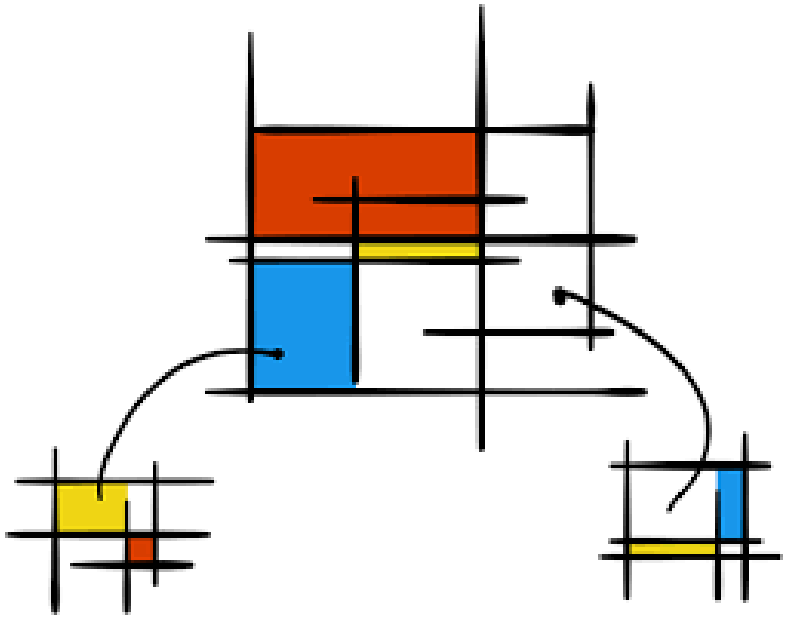


tipi di dato astratti

liste pile code alberi grafi

ADT

- *un tipo di dato astratto o ADT (Abstract Data Type) è un tipo di dato le cui istanze possono essere manipolate con **modalità** che dipendono esclusivamente dalla **semantica** del dato e **non** dalla sua **implementazione***
- *nei linguaggi di programmazione che consentono la programmazione per tipi di dati astratti, un tipo di dato viene definito **distinguendo** nettamente la sua **interfaccia**, ovvero le operazioni che vengono fornite per la manipolazione del dato, e la sua **implementazione** interna, ovvero il modo in cui le informazioni di stato sono conservate e in cui le operazioni manipolano tali informazioni al fine di esibire, all'interfaccia, il comportamento desiderato*

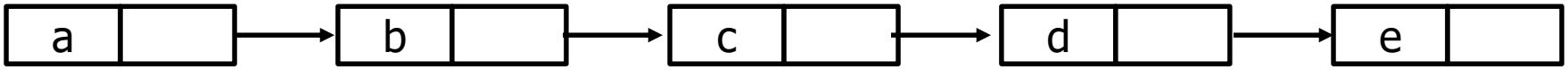
Wikipedia

- non interessa l'implementazione ma interessano le operazioni che è possibile effettuare sui dati

lista pila coda

STRUTTURE DATI LINEARI

- per gestire un insieme ***dinamico*** di elementi è necessario implementarlo mediante una struttura dati
- una struttura dati è composta da ***nodi***, ciascuno dei quali contiene un ***elemento*** dell'insieme ed eventuali ***altre informazioni*** (*riferimenti*, *puntatori*) che servono per la **gestione della struttura**



struttura fisica dei dati

- per implementare le strutture dati, si devono utilizzare le ***strutture fisiche*** fornite dai ***linguaggi*** che possono avere dimensione statica o dinamica
- un ***array***, per esempio, è una struttura di memorizzazione dalla dimensione ***statica***
- strutture fisiche dalla dimensione dinamica si possono implementare grazie all'uso dei ***puntatori*** o all'***istanziazione dinamica*** di oggetti

- una struttura dati si dice ***lineare*** se i suoi elementi sono organizzati in modo ***sequenziale*** (sono posizionati uno dopo l'altro)
  - ***pila***
  - ***coda***
  - ***lista***

pila (stack)

- la ***pila*** è una struttura dati di tipo ***LIFO*** che garantisce che l'***ultimo*** elemento ***depositato*** nella pila sia il ***primo*** a essere ***servito***
- LIFO (***Last-In First-Out***, "l'ultimo arrivato è il primo ad essere servito")
- esempi:
  - *pila di piatti*
  - *pila di libri*
  - *pila di monete*

operazioni sulla pila

- ***push***
  - ***inserisce*** un nuovo elemento in ***testa*** alla pila
- ***pop***
  - ***estrae*** il primo elemento in ***testa*** alla pila
- ***top***
  - ***fornisce*** il primo elemento in cima alla pila senza estrarlo
- ***vuota***
  - restituisce ***true*** se la pila è vuota, ***false*** altrimenti

- la ***coda*** è una struttura dati di tipo ***FIFO*** che garantisce che il ***primo*** elemento inserito sia il ***primo*** a essere servito
- ***FIFO*** (***First-In First-Out***), il primo elemento a entrare è anche il primo a uscire

operazioni sulla coda

- le tipiche operazioni che si possono effettuare su una coda sono le seguenti:
- ***enqueue*** (accodare)
  - accoda (***inserisce***) un elemento alla coda
- ***dequeue*** (togliere dalla coda)
  - ***elimina*** l'elemento che da più tempo è presente nella coda

- la ***lista concatenata*** è una ***collezione ordinata*** di elementi, ciascuno dei quali contiene un ***riferimento al successivo***
- in una lista concatenata ***non*** è possibile ***accedere*** in modo ***diretto*** a un elemento, ma è necessario ***scorrere*** tutti gli elementi fino a raggiungere quello cercato
- ogni elemento della lista è contenuto in un ***nodo***, in cui è presente anche un riferimento all'***elemento successivo***

operazioni sulla lista

- le operazioni principali che si possono effettuare su una lista sono:
  - ***inserimento***
  - ***ricerca***
  - ***cancellazione***

nodo
implementazione Java

A. Ferrari

```
class Nodo {
  private Object info;
  private Nodo succ;

  public Nodo(){
    info=null;
    succ=null;
  }

  public Nodo(Object x){
    info=x;
    succ=null;
  }

  public Nodo(Object x, Nodo s){
    info = x;
    succ = s;
  }

  //metodi setter e getter

}
```

lista
implementazione Java

A. Ferrari

```
class Lista{
  private Nodo testa;

  public Lista(){
    testa = null;
  }
  public void inseriscitesta(Object x){…}
  public void inseriscicoda(Object x){…}
  public Object eliminatesta(){…}
  public Object eliminacoda() {…}
  public void stampa() {…}
  public boolean vuota() {…}
  …
}
```

esercizio

- implementare la lista di Stringhe in Java
- istanziare una lista e testare i vari metodi utilizzando varie stringhe come informazioni

