Dati e Algoritmi: A. Pietracaprina

Mappe

(Parte 2)

Osservazioni sugli Alberi Binari di Ricerca

DEBOLEZZA degli ABR: Se l'ABR è molto sbilanciato ($\Rightarrow h \in \Theta(n)$ con $n = \#entry$) la complessità dei 3 metodi get, put, remove degrada fino a diventare quella ottenibile con una semplice lista non ordinata

$\Rightarrow$ nessun vantaggio, al caso peggiore, rispetto alle liste o alle Tabelle Hash

DIREZIONI POSSIBILI PER MIGLIORARE GLI ABR

(1) Ribilanciare T ogni volta che lo sbilanciamento supera una soglia prefissata
(Esempi: 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
(Esempi: (2,4)-Tree)

Nei lucidi seguenti 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, gli argomenti trattati saranno:

- Multi-Way Search Tree (MWS-Tree)
- (2,4)-Tree
- Cenni sui Red-Black Tree
- Multimappa

Multi-Way Search Tree

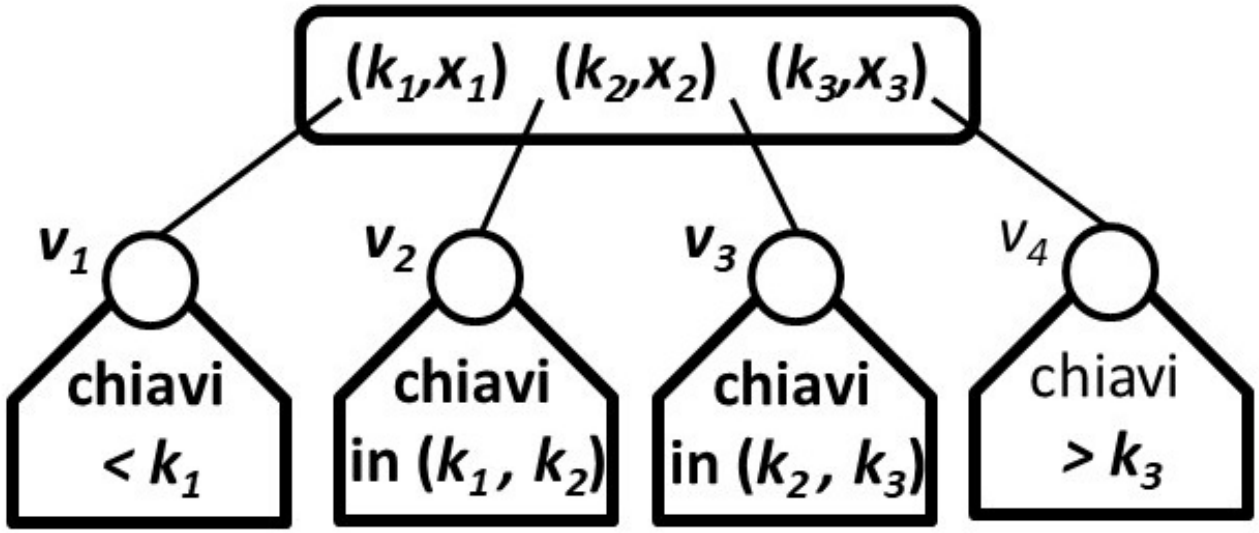
Definizione

Un Multi-Way Search Tree (MWS-Tree) T è un albero ordinato tale che

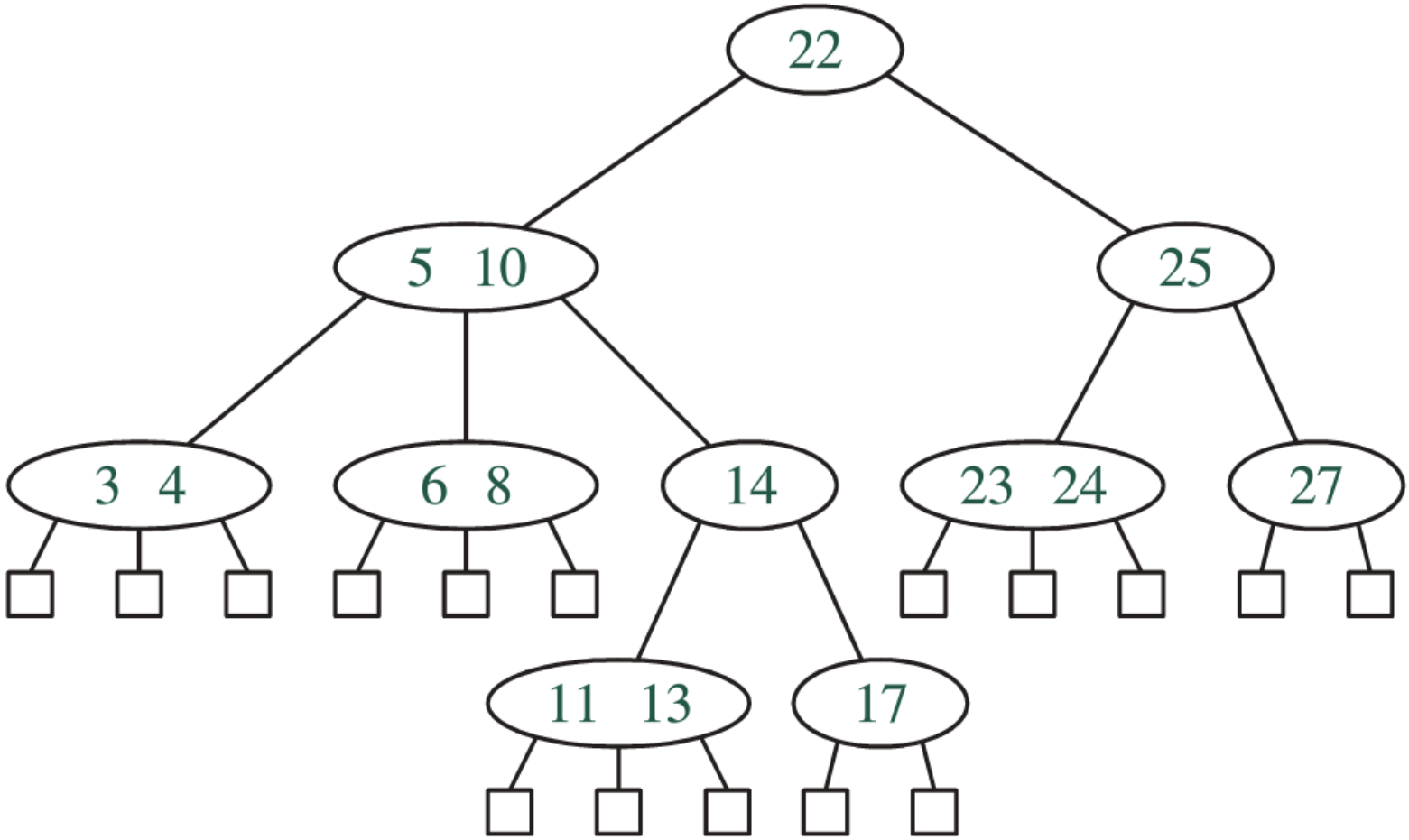
- ogni nodo interno ha ≥ 2 figli
- ogni nodo interno con $d \geq 2$ figli $v_1, v_2, \ldots, v_d$ (d-*node*) soddisfa le seguenti proprietà:
  - memorizza $d-1$ entry: $(k_1,x_1),(k_2,x_2),\ldots,(k_{d-1},x_{d-1})$ dove $k_1 < k_2 < \cdots < k_{d-1}$
  - per $1 \leq i \leq d$ vale che la chiave di ogni entry e memorizzata in un nodo di T_{v_i} soddisfa la relazione $k_{i-1} < e.\texttt{getKey()} < k_i$ (assumendo $k_0 = -\infty$ e $k_d = +\infty$).

Per convenzione, le foglie sono delle sentinelle e non memorizzano entry (come negli alberi binari di ricerca).

Esempio di 4-node



Esempio di MWS-Tree: $n = 15$ entry (solo chiavi)



Osservazioni

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

Proposizione (11.2 [GTG14])

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

Sia T un MWS-Tree che memorizza n entry

Definiamo:

* $A = \{$ nod interni di $T \}$
* $B = \{$ foglie di $T \}$
* $\forall v \in T$ sia $d_v = \#$ figli di v
  - se v è foglia $\Rightarrow d_v = 0$ e v non contiene entry
  - se v è interno $\Rightarrow d_v > 0$ e v contiene $d_v - 1$ entry

$$\Rightarrow n = \sum_{v \in A} (d_v - 1)$$

Dalle Slide 32 e 33 degli alberi generali (e anche Prop. 8.4 del testo) sappiamo che

$$\sum_{v \in T} d_v = |A| + |B| - 1$$

e inoltre vale che $\sum_{v \in T} d_v = \sum_{v \in A} d_v$

$$n = \sum_{v \in A} (d_v - 1)$$

$$= \left(\sum_{v \in A} d_v\right) - |A|$$

$$= \overbrace{|A| + |B| - 1}^{} - |A|$$

$$= |B| - 1$$

$$\Rightarrow |B| = n + 1$$ □

Esercizio: Fare una dimostrazione alternativa della proposizione, per induzione sull'altezza $h \geq 0$ di T

Osservazione

Se tutti i nodi interni fossero dei d-node l' MWS-Tree T sarebbe un albero d-ario. Detto x il numero di nodi interni e y il numero di foglie di T sappiamo (si veda la dispensa di esercizi sugli alberi) 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.

Ricerca in un MWS-Tree T

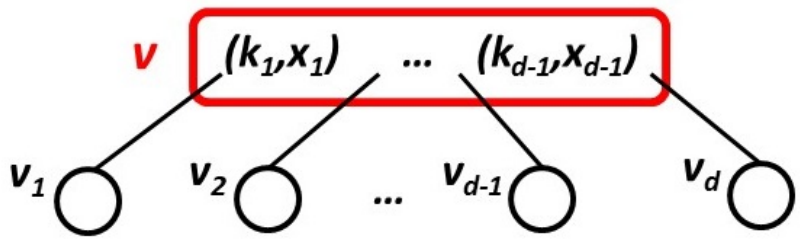
Algoritmo `MWTreeSearch(`k`,`v`)` // (Prima chiamata: $v = T$`.root()`)

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 (k_1, x_1),..., (k_{d-1}, x_{d-1}) *le entry in* v, *con* $k_1 < \cdots < k_{d-1}$, *e siano* $v_1, v_2, \ldots, v_d$ *i figli di* v;



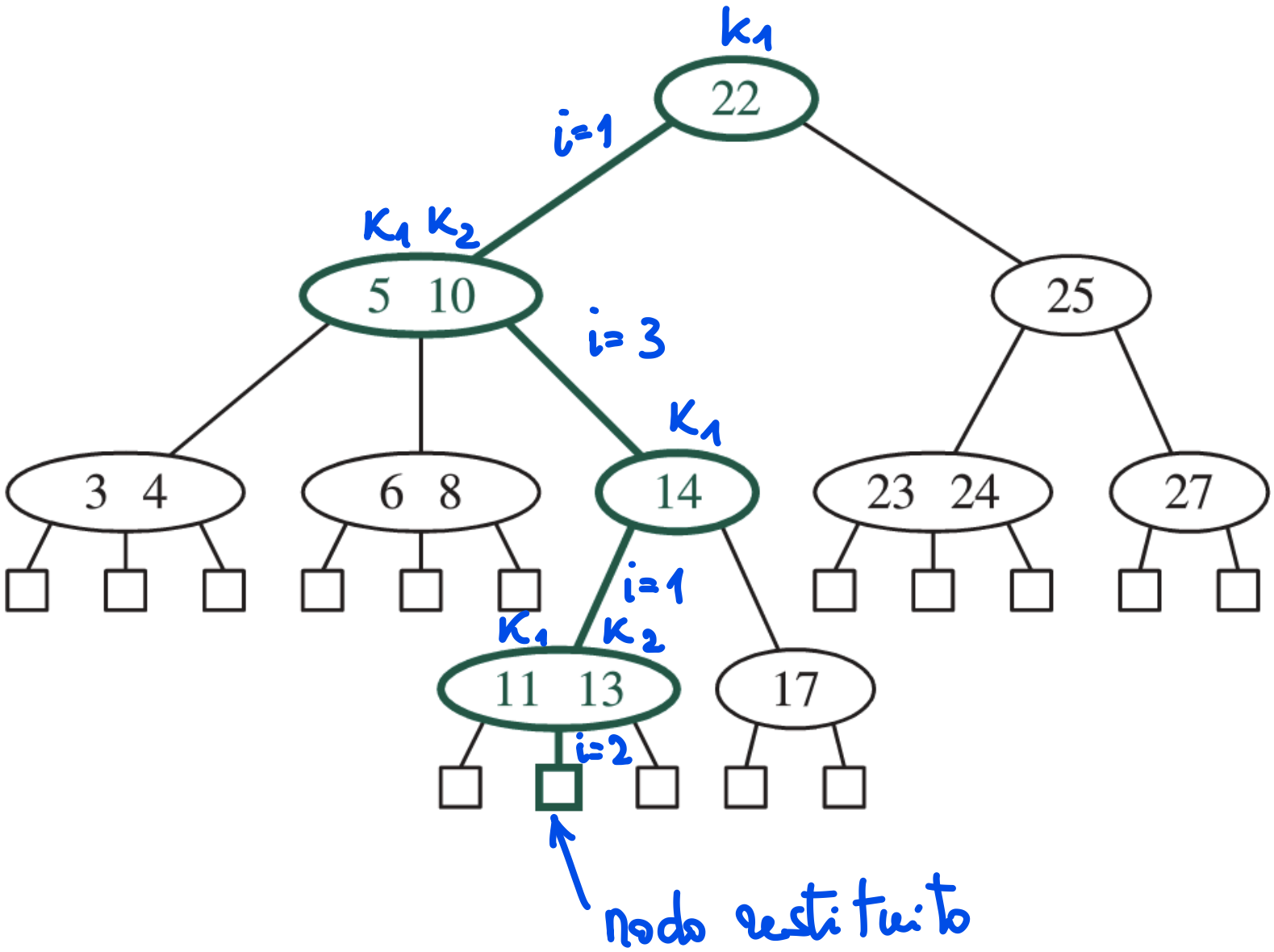
Trova i *tale che* $k_{i-1} < k \leq k_i$ (si assuma $k_0 = -\infty$, $k_d = +\infty$);

if (k=k_i) **then return** v;

