Algorithms Design and Analysis

Day 4 Graphs

2015, AUT-CJLU

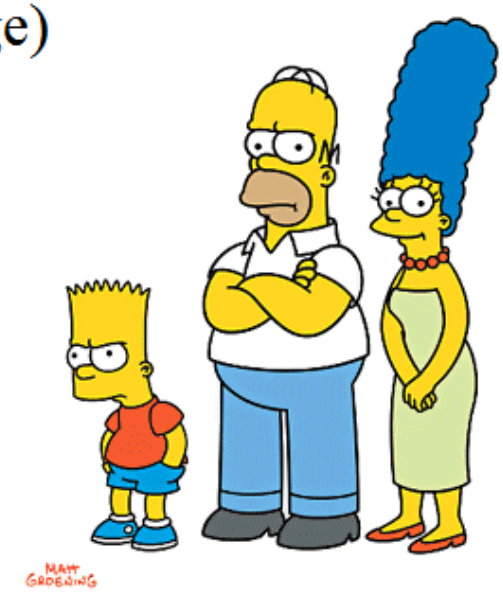
Part I: Basic Terminology

Relations

Relation. A logical or natural association between two or more things; relevance of one to another; connection.

For example:
Marriage Relation: If two persons marry each other, they become related. E.g. (Homer,Marge)

Parent Relation: If a person is the father or mother of another, they are also related.
E.g. (Homer,Bart)

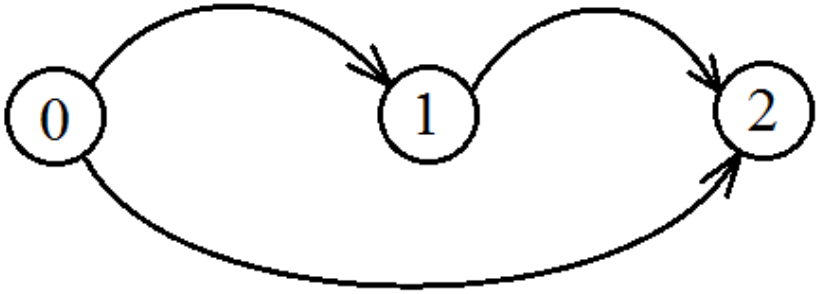


Relations

Relation. A logical or natural association between two or more things; relevance of one to another; connection.

For example:

LessThan (<): Two numbers x,y are in the LessThan relation if $x<y$.
e.g. (0,1), (0,2), (1,2)

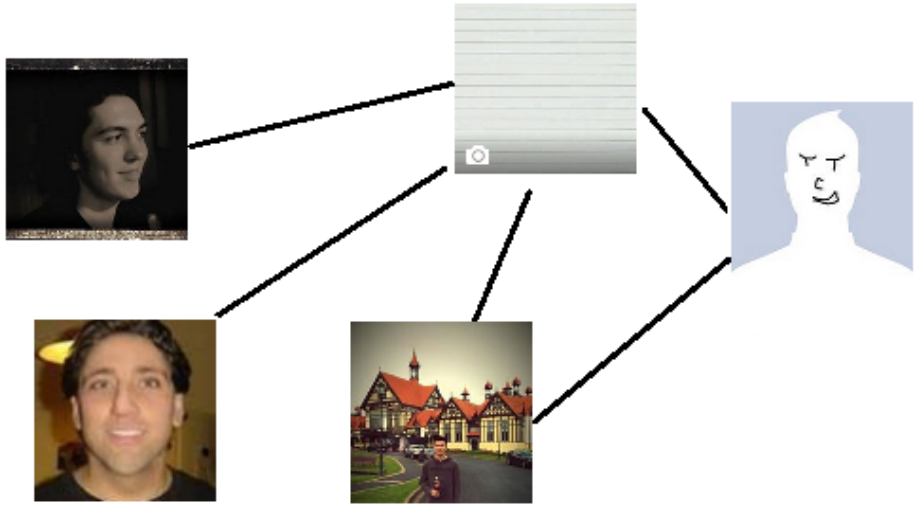


Relations

Relation. A logical or natural association between two or more things; relevance of one to another; connection.

For example:

Friends: Two people are in the friends relation if they are friends on Facebook



Relations

Binary Relations

- A domain is a set of objects (e.g. numbers, persons, etc.)
- A binary relation on a specific domain D is a collection of pairs (x, y) where $x \in D$ and $y \in D$

Example 1: Domain $D = \{Bart, Marge, Homer\}$
Parent = $\{(Homer, Bart), (Marge, Bart)\}$
Married = $\{(Homer, Marge)\}$

Example 2: Domain $D = \{0, 1, 2\}$
$<= \{(0, 1), (0, 2), (1, 2)\}$

Graphs

A graph is a data structure for representing binary relations. There are several different varieties of graphs:

- Directed graphs (digraphs)
- Undirected graphs
- Multi-graphs

Graphs

A graph is a data structure for representing binary relations. There are several different varieties of graphs:

- Directed graphs (digraphs)
- Undirected graphs
- Multi-graphs

Two key ingredients of a graph:

1. Nodes (or Vertices): representing elements in the domain
2. Edges: representing connections between pairs of nodes

Graphs: Definition

Digraphs

A directed graph (or digraph) G is (V, E) where

- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

Note: $(x, y) \neq (y, x)$

Graphs: Definition

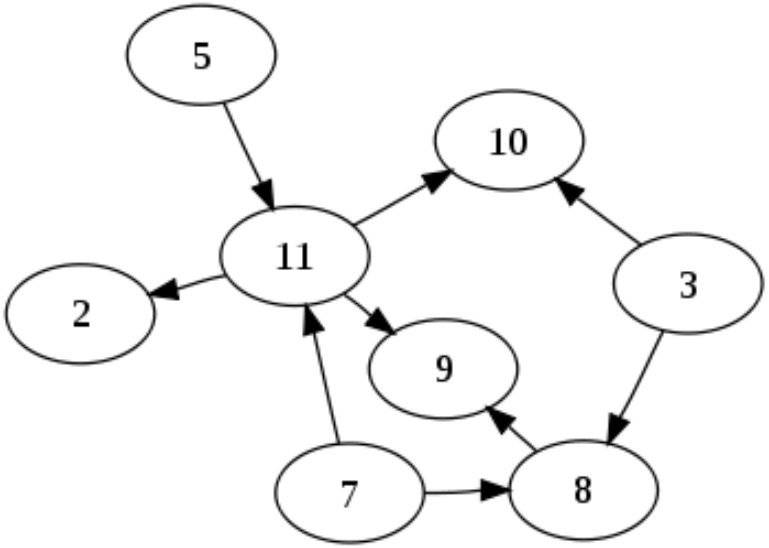
Digraphs

A directed graph (or digraph) G is (V, E) where

- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

Note: $(x, y) \neq (y, x)$

E.g. Nodes $V = \{2, 3, 5, 7, 8, 9, 10, 11\}$,
$E = \{(3, 8), (3, 10), (5, 11), (7, 8), (8, 9), (7, 11), (11, 2), (11, 10), (11, 9)\}$



Graphs: Definition

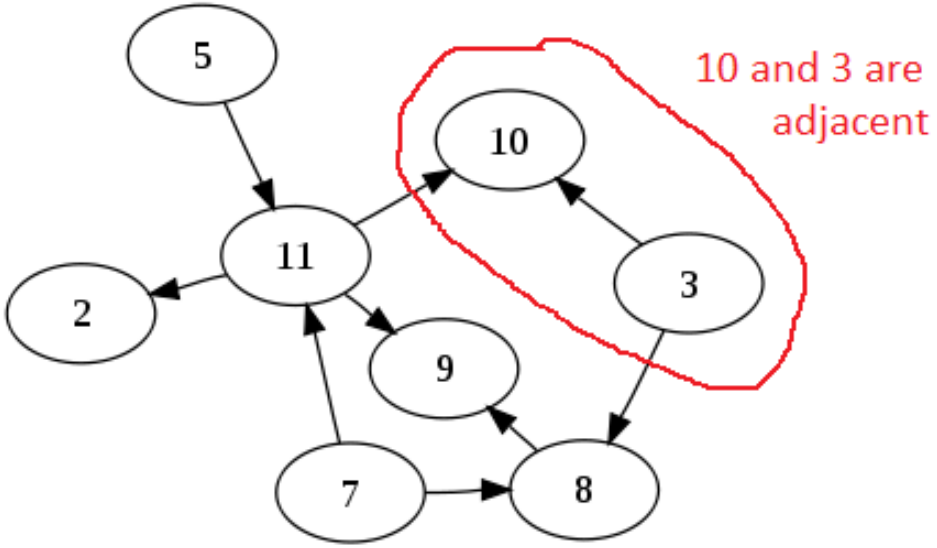
Digraphs

A directed graph (or digraph) G is (V, E) where

- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

Note: $(x, y) \neq (y, x)$

E.g. Nodes $V = \{2, 3, 5, 7, 8, 9, 10, 11\}$,
$E = \{(3, 8), (3, 10), (5, 11), (7, 8), (8, 9), (7, 11), (11, 2), (11, 10), (11, 9)\}$



Graphs: Definition

Digraphs

A directed graph (or digraph) G is (V, E) where

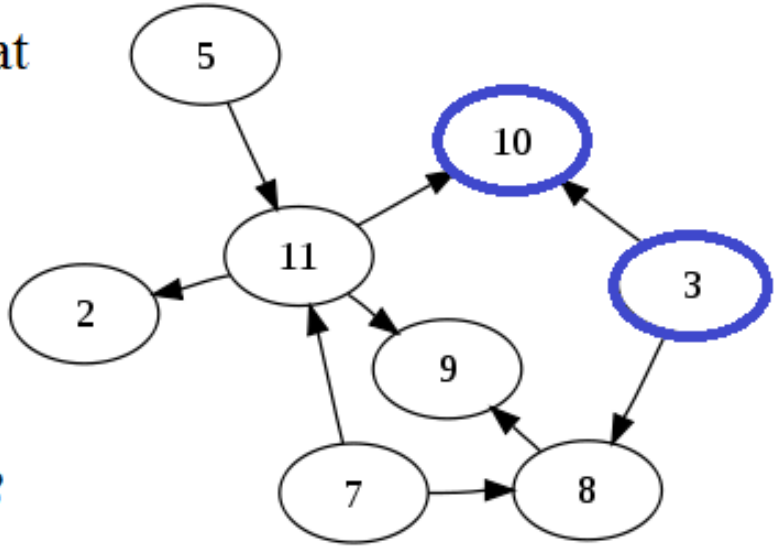
- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

Note: $(x, y) \neq (y, x)$

E.g. Nodes $V = \{2, 3, 5, 7, 8, 9, 10, 11\}$,
$E = \{(3, 8), (3, 10), (5, 11), (7, 8), (8, 9), (7, 11), (11, 2), (11, 10), (11, 9)\}$

If *(x,y)* is an edge, we say that
x,y are adjacent, and
x is an in-neighbour of *y*,
y is an out-neighbour of *x*

e.g. *(3,10)* is an edge;
- *3* is an in-neighbour of *10*
- *10* is an out-neighbour of *3*



Graphs: Definition

Digraphs

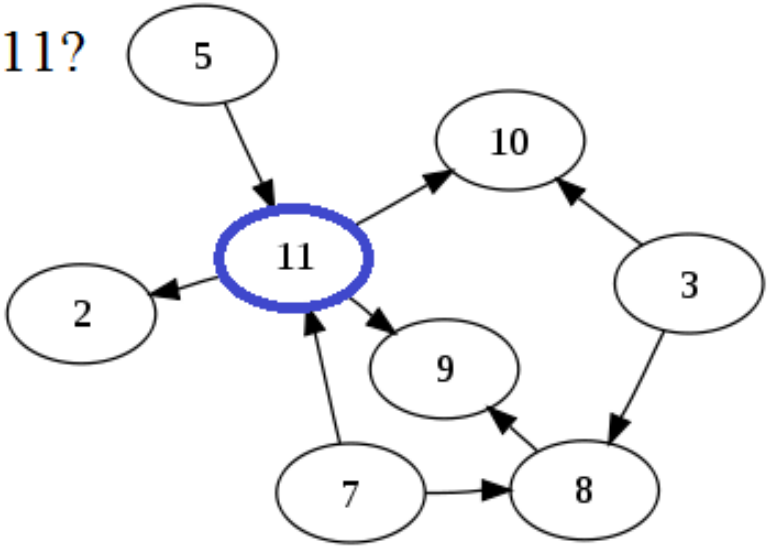
A directed graph (or digraph) G is (V, E) where

- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

Note: $(x, y) \neq (y, x)$

E.g. Nodes $V = \{2, 3, 5, 7, 8, 9, 10, 11\}$,
$E = \{(3, 8), (3, 10), (5, 11), (7, 8), (8, 9), (7, 11), (11, 2), (11, 10), (11, 9)\}$

What are the in-neighbours of 11?



Graphs: Definition

Digraphs

A directed graph (or digraph) G is (V, E) where

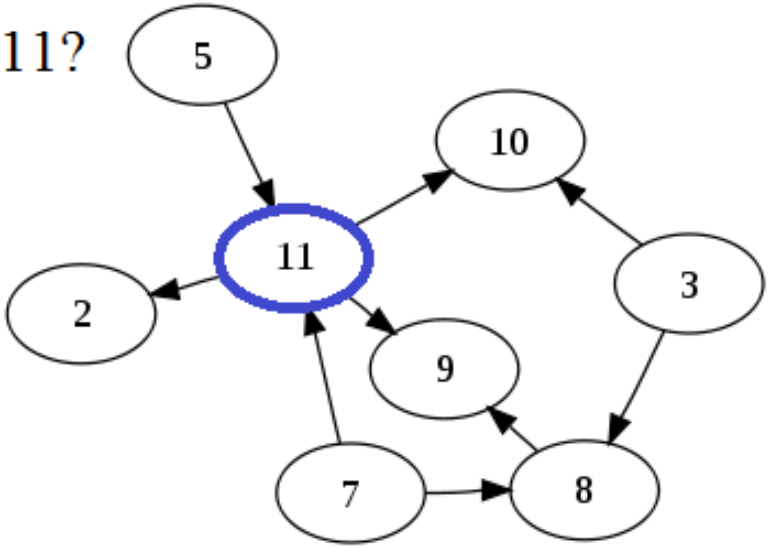
- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

Note: $(x, y) \neq (y, x)$

E.g. Nodes $V = \{2, 3, 5, 7, 8, 9, 10, 11\}$,
$E = \{(3, 8), (3, 10), (5, 11), (7, 8), (8, 9), (7, 11), (11, 2), (11, 10), (11, 9)\}$

What are the in-neighbours of 11?

- 5 , 7



Graphs: Definition

Digraphs

A directed graph (or digraph) G is (V, E) where

- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

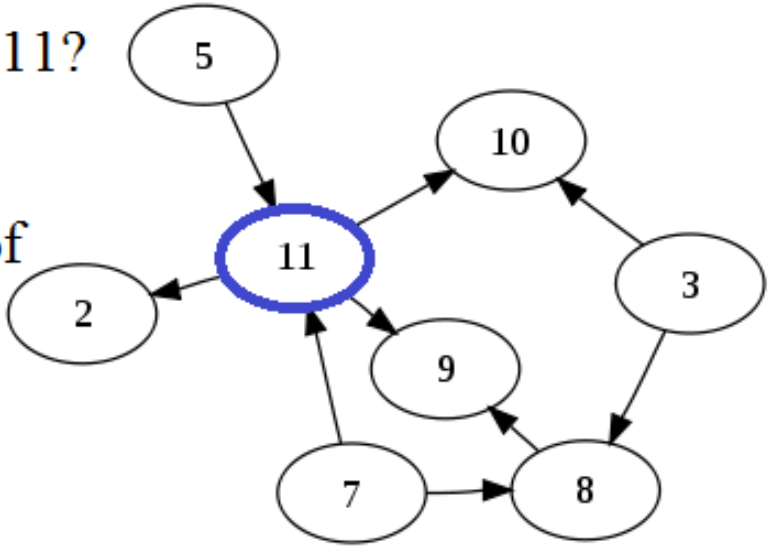
Note: $(x, y) \neq (y, x)$

E.g. Nodes $V = \{2, 3, 5, 7, 8, 9, 10, 11\}$,
$E = \{(3, 8), (3, 10), (5, 11), (7, 8), (8, 9), (7, 11), (11, 2), (11, 10), (11, 9)\}$

What are the in-neighbours of 11?

- 5 , 7

What are the out-neighbours of 11?



Graphs: Definition

Digraphs

A directed graph (or digraph) G is (V, E) where

- V is a finite nonempty set V of nodes (or vertices);
- E is a set of directed edges of the form $(x, y) \in V^2$ where $x \neq y$.

Note: $(x, y) \neq (y, x)$

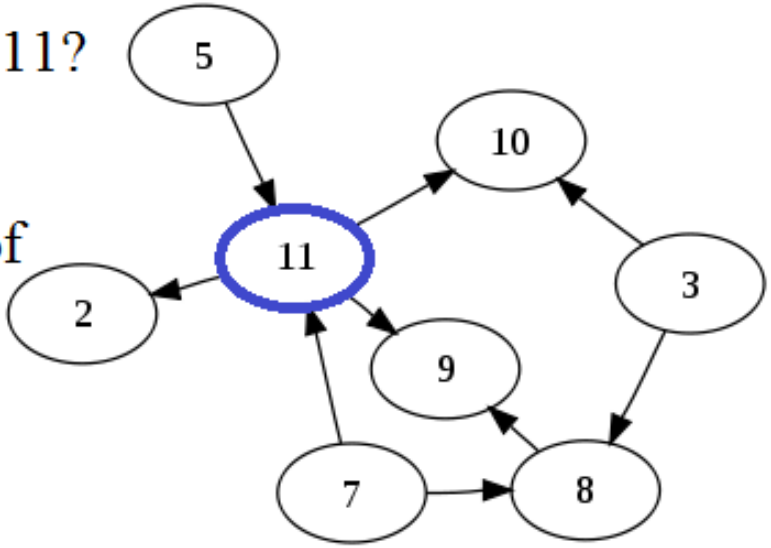
E.g. Nodes $V = \{2, 3, 5, 7, 8, 9, 10, 11\}$,
$E = \{(3, 8), (3, 10), (5, 11), (7, 8), (8, 9), (7, 11), (11, 2), (11, 10), (11, 9)\}$

What are the in-neighbours of 11?

- 5 , 7

What are the out-neighbours of 11?

- 2, 9, 10



Graphs: Definition

Undirected Graphs

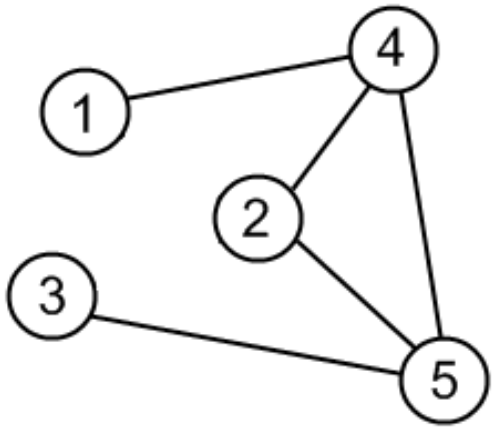
- A binary relation E is symmetric if $(x, y) \in E$ whenever $(y, x) \in E$ (i.e. all connections are two-way).
- A undirected graph (or graph) G is just like a digraph except that E is symmetric, i.e., each edge in E is undirected, denoted by $\{u, v\}$.

Graphs: Definition

Undirected Graphs

- A binary relation E is symmetric if $(x, y) \in E$ whenever $(y, x) \in E$ (i.e. all connections are two-way).
- A undirected graph (or graph) G is just like a digraph except that E is symmetric, i.e., each edge in E is undirected, denoted by $\{u, v\}$.

E.g. Vertices $V = \{1, 2, 3, 4, 5\}$,
Edges $E = \{\{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 5\}\}$



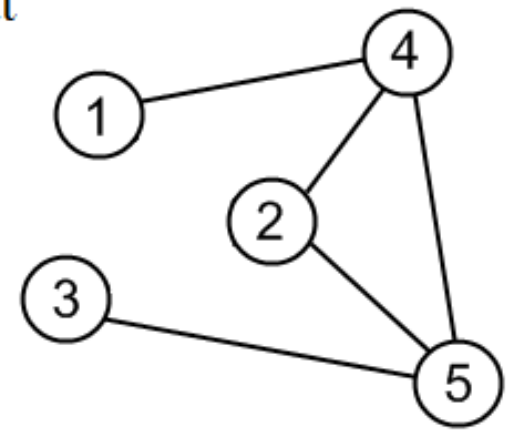
Graphs: Definition

Undirected Graphs

- A binary relation E is symmetric if $(x, y) \in E$ whenever $(y, x) \in E$ (i.e. all connections are two-way).
- A undirected graph (or graph) G is just like a digraph except that E is symmetric, i.e., each edge in E is undirected, denoted by $\{u, v\}$.

E.g. Vertices $V = \{1, 2, 3, 4, 5\}$,
Edges $E = \{\{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 5\}\}$

If $\{x,y\}$ is an edge, we say that x and y are neighbours of each other.



Graphs: Definition

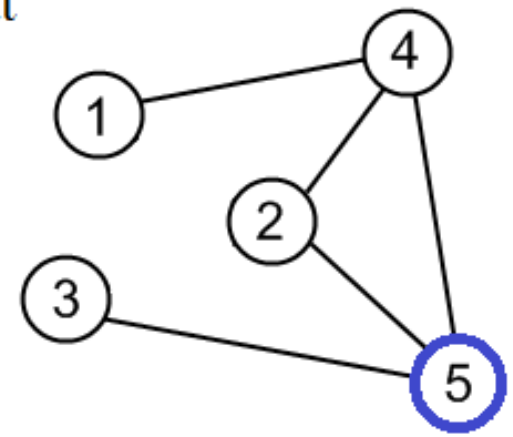
Undirected Graphs

- A binary relation E is symmetric if $(x, y) \in E$ whenever $(y, x) \in E$ (i.e. all connections are two-way).
- A undirected graph (or graph) G is just like a digraph except that E is symmetric, i.e., each edge in E is undirected, denoted by $\{u, v\}$.

E.g. Vertices $V = \{1, 2, 3, 4, 5\}$,
Edges $E = \{\{1, 4\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{4, 5\}\}$

If $\{x,y\}$ is an edge, we say that x and y are neighbours of each other.

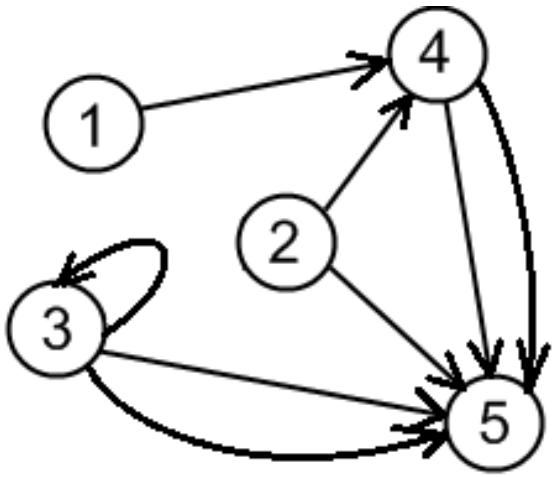
e.g. Neighbours of 5:
- 2,3,4



Variants of Graphs

We may enrich the notion of graphs in the following ways:

- Multigraphs: $G = (V, E)$ where $E \subseteq V^2$ is a multiset, i.e., a set where each element can appear more than once.
- Weighted (di)graphs: $G = (V, E, f, D)$ where (V, E) is a (di)graph and $f : E \to D$ is a weight function.



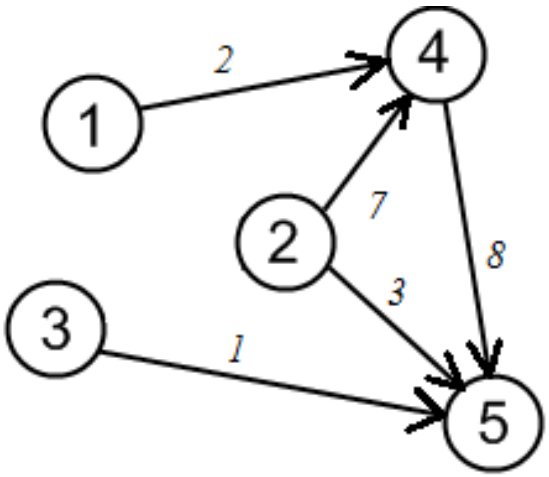
$V=\{1,2,3,4,5\}$

$E=\{(1,4),(2,4),(4,5)$
$(4,5),(2,5),(3,5)$
$(3,5),(3,3)\}$

Variants of Graphs

We may enrich the notion of graphs in the following ways:

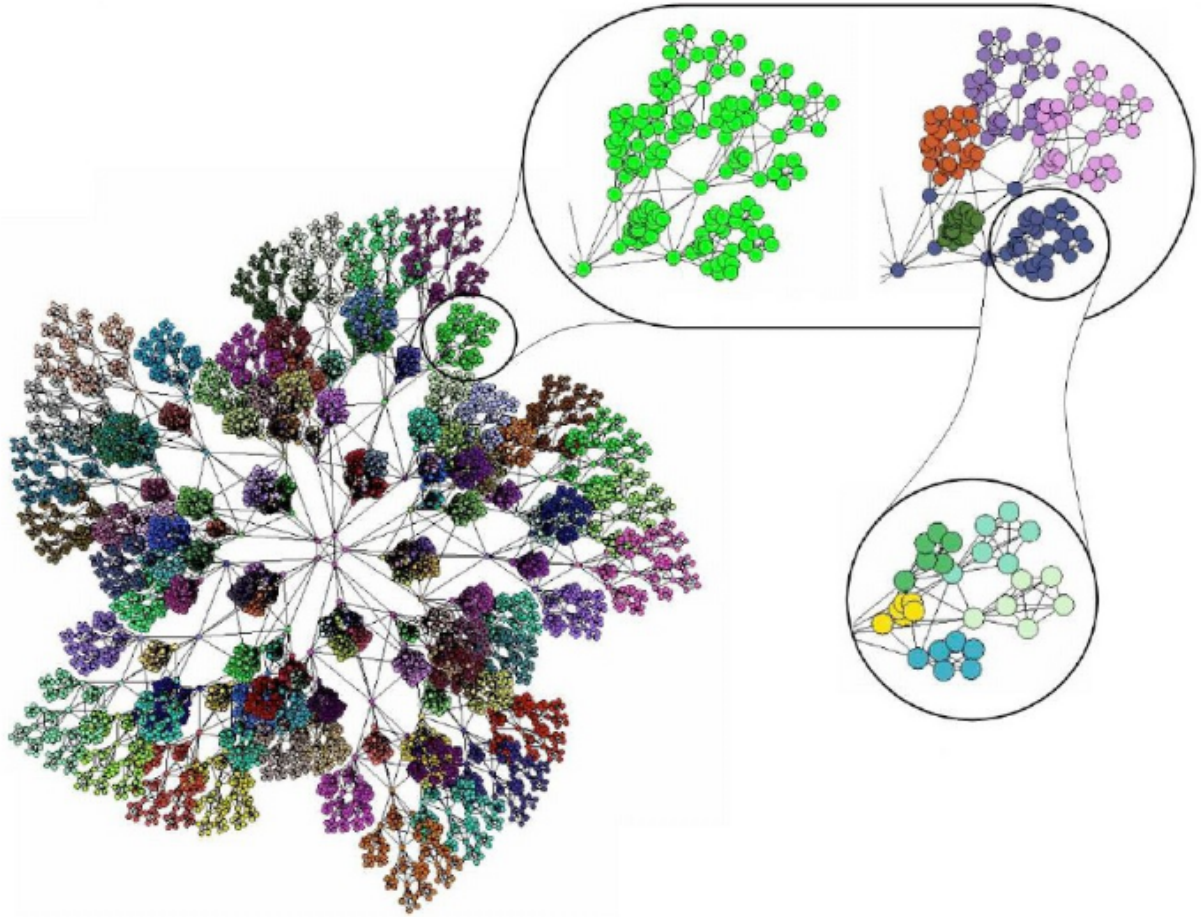
- Multigraphs: $G = (V, E)$ where $E \subseteq V^2$ is a multiset, i.e., a set where each element can appear more than once.
- Weighted (di)graphs: $G = (V, E, f, D)$ where (V, E) is a (di)graph and $f : E \to D$ is a weight function.



