Appunti di Algoritmi e Strutture Dati

Domande e risposte

1) Algoritmo di Quicksort

L'algoritmo **Quicksort** è un algoritmo di ordinamento ricorsivo con tecnica divide et impera, che ha le seguenti caratteristiche:

→ sceglie un elemento k, detto pivot o perno;

→ partiziona l'array V[l,…,r] in sottoarray di V indicati con V[l,…,q] e V[q,…,r] con q appartenente a V tale che tutti gli elementi V[l,…,q] siano <= k e V[q,…,r] siano >= k;

→ ordina ricorsivmente le due parti, con la stessa tecnica.

Pseudocodice di Quicksort

```
Quicksort(V,l,r) {
        if (l<r) {
            q = Partition(V,l,r);
            Quicksort(V,l,q);
            Quicksort(V,q,r);
        }
}
```

Partizionamento dell'array

Esistono due modalità per partizionare un vettore:

1 – Partizione di Hoare: che si compone di quattro sotto – fasi:

→ impone k = V[l];

→ scansiona V da destra a sinistra, fermandosi su un elemento <= k;

→ scansiona V da sinistra a destra, fermandosi su un elemento >= k;

→ scambia i due elementi e riprende la scansione, a meno che i due indici si sono sovvrapposti.

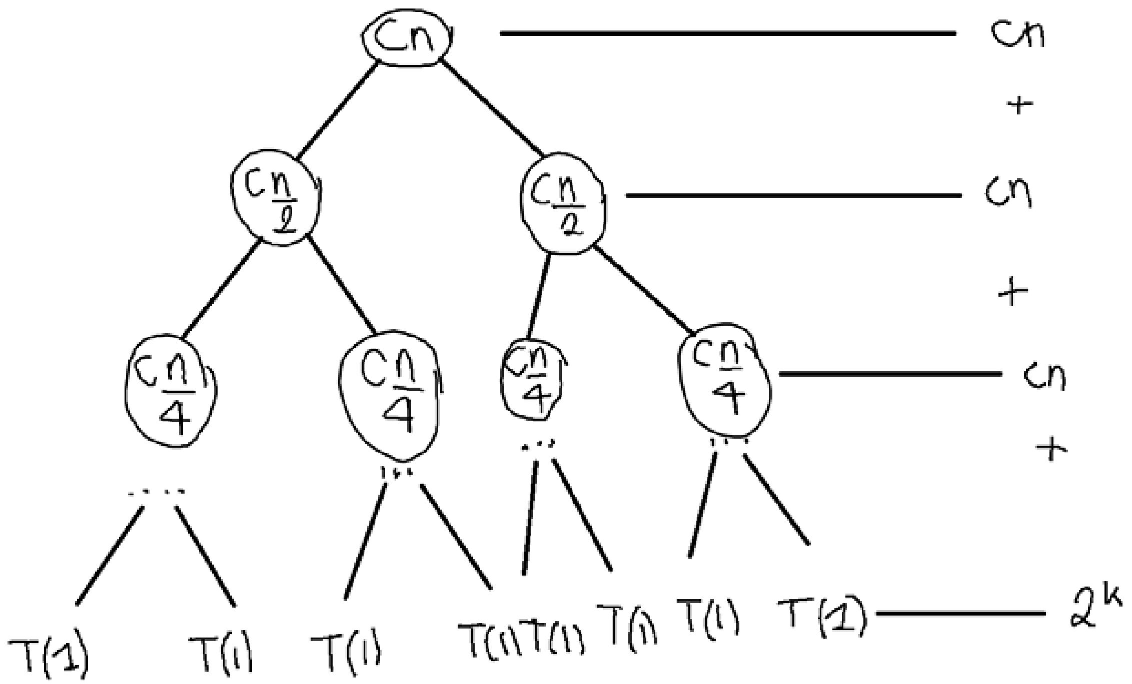
2 – Partizionamento di Lomuto: come Hoare ma al contrario.

Complessità

La sua complessità si risolve con l'albero delle chaimate, secondo:

$$T(n) = \begin{cases} T(1) = 1 \\ T(n) = 2T\left(\frac{n}{2}\right) + cn \end{cases}$$

Con l'albero di ricorrenza si ha



Dall'albero si ottiene:

$$T(n) = kcn + 2^k = \{2^{\frac{n}{k-1}} = 2\} = \{n = 2^k\} = \{k = \log_2 n\}$$

$$T(n) = \log_2 n \cdot cn + n$$

$$T(n) = n(\log_2 n\, c + 1)$$

$$T(n) = \Theta(n \log_2 n)$$

$$T(n) = \Theta(n \log n)$$

La complessità di Quicksort è $\Theta(n\log n)$.

Simulazione Algoritmo di Quicksort

Dato il seguente array, ordinarlo con Quicksort

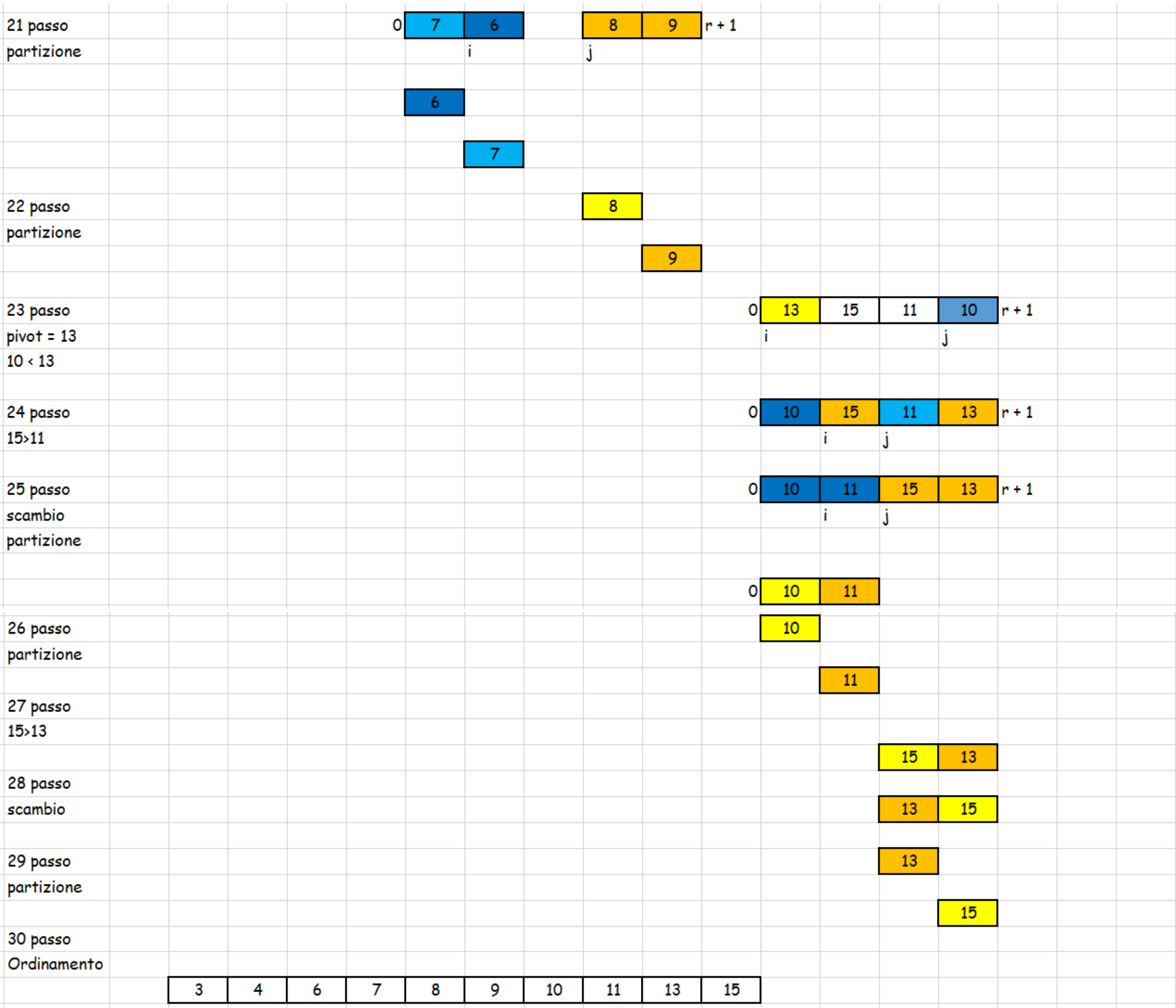
10	8	3	15	7	9	13	4	11	6

Soluzione

1 passo	0	10	8	3	15	7	9	13	4	11	6	r + 1
pivot = 10	i	p									r	j
2 passo	0	10	8	3	15	7	9	13	4	11	6	r + 1
pivot = 10		i									j	
3 passo	0	6	8	3	15	7	9	13	4	11	10	r + 1
scambio		i									j	
4 passo	0	6	8	3	15	7	9	13	4	11	10	r + 1
p < 11		i								j	p	

Gianluca Rota 1 L-31 ALGORITMI E STRUTTURE DATI 01/05/2023

5 passo	0	6	8	3	15	7	9	13	4	11	10	r + 1		
4 < p		i							j		p			
6 passo	0	6	8	3	15	7	9	13	4	11	10	r + 1		
4 < p < 15					i				j		p			
7 passo	0	6	8	3	15	7	9	13	4	11	10	r + 1		
4 < p < 15					i				j		p			
8 passo	0	6	8	3	4	7	9	13	15	11	10	r + 1		
scambio					i				j		p			
9 passo	0	6	8	3	4	7	9	13	15	11	10	r + 1		
stop							j	i			p			
10 passo	0	6	8	3	4	7	9			0	13	15	11	10	r + 1
partizione															
11 passo	0	6	8	3	4	7	9								
pivot = 6															
12 passo	0	6	8	3	4	7	9								
4 < 6		i				j									
13 passo	0	4	8	3	6	7	9								
scambio		i				j									
14 passo	0	4	8	3	6	7	9								
3 < 6		i		j											
15 passo	0	4	8	3	6	7	9								
8 > 6			i	j											
16 passo	0	4	8	3	6	7	9								
8 > 6			i	j											
17 passo	0	4	3	8	6	7	9								
scambio			i	j											
18 passo	0	4	3	r + 1	0	8	6	7	9	r + 1					
partizione															
		3													
partizione															
			4												
19 passo					0	8	6	7	9	r + 1					
7 < 8						p			r						
20 passo					0	7	6	8	9	r + 1					
scambio						p			r						



