# Appunti ASD 2022/2023

## Indice:



---

# 1. Analisi Asintotica

**Big Oh:** diciamo che una funzione $f(n)$ è in $O(g(n))$ se da un "n0" in poi, la nostra $f$**() sta sotto a g(n).**
Esempio:

- $n2 \in O(n3)$
- $n \in O(n2)$

**Omega:** una funzione $f(n)$ è in $\Omega(g(n))$ se da un "n0" in poi, la nostra $f$**() sta sopra a g(n)**. Esempio:

- $12n! \in \Omega(n2)$

**Theta:** è un mix tra Big Oh e Omega, la funzione $f(n)$ è in $\Theta(g(n))$ se da un "n0,n1" in poi, la nostra $f$**() sta in mezzo a due funzioni g(n) moltiplicate da costanti differenti**.

---

# 2. Algoritmi di Ordinamento e Ricerca

## Selection Sort:

Ricevuta una sequenza in input, il selection sort:

1. Individua il **minimo**
2. sposta il minimo all'inizio della sequenza (quindi ho **seq. ordinata** e **seq. NON ordinata**)
3. Procede con il prossimo minimo

**Complessità:**

- Caso migliore: $\theta(n^2)$
- Caso peggiore: $\theta(n^2)$

I due casi coincidono in quanto anche se la sequenza fosse già ordinata in partenza, l'algoritmo esegue comunque tutte le operazioni (visita tutte le celle).

## Insertion Sort:

Preso un elemento (all'inizio il secondo), controlla se è minore di tutti gli elementi precedenti e nel caso aggiusta la posizione.

**Complessità:**

- Caso migliore: $\theta(n)$
- Caso peggiore: $\theta(n^2)$

Se la sequenza fosse già ordinata l'algoritmo esce dal `while` senza chiamare la funzione di `swap()` riducendo la complessità alla sola "lettura" delle celle (aka $\theta(n)$)

## Bubble Sort:

Confronta a due a due gli elementi della sequenza e se "A > B" allora effettua uno `swap()`. L'algoritmo ripete se stesso fintanto che non sono stati effettuati scambi, quindi la lista è ordinata.

**Complessità:**

- Caso migliore: $\theta(n)$
- Caso peggiore: $\theta(n^2)$

# Merge Sort:

L'algoritmo, data una sequenza:

- trova la metà
- chiama `ms` su 1° metà (ricorsivo)
- chiama `ms` su 2° metà (ricorsivo)
- chiama `fondi` sulle due metà

**Complessità:**

Vale $\theta(n\ log\ n)$ sia nel caso migliore che peggiore, infatti **non essendo un algoritmo adattivo**, effettua tutte le chiamate ricorsive necessarie, e tutti i confronti, anche nel caso in cui l'array preso in considerazione sia già ordinato.

La complessità dell' algoritmo deriva dal calcolo dell'espressione: **n° livelli * costo ciascun problema**.

| LIVELLO | NUMERO PROBLEMI | DIMENSIONE PROBLEMA |
|---|---|---|
| 0 | 1 | n |
| 1 | 2 | n/2 |
| 2 | 4 | n/4 |
| 3 | 8 | n/8 |
| **j** | **2^j** | **n/2^j** |

- **Costo di ogni livello:**

  Ad ogni livello **j** si hanno $2^j$ sottoproblemi di tipo `merge`, ognuno lungo $n/2^j$ e quindi risolvibile in $\theta(n/2^j)$. Per conoscere il costo di ogni livello, bisogna **moltiplicare il costo di `merge` per il numero di volte che viene chiamato:**

  $2^j * n/2^j = n \rightarrow \theta(\mathbf{n})$ (costo di ogni livello di ricorsione).

- **Numero di livelli:**

  Sapendo che all'ultimo livello della ricorsione, i sottoproblemi assumono dimensione **1**, e che su ogni livello "j", i sottoproblemi sono di dimensione $n/2^j$, per quale j si ha che $n/2^j = 1$? ——> $j = log_2(n)$

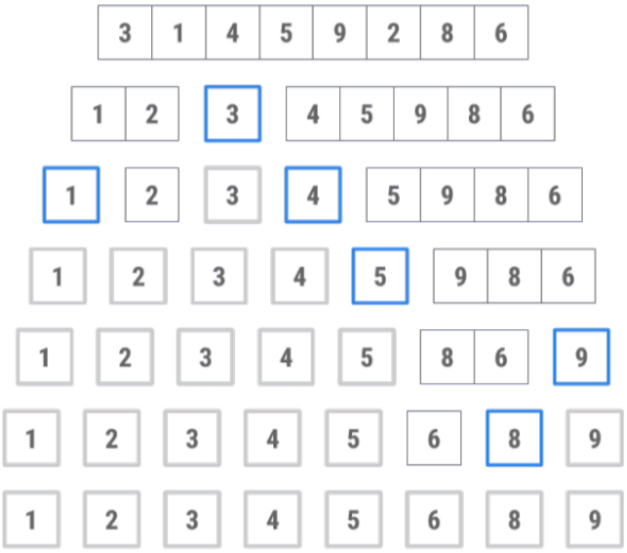
  Quindi il numero di livelli è: **log2 n + 1** [+1 perche si parte dal livello zero]

In conclusione il **costo di MergeSort** è dato da: $\theta(\mathbf{n}) \times \mathbf{log_2\ n} \rightarrow \theta(\mathbf{n\ log\ n})$ sia nel caso migliore che nel caso peggiore.

# Quick Sort:

É un algoritmo ricorsivo che fa due chiamate ricorsive. Funziona nel seguente modo:

- prende un elemento dell'array (il **pivot**)
- Partiziona gli elementi dell'array in modo tale che quelli a sinistra siano minori del pivot e quelli a destra siano maggiori o uguali al pivot
- dopo queste operazioni il pivot è nella sua posizione finale
- Richiama quicksort sulle due parti dell'array ottenute




**Complessità:**

L' efficienza del quicksort dipende **da come scegliamo il pivot.**

- caso **migliore**: $\theta(n\ log\ n)$ –> è un caso ipotetico in quando si sceglie come pivot il **mediano** che però non possiamo sapere senza prima riordinare la sequenza. Essendo che il pivot è sempre l'elemento mediano, l'array verrà diviso dalla partition in due sottosequenze di lunghezze circa n/2. Quindi, come nel caso del mergesort, **ottengo un albero binario bilanciato** che ha altezza log2(n), che è il numero di iterazioni della funzione ricorsiva. Ad ogni livello j dell'albero la partition viene effettuata su $(2^j)$*$(n/2^j)$ elementi e ha quindi complessità Θ(n). La complessità finale sarà dunque Θ(n*log2(n)).
- caso **medio**: $\theta(n\ log\ n)$ simile alla complessità del mergesort, abbiamo che:
  - "n" deriva dal numero di operazioni svolte ad ogni livello dell'albero della ricorsione: al livello j si effettuano $2^j$ chiamate partition, ciascuna su una porzione di array lunga $n/2^j$.

    Ciascuna di queste chiamate ha **complessità lineare** nella dimensione della porzione di array su cui viene chiamata, quindi **O(n/2^j).** Ad ogni livello faccio **$2^j$ * O(n/$2^j$)** operazioni che determinano **O(n).**
  - "log n" deriva dal numero di livelli del'albero della ricorsione

- caso **peggiore**: $\theta(n^2)$ quando si sceglie come pivot l'elemento **minore** o **maggiore** della sequenza su cui partition viene chiamata o quando la sequenza è già ordinata. Questa è data dalla partition poiché al lv. 0 costerà n, al lv. 1 costerà n-1,al lv 2 costerà n-2 e cosi via,quindi si ha che n+(n-1)+(n- 2)+(n-3)..... = n*(n+1)/2 = $n^2$.

  > *Come si calcola la complessità nel caso peggiore di quicksort?*
  >
  > Si sommano le operazioni fatte ad ogni livello dell'albero delle chiamate ricorsive.
  >
  > Tali operazioni sono dovute alla chiamata `partition`, che al livello 0 verrà chiamata su `n` elementi (quindi n operazioni), al livello 1 verrà chiamata su n-1 elementi,[...], al livello n verrà chiamata su un elemento e poi non ci saranno altre chiamate ricorsive.
  >
  > Questo calcolo si sviluppa nella **sommatoria per 'i' che va da 1 a n** in **O(n^2).**

### Quicksort Random

Per ogni array di dimensione n, il tempo di esecuzione di quickSort randomizzato nel caso medio è **Θ(n log n)**

# Binary Search:

É un algoritmo della famiglia *Divide et Impera* ossia ad ogni iterazione, la parte di dati che andremo ad esplorare sarà divisa sempre per 2.

**Complessità:**

- Caso migliore: $O(1)$
- Caso peggiore: $O(log\ n)$

---

# 3. Modularità e information hiding, Utilizzo dei namespace

## Modularità

La modularizzazione è la suddivisione in parti (moduli) di un sistema, in modo che esso risulti più semplice da comprendere e manipolare.

