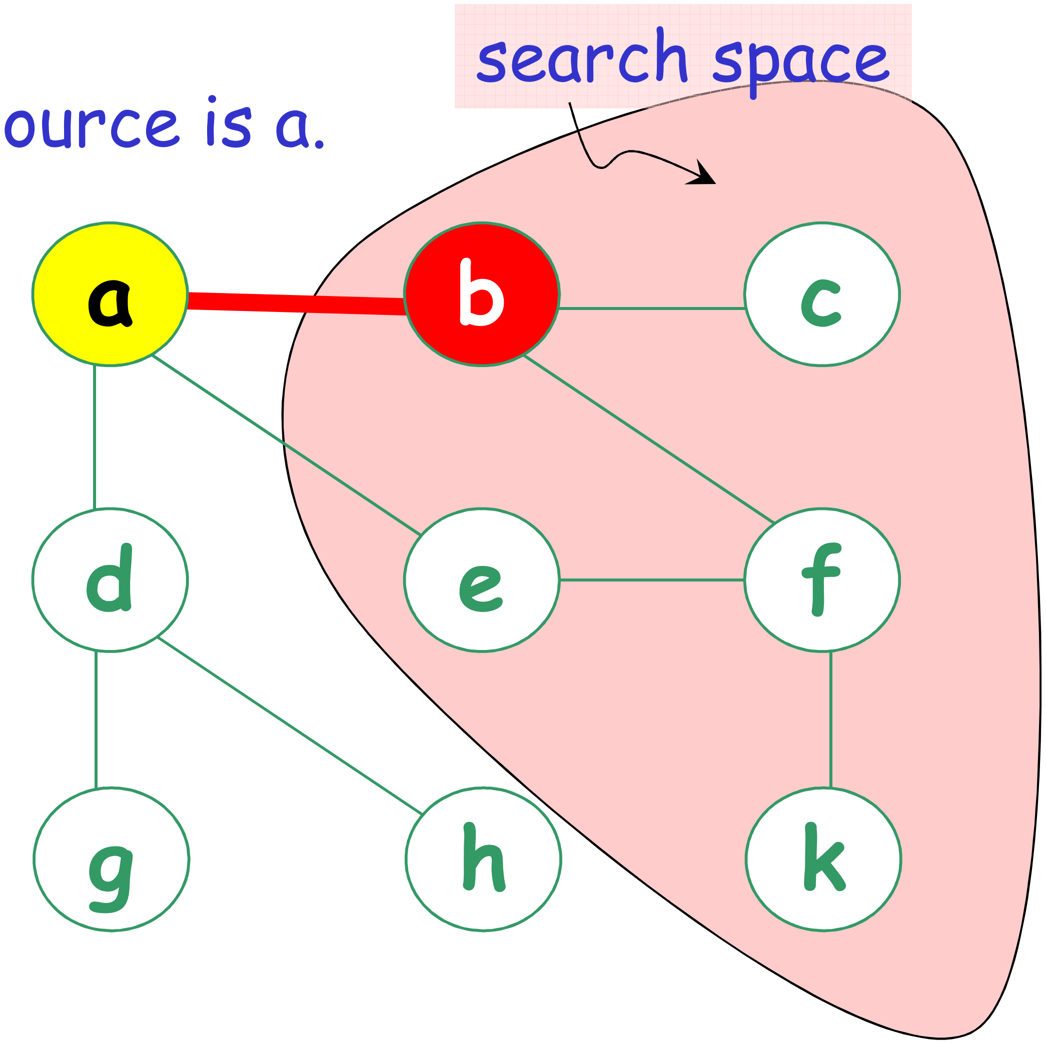
a,

DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



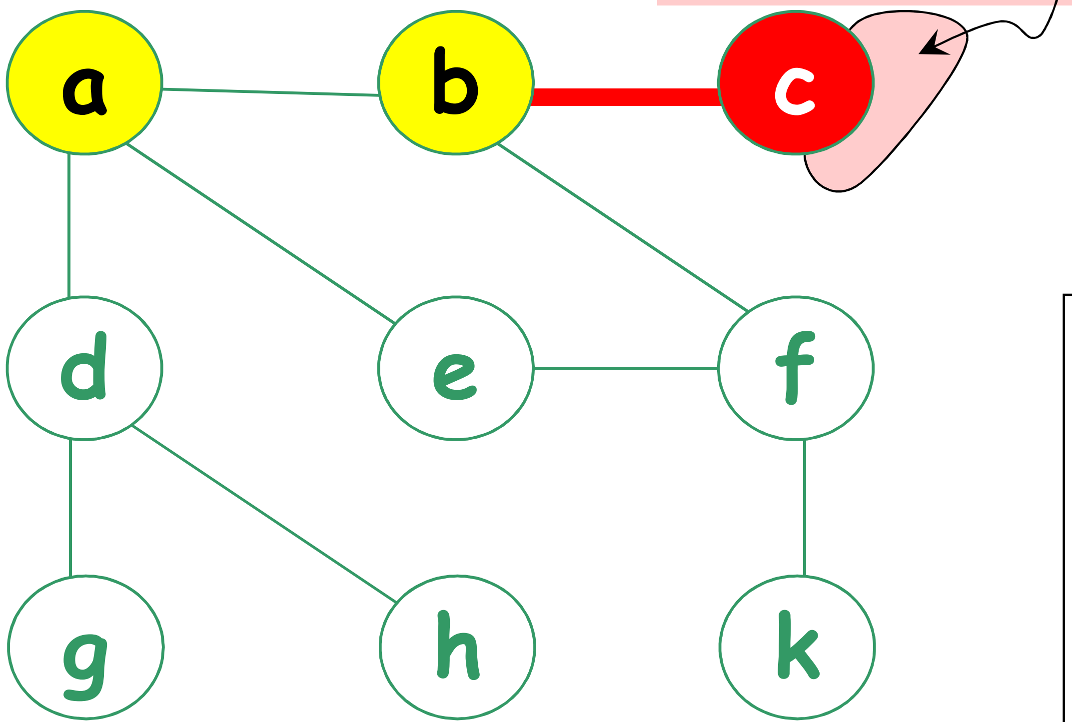
Order of exploration
a, **b**

DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



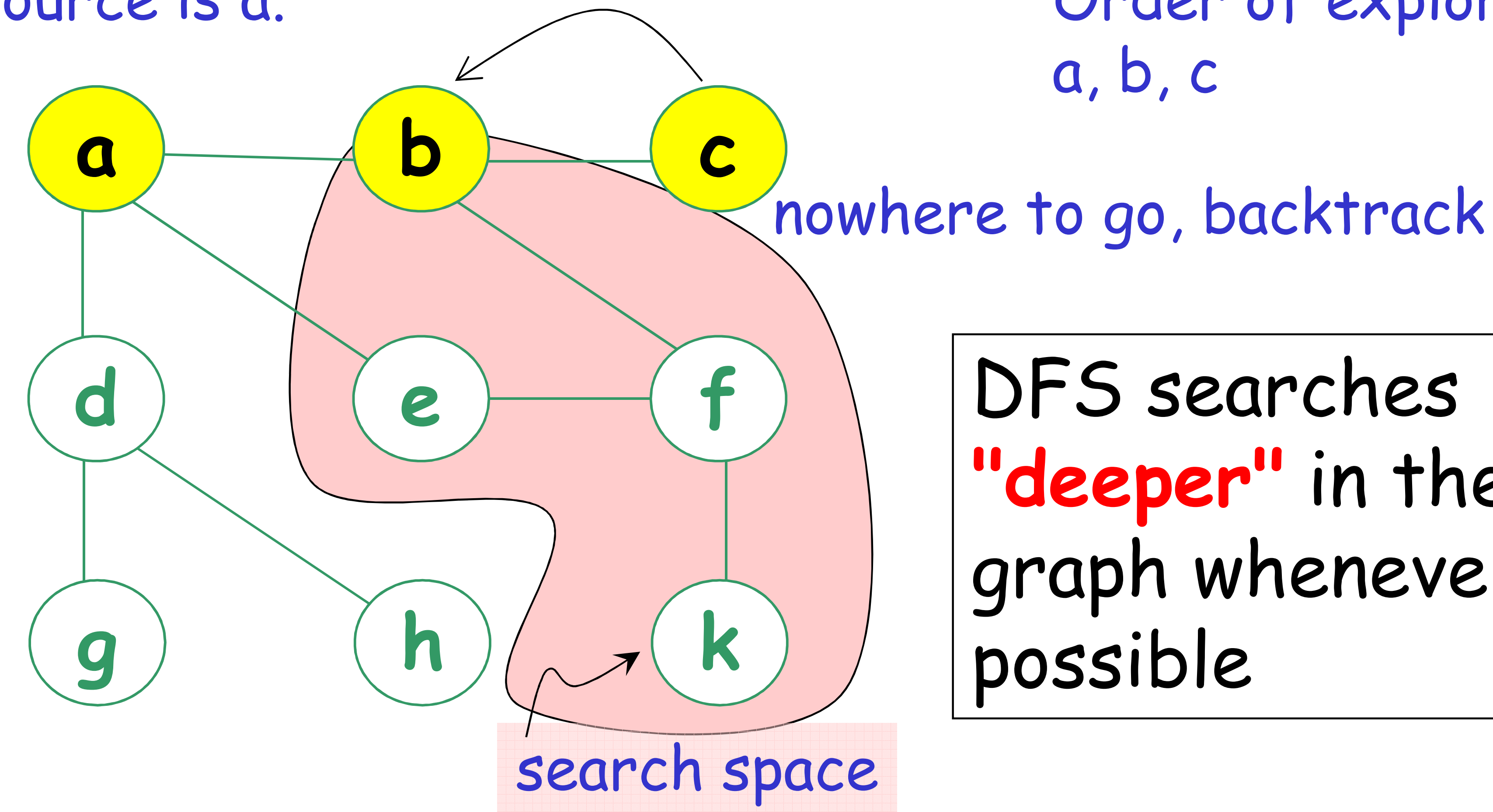
Order of exploration
a, b, **c**

DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



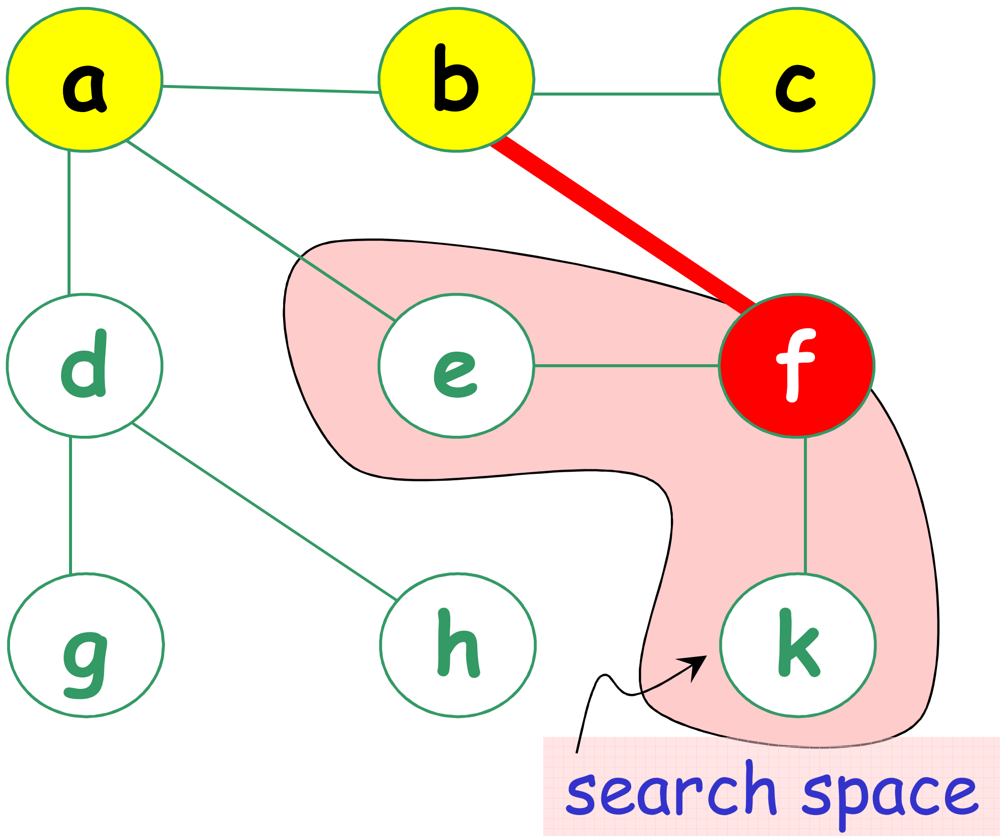
