Fondamenti di Informatica (Elettronici)

n-alberi e grafi – Lezione 39

20 gennaio 2021

n-alberi e grafi

Section 1

Alberi binari di ricerca

Algoritmo di ricerca in albero binario ordinato

Si assume che i nodi possano essere ordinati,
ad esempio: cognomi ordinati
alfabeticamente, oppure codici fiscali.

Algoritmo di ricerca in albero binario ordinato

Si assume che i nodi possano essere ordinati, ad esempio: cognomi ordinati alfabeticamente, oppure codici fiscali.

Un albero binario di ricerca assume che:

- tutti i nodi del sottoalbero sinistro abbiano valore minore o eguale alla radice

In tal caso si puo' trovare un elemento (se esiste) oppure verificare la sua non esistenza con un numero di confronti non superiore al piu' lungo cammino sull'albero.

Algoritmo di ricerca in albero binario ordinato

Si assume che i nodi possano essere ordinati, ad esempio: cognomi ordinati alfabeticamente, oppure codici fiscali.

Un albero binario di ricerca assume che:

- tutti i nodi del sottoalbero sinistro abbiano valore minore o eguale alla radice
- tutti i nodi del sottoalbero destro abbiano valore maggiore

In tal caso si puo' trovare un elemento (se esiste) oppure verificare la sua non esistenza con un numero di confronti non superiore al piu' lungo cammino sull'albero.

Algoritmo di ricerca in albero binario ordinato

Si assume che i nodi possano essere ordinati, ad esempio: cognomi ordinati alfabeticamente, oppure codici fiscali.

Un albero binario di ricerca assume che:

- tutti i nodi del sottoalbero sinistro abbiano valore minore o eguale alla radice
- tutti i nodi del sottoalbero destro abbiano valore maggiore

In tal caso si puo' trovare un elemento (se esiste) oppure verificare la sua non esistenza con un numero di confronti non superiore al piu' lungo cammino sull'albero.

Algoritmo di ricerca in albero binario ordinato

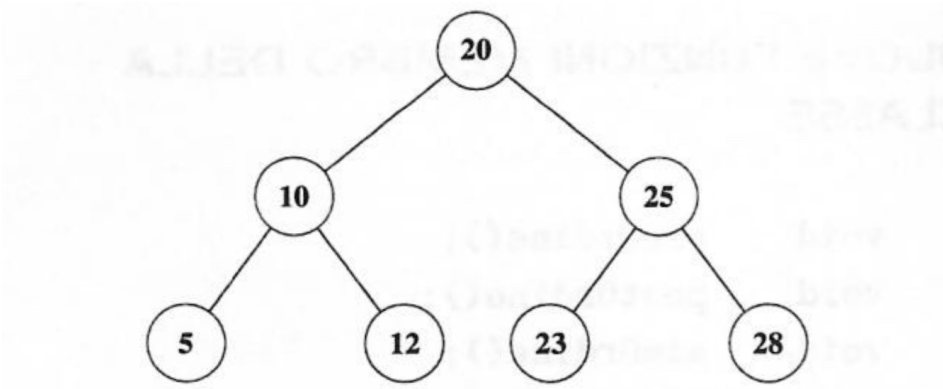
Si assume che i nodi possano essere ordinati, ad esempio: cognomi ordinati alfabeticamente, oppure codici fiscali.

Un albero binario di ricerca assume che:

- tutti i nodi del sottoalbero sinistro abbiano valore minore o eguale alla radice
- tutti i nodi del sottoalbero destro abbiano valore maggiore

In tal caso si puo' trovare un elemento (se esiste) oppure verificare la sua non esistenza con un numero di confronti non superiore al piu' lungo cammino sull'albero.

Se l'albero e' bilanciato, e contiene n nodi, allora tali cammini hanno complessità $O(\log n)$




```
algoritmo cerca (x, T)
    if vuoto(T)
        stampa 'x non in T'
    else
        if x = radice(T)
            ind(x)
        else
            if x < radice(T)
                cerca (x, sinistro(T)
            else
                cerca (x, destro(T)
```

Alberi binari di ricerca

FILE A INDICI

Gli alberi binari di ricerca sono usati come indici su grandi archivi ordinati

Per indicare tali tipi di archivi si parla anche di file a indici

In un archivio di questo tipo la ricerca di un elemento si esegue in un tempo logaritmico con il numero di elementi (es: 20 accessi per trovare un elemento in un archivio di un milione di elementi)

La ricerca resta efficiente se l'albero di ricerca resta bilanciato.

