Introduzione agli Algoritmi

Simone Lidonnici

16 aprile 2024

Indice

1

Notazione asintotica

1.1 Tipi di notazione asintotica

La notazione asintotica serve per valutare l'efficienza di un algoritmo con una formula matematica e permette di confrontare il tasso di crescita di una funzione rispetto ad un'altra. In informatica si usa per stimare quanto aumenta il tempo di un algoritmo al crescere della grandezza dell'input. Ci sono tre tipi di notazioni asintotiche:

- **O grande**
- Ω (**Omega**)
- Θ (**Theta**)

1.1.1 O grande

Definizione di O grande

Date due funzioni $f(n), g(n) \geq 0$ si dice che $f(n)$ è in $O(g(n))$ se:

$$\exists c, n_0 | f(n) \leq c \cdot g(n) \ \forall n \geq n_0$$

Esempio:
$f(n) = 3n + 3$
$f(n) = O(n^2)$ perché con $c = 6 \implies cn^2 \geq 3n + 3 \ \forall n \geq 1$

1.1.2 Omega

Definizione di Ω

Date due funzioni $f(n), g(n) \geq 0$ si dice che $f(n)$ è in Ω(g(n)) se:

$$\exists c, n_0 | f(n) \geq c \cdot g(n) \ \forall n \geq n_0$$

Esempio:
$f(n) = 2n^2 + 3$
$f(n) = \Omega(n)$ perché $f(n) \geq n \ \forall n \geq 1$

1.1.3 Theta

Definizione di Θ

Date due funzioni $f(n), g(n) \geq 0$ si dice che $f(n)$ è in $\Theta(g(n))$ se:

$$\exists c_1, c_2, n | c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n) \ \forall n \geq n_0$$

Esempio:
$f(n) = \log_a n = \Theta(\log_b n) \ \forall a, b > 0$
Dimostrazione:
$\log_a n = \log_b n \cdot \log_a b = \log_b n \cdot c$

1.2 Algebra della notazione asintotica

1.2.1 Gerarchia

$c = O(\log(n))$
$log(n) = O(\sqrt{n})$
$\sqrt{n} = O(n^k)$
$n^k = O(a^n)$
$a^n = O(n!)$
$n! = O(n^n)$
Data una funzione $f(n)$, esistono infinite funzioni $g(n)$ per cui $f(n) = O(g(n))$ e infinite funzioni $h(n)$ per cui $f(n) = \Omega(h(n))$
Possiamo comparare facilmente due funzioni usando i limiti:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = \begin{cases} k > 0 \implies f(n) = \Theta(g(n)) \\ +\infty \implies f(n) = \Omega(g(n)) \\ 0 \implies f(n) = O(g(n)) \end{cases}$$

Regola sulle costanti moltiplicative

Per ogni $k > 0$ se $f(n) = O(g(n))$ allora anche $k \cdot f(n) = O(g(n))$
Questa regola vale solo se k non è all'esponente.
Dimostrazione:
$\exists c, n_0 | f(n) \leq c \cdot g(n) \ \forall n \geq n_0 \implies k \cdot f(n) \leq k \cdot c \cdot g(n) \implies k \cdot f(n) = O(g(n))$

Regola sulla commutatività con la somma

Se $f(n) = O(g(n)) \land d(n) = O(h(n)) \implies f(n) + d(n) = O(g(n) + h(n)) = O(\max(g(n), h(n)))$
Dimostrazione:
$f(n) = O(g(n)) \implies \exists c_1, n_1 | f(n) \leq c_1 \cdot g(n)$
$d(n) = O(h(n)) \implies \exists c_2, n_2 — d(n) \leq c_2 \cdot h(n)$
$f(n) + d(n) \leq c_1 \cdot g(n) + c_2 \cdot h(n) \implies c = \max(c_1, c_2) \implies f(n) + d(n) \leq c \cdot g(n) + c \cdot h(n) \implies f(n) + d(n) = O(g(n) + h(n))$

Regola sulla commutatività col prodotto

Se $f(n) = O(g(n)) \land d(n) = O(h(n)) \implies f(n) \cdot d(n) = O(g(n) \cdot h(n))$
Dimostrazione:
$f(n) = O(g(n)) \implies \exists c_1, n_1 | f(n) \leq c_1 \cdot g(n)$
$d(n) = O(h(n)) \implies \exists c_2, n_2 — d(n) \leq c_2 \cdot h(n)$
$f(n) \cdot d(n) \leq c_1 \cdot g(n) \cdot c_2 \cdot h(n) \implies c = \max(c_1, c_2) \implies f(n) \cdot d(n) \leq$
$c \cdot g(n) \cdot c \cdot h(n) \implies f(n) \cdot d(n) = O(g(n) \cdot h(n))$

1.2.2 Sommatorie notevoli

Questa è una lista contenente la maggior parte delle sommatorie notevoli utili per questo cosro:

$$\sum_{i=0}^{n} i = \theta(n^2) = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} i^c = \theta(n^{c+1})$$

$$\sum_{i=0}^{n} 2^i = \theta(2^n)$$

$$\sum_{i=0}^{n} c^i = \begin{cases} \frac{c^{n+1}-1}{c-1} & c > 1 \\ 1 & c \leq 1 \end{cases}$$

$$\sum_{i=0}^{n} i2^i = \theta(n2^n)$$

$$\sum_{i=0}^{n} ic^i = \theta(nc^n)$$

$$\sum_{i=0}^{n} \log i = \theta(\log n)$$

$$\sum_{i=0}^{n} \log^c i = \theta(n \log^c n)$$

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

2

Costo computazionale

Possiamo calcolare il costo computazionale di un algoritmo usando il criterio del costo uniforme. Il costo computazionale è una funzione monotona crescente al crescere della dimensione dell'input. Visto che viene utilizzata la notazione asintotica il costo computazionale è calcolabile solo asintoticamente(cioè con input abbastanza grandi). Il costo di un algoritmo è la somma di tutte le istruzioni che lo compongono. Alcuni algoritmi potrebbero avere tempi di esecuzioni diversi in base all'input, in questi casi devo calcolare a parità di dimensione di input il caso peggiore e il caso migliore. Se voglio scrivere il tempo di esecuzione a prescindere dall'input devo considerare il caso peggiore.

2.1 Costo delle istruzioni

Ogni tipo di istruzione ha un costo computazionale diverso in base a come viene eseguita.

2.1.1 Istruzioni elementari

Le istruzioni elementari hanno costo $\Theta(1)$ e sono:

- Operazioni aritmetiche
- Lettura di un valore da una variabile
- Assegnazione di un valore
- Condizione logica su un numero costante di operandi
- Stampa di un valore

Nel caso di codici con `if` il costo è il costo della verifica della condizione sommato al amsimo tra il caso vero e il caso falso.

2.1.2 Istruzioni iterative

Nel caso di cicli `for` o `while` il costo è pari alla somma del costo di ciascuna iterazione (contando anche la verifica della condizione). Ci sono due casi:

- Se il costo di ogni iterazione è uguale allora basta moltiplicare il costo di una iterazione per il numero di cicli
- Se il costo delle iterazioni è diverso allora bisogna sommare il costo delle iterazioni

Esempi:

Algoritmo: Massimo di un vettore di n numeri

```
def Trova_max(A):
    n=len(A)
    max=A[0]
    for i in range(1,n) :  // n − 1 iterazioni
        if A[i]>max :
            max=A[i]
    return max
```

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

Algoritmo: Somma dei primi n numeri interi

```
def Somma(n):
    somma=0
    for i in range(1,n+1) :  // n iterazioni
        somma+=i
    return somma
```

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

Algoritmo: Valutazione di un polinomio

```
def Calcolo_poliniomio(A,c):
    somma=A[0]
    for i in range(len(A)) : // n iterazioni
        potenza=1
        for j in range(i) :  // i iterazioni
            potenza=c*potenza
        somma+=A[i]*potenza
    return somma
```

$$T(n) = n\Theta(i) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

2.1.3 Calcolo del costo computazionale

Nel caso in cui il caso peggiore ed il caso migliore hanno la stessa formula allora il costo computazionale è tale formula e si utilizza la notazione Θ. Nel caso invece le due formule sono diverse allora il costo computazionale si può scrivere con la notazione O della formula che identifica il caso peggiore oppure posso calcolare il costo medio.

2.2 Tempi di esecuzione

Il tempo di esecuzione è calcolato moltiplicando il numero n di dati per il costo dell'algoritmo e dividendolo per il numero di operazioni al secondo.

$$T(n) = \frac{\text{algoritmo}(n)}{\text{operazioni}/s}$$

Esempio:
10^9 opeazioni al secondo e $n = 10^6$:

- $O(n) = \frac{10^6}{10^9} = 10^{-3}s$
- $O(n \log n) = \frac{10^6 \cdot \log(10^6)}{10^9} = 2 \cdot 10^{-2}s$
- $O(n^2) = \frac{(10^6)^2}{10^9} = 10^3 s$

3

Algoritmi di Ricerca

Uno dei problemi principali dell'informatica è la ricerca di un elemento in un insieme di dati e per fare ciò si utilizza un algoritmo di ricerca.
L'algoritmo contiene:

- **Input**: un array di n elementi e un valore da trovare
- **Output**: un indice i tale che A[i]=valore o un valore None se il valore non è presente

3.1 Ricerca sequenziale

Questo algoritmo di ricerca scorre tutti i valori dell'array uno ad uno e li confronta con il valore, fermandosi quando lo si trova:

Algoritmo: Ricerca sequenziale

```
def cerca_v(A,v):
    i=0
    while i<len(A) :
        if A[i]==v :
            return i
    return None
```

Caso peggiore: $T(n) = \Theta(n)$
Caso migliore: $T(n) = \Theta(1)$

Costo medio:
Visto che il caso migliore e peggiore hanno due formule diverse per calcolare il costo medio usiamo l'ipotesi che v si possa trovare in tutte le posizioni con la stessa possibilità, quindi la probabilità che v si trovi nella posizione i è $\frac{1}{n}$

