6 Alberi Binari di Ricerca

Facciamo un breve ripasso della struttura dati dizionario.
La struttura dati dizionario è un insieme dinamico che implementa le seguenti funzionalità:

- ITEM lookup(ITEM v) permette di cercare per una certa chiave
- insert(ITEM k, ITEM v) permette di associare una chiave ad un valore
- remove(ITEM k) permette di rimuovere una certa associazione chiave-valore

Tabella 1: Possibili implementazioni della struttura dati dizionario e delle relative complessità

Struttura dati	lookup	insert	remove
Vettore ordinato	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Vettore non ordinato	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$
Lista non ordinata	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$

* assumendo che l'elemento sia già stato trovato

Ora vedremo la struttura dati dizionario implementata come un albero binario di ricerca.

L'idea che ha portato allo sviluppo degli alberi binari di ricerca è quella di portare la ricera binaria (o dicotomica) negli alberi, avendo quindi un meccanismo dinamico per la memorizzazione delle informazioni ma sulla ricerca binaria per recuperarle.
Le associazioni chiave-valore vengono memorizzate in un albero binario. Ogni nodo contiene una coppia ($k.key$, $u.value$). Le chiavi devono appartene ad un insieme *totalmente ordinato*, ossia dev'essere possibile stabilire, date due chiavi, stabilire una relazione di precendenza fra di loro.

Proprietà. *Le seguenti proprietà permettono di realizzare un algoritmo di ricerca dicotomica:*

1. *Le chiavi contenute nei nodi del sottoalbero sinistro di u sono minori di $u.key$*
2. *Le chiavi contenute nei nodi del sottoalbero destro di u sono maggiori di $u.key$*

Nota. *Queste proprietà valgono per ogni nodo e riguardano l'intero sottoalbero.*

Vedremo un algoritmo per verificare se un albero binario è un albero binario di ricerca più avanti (verifyABR), il quale controllerà se queste proprietà sono soddisfatte.

Specifica ABR

```
// CONTENUTO DI UN NODO
TREE parent
TREE left
TREE right
TREE key
TREE value

// GETTERS
ITEM key
ITEM value
TREE left
TREE right
TREE parent

// ORDINAMENTO
TREE successorNode(TREE t)
TREE predecessorNode(TREE t)
TREE min
TREE max

// FUNZIONI DIZIONARIO
ITEM lookup(ITEM k)
insert(ITEM k, ITEM v)
remove(ITEM k)

// FUNZIONI INTERNE
ITEM lookupNode
insertNode(TREE T, ITEM k, ITEM v)
removeNode(TREE T, ITEM k)
```

Ricerca di un nodo

Implementazione DICTIONARY con ABR

```
int lookup(ITEM k)
    TREE t = lookupNode(tree, k)
    se t ≠ nil allora
        ritorna t.value
    allora
        ritorna nil

// RICERCA DI UN NODO, iterativa
TREE lookupNode(TREE T, ITEM k)
    TREE u = T  // parto dalla radice
    finché u ≠ nil and u.key ≠ k fai
        u ← iif(k < u.key, u.left, u.right)

// RICERCA DI UN NODO, ricorsiva
TREE lookupNode(TREE T, ITEM k)
    se T == nil or T.key == k allora
        ritorna T
    allora
        ritorna lookupNode(iif(k < u.key, u.left, u.right), k)
```

Ricerca del minimo e del massimo

Implementazione DICTIONARY con ABR

```
// RICERCA DEL MINIMO
TREE min(TREE T)
    TREE u = T  // parto dalla radice
    finché u.left ≠ nil fai
        u ← u.left
    ritorna u
```

```
// RICERCA DEL MASSIMO
TREE max(TREE T)
    TREE u = T  // parto dalla radice
    finché u.right ≠ nil fai
        u ← u.right
    ritorna u
```

Ricerca del predecessore, successore

Implementazione DICTIONARY con ABR

```
// RICERCA DEL PREDECESSORE
TREE predecessorNode(TREE t)
    se t == nil allora
        ritorna t
    se t.left ≠ nil allora
(1)     ritorna max(t.left)
    allora
(2)     TREE p ← t.parent
        finché p ≠ nil and t == p.left fai
            t ← p  // padre
            p ← p.parent  // nonno
    ritorna p
```

```
// RICERCA DEL SUCCESSORE
TREE successorNode(TREE t)
    se t == nil allora
        ritorna t
    se t.right ≠ nil allora
(3)     ritorna min(t.right)
    allora
(4)     TREE p ← t.parent
        finché p ≠ nil and t == p.right fai
            t ← p
            p ← p.parent
    ritorna p
```

(1) u ha figlio sinistro: il successore è il minimo del sottoalbero destro di u;

(2) u non ha figlio sinistro: risalendo attraverso i padri, il sucessore è il primo avo v tale per cui u sta nel sottoalbero sinistro di v;

(3) u ha figlio destro: il successore è il minimo del sottoalbero destro di u;

(4) u non ha figlio destro: risalendo attraverso i padri, il sucessore è il primo avo v tale per cui u sta nel sottoalbero sinistro di v.

Nota. *Posso trovare* **nil** *se passo alla funzione* successorNode *il nodo massimo o alla funzione* predecessorNode *il minimo, usciranno dal ciclo restituendo* p *che sarà pari a* **nil**.

Inserimento di un nodo

La funzione insertNode inserisce un'associazione chiave-valore (k, v) nell'albero T. Se la chiave è già presente, sostituisce il valore associato; altrimenti, viene inserita una nuova associazione. Se l'albero è vuoto ($T == \textbf{nil}$) restituisce il primo nodo dell'albero, altrimenti restituisce l'albero T inalterato. La funzione ausiliaria link si occupa di inserire il nodo collegandolo al corretto genitore.

