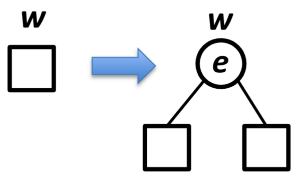
Lezione_21_DeA

6.5.2.3.2. `put`

Metodo `put(k,x)`

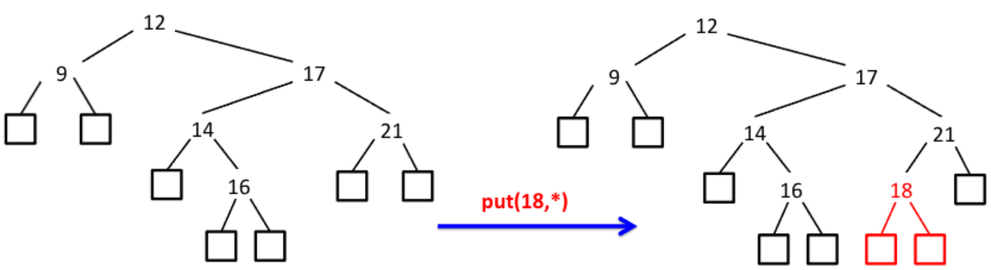
```
w <- TreeSearch(k, T.root());
if (T.isInternal(w)) then {
     y <- w.getElement().getValue();
     sostituisce x a y nella entry w.getElement();
     return y;
}
expandExternal(w, e = (k,x));
incrementa numero di entry di T di 1;
return null;
```

Il metodo `expandExternal(w,e)` trasforma w in nodo interno contenente la entry e:



La *complessità* è $\Theta(h)$ perché è dominata dalla `TreeSearch`.

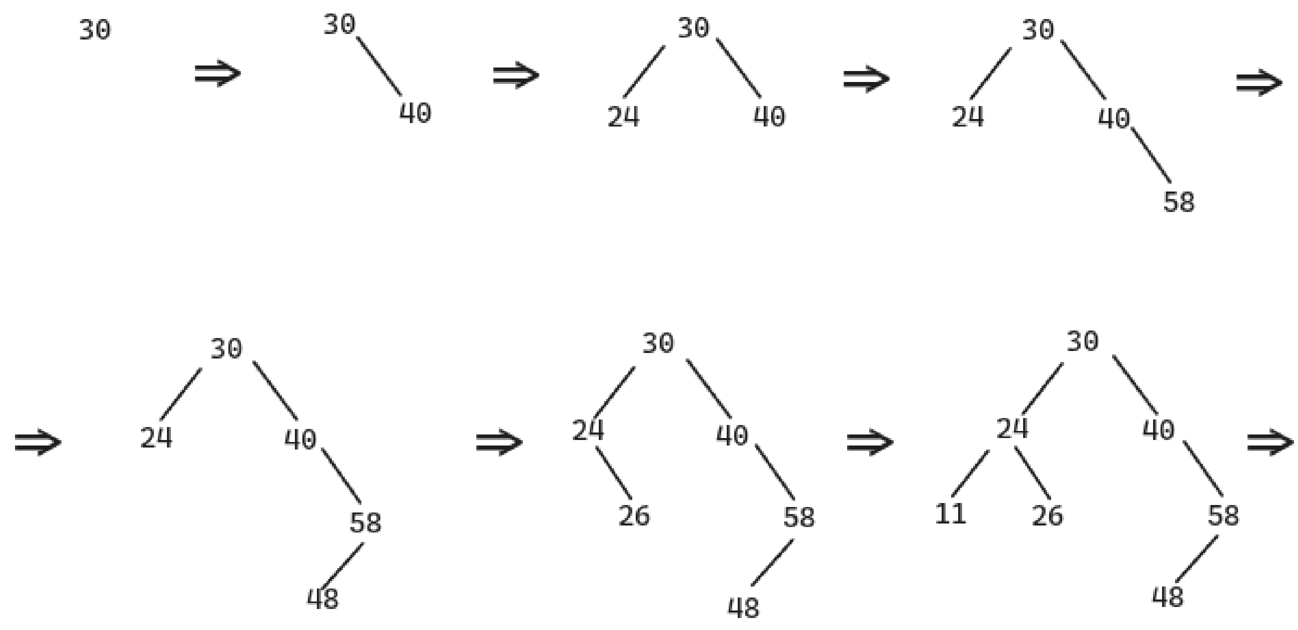
6.5.2.3.2.1. Esempio (solo chiavi)

