Programmazione Orientata Agli Oggetti

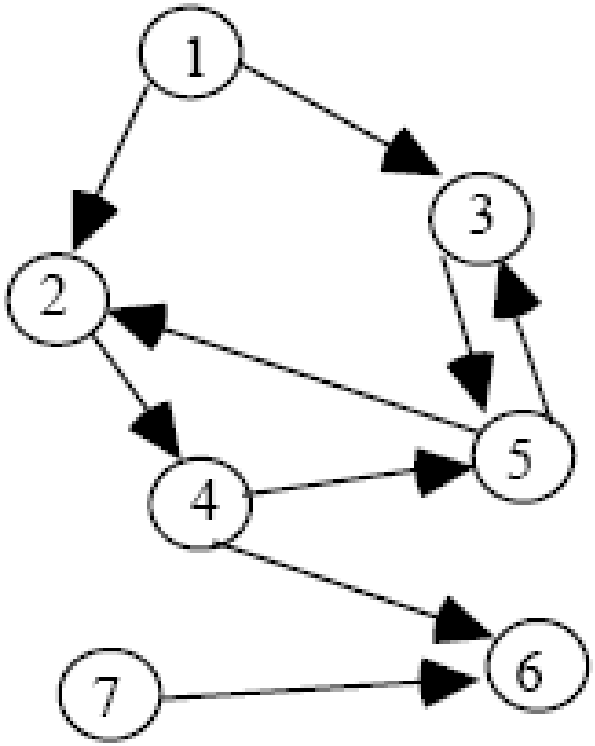
Grafi, Temi di Esame

Libero Nigro

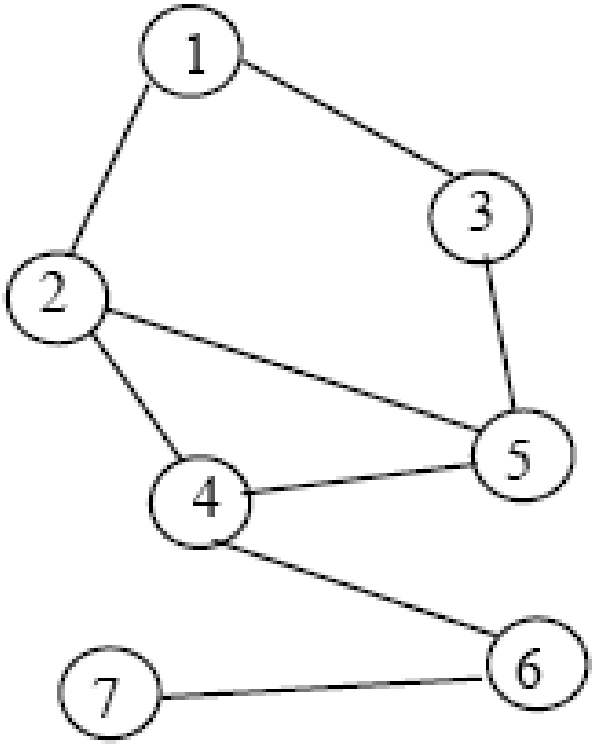
Introduzione ai grafi

- I grafi sono strutture dati molto importanti per le applicazioni. Gli alberi sono un caso particolare dei grafi.
- Un grafo esprime una *rete di relazioni* tra oggetti.
- Formalmente, un grafo è una coppia G={N,A} dove N è un *insieme di nodi* (o vertici) ed A un *insieme di archi*, ossia coppie <ni,nj> con ni,nj $\in$ N.
- Un arco collega due nodi in relazione tra loro. Dato l'arco <ni,nj> si dice che *nj è adiacente a ni* (o raggiungibile a partire da ni in un "sol passo").
- L'insieme degli archi costituisce matematicamente una *relazione*, ossia è un sotto insieme delle possibili coppie <ni,nj> del prodotto cartesiano NxN.
- Un *grafo* si dice *orientato* (o diretto) se un arco <ni,nj> specifica che da ni si può andare ad nj ma non viceversa. Graficamente, un arco di un grafo orientato è una freccia che fuoriesce dal nodo ni e punta al nodo adiacente nj. Se è richiesto che da nj si possa ritornare ad ni, occorre che esista *anche* l'altro arco <nj,ni>.
- Un *grafo* si dice *non orientato* se un arco ha significato bidirezionale, ossia è doppio: <ni,nj> indica che da ni si può andare su nj e viceversa. Ogni nodo partecipante dell'arco è adiacente all'altro nodo dell'arco.

Esempi di grafi



a) Grafo orientato



b) Grafo non orientato

- Entrambi gli esempi utilizzano nodi etichettati con numeri interi. Tuttavia si possono utilizzare anche altri tipi di etichette (es. stringhe); si pensi ad un grafo che coinvolge città, ed i cui archi esprimono collegamenti stradali tra città. Un nome potrebbe essere la sigla di una provincia (CS, RC, TO etc).
- Entrambi i grafi di esempio hanno come insieme di nodi N={1,2,3,4,5,6,7}.
- Il grafo orientato ha come insieme di archi:
  A={<1,2>,<1,3>,<2,4>,<3,5>,<4,5>,<5,2>,<5,3>,<4,6>,<7,6>}
- In modo analogo si possono enumerare gli archi (stavolta bidirezionali) del secondo esempio relativo ad un grafo non orientato.
- Un grafo può essere utilizzato, ad es., per:
  - esprimere una mappa di città e relativi collegamenti stradali;
  - esprimere i comuni di una provincia e le relazioni di confinanza: i nodi sono i comuni, gli archi riflettono le confinanze;
  - esprimere la relazione di precedenza (propedeuticità) tra corsi universitari: ogni corso è un nodo, una freccia dal corso ci al corso cj specifica che ci è precondizione per sostenere cj;
  - esprimere le relazioni di parentela tra persone. Si nota che un albero binario (o n-nario) è un caso particolare di grafo orientato
  - etc.

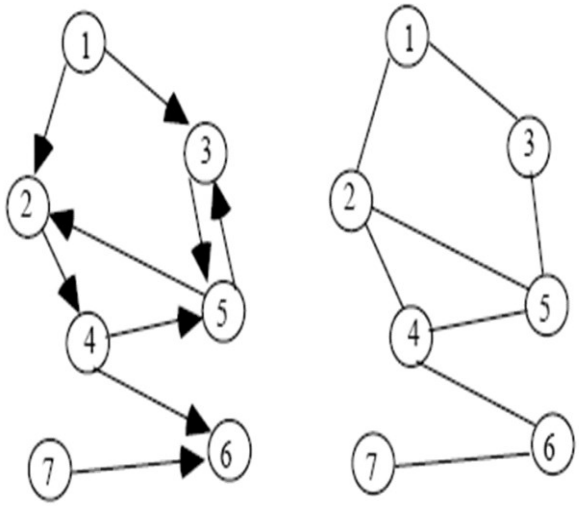
Altri concetti e definizioni

Un **cammino** (o percorso) in un grafo è una successione di archi contigui, ossia ogni nuovo arco inizia col nodo dove finisce il precedente:
{<n1,n2>,<n2,n3>,….,<nk-1,nk>}
Il numero di archi coinvolti in un cammino costituisce la sua **lunghezza.**
Un nodo nj è **raggiungibile** dal nodo ni se esiste un percorso che parte da ni e finisce su nj.
Un percorso si dice **ciclo** se il nodo finale coincide col nodo inziale: nk==n1.
Un particolare ciclo è l'auto-anello che collega un nodo con sé stesso.
Esempio di ciclo: {<2,4>,<4,5>,<5,2>} di lunghezza 3.
Un grafo si dice **connesso** se comunque si scelga un nodo esso è raggiungibile a partire da un qualsiasi altro nodo del grafo.
Un grafo si dice **completo** se per ogni coppia di nodi esiste un arco che li collega.
Un grafo si dice **pesato** se ad ogni arco è associata un'informazione numerica di *peso* o *costo* (es., la distanza in km tra due città).
In un grafo non orientato, si dice **grado** di un nodo, il numero di archi (entranti/uscenti) che coinvolgono il nodo. Nell'esempio di grafo non orientato mostrato in precedenza, grado(4)=3 etc.
In un grafo orientato, si chiama **grado di entrata** di un nodo, il numero di archi entranti sul nodo, **grado di uscita** il numero di archi uscenti. Per l'esempio di grafo orientato precedente, gradoEntrata(4)=1, gradoUscita(4)=2 etc.

