Interval Trees

Andrea Cascino
Matricola 1000002539

Indice

Introduzione ai Red-Black Trees

Un Red-Black Tree non è altro che un albero binario di ricerca dove però, ad ogni nodo, viene assegnato un colore tra Rosso e nero

Le complessità della maggioranza delle operazioni sugli Alberi binari di ricerca dipendono dall'altezza della struttura, nel caso in cui avessimo un albero completamente sbilanciato le operazioni ne risentono molto a livello computazionale

I Red-Black Tree cercano di mantenere l'albero approssimativamente bilanciato in maniera tale da poter eseguire le operazioni su di esso in maniera più rapida possibile
L'albero viene tenuto bilanciato tramite regole sulla colorazione dei nodi lungo un qualsiasi percorso dalla radice alla foglia

Cosa si intende per albero bilanciato?

Prima di dare la definizione di albero bilanciato bisogna definire il concetto di

Fattore di bilanciamento

Sia T un albero e v un nodo appartenente a T

Il fattore di bilanciamento B(v) è uguale alla differenza tra l'altezza del suo sottoalbero sinistro e del suo sottoalbero destro

B(v) = altezza(left(v)) – altezza(right(v))

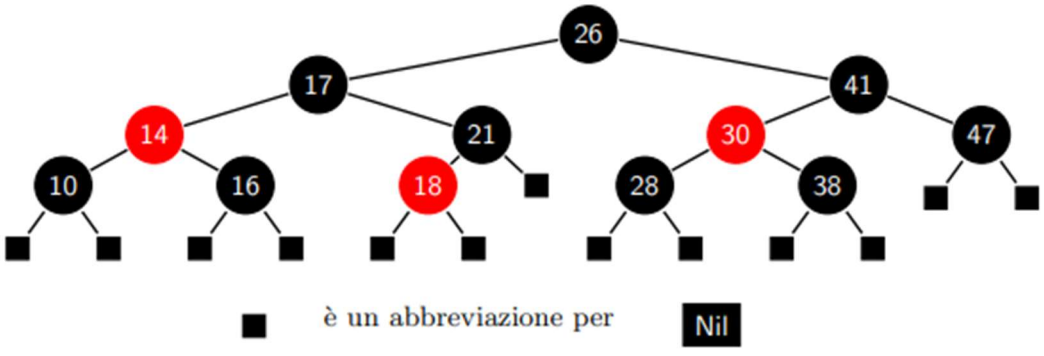
Un albero viene detto bilanciato in altezza se per ogni suo nodo **|B(v)| <= 1**

Proprietà dei Red-Black Trees

Un Red-Black Tree è un albero binario di ricerca che soddisfa le seguenti priorità

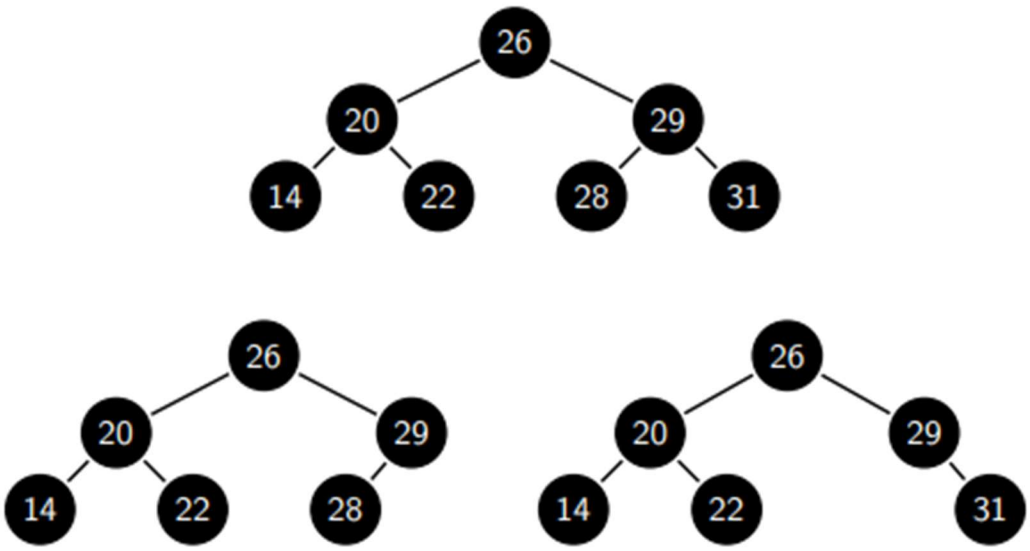
- Ogni nodo è rosso o nero
- La radice è nera
- Ogni foglia è nera
- Se un nodo è rosso allora i suoi figli devono essere neri
- Tutti i percorsi da qualsiasi nodo n ad una qualsiasi delle sue foglie discendenti contengono lo stesso numero di nodi neri