Order of exploration
a, b, c

DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



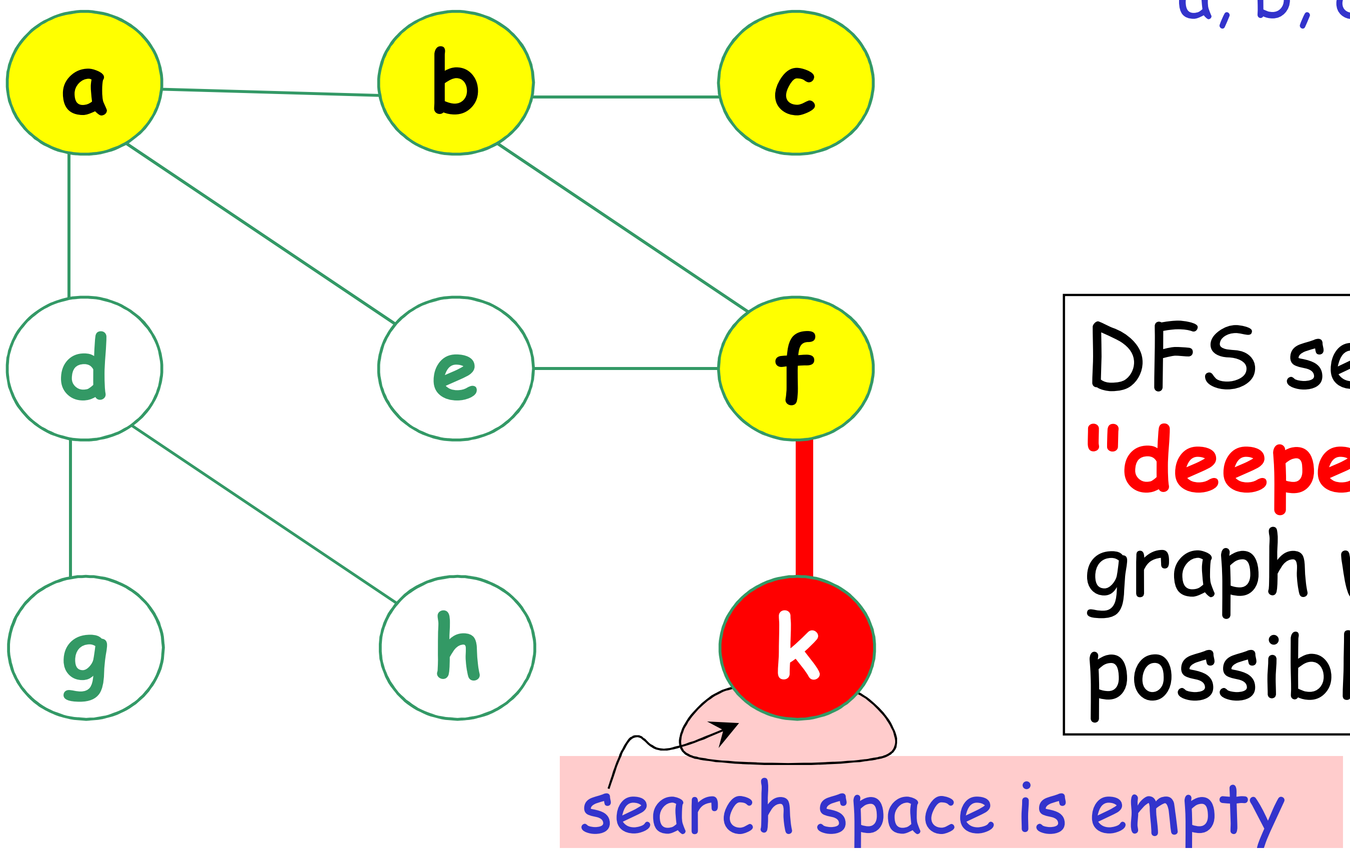
Order of exploration
a, b, c, **f**

DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



Order of exploration
a, b, c, f, k

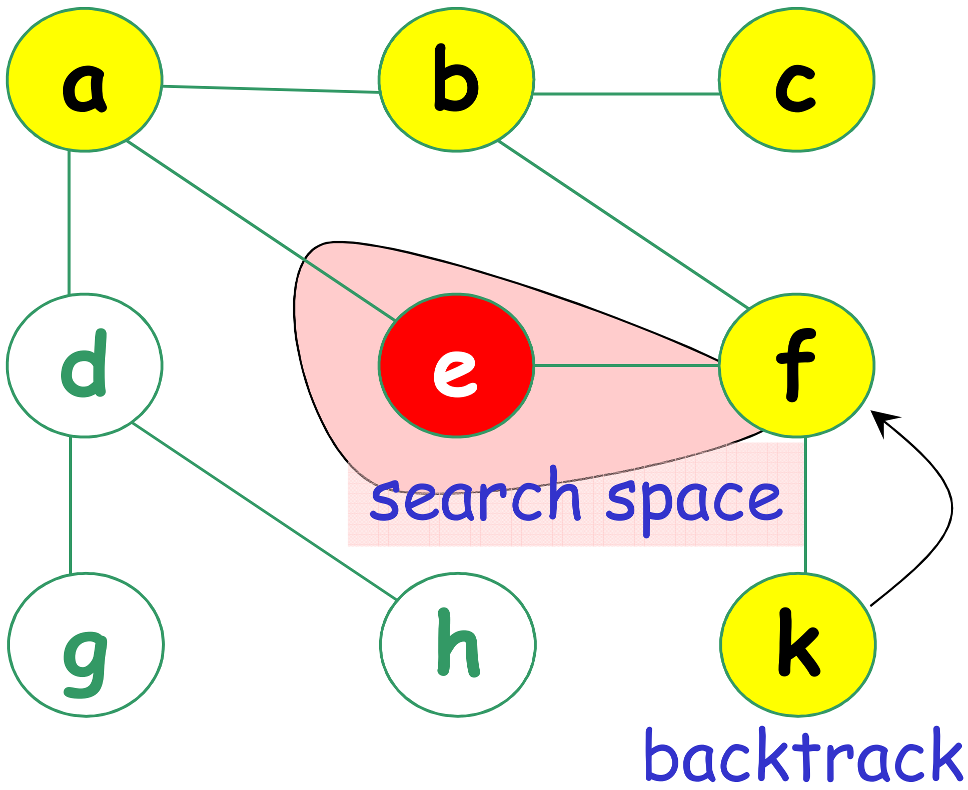
DFS searches "deeper" in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.

