

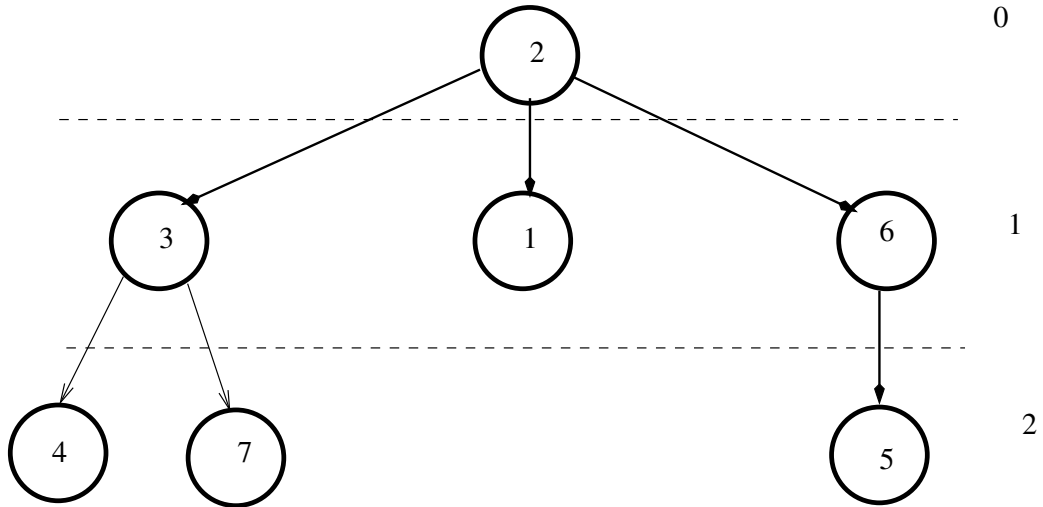
Strutture ad albero

Salvatore Filippone
salvatore.filippone@uniroma2.it

Gli alberi sono un tipo di struttura dati presente in innumerevoli applicazioni in informatica.

Alberi, Nodi, Radici e Foglie

- Ogni albero è costituito da un insieme di *nodi* collegati tra loro;
- Un albero che non sia vuoto contiene uno ed un solo *nodo radice*;
- Ogni nodo j (tranne la radice) ha uno ed un solo *nodo genitore*, ossia esiste un solo nodo i tale che (i, j) è un ramo dell'albero; un nodo generico può avere degli *antenati*;
- Il nodo radice non ha un genitore (né tantomeno antenati);
- Ogni nodo *può* avere dei *figli* e dei *discendenti*;
- Da ogni nodo parte un *sottoalbero* (che potrebbe contenere solo il nodo stesso) di cui il nodo è radice;
- Un nodo che non abbia figli si dice *foglia*;
- I nodi possono contenere delle informazioni (determinate dalla applicazione).



- L'ordine in cui compaiono i figli è significativo;
- Un albero è una *generalizzazione* di una lista;
- Una lista può essere un caso particolare di un albero;
- Concettualmente, un nodo e una radice di un sottoalbero sono la stessa cosa;

Una operazione fondamentale in un albero è la

Visita

Esame di tutti i nodi di un albero secondo un ordine prestabilito

Durante la “visita” di ciascun nodo si possono intraprendere azioni, quali

- Stampare il contenuto del nodo;
- Accodare il contenuto del nodo ad un insieme di dati;

Principali tipi di visita

- In profondità (preordine, postordine, inordine)
- In ampiezza.

Visita in *preordine*

```
visitaProfondità(Tree t);  
t  $\neq$  nil;  
visita la radice di t ;  
Tree u  $\leftarrow$  t.leftmostChild();  
while u  $\neq$  nil do  
    visitaProfondità(u);  
    u  $\leftarrow$  u.rightSibling();
```

La visita in preordine fornisce p.es. la linea di successione delle famiglie reali.

Visita in *postordine*

```
visitaProfondità(Tree  $t$ );  
 $t \neq \mathbf{nil}$ ;  
Tree  $u \leftarrow t.\text{leftmostChild}()$ ;  
while  $u \neq \mathbf{nil}$  do  
    visitaProfondità( $u$ );  
     $u \leftarrow u.\text{rightSibling}()$ ;  
visita la radice di  $t$  ;
```

Si consideri l'elenco dei k figli di un nodo, e si dividano in due gruppi, T_1, \dots, T_i e T_{i+1}, \dots, T_k e si proceda come segue

Visita in *inordine*

```
 $t \neq \text{nil};$   
Tree  $u \leftarrow t.\text{leftmostChild}();$   
 $l = 1;$   
while  $l \leq i$  do  
    visitaProfondità( $u$ );  
     $u \leftarrow u.\text{rightSibling}();$   
     $l \leftarrow l + 1;$   
visita la radice di  $t$  ;  
while  $l \leq k$  do  
    visitaProfondità( $u$ );  
     $u \leftarrow u.\text{rightSibling}();$   
     $l \leftarrow l + 1;$ 
```


Visita in *ampiezza*

O per livelli; necessita di una *coda* ausiliaria

```
t  $\neq$  nil;  
Queue Q  $\leftarrow$  Queue();  
Q.enqueue(t);  
while not Q.isEmpty() do  
    u  $\leftarrow$  Q.dequeue();  
    visita (u);  
    Tree u  $\leftarrow$  u.leftmostChild();  
    while u  $\neq$  nil do  
        Q.Enqueue(u);  
        u  $\leftarrow$  u.rightSibling();
```

Dimostrare che tutti i nodi del livello k vengono visitati prima dei nodi del livello $k + 1$.

Operatori

`Tree(v)` Costruisce un nuovo albero con un solo nodo;

`Item read()` “legge” il contenuto di un nodo;

`write(Item v)` “scrive” informazioni in un nodo;

`Tree parent()` restituisce il nodo genitore;

`Tree leftmostChild()` restituisce il primo figlio oppure `nil`;

`Tree rightSibling()` restituisce il prossimo figli, oppure `nil`;

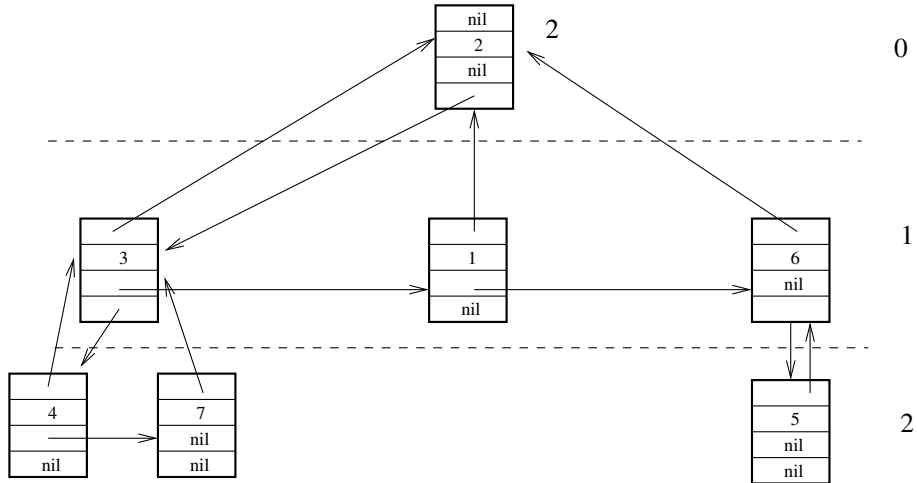
Operatori

`insertChild(Tree t)` Inserisci il sottoalbero `t` come primo figlio (`t.parent` deve essere `nil`);

`insertSibling(Tree t)` Inserisci il sottoalbero `t` come prossimo figlio (`t.parent` deve essere `nil`);

`deleteChild()` Elimina il sottoalbero con radice il primo figlio corrente;

`deleteSibling()` Elimina il sottoalbero con radice il prossimo figlio;



```
Node parent;
```

```
Node child;
```

```
Node sibling;
```

```
Item value;
```

```
Function Tree (item v)
```

```
    Tree t  $\leftarrow$  new Tree;
```

```
    t.value  $\leftarrow$  v;
```

```
    t.parent  $\leftarrow$  t.child  $\leftarrow$  t.sibling  $\leftarrow$  nil;
```

```
    Risultato t
```

```
Function insertChild (Tree t)
```

```
    t.parent  $\leftarrow$  this;
```

```
    t.sibling  $\leftarrow$  child;
```

```
    child  $\leftarrow$  t;
```

```
Function insertSibling (Tree t)
```

```
    t.parent  $\leftarrow$  parent;
```

```
    t.sibling  $\leftarrow$  sibling;
```

```
    sibling  $\leftarrow$  t;
```

```
Function deleteChild ()
```

```
    Node newChild  $\leftarrow$  child.rightSibling();
```

```
    delete (child);
```

```
    child  $\leftarrow$  newChild;
```

```
Function deleteSibling ()
```

```
    Node newSibling  $\leftarrow$  sibling.rightSibling();
```

```
    delete (sibling);
```

```
    sibling  $\leftarrow$  newSibling;
```

```
Function delete (Tree t)
```

```
    Node u  $\leftarrow$  t.leftmostChild();
```

```
    while u  $\neq$  nil do
```

```
        | Tree next  $\leftarrow$  u.rightSibling();
```

```
        delete (u);
```