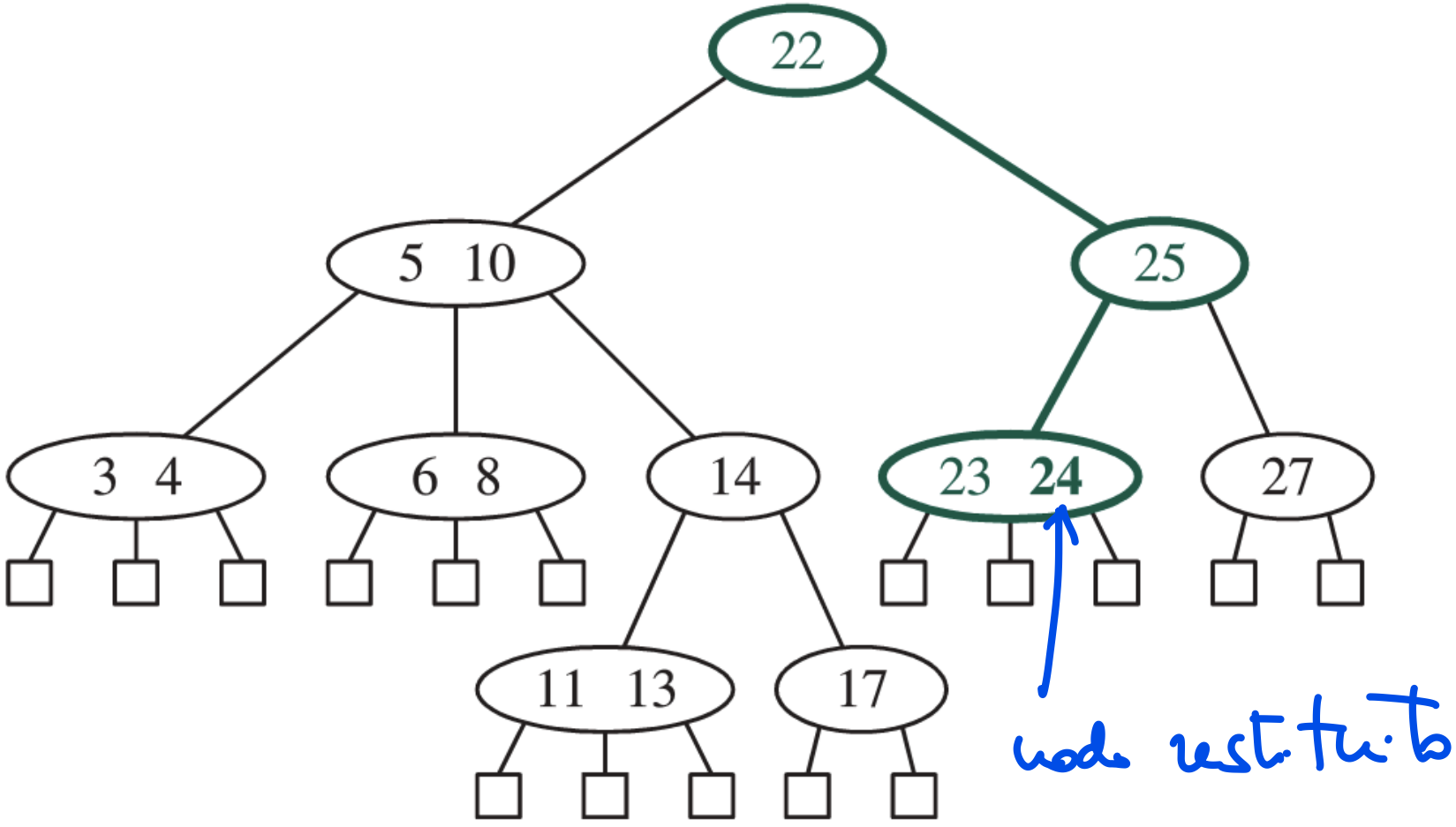
else return `MWTreeSearch(`k`,`v_i`)`;

Oss. In [GTG14] l'algoritmo è descritto solo a parole

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



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



Complessità di MWTreeSearch

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

Albero della ricorsione per MWTreeSearch(k, T.root())

* $\leq h+1$ chiamate ricorsive, con h = altezza di T
* Costo associato a ciascuna chiamata ricorsiva è $\Theta(d_{max})$ al caso peggiore, dove d_{max} è il massimo # di figli di un nodo

$\Rightarrow$ Complessità $\in \Theta(h\, d_{max})$

le complessità non migliora in generale quella della TreeSearch per gli ABR in quanto per i MWS-Tree non abbiamo limiti superiori su h e d max, se non (per entrambi) il limite banale di $n = \#$ entry in T

Implementazione dei metodi della Mappa: get

get(k**)**

$w \leftarrow$ `MWTreeSearch(`k`,`T`.root());`
if `(`T`.isExternal(`w`))` **then** **return** null;
else
    $trova\ e \in w\ tale\ che$ `e.getKey()`$= k$;
    return `e.getValue();`

Complessità: è dominata dal quella di `MWTreeSearch`, ed è quindi $O(d_{\max} h)$, dove $d_{\max}$ è il massimo numero di figli di un nodo e h è l'altezza dell'albero.

(2,4)-Tree

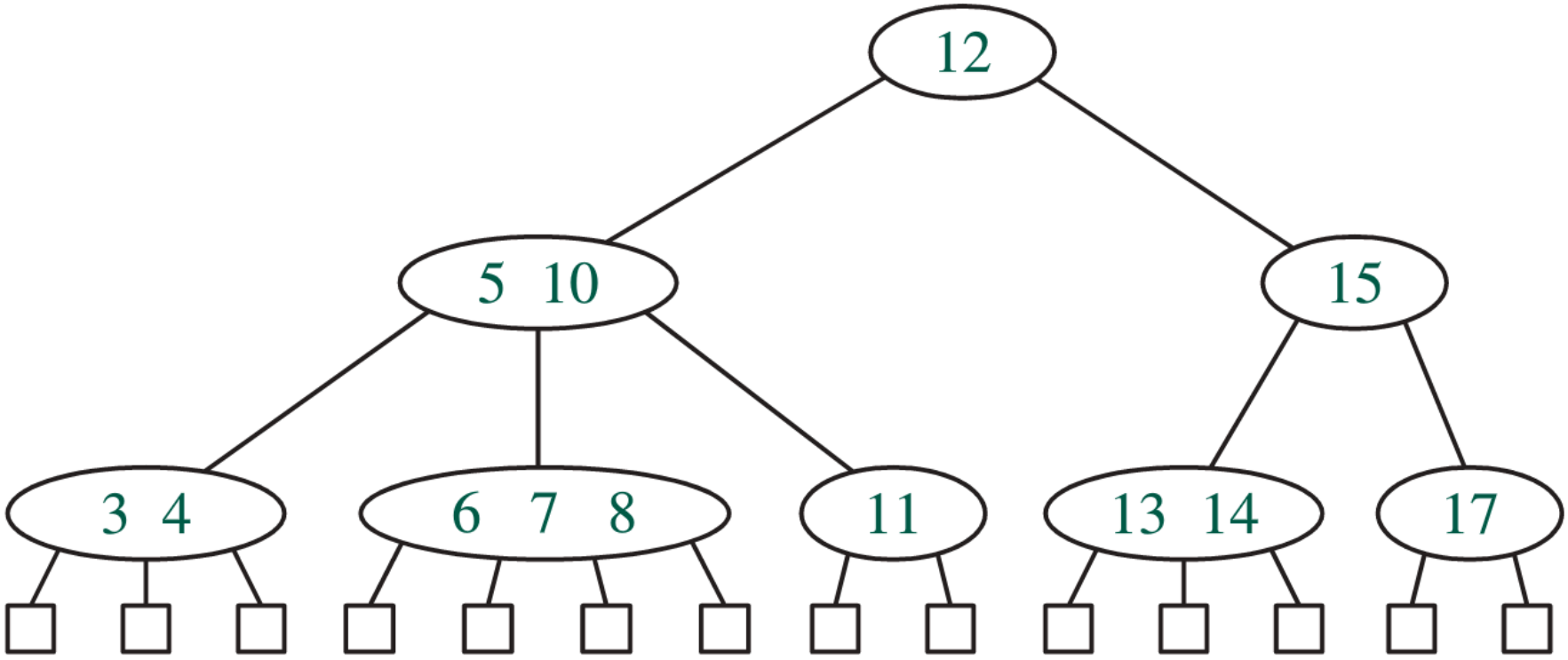
Definizione

Un (2,4)-Tree è un MWS-Tree tale che

- ogni nodo interno è un d-node con $2 \leq d \leq 4$ (quindi ha d figli e $d-1$ entry);
- tutte le foglie hanno la stessa profondità.

Osservazione: Si può definire un (2,3)-Tree con $2 \leq d \leq 3$ per il quale si applicano gli stessi algoritmi e le stesse complessità. Il vantaggio del (2,4)-Tree è che si generalizza al B-tree che è una struttura dati molto usata per la realizzazione di indici in memoria secondaria (ad es., nelle basi di dati).

Esempio di (2,4)-Tree



Altezza di un (2,4)-Tree

Proposizione 11.3 [GTG14]

Un (2,4)-Tree con $n > 0$ entry ha altezza $\Theta(\log n)$.

Il seguente corollario è una conseguenza immediata della proposizione e di quanto visto prima.

Corollario

In (2,4)-Tree con n entry, la complessità di `MWTreeSearch` e del metodo `get` della mappa è $\Theta(\log n)$.

Dimostrazione della Proposizione e del Corollario

Sia T un $(2,4)$-Tree con n entry e altezza h

$m_i = \#$ nod. al livello i, per $0 \le i \le h$. Si ha che

$$m_0 = 1$$

$$2 \le m_1 \le 4$$

$$2^2 \le m_2 \le 4^2$$

$$\vdots$$

$$2^i \le m_i \le 4^i$$

$$\vdots$$

$$2^h \le m_h \le 4^h$$

* So che, per definizione le foglie sono tutti i nodi del livello h

* Poiché il (2,4)-Tree è un MWS-Tree, so che il numero di foglie è n+1

$$\Rightarrow \quad 2^h \leq m_h = n+1 \leq 4^h$$

$$\Rightarrow \quad h \leq \log_2(n+1) \leq \log_2(4^h) = 2h$$

$$\Rightarrow \quad h \leq \log_2(n+1) \quad \text{e} \quad h \geq \log_2(n+1)/2$$

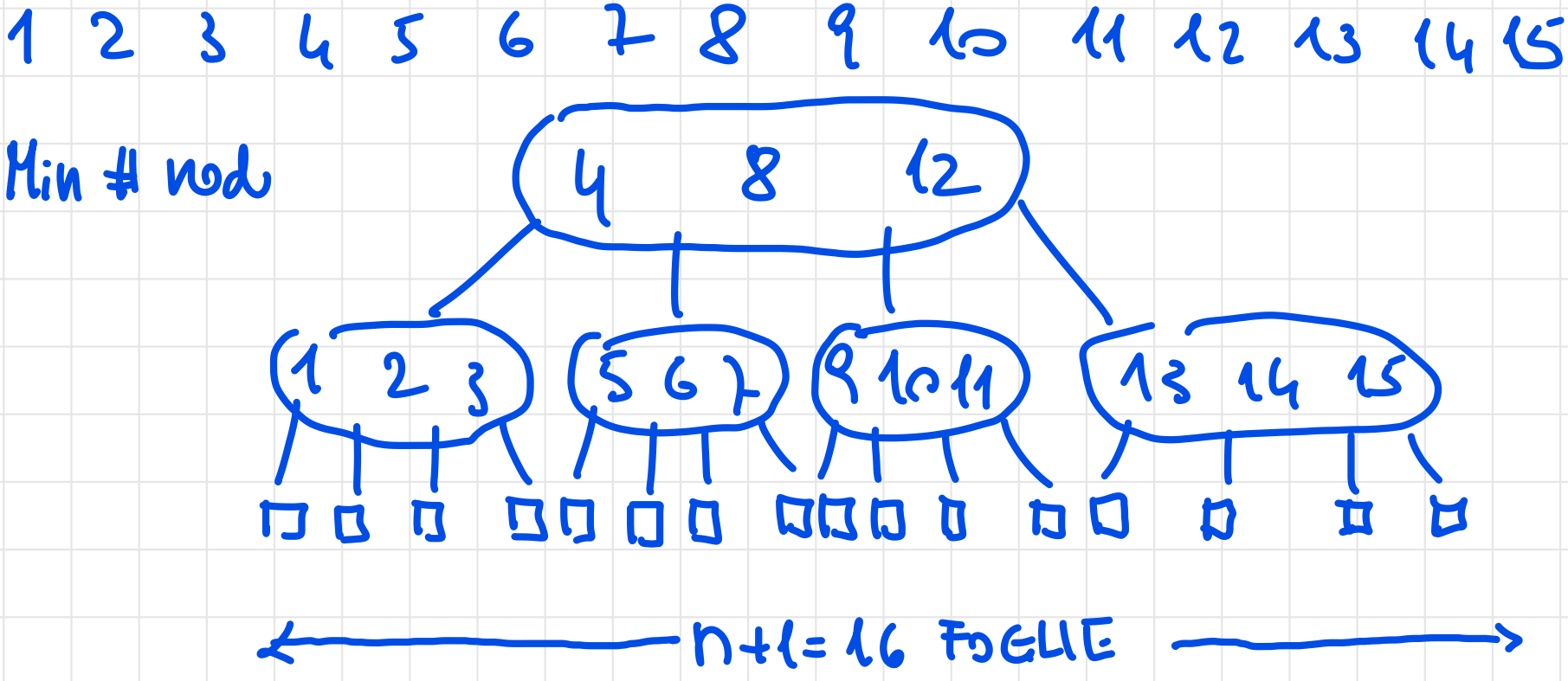
$$\Rightarrow \quad h \in \Theta(\log n)$$

