

Gli algoritmi

Cenni storici

Origine del nome Algoritmo: Muhammad ibn Musa al-Kwaritzmi raccolse nozioni matematiche provenienti da paesi orientali in vari testi scritti in arabo.

- Il primo testo, il cui titolo contiene la parola algebra, permise di portare in Europa le equazioni di secondo grado e fu tradotto in latino completamente.
- Il secondo testo si intitola "Algoritmi dei numeri Indorum" e si ha solo la traduzione latina, ed è la prima opera completa che introduce la numerazione posizionale ed il numero 0.
- Il termine algoritmo venne introdotto nel diciannovesimo secolo.
- Le caratterizzazioni dell'algoritmo vennero date nel primo novecento.
 - Church (1916): sistema per calcolare le funzioni.
 - Turing (1916): modello matematico che permette di simulare qualsiasi algoritmo.

Concepto di algoritmo

Insieme finito di passi/istruzioni tramite i quali si arriva alla soluzione di un problema. Ogni passo deve essere ben definito e deve essere eseguibile in un tempo finito da un agente che è in grado di calcolarlo. È possibile avere dei risultati intermedi che posso e devo memorizzare.



Algoritmi presenti da millenni: presenti babilonesi, egizi, greci, tra cui l'algoritmo di Euclide e il setaccio di Eratostene

L'algoritmo di Euclide → Calcolo del MCD tra due numeri NON NEGATIVI

Obiettivo: trovare un algoritmo migliore della scomposizione in fattori primi

PROPRIETÀ 1: MCD tra due numeri positivi = MCD fra più piccolo e differenza tra i due

DIMOSTRAZIONE: Se m ed n sono divisibili per x, allora anche (m-n), con m>n, è divisibile per x, ovvero la differenza conserva i fattori comuni di m ed n

Posso fare sottrazioni successive: m-n se m>n oppure l'opposto in caso che n>m. Se m=n so già qual è il massimo comune divisore.

↓ Devo verificare la convergenza, ovvero la condizione di Halt → numeri sono uguali.

Esempio in linguaggio di programmazione:

int MCD (int x, int y){	x= 30	y= 21
while (x!=y)	x= 30 - 21 = 9	y= 21
if (x<y) y=y-x; else x=x-y;	x= 9	y= 21 - 9 = 12
return x;	x= 9	y= 12 - 9 = 3
}	x= 9 - 3 = 6	y= 3
	x= 6 - 3 = 3	y= 3 → Condizione di halt: 3 è MCD

PROPRIETÀ 2: il MCD tra due numeri m ed n (m>n) è uguale al MCD tra n ed r= resto della divisione tra m ed n

↓ Ogni numero intero che divide m ed n divide anche r perché divide la differenza.



Convergenza: Resto < del più piccolo tra m ed n e si conserva il più piccolo tra m ed n → Halt: r=0 ed MCD è il resto precedente

Esempio in linguaggio di programmazione:

int MCD (int x, int y){	x= 30	y= 21
while (y!=0){	x= 30	y= 30 % 21 = 9
int k=x;	x= 21	y= 21 % 9 = 3
x=y;	x= 9	y= 3 % 3 = 0 → 3 è il MCD
y=k%y;	x= 3	
return x;		

↓
Differenze

→ C'è una differenza sia di passaggi che di memoria per i risultati intermedi

→ Nel secondo ci sono meno passi e non essendoci il confronto, i passi sono anche più brevi in termini di tempo

↓ Differenza di esecutore: il primo esegue operazioni molto semplici, il secondo più complesse



Algoritmi equivalenti: stessi ingressi, stessi risultati con tempi di esecuzione differenti, quantità di memoria utilizzata differente e costi di macchina differenti.

L'algoritmo di Eratostene: trovare i numeri primi $\leq n$

Primo algoritmo utilizzato: divisione di ogni numero $\leq n$ per tutti quelli prima e stabilire se è divisibile. \rightarrow Inefficiente

Secondo algoritmo utilizzato: divisione di ogni numero per tutti predecessori tra 2 e \sqrt{n} . \rightarrow Sempre molto inefficiente

Algoritmo di Eratostene:

- Scrittura di tutti i numeri tra 2 ed n (100)

2	3	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100



- Eliminazione di tutti i multipli di 2

2	3	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100



- Eliminazione di tutti i multipli di 3

2	3	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100



- Eliminazione di tutti i multipli di 5

2	3	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

- Eliminazione di tutti i multipli di 7

2	3	-4	5	-6	7	-8	-9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100



STOP perché $11 \times 11 > 100$

L'implementazione dell'algoritmo

```
void setaccio (int n){  
    bool primi [n]; //contiene booleani: se contiene true è primo, altrimenti no  
    primi[0]=primi[1]=false; //non ci interessano  
    for (int i=2; i<n; i++) primi [i]=true; //Suppongo che siano tutti primi  
    int i=1; //contatore  
    for (i++; i*i<n; i++){  
        while (!primi[i]) i++; //scorro tutti quelli false (viene usato a partire da 4 per esempio)  
        for (int k=i*i; k<n; k+=i) primi [k]=false; //metto false a tutti i multipli che incontro  
    }  
    for (int j=2; j<n; j++){  
        if (primi[j]) cout<<j<<endl; //stampa tutti i numeri primi  
    }  
}
```

L'algoritmo non esegue divisioni ma richiede memoria per risultati intermedi

La complessità degli algoritmi

Definizione

Funzione nota che associa alla dimensione del problema il costo della sua risoluzione

→ Applicata a tutti i possibili valori del dato di ingresso, qual è il costo?

Dimensione → Dati del problema

Costo → Valutato in tempo, spazio ed altri parametri rilevanti alla risoluzione del problema

Confrontare due algoritmi significa confrontare le complessità relative dei due algoritmi.

Primo esempio

```
int max(int a[], int n) {  
    int m=a[0];  
    for (int i=1; i < n;i++)  
        if (m < a[i]) m = a[i];  
    return m;  
}
```

→ Se io considero lo stesso tempo per eseguire ogni istruzione, il tempo totale impiegato è $4n$.

→ n rappresenta la dimensione dell'array

Consideriamo il caso peggiore, quindi il caso con più esecuzioni (array ordinato al contrario, n scambi).

Nel for, un n viene considerato per il test del for, un n per l'incremento di i , un altro per l'if e l'ultimo per l'assegnamento. Le costanti non si considerano nel risultato in questo caso

→ $T_{\max}(n) = 4n$

Caratteristiche del calcolo della complessità

ASTRAZIONE

- ↳ Dal computer sul quale viene eseguito
- ↳ Dal linguaggio in cui è scritto

→ CALCOLO ASINTOTICO: indipendente dalla dimensione dei dati → $n \rightarrow +\infty$

Esempi

$T_p(n)=2n^2$; $T_q(n)=100n$; $T_r(n)=5n$

Ad esempio fino a 50 $T_p(n) \leq T_q(n)$, poi si invertono perché il quadrato poi cresce più velocemente. Posso quindi dire che p è più complesso di q da un certo n in poi ma non il contrario.

Per quanto riguarda r si ha che per $n > 3$ $T_p > T_r$ in complessità

Per quanto riguarda invece T_q e T_r , posso moltiplicare T_r per una costante (20) ed ottengo 100n.

Quindi avevo $T_r(n) \leq T_q(n)$ inizialmente e dopo la moltiplicazione ho $T_q(n) \leq 20T_r(n)$, quindi essi hanno la stessa complessità.



Quindi infine basta trovare una costante con la quale moltiplicare entrambi per renderli uguali. Infatti non importa il risultato che è differente, ma importa solo il costo dell'algoritmo

→ Date due funzioni $f(n)$ e $g(n)$, posso trovare una costante c in modo da peggiorare il comportamento di $g(n)$ e quindi posso far stare $cg(n)$ sopra a $f(n)$

Notazione di O grande → DINITE ASINTOTICO SUPERIORE

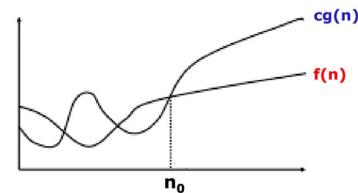
→ $f(n)$ è di ordine $O(g(n))$ se esistono un intero n_0 ed una costante $c > 0$ tali che per ogni $n \geq n_0$: $f(n) \leq cg(n)$

→ Notazione: $f(n) \in O(g(n))$ oppure con simbolo di appartenenza
↓

In sostanza vuol dire che $f(n)$ non è mai peggiore di $g(n)$ e al massimo si comporta allo stesso modo

Riguardo all'esempio di prima

- $T_p(n) \in O(TQ(n))$ [$n_0=50, c=1$] oppure [$n_0=1, c=50$]
- $T_r(n) \in O(TP(n))$ [$n_0=3, c=1$]
- $T_r(n) \in O(TQ(n))$ [$n_0=1, c=1$]
- $T_p(n) \in O(TR(n))$ [$n_0=1, c=20$]



→ quando una funzione calcola un'espressione, la funzione ha la complessità dell'espressione che essa calcola

→ È importante sapere la classe di funzioni più vicina

→ Tutte le funzioni che hanno la stessa complessità di $f(n)$ sono indicate con $O(f(n))$

