Dati e Algoritmi 1: A. Pietracaprina

Mappa

In generale, una *Mappa* è una collezione di entry che supporta, come metodi principali: *ricerca*, *inserimento*, *rimozione*.

Gradi di libertà:

- **Universo delle chiavi non ordinato/ordinato**: si parla di Mappa quando non si assume che le chiavi provegnano da un universo, altrimenti si parla di Mappa ordinata
- **Chiavi distinte o replicate**: di solito si assume che le chiavi delle entry siano tutte distinte. Ci sono casi (ad es., indici secondari) in cui esse possono essere replicate. Si parla allora di Multimap o Dictionary (Dizionario).

Applicazioni

- **Database**. Ad es., su Uniweb: entry $\equiv$ studente
  - chiave: numero di matricola
  - valore: tutte le informazioni dello studente
- **Compilatori**. Ad es., per il type checking: entry $\equiv$ variabile
  - chiave: nome della variabile
  - valore: tipo
- **Motori di ricerca**. Ad es., entry $\equiv$ lista invertita
  - chiave: parola
  - valore: lista di documenti (ad es., pagine web)
- **Data analysis**. Ad es., per il conteggio delle frequenze di oggetti: entry $\equiv$ oggetto
  - chiave: oggetto
  - valore: numero di occorrenze

Mappa: definizione e interfaccia

Definizione

Mappa: collezione di entry con chiavi distinte provenienti da un universo U su cui è definito l'operatore "$=$", che supporta i metodi `get`, `put`, e `remove` specificati sotto (*concetto analogo di indice*).

```
public interface Map<K,V>{
  int size();
  boolean isEmpty();
  V get (K key);
  V put (K key, V value);
  V remove (K key);
  Iterable<K> keySet();
  Iterable<V> values();
  Iterable<Entry<K,V>> entrySet();
}
```

N.B. Il dominio non è necessariamente ordinato.

- `get(K key)`: se $\exists$ `(key,x)` restituisce `x` altrimenti restituisce `null`
- `put(K key, V value)`: se $\exists$ `(key,x)` mette `value` al posto di `x` e restituisce `x`, altrimenti inserisce l'entry `(key, value)` e restituisce `null`
- `remove(K key)`: se $\exists$ `(key,x)` rimuove la entry e restituisce `x`, altrimenti restituisce `null`
- `keySet()`, `values()`, `entrySet()`: restituiscono strutture (Iterable) contenenti, rispettivamente, le chiavi, i valori e le entry della mappa che possono essere enumerate da iteratori (iterator).

N.B.: la *mappa* è vista come *associative array* nel senso che la chiave della entry è usata come un "indice" di accesso alla mappa.

Mappa in Java

In `java.util`: interface `Map<K,V>`
al suo interno è definita una interfaccia *statica* `Map.Entry<K,V>`

- l'interfaccia `Map.Entry` può essere usata ovunque l'interfaccia `Map` sia visibile
- dichiarare `Entry` all'interno di `Map` serve solo per comunicare all'utente che verrà usata solo nel contesto dell'uso di `Map`
- `Entry` è statica nel senso che è associata alla classe `Map` e non a singoli oggetti di quella classe. In realtà tutte le interfacce nested sono `static` per default.

Implementazioni della Mappa

Implementazioni semplici ma inefficienti:

- Lista (position-based): `get`, `put`, `remove` in tempo $\Theta(n)$ (al caso pessimo) dove n= numero di entry nella mappa; spazio $\Theta(n)$ (INEFFICIENTE in TEMPO).

  Osservazione: *la complessità è dovuta al fatto che tutti e tre i metodi richiedono la ricerca di una chiave (anche* `put`*, dato che se la chiave da inserire è presente bisogna solo modificarne il valore).*

- Array di taglia $|U|$: `get`, `put`, `remove` in tempo $\Theta(1)$ (al caso pessimo); spazio $\Theta(|U|)$. (INEFFICIENTE in SPAZIO)

Implementazioni più efficienti:

- Tabella Hash
- Strutture ad Albero: *Albero Binario di Ricerca*, *(2,4)-Tree*

Esempio: most frequent word

Problema computazionale: dato un documento D contenente N parole (non necessariamente distinte) determinare la parola più frequente in D (*Si veda anche* [GTG14, Par.10.1.2]).

Algoritmo `WordCount`(D)

Input: Documento D di N parole: $D[i]$, $0 \leq i < N$
Output: Parola più frequente in D

```
M ← Mappa vuota di entry (parola,conteggio);
maxCount ← 0;
for i ← 0 to N − 1 do
    count ← M.get(D[i]);
    if (count = null) then count ← 0;
    M.put(D[i],count+1);
    if (count+1 > maxCount) then
        maxWord ← D[i];
        maxCount ← maxCount+1
return maxWord
```

Esempio: most frequent word

Supponiamo che:

- $N = 10^8$, ma solo 10^4 parole distinte in D.
- Ogni parola ha al più 15 caratteri e viene vista come una stringa di lunghezza 15 su un alfabeto di 27 caratteri (26 + carattere vuoto).

Con l'implementazione tramite lista, per ciascuna delle N parole in D la ricerca nella lista toccherebbe in media 5000 parole, per un totale di $5 * 10^{11}$ confronti!

Con l'implementazione tramite array, l'array dovrebbe avere $27^{15} > 10^{21}$ celle!

$\Rightarrow$ *entrambe le implementazioni hanno performance inaccettabili*.

L'implementazione della mappa tramite tabella hash, di cui parliamo adesso, dà in questo caso prestazioni molto migliori e rappresenta un compromesso tra le due implementazioni semplici.

Tabella Hash

Tabella Hash

Una Tabella Hash è definita dai seguenti 3 ingredienti principali:

- **Funzione hash** h

$$h : U = \{\text{chiavi}\} \rightarrow [0, N-1]$$

più in particolare (convenzione Java):

$$h : k \xrightarrow{\text{hash code}} \mathbb{Z} \xrightarrow{\text{compression function}} [0, N-1]$$

- **Bucket Array** A di capacità N. $A[i]$ rappresenta l'i-esimo bucket al quale vengono associate tutte le entry $< k, v >$ tali che $h(k) = i$ $(0 \leq i < N)$
- **Metodo di risoluzione delle collisioni** che sono costituite da chiavi distinte associate allo stesso bucket dalla funzione hash.

Tabelle Hash (continua)

Idea: usare il bucket $A[i]$ per memorizzare tutte le entry $< k, v >$ con $h(k) = i$ gestendo opportunamente le collisioni.

La scelta della funzione hash (nelle sue 2 componenti) diventa cruciale. In particolare:

- h deve "assomigliare" il più possibile a un processo random (*uniform hashing*) che associa a ogni chiave in U un intero su $[0, N-1]$ tale che $\forall k \neq k' \in U$ e $\forall i, j \in [0, N-1]$:
  - $\Pr[h(k) = i] = \frac{1}{N}$
  - $\Pr[h(k) = i | h(k') = j] = \Pr[h(k) = i] = \frac{1}{N}$
- h deve essere veloce da calcolare

Metodo HashCode in Java

- metodo `hashCode()` della classe `Object` restituisce un `int` che dipende dalla rappresentazione in memoria dell'oggetto.

  Non assicura che l'hashCode di un oggetto rimanga inalterato in diverse esecuzioni di un programma o in diverse implementazioni di Java

  $\Rightarrow$ non è un mapping "puro" da U a `int`

  Il metodo `hashCode` può essere riscritto in vari modi a seconda del tipo delle chiavi, restituendo sempre un `int`

Hash code per chiavi numeriche

In Java: supponiamo $\mathbb{Z} \equiv$ `int`

- `byte`, `short`, `char`, `int` $\rightarrow$ `int` (*cast*)
- `float` k (32 bit) $\rightarrow$ `Float.floatToIntBits(`k`)`
- `long` k (64 bit) $\rightarrow$ (`int`) ((`k` $>>$ 32)+(`int` k))
  - "$k >> 32$": 32 bit *più* significativi
  - "(int) k": 32 bit *meno* significativi

  N.B. solo "`(int)` k" perde l'informazione di metà dei bit
- `double` k (64 bit) $\rightarrow$ `Double.doubleToLongBits(`k`)` $\rightarrow$ `int`
  - `Double.doubleToLongBits(`k`)`: di tipo `long`
  - Trasformazione a `int`: come per `long`

Hash code per stringhe di char

$$S \equiv s_0 s_1 \ldots s_{k-1}$$

- $h(S) = \sum_{i=0}^{k-1} s_i$

  Oss. Non è un buon hashcode: es., $h(stop) = h(spot) = h(tops)$

- **Polynomial hash code**

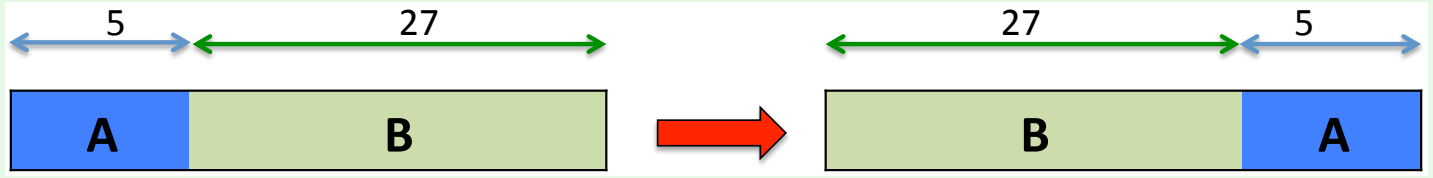
  $h(S) = \sum_{i=0}^{k-1} s_i \cdot a^{k-1-i}$

  Oss. Per parole inglesi vanno bene $a = 31, 33, 37, 39, 41$. La classe `String` in Java implementa il metodo hashCode in questo modo con $a = 31$

Hash code per stringhe di char (continua)

- **Cyclic Shift**: si sommano i singoli caratteri applicando dopo ogni addizione un cyclic shift alla somma parziale (shift di 5 posizioni va bene in pratica)

Cyclic shift di 5



$h \leftarrow s_0$ // *h a 32 bit*
for $i \leftarrow 1$ *to* $k-1$ **do**
  $h \leftarrow (h << 5) \mid (h >> 27)$;
  $h \leftarrow h + s_i$;
return h;

"<<": shift a sx di 5 posizioni, padding con 0;
">>": shift a dx di 27 posizioni, padding con 0;
"|": bit-wise or.

Compression Function

- **Division Method**
  Dato i= intero prodotto dall'hash code:

  $$i \to i \mod N$$

  dove N = capacità del Bucket Array

  Osservazione. Se N non è primo è più probabile che correlazioni negli hash code siano mantenute dopo l'applicazione di h.
  Ad esempio:
  - $N = 2^p \Rightarrow i \mod N$ equivale a prendere i p bit meno significativi di i, mentre è meglio che $i \mod N$ dipenda da tutti i bit di i
  - $N = 10^p \Rightarrow i \mod N$ equivale a prendere le p cifre meno significative di i in base 10. Come prima non dipende da tutta la rappresentazione di i

  In pratica: scegliere N *primo* distante da una potenza di 2

Compression Function (continua)

- **Multiply-Add-Divide (MAD) Method**

$$i \rightarrow [(ai + b) \mod p] \mod N$$

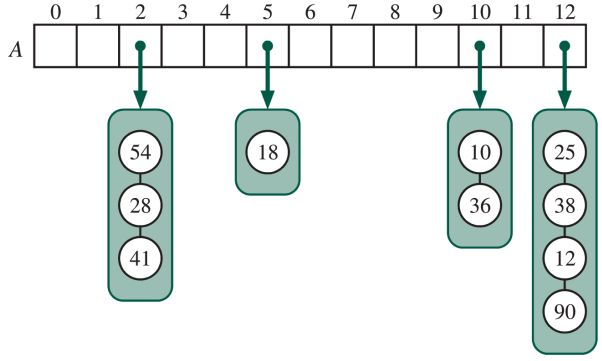
dove

- $p > N$ con p primo
- $a, b \in [0, p-1]$ scelti a caso, $a > 0$

Risoluzione delle collisioni

Collisione: $< k_1, v_1 >$, $< k_2, v_2 >$ con $k_1 \neq k_2$ e $h(k_1) = h(k_2)$

Separate Chaining: ogni bucket è visto come una `Map` più piccola implementata tramite lista



Definizione

Per una tabella hash con n entry si definisce il *load factor* λ come:

$$\lambda = \frac{n}{N},$$

ovvero la lunghezza media di un bucket

Implementazione dei metodi `get`, `put`, `remove`

- `get(`k`)`

  if (∃ una entry (k,x) in $A[h(k)]$) **then return** x;
  else return *null*;

- `put(`k`,`v`)`

  if (∃ una entry (k,x) in $A[h(k)]$) **then**
  - *sostituisci* x *con* v;
  - **return** x;

  else
  - *inserisci* (k,v) *in coda al bucket* $A[h(k)]$;
  - *incrementa di* 1 *la size della tabella*;
  - **return** *null*;

Implementazione dei metodi (continua)

- `remove(`k`)`

  if (∃ `una entry (`k`,`x`) in` $A[h(k)]$) **then**

  | *rimuovi* (k,x) *da* $A[h(k)]$;
  | *decrementa di* 1 *la size della tabella*;
  | **return** x;

  else return *null*;

Esempio ([GTG14, Es. R-10.5])

Inseriamo 11 entry con chiavi $12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5$, in una Tabella Hash, inizialmente vuota, con $N = 11$, $h(i) = (3i + 5) \mod 11$ e risoluzione delle collisioni tramite separate chaining. La tabella risultante ha $\lambda = 11/11 = 1$ ed è:

Bucket	Chiavi
0	13
1	94 39
2	
3	
4	
5	44 88 11
6	
7	
8	12 23
9	16 5
10	20

$$
\begin{aligned}
3 \cdot 12 + 5 &= 41 \to 8 \\
3 \cdot 44 + 5 &= 137 \to 5 \\
3 \cdot 13 + 5 &= 44 \to 0 \\
3 \cdot 88 + 5 &= 269 \to 5 \\
3 \cdot 23 + 5 &= 74 \to 8 \\
3 \cdot 94 + 5 &= 287 \to 1 \\
3 \cdot 11 + 5 &= 38 \to 5 \\
3 \cdot 39 + 5 &= 122 \to 1 \\
3 \cdot 20 + 5 &= 65 \to 10 \\
3 \cdot 16 + 5 &= 53 \to 9 \\
3 \cdot 5 + 5 &= 20 \to 9
\end{aligned}
$$

Complessità di `get`, `put` e `remove`

Studieremo:

- Complessità al caso pessimo, in funzione del numero di entry presenti nella tabella hash
- Complessità al caso medio (definito meglio dopo), in funzione del load factor λ della tabella hash.

Assumeremo sempre che il calcolo della funzione hash per un dato valore k della chiave richieda $O(1)$ operazioni.

Complessità al caso pessimo

Si consideri una tabella hash contenente n entry. La complessità al caso pessimo di `get`, `put`, `remove` è $\Theta(n)$, ed è dominata dalla ricerca della entry con chiave k, *necessaria* per tutti e tre i metodi. In particolare:

- $O(n)$: banale
- $\Omega(n)$: (istanza cattiva) tutte le entry nello stesso bucket e chiave k assente.

Esercizio (R-10.9 [GTG14])

Determinare la complessità nel caso migliore (best-case) e nel caso pessimo (worst-case) per l'inserimento di n entry in una Mappa implementata tramite Tabella Hash con separate chaining, inizialmente vuota. (Si assuma $N > n$)

Svolgimento

- best-case $O(n)$: no collisioni
- worst-case $\Theta(n^2)$: chiavi distinte, tutte le entry nello stesso bucket $\Rightarrow$ l'i-esimo inserimento richiede $\Theta(i)$, per $1 \leq i \leq n$.

Complessità al caso medio

Il seguente teorema dimostra che al caso medio le prestazioni della tabella hash sono molto buone, e questo è il motivo del suo successo e del largo impiego.

Teorema

Sotto l'ipotesi di hashing uniform, la *complessità al caso medio* di `get`, `put`, `remove` è $O(1+\lambda)$, sia nel caso di chiave presente che nel caso di chiave non presente.

La complessità al caso medio è intesa in questo senso:

- chiave non presente ($\Rightarrow$ *ricerca senza successo*): si media su tutti i possibili valori di $h(k)$ considerati equiprobabili
- chiave presente ($\Rightarrow$ *ricerca con successo*): si media su tutte le chiavi presenti considerate equiprobabili per la ricerca, e inserite una dopo l'altra senza cancellazioni

Osservazione. In pratica si impone $\lambda < 0.9$ (`java.util.HashMap` usa separate chaining con $\lambda = 0.75$ come default)

Rehashing

Si esegue quando λ sopra la soglia prefissata (ad es., $\lambda > 0.75$):

1. creazione di un nuovo bucket array di capacità $N' \geq 2N$.
2. scelta di una nuova funzione hash.
3. trasferimento delle delle entry dalla vecchia alla nuova tabella hash.