Implementazione DICTIONARY con ABR

```
// IMPLEMENTAZIONE DIZIONARIO
insert(ITEM k, ITEM v)
    tree = insertNode(tree, k, v)

// INSERIMENTO DI UN NODO
TREE insertNode(TREE T, ITEM k, ITEM v)
    TREE p ← nil  // padre
    TREE u ← T  // parto dalla radice

    // cerco posizione inserimento
    finché u ≠ nil and u.key ≠ k fai
        p ← u
        u ← iif(k < u.key, u.left, u.right)

    se u ≠ nil and u.key == k allora
        // la chiave è già presente, aggiorno il valore
        u.value ← v
    altrimenti
        // la chiave non è presente
        // creo un nodo coppia chiave-valore
        TREE new ← Tree(k, v)

        // collego il nodo creato
        link(p, new, k)

        se p == nil allora
            T ← new  // primo nodo ad essere inserito

    // restituisco l'albero non modificato o il nuovo nodo
    ritorna T

// collega un nodo padre p ad un nodo figlio u
link(TREE p, TREE u, ITEM x)
    se u ≠ nil allora
        // il nodo è stato cancellato
        u.parent = p  // registro il padre

    se p ≠ nil allora
        // collego il nodo sul figlio corretto
        u ← iif(x < p.key, p.left, p.right)
```

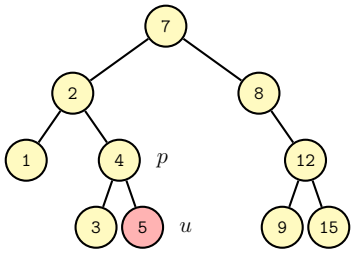
Rimozione di un nodo

Rimuove il nodo contenente la chiave k dall'albero T, restituisce la radice dell'albero (potenzialmente cambiata).

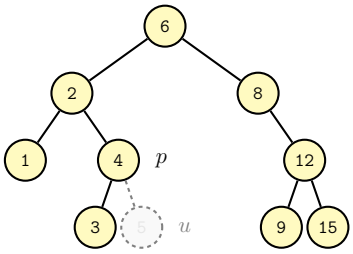
Implementazione della rimozione di un nodo in un DICTIONARY realizzato tramite ABR

```
// IMPLEMENTAZIONE DIZIONARIO
remove(ITEM k)
    TREE tree ← removeNode(tree, k)

// RIMOZIONE DI UN NODO
TREE removeNode(TREE T, ITEM k)
    // individuo il nodo da rimuovere
    TREE u ← lookupNode(T, k)

    // se il nodo da rimuovere è presente nell'albero...
    se u ≠ nil allora
(1)     // ...e non ha figli
        se u.left == nil and u.right == nil allora
            se u.parent ≠ nil allora // se esiste il padre
                link(u.parent, nil, k) // rimuovo il puntatore al figlio

            // rimuovo direttamente il nodo
            delete u
(3)     // ...ed ha due figli
        se u.left ≠ nil and u.right ≠ nil allora
            TREE s = successorNode // individuo il successore

            link(s.parent, s.right, s.key) // collego il sottoalbero destro

            // copio il successore
            // nella posizione del nodo rimosso
            u.key ← s.key
            u.value ← s.value

            // rimuovo il successore
            delete s
(2)     // ...ed ha un solo figlio (sinistro)
        se u.left ≠ nil and u.right == nil allora
            link(u.parent, u.left, k) // collega il figlio al padre

            se u.parent == nil allora // se il padre non esiste
                T == u.right // il figlio diventa la radice
        // ...ed ha un solo figlio (sinistro)
        altrimenti
            link(u.parent, u.right, k) // collega il figlio al padre

            se u.parent == nil allora // se il padre non esiste
                T == u.right // il figlio diventa la radice

    // restituisco la radice
    ritorna T
```

(1) se il nodo da eliminare u non ha figli: lo si elimina semplicemente, in quanto togliere una foglia non altera le proprietà di ordinamento dell'albero;

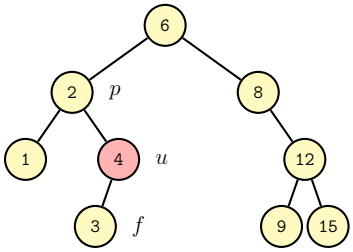


(a) Individuazione nodo foglia

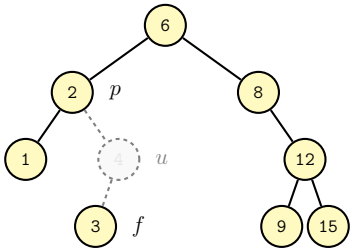


(b) Rimozione del nodo foglia

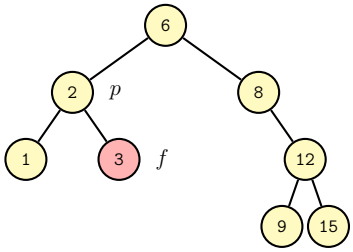
(2) se il nodo da eliminare ha un solo figlio f (destro o sinistro): si elimina u e si collega f all'ex-padre p di u in sostituzione di u (tramite la funzione link); le proprietà di ordinamento non vengono alterate in quanto tutti i nodi del sottoalbero destro di p sono maggiori di p stesso;