Order of exploration
a, b, c, f, k, **e**



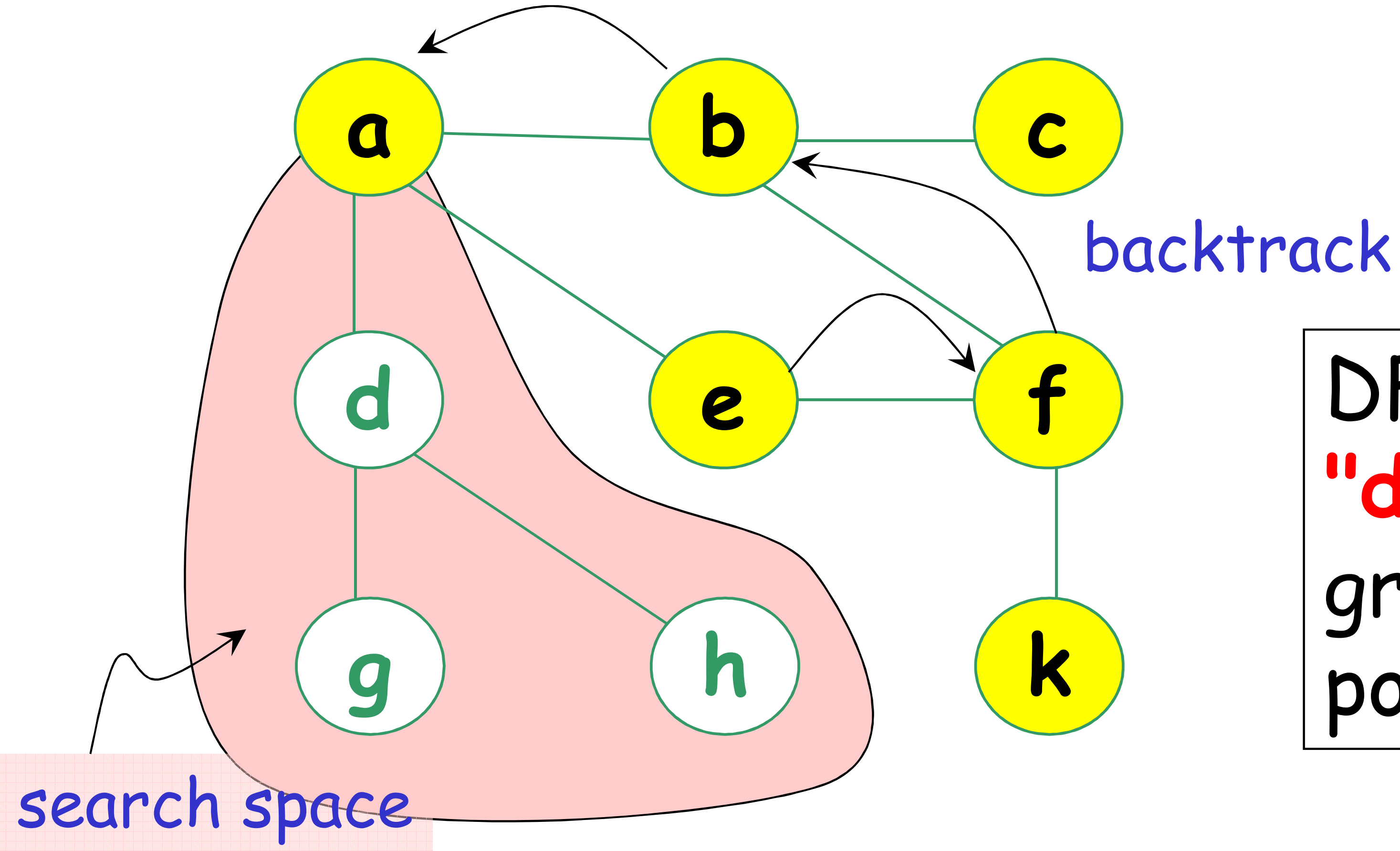
DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.

Order of exploration
a, b, c, f, k, e



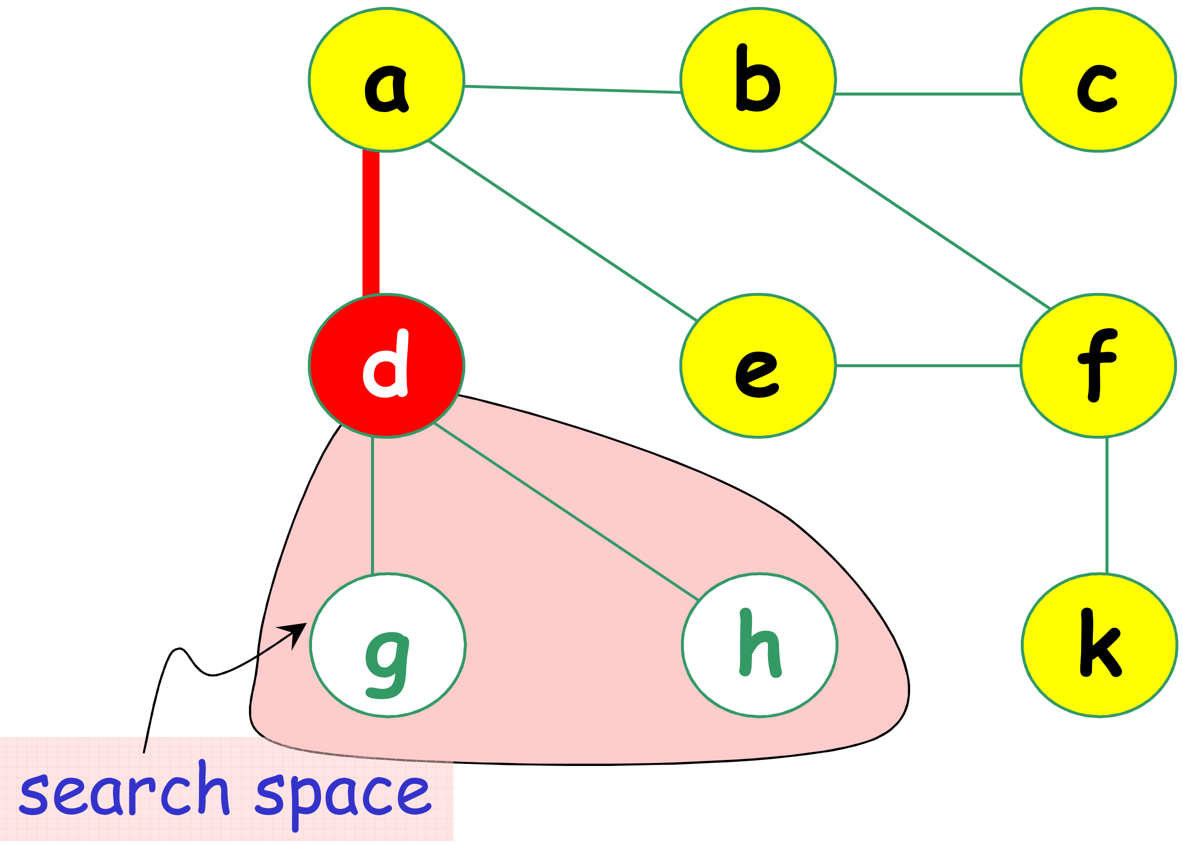
DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.

Order of exploration
a, b, c, f, k, e, d

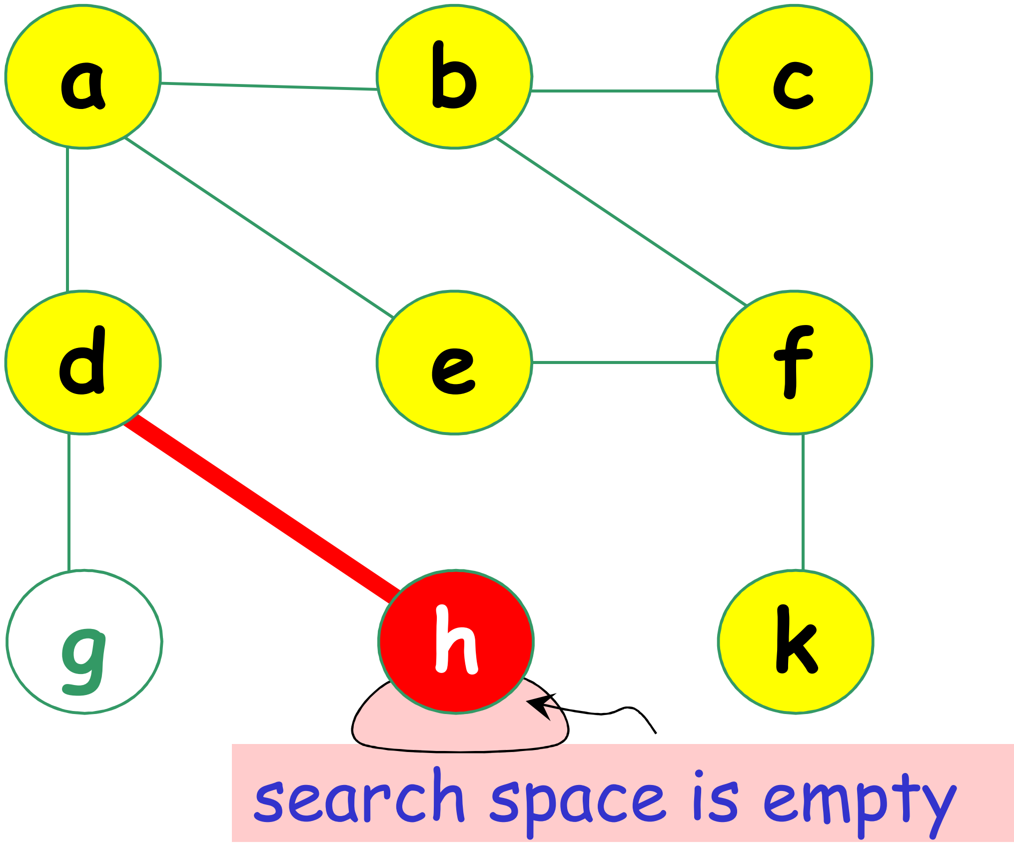


DFS searches "deeper" in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



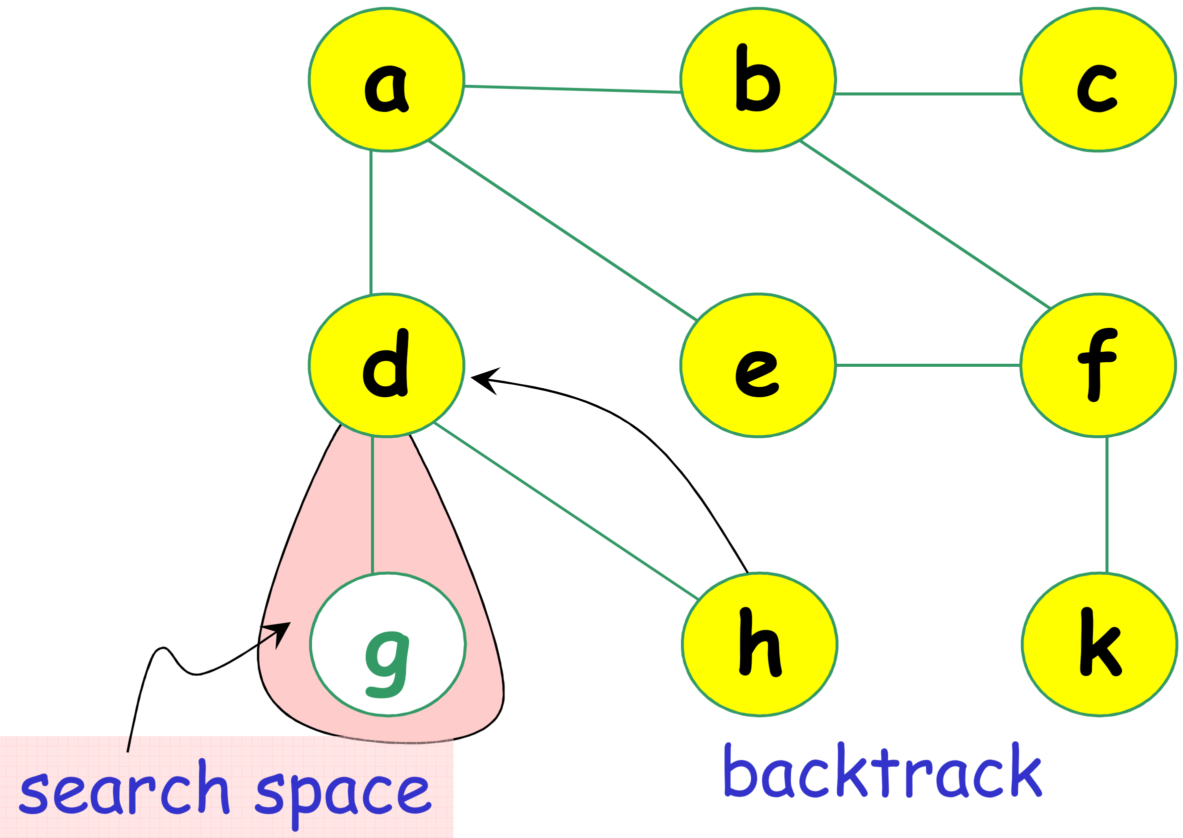
Order of exploration
a, b, c, f, k, e, d, h

DFS searches "deeper" in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.



