

Algoritmi e strutture dati

Analisi degli algoritmi

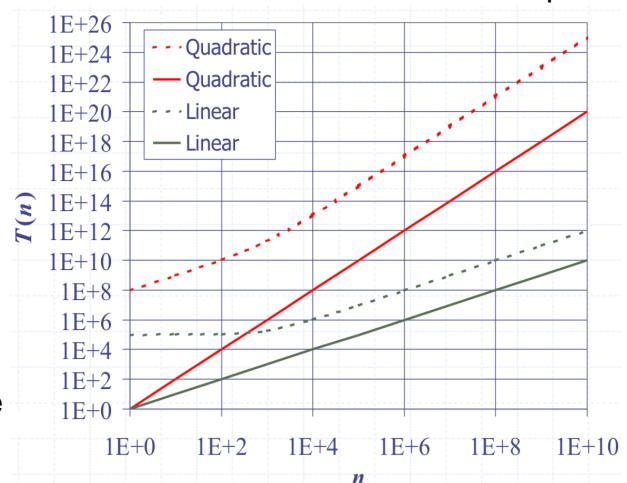
Un algoritmo è una procedura specificata passo passo per risolvere un problema in una quantità di tempo finita.

La sua implementazione può essere complessa ed i tempi di esecuzione reali sono influenzati dalla dimensione dell'input e dalla piattaforma su cui è utilizzato, per questo l'analisi si effettua a livello teorico su uno pseudocodice ad alto livello.

La valutazione del costo computazionale teorico viene fatta in modo asintotico, usando la notazione generica $T(N)$ per indicare la funzione matematica che indica la relazione esistente tra il numero di operazioni di un programma e la dimensione N dei dati di input.

Il calcolo viene eseguito assumendo che le operazioni elementari abbiano costo unitario, quindi le uniche operazioni che influenzano realmente il costo sono quelle sui cicli, direttamente influenzate dalla quantità di dati in input.

Si effettua seguendo la filosofia del **worst case**, ovvero si studiano il numero di passi da effettuare nel caso peggiore, così si ha sempre il costo massimo che può scaturire.



Il costo totale è dato da un'equazione polinomiale formata dalla somma di tutti i singoli costi, il costo asintotico finale è però dato solamente dall'istruzione dominante (ovvero quella con costo maggiore) tramite la notazione $O(\dots)$ e trascurando i fattori costanti.

Notazioni

- O -grande => upper bound (al massimo come), $f(n)$ è $O(g(n))$ se esistono 2 costanti c e n_0 t.c.
$$f(n) \leq c * g(n) \quad \forall n \geq n_0$$
- Ω -grande => lower bound (almeno come), $f(n)$ è $\Omega(g(n))$ se esistono 2 costanti c e n_0 t.c.
$$f(n) \geq c * g(n) \quad \forall n \geq n_0$$
- Θ -grande => esattamente come, $f(n)$ è $\Theta(g(n))$ se esistono 3 costanti c' , c'' e n_0 t.c.
$$c' * g(n) \leq f(n) \leq c'' * g(n) \quad \forall n \geq n_0$$

OSS Nel caso in cui l'input (riportato nella funzione di costo) sia un numero, e non un vettore, la sua dimensione è data dalla sua rappresentazione binaria

$$z = \log_2(n) \Rightarrow 2^z = n$$

OSS Vale sempre la proprietà
complessità temporale \geq complessità spaziale

OSS Funzioni logaritmiche uguali con basi differenti sono equivalenti

OSS Un algoritmo si dice **in-place** quando non necessita di utilizzare strutture di memoria ausiliarie, ovvero la memoria necessaria per il funzionamento dell'algoritmo non cambia al crescere dell'input.

ATTENZIONE Gli algoritmi ricorsivi non posso mai essere in place a causa dell'aumentare della stack ad ogni chiamata ricorsiva, l'eliminazione della ricorsione può favorire la velocità del codice ma aggiunge spesso complessità e può necessitare dell'utilizzo di una pila per sostituire che compensi la stack

Ex. 1

```

Algorithm prefixAverages1(X, n)
Input array X di n interi
Output array A delle medie dei prefissi di X # operazioni
A  $\leftarrow$  nuovo array di n interi      n
for i  $\leftarrow$  0 to n - 1 do      n
    s  $\leftarrow$  X[0] n
    for j  $\leftarrow$  1 to i do 1 + 2 + ... + (n - 1)
        s  $\leftarrow$  s + X[j] 1 + 2 + ... + (n - 1)
    A[i]  $\leftarrow$  s / (i + 1) n
return A 1

```

Considerando la seguente proprietà:

$$\sum_{i=1}^n i = \frac{n(n-1)}{2} \Rightarrow n^2$$

Si ha quindi che il costo asintotico dell'algoritmo è $O(n^2)$

OSS altra proprietà importante è la serie geometrica

$$\sum_{i=0}^n q^i = \frac{q^{n+1} - 1}{q - 1}$$

Ex. 2**Algorithm** BinarySum(A, i, n):**Input:** Un array A e interi i ed n **Output:** La somma di n interi in A iniziando dall'indice i **if** $n = 1$ **then****return** $A[i]$ **return** $\text{BinarySum}(A, i, n/2) + \text{BinarySum}(A, i + n/2, n/2)$

Avendo una funzione ricorsiva bisogna utilizzare le equazioni di ricorrenza, in questo caso

$$\begin{cases} T(1) = c' \\ T(n) = 2T\left(\frac{n}{2}\right) + c \end{cases}$$

Nel caso $T\left(\frac{n}{2}\right)$ si ha quindi $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c$

ovvero $T(n) = 2[2T\left(\frac{n}{4}\right) + c] + c = 4T\left(\frac{n}{4}\right) + 3c$

Iterando il procedimento si arriva a $T(n) = 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)c$

considerando che $\frac{n}{2^i} = 1 \Leftrightarrow n = 2^i$ si ha che $i = \log_2(n)$

Quindi la funzione di costo totale risulta $2^{\log_2(n)} + (2^{\log_2(n)} - 1)c$

ma $2^{\log_2(n)} = n$ quindi il costo asintotico risulta $O(n)$

Alberi

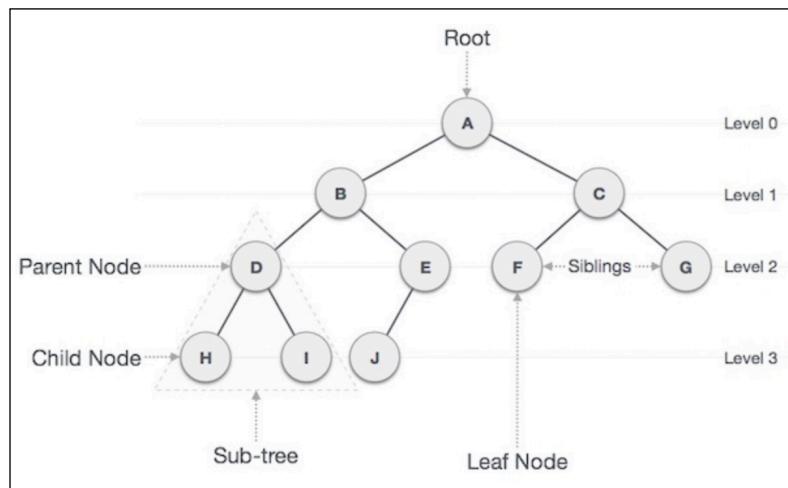
Struttura gerarchica a stadi.

Definizione ricorsiva:

- Ogni nodo è un albero;
- Ogni albero ha un qualsiasi numero di sottoalberi;

Altre definizioni:

- *Profondità nodo* => numero di antenati
- *Altezza albero* => massima profondità (dell'ultima foglia)



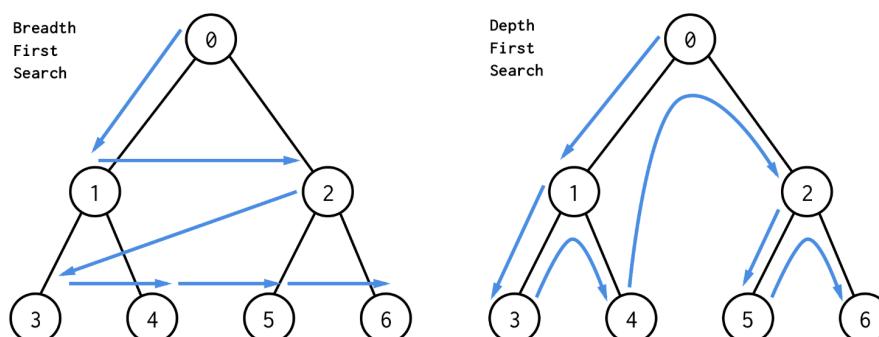
Ogni nodo ha un solo arco entrante ed n archi uscenti.

Un albero con k nodi ha k-1 archi (poiché radice non ha arco genitore).

<ul style="list-style-type: none"> ◆ Si usano le Position per astrarre i nodi <ul style="list-style-type: none"> ▪ Position: solo metodo <code>Object element();</code> 	<ul style="list-style-type: none"> ◆ Metodi di query: <ul style="list-style-type: none"> ▪ boolean <code>isInternal(p)</code> ▪ boolean <code>isExternal(p)</code> ▪ boolean <code>isRoot(p)</code>
<ul style="list-style-type: none"> ◆ Metodi generici: <ul style="list-style-type: none"> ▪ integer <code>size()</code> ▪ boolean <code>isEmpty()</code> ▪ Iterator <code>elements()</code> ▪ Iterator <code>positions()</code> 	<ul style="list-style-type: none"> ◆ Metodi di aggiornamento : <ul style="list-style-type: none"> ▪ object <code>replace (p, o)</code>
<ul style="list-style-type: none"> ◆ Metodi di accesso: <ul style="list-style-type: none"> ▪ position <code>root()</code> ▪ position <code>parent(p)</code> ▪ positionIterator <code>children(p)</code> 	<ul style="list-style-type: none"> ◆ Ulteriori metodi di aggiornamento possono essere definiti attraverso strutture dati che implementano il TDA Tree

La visita di un albero si può effettuare principalmente in 2 modi:

- **BFS** (breadth first search)¹ => la visita avviene prima in larghezza, ovvero i fratelli vengono visitati prima dei figli
- **DFS** (depth first search) => la visita avviene prima in profondità, ovvero i figli vengono visitati prima dei fratelli



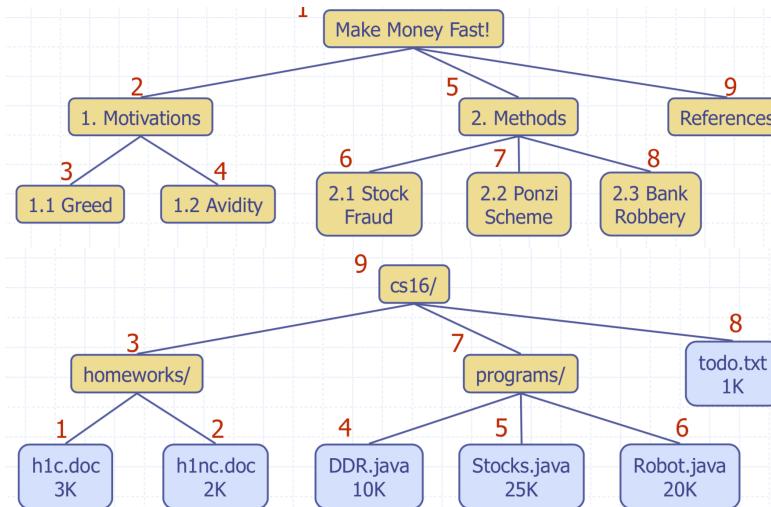
¹ BFS necessita di struttura d'appoggio per salvare i nodi dei quali bisogna ancora visitare i figli

Inoltre la DFS può essere implementata in 2 modi differenti:

- **Pre-order** => ogni nodo viene visitato prima dei suoi discendenti

- **Post-order** => ogni nodo viene visitato dopo dei suoi discendenti

Entrambi hanno **costo $O(n)$** , il post order, ad esempio, si usa nel caso di un file System per calcolare la dimensione della cartella tramite la dimensione dei file che contiene.



Algorithm *preOrder(v)*

visit(v)

for each child w of v
preorder(w)

Algorithm *postOrder(v)*

for each child w of v
postOrder(w)
visit(v)

Gli alberi binari

Albero con al massimo 2 figli, distinti in figlio sinistro e figlio destro.

L'albero si dice completo se ha esattamente 2 figli per nodo, quasi completo se tutti i livelli tranne l'ultimo rispettano la proprietà di completezza.

