Lezione_19_DeA

5.7. Implementazione di Priority Queue in Java

Il package `java.util` contiene la classe `PriorityQueue<E>`, i cui oggetti sono Priority Queue Q implementate tramite heap.
Le sue caratteristiche sono:

- Q non contiene coppie `(chiave, valore)`, ma oggetti di tipo E;
- La priorità è definita utilizzando tutto l'oggetto come chiave, come se gli elementi di Q fossero le chiavi di entry vuote;
- Di default, gli elementi di Q sono confrontati in base all'ordine naturale associato al tipo E;

In pratica, se vogliamo usare la classe per realizzare una Priority Queue le cui entry sono coppie `(chiave, valore)` dobbiamo rappresentare le entry tramite una classe che estende l'interfaccia `Comparable`, definire il metodo `compareTo` imponendo un *ordinamento naturale delle entry* basato sulle chiavi.

5.7.1. Esempio

```
class MyEntry implements Comparable<MyEntry>{
     /* Variabili e metodi che definiscono la entry */
     @Override
     public int compareTo(MyEntry x) {

     /* Confronto tra chiamante e x basato sulla chiave */

     }
}
```

E creiamo la Priority Queue come segue: `PriorityQueue<MyEntry> Q = new PriorityQueue<MyEntry>();` .
A questo punto, il metodo `Q.poll()` restituirà la entry con chiave minima.

Riepilogo generale su Priority Queue

- Priority Queue: definizione e implementazione tramite liste;
- Albero binario completo: definizione e altezza;

- Heap: definizione, mapping efficiente su array tramite level numbering;
- Implementazione dei metodi della Priority Queue tramite heap (su array);
- Costruzione (top-down e bottom-up) di uno heap partendo da un array di n entry;
- pqSort
  - Implementazione con lista non ordinata (`SelectionSort`);
  - Implementazione con lista ordinata (`InsertionSort`);
  - Implementazione con heap su array (`HeapSort`).

---

6. Mappe

6.1. Definizione, interfaccia, applicazioni

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

Varie applicazioni sono:

- Database (Ed: studenti su Uniweb, con chiave il numero di matricola e come valore tutte le informzioni dello studente);
- Compilatori (Es: per il type checking di variabili, che hanno come chiave il nome della variabile, mentre come valore il tipo);
- Motori di ricerca (Es: le liste invertite, che hanno come chiave una parola, mentre come valore una lista di documenti, come le pagine web);
- Data analysis (Es: conteggio di frequenze di oggetti, che hanno come chiave un oggetto e come valore il numero di occorrenze).

> Una *Mappa* è una 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`.

```
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 esiste `(key, x)` restituisce `x`, altrimenti restituisce `null`;
- `put(K key, V value)`: se esiste `(key, x)` mette `value` al posto di `x` restituisce `x`, altrimenti inserisce l'entry `(key, value)` e restituisce `null`;
- `remove(K key)`: se esiste `(key, x)` rimuove la entry e restituisce `x`, altrimenti restituisce `null`;
- `keySet()`, `values()`, `entrySet()`: restituiscono strutture (Iterable) contenti, rispettivamente, le chiavi, i valori e le entry della mappa che possono essere enumerate da iteratori (iterator).

La *mappa* è vista come *associative array*, nel senso che la chiave della entry è usata come un "indice" di accesso alla mappa.

6.2. Implementazioni semplici, inefficienti

6.2.1. Lista non ordinata

Sia n il numero di entry nella Mappa.
Allora la complessità di `get`, `put` e `remove` è $\Theta(n)$, infatti tutti e tre i metodi richiedono la ricerca di una entry con la chiave data, se una tale entry non c'è devono guardarle tutte. Sono quindi *inefficienti* nel *tempo*.

6.2.2. Array di taglia |U|

Si assume che U sia finito e che sia disponibile un mapping 1:1 tra U e gli indici in $[0, |U| - 1]$. Tale mapping potrebbe non essere banale da trovare.
La complessità di `get`, `put` e `remove` p $\Theta(1)$.
Risulta *inefficiente* nello *spazio* se $|U|$ è molto più grande del numero di entry della mappa.

6.2.2.1. Esempio

Una Mappa per tutti i cittadini italiani ha $n \approx 6 \cdot 10^7$, la chiave può essere il codice fiscale (ovvero una stringa di 16 caratteri alfanumerici provenienti da un alfabeto di 36 caratteri). Allora ci sarebbe $|U| = 36^{16} \approx 8 \cdot 10^{24}$.

Il mapping tra U e gli indici in $[0, |U| - 1]$ si può ottenere venendo un codice fiscale come un intero rappresentato in base 36, usando un semplice mapping tra lettere/cifre e interi in $[0, 35]$.