La suddivisione di un sistema in parti richiede che ciascuna di esse realizzi un aspetto o un comportamento ben definito all'interno del sistema.

Un modulo in un sistema software è un componente che può essere utilizzato per realizzare una astrazione. Può esportare all'esterno: servizi/funzioni ("astrazione sul controllo"); dati ("astrazione sui dati"); identificativi.

A sua volta può importare dati, funzionalità e identificativi da altri moduli. Definisce un AMBIENTE DI VISIBILITA'.
Un modulo è dotato di una chiara separazione tra:

- **Interfaccia**: specifica "cosa" fa il modulo e "come" si utilizza. Deve essere visibile all'esterno del modulo per poter essere utilizzata dall'utente del modulo al fine di usufruire dei servizi/dati esportati dal modulo.
- **Corpo**: descrive "come" l'astrazione è realizzata e contiene l'implementazione delle funzionalità/strutture dati esportate dal modulo che sono nascoste e protette all'interno del modulo. L'utente può accedere ad esse solo attraverso l'interfaccia.

## Information Hiding

Consiste nel nascondere e proteggere (incapsulare) alcune informazioni di una entità all'interno del modulo.

Le interfacce devono essere:

- **minimali**, cioè devono esporre solo ciò che è strettamente necessario;
- **stabili**, cioè possibilmente non devono subire cambiamenti nel tempo: se l' interfaccia non cambia, le informazioni nascoste possono essere modificate senza che questo influisca sulle altre parti del sistema di cui l'entità fa parte

## Utilizzo dei NameSpace

I "namespace" sono stati introdotti nel linguaggio C++ per evitare collisioni fra nomi.

Con l'introduzione dei namespace è possibile specificare a quale namespace appartiene una particolare funzione, variabile o classe.

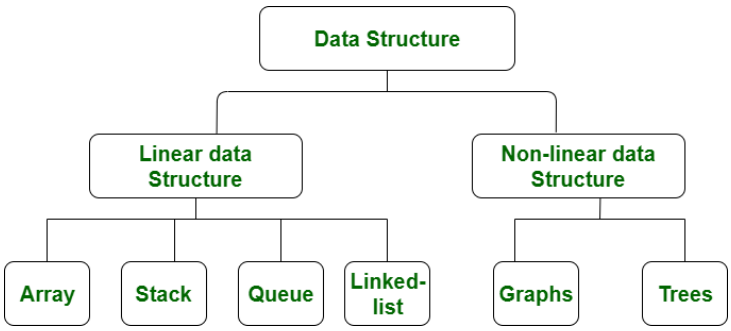
Per indicare al compilatore che ad esempio l'identificatore cout deve essere cercato nello spazio dei nomi standard (std) anteponiamo a cout, std:: , dove:

- `std` è lo spazio dei nomi standard
- `::` è l'operatore di risoluzione dell'ambito

I nomi raggruppati dal namespace hanno visibilità locale al namespace, in questo senso sono “incapsulati” nel namespace e non sono visibili all’esterno; in questo modo si evitano i conflitti nel caso gli stessi nomi si ripetano nell’ambito del programma.

---

# Strutture dati

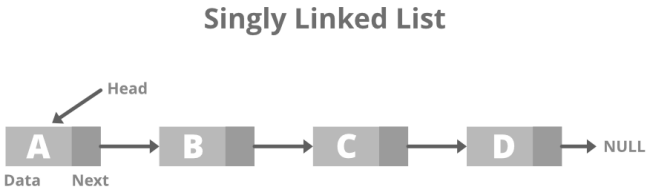



---

# 4. Liste

Le liste sono una **struttura dati dinamica** ossia può facilmente espandersi o ridursi senza allocare in anticipo lo spazio in memoria.

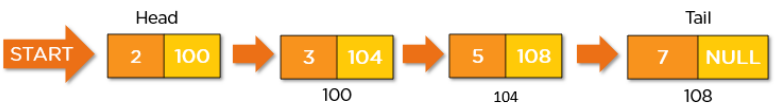
Le liste collegate memorizzano ogni elemento come un oggetto separato. Ciò significa che gli elementi di una lista collegata non vengono archiviati in slot di memoria contigui, invece, **ogni elemento** (chiamato nodo) **contiene un puntatore alla posizione del nodo successivo**. Questi puntatori mantengono la connessione tra i nodi. Oltre al puntatore al nodo successivo, **un nodo contiene anche un campo dati**.



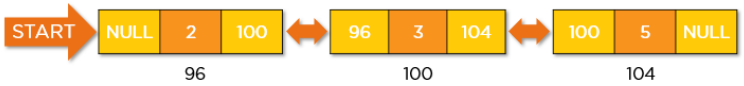

Con gli elenchi collegati, **non è possibile accedere direttamente a un singolo elemento senza attraversare l’elenco** a partire dalla testa. Ciò conferisce all’operazione di accesso una complessità temporale di Θ (n).

Esistono diverse tipologie di liste, divise per tipologia di collegamento. Individuiamo:

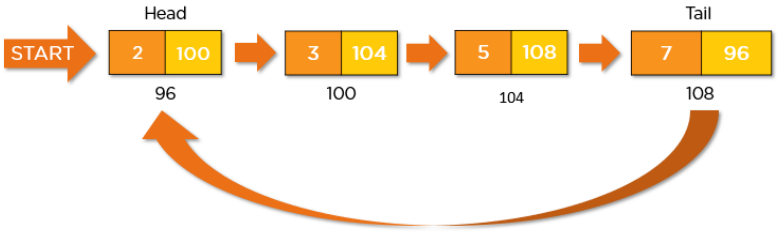
- **semplici:** le liste semplici sono liste **unidirezionali**: si possono attraversare in una sola direzione.



- **doppiamente collegate:** sono liste **bidirezionali**: ogni nodo contiene anche il puntatore al nodo precedente.

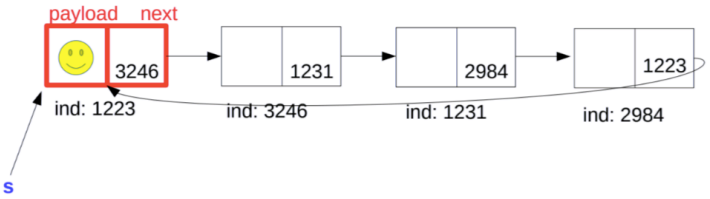



- **circolari**: sono come le liste semplici ma **l'ultimo nodo punta al primo**:

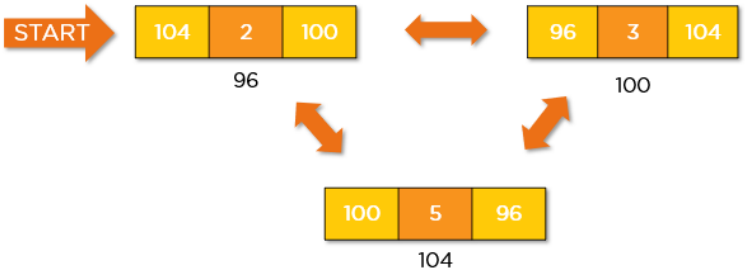



- **circolari con sentinella**: aggiungendo una cella fittizia all'inizio della lista si può ovviare al problema di avere due casi distinti nel caso in cui la lista sia vuota o con degli elementi. Quindi la **lista vuota sarà una lista con solo la sentinella** e tutte le celle che vorremo aggiungere verranno inserite dopo la sentinella.

  Avendo la sentinella come primo elemento non dovremo neanche preoccuparci di modificare il puntatore alla lista nel caso modificassimo l'ordine degli elementi o eliminassimo il primo.




- **circolari e doppiamente collegate**: sono **liste doppiamente collegate** in cui **l'ultimo elemento punta al primo**.




- **doppiamente collegate, circolari e con sentinella**: come sopra ma grazie alla sentinella non dobbiamo più preoccuparci del caso "lista vuota" in quanto non esiste.