Alcune proprietà (alcune richiedono completezza):

- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
- $h \geq \log_2 e$ (vedere teorema heap)
- $h \geq \log_2(n + 1) - 1$

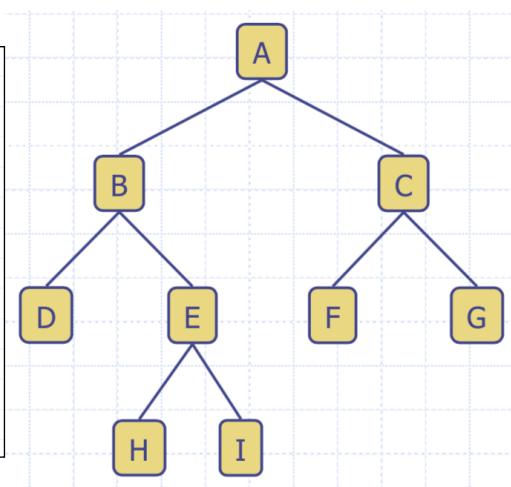
$n \Rightarrow$ numero di nodi
 $e \Rightarrow$ numero di nodi esterni
 $i \Rightarrow$ numero di nodi interni
 $h \Rightarrow$ altezza

◆ Il TDA BinaryTree estende il TDA Tree, cioè eredita tutti i metodi del TDA Tree

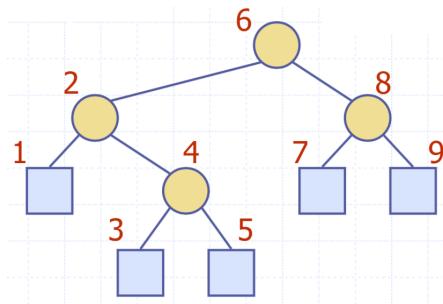
◆ Metodi addizionali:

- **position left(p)**
- **position right(p)**
- **boolean hasLeft(p)**
- **boolean hasRight(p)**

◆ I metodi di aggiornamento possono essere definiti attraverso le strutture dati che implementano il TDA Tree



Questi introducono un nuovo tipo di DFS, la visita in ordine, comoda ad esempio per stampare un expression tree (albero a cui è associata un'espressione aritmetica).



Algorithm *inOrder(v)*

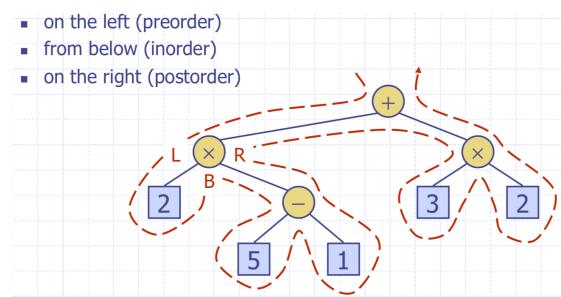
```

if hasLeft (v)
    inOrder (left (v))
    visit (v)
if hasRight (v)
    inOrder (right (v))

```

Operazioni differenti richiedono un tipo differente di visita (ex. Valutazione espressione e stampa espressione), si necessiterebbe quindi di scorrere più volte l'albero con un solo algoritmo per eseguire tutte le operazioni necessarie.

Invece di far questo, si può usare il **cammino Euleriano**, che unisce tutti e 3 tipi di DFS inserendo una visita prima di ogni nodo, dopo la visita del figlio sinistro e dopo la visita del figlio destro.



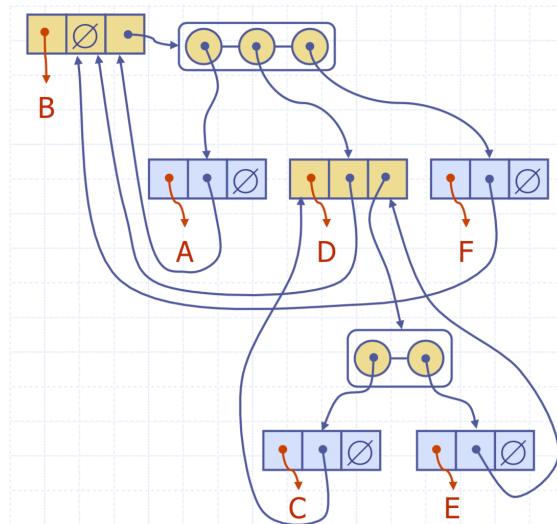
Implementazioni alberi

Generalmente implementati tramite una struttura collegata per alberi, ogni nodo è rappresentato da un oggetto node, composto da:

- Elemento
- Puntatore nodo genitore
- Sequenza puntatori nodi figli

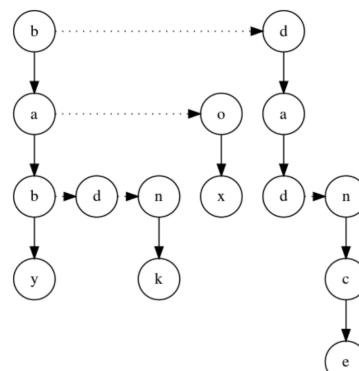
Nel caso di albero binario invece di avere una lista di puntatori ai nodi figli, ogni nodo è formato da:

- Elemento
- Puntatore nodo figlio sinistro
- Puntatore nodo figlio destro
- Puntatore nodo genitore
(opzionale, generalmente meglio di no)



Altrimenti si possono implementare tramite una struttura minimale nel quale ogni nodo ha solo:

- Elemento
- Puntatore al primo figlio
- Puntatore al fratello



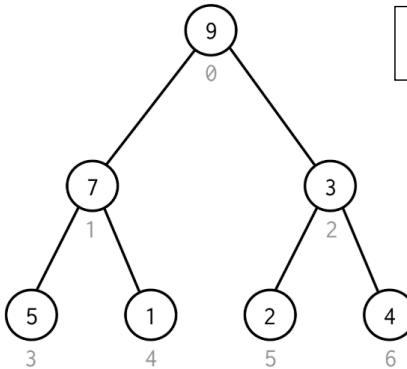
Nel caso di alberi binari è inoltre possibile implementare l'albero come un semplice array nel quale la posizione è data dagli indici, definito come:

- $\text{Rank}(\text{root}) = 1$
- figlio sinistro di node = $2 * \text{rank}(\text{node})$
- figlio destro di node = $2 * \text{rank}(\text{node}) + 1$

$$\text{get_parent_index} \Rightarrow (i-1)/2$$

Ottobre partendo da 0:

- $\text{Rank}(\text{root}) = 0$
- figlio sinistro di node = $2 * \text{rank}(\text{node}) + 1$
- figlio destro di node = $2 * \text{rank}(\text{node}) + 2$



Code di priorità (queue)

Collezione di elementi che generalmente segue ordinamento impartito dal valore delle chiavi, ma che in certi casi può essere basata sulla politica FIFO.

Si ha quindi che ogni entry della cosa è formata da una coppia **<key, value>**.
OSS Le pile (stack) usano invece una politica LIFO.

ATTENZIONE possono esistere 2 elementi con la stessa chiave

Metodi principali:

- Insert (enqueue) => inserisce elemento nella coda (accodamento)
- Remove (dequeue) => rimuove elemento dalla coda

Inoltre nel caso in cui non sia possibile associare direttamente un ordine totale alle chiavi bisogna implementare il metodo compare che le confronta e ne determina l'ordine.

Per quanto riguarda le entry si hanno i metodi:

- key => che restituisce valore della chiave
- Value => che restituisce il valore contenuto

La coda può essere implementata secondo 2 filosofie:

- Non ordinata, ovvero gli elementi vengono inseriti in modo casuale e rimossi ordinati
 - Insert => $O(1)$
 - Remove => $O(n)$
- Ordinata, ovvero gli elementi vengono ordinati già durante l'inserimento
 - Insert => $O(n)$
 - Remove => $O(1)$

Algorithm **PQ-Sort(S, C)**

Input sequence S , comparator C for the elements of S

Output sequence S sorted in increasing order according to C

$P \leftarrow$ priority queue with comparator C

while $\neg S.\text{isEmpty}()$

$e \leftarrow S.\text{removeFirst}()$

$P.\text{insert}(e, 0)$

while $\neg P.\text{isEmpty}()$

$e \leftarrow P.\text{removeMin}().\text{key}()$

$S.\text{insertLast}(e)$

Usando il principio del PQ-sort (priority queue sort) si hanno 2 algoritmi di ordinamento:

- **selection-sort** => la coda di priorità viene implementata tramite una lista non ordinata, gli elementi verranno riordinati al momento della rimozione
- **Insertion-sort** => la coda di priorità viene implementata tramite una lista ordinata, gli elementi vengono quindi ordinati al momento dell'inserimento e rimossi già ordinati

In entrambi i casi si ha tempo di esecuzione $O(n^2)$, insertion-sort è però considerato migliore poiché, in caso di lista ordinata, si può considerare il best case con costo $\Omega(n)$.

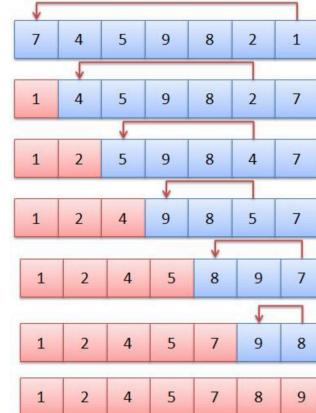
Selection-sort		Insertion-sort			
	Sequenza S	Coda di priorità P			
Input:	(7,4,8,2,5,3,9)	()	Input:	(7,4,8,2,5,3,9)	()
Phase 1			Phase 1		
(a)	(4,8,2,5,3,9)	(7)	(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)	(b)	(8,2,5,3,9)	(4,7)
..		(c)	(2,5,3,9)	(4,7,8)
.	.		(d)	(5,3,9)	(2,4,7,8)
(g)	()	(7,4,8,2,5,3,9)	(e)	(3,9)	(2,4,5,7,8)
Phase 2			(f)	(9)	(2,3,4,5,7,8)
(a)	(2)	(7,4,8,5,3,9)	(g)	()	(2,3,4,5,7,8,9)
(b)	(2,3)	(7,4,8,5,9)			
(c)	(2,3,4)	(7,8,5,9)			
(d)	(2,3,4,5)	(7,8,9)			
(e)	(2,3,4,5,7)	(8,9)			
(f)	(2,3,4,5,7,8)	(9)			
(g)	(2,3,4,5,7,8,9)	()			
		Code di Priorità			

OSS è possibile realizzarli **in-place** dividendo l'array in una prima parte "ordered" ed una seconda "unordered".

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```



Esegue un ciclo sull'array da ordinare per ogni elemento dell'array totale, ognuno dei quali trova il minimo e lo posiziona alla fine dell'array ordinato

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```



Ad ogni iterazione del ciclo principale l'algoritmo salva il valore della casella corrispondente in una variabile ausiliaria, dopodiché inizia un ciclo² per trovare la posizione corrispondente al valore corrente.

Ad ogni iterazione shifta a destra l'elemento che risulta più grande del valore corrente

² partendo dall'indice in uso ed andando verso sinistra (permette O(n) con array già ordinato)

Heap

Albero binario implementato tramite array che permette la realizzazione di code di priorità efficienti, ad ogni entry è quindi associata una coppia **<key, value>**.

L'heap ha le seguenti proprietà:

- Per ogni nodo si ha $key(v) \geq key(parent(v))$ nel caso di min-heap, il contrario nel caso di max-heap
- Data h altezza dell'albero, l'heap è un albero completo almeno fino a profondità $h-1$, nel caso in cui l'ultimo livello non sia completo questo è bilanciato a sinistra

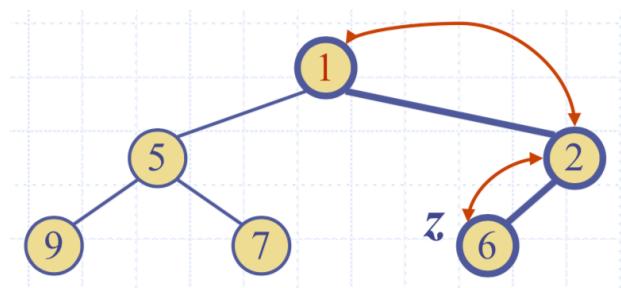
Teorema Un heap che memorizza n chiavi ha altezza **$O(\log(n))$**

Poiché essendoci 2^i chiavi a profondità $i = 0, \dots, h-1$ ed almeno una chiave a profondità h , si ha $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$.