Nota: Per rilassare la proprietà 3 vengono aggiunti dei nodi NIL fittizi come foglie



Fonte: Slides RBTrees Prof. Simone Faro

Nota: Un albero senza nodi rossi, per la proprietà 5, sarà un albero perfettamente bilanciato a meno dell'ultimo livello. I nodi rossi servono a rilassare questa proprietà



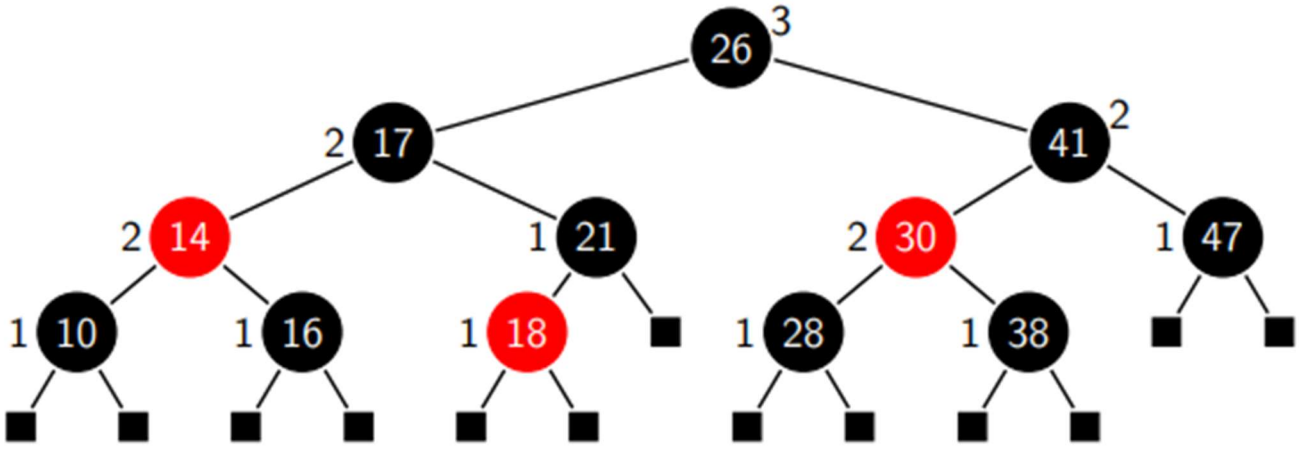
Fonte: Slides RBTrees Prof. Simone Faro

Altezza Nera

Sia x un nodo di un RB Tree T. L'altezza nera di x è pari al numero di nodi neri (x escluso) lungo un cammino da x a una delle sue foglie discendenti

L'altezza nera di un nodo x viene indicata con il simbolo bh(x)

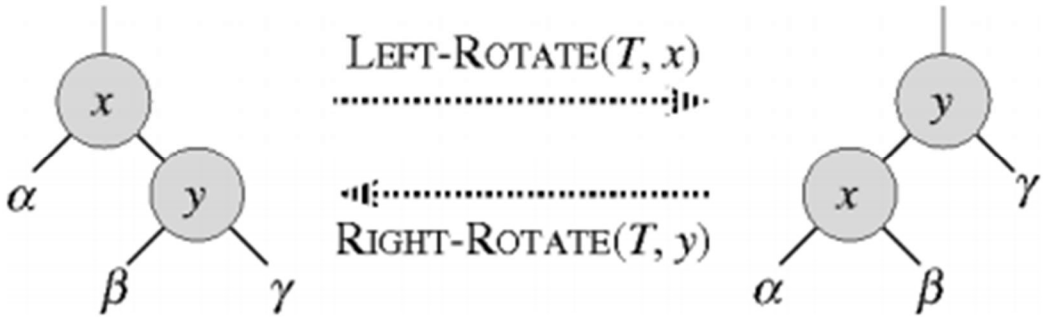
Per la proprietà 5 il concetto di altezza nera è ben definito, dato che il percorso da un nodo x a una sua qualsiasi foglia ha lo stesso numero di nodi neri



