Dati e Algoritmi: A. Pietracaprina

Mappe

(Parte 1)

Descrizione generale e applicazioni

Una *Mappa* è una collezione di entry che permette di ricercare, inserire e rimuovere entry in base alle loro chiavi (come un *indice*).

APPLICAZIONI:

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

Definizione e interfaccia

Definizione di Mappa

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();
}
```

- `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.

Esercizio

Sia D un documento di N parole, rappresentato da un array $D[0], D[1], \ldots, D[N-1]$. Progettare un algoritmo che, usando una Mappa d'appoggio, restituisca la parola con il massimo numero di occorrenze in D. Se ne esistono più di una, l'algoritmo ne restituisce una arbitraria tra esse.

L'uso della Mappa deve essere fatto tramite i metodi dell'interfaccia `Map`. *L'analisi della complessità sarà effettuata in un altro esercizio, dopo aver studiato le possibili implementazioni della Mappa.*

Osservazione

Contare le frequenze delle parole è una primitiva importante per l'analisi statistica di documenti, utilizzata ad esempio per classificare i documenti in categorie.

Implementazioni semplici ma inefficienti della Mappa

Lista non ordinata

n = # entry nella Mappa

Complessità di get, put, remove: $\Theta(n)$

Giustificazione: tutte e 3 i metodi richiedono la ricerca di una entry con la chiave data, e se una tale entry non c'è, devono guardarle tutte

$\Rightarrow$ INEFFICIENTE in TEMPO

Implementazioni semplici ma inefficienti della Mappa

Array di taglia $|U|$

Si assume che U sia finito e che sia disponibile un mapping 1-1 tra U e gli indici in $[0, |U|-1]$

Un tale mapping potrebbe non essere banale da trovare

Complessità di get, put, remove : $\Theta(1)$

$\Rightarrow$ INEFFICIENTE in SPAZIO se $|U|$ è molto più grande del numero di entry nella mappa

ESEMPIO: Mappa per tutti i cittadini Italiani $n = 6 \cdot 10^7$

chiave: CF = stringa di 16 caratteri alfanumerici

provenienti da un alfabeto di 36 caratteri

$\Rightarrow |U| = 36^{16} \approx 8 \cdot 10^{24}$

Il mapping tra U e gli indici va $[0, |U|-1]$

Si può ottenere vedendo un CF come un intero

rappresentato in base 36, usando un semplice

mapping tra lettere/cifre e interi in $[0, 35]$

Osservazioni

- **Universo delle chiavi:** non necessariamente ordinato. Se ordinato, si parla di Mappa ordinata (Sorted Map in Inglese) e, in questo caso, sono possibili implementazioni più efficienti al caso pessimo.
- **Generalizzazione per chiavi replicate:** la Mappa assume che le chiavi siano tutte distinte. Una variante della Mappa, chiamata Multimap, permette di avere più entry con la stessa chiave.

Noi studieremo **implementazioni basate su:**

- Tabelle hash (Mappa)
- Alberi di ricerca (binari e non) (Mappa ordinata)

e discuteremo come implementare una Multimap.

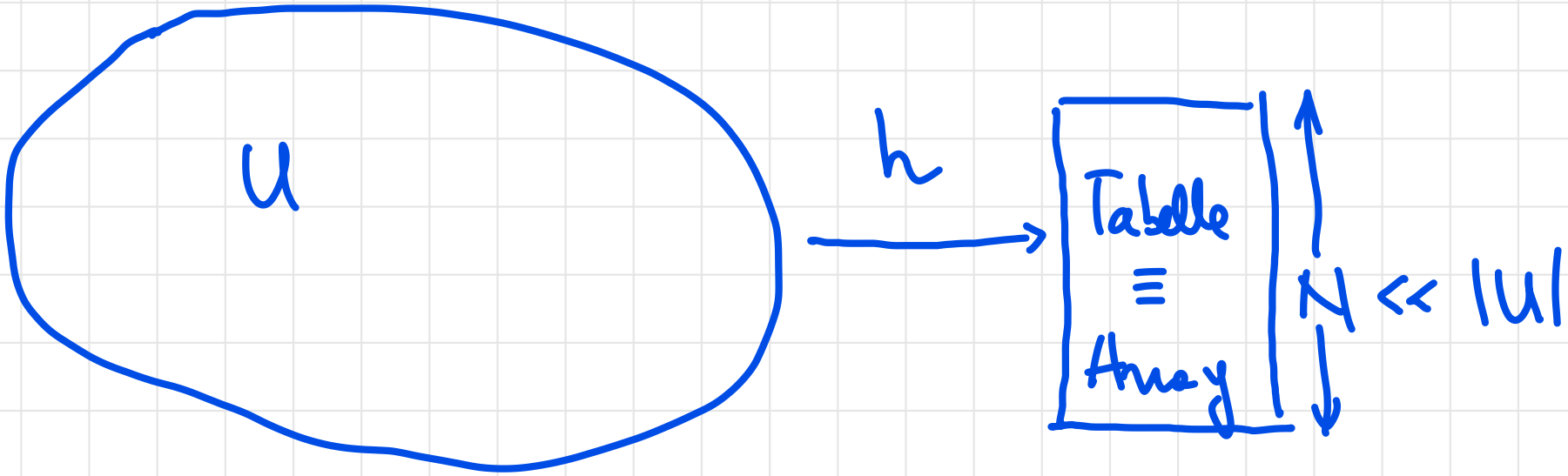
Mappe in Java

Nel package `java.util` troviamo

- Interfaccia `Map<K,V>`: contiene i metodi visti prima. Al suo interno è definita una interfaccia *statica* `Map.Entry<K,V>`:
  - `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.)
  - `Map.Entry` può essere usata ovunque l'interfaccia `Map` sia visibile.
- classe `HashMap<K,V>`: implementazione di una Mappa tramite tabella hash con separate chaining.
- classe `TreeMap<K,V>`: implementazione di una Mappa tramite Red-Black Tree.

Implementazione della Mappa tramite Tabella Hash

Tabelle Hash



OBIETTIVO: Ottenere prestazioni in tempo simili a quelle della soluzione tramite array di taglia $|U|$ ma garantendo efficienza in spazio