Quindi $n \geq 2^h$, ovvero $h \leq \log(n)$

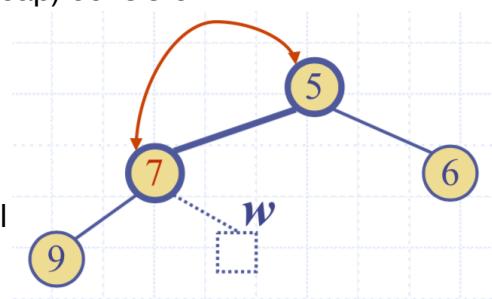
l'inserimento di una nuova entry consiste in:

1. Trovare l'ultimo nodo disponibile z
2. Memorizzare l'entry in z
3. Eseguire **Upheap** che riordina l'heap scambiando la nuova chiave lungo tutto il cammino ascendente dal nodo inserito



La rimozione del minimo (o massimo nel caso di max-heap) consiste in:

1. Sostituire radice con l'ultima entry dell'heap
2. Rimuovere ultimo nodo
3. Eseguire **Downheap** (heapify) che riordina l'heap scambiando la nuova root lungo tutto il cammino descendente fino a diventare una foglia oppure avere che entrambi i figli sono maggiori (o minori nel caso di max-heap)



Siccome l'heap mantiene sempre altezza $O(\log(n))$ entrambe le operazioni di inserimento e rimozione hanno costo $O(\log(n))$.

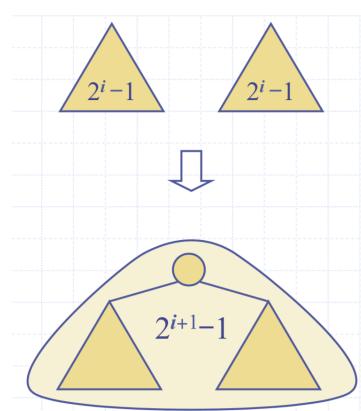
ATTENZIONE la ricerca di un nodo mantiene costo $O(n)$.

È quindi possibile formulare un algoritmo, chiamato **heap-sort** (vedere esercitazioni), che permette l'ordinamento dell'heap con un costo $O(n \log(n))$, implementabile in-place.

Partendo dal presupposto che un heap è fondamentalmente un array e che 2 heap della stessa altezza possono essere fusi utilizzando un solo nodo aggiuntivo in cima (eseguendo un downheap) è possibile effettuare una costruzione bottom-up dell'heap come $\log(n)$ unioni di sottoalberi.

1. Si posizionano tutte le entry dell'ultimo livello
2. Si posiziona il secondo livello che permette di unire i sottoalberi dell'ultimo livello
3. Si procede ricorsivamente fino alla root

OSS La costruzione bottom-up è più veloce di n inserimenti.



ATTENZIONE l'implementazione tramite array limita la grandezza dell'heap, quindi ad ogni nuovo inserimento fuori della dimensione massima si effettua un analisi ammortizzata raddoppiando le dimensioni dell'array per i prossimi inserimenti.

Mappe

Collezione di elementi (archivio), ogni entry è formata da una coppia **<key, value>**.

ATTENZIONE Non sono ammessi elementi multipli nella mappa.

◆ Metodi del TDA mappa:

- **get(k)**: se la mappa M ha una entry con chiave k, restituisce il valore ad essa associato, altrimenti restituisce null
- **put(k, v)**: inserisce nella mappa M la entry (k, v); se la chiave k non è già in M restituisce il risultato null; altrimenti, se w è il valore già presente associato a k, sostituisce w con v, restituendo come risultato proprio w
- **remove(k)**: se la mappa M ha una entry con chiave k, rimuove tale entry da M e restituisce il valore associato a k; altrimenti, restituisce null
- **size(), isEmpty()**
- **keys()**: restituisce una collection iterabile contenente tutte le chiavi contenute in M (keys().iterator() restituisce un iteratore alle chiavi)
- **values()**: restituisce una collection iterabile contenente tutti i valori associati alle chiavi contenute in M (values().iterator() restituisce un iteratore ai valori)
- **entries()**: restituisce una collection iterabile contenente tutte le entry chiave-valore contenute in M (entries().iterator() restituisce un iteratore alle entry)

I modi più semplici (ma non efficienti) per implementare le mappe sono:

	get(k)	put(k, v)	remove(k)
Lista non ordinata	O(n)	O(n)	O(n)
Lista ordinata	O(n)	O(n)	O(n)
Array non ordinato	O(n)	O(n)	O(n)
Array ordinato	O(log(n)) ***	O(n)	O(n)

*** L'algoritmo usato è la ricerca binaria

I costi sono così alti a causa della condizione di non ripetizione delle chiavi, che necessita la ricerca della stessa prima di ogni altra operazione.

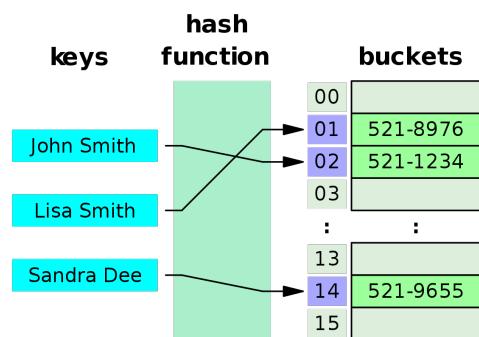
Le operazioni put e remove su array ordinato, pur potendo usufruire della ricerca binaria per la get, hanno costo $O(n)$ dovendo shiftare ogni elemento dopo l'elemento inserito o rimosso.

Tabelle HASH

Strutture dati tramite le quali è possibile implementare mappe (o dizionari), basate su:

- Una **funzione hash** h che trasforma (mappa) delle chiavi su interi appartamenti ad un intervallo prefissato $[0, N-1]$
 $h(x)$ è detto codice hash della chiave x
- Una **tabella hash**, ovvero un array di dimensione N che data la funzione hash corrispondente permette di posizionare le chiavi su di esso (tramite shuffling).

Si ha quindi che ogni chiave viene associata univocamente ad un indice dell'array, portando teoricamente tutte le operazioni ad un costo $\Theta(1)$.
 OSS $O(n)$ nel caso peggiore, che però è molto raro.



La funzione hash $h(x) = h_2(h_1(x))$ è suddivisa a sua volta in:

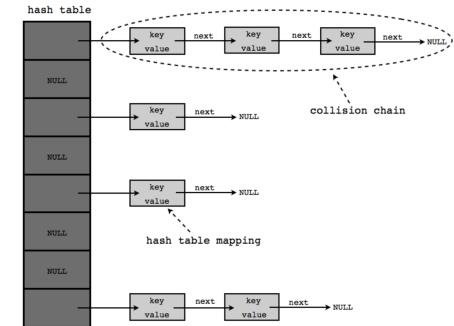
- Codice hash h_1 (chiavi \rightarrow interi) \Rightarrow associa un valore intero ad ogni chiave
- **Funzione di compressione³** h_2 (interi $\rightarrow [0, N-1]$) \Rightarrow associa una casella ad ogni intero, necessaria poiché il h_1 può generare un numero qualsiasi non compreso nell'intervallo OSS il costo della funzione di hash non deve dipendere dalla dimensione dell'array.

Gestione delle collisioni

Avendo che la funzione hash può associare diverse chiavi allo stesso intero, si possono verificare collisioni, ovvero stesse chiavi si vogliono mappare nella stessa casella, in questo caso si possono usare diversi "metodi per la gestione delle collisioni":

- **Separate checking** \Rightarrow consiste nell'associare ad ogni cella dell'array una lista collegata sulla quale salvare le chiavi associate alla stessa casella.

Non è adatta per strutture di grandi dimensioni e richiede una ricerca della chiave sulla lista ad ogni inserimento per evitare i duplicati.



- **Open addressing** \Rightarrow se la casella è già occupata si inserisce la nuova entry in un'altra casella dell'array, questo evita di usare strutture extra ma può rendere necessario scandire tutto l'array per trovare l'elemento ad ogni ricerca (o rimozione)⁴. Per la scelta della nuova casella si possono usare 3 metodi:

- **Linear probing** \Rightarrow se la casella è già occupata la nuova entry viene posizionata nella prima casella successiva disponibile, ripartendo dalla prima se si arriva alla fine.

Soffre del problema di **clustering primario**, aumentando l'agglomerazione in caselle vicine si ha sempre più probabilità di collisione, poiché alla probabilità di collisione di una casella si aggiunge anche quella di tutte le caselle vicine.

Buono per tabelle con fattore di carica non superiore al 30%.

- **Quadratic probing** \Rightarrow come linear probing ma si aumenta la distanza tra le caselle, risolve clustering primario ma non secondario.
- **Hashing doppio** \Rightarrow se la casella è già occupata la nuova entry viene posizionata utilizzando una funzione hash secondaria, risolvendo i problemi di clustering ed aumentando le prestazioni della tabella per grossi carichi.

Come contro il calcolo di 2 funzioni hash è più oneroso rispetto ad uno shift.

Keys	Indices	Key-value pairs (records)
John Smith	0	Lisa Smith +1-555-8976
Lisa Smith	1	
Sam Doe	872	John Smith +1-555-1234
	873	Sandra Dee +1-555-9655
Sandra Dee	874	
	998	Sam Doe +1-555-5030
	999	

Le prestazioni delle tabelle hash sono molto variabili e dipendono dal **load Factor α** , ovvero dalla quantità di caselle occupate (n) rispetto alle caselle totali della tabella (N),

$$\text{che non dovrebbe superare } \alpha = \frac{n}{N} \leq \frac{1}{2}$$

si cerca quindi di avere sempre una tabella doppia rispetto alla quantità di entry previste.

ATTENZIONE prestazioni pessime in caso di range query (non ottimale per dizionari)

³ si usa spesso la funzione $h_2(x) = x \bmod(N)$

⁴ Questo succede poiché ad ogni rimozione si ha la cancellazione dell'elemento senza verificare che non siano avvenute collisioni, effettuando la nuova ricerca si potrebbe quindi avere che l'elemento si trovi shiftato di alcune posizioni anche se la casella che gli spetterebbe è vuota. Per diminuire il problema si possono inserire marcatori che indicano l'effettuata rimozione di un elemento dalla casella, evitando ricerche inutili da caselle che sono sempre state vuote

Dizionari

Collezione di elementi, ogni entry è formata da una coppia **<key, value>**.

Generalizzazione dell'idea di mappa, ma al contrario di esse, qui sono ammessi elementi duplicati (con la stessa chiave).

- ◆ Metodi del TdA dizionario: :
- **find(*k*)**: se il dizionario ha una entry con chiave *k* allora la restituisce, altrimenti restituisce null
 - **findAll(*k*)**: restituisce un Iterable per tutte le entry aventi chiave *k*
 - **insert(*k, o*)**: inserisce e restituisce la entry (*k, o*)
 - **remove(*e*)**: rimuove la entry *e*, restituendola, o restituisce null se non presente
 - **entries()**: restituisce un insieme iterabile di tutte le entry
 - **size(), isEmpty()**

Anche in questo caso i modi più semplici per implementare le mappe sono:

	get(x)	put(x)	remove(x)	findAll()
Lista non ordinata	O(n)	O(1)	O(n)	O(n)
Lista ordinata	O(n)	O(n)	O(n)	O(n)
Array non ordinato	O(n)	O(1)	O(n)	O(n)
Array ordinato	O(log(n))	O(n)	O(n)	O(log(n) + k) ***

*** Dipende dalla dimensione in output del dizionario, fino a un massimo di O(n).

Gli inserimenti non necessitano più della ricerca dell'elemento quindi sono immediati. I dizionari ad array ordinati secondo un ordine totale portano il costo di ricerca da O(*n*) a O(log(*n*)) ed ampliano il TdA aggiungendo anche le operazioni:

- ◆ Nuove operazioni:
- **first()**: prima entry secondo l'ordinamento definito
 - **last()**: ultima entry secondo l'ordinamento definito
 - **successors(*k*)**: iteratore sulle entry con chiavi non inferiori a *k* (ordine crescente)
 - **predecessors(*k*)**: iteratore sulle entry con chiavi non superiori a *k* (ordine decrescente)

Permettono inoltre di effettuare la ricerca binaria degli elementi, ovvero l'array viene ripetutamente diviso in 2 metà per poi sceglierne una (sezione minore o maggiore) e continuare ricorsivamente fino a trovare l'elemento.

Anche in questo caso è possibile l'implementazione tramite hash table, solitamente con separate checking, ma si preferisce quella tramite binary search tree (per findAll).

BST (Binary Search Tree)

Alberi binari con in più le seguenti caratteristiche:

- Ogni entry è formata da una coppia **<key, value>**
- Avendo che **v** è il nodo genitore, **u** il figlio sinistro e **w** il figlio destro, si ha che questi rispettano la seguente proprietà: $key(u) \leq key(v) \leq key(w)$

Per verificare tale proprietà si può effettuare una visita *inorder* dell'albero, se gli elementi sono ordinati l'albero è un BST.