$V=\{1,2,3,4,5\}$
$E=\{ (1,4), (2,4), (3,5), (4,5), (2,5) \}$
$f:E \to \mathbb{N}$ where
$f(1,4)=2, f(2,4)=7, f(4,5)=8$
$f(2,5)=3, f(3,5)=1$

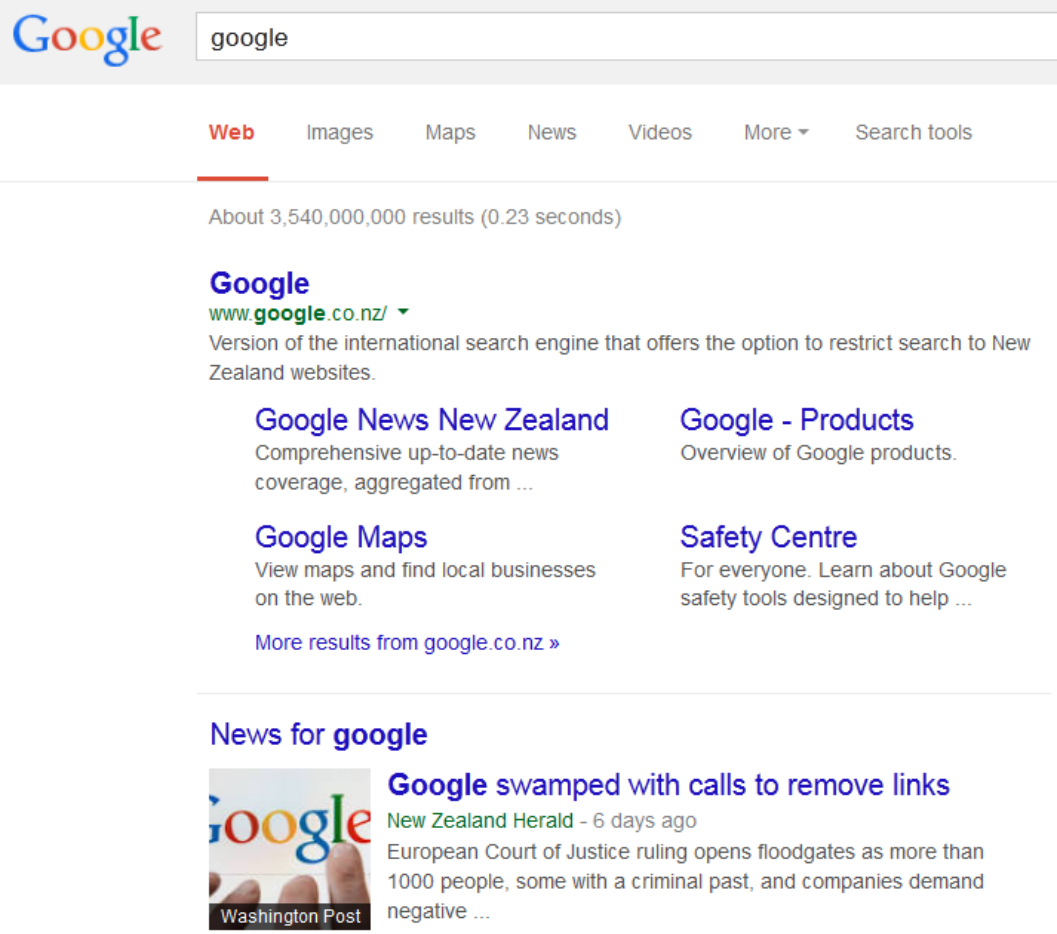
Graphs: Applications

Graphs: Models of social networks
Goal: Detecting communities in the social network



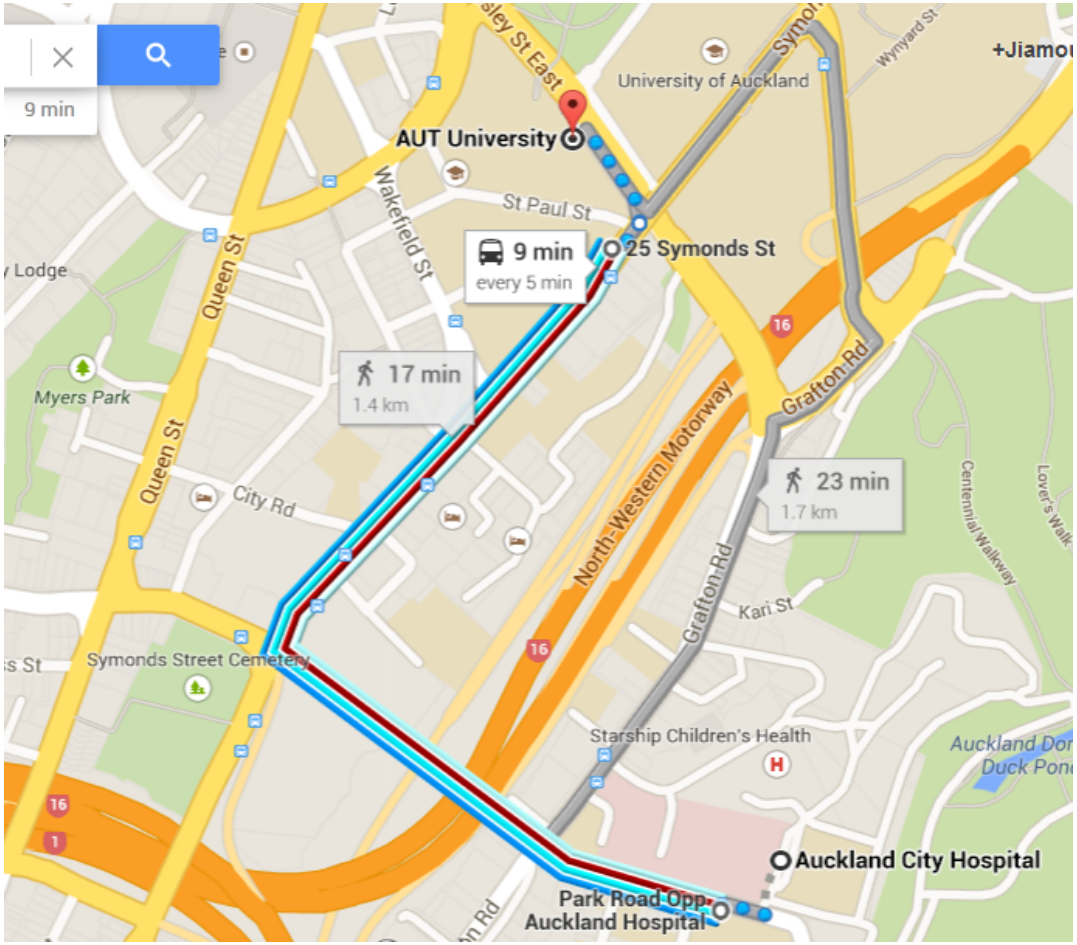
Graphs: Applications

Graphs: Models of the Internet
Goal: Ranking hyperlinks for fast web search



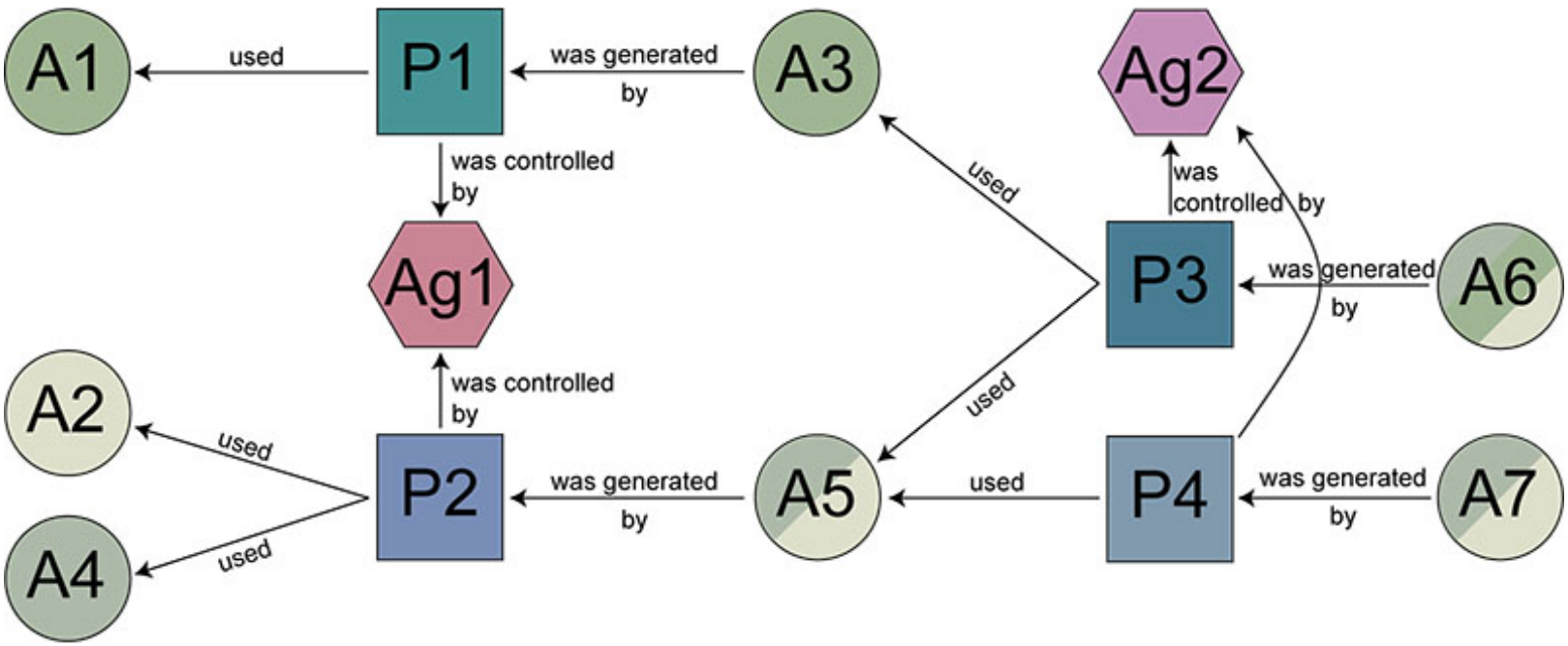
Graphs: Applications

Graphs: Models of Road Networks
Goal: Finding shortest routes



Graphs: Applications

Provenance Graphs: Capturing information and process dependency in scientific workflow
Goal: Validating and debugging workflow results



Graphs: Terminology

Basic Terminology

- The order of a digraph is the number of nodes in the digraph (often denoted by n).
- The size of a digraph is the number of directed edges in the digraph (often denoted by m).
- The size of a graph is the number of undirected edges (often denoted by m).

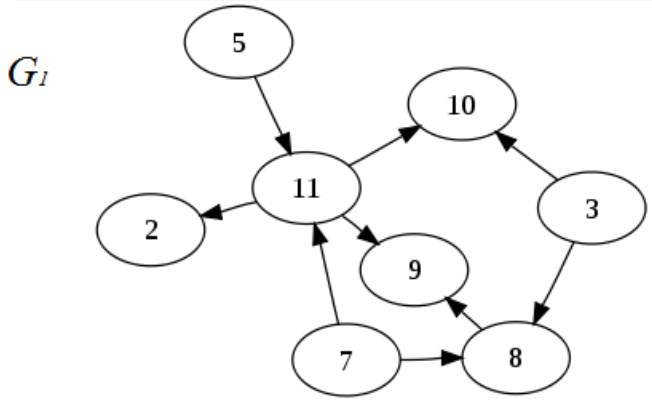
Complexity of algorithms performed on digraphs and graphs are often stated in terms of n and m.

Graphs: Terminology

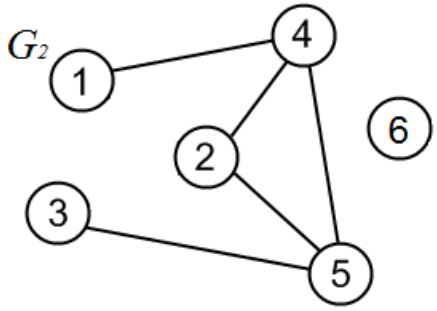
Basic Terminology

- The order of a digraph is the number of nodes in the digraph (often denoted by n).
- The size of a digraph is the number of directed edges in the digraph (often denoted by m).
- The size of a graph is the number of undirected edges (often denoted by m).

Complexity of algorithms performed on digraphs and graphs are often stated in terms of n and m.



Order: 8 Size: 9

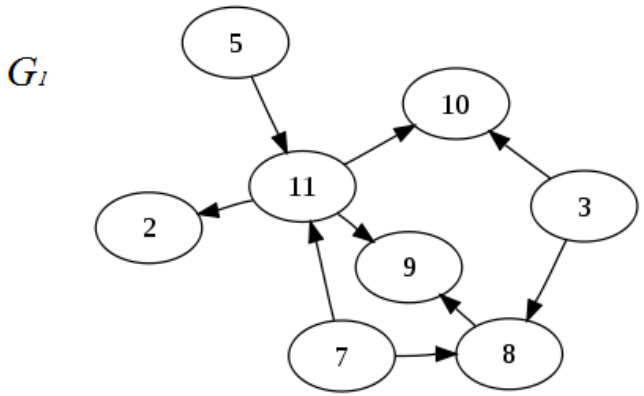


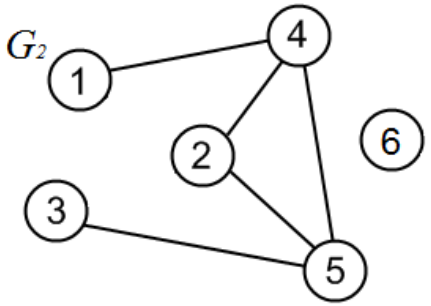
Order: 6 Size: 5

Graphs: Terminology

Basic Terminology

- The out-degree of a node u in a digraph is the number of out-neighbours of u.
- The in-degree of a node u in a digraph is the number of in-neighbours of u.
- The degree of a node u in a graph is the number of neighbours of u.



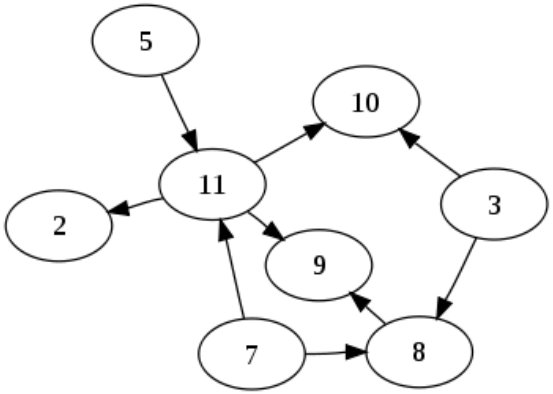


Graphs: Terminology

Basic Terminology

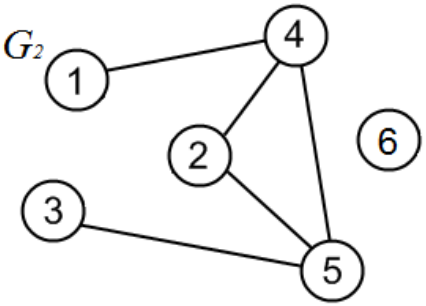
- The out-degree of a node u in a digraph is the number of out-neighbours of u.
- The in-degree of a node u in a digraph is the number of in-neighbours of u.
- The degree of a node u in a graph is the number of neighbours of u.

G_1



in-degree(11) =
out-degree(11) =
in-degree(7) =
out-degree(10) =

G_2



degree(2) =

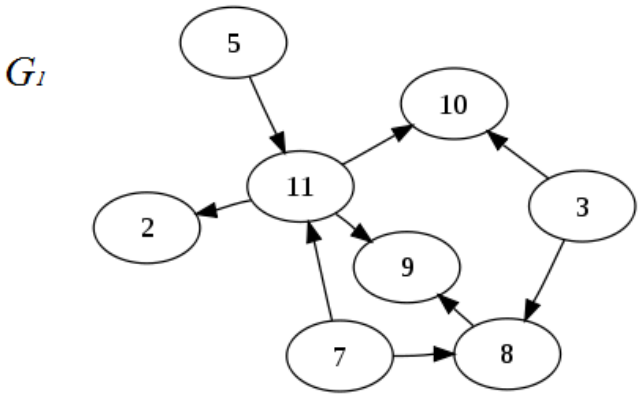
degree(3) =

degree(6) =

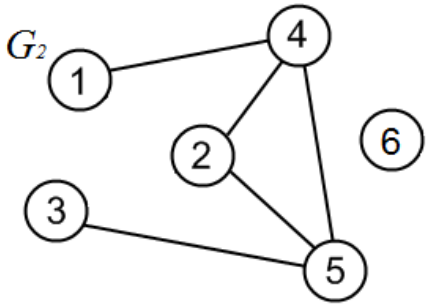
Graphs: Terminology

Basic Terminology

- The out-degree of a node u in a digraph is the number of out-neighbours of u.
- The in-degree of a node u in a digraph is the number of in-neighbours of u.
- The degree of a node u in a graph is the number of neighbours of u.



in-degree(11) = 2
out-degree(11) = 3
in-degree(7) = 0
out-degree(10) = 0



degree(2) =

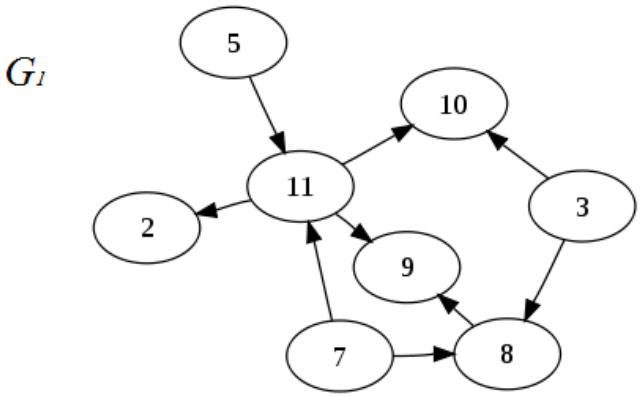
degree(3) =

degree(6) =

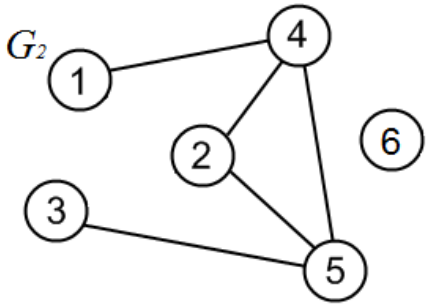
Graphs: Terminology

Basic Terminology

- The out-degree of a node u in a digraph is the number of out-neighbours of u.
- The in-degree of a node u in a digraph is the number of in-neighbours of u.
- The degree of a node u in a graph is the number of neighbours of u.



in-degree(11) = 2
out-degree(11) = 3
in-degree(7) = 0
out-degree(10) = 0



degree(2) = 2

degree(3) = 1

degree(6) = 0

Graphs: Terminology

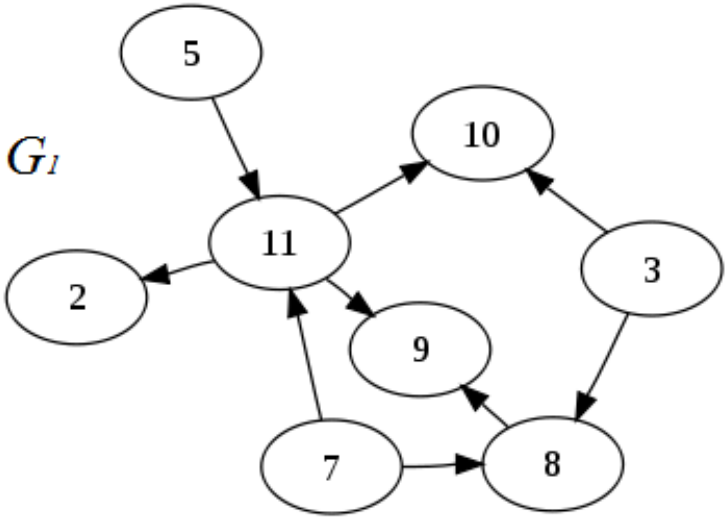
Basic Terminology

The underlying graph of a digraph is obtained by removing all directions from any directed edges.
i.e. For digraph $G = (V, E)$, the underlying graph of G is an undirected graph (V, E') where $\{u, v\} \in E'$ if and only if $(u, v) \in E$ or $(v, u) \in E$.

Graphs: Terminology

Basic Terminology

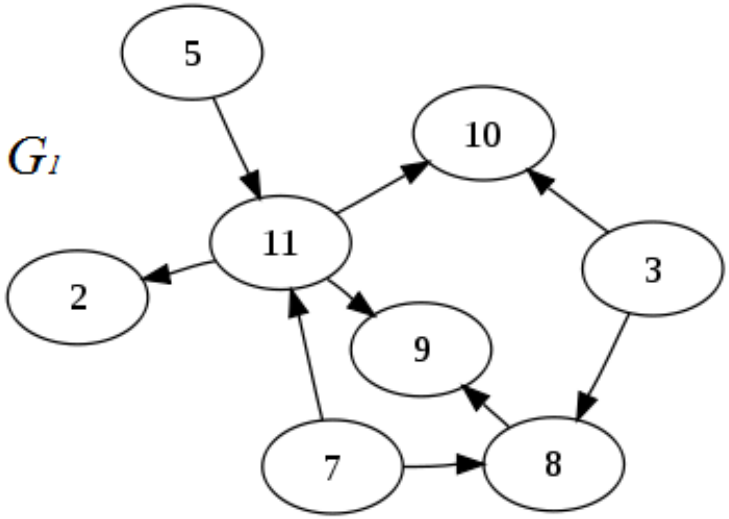
The underlying graph of a digraph is obtained by removing all directions from any directed edges.
i.e. For digraph $G = (V, E)$, the underlying graph of G is an undirected graph (V, E') where $\{u, v\} \in E'$ if and only if $(u, v) \in E$ or $(v, u) \in E$.

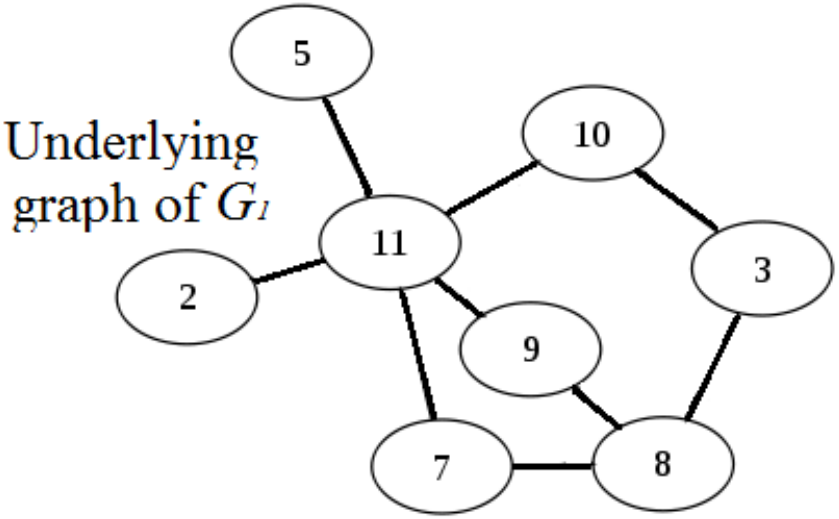


Graphs: Terminology

Basic Terminology

The underlying graph of a digraph is obtained by removing all directions from any directed edges.
i.e. For digraph $G = (V, E)$, the underlying graph of G is an undirected graph (V, E') where $\{u, v\} \in E'$ if and only if $(u, v) \in E$ or $(v, u) \in E$.

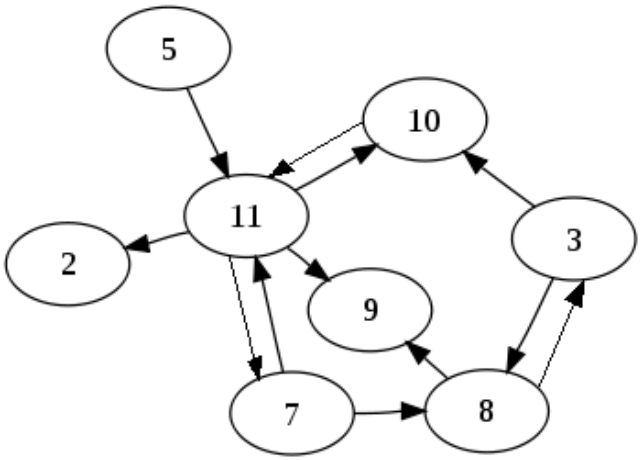




Graphs: Terminology

Paths

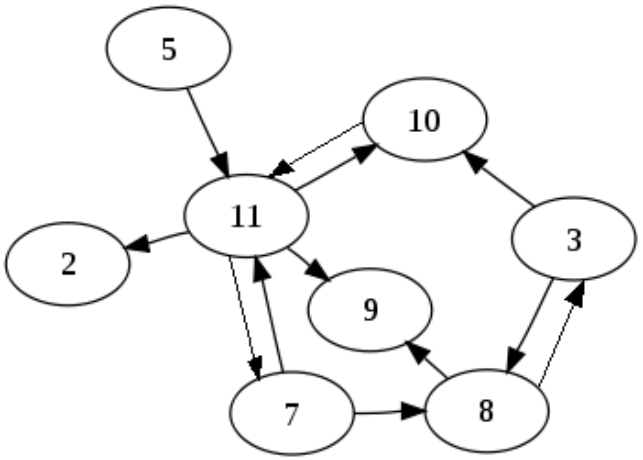
- A path (or walk) in a digraph $G = (V, E)$ is a sequence of nodes $v_0, v_1, \ldots, v_k$ such that $(v_0, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ are all directed edges in G.
- A simple path is a walk in which no node is repeated.



Graphs: Terminology

Paths

- A path (or walk) in a digraph $G = (V, E)$ is a sequence of nodes $v_0, v_1, \ldots, v_k$ such that $(v_0, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ are all directed edges in G.
- A simple path is a walk in which no node is repeated.

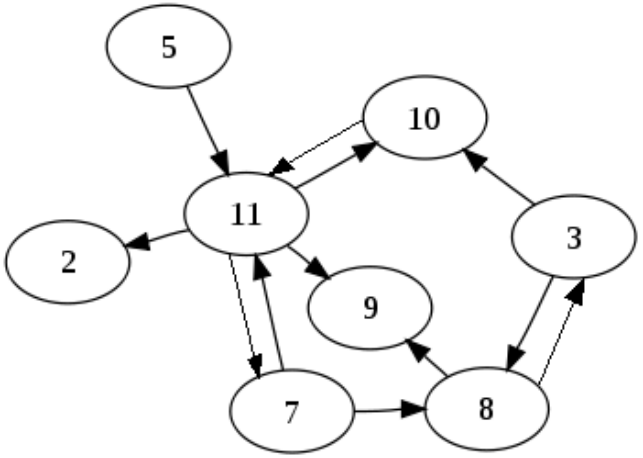


- A path: $(5, 11), (11, 7), (7, 8), (8, 3), (3, 8), (8, 9)$
  (or simply $5, 11, 7, 8, 3, 8, 9$)
- A simple path: $(5, 11), (11, 7), (7, 8), (8, 3)$
  (or simply $5, 11, 7, 8, 3$)

Graphs: Terminology

Cycles

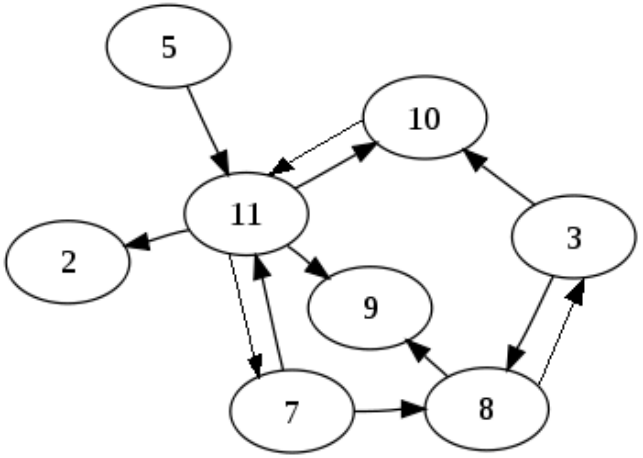
- A cycle (or circuit) is a path in which $v_0 = v_k$ and no other nodes are repeated.
- The length of a path is the number of edges ("steps") in it.
- The distance from a node u to v, $dist(u, v)$, is the length of the shortest path from u to v. If no path exist, then $dist(u, v) = \infty$.



Graphs: Terminology

Cycles

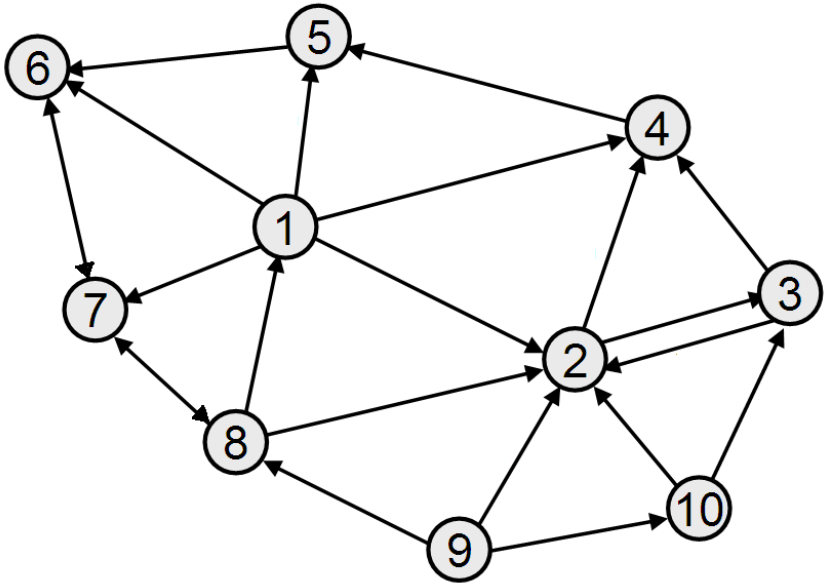
- A cycle (or circuit) is a path in which $v_0 = v_k$ and no other nodes are repeated.
- The length of a path is the number of edges ("steps") in it.
- The distance from a node u to v, $dist(u, v)$, is the length of the shortest path from u to v. If no path exist, then $dist(u, v) = \infty$.