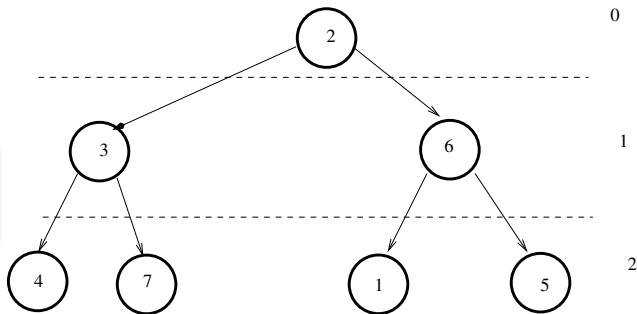
```
        u  $\leftarrow$  next;
```

```
    delete t;
```

Un caso particolare (e molto comune)
è quello in cui

Alberi binari

Ogni nodo ha al più due figli, il figlio
sinistro ed il *destro*



Un albero binario può essere memorizzato in un vettore (vedi *heap*): se il genitore è in
posizione i i due figli sono nelle posizioni $2i$ e $2i + 1$.

La realizzazione più comune fa uso di tre puntatori, genitore, figlio sinistro e figlio destro.

Node parent;

Node left;

Node right;

Item value;

Function *Tree* (*item v*)

Tree t \leftarrow *new Tree*;

t.value \leftarrow *v*;

t.parent \leftarrow *t.left* \leftarrow *t.right* \leftarrow **nil**;

Risultato *t*

Function *insertLeft* (*Tree t*)

t.parent \leftarrow **this**;

left \leftarrow *t*;

Function *insertRight* (*Tree t*)

t.parent \leftarrow **this**;

right \leftarrow *t*;

Function *deleteLeft* ()

if *left* \neq **nil** **then**

left.deleteLeft();

left.deleteRight();

delete (*left*);

left \leftarrow **nil**;

Function *deleteRight* ()

if *right* \neq **nil** **then**

right.deleteLeft();

right.deleteRight();

delete (*right*);

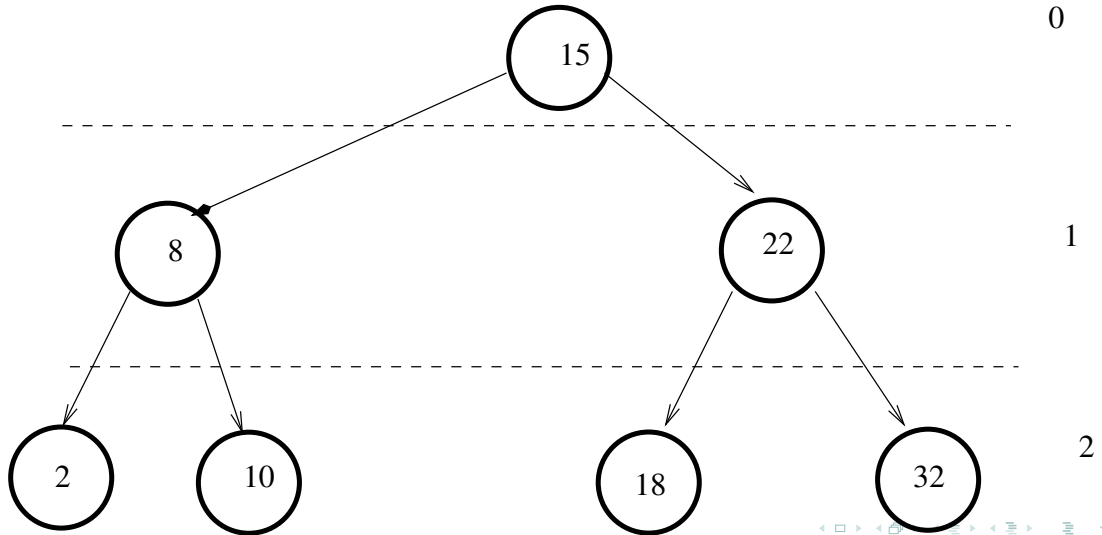
right \leftarrow **nil**;

Un uso naturale degli alberi binari è la realizzazione di un *dizionario*, ovvero il mantenimento di un insieme di dati ordinato per chiavi

Dizionario

- A ciascun nodo è associato un record, con la sua chiave;
- Per ciascun nodo, tutte le chiavi nel sottoalbero radicato nel figlio sinistro sono minori della chiave del nodo corrente;
- Per ciascun nodo, tutte le chiavi nel sottoalbero radicato nel figlio destro sono maggiori della chiave del nodo corrente;

Siamo quindi in una situazione perfettamente analoga alla ricerca binaria



Function *min* (*Tree T*)**while** *T.left* \neq *nil* **do***T* \leftarrow *T.left*;**Risultato** *T***Function** *max* (*Tree T*)**while** *T.right* \neq *nil* **do***T* \leftarrow *T.right*;**Risultato** *T***Function** *lookupNode* (*Tree T*, *item x*)**while** *T* \neq *nil* **and** *T.key* \neq *x* **do***T* \leftarrow *if* (*x* < *T.key*, *T.left*, *T.right*);**Risultato** *T*

Function *link* (*Tree p*, *Tree u*, *item x*)**if** *u* \neq *nil* **then***u.parent* \leftarrow *p*;**if** *p* \neq *nil* **then****if** *x* < *p.key* **then***p.left* \leftarrow *u*;**else***p.right* \leftarrow *u*;

Function *insertNode* (*Tree t*, *item x*, *item v*)

```
Tree p  $\leftarrow$  nil;  
Tree u  $\leftarrow$  T;  
while u  $\neq$  nil and u.key  $\neq$  x do  
    | p  $\leftarrow$  u;  
    | u  $\leftarrow$  if (x < u.key, u.left, u.right);  
if u  $\neq$  nil then  
    | u.value  $\leftarrow$  v;  
else  
    | Tree n  $\leftarrow$  Tree(x, v);  
    | link(p, n, x);  
    | if p = nil then return n;  
return T;
```

Inserimento con chiave *x* e valore *v*

Function *removeNode* (*Tree T*, *item x*)

```
Tree u  $\leftarrow$  lookupNode(T, x);  
if u  $\neq$  nil then  
    | if u.left  $\neq$  nil and u.right  $\neq$  nil then  
    |     | Tree s  $\leftarrow$  u.right;  
    |     | while s.left  $\neq$  nil do  
    |     |     | s  $\leftarrow$  s.left;  
    |     | u.key  $\leftarrow$  s.key, u.value  $\leftarrow$  s.value;  
    |     | u  $\leftarrow$  s, x  $\leftarrow$  s.key;  
    | Tree t;  
    | if u.left  $\neq$  nil and u.right = nil then  
    |     | t  $\leftarrow$  u.left;  
    | else  
    |     | t  $\leftarrow$  u.right;  
    | link(u.parent, t, x);  
    | if u.parent = nil then T  $\leftarrow$  t;  
    | delete u;  
return T;
```

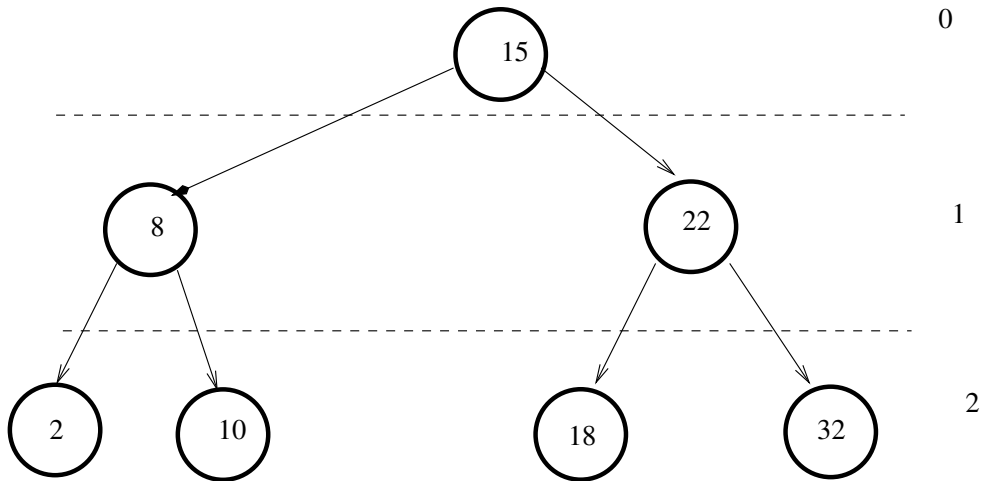
Function *Tree* successorNode (*Tree* *t*)

```
if t = nil then
    | Risultato t
if t.right ≠ nil then
    | Risultato min(t.right)
Tree p ← t.parent;
while p ≠ nil and t = p.right do
    | t ← p;
    | p ← p.parent;
Risultato p
```

