



Gli Alberi Binari di Ricerca (Binary Search Trees, BST)

Paolo Camurati e Gianpiero Cabodi

Alberi Binari

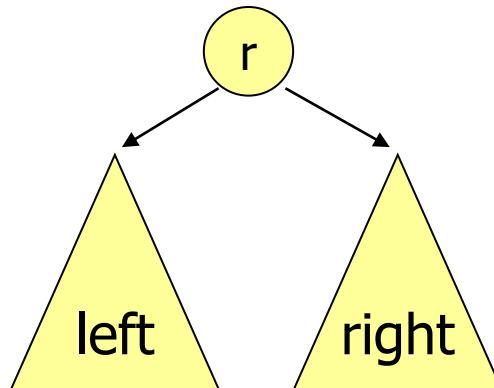
DEFINIZIONI

Alberi binari

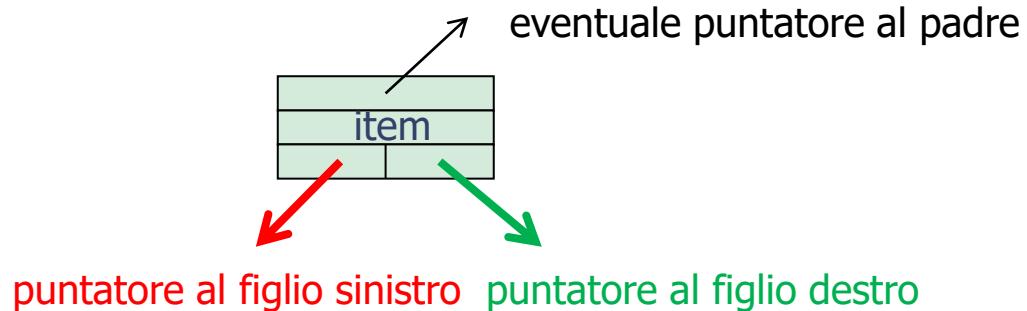
Definizione ricorsiva:

Un albero binario T è:

- un insieme vuoto di nodi vuoto
- una terna formata da una radice, un sottoalbero sinistro e un sottoalbero destro.



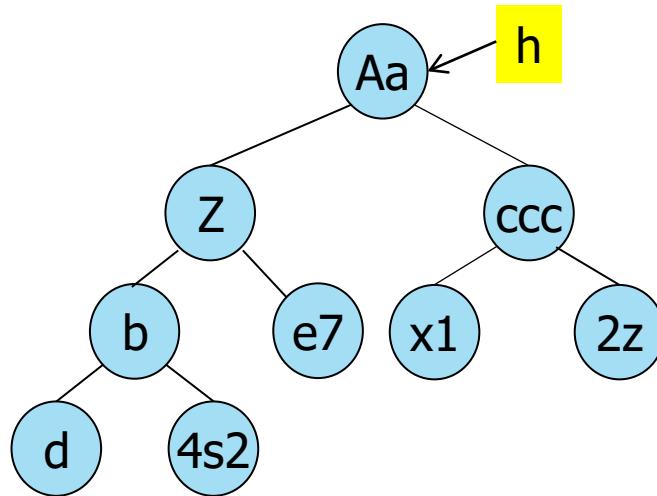
Nodo:



```
typedef struct node* link;
struct node {
    Item item;
    link left;
    link right;
};
```

Accesso

All'albero si accede tramite il puntatore h alla radice:



Calcolo di parametri

numero di nodi

```
int count(link root) {  
    if (root == NULL)  
        return 0;  
    return count(root->left) + count(root->right) + 1;  
}
```

```
int height(link root) {  
    int u, v;  
    if (root == NULL)  
        return -1;  
    u = height(root->left); v = height(root->right);  
    if (u>v)  
        return u+1;  
    return v+1;  
}
```

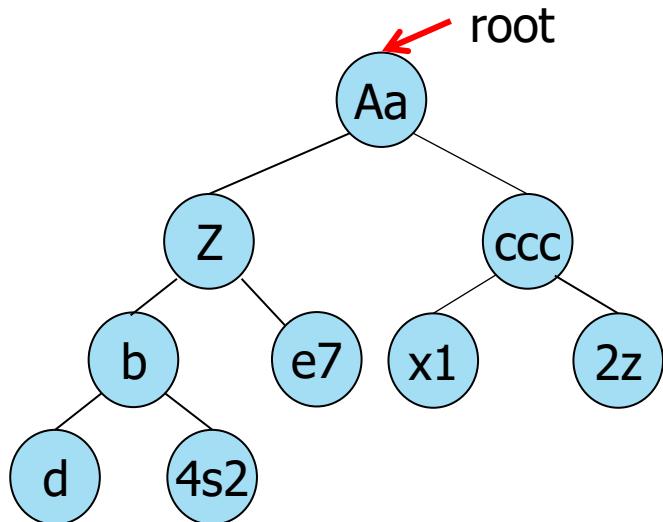
altezza

Visite

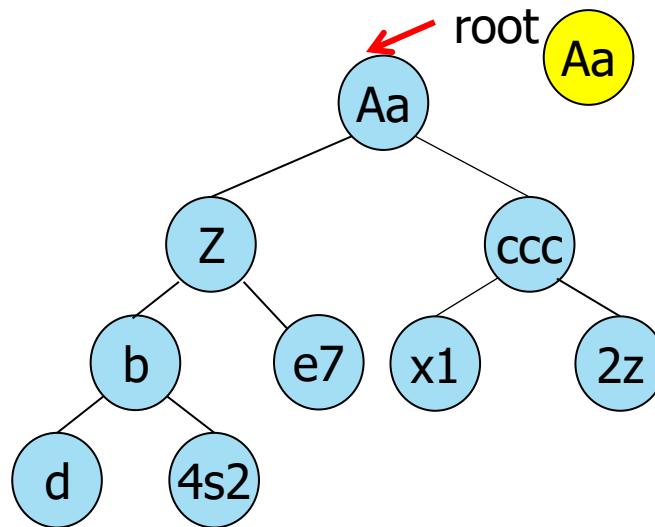
Attraversamento o visita: elenco dei nodi secondo una strategia:

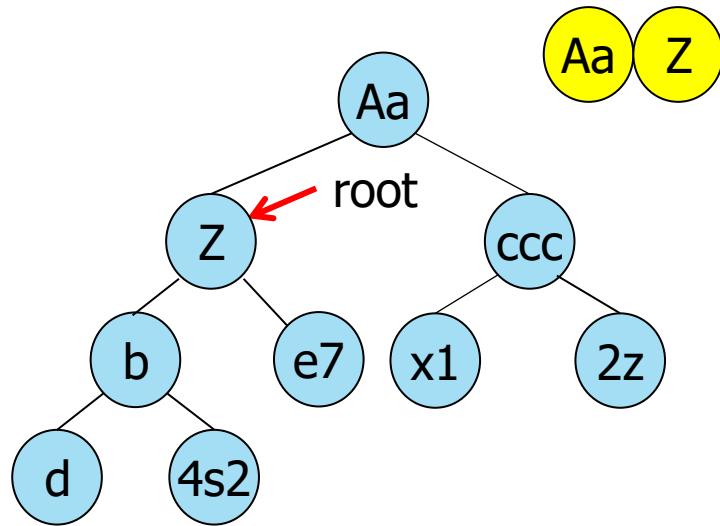
- **pre-ordine**: root, Left(root), Right(root)
- **in-ordine**: Left(root), root, Right(root)
- **post-ordine**: Left(root), Right(root), root

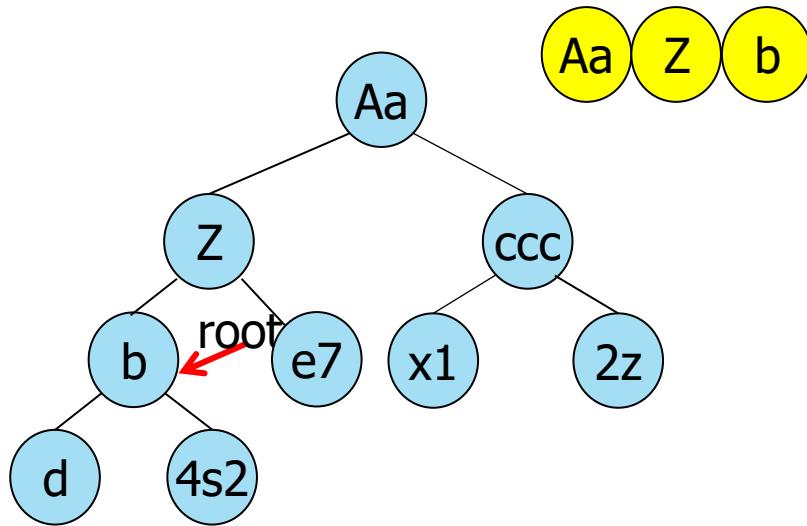
Pre-ordine

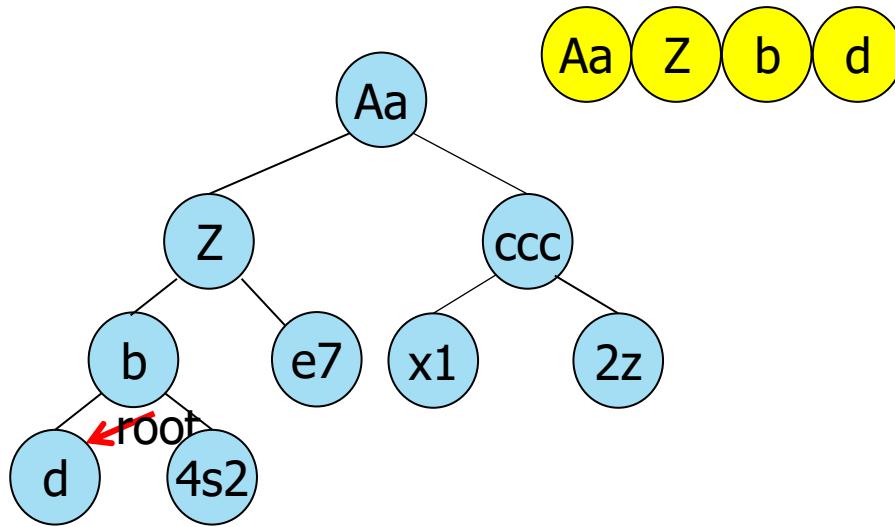


```
void preOrder(link root){  
    if (root == NULL)  
        return;  
    printf("%s ",root->name);  
    preOrder(root->left);  
    preOrder(root->right);  
}
```

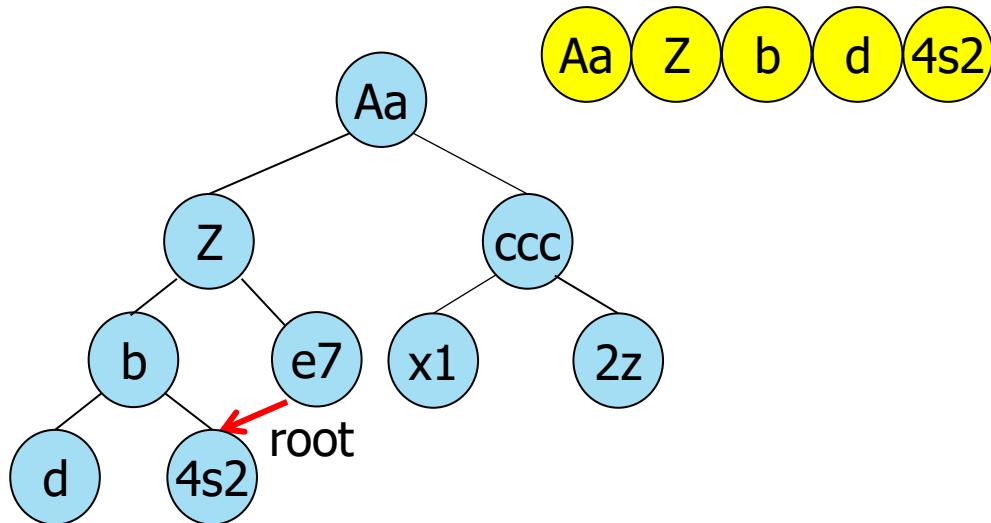


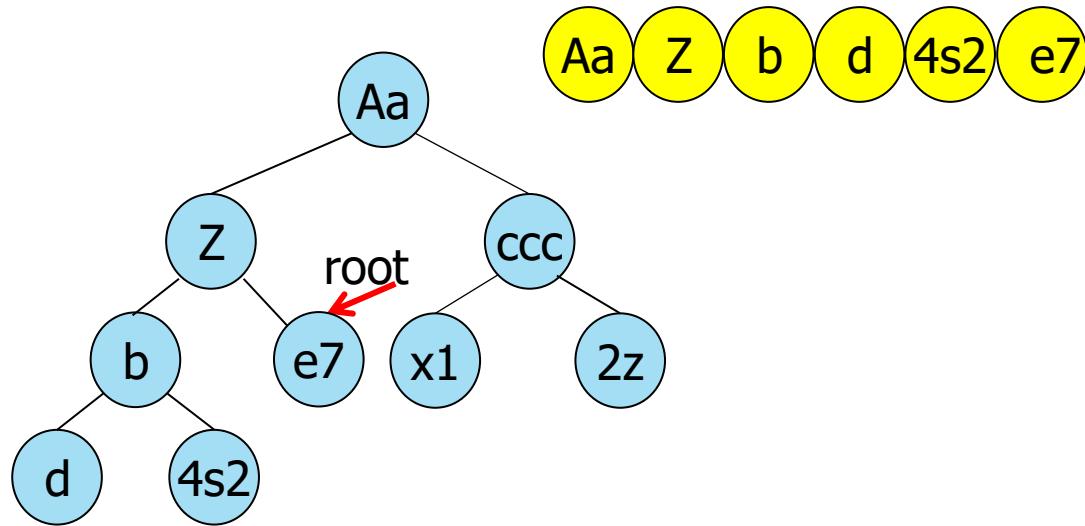


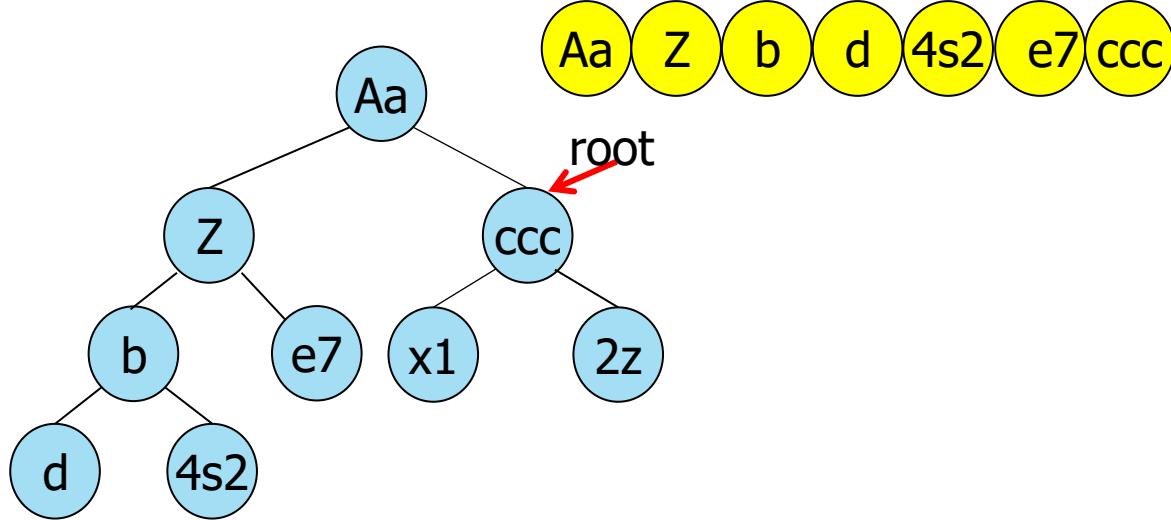


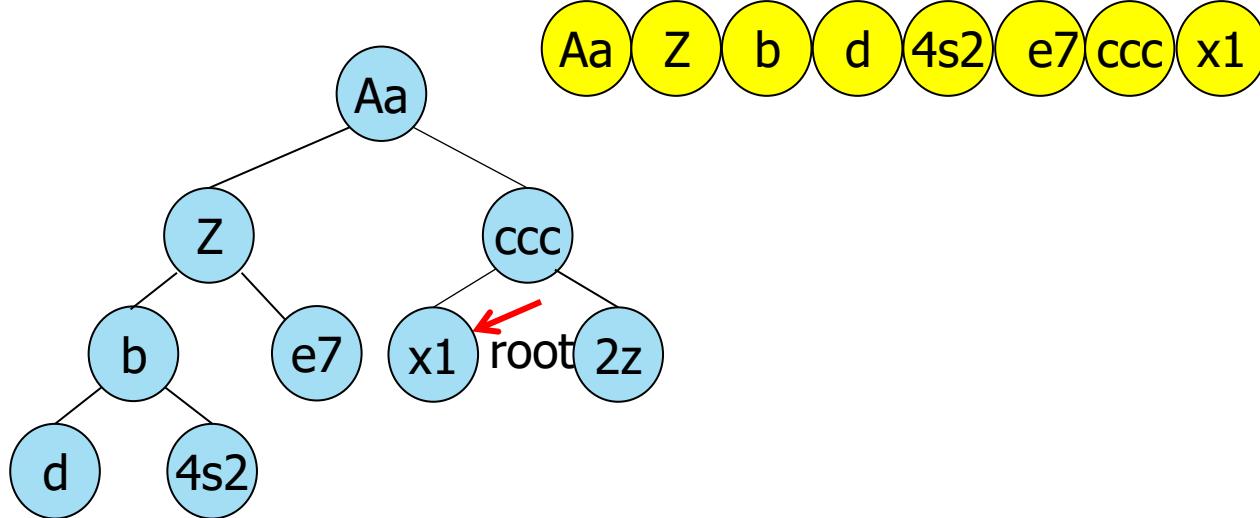


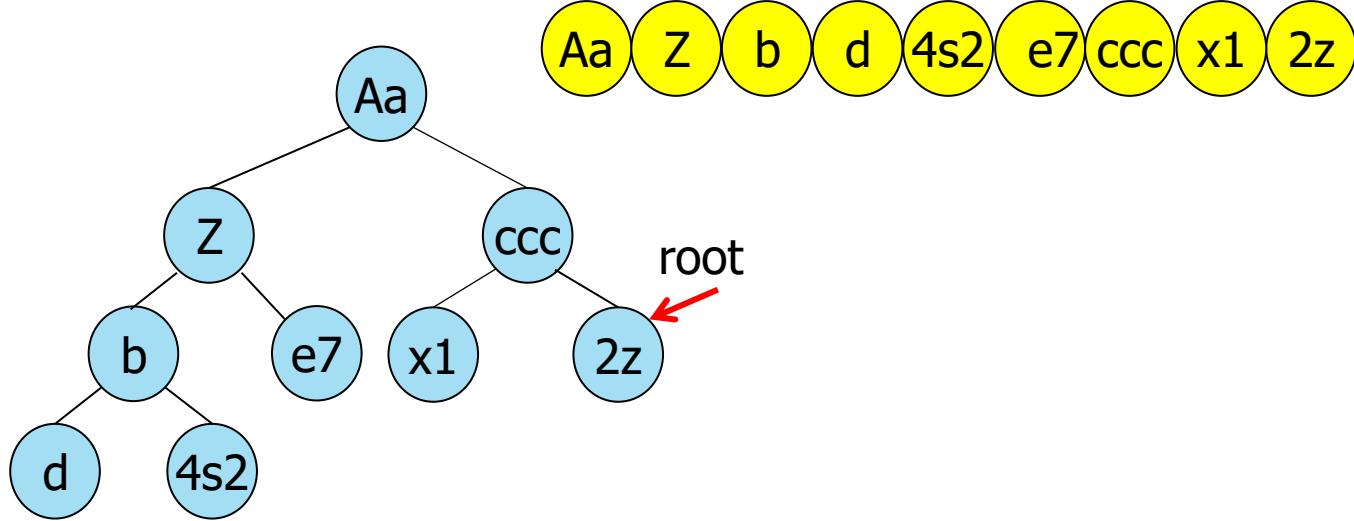
Aa Z b d



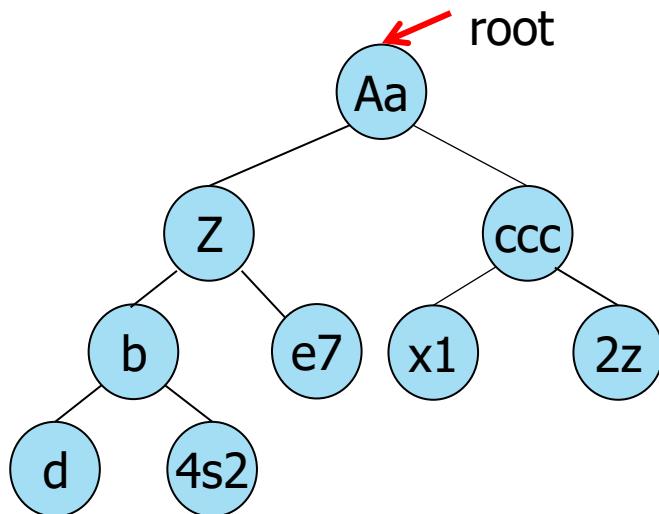




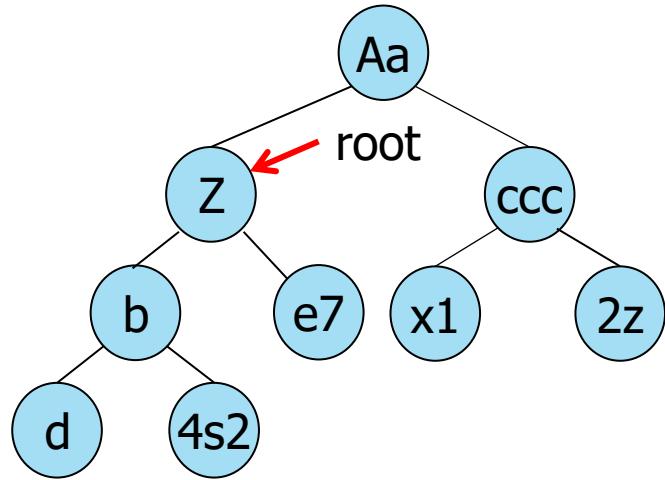


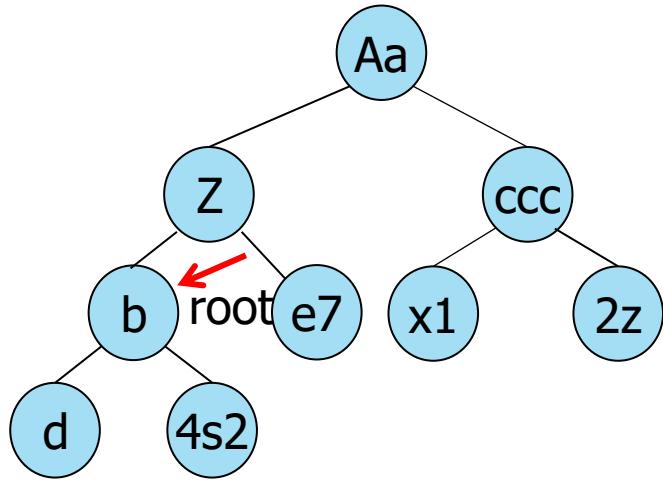


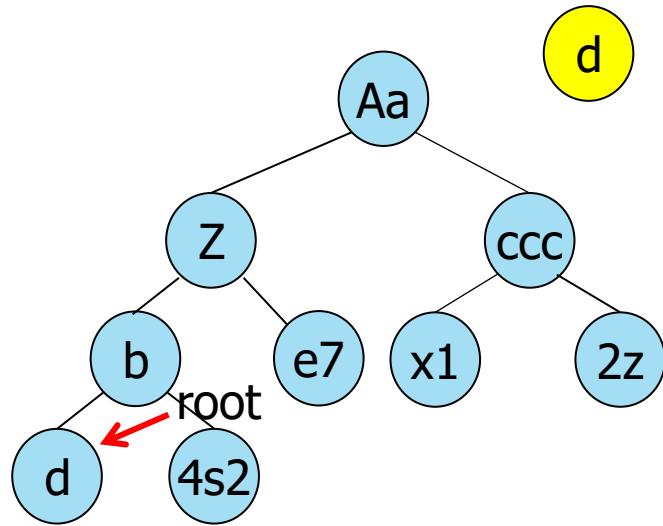
In-ordine

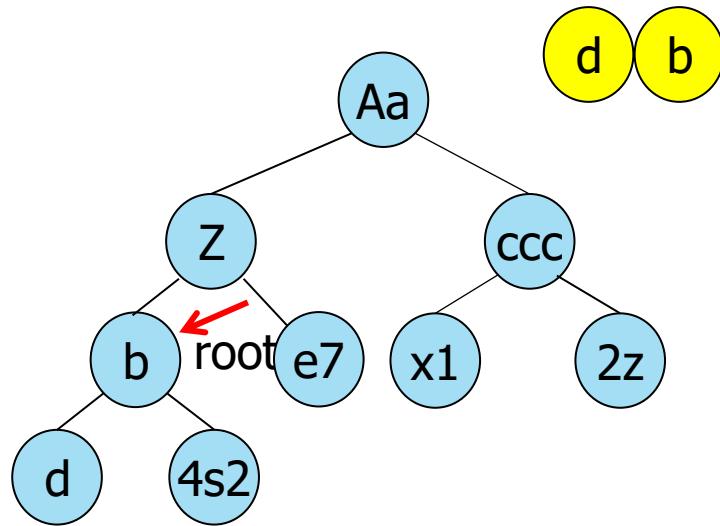


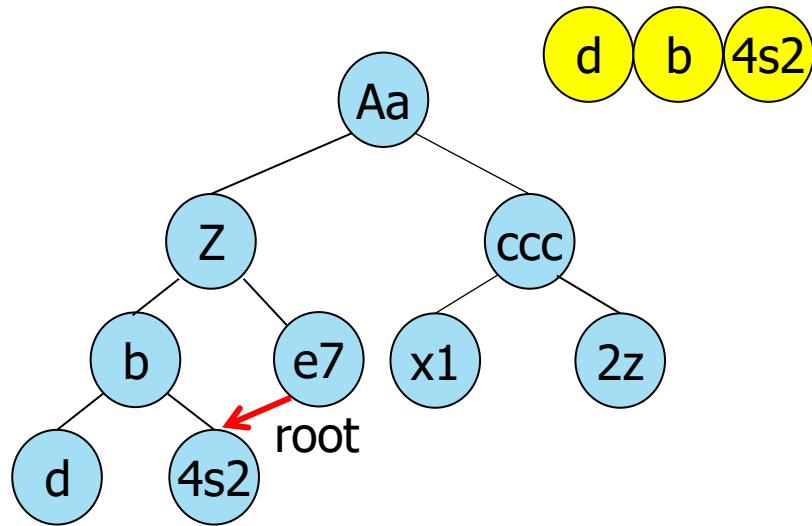
```
void inorder(link root){  
    if (root == NULL)  
        return;  
    inorder(root->left);  
    printf("%s ",root->name);  
    inorder(root->right);  
}
```



