Appunti di Dati e algoritmi

Giacomo Simonetto

Primo semetre 2024-25

Sommario

Appunti del corso di Dati e algoritmi della facoltà di Ingegneria Informatica dell'Università di Padova.

Indice

1 Introduzione all'algoritmica

1.1 Progettazione di un algoritmo

- specifica del problema computazionale
- progetto dell'algoritmo (pseudocodice)
- analisi dell'efficacia e efficienza dell'algoritmo (correttezza e complessità)
- codifica programma (linguaggio di programmazione)

1.2 Problema computazionale

Un problema computazionale è composto da tre componenti:

- I l'insieme delle istanze (input)
- S l'insieme delle soluzioni (output)
- $\pi \subseteq I \times S$ una relazione che associa ad ogni elemento di I un corrispettivo elemento di S.

Se esistono più soluzioni per una generica istanza, l'algorimto ne calcola una arbitraria

1.3 Algoritmo

Un algoritmo è una procedura computazionale che trasforma un dato input in un output eseguendo una sequenza di passi elementari. Sfrutta il modello di calcolo *RAM* (Random Access Machine) con le seguenti caratteristiche:

- è simile al modello di Von Neumann
- in memoria sono contenuti i dati di input, output e dati intermedi
- la CPU utilizza operazioni elementari per risolvere il problema

Il problema che risolve l'algoritmo è un problema computazionale, ovvero trova le soluzioni abbinate ad una determinata istanza.

Pseudocodice

Per descrivere le azioni di un algoritmo si utilizza lo pseudocodice nella seguente struttura:

0: **Algoritmo:** MyAlgoritm(parametri)
Input: descrizione dell'input
Output: descrizione dell'output

1: descrizione dell'algorimo in pseudocodice
2: ...
3: ...

1.4 Complessità - efficienza di un algoritmo

Taglia di un'istanza

La taglia di un'istanza è l'indice di misura della dimensione dei dati di input, scelto in base all'algoritmo.

Complessità temporale

La complessità temporare indica una stima asintotica del numero di operazioni elementari eseguite dall'algoritmo per una certa istanza. Suggerisce una stima del tempo di esecuzione di un algoritmo per una certa istanza ed è utilizzata per confrontare algoritmi diversi in base alla loro efficienza computazionale.

Caratteristiche

- la complessità si esprime in maniera generale in funzione della taglia dell'istanza
- è indipendendete dall'implementazione e dal linguaggio usato
- non richiede di conoscere il tempo esatto di esecuzione, per cui non richiede l'implementazione dell'algoritmo in un certo linguaggio di programmazione, ma basta lo pseudocodice

Notazione

Per indicare una complessità asintotica, si utilizzano gli ordini di grandezza e le notazioni Ω, Θ, O.

- $f(n) \in O(g(n)) \quad \Rightarrow \quad \exists c > 0, n_0 \geq 1$ t.c. $f(n) \leq c\, g(n) \quad \forall n \geq n_0$
- $f(n) \in \Omega(g(n)) \quad \Rightarrow \quad \exists c > 0, n_0 \geq 1$ t.c. $f(n) \geq c\, g(n) \quad \forall n \geq n_0$
- $f(n) \in \Theta(g(n)) \quad \Rightarrow \quad f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$
- $O(1) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n) \leq \ldots$

Calcolo

Per trovare la complessità asintotica al caso pessimo si effettuano le seguenti analisi:

1. si cerca un upper-bound attraverso l'analisi del caso pessimo (in assoluto) e lo si esprime come $O(f(n))$
2. si cerca un lower-bound attraverso l'analisi di una istanza "cattiva" e lo si esprime come $\Omega(g(n))$
3. se $f(n) = g(n)$ e la complessità dell'algoritmo $\in O(f(n))$ e $\in \Omega(g(n))$, allora si conclude che l'algoritmo ha complessità al caso pessimo $\in \Theta(f(n))$

Osservazioni

- un algoritmo ha complessità ottima se è $O(\log n)$, buona se $O(n)$ o $O(n \log n)$, discreta se $O(n^2)$, pessima se $O(2^n)$ o $O(n^n)$
- alcune volte il valore n_0 è per istanze molto grandi e può capitare che un algoritmo di complessità $\Theta(n^3)$ sia più efficiente di uno di complessità $\Theta(\log n)$
- alcune volte, il caso pessimo si verifica per istanze patologiche che si verificano estremamente raramente, per cui si ricorre all calcolo della complessità al caso medio

1.5 Correttezza e invariante - efficacia di un algoritmo

Criteri di correttezza

Un algoritmo è corretto se:

- termina in un numero finito di passi
- restituisce una soluzione valida per il problema computazionale

Invariante

L'invariante è una proprietà costante espressa in funzione delle variabili dell'algoritmo e descrive lo stato ottenuto in ogni segmento dell'algoritmo stesso. Per avere senso, deve essere funzionale alla correttezza dell'algoritmo. L'invariante è spesso usato per verificare la correttezza di un ciclo o di una serie di chiamate ricorsive: lo si esprime in funzione delle variabili del ciclo o dei parametri della chiamata ricorsiva e si verifica che esso sia soddisfatto alla fine di ogni iterazione o chiamata ricorsiva.

Verifica della correttezza in un ciclo

Per verificare la correttezza di un ciclo si utilizza un processo induttivo:

1. si determina l'invariante per il ciclo
2. si dimostra che l'invariante vale per l'iterazione -1 (ovvero prima dell'inizio del ciclo)
3. si verifica che se vale per una certa iterazione i, allora vale per l'iterazione $i + 1$
4. si verifica che l'invariante alla fine del ciclo combacia con la condizion di correttezza della soluzione

1.6 Algoritmi ricorsivi

Definizione

Un algoritmo si dice ricorsivo se richiama se stesso su istanze di minore dimensione fino ad arrivare ad un caso base in cui la soluzione è immediata.

Esecuzione e stack

Ad ogni nuova chiamata ricorsiva, viene aggiunto un nuovo record di attivazione per tale chiamata nello stack. Nell'RDA sono contenute tutte le variabili e i riferimenti agli oggetti memorizzati nell'heap. Alla chiusura del metodo, si elimina l'RDA di tale metodo e si apre quello sottostante.

