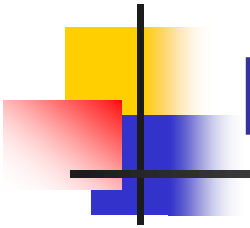
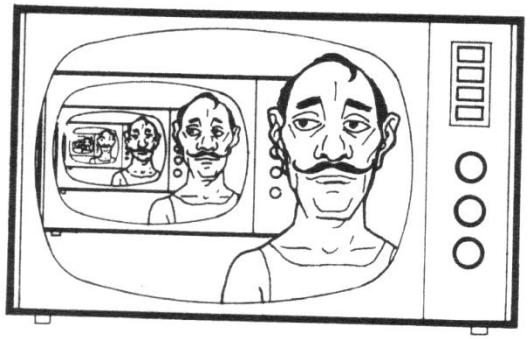
Ricorsione e paradigma divide et impera

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino



Definizione

- procedura *ricorsiva*:
  - all'interno della propria definizione chiamata alla procedura stessa (*ricorsione diretta*)
  - chiamata ad almeno una procedura la quale, direttamente o indirettamente, chiama la procedura stessa *(ricorsione indiretta)*
- algoritmo *ricorsivo*: si basa su procedure ricorsive.



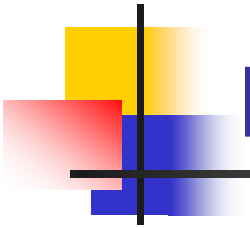
- La soluzione di un problema S applicato ai dati D è ricorsiva se si può esprimere come:

$$S(D) = f(S(D')) \qquad D \mathrel{!}= D_0$$

$$S(D_0) = S_0$$

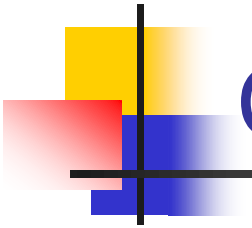
D' più semplice di D

condizione di terminazione



Motivazioni

- Natura di molti problemi:
  - risoluzione di sotto-problemi analoghi a quello di partenza (ma più piccoli)
  - combinazione di soluzioni parziali nella soluzione del problema originario
  - ricorsione come base del paradigma di problem-solving noto come **divide et impera**.
- Eleganza matematica della soluzione.



Condizione di terminazione

Ogni algoritmo deve terminare $\Rightarrow$ ricorsione finita.

Sottoproblemi semplici e risolvibili:

- banali (es.: insiemi di 1 solo elemento)
- esaurimento delle scelte lecite (es.: nel grafo è terminate la lista delle adiacenze).

Il Paradigma Divide et Impera

Divide

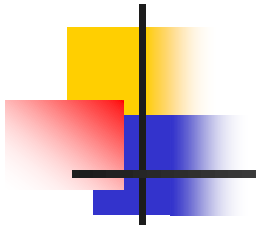
- da problema di dimensione n in $a \geq 1$ problemi *indipendenti* di dimensione $n' < n$

Impera

- risoluzione di problema elementare (condizione di terminazione)

Combina

- ricostruzione di soluzione complessiva combinando le soluzioni parziali.



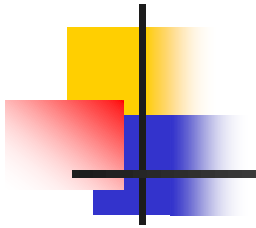
Condizione di terminazione

"a" sottoproblemi, ciascuno più piccolo del problema originale

Risolvi(Problema):

- Se il problema è elementare:
  - Soluzione = Risolvi_banale(Problema)
- Altrimenti:
  - $\text{Sottoproblema}_{1,2,3,\ldots,a}$ = Dividi(Problema) ;
  - Per ciascun Sottoproblema_i:
    - Sottosoluzione_i = Risolvi(Sottoproblema_i) ;
  - Return Soluzione = Combina($\text{Sottosoluzione}_{1,2,3,\ldots,a}$) ;

chiamata ricorsiva

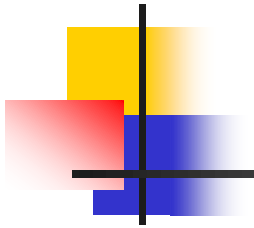


Valori di a

- a=1: ricorsione lineare
- a>1: ricorsione multi-via

Valori di n': ad ogni passo la dimensione si riduce di:

- un valore costante, non sempre uguale per tutti i sottoproblemi
- un fattore costante, in generale lo stesso per tutti i sottoproblemi
- una quantità variabile, sovente difficile da stimare.



Terminologia incontrata in letteratura:

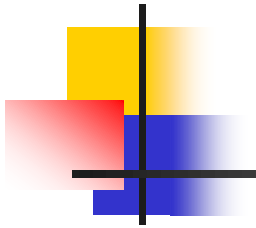
- Divide and conquer: a>1 e fattore o valore di riduzione in generale costante
- Decrease and conquer: a=1 e valore di riduzione in generale costante.

L'albero della ricorsione

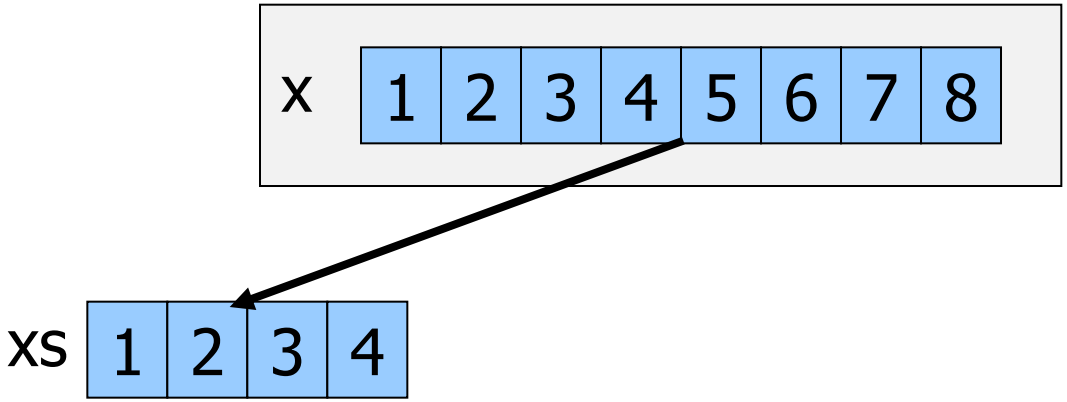
divide and conquer
a = 2 b = 2

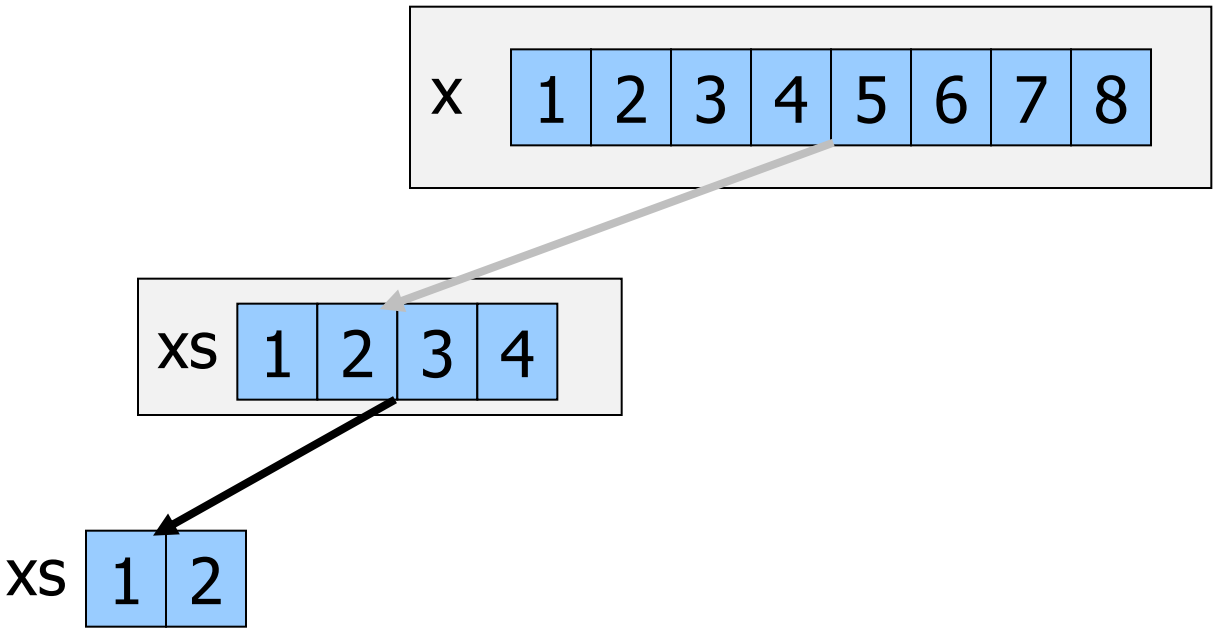
Esempio 1:

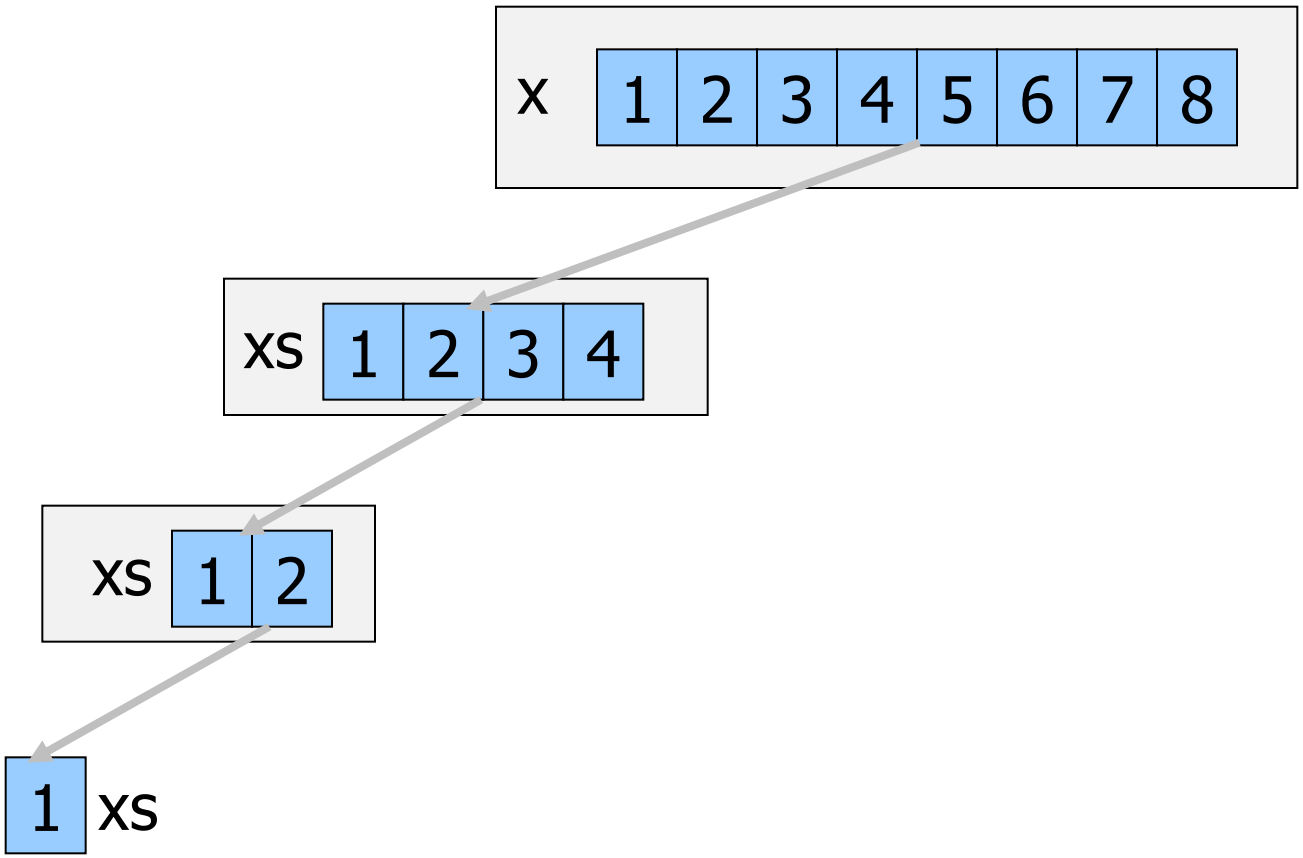
dato un vettore di $n=2^k$ interi, suddividerlo ricorsivamente in sottovettori di dimensione metà, fino alla condizione di terminazione (sottovettore di 1 sola cella).