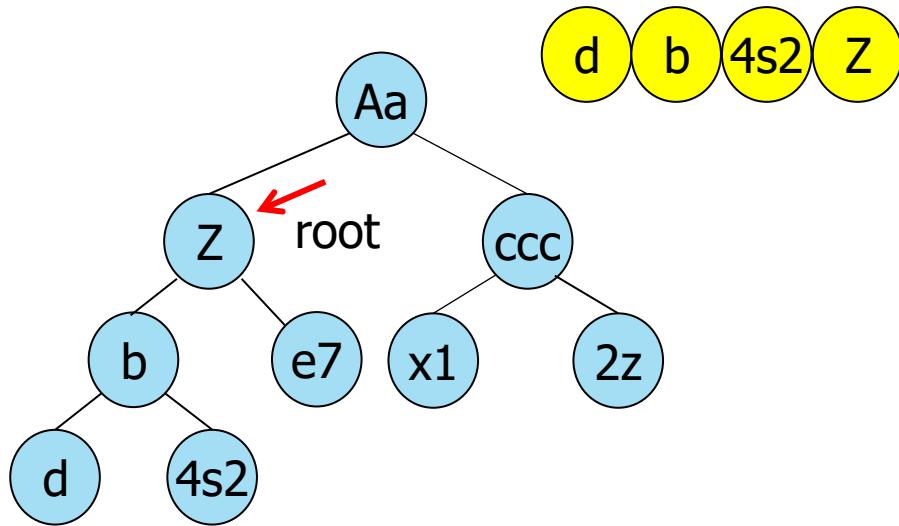




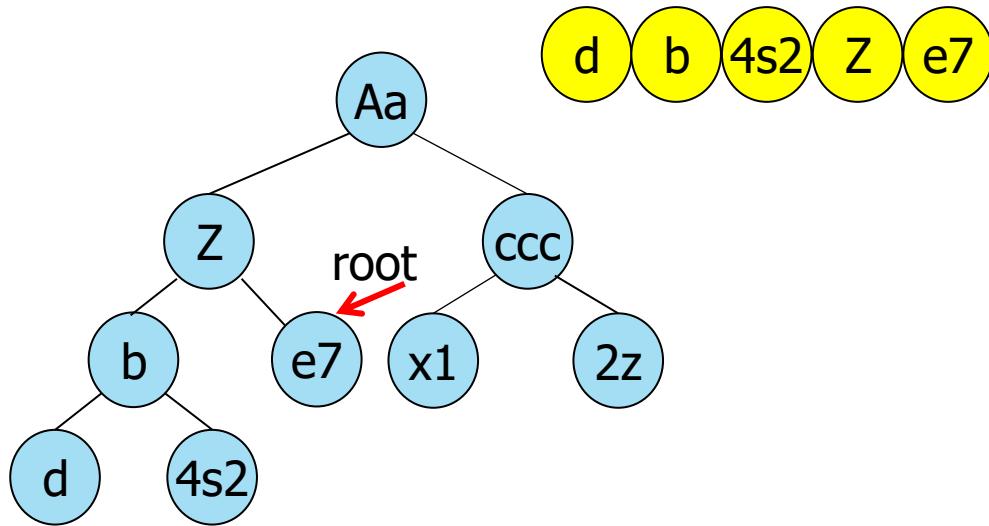


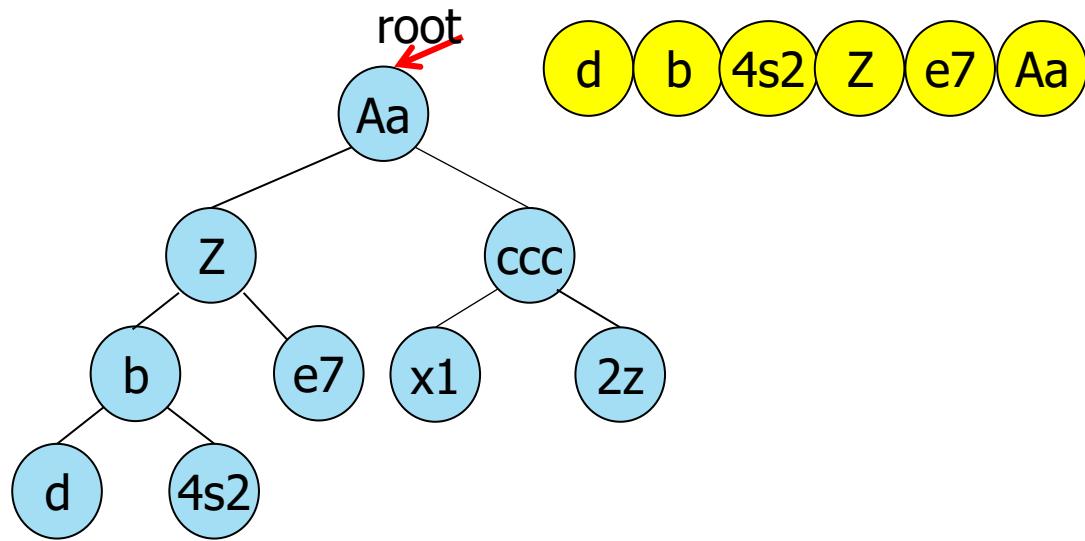


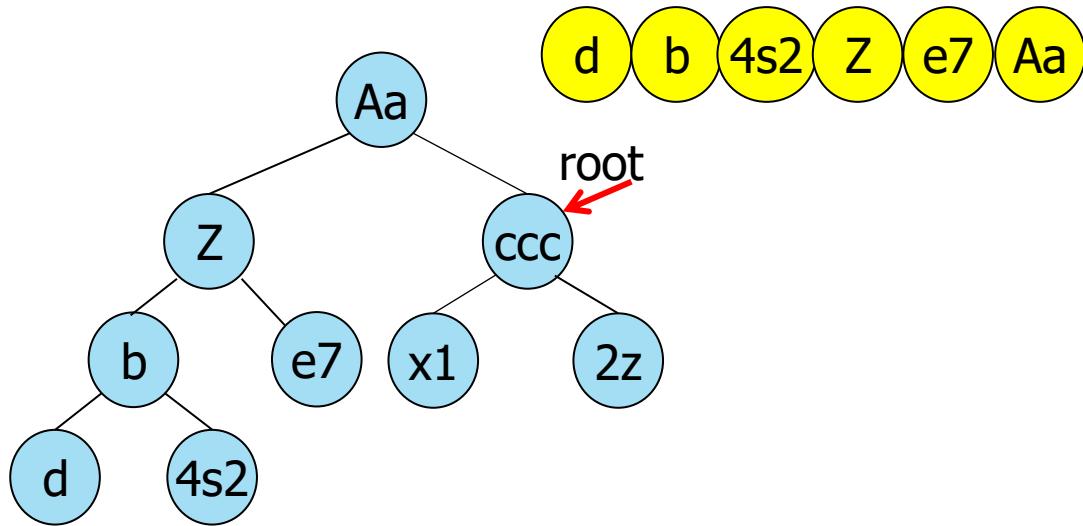
d b 4s2

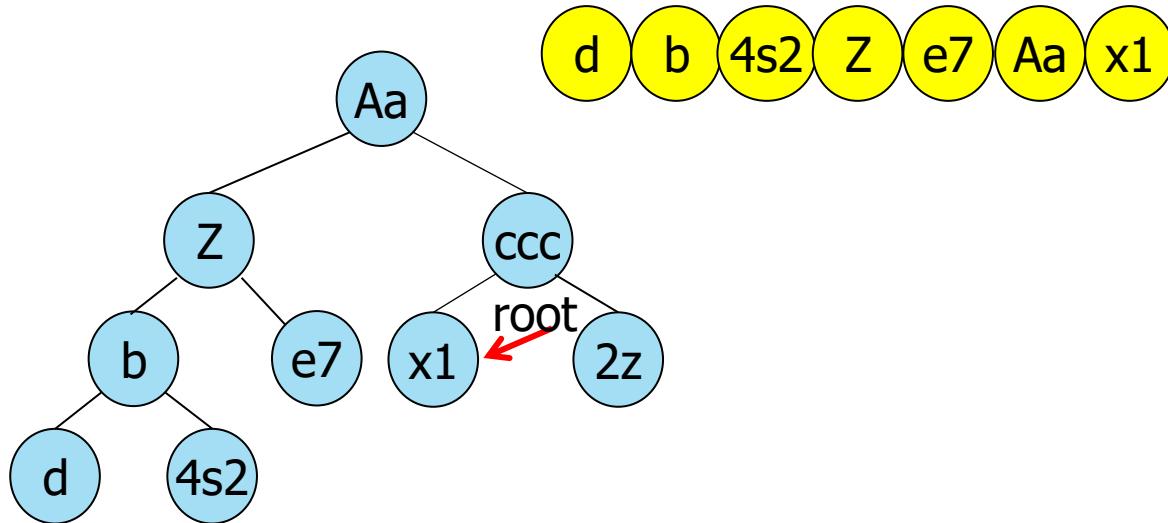


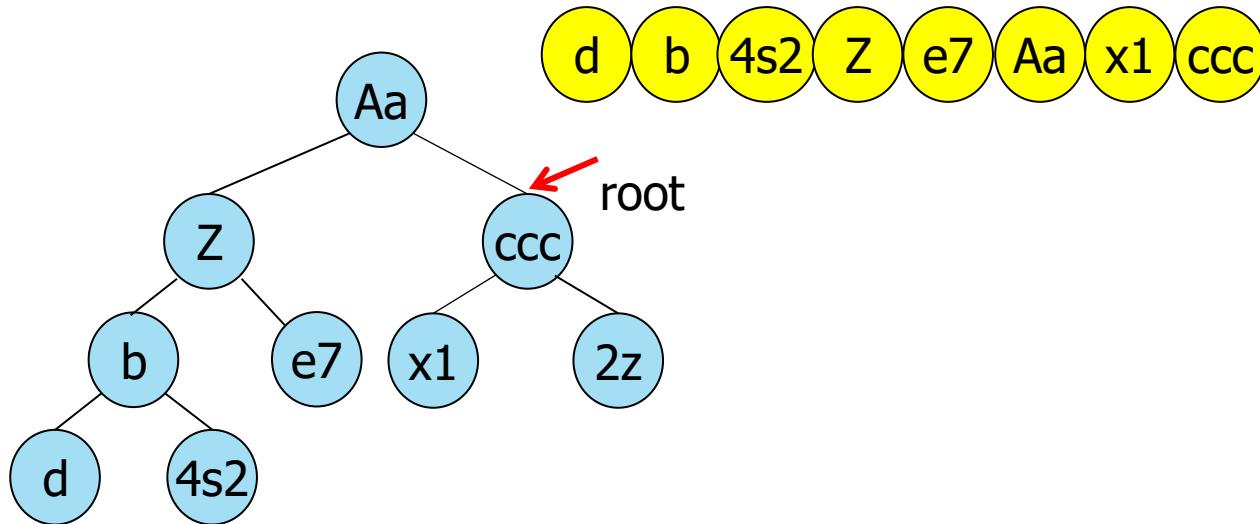
d b 4s2 z

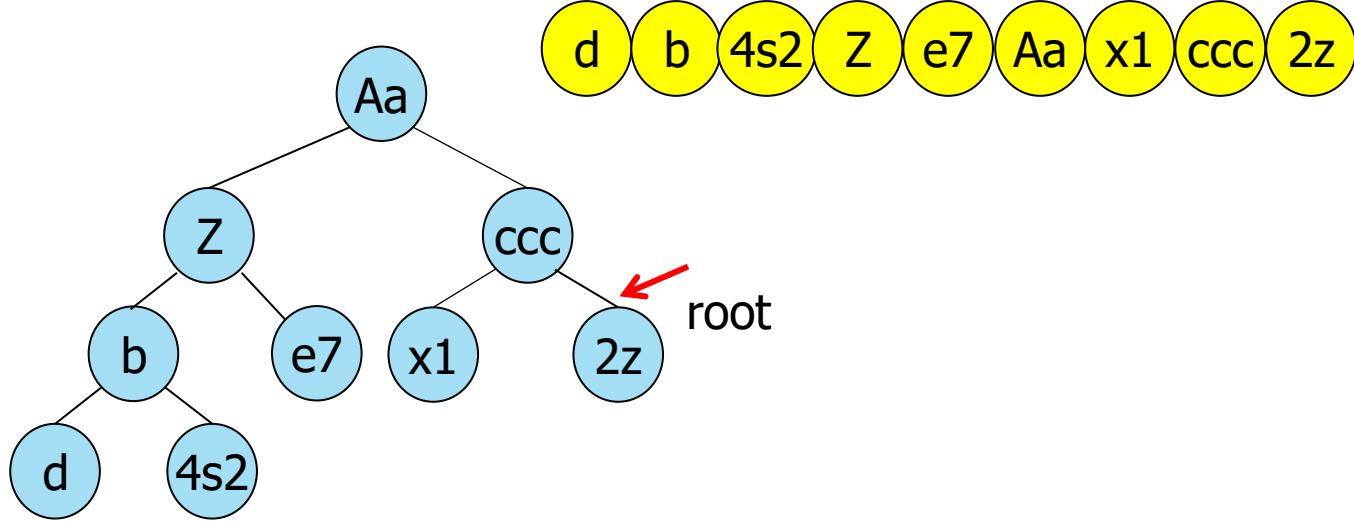




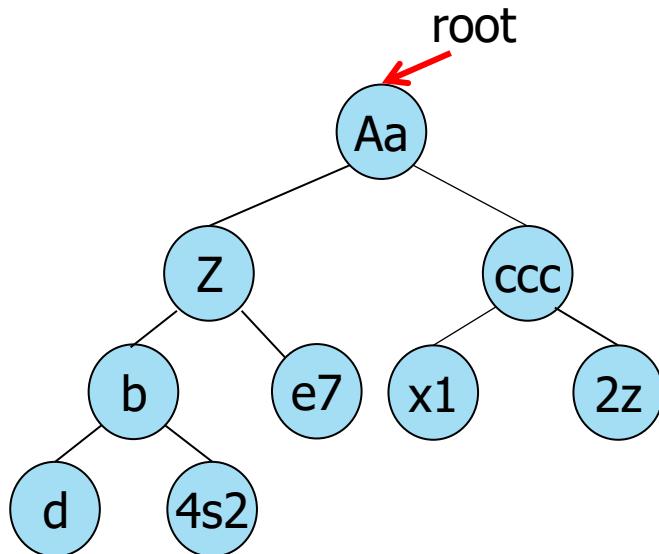




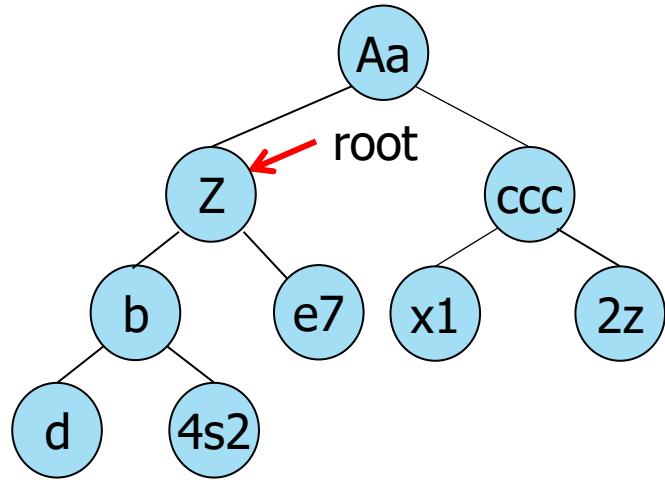


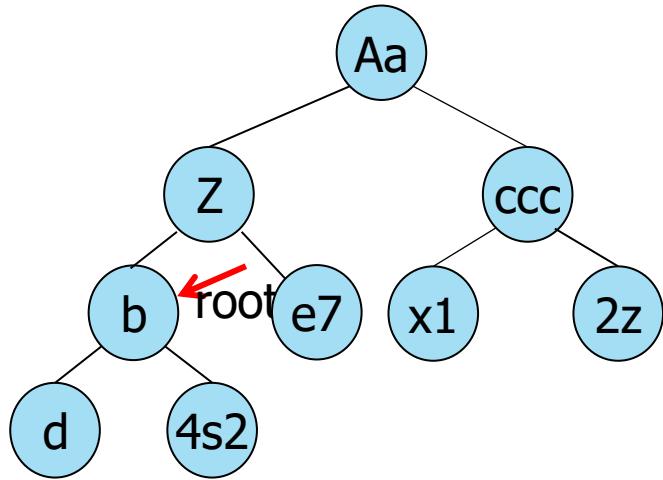


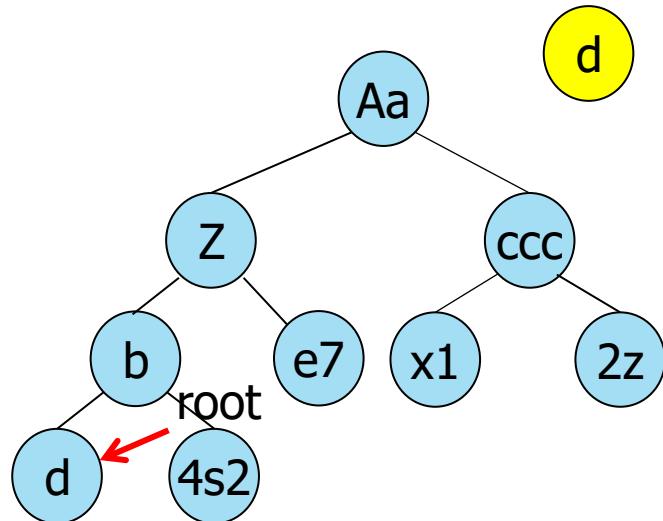
Post-ordine

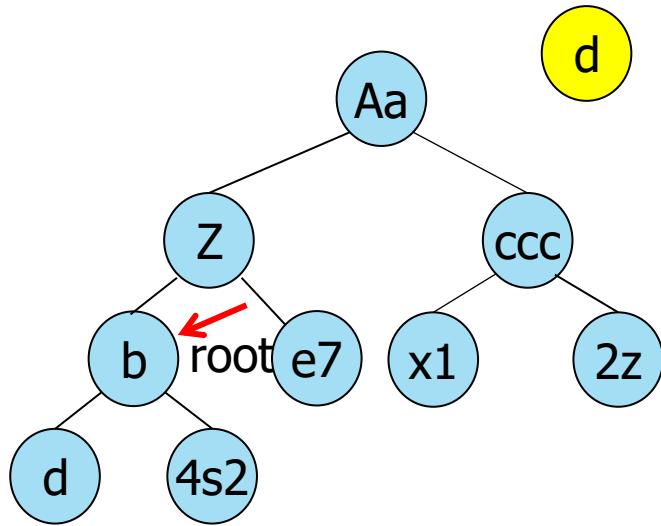


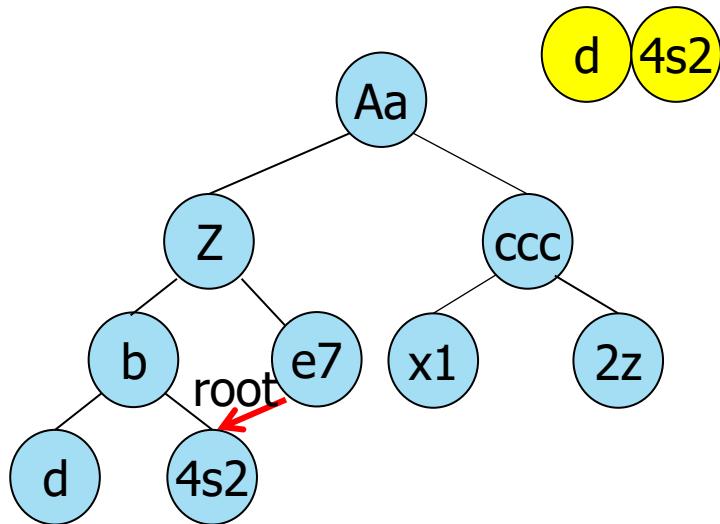
```
void postorder(link root){  
    if (root == NULL)  
        return;  
    postOrder(root->left);  
    postOrder(root->right);  
    printf("%s ", root->name);  
}
```