Esecuzione e recursion tree

All'esecuzione di un algoritmo ricorsivo si associa un albero della ricorsione con le seguenti caratteristiche:

- ogni nodo corrisponde ad una chiamata ricorsiva
- la radice corrisponde alla prima chiamata ricorsiva
- i figli di un nodo corrispondono alle chiamate ricorsive effettuate nell'esecuzione della chiamata associata al nodo padre
- le foglie corrispondo ai casi base

Differenza tra algoritmo iterativo e algoritmo ricorsivo

- un algoritmo iterativo, per definizione, deve contenere cicli al suo interno
- un algoritmo ricorsivo, per definizione, deve contenere almeno una chiamata a se stesso
- iterativo e ricorsivo sono mutualmente esclusivi: non esiste un algoritmo ricorsivo e iterativo
- un algoritmo ricorsivo può avere o non avere cicli, ma un algoritmo iterativo non ha chiamate ricorsive perché altrimenti sarebbe un algoritmo ricorsivo

Complessità di algoritmi ricorsivi

Per determinare la complessità di un algoritmo ricorsivo si può procedere in vari modi:

- contare il numero di operazioni per livello e moltiplicarlo per il numero di livelli dell'albero della ricorsione, come nel caso di mergesort
- contare tutte le operazioni eseguite ad ogni chiamata ricorsiva e sommarne i valori

Correttezza di algoritmi ricorsivi

Per determinare la correttezza di un algoritmo ricorsivo si procede con una dimostrazione per induzione:

1. si dimostra che l'algoritmo risolve correttamente i casi base
2. si verifica che se l'algoritmo è corretto per una determinata istanza, allora vale anche un'istanza di taglia superiore

2 Alberi

2.1 Introduzione

Definizione di albero

Un albero radicato è una collezione di nodi vuota o con le seguenti proprietà:

- $\exists\, r$ nodo detto radice
- se un generico nodo $n \neq r$, allora deve avere un unico padre
- da ogni nodo, risalendo di padre in padre si arriva ad r

Altre definizioni

- **antenato**:
  un nodo x è antenato di y se $x = y$ o se x è antenato del padre di y
- **discendente**:
  un nodo x è discendente di y se y è antenato di x
- **nodo interno**:
  un nodo x è interno se ha dei figli
- **nodo esterno o foglia**:
  un nodo x è esterno o una foglia se non ha figli
- **sottoalbero con radice v**:
  indicato con T_v è un albero formato dai discendenti di v
- **albero ordinato**:
  un albero si dice ordinato se è presente un ordinamento lineare tra i figli di un nodo
- **albero radicato**:
  un albero si dice radicato se è composto da un nodo radice r che ha come figli, altri alberi radicati nei nodi figli di r
- **profondità di un nodo**:
  la profondità di un nodo è la distanza tra il nodo e la radice
- **altezza di un nodo**:
  l'altezza di un nodo è la distanza tra il nodo e la foglia più distante
- **livello**:
  il livello i di un albero è l'insieme di nodi con profondità i
- **altezza di un albero**:
  l'altezza di un albero è la massima profondità delle foglie

2.2 Algoritmi base: profondità e altezza di un nodo

Profondità di un nodo - ricorsiva

- L'algoritmo calcola la distanza tra il nodo v e la radice, risalendo di padre in padre
- La complessità dell'algoritmo è $\Theta(n)$, più precisamente $\Theta(d_v)$, con d_v profondità del nodo v.

0: **Algoritmo:** DEPTH_RIC(v)
Input: v nodo dell'albero T
Output: profondità di v

1: **if** (T.isRoot(v)) **then**
2: return 0
3: **else**
4: return 1 + depth(T.parent(v))

Profondità di un nodo - iterativa

- L'algoritmo calcola la distanza tra il nodo v e la radice, risalendo di padre in padre
- La complessità dell'algoritmo è $\Theta(n)$, più precisamente $\Theta(d_v)$, con d_v profondità del nodo v.

0: **Algoritmo:** DEPTH_ITER(v)
Input: v nodo dell'albero T
Output: profondità di v

1: $d \leftarrow 0$
2: $u \leftarrow v$
3: **while** (!T.isRoot(u)) **do**
4: $u \leftarrow$ T.parent(u)
5: $d \leftarrow d + 1$
6: return d

Atezza di un nodo

- L'algoritmo calcola prima l'altezza dei discendenti di v e prosegue verso l'alto fino a v
- La complessità dell'algoritmo è $\Theta(n)$, più precisamente $\Theta(n_v)$, con $n_v = \#$ nodi del sottoalbero T_v.

0: **Algoritmo:** HEIGHT(v)
Input: v nodo dell'albero T
Output: altezza di v

1: $h \leftarrow 0$
2: $w \leftarrow v$
3: **for all** w **in** T.children(v) **do**
4: $h \leftarrow \max\{h, 1 + \text{height}(w)\}$
5: return d

2.3 Visite: preorder e postorder

Visita in preorder

- L'algoritmo prevede di visitare prima il nodo v e successivamente i figli
- La complessità dell'algoritmo è $\Theta\left(n + \sum_{v \in T} t_v\right)$, con t_v costo della visita di un nodo

---

0: **Algoritmo:** PREORDER(v)
Input: v nodo dell'albero T
Output: /

1: visita v
2: **for all** w **in** T.children(v) **do**
3: preorder(w)

Visita in postorder

- L'algoritmo prevede di visitare prima i figli e successivamente il nodo v
- La complessità dell'algoritmo è $\Theta\left(n + \sum_{v \in T} t_v\right)$, con t_v costo della visita di un nodo

---

0: **Algoritmo:** POSTORDER(v)
Input: v nodo dell'albero T
Output: /

1: **for all** w **in** T.children(v) **do**
2: postorder(w)
3: visita v

---

3 Alberi binari

3.1 Introduzione

Definzione di albero binario

Un albero binario è un albero ordinato in cui ogni nodo ha al più die figli. Ogni nodo è etichettato come figlio sinistro o figlio destro, in base alla posizione di ordinamento.

Altre definizioni

- **albero binario proprio**:
  un albero binario si dice proprio se ogni nodo ha esattamente due figli
- **albero binario proprio estremo**:
  un albero binario proprio si dice estremo se nessun figlio destro (o sinistro) ha discendenti, ovvero se è molto sbilanciato verso sinistra (o verso destra)
- **albero binario proprio perfettamente bilanciato**:
  un albero binario proprio si dice perfettamente bilanciato se tutte le foglie sono allo stesso livello
- **albero binario completo**:
  un albero binario con i livelli si dice completo se tutti i livelli da 0 a $i-1$ sono completi (contengono 2^i nodi) e l'ultimo livello i è popolato a partire da sinistra (approfondito nella sezione heap).

Proprietà di un albero binario proprio

Dati: n = numero di nodi in T $\quad$ h = altezza di T
m = numero di foglie in T $\quad$ $n-m$ = numero di nodi interni in T

Un albero binario proprio ha le seguenti proprietà:

binario generico	**binario estremo**	**binario perf. bil.**
$m = n - m + 1$	$m = n - m + 1$	$m = n - m + 1$
$h + 1 \leq m \leq 2^h$	$h + 1 = m$	$m = 2^h$
$h \leq n - m \leq 2^h - 1$	$h = n - m$	$n - m = 2^h - 1$
$2h + 1 \leq n \leq 2^{h+1} - 1$	$2h + 1 = n$	$n = 2^{h+1} - 1$
$\log_2(n+1) - 1 \leq h \leq \frac{n-1}{2}$	$h = \frac{n-1}{2}$	$h = \log_2(n+1) - 1$

Dall'ultima proprietà si osserva che la miglior stima dell'altezza h di un generico albero binario è $O(n)$ e non è sempre detto che sia $O(\log n)$.