Costo medio $= \frac{1}{n}\sum_{i=0}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$
Possiamo anche usare un'altra formula:
Costo medio$=\sum_{i=0}^{n} i \cdot \frac{\text{numero di permutazioni in cui v è nel punto i}}{\text{numero di permutazioni totali}} = \frac{n(n+1)}{2}\frac{(n-1)!}{n!} = \frac{n+1}{2}$

3.2 Ricerca binaria

Questo algoritmo controlla l'elemento centrale dell'array (l'array deve essere ordinato) e lo confronta con il valore, se è più grande controlla la metà superiore, se è minore controlla la metà inferiore, ripetendo il procedimento:

Algoritmo: Ricerca binaria

```
def ricerca_binaria(A,v):
    a=0
    b=len(A)-1
    m=(a+b)/2
    while A[m]!=v :
        if A[m]>v :
            b=m-1
        else
            a=m-1
        if a>b :
            return -1
        m=(a+b)/2
    return m
```

Caso peggiore: $T(n) = \Theta(\log n)$
Caso migliore: $T(n) = \Theta(1)$

Costo medio:
Visto che il caso migliore e peggiore hanno due formule diverse per calcolare il costo medio usiamo l'ipotesi che v si possa trovare in tutte le posizioni con la stessa possibilità.
Dopo i iterazioni sono state controllate 2^{i-1} posizioni, quindi la probabilità che il valore sia in una di queste è $\frac{2^{i-1}}{n}$.

$$\text{Costo medio}=\frac{1}{n}\sum_{i=1}^{\log n} i \cdot 2^{i-1} \implies \frac{1}{n}(\log n - 1)2^{\log n} + 1 = \log n + \frac{1}{n} - 1$$

4

Ricorsione

Un algoritmo è ricorsivo quando:

- è espresso in termini di se stesso
- la soluzione del problema è data dalla risoluzione di sotto-problemi di grandezza minore combinati insieme
- la successione dei sotto-problemi deve infine convergere ad un caso base che termina la ricorsione

Esempi:

Algoritmo: Fattoriale

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

Algoritmo: Ricerca binaria ricorsiva

```
def ricerca_bin(A,v,i_min=0,i_max=len(A)):
    if i_min>i_max :
        return None
    m=(i_min+i_max)/2
    if A[m]==v :
        return m
    elif A[m]>v :
        return ricerca_binaria(A,v,i_min,m-1)
    else
        return ricerca_binaria(A,v,m+1,i_max)
```

4.1 Iterazione vs Ricorsione

In diversi casi potrebbe essere migliore un tipo di algoritmo invece di un altro:

- **Ricorsivo**: se la formulazione della soluzione è aderente al problema stesso e quella iterativa è più complicata
- **Iterativo**: se la soluzione iterativa è evidente o se l'efficienza è importante

Questa distinzione si ha perché ogni funzione ha bisogno di una certa quantità di memoria e le funzioni ricorsive ne richiedono di più.
Esempio:

Algoritmo: Fibonacci iterativo

```
def fib(n):
    if n≤1 :
        return n
    fib1=1
    fib2=0
    for i in range(2,n+1) :
        fib2=fib1
        fib1+=fib2
    return fib1
```

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

Algoritmo: Fibonacci ricorsivo

```
def fib(n):
    if n≤1 :
        return n
    return fib(n-1)+fib(n-2)
```

Il numero di chiamate della funzione cresce molto velocemente perché molti calcoli vengono ripetuti più di una volta e quando si arriva al caso base ci sono una catena di chiamate ancora aperte. Per risolvere questo problema di dimensione n bisogna risolvere 2 sotto-problemi di dimensione $n-1$ e $n-2$.
$T(n) = T(n-1) + T(n-2) + \Theta(n)$
il costo è quindi esponenziale ma per calcolarlo bisogna usare le equazioni di ricorrenza.

5

Equazioni di ricorrenza

5.1 Scrittura di un'equazione di ricorrenza

Per calcolare il costo computazionale di un algoritmo ricorsivo ci si ritrova a dover risolvere una funzione ricorsiva, questa funzione si chiama equazione di ricorrenza.
Esempio:

Algoritmo: Fattoriale ricorsivo

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

Il costo computazionale di questo algoritmo è:

$$T = \begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(0) = \Theta(1) \end{cases}$$

La parte generale deve essere composta dalla somma del costo computazionale non ricorsivo e dalla parte ricorsiva, ci deve inoltre essere almeno un caso base.
Per calcolare il costo computazionale di una equazione di ricorrenza ci sono 4 metodi:

- Metodo iterativo
- Metodo dell'albero
- Metodo di sostituzione
- Metodo principale

5.2 Metodo iterativo

Nel metodo iterativo si sviluppa l'equazione di ricorrenza per far si che sia una somma di termini dipendenti dal caso generico e dal caso base.
Esempio:

$$T = \begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(0) = \Theta(1) \end{cases}$$

Se sviluppiamo $T(n)$ come somma dei suoi sotto-termini:
$T(n) = T(n-1) + \Theta(1) = T(n-2) + 2 \cdot \Theta(1) = T(n-k) + k \cdot \Theta(1)$

La ricorsione continua finché $n-k=1 \implies k=n-1$ e l'equazione quindi diventerà:
$T(n)=T(n-k)+k\cdot\Theta(1)=T(n-n+1)+(n-1)\cdot\Theta(1)=T(1)+(n-1)\cdot\Theta(1)=\Theta(n)$

5.2.1 Caso particolare: sequenza di Fibonacci

Equazione di ricorrenza dell'n-esimo numero di Fibonacci:

$$T=\begin{cases}T(n)=T(n-1)+T(n-2)+\Theta(1)\\T(1)=\Theta(1)\end{cases}$$

Se sviluppiamo l'equazione:
$T(n)=T(n-1)+T(n-2)+\Theta(1)=T(n-2)+2T(n-3)+T(n-3)+3\Theta(1)=...$
Visto che non si può generalizzare il problema, cerchiamo di calcolare il costo O e il costo Ω

Costo O:
$T(n)=T(n-1)+T(n-2)+\Theta(1)\leq T(n-1)+T(n-1)+\Theta(1)=T_1(n)$
Quindi $T_1(n)$:

$$T_1=\begin{cases}T_1(n)=2T_1(n-1)+\Theta(1)\\T_1(1)=\Theta(1)\end{cases}$$

Sviluppando $T_1(n)$:
$T_1(n)=2T_1(n-1)+\Theta(1)=2[2T_1(n-2)+\Theta(1)]+\Theta(1)$
Caso base $=n-k=1 \implies k=n-1$
Generalizzando:

$$T_1(n)=2^kT_1(n-k)+\sum_{i=0}^{k-1}2^i\Theta(1)=2^{n-1}T_1(1)+\sum_{i=0}^{n-2}2^i\Theta(1)=\Theta(2^n)+(2^{n-1}-1)\Theta(1)=\Theta(2^n)$$

Quindi visto che $T(n)\leq T_1(n)=\Theta(2^n) \implies T(n)=O(2^n)$

Costo Ω:
$T(n)=T(n-1)+T(n-2)+\Theta(1)\geq T(n-2)+T(n-2)+\Theta(1)=T_2(n)$
Quindi $T_2(n)$:

$$T_2=\begin{cases}T_2(n)=2T_2(n-2)+\Theta(1)\\T_2(1)=\Theta(1)\end{cases}$$

Sviluppando $T_2(n)$:
$T_2(n)=2T_2(n-2)+\Theta(1)=2[2T_2(n-4)+\Theta(1)]+\Theta(1)$
Caso base $=n-2k=1 \implies k=\frac{n}{2}$
Generalizzando:

$$T_2(n)=2^kT_2(n-2k)+\sum_{i=0}^{k-1}2^i\Theta(1)=2^{\frac{n}{2}}T_2(1)+\sum_{i=0}^{\frac{n}{2}-1}2^i\Theta(1)=\Theta(2^{\frac{n}{2}})+(2^{\frac{n}{2}}-1)\Theta(1)=\Theta(2^{\frac{n}{2}})=\Theta(\sqrt{2^n})$$

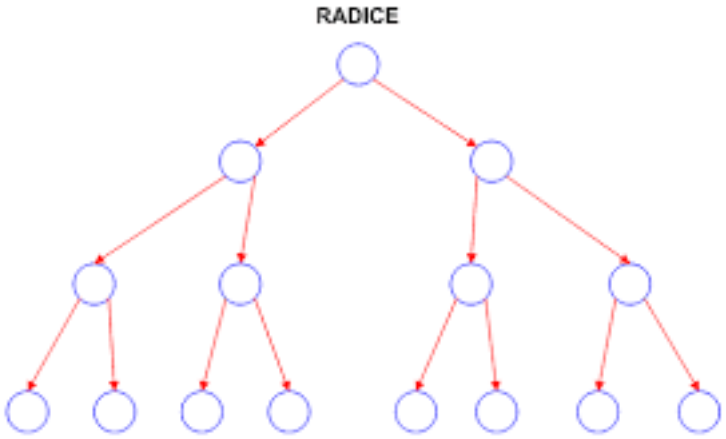
Quindi visto che $T(n)\geq T_2(n)=\Theta(\sqrt{2^n}) \implies T(n)=\Omega(\sqrt{2^n})$

Costo finale:

$$\sqrt{2^n}\leq T(n)\leq 2^n$$

5.3 Metodo dell'albero

Nel metodo dell'albero si rappresenta graficamente lo sviluppo del costo computazionale attraverso un albero binario per calcolarlo più facilmente.
Un albero binario completo di altezza h è un albero in cui tutti i nodi hanno due figli tranne quelli nel livello h che non hanno figli (i nodi nell'ultimo livello si chiamano foglie).