Function *Tree* predecessorNode (*Tree* *t*)

```
if t = nil then
    | Risultato t
if t.left ≠ nil then
    | Risultato max(t.left)
Tree p ← t.parent;
while p ≠ nil and t = p.left do
    | t ← p;
    | p ← p.parent;
Risultato p
```

Costruzione di un albero “generico”



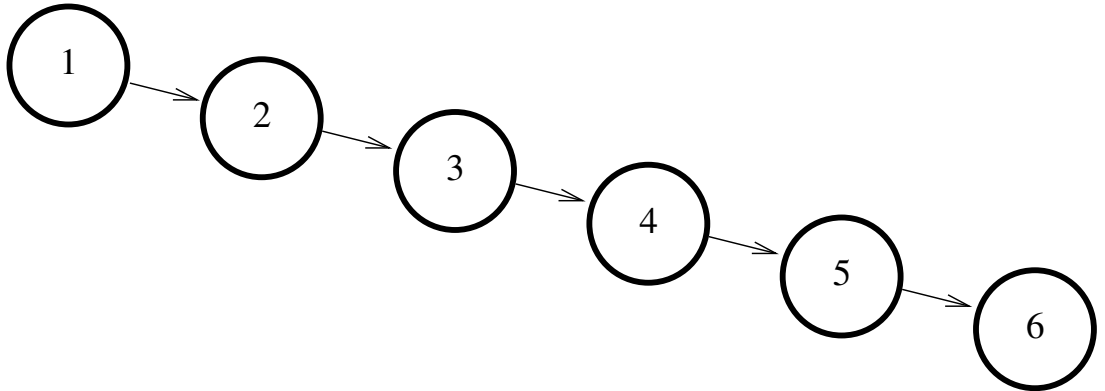
Quale sequenza di inserimenti potrebbe portare a questo albero?

Cosa succede se si inserisce la sequenza

$$V = [1, 2, 3, 4, 5, 6]?$$

Cosa succede se si inserisce la sequenza

$$V = [1, 2, 3, 4, 5, 6]?$$



$(k-)$ Bilanciamento

Un albero è perfettamente bilanciato se tutte le foglie si trovano allo stesso livello.

Un albero (binario) è k -bilanciato se la differenza di altezza tra tutti i sottoalberi figli dei vari nodi (tra i due sottoalberi figli destro e sinistro di un nodo) è al più k .

Tipologie di alberi bilanciati:

Alberi AVL

B-alberi

Alberi 2-3

Alberi rosso-neri

Gli alberi di Adel'son-Vel'skii e Landis sono degli alberi 1-bilanciati.

Fattore di bilanciamento

In un albero AVL ogni nodo contiene un campo aggiuntivo che distingue tre situazioni possibili:

- 1 Il sottoalbero sinistro è più profondo (di un livello) del destro;
- 0 Il sottoalbero sinistro e quello destro hanno la stessa profondità;
- +1 Il sottoalbero destro è più profondo (di un livello) del sinistro;

Come per tutti gli alberi bilanciati, la ricerca procede come per un albero binario semplice, mentre l'inserimento richiede delle considerazioni più sofisticate.

Vediamo una formulazione ricorsiva della procedura di inserimento.

```

Node *Insert(Node *root, Item *key,
             Bool *taller) {
    int cmp;
    if (root == NULL) {
        root=(Node *) malloc(sizeof(Node));
        assert(root!=NULL);
        root->key = key;
        root->left = root-> right = NULL;
        root->bal = 0;
        *taller=TRUE;
    } else {
        cmp = compare(key,root->key);
        if (cmp ==0) {
            update(root,key);
            *taller=FALSE;
        } else if (cmp == -1) {
            // LEFT INSERTION
            root->left = Insert(root->left,key,
                               *taller);

            if (*taller) {
                if (root->bal==1){
                    root = LeftBalance(root,taller);

```

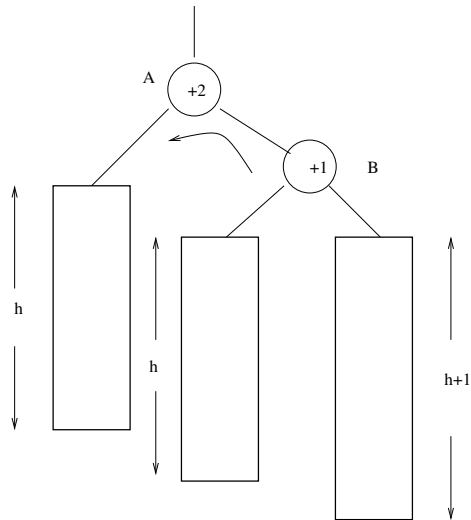
```

                } else if (root->bal==0){
                    root->bal = -1;
                } else if (root->bal==1){
                    root->bal = 0; *taller=FALSE;
                }
            }
        } else {
            // RIGHT INSERTIOIN
            root->right = Insert(root->right,key,
                                *taller);

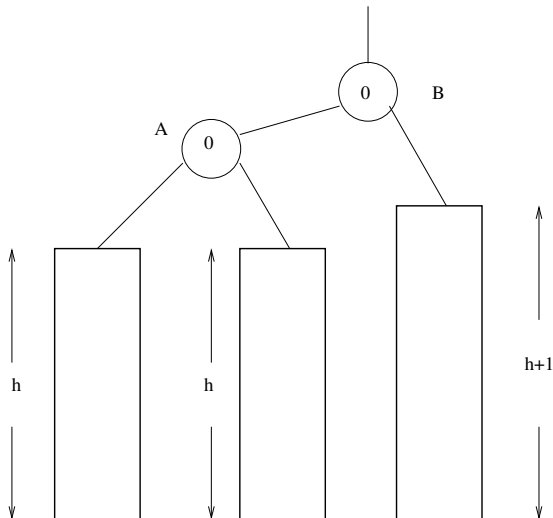
            if (*taller) {
                if (root->bal==1){
                    root->bal = 0; *taller=FALSE;
                } else if (root->bal==0){
                    root->bal = +1;
                } else if (root->bal==1){
                    root = RightBalance(root,taller);
                }
            }
        }
    }
    return(root);
}

```

Consideriamo il caso l'inserimento è avvenuto nel sottoalbero di destra di un nodo il cui bilanciamento diventa $+2$, mentre la radice del sottoalbero ha bilanciamento $+1$.



Effettuando una *rotazione a sinistra* si ripristina il bilanciamento (in tempo $O(1)$).



Left

```
Node *RotateLeft(Node *p)
{
    Node *temp=p;

    assert(p!=NULL);
    assert(p->right!=NULL);

    temp = p->right;
    p->right = temp->left;
    temp->left = p;
    return(temp);
}
```

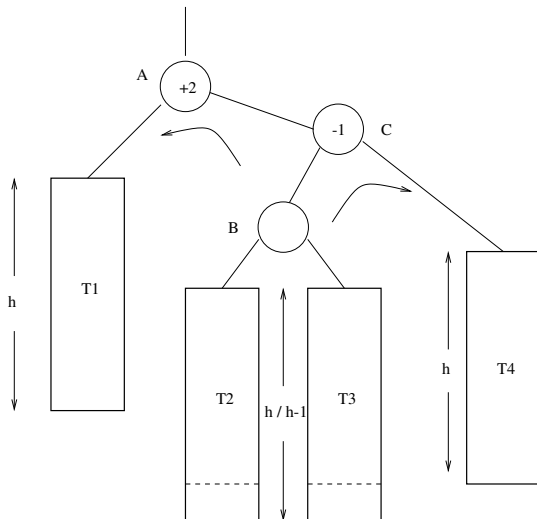
Right

```
Node *RotateRight(Node *p)
{
    Node *temp=p;

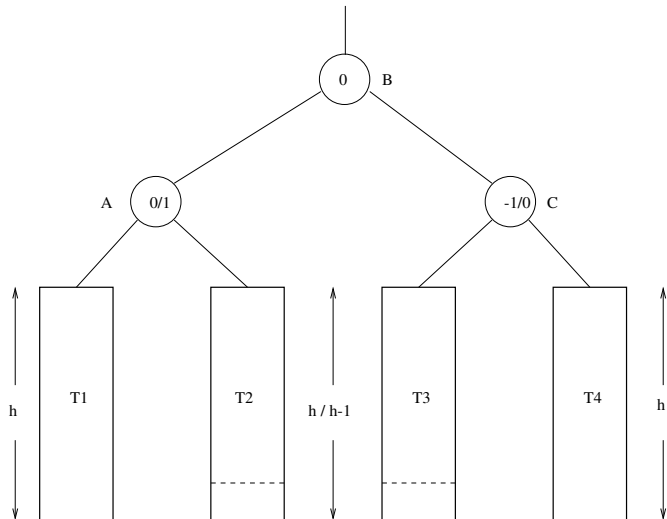
    assert(p!=NULL);
    assert(p->left!=NULL);

    temp = p->left;
    p->left = temp->right;
    temp->right = p;
    return(temp);
}
```

Ora invece consideriamo il caso l'inserimento è avvenuto nel sottoalbero di destra di un nodo il cui bilanciamento diventa $+2$, mentre la radice del sottoalbero ha bilanciamento -1 .