3.2 Visite: inorder

Visita inorder

- L'algoritmo prevede di visitare prima il figlio sinistro, poi il nodo stesso e infine il figlio destro.
- La complessità dell'algoritmo è $\Theta\left(n + \sum_{v \in T} t_v\right)$, con t_v costo della visita di un nodo

---

0: **Algoritmo:** INORDER(v)
Input: v nodo dell'albero T
Output: /

1: **if** (T.left(v) $\neq$ null) **then**
2: inorder(T.left(v))
3: visita v
4: **if** (T.right(v) $\neq$ null) **then**
5: inorder(T.right(v))

3.3 Parser Tree

Definizione

Il Parse Tree T associato ad una espressioe aritmetica E (con operatori solo binari) è un albero binario proprio in cui i nodi foglia contengono le variabili/costanti di E e i nodi interni contengono gli operatori di E, in modo tale che:

- se $E = a$, con a costante/variabile, allora T è costituito da un'unica foglia contenente a
- se $E = E_1 op E_2$, la radice T contiene l'operatore op e ha come sottoalbero sinistro E_1 e come sottoalbero destro E_2

Notazione infissa

- L'algoritmo sfrutta la visita inorder degli alberi binari.
- La complessità dell'algoritmo è $\Theta(n)$, in quanto t_v costo della visita di un nodo è $\Theta(1)$.

0: **Algoritmo:** INFIX(T, v, L)
Input: Parse Tree T, $v \in T$, lista L
Output: E_v in notazione infissa nella lista L

1: **if** (T.isExternal(v)) **then**
2: L.addLast(v.getElement())
3: **else**
4: L.addLast("(")
5: infix(T, T.left(v), L)
6: L.addLast(v.getElement())
7: infix(T, T.right(v), L)
8: L.addLast(")")

Notazione postfissa

- L'algoritmo sfrutta la visita in postorder applicata agli alberi binari.
- La complessità dell'algoritmo è $\Theta(n)$, in quanto t_v costo della visita di un nodo è $\Theta(1)$.

0: **Algoritmo:** POSTFIX(T, v, L)
Input: Parse Tree T, $v \in T$, lista L
Output: E_v in notazione postfissa nella lista L

1: **if** (T.isExternal(v)) **then**
2: L.addLast(v.getElement())
3: **else**
4: postfix(T, T.left(v), L)
5: postfix(T, T.right(v), L)
6: L.addLast(v.getElement())

3.4 Implementazione in Java

Interfaccia Iterator

```java
// cursore che permette di muoversi tra gli elementi di una collezione
public interface Iterator<E> {
    /** Returns true if the scan of the collection is not over */
    boolean hasNext();
    /** Returns the next element in the collection */
    E next();
}
```

Interfaccia Iterable

```java
// struttura dati iterabile
public interface Iterable<E> {
    /** Returns an iterator of the collection */
    Iterator<E> iterator()
}
```

Interfaccia Tree

```java
// Albero
public interface Tree<E> extends Iterable {
    /** Returns the number of positions in the tree */
    int size();
    /** Returns true if the tree contains no positions */
    boolean isEmpty();
    /** Returns the Position of the root (or null if empty)*/
    Position<E> root();
    /** Returns the Position of p's parent (or null if p is the root) */
    Position<E> parent(Position<E> p);
    /** Returns an iterable containing p's children */
    Iterable<Position<E>> children(Position<E> p);
    /** Returns the number of children of p */
    int numChildren(Position<E> p);
    /** Returns true if p is internal */
    boolean isInternal(Position<E> p);
    /** Returns true if p is external */
    boolean isExternal(Position<E> p);
    /** Returns true if p is root */
    boolean isRoot(Position<E> p);
    /** Returns an iterator to all element in the tree */
    Iterator<E> iterator();
    /** Returns an iterable containing all positions in the tree */
    Iterable<Position<E>> positions();
}
```

Interfaccia BinaryTree

```java
// Albero binario
public interface BinaryTree<E> extends Tree<E> {
    /** Returns the Position of p's left child(or null if empty)*/
    public Position<E> left(Position<E> p);
    /** Returns the Position of p's right child(or null if empty)*/
    public Position<E> right(Position<E> p);
    /** Returns the Position of p's sibling (or null p is an only child)*/
    public Position<E> sibling(Position<E> p);
}
```

4 Priority Queue

4.1 Introduzione

Definizione

Una priority queue è una collezione di entry in cui le chiavi rappresentano una priorità e provengono da un insieme totalmente ordinato. Minore è il valore della chiave, maggiore è la priorità.

Metodi e interfaccia priority queue

```
// Priority queue
public interface PriorityQueue<k,v> {
    int size();
    boolean isEmpty();
    /** Inserts and returns a new entry (key,value) */
    Entry<K,V> insert(K key, V value);
    /** Returns an entry with min key, without removing it */
    Entry<K,V> min();
    /** Returns and removes an entry with min key */
    Entry<K,V> removeMin();
}
```

4.2 Implementazioni e relativa complessità

metodo	unordered list	ordered list	heap
`min`	$O(n)$	$O(1)$	$O(1)$
`removeMin`	$O(n)$	$O(1)$	$O(\log(n))$
`insert`	$O(1)$	$O(n)$	$O(\log(n))$

Nell'implementazione con la **lista non ordinata**, l'inserimento impiega tempo costante, ma per l'accesso alla chiave minima e di conseguenza la relativa rimozione è necessario applicare un algoritmo di ricerca che impiega tempo lineare.

Nell'implementazione con la **lista ordinata**, l'accesso alla chiave minima e la relativa rimozione impiegano tempo costante, ma per l'inserimento è necessario applicare l'algoritmo di `insertionsort` che impiega tempo lineare.

Nell'implementazione con lo **heap**, l'inserimento e la rimozione impiegno un tempo logaritmico, per come definiti i metodi di inserimento e rimozione nell'heap (up heap bubbling e down heap bubbling), mentre l'accesso alla chiave minima impiega tempo costante. Quest'ultima implementazione offre una migliore efficienza computazionale e anche una migliore efficienza spaziale (se l'heap è implementato efficientemente con un array).

5 Heap

5.1 Introduzione

Definizione

Lo heap (mucchio in italiano) è un albero binario completo in cui i nodi contengono delle entry $< k, v >$ e soddisfano una proprietà chiamata heap order property: siano $< k_2, v >$ e $< k_3, v >$ figli di $< k_1, v >$, allora vale $k_1 \leq \min\{k_2, k_3\}$.

Proprietà

- un albero binario completo ha altezza $h = \lfloor \log n \rfloor$
- esiste ed è unico, l'albero binario completo di n entry
- le chiavi lungo un cammino dalla radice alle foglie formano una sequenza non decrescente
- per qualsiasi figlio u di v vale v.getKey() $\leq u$.getKey()
- la radice contiene la entry con chiave minima
- se le entry sono tutte distinte, la entry con chiave massima si trova in una foglia

Level numbering e rappresentazione tramite array

La rappresentazione Level numbering permette di assegnare un indice a ciascun nodo dello heap secondo le seguenti regole:

1. la radice ha indice 0 e si trova in $P[0]$
2. i figli di $P[i]$ hanno indici $2i + 1$ e $2i + 2$, si trovano in $P[2i + 1]$ e $P[2i + 2]$
3. il padre di $p[i]$ ha indice $\lfloor (i - 1)/2 \rfloor$ e si trova in posizione $P[\lfloor (i - 1)/2 \rfloor]$

Si osserva che i nodi di ciascun livello hanno indici crescenti da sinistra verso destra. In questo modo è possibile rappresentare uno heap su un array, ottenuto giustapponendo i nodi di ogni livello mantenendo le posizioni recicproche dei nodi da sinistra verso destra. Tale rappresentazione è space efficient perché c'è corrispondenza 1:1 tra un albero binario completo e un array.