Esempi

$T_{\max}(n) = 4n \in O(n)$ [$n_0=1, c=4$] $T_{\max}(n) = 4n \in O(n^2)$ [$n_0=4, c=1$]

$T_Q(n), T_R(n) \in O(n)$

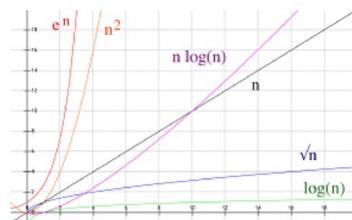
$2^{(n+10)} \in O(2^n)$ [$n_0=1, c=210$] $n^2 \in O(1/100 n^2)$ [$n_0=1, c=100$] $n^2 \in O(2^n)$ [$n_0=4, c=1$]

Regole

- Regola dei fattori costanti → Per ogni costante positiva k , $O(f(n)) = O(kf(n))$
- Regola della somma → Se $f(n)$ è $O(g(n))$, allora $f(n) + g(n)$ è $O(g(n))$ → complessità più alta tra tutte le istruzioni presenti
- Regola del prodotto → Se $f(n)$ è $O(f_1(n))$ e $g(n)$ è $O(g_1(n))$, allora $f(n)g(n)$ è $O(f_1(n)g_1(n))$ → Prodotto delle singole complessità
- Proprietà transitiva → Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora $f(n)$ è $O(h(n))$
- Le costanti hanno complessità $O(1)$
- Per $m <= p$, n^m è $O(n^p)$
- Un polinomio di grado m è $O(n^m)$

Le classi di complessità più comuni

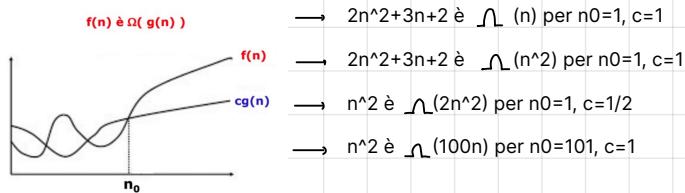
- $O(1)$: classe costante, dove la complessità è nota e non varia
- $O(\log n)$: classe logaritmica, dove la base non viene specificata in quanto è possibile cambiare la base senza variare la complessità dell'algoritmo
- $O(n)$: classe lineare: le funzioni con complessità minore si dicono sottolineari, mentre quelle con complessità superiori sopralineari
- $O(n\log n)$: classe semilogaritmica
- $O(n^2)$: classe quadratica
- $O(n^3)$: classe cubica
- $O(n^p)$: classe polinomiale
- $O(2^n)$
- $O(n^n)$



→ Teorema: per ogni k e per ogni $a > 1$ n^k è $O(a^n)$

Il limite asintotico inferiore (\underline{L})

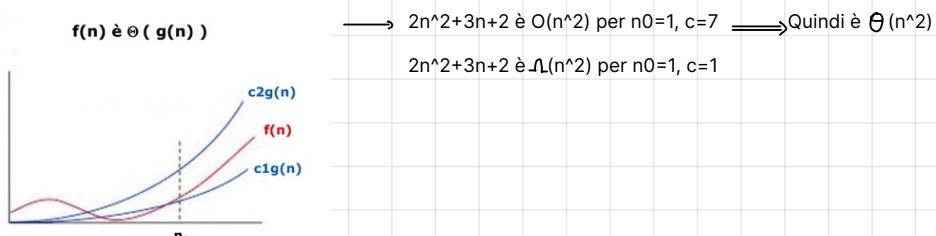
- $f(n)$ è $\underline{L}(g(n))$ se esistono n_0 ed una costante $c > 0$ tali che per ogni $n >= n_0$ si ha $f(n) >= cg(n)$



→ $f(n)$ è $O(g(n))$ se e solo se $g(n)$ è $\underline{L}(f(n))$

Il limite asintotico stretto (Θ)

- $f(n)$ è $\Theta(g(n))$ se $f(n)$ è $O(g(n))$ e $f(n)$ è $\underline{L}(g(n))$ → Esistono un intero n_0 e due costanti c_1 e $c_2 > 0$ tali che per ogni $n >= n_0$: $c_1g(n) <= f(n) <= c_2g(n)$



→ Vale anche il contrario

→ Vale quando sono della stessa complessità

I programmi iterativi: V=costante, I=variabile, E=espressione, C= comando

- $C[V] = C[I] = O(1)$ → Il tempo per una espressione costante V o un identificabile I è costante
- $C[E1 \text{ op } E2] = C[E1] + C[E2]$ → Il tempo per eseguire espressioni composte è dato dalla somma di quello delle componenti
- $C[I[E]] = C[E]$ variabile indicizzata → Tempo dato dal tempo necessario per valutare l'indice + il tempo per raggiungere la cella di memoria
- $C[I=E;] = C[E]$ → Assegnamento: tempo per calcolare l'espressione di destra + modifica cella di memoria (costante)
- $C[I[E1]=E2;] = C[E1] + C[E2]$ → Assegnamento a variabile indicizzata: valutazione indice + espressione a destra
- $C[\text{return } E] = C[E]$ → Return: tempo necessario per calcolare l'espressione di cui si fa return
- $C[\text{if } (E) C] = C[E] + C[C]$ → Condizionale: Valutazione espressione + esecuzione del comando
- $C[\text{if } (E) C1 \text{ else } C2] = C[E] + C[C1] + C[C2]$ → Condizionale + alternativa: Valutazione espressione + esecuzione del comando + alternativa
- $C[\text{for } (E1; E2; E3) C] = C[E1] + C[E2] + (C[C] + C[E2] + C[E3])O(g(n))$ → $O(g(n))$: complessità numero di volte. Tempo per initializzazione + valutazione della condizione (prima volta) sommato al tempo della valutazione, di esecuzione del comando e del passo per il numero di esecuzioni
- $C[\text{while } (E) C] = C[E] + (C[C] + C[E])O(g(n))$ → Prima valutazione + Valutazione condizione ed esecuzione moltiplicato il numero di esecuzioni
- $C[\{C1 \dots Cn\}] = C[C1] + \dots + C[Cn]$ → Blocco: somma dei singoli tempi di esecuzione

Algoritmi di ordinamento di vettori: Selection Sort

↓
il codice:

→ funzione scambia (exchange)

```
void exchange (int &x, int &y) {
    int temp=x; → O(1)
    x=y; → O(1)
    y=temp; → O(1)
}
```

→ ALGORITMO:

```
void selectionSort (int A[], int n) {
    for (int i=0; i<n; i++) { → O(n)
        int min=i; → O(1)
        for (int j=i+1; j<n; j++) { → O(n)
            if (A[j]<A[min]) min=j; } → O(1)
        exchange (A[i], A[min]); } → O(1)
    }
```

→ Complessità totale: $O(n \cdot n) = O(n^2)$

└ numero scambi: n^2

Algoritmi di ordinamento di vettori: Bubble Sort

↓
il codice:

```
for (int i=0; i<n-1; i++) { → O(n)
    for (int j=n-1; j>=i+1; j--) { → O(n)
        if (A[j]>A[j-1]) exchange (A[j], A[j-1]); } → O(1)
    }
```

→ Complessità totale: $O(n \cdot n) = O(n^2)$

└ numero scambi: n^2

Esempi di algoritmi (e loro complessità)

```
int f (int x) {
    return x; → Complessità: O(1)
}
```

Risultato: $O(n)$

```
void g (int n) {
    for (int i=1; i<=f(n); i++) { → O(n)
        cout<<f(n); → O(1)
    }
}
```

```
void g (int n) {
    for (int i=1; i<=h(n); i++) { → O(n^2)
        cout<<h(n);
    }
}
```

```
void g (int n) {
    for (int i=1; i<=k(n); i++) { → Eseguito n^2 volte: O(n^4)
        cout<<k(n);
    }
}
```

```
void p (int n) {
    int b=f(n); → O(n)
    for (int i=1; i<=b; i++) { → O(n)
        cout<<b; → O(1)
    }
}
```

```
void p (int n) {
    int b=h(n); → O(n^2)
    for (int i=1; i<=b; i++) { → O(n^2) → O(n^2) + O(n) · O(n^2)
        cout<<b; → O(1)
    }
}
```

```
void p (int n) {
    int b=k(n); → O(n^4)
    for (int i=1; i<=b; i++) { → O(n^4) → O(n^4) + O(n^2) · O(n^2)
        cout<<b;
    }
}
```

```
int h (int x) {
    return x*x; → Complessità: O(1)
}
```

Risultato: $O(n^2)$

```
int k (int x) {
    int a=0; → O(1)
    for (int i=1; i<=x; i++) { → O(n)
        a++; → O(1)
    }
}
```

Risultato: $O(n)$

return a; → $O(n)$

Algoritmi ricorsivi

- 3 fasi:
 - Individuare casi base dove si conosce il valore della funzione
 - Chiamare la funzione su un insieme sempre più piccoli
 - Verificare la convergenza della funzione ad uno dei casi base

→ Induzione naturale: sia P una proprietà sui naturali.

