Programmazione Orientata Agli Oggetti

Strutture dati ricorsive e non lineari

Libero Nigro

Elaborazioni ricorsive sulla lista concatenata

- La lista concatenata è una struttura dati naturalmente ricorsiva. Tutto ciò risulta intanto dalla struttura del tipo nodo che ammette un campo next di tipo nodo (dichiarazione ricorsiva).
- Una lista concatenata è suscettibile di essere interpretata in forma ricorsiva cosi da essere manipolabile con metodi ricorsivi quando si consideri la seguente definizione:
- Una lista o
  - è vuota

  oppure
  - consiste di un primo nodo (capolista) seguito da una lista (il next del capolista) residua.
- Dovendo cercare un elemento in una lista, si può sfruttare la ricorsione come segue: se la lista è vuota allora la ricerca fallisce; altrimenti si verifica se l'elemento è nel capolista: se si, la ricerca termina con successo; se no si prosegue la ricerca sulla lista residua, invocando il metodo di ricerca sul next del capolista
- Come esempio dimostrativo, di seguito si considera una lista concatenata generica che implementa una collezione ordinata. Per ragioni di leggibilità del codice, la classe nodo è convenientemente ridenominata Lista, e la testa è nominata lista. Si ignora l'iteratore

```
package poo.recursion;

public interface CollezioneOrdinata<T extends Comparable<? super T>> {
	public int size();
	public boolean contains( T elem );
	public T get( T elem );
	public boolean isEmpty();
	public boolean isFull();
	public void clear();
	public void add( T elem );
	public void remove( T elem );
}//CollezioneOrdinata

package poo.recursion;
public class ListaRec<T extends Comparable<? super T>> implements CollezioneOrdinata<T>{
	private static class Lista<E>{
		E info;
		Lista<E> next;
	}//Lista
	private Lista<T> lista=null;
```

- A questo punto (gran parte de)i metodi di ListaRec si possono sviluppare in veste ricorsiva secondo questo schema: ogni metodo ha una versione pubblica (per i clienti) che si ricollega alla interfaccia CollezioneOrdinata, ma delega un metodo ricorsivo privato a fornire il corpo di esecuzione (risposta). Vediamo il metodo size():

```
public int size(){ //versione pubblica
    return size( lista ); //invocazione metodo privato ricorsivo con la lista (testa)
}//size

private int size( Lista<T> lista ){//versione private ricorsiva
    if( lista==null ) return 0;
    return 1+size( lista.next );
}//size
```

- Il metodo ricorsivo sfrutta direttamente la definizione ricorsiva della lista. Se essa è vuota, allora size è 0; se non è vuota, la size della lista è 1 (conta il capolista) in più della size della lista residua (ricorsione)

- Il metodo contains() sfrutta la ricorsione e l'ordinamento:

```
public boolean contains( T elem ){ //versione pubblica
      return contains( lista, elem );
}//contains

private boolean contains( Lista<T> lista, T elem ){//versione privata ricorsiva
          if( lista==null ) return false;
          if( lista.info.equals(elem) ) return true;
          if( lista.info.compareTo(elem)>0 ) return false;
          return contains( lista.next, elem );
}//contains

public T get( T elem ){ return get( lista, elem );  }//get

private T get( Lista<T> lista, T elem ){
          if( lista==null || lista.info.compareTo(elem)>0 ) return null;
          if( lista.info.equals(elem) ) return lista.info;
          return get( lista.next, elem );
}//get
```

- Più complesso è il metodo add() che aggiunge alla lista un nuovo elemento in ordine.

```
public void add( T elem ){
        lista=add( lista, elem );
}//add

private Lista<T> add( Lista<T> lista, T elem ){
     if( lista==null ){
        lista=new Lista<T>();
        lista.info=elem; lista.next=null;
        return lista;
     }
     if( lista.info.compareTo(elem)>=0 ){
        Lista<T> nuovo=new Lista<T>();
        nuovo.info=elem; nuovo.next=lista;
        return nuovo;
      }
      lista.next=add( lista.next, elem );
      return lista;
}//add
```

Siccome il metodo add() modifica la lista in quanto aggiunge un nuovo nodo in una qualunque posizione (anche in testa), la versione ricorsiva è stata programmata in modo da restituire una lista (la lista modificata) da assegnare (nel metodo public) alla lista this

Il metodo privato ricorsivo riceve due parametri: una lista e l'elemento da inserire.

Se la lista ricevuta è vuota, un nuovo nodo è creato ed inizializzato con l'elemento e restituito come lista-risultato del metodo

Se la lista non è vuota e l'elemento va posto prima del primo elemento (inserimento in testa) allora si crea un nuovo nodo e lo si inizializza con l'elemento, si collega il nodo in modo da avere come lista residua la lista ricevuta come parametro; infine si ritorna il nodo come lista-risultato

Se la lista non è vuota e l'elemento va posto dopo il capolista, allora non resta che invocare ricorsivamente il metodo sulla lista residua di quella ricevuta, stando attenti che il risultato che la chiamata ricorsiva restituirà andrà usato come nuova lista residua della lista ricevuta e quest'ultima dovrà essere ritornata come risultato del metodo

- L'esperienza di add suggerisce anche come realizzare la remove() ricorsiva, il cui studio è lasciato al lettore

```
public void remove( T elem ){
        Lista=remove( lista, elem );
}//remove

private Lista<T> remove( Lista<T> lista, T elem ){
        If( lista==null ) return lista;
        if( lista.info.compareTo(elem)>0 ) return lista;
        if( lista.info.equals(elem) ){
              return lista.next;
        }
        else{
              lista.next=remove( lista.next, elem );
              return lista;
        }
}//remove
```

- I metodi isEmpty(), isFull(), clear() si possono realizzare in modo banale col solo metodo pubblico come al solito:

```
public boolean isEmpty(){ return lista==null; }//isEmpty

public boolean isFull(){ return false; }//isFull

public void clear(){ lista=null; }//clear
```

- Naturalmente la classe ListaRec<T> dovrebbe essere dotata anche dei metodi equals(), toString() e hashCode(). Di seguito si mostra solo il toString(). Gli altri due metodi, da realizzare anch'essi in modo ricorsivo, sono lasciati come esercizio del lettore.

```
public String toString(){ //versione pubblica
        StringBuilder sb=new StringBuilder(200); //sb e' passato al metodo ricors
        sb.append('['); toString( lista, sb ); sb.append(']');
        return sb.toString();
}//toString
```

```
private void toString( Lista<T> lista, StringBuilder sb ){
        if( lista==null ) return;
        sb.append( lista.info );
        if( lista.next!=null ) sb.append(',');
        toString(lista.next, sb);
}//toString
```