Osservazioni:

- Il trasferimento di n entry da una tabella all'altra può essere implementato in tempo $\Theta(n)$ al caso pessimo (non serve ricercare una entry prima di inserirla, si sa già che le chiavi sono distinte).
- Dato che un rehash si esegue quando λ supera una data costante, e la capacità del bucket array almeno raddoppia, il rehash di n entry è preceduto da $\Omega(n)$ `put` senza rehash $\Rightarrow$ il costo del rehash non domina rispetto a quello aggregato di questi `put`. Quindi, nella vita della tabella hash il costo dei rehash viene *ammortizzato* (ovvero nascosto) da quello delle altre operazioni.
- In un rehash può essere necessario cercare un numero primo $\geq 2N$, che potrebbe essere costoso. Noi assumiamo che il costo di questa operazione non domini rispetto a quello del trasferimento delle entry.

PROS & CONS delle Hash Table

PROS

- facile implementazione
- buone prestazioni (al caso medio)
- non richiede che le chiavi vengano da un universo ordinato

CONS

- complessità elevata al caso pessimo
- alea dovuta alla bontà della funzione hash
- spreco di spazio (per mantenere un basso load factor)

Altri usi importanti delle funzioni hash: firma digitale

Consideriamo la firma digitale di un documento m che A deve trasmettere via rete a B. Requisiti:

- Autenticazione: il destinatario deve poter verificare l'identità del mittente
- Non ripudio: al mittente non deve essere permesso di disconoscere un documento da lui firmato
- Integrità: deve essere possibile accertare la conformità del documento ricevuto con quello inviato
- Privacy: a un soggetto terzo non deve essere consentito di leggere il documento

Supponiamo che A e B abbiano: $K_{\text{priv}}(A), K_{\text{pub}}(A), K_{\text{priv}}(B), K_{\text{pub}}(B)$. Inoltre, sia h una funzione hash che mappa un messaggio m in un messaggio $h(m)$ molto più piccolo.

$$\begin{aligned} \text{Firma} \quad &: \quad m \longrightarrow \text{Cif}(m, K_{\text{pub}}(B)) + \text{Cif}(h(m), K_{\text{priv}}(A)) \\ \text{Verifica} \quad &: \quad \text{Dec}(\text{Cif}(m, K_{\text{pub}}(B)), K_{\text{priv}}(B)) \longrightarrow m \\ &\quad\;\; \text{Dec}(\text{Cif}(h(m), K_{\text{priv}}(A)), K_{\text{pub}}(A)) \longrightarrow h(m) \end{aligned}$$

Implementazione della Mappa tramite Alberi

- Assumiamo che le chiavi provengano da un universo ordinato
- Strutture basate su alberi:
  - Albero Binario di Ricerca
  - Multi-Way Search Tree
  - (2,4)-Tree

Alberi Binari di Ricerca

(Paragrafo 11.1 [GTG14])

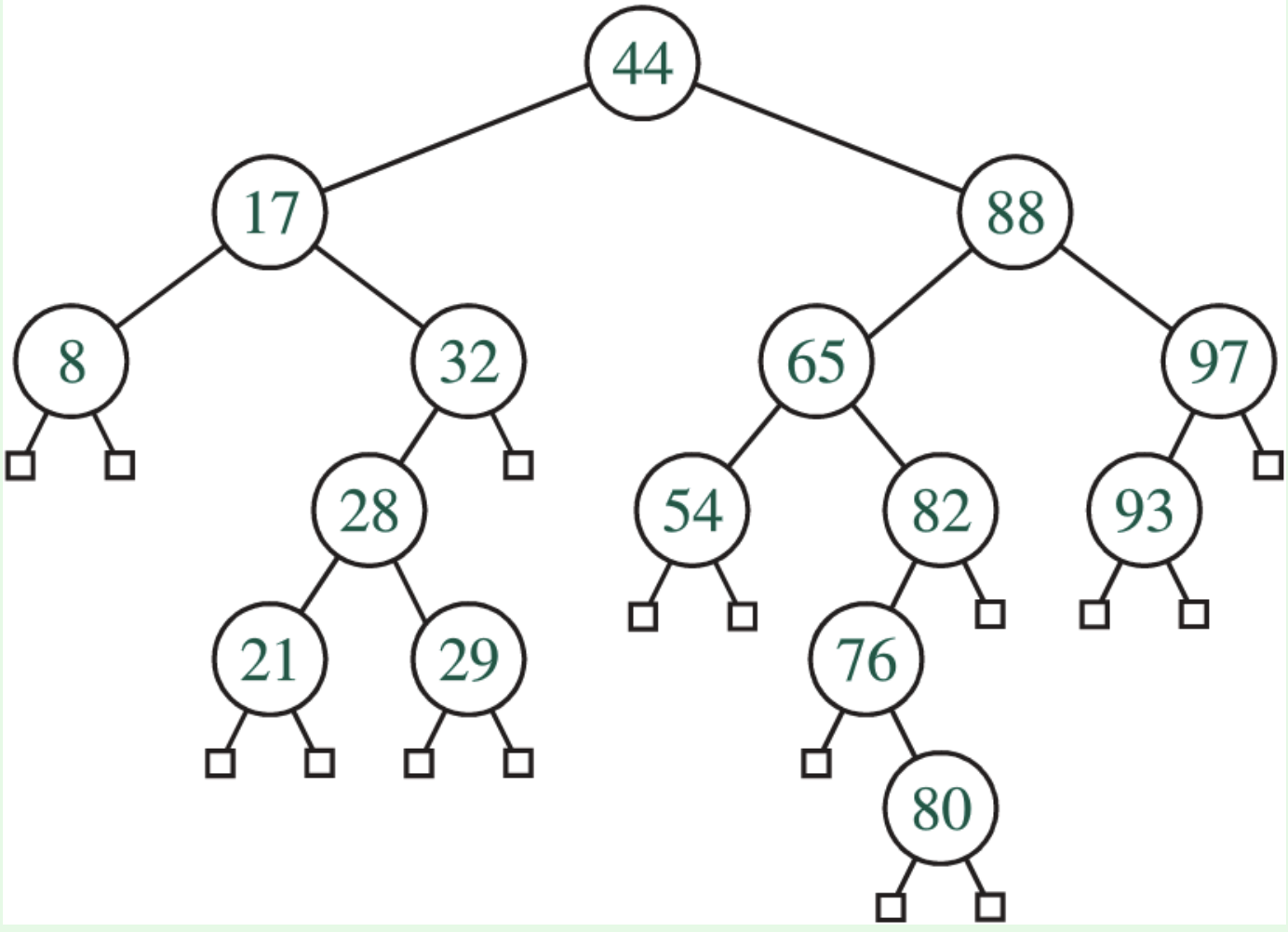
Alberio Binario di Ricerca

Definizione

Albero Binario di Ricerca: albero binario proprio i cui *nodi interni* memorizzano entry con chiavi provenienti da un universo ordinato, tale che per ogni nodo interno v la cui entry ha chiave k:

- le chiavi nel sottoalbero sx di v sono $< k$
- le chiavi nel sottoalbero dx di v sono $> k$

Esempio (solo chiavi)



N.B. La visita inorder tocca le entry in ordine non decrescente.

TreeSearch

Sia T un Albero Binario di Ricerca

Algoritmo `TreeSearch(`k`, `v`)`
Input: chiave k, nodo $v \in T$
Output: nodo di T_v con chiave k, o foglia in posizione "giusta" per k

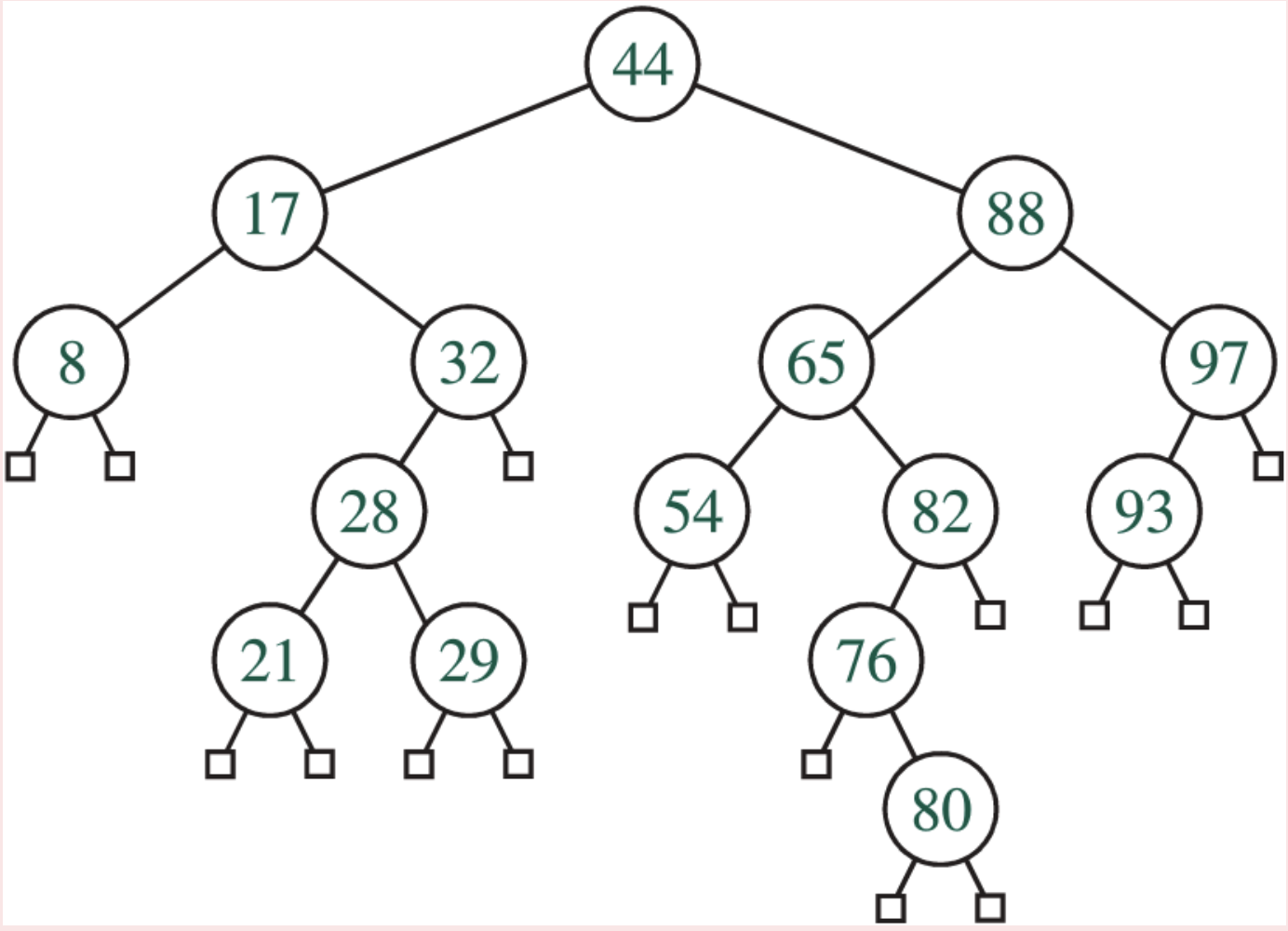
```
if (T.isExternal(v) OR (v.getElement().getKey()=k)) then
    return v;
if (k < v.getElement().getKey()) then
    return TreeSearch(k, T.left(v));
else
    return TreeSearch(k, T.right(v))
```

Osservazioni

- Per ricercare in tutto l'albero si parte con $v = T$`.root()`.
- Correttezza: banale

Esercizio

Nel seguente albero T visualizzare il percorso di `TreeSearch`$(k, T.\text{root}())$, per $k = 28, 82, 61, 3$



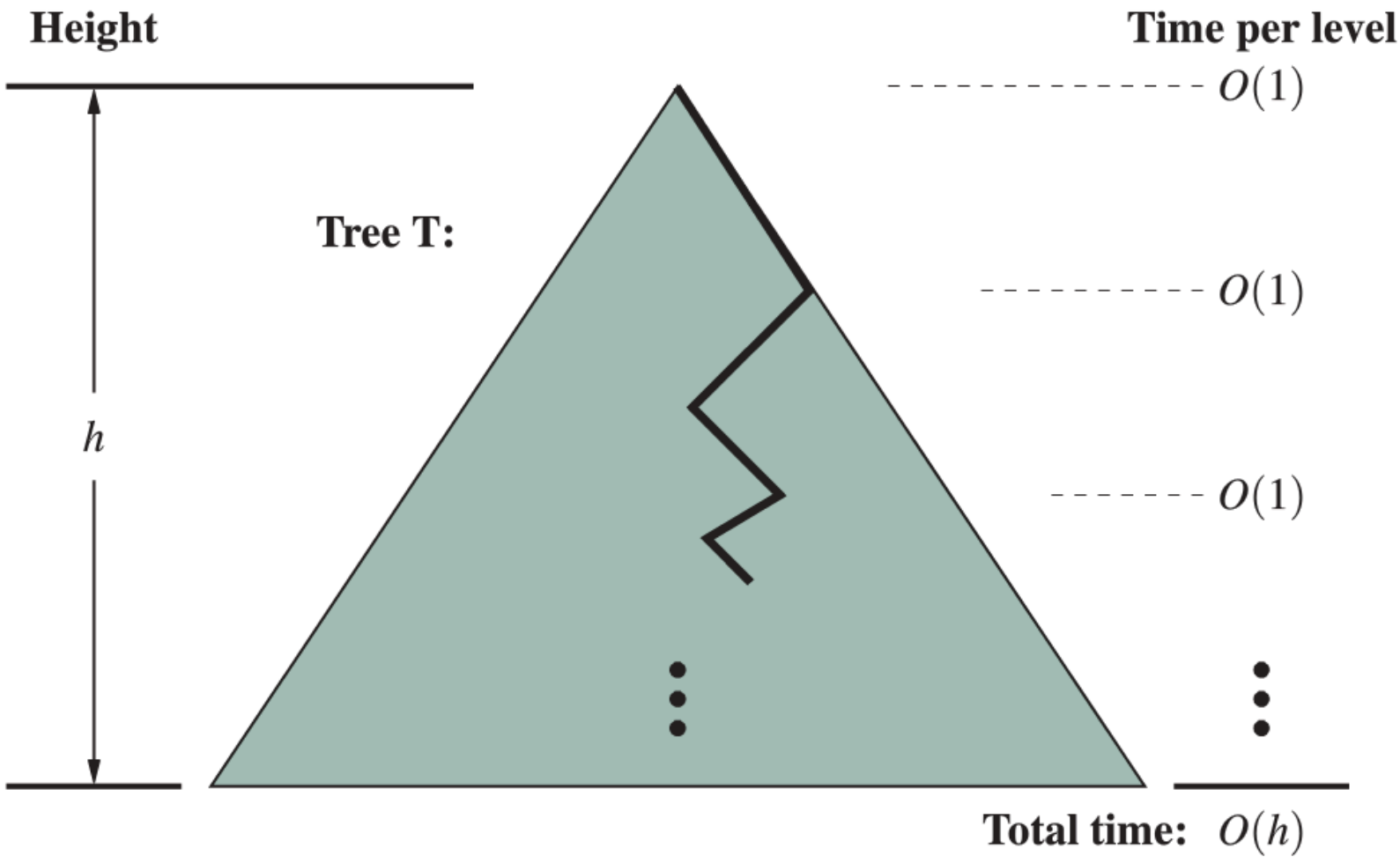
Complessità di `TreeSearch`

Sia h l'altezza di T e studiamo la complessità di `TreeSearch(k,T.root())`, tramite l'albero della ricorsione.

- L'albero della ricorsione ha $\leq h+1$ nodi, dato che `TreeSearch` è invocato sui nodi di un cammino di lunghezza $\leq h$ (un'invocazione per nodo).
- Ogni invocazione esegue $\Theta(1)$ operazioni, oltre a un'eventuale chiamata ricorsiva.
- Esistono istanze per cui `TreeSearch` scende effettivamente lungo un cammino di lunghezza $\Theta(h)$ (quando restituisce la foglia più profonda).

