

Introduzione agli Algoritmi

Introduzione

Calcolo Tempistiche

Problem Solving

Teorema del Costo Uniforme

Nozione asintotica

La notazione O

Dimostrazione per induzione

La notazione Ω

La notazione Θ

Calcolo notazione asintotica tramite limite

Algebra della relazione asintotica

Regola 1 - costanti moltiplicative

Regola 2

Regola 3

Alcune sommatorie notevoli

Valutazione del costo computazionale

Costo delle istruzioni

Istruzioni elementari

Blocchi if/else

Blocchi iterativi

Casistiche

Ricerca binaria

La Ricorsione

Introduzione

Equazioni di ricorrenza

Metodo iterativo

Metodo dell'albero

Metodo di sostituzione

Metodo principale

Enunciato

Esempio di applicazione del primo caso

Esempio di applicazione del secondo caso

Esempio di applicazione del terzo caso

Esempio di impossibilità di applicazione

Algoritmi di sorting

Il problema dell'ordinamento

Insertion sort

Pseudo codice

Selection sort

Pseudo codice

Bubble sort

Pseudo codice

La complessità dell'ordinamento

Merge Sort

Quick Sort

Heap Sort

La struttura dati Heap

Funzione Heapify

Funzione BuildHeap

Ordinamenti Lineari

Counting Sort

Counting Sort con dati satellite

Bucket Sort

Strutture dati

Cos'è una struttura dati

Strutture dati fondamentali - Insieme dinamico

Lista puntata

operazioni sulla lista puntata

Liste doppiamente puntate

Pila

Coda

Implementazione con Array

Code e Pile con Priorità

Alberi

Alberi Radicati

Rappresentazione in memoria

Rappresentazione tramite record e puntatori

Rappresentazione Posizionale

Rappresentazione con Vettore dei Padri

Confronto tra tipi di rappresentazione degli alberi

Visite ad alberi

Applicazioni delle visite

Visita per livelli

I dizionari

Tabelle ad indirizzamento diretto

Tabelle Hash

Alberi binari di ricerca

Ricerca

Inserimento

Predecessore e Successore

Cancellazione

Bilanciamento dell'altezza

Alberi Rosso - Neri

B-Altezza

Operazioni su alberi RB

Introduzione

Calcolo Tempistiche

Un algoritmo è una sequenza di comandi elementari ed univoci che terminano in un *tempo finito* ed operano su *strutture dati* (quest'ultime memorizzano i dati semplificandone l'accesso e la modifica).

Ci sono diverse strutture dati ed hanno diverse applicazioni, ognuna di esse è utilizzata per risolvere specifici problemi meglio di altre.

Un algoritmo deve essere *efficiente*, è quindi necessario che finisca la computazione in un tempo "ragionevole", l'obiettivo è risolvere un problema nel minor tempo possibile, quindi ci occuperemo del cosiddetto *costo computazionale* in termini di tempo e spazio.

Esempio :

Dobbiamo ordinare $n = 10^6$ numeri interi. Abbiamo due calcolatori :

- Il calcolatore V (veloce) effettua 10^9 operazioni al secondo.
- Il calcolatore L (lento) effettua 10^7 operazioni al secondo.

Abbiamo poi due diversi algoritmi di ordinamento :

- L'algoritmo IS (Insertion sort) richiede $2n^2$ operazioni
- L'algoritmo MS (Merge sort) richiede $50\log_2(n)$ operazioni

Se dovessimo utilizzare l'algoritmo più lento sul calcolatore più rapido, e l'algoritmo più veloce sul calcolatore più lento, la maggiore efficienza di V riuscirebbe a bilanciare la minor efficienza dell'algoritmo IS? Vediamo con il calcolo dei tempi :

$$\text{tempo V (IS)} = \frac{2(10^6)^2 \text{ operazioni algoritmo}}{10^9 \text{ operazioni al secondo}} = 33 \text{ minuti}$$

$$\text{tempo L (MS)} = \frac{50 * 10^6 * \log_2(10^6) \text{ operazioni algoritmo}}{10^7 \text{ operazioni al secondo}} = 1.5 \text{ minuti}$$

È chiaro che un algoritmo più veloce performi meglio anche su un computer lento, rispetto ad un algoritmo lento su un computer più veloce.

Adesso aumentiamo la quantità di dati da computare a

Un algoritmo più lento, all'aumentare dei dati da computare, aumenta esponenzialmente il suo tempo di esecuzione.

Adesso aumentiamo la quantità di dati da computare a 10^7 :

$$\text{tempo V (IS)} = \frac{2(10^7)^2 \text{ operazioni algoritmo}}{10^9 \text{ operazioni al secondo}} = 2 \text{ giorni}$$

$$\text{tempo L (MS)} = \frac{50 * 10^7 * \log_2(10^7) \text{ operazioni algoritmo}}{10^7 \text{ operazioni al secondo}} = 20 \text{ minuti}$$

Un algoritmo più lento, all'aumentare dei dati da computare, aumenta esponenzialmente il suo tempo di esecuzione.

Problem Solving

“Passare da un problema alla soluzione” Approccio al problem solving :

1. Analisi del problema, comprensione e identificazione del problema

2. Esplorazione degli approcci possibili tramite i metodi noti
3. Selezionare un approccio
4. Definizione dell'algoritmo risolutivo, identificando i dati
5. Riflessione critica, ripensando alla soluzione proposta.

Un algoritmo si può definire *corretto* se per ogni istanza di un problema computazionale, termina con l'output corretto.

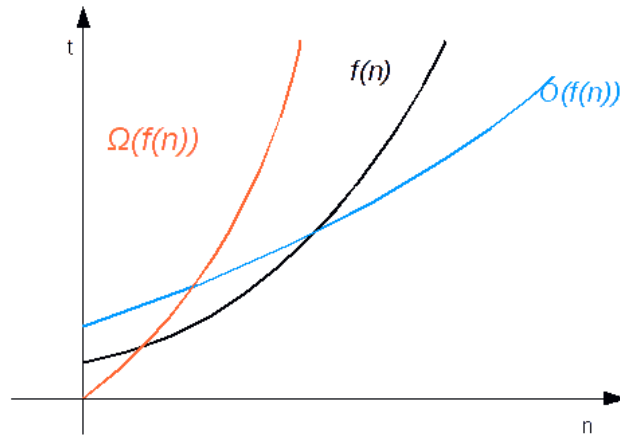
Inoltre per valutare l'efficienza di un algoritmo, dobbiamo analizzarlo senza che l'analisi sia influenzata da una specifica tecnologia. Usiamo quindi una macchina teorica, **la Random access machine**, ha un singolo processore e svolge operazioni elementari, ciascuna di esse richiede per definizione un tempo costante. Esiste un limite alla dimensione di ogni valore memorizzato ed al numero complessivo di valori utilizzati.

Teorema del Costo Uniforme

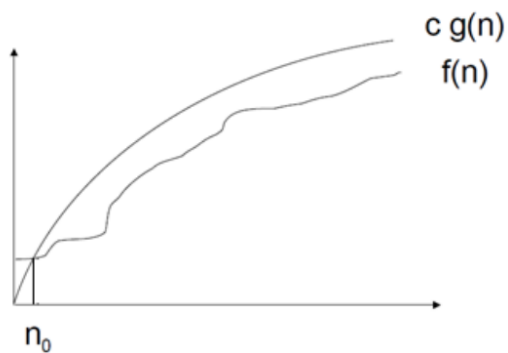
Se ogni dato in input sia un valore minore di $k = 2^{\text{numero di bit della parola di memoria}}$, ciascuna operazione elementare sui dati del problema verrà eseguita in un tempo costante, in tal caso si parla di misura di costo uniforme. In un contesto realistico però, è comune che un dato sia più grande del valore k , si usano quindi più parole.

Nozione asintotica

È importante poter confrontare correttamente due algoritmi valutandone l'efficienza con degli strumenti matematici, ci interessa osservare come si comporta un algoritmo all'aumentare dei dati, si utilizza la nozione asintotica per studiarne il tasso di crescita.



La notazione O



La notazione O (si legge “o grande”) è il limite superiore asintotico, prendendo in considerazione due funzioni $f(n)$ e $g(n)$ diremo che $f(n)$ è in $O(g(n))$ se esistono costanti c e n_0 tali che :

$0 \leq f(n) \leq c * g(n)$ per ogni $n \geq n_0$.
È importante che $f(n)$ da un certo punto in poi sia definitivamente sotto $g(n)$, quindi $f(n)$ è in $O(g(n))$.

Esempio :

$f(n) = 3n + 3$ è in $O(n^2)$?

ponendo una c come $c = 6$, $c * n^2 \geq 3n + 3$ per ogni $n \geq 1$.



data una funzione $f(n)$, esistono infinite funzioni $g(n)$ per cui $f(n)$ risulta in $O(g(n))$.

Dimostrazione per induzione

Poniamo ora un generico polinomio di grado m :

$$f(n) = \sum_{i=0}^m a_i n^i \text{ con } a_m > 0$$

Dimostriamo per induzione che $f(n)$ è in $O(g(n))$.

Caso base :

$$m = 0, \quad f(n) = a_0 \quad O(n_0) = 1$$

a_0 è in $O(1)$ per ogni n e per ogni $c \geq 0$.

Ipotesi induttiva :

$$\sum_{i=0}^k a_i n^i \text{ è in } O(n^k) \text{ per ogni } k < m, \text{ cioè esiste } c \text{ per ogni } m \geq n_0.$$

Passo induttivo :

$$f(n) + h(n) \leq a_m n^m + \sum_{i=0}^{m-1} a_i n^i = a_m n^m = h(n)$$

Adesso abbiamo :

$$f(n) + h(n) \leq a_m n^m + c' n^k \leq a_m n^m + c' n^m = (c' + a_m) n^m$$

Ponendo $c = c' + a_m$ si ha la tesi.

Esempio :

$$f(n) = \log_2(n) \text{ è in } O(\sqrt{n}) ?$$

In generale :



$$\log_2^a(n) \text{ è in } O(n^{1/b}) \text{ per ogni } a, b \geq 1$$



$$f(n) = n^{1/a} \text{ è in } O(n) \text{ per ogni } a \geq 2$$



$$f(n) = n^2 \text{ è in } O(b^n) \text{ per ogni } b \geq 2$$

Gerarchie :

Ogni poli-logaritmo è dominato da ogni radice, ogni radice è dominata da un ogni polinomio, ed ogni polinomio è sempre dominato da un esponenziale.

Quindi :

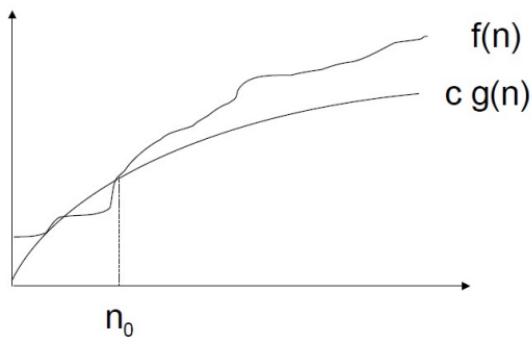
$$\log_2^a(n) < n^{1/a} < n < a^n$$

La notazione Ω

Non è altro che il complementare dell' O , quindi il suo contrario :

$f(n)$ è in $\Omega(g(n))$ se esiste una costante c e n_0 tale che :

$f(n) \geq c * g(n)$ per ogni $n \geq n_0$.



$f(n)$ domina $g(n)$, è quindi in $\Omega(g(n))$,
analogamente $g(n)$ è in $O(f(n))$.

Esempio :

Data $f(n) = 2n^2 + 3$ si ha $f(n)$ è in $\Omega(n)$

Considerazioni :

$O(g(n))$ e $\Omega(f(n))$ sono insiemi di funzioni, ma è comunque comune utilizzare la scrittura $f(n) = O(g(n))$ per intendere che $f(n)$ è in $O(g(n))$.

La notazione Θ

Data una certa funzione $f(n)$, vogliamo trovare quella più in basso tra tutte le funzioni sopra $f(n)$, e quella più in alto tra tutte le funzioni sotto $f(n)$. Solitamente queste due funzioni sono

la stessa, difatti, se abbiamo due funzioni $f(n)$ e $g(n)$, diciamo che $f(n)$ è in $\theta(g(n))$ se abbiamo 3 costanti c_1, c_2 ed n_0 tali che :

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ per ogni } n \geq n_0.$$

Quindi se vale sia $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$ Allora $f(n) = \Theta(g(n))$.

Esempio :

$$f(n) = 3n + 3 \text{ è in } \theta(n).$$

Calcolo notazione asintotica tramite limite

Per due $f(n)$ e $g(n)$, allora :

se $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = k > 0$ (il limite esiste ed è finito), allora $f(n) = \Theta(g(n))$.

se $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = k > 0$ (il limite tende a più infinito), allora $f(n) = \Omega(g(n))$.

se $\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = k > 0$ (il limite tende a zero), allora $f(n) = O(g(n))$.

Se in alcuni casi particolari il limite non dovesse esistere, bisognerebbe procedere diversamente.

Algebra della relazione asintotica

Abbiamo 3 gruppi di **regole** :

Regola 1 - costanti moltiplicative

abbiamo una costante k , se $f(n)$ è in $O(g(n))$, anche $k \cdot f(n)$ sarà in $O(g(n))$, quindi le costanti si possono ignorare (a meno che non vengano moltiplicate agli esponenti).

Regola 2

Per ogni $f(n), d(n) > 0$, se $f(n) = O(g(n))$ e $d(n) = O(h(n))$

Allora $f(n) + g(n) = O(\max(g(n), h(n)))$


- con $\max(g(n), h(n))$ si intende la funzione che cresce più rapidamente tra $g(n)$ e $h(n)$.


Regola 3


Per ogni $f(n), d(n) > 0$, se $f(n) = O(g(n))$ e $d(n) = O(h(n))$


Allora $f(n) \cdot g(n) = O(g(n) \cdot h(n))$


Alcune sommatorie notevoli



$$\sum_{i=0}^n i = \frac{n(n+1)}{2} \text{ è in } \Theta(n^2)$$



$$\sum_{i=0}^n 2^i \text{ è in } \Theta(2^n)$$


$$\sum_{i=0}^n i \cdot 2^i \text{ è in } \Theta(n \cdot c^n) \text{ per ogni } c > 1$$


$$\sum_{i=0}^n \log(i) \text{ è in } \Theta(\log(n))$$


$$\sum_{i=0}^n \frac{1}{i} = \Theta(\log(n))$$


$$\sum_{i=0}^n \log^c(n) \text{ se } c > 1 \text{ è in } \Theta(n \cdot \log(n))$$


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

Valutazione del costo computazionale

Valuteremo il costo computazionale di un algoritmo adottando il **criterio della misura di costo uniforme**. Il costo viene rappresentato come una funzione monotona non decrescente (per il semplice fatto che all'aumentare dei dati è illogico che l'algoritmo impieghi meno tempo), avendo come input i dati da computare. In base a tali input, trovare il parametro della funzione è abbastanza semplice :

- In un algoritmo di ordinamento, il parametro sarà il numero stesso di dati da ordinare
- In un algoritmo che lavora su una matrice, esso sarà il numero di righe per il numero di colonne

Dato che si utilizza la notazione asintotica, il costo computazionale dovrà essere ritenuto **valido solo asintoticamente**, ossia ritenere valido il suo comportamento solo quando l'input è molto grande tendendo all'infinito.

Costo delle istruzioni

Istruzioni elementari

Il costo di un algoritmo è dato dalla somma dei costi delle **istruzioni elementari** in esso presenti, con istruzioni elementari si intendono :

1. Operazioni aritmetiche
2. Lettura e scrittura di una variabile
3. Valutazione di una condizione logica
4. Stampa di un valore a video

Ed hanno un costo costante, equivalente a $\theta(1)$.

```
numero = 10
numero += 10*10
print("il numero è ", numero)
```

$\Theta(1)$

$\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

$\Theta(1)$

Blocchi if/else

Quando bisogna calcolare il costo computazionale di un blocco if/else, il programma potrebbe avere costi diversi a seconda di quale condizione si avvera, quindi se a prescindere viene contato sempre il costo di verifica della condizione, ad esso va sommato il costo del blocco tra if ed else con il costo computazionale maggiore.

$$\text{Costo totale} = \text{Costo verifica} + \max(\text{costo if}, \text{costo else})$$

<pre>if (a>b):</pre>	$\Theta(1)$
<pre> a += b</pre>	$\Theta(1)$
<pre> print(a)</pre>	$\Theta(1)$
<pre>else:</pre>	
<pre> print(b)</pre>	$\Theta(1)$

$$\text{Costo totale} = \Theta(1) + \max(\Theta(1) + \Theta(1), \Theta(1)) = \Theta(1)$$

Blocchi iterativi

Le istruzioni iterative, come i cicli, avranno come costo la somma dei costi di ciascuna delle iterazioni, se tutte le iterazioni sono uguali e hanno lo stesso costo, allora il costo totale sarà il prodotto tra il costo della singola iterazione ed il numero totale di iterazioni, più la verifica dell'ultima iterazione che sarà quella che farà capire all'algoritmo di non rieseguire il ciclo.

$$\text{Costo totale} = \text{NumIterazioni} \cdot \text{Costo iterazione} + \text{Ultima iterazione}$$

<pre>sum = 0</pre>	$\Theta(1)$
<pre>for i in range(5):</pre>	$\Theta(1)$
<pre>sum+=1</pre>	$\Theta(1)$

$$\text{Costo totale} = \Theta(1) + \Theta(1) \cdot 5 + \Theta(1) = \Theta(5)$$

Casistiche

È **importante ricordare** che un algoritmo potrebbe avere tempi di esecuzione molto diversi a seconda dell'input, quindi quando si fa una ricerca sul costo computazionale bisogna valutare come si comporterebbe l'algoritmo nel migliore e nel peggiore dei casi, valutando poi come costo “ufficiale” quest'ultimo. Si calcolano poi entrambi in notazione asintotica per capire quanto differiscono tra loro.

Ricerca binaria

Nell'informatica ci sono problemi particolarmente rilevanti data la frequenza con la quale vengono incontrati, uno di questi problemi, sicuramente uno tra i più comuni, è la ricerca di un elemento in un insieme di dati.

Input : un array A ed un valore V da cercare all'interno dell'array.

Output : L'indice i tale che $A[i] = V$, se l'elemento non è presente nell'array verrà dato un valore nullo (oppure -1) come output. Un esempio di ricerca è la **ricerca sequenziale**, consiste nel selezionare sequenzialmente ogni elemento di dell'array confrontandolo con l'elemento ricercato, non appena viene trovato il valore richiesto, viene interrotta la ricerca e restituito il risultato.



```
def Ricerca (A;v):  
    i = 0  
    while((i<len(A))and(A[i]≠v))  
        i += 1  
    if (i < len(A)):  
        return i  
    else:  
        return -1
```

La Ricorsione

Introduzione

L'algoritmo di ricerca binaria può essere interpretato come un algoritmo **ricorsivo**.



Una **funzione matematica** è detta ricorsiva quando è **definita in termini di se stessa**.

