# Corso di Tecniche di Programmazione
# cdl Medicina e Chirurgia TD
# II anno

# Matrici

# Ripasso sugli array

Un array/lista/vettore è una sequenza di variabili

Creare una lista, esempi:

- lista = [] #lista vuota
- lista = [3,4,5] #lista con 3 elementi
- Accedere ad un elemento della lista:
- a = lista[0] #assegna il primo elemento alla variable a

- Numero di elementi nella lista:
  - n = len(lista) #assegna ad n la lunghezza della lista
- Aggiungere elementi in fondo alla lista:
  - lista.append(10) #lista avrà un elemento in più, in ultima posizione

# La funzione range

- Due varianti:

```
range(n)     #restituisce la lista [0,1,…,n-1]

range(i,j) #restituisce la lista [i,i+1,…,j-1]
```

```
range(0,4) -> [0,1,2,3]
range(2,7) -> [2,3,4,5,6]
```

- La lista restituita è «speciale»:
  - Non può essere modificata

- Viene spesso usata per costruire dei cicli:

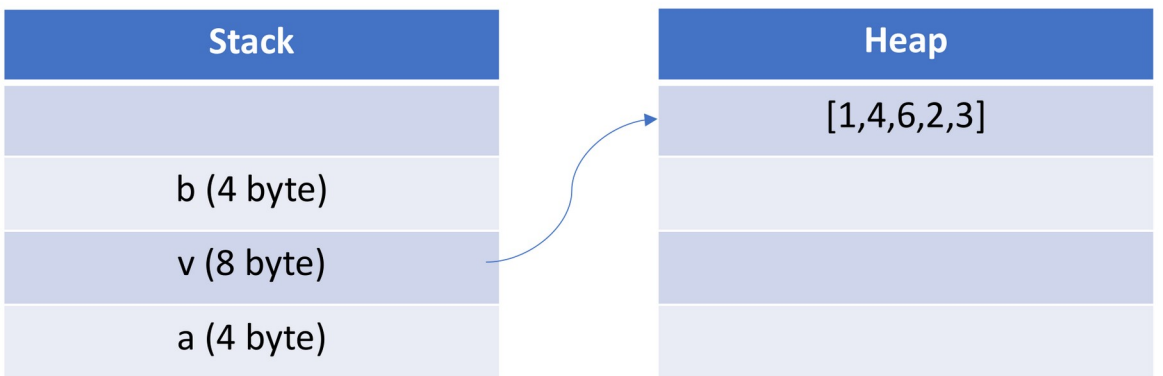
```
for x in range(10): #stampa i numeri da 0 a 9
      print(x)
```

# Leggere valori da tastiera e inserirli in una lista

```
n = int(input('Inserire il numero di elementi: '))

v = []

for i in range(0,n):
    a = int(input('Inserire un elemento: '))
    v.append(a)

print(v)
```

# Come sono memorizzate le liste in memoria?

```
a = 1
v = [1,4,6,2,3]
b = 5
```



# Matrici in Python

- In Python, le matrici vengono gestite come liste di liste, esempio:

```
m = [ [1,2], [4,2], [3,10] ]
```

| | |
|---|---|
| 1 | 2 |
| 4 | 2 |
| 3 | 10 |

- La matrice ha 3 righe e due colonne:

```
m[0] #è la lista corrispondente alla prima riga
m[0][1] #elemento in prima riga, seconda colonna
```

- Numero di righe e colonne di una matrice:

```
num_righe = len(m)
num_colonne = len(m[0])
```

# Matrici in Python

- In Python, le matrici vengono gestite come liste di liste, esempio:

```
m = [ [1,2], [4,2], [3,10] ]
```

| | |
|---|---|
| 1 | 2 |
| 4 | 2 |
| 3 | 10 |

- La matrice ha 3 righe e due colonne:

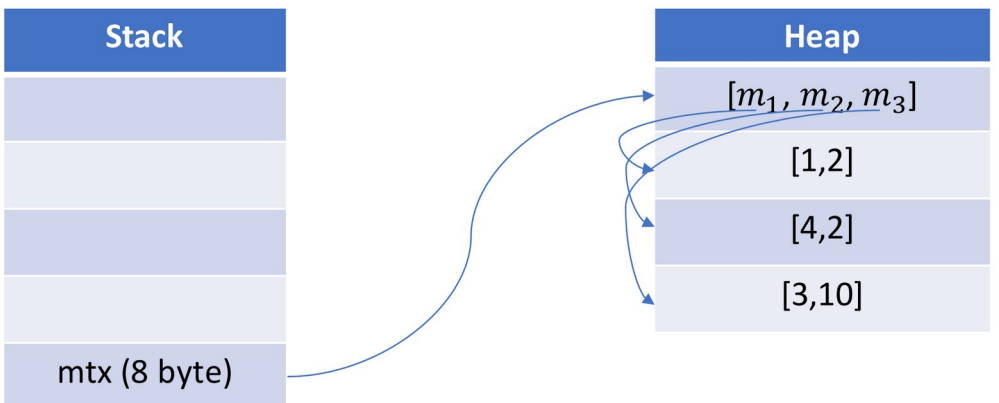
```
m[0] #è la lista corrispondente alla prima riga
m[0][1] #elemento in prima riga, seconda colonna
```

- Numero di righe e colonne di una matrice:

```
num_righe = len(m)
num_colonne = len(m[0])
```

# Come sono memorizzate le matrici in memoria?

```
mtx = [ [1,2], [4,2], [3,10] ]
```



# Leggere una matrice da tastiera

```
def leggiMatrice():
    n = int(input('Inserire numero di righe: '))
    m = int(input('Inserire numero di colonne:'))

    mtx = [] #inizializzo matrice vuota

    #prepariamo prima le righe
    for i in range(n):
        mtx.append([])

    # ora mtx = [ [], [], ..., [] ]

    #ora leggiamo la matrice
    for i in range(n):
        for j in range(m):
            valore = int(input('Inserire valore: '))
            mtx[i].append(valore)

    return mtx
```

# Sommare gli elementi di una matrice di interi

```
def sommamatrice(mtx):
    n = len(mtx) #num. righe
    m = len(mtx[0]) #num. colonne
    s = 0

    for i in range(n):          (n volte)
        for j in range(m):      (m volte)
            s = s + mtx[i][j]

    return s
```

| Stack |
|---|
| j (4 byte) |
| i (4 byte) |
| s (4 byte) |
| m (4 byte) |
| n (4 byte) |
| mtx (8 byte) |
| RA (8 byte) |

| Heap |
|---|
| $[m_1, \dots, m_n]$ |
| [...] |
| ... |
| [...] |

| Tempo T(n,m) | Spazio S(n,m) |
|---|---|
| $\Theta(1)$ (preambolo) | $\Theta(1)$ (preambolo) |
| $\Theta(1)$ (ass. n,m,s) | $\Theta(1)$ (variabili n,m,s,i,j) |
| $n \cdot m \cdot \Theta(1)$ (costo for innestati) | |
| costo istruzione s = s + mtx[i][j] | |
| $\Theta(1)$ (return) | |
| | |
| | |
| CM e CP: $\Theta(n \cdot m)$ | CM e CP: $\Theta(1)$ |

Caso Migliore e Caso Peggiore coincidono:

T(n,m) =

$$\Theta(1) + \Theta(1) + n \cdot m \cdot \Theta(1) + \Theta(1) =$$

$$\Theta(1) + n \cdot m \cdot \Theta(1)$$

T(n,m) è sia un $O(n \cdot m)$ che un $\Omega(n \cdot m)$ quindi è $\Theta(n \cdot m)$

- Calcoli di trasposta di matrice, calcolo di uguaglianza e triangolare superiore...
-

# Liste e strutture dati a lista

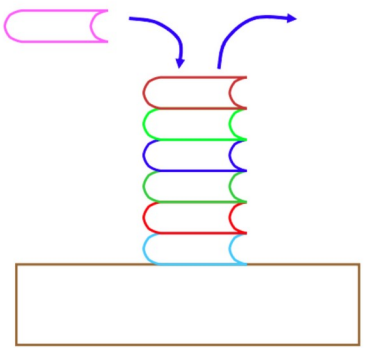
# Pile e code

# Pile (o Stack)

- Struttura dati che consente l'accesso ai dati in essa contenuti in modalita LIFO (Last In First Out)

- Ad esempio, una catasta di libri è una pila

- L'ultimo libro inserito è il primo che viene prelevato



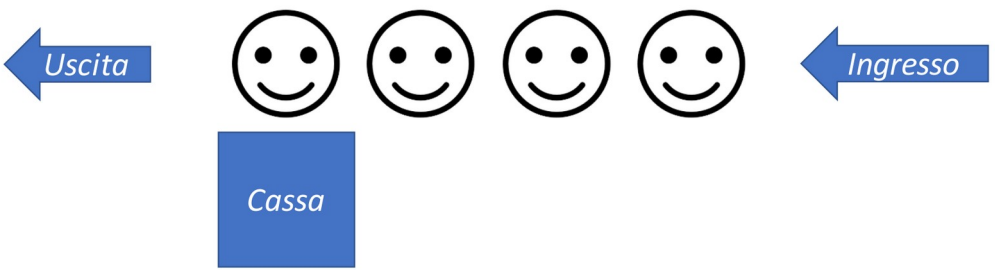
# Pile in Python

```
pila = []

pila.append(2) (push)
pila.append(4) (push)
pila.append(5) (push)

testa = pila.pop() (pop)

TestaSenzaRimuoverla = pila[-1] (peek)

n = len(pila) (size)
```

Equivale a pila[len(pila)-1]

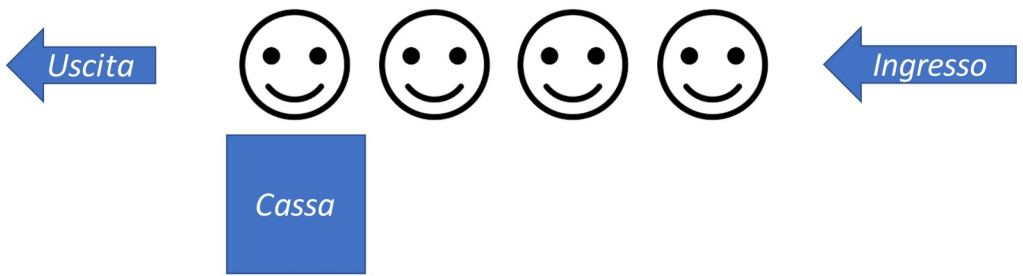
# Code (o Queue)

- Struttura dati che consente l'accesso ai dati in essa contenuti in modalita FIFO (First In First Out)
- Ad esempio, una fila di persone alla cassa è una coda
- La prima persona ad accedere alla cassa è anche la prima ad andarsene



# Code (o Queue) - Operazioni Utili

- Aggiungere un elemento in fondo alla coda (**enqueue**)
- Estrarre l'elemento in testa alla coda (**dequeue**)
- Leggere l'elemento in testa senza rimuoverlo (**peek**)
- Conoscerne la dimensione (**size**)