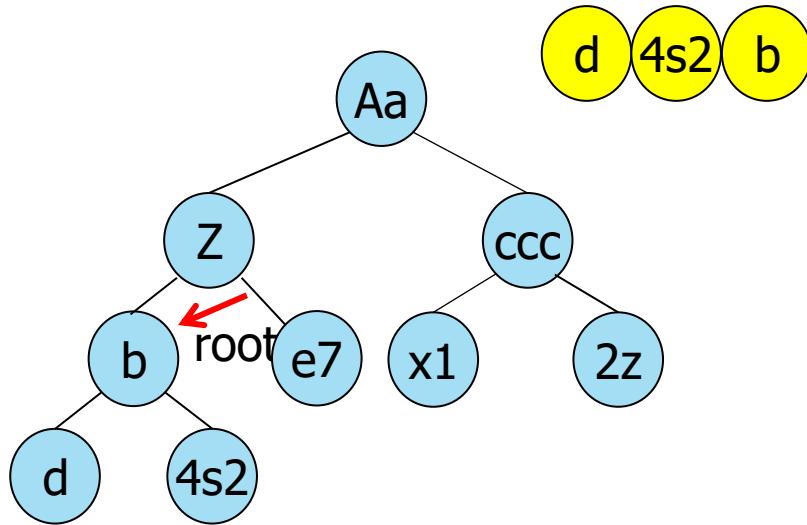


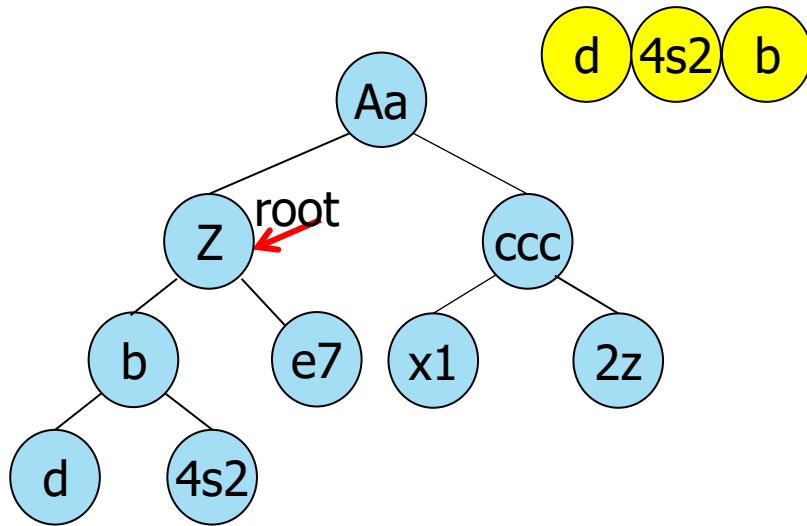


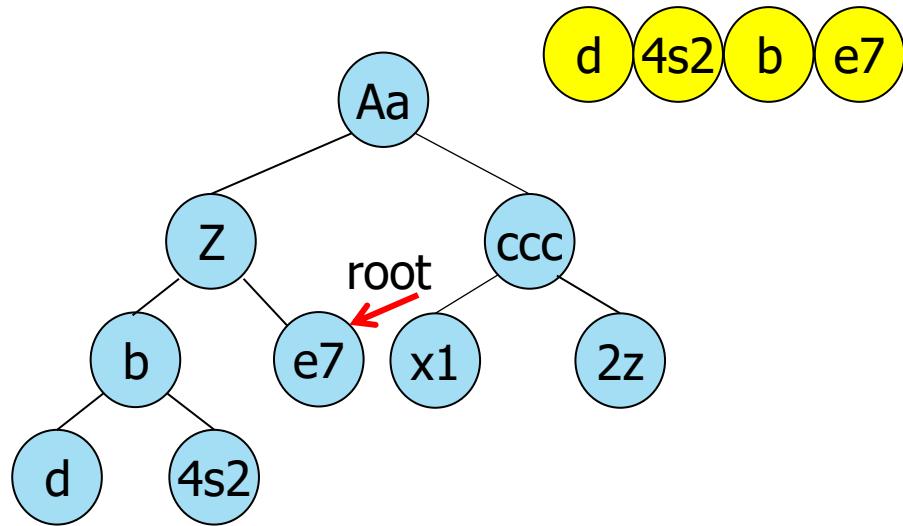


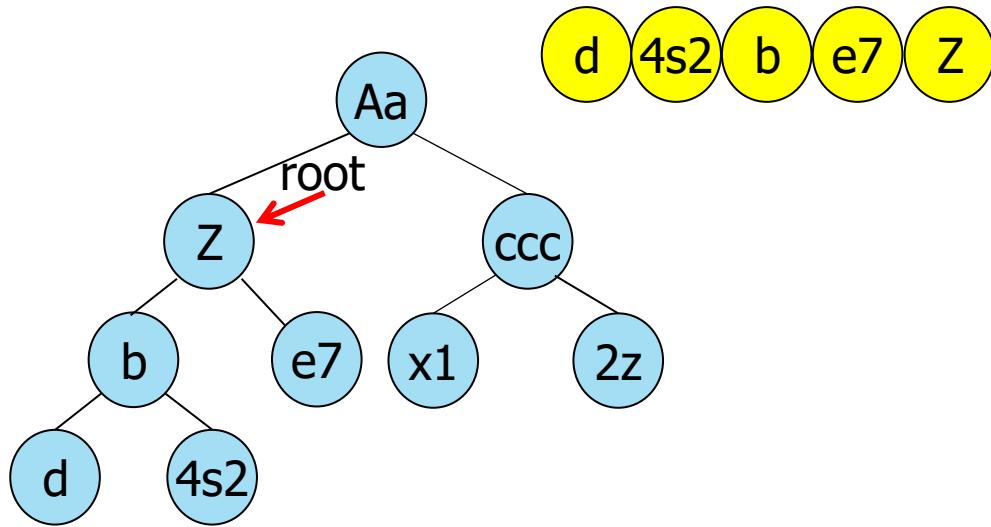


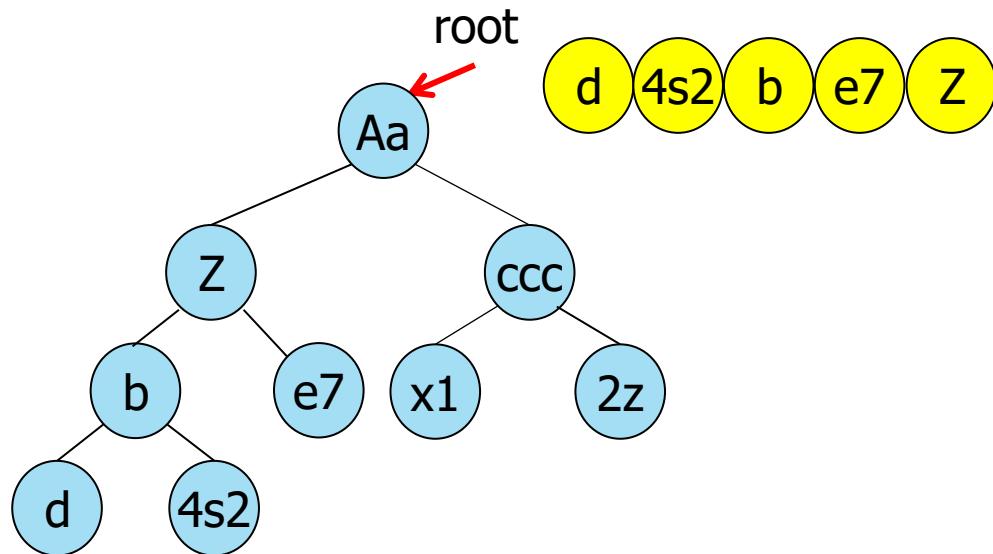


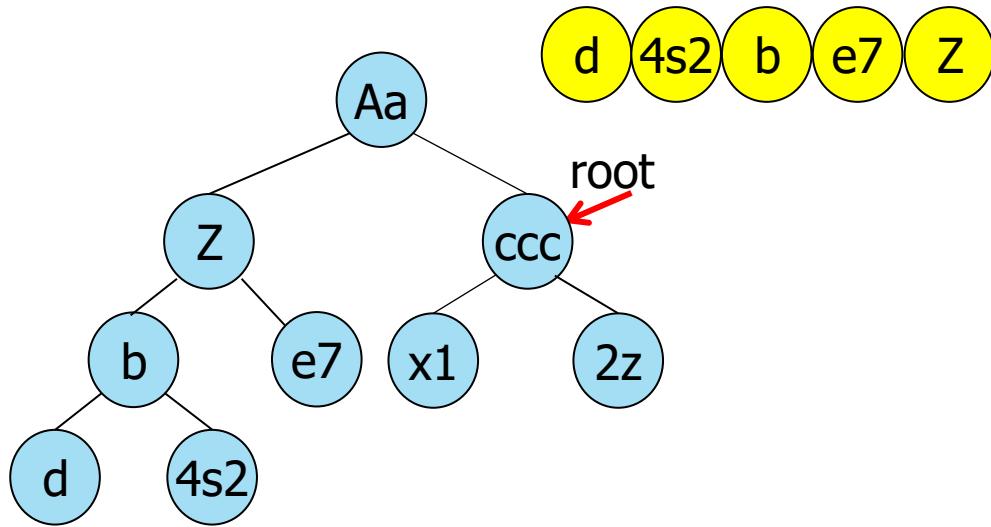






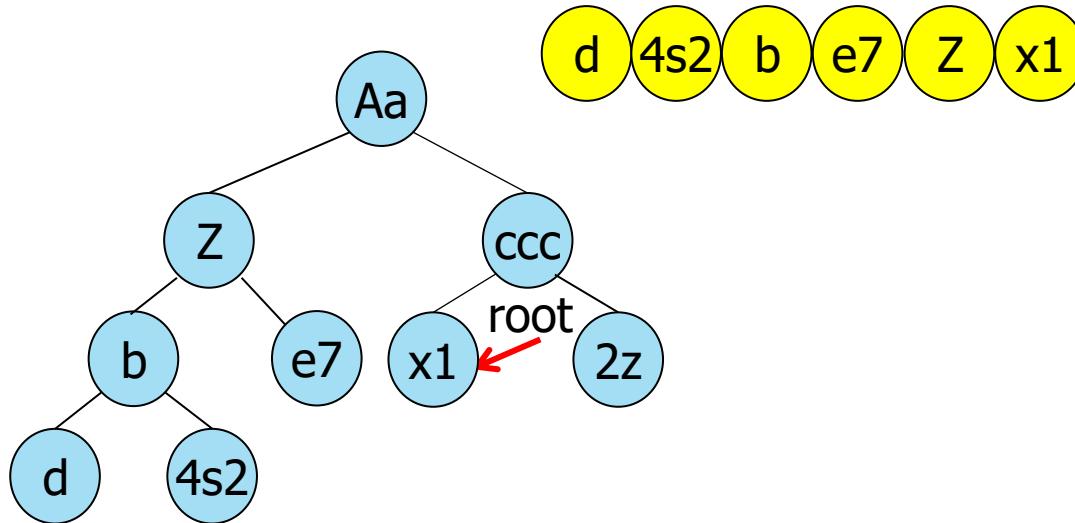


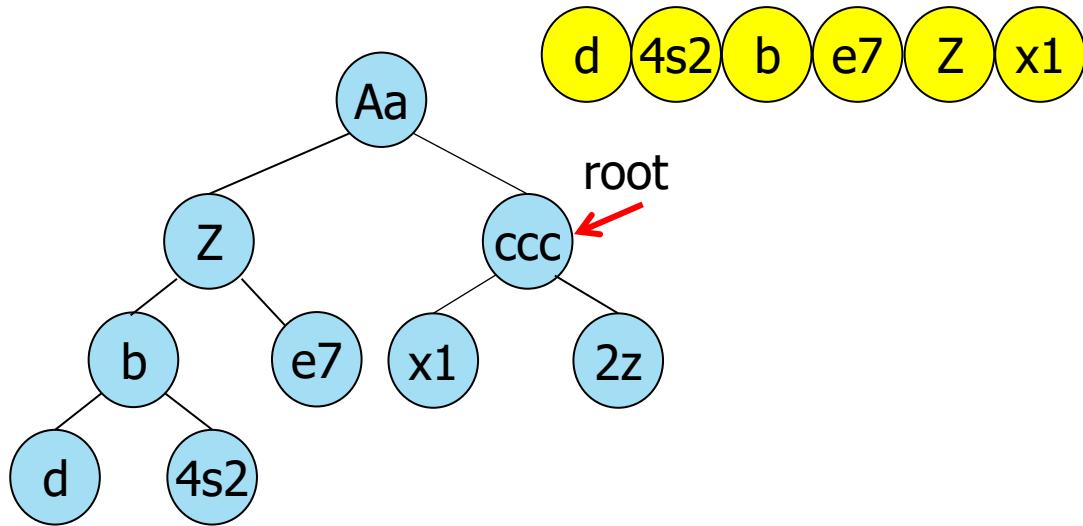


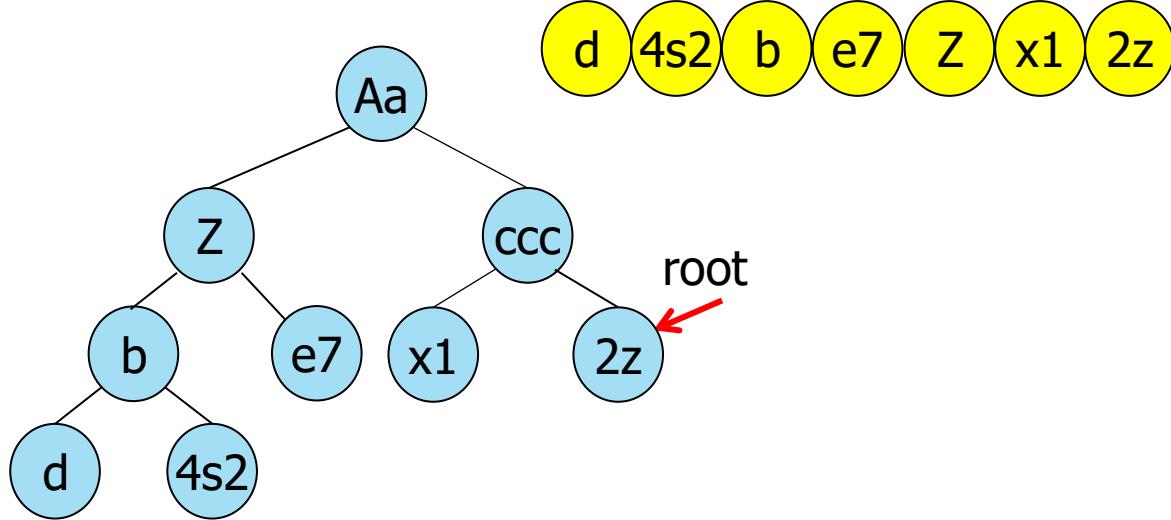


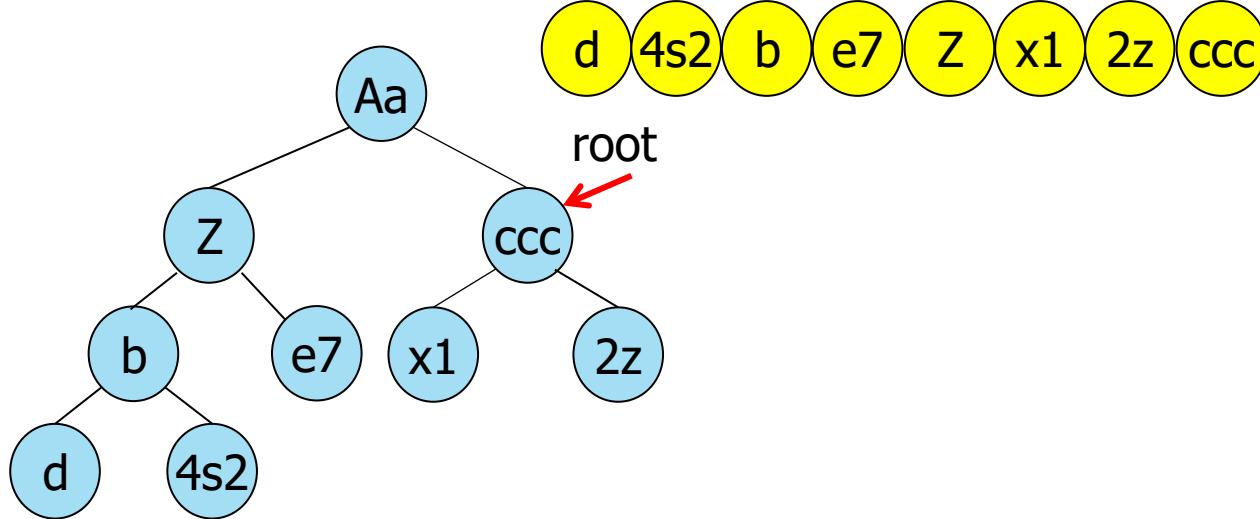
d 4s2 b e7 z

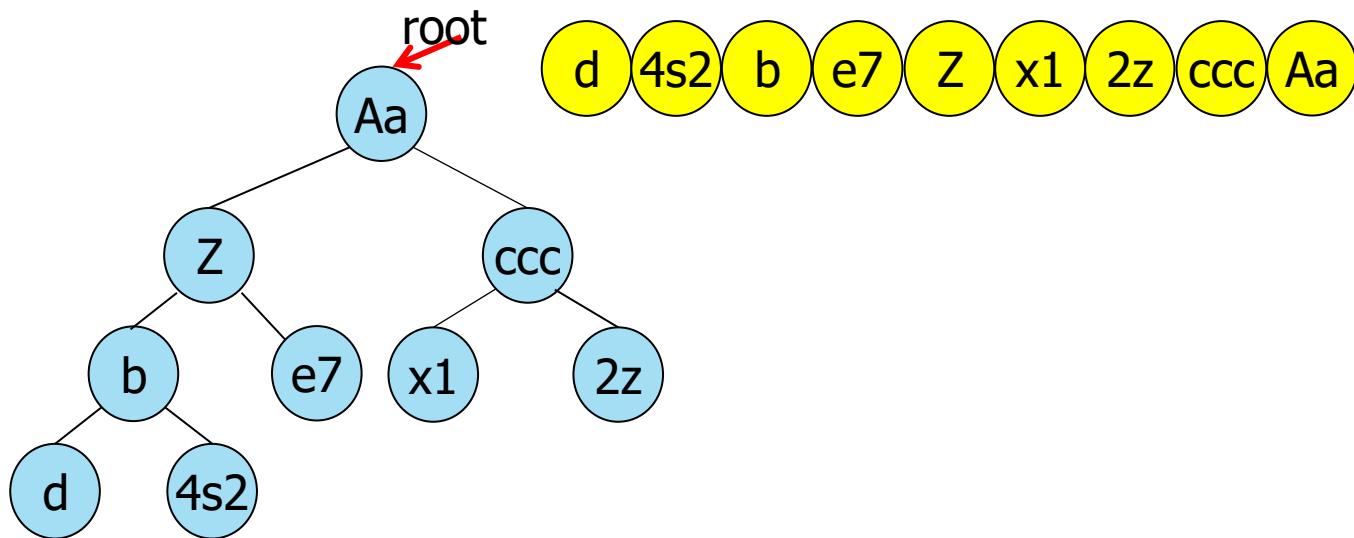
root











Analisi di complessità

Ipotesi 1: albero completo:

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

divide and conquer
 $a = 2$ $b = 2$

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

ESEMPI DI USO

Alberi binari ed espressioni

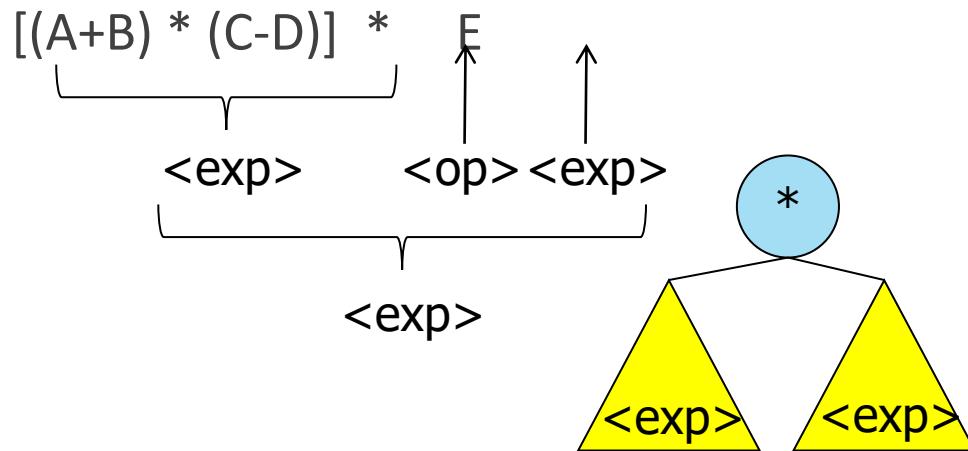
Data un'espressione algebrica in forma infissa (con parentesi per cambiare le precedenze tra operatori), ricostruirne l'albero binario in base alla grammatica semplificata:

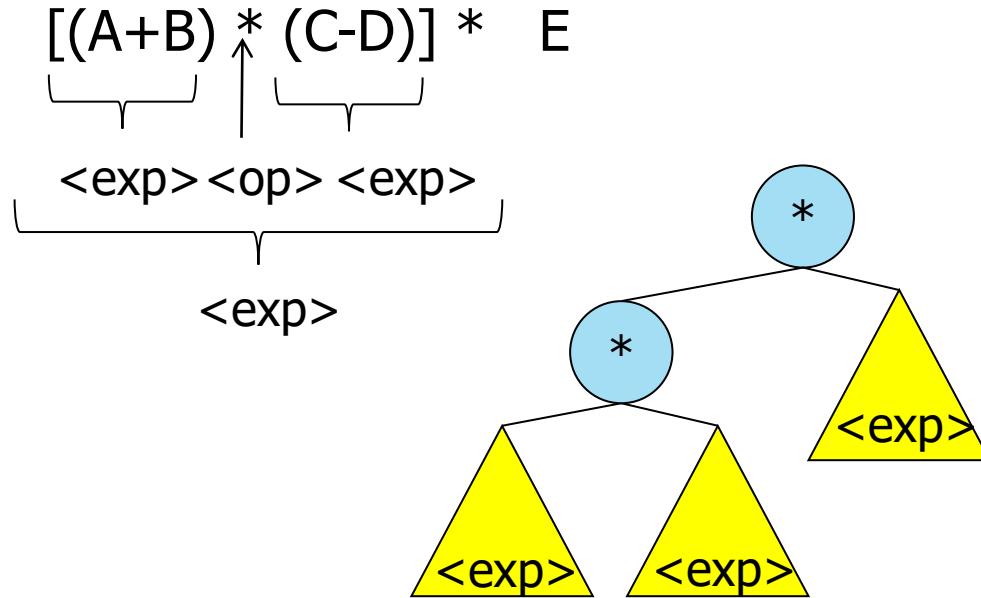
- $\langle \text{exp} \rangle = \langle \text{operand} \rangle \mid \langle \text{exp} \rangle \langle \text{op} \rangle \langle \text{exp} \rangle$
- $\langle \text{operand} \rangle = A \dots Z$
- $\langle \text{op} \rangle = + \mid * \mid - \mid /$

ricorsione

condizione di
terminazione

Dal parsing dell'espressione si ottiene

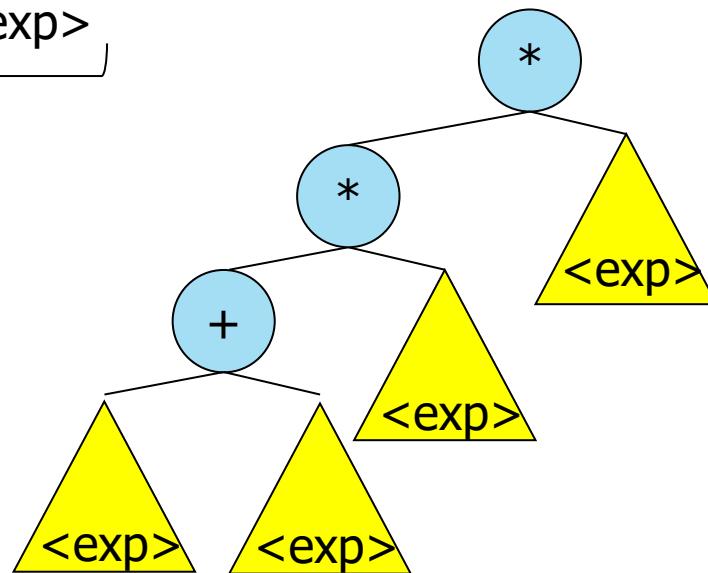


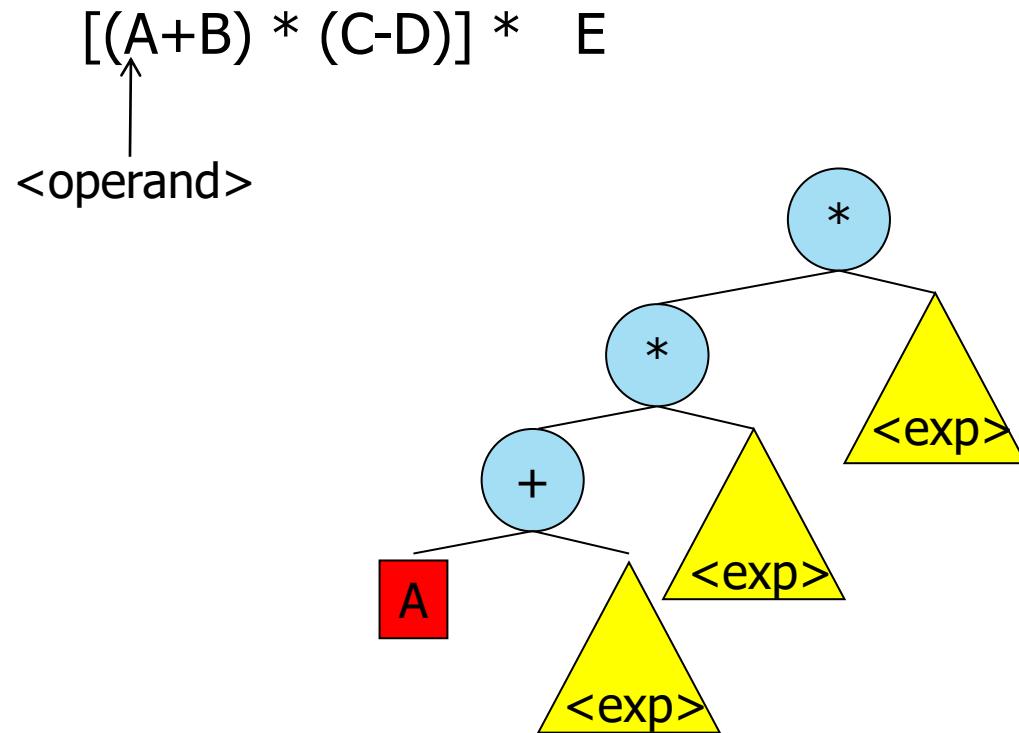


$$[(A+B) * (C-D)] * E$$

$\underbrace{<\text{exp}> <\text{op}> <\text{exp}>}_<\text{exp}>$

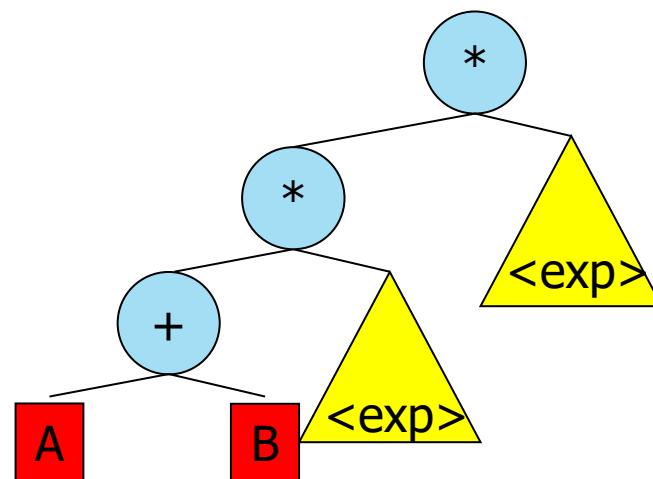
$<\text{exp}>$





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

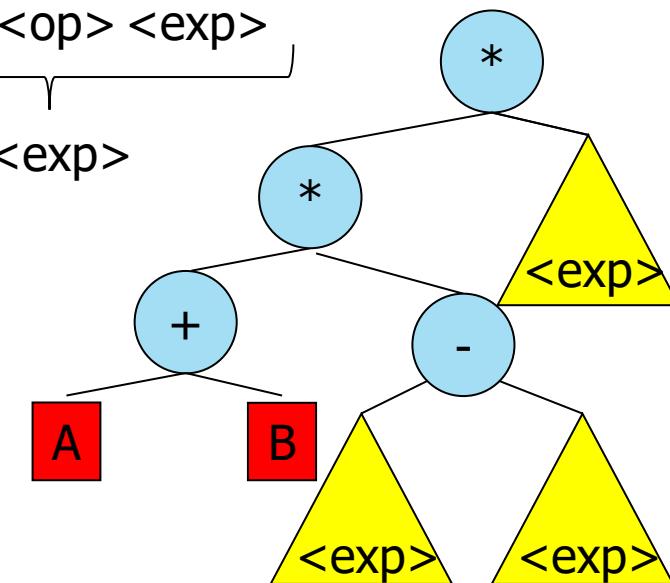
↑
<operand>

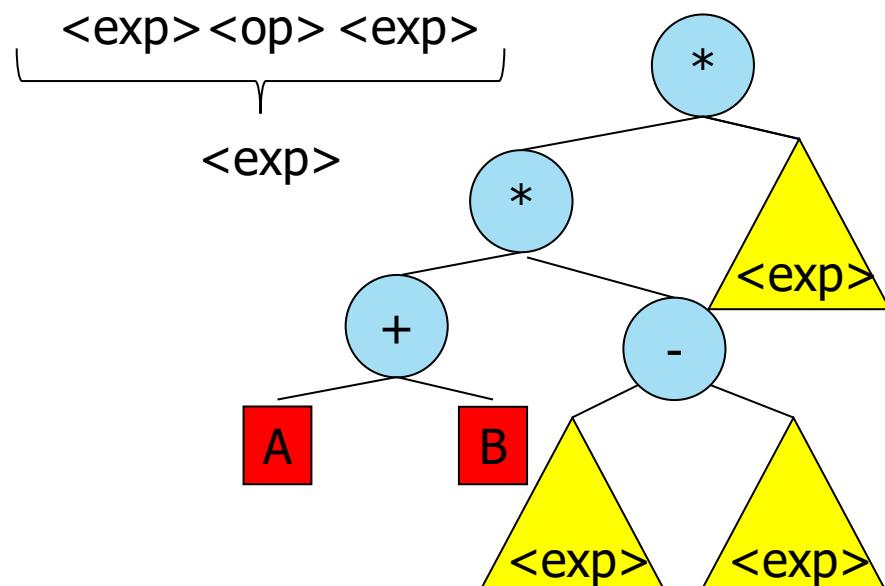


$$[(A+B) * (C-D)] * E$$

$\langle \text{exp} \rangle \langle \text{op} \rangle \langle \text{exp} \rangle$