Questo permette di mantenere i costi delle operazioni legati all'altezza dell'albero, di conseguenza si ha che tutte le operazioni di ricerca, inserimento e rimozione hanno costo **$O(h)$** , **findAll** ha costo $O(h+k)$ dove k è il numero delle chiavi.

Nel caso peggiore si ha un albero totalmente sbilanciato e l'altezza coincide con il numero di chiavi e quindi si ha $O(n)$, ma nel caso migliore, le operazioni, legate all'altezza, raggiungono costo $O(\log(n))$.

Operazioni

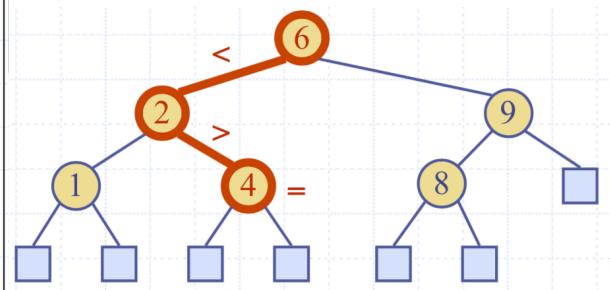
Data la proprietà dei BST si può effettuare la ricerca seguendo il principio della ricerca binaria su array, in questo caso però si effettua tracciando un percorso che parte dalla root ed arriva fino alle foglie, stabilendo ad ogni bivio quale strada prendere tra il valore maggiore di destra e quello minore di sinistra.

Algorithm *TreeSearch(k, v)*

```

if T.isExternal(v)
    return v { v non contiene entry }
if k < key(v)
    return TreeSearch(k, T.left(v))
else if k = key(v)
    return v
else { k > key(v) }
    return TreeSearch(k, T.right(v))

```



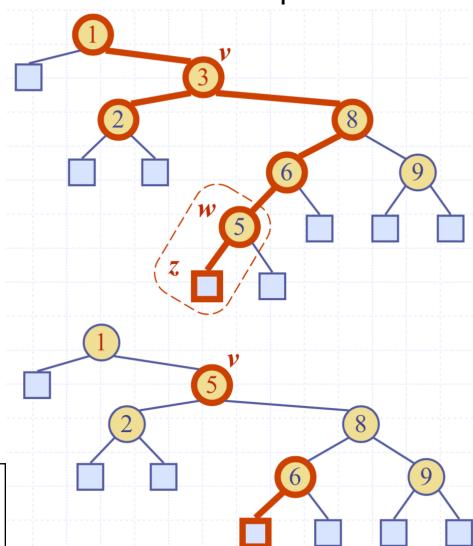
L'inserimento avviene eseguendo un *TreeSearch* fino a raggiungere una foglia nella quale memorizzare il nuovo elemento

Anche la cancellazione inizia con un *TreeSearch*, una volta trovato il nodo si possono avere 3 situazioni:

- Il nodo non ha figli => si rimuove banalmente senza influire sugli altri elementi
- Il nodo ha 1 figlio => si sostituisce il riferimento del nodo con quello del figlio
- Il nodo ha 2 figli => in questo caso bisogna trovare il nodo con la più piccola chiave maggiore rispetto a quella da rimuovere, ovvero l'ultima chiave a sinistra del suo sottoalbero destro.

Una volta individuata si sostituisce con la entry rimossa facendo attenzione a copiare nuovamente tutti i riferimenti ad essa collegati

ATTENZIONE il minimo potrebbe avere figli destri, quindi il primo si sostituisce al nodo rimosso



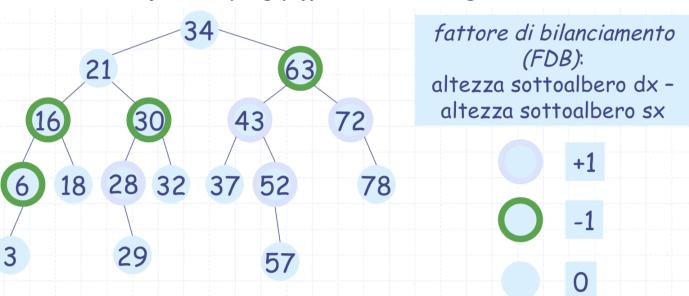
AVL (alberi bilanciati)

Sono un tipo particolare di BST ai quali si aggiunge la proprietà:

- $|altezza(right) - altezza(left)| < 1$ detto **fattore di bilanciamento**, ovvero, per ciascun nodo interno, l'altezza del sottoalbero a destra e quella del sottoalbero a sinistra può differire al massimo di 1

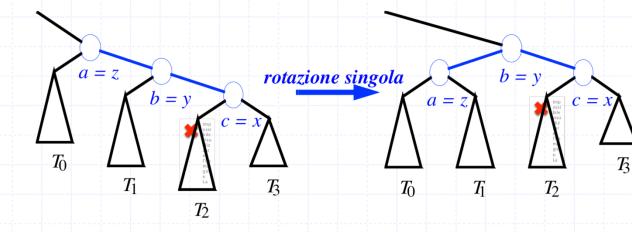
Questa proprietà serve per mantenere l'albero bilanciato, cosa che permette di dimostrare che l'altezza di un AVL con n nodi è sempre $O(\log(n))$, di conseguenza si ha che le operazioni, legate all'altezza, hanno sempre costo **$O(\log(n))$** ⁵.

Per verificare la proprietà si effettua una visita post order e si controlla la profondità dei sottoalberi di ogni nodo. Quindi per verificare che l'albero sia AVL si effettua un cammino Euleriano che combina postorder ed inorder.

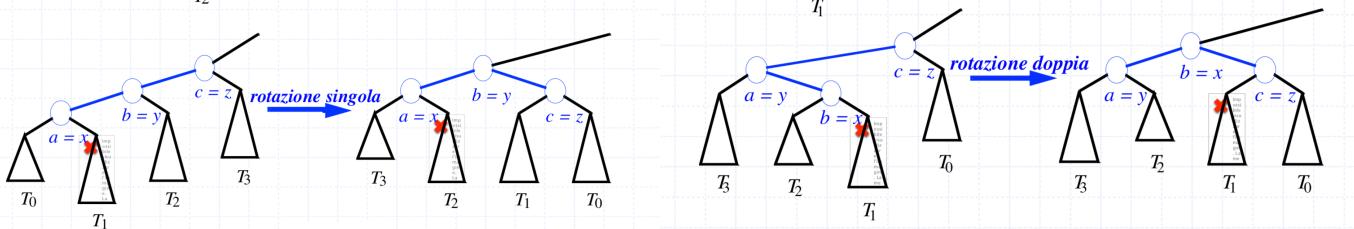


Inserimenti e rimozioni sono inizialmente uguali a quelli di un normale BST ma con l'aggiunta di un ribilanciamento dell'albero, che si può effettuare tramite delle rotazioni, singole (verso destra o sinistra) oppure doppie (in entrambe o nella stessa direzione). Inserimenti nel sottoalbero destro comportano rotazioni a sinistra, inserimenti nel sottoalbero sinistro comportano rotazioni a destra.

◆ Rotazioni singole:



◆ Rotazioni doppie:

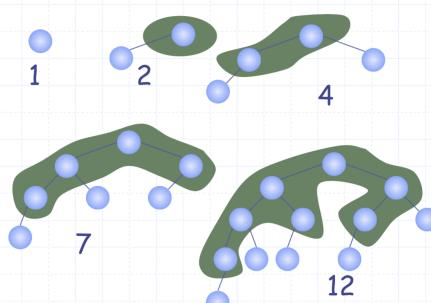


ATTENZIONE mentre per gli inserimenti basta una sola ristrutturazione, per le rimozione potrebbe essere necessario risalire per tutto il ramo ed effettuare $\log(n)$ ristrutturazioni

OSS Gli alberi di fibonacci sono AVL, che, data una determinata altezza, hanno il minor numero possibile di nodi in condizione di bilanciamento, mantenendosi sempre nella condizione più vicina allo sbilanciamento. Inoltre si ha che ogni albero di fibonacci segue la seguente relazione:

$$AVL_i = Fib(i+2) - 1$$

Quindi potando tutte le foglie di un albero di fibonacci si ottiene l'albero corrispondente al numero precedente.



h	F_h	AVL_h
0	0	0
1	1	1
2	1	2
3	2	4
4	3	7
5	5	12
6	8	20
7	13	33

⁵ assumendo che una singola ristrutturazione richiede tempo $O(1)$,

Algoritmi di ordinamento

Il lower bound per gli algoritmi di ordinamento (basati su confronto) è **O(n log(n))**.

Molti di questi algoritmi si basano sul paradigma di programmazione **divide et impera**:

1. Divide => divide l'insieme S in input in 2 sottoinsiemi disgiunti S_1 ed S_2
 2. Ricorri => risolve sottoproblemi associati ad S_1 ed S_2
 3. impera => combina le soluzioni di S_1 ed S_2 in una soluzione per S
- OSS il caso base della ritorsione sono problemi di taglia 0 oppure 1.

MergeSort

Algoritmo di ordinamento basato sulla tecnica di programmazione divide et impera (difficile da implementare in place), composto da 2 funzioni:

- MergeSort => si occupa di dividere ricorsivamente l'array in input in 2 metà fino ad avere tanti piccoli array formati da un singolo elemento
- Merge => si occupa di fondere nuovamente le sequenze ordinate $A(S_1)$ e $B(S_2)$ in ingresso in un'unica sequenza ordinata S contenente l'unione dei 2 insiemi

Algorithm *mergeSort(S)*

Input sequence S with n elements

Output sequence S sorted according to C

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

mergeSort(S_1)

mergeSort(S_2)

$S \leftarrow merge(S_1, S_2)$

Algorithm *merge(A, B)*

Input sequences A and B with $n/2$ elements each

Output sorted sequence of $A \cup B$

$S \leftarrow$ empty sequence

while $\neg A.isEmpty() \wedge \neg B.isEmpty()$

if $A.first().element() < B.first().element()$
 S.insertLast(A.remove(A.first()))

else

S.insertLast(B.remove(B.first()))

while $\neg A.isEmpty()$

S.insertLast(A.remove(A.first()))

while $\neg B.isEmpty()$

S.insertLast(B.remove(B.first()))

return S

È possibile illustrare l'esecuzione dell'algoritmo con un albero binario, tramite il quale è possibile stabilire anche i costi:

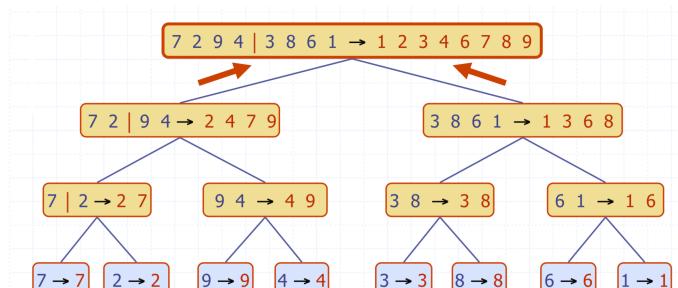
- La divisione a metà che genera l'albero indica un costo $O(h)$, dovendo sempre a metà si ha un albero bilanciato e quindi un costo $O(\log(n))$
- Ad ogni nodo di profondità i si ha costo $O(n)$ per scorrere tutti gli elementi

Moltiplicando il costo di ogni livello per il loro numero si ha il costo totale **$O(n \log(n))$** .

Inoltre si può formulare la seguente equazione di ricorrenza:

$$f(n) = \begin{cases} \theta(1), & n = 2 \\ f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + \theta(n), & n > 2 \end{cases}$$

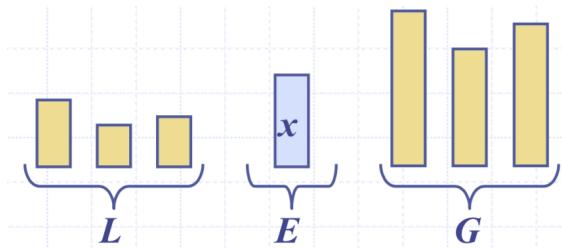
OSS l'implementazione reale rende però questo algoritmo meno efficiente rispetti ad altri con costo equivalente.



QuickSort

Altro algoritmo di ordinamento basato sul concetto di divide et impera, il funzionamento si può riassumere in:

1. Scelta di un pivot, ovvero un elemento dell'array che indica quali elementi mettere in L e quali in G
2. Scorrere tutto l'array e partizionare gli elementi in modo che i minori del pivot siano in L ed i maggiori in G
3. Scegliere un nuovo pivot in entrambi gli insiemi ed operare in entrambi il punto 2 ricorsivamente
4. Una volta arrivati ad avere degli insiemi di un solo elemento si passa alla fusione



OSS la scelta del pivot influisce sulle prestazioni dell'algoritmo facendo variare il costo da $O(n \log n)$ per il caso migliore ad $O(n^2)$ per il caso peggiore⁶.