design pattern Iterator

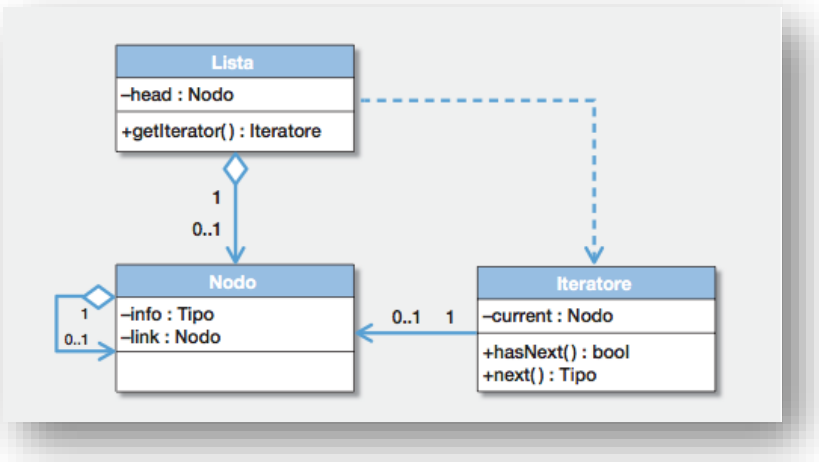
- i ***design patterns*** sono ***soluzioni generiche*** di progettazione software applicabili a ***problemi ricorrenti***
- dato un ***oggetto aggregato*** (*un oggetto che contiene altri oggetti per fornire di questi una visione unitaria, come per esempio una lista*), un ***iteratore*** (Iterator) è un ***oggetto*** che rende disponibili ***metodi*** per ***accedere sequenzialmente*** ai singoli elementi dell'oggetto aggregato ***senza esporne la rappresentazione interna***

iteratore e lista

- caratteristiche che deve avere un iteratore per operare con una lista
  - permettere l'***accesso*** agli elementi della lista ***senza esporre*** la sua ***struttura*** interna
  - fare in modo che l'accesso agli elementi della lista avvenga mediante ***metodi*** che ***non*** sono direttamente parte dell'interfaccia della ***classe*** che implementa la lista stessa, ma di una classe diversa (l'iteratore) i cui oggetti sono restituiti dall'invocazione di uno specifico metodo

la classe Iteratore

- la lista deve avere il metodo ***getIterator*** che restituisce un oggetto iteratore
- l'iteratore ha
  - un riferimento al ***nodo corrente***
  - il metodo ***hasNext()*** che restituisce true se è ancora ***possibile avanzare*** nella sequenza
  - il metodo ***next()*** che ***restituisce l'informazione*** associata al nodo corrente e "passa" al ***nodo successivo***



Iteratore


```
public class Iteratore {
   private Nodo nodo;

   public Iteratore(Nodo nodo) {
     this.nodo = nodo;
   }

   public boolean hasNext() {
     return (!(nodo==null));
   }

   public Object next() {
     if (nodo==null)
       return null;
     Object info = new Object(nodo.getInfo());
     nodo = nodo.getSucc(); // avanzamento nella lista
     return info;
} }
```

nella classe Lista

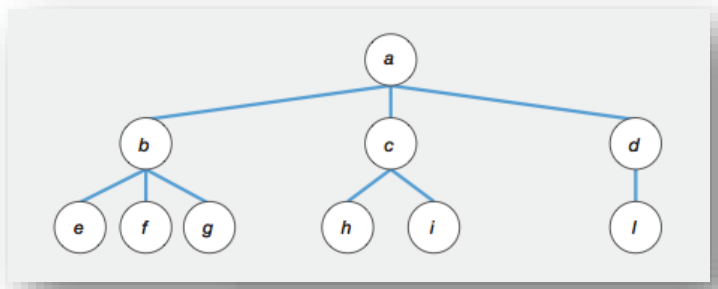

```
public Iteratore iterator() {
   Iteratore i = new Iteratore(testa);
   return i;
}
```

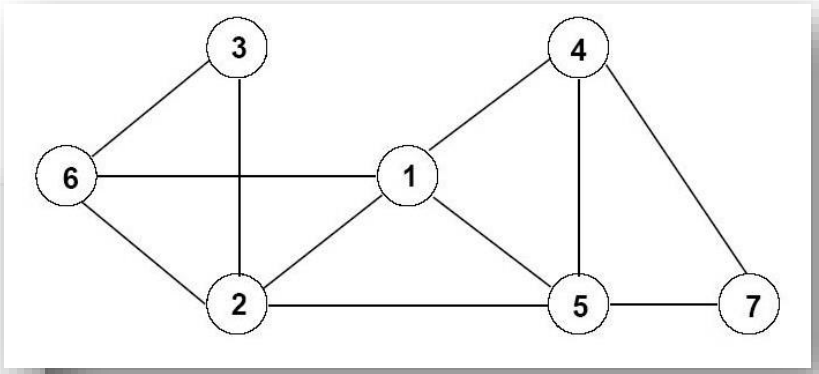
esempio: stampa elementi

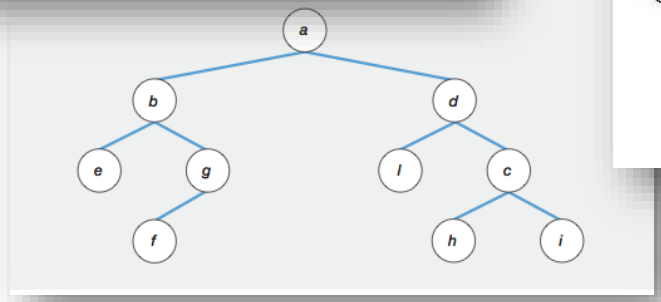
A. Ferrari

```
Iteratore it = f.iterator();
Object info;
while (it.hasNext()) {
  info = it.next();
  System.out.println(info);
}
```

- la **lista** può essere utilizzata per **implementare** le strutture dati lineari **pila** e **coda**







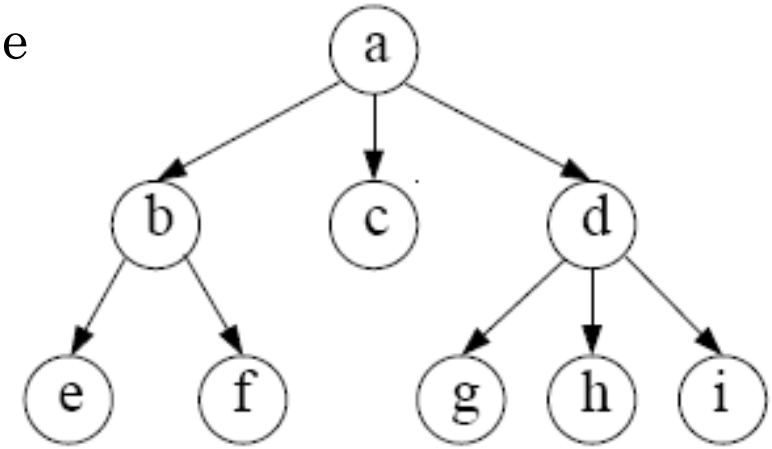
albero – albero binario - grafo

strutture dati non lineari

albero
definizione non formale

A. Ferrari

- un ***albero*** è un insieme di nodi
- ogni nodo (***padre***) è collegato mediante archi ad altri nodi (***figli***)
- un nodo particolare è definito come ***radice*** (non ha padre)
- ogni figlio ha ***un solo padre***
- ogni nodo è '***raggiungibile***' dal nodo radice
- i nodi senza figli vengono definiti ***foglie***
- il ***livello*** di un nodo è dato dal numero di archi che si devono percorrere per raggiungere il nodo dalla radice



Si dice albero con radice, o semplicemente albero, una tripla $T = (N, r, \mathcal{B})$ dove N è un insieme di nodi, $r \in N$ è detto radice e $\mathcal{B}$ è una relazione binaria su N che soddisfa le seguenti proprietà:

- Per ogni $n \in N$, $(n, r) \notin \mathcal{B}$.
- Per ogni $n \in N$, se $n \neq r$ allora esiste uno ed un solo $n' \in N$ tale che $(n', n) \in \mathcal{B}$.
- Per ogni $n \in N$, se $n \neq r$ allora n è raggiungibile da r, cioè esistono $n'_1, \ldots, n'_k \in N$ con $k \geq 2$ tali che $n'_1 = r$, $(n'_i, n'_{i+1}) \in \mathcal{B}$ per ogni $1 \leq i \leq k-1$, ed $n'_k = n$.