Fonte: Slides RBTrees Prof. Simone Faro

Rotazioni

Le rotazioni sono delle operazioni locali che vanno a cambiare la struttura dell'albero. Nonostante le rotazioni modifichino la struttura dell'albero, preservano tutte le proprietà degli alberi binari di ricerca. Possiamo ruotare l'albero sia a destra che a sinistra



Rappresentazione grafica operazioni di Rotazione

Inserimento di un nodo

Come negli alberi binari di ricerca, l'inserimento di un nodo z all'interno dell'albero, cerca un cammino dalla radice dell'albero fino a un nodo p che diventerà il padre di z
I nuovi nodi verranno sempre inseriti come foglie dell'albero (Non considerando i nodi NIL fittizi)

I nuovi nodi vengono sempre colorati di rosso, per evitare inconsistenze nella black height.
Aggiungere un nuovo nodo di colore rosso però potrebbe portare altri 2 tipi di inconsistenze

- Se l'albero precedentemente era vuoto il nuovo nodo sarà la radice dell'albero che deve essere necessariamente di colore nero violando così la proprietà 2
- Se il padre è di colore rosso andremo a violare la proprietà 4, dato che il padre avrà ora un figlio rosso

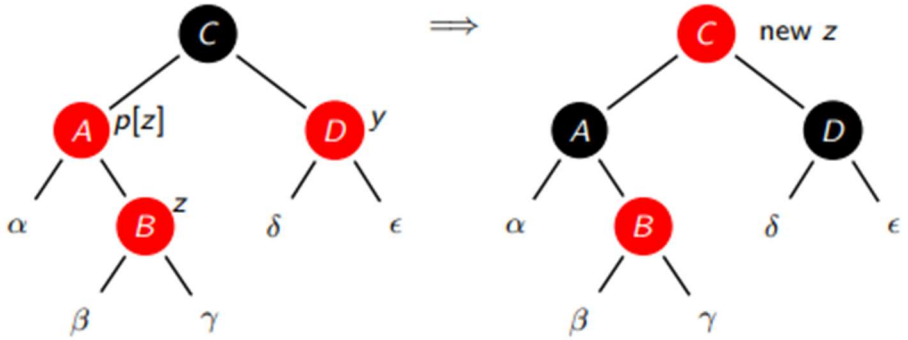
Le inconsistenze mostrate vengono però risolte tramite la procedura RB-Insert-Fixup
La procedura RB-Insert-Fixup ripristina la proprietà 2 andando a colorare il nuovo nodo di nero e la proprietà 4 tramite una serie di rotazioni e di ricolorazioni

Decidiamo cosa fare in base al colore dello zio y di z
Possiamo distinguere 3 casi

1. y è rosso
2. y è nero e z è un figlio sinistro
3. y è nero e z è un figlio destro

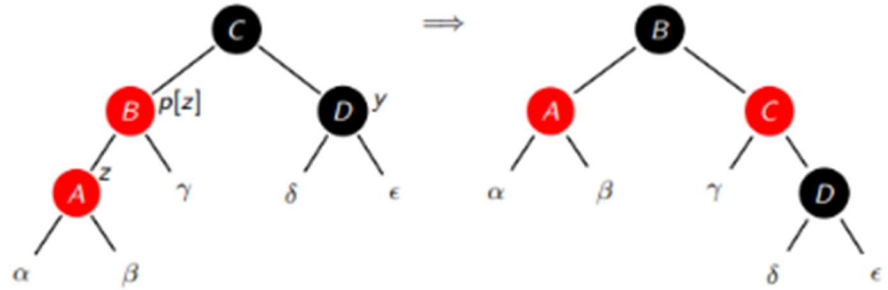
Caso 1.