Il numero di nodi nell'ultimo livello (foglie) è 2^h

Il numero di nodi all'interno dell'albero è $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

Esempio:

$$T = \begin{cases} T(n) = 2T(\frac{n}{2}) + \Theta(n^2) \\ T(1) = \Theta(1) \end{cases}$$

Il costo di ogni livello é:

Livello	n° nodi	Costo per nodo	Contributo per livello
0	2^0	$\Theta(n^2)$	$2^0 \cdot \Theta(n^2)$
1	2^1	$\Theta((\frac{n}{2})^2)$	$2^1 \cdot \Theta((\frac{n}{2})^2)$
2	2^2	$\Theta((\frac{n}{4})^2)$	$2^2 \cdot \Theta((\frac{n}{4})^2)$
...	...	...	...
$h = \log n$	$2^{\log n}$	$\Theta((\frac{n}{2^{\log n}})^2)$	$2^{\log n} \cdot \Theta((\frac{n}{2^{\log n}})^2)$

Quindi il costo totale dell'albero e quindi dell'equazione è:

$$T(n) = \sum_{k=0}^{\log n} 2^k \Theta((\frac{n}{2^k})^2) = \sum_{k=0}^{\log n} \frac{2^k}{2^{2k}} \Theta(n^2) = \Theta(n^2) \sum_{k=0}^{\log n} \frac{1}{2^k} = \Theta(n^2)$$

5.4 Metodo di sostituzione

Nel metodo di sostituzione si ipotizza una soluzione e poi si dimostra per induzione.
Esempio:

$$T = \begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

Ipotizziamo la soluzione:
$\begin{cases} T(n) = T(n-1) + c \cdot \Theta(1) \\ T(1) = d \end{cases}$
Con c e d valori casuali per cui l'equazione funziona.
Ipotizziamo che la soluzione sia $\Theta(n)$ quindi dobbiamo dimostrare che sia $O(n)$ e $\Omega(n)$.

Dimostrare O:
Ipotizziamo $T(n) = O(n) \implies T(n) \leq k \cdot n$ dove k è una costante da determinare.

1. Caso base $(n = 1)$: $\begin{cases} T(1) = d \\ T(1) \leq k \end{cases} \implies k \geq d$
2. Ipotesi induttiva: $T(m) \leq km \; \forall m < n$
3. Dimostrazione:
   $T(n) = T(n-1) + c \cdot \Theta(1) \implies T(n-1) \leq k(n-1) \implies T(n) \leq k(n-1) + c \implies$
   $T(n) \leq kn - k + c \implies kn - k + c \leq kn \implies c \leq k$
   Quindi:
   $T(n) = O(n) \; \forall k \geq c$

Dimostrare Ω:
Ipotizziamo $T(n) = \Omega(n) \implies T(n) \geq hn$ dove h è una costante da determinare.

1. Caso base $(n = 1)$: $\begin{cases} T(1) = d \\ T(1) \geq h \end{cases} \implies d \geq h$
2. Ipotesi induttiva: $T(m) \geq hm \; \forall m < n$
3. Dimostrazione:
   $T(n) = T(n-1) + c \cdot \Theta(1) \implies T(n-1) \geq h(n-1) \implies T(n) \geq h(n-1) + c \implies$
   $T(n) \geq hn - h + c \implies hn - h + c \geq hn \implies c \geq h$
   Quindi:
   $T(n) = \Omega(n) \; \forall h \leq c$

5.5 Metodo principale

Nel metodo principale è un teorema che permette di risolvere le equazioni di ricorrenza.
Funziona solo se $T(n) = aT(\frac{n}{b}) + f(n)$ e $T(1) = \Theta(1)$

Teorema principale

Dati $a \geq 1$ e $b > 1$ una funzione di ricorrenza scritta come:

$$T = \begin{cases} T(n) = a \cdot T(\frac{n}{b}) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

Ha diverse formule risolutive in base a $f(n)$:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{se } f(n) = O(n^{\log_b(a)-\epsilon}) \\ \Theta(n^{\log_b a} \cdot \log n) & \text{se } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{se } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ e } a \cdot f(\frac{n}{b}) \leq c \cdot f(n) \end{cases}$$

Con $\epsilon > 0$ e $c < 1$.

Esempio caso 1:
$T(n) = 9T(\frac{n}{3}) + \Theta(n)$
$n^{\log_b a} = n^{\log_3 9} = n^2$
$f(n) = n^{\log_3 9-\epsilon}$ con $\epsilon = 1 \implies T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Esempio caso 2:
$T(n) = T(\frac{2}{3}n) + \Theta(1)$
$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$
$f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(\log n)$

Esempio caso 3:
$T(n) = 3T(\frac{n}{4}) + \Theta(n \log n)$
$n^{\log_b a} = n^{\log_4 3} = n^{0,7}$
$f(n) = \Omega(n^{n^{\log_4 3+\epsilon}})$ con $\epsilon = 0,2$ e $a \cdot f(\frac{n}{b}) = 3\Theta(\frac{n}{4} \log \frac{n}{4}) = \frac{3}{4}(n \log n - n \log 4) \leq c \cdot n \log n$ con $c = \frac{3}{4} \implies T(n) = \Theta(n \log n)$

6

Algoritmi di Sorting

Un **algoritmo di sorting** è un algoritmo che presa un array di elementi non ordinato lo ritorna in modo ordinato.
Vengono divisi in semplici:

- **Insertion sort**
- **Selection sort**
- **Bubble sort**

e complessi:

- **Merge sort**
- **Heap sort**
- **Quick sort**

6.1 Insertion sort

L'insertion sort estrae un elemento di indice i dall'array spostando verso destra tutti gli elementi maggiori per inserirlo nel posto giusto. Questo processo viene rpetuto per ogni elemento.

Algoritmo: Insertion sort

```
def insertion_sort(A):
    for j in range(1,len(A)) :
        x=A[j]
        i=j-1
        while i≥0 and A[i]>x :
            A[i+1]=A[i]
            i-=1
        A[i+1]=x
    return A
```

Costo computazionale:

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

Per ogni ciclo $1 < t_j < j$ quindi:

- Caso migliore: $T(n) = (n-1)\Theta(1) = \Theta(n)$
- Caso peggiore: $T(n) = \sum_{j=1}^{n-1}(\Theta(1) + \Theta(j)) = \Theta(n^2)$

6.2 Selection sort

Il selection sort cerca il minimo all'interno dell'array, mettendolo in prima posizione per poi cercare il nuovo minore escludendo il primo per metterlo in seconda posizione e così via per ogni elemento.

Algoritmo: Selection sort

```
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]
    return A
```

Costo computazionale:

$$T(n) = \sum_{i=0}^{n-2}(\Theta(1) + (n-i)\Theta(1) + \Theta(1)) + \Theta(1) = \Theta(n^2)$$

6.3 Bubble sort

Il bubble sort confronta da destra verso sinistra ogni coppia di valori, scambiandoli se non sono ordinati, questo processo viene eseguito tante volte quanti sono gli elementi dell'array.

Algoritmo: Bubble sort

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

Costo computazionale:

$$T(n) = \sum_{i=0}^{n-1}(\Theta(1) + (n-i)\Theta(1) + \Theta(1)) + \Theta(1) = \Theta(n^2)$$

6.4 Alberi di decisioni

Per stabilire un limite al costo computazionale sotto il quale non si può andare si usa uno strumento chiamato albero di decisione. Per algoritmi di ordinamento basati su confronti è un albero binario che rappresenta tutti i possibili confronti: i nodi interni rappresentano un confronto e i figli i possibili esiti del confronto.
Con un numero generico n di elementi, il numero delle foglie deve avere tutte le permutazioni possibili quindi $n!$.
Il numero massimo di foglie di un albero binario è 2^h quindi:

$$2^h \geq n! \implies h \geq \log(n!) = \Theta(n \log n)$$

Con questo possiamo dire che il miglior algoritmo di sorting basato su confronti avrà costo: $\Omega(n \log n)$

6.5 Merge sort

Il merge sort uilizza un algoritmo ricorsivo secondo una tecnica chiamata *divide et impera*, che consiste nel dividere il problema generale in sotto-problemi più piccoli che vengono risolti ricorsivamente e poi tutte le soluzioni vengono combinate.

Algoritmo: Merge sort

```
def merge_sort(A,id_inizio,id_fine):
    if id_inizio<id_fine :
        id_medio=(id_inizio-id_fine)/2
        merge_sort(A,id_inizio,id_medio)
        merge_sort(A, id_medio+1,id_fine)
        Fondi(A,id_inizio,id_medio,id_fine)
    return A
```

Costo computazionale:

$$T(n) = \Theta(1) + 2T(\frac{n}{2}) + S(n)$$

$S(n)$ è il costo di **Fondi** che è una funzione che sfrutta il fatto che le due sottosequenze sono ordinate, quindi il minimo sarà il più piccolo tra i due minimi e si continua eliminando di volta in volta i minimi usati.

Algoritmo: Fondi

```
def Fondi(A,id_inizio,id_medio,id_fine):
    i,j=id_medio,id_medio+1
    B=[]
    while i≤id_medio and j≤id_fine :
        if A[i]≤A[j] :
            B.append(A[i])
            i+=1
        else
            B.append(A[j])
            j+=1
    while i≤id_medio :
        B.append(A[i])
        i+=1
    while j≤id_fine :
        B.append(A[j])
        j+=1
    for i in range(len(B)) :
        A[id_inizio+i]=B[i]
    return B
```

Costo computazionale:
$S(n) = \Theta(n)$
Quindi il costo computazionale del merge sort in totale è un'equazione di ricorrenza:

$$T = \begin{cases} T(n) = 2T(\frac{n}{2}) + \Theta(n) \\ T(1) = \Theta(1) \end{cases} \quad = \Theta(n \log n)$$

6.6 Quick sort

Il quick sort è un algoritmo che unisce i vantaggi di ordinare senza un array di appoggio e anche quelli del merge sort.
Nella sequenza si sceglie un pivot e viene divisa la sequenza in due gruppi, uno con tutti gli elementi maggiori del pivot e uno con tutti quelli minori o uguali.
Nonostante abbia un tempo massimo di $\Theta(n^2)$ nella media il tempo di esecuzione è $\Theta(n \log n)$.

