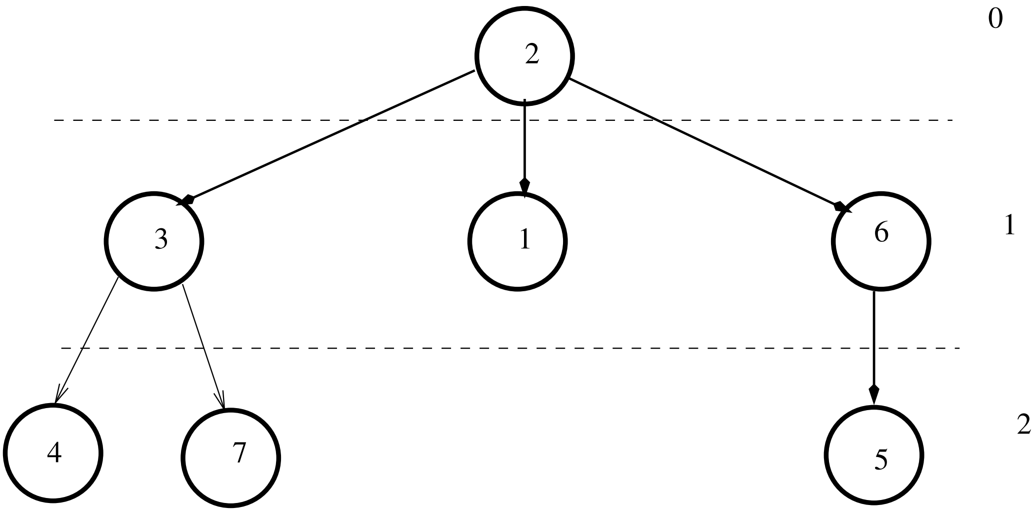
Strutture ad albero

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;
- Ogni nodo contiene delle informazioni (determinate dalla applicazione);
- Ogni nodo (tranne la radice) ha uno ed un solo *nodo padre*, e può avere degli *antenati*;
- La radice non ha un padre;
- 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*.


0
2
3
1
6
1
4
7
5
2

- 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;

Visita di un albero

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 di un albero

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

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

Visita in *inordine*

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

Visita in *ampiezza*

O per livelli; necessita di una *coda* ausiliaria

```
t ≠ nil;
Queue Q ← Queue();
Q.enqueue(t);
while  not Q.isEmpty() do
    u ← Q.dequeue();
    visita (u);
    Tree u ← u.leftmostChild();
    while  u ≠ nil do
        Q.Enqueue(u);
        u ← u.rightSibling();
```

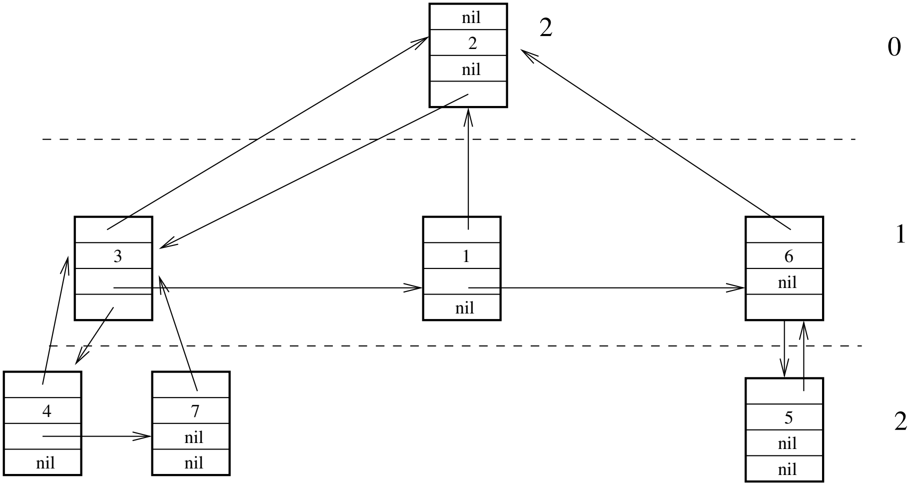
Dimostrare che tutti i nodi del livello k vengono visitati prima dei nodi del livello $k+1$.

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 padre;
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;