Un classico esempio è la funzione fattoriale :

$$n! = n \cdot (n - 1)! \text{ con } 0! = 1 .$$

La soluzione è costruita risolvendo uno o più sotto-problemi di dimensione minore al problema principale, ricombinando poi la soluzione, essendo i sotto-problemi sempre più piccoli, la funzione convergerà ad un sotto-problema definito **caso base**, dove la ricorsione termina, ad esempio, nella funzione fattoriale $0! = 1$.

```
def Fattoriale(n):  
    if (n == 0):  
        return 1  
    return n*Fattoriale(n-1)
```

Se un'equazione ricorsiva è sprovvista di caso base verrà eseguita in eterno, bisogna accertarsi che qualsiasi computazione converga al caso base.

È importante sapere che **qualsiasi problema risolvibile in modo ricorsivo**, può essere risolto anche in modo **iterativo**. Una soluzione iterativa può rendere la soluzione del problema meno chiara ed intuibile, ma più efficiente in termini di esecuzione del codice, dato che le funzioni ricorsive hanno maggiori esigenze in termini di memoria.

Il calcolo del **Numero di Fibonacci** è un chiaro esempio di funzione ricorsiva, definita come $F(n) = F(n - 1) + F(n - 2)$ con caso base $F(1), F(0) = 1$. Vediamo lo pseudo codice e proviamo a calcolarne il costo computazionale :

```
def Fibonacci(n):  
    if (n<=1):  
        return n  
    return Fibonacci(n-1) + Fibonacci(n-2)
```

0(1)
0(1)
T(n-1)+T(n-2)

Non è esplicito trovare subito il costo di una funzione ricorsiva, vediamo che la funzione $T(n)$ ha costo $T(n) = \theta(1) + T(n - 1) + T(n - 2)$, tali equazioni, volte a calcolare il costo computazionale di una funzione ricorsiva, sono definite come **equazioni di ricorrenza**.

Equazioni di ricorrenza

Valutare il costo di una funzione ricorsiva è più laborioso rispetto che nel caso di una iterativa, trovare la funzione ricorsiva del costo computazionale è immediato, essa deve essere però risolta per essere rappresentata sotto-forma di notazione asintotica. Un equazione di ricorrenza definita come $T(n)$ deve essere costituita da almeno 2 addendi, uno contenente **la parte ricorsiva** ed uno rappresentante **il costo computazionale** di ciò che avviene nella funzione, ad esempio :

$$T(n) = \theta(1) + T(n - 1) \quad \text{con} \quad T(1) = \Theta(1)$$

Ovviamente deve essere sempre presente almeno un caso base. Esistono **diversi metodi** per risolvere un equazione di ricorrenza.

Metodo iterativo

L'idea è quella di sviluppare l'equazione ed **esprimerla come somma di termini dipendenti dall'input e dal caso base**, questo porterà inevitabilmente una maggiore quantità di calcoli algebrici rispetto altri metodi, vediamo un **esempio di applicazione** :

Si consideri la funzione $T(n) = \Theta(1) + T(n - 1)$ con $T(1) = \Theta(1)$.

Risulta chiaro che :

$$T(n - 1) = \Theta(1) + T(n - 2)$$

quindi per sostituzione :

$$T(n) = \Theta(1) + \Theta(1) + T(n - 2)$$

Si noti ancora che:

$$T(n - 2) = \Theta(1) + T(n - 3)$$

quindi :

$$T(n) = \Theta(1) + \Theta(1) + \Theta(1) + T(n - 3)$$

A questo punto risulta chiaro che :

$$T(n) = n \cdot \Theta(1) = \Theta(n)$$

Partendo da un'equazione di ricorrenza si è arrivati ad un costo computazionale in notazione asintotica, vediamo **un altro esempio** :

Si consideri la funzione $T(n) = T(\frac{n}{2}) + \theta(1)$

Si noti che :

$$T(\frac{n}{2}) = T(\frac{n}{2}/2) + \theta(1) = T(\frac{n}{2^2}) + \theta(1)$$

Per sostituzione :

$$T(n) = T(\frac{n}{2^2}) + \theta(1) + \theta(1)$$

Risulta poi chiaro che :

$$T(\frac{n}{2^2}) = T(\frac{n}{2^2}/2) + \theta(1) = T(\frac{n}{2^3}) + \theta(1)$$

Quindi :

$$T(n) = T(\frac{n}{2^3}) + \theta(1) + \theta(1) + \theta(1)$$

è quindi chiaro che

$$T(n) = T(\frac{n}{2^k}) + k \cdot \theta(1)$$

Volendo esser ricondotti al caso base $T(1) = \theta(1)$, è importante notare che quando

$k = \log_2(n)$ si ha $\frac{n}{2^k} = 1$, quindi per tale valore di k ci fermiamo ed otteniamo :

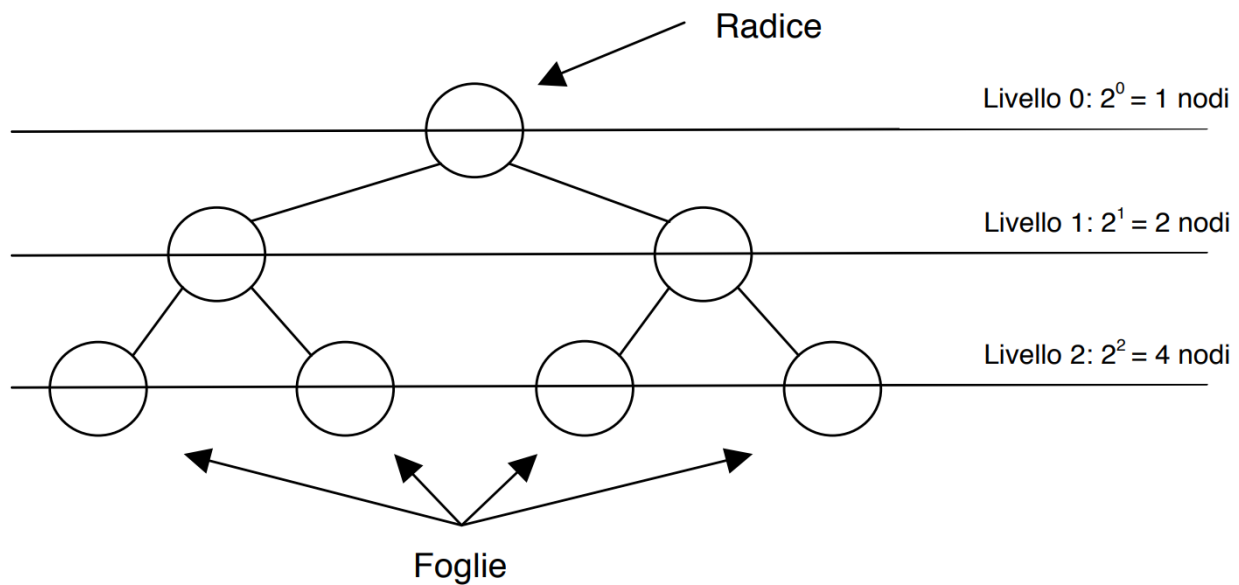
$$T(n) = \theta(1) + \log_2(n) \cdot \theta(1) = \theta(\log_2(n))$$

Metodo dell'albero

Il metodo dell'albero deriva dal metodo iterativo e consiste nel **rappresentare graficamente** lo sviluppo del costo dell'algoritmo per valutarlo con maggiore facilità. Utilizzeremo gli alberi binari, per i quali è importante fare alcune precisazioni :

Parentesi sugli alberi binari

Un albero completo di altezza h :



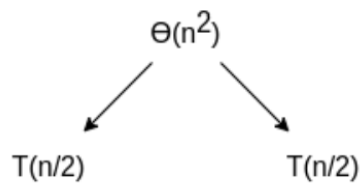
- ha numero di foglie 2^h
- Il numero dei nodi interni è $\sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 2^h - 1$
- Il numero totale dei nodi è $n = 2^{h+1} - 1$

Di conseguenza, l'altezza di un albero vale $\log_2\left(\frac{n+1}{2}\right)$.

Vediamo ora lo sviluppo di un'equazione di ricorrenza, si consideri la seguente equazione :

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n^2) \quad \text{con} \quad T(1) = \theta(1)$$

Ad ogni iterazione, si processa per due volte la metà dell'input iniziale, rappresenteremo graficamente la prima iterazione come segue :

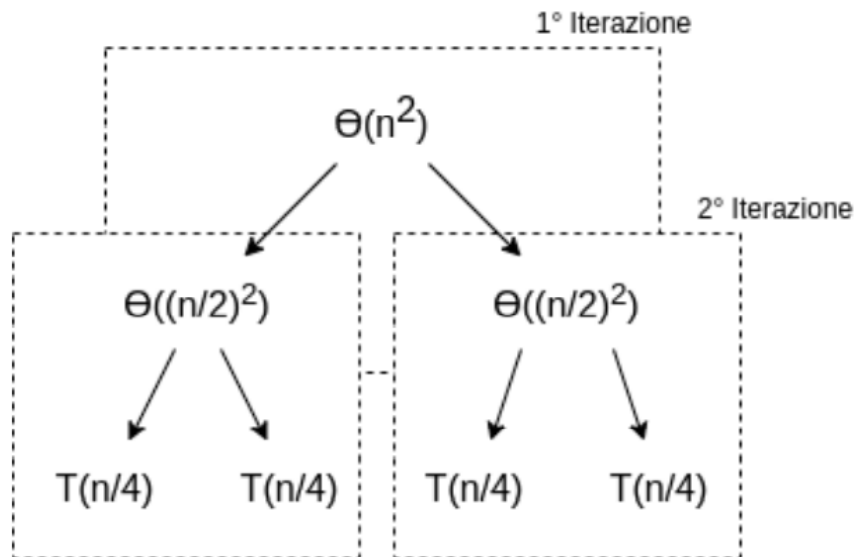


...

Vediamo che $T(\frac{n}{2}) = 2T(\frac{n}{4}) + \theta((\frac{n}{2})^2)$, rappresenteremo quindi la seconda iterazione come

$$T(n) = 2(2T(\frac{n}{4}) + \theta((\frac{n}{2})^2)) + \theta(n^2) = 4T(\frac{n}{4}) + 2\theta((\frac{n}{2})^2) + \theta(n^2),$$

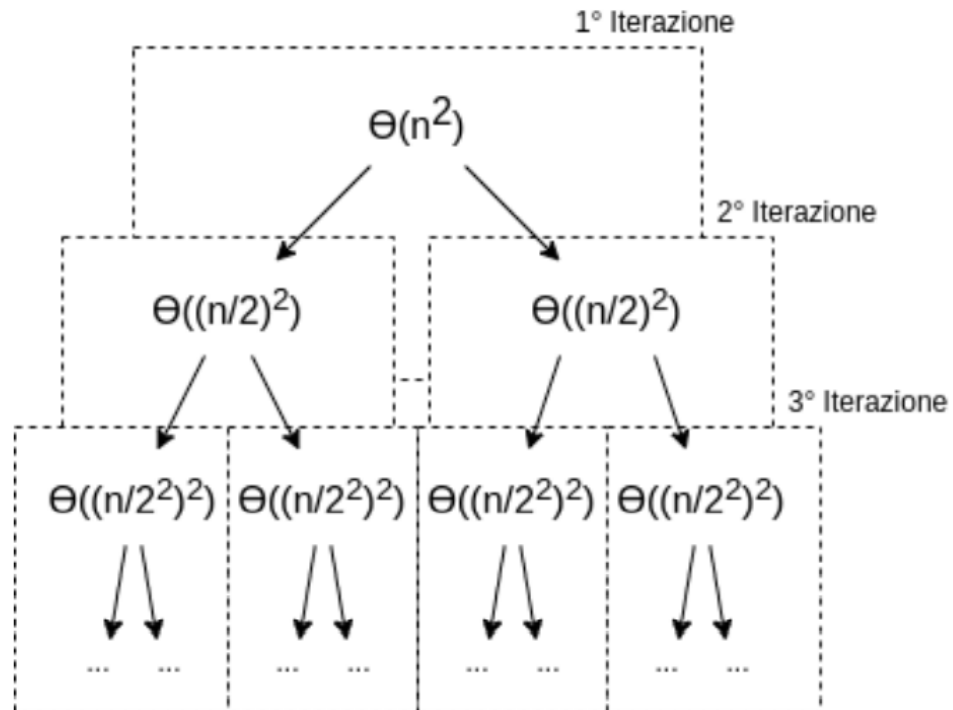
Ne consegue la seguente rappresentazione grafica :



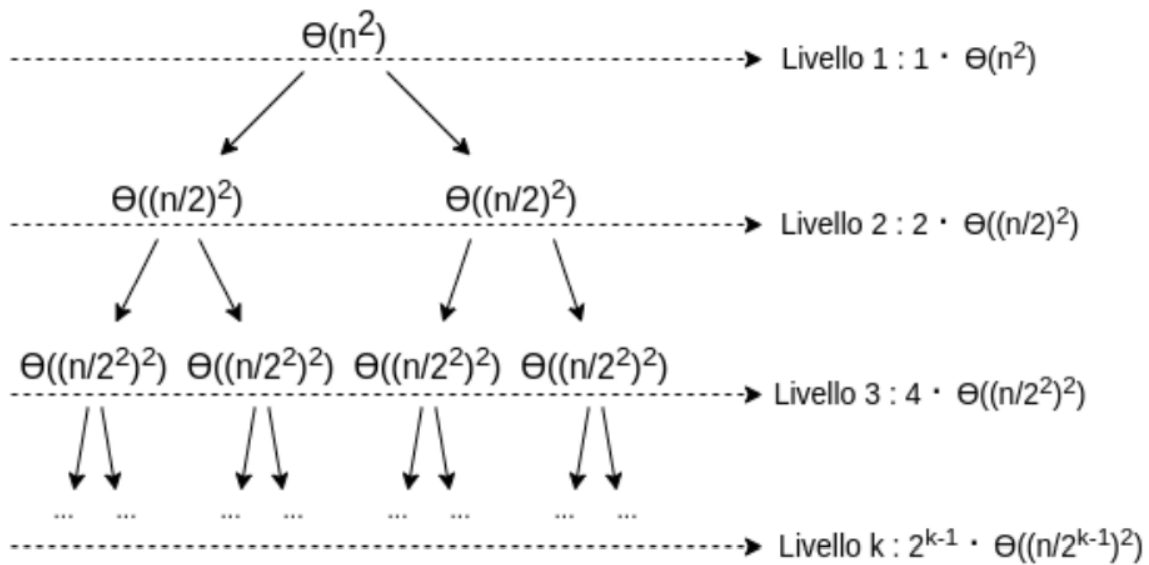
Procedendo con lo svolgimento, sarà chiara la terza iterazione come segue :

$$\begin{aligned}
 T(n) &= 2(2(2T(\frac{n}{8}) + \theta(n^2)) + \theta(n^2)) + \theta(n^2) = \\
 &= 8T(\frac{n}{8}) + 4\theta((\frac{n}{4})^2) + 2\theta((\frac{n}{2})^2) + \theta(n^2)
 \end{aligned}$$

Ne consegue il seguente grafico :



Analizziamo adesso cosa accade ad **ogni livello dell'albero** partendo dalla prima alla k-esima iterazione :



Essendo il costo la **somma di tutti i livelli** vediamo che :

$$T(n) = \sum_{i=0}^{k-1} 2^i \cdot \theta\left(\left(\frac{n}{2^i}\right)^2\right) = \theta\left(\sum_{i=0}^{k-1} \frac{n^2}{2^i}\right)$$

Tali iterazioni vengono eseguite affinché $\frac{n}{2^k} = 1$, quindi per $k = \log_2(n)$

$$T(n) = \theta\left(\sum_{i=0}^{\log_2(n)-1} \frac{n^2}{2^i}\right) = \theta(n^2) \cdot \left(2 - \frac{1}{2^{\log_2(n)+1}}\right) = \theta(n^2)$$

Metodo di sostituzione

Il metodo di sostituzione è particolare, si **ipotizza intuitivamente** una soluzione per l'equazione di ricorrenza data, verificando poi per **induzione** se essa funziona. Tale metodo risulta tedioso in quanto è difficile trovare una funzione vicina alla vera soluzione, de facto, non è molto utile nella pratica se non quanto nelle dimostrazioni. Vediamo un esempio, consideriamo la seguente equazione relativa alla ricerca sequenziale ricorsiva :

$$T(n) = T(n-1) + \theta(1) \quad \text{con} \quad T(1) = \theta(1)$$

Ipotizziamo che $T(n) = O(n)$, ossia che $T(n) \leq k \cdot n$ per una certa costante k . Inoltre è importante **eliminare la notazione asintotica** dall'equazione, sostituendo $\theta(1)$ con due costanti c e d fissate.

$$T(n) = T(n-1) + c \quad \text{con} \quad T(1) = d$$

Analizzando il **caso base** : $n = 1 \implies T(1) \leq k \cdot 1$, essendo che $T(1) = d$ otteniamo che la disuguaglianza è vera se e solo se $d \leq k$.

Analizzando poi il **passo induttivo**, per un n generico si ha $T(n) \leq k \cdot n$, sapendo che

$T(n) = T(n-1) + c$ otteniamo la disuguaglianza $T(n-1) + c \leq k \cdot n$, inoltre per ipotesi induttiva, essendo $T(n) = O(n)$, si ha che $T(n-1) \leq k(n-1)$, dunque riscriviamo l'equazione come :

$$k(n-1) + c \leq k \cdot n \implies kn - k + c \leq kn \implies -k + c \leq 0 \implies c \leq k$$

Poiché esiste sempre un valore k tale che $k \geq c$ e $k \geq d$, la soluzione ipotizzata $T(n) \leq kn$ è corretta dunque $T(n) = O(n)$.

Adesso, per avere un risultato più **stretto**, ipotizziamo la soluzione $T(n) = \Omega(n)$, ossia che

$T(n) \geq h \cdot n$ dove h è una costante ancora da determinare. Analogamente al primo caso, avendo definito le costanti c e d , con $T(1) = d$, sostituiamo nel caso ottenendo $h \leq d$, per ipotesi induttiva avremo anche $T(n) \geq h(n-1) + c \implies h \cdot n - h + c \geq h \cdot n$, vero se e solo se $h \leq c$, essendo possibile trovare un valore h minore sia di c che di d per il quale $T(n) \geq h \cdot n$ deduciamo che $T(n) = \Omega(n)$.

Avendo dimostrato che $T(n) = \Omega(n)$ e $T(n) = O(n)$ possiamo dire con assoluta certezza che $T(n) = \Theta(n)$

Metodo principale

Il metodo principale ci fornisce un teorema, il quale analizzando l'equazione in un certo modo, è in grado di dirci il suo preciso costo computazionale, prendiamo una qualsiasi equazione di ricorrenza :

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Dove a è una costante, che moltiplica $T\left(\frac{n}{b}\right)$ che rappresenta una **frazione dell'equazione totale**, e $f(n)$ che rappresenta la parte computazionale θ . Inoltre, è importante che ci sia sempre il caso base $T(1) = \theta(1)$.