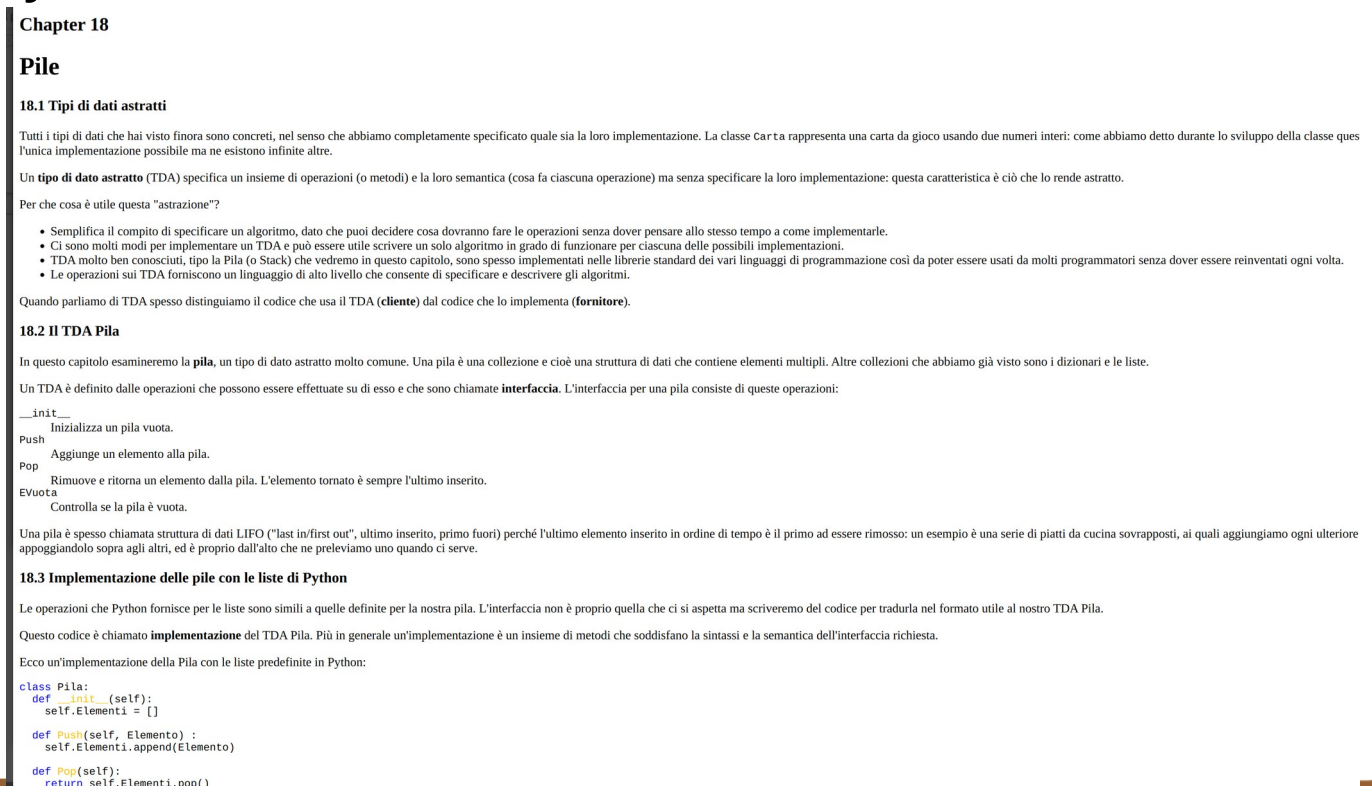


# Code in Python

```
coda= []

coda.append(2) (enqueue)
coda.append(4) (enqueue)
coda.append(5) (enqueue)

testa = coda.pop(0) (dequeue)

testaSenzaRimuoverla = coda[0] (peek)

n = len(coda) (size)
```

# Riferimenti implementativi utili

- https://www.python.it/doc/Howtothink/Howtothink-html-it/chap18.htm



## Chapter 18

# Pile

### 18.1 Tipi di dati astratti

Tutti i tipi di dati che hai visto finora sono concreti, nel senso che abbiamo completamente specificato quale sia la loro implementazione. La classe `Carta` rappresenta una carta da gioco usando due numeri interi: come abbiamo detto durante lo sviluppo della classe ques l'unica implementazione possibile ma ne esistono infinite altre.

Un **tipo di dato astratto** (TDA) specifica un insieme di operazioni (o metodi) e la loro semantica (cosa fa ciascuna operazione) ma senza specificare la loro implementazione: questa caratteristica è ciò che lo rende astratto.

Per che cosa è utile questa "astrazione"?

- Semplifica il compito di specificare un algoritmo, dato che puoi decidere cosa dovranno fare le operazioni senza dover pensare allo stesso tempo a come implementarle.
- Ci sono molti modi per implementare un TDA e può essere utile scrivere un solo algoritmo in grado di funzionare per ciascuna delle possibili implementazioni.
- TDA molto ben conosciuti, tipo la Pila (o Stack) che vedremo in questo capitolo, sono spesso implementati nelle librerie standard dei vari linguaggi di programmazione così da poter essere usati da molti programmatori senza dover essere reinventati ogni volta.
- Le operazioni sui TDA forniscono un linguaggio di alto livello che consente di specificare e descrivere gli algoritmi.

Quando parliamo di TDA spesso distinguiamo il codice che usa il TDA (**cliente**) dal codice che lo implementa (**fornitore**).

### 18.2 Il TDA Pila

In questo capitolo esamineremo la **pila**, un tipo di dato astratto molto comune. Una pila è una collezione e cioè una struttura di dati che contiene elementi multipli. Altre collezioni che abbiamo già visto sono i dizionari e le liste.

Un TDA è definito dalle operazioni che possono essere effettuate su di esso e che sono chiamate **interfaccia**. L'interfaccia per una pila consiste di queste operazioni:

`__init__`
: Inizializza un pila vuota.

`Push`
: Aggiunge un elemento alla pila.

`Pop`
: Rimuove e ritorna un elemento dalla pila. L'elemento tornato è sempre l'ultimo inserito.

`EVuota`
: Controlla se la pila è vuota.

Una pila è spesso chiamata struttura di dati LIFO ("last in/first out", ultimo inserito, primo fuori) perché l'ultimo elemento inserito in ordine di tempo è il primo ad essere rimosso: un esempio è una serie di piatti da cucina sovrapposti, ai quali aggiungiamo ogni ulteriore appoggiandolo sopra agli altri, ed è proprio dall'alto che ne preleviamo uno quando ci serve.

### 18.3 Implementazione delle pile con le liste di Python

Le operazioni che Python fornisce per le liste sono simili a quelle definite per la nostra pila. L'interfaccia non è proprio quella che ci si aspetta ma scriveremo del codice per tradurla nel formato utile al nostro TDA Pila.

Questo codice è chiamato **implementazione** del TDA Pila. Più in generale un'implementazione è un insieme di metodi che soddisfano la sintassi e la semantica dell'interfaccia richiesta.

Ecco un'implementazione della Pila con le liste predefinite in Python:

```
class Pila:
  def __init__(self):
    self.Elementi = []

  def Push(self, Elemento) :
    self.Elementi.append(Elemento)

  def Pop(self):
    return self.Elementi.pop()
```

# PILA

```
class Pila:
def __init__(self):
    self.Elementi = []

  def Push(self, Elemento) :
    self.Elementi.append(Elemento)

  def Pop(self):
    return self.Elementi.pop()

  def EVuota(self):
    return (self.Elementi == [])
```

# Pila Utilizzo

- >>> P = Pila()
- >>> P.Push(54)
- >>> P.Push(45)
- >>> P.Push("+")
-
-

# Alberi, visita degli alberi,inserimento e cancellazione

# Inserzione

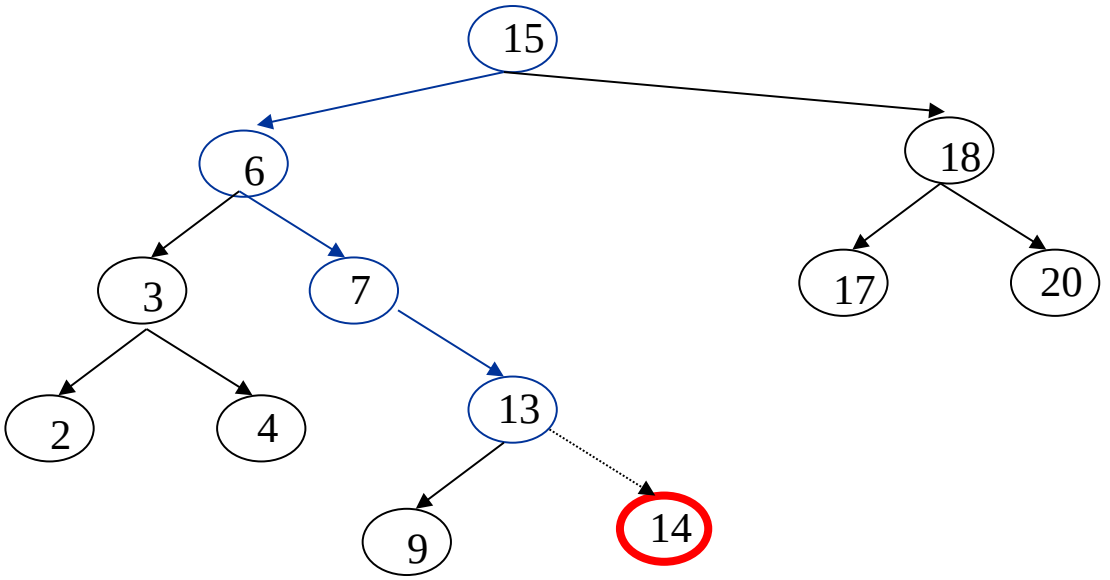
Per inserire un nuovo valore k in un albero binario di ricerca:

- **si passa alla procedura un nodo z tale che key[z]=k, e left[z]=right[z]=p[k]=NIL**
- **si modificano i campi del nuovo nodo per inserirlo opportunamente all'interno dell'albero binario di ricerca**

l'idea è di muoversi all'interno dell'albero a partire dalla radice spostandosi sul sottoalbero destro o sinistro come appropriato (confrontando le chiavi)

una volta arrivati ad una foglia si inserisce il nuovo nodo

# Visualizzazione



# Pseudocodice Inserzione

```
Tree-Insert(T,z)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4  do      y ← x
5  if key[z]<key[x]
6  then x ← left[x]
7  else x ← right[x]
8  p[z] ← y
9  if y=NIL
10 then    root[T] ← z
11 else if key[z] < key[y]
12 then    left[y] ← z
13 else    right[y] ← z
```

# Cancellazione

La procedura di cancellazione è più laboriosa in quanto si deve tenere conto di tre casi possibili

dato un nodo z i casi sono:

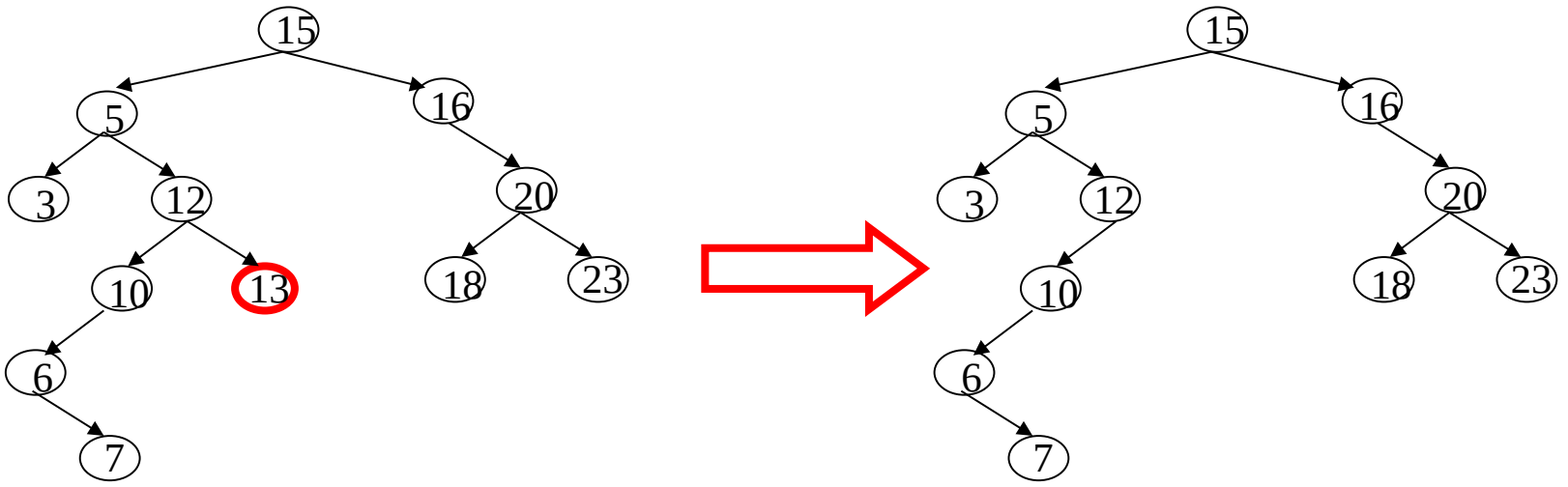
- **z non ha figli**
- **z ha un unico figlio**
- **z ha due figli**

Nel primo caso si elimina direttamente il nodo z

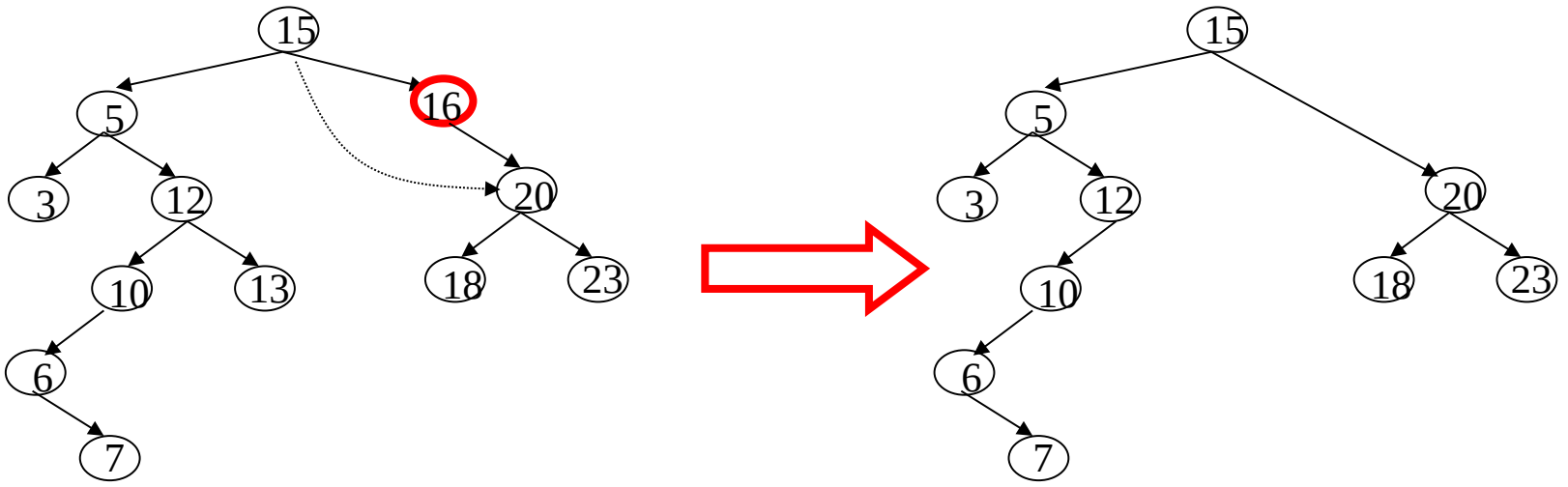
il secondo caso è identico al caso di eliminazione di un nodo da una lista concatenata

Nel terzo caso si determina il successore di z e lo si sostituisce a z

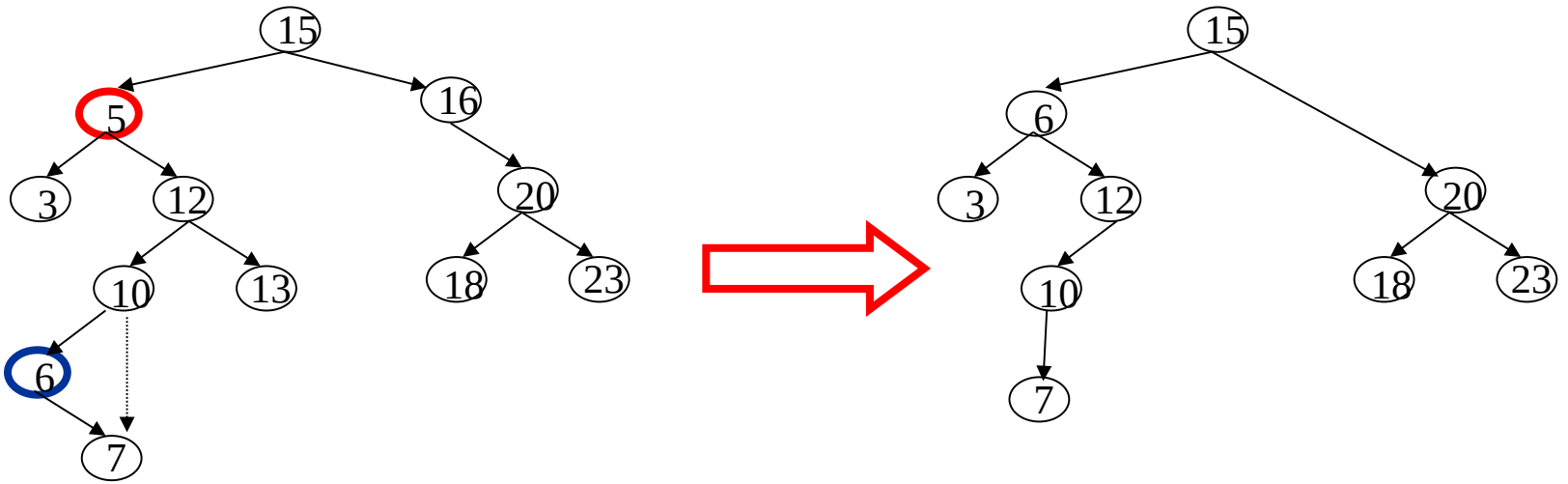
# Visualizzazione caso 1



# Visualizzazione caso 2



# Visualizzazione caso 3



# PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL //identifica il nodo da cancellare o sostituire
2  then    y ← z
3  else    y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else    x ← right[y]
7  if x ≠ NIL //ripristina il padre (cancellazione implicita)
8  then p[x] ← p[y]
9  if p[y] = NIL  //ripristina i figli corretti
10 then    root[T] ← x
11 else    if y=left[p[y]]
12 then    left[p[y]] ← x
13 else    right[p[y]] ← x
14 if y ≠ z
15 then    key[z] ← key[y]
17 return y //per eventuale deallocazione
```

# Implementazione C++

```
#include <iostream>
using namespace std;

template<class T, class  LessClass >
class TreeClass{
private:
  struct Node{Node *left, *right, * parent; T key;};
  Node *root;
public:
  TreeClass():root(0){}
  void insert(T);
  void print(){p_print(root); cout<<endl;}
  bool search(T user_key){return p_search(root, user_key);}
  T minimum();
  T maximum();
private:
  void p_print(Node *);
  bool p_search(Node * x, T user_key);
};
```

# Implementazione C++

```
template<class T, class  LessClass >
void TreeClass<T, LessClass>::p_print(Node *x){
  if(x!=0){
    p_print(x->left);
    cout<<x->key<<" ";
    p_print(x->right);
  }
}
template<class T, class  LessClass >
bool TreeClass<T, LessClass>::p_search(Node * x, T usr_key){
  LessClass less;
  if(x==0) return false;
  if(!less(x->key,usr_key) && !less(usr_key,x->key)) return true; //ugualianza
  if(less(usr_key,x->key)) p_search(x->left, usr_key);
  else p_search(x->right, usr_key);
}
```

```
template<class T, class LessClass >
void TreeClass<T,LessClass>::insert(T usr_key){
  LessClass less;
  //inizializzazione del nodo da aggingere
  Node * z=new Node;
  z->key=usr_key; z->left=0; z->right=0; z->parent=0;

  //ricerca della giusta posizione di inserzione
  Node * y=0;
  Node * x=root;
  while(x != 0){
    y=x;
    if(less(z->key, x->key)) x=x->left;
    else x=x->right;
  }
  //settaggio dei puntatori
  z->parent=y;
  if(y==0) root=z;
  else if(less(z->key, y->key)) y->left=z;
  else y->right=z;
}
```

# Implementazione C++

```
template<class T, class  LessClass >
T TreeClass<T, LessClass>::minimum(){
  Node * x=root;
  while(x->left !=0) x=x->left;
  return x->key;
}

template<class T, class  LessClass >
T TreeClass<T, LessClass>::maximum(){
  Node * x=root;
  while(x->right !=0) x=x->right;
  return x->key;
}
```

# Implementazione C++

```
template<class T> struct LessClass{
  bool operator()(const T & a, const T & b)const{return a<b;}
};
int main(){
  //integer example
  int v[]={2,5,8,1,3,4,7,9,6,0};
  TreeClass<int, LessClass<int> > T;

  for(int i=0;i<10;i++)
    T.insert(v[i]);
  T.print();

  cout<<"Seraching 5:"<<
  (T.search(5)? "Found":"Not found")<<endl;
  cout<<"Seraching 11:"<<
  (T.search(11)? "Found":"Not found")<<endl;
  cout<<"Searching maximum:"<<T.maximum()<<endl;
  cout<<"Searching minimum:"<<T.minimum()<<endl;
```

# Implementazione C++