L'inserzione/cancellazione di elementi deve mantenere bilanciato l'albero, eventualmente "ruotando" dei sottoalberi

Section 2

Alberi n-ari

Albero *n*-ario (Tree)

DEFINIZIONE

Tipo astratto <S, F, C>, dove

```
S = < Tree, nodo, boolean >,
F = < root, parent, firstChild, rightSibling, attach, build, null >,
C = < albero_vuoto >
```

con Tree Albero ordinato detto dominio di interesse, e dove

```
root : Tree -+ nodo;
parent : nodo x Tree -+ nodo;
firstChild : nodo x Tree -+ nodo;
rightSibling : nodo x Tree -+ nodo;
attach : nodo x Tree -+ Tree;
build : Tree x Tree -4 Tree;
null : Tree -+boolean;
```

Descrizione (semantica) delle operazioni

root definita sugli alberi non vuoti; restituisce la radice;

Descrizione (semantica) delle operazioni

root definita sugli alberi non vuoti; restituisce la radice;

parent restituisce il padre del nodo n, se presente in T; se $n \notin T$ oppure n è radice, restituisce il simbolo speciale $\top$

Descrizione (semantica) delle operazioni

root definita sugli alberi non vuoti; restituisce la radice;

parent restituisce il padre del nodo n, se presente in T; se $n \notin T$ oppure n è radice, restituisce il simbolo speciale $\top$

firstChild restituisce il primogenito del nodo n; se $n \notin T$ oppure n è foglia, restituisce il simbolo speciale $\top$

Descrizione (semantica) delle operazioni

root	definita sugli alberi non vuoti; restituisce la radice;
parent	restituisce il padre del nodo n, se presente in T; se $n \notin T$ oppure n è radice, restituisce il simbolo speciale $\top$
firstChild	restituisce il primogenito del nodo n; se $n \notin T$ oppure n è foglia, restituisce il simbolo speciale $\top$
rightSibling	restituisce il prossimo fratello del nodo n; se $n \notin T$ oppure n è ultimo figlio, restituisce il simbolo speciale $\top$

Descrizione (semantica) delle operazioni

root	definita sugli alberi non vuoti; restituisce la radice;
parent	restituisce il padre del nodo n, se presente in T; se $n \notin T$ oppure n è radice, restituisce il simbolo speciale $\top$
firstChild	restituisce il primogenito del nodo n; se $n \notin T$ oppure n è foglia, restituisce il simbolo speciale $\top$
rightSibling	restituisce il prossimo fratello del nodo n; se $n \notin T$ oppure n è ultimo figlio, restituisce il simbolo speciale $\top$
attach	attacca il nodo n come radice di T

Descrizione (semantica) delle operazioni

root definita sugli alberi non vuoti; restituisce la radice;

parent restituisce il padre del nodo n, se presente in T; se $n \notin T$ oppure n è radice, restituisce il simbolo speciale $\top$

firstChild restituisce il primogenito del nodo n; se $n \notin T$ oppure n è foglia, restituisce il simbolo speciale $\top$

rightSibling restituisce il prossimo fratello del nodo n; se $n \notin T$ oppure n è ultimo figlio, restituisce il simbolo speciale $\top$

attach attacca il nodo n come radice di T

build assume in ingresso due alberi; restituisce l'albero costruito attaccando la radice del secondo argomento come ultimo figlio della radice del primo argomento (gli interi alberi restano attaccati)

Descrizione (semantica) delle operazioni

root definita sugli alberi non vuoti; restituisce la radice;

parent restituisce il padre del nodo n, se presente in T; se $n \notin T$ oppure n è radice, restituisce il simbolo speciale $\top$

firstChild restituisce il primogenito del nodo n; se $n \notin T$ oppure n è foglia, restituisce il simbolo speciale $\top$

rightSibling restituisce il prossimo fratello del nodo n; se $n \notin T$ oppure n è ultimo figlio, restituisce il simbolo speciale $\top$

attach attacca il nodo n come radice di T

build assume in ingresso due alberi; restituisce l'albero costruito attaccando la radice del secondo argomento come ultimo figlio della radice del primo argomento (gli interi alberi restano attaccati)

null prende in ingresso un valore $T \in \mathit{Tree}$; restituisce `true` se T è l'albero vuoto; `false` altrimenti.