Questo conclude la prova della Proposizione

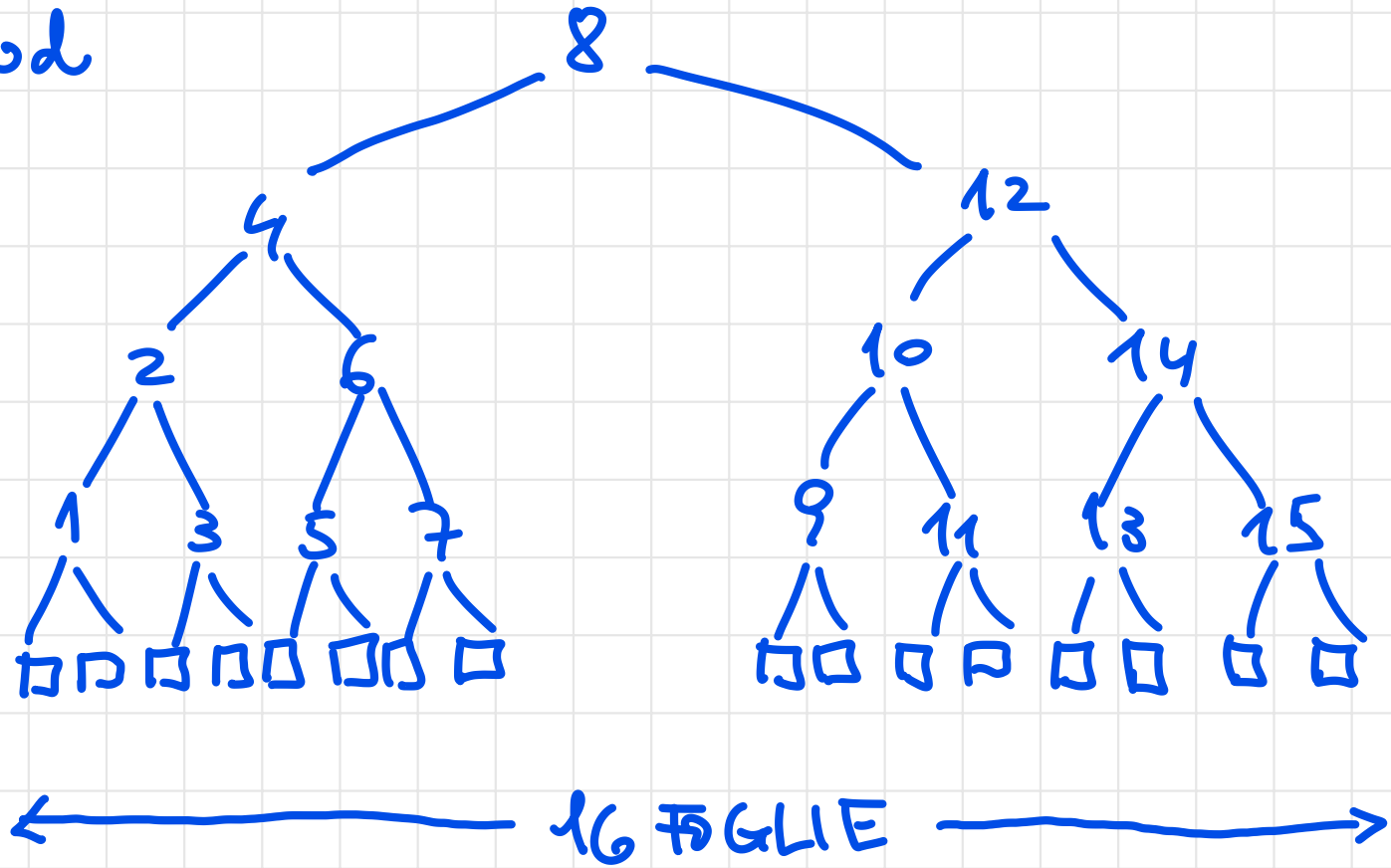
Il corollario segue immediatamente dalla analisi fatta per il MWS-Tree e dal fatto che per il (2,4)-Tree vale che $h \in \Theta(\log n)$ (appena provato) e $d_{max} \leq 4$ (dalla definizione)

Esercizio R-11.17 [GTG14]

Disegnare due (2,4)-Tree con 15 entry le cui chiavi sono gli interi da 1 a 15. Il numero di nodi deve essere minimo in un albero e massimo nell'altro.



Nex # nod



Implementazione dei metodi della Mappa: `put`

IDEA (`put`(k, x)):

- Se la chiave non è presente, inserisci la nuova entry $e = (k, x)$ in un nodo giusto per k ad altezza 1.
- Se il nodo in cui è stata inserita e va in overflow (ovvero ha 4 entry e 5 figli), invoca il metodo Split che ripristina le proprietà del (2,4)-Tree:
  - Sfruttando la flessibilità sul numero di entry ammissibili in un nodo.
  - Propagando, se necessario, l'overflow verso l'alto, che, se arriva alla radice, fa crescere di 1 l'altezza dell'albero.

put(k,x**)**

$w \leftarrow$ `MWTreeSearch(`k`,`T`.root());`
if (`!`T`.isInternal(`w`)`) **then**
    $e \leftarrow$ *entry in* w *con chiave* k;
    $y \leftarrow$ `e.getValue();`
    sostituisci x a y in e;
    return y;

`/* Caso in cui` w `è foglia */`
$e \leftarrow (k,x)$;
if (`!`T`.isRoot(`w`)`) **then** `expandExternal(`w`,`e`);`
else
    $u \leftarrow$ `!`T`.parent(`w`);`
    inserisci e *in* u *aggiungendo una foglia* w';
    if (`!`u è un 5-node`)` **then** `Split(`u`);` `/* overflow */`

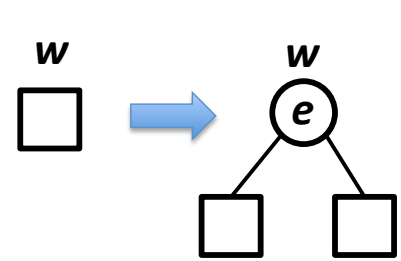
incrementa il numero di entry in T *di* 1;
return `null;`

$\Theta(1)$ operazioni

Descrizione grafica delle operazioni in rosso del lucido precedente:

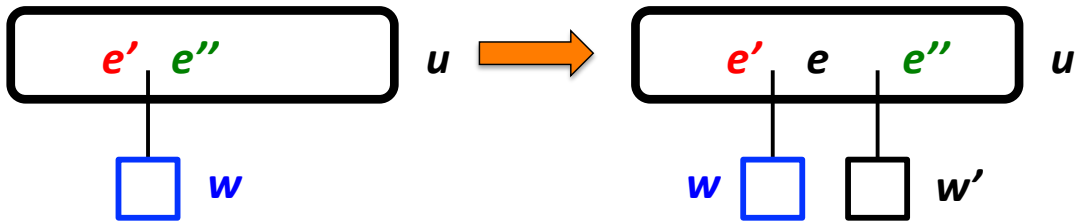
`expandExternal(`w, e`)` (caso w foglia e radice):

T prima delle put



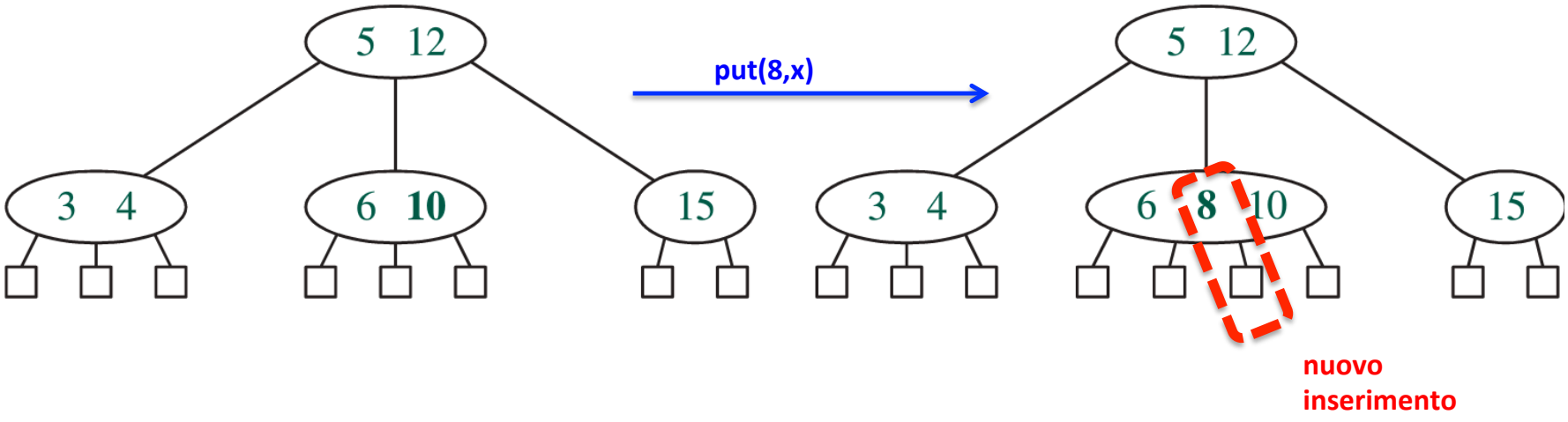
T dopo la put

inserisci e in u aggiungendo una foglia w′ (caso w foglia non radice):



N.B. e' oppure e'' potrebbe non esistere

Esempio (senza overflow)



Metodo Split (ricorsivo)

Input: 5-node $u \in T$, unica violazione delle proprietà di (2,4)-Tree
Output: Ripristino delle proprietà di (2,4)-Tree

Sia $u = (e_1, e_2, e_3, e_4)$ *con figli* u_1, u_2, u_3, u_4, u_5;
Crea due nuovi nodi
$u' = (e_1, e_2)$ con figli u_1, u_2, u_3
$u'' = (e_4)$ con figli u_4, u_5;
if (!T.isRoot(u)) **then**
  $v \leftarrow T$.parent(u);
  inserisci e_3 *in* v *con figlio sx* u' *e figlio dx* u'';
  cancella u;
  if (v è un 5-node) **then** Split(v);
else /* u è radice */
  crea una nuova radice contenente e_3 *e con 2 figli* u', u'';
  cancella u;

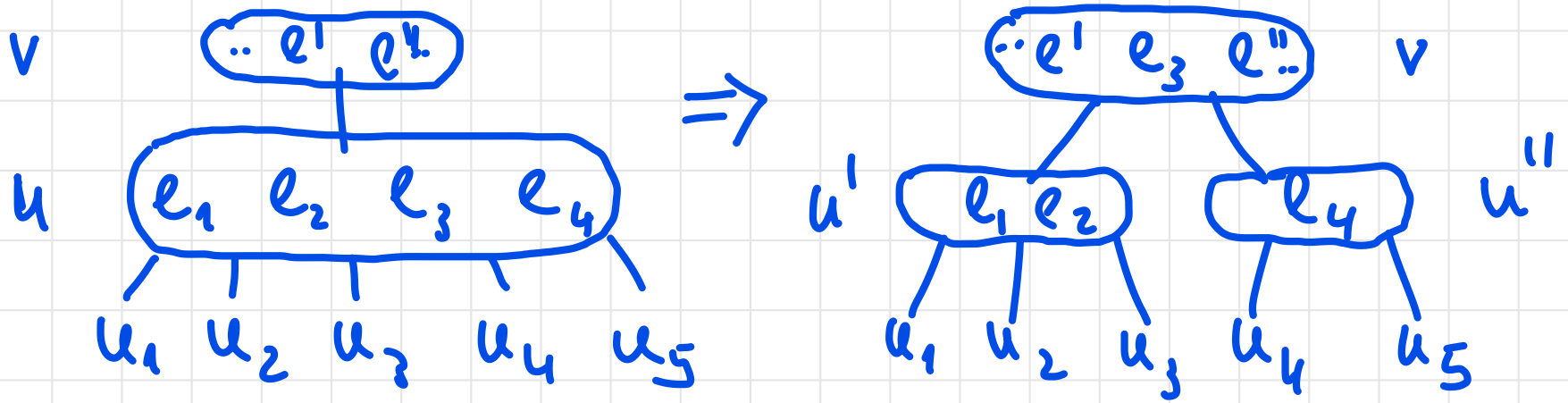
Oss. $\Theta(1)$ operazioni escluse le eventuali chiamate ricorsive

cresce di 1 l'altezza di T

Oss. Il metodo è descritto a parole in [GTG14]