```
  //char example
  char c[]="this_is_a.";
  TreeClass<char, LessClass<char> > Tc;

  for(int i=0;i<10;i++)
    Tc.insert(c[i]);
  Tc.print();

  cout<<"Seraching s:"<<
  (Tc.search('s')? "Found":"Not found")<<endl;
  cout<<"Seraching v:"<<
  (Tc.search('v')? "Found":"Not found")<<endl;
  cout<<"Searching maximum:"<<Tc.maximum()<<endl;
  cout<<"Searching minimum:"<<Tc.minimum()<<endl;

  cout<<endl;
  return 0;
}
```

# Grafi e visite di Grafi

# Grafi

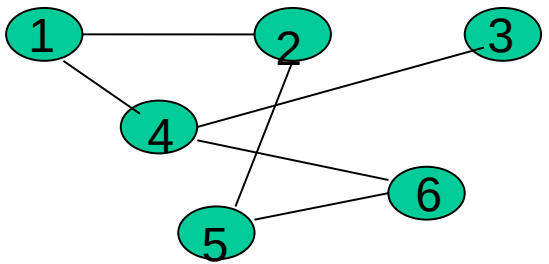
- Definizione di Grafi
- Rappresentazione in memoria
- Algoritmi di visita di grafi
- Algoritmo di chiusura transitiva
- …
-

# Definizioni

- Un grafo G=(V,E) consiste in:
- - un insieme V di nodi
- - un insieme E di nodi, detti archi. Ogni arco connette due nodi
- Un grafo è orientato se ogni arco (i,j) in E stabilisce un collegamento tra i e j (e non vieversa)
- Due nodi i,j sono detti adiacenti se esiste l'arco (i,j)
- Per ogni coppia di nodi (i,j) in E si dice che si dice che l'arco tra i e j è arco incidente tra i due nodi
- Dato un insieme di archi (i,j)…(k,l) si definisce cammino se unisce i nodi da i ad l passando per i nodi intermedi

# Esempio

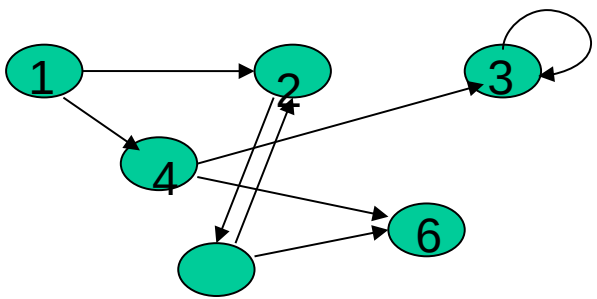
grafo




non orientato

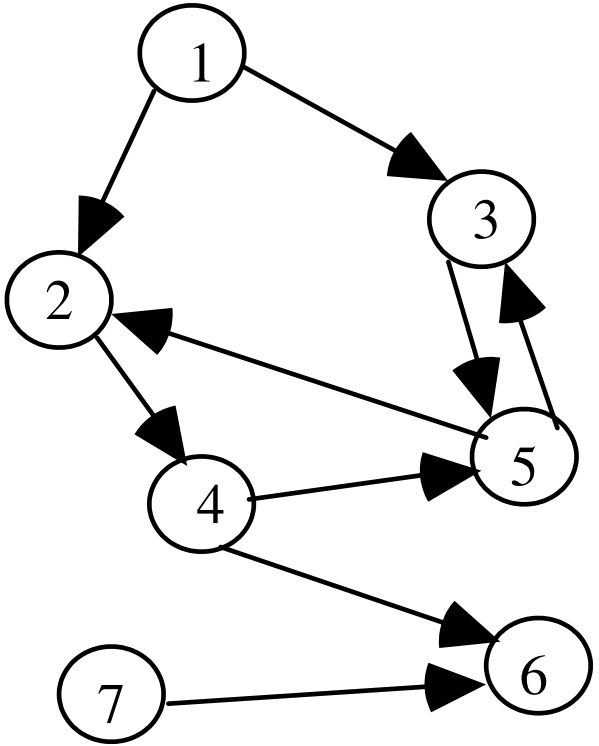
Nodo 4 e 6 sono adiacenti

Il Grafo e' connesso

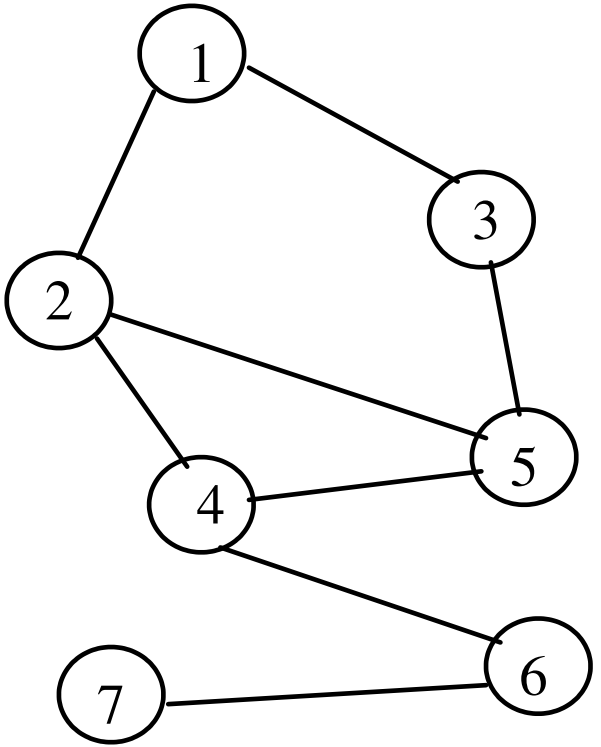



orientato

Esiste un cammino
dal nodo 1 al nodo 6




*a) Grafo orientato*




*b) Grafo non orientato*

# Applicazioni

- Rappresentazione dei collegamenti trasporti (aerei, bus, …)
- Algoritmi di instradamento
- Il web (connessioni e raggiungibilità)
- Flussi e portate (reti idriche…)
- Interazione tra proteine
- Correlazioni tra pazienti (esempio prossimità e trasmissibilità)
- Correlazione e processi tra reparti ospedalieri

# Definizione

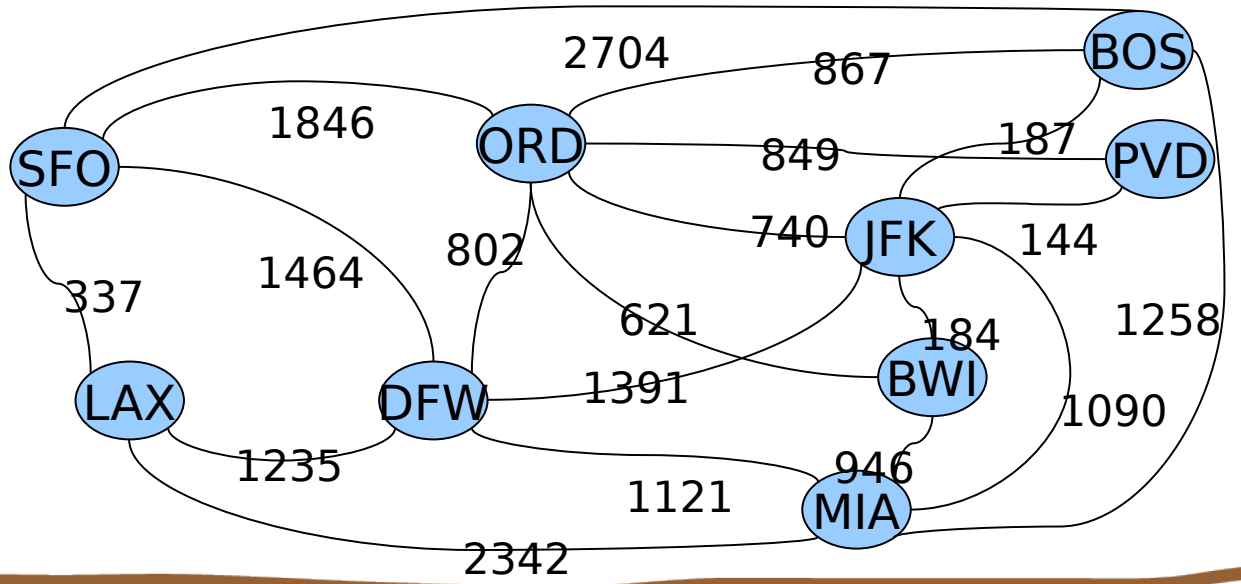
- Un Grafo pesato è una tripla (V, E, P) tale che G (v,E) e' un grafo e P e' una funzione che associa ad ogni arco (i,j) un peso p
- Un grafo si dice connesso se esiste un cammino che collega ogni coppia di nodi
- Grafo fortemente connesso se comunque presi due nodi esiste un cammino tra i due nodi
- Grafo debolmente connesso se il grafo ottenuto ignorando l'orientamento degli archi è connesso

# Problemi ed algoritmi su grafi

- Visita dei grafi
  - BFV(Breadth-first visit), DFV (deep-first-visit)
- Cammino minimo (a partire da un nodo) – Single-source shortest-path:
  - Algoritmo di Djakstra
  - Bellman-ford algoritmo (anche per pesi negativi)
- Cammino minimo per ogni coppia di nodi
  - Floyd-warshall
- Minumum Spanning Tree
  - Algo Kruskal
- K colarabilita
- Chiusura transitiva
  - Algoritmo di floyd
- Ordinamenti topologici (esempio per scheduling: se non ha terminato un evento non puo partire il successivo)

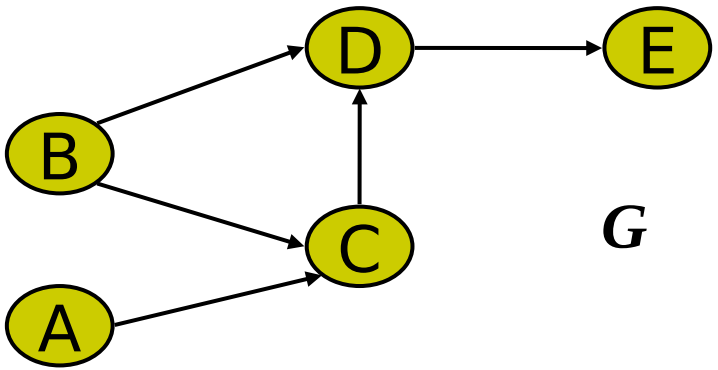
# Esempio: il calcolo del cammino minimo partendo da un nodo

- Grafo dei collegamenti aeroportuali
- Archi costi (es. miglia o euro)
- Calcolare il cammino a costo minimo tra JFK e SFO

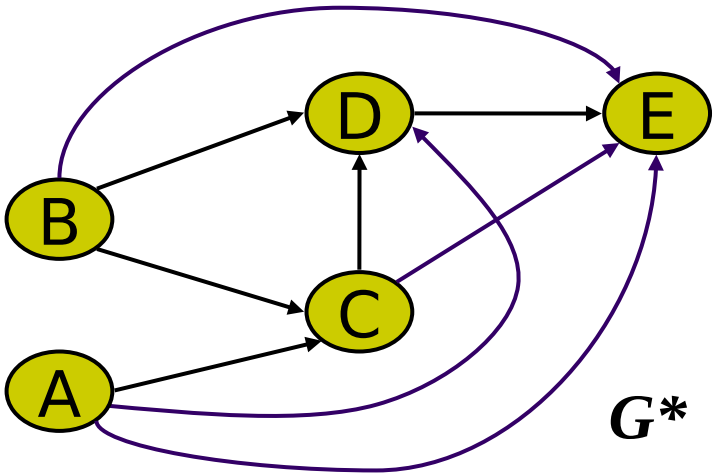


# Esempio: chiusura transitiva

- Dato un grafo diretto, aggiungere un cammino di collegamento diretto tra due nodi se esiste un cammino
- Es. Se (A,C) and (C,D) allora (A,D)

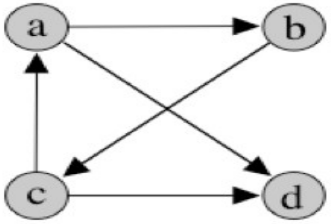



*G*

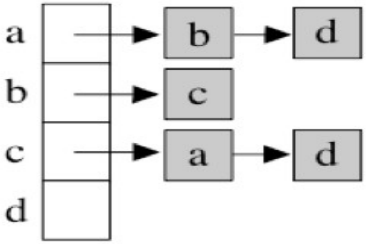



*G**

# Implementazione dei grafi




(a) Grafo orientato G




(c) Liste di adiacenza di G

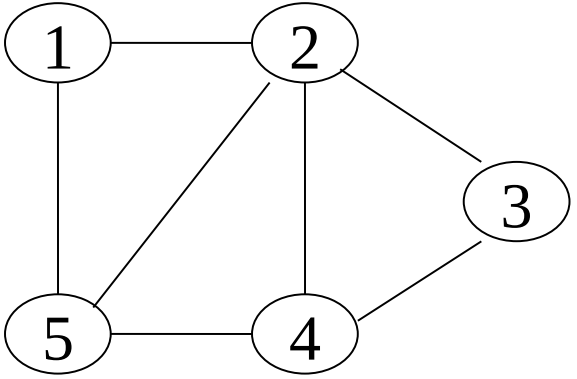
| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 |
| b | 0 | 0 | 1 | 0 |
| c | 1 | 0 | 0 | 1 |
| d | 0 | 0 | 0 | 0 |

(e) Matrice di adiacenza di G

# Strutture dati per rappresentazione dei Grafi in memoria

- Matrice di adiacenza: G(V,E) con |V| = n si usa una matrice nxn in cui l'elemento b(i,j) = 1 se (i,j) arco in E altrimenti b(i,j) = 0
- L'occupazine in memoria e' n2

Grafo non orientato

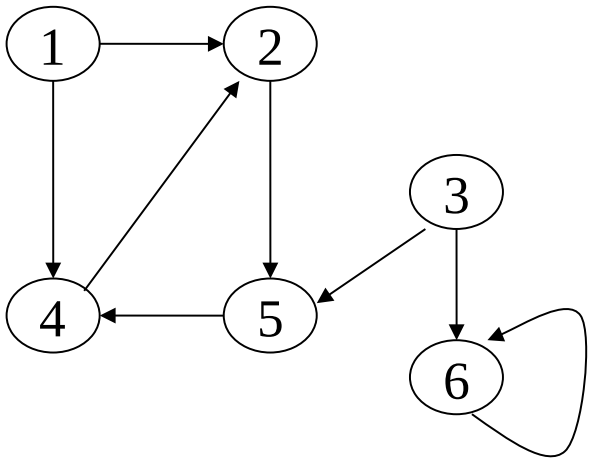



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

# Strutture dati per rappresentazione dei Grafi in memoria

- Matrice di adiacenza: G(V,E) con |V| = n si usa una matrice nxn in cui l'elemento b(i,j) = 1 se (i,j) arco in E altrimenti b(i,j) = 0
- L'occupazine in memoria e' n2

Grafo orientato

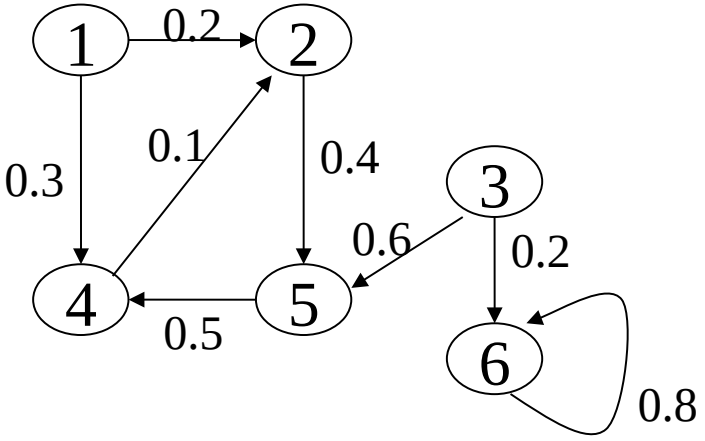



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

# Strutture dati per rappresentazione dei Grafi in memoria

- Matrice di adiacenza: G(V,E) con |V| = n si usa una matrice nxn in cui l'elemento b(i,j) = 1 se (i,j) arco in E altrimenti b(i,j) = 0