In particolare, vogliamo un mapping (funzione h) da U a un array di taglia $N \ll |U|$ tale che un qualsiasi insieme di n entry, ovvero n chiavi di U, siano distribuite bene da h tra gli indici $[0, N-1]$ possibilmente usando un valore N non troppo più grande di n. In questo modo si punta ad avere accesso a una qualsiasi delle n entry in tempo "quasi" costante, usando uno spazio non troppo maggiore di n

Tabelle Hash

Una Tabella Hash è definita dai seguenti 3 ingredienti principali:

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

  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

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

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

- h deve essere veloce da calcolare
- 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 $\forall i, j \in [0, N-1]$:

  (a) $\Pr[h(k) = i] = \frac{1}{N}$

  (b) $\Pr[h(k) = i | h(k') = j] = \Pr[h(k) = i] = \frac{1}{N}$

(a) la probabilità che k sia assegnata al bucket A[i] è la stessa per ogni i

⇒ NON CI SONO BUCKET "PRIVILEGIATI" che h

(b) Sapere che k' è mappata sul bucket A[j], non cambia la probabilità che k sia mappata sul bucket A[i]

⇒ h OFFUSCA POSSIBILI CORRELAZIONI TRA CHIAVI

Oss. Le prestazioni di una tabella hash sono tanto migliori quanto più la funzione h assomiglia a una funzione random che soddisfa (a) e (b)

Metodo HashCode in Java

- metodo `hashCode()` della classe `Object` restituisce un `int` che dipende dall'indirizzo 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

Vediamo come trasformare in Java tipi numerici in `int` ($\mathbb{Z} \equiv$ `int`):

- `byte`, `short`, `char`, `int` $k \rightarrow$ `(int)` k
- `float` (32 bit) $k \rightarrow$ `Float.floatToIntBits(`k`)`
- `long` (64 bit) $k \rightarrow$ `(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` (64 bit) $k \rightarrow$ `Double.doubleToLongBits(`k`)` $\rightarrow$ `int`
  - `Double.doubleToLongBits(`k`)`: di tipo `long`
  - Trasformazione a `int` come per `long`

Hash code per una stringa di char $S \equiv s_0 s_1 \ldots s_{k-1}$

- **Hash code banale:** $h(S) = \sum_{i=0}^{k-1} s_i$

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

- **Polynomial hash code:** $h(S) = \sum_{i=0}^{k-1} s_i \cdot a^{k-1-i}$

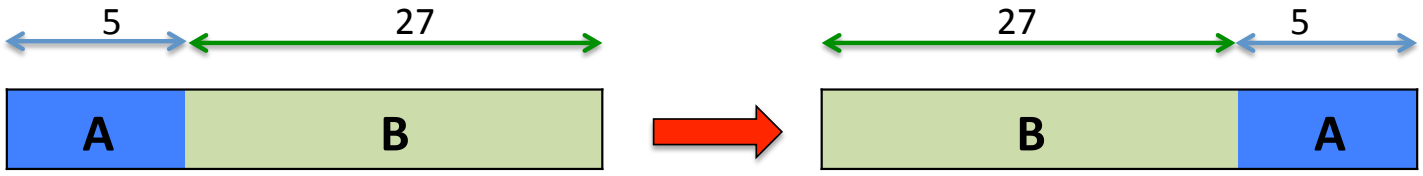
  Oss. Per parole inglesi vanno bene $a = 31, 33, 37, 39, 41$. La classe `String` in Java implementa hashCode in questo modo con $a = 31$

- **Hash code basato su cyclic shift**: si sommano i singoli caratteri applicando dopo ogni addizione un cyclic shift alla somma parziale

  Oss. In pratica, uno shift di 5 posizioni va bene

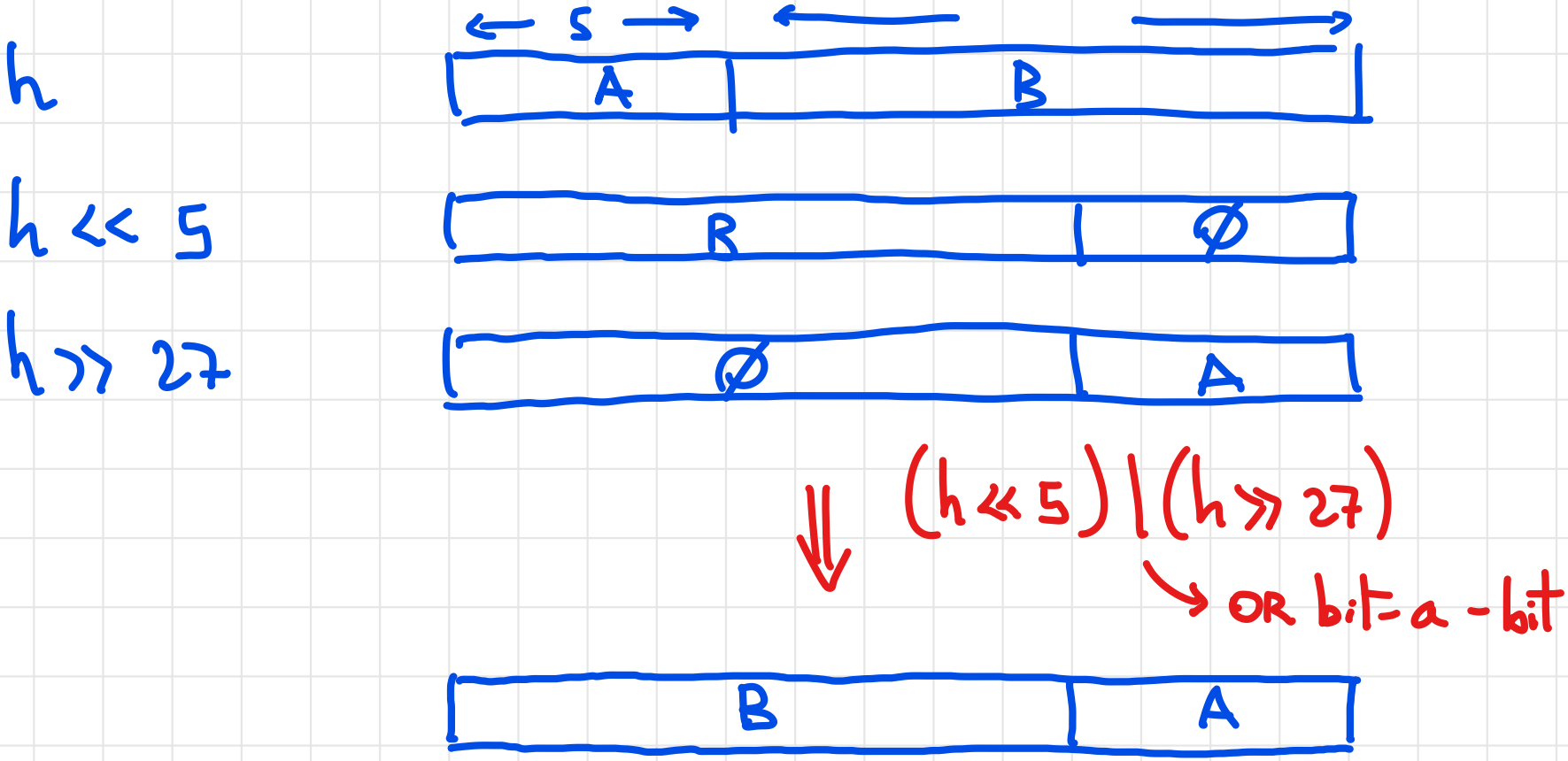
Hash code basato su cyclic shift

Cyclic shift di 5 posizioni a sx



Calcolo dell'hash code:

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


h
5
A
B
h << 5
B
∅
h >> 27
∅
A
(h << 5) | (h >> 27)
OR bit-a-bit
B
A

Compression Function

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

$$i \to i \mod N$$

dove N = capacità del Bucket Array

Osservazione. Per una migliore distribuzione degli hash code tra gli indici del Bucket Array conviene scegliere N *primo e distante da una potenza di 2*.

Scelte di N poco adeguate:

- $N = 2^p \Rightarrow i \mod N$ 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 base 10.

Compression Function

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

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

dove

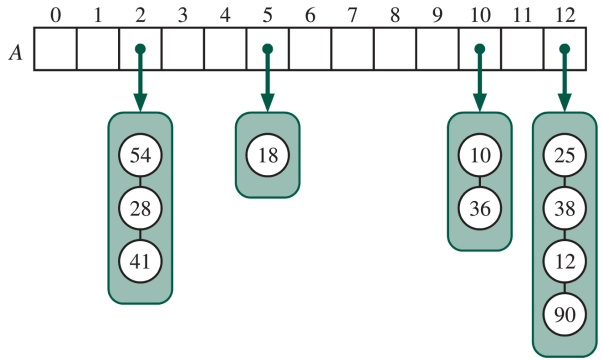
- $p > N$ con p primo
- $a, b \in [0, p-1]$ scelti a caso, $a > 0$

Osservazione: il metodo MAD, leggeremente più costoso dal punto di vista computazionale, assicura una migliore distribuzione degli hash code tra gli indici del Bucket Array.