Nodo Last

Per praticità, il nodo più a destra dell'ultimo livello è chiamato nodo Last e nella rappresentazione tramite array e level numbering si trova nell'ultima cella dell'array con indice $\lfloor n - 1 \rfloor$.

5.2 Algoritmi base: getmin, insert e remove

Accesso alla entry con chiave minima

- L'algoritmo restituisce la entry contenuta nella radice dell'heap
- La complessità dell'algoritmo è $O(1)$

0: **Algoritmo:** GETMIN()
Input: riferimento implicito all'heap rappresentato tramite array
Output: entry con chiave minima

1: return P[0]

Inserimento di una nuova entry

- L'algoritmo inserisce una nuova entry attraverso uno up-heap bubbling.
- La complessità dell'algoritmo è $\Theta(\log n)$, proporzionale all'altezza dell'heap.

0: **Algoritmo:** INSERT(k, v)
Input: entry da inserire e riferimento implicito all'heap $P[]$
Output: /

1: $e \leftarrow (k, v)$
2: $P[++last] \leftarrow e$
3: $i \leftarrow last$
4: **while** ($i > 0$ and $P[\lfloor (i-1)/2 \rfloor]$.getKey() $\geq P[i]$.getKey()) **do** ▷ up-heap bubbling
5: swap($P[i]$, $P[\lfloor (i-1)/2 \rfloor]$)
6: $i \leftarrow \lfloor (i-1)/2 \rfloor$
7: return e

Rimozione della entry con chiave minima

- L'algoritmo rimuove la radice dell'heap attraverso un down-heap bubbling.
- La complessità dell'algoritmo è $\Theta(\log n)$, proporzionale all'altezza dell'heap.

0: **Algoritmo:** REMOVEMIN()
Input: riferimento implicito all'heap $P[]$
Output: entry con chiave minima rimossa

1: $minEntry \leftarrow P[0]$
2: $P[0] \leftarrow P[last--]$
3: $i \leftarrow 0$
4: $j \leftarrow$ indexMinChild(P, i)
5: **while** ($j \neq$ null and $P[i]$.getKey() $> P[j]$.getKey()) **do** ▷ down-heap bubbling
6: swap($P[i]$, $P[j]$)
7: $i \leftarrow j$
8: $j \leftarrow$ indexMinChild(P, i)
9: return $minEntry$

Rimozione di una generica entry

- L'algoritmo rimuove una generica entry dell'heap e ripristino la heap order property con un down-heap bubbling e un up-heap bubbling.
- La complessità dell'algoritmo è $\Theta(\log n)$, proporzionale all'altezza dell'heap.

0: **Algoritmo:** REMOVE(l)
Input: indice della entry da rimuovere e riferimento implicito all'heap $P[]$
Output: entry rimossa

```
1:  retValue ← P[l]
2:  P[l] ← P[last − −]
3:  i ← l
4:  j ← indexMinChild(P, i)
5:  while (j ≠ null and P[i].getKey() > P[j].getKey()) do          ▷ down-heap bubbling
6:      swap(P[i], P[j])
7:      i ← j
8:      j ← indexMinChild(P, i)
9:  while (i > 0 and P[⌊(i − 1)/2⌋].getKey() ≥ P[i].getKey()) do   ▷ up-heap bubbling
10:     swap(P[i], P[⌊(i − 1)/2⌋])
11:     i ← ⌊(i − 1)/2⌋
12: return retValue
```

5.3 Costruzione di un heap inplace

Costruzione di un heap da un array, con approccio Top-Down

- L'algoritmo inserisce le chiavi una ad una, incrementanto j e costruisce l'heap con un up-heap bubbling per ogni chiave inserita
- La complessità dell'algoritmo è $\Theta(n \log n)$, ovvero la ripetizione di n volte dell'up-heap bubbling con complessità $\Theta(\log n)$

0: **Algoritmo:** MAKEHEAP_TOPDOWN()
Input: riferimento implicito all'heap $P[]$
Output: entry con chiave minima rimossa

```
1:  for j ← 1 to n − 1 do
2:      i ← j
3:      // up-heap bubbling
4:      while (i > 0 and P[⌊(i − 1)/2⌋].getKey() ≥ P[i].getKey()) do
5:          swap(P[i], P[⌊(i − 1)/2⌋])
6:          i ← ⌊(i − 1)/2⌋
```

Costruzione di un heap da un array, con approccio Bottom-Up

- L'algoritmo costruisce tanti sottoheap che si sviluppano dal livello i fino alla base. Ad ogni iterazione del for i sottoheap si uniranno a due a due con l'inserimento della entry dal livello $i-1$ con un down-heap bubbling.
- La complessità dell'algoritmo è $\Theta(n)$, più vantaggioso rispetto al precedente approccio top-down.

0: **Algoritmo:** makeHeap_BottomUp()
Input: riferimento implicito all'heap $P[]$
Output: entry con chiave minima rimossa

1: **for** $j \leftarrow \lfloor (n-2)/2 \rfloor$ **to** 0 **do** ▷ $P[\lfloor (n-2)/2 \rfloor]$ nodo più a dx nel penultimo livello
2: $i \leftarrow j$
3: $k \leftarrow$ indexMinChild(P,i)
4: // down-heap bubbling
5: **while** ($k \neq$ null and $P[i]$.getKey() $>$ $P[k]$.getKey()) **do**
6: swap($P[i]$, $P[k]$)
7: $i \leftarrow k$
8: $k \leftarrow$ indexMinChild(P,i)

5.4 HeapSort

Ordinamento di un array attraverso una priority queue implementata come heap su array

- L'algoritmo si sviluppa in due fasi:
  fase1: creazione di un max-heap inplace con approcco bottom-up, con complessità $\Theta(n)$
  fase2: estrazione delle chiavi inplace con leggere modifiche, con complessità $\Theta(n \log n)$
- La complessità dell'algoritmo è $\Theta(n \log n)$,