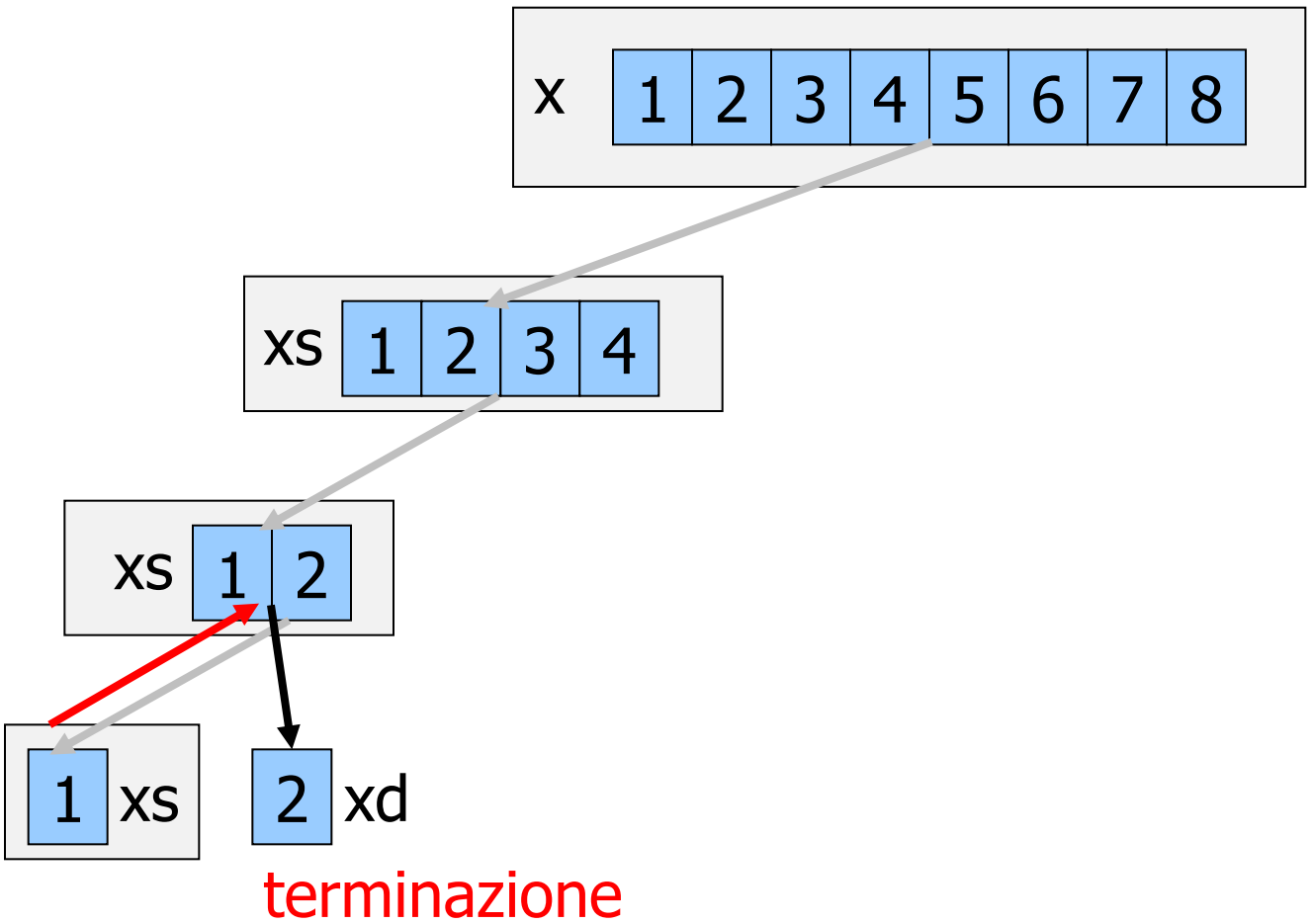

x
1
2
3
4
5
6
7
8

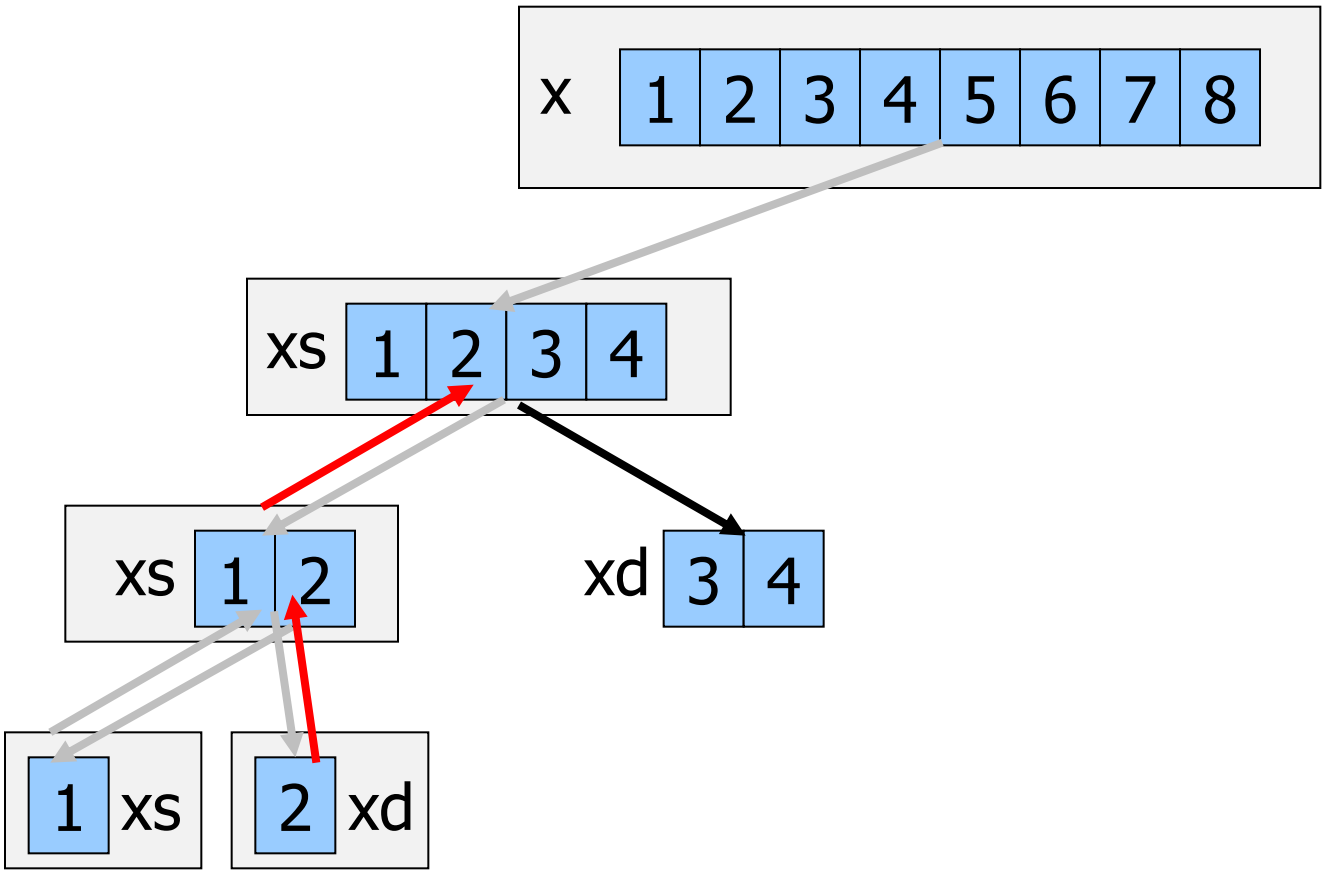

x
1
2
3
4
5
6
7
8
xs
1
2
3
4

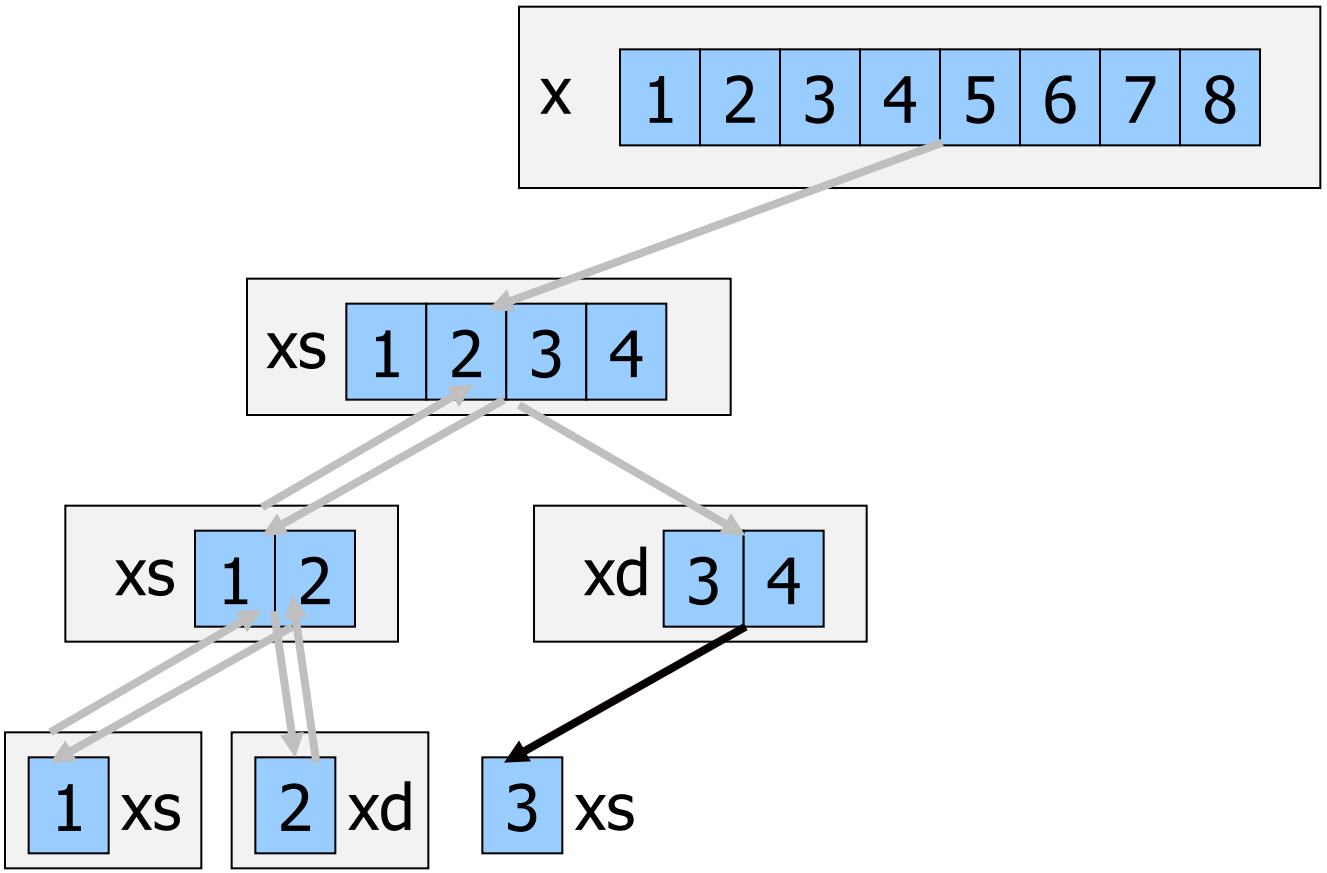

x
1
2
3
4
5
6
7
8
xs
1
2
3
4
xs
1
2



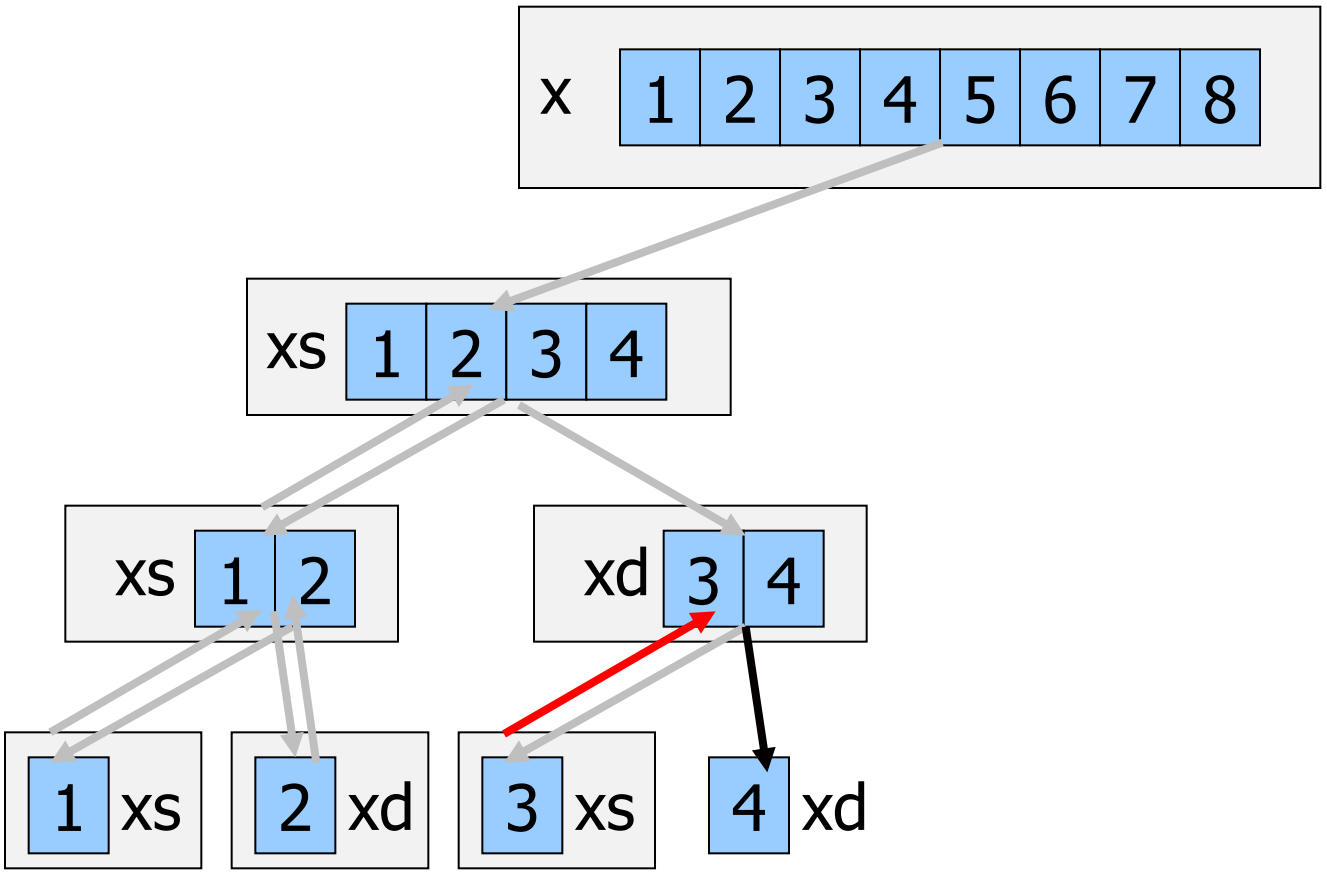
terminazione


x
1 2 3 4 5 6 7 8
xs
1 2 3 4
xs
1 2
1 xs
2 xd
terminazione

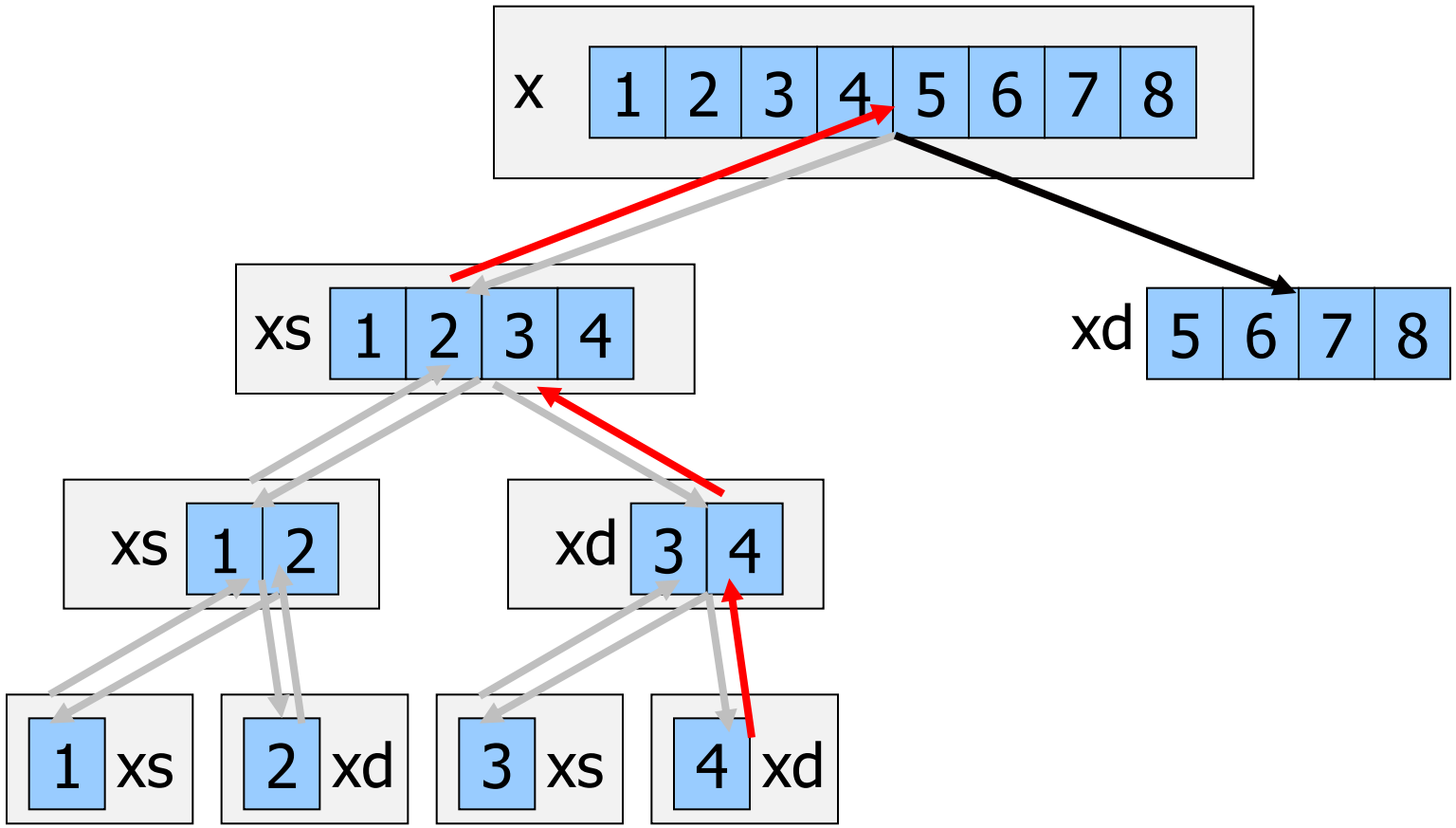

x
1
2
3
4
5
6
7
8
xs
1
2
3
4
xs
1
2
xd
3
4
1
xs
2
xd

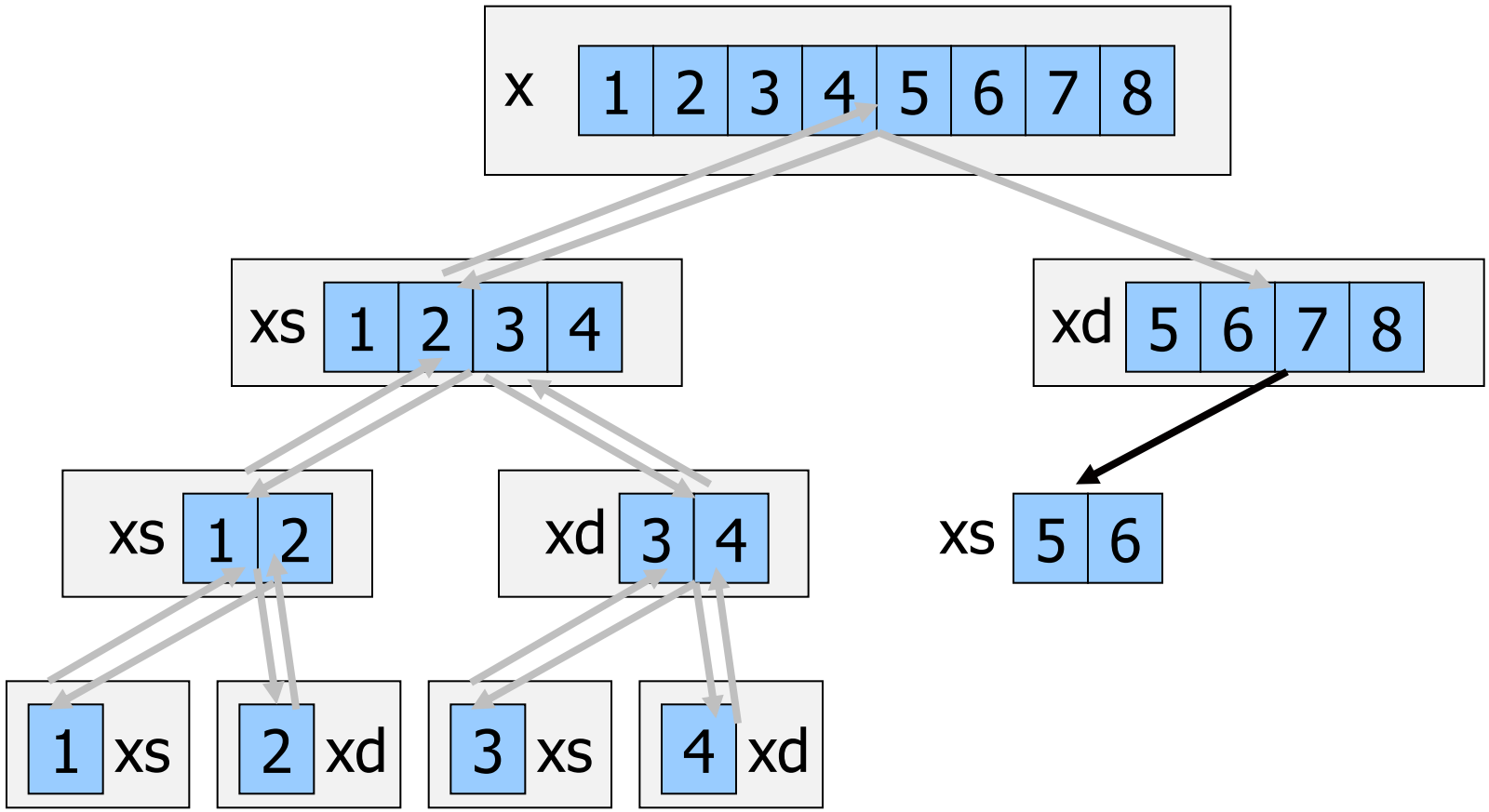


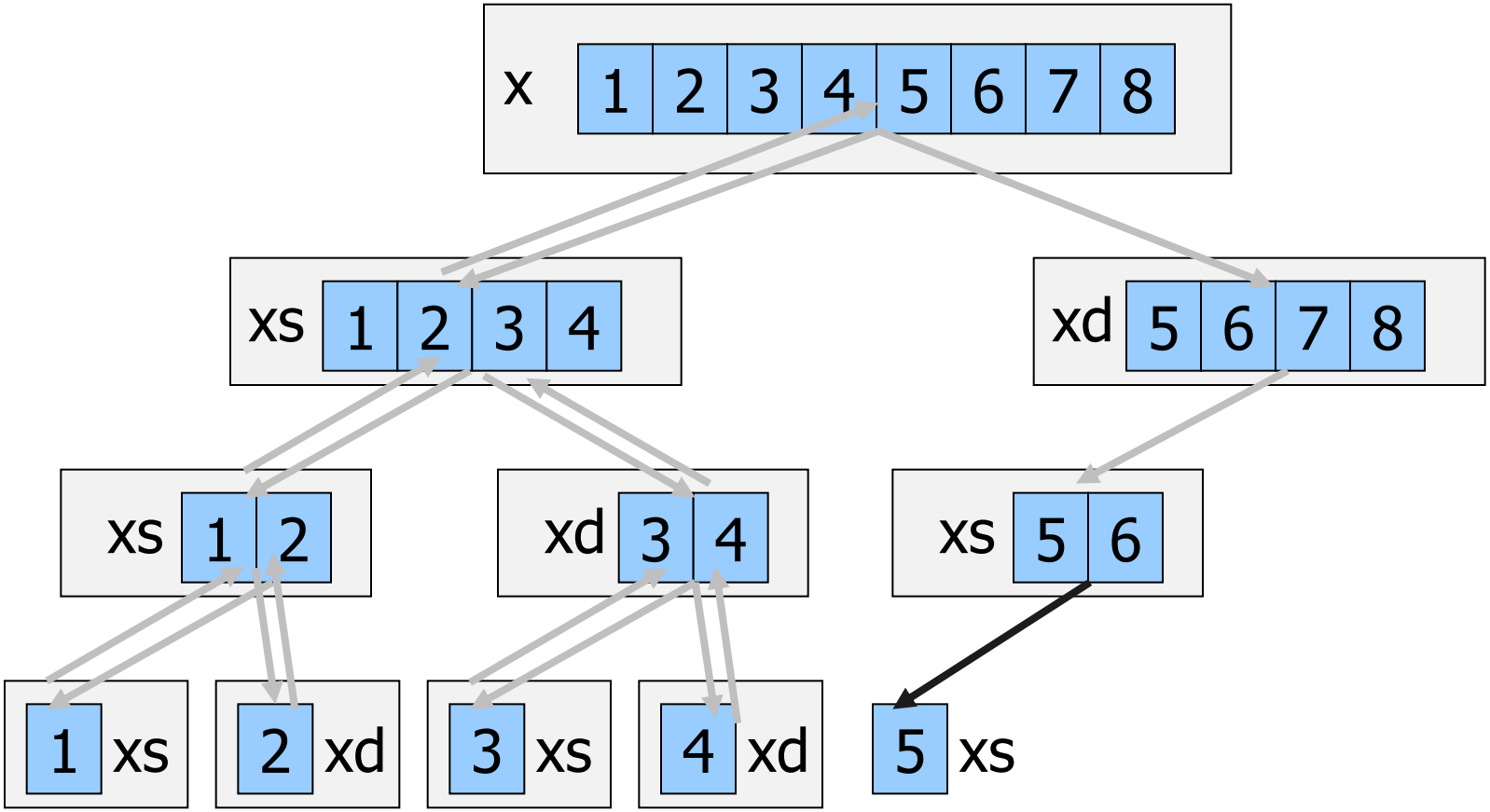
terminazione



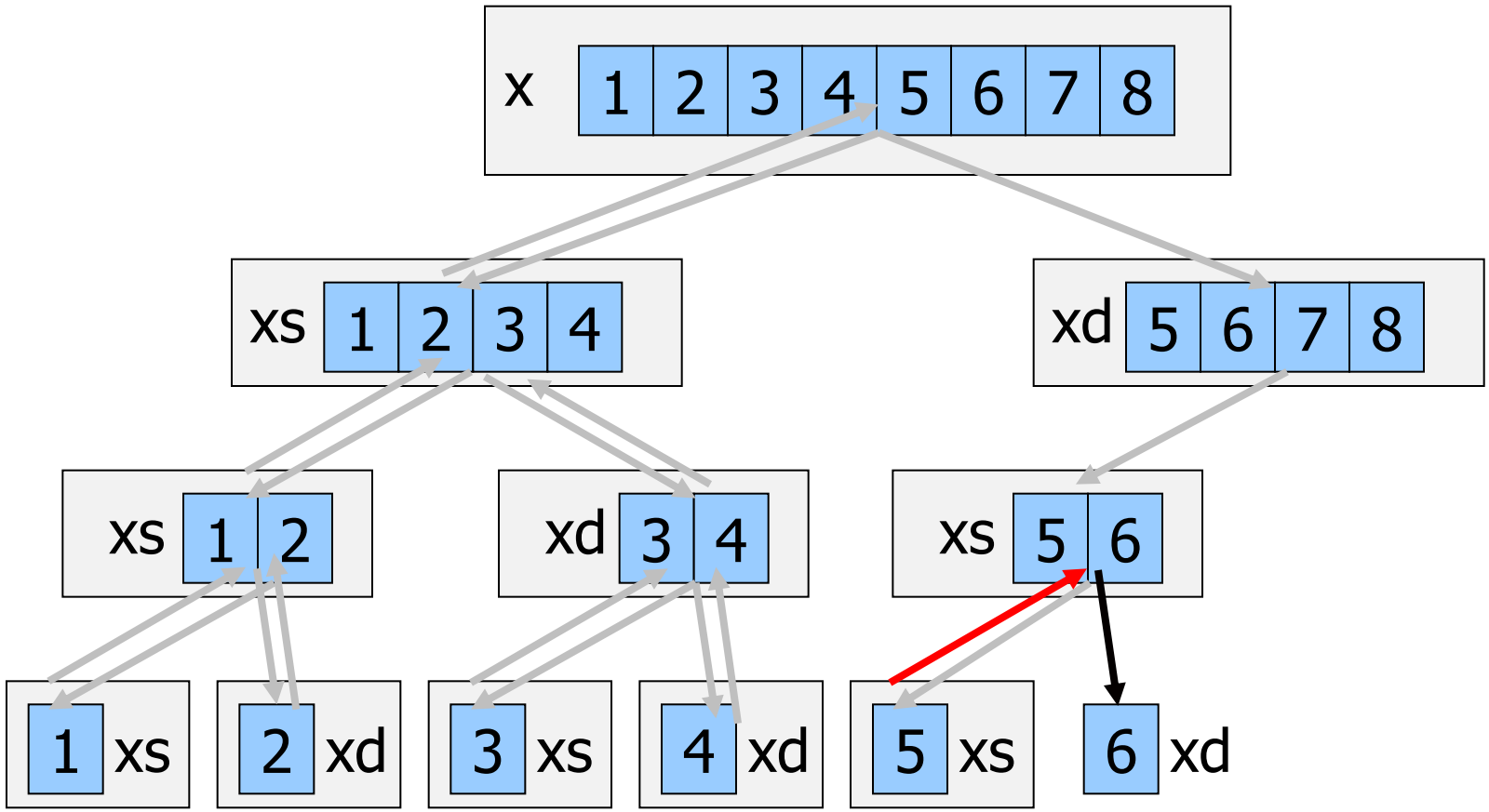
terminazione

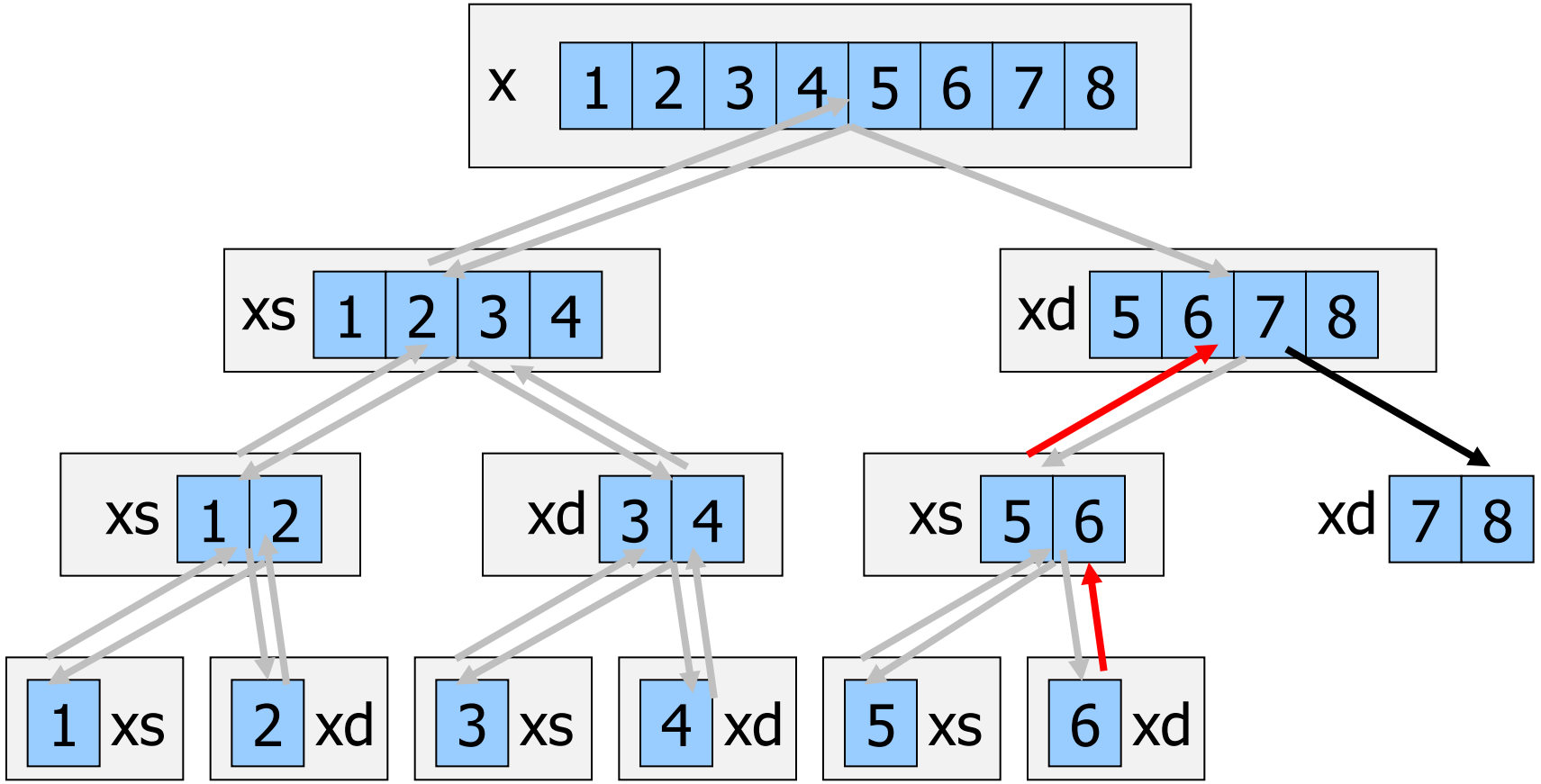

x
1 2 3 4 5 6 7 8
xs
1 2 3 4
xd
5 6 7 8
xs
1 2
xd
3 4
1 xs
2 xd
3 xs
4 xd

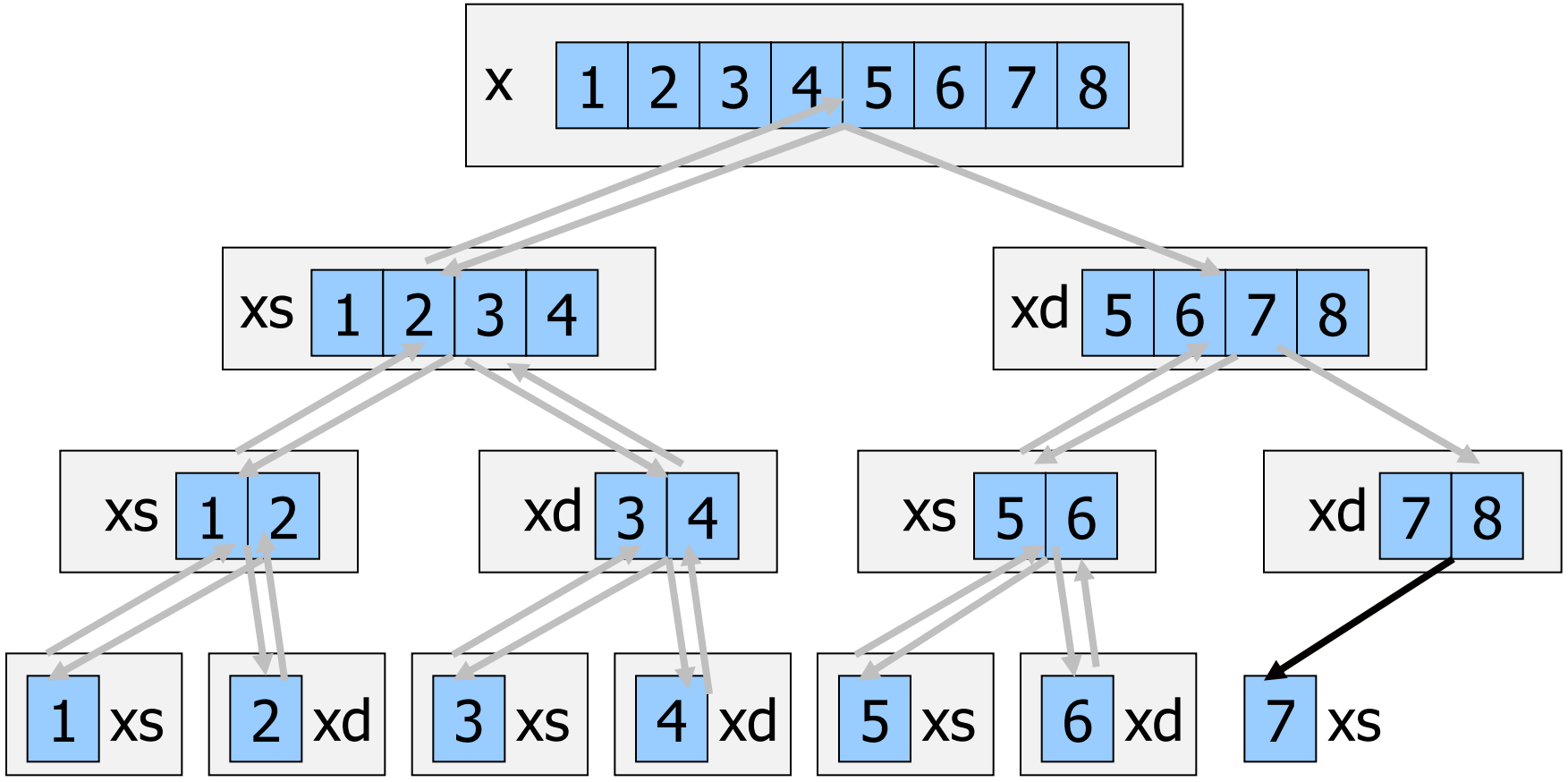

x
1 2 3 4 5 6 7 8
xs
1 2 3 4
xd
5 6 7 8
xs
1 2
xd
3 4
xs
5 6
1 xs
2 xd
3 xs
4 xd

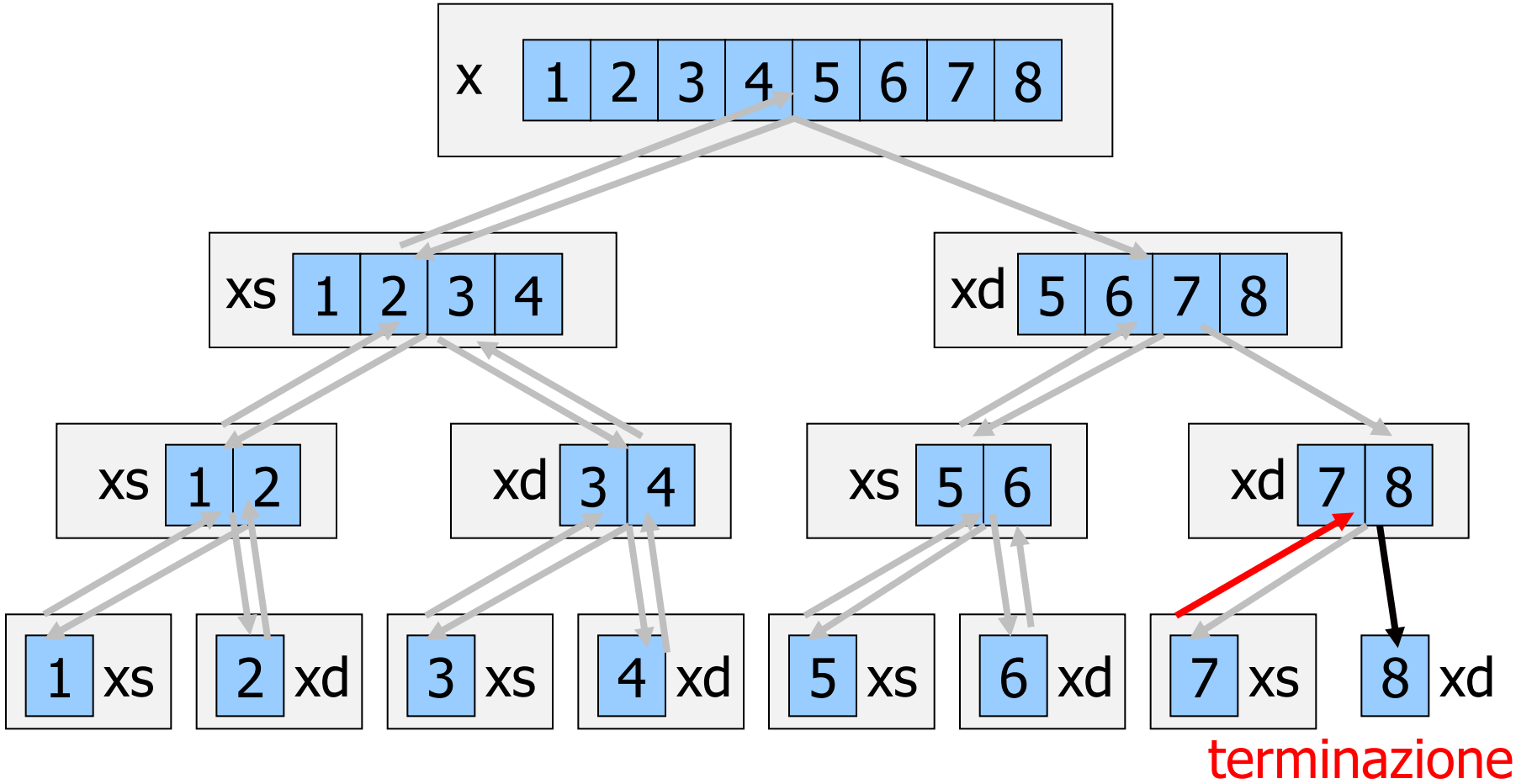


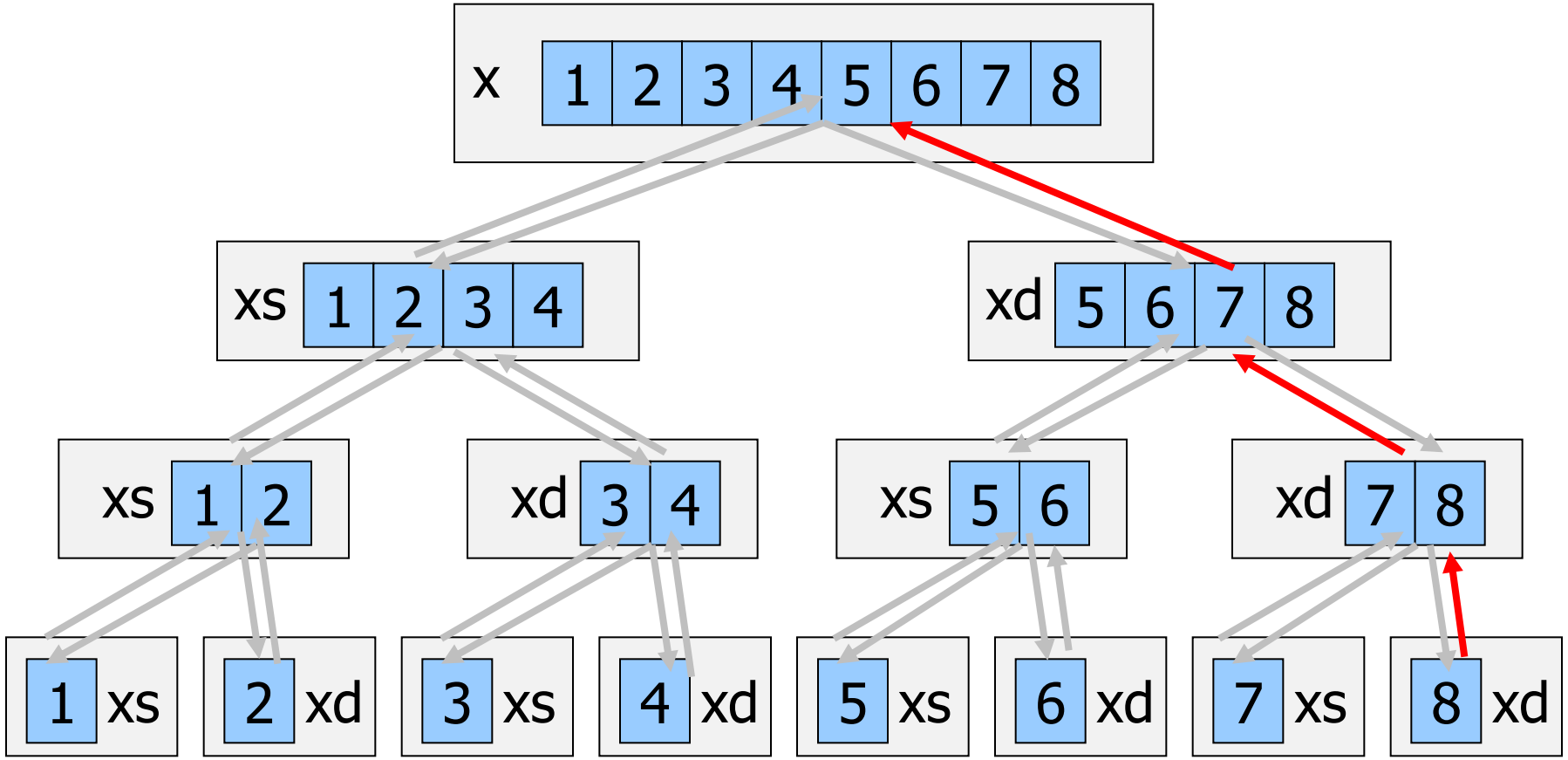
terminazione


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
terminazione


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
7 xs
terminazione

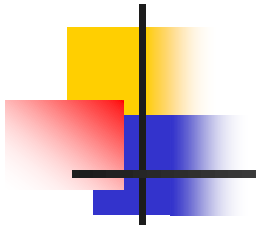

x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
7 xs
8 xd
terminazione


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
7 xs
8 xd

condizione di prosecuzione della ricorsione

00show_recursion_tree.c

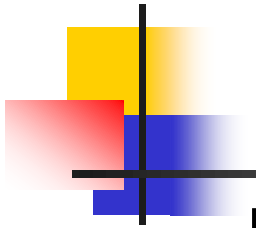
```
void show(int x[], int l, int r) {
  int i, c;
  if (l<r) {
    c = (r+l)/2;
    printf("xs = ");
    for (i=l; i <= c; i++)
      printf("%d", x[i]);
    printf("\n");
    show(x, l, c);
    printf("xd = ");
    for (i=c+1; i <= r; i++)
      printf("%d", x[i]);
    printf("\n");
    show(x, c+1, r);
  }
  return;
}
```



divide and conquer
a = 2 b = 2

Esempio 2:

dato un vettore di $n=2^k$ interi, determinarne il massimo.

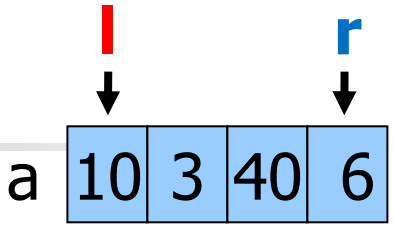


Massimo di un vettore di interi:

- se la dimensione è n=1, l'unico elemento è anche il massimo
- per n>1
  - divido il vettore in due sottovettori metà
  - applico ricorsivamente la ricerca del massimo a ciascun sottovettore
  - confronto i risultati e restituisco il più grande.

Nel main:

```
result = max(a, 0, 3);
```



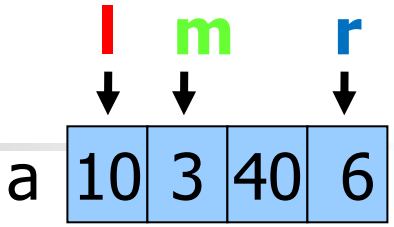
01max_array.c

$n = 2^2$

l = 0 r = 3

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

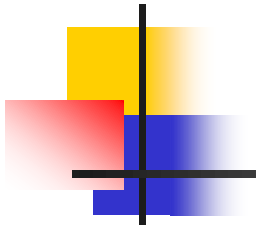
```
max(a, 0, 3);
```



l = 0 **r** = 3 **m** = 1

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```



a	10	3	40	6

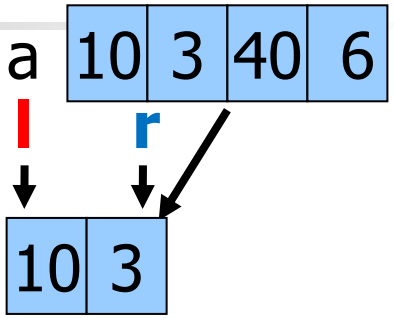
l = 0 **r** = 3 **m** = 1

chiamata ricorsiva

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

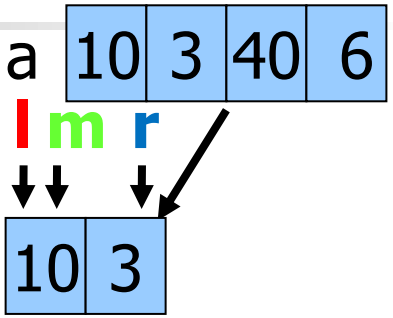
```
max(a, 0, 1);
```



l = 0 r = 1

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

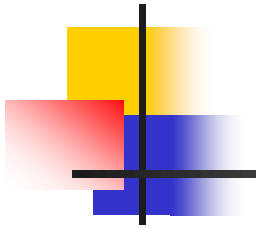
```
max(a, 0, 1);
```

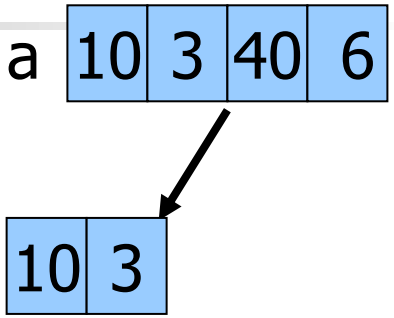


l = 0 r = 1 m = 0

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```





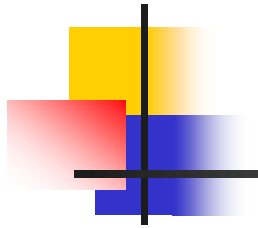
l = 0 **r** = 0 **m** = 0

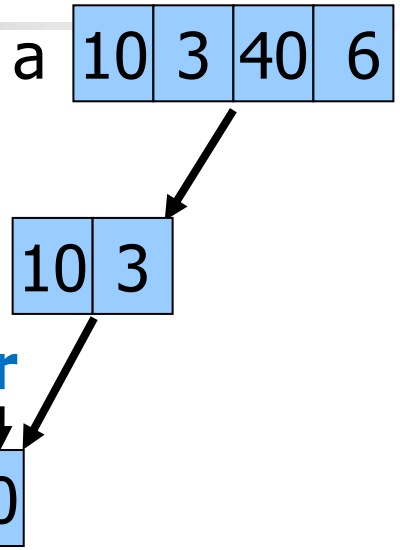
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 0, 0);
```

```
max(a, 0, 0);
```

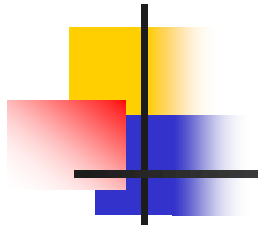


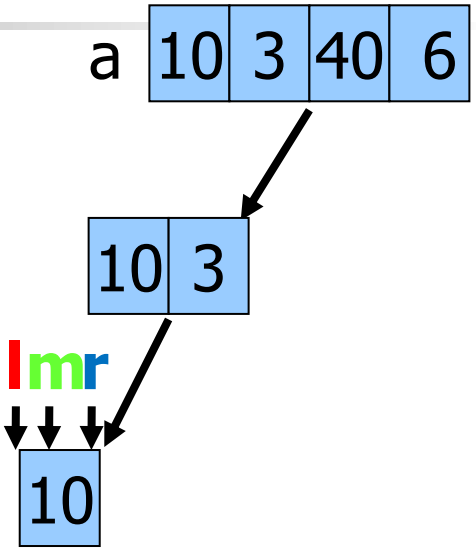


l = 0 r = 0

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 0);
```

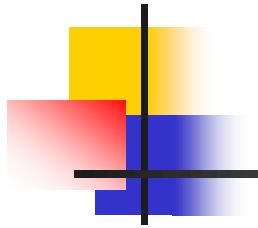


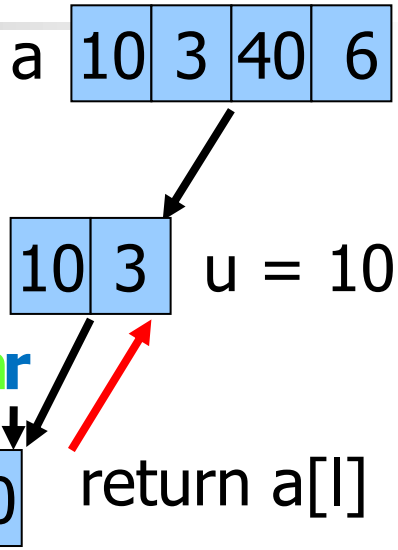


l = 0 r = 0 m = 0

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

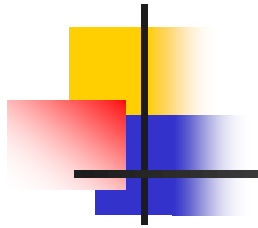
```
max(a, 0, 0);
```

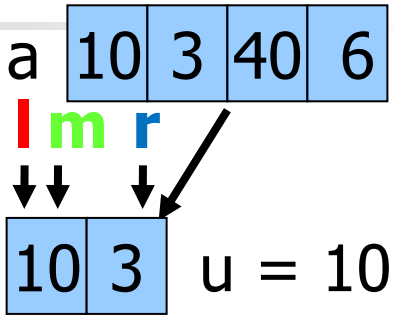





```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```

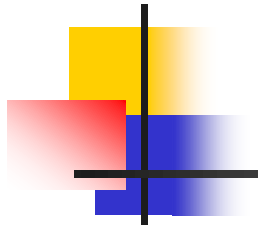


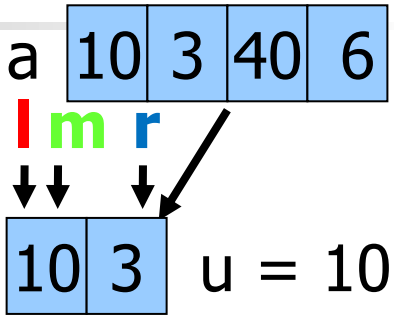


l = 0 r = 1 m = 0

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```





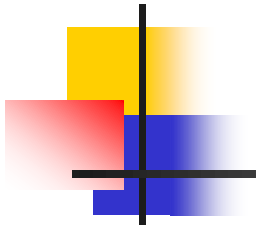
l = 0 r = 1 m = 0

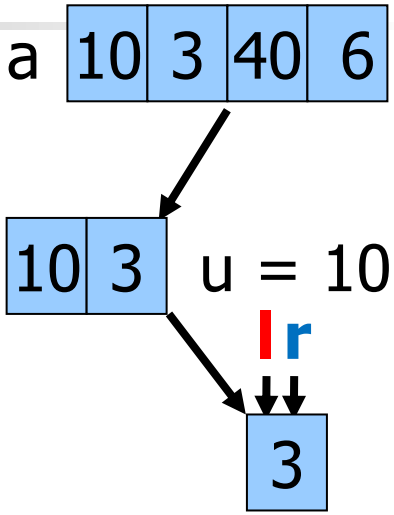
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 1, 1);
```

```
max(a, 1, 1);
```

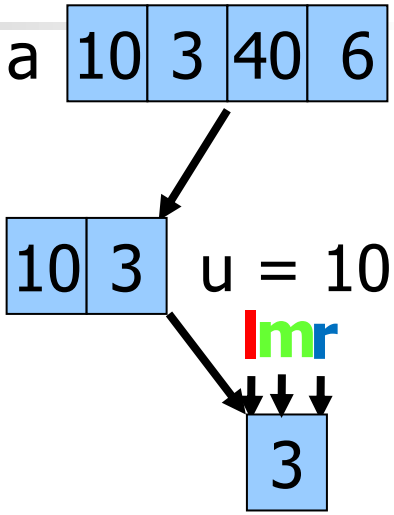




l = 1 r = 1

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

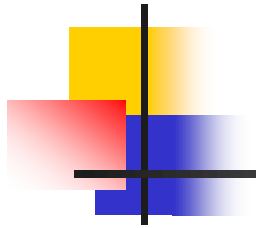
```
max(a, 1, 1);
```

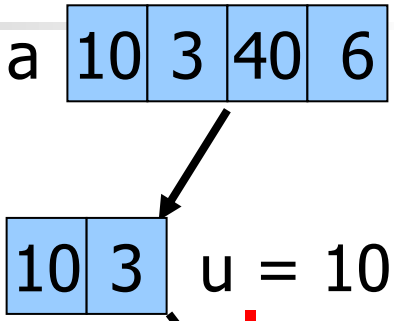


l = 1 r = 1 m = 1

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 1, 1);
```

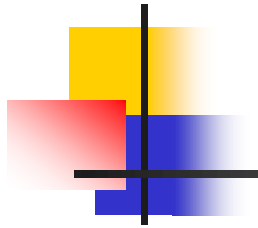


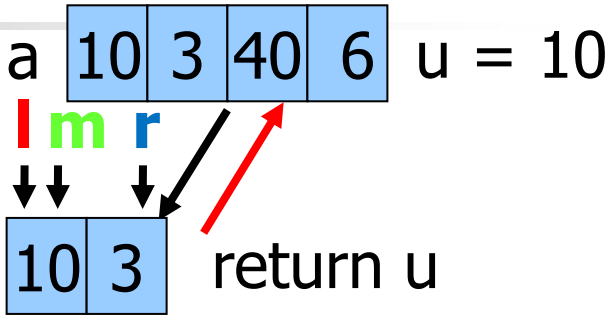


l = 1 r = 1 m = 1

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```

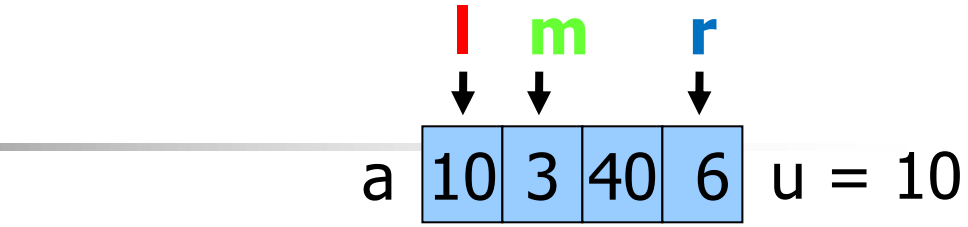




l = 0 r = 1 m = 0

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

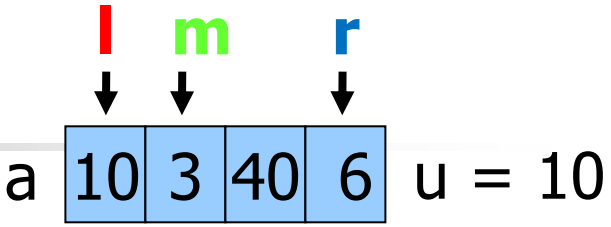
```
max(a, 0, 3);
```



l = 0 **r** = 3 **m** = 1

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 3);
```



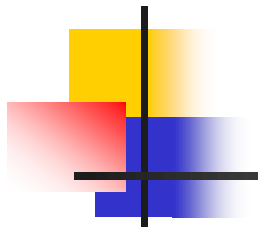
l = 0 **r** = 3 **m** = 1

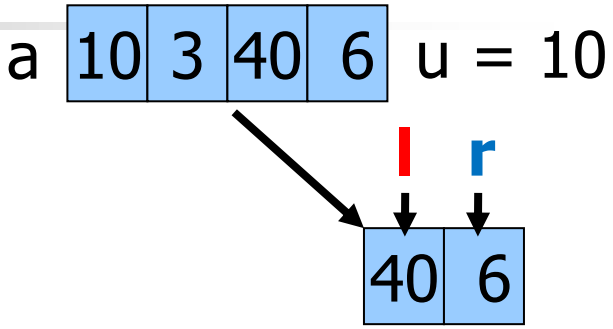
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 2, 3);
```

```
max(a, 2, 3);
```

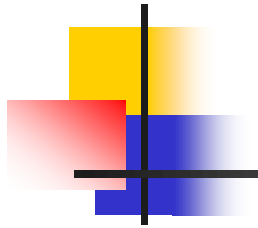


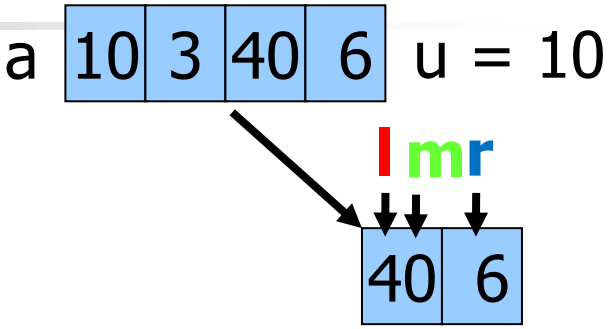


l = 2 r = 3

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```

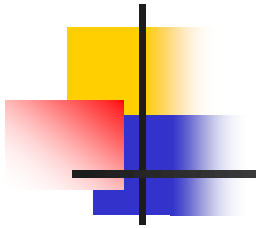


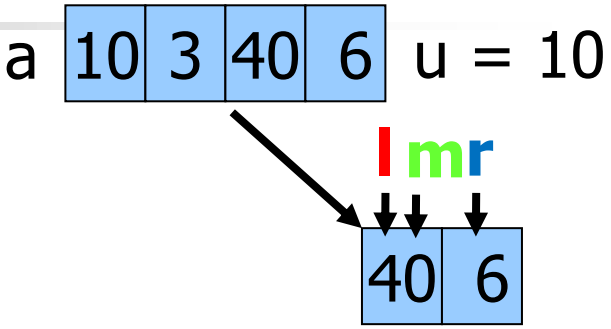


l = 2 **r** = 3 **m** = 2

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```





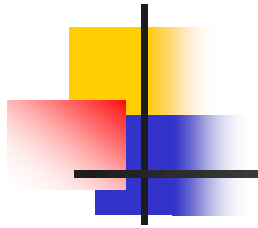
l = 2 **r** = 3 **m** = 2

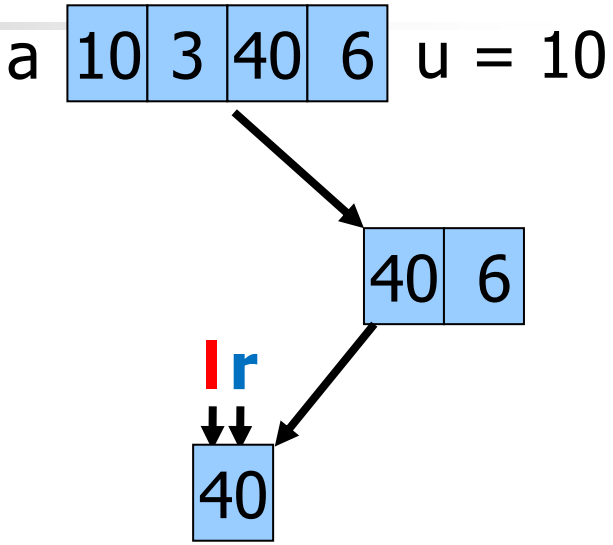
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 2, 2);
```

```
max(a, 2, 2);
```

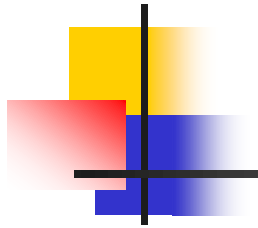


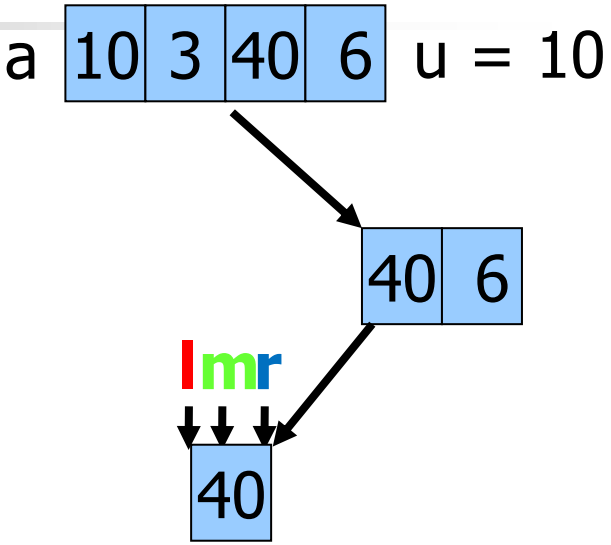


l = 2 r = 2

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 2);
```

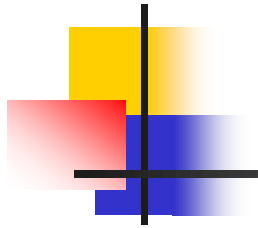


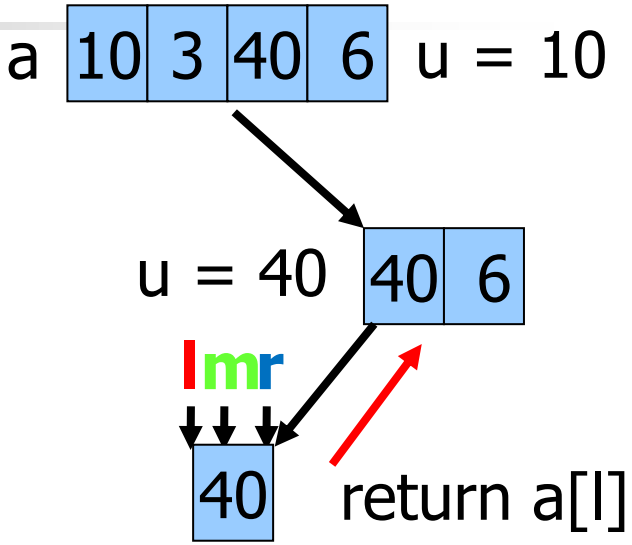


l = 2 r = 2 m = 2

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 2);
```

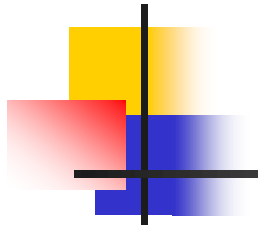


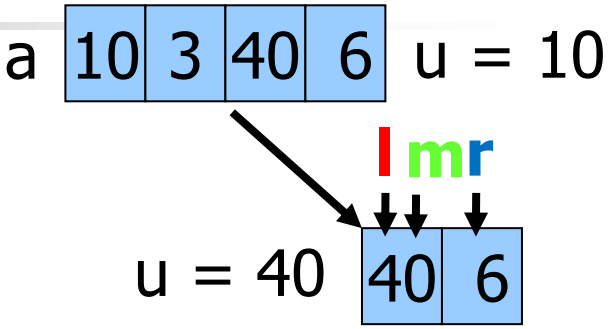


l = 2 r = 2 m = 2

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```

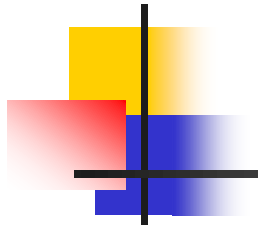


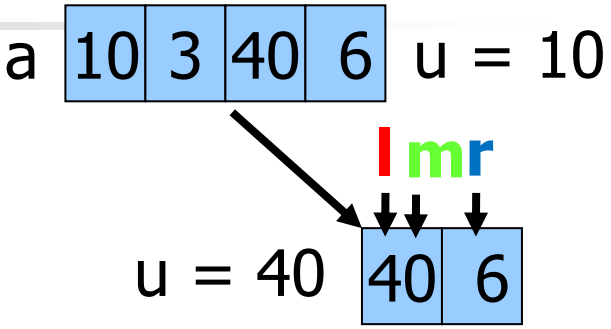


l = 2 **r** = 3 **m** = 2

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```





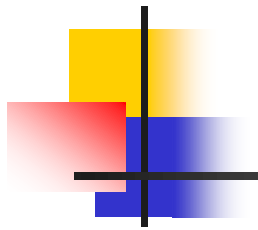
l = 2 **r** = 3 **m** = 2

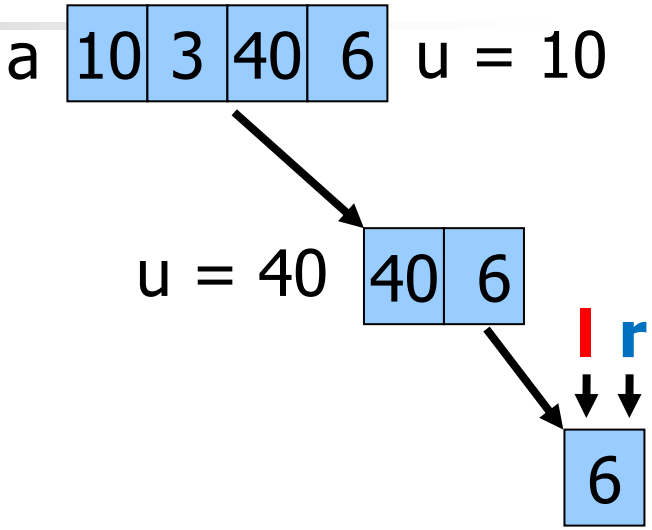
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 3, 3);
```

```
max(a, 3, 3);
```

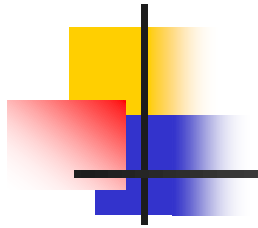


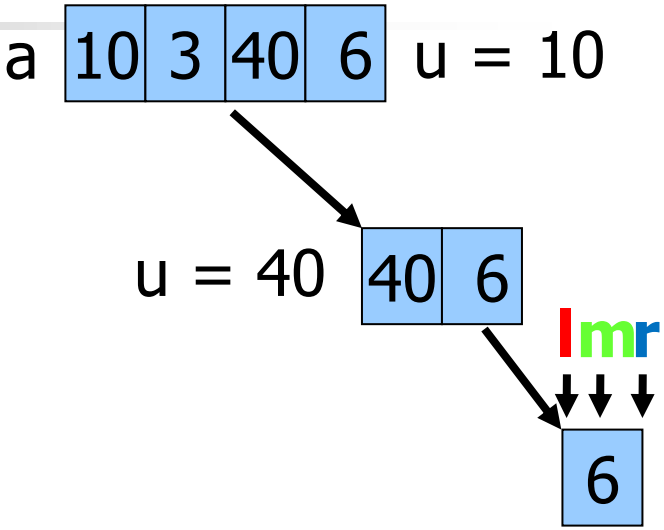


l = 3 r = 3

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 3, 3);
```

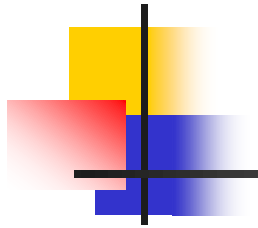


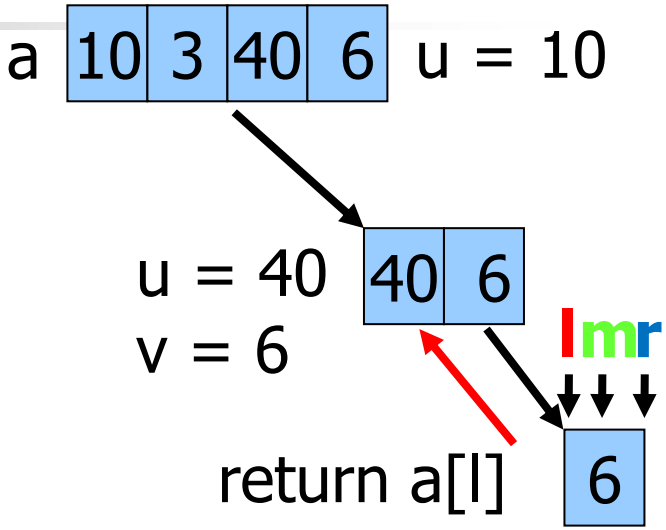


l = 3 r = 3 m = 3

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

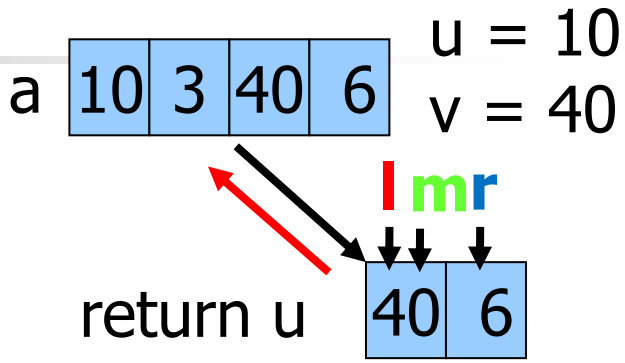
```
max(a, 3, 3);
```






```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```



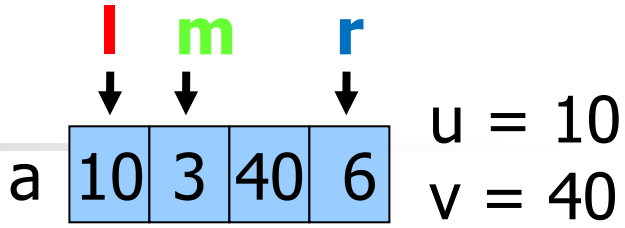
l = 2 r = 3 m = 2

u = 40

v = 6

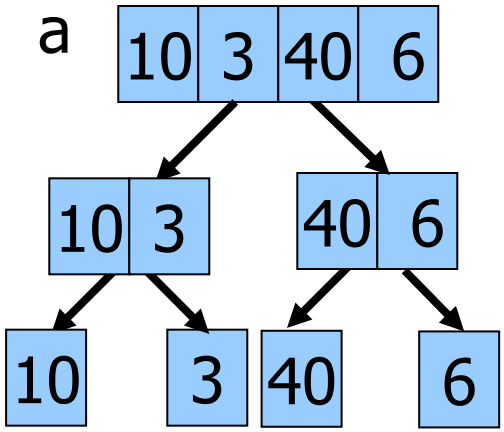
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

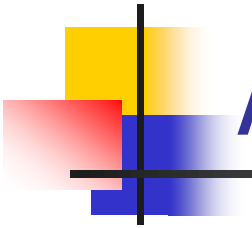
`result = max (A, 0, 3);` ⇨ `result = 40`



l = 0 r = 3 m = 1 return v

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```


a
10
3
40
6
10
3
40
6
10
3
40
6



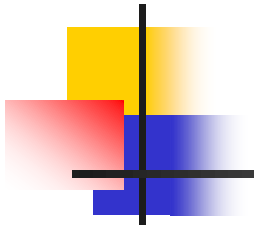
Analisi di Complessità

Equazione alle Ricorrenze:

T(n) viene espressa in termini di:

- D(n): costo della divisione
- tempo di esecuzione per input più piccoli (ricorsione)
- C(n): costo della ricombinazione

Si suppone che il costo della soluzione elementare sia unitario $\Theta(1)$.

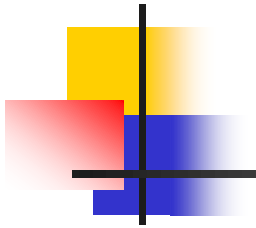


Se:

- a è il numero di sottoproblemi che risulta dalla fase di Divide
- b è il fattore di riduzione, quindi n/b è la dimensione di ciascun sottoproblema

l'equazione alle ricorrenze ha forma:

$$T(n) = D(n) + a\,T(n/b) + C(n) \qquad n > c$$
$$T(n) = \Theta(1) \qquad n \leq c$$



T(n)

Risolvi(Problema):

- Se il problema è elementare:
  - Soluzione = Risolvi_banale(Problema)

Θ(1)

- Altrimenti:

D(n)

  - Sottoproblema$_{1,2,3,\ldots,a}$ = Dividi(Problema) ;
  - Per ciascun Sottoproblema$_i$:

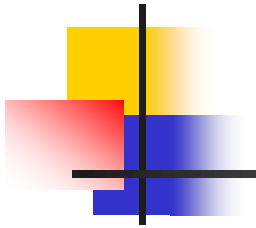
a sottoproblemi

    - Sottosoluzione$_i$ = Risolvi(Sottoproblema$_i$) ;

T(n/b)

  - Return Soluzione Combina(Sottosoluzione$_{1,2,3,\ldots,a}$) ;

C(n)



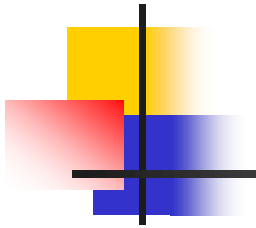
Se:

- a è il numero di sottoproblemi che risulta dalla fase di Divide
- la riduzione è di un valore k_i, che può variare di passo in passo

l'equazione alle ricorrenze ha forma:

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-ki) + C(n) \qquad n > c$$

$$T(n) = \Theta(1) \qquad n \leq c$$



T(n)

Risolvi(Problema):

- Se il problema è elementare:
  - Soluzione = Risolvi_banale(Problema)

Θ(1)

- Altrimenti:

D(n)

  - Sottoproblema$_{1,2,3,\ldots,a}$ = Dividi(Problema) ;
  - Per ciascun Sottoproblema$_i$:

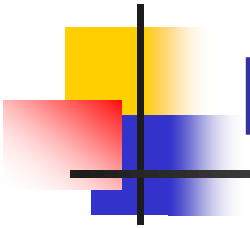
a sottoproblemi

    - Sottosoluzione$_i$ = Risolvi(Sottoproblema$_i$) ;

T(n-k$_i$)

  - Return Soluzione Combina(Sottosoluzione$_{1,2,3,\ldots,a}$) ;

C(n)



Problemi ricorsivi semplici

Matematici:

- fattoriale
- numeri di Fibonacci
- massimo comun divisore
- prodotto di 2 interi positivi
- determinante di una matrice.

Il fattoriale

decrease and conquer
a = 1 k_i = 1

Fattoriale (definizione ricorsiva)

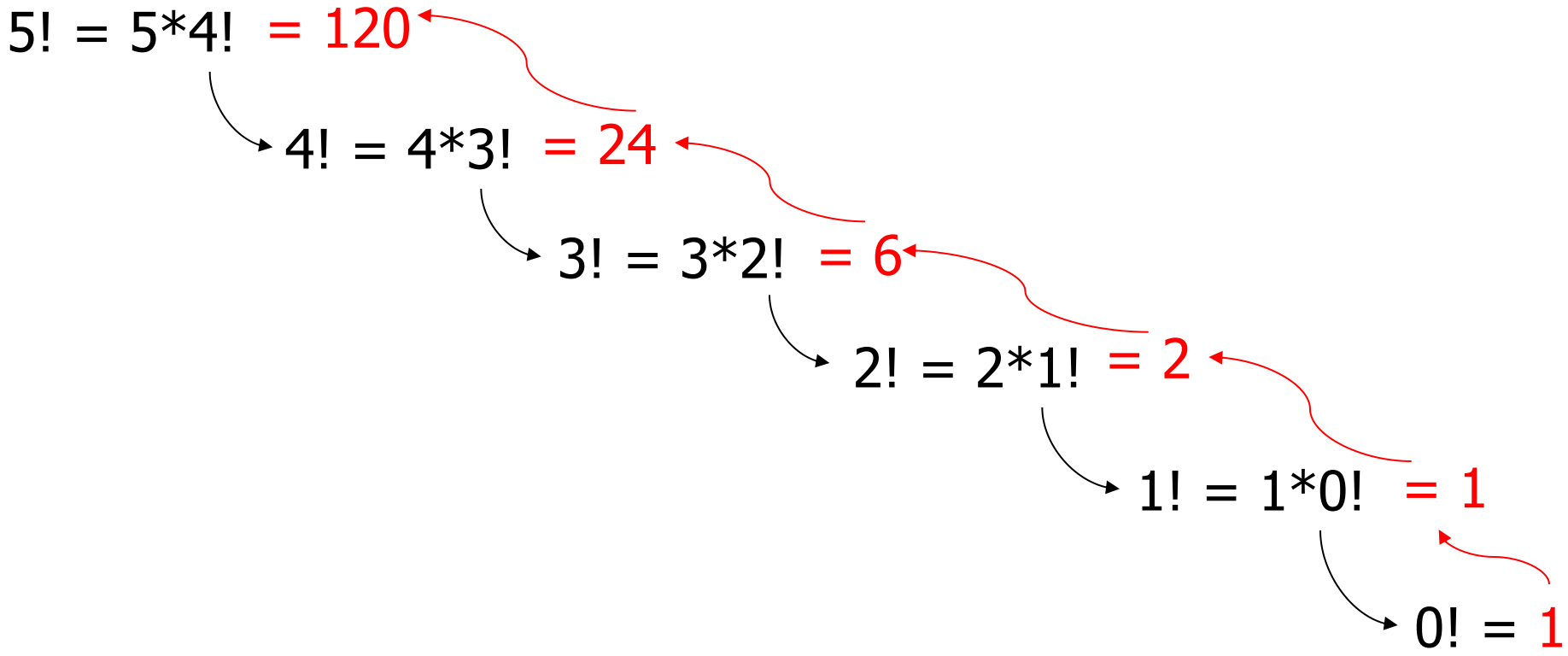
n! ≡ n * (n-1)! n≥1

0! ≡ 1

Fattoriale (definizione iterativa)

n! ≡ $\prod_{i=0}^{n-1}(n-i)$ = n * (n-1) * …. 2 * 1

Esempio



02recursive_factorial.c

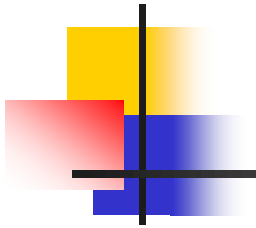
```c
#include <stdio.h>
int fact(int n);
main() {
  int n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("factorial of %d is: %d \n", n, fact(n));
}
int fact(int n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$
- equazione alla ricorrenze:

$T(n) = 1 + T(n-1)$ $\quad n > 1$

$T(1) = 1$



- Risoluzione per sviluppo (unfolding):

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

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

T(n-2) = 1 + T(n-3)

Sostituendo in T(n)

T(n) = 1+1+1+T(n-3) = $\sum_{i=0}^{n-1} 1 = 1 + (n-1) = n$

Terminazione:
n-i = 1
i= n -1

Quindi:

T(n) = O(n)

I numeri di Fibonacci

divide and conquer
$a = 2$ $k_i = 1$ $k_{i-1} = 2$

Numeri di Fibonacci:

$$FIB_n = FIB_{n-2} + FIB_{n-1} \qquad n>1$$

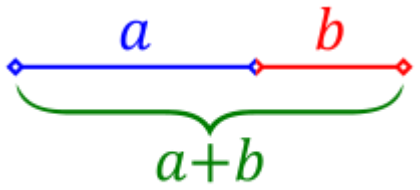
$$FIB_0 = 0$$

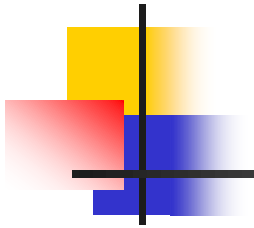
$$FIB_1 = 1$$

La sezione aurea $\varphi = a/b$

La sezione aurea o rapporto aureo o numero aureo o costante di Fidia o proporzione divina è il rapporto fra due segmenti diversi a e b tale per cui il maggiore (a) è medio proporzionale tra il minore (b) e la somma dei due. Lo stesso rapporto esiste anche tra il minore e la differenza.

