

NICOLETTA DE FRANCESCO

**Corso “Fondamenti di Informatica II”
Modulo “Algoritmi e
strutture dati”**

a.a. 2014/2015

Classi di complessità

1.1 Tempo di esecuzione dei programmi

complessità di un algoritmo

funzione (sempre positiva) che associa alla **dimensione** del problema il **costo** della sua risoluzione in base alla **misura** scelta

$T_p(n)$ = Complessità con costo=**tempo** del programma P al variare di **n**:

```
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;  
}
```

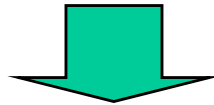
Se **tutti** i tempi
costanti sono uguali
a **1**:

$$T_{\max}(n) = 4n$$

1.1 tre programmi P, Q ed R

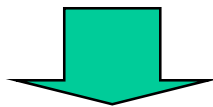
$$T_P(n) = 2n^2 \quad T_Q(n) = 100n \quad T_R(n) = 5n$$

Per $n \geq 50$, $T_Q(n) \leq T_P(n)$



$T_Q(n)$ ha complessità **minore o uguale** a $T_P(n)$
ma non vale il contrario

Per $n \geq 3$, $T_R(n) \leq T_P(n)$

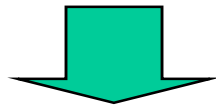


$T_R(n)$ ha complessità **minore o uguale** a $T_P(n)$
ma non vale il contrario

1.1 tre programmi P, Q ed R

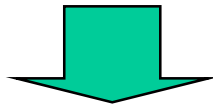
$$T_P(n) = 2n^2 \quad T_Q(n) = 100n \quad T_R(n) = 5n$$

Per ogni n , $T_R(n) \leq T_Q(n)$



$T_R(n)$ ha complessità minore o uguale a $T_Q(n)$

Per ogni n , $T_Q(n) \leq 20T_R(n)$



$T_Q(n)$ ha complessità **minore o uguale a** $T_R(n)$

$T_Q(n)$ e $T_R(n)$ hanno la **stessa** complessità

1.2 Complessità computazionale

$g(n)$ è di ordine $O(f(n))$ se esistono un intero

n_0 ed una costante $c > 0$ tali che

per ogni $n \geq n_0$: $g(n) \leq c f(n)$

1.2 Complessità computazionale

$T_Q(n)$ è $O(T_P(n))$ [$n_0=50, c=1$] oppure [$n_0=1, c= 50$]

$T_R(n)$ è $O(T_P(n))$ [$n_0=3, c=1$]

$T_R(n)$ è $O(T_Q(n))$ [$n_0=1, c=1$]

$T_Q(n)$ è $O(T_R(n))$ [$n_0=1, c=20$]

$T_P(n)$ non è $O(T_Q(n))$

$T_P(n)$ non è $O(T_R(n))$

$$T_P(n) = 2n^2$$

$$T_Q(n) = 100n$$

$$T_R(n) = 5n$$

Notazioni

$g(n)$ è di ordine $O(f(n))$

$g(n)$ è $O(f(n))$

$g(n) \in O(f(n))$

Una funzione $f(n)=\text{expr}$ si indica soltanto con **expr**

$f(n)= 3-n$



3-n

$f(n)=100n$ è $O(g(n)=5n)$



100n è $O(5n)$

1.2 esempi

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

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

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

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

$$n^2 \in O(1/100 n^2) \quad [n_0=1, c=100]$$

$$n^2 \in O(2^n) \quad [n_0=4, c=1]$$

1.2 Complessità computazionale

**$O(f(n))$ = insieme delle funzioni
di ordine $O(f(n))$**

$$O(n) = \{ \text{costante}, n, 4n, 300n, 100 + n, .. \}$$

$$O(n^2) = O(n) \cup \{ n^2, 300 n^2, n + n^2, ... \}$$

1.2 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 $O(g(n))$.

REGOLA DELLA PRODOTTO

Se $f(n)$ è $O(f_1(n))$ e $g(n)$ è $O(g_1(n))$, allora $f(n)g(n)$ è $O(f_1(n)g_1(n))$.

1.2 regole

- Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$,
allora $f(n)$ è $O(h(n))$
- per ogni costante k , k è $O(1)$
- per $m \leq p$, n^m è $O(n^p)$
- Un polinomio di grado m è $O(n^m)$

1.2 esempi

- $2n + 3n + 2$ è $O(n)$
- $(n+1)^2$ è $O(n^2)$
- $2n + 10n^2$ è $O(n^2)$

1.2 due funzioni

$$f(n) = \begin{cases} n^3 & \text{se } n \text{ e' pari} \\ n^2 & \text{se } n \text{ e' dispari} \end{cases}$$
$$g(n) = \begin{cases} n^2 & \text{se } n \text{ e' primo} \\ n^3 & \text{se } n \text{ e' composto} \end{cases}$$

$$f(n) \text{ è } O(g(n)) \quad n_0=3, \quad c=1$$

**non vale il contrario: esistono infiniti numeri
composti dispari**

1.2 funzioni incommensurabili

$$f(n) = \begin{cases} n & \text{se } n \text{ e' pari} \\ n^2 & \text{se } n \text{ e' dispari} \end{cases}$$

$$g(n) = \begin{cases} n^2 & \text{se } n \text{ e' pari} \\ n & \text{se } n \text{ e' dispari} \end{cases}$$

1.3 Classi di Complessità

$O(1)$	costante	
$O(\log n)$	logaritmica	$(\log_a n = \log_b n \log_a b)$
$O(n)$	lineare	
$O(n \log n)$	nlogn	
$O(n^2)$	quadratica	
$O(n^3)$	cubica	
..		
$O(n^p)$	polinomiale	
$O(2^n)$	esponenziale	
$O(n^n)$	esponenziale	

1.3 teorema

per ogni k , $n^k \in O(a^n)$, per ogni $a > 1$

Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale

Complessità dei programmi iterativi

2.1 Programmi iterativi

C: costrutti del linguaggio -> Classi di complessità

$$C [V] = C [I] = O(1)$$

$$C [E1 \text{ op } E2] = C [E1] + C [E2]$$

$$C [I[E]] = C [E]$$

$$C [I=E;] = C [E]$$

$$C [I[E1] =E2;] = C [E1] + C [E2]$$

2.1 Programmi iterativi

$C [\text{return } E;] = C [E]$

$C [\text{if } (E) C] = C [E] + C [C]$

$C [\text{if } (E) C1 \text{ else } C2] =$
 $C [E] + C [C1] + C [C2]$

2.1 Programmi iterativi

$C[\text{for} (E1; E2; E3) C] =$
 $C[E1] + C[E2] +$
 $(C[C] + C[E2] + C[E3]) O(g(n))$

$g(n)$: numero di iterazioni

$C[\text{while} (E) C] =$
 $C[E] + (C[C] + C[E]) O(g(n))$

2.1 Programmi iterativi

$$C [\{ C1 \dots Cn \}] = C [C1] + \dots + C [Cn]$$

$$C [F (E1, \dots En)] =$$

$$C [E1] + \dots + C [En] + C [\{ C \dots C \}]$$

$$\text{se } T F (T1 I1, \dots, Tn In) \{ C \dots C \}$$

2.2 Selection sort

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

```
void selectionSort(int A[ ], int n) {  
    O(n2) for (int i=0; i< n-1; i++) {  
        O(1) int min= i;  
        O(n) for (int j=i+1; j< n; j++)  
            O(1) if (A[ j ] < A[min]) min=j;  
        O(1) exchange(A[i], A[min]);  
    }  
}
```

2.2 Bubblesort

```
void bubbleSort(int A[], int n) {  
   $O(n^2)$  for (int i=0; i < n-1; i++)  
     $O(n)$       for (int j=n-1; j >= i+1; j--)  
       $O(1)$       if (A[j] < A[j-1]) exchange(A[j], A[j-1]);  
}
```

$O(n^2)$ numero di scambi = **$O(n^2)$**

con selectionSort numero di scambi = **$O(n)$**

2.1 Esempio (I)

```
int f (int x){  
    return x;  
}
```

risultato: $O(n)$

complessità: $O(1)$

```
int h (int x){  
    return x*x;  
}
```

risultato: $O(n^2)$

complessità: $O(1)$

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

risultato: $O(n)$

complessità: $O(n)$

2.1 Esempio (II)

```
void g (int n){    // n >= 0
    for (int i=1; i <= f(n); i++)
        cout << f(n);
}
```

complessità: $O(n)$

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

complessità: $O(n^2)$

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

complessità: $O(n^2)$

2.1 Esempio (III)

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

complessità: $O(n)$

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

complessità: $O(n^2)$

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

complessità: $O(n)$

Moltiplicazione fra matrici

```
void matrixMult (int A[N] [N], int B[N] [N], int C [N] [N]) {
```

```
     $O(n^3)$  for (int i=0; i < N; i++)
```

```
         $O(n^2)$  for (int j=0; j < N; j++) {
```

```
             $O(1)$  C[ i ] [ j ]=0;
```

```
             $O(n)$  for (int k=0; k < N; k++)
```

```
                 $O(1)$  C[ i ] [ j ]+=A[ i ] [ k ] * B[ k ] [ j ];
```

```
        }
```

```
    }
```

$O(n^3)$

2.1 Ricerca lineare e div2

```
int linearSearch (int A [], int n, int x) {  
    for (int i=0; i<n; i++)  
        if (A[ i ] == x) return 1;  
    return 0;  
}
```

$O(n)$

```
int div2(int n) {  
    int i=0;  
    while (n > 1) {  
        n=n/2;  
        i++;  
    }  
    return i;  
}
```

$O(\log n)$

Complessità dei programmi ricorsivi

3 Programmi ricorsivi : definizioni iterative e induttive

Fattoriale di un numero naturale : $n!$

$$0! = 1$$

$$n! = 1 \times 2 \times \dots \times n \text{ se } n > 0 \quad \text{definizione iterativa}$$

$$0! = 1$$

$$n! = n \times (n-1)! \text{ se } n > 0 \quad \text{definizione induttiva (o ricorsiva)}$$

fattoriale: algoritmo iterativo

$0! = 1$

$n! = 1 \times 2 \times \dots \times n$

```
int fact(int n) {  
    if (n == 0) return 1;  
    int a=1;  
    for (int i=1; i<=n; i++) a=a*i;  
    return a;  
}
```


fattoriale: algoritmo ricorsivo

$0! = 1$

$n! = n * (n-1)! \text{ se } n > 0$

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

Programmi ricorsivi: moltiplicazione

mult (0, y) = 0

mult (n,y)= y + mult (n-1,y) se n>0

```
int mult(int x, int y) {  
    if (x == 0) return 0;  
    return y + mult(x-1,y);  
}
```

Programmi ricorsivi : pari e massimo comun divisore

```
int pari(int x) {  
    if (x == 0) return 1;  
    if (x == 1) return 0;  
    return pari(x-2);  
}
```

```
int mcd(int x, int y) {  
    if (x == y) return x;  
    if (x < y) return mcd(x, y-x);  
    return mcd(x-y, y);  
}
```

Regole da rispettare

Regola 1

**individuare i casi base in cui la funzione è definita
immediatamente**

Regola 2

**effettuare le chiamate ricorsive su un insieme più
"piccolo" di dati**

Regola 3

**fare in modo che alla fine di ogni sequenza di chiamate
ricorsive, si ricada in uno dei casi base**

Regole da rispettare

```
int pari_errata(int x) {  
    if (x == 0) return 1;  
    return pari_errata(x-2);  
}
```

```
int mcd_errata(int x, int y) {  
    if (x == y) return x;  
    if (x < y) return mcd_errata(x, y-x);  
    return mcd_errata(x, y);  
}
```

4 programmi ricorsivi su liste

definizione di **LISTA**

- **NULL** (sequenza vuota) è una **LISTA**
- un elemento seguito da una **LISTA** è una **LISTA**

```
struct Elem {  
    InfoType inf;  
    Elem* next;  
};
```

4 programmi ricorsivi su liste

```
int length(Elem* p) {  
    if (p == NULL) return 0;           // (! p)  
    return 1+length(p->next);  
}  
  
int howMany(Elem* p, int x) {  
    if (p == NULL) return 0;  
    return (p->inf == x)+howMany(p->next, x);  
}
```

4 programmi ricorsivi su liste

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

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


4 programmi ricorsivi su liste

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

4 programmi ricorsivi su liste

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

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

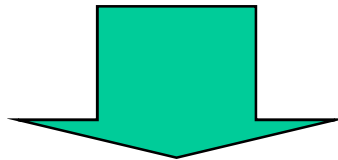
Induzione naturale

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 $(n+1)$



P vale per tutti i naturali

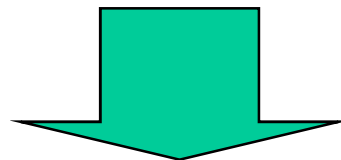
Induzione completa

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 $(n+1)$



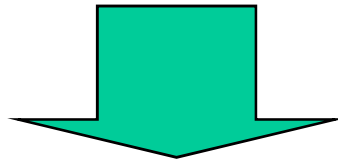
P vale per tutti i naturali

Insieme ordinato S

Base. P vale per i minimali di S

Passo induttivo. per ogni elemento E di S è vero che:

Se P vale per ogni **elemento minore** di E allora P vale per E



P vale per S

4 Complessità dei programmi ricorsivi

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

$$T(0) = a$$

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

Relazione di ricorrenza

4 soluzione

$$\begin{aligned}T(0) &= a \\ T(n) &= b + T(n-1)\end{aligned}$$

$$T(0) = a$$

$$T(1) = b + a$$

$$T(2) = b + b + a = 2b + a$$

$$T(3) = 3b + a$$

.

.

$$T(n) = nb + a$$

$T(n)$ è $O(n)$

4 selection sort ricorsiva

```
void r_selectionSort (int* A, int m, int i=0) {  
    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)  
}
```

$$T(1) = a$$

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

4 soluzione

$$T(1) = a$$

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

$$T(1) = a$$

$$T(2) = 2b + a$$

$$T(3) = 3b + 2b + a$$

.

.

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

$T(n)$ è $O(n^2)$

4.3 QuickSort

```
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);  
}
```

4.3 QuickSort

quicksort([3,5,2,1,1], 0, 4)

quicksort([1,1,2,5,3], 0, 1)

quicksort([1,1,2,5,3], 3, 4)

4.3 Quicksort

$$T(1) = a$$

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

Se $k=1$:

$$T(1) = a$$

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

$O(n^2)$

Se $k=n/2$:

$$T(1) = a$$

$$T(n) = bn + 2T(n/2)$$

4 soluzione

$$T(1) = a$$

$$T(n) = bn + 2T(n/2)$$

$$T(1) = a$$

$$T(2) = 2b + 2a$$

$$T(4) = 4b + 4b + 4a$$

$$T(8) = 8b + 8b + 8b + 8a = 3(8b) + 8a$$

$$T(16) = 16b + 16b + 16b + + 16b + 16a = 4(16b) + 16a$$

.

.

$T(n)$ è $O(n \log n)$

$$T(n) = (n \log n) b + na$$

Ricerca in un insieme

```
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);  
}
```

$$T(0) = a$$

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

$$O(n)$$

Ricerca in un insieme

```
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);  
}
```

$$T(0) = a$$

$$T(n) = b + T(n/2)$$

soluzione

$$T(0) = a$$

$$T(n) = b + T(n/2)$$

$$T(0) = a$$

$$T(1) = b + a$$

$$T(2) = b + b + a$$

$$T(4) = b + b + b + a$$

.

.

$$T(n) = (\log n + 1) b + a$$

$T(n)$ è $O(\log n)$

Ricerca in un insieme

```
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) || Search(A, x, k+1, j);  
}
```

$$T(0) = a$$

$$T(n) = b + 2T(n/2)$$

soluzione

$$T(0) = a$$

$$T(n) = b + 2T(n/2)$$

$$T(0) = a$$

$$T(1) = b + 2a$$

$$T(2) = b + 2b + 4a = 3b + 4a$$

$$T(4) = b + 6b + 8a = 7b + 8a$$

.

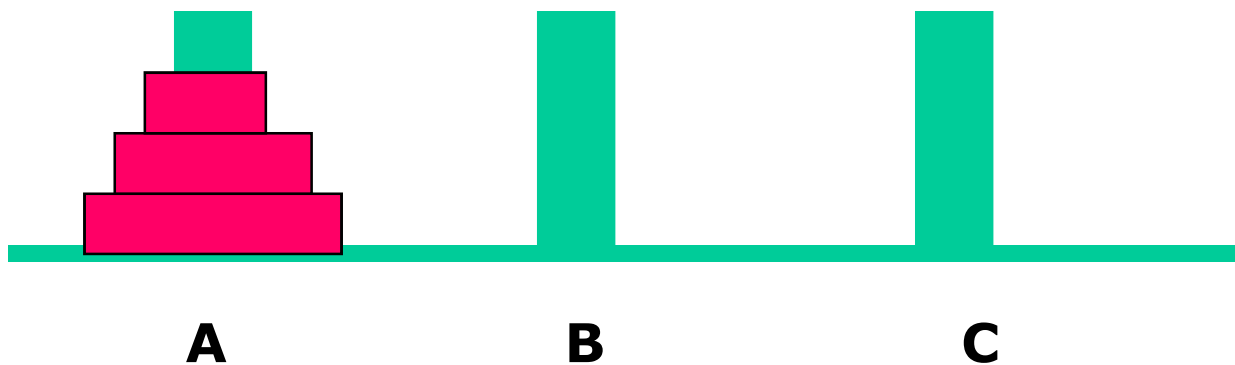
.

$$T(n) = (2^n - 1)b + 2^n a$$

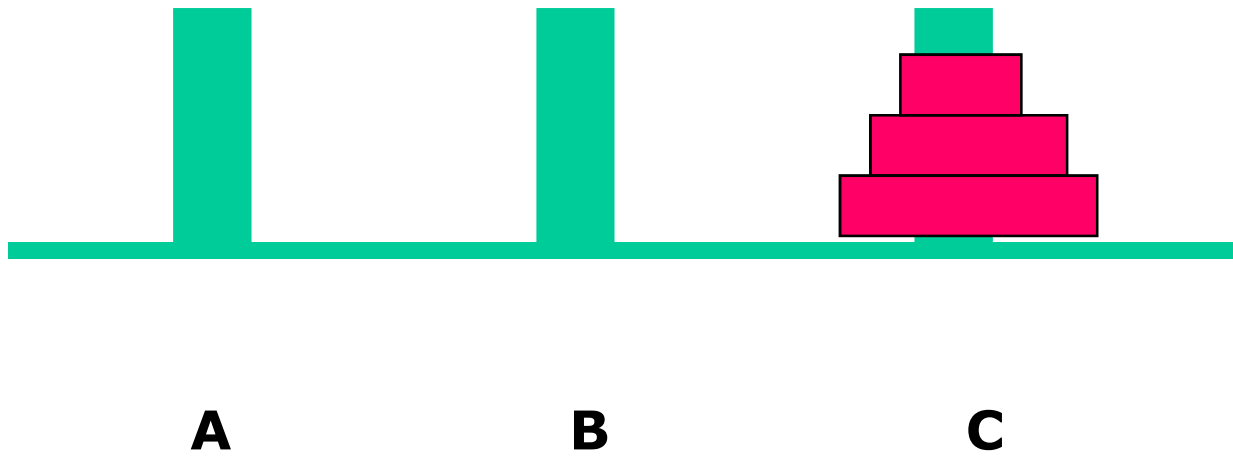
$T(n)$ è $O(n)$

Torre di Hanoi

- **3 paletti**
- **bisogna spostare la torre di n cerchi dal paletto sorgente A a quello destinatario C usando un paletto ausiliario B**
- **Un cerchio alla volta**
- **Mai un cerchio sopra uno piu' piccolo**



Torre di Hanoi



Torre di Hanoi

void trasferisci una torre di n cerchi da A a C {

Se $n=1$ sposta il cerchio dal A a C;

altrimenti

**{ trasferisci la torre degli $n-1$ cerchi più piccoli da A a B
usando C come paletto ausiliario;**

sposta il cerchio più grande dal A a C;

**trasferisci la torre degli $n-1$ cerchi più piccoli da B a C
usando A come paletto ausiliario;**

} }

Torre di Hanoi

```
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);
    }
}
```

$$T(1) = a$$

$$T(n) = b + 2T(n-1)$$

hanoi(3, A, B, C)

hanoi(2, A, C, B)

hanoi(1, A, B, C)

sposta(A, C);

sposta(A, B);

hanoi(1, C, A, B)

sposta(C, B);

sposta(A,C);

hanoi(2, B, A, C)

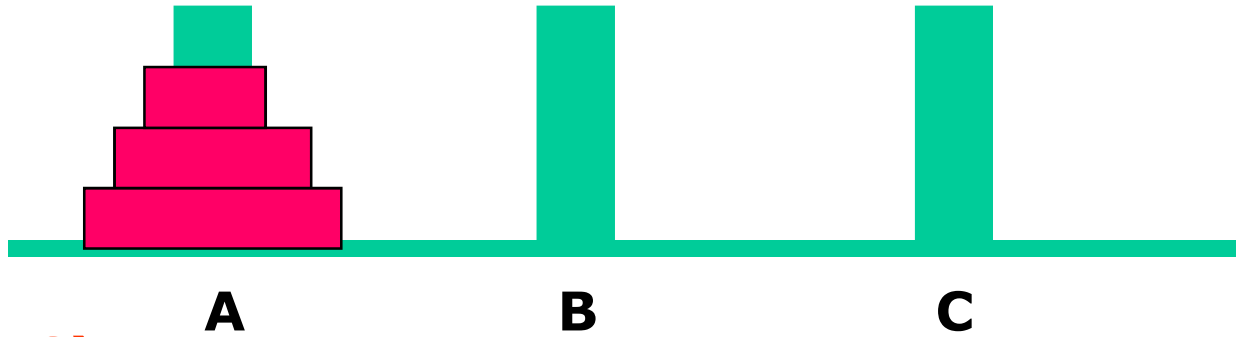
hanoi(1, B, C, A)

sposta(B, A);

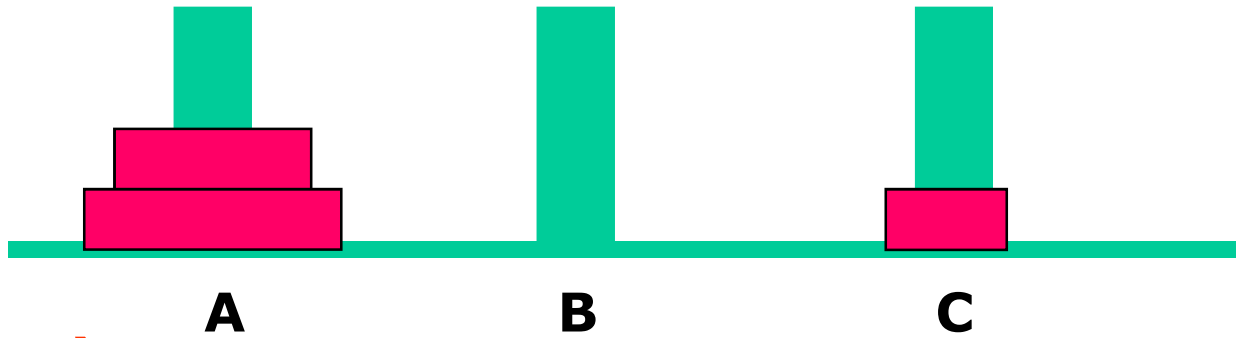
sposta(B, C);

hanoi(1, A, B,C)

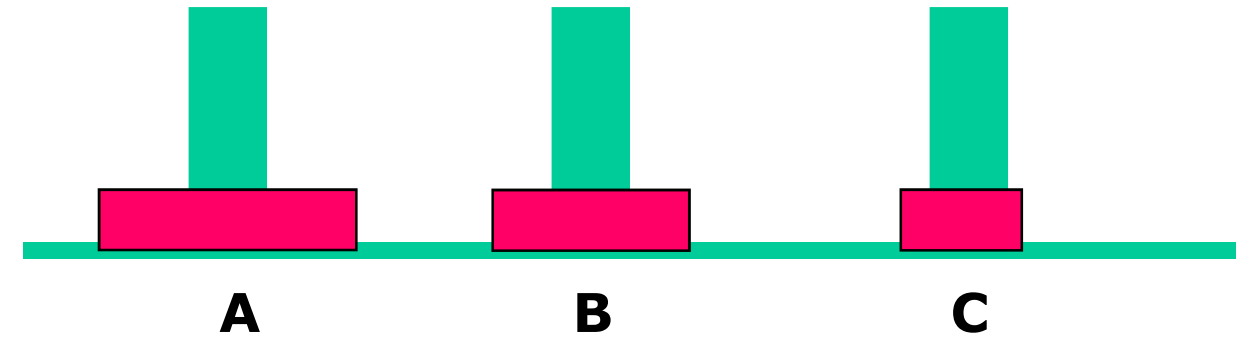
sposta(A,C);



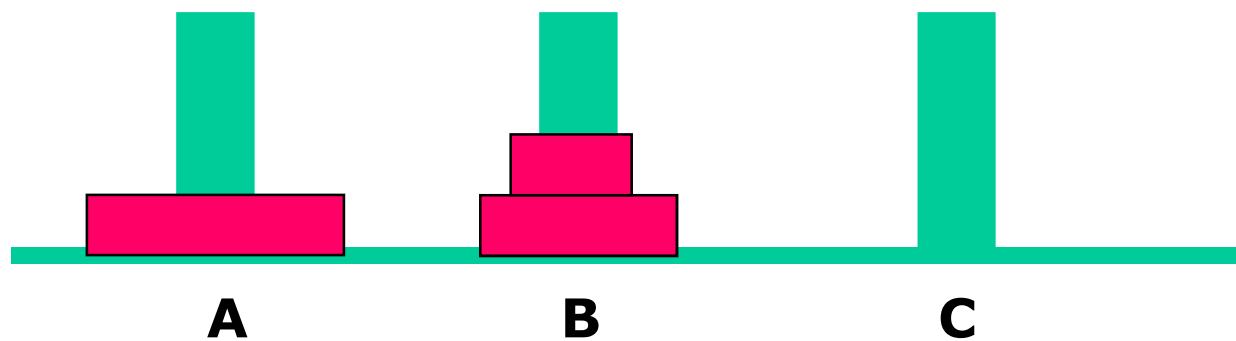
sposta(A, C);



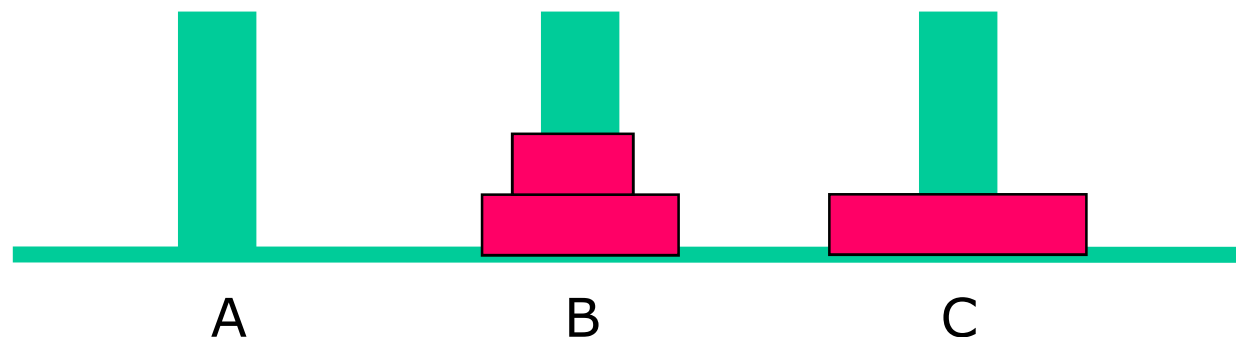
sposta(A, B);