Siano $T = (N, r, \mathcal{B})$ un albero ed $n \in N$. Si dice sottoalbero generato da n l'albero $T' = (N', n, \mathcal{B}')$ dove N' è il sottoinsieme dei nodi di N raggiungibili da n e $\mathcal{B}' = \mathcal{B} \cap (N' \times N')$.

Sia $T = (N, r, \mathcal{B})$ un albero e siano $T_1 = (N_1, n_1, \mathcal{B}_1)$ e $T_2 = (N_2, n_2, \mathcal{B}_2)$ i sottoalberi generati da $n_1, n_2 \in N$. Allora $N_1 \cap N_2 = \emptyset$ oppure $N_1 \subseteq N_2$ oppure $N_2 \subseteq N_1$.

Sia $T = (N, r, \mathcal{B})$ un albero:

- Se $(n, n') \in \mathcal{B}$, allora n è detto padre di n' ed n' è detto figlio di n.
- Se $(n, n_1), (n, n_2) \in \mathcal{B}$, allora n_1 ed n_2 sono detti fratelli.
- I nodi privi di figli sono detti nodi esterni o foglie, mentre tutti gli altri nodi sono detti nodi interni.
- Gli elementi di $\mathcal{B}$ sono detti rami.

grado, livello, altezza, larghezza

A. Ferrari

Sia $T = (N, r, \mathcal{B})$ un albero:

- Si dice grado di T il massimo numero di figli di un nodo di T:
$$d(T) = \max_{n \in N} |\{n' \in N \mid (n, n') \in \mathcal{B}\}|$$
- Si dice che:
  - r è al livello 1.
  - Se $n \in N$ è al livello i ed $(n, n') \in \mathcal{B}$, allora n' è al livello $i + 1$.
- Si dice altezza o profondità di T il massimo numero di nodi che si attraversano nel percorso di T che va dalla radice alla foglia più distante:
$$h(T) = \max\{i \in \mathbb{N} \mid \exists n \in N.\ \ n \text{ è al livello } i\}$$
- Si dice larghezza o ampiezza di T il massimo numero di nodi di T che si trovano allo stesso livello:
$$b(T) = \max_{1 \leq i \leq h(T)} |\{n \in N \mid n \text{ è al livello } i\}|$$

albero binario

A. Ferrari

- ogni nodo ha al più ***due*** nodi ***figli***
  - figlio ***sinistro***
  - figlio ***destro***

Un albero $T = (N, r, \mathcal{B})$ è detto binario se:

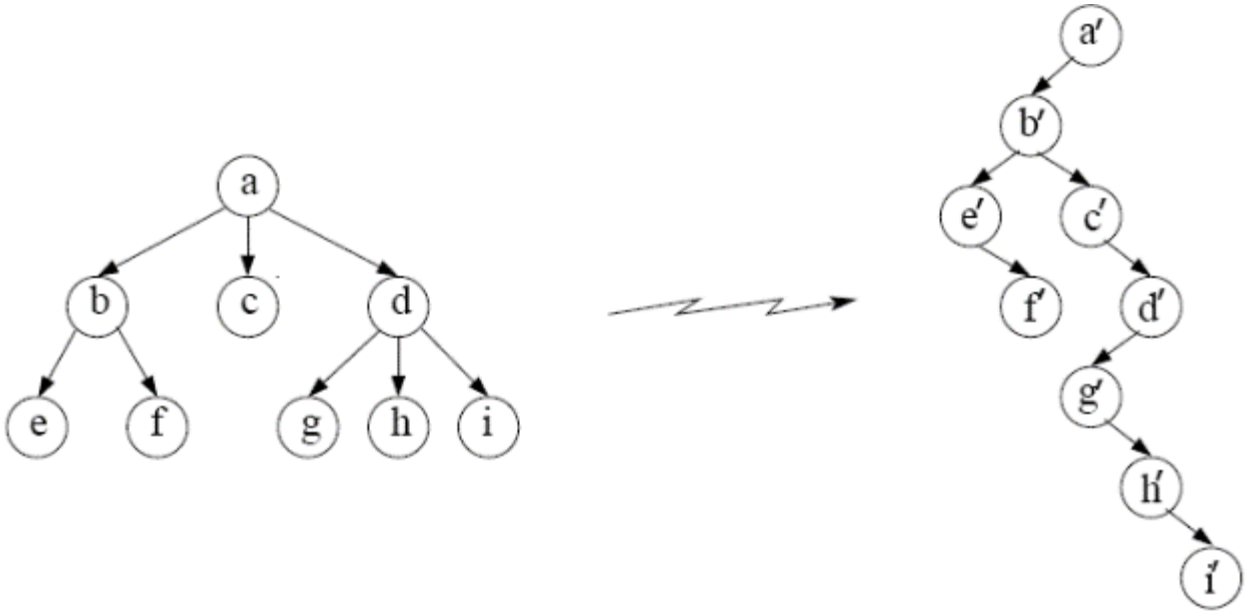
- $\mathcal{B} = \mathcal{B}_{sx} \cup \mathcal{B}_{dx}$.
- $\mathcal{B}_{sx} \cap \mathcal{B}_{dx} = \emptyset$.
- Per ogni $n, n_1, n_2 \in N$, se $(n, n_1) \in \mathcal{B}_{sx}$ (risp. $\mathcal{B}_{dx}$) ed $(n, n_2) \in \mathcal{B}_{sx}$ (risp. $\mathcal{B}_{dx}$), allora $n_1 = n_2$.