## Complessità:

- **liste semplici**:

| Operazione | Caso Migliore | Caso Peggiore |
|---|---|---|
| Set() | $\Theta(1)$ | $\Theta(n)$ |
| Add() | $\Theta(1)$ | $\Theta(n)$ |
| AddFront() | $\Theta(1)$ | $\Theta(1)$ |
| AddBack() | $\Theta(n)$ | $\Theta(n)$ |
| RemovePos() | $\Theta(1)$ | $\Theta(n)$ |
| Get() | $\Theta(1)$ | $\Theta(n)$ |
| IsEmpty() | $\Theta(1)$ | $\Theta(1)$ |
| Size() | $\Theta(n)$ | $\Theta(n)$ |
| EmptyList() | $\Theta(1)$ | $\Theta(1)$ |

- **liste doppiamente collegate, circolari e con sentinella**:

| Operazione | Caso Migliore | Caso Peggiore |
|---|---|---|
| Set() | $\Theta(1)$ | $\Theta(1)$ |
| Add() | $\Theta(1)$ | $\Theta(n)$ |
| AddFront() | $\Theta(1)$ | $\Theta(1)$ |
| AddBack() | $\Theta(1)$ | $\Theta(1)$ |
| RemovePos() | $\Theta(1)$ | $\Theta(n)$ |
| Get() | $\Theta(1)$ | $\Theta(n)$ |
| IsEmpty() | $\Theta(1)$ | $\Theta(1)$ |
| Size() | $\Theta(n)$ | $\Theta(n)$ |
| EmptyList() | $\Theta(1)$ | $\Theta(1)$ |

- **Array Dinamico + size e maxsize (con ridimensionamento)**:

| Operazione | Caso Migliore | Caso Peggiore |
|---|---|---|
| Set() | $\Theta(1)$ | $\Theta(1)$ |
| Add() | $\Theta(1)$ | $\Theta(n)$ |
| AddFront() | $\Theta(n)$ | $\Theta(n)$ |
| AddBack() | $\Theta(1)$ | $\Theta(n)$ |
| RemovePos() | $\Theta(1)$ | $\Theta(n)$ |
| Get() | $\Theta(1)$ | $\Theta(1)$ |
| IsEmpty() | $\Theta(1)$ | $\Theta(1)$ |
| Size() | $\Theta(1)$ | $\Theta(1)$ |
| EmptyList() | $\Theta(1)$ | $\Theta(1)$ |

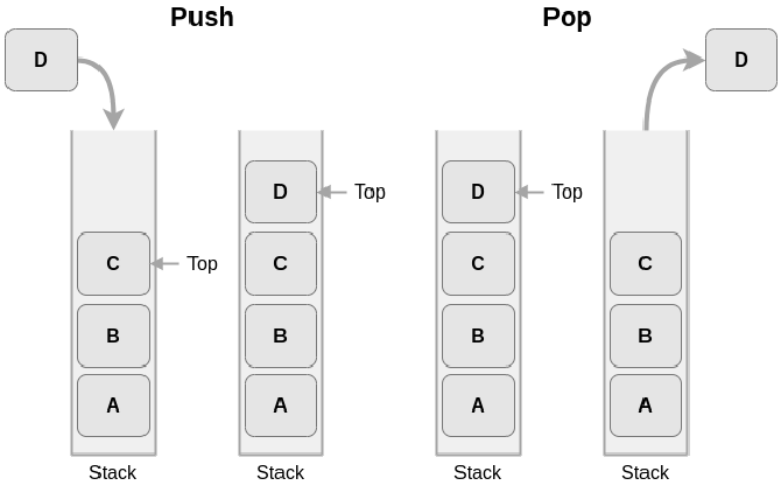
# 5. Pile e Code

## Pile (stack)

Una pila o stack è una collezione di elementi che restringe le modalità di accesso a una logica **Last In First Out** (**LIFO**; letteralmente, l'ultimo a entrare è il primo a uscire): è possibile accedere solo all'elemento inserito per ultimo tra quelli ancora presenti nella pila, e per accedere ad un elemento generico è necessario prima rimuovere tutti quelli che sono stati inseriti successivamente ad esso.

Operazioni:

- **push(elem e)**: inserisce un nuovo elemento in cima alla pila. Un elemento appena aggiunto diventa il nuovo top.
- **pop()**: rimuove un elemento dalla cima della pila.
- **isEmpty()**: restituisce true o false
- **top()**: restituisce l'elemento in cima alla pila senza toglierlo (leggo)




**Complessità:**

Posso **implementare uno stack** usando:

- array dinamici —> tutte le operazioni sono in **Θ (1)** nel caso migliore, nel caso peggiore invece la `push` vale **Θ (n)**
- **liste semplici** —> tutte le operazioni sono in **Θ (1).**
- **liste doppiamente collegate e circolari** —> tutte le operazioni sono in **Θ (1).**

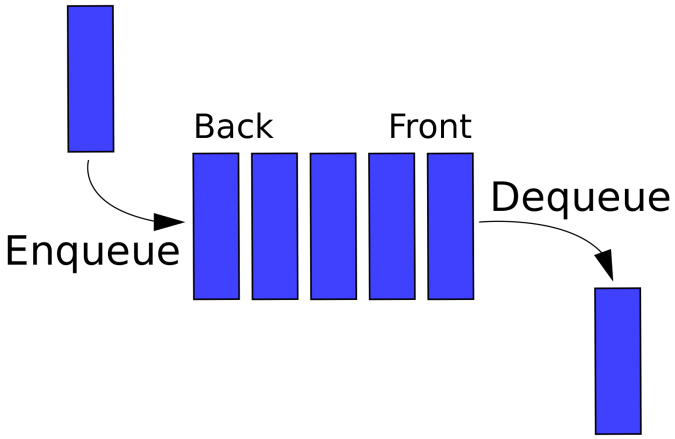
# Code (queue)

“Il primo che entra è il primo che esce” (**FIFO**).

E’ una sequenza di elementi di un certo tipo, in cui è possibile aggiungere elementi ad un estremo e toglierne dall'estremo opposto.

Operazioni:

- **isEmpty()** → result restituisce true se S è vuota
- **enqueue(elem e)** → aggiunge e come ultimo elemento di S
- **dequeue()** → elem toglie da S il primo elemento e lo restituisce
- **first()** → elem restituisce il primo elemento di S




**Complessità:**

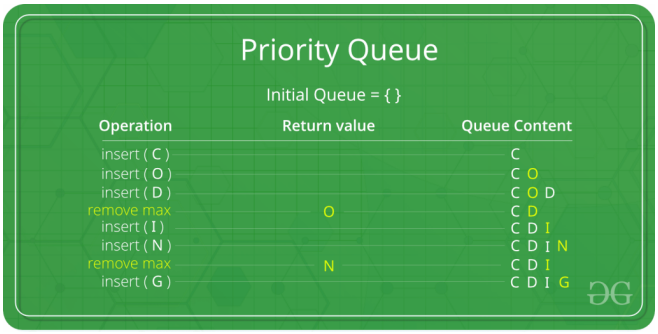
Posso **implementare una queue** usando:

- **liste doppiamente collegate e circolari** —> tutte le operazioni sono in **Θ (1).**
- liste semplici —> tutte le operazioni sono in **Θ (1)** tranne per la `enqueue` che in entrambi i casi vale **Θ (n)**.
- array dinamici —> la `dequeue` vale **Θ (n)** in entrambi i casi; la `enqueue` vale **Θ (n)** solo nel caso peggiore. Il resto è in **Θ (1)**.

## Priority Queue

Le code a priorità sono un tipo di dato in cui **ad ogni elemento è associata una sua priorità** che determina l’ordine in cui gli elementi son rimossi dalla coda.
Quindi tutti gli elementi sono disposti in ordine crescente o decrescente.

Esempio di priority queue con valore ASCII usato per assegnare priorità:




Operazioni:

- **insert(p):** inscerisce un nuovo elemeneto con priorità ‘p’
- **deleteMax():** estrae elemento con priorità massima
- **remove(i):** rimuove elemento puntato da iteratore ‘i’
- **findMax():** ritorna elemento con priorità massima
- **changePriority(i, p):** cambia la priorità dell’elemento puntato da ‘i’.

**Complessità:**

In entrambi i casi le operazioni di `insert(p)` e `deleteMax()` avranno un costo di $O(log\ n)$ sia per insert che extract.

| Operations | findMax | insert | deleteMax |
|---|---|---|---|
| Linked List | `O(1)` | `O(n)` | `O(1)` |
| **Binary Heap** | `O(1)` | `O(log n)` | `O(log n)` |
| **Binary Search Tree** | `O(1)` | `O(log n)` | `O(log n)` |

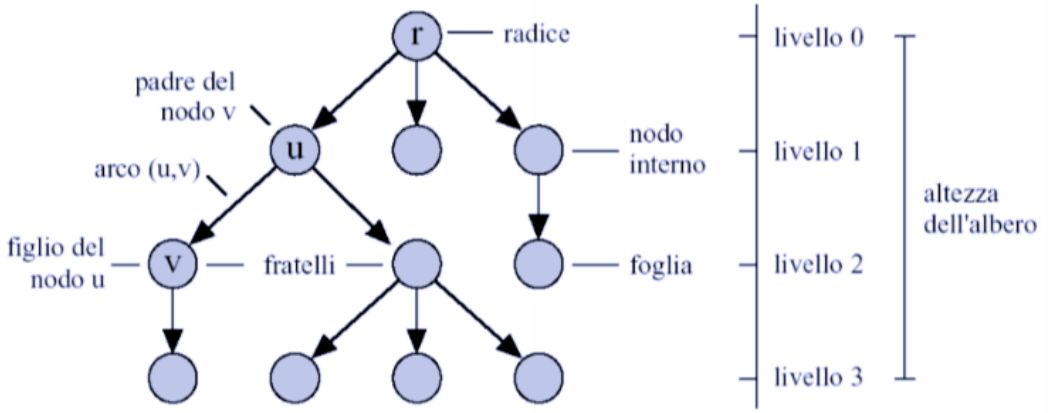
---

# 6. Alberi

Un albero radicato è una coppia T = (N, A) costituita da un insieme N di **nodi** (o “vertici”) e un insieme A (sottoinsieme di N x N) di coppie di nodi dette **archi**.