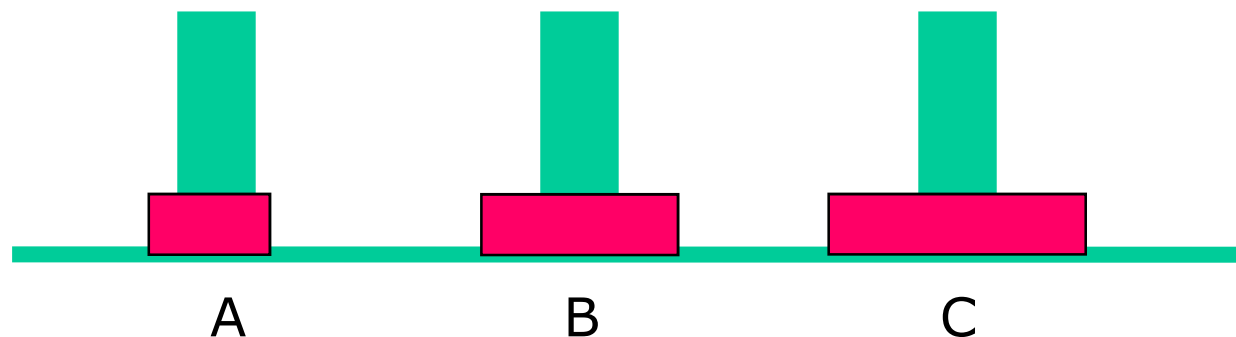
sposta(C, B);



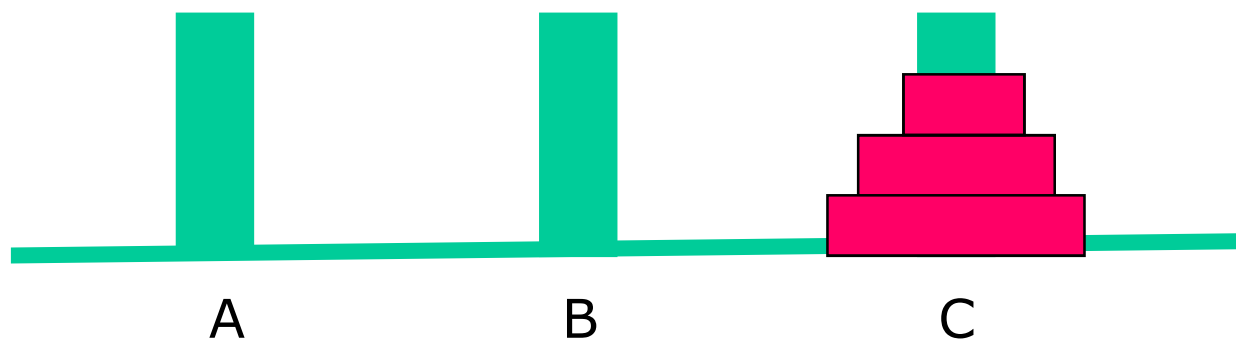
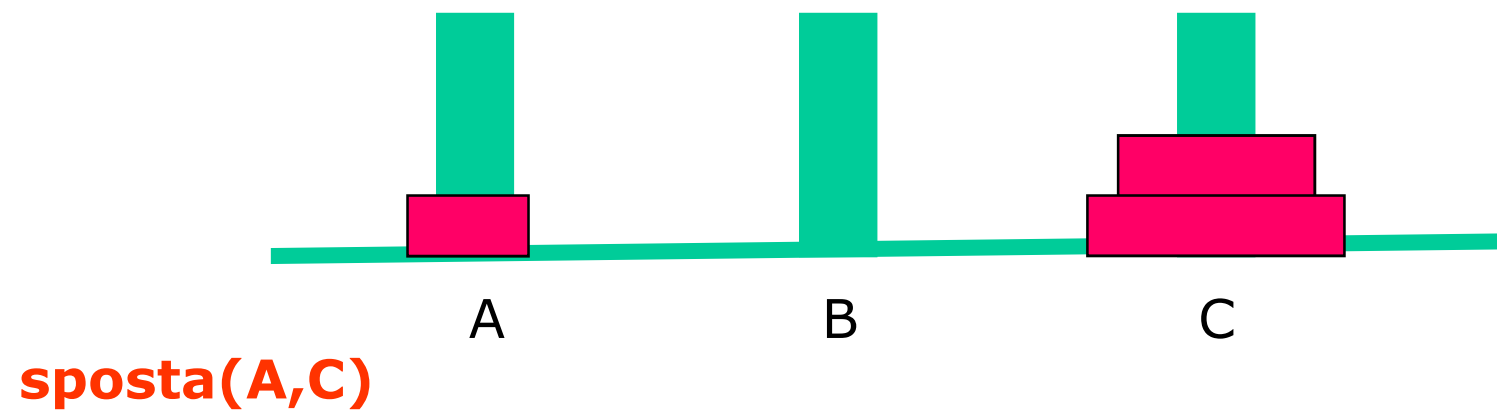
sposta(A,C)



sposta(B,A)



sposta(B,C)



soluzione

$$T(1) = a$$

$$T(n) = b + 2T(n-1)$$

$$T(1) = a$$

$$T(2) = b + 2a$$

$$T(3) = b + 2b + 4a = 3b + 4a$$

$$T(4) = 7b + 8a$$

.

.

$$T(n) = (2^{n-1} - 1)b + 2^{n-1}a$$

$$T(n) \text{ è } O(2^n)$$

5.1 Metodo divide et impera

```
void dividetimpera( S ) {  
  
    if ( |S| <= m )  
        <risolvi direttamente il problema>;  
    else {  
        <dividi S in b sottoinsiemi  $S_1.. S_b$ >;  
        dividetimpera( $S_{i1}$  );  
        ...  
        dividetimpera( $S_{ia}$  );  
  
        <combina i risultati ottenuti>;  
    }  
}
```

5.1 Metodo divide et impera

$$T(0) = d$$

$$T(n) = b + T(n/2)$$

$$O(\log n)$$

$$T(0) = d$$

$$T(n) = b + 2T(n/2)$$

$$O(n)$$

$$T(0) = d$$

$$T(n) = bn + 2T(n/2)$$

$$O(n \log n)$$

5.1 Metodo divide et impera

$$T(n) = d \quad \text{se } n \leq m$$

$$T(n) = hn^k + aT(n/b) \quad \text{se } n > m$$

$$h > 0$$

$$T(n) \in O(n^k) \quad \text{se } a < b^k$$

$$T(n) \in O(n^k \log n) \quad \text{se } a = b^k$$

$$T(n) \in O(n^{\log_b a}) \quad \text{se } a > b^k$$

12.1 Moltiplicazione veloce

$$A = A_s 10^{n/2} + A_d$$

$$B = B_s 10^{n/2} + B_d$$

$$A * B = A_s B_s 10^n + (A_s * B_d + A_d * B_s) 10^{n/2} + A_d * B_d$$

$$(A_s + A_d)(B_s + B_d) = A_s * B_d + A_d * B_s + A_s * B_s + A_d * B_d$$

$$A_s * B_d + A_d * B_s = (A_s + A_d) * (B_s + B_d) - A_s * B_s - A_d * B_d$$

$$AB = A_s * B_s 10^n + ((A_s + A_d) * (B_s + B_d) - A_s * B_s - A_d * B_d) 10^{n/2} + A_d * B_d$$

12.1 Moltiplicazione veloce

$$T(1) = d$$

$$T(n) = bn + 3T(n/2)$$

$$T(n) \in O(n^{\log_2 3})$$

$$T(n) \in O(n^{1.59})$$

5.2 Relazioni lineari

$$T(0) = d$$

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

$$O(n)$$

$$T(1) = a$$

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

$$O(n^2)$$

$$T(0) = d$$

$$T(n) = b + 2T(n-1)$$

$$O(2^n)$$

5.2 Relazioni lineari

$$T(0) = d$$

$$T(n) = bn^k + a_1T(n-1) + a_2T(n-2) + a_rT(n-r)$$

polinomiale solo se esiste al più un solo $a_i = 1$ e gli altri a_i sono tutti 0 (c'è una sola chiamata ricorsiva)

5.2 Soluzione di una classe di relazioni lineari

$$T(0) = d$$

$$T(n) = bn^k + T(n-1)$$

$$b > 0$$

$$T(n) \in O(n^{k+1})$$

5.2 Numeri di Fibonacci

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

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

$$T(0) = T(1) = d$$

$$T(n) = b + T(n-1) + T(n-2)$$

$$T(n) \in O(2^n)$$

5.2 Numeri di Fibonacci

```
int fibonacci(int n) {  
    int k; int j=0; int f=1;  
    for (int i=1; i<=n; i++) {  
        k=j; j=f; f=k+j;  
    }  
    return j;  
}
```

$T(n) \in O(n)$

5.2 Numeri di Fibonacci

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

$$T(0) = d$$

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

$$T(n) \in O(n)$$

Mergesort

```
void mergeSort( sequenza S ) {  
  if ( |S| <= 1 )  
    return;  
  else {  
    < dividi S in 2 sottosequenze S1 e S2 di uguale  
        lunghezza >;  
    mergeSort(S1 );  
    mergeSort(S2 );  
    < fondi S1 e S2 >;  
  }  
}
```

Mergesort

```
void mergeSort(Elem*& s1) {  
    if (s1 == NULL || s1->next == NULL) return;  
    Elem* s2 = NULL;  
  
    split (s1, s2);  
  
    mergeSort (s1);  
  
    mergeSort (s2);  
  
    merge (s1, s2);  
}
```

$$T(0) = T(1) = d$$

$$T(n) = bn + 2T(n/2)$$

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

Mergesort

```
void split (Elem* & s1, Elem* & s2) {  
    if (s1 == NULL || s1->next == NULL)  
        return;  
    Elem* p = s1->next;  
    s1->next = p->next;  
    p->next = s2;  
    s2 = p;  
    split (s1->next, s2);  
}
```

$$T(0) = T(1) = d$$

$$T(n) = b + T(n-2)$$

$$T(n) \in O(n)$$

Mergesort

```
void merge (Elem* & s1, Elem* s2) {  
    if (s2 == NULL)  
        return;  
    if (s1 == NULL) {  
        s1 = s2;  
        return;  
    }  
    if (s1->inf <= s2->inf)  
        merge (s1-> next, s2);  
  
    else {  
        merge (s2-> next, s1);  
        s1 = s2;  
    }  
}
```

$$T(0) = d$$

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

$$T(n) \in O(n)$$

mergeSort([2,1,3,5])

dividi([2,1,3,5])

mergeSort([2,3])

dividi([2,3])

mergeSort([2])

mergeSort([3])

fondi([2] , [3])

mergeSort([5,1])

dividi([5,1])

mergeSort([5])

mergeSort([1])

fondi([5] , [1])

fondi([2,3], [1,5])

Esempio mergesort

[2]

[3]

[2,3]

[5]

[1]

[1,5]

[1,2,3,5]

Alberi

6. Alberi binari

- **NULL** è un albero binario;
- un nodo **p** più **due alberi binari** **Bs** e **Bd** forma un albero binario

p è **radice**

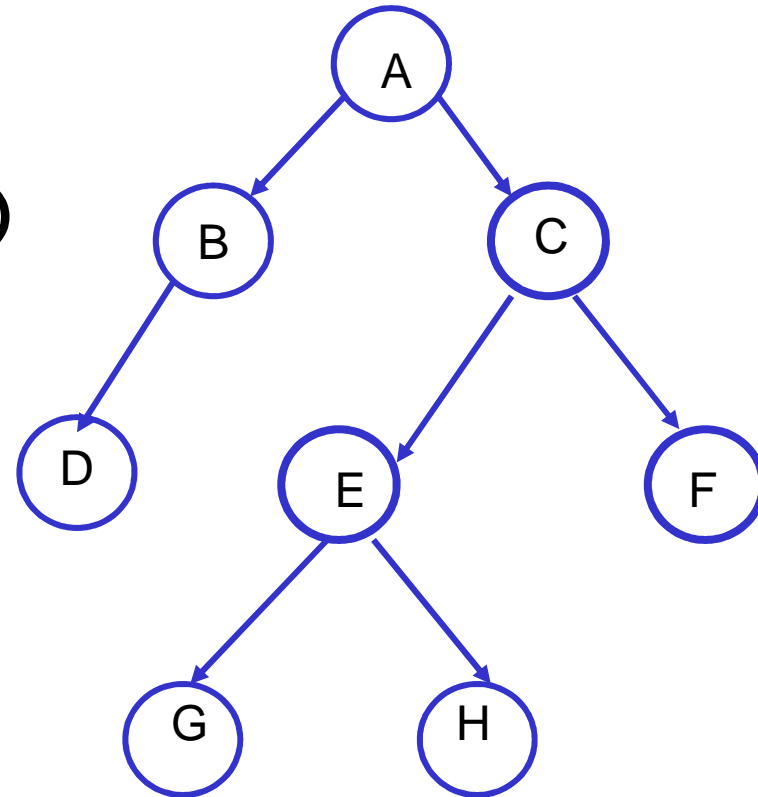
Bs è il **sottoalbero sinistro** di **p**

Bd il **sottoalbero destro** di **p**

alberi **etichettati**

6. Alberi binari

- **padre**
- **figlio sinistro (figlio destro)**
- **antecedente**
- **foglia**
- **discendente**
- **livello di un nodo**
- **livello dell'albero**



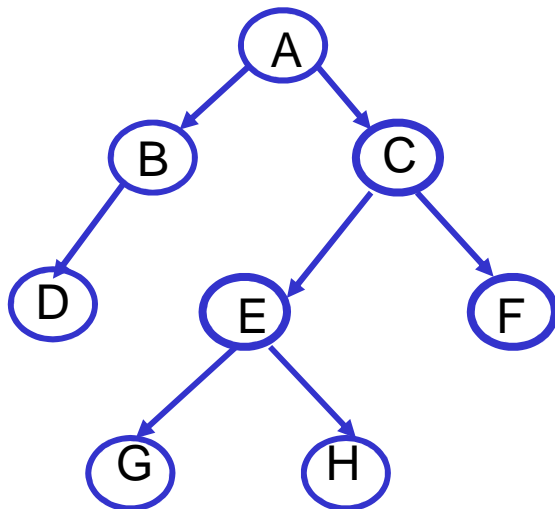
6. Ricorsione su alberi binari

caso base **albero vuoto (NULL)**

caso ricorsivo **radice + due sottoalberi**

6. Visita anticipata (preorder)

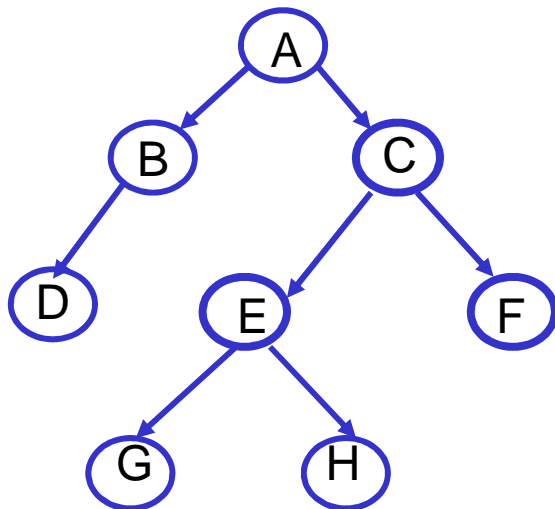
```
void preOrder ( albero ) {  
    se l'albero binario non e' vuoto {  
        esamina la radice;  
        preOrder ( sottoalbero sinistro);  
        preOrder ( sottoalbero destro);  
    }  
}
```



A B D C E G H F

6. Visita differita (postorder)

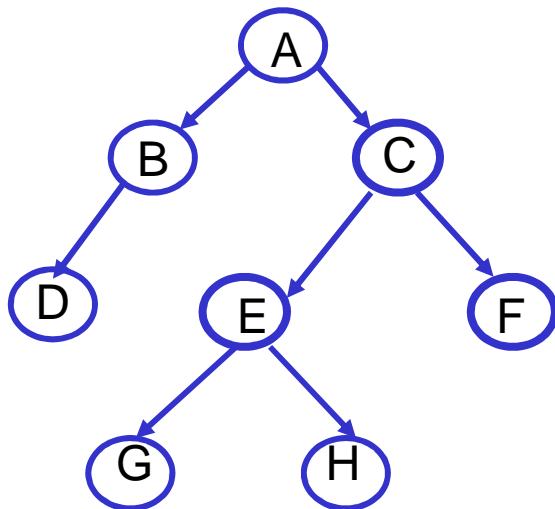
```
void postOrder ( albero ) {  
    se l'albero binario non e' vuoto {  
        postOrder ( sottoalbero sinistro);  
        postOrder ( sottoalbero destro);  
        esamina la radice;  
    }  
}
```



D B G H E F C A

6. Visita simmetrica (inorder)

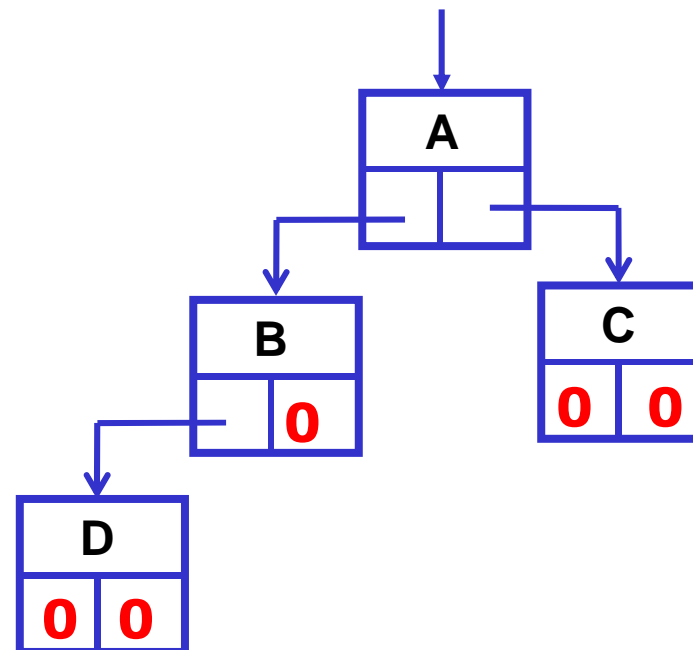
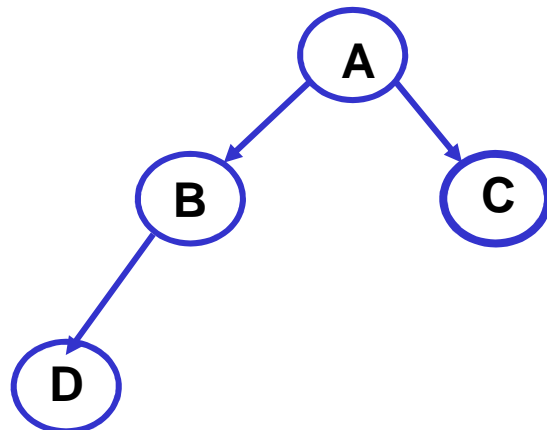
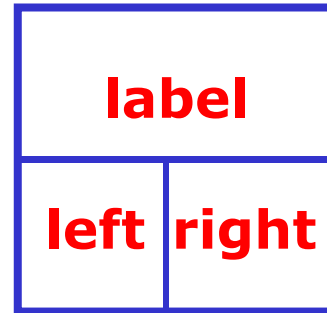
```
void inOrder ( albero ) {  
    se l'albero binario non e' vuoto {  
        inOrder ( sottoalbero sinistro);  
        esamina la radice;  
        inOrder ( sottoalbero destro);  
    }  
}
```



D B A G E H C F

6. Memorizzazione in lista multipla

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



6. visite in C++

```
void preOrder(Node* tree) {  
    if (tree) {  
        <esamina tree->label>;  
        preOrder(tree->left);  
        preOrder(tree->right);  
    }  
}
```

```
void preOrder(Node* tree) {  
    if (tree) {  
        cout << tree->label;  
        preOrder(tree->left);  
        preOrder(tree->right);  
    }  
}
```

6. Visite in C++

```
void postOrder(Node* tree) {  
    if (tree) {  
        postOrder(tree->left);  
        postOrder(tree->right);  
        <esamina tree->label>;  
    }  
}
```

```
void inOrder(Node* tree) {  
    if (tree) {  
        inOrder(tree->left);  
        <esamina tree->label>;  
        inOrder(tree->right);  
    }  
}
```

6. Complessità delle visite

Complessità in funzione del numero di nodi:

$$T(0) = a$$

$$T(n) = b + T(n_s) + T(n_d) \quad \text{con } n_s + n_d = n - 1 \quad n > 0$$

Caso particolare:

$$T(0) = a$$

$$T(n) = b + 2T((n-1)/2)$$

$$T(n) \in O(n)$$

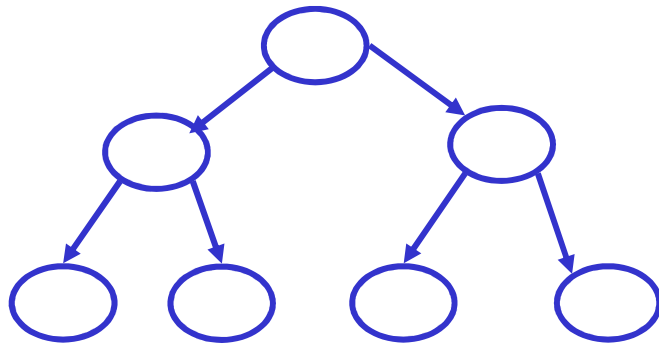
Visita iterativa

```
void preOrder(Node* tree) {  
    stack<Node*> miapila(100);  
    for (;;) {  
        while (tree) {  
            <esamina tree->label>;  
            miapila.push(tree);  
            tree=tree->left;  
        }  
        if (miapila.empty()) return;  
        tree=miapila.pop()->right;  
    } }  
}
```

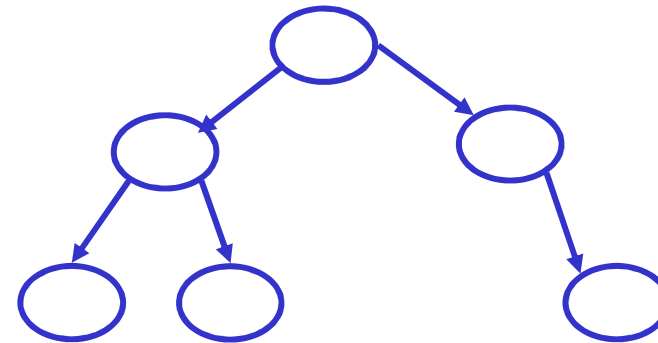
6. Alberi binari bilanciati

ALBERO BINARIO BILANCIATO

i nodi di tutti i livelli tranne quelli dell'ultimo hanno due figli



bilanciato



non bilanciato

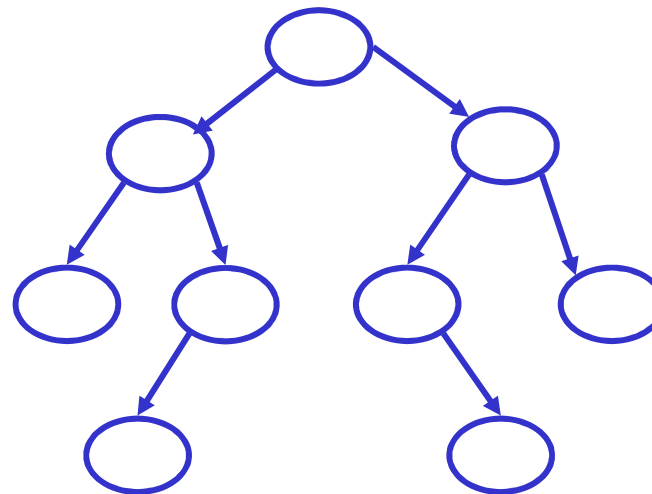
Un albero binario bilanciato con livello k ha $2^{(k+1)} - 1$ nodi e 2^k foglie

6. Alberi binari

ALBERO BINARIO QUASI BILANCIATO

fino al penultimo livello è un albero bilanciato

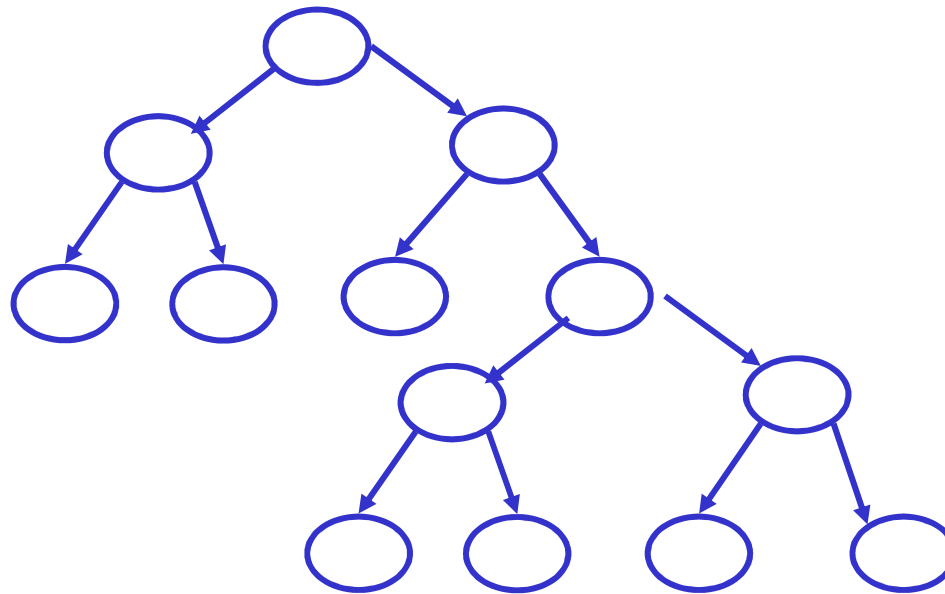
(un albero bilanciato è anche quasi bilanciato)



6. Alberi binari

ALBERO PIENAMENTE BINARIO

Tutti i nodi tranne le foglie hanno 2 figli



Un albero binario pienamente binario ha tanti nodi interni quante sono le foglie meno 1

6. Complessità delle visite nel numero dei livelli

Complessità in funzione dei livelli (se l'albero è bilanciato):

$$T(0) = a$$

$$T(k) = b + 2T(k-1)$$

$$T(k) \in O(2^k)$$

6. Alberi binari: conta i nodi e le foglie

conta i nodi

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

conta le foglie

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

$$T(n) \in O(n)$$

6. Alberi binari: cerca un'etichetta

ritorna il puntatore al nodo che contiene l'etichetta **n**. Se l'etichetta non compare nell'albero ritorna NULL. Se più nodi contengono **n**, ritorna il primo nodo che si incontra facendo la visita anticipata

```
Node* findNode (Infotype n, Node*tree) {  
    if (!tree) return NULL;           //albero vuoto: l'etichetta non c'è  
    if (tree->label==n)                // trovata:ritorna il puntatore  
        return tree;  
    Node* a=findNode(n, tree->left);  // cerca a sinistra  
    if (a) return a;                  // se trovata ritorna il puntatore  
    else return findNode(n, tree->right); // cerca a destra  
}
```

6. Alberi binari: cancella tutto l'albero

```
void delTree(Node* &tree) {  
    if (tree) {  
        delTree(tree->left);  
        delTree(tree->right);  
        delete tree;  
        tree=NULL;    }  
}
```

alla fine il puntatore deve essere NULL

6. Alberi binari: inserisci un nodo

inserisce un nodo (**son**) come figlio di **father**, sinistro se **c='l'**, destro se **c='r'**. Ritorna 1 se l'operazione ha successo, 0 altrimenti. Se l'albero è vuoto, inserisce il nodo come radice

```
int insertNode
(Node* & tree, InfoType son, InfoType father, char c){
    if (!tree) {                // albero vuoto
        tree=new Node;
        tree ->label=son;
        tree ->left = tree ->right = NULL;
        return 1;
    }
}
```

6. Alberi binari: inserisci un nodo (cont.)

```
Node* a=findNode(father,tree);    //cerca father
if (!a) return 0;                 //father non c'è
if (c=='l' && !a->left) {         //inserisci come figlio sinistro
    a->left=new Node;
    a->left->label=son;
    a->left->left =a->left->right=NULL;
    return 1;
}
```


6. Alberi binari: inserisci un nodo (cont.)

```
if (c=='r' && !a->right) {           //inserisci come figlio destro  
    a->right=new Node;  
    a->right->label=son;  
    a->right->left = a->right->right = NULL;  
    return 1;  
}  
return 0;                           //inserimento impossibile  
}
```

6. Class BinTree

```
template<class InfoType>
class BinTree {
    struct Node {
        InfoType label;
        Node *left, *right;
    };
    Node *root;
    Node* findNode(InfoType, Node*);
    void preOrder(Node*);
    void inOrder(Node*);
    void postOrder(Node*);
    void delTree(Node*&);
    int insertNode(Node*&, InfoType, InfoType, char)
```

6. Class BinTree

public:

```
BinTree() { root = NULL; };
```

```
~BinTree(){ delTree(root); };
```

```
int find(InfoType x) { return findNode(x, root); };
```

```
void pre() { preOrder(root); };
```

```
void post() { postOrder(root); };
```

```
void in() { inOrder(root); };
```

```
int insert(InfoType son, InfoType father, char c) {  
    insertNode(root,son, father,c);  
};
```

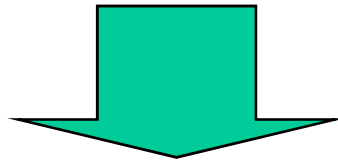
```
};
```

L'ordinamento e' basato sulla struttura

Base. P vale l'albero vuoto

Passo induttivo. Per un albero non vuoto B è vero
che:

Se P vale per B_s e per B_d allora vale per B



P vale per B

esempio

P: in ogni albero binario il numero dei sottoalberi vuoti è uguale al numero dei nodi +1

Base. Vero per l'albero vuoto: $Nodi=0$, $Vuoti=1$

Passo induttivo.