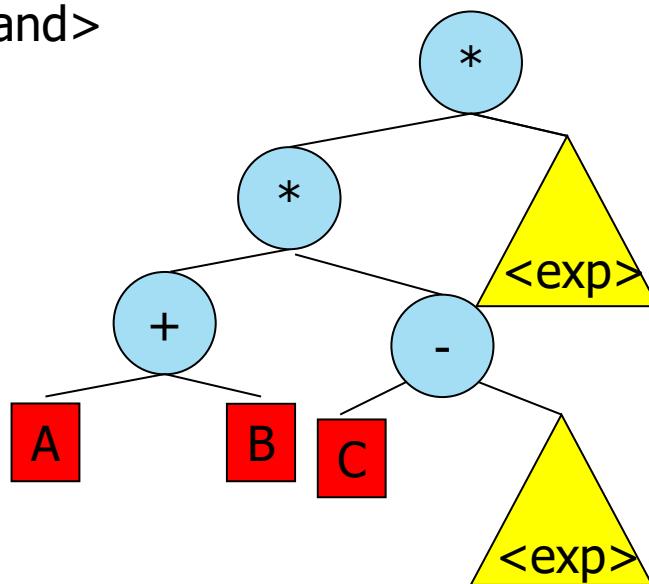
$\langle \text{exp} \rangle$



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


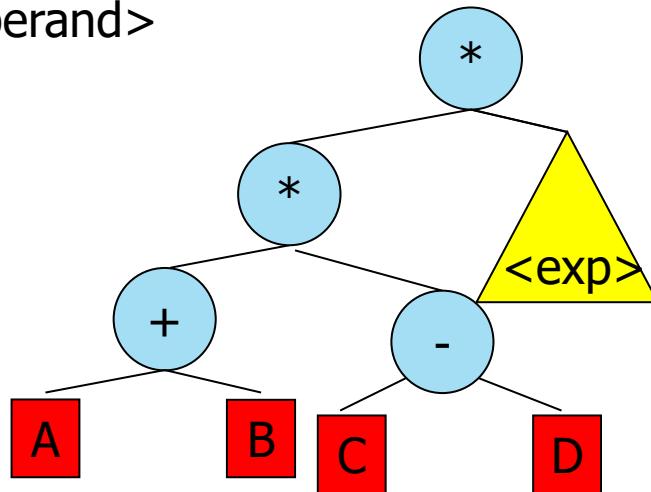
$$[(A+B) * (C-D)] * E$$

<operand>



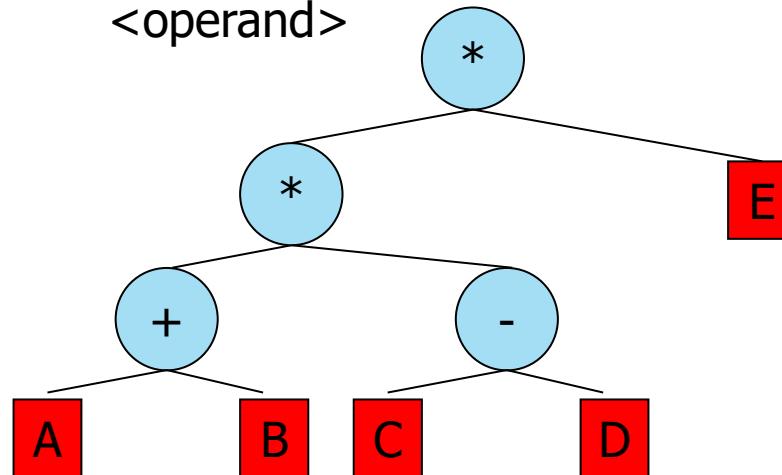
$$[(A+B) * (C-D)] * E$$

↑
<operand>



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

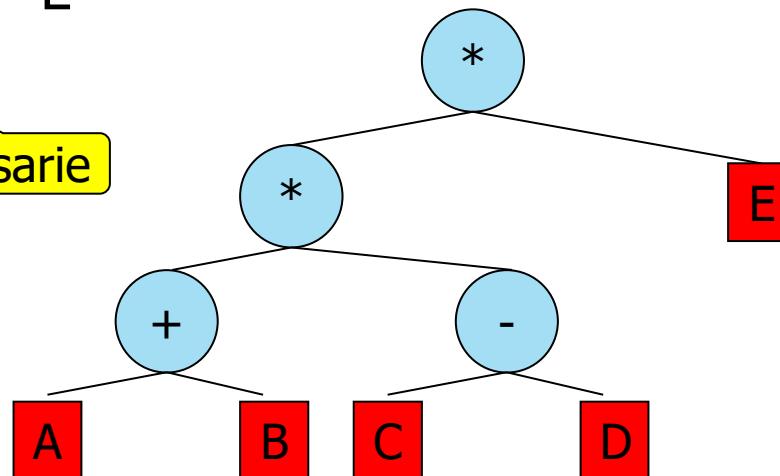
↑
<operand>



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 *

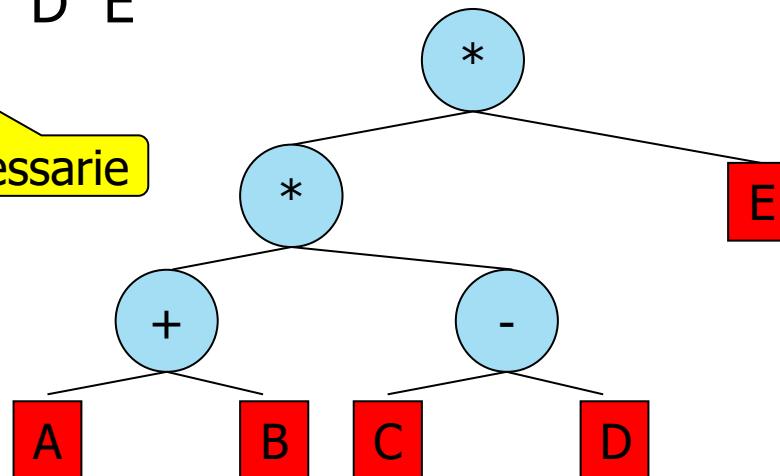
parentesi non più necessarie



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)

- $\langle \text{exp} \rangle = \langle \text{operand} \rangle \mid \langle \text{op} \rangle \langle \text{exp} \rangle \langle \text{exp} \rangle$
- $\langle \text{operand} \rangle = \text{digit} \mid \text{digit catenate operand}$
- $\langle \text{digit} \rangle = 0..9$
- $\langle \text{op} \rangle = + \mid *$

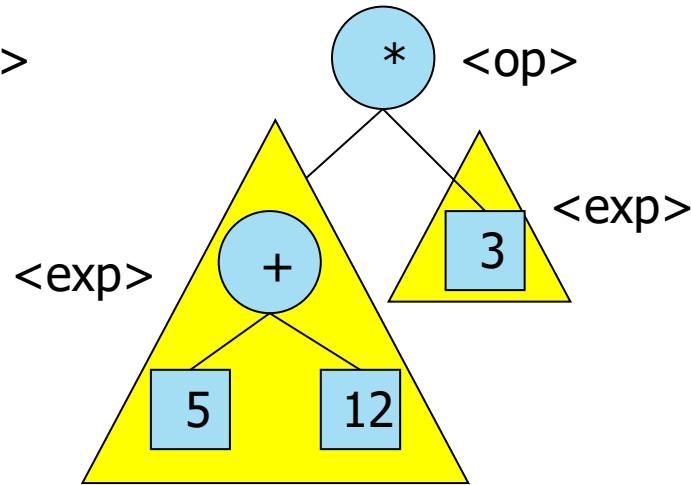
Esempio:

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

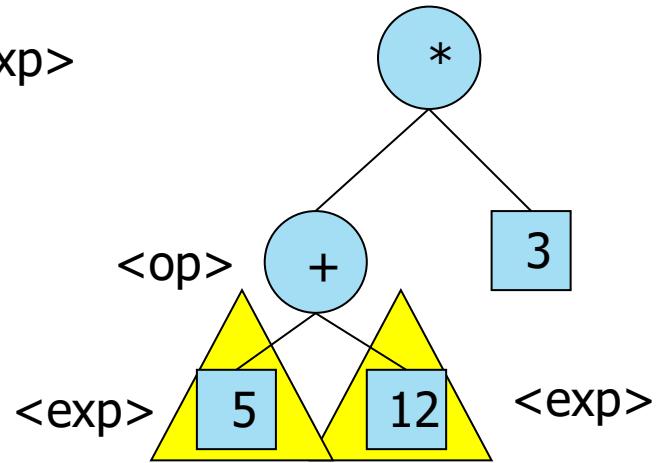
$* + 5 \ 12 \ 3$

$\langle op \rangle$ $\langle exp \rangle$

$\langle exp \rangle$



+ 5 12 <exp>
<op> <exp>

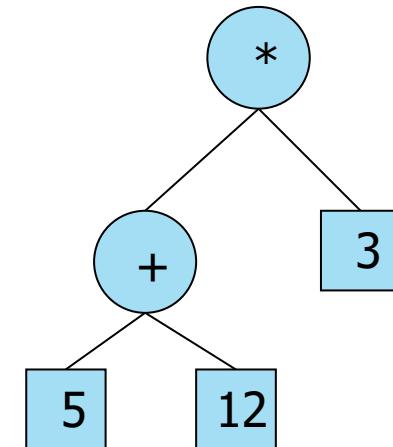


Espressione prefissa memorizzata in vettore a di caratteri (spazi per separare):

a	*	+	5		1	2		3
---	---	---	---	--	---	---	--	---

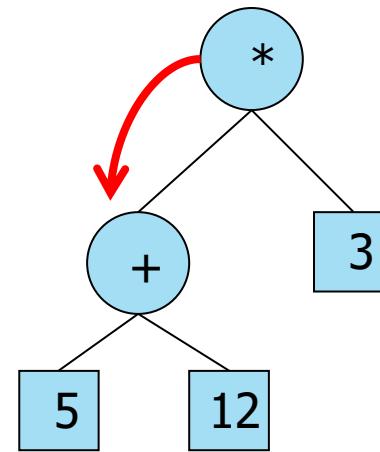
Valutazione: scansione

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

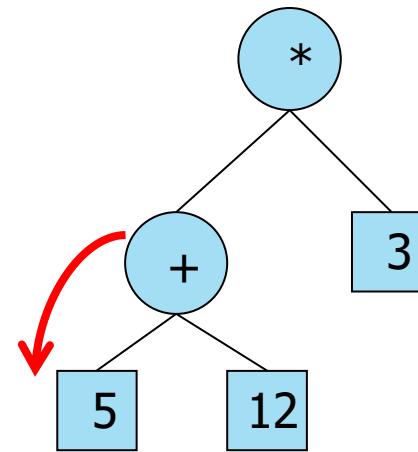
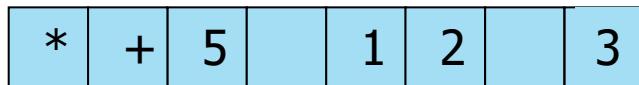


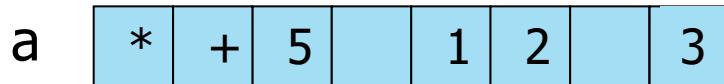
a

*	+	5		1	2		3
---	---	---	--	---	---	--	---

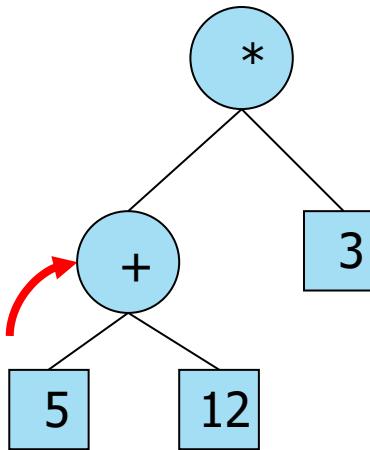


a

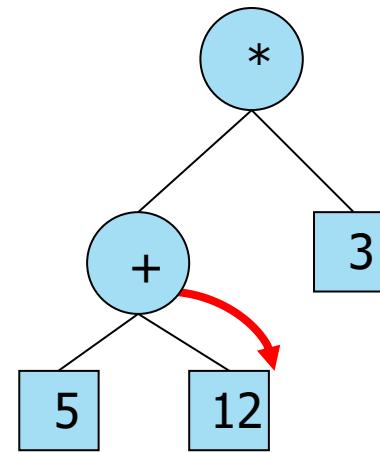
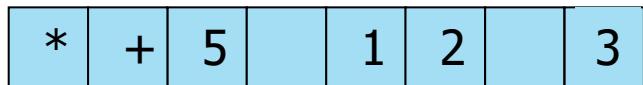




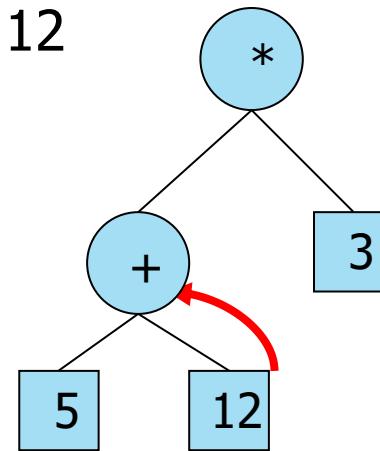
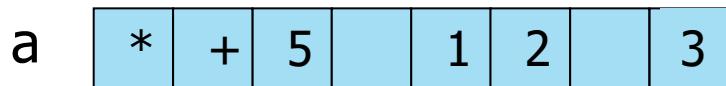
return 5



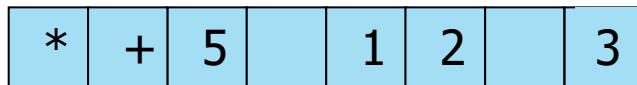
a



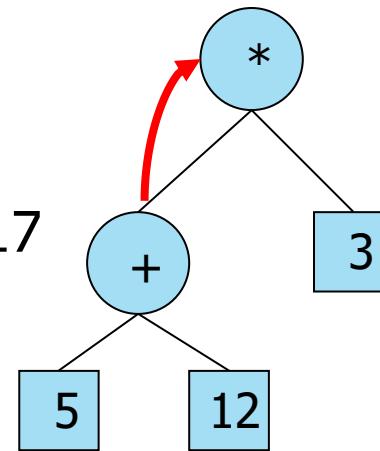
return 12



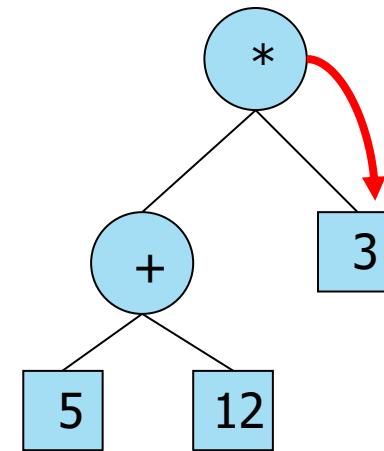
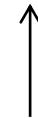
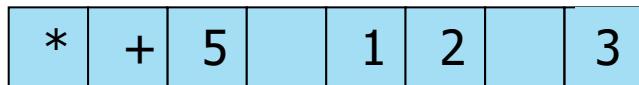
a



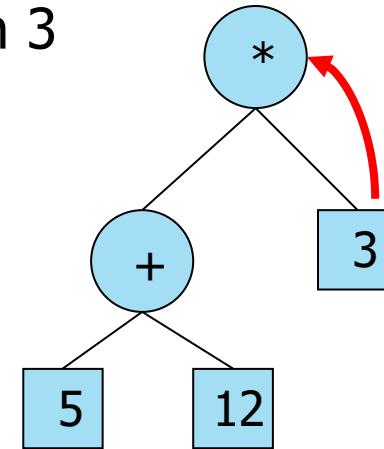
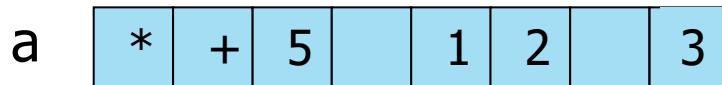
return 17



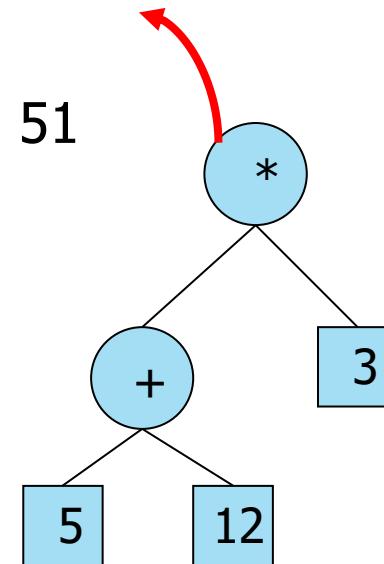
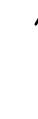
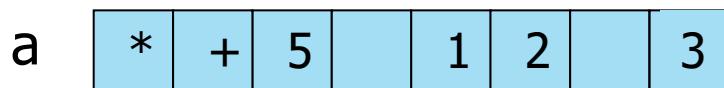
a



return 3



return 51



Nel main si dichiarano 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;  
}
```

```
static char *a;  
static int i;
```

Alberi Binari di Ricerca

BST: DEFINIZIONE E IMPLEMENTAZIONE COME ADT

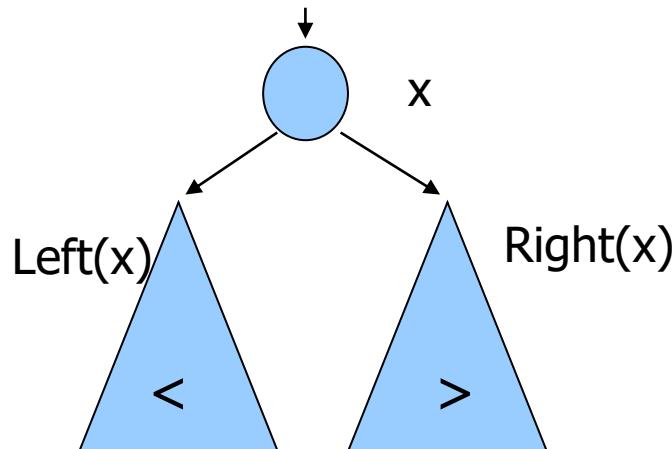
Alberi binari di ricerca (BST)

ADT albero binario con proprietà:

\forall nodo x vale che:

- \forall nodo $y \in \text{Left}(x)$, $\text{key}[y] < \text{key}[x]$
- \forall nodo $y \in \text{Right}(x)$, $\text{key}[y] > \text{key}[x]$

chiavi distinte!



ADT di classe BST

BST.h

```
typedef struct binarysearchtree *BST;

BST    BSTinit() ;
void   BSTfree(BST bst);
int    BSTcount(BST bst);
int    BSTempty(BST bst);
Item   BSTsearch(BST bst, Key k);
void   BSTinsert_leafI(BST bst, Item x);
void   BSTinsert_leafR(BST bst, Item x);
void   BSTinsert_root(BST bst, Item x);
Item   BSTmin(BST bst);
Item   BSTmax(BST bst);
void   BSTvisit(BST bst, int strategy);
```

BST.c

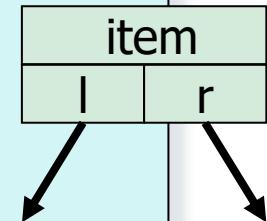
```
#include <stdlib.h>
#include "Item.h"
#include "BST.h"

typedef struct BSTnode* link;
struct BSTnode { Item item; link l; link r; };
struct binarysearchtree { link root; link z; };

static link NEW(Item item, link l, link r) {
    link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r;
    return x;
}

BST BSTinit( ) {
    BST bst = malloc(sizeof *bst) ;
    bst->root= ( bst->z = NEW(ITEMsetNull(), NULL, NULL));
    return bst;
}
```

BSTnode



nodo sentinella

```
void BSTfree(BST bst) {
    if (bst == NULL)
        return;
    treeFree(bst->root, bst->z);
    free(bst->z);
    free(bst);
}

static void treeFree(link h, link z) {
    if (h == z)
        return;
    treeFree(h->l, z);
    treeFree(h->r, z);
    free(h);
}
```

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

int BSTcount(BST bst) {
    return countR(bst->root, bst->z);
}

int BSTempty(BST bst) {
    if (BSTcount(bst) == 0)
        return 1;
    return 0;
}
```

BSTsearch

Ricerca ricorsiva di un nodo che contiene un item con una chiave data:

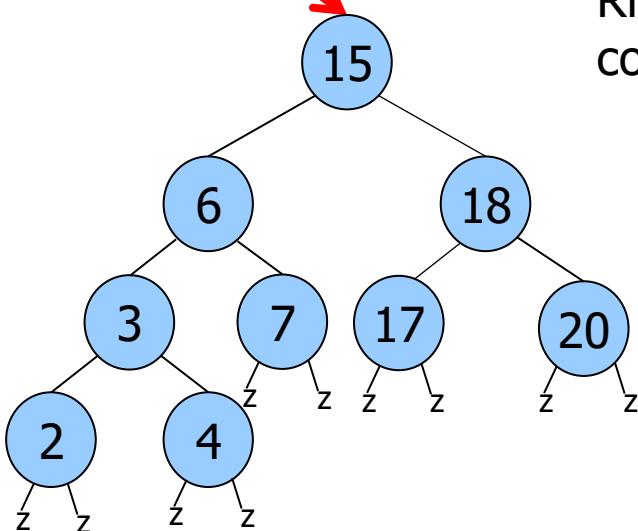
- percorrimento dell'albero dalla radice
- terminazione: la chiave dell'item cercato è uguale alla chiave del nodo corrente (search hit) oppure si è giunti ad un albero vuoto (search miss)
- ricorsione: dal nodo corrente
 - su sottoalbero sinistro se la chiave dell'item cercato è < della chiave del nodo corrente
 - su sottoalbero destro altrimenti

```
Item searchR(link h, Key k, link z) {
    int cmp;
    if (h == z)
        return ITEMsetNull();
    cmp = KEYcmp(k, KEYget(h->item));
    if (cmp == 0)
        return h->item;
    if (cmp == -1)
        return searchR(h->l, k, z);
    return searchR(h->r, k, z);
}
```

```
Item BSTsearch(BST bst, Key k) {
    return searchR(bst->root, k, bst->z);
}
```