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



Open Addressing: non fa ricorso a strutture ausiliarie ma memorizza le entry direttamente nelle celle del Bucket Array. In questo modo si risparmia spazio ma si complica la gestione delle collisioni. Noi non studiamo questo approccio.

Implementazione dei metodi della Mappa

Si consideri l'implementazione di una Mappa tramite una tabella hash (A, h) con separate chaining.

- **Metodo** `get(`k`)`

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

- **Metodo** `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 della Mappa

- **Metodo** `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;`

Esercizio R-10.5 [GTG14]

Creare una tabella hash di capacità $N = 11$ con le $n = 11$ chiavi:

$$12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5$$

Funzione hash: $h(k) = (3k + 5) \bmod 11$

Indice i	Bucket $A[i]$
0	13
1	94 39
2	
3	
4	
5	44 88 11
6	
7	
8	12 23
9	16 5
10	20

Load Factor

Definizione: Load Factor

Per una tabella hash di capacità N che memorizza n entry il *load factor* λ è definito come:

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

In altre parole, λ rappresenta la lunghezza media di un bucket

Osservazione: il load factor ha un impatto significativo sulla efficienza computazionale di una tabella hash.

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, e si assuma che per una data chiave k il valore $h(k)$ sia calcolabile in tempo costante.

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.

- $O(n)$: banale
- $\Omega(n)$: motivata dalla seguente istanza:
  * Tutte le entry sono nello stesso bucket in cui viene mappata la chiave k
  * k non è presente nella tabella e quindi si devono guardare tutte le n entry nel bucket $A[h(k)]$

Esercizio (R-10.9 [GTG14])

Stimare asintoticamente il numero di operazioni richiesto al caso migliore (best-case) e al caso pessimo (worst-case) per l'inserimento di n entry in una Mappa implementata tramite Tabella Hash con separate chaining, inizialmente vuota. (Si assuma $N > n$.)

chiavi distinte

BEST CASE: le n entry sono mappate su bucket distinti ⇒ ogni inserimento richiede $O(1)$ operazioni ⇒ $O(n)$ operazioni per tutti gli inserimenti

WORST CASE: tutte le entry sono mappate nello stesso bucket, quindi l'i-esimo inserimento

richiede $\Theta(i)$ operazioni, e tutti gli n inserimenti richiedono un numero aggregato di operazioni:

$$\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$$

Complessità al caso medio

Teorema

Sotto l'ipotesi di uniform hashing, in una tabella hash con separate chaining e load factor λ la complessità al caso medio di `get`, `put`, `remove` è $O(1+\lambda)$. Tale complessità vale per:

(A) qualsiasi chiave k non presente: la media è fatta su tutti i possibili valori di $h(k)$, che sono equiprobabili sotto l'ipotesi di uniform hashing.

(B) chiave k presente: la media è fatta assumendo k scelta a caso, con probabilità uniforme, tra le chiavi presenti nella tabella.

Corollario

Quando $\lambda \in O(1)$, i tre metodi hanno complessità $O(1)$ al caso medio

Dimostrazione di (A): k non presente

n_i = # entry presenti nel bucket di indice i, $0 \le i < N$

$$\Rightarrow \sum_{i=0}^{N-1} n_i = n$$

la complessità dei 3 metodi è dominata da quella richiesta per la ricerca di k che è

$$O\left(\frac{1}{N} \sum_{i=0}^{N-1} (n_i + 1)\right)$$

* $\frac{1}{N}$ serve per fare la media su tutti i possibili bucket in cui può essere mappata k

* $\Theta(n_i+1)$: # operazioni richieste per cercare k nel bucket i

Si ha che

$$\frac{1}{N}\cdot\sum_{i=0}^{N-1}(n_i+1) = \frac{1}{N}\cdot\sum_{i=0}^{N-1} n_i + \frac{1}{N}\sum_{i=0}^{N-1} 1$$