- **Esercizio**

  Introdurre nella classe ListaRec un metodo mutatore

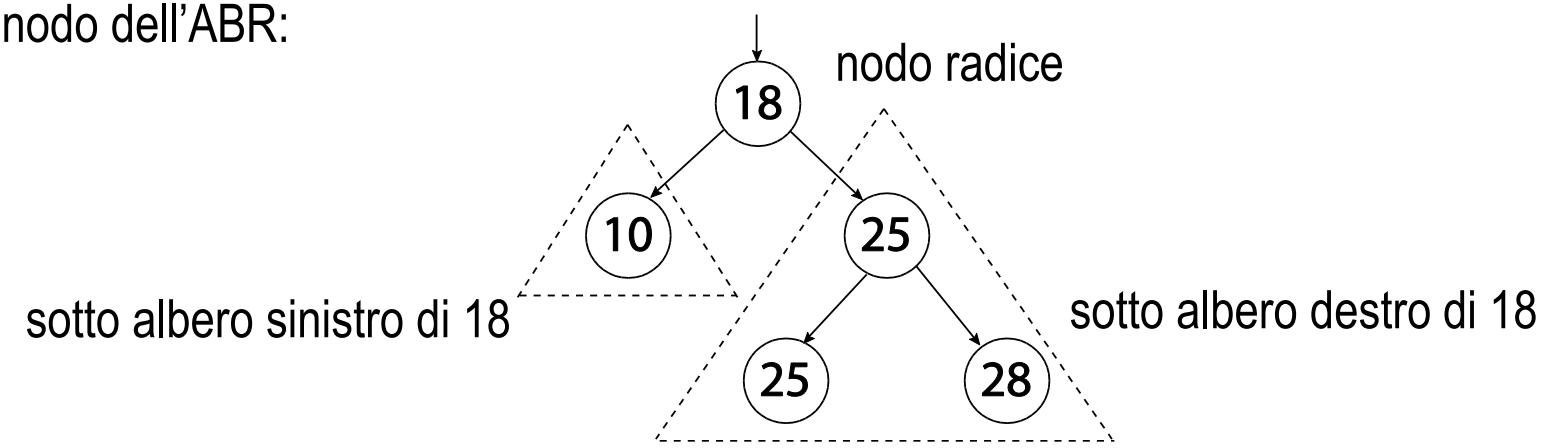
  public void reverse(){…}

  che modifica la lista this in modo da invertirne il contenuto (ossia i puntatori). Dopo l'operazione, quello che prima era il nodo di coda, ora è il capolista, il penultimo nodo è il secondo etc. quello che inizialmente era il capolista ora è il nuovo elemento di coda. **Attenzione**: non è possibile creare nuovi nodi.

  Fornire una implementazione basata su un metodo privato ricorsivo. Ottenere la realizzazione sfruttando in modo naturale la definizione ricorsiva della lista. Successivamente, fornire anche una implementazione basata su un metodo privato iterativo.

Albero binario

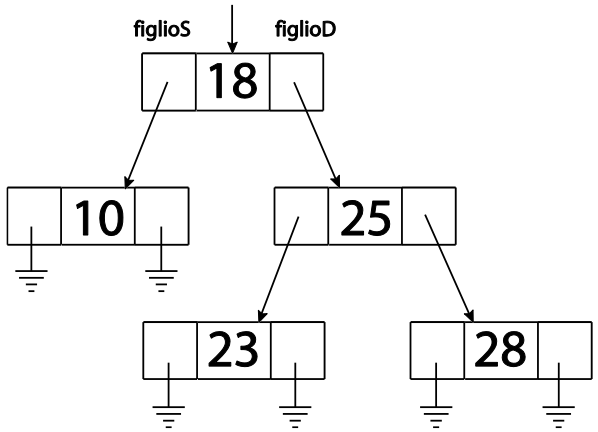
- Un albero binario rappresenta una struttura dati non lineare in cui gli elementi sono posti secondo una relazione gerarchia padre-figlio. La struttura è intrinsecamente ricorsiva: un albero binario o è vuoto o contiene un nodo detto *radice* dell'albero cui sono "attaccati" due (sotto) alberi detti rispettivamente il *sotto albero sinistro* ed il *sotto albero destro* della radice. Tutto vale ricorsivamente su ogni nodo dell'albero
- Un albero binario si dice di ricerca (ABR) se l'informazione nel nodo radice è non minore delle informazioni esistenti nel sotto albero sinistro della radice (nodi *predecessori*), e non maggiore delle informazioni esistenti nel sotto albero destro della radice (nodi *successori*). Questa definizione è verificata ricorsivamente su ogni nodo dell'ABR:



Esempio di albero binario di ricerca

- Un ABR si presta a supportare la ricerca binaria: dovendo cercare un elemento x, se esso si trova nella radice, la ricerca termina con successo; se non si trova nella radice, la ricerca prosegue o sul sotto albero sinistro (x è minore della radice) o nel sotto albero destro (x è maggiore della radice). Quando la ricerca interessa un albero vuoto, allora essa termina con fallimento
- La ricerca delineata su un ABR approssima tanto più la ricerca binaria quanto più l'albero è *bilanciato*, vale a dire che la numerosità degli elementi nel sotto albero sinistro è la "stessa" di quella del sotto albero destro e ricorsivamente ciò è verificato su ogni nodo dell'albero. Si può assumere che un albero binario sia bilanciato se preso un qualsiasi nodo, la cardinalità del sotto albero sinistro è uguale o al più differisce di 1 da quella del sotto albero destro. L'albero di esempio non è bilanciato
- Nodi come 10, 23 e 28 nell'albero di figura sono detti *terminali* o *foglie* dell'albero. 25 è un esempio di nodo intermedio o *non terminale*
- Si chiama *cammino* in un albero ogni percorso che dalla radice conduca ad una foglia
- Si dice *altezza* di un albero binario la lunghezza massima dei cammini. Nel caso dell'esempio, l'altezza è 2 (misurata in numero di archi). Detta h l'altezza ed n il numero dei nodi dell'albero si ha che: $h+1 \leq n < 2^{h+1}$
- I nodi di un albero binario sono disposti su livelli. La radice è sul livello 0. I figli della radice sono sul livello 1 etc. Un livello *l* ha al massimo 2^l nodi. Se liv è il numero dei livelli di un albero, il numero complessivo dei nodi n risulta: $n < 2^{liv}$.

- Un albero binario può essere rappresentato in memoria come mostrato in figura:



Rappresentazione in memoria di un ABR

- Ogni nodo possiede l'informazione di un elemento e due puntatori, figlioS e figlioD, rispettivamente riferimenti alla radice del sotto albero sinistro e alla radice del sotto albero destro. Un sotto albero vuoto è denotato dal valore null del relativo puntatore
- Un ABR ha proprietà simili ad un TreeSet di java.util. La differenza è che in un TreeSet non sono ammessi duplicati, mentre in un ABR in generale ciò è possibile
- Al fine di dimostrare come si possa gestire un albero binario ed in particolare un ABR in Java, si propone una classe generica AlberoBinarioDiRicerca che implementa l'interfaccia CollezioneOrdinata:

```
package poo.recursion;
public class AlberoBinarioDiRicerca<T extends Comparable<? super T>>
    implements CollezioneOrdinata<T>{

    private static class Albero<E>{
        E info;
        Albero<E> figlioSinistro, figlioDestro;
    }//Albero
    private Albero<T> radice=null;

    //metodi pubblici di interfaccia – alcuni rimandano a metodi privati ricorsivi
    public int size(){ return size(radice); }//size
    public boolean contains( T elem ){ return contains(radice,elem); }//contains
    public T get( T elem ){ return get( radice, elem ); }//get
    public void clear(){ radice=null; }//clear
    public boolean isEmpty(){ return radice==null; }//isEmpty
    public boolean isFull(){ return false; }//isFull
    public void add( T elem ){ radice=add( radice, elem ); }//add
    public void remove( T elem ){ radice=remove( radice, elem ); }//remove
```