Esempio

$h = bst->root$

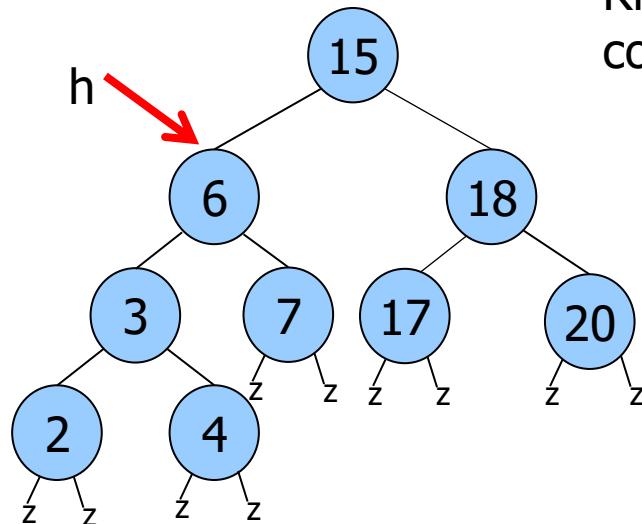


Ricerca dell'item
con chiave 7

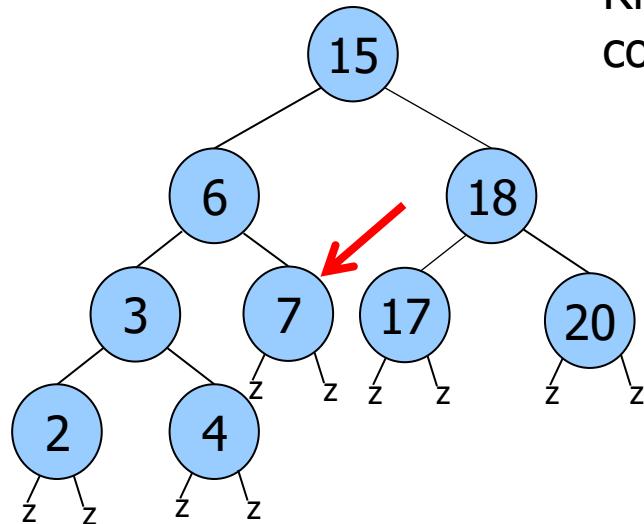
Sono visualizzate solo le
chiavi intere, non gli item

z: nodo sentinella

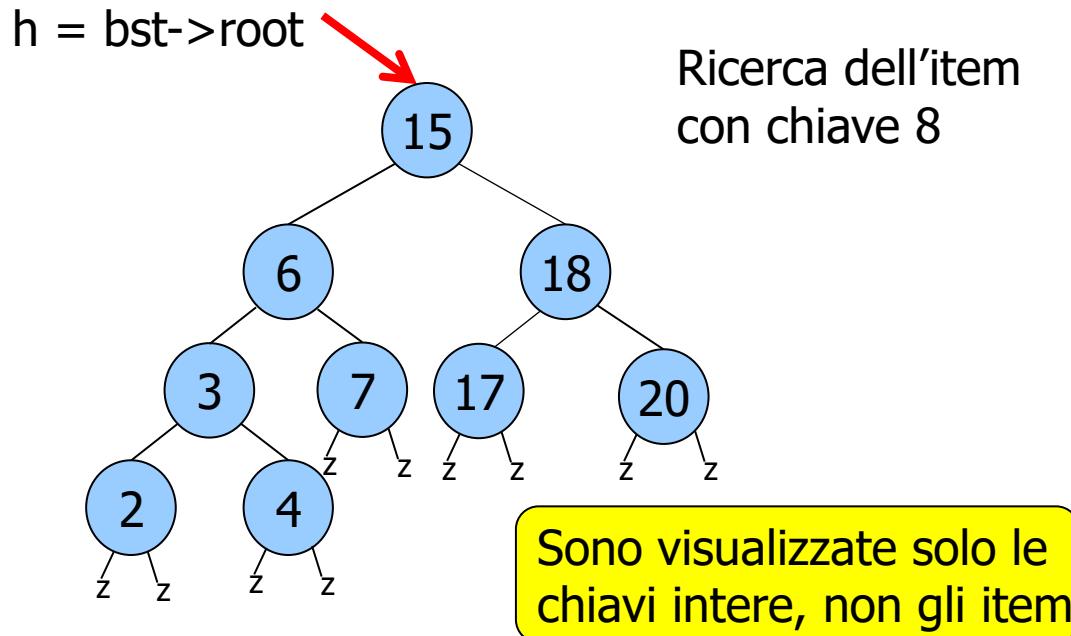
Ricerca dell'item
con chiave 7



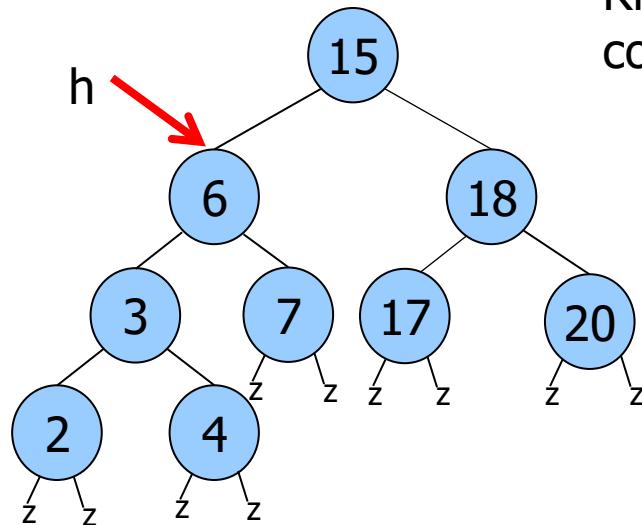
Ricerca dell'item
con chiave 7
Hit!



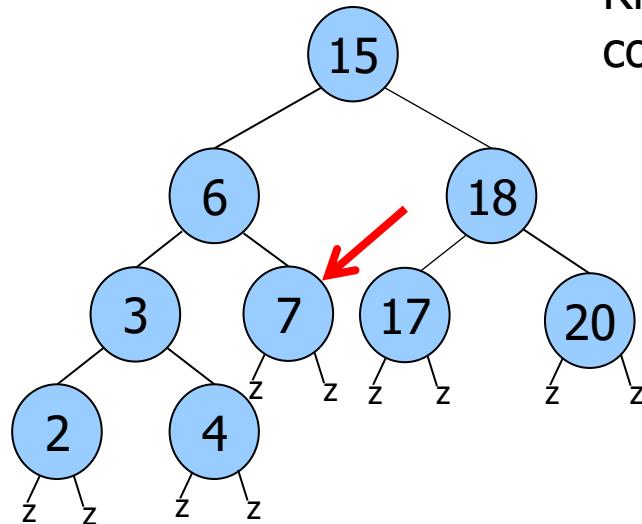
Esempio



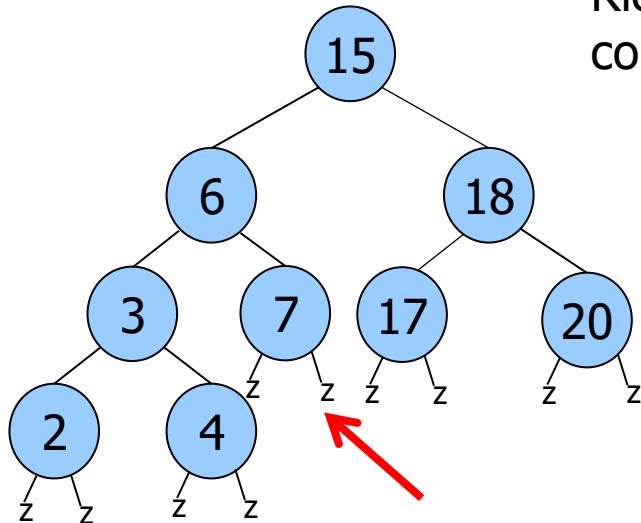
Ricerca dell'item
con chiave 8



Ricerca dell'item
con chiave 8



Ricerca dell'item
con chiave 8
Miss!



BSTmin

```
Item minR(link h, link z) {
    if (h == z)
        return ITEMsetNull();
    if (h->l == z)
        return (h->item);
    return minR(h->l, z);
}

Item BSTmin(BST bst) {
    return minR(bst->root, bst->z);
}
```

Seguire il puntatore al sottoalbero sinistro finché si arriva al nodo sentinella z

BSTmax

```
Item maxR(link h, link z) {
    if (h == z)
        return ITEMsetNull();
    if (h->r == z)
        return (h->item);
    return maxR(h->r, z);
}

Item BSTmax(BST bst) {
    return maxR(bst->root, bst->z);
}
```

Seguire il puntatore al sottoalbero destro finché si arriva al nodo sentinella z

BSTinsert (in foglia)

Inserire in un albero binario di ricerca un nodo che contiene un item \Rightarrow mantenimento della proprietà:

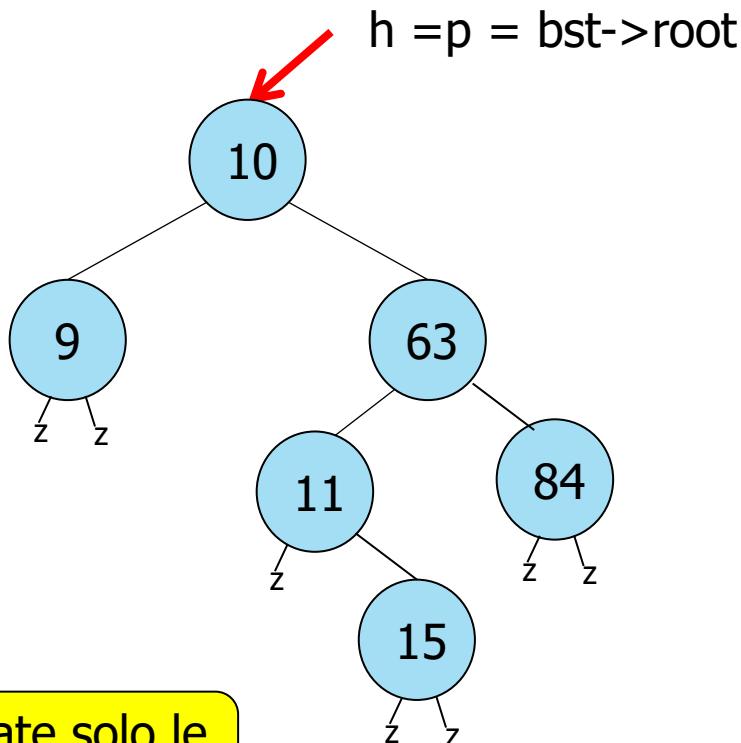
- se il BST è vuoto, creazione del nuovo albero
- inserimento **ricorsivo** nel sottoalbero sinistro o destro a seconda del confronto tra la chiave dell'item e quella del nodo corrente
- inserimento **iterativo**: prima si ricerca la posizione, poi si appende il nuovo nodo.

```
static link insertR(link h, Item x, link z) {
    if (h == z)
        return NEW(x, z, z);
    if (KEYcmp(KEYget(x), KEYget(h->item)) == -1)
        h->l = insertR(h->l, x, z);
    else
        h->r = insertR(h->r, x, z);
    return h;
}

void BSTinsert_leafR(BST bst, Item x) {
    bst->root = insertR(bst->root, x, bst->z);
}
```

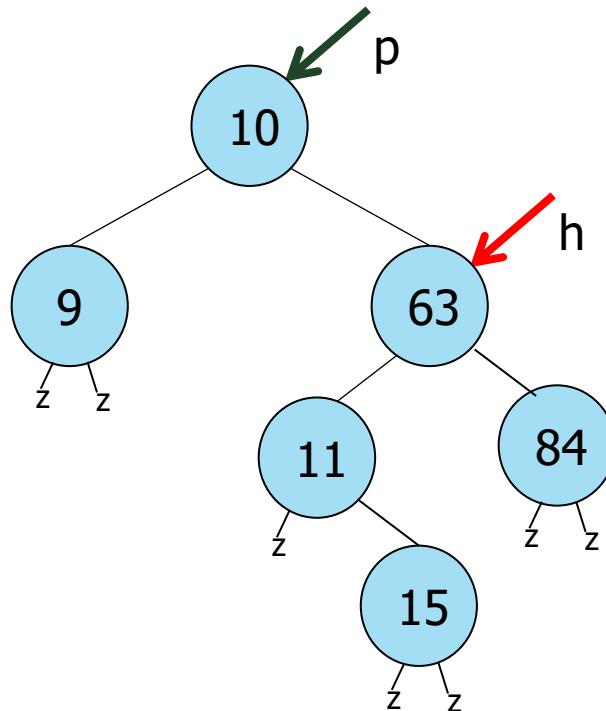
```
void BSTinsert_leafI(BST bst, Item x) {
    link p = bst->root, h = p;
    if (bst->root == bst->z) {
        bst->root = NEW(x, bst->z, bst->z);
        return;
    }
    while (h != bst->z) {
        p = h;
        h=(KEYcmp(KEYget(x), KEYget(h->item))==-1) ? h->l : h->r;
    }
    h = NEW(x, bst->z, bst->z);
    if (KEYcmp(KEYget(x), KEYget(p->item))==-1)
        p->l = h;
    else
        p->r = h;
}
```

Esempio

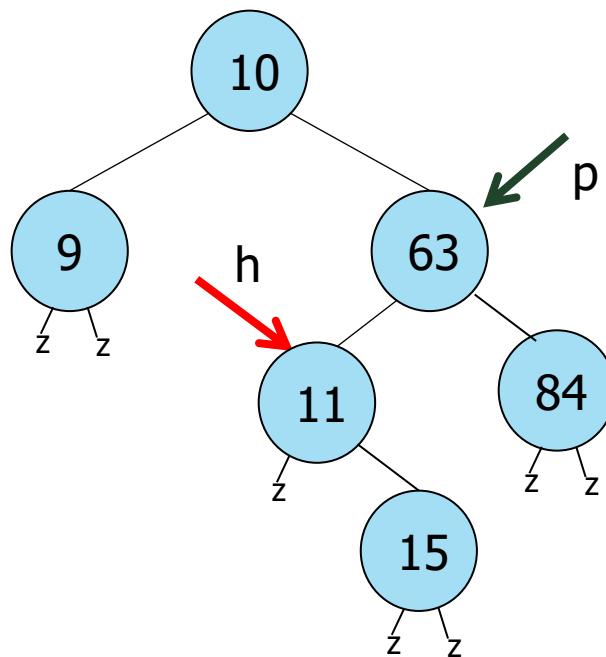


inserimento dell'item con
chiave 62

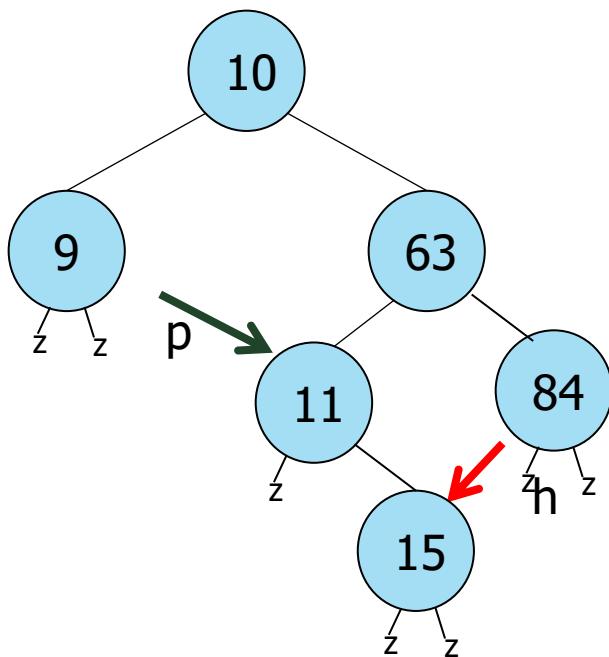
Sono visualizzate solo le
chiavi intere, non gli item



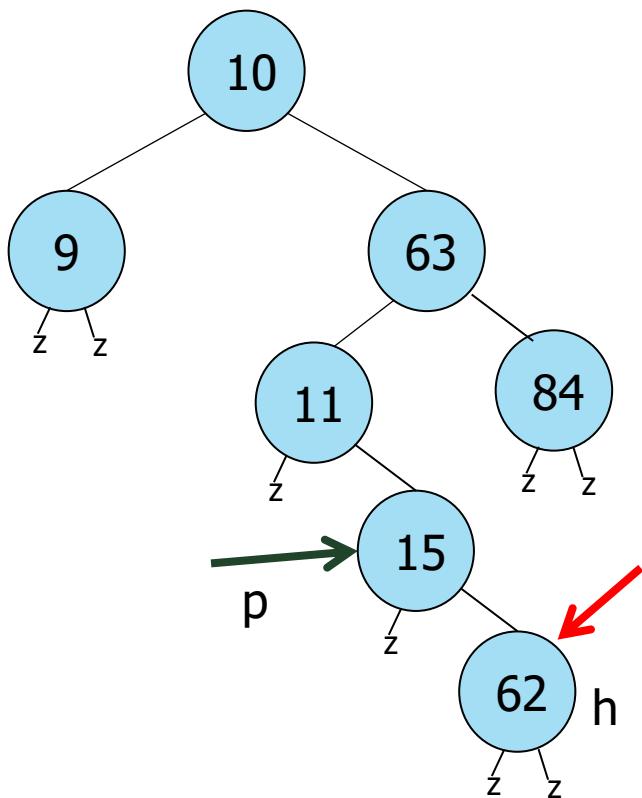
inserimento dell'item con
chiave 62



inserimento dell'item con
chiave 62



inserimento dell'item con
chiave 62



inserimento dell'item con
chiave 62

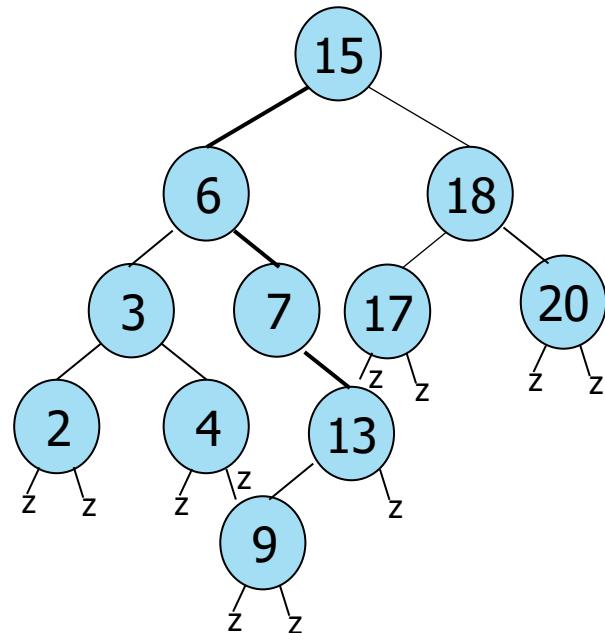
Complessità

Le operazioni hanno complessità $T(n) = O(h)$:

- albero con n nodi completamente bilanciato
 - altezza $h = \alpha(\log_2 n)$
- albero con n nodi completamente sbilanciato ha
 - altezza $h = \alpha(n)$
- $O(\log n) \leq T(n) \leq O(n)$

BSTvisit

Attraversamento in-ordine: **ordinamento crescente delle chiavi.**



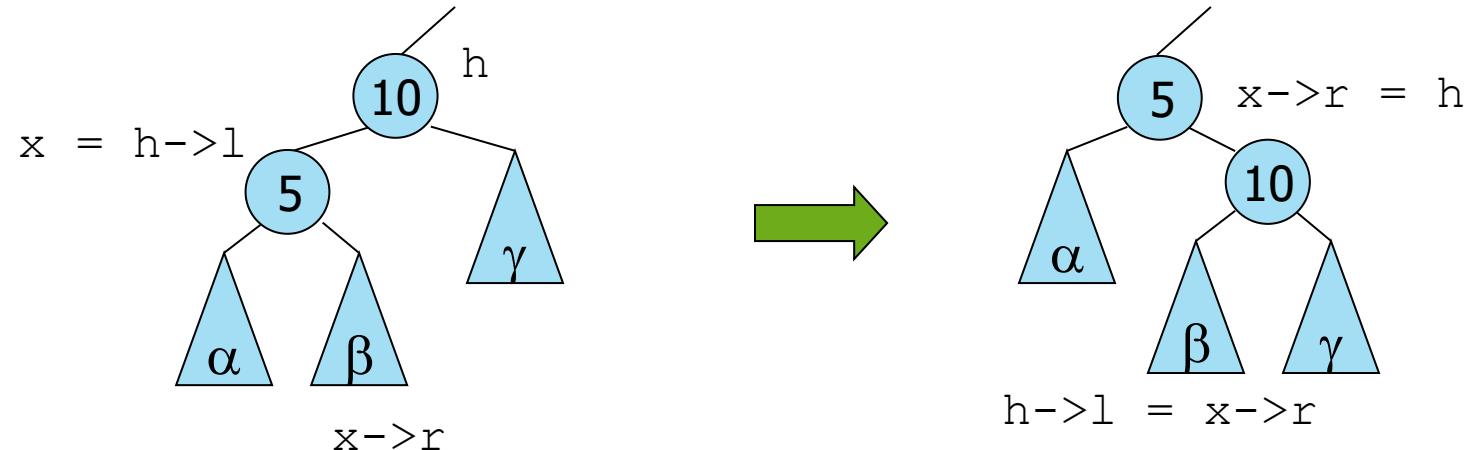
2 3 4 6 7 9 13 15 17 18 20



La chiave **mediana** (inferiore) di un insieme di n elementi è l'elemento che si trova in posizione $\lfloor(n + 1)/2\rfloor$ nella sequenza ordinata degli elementi dell'insieme

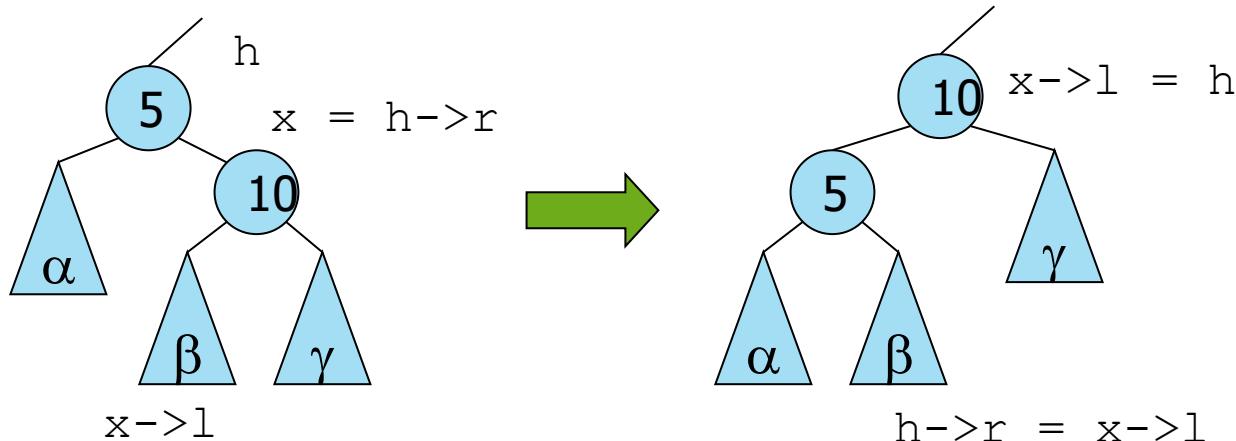
```
static void treePrintR(link h, link z, int strategy) {
    if (h == z)
        return;
    if (strategy == PREORDER)
        ITEMstore(h->item);
    treePrintR(h->l, z, strategy);
    if (strategy == INORDER)
        ITEMstore(h->item);
    treePrintR(h->r, z, strategy);
    if (strategy == POSTORDER)
        ITEMstore(h->item);
}
void BSTvisit(BST bst, int strategy) {
    if (BSTempty(bst))
        return;
    treePrintR(bst->root, bst->z, strategy);
}
```

Rotazione a destra di BST



```
link rotR(link h) {  
    link x = h->l;  
    h->l = x->r;  
    x->r = h;  
    return x;  
}
```