$$= \frac{n}{N} + 1$$

$$= \lambda + 1 \qquad \square$$

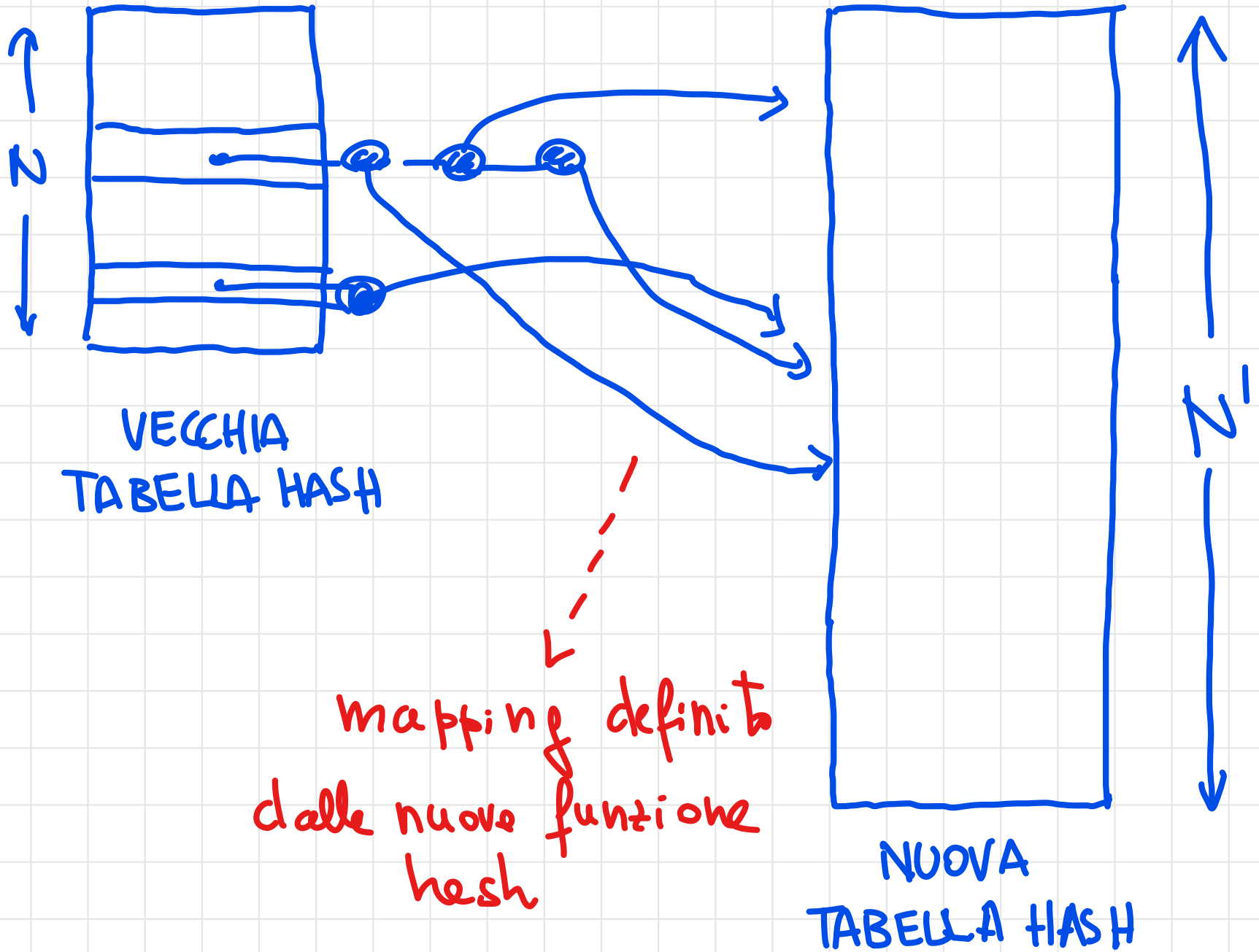
Dimostrazione di (B): la saltiamo

Rehashing

Osservazione. In pratica si suggerisce di mantenere $\lambda < 0.9$. In Java, la classe `HashMap` del package `java.util`, che implementa la tabella hash con separate chaining, usa $\lambda \leq 0.75$ come default)

Quando λ supera la soglia prefissata si esegue un REHASH:

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


N
VECCHIA
TABELLA HASH
mapping definito
dalla nuova funzione
hash
N'
NUOVA
TABELLA HASH

Osservazioni

- In un rehash può essere necessario cercare un numero primo $\geq 2N$, che potrebbe essere costoso (ma noi ignoriamo tale costo).

(1)
- Il trasferimento di n entry da una tabella all'altra può essere implementato in tempo $\Theta(n)$ al caso pessimo.

(2)
- 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 viene *ammortizzato* (ovvero nascosto) da quello aggregato dei `put`.

Giustificazione di (1): per spostare le n entry da una tabella all'altra posto eseguire n inserimenti nella nuova tabella DISABILITANDO il controllo se ogni nuova chiave inserita è già presente, perché so già che le n entry da trasferire hanno tutte chiavi distinte $\Rightarrow$ il trasferimento delle n entry costa $\Theta(n)$ al caso peggiore

Giustificazione di (2):

Analizziamo un generico rehash: $N \to N_{new} \geq 2N$

* $n = \lambda N$ = # entry da trasferire
* CASO 1: quello considerato è il primo rehash eseguito $\Rightarrow$ le n entry da trasferire sono state tutte inserite senza rehash e prima del rehash considerato
* CASO 2: quello considerato non è il primo rehash nella vita della mappa

Consideriamo gli ultimi 2 rehash eseguiti

$$N_{old} \rightarrow N \geq 2N_{old} \rightarrow N_{new} \geq 2N$$

Il primo dei 2 rehash ($N_{old} \rightarrow N$) ha trasferito

$$n_{old} = \lambda N_{old} \text{ entry}$$

Il secondo dei 2 rehash ($N \rightarrow N_{new}$) ha trasferito

$$n = \lambda N \text{ entry}$$

$\Rightarrow$ tra i 2 rehash ci devono essere stati

almeno $n - n_{old}$ inserimenti, e

$$n - n_{old} = \lambda N - \lambda N_{old} \geq \lambda N - \lambda N/2 = \lambda N/2 = n/2$$

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)

Alberi Binari di Ricerca

Definizione

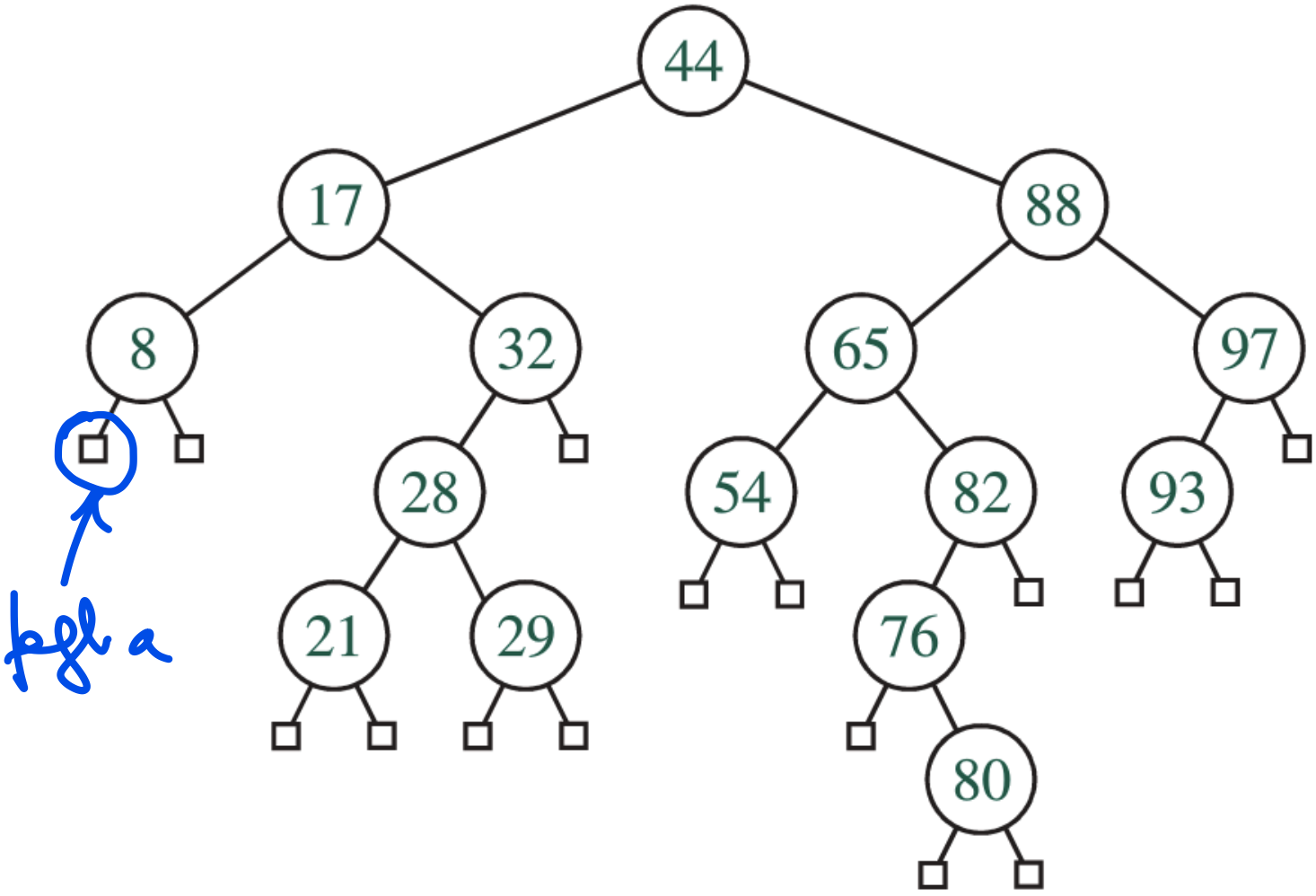
Definizione: Albero Binario di Ricerca

Un Albero Binario di Ricerca è un 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 $> k$

Oss. I nodi foglie sono "sentinelle" che delimitano i confini dell'albero e possono essere implementati con puntatori "null" nei loro fachi.

Esempio (solo chiavi)



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

TreeSearch per un Albero Binario di Ricerca T

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

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

Osservazioni

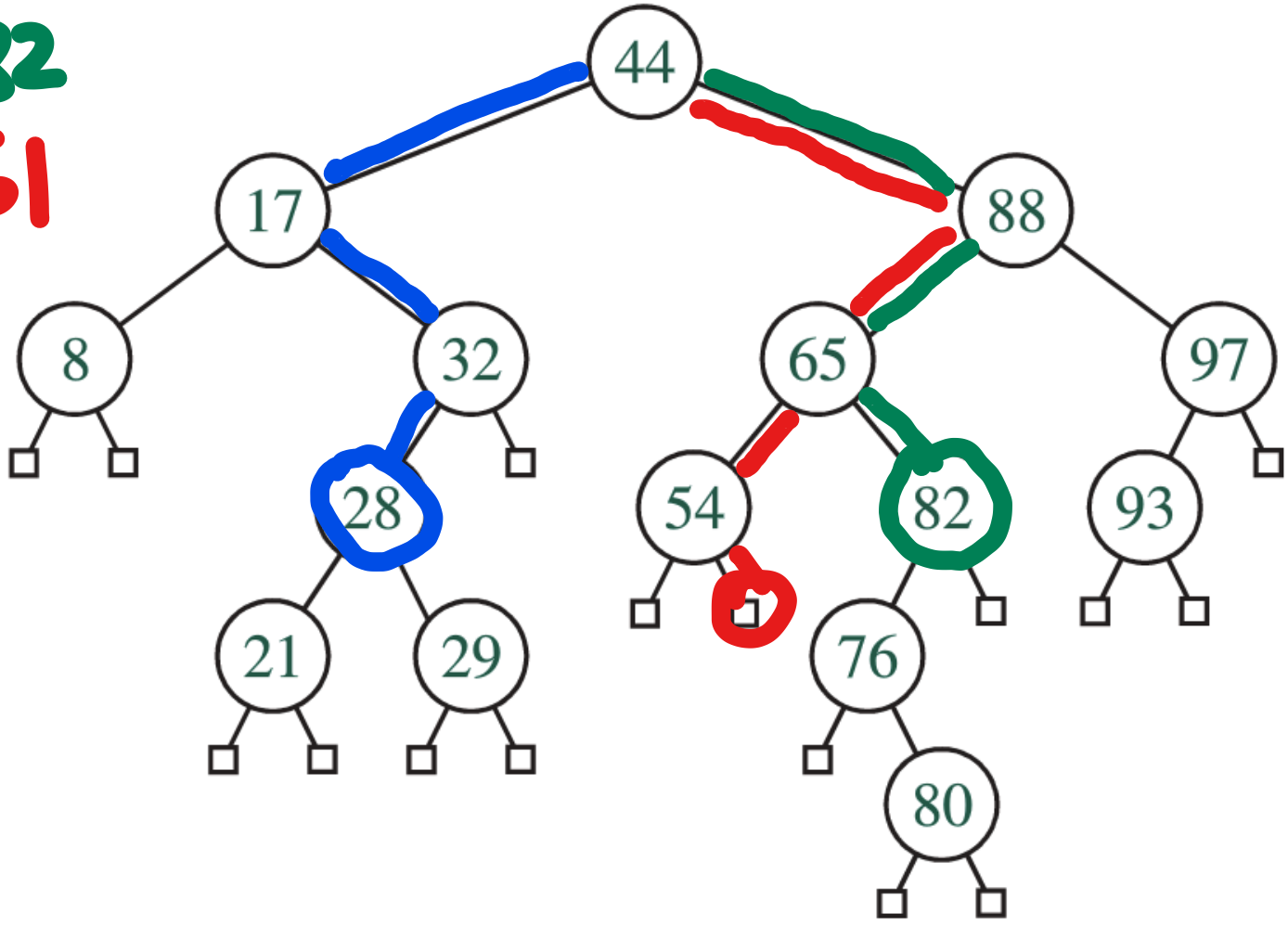
- Una foglia "giusta" per k è una foglia che, se trasformata in nodo interno, può contenere una entry con chiave k senza violare le proprietà di ABR.
- Per ricercare in tutto l'albero si parte con $v = T$`.root()`.
- Correttezza: semplice esercizio.

Esempi

k = 28

K = 82

K = 61



Complessità di TreeSearch

Sia h l'altezza di T. Analizziamo `TreeSearch(k,T.root())` usando l'albero della ricorsione:

- L'albero della ricorsione ha $\leq h+1$ nodi, dato che ciascuna invocazione ricorsiva di `TreeSearch` scende di un livello in T.
- 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 $h=0$)

ATTENZIONE: senza ipotesi sul grado di bilanciamento di T, h può essere $\Theta(n)$

Implementazione dei metodi della Mappa: get

- `get(`k`)`