Quindi una volta che si ha un'equazione di ricorrenza uguale ad una frazione di se stessa ripetuta a volte alla quale viene sommata la parte computazionale, si ha il seguente enunciato :

Enunciato



Dati $a \geq 1$, $b > 1$, una funzione asintoticamente positiva ed un'equazione di ricorrenza : $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, vale che :

- **CASO 1** - Se $f(n) = O(n^{\log_b a - \epsilon})$ per un qualsiasi $\epsilon > 0$, allora $T(n) = \theta(n^{\log_b a})$

- **CASO 2** - Se $f(n) = \theta(n^{\log_b a})$, allora $T(n) = \theta(n^{\log_b a} \cdot \log n)$
- **CASO 3** - Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per un qualsiasi $\epsilon > 0$, e $f(\frac{n}{b}) \leq c \cdot f(n)$ per un qualsiasi $c < 1$ allora $T(n) = \theta(f(n))$

Vediamo adesso alcuni esempi di applicazione del teorema :

Esempio di applicazione del primo caso

Abbiamo la seguente equazione di ricorrenza :

$$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + \theta(n)$$

Si noti che : $a = 9$, $b = 3$ e $f(n) = \theta(n)$

È utile ricavarsi subito $n^{\log_b a} = n^{\log_3 9} = n^2$

Secondo il primo caso, vediamo se esiste un $\epsilon > 0$ tale che $f(n) = O(n^{\log_b a - \epsilon})$.

domanda : $\exists \epsilon > 0$ tale che $\theta(n) = O(n^{2-\epsilon})$?

risposta : sì, $\forall \epsilon \leq 1$.

Quindi secondo il teorema, $T(n) = \theta(n^{\log_b a})$, cioè $T(n) = \theta(n^2)$

Esempio di applicazione del secondo caso

Abbiamo la seguente equazione di ricorrenza :

$$T(n) = T\left(\frac{2}{3}n\right) + \theta(1)$$

Si noti che : $a = 1$, $b = \frac{3}{2}$ e $f(n) = \theta(1)$

È utile ricavarsi subito $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

Secondo il secondo caso, $f(n) = \theta(n^{\log_b a})$?

domanda : $\theta(1) = \theta(1)$?

risposta : sì

Quindi secondo il teorema, $T(n) = \theta(n^{\log_b a} \cdot \log n)$, cioè $T(n) = \theta(\log n)$

Esempio di applicazione del terzo caso

Abbiamo la seguente equazione di ricorrenza :

$$T(n) = 3T\left(\frac{n}{4}\right) + \theta(n \cdot \log n)$$

Si noti che : $a = 3$, $b = 4$ e $f(n) = \theta(n \cdot \log n)$

$$n^{\log_b a} = n^{\log_4 3} = n^{0.7 \text{circa}}$$

Secondo il terzo caso, vediamo se esiste un ϵ tale che $f(n) = \theta(n^{\log_b a + \epsilon})$, e se esiste un $c < 1$, tale che $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$.

domanda : $\exists \epsilon \text{ t.c. } \theta(n \cdot \log n) = \Theta(n^{0.7 \text{circa} + \epsilon})$, ed $\exists c \text{ t.c. } 3 \cdot f\left(\frac{n}{4}\right) \leq c \cdot n \log n$?

risposta : sì, poniamo $c = \frac{3}{4}$

Quindi secondo il teorema, $T(n) = \Theta(n \log n)$

Esempio di impossibilità di applicazione

Vediamo l'equazione di ricorrenza :

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n \log n)$$

$$a = 2 \quad b = 2 \quad f(n) = \theta(n \log n) \quad n^{\log_b a} = n^{\log_2 2} = n$$

è vero che $n \log n = O(n)$, ma **non polinomialmente**, perchè non esiste nessun $\epsilon > 0$ tale che : $n \log n = O(n^{1-\epsilon})$, quindi non è applicabile il metodo principale.

Algoritmi di sorting

Il problema dell'ordinamento

Una parte rilevante dei tempi di calcolo è dovuto ad algoritmi di ordinamento, è un problema comune e ricorrente e deve essere spesso trattato a fronte di risolvere problemi più complessi.

- è un algoritmo in grado di ordinare gli elementi di un insieme in base ad una relazione d'ordine

Tipicamente, ordinare dati è utile quando tali dati dovranno essere soggetti ad una ricerca, esistono diversi tipi di algoritmi di ordinamento, vedremo inizialmente algoritmi **NAIF**,

estremamente semplici ed il loro costo computazionale è solitamente $\theta(n^2)$, per migliorare l'efficienza vedremo poi algoritmi più complessi.

NAIF :

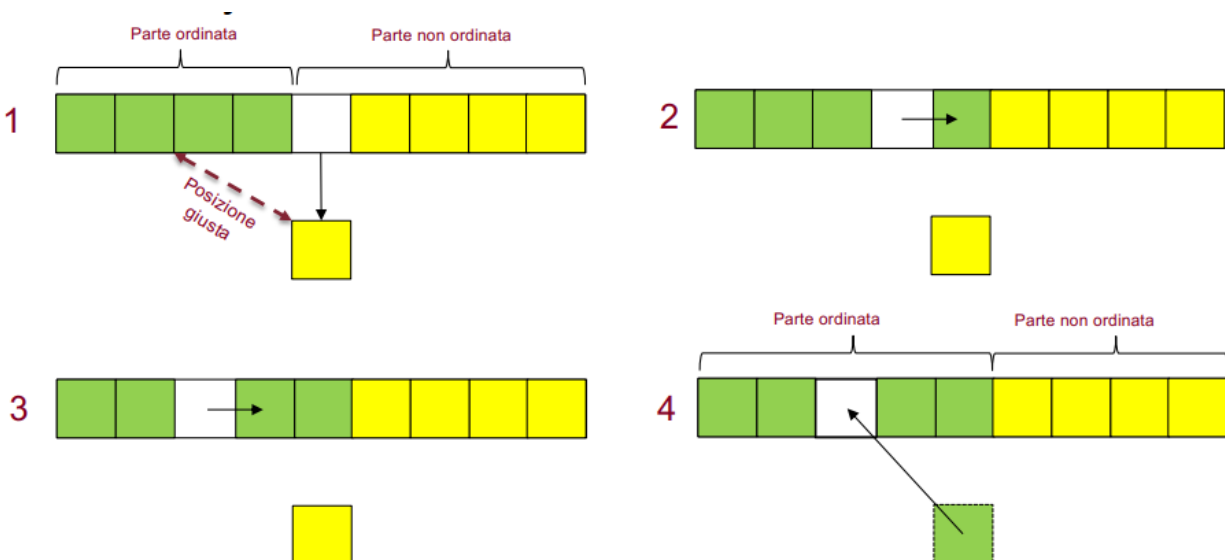
- Insertion sort
- Selection sort
- Bubble sort

COMPLESSI :

- Merge sort
- Heap sort
- Quick sort

Insertion sort

Estraggo il primo elemento, lo sposto verso destra liberando la sua posizione corrente, sposto verso destra tutti gli elementi alla sua sinistra che sono maggiori di esso e si reinserisce l'elemento nella posizione liberata.



Pseudo codice


```

def Insertion_Sort(A)
  for j in range(1, len(A)):
    x = A[j]
    i = j - 1
    while ((i >= 0) and (A[i] > x)) #tj
      A[i+1] = A[i]
      i = i - 1
    A[i+1] = x

```

$$T(n) = \sum_{j=1}^{n-1} (\theta(1) + t_j \cdot \theta(1) + \theta(1)) + \theta(1)$$

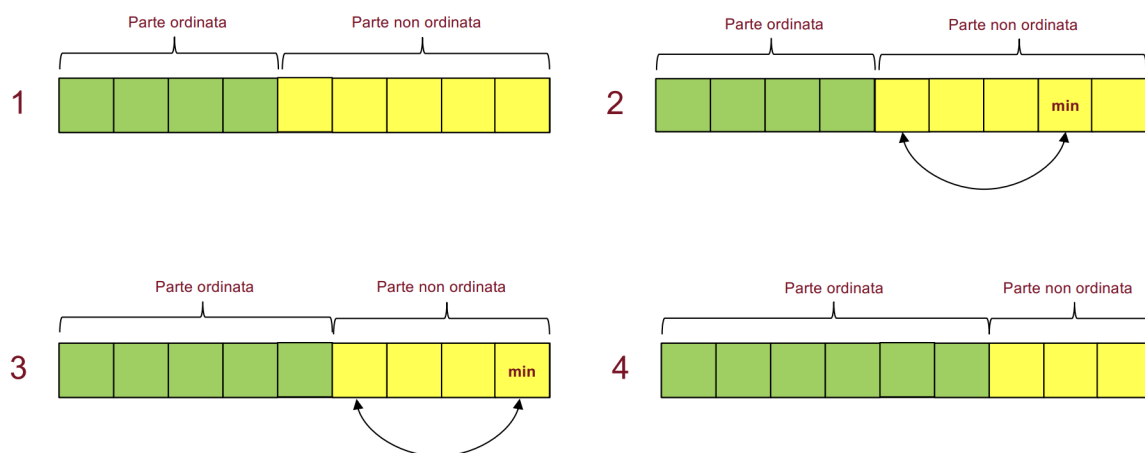
Caso migliore : $T(n) = \theta(n)$ se $t_j = 0$

Caso peggiore : $T(n) = \theta(n^2)$

Il caso migliore si verifica se l'array è già ordinato.

Selection sort

Nel selection sort, si ricerca l'elemento più piccolo nell'insieme iniziale disordinato, e lo pone come primo elemento dell'array, rieseguendo l'operazione ignorando l'elemento a sinistra appena inserito, che rappresenterà la parte ordinata.



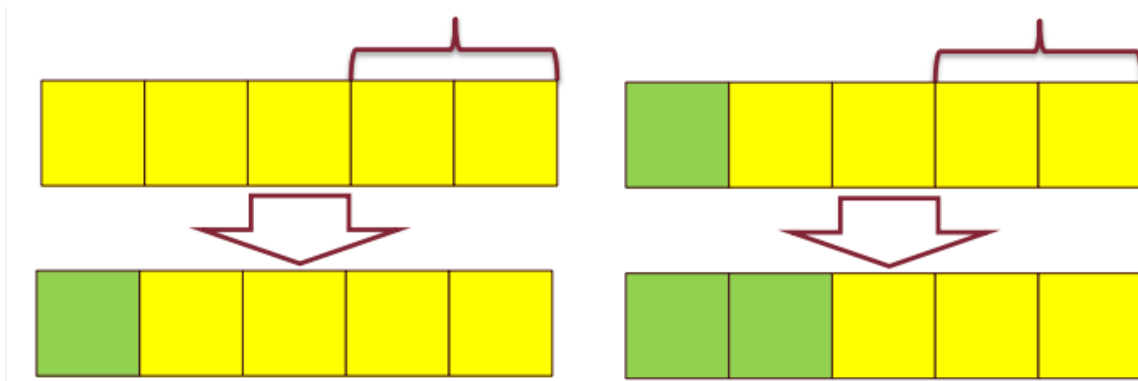
Pseudo codice

```
def Selection_Sort(A)
    for i in range(len(A)-1):
        m = i
        for j in range(i+1, len(A)):
            if (A[j] < A[m]):
                m = j
        A[m], A[i] = A[i], A[m]
```

In tutti i casi il suo costo computazionale è $\theta(n^2)$.

Bubble sort

Nel bubble sort si confronta ogni elemento con tutti gli altri dell'array, ogni volta che due elementi non sono ordinati vengono scambiati, quindi controlla tutti gli elementi adiacenti due a due, da destra verso sinistra, scambiando quelli non ordinati.



Pseudo codice

```
def Bubble_Sort(A)
    for i in range(len(A)):
        for j in range(len(A)-1, i, -1):
            if (A[j] < A[j - 1]):
                A[j], A[j - 1] = A[j-1], A[j]
```

Il costo del primo *for* è $n \cdot \theta(1) + \theta(1)$, del secondo è $(n - i) \cdot \theta(1) + \theta(1)$,

Il costo è quindi $\sum_{i=0}^{n-1} (\theta(1) + (n-i)\theta(1) + \theta(1)) + \theta(1) = \theta(n^2)$

Anche in questo algoritmo, non c'è differenza di costi tra caso peggiore e migliore.

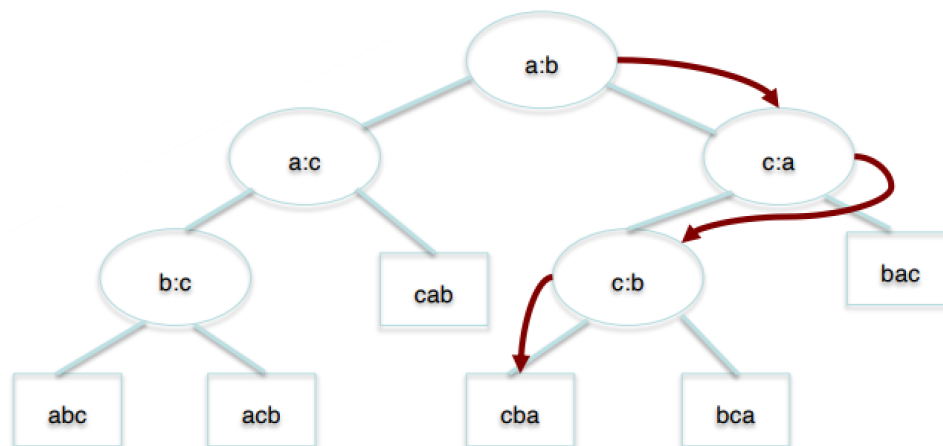
La complessità dell'ordinamento

In ambito di ordinamenti ci si pone la domanda : Qual è il **numero minimo di operazioni** che un algoritmo deve computare? Una volta trovata una risposta ci si chiede se tale numero può essere abbassato ancora di più, e di quanto, quindi si trova un numero minimo di operazioni necessarie.

Qual è il numero minimo di operazioni da fare in un algoritmo di ordinamento basato su confronti a coppie?

Per trovare una risposta si utilizza uno strumento denominato **albero delle decisioni**, esso rappresenta graficamente le strade/alberazioni che un processo computazionale può intraprendere, è un *albero binario*, vediamo un esempio su un ordinamento di un array composto da 3 elementi. $A = [a, b, c]$

- $a : b$ = confrontando a con b



La lunghezza degli archi rappresenta il numero di confronti necessari per trovare la soluzione, la lunghezza del **percorso più lungo** dalla radice ad una foglia (altezza dell'albero) rappresenta il numero di confronti necessari nel caso peggiore. Il numero delle foglie rappresenterà tutte le possibili soluzioni dell'algoritmo, ed esso sarà sempre $\geq n!$, dove n è il numero di dati in input.

Inoltre, un albero binario di altezza h , non può avere più di 2^h soluzioni/foglie. Segue che l'altezza h di un algoritmo basato sui confronti deve essere tale per cui $2^h \geq n!$,

ossia $h \geq \log(n!)$, adesso valutiamo $\log(n!)$ assumendo che n sia pari :

$$\log(n!) = \log(n \cdot (n-1) \cdot (n-2) \cdot (n-3) \dots \cdot 2 \cdot 1) = \sum_{i=1}^n \log(i) +$$

$$\sum_{i=\frac{n}{2}+1}^n \log(i)$$

$$\geq \sum_{i=\frac{n}{2}+1}^n \log\left(\frac{n}{2}\right) = \frac{n}{2} \log(n) - \frac{n}{2} = \theta(n \cdot \log(n))$$

Abbiamo dimostrato che $n! = \theta(n \cdot \log(n))$, quindi $h = \Omega(n \cdot \log(n))$, da qui ne consegue il seguente **Teorema** :

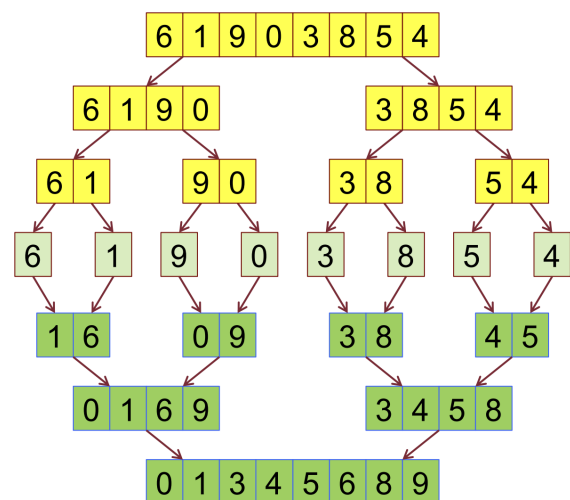


Il costo computazionale di qualunque algoritmo basato sui confronti è $\Omega(n \cdot \log(n))$.

Merge Sort

Il Merge Sort adotta una tecnica definita come “**Divide et impera**”, essa consiste nel **suddividere il problema complessivo in più sotto-problemi** che si risolvono **ricorsivamente**, le soluzioni dei sotto-problemi poi si **ricompongono** per definire la soluzione finale.

- **Divide** - La sequenza di n elementi viene suddivisa in 2 sequenze da $\frac{n}{2}$ elementi.
- **Impera** - Le due sotto-sequenze vengono ordinate ricorsivamente. Il caso base si verifica quando la sequenza è costituita da 1 solo elemento.
- **Combina** - Le due sotto-sequenze da $\frac{n}{2}$ elementi vengono combinate insieme in un'unica sequenza di n elementi.



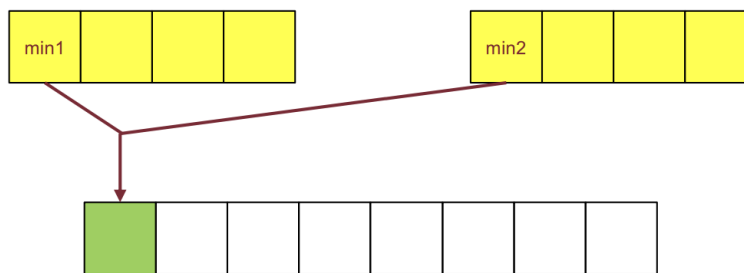
```

def Merge_sort (A, ind_primo, ind_ultimo):           #T(n)
    if (ind_primo < ind_ultimo)                       #Θ(1)
        ind_medio = (ind_primo+ind_ultimo)//2        #Θ(1)
        Merge_sort (A, ind_primo, ind_medio)         #T(n/2)
        Merge_sort (A, ind_medio + 1, ind_ultimo)    #T(n/2)
        Fondi (A, ind_primo, ind_medio, ind_ultimo)  #S(n)

```

Si ha la seguente equazione di ricorrenza : $T(n) = 2T(\frac{n}{2}) + S(n) + \Theta(1)$

La funzione `Fondi()` sfrutta il fatto che le due sotto-sequenze siano ordinate, il minimo della sequenza complessiva sarà il minimo tra i due minimi delle sotto-sequenze, dopo aver eliminato un elemento da una delle due sotto-sequenze, l'elemento successivo della prossima sequenza non può che essere ancora il minimo tra le due sotto-sequenze.



```

def Fondi (A, ind_primo, ind_medio, ind_ultimo):
    i, j = indice_primo, indice_medio+1
    B=[] #è necessario un Array ausiliario B
    while ((i≤ind_medio) and (j≤ind_ultimo))
        if (A[i]≤A[j]):
            B.append(A[i])
            i += 1
        else:
            B.append(A[j])
            j += 1
    while (i≤ind_medio) #il primo sottoarray non è terminato
        B.append(A[i])
        i += 1
    while (j≤ind_ultimo) #il secondo sottoarray non è terminato

```

```

B.append(A[j])
j += 1
for i in range(len(B)):
    A[primo+i] = B[i]

```

Il costo della funzione `fondi` è chiaramente $\Theta(n)$.

