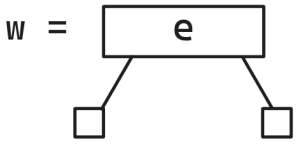
Lezione_23_DeA

Metodo `remove(k)`

```
w <- MWTreeSearch(k, T.root());
if (T.isExternal(w)) then {
     return null;
}
else{
     trova e∈w tale che e.getKey() = k;
     y <- e.getValue();
     if(altezza(w) = 1) then{ //se e solo se ha foglie
          Delete(e,w);
     }
     else{
          v <- figlio di w a sinistra di e;
          e` <- entry con chiave max in T_v; //O(log n) operazioni
          z <- nodo contenente e`; //z è ad altezza 1
          metti una copia di e` al posto di e in w;
          Delete(e`,z);
     }
}
decrementa di 1 il numerop di entry in T;
return y;
```

Osservazione

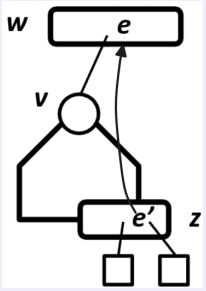
Nel caso in cui l'altezza di w sia uno, si rimuove e e la foglia alla sua destra. Se w va in underflow, cioè se e era l'unica entry in w, la `Delete` ripristina le proprietà del (2,4)-Tree.



Osservazione

Nel caso in cui l'altezza di w sia *maggiore* di uno, si sostituisce e' al posto di e in w e si rimuove e' da z insieme al suo figlio destro. Se dopo questa rimozione z va in underflow,

la `Delete` ripristina le proprietà del (2,4)-Tree.



6.6.5.2.1. Metodo `Delete` (ricorsivo)

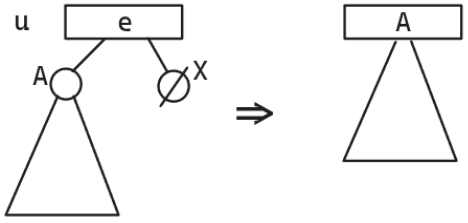
Metodo `Delete(e,u)`
Input: $u \in T$ con entry e, con un figlio foglia o vuoto a sinistra o destra di e.
Output: rimozione di e da T ripristinando le proprietà di (2,4)-Tree.

Siano A, X i figli di u discriminati da e dove X è foglia o vuoto.


```
rimuovi e, X;
if (u non ha più entry) then{
     Caso 1: u radice;
     Casi 2 e 3: u con un fratello (sinistro o destro) d-node, d = 3,4
     Casi 4 e 5: u con solo fratellli 2-node;
     //Le operazioni da eseguire sono elencate in seguito
}
```

6.6.5.2.1.1. Caso 1: u radice

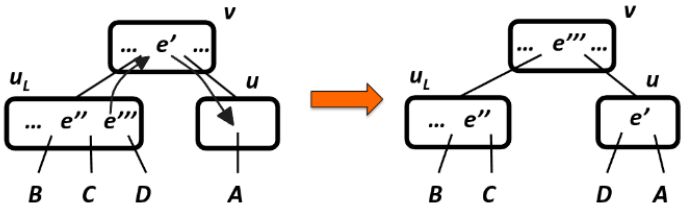
Operazione: `imposta A come nuova radice del (2,4)-Tree;`



> **Osservazione**
>
> Questo è il caso in cui l'altezza dell'albero diminuisce di uno.

6.6.5.2.1.2. Caso 2: u ha un fratello u_L a sinistra che è un d-node, con $d \geq 3$

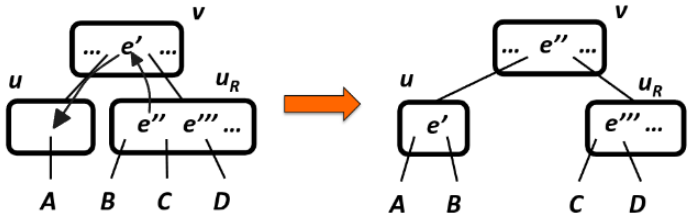
Operazione:



> **Osservazione**
>
> Con questa rotazione la `Delete` termina la sua esecuzione.

6.6.5.2.1.3. Caso 3: u ha un fratello u_R a destra che è un d-node, con $d \geq 3$

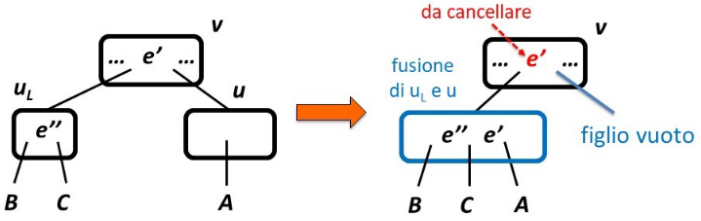
Operazione:



> **Osservazione**
>
> Con questa rotazione la `Delete` termina la sua esecuzione.