$\Rightarrow$ **complessità** $\Theta(h)$

N.B.: Con $\Theta(h)$ intendiamo $\Theta(h+1)$, che copre il caso $h=0$


Height
Time per level
$O(1)$
Tree T:
$O(1)$
h
$O(1)$
Total time: $O(h)$

Implementazione dei metodi della Mappa: get

- `get(`k`)`

  $w \leftarrow$ `TreeSearch(`k`,` T`.root());`
  if T`.isExternal(`w`)` **then return** *null*`;`
  else return w`.getElement().getValue();`

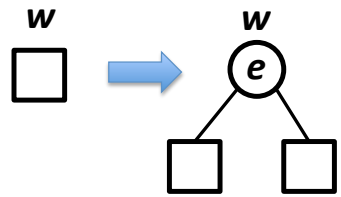
Complessità: $\Theta(h)$ dominata da `TreeSearch`

Implementazione dei metodi della Mappa: put

- `put(`k`,`x`)`

  $w \leftarrow$ `TreeSearch(`k`,` T`.root());`
  if T`.isInternal(`w`)` **then**
  $y \leftarrow w$.getElement().getValue();
  sostituisci x *a* y *nella entry* w.getElement();
  return y

  trasforma w *in nodo interno contenente* $e = (k, v)$ *con 2 figli foglia*;

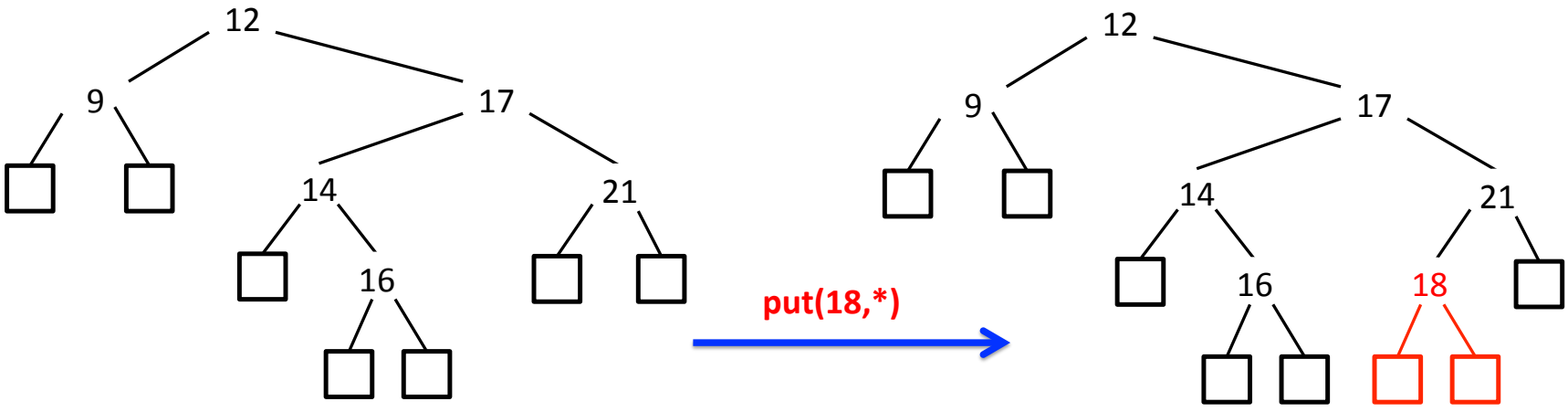


  incrementa numero di entry di T *di* 1;
  return *null*

  Complessità: $\Theta(h)$ dominata da `TreeSearch`

N.B.: In [GTG14] mancano le istruzioni di **return**, e per trasformare w in nodo interno si invoca `expandExternal`.

Esempio (solo chiavi)



Esercizio R-11.1 [GTG14]

Inserire in un albero binario di ricerca vuoto 8 entry con chiavi $30, 40, 24, 58, 48, 26, 11, 13$, e far vedere l'albero risultante dopo ciascun inserimento.

Esercizio

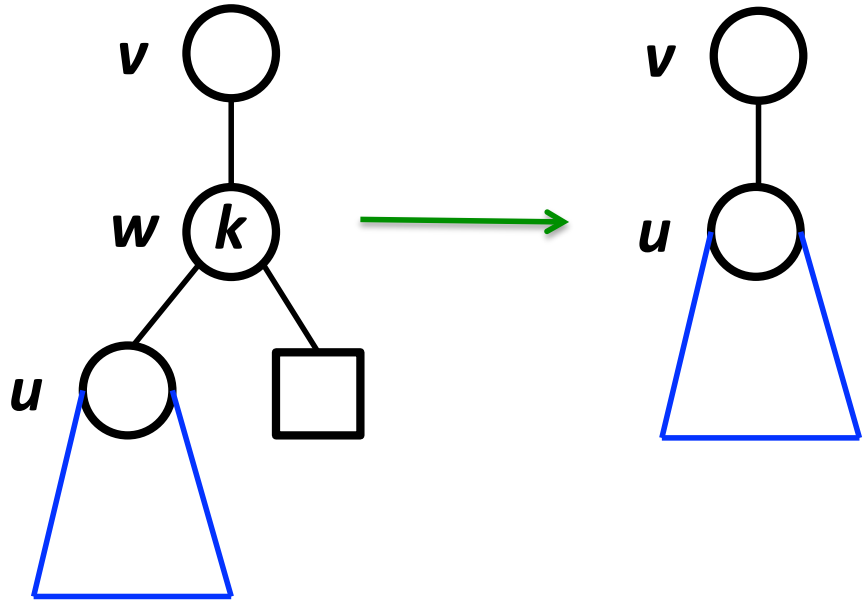
Se nell'esercizio precedente le stesse entry fossero inserite in un ordine diverso, alla fine si otterrebbe lo stesso albero? Motivare la risposta.

Implementazione dei metodi della Mappa: remove

- remove(k)

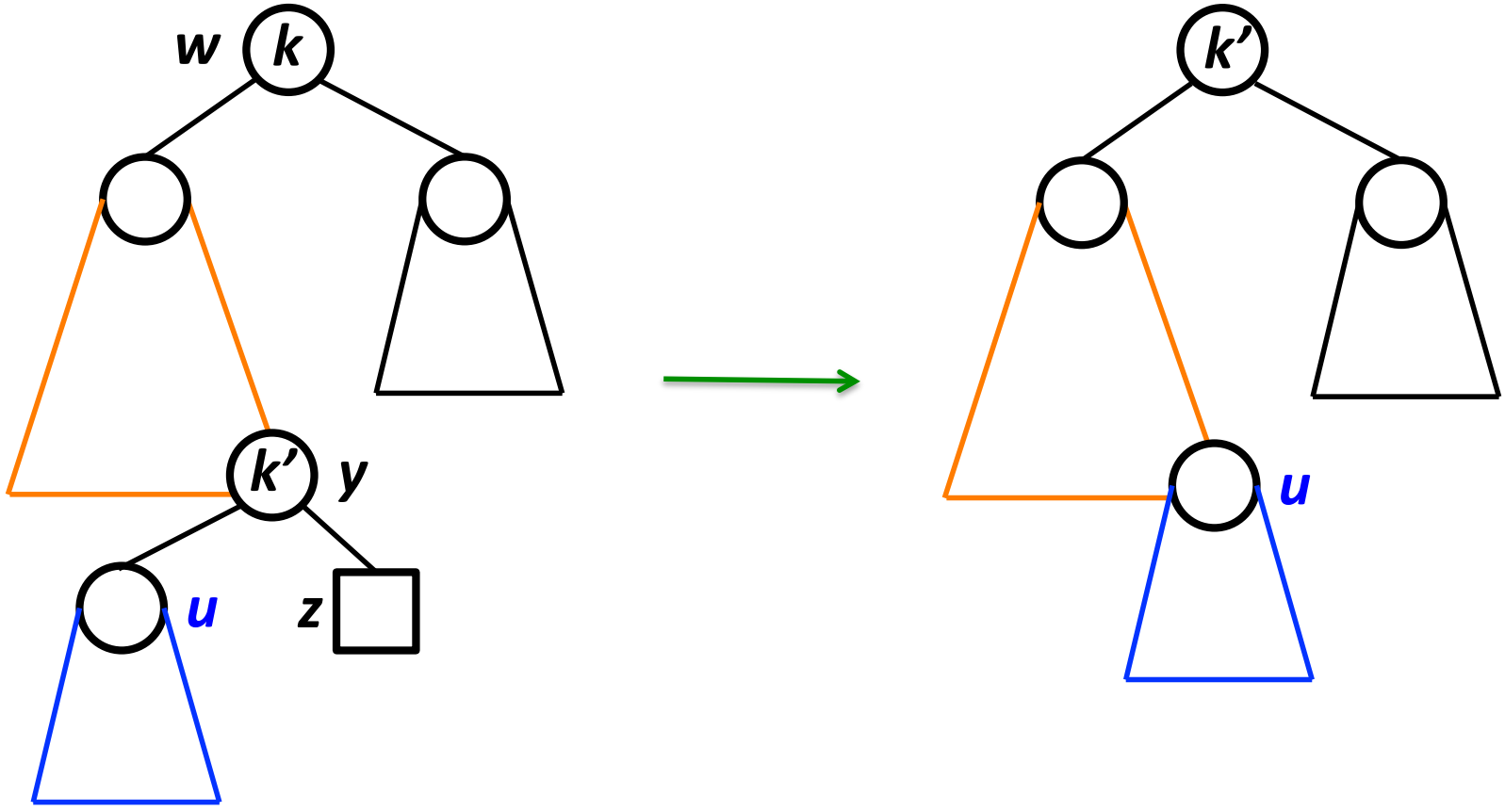
```
w ← TreeSearch(k, T.root());
if T.isExternal(w) then return null;
else
    value ← w.getElement().getValue();
    decrementa numero di entry in T di 1;
    if ((T.isExternal(T.left(w)) OR
     (T.isExternal(T.right(w))) then
        /* w interno e padre di almeno 1 foglia   */
        esegui Caso 1 (vedi lucidi successivi);
    else
        /* w interno e padre di 2 nodi interni    */
        esegui Caso 2 (vedi lucidi successivi);
return value;
```

Caso 1: w interno e padre di almeno 1 foglia



- se w era radice $\Rightarrow$ u diventa radice;
- caso analogo se foglia è a sx

Caso 2: w interno e padre di 2 nodi interni



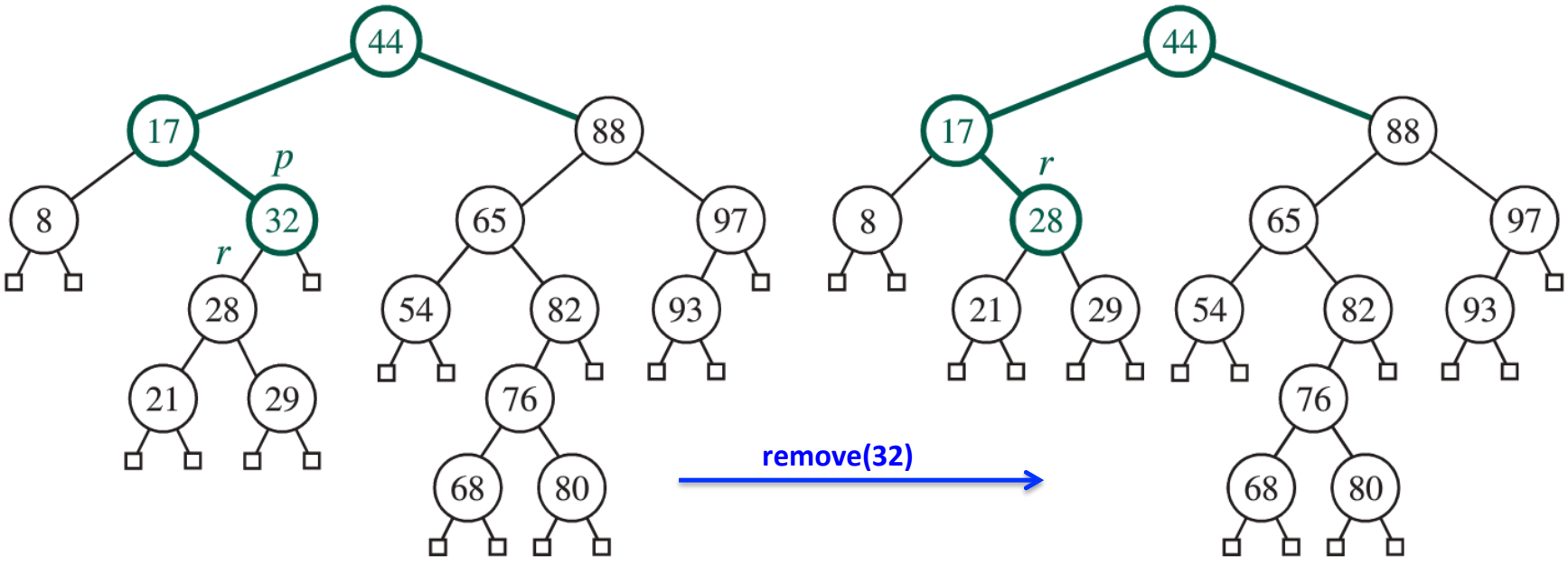
- y = *predecessore* "interno" di w nella visita inorder
  $\Rightarrow$ k' *predecessore* di k nell'ordine crescente delle chiavi

Complessità di `remove`

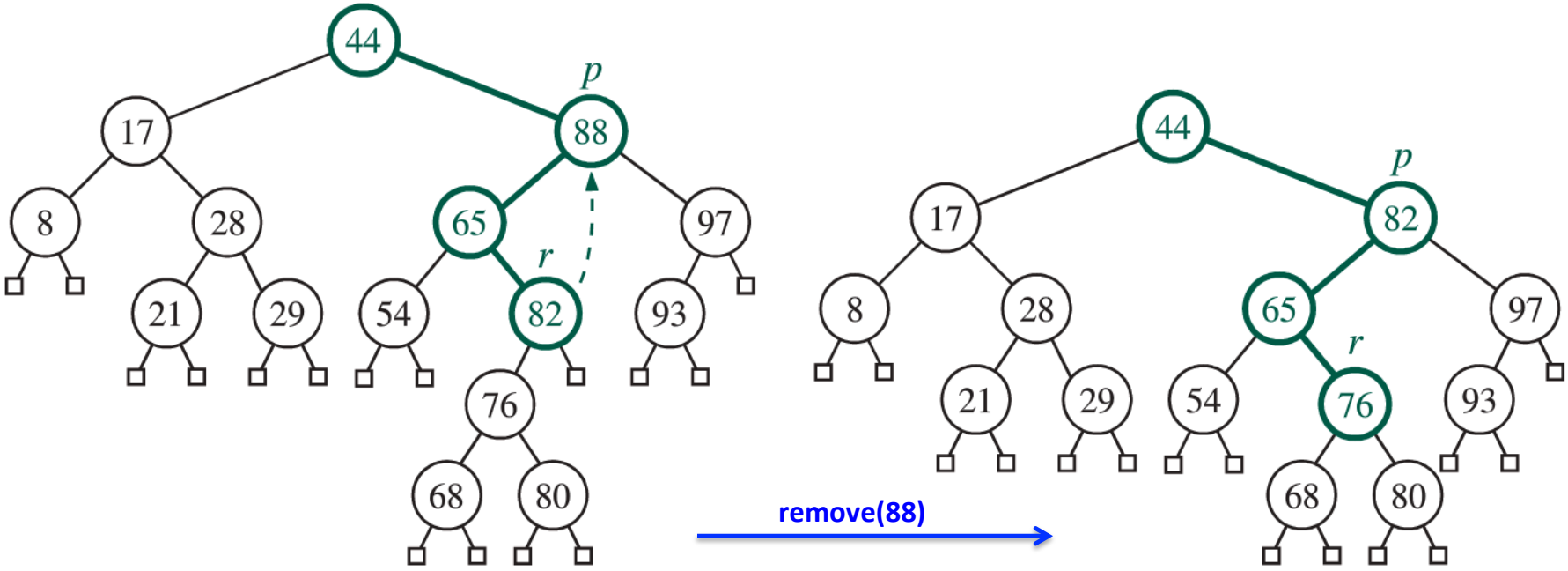
- $\Theta(h)$ per `TreeSearch` (con h l'altezza di T)
- $O(h)$ per trovare il predecessore interno di w nella visita inorder;
- $O(1)$ per fare l'update dei nodi

$\Rightarrow$ **Complessità:** $\Theta(h)$

Esempio (da [GTG14])



Esempio (da [GTG14] - continua)



Esercizio

Scrivere una versione iterativa di `TreeSearch` con la stessa specifica.

Svolgimento

Algoritmo `TreeSearch(`k`, `v`)`
Input: chiave k, nodo $v \in T$
Output: nodo $w \in T_v$ con chiave k, o w foglia nella posizione "giusta" per k

```
w ← v;
while true do
    if (T.isExternal(w) OR
     (w.getElement().getKey()=k)) then  return w;
    else
        if (k < w.getElement().getKey()) then
            w ← T.left(w);
        else
            w ← T.right(w);
```

Complessità: $\Theta(h)$ dato che in ciascuna iterazione del while (a) si eseguono $\Theta(1)$ operazioni e (b) w scende di un livello.

Esercizio

Progettare un algoritmo non ricorsivo che dato un nodo v di un albero binario di ricerca T e un intervallo $[A, B]$ di valori per le chiavi, restituisca un nodo $w \in T_v$ contenente una entry con chiave $k \in [A, B]$, se esiste, altrimenti una foglia nella posizione "giusta" per una chiave in $[A, B]$, e analizzarne la complessità.

Esercizio

Sia T un albero binario di ricerca dove ogni nodo $v \in T$ memorizza in una variabile `v.size` il numero di entry in T_v (inclusa quella in v). Progettare un algoritmo ricorsivo che conti quante entry in T hanno chiave $\leq k$, e analizzarne la complessità.