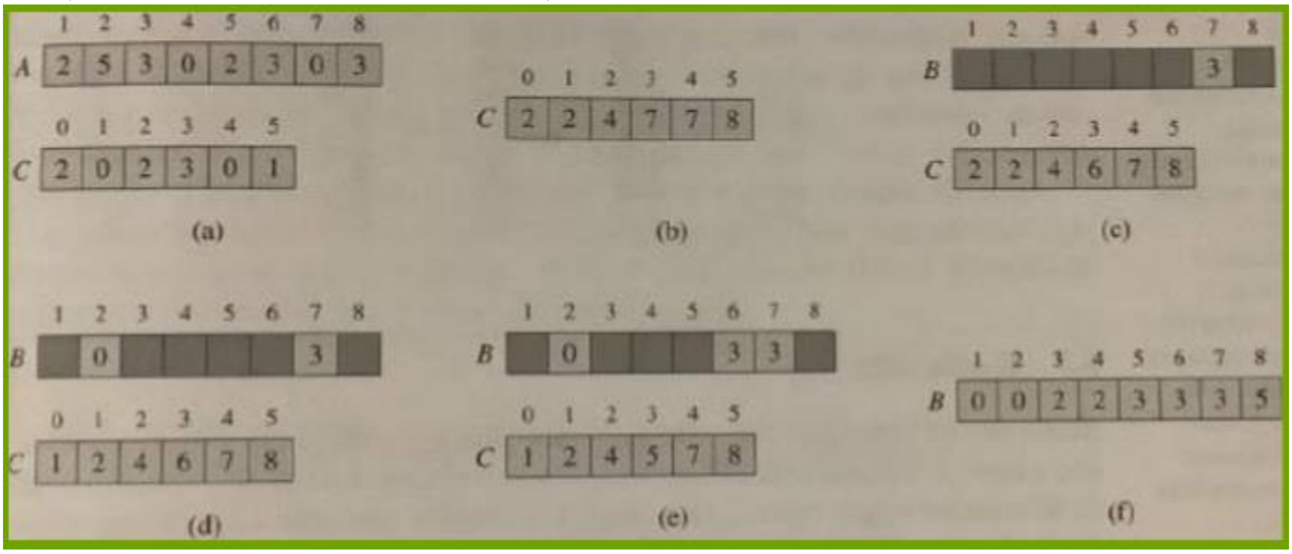
2) Algoritmo di Counting Sort

L'algoritmo di Counting Sort è un algoritmo di ordinamento per valori numerici interi con complessità lineare. L'algoritmo si basa sulla conoscenza a priori dell'intervallo in cui sono compresi i valori da ordinare.

Descrizione intuitiva

L'algoritmo conta il numero di occorrenze di ciascun valore presente nell'array da ordinare, memorizzando questa informazione in un array temporaneo di dimensione pari all'intervallo di valori. Il numero di ripetizioni dei valori inferiori indica la posizione del valore immediatamente successivo.
Supponiamo che ciascuno degli n elementi di input sia un numero intero

compreso nell'intervallo da 0 a k, per qualche intero k.



Valutazione tempi: $T(n) = \Theta(n + k)$

3) Algoritmo di Radix Sort

L'algoritmo di Radix Sort è un algoritmo di ordinamento dotato con più campi chiave (ad esempio per ordinare le date dotate di giorno, mese e anno).
Supponendo un array A di elementi con d cifre, dove d è quella di ordine più alto, si ha:

```
RadixSort(A,d) {
   for (i=1 to d)
        //ordinamento stabile per ordinare l'array A sulla cifra
}
```

Valutazione tempi: $T(n) = O(k*n)$

4) Struttura dati: significato

Una struttura dati è un insieme di dati logicamente correlati e opportunamente memorizzati, per i quali sono definiti degli operatori di costruzione, selezione e manipolazione.
Le strutture dati basata sulla loro occupazione di memoria:
→ strutture dati statiche: la quantità di memoria di cui esse necessitano è determinabile a priori (array);
→ strutture dati dinamiche: la quantità di memoria di cui esse necessitano varia a tempo d'esecuzione e può essere diversa da esecuzione a esecuzione (liste, alberi, grafi).

5) Heap: definizione

Struttura dati costituita da un array A su un dominio D cui corrisponde un albero binario quasi completo (tutti i livelli sono riempiti completamente, tranne al più l'ultimo che è riempito da sinistra fino ad un certo punto).

Ad ogni nodo è associato un indice dell'array tale che:

→ i = 1 è l'indice della radice;

→ se i è l'indice di un nodo allora:

a) [i/2] → è l'indice del padre (`parent(i)`);

b) 2i è l'indice del figlio sinistro (`left(i)`);

c) 2i + 1 è l'indice del figlio destro (`right(i)`);

→ il nodo indice i è etichettato e contiene A[i];

→ `A.heap_size` indica il numero di elementi dello heap memorizzati in A → 0 <= `A.heap_size` <= `A.length`

6) Proprietà Heap

Vale che per ogni valore [i] appartenente a {2, …, `A.heap_size`}, si ha:

→ `A[parent(i)] >= A[i]` → max_heap (radice elemento maggiore)

→ `A[parent(i)] <= A[i]` → min_heap

Attenzione: un array A ordinato in modo decrescente corrisponde ad uno heap, non vale viceversa.

7) Algoritmo di Heapify

L'algoritmo di Heapify permette di trasformare un qualunque array A in modo che gli sia associato un HEAP sfruttando le sue proprietà:

→ input: A e i tale che `left(i)` e `right(i)` sono radici di alberi binari che rappresentano heap ma non è detto che l'albero con radice i rappresenti uno heap;

→ output: A modificato in modo tale che anche l'albero con radice i rappresenti uno heap;

→ algoritmo: far "scendere" A[i] in uno dei due sottoalberi dell'albero la cui radice ha indice i, ovvero questi con radice maggiore:

`left(i)`: se `A[left(i)] >= A[right(i)]`

`right(i)`: se `A[left(i)] < A[right(i)]`

Implementazione Heapify

```
Heapify(A,i) {
        l = left(i);
        r = right(i);
        if (l <= A.heap_size && A[l] > A[i])
                max = l;
        else
                max = i;
        if (r <= A.heap_size && A[r] > A[i])
                max = r;
        else
                max = i;
        if (max != i) {
                swap(A[i],A[max])
                heapify(A,max);
        }
}
```

Implementazione Build Heap

```
BuildHeap(A) {
        A.heap_size = A.length;
        for (i = [A.length/2] down to 1)
                Heapify(A,i);
}
```

Valutazione tempi: T(n) = O(nlogn)

8) Algoritmo di Heapsort

L'algoritmo di Heapsort ordina un array A di lunghezza n:

1) BuildHeap(A) ora in A[i] c'è l'elemento più grande;
2) A[i] ←→ A[n], A[1,..., n-1] non è detto che si uno heap;
3) A.heapsize – si vuole ora trasformare A[1, …, n-1] in heap;
4) Heapify(A,1), ora in A[i] c'è il più grande di A[1, …, n-1].

Implementazione di Heapsort

```
HeapSort(A) {
        BuildHeap(A);
        for (i = A.length down to 2) {
                scambia(A[1],A[i]);
                A.heap_size--;
                Heapify(A,1);
        }
}
```

Valutazione tempi:

T(n) = O(n) + (n-1) (2 + nO(logn))

$T(n) = O(n) + O(n\log n)$

$T(n) = O([n/2]\log n)$

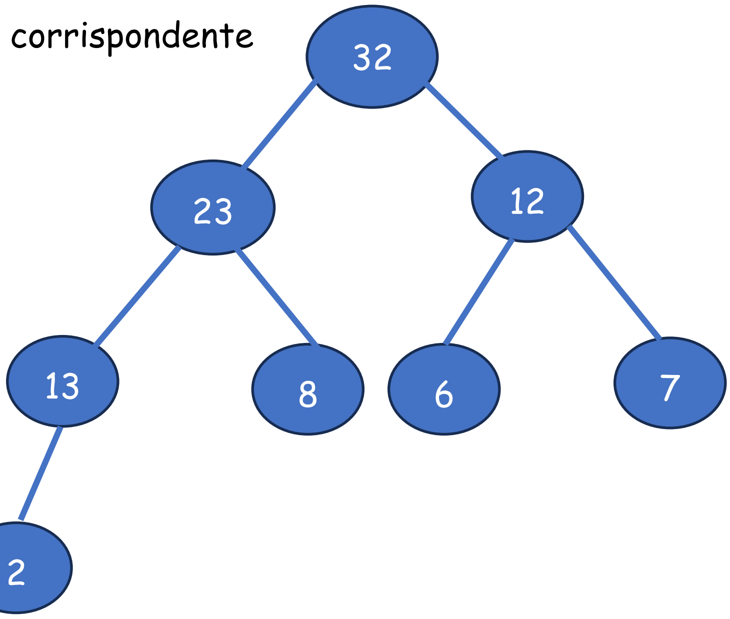
Simulazione

Dato il seguente array A, ordinarlo con HEAPSORT.