```
w ← TreeSearch(k, T.root());
if (T.isExternal(w)) then return null;
else return w.getElement().getValue();
```

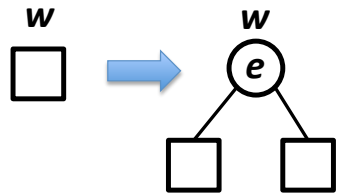
Complessità: $\Theta(h)$ perché dobbiamo usare la TreeSearch

Implementazione dei metodi della Mappa: put

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

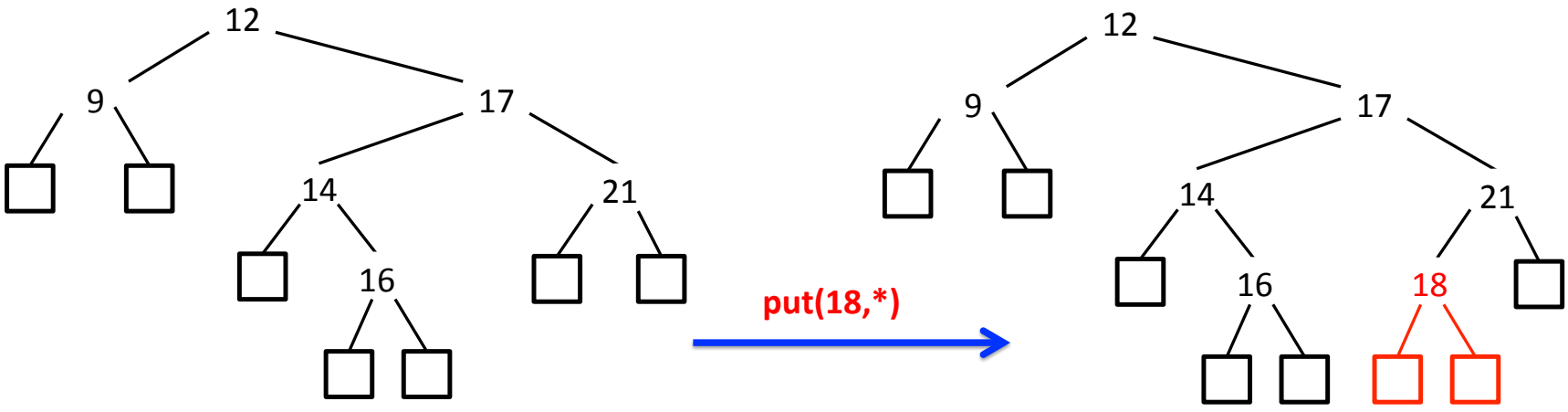
  $w \leftarrow$ `TreeSearch(`k`,` T`.root());`
  if `(`T`.isInternal(`w`))` **then**
  $y \leftarrow$ w.getElement().getValue();
  sostituisci x *a* y *nella entry* w.getElement();
  return y
  `expandExternal(`$w, e = (k, x)$`);`
  incrementa numero di entry di T *di* 1;
  return *null*

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



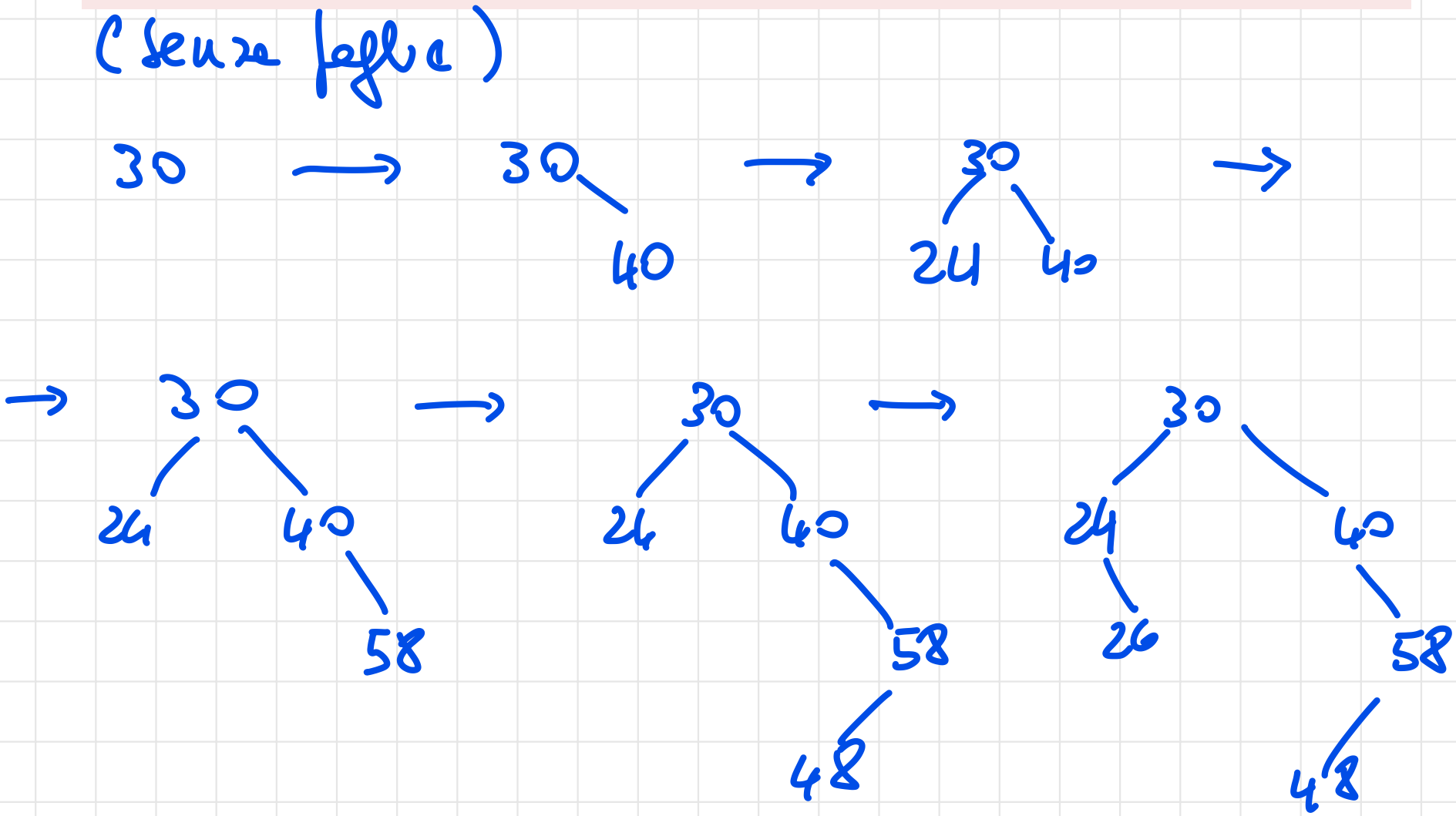
Complessità: $\Theta(h)$ perché dominata dalla TreeSearch

Esempio (solo chiavi)

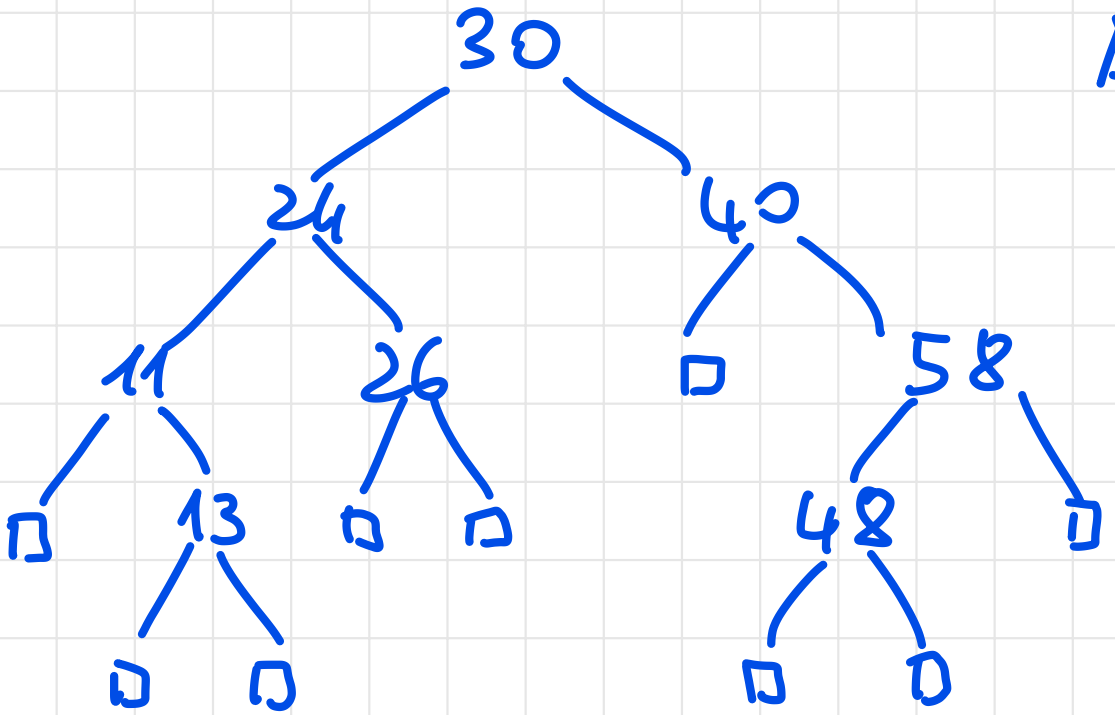


Esercizio R-11.1 [GTG14]

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.



...



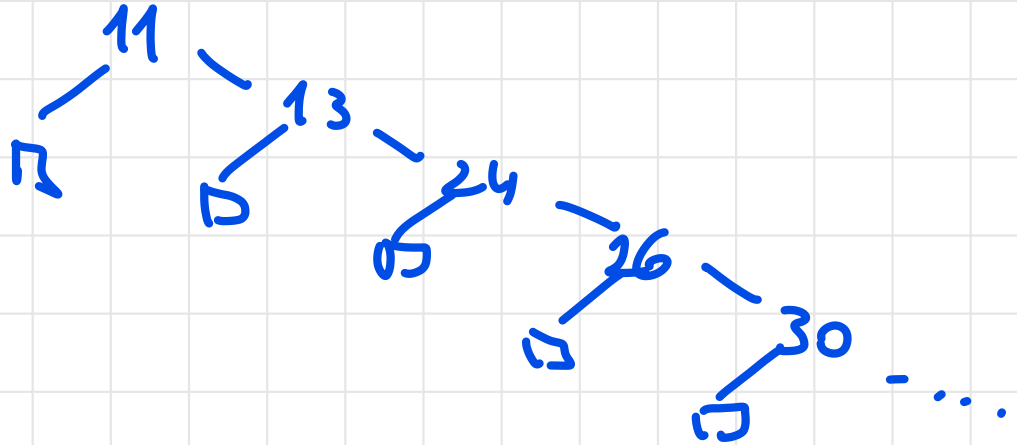
Albero foglie

Esercizio

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

No, basta che la prima entry sia diversa da 30 che la radice dell'albero risultante sarebbe diversa

ad es. 11 13 24 26 30 45 48 58



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
        // CASO 1: w interno con almeno 1 figlio foglia
        esegui Caso 1 (vedi lucidi successivi);
    else
        // CASO 2: w interno con 2 figli interni
        esegui Caso 2 (vedi lucidi successivi);
return value;
```