(a) Individuazione nodo u da eliminare

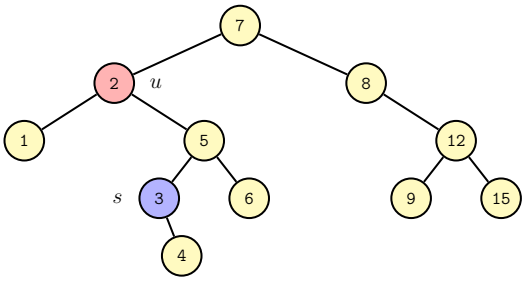


(b) Rimozione nodo u

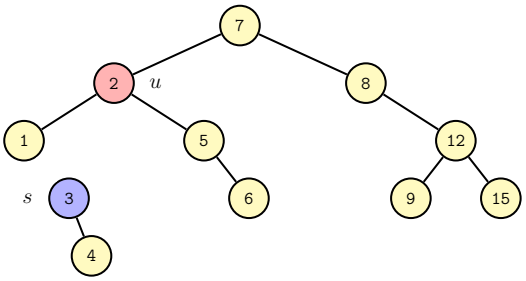


(c) Collegamento del sottoalbero f di u al padre p di u

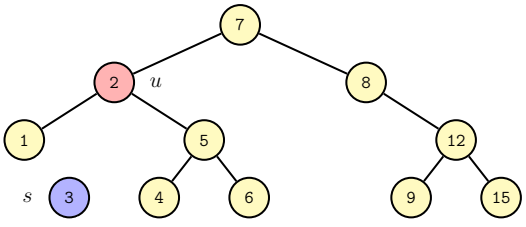
(3) se il nodo da eliminare u ha due figli: cerchiamo di ricadere nel caso (2); (a) individuiamo il successore (predecessore) s di u, il quale è il più piccolo valore maggiore di u (il più grande valore minore di u) e di conseguenza non ha figli sinistri (non ha figli destri); (b) si "stacca" il successore s; (c) si collega l'eventuale figlio destro di s al padre (tramite la funzione link) in quanto trovandosi nel sottoalbero sinistro del nonno vuol dire che sicuramente il suo valore non è maggiore del padre di s; (d) si copia s su u, si rimuove il nodo s, così facendo rispetto comunque l'ordine parziale.



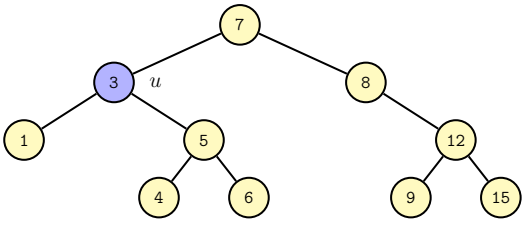
(a) Individuiamo il successore s di u



(b) Si "stacca" il successore s



(c) Si collega l'eventuale figlio destro di s al padre



(d) Si copia s su u, si rimuove il nodo s

Costo computazionale delle operazioni

Tutte le operazioni sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia. Quindi se l'altezza dell'albero è definita come h, il tempo di ricerca ha complessità $\mathcal{O}(h)$.

Il caso pessimo è rappresentanto da un albero sbilanciato completamente a destra o completamente a sinistra. Questo caso può accadere quando si inseriscono ordinatamente i dati nell'albero. Questo caso, dove l'altezza $h = n$ porta ad una complessità $\mathcal{O}(n)$.

Mentre il caso ottimo è rappresentanto da un albero bilanciato. Nell'esempio è mostrato un albero perfetto con $2^h - 1$ nodi, dove h è l'altezza. In questo caso la complessità è pari a $\mathcal{O}(\log n)$, ad esempio con $h = 2^3 - 1$ la complessità è $\mathcal{O}(\log h) = \mathcal{O}(\log 7) < 3$.

Ci domandiamo quindi quale sia l'altezza media di un albero binario di ricerca. Il caso "semplice" è quello di considerare che gli inserimenti avvengano in maniera statisticamente uniforme, è possibile dimostrare che l'altezza media è $\mathcal{O}(\log n)$, mentre il caso generale, ossia quello in cui avvengono sia inserimenti che cancellazioni è di difficile trattazione. Per evitare questa casisistica si utilizzano varie tecniche per mantenere l'albero bilanciato. Per capire queste tecniche abbiamo prima bisogno di fissare un concetto.

Definizione 6.1 (Fattore di bilanciamento)**.** *Il fattore di bilanciamento $\beta(v)$ di un nodo v è la massima differenza di altezza fra i sottoalberi di v.*

Negli anni sono state usate diverse tecnische, ora in disuso:

- Alberi AVL (1962): $\beta(v) \leqslant 1$ per ogni nodo v, il bilanciamento dell'albero avveniva tramite rotazioni;
- B-Alberi (1972): $\beta(v) = 0$ per ogni nodo v, sono specializzati per strutture in memoria secondaria;
- Alberi 2-3 (1983): $\beta(v) = 0$ per ogni nodo v, in cui ogni nodo può avere o 2 o 3 figli, se ad un nodo viene aggiunto un ulteriore figlio, il ramo viene spezzato in due rami con 2 figli ciascuno, mentre se ad un ramo con 2 figli ne viene tolto uno allora l'unico figlio rimanente viene collegato al padre, questo potrebbe riportare il problema al primo caso; il bilanciamento viene ottenuto quindi tramite merge/split, il grado è variabile.

Meccanismo di rotazione

Il meccanismo di rotazione ci permette di abbassare il fattore di sbilanciamento rispettando le proprietà di ordinamento parziale.

6.1 Alberi Binari di Ricerca bilanciati

Definizione 6.2 (Albero Red-Black)**.** *Un albero red-black è un albero binario di ricerca in cui:*

- *ogni nodo è colorato di rosso o di nero;*
- *le chiavi vengono mantenute solo nei nodi interni dell'albero;*
- *le foglie sono costituite solo da nodi speciali* ***Nil****.*

I nodi speciali **Nil** sono dei nodi sentinella il cui unico scopo è quello di evitare di trattare diversamente i puntatori ai nodi, dai puntatori **nil**; infatti al posto di un puntatore **nil** si usa un puntatore ad un nodoo **Nil**; in memoria ne esiste solo uno per motivi di economia. I nodi con figli **Nil** sono le foglie nell'albero binario di ricerca corrispondente.

Un albero red-black deve rispettare i seguenti vincoli:

1. La radice è nera;
2. Tutte le foglie sono nere;
3. Entrambi i figli di un nodo rossso sono neri;
4. Ogni cammino semplice da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numeo di nodi neri.

