Introduzione agli Algoritmi

1-Notazione Asintotica
- 1.1-Tipi di notazioni asintotiche
  - 1.1.1-O grande
  - 1.1.2-Omega
  - 1.1.3-Theta
- 1.2-Algebra della notazione asintotica
  - 1.2.1-Gerarchie
  - 1.2.2-Regola sulle costanti moltiplicative
  - 1.2.3-Regola sulla commutatività con la somma
  - 1.2.4-Regola sulla commutatività col prodotto
  - 1.2.5-Sommatorie notevoli

2-Costo computazionale
- 2.1-Costo delle istruzioni
  - 2.1.1-Istruzioni elementari
  - 2.1.2-Istruzioni iterative
  - 2.1.3-Calcolo del costo computazionale
- 2.2-Tempi di esecuzione

3-Algoritmi di Ricerca
- 3.1-Ricerca sequenziale
- 3.2-Ricerca binaria

4-Ricorsione
- 4.1-Iterazione vs Ricorsione

5-Equazioni di ricorrenza
- 5.1-Metodo iterativo
- 5.2-Metodo dell'albero
  - 5.2.1-Alberi binari
- 5.3-Metodo di sostituzione
- 5.4-Metodo principale

6-Algoritmi di sorting
- 6.1-Insertion sort
- 6.2-Selection sort
- 6.3-Bubble sort
- 6.4-Alberi di decisioni
- 6.5-Merge sort
- 6.6-Quick sort
- 6.7-Heap sort
  - 6.7.1-Struttura dati Heap
  - 6.7.2-Funzioni ausiliarie
  - 6.7.3-Heap sort

7-Algoritmi di ordinamento con costo lineare
- 7.1-Counting sort
  - 7.1.1-Counting sort con dati satellite
- 7.2-Bucket sort

8-Strutture dati
- 8.1-Insiemi dinamici
  - 8.1.1-Operazioni di interrogazione
  - 8.1.2-Operazioni di manipolazione
- 8.2-Array
  - 8.2.1-Operazioni su array
- 8.3-Liste puntate semplici
  - 8.3.1-Search
  - 8.3.2-Insertion
  - 8.3.3-Delete

8.3.4-Liste doppiamente puntate
8.3.5-Costi computazionali
8.4-Pile
8.4.1-Operazioni su pile
8.5-Code
8.5.1-Operazioni su code
8.5.2-Code implementate con array
8.5.3-Code con priorità
9-Alberi
9.1-Grafi
9.2-Definizione di albero
9.3-Alberi radicati
9.3.1-Alberi binari
9.4-Rappresentazione in memoria
9.4.1-**Memorizzazione tramite record e puntatori**
9.4.2-**Rappresentazione posizionale**
9.4.3-**Vettore dei padri**
9.4.4-Confronto tra strutture dati
9.5-Visite di alberi
9.5.1-Differenze tra le visite
9.5.2-Visite per livelli
10-Dizionari
10.1-Nomenclatura
10.2-Tabelle ad indirizzamento diretto
10.3-Tabelle hash
10.3.1-Liste di trabocco
10.3.2-Indirizzamento aperto
10.4-Alberi binari di ricerca
10.4.1-Alberi binari come code con priorità
10.4.2-Search
10.4.3-Insert
10.4.4-Massimo e minimo
10.4.5-Predecessore e Successore
10.4.6-Delete
10.5-Alberi rosso-neri
10.5.1-B-altezza
10.5.2-Operazioni base
10.5.3-Rotazioni
10.5.4-Insert

1-Notazione Asintotica

1.1-Tipi di notazioni asintotiche

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 al crescere della grandezza dell'input.

Ci sono tre tipi di notazioni asintotiche:

- **O grande**
- **Ω (omega)**
- **Θ (theta)**

1.1.1-O grande

Date due funzioni $f(n), g(n) \geq 0$ si dice che f(n) è in O(g(n)) se esistono due costanti c, n_0 t.c. $f(n) \leq c \cdot g(n) \forall n \geq n_0$ c e n_0

Esempio:

f(n) = 3n+3

f(n) è in O(n^2) perché se $c = 6 \implies cn^2 \geq 3n + 3 \forall n \geq 1$

1.1.2-Omega

Date due funzioni $f(n), g(n) \geq 0$ si dice che f(n) è in Ω(g(n)) se esistono due costantic, n_0 t.c. $f(n) \geq g(n) \forall n \geq n_0$

> Se f(n) è in Ω(g(n))⇒g(n) è in O(f(n))

Esempio:

f(n)=2n^2+3

f(n) è in Ω(n) perché $f(n) \geq n \forall c \geq 1$

1.1.3-Theta

Date due funzioni $f(n), g(n) \geq 0$ si dice che f(n) è in $\Theta(g(n))$ se esistono tre costanti c_1, c_2, n t.c. $c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n) \forall n \geq n_0$