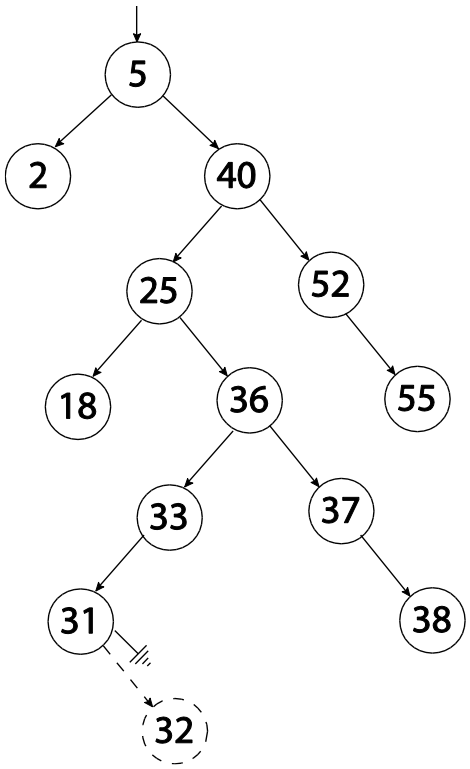
```
@SuppressWarnings("unchecked")
public boolean equals( Object x ){
        if( !(x instanceof AlberoBinarioDiRicerca) ) return false;
        if( x==this ) return true;
        return equals( this.radice, ((AlberoBinarioDiRicerca)x).radice );
}//equals

public String toString(){
        StringBuilder sb=new StringBuilder(200);
        sb.append('['); toString( radice, sb ); sb.append(']');
        return sb.toString();
}//toString

public int hashCode(){ return hashCode(radice); }//hashCode

public void visitaSimmetrica(){ visitaSimmetrica(radice); }//visitaSimmetrica
public void visitaSimmetrica( List<T> l){ visitaSimmetrica(radice,l); }//visitaSimmetrica

//metodi privati ricorsivi
private int size( Albero<T> radice ){
        if( radice==null ) return 0;
        return 1+size(radice.figlioSinistro)+size(radice.figlioDestro);
}//size
```

```
private boolean contains( Albero<T> radice, T elem ){
      if( radice==null )
            return false;
      if( radice.info.equals(elem) )
            return true;
      if( radice.info.compareTo(elem)>0 )
           return contains(radice.figlioSinistro, elem);
      return contains(radice.figlioDestro,elem);
}//contains

private T get( Albero<T> radice, T elem ){
      if( radice==null )
            return null;
      if( radice.info.equals(elem) )
            return radice.info;
      if( radice.info.compareTo(elem)>0 )
           return get(radice.figlioSinistro, elem);
      return get(radice.figlioDestro,elem);
}//get
```

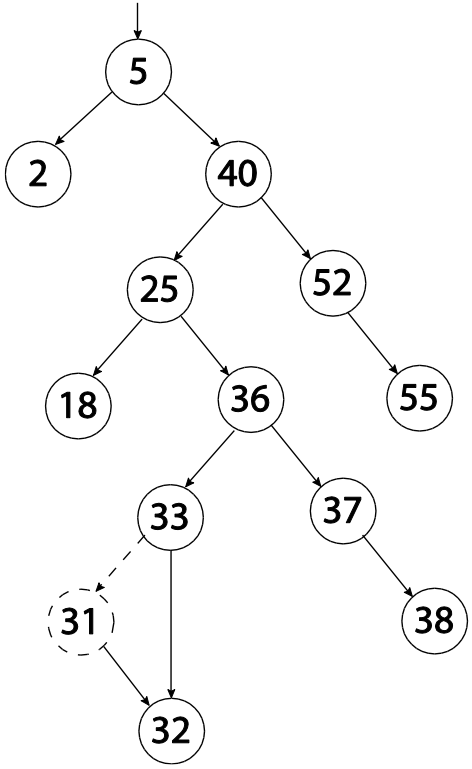
```
private Albero<T> add( Albero<T> radice, T elem ){
        if( radice==null ){
            radice=new Albero<T>();
            radice.info=elem;
            radice.figlioSinistro=null;
            radice.figlioDestro=null;
            return radice;
        }
        if( radice.info.compareTo(elem)>=0 ){
             radice.figlioSinistro=add( radice.figlioSinistro,elem );
             return radice;
        }
        radice.figlioDestro=add( radice.figlioDestro,elem );
        return radice;
}//add
```

Il metodo add() segue la stessa logica vista sulla lista concatenata ricorsiva, adattata al caso dell'albero binario di ricerca. Il metodo ritorna un albero per tener conto della modifica che la struttura subisce a causa dell'inserimento di un nuovo elemento

Più complessa è l'operazione di rimozione di un elemento data la caratteristica gerarchica dell'albero. Si possono individuare tre casi principali a seconda che la rimozione riguardi: a) una foglia dell'ABR; b) un nodo con un solo figlio; c) un nodo che ammette entrambi i figli. I casi a) e b) possono essere agevolmente realizzati, come mostrato nelle due figure che seguono

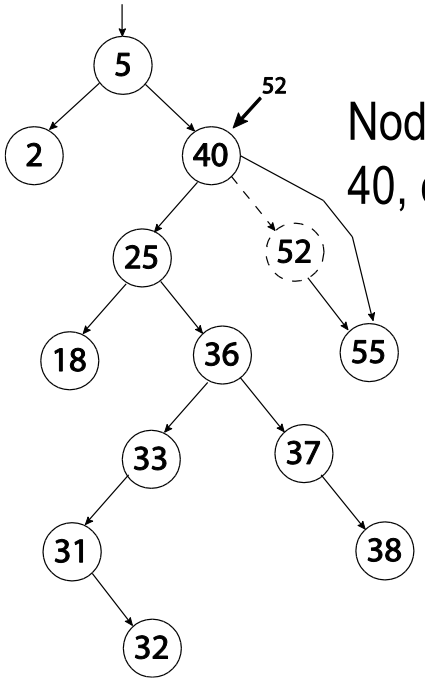


Nodo da eliminare 32, caso a)

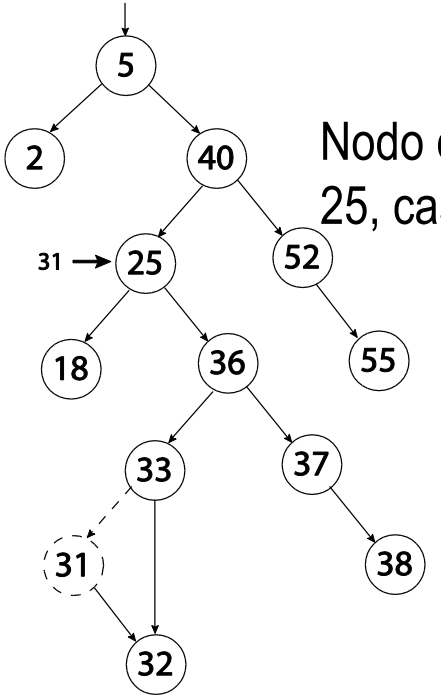


Nodo da eliminare 31, caso b)