0: **Algoritmo:** heapSort(P)
Input: array P con chiavi da riordinare
Output: array P con chiavi ordinate
▷ 1. Creazione del max-heap con approccio bottom-up
1: **for** $j \leftarrow \lfloor (n-2)/2 \rfloor$ **to** 0 **do**
2: $i \leftarrow j$
3: $k \leftarrow$ indexMaxChild(P, i) ▷ NB: max-heap: uso indexMaxChild()
4: **while** ($k \neq$ null AND $P[i]$.getKey() $<$ $P[k]$.getKey()) **do** ▷ NB: max-heap: $P[i] < P[k]$
5: swap($P[i]$, $P[k]$)
6: $i \leftarrow k$
7: $k \leftarrow$ indexMaxChild(P, i) ▷ NB: max-heap: uso indexMaxChild()
▷ 2. Estrazione delle entry e down-heap bubbling
8: $last \leftarrow n-1$
9: **for** $j \leftarrow 1$ **to** 0 **do**
10: swap($P[0]$, $P[last]$) ▷ sposto chiave massima in fondo
11: $last \leftarrow n-j-1$ ▷ escludo la chiave massima appena spostata
12: $i \leftarrow 0$ ▷ down-heap bubbling di $P[0]$
13: $k \leftarrow$ indexMaxChild(P, i)
14: **while** ($k \neq$ null AND $P[i]$.getKey() $<$ $P[k]$.getKey()) **do**
15: swap($P[i]$, $P[k]$)
16: $i \leftarrow k$
17: $k \leftarrow$ indexMaxChild(P, i) ▷ NB: max-heap: uso indexMaxChild()

---

6 Mappe

6.1 Introduzione

Definizione

Una mappa è una collezione di entry con chiavi distinte da un insieme U per cui è definito l'operatore $=$. La mappa implementa l'accesso, l'inserimento e la rimozione delle entry attraverso le chiavi.

Metodi e interfaccia in Java

```
public interface Map<K,V>{
    int size();
    boolean isEmpty();
    /** Returns the value of the entry (k,v) with k = key if exists, otherwise null */
    V get (K key);
    /** If there is an entry (k,v) with k = key then update the value of that entry and
    returns the old value, otherwise inserts a new entry (key, value) and returns null */
    V put (K key, V value);
    /** Removes the entry (k,v) with k = key and returns the value of the entry,
    otherwise return null*/
    V remove (K key);
    Iterable<K> keySet();
    Iterable<V> values();
    Iterable<Entry<K,V>> entrySet();
}
```

6.2 Implementazioni e relativa complessità

metodo	position-based list	array	tabella hash	MWST, (2,4)-Treee, RB-Tree
get	$O(n)$	$O(1)$	$O(1+\lambda)$	$O(\log n)$
put	$O(n)$	$O(1)$	$O(1+\lambda)$	$O(\log n)$
remove	$O(n)$	$O(1)$	$O(1+\lambda)$	$O(\log n)$

Implementazione con position-based list

L'implementazione con una position-based list è efficiente dal punto di vista spaziale, in quanto lo spazio occupato in memoria è proporzionale al numero di entry, ma è molto svantaggiosa dal punto di vista computazionale in quanto i metodi utilizzano la ricerca in una lista che impiega complessità $O(n)$.

Implementazione con array

Si suppone di poter creare un array con gli indici $[0, |U|-1]$ e di disporre di un mapping 1:1 tra l'insieme delle chiavi e l'indice dell'array. È possibile riservare una cella dell'array ad ogni possibile entry della mappa. In questo modo i vari metodi hanno complessità costante, ma nel caso in cui $|U| \gg \#$ entry, la struttura diventa molto svantaggiosa dal punto di vista della memoria occupata.

Implementazione con tabella hash

Simile all'implementazione precedente, ma utilizzando le hash table e le funzioni hash è possibile ridurre notevolmente la memoria occupata a discapito di una lievemente peggiore complessità computazionale. La complessità dipende dal load factor λ o indice di riempimento medio della tabella hash. Il load factor λ rappresenta un trade-off tra lo spazio occupato in memoria e la complessità computazionale, per λ grandi si avrà migliore complessità spaziale e peggiore complessità computazionale, vicecersa per λ piccoli si avrà il contrario. In genere si sceglie $\lambda < 0.9$ e se $\lambda \in O(1)$, la complessità finale è $O(1)$.

Implementazione con alberi binari di ricerca

Supponendo che le chiavi provengano da un insieme totalmente ordinato, è possibile implementare la mappa attraverso particolari alberi binari di ricerca (2,4-Tree o Red-Black Tree) in cui la ricerca e i tre metodi hanno complessità $O(\log n)$.

7 Hash Table

7.1 Introduzione

Definizione

Una tabella hash è una struttura che possiede i seguenti elementi:

1. una funzione di hash $h : \{\text{chiavi}\} \to [0, N-1]$
2. un bucket array A con capacità N, in cui ogni bucket $A[i]$ sono memorizzate tutte le entry $< k, v >$ per cui $h(k) = i$, con $0 \leq i < N$
3. un metodo di risoluzione delle collisioni (chiavi distinte associate allo stesso bucket)

Funzione di hash

Una buona funione di Hash deve essere il più possibile un processo random:

- $P[h(k) = i] = 1/N$, ovvero la distribuzione delle chiavi dei bucket deve tendere all'uniforme
- $P[h(k) = i \mid h(k') = j] = P[h(k) = i]$, ovvero gli assegnamenti devono essere indipendenti tra loro

Esempi di funzioni Hash per tipi built-in di Java:

- `byte`, `short`, `char`, `int`: $k \mapsto (\text{int})k$
- `float` (32 bit): $k \mapsto \text{Float.floatToIntBits}(k)$
- `long` (64 bit): $k \mapsto (\text{int})(k \gg 32 + (\text{int})k)$
- `double` (64 bits): $k \mapsto ((\text{int})(k \gg 32 + (\text{int})k)) \circ (\text{Double.doubleToIntBits}(k))$
- `string`: due possibilità:
  - Polinomial hash code: $h(S) = \sum_{i=0}^{k-1} s_i \cdot a^{k-i-1}$ con $a = 31, 33, 37, 39, 41$, Java usa $a = 31$
  - Cyclic Shift: somma carattere per carattere e dopo ogni somma si esegue una rotazione a sinistra di 5 posizioni (più facile implementazione rispetto al calcolo delle potenze di a)

Funzione di compressione

Per convertire un generico numero $\in \mathbb{N}$ generato dalla funzione di hash in un intero valido per accedere alla sequenza, si utilizzano due funzioni di compressione:

- **Division Method**: $i \mapsto i \mod N$
  è bene scegliere N primo e distante da potenze di 2: se $N = 2^p$ è come considerare solo i p bit meno significativi, se $N = 10^p$ è come considerare solo le p cifre meno significative in base 10.
- **Multiply-Add-Divide MAD**: $i \mapsto [(ai + b) \mod p] \mod N$
  con p primo, $p > N$, $a, b \in [0, p-1]$ scelti a caso, $a > 0$, più costoso ma con migliore distribuzione

Risoluzione delle collisioni

Per la risoluzione delle collisioni ricorre a due metodi:

- **Separate Chaining**: ogni bucket è visto come una Map più piccola implementata come lista
- **Open Adressing**: consiste nel salvare le entry direttamente nelle celle del bucket array, senza far ricorso a strutture aggiuntive, ma si complica la loro gestione; non affrontato a lezione

7.2 Algoritmi base: get, put, remove

Accesso ad un elemento - get

- L'algoritmo si basa sulla ricerca della chiave nei bucket con funzione di hash e sulla ricerca nella lista concatenata associata al bucket in questione.
- La complessità dell'algoritmo al caso medio è $\Theta(1 + \lambda)$.