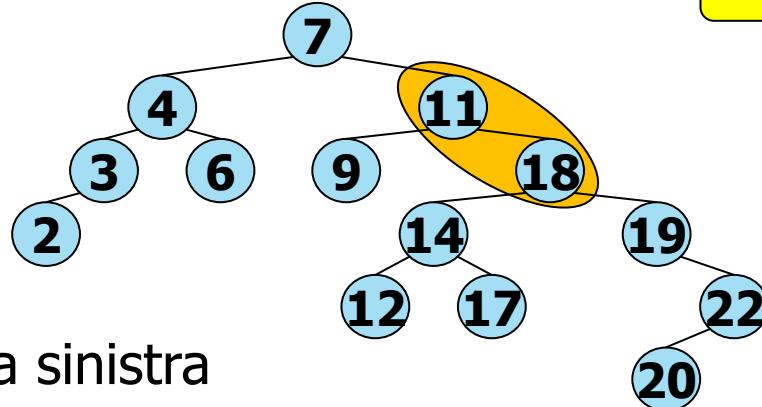
Rotazione a sinistra di BST



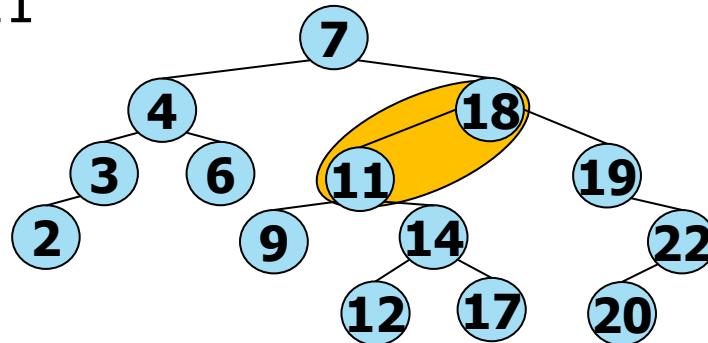
```
link rotL(link h) {  
    link x = h->r;  
    h->r = x->l;  
    x->l = h;  
    return x;  
}
```

Esempio

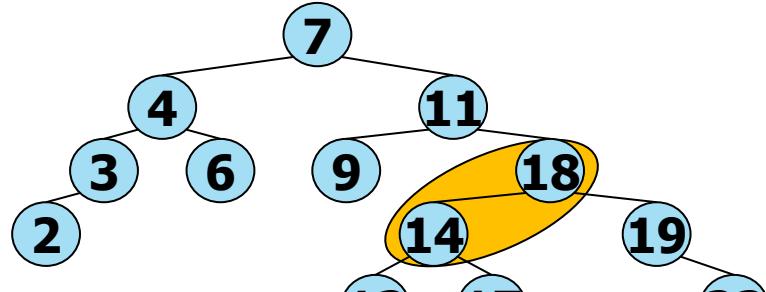
Nodi sentinella z non riportati



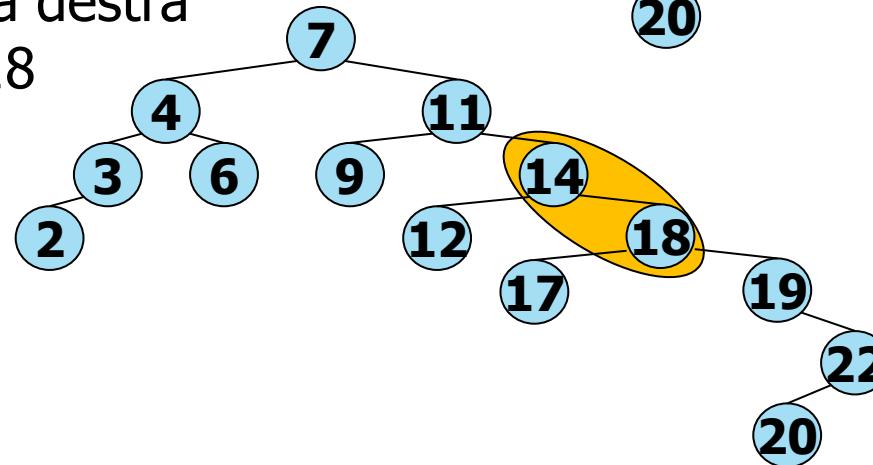
Rotazione a sinistra
attorno a 11



Esempio



Rotazione a destra
attorno a 18

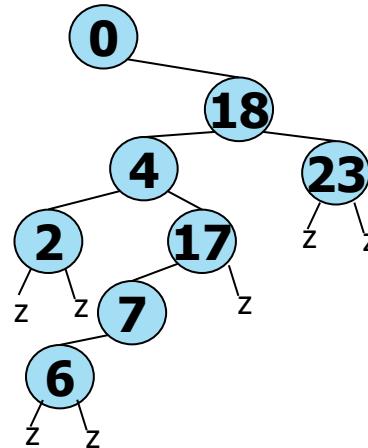
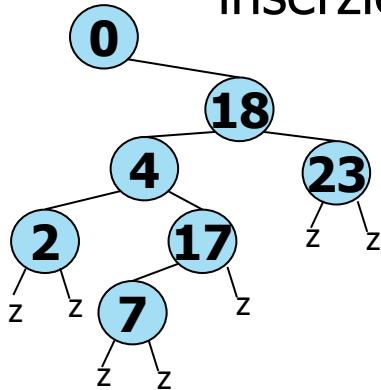


BSTinsert (in radice)

- Inserimento dalle foglie a scelta e non obbligatorio
- Nodi più recenti nella parte alta del BST
- Inserimento ricorsivo alla radice:
 - inserimento nel sottoalbero appropriato
 - rotazione per farlo diventare radice dell'albero principale.

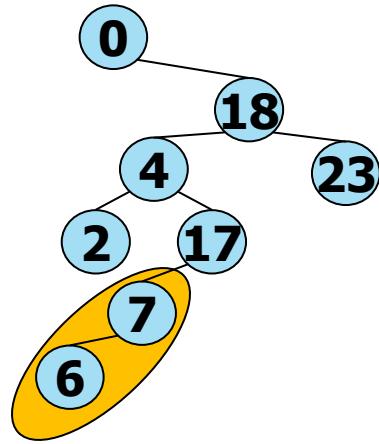
Esempio

inserzione di 6

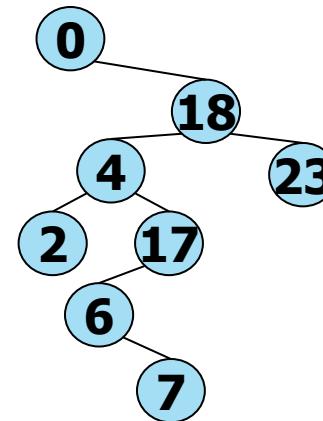


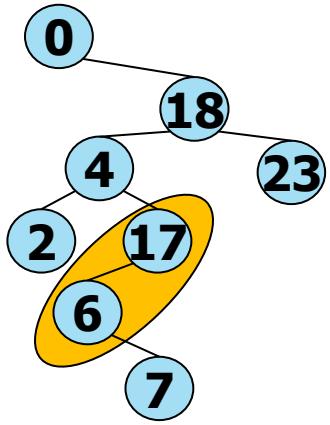
inserzione di 6 alla radice
del sottoalbero opportuno

Nodi sentinella z non più riportati

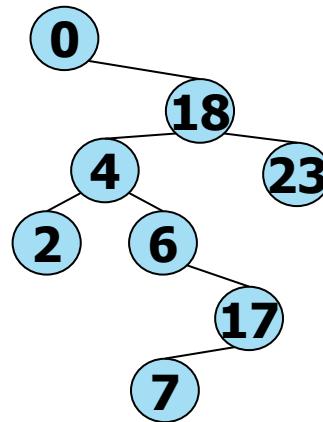


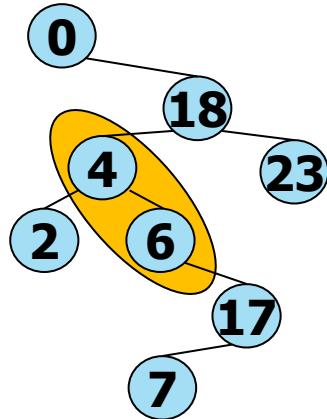
rotazione a DX



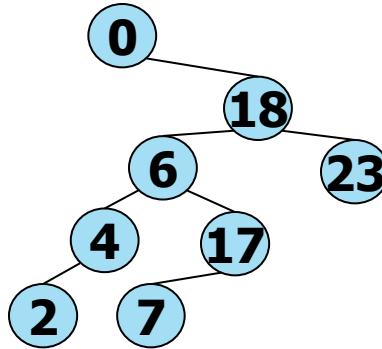


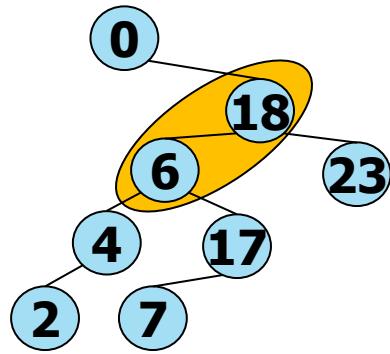
rotazione a DX



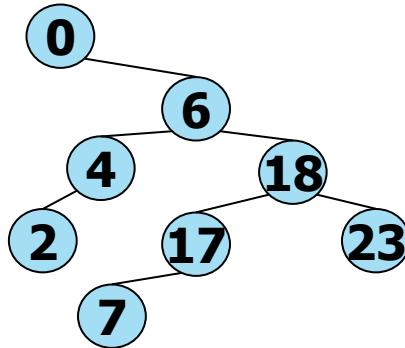


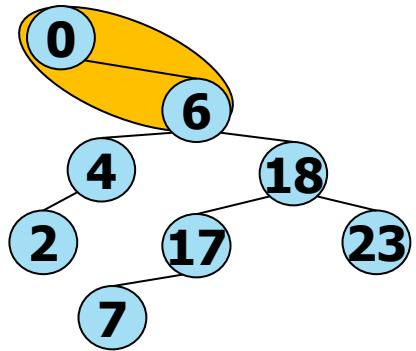
rotazione a SX



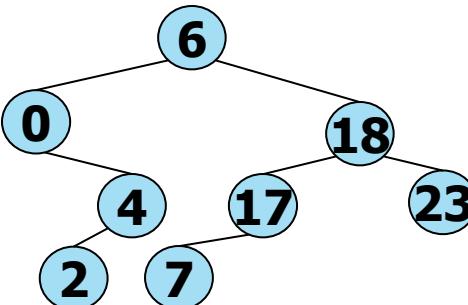


rotazione a DX





rotazione a SX



```
static link insertT(link h, Item x, link z) {
    if (h == z)
        return NEW(x, z, z);
    if (KEYcmp(KEYget(x), KEYget(h->item))== -1) {
        h->l = insertT(h->l, x, z);
        h = rotR(h);
    }
    else {
        h->r = insertT(h->r, x, z);
        h = rotL(h);
    }
    return h;
}

void BSTinsert_root(BST bst, Item x) {
    bst->root = insertT(bst->root, x, bst->z);
}
```

Estensioni dei BST elementari

PUNTATORE AL PADRE, NUMERO DI NODI NEL SOTTO-ALBERO

Estensione dei BST elementari

Al nodo elementare si possono aggiungere informazioni che permettono lo sviluppo semplice di nuove funzioni:

- puntatore al padre
- numero di nodi dell'albero radicato nel nodo corrente.

Queste informazioni devono ovviamente essere gestite (quando necessario) da tutte le funzioni già viste.

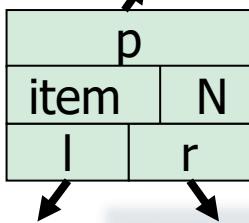
ADT di classe BST

nuove funzioni
funzioni modificate

BST.h

```
typedef struct binarysearchtree *BST;

BST    BSTinit();
void   BSTfree(BST bst); int BSTcount(BST bst);
int    BSTempty(BST bst);
Item   BSTmin(BST bst); Item BSTmax(BST bst);
void   BSTinsert_leafI(BST bst, Item x);
void   BSTinsert_leafR(BST bst, Item x);
void   BSTinsert_root(BST bst, Item x);
Item   BSTsearch(BST bst, Key k);
void   BSTdelete(BST bst, Key k);
Item   BSTselect(BST bst, int r); Order-Statistic BST
void   BSTvisit(BST bst, int strategy);
Item   BSTsucc(BST bst, Key k);
Item   BSTpred(BST bst, Key k);
void   BSTbalance(BST bst);
```



BSTnode



BST.c

```

#include <stdlib.h>
#include "Item.h"      puntatore al padre
#include "BST.h"       dimensione sottoalbero

typedef struct BSTnode* link;
struct BSTnode {Item item; link p; link l; link r; int N;};
struct binarysearchtree { link root; link z; };

static link NEW(Item item, link p, link l, link r, int N){
    link x = malloc(sizeof *x);
    x->item = item;
    x->p = p; x->l = l; x->r = r; x->N = N;
    return x;
}

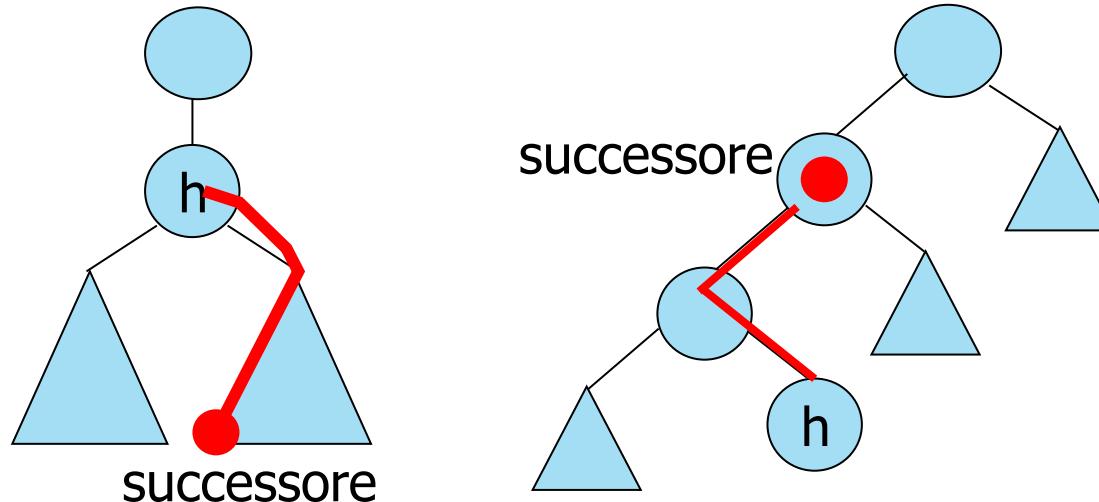
```

```
BST BSTinit( ) {  
    BST bst = malloc(sizeof *bst) ;  
    bst->root=(bst->z=NEW(ITEMsetNull(), NULL, NULL, NULL, 0));  
    return bst;  
}
```

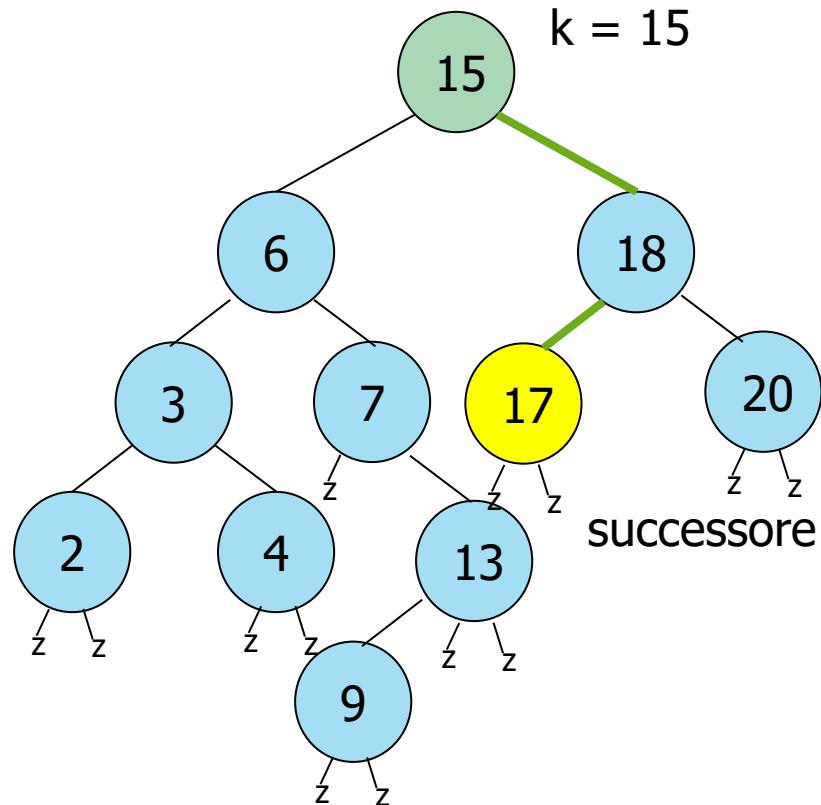
Successore

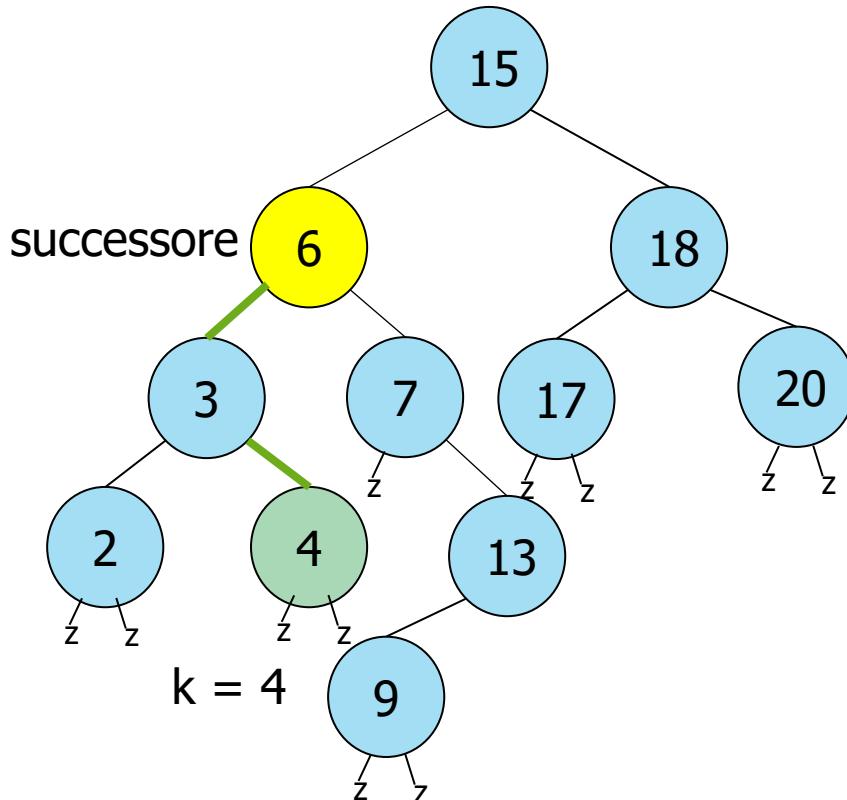
Successore di un item: se esiste, nodo h con un item con la più piccola chiave $>$ della chiave di item, altrimenti item vuoto. Due casi:

- $\exists \text{Right}(h)$: $\text{succ}(\text{key}(h)) = \min(\text{Right}(h))$
- $\nexists \text{Right}(h)$: $\text{succ}(\text{key}(h)) = \text{primo antenato di } h \text{ il cui figlio sinistro è anche un antenato di } h.$



Esempio





```
Item searchSucc(link h, key k, link z) {
    link p;
    if (h == z) return ITEMsetNull();
    if (KEYcmp(k, KEYget(h->item))==0) {
        if (h->r != z) return minR(h->r, z);
        else {
            p = h->p;
            while (p != z && h == p->r) {
                h = p; p = p->p;
            }
            return p->item;
        }
    }
    if (KEYcmp(k, KEYget(h->item))==-1)
        return searchSucc(h->l, k, z);
    return searchSucc(h->r, k, z);
}
Item BSTsucc(BST bst, Key k) {
    return searchSucc(bst->root, k, bst->z);
}
```

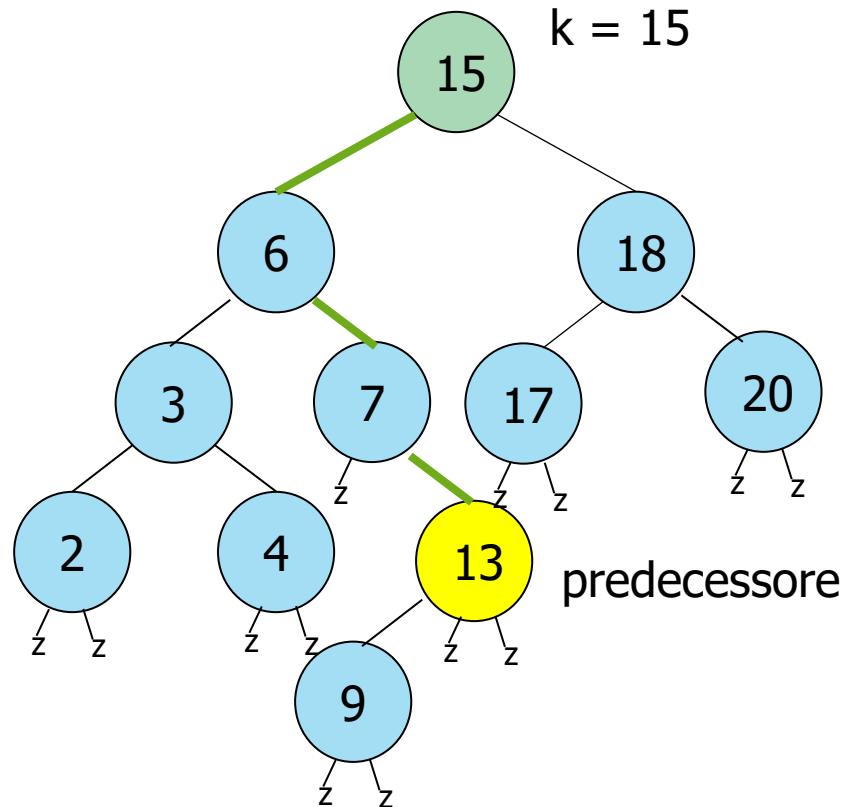
Predecessore

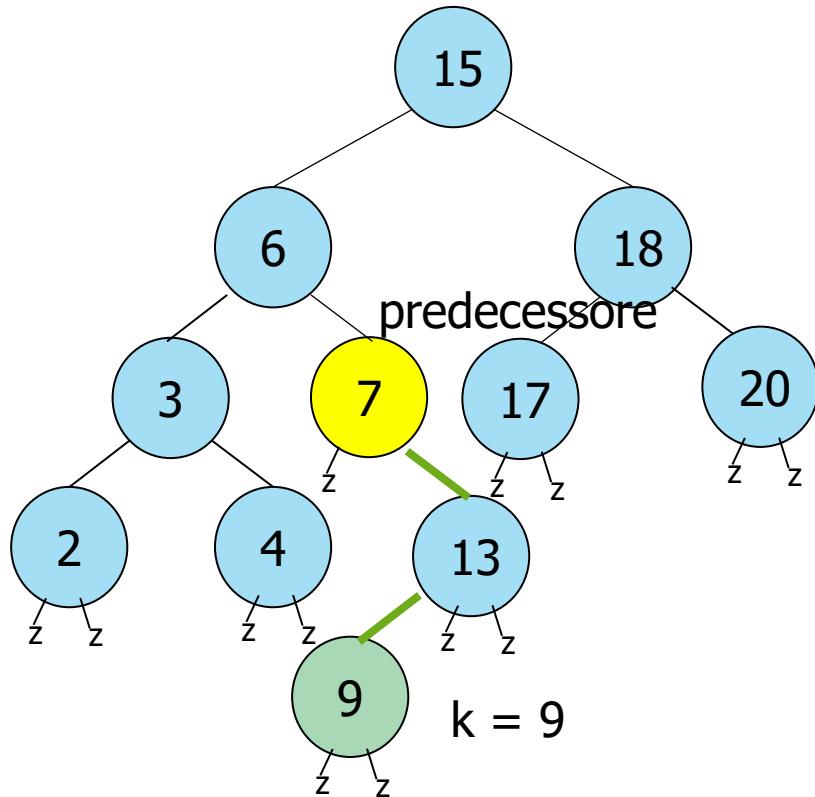
Predecessore di un item: nodo h con item con la più grande chiave $<$ della chiave di item.

Due casi:

- $\exists \text{Left}(h)$: $\text{pred}(\text{key}(h)) = \max(\text{Left}(h))$
- $\nexists \text{Left}(h)$: $\text{pred}(\text{key}(h)) = \text{primo antenato di } h \text{ il cui figlio destro è anche un antenato di } h$.