Caso 1: w interno e padre di almeno 1 foglia

$u_L \leftarrow T$.left(w); $u_R \leftarrow T$.right(w);

if (`T.isExternal(`u_L`)`) **then**

  Cancella w *e* u_L;

  Fai salire u_R *al posto di* w;

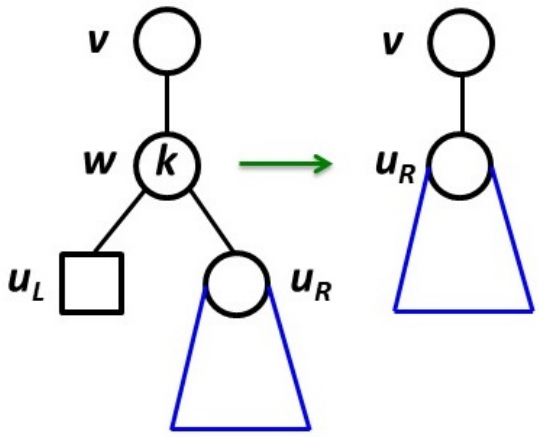
else

  Cancella w *e* u_R;

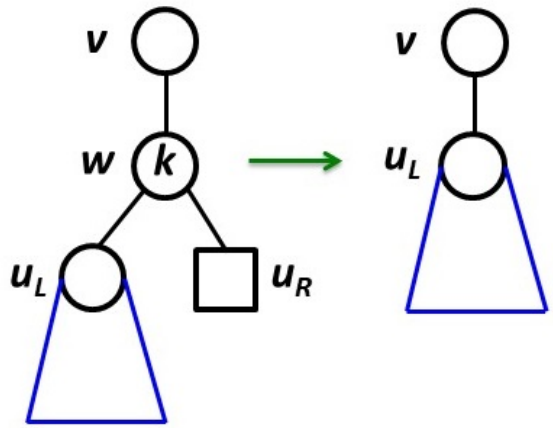
  Fai salire u_L *al posto di* w;

funziona correttamente anche se entrambi i figli di w, u_L e u_R sono foglie

ramo then



ramo else



Oss.: se w era radice $\Rightarrow$ la nuova radice è il figlio messo al suo posto.

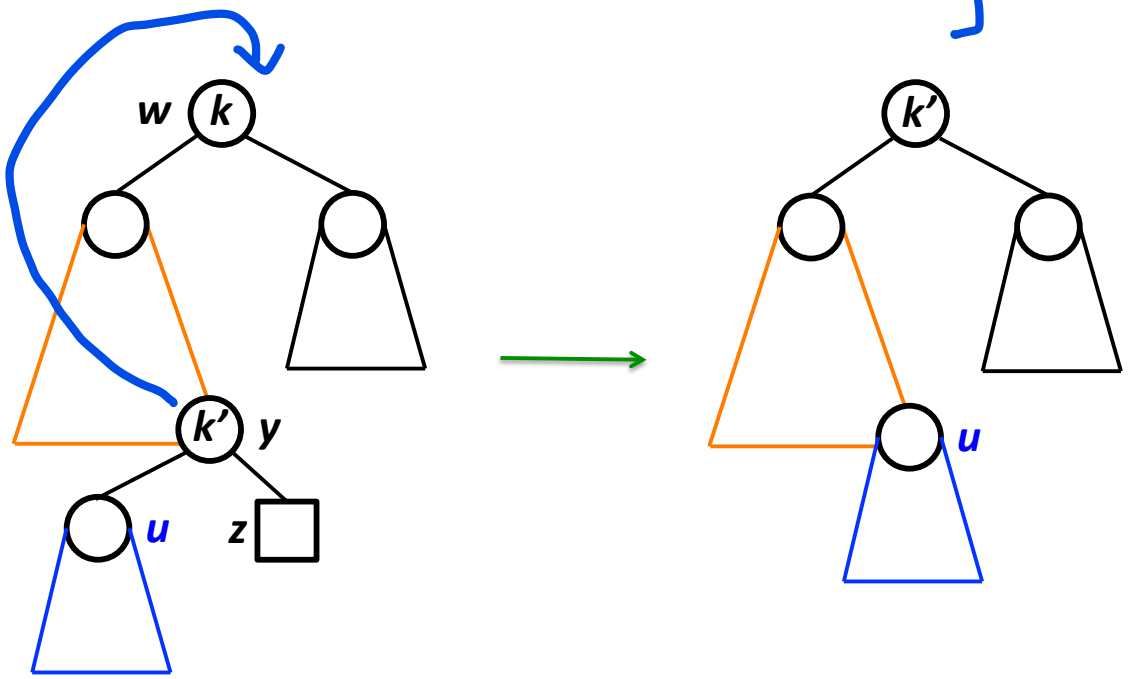
Caso 2: w interno e padre di 2 nodi interni