> Un nodo singolo è un albero (è la radice dell'albero)

- La **profondità** (o livello) di un nodo è il numero di archi che bisogna attraversare per raggiungerlo a partire dalla radice.
- Nodi con lo stesso genitore vengono detti **fratelli** e hanno la stessa profondità.
- **L'altezza** di un albero è la massima profondità a cui si trova una foglia.
- Un albero è un particolare tipo di grafo: è un **grafo connesso minimale** (se si toglie un arco, non è più connesso) o, se vogliamo dare una caratterizzazione in termini della ciclicità, è un grafo **aciclico** massimale (se si aggiunge un arco, non è più aciclico).




Gli alberi sono classificati in base al **numero massimo di ramificazioni** che si dipartono dai nodi. Particolarmente importanti sono gli alberi **binari**, ovvero con nodi che hanno al più due figli.

L'importanza assunta dagli alberi come struttura dati per la rappresentazione di insiemi dinamici risiede nel fatto che la complessità di calcolo delle principali funzioni (inserimento, ricerca e cancellazione) è proporzionale all'altezza dell'albero. In condizioni **ottimali** vale $O(log\ n)$ (n è il numero totale di elementi contenuti).

## Visite di Alberi: DFS e BFS

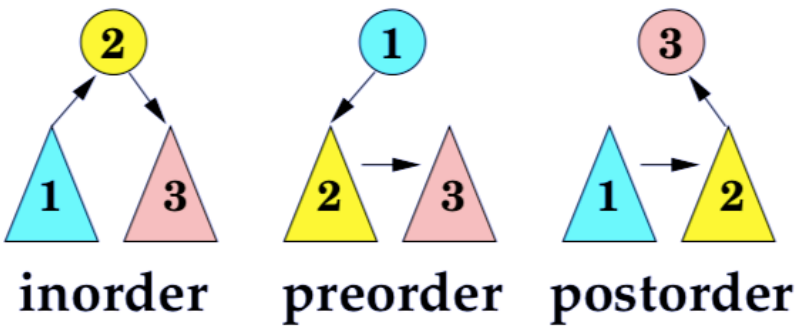
### DFS (Depth First Search)

La visita opera in questo modo: parte dalla radice e procede visitando i nodi di figlio in figlio chiamando se stessa ricorsivamente.

Una volta arrivata ad una foglia retrocede fino al primo antenato che ha ancora dei figli non visitati e ripete lo stesso procedimento fino ad arrivare di nuovo ad un'altra foglia.

```
visitaDFSRicorsiva(nodo r):
 if ( n == null) return null
 visitaNodo();
 visitaDFSRicorsiva(r->sx)
 visitaDFSRicorsiva(r->dx)
```

### Visite ordinate

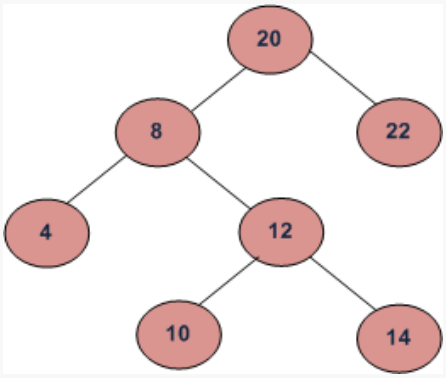


**BFS (Breadth First Search)**

L'algoritmo di visita in ampiezza parte dalla radice e procede visitando nodi per livelli successivi.

Un nodo sul livello "i" può essere visitato solo se tutti i nodi sul livello i-1 sono stati visitati.

**Esempio Visite**:




**BFS restituisce:**
**20 - 8 - 22 - 4 -12 -10 -14**

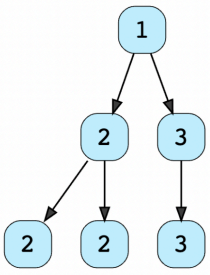
**DFS restituisce:**
**inorder: 4 - 8 - 10 - 12 - 14 - 20 - 22**
**preorder: 20 - 8 - 4 - 12 - 10 - 14 - 22**
**postorder: 4 - 10 - 14 - 12 - 8 - 22 - 20**

# Binary Tree

Gli alberi binari sono un tipo di albero in cui **ogni nodo non ha più di due figli**: figlio sinistro e figlio destro.

Un albero binario può essere:

- **completo**: ogni livello è completamente pieno
- **quasi completo**: ogni livello, tranne eventualmente l'ultimo, è completamente pieno, e tutti i nodi sono il più a sinistra possibile.




**Proprietà**:

> - Un albero binario completo di altezza h ha $2^{h+1} - 1$ nodi
> - Sia T un albero binario quasi completo di altezza h. Allora il numero n di nodi di è tale che 2h ≤ n ≤ 2h+1−1
> - L'altezza di un albero binario quasi completo T con n nodi è $h = log_2\ n$

# Binary Search Tree

Un **albero binario è ordinato se**, ricorsivamente per ogni suo nodo, si verifica che:

- il valore della chiave del nodo è maggiore di tutti i valori delle chiavi contenute nel suo sottoalbero sinistro,
- e minore di tutti i nodi contenuti nel suo sottoalbero destro.

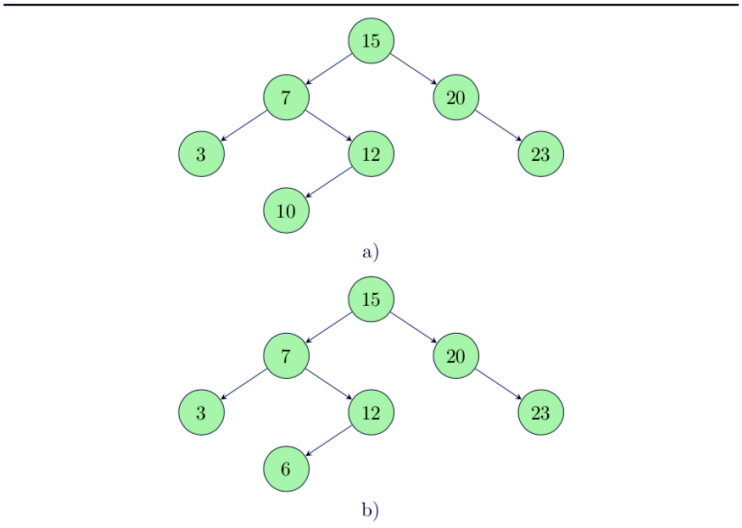



**Figura 7.2:** Due alberi binari: l'albero rappresentato in a) rispetta la condizione di ordinamento; viceversa l'albero b) non la rispetta poichè il sottoalbero con radice 7 ha, nel proprio sottoalbero destro, un valore inferiore a 7 (in particolare il valore 6).

Dato un insieme dinamico con un prefissato insieme di valori, **esistono più alberi binari ordinati** in grado di rappresentarlo.

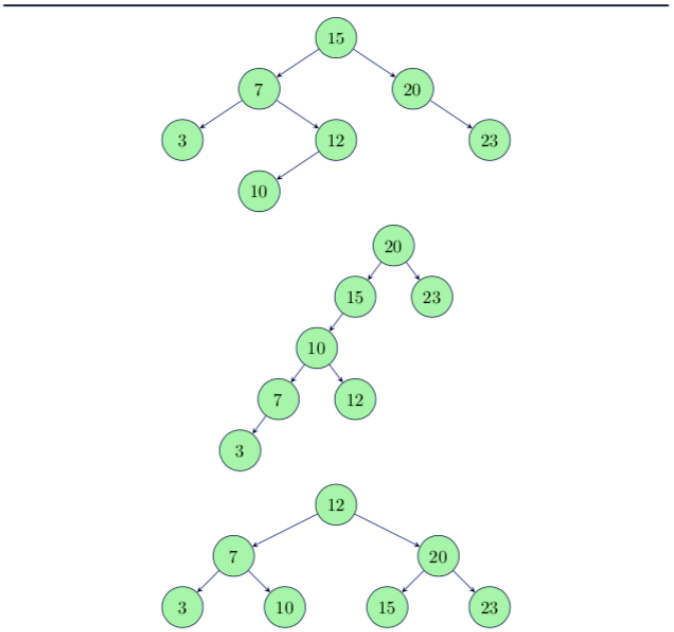
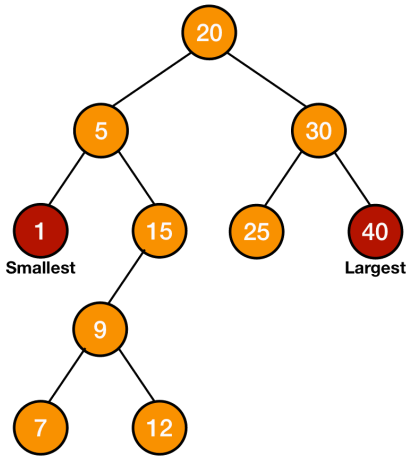



**Figura 7.3:** Tre alberi binari che rappresentano l'insieme dinamico S = {3,7,10,12,15,20,23}; si noti la loro diversità di altezza.

