Lezione_24_DeA

6.8.2. Esempio

Consideriamo una graduatoria come esempio di Multimappa.
Una entry è strutturata così: `(cognome, punteggio)`.
Si ha una collezione delle seguenti entry: `(Bianchi, 75)`, `(Bianchi, 73)`, `(Bianchi, 60)`, `(Rossi, 15)`, `(Rossi, 96)`, `(Verdi, 59)`.
La Multimappa risultante è: `(Bianchi, 75-73-60)`, `(Rossi, 15-96)`, `(Verdi, 59)`.
Dall'esecuzione di `get(Bianchi)`, `put(Rossi, 30)` e `remove(Bianchi, 75)` risulta:

Operazione	Output
`get(Bianchi)`	`75-73-60`
`put(Rossi, 30)`	Non restituisce output. `(Rossi, 15-96)` diventa `(Rossi, 15-96-30)`.
`remove(Bianchi, 75)`	`TRUE`. La entry `(Bianchi, 75-73-60)` diventa `(Bianchi, 73-60)`.

Indici primari e secondari

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

> Un *Indice primario* è (una struttura di accesso a) una collezione di entry basata su una chiave che *non ammette duplicati*. Viene *realizzato tramite una Mappa*.

Un esempio è il numero di matricola degli studenti.

> Un *indice secondario* è (una struttura di accesso a) una collezione di entry basata su una chiave che *ammette duplicati*. Viene *realizzato tramite Multimappa*.

Ad esempio il cognome degli studenti.

6.8.2. Complessità dei metodi

Ipotesi: L_k implementata tramite lista.
Definiamo:

- n come numero di *chiavi distinte* presenti nella Multimappa (quindi le entry potrebbero essere molte di più);
- $s = \max_k |L_k|$, il massimo è su tutte le chiavi k presenti nella Multimappa;
- `get(k)`: Stessa complessità della Mappa, usando però la nuova definizione di n;
- `put(k, v)`: Stessa complessità della Mappa, usando però la nuova definizione di n e assumendo che v sia inserito in testa o in cosa a L_k;
- `remove(k,v)`: alla complessità di `remove(k)` della Mappa (con la nuova definizione di n) si aggiunge un termine additivo $\Theta(s)$ per la ricerca di v in L_k.

Riepilogo complessità per la Mappa

Considerando una Mappa con n entry avrò:

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

Nota bene

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

Riepilogo complessità per la Multimappa

Considerando una Multimappa con n chiavi distinte avrò:

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

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

Nota bene

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

Riepilogo Mappe

- Mappa: definizione come ADT;
- Tabelle Hash:
  - Ingredienti principali;
  - Funzione hash: hashcode e compression function;
  - Risoluzione delle collisioni tramite chaining;
  - Implementazione dei metodi della Mappa e loro complessità al caso medio sotto l'ipotesi di uniform hashing;
  - Load factor e rehashing;
- Alberi Binari di Ricerca:
  - Definizione;
  - Metodo TreeSearch;
  - Implementazione dei metodi della Mappa;
- (2,4)-Tree:
  - Definizione di Multi-Way Search (MWS) Tree;
  - Relazione tra numero di entry e foglie in un MWS-Tree;
  - Metodo MWTreeSearch;
  - Definizione di un (2,4)-Tree;
  - Altezza di un (2,4)-Tree;
  - Implementazione dei metodi della Mappa;
- Cenni sui Red-Black Tree;

- Implementazione di una Multimappa.

6.9. Esercizi

6.9.1. Mappe pt 2

6.9.1.1. Slide 83

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

Algoritmo `24Height(T)`
Input: (2,4)-Tree T.
Output: Altezza di T.

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

```
h <- 0;
v <- T.root();
while(T.isInernal(v)) do {
     v <- figlio può a sx di v;
     h <- h + 1;
}
return h
```

Correttezza: Banale.

Complessità:

- h_T iterazioni del while ($h_T \equiv$ altezza di T)
- $\Theta(1)$ operazioni per iterazione
  $\implies$ Complessità $\in \Theta(h_T) = \Theta(\log n)$, con n il numero di entry in T

6.9.2. Mappe pt 2

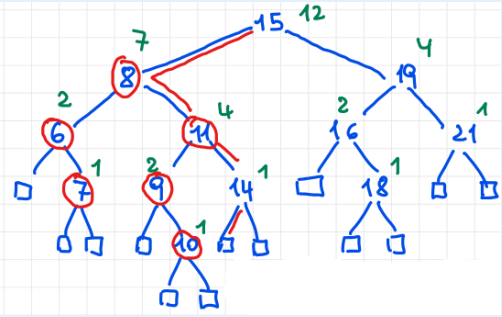
6.9.2.1. Slide 73

> Sia T un albero binario di ricerca dove ogni nodo $v \in T$ memorizza 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.