$$(a+b) : a = a : b = b : (a-b)$$





Ponendo $\varphi = a/b$, $(a+b)/a = a/b$ diventa:

$$(\varphi + 1)/\varphi = \varphi \text{ quindi } \varphi^2 - \varphi - 1 = 0$$

Risolvendo e considerando solo la soluzione positiva

$$\varphi = (1 + \sqrt{5})/2 = 1.61803$$

Sezione aurea e numeri di Fibonacci

Considerando anche la soluzione negativa:

$$\varphi' = (1 - \sqrt{5})/2 = -0.61803$$

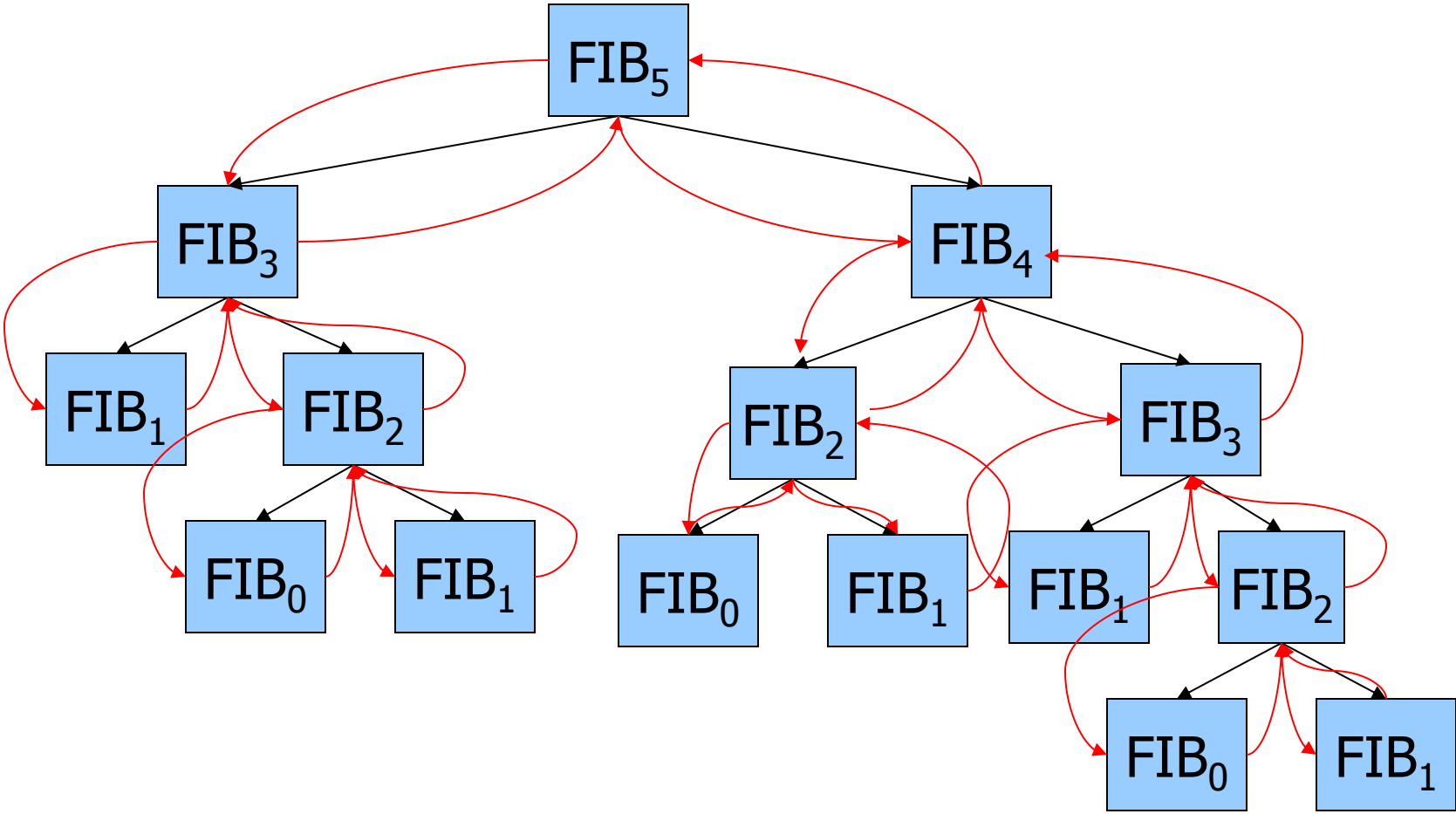
l'n-esimo numero di Fibonacci FIB_n si può esprimere come:

$$FIB_n = (\varphi^n - \varphi'^n)/ \sqrt{5}$$

03recursive_fibonacci.c

```c
#include <stdio.h>
int fib(int n);
main() {
  int n;
   printf("Input n:  ");
  scanf("%d", &n);
  printf("fibonacci of %d is: %d \n", n, fib(n));
}

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

Esempio


FIB5
FIB3
FIB4
FIB1
FIB2
FIB2
FIB3
FIB0
FIB1
FIB0
FIB1
FIB1
FIB2
FIB0
FIB1

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2$, $k_i = 1$, $k_{i-1} = 2$
- equazione alla ricorrenze:

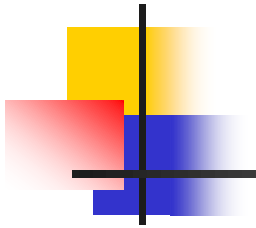
$T(n) = 1 + T(n-1) + T(n-2)$ $\quad n > 1$

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

- approssimazione conservativa: essendo $T(n-2) \leq T(n-1)$, lo sostituisco con $T(n-1)$

$T(n) = 1 + 2T(n-1)$ $\quad n > 1$

$T(n) = 1$



- Risoluzione per sviluppo (unfolding):

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

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

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

Sostituendo in T(n)

$T(n) = 1 + 2 + 4 + 2^3T(n-3) = \sum_{i=0}^{n-1} 2^i = 2^{n-1}$

Terminazione:
n-i = 1
i= n -1

$\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x-1)$

Quindi:

$T(n) = O(2^n)$

Stima migliore: $T(n) = O(\varphi^n)$

decrease and conquer
a = 1 k_i variabile

Il massimo comun divisore

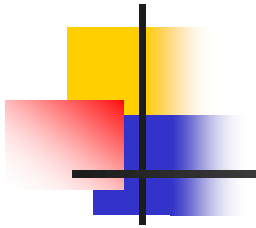
Il massimo comun divisore *gcd* di due interi *x* e *y* non entrambi nulli è il più grande dei divisori comuni di *x* e *y.*

Esempio: gcd(600,54) = 6

Algoritmo inefficiente basato sulla scomposizione in fattori primi:

x = $p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_r^{e_r}$ y = $p_1^{f_1} \cdot p_2^{f_2} \cdot \ldots \cdot p_r^{f_r}$

gcd(x,y) = $p_1^{\min(e_1,f_1)} \cdot p_2^{min(e_2,f_2)} \cdot \ldots \cdot p_r^{\min(e_r,fr)}$



Algoritmo di Euclide (Djikstra):

- versione 1: sottrazione
  - se x > y

    gcd(x, y) = gcd(x-y, y)

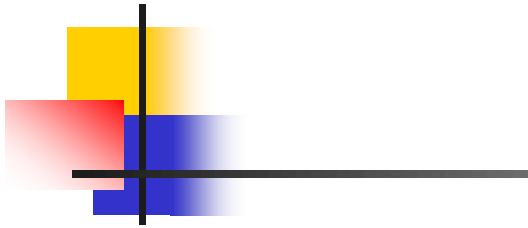
    altrimenti

    gcd(x, y) = gcd(x, y-x)
  - terminazione:

    se x=y ritorna x

04recursive_gcd.c

```c
#include <stdio.h>
int gcd(int x, int y);
main() {
  int x, y;
  printf("Input x and y:  ");
  scanf("%d%d", &x, &y);
  printf("gcd of %d and %d: %d \n", x, y, gcd(x, y));
}
int gcd(int x, int y) {
  if(x == y)
    return(x);
  if (x > y)
    return gcd(x-y, y);
  else
    return gcd(x, y-x);
}
```



Algoritmo di Euclide:

- versione 2: resto della divisione intera
- se x > y

  gcd(x, y) = gcd(y, x%y)

terminazione:

  se y = 0 ritorna x

04recursive_gcd.c

```c
#include <stdio.h>
int gcd(int x, int y);
main() {
  int x, y;
   printf("Input x and y:  ");
  scanf("%d%d", &x, &y);
  printf("gcd of %d and %d: %d \n", x, y, gcd(x, y));
}

int gcd(int x, int y) {
   if(y == 0)
     return(x);
   return gcd(y, x % y);
}
```

Esempi

```
gcd(600,54)
 gcd(54, 6)
  gcd(6, 0)      return 6

gcd(314159,271828)
 gcd(271828,42331)
  gcd(42331,17842)
   gcd(17842,6647)
    gcd(6647,4548)
     gcd(4548,2099)
      gcd(2099,350)
       gcd(350,349)
        gcd(349,1)
         gcd(1,0) return 1
```

314159 e 271828 sono primi tra di loro

Analisi di complessità

- $D(x,y) = \Theta(1)$, $C(x,y) = \Theta(1)$
- a = 1, riduzione variabile
- Caso peggiore: x e y sono 2 numeri di Fibonacci consecutivi:

  x = FIB(n+1) y = FIB(n)

Equazione alle ricorrenze

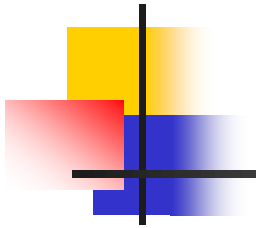
$T(x,y) = T(FIB(n+1), FIB(n))$

$= 1 + T(FIB(n),FIB(n+1)\%FIB(n))$

$T(x,0) = 1$

ma $FIB(n+1)\%FIB(n) = FIB(n-1)$

Terminazione: n passi



T(x,y) = T(FIB(n+1), FIB(n))

= 1 + T(FIB(n),FIB(n+1)%FIB(n))

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

T(x, y) = O(n), ma, visto che

y = FIB(n) = $(\varphi^n - \varphi'^n)/\sqrt{5} = \Theta(\varphi^n)$, allora

n = $\log_\varphi(y)$

Quindi:

T(n) = O(log(y))

Il massimo di un vettore

divide and conquer
a = 2 b = 2

Analisi di complessità:

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- a = 2, b = 2

Equazione alle ricorrenze

$T(n) = 2T(n/2) + 1 \qquad n > 1$

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

Terminazione:
$n/2^i = 1$
$i = \log_2 n$

- Risoluzione per sviluppo (unfolding):

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

$T(n/2) = 1 + 2T(n/4)$

$\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x-1)$

Sostituendo in T(n)

$T(n) = 1 + 2 + 4 + 2^3 T(n/8)$

$= \sum_{i=0}^{log_2 n} 2^i = (2^{\log_2 n + 1} - 1)/(2-1)$

$= 2 * 2^{\log_2 n} - 1 = 2n - 1$

Quindi:

$T(n) = O(n)$

divide and conquer
a = 4 b = 2

Il prodotto di 2 interi

Moltiplicazione di 2 interi positivi x e y di n cifre (con $n = 2^{k)}$:

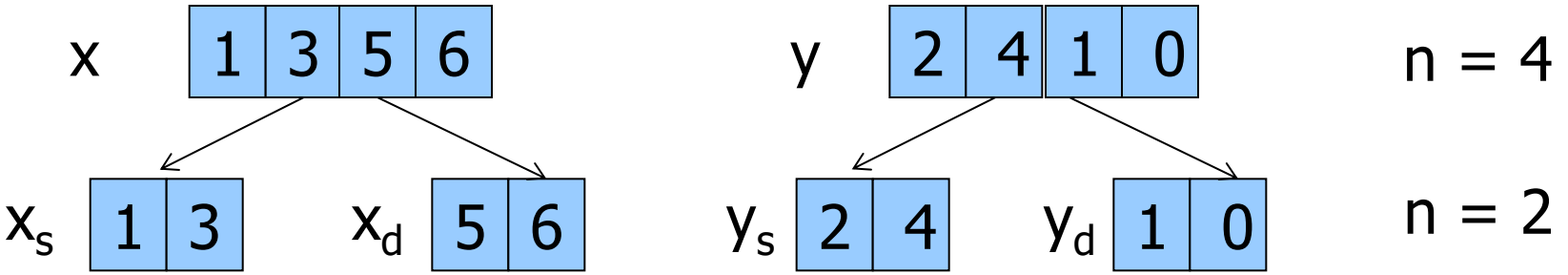
- se la dimensione è n=1, calcola x * y (tabelline pitagoriche)
- per n>1
  - dividi x in 2: $x = 10^{n/2} * x_s + x_d$
  - dividi y in 2: $y = 10^{n/2} * y_s + y_d$
  - calcola ricorsivamente x_s*y_s, x_s*y_d, x_d*y_s, x_d*y_d,
  - calcola

$$x * y = 10^n * x_s * y_s + 10^{n/2} * (x_s*y_d + x_d*y_s) + x_d*y_d$$

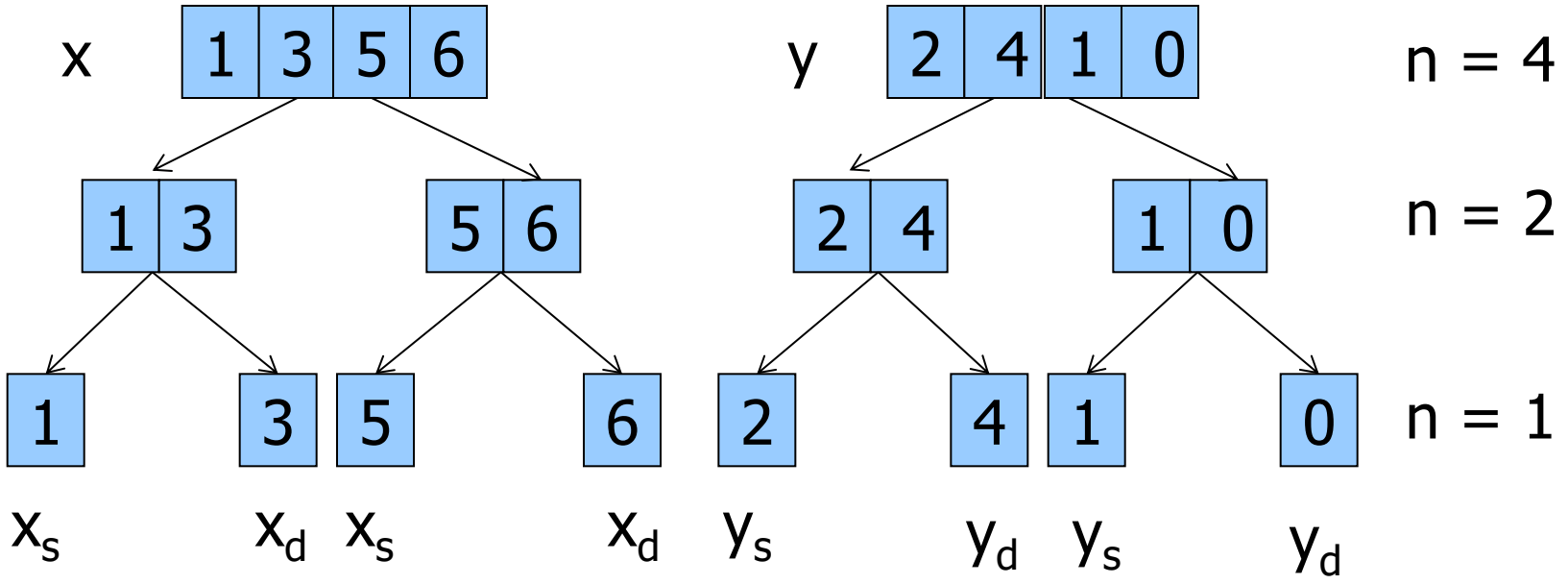
Esempio

1356 * 2410 = 3.267.960

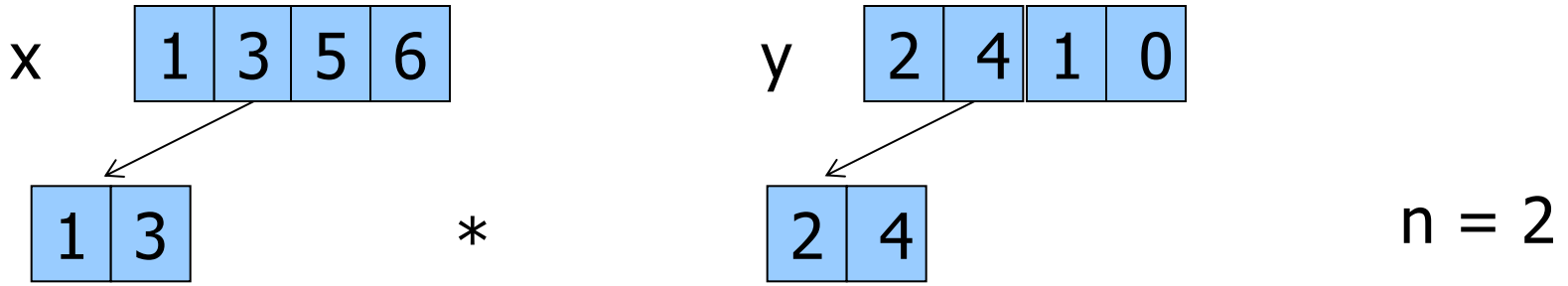
1356 * 2410 = 3.267.960



1356 * 2410 = 3.267.960



$$1356 * 2410 = 3.267.960$$

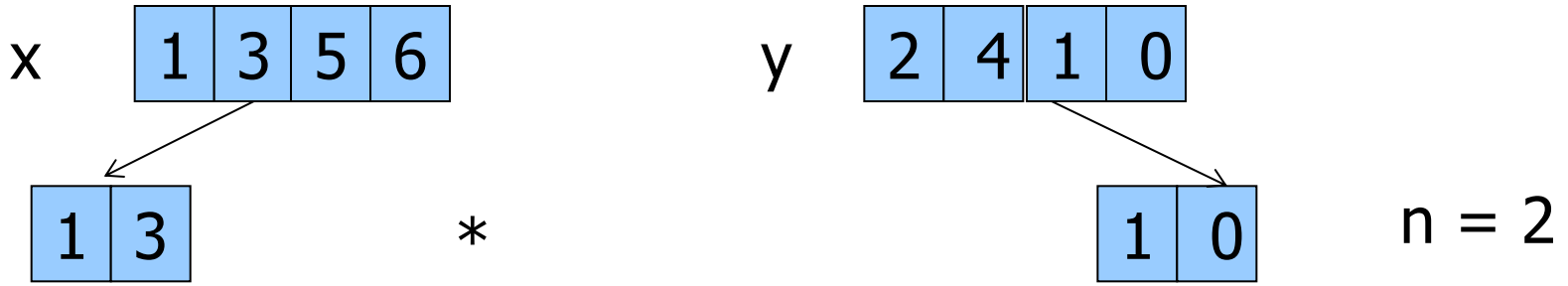


$$10^n \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d$$

$$10^2 * 1 * 2 + 10^1 * (1 * 4 + 3 * 2) + 3 * 4$$

$$13 * 24 = 10^2 * 2 + 10^1 * 10 + 12 = 312$$

$$1356 * 2410 = 3.267.960$$

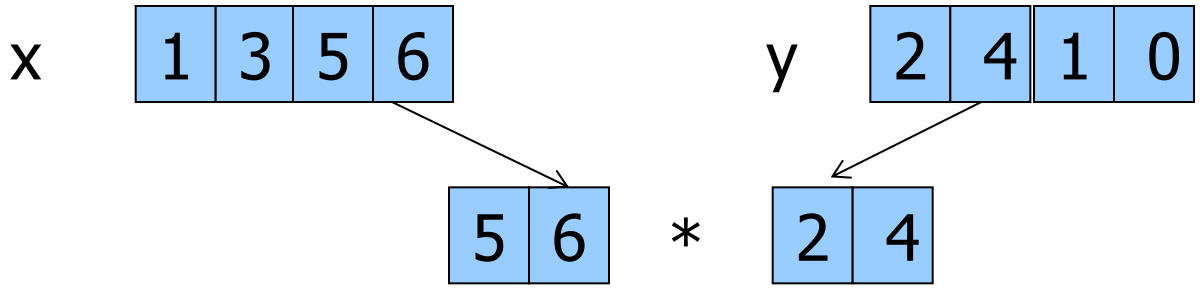


$$10^n \; x_s \; y_s \quad 10^{n/2} \; x_s \; y_d \; x_d \; y_s \; x_d \; y_d$$

$$10^2 * 1 * 1 + 10^1 * (1 * 0 + 3 * 1) + 3 * 0$$

$$13 * 10 = 10^2 * 1 + 10^1 * 3 + 0 = 130$$

$$1356 * 2410 = 3.267.960$$

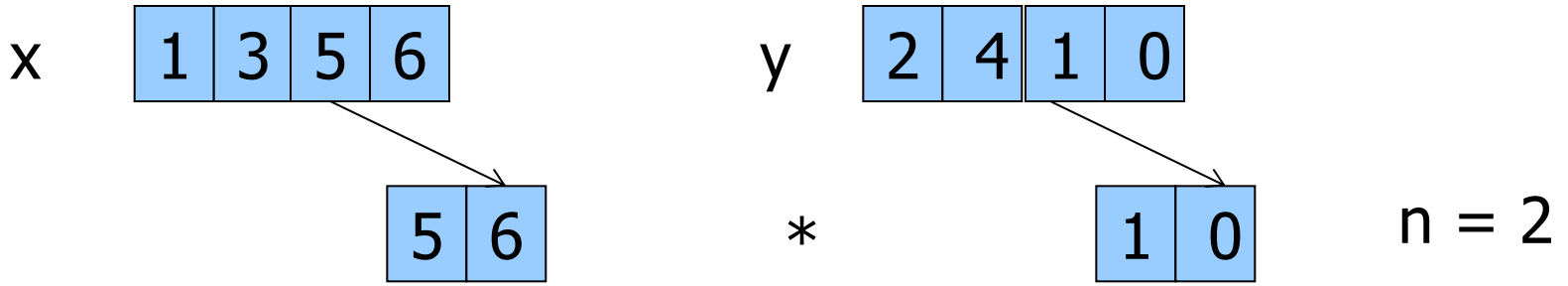


$$10^n \; x_s \; y_s \quad 10^{n/2} \; x_s \; y_d \; x_d \; y_s \quad x_d \; y_d$$

$$10^2 * 5 * 2 + 10^1 * (5 * 4 + 6 * 2) + 6 * 4$$

$$56 * 24 = 10^2 * 10 + 10^1 * 32 + 24 = 1344$$

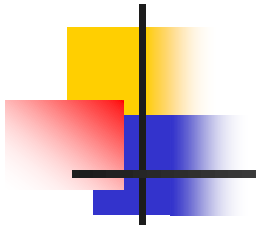
$$1356 * 2410 = 3.267.960$$



$$10^n \; x_s \; y_s \quad 10^{n/2} \; x_s \; y_d \; x_d \; y_s \quad x_d \; y_d$$

$$10^2 * 5 * 1 + 10^1 * (5 * 0 + 6 * 1) + 6 * 0$$

$$56 * 10 = 10^2 * 5 + 10^1 * 6 + 0 = 560$$



$$1356 * 2410 = 3.267.960$$

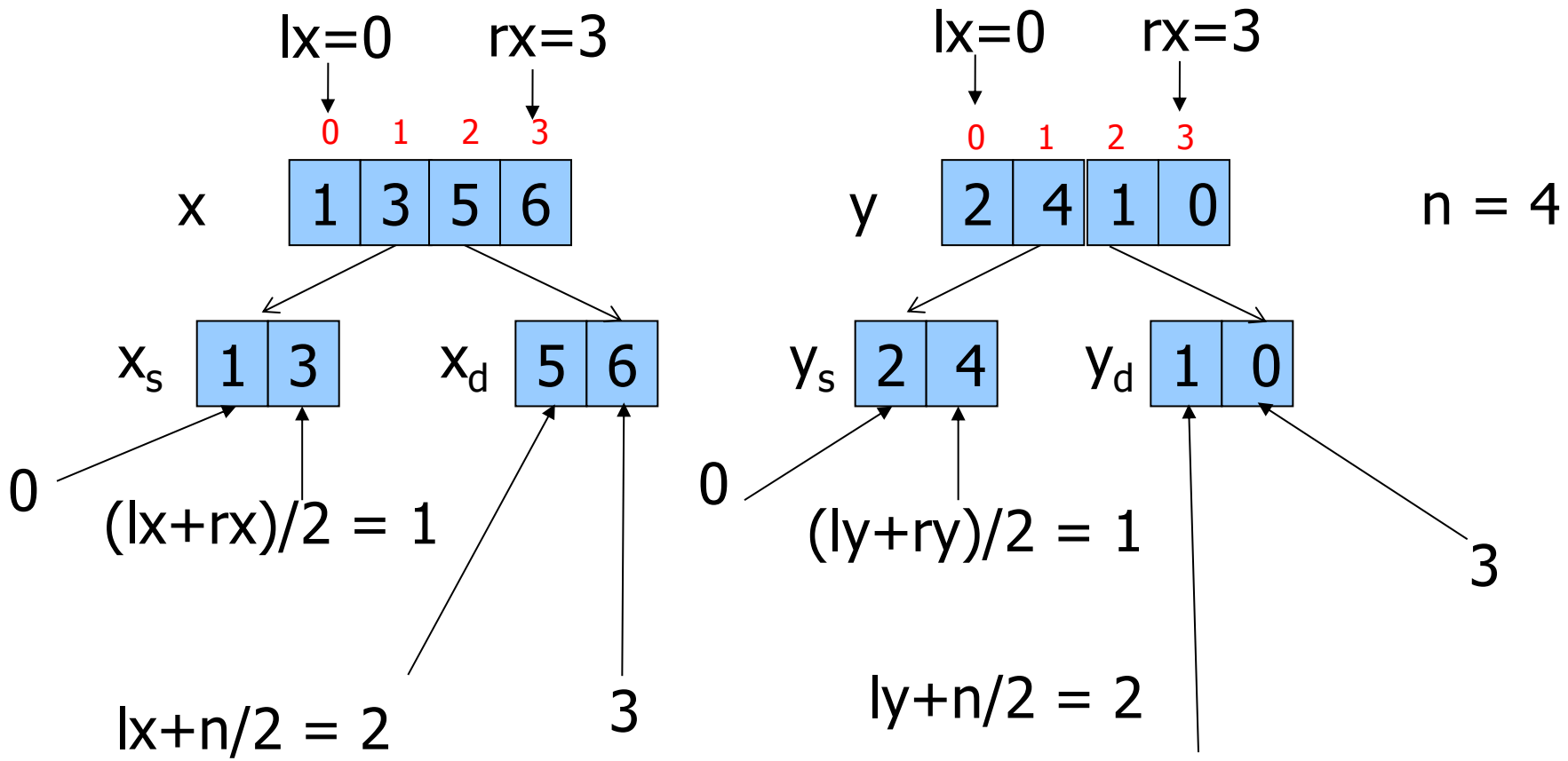
x | 1 | 3 | 5 | 6 | * | y | 2 | 4 | 1 | 0 | n = 4

10^n x_s y_s $10^{n/2}$ x_s y_d x_d y_s x_d y_d

$10^4 * 13 * 24 + 10^2 * (13 * 10 + 56 * 24) + 56 * 10$

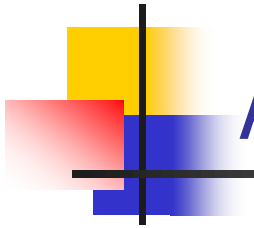
$$1356 * 2410 = 10^4 * 312 + 10^2 * (130 + 1344) + 560 = 3.267.960$$

Identificazione dei sottovettori sinistro e destro:



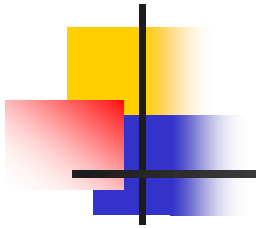
05recursive_integer_product.c

```
long prod(int *x,int lx,int rx,int *y,int ly,int ry,int n) {
  int  i, t1, t2, t3;
  if (n > 1) {
    t1 = prod(x, lx, (lx+rx)/2, y, ly, (ly+ry)/2, n/2);
    t2 = prod(x, lx, (lx+rx)/2, y, ly+n/2, ry, n/2)
         + prod(x, lx+n/2, rx, y, ly, (ly+ry)/2, n/2);
    t3 = prod(x, lx+n/2, rx, y, ly + n/2, ry, n/2);
    return t1 * pow(10,n) + t2 * pow (10, n/2) + t3;
    }
  else
    return (x[lx]*y[ly]);
}
```



Analisi di complessità

- moltiplicazione per potenza di 10: shift a sinistra (costo unitario) (2 moltiplicazioni)
- somma di numeri su n cifre: costo lineare in n (3 somme)
- moltiplicazione: costo della ricorsione (4 moltiplicazioni)



- $D(n) = \Theta(1)$, $C(n) = \Theta(n)$
- $D(n) + C(n) = \Theta(n)$
- $a = 4$, $b = 2$

Equazione alle ricorrenze:

$T(n) = 4T(n/2) + n$ $\quad n > 1$

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

Terminazione:
$n/2^i = 1$
$i = \log_2 n$

- Risoluzione per sviluppo (unfolding)

$T(n/2) = 4T(n/4) + n/2$

$T(n/4) = 4T(n/8) + n/4$ etc.

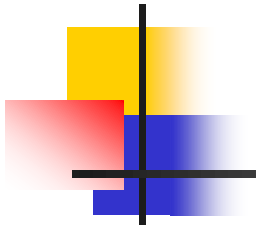
$\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x-1)$

$T(n) = n + 4*(n/2) + 4^2 *(n/4) + 4^3 *T(n/8)$

$= \Sigma_{0 \le i \le \log_2 n} 4^i / 2^i * n = n * \Sigma_{0 \le i \le \log_2 n} 2^i$

$= n*(2^{\log_2 n + 1} - 1)/(2-1) = n *(2 * 2^{\log_2 n} - 1) = 2n^2 - n$

$T(n) = O(n^2)$



divide and conquer
a = 3 b = 2

Algoritmo di Karatsuba (1962):

Riduzione del numero di moltiplicazioni:

$x_s * y_d + x_d * y_s = x_s * y_s + x_d * y_d - (x_s - x_d) * (y_s - y_d)$

3 moltiplicazioni ricorsive, anziché 4

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(n)$
- $D(n) + C(n) = \Theta(n)$
- $a = 3$, $b = 2$

Equazione alle ricorrenze:

$T(n) = 3T(n/2) + n$ $\quad n > 1$

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

Terminazione:
$n/2^i = 1$
$i = \log_2 n$

- Risoluzione per sviluppo (unfolding)

$T(n/2) = 3T(n/4) + n/2$

$T(n/4) = 3T(n/8) + n/4$ etc.

$T(n) = n + 3*(n/2) + 3^2 *(n/4) + 3^3 *T(n/8)$

$= \Sigma_{0 \le i \le \log 2n} 3^i / 2^i * n = n * \Sigma_{0 \le i \le \log 2n} (3/2)^i$

$= n*((3/2)^{\log_2 n + 1} - 1)/(3/2-1)$

$= 2n *3/2 * ((3^{\log_2 n} / 2^{\log_2 n}) - 1)$

$= 3n*(n^{\log_2 3} /n-1)$

$= 3n^{\log_2 3} - n$

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

$\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x-1)$

$a^{\log_b n} = n^{\log_b a}$

Il determinante di una matrice nxn

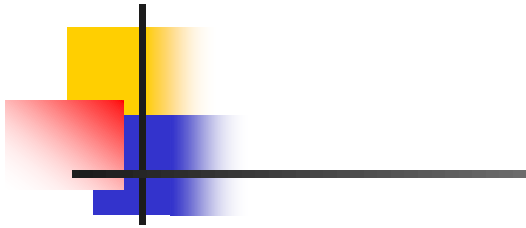
divide and conquer
$a = n \quad k_i = 2n-1$

Algoritmo di Laplace con sviluppo sulla riga i:

- matrice quadrata M nxn con indici da 1 a n

$$\det(M) = \Sigma_{1 \le j \le n} (-1)^{i+j} M[i][j] \cdot \det(M_{\text{minore } i,\, j})$$

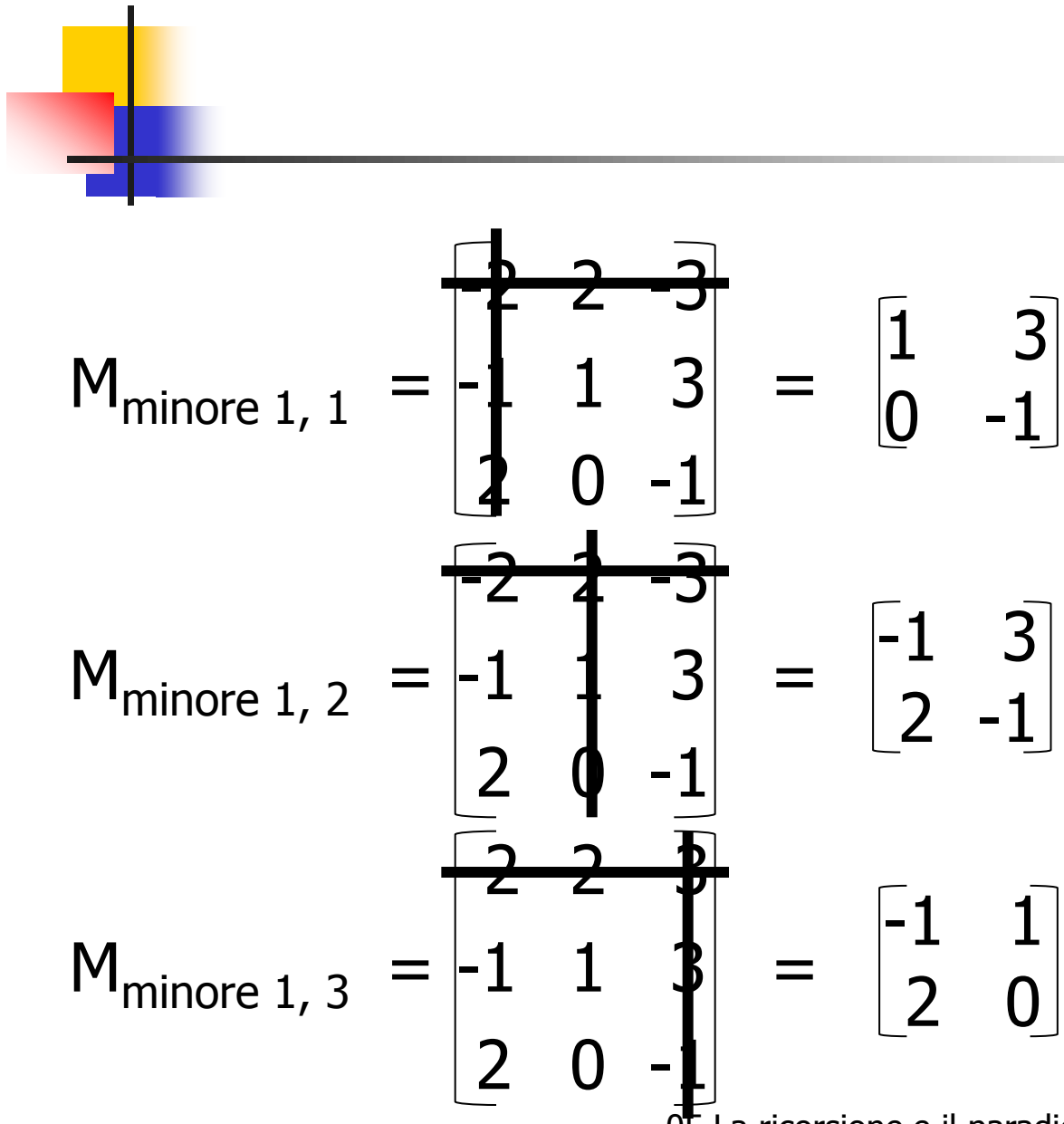
dove $M_{\text{minore } i,\, j}$ si ottiene da M eliminando la riga i e la colonna j



Esempio:

$$M = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix}$$

$$\begin{aligned} \det(M) = \quad & (-1)^{1+1} \cdot (-2) \cdot \det(M_{\text{minore } 1,\,1}) + \\ & (-1)^{1+2} \cdot (2) \cdot \det(M_{\text{minore } 1,\,2}) + \\ & (-1)^{1+3} \cdot (-3) \cdot \det(M_{\text{minore } 1,\,3}) \end{aligned}$$


$M_{\text{minore } 1,1} = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix}$
$M_{\text{minore } 1,2} = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 3 \\ 2 & -1 \end{bmatrix}$
$M_{\text{minore } 1,3} = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 2 & 0 \end{bmatrix}$

Caso elementare:

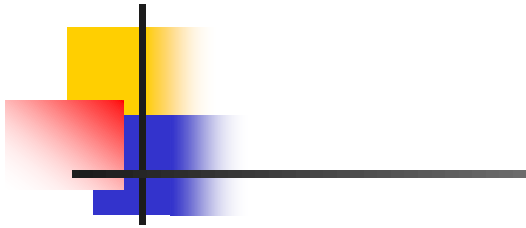
- matrice quadrata M 2x2

$$\det(M) = M[1][1] \cdot M[2][2] - M[1][2] \cdot M[2][1]$$

$$\det\left(\begin{bmatrix} 1 & 3 \\ 0 & -1 \end{bmatrix}\right) = -1 - 0 = -1$$

$$\det\left(\begin{bmatrix} -1 & 3 \\ 2 & -1 \end{bmatrix}\right) = 1 - 6 = -5$$

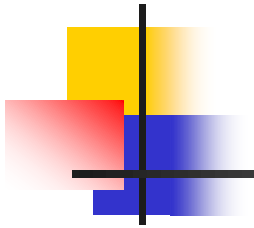
$$\det\left(\begin{bmatrix} -1 & 1 \\ 2 & 0 \end{bmatrix}\right) = 0 - 2 = -2$$



Esempio:

$$M = \begin{bmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix}$$

$$\begin{aligned} \det(M) = \; & (-1)^{1+1} \cdot (-2) \cdot \det(M_{\text{minore } 1,\,1}) + \\ & (-1)^{1+2} \cdot (2) \cdot \det(M_{\text{minore } 1,\,2}) + \\ & (-1)^{1+3} \cdot (-3) \cdot \det(M_{\text{minore } 1,\,3}) \\ = \; & (1)\cdot(-2)\cdot(-1) + (-1)\cdot(2)\cdot(-5) + (1)\cdot(-3)\cdot(-2) \\ = \; & 18 \end{aligned}$$



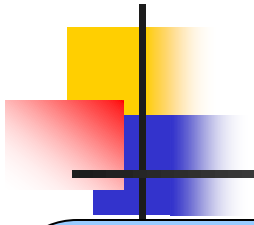
Algorimo ricorsivo (indici tra 0 e n-1):

- se n = 2, calcola M[0][0] · M[1][1] - M[0][1] · M[1][0]
- per n>2
  - con la riga a 0 e le colonne che variano tra 0 e n-1
    - scrivi in tmp $M_{\text{minore } 0,\ j}$
    - calcola ricorsivamente det($M_{\text{minore } i,\ j}$)
    - accumula il risultato in:

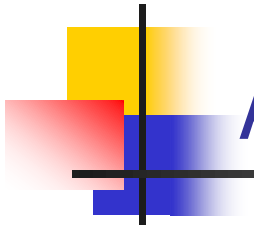
```
sum = sum + M[0][k]*pow(-1,k)*determinant(tmp,n-1);
```

06determinant.c

```
int det2x2(int m[MAX][MAX]) {
  return(m[0][0]*m[1][1] - m[0][1]*m[1][0]);
}
void minor(int m[MAX][MAX],int i,int j,int n,int m2[MAX][MAX]){
  int r, c, rr=0, cc=0;
  for (r = 0; r < n; r++)
    if (r != i) {
      for (c = 0; c < n; c++)
        if (c != j) {
           m2[rr][cc] = m[r][c];
           cc++;
        }
      if (cc == n-1) {
        rr++;
        cc = 0;
      }
    }
}
```

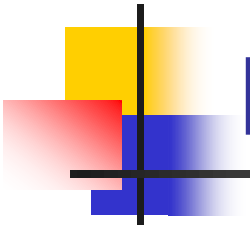


```
int determinant(int a[MAX][MAX], int n) {
  int sum, k, i, j, r, c;
  int tmp[MAX][MAX];
  sum = 0;

  if (n == 2)
    return (det2x2(a));

  for (k = 0; k < n; k++) {
    minor(a, 0, k, n, tmp);
    sum = sum + a[0][k]*pow(-1,k)*determinant(tmp,n-1);
  }
  return(sum);
}
```



Analisi di complessità

- Dimostrazione che esula dallo scopo di questo corso
- T(n) = O(N!)



Problemi ricorsivi semplici

Informatici:

- ricerca binaria o dicotomica
- stampa in ordine inverso
- elaborazione di liste concatenate
  - conteggi
  - attraversamenti
  - cancellazione
- alberi binari
  - calcolo di parametri
  - visite
  - espressioni

La ricerca binaria

divide and conquer
a = 1 b = 2

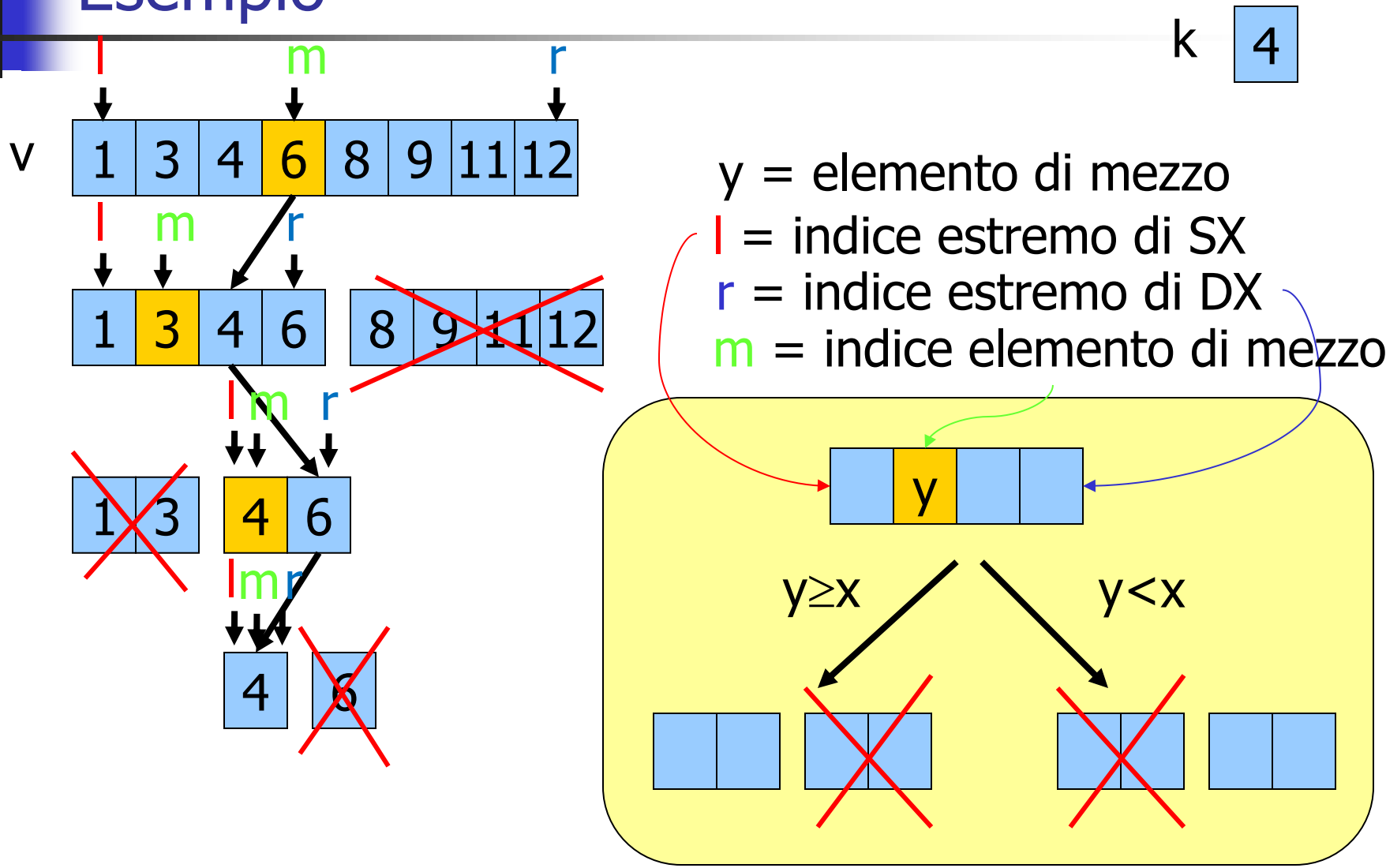
Ricerca binaria o dicotomica: la chiave k è presente all'interno di un vettore ordinato v[n]? Sì/No

ipotesi: $n = 2^p$

Approccio

- ad ogni passo: confronto k con elemento centrale del vettore:
  - =: terminazione con successo
  - <: la ricerca prosegue nel sottovettore di SX
  - >: la ricerca prosegue nel sottovettore di DX

Esempio


k
4
l
m
r
v
1 3 4 6 8 9 11 12
l m r
1 3 4 6
8 9 11 12
l m r
1 3
4 6
l m r
4
6
y = elemento di mezzo
l = indice estremo di SX
r = indice estremo di DX
m = indice elemento di mezzo
y
y≥x
y<x

07recursive_binsearch.c

```
int BinSearch(int v[], int l, int r, int k){
  int m;

  if((r-l) == 0)
    if(v[l]==k)
      return(l);
    else
      return(-1);

  m = (l+r) / 2;

  if(v[m] >= k)
    return(BinSearch(v, l, m, k));
  else
    return(BinSearch(v, m+1, r, k));
}
```

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $b = 2$

Equazione alla ricorrenze:

$T(n) = T(n/2) + 1$ $\quad n > 1$

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

Terminazione:
$n/2^i = 1$
$i = \log_2 n$

- Risoluzione per sviluppo (unfolding)

$T(n/2) = T(n/4) + 1$

$T(n/4) = T(n/8) + 1$

$T(n) = 1 + 1 + 1 + T(n/8)$

$= \sum_{i=0}^{log_2 n} 1$

$= 1 + \log_2 n$

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

decrease and conquer
$a = 1\ k_i = 1$

Stampa in ordine inverso

- Leggere da input una stringa
- Stamparla in ordine inverso

08reverse_print.c

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$

Equazione alla ricorrenze:

$T(n) = 1 + T(n-1)$ $\quad n > 1$

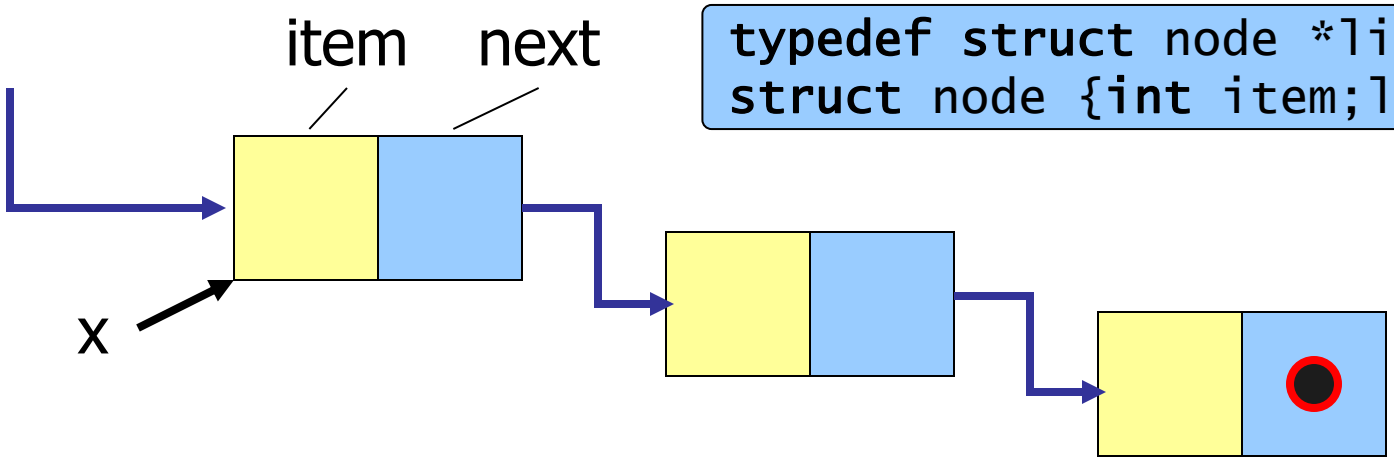
$T(1) = 1$

$T(n) = O(n)$

decrease and conquer
$a = 1\ k_i = 1$

Elaborazione di liste concatenate

Lista concatenata semplice

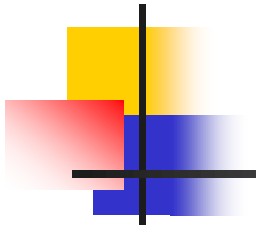



```
typedef struct node *link;
struct node {int item;link next;};
```

- Conta il numero di elementi della lista
- Attraversa la lista in ordine
- Attraversa la lista in ordine inverso
- Cancella un elemento dalla lista

09recursive_list_processing.c