Base: P vale per 0

Passo induttivo: Per ogni naturale n è vero che: se P vale per n allora P vale per (n+1) e di conseguenza anche per tutti i numeri naturali

Esempio:

Dimostrare con il principio di induzione che la somma dei primi m numeri è $m(m+1)/2$

$$P(0)=0 : \checkmark \quad P(n)=\frac{1(1+n)}{2} = \frac{n(n+1)}{2}$$

$$P(n) \longrightarrow P(n+1) : 1+...+n+n+1 = \frac{n(n+1)}{2} + n+1 = \frac{n(n+1)+2n+2}{2} = \frac{n^2+n+2n+2}{2} = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2}$$

- Induzione completa: sia P una proprietà sui naturali

Base: P vale per 0

Passo induttivo: per ogni naturale n è vero che: se P vale per ogni $m \leq n$ allora P vale per $n+1$ e di conseguenza per tutti i naturali

Calcolo del fattoriale ricorsivo

→ CASO BASE: $0! = 1$

COMPLESSITÀ $T(0) = \alpha$

$T(0) = \alpha$

→ Codice: int fact (int x){

$T(n) = b + T(n-1)$

$T(1) = b + \alpha$

```
    if (x==0) return 1;
    else return x*fact(x-1);
```

$T(2) = b + b + \alpha = 2b + \alpha$

$T(n) = nb + \alpha \longrightarrow O(n)$

}

Moltiplicazione ricorsiva

→ CASO BASE: $mult(0, y) = 0$

→ Codice: int mult (int x, int y){

```
    if (x==0) return 0;
    else return (y + mult (x-1, y));
```

}

Algoritmo di Euclide ricorsivo

→ CASO BASE: $x=y$

→ Codice: int MCD (int x, int y){

```
    if (x==y) return x;
    if (x<y) return (x, y-x);
    else return (x-y, x);
```

}

Il selection sort ricorsivo

```
void r_selectionSort(int* A, int m, int i=0) { → Epressione ricorsiva
    if (i == m - 1) return;
    int min = i;
    for (int j=i+1; j < m; j++){
        if (A[j] < A[min]) min=j;
    }
    exchange(A[i], A[min]);
    r_selectionSort (A, m, i+1);
}
```

→ $T(1) = \alpha$: viene subito eseguito il return
 → $T(2) = 2b + \alpha$: viene eseguito il for
 → $T(3) = 3b + 2b + \alpha$
 → $T(4) = 4b + 3b + 2b + \alpha$
 → \vdots
 → $T(n) = nb + (n-1)b + (n-2)b + \dots + 2b + \alpha = (n + n - 1 + n - 2 + \dots + 2)b + \alpha = \frac{(n(n-1)-1)}{2}b + \alpha$

RELAZIONE NOTA: Somma dei primi numeri naturali
 $O(n^2)$

Algoritmo per l'individuazione di stringhe palindromi

int palindroma(int *a, int i=0, int j=n-1){

→ Epressione ricorsiva

```
    if (j < i) return 1;
    if (a[i]==a[j]) return palindroma(a,i+1,j-1);
    return 0;
```

→ $T(1) = \alpha$

→ $T(1) = \alpha$

$T(2) = b + \alpha$

$T(3) = b + T(2) = b + b + \alpha$

$T(4) = b + T(3) = b + b + b + \alpha$

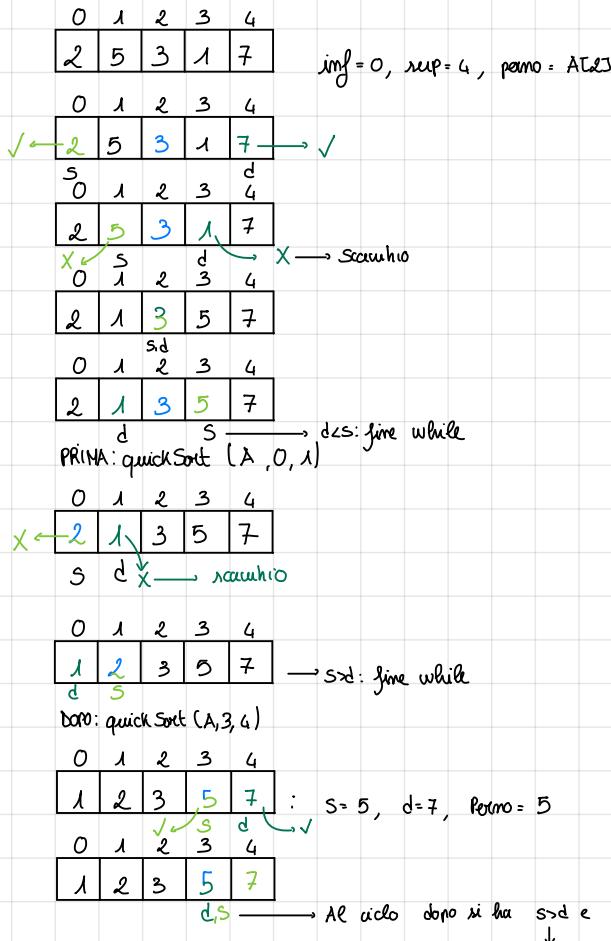
$T(n) = (n-1)b + \alpha \longrightarrow O(n)$

L'algoritmo quicksort

Viene dato un array del quale vengono specificati estremo inferiore e superiore. L'array viene analizzato e si fa un modo che la prima metà, ovvero quella che va da inf a $k-1$, contenga elementi minori o uguali a k , mentre la seconda metà deve contenere elementi maggiori o uguali a k . L'elemento k viene chiamato perno. La funzione viene poi applicata ricorsivamente fin quando non si hanno array di un solo elemento (caso base). Viene scelto come perno l'elemento centrale e poi viene fatta la suddivisione dell'array tramite il while utilizzando gli indici s e d , posizionandoli inizialmente su inf e sup . Questi verranno poi fatti scorrere rispettivamente verso sinistra e destra fin quando s non scavalca d . Il perno può cambiare posizione e questo deve essere controllato. Se $inf > d$ allora il perno si trova in prima posizione, mentre se $s > sup$ il perno si trova in ultima posizione. Il tempo di esecuzione $T(n)$ è diverso a seconda del valore assunto da k per ogni chiamata ricorsiva. Il caso migliore si ha quando il perno è circa $n/2$ e la complessità in questo caso è $O(n \log n)$. Il perno si trova sempre dopo la metà. Il caso peggiore invece si ha quando $k=1$ oppure $n-k=1$. La complessità in questo caso è $O(n^2)$. Un possibile miglioramento a selectionsort si può avere nel caso in cui si riesca a scegliere sempre il perno ottimale, seppure non si modifichi il tempo di esecuzione effettivo dell'algoritmo.

Codice ed esempio

```
void quickSort(int A[], int inf=0, int sup=n-1) {
    int perno = A[(inf + sup) / 2], s = inf, d = sup;
    while(s <= d) {
        while(A[s] < perno) s++;
        while(A[d] > perno) d--;
        if(s > d) break;
        exchange(A[s], A[d]);
        s++;
        d--;
    };
    if(inf < d) quickSort(A, inf, d);
    if(s < sup) quickSort(A, s, sup);
}
```



Inoltre, si ha $inf > d$ e $sup < s$ e di conseguenza NON viene effettuata alcuna chiamata ricorsiva

Analisi complessità

$$\begin{aligned} T(1) &= a \\ T(2) &= bn + T(n) + T(n-1) \end{aligned}$$

Case $k=1$

$$\begin{aligned} &\rightarrow T(1)=a \\ &\rightarrow T(2)=bn+a+T(n)+T(n-1) \\ &\quad \rightarrow O(n^2) \end{aligned}$$

$$\begin{aligned} \text{Case } k = \frac{m}{2} \\ T(1) &= a \\ T(m) &= bn + 2T\left(\frac{m}{2}\right) \\ &\rightarrow T(1)=a \\ &\rightarrow T(2)=2b+a \\ T(4) &= 4b+4b+2a = 8b+6a \\ T(8) &= 8b+2b+8b+8a = 3(8b)+8a \\ &\vdots \\ T(n) &= n \log n + \dots \rightarrow \text{Per la regola della somma: } O(n \log n) \end{aligned}$$

L'algoritmo di ricerca lineare

```
int RlinearSearch (int A[], int x, int m, int i=0){
    if (i==m) return 0;
    if (A[i]==x) return 1;
    return RlinearSearch(A, x, m, i+1);
}
```

$$\begin{aligned}
 T(0) &= a \\
 T(n) &= b + T(n-1) \\
 T(1) &= b + a \\
 T(2) &= b + b + a = 2b + a \\
 T(3) &= b + b + b + a = 3b + a \\
 &\vdots \\
 T(n) &= nb + a \longrightarrow O(n)
 \end{aligned}$$

Restituisce 0 se gli indici di scorrimento sono uguali (non ha trovato l'elemento), oppure 1 nel caso in cui l'elemento sia stato individuato. Dato che viene fatto uno scorrimento completo del vettore, ad ogni chiamata della funzione successiva si ha un aumento dell'indice di scorrimento.