È necessaria una sequenza di *due* rotazioni, con un costo che comunque è $O(1)$.

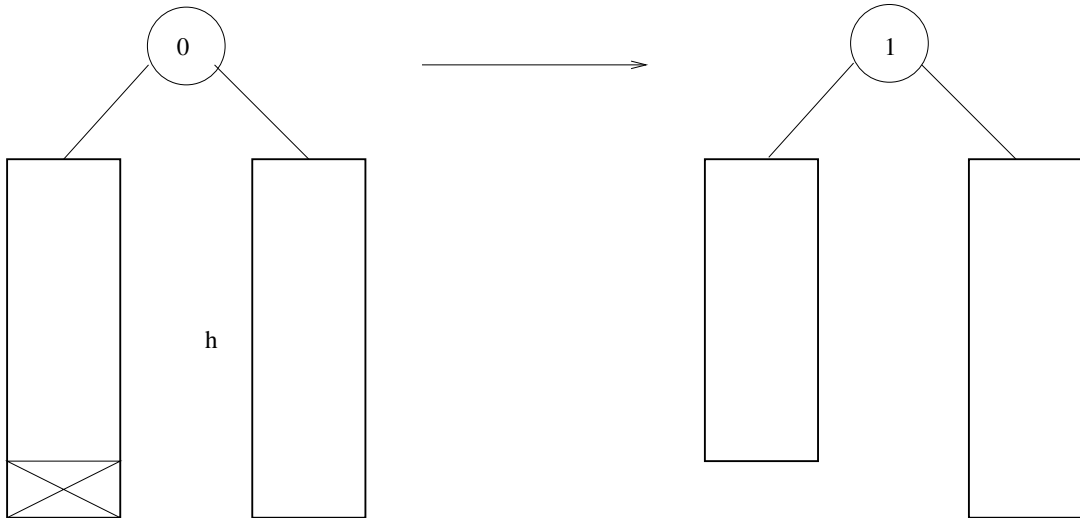


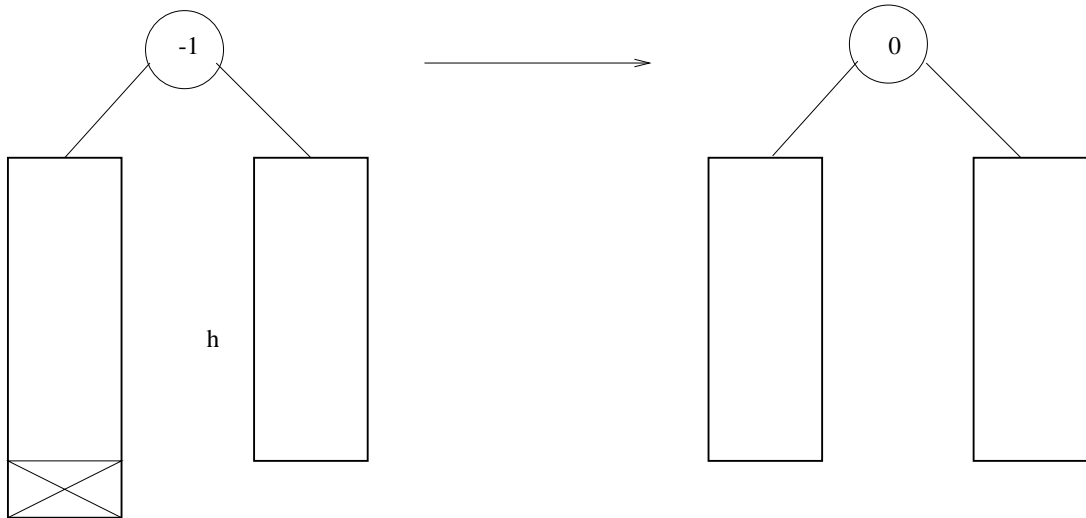
```
Node *RightBalance(Node *root, Bool *taller)
{
    Node *rs=root->right, *ls;
    assert(rs->bal != 0);
    if (rs->bal == +1) {
        root->bal=rs->bal=0;
        // Single rotation
        root=RotateLeft(root);
        *taller=FALSE;
    } else if (rs->bal == -1) {
```

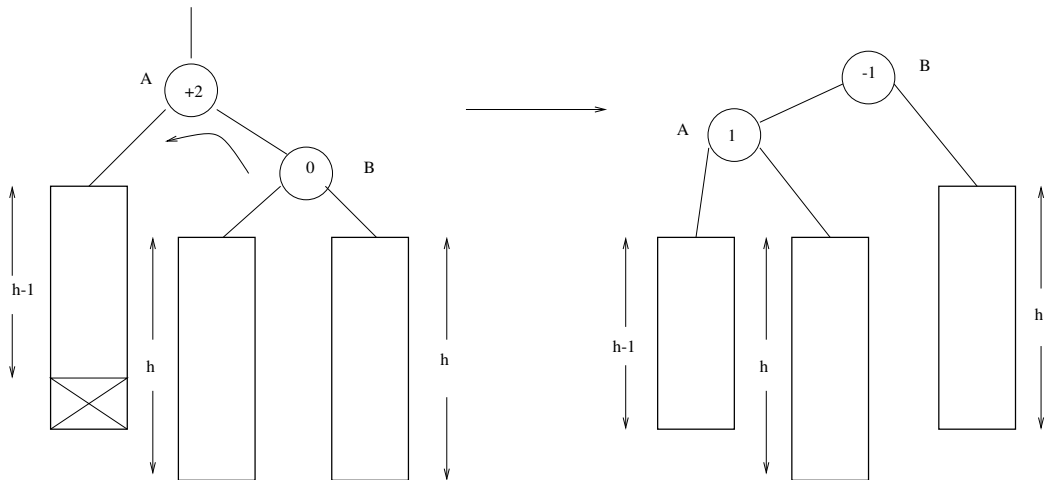
```
        // Double rotation
        ls=rs->left;
        if (ls->bal==+1){
            root->bal=-1;
            rs->bal=0;
        } else if (ls->bal==0){
            root->bal=rs->bal=0;
        } else if (ls->bal==-1){
            root->bal=0;
            rs->bal=+1;
        }
        ls->bal=0;
        root->right=RotateRight(rs);
        root=RotateLeft(root);
        *taller=FALSE;
    }
}
```

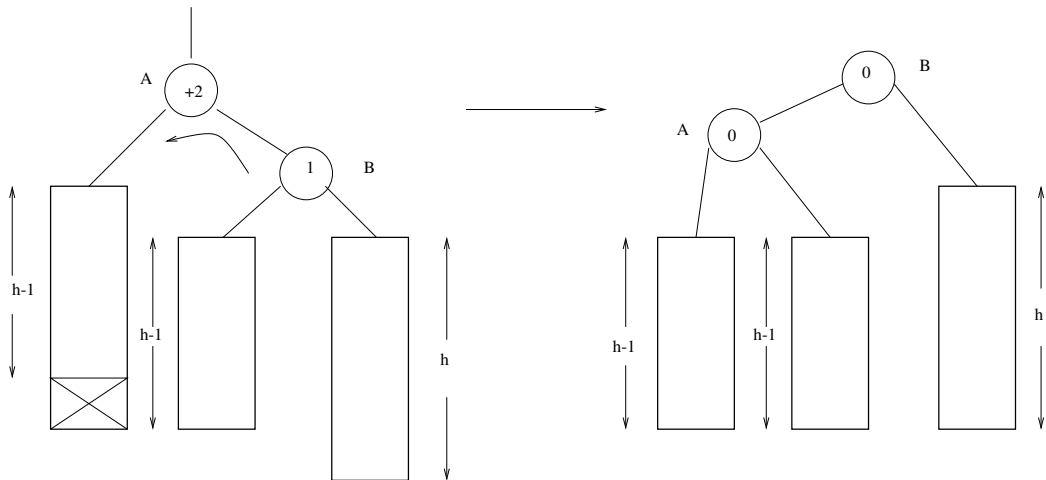

- ❶ Riduciamo il problema alla cancellazione di un nodo x che ha al più un figlio; altrimenti, si cerca il predecessore immediato muovendosi una volta a sinistra e poi sempre a destra; il nuovo nodo y non può avere un figlio destro. Sostituiamo y dove era x , poi cancelliamo y usandolo sempre al posto di x con:
- ❷ Colleghiamo il genitore di x con il singolo figlio di x ; il relativo sottoalbero si è accorciato e tracciamo con una variabile `shorter` che parte da `TRUE`;
- ❸ Si ripetono i passi seguenti per tutti i nodi p dal genitore di x fino alla radice, ma se `shorter` diventa `FALSE` si esce;
 - Caso 1: Il nodo p ha bilanciamento 0: si fa diventare $+1$ o -1 a seconda di quale sottoalbero si era accorciato, e `shorter=FALSE`;
 - Caso 2: Il nodo p è sbilanciato, ma è stato accorciato il sottoalbero più alto; il bilanciamento diventa 0 e `shorter=TRUE`;