Order of exploration
a, b, c, f, k, e, d, h

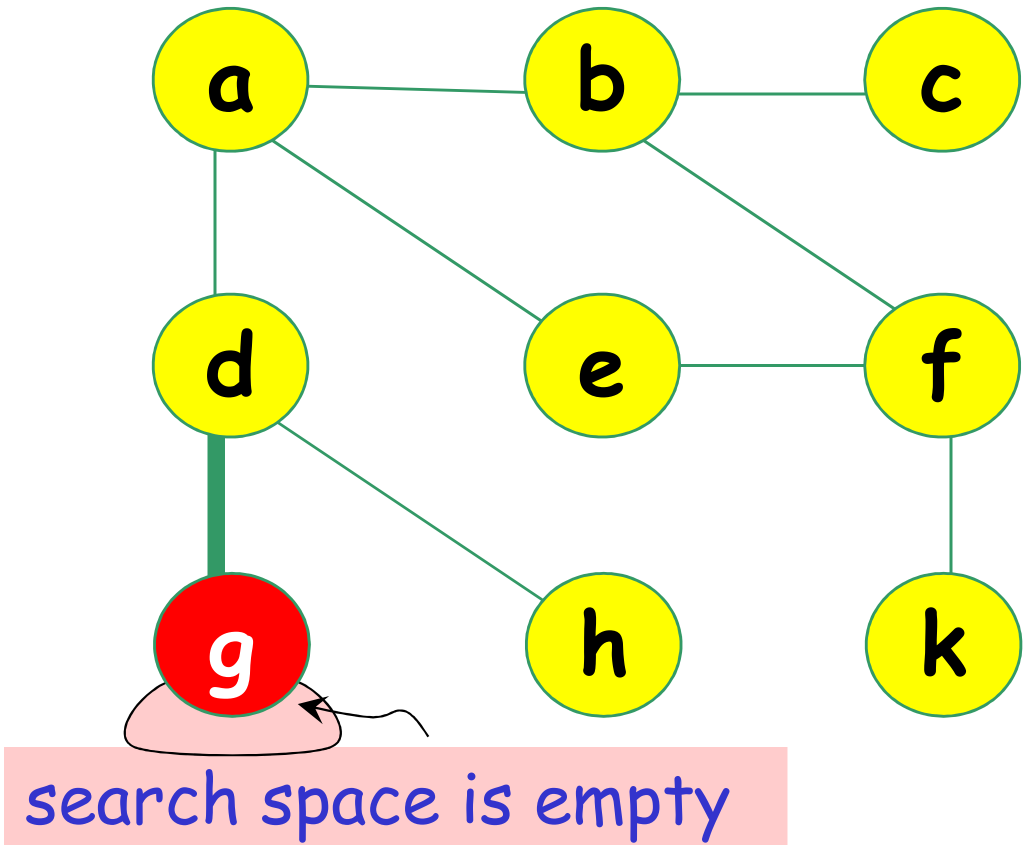
DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.

Order of exploration
a, b, c, f, k, e, d, h, **g**



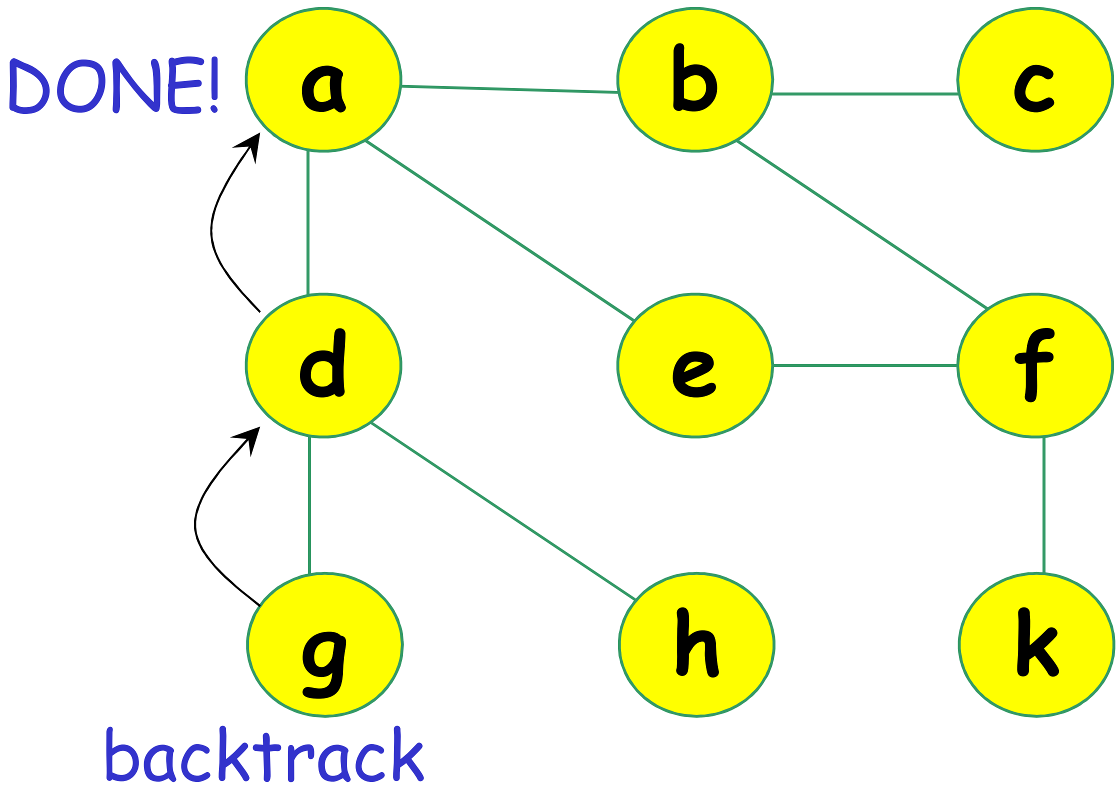
DFS searches **"deeper"** in the graph whenever possible

Depth First Search (DFS)

Edges are explored from the most recently discovered vertex, backtracks when finished

The source is a.

Order of exploration
a, b, c, f, k, e, d, h, g



DFS searches **"deeper"** in the graph whenever possible

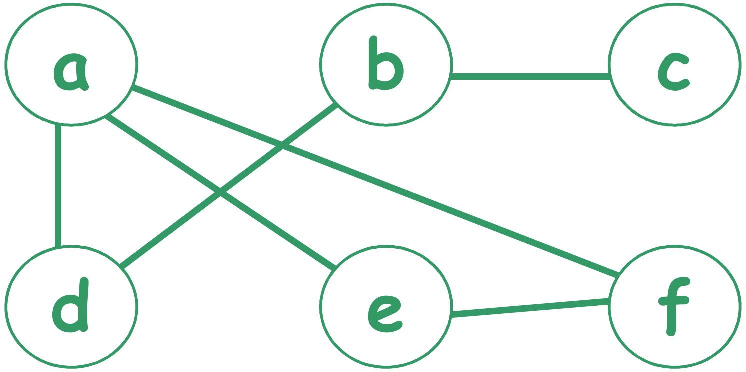
Depth First Search (DFS)

Depth-first search is another strategy for exploring a graph; it search **"deeper"** in the graph whenever possible.

- Edges are explored from the most recently discovered vertex ***v*** that still has unexplored edges leaving it.
- When all edges of ***v*** have been explored, the search ***"backtracks"*** to explore edges leaving the vertex from which ***v*** was discovered.

Exercise – DFS

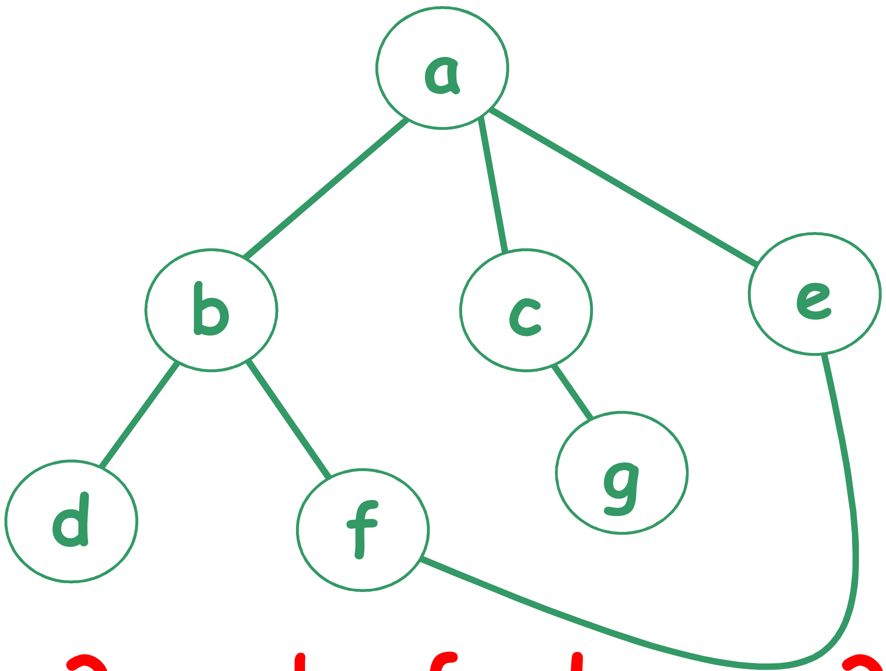
Apply **DFS** to the following graph starting from vertex **a** and list the order of exploration



a, f, d, b, c, e??