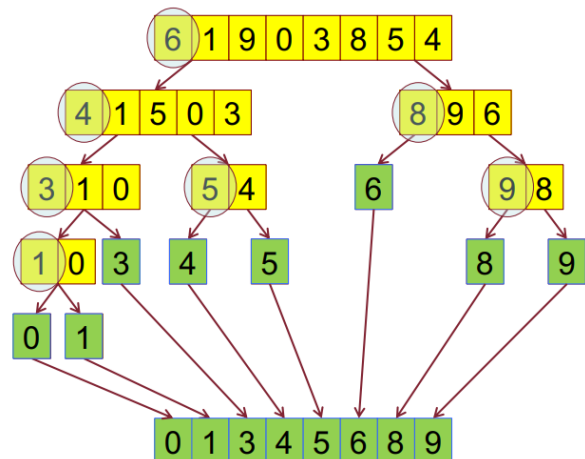
Detto ciò, l'equazione di ricorrenza del merge sort risulta $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Risolvendo l'equazione con il teorema principale, otteniamo il costo $\Theta(n \cdot \log(n))$, che si ricordi essere il minimo possibile per gli algoritmi basati sui confronti.

Quick Sort

Il quicksort, nel suo caso peggiore ha costo $O(n^2)$, ma nella **pratica**, per grandi valori di n riesce a **mantenere un costo medio** di $\Theta(n \cdot \log(n))$, e soprattutto, diversamente dal Merge Sort, permette l'ordinamento “*in loco*”, ossia senza l'utilizzo di un array di appoggio, è un algoritmo ricorsivo che adotta il paradigma *Divide et Impera*.

- **Divide** - Nella sequenza di n elementi si seleziona un Pivot, la sequenza viene quindi divisa in 2 sottosequenze, quella degli elementi minori o uguali al Pivot, e quella degli elementi maggiori o uguali al Pivot.
- **Impera** - Le 2 sotto-sequenze vengono ordinate ricorsivamente. Il caso base si verifica quando le sotto-sequenze sono costituite da un solo elemento



```

def Quick_sort (A, ind_primo, ind_ultimo):
    if (ind_primo < ind_ultimo):
        ind_medio=Partiziona(A,ind_primo,ind_ultimo)
        Quick _sort (A,ind_primo,ind_medio)
        Quick _sort (A,ind_medio+1,ind_ultimo)

```

In questa implementazione `ind_medio` è l'indice dell'estremo superiore della porzione di sinistra (quella contenente elementi minori o uguali del pivot). Il suo valore è sempre compreso fra 1 ed $n - 1$. Vediamo la funzione *Partiziona* :

```
def Partiziona(A, ind_primo, ind_ultimo):
    pivot = A[ind_primo]                #Θ(1)
    i = ind_primo - 1                  #Θ(1)
    j = ind_primo + 1                  #Θ(1)
    while true:                         #Θ(n)
        while(A[j]<=pivot):
            j=j-1
        while(A[i]>=pivot):
            i=i+1
        if(i<j):
            A[i],A[j] = A[j],A[i]
        else:
            return j
```

Il costo del primo `while` è $\Theta(n)$, in quanto ciascuna iterazione di ognuno dei due `while` interni costa $\Theta(1)$ e avvicina di una posizione un indice all'altro. Dopo i due `while` si trova una condizione `if` che costa $\Theta(1)$, quindi il costo complessivo di *Partiziona* è $\Theta(n)$.

L'equazione di ricorrenza del Quick Sort sarà quindi $T(n) = T(k) + T(n - k) + \Theta(n)$, che non sappiamo risolvere con nessuno dei 4 metodi visti in precedenza. Possiamo però *derivare* la soluzione secondo due casi :

- **Caso migliore** - È quello in cui ad ogni passo ricorsivo, la dimensione dei due sotto-problemi è identica, l'equazione diventa $T(n) = 2T(\frac{n}{2}) + \Theta(n) \implies T(n) = \Theta(n \cdot \log(n))$
- **Caso peggiore** - È quello in cui ad ogni passo ricorsivo, la dimensione di una delle due sotto-sequenze è 1, l'equazione diventa: $T(n) = T(n - 1) + \Theta(n) \implies T(n) = \Theta(n^2)$

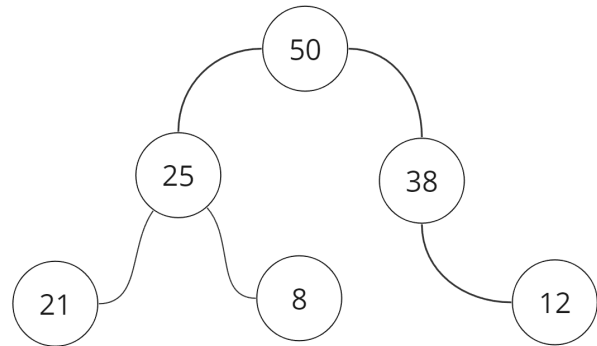
Il miglior caso si verifica quanto il Pivot è **equiprobabile**, conviene *randomizzare* la sequenza da ordinare, per disgregare l'eventuale regolarità interna e rendere l'algoritmo indipendente dall'input. Il Quick Sort è considerato ideale per algoritmi di grandi dimensioni.

Heap Sort

L'Heap Sort è un **algoritmo piuttosto complesso** che esibisce ottime caratteristiche, ha un costo computazionale, anche nel caso peggiore di $O(n \cdot \log(n))$ e permette l'ordinamento “*in loco*”, sfrutta una struttura dati che garantisce specifiche proprietà chiamata **Heap**.

La struttura dati Heap

L'Heap non è altro che un **albero binario** completo o quasi completo, con la proprietà che ogni nodo, ha chiave maggiore o uguale a quella dei propri figli, per implementare un Heap si utilizza la notazione posizionale, utilizzando un array, con tante posizioni quanti sono i nodi dell'albero, con la radice nell'indice 0.

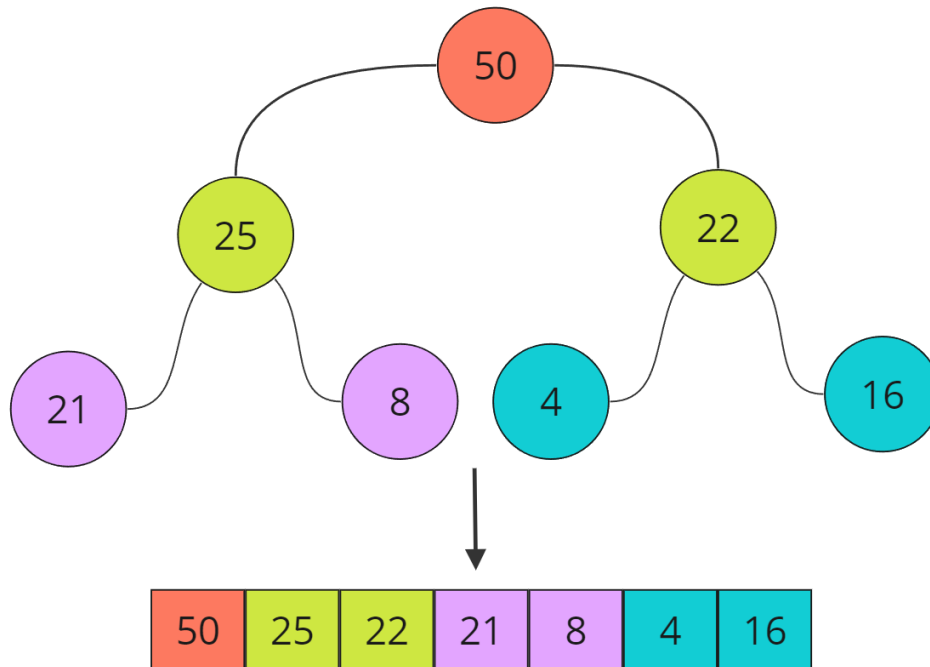


Ogni chiave di indice i , ossia $A[i]$, ha i suoi figli in $A[2i + 1]$ e $A[2i + 2]$, il padre di ogni nodo $A[i]$ si trova nella posizione $A[(i - 1)/2]$.

```
Left(i) = 2i+1  
Right(i) = 2i+2  
Parent(i) =
```

```
 $\lfloor (i-1)/2 \rfloor$  arrotondato per eccesso
```

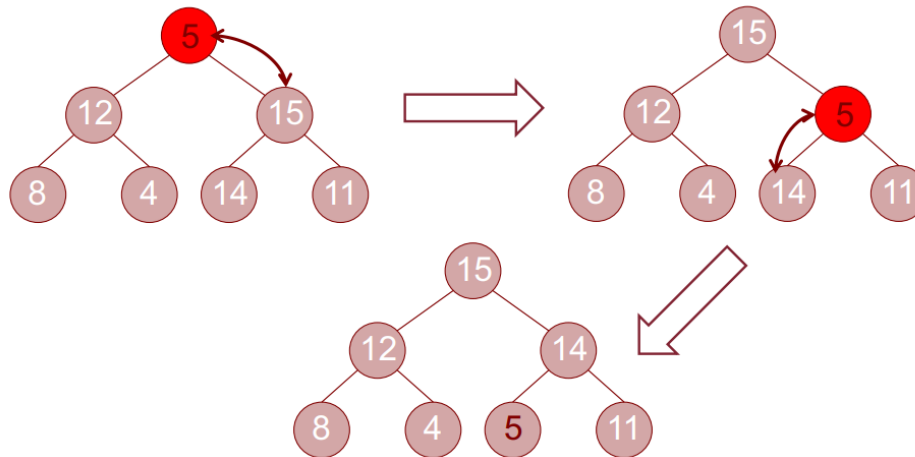
Le operazioni di ricerca, inserimento e cancellazione condividono un costo $O(\log(n))$.



Poiché L'Heap ha tutti i livelli completamente pieni tranne al più l'ultimo, ha altezza $\log(n)$, per la proprietà di ordinamento verticale, vale che $A[i] \leq A[\text{parent}(i)]$, e l'**elemento massimo** risiede nella radice, può essere trovato quindi in $O(1)$. L'algoritmo Heap sort per il suo funzionamento ha bisogno di 2 funzioni ausiliarie.

Funzione Heapify

La funzione Heapify ha lo scopo di “**riaggiustare**” un albero sulla quale sia **garantita la proprietà di ordinamento verticale** per i suoi due sotto-alberi, ma **non per la radice**, che può essere minore di uno o di entrambi i figli. La funzione opera sulla radice, scambiandola se necessario col maggiore dei suoi figli. Dopo lo scambio, si verifica se la proprietà dell'Heap sia garantita anche per il figlio scambiato, se no, si ripete ricorsivamente l'operazione su tale nodo.



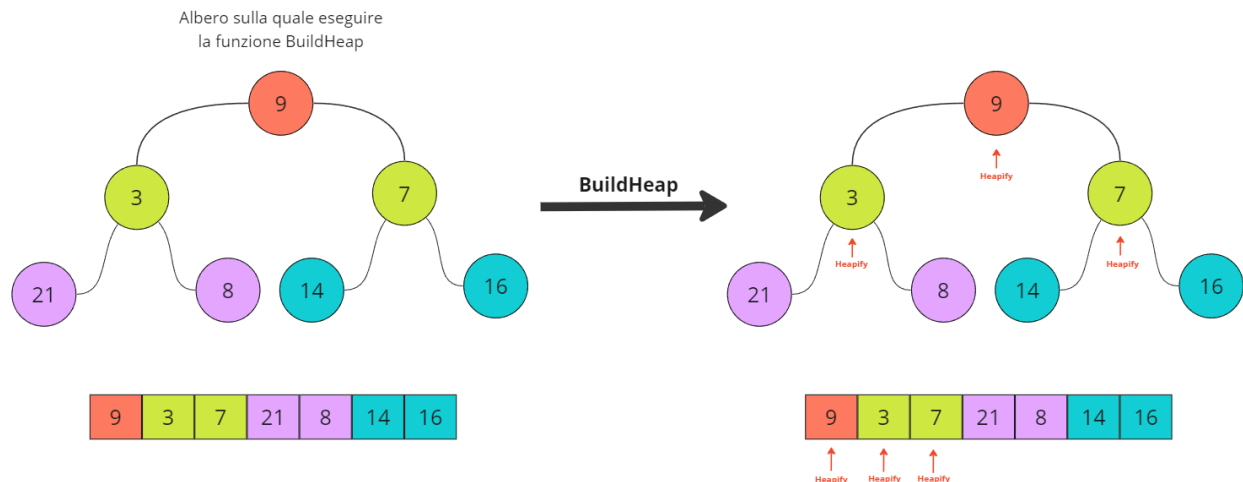
```
def Heapify (A, i, heap_size)
    L=left(i); R=right(i); indice_max=i
    if ((L < heap_size) and (A[L] > A[i])):
        indice_max=L
    if ((R ≤ heap_size) and (A[R] >A[indice_max]))
        indice_max=R
    if (indice_max ≠ i)
        A[i], A[indice_max]=A[indice_max], A[i]
        Heapify (A, indice_max, heap_size)
```

Ha costo $T(n) = \Theta(1) + T(n')$ dove n' è il numero di nodi del sottoalbero che ha più nodi.

Essendo che i sotto-alberi della radice non possono avere più di $2n/3$ nodi, l'equazione di ricorrenza diventa $T(n) = T(2/3n) + \Theta(1) \implies T(n) = O(\log(n))$.

Funzione BuildHeap

La funzione BuildHeap non fa altro che trasformare qualunque array contenente n elementi in un heap, chiamando ripetutamente *Heapify* sugli opportuni nodi. Essendo che Heapify da per scontato che i sotto-alberi del nodo che si controlla siano già degli Heap ordinati, la funzione inizierà a lavora sulle foglie dell'albero non ordinato, basterà chiamare Heapify a partire dal nodo interno più a destra, che ha indice $\lfloor n/2 \rfloor$, scorrendo poi verso sinistra.



```
def Build_heap (A):
    for i in reversed(range(len(A)//2)):
        Heapify (A, i, heap_size)
```

La funzione BuildHeap effettua n chiamate di Heapify, che ricordiamo avere costo $\Theta(\log(n))$, quindi la funzione totale HeapSort avrà costo $T(n) = O(n \cdot \log(n))$

Ordinamenti Lineari

Precedentemente, abbiamo dimostrato che **ogni algoritmo di ordinamento che opera sui confronti ha costo $\Omega(n \cdot \log(n))$** , come si può quindi avere degli algoritmi di ordinamento di **costo lineare**, ossia $\Theta(n)$?

Counting Sort

Il Counting sort per poter essere attuato necessita di **ipotesi aggiuntive**.

- Ciascuno degli elementi da ordinare è necessariamente un **intero** di valore compreso in un intervallo $[0, k]$.
- Il costo computazionale è di $\Theta(n + k)$, se $k = O(n)$, allora l'algoritmo ordina n elementi in un tempo lineare, ossia $\Theta(n)$.

L'idea è quella di fare in modo che ogni elemento della sequenza, **determini direttamente** la sua posizione nella sequenza ordinata.

Utilizzeremo un **array ausiliario** C , che conterrà le occorrenze, scorrendo la sequenza disordinata A , per ogni indice, conteggiamo il numero di occorrenze di ciascun valore (ad esempio, se in A il numero 3 appare 2 volte, l'array C nell'indice 3, avrà valore 2), una volta riempito C , iteriamo per tale array, ricopiando in A ciascun indice di C tante volte quanto è il valore di C in quell'indice.

Si dia il caso che vogliamo ordinare una sequenza di 6 interi, tutti in un intervallo $[0, 4]$:

Array A	1	4	0	2	3	2
---------	---	---	---	---	---	---

Array C	0	0	0	0	0	0
---------	---	---	---	---	---	---

Scorriamo A , e per ogni numero i , aggiungeremo 1 a $C[i]$:

Array A	1	4	0	2	3	2
---------	---	---	---	---	---	---

Array C	1	1	2	1	1	0
---------	---	---	---	---	---	---

Avendo adesso C riempito, lo riscorreremo, ed ogni indice i di C verrà copiato in A tante volte quanto vale $C[i]$, ordinando così l'array A :

Array A	0	1	2	2	3	4
---------	---	---	---	---	---	---

Array C	1	1	2	1	1	0
---------	---	---	---	---	---	---

```

def Counting_sort (A):
    k=max(A) #O(n)
    n=len(A) #O(1)
    C[0]*(k+1) #O(k)
    for j in range(n):# n volte
        C[A[j]] +=1 #O(1)
        #C[i] ora contiene il numero di elementi uguali a i
    j = 0 #O(1)
    for i in range(k):
        while (C[i] > 0) #C[i] volte
            A[j]=i #O(1)
            j+=1 #O(1)
            C[i]-=1

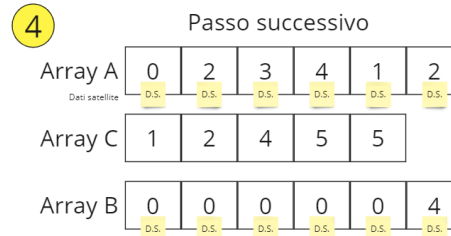
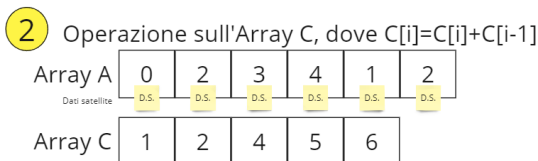
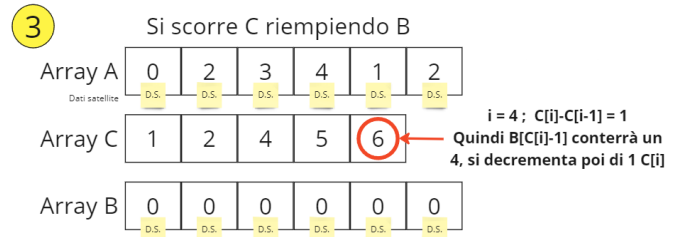
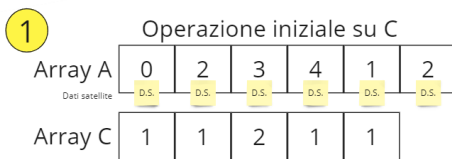
```

Counting Sort con dati satellite

L'algoritmo mostrato appena precedentemente è adeguato nel solo caso non vi siano **dati satellite**, di fatti, il ciclo che riempie il vettore A ne sovrascrive i dati, per mantenere la struttura corretta nel caso dovessero essere presenti dati satellite, dovremmo introdurre un nuovo vettore B di n elementi.

Dopo aver conteggiato il vettore C e riempito A , Si esegue un'altra iterazione su C , da sinistra verso destra partendo dall'elemento $C[1]$, nella quale ogni elemento $C[i]$, assume valore $C[i] + C[i - 1]$, ed indica quanti elementi con valore uguale a $C[i]$ sono presenti.

Dopo aver modificato C , si copia in B ogni elemento di $A[j] = k$ nella posizione giusta che sarebbe $C[k]$, decrementando di 1 $C[k]$, Inoltre, iterando C , $C[i] - [i - 1]$ ci dirà quanti valori uguali a $C[i]$ sono presenti.



```
def Counting_sort_con_Dati_Satellite (A)
    k=max(A); n=len(A)
    C[0]*(k+1); B=[0]*n
    for j in range(n)
        C[A[j]]+=1 #in C[i] ora c'è il # di elem. = i
    for i in range(1,k)
        C[i]+=C[i-1] #in C[i] ora c'è il # di elem. ≤ i
    for j in range(n, -1)
        B[C[A[j]]]=A[j]
        C[A[j]]-=1
    return B
```

Bucket Sort

Il Bucket Sort è un altro algoritmo di ordinamento lineare, dal costo medio di $\Theta(n)$ che necessita di un'ipotesi, gli n elementi da ordinare devono essere distribuiti in modo **uniforme** sull'intervallo $[1, k]$, senza restrizioni su k . L'idea è quella di dividere l'intervallo $[1, k]$ in n sotto-intervalli di uguali dimensioni $\frac{k}{n}$ detti **bucket**, e distribuire gli elementi nei bucket. Poiché è valida l'ipotesi che gli elementi siano uniformemente distribuiti, non ci si aspetta che molti elementi cadano nello stesso bucket.