6.6.5.2.1.4. Caso 4: u ha un fratello u_L a sinistra che è un 2-node

Operazione:

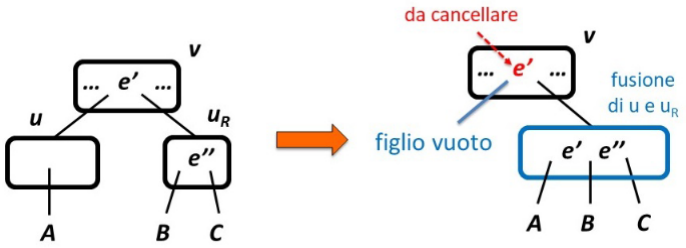


`Delete(e',v);`

> **Osservazione**
>
> Viene propagato l'underflow verso l'alto se la rimozione di e' da v genera underflow, cioè se e' era l'unica entry in v.

6.6.5.2.1.5. Caso 5: u ha un fratello u_R a destra che è un 2-node

Operazioni:



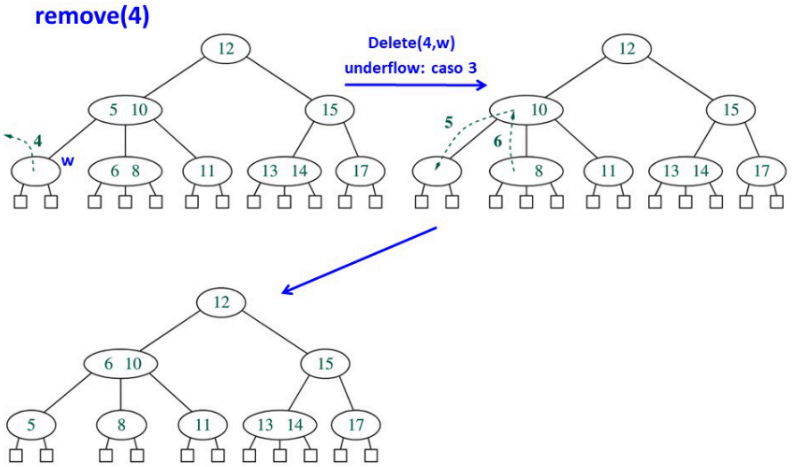

```
Delete(e',v);
```

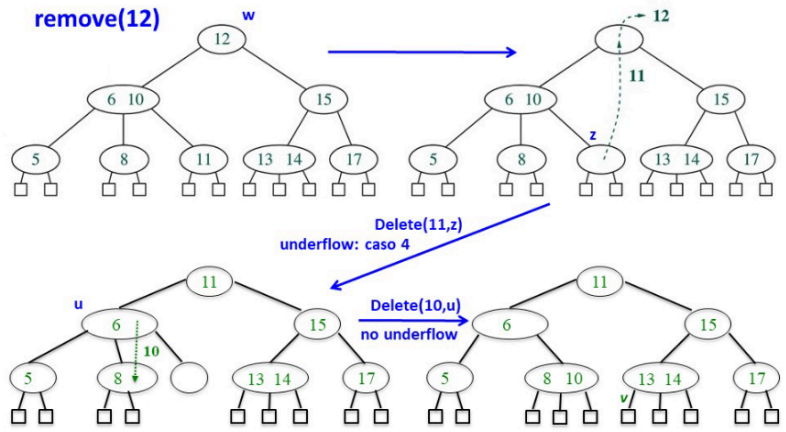
> **Osservazione**
>
> Viene propagato l'underflow verso l'alto se la rimozione di e' da v genera underflow, cioè se e' era l'unica entry in v.

> **Osservazione**
>
> I cinque casi possono essere esaminati nell'ordine 1-2-3-4-5 sino a trovare il primo che può essere applicato (uno sicuramente sarà trovato perché i cinque casi coprono tutti i possibili scenari).