- A cycle: $11, 7, 8, 3, 10, 11$
- Length of this cycle: 5
- Distance between 11 and 3: 3

Graphs: Terminology

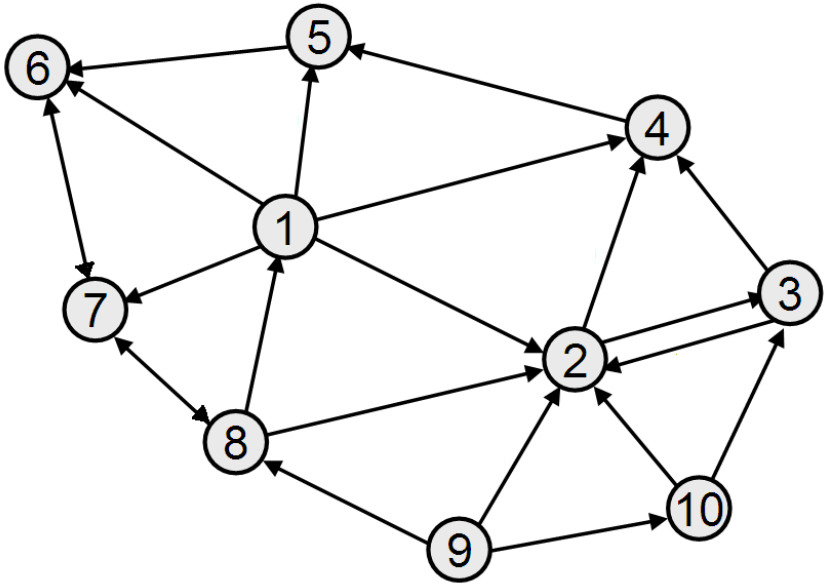
Example:



Pick a path between 7 and 8 of length 5:

Graphs: Terminology

Example:

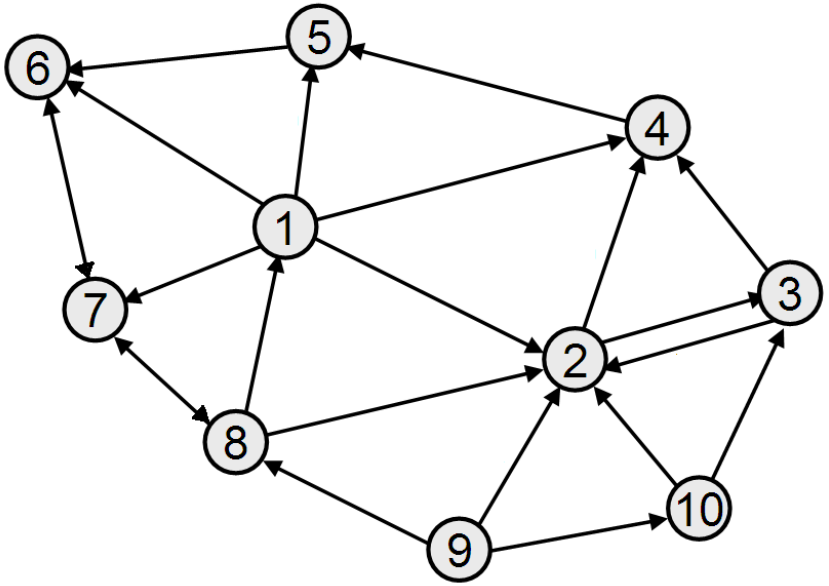


Pick a path between 7 and 8 of length 5:

7,8,1,6,7,8

Graphs: Terminology

Example:



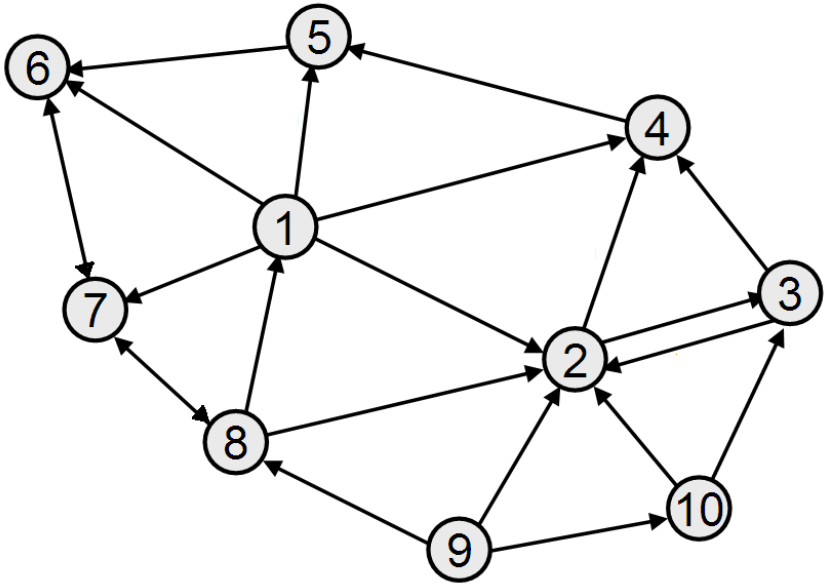
Pick a path between 7 and 8 of length 5:

7,8,1,6,7,8

Pick a simple path between 2 and 7:

Graphs: Terminology

Example:



Pick a path between 7 and 8 of length 5:

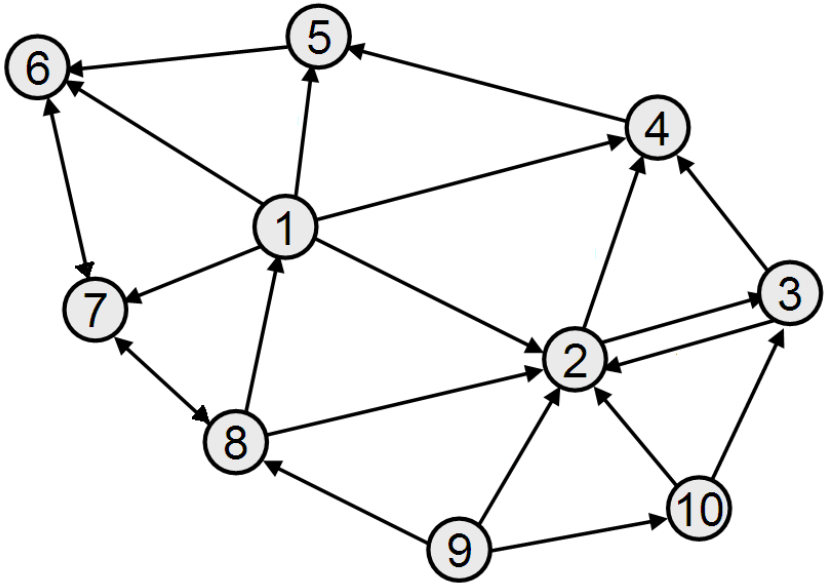
7,8,1,6,7,8

Pick a simple path between 2 and 7:

2,4,5,6,7

Graphs: Terminology

Example:



Pick a path between 7 and 8 of length 5:

7,8,1,6,7,8

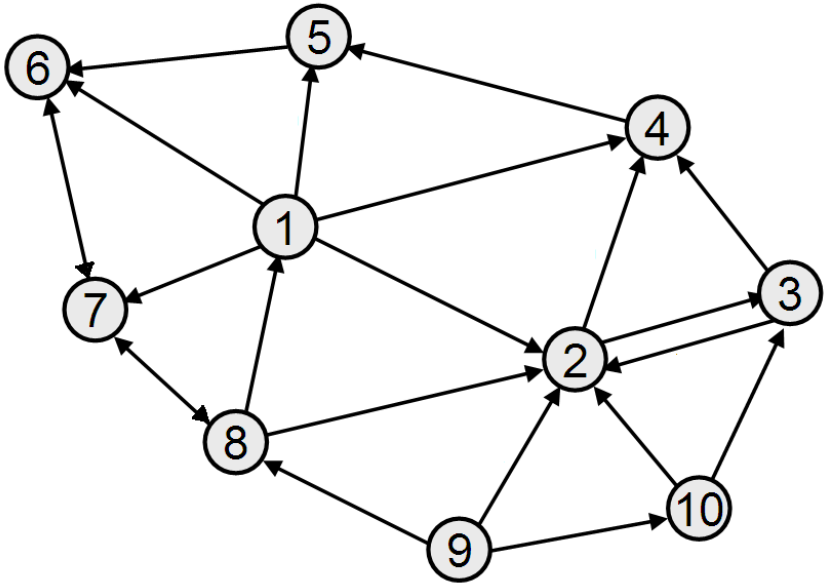
Pick a simple path between 2 and 7:

2,4,5,6,7

Pick a cycle containing 6:

Graphs: Terminology

Example:



Pick a path between 7 and 8 of length 5:

7,8,1,6,7,8

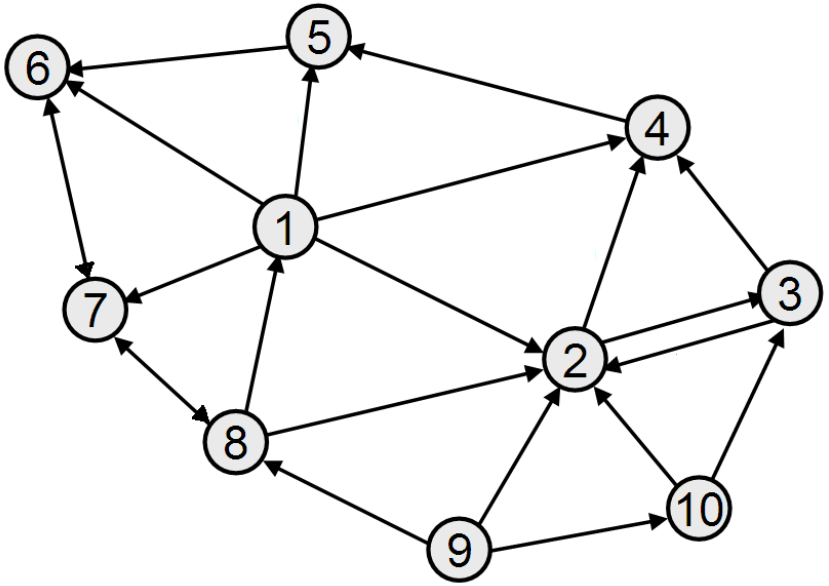
Pick a simple path between 2 and 7:

2,4,5,6,7

Pick a cycle containing 6: 6,7,6

Graphs: Terminology

Example:



Pick a path between 7 and 8 of length 5:

7,8,1,6,7,8

Pick a simple path between 2 and 7:

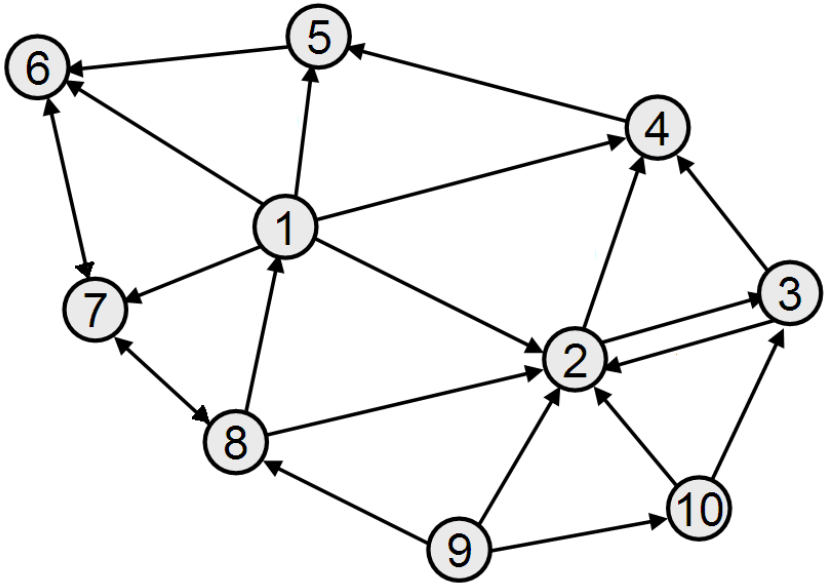
2,4,5,6,7

Pick a cycle containing 6: 6,7,6

The distance between 2 and 7:

Graphs: Terminology

Example:



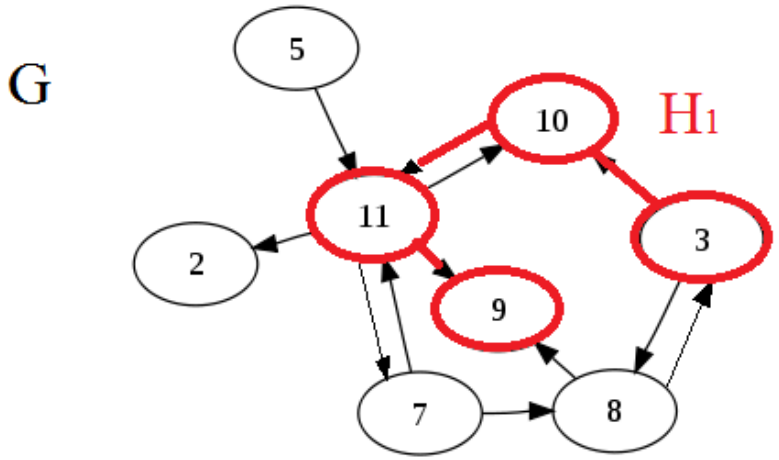
Pick a path between 7 and 8 of length 5:
7,8,1,6,7,8
Pick a simple path between 2 and 7:
2,4,5,6,7
Pick a cycle containing 6: 6,7,6
The distance between 2 and 7: 4

Graphs: Terminology

Let $G = (V, E)$ and $U \subseteq V$. We set $E \restriction U$ as $\{(u, v) \in E \mid u, v \in U\}$.

Subgraphs

- A subgraph of a digraph $G = (V, E)$ is a digraph $H = (U, F)$ where $U \subseteq V$ and $F \subseteq E \restriction U$.
- If $F = E \restriction U$, then H is an induced subgraph of G.

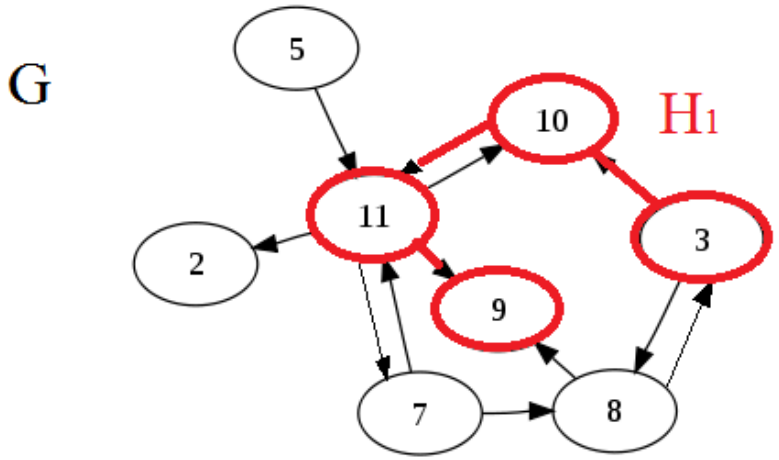


Graphs: Terminology

Let $G = (V, E)$ and $U \subseteq V$. We set $E \restriction U$ as $\{(u, v) \in E \mid u, v \in U\}$.

Subgraphs

- A subgraph of a digraph $G = (V, E)$ is a digraph $H = (U, F)$ where $U \subseteq V$ and $F \subseteq E \restriction U$.
- If $F = E \restriction U$, then H is an induced subgraph of G.



E.g. $U = \{3, 9, 10, 11\}$.
$E \restriction U = \{(10, 11), (11, 10), (3, 10), (11, 9)\}$
A subgraph of G:

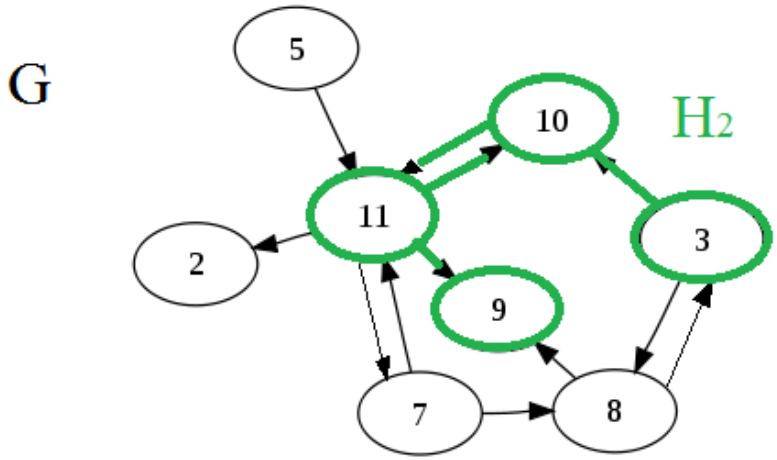
$$H_1 = (\{3, 9, 10, 11\}, \{(3, 10), (10, 11), (11, 9)\})$$

Graphs: Terminology

Let $G = (V, E)$ and $U \subseteq V$. We set $E \restriction U$ as $\{(u, v) \in E \mid u, v \in U\}$.

Subgraphs

- A subgraph of a digraph $G = (V, E)$ is a digraph $H = (U, F)$ where $U \subseteq V$ and $F \subseteq E \restriction U$.
- If $F = E \restriction U$, then H is an induced subgraph of G.



E.g. $U = \{3, 9, 10, 11\}$.
$E \restriction U = \{(10, 11), (11, 10), (3, 10), (11, 9)\}$
An induced subgraph of G:

$$H_2 = (\{3, 9, 10, 11\}, \{(3, 10), (10, 11), (11, 9), (11, 10)\})$$

Graphs: Example

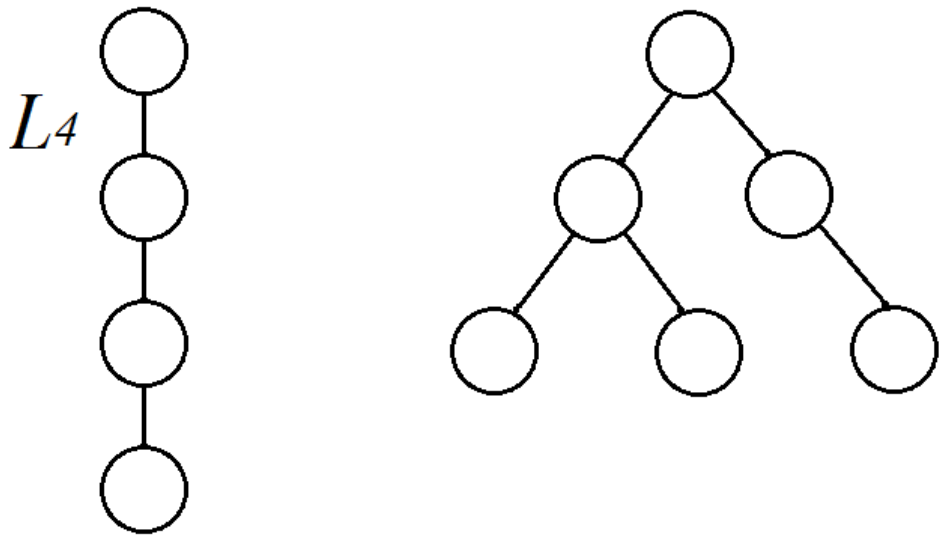
Important Classes of Graphs

- Line graphs: Graphs that look like link lists.

  $L_n = (\{1, 2, \ldots, n\}, \{\{1, 2\}, \{2, 3\}, \ldots, \{n-1, n\}\})$

- Tree graphs: Graphs that look like trees.

  A tree graph is a undirected graph without any cycle of length ≥ 3.

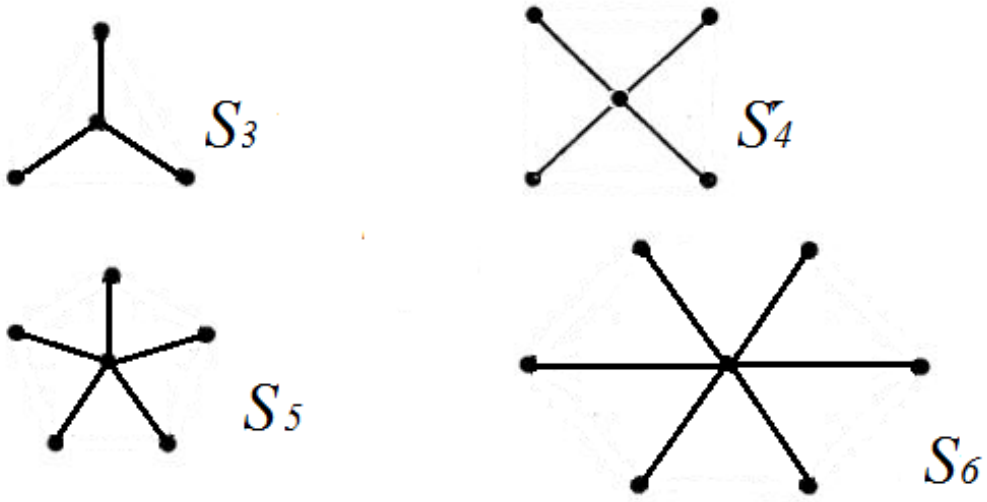


Graphs: Example

Important Classes of Graphs

Star graphs: $S_n = (V, E)$ where

- $V = \{c, 1, 2, \ldots, n\}$
- $E = \{\{1, c\}, \{2, c\}, \ldots, \{n, c\}\}$

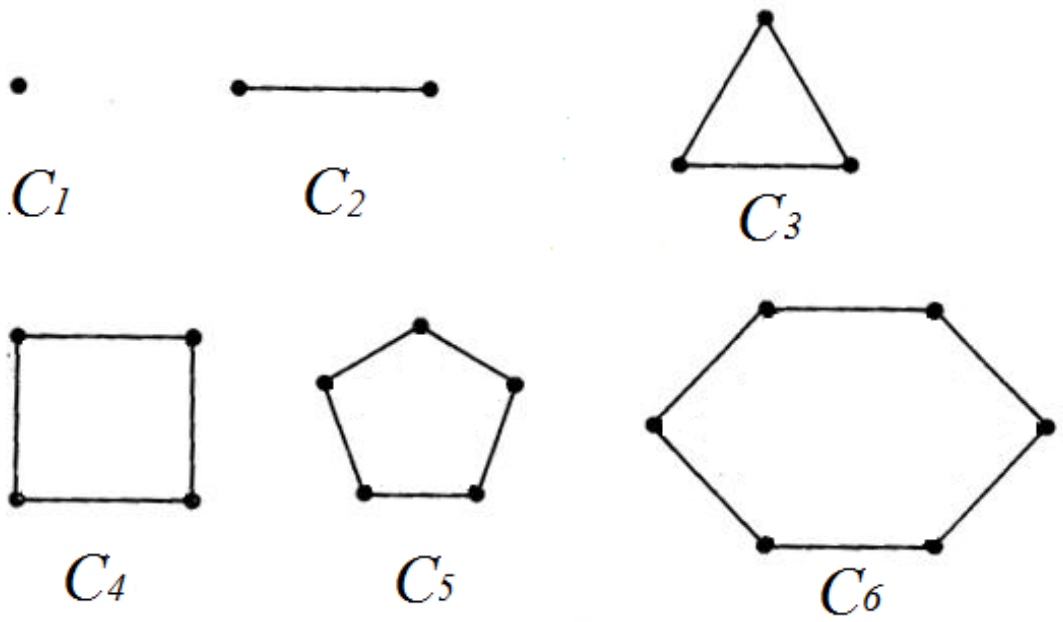


Graphs: Example

Important Classes of Graphs

Cycle graphs: A graph consisting of only a cycle
$C_n = (V, E)$ where

- $V = \{1, 2, 3, \ldots, n\}$, and
- $E = \{\{1, 2\}, \{2, 3\}, \ldots, \{n-1, n\}, \{n, 1\}\}$

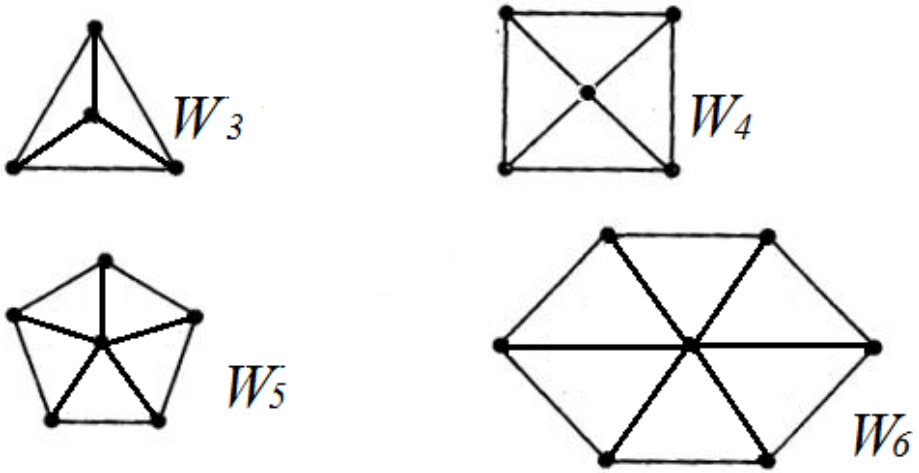


Graphs: Example

Important Classes of Graphs

Wheel graphs: A graph that looks like a wheel
$W_n = (V, E)$ where

- $V = \{c, 1, 2, 3, \ldots, n\}$, and
- $E = \{\{1,2\},\{2,3\},\ldots,\{n-1,n\},\{n,1\}\} \cup \{\{1,c\},\{2,c\},\ldots,\{n,c\}\}$

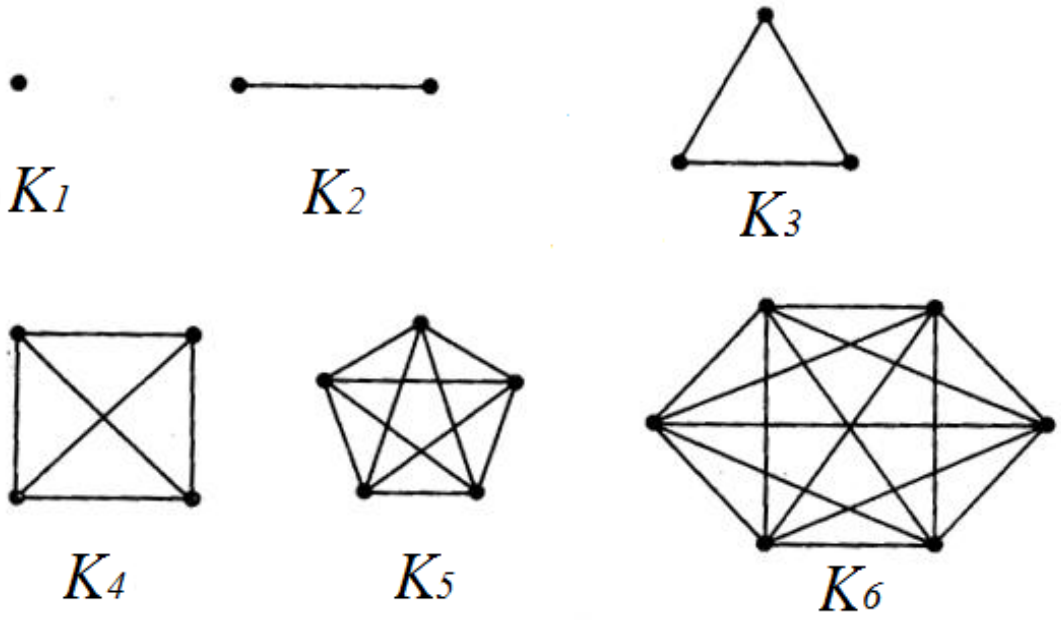


Graphs: Example

Important Classes of Graphs

Complete graphs: A graph with maximal connectivity
$K_n = (V, E)$ where

- $V = \{1, 2, 3, \ldots, n\}$
- $E = \{\{x, y\} \mid 1 \leq x < y \leq n\}$



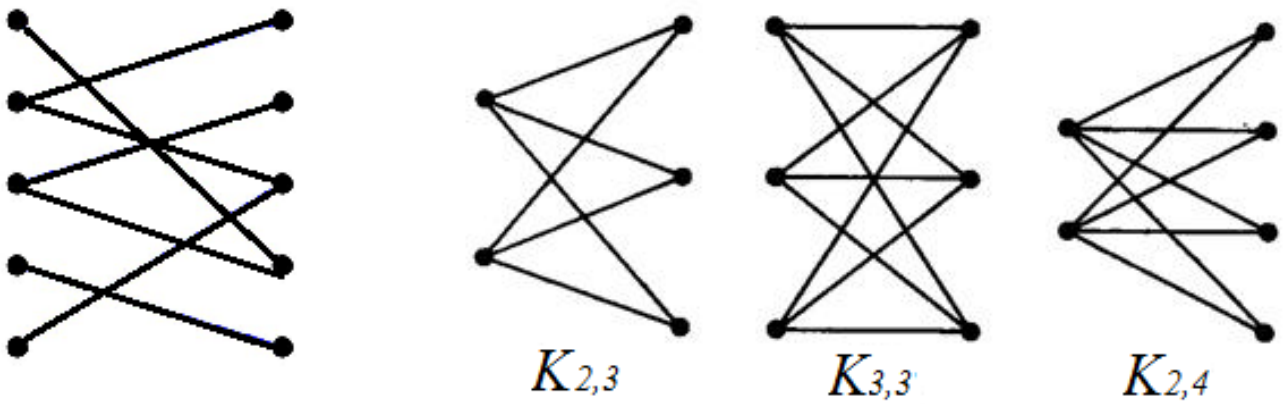
Graphs: Example

Important Classes of Graphs

- Bipartite graphs: The graph consists of two parts, and edges are only between (not within) these two parts.
- Complete bipartite graphs: The most densely connected bipartite graphs.