Ipotesi: $Vuoti_s = Nodi_s + 1$, $Vuoti_d = Nodi_d + 1$

Tesi: $Vuoti_B = Nodi_B + 1$

Dim. $Nodi_B = Nodi_s + Nodi_d + 1$

$Vuoti_B = Vuoti_s + Vuoti_d$

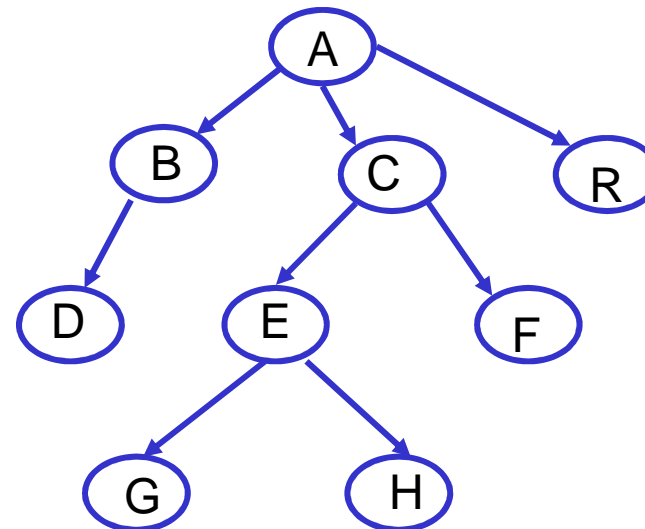
Usandi l'ip. induttiva:

$Vuoti_B = Nodi_s + 1 + Nodi_d + 1 = Nodi_B + 1$

7.1 Alberi generici: definizione

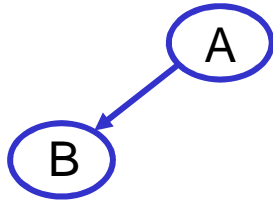
- un nodo p è un **albero**
- un nodo + una **sequenza di alberi** $A_1 \dots A_n$ è un albero

- radice
- padre
- i -esimo sottoalbero
- i -esimo figlio
- livello

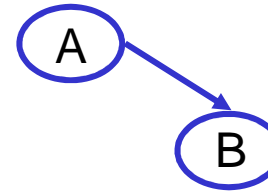


7.1 Alberi generici: differenza con alberi binari

alberi binari



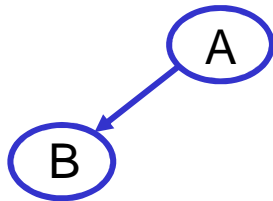
diverso da



sottoalbero **destro** vuoto

sottoalbero **sinistro** vuoto

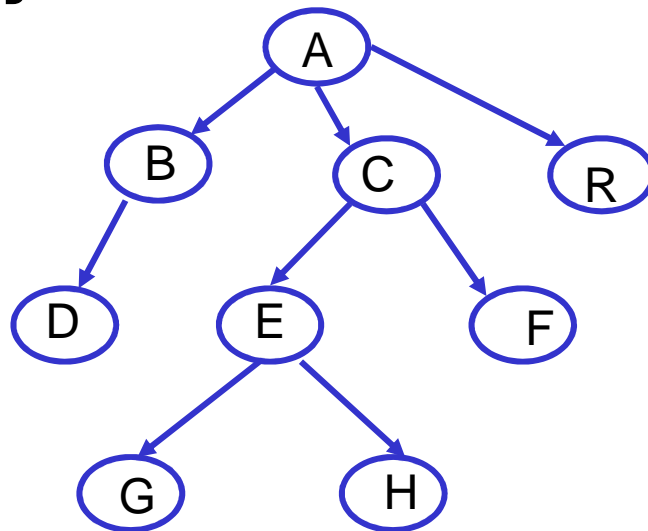
alberi generici



unico albero: radice: A, un sottoalbero

7.1 Alberi generici: visite

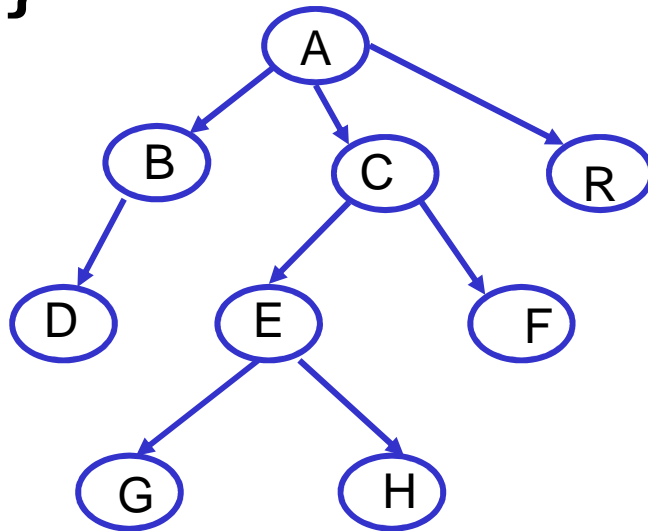
```
void preOrder ( albero ) {  
    esamina la radice;  
    se l'albero ha n sottoalberi {  
        preOrder ( primo sottoalbero);  
        ...  
        preOrder ( n-esimo sottoalbero);  
    }  
}
```



A B D C E G H F R

7.1 Alberi generici: visite

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

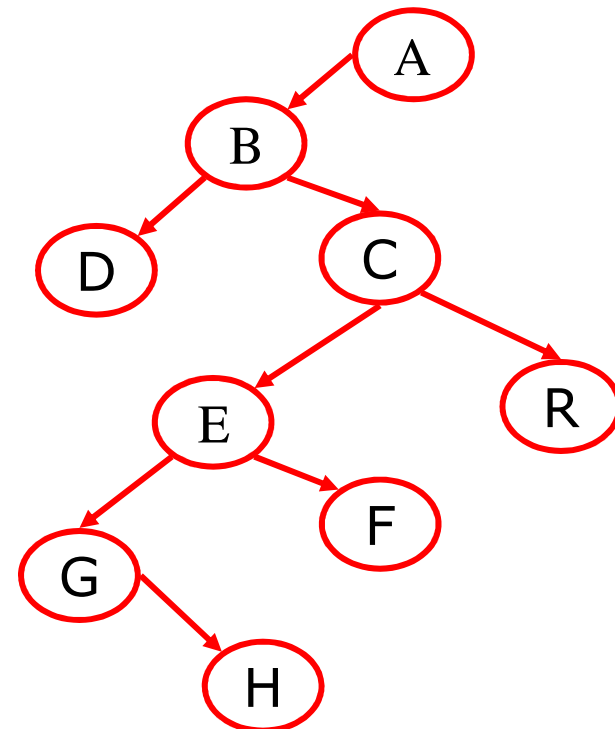
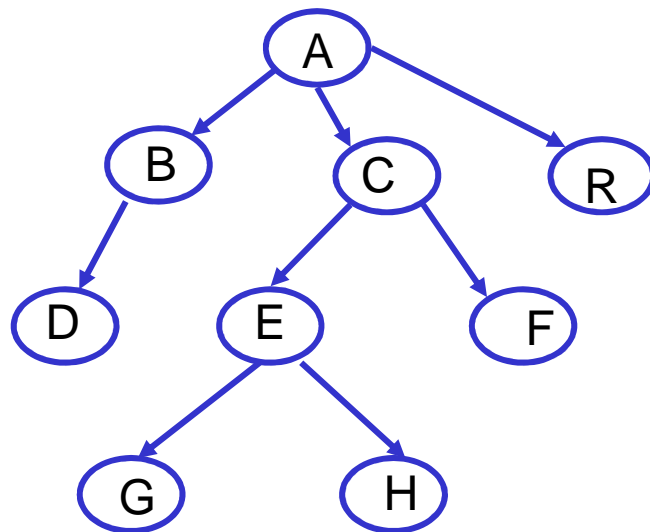


D B G H E F C R A

7.1 Alberi generici: memorizzazione

MEMORIZZAZIONE FIGLIO-FRATELLO

- **primo figlio a sinistra**
- **primo fratello a destra**



7.1 Alberi generici: corrispondenza fra visite

Utilizzando la memorizzazione figlio-fratello:

la visita **preorder** del trasformato corrisponde
alla visita **preorder** dell'albero generico

la visita **inorder** del trasformato corrisponde alla
visita **postorder** dell'albero generico

7.2 Esempi di programmi su alberi generici: conta nodi e foglie

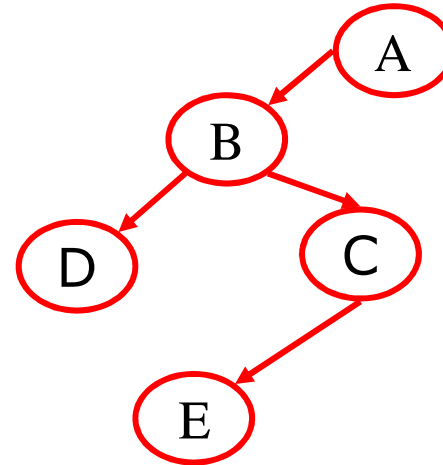
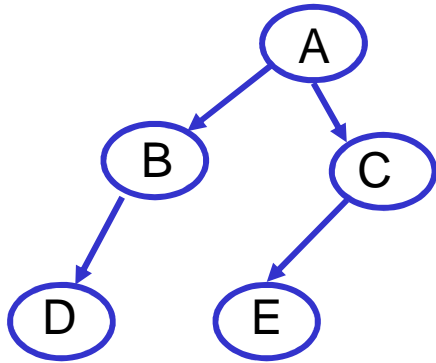
conta i nodi (vedi albero binario)

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

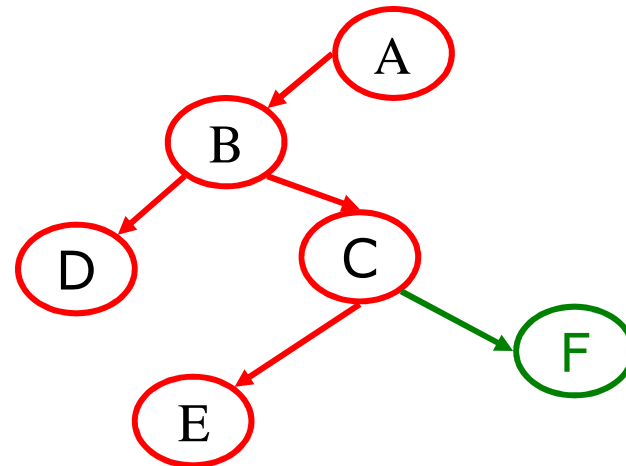
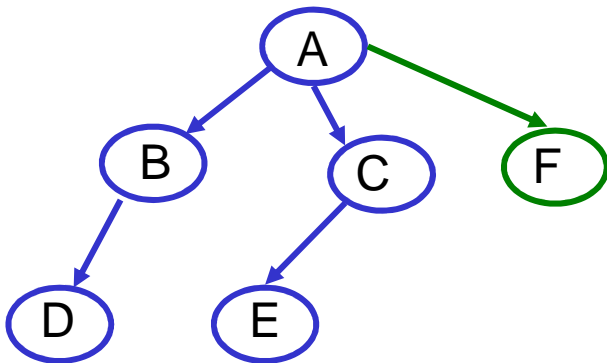
conta le foglie

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

7.2 Esempi di programmi su alberi generici: inserimento



Inserisci F come ultimo figlio di A



7.2 Esempi di programmi su alberi generici: inserimento

inserisce un nodo in fondo a una lista di fratelli

```
void addSon(InfoType x, Node* &tree) {  
    if (!tree) { //lista vuota  
        tree=new Node;  
        tree->label=x;  
        tree->left = tree->right = NULL;  
    }  
    else //lista non vuota  
        addSon(x, tree->right);  
}
```

7.2 Esempi di programmi su alberi generici: inserimento

inserisce **son** come ultimo figlio di **father**. Se l'albero e' vuoto, lo inserisce come radice

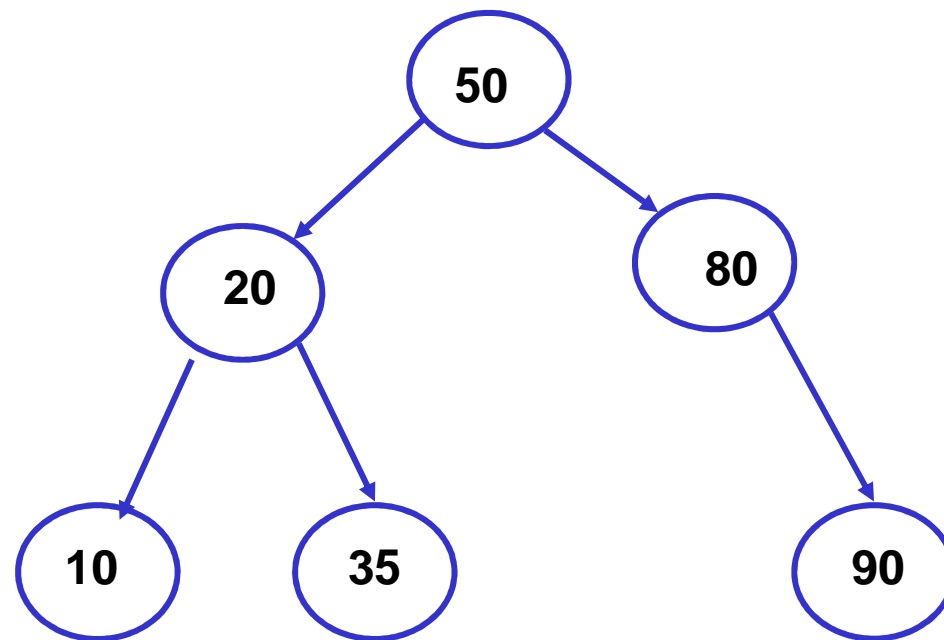
```
int insert(InfoType son, InfoType father, Node* &tree) {  
    if (!tree) {                                // albero vuoto  
        tree=new Node;  
        tree->label=son;  
        tree->left = tree->right = NULL;  
        return 1;  
    }  
    Node* a=findNode(father, tree); // a: puntatore di father  
    if (!a) return 0;                // father non trovato  
    addSon(son, a->left);  
    return 1;  
}
```

8. Alberi binari di ricerca: definizione

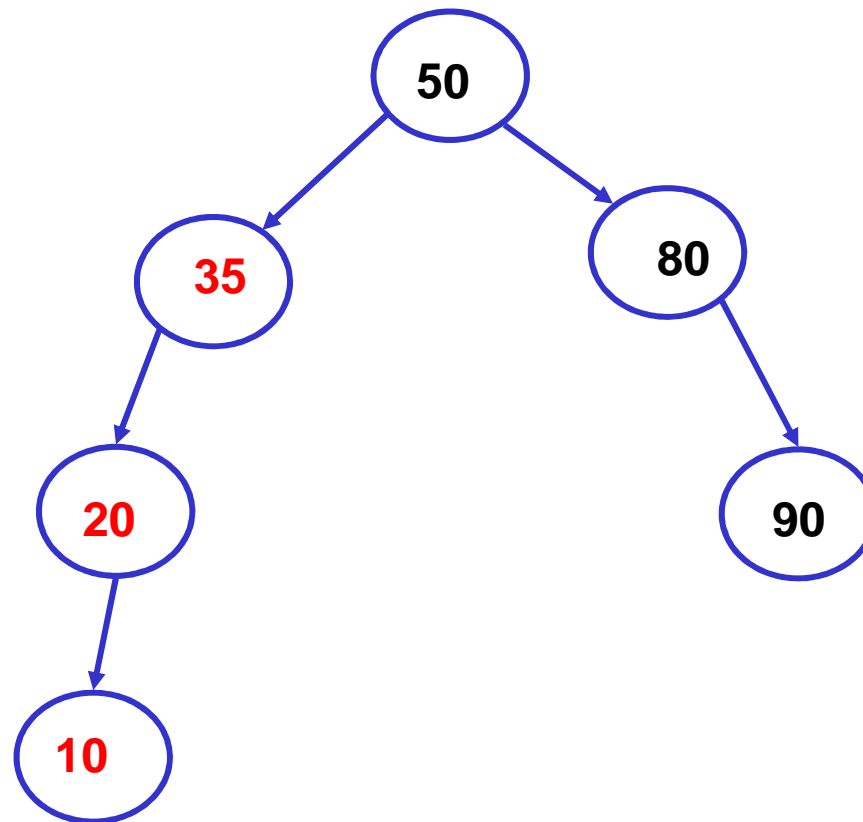
Un **albero binario di ricerca** è un albero binario tale che per ogni nodo p :

- i nodi del sottoalbero sinistro di p hanno etichetta **minore dell'etichetta di p**
- i nodi del sottoalbero destro di p hanno etichetta **maggiore dell'etichetta di p**

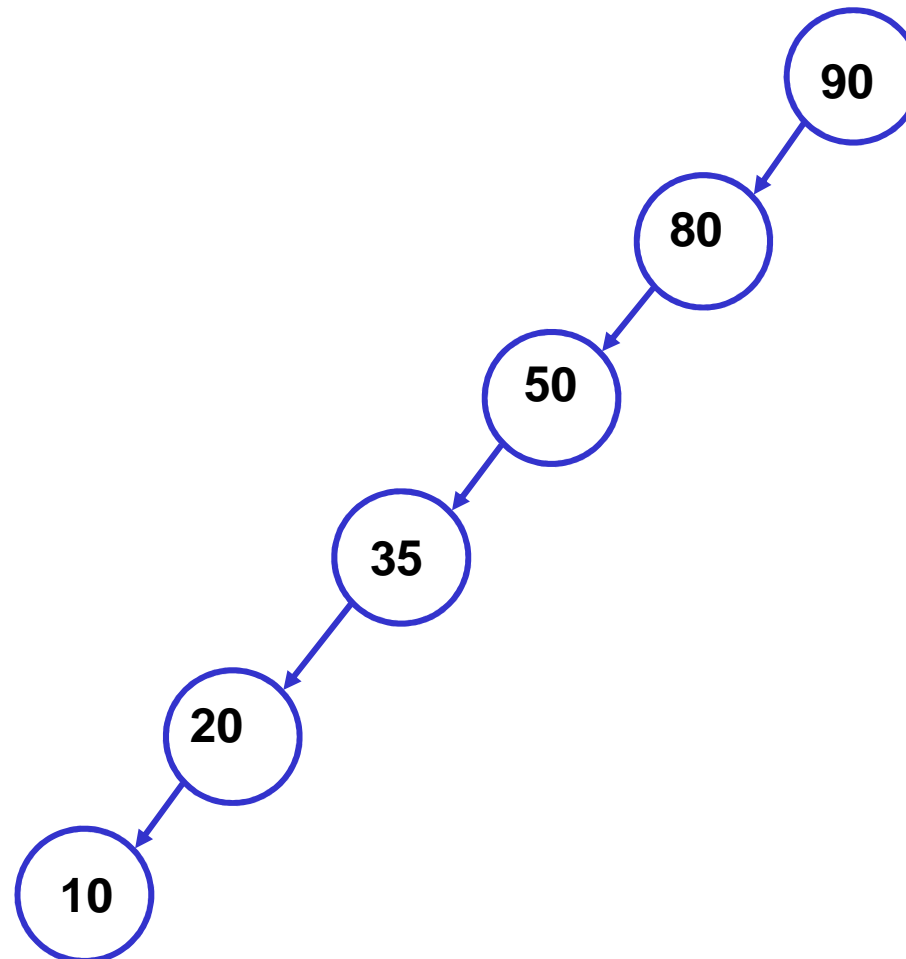
8. Un albero binario di ricerca



8. Un albero binario di ricerca con gli stessi nodi



8. Un albero binario di ricerca con gli stessi nodi



8. Alberi binari di ricerca: proprietà e operazioni

- **non ci sono doppi**
- **la visita simmetrica elenca le etichette in ordine crescente**

OPERAZIONI

- **ricerca** di un nodo
- **inserimento** di un nodo
- **cancellazione** di un nodo

8. Alberi binari di ricerca: ricerca

```
Node* findNode (InfoType n, Node* tree) {  
    if (!tree) return 0;                // albero vuoto  
  
    if (n == tree->label) return tree;  // n=radice  
  
    if (n < tree->label)                // n < radice  
        return findNode(n, tree->left);  
  
    return findNode(n, tree->right);    // n > radice  
}
```

8. Alberi binari di ricerca: ricerca

$$T(0)=a$$

$$T(n)= b + T(k) \quad k < n$$

$$T(0)=a$$

$$T(n)= b + T(n/2)$$

$$O(\log n)$$

$$T(0)=a$$

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

$$O(n)$$

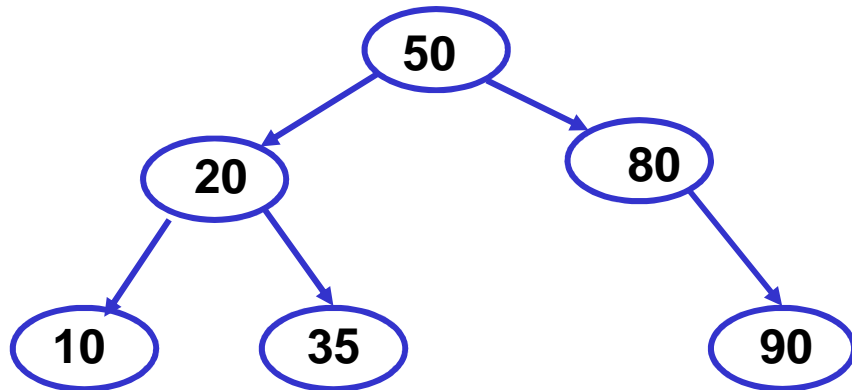
in media : **$O(\log n)$**

8. Alberi binari di ricerca: inserimento

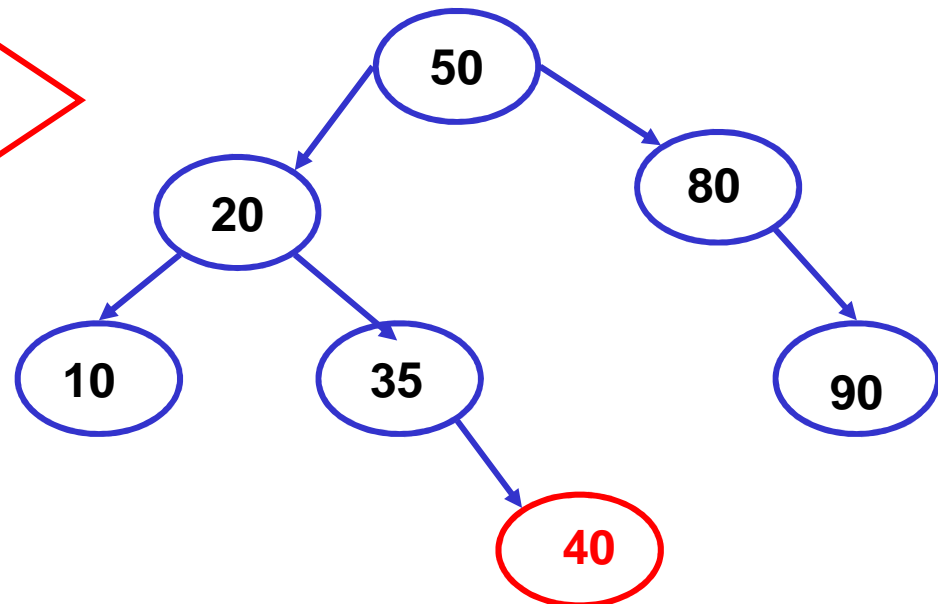
```
void insertNode (InfoType n, Node* &tree) {  
    if (!tree) { // albero vuoto: creazione nodo  
        tree=new Node;  
        tree->label=n;  
        tree->left = tree->right = NULL; return;  
    }  
    if (n<tree->label) // n<radice  
        insertNode (n, tree->left);  
    if (n>tree->label) // n>radice  
        insertNode (n, tree->right);  
}
```

$O(\log n)$

8. Esempio di inserimento



inserisco 40



8. Alberi binari di ricerca: cancellazione

restituisce l'etichetta del nodo più piccolo di un albero ed elimina il nodo che la contiene