Specifica Red-Black Tree

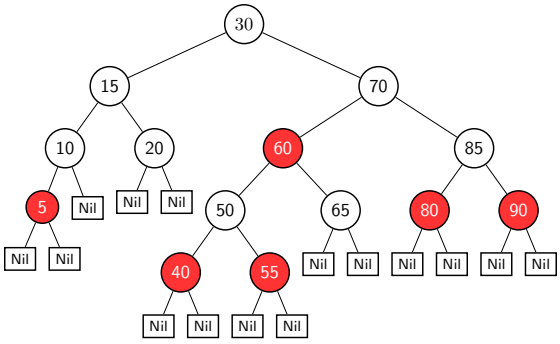
```
// CONTENUTO DI UN NODO
Tree parent
Tree left
Tree right
int color  // red o black
Tree key
Tree value
```

Proprietà (Altezza nera di un nodo v)**.** *L'altezza nera $b(v)$ di un nodo v è il numero di nodi neri lungo ogni percorso da v (escluso) ad ogni foglia (inclusa) del suo sottoalbero.*

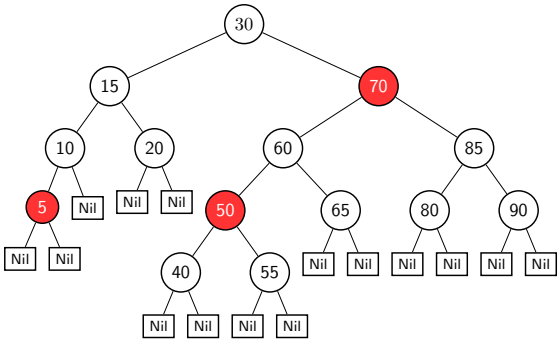
Proprietà (Altezza nera di un albero Red-Black)**.** *L'altezza nera di un albero Red-Black è pari all'altezza nera della sua radice.*

Entrambe le proprietà sono ben definite perché tutti i percorsi hanno lo stesso numero di nodi neri (per via della regola no. 4).

Esempi

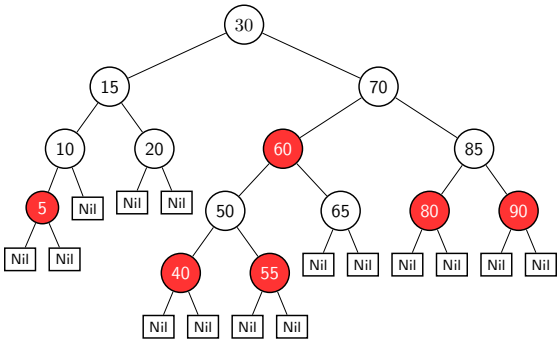


Es. 1 Entrambi i figli di un nodo rosso sono neri (3), ma un nodo nero può avere figli neri.

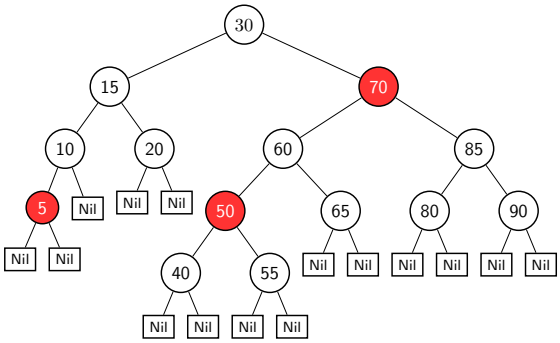


Es. 2 Ogni percorso da un nodo interno ad un nodo **Nil** ha lo stesso numero di nodi neri (4). Altezza nera di quest'albero: 3

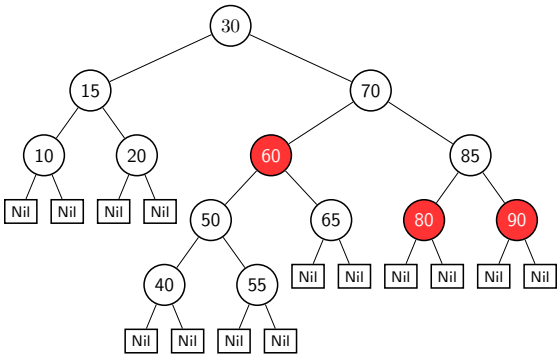
Lemma 1. *L'altezza totale di un albero è al più il doppio della sua altezza nera.*



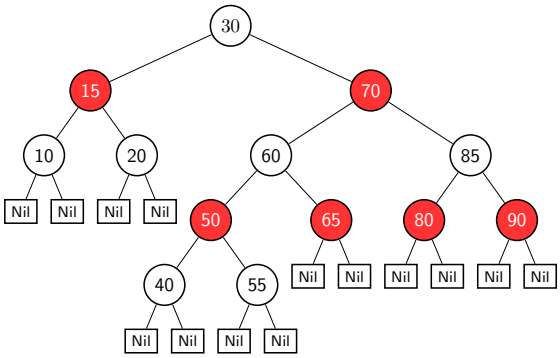
Es. 3 Più colorazioni sono possibili (versione 1). Altezza di questo albero: 3



Es. 4 Più colorazioni sono possibili (versione 2). Altezza di questo albero: 3



Es. 5 Cambiare colorazione può cambiare l'altezza nera. Altezza di questo albero: 3



Es. 6 Cambiare colorazione può cambiare l'altezza nera. Stesso albero, altezza nera di questo albero: 2

6.1.1 Inserimento di un nodo

Durante la modifica di un albero Red-Black è possibile che le condizioni di bilanciamento risultino violate. Quando i vincoli Red-Black vengono violate si può agire in due modi:

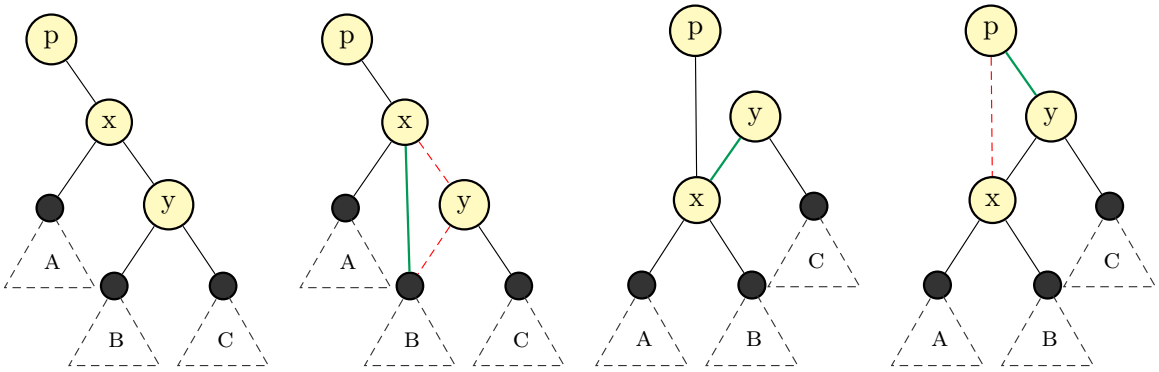
- modificando i colori nella zona della violazione;
- operando dei bilanciamenti dell'albero tramite rotazioni (a destra o a sinistra)

6.1.2 Bilanciamento dell'albero

Rotazione a sinistra

```
    // effettua una rotazione verso sinistra
    rotateLeft(TREE p, TREE u, ITEM x)
(1) |   TREE y ← x.right
    |   TREE p ← x.parent
(2) |   x.right ← y.left  // il sottoalbero B diventa figlio destro di x
    |   se y.left ≠ nil allora
    |   |   y.left.parent ← x
(3) |   y.left ← x  // x diventa figlio sinistro di y
    |   x.parent ← y
(4) |   y.parent ← p  // y diventa figlio di p
    |   se p ≠ nil allora
    |   |   se p.left == x allora
    |   |   |   p.left ← y
    |   |   p.right ← y
    |   ritorna y
    |
```

Nota. *I disegno differisce un minimo da quello che si trova sulle slide per motivi di comodità nel disegnarli.*



(1) Stato attuale

(2) Il sottoalbero B diventa figlio dx di x

(3) x diventa figlio sx di y

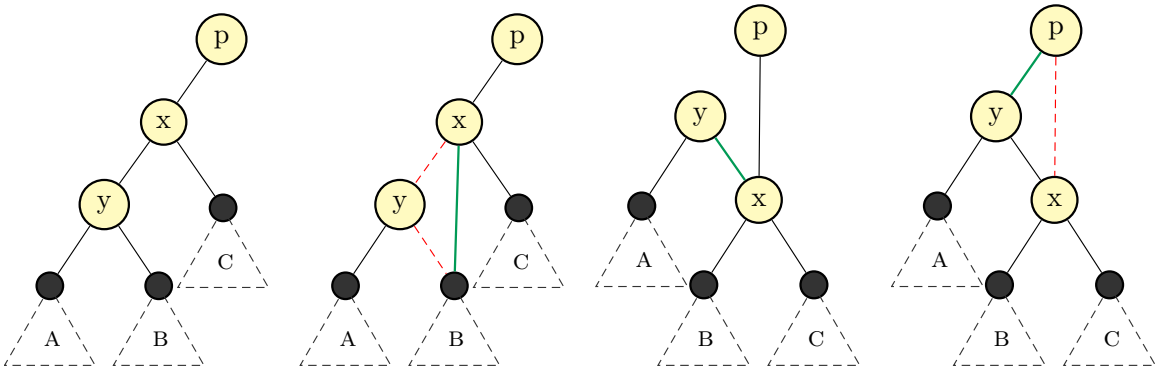
(4) y diventa figlio di p, il vecchio padre di x

Figura 5: Esempio di rotazione a sinistra

Rotazione a destra

La rotazione a destra è simmetrica e viene spiegata ed illustrata per completezza.

```
   // effettua una rotazione verso destra
   rotateRight(TREE p, TREE u, ITEM x)
       // entrambi potrebbero essere nil
(1)    TREE y ← x.left
       TREE p ← x.parent
(2)    x.left ← y.right  // il sottoalbero B diventa figlio sinistro di x
       se y.right ≠ nil allora
           y.right.parent ← x
(3)    y.right ← x  // x diventa figlio destro di y
       x.parent ← y
(4)    y.parent ← p  // y diventa figlio di p
       se p ≠ nil allora
           se p.right == x allora
               p.right ← y
           p.left ← y
       ritorna y
```



(1) Stato attuale

(2) Il sottoalbero B diventa figlio sx di x

(3) x diventa figlio dx di y

(4) y diventa figlio di p, il vecchio padre di x

Figura 6: Esempio di rotazione a destra

Inserimento di un nodo in un albero Red-Black

Per inserire un nodo in un albero Red-Black si usa la stessa procedura usata per gli alberi binari di ricerca e si colora il nuovo nodo di RED. Il vincolo che violeremo è il terzo, quello che prevede che entrambi i figli di un nodo rosso siano neri.

Inserimento di un nodo in un RED-BLACK TREE

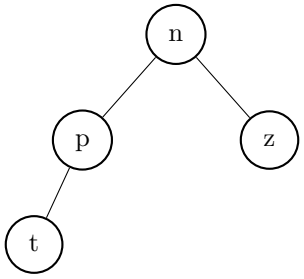
```
// Inserimento di un nodo in un albero Red-Black
insertNode(TREE T, TREE k, ITEM x)
    TREE p ← nil  // riferimento al padre
    TREE u ← T  // riferimento alla radice

    // cerco posizione inserimento
    finché u ≠ nil and u.key ≠ k fai
        p ← u
        u ← iif(k < u.key, u.left, u.right)

    se u ≠ nil and u.key == k allora
        // la chiave è già presente, aggiorno il valore
        u.value ← v
    altrimenti
        // la chiave non è presente
        // creo un nodo coppia chiave-valore
        TREE new ← Tree(k, v)

        // collego il nodo creato
        link(p, new, k)
        balanceInsert(new)

        se p == nil allora
            T ← new  // primo nodo ad essere inserito

    // restituisco l'albero non modificato o il nuovo nodo
    ritorna T
```

Ci spostiamo verso l'alto lungo il percorso di inserimento; cercheremo di ripristinare il terzo vincolo; sposteremo le violazioni verso l'alto rispettando il quarto vincolo (mantenendo l'altezza nera dell'albero); al termine, coloreremo la radice di nero (onorando il primo vincolo).