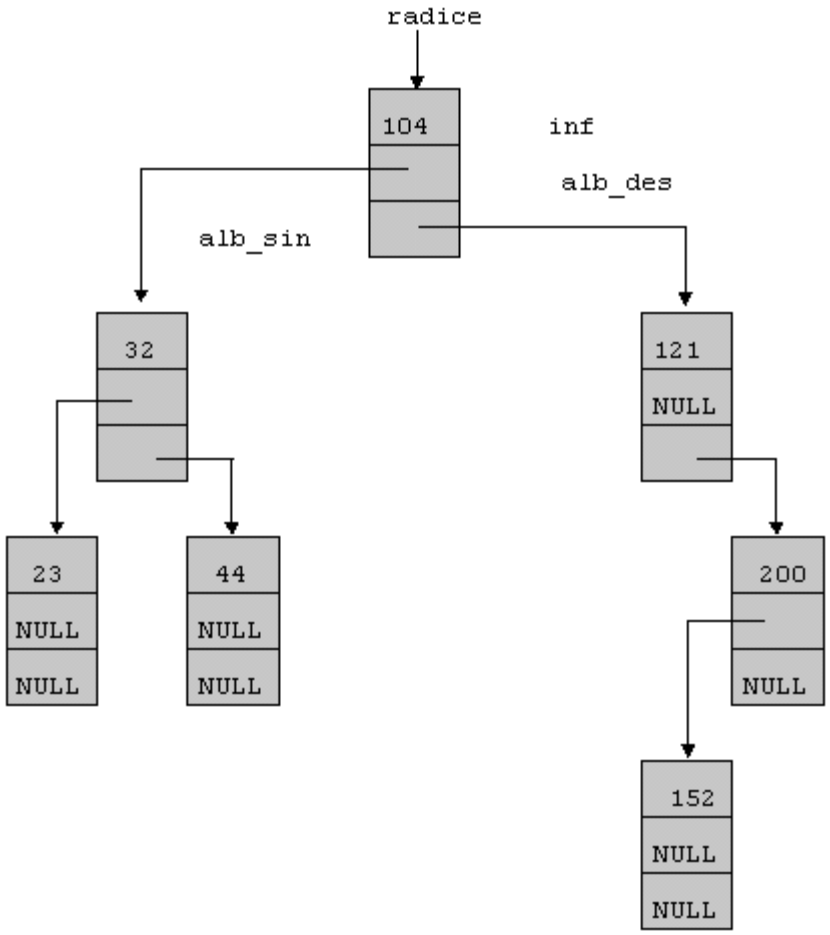
Se $(n, n') \in \mathcal{B}_{sx}$ (risp. $\mathcal{B}_{dx}$), allora n' è detto figlio sinistro (risp. destro) di n.

trasformazione
da albero ad albero binario

A. Ferrari

- Creare i nuovi nodi $n', n'_1, \ldots, n'_k$.
- Mettere n'_1 come figlio sinistro di n'.
- Per ogni $i = 1, \ldots, k-1$, mettere n'_{i+1} come figlio destro di n'_i.




radice
104
inf
alb_des
alb_sin
32
121
NULL
23
NULL
NULL
44
NULL
NULL
200
NULL
152
NULL
NULL

- definire una struttura dati che permetta di rappresentare un albero binario
- per semplicità l'informazione associata ad ogni nodo si considera che sia una stringa

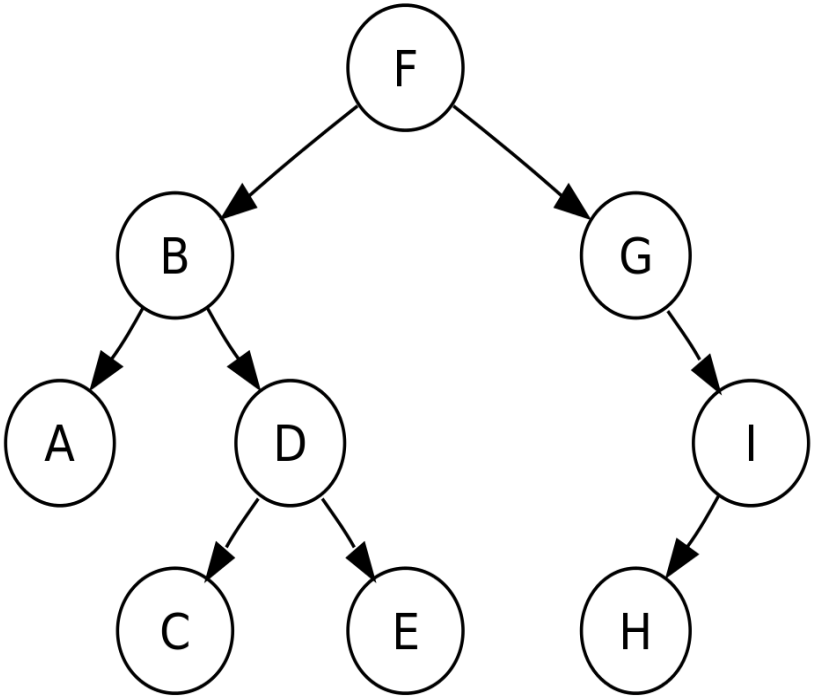
algoritmi di visita

- la visita consiste nell'accesso ***una e una sola volta*** a tutti i nodi dell'albero
- per gli alberi binari sono possibili più algoritmi di visita che generano ***sequenze diverse*** (per ordine) di nodi
  - visita in ordine ***anticipato***
  - visita in ordine ***simmetrico***
  - visita in ordine posticipato (***differito***)

visita in ordine anticipato

A. Ferrari

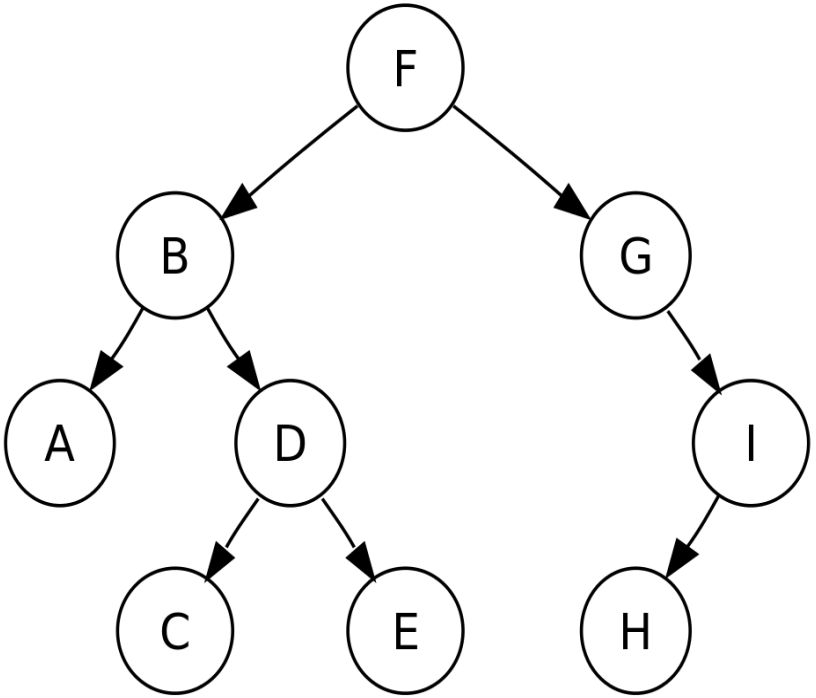
- visita la ***radice***
- visita il sottoalbero ***sinistro*** in ordine ***anticipato***
- visita il sottoalbero ***destro*** in ordine ***anticipato***
- *lista dei nodi*:



visita in ordine simmetrico

A. Ferrari

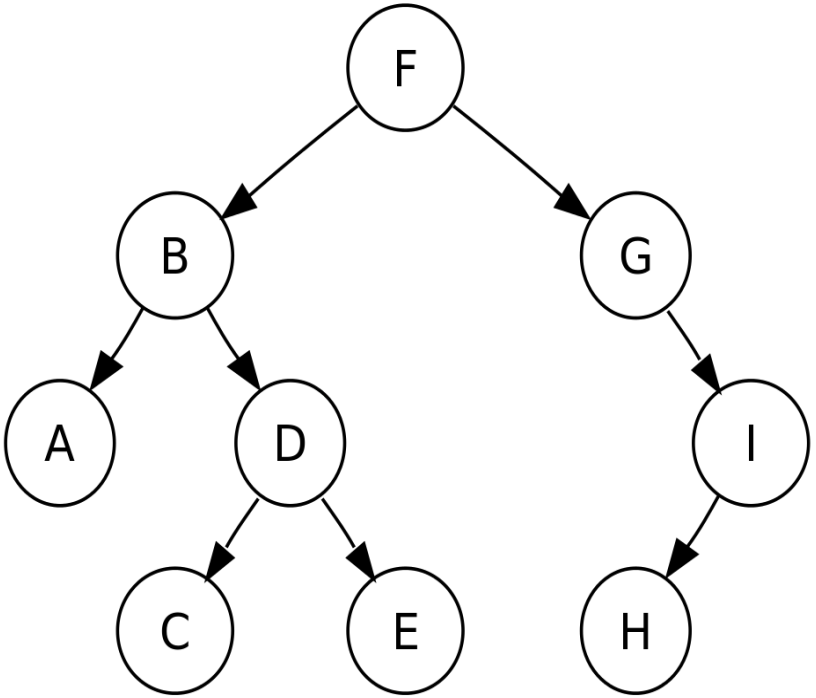
- visita il sottoalbero ***sinistro*** in ordine simmetrico
- visita la ***radice***
- visita il sottoalbero ***destro*** in ordine simmetrico
- *lista dei nodi*:



visita in ordine posticipato

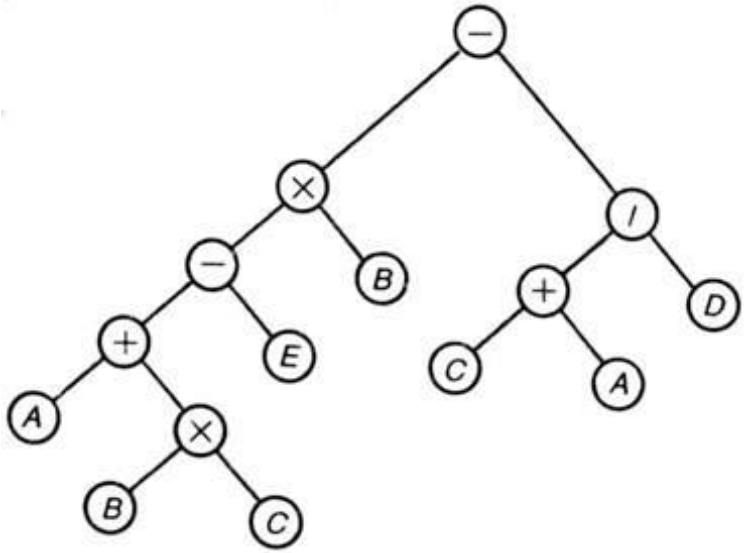
A. Ferrari

- visita il sottoalbero ***sinistro*** in ordine posticipato
- visita il sottoalbero ***destro*** in ordine posticipato
- visita la ***radice***
- *lista dei nodi*:



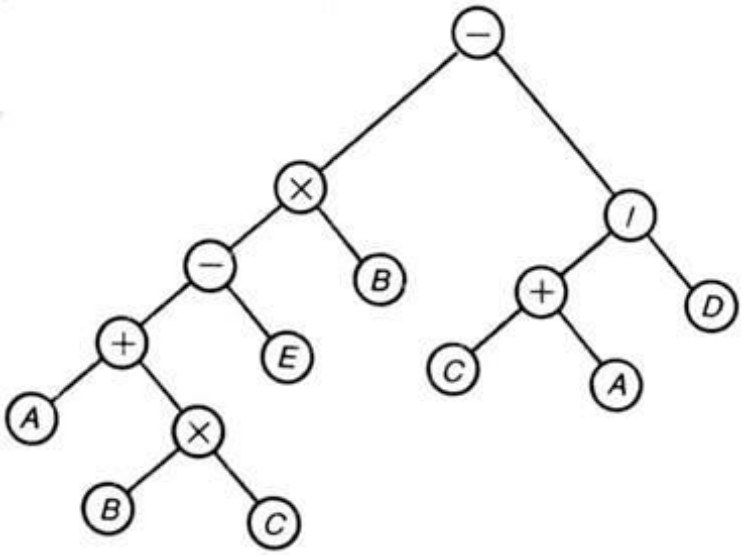
alberi ed espressioni

- ogni ***nodo*** che contiene un ***operatore*** è radice di un sottoalbero
- ogni ***foglia*** contiene un valore ***costante*** o una variabile



esercizio

- definire la sequenza di nodi che si ottiene visitando l'albero in ordine
  - anticipato
  - simmetrico
  - differito



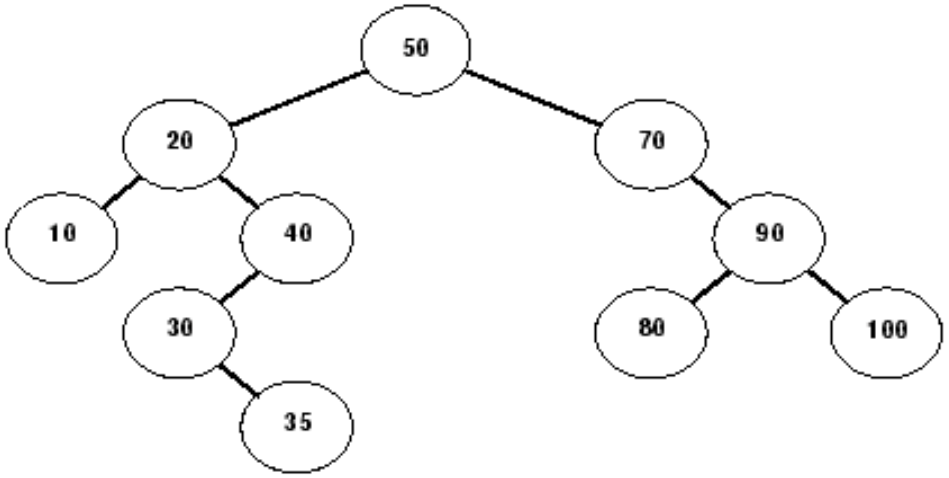
- Jan Łukasiewicz
  - http://it.wikipedia.org/wiki/Jan_%C5%81ukasiewicz
- Notazione polacca
  - http://it.wikipedia.org/wiki/Notazione_polacca