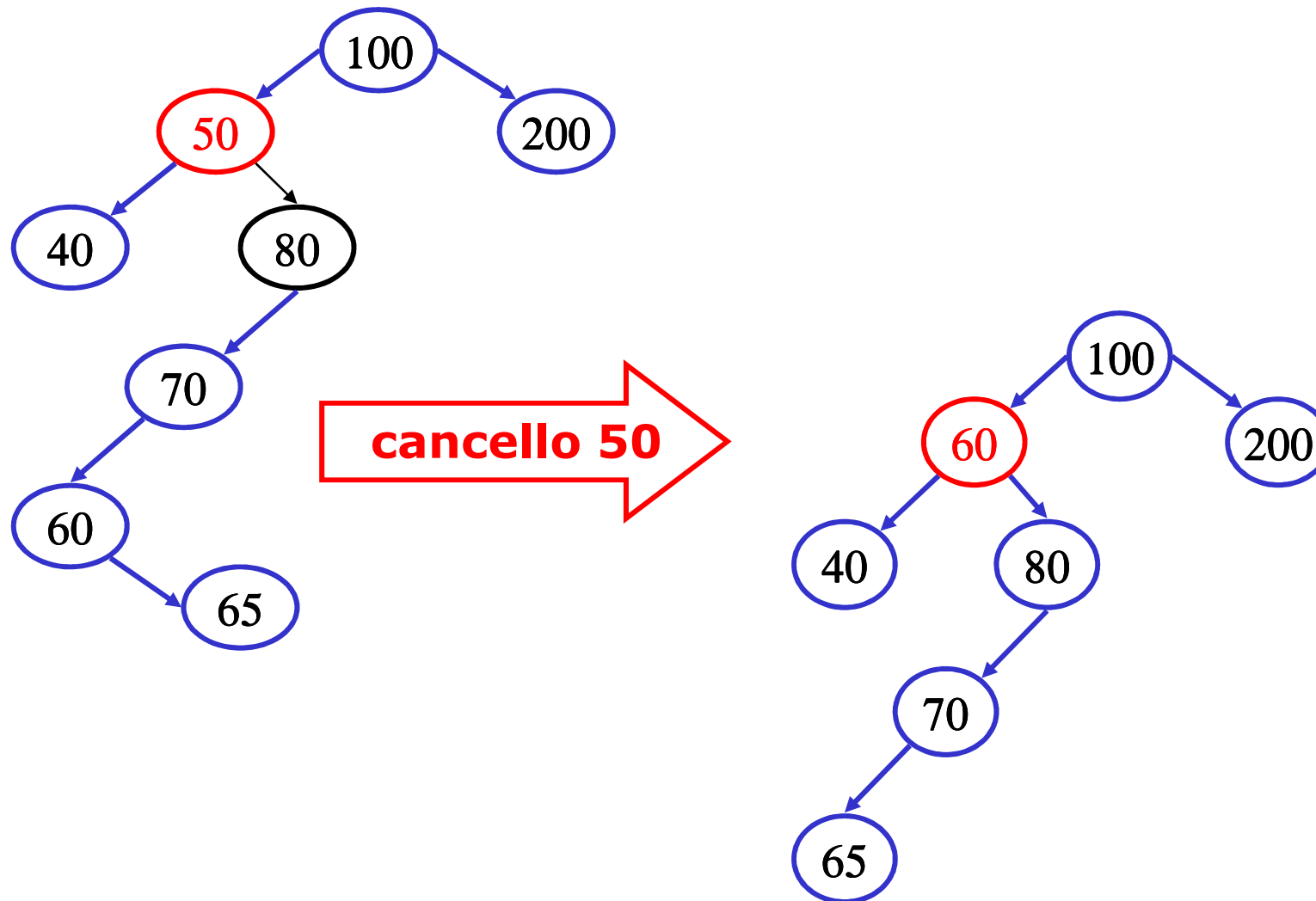
```
void deleteMin (Node* &tree, InfoType &m) {  
    if (tree->left)                //c'è un nodo più piccolo  
        deleteMin(tree->left, m);  
    else {  
        m=tree->label;              //restituisco l'etichetta  
        Node* a=tree;  
        tree=tree->right;           //connetto il sottoalbero destro di  
                                   // m al padre di m  
        delete a;                  //elimino il nodo  
    }  
}
```

8. Alberi binari di ricerca: cancellazione

```
void deleteNode(InfoType n, Node* &tree) {  
    if (tree)  
        if (n < tree->label)                //n minore della radice  
            { deleteNode(n, tree->left); return; }  
        if (n > tree->label)                //n maggiore della radice  
            { deleteNode(n, tree->right); return; }  
        if (!tree->left)                    //n non ha figlio sinistro  
            { Node* a=tree; tree=tree->right; delete a;return;}  
        if (!tree->right)                   //n non ha figlio destro  
            { Node* a=tree; tree=tree->left; delete a; return;}  
        deleteMin (tree->right, tree->label); //n ha entrambi i figli  
}
```

$O(\log n)$

8. Esempio di cancellazione



Limiti inferiori delle funzioni

$g(n)$ è di ordine $\Omega (f(n))$ se esistono un intero

n_0 ed una costante $c > 0$ tali che

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

Limiti inferiori: ragionamento intuitivo

Un problema è di ordine $\Omega (f(n))$ se non è possibile trovare un algoritmo che lo risolva con complessità minore di $f(n)$ (tutti gli algoritmi che lo risolvono hanno complessità $\Omega (f(n))$)

Si applica soltanto agli algoritmi

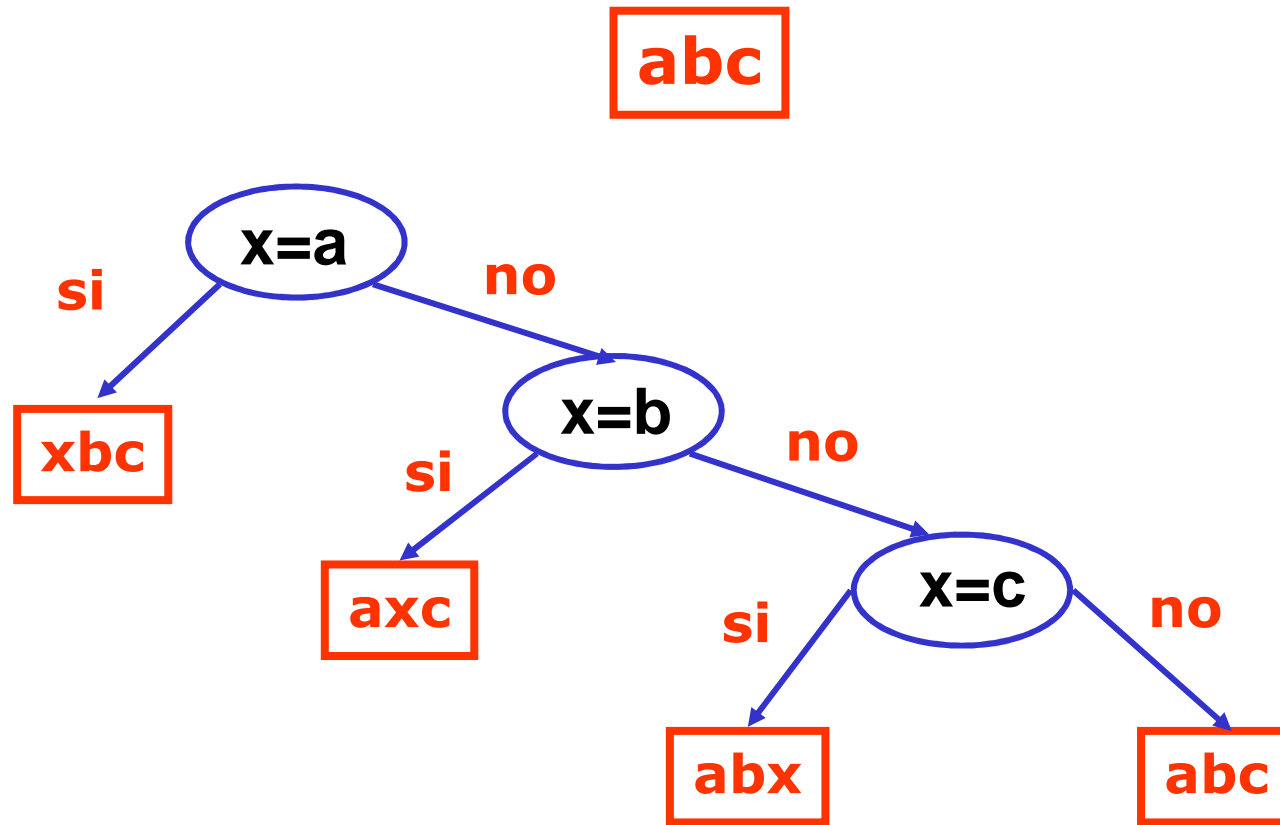
- basati su **confronti**
- che hanno complessità proporzionale al **numero di confronti** che vengono effettuati durante l'esecuzione dell'algoritmo

Limiti inferiori: alberi di decisione

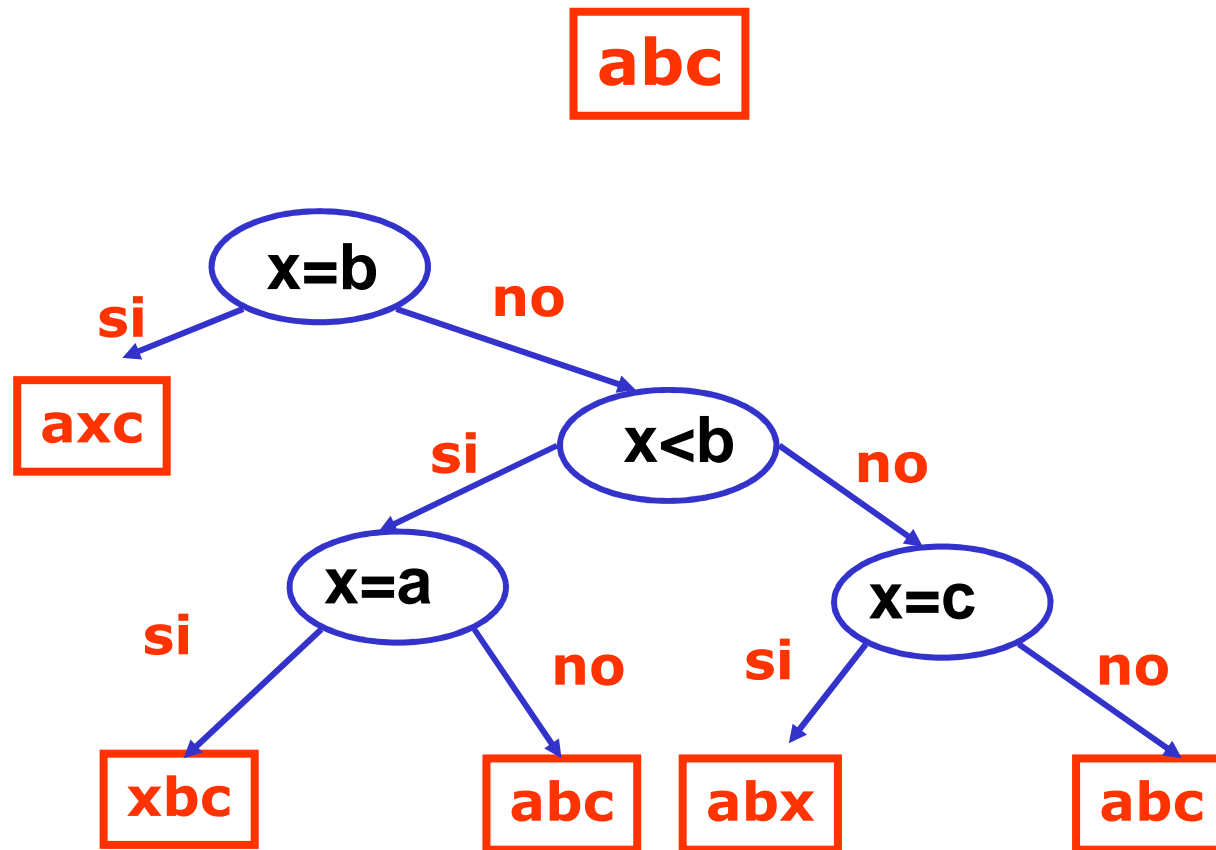
albero binario che corrisponde all'algoritmo:

- ogni **foglia** rappresenta una **soluzione** per un particolare assetto dei dati iniziali.
- ogni **cammino** dalla radice ad una foglia rappresenta una **esecuzione** dell'algoritmo (sequenza di confronti) per giungere alla soluzione relativa alla foglia

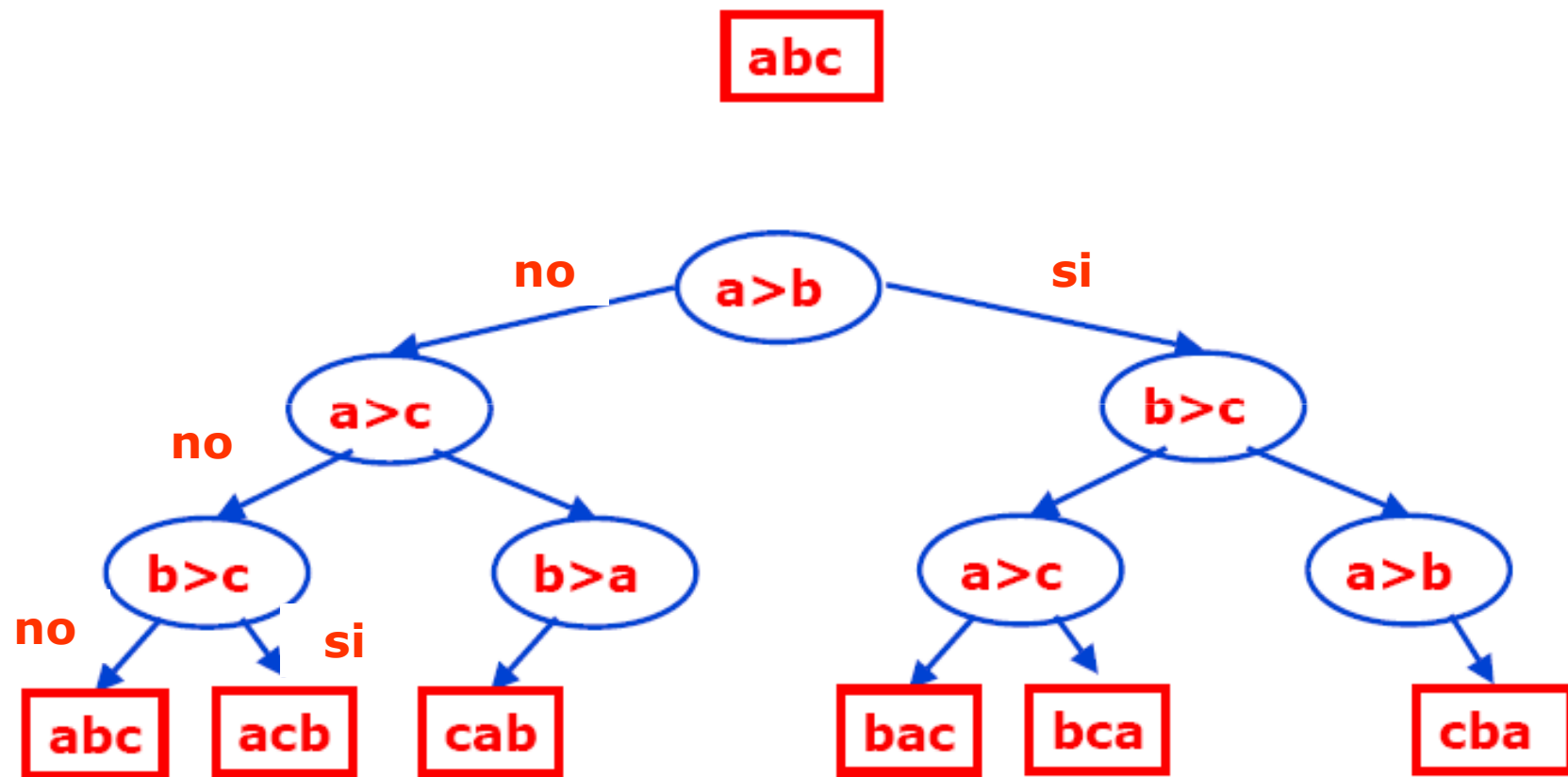
albero di decisione per la ricerca lineare



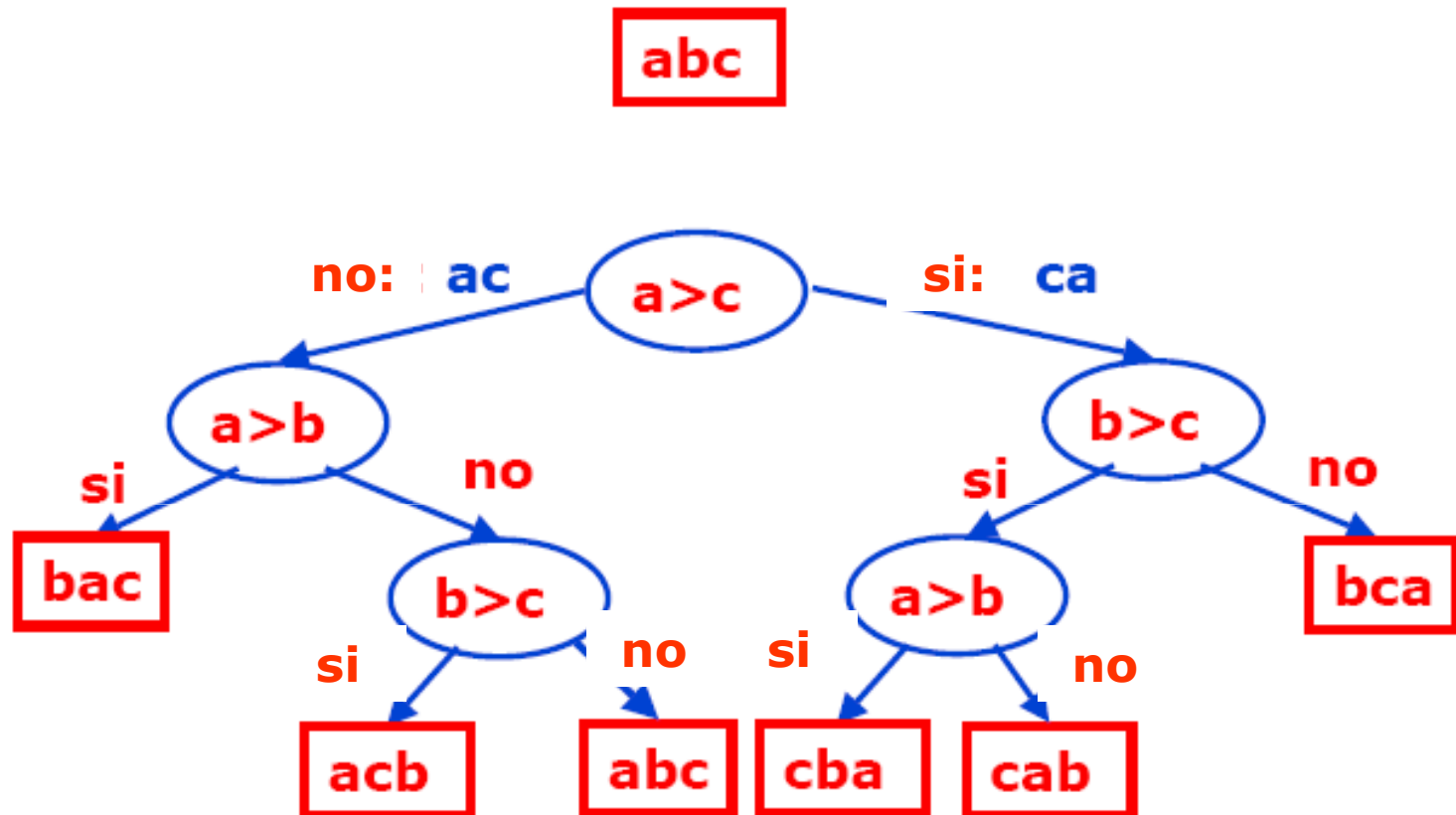
albero di decisione per la ricerca binaria



Albero del selection sort con 3 elementi



Albero del mergesort con 3 elementi



Limiti inferiori: alberi di decisione

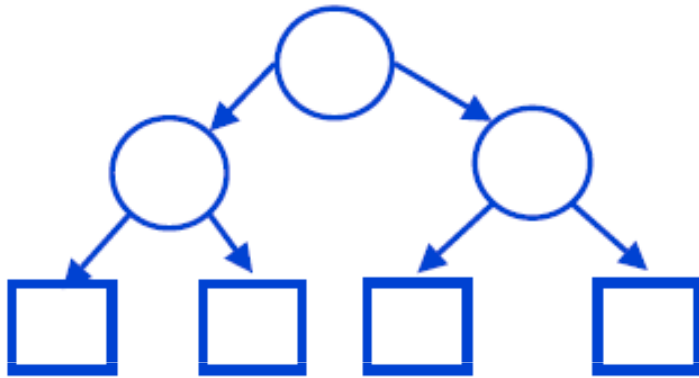
Ogni algoritmo che risolve un problema che ha $s(n)$ soluzioni ha un albero di decisione con almeno $s(n)$ foglie.

Un algoritmo ottimo nel caso peggiore (medio) ha il più corto cammino \max (medio) dalla radice alle foglie

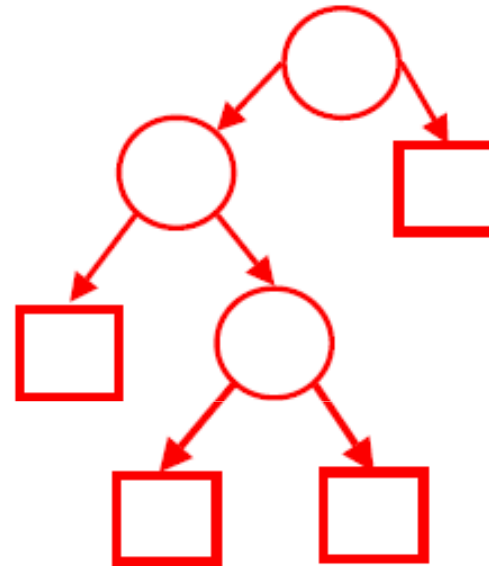
fatti

- Un albero binario con k livelli ha al massimo 2^k foglie (ce l'ha quando è bilanciato)
- Un albero binario con s foglie ha almeno $\log_2 s$ livelli
- Gli alberi binari bilanciati minimizzano sia il caso peggiore che quello medio: hanno $\log s(n)$ livelli.

Confronto fra algoritmi con 4 soluzioni



cammino max :2
cammino medio: 2



cammino max : 3
cammino medio: 2,25

$$(1+2+2*3)/4=9/4=2,25$$

algoritmi di ordinamento

$$n! = (n/e)^n$$

Numero soluzioni : $n!$

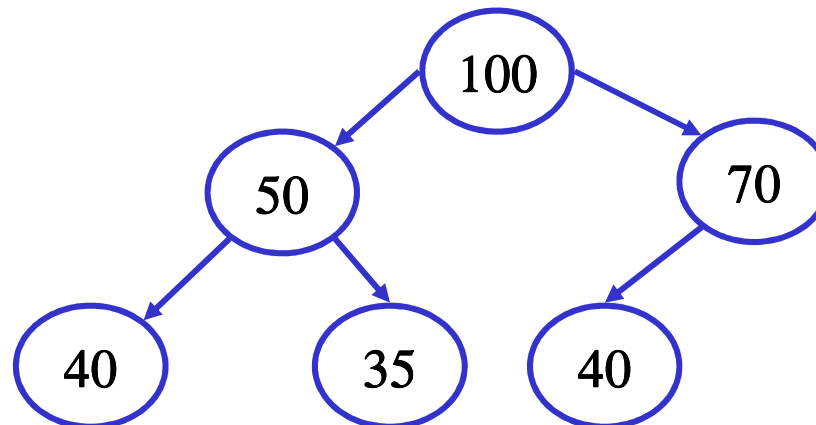
cammino medio e max: $\log(n!) \approx n \log n$

- **Mergesort è ottimo**
- **Quicksort è ottimo nel caso medio**
- **Non sempre il limite è raggiungibile
(la ricerca è $\Omega(\log n)$)**

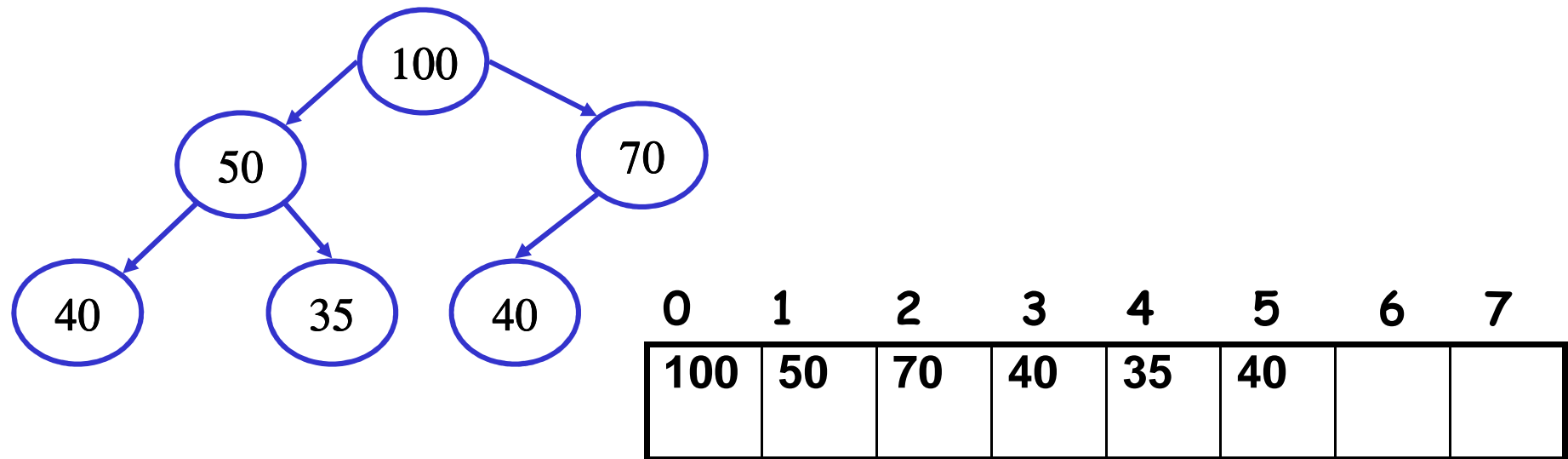
9. Heap: definizione

Heap: albero binario **quasi bilanciato** con le proprietà:

- i nodi dell'ultimo livello sono **addossati a sinistra**
- in ogni sottoalbero l'etichetta della radice é **maggiore o uguale** a quella di tutti i discendenti.