```
int count (link x) {
  if (x == NULL)
    return 0;
  return 1 + count(x->next);
}
void traverse (link h) {
  if (h == NULL)
    return;
  printf("%d", h->item);
  traverse(h->next);
}
void traverseR (link h) {
  if (h == NULL)
    return;
  traverseR(h->next);
  printf("%d", h->item);
}
```



```
link delete(link x, Item v) {
  if ( x == NULL )
    return NULL;
  if ( x->item == v) {
    link t = x->next;
    free(x);
    return t;
  }
  x->next = delete(x->next, v);
}
```

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$

Equazione alla ricorrenze:

$T(n) = 1 + T(n-1)$ $\quad n > 1$

$T(1) = 1$

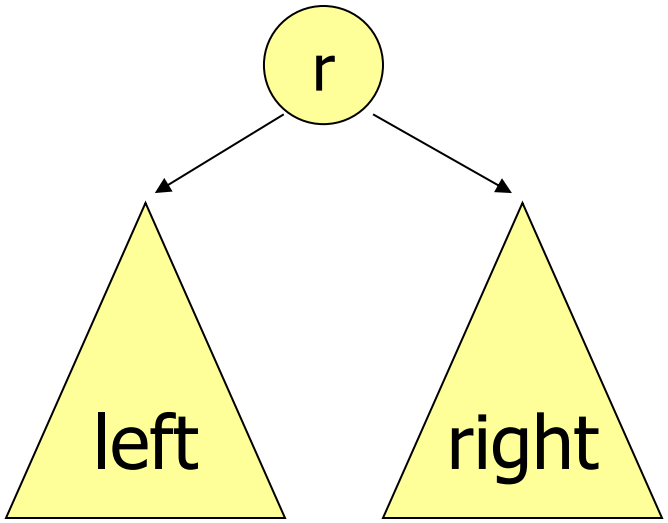
$T(n) = O(n)$

Alberi binari

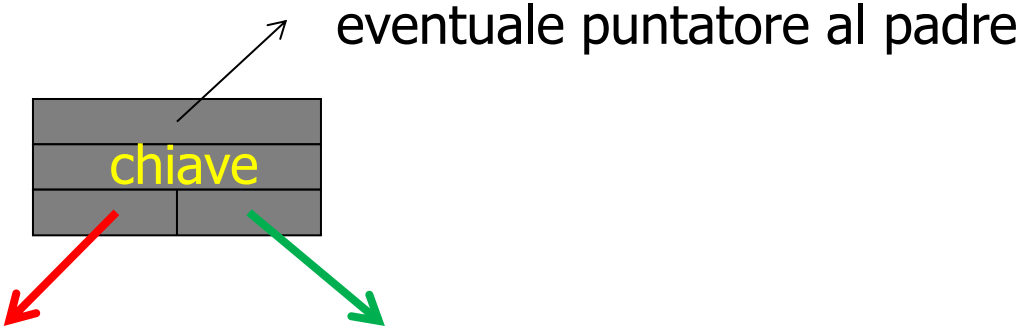
Definizione ricorsiva:

T:

- insieme di nodi vuoto
- radice, sottoalbero sinistro, sottoalbero destro.



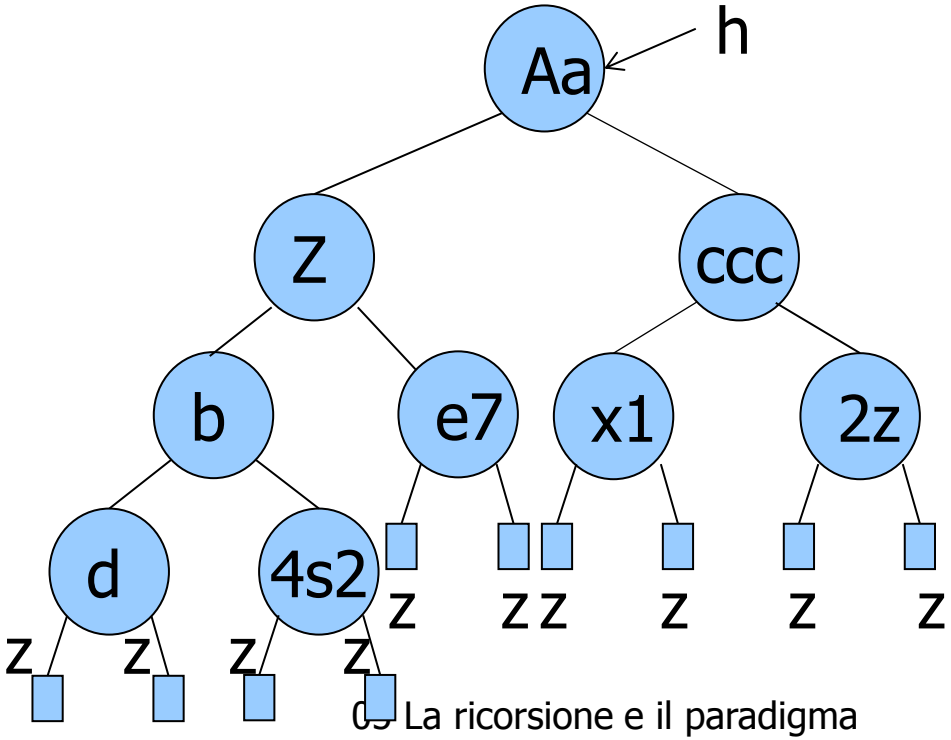
Nodo:




```
typedef struct node* link;
struct node {
  char *name;
  link l;
  link r;
};
```

Albero:

- accesso tramite il puntatore h alla radice
- nodo sentinella fittizio z



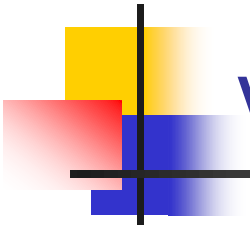
Calcolo di parametri

numero di nodi

```
int count(link h, link z) {
 if (h == z)
   return 0;
 return count(h->l, z) + count(h->r, z) + 1;
}
```

altezza

```
int height(link h, link z) {
 int u, v;
 if (h == z)
   return -1;
 u = height(h->l, z); v = height(h->r, z);
 if (u>v)
   return u+1;
 else
   return v+1;
}
```

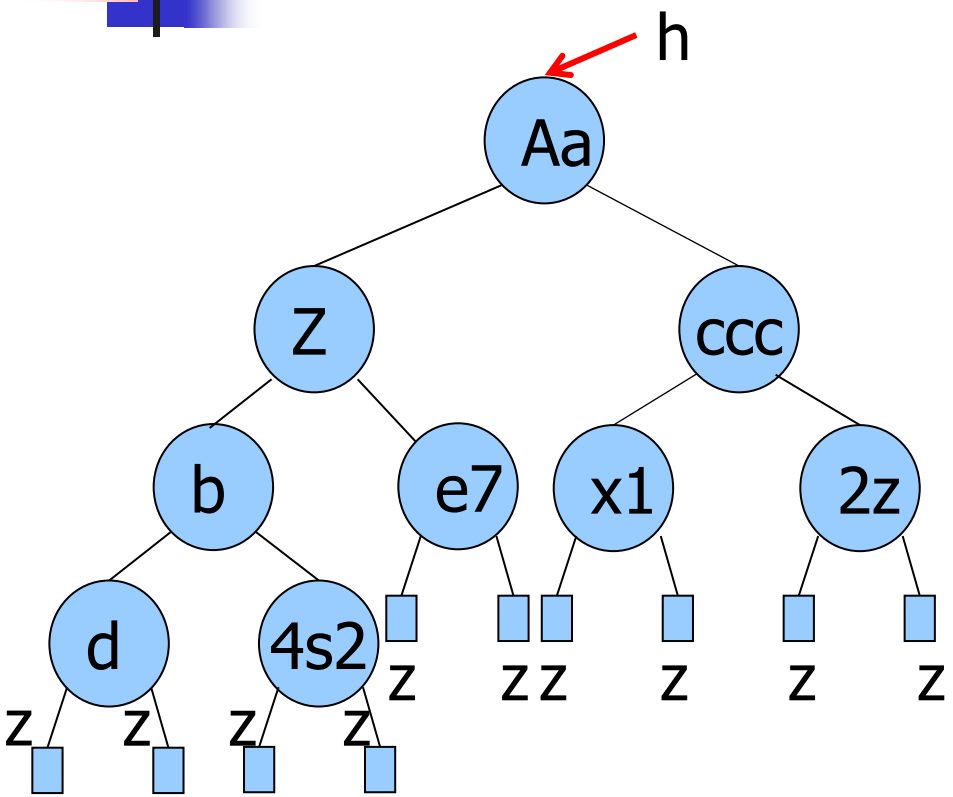


Visite

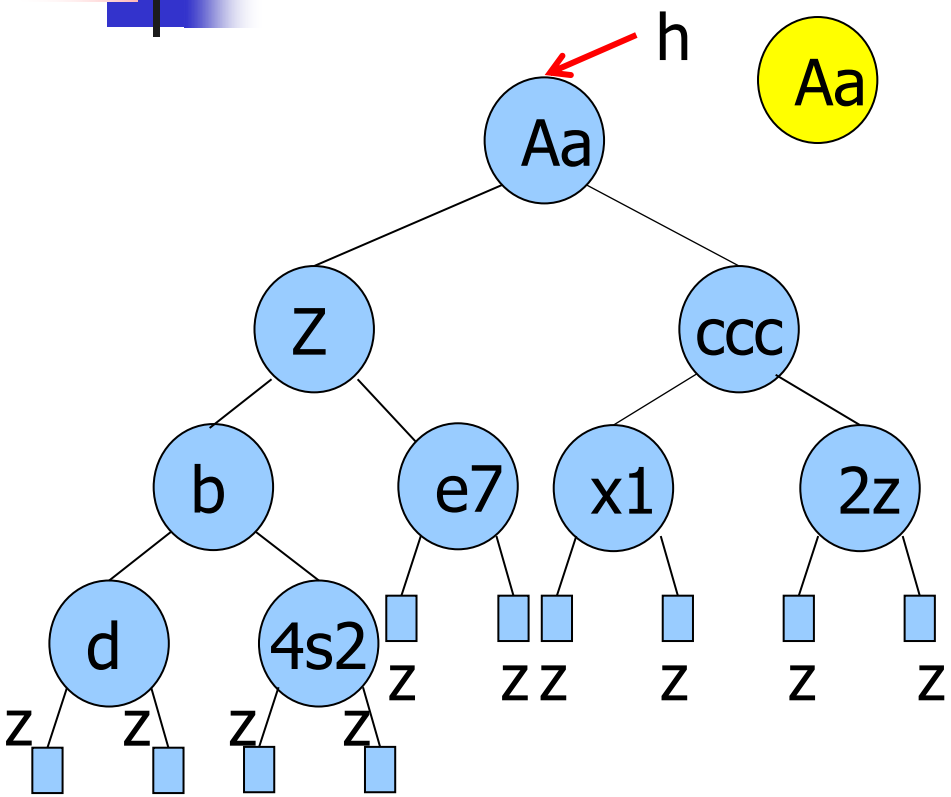
Attraversamento o visita: elenco dei nodi secondo una strategia:

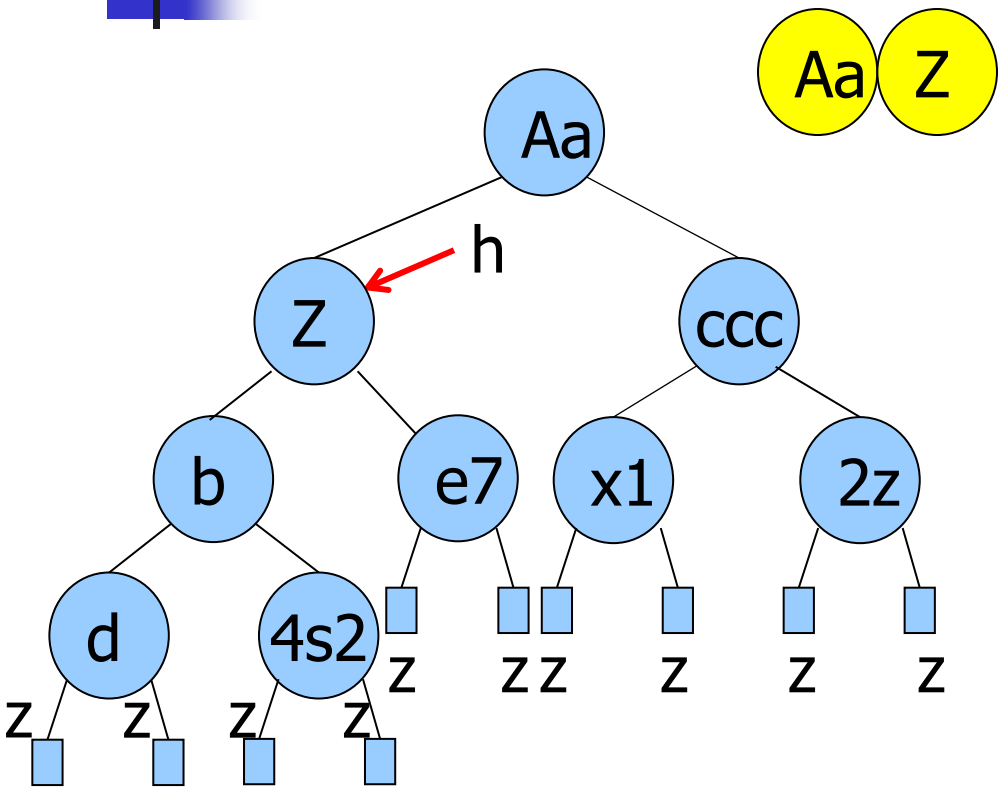
- pre-ordine: h, Left(h), Right(h)
- in-ordine: Left(h), h, Right(h)
- post-ordine: Left(h), Right(h), h

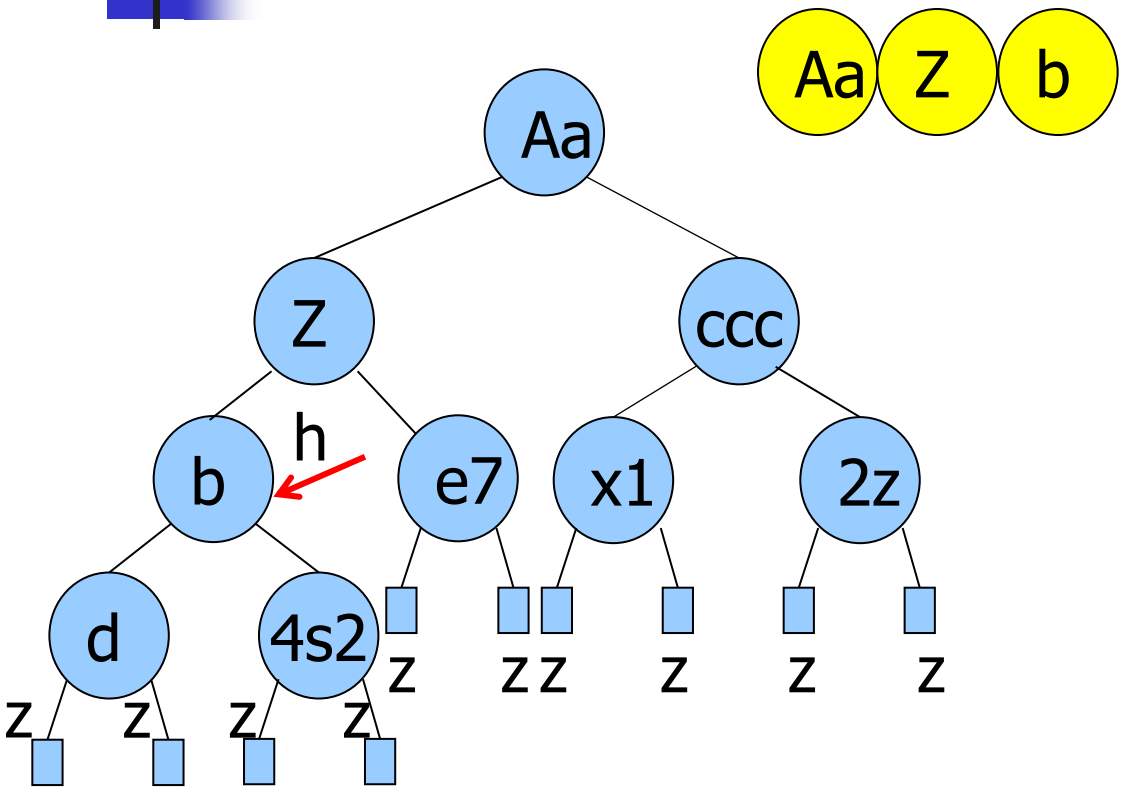
Pre-ordine

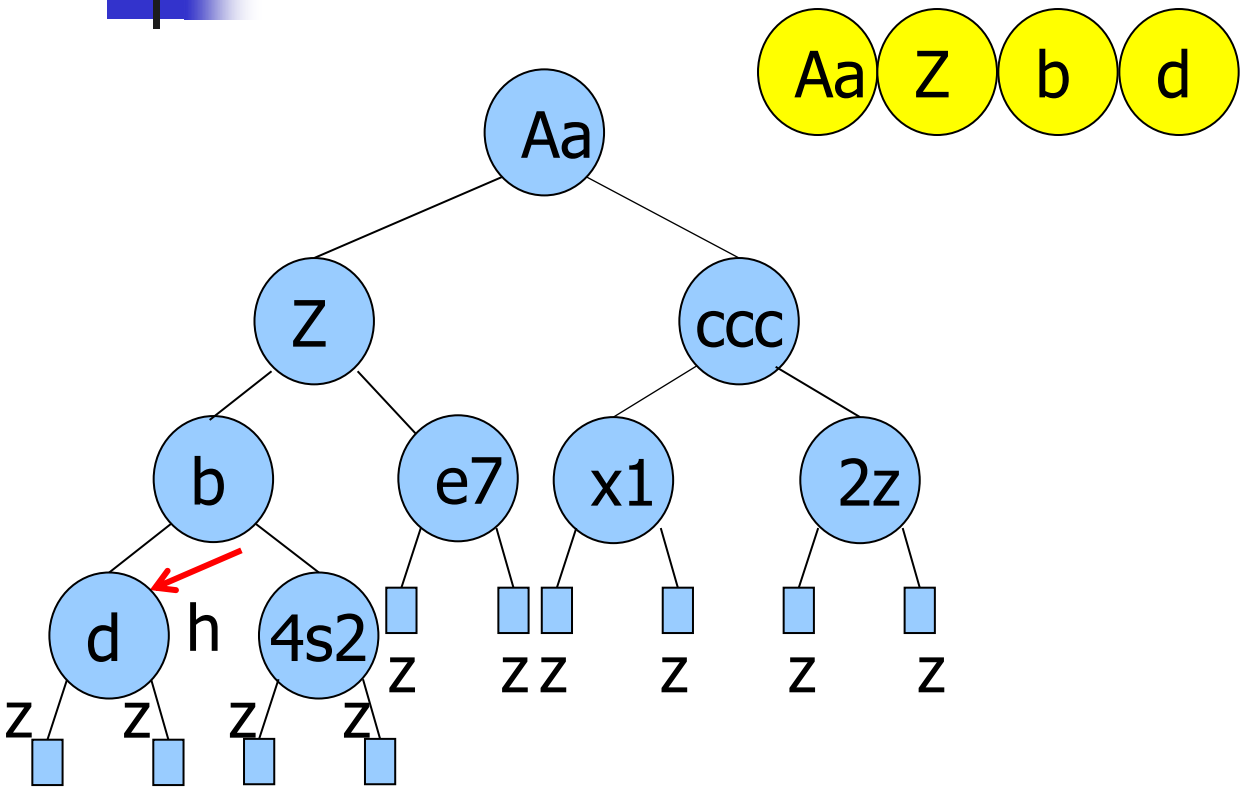


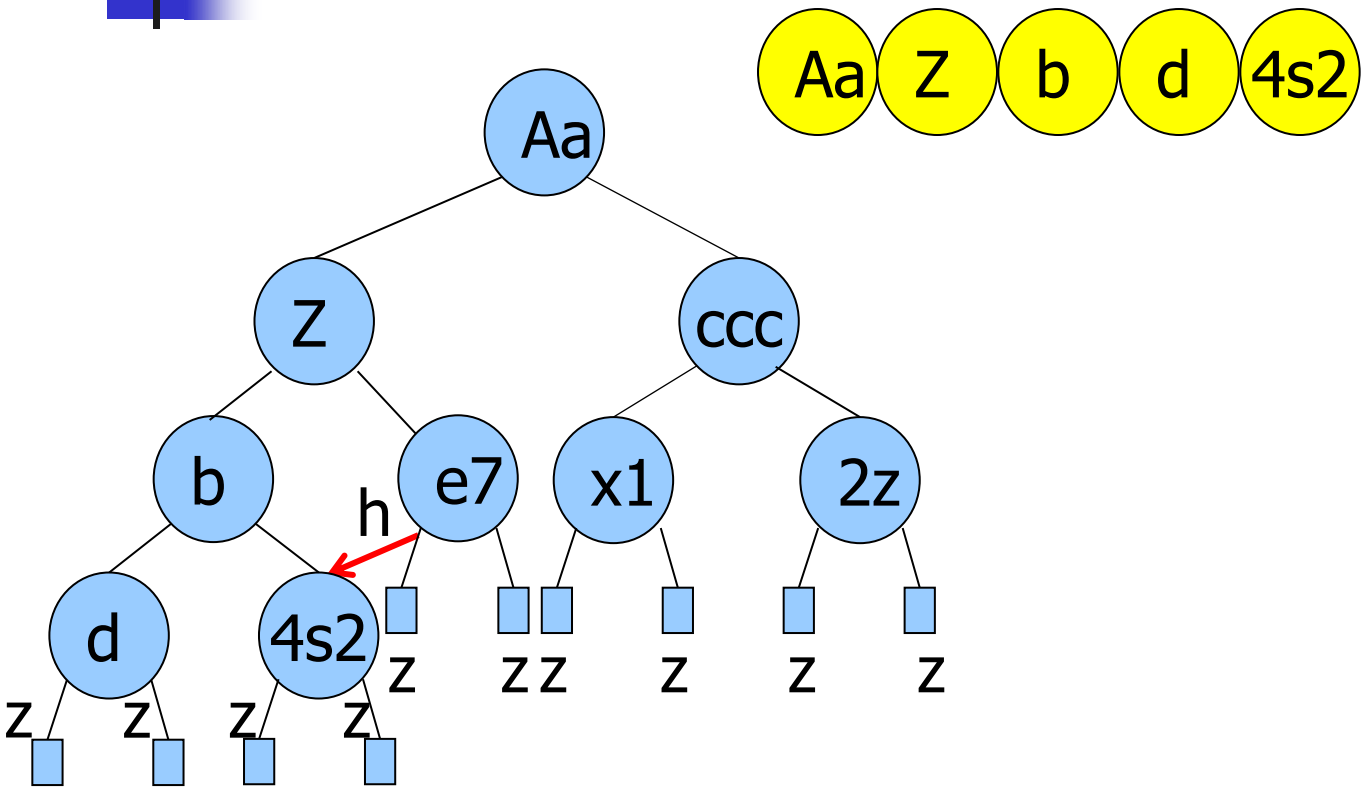

```
void preorderR(link h,link z){
  if (h == z)
    return;
  printf("%s ",h->name);
  preorderR(h->l, z);
  preorderR(h->r, z);
}
```

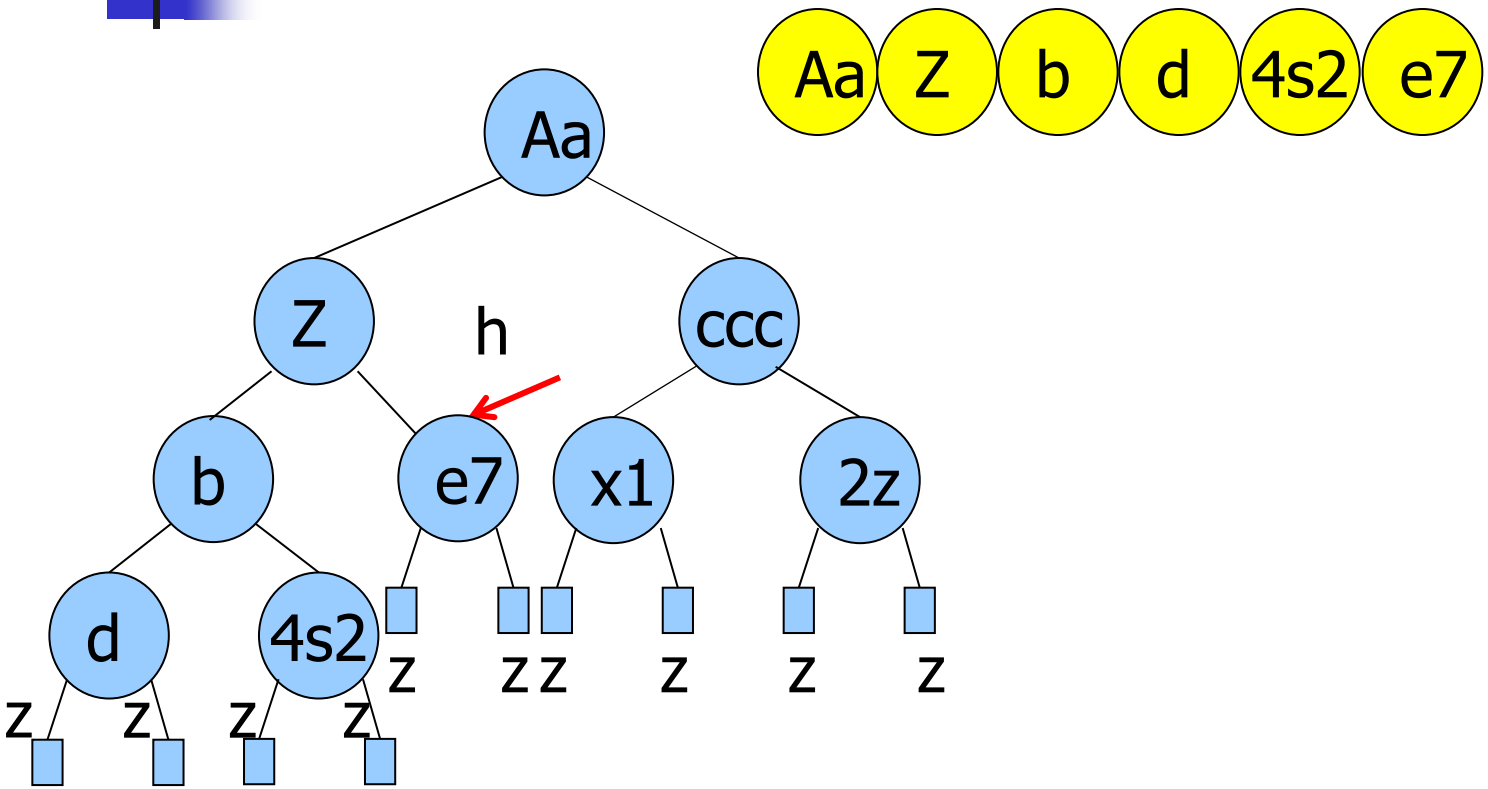

h
Aa
Aa
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

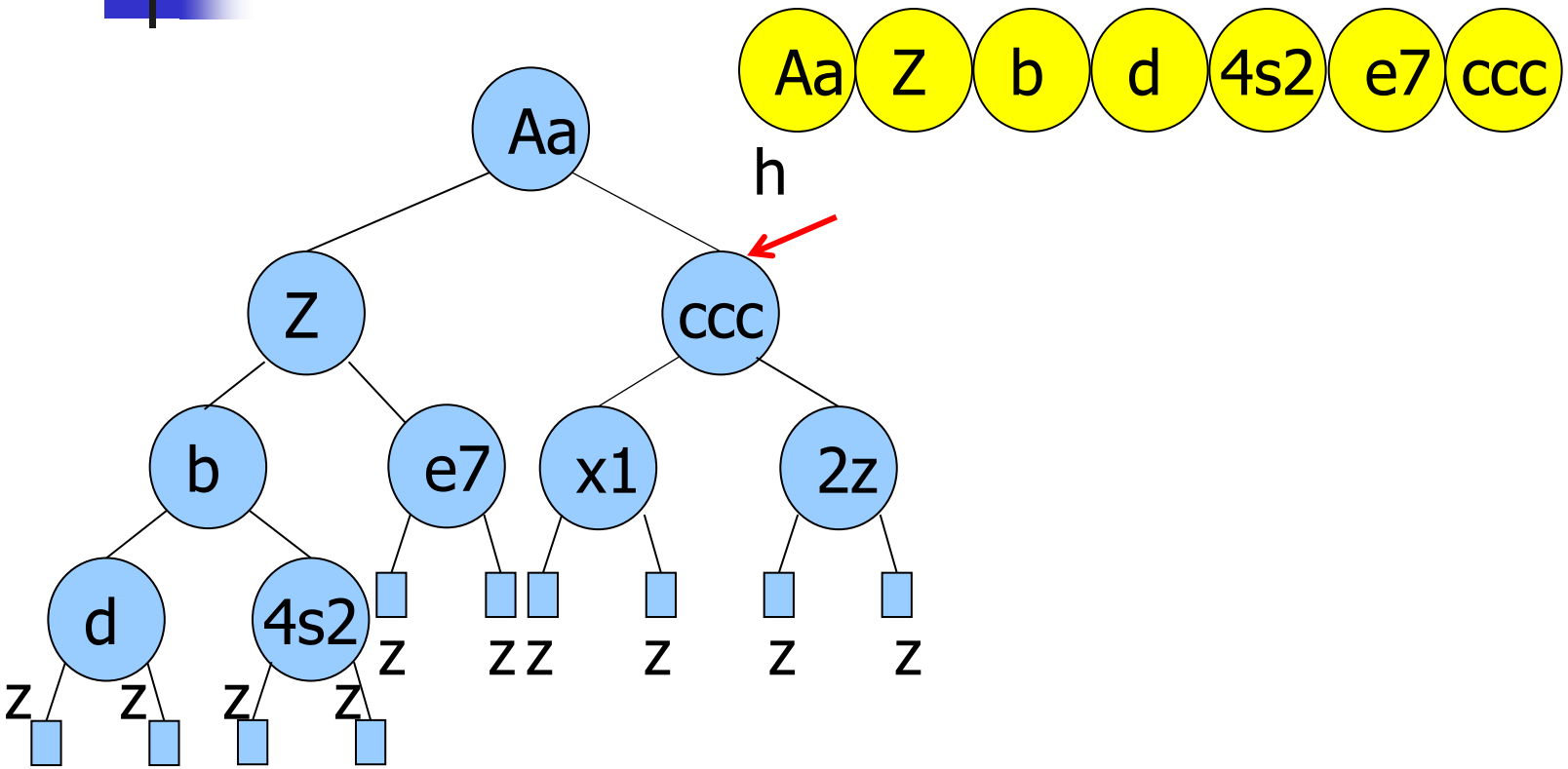

Aa
Z
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

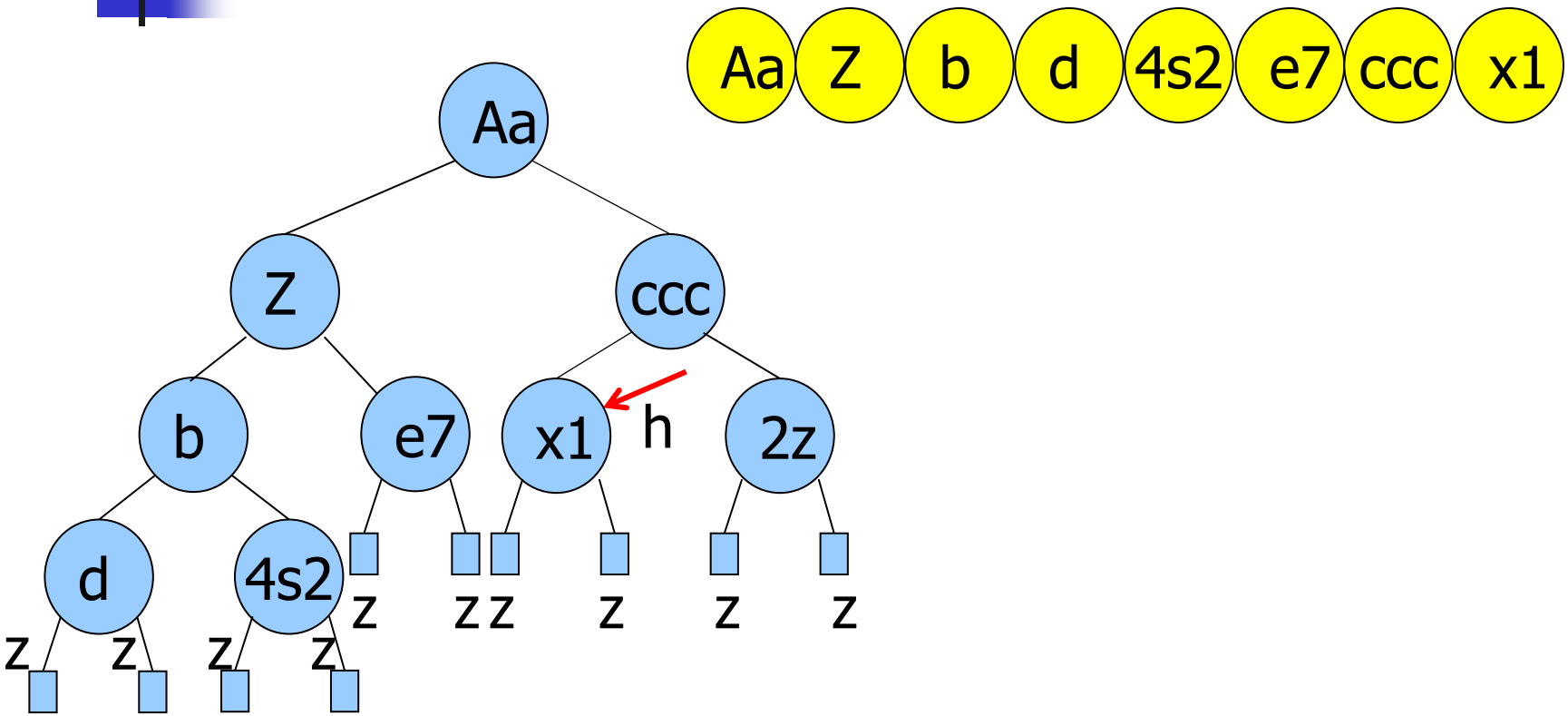

Aa
Z
b
Aa
Z
ccc
b
h
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

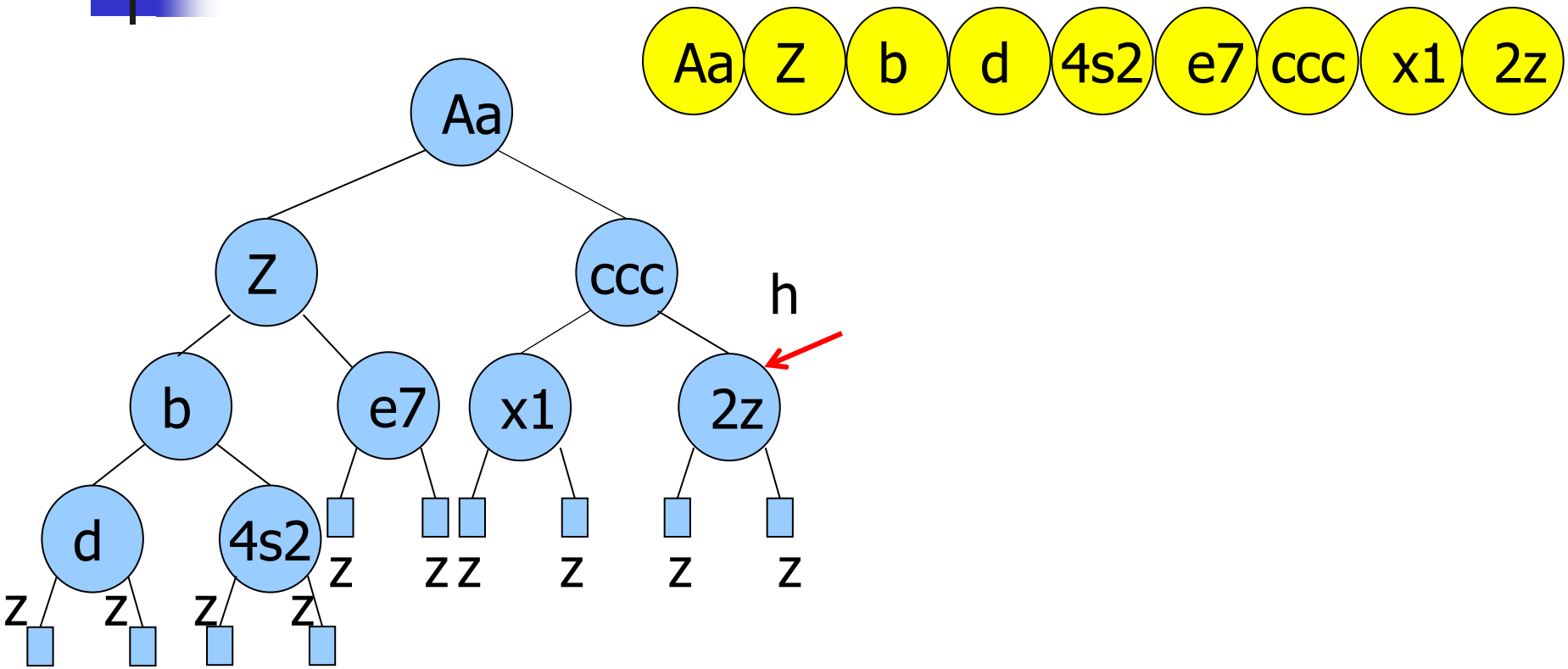

Aa
Z
b
d
Aa
Z
ccc
b
e7
x1
2z
d
h
4s2
z
z z
z
z
z
z
z
z
z


Aa
Z
b
d
4s2
Aa
Z
ccc
b
e7
x1
2z
h
d
4s2
z
z
z
z
z
z
z
z
z
z
z
z

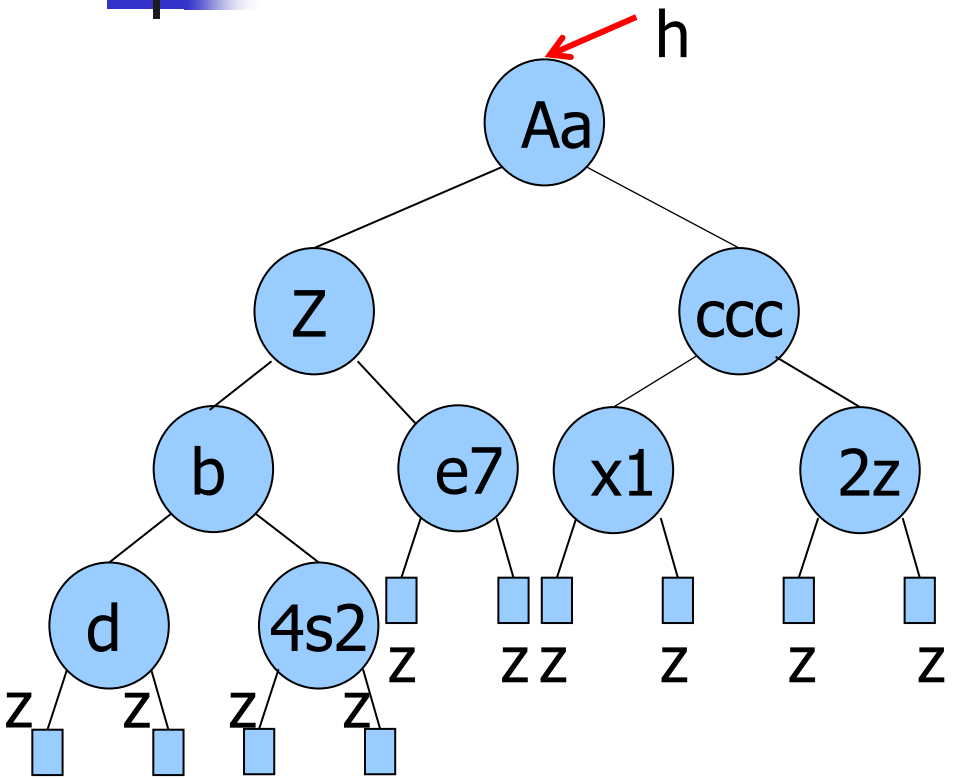

Aa
Z
b
d
4s2
e7
Aa
Z
h
ccc
b
e7
x1
2z
d
4s2
z
z
z
z
z
z
z
z
z
z
z
z


Aa
Z
b
d
4s2
e7
ccc
h
Aa
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

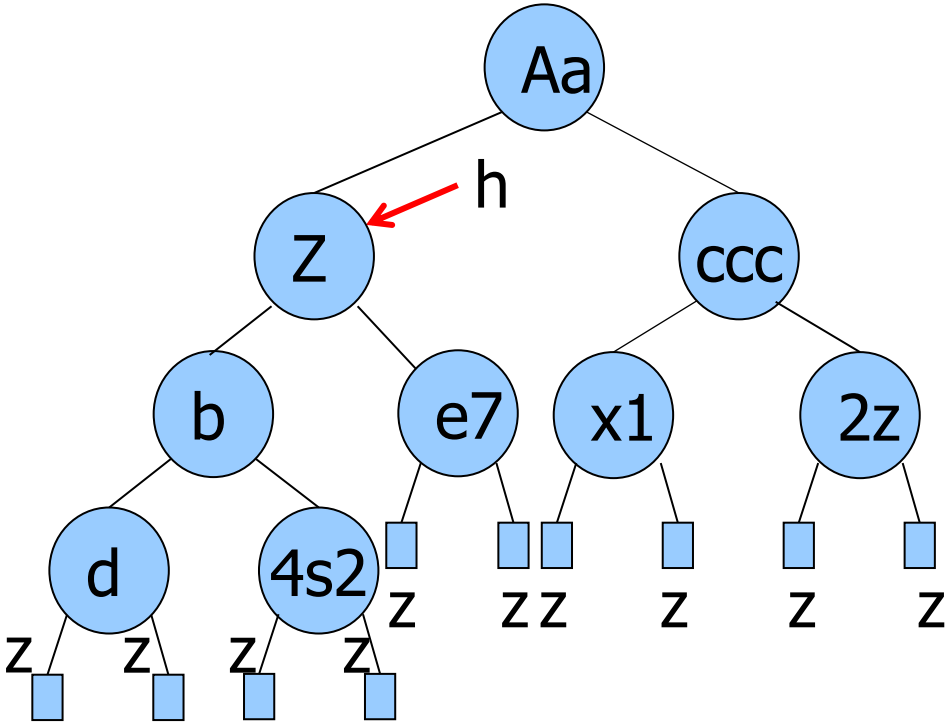

Aa
Z
b
d
4s2
e7
ccc
x1
Aa
Z
ccc
b
e7
x1
h
2z
d
4s2
z
z
z
z
z
z
z
z
z
z
z
z

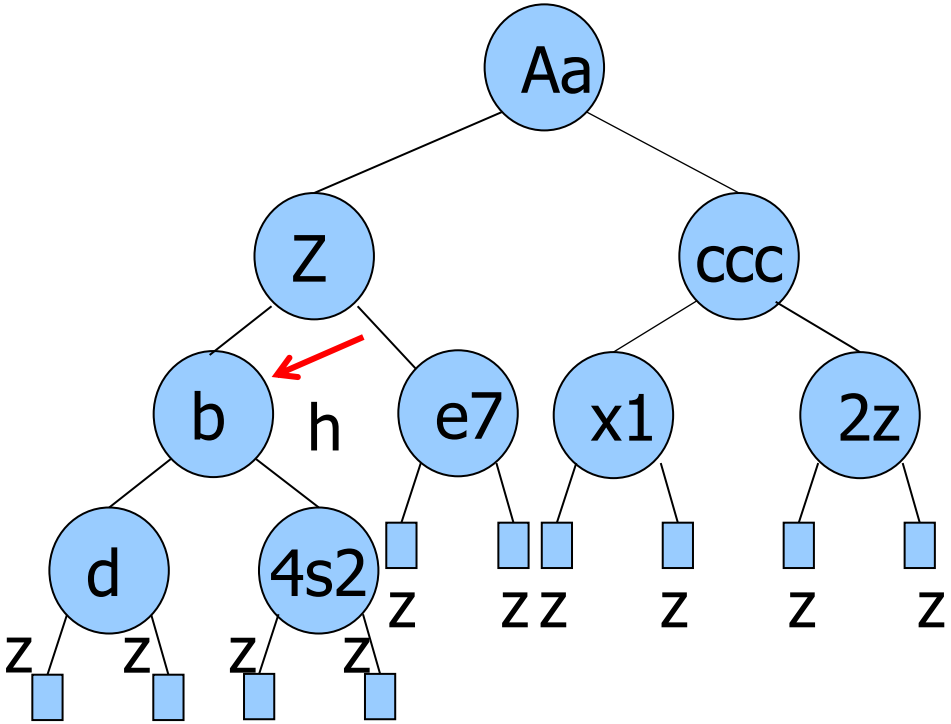

Aa
Z
b
d
4s2
e7
ccc
x1
2z
Aa
Z
ccc
h
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z
z

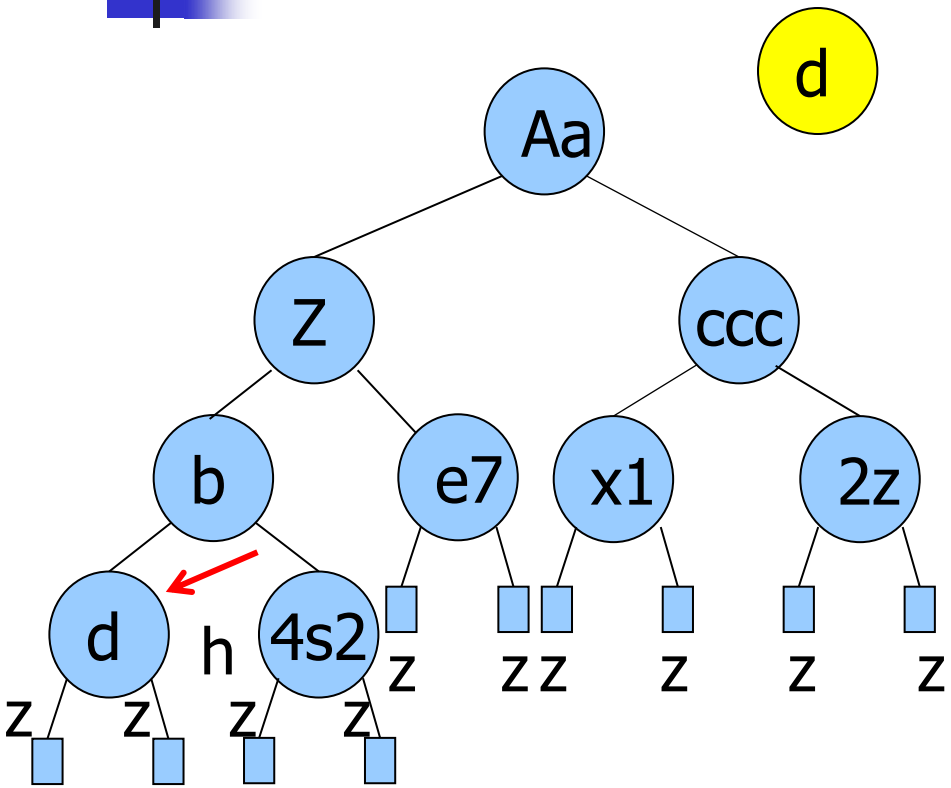
In-ordine

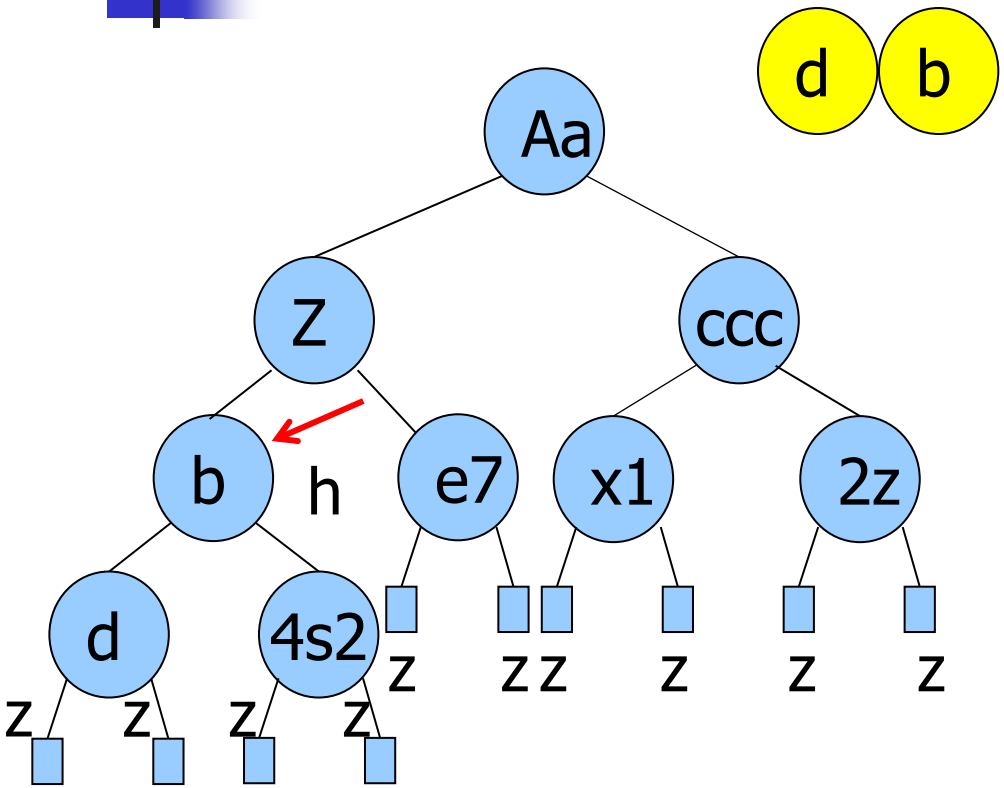


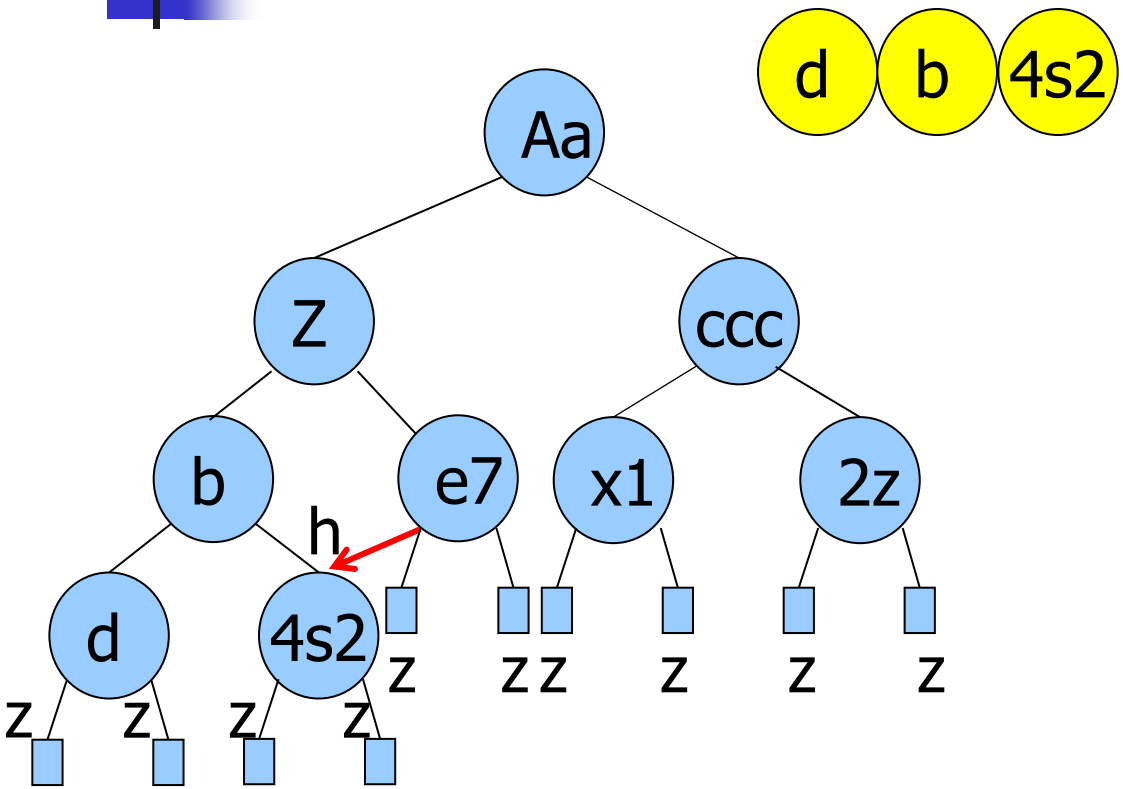