Nel caso c) le difficoltà sono legate al fatto che non si può rimpiazzare un puntatore (quello che riferisce il nodo da eliminare) con due puntatori (i riferimenti figlioSinistro e figlioDestro che escono dal nodo obiettivo). Pertanto, si segue una logica differente. Si cerca un nodo "vittima" nel sotto albero destro o in quello sinistro del nodo da rimuovere, in modo che il nodo vittima rientri sperabilmente in uno dei casi a) o b). Trovata la vittima, la si "promuove" ossia si scrive la sua informazione nel nodo candidato alla rimozione, quindi si rimuove la vittima.
Come esempio concreto, di seguito si decide di eleggere come vittima il nodo più a sinistra nel sotto albero destro del nodo da cancellare, ossia il minimo nel sotto albero destro. Anche in questa prospettiva, il caso c) si specializza in due sottocasi: c1) il minimo del sotto albero destro coincide con la radice del sotto albero destro; c2) il minimo del sotto albero destro (massima generalità) è il nodo più a sinistra del sotto albero sinistro della radice del sotto albero destro del nodo da cancellare:



Nodo da eliminare
40, caso c1), vittima 52

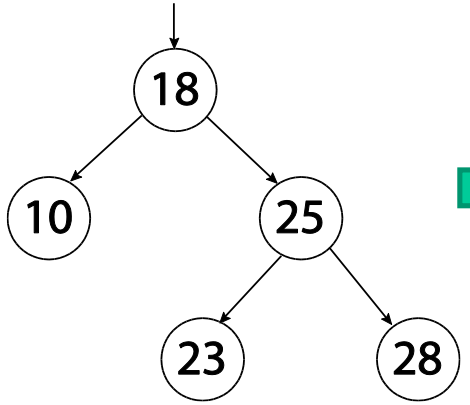


Nodo da eliminare
25, caso c2), vittima 31

```
private Albero<T> remove( Albero<T> radice, T elem ){
    if( radice==null ) return radice;
    if( radice.info.compareTo(elem)>0 ){
        radice.figlioSinistro=remove( radice.figlioSinistro, elem ); return radice;
    }
    if( radice.info.compareTo(elem)<0 ){
        radice.figlioDestro=remove( radice.figlioDestro, elem ); return radice;
    }
    //trovato nodo radice con elem
    if( radice.figlioSinistro==null && radice.figlioDestro==null )//caso a)
        return null;
    if( radice.figlioSinistro==null )//caso b)
        return radice.figlioDestro;
    if( radice.figlioDestro==null )//caso b)
        return radice.figlioSinistro;
    //Albero radice con entrambi i figli
    if( radice.figlioDestro.figlioSinistro==null ){//caso c1)
        radice.info=radice.figlioDestro.info; //promozione vittima
        //eliminazione vittima
        radice.figlioDestro=radice.figlioDestro.figlioDestro; return radice;
    }
    //caso c2)
    Albero<T> padre=radice.figlioDestro, figlio=radice.figlioDestro.figlioSinistro;
    while( figlio.figlioSinistro!=null ){
        padre=figlio; figlio=figlio.figlioSinistro;
    }
    radice.info=figlio.info; //promozione vittima
    padre.figlioSinistro=figlio.figlioDestro; //eliminazione vittima
    return radice;
}//remove
```

```
private void toString( Albero<T> radice, StringBuilder sb ){
        if( radice==null ) return;
        toString( radice.figlioSinistro, sb );
        if( sb.charAt(sb.length()-1)!='[' ) sb.append(',');
        sb.append( radice.info );
        toString( radice.figlioDestro, sb );
}//toString

private boolean equals( Albero<T> a1, Albero<T> a2 ){…}//lasciato come esercizio
private int hashCode( Albero<T> radice ){…}//lasciato come esercizio

private void visitaSimmetrica( Albero<T> radice ){
        if( radice!=null ){
            visitaSimmetrica( radice.figlioSinistro );
            System.out.print(radice.info+" ");
            visitaSimmetrica( radice.figlioDestro );
        }
}//visitaSimmetrica
…
```

Si nota che due alberi sono uguali se sono entrambi vuoti o, essendo entrambi non vuoti, sono ordinatamente uguali le radici e i sotto alberi corrispondenti (ricorsione).
Il metodo visitaSimmetrica() (o in ordine) scrive su output la successione ordinata degli elementi dell'albero. Essa realizza una visita in profondità (*depth-first*) secondo la regola: visita prima il sotto albero sinistro, quindi la radice, quindi il sotto albero destro. Ma come si visitano il sotto albero sinistro e quello destro ? Semplice: con la stessa regola! Dunque un metodo intrinsecamente ricorsivo. Si può verificare che il metodo toString() segue anch'esso il criterio della visita simmetrica
In generale si possono definire altri due metodi di visita per un albero binario: visita anticipata e visita posticipata. La visita anticipata visita prima la radice, poi il sotto albero sinistro, poi quello destro. La visita posticipata visita prima il sotto albero sinistro, poi quello destro, poi la radice. Tuttavia per un ABR ha senso solo la visita simmetrica . Di seguito si illustrano i tre metodi di visita:



Visita simmetrica: 10 18 23 25 28
Visita anticipata: 18 10 25 23 28
Visita posticipata: 10 23 28 25 18

La visita simmetrica può essere facilmente adattata in modo da restituire la sequenza ordinata dal massimo al minimo; è sufficiente visitare prima il sotto albero destro, poi la radice, poi il sotto albero sinistro

- La seguente variante di visitaSimmetrica() restituisce la sequenza ordinata su una List ricevuta come parametro, anziché scriverla su output

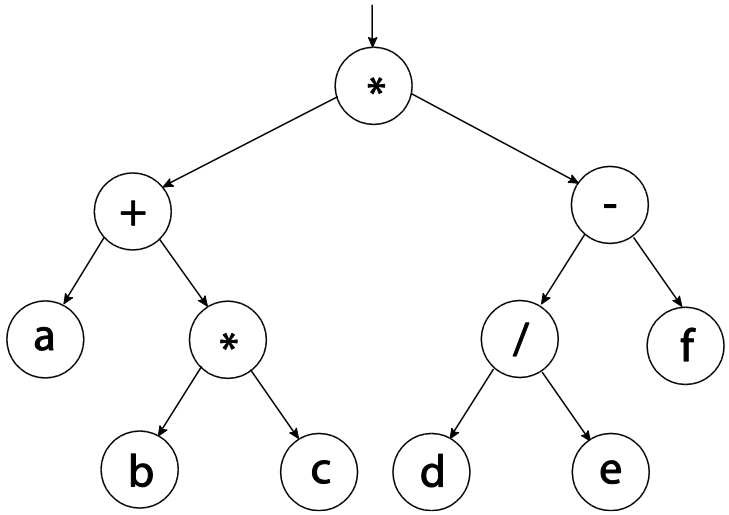
```
private void visitaSimmetrica( Albero<T> radice, List<T> l ){
        if( radice!=null ){
                visitaSimmetrica( radice.figlioSinistro, l );
                l.add(radice.info);
                visitaSimmetrica( radice.figlioDestro, l );
        }
}//visitaSimmetrica

}//AlberoBinarioDiRicerca
```

Esercizio

Sviluppare nel package poo.util una versione iterabile di AlberoBinarioDiRicerca.