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

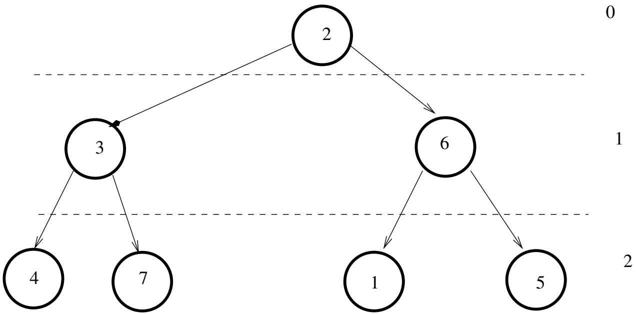
```
Node parent;
Node child;
Node sibling;
Item value;
Function Tree (item v)
    Tree t ← new Tree;
    t.value ← v;
    t.parent ← t.child ← t.sibling ← nil;
    Risultato t
Function insertChild (Tree t)
    t.parent ← this;
    t.sibling ← child;
    child ← t;
Function insertSibling (Tree t)
    t.parent ← parent;
    t.sibling ← sibling;
    sibling ← t;
```

```
Function deleteChild ()
    Node newChild ← child.rightSibling();
    delete (child);
    child ← newChild;
Function deleteSibling ()
    Node newSibling ← sibling.rightSibling();
    delete (sibling);
    sibling ← newSibling;
Function delete (Tree t)
    Node u ← t.leftmostChild();
    while u ≠ nil do
        Tree next ← u.rightSibling();
        delete (u);
        u ← 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 padre è in posizione i i due figli sono nelle posizioni $2i$ e $2i+1$.
La realizzazione più comune fa uso di tre puntatori, padre, figlio sinistro e figlio destro.

```
Node parent;
Node left;
Node right;
Item value;
Function Tree (item v)
    Tree t ← new Tree;
    t.value ← v;
    t.parent ← nil ← t.left ← t.right ← nil;
    Risultato t
Function insertLeft (Tree t)
    t.parent ← this;
    left ← t;
Function insertRight (Tree t)
    t.parent ← this;
    right ← t;
```

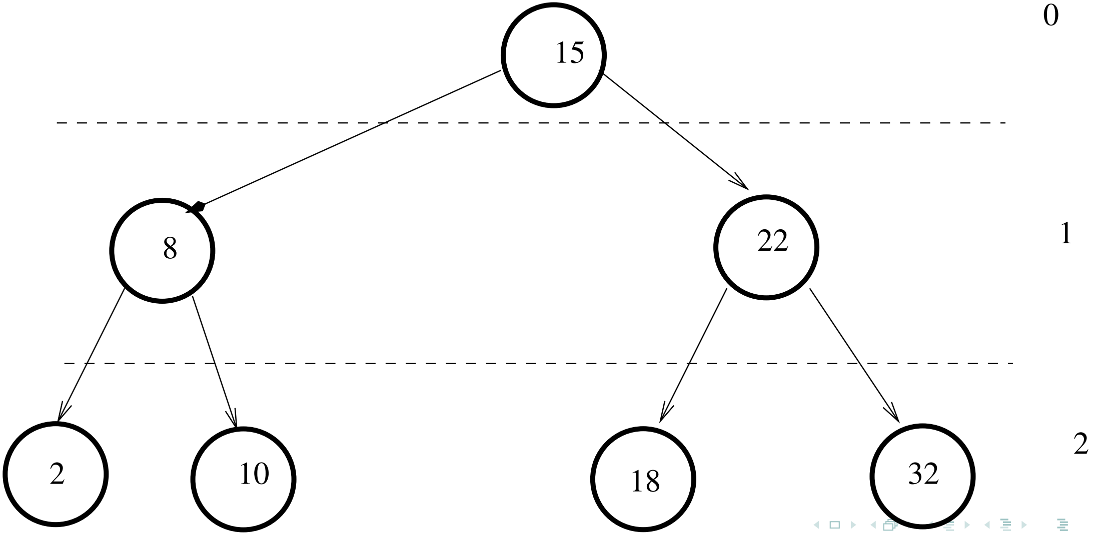
```
Function deleteLeft ()
    if left ≠ nil then
        left.deleteLeft();
        left.deleteRight();
        delete (left);
        left ← nil;
Function deleteRight ()
    if right ≠ nil then
        right.deleteLeft();
        right.deleteRight();
        delete (right);
        right ← nil;
```

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


15
8
22
2
10
18
32
0
1
2

```
Function min (Tree T)
    while T.left ≠ nil do
        T ← T.left;
    Risultato T
Function max (Tree T)
    while T.right ≠ nil do
        T ← T.right;
    Risultato T
Function lookupNode (Tree T, item x)
    while T ≠ nil and T.key ≠ x do
        T ← if (x < T.key, T.left, T.right);
    Risultato T
```

```
Function link (Tree p, Tree u, item x)
    if u ≠ nil then
        u.parent ← p;
    if p ≠ nil then
        if x < p.key then
            p.left ← u;
        else
            p.right ← u;
```