```
void inorderR(link h,link z){
  if (h == z)
    return;
  inorderR(h->l, z);
  printf("%s ",h->name);
  inorderR(h->r, z);
}
```

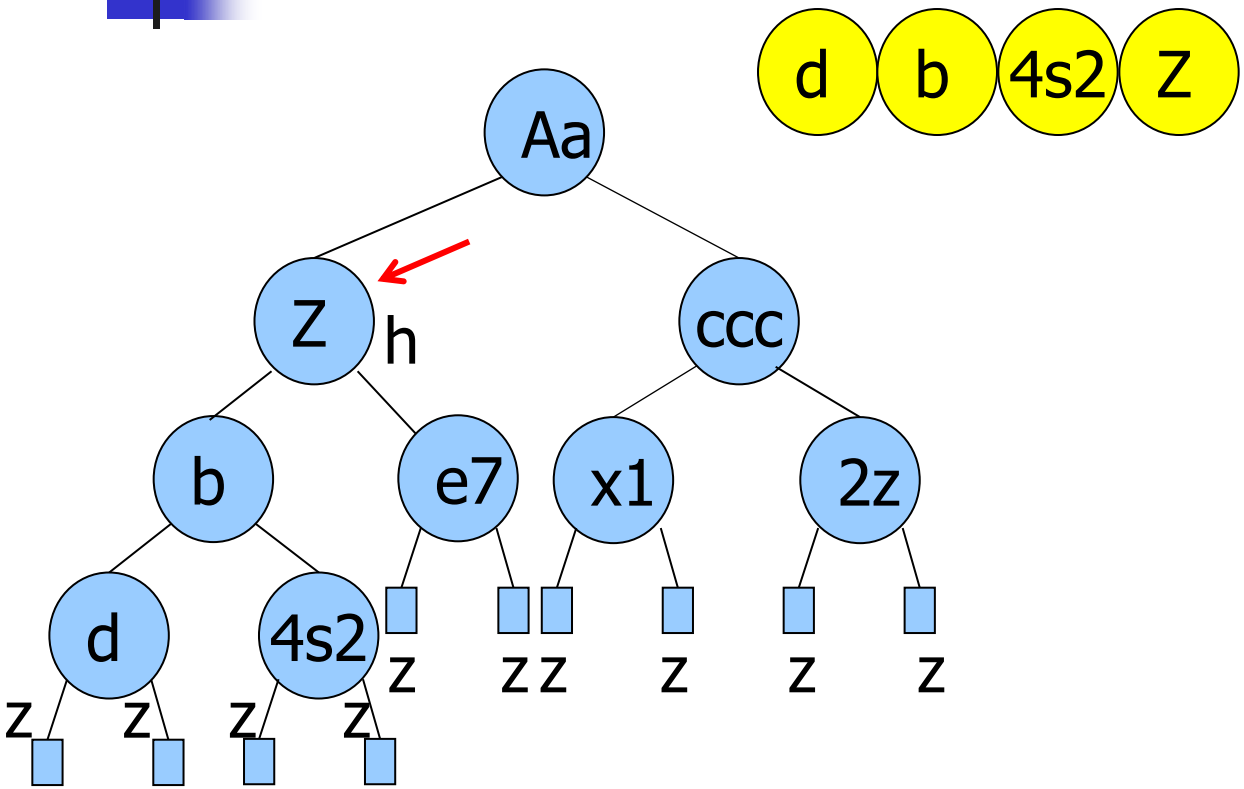

Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

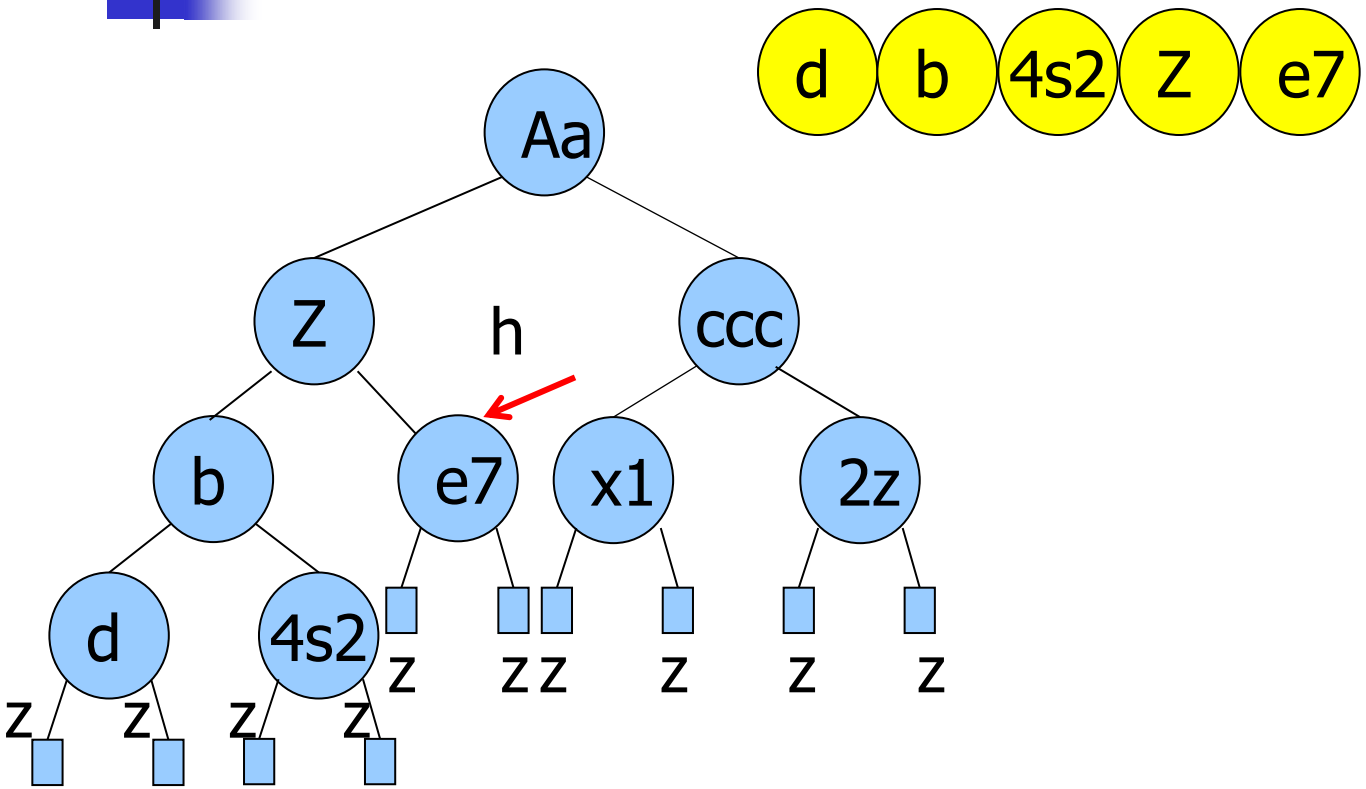

Aa
Z
ccc
b
h
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

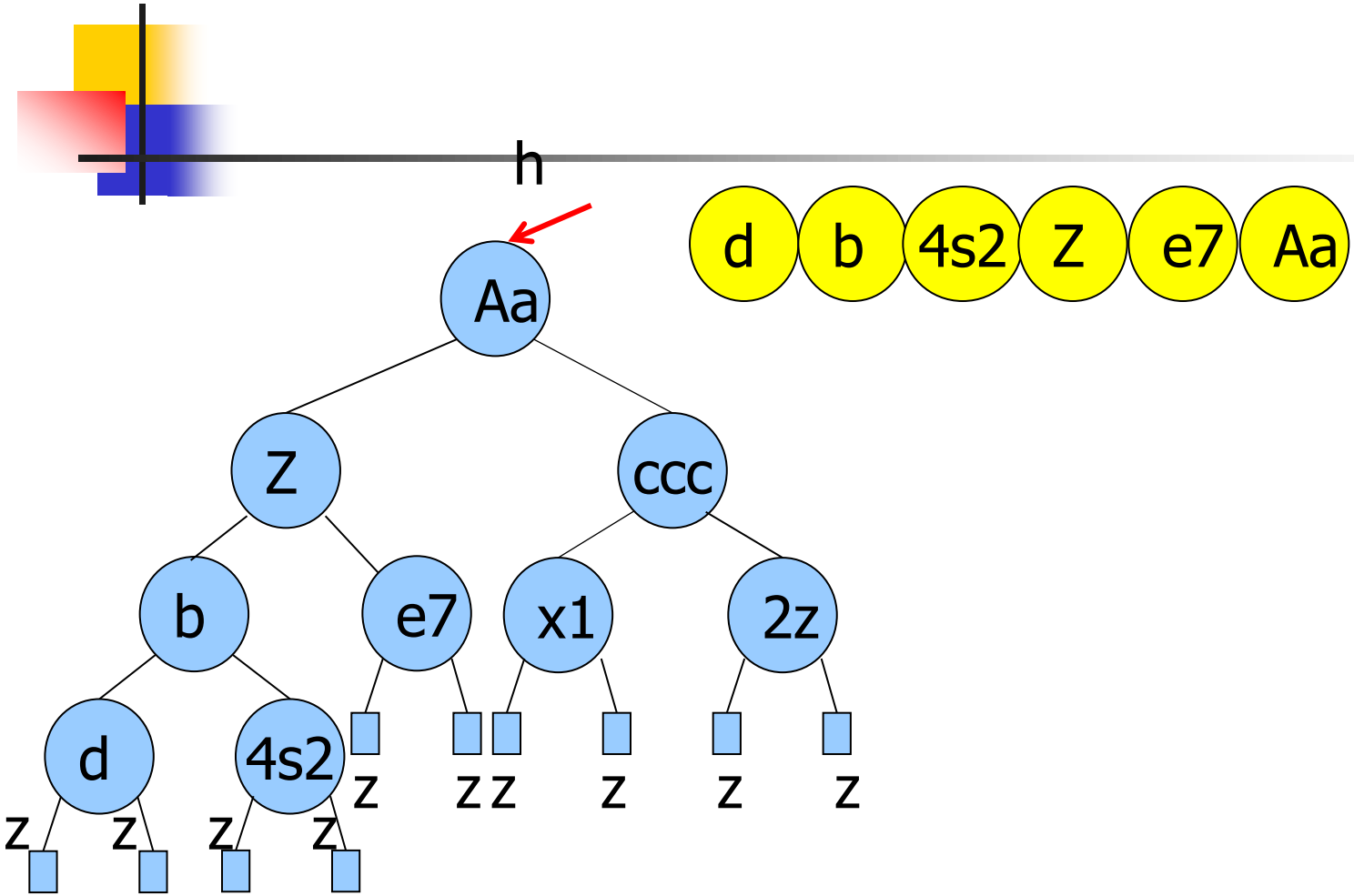

d
Aa
Z
ccc
b
e7
x1
2z
d
h
4s2
z
z z
z
z
z
z
z
z
z

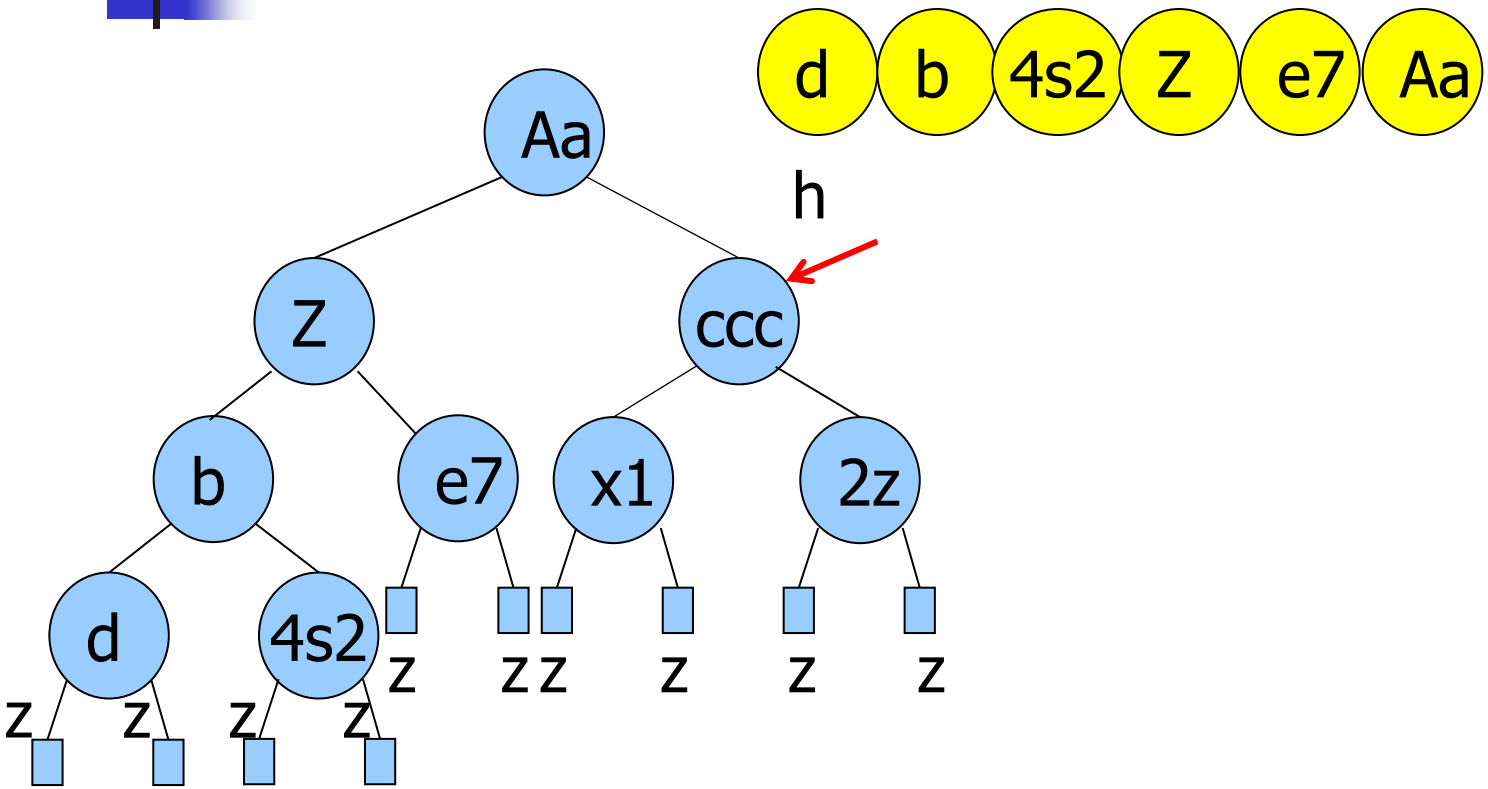

d
b
Aa
Z
ccc
b
h
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

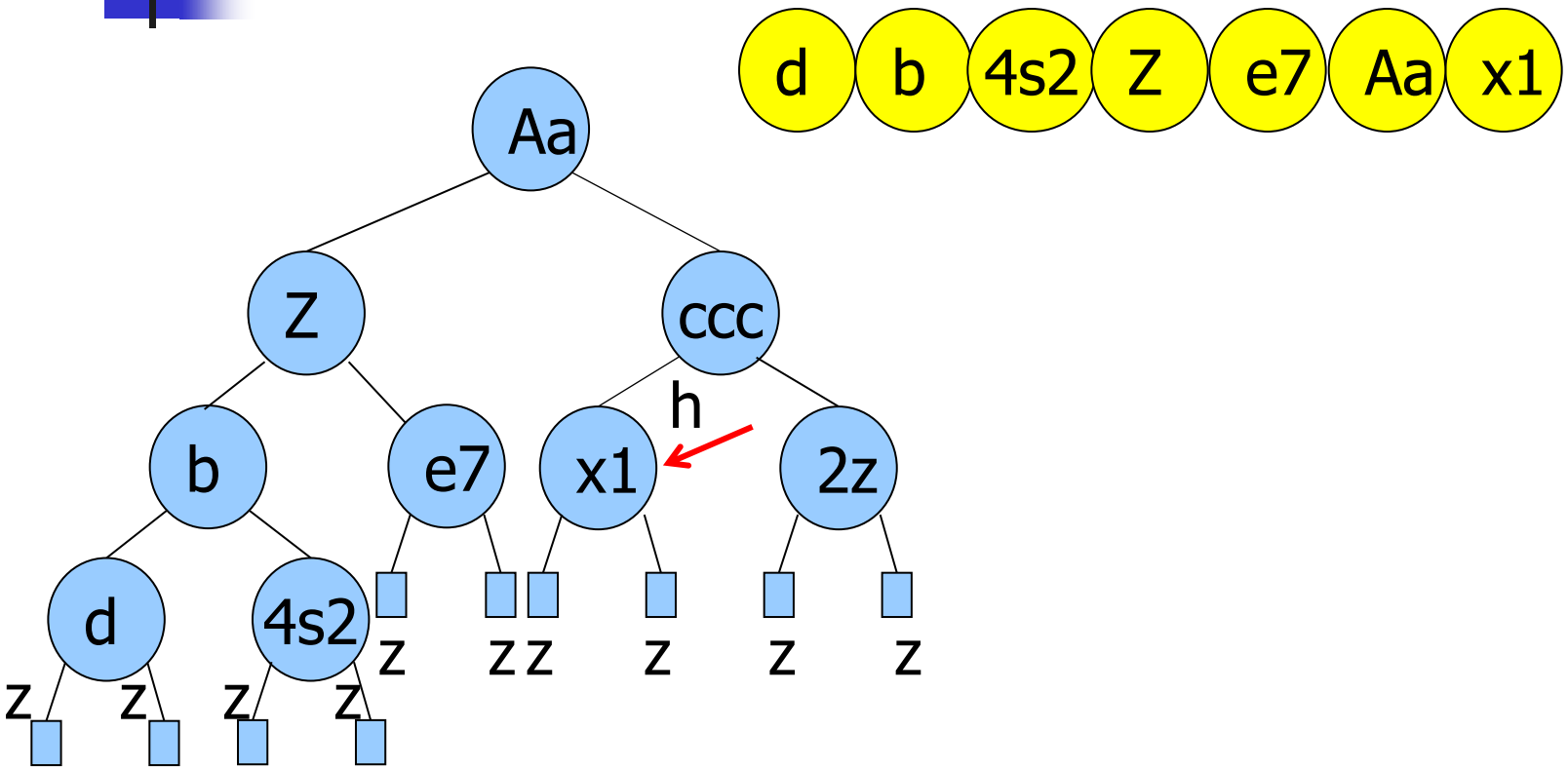

d
b
4s2
Aa
Z
ccc
b
e7
x1
2z
h
d
4s2
z
z z
z
z
z
z
z
z
z

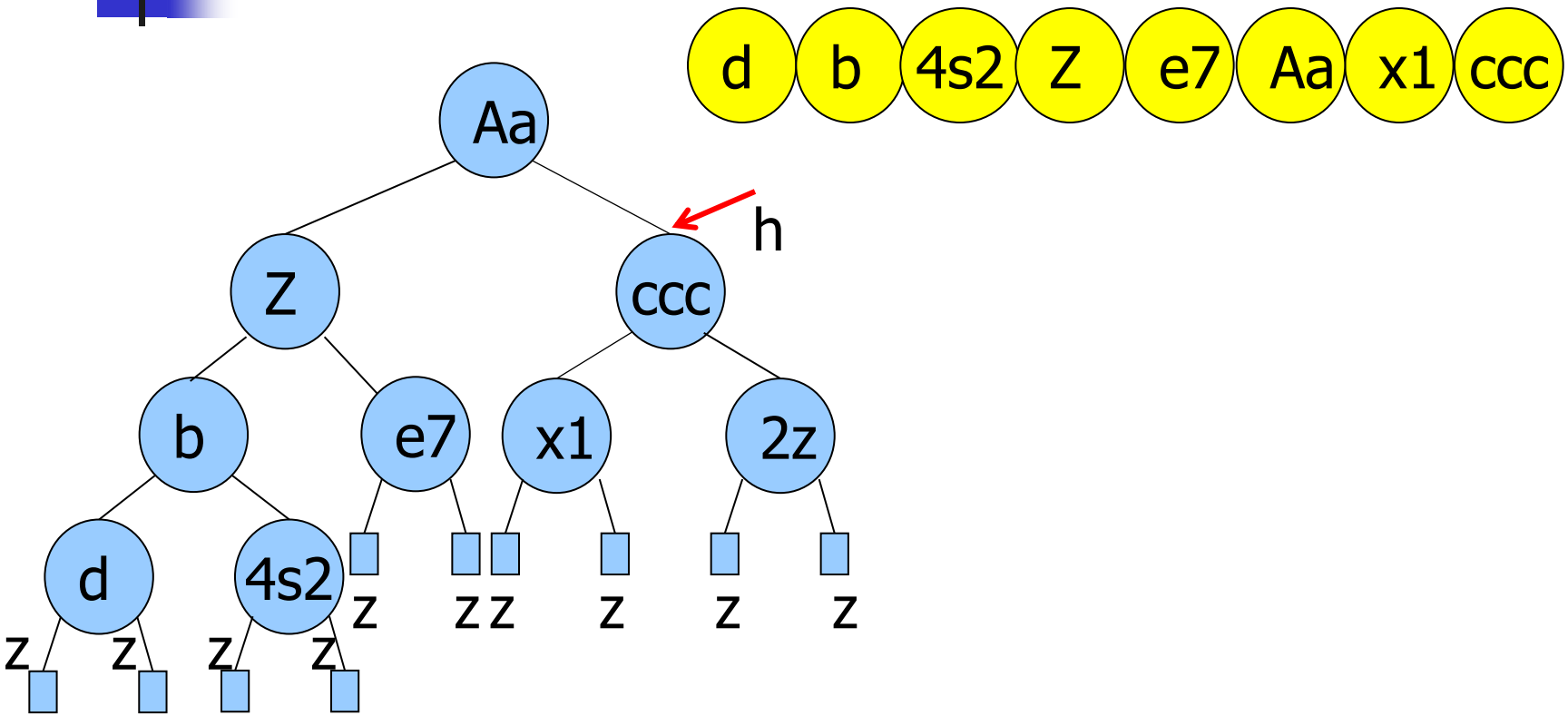

d
b
4s2
Z
Aa
Z
h
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

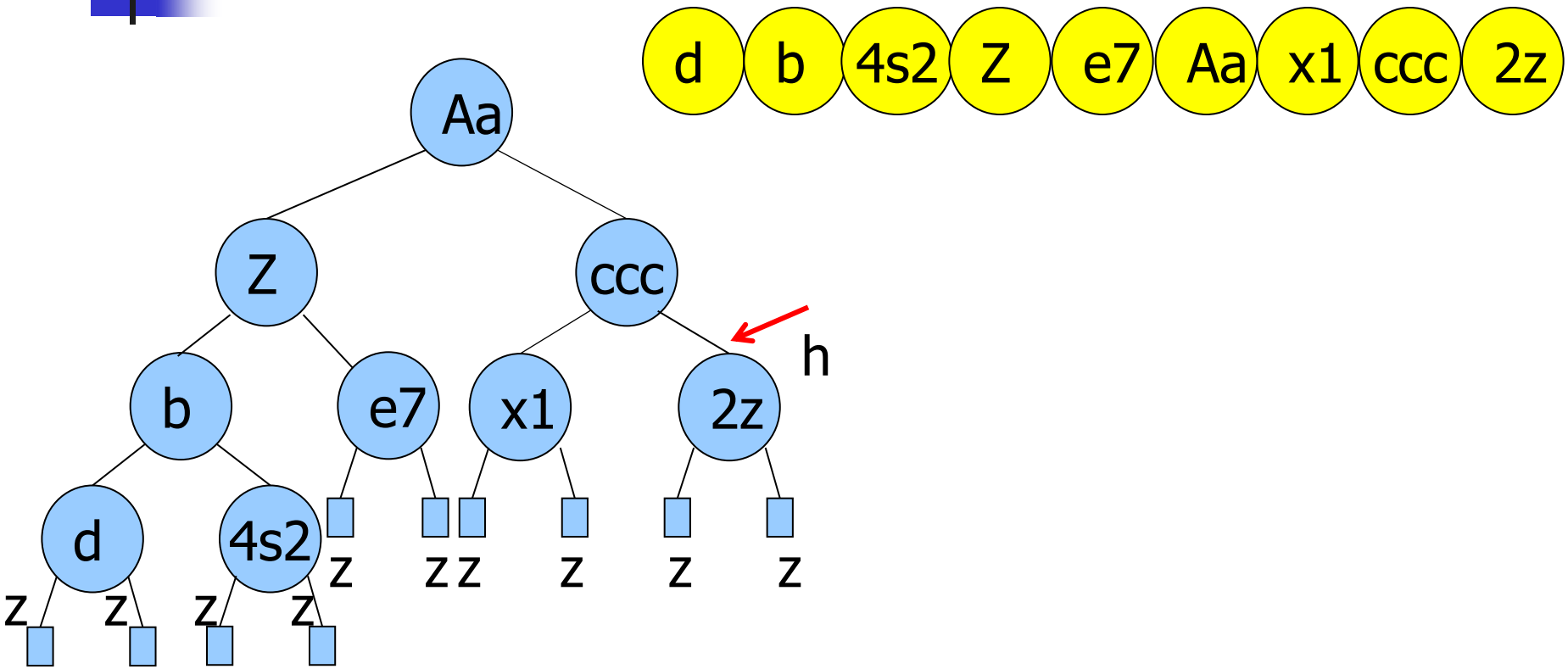

d
b
4s2
Z
e7
Aa
Z
h
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z
z


h
d
b
4s2
Z
e7
Aa
Aa
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z
z

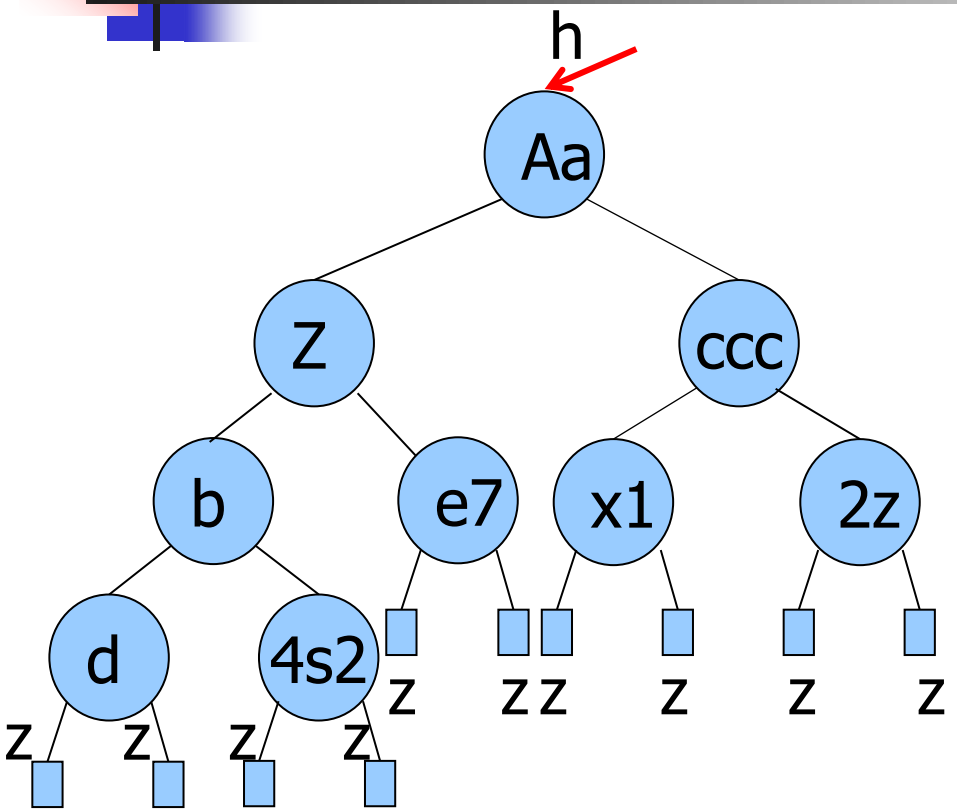

d
b
4s2
Z
e7
Aa
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z


d
b
4s2
Z
e7
Aa
x1
Aa
Z
ccc
h
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z
z

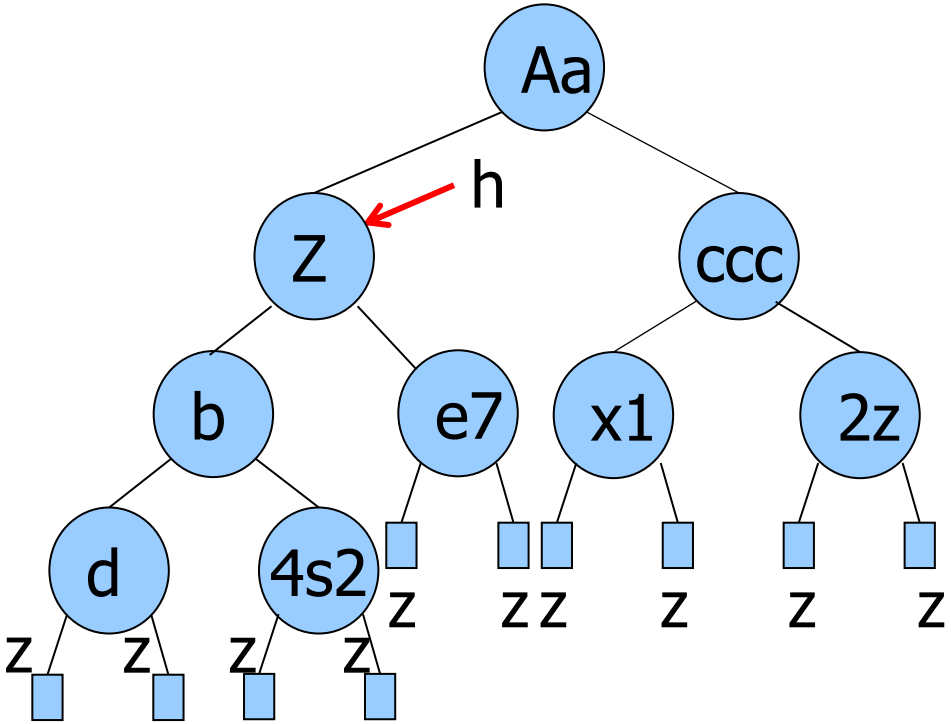

d
b
4s2
Z
e7
Aa
x1
ccc
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

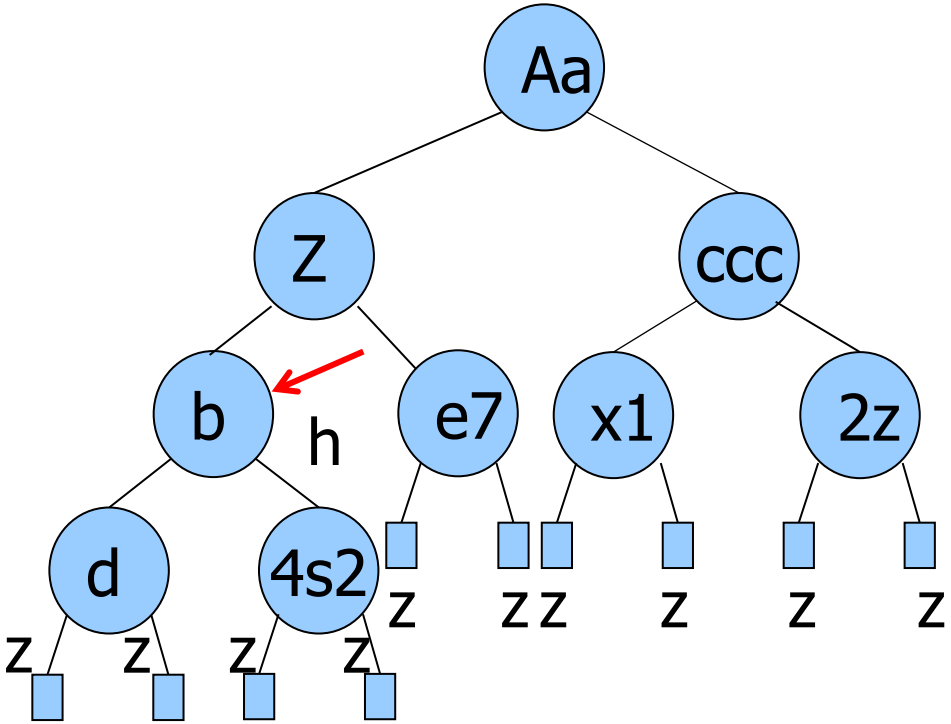

d
b
4s2
Z
e7
Aa
x1
ccc
2z
Aa
Z
ccc
h
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z
z

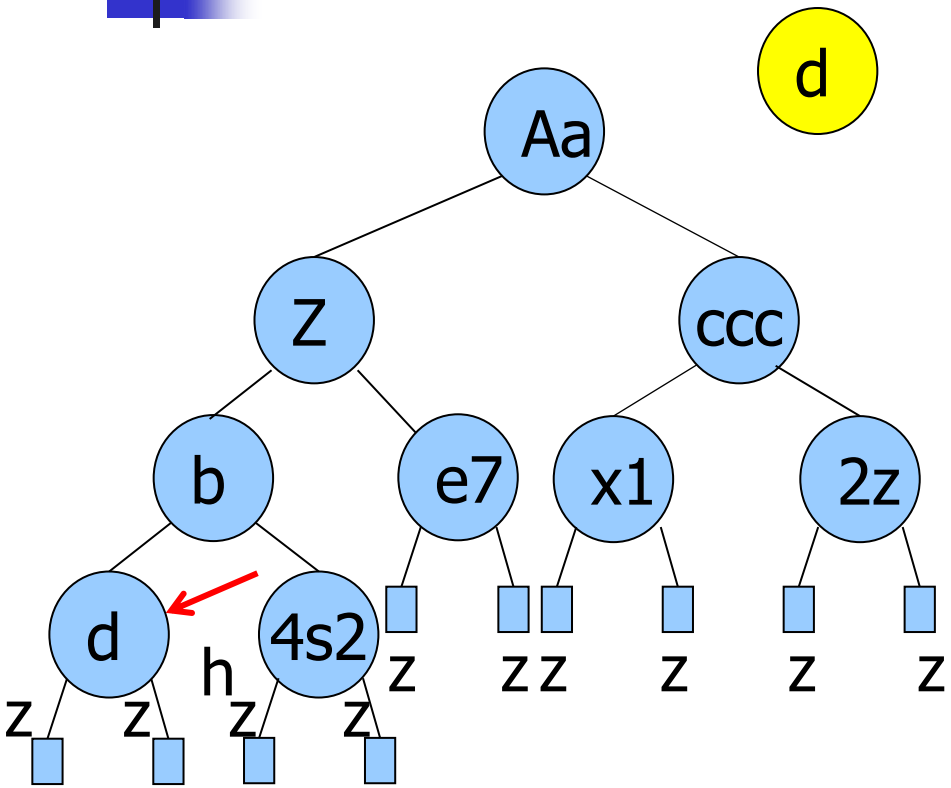
Post-ordine

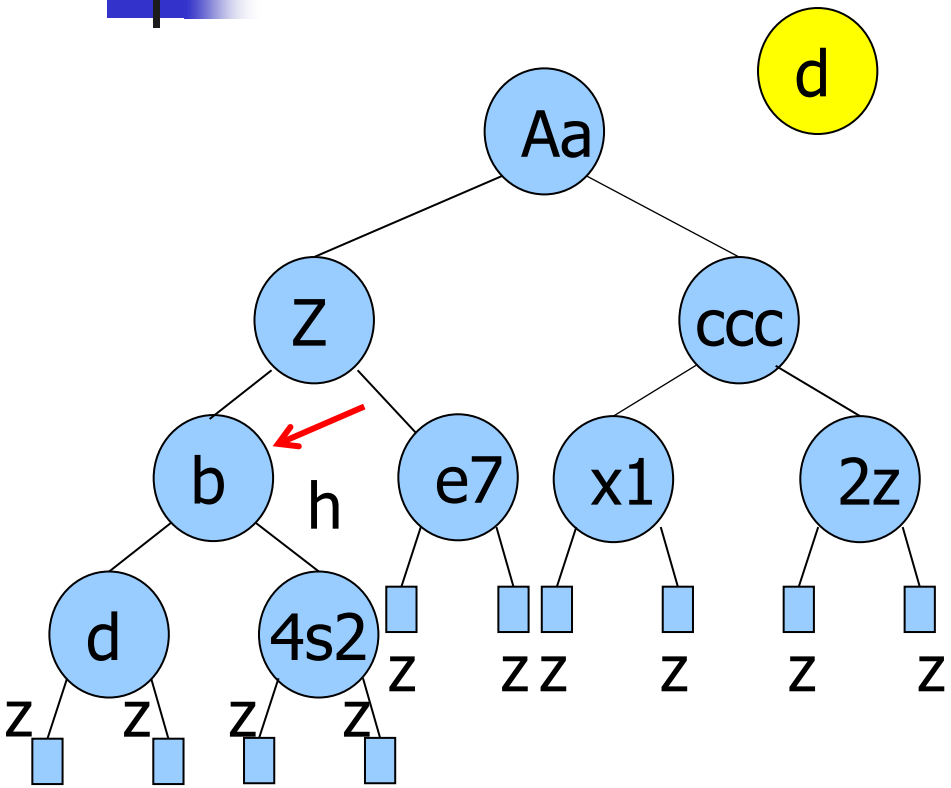


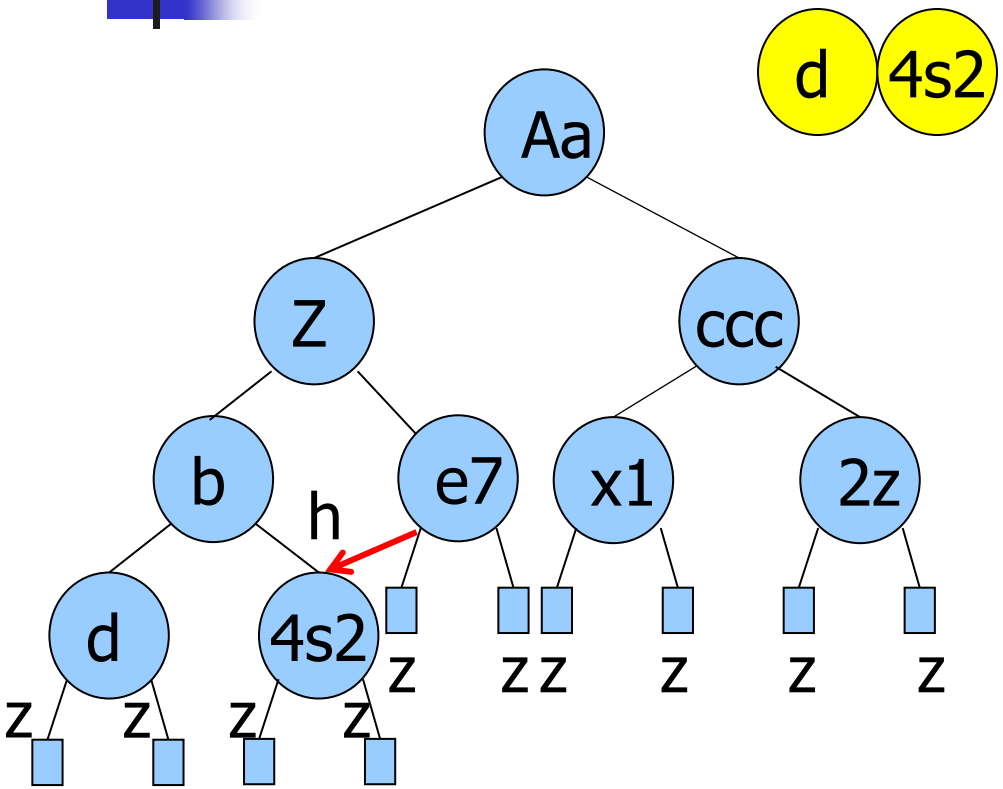

