WS 2020/21

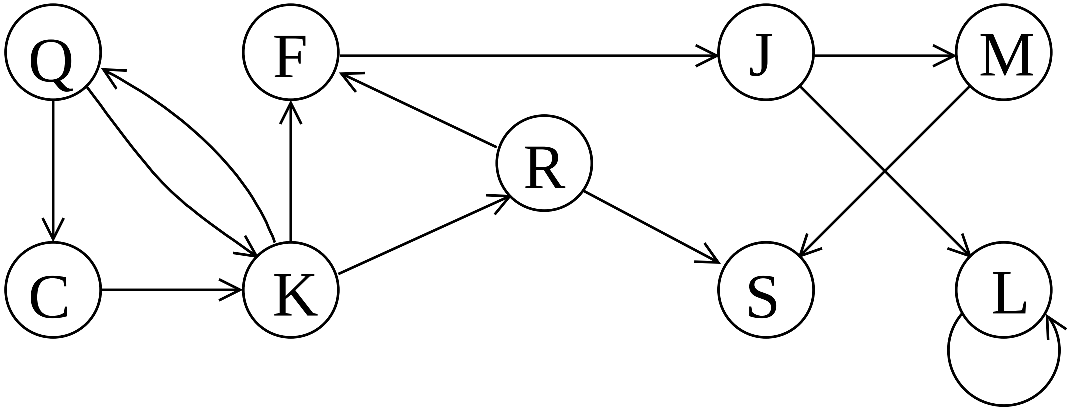
Algorithms

Chapter 3.1

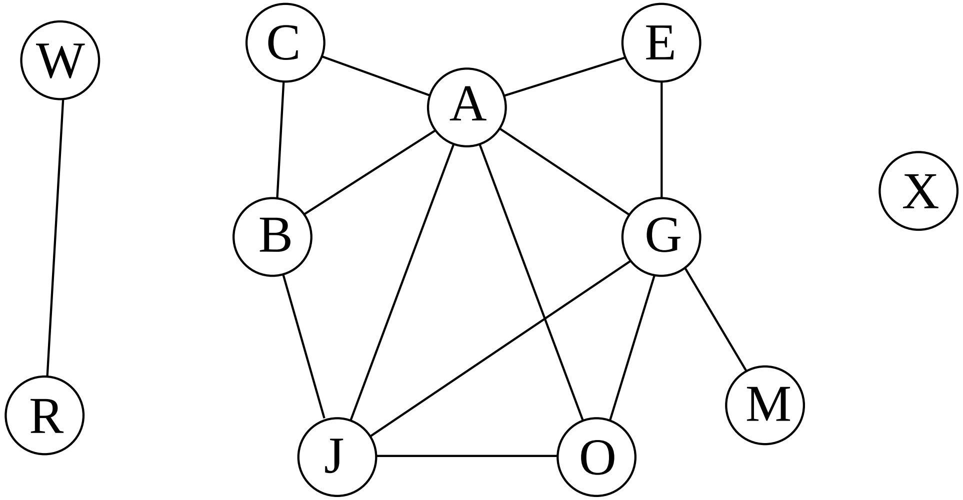
Graphs and directed graphs

Martin Dietzfelbinger

February 2021



Directed graph (Digraph)

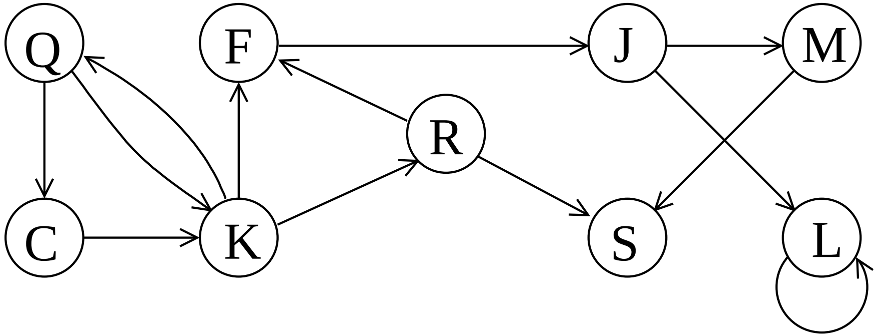


(Undirected) Graph

Graphs and **Digraphs** are an ubiquitous (data) structure for modelling application situations, inside and outside computer science.

They model:

- cities and interstate roads, crossings and innercity streets
- gates and wires on a chip
- components of a “system” and interconnections/relations
- states of a system and transitions
- flow diagrams in program design
- data flow diagrams for program analysis
- actions with incompatibility relation
- terminals of a transportation system, with capacities for transport
- people and social relations
- and many more.

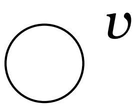


Definition 3.1.1

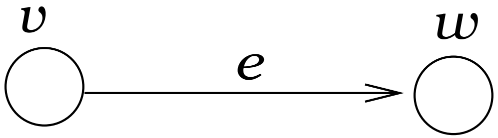
A **directed graph** or **digraph** G

is a pair $\boldsymbol{(V,E)}$, where V is a finite set and
E is a subset of $V \times V = \{(v,w) \mid v,w \in V\}$.

The elements of V are called **nodes** (or **vertices**),
the elements of E are called **edges** (or **arcs**).



Nodes are drawn as little circles,



edges as arrows.

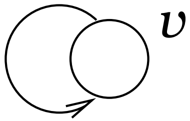
If $e = (v, w)$ is an edge (of G), then

v are w **incident** with e (v, w **lie on** e),

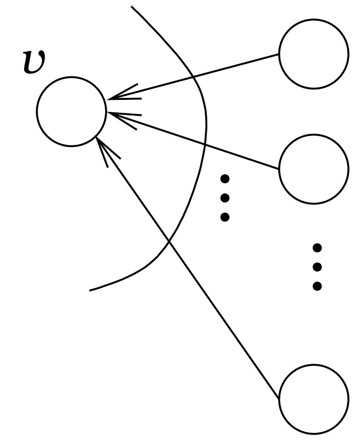
v and w are called **adjacent**,

w is also called a **successor** of v,

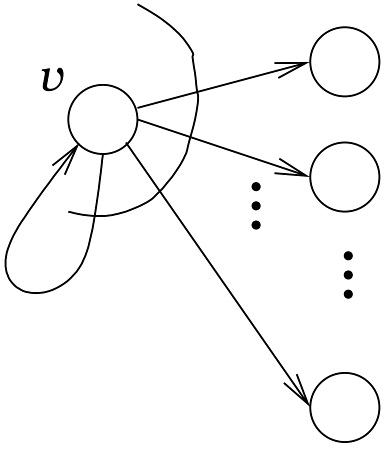
v is also called a **predecessor** of w.