```
Function insertNode (Tree t, item x, item v)
    Tree p ← nil;
    Tree u ← T;
    while u ≠ nil and u.key ≠ x do
        p ← u;
        u ← if (x < u.key, u.left, u.right);
    if u ≠ nil then
        u.value ← v;
    else
        Tree n ← 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 ← lookupNode(T, x);
    if u ≠ nil then
        if u.left ≠ nil and u.right ≠ nil then
            Tree s ← u.right;
            while s.left ≠ nil do
                s ← s.left;
            u.key ← s.key, u.value ← s.value;
            u ← s, x ← s.key;
        Tree t;
        if u.left ≠ nil and u.right = nil then
            t ← u.left;
        else
            t ← u.right;
        link(u.parent, t, x);
        if u.parent = nil then T ← t;
        delete u;
    return T;
```

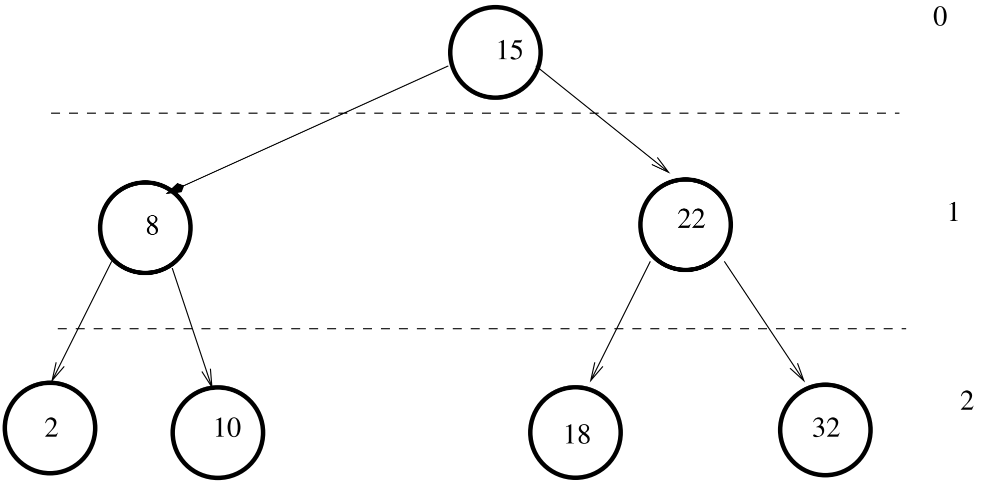
Function *Tree successorNode* (*Tree t*)

```
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"



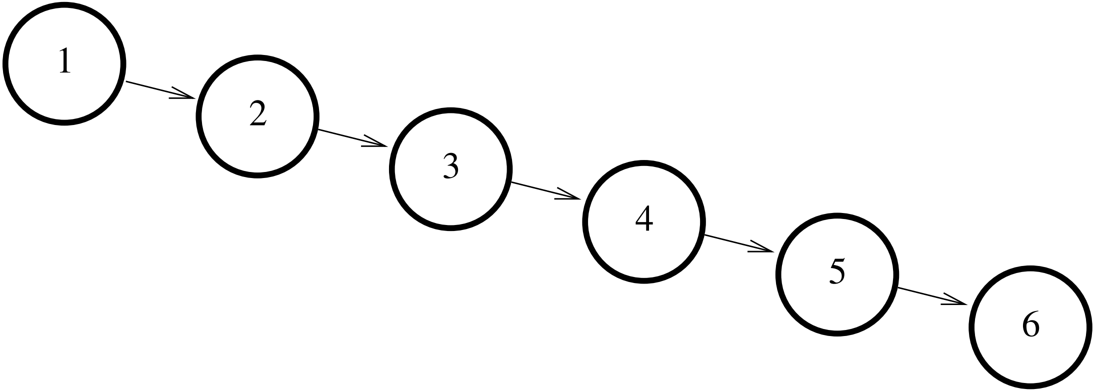
Quale sequenza di inserimenti potrebbe portare a questo albero?

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 (o k-bilanciati):

Alberi AVL

B-alberi

Alberi 2-3

Alberi rosso-neri

Alberi rosso-neri

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

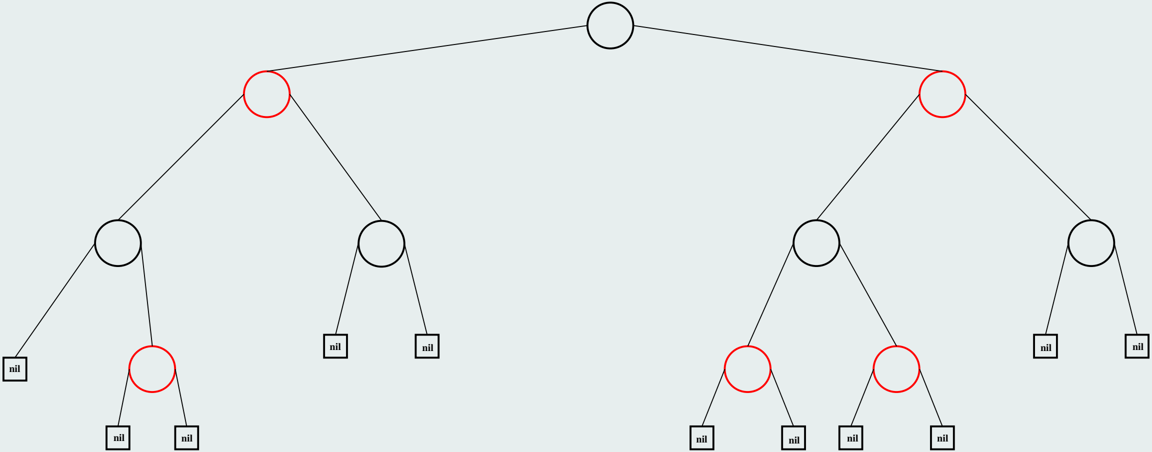
Proprietà costitutive degli alberi RN

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

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

$\square$

Teorema

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

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

Teorema

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

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

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.