Esempio





```

Item searchPred(link h, Key k, link z) {
    link p;
    if (h == z) return ITEMsetNull();
    if (KEYcmp(k, KEYget(h->item))==0) {
        if (h->l != z) return maxR(h->l, z);
        else {
            p = h->p;
            while (p != z && h == p->l) {h = p; p = p->p;}
            return p->item;
        }
    }
    if (KEYcmp(k, KEYget(h->item))==-1)
        return searchPred(h->l, k, z);
    return searchPred(h->r, k, z);
}
Item BSTpred(BST bst, Key k) {
    return searchPred(bst->root, k, bst->z);
}

```

BSTinsert (in foglia)

RICORSIVO

```
link insertR(link h, Item x, link z) {
    if (h == z)
        return NEW(x, z, z, z, 1);
    if (KEYcmp(KEYget(x), KEYget(h->item)) == -1) {
        h->l = insertR(h->l, x, z); h->l->p = h;
    }
    else {
        h->r = insertR(h->r, x, z); h->r->p = h;
    }
    (h->N)++;
    return h;
}
void BSTinsert_leafR(BST bst, Item x) {
    bst->root = insertR(bst->root, x, bst->z);
}
```

BSTinsert (in foglia)

ITERATIVO

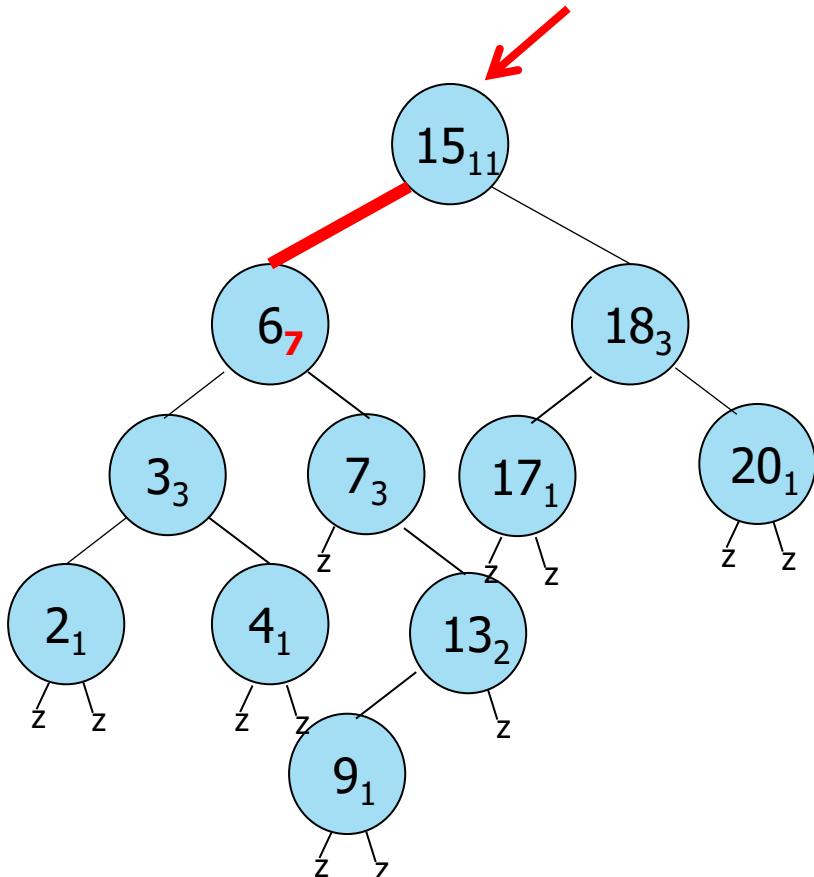
```
void BSTinsert_leafI(BST bst, Item x) {
    link p = bst->root, h = p;
    if (bst->root == bst->z) {
        bst->root = NEW(x, bst->z, bst->z, bst->z, 1);
        return;
    }
    while (h != bst->z) {
        p = h; h->N++;
        h=(KEYcmp(KEYget(x), KEYget(h->item))==-1) ? h->l : h->r;
    }
    h = NEW(x, p, bst->z, bst->z, 1);
    if (KEYcmp(KEYget(x), KEYget(p->item))==-1)
        p->l = h;
    else
        p->r = h;
}
```

BSTselect

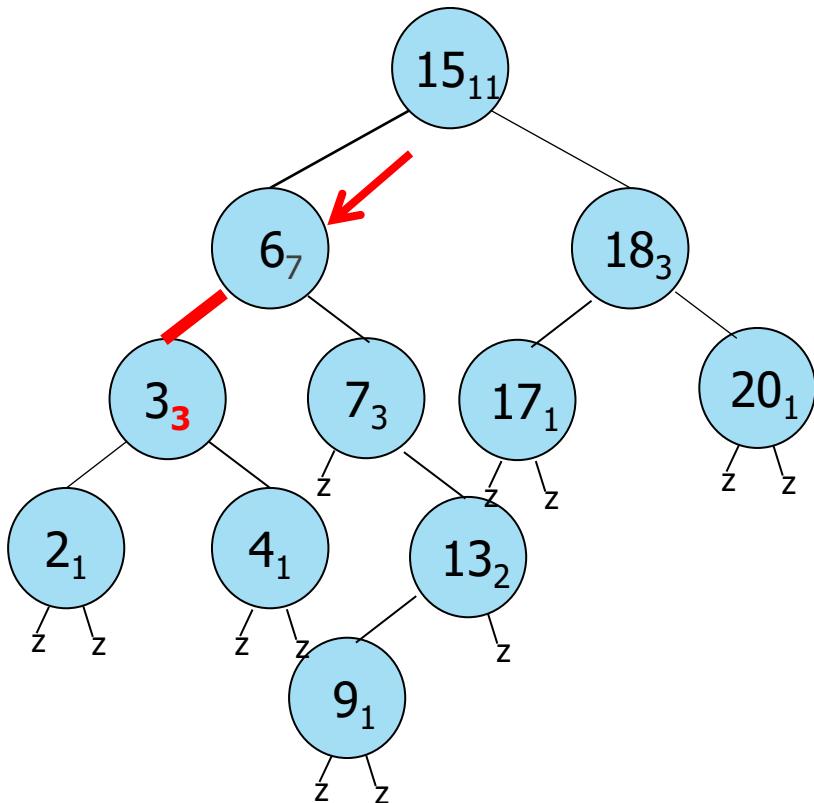
Selezione dell'item con la r-esima chiave più piccola (rango r = chiave in posizione r nell'ordinamento, ad esempio se r=0 item con chiave minima): t è il numero di nodi del sottoalbero sinistro:

- $t = r$: ritorno la radice del sottoalbero
- $t > r$: ricorsione nel sottoalbero sinistro alla ricerca della k-esima chiave più piccola
- $t < r$: ricorsione nel sottoalbero destro alla ricerca della $(r-t-1)$ -esima chiave più piccola

Esempio



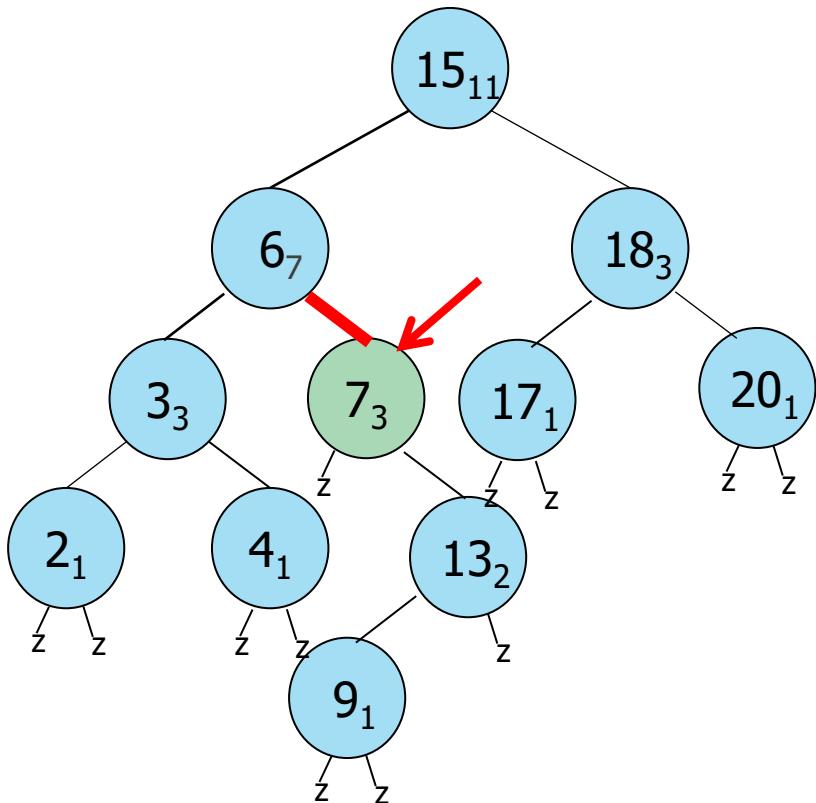
rango = r = 4
quinta più
piccola chiave
t=7
7>4 scendo a SX



$$r = 4$$

$$t=3$$

3<4 scendo a DX
cerco $r=4-3-1=0$

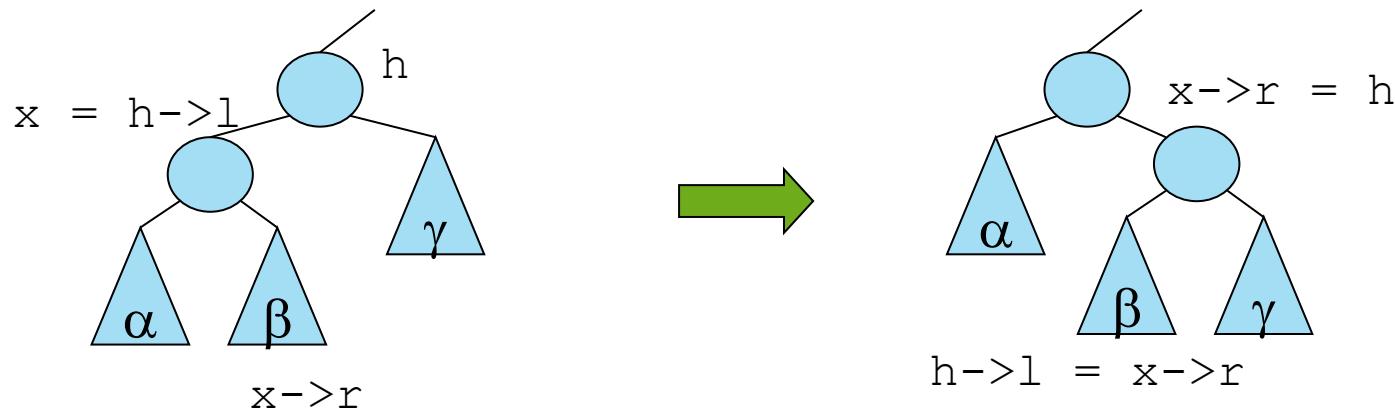


$r = 0$
 $t=0$
return 7

```
Item selectR(link h, int r, link z) {
    int t;
    if (h == z)
        return ITEMsetNull();
    t = h->l->N;
    if (t > r)
        return selectR(h->l, r, z);
    if (t < r)
        return selectR(h->r, r-t-1, z);
    return h->item;
}
```

```
Item BSTselect(BST bst, int r) {
    return selectR(bst->root, r, bst->z);
}
```

Rotazione a destra di BST

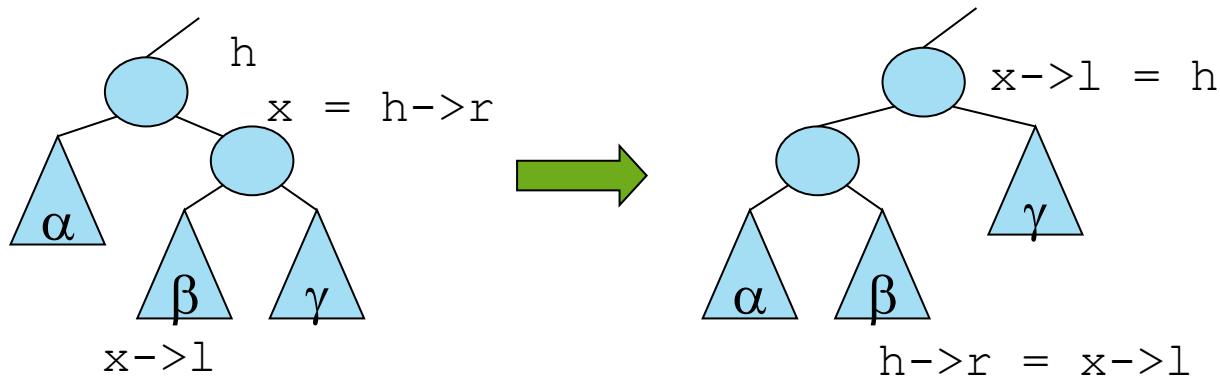


aggiornamento
dimensione
sottoalberi

```
link rotR(link h) {  
    link x = h->l;  
    h->l = x->r;  
x->r->p = h;  
    x->r = h;  
x->p = h->p;  
h->p = x;  
    x->N = h->N;  
    h->N = 1;  
    h->N += (h->l) ? h->l->N : 0;  
    h->N += (h->r) ? h->r->N : 0;  
    return x;  
}
```

aggiornamento
puntatore
al padre

Rotazione a sinistra di BST



aggiornamento
dimensione
sottoalberi

```
link rotL(link h) {  
    link x = h->r;  
    h->r = x->l;  
    x->l->p = h;  
    x->l = h;  
x->p = h->p;  
h->p = x;  
    x->N = h->N;  
    h->N = 1;  
    h->N += (h->l) ? h->l->N : 0;  
    h->N += (h->r) ? h->r->N : 0;  
    return x;  
}
```

aggiornamento
puntatore
al padre

BSTinsert (in radice)

```
link insert(link h, Item x, link z) {
    if ( h == z)
        return NEW(x, z, z, z, 1);
    if (KEYcmp(KEYget(x), KEYget(h->item))==-1) {
        h->l = insertT(h->l, x, z); h = rotR(h); h->N++;
    }
    else {
        h->r = insertT(h->r, x, z); h = rotL(h); h->N++;
    }
    return h;
}

void BSTinsert_root(BST bst, Item x) {
    bst->root = insertT(bst->root, x, bst->z);
}
```

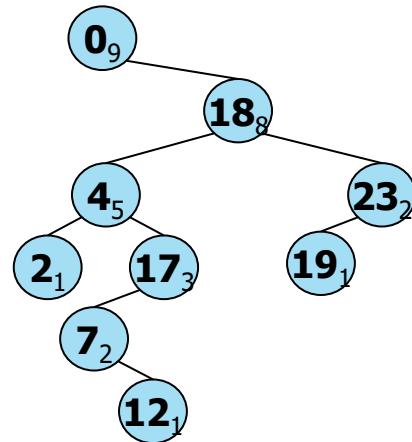
BSTpartition

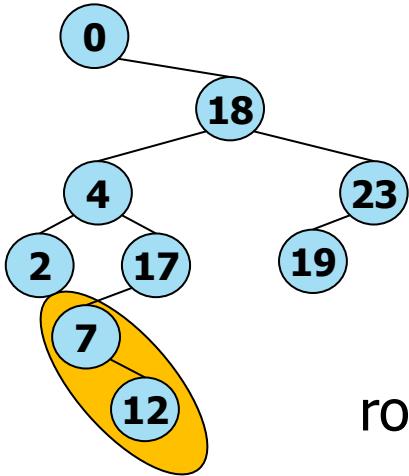
- Riorganizza l'albero avendo l'item con la k-esima chiave più piccola nella radice:
 - porre il nodo come radice di un sottoalbero:
 - $t > k$: ricorsione nel sottoalbero sinistro, partizionamento rispetto alla k-esima chiave più piccola, al termine rotazione a destra
 - $t < k$: ricorsione nel sottoalbero destro, partizionamento rispetto alla $(k-t-1)$ -esima chiave più piccola , al termine rotazione a sinistra
 - Sovente il partizionamento si fa attorno alla chiave mediana

Esempio

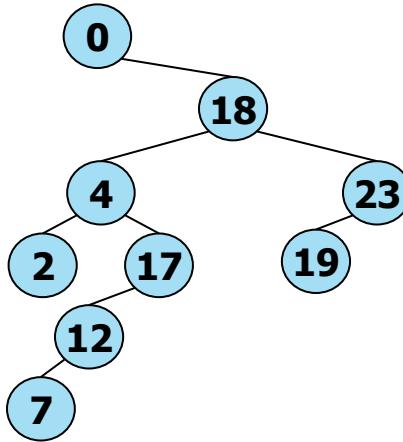
Nodi sentinella z non riportati

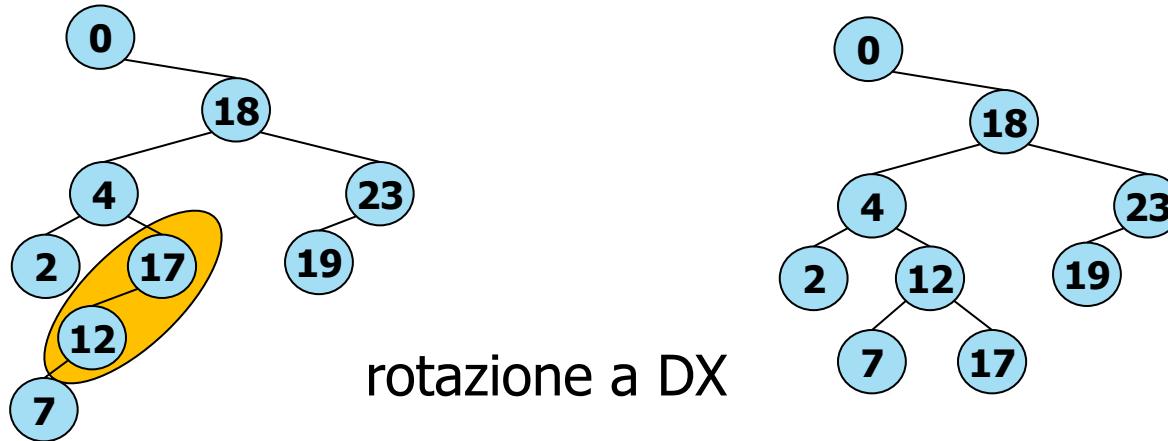
Partizionamento rispetto alla
5a chiave più piccola (12, k=4)

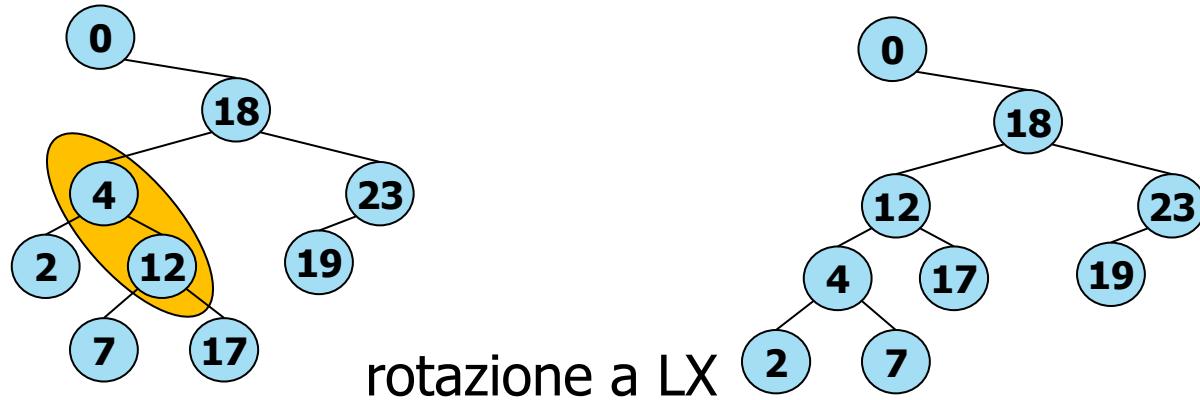


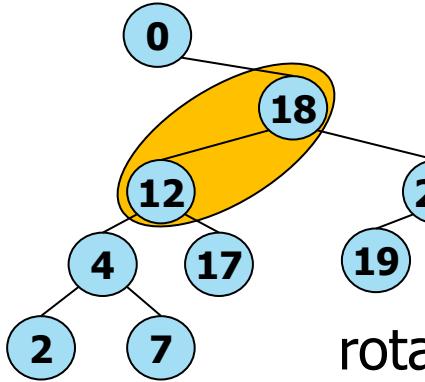


rotazione a SX

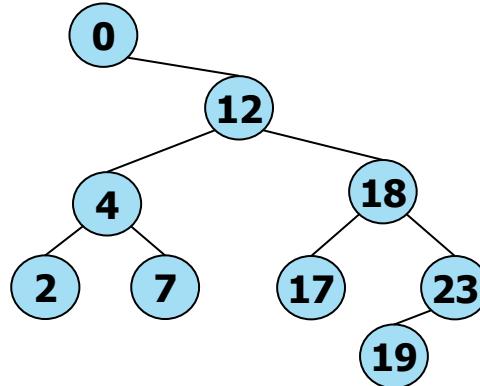


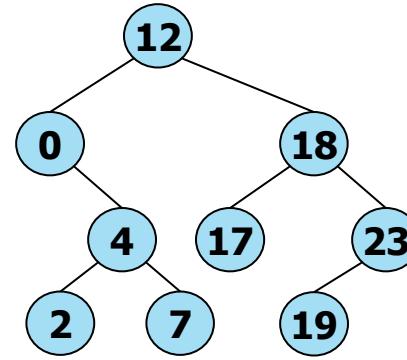
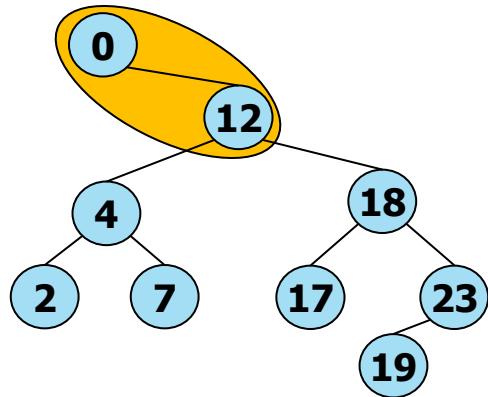






rotazione a DX





rotazione a SX

```
link partR(link h, int r) {
    int t = h->l->N;
    if (t > r) {
        h->l = partR(h->l, r);
        h = rotR(h);
    }
    if (t < r) {
        h->r = partR(h->r, r-t-1);
        h = rotL(h);
    }
    return h;
}
```

BSTdelete

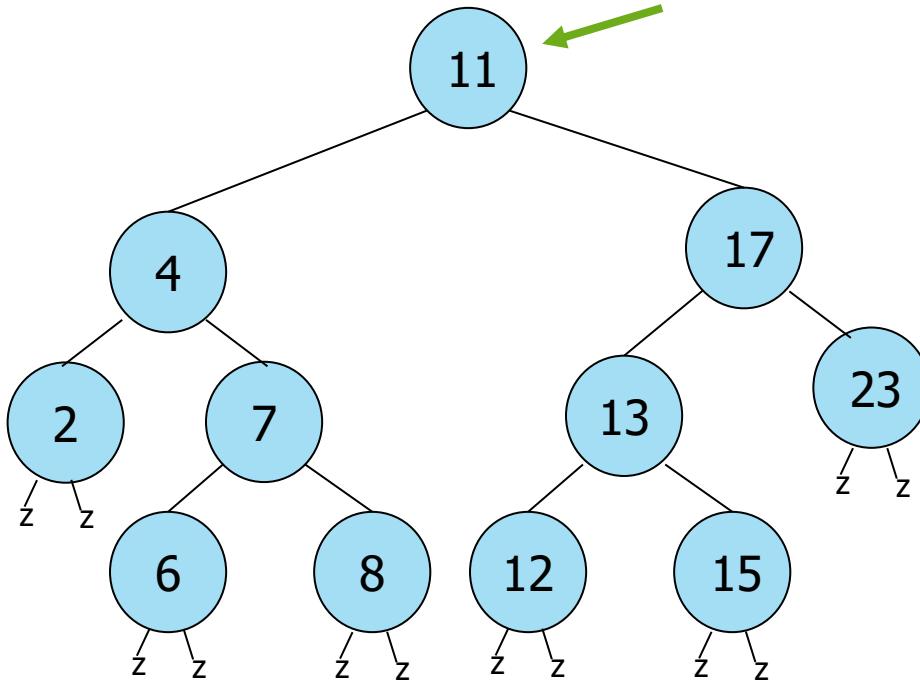
Per cancellare da un albero binario di ricerca un nodo con item con chiave k bisogna mantenere:

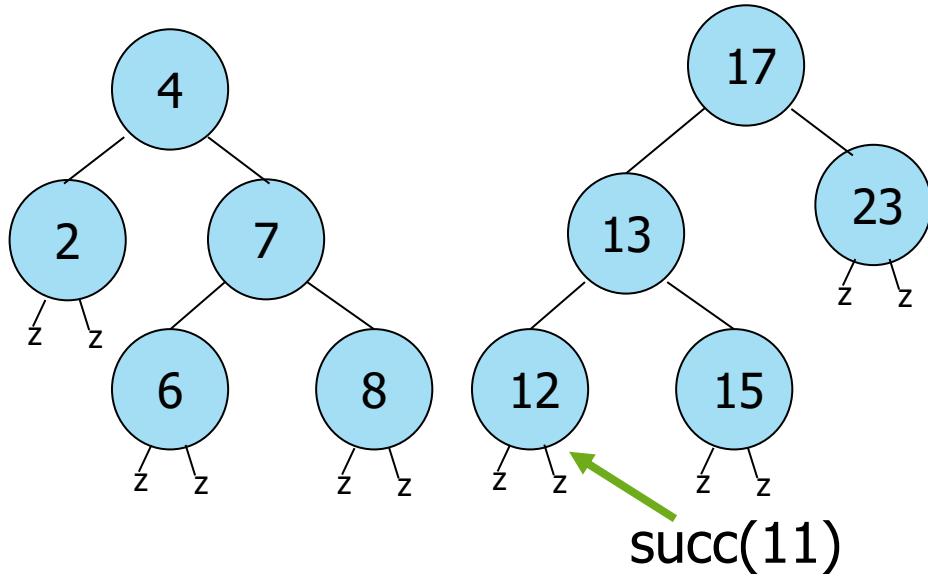
- la proprietà dei BST
- la struttura ad albero binario

Passi:

- controllare se il nodo con l'item da cancellare è in uno dei sottoalberi. Se sì, cancellazione ricorsiva nel sottoalbero
- se è la radice, eliminarlo e ricombinare i 2 sottoalberi. La nuova radice è il successore o il predecessore dell'item cancellato.