Questo è dimostrabile tramite lo stesso albero binario creato dalle suddivisioni del merge sort, che in questo caso però non ha garanzie sul bilanciamento essendo gli insiemi divisi in base al pivot scelto invece che sempre a metà.

Nel caso in cui le suddivisioni creano un albero sbilanciato con profondità pari all'input, avendo che ad ogni livello le operazioni hanno costo $O(n)$, il costo totale risultante arriva quindi ad $O(n^2)$.

Algorithm *partition(S, p)*

Input sequence S , position p of pivot

Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.

$L, E, G \leftarrow$ empty sequences

$x \leftarrow S.\text{remove}(p)$

$E.\text{insertLast}(x)$

while $\neg S.\text{isEmpty}()$

$y \leftarrow S.\text{remove}(S.\text{first}())$

if $y < x$

$L.\text{insertLast}(y)$

else if $y = x$

$E.\text{insertLast}(y)$

else { $y > x$ }

$G.\text{insertLast}(y)$

return L, E, G

3

Algorithm *inPlaceQuickSort(S, l, r)*

Input sequence S , ranks l and r

Output sequence S with the elements of rank between l and r rearranged in increasing order

if $l \geq r$

return

$i \leftarrow$ a random integer between l and r

$x \leftarrow S.\text{elemAtRank}(i)$

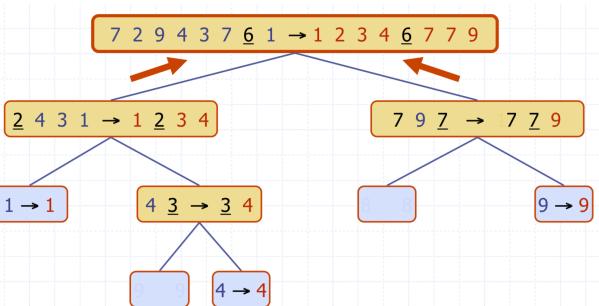
$(h, k) \leftarrow \text{inPlacePartition}(x)$

$\text{inPlaceQuickSort}(S, l, h - 1)$

$\text{inPlaceQuickSort}(S, k + 1, r)$

È importante notare che il passo partition del quick sort può essere implementato in place, usando una combinazione di indici, ad ogni iterazione:

1. Si sceglie il pivot
2. Si crea un indice i che scandisce l'array ed un indice j che segnala la posizione in cui si dividono gli elementi inferiori dai maggiori
3. Ogni volta che i incontra un elemento più piccolo lo scambia con j che aumenta di 1
4. Finito di scandire l'array si scambia l'elemento corrispondente al pivot nella posizione contrassegnata da j e si ritorna il valore per indicare dove avviene divisione



⁶ solitamente il valore nel mezzo è il migliore poiché evita problemi nel caso di array preordinati.

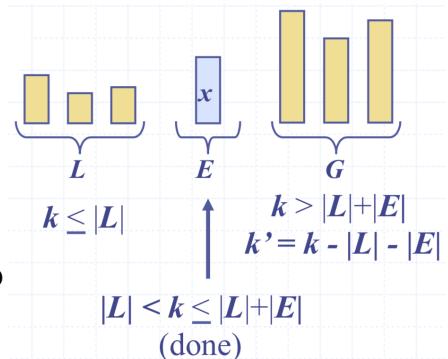
Quick select

Algoritmo derivante dal quick sort che permette di risolvere il problema della selezione, ovvero dato un insieme di n elementi x_1, x_2, \dots, x_n ed un intero k , trovare il k -esimo elemento dell'insieme.

Si potrebbe riordinare l'insieme ma questo ha un costo minimo di $O(n \log(n))$, tramite questo algoritmo è invece possibile portare il costo ad $O(n)$.

L'esecuzione si può riassumere in:

1. Scelta di un pivot
 2. Scorrere tutto l'array e partizionare gli elementi in modo che i minori del pivot siano in L ed i maggiori in G
 3. Esaminando gli insiemi si possono avere le seguenti situazioni:
 - A. Il pivot, adesso contenuto in E , corrisponde al k -esimo elemento, quindi si ritorna il pivot
 - B. La dimensione di L è maggiore di k , si esegue ricorsivamente quick select su L
 - C. La dimensione di L è minore di k , si esegue ricorsivamente quick select su G
- ATTENZIONE** bisogna considerare anche la L precedente



BucketSort

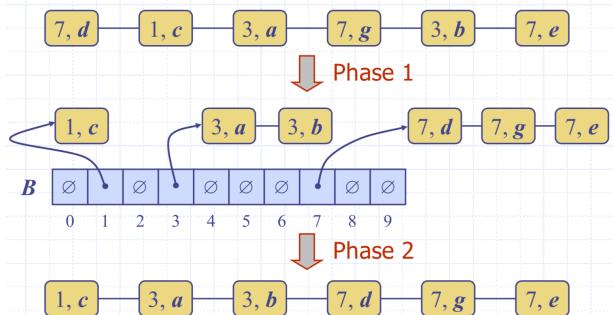
Algoritmo d'ordinamento non basato sul confronto utilizzabile solo su insiemi di numeri interi e non implementabile in-place.

Funzionamento:

1. Si trova l'elemento con valore maggiore (e in una versione ottimizzata anche minore) nell'array
2. Si crea un array d'appoggio con dimensione corrispondente a max (o differenza tra i 2 valori trovati), e tutte le caselle inizializzate a 0
3. Si scandisce l'array in input e ad ogni occorrenza di un numero si incrementa il valore della casella corrispondente nell'array d'appoggio⁷
4. Si scandisce l'array d'appoggio e per ogni casella con valore >0 si inseriscono nell'array di output tanti valori corrispondenti al numero della casella quanto è il valore contenuto.

```
countingSort(A[])
  //Calcolo degli elementi max e min
  max ← A[0]
  min ← A[0]
  for i ← 1 to length[A] do
    if (A[i] > max) then
      max ← A[i]
    else if(A[i] < min) then
      min ← A[i]
  //Costruzione dell'array C
  * crea un array C di dimensione max - min + 1
  for i ← 0 to length[C] do
    C[i] ← 0
  for i ← 0 to length[A] do
    C[A[i] - min] = C[A[i] - min] + 1
  //Ordinamento in base al contenuto dell'array delle frequenze C
  k ← 0
  for i ← 0 to length[C] do
    while C[i] > 0 do
      A[k] ← i + min
      k ← k + 1
      C[i] ← C[i] - 1
```

Costo **$O(\max + n)$** poiché bisogna scorrere anche l'array d'appoggio di lunghezza max. Da notare che max dipende dalla sua rappresentazione binaria portando quindi ad un costo esponenziale, l'algoritmo è quindi utilizzabile solo su insiemi con max bassi



⁷ facendo attenzione a non alterare i valori nel caso in cui min sia diverso da 0

Ordine lessicografico

Ordine impartito su insiemi composti da chiavi multidimensionali, ovvero del tipo:

$$(x_1, x_2, \dots, x_d) < (y_1, y_2, \dots, y_d)$$

$$\Leftrightarrow$$

$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \dots, x_d) < (y_2, \dots, y_d)$$

In cui ogni tupla è ordinata comparando per prima la prima dimensione, poi la seconda, la terza,

Il LexicographicSort è implementato richiamando ad ogni iterazione, corrispondente ad una dimensione, lo stesso algoritmo di ordinamento (a scelta) che sia occupa di ordinare solo quella dimensione.
È importante notare che si parte dall'ultima dimensione, poiché è la meno significativa per l'ordinamento totale.

Il costo dell'algoritmo è quindi **O(d(T(n)))** dove con $T(n)$ si indica il costo del sottoalgoritmo di ordinamento usato per ogni dimensione.

OSS Un algoritmo di ordinamento si dice **stabile** quando questo, trovando 2 elementi uguali, questo non ne inverte le posizioni

RadixSort

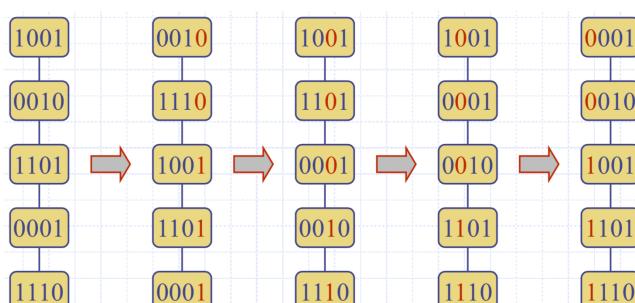
È una specializzazione del lexicographicSort che utilizza BucketSort come sottoalgoritmo di ordinamento.

Questo lo porta ad avere un costo **O(d(max+n))** dove valgono le stesse considerazioni fatte per gli altri 2 algoritmi.

Il vero vantaggio dell'utilizzo di questo algoritmo sta nella sua versione binaria, ovvero che prende in input una sequenze di soli bit.

$$x = x_{b-1} \dots x_1 x_0$$

Questo permette di avere un bucketSort con max costante uguale a 2 e quindi un costo totale lineare pari a **O(d(n))**



Algorithm *lexicographicSort(S)*

Input sequence S of d -tuples
Output sequence S sorted in lexicographic order

```
for  $i \leftarrow d$  downto 1
    stableSort( $S, C_i$ )
```

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)
(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)
(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)
(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

Il costo dell'algoritmo è quindi **O(d(T(n)))** dove con $T(n)$ si indica il costo del sottoalgoritmo di ordinamento usato per ogni dimensione.

Algorithm *radixSort(S, N)*

Input sequence S of d -tuples such that $(0, \dots, 0) \leq (x_1, \dots, x_d)$ and $(x_1, \dots, x_d) \leq (N-1, \dots, N-1)$ for each tuple (x_1, \dots, x_d) in S

Output sequence S sorted in lexicographic order

```
for  $i \leftarrow d$  downto 1
    bucketSort( $S, N$ )
```

Algorithm *binaryRadixSort(S)*

Input sequence S of b -bit integers

Output sequence S sorted
replace each element x of S with the item $(0, x)$

```
for  $i \leftarrow 0$  to  $b - 1$ 
    replace the key  $k$  of each item  $(k, x)$  of  $S$  with bit  $x_i$  of  $x$ 
```

bucketSort($S, 2$)

Insiemi

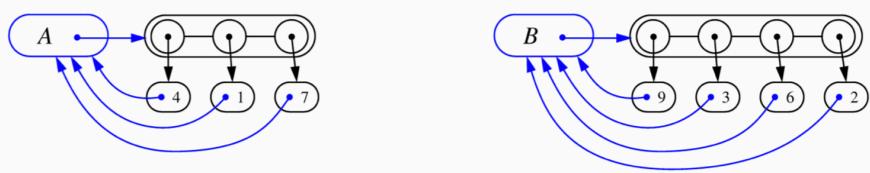
È possibile rappresentare gli insiemi come TdA, il metodo più semplice per implementarli è tramite le liste collegate, dati due insiemi A e B questo permette di eseguire le operazioni di unione ed intersezione con costi $O(n_A + n_B)$.

Per diminuire i tempi è possibile usare le union-find basate sulle operazioni:

- **makeSet(x)**: crea un insieme composto dal solo elemento x e restituisci la posizione che memorizza x in questo insieme
- **union(A, B)**: Restituisci l'insieme $A \cup B$, distruggendo i vecchi A e B
- **find(p)**: Restituisci l'insieme contenente l'elemento in posizione p

Questo metodo, nel caso delle liste, prevede di modificare la struttura dati aggiungendo ad ogni elemento un puntatore al nodo che identifica l'insieme.

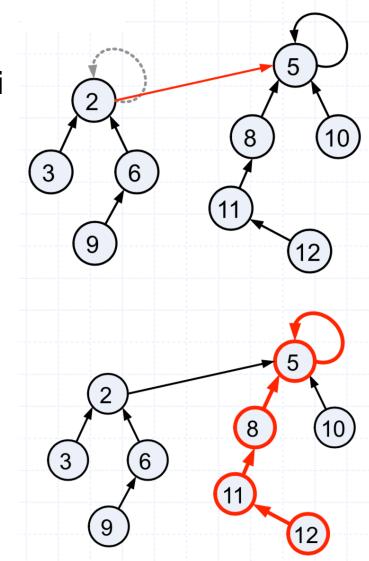
Ad ogni operazione di unione si muovono sempre gli elementi dell'insieme più piccolo verso quelle più grande.