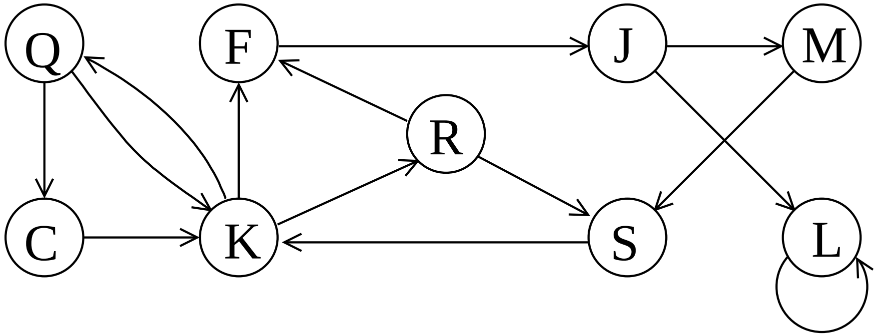
An edge (v, v) is called a **loop**.



The **indegree** of a node v
is the number of edges that enter v:
$\textbf{indeg}(v) = |\{e \in E \mid e = (u, v) \text{ for some } u \in V\}|$.



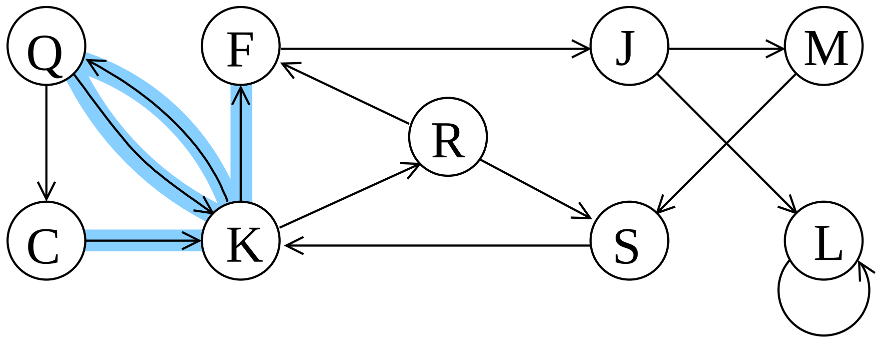
The **outdegree** of a node v
is the number of edges that leave v:
$\textbf{outdeg}(v) = |\{e \in E \mid e = (v, w) \text{ for some } w \in V\}|$.



Lemma 3.1.1

$$\sum_{v \in V} \mathrm{indeg}(v) = \sum_{v \in V} \mathrm{outdeg}(v) = |E|.$$

Proof: In both sums every edge is counted exactly once. □



Definition 3.1.2

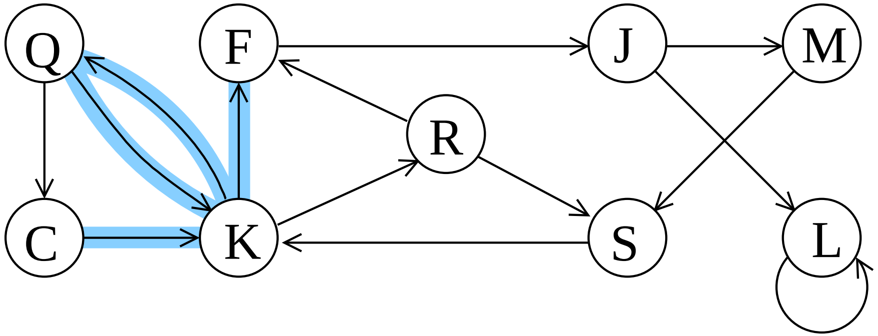
Let $G = (V, E)$ be a digraph.

(a) A **walk** in G

is a sequence $p = (v_0, v_1, \ldots, v_k)$ of nodes, where $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$.

Equivalent: A sequence $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ of edges.

Examples: $(\mathrm{C}, \mathrm{K}, \mathrm{R}, \mathrm{S})$, $(\mathrm{C}, \mathrm{K}, \mathrm{Q}, \mathrm{K}, \mathrm{F})$, $(\mathrm{F}, \mathrm{J}, \mathrm{L}, \mathrm{L}, \mathrm{L}, \mathrm{L})$.



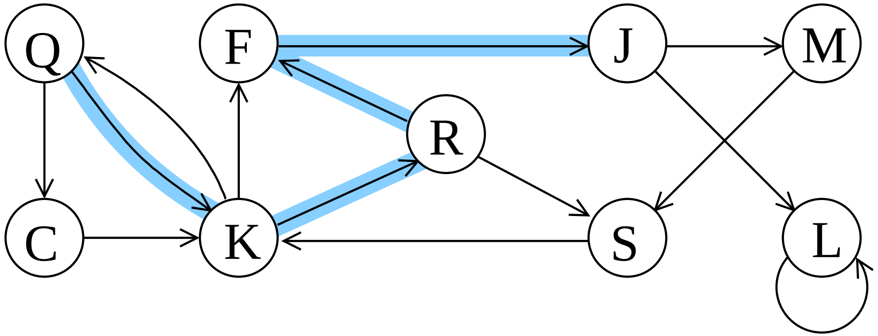
(b) The **length** of $(v_0, v_1, \ldots, v_k)$ is k (the **number of edges**, or number of hops).
(E.g.: $(\mathrm{C}, \mathrm{K}, \mathrm{Q}, \mathrm{K}, \mathrm{F})$ has length 4.)
Walk (v) has no edge, hence its length is 0.

(c) We write $\boldsymbol{v \rightsquigarrow_G w}$ or $\boldsymbol{v \rightsquigarrow w}$, if there is a walk in G $(v_0, v_1, \ldots, v_k)$ such that $v = v_0$ and $w = v_k$ (**"a walk from $\boldsymbol{v}$ to $\boldsymbol{w}$"**).

Example: $\mathrm{F} \rightsquigarrow \mathrm{M}$, $\mathrm{Q} \rightsquigarrow \mathrm{L}$, $\mathrm{S} \rightsquigarrow \mathrm{S}$, but **not** $\mathrm{L} \rightsquigarrow \mathrm{F}$.