Rappresentazioni in memoria di un grafo

- Un grafo può essere rappresentato mediante alcune strutture dati canoniche: *matrice di adiacenze* e *liste di adiacenze.*
- Tuttavia altre particolarizzazioni possono essere utilizzate dal progettista.
- Una matrice di adiacenze è di dimensione |N|x|N| dove |N| è la cardinalità dell'insieme dei nodi. Se il grafo è non pesato, la matrice può contenere booleani: nella cella [i,j] si pone true se esiste l'arco<i,j>, false altrimenti.
- La matrice delle adiacenze del **grafo orientato** precedente è mostrata nella slide che segue

Esempio di Matrice di Adiacenze



a) Grafo orientato

b) Grafo non orientato

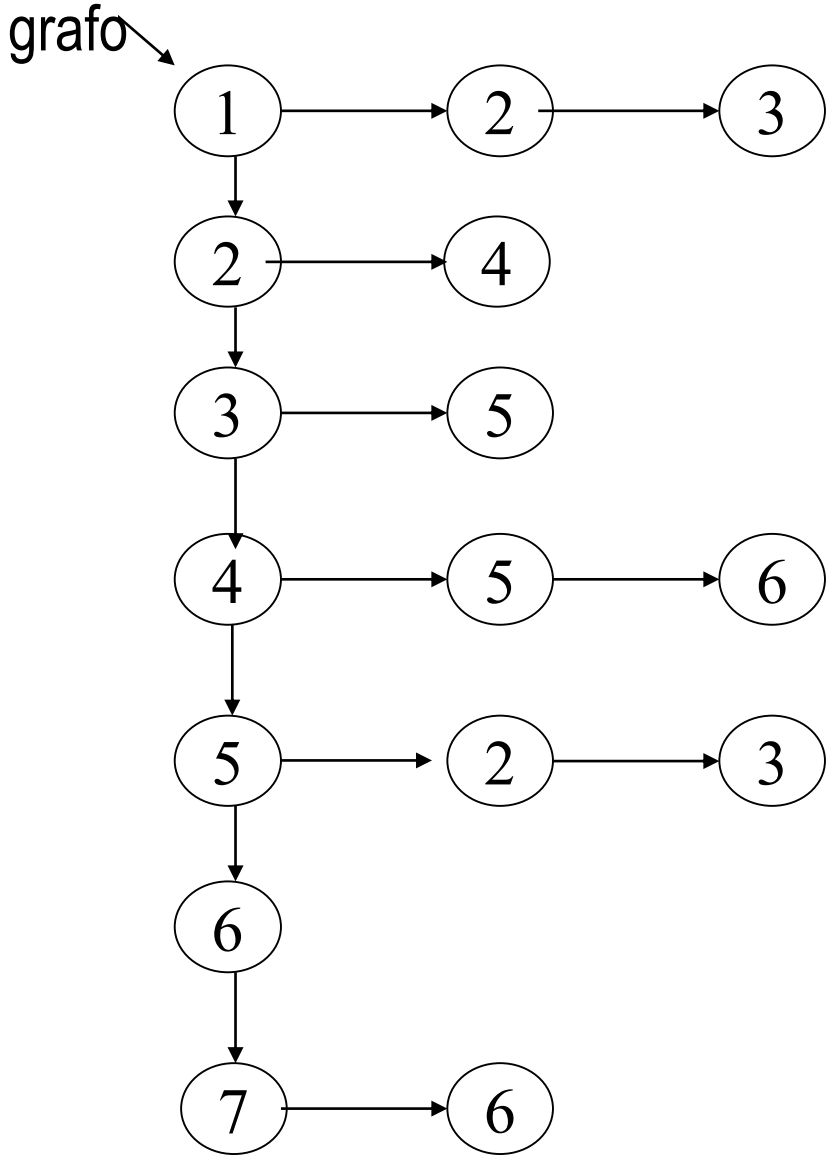
	1	2	3	4	5	6	7
1		T	T				
2				T			
3					T		
4					T	T	
5		T	T				
6							
7						T	

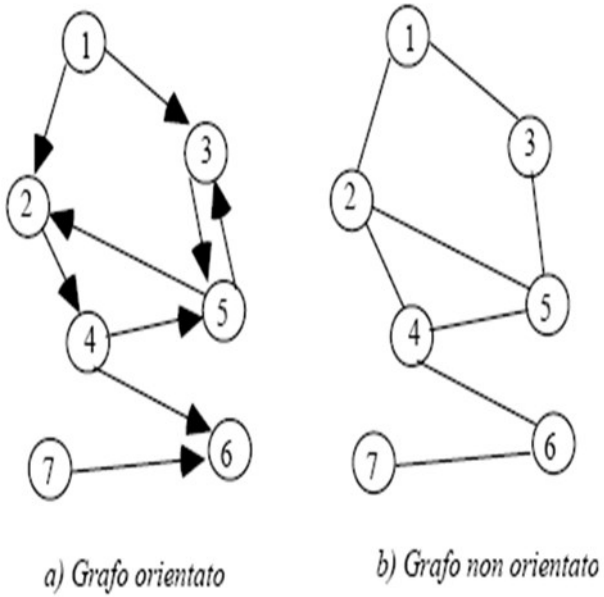
- La matrice di adiacenze si specializza in una matrice numerica se il grafo è pesato.
- In questo caso, all'intersezione tra riga ni e colonna nj si pone il peso dell'arco (se esiste) <ni,nj>.
- Per esprimere la non esistenza di un arco si può utilizzare il simbolo di infinito ∞.
- Una matrice di adiacenze comporta una complessità spaziale (ingombro di memoria) del tipo $O(n^2)$ se n è il numero dei nodi. Si nota un certo spreco di memoria corrispondente alla rappresentazione di tutte le possibili coppie di nodi <ni,nj>

Liste di adiacenze

- In questo caso si usano liste concatenate per rappresentare i nodi e gli archi.
- In modo naturale la rappresentazione è del tipo "lista di liste".
- I nodi, come caso particolare, potrebbero essere memorizzati in un array cosi da avere un "array di liste concatenate".
- Al posto dell'array si potrebbe utilizzare equivalentemente una mappa che associa ad un nodo (chiave) la lista corrispondente dei nodi adiacenti (valore).
- Per lo stesso grafo orientato presentato in precedenza, una rappresentazione con liste di adiacenze è mostrata di seguito.

Esempio di Liste di adiacenze





La lista verticale è il grafo.

In ogni nodo della lista verticale (grafo) c'è la testa della lista di adiacenze del nodo.

- Se n=|N| ed m=|A| ossia se n sono i nodi ed m gli archi del grafo, allora l'ingrombro spaziale di una lista di adiacenze è O(n+m).
- Se il grafo è pesato, allora in una lista di adiacenze si conveniente memorizzare direttamente **oggetti archi** che contengono, tra l'altro, *anche* l'informazione di peso o costo dell'arco.
- Mentre una struttura ad albero è "*radicata*", ossia la struttura è caratterizzata dal suo nodo radice (nodo di partenza per le elaborazioni es. ricerca etc), i *grafi non sono radicati*. In molti algoritmi sui grafi, occorre specificare il nodo da assumere come partenza per l'elaborazione.
- Numerosi sono gli algoritmi che si possono esprimere sui grafi.
- Il tipo astratto Grafo che segue specifica solo i *meccanismi* di base per costruire o modificare un grafo.
- Particolari algoritmi possono essere sviluppati in termini delle operazioni esportate da Grafo (o sue estensioni).
- Di seguito si specificano tre operazioni fondamentali sui grafi: quelle di **visita** e quella concernente la **raggiungibilità** dei nodi.

Il grafo come ADT (un esempio)

```
package poo.grafo;
import java.util.*;
public interface Grafo<N > extends Iterable<N>{
    int numNodi();
    int numArchi();
    boolean esisteNodo( N u );
    boolean esisteArco( Arco<N> a );
    boolean esisteArco( N u, N v );
    void insNodo( N u );
    void insArco( Arco<N> a );
    void insArco( N u, N v );
    void rimuoviNodo( N u );
    void rimuoviArco( Arco<N> a );
    void rimuoviArco( N u, N v );
    Iterator<? extends Arco<N>> adiacenti( N u );
    void clear();
    Grafo<N> copia();
}//Grafo
```

N è il tipo generico parametrico delle etichette dei nodi.

Quando si rimuove un nodo, si rimuovono nel contempo tutti gli archi che lo riguardano etc.