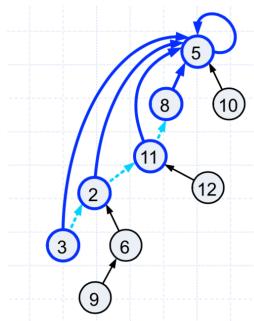
Nel caso degli alberi invece si inverte il senso di quelli già presenti, ovvero, invece di avere che ogni nodo ha vari puntatori ai figli, ogni figlio ha un puntatore al padre e la radice un puntatore a se.

Quindi per eseguire un unione basta modificare il puntatore alla radice di un albero facendolo puntare al nodo radice dell'altro.

Secondo la **union-by-size** è l'albero con meno nodi a dover essere unito, secondo la **union-by-rank** quello più basso in modo da avere un albero il più bilanciato possibile.



Nel caso del find è possibile partire dal nodo assegnato ripercorrendo tutto l'albero fino a trovare il nodo radice, ovvero quello con il puntatore a se stesso.



Un metodo per aumentare le prestazioni della find e dell'albero in generale è seguire una **path-compression**, ovvero partendo dall'idea della find ad ogni passaggio da un nodo all'altro si modifica il puntatore al nodo genitore con un puntatore diretto alla root.

Il costo delle euristiche union-by-size, union-by-rank e path-compression su n elementi è $O(m a(n))$, dove $a(n)$ è l'inverso della funzione di Ackermann.

Grafi

ATTENZIONE i grafi sono insiemi, quindi gli elementi (nodi) sono unici

Un grafo è una coppia (V, E) , dove:

- V è un insieme di nodi, detti vertici (vertex)
- E è una collezione di coppie di vertici, detti archi o spigoli (edge)
- Vertici e spigoli sono Position e contengono elementi

Se viene associato un valore agli archi questo si dice grafo pesato, inoltre se viene associato un verso agli archi si dice anche grafo diretto (o orientato).

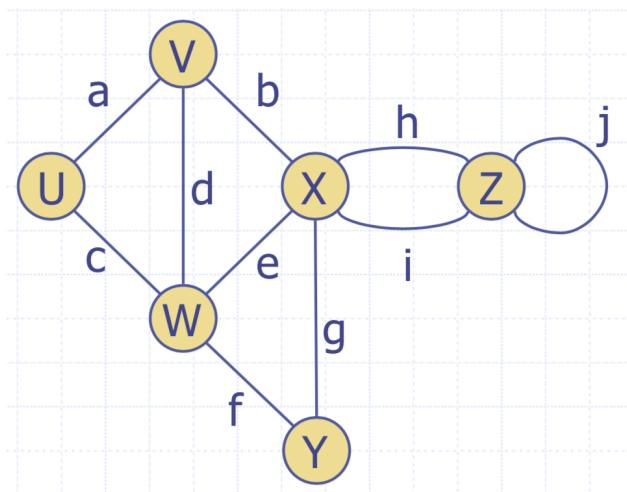
Un grafo è essenzialmente una relazione binaria $R = VxV$ su V .

Dal punto di vista matematico esiste solo il grafo orientato poiché ogni coppia è ordinata, ovvero $(V_i, V_j) \neq (V_j, V_i)$, però soddisfa comunque la proprietà di simmetria, ovvero se (a,b) appartiene alla relazione, vi appartiene anche (b,a) .

Quindi se tutti gli archi di andata coincidono con quelli di ritorno il grafo non è orientato.

Terminologia

- Uno o più spigoli si dicono **incidenti** ad un vertice se sono connessi ad esso
ex: a, b, d sono incidenti su V
- Due vertici si dicono **adiacenti** se collegati dallo stesso arco
ex: U e V sono adiacenti
- Il **grado** di un vertice corrisponde al numero dei suoi archi uscenti
ex: X ha grado 5
- Due archi si dicono **paralleli** se collegano gli stessi nodi
ex: h ed i sono paralleli
- Un **self-loop** è un arco che loopa su sullo stesso nodo
ex: j è un self-loop
- Un **pozzo** è un nodo senza archi uscenti (vale solo per grafi orientati)



Un **Path** (percorso) è una sequenza alternata di nodi ed archi collegati tra loro, parte con un nodo e finisce con un nodo, può essere:

- Path semplice => tutti gli archi ed i nodi sono distinti, no ripetizioni
- Path non semplice => ci possono essere ripetizioni di nodi e/o archi

Inoltre il path può essere ciclico, ovvero sequenza circolare di nodi ed archi collegati tra loro in cui ogni spigolo è preceduto e seguito dai sui punti terminali (semplice o no)

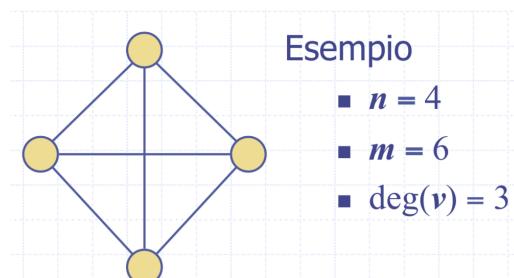
Proprietà

Indicando con n il numero nodi, m il numero archi e $\deg(v)$ il grado del vertice V

Si hanno le proprietà:

1. $\sum_v \deg(v) = 2m$
2. In un grafo diretto e privo di autoanelli e spigoli multipli si ha:
$$m \leq (n - 1)/2$$

Quindi un grafo con $m = n^2$ si dice denso, un grafo con $m = n$ si dice sparso



TdA grafo

<ul style="list-style-type: none"> ◆ Vertici e spigoli <ul style="list-style-type: none"> ■ sono Position ■ contengono elementi ◆ Metodi accessori <ul style="list-style-type: none"> ■ endVertices(e): array con i due vertici terminali di e ■ opposite(v, e): il vertice opposto di v, sullo spigolo e ■ areAdjacent(v, w): vero se e solo se v e w sono adiacenti ■ replace(v, x): sostituisce l'elemento nel vertice v con x ■ replace(e, x): sostituisce l'elemento nello spigolo e con x 	<ul style="list-style-type: none"> ◆ Metodi modificatori <ul style="list-style-type: none"> ■ insertVertex(o): inserisce un vertice contenente l'elemento o ■ insertEdge(v, w, o): inserisce uno spigolo (v, w) contenente l'elemento o ■ removeVertex(v): rimuove il vertice v (e gli spigoli incidenti) ■ removeEdge(e): rimuove lo spigolo e ◆ Metodi iteratori <ul style="list-style-type: none"> ■ incidentEdges(v): spigoli incidenti su v ■ vertices(): tutti i vertici nel grafo ■ edges(): tutti gli spigoli nel grafo
--	---

Implementazioni

La prima tipologia è la **lista degli spigoli**, basata su due liste, una contente tutti i vertici ed una tutti gli archi.

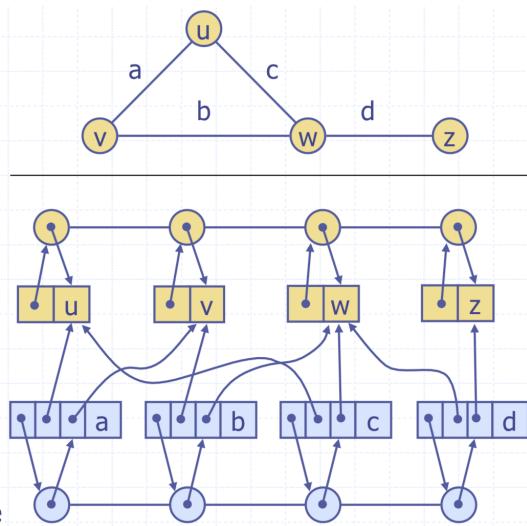
Ogni elemento vertex è composto da:

- Elemento
- Riferimento alla posizione nella sequenza dei vertici

Ogni elemento edge è composto da:

- Elemento
- Riferimenti ad i 2 vertici che collega
- Riferimento alla posizione nella sequenza di archi.

Ha il problema di non poter trovare tutti gli spigoli adiacenti velocemente.



La seconda tipologia è la **lista delle adiacenze**, anche in questo caso si hanno le 2 liste di vertici e spigoli ma cambiano gli oggetti contenuti.

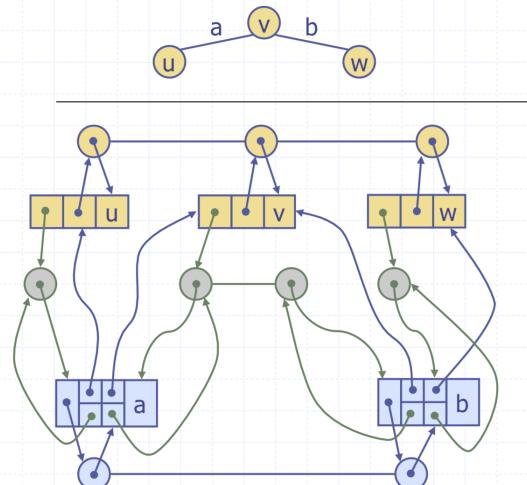
Ogni elemento vertex è composto da:

- Elemento
- Riferimento alla posizione nella sequenza dei vertici
- Riferimenti a tutti gli oggetti spigolo collegati al vertice

Ogni elemento edge è composto da:

- Elemento
- Riferimenti ad i 2 vertici che collega
- Riferimento alla posizione nella sequenza di archi.
- Riferimenti agli oggetti vertici corrispondenti ad i suoi vertici terminali

Rende più efficiente la ricerca dei vertici adiacenti, infatti dato un vertice con k nodi adiacenti permette di scorrerli in un tempo $O(k)$, con $O(1)$ per ogni nodo.



La terza tipologia è la **matrice delle adiacenze**, anche in questo caso si hanno le 2 liste di vertici e spigoli, inoltre gli oggetti contenuti sono simili a quelli della lista degli spigoli.

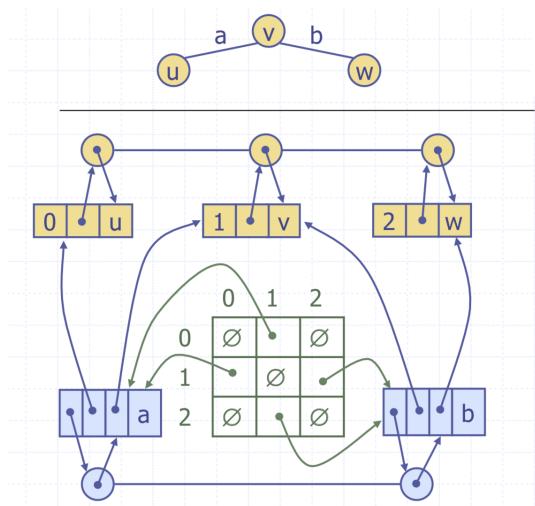
L'unico cambiamento si ha negli oggetti vertice ad i quali viene aggiunta una chiave intera che funge da indice per la matrice.

Associando gli indici della matrice con gli indici dei vertice si ha che ad ogni intersezione corrisponde il riferimento all'arco.

Nel caso in cui l'arco non è presente si ha NULL. Permette quindi di verificare le adiacenze in $O(1)$.

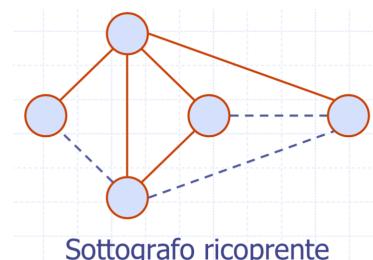
I costi si riassumono in:

	Lista spigoli	Lista adiacenze	Matrice adiacenze
◆ n vertici, m spigoli			
◆ no spigoli multipli			
◆ no autoanelli			
◆ Costi in O -grande			
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

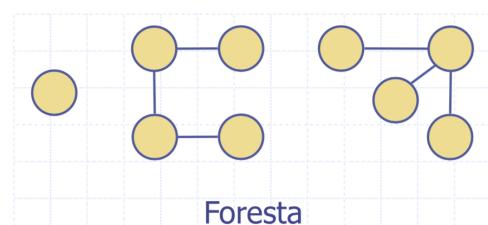


Altre definizioni

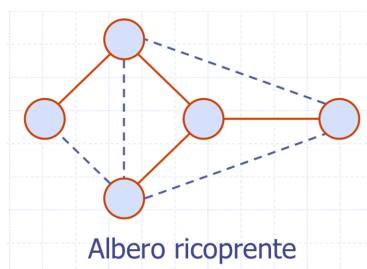
- **Sottografo** => un sottografo S di un grafo G è un grafo tale che i suoi vertici ed i suoi spigoli sono un sottoinsieme di G
- **Sottografo ricoprente** => un sottografo ricoprente S di un grafo G è un sottografo contenente tutti i vertici di G
- **Connettività** => un grafo si dice connesso se esiste un percorso (path) fra ogni coppia di vertici
- **Albero** => è un grafo non orientato T, tale che:
 - T è connesso
 - T è aciclico
- Una **foresta** è un grafo aciclico non orientato avente componenti non connesse, ovvero altri alberi
- Un **Minimum spannino tree** (albero ricoprente) di un grafo connesso è un sottografo ricoprente con l'ulteriore caratteristica di essere un albero
 OSS l'albero ricoprente di un grafo non è un unico, a meno che il grafo non sia un albero



Sottografo ricoprente



Forest



Albero ricoprente

DFS (depth-first-search)

Algoritmo per l'attraversamento dei grafi, dato un grafo con n vertici ed m spigoli, la DFS richiede un tempo $O(n+m)$ ⁸.