Albero binario degli operatori di un'espressione aritmetica

- L'albero binario può essere utilizzato anche per rappresentare in memoria un'espressione aritmetica, in presenza delle usuali precedenze tra gli operatori
- Ad es. l'espressione (a+b*c)*(d/e-f) dà luogo all'albero degli operatori mostrato in figura:



Albero dell'espressione (a+b*c)*(d/e-f)

- Si nota che gli operatori occupano nodi non terminali, mentre le foglie rappresentano gli operandi
- Un albero di espressione può essere utilizzato per valutare l'espressione (posto che siano noti i valori degli operandi letterali, es. interi senza segno)

- Mentre normalmente un'espressione utilizza la convenzione di porre l'operatore tra gli operandi sui quali si applica (notazione infissa), altre formulazioni sono possibili nelle quali l'operatore precede o segue o suoi operandi. Si parla rispettivamente di notazione prefissa e postfissa. Entrambe queste notazioni sono interessanti in quanto evitano l'uso di parentesi. Per l'espressione di esempio si ha:

  forma prefissa: * + a * b c - / d e f

  forma postfissa: a b c * + d e / - *
- Per ottenere queste formulazioni è sufficiente visitare l'albero dell'espressione rispettivamente in modo anticipato (preOrder) e posticiparo (postOrder)
- A visita simmetrica non restituisce in generale l'espressione infissa originaria in quanto possono non essere rispettate le precedenze degli operatori. Ad es. la visita simmetrica (inOrder) dell'albero precedente fornisce:

  forma "piatta" simmetrica: a + b * c * d / e - f

  che *non* è equivalente all'espressione originaria. Decidendo di avviluppare ogni operatore in una coppia di parentesi (e) si ottiene un'espressione equivalente (sebbene ridondante di parentesi):

  forma infissa parentetica: ((a+(b*c))*((d/e)-f))

Caso di studio

- Si considera il problema di leggere da input un'espressione aritmetica in cui siano ammessi gli usuali operatori +,-,*,/,% e gli operandi siano costanti intere senza segno. I simboli non sono separati da spazi. Per semplicità gli operatori sono assunti equiprioritari. Per recuperare le precedenze della matematica si usano parentesi le cui sotto espressioni sono processate prioritariamente
- L'espressione (2+3*4)*(17/2-5) dovrà essere fornita come segue:

  (2+(3*4))*((17/2)-5) in cui si forza ad es. la valutazione di b*c impedendo la somma a+b ed il risultato moltiplicato per c (ordinamento non rispettoso delle precedenze degli operatori)
- Data un'espressione in input, si vuole costruire il corrispondente albero binario e quindi provvedere alla valutazione dell'espressione e alla visualizzazione della forma infissa, prefissa e postfissa dell'espressione
- L'applicazione è guidata da un main interattivo che consente di inserire un'espressione e verificarne subito il valore, dopo di che è possibile visualizzare l'ultima l'espressione con i comandi: **in** per inOrder, **pre** per preOrder, **post** per postOrder). Il **.** fa uscire dal programma

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

public class AlberoEspressione {
	//struttura dei nodi
	private static class Nodo{
		Nodo figlioS, figlioD;
	}//Nodo
	private static class NodoOperando extends Nodo{
		int info;
		public String toString(){ return ""+info; }
	}//NodoOperando
	private static class NodoOperatore extends Nodo{
		char op;
		public String toString(){ return ""+op; }
	}//NodoOperatore
	private Nodo radice=null;
	private String EXPR="[\\+\\-\\*/\\%\\(\\)\\d]+";  //solo condizione necessaria
```

```
public void inOrder(){ inOrder(radice); }//inOrder
public void preOrder(){ preOrder(radice); } //preOrder
public void postOrder(){  postOrder(radice); }//postOrder

public int valore(){
        if( radice==null ) throw new RuntimeException("Albero vuoto!");
        return valore(radice);
}

public void build( String expr ){
        if( !expr.matches(EXPR) ) throw new RuntimeException("Espressione malformata!");
        StringTokenizer st=new StringTokenizer(expr,"+-*/%()",true);
        radice=buildEspressione( st );
}//build

private Nodo buildEspressione( StringTokenizer st ){
        Nodo radice=buildOperando(st);
        while( st.hasMoreTokens() ){
              char op=st.nextToken().charAt(0);
              if( op==')' ) return radice;
              NodoOperatore operatore=new NodoOperatore();
              operatore.op=op; operatore.figlioS=radice;
              Nodo opnd=buildOperando(st);
              operatore.figlioD=opnd;
              radice=operatore;
        }
        return radice;
}//buildEspressione
```

```
private Nodo buildOperando( StringTokenizer st ){
        String opnd=st.nextToken();
        if( opnd.charAt(0)=='(' ){
            Nodo operand=buildEspressione(st);
            return operand;
        }
        else{
            NodoOperando numero=new NodoOperando();
            numero.info=Integer.parseInt(opnd);
            numero.figlioS=null; numero.figlioD=null;
            return numero;
        }
}//buildOperando

private int valore( Nodo radice ){
        if( radice instanceof NodoOperando )  return ((NodoOperando)radice).info;
        else{
             int val1=valore( radice.figlioS );
             int val2=valore( radice.figlioD );
             switch( ((NodoOperatore)radice).op ){
                      case '+': return val1+val2;
                      case '-': return val1-val2;
                      case '*': return val1*val2;
                      case '/': return val1/val2;
                      case '%': return val1%val2;
                      default: throw new RuntimeException();
             }//switch
        }
}//valore
```

```
private void inOrder( Nodo radice ){
        if( radice!=null ){
            if( radice instanceof NodoOperatore ) System.out.print('(');
            inOrder( radice.figlioS );
            System.out.print(radice+" ");
            inOrder( radice.figlioD );
            if( radice instanceof NodoOperatore ) System.out.print(')');
        }
}//inOrder

private void preOrder( Nodo radice ){
        if( radice!=null ){
            System.out.print(radice);
            preOrder( radice.figlioS );
            preOrder( radice.figlioD );
        }
}//preOrder

private void postOrder( Nodo radice ){
        if( radice!=null ){
            postOrder( radice.figlioS );
            postOrder( radice.figlioD );
            System.out.print(radice+" ");
        }
}//postOrder
```