La classe Arco<N>

```
package poo.grafo;
public class Arco<N>{
   private N origine, destinazione;
   public Arco( N origine, N destinazione ){
      this.origine=origine;
      this.destinazione=destinazione;
   }
   public N getOrigine(){ return origine; }//getOrigine
   public N getDestinazione(){ return destinazione; }//getDestinazione
   @SuppressWarnings("unchecked")
   public boolean equals( Object o ){
      if( !(o instanceof Arco ) ) return false;
      if( o==this ) return true;
      Arco<N> a=(Arco<N>)o;
      return this.origine.equals( a.getOrigine() ) &&
            this.destinazione.equals( a.getDestinazione() );
   }//equals
   public int hashCode(){
      final int numero_primo=811;
      return origine.hashCode()*numero_primo+destinazione.hashCode();
   }//hashCode
   public String toString(){
      return "<"+origine+","+destinazione+">";
   }//toString
}//Arco
```

Operazioni di visita

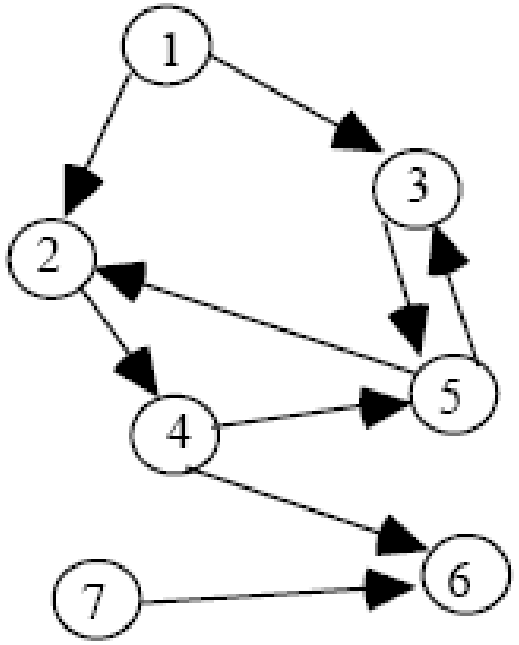
- Sussistono due tipi di visita: la **visita in ampiezza** (o *breadth-first visit*), detta spesso anche *visita a ventaglio*, e la **visita in profondità** (*depth-first visit*) detta spesso anche *visita a scandaglio.*

- La visita in ampiezza corrisponde alla visita per livelli di un albero binario. Si assume un nodo come partenza. Si visita tale nodo. Quindi si esaminano i nodi adiacenti di questo nodo. Tutti questi nodi vanno visitati *prima* che un qualsiasi altro nodo adiacente successivo possa essere visitato etc. Si prosegue sino a che non ci sono altri nodi da visitare. Poiché gli archi potrebbero far si che si ritorni su un nodo già visitato, occorre marcare i nodi visitati in modo da considerarli una sola volta.

- Una visita in ampiezza si può portare a termine con la mediazione di una *coda* su cui si memorizzano i nodi adiacenti del nodo corrente (nodi pending o ancora da considerare). È importante visitare tutti i nodi adiacenti *prima* di procedere con gli adiacenti di livello successivo. In quanto segue si suppone che la coda dei pending contenga nodi *già* visitati.

Visita in ampiezza: pseudo codice

sia **u** il nodo di partenza
sia **coda** una coda di nodi pending *già* visitati
coda←**u**
visita **u** e marcalo visitato
while(coda non è vuota){
    x ← estrai da coda il primo nodo
    per tutti i nodi adiacenti **v** di **x**{
        if(**v** non è visitato){
            visita **v** e marcalo visitato
            coda ←**v**
        }
    }
}



Nodo di partenza 1: [1, 2, 3, 4, 5, 6]
Nodo di partenza 7: [7,6]

Visita in profondità

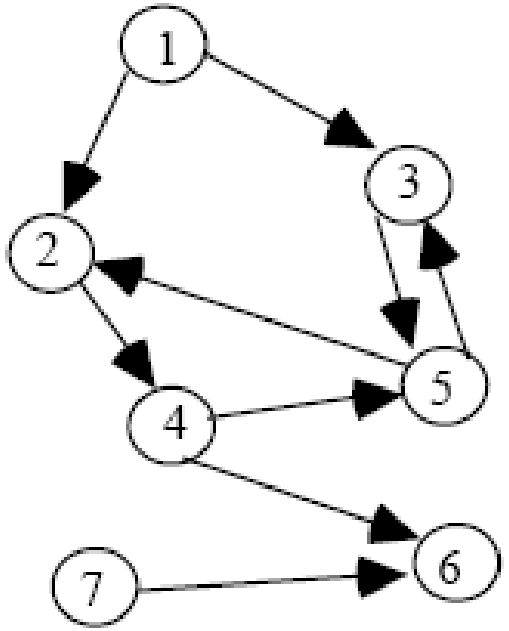
- Si considera un nodo di partenza. Si visita questo nodo. Se esso ammette nodi adiacenti, allora si attiva ricorsivamente la visita in profondità a partire da ciascun nodo adiacente. Se un nodo non ha adiacenti, la visita si arresta.

- La visita in profondità di un grafo può essere messa in relazione con le visite ricorsive (inOrder(), preOrder(), postOrder()) di un albero binario: l'obiettivo è scendere sino ad una foglia e poi risalire e continuare con il figlio destro etc. La visita cioè arriva sino in fondo e poi risale etc.

- Ovviamente la successione (come anche per la visita in ampiezza) dipende ultimamente dall'ordine come vengono esaminati i nodi adiacenti, ad es., con liste di adiacenza dall'ordine di successione degli archi su tali liste.

Visita in profondità: pseudo codice

```
visita-in-profondita(g,u){
    visita e marca u come visitato
    per tutti i nodi v adiacenti ad u {
        if( v non è visitato ){
            //ricorsione
            visita-in-profondita(g,v);
        }
    }
}
```



Nodo di partenza 1: [1, 2, 4, 5, 3, 6]
Nodo di partenza 2: [2, 4, 5, 3, 6]

Raggiungibilità

- Risponde ad un'esigenza fondamentale: sapere se un qualsiasi nodo nj è raggiungibile partendo da un dato nodo ni.
- La raggiungibilità può essere definita come la ***chiusura transitiva*** (R^+) della relazione di adiacenza *R* stabilita dall'insieme degli archi.
- Formalmente:

  ni R^+ nj se 1) ni *R nj* oppure: ∃ nk tale che:

  2) ni R^+ nk && nk *R* nj
- In altre parole nj è raggiungibile da ni o perché esiste un arco che collega ni ad nj o perché esiste un nodo nk, raggiungibile da ni, ed nj è adiacente ad nk.
- Per mettere su le informazioni richieste dalla relazione di raggiungibilità occorre generare tutti i possibili percorsi per ogni coppia di nodi <ni,nj>. I percorsi hanno lunghezza k,