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-Gerarchie

> $c = O(log(n))$
> $log(n) = O(\sqrt{n})$
> $\sqrt{n} = O(n^k)$
> $n^k = O(a^n)$
> $a^n = O(n!)$
> $n! = O(n^n)$
>
> Data una funzione f(n), esistono infinite funzioni g(n) per cui f(n) è in O(g(n)) e infinite funzioni g(n) per cui f(n) è in Ω(g(n))

se $\lim_{n \to \infty} \frac{f(n)}{g(n)} = k > 0 \implies f(n) = \Theta(g(n))$

se $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \Omega(g(n))$

se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = O(g(n))$

1.2.2-Regola sulle costanti moltiplicative

Per ogni k>0 se f(n) è in O(g(n)) allora anche $k \cdot f(n)$ è in O(g(n))

> Si può fare solo se k non è all'esponente

Dimostrazione:

$\exists c, n_0$ t.c. $f(n) \leq c \cdot g(n)\ \forall n \geq n_0 \implies k \cdot f(n) \leq k \cdot c \cdot g(n) \implies k \cdot f(n) = O(g(n))$

1.2.3-Regola sulla commutatività con la somma

Se f(n) è in O(g(n)) e d(n) è in O(h(n))⇒ f(n) + d(n) = O(g(n)+h(n)) = O(max(g(n),h(n))

Dimostrazione:

$f(n) = O(g(n)) \implies \exists c_1, n_1$ t.c. $f(n) \leq c_1 \cdot g(n)$

$d(n) = O(h(n)) \implies \exists c_2, n_2$ t.c. $d(n) \leq c_2 \cdot h(n)$

$f(n) + d(n) \leq c_1 \cdot g(n) + c_2 \cdot h(n) \implies c = max(c_1, c_2) \implies f(n) + d(n) \leq c \cdot g(n) + c \cdot h(n) \implies f(n) + d(n) = O(g(n) + h(n))$

1.2.4-Regola sulla commutatività col prodotto

Se f(n) è in O(g(n)) e d(n) è in O(h(n))⇒ $f(n) \cdot d(n) = O(g(n) \cdot h(n))$

Dimostrazione:

$f(n) = O(g(n)) \implies \exists c_1, n_1$ t.c. $f(n) \leq c_1 \cdot g(n)$

$d(n) = O(h(n)) \implies \exists c_2, n_2$ t.c. $d(n) \leq c_2 \cdot h(n)$

$f(n) \cdot d(n) \leq c_1 \cdot g(n) \cdot c_2 \cdot h(n) \implies c = max(c_1, c_2) \implies f(n) \cdot d(n) \leq c \cdot g(n) \cdot c \cdot h(n) \implies f(n) \cdot d(n) = O(g(n) \cdot h(n))$

1.2.5-Sommatorie notevoli

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

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

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

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

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

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

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

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

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 massimo tra l'istruzione 1 e l'istruzione 2.

```
if condizione:
    istruzione1
else:
    istruzione2
```

2.1.2-Istruzioni iterative

Nel caso di cicli for o while il costo è pari alla somma dei costi di ciascuna iterazioni (contando anche la verifica della condizione).

Casi:

- Se tutte le iterazioni hanno lo stesso costo allora costo dell'iterazione è il costo di una singola iterazione moltiplicata per il numero di cicli(la verifica della condizione va contata una volta in più perché è quella che fa terminare il ciclo)
- Se le iterazioni hanno costo diverso bisogna sommare il costo di tutte le iterazioni

Esempi:

Calcolare il massimo in un vettore di n numeri disordinato:

```
def Trova_max(A):
    n=len(A)              #Theta(1)1
    max=A[0]              #Theta(1)2
    for i in range(1,n):  #n-1 iterazioni + Theta(1)3
```

```
        if A[i] > max:      #Theta(1)4
            max = A[i]        #Theta(1)5
    return max            #Theta(1)6
```

$T(n) = \Theta(1)_1 + \Theta(1)_2 + (n-1) * (\Theta(1)_4 + \Theta(1)_5) + \Theta(1)_3 + \Theta(1)_6 = 3\Theta(1) + \Theta(n-1) = \Theta(1) + \Theta(n) = \Theta(n)$

Calcolare la somma dei primi n interi:

```
def Somma(n):           #Theta(1)1
    somma=0               #Theta(1)2
    for i in range(1,n+1): #n iterazioni + Theta(1)3
        somma+=i            #Theta(1)4
    return somma          #Theta(1)5
```

$T(n) = \Theta(1)_1 + \Theta(1)_2 + n * \Theta(1)_4 + \Theta(1)_3 + \Theta(1)_5 = \Theta(n)$

Questo algoritmo poteva essere fatto in modo più rapido:

```
def Somma(n):
    somma=n*(n+1)/2 #Theta(1)1
    return somma    #Theta(1)2
```

$T(n) = \Theta(1)_1 + \Theta(1)_2 = \Theta(1)$

Valutazione di un polinomio:

$\sum_{k=0}^{n} a_i x^i$ nel punto x = c

```
def Calcolo_polinomio(A,c):
    somma=A[0]                    #Theta(1)1
    for i in range(len(A)):       #n iterazioni + Theta(1)2
        potenza = 1                 #Theta(1)3
        for j in range(i):          #i iterazioni + Theta(1)4
            potenza=c*potenza         #Theta(1)5
        somma=somma+A[i]*potenza    #Theta(1)6
    return somma                  #Theta(1)7
```

$T(n) = \Theta(1)_1 + n(\Theta(1)_3 + i(\Theta(1)_5) + \Theta(1)_4 + \Theta(1)_6) + \Theta(1)_2 + \Theta(1)_7 = \Theta(1)_1 + n(\Theta(i)) + \Theta(1)_2 + \Theta(1)_7 = \sum_{i=1}^{n}(\Theta(i)) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$

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.