Exercise (2) – DFS

Apply **DFS** to the following graph starting from vertex **a** and list the order of exploration

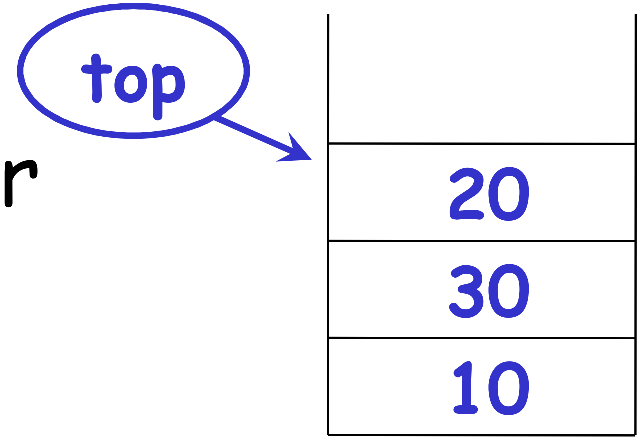


a, e, b, …? a, b, f, d, c, …?

DFS – Pseudo code (recursive)

```
Algorithm DFS(vertex v)
  visit v
  for each unvisited neighbor w of v do
  begin
      DFS(w)
  end
```

Data Structure - Stack



Data organised in a vertical manner

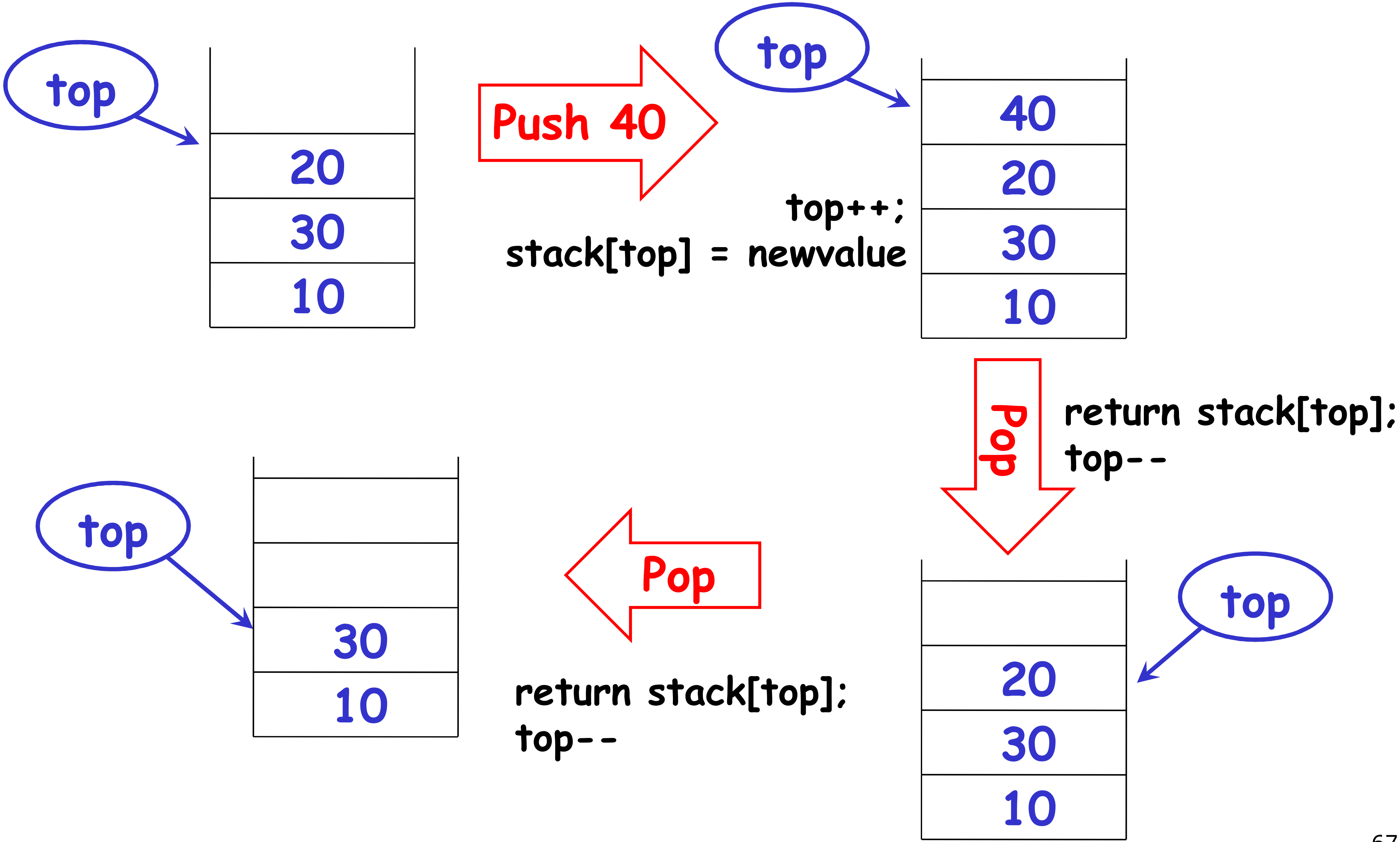
LIFO: last-in-first-out

Top: top of stack

Operations: push & pop

- **push:** adds a new element on top of stack
- **pop:** remove the element from top of stack

Data Structure - Stack

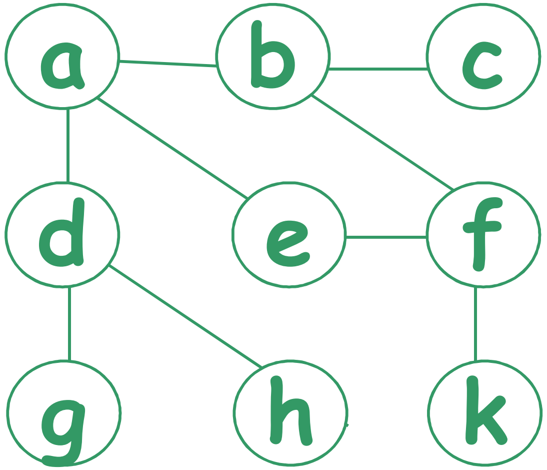


(Graph)

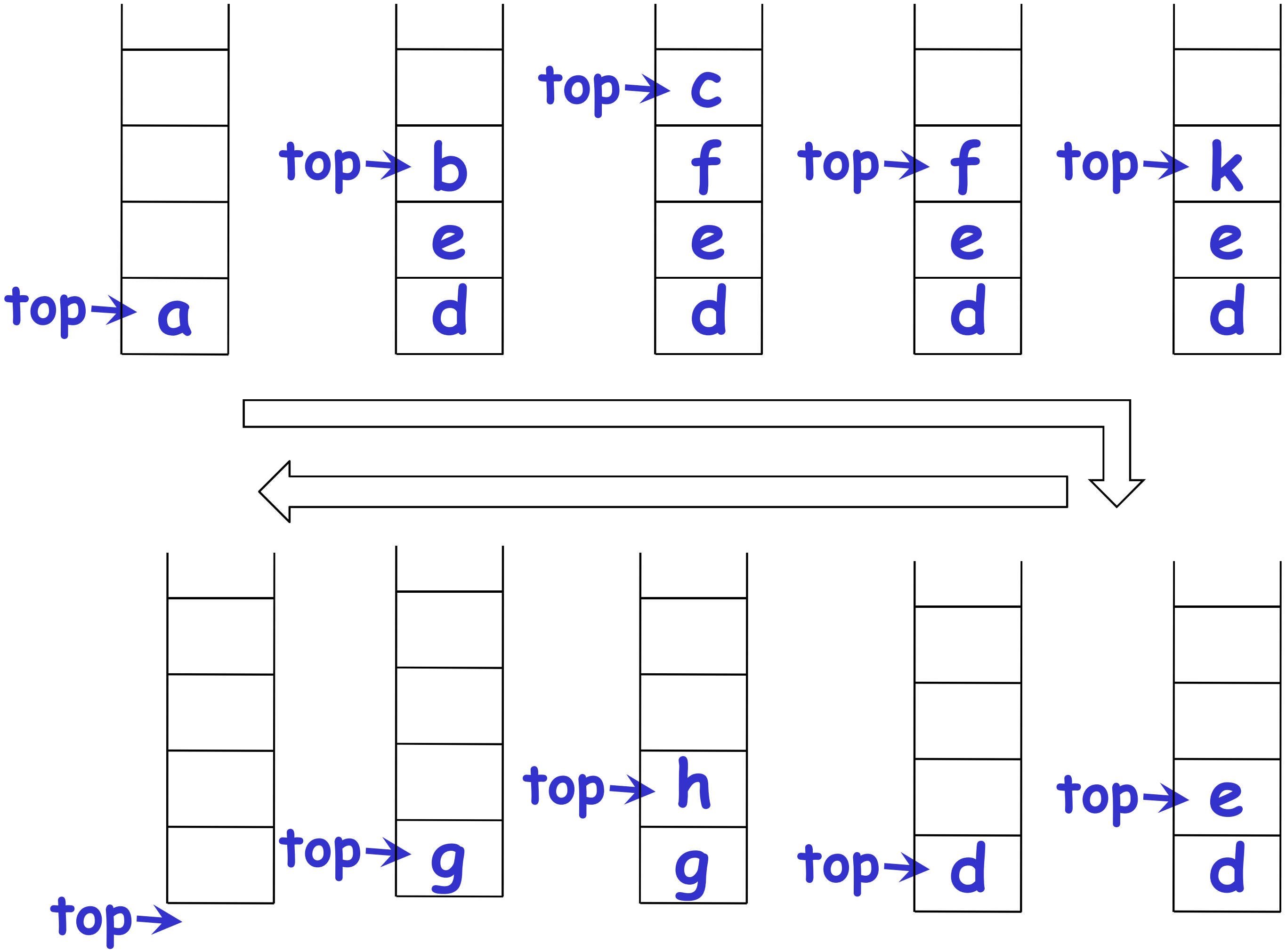
DFS – Pseudo code (using stack)

```
unmark all vertices
push starting vertex u onto top of stack S
while S is nonempty do
begin
   pop a vertex v from top of S
   if (v is unmarked) then
   begin
      visit and mark v
      for each unmarked neighbor w of v do
         push w onto top of S
   end
end
```

DFS using Stack



a, b, c, f, k, e, d, h, g



Tree ...