Algoritmo: Quick sort

```
def quick_sort(A,id_inizio,id_fine):
    if id_inizio<id_fine :
        id_medio=Partizione(A,id_inizio,id_fine)
        quick_sort(A,idi_inizio,id_medio)
        quick_sort(A,id_medio+1,id_fine)
    return A
```

Costo computazionale:
$T(n) = T(k) + T(n-k) + S(n)$
In cui $S(n)$ è il costo di **Partiziona**:

Algoritmo: Partiziona

```
def Partiziona(A,id_inizio,id_fine):
    pivot=A[id_fine]
    i=id_inizio-1
    for j in range(id_inizio,id_fine) :
        if A[j]≤pivot :
            i+=1
            A[i],A[j]=A[j],A[i]
    A[i+1],A[id_fine]=A[id_fine],A[i+1]
    return i+1
```

Costo computazionale:
$S(n) = \Theta(n)$
Quindi il costo toale del quick sort corrisponde all'equazione di ricorrenza:

$$T(n) = \begin{cases} T(k) + T(n-k) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$$

- Caso migliore: $k = \frac{n}{2} \implies T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$
- Caso peggiore: $k = 1 \implies T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
- Caso medio: $T(k) = \frac{1}{n-1}(\sum_{k=0}^{n-1} T(k) + T(n-k)) + \Theta(n) = \frac{2}{n-1} \sum_{k=1}^{n-1} T(k) + \Theta(n) = \Theta(n \log n)$

6.7 Heap sort

L'heap sort ordina il loco come il selection sort ma ha bisogno di una precisa struttura dati che deve essere mantenuta per far funzionare l'algoritmo. L'algoritmo preleva il massimo e mette l'ultimo valore dell'array al suo posto.

6.7.1 Struttura dati Heap

Un heap è un albero binario in cui tutti i livelli sono pieni tranne l'ultimo livello in cui i nodi sono tutti a sinistra.
Per memorizzare un heap si utilizza un array con elementi pari al numero di nodi e viene riempito con i valori di ogni livello da destra verso sinistra. Ogni elemento $A[i]$ ha il figlio sinistro all'elemento $A[2i+1]$ e il figlio destro all'elemento $A[2i+2]$. Il padre di un nodo si trova all'elemento $A[(i-1)/2]$ (si arrotonda per difetto nel caso di indici non interi).
L'heap ha delle proprità specifiche:

- Essendo un albero binario con tutti i livelli pieni l'altezza è $\log n$
- Ogni elemento è minore del proprio padre

6.7.2 Funzioni ausilirie dell'heap sort

L'algoritmo dell'heap sort utilizza due funzioni ausilarie:

- Funzione Heapify
- Funzione Buildheap

Heapify:
La funzione Heapify mantiene la proprietà di heap dell'albero(figli maggiori dei padri), partendo da un array in cui solo la radice può essere più piccola(perchè è stata scambiata con l'ultimo valore) dei figli mentre i sotto alberi sono corretti. Il risultato è un array in cui è di nuovo presente la proprietà di heap.

Algoritmo: Heapify

```
def Heapify(A):
    L=2*i+1// indice del figlio sinistro
    R=2*i+2// indice del figlio destro
    id_max=i
    if L<heap_size and A[L]>A[i] :
        id_max=L
    if R≤heap_size and A[R]>A[id_max] :
        id_max=R
    if id_max!=i :
        A[i],A[id_max]=A[id_max],A[i]
        Heapify(A,id_max,heap_size)
    return A
```

Costo computazionale:
$T(n) = T(n') + \Theta(1)$
In cui n' è il numero di nodi del sottoalbero con più nodi.
Caso peggiore:
$n' = \frac{2}{3}n \implies T(n) = T(\frac{2}{3}n) + \Theta(1) = \Theta(\log n)$

Buildheap:
La funzione buildheap prende un qualsiasi array disordinato e lo trasforma in un array con la proprietà di heap.

Algoritmo: Buildheap

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

Costo computazionale:
$T(n) = O(n)$

6.7.3 Algoritmo completo dell'heap sort

Trasforma un array di dimensione n in un heap tramite Buildheap, scambia la radice con il nodo all'indice n-1 e poi viene richiamato Heapify sull'array di n-1 elementi e così via fino alla fine di tutti gli n elementi.

Algoritmo: Heap sort

```
def heap_sort(A):
    Buildheap(A,len(A))
    for x in reversed(range(1,len(A))) :
        A[0],A[x]=A[x],A[0]
        Heapify(A,0,x)
    return A
```

Costo computazionale:
$T(n) = O(n) + n(O(\log n)) = O(n \log n)$

6.8 Algoritmi di ordinamento con costo lineare

Abbiamo visto che un algoritmo di ordinamento basato su confronti non può avere un costo computazionale minore di $\Omega(n \log n)$.
Esistono però algoritmi di ordinamento non basati su contronti che quindi possono avere costo $\Theta(n)$.

6.8.1 Counting sort

Il counting sort funziona seguendo l'ipotesi che ciascuno degli elementi ha un valore compreso in un intervallo $[0, k]$, quindi il costo computazionale è $\Theta(n+k)$ e se $k = O(n)$ allora l'algoritmo ha un costo computazionale $\Theta(n)$.
Il counting sort usa un array ausiliario di lunghezza $k + 1$ in cui viene scritto per ogni indice quante volte quel numero è compreso nell'array iniziale. Poi viene sovrascritto l'array iniziale scrivendo ogni indice quante volte è indicato nell'array ausiliario.

Algoritmo: Counting sort

```
def counting_sort(A):
    k=max(A)
    n=len(A)
    C=[0 for i in range(k+1)]
    for j in range(n) :
        C[A[j]]+=1
    j=0
    for i in range(k) :
        while C[i]>0 :
            A[j]=i
            j+=1
            C[i]-=i
    return A
```

Costo computazionale:

$$T(n) = \Theta(n) + \Theta(k) \sum_{i=0}^{k} C[i]\Theta(1) = \Theta(n+k)$$

Una variazione del counting sort è uno in cui viene fatta una seconda passata all'array ausiliario e ai valori degli indici vengono sommati i valori dell'indice prima in modo che ogni indice indichi l'ultimo posto dove va scritto il valore di quel determinato indice.

Algoritmo: Counting sort con dati satellite

```
def counting_sort(A):
    k=max(A)
    n=len(A)
    C=[0 for i in range(k+1)]
    B=[0 for i in range(n)]
    for j in range(n) :
        C[A[j]]+=1
    for i in range(1,k) :
        C[i]+=C[i-1]
    for j in range(n,-1,-1) :
        B[C[A[j]]]=A[j]
        C[A[j]]-=1
    return B
```

6.8.2 Bucket sort

Il bucket sort si basa sull'ipotesi che gli elementi siano distribuiti in modo uniforme nell'intervallo $[1, k]$.
Mette i valori dell'array in dei bucket che poi vengono sortati e poi copiati sull'array.

Algoritmo: Bucket sort

```
def bucket_sort(A,k,n):
    for i in range(1,n) :
        B.append(A[i])
    for i in range(1,n) :
        insertion_sort(B[i])
    for i in range(1,n) :
        B+=B[i]
    for i in range(n) :
        A[i]=B[i]
    return A
```

7

Strutture dati

Una struttura dati è composta da:

- Un modo sistematico di organizzare i dati
- Un insieme di operatori che permettono di manipolare i dati

Le strutture possono essere:

- Lineari o non lineari: se esiste una sequenzializzazione
- Statiche o dinamiche: se possono cambiare dimensione nel tempo

Una struttura dati serve per memorizzare un insieme dinamico, cioè in cui gli elementi possono cambiare in base agli algoritmi che li utilizzano.

7.1 Insiemi dinamici

Gli insiemi dinamici possono avere elementi complessi e contenere multipli dati elementari, per questo di solito contengono:

- Una chiave: che serve per distinguere gli elementi tra loro quando si manipola l'insieme (di solito fanno parte di un insieme ordinato come numeri o lettere)
- Dati satellite: che sono relativi ad un elemento ma non vengono usati dagli algoritmi

Le operazioni svolte su un insieme dinamico si dividono in due categorie:

- Operazioni di interrogazione
- Operazioni di manipolazione

Esempi di operazioni di interrogazione sono:

- **Search**: cerca un determinato elemento con chiave k
- **Min/Max**: cerca l'elemento con chiave massimo o minima
- **Predecessor/Successor**: cerca l'elemento che precede o succede una determinato elemento con chiave k

Esempi di operazioni di manipolazione sono:

- **Insert**: inserisce un elemento con chiave k
- **Delete**: elimina un elemento con chiave k

7.2 Array

Un array è una struttura dati statica, anche se in alcuni linguaggi sembra possibile variare la dimensione, ma solo perché viene usata una struttura dati dinamica che simula gli array.
Il costo computazionale delle operazioni in base a se l'array è ordinato o no:

Struttura	Search(A,k)	Min(A)/Max(A)	Predecessor(A,k) Successor(A,k)	Insert(A,k)	Delete(A,k)
Array non ordinato	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Array ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$

7.3 Liste puntate

Un puntatore è una variabile che ha come valore un indirizzo di memoria. Il nome di una variabile fa riferimento ad un valore in modo diretto mentre un puntatore in modo indiretto.
Ogni elemento della lista è composto da due campi:

- Key: contiene l'informazione (scritto come `p->key`)
- Next: contiene il puntatore al prossimo elemento della lista (scritto come `p->next`)

Le liste puntate hanno delle proprietà:

- L'accesso avviene ad un'estremità della lista attraverso un puntatore alla testa della lista
- È permesso solo un accesso sequenziale agli elementi (implica costo $\Theta(n)$ per qualsiasi accesso ad un elemento)

7.3.1 Search nelle liste puntate

Algoritmo: Ricerca di un elemento con chiave k in una lista puntata

Input:
- p: puntatore alla testa della lista
- k: chiave dell'elemento da cercare

```
def search(p,k):
    p_corr=p
    while p_corr!=None and p_corr->key!=k :
        p_corr=p_corr->next
    return p_corr
```

Costo computazionale:
$T(n) = O(n)$

7.3.2 Insert nelle liste puntate

Algoritmo: Inserimento di un elemento k in testa alla lista puntata

Input:

- p: puntatore alla testa della lista
- k: elemento da inserire

```
def insert(p,k):
    if k!=None :
        k->next=p
    p=k
    return p
```

Costo computazionale:
$T(n) = \Theta(1)$

Algoritmo: Inserimento di un elemento k dopo un elemento d

Input:

- p: puntatore alla testa della lista
- k: elemento da inserire
- d: elemento precedente a k

```
def insert(p,k,d):
    if d!=None :
        k->next=d->next
        d->next=k
    return p
```

Costo computazionale:
$T(n) = \Theta(1)$

7.3.3 Delete nelle liste puntate

Algoritmo: Eliminare un elemento k dalla lista puntata

Input:

- p: puntatore alla testa della lista
- k: elemento da eliminare

```
def delete(p,k):
    if k!=None :
        if k==p :
            p=p->next
            return p
        p_corr=p
        while p_corr->next!=k :
            p_corr=p_corr->next
        p_corr->next=k->next
    return p
```

Costo computazionale:
$T(n) = O(n)$

La funzione delete può anche essere scritta in modo ricorsivo:

Algoritmo: Eliminare un elemento k dalla lista puntata in modo ricorsivo

Input:

- p: puntatore alla testa della lista
- k: elemento da eliminare

```
def delete(p,k):
    if k==p :
        p=p->next
    else
        p->next=delete(p->next,k)
    return p
```

7.3.4 Liste doppiamente puntate

Per facilitare alcune operazioni si può organizzare la lista facendo in modo che ad ogni elemento sia possibile accedere sia dall'elemento prima che da quello dopo, aggiungendo un campo prev che punta all'elemento precedente.
La differenza di costo per le operazioni tra lista puntata singola e doppia:

Struttura	Search(A,k)	Min(A)/Max(A)	Predecessor(A,k)	Insert(A,k)	Delete(A,k)
Lista semplice	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Lista doppia	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

7.4 Pile

Una pila è una struttura dati che ha un comportamento **LIFO (Last In First Out)**, cioè gli elementi vengono prelevati in ordine inverso rispetto a come vengono inseriti.
Su una pila si possono compiere solo due operazioni, l'inserimento (Push) e l'estrazione(Pop), non è possibile scandire gli elementi né eliminare elementi senza il Pop. Le operazioni di Push e Pop operano sullo stesso puntatore, quello dell'ultimo elemento.
Push:

Algoritmo: Push di un elemento e in una pila

Input:

- top: puntatore alla testa della pila
- e: elemento da inserire

```
def push(top,e):
    e->next=top
    top=e
    return top
```

Pop:

Algoritmo: Pop del primo elemento di una pila

Input:

- top: puntatore alla testa della pila

```
def pop(top):
    if top==None :
        return None
    e=top
    top=e->next
    e->next=None
    return e,top
```

7.5 Code

La coda è una struttura dati che ha un comportamento **FIFO (First In First Out)**, cioè gli elementi vengono prelevati nello stesso ordine in cui vengono inseriti. Su una coda si possono eseguire solo l'operazione di inserimento (Enqueue) e di estrazione (Dequeue). Enqueue e Dequeue operano su due puntatori diversi, il primo sulla fine (tail) e il secondo sull'inizio (head). **Enqueue:**

Algoritmo: Enqueue

Input:

- head: puntatore alla testa della coda
- tail: puntatore alla fine della coda
- e: elemento da inserire in coda

```
def enqueue(head,tail,e):
    if tail==None :
        tail=e
        head=e
    else
        tail->next=e
        tail=e
    return tail, head
```

Dequeue:

Algoritmo: Dequeue

Input:

- head: puntatore alla testa della coda
- tail: puntatore alla fine della coda

```
def enqueue(head,tail,e):
    if tail==None :
        return None
    e=head
    head=e->next
    e->next=None
    if head==None :
        tail=None
    return tail, head, e
```

7.5.1 Code implmementate con array

Le code possono essere implementate con array se il numero massimo di elementi è noto a priori Per farlo si considera l'array in modo circolare, cioè il primo elemento è il successore dell'ultimo.

7.5.2 Code con priorità

Le code con priorità sono una variante della coda, in cui però l'ordine viene definito non dall'ordine di inserimento ma da una determinata grandezza, quindi gli elementi estratti non sono quelli più vecchi ma quelli con grandezza massima. Un esempio di coda con priorità è l'array ordinato, in cui la grandezza che decide l'ordine è il valore di key. C'è la possibilità che ci sia un problema di starvation (attesa illimitata), cioè che un elemento non venga mai estratto perché viene scavalcato da elementi con priorità più alta.

8

Alberi

L'albero è una struttura dati molto versatile e che può modellare molte situazioni reali per progettare delle soluzioni algoritmiche. Per definire bene gli alberi bisogna definire un'altra struttura dati, il grafo.

Grafo

Un grafo $G = (V, E)$ è una coppia di insiemi:

- Un insieme di nodi V
- Un insieme $E \subseteq V \times V$ di coppie ordinate di nodi chiamate archi

In un grafo si definisce un **cammino** una sequenza di nodi $\{v_1, v_2, ..., v_k\}$ tale che (v_i, v_{i+1}) sia un arco di E. Se nel cammino v_1 e v_k sono collegati allora il grafo è ciclico. Se per ogni coppia di nodi c'è almeno un cammino che li collega, il grafo si dice connesso.

8.1 Definizione di albero

Un albero è un grafo connesso e aciclico per cui se elimino qualsiasi arco dal grafo $G = (V, E)$ ottengo due grafi $G_1 = (V_1, E_1)$ e $G = (V_2, E_2)$ anch'essi connessi e aciclici.

Teorema su grafi e alberi

Dato un grafo $G = (V, E)$:

$$G \text{ albero} \iff |E| = |V| - 1 \tag{8.1}$$

8.2 Alberi radicati

Negli **alberi radicati** si sceglie un nodo principale detto **radice** e si rappresentano in modo che il percorso da ogni nodo alla radice sia dal basso verso l'alto. I nodi di questo albero vengono divisi in livelli in base alla loro distanza dalla radice (la radice è a livello 0). L'altezza è la lunghezza massimo tra un nodo e la radice.

Preso un nodo v:

- **Padre**: primo nodo sul cammino tra il nodo e la radice
- **Antenati**: tutti i nodi sul cammino tra il nodo e la radice
- **Fratelli**: nodi con lo stesso padre di v
- **Figli**: nodi che hanno v come padre (se un nodo non ha figli viene chiamato foglia)
- **Discendenti**: nodi che hanno v come antenato

Un albero viene detto ordinato se preso un nodo con k figli, posso definirne uno come primo, secondo fino al k-esimo.

8.2.1 Alberi binari

Un tipi di albero radicato e ordinato sono gli **alberi binari** in cui ogni nodo ha al massimo 2 figli e quello sinistro viene prima di quello destro. Se tutti i livelli hanno il numero massimo di nodi l'albero si dice **completo**, se tutti tranne l'ultimo hanno il numero massimo di nodi si dice **quasi completo**.
Preso un albero binario completo di altezza h:

- Numero i foglie: 2^h
- Numero totale dei nodi: $2^{h+1} - 1$

Quindi l'altezza h di un albero è:
$h = log(n+1) - 1 = log(\frac{n+1}{2})$
dove n è il numero di nodi.

8.3 Rappresentazione degli alberi in memoria

8.3.1 Memorizzazione tramite record e puntatori

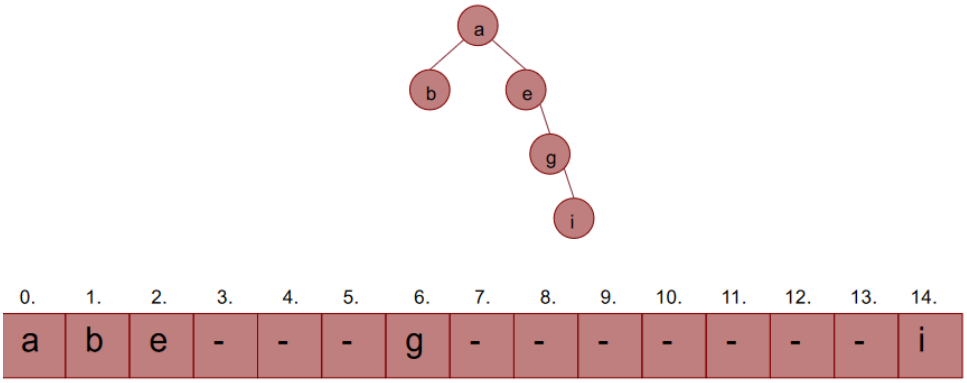
Ogni nodo è costituito da un record contenente:

- key: informazioni sul nodo stesso
- left: puntatore al figlio sinistro (`None` se non c'è)
- right: puntatore al figlio destro (`None` se non c'è)

Si accede all'albero tramite il puntatore alla radice. Questo tipo di memorizzazione ha tutti i vantaggi e l'elasticità delle strutture dinamiche basate sui puntatori (inserire nodi, spostarli) ma ha un problema, per accedere ad un nodo bisogna scendere verso di esso partendo dalla radice ma ogni volta non si sa se bisogna andare verso il figlio destro o quello sinistro.