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

Esempi con 10^9 operazioni al secondo e $n = 10^6$:

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

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

Esempio:

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

$T(n) = \Theta(n) \Rightarrow$ caso peggiore

$T(n) = \Theta(1) \Rightarrow$ caso migliore

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

```
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
```

$T(n) = \Theta(logn) \Rightarrow$ caso peggiore

$T(n) = \Theta(1) \Rightarrow$ caso migliore

Costo medio:

Per calcolare il costo medio ipotizziamo che tutte le posizioni abbiano la stessa probabilità.

Dopo i iterazioni sono state controllate 2^{i-1} posizioni, quindi la probabilità che il valore sia in una di queste posizioni è $\frac{2^{i-1}}{n}$.

costo medio = $\frac{1}{n}\sum_{i=1}^{\log n} i \cdot 2^{i-1} \Rightarrow$ per la sommatoria notevole $\sum_{i=1}^{k} i \cdot 2^{i-1} = (k-1)2^k + 1 \Rightarrow \frac{1}{n}(\log n - 1)2^{\log n} + 1 = \log n - 1 + \frac{1}{n}$

4-Ricorsione

Un algoritmo è ricorsivo quando:

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

Esempi:

Algoritmo per calcolare il fattoriale

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

Algoritmo della ricerca binaria

i_max viene inizializzato come len(A) e i_min come 0.

```
def ricerca_bin(A,v,i_min,i_max):
    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_bin(A,v,i_min,m-1)
    else:
        return ricerca_bin(A,v,m+1,i_max)
```

4.1-Iterazione vs Ricorsione

Quale algoritmo è meglio usare nei vari casi:

- 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:

Calcolo dell'n-esimo numero di Fibonacci

Versione iterativa:

```
def fibonacci(n):
    if n<=1:
        return n
    fib_1 = 1
    fib_2 = 0
    for i in range(2,n+1):
        fib_2=fib_1
        fib_1+=fib_2
    return fib_1
```

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

Versione ricorsiva:

```
def fibonacci(n):
    if n<=1:
        return n
    return (fibonacci(n-1)+fibonacci(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 sottoproblemi di dimensione n-1 e n-2.

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

Il costo è quindi esponenziale ma per calcolarlo servono le equazioni di ricorrenza.

5-Equazioni 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:

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

Il costo computazionale di questo algoritmo è:

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

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

5.1-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.

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

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 $n - k = 1 \implies k = n - 1$, l'equazione quindi diventerà:

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

Casi particolari:

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:

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

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:

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

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

Costo finale:

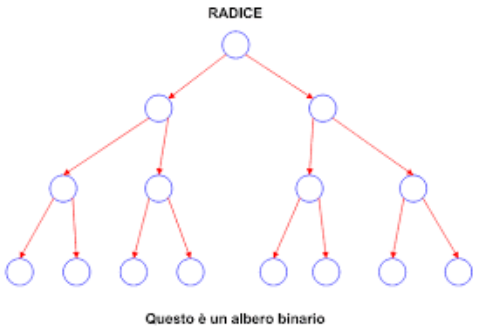
$$\sqrt{2^n} \leq T(n) \leq 2^n$$

5.2-Metodo dell'albero

Nel metodo dell'albero si rappresenta graficamente lo sviluppo del costo computazionale per calcolarlo più facilmente.

5.2.1-Alberi binari

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.



Questo è un albero binario

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

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

Il numero di nodi totali 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 è:

0: $T(n) = \Theta(n^2)$

1: $T(\frac{n}{2}) = \Theta((\frac{n}{2})^2)$

2: $T(\frac{n}{4}) = \Theta((\frac{n}{4})^2)$

…:

h: $T(1) = \Theta(1)$

Per calcolare il costo computazionale:

Livello	#nodi	costo per nodo	contributo per livello
0	2^0	$\Theta(n^2)$	$2^0\Theta(n^2)$

1	2^1	$\Theta((\frac{n}{2})^2)$	$2^1\Theta((\frac{n}{2})^2)$
2	2^2	$\Theta((\frac{n}{4})^2)$	$2^2\Theta((\frac{n}{4})^2)$
…	…	…	…
h = log(n)	$2^{\log n}$	$\Theta((\frac{n}{2^{\log n}})^2)$	$2^{\log n}\Theta((\frac{n}{2^{\log n}})^2)$

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

5.3-Metodo di sostituzione

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

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

Ipotizziamo la soluzione:

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

$T(1) = d$

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 $\Omega(n)$.

Trovare O:

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

Per induzione:

Caso base(n=1):

$$\begin{cases} T(1) = d \\ T(1) \leq k \end{cases} \implies k \geq d$$

Ipotesi induttiva:

$T(m) \leq km \ \ \forall m < n$

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:

$T(n) = O(n) \ \ \forall k \geq c$

Trovare Ω :

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

Caso base:

$$\begin{cases} T(1) = d \\ T(1) \geq h \end{cases} \implies d \geq h$$

Ipotesi induttiva:

$T(m) \geq hm \ \ \forall m < n$

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:

$T(n) = \Omega(n) \ \ \forall h \leq c$

5.4-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 si scrive come:

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

Casi:

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

> $\epsilon > 0$
>
> $c < 1$

Esempio 1:

$T(n) = 9T(\frac{n}{3}) + \Theta(n)$

a=9, b=3

$f(n) = \Theta(n)$

$n^{\log_b a} = n^{\log_3 9} = n^2$

$f(n) = n^{\log_3 9 - \epsilon}$ con $\epsilon = 1 \implies T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Esempio 2:

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

a=1, b=$\frac{3}{2}$

$f(n) = \Theta(1)$

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

$f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(\log n)$

Esempio 3:

$T(n) = 3T(\frac{n}{4}) + \Theta(n \log n)$

a=3, b=4

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

$n^{\log_b a} = n^{\log_4 3} = n^{0,7}$

$f(n) = \Omega(n^{n^{\log_4 3 + \epsilon}})$ 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

Esistono diversi algoritmi di sorting, i più semplici sono:

- Insertion sort
- Selection sort
- Bubblesort

Gli algoritmi più complicati sono:

- Merge sort
- Heip sort
- Quick sort

6.1-Insertion sort

Estrae l'elemento in posizione i dall'array e si spostano tutti verso destra gli elementi alla sinistra che sono maggiori e lo inserisco nel punto giusto.