$\square$

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

$$\begin{aligned} 2^{h/2} &\leq N+1 \\ h/2 &\leq \log(N+1) \\ h &\leq 2\log(N+1) \end{aligned}$$

□

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.

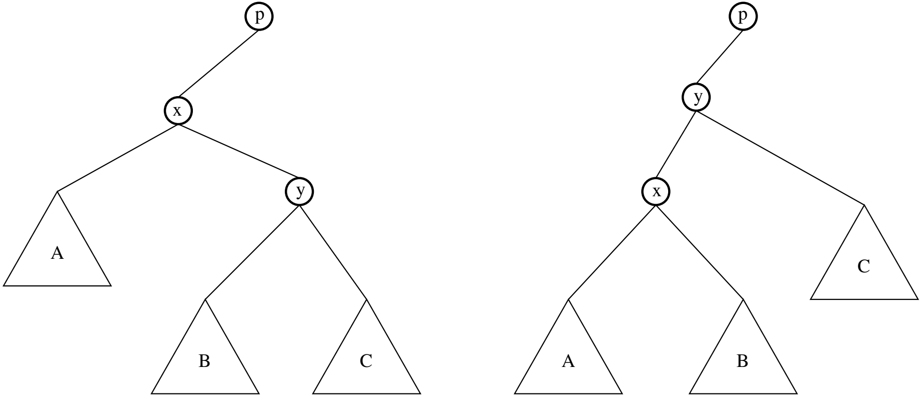
Le operazioni di ricerca di un nodo, del successore, del massimo o del minimo sono realizzabili esattamente con lo stesso codice che per un albero binario.

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 ← x.right;
    Tree p ← x.parent;
    x.right ← y.left;
    if y.left ≠ nil then y.left.parent ← x;
    y.left ← x;
    x.parent ← y;
    y.parent ← p;
    if p ≠ nil then
        if p.left = x then
            p.left ← y;
        else
            p.right ← y;
    Risultato y
```

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

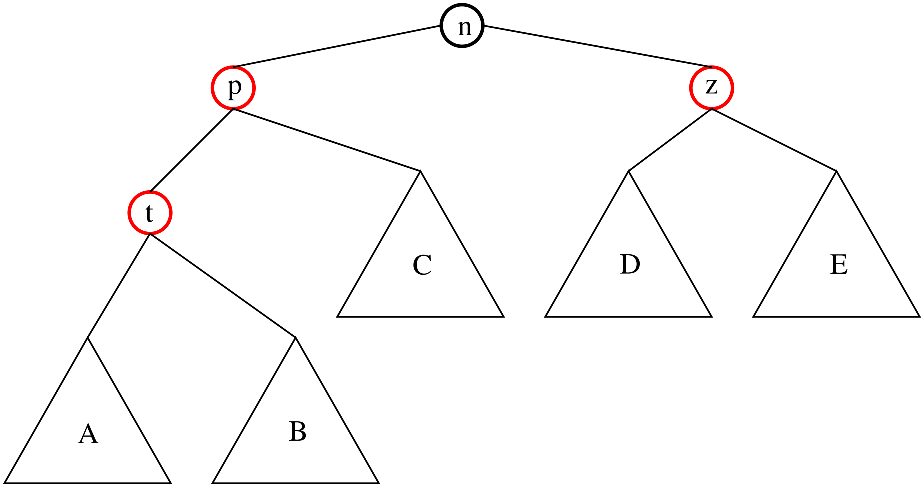
```
Tree p ← nil;
Tree n ← u ← t;
while u ≠ nil and u.key ≠ x do
    p ← u;
    u ← if (x < u.key, u.left, u.right);
if u ≠ nil then
    u.value ← v;
else
    Tree n ← Tree(x, v);
    link (p, n, x);
    balanceInsert (n);
while n.parent ≠ nil do
    n ← n.parent;
Risultato n
```