```
void postorderR(link h,link z){
  if (h == z)
    return;
  postorderR(h->l, z);
  postorderR(h->r, z);
  printf("%s ",h->name);
}
```

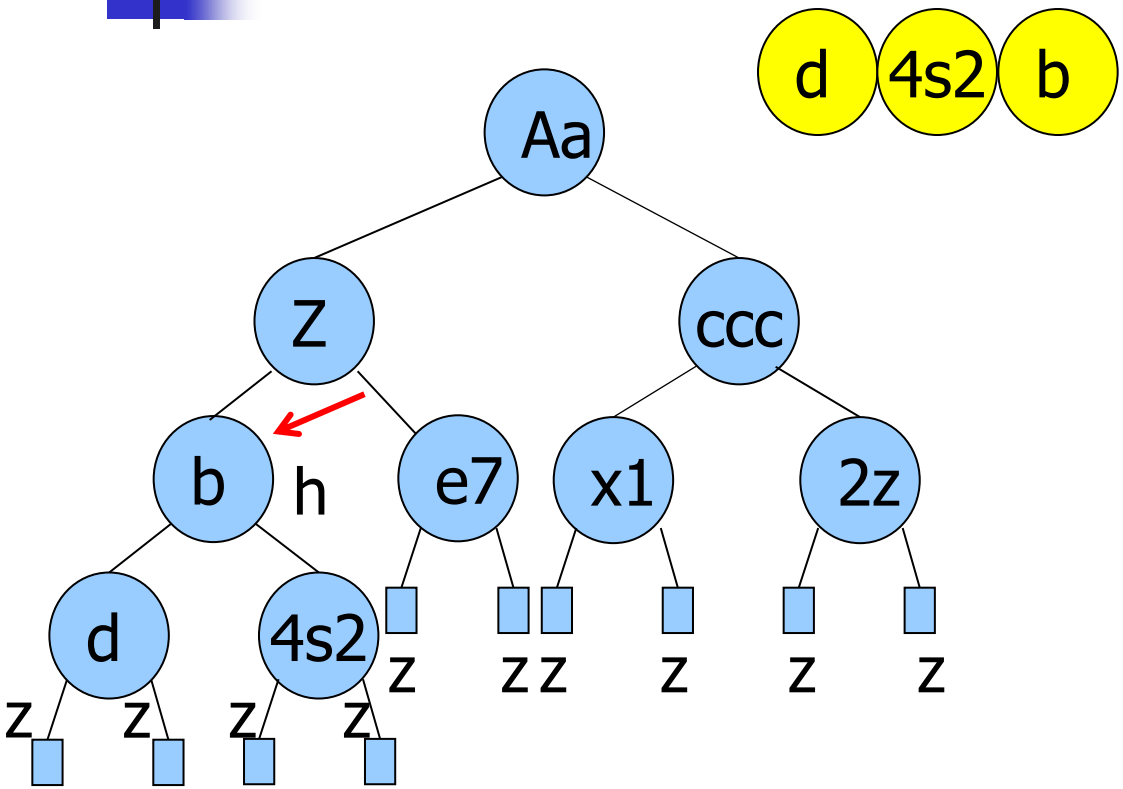

Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

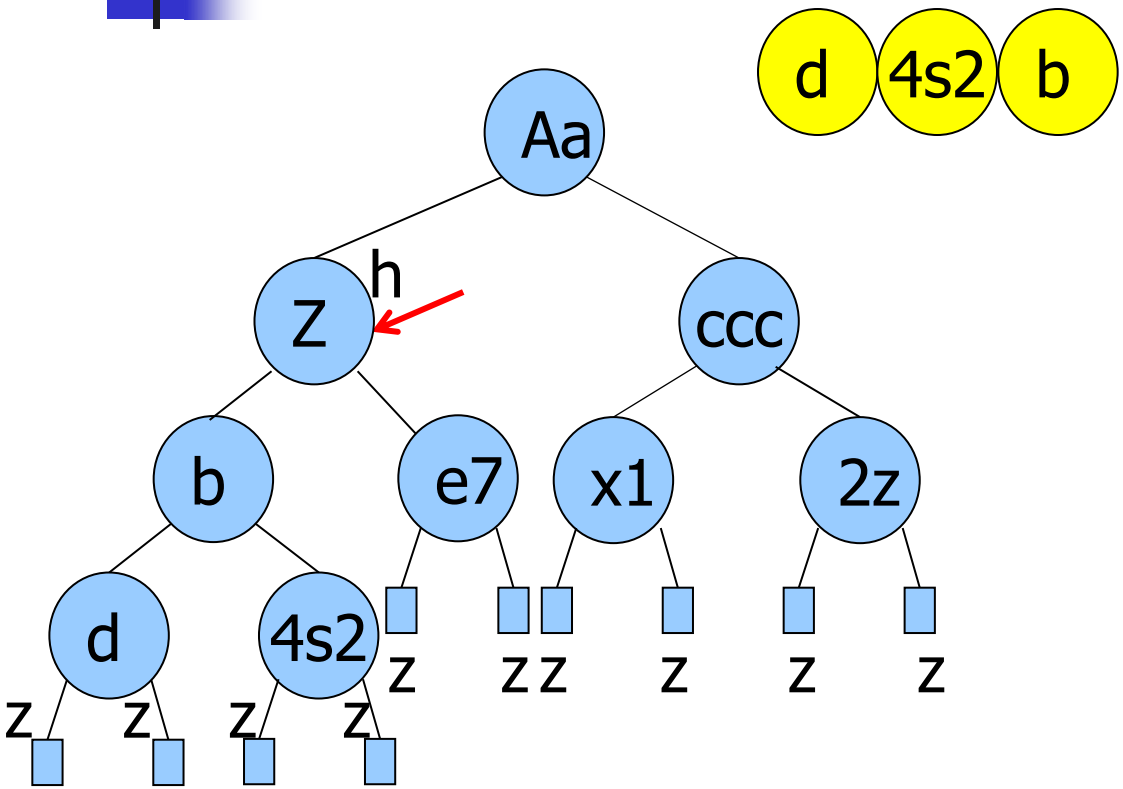

Aa
Z
ccc
b
h
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

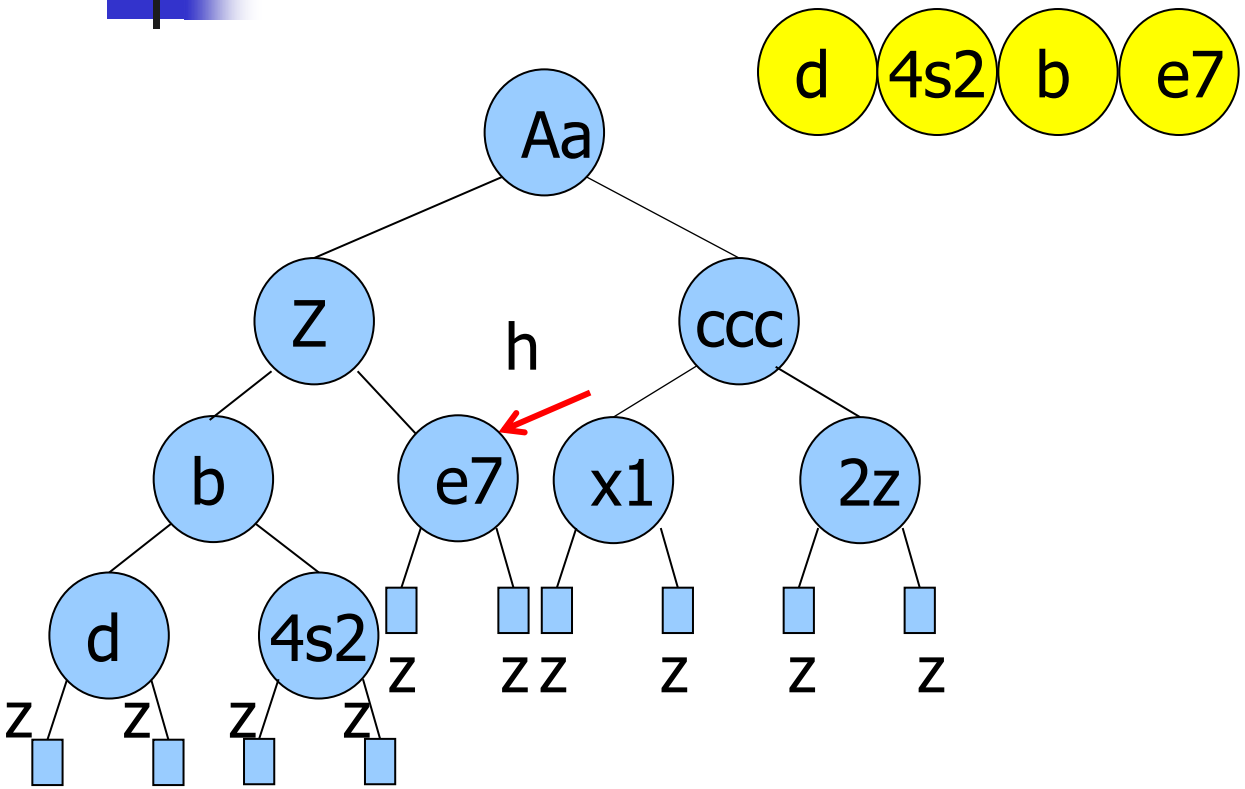

d
Aa
Z
ccc
b
e7
x1
2z
d
4s2
h
z
z
z
z
z
z
z
z
z
z

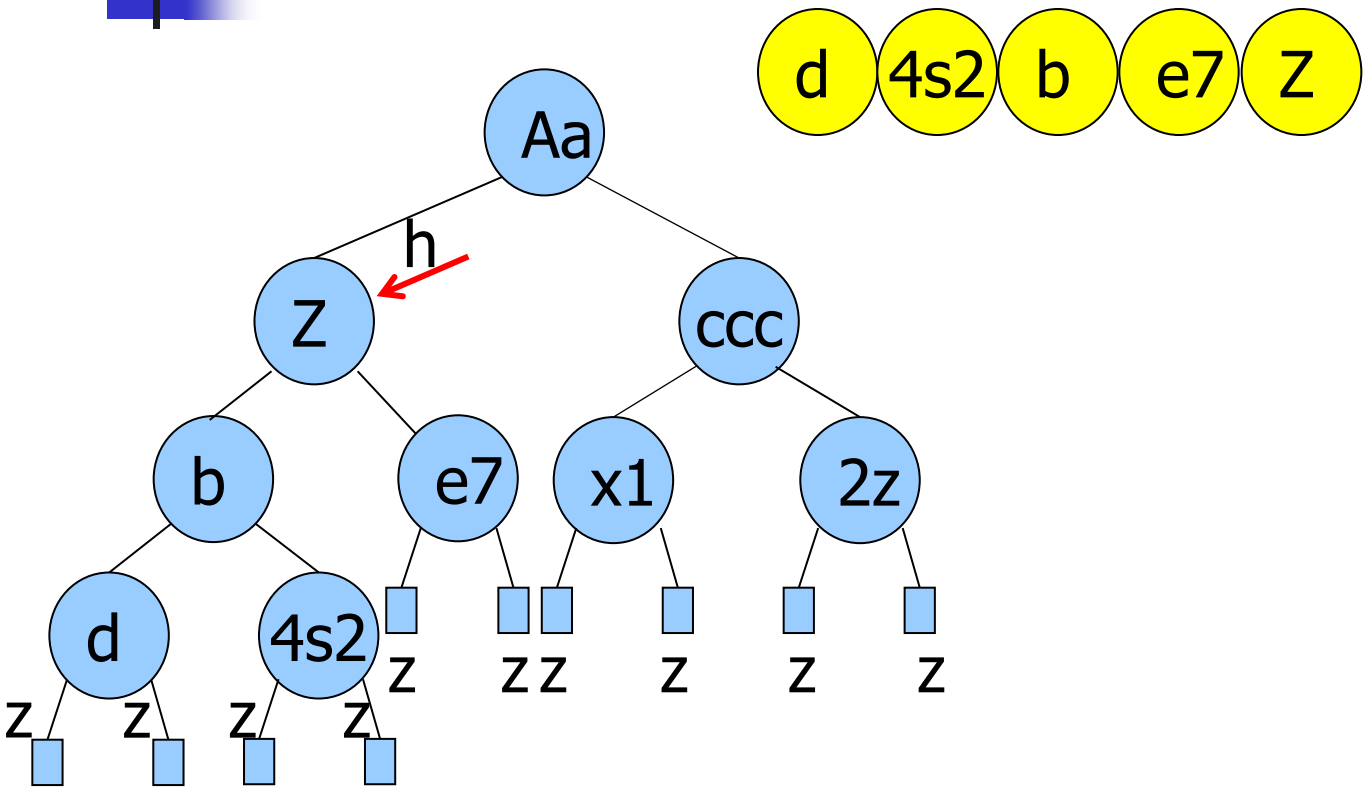

d
Aa
Z
ccc
b
h
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

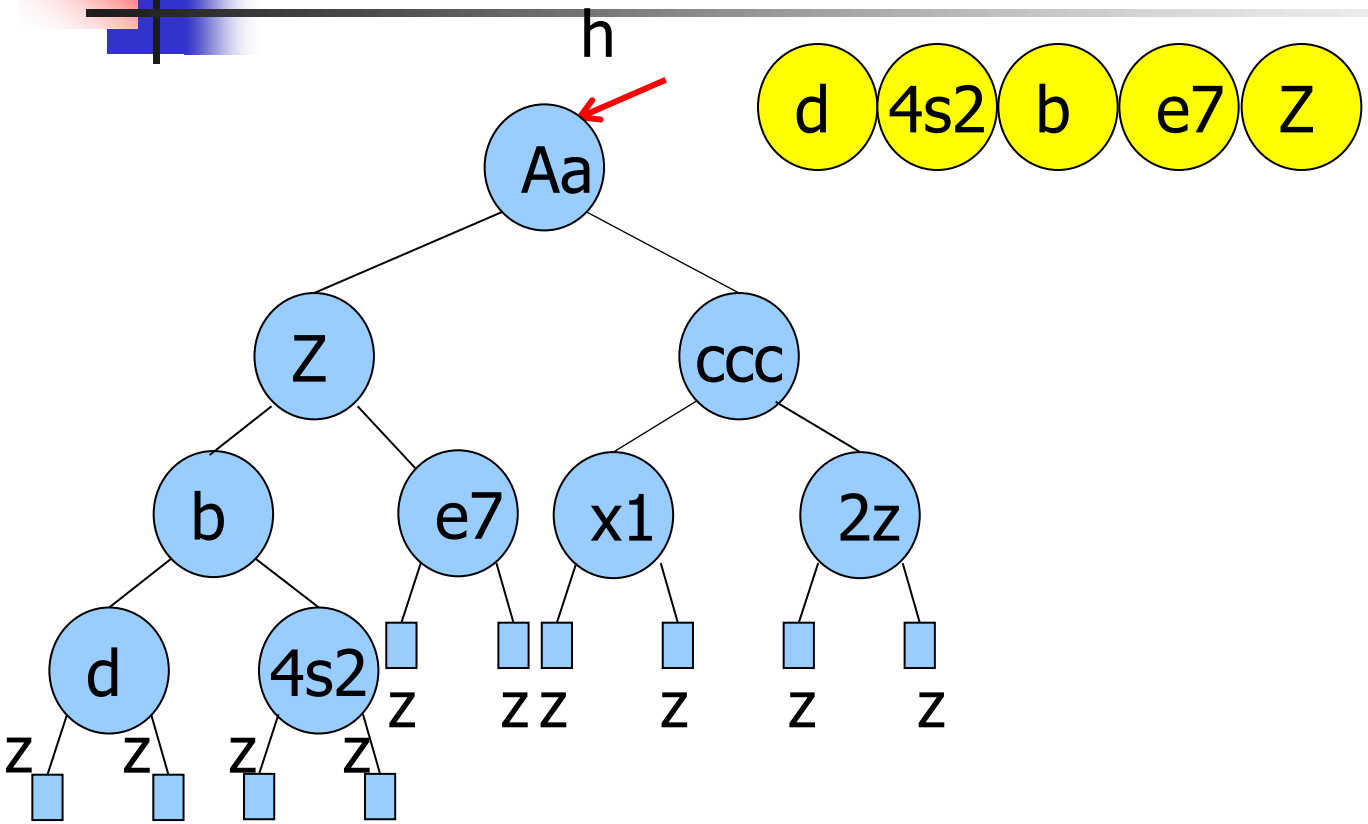

d
4s2
Aa
Z
ccc
b
h
e7
x1
2z
d
4s2
z
z
z
z
z
z
z
z
z
z

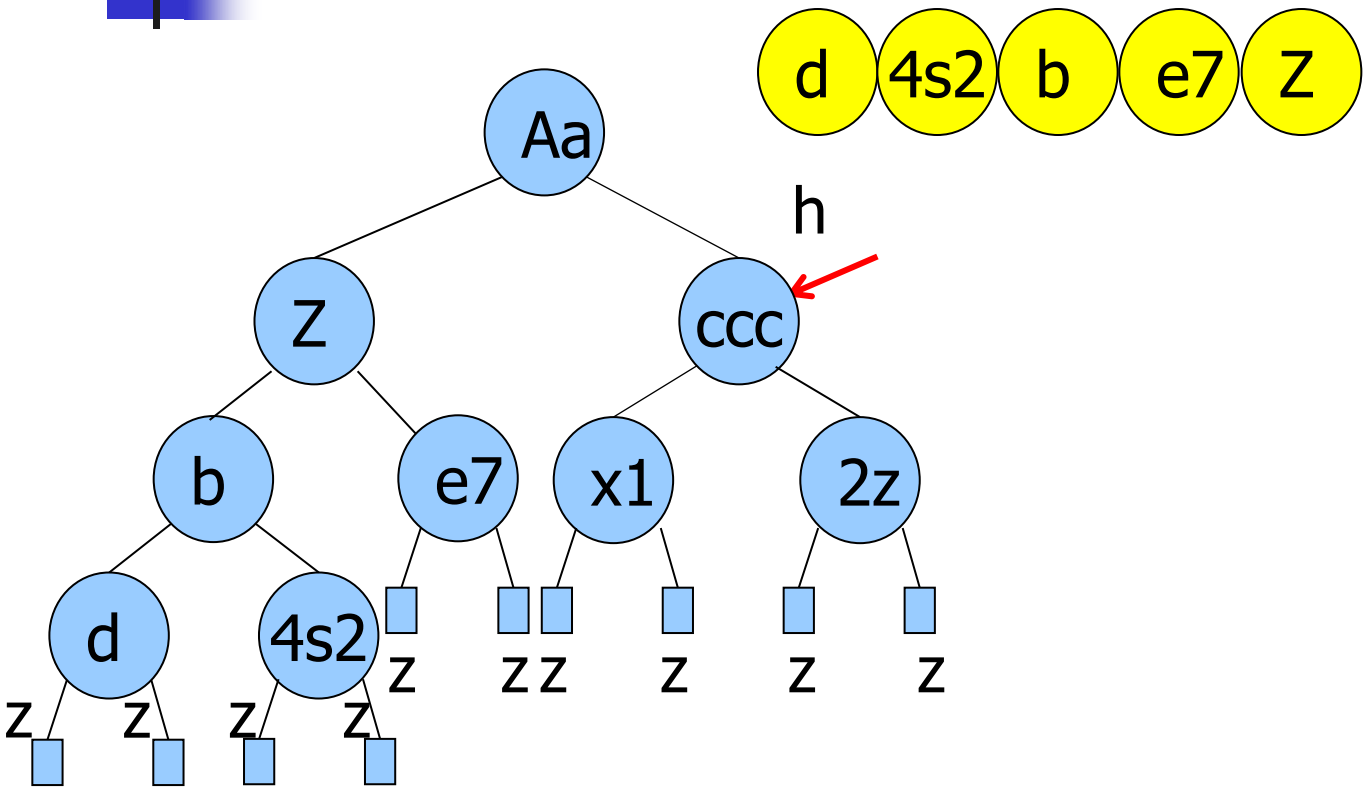

d
4s2
b
Aa
Z
ccc
b
h
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

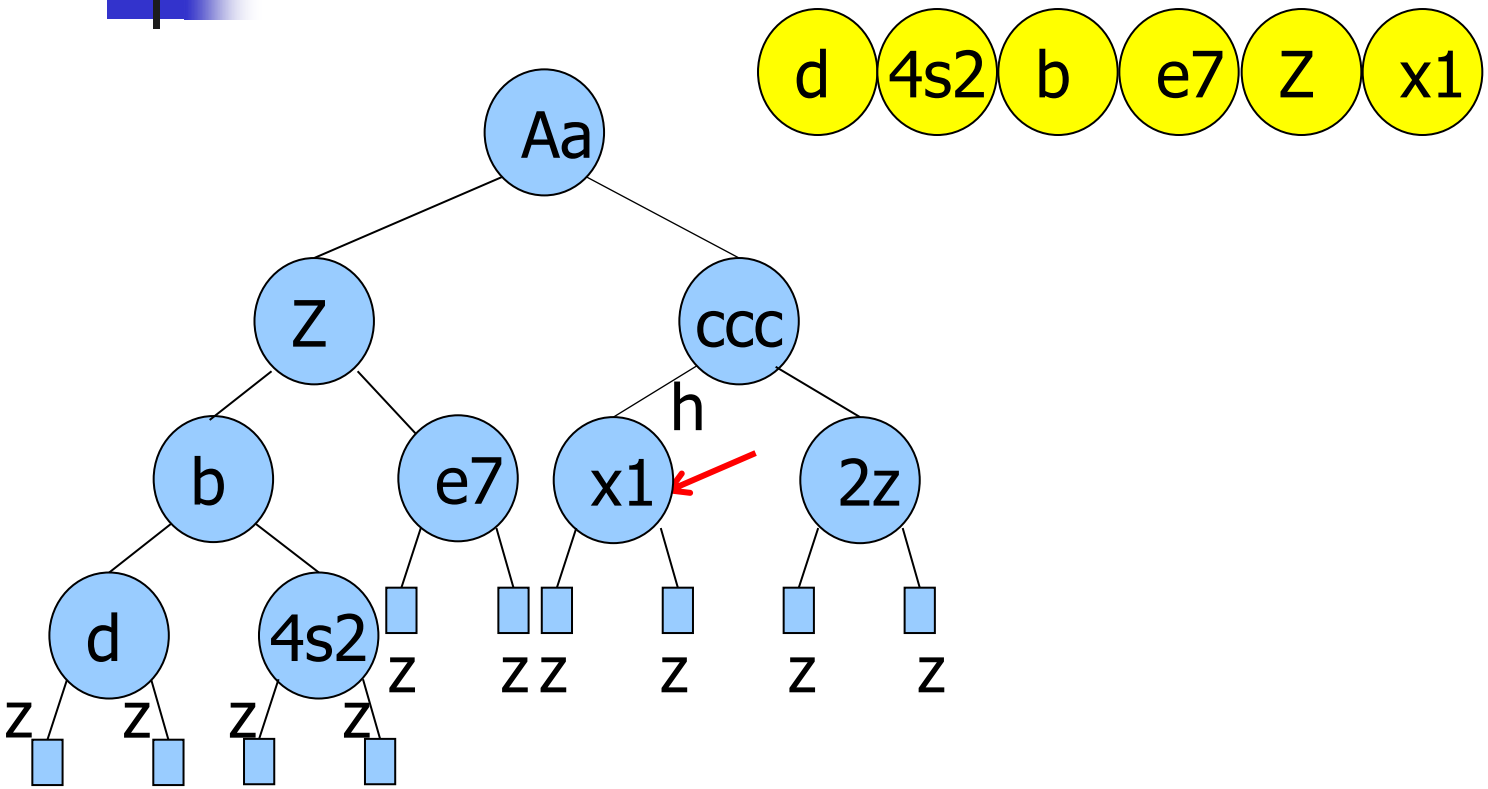

d
4s2
b
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

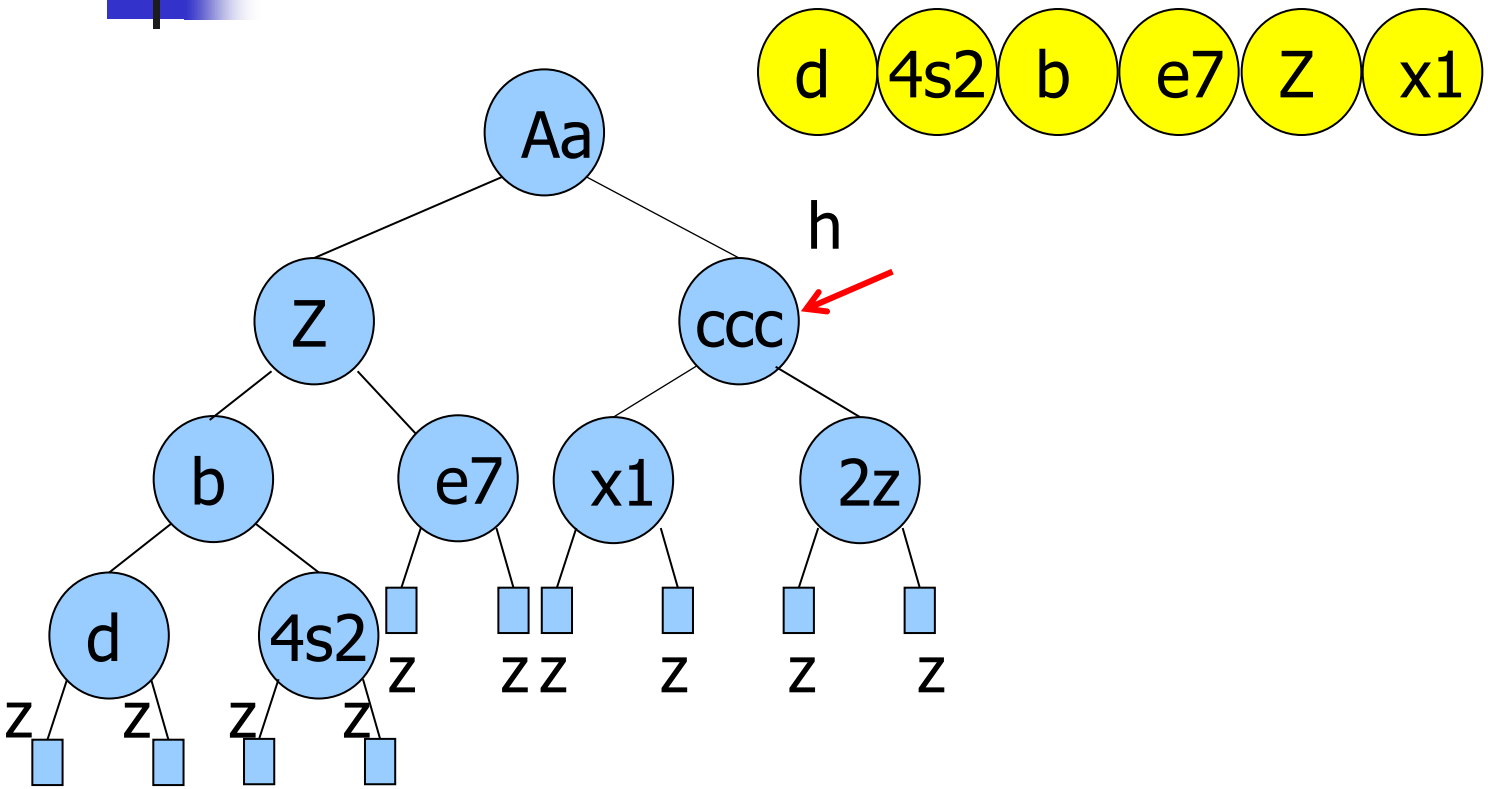

d
4s2
b
e7
Aa
Z
h
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

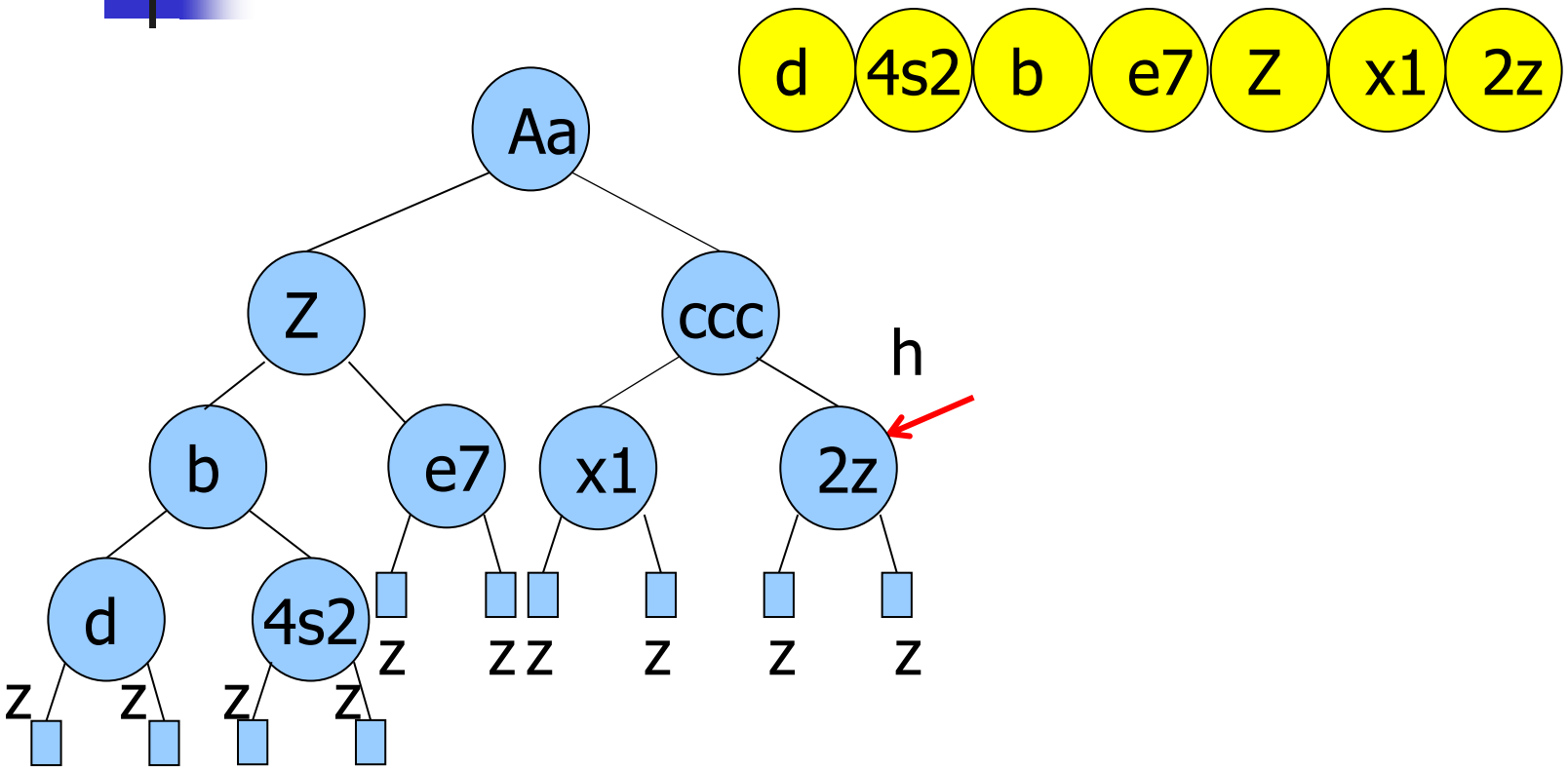

d
4s2
b
e7
Z
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

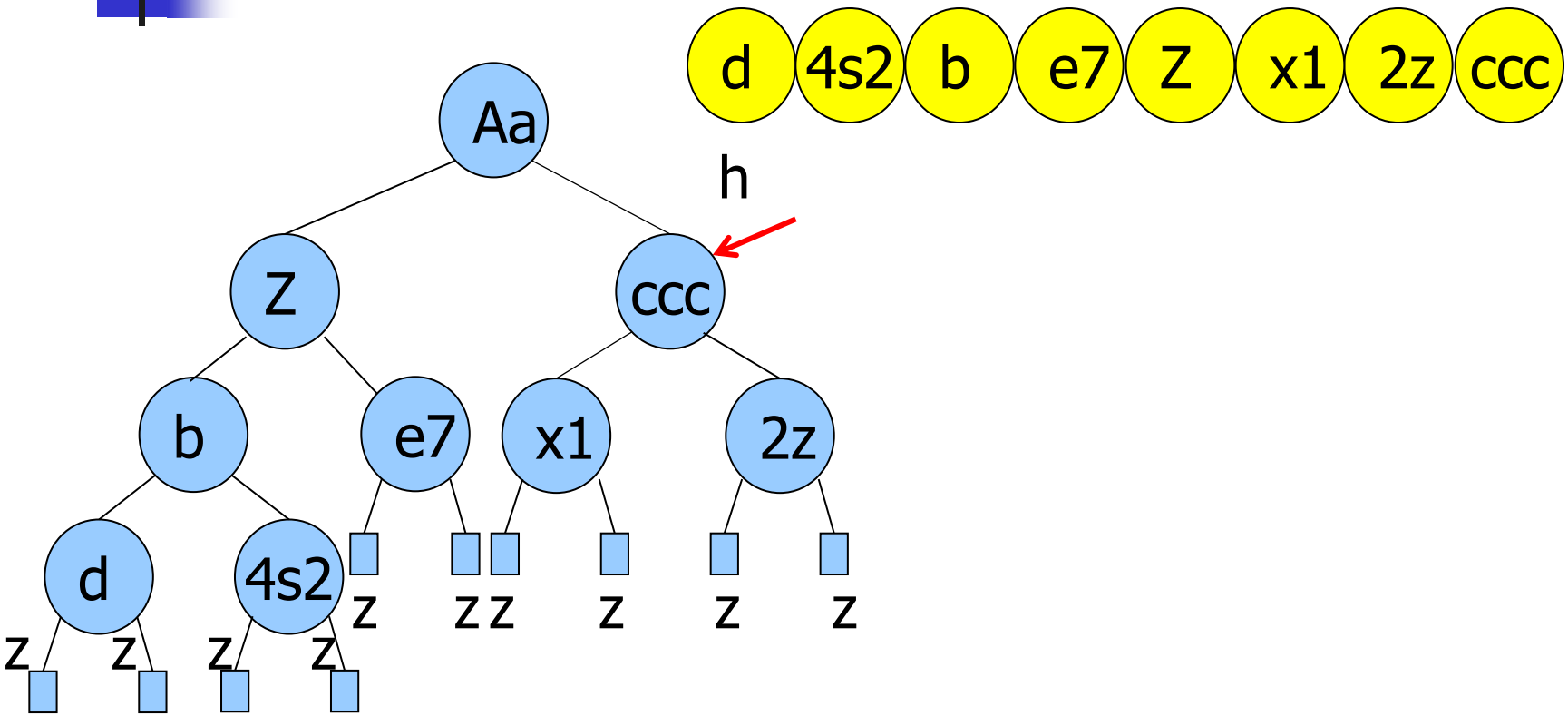

h
d
4s2
b
e7
Z
Aa
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z

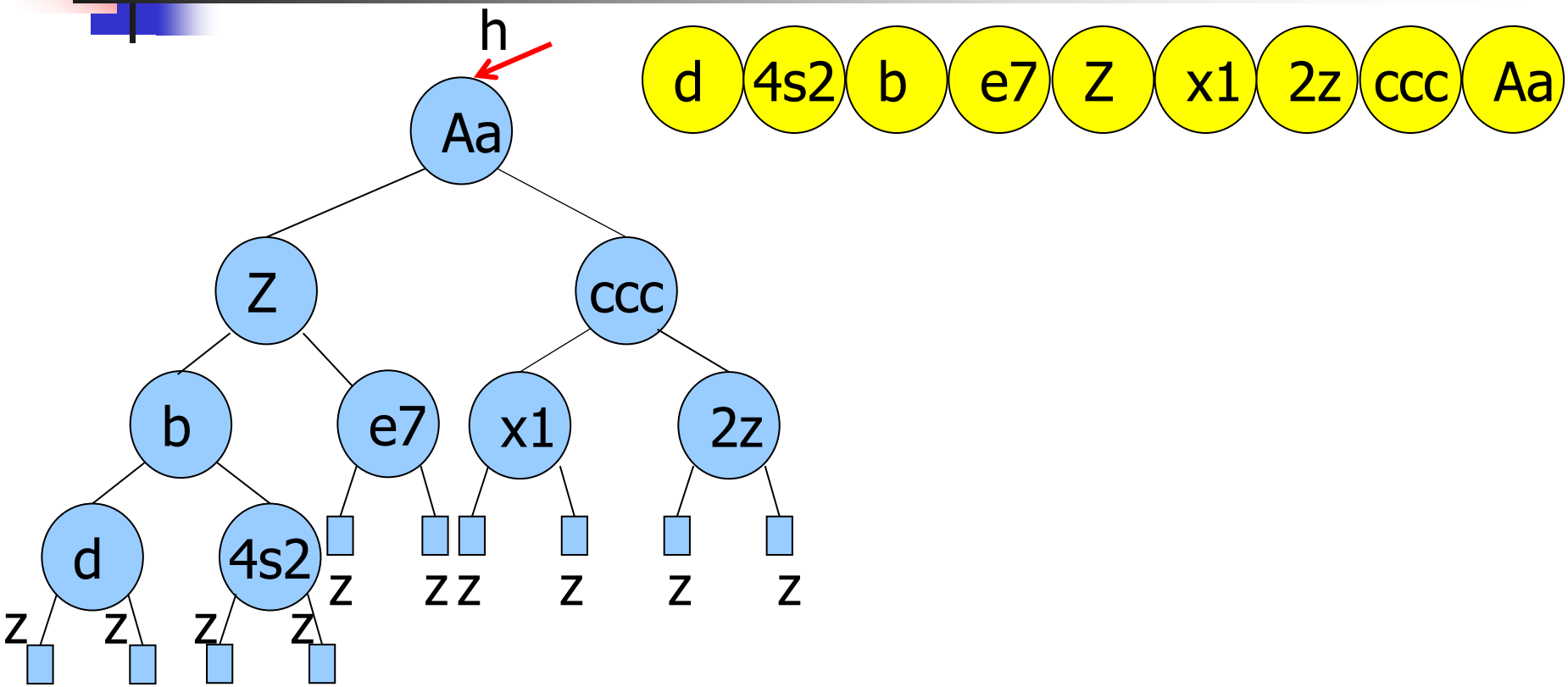

d
4s2
b
e7
Z
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z


d
4s2
b
e7
Z
x1
Aa
Z
ccc
h
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z


d
4s2
b
e7
Z
x1
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z


d
4s2
b
e7
Z
x1
2z
Aa
Z
ccc
h
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z z z z


d
4s2
b
e7
Z
x1
2z
ccc
Aa
h
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z
z
z
z


h
Aa
Z
ccc
b
e7
x1
2z
d
4s2
z
z z
z
z
z
z z z z
d 4s2 b e7 Z x1 2z ccc Aa

Analisi di complessità

divide and conquer
a = 2 b = 2

Ipotesi 1: albero completo:

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- a = 2, b = 2 (sottoproblemi di dimensione n-1, approssimata conservativamente a n)

Equazione alla ricorrenze:

$T(n) = 1 + 2T(n/2)$ $\quad n > 1$

$T(1) = 1$

$T(n) = O(n)$

decrease and conquer
$a = 1$ $k_i = 1$

Ipotesi 2: albero totalmente sbilanciato (degenerato in una lista) :

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$

Equazione alla ricorrenze:

$T(n) = 1 + T(n-1)$ $\quad n > 1$

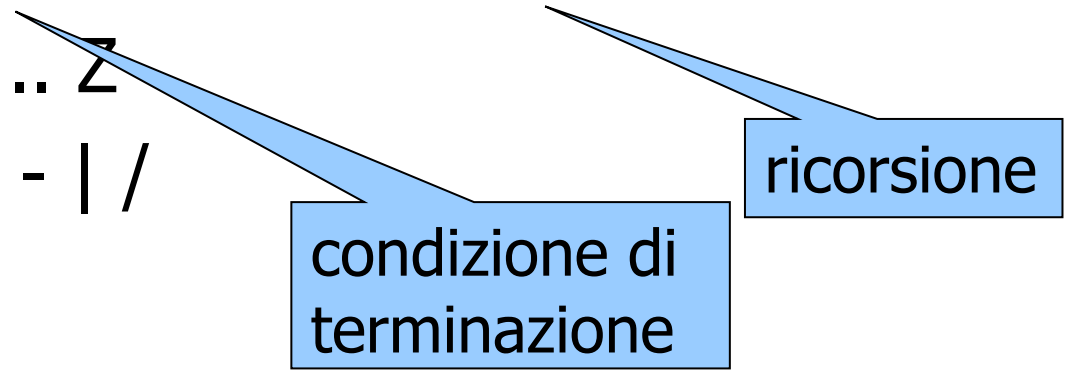
$T(1) = 1$

$T(n) = O(n)$

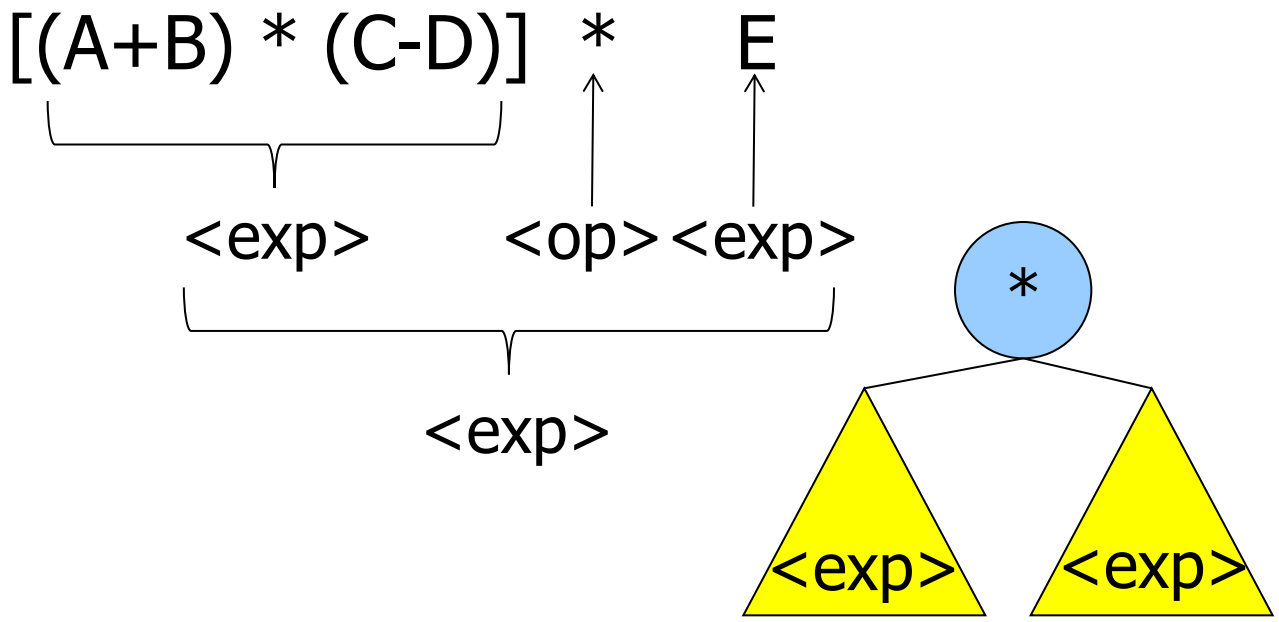
Alberi binari ed espressioni

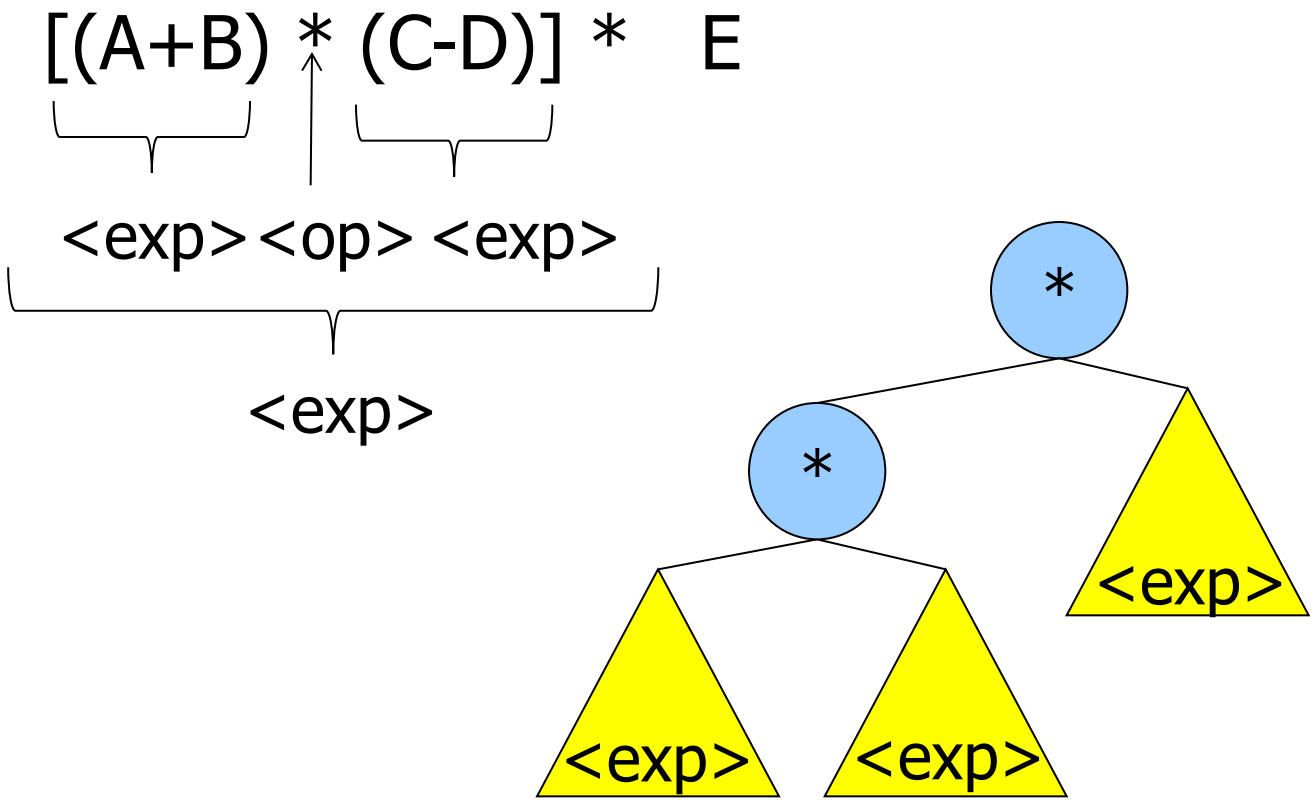
Data un'espressione algebrica in forma infissa (con parentesi per cambiare le precedenze tra operatori), ricostruirne l'albero binario in base alla grammatica semplificata:

- <exp> = <operand> | <exp> <op> <exp>
- <operand> = A .. Z
- <op> = + | * | - | /



Dal parsing dell'espressione si ottiene

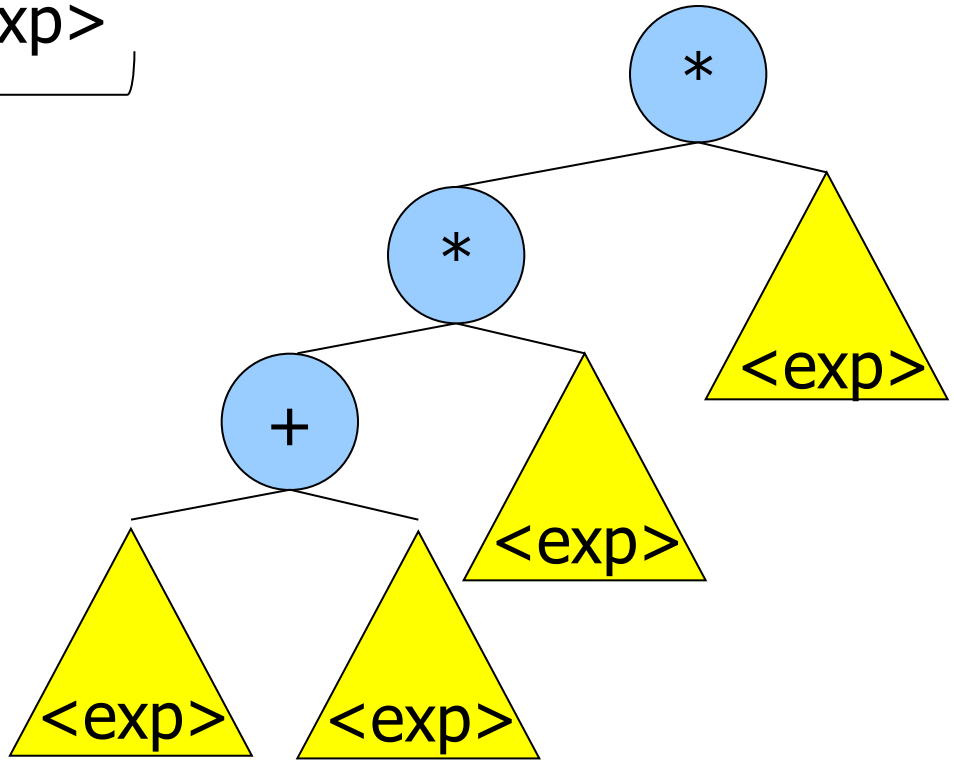



[(A+B) * (C-D)] * E
<exp> <op> <exp>
<exp>
*
*
<exp>
<exp>
<exp>

[(A+B) * (C-D)] * E

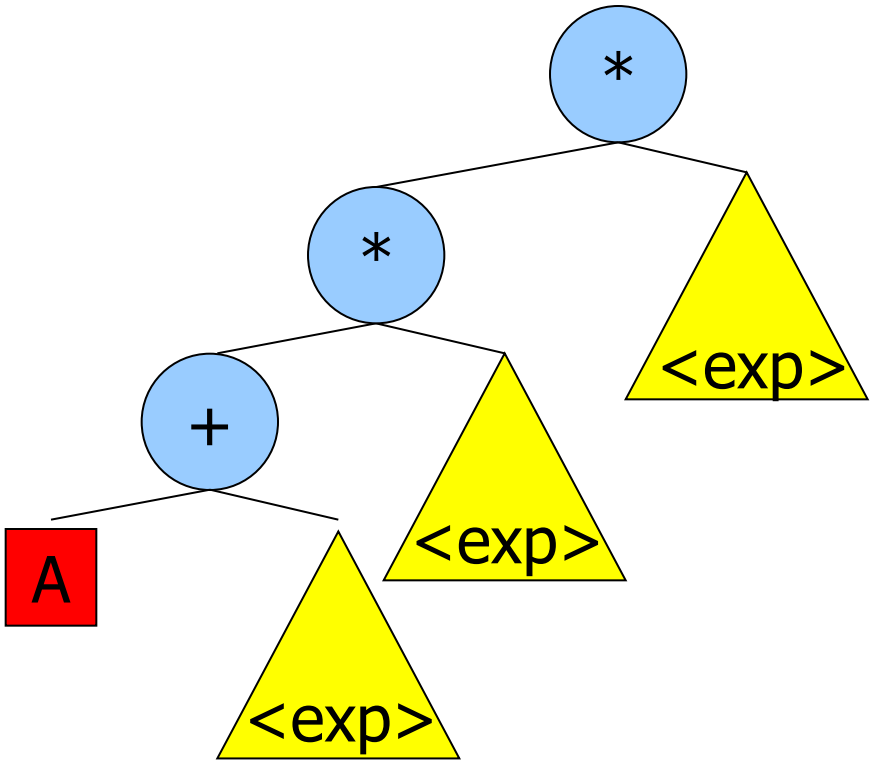
<exp><op> <exp>

<exp>



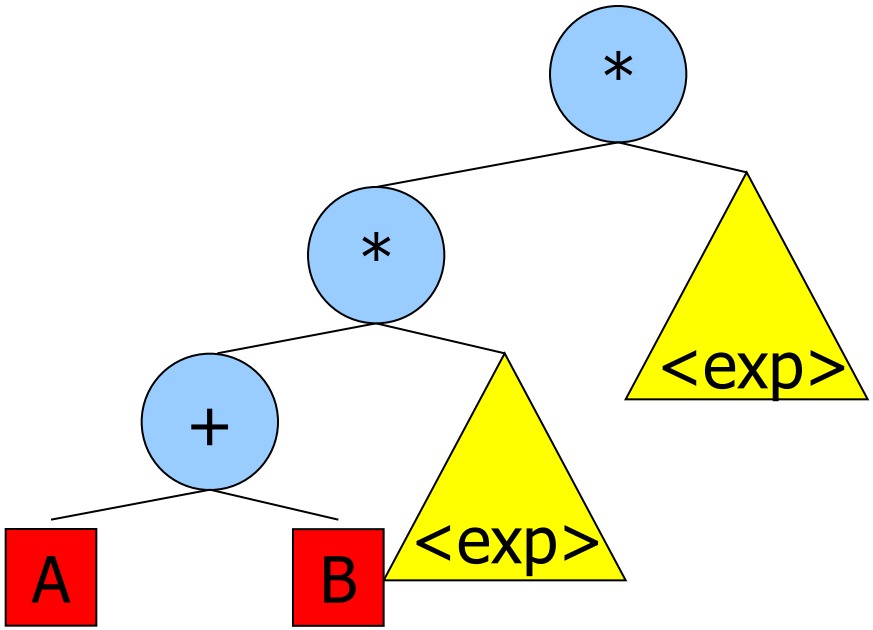
[(A+B) * (C-D)] * E

<operand>



[(A+B) * (C-D)] * E

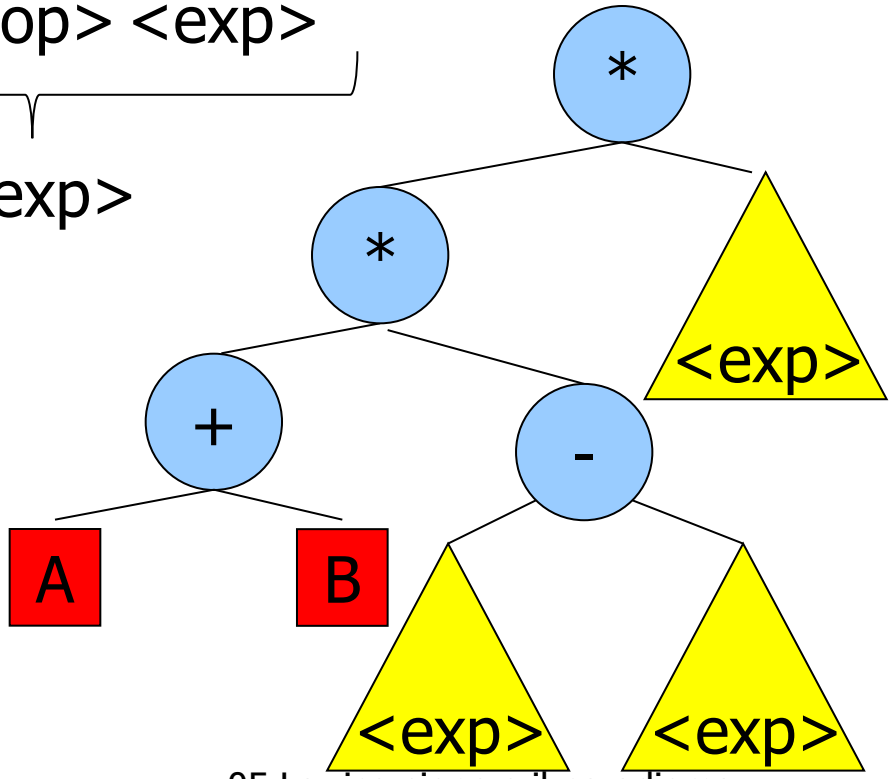
<operand>



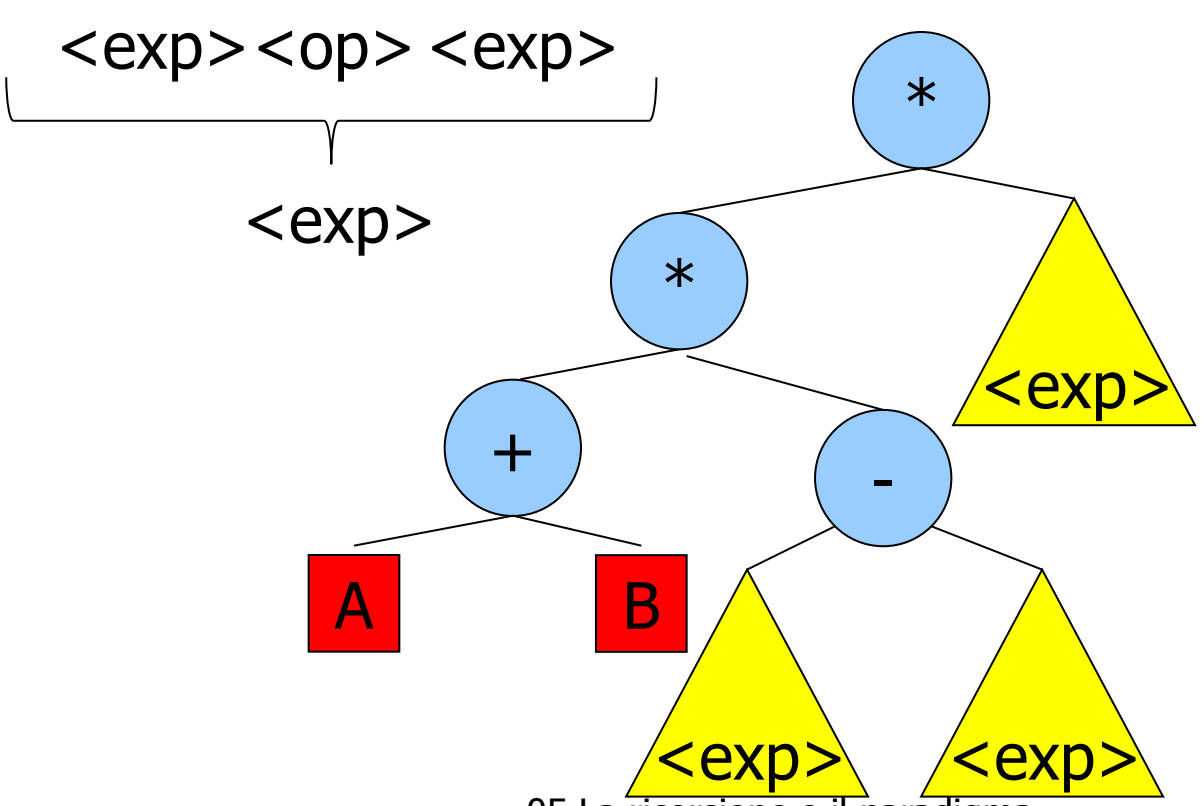
[(A+B) * (C-D)] * E

<exp><op> <exp>

<exp>

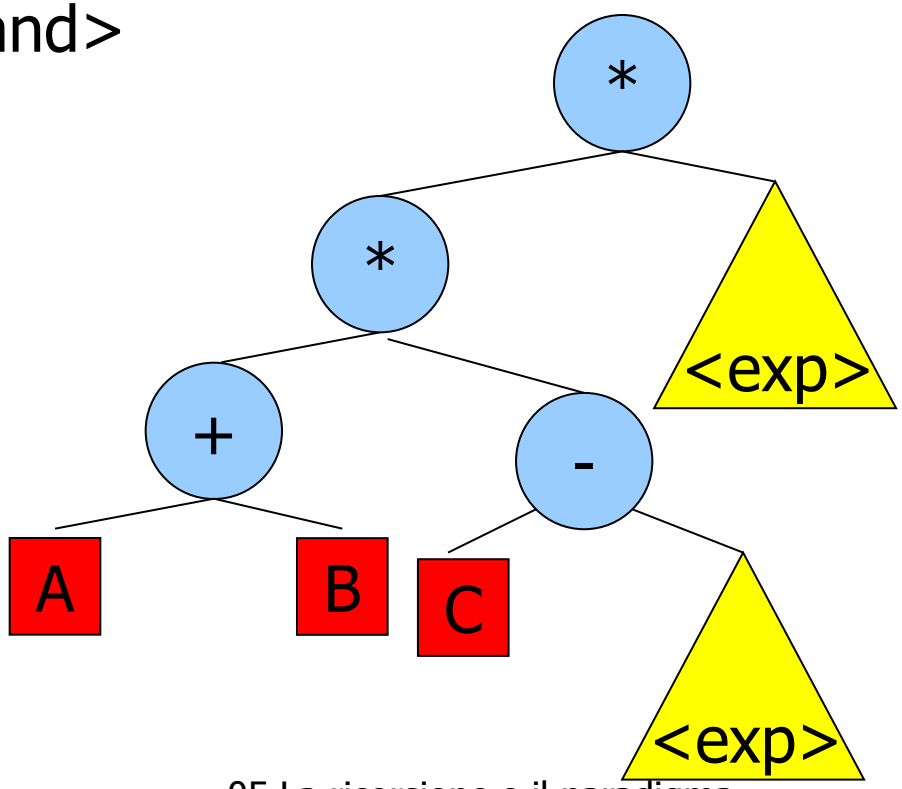


[(A+B) * (C-D)] * E



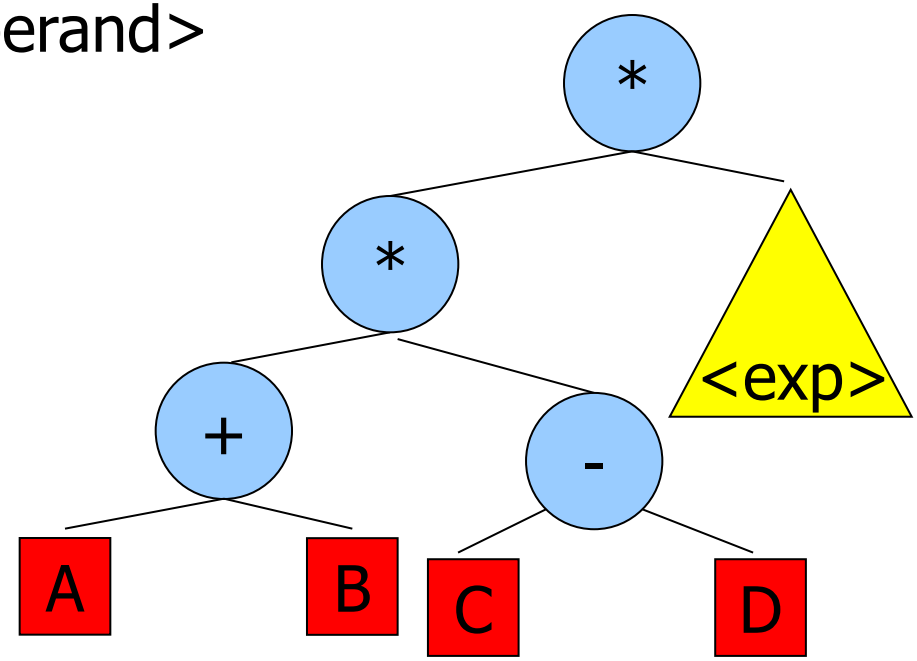
[(A+B) * (C-D)] * E

<operand>



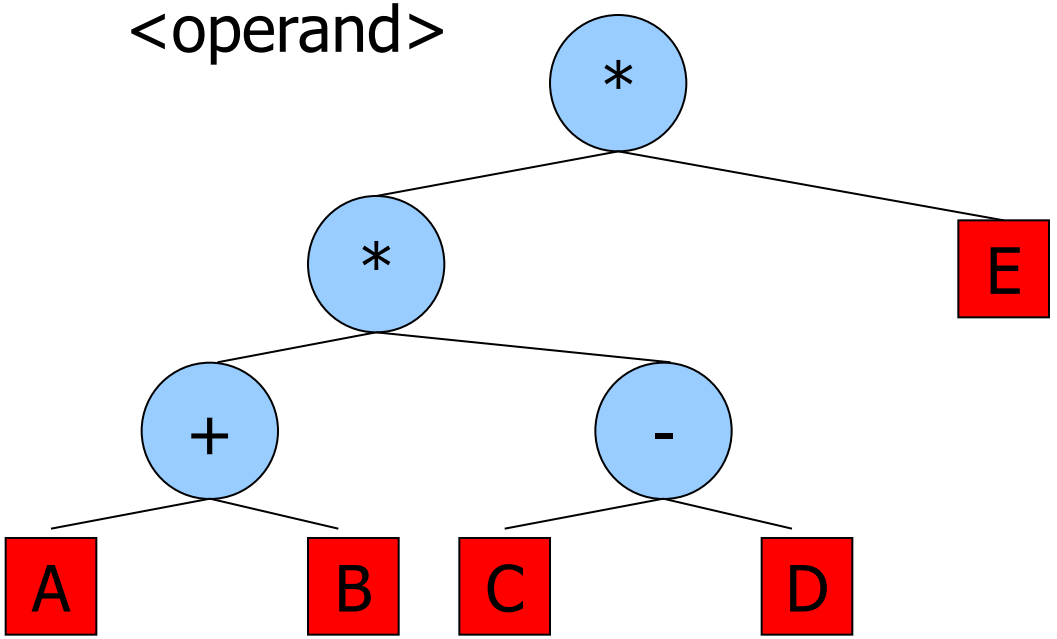
[(A+B) * (C-D)] * E

<operand>

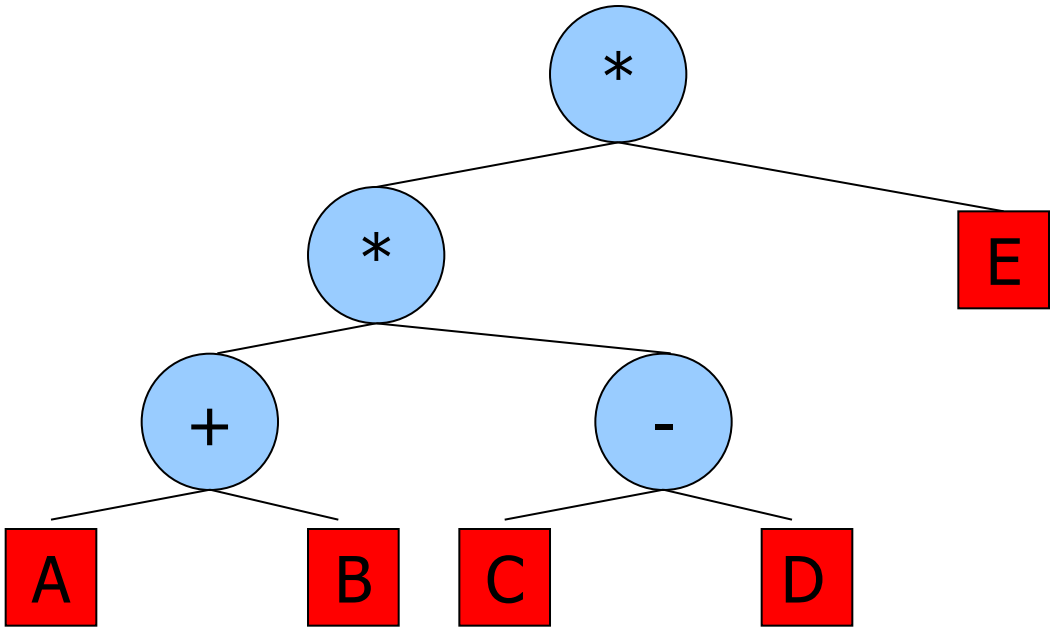


[(A+B) * (C-D)] * E

<operand>



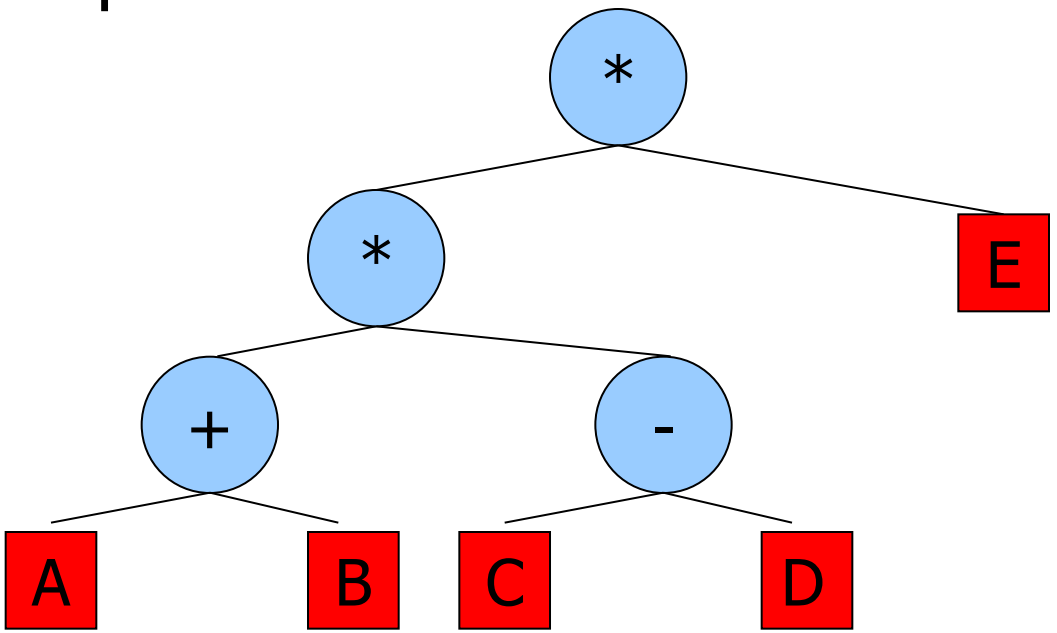
L'attraversamento in post-ordine dell'albero dà la forma postfissa (Notazione Polacca Inversa o Reverse Polish Notation) dell'espressione



A B + C D - * E *

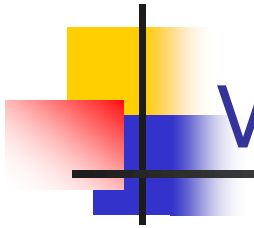
parentesi non più necessarie

L’attraversamento in pre-ordine dell’albero dà la forma prefissa (Notazione Polacca), poco usata in pratica, dell’espressione



* * + A B – C D E

parentesi non più necessarie



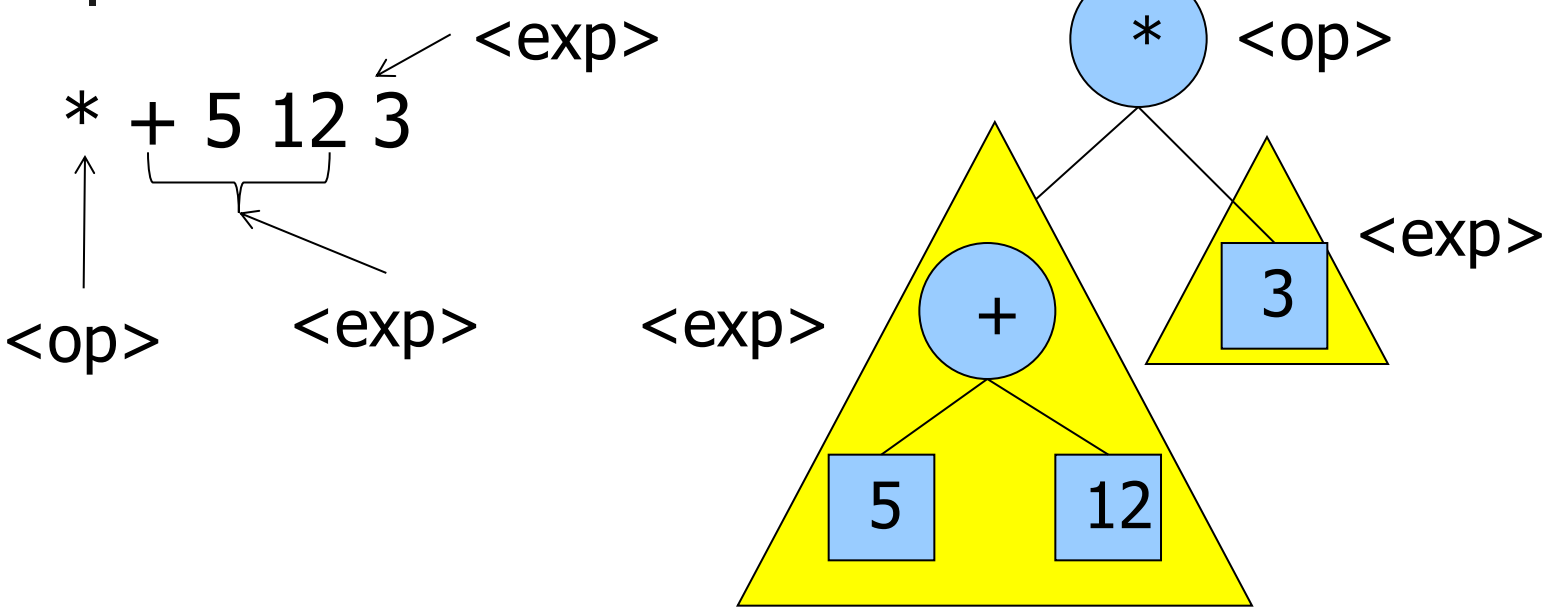
Valutazione di espressione in forma prefissa

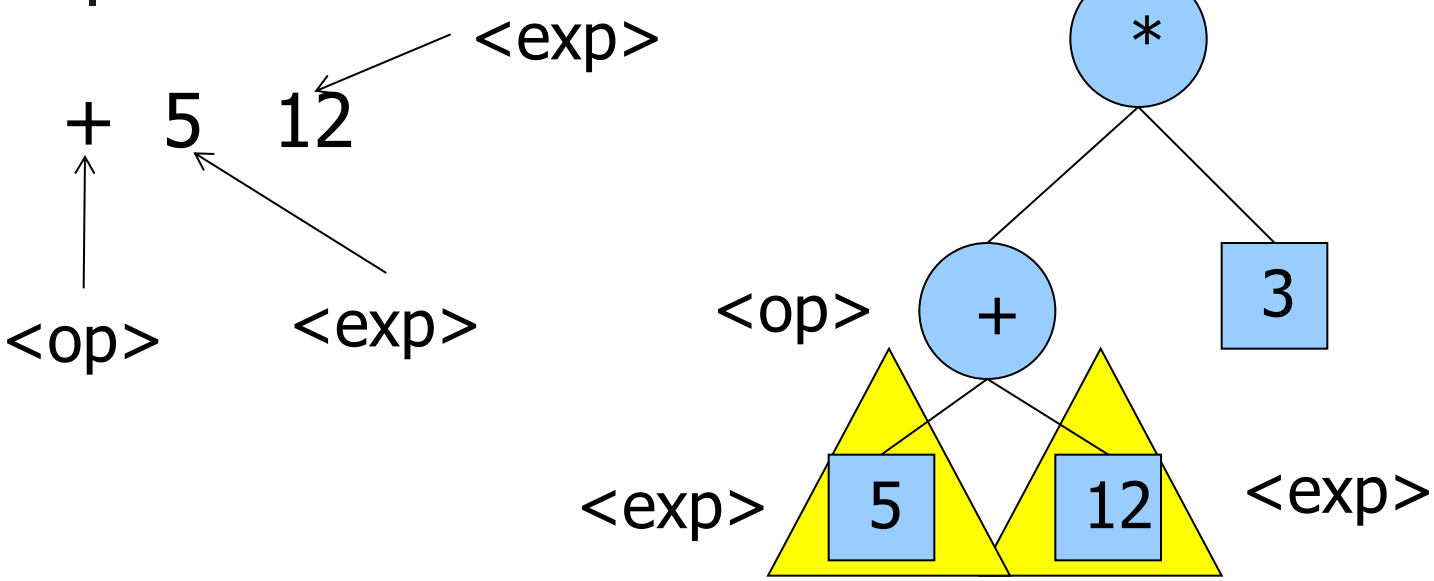
Grammatica per espressioni in forma prefissa (Notazione Polacca) semplificate (solo operatori + e *, operandi interi positivi)

- <exp> = <operand> | <op> <exp> <exp>
- <operand> = digit | digit catenate operand
- <digit> = 0..9
- <op> = + | *

Esempio:

- in forma infissa (5 + 12) * 3
- in forma prefissa * + 5 12 3

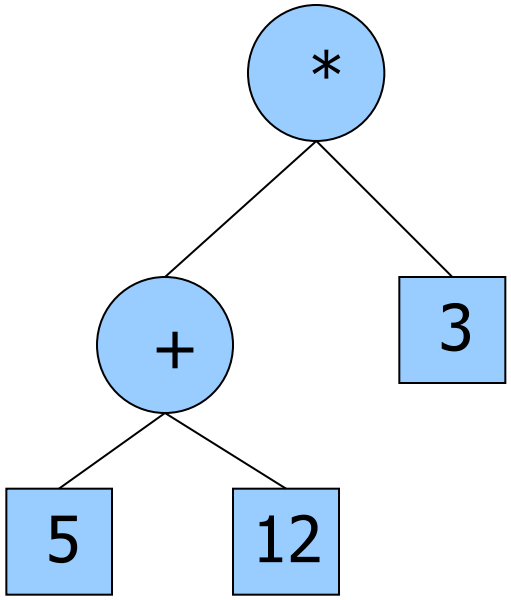

<exp>
* + 5 12 3
<op>
<exp>
*
<op>
<exp>
<exp>
+
3
5
12


<exp>
+ 5 12
<op>
<exp>
*
<op>
+
3
<exp>
5
12
<exp>

Espressione prefissa memorizzata in vettore a di caratteri (spazi per separare):

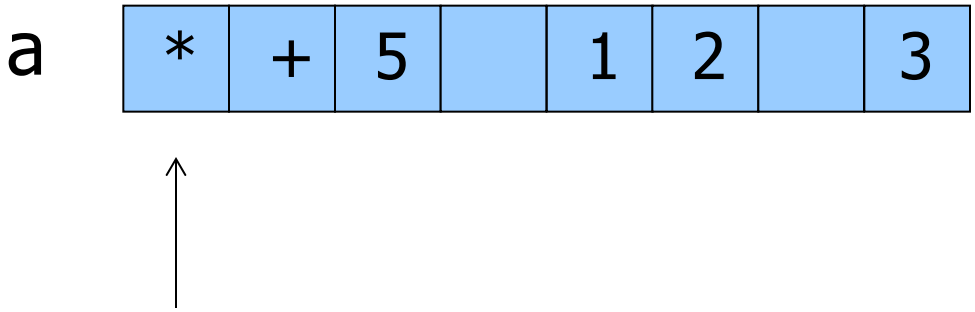
a

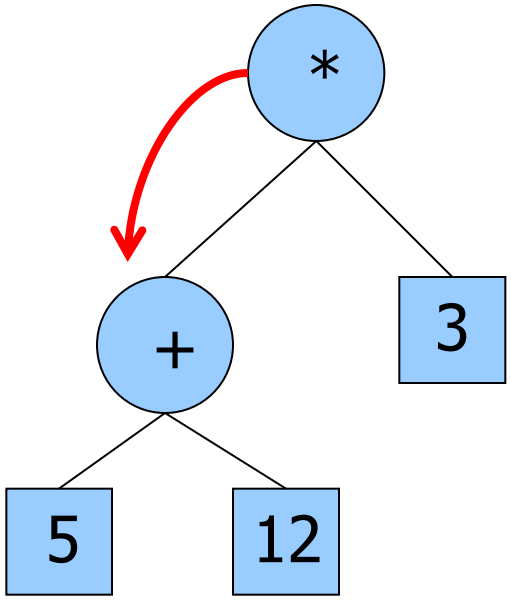
*	+	5		1	2		3

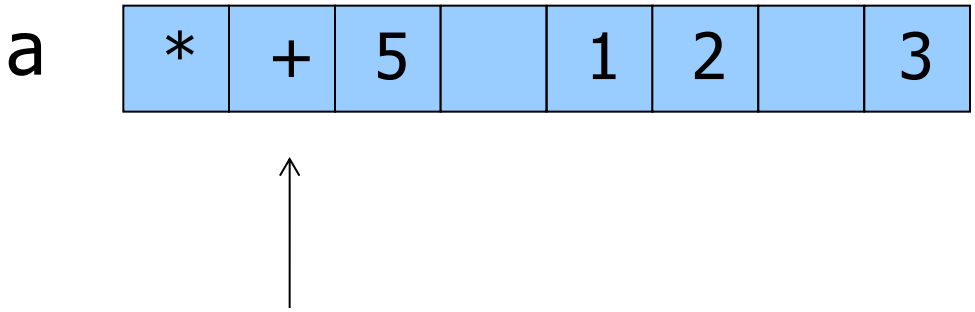


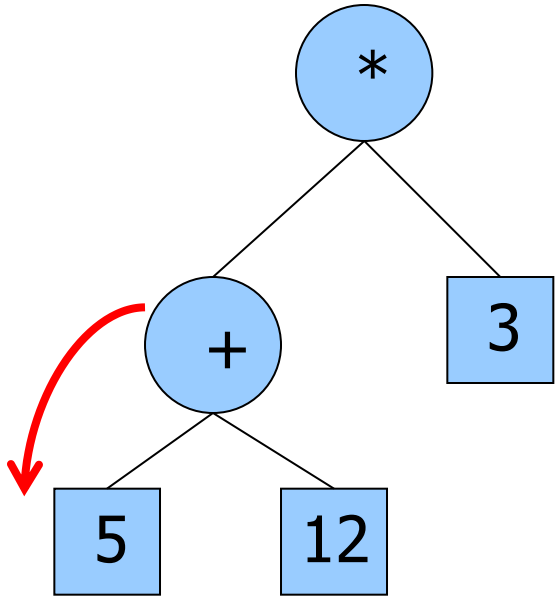
Valutazione: scansione

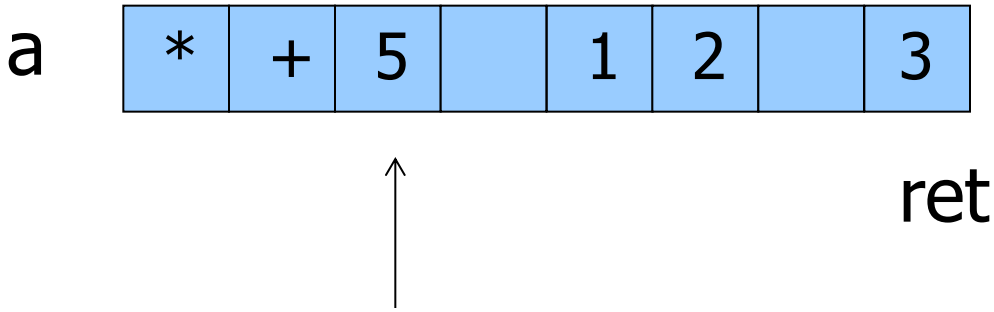
- se a[i] è un operatore op, ritorna eval() op eval()
- terminazione: se a[i] è una cifra, calcola il valore dell'intero formato da cifre fino allo spazio, ritorna l'intero


a
*
+
5
1
2
3

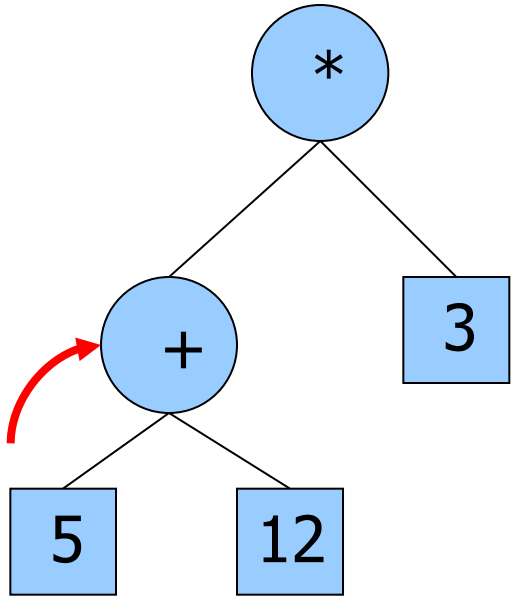

*
+
3
5
12

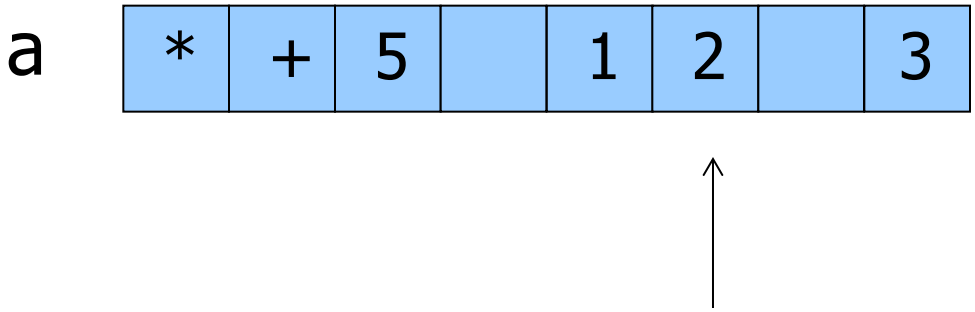

a
*
+
5
1
2
3

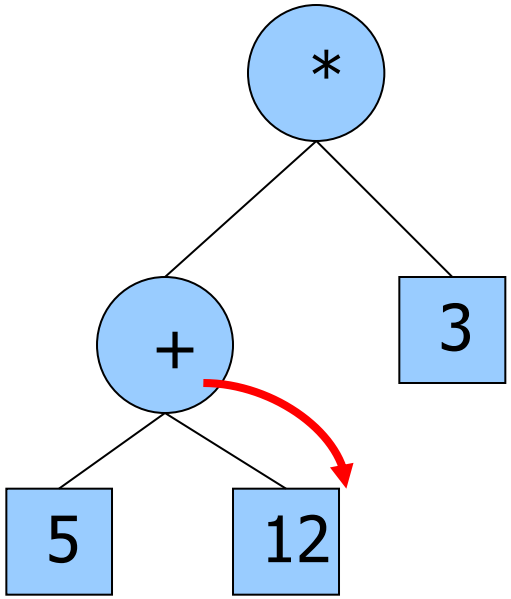

*
+
3
5
12



return 5

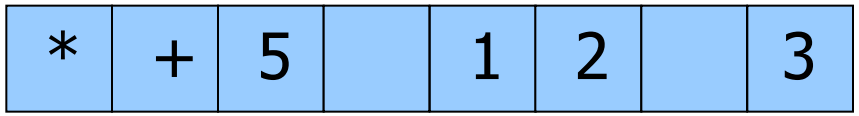


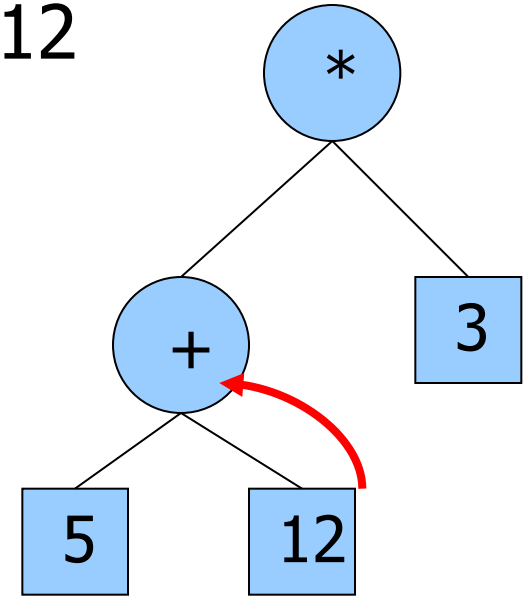

a
*
+
5
1
2
3

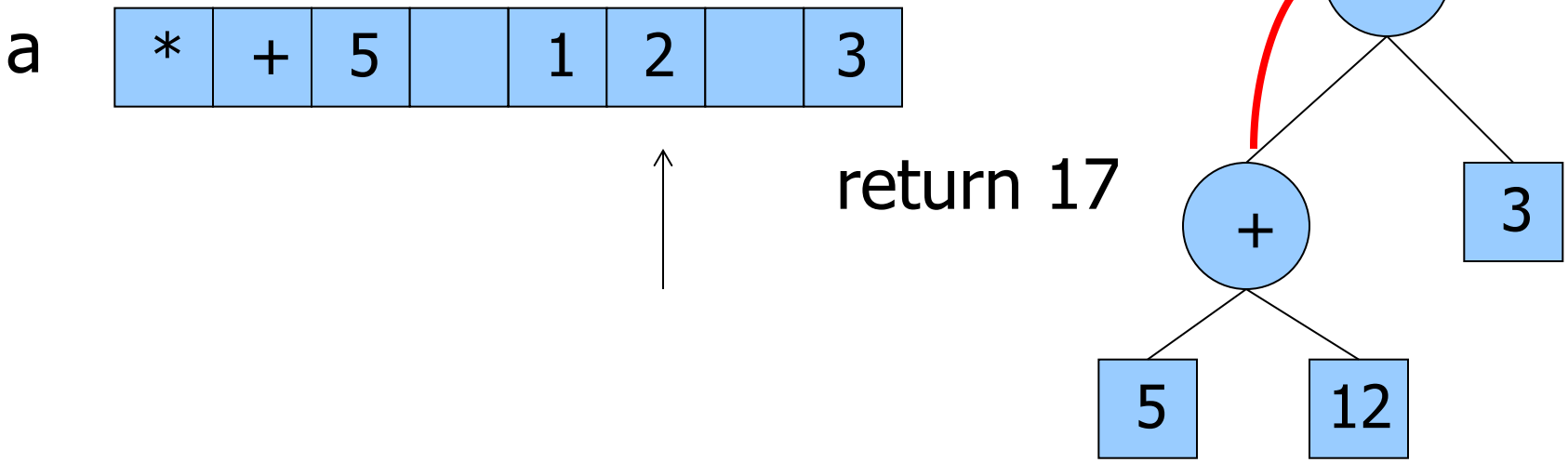

*
+
3
5
12

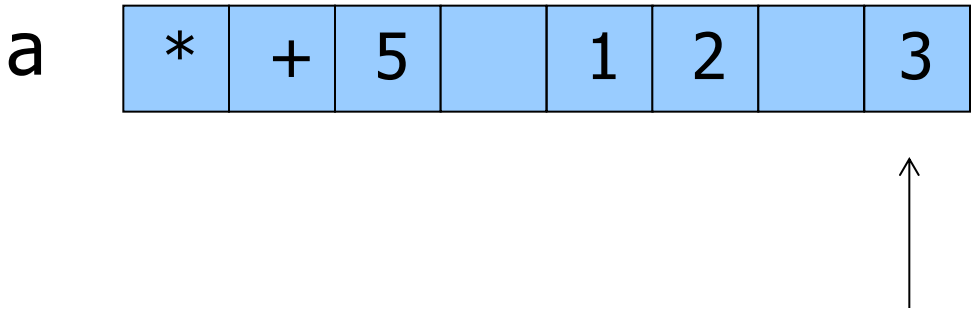
return 12

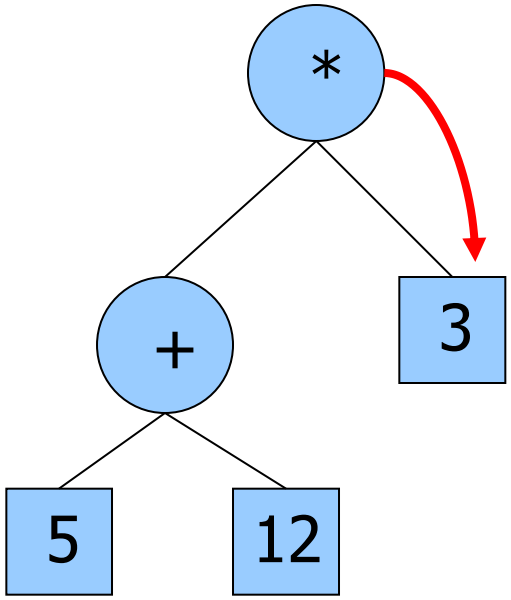
a





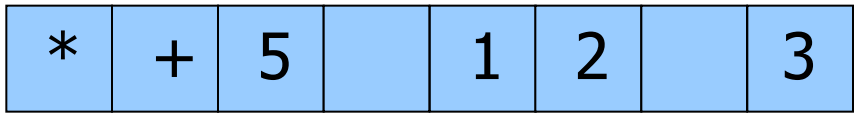

a
*
+
5
1
2
3
return 17
*
+
3
5
12

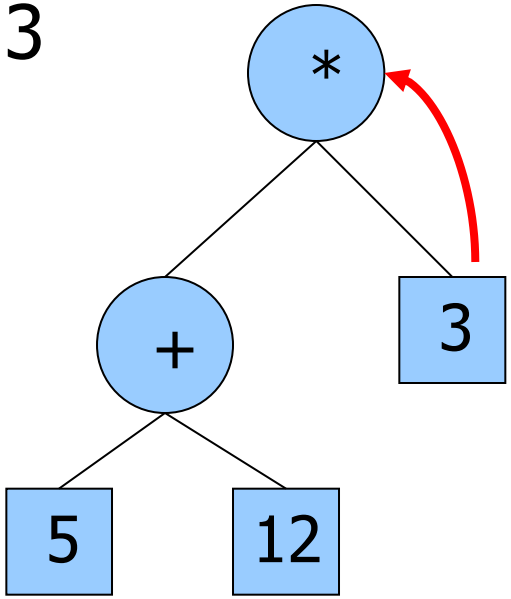

a
*
+
5
1
2
3


*
+
3
5
12

return 3

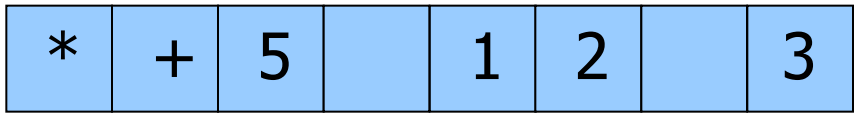
a

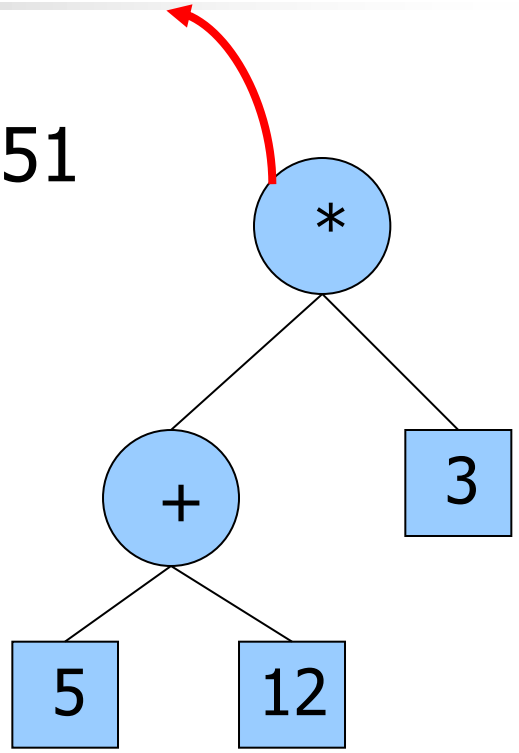


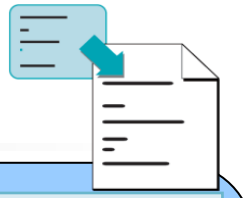


return 51

a





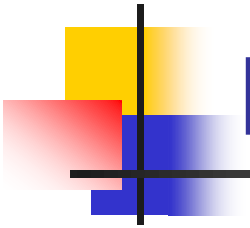


10prefix_eval.c