In questo caso andiamo a modificare il colore di p[z] e y settandoli su nero. A questo punto andiamo a colorare p[p[z]] di rosso per ripristinare il valore della black height.
p[p[z]] potrebbe aver provocato una nuova violazione visto che abbiamo portato il suo colore a rosso, andiamo quindi a richiamare RB-Insert-Fixup su p[p[z]]



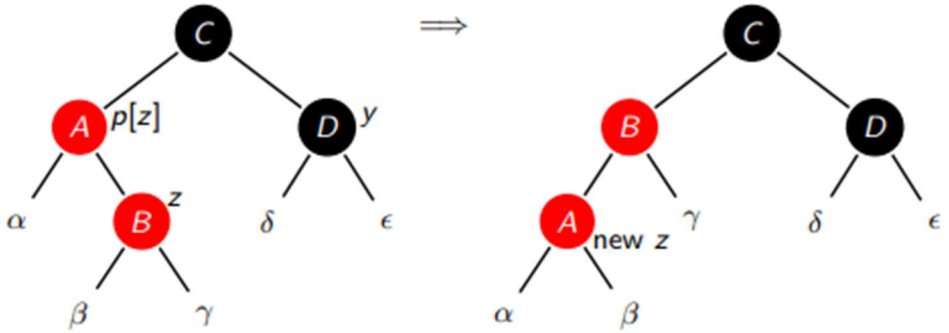
Caso 2.

Ruotiamo il nodo C a destra e coloriamolo di Rosso. Coloriamo il padre di z di nero



Caso 3.

Il caso 3 può essere banalmente ricondotto al caso 2 andando ad effettuare una rotazione a sinistra del nodo p[z] e richiamando RB-Insert-Fixup su p[z]



Nota: Bisogna ovviamente porre attenzione sui casi simmetrici, tutti i casi descritti danno per scontato che z si trovi sul sottoalbero sinistro del nodo c ma potrebbe trovarsi sul sottoalbero destro

A causa del Caso 1. la complessità di RB-Insert-Fixup è pari a $O(\log_2(n))$ nel caso peggiore, visto che la procedura potrebbe essere richiamata ricorsivamente fino alla radice. Di conseguenza pure la complessità di RB-Insert è pari a $O(\log_2(n))$ nel caso peggiore

Eliminazione di un nodo

Possiamo limitarci a studiare il caso in cui il nodo z da eliminare abbia al più un figlio, questo perché, nel caso in cui il nodo z abbia 2 figli, possiamo limitarci a richiamare la procedura **y = TREE_SUCCESSOR**, scambiare le posizioni tra il nodo z e y per poi limitarci a cancellare il nodo z nella sua nuova posizione.

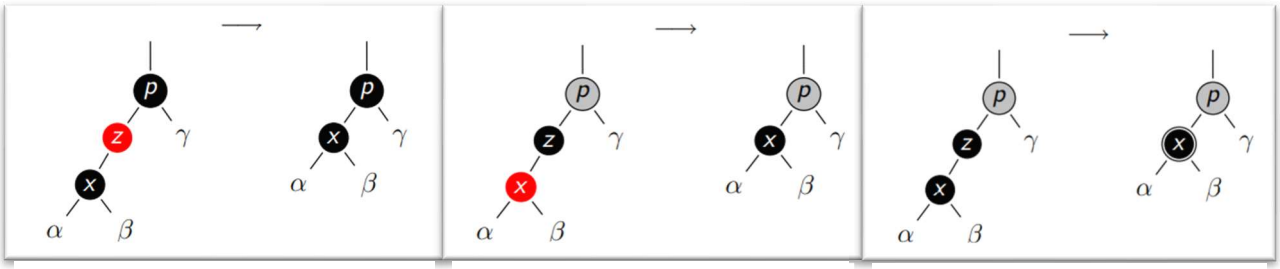
Osserviamo che, dopo aver scambiato le posizioni del nodo z e y, z avrà al più un figlio (quello destro), per le proprietà degli alberi binari di ricerca

Quando andiamo ad eliminare un nodo con al più un figlio possiamo trovarci di fronte a 3 casi:

1. z è rosso
2. z è nero e suo figlio è rosso
3. z è nero e suo figlio è nero

I 3 casi sono gestibili nei seguenti modi

1. Cancello il nodo z e sistemo la struttura dei puntatori
2. Cancello il nodo z e per non variare la black height assegno il colore nero al figlio di z. In seguito sistemo la struttura dei puntatori
3. Cancello il nodo z e per non variare la black height aggiungo un extra credito al figlio di z. In questo caso il figlio di z avrà il colore "Doppio nero".