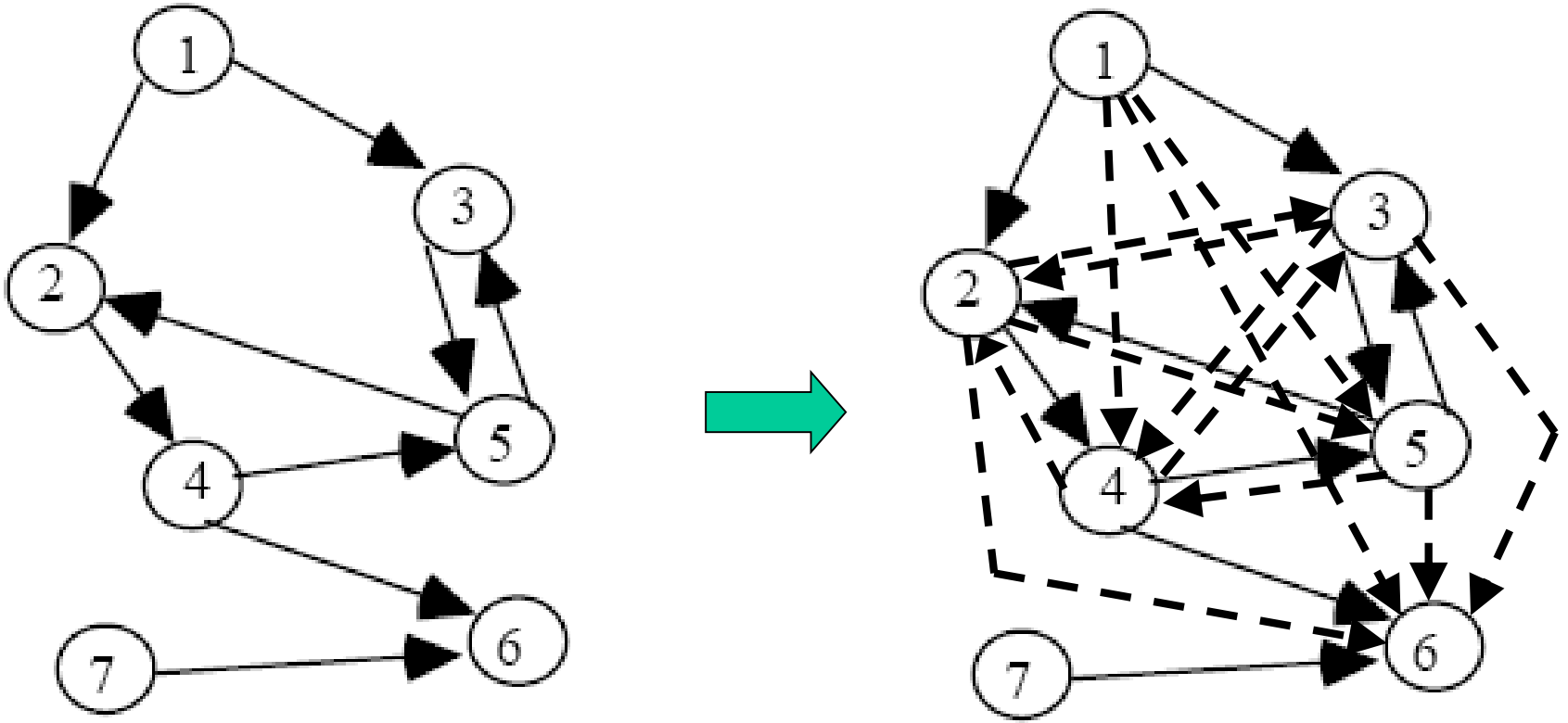
  1<=k<=n-1
- La lunghezza k=1 si riferisce ai percorsi/archi che già esistono.
- I percorsi da generare sono quelli da 2 a n-1, se n sono i nodi del grafo.
- Si nota che al più un percorso è lungo n-1 quando, ovviamente, si passa una sola volta per ogni nodo (ripassando più volte si allunga inutilmente la lunghezza di un percorso).

- Per rispondere ai quesiti sulla raggiungibilità, si può creare un nuovo grafo GR (Grafo Raggiungibilità), da inizializzare come copia del grafo di partenza (in modo da ereditarne i nodi e gli archi).
- Sul grafo GR si aggiunge un arco tra ni ed nj (che prima non esisteva) se esiste un percorso di lunghezza k (k>=2 e k<=n-1) che congiunge ni ad nj.
- Costruito GR, per conoscere se ni è raggiungibile da nj, basta vedere se nj è adiacente ad ni su GR.

- **Algoritmo pseudo codice:**

```
crea GR come copia di G
for( int k=2; k<g.numNodi(); k++ ){
    for( tutti i nodi i di G )
        for( tutti i nodi j di G )
            for( tutti i nodi m di G )
                if( GR.esisteArco(i,m) && G.esisteArco(m,j) )
                    GR.insArco(i,j);
}
ritorna GR
```

Esempio di Grafo di Raggiungibilità



G GR

Tratteggiati sono gli extra archi aggiunti a GR enumerando sistematicamente tutti i percorsi tra coppie di nodi.

Note sulle API sviluppate

- È disponibile, nell'ambito del corso di POO, la libreria **grafi.jar** che contiene tutte le classi e interfacce delle API sui grafi, package **poo.grafo**.
- Per maggiori informazioni sui metodi etc consultare il libro di testo.
- Per usare tali entità è sufficiente, in Eclipse, cliccare col destro sul nome del progetto Java (es. corso-poo-2020-2021), quindi Build Path, quindi Configure Build Path, scegliere Libraries e quindi Add External JARs e navigare sul file system sino alla posizione dove è situato il file grafo.jar e includerlo. Da questo momento in poi, è sufficiente l'import usuale per avvalersi, in una propria classe, delle API sui grafi.
- Lavorando fuori da Eclipse, è necessario modificare la variabile di ambiente CLASSPATH in modo che contenga anche il path assoluto di grafo.jar, es. …;c:\directory1\directory2\grafi.jar;…

La classe di utilità Grafi

Un metodo visitaInampiezza

```
public static <N> void visitaInAmpiezza(Grafo<N> g, N u, LinkedList<N> lista){
    if( g==null || !g.esisteNodo(u) )  throw new IllegalArgumentException();
    Set<N> visitato=new HashSet<N>();
    visitaInAmpiezza( g, u, visitato, lista);
}//visitaInAmpiezza
```

```
private static <N> void visitaInAmpiezza(
                        Grafo<N> g, N u, Set<N> visitato, LinkedList<N> lista ){
  //breadth-first visit
  LinkedList<N> coda=new LinkedList<N>(); //coda di nodi pending, gia' visitati
  coda.addLast(u);
  lista.addLast(u); //"visita" u
  visitato.add(u); //"marca" u come visitato
  while( !coda.isEmpty() ){
    N x=coda.removeFirst();
    Iterator<? extends Arco<N>> it=g.adiacenti(x);
    while( it.hasNext() ){
      N nodoAdiacente=it.next().getDestinazione();
      if( !visitato.contains(nodoAdiacente) ){
        coda.addLast( nodoAdiacente ); //segnala nodoAdiacente pending
        lista.addLast( nodoAdiacente ); //visita nodoAdiacente
        visitato.add( nodoAdiacente); //marca nodoAdiacente come visitato
      }
    }
  }//while
}//visitaInAmpiezza
```

Metodo visitaInProfondita

```
public static <N> void visitaInProfondita(
                    Grafo<N> g, N u, LinkedList<N> lista ){
    if( g==null || !g.esisteNodo(u) )
        throw new IllegalArgumentException();
    Set<N> visitato=new HashSet<N>();
    visitaInProfondita( g, u, visitato, lista );
}//visitaInProfondita
```

```
private static <N> void visitaInProfondita(
        Grafo<N> g, N u, Set<N> visitato, LinkedList<N> lista ){
  //depth-first visit
  lista.addLast(u);
  visitato.add(u);
  Iterator<? extends Arco<N>> it=g.adiacenti(u);
  while( it.hasNext() ){
    N nodoAdiacente=it.next().getDestinazione();
    if( !visitato.contains( nodoAdiacente ) )
      visitaInProfondita( g, nodoAdiacente, visitato, lista);
  }
}//visitaInProfondita
```

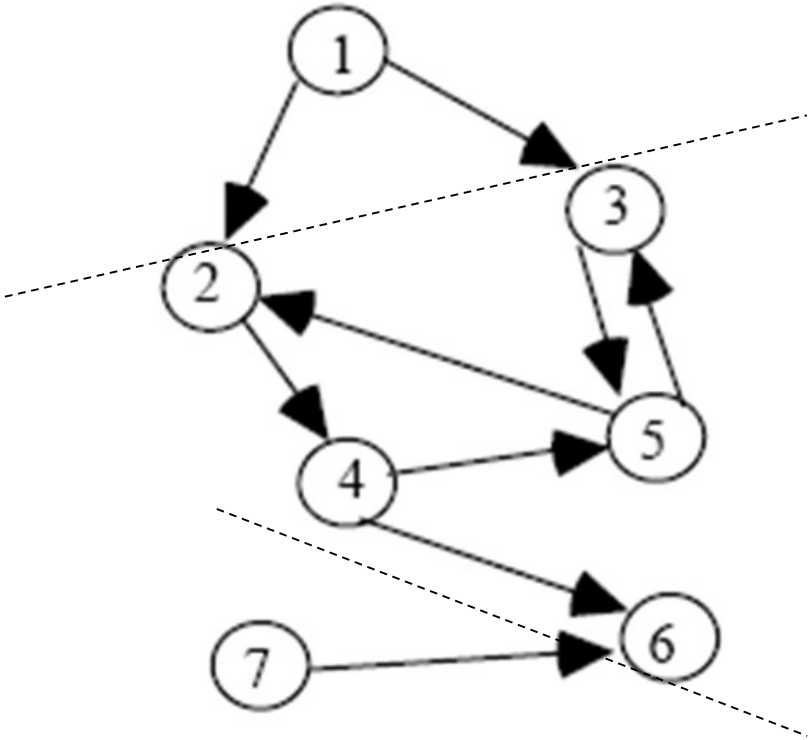
Raggiungibilità

```
public static <N> Grafo<N> raggiungibilita( Grafo<N> g ){
    Grafo<N> grafoR=g.copia();
    //genera tutti i percorsi di lunghezza k tra 2 ed n-1
    for( int k=2; k<g.numNodi(); k++ ){
      for( N i: g )
        for( N j: g )
            for( N m: g )
                if( grafoR.esisteArco(i,m) && g.esisteArco(m,j) )
                    grafoR.insArco(i,j);
    }
    return grafoR;
}//raggiungibilita
```