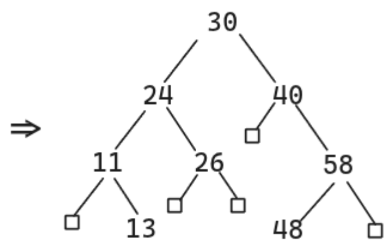


6.5.2.3.2.2. Esercizio

> Inserire un un albero di ricerca vuoto otto entry con chiavi 30, 40, 24, 58, 48, 26, 11, 13 e far vedere l'albero risultante dopo ciascun inserimento.

(senza foglie)





⟸ Albero finale con foglie.

6.5.2.3.3. `remove`

Metodo `remove(k)`

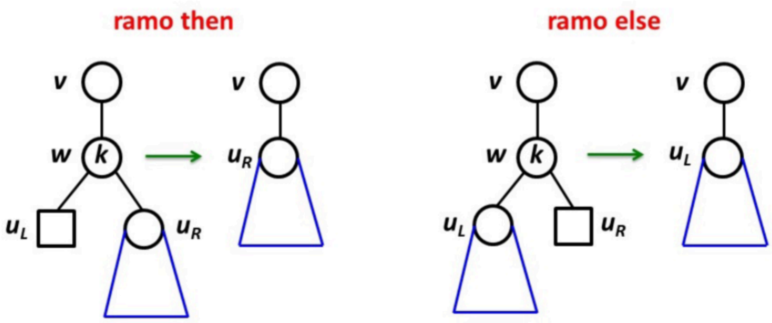
```
w <- TreeSearch(k, T.root());
if (T.isExternal(w)) then {
     return null;
}
else{
     value <- w.getElement().getValue();
     decrementa il numero di entry in T di 1;
     if((T.isExternal(T.left(w)) OR (T.isExternal(T.right(w))) then {
          //Caso 1: w interno con almeno 1 figlio foglia
          //funziona correttamente anche se entrambi i figli di w sono
entrambi foglie
          u_L <- T.left(w);
          u_R <- T.right(w);
          if (T.isExternal(u_L)) then {
               cancella w e u_L;
               fai salire u_R al posto di w;
          }
          else{
```

```
                cancella w e u_R;
                fai salire u_L al posto di w;
            }
        }
        else{
            //Caso 2: w interno con due figli interni
            y <- nodo con chiave max nel sottoablero sinistro di w;
            sposta la entry in y nel nodo w;
            cancella y e il suo figlio destro;
            fai salire il figlio sinistro di y al posto di y;
        }
}
return value
```

6.5.2.3.3.1. Caso 1

In questo caso w ha almeno un figlio foglia.

```
u_L <- T.left(w);
u_R <- T.right(w);
if (T.isExternal(u_L)) then {
    cancella w e u_L;
    fai salire u_R al posto di w;
}
else{
    cancella w e u_R;
    fai salire u_L al posto di w;
}
```



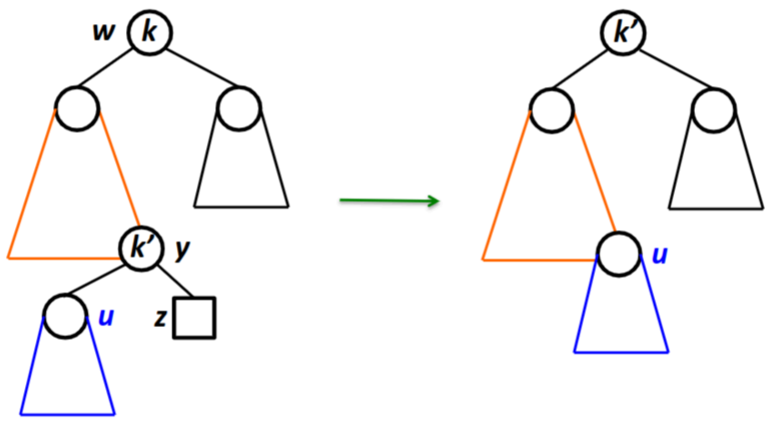
> **Osservazione**
>
> Se w era radice, allora la nuova radice è il figlio messo al suo posto.