Svolgimento

Algoritmo `CountLE(`k`,`v`)`
Input: chiave k, nodo $v \in T$
Output: numero entry in T_v con chiave $\leq k$

```
if (T.isExternal(v)) then return 0;
if (v.getElement().getKey() > k) then
    return CountLE(k, T.left(v));
else return 1 + T.left(v).size+CountLE(k, T.right(v));
```

Complessità. Come `TreeSearch`: $\Theta(h)$

Esercizio

Modificare l'esercizio precedente per contare quante entry hanno chiave $\geq k$.

Esercizio

Risolvere lo stesso problema dell'esercizio precedente usando un algoritmo non ricorsivo.

Esercizio

Sia T un albero binario di ricerca. Progettare un algoritmo ricorsivo che conti quante entry in T hanno chiave $\leq k$, e analizzarne la complessità. (N.B. Non si ha a disposizione la variabile `size` come nell'esercizio precedente.)

Svolgimento

Idea: stesso algoritmo di prima, sostituendo l'uso della variabile `size` con una chiamata ricorsiva.

Algoritmo `CountLE(`k`,`v`)`
Input: chiave k, nodo $v \in T$
Output: numero entry in T_v con chiave $\leq k$

```
if (T.isExternal(v)) then return 0;
if (v.element().getKey() > k) then
    return CountLE(k,T.left(v));
else
    return
     1+CountLE(k,T.left(v))+CountLE(k,T.right(v))
```

Analisi di complessità (tramite albero della ricorsione)

Dato che il costo di ciascuna chiamata ricorsiva (escluse le chiamate fatte al suo interno) è $O(1)$, la complessità dell'algoritmo è proporzionale al numero di nodi dell'albero della ricorsione.

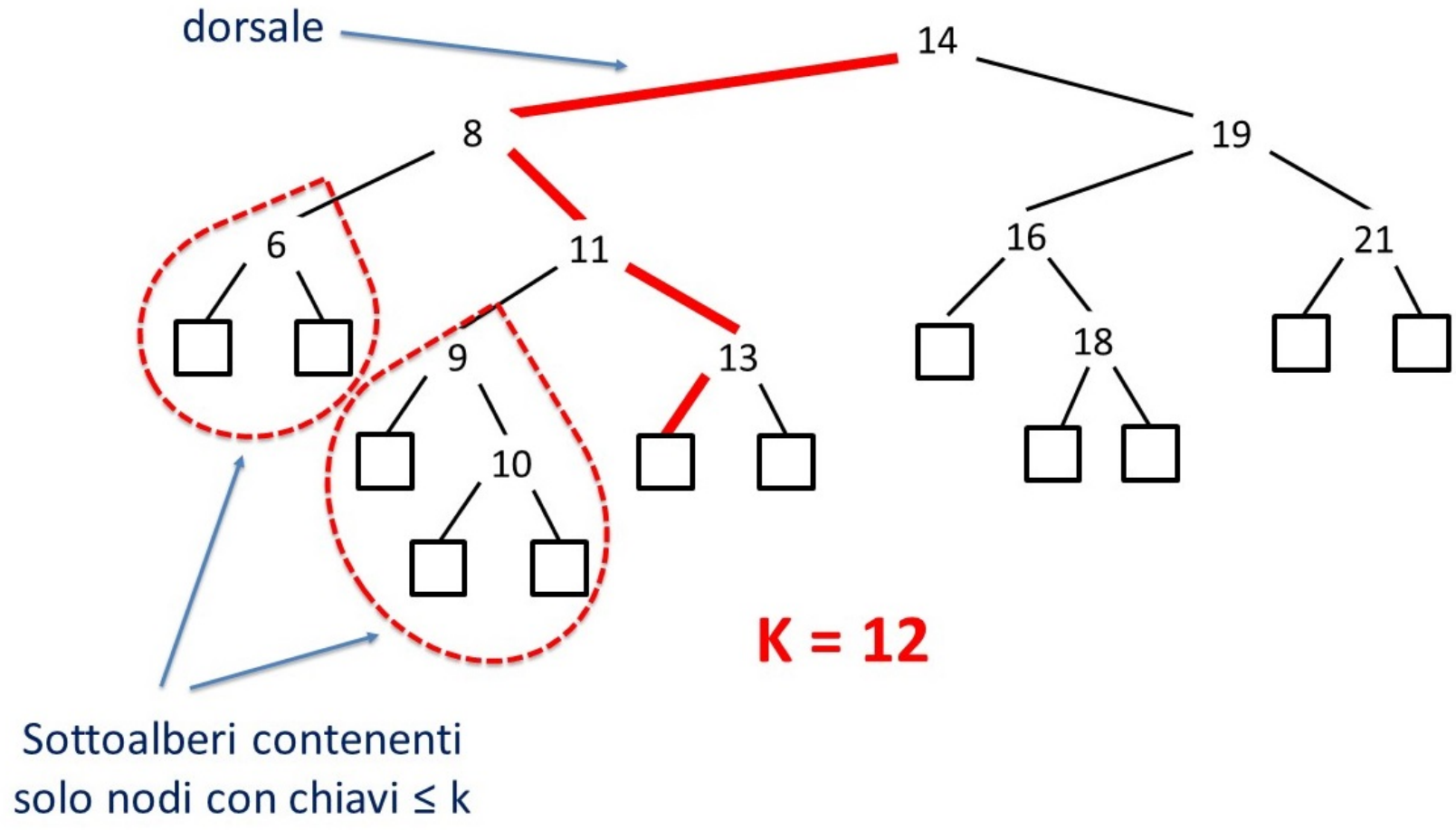
Stimiamo il numero di nodi nell'albero della ricorsione associato alla esecuzione di `CountLE(k,T.root())`, per una generica chiave k. A tale fine, consideriamo una generico nodo dell'albero della ricorsione associato a `CountLE(k,v)`, per un qualche $v \in T$. Si osservi che:

- Se tutte le entry in T_v hanno chiave $\leq k$ allora l'algoritmo verrà invocato su ciascun $u \in T_v$.
- Se la chiave nel nodo v è $> k$, l'algoritmo richiama se stesso solo sul figlio sx di v.
- Se la chiave nel nodo v è $\leq k$, l'algoritmo richiama se stesso su entrambi i figli. Il figlio sx è radice di un sottoalbero che contiene solo entry con chiave $\leq k$, e, per il primo punto, l'algoritmo verrà invocato su tutti i nodi di tale sottoalbero.

Sia h l'altezza di T e s il numero di entry con chiave $\leq k$ in T.

- Definiamo una *dorsale* di nodi sui quali viene invocato l'algoritmo, che parte da T`.root()` e arriva a una foglia: se un nodo v della dorsale ha chiave $> k$, il successore nella dorsale è il suo figlio sx, mentre se ha chiave $\leq k$ il successore nella dorsale è il suo figlio dx (ma in questo caso l'algoritmo sarà invocato anche sul figlio sx).
- Vi sono $\leq h$ nodi nella dorsale (percorso radice-foglia).
- Vi sono $\leq s$ nodi al di fuori della dorsale sui quali viene invocato l'algoritmo. L'insieme di questi nodi coincide con l'unione di sottoalberi T_u che contengono solo nodi con chiave $\leq k$, dove u non appartiene alla dorsale ma è figlio sx di un nodo della dorsale.

$\Rightarrow$ **Complessità:** $O(h + s)$.


dorsale
14
8
19
6
11
16
21
9
13
18
10
K = 12
Sottoalberi contenenti
solo nodi con chiavi ≤ k

Esercizio

Sia T un albero binario di ricerca contenente n entry con chiavi reali distinte. Descrivere un algoritmo non ricorsivo che determina la più piccola chiave positiva presente in T, e analizzarne la complessità. Se in T non esistono chiavi positive, l'algoritmo deve restituire `'no positive key'`.

Esercizio

Sia T un albero binario di ricerca le cui entry rappresentano studenti di un'università. Ogni studente è associato a una entry (k, x), dove k è la matricola, e x indica se lo studente è straniero ($x = 1$) o italiano ($x = 0$). Per ogni nodo $v \in T$ esiste un intero v`.numStr` che riporta il numero di studenti stranieri in T_v (sottoalbero con radice v). Progettare un algoritmo ricorsivo `MinMatStraniero`(T, v) che dato un nodo $v \in T$ restituisce *la più piccola matricola di uno studente straniero in* T_v, e analizzarne la complessità. Se non ci sono studenti stranieri in T_v l'algoritmo restituisce `null`.

Multi-Way Search Tree e (2,4)-Tree

(Paragrafo 11.4 [GTG14])

Multi-Way Search Tree (MWS-Tree)

I MWS-Tree generalizzano gli Alberi Binari di Ricerca permettendo ai nodi di contenere più entry, cosa che rende più agevole il bilanciamento (variante (2,4)-*Tree*)

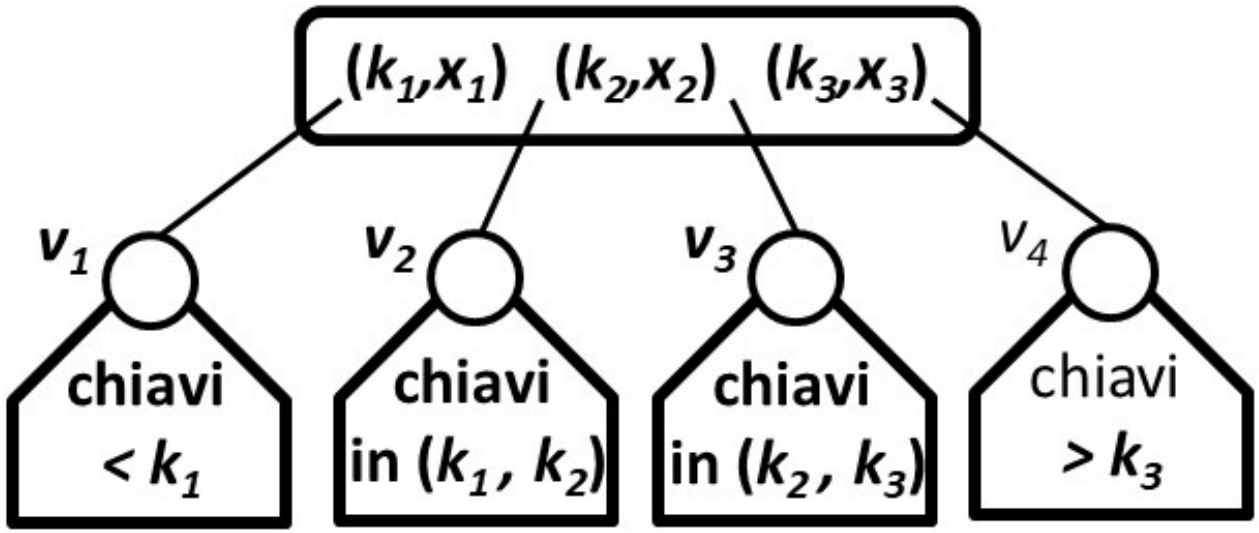
Definizione

Un MWS-Tree T è un albero ordinato tale che

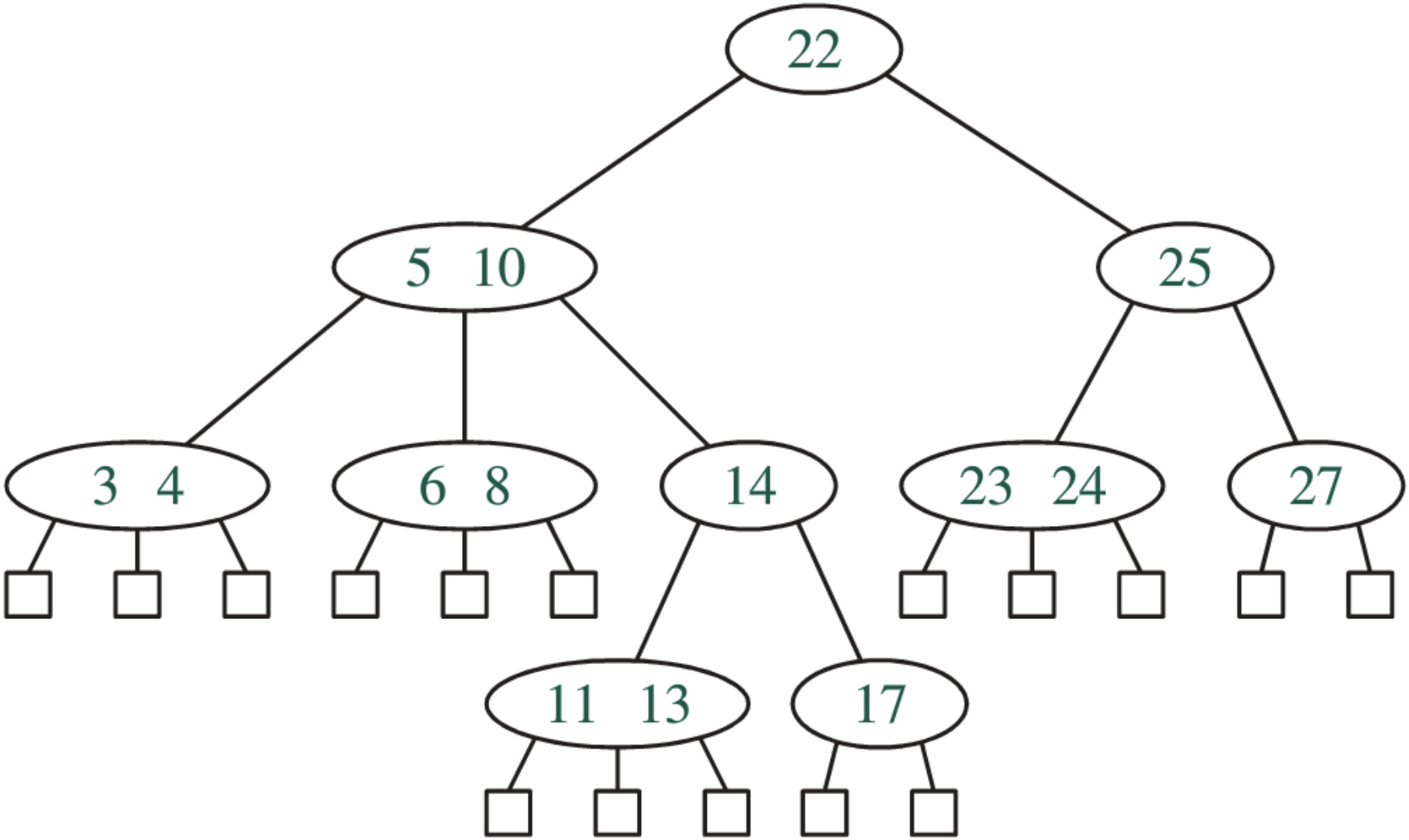
- ogni nodo interno ha ≥ 2 figli
- ogni nodo interno con $d \geq 2$ figli $v_1, v_2, \ldots, v_d$ (*d-node*) soddisfa le seguenti proprietà:
  - memorizza $d-1$ entry: $(k_1,x_1),(k_2,x_2),\ldots,(k_{d-1},x_{d-1})$ dove $k_1 < k_2 < \cdots < k_{d-1}$
  - per $1 \leq i \leq d$ vale che la chiave di ogni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

N.B.: come per gli alberi binari di ricerca, per convenzione le foglie non memorizzano entry

Esempio di 4-node



Esempio di MWS-Tree (solo chiavi)



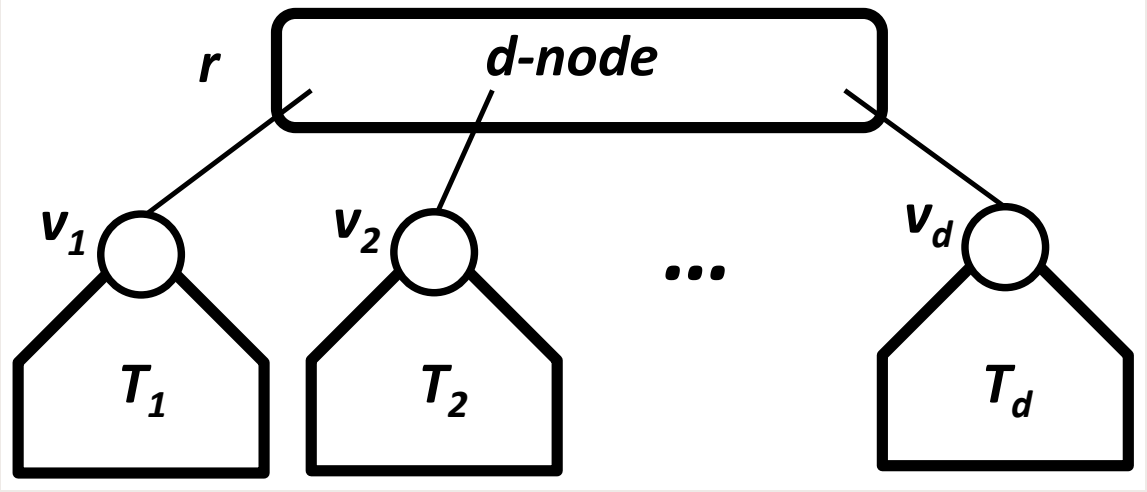
Proposizione (11.2 [GTG14])

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

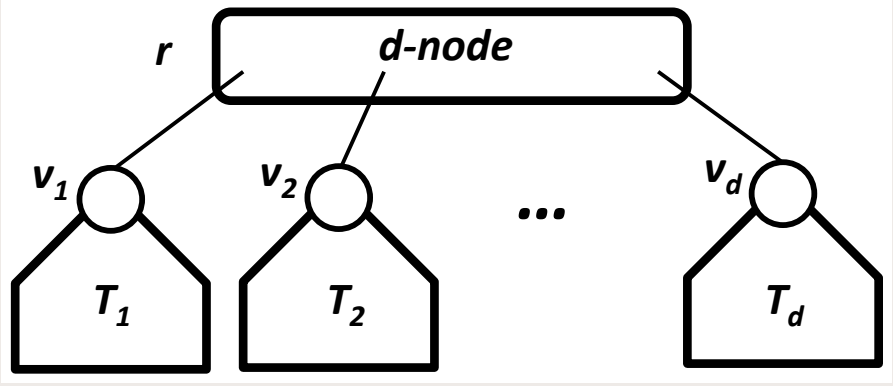
Dimostrazione (Esercizio C-11.46)

Per induzione sull'altezza h.