Verifica che un grafo orientato sia aciclico

Un algoritmo basato sul concetto di riduzione.
Un grafo orientato è aciclico se è riducibile completamente.
È ciclico, invece, se è irriducibile.

sia **gc** una copia del grafo assegnato **g**
while(esistono nodi in **gc** con grado di entrata pari a 0){
  sia **u** un tale nodo
  rimuovi **u** da **gc** (e gli archi che da esso emanano)
}
se il grafo **gc** si è svuotato, allora **g** è aciclico
altrimenti **g** è ciclico



1 e 7 sono nodi con grado di entrata 0.
Prendendo il 7 e rimuovendolo, si elimina anche l'arco <7,6>
Rimane ancora 1. Rimuovendo 1, si eliminano gli archi <1,2> e <1,3>. A questo punto non esistono altri nodi candidati alla rimozione, l'algoritmo termina, il grafo è ancora non vuoto. Esso è ciclico!
Infatti esistono i due cicli: {<2,4>,<4,5>,<5,2>} e {<3,5>,<5,3>}

Un algoritmo concreto

- Se il grafo g è molto grande, non è conveniente farne una copia gc e procedere alla riduzione di gc come indicato.

- Meglio è eseguire le rimozioni di nodi e archi in g, ma solo in modo *logico*, attraverso *strutture dati di supporto*.

  - Set<N> **rimossi** – memorizza i nodi rimossi (rimozione logica)
  - Map<N,Integer> **gradoEntrata** – contiene i nodi e il loro grado di entrata
  - LinkedList<N> **daRimuovere** – lista dei nodi pending da rimuovere, ossia nodi con il grado di entrata 0. Quando si svuota questa lista, l'algoritmo termina

```
public static <N> boolean aciclico( GrafoOrientato<N> g ){
  Set<N> rimossi=new HashSet<N>();
  Map<N,Integer> gradoEntrata=new HashMap<N,Integer>();
  LinkedList<N> daRimuovere=new LinkedList<N>();
  for( N u: g ){
    int gE=g.gradoEntrata(u);
    gradoEntrata.put( u, gE );
    if( gE==0 ) daRimuovere.addLast(u);
  }
  while( !daRimuovere.isEmpty() ){
    N n=daRimuovere.removeFirst();
    rimossi.add(n); //rimozione logica di nodo
    Iterator<? extends Arco<N>> it=g.adiacenti(n); //vai sugli archi adiacenti di n
    //decrementa il grado di entrata di ogni nodo adiacente ad n
    while( it.hasNext() ){
        N v=it.next().getDestinazione();
        gradoEntrata.put( v, gradoEntrata.get(v)-1 );
        if( gradoEntrata.get(v)==0 )  daRimuovere.addLast( v);
    }
  }
  //ritorna true se rimossi contiene tutti i nodi del grafo
  for( N u: g ) if( !rimossi.contains(u) ) return false;
  return true;
}//aciclico
```

Un tema di esame

Esercizio 1. Data la sequenza di input: 14, -2, 5, 4, 12, -14, 7, 3, 10, -1, mostrare il contenuto dell'heap corrispondente. Nell'ipotesi che venga rimosso il primo elemento dell'heap, derivare e riportare il suo nuovo contenuto.

Heap iniziale: //dimostrare graficamente i risultati forniti
[-14,-1,-2,4,3,5,7,14,10,12]

Heap dopo la rimozione del minimo:
[-2,-1,5,4,3,12,7,14,10]

Esercizio 2. Scrivere un metodo ricorsivo che genera tutte le stringhe di n bit (in numero 2^n), utilizzando lo schema backtracking.

```
package poo.appello_virtuale;
public class Esercizio2{
    static void disponi( int[] b, int i ){
        for( int bit=0; bit<=1; bit++ ){ //possibili scelte per b[i]
            b[i]=bit; //assegnamento sempre sicuro – non va controllato con assegnabile
            if( i==b.length-1 ){
                System.out.println( java.util.Arrays.toString( b ) );
            }
            else
                disponi( b, i+1 );
            //deassegnamento del valore di b[i] inutile
        }
    }//disponi
    public static void main( String[] args ){
        int []b=new int[16];
        disponi( b, 0 );
    }
}//Esercizio2
```

Esercizio 3: Verifica aciclicità di un grafo orientato.
Soluzione già mostrata. Si riporta solo un main di test.

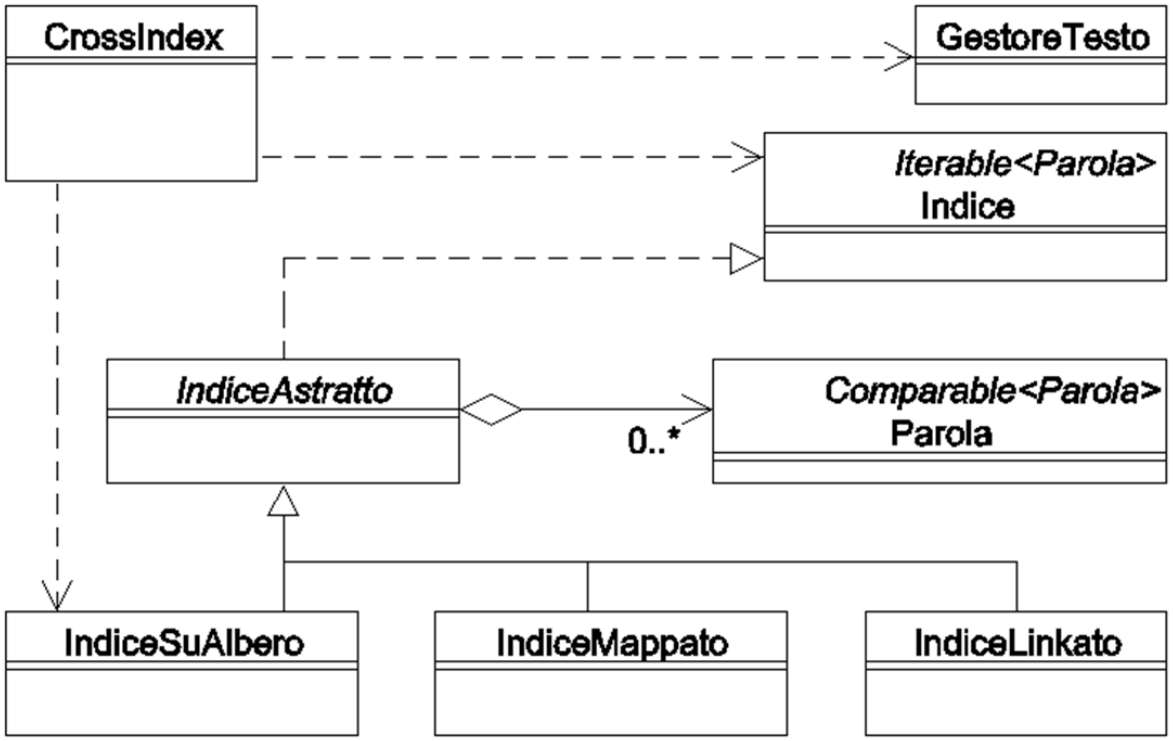
```
package poo.appello_virtuale;
import poo.grafo.*; import java.util.*;
public class Esercizio3{

  public static <N> boolean aciclico( GrafoOrientato<N> g ){…}//vedi pag. 41

  public static void main( String[] args ){
    GrafoOrientato<Integer> go=new GrafoOrientatoImpl<>();
    //Esempio
    go.insNodo(1); go.insNodo(2); go.insNodo(3); go.insNodo(4);
    go.insNodo(5); go.insNodo(6); go.insNodo(7);
    go.insNodo(8); go.insNodo(9); go.insNodo(10);
    go.insArco(1,2); go.insArco(2,10); go.insArco(1,3);
    go.insArco(6,3); go.insArco(4,6); go.insArco(9,4); go.insArco(9,10);
    go.insArco(3,5); go.insArco(4,8); go.insArco(5,8);
    go.insArco(7,5); go.insArco(7,9);
    System.out.println(go);

    if( aciclico(go) ) System.out.println("Il grafo e' aciclico");
    else System.out.println("Il grafo non e' aciclico");
  }
}//Esercizio3
```