Caso 1 *Caso 2* *Caso 3*

Il colore "doppio nero" serve per ricordarci che abbiamo collassato 2 nodi neri in uno. Esso nel conteggio della black height varrà doppio, ma dovrà essere spinto verso la radice, dove verrà ignorato. Se durante il cammino verso la radice incontriamo dei nodi rossi, il nodo verrà semplicemente colorato di nero

La procedura che si occupa della gestione del colore "doppio nero" è RB-DELETE-FIXUP.

Nella gestione del ripristino della proprietà 5 tramite la procedura RB-DELETE-FIXUP dobbiamo tener conto di altri 3 casi che variano in base al colore del nodo "fratello" w e dei suoi figli

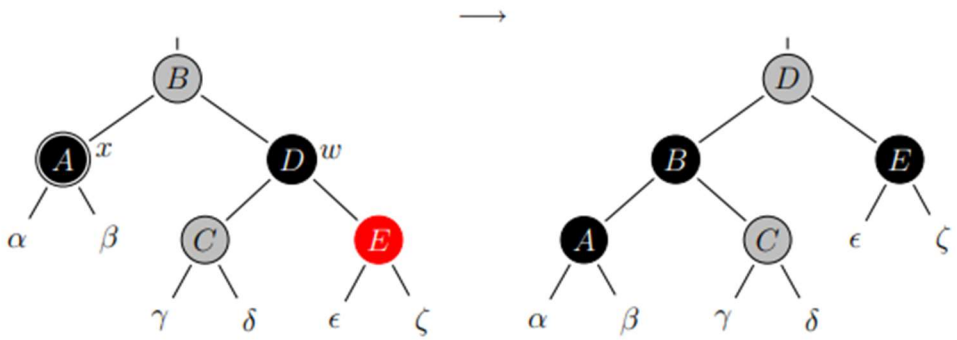
1. w è nero e ha almeno uno dei 2 figli rosso. Possiamo distinguere altri 2 sottocasi
    a. Il figlio destro di w è rosso
    b. Il figlio sinistro di w è rosso
2. w è nero e non ha figli rossi. Anche in questo caso possiamo distinguere 2 sottocasi
    a. p[x] è rosso
    b. p[x] è nero
3. w è rosso

I casi enunciati possono essere gestiti nel seguente modo

Caso 1a:

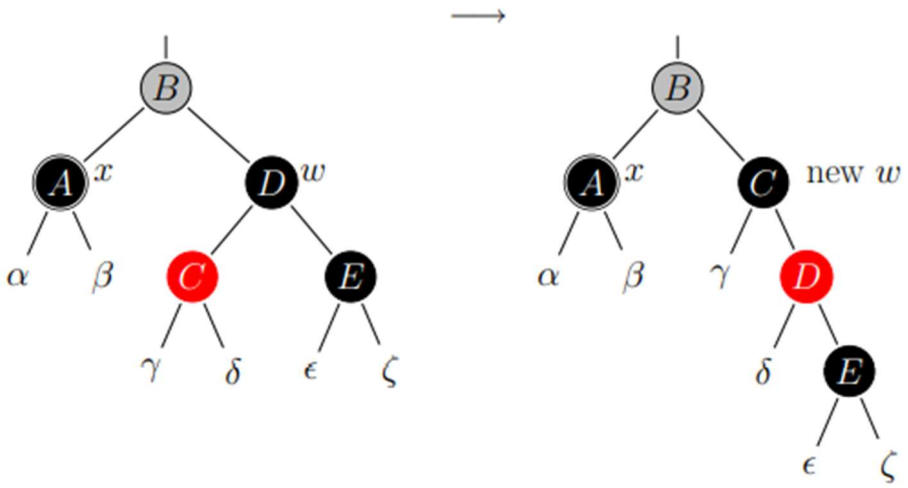
Scambiamo il colore di w con quello di p[x] e ruotiamo p[x] a sinistra