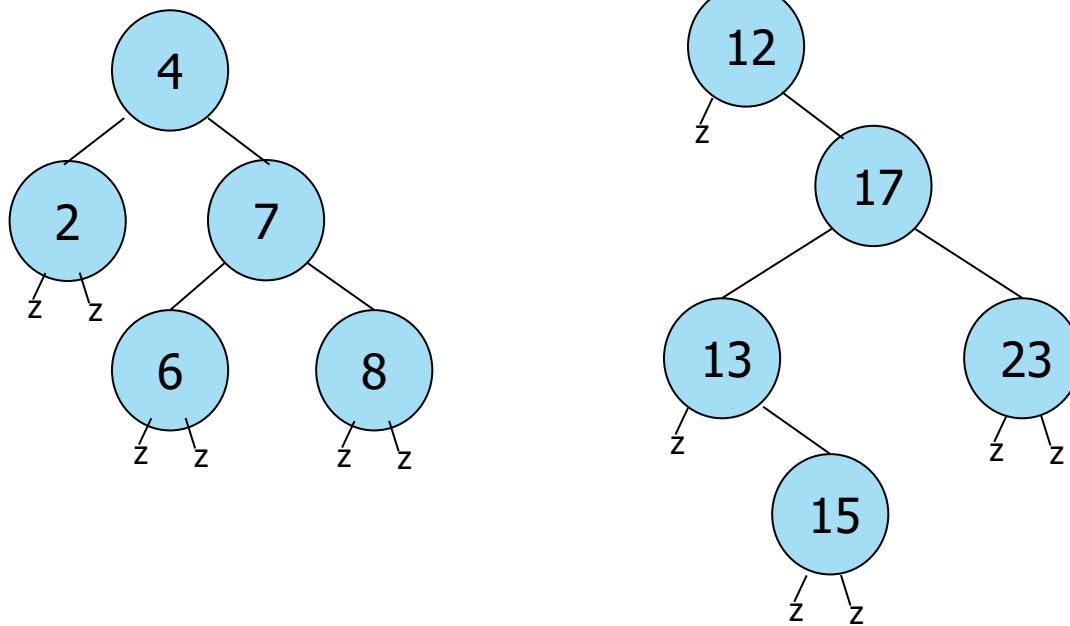
Cancellazione di una radice



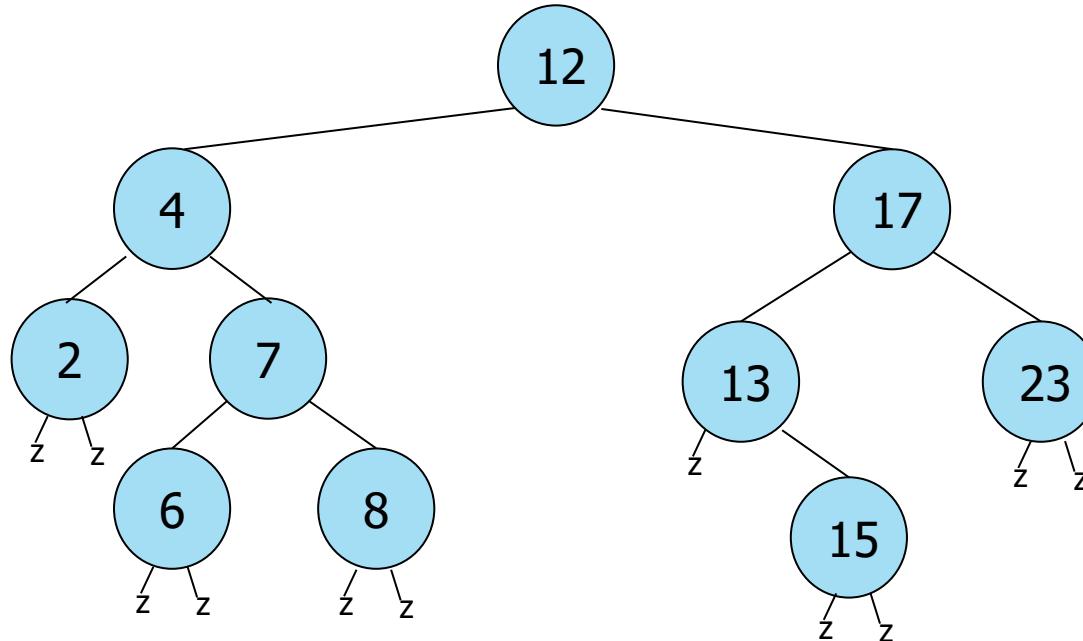


$\text{succ}(11)$

partition rispetto a 12



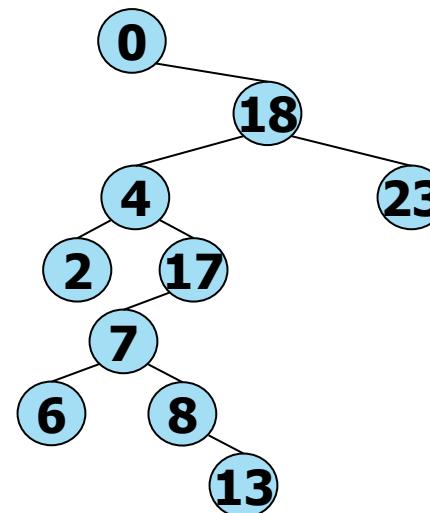
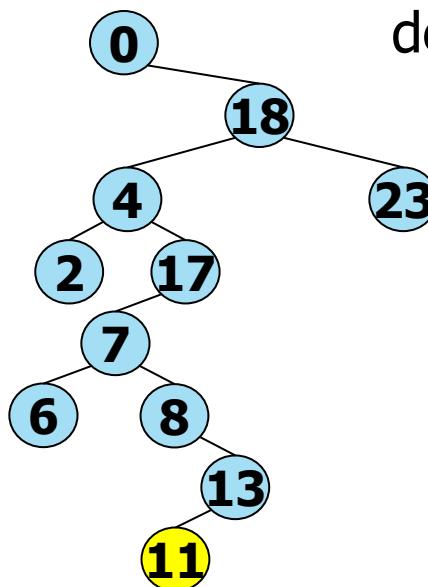
ricostruzione dell'albero con radice 12



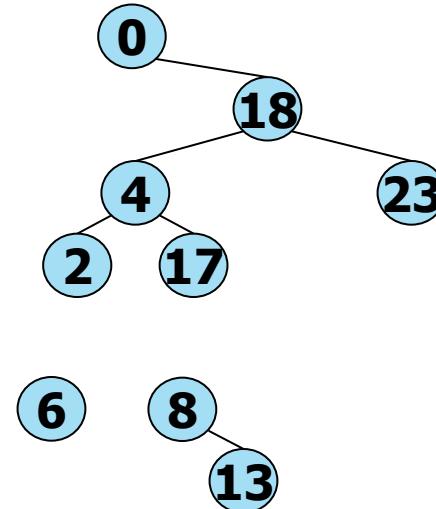
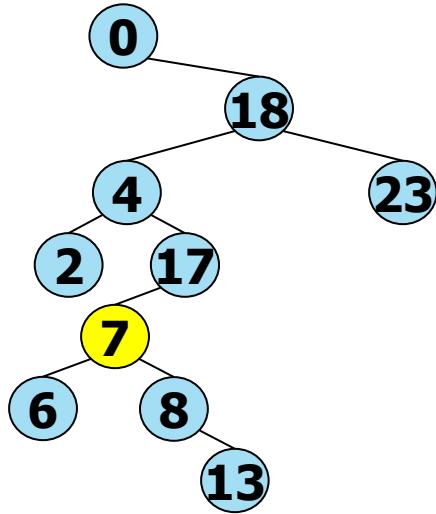
Esempio

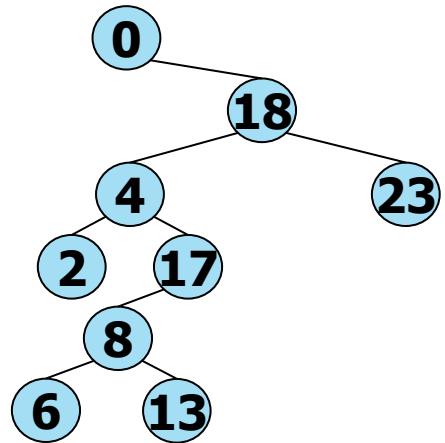
Nodi sentinella z non riportati

cancellazione in sequenza di 11, 7, 4

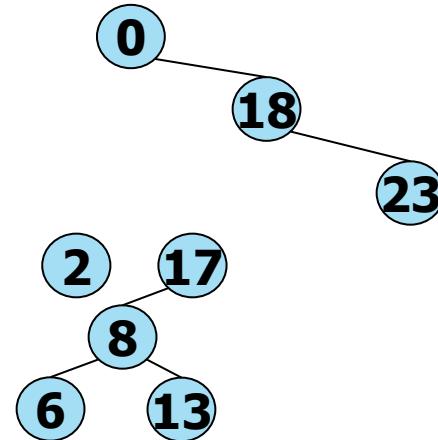
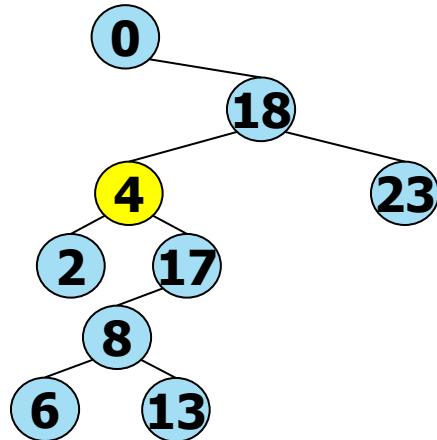


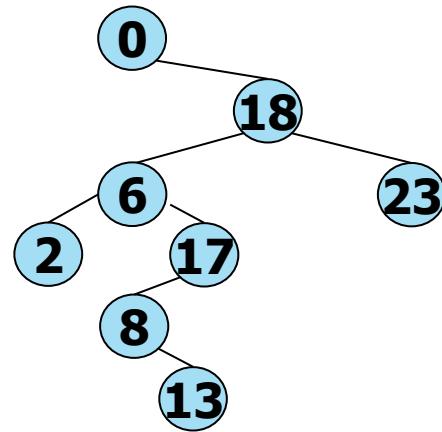
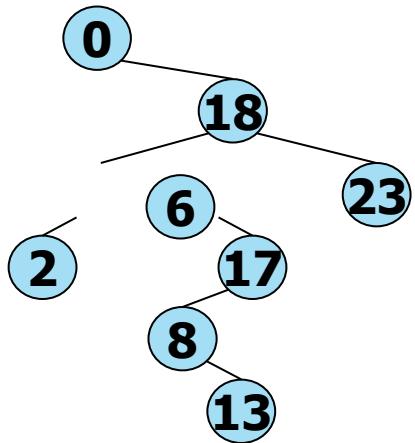
delete 7





delete 4





```
link joinLR(link a, link b, link z) {  
    if (b == z)  
        return a;  
    b = partR(b, 0);  
    b->l = a;  
    a->p = b;   
    b->N = a->N + b->r->N +1;  
    return b;  
}
```

aggiornamento
puntatore
al padre

aggiornamento
dimensione
sottoalberi

```
link deleteR(link h, key k, link z) {  
    link y, p;  
    if (h == z) return z;  
    if (KEYcmp(k, KEYget(h->item)) == -1)  
        h->l = deleteR(h->l, k, z);  
    if (KEYcmp(k, KEYget(h->item)) == 1)  
        h->r = deleteR(h->r, k, z);  
    (h->N)--;  
    if (KEYcmp(k, KEYget(h->item)) == 0) {  
        y = h;  p = h->p;  
        h = joinLR(h->l, h->r, z);  h->p = p;  
        free(y);  
    }  
    return h;  
}  
void BSTdelete(BST bst, key k) {  
    bst->root = deleteR(bst->root, k, bst->z);  
}
```

aggiornamento
numero di nodi

aggiornamento
puntatore
al padre

Bilanciamento

- A priori: vincoli che garantiscono un albero perfettamente bilanciato (B-tree) o con sbilanciamento limitato (alberi 2-3-4, RB-tree)
- Ribilanciamento a richiesta:
 - partizionamento ricorsivo attorno alla chiave mediana inferiore (algoritmo semplice)
 - algoritmo di Day, Stout e Warren, di complessità $O(n)$ costruisce un albero quasi completo (tutti i livelli completi, tranne l'ultimo riempito da sinistra a destra, cfr heap)

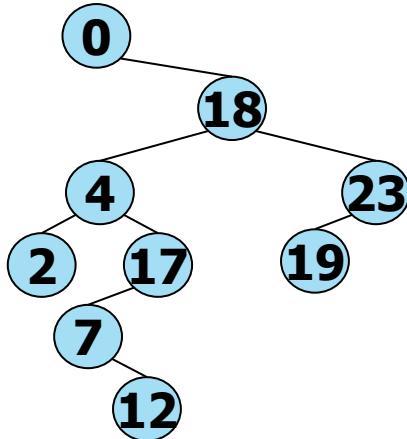
Bisogna valutare il rapporto tra costo e frequenza di ribilanciamento.

```
static link balanceR(link h, link z) {
    int r;
    if (h == z)
        return z;
    r = (h->N+1)/2-1;
    h = partR(h, r);
    h->l = balanceR(h->l, z);
    h->r = balanceR(h->r, z);
    return h;
}

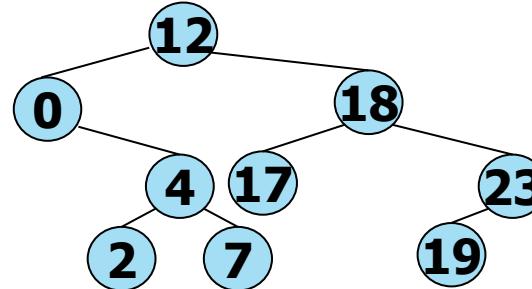
void BSTbalance(BST bst) {
    bst->root = balanceR(bst->root, bst->z);
}
```

Esempio

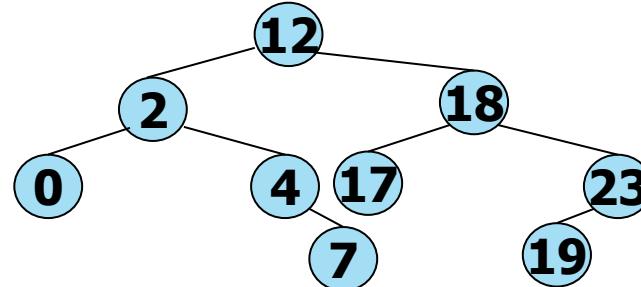
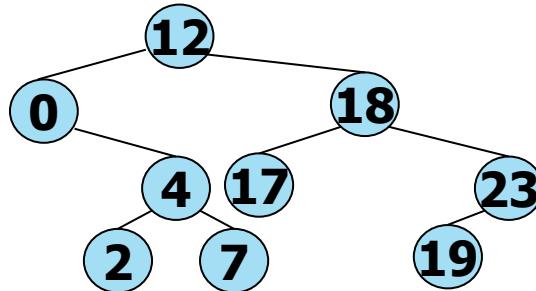
Nodi sentinella z non riportati



La mediana inferiore è 12. partR
con $r=4$ dà:

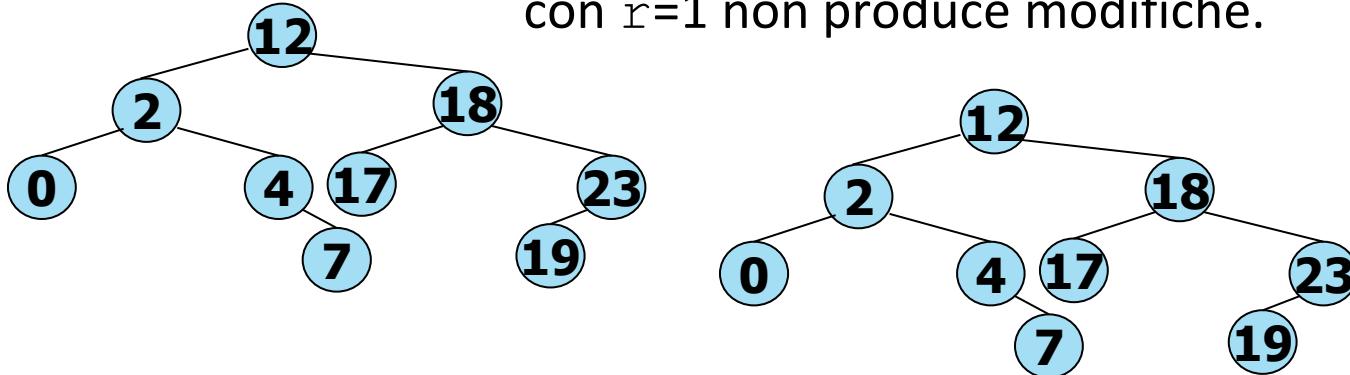


Ricorsione sul sottoalbero sinistro di 12. La mediana inferiore è 2. partR con $r=1$ dà:



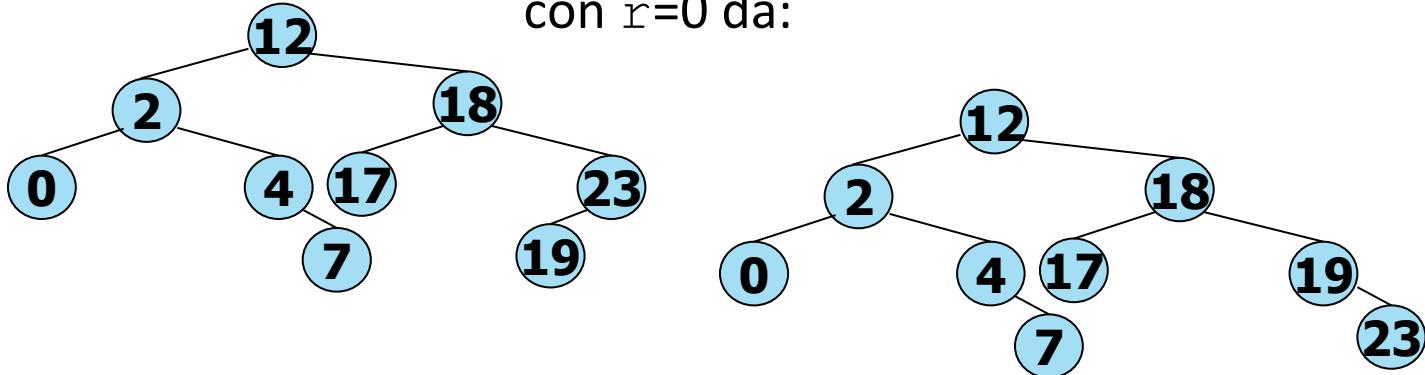
Sui sottoalberi radicati in 0, 4 e 7 la ricorsione non produce modifiche.

Ricorsione sul sottoalbero destro di 12. La mediana inferiore è 18. partR con $r=1$ non produce modifiche.



Sul sottoalbero radicato in 17 la ricorsione non produce modifiche.

Ricorsione sul sottoalbero destro di 18. La mediana inferiore è 19. partR con $r=0$ dà:



Sul sottoalbero radicato in 23 la ricorsione non produce modifiche.

Estensioni delle strutture dati

Prima di sviluppare una nuova struttura dati è bene valutare se si possano «estendere» strutture esistenti con informazioni opportune.

Procedura:

1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

Esempio: Order-Statistic BST

Procedura:

1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

BST

dimensione
del sottoalbero

$O(1)$

```
Item BSTselect(BST, int);
```

Interval BST

BST la cui chiave è un intervallo chiuso: coppia ordinata di reali $[t_1, t_2]$, dove $t_1 \leq t_2$ e $[t_1, t_2] = \{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$.

L'item intervallo $[t_1, t_2]$ può essere realizzato da una **struct** con campi **low** = t_1 e **high** = t_2 .

BST con inserzione secondo
l'estremo inferiore

Procedura:

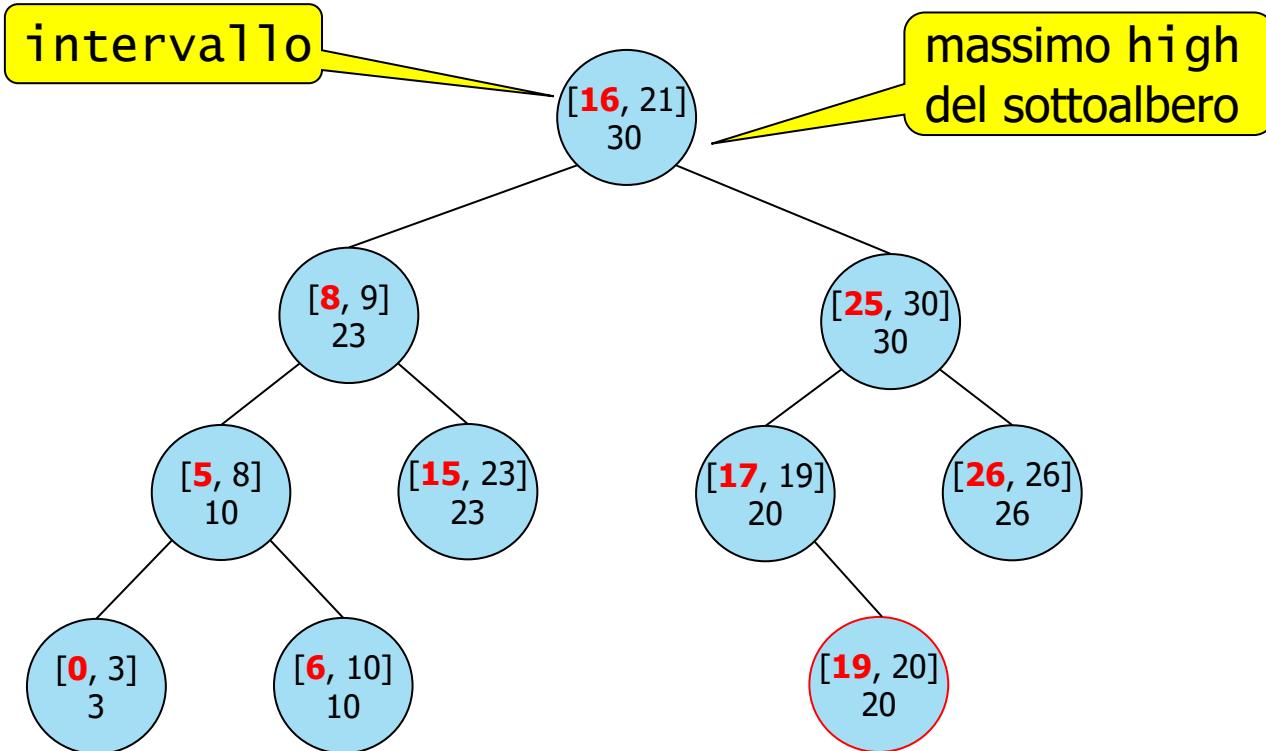
1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari
senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

max: massimo high
del sottoalbero

O(1)

Item IBSTsearch(BST, Item);

Esempio



Riferimenti

- Alberi binari
 - Sedgewick 5.6, 5.7
- Binary Search Trees
 - Cormen 13.1, 13.2, 13.3
 - Sedgewick 12.5, 12.8, 12.9
- Order-statistic BST
 - Cormen 15.1
- Interval BST
 - Cormen 15.3
 - Lucidi di approfondimento