Esercizio 4: Scrivere un programma per la generazione *dell'indice dei riferimenti i ncrociati* (cross index) di un testo conservato su un file di tipo testo.
Il nome del file di tipo testo, supposto esistente, va ottenuto preliminarmente da tastiera. In un cross index, ogni parola distinta è associata all'elenco (ordinato) dei numeri di linea su cui compare. Tutto l'indice va prodotto per lunghezza crescente di parola ed a parità di lunghezza in senso lessicografico delle parole.



```
package poo.appello_virtuale;
import java.io.*; import java.util.*;
public class CrossIndex{
   public static void main( String []args ) throws IOException{
      System.out.println("Indice dei riferimenti incrociati");
      Scanner s=new Scanner( System.in );
      String nomeFile=null; File f=null;
      do{
         System.out.print("Nome file testo = ");  nomeFile = s.nextLine();
         f = new File(nomeFile);
         if( !f.exists() ) System.out.println("File inesistente. Ridarlo!");
      }while( !f.exists() );
      GestoreTesto gt = new GestoreTesto( f, "\\W+"); //caratteri non di word sono delimitatori
      Indice indice = new IndiceSuAlbero();
      String word = null;
      int numLinea = 0;
      GestoreTesto.Simbolo simbolo = null;
      for(;;){
         simbolo = gt.prossimoSimbolo();
         if( simbolo==GestoreTesto.Simbolo.EOF ) break;
         word = gt.getString().toUpperCase();
         numLinea = gt.getNumeroLinea();
         indice.add( word, numLinea );
      }
      s.close(); //chiudi scanner su tastiera
      System.out.println(); System.out.println("Contenuto dell'indice");System.out.println( indice );
   }//main
}//CrossIndex
```

```
package poo.util;
import java.io.*;
import java.util.*;

public class GestoreTesto {

  public enum Simbolo{ WORD, EOF }
  private boolean EOF=false;
  private String linea=null;
  private Scanner input, scan;
  private int numeroLineaCorrente=0;
  private String word, delim;

  public GestoreTesto( File f, String delim ) throws IOException{
    input=new Scanner( f );
    this.delim=delim;
  }//costruttore
```

```
private void avanza(){
  try{
    if( linea==null || !scan.hasNext() ){
      linea=input.nextLine();
      numeroLineaCorrente++;
      //echo di linea su output
      System.out.println( numeroLineaCorrente+": "+linea );
      scan=new Scanner( linea );
      scan.useDelimiter( delim );
    }
  }catch( Exception ioe ){
    EOF=true; input.close();
  }
}//avanza
```

```
  public Simbolo prossimoSimbolo() throws IOException{
    do{
      avanza();
    }while( !EOF && !scan.hasNext() );
    if( EOF ) return Simbolo.EOF;
    word = scan.next();
    return Simbolo.WORD;
  }//prossimoSimbolo

  public String getString(){ return word;   }//getString

  public int getNumeroLinea(){
    return numeroLineaCorrente;
  }//numeroLinea

}//GestoreTesto
```

```
package poo.appello_virtuale;
import java.util.*;
public class Parola implements Comparable<Parola>{
   private String ortografia;
   private Set<Integer> elenco=new TreeSet<Integer>();//numeri di linea su cui compare ortografia
   public Parola( String ortografia ){ this.ortografia=ortografia; }
   public void add( int nr ){ elenco.add( nr ); }//add
   public int size(){ return elenco.size(); }
   public String getOrtografia(){ return ortografia; }
   public boolean equals( Object o ){
      if( !(o instanceof Parola) ) return false;
      if( o==this ) return true;
      Parola p=(Parola)o;
      return ortografia.equals( p.ortografia );
   }//equals
   public int compareTo( Parola p ){
      if( ortografia.length()<p.ortografia.length() ||  ortografia.length()==p.ortografia.length() &&
         ortografia.compareTo(p.ortografia)<0 ) return -1;
      if( this.equals(p) ) return 0;
      return +1;
   }//compareTo
   public String toString(){
      String s=ortografia+"\n"; Iterator<Integer> i=elenco.iterator();
      while( i.hasNext() ){  s+=i.next()+" "; }
      s+="\n"; return s;
   }//toString
   public int hashCode(){ return ortografia.hashCode();}//hashCode
}//Parola
```

```
package poo.appello_virtuale;
import java.util.Iterator;
public interface Indice extends Iterable<Parola>{ //ADT
  default int size(){
    int c=0;
    for( Iterator<Parola> it=this.iterator(); it.hasNext(); it.next(), c++ );
    return c;
  }//size
  default int occorrenze( String ortografia ){
    Parola orto=new Parola( ortografia );
    for( Parola p: this ){
      if( p.equals(orto) ) return p.size();
      if( p.compareTo(orto)>0 ) return 0;
    }
    return 0;
  }//occorrenze
  void add( String ortografia, int numeroRiga );
}//Indice
```

```
package poo.appello_virtuale;
import java.util.Iterator;
public abstract class IndiceAstratto implements Indice {
   public String toString(){
      StringBuilder sb=new StringBuilder( 400 );
      for( Parola p: this )  sb.append(p);
      return sb.toString();
   }//toString
   public boolean equals( Object o ){
      if( !(o instanceof Indice) ) return false;
      if( o==this ) return true;
      Indice ix=(Indice)o;
      if( this.size()!=ix.size() ) return false;
      Iterator<Parola> i1=this.iterator(), i2=ix.iterator();
      while( i1.hasNext() ){
         Parola p1=i1.next(), p2=i2.next();
         if( !p1.equals(p2) ) return false;
      }
      return true;
   }//equals
   public int hashCode(){
      final int MOLT=43;
      int h=0;
      for( Parola p: this ) h=h*MOLT+p.hashCode();
      return h;
   }//hashCode
}//IndiceAstratto
```

```
package poo.appello_virtuale;
import poo.util.*; import java.util.*;

public class IndiceSuAlbero extends IndiceAstratto{
   private AlberoBinarioDiRicerca<Parola> indice = new AlberoBinarioDiRicerca<Parola>();
   public int occorrenze( String ortografia ){
      Parola p = new Parola( ortografia );
      p = indice.get( p );
      if( p==null ) return 0;
      return p.size();
   }//occorrenze
   public void add( String ortografia, int nr ){
      Parola p = new Parola( ortografia );
      if( !indice.contains( p ) ){
            p.add( nr ); indice.add( p );
      }
      else{
         p = indice.get( p );
         p.add( nr );
      }
   }//add
   public Iterator<Parola> iterator(){
      return indice.iterator();
   }//iterator
}//IndiceSuAlbero
```

```
package poo.appello_virtuale;
import java.util.*;
public class IndiceMappato extends IndiceAstratto{
  private Map<Parola,Parola> indice=new TreeMap<Parola,Parola>();
  public Iterator<Parola> iterator(){ return indice.values().iterator(); }
  public int size(){ return indice.size(); }
  public int occorrenze( String ortografia ){
    Parola p = indice.get( new Parola(ortografia) );
    if( p == null ) return 0;
    return p.size();
  }//occorrenze
  public void add( String ortografia, int nr ){
    Parola p=indice.get( new Parola(ortografia) );
    if( p==null ){
      p=new Parola( ortografia ); p.add( nr ); indice.put( p, p );
    }
    else p.add( nr );
  }//add
}//IndiceMappato
```

```
package poo.appello_virtuale;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class IndiceConcatenato extends IndiceAstratto{
   private static class Nodo{
      Parola info;
      Nodo next;
   }
   private Nodo testa=null;
   private int size=0;
   public int size() { return size; }
   public void add( String orto, int nL ) {
      Nodo cor=testa, pre=null;
      Parola p=new Parola(orto); p.add(nL);
      while( cor!=null && cor.info.compareTo(p)<0 ) {
         pre=cor; cor=cor.next;
      }
      if( cor!=null && cor.info.equals(p) ) {
         cor.info.add(nL);
      }
      else if( cor==null || cor.info.compareTo(p)>0 ) {
         Nodo n=new Nodo(); n.info=p; n.next=cor;
         if( cor==testa ) testa=n;
         else pre.next=n;
         size++;
      }
   }
```