```
public static void main( String []args ){
        Scanner sc=new Scanner(System.in);
        System.out.println("Valutatore di espressioni aritmetiche intere.");
        System.out.println("Forma infissa con operatori + - * / % assunti equiprioritari.");
        System.out.println("Non sono ammessi gli spazi bianchi.");
        System.out.println("E' possibile avviluppare un'espressione in ().");
        System.out.println("pre visualizza la versione prefissa.");
        System.out.println("post visualizza la versione postfissa.");
        System.out.println("in visualizza la versione infissa parentetica.");
        System.out.println(". chiude il programma.");
        String expr=null;
        String EXPR="[\\+\\-\\*/\\%\\\(\\)\\d]+";
        AlberoEspressione ae=new AlberoEspressione();
        for(;;){
            System.out.print("<<");
            expr=sc.nextLine();
            if( expr.equals(".") ) break;
            if( expr.equalsIgnoreCase("pre") ){
               ae.preOrder(); System.out.println();
            }
```

```
            else if( expr.equalsIgnoreCase("post") ){
                ae.postOrder(); System.out.println(); }
            else if( expr.equalsIgnoreCase("in") ){
                ae.inOrder(); System.out.println(); }
            else{
                try{
                    if( !expr.matches(EXPR) ) throw new RuntimeException();
                    //matches: solo condizione necessaria
                    ae.build(expr);
                    System.out.println(">>"+ae.valore());
                }catch(Exception e){
                    System.out.println("Espressione malformata!");
                }
            }
        }
        System.out.println("Bye.");
    }//main

}//AlberoEspressione
```

Esempio di sessione

Valutatore di espressioni aritmetiche intere.
Forma infissa con operatori + - * / % assunti equiprioritari.
Non sono ammessi gli spazi bianchi.
E' possibile avviluppare un'espressione in ().
pre visualizza la versione prefissa.
post visualizza la versione postfissa.
in visualizza la versione infissa parentetica.
. chiude il programma.

<<(2+(3*4))*((17/2)-5)
>>42
<<in
((2+(3*4))*((17/2)-5))
<<pre
* + 2 * 3 4 - / 17 2 5
<<post
2 3 4 * + 17 2 / 5 - *
<<.
Bye.

In **neretto** si indica l'input
che va chiuso immediatamente
da INVIO

PostOrder iterativo

- Si introduce nella classe AlberoEspressione l'enumerazione

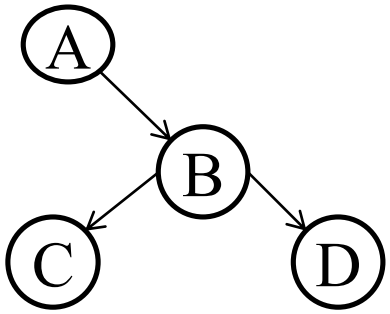
  private static enum Op{ *VISITA, SCRIVI }*
- Seguendo le indicazioni fornite nel capitolo 16 sulla conversione ricorsione-iterazione si ha:

```
private void postOrder_ite( Nodo radice ){
    class Pair{ //inner class del metodo – simula area dati
            Nodo radice;
            Op op;
            public Pair( Nodo radice, Op op ){ this.radice=radice; this.op=op;}
            public Nodo getRadice(){ return this.radice; }
            public Op getOp(){ return this.op; }
    }//Pair
```

```
    poo.util.Stack<Pair> pila=new StackConcatenato<Pair>();
    //simula prima chiamata
    pila.push( new Pair(radice,Op.VISITA) );