> **Nota bene**
>
> Il caso 1 funziona correttamente anche se entrambi i figli di w, u_L e u_R, sono foglie.

6.5.2.3.3.2. Caso 2

In questo caso w ha due figli interni.

```
y <- nodo con chiave max nel sottoablero sinistro di w;
sposta la entry in y nel nodo w;
cancella y e il suo figlio destro;
fai salire il figlio sinistro di y al posto di y;
```



La prima istruzione ha $\Theta(h)$ operazioni, mentre le altre tre $\Theta(1)$ operazioni.

> **Osservazione**
>
> y è il predecessore "interno" di w nella visita inorder e contiene la chiave massima (k') tra quelle nel sottoalbero sinistro di w, cioè quella che precede k nell'ordine crescente delle chiavi.

6.5.2.3.3.3. Complessità

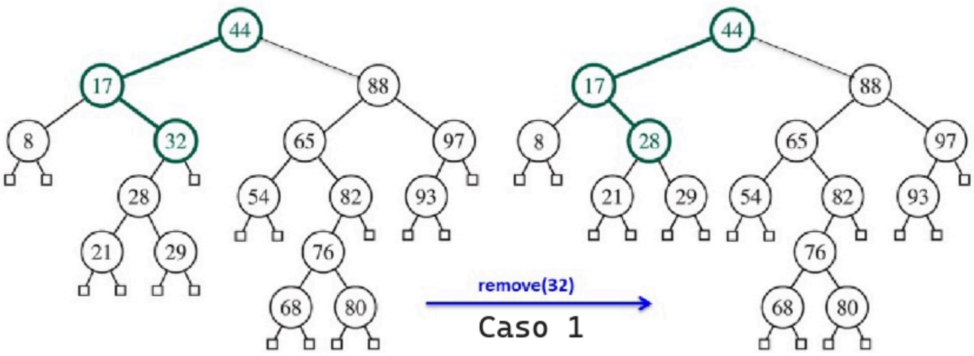
Sia h l'altezza di T. La complessità di `remove(k)` è ottenuta sommando i seguenti contributi:

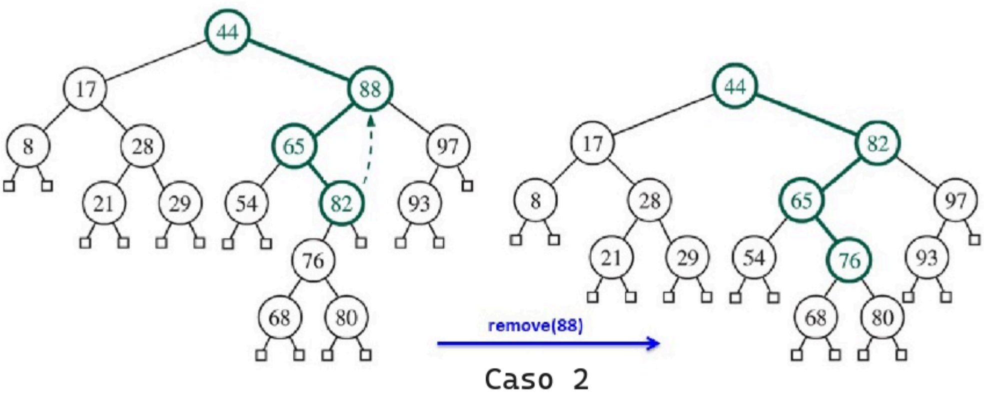
- `TreeSearch`: $\Theta(h)$;
- Nel caso 2, la ricerca di y: $\Theta(h)$;

- Altre operazioni: $\Theta(1)$;

  $\Longrightarrow$ Complessità $\in \Theta(h)$.

6.5.2.3.3.4. Esempio





6.5.2.3. `TreeSearch` iterativo

Algoritmo `TreeSearch(T,k)`
Input: ABR T e una chiave k;
Output: nodo $w \in T$ contenente una entry con chiave k, se esiste, o una foglia "giusta" per k.

```
w <- T.root();
while (T.isInternal(w)) do{
     x <- w.getElement().getKey();
     if(x = k) then{
          return w;
     }
     if(k < x) then{
          w <- T.left(w);
     }
     else{
          w <- T.right(w);
     }
}
return w;
```

La *complessità* è $\Theta(h)$, infatti:

- vengono eseguite $\Theta(1)$ operazioni fuori dal while;
- vengono eseguite $\Theta(h)$ operazioni del while al caso pessimo;
- vengono eseguite $\Theta(1)$ operazione in ciascuna iterazione del while.