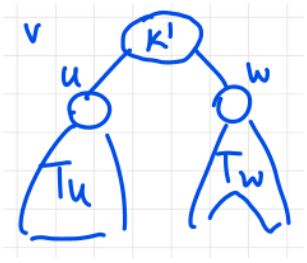
Algoritmo: `CountLE(k,v)`
Input: chiave k, nodo $v \in T$
Output: numero di entry in T_v con chiave $\leq k$
Prima invocazione: `v = T.root()`

Esempio



(I verde le variabili size)
Per $k = 13$ l'output è: 6

Idea:



- Se $k' > k \implies$ restituisco CountLE(k,u) dato che né k' né le chiavi in T_w possono contribuire all'output
- Se $k' \leq k$ restituisco $1 + u.size + CountLE(k, w)$ dato che certamente sia k' che tutte le chiavi in T_u contribuiscono all'output

```
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() + CoutLE(k, T.right(v));
}
```

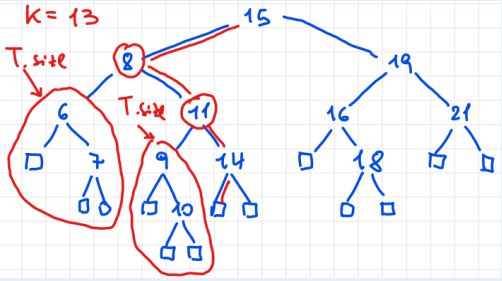
Complessità: Stessa struttura di `TreeSearch`

- $\leq h$ invocazioni ricorsive
- $\Theta(1)$ operazioni in ciascuna invocazione ricorsiva (tranne in invocazioni ricorsive al suo interno)
  $\implies$ Complessità $\in \Theta(h)$, con h l'altezza di T

6.9.2.2. Slide 81

> Analizzare la complessità dell'algoritmo appena sviluppato, 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).

Esempio




```
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.size(T.left(v)) + CoutLE(k, T.right(v));
}
```

Complessità di `CountLE(k, T.root())`
Chiamo X il numero di operazioni eseguite, escludendo quelle relative a `T.size`, e chiamo Y il numero aggregato di operazioni di tutte le chiamate a `T.size`.
$\implies$ Complessità $\in \Theta(X+Y)$

So già che $X \in \Theta(h)$ dall'analisi dell'algoritmo che usa le variabili `v.size`.

Sia m il valore finale restituito da `CounetLE(k, T.root())`; sia m_u il valore restituito da `T.size(u)`; sia $U = \{u : \text{l'algoritmo invoca } \texttt{T.size(u)}\}$, nell'esempio $U = \{6, 9\}$.
Si ha che $m \geq \sum_{u \in U} m_u$ e al caso pessimo $\sum_{u \in U} m_u \in \Theta(m)$.
Ed è facile vedere che $Y \in \Theta\left(\sum_{u \in U} m_u\right) \implies Y \in O(m)$.

$\implies$ Complessità di `CountLE(k, T.root()` è $\Theta(\underbrace{h}_{X} + \underbrace{m}_{Y})$.

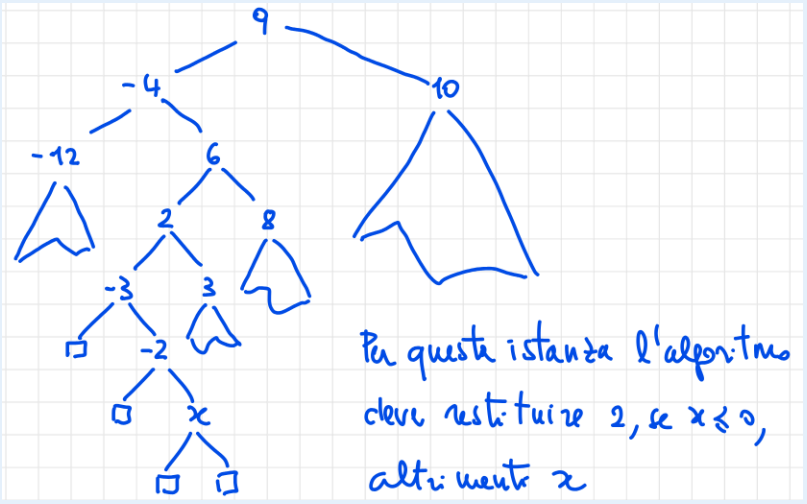
> **Suggerimento**
>
> Sostituire `T.size(T.left(v))` con `CountLE(k, T.left(v))`
> Il caso aggregato di tutte le invocazioni di CountLE fatta in sostituzione di T.size sarà $\Theta(m)$ (con m valore finale restituito)
> $\implies$ Complessità $\in \Theta(h + m)$

6.9.2.3. Slide 80, secondo 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'.

Algoritmo: `MinPositive(T)`
Input: ABR T con chiavi reali distinte.
Output: Minima chiave positiva (se esiste) o "no positive key" altrimenti.

> **Esempio**



Idea:

- Si parte dalla radice;
- Per ogni nodo esaminato v con chiave k
  - Se $k > 0$: salva k come candidato e vai sul figlio sinistro;
  - Se $k \leq 0$: vai sul figlio destro;

```
v <- T.root();
minPk <- +∞;
while (T.isInternal(v)) do{
     k <- v.getElement().getKey();
     if (k > 0) then {
          minPk <- k;
          v <- T.left(v);
     }
     else{
          v <- T.right(v);
     }
}
if(minPk < +∞) then{
     return minPk;
}
else {
     return "no positive key";
}
```

Complessità: $\Theta(h)$, dove h è l'altezza di T.
È dominata dal while:

- $\leq h$ iterazioni;

- $\Theta(1)$ operazioni per iterazione.