Questo algoritmo permette di:

- Visitare tutti vertici e gli spigoli
- Determinare se G è连通的
- Calcolare le componenti connesse
- Calcolare una foresta ricoprente

Algorithm $DFS(G)$

Input grafo G

Output etichettatura degli spigoli di G come tree-edge e back-edge

```
for all  $u \in G.vertices()$ 
    setLabel( $u$ , UNEXPLORED)
for all  $e \in G.edges()$ 
    setLabel( $e$ , UNEXPLORED)
for all  $v \in G.vertices()$ 
    if getLabel( $v$ ) = UNEXPLORED
        DFS( $G$ ,  $v$ )
```

Algorithm $DFS(G, v)$

Input grafo G vertice iniziale v di G

Output etichettatura degli spigoli di G nella componente connessa di v come tree-edge e back-edge

setLabel(v , VISITED)

```
for all  $e \in G.incidentEdges(v)$ 
    if getLabel( $e$ ) = UNEXPLORED
         $w \leftarrow opposite(v, e)$ 
        if getLabel( $w$ ) = UNEXPLORED
            setLabel( $e$ , DISCOVERY)
            DFS( $G$ ,  $w$ )
        else
            setLabel( $e$ , BACK)
```

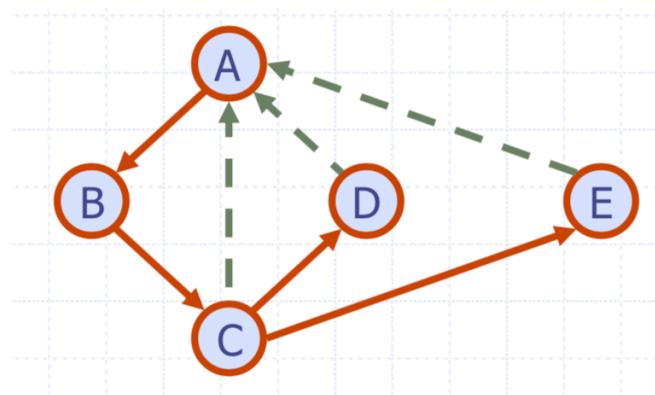
Funzionamento:

1. Tutti i nodi ed archi vengono marchiati come UNEXPLORED, per poi iniziare la visita ricorsiva da un nodo generico (tra quelli inesplorati)
2. Per prima cosa il vertice viene settato come VISITED, si inizia quindi la visita dei nodi adiacenti, con le seguenti caratteristiche:
 - Arco UNEXPLORED => anche il nodo potrebbe essere UNEXPLORED, quindi
 - Nodo UNEXPLORED => arco viene settato come DISCOVERY e si effettua chiamata ricorsiva sul nuovo nodo
 - Nodo VISITED => arco è di ritorno, quindi si contrassegna come BACK
3. Se ci sono vertici ancora non visitati si rilancia la DFS ricorsiva
OSS questo implica che il grafo non è连通的

Proprietà

1. $DFS(G, v)$ visita tutti i vertici e gli spigoli nella componente connessa che contiene v
2. I tree-edges etichettati da $DFS(G, v)$ formano un albero ricoprente della componente connessa che contiene v

OSS questo funzionamento lo rende quindi ottimo per la ricerca di percorsi e l'attraversamento di labirinti.



⁸ questo a condizione di implementare il grafo tramite la lista delle adiacenze

Ricerca path

La specializzazione per la ricerca di percorsi avviene aggiungendo il parametro z che rappresenta la fine del percorso.

Raggiunto z la ricerca termina.

```
Algorithm pathDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements()
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v, e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                S.push(e)
                pathDFS(G, w, z)
                S.pop(e)
            else
                setLabel(e, BACK)
    S.pop(v)
```

Ricerca cicli

Inoltre è ottimo per la ricerca di cicli, infatti è possibile modificare l'algoritmo ricorsivo in modo che, quando incontra un arco back, restituisce la lista di tutti i vertici che formano il ciclo, ovvero tutti gli elementi presenti nella stack *S* fino al corrispondente del nodo appena trovato.

```
Algorithm cycleDFS(G, v)
    setLabel(v, VISITED)
    S.push(v)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v, e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                cycleDFS(G, w)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

Digrafi (grafi diretti)

Grafo avente tutti gli spigoli orientati, ovvero per ogni spigolo vale la proprietà:

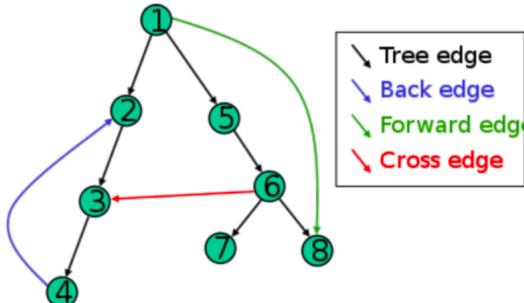
- Lo spigolo (a, b) va da a a b , ma non da b ad a

Inoltre se il grafo è semplice vale la proprietà $m \leq n * (n - 1)$

OSS Facendo uso delle liste di adiacenza è possibile ottimizzare questi grafi separando le liste di grafi entranti ed uscenti.

Nell'implementazione della DFS "diretta" si aggiungono 2 nuovi tipi di arco:

- **Forward** => porta a nodo discendente già visitato tramite altri path
- **Cross** => porta a nodo non parente già visitato tramite altro path



Terminologia

- **Raggiungibilità** => indica quali altri vertici sono percorribili partendo da un vertice v
- **Connettività forte** => si dice che un grafo orientato ha connettività forte quando, tramite ciascun vertice, è possibile raggiungere tutti gli altri vertici del grafo.

Se il grafo è fortemente connesso anche il suo trasposto è fortemente connesso

OSS usando solo la parte ricorsiva della DFS su un nodo generico, se esistono uno o più vertici non visitati da essa, allora il grafo non è fortemente connesso.

- **Connettività debole** => il grafo non orientato è fortemente connesso
- **Componenti fortemente connesse** => sottografi massimali del grafo principale che godono della proprietà di connettività forte

Verificabili tramite una doppia DFS (su sottografo e trasposto del sottografo)

Chiusura transitiva

Dato un grafo G , la chiusura transitiva di G è un grafo G^* tale che

- G^* ha gli stessi vertici di G
- Se G ha un percorso orientato da u a v ($u \neq v$), allora G^* ha uno spigolo orientato che va da u a v

Ovvero se esiste un modo per andare da a a b e da b a c , allora ne esiste uno per andare da a a c

La chiusura transitiva permette di avere informazioni sulla raggiungibilità in un grafo.

Sarebbe possibile eseguire n DFS, ma si avrebbe un costo $O(n(n^*m))$, alternativamente si può usare la "programmazione dinamica" come nel caso di **Floyd-Warshall**.

Può essere seguito in tempo $O(n^3)$ se il metodo AreAdicent ha costo $O(1)$, ovvero tramite la matrice di adiacenza.

```

Algorithm FloydWarshall( $G$ )
Input digrafo  $G$ 
Output la chiusura transitiva  $G^*$  di  $G$ 
 $i \leftarrow 1$ 
for all  $v \in G.vertices()$ 
    denota  $v$  come  $v_i$ 
     $i \leftarrow i + 1$ 
 $G_0 \leftarrow G$ 
for  $k \leftarrow 1$  to  $n$  do
     $G_k \leftarrow G_{k-1}$ 
    for  $i \leftarrow 1$  to  $n$  ( $i \neq k$ ) do
        for  $j \leftarrow 1$  to  $n$  ( $j \neq i, k$ ) do
            if  $G_{k-1}.areAdjacent(v_p, v_k) \wedge$ 
                 $G_{k-1}.areAdjacent(v_k, v_j)$ 
            if  $\neg G_k.areAdjacent(v_p, v_j)$ 
                 $G_k.insertDirectedEdge(v_p, v_j, k)$ 
    return  $G_n$ 

```

DAG e ordinamento topologico

Un DAG (directed acyclic graph) è un digrafo senza cicli orientati, ovvero esso descrive un partially order set.

Un ordinamento topologico di un digrafo è una numerazione v_1, v_2, \dots, v_n dei vertici tale che per ogni spigolo (v_i, v_j) risulta $i < j$.

Teo un digrafo ammette ordinamento topologico se e solo se è un DAG

È possibile effettuare topological sort tramite una variante della DFS in tempo $O(n + m)$.

ATTENZIONE la DFS deve essere effettuata su una copia del grafo in input.

L'algoritmo itera la DFS fino a trovare un pozzo, quando lo trova assegna a ritroso i valori da n ad 1 ad ogni nodo.

Algorithm *topologicalDFS(G)*

Input DAG G

Output ordinamento topologico di G

$n \leftarrow G.numVertices()$

for all $u \in G.vertices()$

setLabel(u , UNEXPLORED)

for all $e \in G.edges()$

setLabel(e , UNEXPLORED)

for all $v \in G.vertices()$

if *getLabel*(v) = UNEXPLORED

topologicalDFS(G, v)

Algorithm *topologicalDFS(G, v)*

Input grafo G e vertice iniziale v di G

Output etichettatura dei vertici di G nella componente connessa di v

setLabel(v , VISITED)

for all $e \in G.incidentEdges(v)$

if *getLabel*(e) = UNEXPLORED

$w \leftarrow \text{opposite}(v, e)$

if *getLabel*(w) = UNEXPLORED

setLabel(e , DISCOVERY)

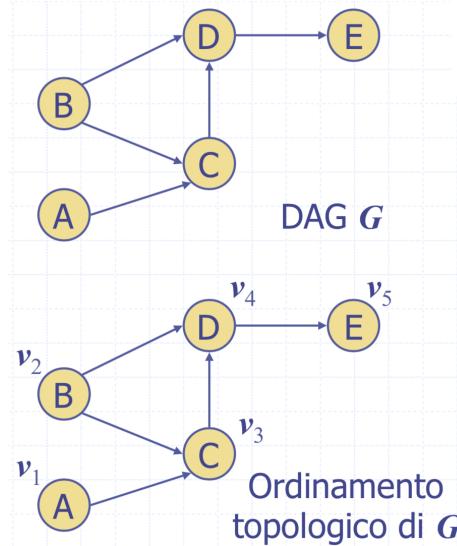
topologicalDFS(G, w)

else

 { e è uno spigolo forward o cross }

 etichetta v con il numero n

$n \leftarrow n - 1$ // side effect!



BFS (breadth-first search)

Algoritmo per la visita dei nodi filosoficamente opposto alla DFS, anche in questo caso l'algoritmo richiede $O(n + m)$ se implementato con lista delle adiacenze.

Questo algoritmo permette di:

- Visitare tutti vertici e gli spigoli
- Determinare se G è连通的
- Calcolare le componenti connesse
- Calcolare una foresta ricoprente

Algorithm $BFS(G)$

Input graph G

Output labeling of the edges
and partition of the
vertices of G

```

for all  $u \in G.vertices()$ 
  setLabel( $u$ , UNEXPLORED)
for all  $e \in G.edges()$ 
  setLabel( $e$ , UNEXPLORED)
for all  $v \in G.vertices()$ 
  if getLabel( $v$ ) = UNEXPLORED
     $BFS(G, v)$ 
```

Algorithm $BFS(G, s)$