bbbbbb

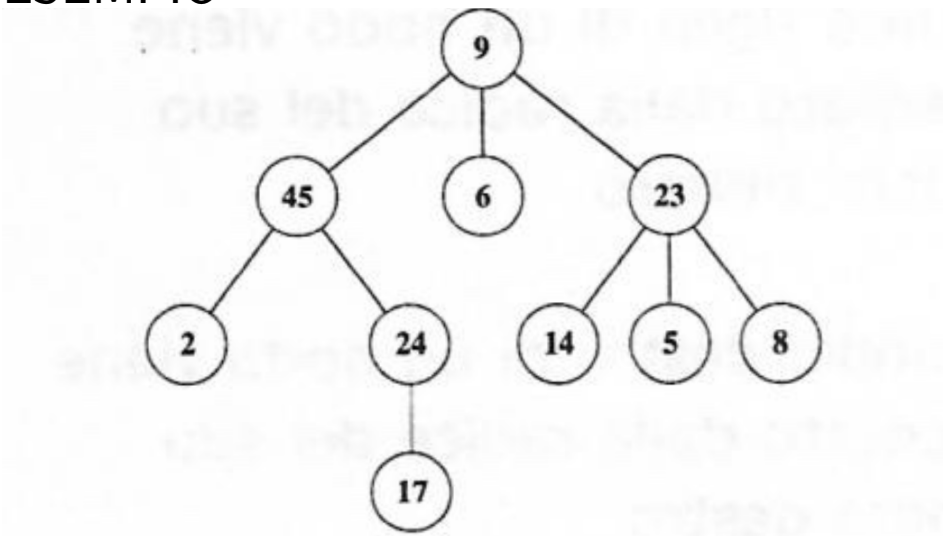
DEFINIZIONE (induttiva)

Un albero T e' un grafo in cui

- Esiste un nodo r con n nodi successori $(a_1, \ldots, a_n)$; r e' detto radice .

Grado di un nodo e' il numero dei suoi sottoalberi immediati
Le foglie hanno grado 0

ESEMPIO



Rappresentazione grafica di un albero n-ario

bbbbbb

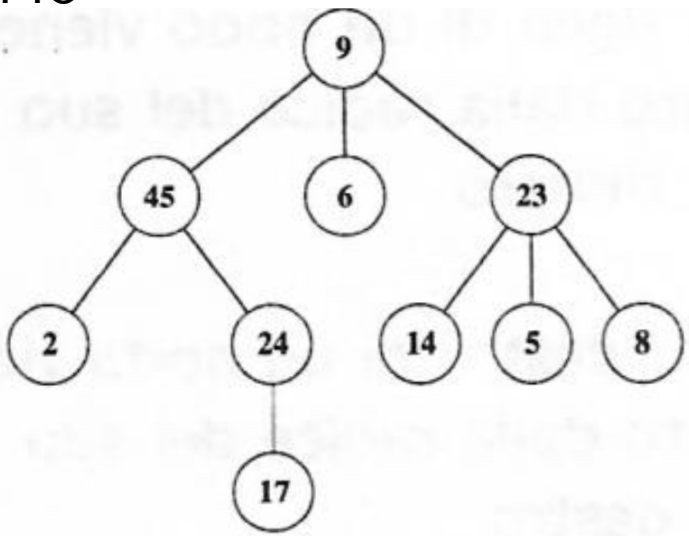
DEFINIZIONE (induttiva)

Un albero T e' un grafo in cui

- Esiste un nodo r con n nodi successori $(a_1, \ldots, a_n)$; r e' detto radice .
- Tutti i nodi di T, tranne r, possono essere ripartiti in n insiemi disgiunti, ciascuno dei quali individua in T un albero $T_1, \ldots, T_n$ con radice a_i, rispettivamente.

Grado di un nodo e' il numero dei suoi sottoalberi immediati
Le foglie hanno grado 0

ESEMPIO



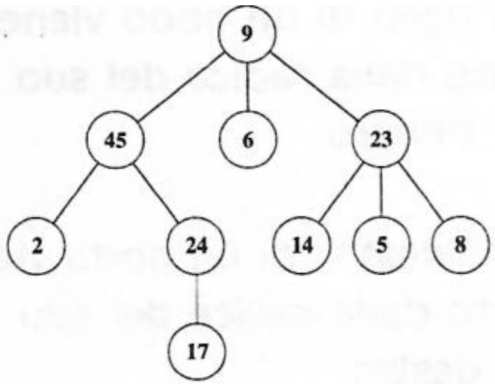
Rappresentazione grafica di un albero n-ario

Algoritmi fondamentali

Visita in profondità
(Depth First Search - DFS) in Preordine

```
Algoritmo Preordine(T)

if T è non vuoto:
    elabora root(T);
    for each sottoalbero Ti do
        Preordine(Ti)
```



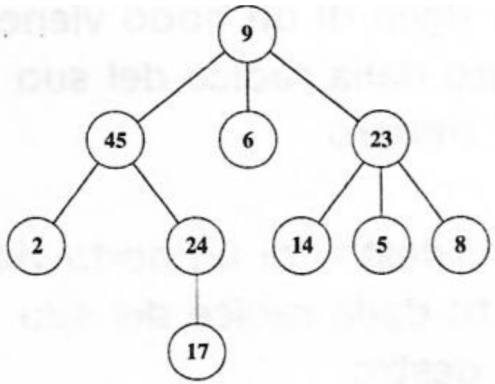
9 45 2 24 17 6 23 14 5 8

Algoritmi fondamentali

Visita in profondità
(Depth First Search - DFS) in Preordine

```
Algoritmo Preordine(T)

if T è non vuoto:
    elabora root(T);
    for each sottoalbero Ti do
        Preordine(Ti)
```