## Max / min di un BST

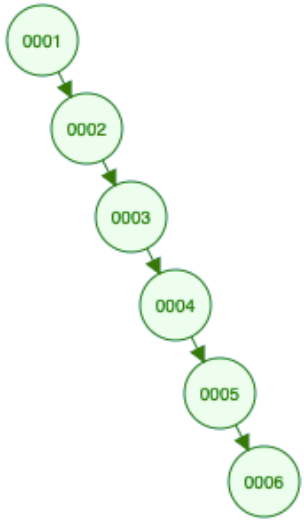
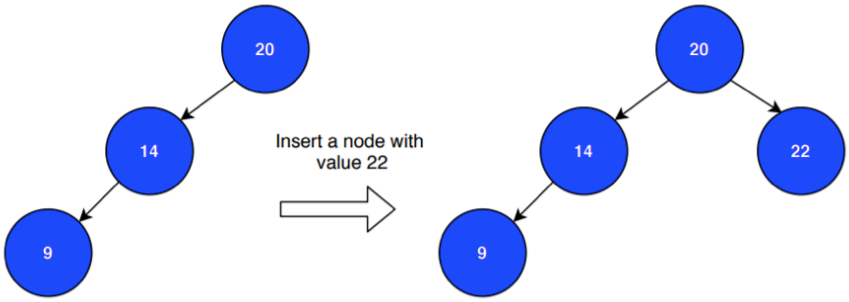
L'elemento minore di un BST è quello più a sinistra, mentre il maggiore è quello più a destra.




## Inserimento di un nuovo nodo

Per inserire un nuovo nodo in un BST è sufficente scorrere l'albero fino a trovare il posto giusto. (Spoiler: sarà una foglia).

Se inserisco una sequenza **ordinata** (1,2,3,4,5,6) avrò una **complessità lineare** in quanto ogni volta devo scorrere tutto l'albero risultante (Worst-case scenario). Altrimenti nel caso medio si ha complessità **logaritmica** O( Log(n) ).

| Esempio worst case : | Esempio best case : |
| --- | --- |
|  |  |





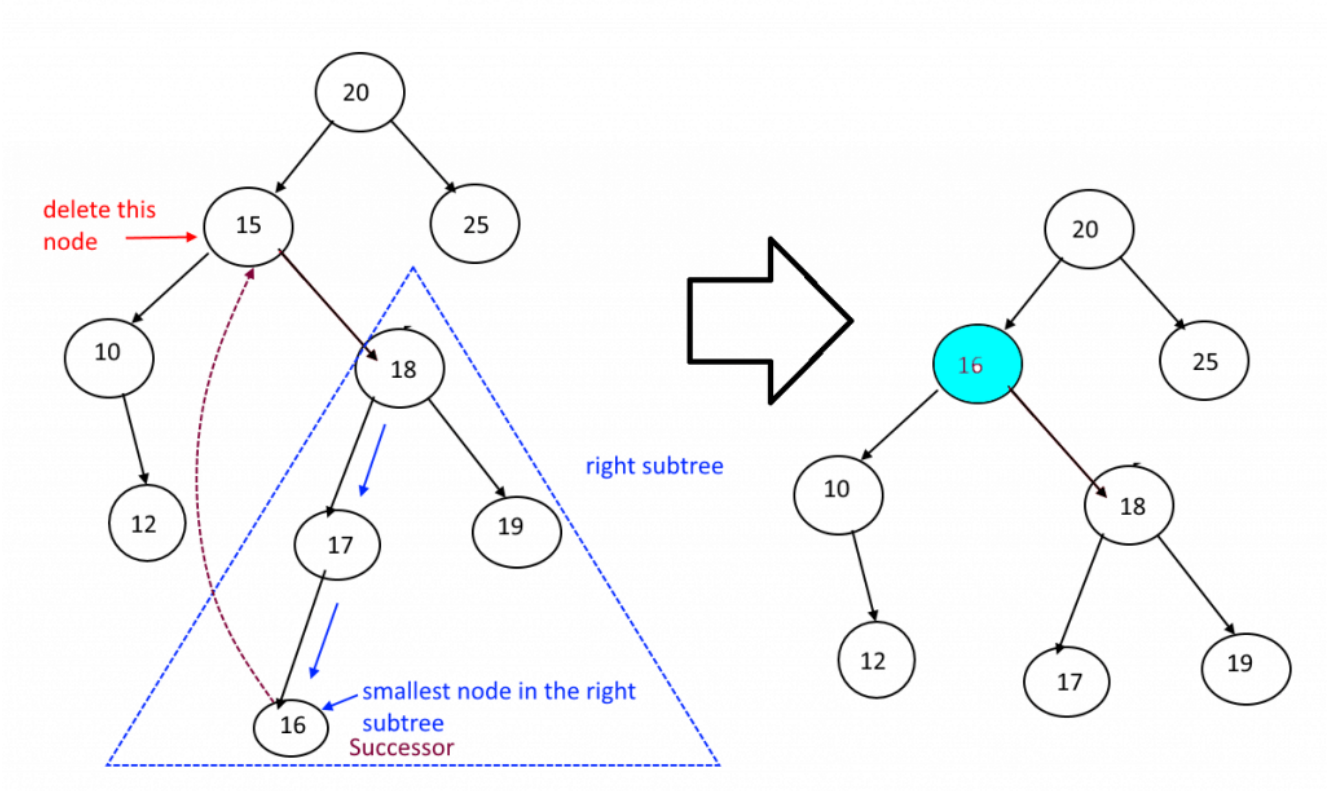
## Cancellazione di un nodo

La cancellazione di un nodo da un albero presenta una maggiore difficoltà rispetto all' inserimento, poiché il nodo da cancellare può occupare posizioni diverse. In particolare si distinguono i seguenti casi:

1. il nodo da cancellare **è una foglia** dell'albero: si cancella la foglia

2. il nodo da cancellare **ha un solo figlio** (indifferentemente il sottoalbero destro o sinistro): cancello il nodo e collego il sottoalbero rimasto all' albero iniziale
3. il nodo da cancellare **possiede entrambi i figli**: dopo aver identificato il nodo da cancellare, provvede ad eseguire i seguenti passi:
    1. la sostituzione del valore presente nel nodo da cancellare, con il valore **minimo del sottoalbero destro** (o  con il max nel sottoalbero sx );
    2. la cancellazione del valore minimo dal sottoalbero destro (o del max dal sottoalbero sx ).

    Esempio (prende il minimo del sottoalbero dx):




## Complessità BST

| Algoritmo | Average | Worst Case |
|---|---|---|
| Search | O(log *n*) | O(*n*) |
| Insert | O(log *n*) | O(*n*) |
| Delete | O(log *n*) | O(*n*) |

**Alberi Rosso/Neri**

Un albero rosso-nero **è un albero binario di ricerca** in cui ad ogni nodo associamo un colore, che può essere rosso o nero. Vincolando il modo in cui possiamo colorare i nodi lungo un qualsiasi percorso che va dalla radice ad una foglia, riusciamo a garantire che l'abero sia **approssivatimente bilanciato**.

Ogni nodo dell'albero ha quattro campi: color, key, left, right ep.

Un RB tree è un albero binario di ricerca che soddisfa le seguenti proprietà:

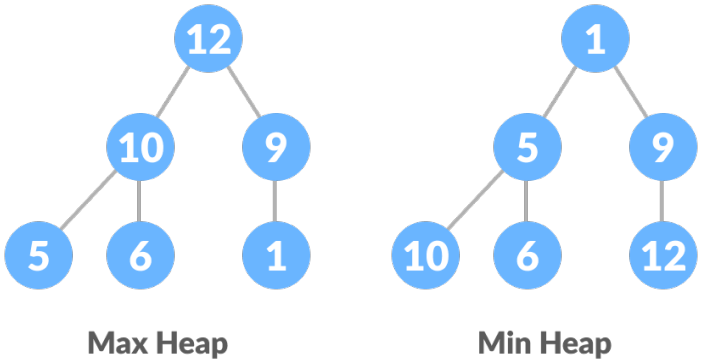
1. ogni nodo è rosso o nero
2. la radice è nera
3. ogni foglia è nera
4. se un nodo è rosso, entrambi i suoi figli devono essere neri
5. per ogni nodo n, tutti i percorsi che vanno da n alle foglie sue discendenti contengono lo stesso numero di nodi neri

L' **altezza massima** di un RB tree con n nodi interni e' **2*log(n + 1)**.