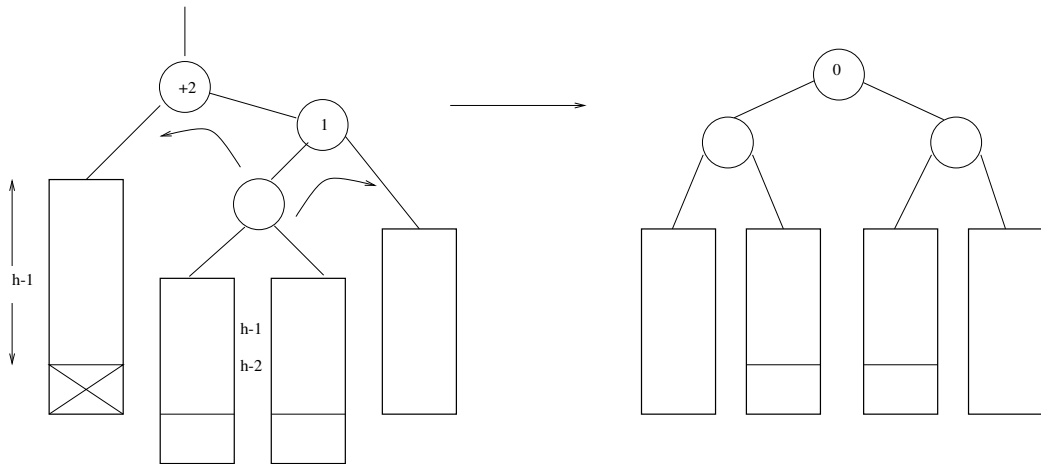
- Caso 3: Il nodo p è sbilanciato ed è stato accorciato il sottoalbero più basso; sia q la radice del sottoalbero più alto:
 - Caso 3a: il nodo q è bilanciato: si applica una rotazione singola, si aggiusta il bilanciamento di p e q e `shorter=FALSE`;
 - Caso 3b: il nodo q è sbilanciato come p : si applica una rotazione singola, si aggiusta il bilanciamento di p e q e `shorter=TRUE`;
 - Caso 3c: il nodo q è sbilanciato all'opposto di p : si applica una rotazione doppia, si aggiusta il bilanciamento della nuova radice a 0 e gli altri opportunamente, e `shorter=TRUE`;











Teorema (Massima altezza di un albero AVL)

L'altezza h di un albero bilanciato con N nodi interni è limitata da

$$\log(N + 1) < h < 1.4404 \log(N + 2) - 0.3277.$$

Dimostrazione (Parte 1).

- 1 Il numero di nodi esterni (foglie) di un albero di altezza h è chiaramente $\leq 2^h$, per cui $N + 1 \leq 2^h$ ovvero $h \geq \lceil \log(N + 1) \rceil$.



Dimostrazione (Parte 2).

- ① Si consideri un albero bilanciato di altezza h con il *minimo* numero possibile di nodi; allora un sottoalbero della radice (p.es. il sinistro) deve avere altezza $h - 1$, e l'altro $h - 2$, e devono essere entrambi con numero minimo di nodi; quindi

$$N_h = N_{h-1} + N_{h-2} + 1,$$

ossia si ha un *albero di Fibonacci*; pertanto

$$N \geq F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2,$$

(vedi D. Knuth: TAOCP, vol 3).



Gli alberi rosso-neri (Red-Black) utilizzano un attributo (*colore*) aggiuntivo per ogni nodo, attributo che può appunto prendere i valori *rosso* oppure *nero*.

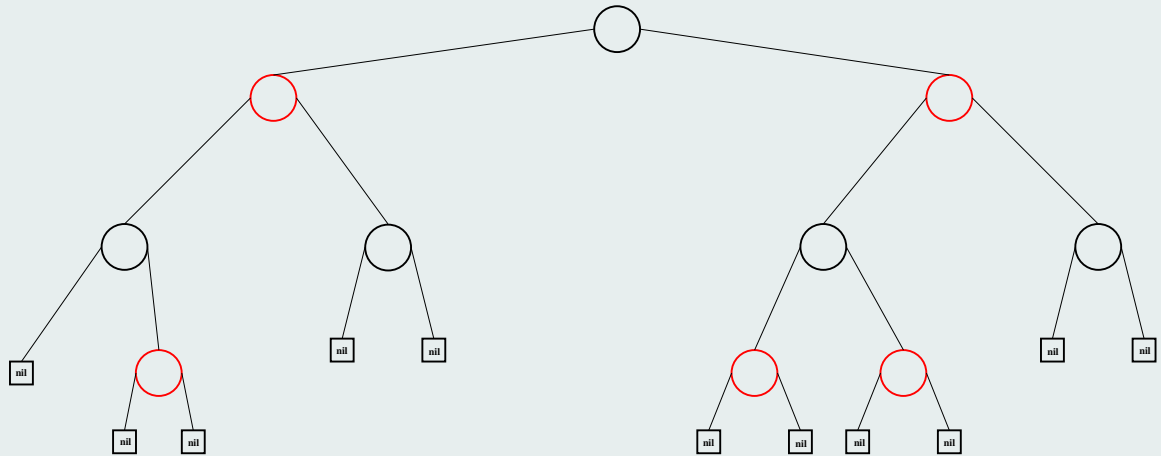
Proprietà costitutive degli alberi RN

- 1 La radice è nera;
- 2 Tutte le foglie sono nere;
- 3 Entrambi i figli di un nodo rosso sono neri;
- 4 Tutti i cammini da un nodo u ad una foglia nell'albero radicato in u contengono lo stesso numero di nodi neri.

La presentazione e le dimostrazioni di correttezza si semplificano assumendo che

- ① Tutte le foglie sono memorizzate esplicitamente,
- ② Le foglie non contengono dati;
- ③ Sono in numero tale che tutti i nodi (interni) abbiano esattamente due figli (che possono anche essere foglie).

Esempio



Definizione

La altezza nera $b(u)$ è il numero di nodi neri da u (escluso) ad una (qualsiasi) foglia.

Lemma

Il sottoalbero di radice u contiene almeno $N = 2^{b(u)} - 1$ nodi interni.

Dimostrazione.

Per induzione sulla altezza h (totale) dell'albero.

- Se $h = 0$, allora siamo su una foglia, ed il numero di nodi interni è $N = 0 = 2^0 - 1$.
- Se $h > 0$ allora siamo su un nodo interno che ha 2 figli; se il figlio è rosso allora la sua altezza nera è eguale a $b(u)$, altrimenti è $b(u) - 1$; per induzione, il numero di nodi interni del sottoalbero radicato in ciascun figlio è almeno $2^{b(u)-1} - 1$, e quindi il numero totale di nodi interni (contando anche u) è

$$N \geq 2^{b(u)-1} - 1 + 2^{b(u)-1} - 1 + 1 = 2 \times 2^{b(u)-1} - 1 = 2^{b(u)} - 1.$$



Teorema

La lunghezza del cammino radice-foglia più lungo non supera il doppio di quella del più corto

$$\max\{l(u) : u.left = u.right = \mathbf{nil}\} \leq 2 \min\{l(u) : u.left = u.right = \mathbf{nil}\}$$

Teorema

La lunghezza del cammino radice-foglia più lungo non supera il doppio di quella del più corto

$$\max\{l(u) : u.\text{left} = u.\text{right} = \mathbf{nil}\} \leq 2 \min\{l(u) : u.\text{left} = u.\text{right} = \mathbf{nil}\}$$

Dimostrazione.

- Tutti i cammini dalla radice alle foglie hanno la stessa altezza nera $b(r)$;
- Nessun cammino può contenere due nodi rossi consecutivi;
- Pertanto, i nodi rossi sul cammino più lungo sono al più tanti quanti quelli neri;
- Il cammino più corto ha lo stesso numero di nodi neri del più lungo;
- Pertanto il cammino più lungo è al più il doppio del più corto.



Lemma

L'altezza di un albero rosso-nero con N nodi interni è al più $2 \log(N + 1)$.

Dimostrazione.

Dai lemmi precedenti:

- Altezza nera: $N \geq 2^{b(u)} - 1$;
- Altezza complessiva: $N \geq 2^{h/2} - 1$;

da cui

$$2^{h/2} \leq N + 1$$

$$h/2 \leq \log(N + 1)$$

$$h \leq 2 \log(N + 1)$$



Quindi, gli alberi rosso-neri possono essere *sbilanciati* di un *fattore* 2, ma vale comunque che

La ricerca in un albero rosso-nero ha una complessità $O \log(N)$ nel numero di nodi N .