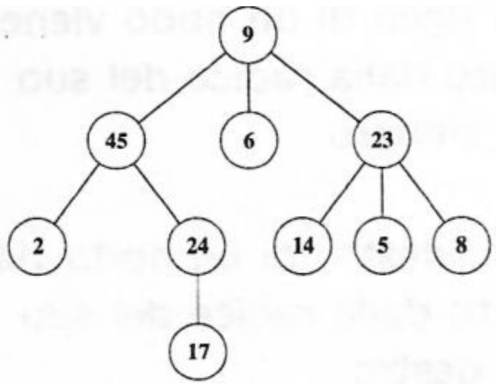


9 45 2 24 17 6 23 14 5 8

Visita in profondità
(Depth First Search - DFS) in Postordine

```
Algoritmo Postordine(T)

if T è non vuoto:
    for each sottoalbero Ti do
        Postordine(Ti)
    elabora root(T)
```



2 17 24 45 6 14 5 8 23 9

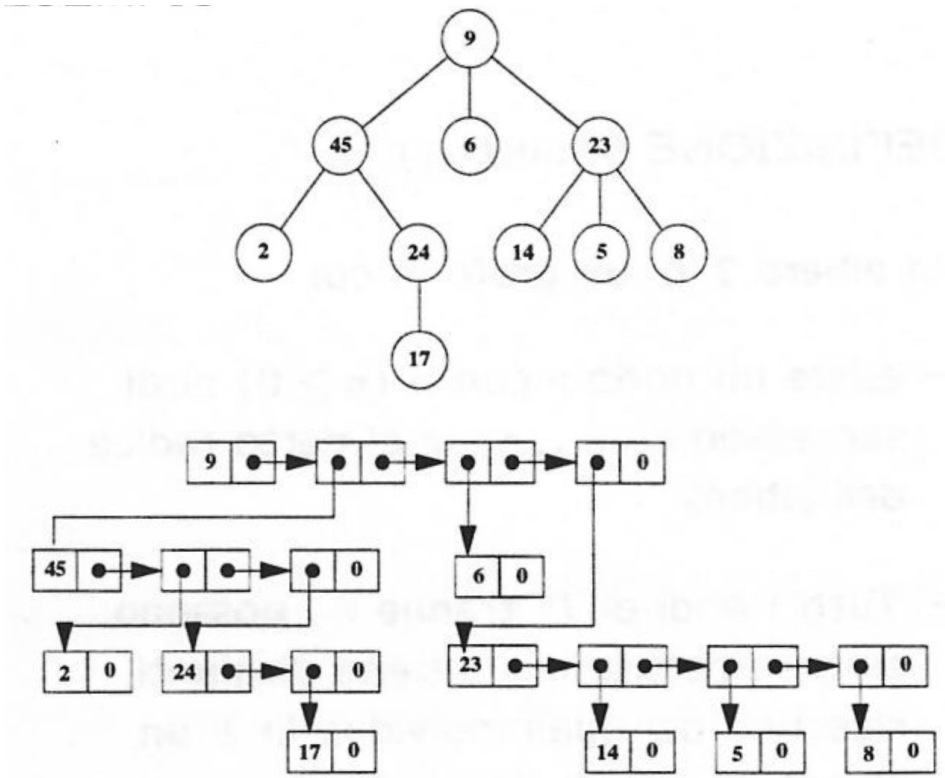
Rappresentazione con liste multiple

LISTA MULTIPLA = Lista i cui elementi sono atomi oppure liste

DEFINIZIONE

Un albero n-ario puo' essere rappresentato con liste multiple seguendo le regole seguenti:

- un albero vuoto e' rappresentato dalla lista vuota



OGNI NODO E' rappresentato da un lista di $k+1$ nodi, se k sono gli archi uscenti