```

 $L_0 \leftarrow$  new empty sequence
 $L_0.insertLast(s)$ 
setLabel( $s$ , VISITED)
 $i \leftarrow 0$ 
while  $\neg L_r.isEmpty()$ 
   $L_{i+1} \leftarrow$  new empty sequence
  for all  $v \in L_r.elements()$ 
    for all  $e \in G.incidentEdges(v)$ 
      if getLabel( $e$ ) = UNEXPLORED
         $w \leftarrow opposite(v, e)$ 
        if getLabel( $w$ ) = UNEXPLORED
          setLabel( $e$ , DISCOVERY)
          setLabel( $w$ , VISITED)
           $L_{i+1}.insertLast(w)$ 
        else
          setLabel( $e$ , CROSS)
   $i \leftarrow i + 1$ 
```

Funzionamento:

1. Tutti i nodi ed archi vengono marchiati come UNEXPLORED, per poi iniziare la visita da un nodo generico
2. Per prima cosa il vertice viene settato come VISITED e si inizializza una lista per i vicini, quindi si comincia la visita dei nodi adiacenti. Si può quindi avere:
 - Arco UNEXPLORED => anche il nodo potrebbe essere UNEXPLORED, quindi
 - Nodo UNEXPLORED => arco viene settato come DISCOVERY e si setta come VISITED per poi inserirlo nella lista dei nodi da usare per la prossima iterazione (ovvero quelli a distanza 2)
 - Nodo VISITED => arco è di ritorno, quindi si contrassegna come CROSS
3. Finiti i vertici a distanza 1 si prende uno dei vertici inseriti il L e si ripete il punto 2 per visitare tutti i vertici a distanza 2 dal nodo, per poi passare a 3...
4. Se ci sono vertici ancora non visitati si rilancia la BFS interna

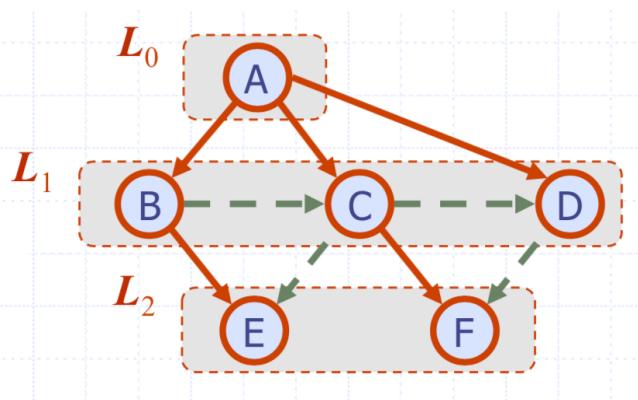
Quindi in questo caso si ha che, partendo da un nodo generico, si visitano prima tutti i nodi a distanza 1 (ovvero quelli adiacenti), poi quelli a distanza 2, 3, ...

Questo la rende ottima per la ricerca dei percorsi minimi tra i nodi.

Proprietà

Data una componente连通的 G_s di un grafo G

1. La BFS visita tutti i vertici e gli archi di G_s
2. Gli archi etichettati DISCOVERY dalla BFS formano uno spennino tree T_s di G_s
3. Per ogni vertice v in L_i
 - Il cammino di T_s da s a v ha i archi
 - Ogni cammino da s a v in G_s ha almeno i archi



DFS vs BFS

Applicazioni	DFS	BFS
Spanning forest, componenti connesse, cammini, cicli	✓	✓
Cammini minimi		✓
Componenti biconnesse	✓	

DFS

BFS

Back edge (v, w)	Cross edge (v, w)
<ul style="list-style-type: none"> w e' un antenato di v nell' albero degli archi discovery 	<ul style="list-style-type: none"> w e' nello stesso livello di v o nel livello successivo nell' albero degli archi discovery
<p>DFS</p>	<p>BFS</p>

Shortest path e minimum spanning tree

Lo shortest path (cammino minimo) è un problema secondo il quale, dato un grafo pesato e 2 vertici u e v , si vuole trovare il cammino con il minimo costo totale tra u e v . Il minimum spanning tree è stato introdotto nelle definizioni preliminari.

Proprietà

1. Un sottocammino di un cammino minimo è un cammino minimo
2. l'insieme dei cammini minimi da un vertice a tutti gli altri vertici forma un albero (**MST**)

Algoritmo di Dijkstra

Assunzioni per l'utilizzo dell'algoritmo:

- Il grafo è连通的
- Gli archi sono non diretti
- I pesi degli **archi non sono negativi**

Non funziona con pesi negativi
poiché in presenza di cicli il
costo negativo potrebbe alterare
le distanze già considerate

Basato sull'idea di creare una nuvola che si espande dal vertice di partenza fino a ricoprire tutti i vertici del grafo, per ogni vertice v viene memorizzata la distanza $d(v)$, eseguendo un rilassamento ogni volta che si aggiunge un nodo alla nuvola.

Dato un arco $e = (u, z)$, con u il vertice aggiunto più recentemente e z non appartenente alla nuvola, il rilassamento dell'arco e aggiorna la distanza $d(z)$ come:

$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$

Quindi ogni volta che si percorre un arco che porta ad un nuovo nodo il rilassamento permette di decidere se mantenere il vecchio percorso o sostituirlo con il nuovo.

Algorithm *DijkstraDistances*(G, s)

```
 $Q \leftarrow$  new heap-based priority queue
for all  $v \in G.\text{vertices}()$ 
  if  $v = s$ 
    setDistance( $v, 0$ )
  else
    setDistance( $v, \infty$ )
   $l \leftarrow Q.\text{insert}(\text{getDistance}(v), v)$ 
  setLocator( $v, l$ )
while  $\neg Q.\text{isEmpty}()$ 
   $u \leftarrow Q.\text{removeMin}()$ 
  for all  $e \in G.\text{incidentEdges}(u)$ 
    { relax edge  $e$  }
     $z \leftarrow G.\text{opposite}(u, e)$ 
     $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$ 
    if  $r < \text{getDistance}(z)$ 
      setDistance( $z, r$ )
       $Q.\text{replaceKey}(\text{getLocator}(z), r)$ 
```

Funzionamento:

1. L'algoritmo comincia impostando la distanza di tutti i vertici ad ∞ e salvandoli in una coda di priorità Q
2. Ad ogni iterazione del while si estrae il primo vertice dalla coda e per ogni vertice vicino ad esso si esegue l'operazione di rilassamento

L'implementazione del locator serve per memorizzare la posizione del nodo nella queue, in questo modo risulta più semplice trovarlo ad aggiornarne la distanza.

Dato che bisogna eseguire n inserimenti nella coda con costo $O(\log(n))$, ed ogni modifica di etichetta comporta un nuovo inserimento, considerando che ci sono $\deg(v) = 2m$ archi, utilizzando una lista delle adiacenze, il costo totale dell'algoritmo è $O((n+m) \log(n))$.

Dijkstra può essere inoltre esteso per restituire un minimum spanning tree dal vertice di partenza a tutti gli altri vertici.

Si può fare aggiungendo alle 2 etichette, distanza e posizione locator, una terza etichetta Parent.
Questa deve però essere aggiornata ogni volta che si segue il rilassamento di un arco.

Detto anche algoritmo di **Prim-Jarnik's**

```
Algorithm DijkstraShortestPathsTree( $G, s$ )
...
for all  $v \in G.vertices()$ 
...
setParent( $v, \emptyset$ )
...
for all  $e \in G.incidentEdges(u)$ 
...
if  $r < getDistance(z)$ 
    setDistance( $z, r$ )
    setParent( $z, e$ )
    Q.replaceKey( $getLocator(z), r$ )
```

Algoritmo di Kruskal

Basato sulle proprietà di ciclo e partizione

- Ciclo => Sia T il minimum spanning tree di un grafo pesato G , sia e un arco di G che non è in T e C il ciclo formato da e in T , per ogni arco f di C vale

$$weight(f) \leq weight(e)$$
- Partizione => avendo una partizione dei vertici di G in due sottoinsiemi U e V , e sia e un arco di peso minimo che attraversa la partizione, allora il MST contiene e

Algorithm Kruskal(G)

Input: A weighted graph G .

Output: An MST T for G .

Implementazione prevede utilizzo di makeSet, union e find su insiemi

Let P be a partition of the vertices of G , where each vertex forms a separate set.

Let Q be a priority queue storing the edges of G , sorted by their weights

Let T be an initially-empty tree

while Q is not empty **do**

$(u, v) \leftarrow Q.removeMinElement()$

if $P.find(u) \neq P.find(v)$ **then**

 Add (u, v) to T

$P.union(u, v)$

return T

Running time: $O((n + m)\log n)$

Funzionamento:

1. Vengono isolati tutti i vertici creando una partizione per ognuno di essi
2. Si crea una priority queue contenente tutti gli archi appartenenti al grafo ordinati secondo il rispettivo peso
3. Iterativamente si estraggono tutti gli archi in ordine di peso, se non esiste già un arco che collega i 2 vertici (partizioni) allora questo viene aggiunto all'albero

Il costo teorico dell'algoritmo, dato solo dall'estrazione degli archi dalla lista, è $O(n\log(n))$

Counting sort (simile bucket)

Algoritmo d'ordinamento non basato sul confronto che permette di avere costo **$O(n)$** , utilizzabile solo su insiemi di numeri interi e non implementabile in-place.

Funzionamento:

1. Si trovano gli elementi con valore maggiore e minore nell'array
2. Si crea un array d'appoggio con dimensioni corrisposti alla differenza tra i 2 valori trovati e tutte le caselle inizializzate a 0
3. Si scandisce l'array in input e ad ogni occorrenza di un numero si incrementa il valore della casella corrispondente nell'array d'appoggio⁹
4. Si scandisce l'array d'appoggio e per ogni casella con valore >0 si inseriscono nell'array di output tanti valori corrispondenti al numero della casella quanto è il valore contenuto.

```
countingSort(A[])
    //Calcolo degli elementi max e min
    max ← A[0]
    min ← A[0]
    for i ← 1 to length[A] do
        if (A[i] > max) then
            max ← A[i]
        else if(A[i] < min) then
            min ← A[i]
    //Costruzione dell'array C
    * crea un array C di dimensione max - min + 1
    for i ← 0 to length[C] do
        C[i] ← 0
    for i ← 0 to length[A] do
        C[A[i] - min] = C[A[i] - min] + 1
    //Ordinamento in base al contenuto dell'array
    delle frequenze C
    k ← 0
    for i ← 0 to length[C] do
        while C[i] > 0 do
            A[k] ← i + min
            k ← k + 1
            C[i] ← C[i] - 1
```

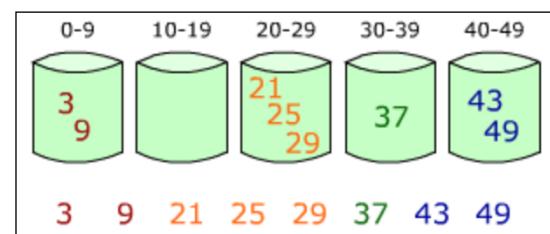
V2 Bucket sort

Altro algoritmo d'ordinamento non basato sul confronto, non esegue un vero e proprio ordinamento ma divide l'array input in n sotto-array sui quali usare un altro algoritmo supplementare.

Questo permette di avere insiemi più semplici da ordinare diminuendo la complessità generale e portandola, almeno in linea teorica¹⁰, a **$O(n)$** .

Funzionamento:

1. Si cerca il valore massimo nell'array
2. Si crea un array in cui ogni casella corrisponde ad una lista collegata dove salvare i numeri compresi in determinati intervalli scelti
3. Si scorre l'array in input e si inserisce ogni elemento nel "contenitore" corrispondente
4. Tramite un algoritmo supplementare si ordina ogni bucket separatamente
5. Si uniscono tutti bucket in sequenza precedentemente ordinati



```
BucketSort(array A, intero N)
    for i ← 1 to length[A] do
        // restituisce un indice di bucket per l'elemento A[i]
        bucket ← f(A[i], N)
        // inserisce l'elemento A[i] nel bucket corrispondente
        aggiungi(A[i], B[bucket])
    for i ← 1 to N do
        // ordina il bucket
        ordina(B[i])
    // restituisce la concatenazione dei bucket
    return concatena(B)
```

⁹ facendo attenzione a non alterare i valori nel caso in cui min sia diverso da 0

¹⁰ le prestazioni di questo algoritmo dipendono molto dalla distribuzione dei valori

V2 Radix sort

Come gli altri 2 algoritmi basati sul confronto non effettua comparazioni ma suddivide gli elementi in gruppi, in questo caso corrispondenti ad una cifra decimale 0-9.

Funzionamento:

1. Si creano 10 liste corrispondenti ad i valori decimali da 0 a 9
2. Prendendo in considerazione l'ultima cifra di ogni numero si posiziona l'elemento nella queue corrispondente
3. Si ripete il punto 2 per ogni altra cifra ricordando di mantenere l'ordine (implicito) già imposto alle cifre precedenti

```
a[n] array of integers, length n, D decimal digits
q[10] queues

for (p=0; p<D; p++)
    for (i=0; i<n; i++)
        q[(a[i]/10^p)%10].insert(a[i])
    i=0;
    for (j=0; j<10; j++)
        while (q[j].notEmpty())
            a[i] = q[j].remove()
            i++
```

Si ha quindi che l'algoritmo ha una complessità $O(d*n)$ dove d è il numero delle cifre di cui sono composti i numeri in input.

OSS ovviamente nei numeri con meno cifre le più significative corrispondono a 0.