Observation The relation $\rightsquigarrow$ is **reflexive** and **transitive**.

((v) is a walk; walks can be **concatenated**.)



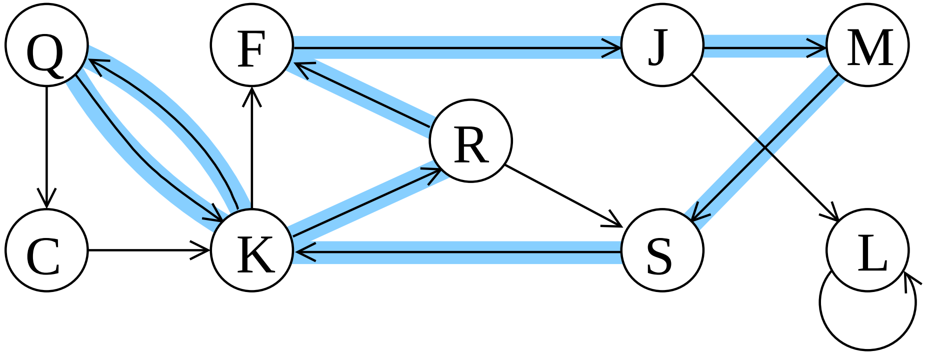
Definition 3.1.3

A walk $(v_0, v_1, \ldots, v_k)$ in in a digraph G is called a (simple) **path**, if $v_0, v_1, \ldots, v_k$ are distinct.

Example: $(\mathrm{Q}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J})$.

Observation If $v \rightsquigarrow w$, then there is a path $(v_0, v_1, \ldots, v_l)$ with $v = v_0$ and $w = v_l$ (a path "from v to w").

(If walk $(v_0, v_1, \ldots, v_k)$ contains u twice, replace subsequence $\ldots, u, \ldots, u, \ldots$ by $\ldots, u, \ldots$, and repeat, if necessary.)



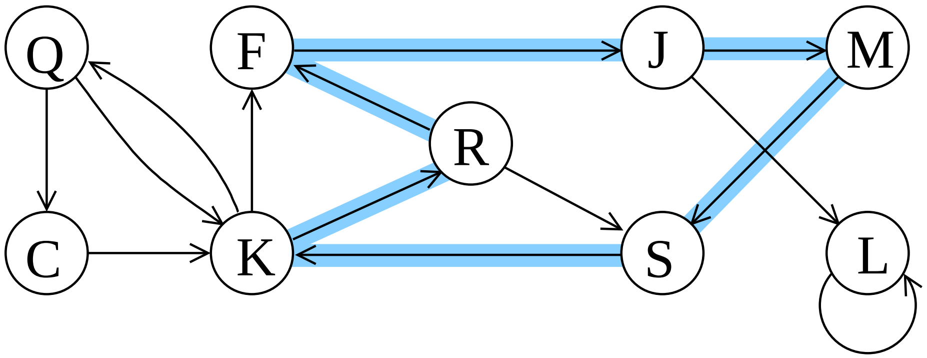
Definition 3.1.4

(a) A walk $(v_0, v_1, \dots, v_k)$ in a digraph G is called a **cycle** if $k \geq 1$ and $v_0 = v_k$.

Remark: Each loop $(v, v) \in E$ is a cycle of length 1.

Example: $(\mathrm{K}, \mathrm{Q}, \mathrm{C}, \mathrm{K})$, (L, L), $(\mathrm{L}, \mathrm{L}, \mathrm{L})$, $(\mathrm{Q}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{Q})$ are cycles.

Remark: Cycles that differ only by a cyclic shift, as $(\mathrm{K}, \mathrm{Q}, \mathrm{C}, \mathrm{K})$ and $(\mathrm{Q}, \mathrm{C}, \mathrm{K}, \mathrm{Q})$, are regarded as the same cycle.



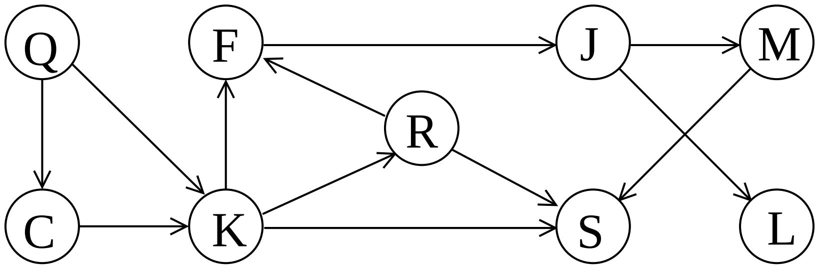
(b) A cycle $(v_0, v_1, \ldots, v_{k-1}, v_0)$ is called **simple** if $v_0, \ldots, v_{k-1}$ are different.

Example: $(\mathrm{J}, \mathrm{M}, \mathrm{S}, \mathrm{K}, \mathrm{R}, \mathrm{F}, \mathrm{J})$, $(\mathrm{K}, \mathrm{C}, \mathrm{Q}, \mathrm{K})$, (L, L) are simple cycles.

Observation:

If digraph G contains a cycle, it also contains a simple cycle.

(If $(v_0, \ldots, v_{k-1})$ contains node u twice, replace subsequence $\ldots, u, \ldots, u, \ldots$ by $\ldots, u, \ldots$, repeat if necessary.)



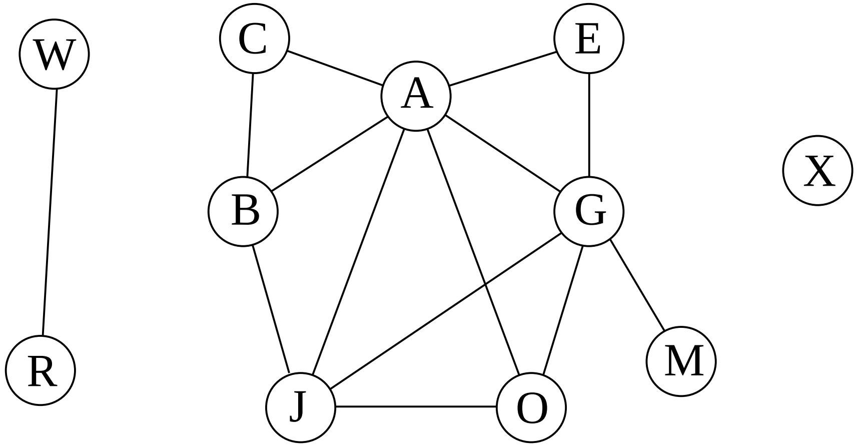
Definition 3.1.5

A digraph G is **acyclic**, if G does not contain a cycle. Otherwise G is called **cyclic**.