Si vuole ordinare un array di 5 elementi nell'intervallo [1,9]

Array A

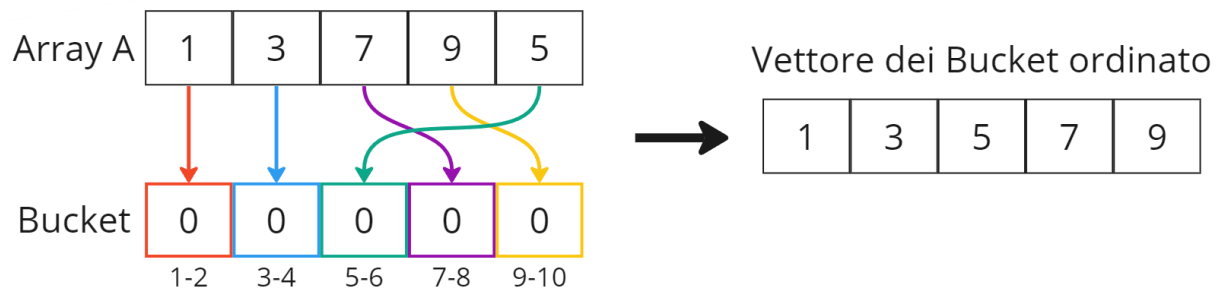
1	3	7	9	5
---	---	---	---	---

bucket, 5 intervalli di dimensione $9/5=2$ (arrotondato per eccesso)

Bucket

0	0	0	0	0
1-2	3-4	5-6	7-8	9-10

Inseriamo gli elementi di A nel vettore bucket B , essendo uniformemente distribuiti, nessun elemento di A ricadrà nello stesso bucket :



Lo pseudo codice :

```
void bucketSort(float arr[], int n)
{
    float max=findMax(arr,n);
    vector<float> b[n];
    // 2) Put array elements in different buckets
    for (int i=0; i<n; i++)
    {
        int bi = n*arr[i]/(max+1); // Index in bucket
        b[bi].push_back(arr[i]);
    }
    // 3) Sort individual buckets
    for (int i=0; i<n; i++)
        sort(b[i].begin(), b[i].end());
    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
```

```
    arr[index++] = b[i][j];  
}
```

Strutture dati

Cos'è una struttura dati

Sappiamo che in un linguaggio di programmazione, un **dato** è un valore che una variabile può assumere. Le **strutture dati** sono particolari tipi di dato, caratterizzate dall'**organizzazione dei dati** in modo matematico, e presentano un insieme di **operazioni** che permettono di manipolare tali dati. Una struttura dati può essere **lineare**(i dati sono organizzati in modo sequenziale) o **non lineare**(i dati non sono organizzati in modo sequenziale), inoltre possono essere **statiche** (di dimensione fissa e prestabilita) o **dinamiche** (di dimensione variabile), inoltre una struttura dati è **omogenea** se i dati organizzati all'interno di essa sono dello stesso tipo, altrimenti è **disomogenea**.

Strutture dati fondamentali - Insieme dinamico

L'insieme dinamico, è una **struttura dati astratta**, in cui gli elementi, possono variare nel tempo in funzione delle operazioni compiute dall'algoritmo che li utilizza. Gli elementi sono denominati **record**, e presentano una **chiave** univoca per ogni elemento, utilizzata per distinguerli all'interno dell'insieme, e ulteriori **dati satellite**, relativi all'elemento stesso, ma non direttamente utilizzati nelle operazioni di manipolazione

matricola	data di nascita
cognome	
nome	
esami sostenuti	

In questo caso la **matricola** è la chiave, univoca e differente per ogni **record** (in questo caso gli studenti), “data di nascita”, “cognome”, “nome” e “esami sostenuti” sono tutti **dati satellite**, quindi attributi associati all'elemento specifico.

Gli insiemi dinamici presentano 2 tipi di operazioni :

- **Operazioni di interrogazione** - non modificano la consistenza dell'insieme, come :
 1. `Search(S, k)` : Ricerca l'elemento di chiave `k` contenuto nell'insieme `S`.
 2. `Min(S)` o `Max(S)` : Recupera il minimo o il massimo valore presente nell'insieme `S`.
 3. `Predecessor(S, k)` o `Successor(S, k)` : recuperano l'elemento presente in `S` che precede o segue l'elemento di chiave `k`.
- **Operazioni di manipolazione** - vanno a modificare la consistenza dell'insieme, come :
 1. `Insert(S, k)` : Inserisce l'elemento di chiave `k` nell'insieme `S`.
 2. `Delete(S, k)` : Elimina l'elemento di chiave `k` nell'insieme `S`.

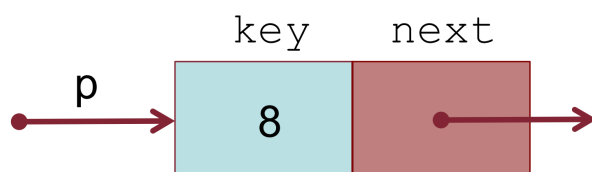
Le differenti strutture dati sono caratterizzate dalle proprietà che le caratterizzano e dai rispettivi costi computazionali associati alle loro operazioni. **L'array** è un esempio di struttura dati statica. Vediamo adesso altri esempi di strutture dati molto importanti e di comune reperibilità.

Lista puntata

Necessitiamo di una definizione :



Un **Puntatore**, non è altro che una variabile che assume come valore un **indirizzo di memoria**, facendo indirettamente riferimento ad una variabile (contenuta appunto nell'indirizzo stesso).

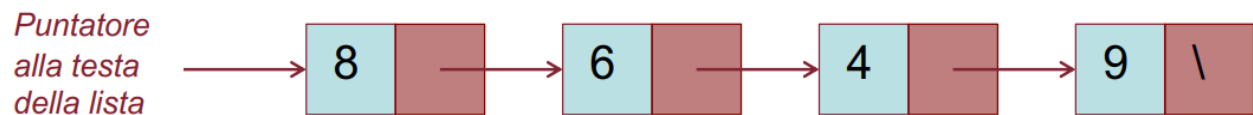


Ogni elemento della lista puntata ha quindi 2 campi fondamentali, il campo **key**, che contiene l'informazione vera e propria, ed il campo **next**, contenente il puntatore che consente l'accesso all'elemento successivo della lista.

Nello pseudo codice, utilizzeremo la sintassi

`p->key` per riferirci al contenuto del campo `key` puntato dal puntatore `p`, e `p->next` per riferirci al contenuto del campo `next` puntato dal puntatore `p`, ossia l'elemento successivo.

L'accesso avviene sempre ad un'estremità, per mezzo di un puntatore che identifica il primo elemento della lista, inoltre ogni elemento contiene un puntatore che identifica l'elemento successivo, l'accesso è quindi permesso sequenzialmente.



L'accesso diretto non è quindi consentito, facendo sì che nel caso peggiore, la selezione di un elemento può assumere costo $\theta(n)$.

operazioni sulla lista puntata

Vediamo adesso le operazioni definite nella lista puntata.

Ricerca

```
def Search (p: puntatore alla lista; k: valore)
    p_corr = p
    while ((p_corr ≠ NULL) and (p_corr-> key ≠ k))
        p_corr = p_corr-> next
    return p_corr
```

Assume costo $O(n)$.

Inserimento

```
def Insert_in_testa (p: puntatore alla lista; k: punt. alla lista)
    if (k ≠ None)
        k->next = p
    p = k
    return p
```

Assume costo $\theta(1)$.

Nell'inserimento il puntatore `k` per l'elemento da inserire va creato tramite una **allocazione di memoria**, è questo ciò che rende la lista una struttura dati dinamica-

Vediamo poi come implementare una funzione che inserisca un elemento in una qualsiasi posizione.

```
def Insert_Dopo_d(p: punt. testa;
                  k: punt. a elem. da ins.)
    if d ≠ None
        k->next = d->next;
        d->next = k;
        return p
    else
        return None
```

Eliminazione

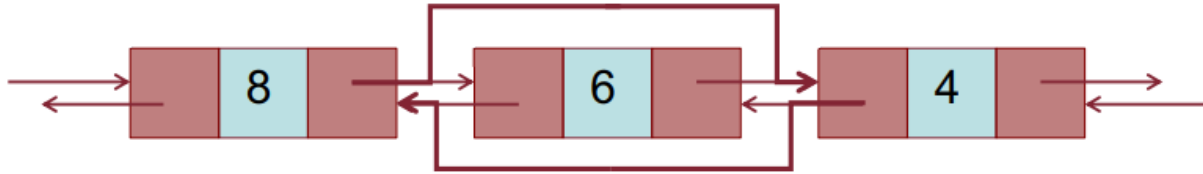
```
def Delete (p: punt. alla lista;
            k: puntat. all'elem. da cancellare)
    if (k ≠ None) // se k=NULL non c'è niente da cancellare
        if k = p // cancel. 1° elem
            p = p->next; return p
        p_corr = p
        while (p_corr-> next ≠ k)
            p_corr = p_corr-> next // qui, p_corr punta all'elem
        p_corr-> next = k-> next
    return p
```

Assume costo $O(n)$.

Liste doppiamente puntate

Alcuni problemi riscontrati nelle liste puntate semplici (ad esempio complessità lineare nella cancellazione) possono essere risolti organizzando una struttura dati, nella quale ogni elemento

punta si all'elemento successivo, ma anche all'**elemento precedente**, tale struttura si chiama **lista doppia** o **lista doppiamente puntata**.

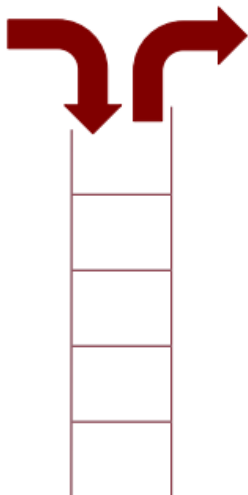


Riassumendo in una tabella i costi delle operazioni delle liste puntate abbiamo :

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k)	Insert(S,k)	Delete(S,k)
Lista semplice	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$
Lista doppia	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Pila

La **Pila** è una struttura dati astratta, assume un comportamento **LIFO** (Last In First Out), ossia, gli elementi inseriti nella struttura, verranno estratti in ordine contrario di inserimento, l'ultimo elemento inserito, sarà il primo estratto.



Non ha quindi senso parlare di relazioni d'ordine tra gli elementi. Un esempio di applicazione è la **pila di sistema**, che gestisce le chiamate a funzione ricorsive, eseguendo per prima, l'ultima funzione richiamata. Per definizione, la pila ha solo 2 funzioni, inserimento, detto **Push**, e rimozione, detto **Pop**, tali operazioni agiscono sulla stessa estremità, definita **Top**, nome del medesimo puntatore, non è quindi necessario gestire le operazioni di ricerca.

Vediamo la funzione **Push** :

```
Func Push(Top : puntatore, E : puntatore per l'elemento da inserire)
    e->next = top;
    top = e;
    return top;
```

Tale funzione ha costo $\theta(1)$.

Vediamo la funzione

Pop:

```
Func Pop(Top : puntatore):
    if(Top==None):
        scrivi "errore, coda vuota";
        return None;
    e=Top;
    Top= e->Next;
```

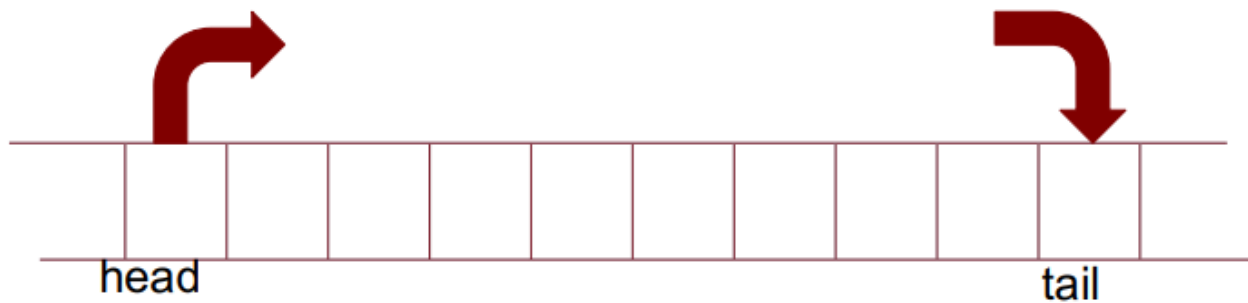
```
e->Next = None; #separo l'elemento dalla pila
return e, Top;
```

Tale funzione ha costo $\theta(1)$.

Risulta chiaro come siano strutture dati estremamente efficienti, ma non molto duttili.

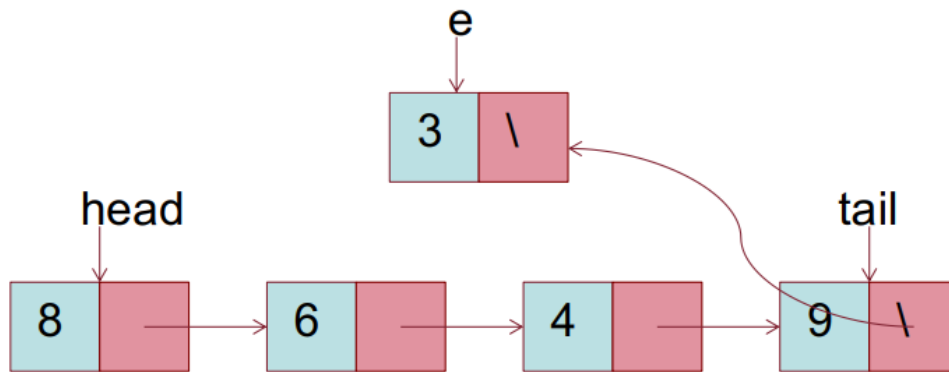
Coda

La Coda è una struttura di tipo **FIFO** (First In First Out), ossia, l'oggetto da estrarre è quello presente nella struttura dati da più tempo, anche la coda ha 2 funzioni, l'inserimento, detto **Enqueue**, e la rimozione, detta **Dequeue**, un esempio di applicazione è la coda di stampa. L'Enqueue opera sull'estremità detta "Head", il Dequeue opera sull'estremità detta "Tail".



Vediamo la funzione **Enqueue**:

```
Func Enqueue(Head : Punt; Tail : Punt; e : Punt. per l'elemento
  if(Tail==None): #la coda è vuota
    Tail = e;
    Head = e;
  else:
    Tail->Next = e;
    Tail = e;
  return Head, Tail;
```



Tale funzione ha costo $\theta(1)$.

Vediamo la funzione **Dequeue**:

```

Funzione Dequeue (head: puntatore; tail: puntatore)
  if (head == None) //la coda è vuota
    scrivi "Errore: coda vuota";
    return NULL;
  e = head;
  head = e->next;
  if (head == None):
    tail = None;
  return head, tail, e;

```

Tale funzione ha costo $\theta(1)$.

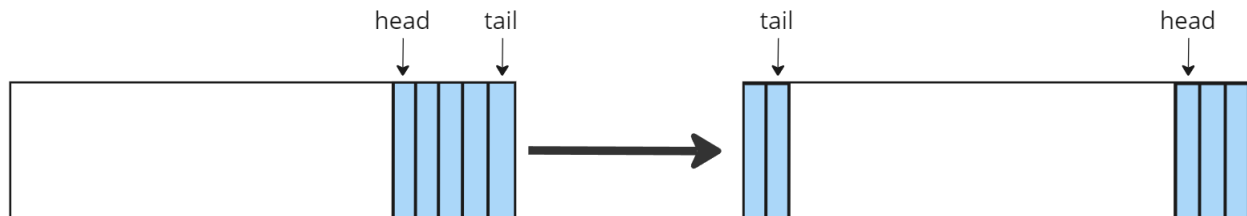
Implementazione con Array

Per implementare una coda o una pila possiamo utilizzare le liste o gli **array**, il problema è che gli array sono **statici**, quindi bisogna gestire il vettore in maniera **circolare**, cioè, **il primo elemento è successore dell'ultimo**, essendo però un array, avrà una dimensione finita n .

Bisogna però controllare che l'head o la tail non raggiungano l'ultimo elemento dell'array, dato che i puntatori, man mano che vengono fatte operazioni, si spostano sempre più verso destra,

dato che l'inserimento e l'estrazione tendono a **spostare gli elementi verso la fine dell'array**.

Essendo però circolare, raggiungendo il limite a destra, la coda ricomincerà a scorrere da sinistra, tipo effetto pac-man.



Code e Pile con Priorità

Vediamo adesso una terza struttura dati, che è però una generalizzazione di code e pile, è una variante di esse, con la differenza che **la posizione di ciascun elemento non dipende dall'ordine di inserimento**, ma da un determinato **valore**, cioè una relazione d'ordine prestabilita detta **priorità**.

Ogni elemento della coda/pila avrà quindi 3 valori fondamentali, la **key**, il puntatore **next** per l'elemento successivo, ed il valore della **priorità** che ne determinerà la posizione all'interno della coda/pila. Un tipico esempio è la *coda di sistema*.

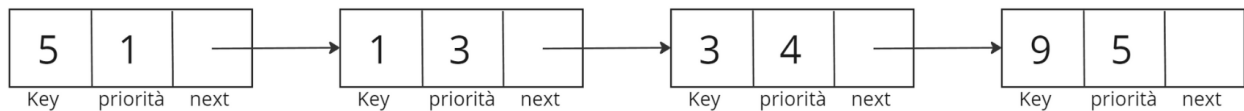
Si vuole aggiungere il seguente elemento:



Alla seguente coda :



Diventerà quindi, ordinato in base al valore di priorità :

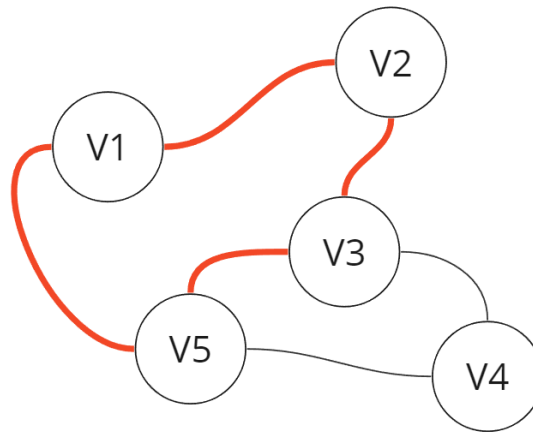


Un tipico problema comune delle code/pile con priorità è la

Starvation, ossia quella situazione in cui un elemento con un valore di priorità basso, viene costantemente anticipato da valori, seppur aggiunti dopo, con una priorità più alta, facendo sì che tale elemento rimanga nella coda/pila troppo a lungo senza mai essere estratto, una possibile soluzione a tale problema è quello di aumentare gradualmente il valore di priorità in base alla permanenza di un elemento nella coda/pila.