MEMORIA: $Space(|N| + |A|) = O(n)$

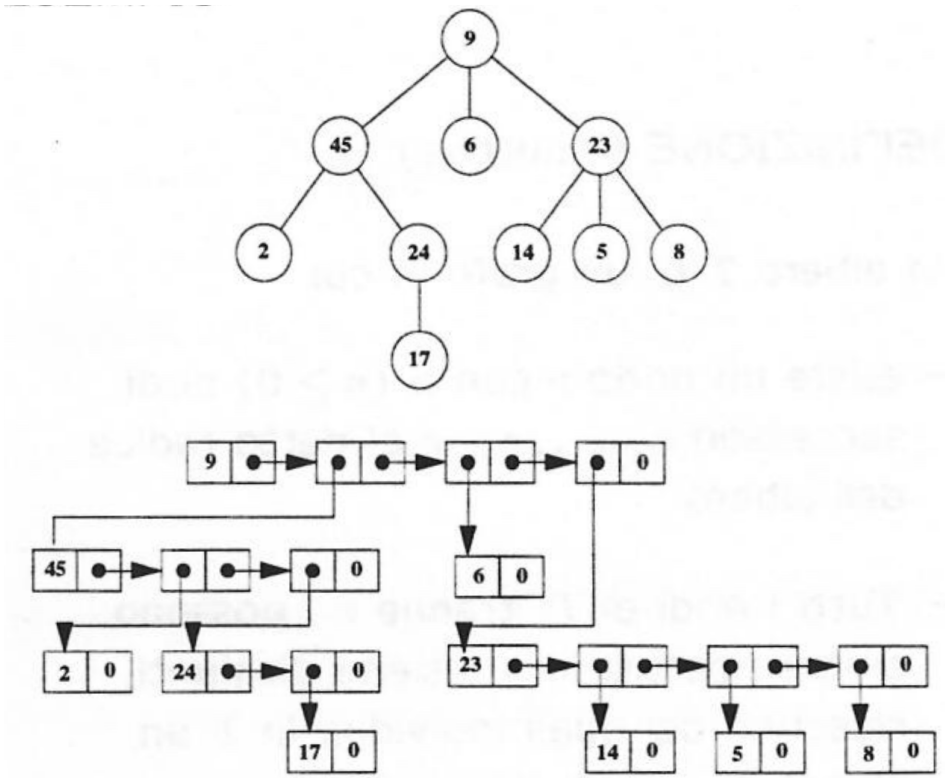
Rappresentazione con liste multiple

LISTA MULTIPLA = Lista i cui elementi sono atomi oppure liste

DEFINIZIONE

Un albero n-ario puo' essere rappresentato con liste multiple seguendo le regole seguenti:

- un albero vuoto e' rappresentato dalla lista vuota
- se l'albero e' formato da una radice e k sottoalberi e' rappresentato da una lista di $k+1$ elementi; il primo e' un atomo che rappresenta la radice; gli altri sono liste che rappresentano ordinatamente i sottoalberi



OGNI NODO E' rappresentato da un lista di $k+1$ nodi, se k sono gli archi uscenti

MEMORIA: $Space(|N| + |A|) = O(n)$

Rappresentazione con alberi binari

RAPPRESENTAZIONE CON ALBERI BINARI

Ogni valore di tipo albero n-ario e' rappresentato da un valore di tipo albero binario, seguendo le seguenti

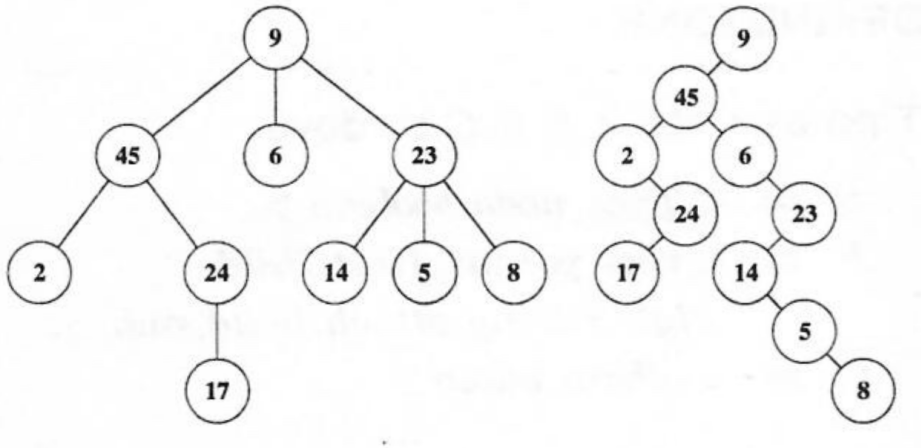
REGOLE:

(a) il primo figlio di un nodo viene rappresentato dalla radice del suo sottoalbero sinistro

Rappresentazione di un albero n-ario con un albero binario

MEMORIA: O(n)

Più efficiente che con liste !!