  $K_{m,n} = (V, E)$ where

  - $V = \{v_1, v_2, \ldots, v_m, u_1, u_2, \ldots, u_n\}$
  - $E = \{\{u_i, v_j\} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$



Part II: Implementing Graphs

Graphs

Directed Graphs.
= Nodes + Directed Edges

Graphs.
= Nodes + Edges

Graphs

Directed Graphs.
= Nodes + Directed Edges

Graphs.
= Nodes + Edges

Question: How to implement a graph?

The Graph ADT

The Graph ADT (Abstract Data Type) maintains a graph in the following way:

- Nodes: Each node identified by a unique `label`
- Edges: Each edge identified by a "source" and a "target" node

and provides operations for querying and updating a graph:

- Membership check for a node/edge
- Iterating nodes
- Adding/Removing a node
- Adding/Removing an edge

Note: We describe implementations of digraphs although using the name "`Graph`".

Graphs: Interface

```
public interface Graph<N> {
    public void add(N node);
    // pre:  node is non-null
    public void addEdge(N node1, N node2);
    // pre:  node1 and node2 are labels of existing nodes
    // post:  an edge is inserted between node1 and node2
    public N remove(N node)
    // pre:  node is non-null
    // post:  node is removed, if found
    public N removeEdge(N node1, N node 2)
    // pre:  node1 and node2 are existing nodes
    // post:  edge is removed, return its target
    public boolean contains(N label)
    // post:  returns true iff node with label exists
    public boolean containsEdge(N node1, N node2)
    // post:  return true iff (node1,node2) is an edge
    public Iterator<N> iterator();
    // post:  returns iterator across all nodes
}
```

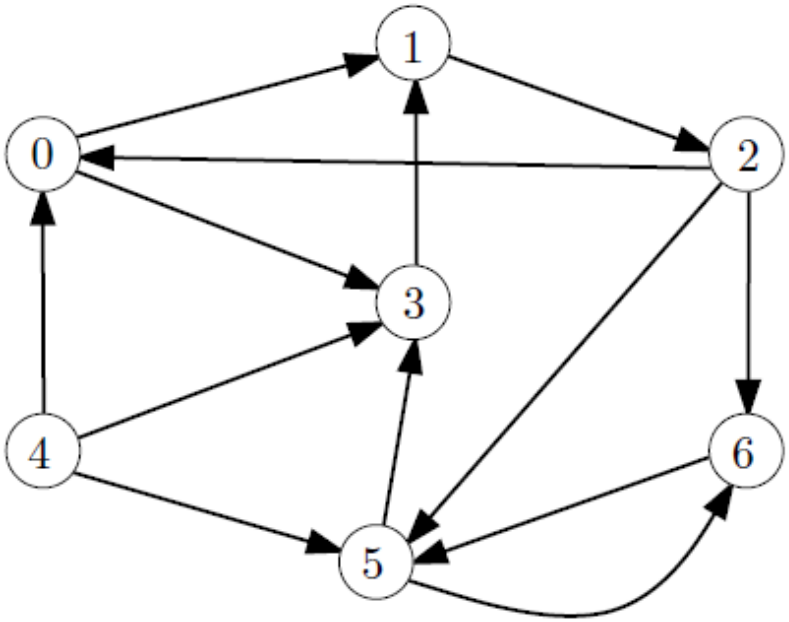
Implementation 1: Adjacency Matrix

Idea: Using an $n \times n$ Boolean matrix to represent the edge relation.

Implementation 1: Adjacency Matrix

Idea: Using an $n \times n$ Boolean matrix to represent the edge relation.

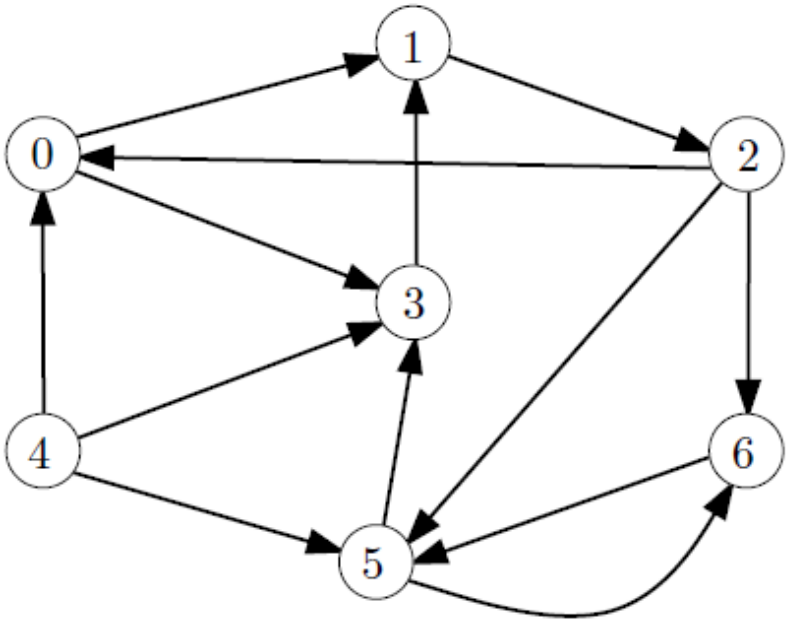
The adjacency matrix of a digraph is an $n \times n$ Boolean matrix whose (i, j)-entry is true if $(i, j) \in E$.



Implementation 1: Adjacency Matrix

Idea: Using an $n \times n$ Boolean matrix to represent the edge relation.

The adjacency matrix of a digraph is an $n \times n$ Boolean matrix whose (i, j)-entry is true if $(i, j) \in E$.



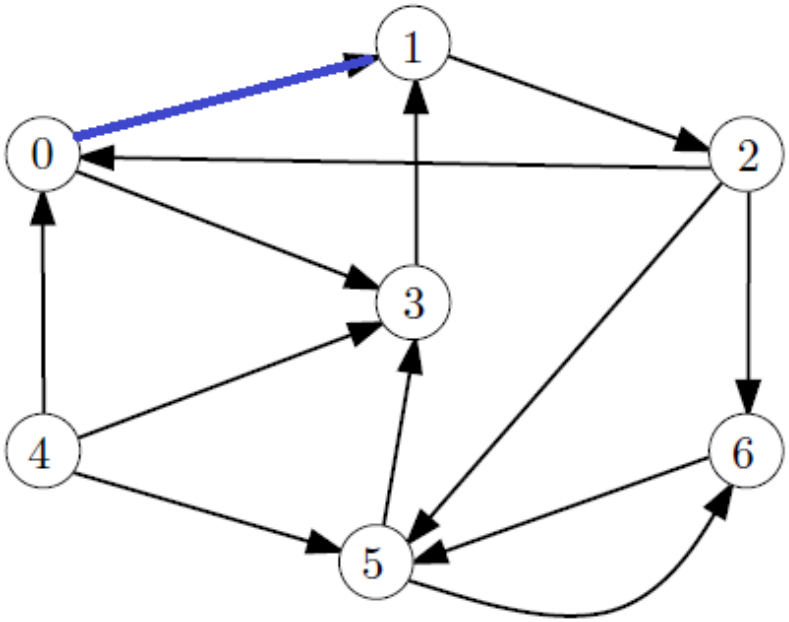
Adjacency Matrix

$$
\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Implementation 1: Adjacency Matrix

Idea: Using an $n \times n$ Boolean matrix to represent the edge relation.

The adjacency matrix of a digraph is an $n \times n$ Boolean matrix whose (i, j)-entry is true if $(i, j) \in E$.



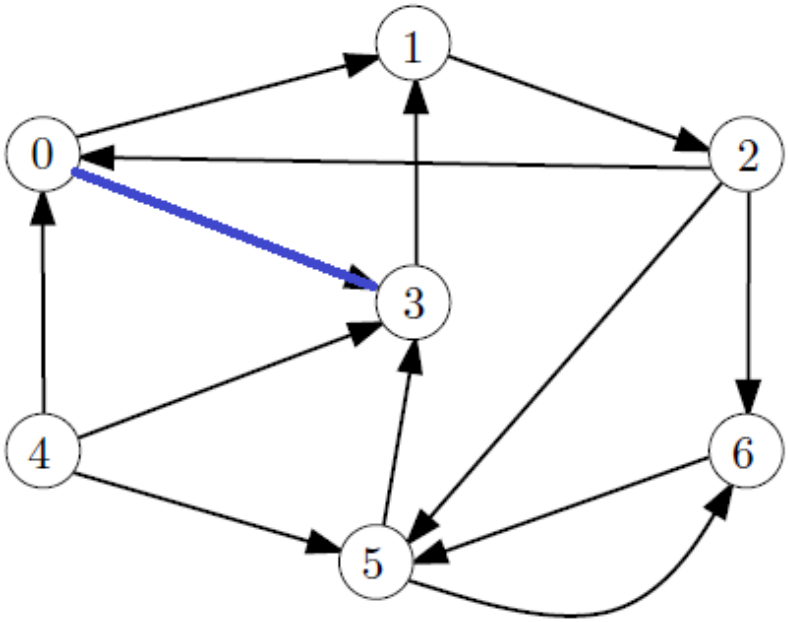
Adjacency Matrix

$$
\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & 0 & \boxed{1} & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Implementation 1: Adjacency Matrix

Idea: Using an $n \times n$ Boolean matrix to represent the edge relation.

The adjacency matrix of a digraph is an $n \times n$ Boolean matrix whose (i, j)-entry is true if $(i, j) \in E$.



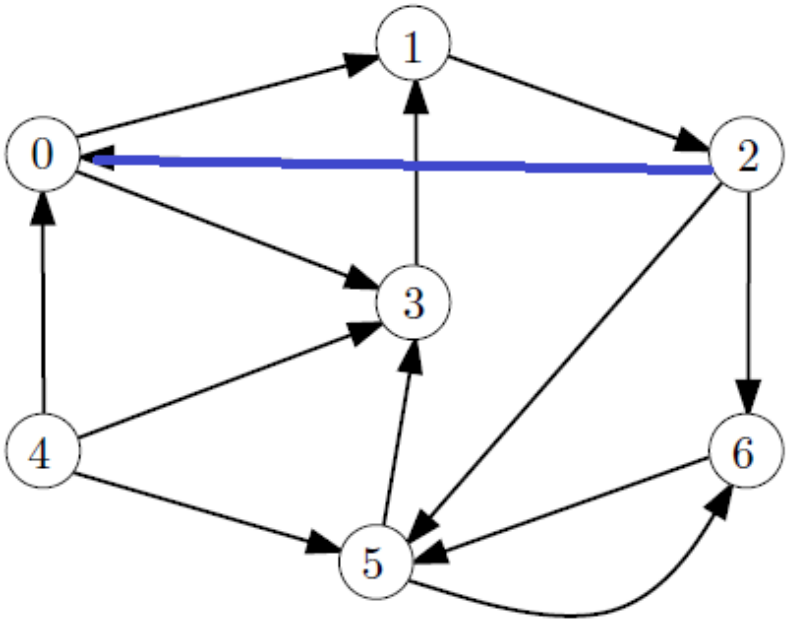
Adjacency Matrix

$$
\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & 0 & 1 & 0 & \boxed{1} & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Implementation 1: Adjacency Matrix

Idea: Using an $n \times n$ Boolean matrix to represent the edge relation.

The adjacency matrix of a digraph is an $n \times n$ Boolean matrix whose (i, j)-entry is true if $(i, j) \in E$.



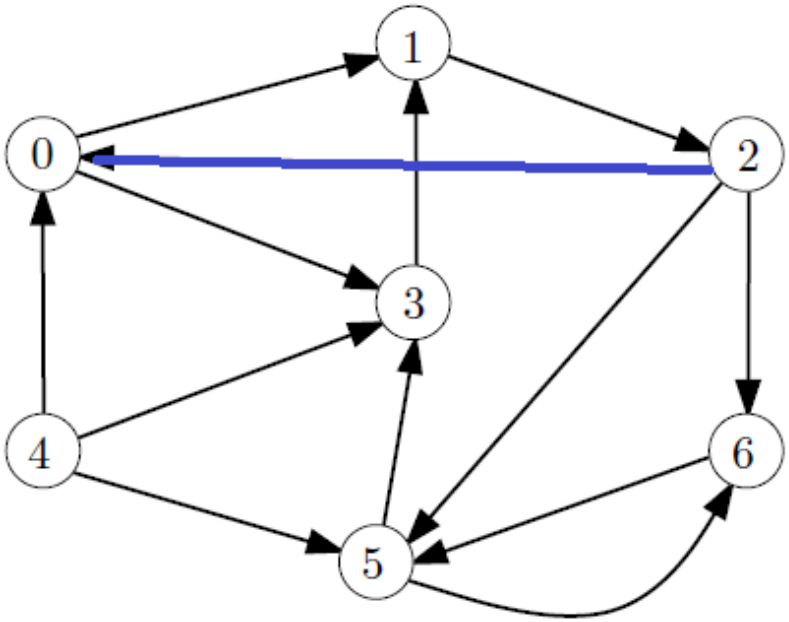
Adjacency Matrix

$$
\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & \boxed{1} & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Implementation 1: Adjacency Matrix

Idea: Using an $n \times n$ Boolean matrix to represent the edge relation.

The adjacency matrix of a digraph is an $n \times n$ Boolean matrix whose (i, j)-entry is true if $(i, j) \in E$.



Adjacency Matrix

$$
\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & \boxed{1} & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Row of node i: The outgoing edges of i
Column of node i: The incoming edges of i

Implementation 1: Adjacency Matrix

```
public class MatrixGraph<N> implements Graph<N>{
    // implement a matrix as a 2D array
    protected boolean data[][];
    // A map that associates nodes with indices
    protected Map<N,Integer> dict;
    // A set of free indices -- for remove and add
    protected Hashtable<Integer> free;

    public MatrixGraph(int ord){
    // post:  construct an empty graph, where
    //     order is the initial capacity of data
        data = new boolean[ord][ord];
        dict = new HashMap<N,Integer>();
        free = new Hashtable<Integer>();
        for (int row = 0; row<data.length; row++)
            free.put(new Integer(row));
    }
```

Implementation 1: Adjacency Matrix

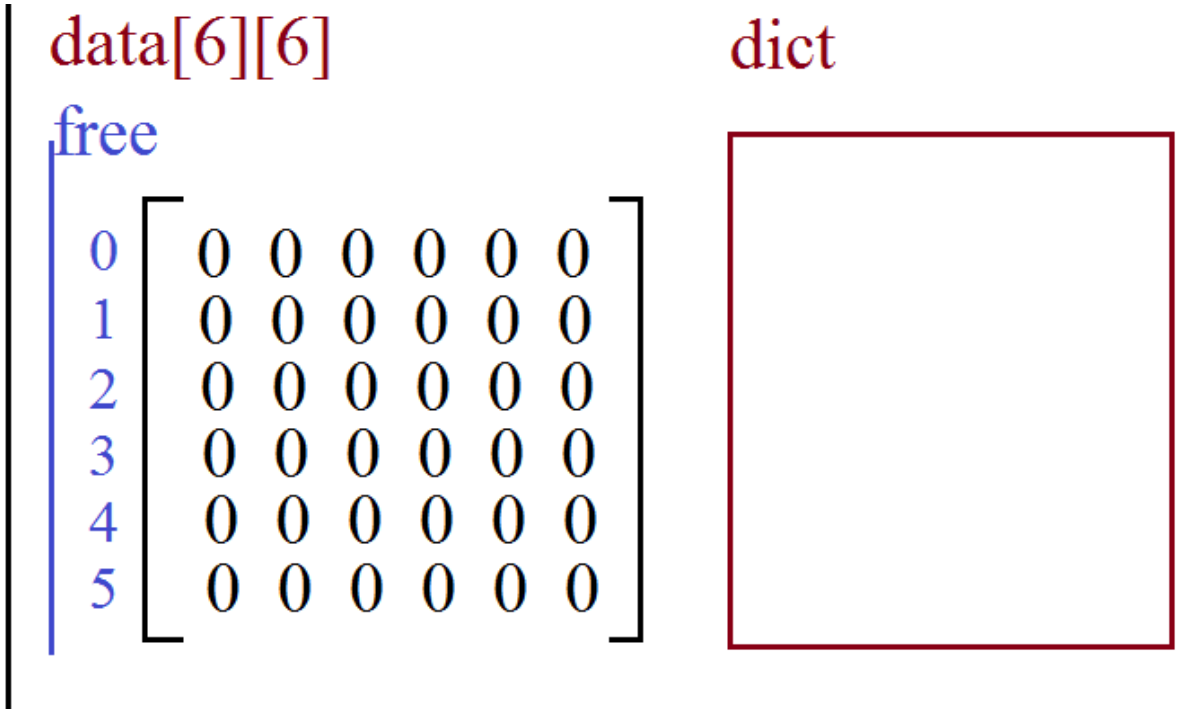
Updating: `add(N node)`

1. Check if `node` is already in dict; if so, return
2. Check if `free` is empty; if so, expand its capacity
3. Pick any `index` from `free`
4. Remove `index` from `free`
5. Add `(node, i)` to dict

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:



Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add("Homer")

data[6][6]

free

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

dict

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add("Homer")

data[6][6]

free

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

dict

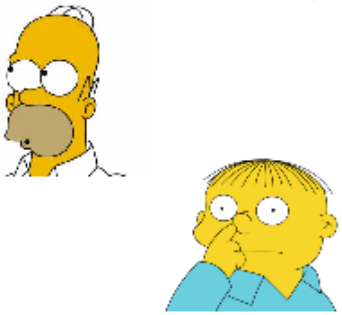
("Homer",0)

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add(" Ralph ")



data[6][6]

free

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

dict

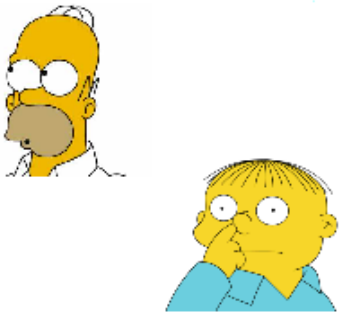
("Homer",0)

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add(" Ralph ")



data[6][6]

free

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

dict

("Homer",0)

("Ralph", 1)

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add(" Lisa ")

data[6][6]

free

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

dict

("Homer",0)

("Ralph", 1)

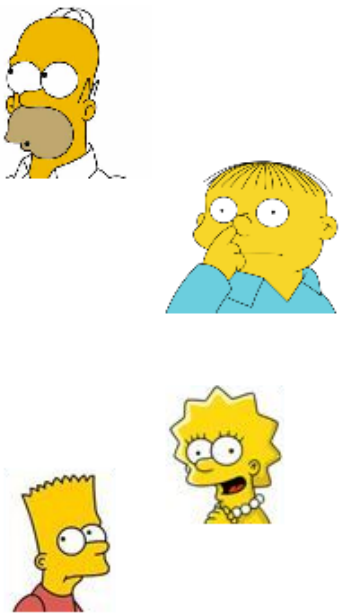
("Lisa", 2)

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add(" Bart ")



data[6][6]

free

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

dict

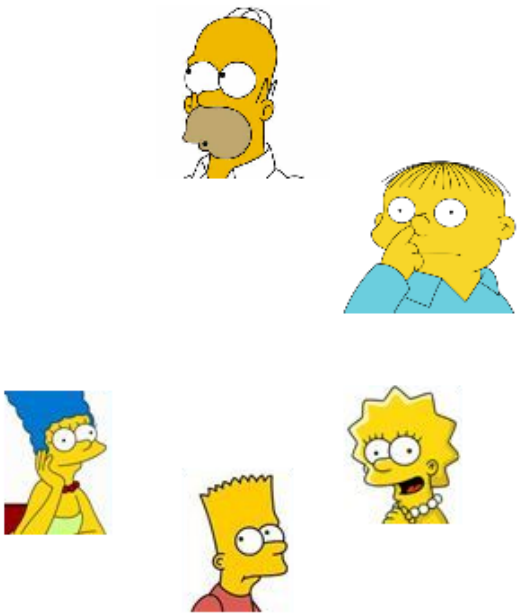
("Homer",0)
("Ralph", 1)
("Lisa", 2)
("Bart", 3)

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add("Marge")



data[6][6]

free

$$
\begin{array}{c}
0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5
\end{array}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

dict

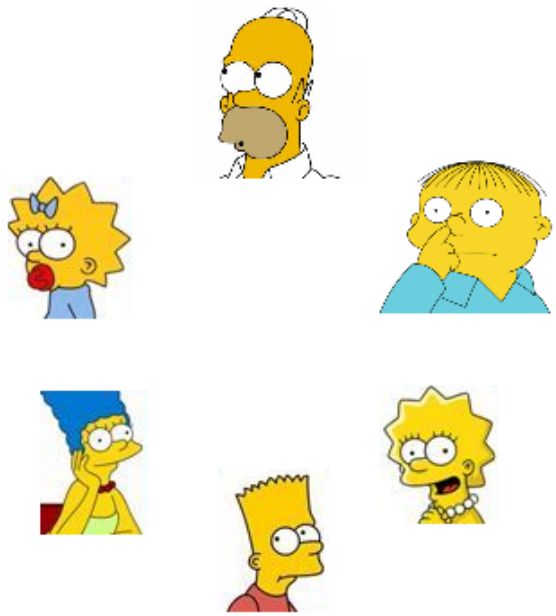
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

Example:

add("Maggie")



data[6][6]

free

$$
\begin{array}{c}
0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5
\end{array}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\begin{array}{l}
\leftarrow \text{("Homer",0)} \\
\leftarrow \text{("Ralph", 1)} \\
\leftarrow \text{("Lisa", 2)} \\
\leftarrow \text{("Bart", 3)} \\
\leftarrow \text{("Marge",4)} \\
\leftarrow \text{("Maggie",5)}
\end{array}
$$

dict

Implementation 1: Adjacency Matrix

Updating: `add(N node)`

```
public void add(N node){
    // Check if node is already in dict
    if(dict.contains(node)) return;
    // Check if free is empty
    if(free.isEmpty) expandCapacity();
    // Pick a free row index from free, assuming
    // we have a getFirst() method in hashtable
    Integer index=free.getFirst();
    // Remove index from free
    free.remove(index);
    // Add (node,index) to dict
    dict.put(node, index);
}
```

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

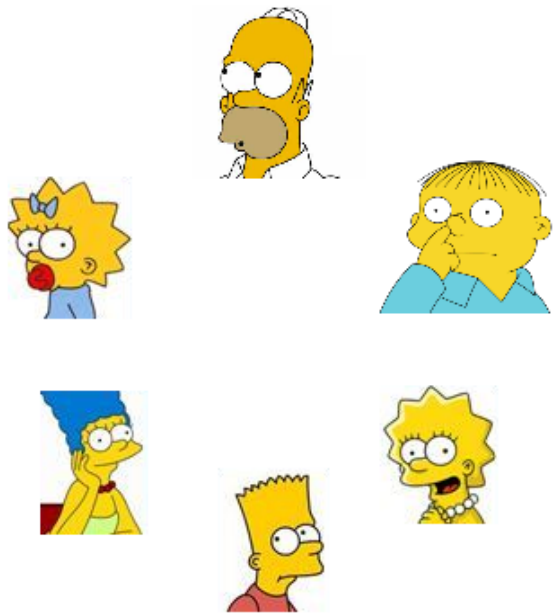
1. Check if both `node1`, `node2` are in the graph
2. If found, get the node from `dict` to find i, j corresponding to `node1`, `node2` respectively
3. Set `data[i][j]` to `true`

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

Example:

addEdge("Ralph","Bart")



data[6][6]

free

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

dict

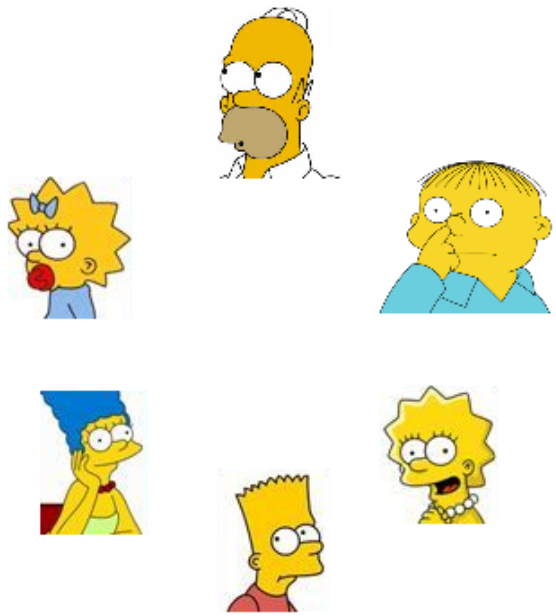
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

Example:

addEdge("Ralph","Bart")



data[6][6]

free

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & \boxed{0} & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

dict

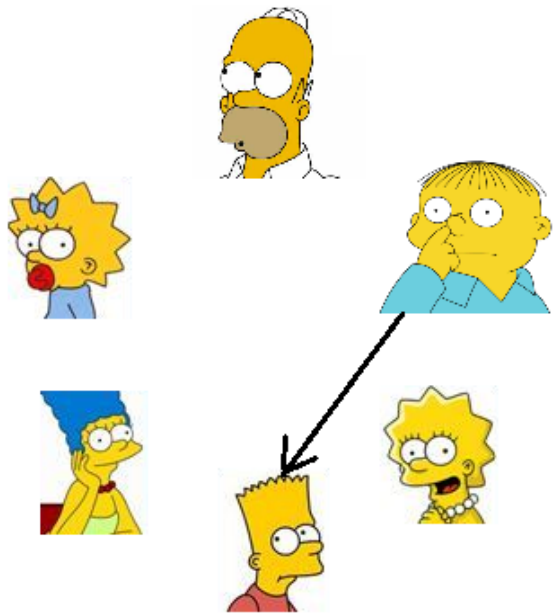
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

Example:

addEdge("Ralph","Bart")



data[6][6]

free

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & \boxed{1} & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

dict

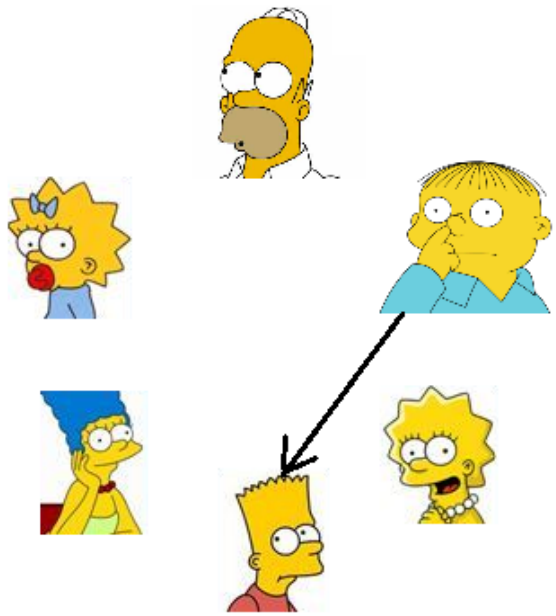
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

Example:

addEdge("Bart","Homer")



data[6][6]

free

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

dict

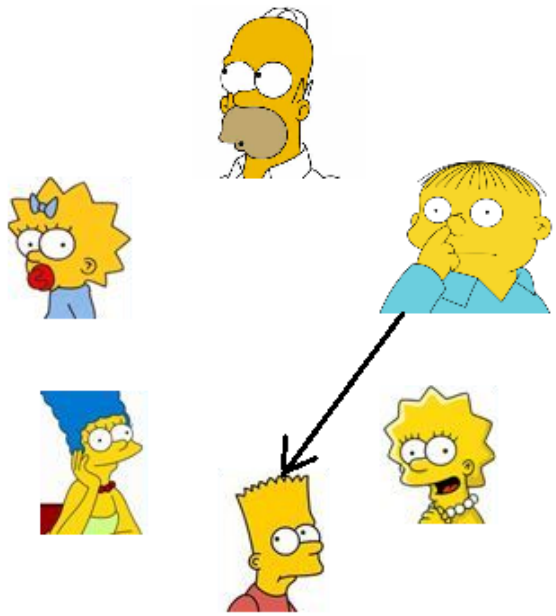
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

Example:

addEdge("Bart","Homer")



data[6][6]

free

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

dict

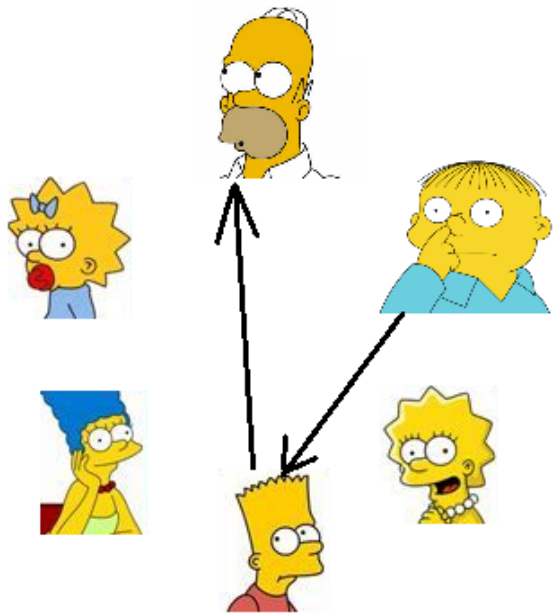
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