- Base. $h = 0$: $n = 0 \Rightarrow 1$ foglia $\Rightarrow$ OK
- Passo induttivo. Hp. induttiva: OK fino a $h \geq 0$
  $\Rightarrow$ consideriamo $h+1$ T = MWS-Tree di altezza $h+1$



Dimostrazione (continua)



Per $i, 1 \leq i \leq d$: $n_i =$ num. entry in T_i $\quad m_i =$ num. foglie in T_i

$n =$ num. entry in T $\quad m =$ num. foglie in T

$$n = d - 1 + \sum_{i=1}^{d} n_i \qquad m = \sum_{i=1}^{d} m_i$$

Per hp. induttiva (perchè altezza$(T_i) \leq h$):

$$m = \sum_{i=1}^{d} m_i = \sum_{i=1}^{d} (n_i + 1) = d + \sum_{i=1}^{d} n_i = 1 + (d-1) + \sum_{i=1}^{d} n_i = 1 + n \quad \square$$

Osservazione

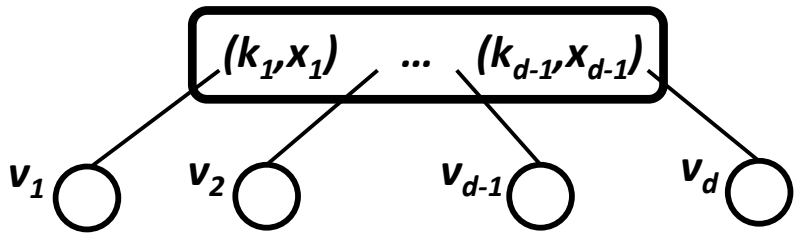
Se tutti i nodi interni fossero dei d-node l' MWS-Tree T sarebbe un albero d-ario. Detto x il numero di nodi interni e y il numero di foglie di T sappiamo (Problema 8 della dispensa di esercizi sugli alberi) che $y = (d-1)x + 1$, che è coerente con la proposizione precedente dato che l'albero in questo caso memorizzerebbe $n = (d-1)x$ entry.

Ricerca in un MWS-Tree T

(In [GTG14] l'algoritmo è descritto solo a parole)

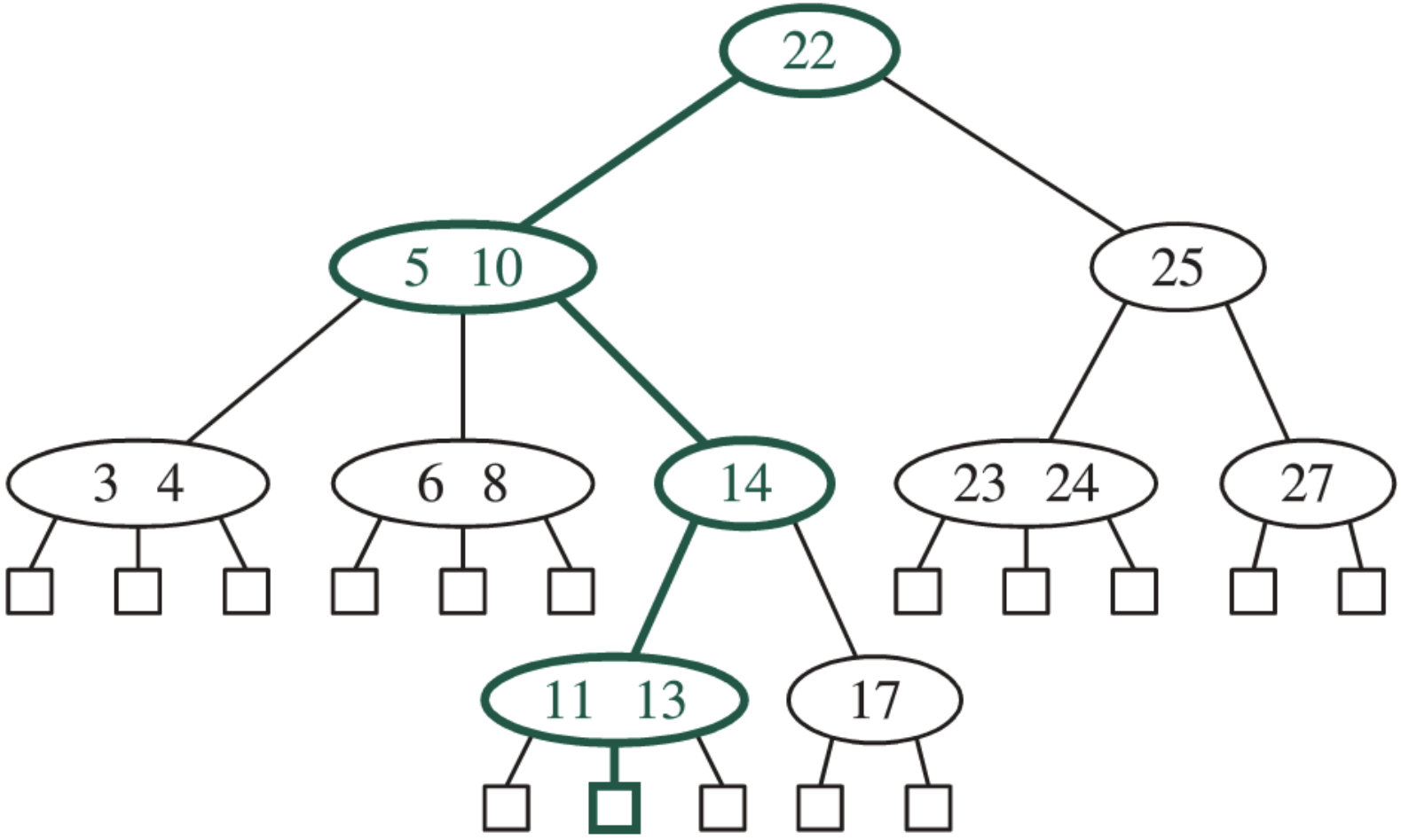
Algoritmo `MWTreeSearch(`k`,`v`)`
Input: chiave k, nodo $v \in T$
Output: nodo di T_v contenente una entry con chiave k, se esiste, o foglia in posizione "giusta" per k

if (`T.isExternal(`v`)`) **then return** v;
Siano (k_1, x_1),..., (k_{d-1}, x_{d-1}) *le entry in* v *con* $k_1 < \cdots < k_{d-1}$;

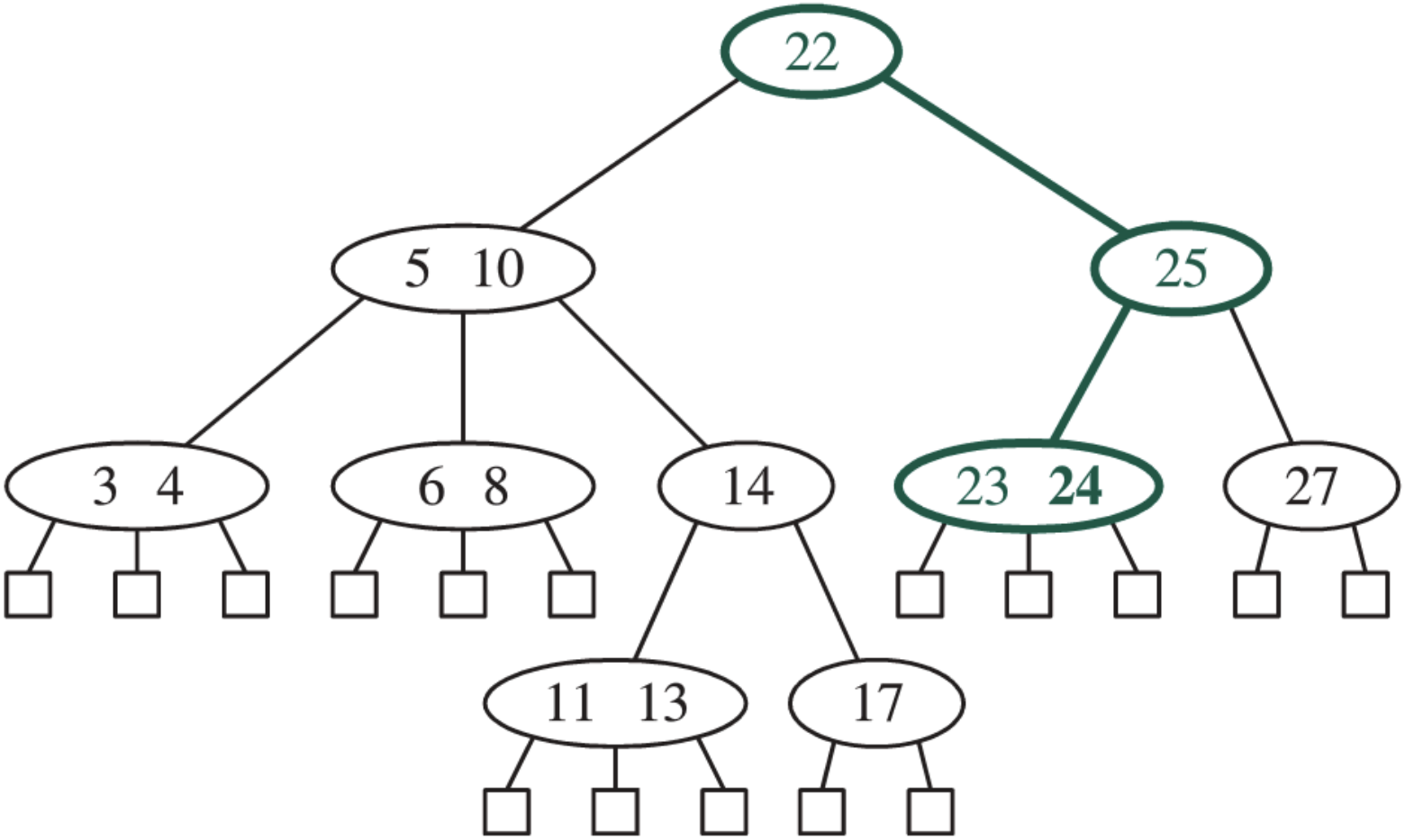


Trova i *tale che* $k_{i-1} < k \leq k_i$ (si assuma $k_0 = -\infty$, $k_d = +\infty$);
if (k=k_i) **then return** v;
else return `MWTreeSearch(`k`,`v_i`)`;

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



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



Complessità di MWTreeSearch

Si assuma di memorizzare le entry in ciascun nodo tramite una mappa (ad esempio una lista) che costituisce quindi una struttura dati *secondaria* all'interno di quella *primaria* (l'MWS-Tree).

Analizziamo la complessità di `MWTreeSearch(k,T.root())`, per una chiave k arbitraria, usando l'albero della ricorsione. Sia h l'altezza di T e $d_{\max}$ il numero massimo di figli di un nodo interno:

- `MWTreeSearch` è invocato sui nodi di un cammino di lunghezza $\leq h$ (un'invocazione per nodo).
- Ogni invocazione esegue $O(d_{\max})$ operazioni, per trovare il nodo v_i su cui continuare la ricerca, oltre a un'eventuale chiamata ricorsiva.