Rappresentazione con alberi binari

RAPPRESENTAZIONE CON ALBERI BINARI

Ogni valore di tipo albero n-ario e' rappresentato da un valore di tipo albero binario, seguendo le seguenti

REGOLE:

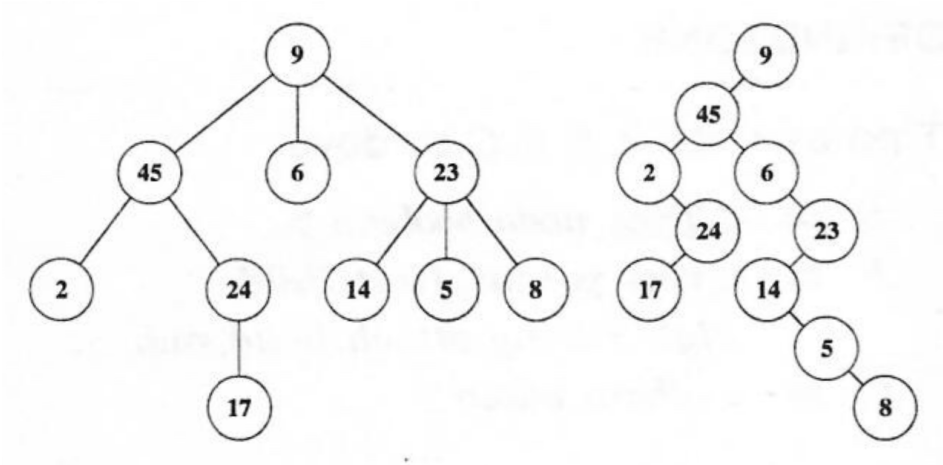
(a) il primo figlio di un nodo viene rappresentato dalla radice del suo sottoalbero sinistro

(b) il fratello destro di un nodo viene rappresentato dalla radice del suo sottoalbero destro

Rappresentazione di un albero n-ario con un albero binario

MEMORIA: O(n)

Più efficiente che con liste !!



Section 3

Rappresentazioni dei grafi

Grafi orientati

x

Ricordiamo la definizione di grafo orientato:

Si dice grafo orientato G una coppia ordinata (N,A), dove N è un insieme finito di elementi qualsiasi, detti nodi, e A, detto insieme degli archi, è una relazione binaria su N

Un grafo non orientato G = (V, E) è costituito da un insieme V di vertici e da un insieme E di spigoli, elementi di una relazione simmetrica (e normalmente antiriflessiva) su V

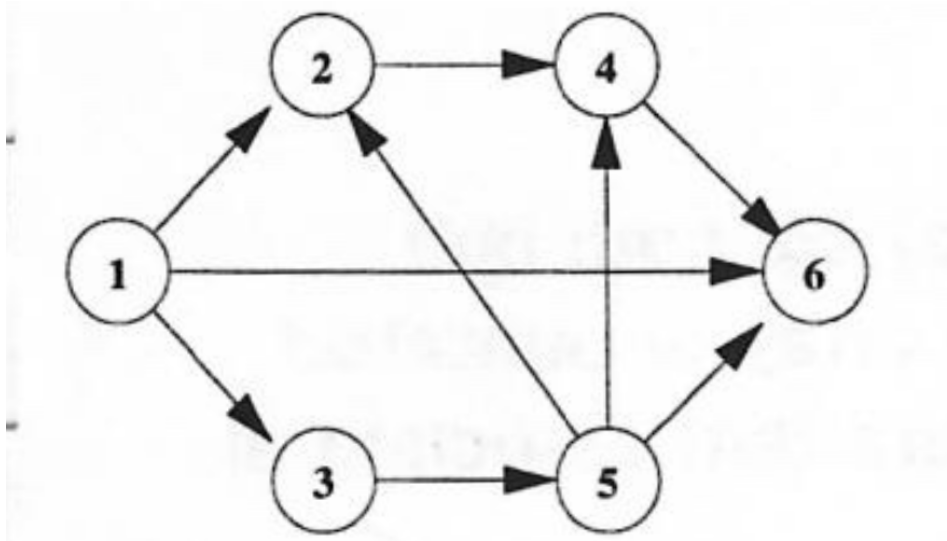
Nel seguito ci occupiamo in particolare di grafi orientati

Matrici di adiacenza

DEFINIZIONE
Si dice matrice di adiacenza (nodi/nodi) di un grafo G = (N, A) una matrice binaria B così definita:

$$A = (b_{ij}), \quad b_{ij} = \begin{cases} 1 & \text{se } (n_i, n_j) \in A \\ 0 & \text{altrimenti} \end{cases}$$

```
A = [0 1 1 0 0 1
     0 0 0 1 0 0
     0 0 0 0 1 0
     0 0 0 0 0 1
     0 1 0 1 0 1
     0 0 0 0 0 0]
```



