Lezione_20_DeA

6.4.5. 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 {
     sostituisce x con v;
     return x;
}
else{
     inserisci (k,v) in cosa al bucket A[h(k)];
     incrementa di 1 la size della tabella;
     return null;
}
```

Metodo `remove(k)`

```
if (∃ una entry (k,x) in A[h(k)]) then {
     rimuovi (k,x) da A[h(k)];
     deceremenda di 1 la size della tabella;
     return x;
}
else{
     return null;
}
```

6.4.5.1. Esercizio

> Creare una tabella hash di capacità $N = 11$ con le $n = 11$ chiavi: 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5. La funzione hash è: $h(k) = (3k + 5) \bmod 11$.

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

6.4.6. 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 impatti sgnificativo sulla efficienza computazionale di una tabella hash.

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

Studieremo la complessità al caso pessimo, in funzione del numero di entry presenti nella tabella hash, e la complessità al caso medio, in funzione del load factor λ della tabella hash.

Nota bene

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

6.4.7.1. Complessità al caso pessimo

Si consideri una tabella hash contenente n entry, 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` e `remove` è $\Theta(n)$ ed è *dominata dalla ricerca* della entry con chiave k, necessaria per tutti e tre i metodi.
Che sia $O(n)$ è banale, mentre che sia $\Omega(n)$ 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 $A[h(k)]$.

6.4.7.2. 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` e `remove` è

$$O(1+\lambda)$$

Tale complessità vale per:

1. 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;
2. Una chiave k presente: la media è fatta assumento 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.

6.4.7.2.1. Dimostrazione

Dimostriamo il punto (1), saltiamo la dimostrazione di (2).
Assumiamo che k non sia presente.
Sia n_i il numero di entry presenti nel bucket di indice i, $0 \leq i < N$, allora $\sum_{i=0}^{N-1} n_i = n$.

La complessità dei tre 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;
- $\Theta(n_i+1)$ è il numero di operazioni richieste per cercare k nel bucket i.

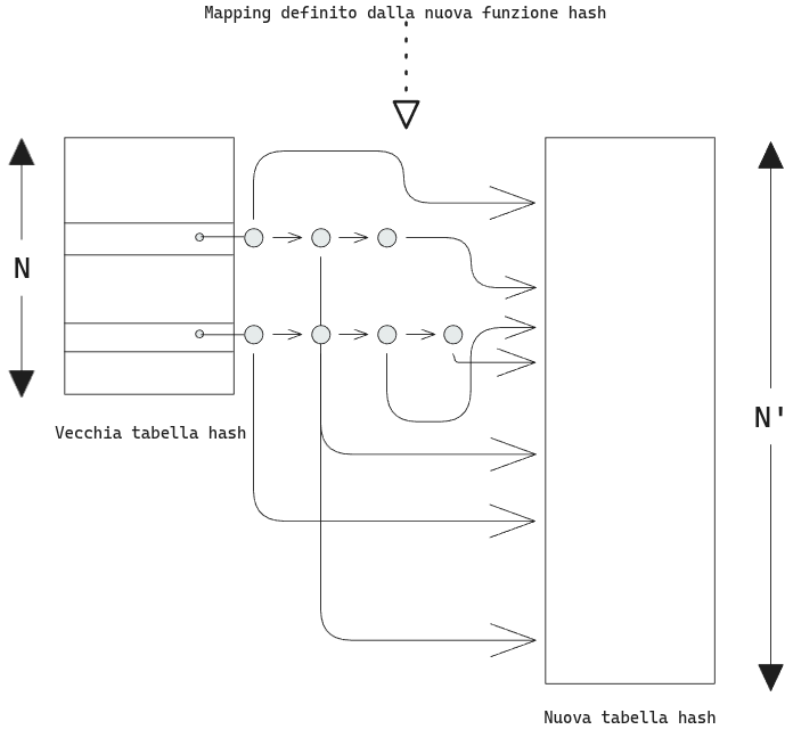
Si ha che:
$\frac{1}{N} \cdot \sum_{i=0}^{N-1}(n_i+1)) \frac{1}{N} \cdot \sum_{i=0}^{N-1} n_i + \frac{1}{N}\sum_{i=0}^{N-1} 1 = \frac{n}{n} + 1 = \lambda + 1$
□

6.4.8. 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.



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. Quindi il costo del rehash viene *ammortizzato* (ovvero nascosto) da quello aggregato dei `put`.

6.4.8.1. Dimostrazione

Giustifichiamo le osservazioni (1) e (2).

Per giustificare (1) diciamo che per spostare le entry da una tabella all'altra, posso eseguire n inserimenti nelle nuove tabelle *disabilitando* il controllo se ogni nuova chiave inserita è già presente, perché so già che le n entry da trasferire hanno tutte chiavi distinte. Allora il trasferimento delle n entry costa $\Theta(n)$ al caso pessimo.

Per giustificare (2) analizziamo un generico rehash: $N \to N_{new} \geq 2N$. Sia $n = \lambda N$ il numero di entry da trasferire. Consideriamo due casi:

- Caso 1: quello considerato è il primo rehash eseguito, allora le n entry sono state tutte inserite in precedenza, senza rehash e prima del rehash considerato;
- Caso 2: quello considerato non è il primo rehash nella vita della mappa. Consideriamo allora gli ultimi due rehash eseguiti: $N_{old} \to N \geq 2N_{old} \to N_{new} \geq 2N$. Il primo dei due rehash ($N_{old} \to N$) ha trasferito $n_{old} = \lambda N_{old}$ entry; il secondo dei due rehash ($N \to N_{new}$) ha trasferito $n = \lambda N$ entry.
  Allora tra i due rehash devono esserci stati almeno $n - n_{old}$ inserimenti e vale $n - n_{old} = \lambda N - \lambda N_{old} \geq \lambda N - \lambda \frac{N}{2} = @ł\frac{N}{2} = \frac{n}{2}$

Pro e contro delle Hash Table

Pro:

- Facile implementazione
- Buone prestazioni (al caso medio)
- Non richiede che le chiavi vengano da un universo ordinato

Contro:

- Complessità elevata al caso pessimo
- Alea dovuta alla bontà della funzione hash
- Spreco di spazio (per mantenere un basso load factor)

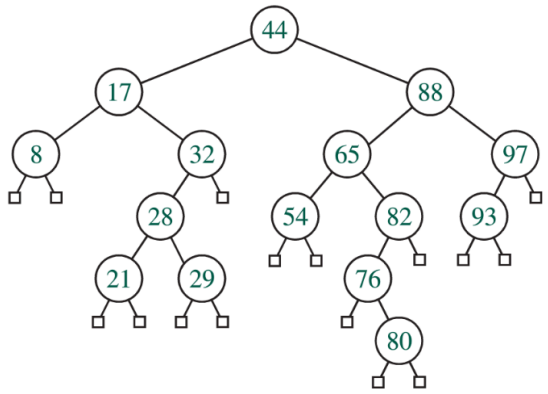
6.5. Alberi binari di ricerca

> Un *albero binario di ricerca* è un albero binario proprio i cui *nodi interni* memorizzano entry con chiavi proveinenti da un universo ordinato, tale che, per ogni nodo interno v la cui entry ha chiave k, le *chiavi nel sottoalbero* ***sinistro*** *di v sono* ***minori*** *di k*, mentre *le chiavi nel sottoalbero* ***destro*** di v sono ***maggiori*** *di k.*

Osservazione

I nodi foglia sono "sentinelle" che delimitano i confini dell'albero e possono essere implementati con puntatori `null` nei loro padri.

6.5.1. Esempio (solo chiavi)



> **Nota bene**
>
> La visita *inorder* tocca le entry in ordine non decrescente.

6.5.2. `TreeSearch` per un albero binario di ricerca T

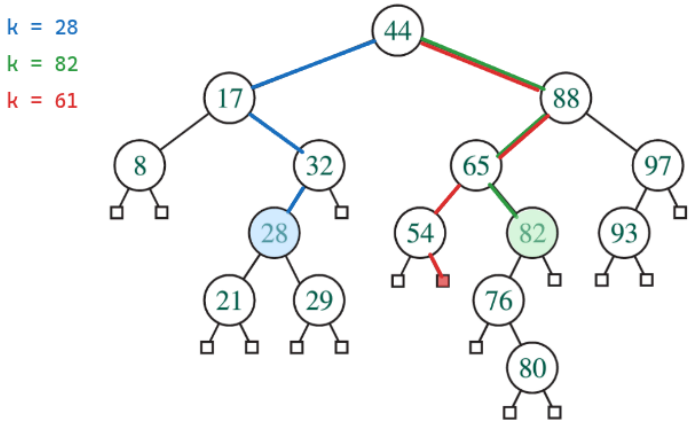
Algoritmo `TreeSearch(k,v)`
Input: chiave k, nodo $v \in T$.
Output: nodo di T_v con chiave k (se esiste) o foglia in posizione "giusta" per k.

```
if (T.isExternal(v) OR (v.getElement().getKey() = k)) then{
     rerurn v; //Caso base
}
if(k < v.getElement().getKey()) then{
     return TreeSearch(k, T.left(v));
}
else{
     return TreeSearch(k, T.right(v));
}
```

> **Osservazione**
>
> Una foglia "giusta" per k è una foglia che, se trasformata in un nodo interno, può contenere una entry con chiave k senza violare la proprietà di ABR.

La *chiamata iniziale* per ricercare in tutto l'albero ha `v = T.root()`.

6.5.2.1. Esempi



6.5.2.2. Complessità

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, olttre 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).
  $\implies$ Complessità $\Theta(h)$

> **Nota bene**
>
> 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 $\Theta(n)$.

6.5.2.3. Implementazione dei metodi della Mappa

6.5.2.3.1. `get`

Metodo `get(k)`

```
w <- TreeSearch(k, T.root());
if (T.isExternal(w)) then{
```

```
        return null;
    }
    else{
        return w.getElement().getValue();
    }
```

La *complessità* è $\Theta(h)$ perché è dominata dalla `TreeSearch`.