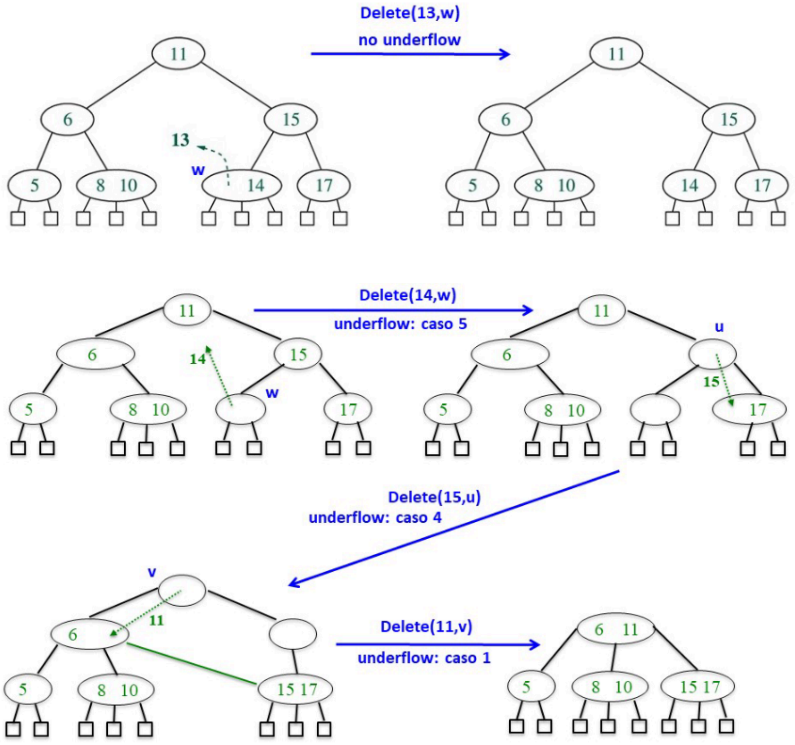
> **Osservazione**
>
> È facile vedere che in tutti e cinque i casi, le operazioni fatte mantengono valide le proprietà del (2,4)-Tree.

6.6.5.2.2. Esempi





remove(13)



6.6.5.2.3. Complessità

> **Proposizione**
>
> Per una mappa con n entry implementata tramite un (2,4)-Tree, la complessità di `remove` è $\Theta(\log n)$.

La complessità è dominata da:

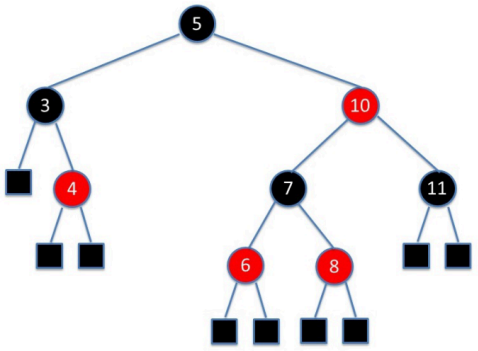
- `MWTreeSearch`: $\Theta(\log n)$ operazioni;
- Eventuale ricerca della entry e' con altezza 1 da sostituire al posto di e, se e e ad altezza maggiore di uno: $\Theta(\log n)$ operazioni;
- `Delete`: algoritmo ricorsivo:

- $\leq h$ invocazioni ricorsive perché parte da un nodo ad altezza uno e verrà invocata lungo il percorso da questo nodo alla radice, al più una volta per livello;
- Ogni invocazione ricorsiva richiede $\Theta(1)$ operazioni;
  $\implies$ Complessità di `Delete` è proporzionale all'altezza h del (2,4)-Tree che abbiamo dimostrato essere $\Theta(\log n)$;
  $\implies$ La complessità di `remove(k)` è $\Theta(\log n)$.
  □

6.7. Red-Black Tree

I Red-Black Tree sono Alberi Binari di Ricerca che, *a differenza del caso generale*, hanno *altezza sempre logaritmica* nel numero di nodi.

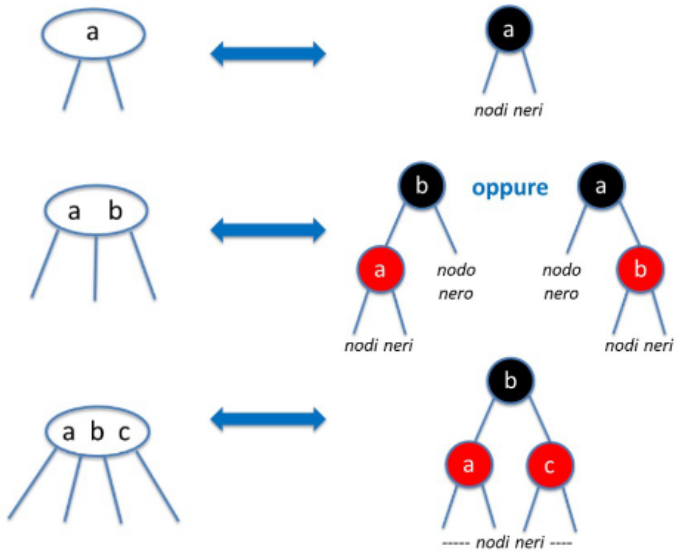
> Un *Red-Black Tree* (RB-Tree) T è un ABR i cui nodi hanno un colore rosso o nero e in cui valgono le seguenti proprietà:
>
> - La radice è nera (*Root Property*);
> - Le foglie sono nere (*Esternal Property*);
> - I figli di un nodo rosso sono neri (*Red Property*);
> - Tutte le foglie hanno la stessa *black depth*, ovvero lo stesso numero di antenati propri neri (*Depth Property*).