L'algoritmo di ricerca binaria

```
int binSearch (int A[], int x, int i=0, int j=m-1){
    if (i>j) return 0;
    int k=(i+j)/2;
    if (x==A[k]) return 1;
    if (x<A[k]){
        return binSearch (A, x, i, k-1);
    }
    else{
        return binSearch (A, x, k+1, j);
    }
}
```

$$\begin{aligned}
 T(0) &= a \\
 T(n) &= b + T\left(\frac{n}{2}\right) \\
 T(1) &= b + a \\
 T(2) &= b + b + a \\
 T(3) &= b + b + b + a \\
 T(4) &= b + b + b + b + a = 3b + a = (3^1)b + a \xrightarrow{\text{logn}} O(\log n)
 \end{aligned}$$

Analizza un vettore ORDINATO e l'elemento in posizione mediana. Se l'elemento cercato è minore di quello in posizione mediana allora considera la metà di sinistra del vettore, altrimenti quella di destra. L'algoritmo continua fin quando l'elemento viene individuato o l'indice di sinistra diventa maggiore dell'indice di destra.

L'algoritmo di ricerca (gemeric)

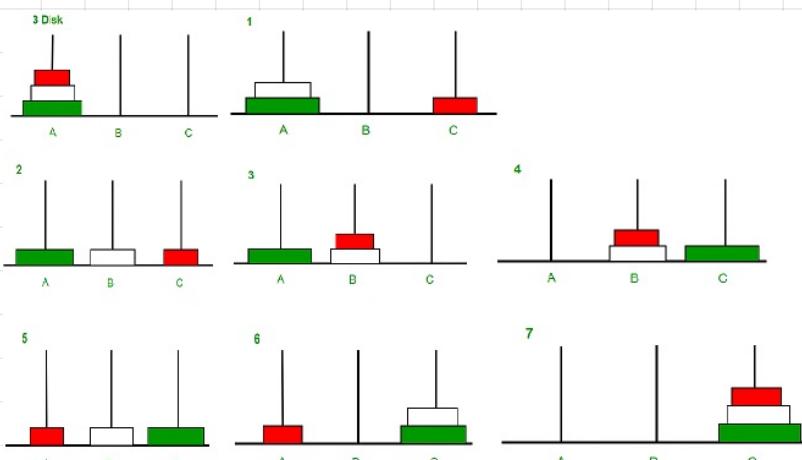
```
int Search (int A[], int x, int i=0, int j=n-1){
    if (i>j) return 0;
    int k=(i+j)/2;
    if (x==A[k]) return 1;
    return Search (A, x, i, k-1) in parallelo a Search (A, x, k+1, j);
}
```

$$\begin{aligned}
 T(0) &= a \\
 T(n) &= b + 2T\left(\frac{m}{2}\right) \\
 T(1) &= b + 2a \\
 T(2) &= b + 2b + 4a = 3b + 4a \\
 T(3) &= b + 6b + 8a = 7b + 8a \\
 T(4) &= (2n-1)b + 2na \longrightarrow T(n) \in O(n)
 \end{aligned}$$

La torre di Hanói

```
void hanoi(int n, pal A, pal B, pal C){
    if (n == 1){
        sposta(A, C);
    }
    else {
        hanoi(n - 1, A, C, B);
        sposta(A, C);
        hanoi(n - 1, B, A, C);
    }
}
```

Sono presenti 3 cerchi posti in ordine decrescente dall'alto verso il basso sul paletto A che devono essere spostati sul paletto C. E' quindi necessario impiegare due chiamate ricorsive. La prima chiamata ricorsiva trasferisce la torre degli n-1 cerchi più piccoli da A a B utilizzando C come paletto ausiliario. Una volta fatto questo è necessario spostare il cerchio più grande (l'unico presente) da A a C. A questo punto si ha la seconda chiamata ricorsiva per spostare gli elementi da B a C utilizzando il paletto A come ausiliario.



- La sequenza di passaggi nel caso di 3 elementi è la seguente:
- Il disco piccolo viene spostato da A a C
 - Il disco medio viene spostato da A a B
 - Il disco piccolo viene spostato da C a B
 - Il disco grande viene spostato da A a C
 - Il disco medio viene spostato da B a C
 - Il disco piccolo viene infine spostato da B a C

In questo modo i 3 dischi sono stati spostati dal paletto A al paletto C senza perdere il loro ordinamento iniziale

$$\begin{aligned}
 & \rightarrow T(0) = a \\
 & \rightarrow T(n) = b + 2T(n-1) \quad \xrightarrow{\text{Svolgendo i calcoli si ottiene}} \\
 & \quad \rightarrow T(0) = a \\
 & \quad \rightarrow T(1) = b + 2a \\
 & \quad \rightarrow T(2) = b + 2b + 4a = 3b + 4a \\
 & \quad \rightarrow T(3) = b + 6b + 8a = 7b + 8a \\
 & \quad \rightarrow T(4) = b + 14b + 16a = 15b + 16a \\
 & \quad \rightarrow T(5) = b + 30b + 32a = 31b + 32a \\
 & \quad \rightarrow T(n) = (2^{n-1} - 1)b + 2^n a \quad \longrightarrow O(2^n)
 \end{aligned}$$

Gli algoritmi ricorsive di tipo divide et impresa

```

void dividetimpera(S) {
    if (|S| <= m) {
        <risolvi direttamente il problema>;
    } else {
        <dividi S in b sottoinsiemi S1...Sb>;
        dividetimpera(S1);
        ...
        dividetimpera(Sia);
        <combina i risultati ottenuti>;
    }
}

```

L'insieme a cui viene applicato viene partizionato. Grazie a questa partizione si ottiene una situazione di complessità nota e negli altri casi è possibile partizionare i dati in vari sottoinsiemi (il partizionamento dipende dalla chiamata ricorsiva). Infine può essere presente una fase di combinazione dei risultati ottenuti. Il tempo di combinazione dei risultati può essere costante oppure dipendente da n. Il tempo totale dell'esecuzione sarà dato quindi dalla somma del tempo impiegato per la divisione e del tempo impiegato successivamente per la fase di combinazione dei risultati. In generale, il tempo di combinazione viene specificato nella chiamata n-esima della funzione. Quando il tempo della combinazione è costante si ha solo c, mentre se dipende da n è presente cn (oppure la funzione di dipendenza da n generica). Grazie a questi comportamenti generici, è possibile ricavare dalle relazioni di ricorrenza alcune classi di complessità note.

Formule generali

$$\begin{cases}
 T(n) = d & \text{se } n \leq m \text{ (complessità del passo noto)} \\
 T(n) = c(n) + aT\left(\frac{n}{b}\right) & \text{se } n > m
 \end{cases}$$

$$\begin{cases}
 T(n) \in O(n^k) & \text{se } a < b^k \\
 T(n) \in O(n^k \log n) & \text{se } a = b^k \\
 T(n) \in O(n^{\log_b a}) & \text{se } a > b^k
 \end{cases}$$

Relazioni di ricorrenza lineari

$$\begin{cases}
 T(n) = d \\
 T(n) = cn^k + a_1T(n-1) + a_2T(n-2) + \dots + a_rT(n-r)
 \end{cases}$$

→ Può essere polinomiale solo se un solo coefficiente di T è diverso da 0 (ed in particolare è 1), mentre in tutti gli altri casi la complessità è esponenziale

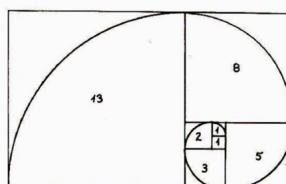
$$\begin{cases}
 T(n) = d \\
 T(n) = cn^k + T(n-1) \quad \longrightarrow T(n) \in O(n^k)
 \end{cases}$$

La serie di Fibonacci

→ Nella serie di Fibonacci si ha che ciascun elemento è dato dalla somma dei due precedenti

↳ La sezione aurea: si ha quando si verifica un determinato rapporto tra segmenti e, più specificatamente se presi due segmenti a e b e definito c = a+b, a è medio proporzionale fra c e b, posto a > b

$$\begin{array}{c}
 \text{Diagramma di Fibonacci: un segmento } a+b \text{ è diviso in } a \text{ e } b. \\
 \text{Il rapporto } a:b \text{ è uguale al rapporto } a+b:a. \\
 \text{Quindi } a+b:a = a:b \Rightarrow a+b = a \cdot \frac{a}{b} = a^2/b. \\
 \text{Ponendo } \varphi = \frac{a}{b}, \text{ si ha } \varphi + 1 = \varphi^2 \Rightarrow \varphi^2 - \varphi - 1 = 0 \Rightarrow \varphi = \frac{1+\sqrt{5}}{2} = 1.61803
 \end{array}$$



→ In sostanza quindi la relazione che lega la sezione aurea alla serie di Fibonacci è il numero phi, che rappresenta il risultato del limite tra ogni numero della serie di Fibonacci e quello che lo precede. In quanto a rappresentazione grafica, ogni numero della serie di Fibonacci può essere rappresentato all'interno della spirale illustrata qui di fianco

Algoritmi per il calcolo

① Algoritmo INDUTTIVO che lo calcola con la definizione

$$f(0) = 0, f(1) = 1$$

$$f(n) = f(n-1) + f(n-2) \text{ per } n > 1$$

→ Complessità → $T(0) = T(1) = a$

```

int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}

```

$T(n) = b + T(n-1) + T(n-2)$ → Per le regole di complessità lineare si ha che $\in O(2^n)$