    while( !pila.isEmpty() ){
        Pair p=pila.pop();
        if( p.getOp()==Op.SCRIVI ){
            Nodo rad=p.getRadice();
            System.out.print( rad+" " );
        }
        else{//simula chiamate ricorsive - in ordine inverso
            if( p.getRadice()!=null ){
                pila.push( new Pair(p.getRadice(),Op.SCRIVI) );
                pila.push( new Pair(p.getRadice().figlioD, Op.VISITA) );
                pila.push( new Pair(p.getRadice().figlioS, Op.VISITA) );
            }
        }
    }//while
}//postOrder_ite
```

Esercizi

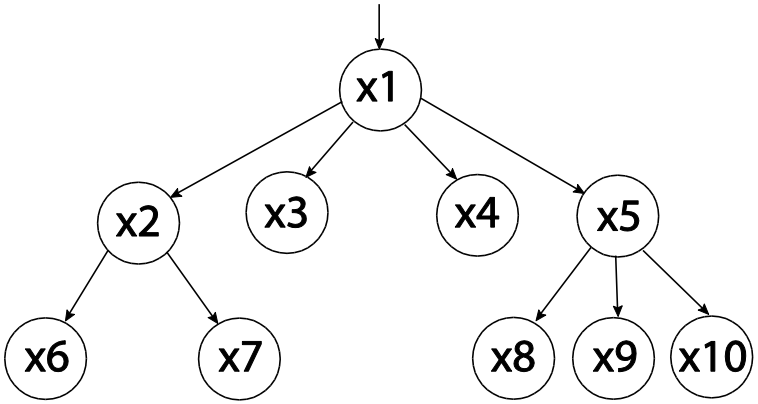
1. Nell'ipotesi di fornire in input un'espressione aritmetica in veste prefissa, progettare e testare un metodo buildPre(String expr), da aggiungere alla classe AlberoEspressione, che riceve l'espressione e costruisce l'albero corrispondente degli operatori
2. Come 1 ma quando l'espressione in input è fornita in veste postfissa. Il metodo si chiami buildPost(String expr)
3. Si consideri un albero binario di caratteri. Scrivere una classe AlberoChar che consenta di: costruire l'albero a partire da una sua forma linearizzata, e visualizzi il suo contenuto in accordo ai tre metodi di visita. Come forma linearizzata si utilizzi la seguente: si specifica prima il nodo radice quindi i due sotto alberi e cosi via ricorsivamente. Quando un sotto albero è vuoto, si scrive '.'. Ad esempio, la scrittura A.BC..D.. corrisponde all'albero:



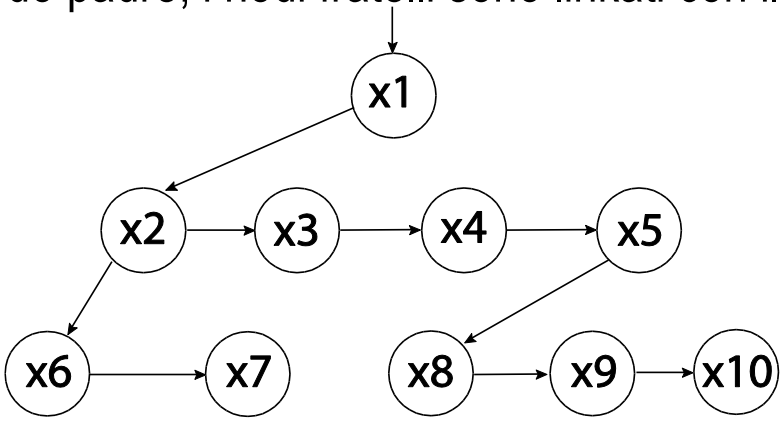
4. Scrivere un metodo di una classe di albero binario che visualizza il contenuto dell'albero per livelli, ogni livello da sinistra a destra

Alberi n-ari

- Costituiscono la specie più generale di albero, nella quale un nodo può avere 0, uno o più figli:



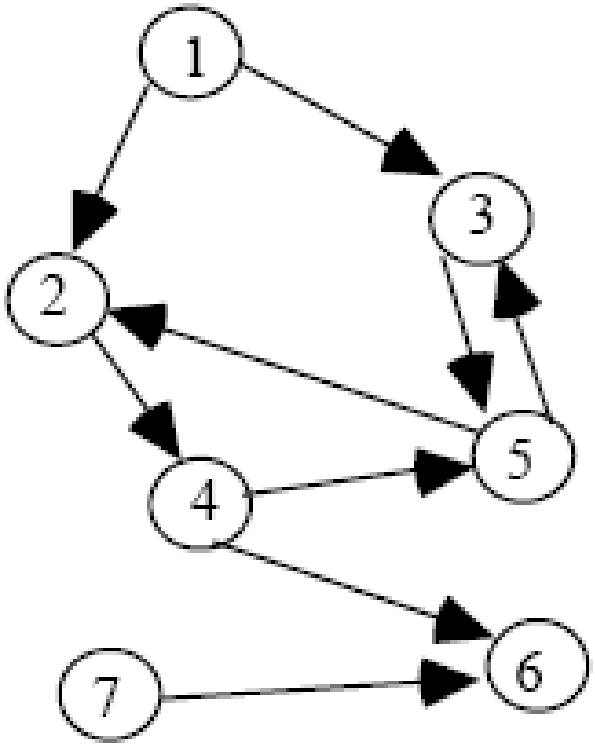
- Qui ci si limita ad osservare che un albero n-ario viene spesso impiegato nelle applicazioni mappandolo su un corrispondente albero binario. I figli di uno stesso nodo vengono collegati in una lista concatenata la cui testa è mantenuta nel campo figlioSinistro del nodo padre; i nodi fratelli sono linkati con il campo figlioDestro:



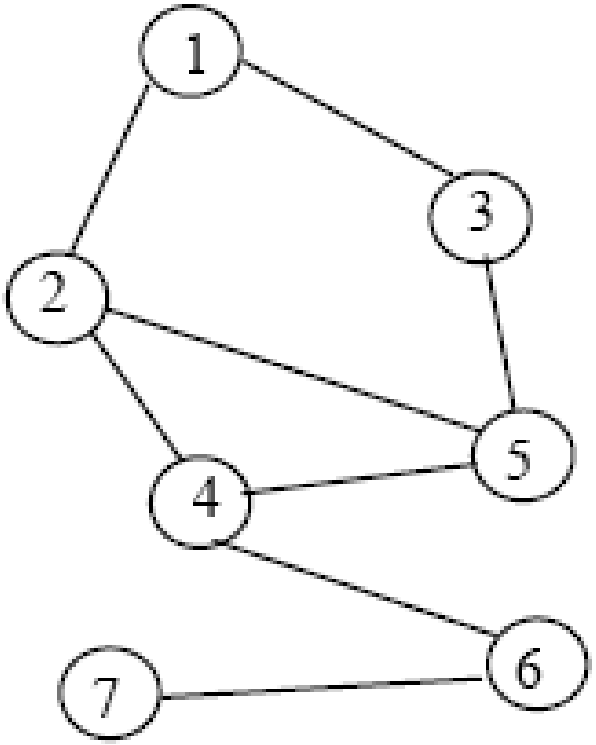
Grafi

- 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
- Più esattamente, un grafo formalmente 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 come 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 di 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

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 potrebbe collegare 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