$\Rightarrow$ **complessità** $O(d_{\max} h)$

Implementazione dei metodi della Mappa: get

get(*k*)

```
w ← MWTreeSearch(k, T.root());
if (T.isExternal(w)) then return null;
else
    trova e ∈ 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

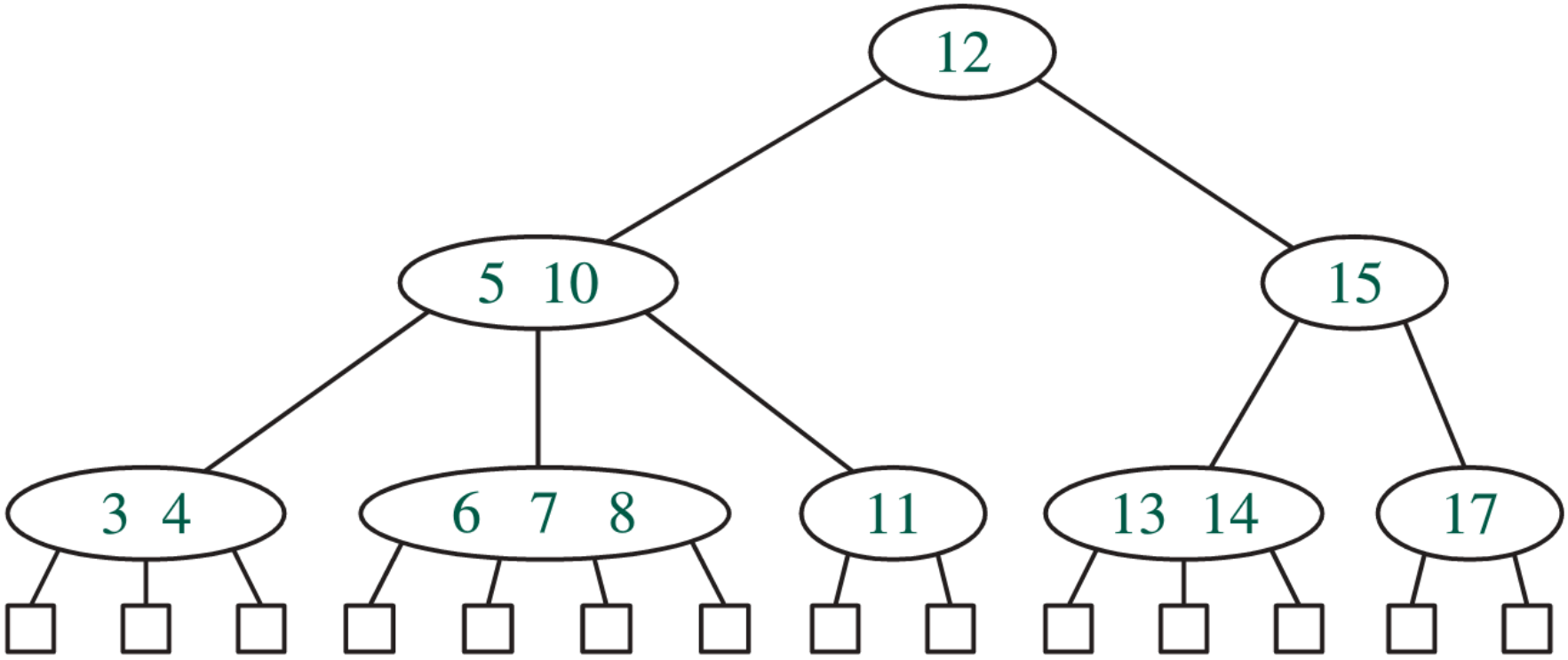
Definizione

Un (2,4)-Tree è un MWS-Tree tale che

- ogni nodo interno è un d-node con $2 \leq d \leq 4$;
- 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 (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 11.3

m_i = num. nodi al livello i ($0 \leq i \leq h$, h = livello delle foglie = altezza albero)

$$
\begin{aligned}
m_0 &= 1 \\
2 \leq m_1 &\leq 4 \\
2^2 \leq m_2 &\leq 4^2 \\
&\dots \\
2^i \leq m_i &\leq 4^i \\
\Rightarrow 2^h \leq m_h &\leq 4^h
\end{aligned}
$$

Poiché $m_h = n + 1$ abbiamo che:

$$
\begin{aligned}
n + 1 \geq 2^h &\Rightarrow h \leq \log_2(n+1) \\
n + 1 \leq 4^h &\Rightarrow \tfrac{1}{2}\log_2(n+1)
\end{aligned}
$$

$\Rightarrow h \in \Theta(\log n)$ □

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.

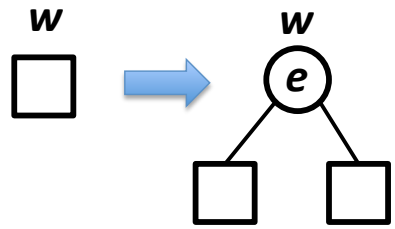
Implementazione dei metodi della Mappa: put

put(k,x**)**

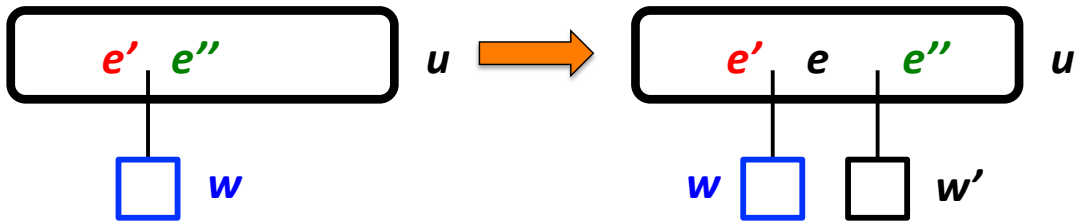
$w \leftarrow$ `MWTreeSearch(`k`,`T`.root());`
if (`T.isInternal(`w`)`) **then**
  trova $e \in w$ tale che e.`getKey()`$= k$;
  $y \leftarrow e$.`getValue()`;
  sostituisci x a y in e;
  return y;
$e \leftarrow (k,x)$;
if (`T.isRoot(`w`)`) **then**
  trasforma w in nodo interno contenente e con 2 figli foglia
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*;

Descrizione grafica delle operazioni in rosso del lucido precedente:

trasforma w in nodo interno contenente e con 2 figli foglia (*w* radice):

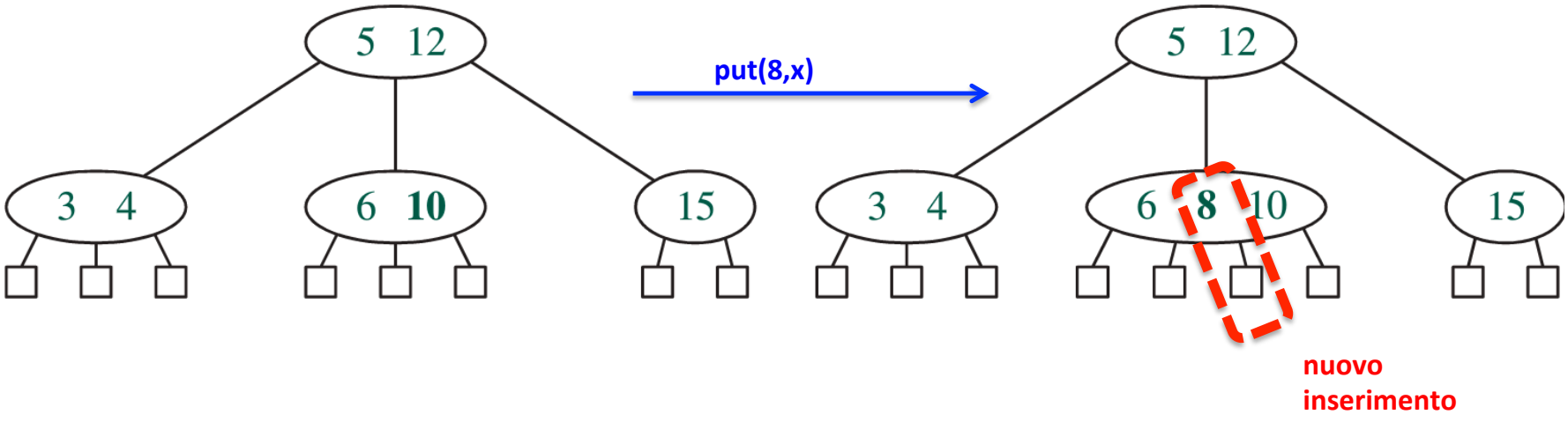


inserisci e in u aggiungendo una foglia w′ (*w* non radice):



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

Esempio (senza split)



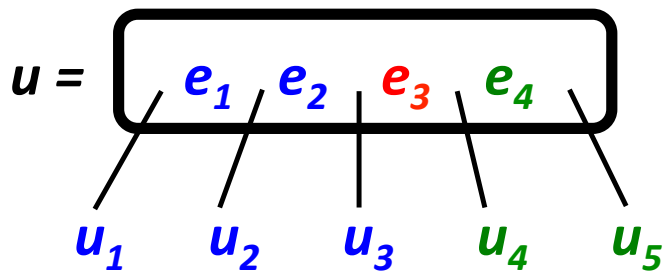
`split(`u`)`

Descritta a parole in [GTG14]

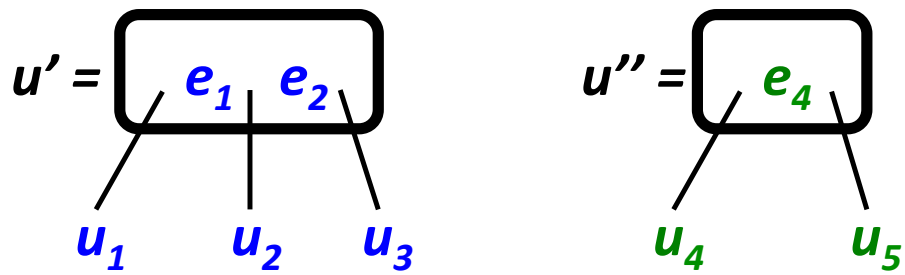
Input: 5-node $u \in T$, unica violazione delle proprietà di (2,4)-Tree

Output: Ripristino dele proprietà di (2,4)-Tree

Sia



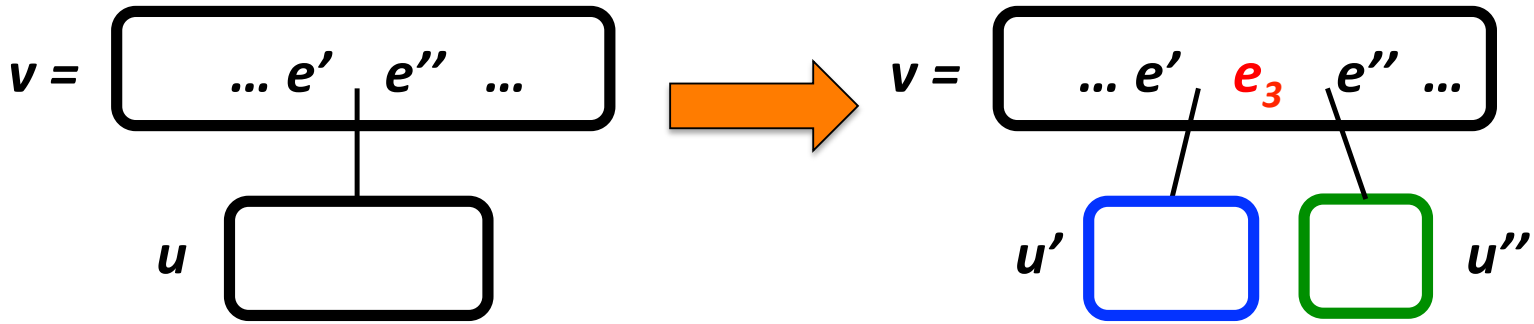
Crea



...

split(u) (continua)