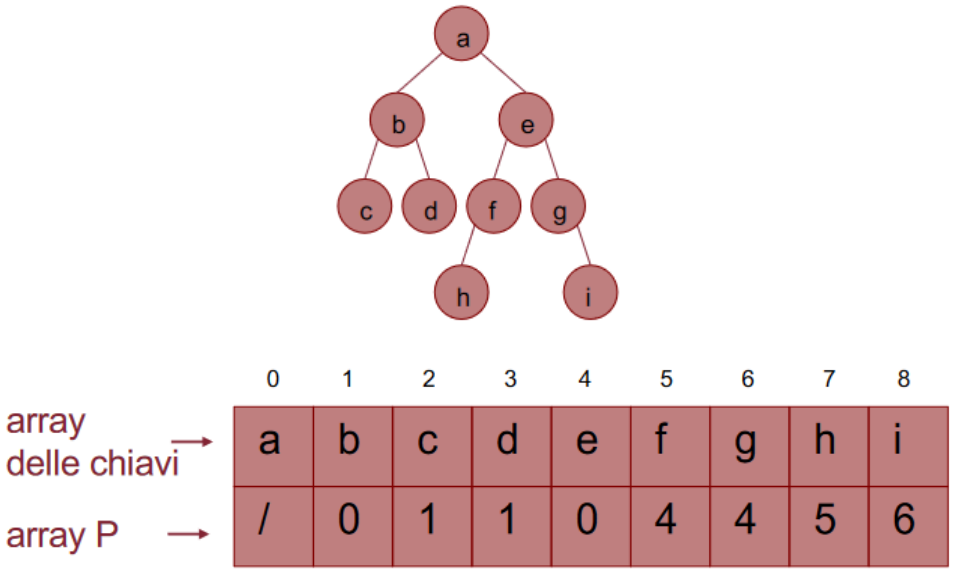
8.3.2 Rappresentazione posizionale

Si memorizzano i nodi in un array in cui la radice è all'indice 0 e i figli di un nodo di indice i sono agli indice $2i+1$ e $2i+2$. Un esempio è l'heap. Questa rappresentazione richiede di sapere in anticipo l'altezza dell'albero e ha bisogno di un array di lunghezza $2^{h+1}-1$ per contenere tutti gli elementi.



8.3.3 Vettore dei padri

Si utilizza un secondo array in cui ogni elemento indica l'indice del padre dell'elemento con indice corrente. Cioè prendendo un nodo con indice i l'array nel punto $P[i]$ avrà l'indice del padre del nodo. In questo modo non ci sono celle al centro dell'array con $None$ e può essere usato anche per alberi non binari.



8.3.4 Confronto tra i tipi di memorizzazione

La differenza a livello di costo delle operazione in base alla memorizzazione degli alberi:

Tipo di memorizzazione	Trovare il padre	Trovare i figli	Trovare il livello
Puntatori	non si può sapere	$\Theta(1)$	non si può sapere
Rappresentazione posizionale	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Vettore dei padri	$\Theta(h)$	$\Theta(2^{h+1}-1)$	$\Theta(h)$

8.4 Visite di alberi

Un'operazione basilare su un albero è l'accesso a tutti i nodi.
Nel caso degli alberi binari abbiamo tre modi per farlo:

1. Visita in **preordine**: viene effettuata l'operazione sul nodo prima dei sui sottoalberi
2. Visita in **ordine**: viene effettuata l'operazione sul nodo dopo il sottoalbero sinistro ma prima di quello destro
3. Visita in **postordine**: viene effettuata l'operazione sul nodo sia dopo il sottoalbero sinistro che quello destro

Nel caso in cui l'albero sia memorizzato con dei puntatori: **Costo computazionale:**

Algoritmo: Visita di un albero

Input:
- p: puntatore alla radice

```
def visita(p):
    if p!=None :
        // operazione sul nodo se in preordine
        visita(p->left)
        // operazione sul nodo se in ordine
        visita(p->right)
        // operazione sul nodo se in postordine
    return
```

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

$$T(n) = \begin{cases} T(k) + T(n-k-1) + \Theta(1) \\ T(0) = \Theta(1) \end{cases}$$

Nel caso peggiore: $T(n) = T(n-1) + \Theta(1) = \Theta(n)$

8.4.1 Operazioni su alberi

Per ogni operazione solitamente c'è un tipo di visita che è migliore delle altre.
Calcolo numero dei nodi:

Algoritmo: Calcolo numero nodi in postordine

```
def calcola_nodi(p):
    if p!=None :
        L=calcola_nodi(p->left)
        R=calcola_nodi(p->right)
        return L+R+1
    return 0
```

Ricerca di un nodo:

Algoritmo: Ricerca in preordine

Input:

- p: puntatore alla radice
- k: nodo da cercare

```
def cerca(p,k):
    if p!=None :
        if p==k :
            return True
        elif cerca(p->left,k)==True :
            return True
        elif cerca(p->right,k)==True :
            return True
    return False
```

Calcolo altezza dell'albero:

Algoritmo: Calcolo altezza in postordine

```
def calcola_h(p):
    if p==None :
        return -1
    if p->left==None and p->right==None :
        return 0
    h=max(calcola_h(p->left),calcola_h(p->right))
    return h+1
```

Conteggio dei nodi ad un livello k:

Algoritmo: Calcolo nodi al livello k in postordine

Input:

- p: puntatore alla radice
- k: livello da cercare
- i: livello corrente

```
def conta_k(p,k,i=0):
    if p==None :
        return 0
    if k==i :
        return 1
    L=conta_k(p->left,k,i+1)
    R=conta_k(p->right,k,i+1)
    return L+R
```

8.4.2 Visita per livelli

Se vogliamo accedere ai nodi per livelli dobbiamo usare una coda d'appoggio, nella quale inserire i nodi per poi visitarli. Usiamo una coda che ci permette di inserire direttamente i puntatori ai nodi.

Algoritmo: Visita per livelli di un albero

Input:

- r: puntatore alla radice
- head: puntatore alla testa della coda
- tail: puntatore alla fine della coda

```
def visita_lvl(r,head,tail):
    if r==None :
        return
    Enqueue(head,tail,r)
    while head!=None :  // finchè la coda non è vuota
        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
```

Questa implementazione permette di stampare tutti i nodi del livello i prima di stampare i nodi del livello $i+1$.

Costo computazionale:

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

9
Dizionari

I dizionari sono strutture dati che permettono di gestire un insieme dinamico di dati (di norma ordinato) tramite tre operazioni:

- insert
- search
- delete

Possiamo creare un dizionario in tre modi diversi:

- tabelle ad indirizzamento diretto
- tabelle hash
- alberi binari di ricerca

Per definirli useremo una determinata nomenclatura in cui:

- U: insieme dei valori chele chiavi possono assumere, costituito da valori interi
- m: numero delle posizioni disponibili nella struttura dati
- n: numero di elementi da memorizzare nel dizionario, i valori delle chiavi devono essere tutti diversi tra loro

9.1 Tabelle ad indirizzamento diretto

Una **tabella ad indirizzamento diretto** è formata da un vettore in cui ogni indice corrisponde al valore della chiave dell'elemento in quella posizione. Ipotizzando che $n \leq |U| = m$ allora un array di m posizioni permette di eseguire tutte e tre le operazioni con costo $\Theta(1)$:

Algoritmo: Insert

Input:

- D: dizionario
- k: chiave dell'elemento a cui dobbiamo inserire i dati
- dati: dati da inserire

```
def insert(D,k,dati):
    D[k]=dati
    return
```

Algoritmo: Search

Input:

- D: dizionario
- k: chiave dell'elemento di cui vogliamo i dati

```
def search(D,k):
    return D[k]
```

Algoritmo: Delete

Input:

- D: dizionario
- k: chiave dell'elemento di cui vogliamo eliminare i dati

```
def delete(D,k):
    D[k]=None
    return
```

Nella realtà però U può essere enorme, così tanto da rendere impossibile creare un array di lunghezza sufficiente oppure il numero delle chiavi usate potrebbe essere infinitamente più piccolo di $|U|$ sprecando moltissima memoria.

9.2 Tabelle hash

Si utilizzano quando l'insieme U è molto più grande del numero n di chiavi, quindi si utilizza un array da m elementi e visto che non possiamo mettere in relazione direttamente l'indice con la chiave si usa una funzione hash che calcola la posizione in base al valore della chiave. Ci potrebbero essere chiavi $k_1 \neq k_2$ che però abbiano $h(k_1) = h(k_2)$, queste vengono dette collisioni e non c'è modo di evitarle.
Una buona funzione hash deve rendere **equiprobabili** (per quanto possibile) tutti i vaori tra 0 e $m-1$ come risultati della funzione e deve essere **deterministica**, cioè se si applica più volte alla stessa chiave deve dare sempre lo stesso risultato.
La situazione ideale sarebbe quella in cui ciascuna delle m posizioni della tabella è scelta con la stessa probabilità: **ipotesi di uniformità semplice della funzione hash**. In ogni caso queste collisioni possono comunque avvenire, bisogna quindi risolverle.