- con opportune modifiche si può adattare un qualunque algoritmo di visita per ottenere un ***algoritmo di ricerca***
- nel caso ***pessimo*** la ricerca attraverserà tutti nodi dell'albero quindi avrà complessità ***O(n)***

alberi binari di ricerca

A. Ferrari

- un ***albero binario di ricerca*** è un albero binario tale che:
  - per ogni nodo che contiene una chiave di valore ***k***
  - ogni nodo del suo sottoalbero ***sinistro*** contiene una chiave di valore ≤ ***k***
  - ogni nodo del suo sottoalbero ***destro*** contiene una chiave di valore ≥ ***k***



ricerca in alberi binari di ricerca

- non è necessario visitare tutti i nodi
- basta fare un ***unico percorso*** tra quelli che partono dalla radice, scendendo ad ogni nodo incontrato che non contiene il valore dato a sinistra o a destra a seconda che il valore dato sia minore o maggiore, rispettivamente, della chiave contenuta nel nodo
- la complessità della ricerca dipende quindi dalla ***profondità*** dell'albero

esercizio 1

- implementare la ***struttura dati*** che permette di implementare un ***albero binario***
- implementare gli ***algoritmi*** di
  - ***inserimento***
  - ***visita*** in ordine anticipato, simmetrico, differito
  - ***ricerca*** (dato il valore dell'informazione restituire il Nodo)
  - ***eliminazione*** di un nodo

esercizio 2

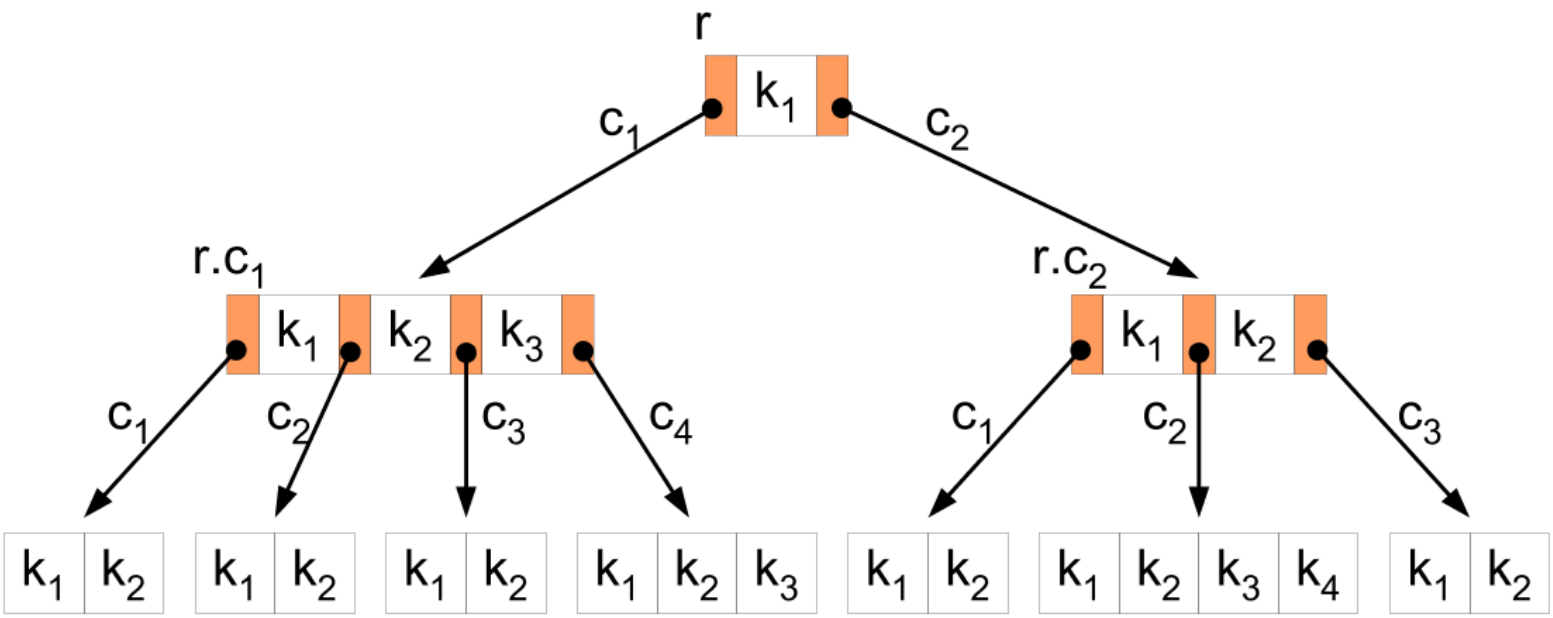
- implementare la ***struttura dati*** che permetta di implementare un ***albero binario di ricerca***
- implementare gli algoritmi di
  - ***inserimento***
  - ***visita*** in ordine anticipato, simmetrico, differito
  - ***ricerca*** (dato il valore dell'informazione restituire il Nodo)
  - ***eliminazione*** di un nodo (vedi suggerimenti)

eliminazione di un nodo

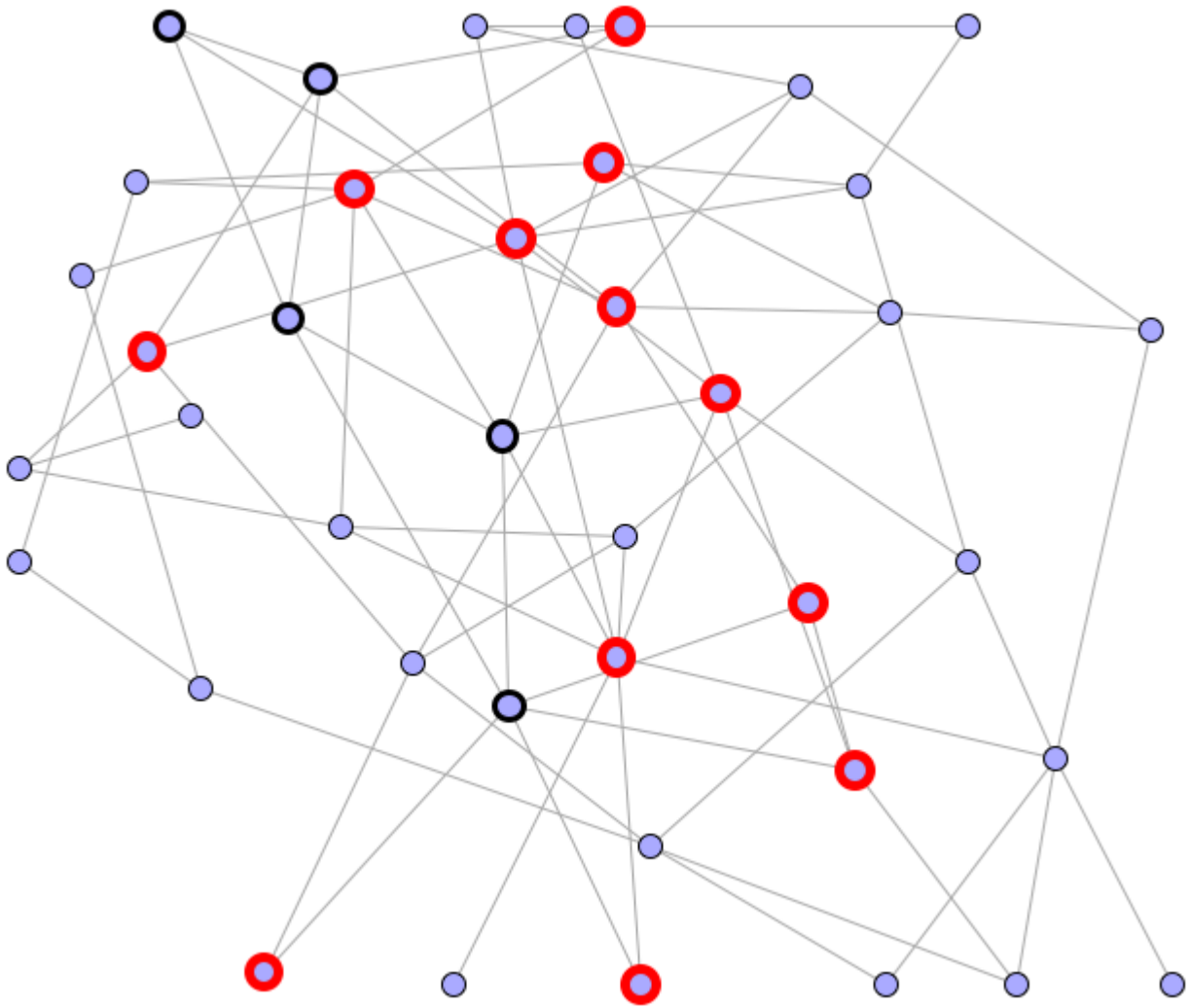
- *l'algoritmo di rimozione di un valore da un albero binario di ricerca deve garantire che l'albero binario ottenuto a seguito della rimozione sia ancora di ricerca*
  - *se il nodo contenente il valore da rimuovere è una foglia, basta eliminarlo*
  - *se il nodo contenente il valore da rimuovere ha un solo figlio, basta eliminarlo collegando suo padre direttamente a suo figlio*
  - *se il nodo contenente il valore da rimuovere ha ambedue i figli, si procede sostituendone il valore con quello del nodo più a destra del suo sottoalbero sinistro, in quanto tale nodo contiene la massima chiave minore di quella del nodo da rimuovere (in alternativa, si può prendere il nodo più a sinistra del sottoalbero destro)*