Matrici di adiacenza

- La somma degli elementi della matrice eguaglia il numero degli archi:

$$\sum_i \sum_j = |A|$$

- La somma per righe è pari al numero di archi uscenti da un nodo:

$$\sum_j = outdegree(n_i)$$

- La somma per colonne è pari al numero di archi entranti in un nodo:

$$\sum_i = indegree(n_j)$$

Matrici di adiacenza

```
[0 1 1 0 0 1;        [0 0 0 0 0 0;        [0 1 1 0 0 1;
 0 0 0 1 0 0;         1 0 0 0 1 0;         1 0 0 1 1 0;
 0 0 0 0 1 0;  +      1 0 0 0 0 0;  =      1 0 0 0 1 0;
 0 0 0 0 0 1;         0 1 0 0 1 0;         0 1 0 0 1 1;
 0 1 0 1 0 1;         0 0 1 0 0 0;         0 1 1 1 0 1;
 0 0 0 0 0 0]         1 0 0 1 1 0]         1 0 0 1 1 0]
```

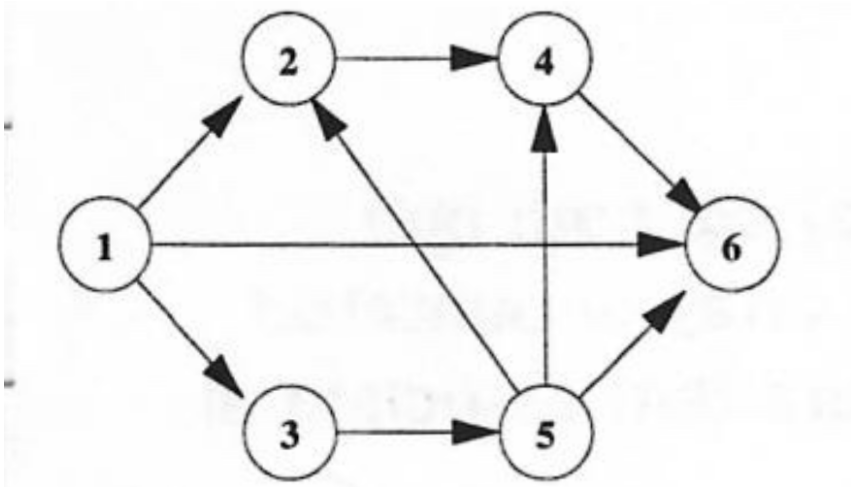
Data la matrice di adiacenza A di un grafo orientato (senza cappi e archi doppi), la matrice di adiacenza $B^\top$, del corrispondente grafo non orientato sugli stessi nodi si ottiene sommando B e la sua trasposta

```
A + A' = B
```

Matrici di incidenza

Si dice matrice di incidenza (nodi/archi) di un grafo orientato $G = (N, A)$ una matrice C a valori in $\{-1, 0, 1\}$ così definita:

$$C = (c_{ij}), \quad c_{ij} = \begin{cases} -1 & \text{se } n_i \in A_j \\ 1 & \text{se } n_j \in A_i \\ 0 & \text{altrimenti} \end{cases}$$




```
C = [-1   0   0  -1   0   0  -1   0   0
      1  -1   0   0   0   0   0   1   0
      0   0   0   1  -1   0   0   0   0
      0   1  -1   0   0   0   0   0   1
      0   0   0   0   1  -1   0  -1  -1
      0   0   1   0   0   1   1   0   0]
```

Section 4

Operazioni topologiche su grafi

bbbbbb

Section 5

Noteworthy differences from Python

Noteworthy differences from Python 1/3

https://docs.julialang.org/en/v1/manual/noteworthy-differences/

1. Julia's for, if, while, etc. blocks are terminated by the end keyword. Indentation level is not significant as it is in Python. Unlike Python, Julia has no pass keyword.

Noteworthy differences from Python 1/3

https://docs.julialang.org/en/v1/manual/noteworthy-differences/

1. Julia's for, if, while, etc. blocks are terminated by the end keyword. Indentation level is not significant as it is in Python. Unlike Python, Julia has no pass keyword.
2. Strings are denoted by double quotation marks ("text") in Julia (with three double quotation marks for multi-line strings), whereas in Python they can be denoted either by single ('text') or double quotation marks ("text"). Single quotation marks are used for characters in Julia ('c').

Noteworthy differences from Python 1/3

https://docs.julialang.org/en/v1/manual/noteworthy-differences/

1. Julia's for, if, while, etc. blocks are terminated by the end keyword. Indentation level is not significant as it is in Python. Unlike Python, Julia has no pass keyword.
2. Strings are denoted by double quotation marks ("text") in Julia (with three double quotation marks for multi-line strings), whereas in Python they can be denoted either by single ('text') or double quotation marks ("text"). Single quotation marks are used for characters in Julia ('c').
3. String concatenation is done with * in Julia, not + like in Python. Analogously, string repetition is done with ^, not *. Implicit string concatenation of string literals like in Python (e.g. 'ab' 'cd' == 'abcd') is not done in Julia.

