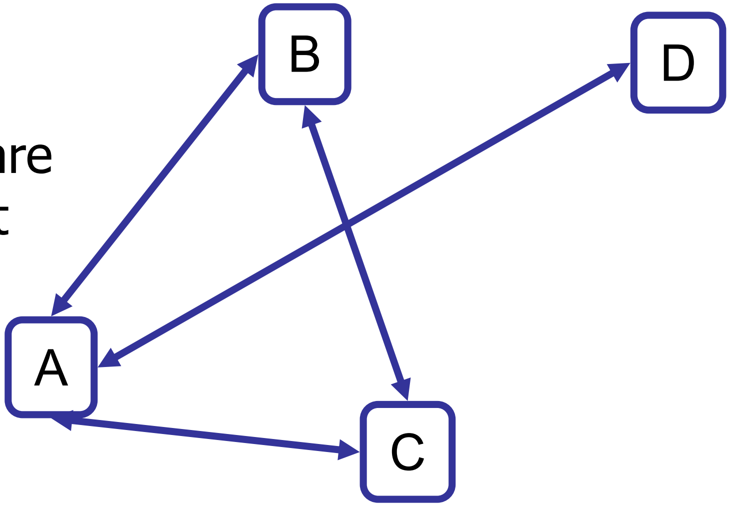
# GRAFI

## ESERCIZIO

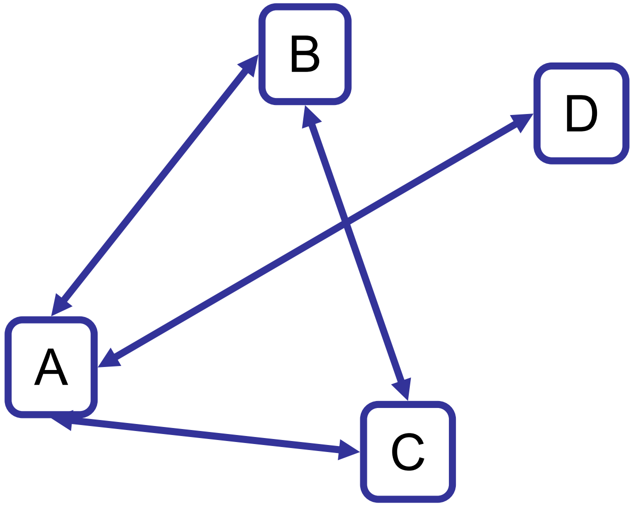
# Grafi (ADT)

```
public class Graph {
  // OVERVIEW:
  //      A Graph is a mutable type that
  //      represents an undirected
  //      graph.  It consists of nodes that are
  //      named by Strings, and edges that
  //      connect a pair of nodes.
  //      A typical Graph is:
  //            < Nodes, Edges >
  //       where
  //         Nodes = { n1, n2, …, nm }
  //       and
  //         Edges = { {from_1, to_1},
  //                         …, {from_n, to_n} }
```



Nodes = { A, B, C, D }
Edges = { { A, B }, { A, C },
{ B, C }, { A, D } }

# Rappresentazione



- Insiemi di Nodes e Edges

  Nodes = { A, B, C, D }

  Edges = { <A, B>, <A, C>,  <A, D>, <B, C> }

- Insiemi di Nodes e Neighbors

  Graph = { <A, {B, C, D}>, <B, {A, C}>,
  <C, {A, B}>, <D, {A}> }

  <nodo, insieme dei nodi connessi>

# Altra idea

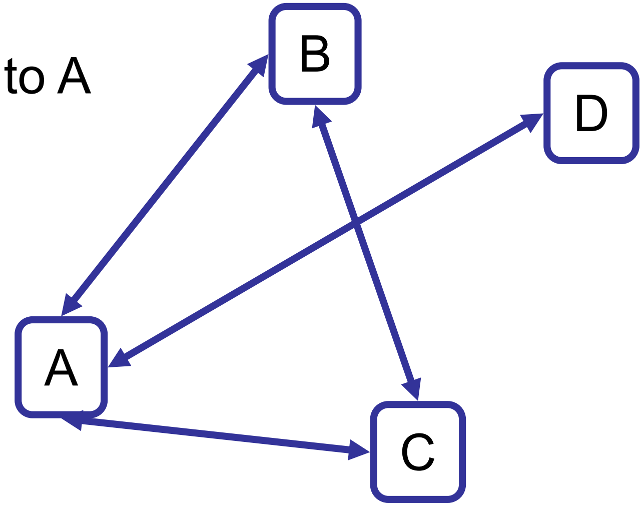
- Insieme di nodi e la matrice booleana delle connessioni

Nodes = [ A, B, C, D ]

No edge from A to A

```
Edges = [ [0 1 1 1 ]
          [ 1 0 1 0 ]
          [ 1 1 0 0 ]
          [ 1 0 0 0 ] ]
```