```
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 $1 < t_j < j$

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

Cerca il minimo all'interno dell'array e lo mette in prima posizione scambiandolo e così via per ogni elemento

```
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

Confronta da destra verso sinistra ogni coppia di valori e se non sono ordinati li scambia, questo processo viene eseguito n volte

```
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]
    return A
```

Costo computazionale:

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

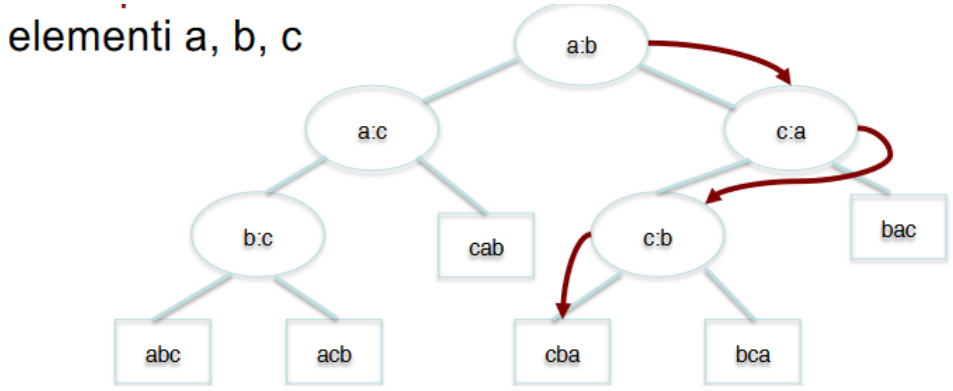
6.4-Alberi di decisioni

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 basato 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.

Esempio:

Albero di decisioni dell'insertion sort con 3 elementi:



Avendo n 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:

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

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

Con questo possiamo dire che il miglior algoritmo di sorting basato su confronti è $\Omega(nlogn)$

6.5-Merge sort

Usa un algoritmo ricorsivo con una tecnica chiamata ***divide et impera***, cioè si divide il problema in sottoproblemi risolti ricorsivamente e poi tutte le soluzioni vengono combinate.

```
def Merge_sort(A,ind_inizio,ind_fine):
    if ind_inizio<ind_fine:
        ind_medio=(ind_inizio+ind_fine)//2
        Merge_sort(A,ind_inizio,ind_medio)
        Merge_sort(A, ind_medio+1,ind_fine)
        Fondi(A,ind_inizio,ind_medio,ind_fine)
    return A
```

Costo computazionale:

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

$S(n)$ è il costo di Fondi

La funzione Fondi 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.

```
def Fondi(A,ind_inizio,ind_medio,ind_fine):
    i,j=ind_inizio, ind_medio+1
    B=[]
    while i<=ind_medio and j<=ind_fine:
        if A[i]<=A[j]:
        B.append(A[i])
        i+=1
    else:
        B.append(A[j])
        j+=1
    while i<=ind_medio:
        B.append(A[i])
        i+=1
    while j<=ind_fine:
        B.append(A[j])
        j+=1
    for i in range(len(B)):
        A[ind_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:

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

6.6-Quick sort

Nonostante abbia un tempo massimo di $\Theta(n^2)$ nella media il tempo di esecuzione è $\Theta(n \log n)$

Unisce i vantaggi di ordinare senza un array di appoggio e anche quelli del merge sort(***divide et impera***).