0: **Algoritmo:** GET(k)
Input: chiave k dell'elemento cercato e riferimento implicito al bucket array A
Output: valore della entry con chiave k

1: **if** ($\exists$ entry $(k, x) \in A[h(k)]$) **then**
2: return x
3: **else**
4: return null

Inserimento di un elemento - put

- L'algoritmo si basa sulla ricerca della chiave nei bucket con funzione di hash e sulla ricerca nella lista concatenata associata al bucket in questione.
- La complessità dell'algoritmo al caso medio è $\Theta(1 + \lambda)$.

0: **Algoritmo:** PUT(k, v)
Input: entry (k, v) da inserire e riferimento implicito al bucket array A
Output: vecchio valore della entry (k, v) se presente

1: **if** ($\exists$ entry $(k, x) \in A[h(k)]$) **then**
2: sostituisci x con v
3: return x
4: **else**
5: inserisci (k, v) in coda al bucket $A[h(k)]$
6: incrementa di 1 la *size* della hashTable
7: return null

Rimozione di un elemento - remove

- L'algoritmo si basa sulla ricerca della chiave nei bucket con funzione di hash e sulla ricerca nella lista concatenata associata al bucket in questione.
- La complessità dell'algoritmo al caso medio è $\Theta(1 + \lambda)$.

0: **Algoritmo:** GET(k)
Input: chiave k dell'elemento cercato e riferimento implicito al bucket array A
Output: valore della entry con chiave k

1: **if** ($\exists$ entry $(k, x) \in A[h(k)]$) **then**
2: rimuovi (k, x) da $A[h(k)]$
3: decrementa di 1 la *size* della hashTable
4: return x
5: **else**
6: return null

7.3 Load Factor e complessità

Load Factor

Per una tabella hash di capacità N, con n entry memorizzate, il load factor è $\lambda := \frac{n}{N}$.
Rappresenta la lunghezza media delle liste nei bucket.

Complessità al caso pessimo

Si osserva che per una tabella hash di capacità N, con n entry memorizzate, la complessità dei tre metodi al caso pessimo è $\Theta(n)$. Se tutte le entry fossero contenute nello stesso bucket la ricerca (prevista da ogni metodo) di una chiave in una lista è $\Theta(n)$. Il risultato di n inserimenti nella tabella diventerà $\Theta(n^2)$.

Complessità al caso medio

Al caso medio, il costo di un inserimento si ottene considerando il costo della ricerca della chiave k in nella lista i contenente n_i entry, ovvero $O(n_i + 1)$. Si ha, quindi:

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

- se $\lambda \in O(1)$, allora la complessità finale sarà $\Theta(1)$
- in generale si mantiene $\lambda < 0.9$, in Java $\lambda \leq 0.75$
- il load factor λ rappresenta un trade-off tra lo spazio occupato in memoria e la complessità computazionale, per λ grandi si avrà migliore complessità spaziale e peggiore complessità computazionale, vicecersa per λ piccoli si avrà migliore complessità computazionale, ma peggiore complessità spaziale.

7.4 Rehashing

Funzionamento

Con il popolamento della hashTable, il load factor cresce e si rischia di superare la soglia impostata. Per mantenere un load factor accettabile, è necessario eseguire un ridimensionamento del bucket array e di conseguenza è necessario eseguire un rehashing delle entry. Per ottenere un buon rehash è bene scegliere la nuova dimensione $N' > 2N$ con N' numero primo.

Complessità

Siccome so già che le chiavi in una hash table sono distinte, quando le inserisco nel nuovo bucket array ridimensionato non ho bisogno di eseguire la ricerca per verificare se esiste già una entry con la stessa chiave. Questo bypass mi permette di eseguire gli n inserimenti individualmente in tempo costante e complessivamente il rehash ha una complessità di $\Theta(n)$.

Il rehash viene eseguito solo dopo n inserimenti con load factor ottimale che possiedono un costo aggregato di $\Theta(n\lambda) = \Theta(n)$. Il costo del rehash ($\Theta(n)$) viene, ammortizzato da quello aggregato dei precedenti n inserimenti.

8 Alberi binari di ricerca

8.1 Alberi binari di ricerca in generale

Definizione

Un albero binario di ricerca è un albero binario in cui i nodi interni memorizzano entry con chiavi provenienti da un universo ordinato e per ogni nodo v con chiave k, le chiavi di T.left(v) sono $< k$, mentre le chiavi di T.right(v) sono $> k$. Le foglie sono sentinelle e non contengono entry.

Algoritmo di ricerca

- L'algoritmo si basa sull'ordinamento reciproco dei nodi all'interno dell'albero ed esegue una discesa dalla radice fino al nodo che contiene la entry cercata o alla foglia dove inserirla.
- La complessità dell'algoritmo è $\Theta(h)$, che può diventare $\Theta(n)$ se l'albero è molto sbilanciato.

0: **Algoritmo:** TreeSearch(k, v)
Input: chiave k della entry cercata, nodo v da cui cercare e riferimento implicito all'ABR T
Output: nodo in T_v dove è presente la entry con chiave k o dove va inserita

```
1:    if (T.isExternal(v) OR v.getElement().getKey() = k) then
2:        return v
3:    if (k < v.getElement().getKey()) then
4:        return TreeSearch(k,T.left(v))
5:    else
6:        return TreeSearch(k,T.right(v))
```

Limiti degli alberi binari di ricerca in generale

Il problema degli alberi binari di ricerca è che la complessità dei vari metodi varia tra $\Theta(\log n)$ se l'albero è perfettamente bilanciato e $\Theta(n)$ se l'albero è fortemente sbilanciato. In quest'ultimo caso l'ABR non migliora le prestazioni di una comune lista concatena. Le possibili soluzioni sono:

- ribilanciare l'albero quando lo sbilanciamento supera una soglia prefissata (AVL o Red-Black Tree)
- rendere i nodi più capienti per assorbire meglio gli effetti di inserimento e rimozione (MW Search Tree o (2,4)-Tree)

Algortimo di accesso - get

- L'algoritmo si basa sulla ricerca vista sopra.
- La complessità dell'algoritmo è $\Theta(h)$, che può diventare $\Theta(n)$ se l'albero è molto sbilanciato.

0: **Algoritmo:** get(k)
Input: chiave k della entry cercata e riferimento implicito all'ABR T
Output: valore della entry con chiave k

```
1:    w ← TreeSearch(k, T.root())
2:    if (T.isInternal(w)) then
3:        return w.getElement().getValue()
4:    else
5:        return null
```

Algortimo di inserimento - insert

- L'algoritmo si basa sulla ricerca vista sopra.
- La complessità dell'algoritmo è $\Theta(h)$, che può diventare $\Theta(n)$ se l'albero è molto sbilanciato.