32	23	12	13	8	6	7	2

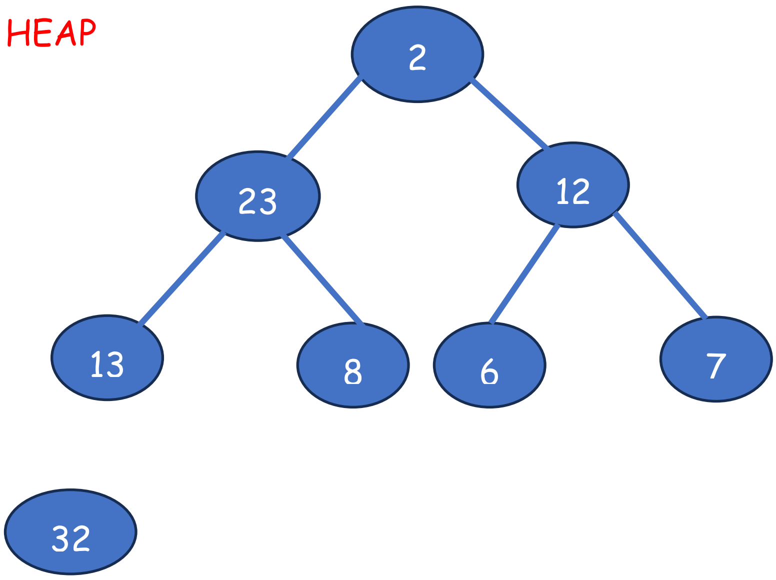
Soluzione

1 – HEAP corrispondente



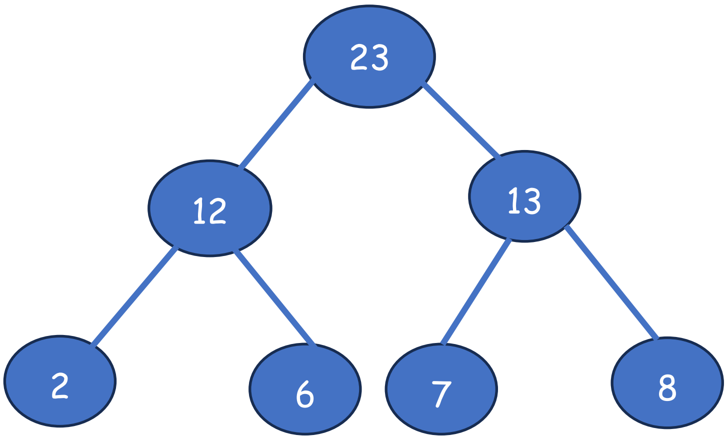
32	23	12	13	8	6	7	2

2 – Scambio 32 con 2 e rimuovo 32 dallo HEAP. Decremento la dimensione dello HEAP



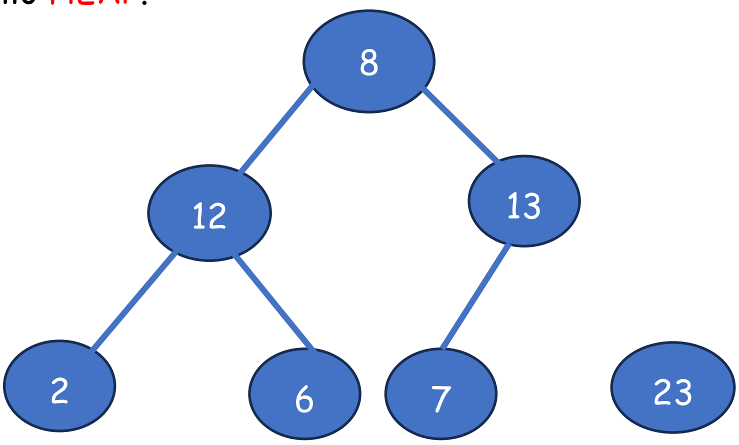
2	23	12	13	8	6	7	32

3 – Applico la Build Max Heap



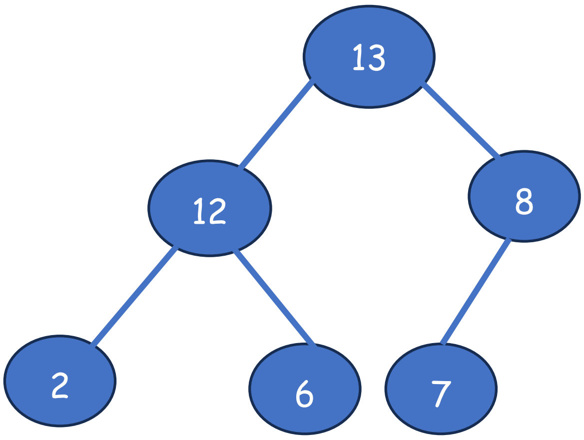
23	12	13	2	6	7	8	32

4 – Scambio il 23 con 8 e rimuovo 23 dallo HEAP. Decremento la dimensione dello HEAP.



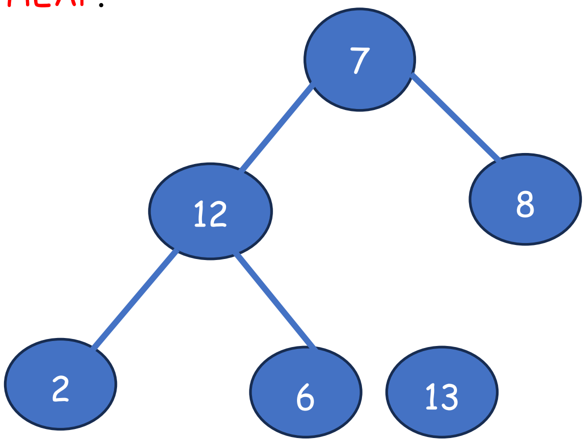
8	12	13	2	6	7	23	32

5 – Applico la Build Max Heap



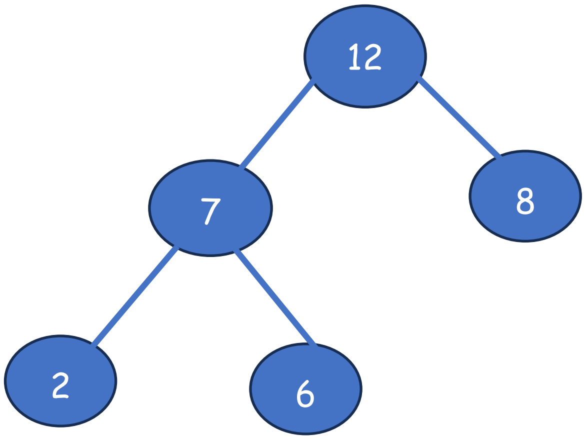
13	12	8	2	6	7	23	32

6 – Scambio 7 con 13 e rimuovo 13 dallo HEAP. Decremento la dimensione dello HEAP.



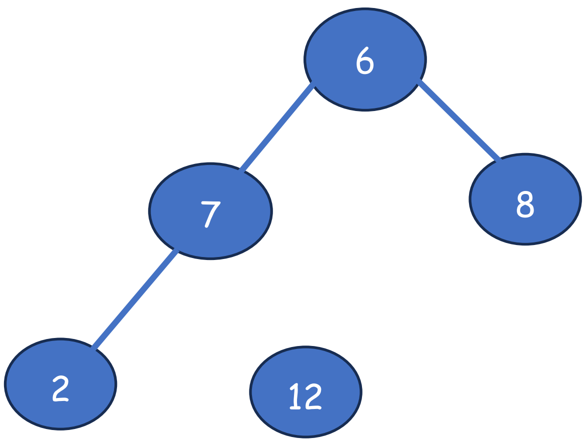
7	12	8	2	6	13	23	32

7 – Applico la Build Max Heap sugli elementi rimanenti.



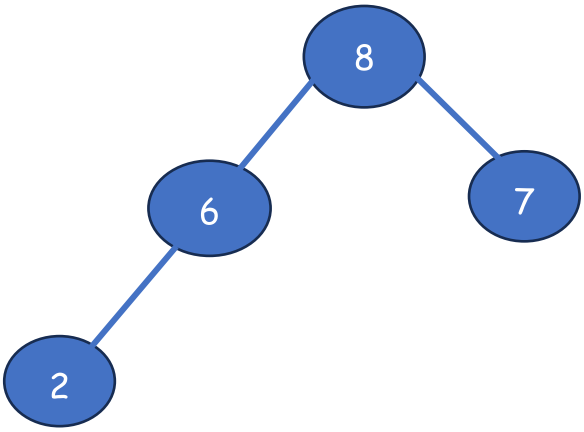
12	7	8	2	6	13	23	32

8 – Scambio 12 con 6 rimuovo 12 dallo HEAP. Decremento la dimensione dello HEAP.



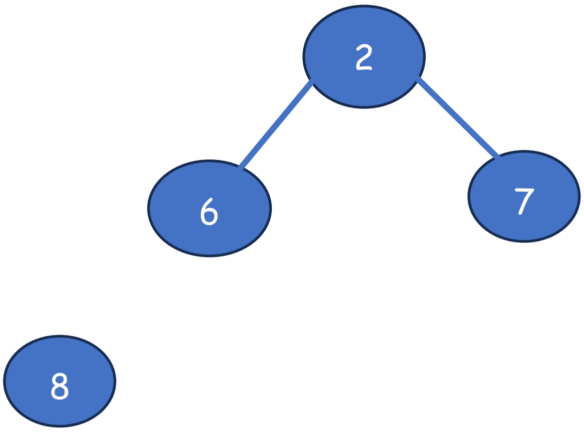
6	7	8	2	12	13	23	32

9 – Applico la Build Max Heap.



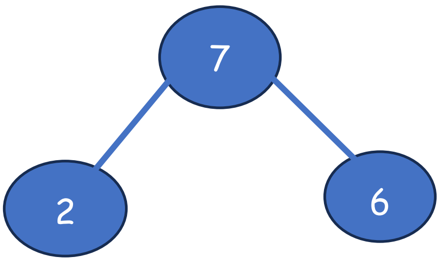
8	6	7	2	12	13	23	32

10 – Scambio 8 con 2 e rimuovo 8 dallo HEAP. Decremento la dimensione dello HEAP.



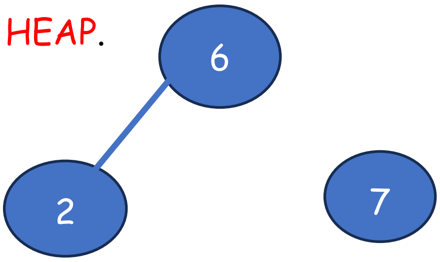
2	6	7	8	12	13	23	32

11 – Riapplico la Build Max Heap



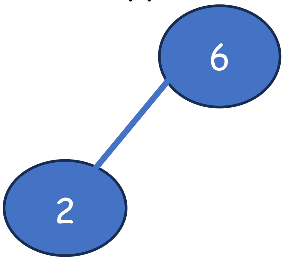
7	2	6	8	12	13	23	32

12 – Scambio 6 con 7 e rimuovo 7 dallo HEAP. Decremento la dimensione dello HEAP.



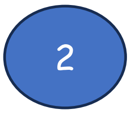
6	2	7	8	12	13	23	32

13 – Riapplico la Build Max Heap per i due elementi rimasti.



6	2	7	8	12	13	23	32

14 – Elimino 6 dallo HEAP scambiando 6 e 2 e decremento la dimensione dello HEAP.



2	6	7	8	12	13	23	32

15 – Gli elementi dell'array sono ordinati.

2	6	7	8	12	13	23	32

9) Code con Priorità

La coda con priorità (priority queue) è una struttura dati che ha le seguenti caratteristiche:

→ memorizza un insieme dinamico su un dominio D;

→ consente di accedere al Max in tempo costante;

→ estrazione/rimozione del Max;

→ inserimento di un elemento;

→ realizzato tramite HEAP.