Example:

addEdge("Bart","Homer")



data[6][6]

free

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & \boxed{1} & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

dict

- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `addEdge(N node1,N node2)`

```
public void addEdge(N node1, N node2){
    // Get the value of node1, node2 from dict
    if (!dict.contains(node1)||!dict.contains(node2))
        return;
    int index1 = dict.getValue(node1);
    int index2 = dict.getValue(node2);
    data[index1][index2] = true;
}
```

Implementation 1: Adjacency Matrix

Updating: `removeEdge(N node1,N node2)`

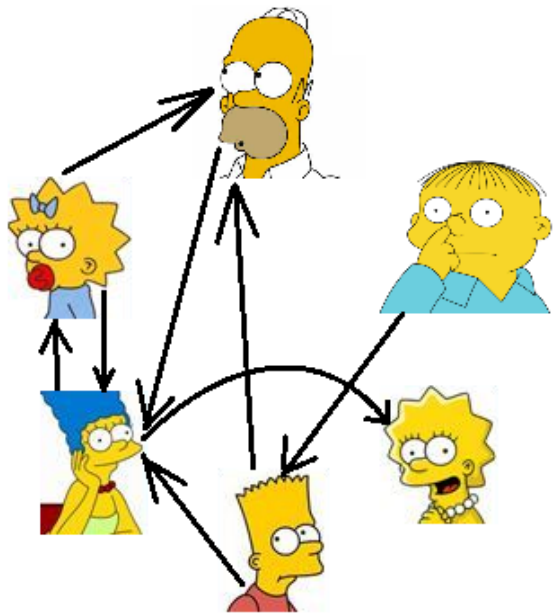
1. Check if node1, node2 are both in the graph
2. If found, get the node from `dict` to find i, j corresponding to `node1`, `node2` respectively
3. Set `data[i][j]` to `false`

Implementation 1: Adjacency Matrix

Updating: `removeEdge(N node1,N node2)`

Example:

removeEdge("Bart","Marge")



data[6][6]

free

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

dict

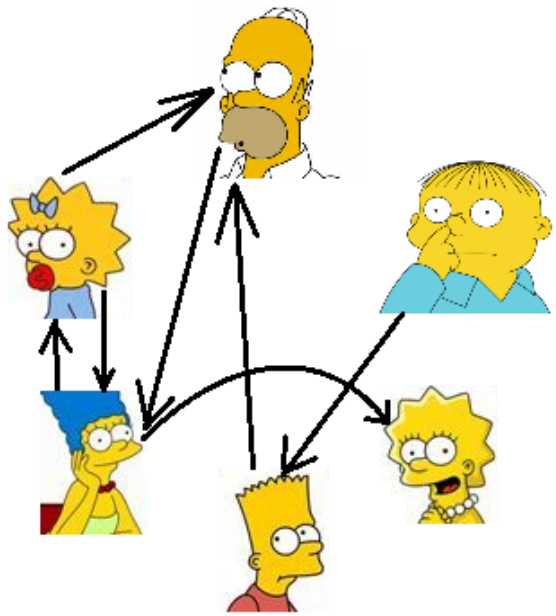
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `removeEdge(N node1,N node2)`

Example:

removeEdge("Bart","Marge")



data[6][6]

free

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & \boxed{0} & 0 \\
4 & 0 & 0 & 1 & 0 & 0 & 1 \\
5 & 1 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

dict

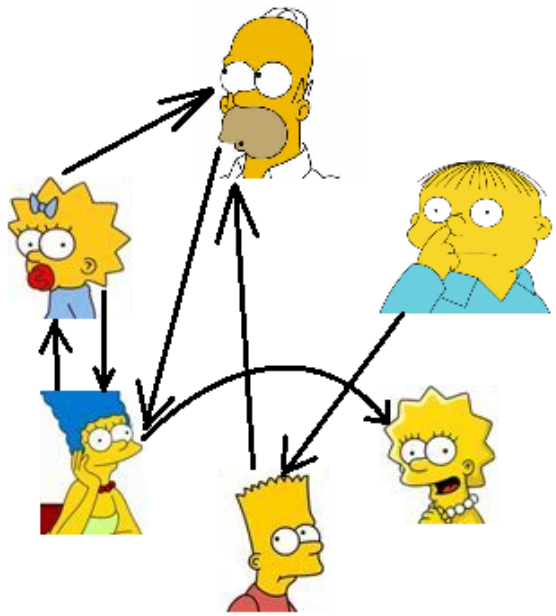
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `removeEdge(N node1,N node2)`

Example:

removeEdge("Marge", "Lisa")



data[6][6]

free

$$
\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 0 & 1 \\
5 & 1 & 0 & 0 & 0 & 1 & 0
\end{array}
$$

dict

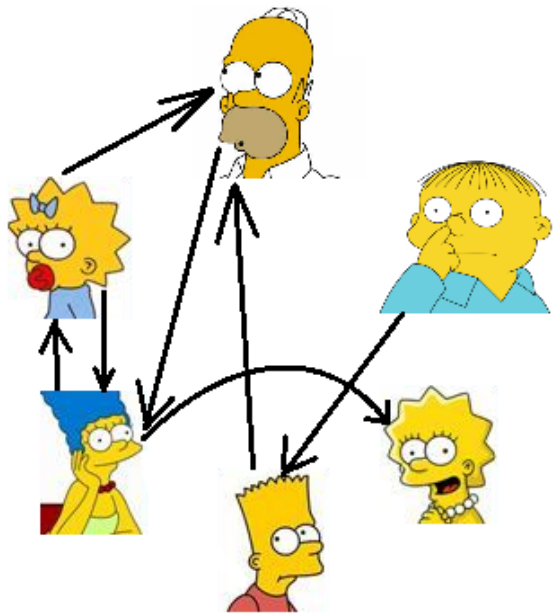
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `removeEdge(N node1,N node2)`

Example:

removeEdge("Marge", "Lisa")



data[6][6]

free

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

dict

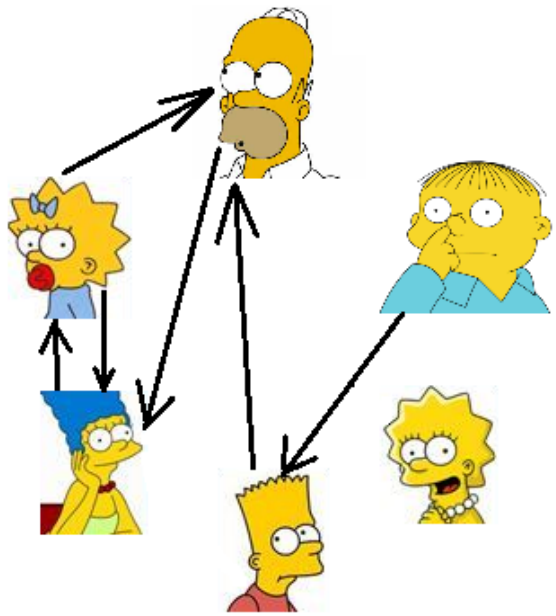
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `removeEdge(N node1,N node2)`

Example:

removeEdge("Marge", "Lisa")



data[6][6]

free

	0	1	2	3	4	5
0	0	0	0	0	1	0
1	0	0	0	1	0	0
2	0	0	0	0	0	0
3	1	0	0	0	0	0
4	0	0	$\boxed{0}$	0	0	1
5	1	0	0	0	1	0

dict

- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `removeEdge(N node1,N node2)`

```
public N removeEdge(N node1, N node2){
    // Get the value of node1, node2 from dict
    if (!dict.contains(node1)||!dict.contains(node2))
        return null;
    int index1 = dict.getValue(node1);
    int index2 = dict.getValue(node2);
    data[index1][index2] = false;
    return node2;
}
```

Implementation 1: Adjacency Matrix

Updating: `remove(N node)`

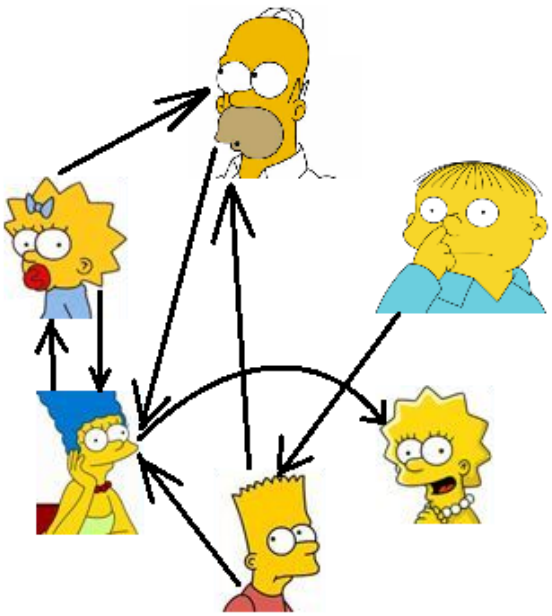
1. Check if `node` is in the graph; if not, return
2. Get `node` from `dict` to get its row number i
3. Set all entries in row i and column i to `false`
4. Remove `(node,i)` from `dict`
5. Put i into `free`

Implementation 1: Adjacency Matrix

Updating: `remove(N node)`

Example:

remove("Bart")



data[6][6]

free

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

dict

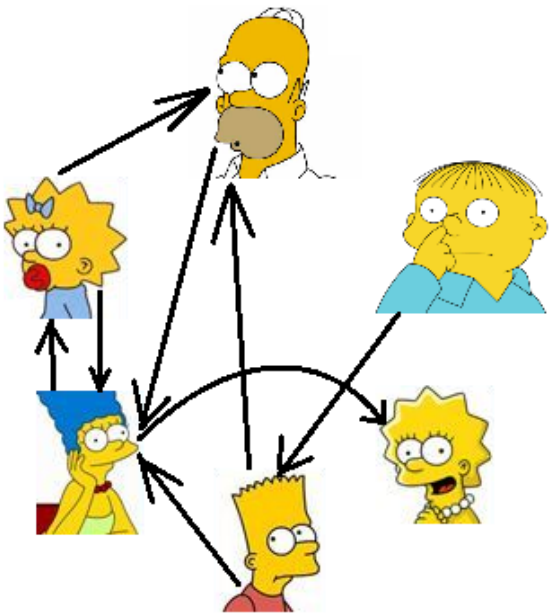
- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `remove(N node)`

Example:

remove("Bart")



data[6][6]

free

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

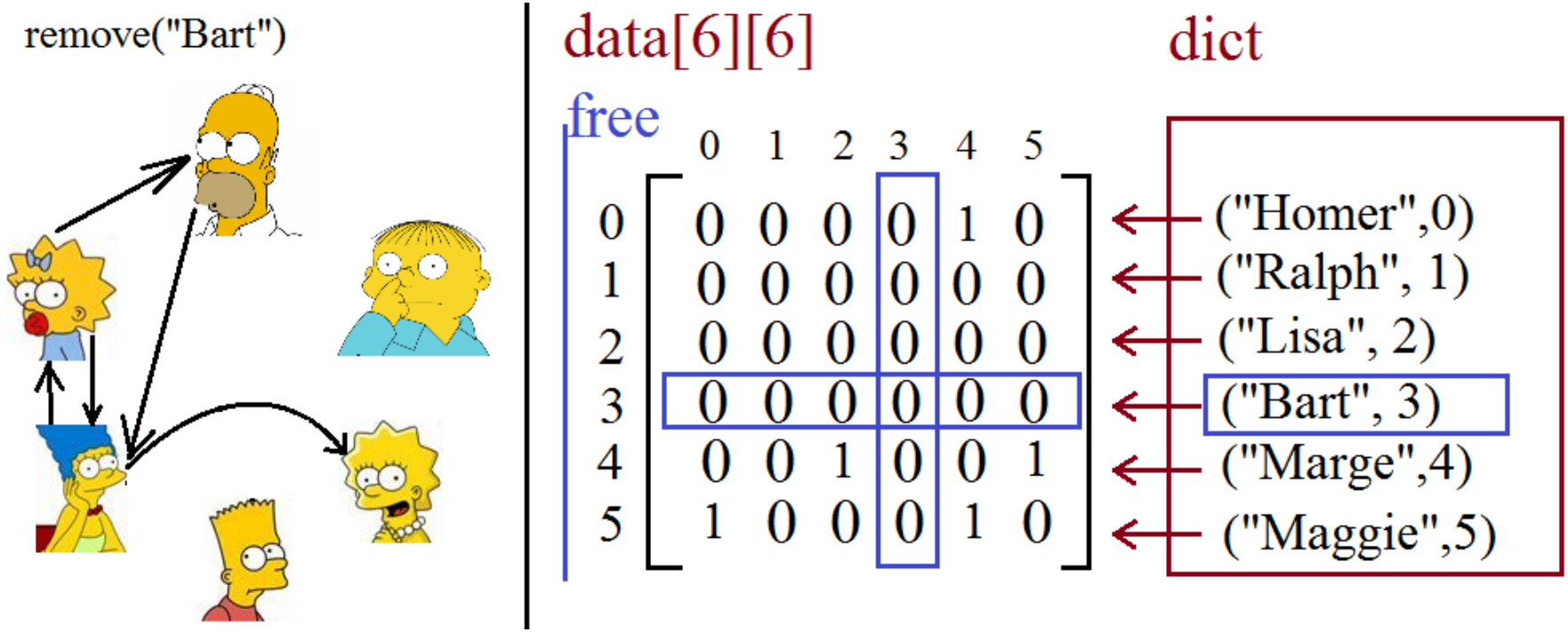
dict

- ("Homer",0)
- ("Ralph", 1)
- ("Lisa", 2)
- ("Bart", 3)
- ("Marge",4)
- ("Maggie",5)

Implementation 1: Adjacency Matrix

Updating: `remove(N node)`

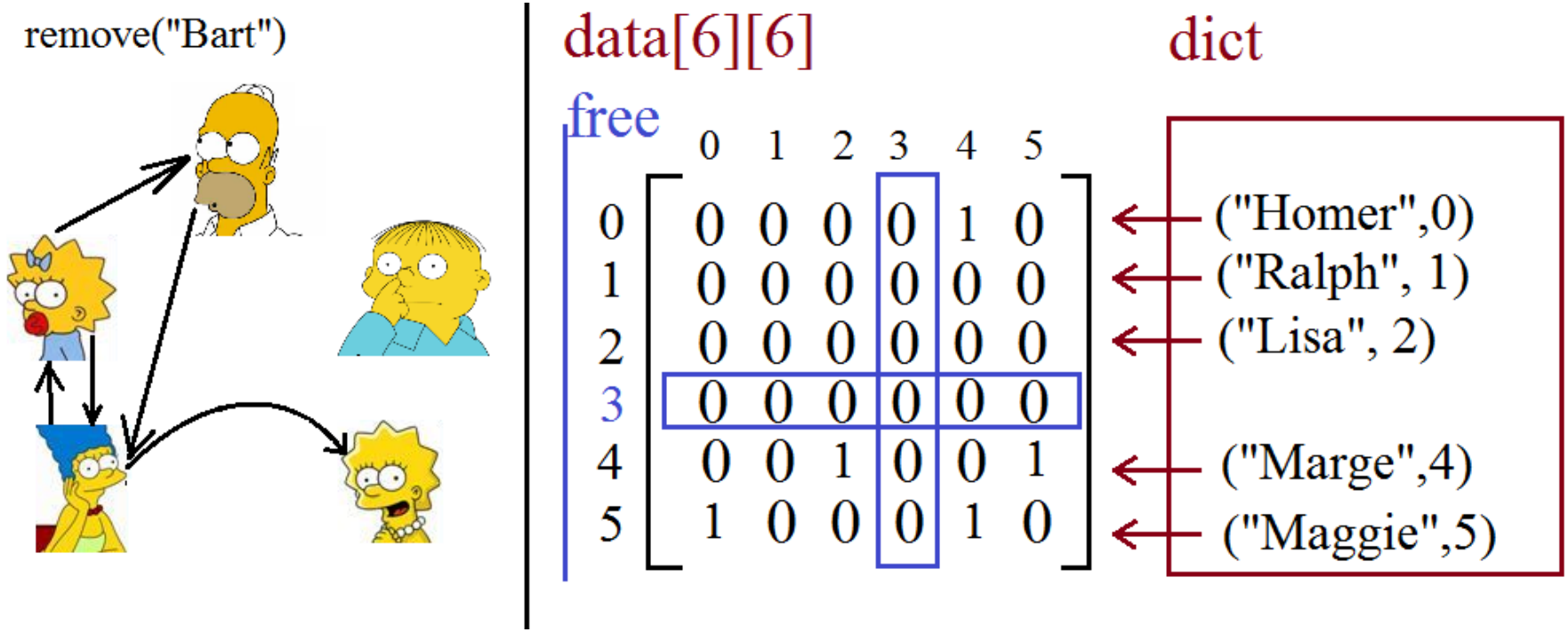
Example:



Implementation 1: Adjacency Matrix

Updating: `remove(N node)`

Example:



Implementation 1: Adjacency Matrix

Updating: `remove(N node)`

```
public N remove(N node){
    if (!dict.contains(node)) return null;
    int index = dict.getValue(node);
    for (int row=0; row<data.length; row++){
        data[row][index]=false;
        data[index][row]=false;
    }
    dict.remove(node);
    free.put(node);
}
```

Implementation 1: Adjacency Matrix

Queries:

- `contains(N node)`

  ```
  return dict.contains(node);
  ```

- `containsEdge(N node1, N node2)`

  ```
  int ind1=dict.getValue(node1);
  int ind2=dict.getValue(node2);
  return data[ind1][ind2];
  ```

- `iterator()`

  ```
  return data.keySet().iterator();
  ```

Implementation 1: Adjacency Matrix

Time Complexity:

- Assume an efficient hash table implementation.
* Assume we do not need to expand capacity
** Assume we need to expand capacity

Operations	Time Complexity
contains(node)	$O(1)$
containsEdge(node1, node2)	$O(1)$
add(node)*	$O(1)$
add(node)**	$O(n^2)$
remove(node)	$O(n)$
addEdge(node1, node2)	$O(1)$
removeEdge(node1, node2)	$O(1)$

Advantages: Efficient update, efficient membership check

Disadvantage: Take up too much space
Inefficient node addition
Inefficient for obtaining outgoing edges $O(n)$

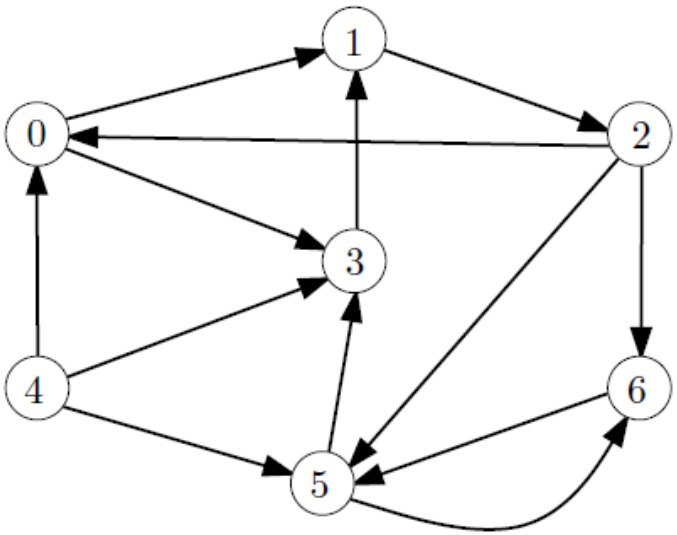
Implementation 2: Adjacency List

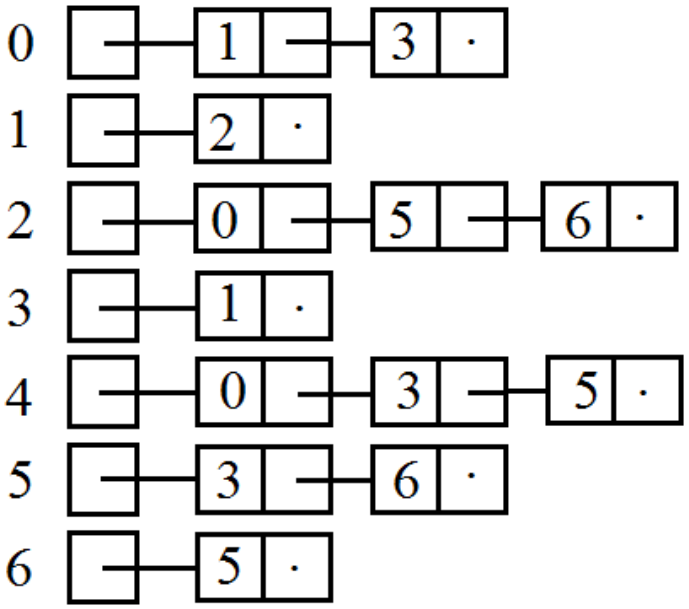
Idea: Using a collection of lists to represent the edge relation.

Implementation 2: Adjacency List

Idea: Using a collection of lists to represent the edge relation.

The adjacency list of a digraph consists of a collection of lists; where every node v is associated with a list of nodes indicating its out-going edges.





Implementation 2: Adjacency List

```
public class ListGraph<N> implements Graph<N>{
    // A map that associates nodes with their lists;
    // the list of a node stores all its out-neighbours
    protected Map<N,List<N>> data;

    protected ListGraph(){
        data = new HashMap<N,List<N>>();
    }
```

Implementation 2: Adjacency List

Updating: `add(N node)`

1. Check if `node` is already in the graph; if so, return
2. Create an empty list
3. Associate `node` with the list and put it in `data`

Implementation 2: Adjacency List

Updating: `add(N node)`

Example:

add("Homer")

data

Implementation 2: Adjacency List

Updating: `add(N node)`

Example:

add("Homer")

data

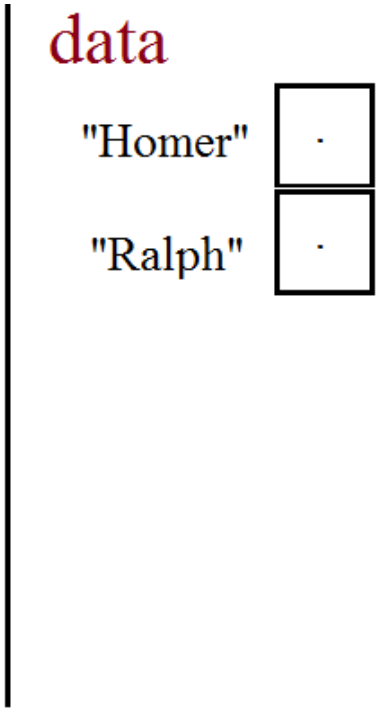
"Homer"

Implementation 2: Adjacency List

Updating: `add(N node)`

Example:

add(" Ralph ")



Implementation 2: Adjacency List

Updating: `add(N node)`

Example:

add(" Lisa ")

data

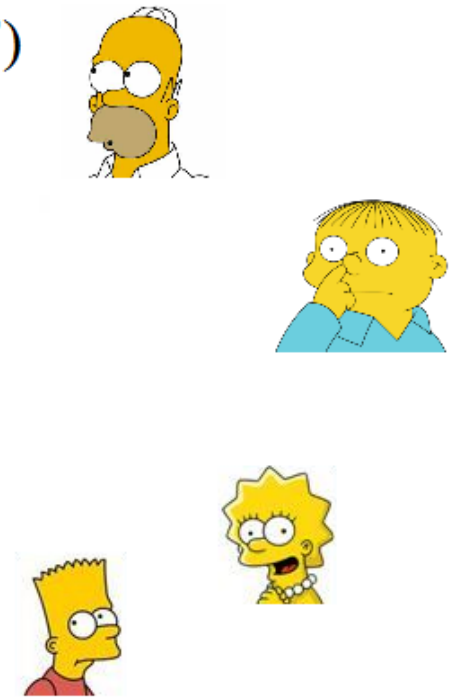
"Homer"	·
"Ralph"	·
"Lisa"	·

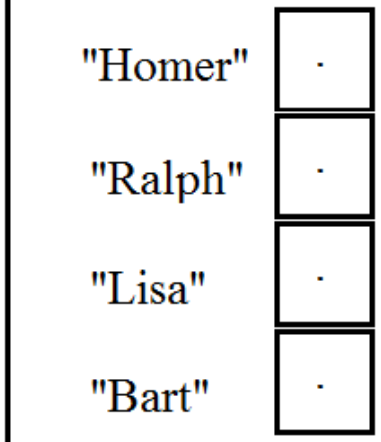
Implementation 2: Adjacency List

Updating: `add(N node)`

Example:

add(" Bart ")





Implementation 2: Adjacency List

Updating: `add(N node)`

```
public void add(N node){
    if(data.contains(node)) return;
    List<N> list= new LinkedList<N>();
    data.put(node, list);
}
```

Implementation 2: Adjacency List

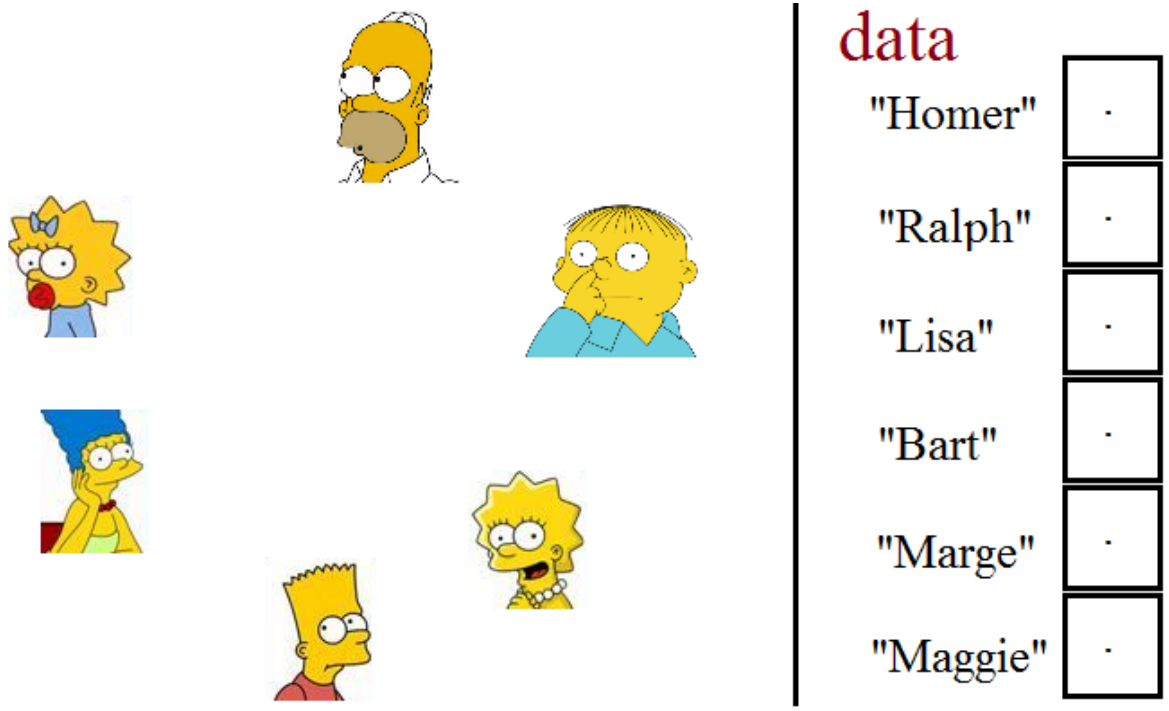
Updating: `addEdge(N node1, N node2)`

1. Check if `node1` and `node2` are both in the graph; return if not
2. Get from `data` the list of `node1`
3. Add `node2` to the list

Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

Example:

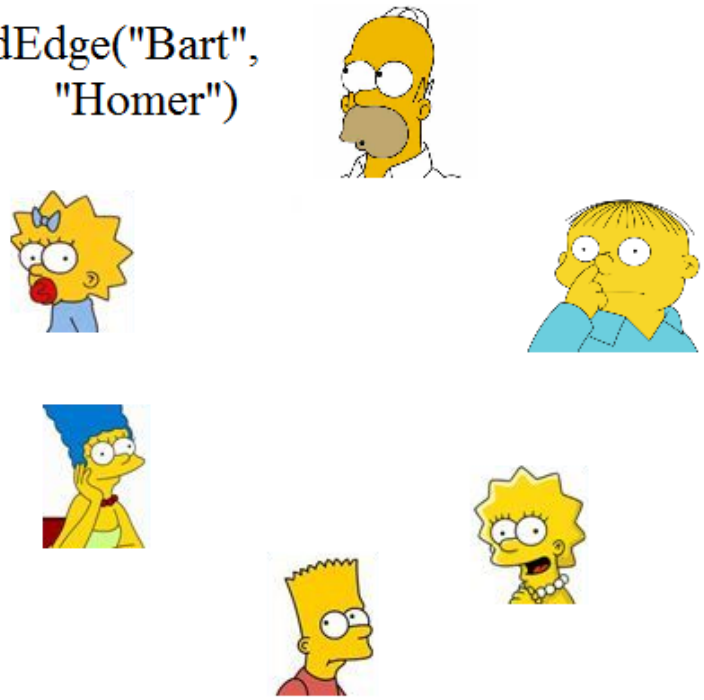


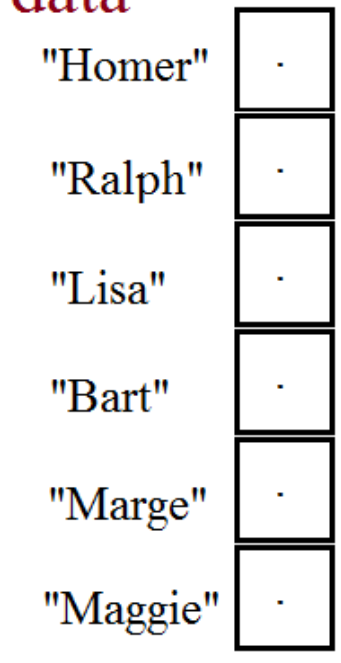
Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