Abbiamo aggiunto un livello nero al sottoalbero sinistro e ne abbiamo tolto uno a destra. Andiamo quindi a compensare assegnando il colore nero sul sottoalbero destro di w e rimuoviamo il doppio nero da x



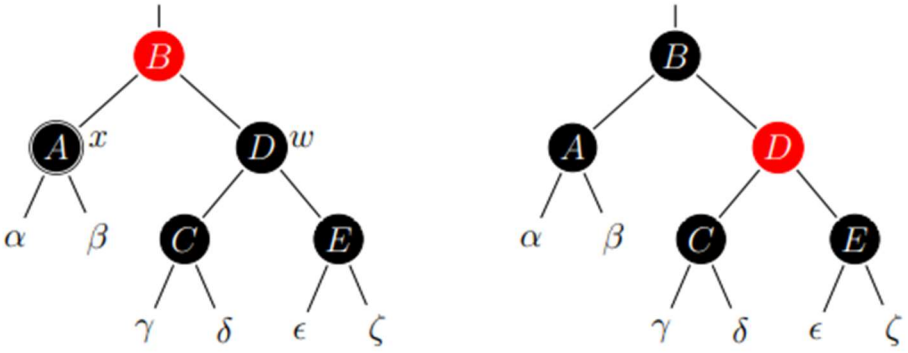
Caso 1b:

Ruotiamo w a destra e richiamiamo nuovamente la procedura RB-DELETE-FIXUP. In questo modo ricadiamo nel caso 1a



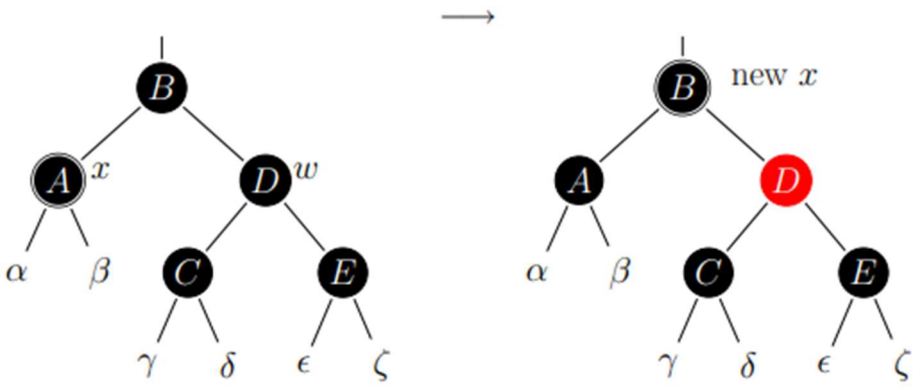
Caso 2a:

Togliamo un credito nero sia da x che da w e aggiungiamo un credito nero a p[x].



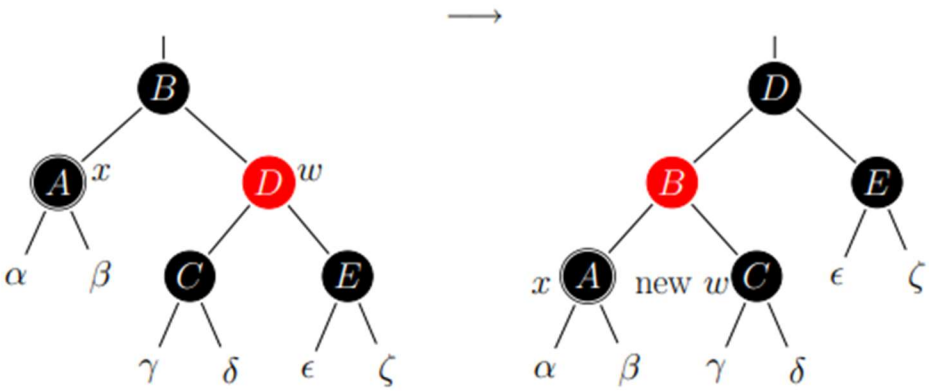
Caso 2b:

Identico al caso 2a ma in questo caso, dato che aggiungiamo un credito nero a un nodo nero, il nodo p[x] assumerà colore doppio nero. Richiamiamo nuovamente la procedura considerando come x il nodo p[x]



Caso 3:

Nel caso in cui w sia rosso, scambiamo i colori di p[x] e w e ruotiamo a sinistra p[x]. Il caso 3 viene ricondotto a uno dei casi precedenti. Possiamo richiamare nuovamente la procedura RB-DELETE-FIXUP.