# Heap

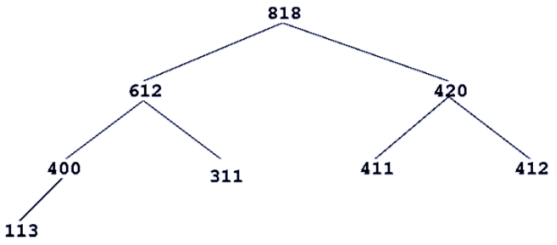
Struttura dati basata sugli alberi utilizzata principalmente per l'ordinamento e l'**implementazione delle code prioritarie**. Sono alberi binari completi che hanno le seguenti caratteristiche:

- Ogni livello viene riempito tranne i nodi foglia (i nodi senza figli sono chiamati foglie).
- Tutti i nodi nell'albero seguono la proprietà che sono maggiori dei loro figli, cioè l'elemento più grande è alla radice e entrambi i suoi figli sono più piccoli della radice e così via (max heap). Se invece tutti i nodi sono più piccoli dei loro figli, si parla di min-heap
- Tutti i nodi sono il più a sinistra possibile, questo significa che ogni bambino è alla sinistra del suo genitore.




> A differenza dei BST, gli heap non devono per forza avere un ordinamento. Vedi l'immagina sopra: sono entrambi heap validi ma NON sono binary search tree.

Uno heap binario si può rappresentare come array in cui i figli di un nodo alla posizione *x* sono memorizzati nelle posizioni ***2x*** **e** ***2x+1.***




Array H

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 818 | 612 | 420 | 400 | 311 | 411 | 412 | 113 |

**Complessità:**

- findMax(pq) –> restituisce l'elemento max (array[0] ) –> $O(1)$
- Insert(elem, key, pq) –> inserisce l'elemento nell' heap (chiama funzione `muoviAlto()` per sistemare la posizione) –> $O(log_2\ n)$
- deleteMax(pq) –> elimina il massimo (chiama funzione `muoviBasso()` per sistemare la posizione) –> $O(log_2\ n)$

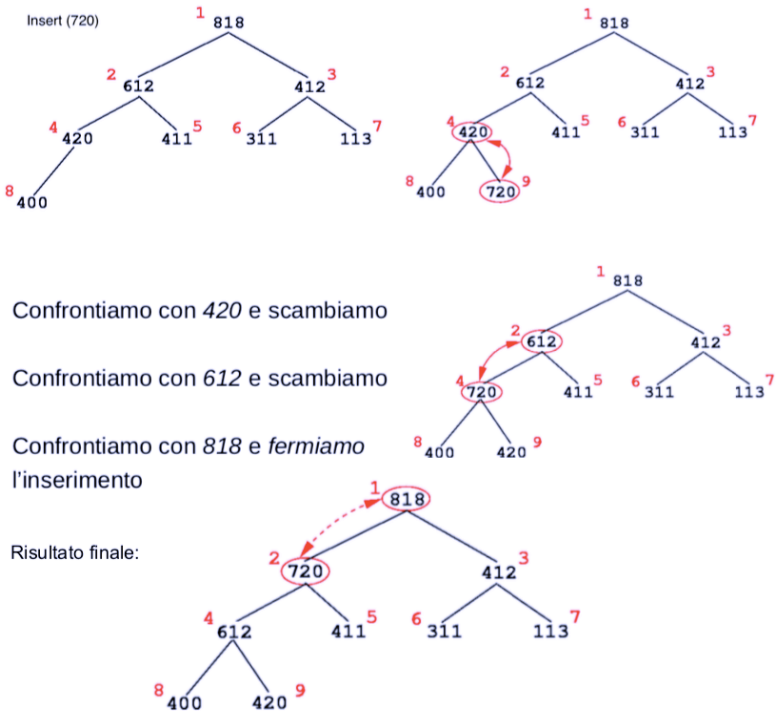
## Inserimento in un heap

Si inserisce **(k, e)** come **ultimo elemento** dello heap (come ultimo elemento dell'array H). La proprietà dell'ordinamento a heap viene ripristinata spingendo il nodo **v** che contiene **(k, e)** verso l'alto tramite ripetuti scambi di nodi.

Procedura **muoviAlto(v, pq)**

```
while(v != radice(pq) && chiave(v) > chiave(padre(v)){
  scambia v e padre(v) in pq (scambiandoli di posto nell'array H) }
```

**Esempio**:



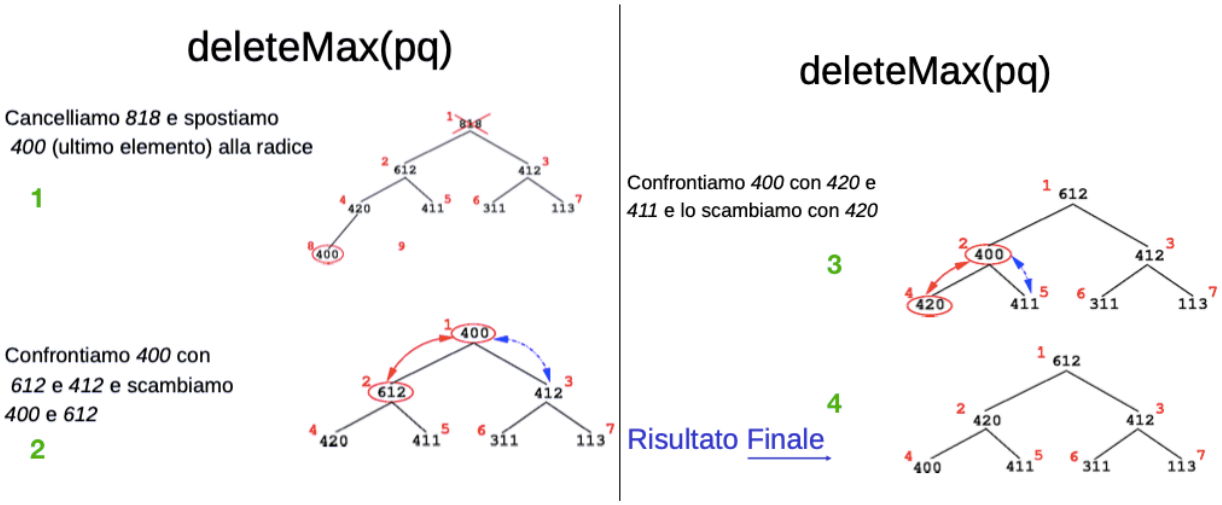

## Delete in un heap

1. Sostituiamo la radice dell'albero con l'ultimo elemento (l'elemento in posizione n dell'array H che codifica lo heap)
2. muoviBasso(v, pq)

Procedura **muoviBasso(v, pq)**

```
while( v ha almeno un figlio con chiave > chiave(v))
  do sia u il figlio di v di chiave massima
  scambia v e u in T
```

**Esempio**:



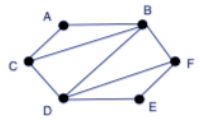
# 7. Grafi

Un grafo è un insieme di Nodi e Vertici definito come G = (V,E) (graph = vertex, edges). I vertici possono essere **pesati**.

Indichiamo con **grado** il **numero di archi** di un vertice e differenziamo:

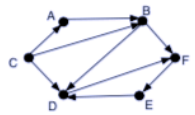
- Grafo **non** orientato: grado = numero archi → Somma dei gradi è 2 volte il numero di archi
- Grafo orientato:
  - gradoEntrante = numero archi entranti
  - gradoUscente = numero archi uscenti

Dato il seguente grafo non orientato:




- l'arco $(A, B)$ è *incidente* sui nodi $A$ e $B$,
- i nodi $A$ e $B$ sono *adiacenti*, $A$ è *adiacente* a $B$, $B$ è *adiacente* ad $A$,
- i nodi adiacenti a un nodo $A$ si chiamano anche i *vicini* di $A$,
- il *grado* $\delta(u)$ di un nodo $u$ è il numero di archi incidenti sul nodo, per esempio $\delta(B) = 4$.

Dato il seguente grafo orientato:




- l'arco $(A, B)$ è *incidente* sui nodi $A$ e $B$, *uscente* da $A$, *entrante* in $B$,
- il nodo $B$ è *adiacente* ad $A$, ma $A$ non è adiacente a $B$,
- i nodi adiacenti a un nodo $A$ si chiamano anche i *vicini* di $A$,
- il *grado* $\delta(u)$ di un nodo $u$ è il numero di archi incidenti sul nodo, per esempio $\delta(B) = 4$,
- il *grado uscente* (*outdegree*) $\delta_{out}(u)$ di un nodo $u$ è il numero di archi uscenti dal nodo, per esempio $\delta_{out}(B) = 2$,
- il *grado entrante* (*indegree*) $\delta_{in}(u)$ di un nodo $u$ è il numero di archi entranti nel nodo, per esempio $\delta_{in}(B) = 2$.