0: **Algoritmo:** PUT(k, v)
Input: entry (k, v) da inserire e riferimento implicito all'ABR T
Output: vecchio valore della entry (k, v) se presente

1: $w \leftarrow$ TreeSearch(k, T.root())
2: **if** (T.isInternal(w)) **then**
3: $x \leftarrow w$.getElement().getKey()
4: sostituisci v con x nella entry w.getElement()
5: return x
6: **else**
7: expandExternal(w,(k, v))
8: incrementa di 1 il numero di entry di T
9: return null

Algortimo di rimozione - remove

- L'algoritmo si sviluppa in due casi:
  1. se il nodo da rimuovere ha una foglia come figlio, faccio salire l'altro figlio
  2. se il nodo da rimuovere non ha foglie come figli, sostituisco il nodo da rimuovere w con il precedente nella visita inorder y (che avrà una foglia come figlio destro) e sposto il sottoalbero sinistro di y al posto di y
- La complessità dell'algoritmo è $\Theta(h)$, ottenuta dalla complessità aggregata di TreeSearch ($\Theta(h)$), del caso 1 ($\Theta(1)$) e del caso 2 ($\Theta(h)$), più altre operazioni ($\Theta(1)$).

0: **Algoritmo:** REMOVE(k)
Input: chiave k della entry da rimuovere e riferimento implicito all'ABR T
Output: valore della entry rimossa

1: $w \leftarrow$ TreeSearch(k, T.root())
2: **if** (T.isInternal(w)) **then**
3: $avlue \leftarrow w$.getElement().getValue()
4: decrementa di 1 il numero di entry di T
5: ▷ Caso 1: w interno con almento un figlio foglia
6: **if** (T.isExternal(T.left(w)) OR T.isExternal(T.right(w))) **then**
7: $u_L \leftarrow T$.left(w) $-$ $u_R \leftarrow T$.right(w)
8: **if** T.isExternal(u_L) **then** ▷ Caso 1.1: la foglia è il figlo sinistro
9: cancella w e u_L
10: fai salire u_R al posto di w
11: **else** ▷ Caso 1.2: la foglia è il figlo destro
12: cancella w e u_R
13: fai salire u_L al posto di w
14: **else** ▷ Caso 2: w interno con due figli
15: $y \leftarrow$ nodo con chiave massima del sottoalbero di sinistra
16: sposta il figlio sinistro di y al posto di y
17: sostituisci il nodo w da rimuovere con il nodo y
18: return $value$
19: **else**
20: return null

8.2 Multi-Way Search Tree - MWST

Definzinione

Un Multi-Way Search Tree è un albero ordinato tale che ogni nodo interno ha $d \geq 2$ figli $(v_1, v_2, \dots v_d)$ detto $d - node$ che soddisfa le seguenti caratteristiche:

- memorizza $d-1$ entry $(k_1, x_1), (k_2, x_2), \dots (k_{d-1}, x_{d-1})$ con $k_1 < k_2 < \cdots < k_{d-1}$
- la chiave di ogni entry $e \in T_{v_i}$ soddisfa $k_{i-1} < e.\text{getKey}() < k_{i+1}$, supponendo $k_0 = 0$, $k_d = +\infty$

Proprietà

- per ogni nodo v con ad esempio 3 entry $(k_1, x_1), (k_2, x_2), (k_3, x_3)$ e 4 figli v_1, v_2, v_3, v_4, le chiavi sono ordinate da sinistra verso destra nel seguente modo:

  $$0 < \{\text{chiavi} \in T_{v_1}\} < k_1 < \{\text{chiavi} \in T_{v_2}\} < k_2 < \{\text{chiavi} \in T_{v_3}\} < k_3 < \{\text{chiavi} \in T_{v_4}\} < +\infty$$

- un MWST con n entry ha $n+1$ foglie
- se tutti i nodi fossero dei $d-$node, allora l'albero viene detto d-ario, con:

  $$\#\text{ nodi interni} = x, \qquad \#\text{ entry} = (d-1)x, \qquad \#\text{ foglie} = (d-1)x + 1$$

Algoritmo di ricerca

- L'algoritmo si basa sull'ordinamento reciproco dei nodi all'interno dell'albero ed esegue una discesa dalla radice fino al nodo che contiene la entry cercata o alla foglia dove inserirla.
- La complessità dell'algoritmo è $\Theta(h \cdot d_{max})$, con d_{max} = massimo valore di d in T_v

---

0: **Algoritmo:** MWTreeSearch(k, v)
Input: chiave k della entry cercata, nodo v da cui cercare e riferimento implicito al MWST T
Output: nodo in T_v dove è presente la entry con chiave k o dove va inserita

1: **if** (T.isExternal(v)) **then**
2: return v
3: trova i tale che $k_{i-1} < k \leq k_i$ con $k_0 = 0, k_d = +\infty$
4: **if** ($k = k_i$) **then**
5: return v
6: **else**
7: return MWTreeSearch(k,v_i)

Algortimo di accesso - get

- L'algoritmo si basa sulla ricerca vista sopra.
- La complessità dell'algoritmo è $\Theta(h \cdot d_{max})$.

0: **Algoritmo:** GET(k)
Input: chiave k della entry cercata e riferimento implicito al MWST T
Output: valore della entry con chiave k

1: $w \leftarrow$ MWTreeSearch(k, T.root())
2: **if** (T.isInternal(w)) **then**
3: trova $e \in w$ tale che e.getKey() $= k$
4: return e.getValue()
5: **else**
6: return null

8.3 (2,4)-Tree

Definzinione

Un (2,4)-Tree è un MWS-Tree tale che:

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

Proprietà

- l'altezza di un (2,4)-Tree con $n > 0$ entry è $\Theta(\log n)$
- la complessità dei metodi di ricerca e di accesso è $\Theta(\log n)$

Algoritmo di inserimento - put

- L'algoritmo si basa sulla ricerca vista sopra e sul fatto che un nodo più contenete più entry
- La complessità dell'algoritmo è $\Theta(\log n)$ data dalla MWTreeSearch ($\Theta(\log n)$) e dallo split ($\Theta(\log n)$)

0: **Algoritmo:** PUT(k, x)
Input: entry (k, x) da inserire e riferimento implicito al (2,4)-Tree T
Output: valore della entry con chiave k

```
1:  w ← MWTreeSearch(k, T.root())
2:  if (T.isInternal(w)) then                ▷ Caso 1: se esiste già una entry con chiave k
3:      e ← entry in w con e.getKey() = k
4:      y ← e.getValue()
5:      sostituisci x con y in e.value()
6:      return y
7:  else                                     ▷ Caso 2: bisogna inserire una nuova entry
8:      e ← (k, x)
9:      if (T.isRoot(w)) then                ▷ Caso 2.1: l'albero è vuoto → creo radice e due figli
10:         expandExternal(w,e)
11:     else                                 ▷ Caso 2.2: lo inserisco nel padre
12:         u ← T.parent(w)
13:         inserisci e in u aggiungendo una foglia w'
14:         if u è un 5-node then            ▷ se c'è overflow, invoco split
15:             split(u)
16:     incrementa di 1 il numero delle entry in T
17:     return null
```