Alberi

L'albero è una struttura dati molto versatile, utile per modellare una grande quantità di situazioni reali, gli alberi sono dei particolari tipi di **Grafi**. Un Grafo è un insieme costituito da 2 elementi, un insieme di **nodi V** ed un insieme di **archi E**, cioè coppie non ordinate di nodi. Un **Cammino** è una sequenza di archi, se il cammino *ritorna* al suo nodo di origine è detto **Ciclo**.



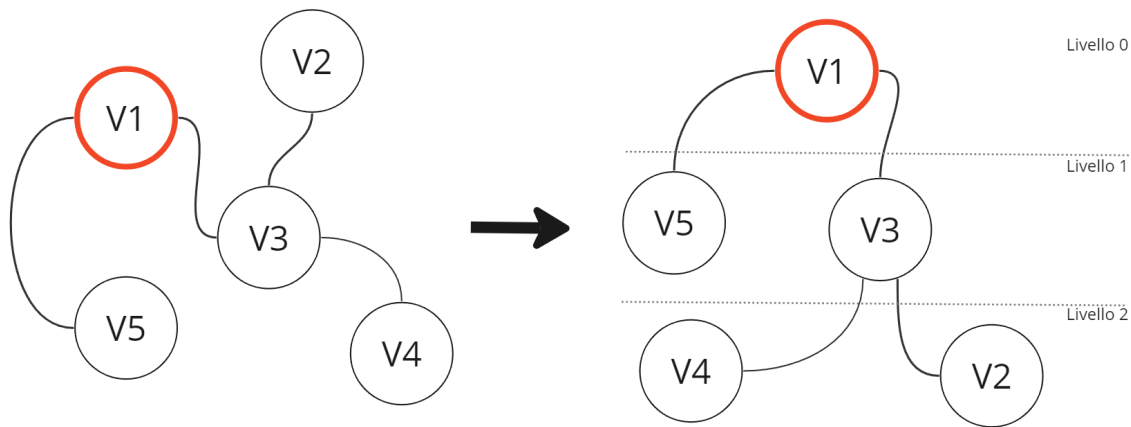
In rosso, evidenziato un **cammino ciclico**.

Un grafo si dice **Connesso** se per ogni coppia di nodi, esiste almeno un cammino, e **Aciclico** se privo di cicli. Gli alberi non sono altro che grafi connessi ed aciclici, inoltre si noti che se ad un grafo connesso ed aciclico rimuoviamo un arco qualsiasi, otteniamo due grafi diversi con le stesse proprietà.

Il numero degli archi in un albero è uguale a $\#E = N - 1$, dove E è l'insieme degli archi ed N il numero dei nodi.

Alberi Radicati

Gli alberi radicali sono alberi, i quali si distinguono per avere un singolo nodo principale detto **radice**, posto “superiormente” agli altri facendo sì che tutti i cammini percorrano una direzione dall’alto verso il basso.



V1 è il nodo **Radice**

I nodi vengono suddivisi in **livelli** in base alla distanza dalla radice, quindi il cammino più lungo da un nodo al nodo radice rappresenta il nodo al livello più alto, tale valore è detto **altezza h** . Preso un qualsiasi livello k il suo adiacente al livello $k - 1$ è detto padre, ed il suo adiacente al livello $k + 1$ è detto figlio. Quindi in un albero radicato è introdotto il concetto di ordinamento.

Una particolare sottoclasse degli alberi radicati sono gli **alberi binari**, dove ogni nodo ha al massimo 2 figli, detti **figlio sinistro** e **figlio destro**.

Rappresentazione in memoria

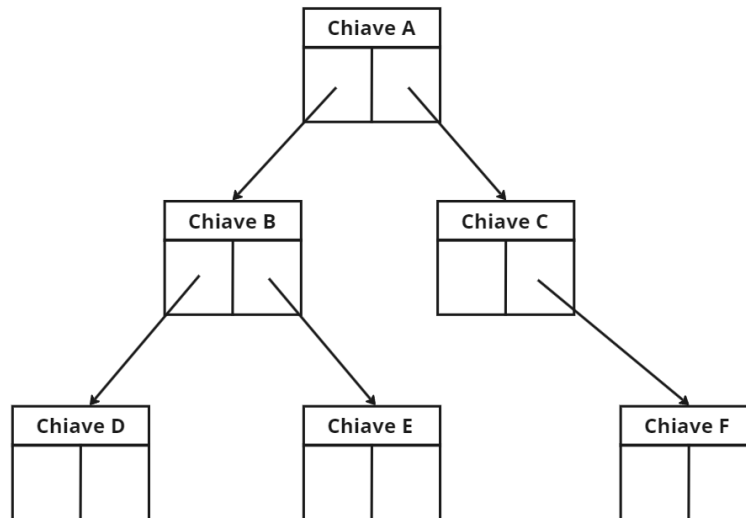
Quando si parla di **implementare fisicamente un albero in memoria**, si hanno diverse soluzioni, ognuna di esse con vantaggi e svantaggi dovuti poi alla lettura e manipolazione dell'albero.

Rappresentazione tramite record e puntatori

In questa rappresentazione ogni nodo è un record, contenete :

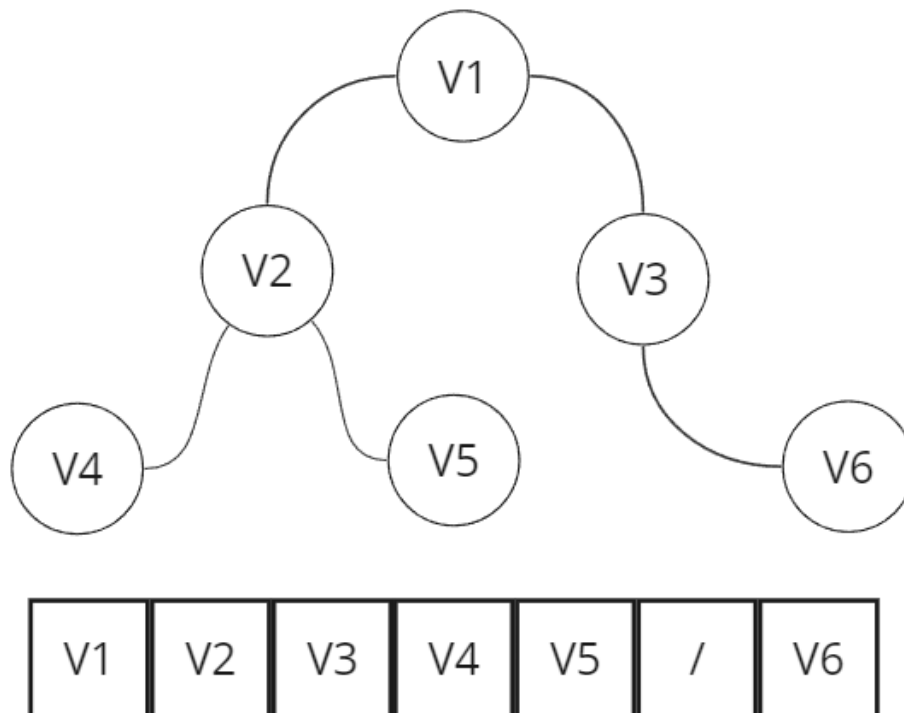
- **Key** : Chiave con l'informazione del nodo stesso
- **Left** : Figlio sinistro
- **Right** : Figlio destro

I figli, destro e sinistro, contengono i puntatori che indirizzano rispettivamente i nodi figli, tale rappresentazione gode dei vantaggi delle strutture dinamiche basate sui puntatori, ma è accessibile esclusivamente in modo sequenziale.



Rappresentazione Posizionale

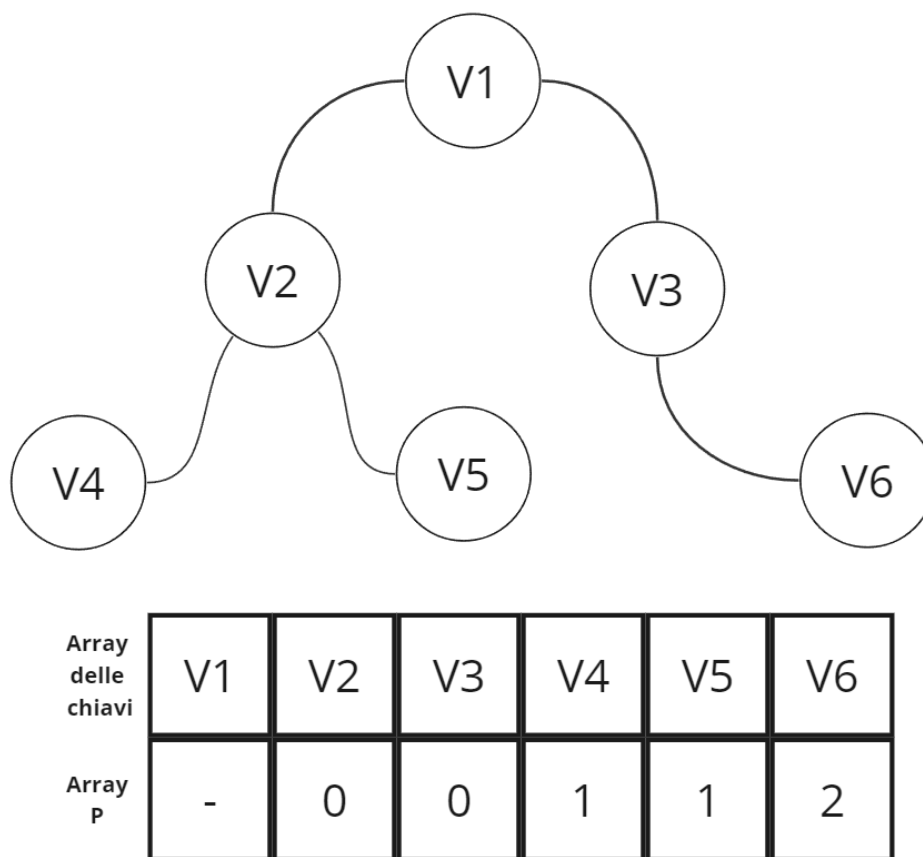
Un'altra possibile rappresentazione è quella posizionale tramite un Array, dove si inseriscono in sequenza tutti i nodi all'interno di un array, dove ogni nodo di indice i , avrà i suoi figli nei rispettivi indici $2i + 1$ e $2i + 2$.



Un problema di tale rappresentazione è il fatto che laddove un nodo presenta un solo figlio, comunque esso per far sì che l'indicizzazione funzioni, occuperà comunque posizione all'interno della memoria, è quindi poco efficiente dal punto di vista di immagazzinamento.

Rappresentazione con Vettore dei Padri

L'ultimo modo di rappresentare un albero è tramite il vettore dei padri, tale rappresentazione sfrutta due array, un **array di chiavi**, contenente tutti i nodi presenti nell'albero in modo non ordinato, ed un array P , nella quale ogni elemento di indice i , corrisponde all'indice del padre dell'elemento $P[i]$ nell'array di chiavi.



Confronto tra tipi di rappresentazione degli alberi

Mettiamo a confronto (riguardo il costo computazionale) le diverse rappresentazioni su alcune operazioni basilari di interrogazione.

Trovare il padre di un nodo

- **Struttura a puntatori** : Al momento non si è in grado di farlo se non implementando un algoritmo più complesso.
- **Rappresentazione posizionale** : Il padre del nodo i banalmente è nella posizione $\lfloor \frac{i-1}{2} \rfloor$ ed assume costo $\theta(1)$.
- **Vettore dei padri** : L'indice del padre di ogni nodo i è memorizzato direttamente nell'elemento $P[i]$ dell'array ed assume costo $\theta(1)$.

Determinare numero dei figli di un nodo

- **Struttura a puntatori** : Si verifica se i campi left e right siano settati a `None` oppure no, assume costo $\theta(1)$.
- **Rappresentazione posizionale** : Si vede se gli elementi di indice $2i - 1$ e $2i$ sono settati a `—` oppure no, assume costo $\theta(1)$.
- **Vettore dei padri** : Si deve scorrere l'intero array P e contarvi il numero di occorrenze dell'elemento i (l'indice del nodo di cui vogliamo contare i figli), assume costo $\theta(n)$.

Determinare la distanza di un nodo dalla radice

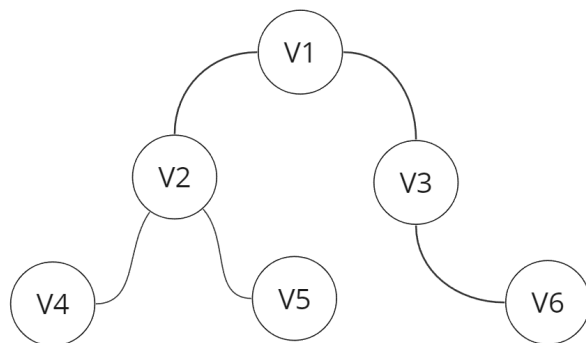
- **Struttura a puntatori** : Al momento non si è in grado di farlo se non implementando un algoritmo più complesso.
- **Rappresentazione posizionale** : Il livello del nodo i (e quindi la sua distanza dalla radice) è banalmente $\log(i + 1)$, assume costo $\theta(1)$.
- **Vettore dei padri** : A partire da i risaliamo di padre in padre passando per $P[i]$, $P[P[i]]$, $P[P[P[i]]]$, ecc. fino alla radice, assume costo $\theta(h)$.

Visite ad alberi

Un'operazione basilare sugli alberi è l'accesso a tutti i suoi nodi, l'accesso progressivo a tutti i nodi dell'albero si chiama **visita all'albero**, essendo l'albero una struttura non lineare, ci sono diversi modi di accedere ai nodi.

- **Visita in preordine** - il nodo è visitato prima di proseguire la visita nei suoi sottoalberi.
- **Visita inordine** - il nodo è visitato dopo la visita del sottoalbero sinistro e prima di quella del sottoalbero destro.

- **Visita in postordine** - il nodo è visitato dopo entrambe le visite dei sottoalberi.



Visita in preordine :

$V1 - V2 - V4 - V5 - V3 - V6$

Visita inordine :

$V4 - V2 - V5 - V1 - V3 - V6$

Visita in post-rdine :

$V4 - V5 - V2 - V6 - V3 - V1$

Il costo computazionale della visita ad un albero nel caso generale risulta $\theta(n)$.

Applicazioni delle visite

Vediamo alcune applicazioni utili delle visite accompagnate da esempi di pseudocodice.

Calcolo altezza dell'albero

```

def Calcola_h (p):
    if (p==None): return -1 #albero vuoto
    if (p->left==None AND p->right==None): return 0
    h = max{Calcola_h(p->left),Calcola_h(p->right)}
    return h + 1
  
```

Dato che si calcola prima l'altezza del figlio sinistro poi di quello destro, e solo in seguito si processa sul nodo "padre", tale visita segue una filosofia in post-rdine, ha costo $\theta(n)$.

Vediamo una visita più complessa

Conteggiare i nodi presenti ad un livello k

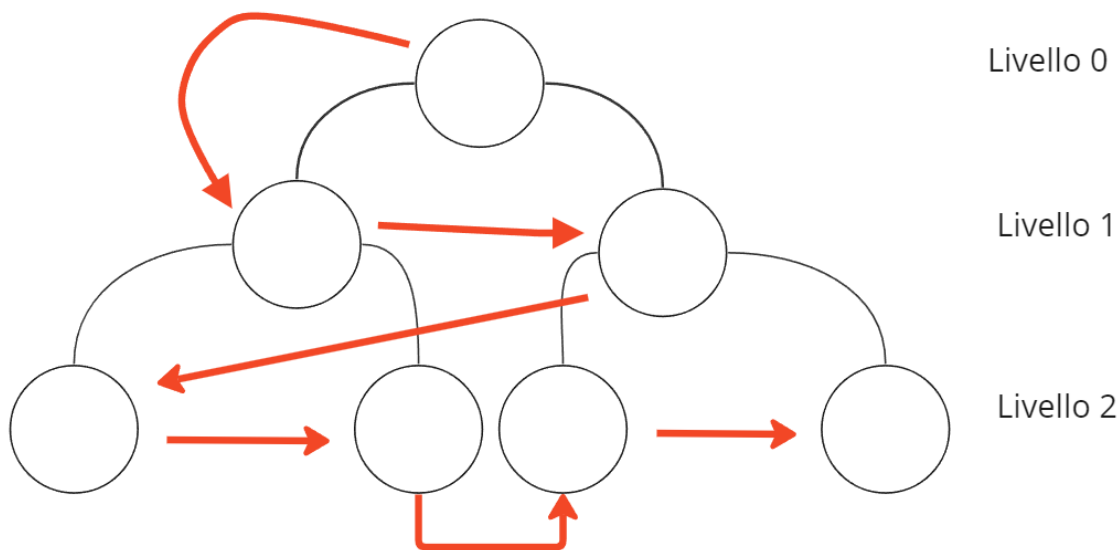
```

def Conta_k (p; k,i: interi):
    if (p == None): return 0 #albero vuoto
    if (k==i): return 1
  
```

```
k_left = Conta_k(p->left,k,i+1)
k_right = Conta_k(p->right,k,i+1)
return k_left + k_right
```

Visita per livelli

Un altro metodo efficace per visitare un albero è di farlo **per livelli**, ossia visitare sequenzialmente tutti i nodi del livello 0, poi del livello 1... e così via fino al livello h .



Visitare i nodi secondo l'ordine designato dalle frecce rosse.

Non possiamo considerare un algoritmo analogo alle visite viste precedentemente, bisogna procedere diversamente, utilizzeremo una **coda di appoggio**, nella quale inserire i nodi nell'ordine corretto per poi iterare su tale coda, estraendo i dati per poi visitarli. Supponiamo che i valori da inserire nella coda siano i puntatori ai nodi dell'albero.

```
def Visita_per_livelli (r; head, tail):
    if (r==None): return
    Enqueue(head, tail, r)
    while (!CodaVuota(head))
        p = Dequeue(head, tail)
        print(p->key)
        if (p->left!=None): Enqueue(head, tail, p->left)
```



```
if (p->right!=None): Enqueue(head, tail, p->right)
return
```

La caratteristica di questa coda, è che vengono inseriti tutti i nodi di livello i prima di un qualsiasi nodo di livello $i + 1$, ricordando che la coda è una struttura FIFO, l'ordine di estrazione sarà il medesimo di inserimento. Ogni operazione sui nodi ha costo costante, Quindi tale algoritmo ha costo $\Theta(n)$.

I dizionari

Il Dizionario è una struttura dati astratta che permette di gestire un insieme dinamico di dati, di norma **totalmente ordinato** tramite queste sole tre operazioni :

- Insert
- Search
- Delete

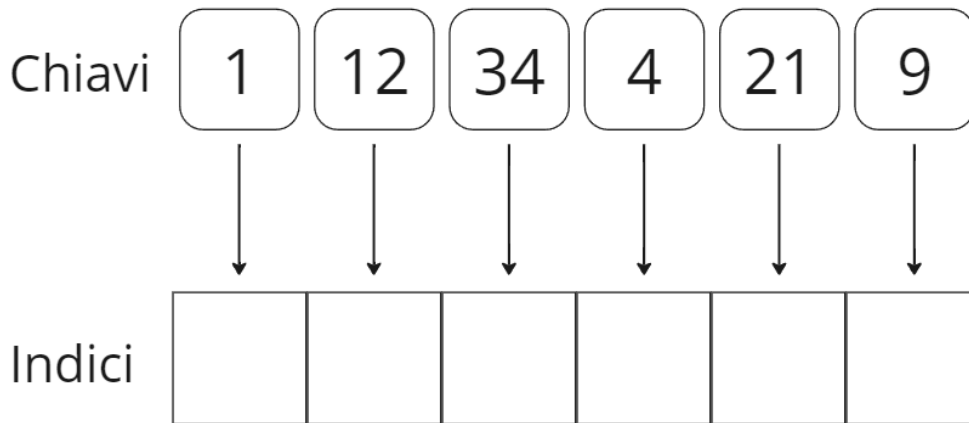
Vediamo alcune definizioni di nomenclatura riguardanti i Dizionari :

- **“U” Insieme Universo** - Insieme di tutti i valori che le chiavi possono assumere (Ad esempio, essendo le targhe composte da 4 lettere e 3 numeri, l'insieme universo di un dizionario che deve identificare delle targhe conterrà tutte le possibili permutazioni, cioè $26^4 \cdot 10^3$).
- **“m”** - Il numero massimo delle posizioni a disposizione nella struttura dati la *capienza*.
- **“n”** - Il numero di elementi da memorizzare nel dizionario, le chiavi di ogni elemento sono tutte diverse tra loro.