9.2.1 Liste di trabocco

Con questo metodo per gestire le collisioni si inseriscono tutte le chiavi che mappano alla stessa posizione in una lista concatenata, chiamata **lista di trabocco**.
Insert:

Algoritmo: Insert

```
def insert(D,k,dati):
    D[h(k)].append(dati)
    return
```

Costo computazionale:
$T(n) = \Theta(1)$

Search:

Algoritmo: Search

```
def search(D,k):
    lista=A[h(k)]
    e=cerca(lista,k)
    return e
```

Costo computazionale:
Caso peggiore: $T(n) = O(n)$
Caso medio (con ipotesi di uniformità semplice): $T(n) = O(\frac{n}{m})$
con $\frac{n}{m} = \alpha$ che viene chiamato fattore di carico della tabella

Delete:

Algoritmo: Delete

```
def delete(D,k):
    lista=A[h(k)]
    elimina(lista,k)
    return
```

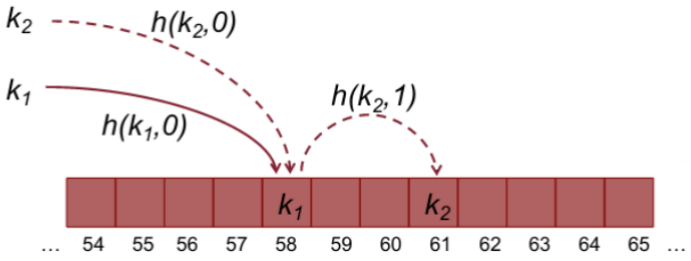
Costo computzionale:
Dipende da come è implementata la lista di trabocco, quindi segue lo stesso principio della cancellazione nelle liste concatenate.

9.2.2 Indirizzamento aperto

Si inseriscono tutti gli elementi direttamente nella tabella, senza strutture dati aggiuntive, si può applicare se:

- $m \geq n$ (quindi $\alpha \leq 1$)
- $|U| >> m$

Invece di avere una lista concatenata si inseriscono tutti gli elementi all'interno dell'array, calcolando le posizioni da esaminare. La funzione hash dipende da due parametri in questo caso, la chiave k e il numero di collisioni già trovate.



Insert:
Se inserendo un elemento la posizione iniziale determinata da $h(k, 0)$ è occupata si scandisce la tabella per trovare una posizione libera tramite funzioni $h(k, 1), h(k, 2), \ldots, h(k, m-1)$.
Costo computazionale:
Caso peggiore: $T(n) = O(n)$
Caso medio: $T(n) = O(\frac{1}{1-\alpha}) = O(\frac{m}{m-n})$

Search:
Si controlla la tabella seguendo la sequenza di funzioni hash fino a quando non si trova l'elemento o una casella vuota.
Costo computazionale:
Caso peggiore: $T(n) = O(n)$
Caso medio: $T(n) = O(\frac{1}{1-\alpha}) = O(\frac{m}{m-n})$

Delete:
Questa operazione ha dei problemi perché non si può lasciare la casella vuota perché non si potrebbero più recuperare tutti valori successivi ad essa. Se si marcasse la casella con un valore deleted il costo computazionale della ricerca non dipenderebbe solamente dal fattore di carico ma anche dalle posizione marcate. Di solito infatti non è implementato il delete nell'indirizzamento aperto.

Hashing doppio:
Per creare funzioni adatte all'indirizzamento aperto si utilizza l'hashing doppio, cioè usare due funzioni hash, una per l'accesso iniziale e una per l'aumento in base alle collisioni:

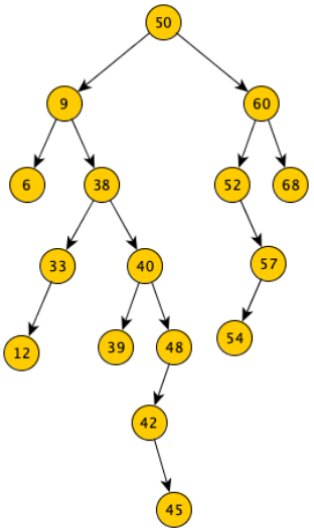
$$h(k,i) = [h_1(k) + i \cdot h_2(k)]\%m \quad \text{con } i \in [0, m-1]$$

Progettando bene le due funzioni è praticamente impossibile che due chiavi $k_1 \neq k_2$ vadano in collisione su entrambe le funzioni.

9.3 Alberi binari di ricerca

Un **albero binario di ricerca** è un albero con le seguenti proprietà:

- Ogni nodo contiene una chiave
- Il valore della chiave è maggiore del valore della chiave in ciascun nodo del sottoalbero sinistro e minore del valore della chiave di ciascun nodo del sottoalbero destro



Gli ABR sono strutture dati che supportano tutte le operazioni già definite in relazione agli insiemi dinamici più altre:

- Search(p,k): restituisce il puntatore al nodo con valore della chiave k
- Max(p)/Min(p): restituisce il puntatore al nodo con valore della chiave massimo/minimo
- Predecessor(p,k)/Successor(p,k): restituisce il puntatore al nodo con valore della chiave successivo/precedente al valore della chiave del nodo puntato da p
- Insert(p,k): inserisce l'elemento di chiave k
- Delete(p,k): elimina l'elemento puntato da k

9.3.1 Alberi binari come code con priorità

Un albero binario di ricerca può anche essere usato come una coda con priorità in cui il massimo è il nodo più a destra e il minimo il nodo più a sinistra.
Può essere anche visto come una struttura dati su cui eseguire un algoritmo di ordinamento in due fasi:

- Inserire tutte le n chiavi da ordinare nell'albero
- Fare una visita in ordine dell'albero

Costo computazionale:
$T(n) = \Theta(\text{costo costruzione albero}) + \Theta(\text{costo visita}) = \Theta(\text{costo costruzione albero}) + \Theta(n)$
L'albero viene memorizzato tramite puntatori e ogni nodo ha i puntatori ai figli destro e sinistro, il valore della chiave e il puntatore al padre.

9.3.2 Search

Per cercare un nodo specifico si scende partendo dalla radice guidati dai valori dei nodi che si incontrano durante il cammino, si va a destra se è il valore da trovare è minore del nodo attuale e a destra se è maggiore.

Algoritmo: Search

```
def search(p,k):
    if p==None or p->key==k :
        return p
    if k<p->key :
        return search(p->left,k)
    else
        return search(p->right,k)
```

Questo algoritmo è molto simile all'algoritmo di ricerca binaria che aveva costo computazionale $T(n) = O(\log n)$, ma questo algoritmo ha costo computazionale $T(n) = \Theta(n)$ nel caso peggiore. Come il Quicksort, nel caso medio l'algoritmo è più simile al caso migliore rispetto al caso peggiore, se abbiamo un albero costruito in modo casuale (ogni nodo ha la stessa possibilità di essere a destra e a sinistra) con n chiavi l'altezza media sarà $O(\log n)$.

9.3.3 Insert

Per inserire un nodo specifico si scende partendo dalla radice guidati dai valori dei nodi che si incontrano durante il cammino, si aggiunge il nodo quando si raggiunge un nodo che ha `None` nel punto dove si dovrebbe proseguire.

Algoritmo: Insert

```
def insert(p,k):
    if p==None :
        return None
    if p->key<k :
        if p->left==None :
            p->left=k
        else
            insert(p->left,k)
    else
        if p->right==None :
            p->right=k
        else
            insert(p->right,k)
    return p
```

Per il costo computazionale valgono le stesse considerazioni della ricerca perché dipende dall'altezza.

9.3.4 Massimo e minimo

Per trovare il massimo basta andare solo a destra partendo dalla radice e per il minimo basta andare solo a sinistra.

Algoritmo: Massimo

```
def max(p):
    if p==None :
        return None
    if p->right=None :
        return p
    while p->right!=None :
        max(p->right)
```

Algoritmo: Minimo

```
def min(p):
    if p==None :
        return None
    if p->left=None :
        return p
    while p->left!=None :
        max(p->left)
```

9.3.5 Predecessore e Successore

Per predecessore e successore non si intende padre o figlio ma il nodo con valore della chiave successivo o precedente rispetto al nodo.
Per trovare il predecessore di un nodo ci sono due casi:

- Il nodo ha un sottoalbero sinistro, allora il predecessore è il massimo (elemento più a destra) di questo sottoalbero
- Il nodo non ha sottoalbero sinistro, allora bisogna salire verso destra finché è possibile e poi una solo volta a sinistra (per vedere se si sta salendo verso destra o sinistra va controllato esplicitamente con `if x==x->parent->left`)

Per il successore si usa lo stesso principio ma invertendo sinistra con destra.

Algoritmo: Predecessore

```
def pred(p,k):
    p=search(p,k)
    if p->left!=None :
        p=p->left
        while p->right!=None :
            p=p->right
        return p
    else
        while p==p->parent->left :
            p=->parent
        return p->parent
```

Algoritmo: Successore

```
def succ(p,k):
    p=search(p,k)
    if p->right!=None :
        p=p->right
        while p->left!=None :
            p=p->left
        return p
    else
        while p==p->parent->right :
            p=->parent
        return p->parent
```

9.3.6 Delete

La cancellazione va eseguita in modo diverso in base a 3 casi:

1. Il nodo non ha figli, allora si elimina e si mette `k->parent->left/right=None`
2. Il nodo ha un solo figlio, allora si collega il padre del nodo con il figlio del nodo
3. Il nodo ha due figli, allora bisogna ricostruire l'albero dopo l'eliminazione, mettendo il successore o il predecessore al posto del nodo e ripetere la cancellazione sul nodo spostato