```
#include <stdio.h>
#include <string.h>
#include "Item.h“
static char *a;
static int i;
int eval();
main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("Error: missing argument\n");
    printf("The correct format is:\n");
    printf("%s \"prefix exp. with + and *\"\n", argv[0]);
    return 0;
  }
  a = argv[1];
  i = 0 ;
  printf("Result = %d", eval());
}
```

variabili globali

```
int eval() {
  int x = 0;
  while (a[i] == ' ')
    i++;
  if (a[i] == '+') {
    i++;
    return eval() + eval ();
  }
  if (a[i] == '*') {
    i++;
    return eval() * eval ();
  }
  while ((a[i] >= '0') && (a[i] <= '9'))
    x = 10 * x + (a[i++]-'0') ;
  return x;
}
```



Problemi ricorsivi semplici

Matematica ricreativa:

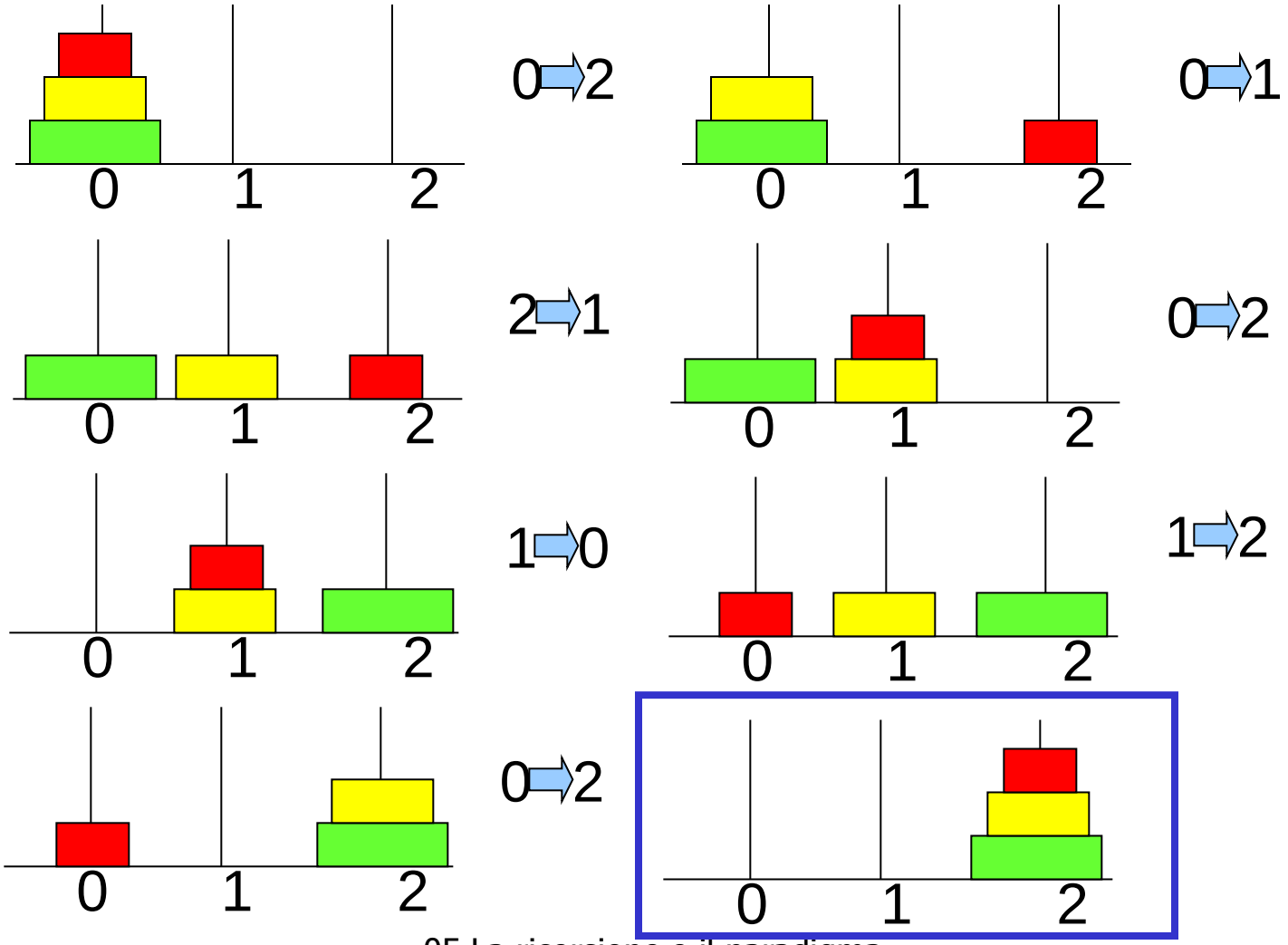
- Le Torri di Hanoi
- Il righello.

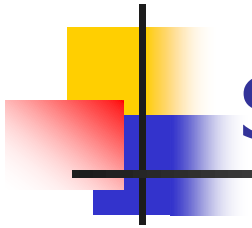
divide and conquer
$a = 2$ $k_i = 1$

Le Torri di Hanoi (E. Lucas 1883)

- Configurazione iniziale:
  - vi sono 3 pioli, 3 dischi di diametro decrescente sul primo piolo
- Configurazione finale:
  - 3 dischi sul terzo piolo
- Regole:
  - accesso solo al disco in cima
  - sopra ogni disco solo dischi più piccoli
- Generalizzabile a n dischi e k pioli.

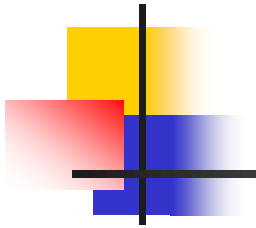
Esempio di soluzione



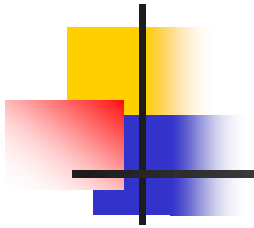


Strategia divide et impera

- Problema iniziale: spostare n dischi da 0 a 2
- Riduzione a sottoproblemi:
  - n-1 dischi da 0 a 1, 2 deposito
  - l'ultimo disco da 0 a 2
  - n-1 dischi da 1 a 2, 0 deposito
- Condizione di terminazione: si muove 1 solo disco.



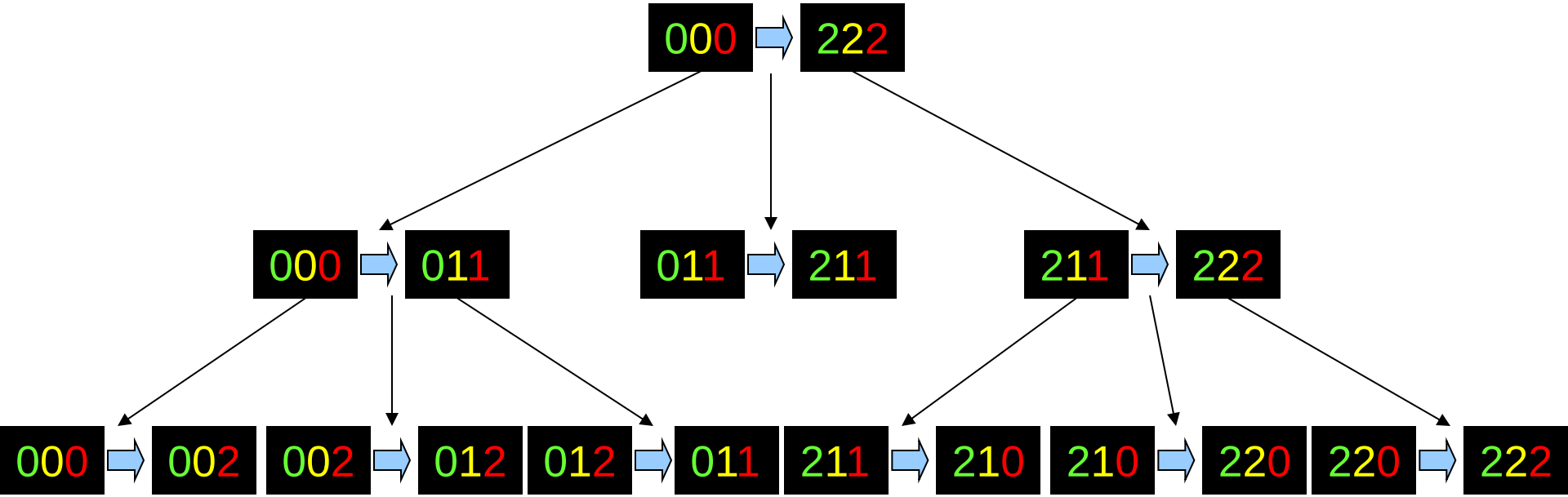
- 0, 1, 2: piolo 0, 1, 2
- disco grande
- disco medio
- disco piccolo
- 0 significa disco piccolo su piolo 0, 2 significa disco grande su piolo 2, etc.
- stato 011
- transizione di stato ⇨



Problema 000 ⇨ 222 decomposto in 3 sottoproblemi di cui 1 elementare:

1. dischi medio e piccolo da 0 a 1 000 ⇨ 011
2. disco grande da 0 a 2 011 ⇨ 211
3. dischi medio e piccolo da 1 a 2. 211 ⇨ 222

Albero della ricorsione



11towers_of_hanoi.c