9. Heap: memorizzazione in array



figlio sinistro di i : $2i+1$

figlio destro di i : $2i+2$

padre di i : $(i-1)/2$

9. Heap: operazioni

OPERAZIONI

- **inserimento** di un nodo
- **estrazione** dell'elemento maggiore (radice)

8. Classe Heap

```
class Heap {  
    int * h;  
    int last; //indice dell'ultimo elemento  
    void up(int);  
    void down(int);  
    void exchange(int i, int j){  
        int k=h[i]; h[i]=h[j];h[j]=k;  
    }  
public:  
    Heap(int);  
    ~Heap();  
    void insert(int);  
    int extract();  
};
```

0	1	2	3	4	5	6	7
100	50	70	40	35	40		

last=5

8. Heap: costruttore e distruttore

```
Heap::Heap(int n){  
    h=new int[n];  
    last=-1;  
}
```

```
Heap::~~Heap() {  
    delete h [n];  
}
```

8. Heap: inserimento

- memorizza l'elemento nella prima posizione libera dell'array
- fai risalire l'elemento tramite scambi figlio-padre per mantenere la proprietà dello heap

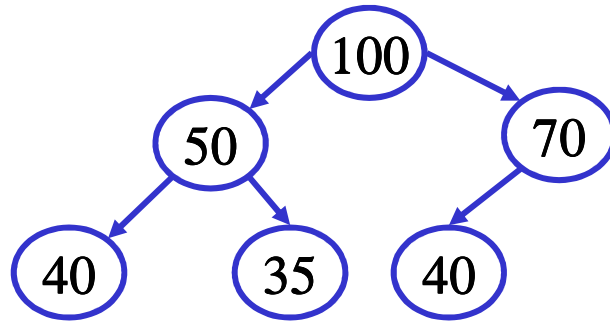
```
void Heap::insert (int x) {  
    h[++last]=x;  
    up(last);  
}
```

8. Heap: inserimento funzione **up**

```
void Heap::up(int i) { // i è l'indice dell'elemento da far risalire
    if (i > 0)          // se non sono sulla radice
        if (h[i] > h[(i-1)/ 2]) { // se l'elemento è maggiore del padre
            exchange(i,(i-1)/2); // scambia il figlio col padre
            up((i-1)/2);          // e chiama up sulla nuova posizione
        }                       // altrimenti termina
    }
```

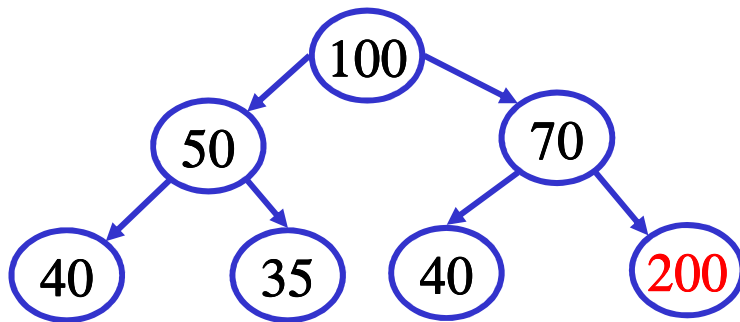
- la funzione termina o quando viene chiamata con l'indice 0 (radice) o quando l'elemento è inferiore al padre
- La complessità è $O(\log n)$ perchè ogni chiamata risale di un livello

8. Heap: esempio di inserimento



0	1	2	3	4	5	6	7
100	50	70	40	35	40		

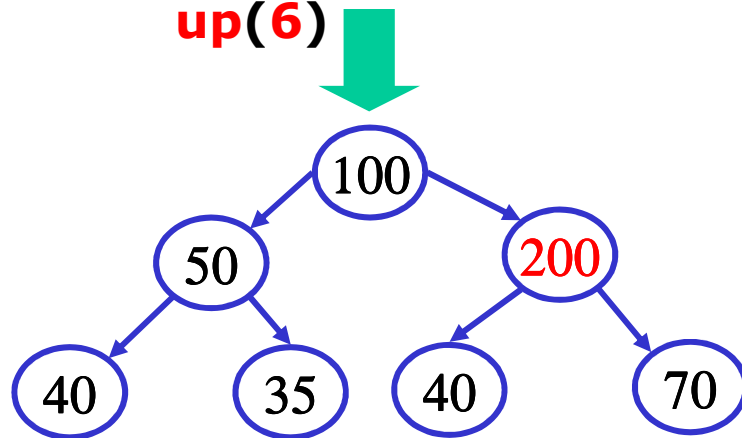
insert(200)



0	1	2	3	4	5	6	7
100	50	70	40	35	40	200	

8. Heap: esempio di inserimento

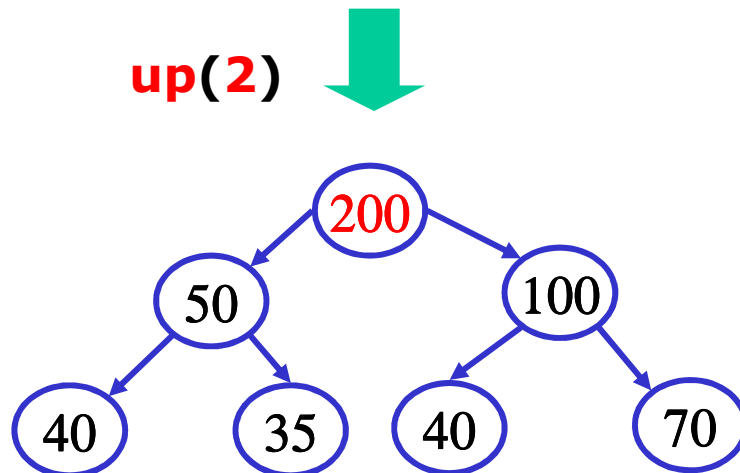
up(6)



0 1 2 3 4 5 6 7

100	50	200	40	35	40	70	
-----	----	-----	----	----	----	----	--

up(2)



0 1 2 3 4 5 6 7

200	50	100	40	35	40	70	
-----	----	-----	----	----	----	----	--

up(0)

8. Heap: estrazione

- restituisci il primo elemento dell'array
- metti l'ultimo elemento al posto della radice e decrementa last
- fai scendere l'elemento tramite scambi padre-figlio per mantenere la proprietà dello heap

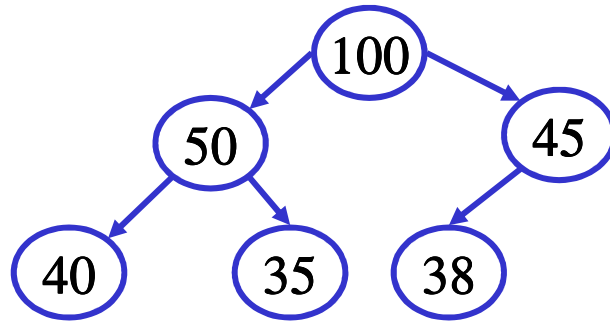
```
int Heap::extract() {  
    int r=h[0];  
    h[0]=h[last--];  
    down(0);  
    return r;  
}
```

8. Heap: estrazione funzione **down**

```
void Heap::down(int i) { // i è l'indice dell'elemento da far scendere
    int son=2*i+1;        // son = indice del figlio sinistro (se esiste)
    if (son == last) {    // se i ha un solo figlio (è l'ultimo dell'array)
        if (h[son] > h[i]) // se il figlio è maggiore del padre
            exchange(i,last); // fai lo scambio, altrimenti termina
    }
    else if (son < last) { // se i ha entrambi i figli
        if (h[son] < h[son+1]) son++; // son= indice del maggiore fra i due
        if (h[son] > h[i]) { // se il figlio è maggiore del padre
            exchange(i,son); // fai lo scambio
            down(son);       // e chiama down sulla nuova posizione
        } // altrimenti termina (termina anche se i non ha figli)
    }
}
```

complessità : $O(\log n)$

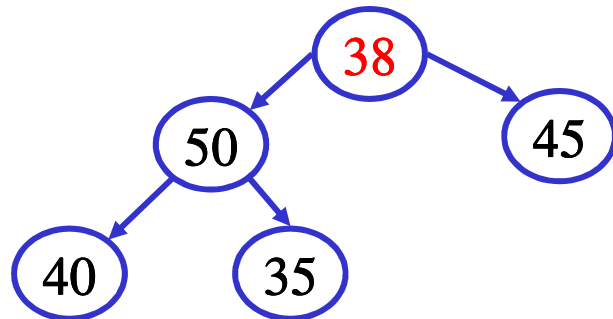
8. Heap: esempio di estrazione



0 1 2 3 4 **5** 6 7

100	50	45	40	35	38		
-----	----	----	----	----	----	--	--

extract() -> **100**

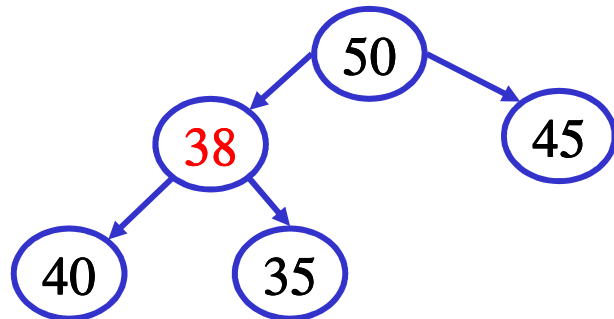


0 1 2 3 **4** 5 6 7

38	50	45	40	35	38		
-----------	----	----	----	----	----	--	--

8. Heap: esempio di estrazione

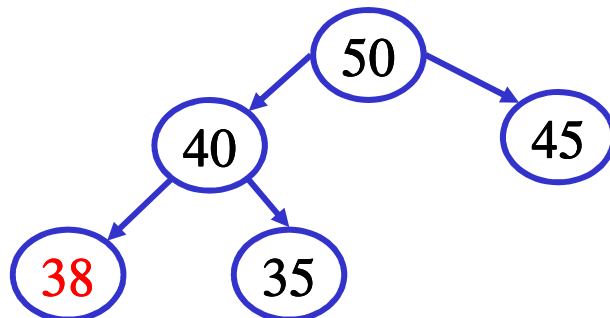
down(0)



0 1 2 3 4 5 6 7

50	38	45	40	35	38		
----	----	----	----	----	----	--	--

down(1)



0 1 2 3 4 5 6 7

50	40	45	38	35	38		
----	----	----	----	----	----	--	--

down(3)

8. Algoritmo di ordinamento Heapsort

- trasforma l'array in uno heap (**buildheap**)
- esegui **n** volte l'estrazione scambiando ogni volta il primo elemento dell'array con quello puntato da **last**

```
void heapSort(int* A, int n) {  
    buildHeap(A,n-1);           // O(n)  
    int i=n-1;  
    while (i > 0) {              // O(nlogn)  
        extract(A,i);  
    }  
}
```

O(nlogn)

8. down modificata

```
void down(int * h, int i, int last) {  
    int son=2*i+1;  
    if (son == last) {  
        if (h[son] > h[i]) exchange(h, i,last);  
    }  
    else if (son < last) {  
        if (h[son] < h[son+1]) son++;  
        if (h[son] > h[i]) {  
            exchange(h, i,son);  
            down(h, son, last);  
        }  
    }  
}
```

$O(\log n)$

**I parametri sono l'array, l'indice
dell'elemento da far scendere,
l'ultimo elemento dello heap**

8. Estract modificata

```
void extract(int* h, int & last) {  
    exchange(h, 0, last--);  
    down(h, 0, last);  
}
```

- I parametri sono l'array e l'ultimo elemento dello heap
- L'ultimo elemento viene scambiato con il primo
- Non si restituisce nulla

$O(\log n)$

8. Trasforma l'array in uno heap (buildheap)

- Esegui la funzione **down** sulla prima metà degli elementi dell'array (gli elementi della seconda metà sono foglie)
- Esegui **down** partendo dall'elemento centrale e tornando indietro fino al primo

```
void buildHeap(int* A, int n) {  
    for (int i=n/2; i>=0; i--) down(A,i,n);  
}
```

$O(n)$

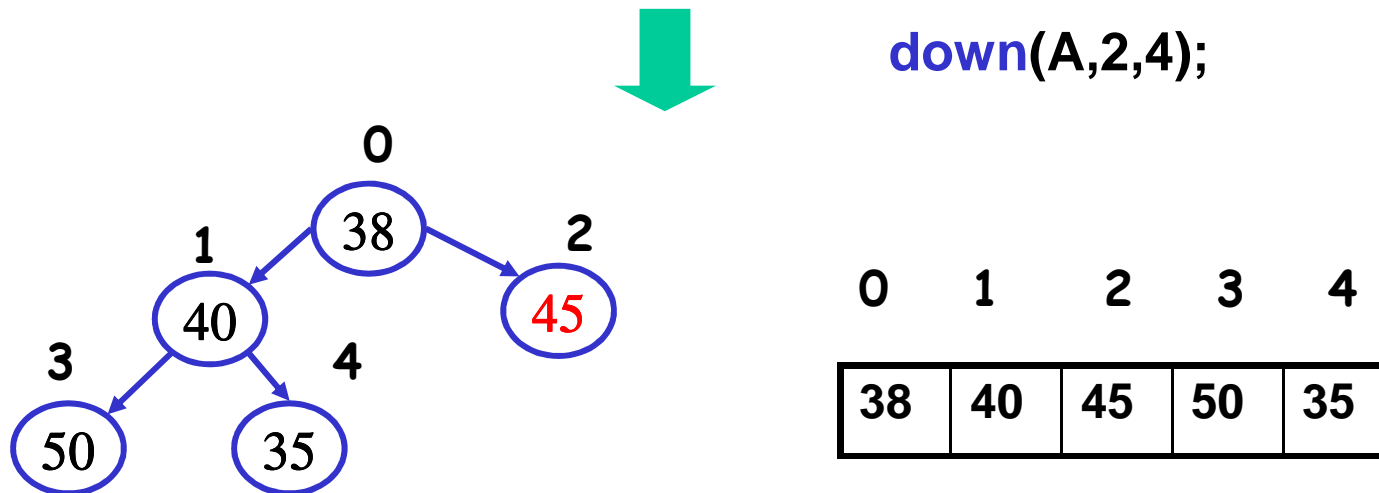
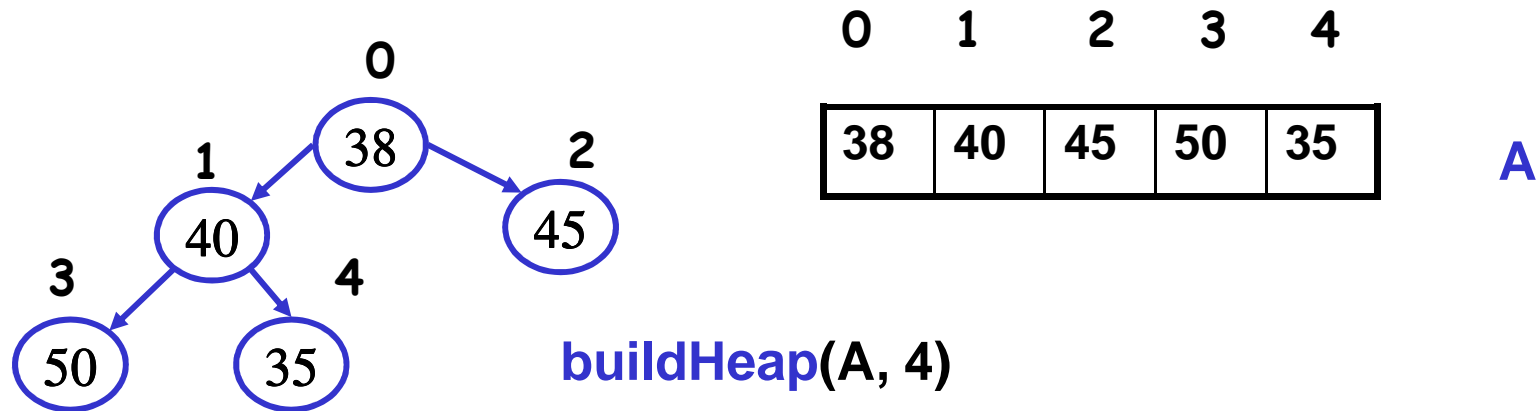
8. Esempio di heapsort

heapSort(A, int 5)

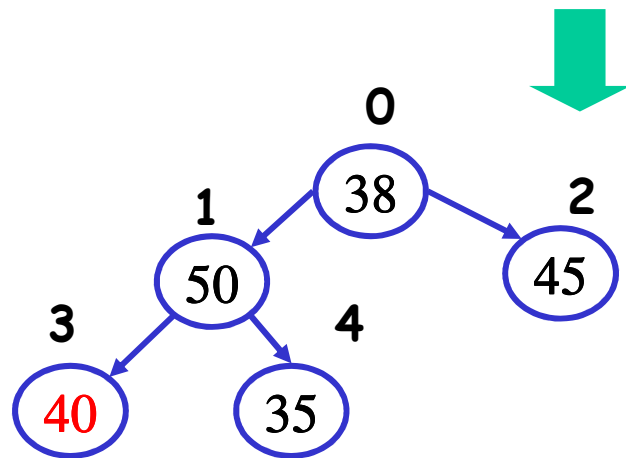
0	1	2	3	4
38	40	45	50	35

A

8. Esempio di heapsort: **buildHeap**

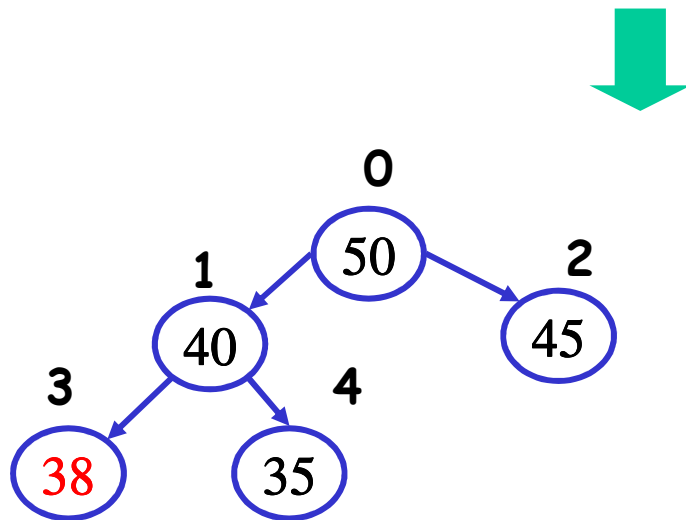


8. Esempio di heapsort: buildHeap



`down(A,1,4);`

0	1	2	3	4
38	50	45	40	35



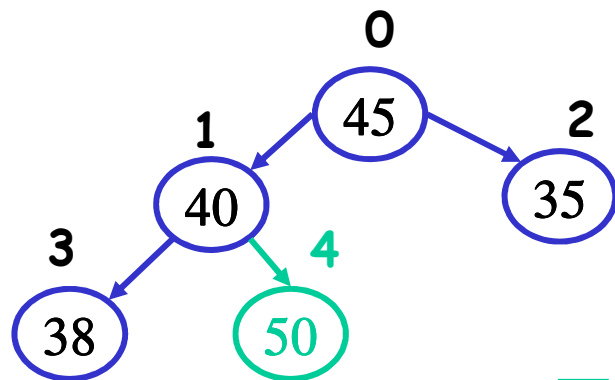
`down(A,0,4);`

0	1	2	3	4
50	40	45	38	35

8. Esempio di heapsort: estrazioni



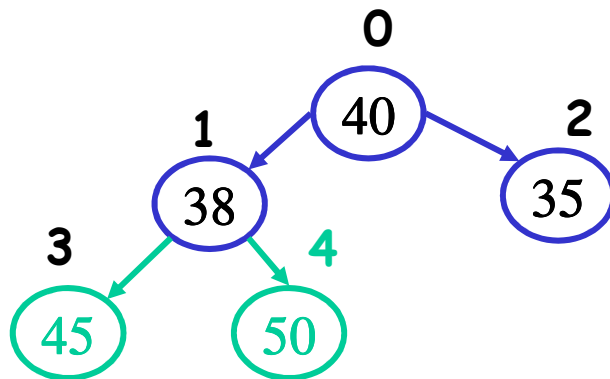
`extract(A,4);`



0	1	2	3	4
45	40	35	38	50



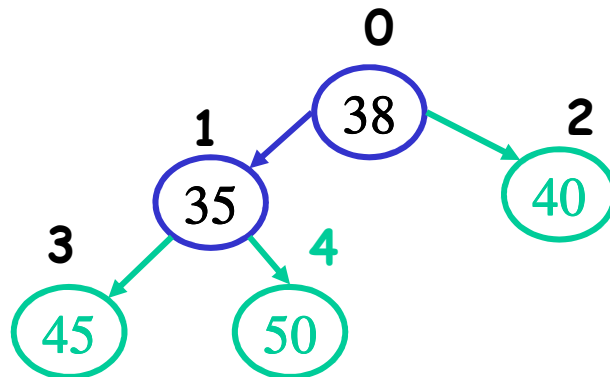
`extract(A,3);`



0	1	2	3	4
40	38	35	45	50

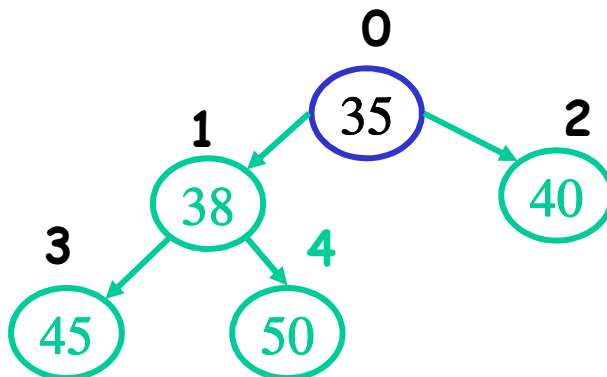
8. Esempio di heapsort: estrazioni

extract(A,2);



0	1	2	3	4
38	35	40	45	50

extract(A,1);



0	1	2	3	4
35	38	40	45	50

Metodo hash

- **metodo di ricerca in array**
- **non basato su confronti**
- **molto efficiente**

Metodo hash

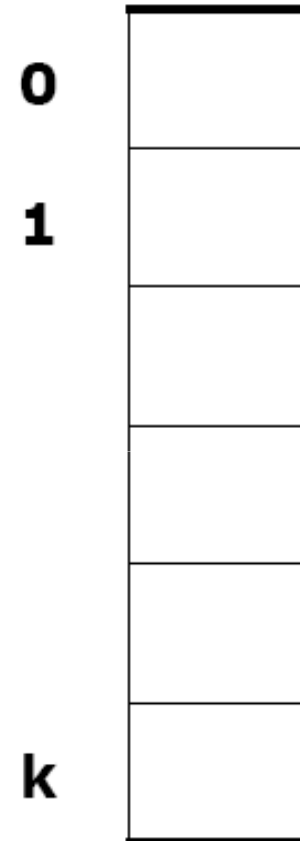
h (funzione hash) : InfoType \rightarrow indici

$x \rightarrow h(x)$

$n \leq k$

n = numero massimo di elementi

k = dimensione dell'array



Metodo hash: accesso diretto

h iniettiva:

$h(x)$: indirizzo hash dell'elemento che contiene x

complessità : $O(1)$

Problema: memoria (k può essere molto grande)

Insiemi di al massimo 5 cifre decimali

```
bool hashSearch (int *A , int k, int x) {
```

```
    int i=h(x);
```

```
    if (A[i ]== 1) return true;
```

```
    else return false ;
```

```
}
```

{ 0, 2, 7 }

n= 5

k= 10

n/k=0,5

h(x)= x

h(0)= 0

h(2)=2

h(7)=7

0	1
1	0
2	1
7	1
8	0
9	0

NB: non è necessario memorizzare l'elemento

Metodo hash ad accesso non diretto

Si rilascia l'iniettività e si permette che due elementi diversi abbiano lo stesso indirizzo hash:

$$h(x1) = h(x2) \text{ collisione}$$

Bisogna gestire le seguenti situazioni:

- **Come si cerca un elemento se si trova il suo posto occupato da un altro**
- **come si inseriscono gli elementi**

Metodo hash ad accesso non diretto

Una prima soluzione:

funzione hash modulare: $h(x) = (x \% k)$

(siamo sicuri che vengono generati tutti e soli gli indici dell'array)

Legge di scansione lineare: se non si trova l'elemento al suo posto, lo si cerca nelle posizioni successive fino a trovarlo o ad incontrare una posizione vuota

L'inserimento è fatto con lo stesso criterio

Esempio: insieme di al massimo 5 cifre

$n=k=5$

$$h(x) = x \% k$$

$n/k=1$

$$h(0) = 0$$

$$h(2) = 2$$

$$h(7) = 2$$

$\{0, 2, 7\}$

0	0
1	-1
2	2
3	7
4	-1

Conseguenze

Agglomerato: gruppo di elementi con indirizzi hash diversi

La presenza di collisioni ed agglomerati aumenta il tempo di ricerca

esempio

Esempio: $h(x) = x \% k$

$k = 99$

$h(x)$

$h(99) = h(198) = h(297) = 0$

$h(199) = h(100) = 1$

99, 198, 199, 297, 100

0

99

1

198

2

199

3

297

100

98

Metodo hash: ricerca con scansione lineare

```
bool hashSearch (int *A , int k, int x) {  
    int i=h(x);  
    for (int j=0; j<k; j++) {  
        int pos = (i+j)%k;      // nota la somma in modulo  
        if (A[pos] == -1) return false ;  
        if (A[pos] == x) return true ;  
    }  
    return false ;  
}
```

-1: posizione vuota

Metodo hash : inserimento

```
int hashInsert (int *A , int k, int x) {  
    int i=h(x);  
    for (int j=0; j<k; j++) {  
        int pos = (i+j)%k;  
        if (A[pos] == -1) {  
            A[pos] = x;  
            return 1;  
        }  
    }  
    return 0;  
}
```


Metodo hash: inserimento in presenza di cancellazioni

```
int hashInsert (int *A , int k, int x) {  
    int i=h(x);  
    for (int j=0; j<k; j++) {  
        int pos = (i+j)%k;  
        if ((A[pos] == -1) || (A[pos] == -2)) {  
            A[pos] = x;  
            return 1;  
        }  
    }  
    return 0;  
}
```

-1: posizione vuota
-2: posizione disponibile

Scansioni

$$\text{scansione_lineare}(x) = (h(x) + \text{cost} * j) \bmod k$$

$$\text{Es: } (h(x) + j) \bmod k \quad j=1, 2, \dots$$

$$\text{scansione_quadratica}(x; j) = (h(x) + \text{cost} * j^2) \bmod k$$

$$\text{Es: } (h(x) + j^2) \bmod k \quad j=1, 2, \dots$$

La diversa lunghezza del passo di scansione riduce gli agglomerati, ma è necessario controllare che la scansione visiti tutte le possibili celle vuote dell'array, per evitare che l'inserimento fallisca anche in presenza di array non pieno.

Tempo medio di ricerca

Il tempo medio di ricerca (numero medio di confronti) dipende da

Rapporto $\alpha = n/k$ (sempre < 1)

Legge di scansione (migliore con la scansione quadratica e altre più sofisticate)

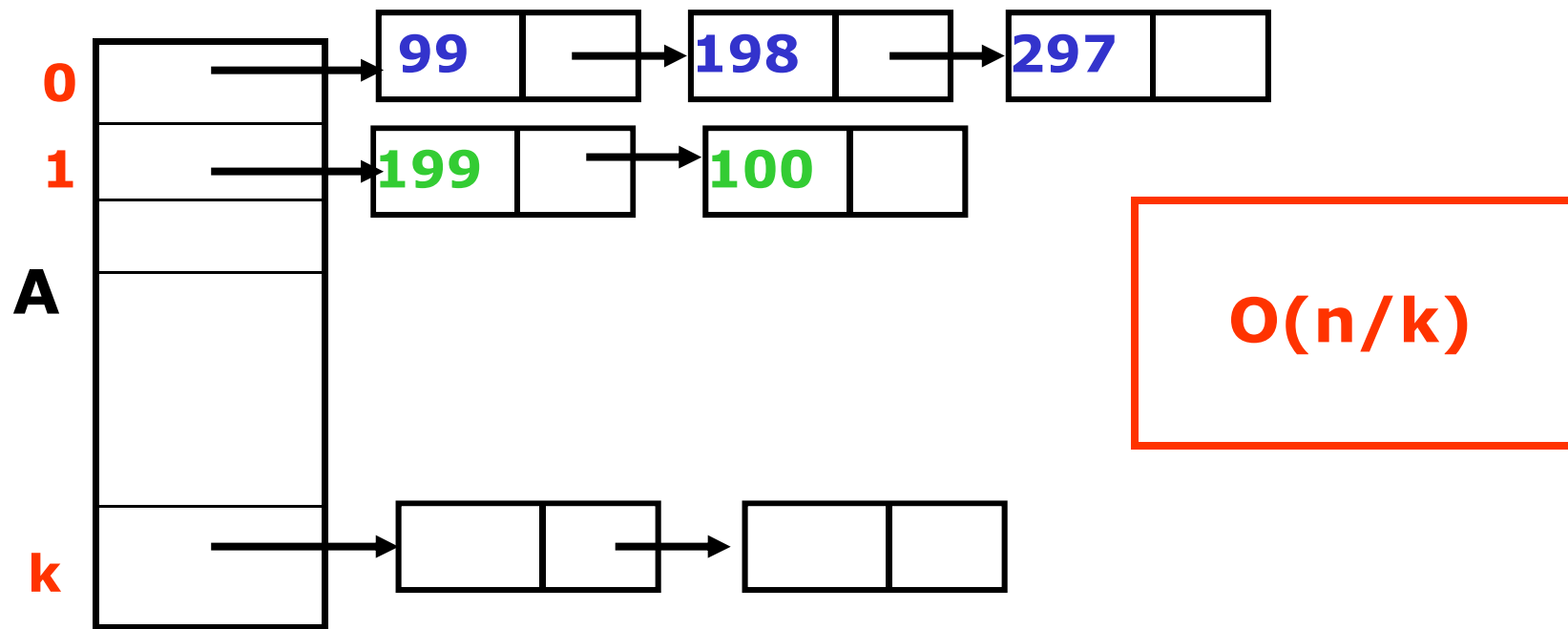
Uniformità della funzione hash (genera gli indici con uguale probabilità)

Problemi con l'indirizzamento aperto

Molti inserimenti e cancellazioni degradano il tempo di ricerca a causa degli agglomerati. E' necessario periodicamente "risistemare" l'array.