$y \leftarrow$ *nodo con chiave max nel sottoalbero sinistro di* w;
Cancella y *e il suo figlio destro*;
Fai salire il figlio sinistro di y *al posto di* y;

$\Theta(h)$ operazioni

$\Theta(1)$ operazioni

Sposta le entry in y nel nodo w



Oss:. 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.

Complessità di remove

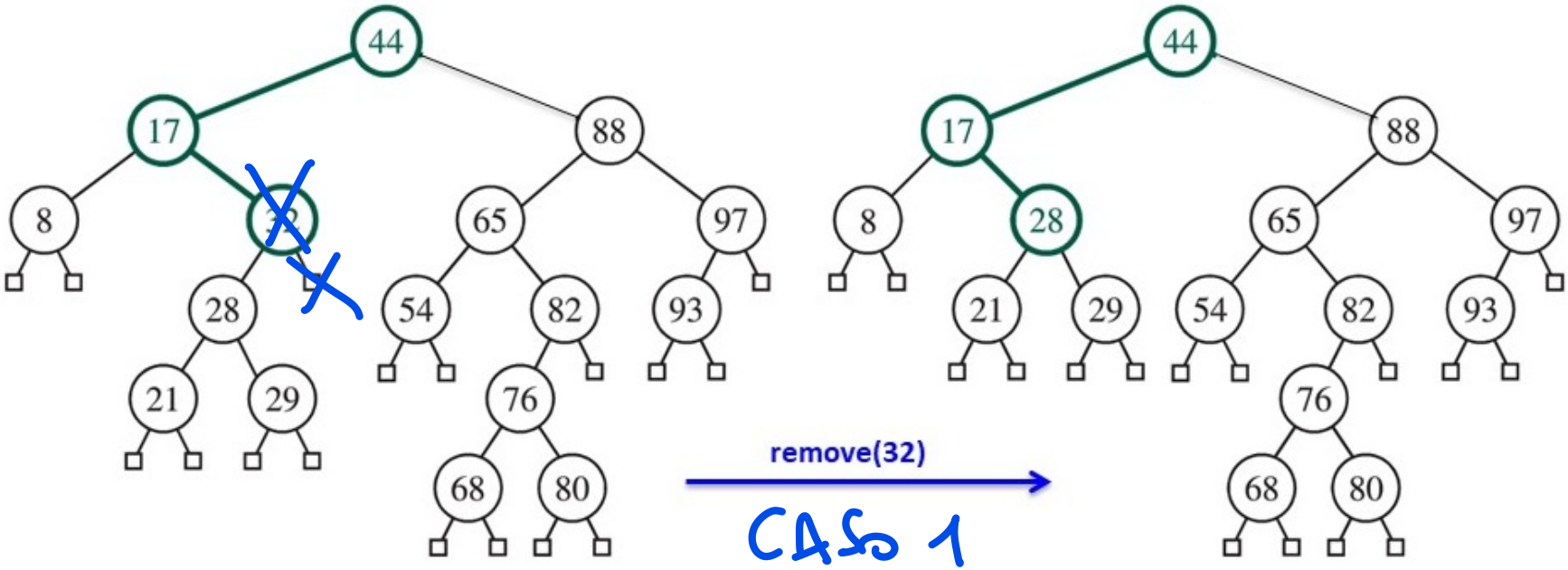
Sia h l'altezza di T. La complessità di remove(k) è ottenuta sommando i seguenti contributi

* TreeSearch: $\Theta(h)$
* (CASO 2): ricerca di y: $\Theta(h)$
* Altre operazioni: $\Theta(1)$

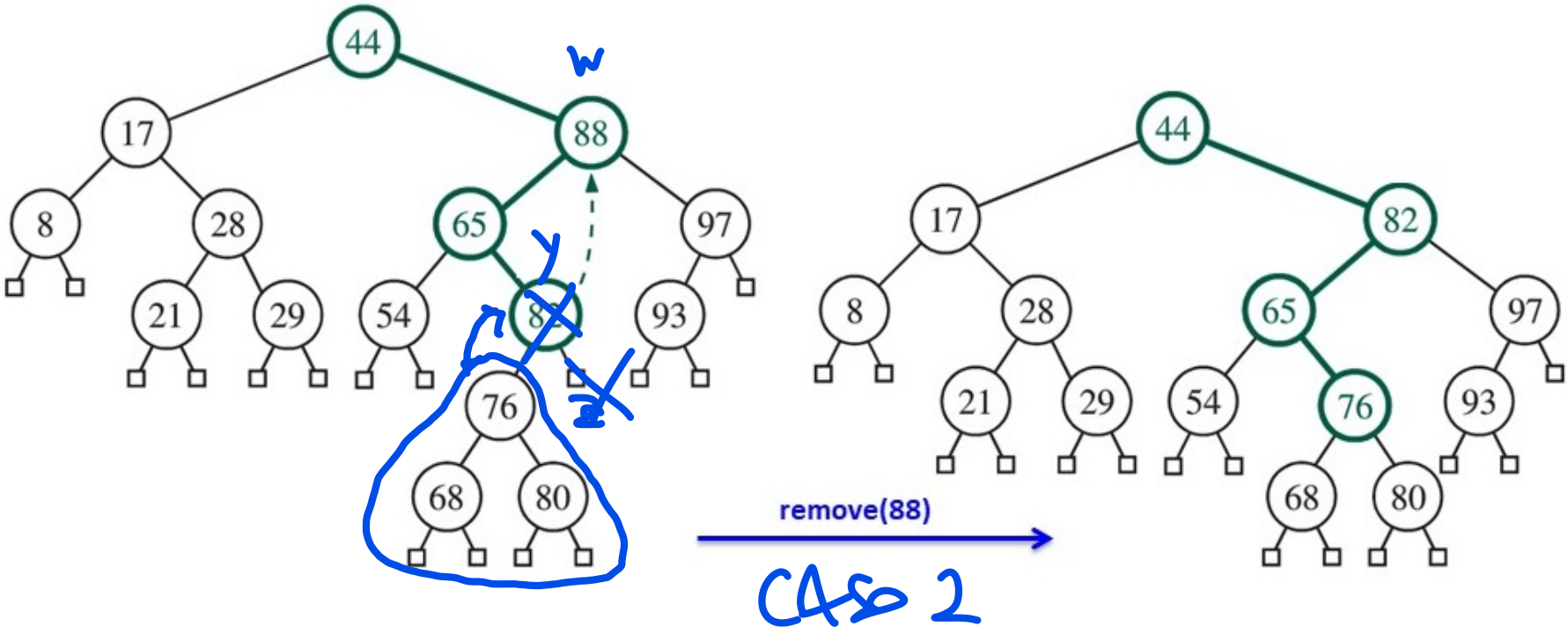
$\Rightarrow$ Complessità: $\Theta(h)$

Esercizio: Scrivere in dettaglio un ciclo per trovare y a partire da w nel CASO 2

Esempio


44
17
88
8
32
65
97
28
54
82
93
21
29
76
68
80
remove(32)
CASO 1
44
17
88
8
28
65
97
21
29
54
82
93
76
68
80

Esempio


44
17
88
8
28
65
97
21
29
54
82
93
76
68
80
w
remove(88)
Caso 2
44
17
82
8
28
65
97
21
29
54
76
93
68
80

Esercizio

Scrivere una versione iterativa di `TreeSearch`.