```
Function balanceInsert (Tree t)
    t.color ← RED;
    while t ≠ nil do
        Tree p ← t.parent;
        Tree n ← if(p ≠ nil, p.parent, nil);
        Tree z ← if(n = nil, nil, if(n.left = p, n.right, n.left));
        if p = nil then
            t.color ← BLACK; t ← nil;
        else if p.color = BLACK then
            t ← nil;
        else if z.color = RED then
            p.color ← z.color ← BLACK; n.color ← RED; t ← n;
        else
            if (t = p.right) and (p = n.left) then
                rotateLeft(p); t ← p;
            else if (t = p.left) and (p = n.right) then
                rotateRight(p); t ← 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 ← BLACK; n.color ← RED; t ← nil;
```

Alberi rosso-neri: ripristino su inserzione

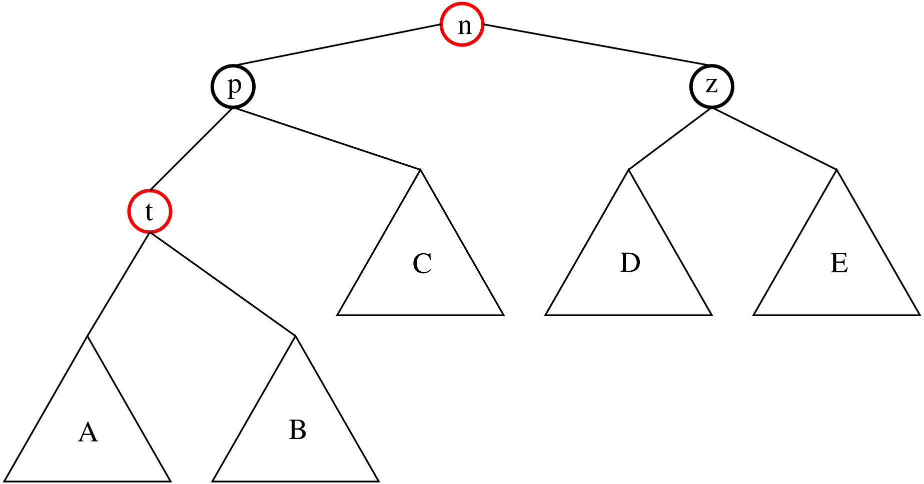
Nei primi due casi:

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



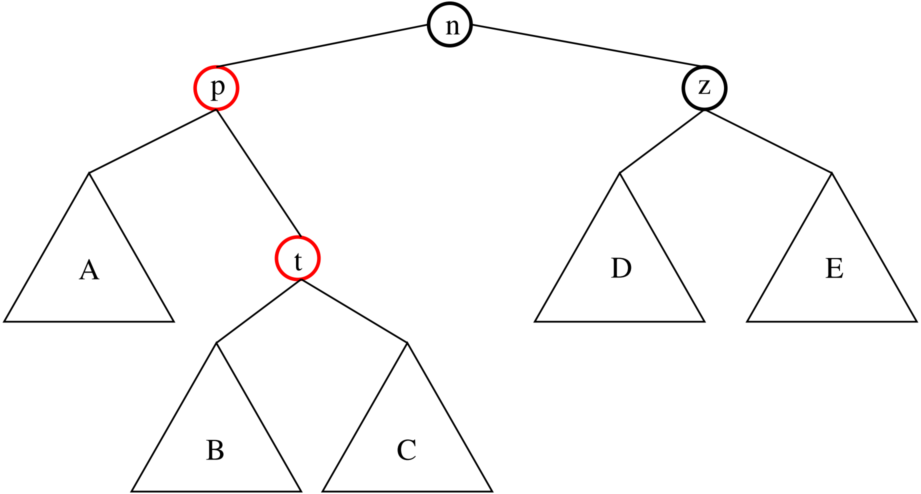
Si aggiusta trasformando in ...

Alberi rosso-neri: ripristino su inserzione caso 3



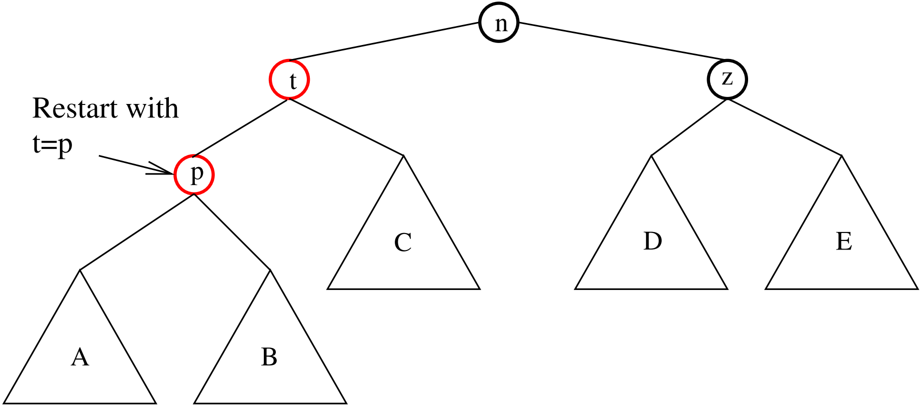
Che però potrebbe necessitare di ulteriore aggiustamento, la procedura deve proseguire dal nodo n.

Alberi rosso-neri: ripristino su inserzione caso 4

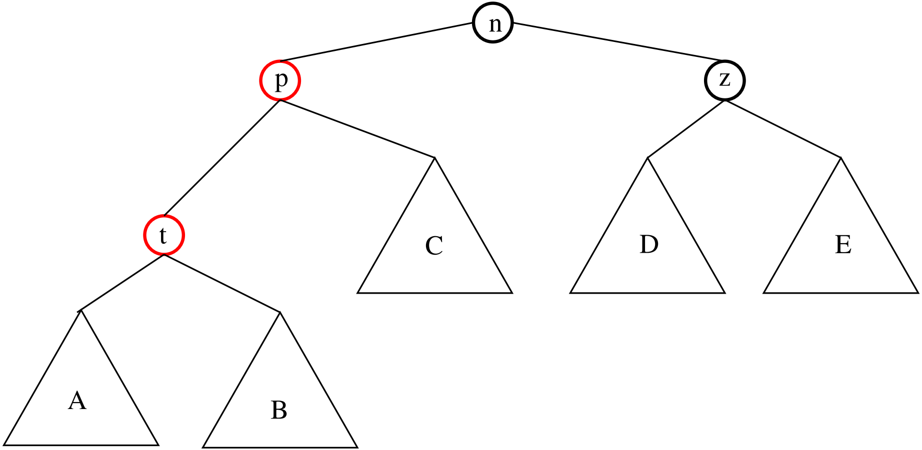


Si aggiusta trasformando in ...

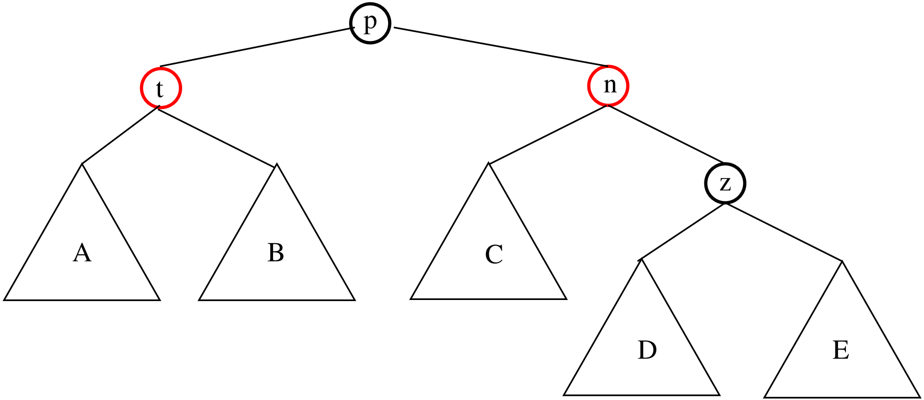
Alberi rosso-neri: ripristino su inserzione caso 4



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



Si aggiusta trasformando in ...

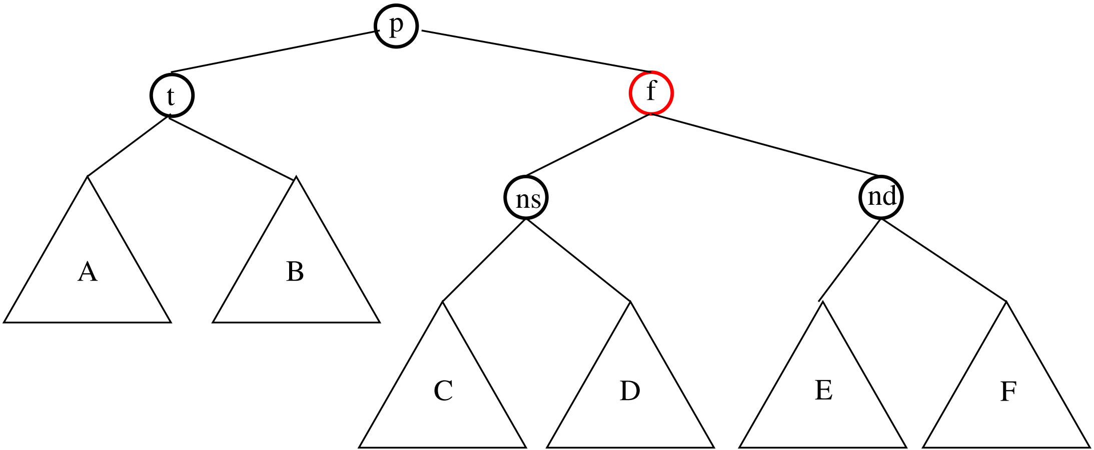

p
t
n
z
A
B
C
D
E