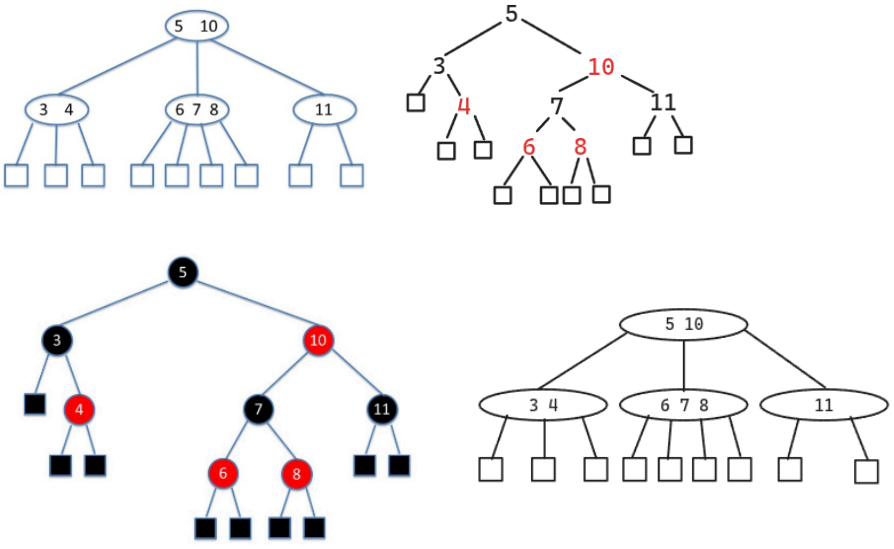
6.7.1. (2,4)-Tree e Red-Black Tree

È possibile trasformare un *(2,4)-Tree in un Red-Black Tree e viceversa*, applicando le seguenti trasformazioni dalla radice verso

le foglie:



6.7.1.1. Esempi



6.7.2. Metodi della Mappa su Red-Black Tree

Valgono le seguenti proprietà:

- Un Red-Black Tree contenente n entry ha *altezza* $\Theta(\log n)$.
  È una diretta conseguenza delle trasformazioni: passando da un (2,4)-Tree a un Red-Black Tree l'altezza al più raddoppia, mentre nel caso opposto al più si dimezza.
- I metodi `get`, `put` e `remove` della Mappa possono essere implementati in un Red-Black Tree con *complessità* $\Theta(\log n)$ al caso pessimo.

> **Osservazione**
>
> L'implementazione dei metodi della mappa risulta efficiente in pratica a parte la TreeSearch, per tutti e tre i metodi le altre

operazioni sono in numero costante.
In Java la classe `TreeMap<k,V>` implementa una Mappa tramite Red-Black Tree.

6.8. Multimappa

> Una *Multimappa* è una Mappa che *ammette la presenza di più entry con la stessa chiave*. Questa differenza richiede una modifica della specifica dei metodi caratterizzanti.

6.8.1. Metodi caratterizzanti

- `get(k)`: restituisce una collezione (eventualmente vuota) con tutti i valori associati alle entry con chiave k;
- `put(k,v)`: inserisce *sempre* una nuova entry (k, v) senza intaccare le altre entry con chiave k già presenti. Non restituisce alcun output;
- `remove(k,v)`: rimuove una entry con chiave k e valore v, se tale entry esiste. Restituisce un boolean: `true` se si è rimossa la entry, `false` altrimenti.

> **Osservazione**
>
> Se esistono più entry (k, v) si può scegliere se rimuoverne una arbitraria o tutte.

Una Multimappa può essere implementata tramite una Mappa in cui le entry sono costituite da coppie (k, L_k), dove k è una chiave e L_k *una lista*, non vuota, di valori associati alla chiave.
Se $L_k = \{v_1, v_2, \ldots, v_l\}$, allora (k, L_k) rappresenta, in modo compatto, le l entry (k, v_1), (k, v_2), $\ldots$, (k, v_l).

6.8.1.1. Implementazione

- `get(k)`: se esiste una entry (k, L_k) restituisce L_k, altrimenti restituisce una collezione vuota;
- `put(k,v)`: se esiste una entry (k, L_k) si aggiunge semplicemente il valore v a L_k, altrimenti si aggiunge la entry $(k, L_k = \{v\})$ alla struttura;
- `remove(k,v)`:
  - Se esiste una entry (k, L_k) e $L_k = \{v\}$, si rimuove la entry (l, L_k) e si restituisce `true`;

- Se esiste una entry (k, L_k) e L_k contiene v e altri valori, si rimuove v da L_k e si restituisce `true`;
- In tutti gli altri casi si restituisce `false`, senza modificare la Multimappa.

Osservazione

Nel caso esistano più coppie della entry (k, v) da rimuovere, si può decidere di rimuovere una sola o tutte.