```
...
if (!T.isRoot(u)) then
    v ← T.parent(u);                    /* vedi figura sotto */
    inserisci e3 in v tra e' ed e'';
    assegna u', u'' come figli di v a sx e dx di e3;
    cancella u;
    if (v è un 5-node) then split(v);
else                                    /* u è radice */
    crea una nuova radice contenente e3 e con 2 figli u', u'';
    cancella u;
```



Nota. Uno tra e' e e'' potrebbe non esistere.

Complessità di `put`

Proposizione

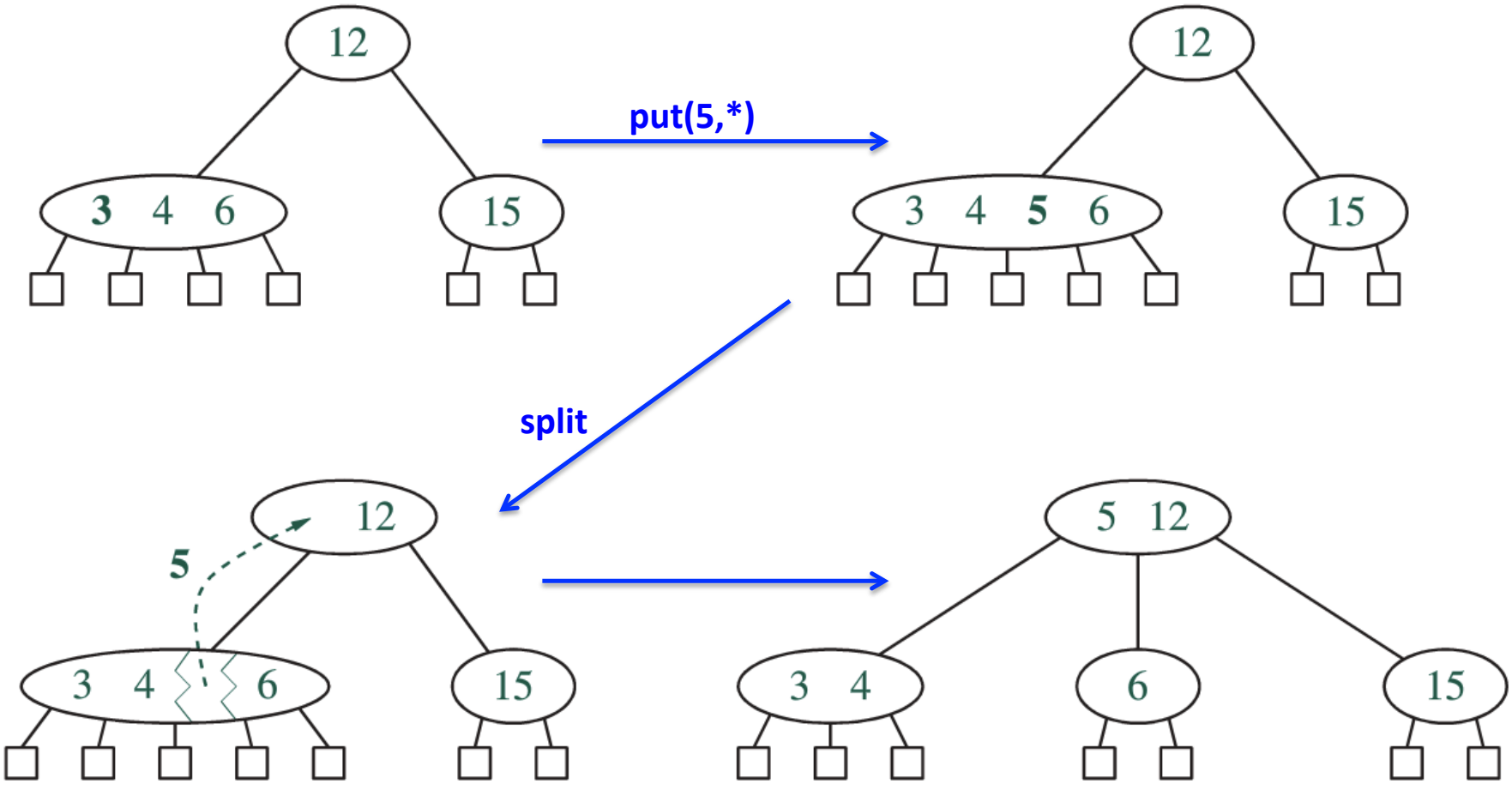
Per una mappa con n entry implementata tramite un (2,4)-Tree la complessità di `put` è $\Theta(\log n)$.

Dimostrazione

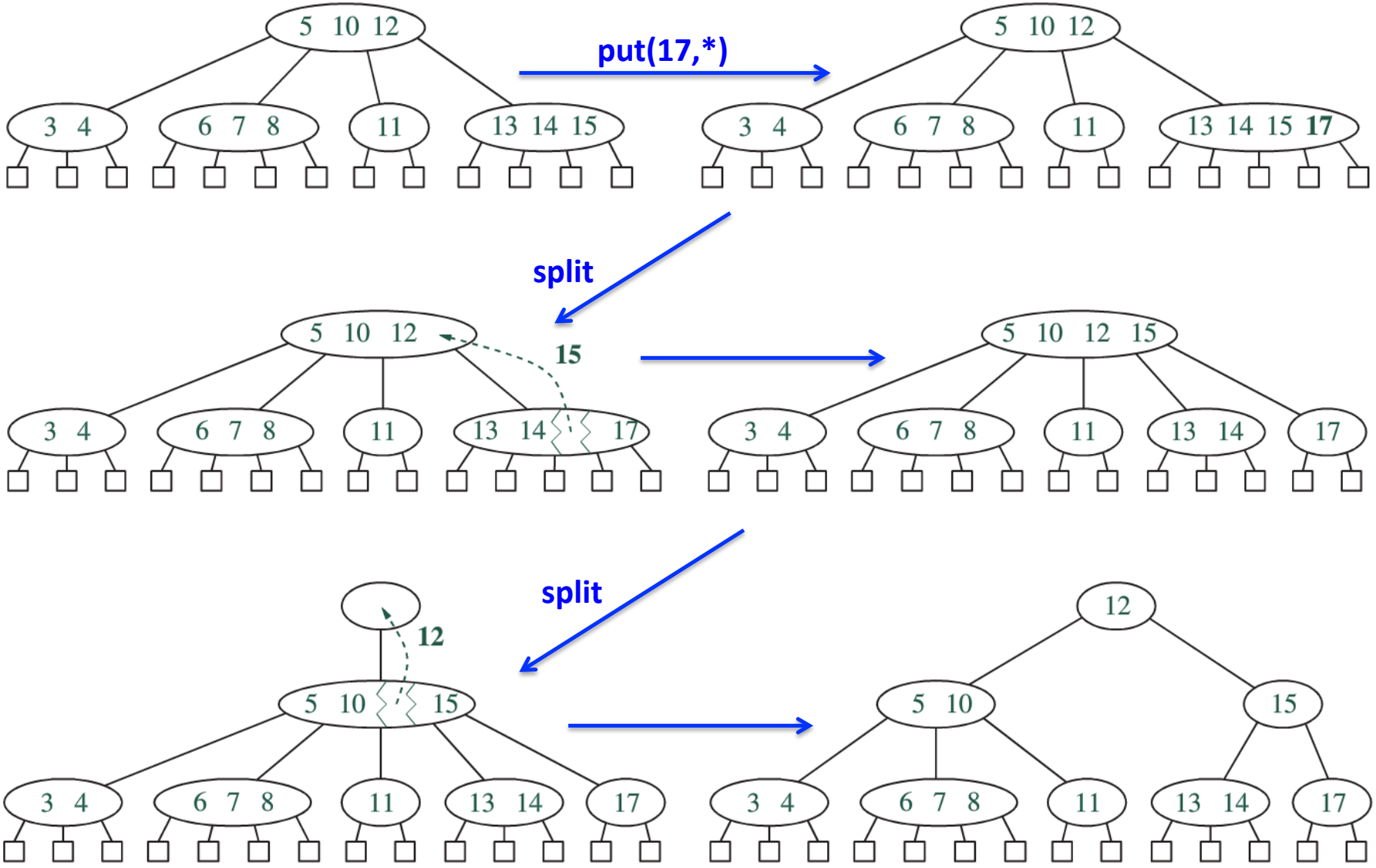
La complessità è dominata dalla esecuzione di `MWTreeSearch` e dalla eventuale esecuzione di `Split`. La complessità di `MWTreeSearch` è $\Theta(\log n)$, come provato in precedenza. `Split` viene invocato (ricorsivamente) sui nodi di un cammino che da una foglia risale verso la radice. Su ciascun nodo esegue $O(1)$ operazioni oltre a un eventuale chiamata ricorsiva sul padre. Se ne deduce che la complessità è al più proporzionale all'altezza $h \in \Theta(\log n)$.

Osservazione: L'albero cresce in altezza dalla radice.

Esempio



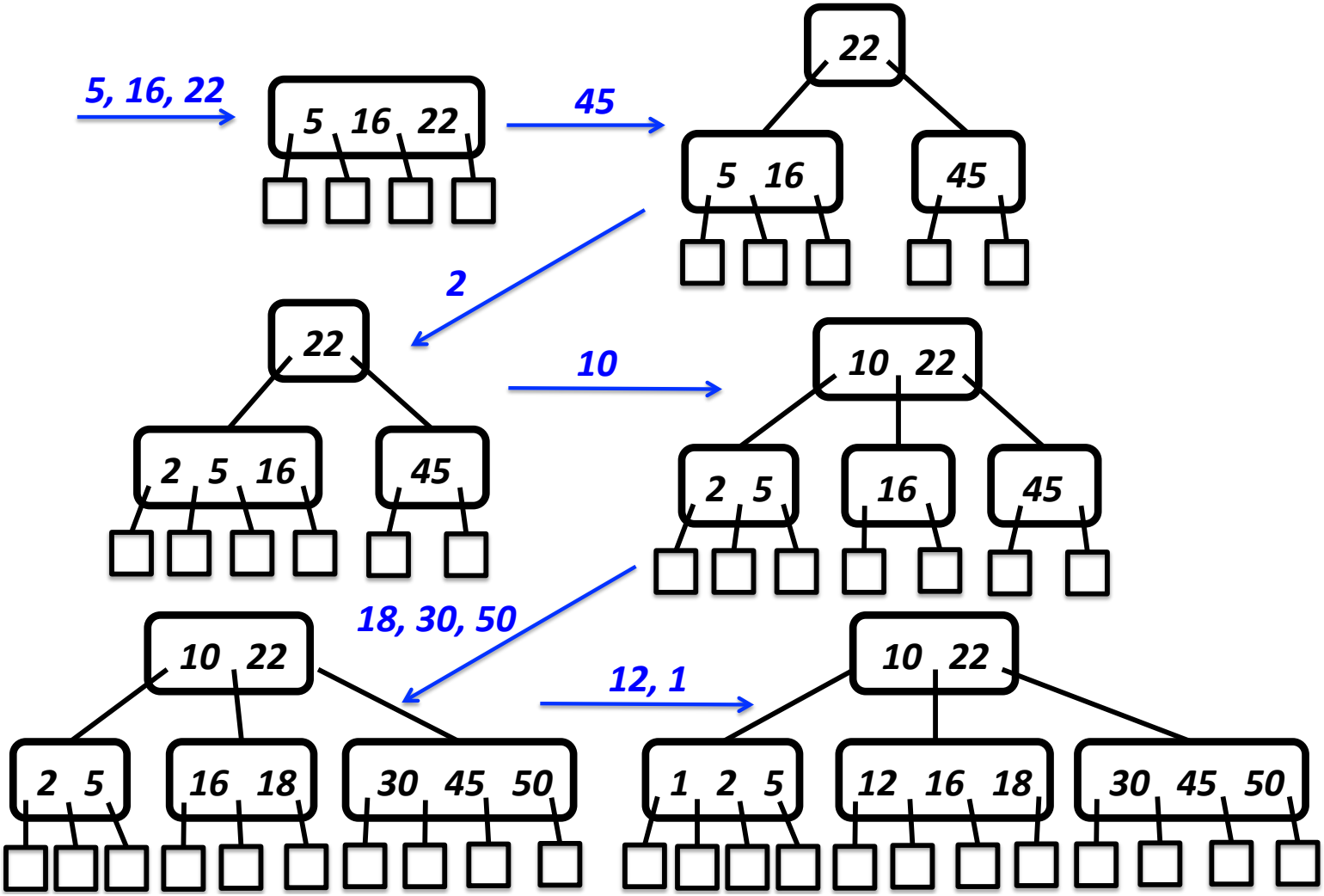
Esempio


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
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

Esercizio (R-11.18.a [GTG14])

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

Svolgimento

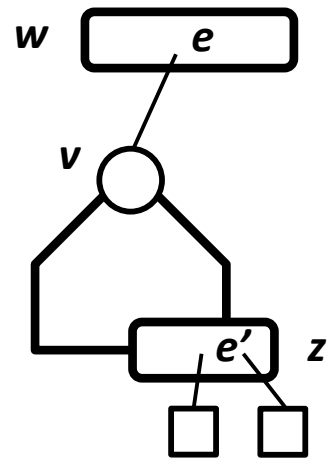


Implementazione metodi della mappa: `remove`

remove(k)

$w \leftarrow$ `MWTreeSearch(`k`,`T`.root());`
if `(`T`.isExternal(`w`))` **then return** *null*;
else
    trova $e \in w$ *tale che* e`.getKey()` $= k$;
    $y \leftarrow e$.getValue();
    if `(altezza(`w`)=1)` **then** `Delete(`e`,`w`,`R`);`
    else
        sia v *il figlio di* w *a sx di* e;
        $z \leftarrow$ *ultimo nodo interno nella visita in preorder di* T_v;
        $e' \leftarrow$ entry con chiave max in z;
        metti una copia di e' *al posto di* e *in* w;
        `Delete(`e'`,`z`,`R`);`
decrementa di 1 *il numero di entry in* T;
return y;

Osservazione: quando il nodo w restituito da `MWTreeSearch` è ad altezza > 1, la entry e' che si sostituisce al posto di e in w è il predecessore di e nella sequenza delle entry ordinate per chiave.



Delete(e, u, α)

Descritta a parole in [GTG14]

Input: entry e nel nodo $u \in T$ tale che il figlio a sx ($\alpha = L$) o a dx ($\alpha = R$) della entry è cancellabile

Output: rimozione di e da T ripristinando le proprietà di (2,4)-Tree

Siano A, Z i figli di u discriminati da e dove Z è quello cancellabile;

rimuovi e e Z;

if (*u* `non ha più entry`) **then** `/* underflow */`

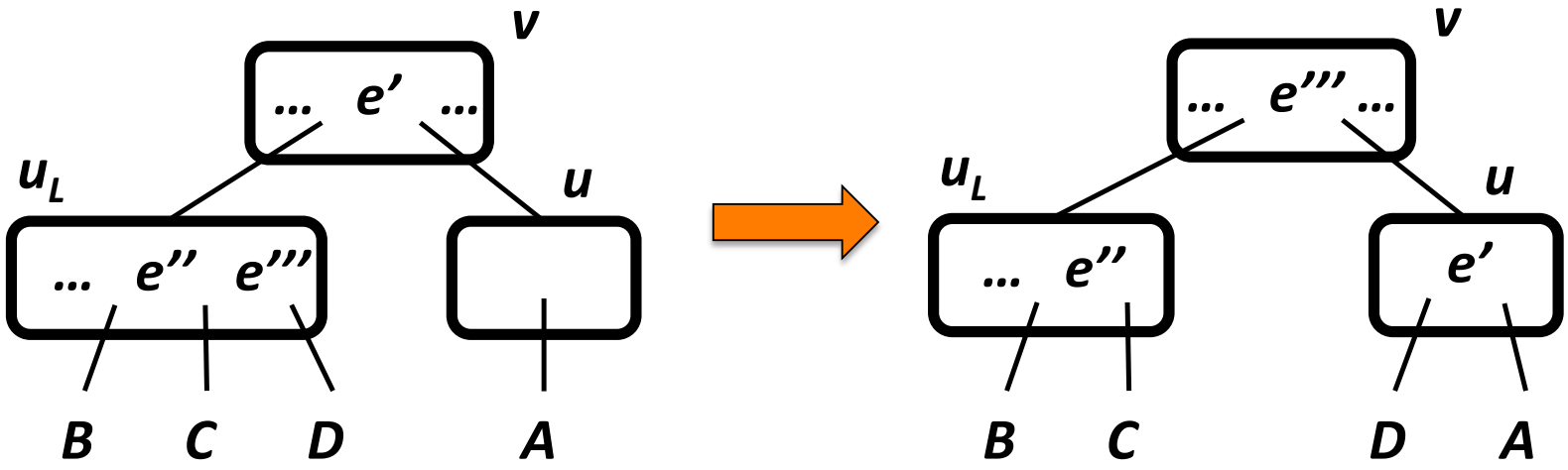
  esegui il caso giusto tra i 5 delle prossime slide;

Caso 1: u radice

imposta A come nuova radice del (2,4)-Tree;
return;

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

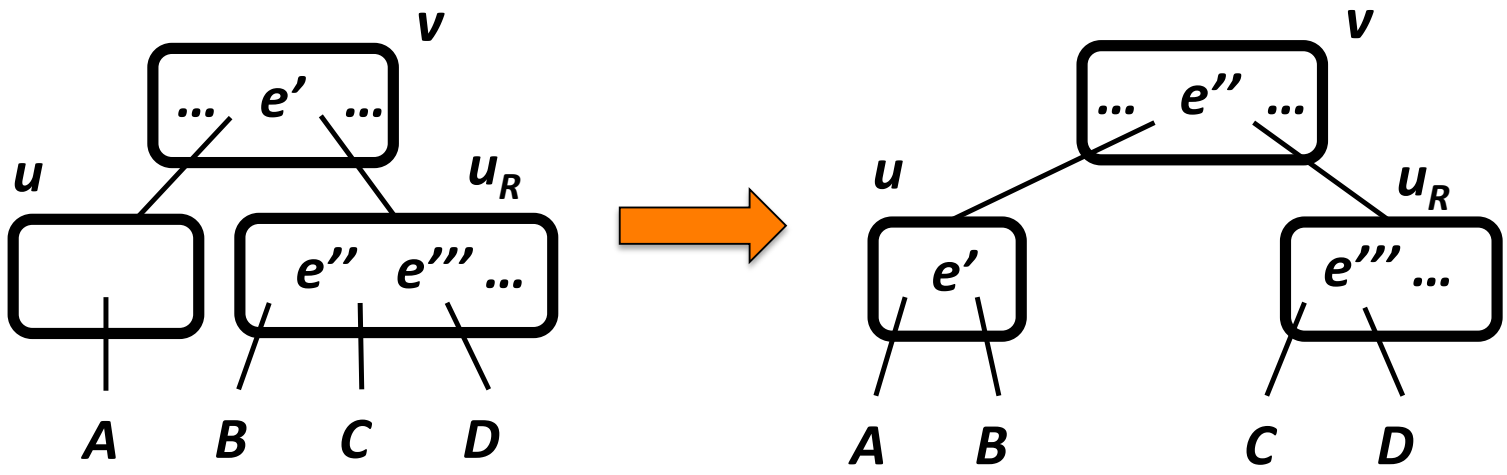
Esegui la seguente operazione:



return;

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

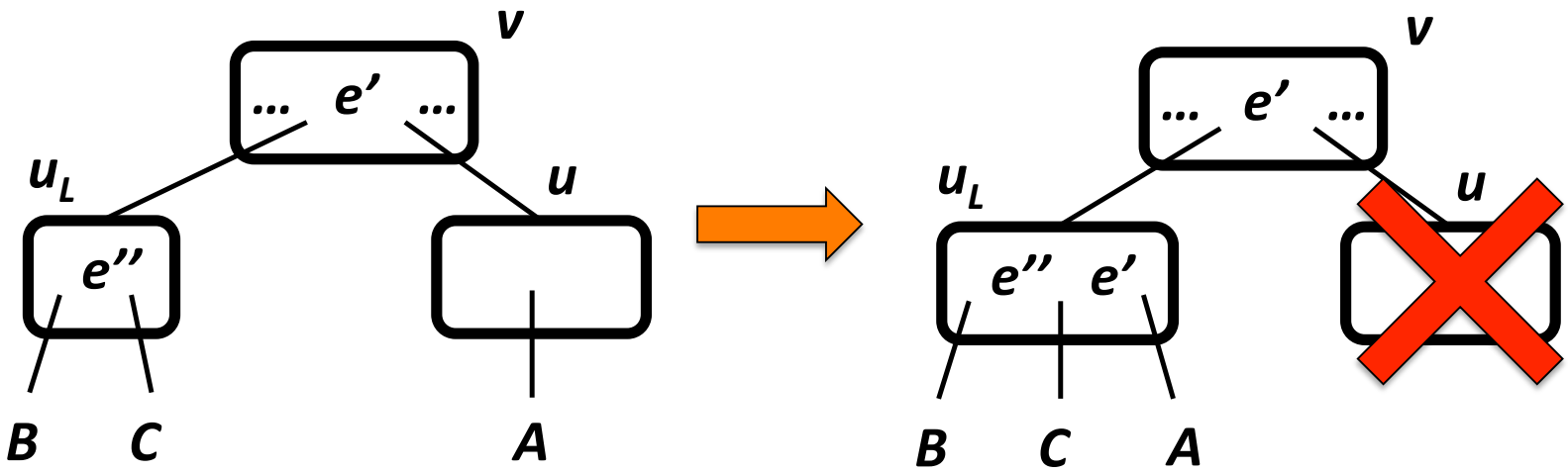
Esegui la seguente operazione:



return;

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

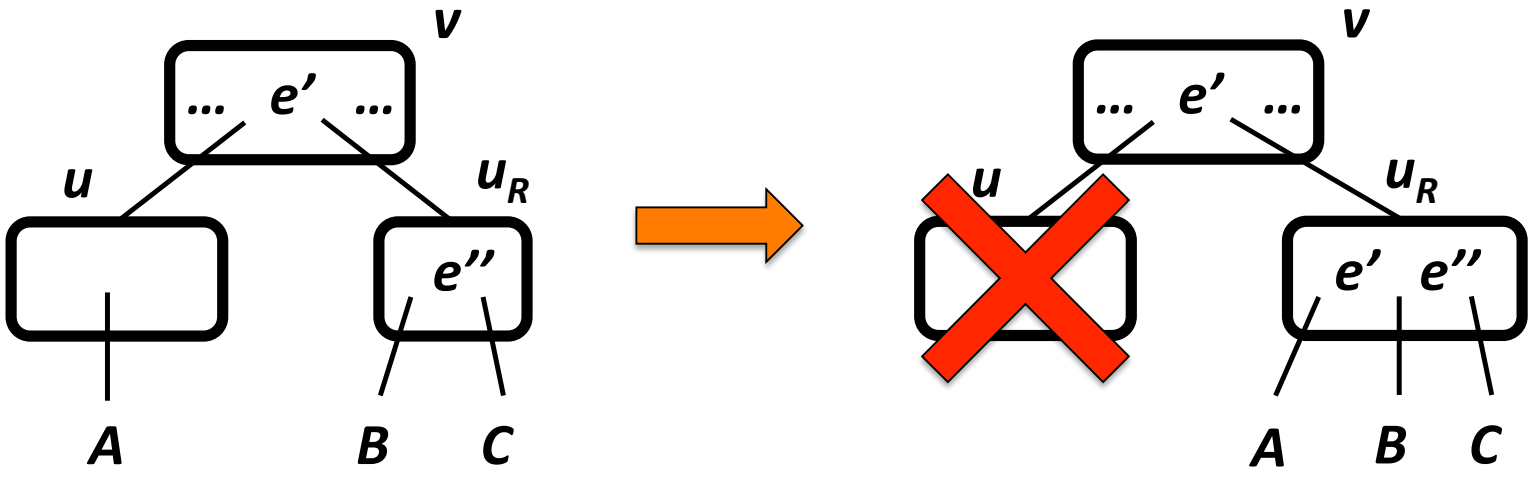
Esegui la seguente operazione:




```
Delete(e′,v,R);        /* propagazione verso l'alto */
return;
```

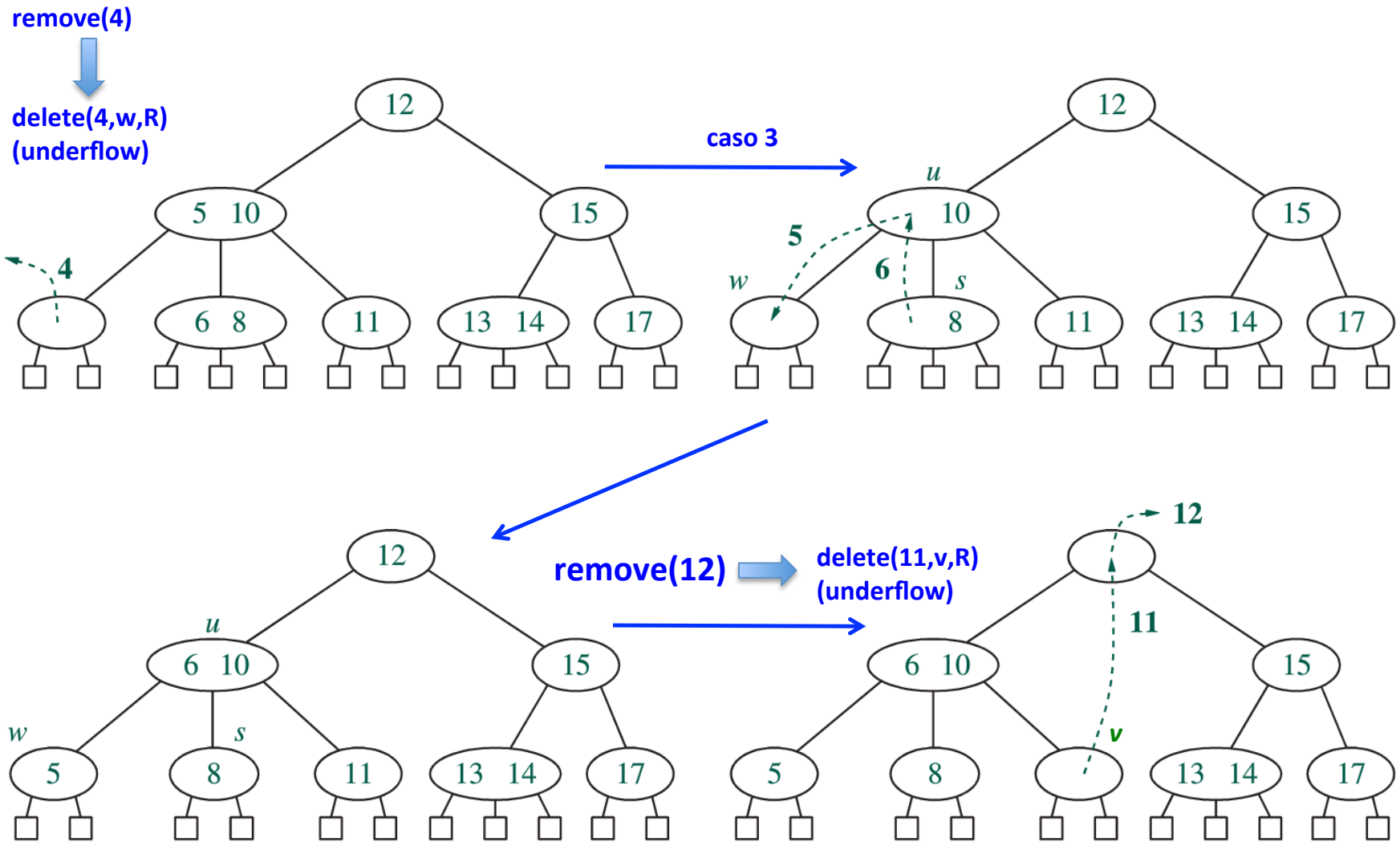
Caso 5: u ha un fratello u_R a dx che è un 2-node

Esegui la seguente operazione:

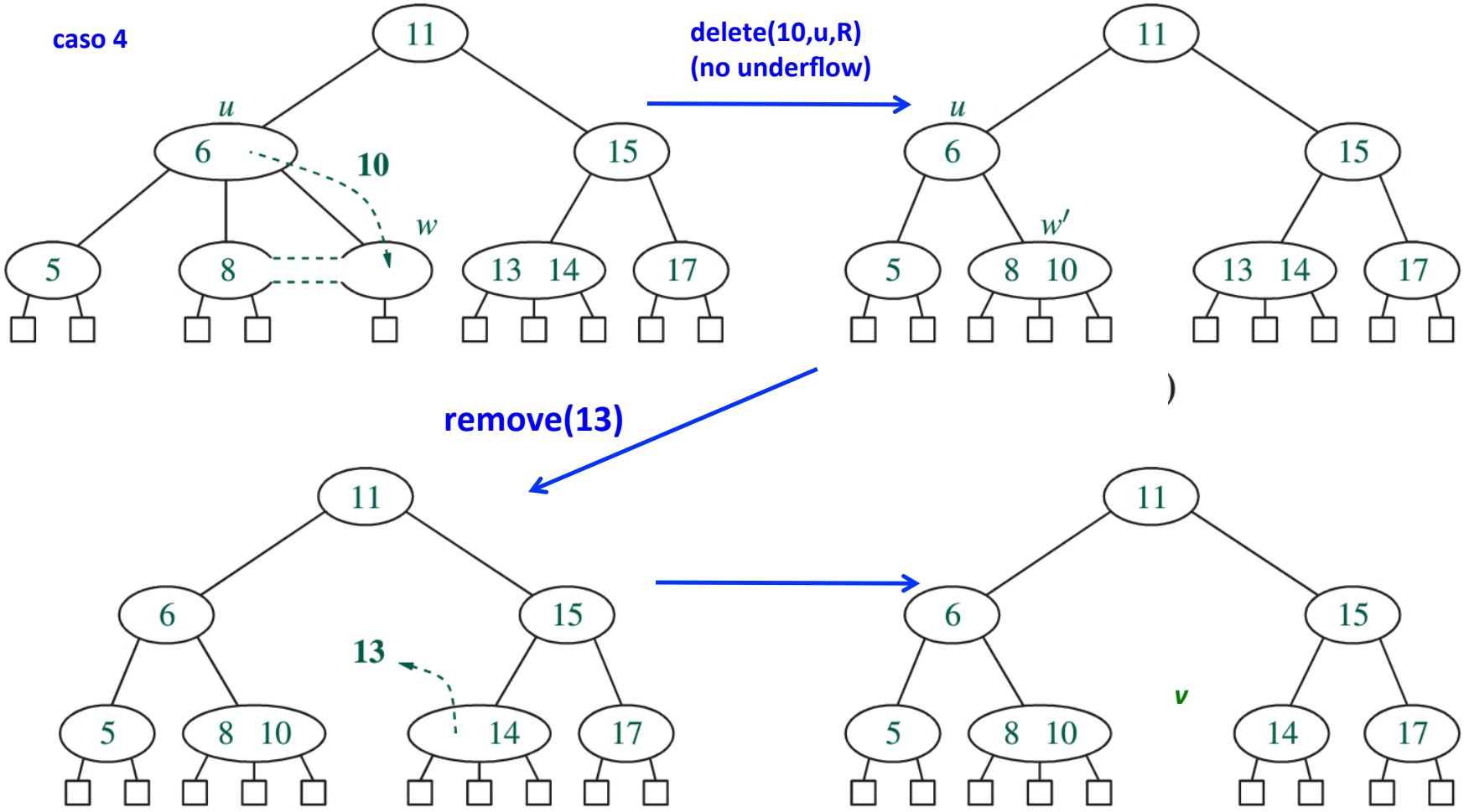


`Delete(`e'`,`v`,`L`);`
return;

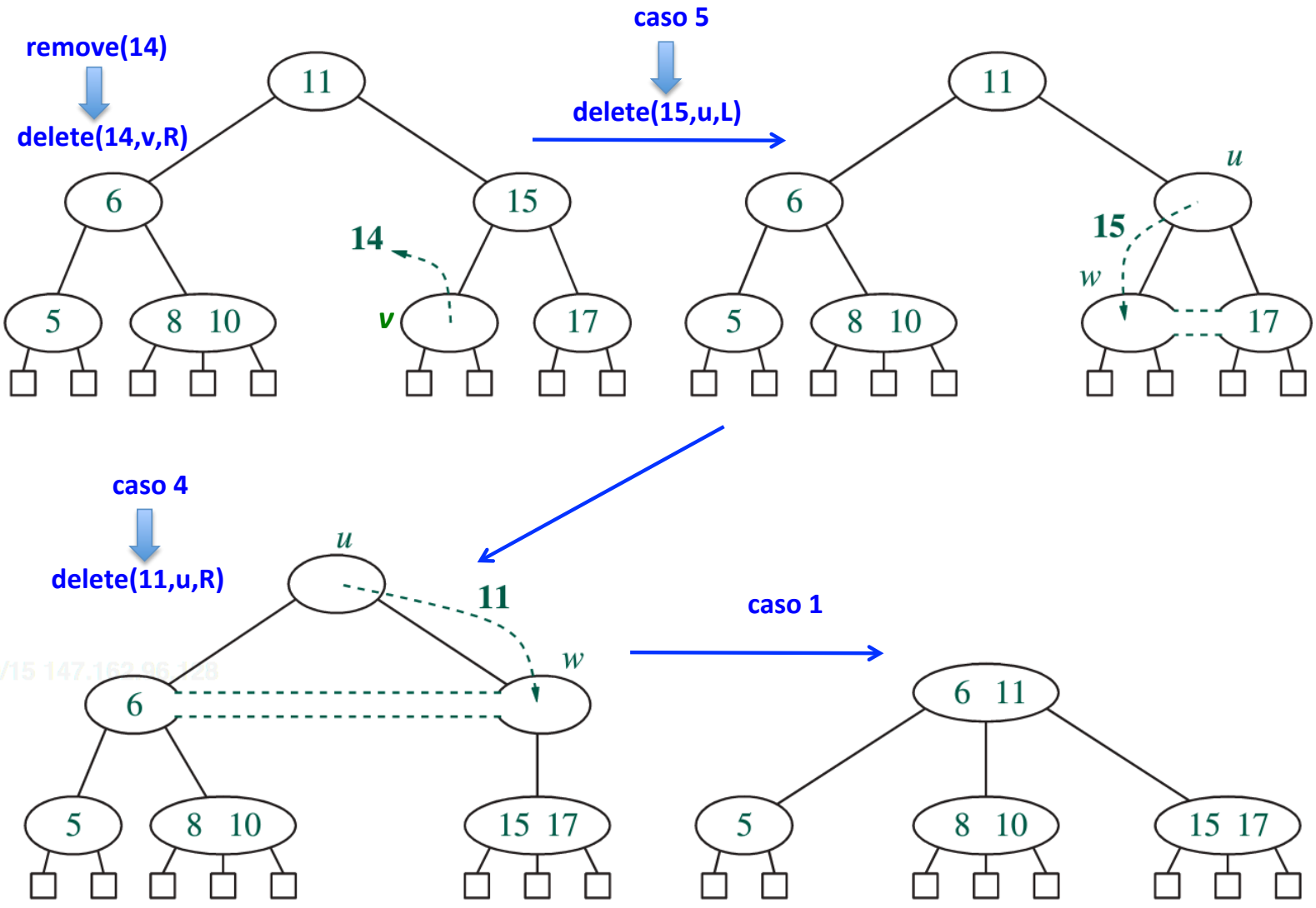
Esempio


remove(4)
delete(4,w,R)
(underflow)
caso 3
remove(12)
delete(11,v,R)
(underflow)
u
w
s
v

Esempio (continua)



Esempio (continua)



Proposizione

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

Dimostrazione

La complessità è dominata dalla esecuzione di `MWTreeSearch` e dalla eventuale esecuzione di `Delete`. La complessità di `MWTreeSearch` è $\Theta(\log n)$, come provato in precedenza. `Delete` viene invocato (ricorsivamente) sui nodi di un cammino che da un nodo ad altezza 1 risale verso la radice. Su ciascun nodo esegue $O(1)$ operazioni oltre a un eventuale chiamata ricorsiva sul padre. Anche in questo caso si deduce immediatamente che la complessità di `Delete` è al più proporzionale all'altezza $h \in \Theta(\log n)$.

Osservazione: L'albero decresce in altezza dalla radice.

Multimap o Dizionario (Paragrafo 10.5.3 [GTG14])

Ammette la presenza di > 1 entry con la stessa chiave

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: yes se si è rimossa la entry e no altrimenti.

Implementazione: tramite una Mappa in cui le entry sono costituite da coppie (k, L_k), dove k è una chiave, e L_k una collezione, 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 $(k, v_1), (k, v_2), \ldots, (k, v_\ell)$.

Multimap o Dizionario (continua)

Implementazione dei metodi. Si modificano i corrispondenti metodi della Mappa come segue:

- `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): (modifica del metodo `remove`(k)) Se esiste una entry (k, L_k) e L_k contiene solo il valore v, si rimuove la entry (k, L_k) dalla mappa, altrimenti si rimuove (un'occorrenza di) v da L_k.

Complessità. Implementando L_k come doubly-linked list, i metodi `get` e `put` hanno la stessa complessità di quelli della Mappa, mentre per `remove` si deve aggiungere un termine additivo s pari alla massima lunghezza di una lista $|L_k|$ (massimo su tutte le entry presenti nella mappa), per tenere conto della ricerca del valore in tale lista L_k.

Riepilogo complessità per ABR e $(2,4)$-Tree

Mappa:

Metodo	ABR	(2,4)-Tree
`get(`k`)`	$\Theta(h)$	$\Theta(\log n)$
`put(`k, v`)`	$\Theta(h)$	$\Theta(\log n)$
`remove(`k`)`	$\Theta(h)$	$\Theta(\log n)$

MultiMap:

Metodo	ABR	(2,4)-Tree
`get(`k`)`	$\Theta(h)$	$\Theta(\log n)$
`put(`k, v`)`	$\Theta(h)$	$\Theta(\log n)$
`remove(`k, v`)`	$\Theta(s+h)$	$\Theta(s+\log n)$

Osservazione: Si può implementare una MultiMap anche senza il ricorso a strutture aggiuntive, ma cambiando la definizione di Albero Binario di Ricerca e $(2,4)$-Tree in modo da permettere la presenza di entry diverse con la stessa chiave nei nodi dell'albero.

Esercizio

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

Svolgimento

L'algoritmo è il seguente:

Algoritmo `24height`(T)
Input: (2,4)-Tree T
Output: altezza di T
$h \longleftarrow 0$; $v \longleftarrow T$.root();
while T.isInternal(v) **do**
    $v \longleftarrow$ un qualsiasi figlio di v;
    $h \longleftarrow h + 1$
return h

La complessità è proporzionale al numero di iterazioni del **while** che è al più l'altezza di T, ovvero $O(\log n)$ se T ha n entry.

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.

Svolgimento

1. È sufficiente creare una nuova radice contenente la entry con chiave massima ($e_{\max}$) di T, rendere T e U figli sinistro e destro della nuova radice, rimuovendo $e_{\max}$ dal nodo originale. Lo pseudocodice è il seguente.

Algoritmo `24TreeMergeEq`(T, U)
Input: (2,4)-Tree T, U di altezza h, max chiave in $T <$ min chiave in U
Output: (2,4)-Tree TU fusione di T e U
$v \longleftarrow T$.root() ;
$z \longleftarrow$ figlio più a destra di v ;
while T.isInternal(z) **do**
    $v \longleftarrow z$;
    $z \longleftarrow$ figlio più a destra di v
$e_{\max} \longleftarrow$ entry con chiave max in v ;
Crea un nuovo nodo r *contenente* $e_{\max}$;
Crea un (2,4)-tree TU *con radice* r *e collega* T *e* U *come sottoalberi sinistro e destro di* r ;
TU.`Delete`($e_{\max}, v, R$) ;
return TU

Complessità: $\Theta(h)$ in quanto il ciclo **while** esegue $\Theta(h)$ iterazioni, ciascuna delle quali richiede $\Theta(1)$ operazioni, e al di fuori del ciclo si eseguono $\Theta(1)$ operazioni e una invocazione del metodo `Delete` che, come visto a lezione, ha complessità $\Theta(h)$.

2. Il (2,4)-tree risultante ha altezza h, se l'underflow a seguito della rimozione di $e_{\max}$ si propaga sino alla radice, altrimenti ha altezza $h+1$. Nel secondo caso, la radice è un 2-node.

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 `24TreeMergeEq`), e sostituirlo a tale sottoalbero.

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 che conti quante entry in T hanno chiave $\leq k$, e analizzarne la complessità.

Svolgimento

L'idea è di adattare `MWTreeSearch`. Sviluppiamo un algoritmo `24CountLE(`v`,`k`)` che invocato su un nodo $v \in T$ restituisce il numero di entry in T_v con chiave $\leq k$. Basterà poi invocarlo con $v = T$.root().

Algoritmo `24CountLE(`v`,`k`)`
Input: nodo $v \in T$, chiave k
Output: numero di entry in T_v con chiave $\leq k$

if (`T.isExternal(v)`) **then return** 0;
Siano (k_1,x_1),..., (k_{d-1},x_{d-1}) *le entry in* v, *con* $k_1 < \cdots < k_{d-1}$;
Siano v_1, v_2, $\cdots$, v_d *i figli di* v;
$j \longleftarrow$ massimo indice tale che $k_j \leq k$; /* $j = 0$ se $k_1 > k$ */
$C \longleftarrow 0$;
for $i \longleftarrow 1$ to j **do** $C \longleftarrow C + v_i.\mathrm{size} + 1$;
if $(k_j < k)$ **then** $C \longleftarrow C$+24CountLE(v_{j+1}, k);
return C

L'analisi è del tutto analoga a quella di `MWTreeSearch` e mostra che `24CountLE` ha complessità $O(\log n)$ quando invocato su un (sotto)albero con n entry.

Esercizio

Risolvere l'esercizio precedente tramite un algoritmo non ricorsivo.

Esercizio

Analizzare l'algoritmo `24CountLE` assumendo che al posto della variabile v.size si invochi un metodo T.size(v) che restituisce sempre il numero di entry in T_v ma abbia una complessità proporzionale al numero restituito.

Riepilogo

- Mappa e Dizionario: 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

Riepilogo (continua)

- (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
- Implementazione di un Dizionario tramite una Mappa

Esempio domande prima parte

- Una funzione hash è caratterizzata da due componenti: un hash code e una compression function. Descrivere il ruolo di ciascun componente.
- Descrivere brevemente l'algoritmo usato per rimuovere una entry e da un albero binario di ricerca T, indicandone la complessità. (Si assuma che il nodo v contenente la entry e sia stato già trovato.)
- Definire un *Multi-Way Search Tree*
- Dimostrare che un Multi-Way Search Tree contenente n entry ha $n + 1$ foglie.
- Nell'inserimento di una nuova entry in un (2,4)-Tree un nodo v può andare in *overflow*
  - Cosa si intende per overflow di v?
  - Descrivere brevemente la procedura Split(v) usata per sanare la condizione di overflow analizzandone la complessità.

Errata

Cambiamenti rispetto alla prima versione dei lucidi:

- Lucido 85: nel caso in cui e' si mette al posto di e in w è stato chiarito che non si tratta di uno swap ma di mettere una copia di e' al posto di e.
- Lucido 98: aggiustato la specifica di `get` e aggiunto la definizione di s.
- Lucido 106: corretta la specifica input-output dell'algoritmo.