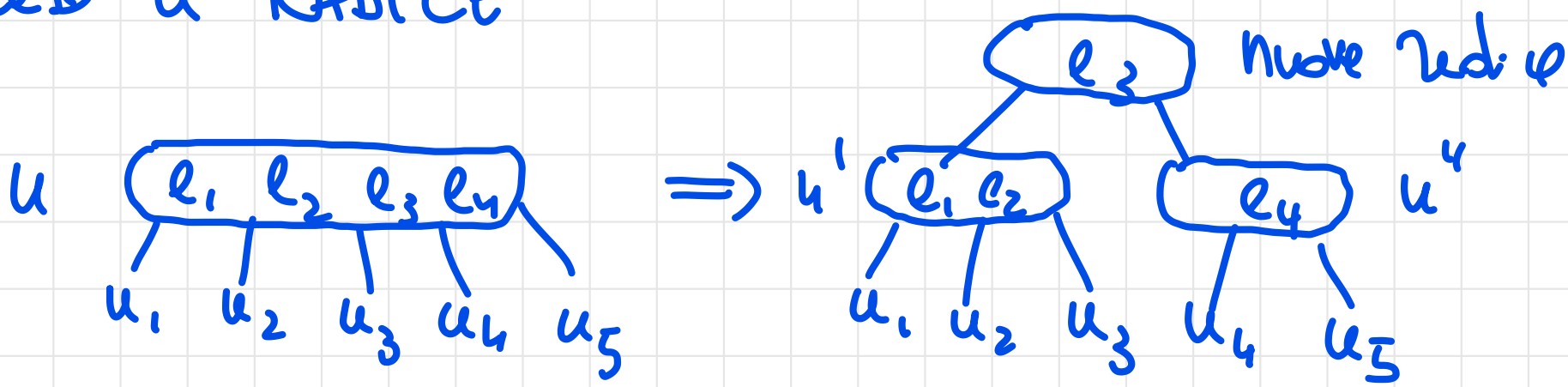
Descrizione grafica del metodo Split

Caso u non RADICE



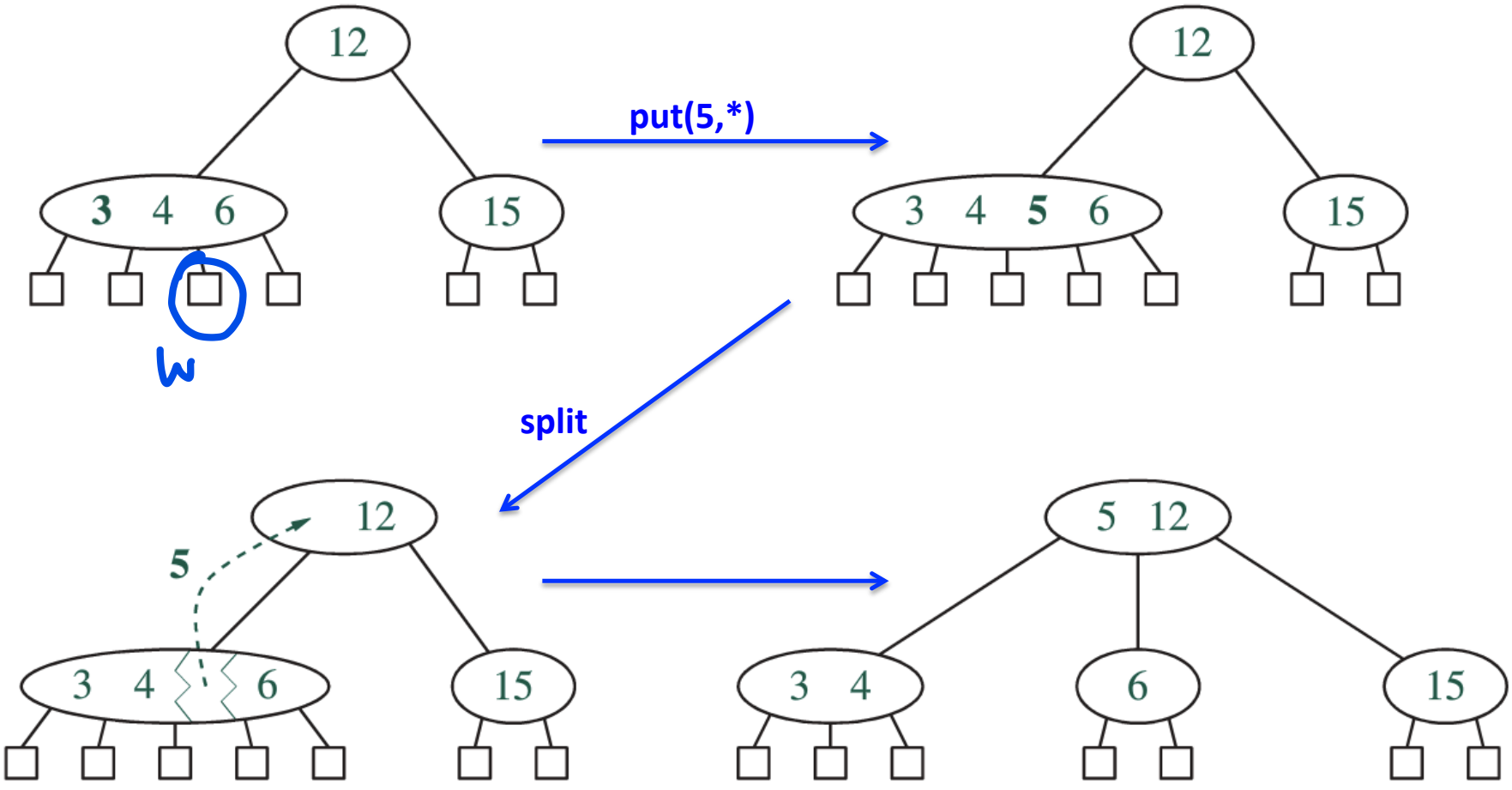
E dopo l'inserimento di e_3 in v, v va in overflow, invochiamo ricorsivamente split(v)

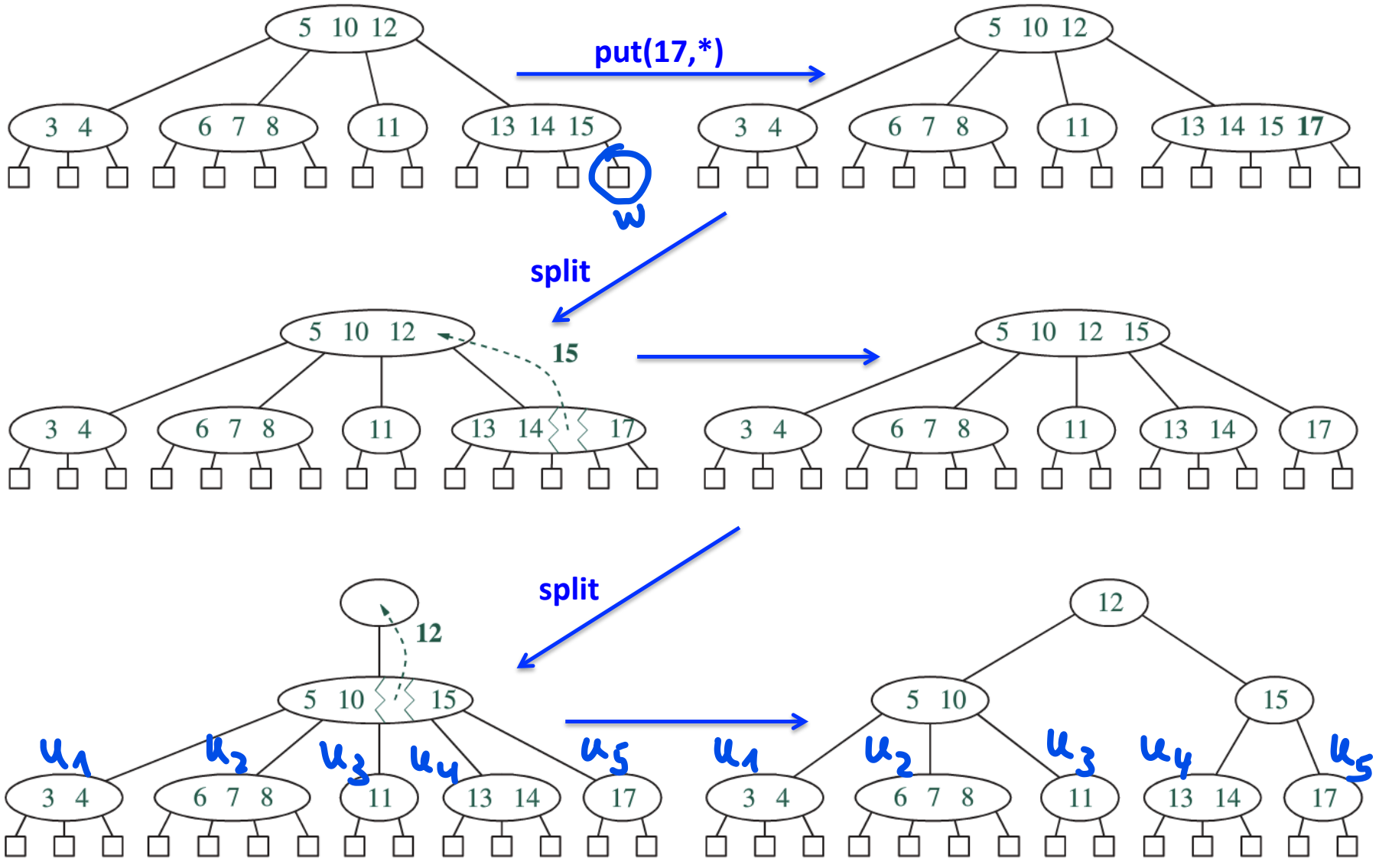
Caso u RADICE



In questo caso Split(u) termina la sua esecuzione

Esempio con overflow




5 10 12
3 4
6 7 8
11
13 14 15
put(17,*)
5 10 12
3 4
6 7 8
11
13 14 15 17
w
split
5 10 12
15
3 4
6 7 8
11
13 14
17
5 10 12 15
3 4
6 7 8
11
13 14
17
split
12
5 10 15
3 4
6 7 8
11
13 14
17
12
5 10
15
3 4
6 7 8
11
13 14
17

Complessità di put

Proposizione

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

Complessità è dominata da MW TreeSearch e da Split

* MW TreeSearch : $\Theta(\log n)$ operazioni
* Split : è un algoritmo ricorsivo
  - prima invocazione su un nodo ad altezza 1
  - successive invocazioni, una per livello

fino ad arrivare eventualmente alla radice, eseguendo $\Theta(1)$ operazioni per invocazione

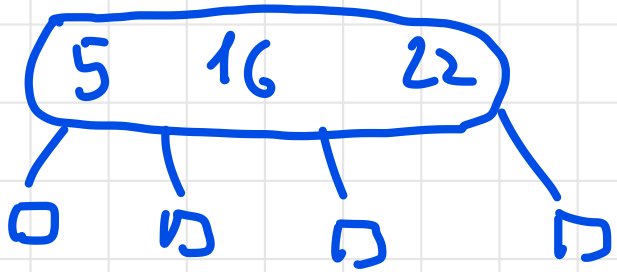
$\Rightarrow$ totale per la Splay: $\Theta(\log n)$ operazioni

$\Rightarrow$ Complessità finale: $\Theta(\log n)$

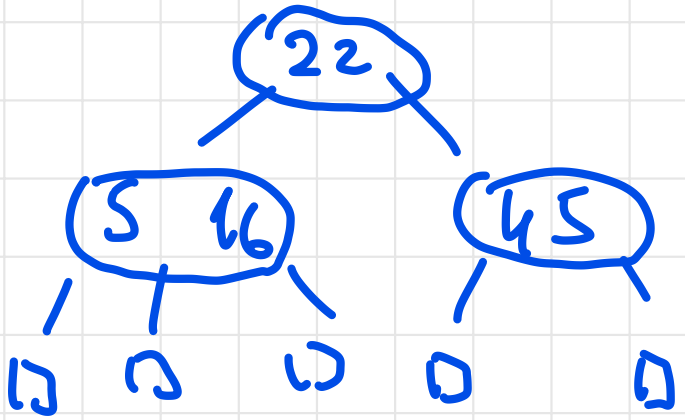
Esercizio (R-11.18.a [GTG14])

Inserire in un (2,4)-Tree inizialmente vuoto entry con chiavi $5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1, 6, 60$, nell'ordine dato.