Example:

addEdge("Bart", "Homer")

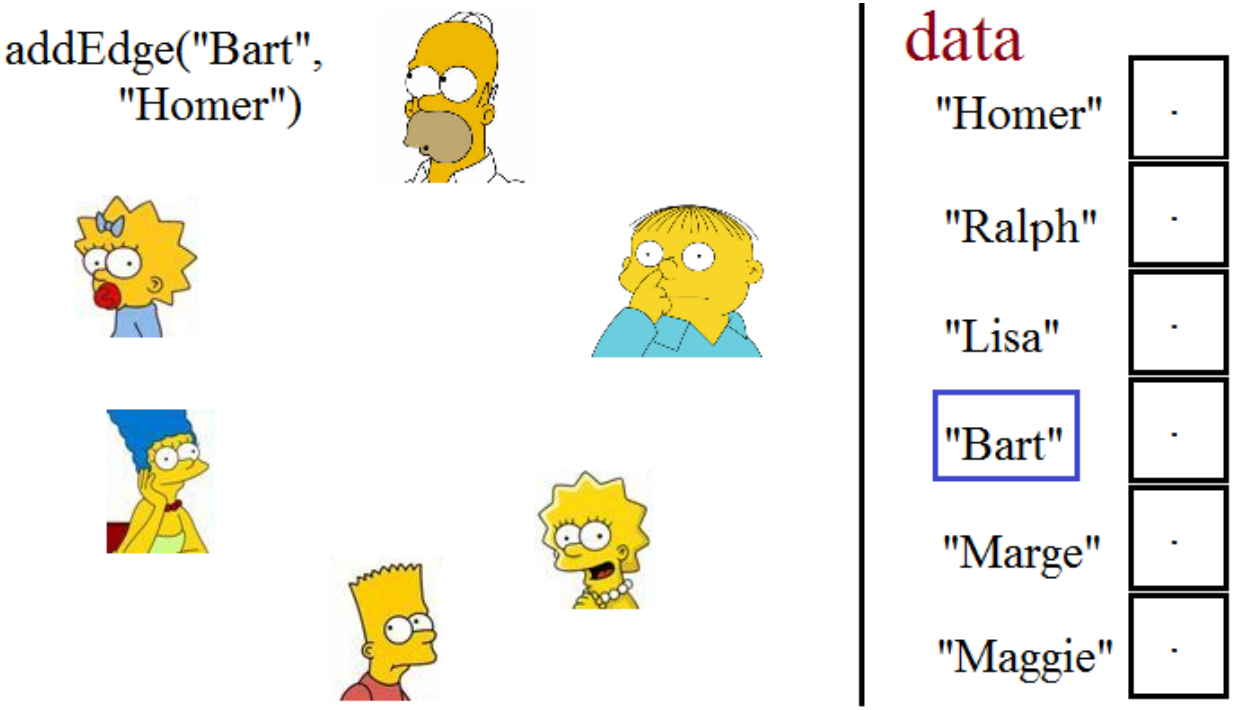




Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

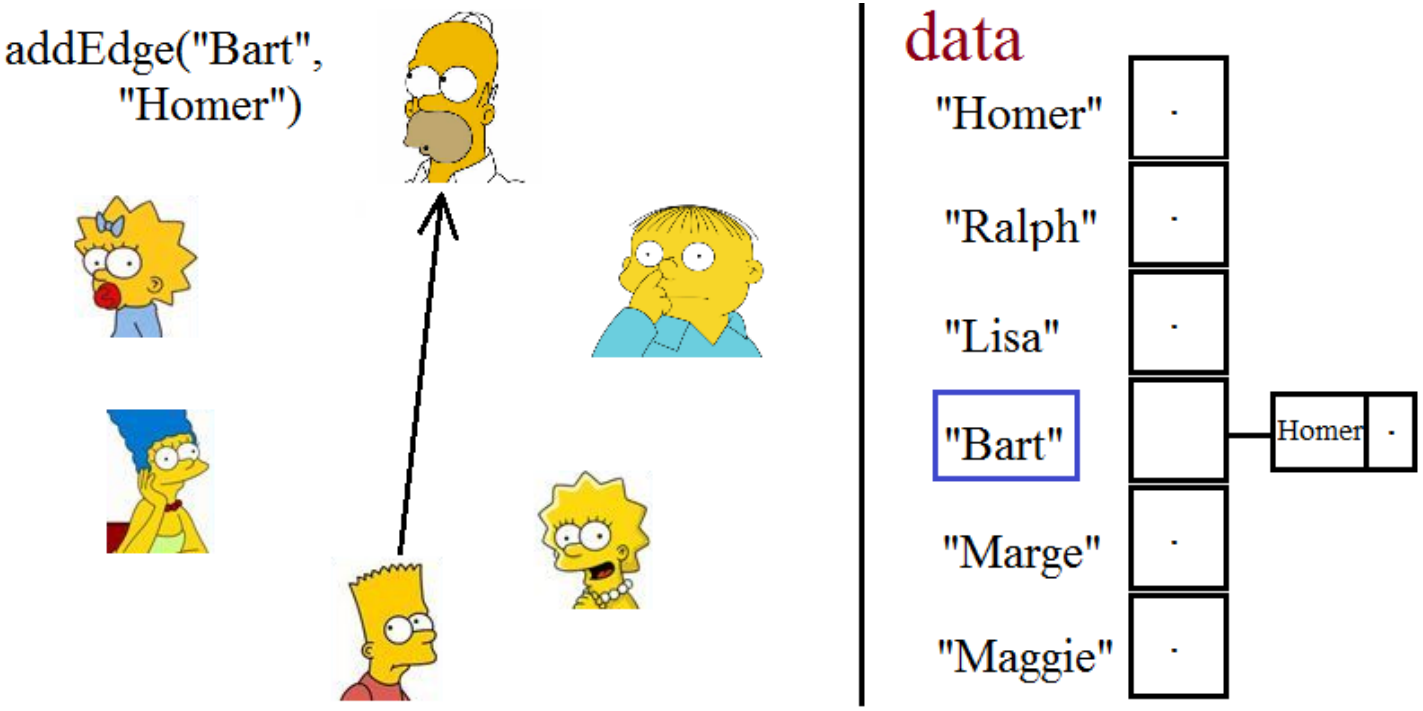
Example:



Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

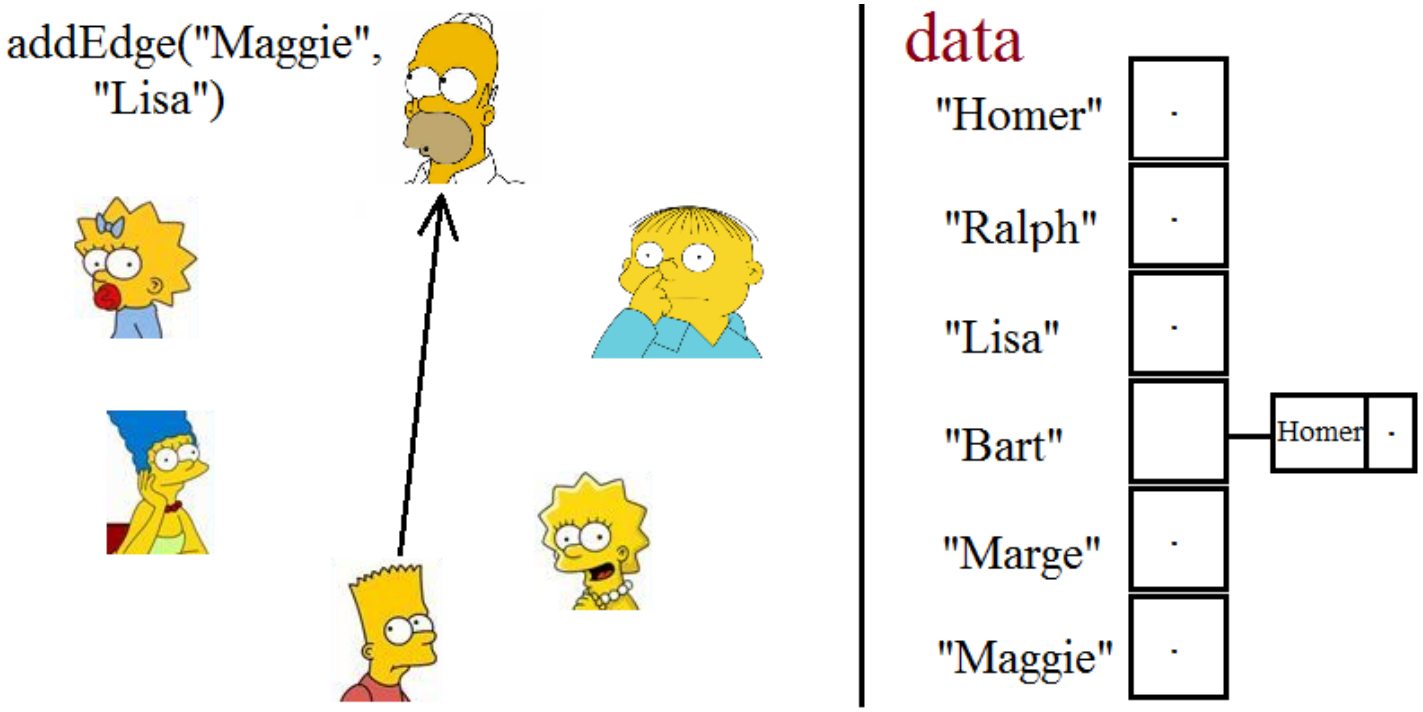
Example:



Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

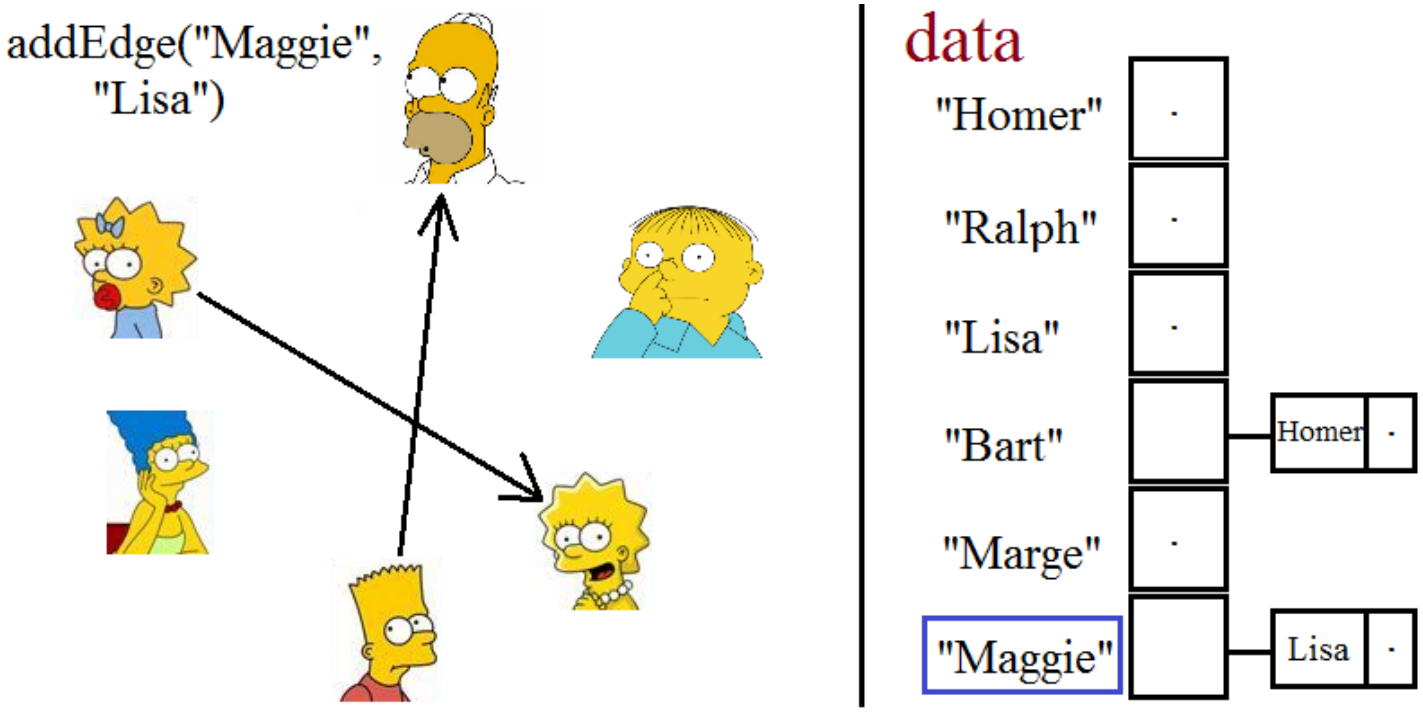
Example:



Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

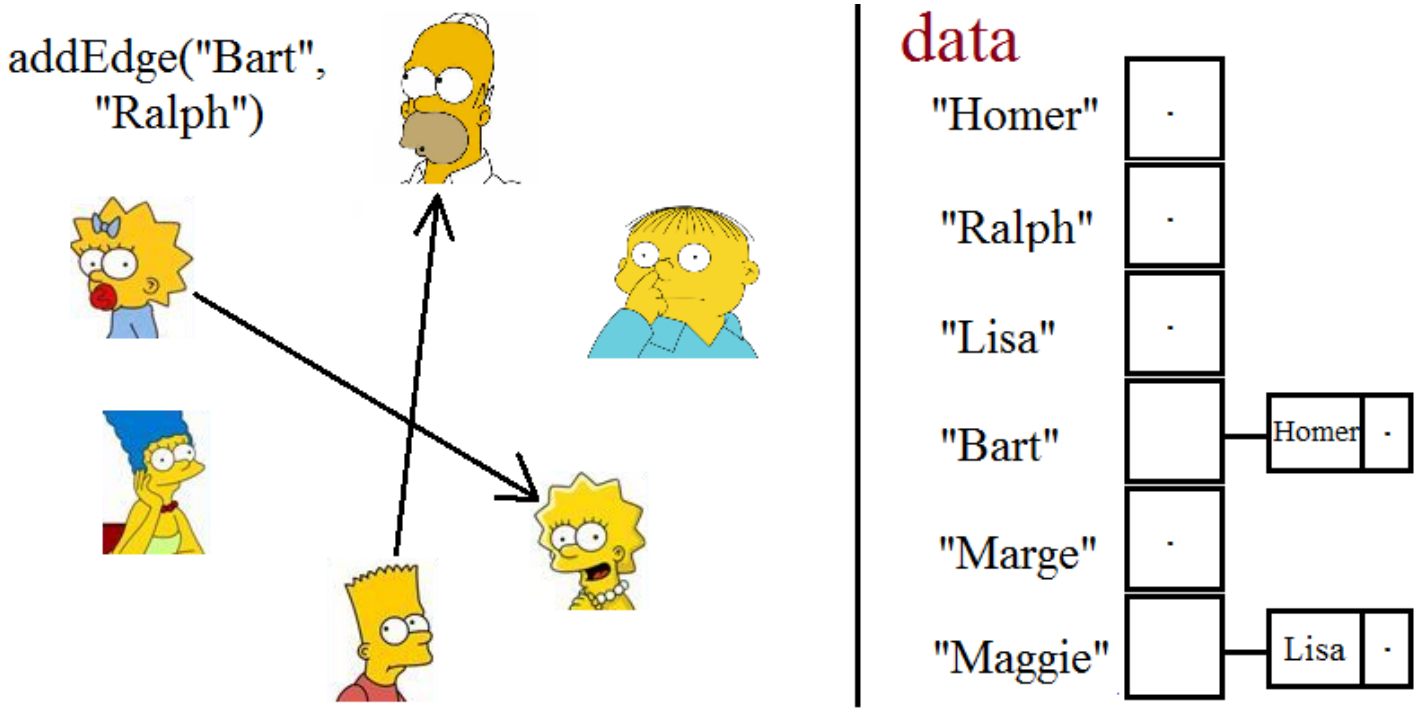
Example:



Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

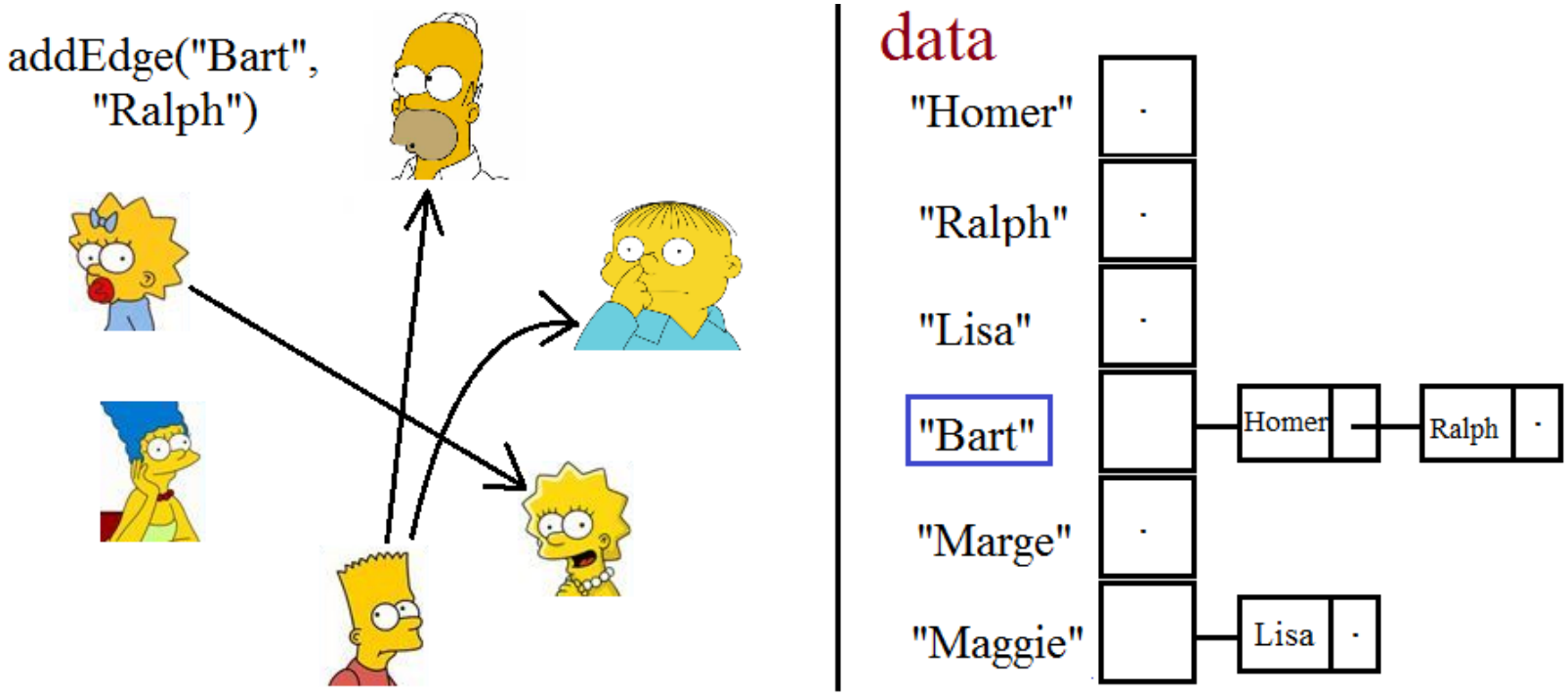
Example:



Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

Example:



Implementation 2: Adjacency List

Updating: `addEdge(N node1, N node2)`

```
public void addEdge(N node1, N node2){
    if(!data.contains(node1) || !data.contains(node2))
        return;
    List<N> list= data.getValue(node1);
    if(!list.contains(node2))
        list.add(node2);
}
```

Implementation 2: Adjacency List

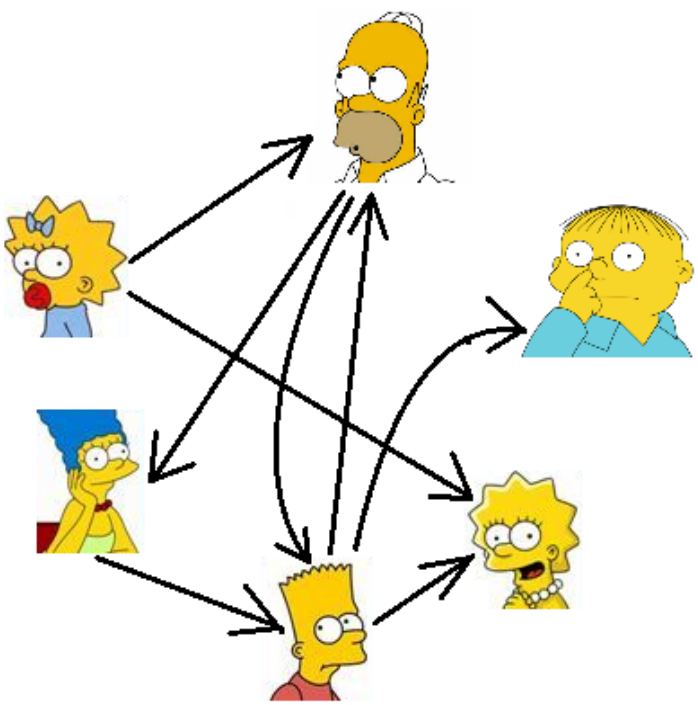
Updating: `removeEdge(N node1, N node2)`

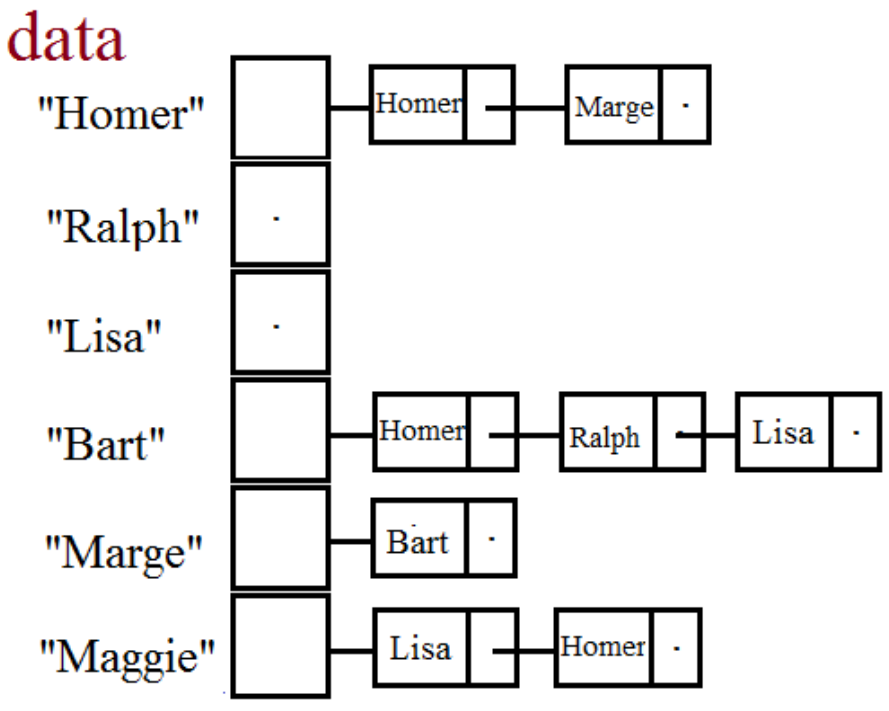
1. Check if `node1` and `node2` are both in the graph; return if not
2. Get from `data` the list of `node1`
3. Remove `node2` from the list

Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

Example:

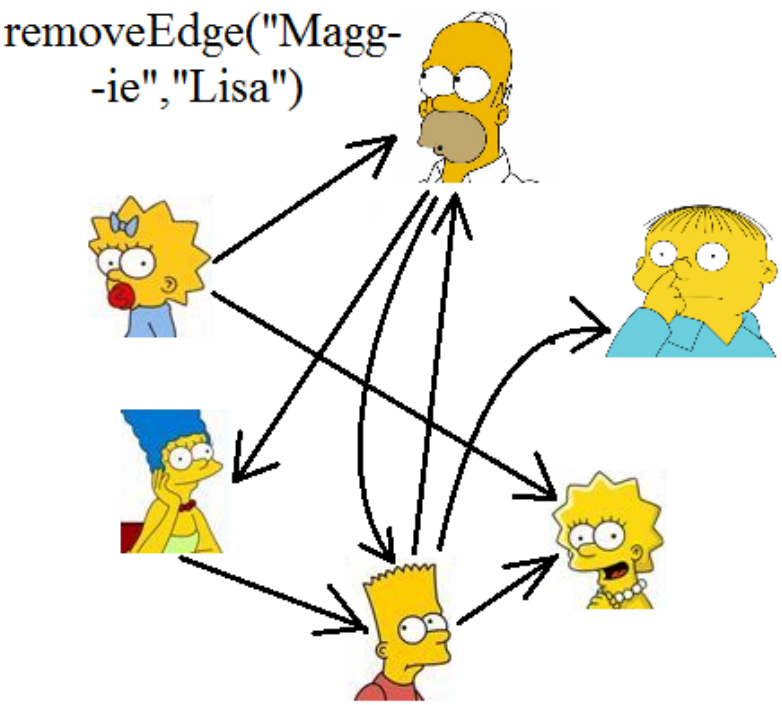


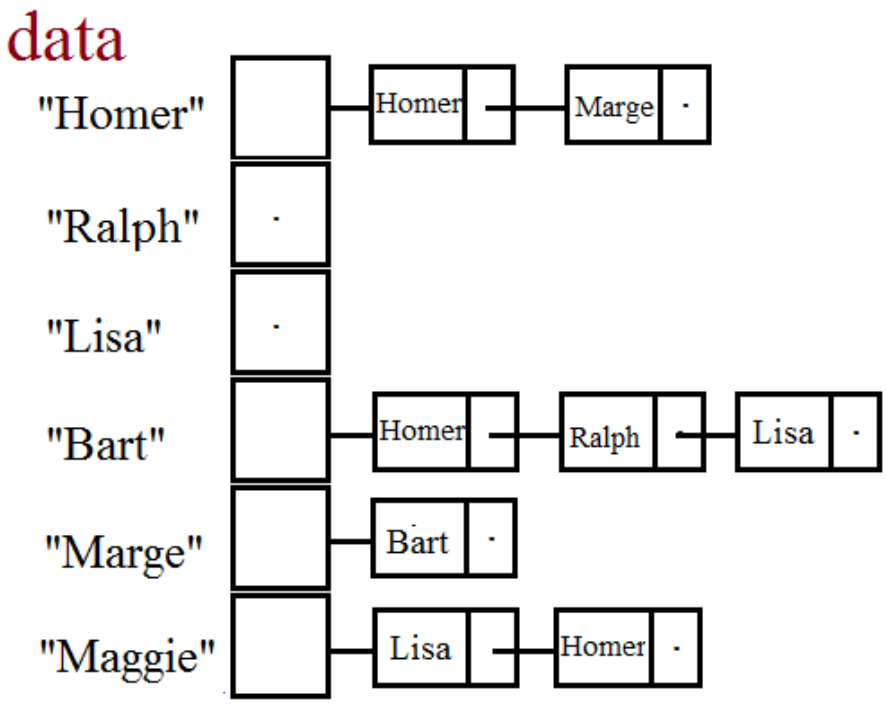


Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

Example:

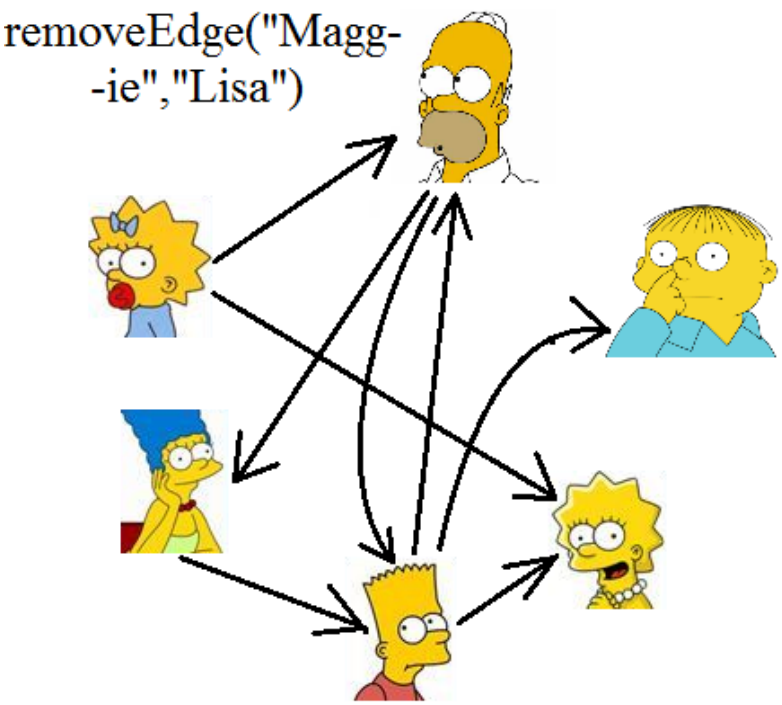


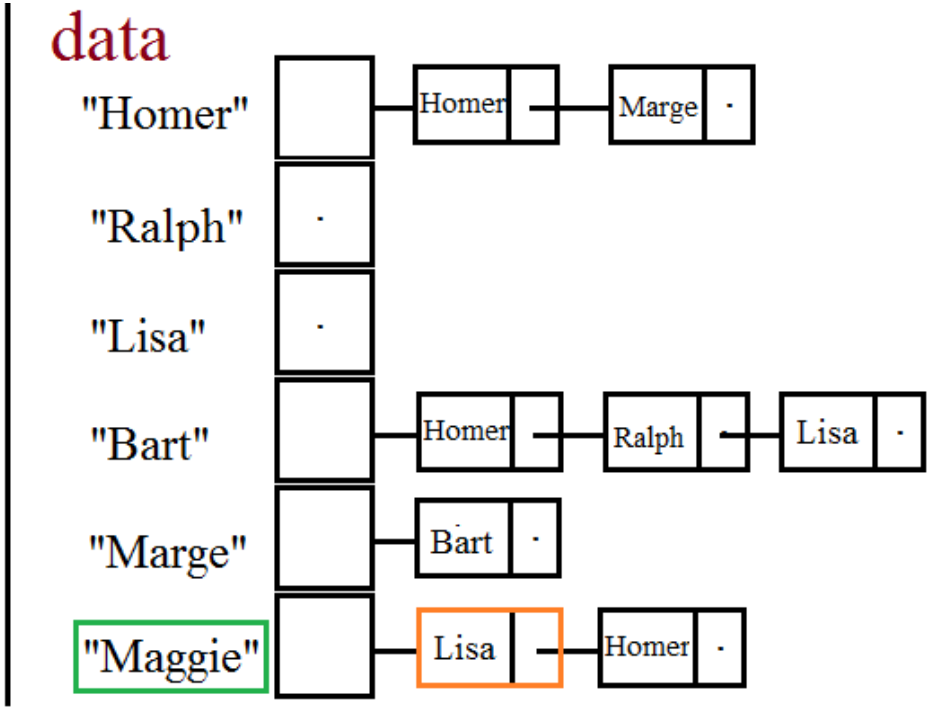


Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

Example:

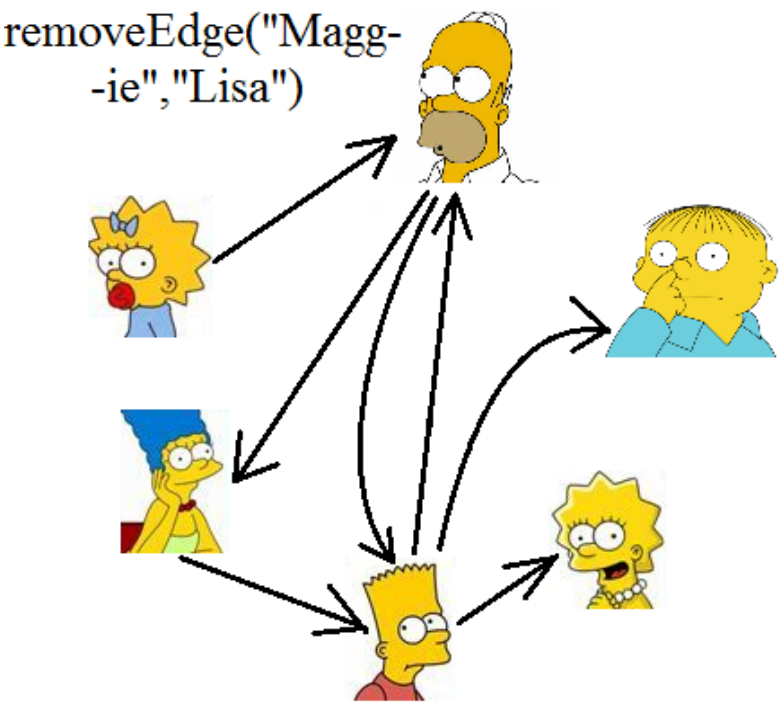


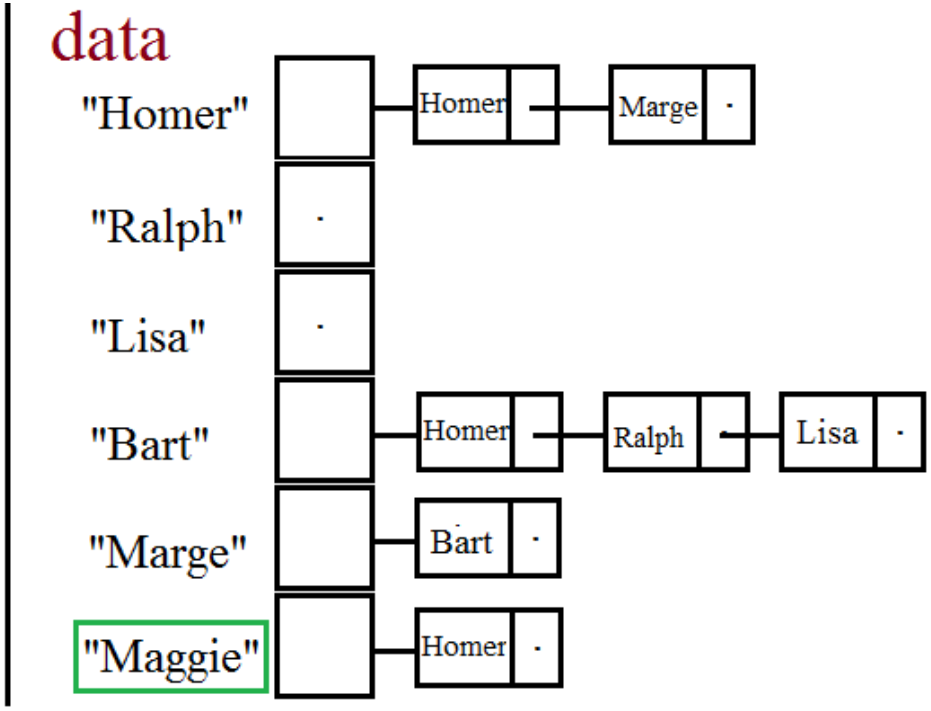


Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

Example:

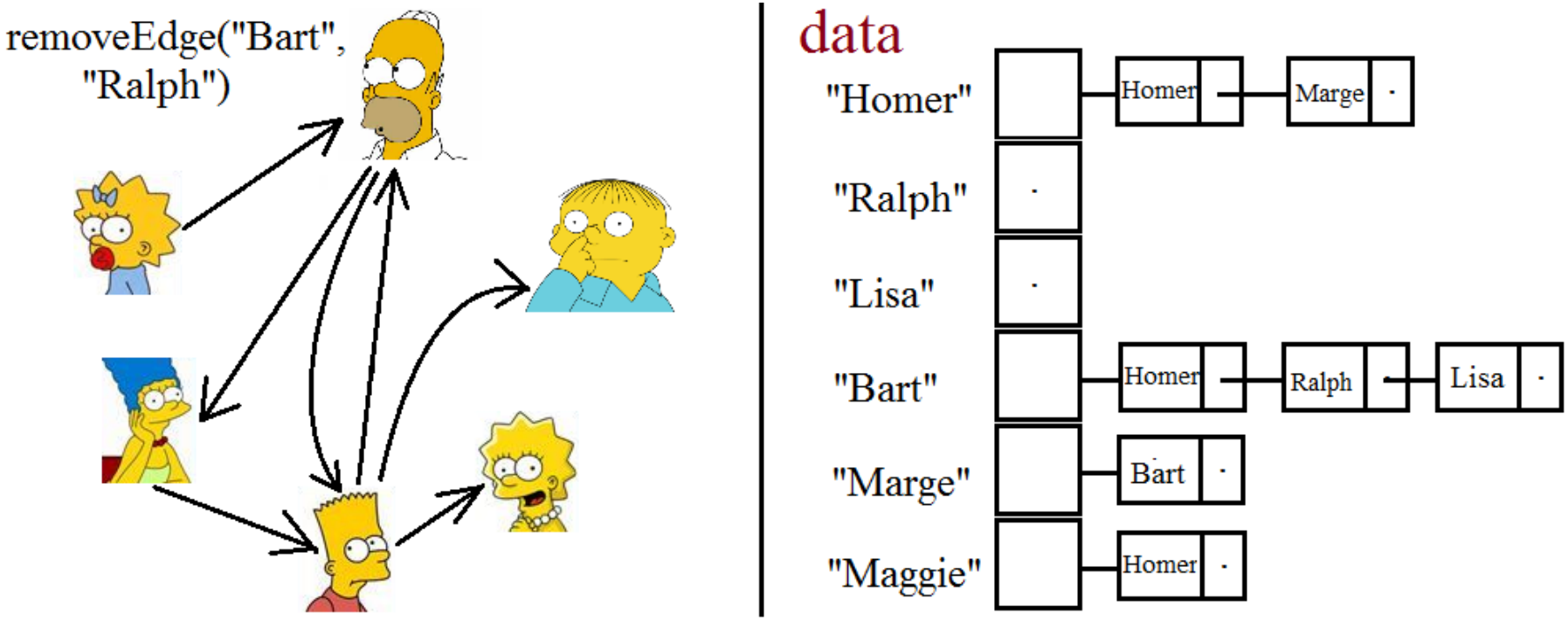




Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

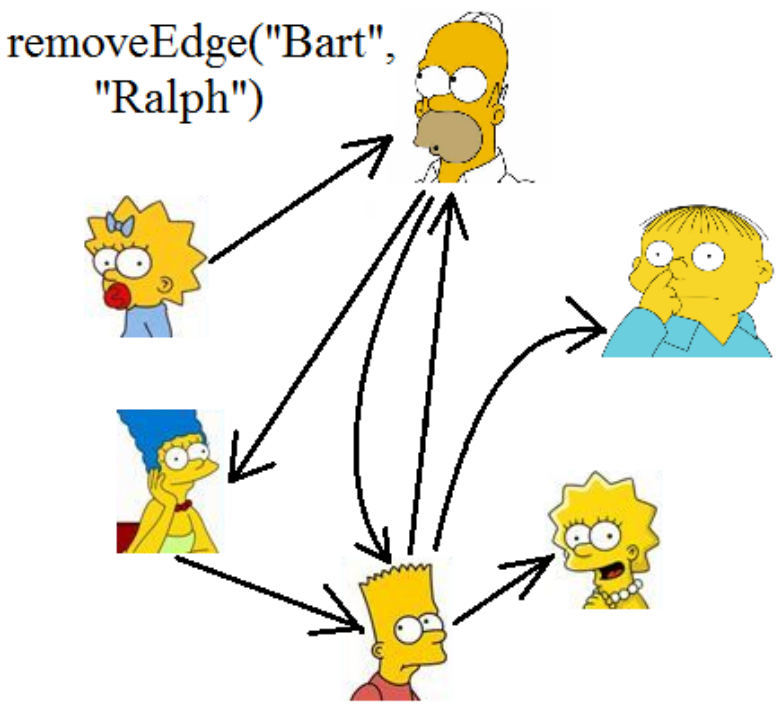
Example:

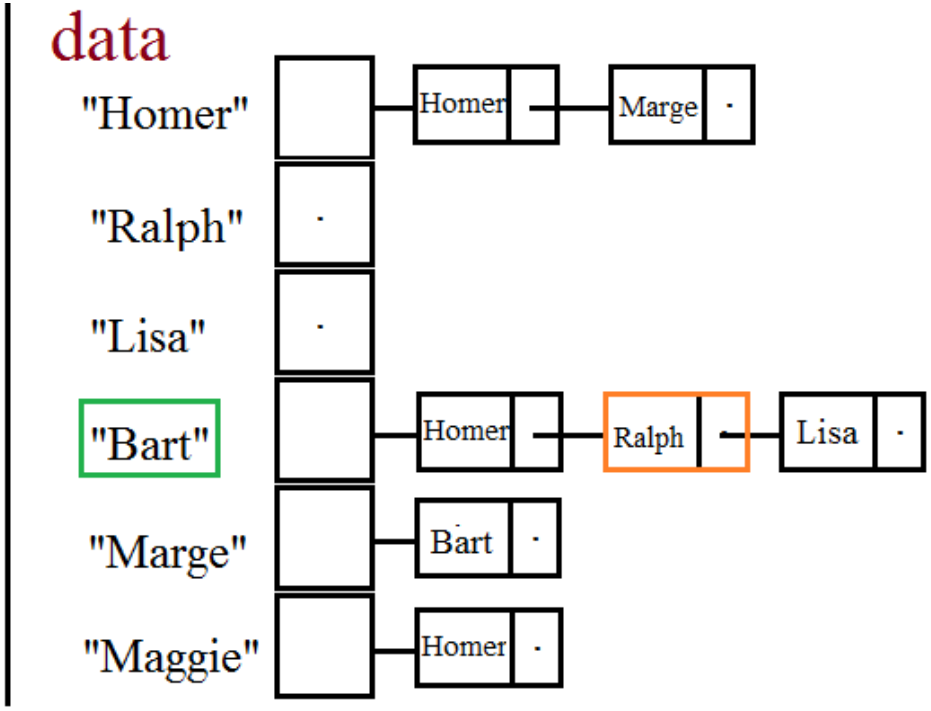


Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

Example:

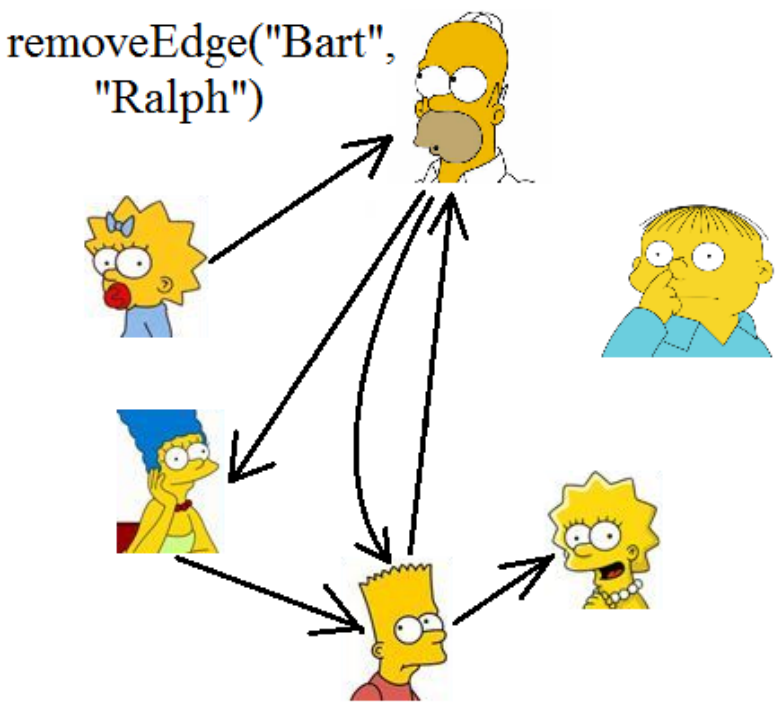


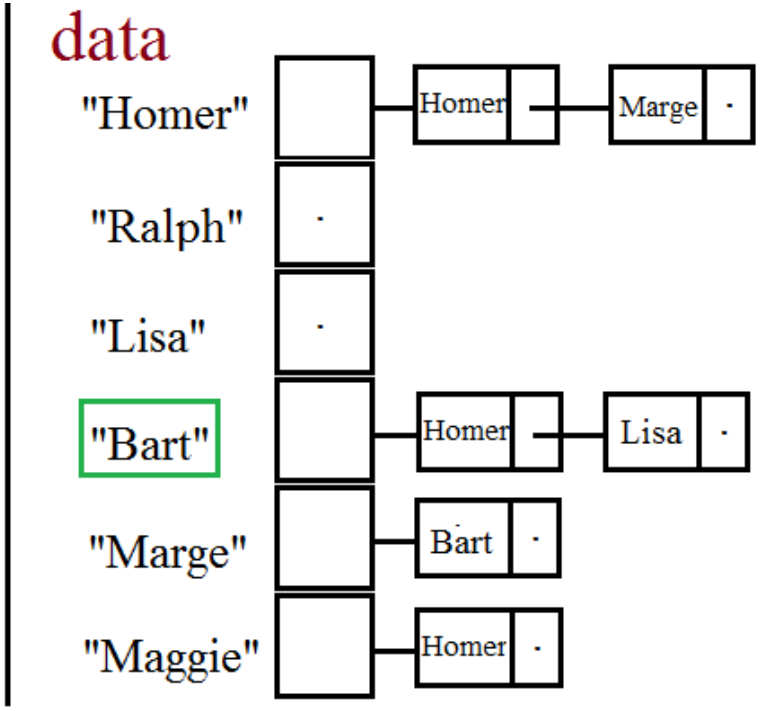


Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

Example:





Implementation 2: Adjacency List

Updating: `removeEdge(N node1, N node2)`

```
public N removeEdge(N node1, N node2){
    if(!data.contains(node1) || !data.contains(node2))
        return null;
    List<N> list= data.getValue(node1);
    list.remove(node2))
}
```

Implementation 2: Adjacency List

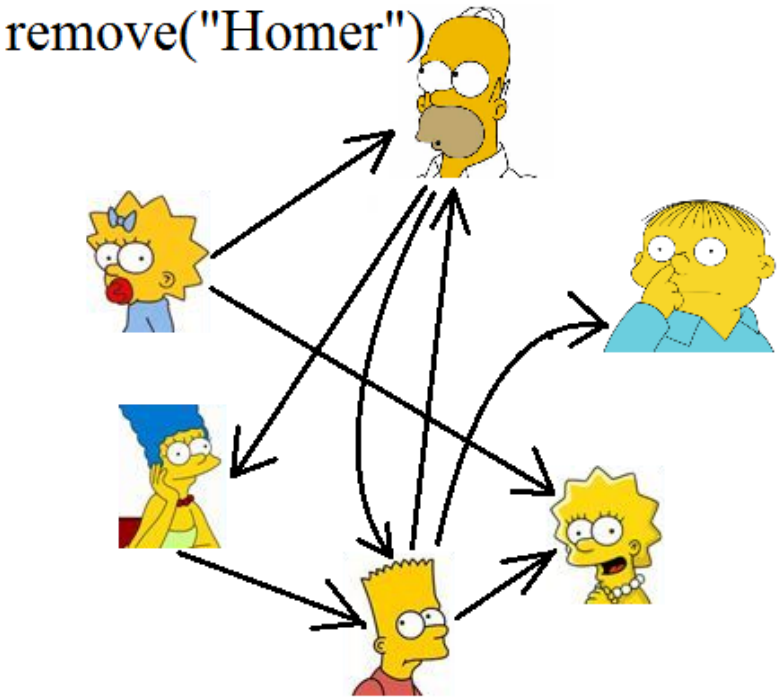
Updating: `remove(N node)`

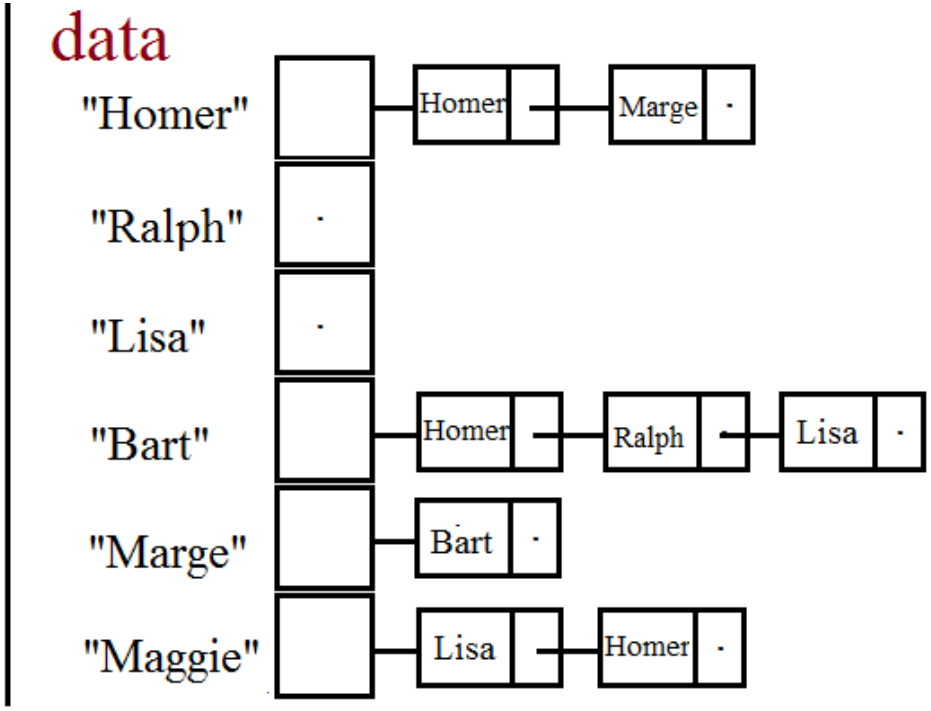
1. If graph doesn't contain `node`, return null
2. Go over all nodes in the graph; for each node v, remove the edge to (v,`node`) if it exists
3. Remove the `node` and its list from `data`
4. return `node`

Implementation 2: Adjacency List

Updating: `remove(N node)`

Example:

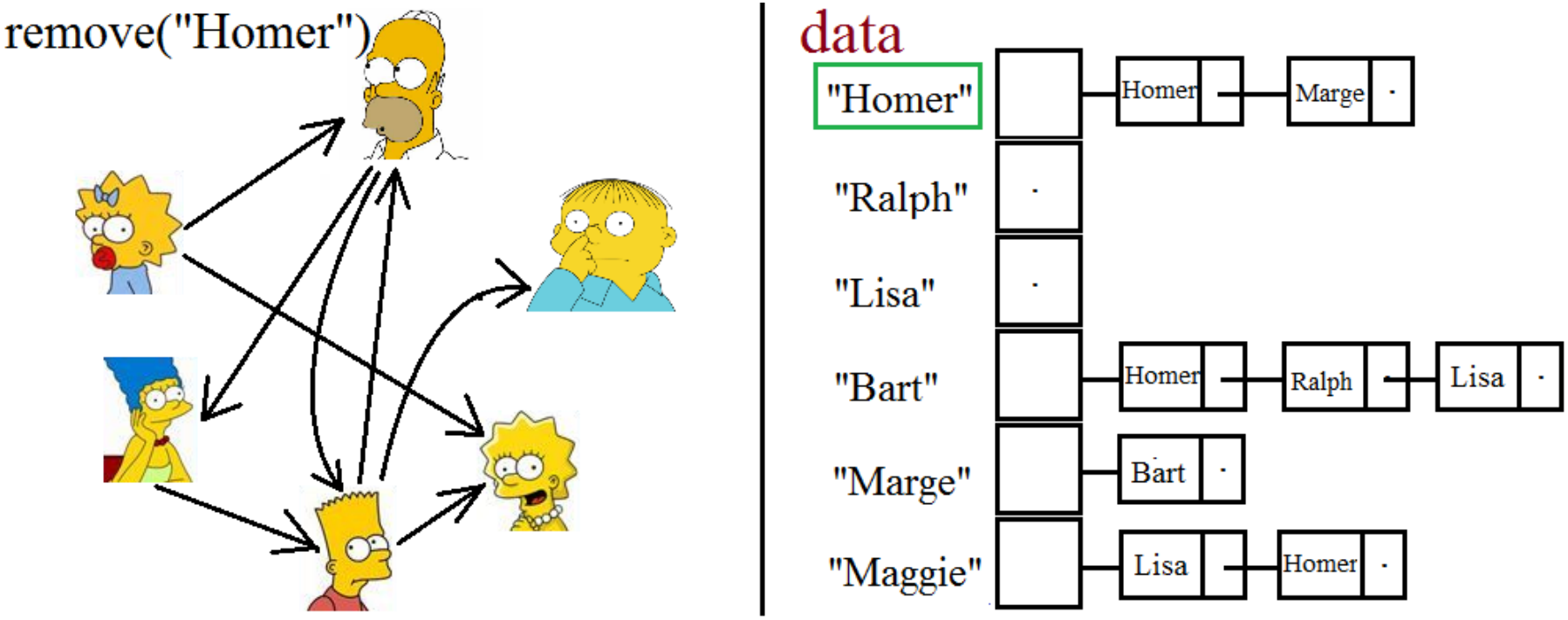




Implementation 2: Adjacency List

Updating: `remove(N node)`

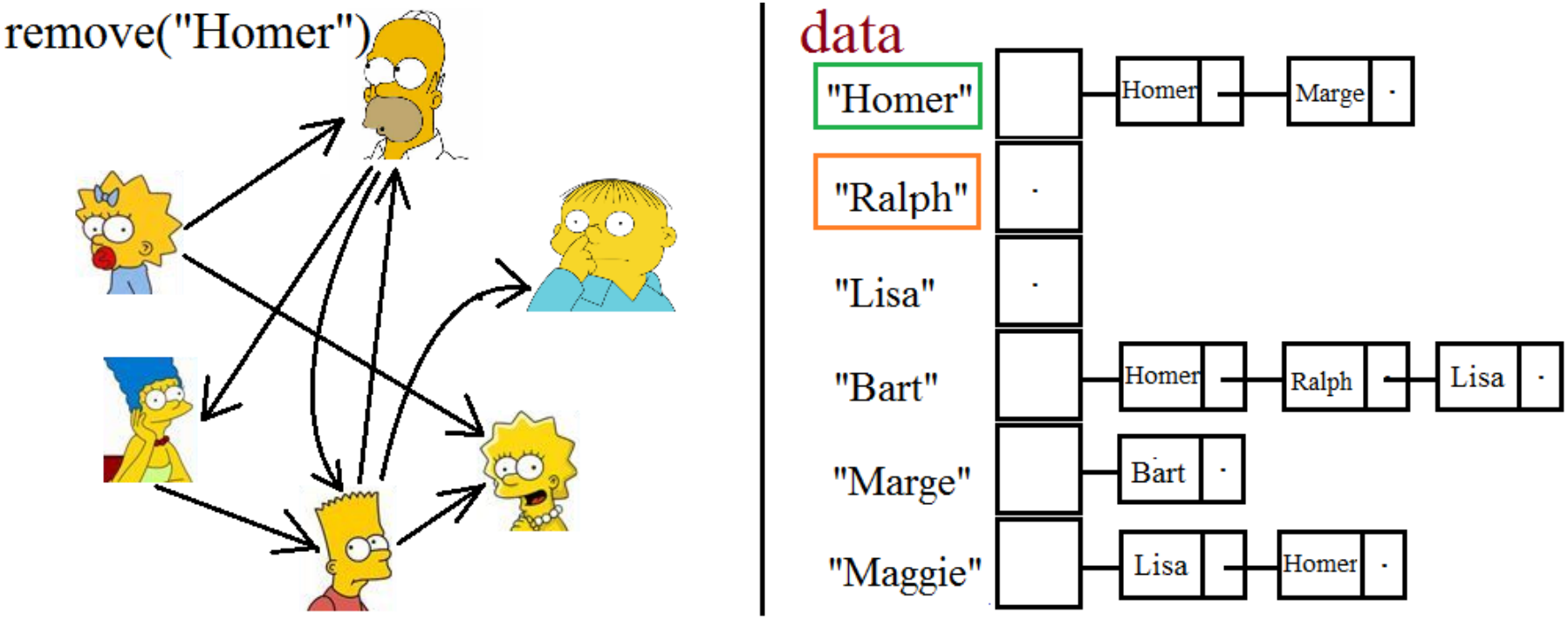
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

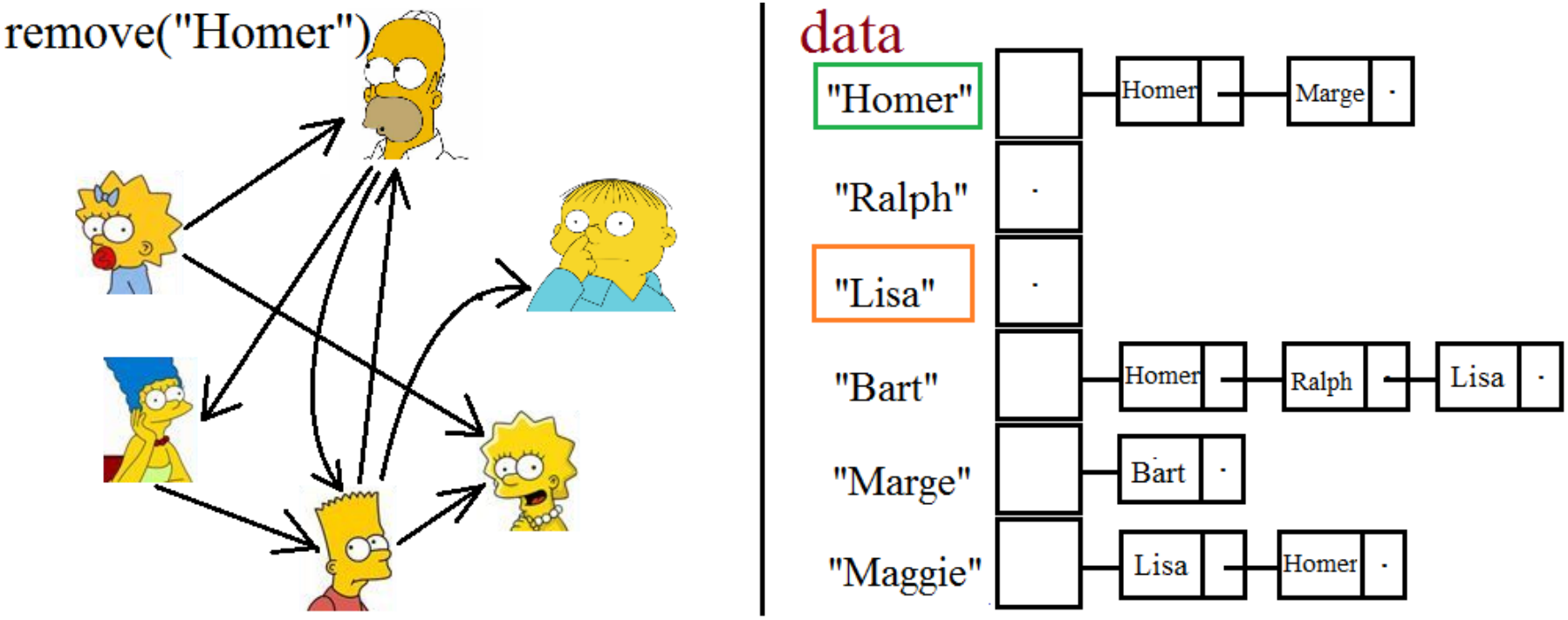
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