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

```
def Quick_sort(A,ind_inizio,ind_fine):
    if ind_inizio<ind_fine:
        ind_medio=Partiziona(A,ind_inizio,ind_fine)
        Quick_sort(A,ind_inizio,ind_medio)
        Quick_sort(A,ind_medio+1,ind_fine)
    return A
```

Costo computazionale:

$T(n) = T(k) + T(n-k) + S(n)$

S(n) è il costo di Partiziona.

```
def Partiziona(A,ind_inizio,ind_fine):
    pivot=A[ind_fine]
    i=ind_inizio-1
    for j in range(ind_inizio, ind_fine):
        if A[j] <= pivot:
            i = i + 1
```

```
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[ind_fine] = A[ind_fine], A[i + 1]
    return i + 1
```

Costo computazionale:

$S(n) = \Theta(n)$

Quindi il costo computazionale totale del quick sort:

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

Caso migliore:

$k = \frac{n}{2}$

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

Caso peggiore:

$k = 1$

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

Caso medio(in cui tutti i valori di k sono equiprobabili):

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

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).

Proprietà:

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

6.7.2-Funzioni ausiliarie

L’algoritrmo heap sort usa due funzioni ausiliarie:

- 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.

```
def Heapify(A):
    L=2*i+1   #indice figlio sinistro
    R=2*i+2   #indice figlio destro
    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)
    return A
```

Costo computazionale:

$$T(n) = T(n') + \Theta(1)$$

T(n') è il numero di nodi del sottoalbero con più nodi

Caso peggiore:

$n' = \frac{2}{3}n$

$T(n) = T(\frac{2}{3}n) + \Theta(1) = \Theta(\log n)$

Buildheap:

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

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

Costo computazionale:

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

Si può però calcolare che il costo è $T(n) = O(n)$

6.7.3-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.

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

Costo computazionale:

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

7-Algoritmi di ordinamento con costo lineare

Abbiamo visto che un algoritmo di ordinamento basato su confronti non può avere un costo computazionale minore di $\Omega(n \log n)$

7.1-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 in quell'indice.

```
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]-=1
    return A
```

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

7.1.1-Counting sort con dati satellite

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 dell'indice.

```
def Counting_sort_2(A):
    k=max(A)
    n=len(A)
    C=[0*(k+1)]
    B=[0]*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):
        B[C[A[j]]]=A[j]
        C[A[j]]-=1
    return B
```

7.2-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.

```
def Bucket_sort(A,k,n):
    for i in range(1,n):
        B.append(A[i])
    for in 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
```

8-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 dati possono essere:

- lineari o non lineari: se esiste una sequenzializzazione
- statiche e 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.

8.1-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 modifica

8.1.1-Operazioni di interrogazione

Le operazioni di interrogazione sono ad esempio:

- Search: cerca l'elemento con chiave k
- Min/Max: trova il minimo o massimo
- Predecessor/Successor: trova l'elemento che precede o succede l'elemento con chiave k

8.1.2-Operazioni di manipolazione

Le operazioni di manipolazione sono ad esempio:

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

8.2-Array

L'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.

8.2.1-Operazioni su array

Il costo computazionale di alcune operazioni su array ordinato e disordinato:

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k) Successor(S,k)	Insert(S,k)	Delete(S,k)
Vettore qualunque	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(1)$
Vettore ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$

8.3-Liste puntate semplici

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:

- campo key: contiene l'informazione(in pseudocodece scritto come `p->key`)
- campo next: contiene il puntatore che consente l'accesso all'elemento successivo(in pseudocodice scritto come `p->next`)

Le proprietà delle liste puntate:

- l'accesso avviene ad un'estremità per mezzo di un puntatore sulla testa della lista
- ogni elemento ha un puntatore che consente l'accesso all'elemento successivo
- è permesso solo un accesso sequenziale agli elementi(questo implica costo computazionale $\Theta(n)$ per qualsiasi accesso ad un elemento)

8.3.1-Search

```
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:

$T(n) = O(n)$

8.3.2-Insertion

Inserimento in testa:

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

Costo computazionale:

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

Inserimento dopo un elemento d:

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

Costo computazionale:

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

8.3.3-Delete

```
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 essere scritta anche in modo ricorsivo:

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

8.3.4-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.

8.3.5-Costi computazionali

I costi computazionali delle operazioni su una lista puntata:

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k)	Insert(S,k)	Delete(S,k)
Lista semplice	*O(n)*	*Θ(n)*	*Θ(n)*	*Θ(1)*	*O(n)*
Lista doppia	*O(n)*	*Θ(n)*	*Θ(n)*	*Θ(1)*	*Θ(1)*

8.4-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.

8.4.1-Operazioni su pile

Push:

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

Pop:

```
def Pop(top):
    if top==None:
        print("Errore: coda vuota")
        return None
    e=top
    top=e->next
    e->next=None
    return e,top
```

8.5-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).

8.5.1-Operazioni su code

Enqueue:

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

Dequeue:

```
def Dequeue(head,tail):
    if head==None:
        print("Errore: coda vuota")
        return None
    e=head
    head=e->next
    e->next=None
    if head==None:
        tail=None
    return head,tail,e
```

8.5.2-Code implementate 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.

8.5.3-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 viene mai estratto perché viene scavalcato da elementi con priorità più alta.

9-Alberi

L’albero è una struttura dati molto versatile e possono modellare molte situazioni reali per progettare delle soluzioni algoritmiche.

Per definire bene gli alberi bisogna definire un’altra struttura dati, il grafo.