Operazioni

Accesso al max: $T(n) = \Theta(1)$;

Rimozione max:

```
A[1] <-> A[A.heap-size]
A. heap-size--;
Heapify(A,1)
```

Worst Case: O (log n)

→ inserimento elemento:

```
A. heap-size++;
A[A.heap-size] = k
```

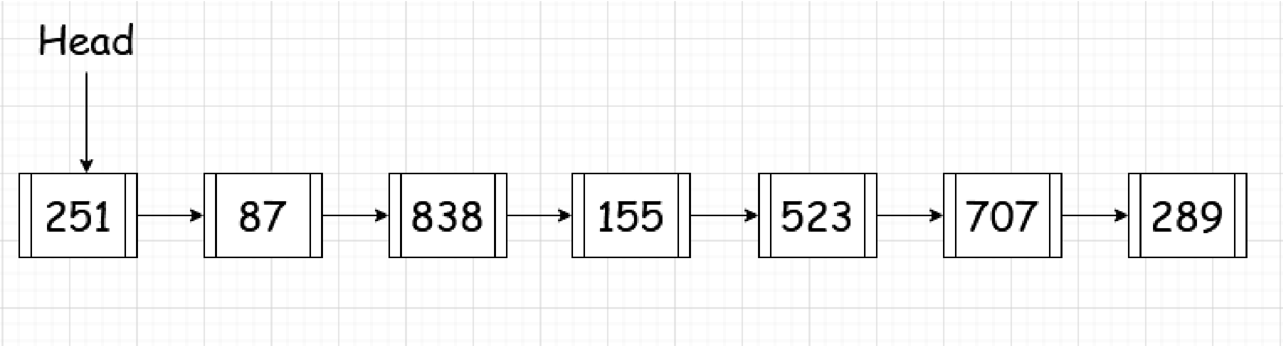
10) Liste: definizione caratteristiche

La lista (list) è un insieme dinamico su un dominio D, visto come una successione di elementi. La lista è una struttura con le seguenti caratteristiche:

→ è formata da elementi tutti dello stesso tipo;

→ è una struttura dinamica, nel senso che la sua dimensione può variare (cioè aumentare o diminuire – struttura a fisarmonica) durante l'esecuzione.

Immagine lista:



Una lista può essere di due tipologie:

→ lista semplicemente concatenata: in cui ciascun record contiene il campo key e un puntatore next (punta al nodo successivo).

→ lista doppiamante concatenata: è una struttura dati che consiste in una sequenza di elementi costituiti da:

a) campo key;

b) due puntatori: next (puntatore al nodo successivo) e prev (puntatore al nodo precedente).

Ciascun elemento è associato ad un oggetto il cui campo key contiene l'elemento:

→ head: punta al primo elemento della lista;

→ si ha un accesso sequenziale, ovvero bisogna effettuare una scansione per effettuare una ricerca di un elemento.

Esempio Liste:

```
-| è una lista          (<>)
(5,-|) è una lista      (<5>)
(8,-|) è una lista      (<8>)
(12,(3,-|)) è una lista (<12,3>)
```

Definizione Struttura lista

```
struct list {
        int key;
        list* next;
}
```

Implementazioni Liste

1) Scansione Lista

```
scanList(L) {
        x = L.head;
        while (x != null) {
                print(x.key);
                x = x.next;
        }
}
```

2) Ricerca Elemento Lista

```
listSearch(L,n) {
        x = L.head;
        while (x != null && x.key != n)
                x = x.next;
        return x;
}
```

3) Inserimento in testa della Lista

```
listHeadInsert(L,n) {
        if (L != <>)
                L.head.prev = n;
        L.head = x;
        x.prev = NULL;
}
```

4) Cancellazione Elemento Lista

```
listDelete(L,x) {
        if (x.prev != NULL)
                x.prev.next = x.next;
        else
                L.head = x.next;
        if (x.next != NULL)
                x.next.prev = x.prev;
}
```

11) Stack (Pile): definizione e caratteristiche

Uno stack (o pila) è una struttura dati lineare a cui si può accedere mediante uno dei suoi estremi, sia per memorizzare sia per estrarre i dati.
Può essere identificata come una sequenza dove:
→ S = <> sequenza vuota;
→ S = <a1,…,an> sequenza di elementi tale che 1 <= i <= n e an è un elemento in cima alla pila S.
Politica LIFO: le operazioni di cancellazione e inserimento da/in una qualunque pila S sono tali che "l'ultimo elemento inserito in S è il primo ad essere cancellato".

Operazioni con lo stack

Sia S l'insieme di tutte le possibili pile su un dominio D:
1 – Inserimento: Push

$S \times D \rightarrow S$

$\forall (S, x) \in S \times D$

se $S = \langle\rangle$, PUSH $(S, x) = \langle x \rangle \in S$

se $S = \langle a_1, \dots, a_n \rangle$, PUSH $(S, x) = \langle a_1, \dots, a_n, x \rangle \in S$

Implementazione

```
Push(S,x) {
    if (A.top == m)
        error("overflow");
    else {
        A.top++;
        A[A.top]=x;
    }
}
```

2 – Cancellazione: Pop

$S \setminus \{\langle\rangle\} \rightarrow S \times D$

$\forall S \in S$ con $S \neq \langle\rangle$ $(S = \langle a_1, \dots, a_n \rangle)$

$POP(S) = (\langle a_1, \dots, a_{n-1} \rangle, a_n)$

Implementazione

```
Pop(S,x) {
   if (A.top == 0)
        error("Underflow");
   else
        A.top--;
   return A[A.top+1];
}
```

3 - Interrogazione: Stack - Empty

$S \to \{0, 1\}$

$\forall s \in S$

$$STACK\_EMPTY(S) = \begin{cases} 0 & S \neq \langle\rangle \\ 1 & S = \langle\rangle \end{cases}$$

Implementazione

```
Stack_Empty(S) {
    if (S = <>)
        return true;
    else
        return false;
}
```

4 - Interrogazione: Top

$S \setminus \{\langle\rangle\} \to D$

$\forall s \in S$ con $s \neq \langle\rangle$ $(s = \langle a_1, \dots, a_n \rangle)$

$TOP(S) = a_n$

Implementazione di una pila mediante Array

Una pila `S = <a1, …, an>` con al più m elementi (cioè n <= m) può essere implementata tramite un array con A.length = m.

Essa ha le seguenti caratteristiche:

→ A.top indice dell'ultimo elemento inserito in S;

→ A [1, …, A.top] memorizza S;

→ A [1] elemento in fondo A[A.top] elemento in cima;

→ A.top = 0 ←→ S = ‹›.

Esempio

`S = <13,4,11,5,15>`

13	4	11	5	15			

A.top = 5 = n

12) Code: definizione e caratteristiche

La coda (queue) è un insieme dinamico su un dominio D che cresce aggiungendo elementi in fondo e si accorcia rimuovendo elementi dall'inzio. Rispetto allo stack la coda possiede due puntatori agli estremi: Testa e Coda che vengono utilizzate per aggiungere e rimuovere elementi dalla struttura. Può essere identificata come una sequenza A dove:

→ Q = <> sequenza vuota;

→ Q = <a1, …, an> sequenza di n elementi t.c. 1 <= i <= n si ha:

1. L'elemento a1 è la testa;
2. L'elemento an è la coda.

Politica FIFO: le operazioni di cancellazione e inserimento da/in una qualunque coda Q sono tali che "il primo elemento inserito in Q è il primo ad essere cancellato".

Operazioni con la coda

1 – Inserimento: Enqueue

$Q \times D \to Q$

$\forall (Q, x) \in Q \times D$

se $Q = \langle \rangle$, $\text{ENQUEUE}(Q, x) = \langle x \rangle$

se $Q = \langle a_1, \dots, a_n \rangle$, $\text{ENQUEUE}(Q, x) = \langle a_1, \dots, a_n, x \rangle$

Implementazione

```
Enqueue(Q) {
    if ((A.head = 1 AND A.tail = A.length) OR (A.head = A.tail+1))
        error("Overflow");
    else {
        A[A.tail] = x;
        if (A.tail = A.length)
                A.tail = 1;
        else
                A.tail++;
    }
}
```

2 – Cancellazione: Dequeue

$Q \setminus \{\langle \rangle\} \to Q \times D$

$\forall Q \in Q$ con $Q \neq \langle \rangle$ $(Q = \langle a_1, \dots, a_n \rangle)$

$\text{DEQUEUE}(Q) = (\langle a_2, \dots, a_n \rangle, a_1)$

Implementazione

```
Dequeue(Q) {
   if (A.head = A.tail)
        error("underflow");
   else {
        x = A[A.head];
        if (A.head = A.length)
            A.head = 1;
        else
            A.head++;
        return x;
   }
}
```

3 – Interrogazione: Empty – Queue

$Q \to \{0,1\}$

$\forall Q \in Q$

$$EMPTY\_QUEUE = \begin{cases} 0 & Q \neq \langle\rangle \\ 1 & Q = \langle\rangle \end{cases}$$

Implementazione

```
Empty_Queue(Q) {
    return(A.head = A.tail);
}
```

Implementazione Coda con array

Anche la coda può essere implementata attraverso un vettore ma attraverso ciò si riscontra un problema in più: quando viene effettuata l'operazione di rimozione di un elemento di un vettore viene richiesto lo shifting di tutti gli elementi della coda.

Testa			Coda		
23	12	56	45		

Dopo la rimozione della Testa:

Testa			Coda		
12	56	45	11		

Soluzione Ottima: spostare la Testa sull'indice successivo durante la fase di dequeue.

	Testa		Coda		
23	12	56	45		

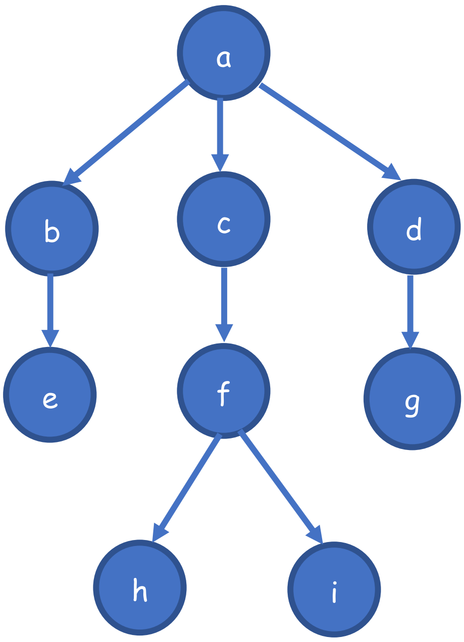
13) Alberi: definizione e caratteristiche

Un albero è un DAG connesso tale che:

1) esiste esttamente un nodo sorgente (radice dell'albero);
2) ogni nodo diverso dalla radice ha un solo arco entrante.