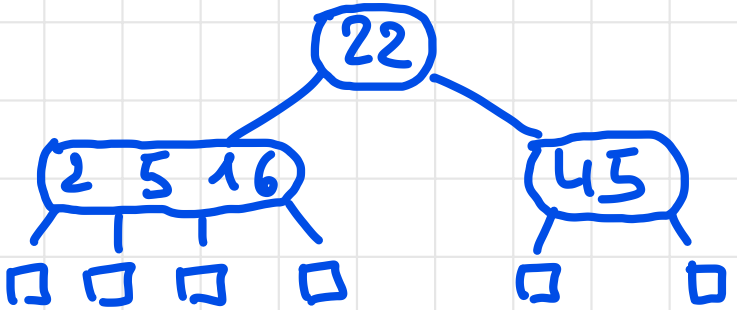
5 16 22 ⇒



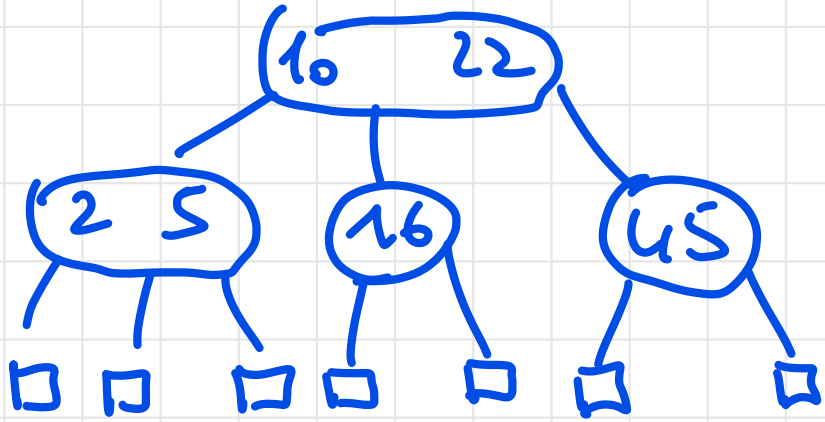
45 ⇒



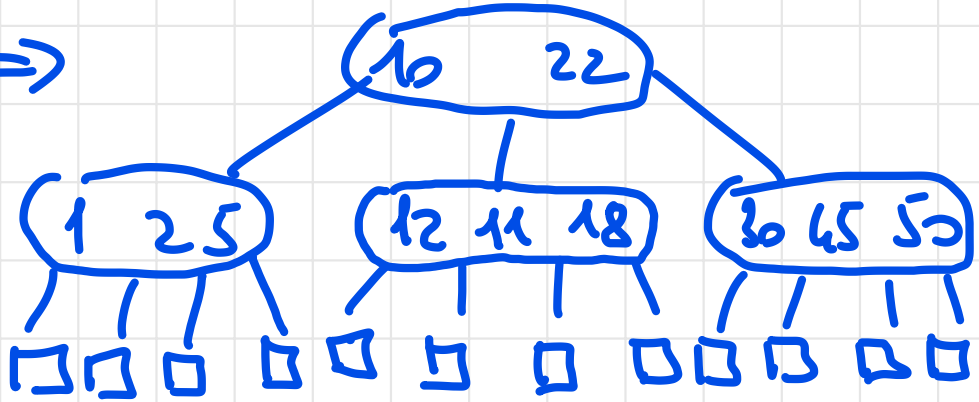
2 ⇒



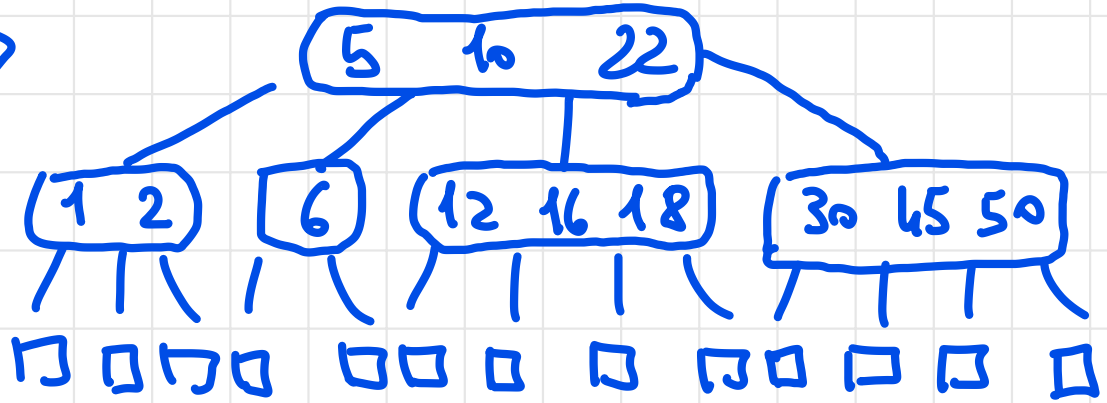
10 ⇒



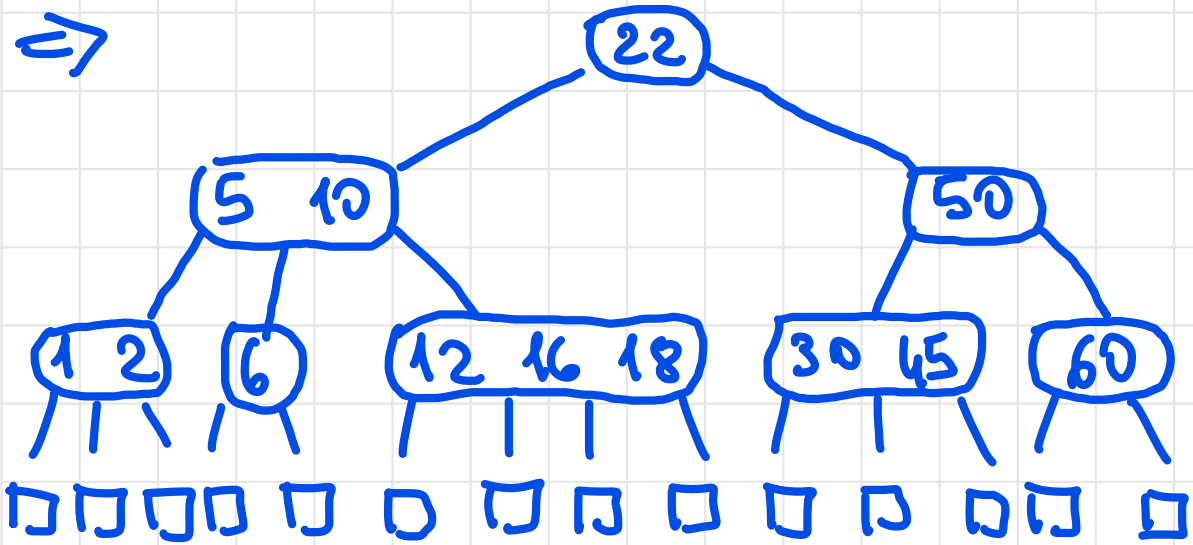
18 30 50 12 1 ⇒



6 ⇒



60



Implementazione dei metodi della Mappa: `remove`(k)

IDEA:

- Si rimuovono solo entry in nodi ad altezza 1.
- Se la entry e da rimuovere è in un nodo ad altezza > 1, sostituiscila con una entry e' ad altezza 1 e rimuovi quella.
- Se il nodo ad altezza 1 da cui è stata rimossa una entry va in underflow (ovvero contiene 0 entry) ripristina le proprietà del (2,4)-Tree:
  - Sfruttando la flessibilità sul numero di entry per nodo.
  - Progagando, se necessario, l'underflow verso l'alto, che, se arriva alla radice e fa diminuire di 1 l'altezza dell'albero.

N.B. La rimozione di una entry da un nodo e la gestione dell'eventuale underflow sarà effettuata tramite il metodo Delete.

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 Delete(e,w);
    else
        v ← figlio di w a sx 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 numero di entry in T;
return y;
```

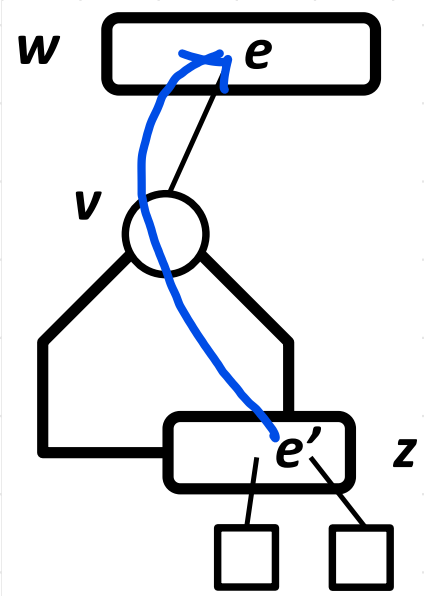
Esercizio: Scrivere uno pseudocodice più dettagliato per la ricerca di e'.

Osservazioni: caso altezza$(w) = 1$

$w =$ 

Si rimuove e e la foglia alla sua dx. Se w va in UNDERFLOW, cioè se e era l'unica entry in w, la Delete ripristina le proprietà del (2,4)-Tree

Osservazioni: caso altezza(w) > 1



Si sostituisce e' al posto di e in w e si rimuove e' da z insieme al suo figlio dx. Se dopo questa rimozione z va in UNDERFLOW, la Delete ripristina le proprietà dd (2,4)-Tree

Metodo Delete (ricorsivo)

Delete(u, e)

Input: $u \in T$ con entry e, e con un figlio foglia o vuoto a sx o dx 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;

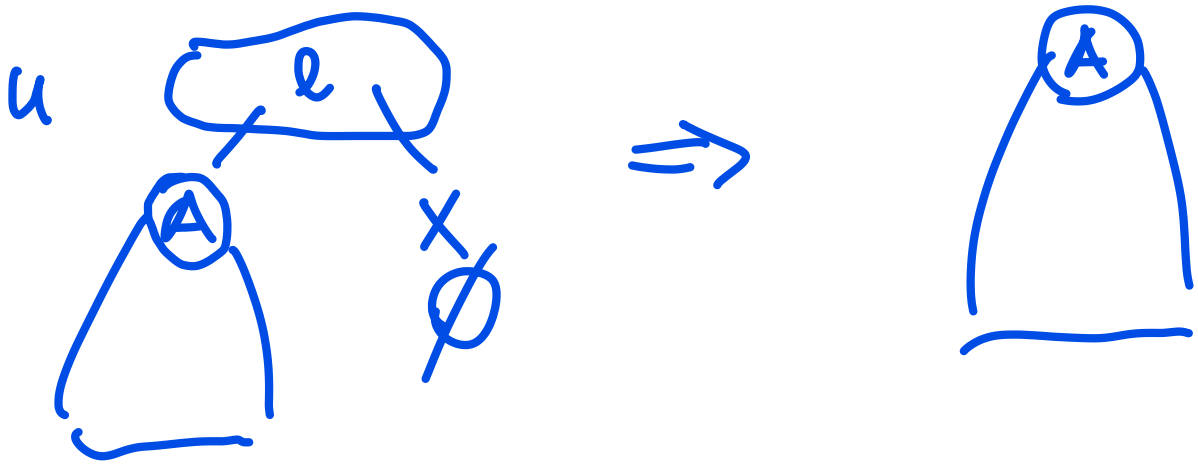
rimuovi e e X;
if (*u* `non ha più entry`) **then** `/* underflow */`
- CASO 1: *u radice*;
- CASI 2 e 3: *u con un fratello (sx o dx) d-node, $d = 3, 4$*;
- CASI 4 e 5: *u con solo fratelli 2-node*;

Le operazioni da fare nei 5 casi sono descritte nei lucidi seguenti

Oss. Il metodo è descritto a parole in [GTG14]

Caso 1: *u* radice

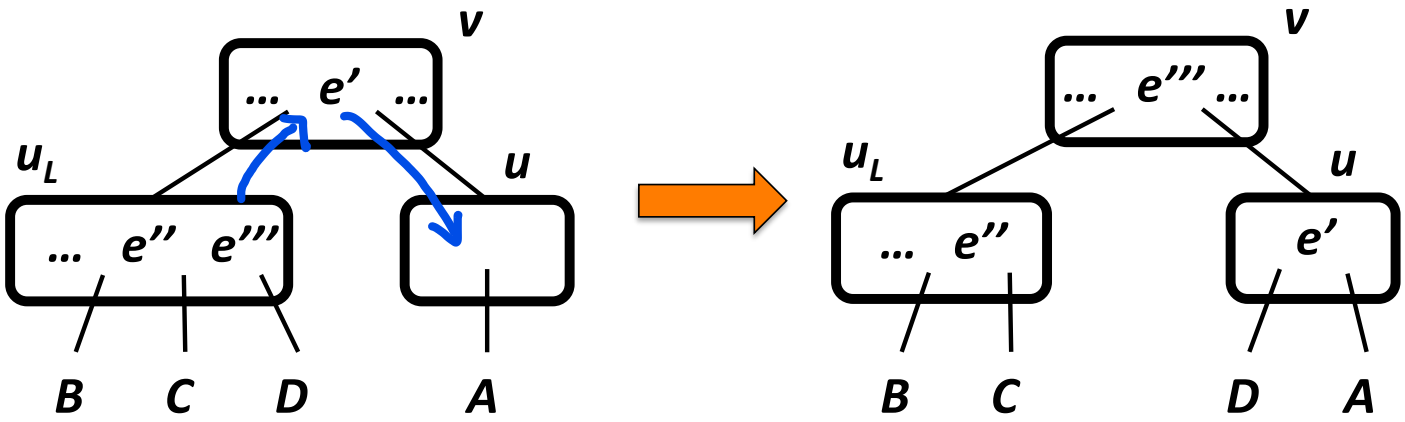
Operazione: *imposta A come nuova radice del (2,4)-Tree*;



Questo è il caso in cui l'altezza dell'albero diminuisce di 1

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

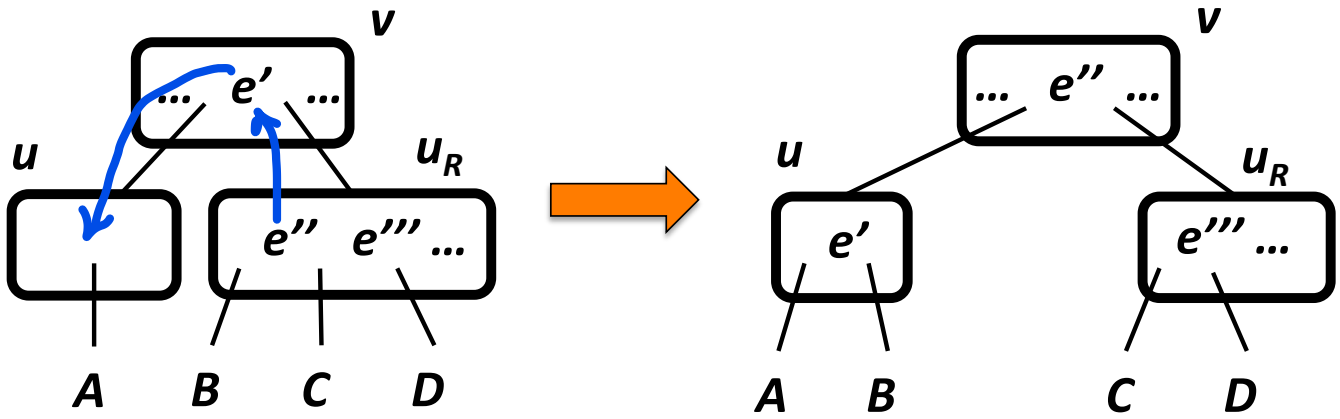
Operazione:



Con questa rotazione la delete termina la sua esecuzione

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

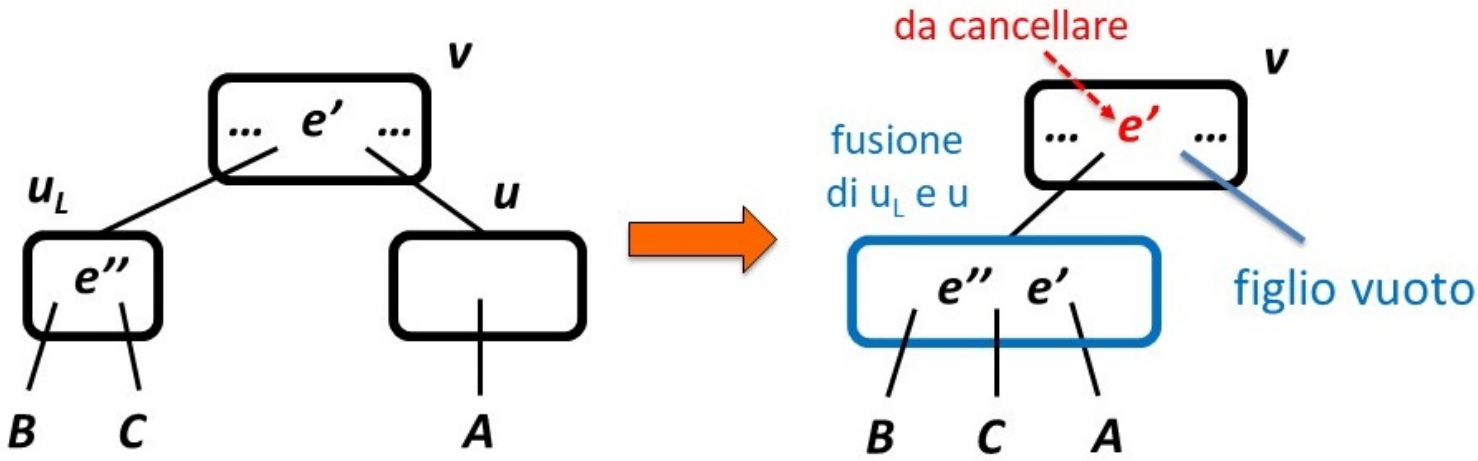
Operazione:



Dopo questa rotazione Delete termina la sua esecuzione

Caso 4: u ha un fratello u_L a sx che è un 2-node

Operazione:

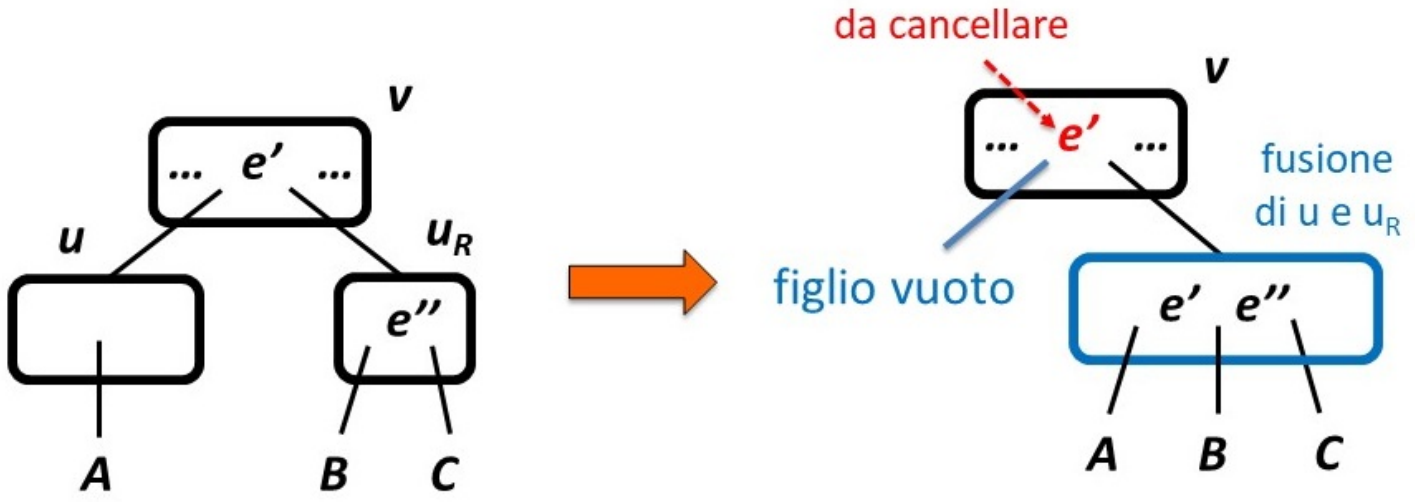