6.5.3. Osservazioni sugli Alberi Binari di Ricerca

Una debolezza degli Alberi Binari di Ricerca si vede quando sono molto sbilanciati (ovvero quando $h \in \Theta(n)$ con n il numero di entry): la complessità dei tre metodi `get`, `put` e `remove` degradano sino a diventare quelle ottenibili con una semplice lista non ordinata. Allora un ABR non ha nessun valore aggiunto, al caso pessimo, rispetto alle liste o alla Tabella Hash.

Alcune direzioni possibili per migliorare gli ABR sono:

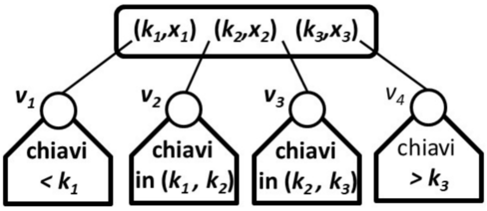
1. Ribilanciare T ogni volta che lo sbilanciamento supera una soglia prefissata (es: AVL, Red-Black Tree);
2. Rendere i nodi più capienti in modo da *assorbire* meglio gli effetti di inserimenti o rimozioni che potrebbero portare a uno sbilanciamento dell'albero (es: (2,4)-Tree).

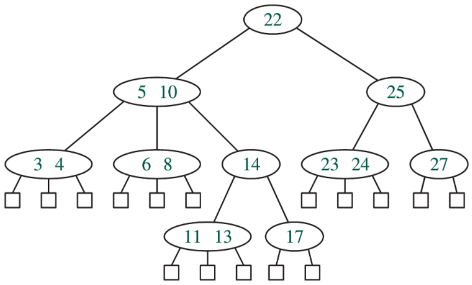
In seguito vedremo implementazioni efficienti della Mappa Ordinata basate su varianti degli Alberi Binari di Ricerca e discuteremo come generalizzare la Mappa per permettere chiavi duplicate (Multimappa). In particolare tratteremo di Multi-Way Search Tree (MWS-Tree), (2,4)-Tree, faremo cenni sui Red-Black Tree e parleremo di Multimappe.

6.6. Multi-Way Search Tree

6.6.1. Definizione, osservazioni, proposizione

> Un *Multi-Way Search Tree* (MWS-Tree) T è un albero *ordinato* tale che ogni nodo interno ha *almeno due figli*, ogni nodo interno con $d \geq 2$ figli $v_1, v_2, \dots v_d$ (*d-node*) soddisfa le seguenti proprietà:
>
> - *Memorizza $d-1$ entry* (k_1, x_1), (k_2, x_2), ..., (k_{d-1}, x_{d-1}) dove $k_1 < k_2 < \dots < k_{d-1}$;
> - Per $1 \leq i \leq d$ vale che la chiave di ogni entry e memorizzata in un nodo T_{v_i} soddisfa la relazione $k_{i-1} < e.getKey() < k_i$ (assumendo $k_0 = -\infty$ e $k_d = +\infty$);
>   Per convenzione, le foglie sono delle sentinelle e non memorizzano entry (come negli ABR).





Osservazione

I Multi-Way Search Tree generalizzano gli Alberi Binari di Ricerca permettendo ai nodi di contenere più entry, cosa che rende più agevole il bilanciamento (nella variante (2,4)-Tree che vedremo in seguito).

Osservazione

Un Albero Binario di Ricerca è un MWS-Tree in cui ogni nodo è un 2-node.

Proposizione

Un MWS-Tree che memorizza n entry ha $n+1$ foglie.

6.6.1.1. Dimostrazione

Sia T un MWS-Tree che memorizza n entry.
Definiamo:

- $A = \{$nodi interni di $T\}$;
- $B = \{$foglie di $T\}$
- $\forall v \in T$ sia d_v il numero di figli di v:
  - se v è foglia, allora $d_v = 0$ e v non contiene entry;
  - se v è interno, allora $d_v > 0$ e v contiene $d_v - 1$ entry;
    $\implies n = \sum_{v \in A} (d_v - 1)$
    Ricordiamo la proprietà per cui $\sum_{v \in T} d_v = |A| + |B| - 1$ e inoltre vale che $\sum_{v \in T} d_v = \sum_{v \in A} d_v$.