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 adiacenza* e *liste di adiacenza.*
- Tuttavia altre particolarizzazioni possono essere utilizzate dal progettista.
- Una matrice di adiacenza 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>.
- La matrice di adiacenza del **grafo orientato** precedente è mostrata nella slide che segue.

Esempio di Matrice di Adiacenze

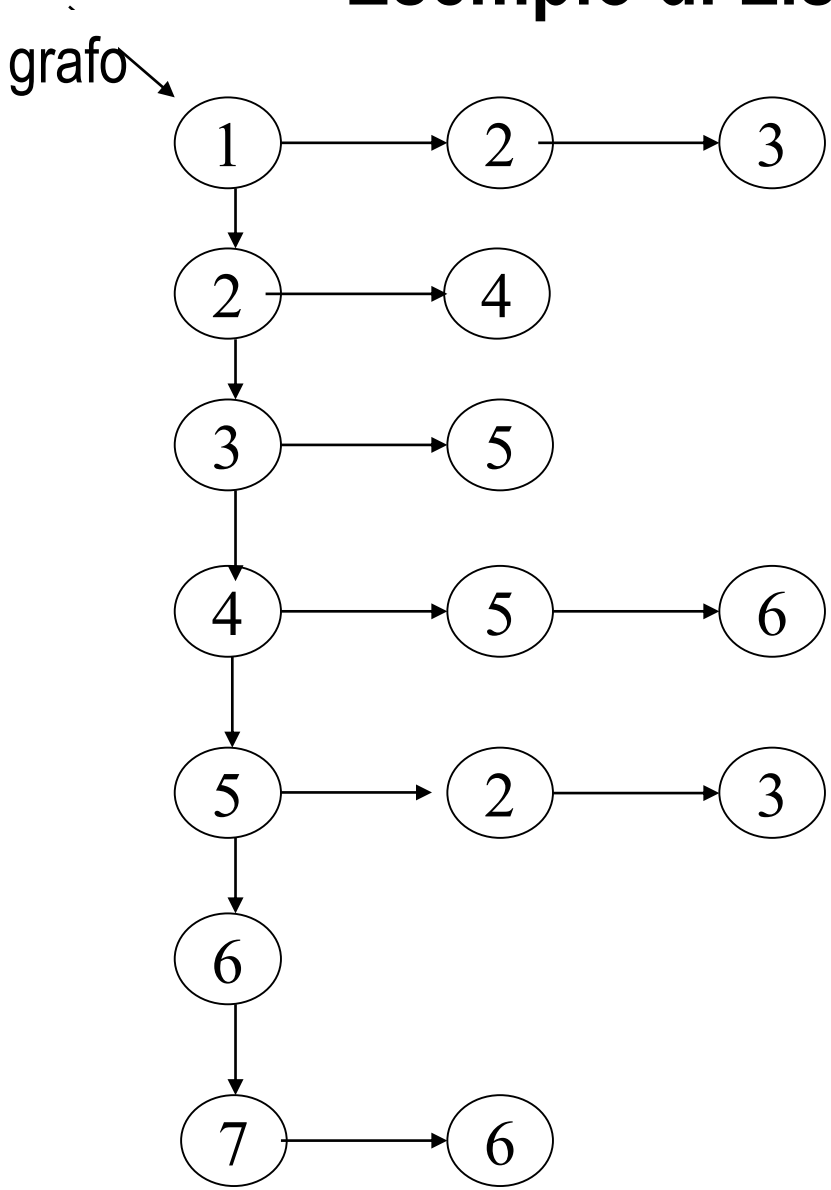
	1	2	3	4	5	6	7
1		T	T				
2				T			
3					T		
4					T	T	
5		T	T				
6							
7						T	

- 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 "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 riportata di seguito.

Esempio di Liste di adiacenze



La lista verticale è il grafo

In ogni nodo del 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 possono 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.
- Gli algoritmi possono essere sviluppati al di fuori di Grafo in termini delle sue operazioni.
- Di seguito si specificano, a titolo di esempio, tre operazioni importanti 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

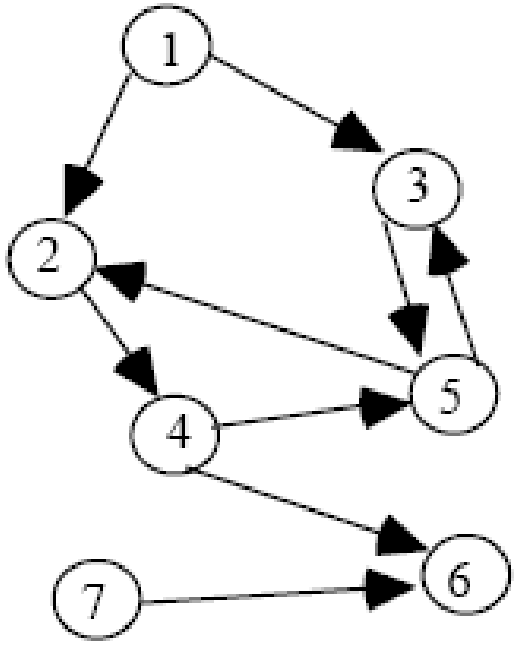
- Si possono considerare due tipi di visita: la **visita in ampiezza** (o *breadth-first visit*), detta spesso anche visita a ventaglio, e la **visita in profondità** (*depth-first visit*) detta spesso anche visita a scandaglio.

- 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. È importante visitare tutti i nodi adiacenti *prima* di procedere con gli adiacenti di livello successivo. Si suppone che la coda dei pending contenga nodi *già* visitati.

Visita in ampiezza (*breadth-first*): 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 (*depth-first*)

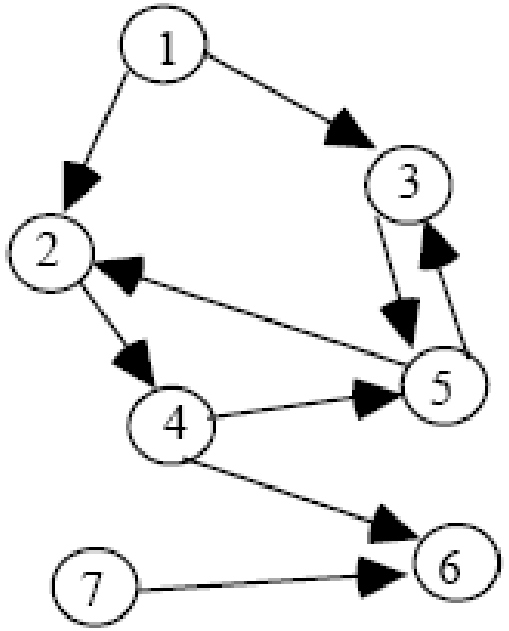
- Si considera un nodo di partenza. Si visita questo nodo. Se esso ammette nodi adiacenti, allora si attiva ricorsivamente la visita in profondità a partire da ciascun nodo adiacente. Se un nodo non ha adiacenti, la visita si arresta.

- 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 sulle 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,

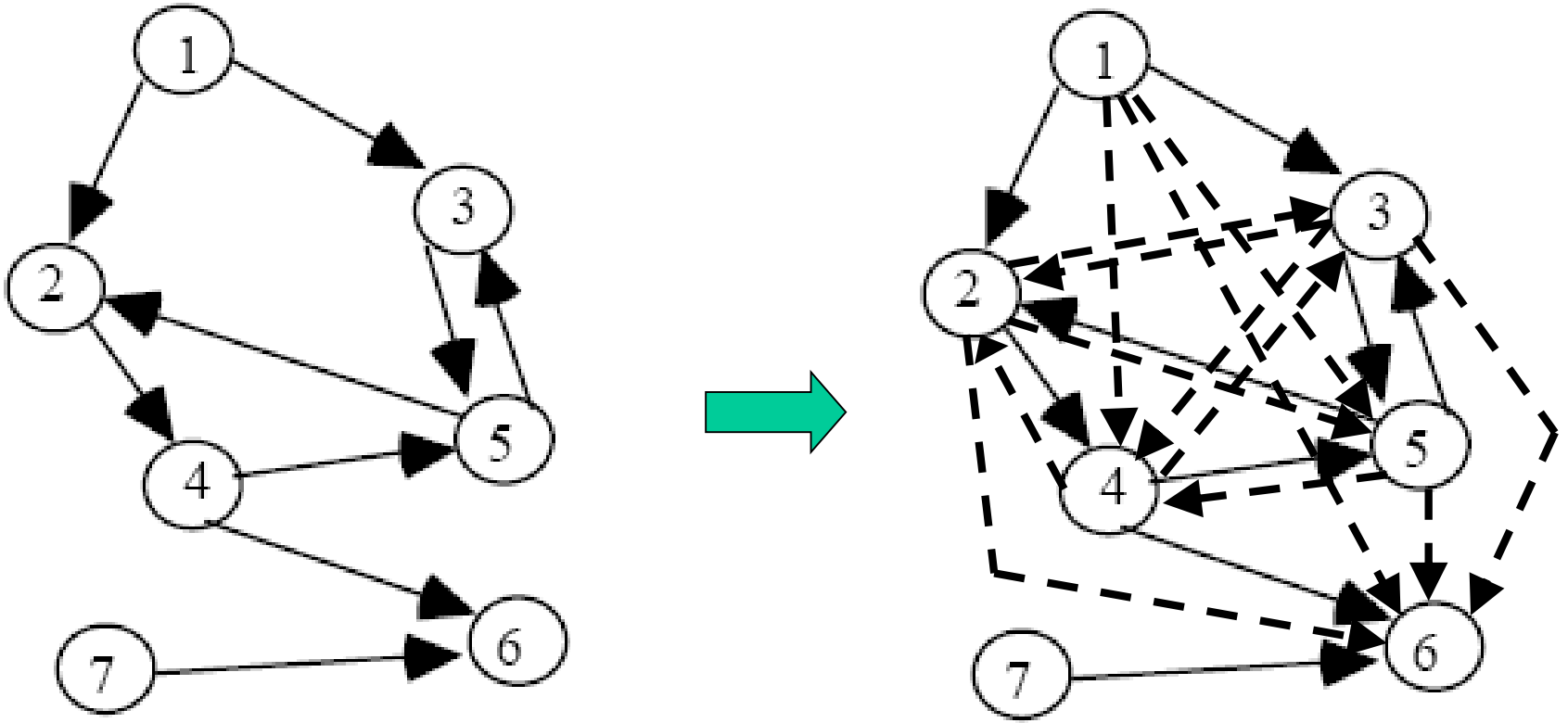
  1<=k<=n-1
- La lunghezza k=1 si riferisce ai percorsi/archi che già esistono.
- I percorsi da generare sono quelli da 2 a n-1, se n sono i nodi del grafo.
- Si nota che al più un percorso è lungo n-1 quando, ovviamente, si passa una sola volta per ogni nodo (ripassando più volte si allunga inutilmente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