Outline

- What is a tree?
- What are subtrees
- How to traverse a binary tree?
  - Pre-order, In-order, Postorder
- Application of tree traversal

Trees

An undirected graph G=(V,E) is a tree if G is connected and acyclic (i.e., contains no cycles)

Other equivalent statements:

1. There is exactly one path between any two vertices in G (G is connected and acyclic)
2. G is connected and removal of one edge disconnects G (removal of an edge {u,v} disconnects at least u and v because of [1])
3. G is acyclic and adding one edge creates a cycle (adding an edge {u,v} creates one more path between u and v, a cycle is formed)
4. G is connected and m=n-1 (where |V|=n, |E|=m)

Lemma: P(n): If a tree T has n vertices and m edges, then m=n-1.

Proof: By induction on the number of vertices.

Base case: A tree with single vertex does not have an edge.

Induction step: $P(n-1) \Rightarrow P(n)$ for $n > 1$?

Remove an edge from the tree T. By [2], T becomes disconnected. Two connected components T_1 and T_2 are obtained, neither contains a cycle (the cycle is also present in T otherwise).

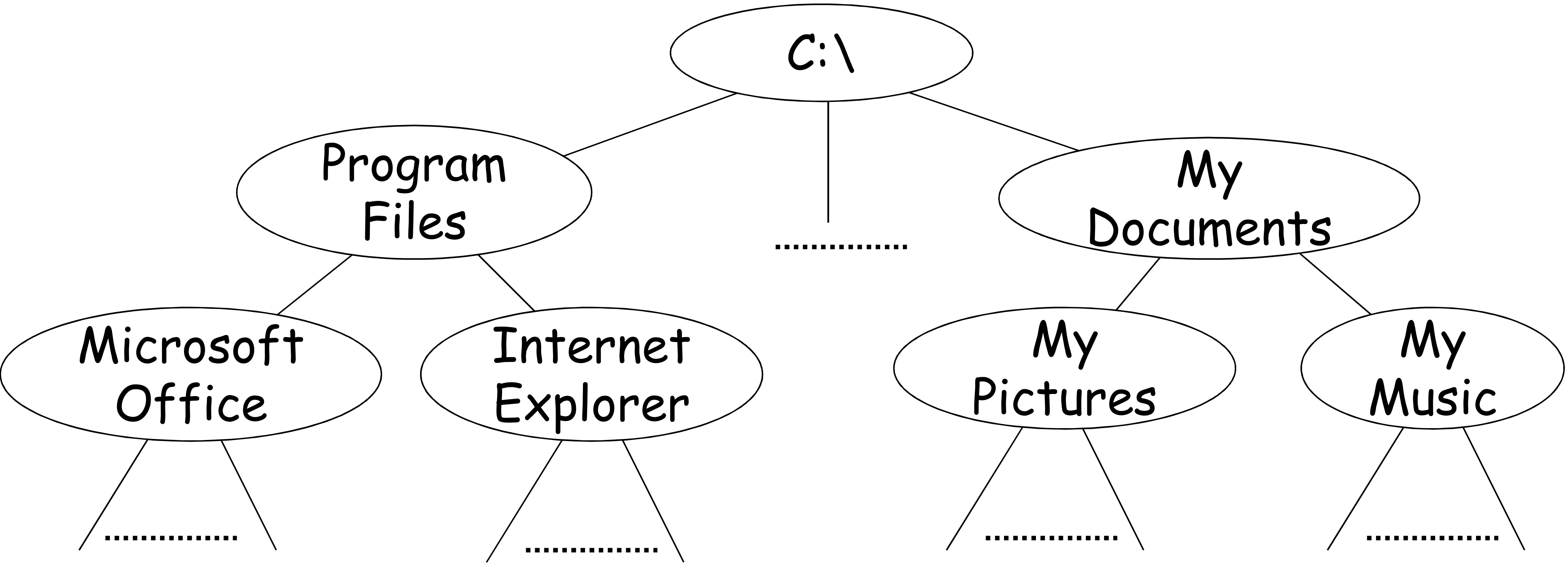
Therefore, both T_1 and T_2 are trees. Let n_1 and n_2 be the number of vertices in T_1 and T_2. **[$n_1+n_2 = n$]**

By the induction hypothesis, **T_1 and T_2 contains n_1-1 and n_2-1 edges**.

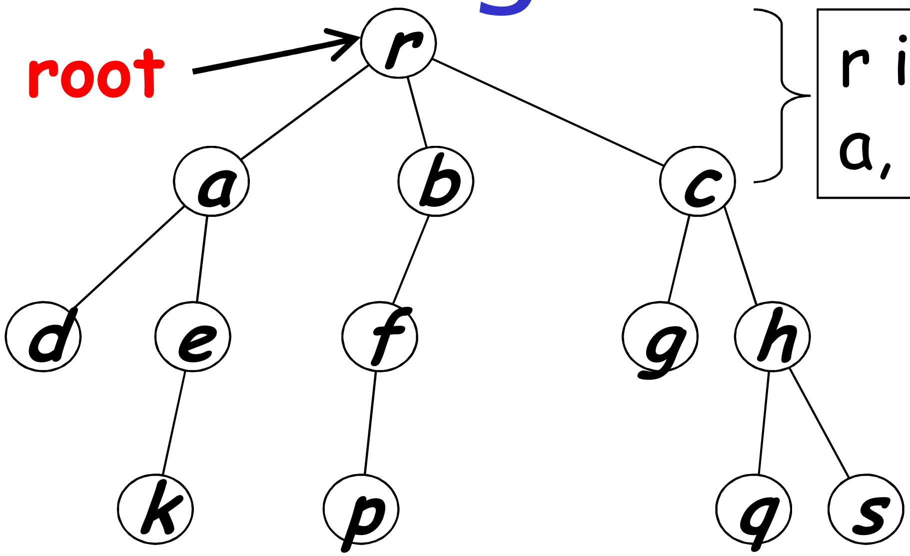
Hence, T contains $(n_1-1) + (n_2-1) + 1 = n-1$ edges.

Rooted trees

Tree with hierarchical structure, e.g., directory structure of file system



Terminologies



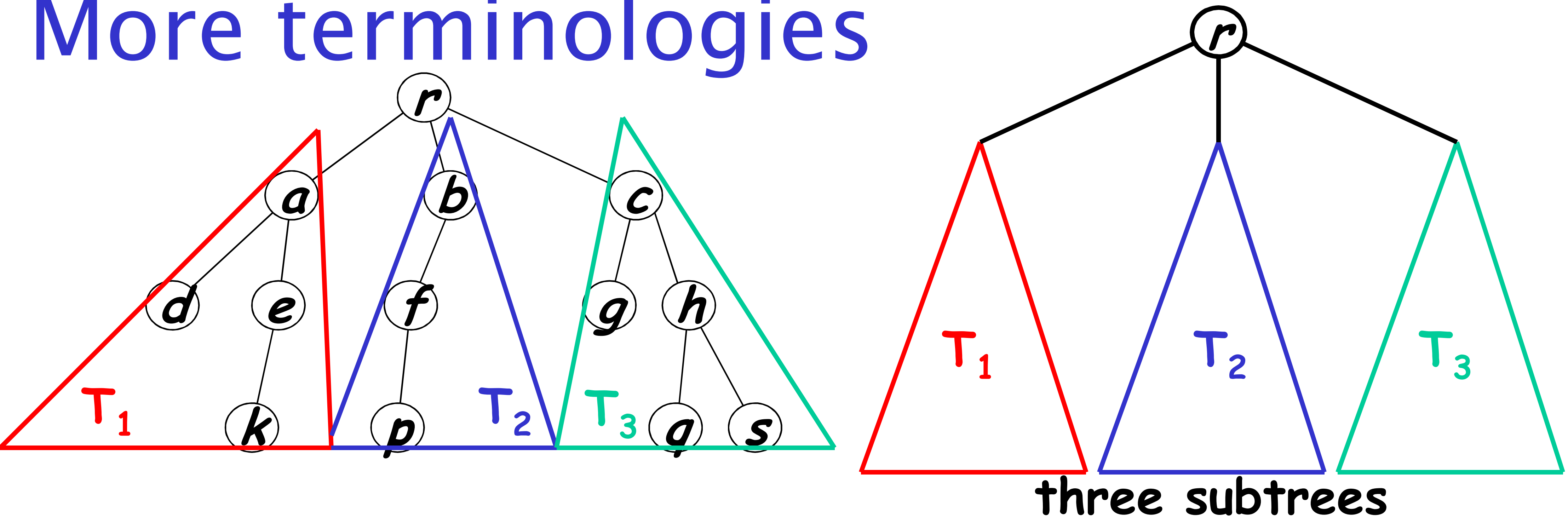
r is **parent** of a, b & c;
a, b, & c are **children** of r

deg-0: d, k, p, g, q, s (**leaves**)
deg-1: b, e, f
deg-2: a, c, h
deg-3: r

What is the degree of this tree?

- Topmost vertex is called the **root**.
- A vertex ***u*** may have some **children** directly below it, ***u*** is called the **parent** of its children.
- **Degree** of a ***vertex*** is the no. of children it has. (N.B. it is different from the degree in an unrooted tree.)
- Degree of a ***tree*** is the max. degree of all vertices.
- A vertex with no child (degree-0) is called a **leaf**. All others are called **internal vertices**.

More terminologies

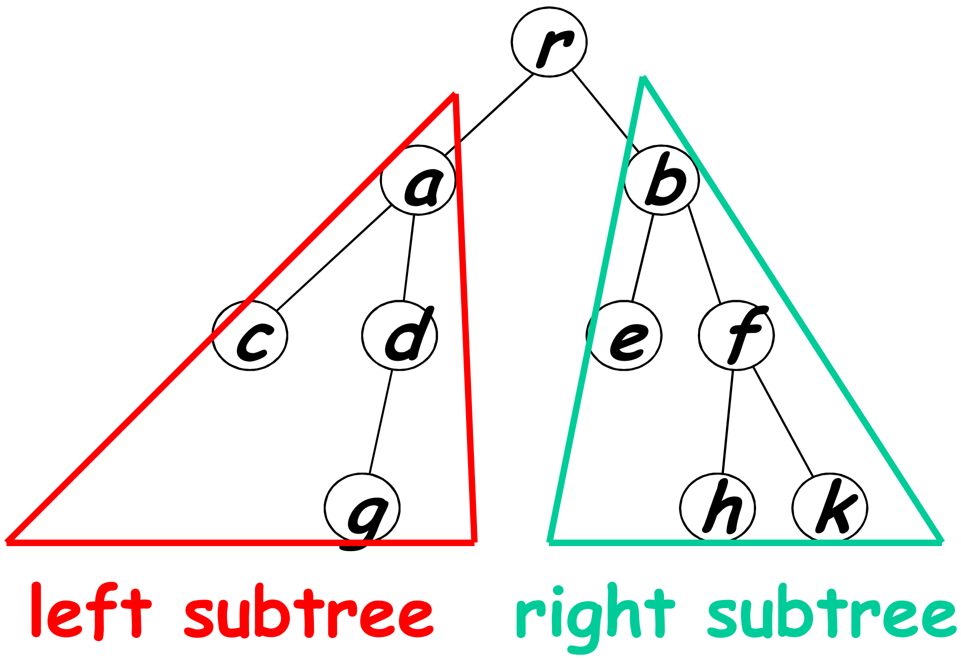


- We can define a tree recursively
  - A single vertex is a tree.
  - If $T_1, T_2, \ldots, T_k$ are **disjoint** trees with roots $r_1, r_2, \ldots, r_k$, the graph obtained by attaching a *new vertex r* to each of $r_1, r_2, \ldots, r_k$ with a single edge forms a tree T with root r.
  - $T_1, T_2, \ldots, T_k$ are called **subtrees** of T.

which are the roots of the subtrees?