Metodo di concatenazione

- Array A di $k \leq n$ puntatori ($n/k \geq 1$)
- elementi che collidono su i nella lista di puntatore $A[i]$
- evita del tutto gli agglomerati



Dizionari (tabelle)

chiave	informazione

Ricerca
Inserimento
Cancellazione

Con **$h(\text{chiave})$** si raggiunge l'informazione

Es: rubrica telefonica, studenti (chiave: matricola)

Programmazione dinamica e algoritmi greedy

Programmazione dinamica

Si può usare quando non è possibile applicare il metodo del divide et impera (non si sa con esattezza quali sottoproblemi risolvere e non è possibile partizionare l'insieme in sottoinsiemi disgiunti)

Metodo: si risolvono **tutti i sottoproblemi a partire dal basso e si conservano i risultati ottenuti per poterli usare successivamente. (strategia bottom-up)**

La complessità del metodo dipende dal numero dei sottoproblemi

Programmazione dinamica

Quando si può applicare

sottostruttura ottima: una soluzione ottima del problema contiene la soluzione ottima dei sottoproblemi

sottoproblemi comuni : un algoritmo ricorsivo richiederebbe di risolvere lo stesso sottoproblema più volte

Più lunga sottosequenza comune (PLSC)

$\alpha = a \ b \ c \ a \ b \ b \ a$ $\beta = c \ b \ a \ b \ a \ c$

$\alpha = \alpha_1 \ \alpha_2 \ \alpha_3 \ \alpha_4 \ \alpha_5 \ \alpha_6 \ \alpha_7$

$\beta = \beta_1 \ \beta_2 \ \beta_3 \ \beta_4 \ \beta_5 \ \beta_6$

3 PLSC: **baba, cbba, caba**

Lunghezza delle PLSC = 4

PLSC

$L(i,j)$ = lunghezza delle PLSC di $\alpha_1 \dots \alpha_i$ e $\beta_1 \dots \beta_j$

$$L(0,0)=L(i,0)=L(0,j)=0$$

$$L(i,j)=L(i-1,j-1)+1 \quad \text{se } \alpha_i = \beta_j$$

$$L(i,j)=\max(L(i,j-1),L(i-1,j)) \quad \text{se } \alpha_i \neq \beta_j$$

```
int length(char* a, char* b, int i, int j) {  
    if (i==0 || j==0) return 0;  
    if (a[i]==b[j]) return length(a, b, i-1, j-1)+1;  
    else  
        return max(length(a,b,i,j-1),length(a,b,i-1,j));  
};
```

$$T(n) = b + 2T(n-1)$$

**Costruisce tutti gli $L(i,j)$ a partire dagli
indici più piccoli (bottom-up):**

$L(0,0), L(0,1) \dots L(0,n),$

$L(1,0), L(1,1) \dots L(1,n),$

\dots

$L(m,0), L(m,1) \dots L(m,n)$

Algoritmo di programmazione dinamica

```
const int m=7; const int n=6;
int L [m+1][n+1];
int quickLength(char *a, char *b) {
    for (int j=0; j<=n; j++) L[ 0 ][ j ]=0; // prima riga

    for (int i=1; i<=m; i++) { // i-esima riga
        L[ i ][ 0 ]=0;
        for (j=1; j<=n; j++)
            if (a[ i ] != b[ j ])
                L[ i ][ j ] = max(L[ i ][ j-1 ],L[ i-1 ][ j ]);
            else L[ i ][ j ]=L[ i-1 ][ j-1 ]+1;
    }
    return L[ m ][ n ];
}
```

$T(n) \in O(n^2)$

PLSC

		c	b	a	b	a	c
	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1
b	0	0	1	1	2	2	2
c	0	1	1	1	2	2	3
a	0	1	1	2	2	3	3
b	0	1	2	2	3	3	3
b	0	1	2	2	3	3	3
a	0	1	2	3	3	4	4

PLSC

		c	b	a	b	a	c
	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1
b	0	0	1	1	2	2	2
c	0	1	1	1	2	2	3
a	0	1	1	2	2	3	3
b	0	1	2	2	3	3	3
b	0	1	2	2	3	3	3
a	0	1	2	3	3	4	4

cbba


```
void print(char *a, char *b, int i=m, int j=n){  
    if ((i==0) || (j==0) ) return;  
    if (a[i]==b[j]) {  
        print(a,b, i-1, j-1);  
        cout << a[i];  
    }  
    else if (L[i][j] == L[i-1][j])  
        print(a,b, i-1, j);  
    else print(a,b, i, j-1);  
}
```

Algoritmi greedy (golosi)

la soluzione ottima si ottiene mediante una sequenza di scelte

In ogni punto dell'algoritmo, viene scelta la strada che in quel momento sembra la migliore

la scelta locale deve essere in accordo con la scelta globale: scegliendo ad ogni passo l'alternativa che sembra la migliore non si perdono alternative che potrebbero rivelarsi migliori nel seguito.

Algoritmi greedy

Metodo **top-down**

Non sempre si trova la soluzione **ottima ma in certi casi si può trovare una soluzione approssimata (esempio del problema dello zaino)**

codici di compressione

Alfabeto : insieme di caratteri (es: a, b, c, d, e, f)

Codice binario: assegna ad ogni carattere una stringa binaria

Codifica del testo: sostituisce ad ogni carattere del testo il corrispondente codice binario.

Decodifica: ricostruire il testo originario.

Il codice può essere a lunghezza fissa o a lunghezza variabile

codici di compressione

	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
Codice a lunghezza fissa	000	001	010	011	100	101
Codice a lunghezza variabile	0	101	100	111	1101	1100

codici prefissi

Codifica di **abc** con codice a lunghezza fissa: :

000 001 010 (9 bit)

Codifica di **abc** con codice a lunghezza variabile :

0 101 100 (7 bit)

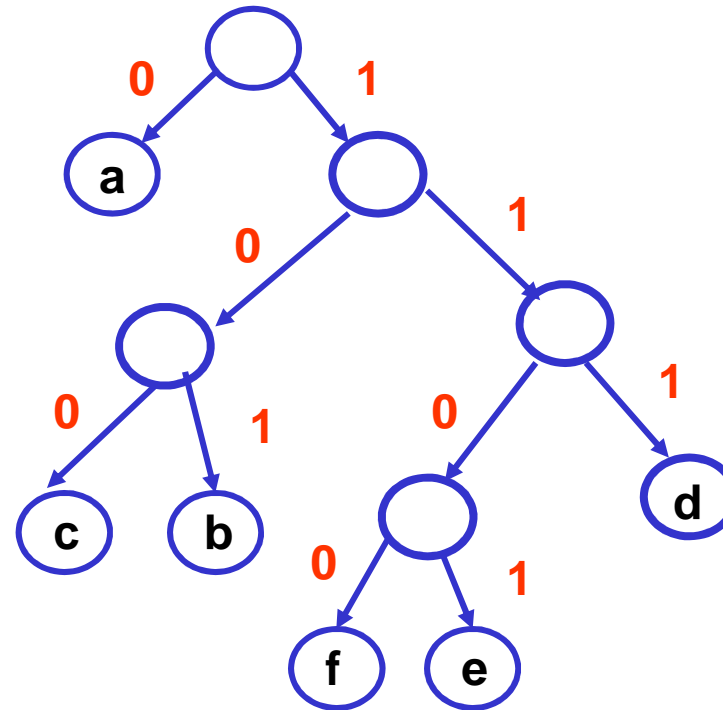
Problema della decodifica

Codice prefisso: nessun codice può essere il prefisso di un altro codice

codici prefissi

**I codici prefissi
possono essere
rappresentati con
alberi binari**

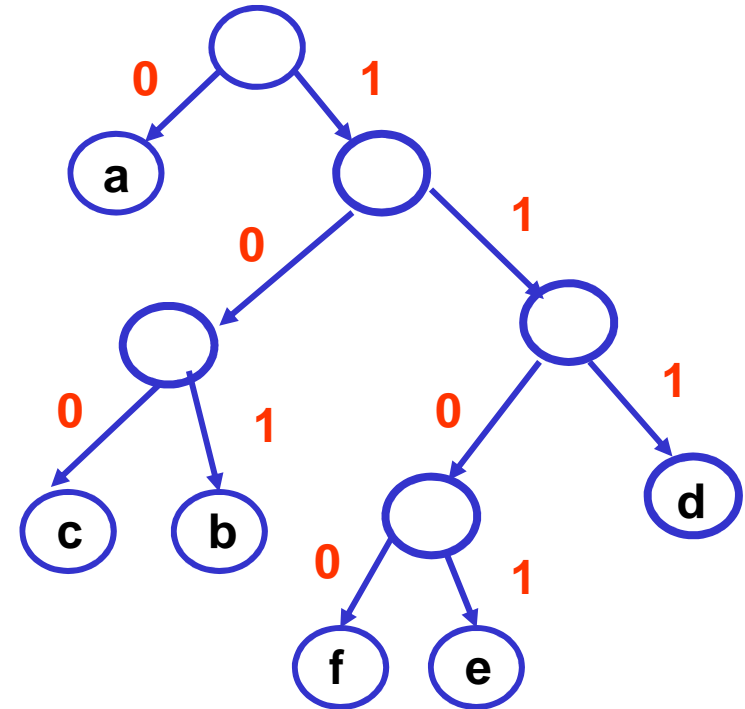
**Rappresentazione
ottima: albero
pienamente binario**



a	b	c	d	e	f
0	101	100	111	1101	1100

L'albero ha tante **foglie** quanti sono i caratteri dell'alfabeto

L'algoritmo di decodifica trova un cammino dalla radice ad una foglia per ogni carattere riconosciuto



I codici di Huffman

Problema: dato un alfabeto e la frequenza dei suoi caratteri, costruire un codice ottimo (che minimizza la lunghezza in bit delle codifiche)

Algoritmo di Huffman

Costruisce l'albero binario in modo bottom-up

È un algoritmo **greedy**

algoritmo di Huffman

Gestisce un foresta di alberi

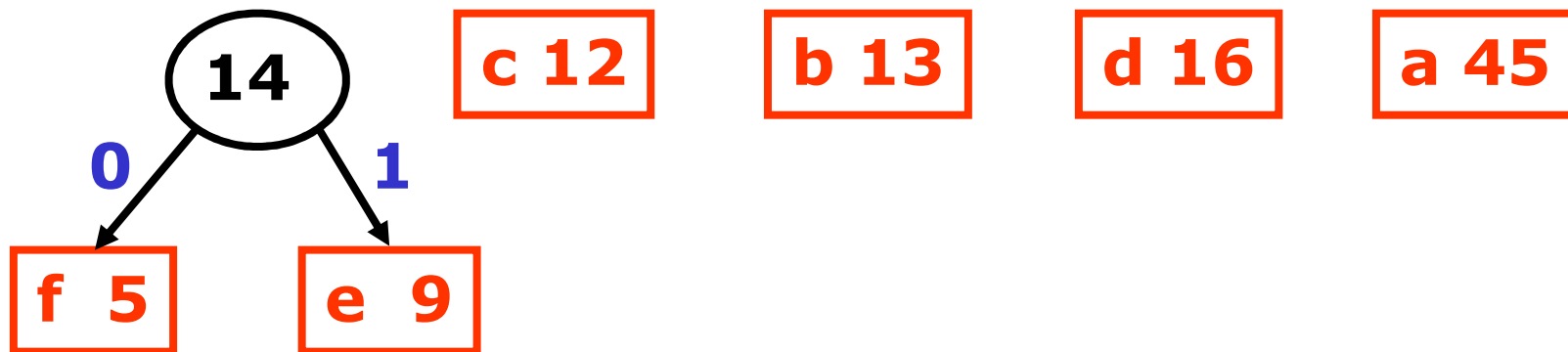
All'inizio ci sono n alberi di un solo nodo con le frequenze dei caratteri.

Ad ogni passo

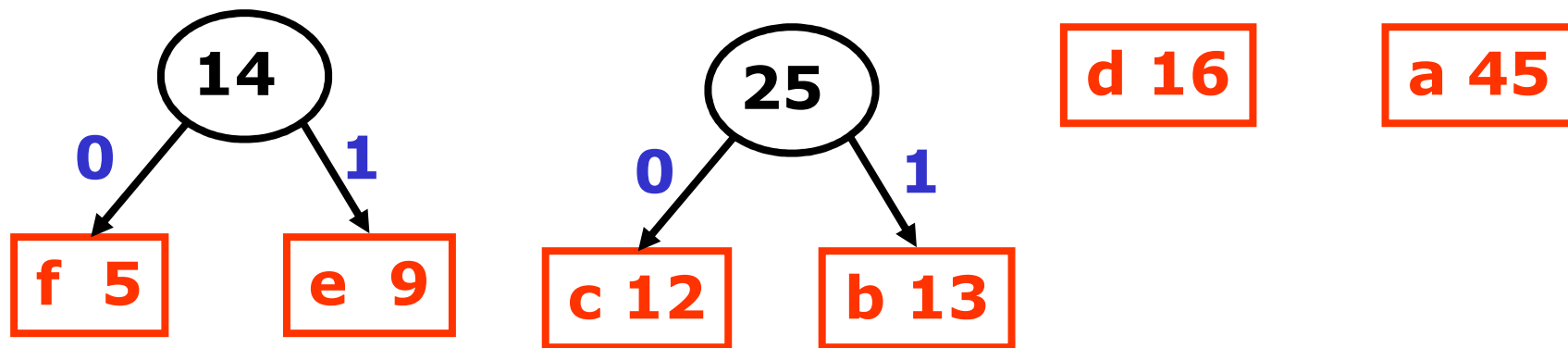
- **vengono fusi i due alberi con *radice minore* introducendo una *nuova radice* avente come etichetta la somma delle due *radici***

algoritmo di Huffman

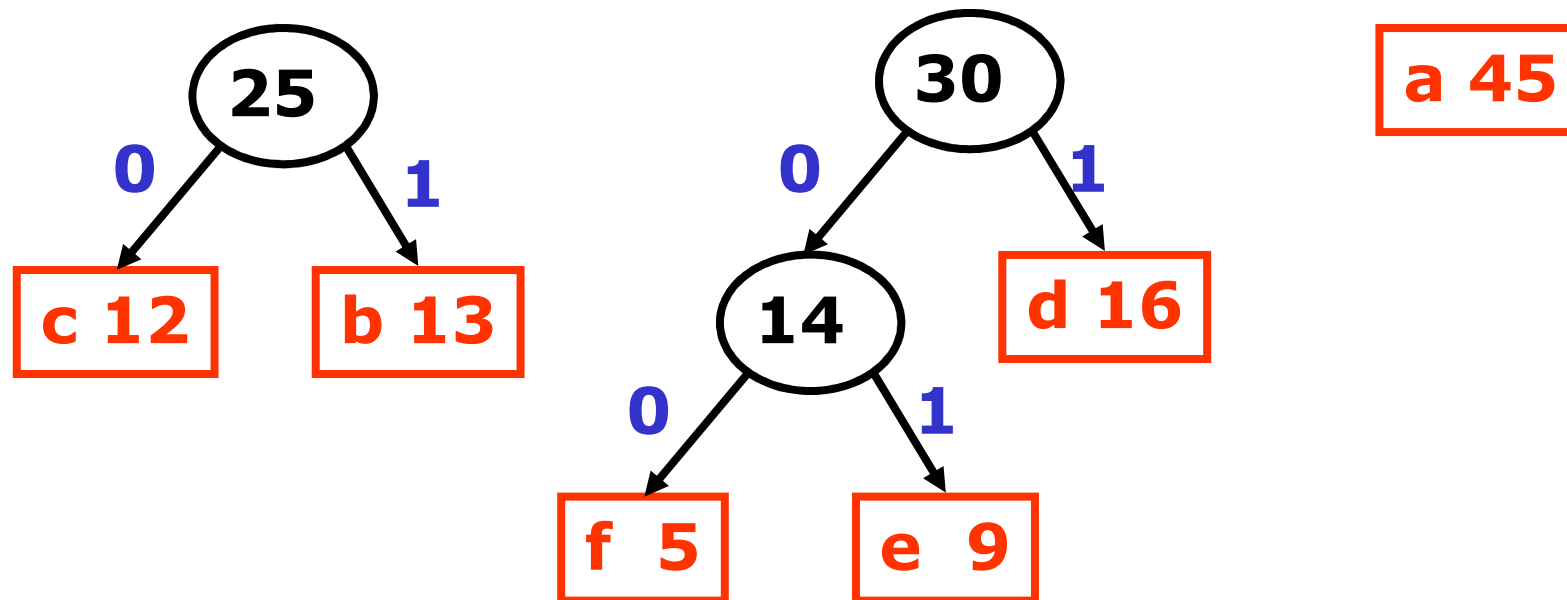
f 5 **e 9** **c 12** **b 13** **d 16** **a 45**



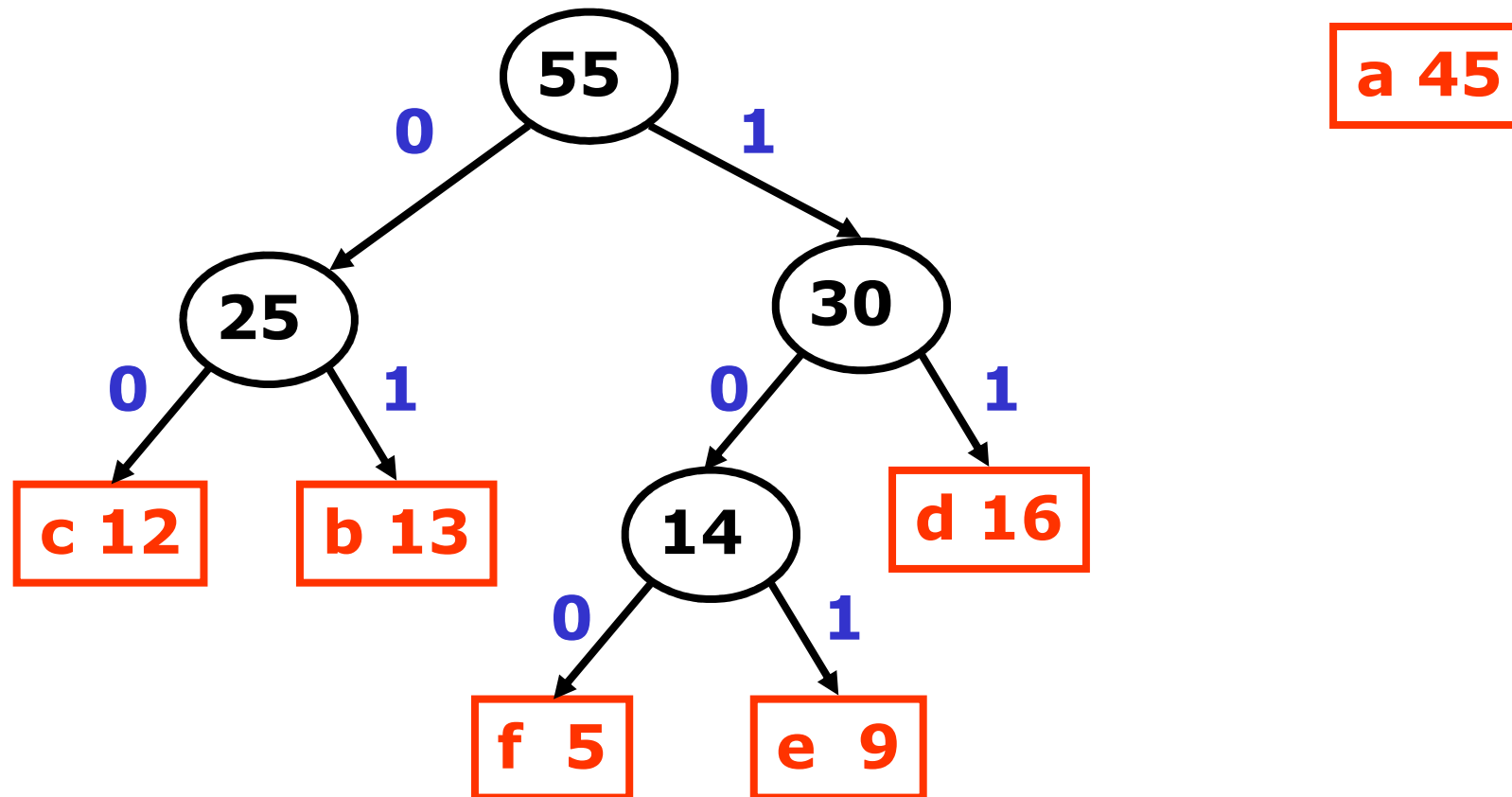
algoritmo di Huffman



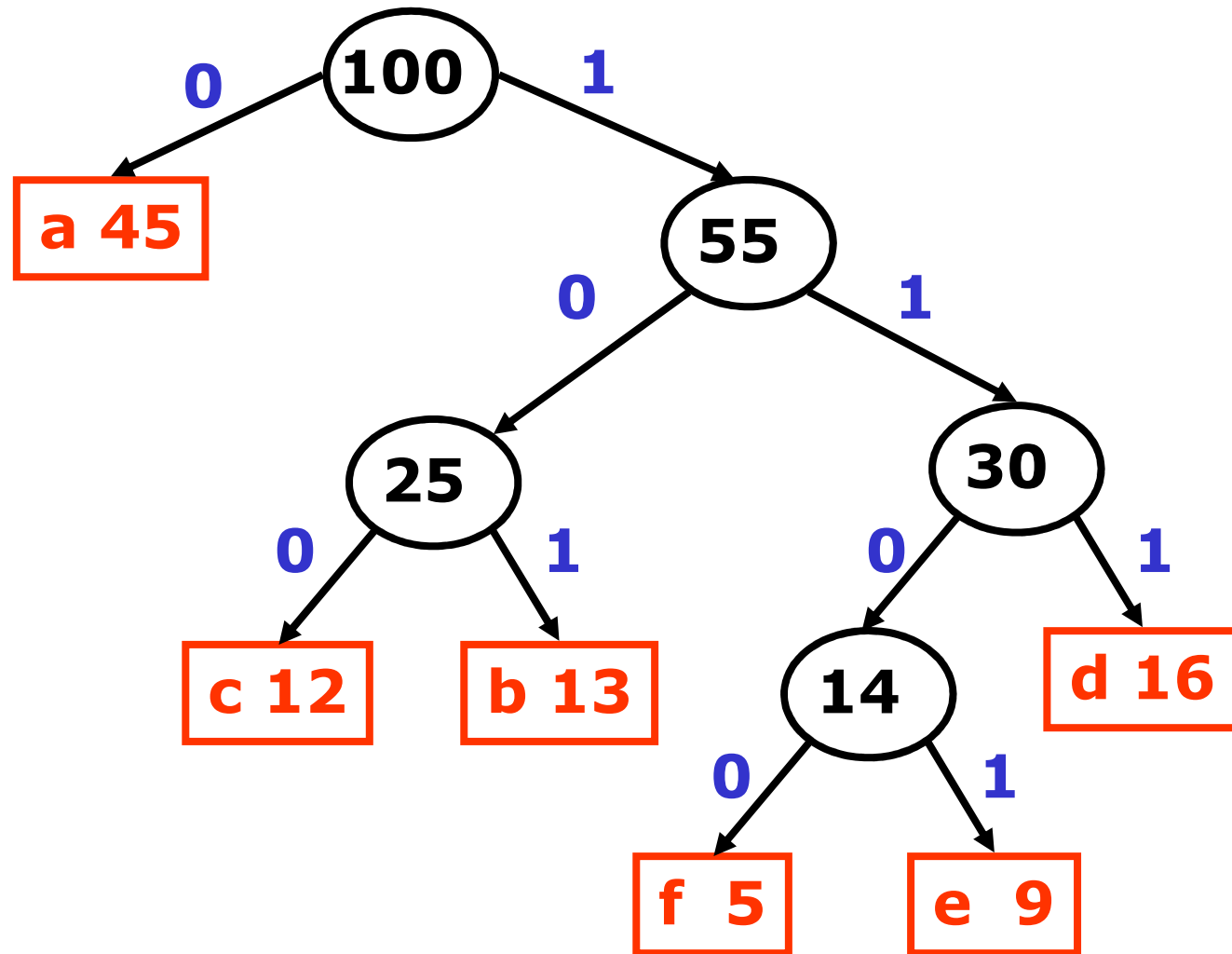
algoritmo di Huffman



algoritmo di Huffman



algoritmo di Huffman



algoritmo di Huffman: complessità

Gli alberi sono memorizzati in uno **heap (con ordinamento inverso : la radice è il più piccolo)**

Si fa un ciclo dove in ogni iterazione:

- **vengono estratti i due alberi con **radice minore****
- **vengono fusi in un nuovo albero avente come etichetta della radice la somma delle due **radici****
- **l'albero risultante e' inserito nello heap**

il ciclo ha **n iterazioni ed ogni iterazione ha complessità **$O(\log n)$** (si eseguono 2 estrazioni e un inserimento)**

$O(n \log n)$

La scelta locale è consistente con la situazione globale:

sistemando prima i nodi con minore frequenza, questi apparterranno ai livelli più alti dell'albero

Grafi

Grafi orientati

GRAFO ORIENTATO = (N , A)

N = insieme di **nodi**

$A \subseteq N \times N$ = insieme di **archi** (**coppie ordinate di nodi**)

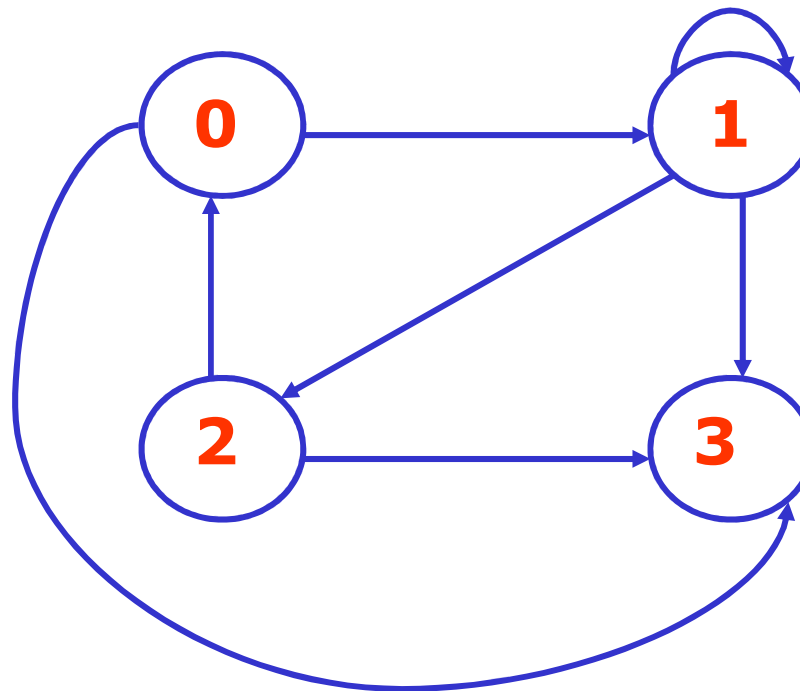
- predecessore
- successore
- cammino (sequenza di nodi-lunghezza = numero di archi)
- ciclo
- grafo aciclico

$n = |N|$ numero dei nodi

$m = |A|$ numero degli archi.