```
void Hanoi(int n, int src, int dest) {
  int aux;
  aux = 3 - (src + dest);
  if (n == 1) {
    printf("src %d -> dest %d \n", src, dest);
    return;
  }
  Hanoi(n-1, src, aux);
  printf("src %d -> dest %d \n", src, dest);
  Hanoi(n-1, aux, dest);
}
```

terminazione

divisione

chiamata ricorsiva

soluzione elementare

divisione

chiamata ricorsiva

Analisi di complessità

- Dividi: considera n-1 dischi

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

- Risolvi: risolve 2 sottoproblemi di dimensione n-1 ciascuno

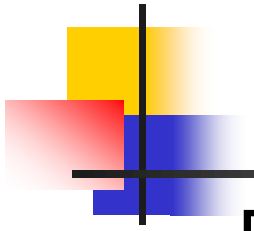
$$2T(n-1)$$

- Terminazione: spostamento di 1 disco

$$\Theta(1)$$

- Combina: nessuna azione

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



Equazione alle ricorrenze:

$$T(n) = 2T(n-1) + 1 \qquad n>1$$

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

$$T(n) = 1 + 2 + 4 + 8T(n-3)$$

$$= \Sigma_{0 \le i \le n-1} 2^i$$

$$= 2^{n-1+1} - 1/(2-1)$$

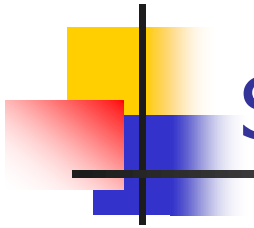
$$= 2^n - 1$$

$$T(n) = O(2^n)$$

Il righello

divide and conquer
a = 2 b = 2

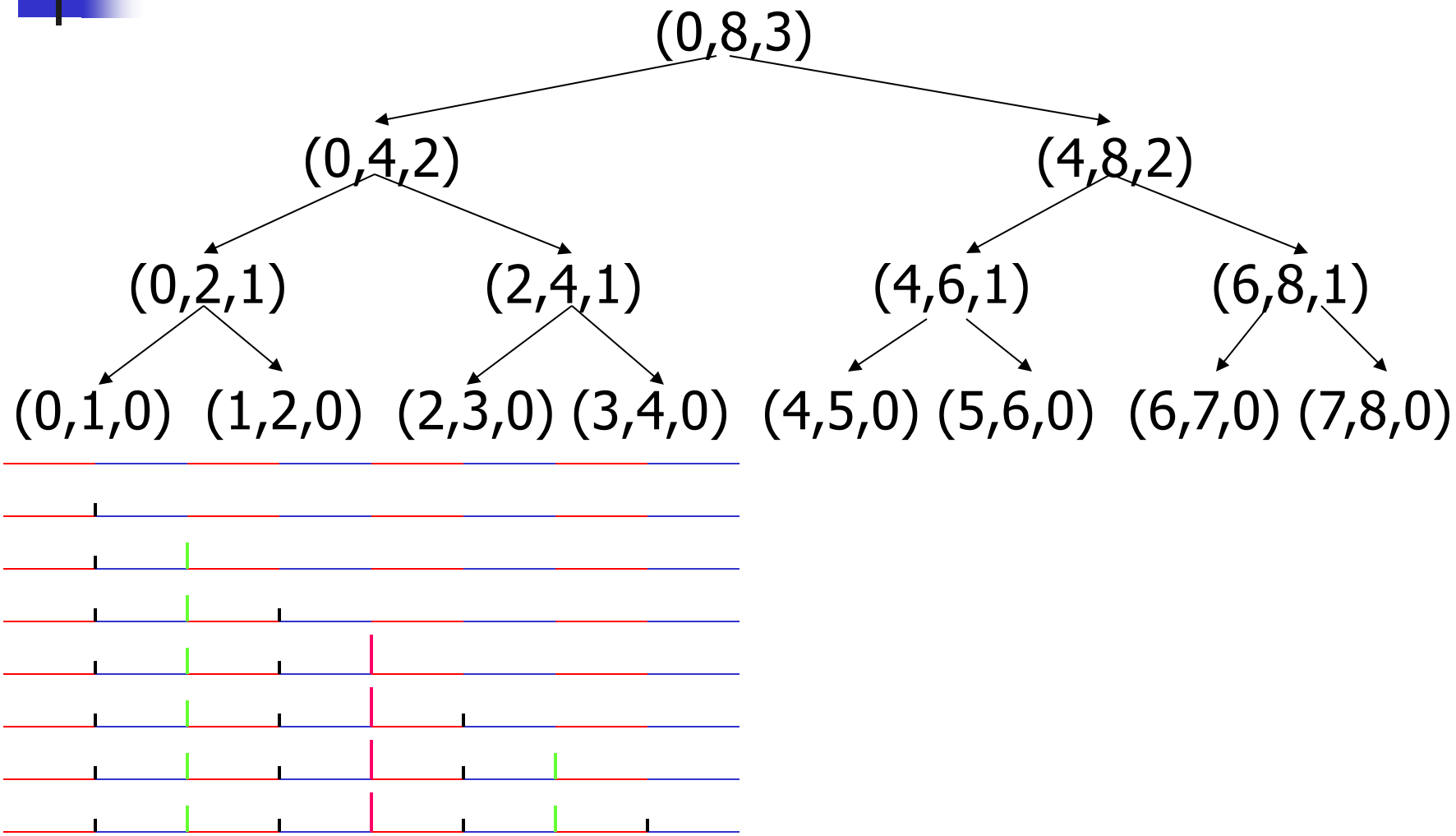
- Tracciare una tacca in ogni punto tra 0 e 2^n estremi esclusi, dove:
  - la tacca centrale è alta n unità,
  - le due tacche al centro delle due metà di destra e sinistra sono alte n-1
  - etc.
  - `mark(x, h)` traccia una tacca alta h unità in posizione x



Strategia divide et impera

- Dividiamo l'intervallo in due metà
- Disegniamo ricorsivamente le tacche (più corte) nella metà di SX
- Disegniamo la tacca (più lunga) al centro
- Disegniamo ricorsivamente le tacche (più corte) nella metà di DX
- Condizione di terminazione: tacche di altezza 0

Esempio



12ruler.c

```
void ruler(int l, int r, int h) {
  int m;
  m = (l + r)/2;
  if (h > 0) {
    ruler(l, m, h-1);
    mark(m, h);
    ruler(m, r, h-1);
  }
}

void mark(int m, int h) {
  int i;
  printf("%d \t", m);
  for (i = 0; i < h; i++)
    printf("*");
  printf("\n");
}
```

discesa ricorsiva/terminazione

chiamata ricorsiva

divisione

soluzione elementare

chiamata ricorsiva

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2$, $b = 2$

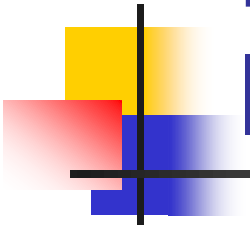
Equazione alla ricorrenze:

$T(n) = 2T(n/2) + 1$ $\quad n > 1$

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

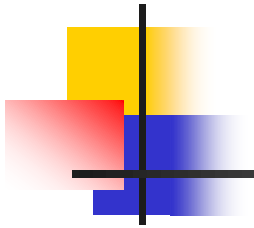
$T(n) = O(n)$

Meccanismi computazionali per l'esecuzione di funzioni ricorsive



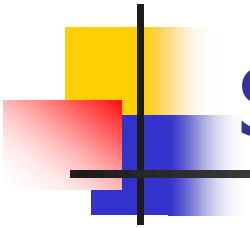
Chiamata a funzione: quando si chiama una funzione:

- si crea una nuova istanza della funzione chiamata
- si alloca memoria per i parametri e per le variabili locali
- si passano i parametri
- il controllo passa dal chiamante alla funzione chiamata
- si esegue la funzione chiamata



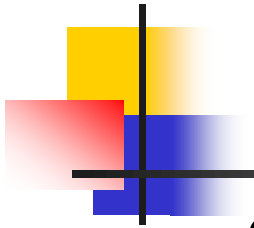
- al suo termine, il controllo ritorna al programma chiamante, che esegue l'istruzione immediatamente successiva alla chiamata a funzione.

È possibile che una funzione ne chiami un'altra. Serve un meccanismo per gestire le chiamate annidate e i relativi ritorni: lo **stack**.



Stack

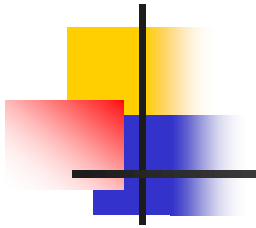
- Definizione: tipo di dato astratto (ADT) che supporta operazioni di:
  - Push: inserimento dell'oggetto in cima allo stack
  - Pop: prelievo (e cancellazione) dalla cima dell'oggetto inserito più di recente
- Terminologia: la strategia di gestione dei dati è detta LIFO (Last In First Out)



Si chiama **stack frame** (o **record di attivazione**) la struttura dati che contiene almeno:

- i parametri formali
- le variabili locali
- l'indirizzo a cui si ritornerà una volta terminata l'esecuzione della funzione
- il puntatore al codice della funzione.

Lo stack frame viene creato alla chiamata della funzione e distrutto al suo termine.



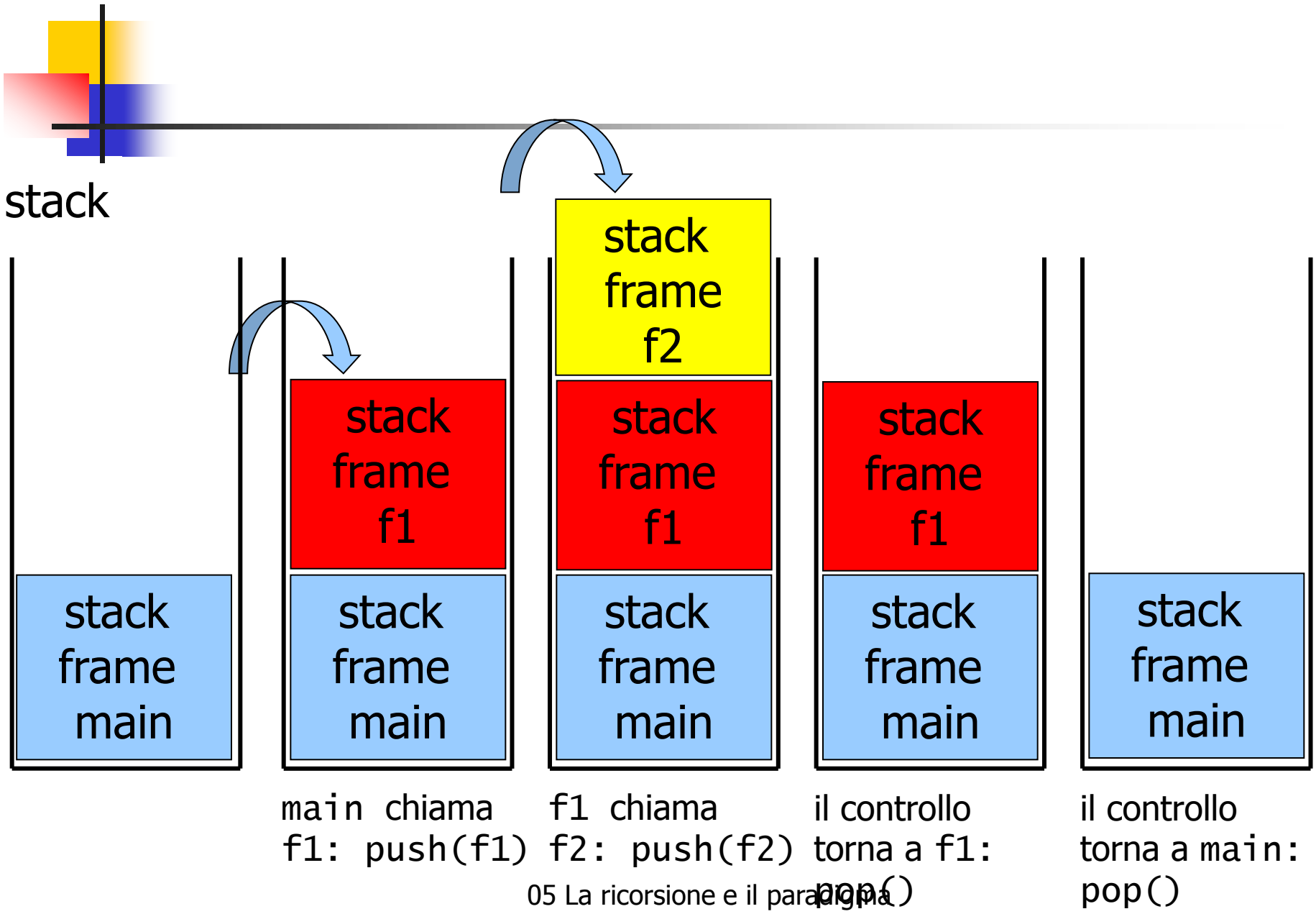
Gli stack frame sono memorizzati nello stack di sistema.

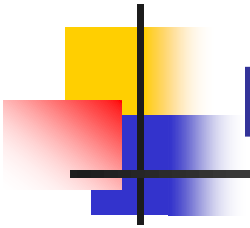
Lo stack di sistema ha a disposizione una quantità prefissata di memoria. Quando oltrepassa lo spazio allocatogli, c'è **stack overflow**.

Lo stack cresce da indirizzi maggiori a indirizzi minori (quindi verso l'alto). Lo **stack pointer SP** è un registro che contiene l'indirizzo del primo stack frame disponibile.

Esempio

```
int f1(int x);
int f2(int x);

main() {
  int x, a = 10;
  x = f1(a);
  printf("x is %d \n", x);
}

int f1(int x) { return f2(x); }

int f2(int x) { return x+1; }
```

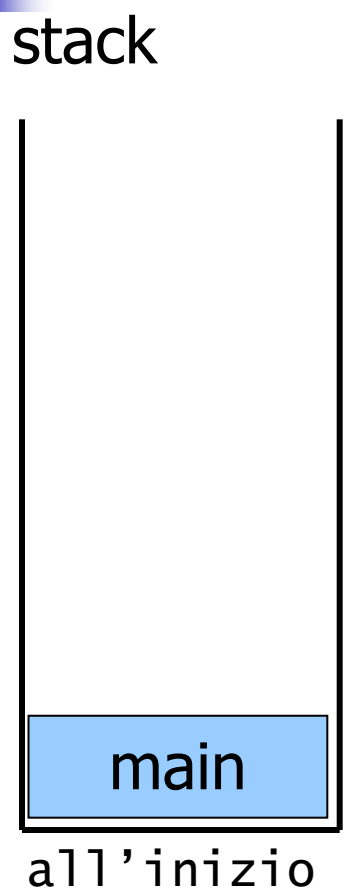

stack
stack frame f2
stack frame f1
stack frame main
stack frame f1
stack frame main
stack frame f1
stack frame main
stack frame main
main chiama f1: push(f1)
f1 chiama f2: push(f2)
il controllo torna a f1: pop()
il controllo torna a main: pop()



Funzioni ricorsive

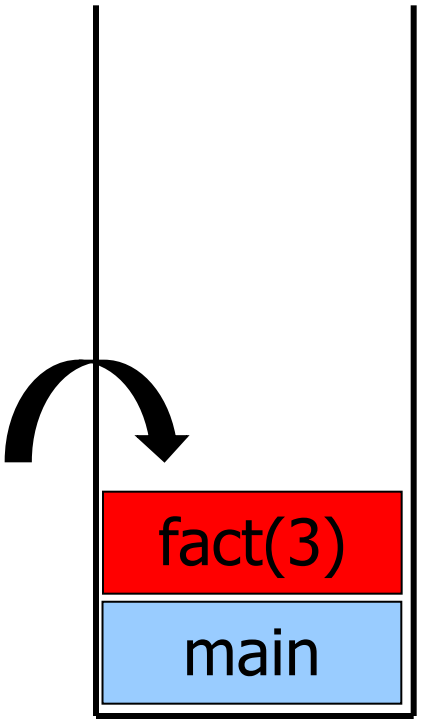
- Funzione chiamante e chiamata coincidono, operano però su valori diversi
- Si usa lo stack di sistema come in una qualsiasi chiamata a funzione
- Un numero eccessivo di chiamate ricorsive può portare allo stack overflow.

Esempio: calcolo di 3!




```
main() {
  int n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(int n) {
  if(n == 0)
    return(1);
  else
    return(n * fact(n-1));
}
```

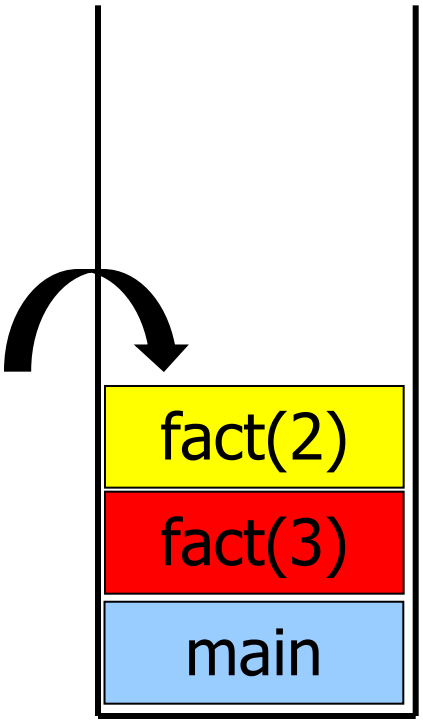
stack



main chiama
fact(3)

3! = 3*2!

stack

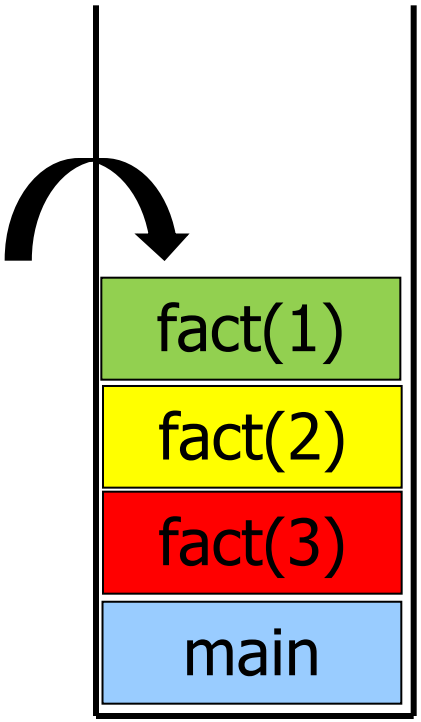


fact(3) chiama fact(2)

3! = 3*2!

2! = 2*1!

stack



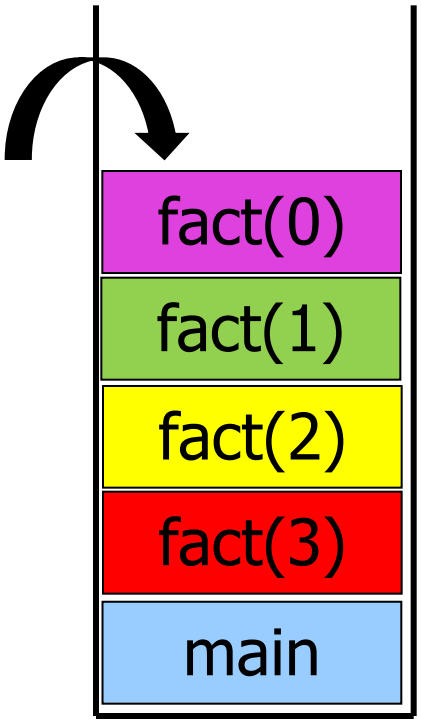
`fact(2) chiama fact(1)`

3! = 3*2!

2! = 2*1!

1! = 1*0!

stack



`fact(1) chiama fact(0)`

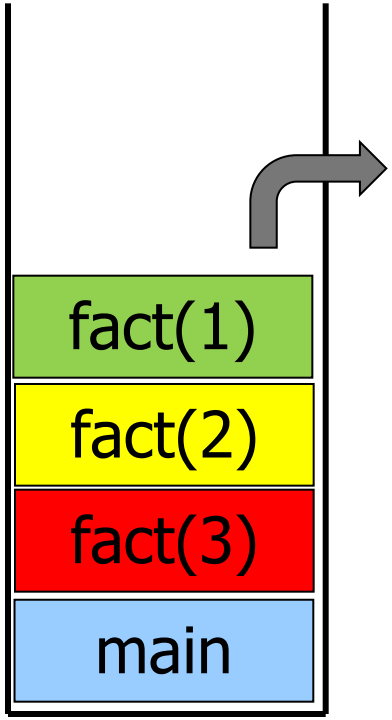
3! = 3*2!

2! = 2*1!

1! = 1*0!

0! = 1

stack



3! = 3*2!

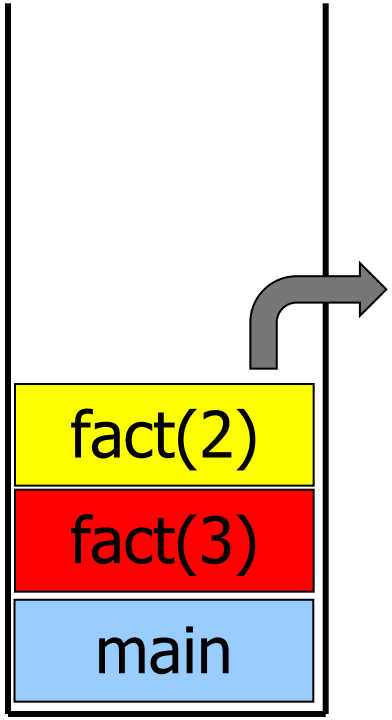
2! = 2*1!

1! = 1*0!

0! = 1

fact(0) termina, restituisce
il valore 1 e torna il controllo
a fact(1)

stack



3! = 3*2!

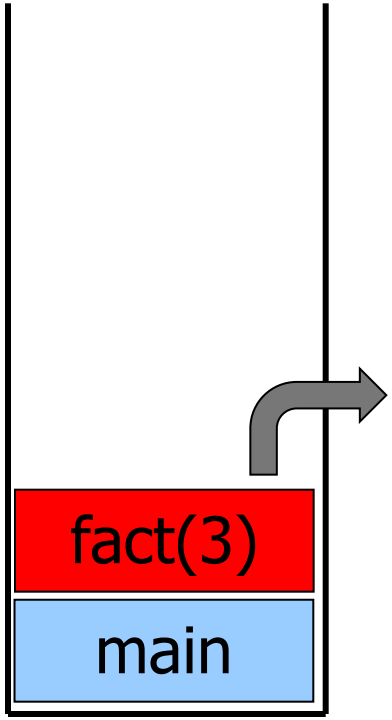
2! = 2*1!

1! = 1*0! = 1

0! = 1

fact(1) termina, restituisce
il valore 1 e torna il controllo
a fact(2)

stack



3! = 3*2!

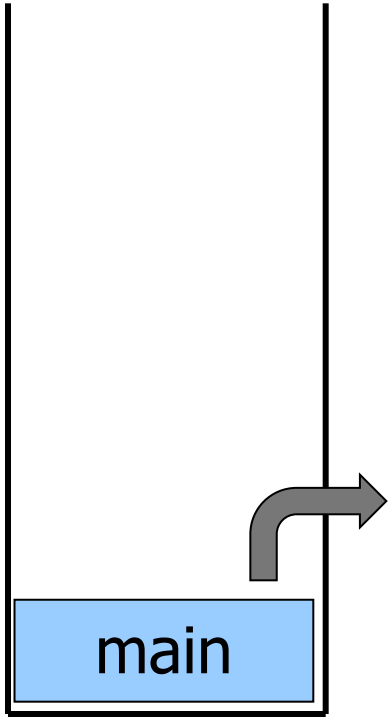
2! = 2*1! = 2

1! = 1*0! = 1

0! = 1

fact(2) termina, restituisce
il valore 2 e torna il controllo
a fact(3)

stack



3! = 3*2!= 6

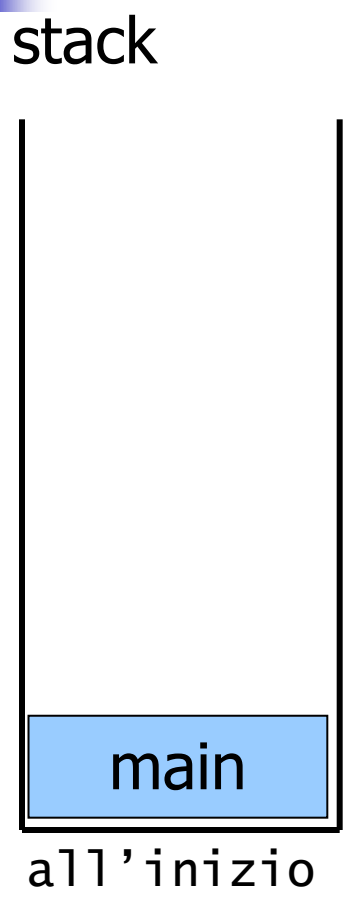
2! = 2*1!= 2

1! = 1*0! = 1

0! = 1

fact(3) termina, restituisce
il valore 6 e torna il controllo
al main

Esempio: reverse_print di "abc"



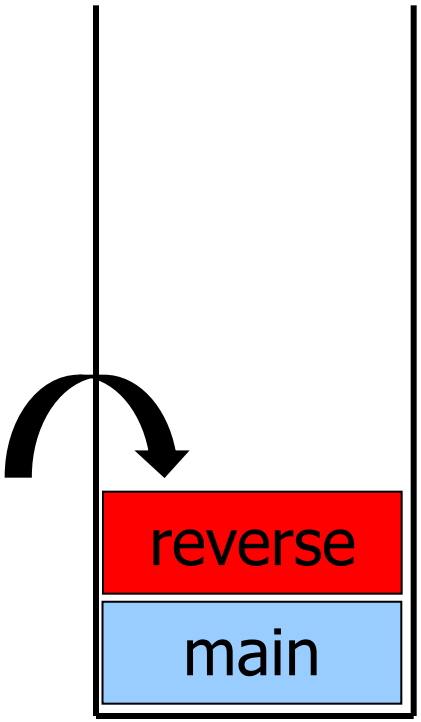

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

Input: "abc"

str

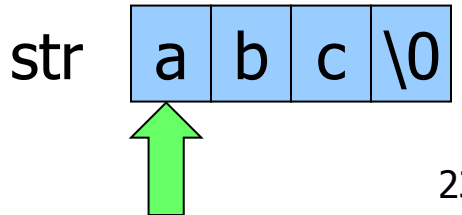
a	b	c	\0

stack

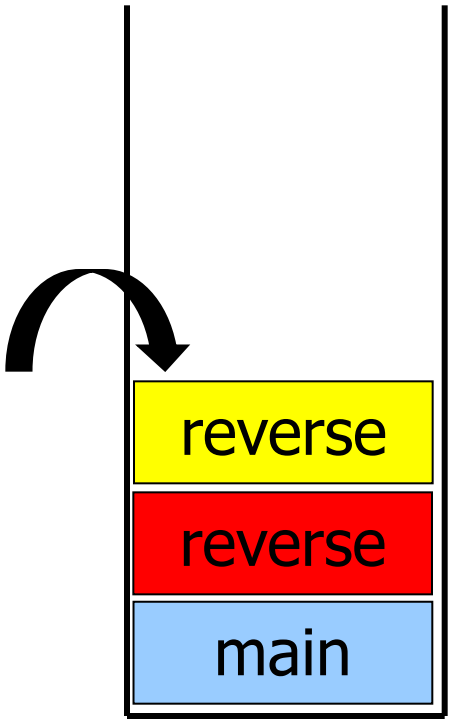


main chiama
reverse_print

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

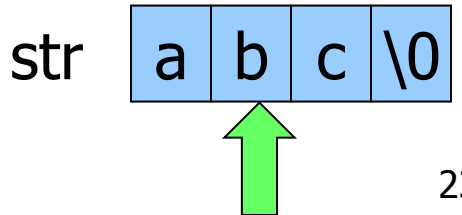


stack

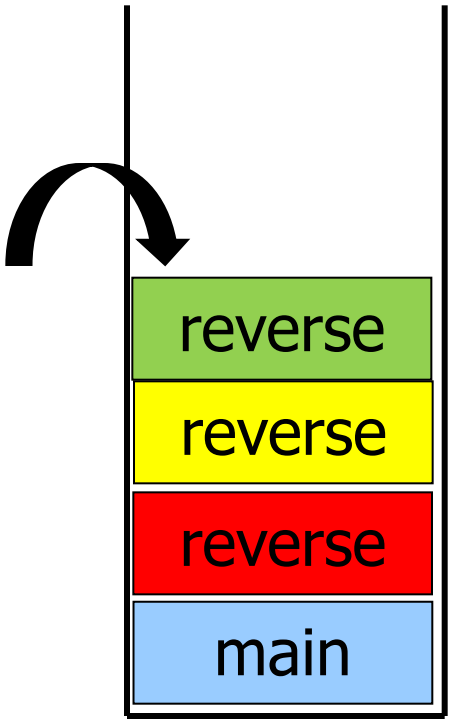


reverse_print chiama se stessa

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

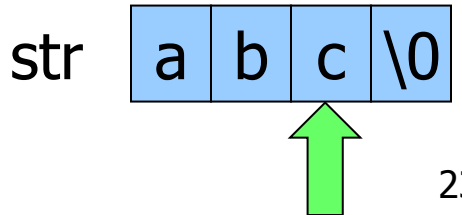


stack

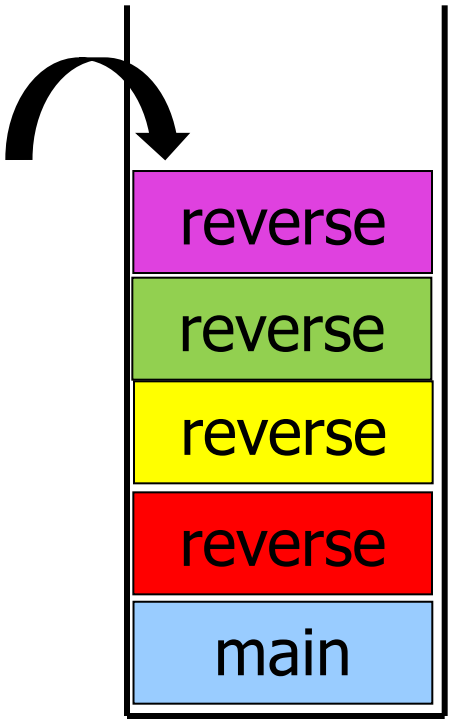


reverse_print chiama se stessa

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

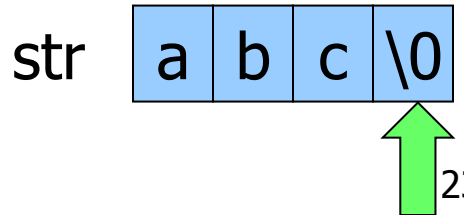


stack

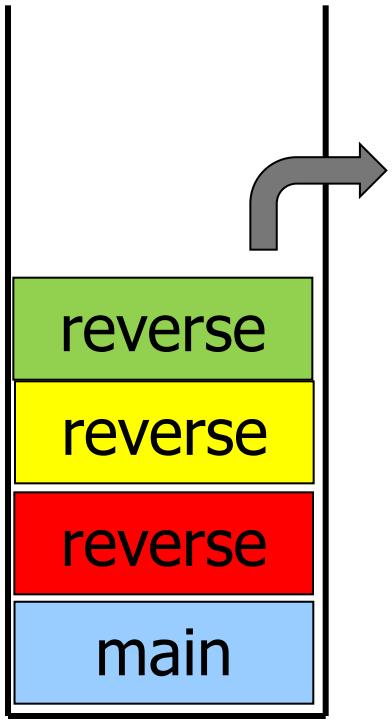


reverse_print chiama se stessa

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

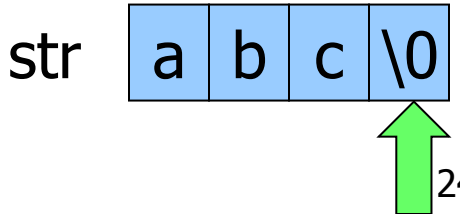


stack

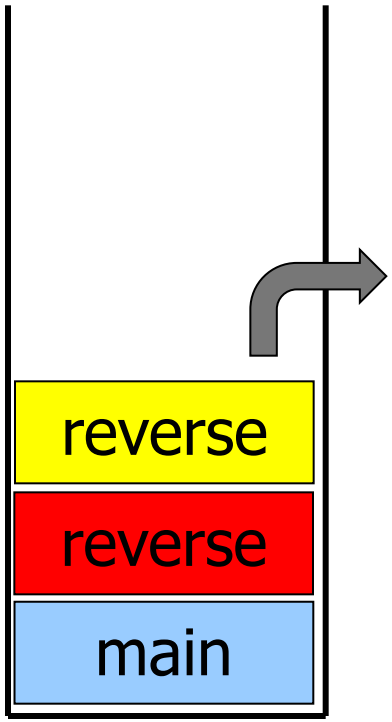


condizione
di terminazione
return

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```



stack

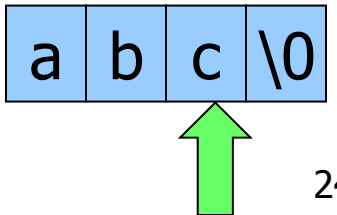


stampa di c
return

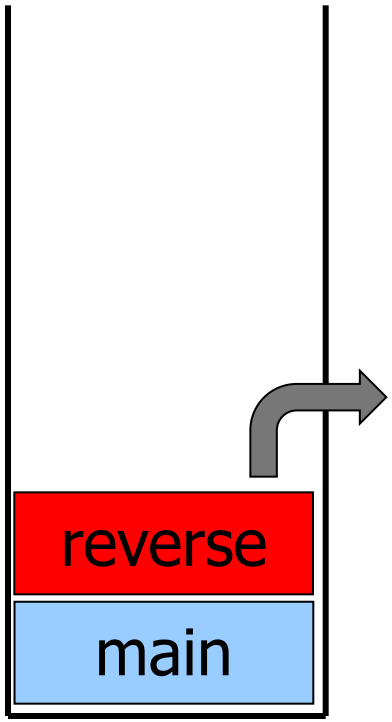
```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

Output: "c"

str



stack

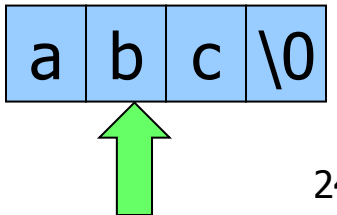


stampa di b
return

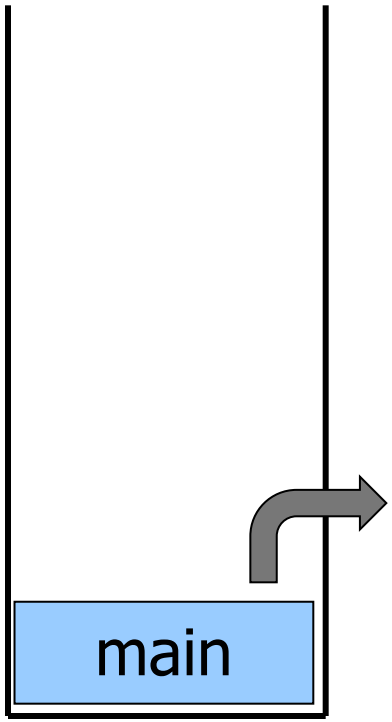
```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

Output: "cb"

str



stack

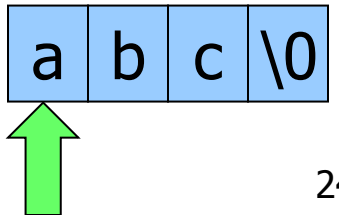


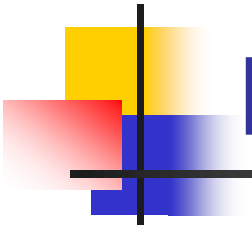
stampa di a
return

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

Output: "cba"

str





Emulazione della ricorsione

- La ricorsione può consumare molta memoria
- Si può emulare la ricorsione gestendo esplicitamente lo stack
- La soluzione migliore (efficienza e chiarezza del codice) dipende dal problema. Opportuno cercare di stare al massimo livello di astrazione possibile per il problema.

Esempio: il fattoriale

ADT S (stack)

```
long factorial(int n) {
  long fact = 1, curr;
  S stack;
  stack = STACKinit(Nmax);
  STACKpush(stack, n);
  while(STACKsize(stack) > 0) {
    curr = STACKpop(stack);
    fact = fact * curr;
    if(curr != 1)
       STACKpush(stack, curr-1);
    }
  return fact;
}
```

Funzioni tail-recursive

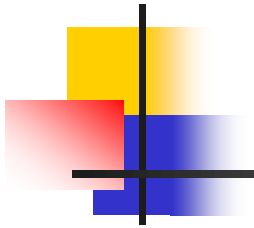
La chiamata ricorsiva è l'ultima operazione da eseguire, eccezion fatta per return

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

non è tail-recursive perché la moltiplicazione può essere eseguita solo dopo il ritorno della chiamata ricorsiva

```
fact(3)
3 * fact(2)
3 * (2 * fact(1))
3 * (2 * (1 * fact(0)))
3 * (2 * (1 * 1))
```

È necessario mantenere uno stack!



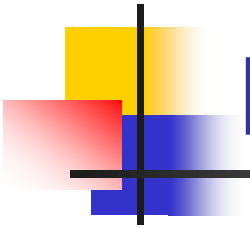
Versione tail-recursive

```
long fact_r(int n, long f) {
  if(n == 0)
    return(f);
  else
    return fact_tr(n-1, n*f);
}
```

è tail-recursive perché la moltiplicazione viene eseguita prima della chiamata ricorsiva

```
fact_tr(3,1)
fact_tr(2,3)
fact_tr(1,6)
fact_tr(0,6)
```

Non è necessario mantenere uno stack!



Dualità ricorsione - iterazione

- Ogni programma ricorsivo può anche essere implementato in modo iterative, in generale facendo uso di uno stack
- La soluzione migliore (efficienza e chiarezza del codice) dipende dal problema.

Non è necessario mantenere uno stack!

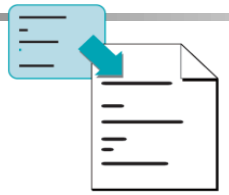
13iterative_factorial.c

Fattoriale:

5! = 1*2*3*4*5 = 120

```
long fact(int n) {
  long tot = 1;
   int i;
   for (i=2; i<=n; i++)
       tot = tot * i;
   return(tot);
}
```

Non è necessario mantenere uno stack!



14iterative_fibonacci.c

Fibonacci:

$FIB_0 = 0$

$FIB_1 = 1$

$FIB_2 = FIB_0 + FIB_1 = 1$

$FIB_3 = FIB_1 + FIB_2 = 2$

$FIB_4 = FIB_2 + FIB_3 = 3$

$FIB_5 = FIB_3 + FIB_4 = 5$

```
long fib(int n) {
  long f1p=1, f2p=0, f;
  int i;
  if(n == 0 || n == 1)
    return(n);
  f = f1p + f2p; /* n==2 */
  for(i=3; i<= n; i++) {
     f2p = f1p;
     f1p = f;
     f = f1p+f2p;
   }
 return(f);
}
```

Non è necessario mantenere uno stack!

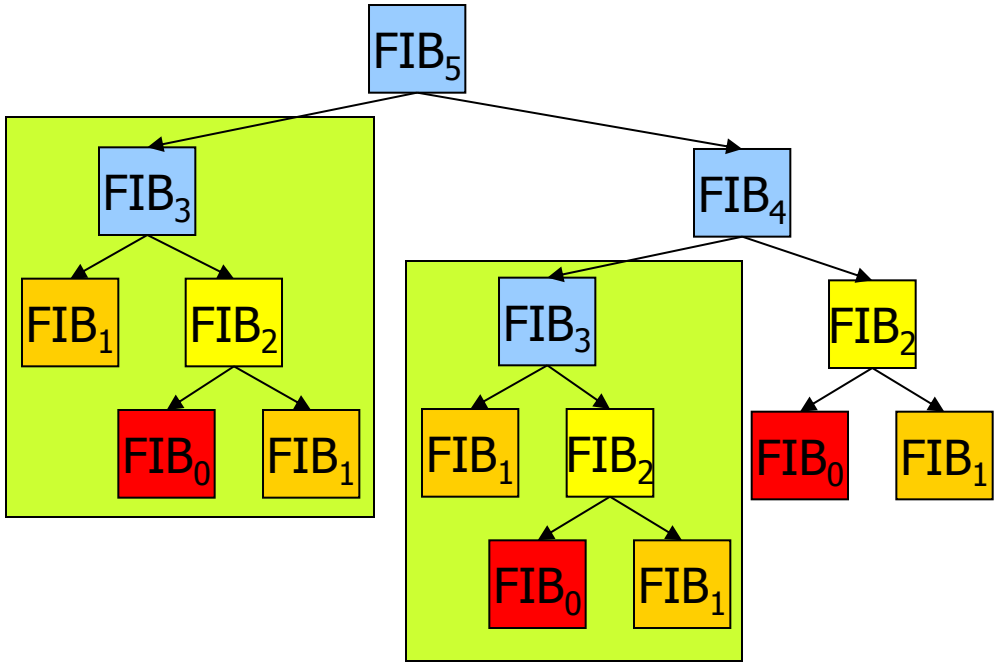
15iterative_bisearch.c

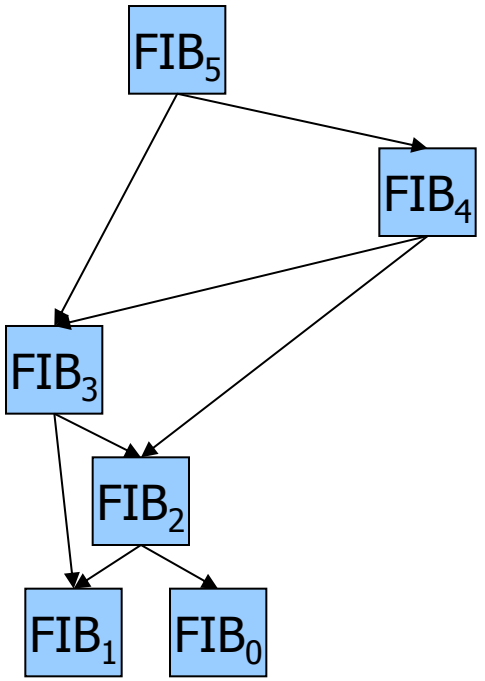
Ricerca binaria

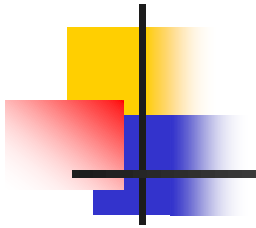
```
int BinSearch(int v[], int l, int r, int k) {
   int m;
   while((r-l) != 0) {
     m = (l+r) / 2;
     if(v[m] >= k)
       r = m;
     else
       l = m+1;
     }
   if(v[l]==k)
     return(l);
   else
     return(-1);
}
```

Limiti del divide et impera

- Ipotesi di indipendenza dei sottoproblemi
- Memoria occupata

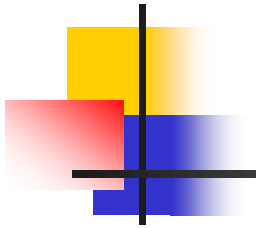




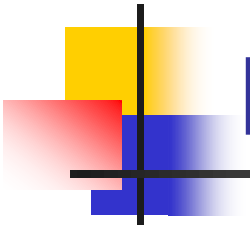


Paradigma alternativo: Programmazione dinamica:

- memorizza le soluzioni ai sottoproblemi man mano che vengono trovate
- prima di risolvere un sottoproblema, controlla se è già stato risolto
- meglio del divide et impera per sottoproblemi condivisi

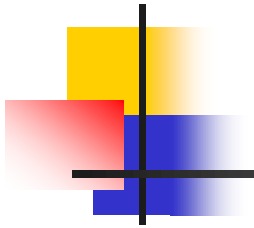


- procede:
  - bottom-up, mentre il divide et impera è top-down
  - top-down e si dice ricorsione con memorizzazione o memoization

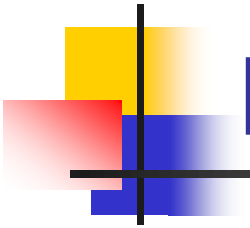


La ricorsione in C

- Strutture dati
  - globali, cioè comuni a tutte le istanze dela funzione ricorsiva
  - locali, cioè locali a ciascuna delle istanze
- Strutture dati globali:
  - dati del problema (matrice, mappa, grafo), vincoli, scelte disponibili, soluzione
- Strutture dati locali:
  - indici di livello di chiamata ricorsiva, copie locali di strutture dati, indici o puntatori a parti di strutture dati globali

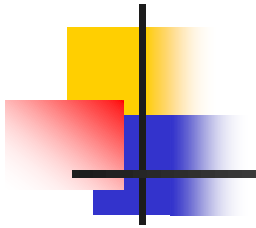


- Globale nell'accezione precedente non implica uso di variabili globali C
- Uso di variabili globali C per strutture dati globali:
  - sconsigliato ma non vietato quando le funzioni ricorsive operano su pochi e ben noti dati
  - vantaggio: pochi parametri passati alle funzioni ricorsive
- Soluzione adottata: tutti i dati (globali e locali) passati come parametri. Possibilità di racchiuderli in una struct per leggibilità.



Riferimenti

- Ricorsione
  - Sedgewick 5.1
  - Deitel 5.14, 5.15
- Divide et Impera
  - Sedgewick 5.2
  - Cormen 1.3.1
- Risoluzione di Equazioni alle Ricorrenze:
  - Cormen 4.2



- Algoritmo di Karatsuba:
  - Crescenzi, Gambosi, Grossi 2.5.2
- Algoritmo di Laplace:
  - Cormen 31.1
- Chiamata di funzioni
  - Deitel 5.7
- Programmazione dinamica:
  - Cormen 16