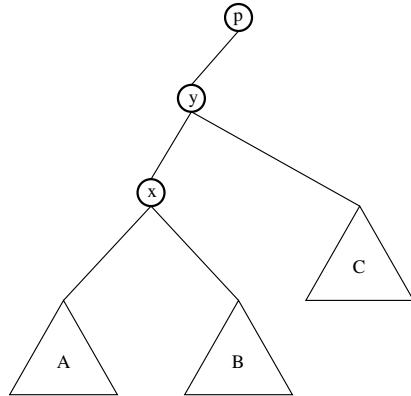
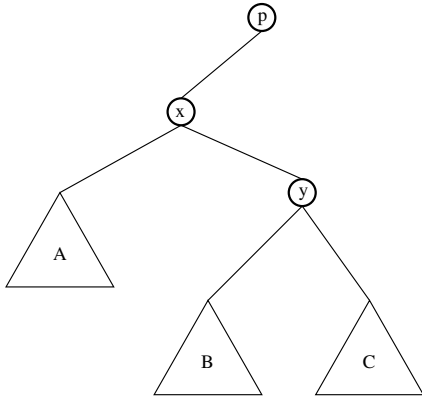
Le operazioni di ricerca di un nodo, del successore, del massimo o del minimo sono realizzabili esattamente con lo stesso codice che per un albero binario.

Modifica di un albero rosso-nero

Le operazioni di inserimento e cancellazione possono alterare la struttura dati; il nostro problema è di trovare dei metodi che ripristino le proprietà di un albero rosso-nero, con una complessità accettabile ($O \log(N)$).

Ripristino di un albero rosso-nero

- Cambio di colore di un nodo;
- “Rotazione”.



Rotazione a sinistra (esercizio: rot. destra)

Function *Tree RotateLeft* (*Tree x*)

```
Tree y  $\leftarrow$  x.right;  
Tree p  $\leftarrow$  x.parent;  
x.right  $\leftarrow$  y.left;  
if y.left  $\neq$  nil then y.left.parent  $\leftarrow$  x;  
y.left  $\leftarrow$  x;  
x.parent  $\leftarrow$  y;  
y.parent  $\leftarrow$  p;  
if p  $\neq$  nil then  
    if p.left = x then  
        p.left  $\leftarrow$  y;  
    else  
        p.right  $\leftarrow$  y;
```

Risultato y

Function *insertNode* (*Tree t*, *item x*, *item v*)

Tree $p \leftarrow \text{nil}$;

Tree $n \leftarrow u \leftarrow t$;

while $u \neq \text{nil}$ and $u.\text{key} \neq x$ **do**

$p \leftarrow u$;

$u \leftarrow \text{if } (x < u.\text{key}, u.\text{left}, u.\text{right})$;

if $u \neq \text{nil}$ **then**

$u.\text{value} \leftarrow v$;

else

 Tree $n \leftarrow \text{Tree}(x, v)$;

 link (p, n, x);

 balanceInsert (n);

while $n.\text{parent} \neq \text{nil}$ **do**

$n \leftarrow n.\text{parent}$;

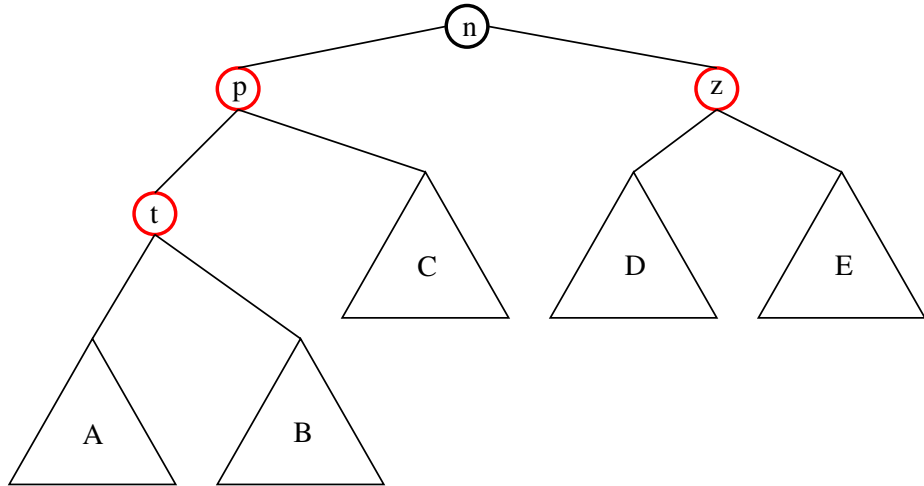
Risultato n

Function *balanceInsert* (*Tree t*)

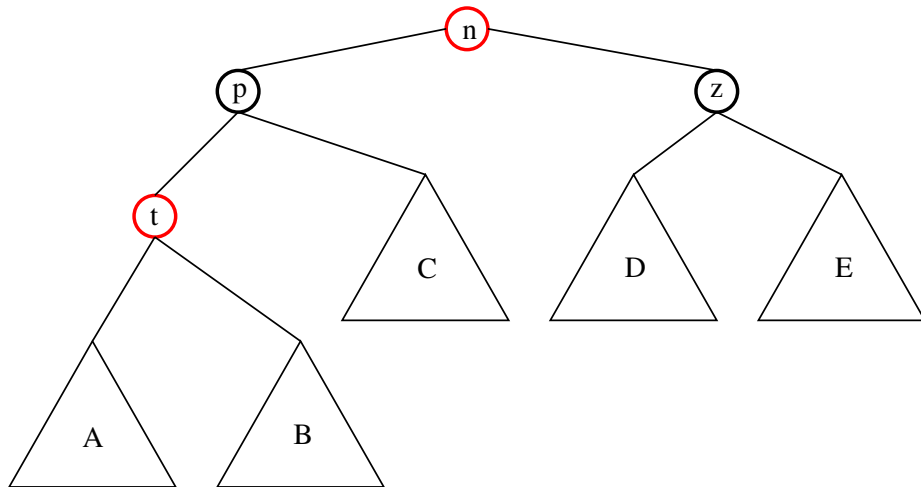
```
t.color  $\leftarrow$  RED;
while t  $\neq$  nil do
    Tree p  $\leftarrow$  t.parent;
    Tree n  $\leftarrow$  if(p  $\neq$  nil, p.parent, nil);
    Tree z  $\leftarrow$  if(n = nil, nil, if(n.left = p, n.right, n.left));
    if p = nil then
        | t.color  $\leftarrow$  BLACK; t  $\leftarrow$  nil;
    else if p.color = BLACK then
        | t  $\leftarrow$  nil;
    else if z.color = RED then
        | p.color  $\leftarrow$  z.color  $\leftarrow$  BLACK; n.color  $\leftarrow$  RED; t  $\leftarrow$  n;
    else
        if (t = p.right) and (p = n.left) then
            | rotateLeft(p); t  $\leftarrow$  p;
        else if (t = p.left) and (p = n.right) then
            | rotateRight(p); t  $\leftarrow$  p;
        else
            if (t = p.left) and (p = n.left) then rotateRight(p) ;
            else if (t = p.right) and (p = n.right) then rotateLeft(p) ;
            p.color  $\leftarrow$  BLACK; n.color  $\leftarrow$  RED; t  $\leftarrow$  nil;
```

Nei primi due casi:

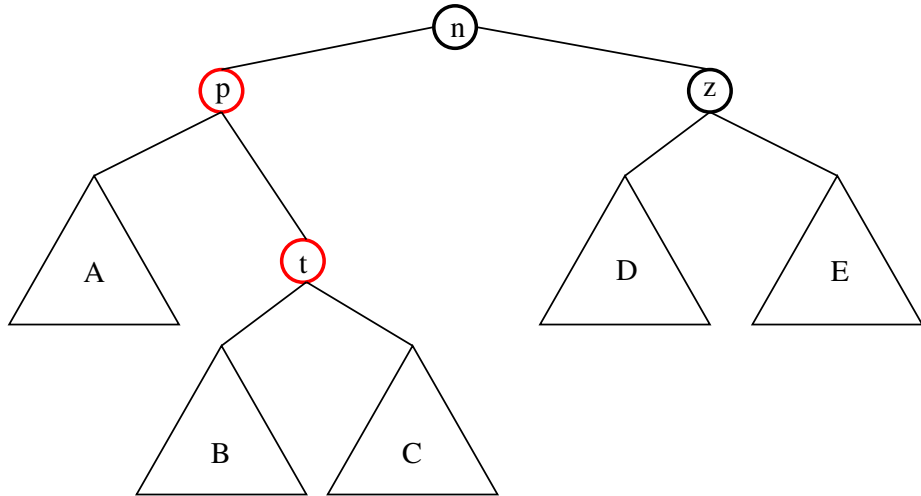
- 1 Il nuovo nodo (che nasce rosso) è il primo (non ha un genitore) quindi basta colorarlo di nero;
- 2 Il nuovo nodo (che nasce rosso) ha un genitore nero, quindi tutti i vincoli sono rispettati e non si deve fare niente.