A very important class of graphs are the

directed acyclic graphs.

(Abbreviation: **DAG**s or **dag**s.)



Definition 3.1.6

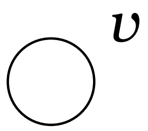
An **undirected graph**, often also: a **graph G**
is a pair (V, E), where V is a finite set and

E is a subset of $[V]_2 = \{\{v, w\} \mid v, w \in V, v \neq w\}$ ist.

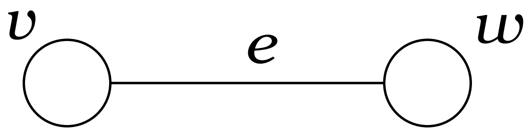
Notation: $\boldsymbol{(v, w)}$ for $\{v, w\}$.

In the picture: $V = \{\mathrm{A, B, C, E, G, J, M, O, R, W, X}\}$,

$$E = \{(\mathrm{A, B}), (\mathrm{A, C}), (\mathrm{C, B}), (\mathrm{A, E}), (\mathrm{A, G}), (\mathrm{G, E}), (\mathrm{A, J}), (\mathrm{A, O}), (\mathrm{B, J}),\\ (\mathrm{J, O}), (\mathrm{J, G}), (\mathrm{O, G}), (\mathrm{G, M}), (\mathrm{R, W})\}.$$



The elements of V are called **nodes**
(or **vertices**).
Nodes are drawn as little circles.



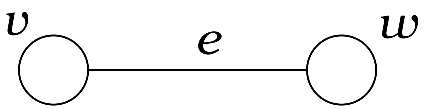
The elements of E are called **edges**.
Edges are drawn as (undirected) lines
(not necessarily straight).

Convention:

Edge $\{u, v\}$ $(= \{v, u\})$ is written as $\boldsymbol{(u, v)}$.

(**Only(!)**) For edges of undirected graphs we have: $(u, v) = (v, u)$.

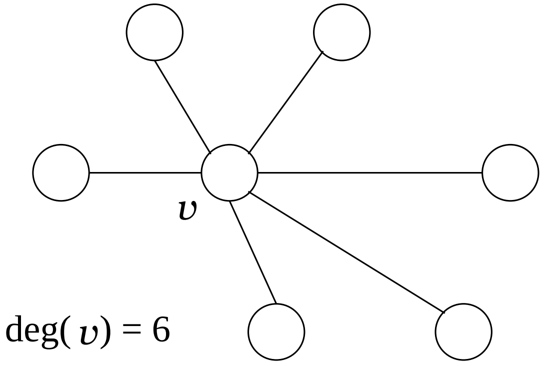
Definition 3.1.7



Let $G = (V, E)$ be an undirected graph.
If $e = (v, w)$ is an edge (of G), then

v and w are **incident** with e, v and w are **adjacent**; v is called a **neighbor** of w and *vice versa*.

"Loops", i.e. "edges" (v, v), normally are not admitted in undirected graphs.



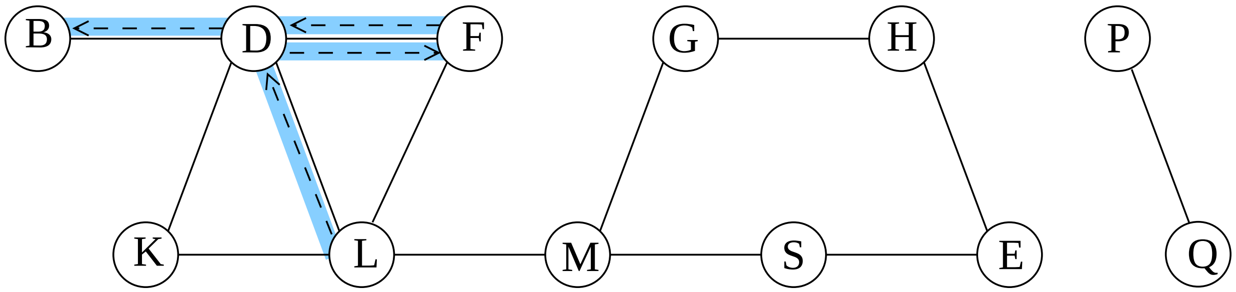
The **degree** of a node v is
deg(v) $= |\{e \in E \mid e = (v, w)$ for some $w \in V\}|$.
Nodes v with degree 0 are called **isolated** (they have no neighbors).

Lemma 3.1.8 ("handshaking lemma")

For a graph $G = (V, E)$ we have the following:

$$\sum_{v \in V} \deg(v) = 2|E|.$$

Proof: Each edge (u, v) in E contributes 1 to $\deg(u)$ and 1 to $\deg(v)$. □

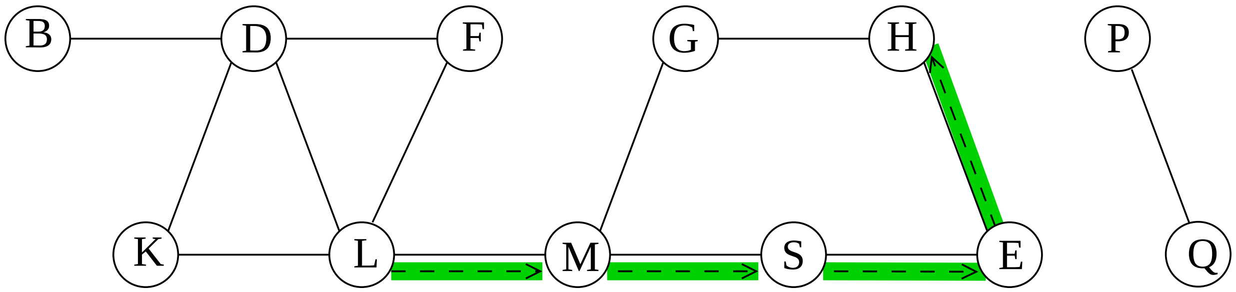


Definition 3.1.9

Let $G = (V, E)$ be a graph.
(a) A **walk** in G is a sequence $(v_0, v_1, \ldots, v_k)$ of nodes,
i.e. elements of V, where $(v_{i-1}, v_i) \in E$ for $1 \le i \le k$.