I nodi pozzo di un'albero sono chiamati foglie oppure nodi esterni.

Tutti gli altri nodi sono chiamati interni.



Nodo	Tipologia
<a>	radice
<b,c,d,f>	padre
<e,g,h,i>	foglia
<e,g,h,i>	figlio

→ Proprietà

Il grado di ingresso di un nodo è:

1 se non è la radice;

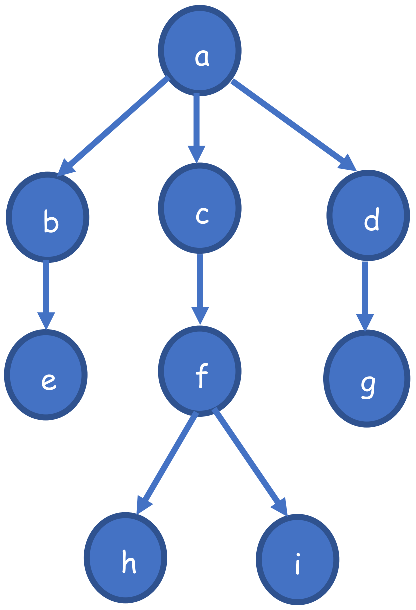
0 se è la radice;

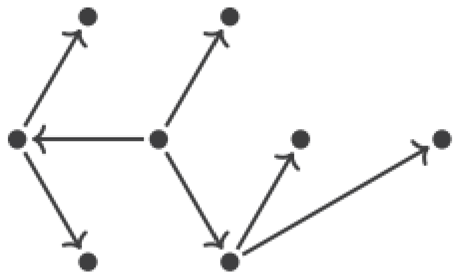
Il grado di uscita di un nodo non ha restrizioni.

Per ogni nodo v che non è la radice, esiste esattamente un cammino della radice a v e non può essere mai vuoto (la radice esiste sempre).

Se un albero è finito, allora esiste almeno una foglia (può essere anche la radice). I nodi intermedi sono sia il padre che il figlio.

→ Rappresentazione



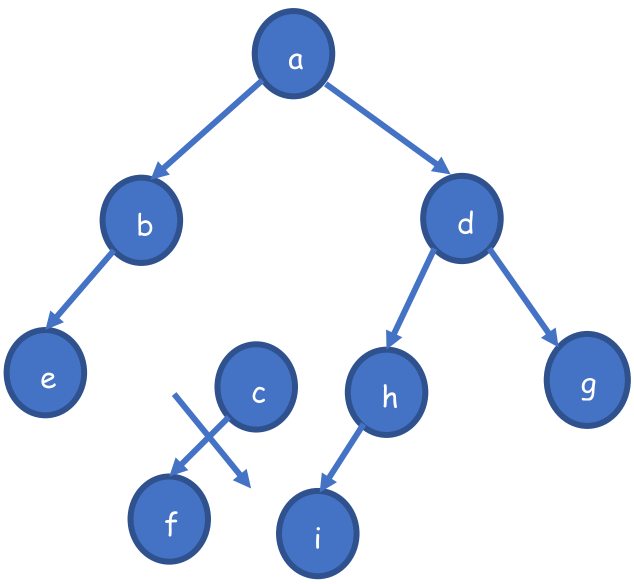


→ Cammini in un albero

In un albero esiste esattamente un cammino dalla radice a qualunque nodo v diverso dalla radice.
Ogni nodo w in questo cammino è un ascendente di v (avo) e v è un discendente di w (radice è l'unico nodo che non ha ascendenti).
Se il cammino da w a v ha lunghezza 1, allora w è il padre di v, e v è un figlio di w.

→ Profondità dell'albero

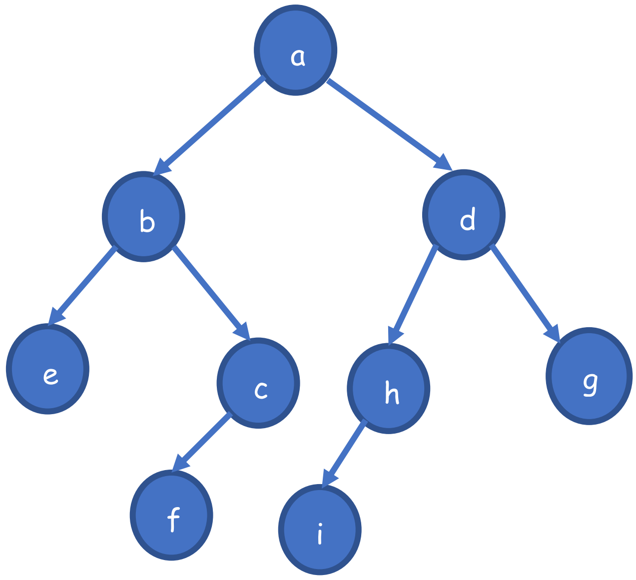
La profondità di un nodo v è la lunghezza del cammino della radice a v.
L'altezza di un albero è la profondità massima dei suoi nodi.



Nodo	Profondità
<a>	radice
<b,c,d,f>	padre
<e,g,h,i>	foglia
<e,g,h,i>	figlio

→ Albero binario

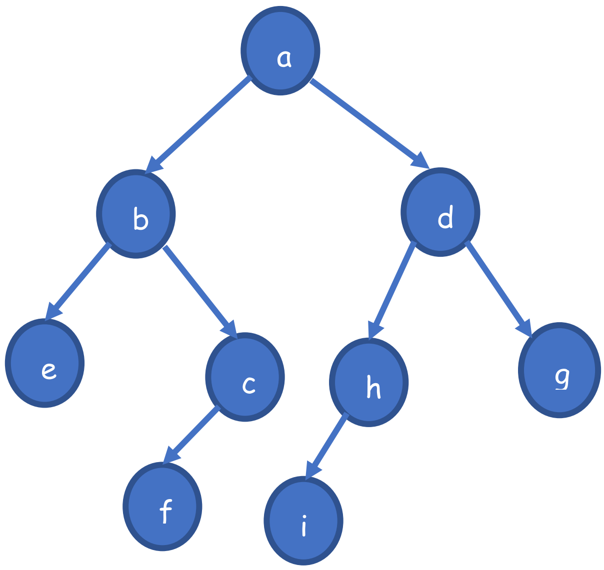
Un albero si dice binario se ogni nodo ha al più due figli. I figli di un nodo in un albero binario sono ordinati (figlio sinistro e figlio destro).



→ Proprietà degli alberi binari

Un albero binario ha al massimo 2^p nodi di profondità p.

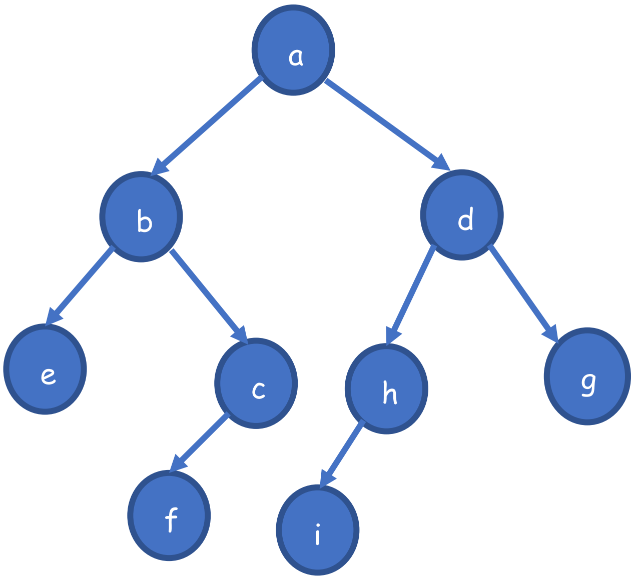
Un albero di altezza n ha al più $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$ nodi.



→ Struttura albero binario

Un albero binario è una struttura ricorsiva composta da:

1) un nodo (radice);
2) un albero binario sinistro (eventualmente vuoto);
3) un albero binario destro (eventualmente vuoto).



Nodo	Tipologia
<a>	radice
<b,c,e,f>	albero sinistro
<d,g,h,i>	albero destro

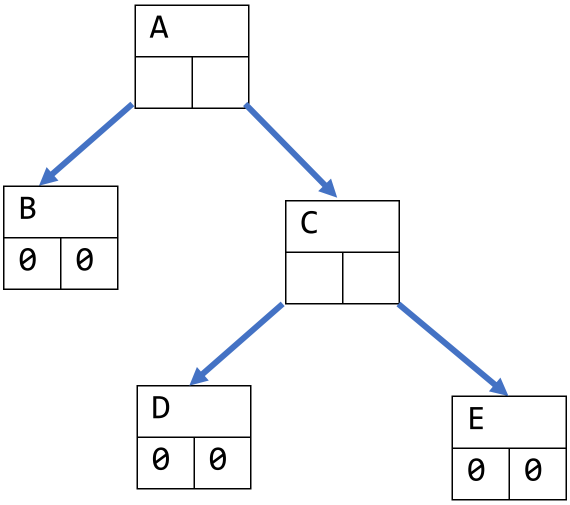
→ Come rappresentarli?

E' possibile rappresentare un albero binario sia come:

1) una collezione di nodi, dove la radice è segnalata e ogni nodo ha due puntatori (alle radici degli alberi destro e sinistro);

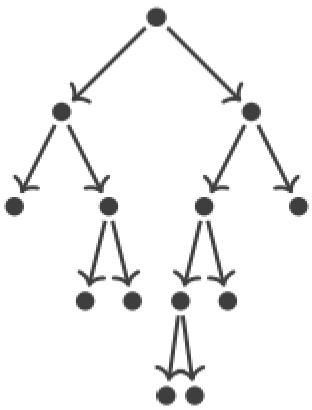
2) come una tabella di $2^{n+1}-1$ righe, dove n è l'altezza dell'albero.

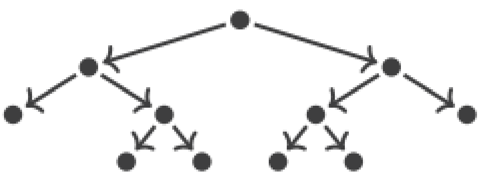
1	A	1
2	B	1
3	C	1
4	0	0
5	0	0
6	D	1
7	E	1



→ Alberi binari pieni

Un albero binario è pieno se ogni nodo interno ha due figli.