```
Delete(e',v);   /* propaga underflow verso l'alto */
```

nel caso in cui e' è l'unica entry in v,
e quindi la sua rimozione genera UNDERFLOW

Caso 5: u ha un fratello u_R a dx che è un 2-node

Operazione:



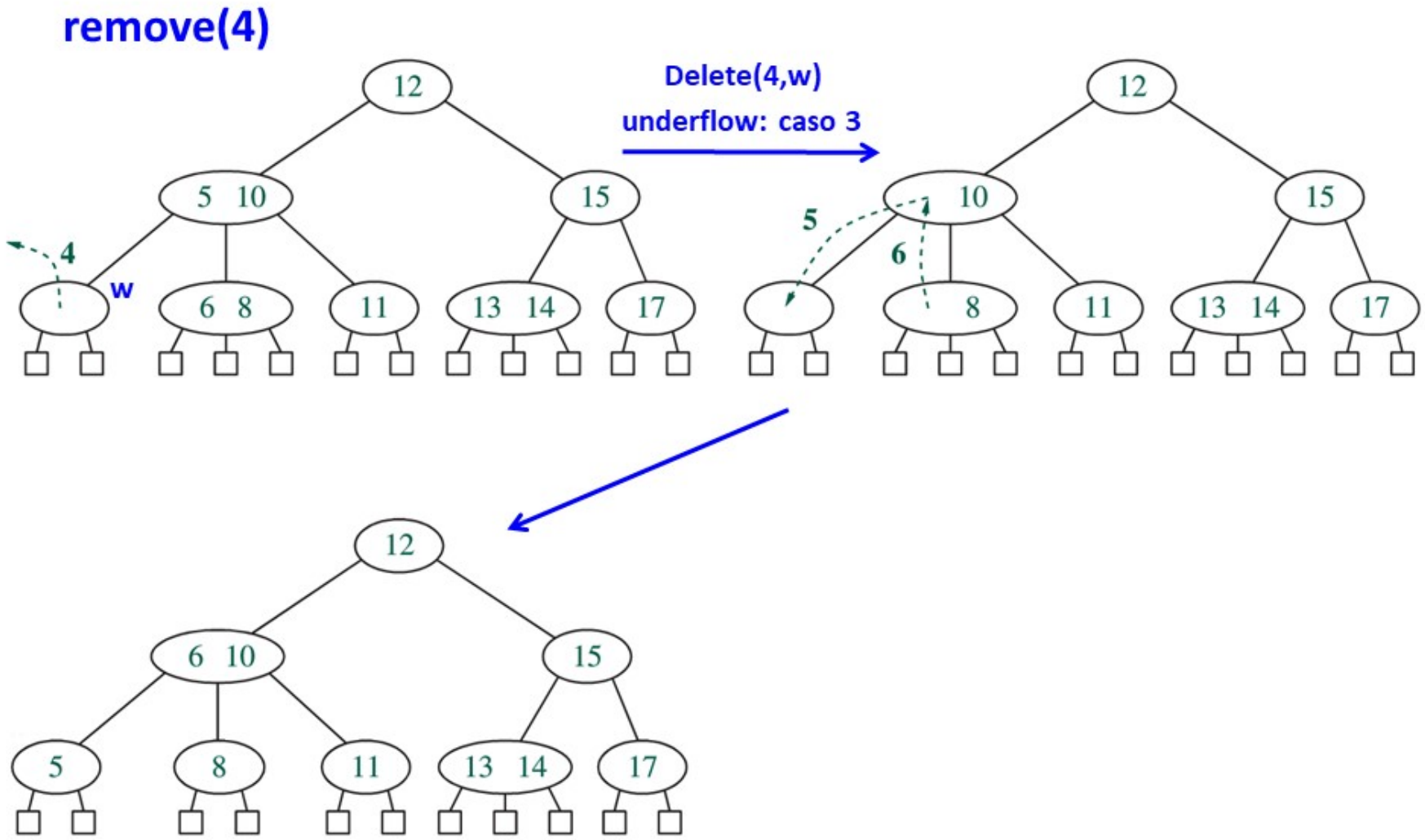