Edge from B to C



# Implementazione 1

```
class Edge {
      // OVERVIEW: Pair type for representing an edge.
     String node1, node2;
     Edge (String n1, String n2) { node1 = n1; node2 = n2; }
}


class Graph {
     // OVERVIEW: A Graph is a mutable type that represents an …

     Vector<String> nodes; // A Vector of String objects
     Vector<Edge> edges; // A Vector of Edge object


      …
}
```

# Rep Invariant

```
class Edge {
    String node1, node 2;
}
class Graph {
    Vector<String> nodes;
    Vector<Edge> edges; … }
```

RI (c) = c.nodes != null && c.edges != null

&& !c.nodes.containsNull && !c.edges.containsNull

&& elements of c.nodes are String objects

&& elements of c.edges are Edge objects

&& no duplicates in c.nodes

E’ precisa?

&& no duplicates in c.edges

&& every node mentioned in c.edges is also in c.nodes

# Abstraction Function

```
public class Graph {
   // OVERVIEW:
   //      A Graph is a mutable type that
   //      represents an undirected
   //      graph.  It consists of nodes that are
   //      named by Strings, and edges that
   //      connect a pair of nodes.
   //      A typical Graph is:
   //             < Nodes, Edges >
   //       where
   //         Nodes = { n1, n2, …, nm }
   //       and
   //         Edges = { {from_1, to_1},
   //                          …, {from_n, to_n} }
```

- Da rep alla rapresentazione astratta (overview)

AF (c) =

< Nodes, Edges >

tali che …

# Abstraction Function

```
class Edge {
    String node1, node 2;
}
class Graph {
   Vector<String> nodes;
    Vector<Edge> edges; … }
```

AF (c) = < Nodes, Edges > tali che

Nodes = { c.nodes[i] | 0 <= i < c.nodes.size () }

Nodi = c.nodes Vector

Edges = { { c.edges[i].node1, c.edges[i].node2 }
| 0 <= i < c.edges.size () }

Archi= c.edges Vector

# Costruttore

```
class Edge {
    String node1, node 2;
}
class Graph {
    Vector<String> nodes;
    Vector<Edge> edges; … }
```

```
public Graph ()
   // EFFECTS: Initializes this to a graph with no nodes or
   //    edges: < {}, {} >.

    nodes = new Vector<String> ();
    edges = new Vector<Edge> ();
 }
```

Soddisfa rep invariant?

# Metodo addNode

```
class Edge {
    String node1, node 2;
}
class Graph {
   Vector<String> nodes;
    Vector<Edge> edges; … }
```

```
public void addNode (String name) {
    // REQUIRES: name is not the name of a node in this
    // MODIFIES: this
    // EFFECTS: adds a node named name to this:
   //     this_post = < this_pre.nodes U { name }, this_pre.edges >
    nodes.addElement (name);
 }
```

Rispetta rep invariant?

# Metodo addEdge

```
class Edge {
    String node1, node 2;
}
class Graph {
    Vector<String> nodes;
    Vector<Edge> edges; … }
```

```
public void addEdge (String fnode, String tnode)
    // REQUIRES: fnode and tnode are names of nodes in this.
    // MODIFIES: this
    // EFFECTS: Adds an edge from fnode to tnode to this:
    //      this_post = < this_pre.nodes,
    //                           this_pre.edges U { {fnode, tnode} } >
```

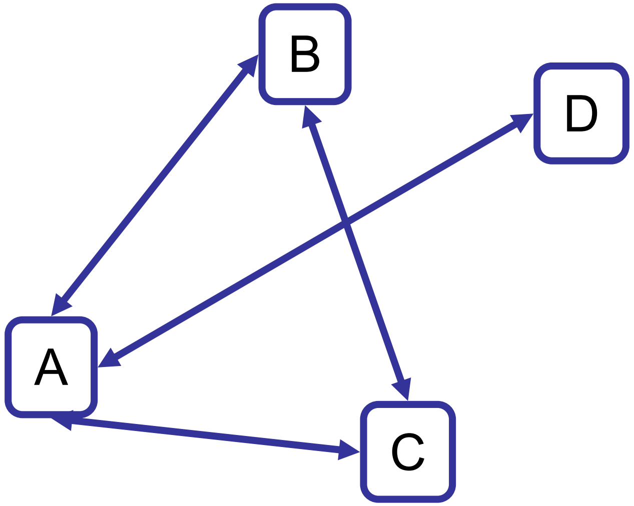
Anche questo metodo rispetta rep invariant?