Algoritmo TreeSearch (T, k)

input ABR T e una chiave k

output nodo w∈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
```

Complessità: $\Theta(h)$

* $\Theta(1)$ operazioni fuori del while
* $\Theta(h)$ iterazioni del while al caso peggiore
* $\Theta(1)$ operazioni in ciascuna iterazione del while.

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.

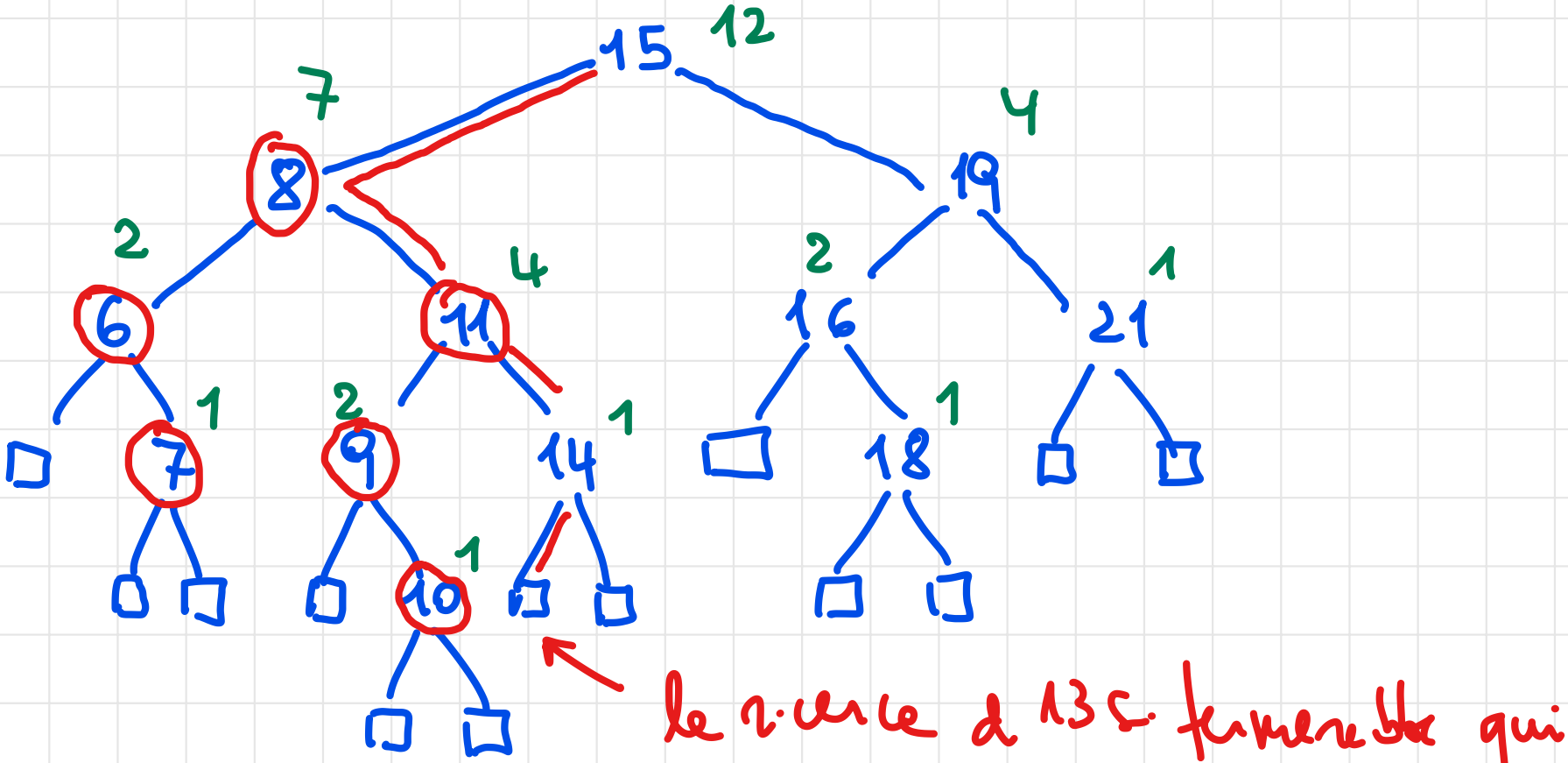
Algoritmo CountLE(k, v)

input chiave k, nodo $v \in T$

Output # entry in T_v con chiave $\leq k$

Prima invocazione: $v = T.root()$

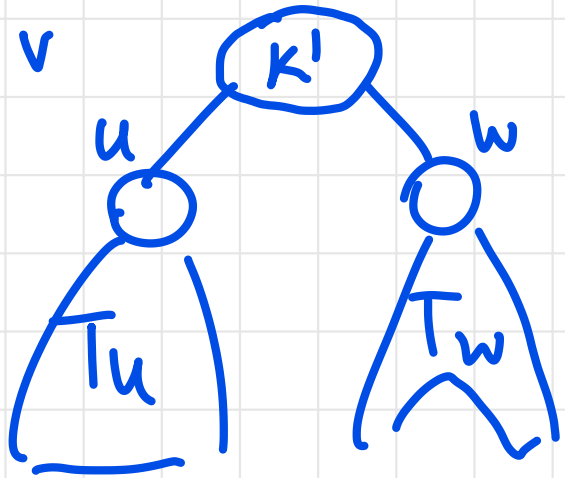
ESEMPIO (variabili size in verde)



la ricerca di 13 si fermerebbe qui

Per K = 13 l'OUTPUT è: 6

IDEA



* Se $k' > k$ ⇒ restituisco CountLE(k, u)
  dato che né k' né le chiavi in T_w
  provo contribuiscono all'output
* Se $k' \leq k$ restituisco 1 + u.size + CountLE(k, w)
  dato che certamente tra k' che tutte

le chiavi in T che contribuiscono all'output

PSEUDOCODICE

```
if (T.isExternal(v)) then return 0 // CASO BASE
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: stessa struttura di TreeSearch

* $\leq h$ invocazioni ricorsive
* $\Theta(1)$ operazioni in ciascuna invocazione ricorsiva (tranne le invocazioni ricorsive al suo interno)

$\Rightarrow$ Complessità $\in \Theta(h)$

h → altezza di T

Esercizio

Si definisca *Interval Tree* un albero binario di ricerca le cui entry sono intervalli $[a, b]$, con $a \leq b$, sulla retta reale. La chiave di una entry $[a, b]$ è rappresentata dall'estremo sinistro a dell'intervallo. Nell'albero non possono esistere due intervalli $[a, b]$ e $[c, d]$ in cui il primo sia interamente contenuto nel secondo, cioè con $c \leq a \leq b \leq d$.

1. Progettare un algoritmo iterativo `InsertCheck` che, dato un intervallo $[a, b]$ e un Interval Tree T, determina se $[a, b]$ può essere inserito in T, ovvero se non esiste in T un intervallo che contiene interamente $[a, b]$ o che è contenuto interamente in $[a, b]$.
2. Analizzare la complessità di `InsertCheck`.

Esercizio

Progettare un algoritmo iterativo che, dato un albero binario di ricerca T contenente entry con chiavi reali *distinte*, 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

Analizzare la complessità dell'algoritmo sviluppato per l'esercizio del Lucido 73 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. (**Suggerimento:** esprimere la complessità come somma di due termini, uno dei quali è il costo aggregato di tutte le invocazioni del metodo `size`.)

Esercizio

Come cambia l'algoritmo sviluppato per l'esercizio del Lucido 73 se non si hanno a disposizione ~~le variabili `size`~~? né le variabili, né il metodo size

Esercizio

Risolvere l'esercizio del Lucido 73 usando un algoritmo iterativo.

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). Si vuole progettare un algoritmo ricorsivo `MinMatStraniero` per determinare *la più piccola matricola di uno studente straniero in* T. Se non ci sono studenti stranieri in T_v l'algoritmo restituisce `null`.

1. Descrivere tramite pseudocodice la generica invocazione di `MinMatStraniero`, specificandone con attenzione l'input e l'output.
2. Analizzare la complessità di `MinMatStraniero`