9.1-Grafi

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

- Un insieme di nodi V
- Un insieme $E \subseteq VxV$ di coppie non ordinate di nodi chiamate archi o spigoli

Un cammino è una sequenza di nodi V$(v_1, v_2, ..., v_k)$ tale che (v_i, v_{i+1}) sia un arco di E per ogni $1 \leq i \leq k+1$

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.

9.2-Definizione di albero

Un albero è un grafo connesso e aciclico.

Se da un grafo $G = (V, E)$ connesso e aciclico elimino un arco qualsiasi, G si divide in due grafi $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ anch’essi connessi e aciclici.

Teorema:

Prendendo un grafo $G = (V, E)$ queste due affermazini sono uguali:

- G è connesso e aciclico(è un albero)
- #E = #V-1

9.3-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.

9.3.1-Alberi binari

Un tipi di albero radicato e ordinato sono gli alberi radicati 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 di 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.

9.4-Rappresentazione in memoria

9.4.1-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

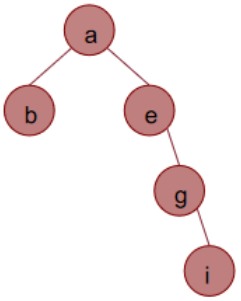
dinamiche basate sui puntatori (inserire nodi, spostarli) ma ha un problema, per accedere ad un nodo bisogna scendere verso di esso partendo dalla

radice ma ogni volta non si sa se bisogna andare verso il figlio destro o quello sinistro.

9.4.2-Rappresentazione posizionale

Si memorizzano i nodi in un array in cui la radice è all'indice 0 e i figli di un nodo all'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.

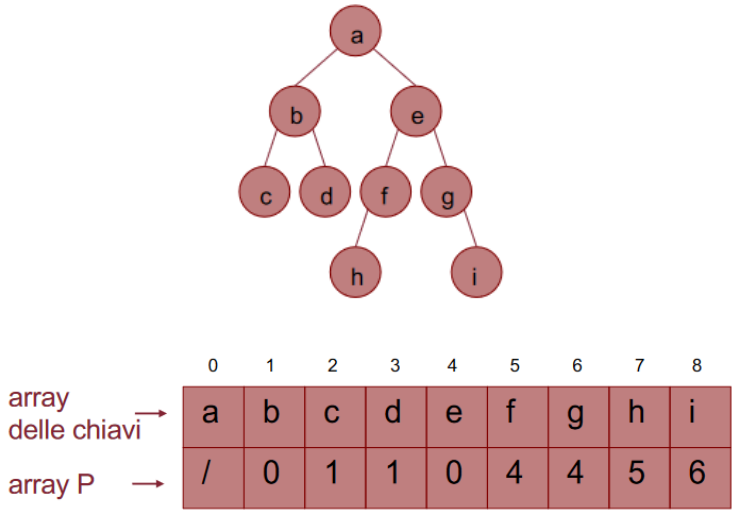


0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
a	b	e	-	-	-	g	-	-	-	-	-	-	-	i

9.4.3-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] indica 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.



9.4.4-Confronto tra strutture dati

Tipo di memorizzazione	Trovare il padre	n^o di figli	Trovare il livello
Puntatori	non lo sappiamo	$\Theta(1)$	non lo sappiamo
Rappresentazione posizionale	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Vettore dei padri	$\Theta(h)$	$\Theta(2^{h+1}-1)$	$\Theta(h)$

9.5-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:

- Visita in preordine: viene effettuata l'operazione sul nodo prima dei sui sottoalberi
- Visita in ordine: viene effettuata l'operazione sul nodo dopo il sottoalbero sinistro ma prima di quello destro
- 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:

```
def Visita(p):
    if p!=None:
        #funzione sul nodo se preordine
        Visita(p->left)
        #funzione sul nodo se in ordine
        Visita(p->right)
        #funzione sul nodo se postordine
    return
```

Costo computazionale:

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

$T(0) = \Theta(1)$

Nel caso peggiore:

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

9.5.1-Differenze tra le visite

Ci sono casi in cui uno dei tipi di visite è migliore degli altri.

Calcolo numero dei nodi:

```
def Calcola_n(p):
    if p!=None:
        num_l=Calcola_n(p->left)
        num_r=Calcola_n(p->right)
        num=num+num_l+num_r
        return num
    return 0
```

In questo caso è più utile fare una visita in postordine.

Ricerca di un nodo:

```
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
```

In questo caso è più utile fare una visita in preordine.

Calcolo altezza dell'albero:

```
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
```

In questo caso è più utile fare una visita in postordine.

Conteggio dei nodi ad un livello k:

```
def Conta_k(p,k,i):
    if p==None:
        return 0
    if k==i:
        return 1
    k_left=Conta_(p->left,k,i+1)
    k_right=Conta_(p->right,k,i+1)
    return k_left+k_right
```

In questo caso è più utile fare una visita in postordine.

9.5.2-Visite 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.

Stampare le chiave dei nodi per livelli:

```
def Visita_per_livelli(r,head,tail):
    if r==None:
    return
    Enqueue(head,tail,r)
    while CodaVuota(head)!=True:
        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:

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

10-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

10.1-Nomenclatura

U: insieme dei valori che le 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

10.2-Tabelle 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 $T(n) = \Theta(1)$:

```
def Insert(A,k):
    A[k]=dati
    return
```

```
def Search(A,k):
    return dati A[k]
```

```
def Delete(A,k):
    A[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.