Un grafo orientato con **n** nodi ha al massimo **n^2** archi

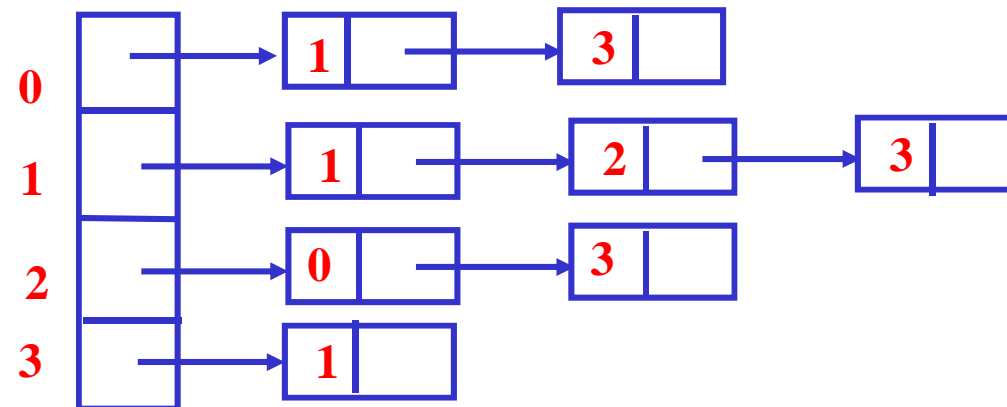
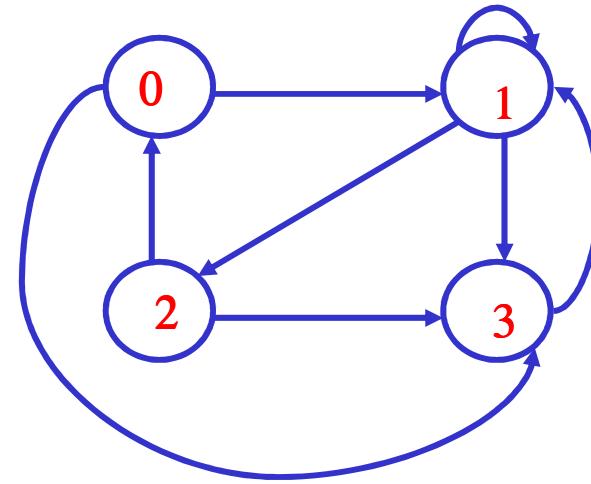
esempio



rappresentazione in memoria dei grafi: **liste di adiacenza**

```
struct Node{  
    int NodeNumber;  
    Node * next;  
};
```

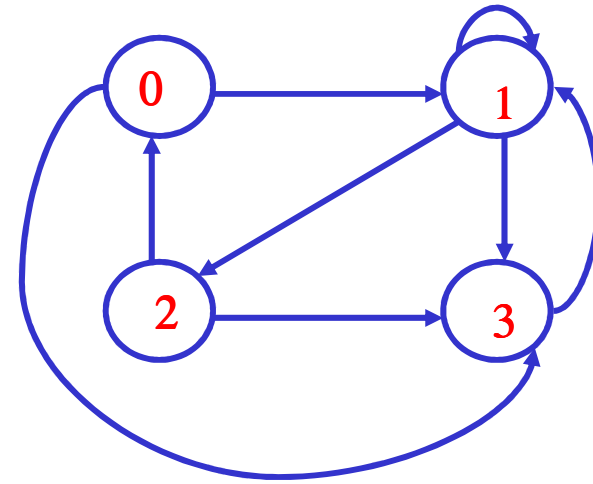
```
Node *graph[N];
```



rappresentazione in memoria dei grafi: **matrici di adiacenza**

```
int graph [N][N];
```

	0	1	2	3
0	0	1	0	1
1	0	1	1	1
2	1	0	0	1
3	0	1	0	0



Con nodi e archi etichettati : Liste di adiacenza

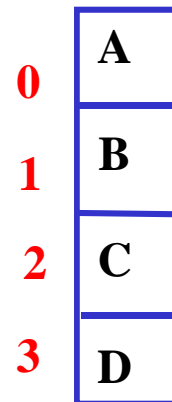
```
struct Node{
    int NodeNumber;
    ArcType arcLabel;
    Node * next;
};
```

```
Node * graph[N];
```

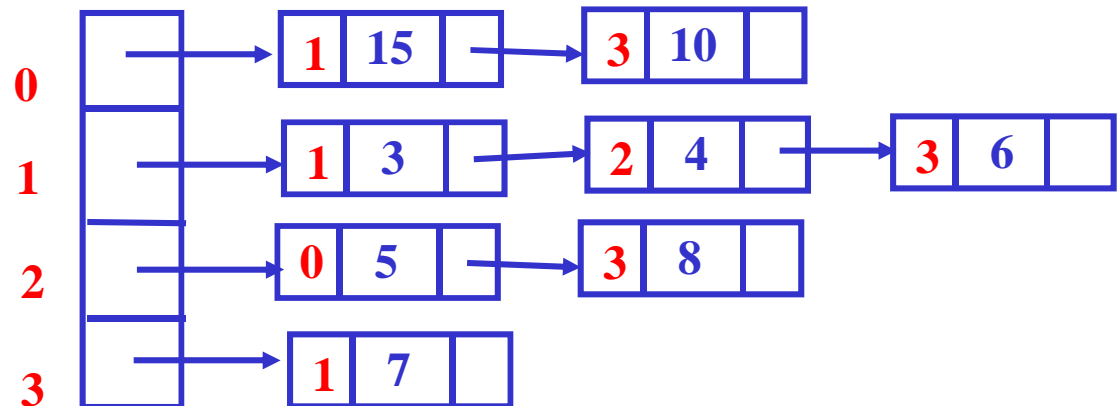
```
NodeType nodeLabels [N];
```

NodeType = char

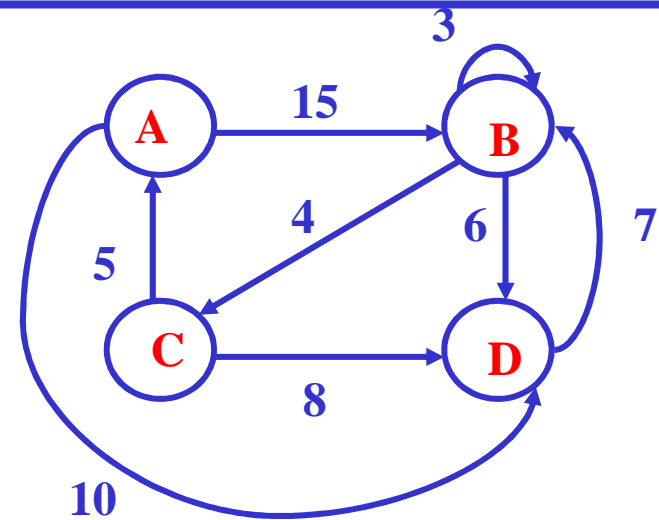
ArcType=int



nodeLabels



graph



Con nodi e archi etichettati : **matrici di adiacenza**

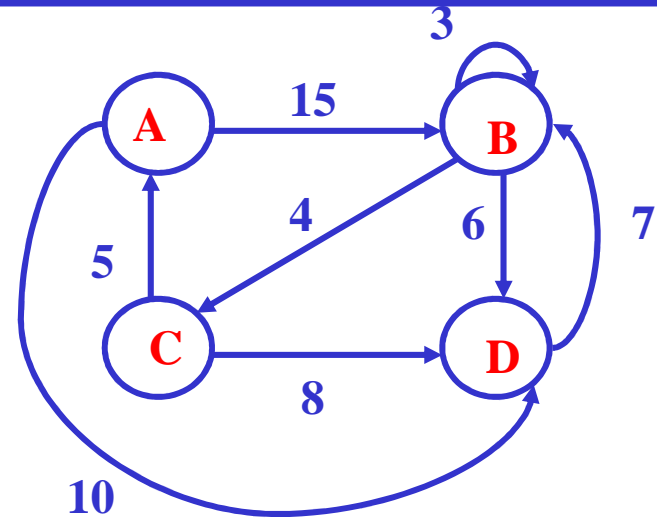
ArcType graph [N][N];

NodeType nodeLabels [N];

0	A
1	B
2	C
3	D

nodeLabels

	0	1	2	3
0	0	15	0	10
1	0	3	4	6
2	5	0	0	8
3	0	7	0	0



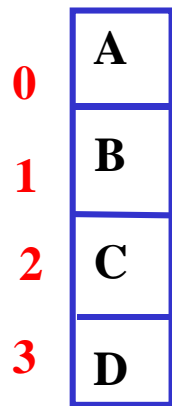
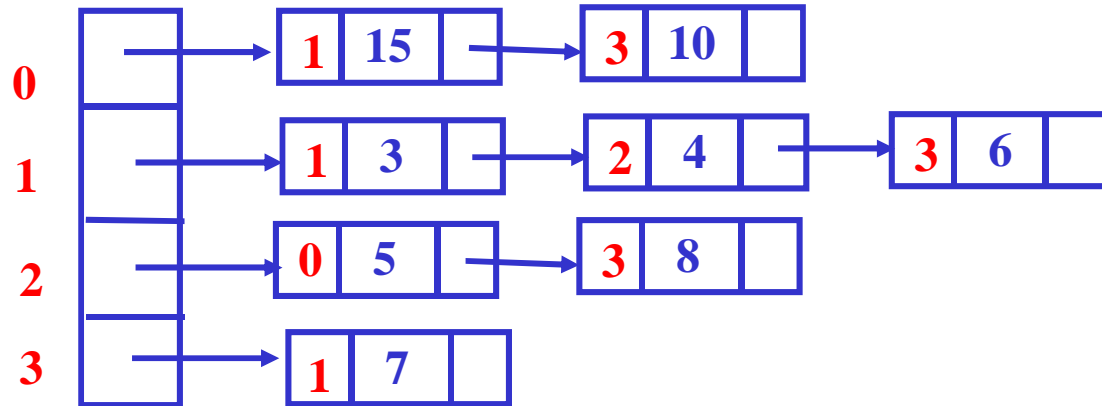
visita in profondità

```
void NodeVisit (nodo) {  
    esamina il nodo;  
    marca il nodo;  
    applica NodeVisit ai successori non marcati del nodo;  
}
```

```
Void DepthVisit Graph(h) {  
    per tutti i nodi:  
        se il nodo non è marcato applica nodeVisit;  
}
```

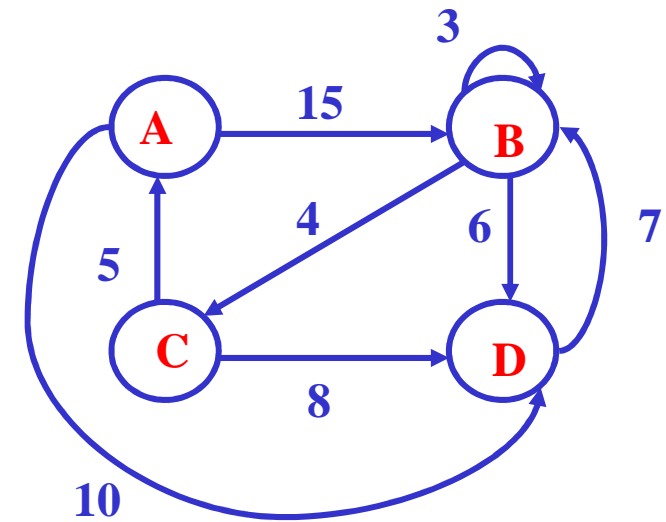
$O(m) + O(n)$

visita in profondità: esempio



0 1 2 3

A B C D



Una classe per i grafi

```
class Graph{
  struct Node {
    int nodeNumber;
    Node* next;
  };
  Node* graph [N];
  NodeType nodeLabels [N];
  int mark[N];
  void nodeVisit( int i) {
    mark[i]=1;
    <esamina nodeLabels[i]>;
    Node* g; int j;
    for (g=graph[i]; g; g=g->next){
      j=g->nodeNumber;
      if (!mark[j]) nodeVisit(j);
    }
  }
}
```

```
public:
  void depthVisit() {
    for (int i=0; i<N; i++)
      mark[i]=0;
    for (i=0; i<N; i++)
      if (! mark[i])
        nodeVisit (i);
  }
  ..
};
```

Grafi non orientati

grafo non orientato = (N , A),

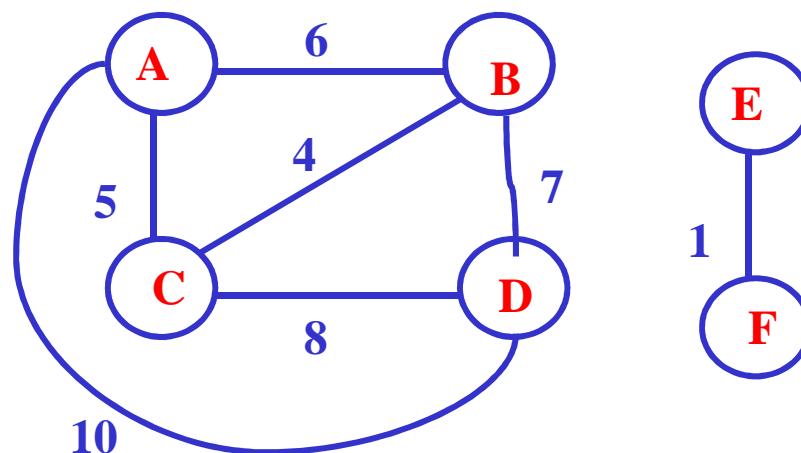
N = insieme di **nodi**

A = insieme di **coppie non ordinate di nodi**

- **nodi adiacenti**
- **ciclo (almeno 3 nodi)**

Un grafo non orientato con n nodi ha al massimo $n(n-1)/2$ archi

Esempio di grafo non orientato



**Come un grafo orientato considerando
che ogni connessione corrisponde a due
archi orientati nelle due direzioni
opposte**

Minimo albero di copertura

- Un grafo non orientato è **connesso** se esiste un cammino fra due nodi qualsiasi del grafo
- **Componente connessa**: sottografo connesso
- Componente connessa **massimale**: nessun nodo è connesso ad un'altra componente connessa
- **Albero di copertura**: insieme di componenti connesse massimali **acicliche**
- **Minimo albero di copertura**: la somma dei pesi degli archi è minima

algoritmo di **Kruskal** per trovare il minimo albero di coperura

1. Elenca gli archi del grafo in ordine crescente, considera una componente per nodo

2. Scorri l'elenco degli archi
per ogni arco **a**:

if (a connette due componenti non connesse), unifica le componenti

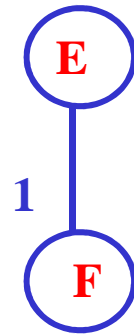
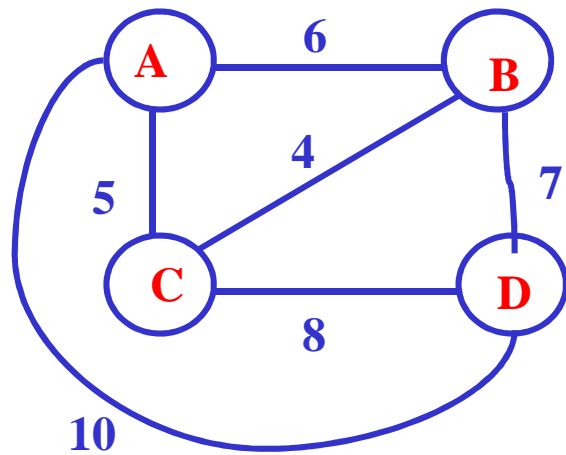
Complessità:

1. $O(m \log m)$

1. Numero iterazioni: $O(m)$

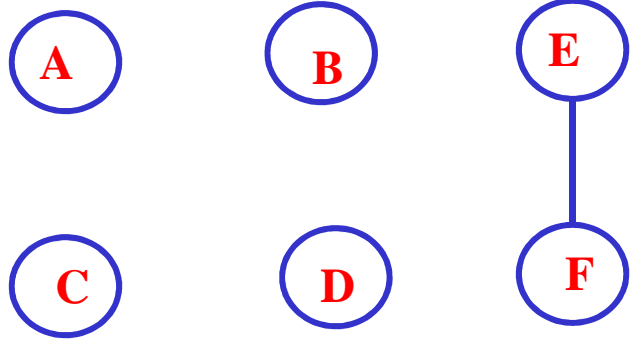
2. Controllo e unificazione : $O(\log n)$

$$O(m \log m) + O(m \log n) \qquad O(m \log n) \quad (m \leq n^2)$$

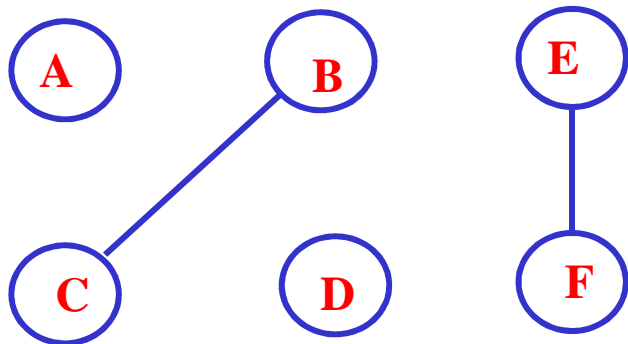


(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

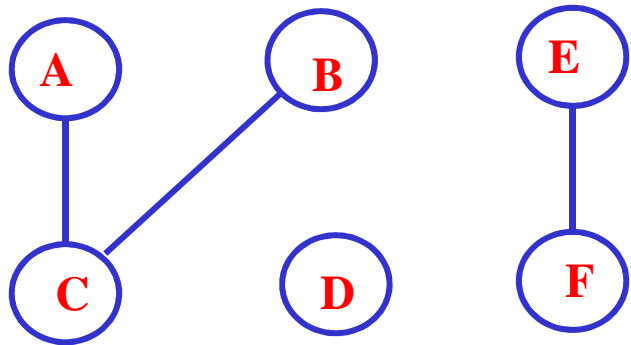




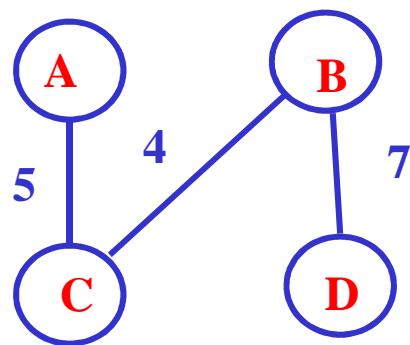
(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

Lunghezza: 17

Implementazione Kruskal

I nodi sono numerati

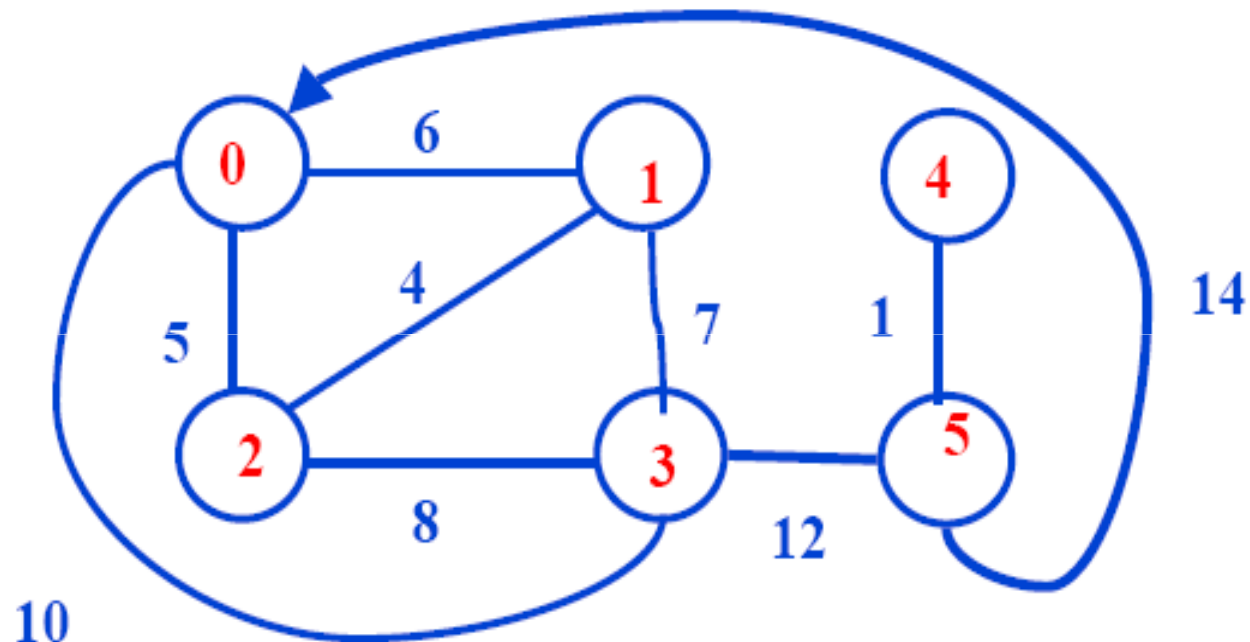
Le componenti sono memorizzate come **insiemi di alberi**

Sono memorizzate in **array: ogni nodo **punta al padre****

Se due nodi appartengono alla stessa componente risalendo si incontra **un antenato comune**

Due alberi sono unificati inserendo quello meno profondo come sottoalbero della radice di quello più profondo

Implementazione Kruskal



Implementazione Kruskal

0 1 4

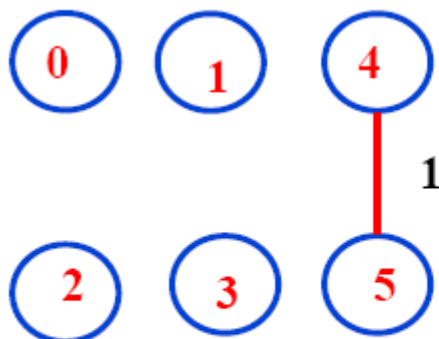
2 3 5

0	1	2	3	4	5
-	-	-	-	-	-
0	0	0	0	0	0

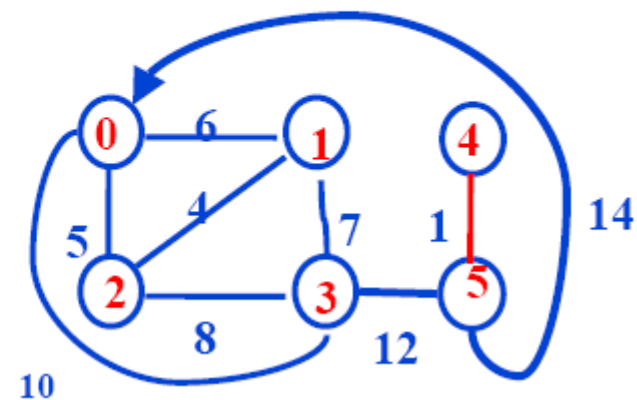
livello

0 1 2 3 4 5

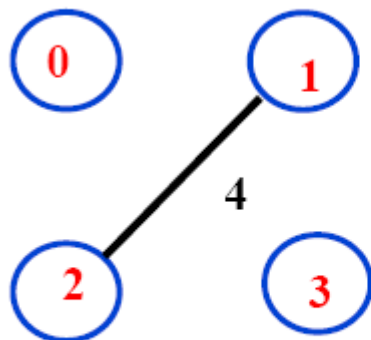
Implementazione Kruskal



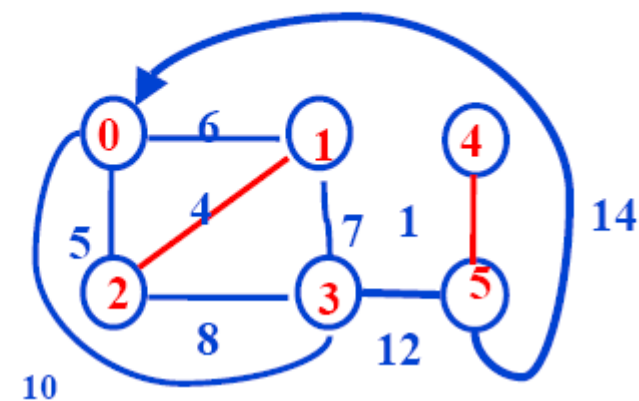
0	1	2	3	4	5
-	-	-	-	-	4
0	0	0	0	1	0



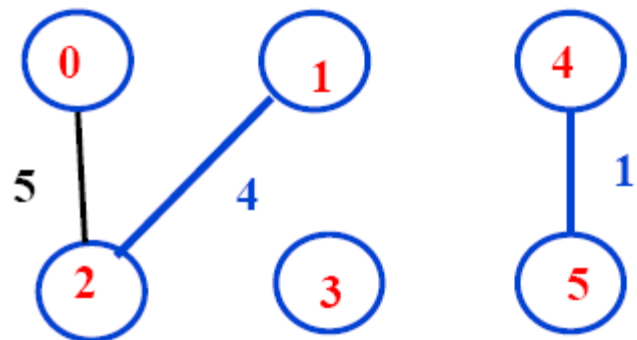
Implementazione Kruskal



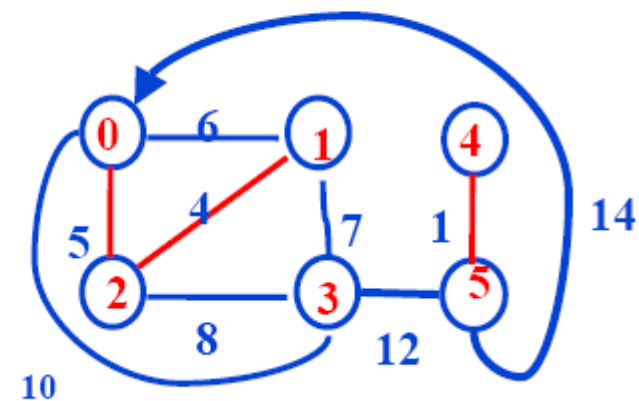
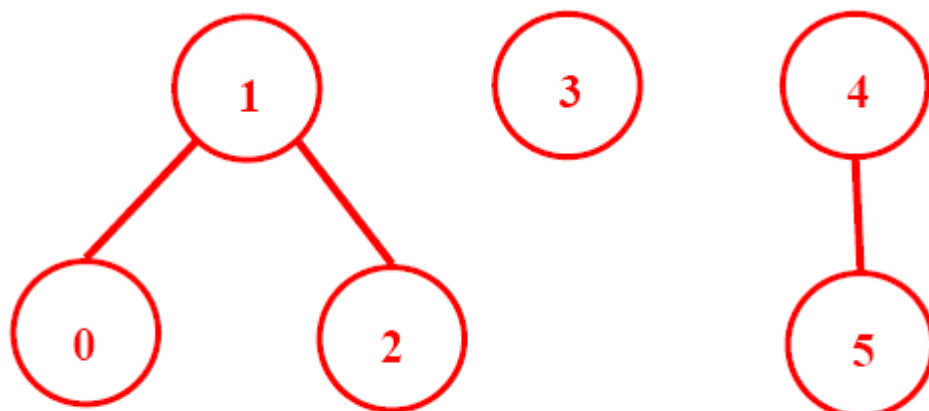
0	1	2	3	4	5
-	-	1	-	-	4
0	1	0	0	1	0