```
Function removeNode (Tree T, item x)
    Tree u ← lookupNode(T, x);
    if u ≠ nil then
        if u.left ≠ nil and u.right ≠ nil then
            Tree s ← u.right;
            while s.left ≠ nil do
                s ← s.left;
            u.key ← s.key; u.value ← s.value; u ← s; x ← s.key;
        Tree t;
        if u.left ≠ nil and u.right = nil then
            t ← u.left;
        else
            t ← u.right;
        link(u.parent, t, x);
        if u.color=BLACK then  balanceDelete(T, t) ;
        if u.parent = nil then  T ← t ;
        delete u;
    while T.parent ≠ nil do
        T ← T.parent;
    return T;
```

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

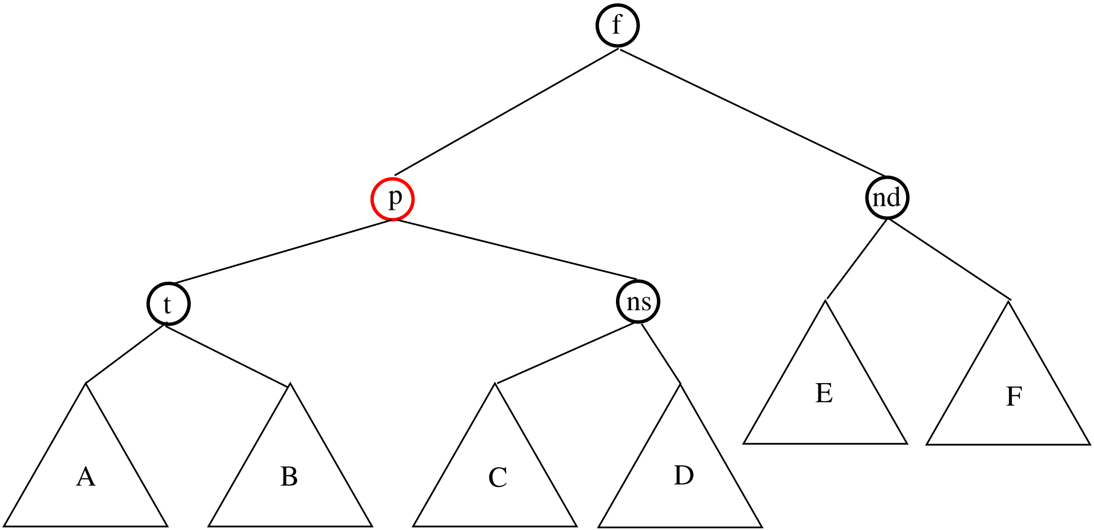
```
while t ≠ T and t.color = BLACK do
    Tree p ← t.parent;
    if t = p.left then
        Tree f ← p.right;
        Tree ns ← f.left;
        Tree ns ← f.right;
        if (f.color = RED) then
            p.color ← RED; f.color ← BLACK; rotateLeft(p);
        else
            if (ns.color = nd.color = BLACK) then
                f.color ← RED; t ← p;