10.3-Tabelle hash

Si utilizzano quando l'insieme U è molto più grande del numero n di chiavi, quindi di 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.

10.3.1-Liste di trabocco

In questa tecnica si inseriscono tutte le chiavi che mappano alla stessa posizione in una lista concatenata, chiamata **lista di trabocco**.

Insert:

```
def Insert(A,k):
    A[h(k)],append(dati)
    return
```

Costo computazionale:

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

Search:

```
def Search(A,k):
    lista= A[h(k)]
    elem=cerca(lista,k)
    return elem
```

Costo computazionale:

Caso peggiore: $T(n) = O(n)$

Caso medio(con ipotesi di uniformità semplice): $T(n) = O(\frac{n}{m})$

$\frac{n}{m} = \alpha =$ fattore di carco della tabella

Delete:

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

Costo computazionale:

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

10.3.2-Indirizzamento aperto

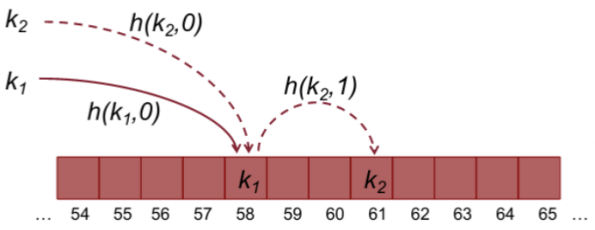
Si inseriscono tutti gli elementi direttamente nella tabella, senza strutture dati aggiuntive, si può applicare se:

- $m \geq n$ (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),…,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'hashig doppio, cioè usare due funzioni hash, una per l'accesso iniziale e una per l'aumento in base alle collisioni:

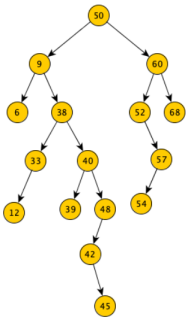
$$h(k,i) = [h_1(k) + i \cdot h_2(k)]\%m \quad \text{con } i \in [0, m-1]$$

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

10.4-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(T,k): restituisce il puntatore al nodo con valore della chiave k
- Max(T)/Min(T): restituisce il puntatore al nodo con valore della chiave massimo/minimo
- Predecessor(T,p)/Successor(T,p): restituisce il puntatore al nodo con valore della chiave successivo/precedente al valore della chiave del nodo puntato da p
- Insert(T,k): inserisce l'elemento di chiave k
- Delete(T,p): elimina l'elemento puntato da p

10.4.1-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.

10.4.2-Search

Si scende partendo dalla radice guidati dai valori dei nodi che si incontrano durante il cammino, si va a destra se è minore e a sinistra se è maggiore.

```
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 $O(\log n)$.

10.4.3-Insert

Si scende partendo dalla radice guidati dai valori dei nodi che si incontrano e si aggiunge il nodo quando si raggiunge un nodo che ha None nel punto dove si dovrebbe proseguire.

```
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
```

Costo computazionale:

Valgono le stesse considerazioni della ricerca perché dipende dall'altezza.

10.4.4-Massimo e minimo

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

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

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

10.4.5-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`)

```
def Predecessor(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=p->parent
        return p->parent
```

```
def Successor(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=p->parent
        return p->parent
```

10.4.6-Delete

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

- Il nodo non ha figli, allora si elimina e si mette `k->parent->left/right=None`
- Il nodo ha un solo figlio, allora si collega il padre del nodo con il figlio del nodo
- 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