```
Delete(e',v);   /* propaga underflow verso l'alto */
```

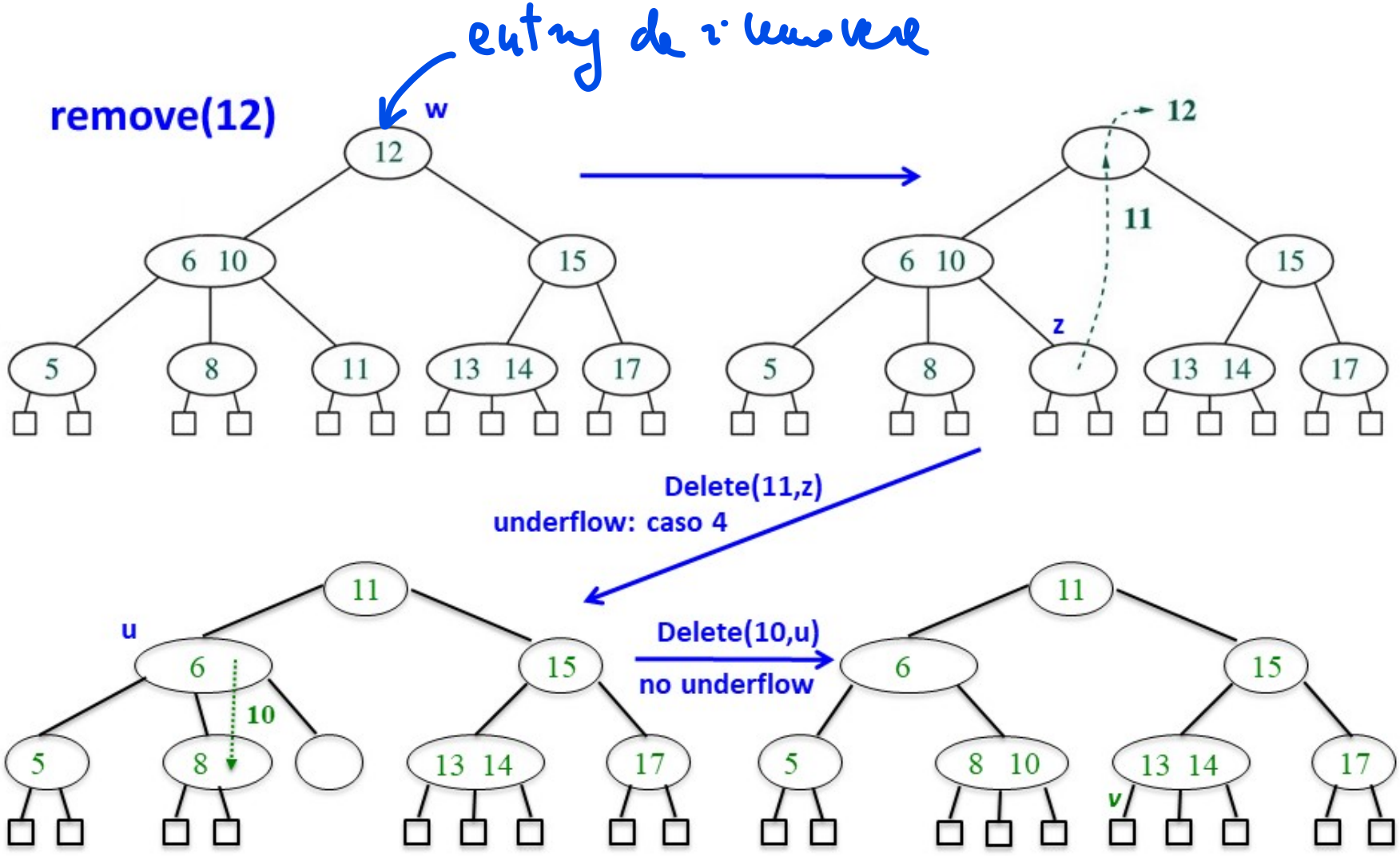
(Vedi commento per il CASO 4)

Osservazioni

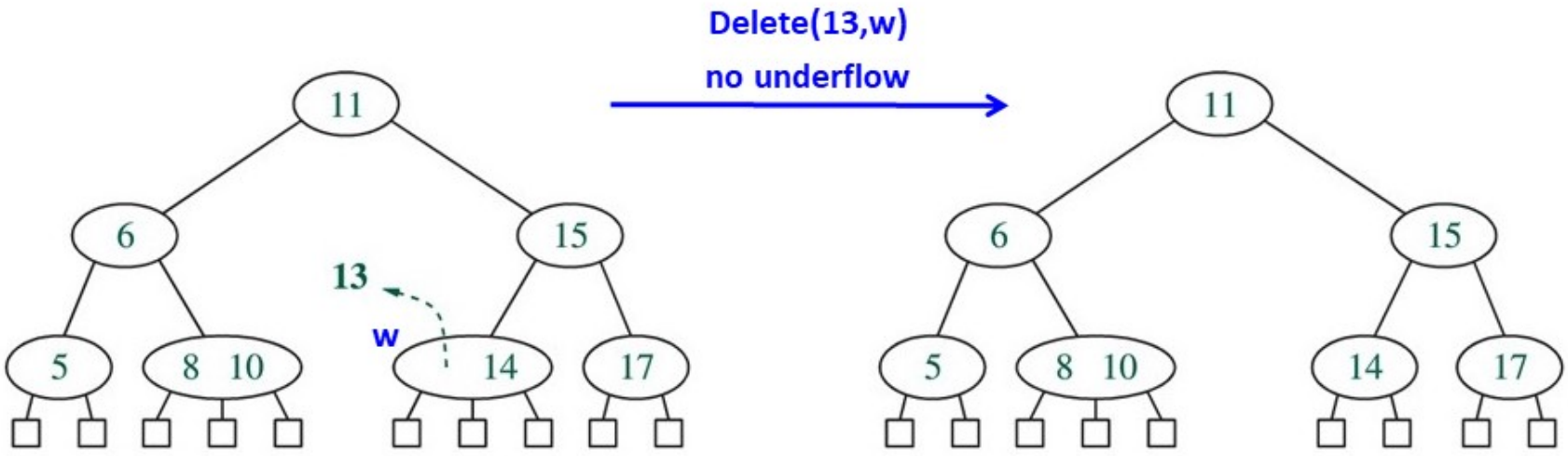
* I 5 CASI possono essere esaminati nell'ordine 1, 2, 3, 4, 5 fino a trovare il primo che può essere applicato (uno sicuramente sarà trovato perché i 5 CASI coprono tutti i possibili scenari.
* È facile vedere che in tutti e 5 i casi, le operazioni fatte mantengono valide le proprietà del (2,4)-Tree

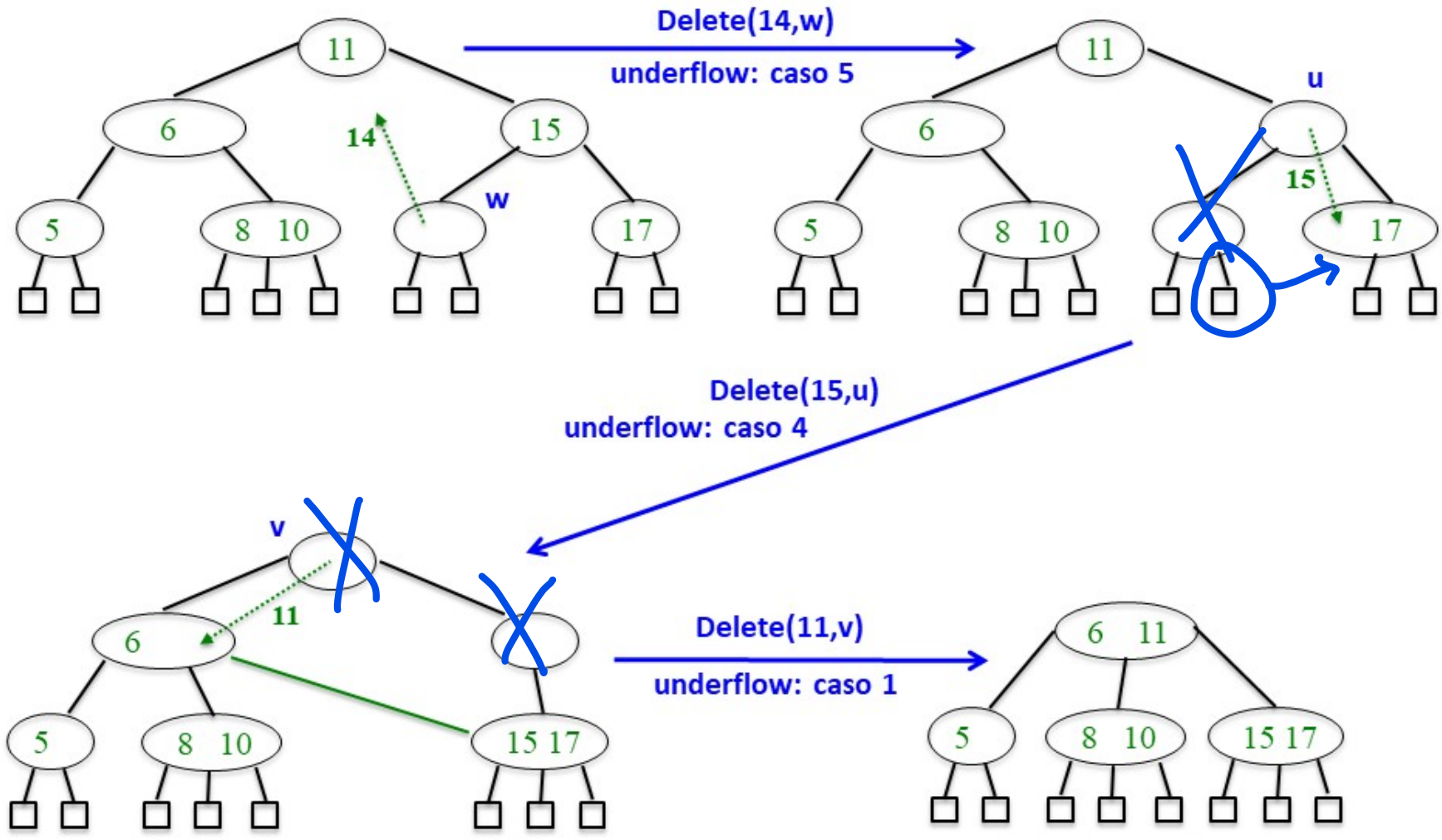
Esempi


remove(4)
Delete(4,w)
underflow: caso 3
12
5 10
15
4
w
6 8
11
13 14
17
12
10
15
5
6
8
11
13 14
17
12
6 10
15
5
8
11
13 14
17


entry da rimuovere
remove(12)
w
12
6 10
15
5
8
11
13 14
17
12
11
z
Delete(11,z)
underflow: caso 4
u
10
Delete(10,u)
no underflow
8 10
v

remove(13)




Delete(14,w)
underflow: caso 5
11
6
14
15
5
8 10
w
17
11
u
6
15
5
8 10
17
Delete(15,u)
underflow: caso 4
v
11
6
5
8 10
15 17
Delete(11,v)
underflow: caso 1
6 11
5
8 10
15 17

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' collocata in v che sostituisce al posto di e, se e è ad altezza > 1: $\Theta(\log n)$ operazioni.
* Delete: algoritmo ricorsivo

- $\leq h$ invocazioni ricorsive perché parte da un nodo ad altezza 1 e viene invocata lungo il percorso da questo nodo alla radice, al più una volta per livello
- Ogni invocazione ricorsiva richiede $\Theta(1)$ operazioni

$\Rightarrow$ Complessità di Delete è proporzionale all'altezza h del (2,4)-Tree che abbiamo dimostrato essere $\Theta(\log n)$

$\Rightarrow$ Complessità di remove(k) : $\Theta(\log n)$

□

Red-Black Tree

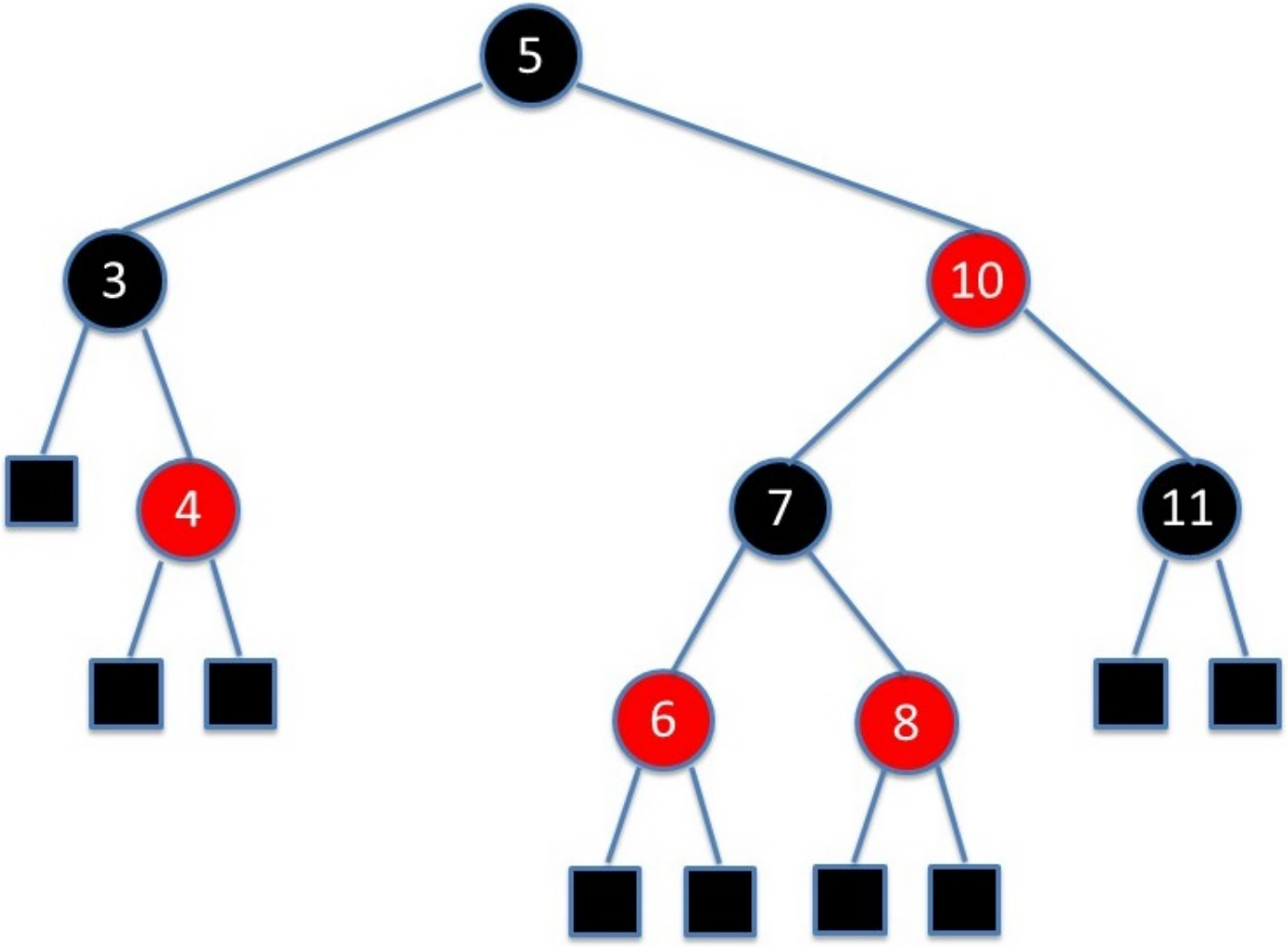
I Red-Black Tree sono Alberi Binari di Ricerca che, *a differenza del caso generale*, hanno altezza sempre logaritmica nel numero di nodi.

Definizione

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:

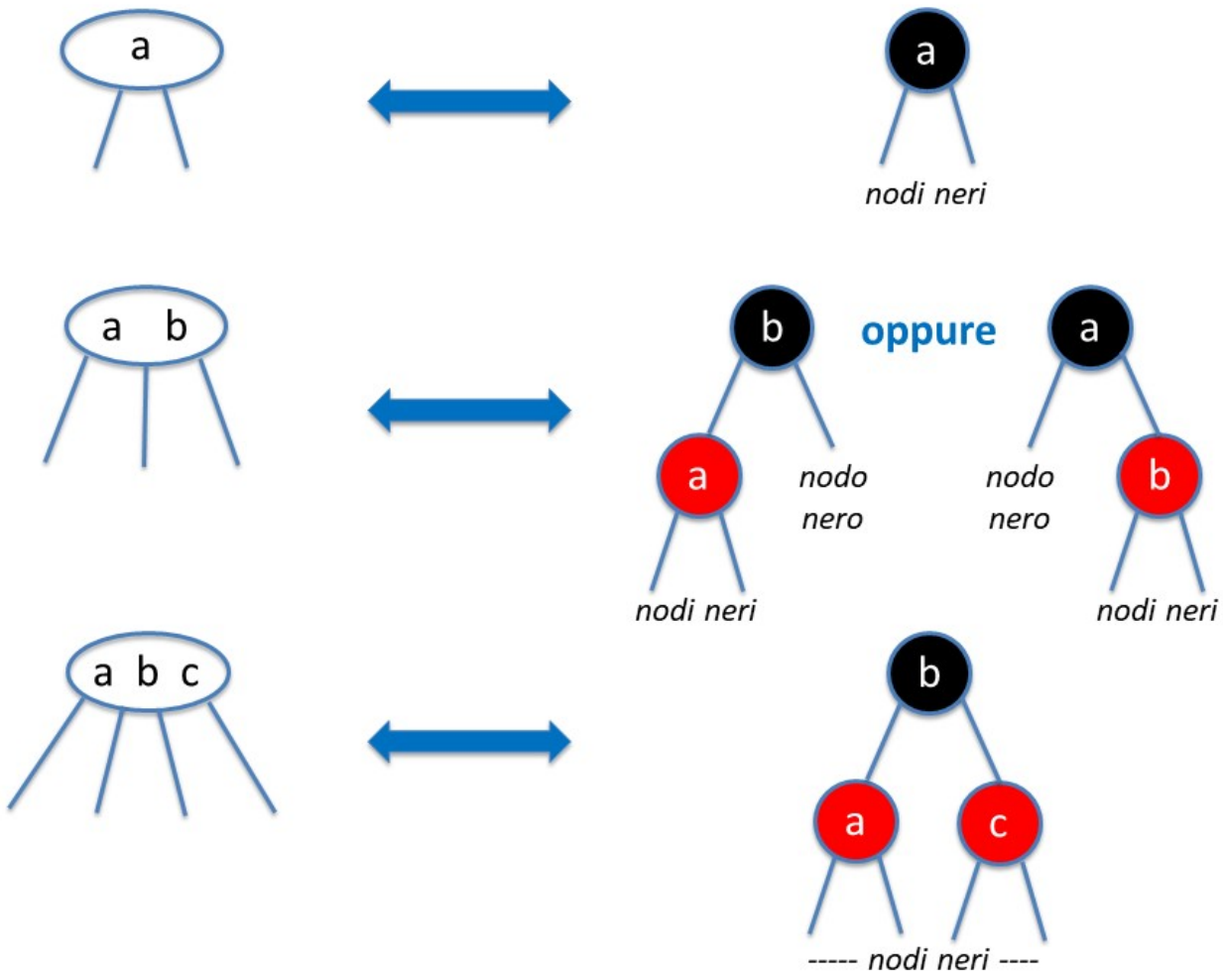
- La radice è nera (*Root Property*);
- Le foglie sono nere (*External Property*);
- I figli di un nodo rosso sono neri (*Red Property*);
- Tutte le foglie hanno la stessa *black depth*, ovvero lo stesso numero di antenati propri neri (*Depth Property*).

Esempio di Red-Black Tree

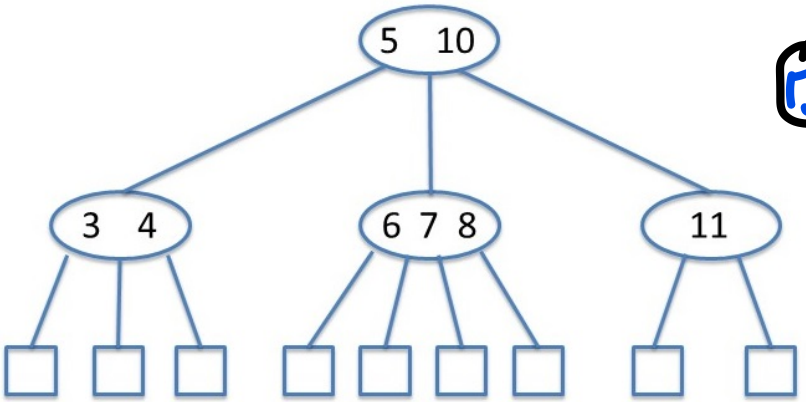


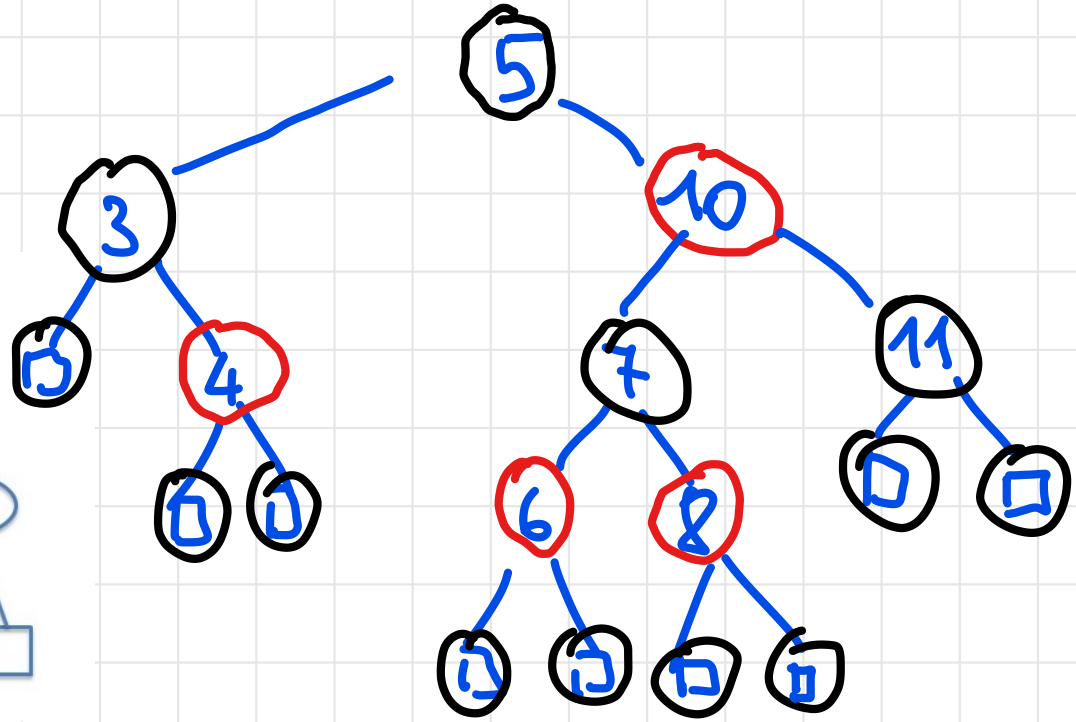
(2,4)-Tree e Red-Black Tree

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

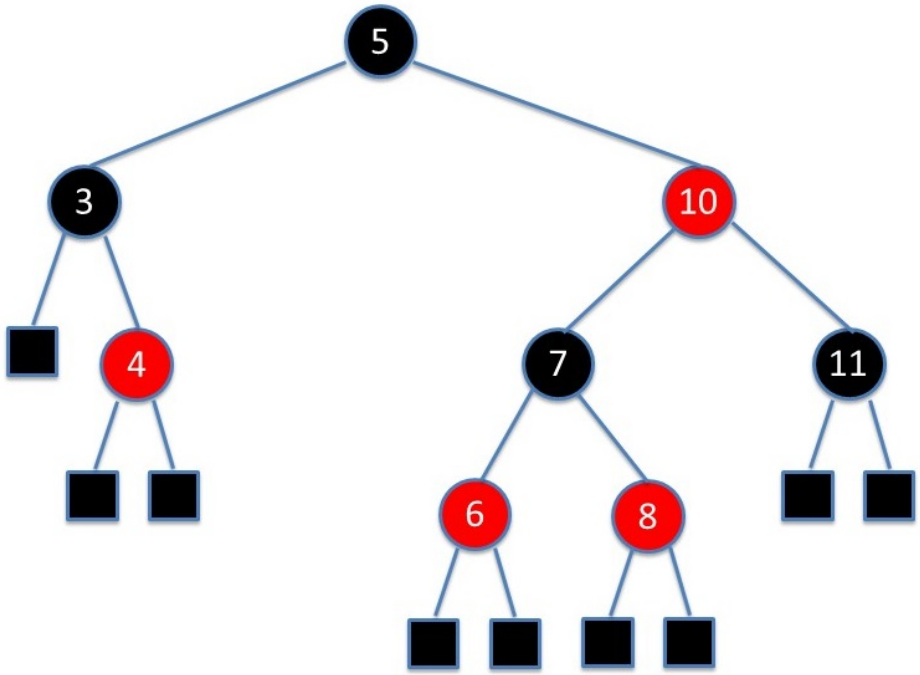


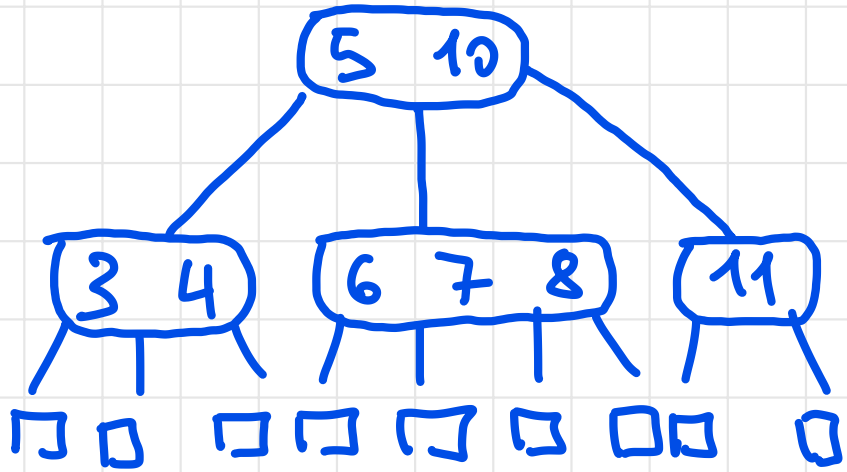
Esempio: (2,4)-Tree $\rightarrow$ Red-Black Tree





Esempio: Red-Black Tree $\rightarrow$ (2,4)-Tree





Metodi della Mappa su Red-Black Tree

Valgono le seguenti proprietà (le prove, benché semplici, non le studiamo):

- 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 la 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.

Multimappa

Definizione e metodi caratterizzanti

Definizione: una Multimappa è una Mappa che *ammette la presenza di* > 1 *entry con la stessa chiave*. Questa differenza richiede una modifica della specifica dei metodi caratterizzanti.

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 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, e **false**, altrimenti.

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

Implementazione

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 $L_k = \{v_1, v_2, \ldots, v_\ell\}$, allora (k, L_k) rappresenta, in modo compatto, le ℓ entry $(k, v_1), (k, v_2), \ldots, (k, v_\ell)$.

Implementazione dei metodi:

- 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.

Implementazione

- `remove`(k, v):
  - se esiste una entry (k, L_k), e $L_k = \{v\}$, si rimuove la entry (k, 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 copie della entry (k, v) da rimuovere, si può decidere di rimuoverne una sola o tutte.

Esempio di Multimappa: graduatoria

Struttura di una entry: (cognome,punteggio)

Collezione di entry:

(Bianchi,75),(Bianchi,73),(Bianchi,60),(Rossi,15),(Rossi,96),(Verdi,59)

Multimappa:

(Bianchi,75-73-60),(Rossi,15-96),(Verdi,59)

Esecuzione di `get(Bianchi)`, `put(Rossi,30)`, `remove(Bianchi,75)`

OPERAZIONE	OUTPUT
get (Bianchi)	75 - 73 - 60
put (Rossi, 30)	non restituisce alcun output ma (Rossi, 15-96) diventa (Rossi, 15-96-30)
remove (Bianchi, 75)	TRUE, e la entry (Bianchi, 75-73-60) diventa (Bianchi, 73-60)

Indici primari e secondari

Nelle Basi di Dati, l'accesso ai dati è reso efficiente dall'utilizzo di indici primari e secondari.

Indice Primario: (struttura di accesso a) una collezione di entry basata su una chiave che *non ammette duplicati*: ad es., numero di matricola degli studenti. Viene realizzato tramite una Mappa.

Indice Secondario: (struttura di accesso a) una collezione di entry basata su una chiave *ammette duplicati*: ad es., cognome degli studenti. Viene realizzato tramite una Multimappa.

Complessità dei metodi della Multimappa

Ipotesi: L_k implementata tramite lista

Definiamo

n = numero di CHIAVI DISTINTE presenti nella Mult. Mappa (⇒ le entry potrebbero essere molte di più)

$s = \max_k |L_k|$ il massimo è su tutte le chiavi k presenti nella Mult. Mappa

* get(k): Stessa complessità della Mappa, usando però la nuova definizione di n
* put(k,v): stessa complessità della Mappa, usando però la nuova definizione di n, e assume solo che v sia inserito in testa o in coda a L_k
* remove(k,v): alla complessità di remove(k) della Mappa (con la nuova definizione di n) si aggiunge un termine additivo $\Theta(s)$ per la ricerca di v in L_k

Riepilogo complessità per la Mappa

Si consideri una Mappa con n entry:

Metodo	Tabella Hash	ABR	(2,4)-Tree/RB-Tree
`get(`k`)`	$\Theta(1+\lambda)$	$\Theta(h)$	$\Theta(\log n)$
`put(`k, v`)`	$\Theta(1+\lambda)$	$\Theta(h)$	$\Theta(\log n)$
`remove(`k`)`	$\Theta(1+\lambda)$	$\Theta(h)$	$\Theta(\log n)$

N.B.:

- Per l'ABR, h è **l'altezza dell'albero** che **può assumere valori compresi tra** $\Theta(\log n)$ **e** $\Theta(n)$.
- Le complessità per la Tabella Hash sono al caso medio e λ è il load factor.

Riepilogo complessità per la Multimappa

Si consideri una Multimappa con n chiavi distinte:

Metodo	Tabella Hash	ABR	(2,4)-Tree/RB-Tree
`get(`k`)`	$\Theta(1+\lambda)$	$\Theta(h)$	$\Theta(\log n)$
`put(`k, v`)`	$\Theta(1+\lambda)$	$\Theta(h)$	$\Theta(\log n)$
`remove(`k, v`)`	$\Theta(s+1+\lambda)$	$\Theta(s+h)$	$\Theta(s+\log n)$

N.B.:

- Per l'ABR, h è **l'altezza dell'albero** che **può assumere valori compresi tra** $\Theta(\log n)$ **e** $\Theta(n)$.
- Il termine s nelle complessità di `remove` indica il massimo numero di entry con la stessa chiave.
- Nelle complessità per la Tabella Hash, il termine $1+\lambda$ (dove λ load factor) si riferisce al tempo medio di ricerca della chiave, mentre, nel caso di `remove`, il termine s per la ricerca della entry è worst-case.

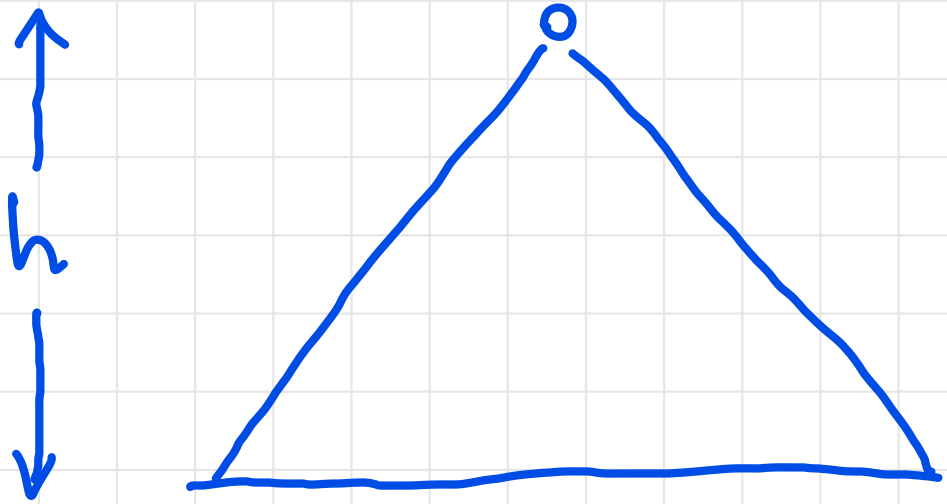
Esercizio

Progettare e analizzare un algoritmo iterativo efficiente che determini l'altezza di un (2,4)-Tree.

Algoritmo 24Height(T)

input: (2,4)-Tree T

output: Altezza di T



IDEA: Sfruttare il fatto che tutte le foglie sono alla stessa profondità, e scendere lungo un qualsiasi cammino radice-foglia tenendo traccia del numero di nodi incontrati.