Esempio di indice su lista concatenata semplice a puntatori espliciti.
Si lascia come esercizio sviluppare un'analoga classe basata su LinkedList di java.util

```
  public Iterator<Parola> iterator(){ return new IteratoreIndice();  }

  private class IteratoreIndice implements Iterator<Parola>{
    private Nodo pre=null, cur=null;
    public boolean hasNext() {
      if( cur==null ) return testa!=null;
      return cur.next!=null;
    }
    public Parola next() {
      if( !hasNext() ) throw new NoSuchElementException();
      if( cur==null ) cur=testa;
      else { pre=cur; cur=cur.next; }
      return cur.info;
    }
    public void remove() {
      if( cur==pre ) throw new IllegalStateException();
      if( cur==testa ) testa=testa.next;
      else pre.next=cur.next;
      size--;
      cur=pre;
    }
  }
}//IndiceConcatenato
```

Statistiche fp-fpq

- È assegnato un file di tipo testo (il cui nome fisico va letto preliminarmente da tastiera). Si desidera valutare, per ogni parola distinta del file **p**, la sua frequenza relativa **fp**, ossia la percentuale del numero di volte in cui p è usata nel file, diviso il numero totale di parole. Similmente, per ogni coppia di parole consecutive **<p,q>**, si desidera conoscere la frequenza **fpq**, ossia il numero di volte che q occorre immediatamente dopo p, diviso il numero di volte con cui p è presente nel file.
- Il programma deve accumulare le informazioni per derivare le statistiche d'uso delle parole di cui sopra, e alla fine deve:
  - Mostrare l'elenco ordinato delle parole distinte, e per ogni parola la sua frequenza relativa fp, e per ogni coppia di parole consecutive <p,q> la sua frequenza relativa fpq.
  - Data poi una parola campione (diciamola **target**) fornita sempre da tastiera, il programma deve visualizzare la parola che **più** *verosimilmente* segue target nel file (ossia avente la massima frequenza relativa fpq), e la parola che **meno** *verosimilmente* segue target sul file (ossia avente la minima frequenza relativa fpq).
  - Scrivere tutti i risultati non solo su standard output, ma anche su un file testo a piacere.

Un'interfaccia Statistica

Ai fini di derivare una soluzione, si considera la seguente interfaccia.

```
package poo.statistiche_fpfpq;
public interface Statistica{
    void arrivoParola( String p );
    void paroleConsecutive( String p, String q );
    int numTotaleParole();
    int frequenza( String p );
    int frequenzaCoppia( String p, String q );
    String parolaCheSeguePiuFrequente( String target );
    String parolaCheSegueMenoFrequente( String target );
}//Statistica
```

Per interfacciarsi col testo in ingresso, si fa riferimento alla stessa classe GestoreTesto vista nel programma CrossIndex.

Un main program complessivo

```
package poo.statistiche_fpfpq;
import java.util.*;
import java.io.*;

public class FpFpq{
   public static void main( String[] args ) throws IOException{
      Scanner sc=new Scanner(System.in);
      String nomeFile=null;
      File f=null;
      do{
         System.out.print("Nome file testo: ");
         nomeFile=sc.nextLine(); f=new File( nomeFile );
         if( !f.exists() ) System.out.println("File "+nomeFile+" inesistente! Ridarlo");
      }while( !f.exists() );

      GestoreTesto gt=new GestoreTesto( f, "\\W+");
      Statistica stat=new StatisticaMap();

      GestoreTesto.Simbolo sim=gt.prossimoSimbolo(); //simbolo corrente
      String ppre=null; //parola precedente
```

```
    while( sim!=GestoreTesto.Simbolo.EOF ){
        String pcor=gt.getString().toUpperCase(); //parola corrente
        stat.arrivoParola( pcor );
        if( ppre!=null ){
          stat.paroleConsecutive( ppre, pcor );
        }
        ppre=pcor;
        sim=gt.prossimoSimbolo();
    }//while

    System.out.print("Parola target="); String target=sc.nextLine().toUpperCase();
    sc.close();
    System.out.println("Statistica d'uso delle parole:"); System.out.println(stat);

    PrintWriter pw=new PrintWriter( new FileWriter("c:\\poo-file\\statistica.txt"));
    pw.println( stat ); pw.close();
    System.out.println("Parola che più verosimilmente segue "+target+"="+
      stat.parolaCheSeguePiuFrequente(target));
    System.out.println("Parola che meno verosimilmente segue "+target+"="+
      stat.parolaCheSegueMenoFrequente(target));
  }//main

}//FpFpq
```

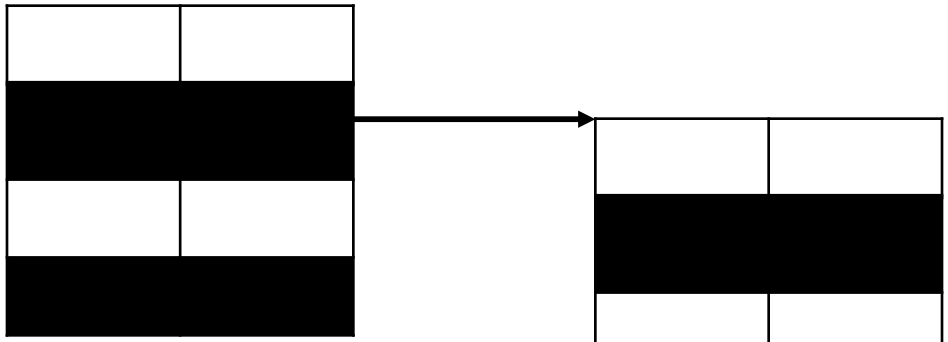
La classe StatisticaMap

- Ai fini del programma, fondamentale è la *raccolta delle informazioni* di uso delle varie parole.
- Una prima possibilità è quella di ricorrere a delle mappe:
  - **Map<String,Integer> fp**, utile per memorizzare le parole (String) distinte e la loro frequenza fp.
  - **Map<String,Map<String,Integer>> fpq**, utile per memorizzare le statistiche d'uso delle coppie <p,q> di parole.
- La prima mappa (supposta ordinata ai fini dell'output), registra le parole distinte con le associate frequenze assolute d'uso.
- La seconda struttura dati è una mappa di mappe. La chiave della prima mappa è una parola distinta del file. Il valore corrispondente è ancora una mappa (di secondo livello) che memorizza le parole adiacenti, ossia le parole che nel file ricorrono immediatamente dopo la parola che è chiave della mappa di primo livello, e per ogni parola adiacente si memorizza come valore la frequenza assoluta fpq.