Binary tree

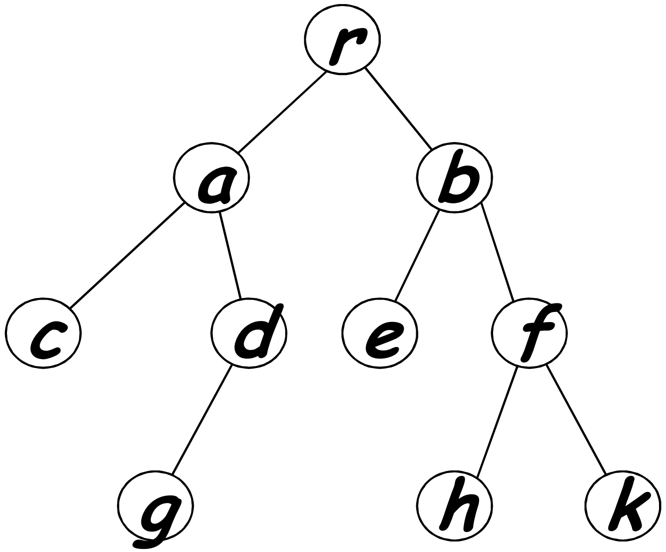
- a tree of degree at most TWO
- the two subtrees are called left subtree and right subtree (may be empty)



There are *three* common ways to traverse a binary tree:

- **preorder** traversal - vertex, left subtree, right subtree
- **inorder** traversal - left subtree, vertex, right subtree
- **postorder** traversal - left subtree, right subtree, vertex

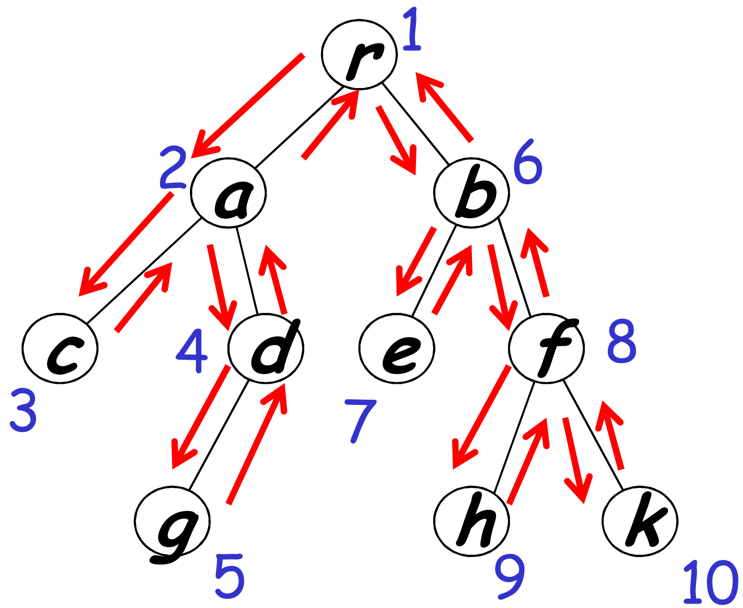
Traversing a binary tree



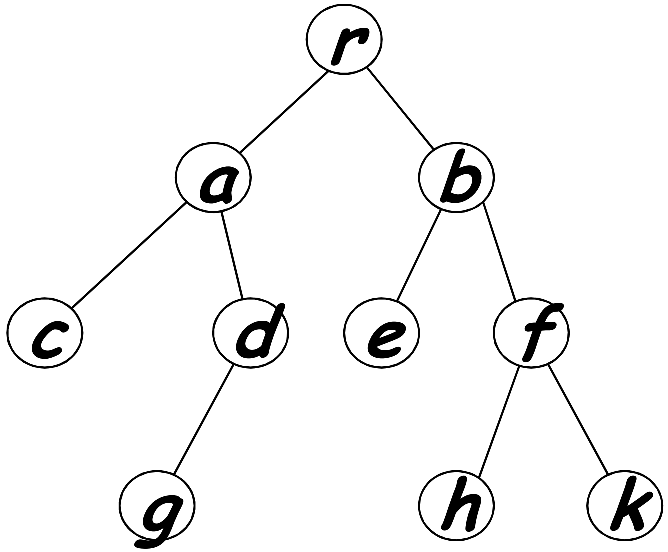
preorder traversal

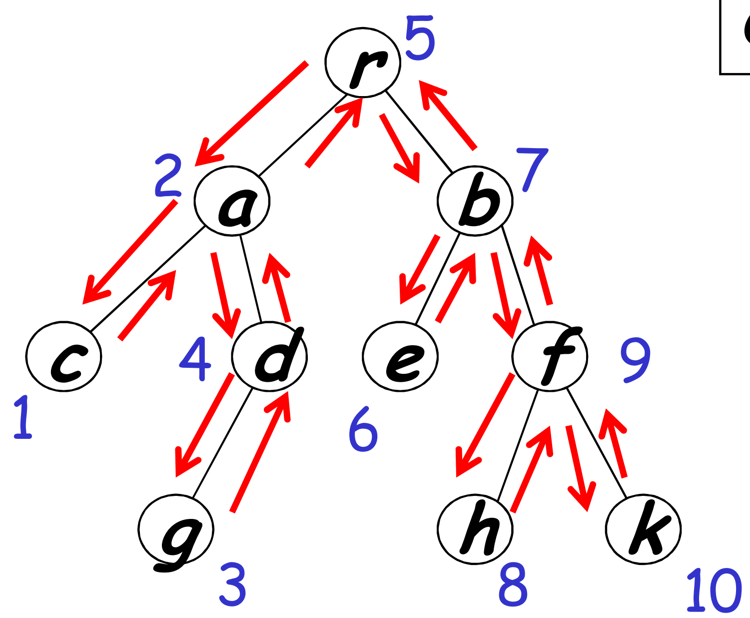
- vertex, left subtree, right subtree

r -> a -> c -> d -> g -> b -> e -> f -> h -> k



Traversing a binary tree





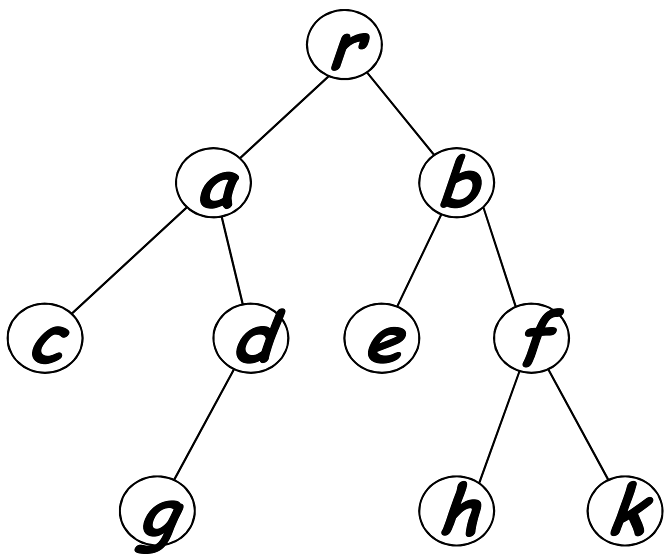
preorder traversal
- vertex, left subtree, right subtree

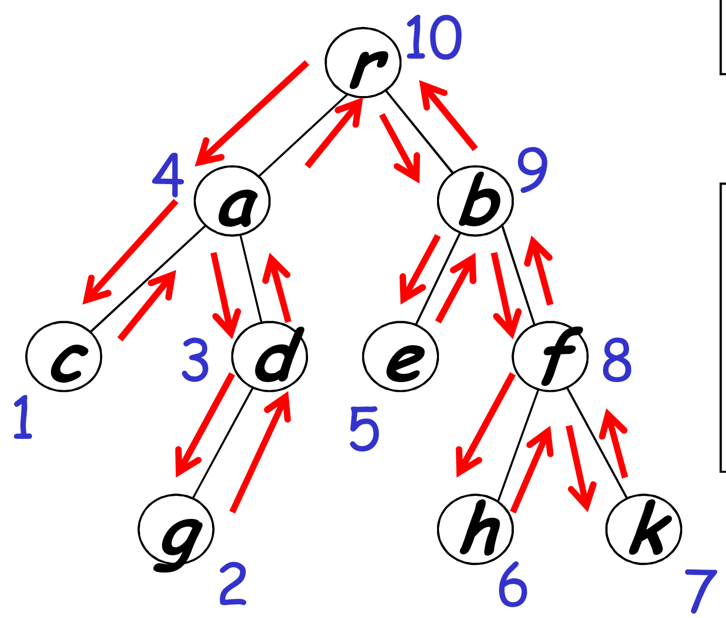
r -> a -> c -> d -> g -> b -> e -> f -> h -> k