```
h ← 0; v ← T.root();
while (T.isInternal(v)) do {
    v ← figlio più a sx di v
    h ← h + 1
}
return h
```

Correttezza: Locale (invariante: esercizio)

Complessità

* h_T iterazioni del while (h_T = altezza di T)
* $\Theta(1)$ operazioni per iterazione

$\Rightarrow$ Complessità $\in \Theta(h_T) = \Theta(\log n)$

con n = # entry in T

Esercizio

Sia T un (2,4)-Tree dove ogni nodo $v \in T$ memorizza in una variabile `v.size` il numero di entry in T_v (incluse quelle in v). Progettare un algoritmo ricorsivo `24CountLE` che conti quante entry in T hanno chiave $\leq k$, e analizzarne la complessità.

Esercizio

Analizzare l'algoritmo `24CountLE` assumendo che al posto della variabile `v.size` si abbia a disposizione un metodo `T.size(v)` che restituisce il numero di entry in T_v, con complessità lineare nel valore restituito.

Esercizio

Analizzare la complessità dell'algoritmo sviluppato per l'esercizio del Lucido 5 della Parte 1, scegliendo una opportuna implementazione per la Mappa.

Esercizio

Siano T e U due (2,4)-Tree di altezza h e tali che la massima chiave in T è *minore* della minima chiave in U.

1. Progettare un algoritmo di complessità $O(h)$ per fondere T e U in un unico (2,4)-Tree TU.
2. Dire quali valori può assumere l'altezza di TU, e se la radice contiene una o più entry

Esercizio

Siano T e U due (2,4)-Tree contenenti rispettivamente n ed m entry e tali che la massima chiave in T è *minore* della minima chiave in U. Progettare un algoritmo di complessità $O(\log n + \log m)$ per fondere T e U in un unico (2,4)-Tree TU.

Suggerimento: Se i due alberi hanno altezze diverse, fondere l'albero di altezza minore con un opportuno sottoalbero dell'altro di uguale altezza (utilizzando l'algoritmo sviluppato per l'esercizio precedente).

Riepilogo (Parti 1 e 2)

- Mappa: definizione come ADT
- Tabelle Hash:
  - Ingredienti principali
  - Funzione hash: hash code e compression function
  - Risoluzione delle collisioni tramite chaining
  - Implementazione dei metodi della Mappa e loro complessità al caso medio sotto l'ipotesi di uniform hashing
  - Load factor e rehashing
- Alberi Binari di Ricerca
  - Definizione
  - Metodo TreeSearch
  - Implementazione dei metodi della Mappa

- (2,4)-Tree
  - Definizione di Multi-Way Search (MWS) Tree
  - Relazione tra numero di entry e foglie in un MWS-Tree
  - Metodo MWTreeSearch
  - Definizione di (2,4)-Tree
  - Altezza di un (2,4)-Tree
  - Implementazione dei metodi della Mappa
- Cenni sui Red-Black Tree
- Implementazione di una Multimappa