B-Albero (B-Tree)

- struttura dati che permette la rapida ***localizzazione*** nei file (*Records o keys*)
- deriva dagli alberi di ricerca, in quanto ogni ***chiave*** appartenente al sottoalbero sinistro di un nodo è di valore inferiore rispetto a ogni chiave appartenente ai sottoalberi alla sua destra
- è garantito il ***bilanciamento***: per ogni nodo, le altezze dei sottoalberi destro e sinistro differiscono al più di una unità
- utilizzati spesso nell'ambito dei database, in quanto permettono di accedere ai nodi in maniera efficiente sia nel caso essi siano disponibili in memoria centrale o in memoria di massa


r
k1
c1
c2
r.c1
k1
k2
k3
r.c2
k1
k2
c1
c2
c3
c4
c1
c2
c3
k1
k2
k1
k2
k1
k2
k1
k2
k3
k1
k2
k1
k2
k3
k4
k1
k2

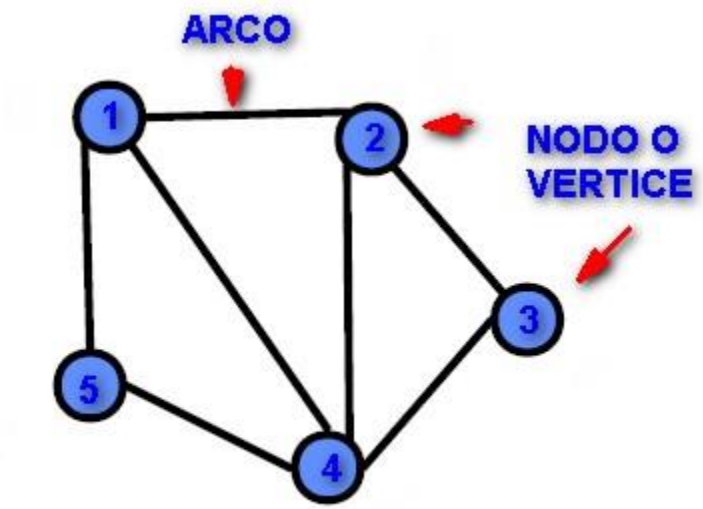
GRAFI



grafo
definizione non formale

A. Ferrari

- un ***grafo*** è un insieme di nodi
- ogni ***nodo*** è collegato mediante ***archi*** ad altri nodi (*nodi adiacenti*)
- archi possono essere unidirezionali (***grafo orientato***) o bidirezionali (***grafo non orientato***)
- agli gli archi è possibile associare un valore numerico (***grafo pesato***)
-



grafo
definizioni formale (1)

A. Ferrari

- Si dice grafo diretto o orientato una coppia $G = (V, \mathcal{E})$ dove V è un insieme di vertice ed $\mathcal{E}$ è una relazione binaria su V.

- Sia $G = (V, \mathcal{E})$ un grafo diretto:
  - Se $(v, v') \in \mathcal{E}$, allora si dice che v' è adiacente a v o, equivalentemente, che c'è un arco da v a v'.
  - Si dice grado uscente di $v \in V$ il numero di vertici adiacenti a v:
    $$d_{\mathrm{o}}(v) = |\{v' \in V \mid (v, v') \in \mathcal{E}\}|$$
  - Si dice grado entrante di $v \in V$ il numero di vertici ai quali v è adiacente:
    $$d_{\mathrm{i}}(v) = |\{v' \in V \mid (v', v) \in \mathcal{E}\}|$$
  - Si dice grado di $v \in V$ il numero di archi in cui v è coinvolto:
    $$d(v) = d_{\mathrm{o}}(v) + d_{\mathrm{i}}(v)$$
  - Se $v \in V$ è tale che:
    - $d_{\mathrm{o}}(v) = 0$ e $d_{\mathrm{i}}(v) > 0$ allora si dice che v è un vertice terminale;
    - $d_{\mathrm{i}}(v) = 0$ e $d_{\mathrm{o}}(v) > 0$ allora si dice che v è un vertice iniziale;
    - $d(v) = 0$ allora si dice che v è un vertice isolato.
  - Si dice grado di G il massimo grado di un vertice di G:
    $$d(G) = \max\{d(v) \mid v \in V\}$$
  - Si dice che G è completo se $\mathcal{E} = V \times V$.

grafo
definizioni formale (2)

A. Ferrari

- Sia $G = (V, \mathcal{E})$ un grafo diretto:
  - Siano $v_1, v_2 \in V$. Si dice che v_2 è raggiungibile da v_1 se esiste un percorso da v_1 a v_2, cioè se esistono $v'_1, \ldots, v'_k \in V$ con $k \geq 2$ tali che $v'_1 = v_1$, $(v'_i, v'_{i+1}) \in \mathcal{E}$ per ogni $1 \leq i \leq k-1$, e $v'_k = v_2$.
  - Si dice che un percorso è semplice se tutti i vertici che lo compongono sono distinti, eccetto al più il primo e l'ultimo vertice.
  - Si dice che un percorso è un ciclo se il suo primo vertice coincide con il suo ultimo vertice.
  - Si dice che G è connesso se per ogni $v_1, v_2 \in V$ esiste un percorso da v_1 a v_2 o da v_2 a v_1.
  - Si dice che G è fortemente connesso se per ogni $v_1, v_2 \in V$ esistono un percorso da v_1 a v_2 e un percorso da v_2 a v_1.