Walks in the example graph: $(\mathrm{L}, \mathrm{D}, \mathrm{F}, \mathrm{D}, \mathrm{B})$ (length 4), $(\mathrm{L}, \mathrm{F}, \mathrm{D}, \mathrm{L}, \mathrm{M}, \mathrm{S})$, (length 5).

(b) The **length** of a walk $(v_0, v_1, \ldots, v_k)$ is the **number of edges k**.
($k = 0$ is legal.)



(c) A walk $(v_0, v_1, \ldots, v_k)$ in a graph G is called a **path** if $v_0, v_1, \ldots, v_k$ are distinct.

Path in example: $(\mathrm{L}, \mathrm{M}, \mathrm{S}, \mathrm{E}, \mathrm{H})$, length 4.

walks, not paths: see previous slide.

Lemma 3.1.10

Let G be a graph. If there is a walk $(v_0, \ldots, v_k)$ with $v_0 = v$ and $v_k = w$ (“from v to w”), then there is a path from v to w.

(*Proof* as for digraphs.)

Definition 3.1.11

Let G be a graph. If $v, w \in V$ are connnected by a walk $p = (v_0, v_1, \ldots, v_k)$ with $v_0 = v$ and $v_k = w$, we write $\boldsymbol{v \sim_G w}$ or $\boldsymbol{v \sim w}$.

Lemma 3.1.12

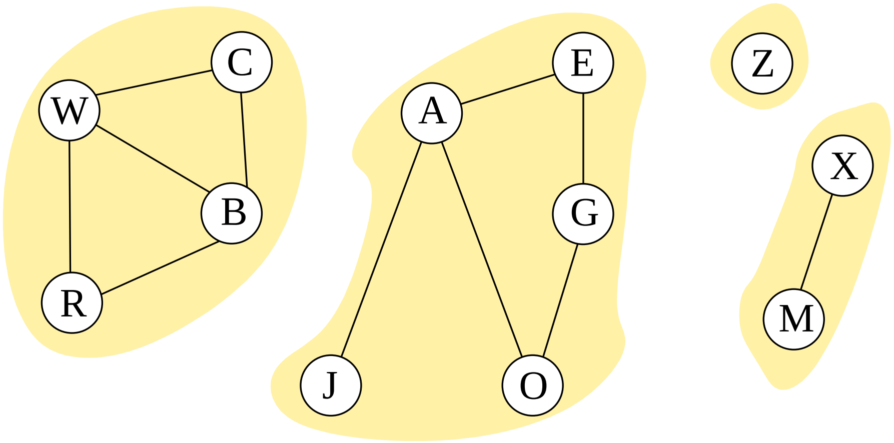
The 2-ary relation $\sim_G$ on V is an **equivalence relation**, i.e. it is
reflexive: $v \sim_G v$,
symmetric: $v \sim_G w \Rightarrow w \sim_G v$,
transitive: $u \sim_G v \wedge v \sim_G w \Rightarrow u \sim_G w$.

Proof:
Reflexivity: (v) is a walk from v to v, length 0;
Symmetry: traverse any walk from v to w in opposite direction;
Transitivity: Can concatenate walks from u to v and from v to w to get walk from u to w. $\square$

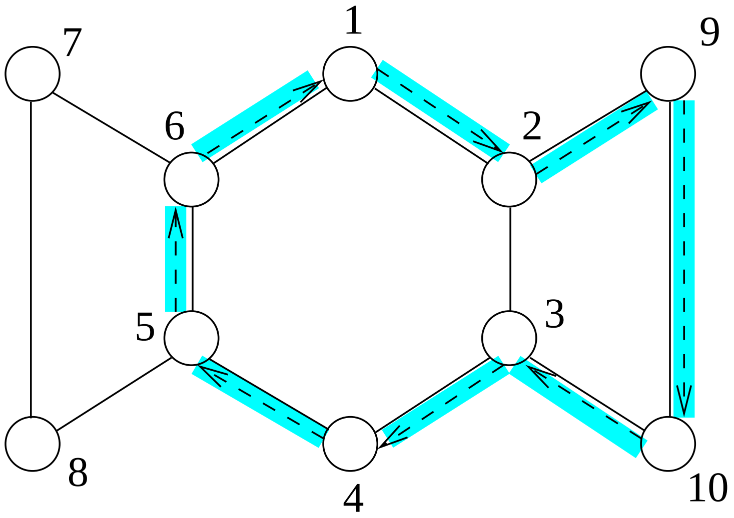
Definition 3.1.13

(a) The equivalence relation $\sim_G$ splits V into equivalence classes, the **(connected) components** of G.

Example: Graph with four connected components $\{\mathrm{B},\mathrm{C},\mathrm{R},\mathrm{W}\}$, $\{\mathrm{A},\mathrm{G},\mathrm{E},\mathrm{J},\mathrm{O}\}$, $\{\mathrm{M},\mathrm{X}\}$, $\{\mathrm{Z}\}$:



(b) A graph G with only one connected component (i.e.,in which $u \sim_G v$ for all $u, v \in V$), is called **connected**.



Simple cycles: (6,1,2,9,10,3,4,5,6)
and (1,6,5,4,3,10,9,2,1)

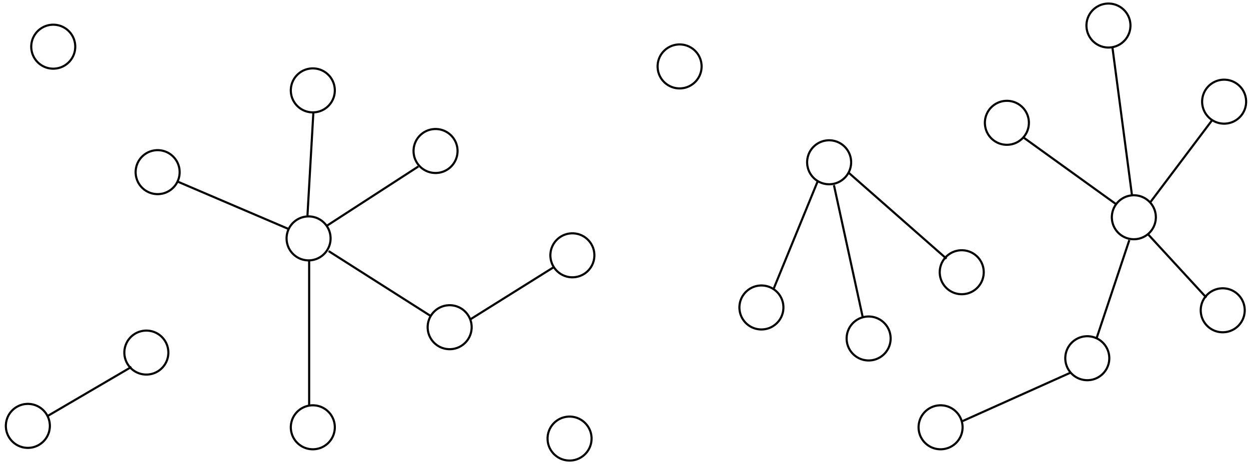
Definition 3.1.14 A walk $(v_0, v_1, \dots, v_k)$ in an (undirected) graph G is called a **(simple) cycle** if $k \geq 3$ and $v_0 = v_k$ and if in addition $v_0, v_1, \dots, v_{k-1}$ are distinct.