Parola p	fp

Mappa fp

Mappa fpq



Mappa
1° livello

Mappa
2° livello

```
package poo.statistiche_fpfpq;
import java.util.Map; import java.util.HashMap;
import java.util.Iterator; import java.util.TreeMap;

public class StatisticaMap implements Statistica{
  private Map<String,Integer> fp=new TreeMap<>();
  private Map<String,Map<String,Integer>> fpq=new TreeMap<>();
  //non sono create, al momento, le mappe di 2 livello ...

  public int numTotaleParole(){
    int ntp=0;
    for( String s: fp.keySet() ){ ntp=ntp+fp.get(s); }
    return ntp;
  }//numTotaleParole

  public void arrivoParola( String p ){
    if( !fp.containsKey(p) ){
      fp.put(p,0);
      fpq.put(p, new HashMap<>());
    }
    fp.put( p, fp.get(p)+1 );
  }//arrivoParola
```

```
public void paroleConsecutive( String p, String q ){
  if( !fp.containsKey(p) || !fp.containsKey(q) )
    throw new RuntimeException("parole "+p+" e/o "+q+" assenti");
  Map<String,Integer> pad=fpq.get(p);
  if( !pad.containsKey(q) ) pad.put(q,0);
  pad.put( q, pad.get(q)+1 );
}//paroleConsecutive

public int frequenza( String p ){ //assoluta
  if( !fp.containsKey(p) ) return 0;
  return fp.get(p);
}//frequenza

public int frequenzaCoppia( String p, String q ){ //assoluta
  if( !fp.containsKey(p) || !fp.containsKey(q) ) return 0;
  Map<String,Integer> pad=fpq.get(p); //parole adiacenti
  if( !fpq.containsKey(q) ) return 0;
  return pad.get(q);
}//frequenzaCoppia
```

```
public String parolaCheSeguePiuFrequente( String target ){
  if( !fp.containsKey(target) ) throw new RuntimeException(target+" inesistente");
  Map<String,Integer> adiacenti=fpq.get(target);
  String ppf=null;
  int max=0;
  for( String p: adiacenti.keySet() )
  if( adiacenti.get(p)>max ){ ppf=p; max=adiacenti.get(p); }
  return ppf;
}//parolaCheSeguePiuFrequente

public String parolaCheSegueMenoFrequente( String target ){
  if( !fp.containsKey(target) ) throw new RuntimeException(target+" inesistente");
  Map<String,Integer> adiacenti=fpq.get(target);
  String pmf=null;
  int min=Integer.MAX_VALUE;
  for( String p: adiacenti.keySet() )
  if( adiacenti.get(p)<min ){ pmf=p; min=adiacenti.get(p); }
  return pmf;
}//parolaCheSegueMenoFrequente
```

```
  public String toString(){
    StringBuilder sb=new StringBuilder(500);
    int totParole=numTotaleParole();
    for( String p: fp.keySet() ){
      sb.append( "f("+p+")=" );
      sb.append( String.format("%.4f%n", ((double)frequenza(p))/totParole) );
      Iterator<String> paroleAdiacenti=fpq.get(p).keySet().iterator();
      sb.append('\t');
      int freqPar=fp.get(p);
      while( paroleAdiacenti.hasNext() ){
        String q=paroleAdiacenti.next();
        sb.append("f("+p+","+q+")=");
        sb.append( String.format("%.4f", ((double)frequenzaCoppia(p,q))/freqPar) );
        if( paroleAdiacenti.hasNext() ) sb.append(" ");
      }
      sb.append('\n');
      }
      return sb.toString();
  }//toString
}//StatisticaMap
```

Altra implementazione di Statistica

- A ben riflettere, la classe **StatisticaMap** basata sull'uso di mappe, altro non è che una realizzazione *custom* di una struttura dati del tipo grafo. Infatti, le mappe di secondo livello in fpq mantengono le informazioni di adiacenza.
- Una classe più naturale al problema (**StatisticaGrafo**) può essere ottenuta riguardando le parole come nodi (o vertici) di un grafo orientato. Le relazioni di adiacenza espresse mediante archi, riflettono gli usi delle parole consecutive. Inoltre, utilizzando un grafo orientato pesato, i pesi degli archi esprimono le frequenze assolute delle coppie adiacenti <p,q>.
- Che il grafo sia orientato è evidente: se q segue p, un arco da p a q deve esistere, ma non necessariamente sussiste un arco da q a p.
- Come altra osservazione, non è opportuno usare una semplice String come etichetta del grafo. Infatti, per ogni parola distinta, serve memorizzare la sua frequenza d'uso assoluta (che poi si trasforma in frequenza relativa al tempo del toString()). Pertanto, la classe StatisticaGrafo introduce al suo interno una classe privata Parola che memorizza una parola e la sua frequenza. Parola è usata come parametro tipo del grafo orientato pesato.

La classe StatisticaGrafo

```
package poo.statistiche_fpfpq;
import java.util.*;
import poo.grafo.*;

public class StatisticaGrafo implements Statistica{
   private static class Parola implements Comparable<Parola>{
      private String parola;
      private int frequenza;
      public Parola( String parola ){ this.parola=parola; frequenza=1; }
      public String getParola(){ return parola; }
      public int getFrequenza(){ return frequenza; }
      public void setFrequenza( int frequenza ){ this.frequenza=frequenza; }
      public String toString(){ return parola; }//toString
      public boolean equals( Object o ){
         if( !(o instanceof Parola) ) return false;
         if( o==this ) return true;
         Parola p=(Parola)o;
         return this.parola.equals(p.parola);
      }//equals
      public int hashCode(){ return parola.hashCode(); }//hashCode
      public int compareTo( Parola p ){
         return this.parola.compareTo(p.parola);
      }//compareTo
   }//Parola
```

```
private GrafoOrientatoPesato<Parola> gp=new GrafoOrientatoPesatoImpl<>();

public int numTotaleParole(){
   int ntp=0;
   for( Parola p: gp ){
      ntp=ntp+p.getFrequenza();
   }
   return ntp;
}//numTotaleParole

public void arrivoParola( String p ){
   Parola par=new Parola(p);
   if( !gp.esisteNodo(par) ) gp.insNodo(par);
   else{
      for( Parola t: gp )
         if( t.equals(par) ){
            t.setFrequenza( t.getFrequenza()+1 );
            break;
         }
   }
}//arrivoParola
```

```
public void paroleConsecutive( String p, String q ){
   Parola par=new Parola(p), pad=new Parola(q);
   if( !gp.esisteNodo(par) || !gp.esisteNodo(pad) )
      throw new RuntimeException("parole "+p+" e/o "+q+" assenti");
   if( !gp.esisteArco(par,pad) ) gp.insArco( new ArcoPesato<Parola>(par,pad,new Peso(1)));
   else{ Iterator<? extends Arco<Parola>> adiacenti=gp.adiacenti(par);
      while( adiacenti.hasNext() ){
         ArcoPesato<Parola> ap=(ArcoPesato<Parola>)adiacenti.next();
         if( ap.equals(new Arco<Parola>(par,pad)) ){
            Peso peso=ap.getPeso(); peso.setVal( peso.val()+1 ); break;
         }
      }
   }
}//paroleConsecutive
public int frequenza( String p ){ Parola par=new Parola(p);
   for( Parola q: gp ) if( q.equals(par) ) return q.getFrequenza();
   return 0;
}//frequenza
public int frequenzaCoppia( String p, String q ){
   Parola par=new Parola(p), pad=new Parola(q);
   if( !gp.esisteArco(par,pad) ) return 0;
   int fpq=0; Iterator<? extends Arco<Parola>> adiacenti=gp.adiacenti(par);
   while( adiacenti.hasNext() ){
      ArcoPesato<Parola> ap=(ArcoPesato<Parola>)adiacenti.next();
      if( ap.equals(new Arco<Parola>(par,pad)) ){fpq=fpq+(int)ap.getPeso().val(); break;  }
   }
   return fpq;
}//frequenzaCoppia
```

```
public String parolaCheSeguePiuFrequente( String target ){
   Parola t=new Parola(target);
   if( !gp.esisteNodo(t) ) throw new RuntimeException(target+" inesistente");
   Iterator<? extends Arco<Parola>> ad=gp.adiacenti(t);
   Parola pf=null; int max=0;
   while( ad.hasNext() ){
      ArcoPesato<Parola> ap=(ArcoPesato<Parola>)ad.next();
      if( ap.getPeso().val()>max ){
         pf=ap.getDestinazione(); max=(int)ap.getPeso().val(); }
   } return pf.getParola();
}//parolaCheSeguePiuFrequente

public String parolaCheSegueMenoFrequente( String target ){
   Parola t=new Parola(target);
   if( !gp.esisteNodo(t) ) throw new RuntimeException(target+" inesistente");
   Iterator<? extends Arco<Parola>> ad=gp.adiacenti(t);
   Parola mf=null; int min=Integer.MAX_VALUE;
   while( ad.hasNext() ){
      ArcoPesato<Parola> ap=(ArcoPesato<Parola>)ad.next();
      if( ap.getPeso().val()<min ){
         mf=ap.getDestinazione(); min=(int)ap.getPeso().val(); }
   } return mf.getParola();
}//parolaCheSegueMenoFrequente
```

```
  public String toString(){
    StringBuilder sb=new StringBuilder(500);
    int ntp=numTotaleParole();
    for( Parola p: gp ){
      sb.append( "f("+p.getParola()+")=" );
      sb.append( String.format("%.4f%n", ((double)p.getFrequenza())/ntp) );

      Iterator<? extends Arco<Parola>> adiacenti=gp.adiacenti(p);
      sb.append('\t');
      int freqPar=p.getFrequenza();
      while( adiacenti.hasNext() ){
        ArcoPesato<Parola> ap=(ArcoPesato<Parola>)adiacenti.next();
        sb.append("f("+p.getParola()+","+ap.getDestinazione().getParola()+")=");
        sb.append( String.format("%.4f", ap.getPeso().val()/freqPar) );
        if( adiacenti.hasNext() ) sb.append(" ");
      }
      sb.append('\n');
    }
    return sb.toString();
  }//toString

}//StatisticaGrafo
```