inorder traversal
- left subtree, vertex, right subtree

c -> a -> g -> d -> r -> e -> b -> h -> f -> k

Traversing a binary tree





preorder traversal
- vertex, left subtree, right subtree

r -> a -> c -> d -> g -> b -> e -> f -> h -> k

inorder traversal
- left subtree, vertex, right subtree

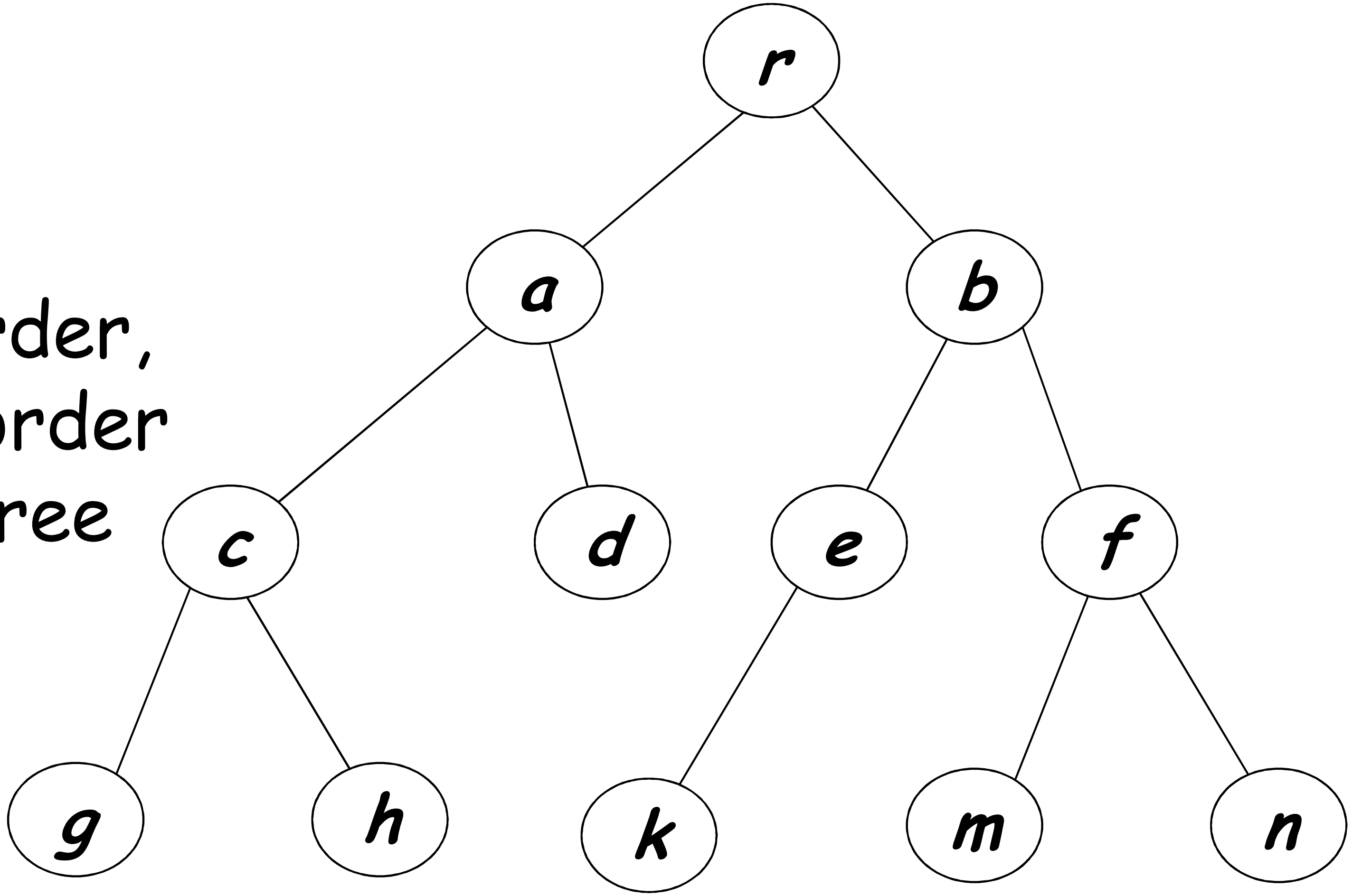
c -> a -> g -> d -> r -> e -> b -> h -> f -> k

postorder traversal
- left subtree, right subtree, vertex

c -> g -> d -> a -> e -> h -> k -> f -> b -> r

Exercise

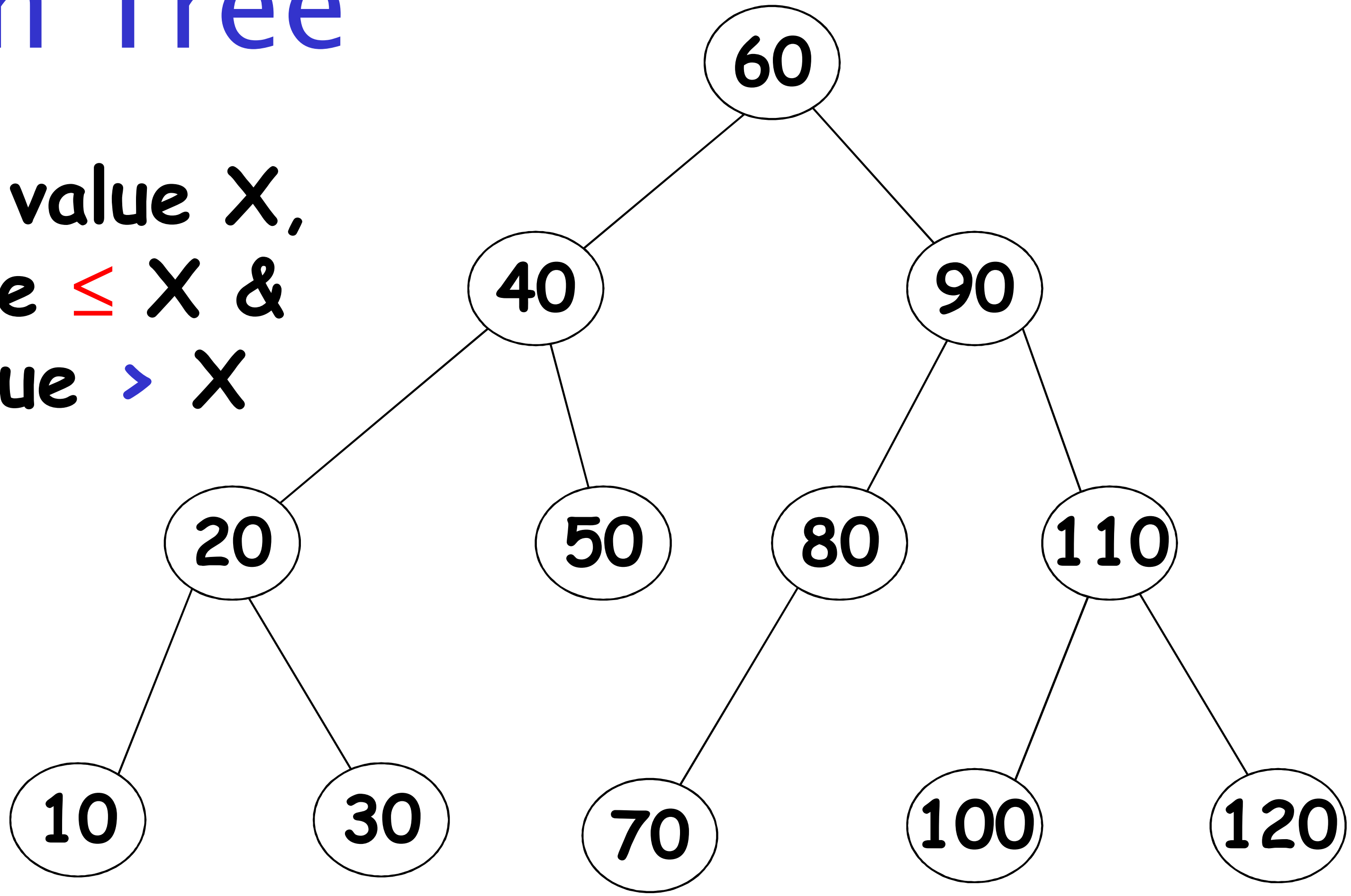
Give the order of traversal of preorder, inorder, and postorder traversal of the tree



preorder:
inorder:
postorder:

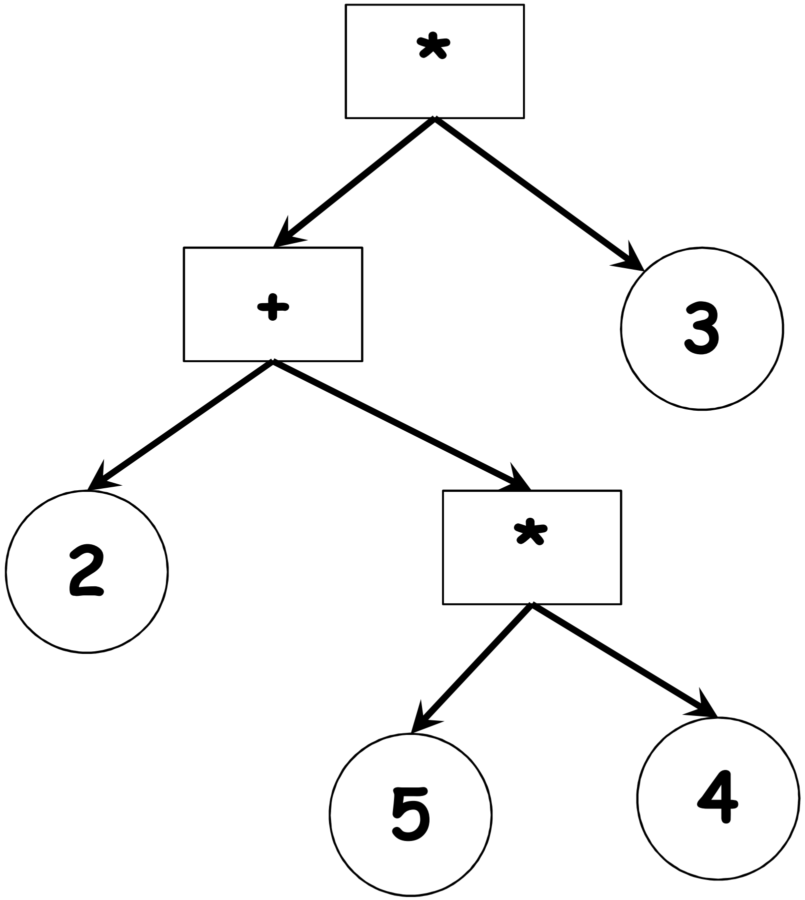
Binary Search Tree

for a vertex with value X,
left child has value ≤ X &
right child has value > X



which traversal gives numbers in ascending order?

Expression Tree



(2+5*4)*3

postorder traversal gives
2 5 4 * + 3 *

1. push numbers onto stack
2. when operator is encountered,
   pop 2 numbers, operate on them & push results back to stack
3. repeat until the expression is exhausted