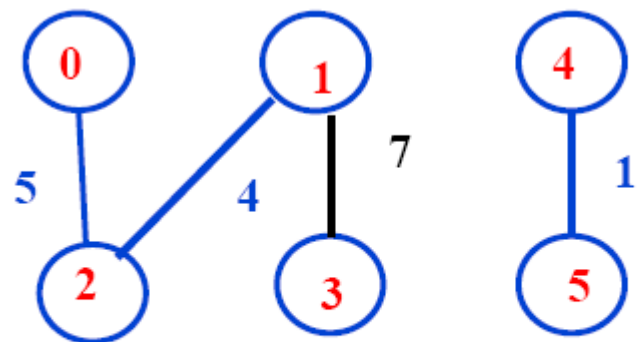
Implementazione Kruskal



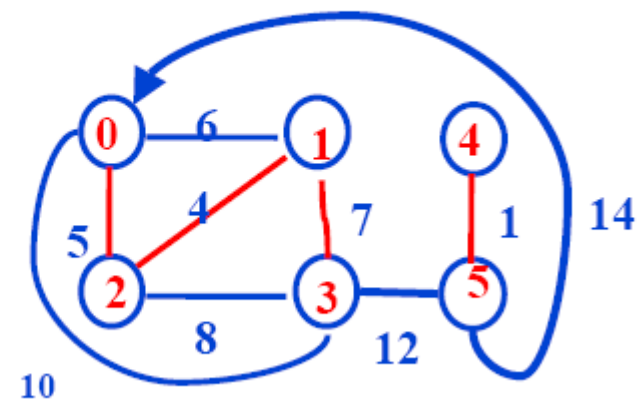
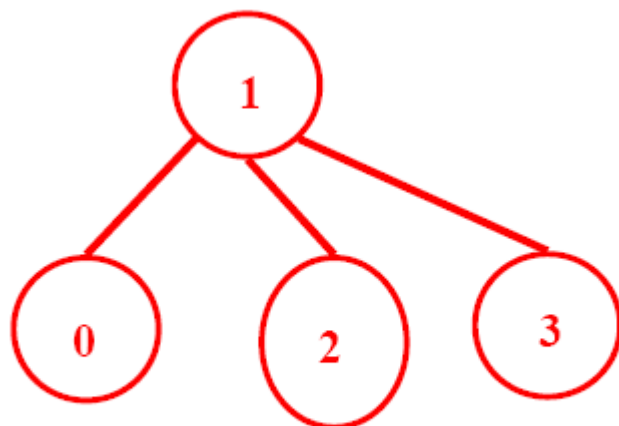
0	1	2	3	4	5
1	-	1	-	-	4
0	1	0	0	1	0



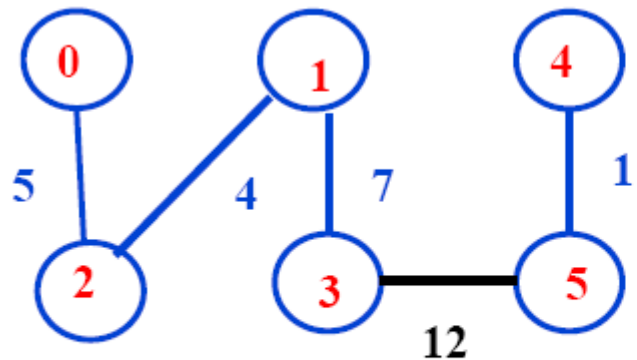
Implementazione Kruskal



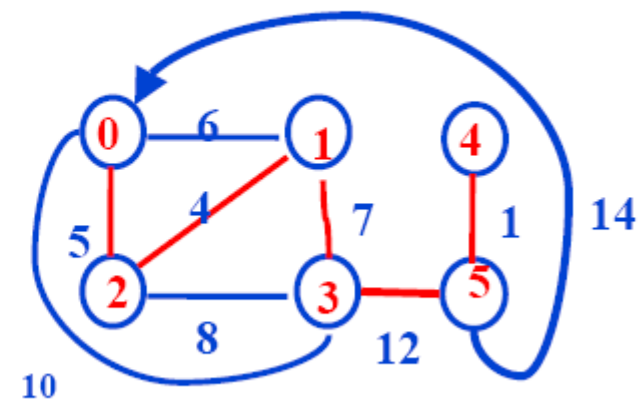
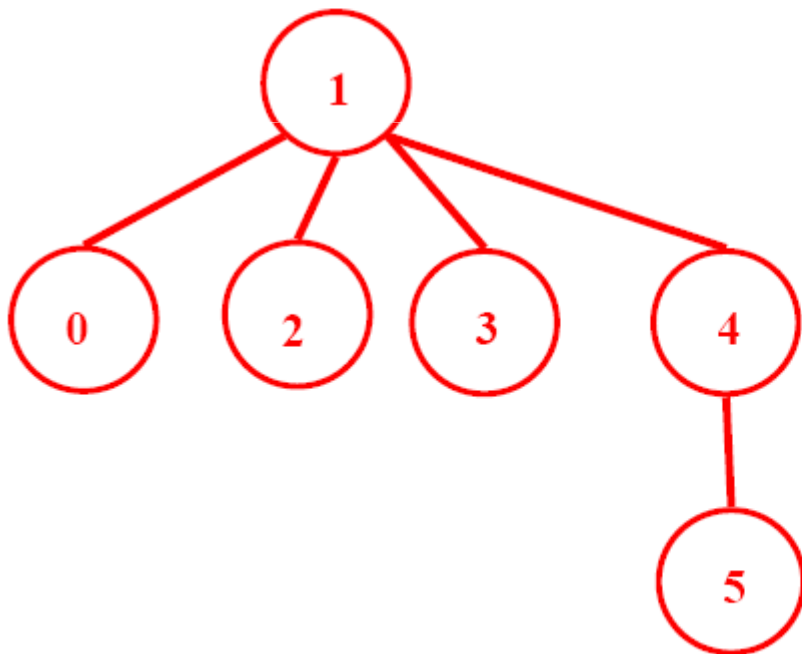
0	1	2	3	4	5
1	-	1	1	-	4
0	1	0	0	1	0



Implementazione Kruskal



0	1	2	3	4	5
1	-	1	1	1	4
0	2	0	0	1	0



algoritmo di Dijkstra

Si applica ai grafi orientati

Trova i cammini minimi da un nodo a tutti gli altri

Basato sulla metodologia greedy

algoritmo di Dijkstra

Utilizza due tabelle **dist (distanza) e **pred** (predecessore) con **n** elementi**

Esegue **n passi ; ad ogni passo :**

si sceglie il nodo con distanza minore in **dist**

si aggiornano **pred e **dist** per i suoi immediati successori**

algoritmo di Dijkstra

```
1  Q = N;
2  per ogni nodo p diverso da p0 {                               // O(n)
    dist(p)=infinito, pred(p)=vuoto;
}
    dist(p0)=0;
4  while (Q contiene più di un nodo) {
5      estrai da Q il nodo p con minima dist(p); // O(logn)
6      per ogni nodo q successore di p {
          lpq=lunghezza dell'arco (p,q);
          if (dist(p)+lpq < dist(q)) {
              dist(q)=dist(p)+lpq;
              pred(q)=p;
7          re-inserisci in Q il nodo q // O(logn)
              modificato;
          }
      }
}
```

algoritmo di Dijkstra

$C[1+2] : O(n)$

$C[5]=C[7] : O(\log n)$

(i valori di dist sono memorizzati in uno heap)

Numero iterazioni del ciclo while : n

Complessità iterazione: $C[5] + m/n C[7]$

$= O(\log n + (m/n) \log n)$

Complessità del ciclo: $O(n(\log n + (m/n) \log n)) =$

$O(n \log n + m \log n) = O(m \log n)$ se $m > n$

Perchè l'algoritmo di Dijkstra funziona

In ogni iterazione del ciclo i nodi già scelti (eliminati da Q) sono "sistemati":

- per i nodi già scelti **dist** contiene la lunghezza del cammino minimo e **pred** permette di ricostruirlo.
- Il cammino minimo per i nodi già scelti passa soltanto da nodi già scelti

esempio

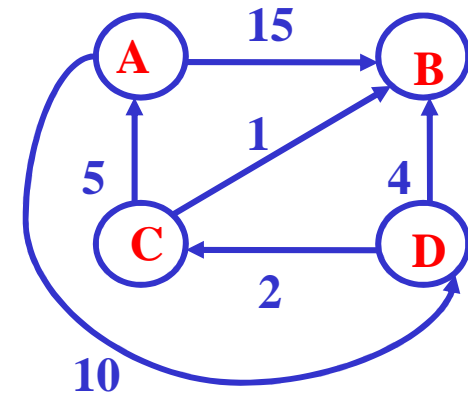
$Q = \{A, B, C, D\}$

dist

pred

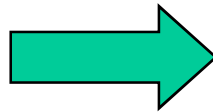
A	B	C	D
0	inf.	inf.	inf.

A	B	C	D
--	--	--	--



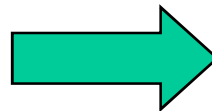
estraggo A: $\text{dist}(A)=0$

$\text{dist}(A) + |(A,B)| < \text{dist}(B)$
 $0 + 15 < \text{inf.}$



$\text{dist}(B)=15, \text{pred}(B)=A$

$\text{dist}(A) + |(A,D)| < \text{dist}(D)$
 $0 + 10 < \text{inf.}$



$\text{dist}(D)=10, \text{pred}(D)=A$

A	B	C	D
0	15	inf.	10

A	B	C	D
--	A	--	A

$Q = \{B, C, D\}$

esempio

$Q = \{ B, C, D \}$

dist

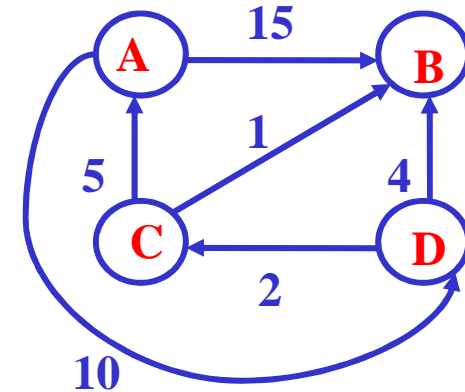
pred

A B C D

A B C D

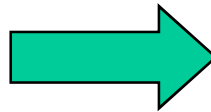
0	15	inf.	10
---	----	------	----

--	A	--	A
----	---	----	---



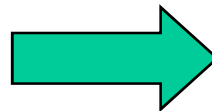
estraggo D: $\text{dist}(D)=10$

$\text{dist}(D) + |(D,B)| < \text{dist}(B)$
 $10 + 4 < 15$



$\text{dist}(B)=14, \text{pred}(B)=D$

$\text{dist}(D) + |(D,C)| < \text{dist}(C)$
 $10 + 2 < \text{inf.}$



$\text{dist}(C)=12, \text{pred}(C)=D$

A B C D

A B C D

0	14	12	10
---	----	----	----

--	D	D	A
----	---	---	---

$Q = \{ B, C \}$

esempio

$Q = \{ B, C \}$

dist

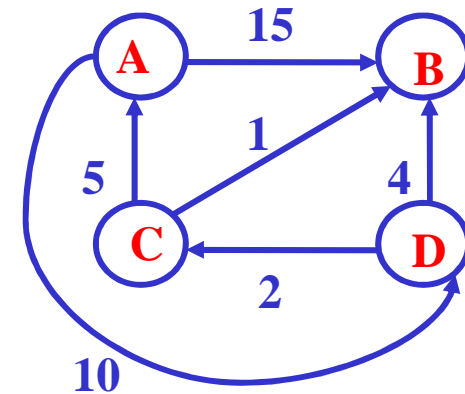
A B C D

0	14	12	10
---	----	----	----

pred

A B C D

--	D	D	A
----	---	---	---



estraggo C: dist(C)=12

$\text{dist}(C) + |(C,A)| < \text{dist}(A)$
 $12 + 5 < 0$?

$\text{dist}(C) + |(C,B)| < \text{dist}(B)$
 $12 + 1 < 14$

NO



dist(B)=13, pred(B)=C

A B C D

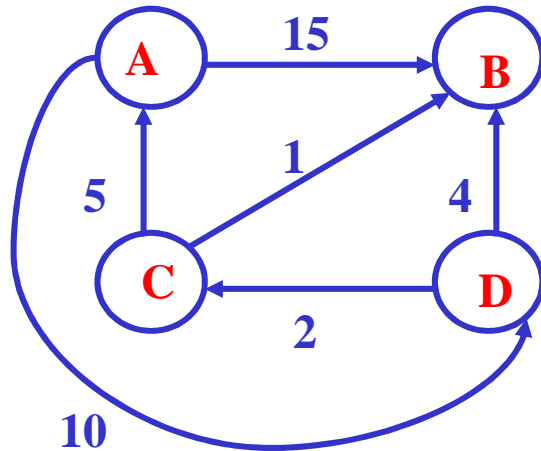
0	13	12	10
---	----	----	----

A B C D

--	C	D	A
----	---	---	---

$Q = \{ B \}$

soluzione



da A a B: A->D->C ->B **lung=13**

da A a C: A->D->C **lung=12**

da A a D: A->D **lung=10**

	A	B	C	D
A	0	i	i	i
D	0	15	i	10
C	0	14	12	10
	0	13	12	10

dist

	A	B	C	D
A	-	-	-	-
D	-	A	-	A
C	-	D	D	A
	-	C	D	A

pred

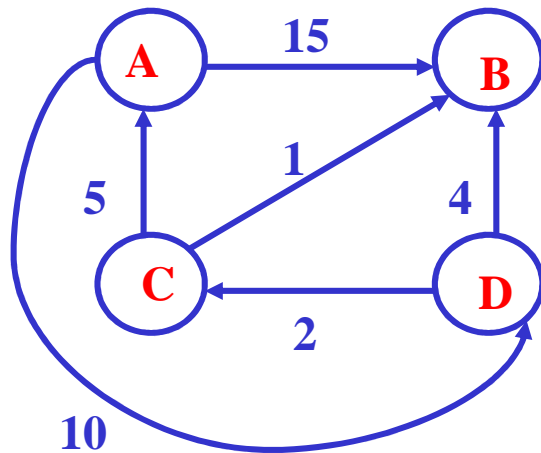
Q = {A, B, C, D}

Q = {B, C, D}

Q = {B, C}

Q = {B}

soluzione



da A a B: A->D->C ->B

lung=13

da A a C: A->D->C

lung=12

da A a D: A->D

lung=10

		A	B	C	D
	Q = {A, B, C, D}	0 /-	i /-	i /-	i /-
A	Q = {B, C, D}	0 /-	15/A	i /-	10/A
D	Q = {B, C}	0 /-	14/D	12/D	10/A
C	Q = {B}	0/-	13/C	12/D	10/A

dist/pred

Cenni alla NP-completezza

Problemi risolubili con complessità esponenziale

Commesso viaggiatore

Date n città, è possibile partire da una città, attraversare ogni città esattamente una volta e tornare alla città di partenza, percorrendo una distanza complessiva non superiore a un intero k ?

n regine

Data una scacchiera con $n \times n$ caselle, è possibile posizionare su di essa n regine in modo che nessuna possa "mangiare" un'altra?

La complessità è esponenziale in n

P_S

P_S : soddisfattibilità di una formula logica

Data una formula F con n variabili, esiste una combinazione di valori booleani che, assegnati alle variabili di F , la rendono vera?

Es.

$F = (x \text{ and not } x)$ $n=1$ non sodd.

$F = (x \text{ or } y) \text{ or } (\text{not } x \text{ and } y) \text{ or } z$ $n=3$ sodd.

soddisfattibilità di una formula logica

ALGORITMO
provare tutte le combinazioni

**Se le variabili che compaiono nella formula sono n ,
le combinazioni da provare sono 2^n**



La complessità è esponenziale: $O(2^n)$

Algoritmi **nondeterministici**

Si aggiunge il comando

choice(I)

dove **I** è un insieme

choice(I) sceglie nondeterministicamente un elemento dell'insieme **I**

Un algoritmo nondeterministico per la **soddisfattibilità**

```
int nsat(Formula f,int *a,int n) {  
    for (int i=0; i < n; i++)  
        a[i]=choice({0,1});  
    if (value(f,a))  
        return 1;  
    else  
        return 0;  
}
```

$O(n)$

Ritorna 1 se esiste almeno una scelta che con risultato 1

Un algoritmo nondeterministico di **ricerca** in array

```
int nsearch(int* a, int n, int x) {  
    int i=choice({0..n-1});  
    if (a[i]==x)  
        return 1;  
    else  
        return 0;  
}
```

$O(1)$

Un algoritmo nondeterministico di **ordinamento**

```
int nsort(int* a, int n) {  
    int b [n];  
    for (int i=0; i<n; i++)  
        b[i]=a[i];  
    for (int i=0; i<n; i++)  
        a[i]=b[choice({0..n-1})];  
    if (ordinato(a))  
        return 1;  
    return 0;  
}
```

$O(n)$

Relazione fra determinismo e nondeterminismo

Per ogni algoritmo **nondeterministico** ne esiste uno **deterministico** che lo **simula**, esplorando lo spazio delle soluzioni, fino a trovare un successo.

Se le soluzioni sono in numero esponenziale, l'algoritmo **deterministico** avrà complessità esponenziale.

Un algoritmo nondeterministico di ricerca in array

P = insieme di tutti i problemi decisionali risolubili in tempo polinomiale con un algoritmo **deterministico**

NP = insieme di tutti i problemi decisionali risolubili in tempo polinomiale con un algoritmo **nondeterministico**

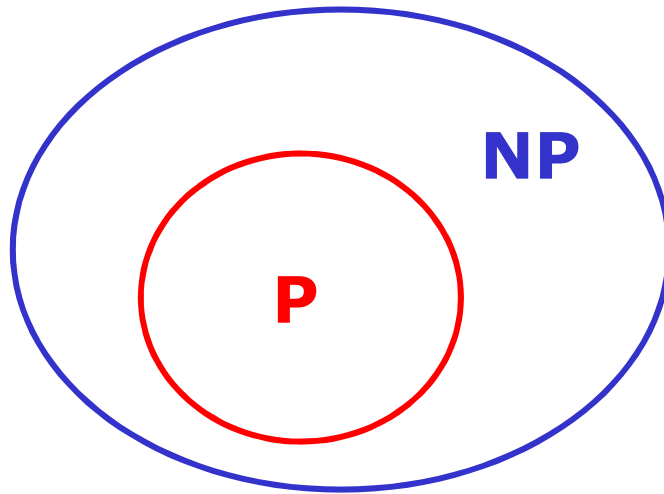
NP : **N**ondeterministico **P**olinomiale

Un algoritmo nondeterministico di ricerca in array

P = { ricerca, ordinamento, ... }

NP = { ricerca, ordinamento, soddisfattibilità,
colorazione mappe, ... }

Un algoritmo nondeterministico di ricerca in array

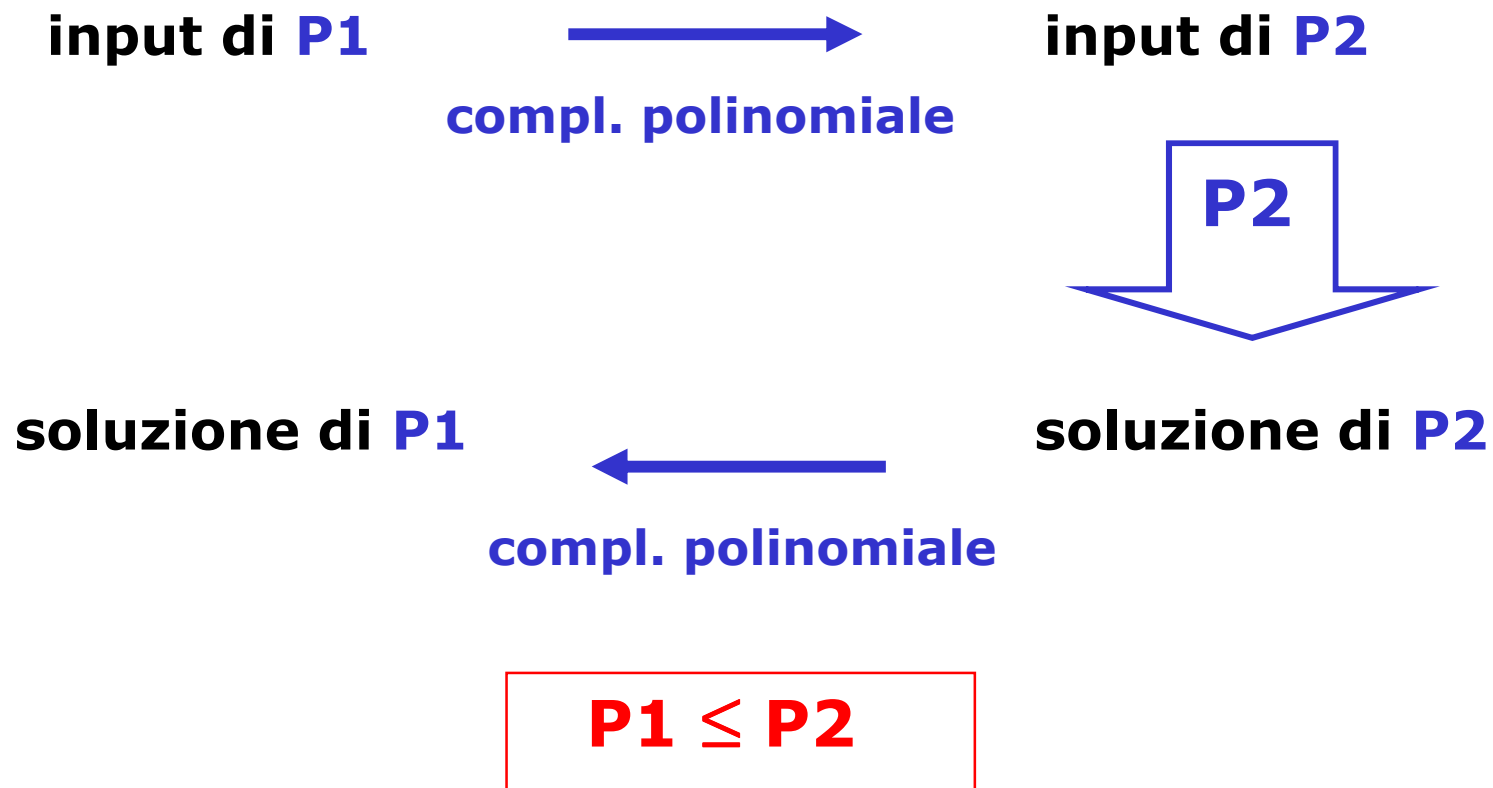


$$P \subseteq NP$$

$$P = NP ?$$

riducibilità

Un problema **P1** **si riduce** a un altro problema **P2** se ogni soluzione di **P1** può ottenersi deterministicamente in tempo **polinomiale** da una soluzione di **P2**



riducibilità

- **$P1 \leq P2$**
- **$P2$ è risolubile in tempo polinomiale**



$P1$ è risolubile in tempo polinomiale

Teorema di Cook

Per qualsiasi problema **R in NP** vale che **R** è riducibile al problema della soddisfattiabilità della formula logica

$$R \leq P_S$$



Se si trovasse un algoritmo polinomiale per **P_S** allora tutti i problemi in NP sarebbero risolubili in tempo polinomiale e quindi **P sarebbe uguale ad NP**

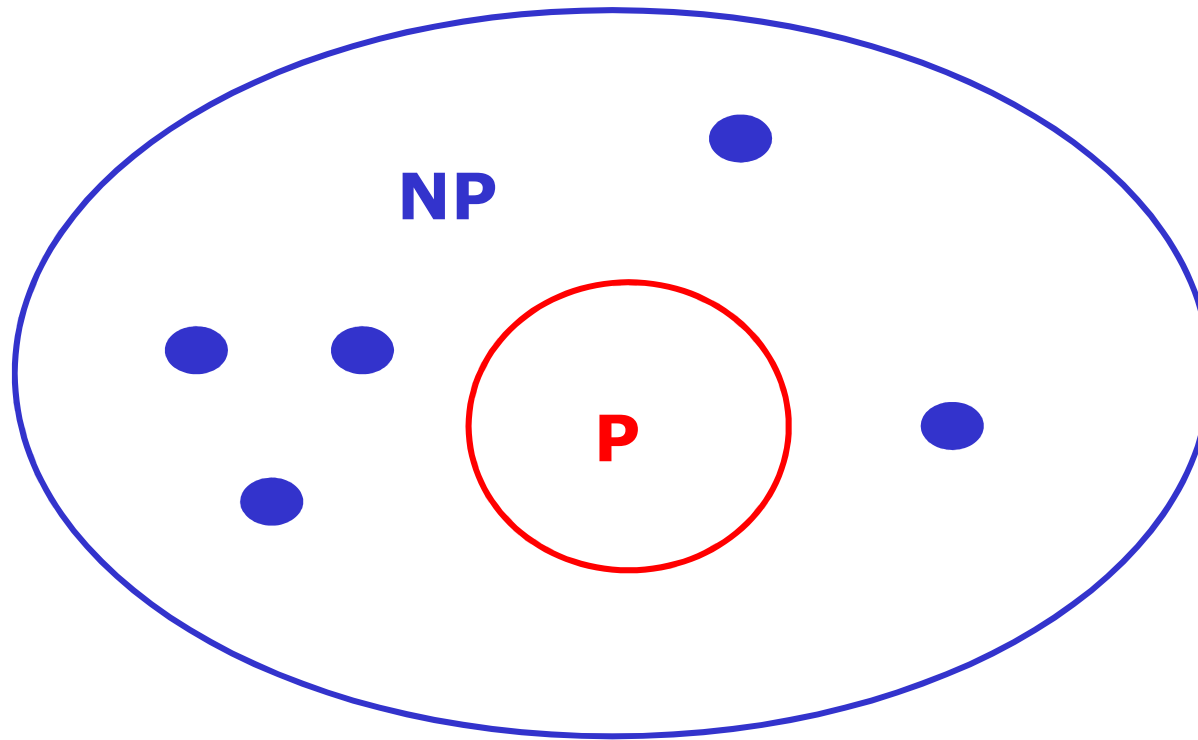
NP-completezza

Un problema **R** è **NP-completo** se

- **R** appartiene ad **NP**; e
- **P_S** ≤ **R**

Se si trovasse un algoritmo polinomiale per un problema **NP-completo**, allora tutti i problemi in NP sarebbero risolvibili in tempo polinomiale e quindi **P sarebbe uguale ad NP**

Problemi NP-completi



Problemi NP-completi

E' stato dimostrato che i seguenti problemi e altri sono NP-completi:

- **Commesso viaggiatore**
- **Colorazione di mappe**
- **Zaino**
- **n regine**

Quindi uno qualsiasi di questi problemi può essere usato al posto di P_S nella dimostrazione di NP-completezza

Utilizzo

Per dimostrare che un problema **R è NP-completo:**

- **R appartiene ad NP**
individuare un algoritmo polinomiale
nondeterministico per risolvere P
- **esiste un problema NP-completo che si riduce a R**
se ne sceglie uno fra i problemi NP-completi
noti che sia facilmente riducibile a R

Utilizzo

Perché ci serve dimostrare che un problema è NP-completo?

Perché non riusciamo a risolverlo con un algoritmo polinomiale e vogliamo dimostrare che non ci si riesce a meno che P non sia uguale ad NP , problema tuttora non risolto

Caratterizzazione alternativa dei problemi NP-completi

problemi NP-completi: Problemi con certificato verificabile in tempo polinomiale

Certificato: soluzione del problema

Es: per il problema della soddisfattibilità della formula logica si può controllare se un assegnamento di valori booleani alle variabili è una soluzione

Problemi neanche NP-completi

Trovare tutte le permutazioni di un insieme

Torre di Hanoi

Problemi indecidibili

Tutti i problemi