**Implementazioni e complessità grafi:**

- Liste di adiacenza (2):

**con vertici memorizzati in array**

| *Operazioni* | *Caso Migliore* | *Caso Peggiore* |
|---|---|---|
| *degree* | *Θ(δ(v))* | *Θ(δ(v))* |
| *incidentEdges* | *Θ(δ(v))* | *Θ(δ(v))* |
| *areAdjacent* | *Θ(1)* | *Θ(min(δ(x), δ(y)))* |
| *addVertex* | *Θ(1)* | *Θ(n) se devo riallocare* |
| *addEdge* | *Θ(1)* | *Θ(min(δ(x), δ(y))) chiama areAdjacent per vedere se l'arco c'è già* |
| *removeVertex* | *Θ(δ(v))* | *Θ(m)* |
| *removeEdge* | *Θ(δ(x) + δ(y))* | *Θ(δ(x) + δ(y))* |

**con vertici memorizzati in lista**

| *Operazioni* | *Caso Migliore* | *Caso Peggiore* |
|---|---|---|
| *degree* | *Θ(n)* | *Θ(n)* |
| *incidentEdges* | *Θ(n)* | *Θ(n)* |
| *areAdjacent* | *Θ(n)* | *Θ(n)* |
| *addVertex* | *Θ(n)* | *Θ(n)* |
| *addEdge* | *Θ(n)* | *Θ(n)* |
| *removeVertex* | *Θ(m+n)* | *Θ(m+n)* |
| *removeEdge* | *Θ(n)* | *Θ(n)* |

- matrice di adiacenza

| Operazioni | Caso Migliore | Caso Peggiore |
|---|---|---|
| degree | $\Theta(n)$ | $\Theta(n)$ |
| incidentEdges | $\Theta(n)$ | $\Theta(n)$ |
| areAdjacent | $\Theta(1)$ | $\Theta(1)$ |
| addVertex | $\Theta(n)$ | $\Theta(n^2)$ richiede riallocazione |
| addEdge | $\Theta(1)$ | $\Theta(1)$ |
| removeVertex | $\Theta(n^2)$ | $\Theta(n^2)$ |
| removeEdge | $\Theta(1)$ | $\Theta(1)$ |

## Visite di un grafo

### Ampiezza - BFS (Breadth First Search)

Si parte dal nodo su cui viene chiamata, si pone il booleano visitato = true, lo si inserisce nella coda e lo si pone come radice dell'albero di ricoprimento.

Dopodichè si procede con la *dequeue* fino a quando la coda non è vuota e per ogni vertice che faccio uscire aggiungo i suoi vertici adiacenti, che hanno il booleano visitato = false, alla coda e li pongo come suoi figli nell'albero.

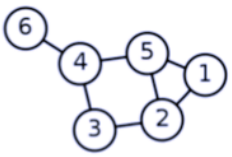
Ripeto il procedimento fino a quando tutti i vertici saranno stati visitati.

```
BFS(G,s): //visita nodi di G raggiungibili a partire da s
  for each (u nodo in G):
    marca u come bianco;

  parent[s] = null
  Q = coda vuota
  Q.add(s); marca s come grigio;

  while (Q non vuota)
    u = Q.remove() //u non nero
    visita u
    for each ((u,v) arco in G):
      if (v bianco)
        marca v come grigio;
        Q.add(v);
        parent[v]=u
    marca u come nero
```

Esempio:




prendendo come esempio il grafo qua sopra:

Chiamo la visita sul vertice 6.

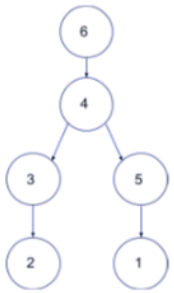
Inserisco il vertice 6 nella coda, pongo il suo booleano = true e lo metto come radice dell'albero. Procedo con la dequeue, faccio uscire il 6, inserisco nella coda il 4, lo pongo come visitato e lo metto come figlio di 6 nell'albero.

Faccio di nuovo la dequeue, faccio uscire il 4, inserisco nella coda il 3 e il 5, lo pongo come visitati e li metto come figli di 4 nell'albero. Continuo fino a quando non ho visitato tutti i vertici del grafo.

Ordine in cui entrano nella coda:

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

L'albero è il seguente:



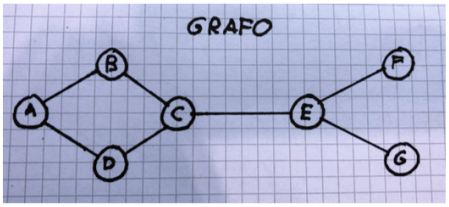

**Osservazione di base**:

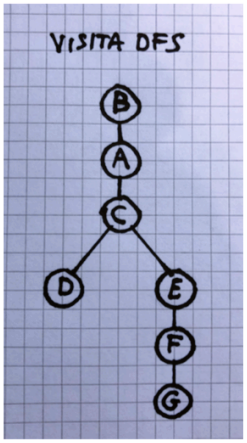
- Ogni vertice è marcato al più una volta, **aggiunto alla coda una volta**, cancellato dalla coda una volta
- Quando utilizziamo una rappresentazione del grafo come **liste di adiacenza** (con elementi della lista di adiacenza che mantengono il puntatore al vertice) la **complessità temporale è pari a *Θ(n+m)***
- Quando utilizziamo una rappresentazione del grafo come **matrice di adiacenza la complessità temporale è pari a *Θ(n^2)***

### Profondità - DFS (Depth First Search)

```
visitaDFSricorsiva (vertice v, albero T){
  marca e visita il vertice v
  foreach (arco (v,w)) do
  if (w non è marcato) then
    aggiungi l'arco (v,w) all albero T
}

visitaDFS (vertice s) → albero{
  T ← albero vuoto // rendi tutti i vertici non marcati
  visitaDFSricorsiva(s,T)
  return T;
}
```




É analoga alla visita in preordine di un albero.

- rappresentazione del grafo come **liste di adiacenza *Θ(n+m)***
- rappresentazione del grafo come **matrice di adiacenza *Θ (n^2)***

## Algoritmo di Dijkstra (bandiera olandese)

Dato un grafo pesato e con costi NON negativi l'algoritmo permette di trovare il **camminio minimo ottimale** da un nodo iniziale a uno finale. É simile ad una visita in ampiezza (BFS) in cui a ogni passo:

- se il nodo è nero (già visitato), conosco la sua distanza
- tutti i nodi non visitati hanno una distanza provvisoria
- si estrae il nodo con distanza minima
- si aggiornano le distanze provvisorie dei nodi adiacenti al nodo estratto, tenendo conto del nuovo arco

```
Djikstra(G,s):
  for each (u nodo in G)
    dist[u] = ∞ // tutti i nodi sono bianchi
  parent[s] = null; dist[s] = 0 // s diventa grigio

  Q = heap vuoto
  for each (u nodo in G)
    Q.add(u, dist[u])

  while (Q non vuota):
    u = Q.getMin() //estraggo nodo a distanza provvisoria minima, u diventa nero
    for each ((u,v) arco in G) //v diventa o resta grigio
      if (dist[u] + cu,v < dist[v])// controllo se la distanza provv. si può migliorare
        parent[v] = u;
        dist[v] = dist[u] + cu,v
        Q.changePriority(v, dist[v]) //moveUp
```

Per semplicità ci conviene rappresentare l'insieme dei nodi ancora da visitare con una **coda a priorità**.

```
1. Assegno a tutte le distanze "infinito"
2. Assegno a sorgente (S) la distanza 0 e inizio a creare l'albero dei cammini minimi
(parent[])
3. Creo coda Heap e ci aggiungo tutti i nodi del grafo con le rispettive distanze
provvisorie
4. WHILE: la coda non è vuota:
  - estraggo il minimo
  - considero tutti i nodi adiacenti (for)
    - controllo se la distanza provvisoria si può migliorare e nel caso aggiorno dist[]
e parent[] oltre che cambiare lo heap (changePriority())
```

**Complessità Dijkstra**

Se la coda a priorità è realizzata come sequenza non ordinata, ogni inserimento in coda o modifica della priorità ha complessità O(1), ma l'estrazione del minimo ha complessità O(n), **quindi la complessità dell'algoritmo è $O(n^2)$.** Analogamente se la coda a priorità è realizzata come sequenza ordinata. Nella versione di Johnson, se n e m sono il numero rispettivamente dei nodi e degli archi, si ha:

- inizializzazione di tutti i nodi come bianchi: O(n)
- n estrazioni dallo heap: O(n log n)
- ciclo interno: ogni arco viene percorso una volta, e per ogni nodo adiacente si ha eventuale moveUp nello heap, quindi O(m log n)

  **Complessivamente si ha quindi O((m + n) log n).** Si noti che se il grafo è denso, cioè $m = O(n^2)$, la complessità diventa $O(n^2 logn)$, quindi peggiore della versione originale quadratica.