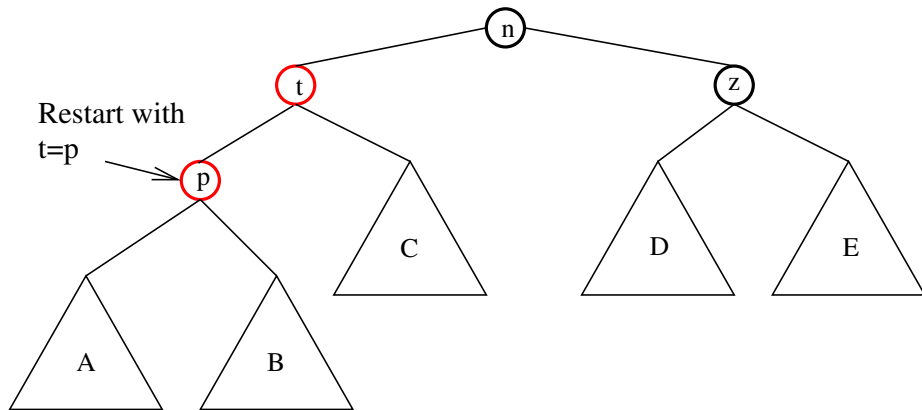
Si aggiusta trasformando in ...



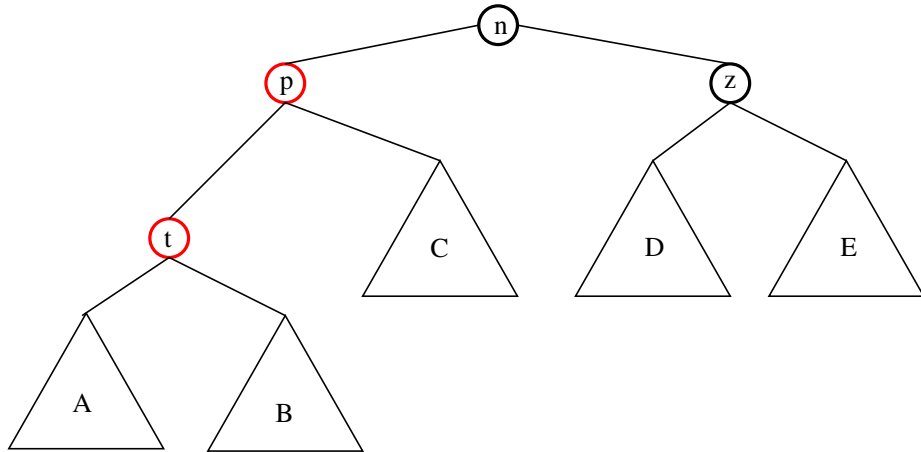
Che però potrebbe necessitare di ulteriore aggiustamento, la procedura deve proseguire dal nodo n .



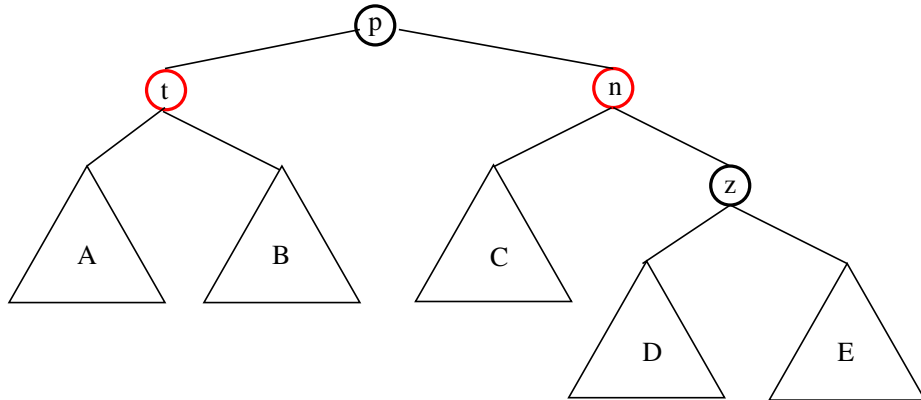
Si aggiusta trasformando in ...



Si ricomincia con $t \leftarrow p$ e ci si ritrova al caso successivo



Si aggiusta trasformando in ...



Function *removeNode* (*Tree T*, *item x*)

Tree $u \leftarrow \text{lookupNode}(T, x)$;

if $u \neq \text{nil}$ **then**

if $u.\text{left} \neq \text{nil}$ **and** $u.\text{right} \neq \text{nil}$ **then**

 Tree $s \leftarrow u.\text{right}$;

while $s.\text{left} \neq \text{nil}$ **do**

$s \leftarrow s.\text{left}$;

$u.\text{key} \leftarrow s.\text{key}$; $u.\text{value} \leftarrow s.\text{value}$; $u \leftarrow s$; $x \leftarrow s.\text{key}$;

Tree t ;

if $u.\text{left} \neq \text{nil}$ **and** $u.\text{right} = \text{nil}$ **then**

$t \leftarrow u.\text{left}$;

else

$t \leftarrow u.\text{right}$;

 link($u.\text{parent}$, t , x);

if $u.\text{color} = \text{BLACK}$ **then** $\text{balanceDelete}(T, t)$;

if $u.\text{parent} = \text{nil}$ **then** $T \leftarrow t$;

 delete u ;

while $T.\text{parent} \neq \text{nil}$ **do**

$T \leftarrow T.\text{parent}$;

 return T ;

Function *balanceDelete* (*Tree T*, *Tree t*)

while $t \neq T$ and $t.color = BLACK$ **do**

Tree p $\leftarrow t.parent$;

if $t = p.left$ **then**

Tree f $\leftarrow p.right$;

Tree ns $\leftarrow f.left$;

Tree ns $\leftarrow f.right$;

if ($f.color = RED$) **then**

$p.color \leftarrow RED$; $f.color \leftarrow BLACK$; *rotateLeft*(p);

else

if ($ns.color = nd.color = BLACK$) **then**

$f.color \leftarrow RED$; $t \leftarrow p$;

else if ($ns.color = RED$ and $nd.color = BLACK$) **then**

$ns.color \leftarrow BLACK$; $f.color \leftarrow RED$; *rotateRight*(f);

else if $nd.color = RED$ **then**

$f.color \leftarrow RED$; $p.color \leftarrow BLACK$;

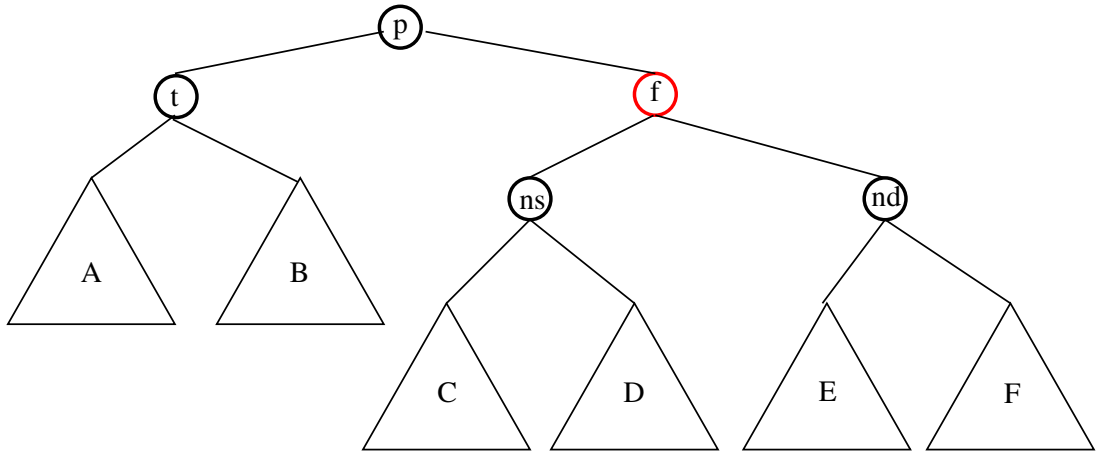
$nd.color \leftarrow BLACK$; *rotateLeft*(p);

$t \leftarrow T$;

