Lezione_25_DeA

6.9.3. File Esercizi Mappe

6.9.3.1. Slide 18

> 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 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`

Algoritmo `MinMatStraniero(T,v)`
Input: ABR T e $v \in T$.
Output: minima matricola di uno straniero in T_v, se ne esiste almeno uno, o `null` altrimenti.
Prima invocazione: `v = T.root()`

Idea:

- se v è foglia o `v.numStr = 0` $\implies$ `null`;
- se v è interno:
  - Se u (figlio sinistro di v) ha `u.numStr` $> 0 \implies$ return `MinMatStraniero(T,u)`;
  - Se `u.numStr = 0` e la entry in v ha flag $1 \implies$ return chiave di v;
  - Altrimenti return `MinMatStraniero(T,w)` (w figlio destro di v).

```
if(T.isExternal(v) OR v.numStr = 0) then {
    return null;
}
m <- T.left(v).numStr;
x <- v.getElement().getValue()
```

```
if(m > 0) then return MinMatStraniero(T,T.left(v))
else{
     if(x = 1) then return v.getElement().getKey()
     else return MinMatStraniero(T, T.right(v))
}
```

Complessità: $\Theta(h)$ con h altezza di T. Stessa analisi di `TreeSearch` (viene fatta al massimo una chiamata ricorsiva per nodo, quindi al massimo proporzionale all'altezza).

Fare una versione iterativa dell'algoritmo per esercizio.

6.9.3.2. Slide 23

> Sia D un documento di N parole, rappresentato da un array $D[0], D[1], \ldots, D[N-1]$.
>
> 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.
> 2. Analizzare la complessità dell'algoritmo scegliendo una opportuna implementazione per la Mappa.

Algoritmo `MostFrequentWord(D)`
Input: Documento $D = D[0], \ldots, D[N-1]$ di N parole.
Output: Una parola w con il massimo numero di occorrenze.

Soluzioni banali:

- Usare due cicli for, contando per ogni parola le sue occorrenze $\implies \Theta(n^2)$;
- Ordinare in maniera lessicografica il documento ($\Theta(n \log n)$) e contare il numero di parole in tempo lineare $\implies \Theta(n \log n)$.

Idea:

- Scansione lineare di D;
- Mantenere in una mappa di appoggio M coppie `(parola, numero occorrenze)`, tenendo anche traccia della parola più frequente.

```
M <- mappa vuota;
maxCount <- 0; //si può inizializzare maxCount, ma non serve, visto che prende il valore della parola con più occorrenze
for i <- 0 to N-1 do{
```

```
        count <- M.get(D[i]);
        if(count = null) then count <- 0;
        M.put(D[i], ++count);
        if(count > maxCount) then{
            maxCount <- count;
            maxWord <- D[i];
        }
    }
    return maxWord
```

Complessità:

- $\Theta(N)$ operazioni (escluse le get e le put sulla mappa);
- N get e N put sulla Mappa, la cui complessità dipende dalla implementazione scelta per la Mappa;

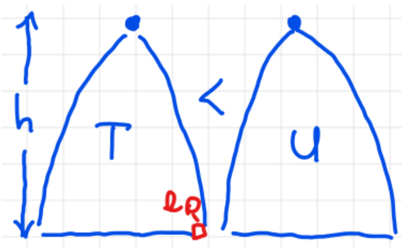
HashTable: N get e N put richiedono $\Theta(N(1+\lambda))$ operazioni al caso medio, dove $\lambda =$ load factor della tabella.
(2,4)-Tree/*Red Black Tree*: N get e N put richiedono $\Theta(N \log m)$ operazioni al caso pessimo, con m il numero di parole distinte in D.

6.9.3.3. Slide 30

> Siano T e U due `(2,4)-Tree` di altezza h e tali che la massima chiave in T è minore della minima chiave in U.
>
> 1. Progettare un algoritmo di complessità $O(h)$ per fondere T e U in un unico `(2,4)-Tree` TU;
> 2. Dire quali valori può assumere l'altezza di TU e se la radice contiene una o più entry.

Algoritmo `24TreeMerge(T,U)`
Input: (2,4)-Tree T e U di altezza h ciascuno e la massima chiave in T è più piccola della minima chiave in U.
Output: (2,4)-Tree TU fuzione di T e U.

Idea:



- $e =$ entry con chiave massima in T;

- metto e in un nuovo nodo r che diventa la radice del nuovo (2-4)-Tree TU e la rimuovo da T invocando `Delete`.

```
v <- T.root();
z <- figlio più a destra di v;
while(T.isInternal(z)) do {
    v <- z;
    z <- figlio più a destra di v
}
e <- entry in v con chiave massima
Crea un nuovo nodo radice r contenente e, con figlio sinistro T e figlio
destro U
TU <- (2,4)-Tree con radice r
TU.Delete(e,v)
return TU
```

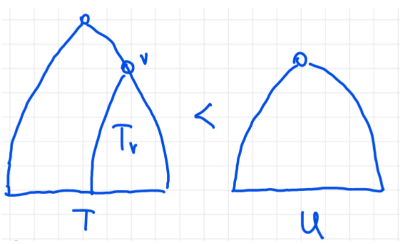
Complessità:

- ricerca di e: $\Theta(h)$
- creazione di TU e r: $\Theta(1)$
- `Delete`: $\Theta(h)$
  $\implies \Theta(h)$

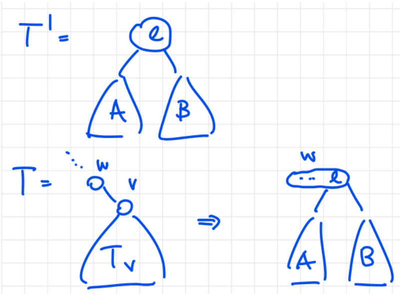
Se `Delete` propagasse l'underflow alla radice, allora l'altezza di TU rimarrebbe h, altrimenti l'altezza di TU sarebbe $h+1$ e la radice conterrebbe una sola entry.

6.9.3.4. Slide 37

> Siano T e U due (2,4)-Tree contenenti rispettivamente n ed m entry e tali che la massima chiave in T è minore della minima chiave in U. Progettare un algoritmo di complessità $O(\log n + \log m)$ per fondere T e U in un unico (2,4)-Tree TU.
> *Suggerimento*: Se i due alberi hanno altezze diverse, fondere l'albero di altezza minore con un opportuno sottoalbero dell'altro di uguale altezza (utilizzando l'algoritmo sviluppato per l'esercizio precedente).



Idea:

- Identifica il nodo v alla destra di T con la stessa altezza di U;
- Fondi T_v e U in un unico (2,4)-Tree T' con il metodo `(2,4)-TreeMerge`;
- Se T' ha la stessa altezza di T_v, allora sostituisco T_v con T';
- Altrimenti T' avrà l'altezza di $T_v + 1$ e dall'esercizio precedente so che in questo caso la sua radice ha una entry. Allora metto T' al posto di T_v fondendo la sua radice con il padre di v e invocando `Split` se il padre di v va in overflow.
  In questo secondo caso ecco cosa accade:
  

  Se w va in overflow con e, si invoca `Split`.

Aggiungere i dettagli come esercizio.