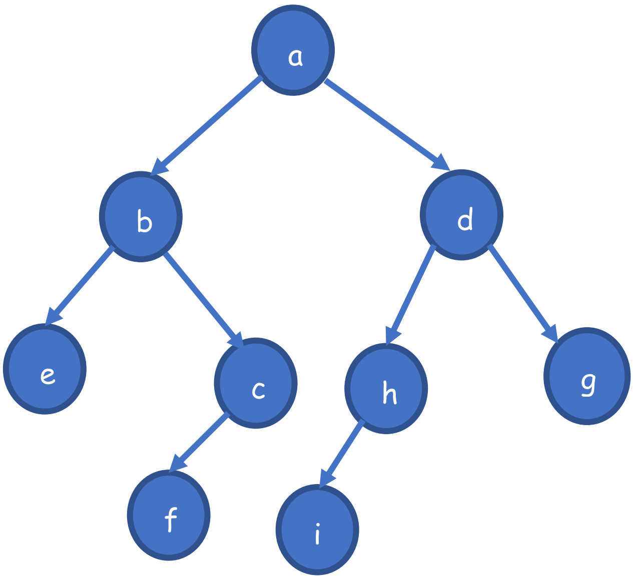
→ Alberio binario completo

Un albero binario è completo se:

1) ha altezza n;

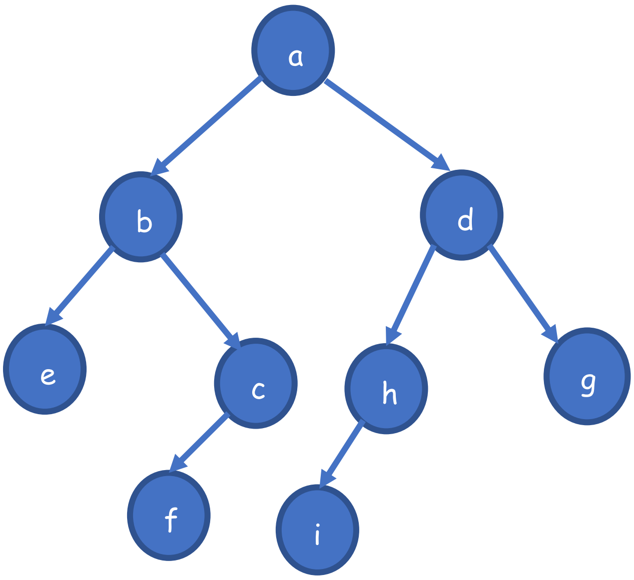
2) ad ogni profondità i, $0 \leq i < n$ ci sono 2^i nodi;

3) l'ultimo livello ` e riempito da sinistra a destra;



→ Albero bilanciato

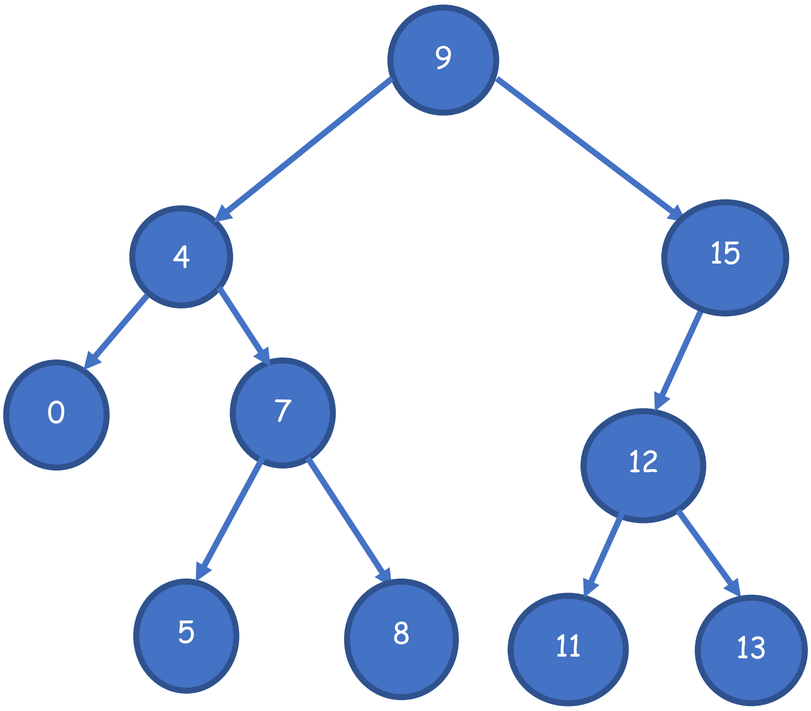
Un albero binario è bilanciato se per ogni nodo v la differenza fra il numero di nodi nell'albero sinistro di v e il numero di nodi nell'albero destro di v è al massimo 1.



→ Albero binario di ricerca

Un albero di ricerca è un albero binario G = (V, E) tale che per ogni nodo z:

1) z ∈ Z;
2) ogni nodo dell'albero sinistro di z è minore a z;
3) ogni nodo dell'albero destro di z è maggiore a z.



→ Attraversamento albero binario

Un attraversamento è un processo che visita tutti i nodi di un albero.
Una enumerazione dei nodi è un attraversamento che elenca ogni nodo esattamente una volta.

→ Tipi di attraversamento

Un attraversamento può essere di due tipologie:

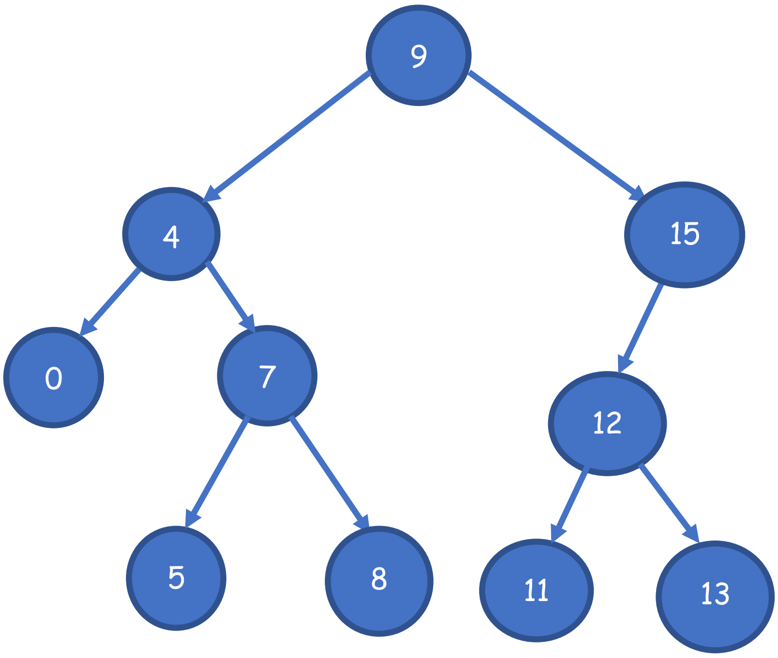
1) in profondità: in cui si esplora ogni ramo dell'albero fino in fondo (figli prima dei fratelli);
2) in ampiezza: in cui si esplora prima i nodi più vicini ala radice (fratelli prima dei figli).

→ Enumerazione in profondità

Ci sono tre tipi diversi di ordini di profondità:

1) L (sinistra);
2) R (destra);
3) V (enumerazione).

I tre ordini di enumerazione in profondità sono:
1) pre – order: si visita prima la radice, poi il sotto albero di sinistra e infine il sotto albero di destra (VLR);
2) in – order: si visita prima il sotto albero sinistro, poi la radice e inifine il sotto albero di destra (LVR);
3) post – order: si visita prima il sotto albero sinistro, poi il sotto albero destro e infine la radice (LRV).



VLR

9	4	0	7	5	8	15	12	11	13

LVR

0	4	5	7	8	9	11	12	13	15

LRV

0	5	8	7	4	11	13	12	15	9

→ Numero di foglie in un albero

Un albero finito ha sempre almeno una foglia e per massimizzare il numero di foglie è necessario avere un albero pieno.
Un albero pieno con n nodi interni ha n+1 foglie.
Il numero di puntatori nulli in un albero binario con n nodi è n + 1.

→ Dimostrazione per induzione

Teorema: un albero pieno con n nodi interni ha n+1 foglie.
Per dimostrare per induzione il teorema, è necessario eseguire tre passi:

1) dimostrare il caso base;
2) specificare l'ipotesi di induzione basata su un numero n;
3) dimostrare il passo di induzione: è vero anche per n+1.

Caso base: un albero non è mai vuoto. Ha almeno 0 nodi interni e 1 foglia.
Ipotesi Induttiva: il risultato è vero per alberi con n nodi interni (hanno n+1 foglie).
Passo Induttivo: dimostrare il caso n + 1; cioè, se un albero pieno ha n + 1 nodi interni, allora ha n + 1 + 1 = n + 2 foglie.

Implementazioni

1) Ricerca di un Valore

```
BTreeSearch(x,k) {
    if (x == NULL OR k == x.key)
        return x;
    if (k<x.key)
        return BTreeSearch(x.left,k);
    else
        return BTreeSearch(x.right,k);
}
```

2) Minimo e massimo di un albero binario

```
BTreeMin(x) {
        if (x = NULL)
                error("Doesn't exist);
        else
                return BtreeMin(x.left);
}

BTreeMax(x) {
        if (x = NULL)
                error("Doesn't exist);
        else
                return BtreeMin(x.right);
}
```

3) Inserimento di un valore

```
Btree_Insert(x,val) {
     if (x == NULL)
        x = z;
     else {
        if (z.key <= x.key)
           Btree_Insert(x.left,val);
        else
           Btree_Insert(x.right,val);
     }
}
```

4) Successivo

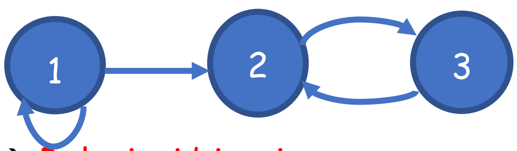
```
BTree_Succ(x) {
   if (x.right != NULL)
        return BTreeMin(x.right);
   else {
        y = x.prev;
        while (y != NULL AND x = y.right) {
                x = y;
                y = y.prev;
        }
        return y;
   }
}
```

14) Grafi: definizioni e caratteristiche

Un grafo è una struttura matematica che è costituita da:
un insieme di nodi (detti vertici);
collegamenti tra vertici che possono essere
- orientati (archi) (grafo orientato);
- non orientati (spigoli) (grafo non orientato);
dati associati a nodi e collegamenti (etichette).

→ Come si rappresentano?