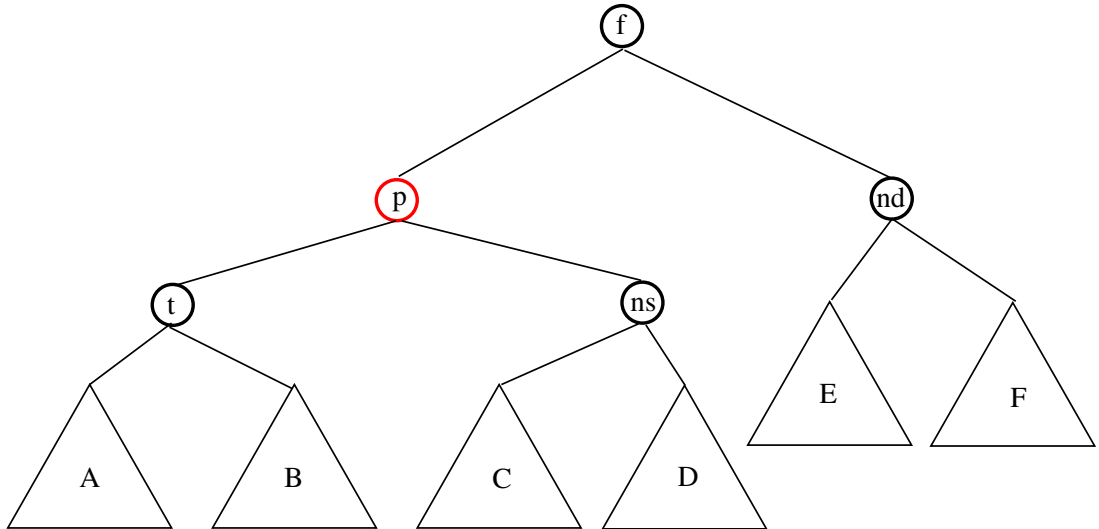
else

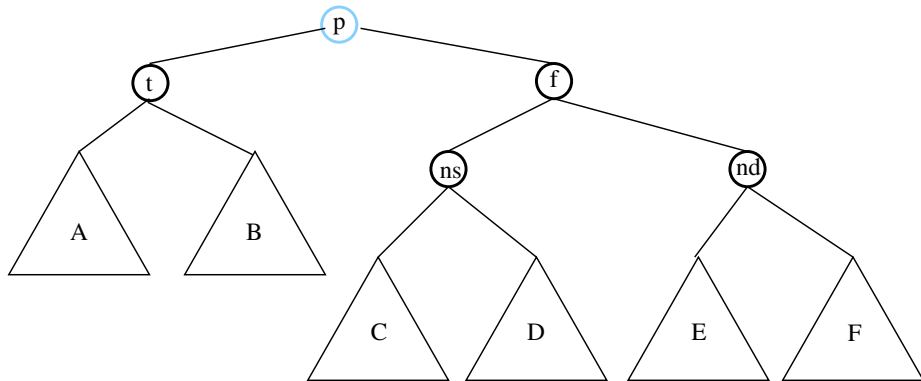
 % Codice speculare al precedente;

if $t \neq nil$ **then** $t.color = BLACK$;

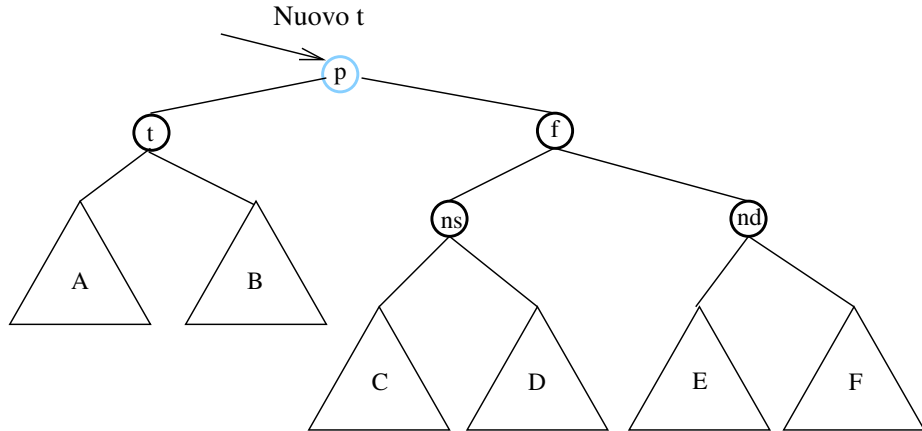


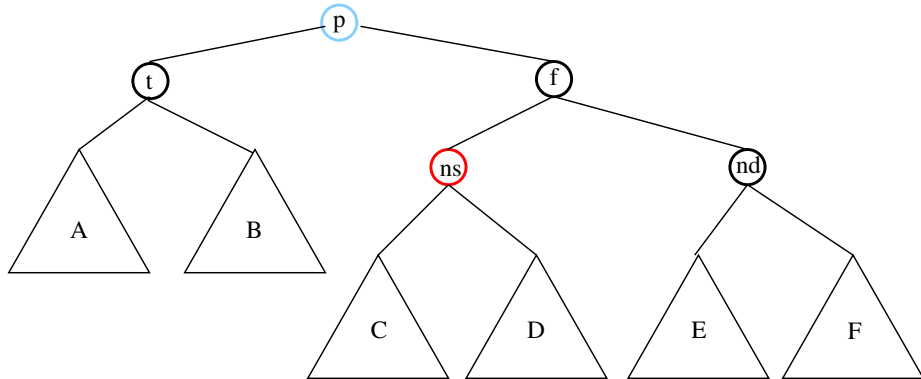
Si aggiusta trasformando in ...



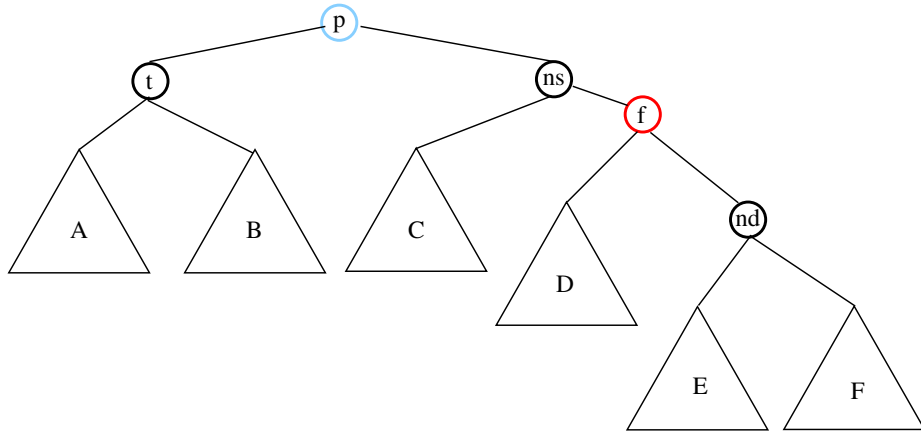


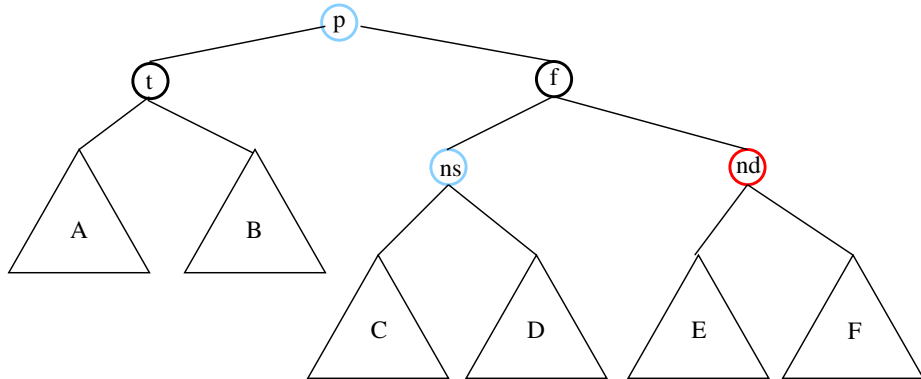
Si aggiusta trasformando in ...



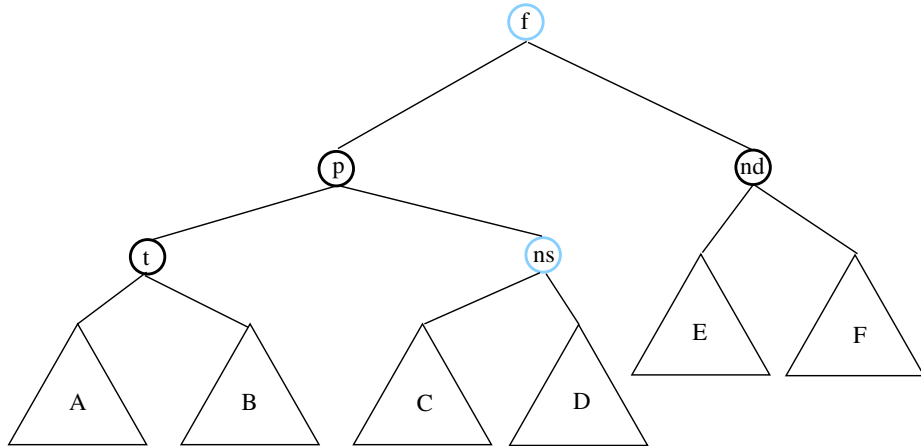


Si aggiusta trasformando in ...





Si aggiusta trasformando in ...



Si pone $t = T$

Rotazione a sinistra