- L'occupazine in memoria e' n2

Grafo Pesato




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | .2 | 0 | .3 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | .4 | 0 |
| 3 | 0 | 0 | 0 | 0 | .6 | .2 |
| 4 | 0 | .1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | .5 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | .8 |

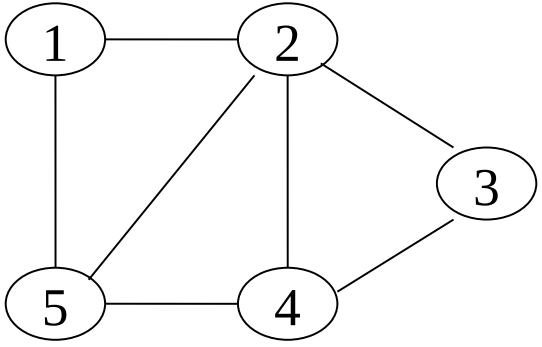
Nel caso di grafo pesato nella matrice si memorizzano i pesi in corrispondenza degli archi (se l'arco non esiste si indica con 0 o NIL)

# Proprietà della rappresentazione con matrice di adiacenza

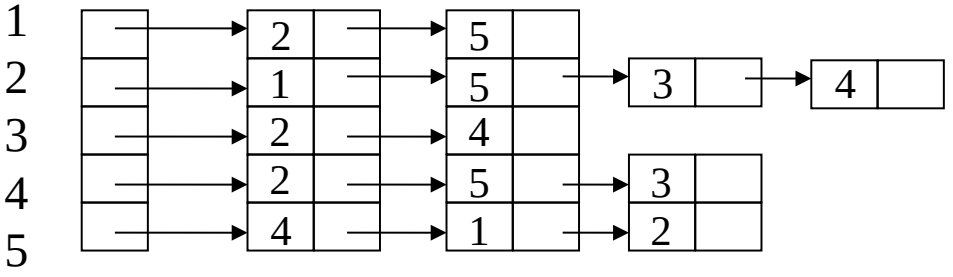
- La rappresentazione di un grafo G=(V,E) con matrice di adiacenza è quadratica nel numero di nodi (V2) indipendentemente dal numero di archi
- La matrice di adiacenza di un grafo non orientato è simmetrica ovvero aij= aji
- Per un grafo non orientato si può allora memorizzare solo i dati sopra la diagonale (diagonale inclusa), riducendo della metà lo spazio per memorizzare la matrice

# Liste di adiacenza

- Si rappresenta un grafo G=(V,E) con un vettore Adj di liste, una lista per ogni vertice del grafo
- per ogni vertice u,  Adj[u] contiene una lista (con ordine arbitrario) con tutti i vertici v adiacenti a u, ovvero quei vertici v tali per cui esiste un arco (u,v) E

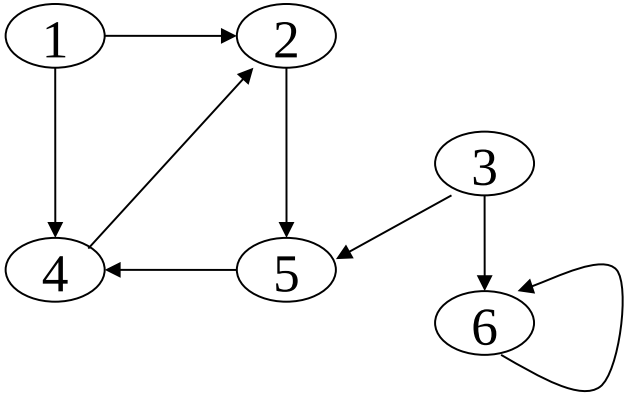



Grafo non orientato

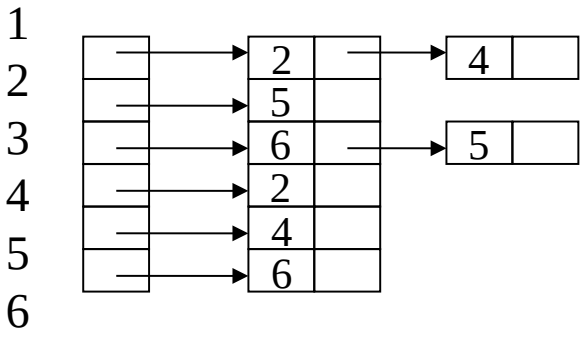


# Liste di adiacenza

- Si rappresenta un grafo G=(V,E) con un vettore Adj di liste, una lista per ogni vertice del grafo
- per ogni vertice u,  Adj[u] contiene una lista (con ordine arbitrario) con tutti i vertici v adiacenti a u, ovvero quei vertici v tali per cui esiste un arco (u,v) E

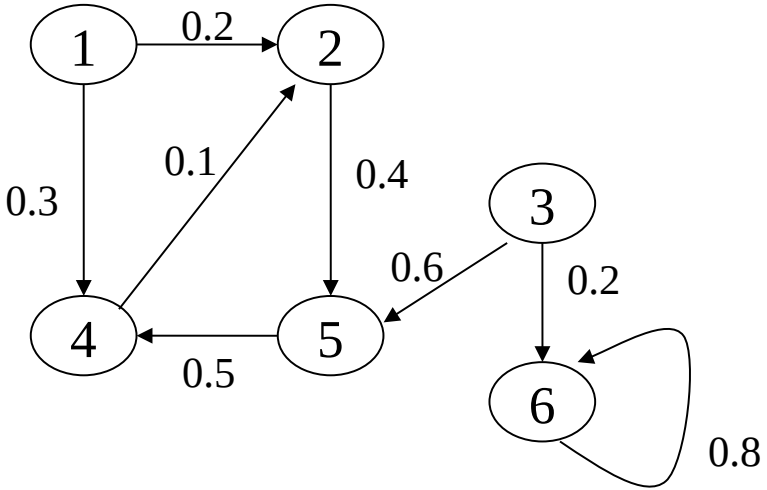


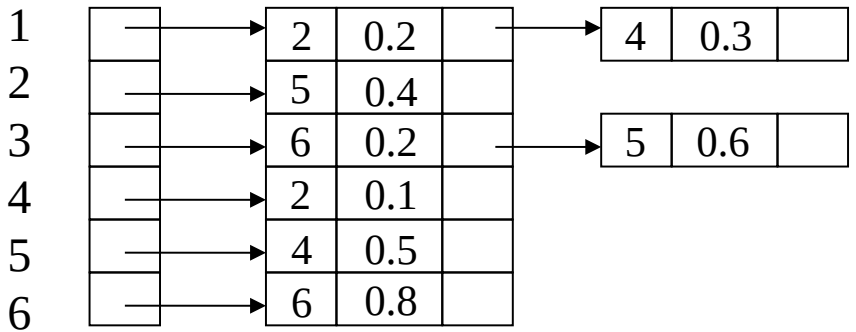

Grafo orientato



# Liste di adiacenza

- Si rappresenta un grafo G=(V,E) con un vettore Adj di liste, una lista per ogni vertice del grafo
- per ogni vertice u, Adj[u] contiene una lista (con ordine arbitrario) con tutti i vertici v adiacenti a u, ovvero quei vertici v tali per cui esiste un arco (u,v) E



# Proprietà della rappresentazione con liste di adiacenza

- Se un grafo è orientato allora la somma delle lunghezze di tutte le liste di adiacenza è |E|
- infatti per ogni arco (u,v) c'è un vertice v nella lista di posizione u
- Se un grafo non è orientato allora la somma delle lunghezze di tutte le liste di adiacenza è 2|E|
- infatti per ogni arco (u,v) c'è un vertice v nella lista di posizione u e un vertice u nella lista di posizione v
- La quantità di memoria necessaria per memorizzare un grafo (orientato o non) è O(max(V,E)) = O(V+E)

# Algoritmi di Visita

- Una visita di un grafo G permette di esaminare i nodi e gli archi di G in modo sistematico
- Problema di base in molte applicazioni
- Esistono vari tipi di visite con diverse proprietà: in particolare:
- visita in ampiezza (BFS=breadth first search) e
- visita in profondità (DFS=depth first search)

# Due esempi di interesse

- La visita in ampiezza: il caso dei contagi e l’analisi dei contatti in caso di cittadino/paziente positivo

- La visita in profondità: l’analisi diagnostica dall’analisi del sangue, all’eco...fino all estremo path

# Visita in profondità

- La visita in profondità (*depth-first-search DFS*) tra i vari nodi utili per il proseguimento della ricerca viene scelto quello visitato piu di recente
- Al generico passo appena visitato il nodo k si cerca un nodo collegato a k non ancora visitato
- Poiché un nodo può esser considerato più volte nel corso dell'algoritmo, per evitare che la scansione dei nodi ricominci si usa una struttura dai ausiliaria (marcatura dei nodi)
- (vers iterativa fa uso di PILA)

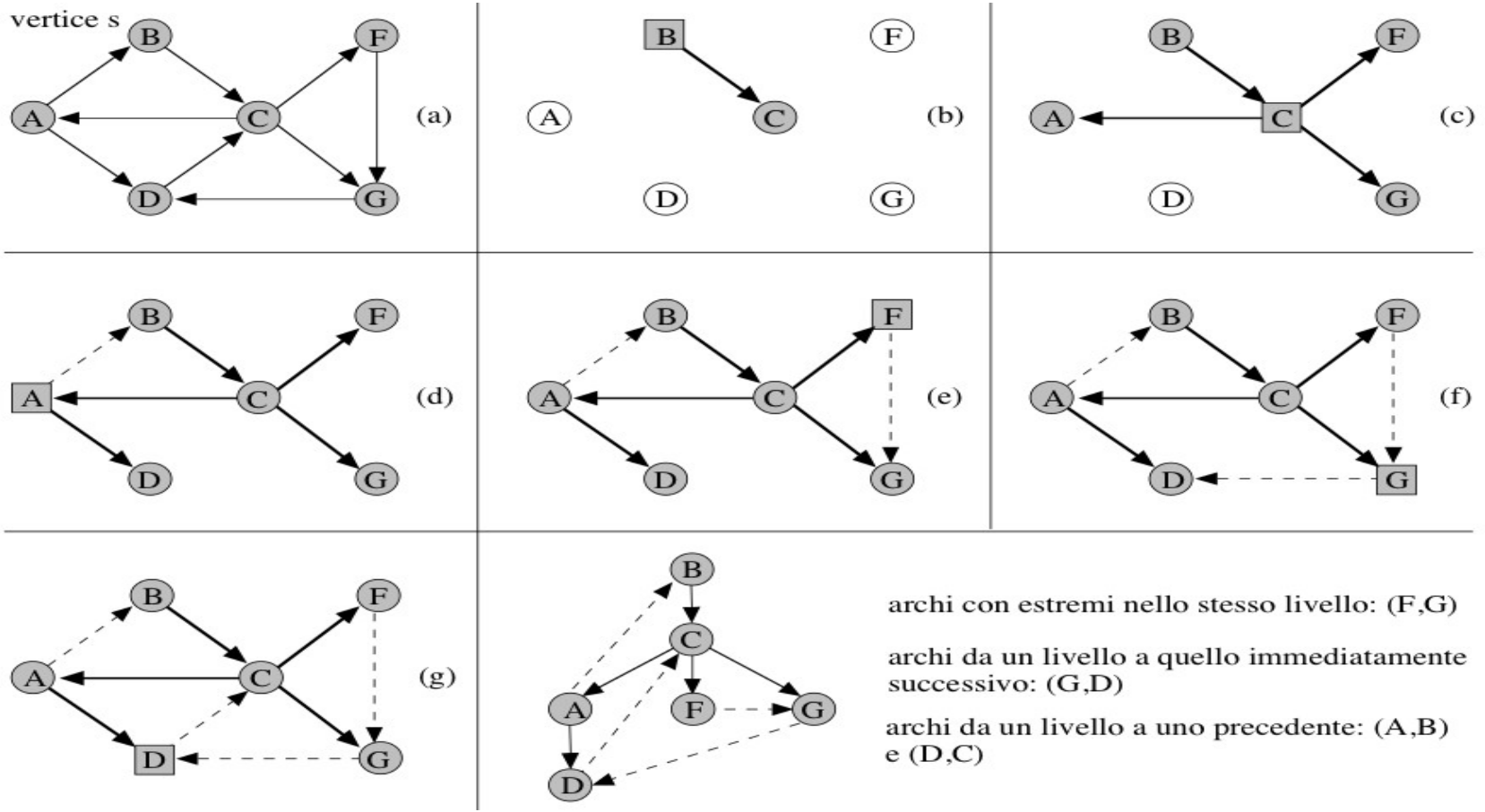
# Algoritmi di Visita

- Una visita di un grafo G permette di esaminare i nodi e gli archi di G in modo sistematico
- Problema di base in molte applicazioni
- Esistono vari tipi di visite con diverse proprietà: in particolare:
  - visita in ampiezza (BFS=breadth first search) e
  - visita in profondità (DFS=depth first search)

# Algoritmo di Visita in Ampiezza

- Un vertice viene marcato quando viene incontrato per la prima volta
- La visita genera un albero di copertura T del grafo
- L'insieme di vertici F T  mantiene la frangia di T:
  - v T-F: v è chiuso, tutti gli archi incidenti su v sono stati esaminati
  - v F: v è aperto,  esistono archi incidenti su v non ancora esaminati

# Algoritmo di Visita in Ampiezza



# Algoritmo di Visita in Ampiezza