Algoritmo di split

- L'algoritmo propaga l'overflow verso l'alto e se serve aumenta l'altezza dell'albero:
  1. se ho la radice in overflow, spezzo la radice in due nodi: $(e1, e_2, e_3, e_4) \rightarrow (e_1, e_2), (e_4)$ e creo una nuova radice con la entry "centrale" (e_3) e con i due nodi $(e_1, e_2), (e_4)$ come figli
  2. altrimenti spezzo il nodo in due $(e1, e_2, e_3, e_4) \rightarrow (e_1, e_2), (e_4)$ e inserisco la entry "centrale" (e_3) nel padre; se il padre è in overflow, invoco lo split sul padre
- La complessità dell'algoritmo è $\Theta(\log n)$ in quanto ogni invocazione ha costo $O(1)$ e # incocazioni è proporzionale all'altezza dell'albero $\Theta(\log n)$

0: **Algoritmo:** SPLIT(u)
Input: nodo in overflow e riferimento implicito al (2,4)-Tree T
Output: albero senza nodi in overflow

1: sia $u = (e1, e_2, e_3, e_4)$ con figli u_1, u_2, u_3, u_4, u_5
2: creo due nodi $u' = (e_1, e_2)$ con figli u_1, u_2, u_3 e $u'' = (e_4)$ con figli u_4, u_5
3: **if** (T.isRoot(u)) **then** ▷ se ho la radice in overflow
4: crea una nuova radice contenente e_3 e con figli u' e u'' e cancella u
5: **else** ▷ se ho un generico nodo in overflow
6: $v \leftarrow T$.parent(u)
7: inserisci e_3 in v con figli u' e u'' e cancella u
8: **if** (v è un 5-node) **then** ▷ se il padre è in overflow
9: split(u)

Algortimo di rimozione - remove

- L'algoritmo si basa sulla ricerca vista sopra e sul fatto che un nodo più contenete più entry
- La complessità dell'algoritmo è $\Theta(\log n)$ data dalla MWTreeSearch ($\Theta(\log n)$) e dal delete ($\Theta(\log n)$)

0: **Algoritmo:** REMOVE(k)
Input: chiave k della entry da rimuovere e riferimento implicito al (2,4)-Tree T
Output: valore della entry rimossa

1: $w \leftarrow$ MWTreeSearch(k, T.root())
2: **if** (T.isInternal(w)) **then** ▷ Caso 1: se esiste la entry con chiave k
3: $e \leftarrow$ entry in w con e.getKey() $= k$
4: $y \leftarrow e$.getValue()
5: **if** (height(w) $= 1$) **then** ▷ Caso 1.1: se la entry si trova alla base (non ha nodi figli)
6: delete(e, w)
7: **else** ▷ Caso 1.2: se la entry ha nodi figli
8: $v \leftarrow$ figlio di w a sx di e
9: $e' \leftarrow$ entry con chiave massima in T_v
10: $z \leftarrow$ nodo contenente e'
11: copia e' al posto di e in w ▷ sostituisco il nodo da eliminare
12: delete(e',z) ▷ elimino il sostituito dalla sua posizione originale
13: decrementa di 1 il numero delle entry in T
14: return y
15: **else** ▷ Caso 2: se la entry non esiste
16: return null

Algoritmo di delete

- L'algoritmo propaga l'underflow verso l'alto e se serve diminuisce l'altezza dell'albero:
  1. se ho la radice in underflow: imposto come nuova radice il nodo non foglia
  2. se il nodo in underflow ha un fratello con $d = 3, 4$: vado in prestito di una entry dal fratello ed eseguo una rotazione delle entry fratello $\rightarrow$ padre $\rightarrow$ nodo in underflow
  3. se il nodo in underflow ha un fratello con $d = 2$: vado in prestito dal padre e sposto la entry che lega i due fratelli dal padre al fratello, aggiungo il sottoalbero del nodo in underflow al fratello e richiamo ricorsivamente delete sul padre per la entry spostata
- La complessità dell'algoritmo è $\Theta(\log n)$ in quanto ogni invocazione ha costo $O(1)$ e # incocazioni è proporzionale all'altezza dell'albero $\Theta(\log n)$

0: **Algoritmo:** DELETE(u, e)
Input: nodo u, entry $e \in u$ da rimuovere e riferimento implicito al (2,4)-Tree T
Output: albero senza nodi in underflow

1: rimuove e da u e figlio vuoto o foglia discriminata da e
2: applico i casi visti sopra

8.4 Red-Black Tree

Definzinione

Un Red-Black Tree (RB-Tree) è un albero binario di ricerca i cui nodi hanno un colore rosso o nero e in cui valgono le seguenti proprietà:

- *Root Property*: la radice è nera
- *External Property*: le foglie sono nere
- *Red Property*: i figli di un nodo rosso sono necessario
- *Depth Property*: tutte le foglie hanno la stessa Black Depth

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

Si osseerva che è possibile trasformare un (2,4)-Tree in un Red-Black Tree e viceversa:

- se ho un solo nodo nero e due foglie $\rightarrow$ 2-nodo con una entry e due foglie
- se ho un nodo nero con un figlio rosso e tre foglie totali $\rightarrow$ 3-nodo con due entry e 3 foglie
- se ho un nodo nero con due figli rossi e quattro foglie totali $\rightarrow$ 4-nodo con tre entry e 4 foglie

9 Multimappe

9.1 Introduzione

Definizione

Una multimappa è una mappa che elimina il vincolo di unicità delle chiavi

Metodi e interfaccia in Java

```
public interface Map<K,V>{
    int size();
    boolean isEmpty();
    /** Returns a collection of all the values of the entrys (k,*) with k = key */
    Iterable<V> get (K key);
    /** Inserts a new entry (key, value) */
    void put (K key, V value);
    /** Removes the entry (k,v) with k = key and returns true if an entry is removed,
    otherwise false */
    boolean remove (K key, V value);
    Iterable<K> keySet();
    Iterable<V> values();
    Iterable<Entry<K,V>> entrySet();
}
```

Indice primario e secondario

Nei database, l'accesso ai dati è reso efficiente attraverso l'accesso ad un indice primario realizzato tramite una mappa e in parallelo attraverso un indice secondario realizzato tramite una multimappa.

9.2 Implementazione e relativa complessità

Implementazione

È possibile implementare una multimappa tramite una mappa in cui le entry sono coppie (k, L_k) con k chiave e L_k una lista che memorizza i valori associati alla chiave. Nel caso in cui si vuole inserire più entry uguali (stessa chiave e valore), si può decidere se bloccare l'inserimento della seconda o memorizzarle entrambe. In quest'ultimo caso, al momento della rimozione si può decidee se rimuoverne solo una o tutte le entry uguali.

Complessità

Per migliore praticità si definisce $n = \#$ chiavi distinte e $s = \max_k |L_k|$ massima lunghezza delle liste associate alle chiavi. Di seguito i metodi e le relative complessità:

- get(k) ha complessità $\Theta(\log n)$ (come quella della mappa, utlizzando la nuova definizione di n)
- put(k,v) ha complessità $\Theta(\log n)$ (come quella della mappa, utlizzando la nuova definizione di n)
- remove(k,v) ha complessità $\Theta(\log n + s)$