            else if (ns.colorRED and nd.color = BLACK) then
                ns.color ← BLACK; f.color ← RED; rotateRight(f);
            else if nd.color = RED then
                f.color ← RED; p.color ← BLACK;
                nd.color ← BLACK; rotateLeft(p);
                t ← T;
    else
        % Codice speculare al precedente;
if t ≠ nil then t.color = BLACK;
```

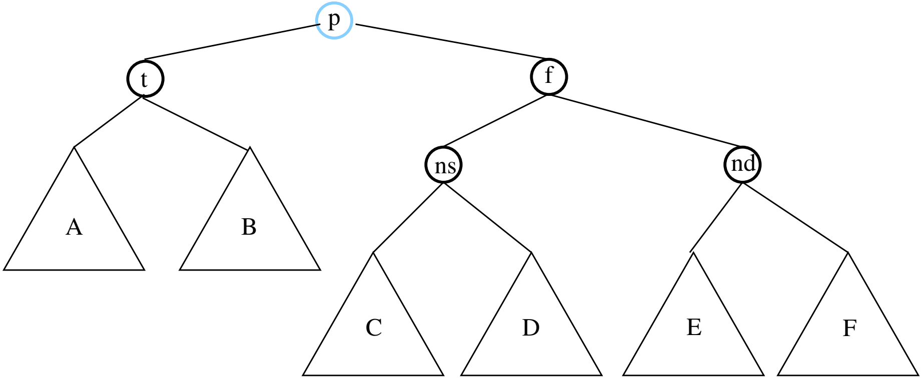
Alberi rosso-neri: ripristino su cancellazione caso 1



Si aggiusta trasformando in ...

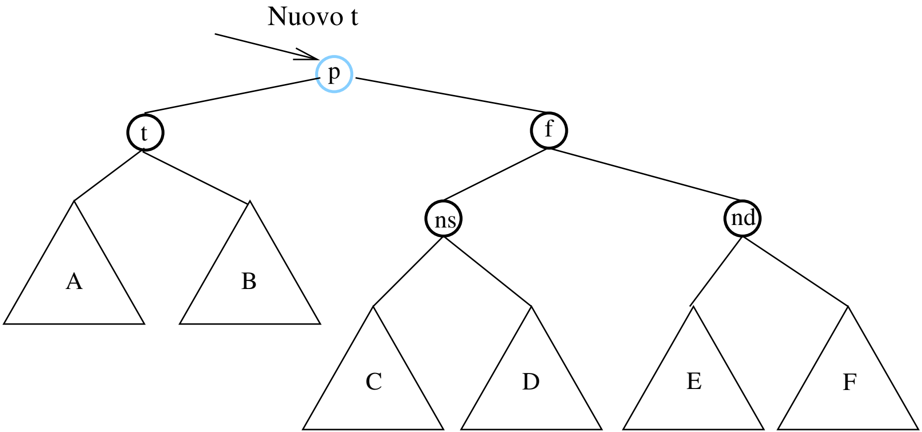
Alberi rosso-neri: ripristino su cancellazione caso 1

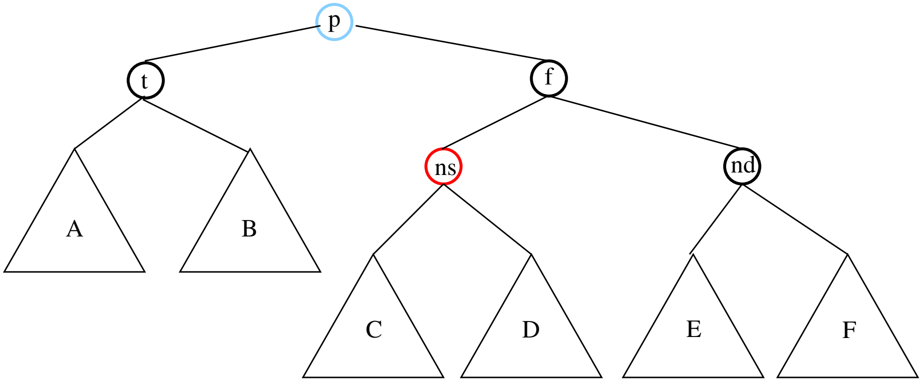




Si aggiusta trasformando in ...

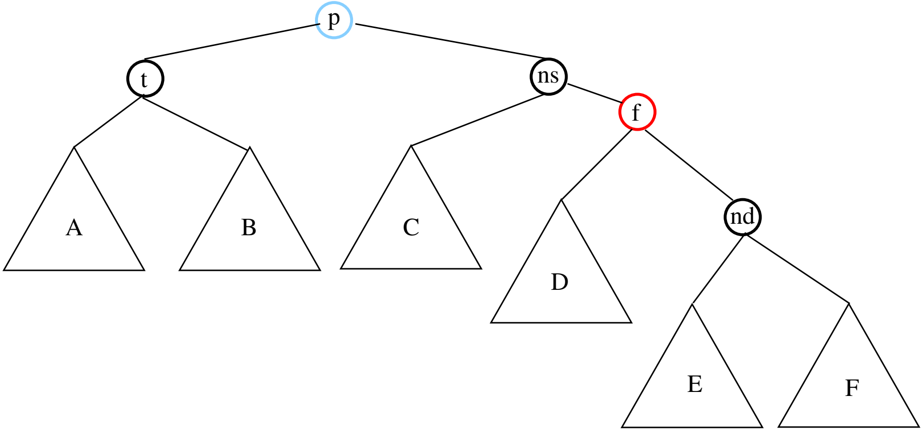
Alberi rosso-neri: ripristino su cancellazione caso 2



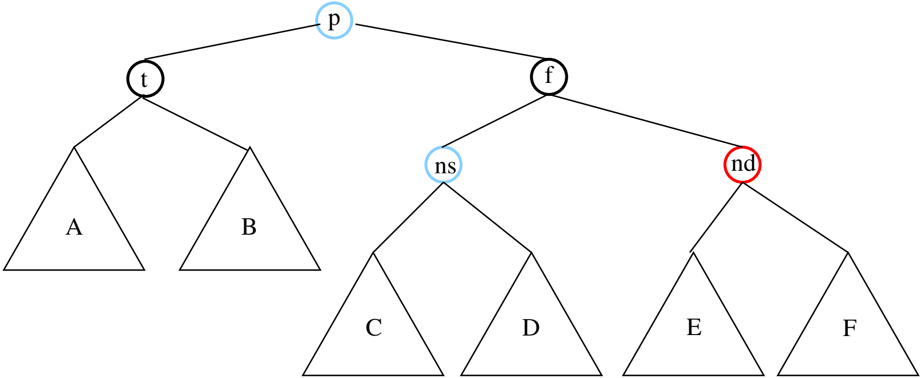


Si aggiusta trasformando in ...

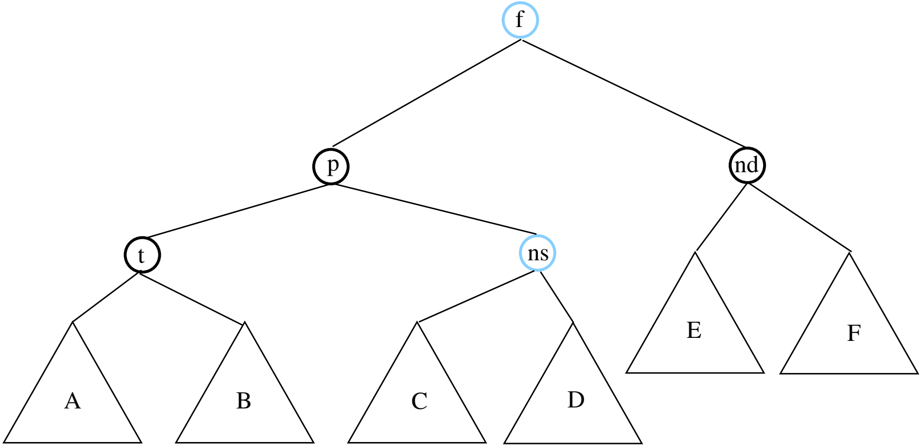
Alberi rosso-neri: ripristino su cancellazione caso 3



Alberi rosso-neri: ripristino su cancellazione caso 4



Si aggiusta trasformando in ...



Si pone $t = T$

Rotazione a sinistra