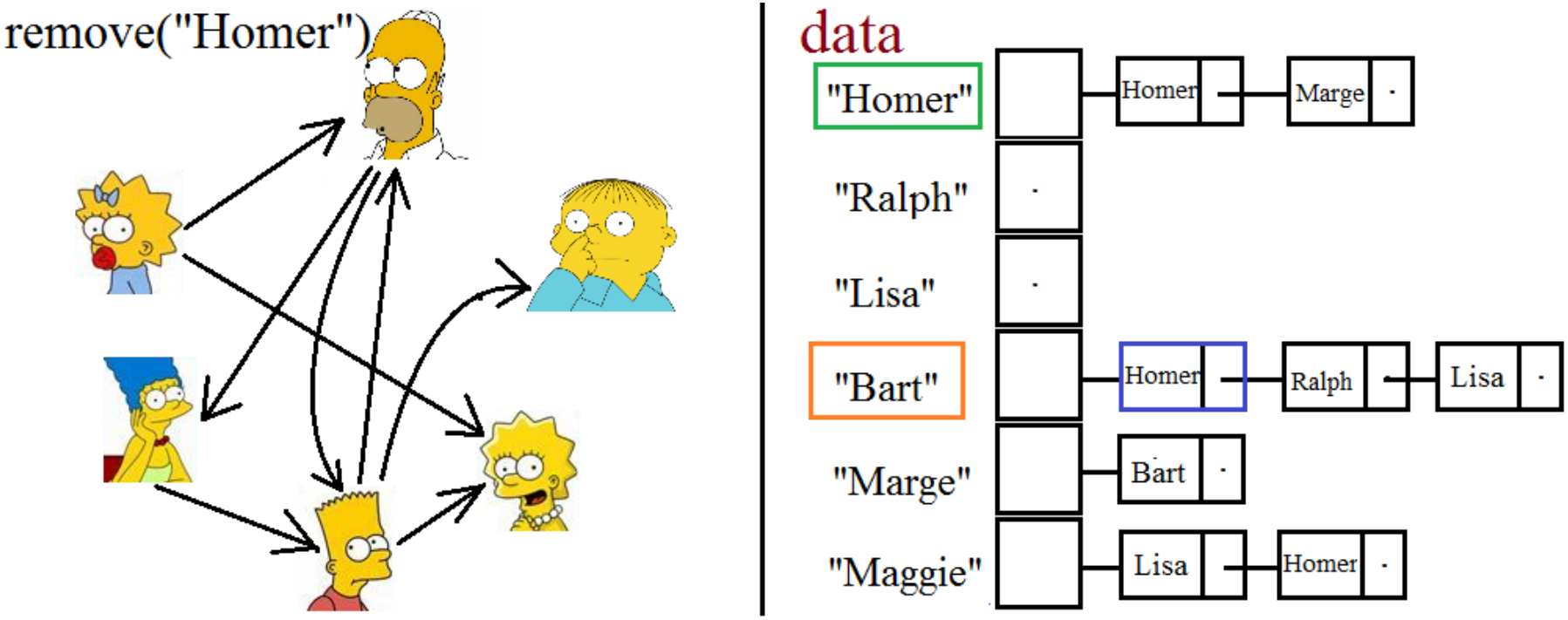
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

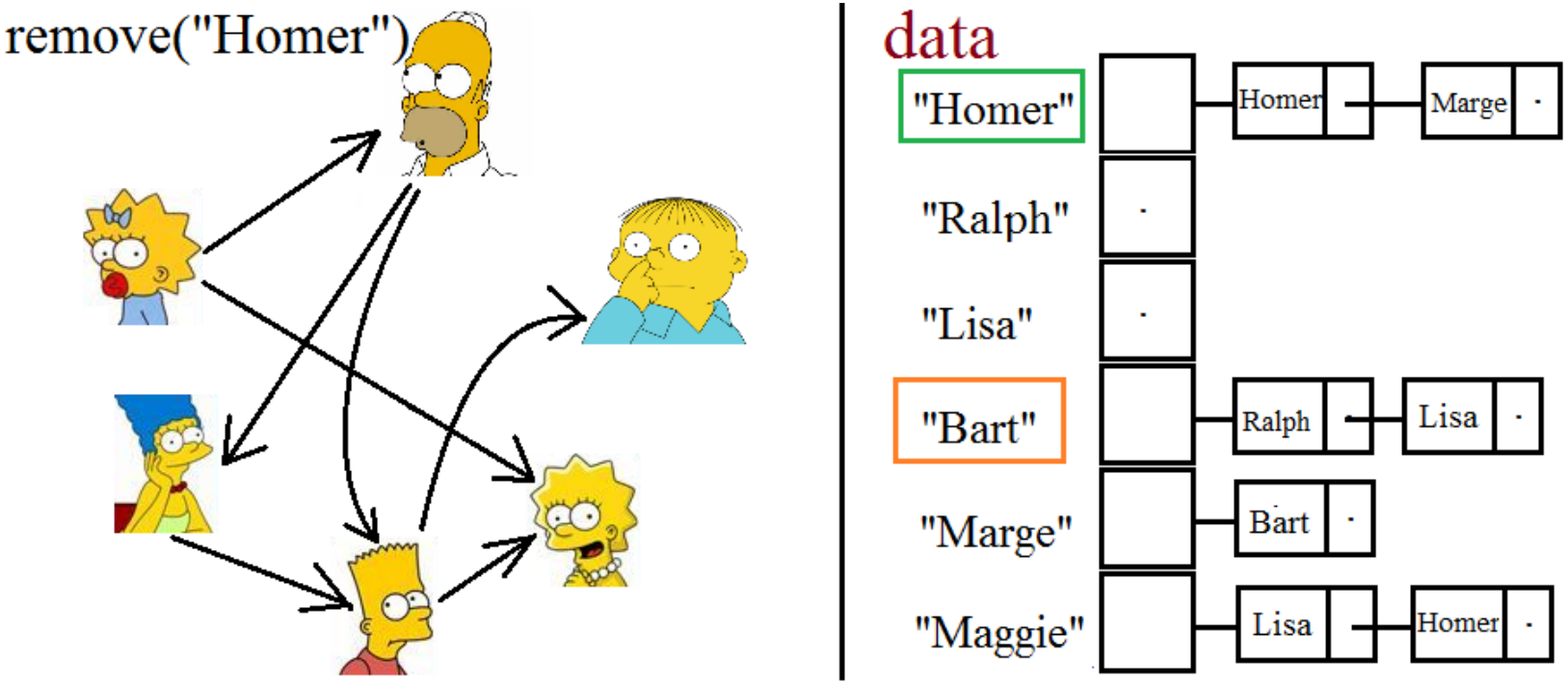
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

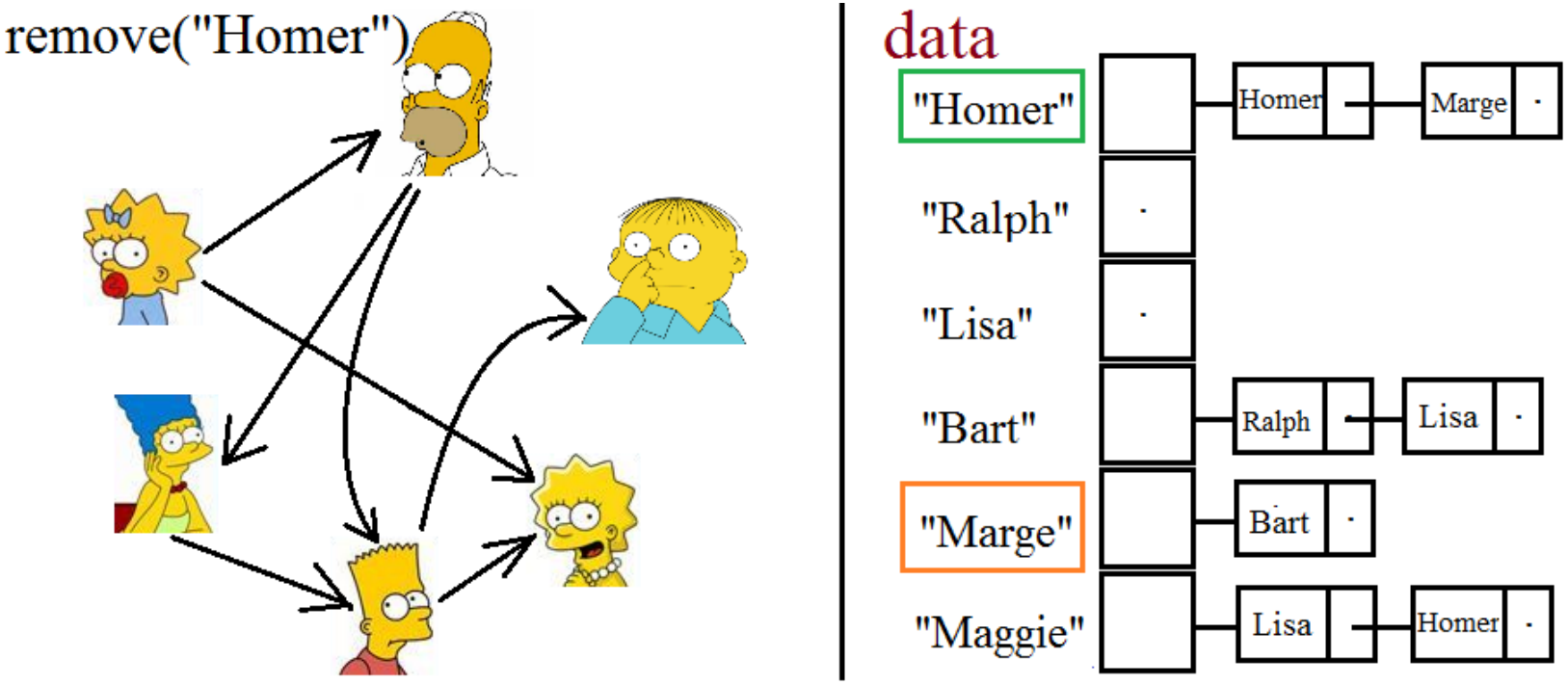
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

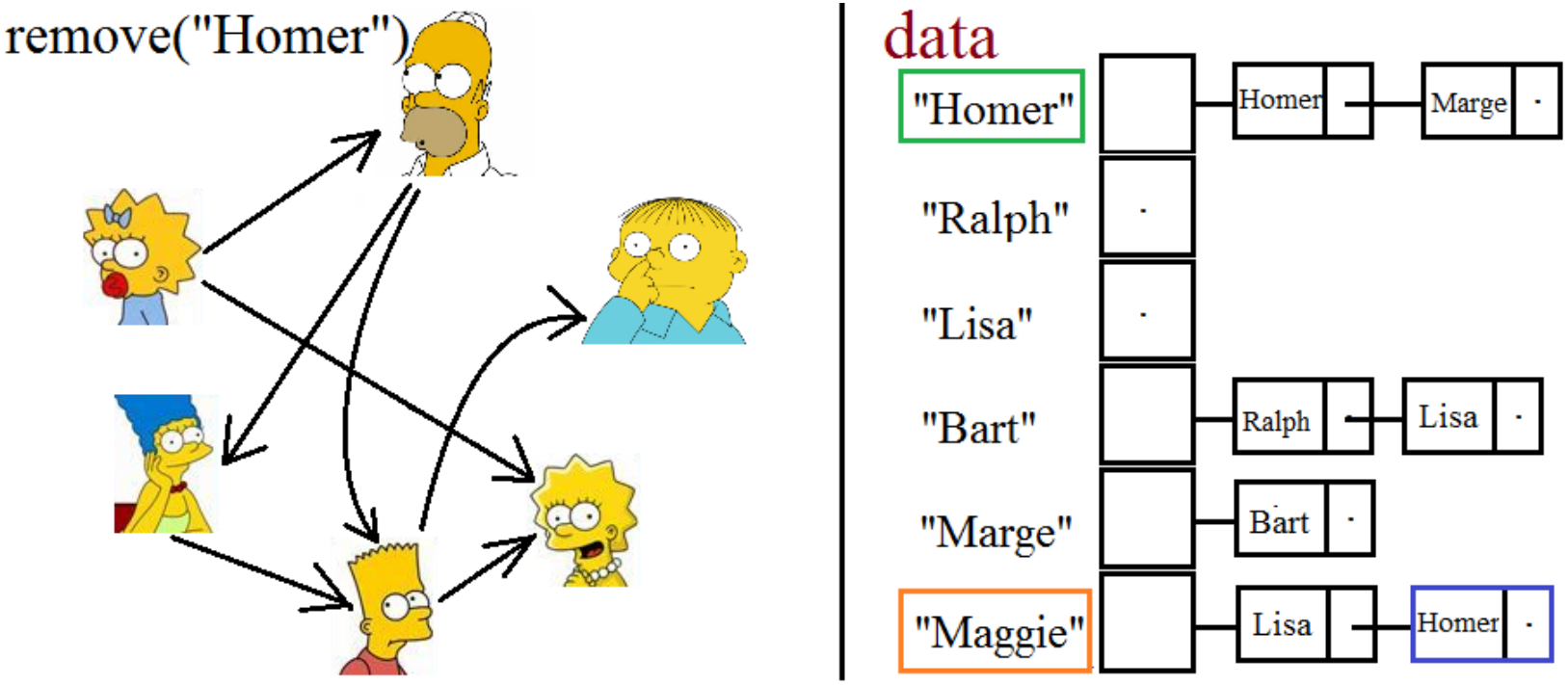
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

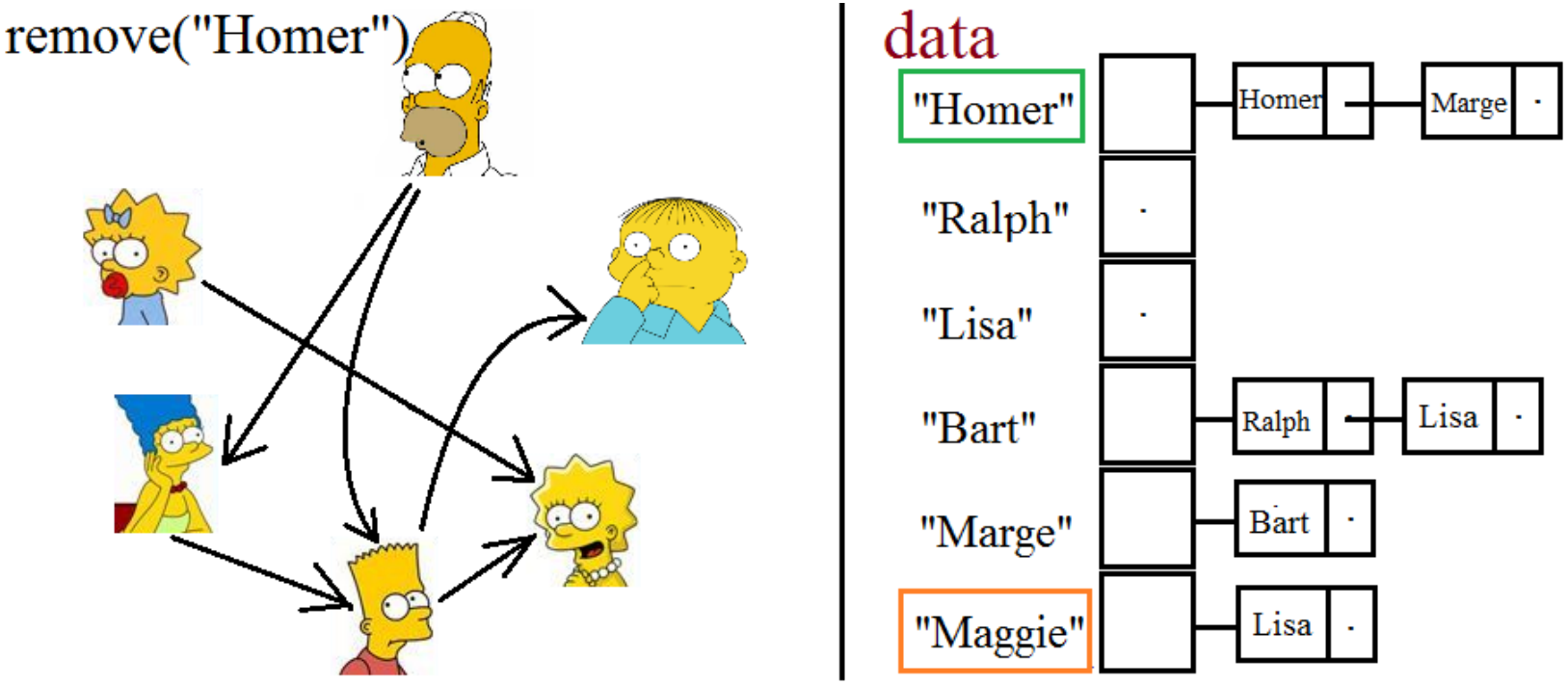
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

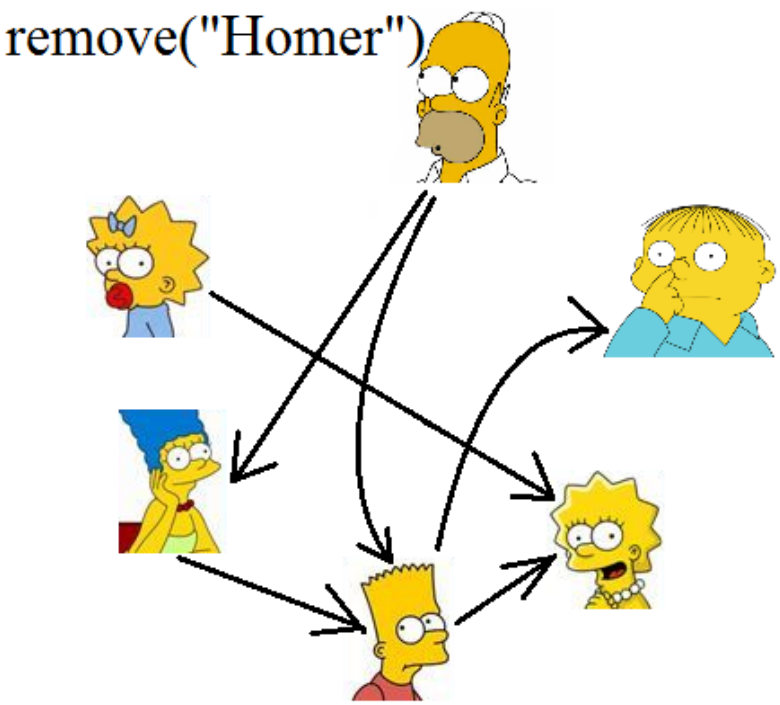
Example:

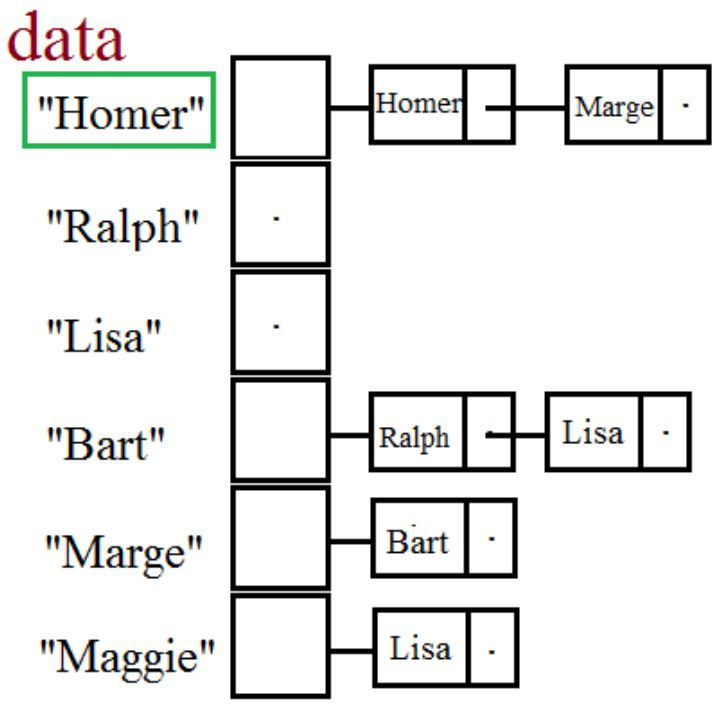


Implementation 2: Adjacency List

Updating: `remove(N node)`

Example:

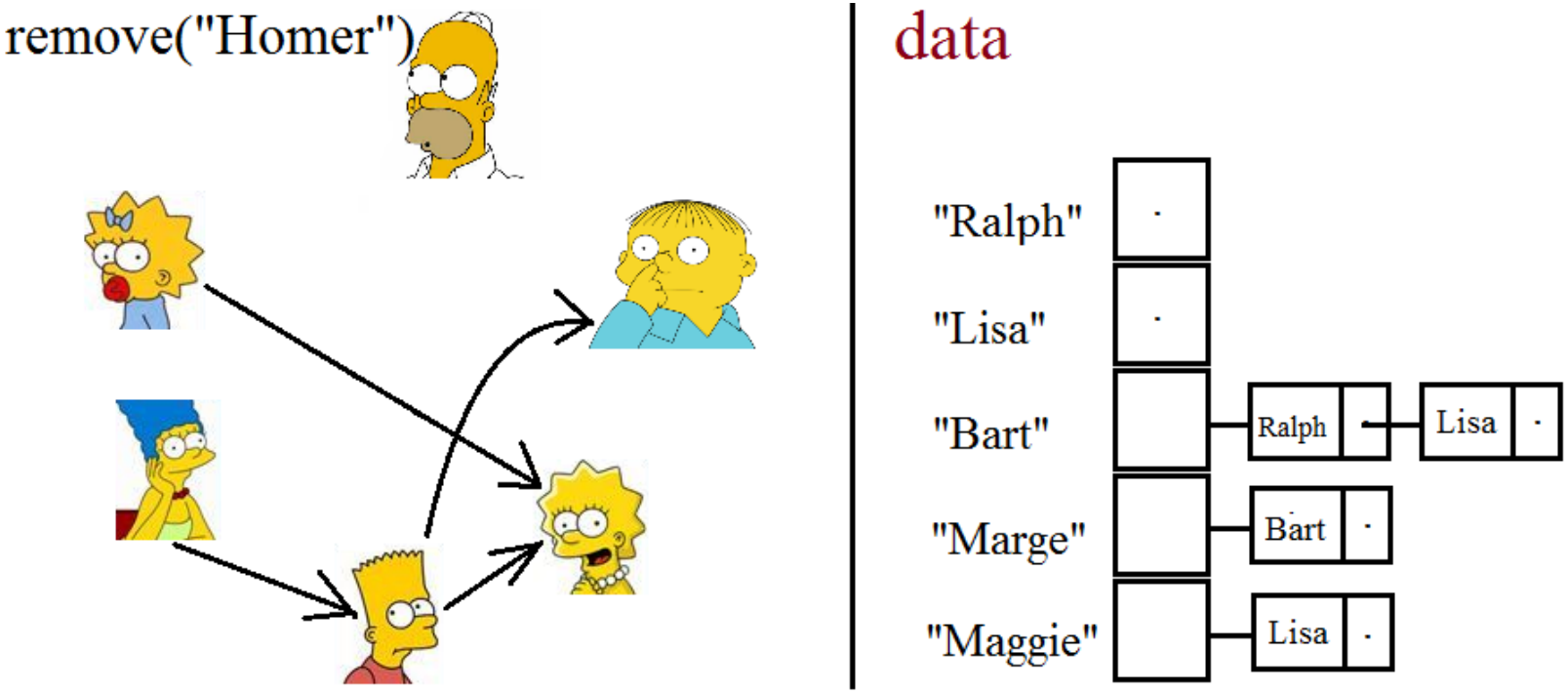




Implementation 2: Adjacency List

Updating: `remove(N node)`

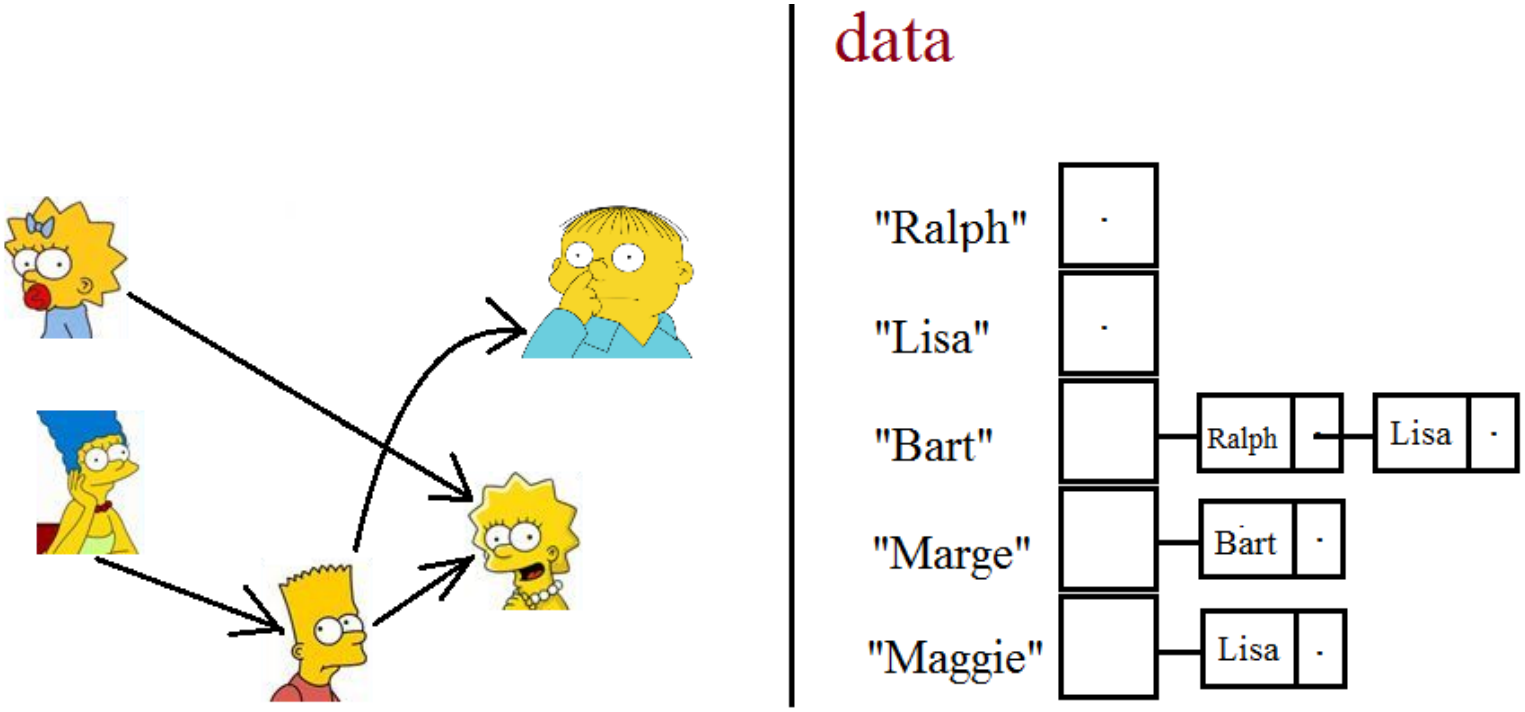
Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

Example:



Implementation 2: Adjacency List

Updating: `remove(N node)`

```
public E remove(N node){
    if (!data.contains(node)) return null;
    Iterator<N> it=iterator();
    while(it.hasNext()){
        N n2=it.next();
        if (!node.equals(n2)) removeEdge(n2,node);
    }
    data.remove(node);
    return node;
}
```

Implementation 2: Adjacency List

Queries:

- `contains(N node)`

  ```
  return data.contains(node);
  ```

- `containsEdge(N node1, N node2)`

  ```
  List<N> list=dict.getValue(node1);
  return list.contains(node2);
  ```

- `iterator()`

  ```
  return data.keySet().iterator();
  ```

Implementation 2: Adjacency List

Time Complexity:

- Assume an efficient hash table implementation.

Operations	Time Complexity
contains(node)	$O(1)$
containsEdge(node1, node2)	$O(n)$
add(node)	$O(1)$
remove(node)	$O(m+n)$
addEdge(node1, node2)	$O(n)$
removeEdge(node1, node2)	$O(n)$

Advantages: Space efficient $O(m+n)$ space taken
Efficient for obtaining outgoing edges.

Disadvantage: Inefficient for node removal
Inefficient for obtaining incoming edges.