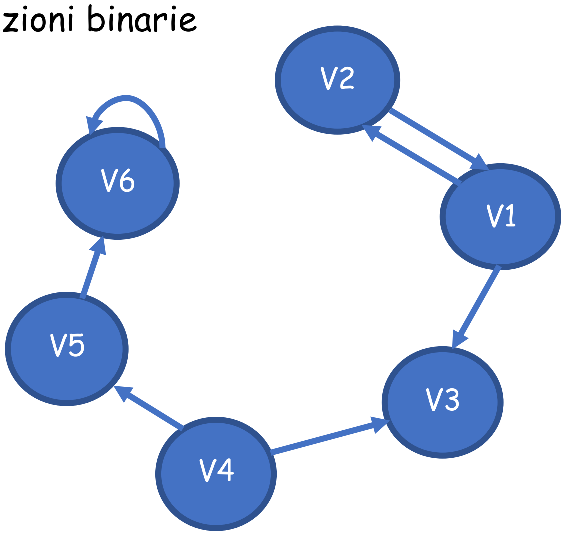
Un grafo viene rappresentato disegnando punti per i nodi, e segmenti o curve per i collegamenti tra i nodi.
Importante:
la posizione o forma dei nodi e dei collegamenti sono irrilevanti.
soltanto la loro esistenza definisce il grafo.



→ Relazioni binarie

I grafi possono rappresentare relazioni binarie
Esempio:
`V = {V1, V2, V3, V4, V5, V6}`

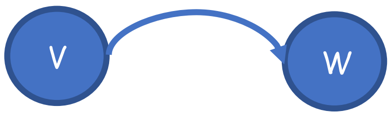
V1	V1
V1	V2
V1	V3
V2	V1



V4	V3
V4	V5
V5	V6
V6	V6

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

→ Terminologie: gradi

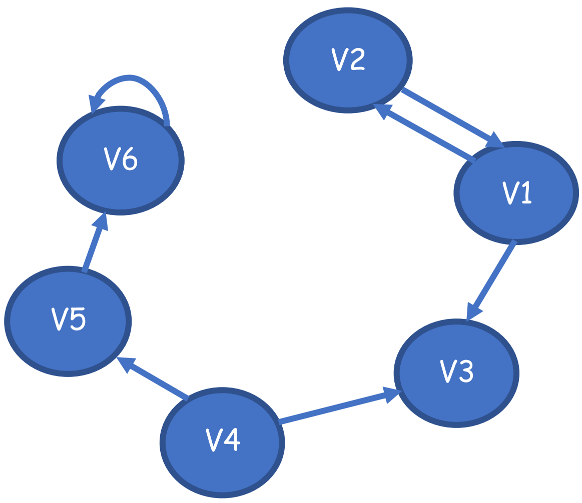


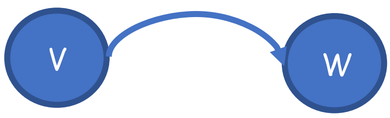
L'arco che connette V e W è detto uscente da V ed entrante in W.
Il numero di archi uscenti dal nodo V è il grado di uscita di V e il numero di archi entranti in V è il grado di ingresso di V.

Un nodo è chiamato:
sorgente: se non ha archi entranti (grado di ingresso è 0);
pozzo: se non ha archi uscenti (grado di uscita 0);
isolato: se non ha archi né uscenti né entranti.

Esempio:

	Grado Uscita	Grado Ingresso	Sorgente	Pozzo
V1	2	1	no	no
V2	1	1	no	si
V3	0	2	no	si
V4	2	0	si	no
V5	1	1	no	no
V6	1	2	no	no





I nodi V e W sono adiacenti se vi è un arco che li connette (qualunque sia la direzione). L'arco è incidente su V e W e il grado di V è il numero di nodi adiacenti a V.

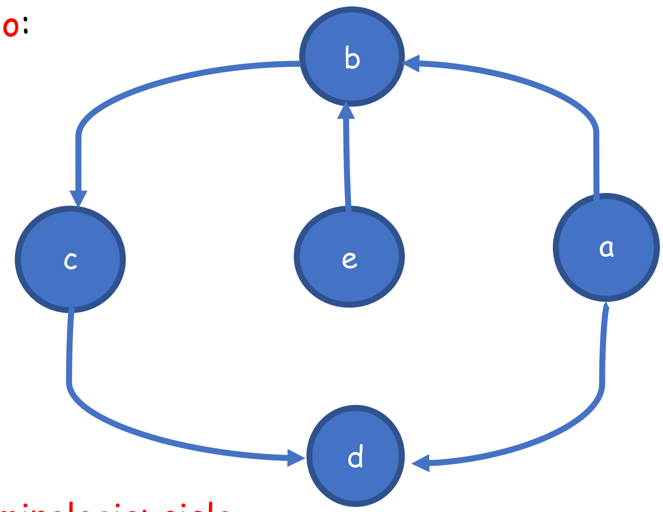
→ Terminologia: cammino

Un cammino è una sequenza finita di nodi <v1, v2, …, vn> tali che per ogni i, 1 <= i <n, esiste un arco uscente da v_i ed entrante in v_{i+1}. Il cammino parte da V a W se v1 = V e vn = W

→ Terminologia: semi cammino

Un semi - cammino è una sequenza finita di nodi <v1, v2, …, vn> tali che per ogni i, 1 <= i <n, esiste un arco che collega v_i e v_{i+1} in direzione arbitraria. La lunghezza di un semi – cammino è il numero di archi che lo compongono (n – 1). Un semi – cammino è semplice se tutti i nodi nella sequenza sono diversi. Un grafo è connesso se esiste sempre un semi – cammino tra due nodi qualsiasi.

Esempio:



Cammini	Lunghezza
<a,d>	1
<a,b,c,d>	3

Semi- Cammini	Lunghezza
<a,b,e>	2
<a,b,c,d,a,b,e>	6

→ Terminologia: ciclo

Un ciclo intorno al nodo V è un cammino tra V e V.

→ Terminologia: semi – ciclo

Un semi – ciclo intorno al nodo V è un semi – cammino tra V e V.

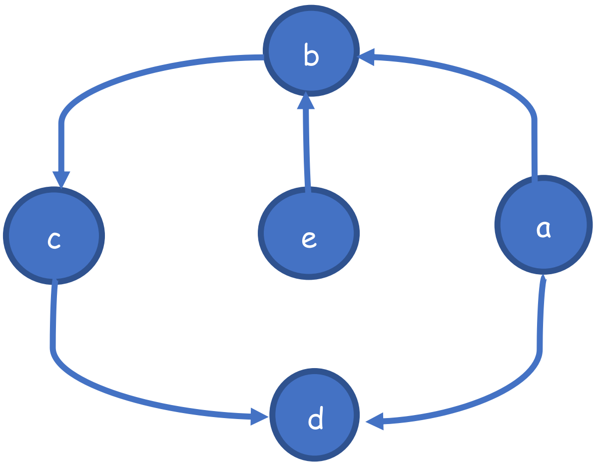
→ Terminologia: cappio

Un cappio intorno a V è un ciclo di lunghezza 1.

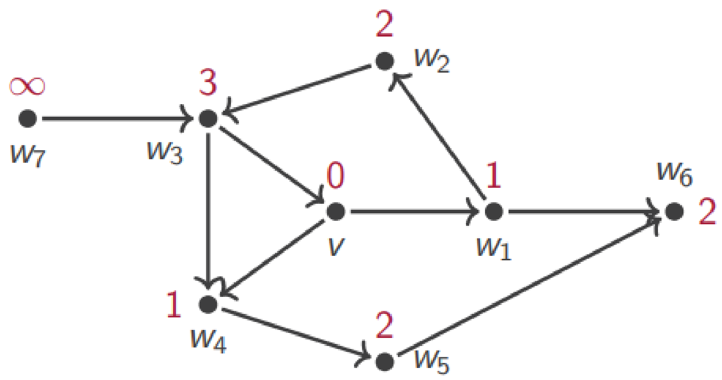
Distanza	Valore
<a,b>	1
<a,d>	1
<b,a>	3
<d,a>	1
<e,d>	3
<a,e>	∞

→ Terminologia: distanza

La distanza da V a W è la lunghezza del cammino più corto tra V e W.
La distanza da V a V è sempre 0.
Se non vi è nessun cammino tra V e W allora la distanza è infinita (∞).
In un grafo ordinato, la distanza da V e W non è sempre equivalente alla distanza da W a V.



→ Trovare le distanze



Le distanze da v ad ogni nodo del grafo.

→ Trovare distanze: Algoritmo

Ricerca in ampiezza delle distanze da v ad ogni nodo.
Inizializzazione:
1) segnare v come visitato con distanza `d(v) = 0`;
2) segnare altri nodi non visitato;

Ciclo: finché ci sono nodi visitato
3) trovare un nodo w visitato con distanza minima `d(w) = n`;
4) segnare w come esplorato;

5) per ogni nodo w' incidente da w: se w' è non visitato, segnare w' come visitato e `d(w') = n+1`

Finalizzazione

ad ogni nodo w non visitato assegnare `d(w) = ∞`

→ Trovare distanze: Algoritmo

Grafo Orientato: è una coppia `G = (V, E)` dove:

1) V è un insieme di nodi;
2) $E \subseteq V \times V$ è una relazione binaria in V.

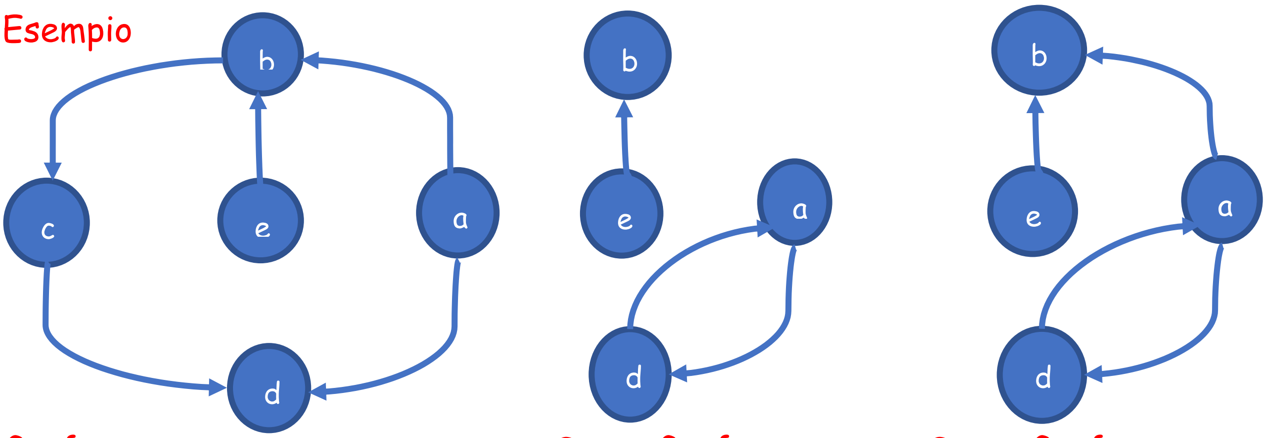
Grafo non orientato: è un grafo orientato dove E è una relazione simmetrica. Gli archi sono rappresentati come coppie non ordinate `(v,w) [(v,w) = (w,v)]`.