The starting point of a cycle is irrelevant: $(\mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{K}, \mathrm{J}, \mathrm{H}, \mathrm{G}, \mathrm{B})$ and $(\mathrm{K}, \mathrm{J}, \mathrm{H}, \mathrm{G}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{K})$ are regarded as "the same cycle".
Often also: Orientation is irrelevant, i.e. $(\mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{K}, \mathrm{J}, \mathrm{H}, \mathrm{G}, \mathrm{B})$ and $(\mathrm{B}, \mathrm{G}, \mathrm{H}, \mathrm{J}, \mathrm{K}, \mathrm{E}, \mathrm{D}, \mathrm{C}, \mathrm{B})$ are regarded as "the same cycle".

Definition 3.1.15

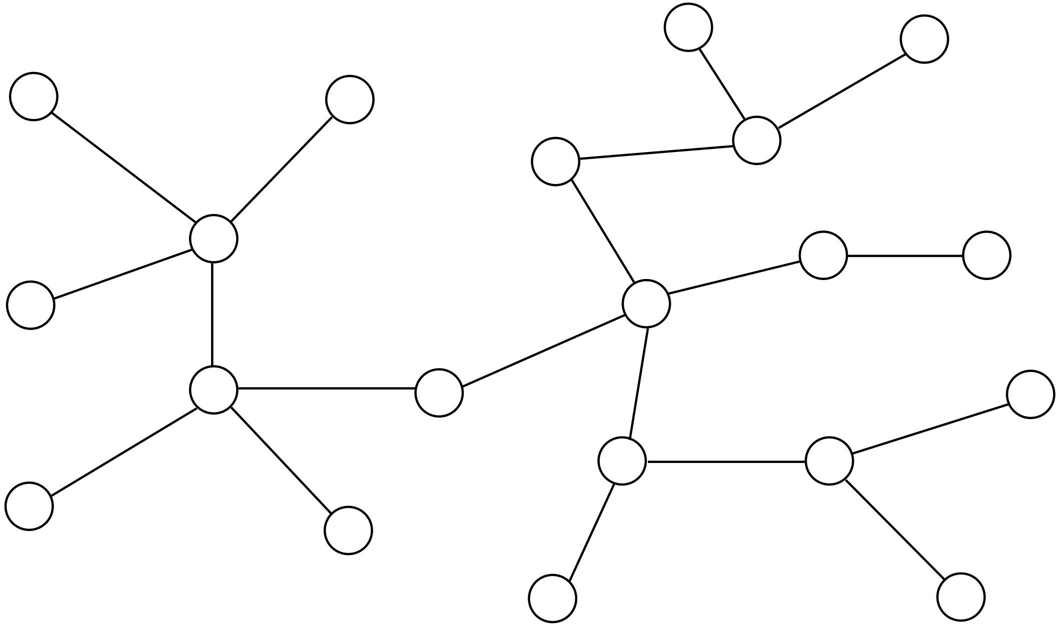
(a) A graph $G = (V, E)$ is **acyclic** if it **does not have a cycle**.

Example: An acyclic graph:



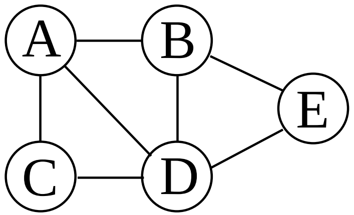
(b) A graph G is called a **free tree** or simply a **tree** if it is connected and acyclic.

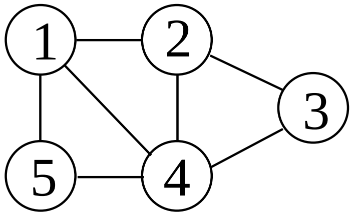
Example: A (free) tree with 20 nodes and 19 edges:



Remarks: The connected components of an acyclic graph are free trees. Acyclic graphs are also called **(free) forests**.

Data Structures for Digraphs and Graphs





nodes:

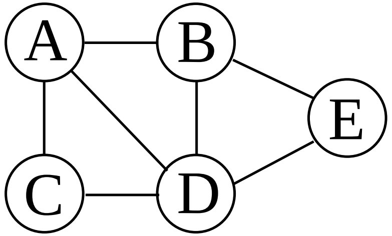
1: A
2: B
3: E
4: D
5: C

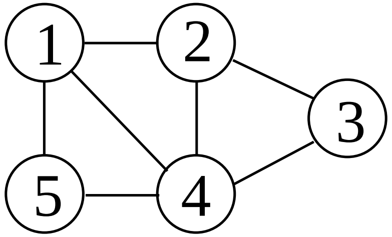
Let $G = (V, E)$ be a graph or a digraph (*directed graph*).

V is an arbitrary finite set. (Here: $\{\mathrm{A,B,C,D,E}\}$.)

Arrange $n = |V|$ nodes arbitrarily, e. g. as
$V = \{v_1, \ldots, v_n\}$ and represent them in an array:

```
nodes: array [1..n] of nodetype
```





nodes:

1: A
2: B
3: E
4: D
5: C

The name v_i of the node and other attributes ("labels") are fields (attributes) in the entries of the `nodes` array.

We assume the nodes are numbered $1, 2, \ldots, n$ and there is a `nodes` array.

In this representation (i, j) is an edge if and only if in the original graph G the pair (v_i, v_j) is an edge.