Nota. *Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi, altrimenti non sono necessarie.*

```
// bilanciamento di un RED-BLACK TREE in seguito all'inserimento di un nodo RED
balanceInsert(TREE t)
    t.color ← RED  // coloro il nodo da inserire di rosso

    // t == nil è la condizione di fino ciclo
    finché t ≠ nil fai
(0)     TREE p ← t.parent                                          // riferimento al padre
        TREE n ← iif(p ≠ nil, p.parent, nil)                       // riferimento al nonno
        TREE z ← iif(n == nil, nil, iif(n.left, n.right, n.left))  // riferimento allo zio
(1)     se p == nil allora
            t.color ← BLACK
            t ← nil  // fine
(2)     altrimenti se p.color == BLACK allora
            t ← nil  // fine
(3)     altrimenti se z.color == RED allora
            p.color ← z.color ← BLACK
            n.color ← RED
            t ← n  // passo il problema al nonno
        altrimenti
(4a)        se (t == p.right) and (p == n.left) allora
                rotateLeft(p)
                t ← p  // passo il problema al padre
(4b)        se (t == p.left) and (p == n.right) allora
                rotateRight(p)
                t ← p  // passo il problema al padre
            altrimenti
(5a)            se (t == p.left) and (p == n.left) allora
                    rotateRight(n)
(5b)            altrimenti se (t == p.right) and (p == n.right) allora
                    rotateLeft(n)

                p.color ← BLACK
                n.color ← RED
                t ← nil  // fine
```

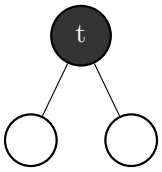
(0) dichiaro dei riferimento al padre, al nonno e allo zio



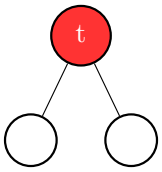
(1) il nuovo nodo t non ha padre; Questo può accadere in due casi:

- è il primo nodo ad essere inserito, oppure
- quando abbiamo spostato la violazione verso l'alto fino a raggiungere la radice

Ricoloriamo t di BLACK in quanto trovandosi sulla radice dell'albero non viola nessun vincolo.

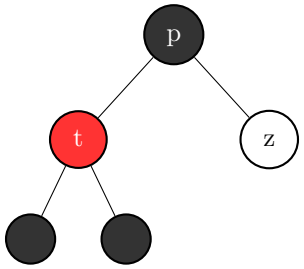


(a) Possibile violazione del primo vincolo

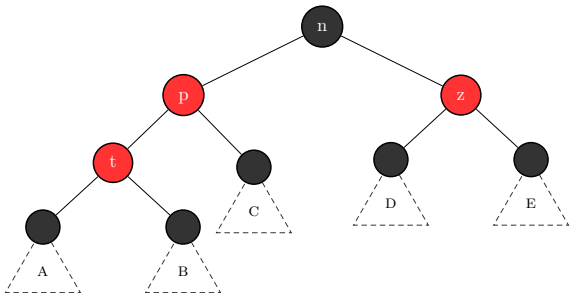


(b) Ricolorazione del nodo t

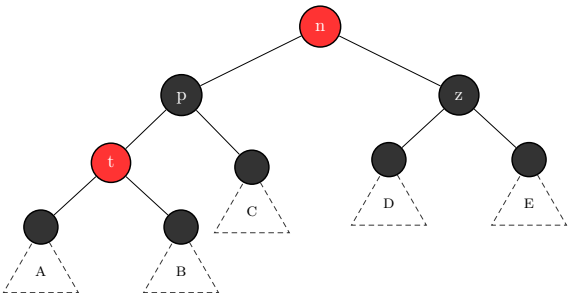
(2) il padre p di t è nero; anche in questo caso non abbiamo violato nessun vincolo perché avendo inserito un nodo rosso la lunghezza dei cammini neri non cambia e avendo inserito un nodo rosso figlio di un nodo nero non violo il terzo vincolo;



(3) il padre p e lo zio z sono rossi; Se z è rosso è possibile ricolorare di nero p e z, e di rosso n; poiché tutti i cammini che passano per z e p passano anche per n, la lunghezza che cammini non è cambiata (non abbiamo violato il quarto vincolo); Abbiamo spostato così il problema verso l'alto, più precisamente sul nonno che potrebbe aver violato il primo o il terzo vincolo, ovvero che n può essere una radice rossa o che abbia un padre rosso. Per risolvere il problema poniamo $t \leftarrow n$ e continuiamo il ciclo.