```
algoritmo visitaBFS(vertice s) → albero
1.    rendi tutti i vertici non marcati
2.    T ← albero formato da un solo nodo s
3.    Coda F
4.    marca il vertice s
5.    F.enqueue(s)
6.    while ( not F.isempty() ) do
7.        u ← F.dequeue()
8.        for each ( arco (u, v) in G ) do
9.            if ( v non è ancora marcato ) then
10.               F.enqueue(v)
11.               marca il vertice v
12.               rendi u padre di v in T
13.   return T
```

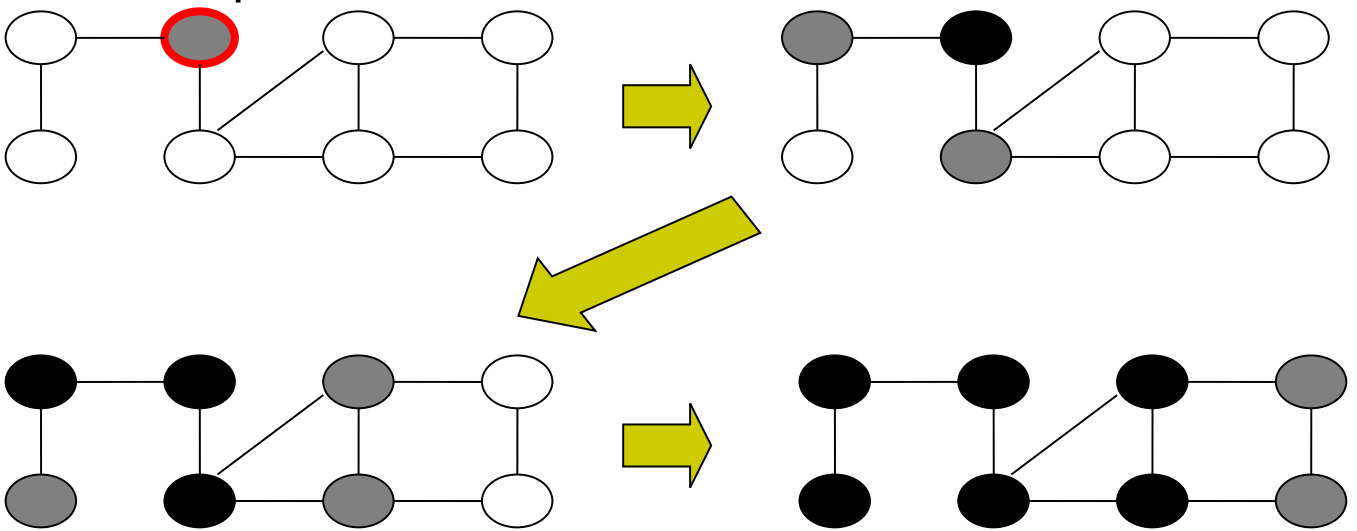
# Algoritmo di Visita in Ampiezza

Il tempo di esecuzione dipende dalla struttura dati usata per rappresentare il grafo:

- Liste di adiacenza: $O(m+n)$
- Matrice di adiacenza: $O(n^2)$

# Visita in ampiezza

- Dato un vertice sorgente s esplora tutti i vertici raggiungibili da s
- Tieni traccia (colora) dello stato di ogni vertice, :
  - bianco: vertice ancora non scoperto
  - grigio: vertice appena scoperto ed appartenente alla frontiera
  - nero: vertice per cui si è terminata la visita



# Algoritmo

- La procedura di visita in ampiezza assume che il grafo G=(V,E) sia rappresentato usando liste di adiacenza
- ad ogni vertice u sono associati inoltre l'attributo
  - colore: **`color[u]`**
  - padre: **`[u]`**
  - la distanza dalla sorgente s: **`d[u]`**
- L'algoritmo fa anche uso di una coda Q per gestire l'insieme dei vertici grigi (i.e. quelli utili per nuovi nodi)
- Coda Q: I nodi raggiunti sono aggiunti in testa, quando serve un prossimo grigio lo recupero dalla coda

# Pseudocodice