```
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=Successor(p,k)
        if p->parent->left==p:
            p->parent->left=suc
        else:
            p->parent->right=suc
        Delete(p->parent,suc)
```

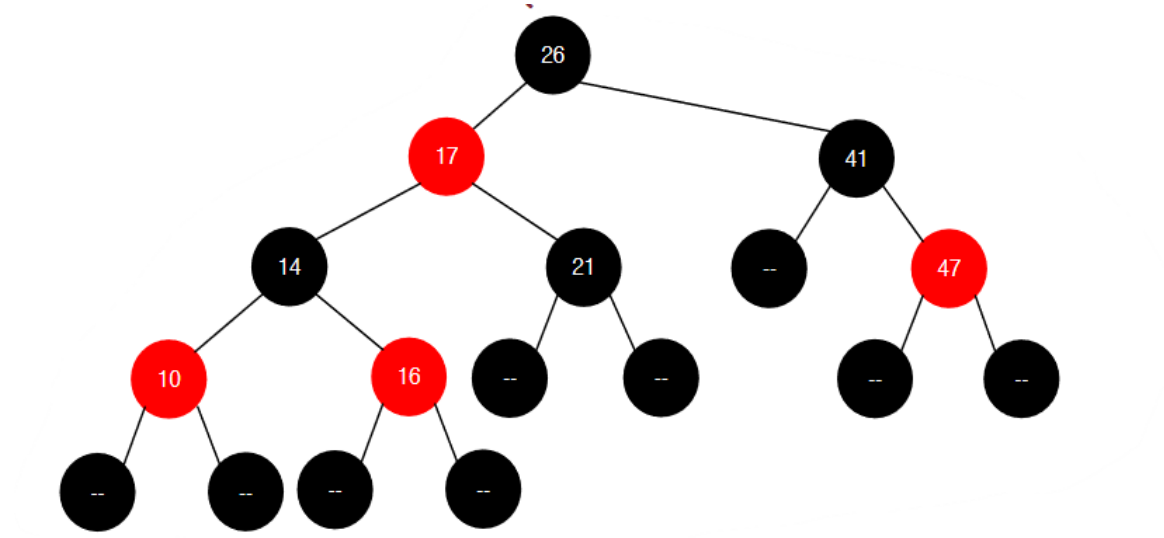
10.5-Alberi rosso-neri

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



10.5.1-B-altezza

La b-altezza di un nodo x($bh(x)$) 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 $h \leq 2\log(n+1)$

10.5.2-Operazioni 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

10.5.3-Rotazioni

A differenza delle altre operazioni l'insert e il delete hanno bisogno che l'albero venga riaggiustato dopo che viene aggiunto o tolto in 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 di in tempo $O(\log n)$ 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.

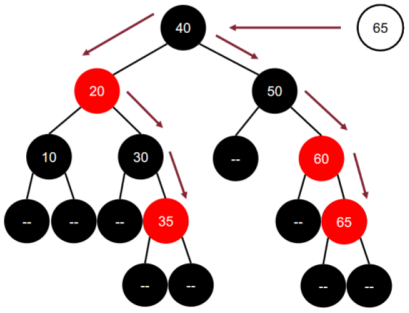
```
def rotazione_sx(p,a):#a=puntatore al nodo su cui si ruota
    b = a->right
    a->right = b->left
    if a->right != None:
        a->right->parent = a
    b->left = a
    b->parent = a->parent
    if a->parent==None:
        p = b
    elif a==a->parent->left:
        a->parent->left = b
    else: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

Costo computazionale:

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

10.5.4-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).

Per sistemare la regola 3 nel caso in cui inserissimo il nodo come figlio di un nodo rosso.

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:

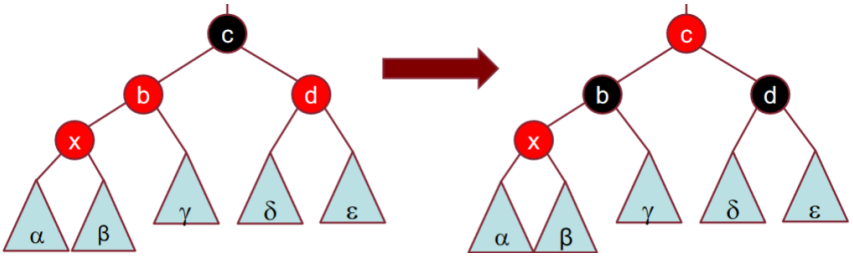
1. Il nodo inserito è la radice
2. Il nodo inserito è figlio di un nodo rosso e lo zio è rosso
3. Il nodo inserito è figlio destro di un rosso e lo zio è nero
4. Il nodo inserito è figlio sinistro di un rosso e lo zio è nero

Caso 1:

Cambiamo il colore del nodo da rosso a nero

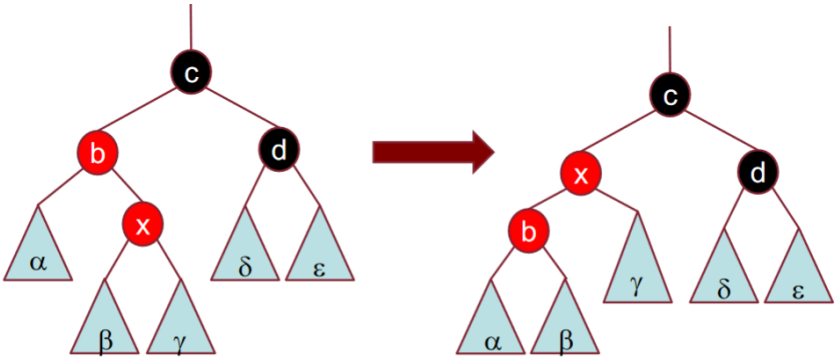
Caso 2:

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



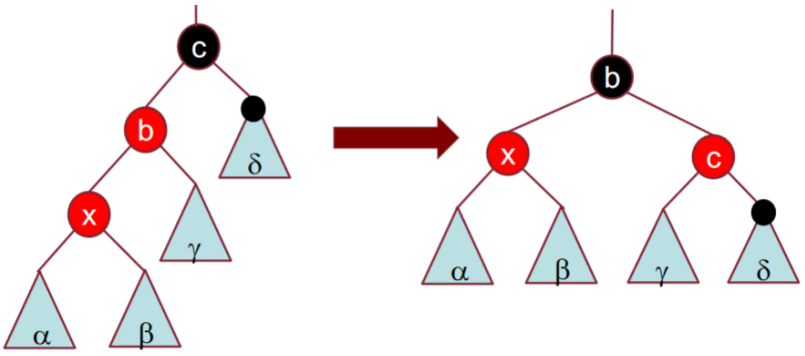
Caso 3:

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.



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 la cancellazione 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.