→ Sotto grafo

Il grafo $G_1 = (V_1, E_1)$ è un sotto grafo di $G_2 = (V_2, E_2)$ solo se $V_1 \subseteq V_2$ ed $E_1 \subseteq E_2$.
Un sotto grafo si ottiene togliendo nodi e/o archi dal grafo.
Sia $G = (V, E)$ un grafo.
Il sotto grafo indotto da $V' \subseteq V$ è il grafo che ha soltanto archi adiacenti agli elementi di V'.
Formalmente, è il grafo $G = (V', E')$ dove

$$E' = \{\langle v, w\rangle \in E \mid v, w \in V'\}$$

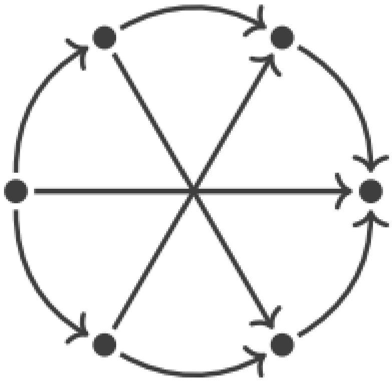
Esempio



Grafo Sotto Grafo Sotto Grafo

→ DAG – Grafo Aciclico Orientato

Un grafo orientato senza cicli, in cui non esiste nessun cammino da un nodo a se stesso.

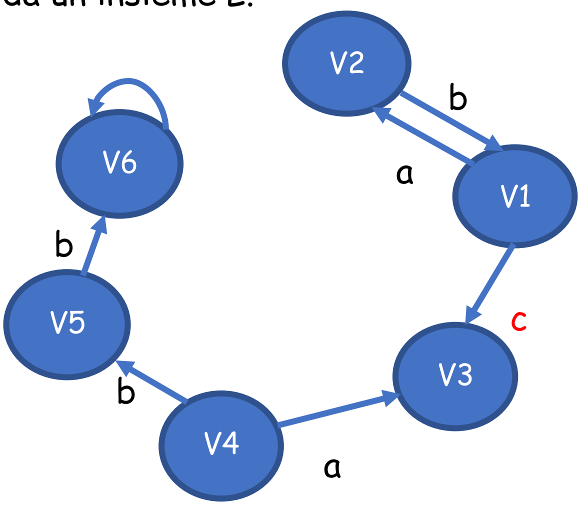


→ Grafo etichettato

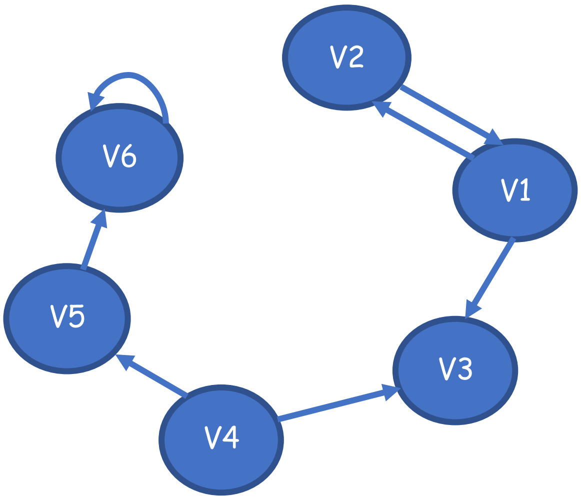
Un grafo etichettato è una tripla `G = (V,E,l)` dove:

1) `(V,E)` è un grafo;

2) `l:E → L` è una funzione totale che associa ad ogni arco e ∈ E una etichetta da un insieme L.



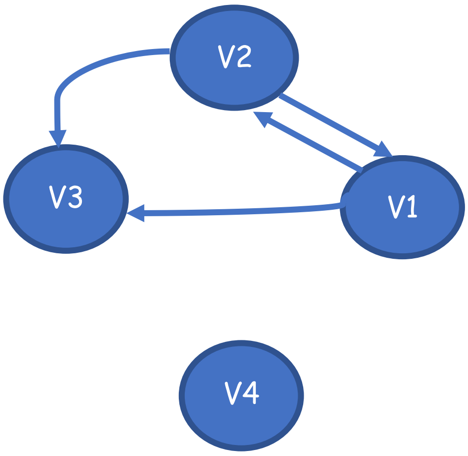
→ Relazioni ternarie



V1	V1	a
V1	V2	b
V1	V3	c
V2	V1	a
V4	V3	a
V4	V5	b
V5	V6	b
V6	V6	c

→ Matrice di adiacenza

La matrice di adiacenza di un grafo G = (V,E) è la matrice booleana della relazione E.



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

La matrice di adiacenza di grafi non orientati è sempre simmetrica.

→ Grafo completo

Un grafo si dice completo collega ogni nodo con tutti gli altri nodi (ma non con se stesso). La matrice di adiacenza ha 0 su tutta la diagonale, ed 1 sulle altre posizioni.

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

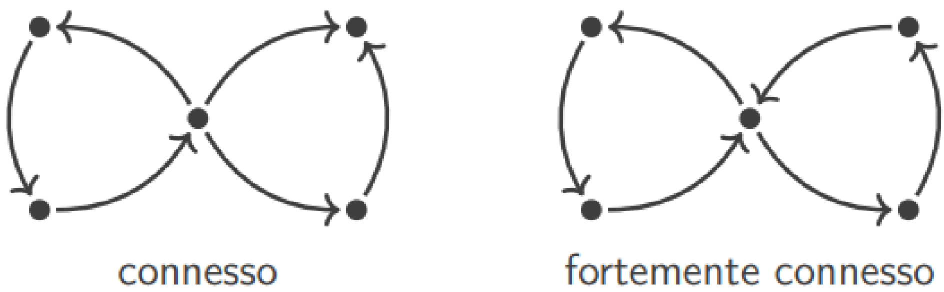
→ Grafo fortemente connesso

Un grafo G si dice fortemente connesso se per ogni due nodi v, w ∈ V esiste un cammino da v a w.

In un grafo fortemente connesso:

1) esiste sempre un ciclo che visita ogni nodo (non necessariamente semplice);
2) non ci sono né sorgenti né pozzi;

Esempio

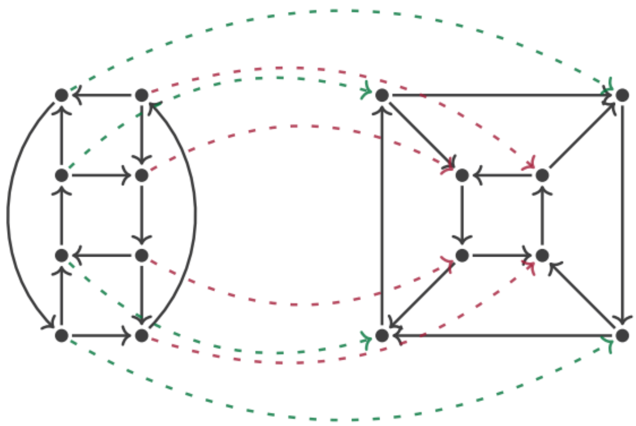


→ Isomorfismi tra grafi

Due grafi `G1 = (V1, E1)` e `G2 = (V2, E2)` sono isomorfi se esiste una funzione biunivoca f : V1 → V2 tale che

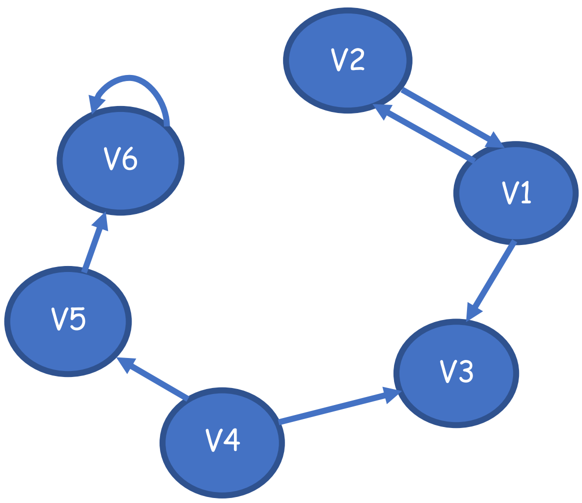
$$\langle v, w\rangle \in E_1 \quad \text{sse} \quad \langle f(v), f(w)\rangle \in E_2$$

L'isomorfismo f mantiene la struttura del grafo G1, ma sostituisce i nomi dei vertici per quelli di G2. Due grafi isomorfi sono in realtà lo stesso grafo con i nodi rinominati.



Algoritmi su Grafi

DFS – Visita



```
DFS_Visit(G,V) {
        v.color := GRIGIO;
        time := time + 1;
        v.discovery := time;
        foreach (v app to G.Adj[v]) {
                if (v.color = BIANCO) {
                        v.pi := v;
                        DFS_Visit(G,V);
                }
        }
        v.color := NERO;
        time := time + 1;
        v.f = time;
}
```

```
DFS(G) {
        foreach(v app to G.V) {
             v.color := BIANCO;
             v.pi := NULL;
        }
        time := 0;
        foreach (v app to G.V) {
             if (v.color = BIANCO)
                DFS_visit(G,V);
        }
}
```

Valutazione tempi

$T(n) = O(|V| + |E|)$ → implementato mediante le liste di adiacenza

$T(n) = O(|V|^2)$ → implementato mediante le matrici

Classificazione degli archi

Gli archi del grafo si distinguono in tre categorie:

→ Tree Edge: archi appartenenti alla foresta DFS;

→ Back Edge: archi non appartenenti alla foresta DFS, che connettono un nodo v al suo antenato w nell'albero DFS;

→ Forward Edge: archi non appartenenti alla foresta DFS, che connettono un nodo v al suo antenato w nell'albero DFS;

→ Cross Edge: tutti gli altri archi.

Ordine topologico

```
TopSort(G) {
    DFS(G)
    ordina in senso decrescente per finishing time i vertici
    return;
}
```