La procedura RB-DELETE-FIXUP ha una complessità nel caso peggiore pari a $O(\log_2(n))$, dato che potrebbero essere effettuate un numero di chiamate pari all'altezza dell'albero.
Dato che RB-DELETE-FIXUP ha una complessità nel caso peggiore pari a $O(\log_2(n))$, allora RB-DELETE ha a sua volta una complessità pari a $O(\log_2(n))$

2. Aggiunta di caratteristiche ad una struttura dati

Nella maggior parte delle situazioni non avremo mai bisogno di reinventare una struttura dati da 0, ma possiamo utilizzare le classiche strutture dati per risolvere la maggioranza dei problemi. Le classiche strutture dati non bastano però per risolvere tutte le situazioni, a volte abbiamo bisogno di aggiungere delle variazioni ad esse. Invece di reinventare una struttura dati per risolvere uno specifico problema possiamo utilizzare le strutture dati più famose, adattandole ad uno specifico problema, aggiungendo informazioni ad esse mantenendo però la maggioranza delle operazioni sulla struttura totalmente invariate

Come aggiungere funzionalità ad una struttura dati

Il processo di aggiunta di funzionalità ad una struttura dati può essere diviso in quattro parti

- Scegliere una struttura dati da modificare
- Determinare quali informazioni aggiungere alla struttura dati
- Verificare se è possibile mantenere le informazioni con le operazioni base della struttura dati
- Aggiunta delle nuove operazioni

Questo processo di aggiunta di funzionalità non è per nulla restrittivo, infatti nulla ci vieta di non seguire alla lettera i passaggi descritti da questo processo, ma aiuta a focalizzare i punti cardine del processo

Quando andiamo ad aggiungere delle informazioni alla nostra struttura dati è bene dimostrare che è possibile mantenere le informazioni senza andare a intaccare le complessità delle operazioni sulla struttura dati

3. Interval Trees

Cerchiamo adesso, di adattare un Red-Black Tree a supportare operazioni su un set di intervalli
Un intervallo non è altro che una coppia di valori [t1, t2], con **t1 <= t2**. Possiamo adattare i Red-Black Trees a supportare intervalli aperti contenenti valori pari a inf, è possibile però limitarci allo studiare i casi in cui gli intervalli siano chiusi e interpretare i valori inf con i valori costanti MAX_INT e MIN_INT.

Gli intervalli possono essere rappresentati tramite classi che contengono i seguenti campi i.high e i.low, il primo indicante il limite superiore dell'intervallo, mentre il secondo indicante il limite inferiore dell'intervallo

Presa una qualunque coppia di intervalli i e i' sappiamo per certo che andranno a rispettare uno di questi 3 casi:

- I 2 intervalli si sovrappongono
- i è alla sinistra di i' (i.high < i'.i.low)
- i è alla destra di i' (i.low > i'.high)

b. Da Red-Black Tree a Interval Tree

Gli interval trees non sono altro che dei Red-Black Trees ma con operazioni che prevedono la presenza di intervalli

Nell'aggiunta delle caratteristiche descritte possiamo distinguere i passaggi descritti nel capitolo precedente (2. Aggiunta di caratteristiche ad una struttura dati)

- **Scelta di una struttura dati:** Utilizziamo un Red-Black Tree come struttura dati di riferimento, utilizzando però come chiave di ordinamento il campo i.low
- **Informazioni aggiuntive da mantenere:** Ogni nodo oltre a mantenere i dati dell'intervallo contenuto, conterranno anche un valore i.max, che indica il valore massimo contenuto nei sottoalberi radicati in x. Questo campo sarà fondamentale per l'implementazione delle operazioni di ricerca
- **Costi nel mantenimento delle informazioni:** E' possibile dimostrare che il mantenimento del campo i.max richiede tempo O(lg(n)). Non va quindi a intaccare la complessità dell'inserimento e dell'eliminazione
- **Implementazione delle nuove operazioni:** Per mantenere le nuove informazioni è possibile utilizzare le classiche operazioni di inserimento e di eliminazione, ma dobbiamo però implementare una nuova operazione di ricerca

c. Ricerca in un Interval Tree

La procedura di ricerca in un interval tree prende in input un intervallo i e scansiona l'albero alla ricerca di un intervallo che si sovrappone con l'intervallo i