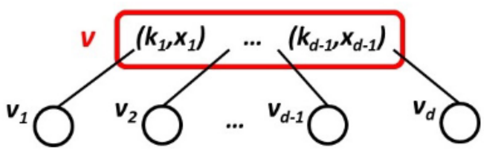
$n = \sum_{v \in A}(d_v - 1) = \left(\sum_{v \in A} d_v\right) - |A| = |A| + |B| - 1 - |A| = |B| - 1$
$\implies |B| = n + 1$
$\square$

> **Osservazione**
>
> Se tutti i nodi interni fossero dei d-node l'MWS-Tree T sarebbe un albero d-ario.
> Detto x il numero dei nodi interni e y il numero di foglie T sappiamo che $y = (d-1)x + 1$, che è coerente con la proposizione precedente dato che l'albero in questo caso memorizzerebbe $n = (d-1)x$ entry.

6.6.2. Ricerca in un MWS-Tree

Algoritmo `MWTreeSearch(k,v)`
Input: chiave k, nodo $v \in T$.
Output: nodo di T_v contenente una entry con chiave k, se esiste, o foglia in posizione "giusta" per k.

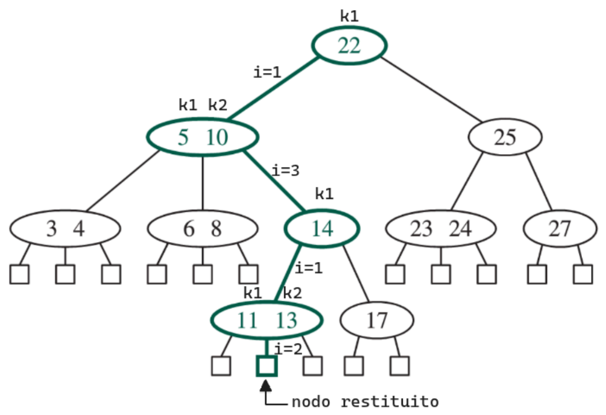
```
if(T.isExternal(v)) then{
     return v;
}
//Siano (k1, x1), ..., (k_{d-1}, x_{d-1}) le entry in v, con k1 < ... <
k_{d-1}, e siano v1, v2, ..., v_d i figli di v (Vedi immagine sottostante)
Trova i tale che k_{i-1} < k <= k_i; //si assuma k0 = -∞, k_d = +∞
if (k = k_i) then{
     return v;
}
else{
     return MWTreeSearch(k, v_i);
}
```



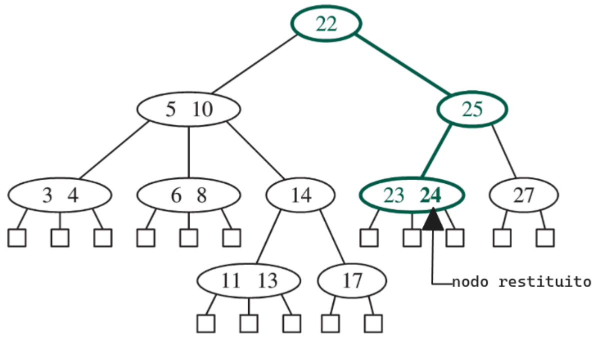
La *chiamata iniziale* per cercare in tutto l'albero corrisponde a `v = T.root()`.

6.6.2.1. Esempi

`MWTreeSearch(12, T.root())`



`MWTreeSearch(24, T.root())`



6.6.2.2. Complessità

Ipotesi: le entry in un nodo interno sono memorizzate in una lista ordinata.

L'albero della ricorsione per `MWTreeSearch(k, T.root())`:

- Ha $\leq h+1$ chiamate ricorsive, dove h è l'altezza di T;
- Il costo associato a ciascuna chiamata ricorsiva è $\Theta(d_{max})$ al caso pessimo, dove d_{max} è il massimo numero di figli di un nodo $\implies$ Complessità $\in \Theta(hd_{max})$.

La complessità non migliora in generale quella della TreeSearch per gli ABR in quanto per i MWS-Tree non abbiamo limiti superiori su altezza e massimo numero di figli, se non (per entrambi) il limite banale di n, dove n è il numero di entry in T.