```
BFS(G,s)
1  for ogni vertice u  V[G]-{s}
2  do  color[u]  WHITE
3      d[u]
4      [u]  NIL
5  color[s]  GRAY
6  d[s]  0
7  [s]  NIL
8  Q  {s}
9  while Q
10 do  u  head[Q]
11     for ogni v  Adj[u]
12     do  if color[v]=WHITE
13         then  color[v]  GRAY
14              d[v]  d[u]+1
15              [v]  u
16              Enqueue(Q,v)
```

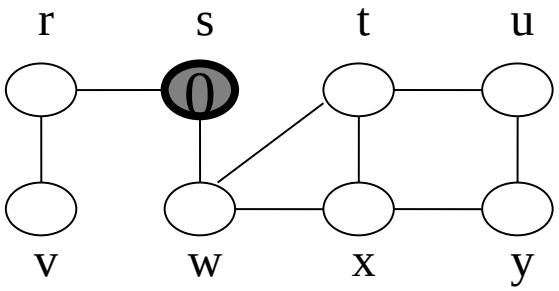
# Spiegazione del codice

- Le linee 1-4 eseguono l’inizializzazione:
  - tutti i vertici sono colorati di bianco
  - la distanza di tutti i vertici è non nota e posta a
  - il padre di ogni vertice inizializzato a nil
- la linea 5 inizializza la sorgente a cui:
  - viene assegnato il colore grigio
  - viene assegnata distanza 0
  - viene assegnato padre nullo nil
- la linea 8 inizializza la coda Q con il vertice sorgente s

# Spiegazione del codice

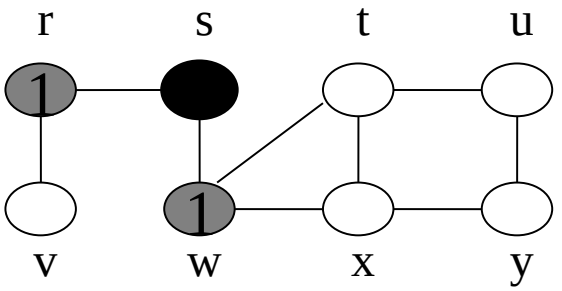
- Il ciclo principale è contenuto nelle linee 9-18
- il ciclo continua fino a quando vi sono vertici grigi in Q, ovvero vertici già scoperti le cui liste di adiacenza non siano state ancora completamente esaminate
- la linea 10 preleva l'elemento in testa alla coda
- nelle linee 11-16 si esaminano tutti i vertici v adiacenti a u
- se v non è ancora stato scoperto lo si scopre
  - si colora di grigio
  - si aggiorna la sua distanza alla distanza di u +1
  - si memorizza u come suo predecesore
  - si pone in fondo alla coda
- quando tutti i vertici adiacenti a u sono stati scoperti allora si colora u di nero e lo si rimuove da Q

# Visualizzazione

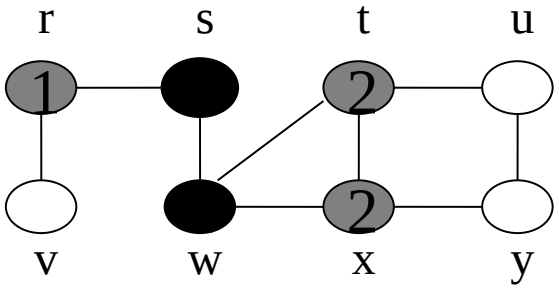



Q:s



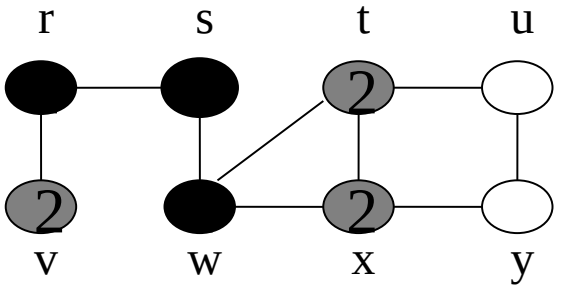



Q:wr



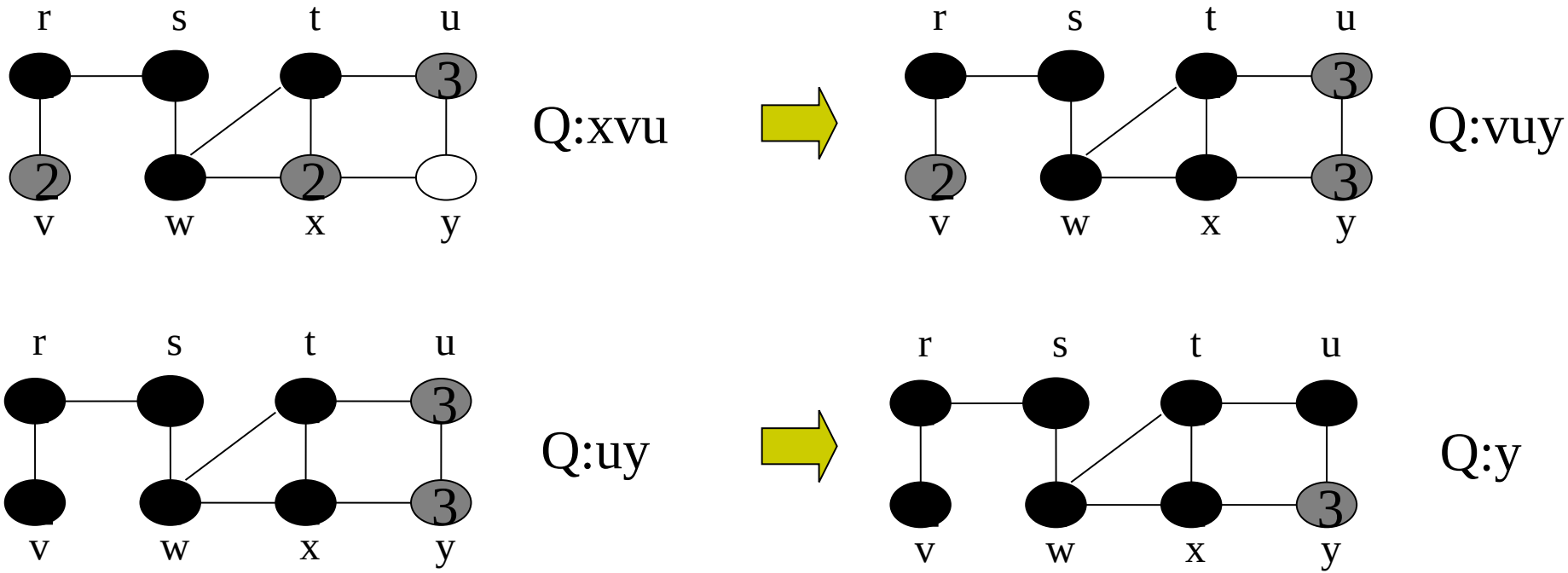

Q:rtx





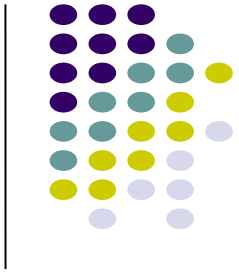

Q:txv

# Visualizzazione




**Costo: la procedura di visita in ampiezza richiede un tempo lineare nella rappresentazione con liste di adiacenza O(V+E)**

# Analisi del costo

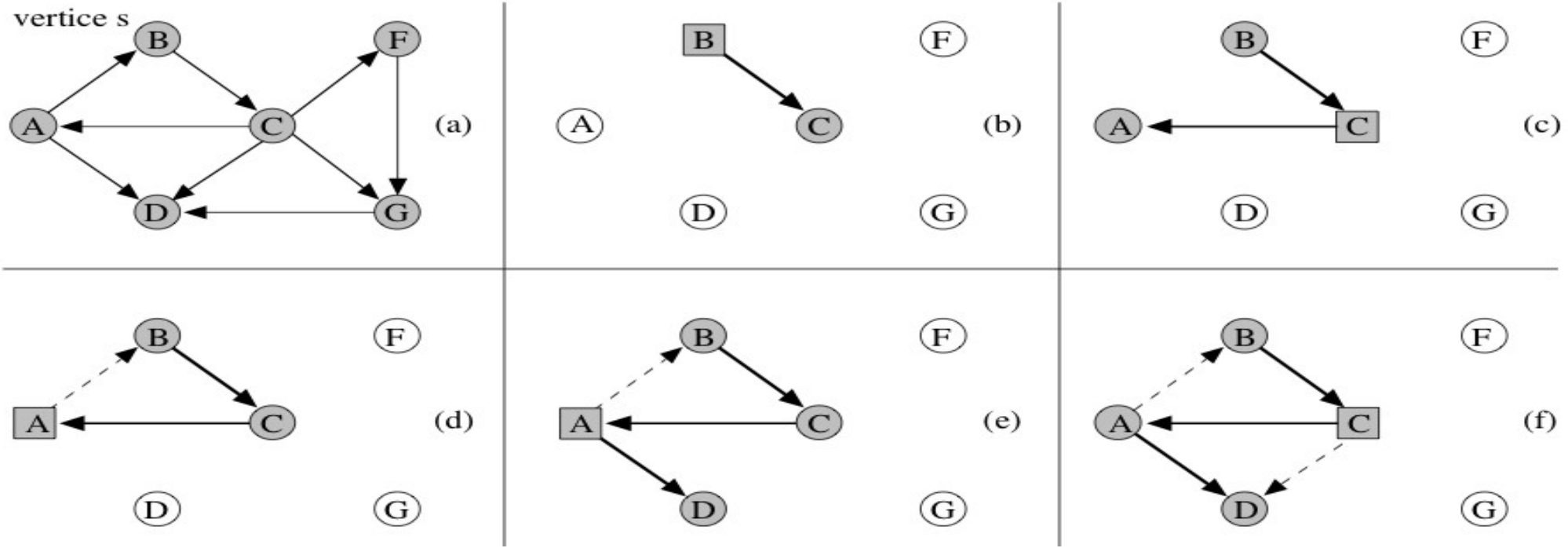


- Il tempo per l'inizializzazione è O(V)
- Dopo l'inizializzazione nessun vertice sarà mai colorato più di bianco
- quindi il test in 12 assicura che ogni vertice sarà inserito nella coda Q al più una volta
- le operazioni di inserimento ed eliminazione dalla coda richiedono un tempo O(1)
- il tempo dedicato alla coda nel ciclo 9-18 sarà pertanto un O(V)

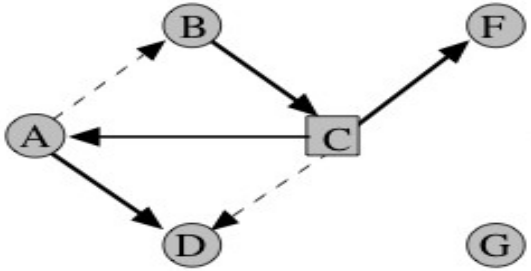
# Analisi del costo

- poiché la lista di adiacenza è scandita solo quando si estrae il vertice dalla coda allora la si scandisce solo 1 volta per vertice
- poiché il numero di archi è pari a |E| allora la somma delle lunghezze di tutte le liste è $\Theta(E)$
- allora il tempo speso per la scansione delle liste complessivamente è O(E)
- in totale si ha un tempo di O(V+E)
- la procedura di visita in ampiezza richiede un tempo lineare nella rappresentazione con liste di adiacenza O(V+E)

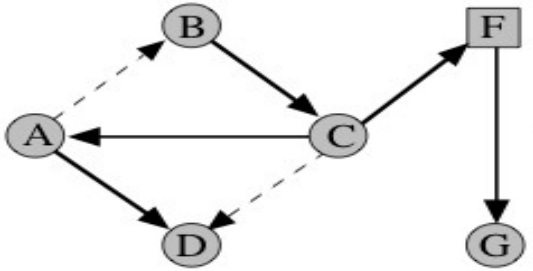
# Algoritmo di Visita in Profondità



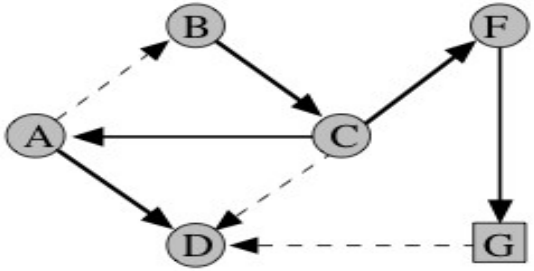
# Algoritmo di Visita in Profondità



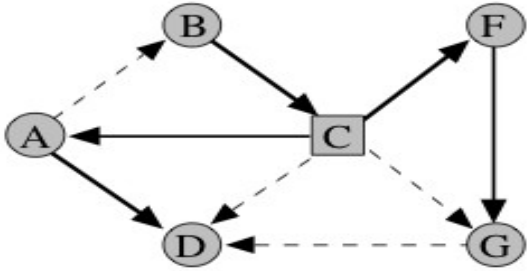

(g)



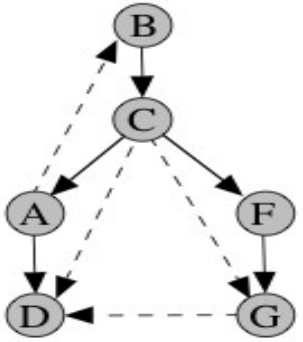

(h)




(i)




(j)




archi in avanti: (C,D) e (C,G)

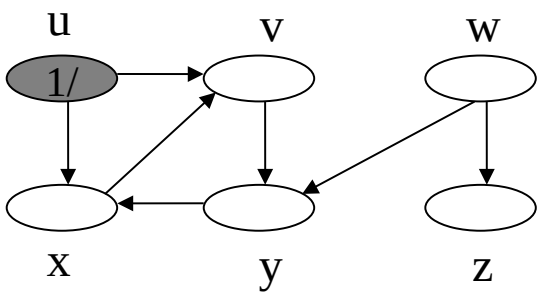
archi all'indietro: (A,B)

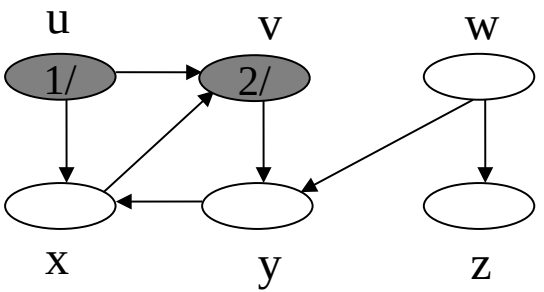
archi trasversali a sinistra: (G,D)

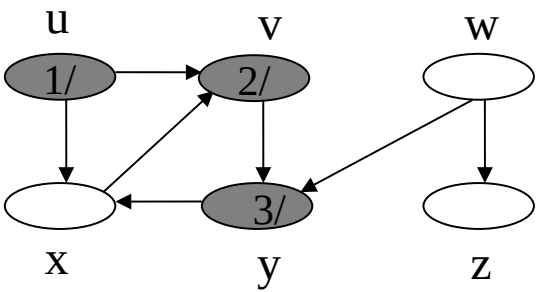
# Algoritmo di Visita in Profondità

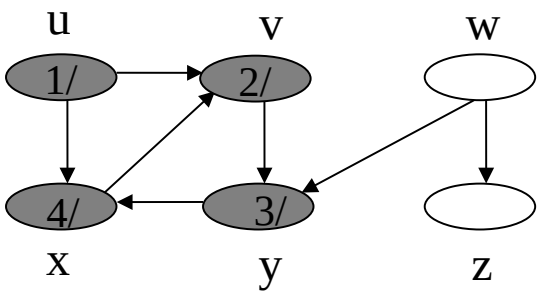
```
    procedura visitaDFSRicorsiva(vertice v, albero T)
1.      marca e visita il vertice v
2.      for each ( arco (v, w) ) do
3.          if ( w non è marcato ) then
4.              aggiungi l'arco (v, w) all'albero T
5.              visitaDFSRicorsiva(w, T)