② Algoritmo iterativo

```
int fibonacci(int n) {
    int k; int j=0; int f=1; → Tutte e 3 O(1) → Complessità: O(n)
    for (int i=1; i<=n; i++) { } O(n)
        k=j; j=f; f=k+j;
    }
    return j; → O(1)
}
```

③ Algoritmo ricorsivo con MEMORIZZAZIONE dell'EVENTO

```
int fibonacci( int n, int a = 0, int b = 1 ) { → Relazione di ricorsività
    if (n == 0) return a;
    return fibonacci( n-1, b, a+b );
}
```

$T(0) = a \rightarrow O(1)$
 $T(n) = b + T(n-1)$

Funzioni ricorsive per la gestione di liste

Le liste di prestano all'utilizzo di funzioni ricorsive in quanto esse stesse strutture ricorsive. Una lista infatti è costituita da diversi campi informazione e da un campo puntatore alla struttura stessa, il che determina la natura ricorsiva della struttura

```
struct Elem{
    InfoType inf; → Campo informazione
    Elem* next; → Campo puntatore alla struttura
};
```

→ Note importanti: Una sequenza vuota è una lista ed anche un elemento seguito da una serie di altri elementi costituisce una lista

Determinazione della lunghezza

```
int length (Elem* p){
    if (p==NULL) return 0;
    else return 1+length(p->next);
}
```

La determinazione della lunghezza si rende necessaria in quanto a differenza degli array, la lista è una struttura di dimensione variabile (e potenzialmente infinita). La funzione in questo caso restituisce 0 nel caso in cui l'elemento p (puntatore ad un elemento della lista) sia NULL (ovvero vuoto), mentre restituisce 1 in caso contrario. Alla fine dell'esecuzione di questa funzione si ottiene quindi la lunghezza della lista

Occorrenze di un elemento in una lista

```
int howMany (Elem* p, int x){
    if (p==NULL) return 0;
    else return (p->inf==x)+howMany(p->next,x);
}
```

La funzione prende in ingresso un puntatore ad un elemento della lista (a differenza della precedente in questo caso il puntatore viene passato per valore e non per riferimento) ed un elemento. La funzione conta quante occorrenze ci sono di quell'elemento nella lista. $p \rightarrow inf == x$ infatti restituisce 1 se vero, che va a sommarsi agli altri 1 derivanti dall'individuazione dell'elemento x nella lista

Appartenenza di un elemento alla lista

```
int belongs (Elem* I, int x){
    if (I==NULL) return 0;
    if (I->inf==x) return 1;
    return belongs (I->next, x);
}
```

La funzione al solito prende in ingresso un puntatore ad un elemento della lista e la scorre fino in fondo. In caso non venga individuato l'elemento oppure nel caso in cui la lista sia vuota viene restituito 0, altrimenti in caso di individuazione dell'elemento viene restituito 1. Ad ogni chiamata ricorsiva viene fatto uno scorrimento della lista

Eliminazione in coda

```
void tailDelete (Elem* &I){
    if (I==NULL) return;
    if (I->next==NULL){
        delete I;
        I=NULL;
    }
    else tailDelete(I->next);
}
```

La funzione prende in ingresso un puntatore ad un elemento della lista passato in questo caso per riferimento. La funzione innanzitutto procede alla verifica del caso in cui la lista sia vuota. In questo caso non viene fatta nessuna operazione. Nel caso non sia vuota, la lista viene scorsa fino ad arrivare all'ultimo elemento, identificato dal puntatore all'elemento successivo NULL. Viene quindi eliminato l'ultimo elemento e posto il puntatore dell'elemento precedente a NULL.

• Inserimento in testa

```
void tailInsert (Elem* &I, int x){  
    if (I==NULL){  
        I=new Elem;  
        I->inf=x;  
        I->next=NULL;  
    }  
    else tailInsert(I->next, x)  
}
```

La funzione, come la tailDelete, prende in ingresso un puntatore di tipo Elem passato per riferimento e l'elemento da inserire. Se il puntatore è NULL, ovvero costituisce l'ultimo elemento della lista, viene creato un nuovo elemento della lista, il cui campo informazione è riempito con l'elemento passato alla funzione ed il cui campo puntatore viene inizializzato a NULL in quanto costituisce l'ultimo elemento della lista. Lo scorrimento della lista viene fatto ad ogni chiamata ricorsiva.

• Concatenazione di due liste: Versione 1

```
void append (Elem* &l1, Elem* &l2){  
    if(l1==NULL) l1=l2;  
    else append(l1->next,l2);  
}
```

La funzione permette di ricevere in ingresso i puntatori di testa di due liste ed effettuarne la concatenazione. La funzione in particolare scorre la prima lista fino ad individuare l'ultimo elemento ($l1 \rightarrow next == NULL$) e poi fa puntare il puntatore $l1$ al primo elemento $l2$, permettendo la concatenazione delle due liste

• Concatenazione di due liste: Versione 2

```
Elem* append (Elem* l1, Elem* l2){  
    if(l1==NULL) return l2;  
    l1->next=append(l1->next, l2);  
    return l1;  
}
```

Anche questa funzione permette di concatenare due liste, il cui passaggio è però fatto per valore. In questo caso viene ritornata la lista $l1$ alla fine della funzione, che è di fatto la lista risultato della concatenazione. La funzione ad ogni esecuzione assegna a $l1 \rightarrow next$ l'elemento successivo ad $l1$ nel caso in cui lo scorrimento abbia raggiunto la fine della lista $l1$, indicato da $l1 == NULL$

• L'algoritmo di ordinamento MergeSort

L'algoritmo di ordinamento MergeSort viene utilizzato sia su array che su liste e fa utilizzo di liste per ordinare le varie strutture dati. L'algoritmo viene definito ricorsivamente nel modo seguente. Se la lista ha lunghezza 0 o 1, allora si ha il caso base in quanto essa è necessariamente ordinata. Altrimenti si procede alla suddivisione in due liste uguali alle quali viene poi applicato l'algoritmo mergeSort. Alla fine queste due liste vengono fuse insieme in modo ordinato e si ottiene in questo modo una lista di elementi ordinata. Per funzionare il mergeSort utilizza due funzioni differenti, la funzione split e la funzione merge.

```
void split (Elem* &list1, Elem* &list2){  
    if (list1 == NULL || list1->next == NULL) return;  
    Elem* l = list1->next;  
    list1->next = l->next;  
    l->next = list2; list2 = l;  
    split (list1->next, list2);  
}
```

La funzione split si occupa di dividere la lista in due sottoliste di uguale dimensione seguendo il seguente criterio. La funzione costruisce una seconda lista chiamata list2 che contiene soltanto gli elementi di list1 (lista di partenza) posti in posizione pari e modifica list1 in modo da contenere soltanto gli elementi di posizione dispari. La complessità di questa funzione, essendo richiamata $n/2$ volte è $O(n)$

```
void merge (Elem* &list1, Elem* list2){  
    if (list1==NULL){  
        list1=list2;  
        return;  
    }  
    if (list2==NULL) return;  
    if (list1->inf <= list2->inf){  
        merge(list1->next, list2);  
    }  
    else{  
        merge(list2->next, list1);  
        list1=list2;  
    }  
}
```

La funzione merge riceve in ingresso due liste già ordinate e le fonde, scorrendole contemporaneamente, in modo da ottenere una lista ordinata. Il risultato viene posto nel parametro list1 (passato per riferimento). Il caso base si ha quando una delle due liste è vuota. In questo caso la funzione restituisce l'altra. Se il primo elemento di list1 è minore o uguale al primo elemento di list2, la merge restituisce la lista che ha come primo elemento il primo elemento di list1 e come resto il risultato di merge applicato a list1 senza il primo elemento e list2. Altrimenti restituisce la lista che ha come primo elemento il primo elemento di list1 e come resto il risultato di merge applicato a list1 e a list2 senza il primo elemento

```
void mergeSort (Elem* &list1) {  
    if (list1==NULL || list1->next==NULL) return;  
    Elem* list2=NULL;  
    split (list1, list2);  
    mergesort (list1);  
    ...  
}
```

...

mergeSort (list2);

merge (list1, list2);

$\rightarrow T(1) = c_1$

$T(n) = 2T\left(\frac{n}{2}\right) + bn \rightarrow O(n \log n)$

Complessità operazioni aritmetiche e moltiplicazione ottimizzata

Le operazioni spesso, in un linguaggio di alto livello, non vengono eseguite in un unico ciclo di clock ma vengono invece scomposte in una serie di istruzioni e poi eseguite in sequenza. Questo accade se non si ha un numero di bit sufficiente per rappresentarli. Inoltre può rendersi necessario, in alcuni casi, eseguire operazioni con numeri più alti del massimo rappresentabile. In tutti questi casi le operazioni aritmetiche vengono scomposte in una serie di operazioni elementari e quindi la complessità non può più essere costante, ma dipende dall'implementazione dell'algoritmo che svolge queste operazioni. Le operazioni di somma hanno complessità lineare $O(n)$ in quanto viene fatta una somma per cifra, mentre l'algoritmo standard per la moltiplicazione ha complessità quadratica $O(n^2)$ in quanto vengono eseguite n^2 moltiplicazioni tra le cifre e somme di n prodotti parziali, ciascuno con al massimo $2n$ cifre.