Esistono 3 modi diversi per implementare un dizionario.

Tabelle ad indirizzamento diretto

È un vettore nella quale gli indici e le chiavi sono in **corrispondenza biunivoca**, quindi deve essere disponibili un indice per ogni elemento possibile dell'insieme universo.



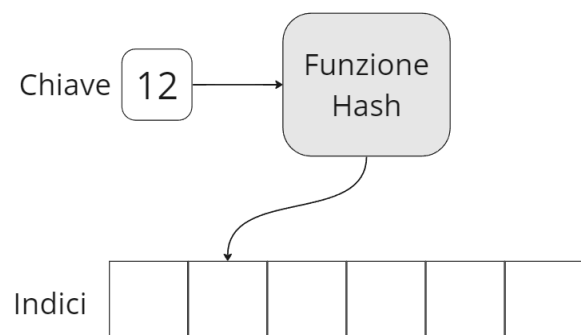
Facendo tale assunzione tutte le operazioni sul dizionario avrebbero costo costante, però **nella realtà l'insieme U è spesso enorme**, tanto da rendere impossibile l'allocazione di un array di sufficiente capienza, inoltre spesso le chiavi effettivamente utilizzate dall'insieme U sono di un ordine decisamente minore, sarebbe quindi un rilevante **spreco di memoria** dover lasciare un indice libero per ogni elemento di U, quando di questo ne verrà mappata solo una frazione.

Ad **esempio**, tutti i possibili codici fiscali sono $2 \cdot 10^{18}$, ma non verranno mai mappati tutti dato che nel mondo reale esistono solamente $6 \cdot 10^7$ cittadini circa.

Tabelle Hash

Si utilizzano quando l'insieme U è molto più grande dell'insieme delle chiavi da memorizzare (Si veda l'esempio sui codici fiscali).

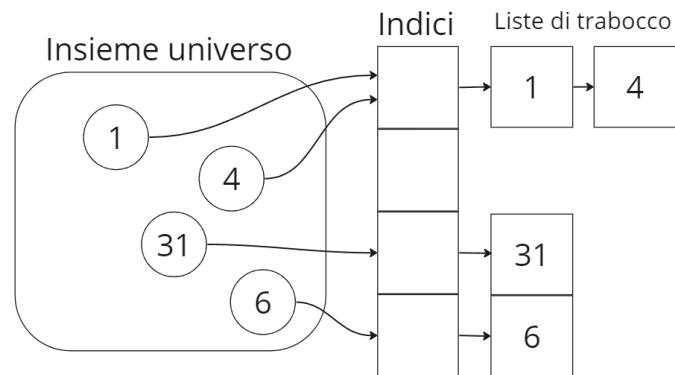
L'idea è quella di utilizzare un array di m (capienza della struttura dati) posizioni, essendo esse molte meno della cardinalità di U, si utilizza un opportuna funzione **Hash**, Che prende in input una chiave, e secondo alcuni parametri ne restituisce un indice.



Si immagini la funzione Hash come una **scatola nera**, che presa una chiave, ne tira fuori un indice secondo dei parametri pseudo-casuali, potrebbe però accadere che più chiavi vengano erroneamente assegnate dallo stesso indice dalla funzione Hash, tale problema viene definito con

il nome di **collisione**, affinché non si generino collisioni il dizionario funziona correttamente, per questo quando avvengono vanno gestite prontamente, la prima cosa da fare è rendere la funzione Hash il più **equiprobabile** possibile, facendo apparire come “casuale” il valore risultante, deve anche essere **deterministica**, se applicata più volte alla stessa chiave deve fornire lo stesso indice come risultato. Una funzione Hash con tali proprietà si riassume nel termine di **uniformità semplice** della funzione.

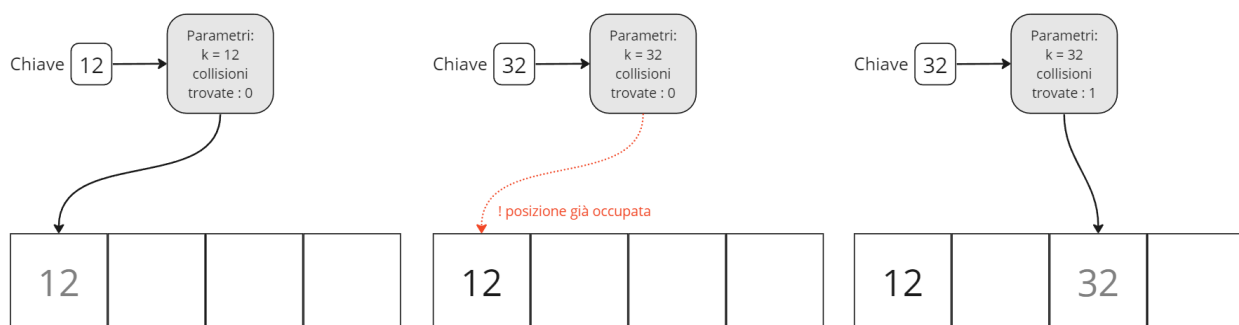
L’uniformità semplice **minimizza la possibilità di collisioni**, ma esse, seppur con meno probabilità, si presenteranno comunque. Tale problema sarà gestito da una **Lista di trabocco**, una tecnica che consiste nell’aggiungere ad una **lista concatenata** tutte le chiavi mappate sullo stesso indice.



Ogni volta che si vuole inserire un elemento però, la chiave dovrà iterare per tutta la lista di trabocco, un inserimento avrà quindi costo $O(n)$, essendo però che la funzione Hash gode di uniformità semplice, sarà molto improbabile che una lista di trabocco abbia più elementi, quindi il caso medio di inserimento vale $O(\frac{n}{m})$. $\frac{n}{m} = \alpha$ - **Fattore di carico** della tabella.

Un'altra possibile tecnica oltre le liste di trabocco è la tabella ad **indirizzamento aperto**.

Tale tecnica consiste nell’inserire tutti gli elementi nella tabella, è applicabile quando , è maggiore di n, ossia quando il fattore di carico non è mai maggiore di 1. La funzione Hash dipenderà da 2 parametri, la chiave k ed il numero di collisioni già trovate, facendo “camminare” la chiave nel prossimo indice calcolato finché non trova una posizione libera.



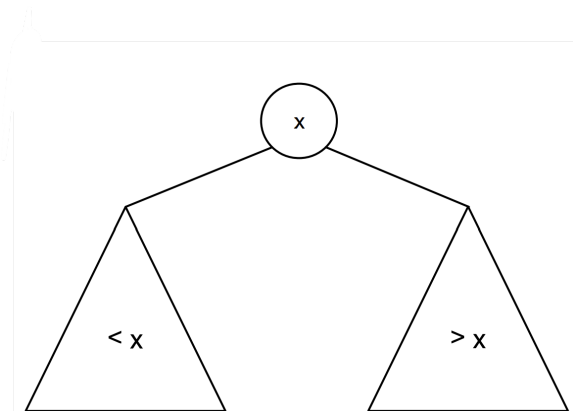
L'inserimento, nel caso peggiore ha costo $O(n)$, quando tutti gli n elementi memorizzati occupano la medesima posizione, ma nel caso medio vale $O(1/(1 - \alpha))$ dove α si ricordi essere il fattore di carico, in questo caso sempre ≤ 1 .

La ricerca funziona nel medesimo modo dell'inserimento fino a quando non si incontra l'elemento cercato, ne condivide anche i costi computazionali.

La **cancellazione non è supportata**, se si lascia un elemento vuoto non si può recuperare qualsiasi elemento memorizzato in caselle successive.

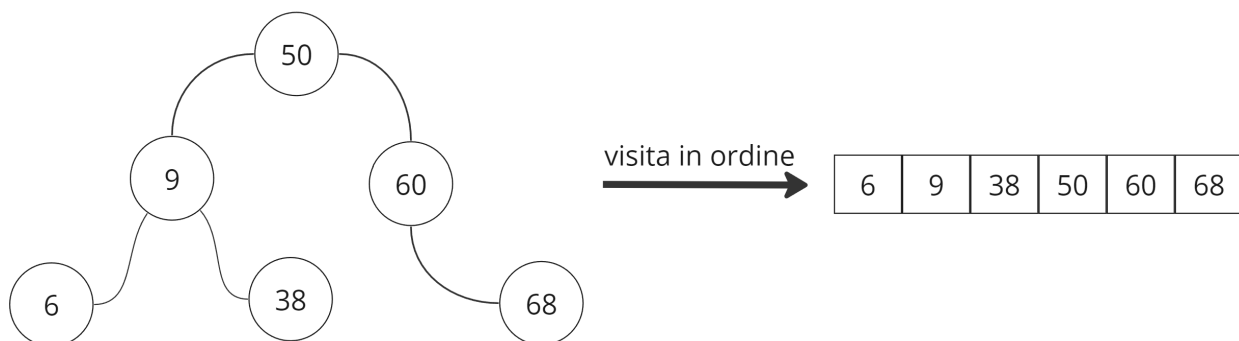
Alberi binari di ricerca

Gli **ABR** sono alberi con una proprietà aggiuntiva, in ogni nodo è contenuta una **chiave**, il valore di tale chiave è maggiore di tutti i valori delle chiavi dei suoi figli sinistri, e minore di tutti i valori delle chiavi dei suoi figli destri.



Gli ABR supportano tutte le operazioni già definite sugli insiemi dinamici, sono estremamente **duttili** e possono essere utilizzati come dizionari o code di priorità.

Il **minimo** dell'albero è il nodo più a sinistra, il **massimo** è il nodo più a destra, per elencare le chiavi in modo sequenziale basta compiere una visita in-ordine.



Una possibile soluzione per un algoritmo di sorting, potrebbe essere utilizzare un ABR per ordinare un insieme di numeri per poi visitarlo in-ordine.

Per implementare in memoria un ABR si possono utilizzare **record e puntatori**, ciascun nodo avrà come record una *Key*, due valori *Left* e *Right* (contenenti i puntatori ai rispettivi figli), ed un campo *Parent*, definito come il puntatore verso il padre del nodo, utile per risalire agli antenati e favorire la cancellazione.

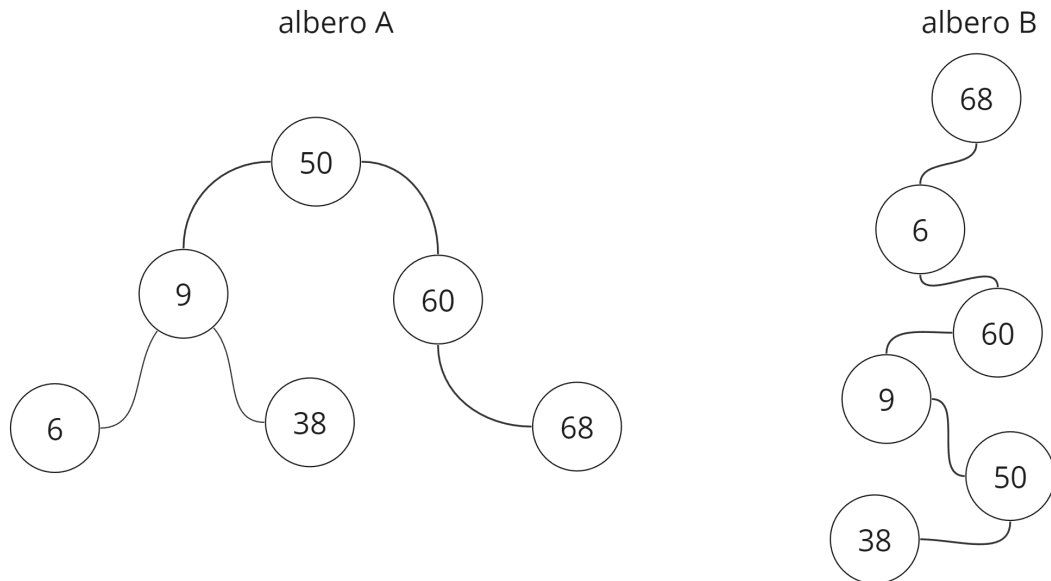
Ricerca

Riguardo la **ricerca**, si esegue una “discesa” guidata dai valori delle chiavi, si vuole cercare un valore k , ogni qualvolta che il nodo che stiamo controllando ha valore $x > k$, si controlla il figlio sinistro, se ha valore $x < k$ si controlla il figlio destro.

```
def ABR_Search(p,k):  #p è il puntatore del nodo in cui ci si trova
                      #k la chiave ricercata
    if (p==None) or (p->key==k):
        return p;
    if(k<p->key):
        return ABR_Search(p->left,k);
    else:
        return ABR_Search(p->right,k);
```

Si parte dalla radice e si percorre un cammino, potenzialmente lungo quanto l'altezza dell'albero, ha quindi costo $O(n)$, il caso medio però è molto più vicino al caso migliore, dato che un ABR di n chiavi costruito casualmente, quindi in una condizione di equiprobabilità, ha in media $\log_2(n)$ chiavi, facendo sì che il caso medio assuma costo $O(\log(n))$.

Per garantire il **costo logaritmico** è importante **bilanciare la crescita in altezza**, con opportune tecniche relativamente complesse.



Si vedano i due esempi, l'albero A e l'albero B hanno le stesse identiche chiavi, solo che l'albero A è costruito in maniera ottimale, avendo un'altezza $h = 2$, mentre l'albero B, non gode di una condizione di equi-probabilità, ha quindi altezza $h = 5$.

Inserimento

Riguardo l'**inserimento**, si esegue la stessa identica procedura della ricerca, scendendo verso il basso finché non si trova il posto libero corretto per l'inserimento della chiave.

```

def ABR_insert (p,k): #Codice scritto in modo iterativo, non ric
    y,x=None,p #y punta sempre al padre di x
    z=NodoABR(k)
    while (x != None) #discesa alla prima pos. disponib.
        y = x
        if z->key < x->key:
            x = x->left
        else:
            x = x->right
    if (y==None) #se albero inizialmente vuoto
        p = z
    else:
        if (z->key < y->key):
            y->left = z
  
```

```

else:
    y->right = z
    z->parent = y #collegam. padre - nodo da inser.
    return p

```

Il costo computazionale è uguale a quello della ricerca, in quanto si esegue appunto una ricerca affinché non si trova la posizione libero.

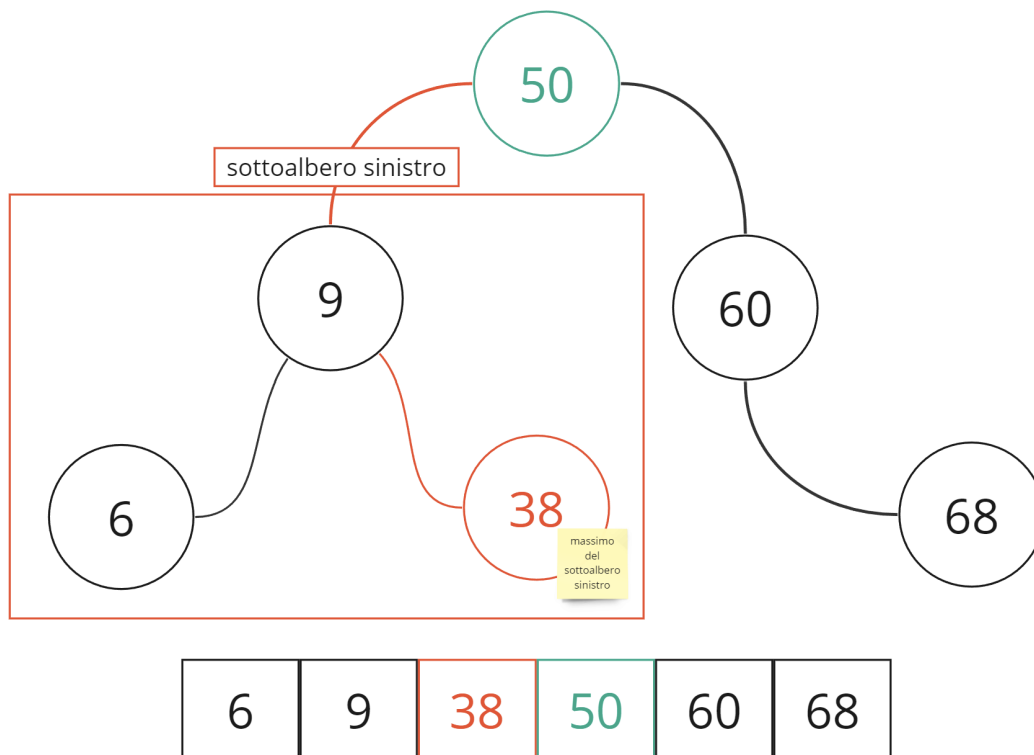
Il **massimo** e **minimo**, come già dichiarato, si trovano agli estremi destri e sinistri dell'albero, basta quindi scendere semplicemente sempre a destra/sinistra finché il figlio destro/sinistro non risulta nullo.

Predecessore e Successore

Riguardo il **predecessore** o **successore** la situazione è leggermente più complessa.

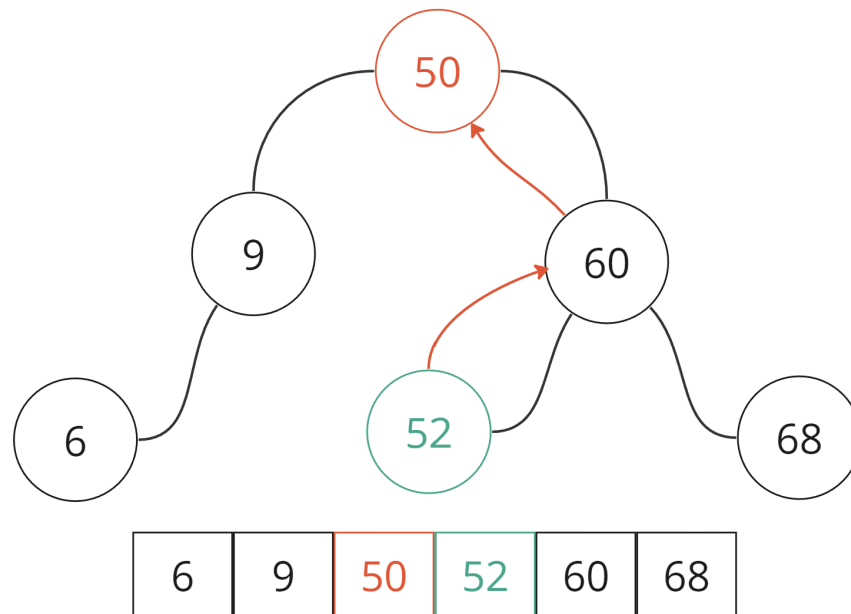
Caso 1

Si noti che il predecessore di una chiave k avente figli sinistro o destro, non è altro che il massimo del suo sottoalbero sinistro, mentre il successore è il minimo del suo albero destro.