La procedura di ricerca ritorna un puntatore ad un nodo nil nel caso in cui non trovasse alcun nodo che si sovrappone all'intervallo ricevuto in input

Pseudo codice

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left
5      else x = x.right
6  return x
```

La procedura comincia a scansionare l'albero dalla radice, fino a quando non trova un intervallo che si sovrappone con i. La scansione come abbiamo detto procede dall'alto verso il basso, la procedura scansionerà il sottoalbero sinistro nel caso in cui esso abbia un valore max maggiore di i.low, altrimenti scansionerà il sottoalbero destro

Si dimostra che se l'algoritmo di ricerca all'interno di un Interval Tree è corretta

c1. Dimostrazione correttezza procedura di ricerca

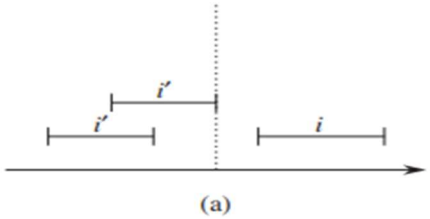
Per dimostrare la correttezza della procedura di ricerca utilizziamo la seguente proprietà invariante: **Se esiste un intervallo all'interno dell'albero T che si sovrappone con i, allora il sottoalbero radicato in x contiene sicuramente un intervallo**

Inizializzazione: Dato che x all'inizio è settato alla radice è banale dire che all'inizio l'invariante vale

Mantenimento: Ad ogni iterazione dell ciclo while vengono eseguite le linee di codice 4 o 5. Se la linea 5 viene eseguita significa che possiamo trovarci in 2 casi:

- **Il sottoalbero sinistro è vuoto:** Dato che il sottoalbero sinistro è vuoto spostarci nel sottoalbero destro è la scelta migliore, dato che è l'unico modo che abbiamo per trovare un'eventuale intervallo che si sovrappone con i

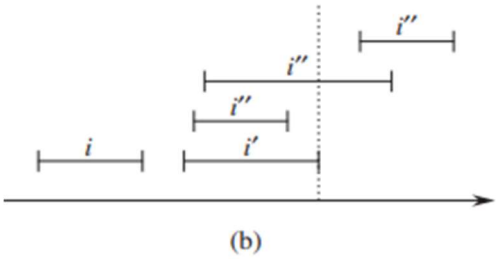
- **X.left.max < i.low:** In questo caso nel sottoalbero sinistro non troveremo sicuramente un intervallo che si sovrappone con i, dato che tutti i nodi presenti nel sottoalbero sinistro avranno sicuramente x.low < i.low, inoltre non ci sarà nessun nodo che avrà x.high > i.low, quindi è impossibile trovare un intervallo che si sovrappone con i.



Nel caso in cui venga eseguita la linea 4 invece possiamo dire con certezza che se all'interno dell'albero è presente un intervallo che si sovrappone con i, si troverà sicuramente nel sottoalbero sinistro.
Spostandoci nel sottoalbero sinistro possiamo trovarci di fronte a 2 casi

- **E' presente un intervallo che si sovrappone con i:** allora i.high > x.low
- **Non è presente un intervallo che si sovrappone con i:** allora x.low > i.high

Dato che l'albero è ordinato con chiave il limite inferiore dell'intervallo possiamo dire con certezza che se un intervallo che si sovrappone con i non è presente all'interno del sotto albero sinitro non si troverà neanche in quello destro, dato che nel sottoalbero destro avremo nodi con x'.low > x.low e che quindi non si sovrapporranno con i



In questo caso la ricerca va a sinistra. Con i' è indicato il nodo presente nel sottoalbero sinistro. Si può notare che se i' non si sovrappone con i allora nessun altro intervallo lo farà

4. Diagramma UML implementazione Interval Tree