L'algoritmo di moltiplicazione tuttavia può essere velocizzato. Supponiamo di avere due numeri naturali A e B con n cifre ciascuno, dove n è una potenza di 2.

Vengono poi identificate con A_{sin} e A_{des} le prime e le ultime $n/2$ cifre di A e si ripete la stessa operazione su B . Si ottengono quindi $A = A_{\text{sin}}10^{(n/2)} + A_{\text{des}}$ e $B = B_{\text{sin}}10^{(n/2)} + B_{\text{des}}$. A questo punto facendo la moltiplicazione si nota che essa è ridotta a 3 moltiplicazioni fra numeri di $n/2$ cifre più alcune addizioni, sottrazioni e traslazioni. La relazione di ricorrenza è la seguente:

$$\begin{aligned} T(1) &= d \\ T(n) &= 3T(n/2) + cn \end{aligned}$$

Gli alberi binari

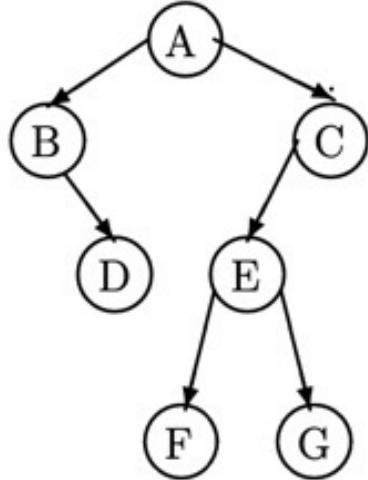
Definizione

→ Gli alberi binari sono strutture dati costituite da un insieme di nodi. Se l'insieme non è vuoto, un nodo è chiamato radice ed i restanti sono suddivisi in sottoalbero sinistro e destro, due alberi binari disgiunti. In più si ha che un nodo vuoto è un albero binario e un nodo p di due alberi B_s e B_d forma un albero di cui p è radice e B_s e B_d sono i sottoalberi sinistro e destro



- Vari elementi
 - p è il padre della radice di B_s e B_d
 - La radice di B_s è il figlio sinistro di p
 - p è antecedente di ogni nodo dell'albero, tranne di se stesso
 - Un nodo che non ha figlio destro né sinistro viene detto foglia
 - Ogni nodo dell'albero, tranne p , è discendente di p
 - Il livello di un nodo è dato dal numero dei suoi antecedenti
 - Il livello di un albero è dato dal livello massimo dei suoi nodi, quindi il cammino più lungo tra la radice ed una foglia

Rappresentazione: grafo avente come nodi i nodi dell'albero denominati da un'etichetta che rappresenta l'informazione in essi contenuta. Ogni padre è collegato al figlio tramite un arco con una freccia orientata dal padre verso il figlio. I figli destro e sinistro sono identificati da un arco spostato rispettivamente a destra o sinistra



- Insiemi ben fondati con la relazione di ordinamento di sottoalbero tra due alberi. Il minimo albero è quello vuoto ed il passo induttivo suppone vera una proprietà su B_{s1} e B_{s2} e la dimostra vera su un nodo con sottoalberi B_s e B_d . Gli alberi fanno quindi utilizzo di algoritmi ricorsivi, anche a causa della riscorsività intrinseca della loro struttura
- Operazioni più comuni: linearizzazione, ricerca, inserimento, cancellazione

Le linearizzazioni (visite) → Trasformano un albero in una sequenza contenente i nomi dei suoi nodi

- Visite in ordine anticipato (preorder): viene controllato il caso base (albero vuoto) e poi viene controllata la radice. In seguito viene richiamata la stessa funzione sul sottoalbero sinistro e poi sul destro. La ricorsione si chiude ogni volta che la funzione ricorsiva incontra il caso base
- Visite in ordine differito (postorder): viene verificato il caso base e poi, in ordine, viene esaminato il sottoalbero sinistro, poi il destro ed infine la radice
- Visite in ordine simmetrico (inorder): viene verificato il caso base e poi in ordine viene visitato il sottoalbero sinistro, viene esaminata la radice ed infine viene visitato il sottoalbero destro

Esempio con albero precedente → ABDCEFG
→ DBFGECA
→ BDAFEGC

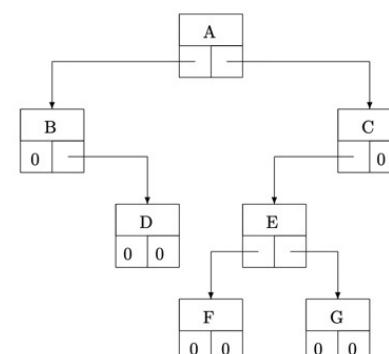
La memorizzazione in lista multipla e le visite in C++

- Viene definita una struttura node simile ad una lista. Analogamente alle liste è presente un campo informazione di un tipo a piacere e , a differenza delle liste sono presenti due campi puntatore, uno al sottoalbero sinistro ed uno al destro. Quando non è presente uno dei due sottoalberi il campo puntatore corrispondente viene posto a NULL

```

struct Node{
    LabelType label;
    Node* left;
    Node* right;
};
  
```

- Visite anticipate: void preOrder (Node* tree){ → ALBERO (puntatore all'albero)
 if (!tree) return; → Controllo del caso base (tree NULL)
 else{
 cout<<tree->label; → Analisi (stampa) della radice
 preOrder(tree->left); → Chiamata sul sottoalbero sinistro
 preOrder(tree->right); → Chiamata sul sottoalbero destro
 }
 }



```

→ Visite inOrder e postOrder: void postOrder (Node* tree){
    if (!tree) return;
    else{
        postOrder (tree->left);
        postOrder (tree->right);
        cout<<tree->label;
    }
}

```

```

void inOrder (Node* tree){
    if (!tree) return;
    else{
        inOrder (tree->left);
        cout<<tree->label;
        inOrder (tree->right);
    }
}

```

La complessità in base al numero di modi

- La complessità generalmente è valutata in funzione del numero n di nodi dell'albero. Supponendo che il tempo per il calcolo del nodo sia costante, la complessità delle visite è data dalle relazioni seguenti: $T(0)=a$

$$\downarrow \quad T(n)=b+T(ns)+T(nd) \rightarrow \text{con ns ed nd vengono indicati rispettivamente il numero di nodi del sottoalbero sinistro e destro. Si ha che } ns+nd=n-1 \text{ in quanto si esclude il caso base}$$

La complessità di una visita è quindi lineare

Alberi bilanciati e pieni

- Un albero viene definito bilanciato se i nodi di tutti i livelli escluso l'ultimo hanno due figli. Un albero può anche essere quasi bilanciato, ed in questo caso è un albero bilanciato soltanto fino al penultimo livello. Un albero bilanciato non vuoto di livello k ha $2^{k+1}-1$ nodi e 2^k foglie. Un albero quasi bilanciato non vuoto di livello k ha un numero di nodi compreso fra 2^k e $2^{k+1}-1$ ed un numero di foglie compreso tra 2^{k-1} e 2^k .
- Un albero pienamente binario è un albero binario in cui tutti i nodi tranne le foglie hanno due figli. In un albero binario pienamente binario il numero di nodi interni è uguale al numero di foglie meno 1

La complessità in funzione dei livelli

- $T(0)=a$ → È come considerare alberi binari che hanno lo stesso numero di nodi del livello precedente -1
- $T(k)=b+2T(k-1)$
- $O(2^k)$

Funzioni utili sugli alberi binari

Conteggio dei nodi:

```

int nodes (Node* tree){
    if (!tree) return 0; → Caso base: Albero vuoto
    return 1+nodes(tree->left)+nodes(tree->right); → Funzione ripetuta sui sottoalberi sx e dx
}

```

Conteggio delle foglie

```

int leaves (Node* tree){
    if (!tree) return 0; → Caso base 1: L'albero è vuoto
    if (!tree->left && !tree->right) return 1; → Caso base 2: individuazione della foglia: NON possiede figli né dx né sx
    return leaves(tree->left)+leaves(tree->right); → Funzione eseguita sui sottoalberi sinistro e destro
}

```

Ricerca di un nodo

```

Node* findNode (LabelType n, Node *tree){ → PARAMETRI: Etichetta ricercata e puntatore ad albero
    if (!tree) return NULL; → Caso base 1: Albero vuoto
    if (tree->label == n) return tree; → Restituzione della PRIMA OCCORRENZA dell'etichetta
    Node* a=findNode(n, tree->left); → Individuata l'etichetta con visita in pre-order → Le funzioni si chiudono
    if (a) return a; → con la chiusura a sinistra
    else return findNode(n, tree->right);
}

```

Eliminazione albero

```

void delTree (Node* &tree){ → ATT: Il puntatore ad albero va passato per riferimento perché deve essere modificato
    if (tree){
        delTree (tree->left); } → Eliminazione di tutti gli elementi
        delTree (tree->right);
        delete tree; → Eliminazione del puntatore precedente
        tree=NULL; → Albero meno VUOTO
    }
}

```

Inserimento di un nodo figlio

```
int insert (Node*& root, LabelType son, LabelType father, char c){ —> Viene passato il modo da inserire, se va inserito a destra o  
if (!root){ —> Se root è NULL  
    root=new Node; —> Nuovo nodo  
    root->label=son; root->left=root->right=NULL; —> NON ha figli né a sinistra, né a destra  
    return 1; —> INSERITO correttamente  
}  
  
Node* a=findNode (father,root); —> Viene ricercato il nodo  
if (!a) return 0; —> Se non viene trovato l'esecuzione si interrompe  
if (c=='l' && !a->left){ —> Se si deve inserire a sinistra, il modo dove si vuole inserire deve essere root a sinistra  
    a->left=new Node; —> Viene creato il nuovo nodo  
    a->left->label=son; a->left->left=a->left->right=NULL; —> Viene assegnata l'etichetta desiderata al nodo sinistro e  
    return 1; —> Inserito con successo  
}  
  
if (c=='r' && !a->right){ —> Analogico ad inserimento sinistro  
    a->right=new Node;  
    a->right->label=son; a->right->left=a->right->right=NULL;  
    return 1;  
}  
  
return 0; —> Inserimento fallito
```

da classe BinTree

La classe contiene il campo privato Node che rappresenta la struttura stessa del nodo. Si ha poi il campo root, che rappresenta il puntatore all'albero istanziato.