Noteworthy differences from Python 1/3

https://docs.julialang.org/en/v1/manual/noteworthy-differences/

1. Julia's for, if, while, etc. blocks are terminated by the end keyword. Indentation level is not significant as it is in Python. Unlike Python, Julia has no pass keyword.
2. Strings are denoted by double quotation marks ("text") in Julia (with three double quotation marks for multi-line strings), whereas in Python they can be denoted either by single ('text') or double quotation marks ("text"). Single quotation marks are used for characters in Julia ('c').
3. String concatenation is done with * in Julia, not + like in Python. Analogously, string repetition is done with ^, not *. Implicit string concatenation of string literals like in Python (e.g. 'ab' 'cd' == 'abcd') is not done in Julia.
4. Python Lists—flexible but slow—correspond to the Julia Vector{Any} type or more generally Vector{T} where T is some non-concrete element type. "Fast" arrays like Numpy arrays that store elements in-place (i.e., dtype is np.float64, [('f1', np.uint64), ('f2', np.int32)], etc.) can be represented by Array{T} where T is a concrete, immutable element type. This includes built-in types like Float64, Int32, Int64 but also more complex types like Tuple{UInt64,Float64} and many user-defined types as well.

Noteworthy differences from Python 2/3

1. In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.

Noteworthy differences from Python 2/3

1. In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
2. Julia's slice indexing includes the last element, unlike in Python. a[2:3] in Julia is a[1:3] in Python.

Noteworthy differences from Python 2/3

1. In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
2. Julia's slice indexing includes the last element, unlike in Python. a[2:3] in Julia is a[1:3] in Python.
3. Julia does not support negative indices. In particular, the last element of a list or array is indexed with end in Julia, not -1 as in Python.

Noteworthy differences from Python 2/3

1. In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
2. Julia's slice indexing includes the last element, unlike in Python. a[2:3] in Julia is a[1:3] in Python.
3. Julia does not support negative indices. In particular, the last element of a list or array is indexed with end in Julia, not -1 as in Python.
4. Julia requires end for indexing until the last element. x[1:] in Python is equivalent to x[2:end] in Julia.

Noteworthy differences from Python 2/3

1. In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
2. Julia's slice indexing includes the last element, unlike in Python. a[2:3] in Julia is a[1:3] in Python.
3. Julia does not support negative indices. In particular, the last element of a list or array is indexed with end in Julia, not -1 as in Python.
4. Julia requires end for indexing until the last element. x[1:] in Python is equivalent to x[2:end] in Julia.
5. Julia's range indexing has the format of x[start:step:stop], whereas Python's format is x[start:(stop+1):step]. Hence, x[0:10:2] in Python is equivalent to x[1:2:10] in Julia. Similarly, x[::-1] in Python, which refers to the reversed array, is equivalent to x[end:-1:1] in Julia.

Noteworthy differences from Python 3/3

1. In Julia, indexing a matrix with arrays like X[[1,2], [1,3]] refers to a sub-matrix that contains the intersections of the first and second rows with the first and third columns. In Python, X[[1,2], [1,3]] refers to a vector that contains the values of cell [1,1] and [2,3] in the matrix. X[[1,2], [1,3]] in Julia is equivalent with X[np.ix_([0,1],[0,2])] in Python. X[[0,1], [0,2]] in Python is equivalent with X[[CartesianIndex(1,1), CartesianIndex(2,3)]] in Julia.

Noteworthy differences from Python 3/3

1. In Julia, indexing a matrix with arrays like X[[1,2], [1,3]] refers to a sub-matrix that contains the intersections of the first and second rows with the first and third columns. In Python, X[[1,2], [1,3]] refers to a vector that contains the values of cell [1,1] and [2,3] in the matrix. X[[1,2], [1,3]] in Julia is equivalent with X[np.ix_([0,1],[0,2])] in Python. X[[0,1], [0,2]] in Python is equivalent with X[[CartesianIndex(1,1), CartesianIndex(2,3)]] in Julia.
2. Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.

Noteworthy differences from Python 3/3

1. In Julia, indexing a matrix with arrays like X[[1,2], [1,3]] refers to a sub-matrix that contains the intersections of the first and second rows with the first and third columns. In Python, X[[1,2], [1,3]] refers to a vector that contains the values of cell [1,1] and [2,3] in the matrix. X[[1,2], [1,3]] in Julia is equivalent with X[np.ix_([0,1],[0,2])] in Python. X[[0,1], [0,2]] in Python is equivalent with X[[CartesianIndex(1,1), CartesianIndex(2,3)]] in Julia.
2. Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
3. Julia arrays are column major (Fortran ordered) whereas NumPy arrays are row major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy (see relevant section of Performance Tips).