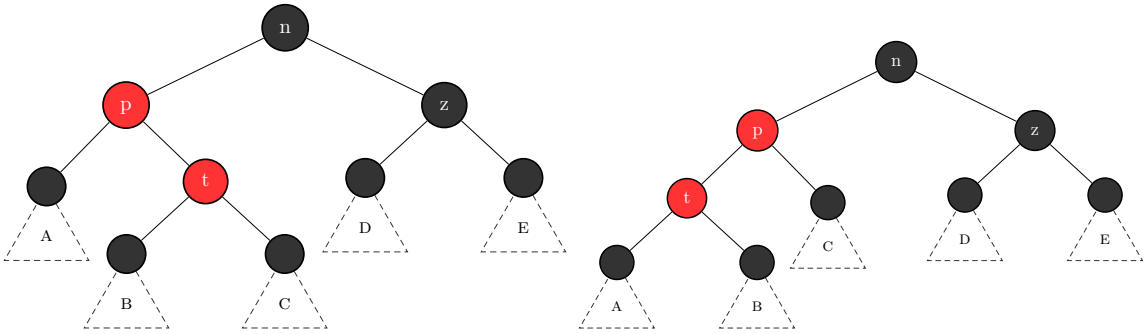
(a) Possibile violazione del primo e del terzo vincolo



(b) Ricolorazione del padre e dello zio di nero

`(4a)` il padre p è rosso e lo zio z è nero; si assuma che t sia figlio *destro* di p e che p sia figlio *sinistro* di n; effettuando una rotazione a sinistra a partire dal nodo p scambiamo i ruoli di t e d p ottenendo il caso `(5a)`, dove i nodi in conflitto sul terzo vincolo sono entrambi figli *sinistri* dei loro padri; i nodi coinvolti nel cambiamento sono p e t, entrambi rossi, quindi la lunghezza dei cammini neri non cambia; abbiamo spostato il problema al padre, quindi poniamo $t \leftarrow p$ e continuiamo il ciclo.

`(4b)` speculare al caso `(4a)` (ossia che che t sia figlio *sinistro* di p e che p sia figlio *destro* di n)

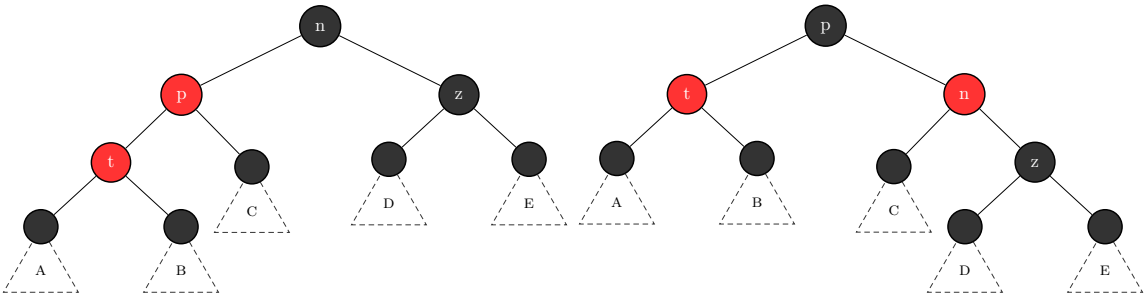


(a) Possibile violazione del primo e del terzo vincolo **(b)** Ricolorazione del padre e dello zio di nero

Figura 9: Rappresentazione grafica del caso `(4a)`

`(5a)` anche in questo caso il padre p è rosso e lo zio z è nero; ma si assuma che t sia figlio *sinistro* di p e che p sia figlio *sinistro* di n; effettuando una rotazione a destra a partire dal nodo n ci porta ad una situazione in cui t e n sono figli di p; colorando p di nero ed n di rosso ci troviamo in una situazione in cui tutti i vincoli vengono rispettati (in particolare, la lunghezza dei cammini neri che passano per la radice è uguale a quella iniziale).

`(5b)` speculare al caso `(5a)` (ossia che che che t sia figlio *destro* di p e che p sia figlio *destro* di n)



(a) Possibile violazione del primo e del terzo vincolo **(b)** Ricolorazione del padre e dello zio di nero

Figura 10: Rappresentazione grafica del caso `(5a)`

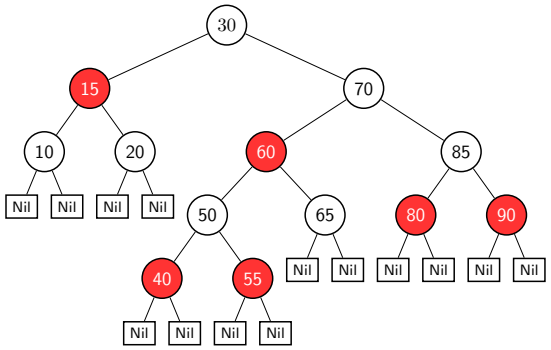
Complessità Ognuna di queste operazione avviene in tempo costante. Ogni volta il problema può salire di uno o due livelli, in quanto l'altezza dell'albero è limitata da $\log n$, il nostro algoritmo è limitato superiormente da $\log n$, ossia $\mathcal{O}(\log n)$.

Più precisamente:

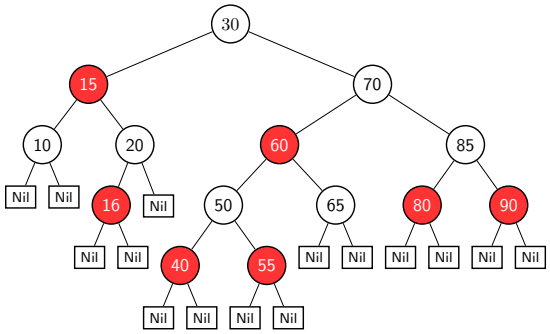
- $\mathcal{O}(\log n)$ per scendere fino al punto di inserimento;
- $\mathcal{O}(1)$ per effettuare l'inserimento;
- $\mathcal{O}(\log n)$ per risalire ed "aggiustare" (caso `3`)

Esempi di inserimento

Proviamo ad inserire il nodo 16 nell'albero red-black sottostante.