Vengono inoltre introdotte, sempre nel campo privato, le funzioni di visita e tutte le altre funzioni di utilità applicabili all'albero. Queste funzioni vengono poi rese opportunamente utilizzabili nel campo pubblico. Sono presenti infine nel campo pubblico il costruttore ed il distruttore.

Gli alberi generici

Definizione

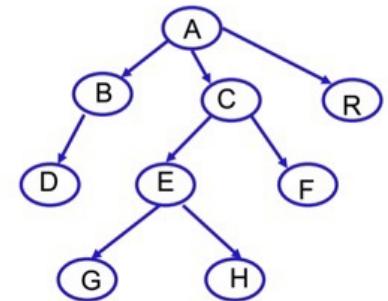
- Sono insiemi non vuoti di nodi, di cui uno è chiamato radice, ed i restanti sono divisi in sottoinsiemi disgiunti, che sono a loro volta alberi e sono detti sottoalberi della radice. Mentre in un albero binario anche un nodo vuoto era un albero, in questo caso non lo è, quindi gli alberi generici non possono essere vuoti. In generale,
- Un nodo p è un albero ed è anche la radice dell'albero.
- Un nodo p più un insieme di alberi {A₁, ..., A_n}, n >= 1 è un albero, di cui p è radice, mentre A₁, ..., A_n sono detti primo, secondo, ..., n-esimo sottoalbero di p.

- Tutti i figli sono alberi generici disgiunti e tutte le radici dei sottoalberi sono radici di altri sottoalberi figli.

→ Si definiscono fratelli i nodi figli della stessa radice.

→ Una foglia è un nodo che non ha figli e non esiste il livello -1 perché l'albero generico non può essere vuoto. L'altezza è il livello massimo delle foglie.

→ Negli alberi generici è significativo l'ordine dei figli, mentre non è importante la loro posizione, in quanto il fatto che il primo sottoalbero parta da sinistra o da destra è solo una convenzione adottata dal programmatore. Per convenzione in questo caso viene scelto il primo figlio a sinistra



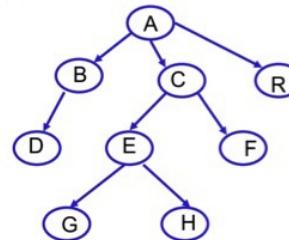
Le visite

Anche per gli alberi generici sono definite le visite come per gli alberi binari. Tuttavia, data la presenza di più di due sottoalberi per ogni radice, non è possibile effettuare visite di tipo inorder.

La visita preorder è analoga a quella fatta sugli alberi binari ma, data l'impossibilità della presenza di alberi vuoti, è assente il caso in cui l'albero è vuoto. Inoltre è necessario, per ogni esecuzione di una visita, considerare tutti gli eventuali n rami.

```

void preOrder (albero){
    esamina la radice;
    se l'albero ha n sottoalberi{
        preOrder (primo sottoalbero);
        ...
        preOrder (n-esimo sottoalbero);
    }
}
  
```

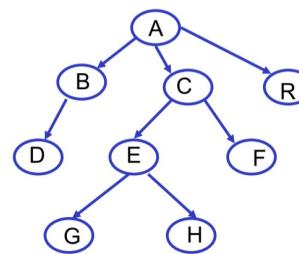


A B D C E G H F R

Anche la visita postorder è analoga a quella degli alberi binari ed anche in questo caso è assente il caso dell'albero vuoto.

```

void postOrder (albero){
    se l'albero ha n sottoalberi{
        postOrder (primo sottoalbero);
        ...
        preOrder (n-esimo sottoalbero);
    }
    esamina la radice
}
  
```



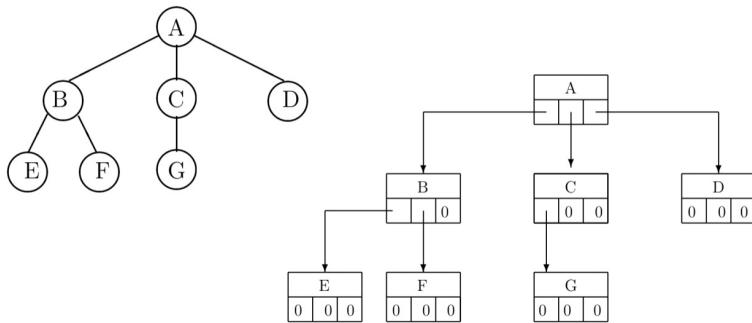
D B G H E F C R A

La memorizzazione figlio-fratello

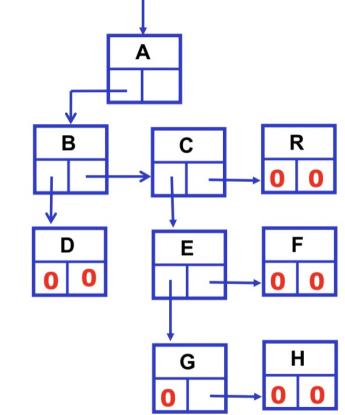
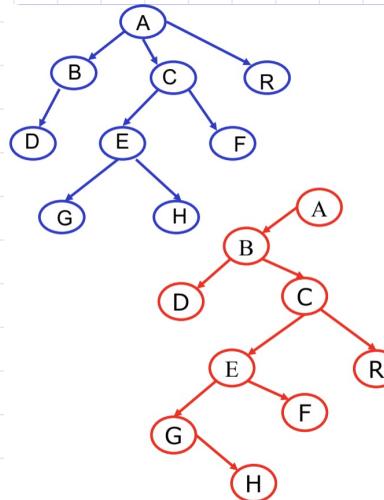
- Essendo strutture dati dalle caratteristiche simili, la memorizzazione potrebbe essere la stessa, ovvero potrebbe essere utilizzata la memorizzazione a liste multiple. Nel caso di alberi generici però i campi puntatore a figli dovrebbero essere in numero pari al numero massimo dei figli che un ciascun sottoalbero può avere. Potrebbe, per questo scopo, essere utilizzato un vettore di puntatori. I problemi principali di questo tipo di memorizzazione sono la necessità di dover specificare il numero massimo di figli e lo spreco di memoria elevato generato dalla rappresentazione dei valori nulli.

- La memorizzazione da scegliere è quindi quella figlio-fratello. In questo tipo di memorizzazione, supposto che il primo sottoalbero sia quello a sinistra e l'ultimo quello a destra, si utilizza la stessa configurazione a 3 puntatori utilizzata per gli alberi binari. Il primo puntatore contiene il nome del nodo, il secondo puntatore indica il primo figlio del nodo ed il terzo puntatore indica il primo fratello del nodo. In questo caso lo spreco di memoria è ridotto, in quanto sono presenti pochi valori NULL. Qualsiasi albero generico può essere rappresentato con questa rappresentazione, essendo essa convenzionale.

Memorizzazione tramite liste multiple ↴



Memorizzazione figlio-fratello ↴



Le visite con la memorizzazione figlio-fratello

Per quanto riguarda le visite, è possibile definire sia visite inorder che preorder, essendo di fatto adesso alberi binari. E' importante notare che nel caso di alberi generici, è assente il caso base di un albero vuoto, in quanto per definizione l'albero generico non può esserlo. La visita inorder di un albero memorizzato tramite la tecnica figlio-fratello è equivalente a quella postorder di un albero generico, mentre la visita preorder di un albero memorizzato tramite la tecnica figlio-fratello è equivalente alla visita preorder dell'albero generico.