Caso 2

Si presupponga che si vuole trovare il predecessore di un nodo k che **non ha sotto-albero sinistro**, vuol dire che è il nodo più a sinistra del suo sotto-albero. Per trovare il suo sotto-albero bisogna quindi salire verso la sua radice verso destra (essendo esso sempre figlio sinistro di nodi sinistr) affinché non si trova un nodo, che è figlio destro di suo padre, quindi si sale “verso sinistra” finché non si raggiunge la radice e sarà possibile fare un solo “passo a destra”, sarà quello il suo predecessore.



Per entrambe le funzioni il costo è limitato dall'altezza dell'albero, è quindi $O(h)$.

Come si fa però a sapere se si sta risalendo verso destra o verso sinistra? Basta controllare esplicitamente se il nodo che si sta controllando è uguale al figlio destro o al figlio sinistro del proprio padre tramite il valore *Parent*.

`if (x==x->parent->left)` se tale condizione è vera, è il figlio sinistro.

`if (x==x->parent->right)` se tale condizione è vera, è il figlio destro.

Cancellazione

Come in altre strutture dati, la **Cancellazione** risulta particolarmente tediosa, ma non impossibile, bisogna gestire 3 diverse casistiche :

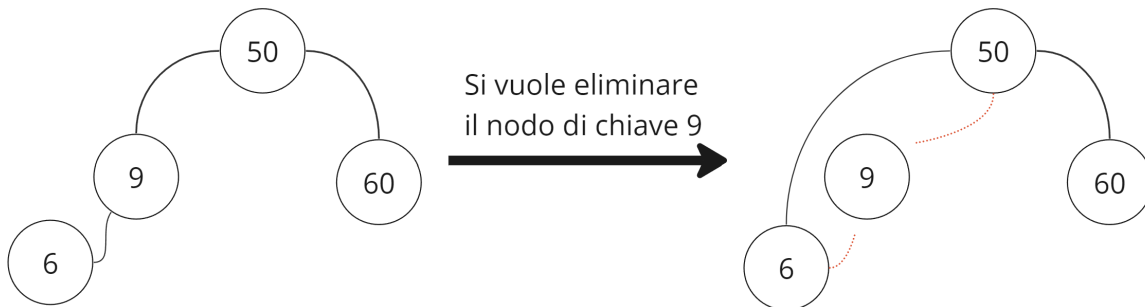
Caso 1

Se il nodo che si vuole eliminare è una **foglia**, quindi non ha ne figlio destro ne figlio sinistro, basta eliminarlo senza problemi mettendo nullo il puntatore del nodo padre.

`x->Parent = None`

Caso 2

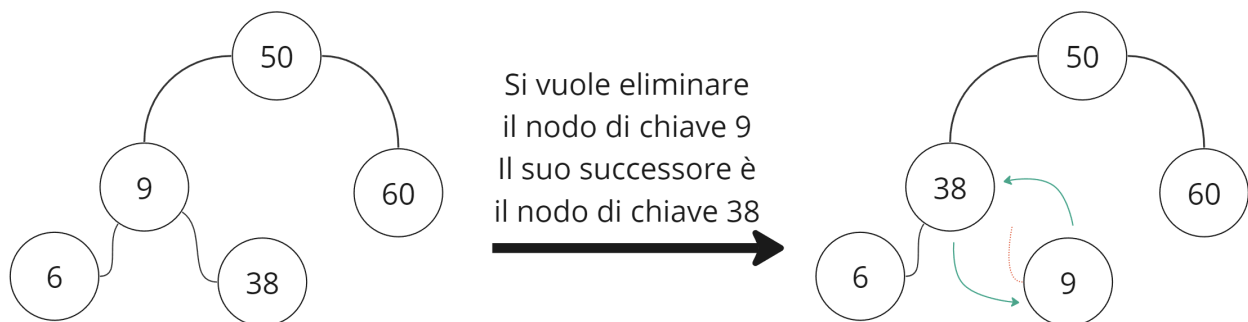
Se il nodo **ha un solo figlio**, si “cortocircuita”, collegando il campo *Parent* di tale nodo con il suddetto figlio.



`x->Parent = x->Left`

Caso 3

Se il nodo che si vuole eliminare ha sia figlio destro che sinistro, è necessario **riaggiustare l'albero** per evitare che un sottoalbero si disconnetta dalla struttura principale. Bisogna trovare un nodo già presente nell'albero da mettere nella posizione del nodo che si vuole eliminare, così da mantenere la struttura e le proprietà dell'ABR intatte. Tale nodo da sostituire non è altro che il predecessore o il successore del nodo da eliminare.



`x->Parent = Successor(x)`

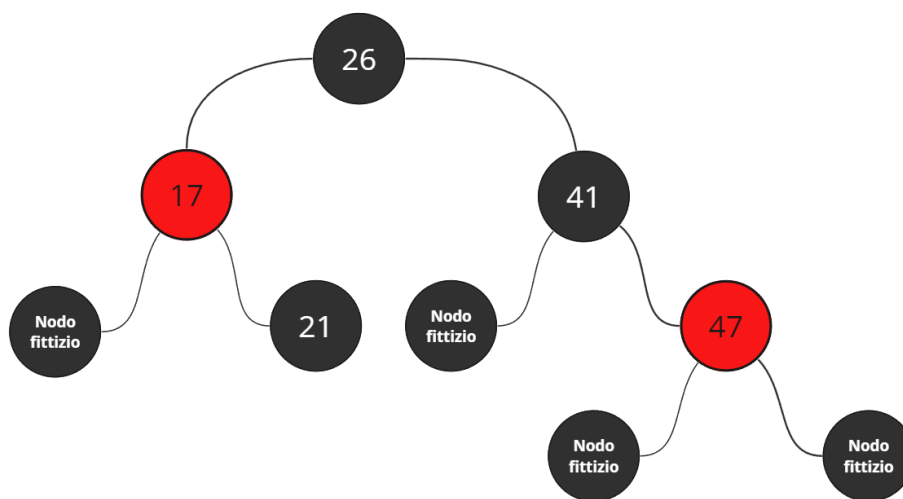
Bilanciamento dell'altezza

Vogliamo far sì che il nostro ABR sia il più completo possibile per garantire il **costo logaritmico** nelle operazioni, l'altezza deve quindi essere $h = O(\log_2(n))$, esiste una struttura dati

apposita nel garantire tale altezza.

Alberi Rosso - Neri

Gli **Alberi RB** sono degli ABR con delle proprietà aggiuntive, ogni nodo ha un campo aggiuntivo, il **colore**, che può essere rispettivamente o **rosso**, o **nero**. Agli ultimi dell'albero si aggiungono delle foglie **fittizie**, non rilevanti per l'albero ma utili nell'implementazione, per far sì che nessun nodo abbia un solo figlio, il nodo fittizio è unico e condiviso da tutti i nodi che lo necessitano.



Tale albero deve soddisfare una lista di **proprietà aggiuntive** per essere definito un albero RB :

1. *Ogni nodo, o è rosso, o è nero.*
2. *Ogni nodo fittizio è nero.*
3. *Un nodo rosso non può avere figli rossi.*
4. *Un nodo nero non ha vincoli sul colore dei propri figli.*
5. *Ogni singolo possibile cammino radice-foglia deve avere lo stesso numero di nodi neri.*
6. *La radice deve essere nera.*

Segue da queste proprietà che il numero di nodi rossi lungo un cammino, non può essere maggiore del numero di nodi neri lungo lo stesso cammino, inoltre ogni cammino radice-foglia sarà lungo al più il doppio di un altro cammino radice foglia.

B-Altezza

Definiamo come $bh(x)$ la **b-altezza** di un nodo x , ossia il **numero di nodi neri** sui cammini dal nodo x (non incluso) alle foglie sue discendenti, che secondo la proprietà numero 4, è uguale per tutti i cammini.

Da qui segue il seguente **lemma** :



Il sottoalbero radicato in un qualsiasi nodo x contiene almeno $2^{bh(x)} - 1$ nodi interni.

Inoltre, vale il seguente **teorema** :



Un RB-albero con n nodi interni ha altezza h tale che $h \leq 2 \cdot \log_2(n + 1)$

Operazioni su alberi RB

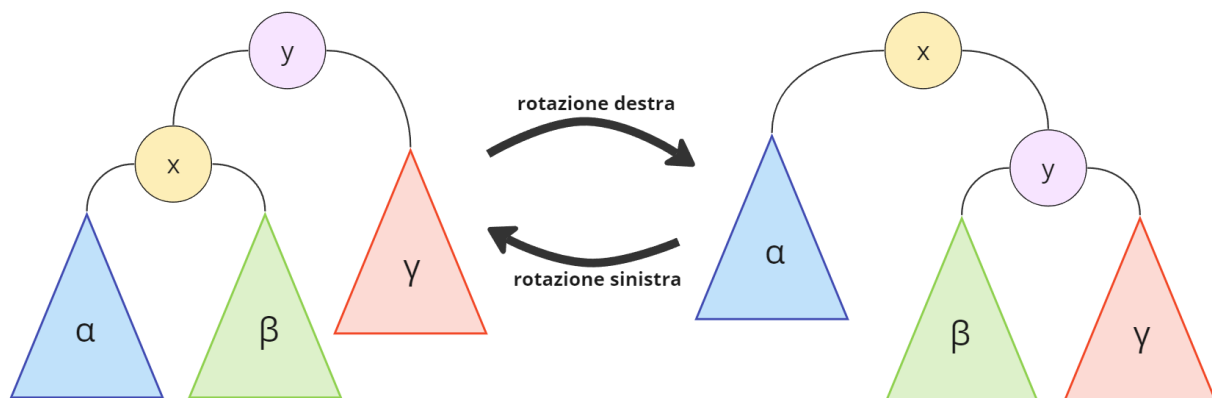
Il teorema precedente garantisce che tutte le operazioni di **ricerca** di una chiave, ricerca del massimo o del minimo, e ricerca del predecessore/successore abbiano costo logaritmico.

L'inserimento risulta più complesso, in quanto una volta inserita una chiave potrebbe essere necessario riaggiustare l'albero per mantenere le proprietà, cambiando i colori assegnati ai nodi o la struttura dei puntatori.

Introduciamo le **rotazioni** :

Le rotazioni permettono di **ripristinare** in tempo $O(\log_2(n))$ le proprietà dell'albero RB dopo un inserimento o cancellazione. Possono essere destre o sinistre, e sono operazioni **locali** che non modificano l'ordinamento della visita in-ordine.

L'operazione consiste nello scambiare due nodi x, y che sono collegati da una relazione padre figlio, essi vengono scambiati, modificando anche le relazioni fra i nodi ad essi collegati.



Nella rotazione sinistra si ha il nodo radice x , avente a destra un figlio y , e a sinistra un sottoalbero α . il nodo y ha come figlio destro un sottoalbero γ , e come figlio sinistro un sottoalbero β .

Eseguita la rotazione, y , che si ricordi essere di chiave maggiore di x , diventerà la radice, ed x diventerà suo figlio sinistro. l'albero sinistro figlio di x resterà α , mentre il precedente figlio sinistro di y , ossia β , diventerà figlio destro di x . il figlio destro di y resterà lo stesso, ossia γ .

La rotazione destra è analoga.

```
def RBA_rotaz_sinistra (p,a): #a=punt. al nodo su cui si ruota
    b = a->right
    a->right = b->left
    if a->right != None:
        a->right->parent = a
    b->left = a
    b->parent = a->parent
    if a->parent==None:
        p = b
    else:
        if a==a->parent->left:
            a->parent->left = b
        else:
            a->parent->right = b
    a->parent = b
    return p
```

Ha costo computazionale $\theta(1)$.

Ora che abbiamo definito cos'è una relazione, possiamo passare
all'inserimento.

Si inserisce un nodo in **modo analogo ad un normale ABR**, attribuendo sempre il colore rosso al nuovo nodo inserito.

Essendo il nuovo nodo inserito di colore rosso, non viene infranta la regola numero 5 :

5 - *Ogni singolo possibile cammino radice-foglia deve avere lo stesso numero di nodi neri.*

Le regole che potrebbero essere infrante sono :

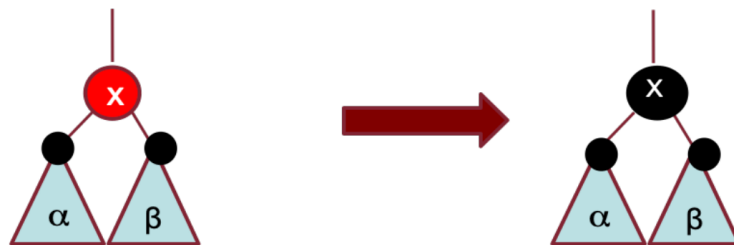
3 - *Un nodo rosso non può avere figli rossi.*

6 - *La radice deve essere nera.*

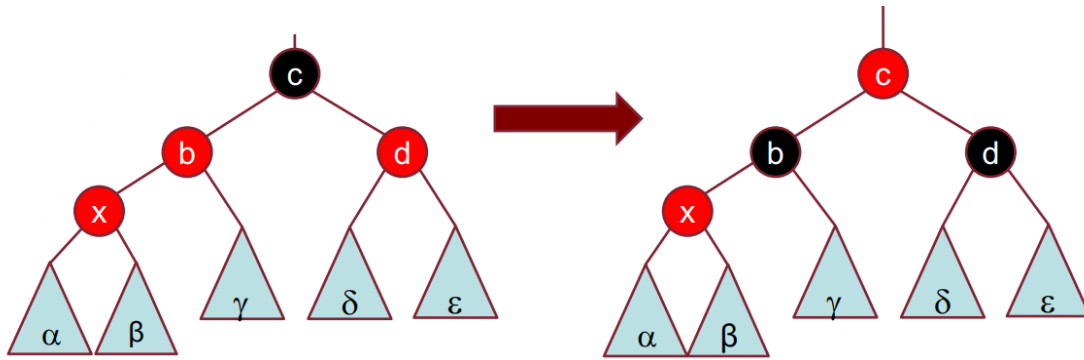
Se la 6 può essere risolta facilmente, la numero 3 risulta problematica.

Quando il nodo inserito (che si ricordi essere rosso) ha come padre un nodo rosso, occorre eseguire il **passo di aggiustamento**, che ricade in 4 possibili casistiche.

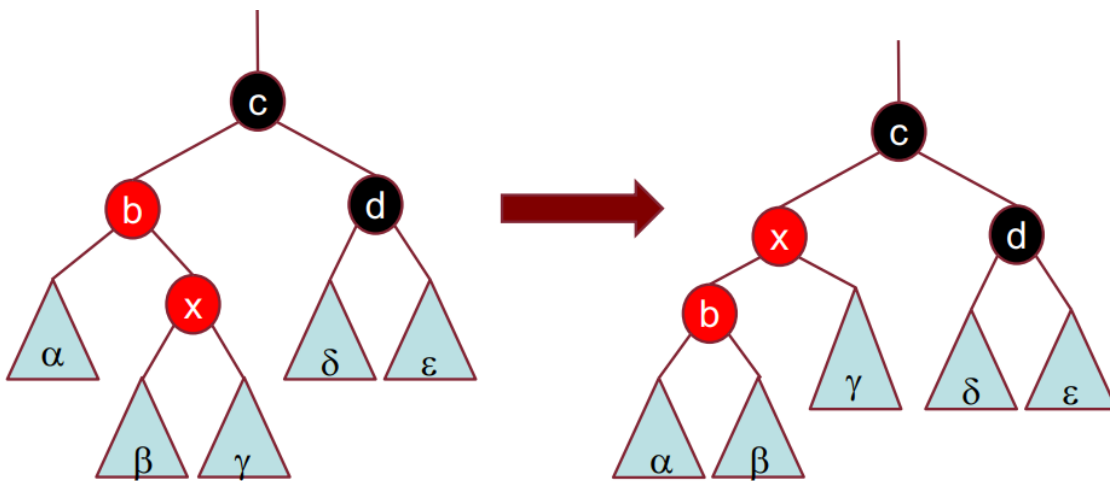
- **Caso 0** - Il nodo inserito x è una radice.
Semplicemente basta cambiare il colore del nodo inserito da rosso a nero.



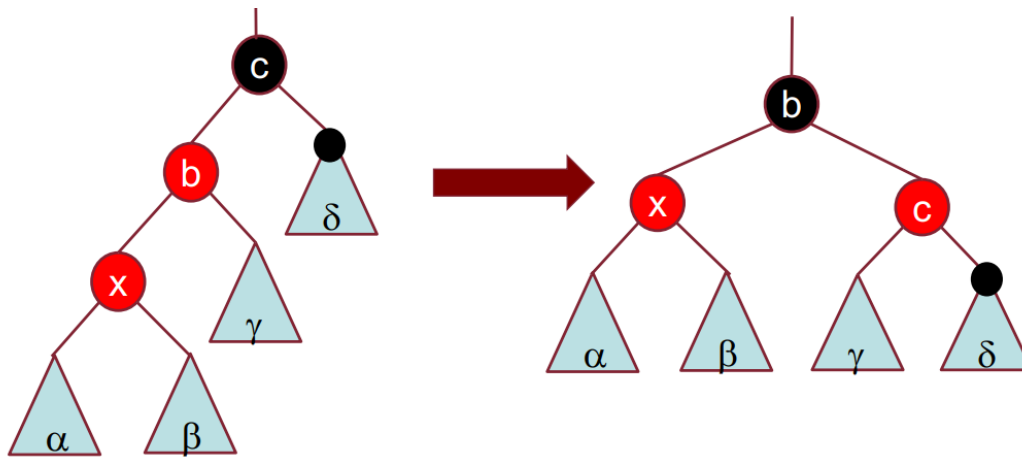
- **Caso 1** - Il nodo inserito x è figlio di un nodo rosso b , ed il fratello di b è rosso a sua volta.
Quindi sia il padre che lo zio di x sono rossi.
Basta cambiare il colore del padre e dello zio di x in nero, e far diventare rosso il nonno di x . Adesso potrebbe essersi causata una violazione sul nonno di x , e si farà su di esso il controllo.



- **Caso 2** - Il nodo inserito x è figlio destro di un nodo rosso b , ed il fratello di b è nero. Semplicemente si esegue una rotazione a sinistra sul padre di x . Questa operazione conduce sempre al caso 3, in quanto essendo sia x che suo padre rossi, scambiandosi rimarrà sempre una relazione padre figlio nella quale entrambi i nodi sono rossi.



- **Caso 3** - Il nodo inserito x è figlio sinistro di un nodo rosso b , ed il fratello di b è nero. A questo punto si effettua una rotazione a destra sul padre di b , e si invertono i colori di b e di suo padre.



È possibile durante un inserimento rientrare ripetutamente nel caso 1 (salendo sempre più in alto), ed eventualmente nel caso 2 che porterà sempre al caso 3.

Essendo che **ogni violazione può portare ad una violazione** più in alto nell'albero, ed essendo una violazione risolta sempre in tempo costante, l'inserimento di un RB costa $O(\log_2(n))$.

Il procedimento è

analogo per la cancellazione.