# Metodo getNeighbors

```
class Edge {
    String node1, node 2;
}
class Graph {
    Vector<String> nodes;
    Vector<Edge> edges; … }
```

```
public StringSet getNeighbors (String node)
   // REQUIRES: node is a node in this
   // EFFECTS: Returns the StringSet consisting of all nodes in this
   //    that are directly connected to node:
   //       result = { n | {node, n} is in this.edges
   StringSet res = new StringSet ();
   Enumeration edgeenum = edges.elements ();
   while (edgeenum.hasMoreElements ()) {
      Edge e = (Edge) edgeenum.nextElement ();
      if (e.node1.equals (node)) { res.insert (e.node2); }
      else if (e.node2.equals (node)) { res.insert (e.node1); }
}
```

# Rappresentazione



- Insiemi di Nodes e Edges

  Nodes = { A, B, C, D }

  Edges = { <A, B>, <A, C>, <A, D>, <B, C> }

- Insiemi di Nodes e Neighbors

  Graph = { <A, {B, C, D}>, <B, {A, C}>,

  <C, {A, B}>, <D, {A}> }

  <nodo, insieme dei nodi connessi>

# Implementazione 2

```
class NodeNeighbors {
     // OVERVIEW: Pair type for representing an edge.
    String node;
    StringSet neighbors; // A Set of String objects
    NodeNeighbors (String n) { node = n; neighbors = new StringSet (); }
}
class Graph {
    // OVERVIEW: A Graph is a mutable type that represents an …
    Vector<NodeNeighbors> nodes; // A Vector of NodeNeighbors objects
     …
}
```

# Rep Invariant

```
class NodeNeighbors {
    String node;
    StringSet neighbors; }
class Graph {
    Vector<NodeNeighbors > nodes; }
```

```
RI (c) =    c.nodes != null
        && !c.nodes.containsNull
        && elements of c.nodes are NodeNeighbors objects

        && no duplicates in c.nodes
        && for each node in c.nodes, each node in
                c.nodes[i].neighbors is a node in c.nodes
                c.nodes[i].neighbors does not contain duplicates
```

# Abstraction Function

```
class NodeNeighbors {
    String node;
    StringSet neighbors;
}
class Graph {
  Vector<NodeNeighbors > nodes; }
```

AF (c) = < Nodes, Edges > where

Nodes = { c.nodes[i].node | 0 <= i < c.nodes.size () }

The set of nodes is the elements of the c.nodes Vector

```
Edges = { { c.nodes[i].node, c.nodes[i].neighbors[e] }
              | 0 <= i < c.nodes.size (),
                    0 <= e <= c.nodes[i].neighbors.size ()
             }
```

# Constructor

```
class NodeNeighbors {
    String node;
    Vector neighbors; // A Vector of String objects
}
class Graph {
    Vector<NodeNeighbors > nodes; }
```

```
public Graph ()
   // EFFECTS: Initializes this to a graph with no nodes or
   //    edges: < {}, {} >.

    nodes = new Vector ();
 }
```

# addNode

```
class NodeNeighbors {
   String node;
   StringSet neighbors;
}
class Graph {
   Vector<NodeNeighbors > nodes; }
}
```

```
public void addNode (String name) {
   // REQUIRES: name is not the name of a node in this
   // MODIFIES: this
   // EFFECTS: adds a node named name to this:
  //     this_post = < this_pre.nodes U { name }, this_pre.edges >
   nodes.addElement (new NodeNeighbors (name));
 }
```

# addEdge

```
class NodeNeighbors {
    String node;
    StringSet neighbors;
}
class Graph {
   Vector<NodeNeighbors > nodes; }
```

```
public void addEdge (String fnode, String tnode)
    // REQUIRES: fnode and tnode are names of nodes in this.
    // MODIFIES: this
    // EFFECTS: Adds an edge from fnode to tnode to this:
    //        this_post = < this_pre.nodes,
    //                          this_pre.edges U { {fnode, tnode} } >
    NodeNeighbors n1 = lookupNode (fnode);
    NodeNeighbors n2 = lookupNode (tnode);
    n1.neighbors.insert (tnode);
    n2.neighbors.insert (fnode);
}
```

Da realizzare
lookupNode
a

# getNeighbors

```
class NodeNeighbors {
    String node;
    StringSet neighbors;
}
class Graph {
   Vector<NodeNeighbors > nodes; }
```

```
public StringSet getNeighbors (String node)
   // REQUIRES: node is a node in this
   // EFFECTS: Returns the StringSet consisting of all nodes in this
   //    that are directly connected to node:
   //       \result = { n | {node, n} is in this.edges
   NodeNeighbors n = lookupNode (node);
   return n.neighbors;
}
```

Esponiamo rep!

# Esporre Rep!!!!

```
Graph g = new Graph ();
g.addNode (“A”);
g.addNode (“B”);
g.addEdge (“A”, “B”);
StringSet neighbors = g.getNeighbors (“A”);
neighbors.insert (“C”);
```

# getNeighbors

```
class NodeNeighbors {
    String node;
    StringSet neighbors;
}
class Graph {
    Vector nodes; // A Vector of NodeNeighbors objects
}
```

```
public StringSet getNeighbors (String node)
   // REQUIRES: node is a node in this
   // EFFECTS: Returns the StringSet consisting of all nodes in this
   //    that are directly connected to node:
   //       result = { n | {node, n} is in this.edges

   NodeNeighbors n = lookupNode (node);
   return n.neighbors;copy ();
}
```