- Si dice grafo pesato una tripla $G = (V, \mathcal{E}, w)$ dove $G = (V, \mathcal{E})$ è un grafo e $w : \mathcal{E} \longrightarrow \mathbb{R}$ è una funzione detta peso. Il peso associato ad un arco rappresenta di solito un tempo, una distanza, una capacità o un guadagno/perdita.

problema dei ponti di Königsberg

A. Ferrari

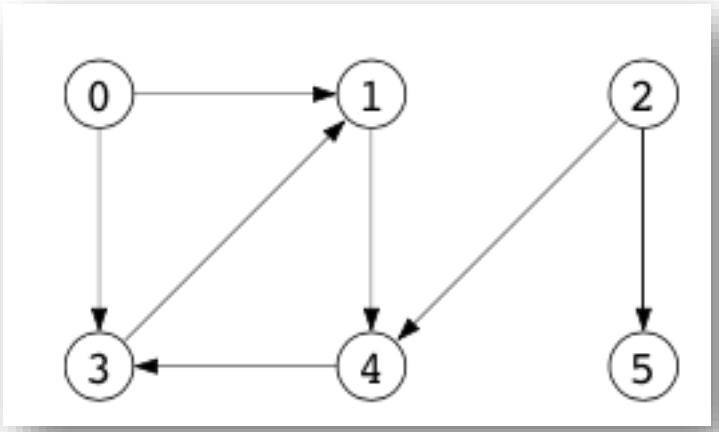
- il problema dei ***sette ponti*** di Königsberg è un problema ispirato da una città reale e da una situazione concreta
- *Königsberg è percorsa dal fiume Pregel e da suoi affluenti e presenta due estese isole che sono connesse tra di loro e con le due aree principali della città da sette ponti*
- nel corso dei secoli è stata più volte proposta la questione se sia possibile con una passeggiata seguire un ***percorso*** che attraversi ***ogni ponte una e una volta soltanto e tornare al punto di partenza***
- nel 1736 Leonhard Euler affrontò tale problema, dimostrando che la passeggiata ipotizzata ***non era possibile***

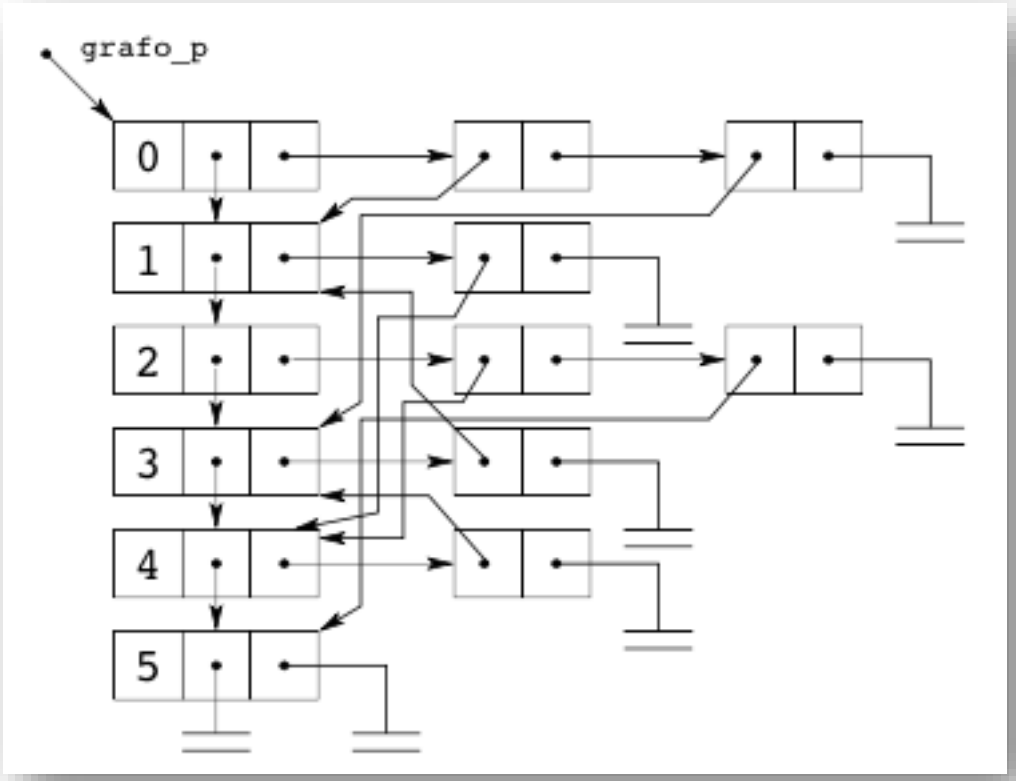
astrazione - modellizzazione

- Eulero formula il problema in termini di ***teoria dei grafi***, ***astraendo*** dalla situazione specifica di Königsberg
  - *eliminazione di tutti gli aspetti contingenti ad esclusione delle aree urbane delimitate dai bracci fluviali e dai ponti che le collegano*
  - sostituzione
    - ogni **area urbana** diventa un ***vertice*** (nodo)
    - ogni **ponte** diventa un ***arco***

grafo: implementazione con lista di adiacenza

A. Ferrari





grafo: implementazione con lista di adiacenza

- il grafo viene rappresentato come una struttura dati dinamica reticolare detta ***lista di adiacenza***, formata da una ***lista primaria dei vertici*** e più ***liste secondarie degli archi***
  - la lista primaria contiene un elemento per ciascun vertice del grafo, il quale contiene a sua volta la testa della relativa lista secondaria
  - la lista secondaria associata ad un vertice descrive tutti gli archi uscenti da quel vertice

problema 1

- definire una struttura dati che permetta di implementare un grafo orientato mediante lista di adiacenza

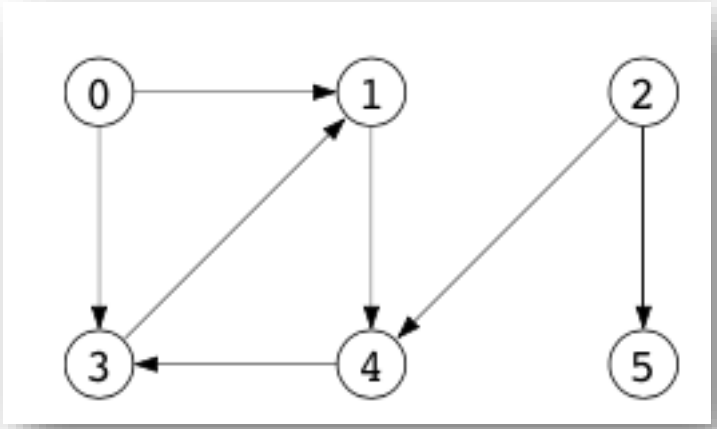
grafo: implementazione con matrice di adiacenza

A. Ferrari

- se la struttura di un grafo non cambia oppure è importante fare accesso rapidamente alle informazioni contenute nel grafo, allora conviene ricorrere ad una rappresentazione a ***matrice di adiacenza***
- la matrice ha tante ***righe*** e tante ***colonne*** quanti sono i ***vertici***
- l'elemento di indici ***i*** e ***j*** vale ***1*** (*true*) se esiste un arco dal vertice i al vertice j, ***0*** (*false*) altrimenti
- per i grafi pesati si può sostituire il valore ***1*** con il ***peso*** del grafo

grafo: implementazione con matrice di adiacenza

A. Ferrari



0	1	0	1	0	0
0	0	0	0	1	0
0	0	0	0	1	1
0	1	0	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

problema 2

- definire una struttura dati che permetta di implementare un grafo orientato mediante matrice di adiacenza

problema 3

- dato un grafo implementato mediante matrice di adiacenza, dati due nodi A e B, verificare se il nodo B è raggiungibile dal nodo A