Definition

If $G = (V, E)$ is a graph or a digraph with node set $V = \{1, \ldots, n\}$ then the **adjacency matrix** of G is the $n \times n$ matrix

$$A = A_G = (a_{ij})_{1 \le i \le n, 1 \le j \le n}$$

with

$$a_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E \\ 0, & \text{if } (i, j) \notin E. \end{cases}$$

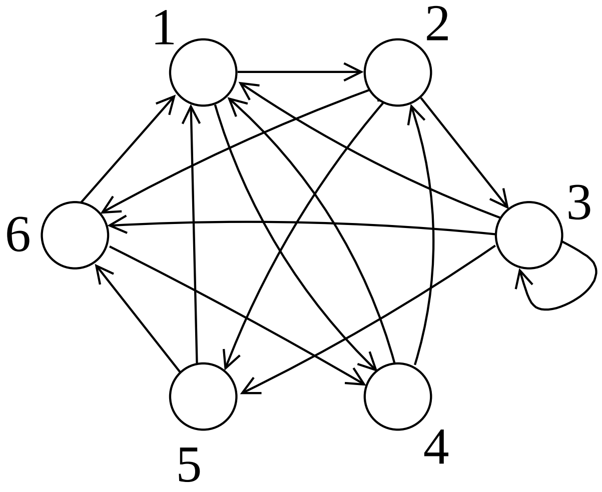
In most programming languages:
Matrix is realized as a 2-dimensional array $\mathtt{A}[1\,..\,n, 1\,..\,n]$ with entries from $\{0, 1\}$.

Obvious: Read or write access to a_{ij} in time $O(1)$.

Example: A Digraph.

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

$\widehat{=}$

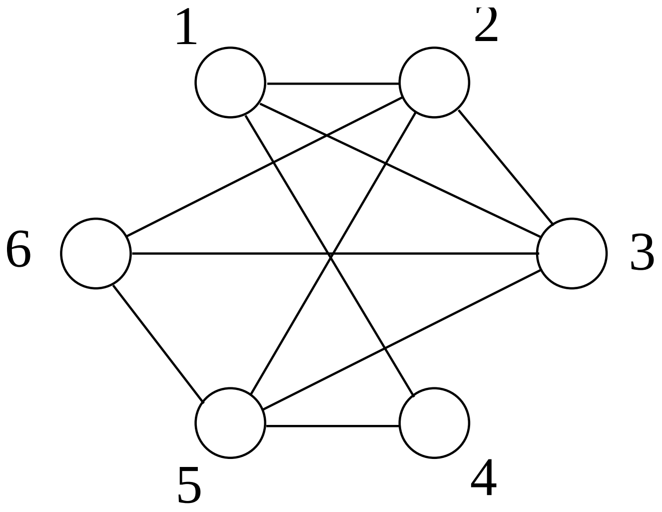


Number of 1s in row i = outdeg(i);
Number of 1s in column j = indeg(j).

Example: An undirected graph.

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

$\widehat{=}$



The adjaceny matrix of an undirected **graph** is **symmetric**.

Number of 1s in row/column $i = \deg(i)$.

Observations

If an n-node graph (directed or undireccted) is represented by an **adjacency matrix**, we have:

(a) storage space is $\Theta(n^2)$ [bits];
(b) in $O(1)$ time one can find (or change) a_{ij};
(c) finding **all** successors, predecessors or neighbors of a node takes time $\Theta(n)$.
(row/column traversal; order: $1, \ldots, n$)

The storage space is rather large if $|E| \ll n^2$ ("sparse" graphs).

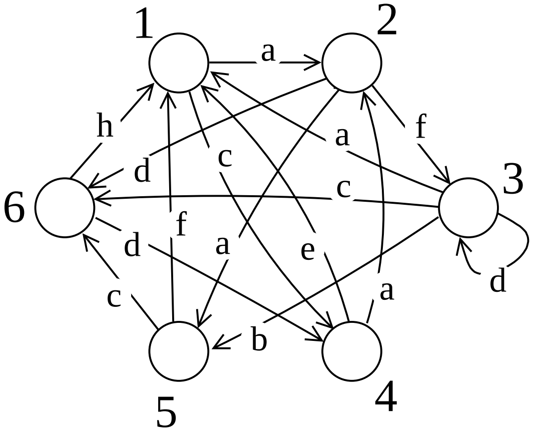
Reducing the space: Store w bits of the matrix in one word of bit length w, as a *bit vector*.

Extension: Edges may be labeled also by elements of M (lengths, weights, costs, capacities, etc.). Then we use an array with entries from $M \cup \{-\}$ or $M \cup \{\infty\}$ ("$-$" resp. "∞" means: "does not exist")

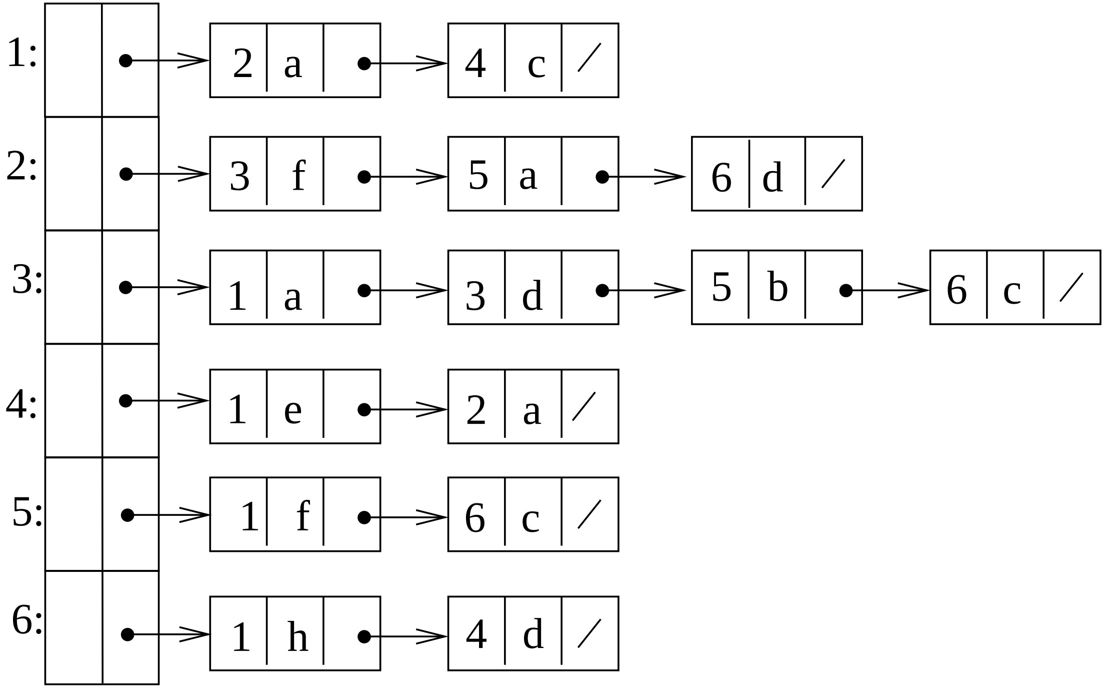
Example:

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

$\widehat{=}$



Adjacency lists:



Adjacency lists:

For each node i there is a list L_i, in which

- the *successors of* i (in digraphs) or
- the *neighbors of* i (in graphs)

are stored.