Algoritmo: Delete

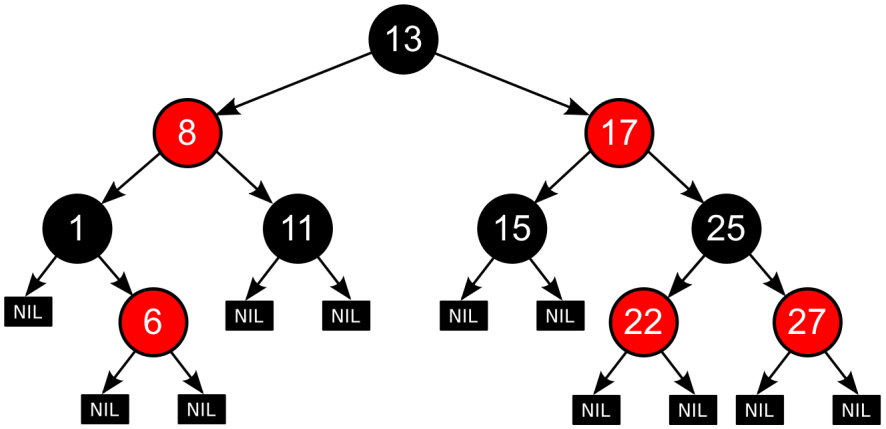
```
def delete(p,k):
    if p==None :
        return None
    p=search(p,k)
    if p->left==None and p->right==None :
        if p->parent->left==p :
            p->parent->left=None
        else
            p->parent->right=None
    elif p->left==None and p->right!=None :
        if p->parent->left==p :
            p->parent->left=p->right
        else
            p->parent->right=p->right
    elif p->left!=None and p->right==None :
        if p->parent->left==p :
            p->parent->left=p->left
        else
            p->parent->right=p->left
    else
        suc=succ(p,k)
        if p->parent->left==p :
            p->parent->left=suc
        else
            p->parent->right=suc
    delete(p->parent,suc)
```

9.4 Alberi rosso-neri

Un albero binario di ricerca permette di svolgere tutte le operazioni base in $O(h)$, ma l'altezza può essere un qualsiasi valore tra $\log n$ e n, potendo potenzialmente rallentare le operazioni (avendo altezza n). Se si riesce a garantire un bilanciamento dell'albero, riducendo l'altezza il più possibile verso $\log n$, si velocizzano di molto le operazioni. Per fare ciò si utilizzano gli **alberi rosso-neri**, in cui i nodi hanno un parametro aggiuntivo, il colore che può essere rosso o nero.

All'albero si aggiungono delle foglie fittizie (senza valore) per fare in modo che tutti i nodi veri abbiano due figli. Per risparmiare memoria tutte le foglie fittizie vengono sostituite da un solo oggetto a cui puntano tutti i puntatori che puntano verso foglie fittizie.
Gli alberi rosso-neri hanno delle proprietà:

1. Ciascun nodo è rosso o nero
2. Ciascuna foglia fittizia è nera
3. Se un nodo è rosso i suoi figli sono entrambi neri
4. Ogni cammino da un nodo a ciascuna delle foglie del suo sottoalbero contiene lo stesso numero di nodi neri
5. La radice è sempre nera
6. Nessun cammino radice-foglia può essere lungo più del doppio di un altro cammino radice-foglia



9.4.1 B-altezza

La b-altezza di un nodo x, chiamata $bh(x)$, è il numero di nodi neri dal nodo x fino alle foglie sue discendenti (uguale per tutte). La b-altezza di un albero è la b-altezza della sua radice. Ogni nodo con radice x ha almeno $2^{bh(x)} - 1$ nodi interni. Quindi un albero rosso-nero con n nodi ha altezza $h \leq 2\log(n+1)$.

9.4.2 Operazioni di base

Avendo un'altezza $h \leq 2\log(n+1)$ viene garantito un costo di $O(\log n)$ per le operazioni di:

- Search
- Max e Min
- Predecessor e Successor

9.4.3 Rotazioni

A differenza delle altre operazioni l'insert e il delete hanno bisogno che l'albero venga riaggiustato dopo che viene aggiunto o tolto un nodo per far si che i colori e i puntatori seguano le regole degli alberi rosso-neri.
Per fare ciò si usano le rotazioni che permettono in tempo $O(\log n)$ di ristabilire le proprietà dell'albero dopo un inserimento o una cancellazione. Le rotazioni possono essere sinistre o destre e non cambiano l'ordinamento dei nodi nel caso di visita in ordine.

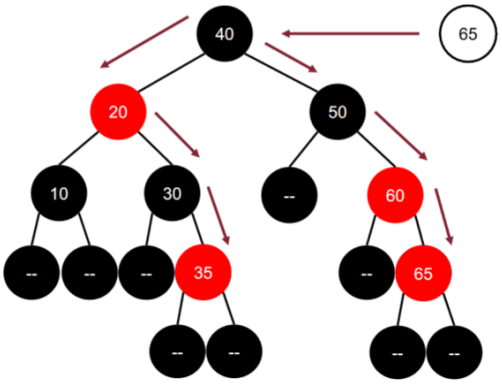
Algoritmo: Rotazione attorno ad un nodo a

```
def rot_sx(p,a):
    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
    elif a==a->parent->left :
        a->parent->left=b
    else
        a->parent->right=b
    a->parent=b
    return p
```

Costo computazionale:
$T(n) = \Theta(1)$

9.4.4 Insert

Si inserisce l'elemento come in un albero binario di ricerca, mettendo colore rosso al nuovo nodo.

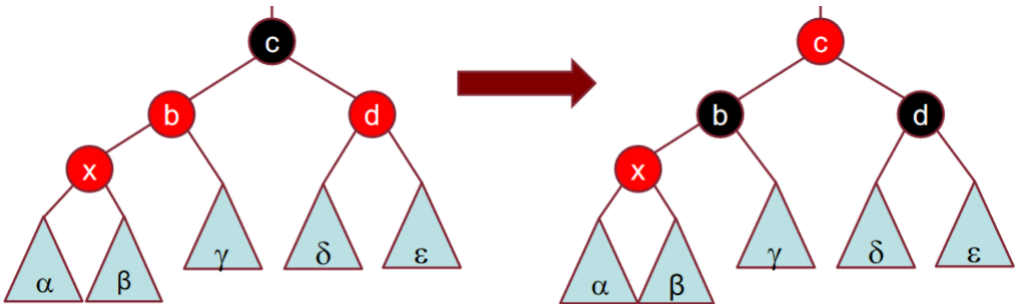


Inserendo un nodo rosso potremmo infrangere la regola 5 (radice sempre nera), sistemabile abbastanza facilmente e la regola 3 (un rosso ha sempre due figli neri). In questa situazione sappiamo che il nonno esiste ed è nero (perché un nodo rosso non può essere radice ed è il padre di un rosso).
Ci sono diversi casi possibili:

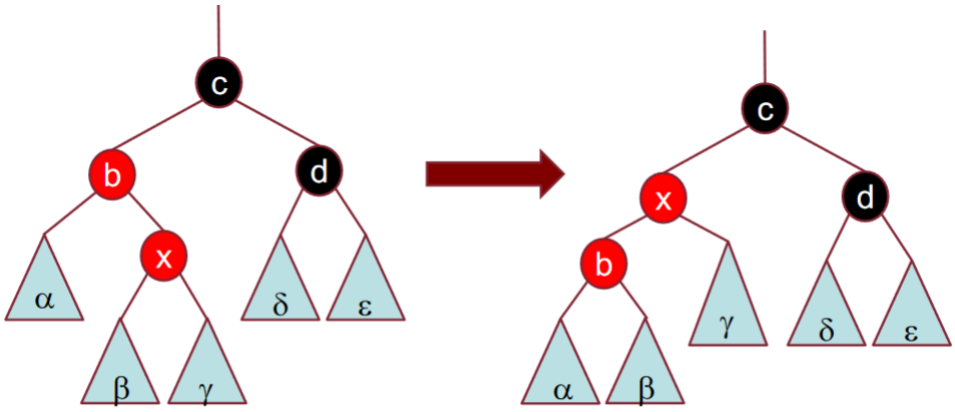
1. Il nodo inserito è la radice
2. Il nodo inserito è figlio di un nodo rosso e lo zio è rosso
3. Il nodo inserito è figlio destro di un rosso e lo zio è nero
4. Il nodo inserito è figlio sinistro di un rosso e lo zio è nero

Caso 1:
Cambiamo il colore del nodo da rosso a nero.

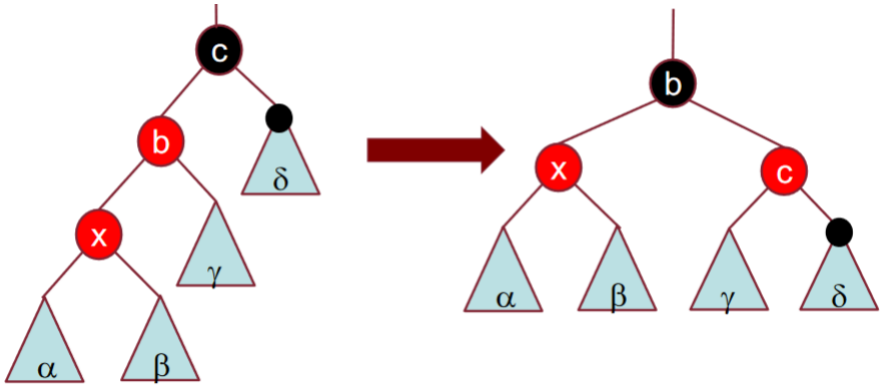
Caso 2:
Cambiamo il colore del padre e dello zio in nero e il nonno in rosso. Dopo averlo fatto o abbiamo risolto il problema o è stato spostato al livello del nonno.



Caso 3:
Si effettua una rotazione a sinistra usando il padre del nodo come perno della rotazione, facendo ciò si arriva nel caso 4.



Caso 4:
Si effettua una rotazione a destra e si cambiano alcuni colori per risolvere il problema.



Costo totale:
Per risolvere il problema si può entrare ripetutamente nel caso 2 (sempre più in alto), eventualmente nel caso 3 che viene risolto col caso 4. Visto che i casi vengono risolti con costo costante e il problema si sposta sempre verso l'alto l'inserimento ha costo $O(\log n)$.
Per l'eliminazione di un nodo vale lo stesso principio dell'insert, si elimina il nodo e lo si sostituisce con una foglia fittizia. Facendo ciò si ricade in uno dei 4 casi precedenti.