(a) Stato attuale

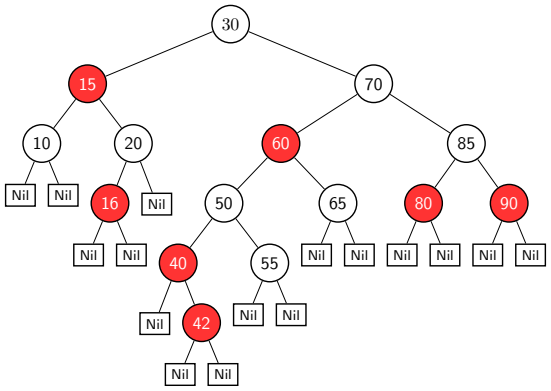


(b) Inserimento del nodo 16 andato a buon fine

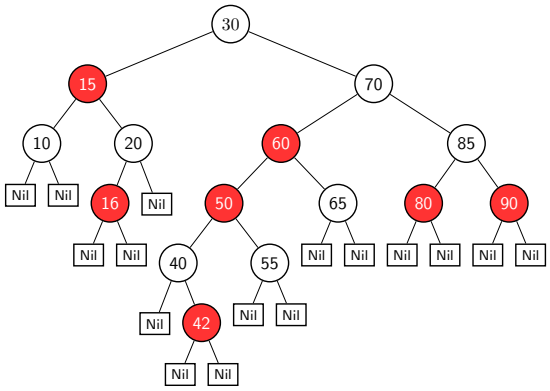
Non violiamo alcun vincolo:

- il padre di 16 è nero;
- non abbiamo modificato l'altezza nera.

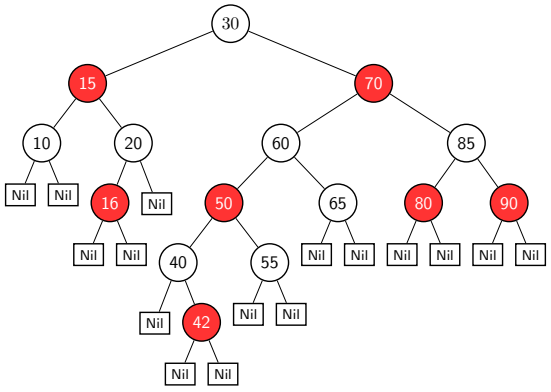
Questo caso rappresenta il caso 2, l'inserimento del nodo 16 è quindi andato a buon fine. Alternativamente proviamo ad inserire il nodo 42 sempre nello stesso albero.



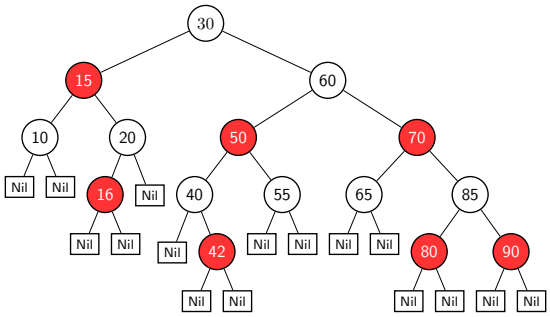
(a) Inserimento del nodo 42, violiamo il secondo vincolo, ci troviamo nel caso 3 (z rosso), quindi coloriamo di nero p e z e di rosso n, il problema si sposta ad n



(b) Violiamo il terzo vincolo, entrambi i nodi rossi sono figli *sinistri* quindi ci troviamo nel caso 5a, quindi coloriamo di nero p, di rosso n ...



(c) ... ed effetuiamo una rotazione a destra con perno n



(d) Abbiamo ripristinato il terzo vincolo, gli altri vincoli non sono mai stati violati, quindi abbiamo finito

Questo esempio ci mostra chiaramente come gli alberi Red-Black attraverso il rispetto dei vincoli tendano a mantenere bilanciato l'albero. Esiste una versione di inserimento *top-down* che scende fino al punto di inserimento "aggiustando" l'albero man-mano e si effettua l'inserimento in una foglia.

6.1.3 Rimozione di un nodo

Se il nodo rimosso è rosso l'altezza nera rimane invariata, non sono stati creati nodi rossi consecutivi e la radice resta nera. Il problema sorge quando si rimuovono nodi neri in quanto possiamo violare primo ed il terzo vincolo e sicuramente abbiamo violato il quarto vincolo in quanto cambiamo l'altezza nera. L'algoritmo balanceDelete(T, t) ripristina la proprietà Red-Black con rotazioni e cambiamenti di colore. Ci sono quattro casi possibili (e 4 simmetrici)
A lezione non è stato spiegato l'algoritmo, viene riportato solo per completezza.

Rimozione di un nodo in un RED-BLACK TREE

```
// bilanciamento di un RED-BLACK TREE in seguito alla rimozione di un nodo RED
balanceDelete(TREE t)
    t.color ← RED  // coloro il nodo da inserire di rosso
    finché (t ≠ T) and (t.color == BLACK) fai
        TREE p ← t.parent                                   // riferimento al padre
        se t == p.left allora
            TREE f ← p.right                                // riferimento al fratello
            TREE ns ← f.left                                // riferimento al nipote sinistro
            TREE nd ← f.right                               // riferimento al nipote destro
(1)         se f.color == RED allora
                p.color ← RED
                f.color ← BLACK
                rotateLeft(p)
                // t viene lasciato inalterato, quindi si ricade nei casi 2, 3, 4
            altrimenti
(2)             se ns.color == nd.color == BLACK allora
                    f.color == RED
                    t ← p  // passo il problema al padre
(3)             altrimenti se (ns.color == RED) and (nd.colorBLACK) allora
                    ns.color ← BLACK
                    f.color ← RED
                    rightRotationf
                    // t viene lasciato inalterato, quindi si ricade nel caso 4
(4)             altrimenti se nd.color == RED allora
                    f.color == p.color
                    p.color ← BLACK
                    nd.color ← BLACK
                    leftRotationp
                    t ← T
        altrimenti
            // casi speculari
```

La cancellazione è concettualmente complicata, ma è efficiente.

(1) si passa ad uno dei casi 2, 3, 4;

(2) si torna ad uno degli altri casi, ma risalendo di un livello l'albero;

(3) si passa al caso 4;

(4) si termina.

Complessità É possibile visitare al massimo un numero $\mathcal{O}(\log n)$ di casi, ognuno dei quali è gestito in $\mathcal{O}(1)$.