Realization: L_i is (singly or doubly linked) linear list, with its head pointer in `nodes`$[i]$, for $1 \leq i \leq n$.

Observations

(i) Length of L_i: outdeg(i) in digraphs, $\deg(i)$ in graphs.

(ii) In graphs we have: i occurs in $L_j \Leftrightarrow$ entry j occurs in L_i.

(iii) The neighbors/successors of a node i are **implicitly sorted** by their **order** in list L_i.

Extensions of adjacency list structure:

1) **Node labels**: place in `nodes` array.

2) **Edge labels**: in extra fields (attributes) in the list items of the adjacency list.

3) In **graphs**: List entry j in L_i can contain a pointer/reference to list entry for i in L_j (the **“reverse edge”**).

4) In **digraphs**: In the representation of the **reverse graph** G^{R} the adjacency list L_i^{R} for i for i contains the nodes j that are *predecessors* of i in G.

Exercise: Build representation of G^{R} from the representation of G in time $O(|V| + |E|)$.

Observation

If a graph or digraph $G = (V, E)$ is represented by adjacency lists, we have:

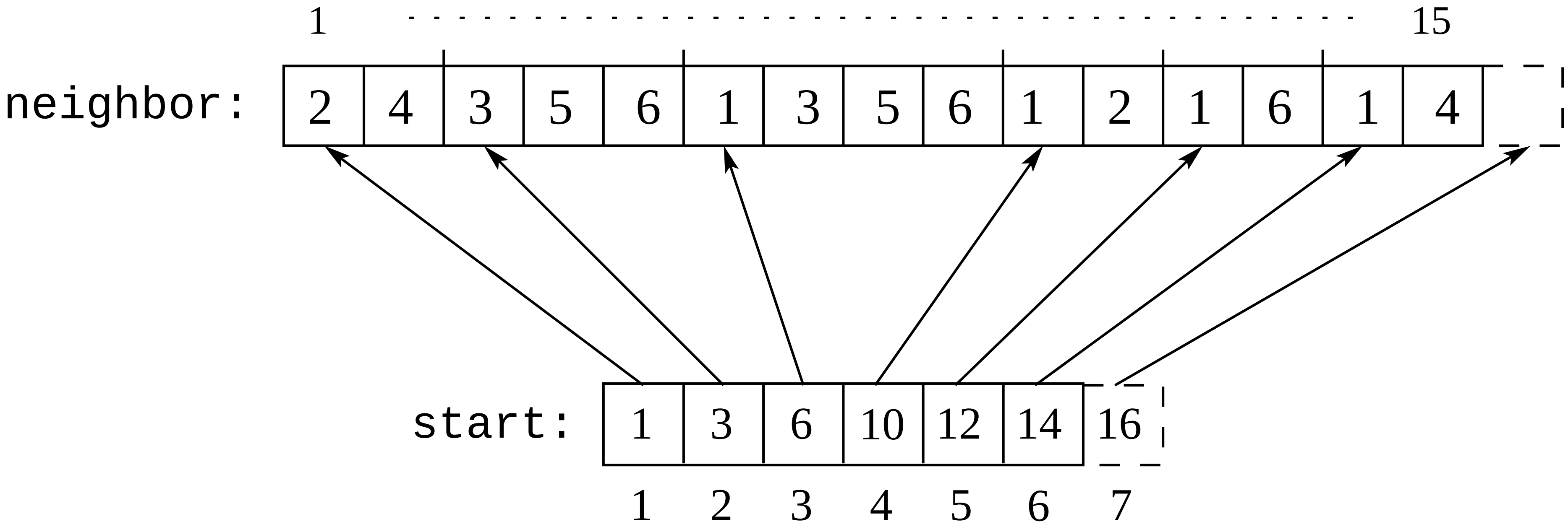
(a) space $O(|V| + |E|)$ is used;

(b) traversing *all* edges can be done in time $O(|V| + |E|)$;

(c) traversing the adjacency list for node i takes time $O(\mathrm{deg}(i))$ resp. $O(\mathrm{outdeg}(i))$;

(d) (only) if lists of predecessors (i. e. the representation of G^{R}) is given, we can also traverse the predecessors of node i in time $O(\mathrm{indeg}(i))$.

Adjacency array:



1 - 15

`neighbor:`	2	4	3	5	6	1	3	5	6	1	2	1	6	1	4	

`start:`	1	3	6	10	12	14	16
	1	2	3	4	5	6	7

Adjacency array representation:

Uses an array `neighbor`$[1..m]$.

In `neighbor` the successors/neighbors of each of $1,\ldots,n$ are listed, in this order.

More exactly: Let $s_i = 1 + \sum_{1\le j<i} \text{(out)deg}(j)$, for $1 \le i \le n+1$.

Then in `neighbor`$[s_i..s_{i+1}-1]$ we store the (indices of the) successors/neighbors of node i.

For navigating conveniently there is another array `start`$[1..n+1]$ with `start`$[i] = s_i$, for $1 \le i \le n+1$.
(This array can also be part of `nodes`$[1..n]$, which would then need an extra position $n+1$.)

The **adjacency array representation** is useful especially in cases where the graph/digraph does not change over time.

Advantages:

- saves storage space (no list pointers).
- Faster access to the names of the successors/neighbors.

Reason: When accessing a position in the adjacency array a whole block is copied into the cache.

Exercise: Describe a method that from the adjacency array representation of a digraph G constructs the adjacency array representation of the reverse graph G^{R}, in linear time.