**Complessità nel dettaglio**:

1° `foreach` –> costo: `N`

2° `foreach` –> cost: `n log n` [logn viene da `add()`; mentre n dal `foreach`]

`Q.getMin()` —> costo: `n log n` [logn viene da `getMin()`; mentre n dal `foreach`]

`Q.changePriority()` —> costo: M log n [ $logn$ per operazioni su heap; mentre M dal `foreach` numero archi]

---

# 8. Tabelle di Hash

Le hash table son state create per superare i problemi delle tabelle ad **accesso diretto**, ossia quelle in cui ogni "chiave" indica l' **indice** dell'array in cui verrà salvata l'informazione –> `v[chiave] = dato`. Notiamo come la chiave dell'elemento deve essere un intero contenuto tra 0 e la dim dell'array.

Le tabelle ad accesso diretto permettono di avere tempi di `inserimento, ricerca e cancellazione` molto ridotti, nell'ordine di O(1), a discapito dello spazio che sarà " O(dim universo chiavi) " (Attenzione: non numero elementi).

> **Fattore di Carico**: indichiamo con $\alpha$ il fattore di carico di una hash table ossia quante celle sono utilizzate sul totale delle celle disponibili –> $\alpha = \frac{numero\ elem\ dizionario}{dimensione\ tabella} = \frac{n}{m}$.
>
> Esempio: tabella con i nomi delle 100 matricole che hanno ASD nel piano di studi, indicizzati da numeri di matricola a 6 cifre n=100 m=10^6 = 0,0001 = 0,01% → Grande spreco di memoria!

Per ovviare agli inconvenienti delle tabelle ad accesso diretto ne consideriamo un' estensione: le **tabelle di hash**.

**Idea:**

- Chiavi prese da un universo totalmente ordinato U (possono non essere numeri)
- Funzione hash: h: U → [0, m-1] (funzione che trasforma chiavi in indici)
- Elemento con chiave **k** in posizione **v[h(k)]**

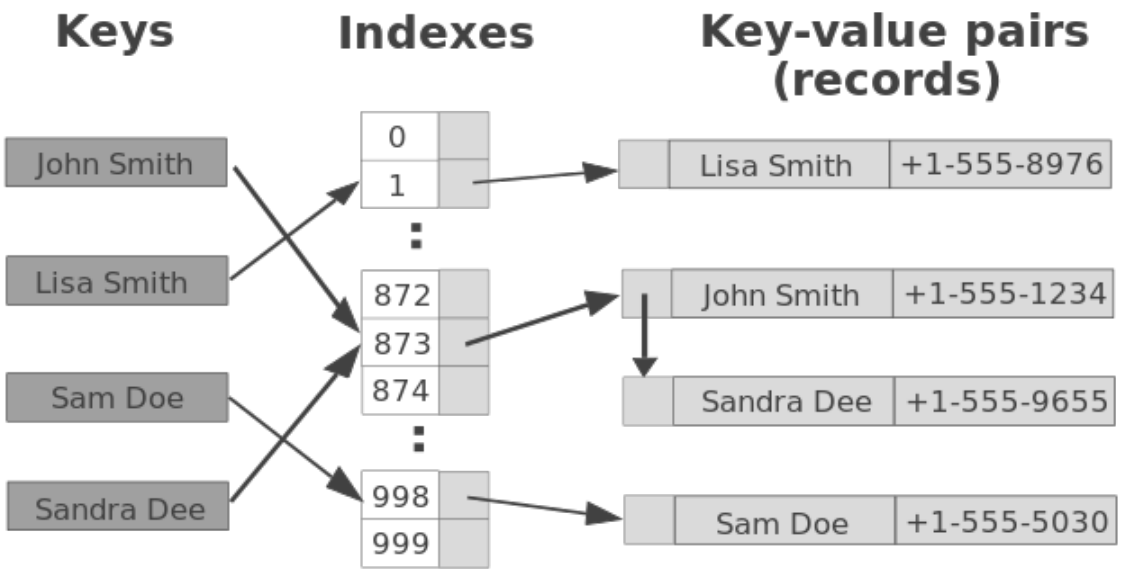
Serve quindi una **funzione di hash** che data una chiave la trasforma in un indice dell'array. Ovviamente la funzione di hash deve essere calcolabile in tempo costante.

> **Funzione di Hash:**
>
> - **Perfette**: sono funzioni **iniettive**, ossia che per ogni input dato, restituiscono un output diverso –> non ho problemi di collisione
> - **non perfetta**: affinchè una funzione sia "buona" deve soddisfare 2 proprietà:
>   - essere calcolabile in tempo costante
>   - essere uniformemente distribuita

Attenzione però alle **collisioni,** infatti se una funzione di hash non è perfetta, può capitare che h(a) == h(b) . Per ovviare a questo problema si usano **funzioni H iniettive!**

Nel caso in cui non si possano evitare le **collisioni**, dobbiamo trovare un modo per risolverle. Due metodi classici sono i seguenti:

- **liste di collisione**: Gli elementi sono contenuti in liste esterne alla tabella (chiamate bucket): v[i] punta alla lista degli elementi tali che h(k)=i. Esempio:



- **indirizzamento aperto**: nel caso in cui la posizione h(k) in cui inserire una chiave k sia già occupata, il metodo prevede di posizionarla in un'altra cella vuota, anche se quest'ultima potrebbe spettare di diritto ad un'altra chiave. Le operazioni vengono realizzate come segue:
    - `Inserimento`: se v[h(k)] è vuota, inserisci la coppia (el, k) in tale posizione; altrimenti, a partire da h(k), ispeziona le celle della tabella e inserisci nella prima cella vuota;
    - `Ricerca`: se, durante la scansione delle celle, ne viene trovata una con la chiave cercata, restituisci l'elemento trovato; altrimenti, se si arriva a una cella vuota o si è scandita l'intera tabella senza successo, restituisci null.
    - `Cancellazione`: affinché la ricerca con il metodo appena descritto funzioni, occorre adottare una strategia particolare per la cancellazione, ossia utilizzare un valore speciale "DELETEDELEM" nel campo el dell'elemento che si vuole rimuovere: in particolare, l'inserimento tratterà tale cella come vuota e si fermerà su di essa, mentre la ricerca la oltrepasserà.

  **Esempio:**

  Supponiamo che m = 8 e le chiavi siano "a, b, c, d" (associate a valori che per l'esempio non ci interessano), con hash value h(a)=3, h(b)=0, h(c)=4 e h(d)=3.

  Usiamo la strategia di rehashing più semplice possibile, il **rehashing lineare**, t.c.
  $h_i(x) = (h(x) + i)\ mod\ m$

**Complessità:**

**Liste di Collisione**

| *Operazioni* | *Caso Migliore* | *Caso Peggiore* |
|---|---|---|
| *insert(elem e, chiave k)* | *Θ(1)* | *Θ(1+n/m)* |
| *delete(chiave k)* | *Θ(1)* | *Θ(1+n/m)* |
| *search(chiave k) → elem* | *Θ(1)* | *Θ(1+n/m)* |

# 9. Set (insieme) & BitVector

Il **set** è un tipo di dato astratto, consistente in una collezione di **valori disposti in ordine casuale e senza ripetizioni.** Oltre alle operazioni classiche di inserimento e cancellazione, il tdd set contiene quelle operazioni che si possono fare su un insieme matematico come: `union, intersection, difference, subset`.

Oltre alle strutture dati "ovvie" (liste collegate, liste semplici, array), una struttura adatta all'implementazione degli insiemi è il **bit vector** (anche detto array vector o bitmap).

**Complessità:**

| | **Insert() [Tbest / Tworst]** | **Delete() [Tbest / Tworst]** | **Union() [Tbest / Tworst]** | **Intersection() [Tbest / Tworst]** |
|---|---|---|---|---|
| **bitVector** | **Θ(1) / Θ(1)** | **Θ(1) / Θ(1)** | **Θ(n) / Θ(n)** | **Θ(n) / Θ(n)** |
| Liste | Θ(1) / Θ(n) | Θ(1) / Θ(n) | Θ(max(n, m, n*m)) / Θ(max(n, m, n*m)) | Θ(n*m) / Θ(n*m) |
| Array | Θ(1) / Θ(n) | Θ(1) / Θ(n) | Θ(max(n, m, n*m)) / Θ(max(n, m, n*m)) | Θ(n*m) / Θ(n*m) |

## BitVector

I bitvector sono degli array di dimensione di una "parola" del computer (8, 16, 32, 64 bit). Quindi invece di usare una "parola" per ogni elemento dell'insieme, tramite i bitvector possiamo usare una "parola" sola per rappresentare tutti gli elementi.

Posso rappresentare per esempio, l'insieme dei numeri primi dove in ogni casella rappresentante un numero da 1 a 32 assegno 0 se il numero non è primo e 1 se lo è.

Oppure posso rappresentare l'insieme {0, 1, 17, 30} con: `01000000000000100000000000000011`

Grazie ai bitvector le operazioni del tdd Set sono implementabili come:

- Intersezione:  si fa l'AND binario tra x e y (`x & y`)
- Unione: si fa l'OR binario (`x | y`)
- Differenza: si usa AND NOT (`x & ~y`)
- subset: si usa XOR (`x ^ y`)