    algoritmo visitaDFS(vertice s) → albero
6.      T ← albero vuoto
7.      visitaDFSRicorsiva(s, T)
8.      return T
```

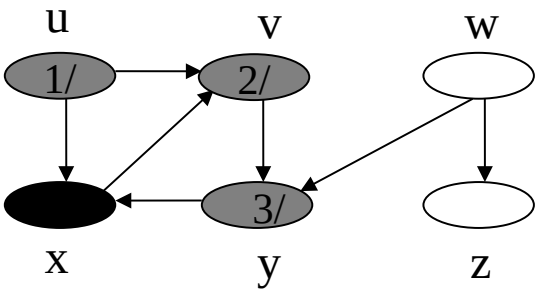
# Visita in profondità

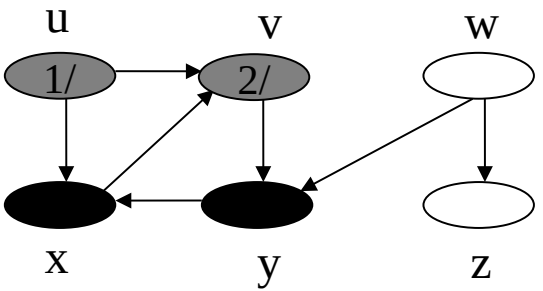


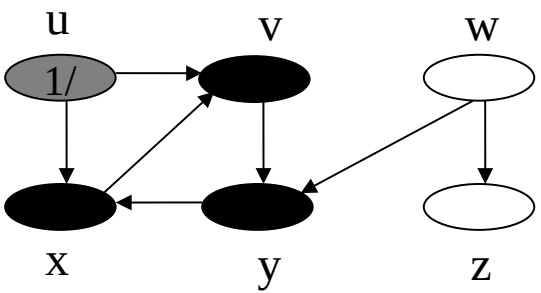



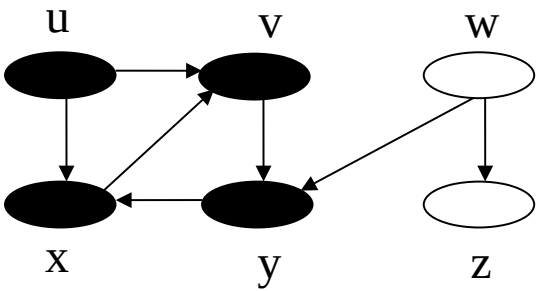



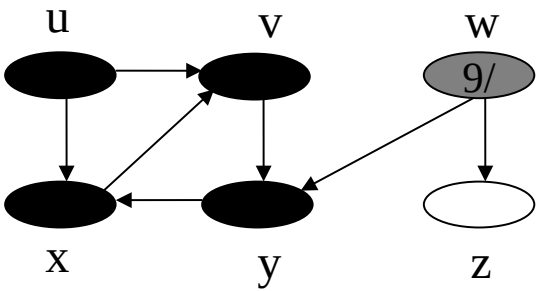



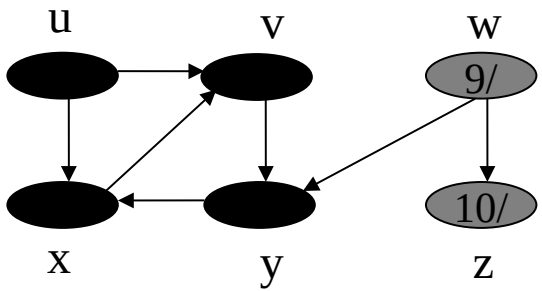



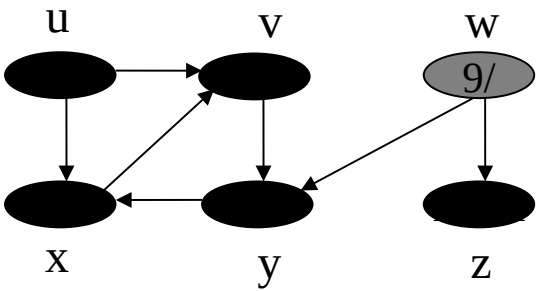
# Visita in profondità

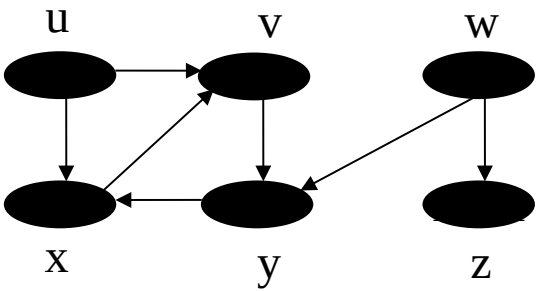


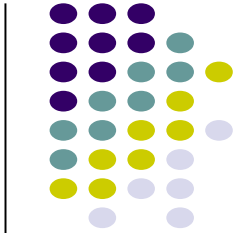



# Strutture dati



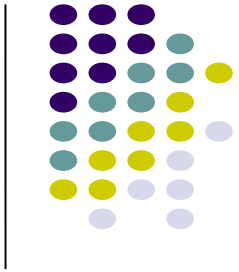
- Oltre al colore si associa ad ogni vertice v due informazioni temporali:
  - tempo di inizio visita d[v], cioè quando è reso grigio per la prima volta
  - tempo di fine visita f[v], cioè quando è reso nero
- il valore temporale è dato dall'ordine assoluto con cui si colorano i vari vertici del grafo
- si usa per questo una variabile globale *tempo* che viene incrementata di uno ogni volta che si esegue un inizio di visita o una fine visita

# Marcatura temporale

- il tempo è un intero compreso fra 1 e 2|V| poiché ogni vertice può essere scoperto una sola volta e la sua visita può finire una sola volta

- per ogni vertice u si ha sempre che d[u]<f[u]
- ogni vertice u è
  - WHITE prima di d[u]
  - GRAY fra d[u] e f[u]
  - BLACK dopo f[u]

# Pseudocodice



```
DFS(G)
1  for ogni vertice u   V[G]
2  do  color[u]   WHITE
3      [u]   NIL
4  time   0
5  for ogni vertice u   V[G]
6      do  if color[u]=WHITE
7          then   DFS-Visit(u)

DFS-Visit(u)
1  color[u]   GRAY
2  d[u]   time   time +1
3  for ogni v   Adj[u]
4  do  if color[v]=WHITE
5      then    [v]   u
6          DFS-Visit(v)
```

# Spiegazione dello pseudocodice

- Le righe 1-4 della procedura DFS eseguono la fase di inizializzazione colorando ogni vertice del grafo di bianco, settando il padre a NIL e impostandola variabile globale time a 0
- il ciclo 5-7 esegue la procedura DFS-Visit su ogni nodo non ancora scoperto del grafo, creando un albero DFS ogni volta che viene invocata la procedura
- In ogni chiamata DFS-Visit(u) il vertice u è inizialmente bianco
- viene reso grigio e viene marcato il suo tempo di inizio visita in d[u], dopo aver incrementato il contatore temporale globale time
- vengono poi esaminati tutti gli archi uscenti da u e viene invocata ricorsivamente la procedura nel caso in cui i vertici collegati non siano ancora stati esplorati
- in questo caso il loro padre viene inizializzato ad u
- dopo aver visitato tutti gli archi uscenti u viene colorato BLACK e viene registrato il tempo di fine visita in f[u]

# Analisi del tempo di calcolo

- Il ciclo di inizializzazione di DFS e il ciclo for 5-7 richiedono entrambi tempo $\Theta(V)$
- la procedura DFS-Visit viene chiamata solo una volta per ogni vertice (poiché viene chiamata quando il vertice è bianco e lo colora immediatamente di grigio)
- in DFS-Visit il ciclo for 3-6 viene eseguito |Adj[v]| volte, e dato che la somma della lunghezza di tutte le liste di adiacenza è $\Theta(E)$, il costo è $\Theta(E)$
- il tempo totale è quindi un $\Theta(V+E)$

# Visita in ampiezza (iterativa) - BFS

- La visita in ampiezza fa uso di una coda per memorizzare tutti gli archi incidenti nel nodo v visitato che portano ad un nodo marcato 0.
- I nodi raggiungibili non marcati vengono quindi marcati.
- La visita procede dall’arco (v,u) in testa alla coda.

# Implementazione della BFS

```
breadthFirstSearch() {
    for (tutti i vertici v)
        num(v) = 0;
    edges = {}; // empty set
    i = 1;
    while (<esiste vertice v tale che num(v) == 0>) {
        num(v) = i++;
        enqueue(v);
        while (<la coda non è vuota>) {
            v = dequeue();
            for (<tutti i vertici u adiacenti a v>)
                if (num(u) == 0) {
                    num(u) = i++;
                    enqueue(u);
                    edges = edges   {(v, u)}
                }
        }
    }
    <visualizza edges>
}
```