Alcune funzioni di utilità

La funzione conta nodi ha come caso base l'albero vuoto e funziona allo stesso modo degli alberi binari.

```
int nodes (Node* tree){  
    if (!tree) return 0;  
    return 1 + nodes (tree->left) + nodes (tree->right);
```

La funzione che conta le foglie controlla sempre che l'albero non sia vuoto ed è in parte differente dalla funzione che conta le foglie degli alberi binari in quanto nella rappresentazione figlio-fratello è sufficiente che il figlio sinistro sia nullo per identificare una foglia. Questo accade perché se è assente il primo figlio, non potranno che essere assenti anche tutti gli altri.

```
int leaves (Node* tree){  
    if (!tree) return 0;  
    if (!tree->left) return 1 + leaves (tree->right);  
    return leaves (tree->left) + leaves (tree->right);  
}
```

La funzione di inserimento permette di inserire un nodo in un punto specificato dal programmatore. Per fare questa operazione è necessaria una funzione di appoggio che permette di inserire un nodo in fondo ad una lista di fratelli. Questa funzione lavora in questo modo: se l'albero è vuoto si presenta il caso base e viene subito inserito il nodo, mentre se non lo è essa scorre fino in fondo la lista a destra in quanto rappresenta quella dei fratelli e quando si presenta il caso base inserisce il nodo.

```
void addSon (InfoType x, Node* &list){  
    ... list -> left = list -> right = NULL;  
    if (!list){  
        list = new Node;  
        list -> label = x;...  
    }  
    else  
        addSon (x, list -> right);  
}
```

Dopo aver definito la funzione addNode, è possibile definire la funzione insert che ne fa utilizzo. E' necessario innanzitutto ricercare il nodo con la funzione findNode già definito e restituire il puntatore a quel nodo. Se il nodo non è presente la funzione restituisce NULL e l'esecuzione della funzione termina.

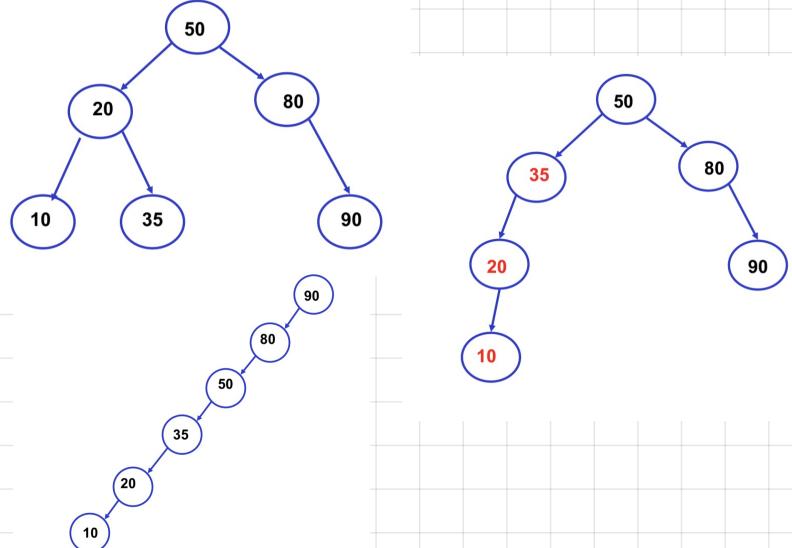
Altrimenti viene chiamata la funzione addSon che aggiunge il figlio a sinistra del nodo specificato e viene infine restituito 1 per indicare che l'operazione è riuscita

```
|_, int insert (InfoType son, InfoType father, Node* &tree){  
    Node* a = findNode (father, tree);  
    if (!a) return 0;  
    addSon (son, a -> left);  
    return 1;  
}
```

Gli alberi binari di ricerca

Definizione

Un albero binario di ricerca è un albero binario etichettato tale che sulle etichette dei nodi è stabilita una relazione di ordinamento e per ogni nodo p vale la seguente proprietà: tutti i nodi del sottoalbero sinistro di p hanno etichette minori di quella di p e tutti i nodi del sottoalbero destro di p hanno etichette maggiori di quella di p. Questa proprietà è ricorsivamente valida per tutti i sottoalberi. Gli alberi binari di ricerca sono fondamentali per semplificare alcuni algoritmi applicabili agli alberi binari, quali la ricerca, oppure per definire alcune tipologie di ordinamento di vettori e/o liste. Per come sono definiti, non è possibile la presenza di elementi uguali all'interno dello stesso albero e una visita simmetrica elenca gli elementi dell'albero in ordine crescente



Alcune funzioni di utilità

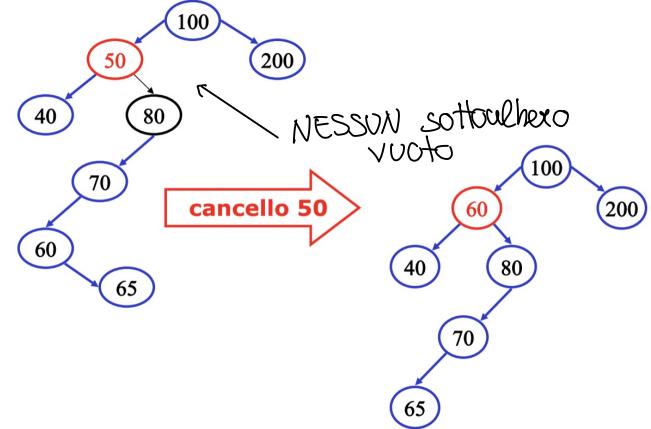
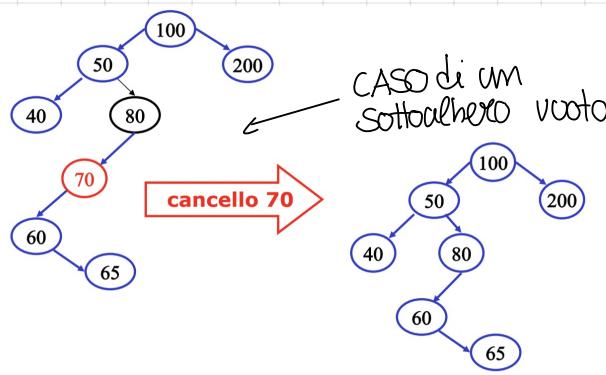
La funzione di ricerca prende come parametri l'albero e l'elemento ricercato e restituisce il puntatore al nodo ove esso è contenuto. Se l'elemento non è presente all'interno dell'albero viene restituito NULL. Grazie all'ordinamento dell'albero è possibile scartare uno dei due sottoalberi nel caso in cui l'elemento cercato sia maggiore o minore di quello della radice. Se il valore cercato è minore della radice infatti si può restringere la ricerca a sinistra, mentre se è maggiore la ricerca può essere ristretta a destra. La complessità nel caso migliore si ha quando l'albero è bilanciato o quasi bilanciato ed è logaritmica, in quanto la relazione di ricorrenza $T(n) = b + T(n/2)$. Nel caso peggiore, ovvero quando l'albero è in realtà una lista, si ha invece una complessità lineare dalla relazione di ricorrenza $T(n) = b + T(n-1)$. In entrambi i casi la complessità del caso base è costante.

```
Node* findNode (InfoType n, Node* tree){  
    if (!tree) return 0; —→ Albero vuoto  
    if (n==tree->label) return tree; —→ Elemento individuato  
    if (n>tree->label)  
        return findNode (n, tree->left); } Ricerca a sinistra nel caso di un elemento minore  
    else  
        return findNode (n, tree->right); } Ricerca a destra nel caso di un elemento maggiore  
}
```

Data la struttura ordinata degli alberi, è più complicato implementare una funzione che permetta di eliminare dei nodi conservando l'ordinamento dell'albero. Una possibile strategia è la seguente. Come per gli altri alberi binari, anche in questo caso bisogna ricercare inizialmente il nodo da eliminare. Una volta individuato, se presente, possono presentarsi due casi. Se il nodo ha un sottoalbero vuoto, il padre viene connesso all'unico sottoalbero non vuoto del nodo, cioè diventa sottoalbero sinistro o destro del nodo, a seconda che esso fosse il figlio sinistro o destro. Se entrambi i sottoalberi non sono vuoti invece viene ricercato l'elemento minimo in quello destro e viene cancellato, per poi sostituire la sua etichetta a quella del nodo iniziale

```
void deleteMin (Node* &tree, LabelType &m){  
    if (tree->left) deleteMin(tree->left, m);  
    else{  
        m=tree->label;  
        Node* a=tree;  
        tree=tree->right;  
        delete a;  
    }  
}  
  
void deleteNode (LabelType n, Node* &tree){  
    if (tree)  
        if (n<tree->label) deleteNode (n, tree->left);  
        else if (n>tree->label) deleteNode (n, tree->right);  
        else if (!tree->left){  
            Node* a=tree; tree=tree->right; delete a;  
        }  
        else if (!tree->right){  
            Node* a=tree; tree=tree->left; delete a;  
        }  
        else deleteMin (tree->right, tree->label);  
    }  
}
```

$O(\log n)$



La funzione di inserimento beneficia della struttura ordinata dell'albero e segue un procedimento analogo a quella di ricerca. La funzione parte dalla radice e sceglie di proseguire a sinistra o destra nel caso in cui l'elemento da inserire sia più piccolo o più grande di quello analizzato. La funzione inserisce l'elemento quando individua un albero vuoto. Se l'albero vuoto è il sottoalbero sinistro di p, l'elemento viene inserito a destra, mentre se è sottoalbero destro di p viene inserito a sinistra. Per definizione di albero generico di ricerca, il nodo non viene inserito se già presente all'interno dell'albero

```
void insertNode (LabelType n, Node* &tree){
    if (!tree){
        tree=new Node;
        tree->label=n; tree->left=tree->right=NULL;
        return;
    }
    if (n<tree->label) insertNode (n, tree->left);
    if (n>tree->label insertNode (n, tree->right);
}
```

$\hookrightarrow O(\log n)$