```
0 1 2 3 4 5 6 7 8 9  A  B C  D ... Z
0 1 2 3 4 5 6 7 8 9 10 11 12 13... 25
```

> **Osservazione**
>
> L'universo delle chiavi *non è necessariamente ordinato*. In caso lo sia, si parlerebbe di *Mappa ordinata* ("Sorted Map" in inglese) e, in questo caso, sarebbero possibili implementazioni più efficienti al caso pessimo.

> **Osservazione**
>
> 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 sulle tabelle hash per la Mappa e sugli alberi di ricerca (binari e non) per la Mappa ordinata; discuteremo anche come implementare una Multimap.

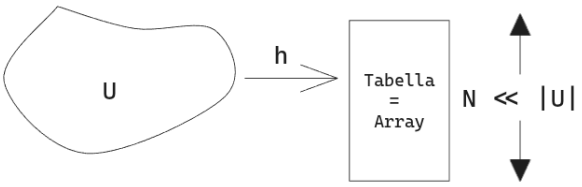
6.3. Mappe un Java

Nel package `java.util` troviamo:

- Interfaccia `Map<K,V>`, la quale contiene i metodi visti prima. Al suo interno è definita un'interfaccia *statica* `Map.Entry<k,V>`. `Map.Entry` è statica nel senso che è associata alla classe `Map` e non ai singoli oggetti della classe (in realtà tute le interfacce nested sono `static` per default). `Map.Entry` può essere usata ovunque l'interfaccia `Map` sia visibile.
- Classe `HashMap<K,V>`, che è l'implementazione di una Mappa tramite tabella hash con separate chaining.
- Classe `TreeMap<K,V>`, che è l'implementazione di una Mappa tramite Red-Black Tree.

6.4. Implementazione Mappa tramite Tabella Hash

6.4.1. Tabelle Hash



L'obiettivo è 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.

> Una Tabella Hash è definita dai seguenti tre ingredienti principali:
>
> - *Funzione hash* $h : U = \{\text{chiavi}\} \to [0, N-1]$.
>   La convenzione di Java stabilisce:
> $h : k \overset{\text{hash code}}{\longrightarrow} \mathbb{Z} \overset{\text{compression function}}{\longrightarrow} [0, N-1]$
> - *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.

L'idea è di 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 due due componenti) diventa cruciale. In particolare: h deve essere veloce da calcolare e 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 $\forall i, j \in [0, N-1]$ valga:

1. $Pr[h(k) = i] = \frac{1}{N}$: La probabilità che k sia assegnata al bucket $A[i]$ è la stessa per ogni $i \implies$ *non ci sono bucket "privilegiati" da h*;

2. $Pr[h(k) = i | h(k') = j] = Pr[h(k) = i] = \frac{1}{N}$: Sapere che k' è mappata sul bucket $A[j]$ non cambia la probabilità che k sia mappata sul bucket $A[i] \implies$ *h offusca possibili correlazioni tra chiavi*.

> **Osservazione**
>
> Le prestazioni di una Tabella Hash sono tanto migliori quanto più la funzione h assomiglia a una funzione random che soddisfa le due proprietà elencate.

> **Metodo HashCode in Java**
>
> Il metodo `hashCode()` della classe `Object` restituisce un `int` che dipende dall'indirizzo in memoria dell'oggetto. Non assicura quindi che l'hashCode di un oggetto rimanga inalterato in diverse esecuzioni di un programma o in diverse implementazioni di Java. Quensto non è un mapping "puro" da U a `int`.
> Il metodo `hashCode` può essere riscritto in vari modi a seconda del tipo di chiavi, restituendo sempre un `int`.

6.4.2. Generazione hash code

6.4.2.1. Hashcode 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`) $((k \gg 32)$ + (`int`) $k)$, dove "$k \gg 32$" rappresenta i 32 bit *più* significativi, mentre "(`int`) k" rappresenta i 32 bit `meno` significativi.

  > **Osservazione**
  > Solo "(`int`) k" perde l'informazione di metà dei bit

- `double` (64 bit) $k \rightarrow$ `Double.doubleToLongBits(k)` $\rightarrow$ `int`, dove il metodo indicato restituisce `long`, poi si effettua la trasformazione come per `long`.

6.4.2.2. Hashcode per una stringa di `char`

Sia $S \equiv s_0 s_1 \dots s_{k-1}$ una stringa di `char`.

- *Hash code banale*: $h(S) = \sum_{i=0}^{k-1} s_i$.
  Non è un buon hashcode, infatti $h(\text{stop}) = h(\text{spot}) = h(\text{tops})$.
- *Polynomial hash code*: $h(S) = \sum_{i=0}^{k-1} s_i \cdot a^{k-1-i}$.
  Per le parole inglesi vanno bene valori $a = 31, 33, 37, 39, 41$. La classi `String` in Java implementa hashCode in questo modo con $a = 31$.

6.4.2.2.1. Hashcode basato su cyclic shift

Nell'hashcode basato su cyclic shift si sommano i singoli caratteri applicando dopo ogni addizione un cyclic shift alla somma parziale. In pratica, uno shift di cinque posizioni va bene.

Calcolo dell'hash code:

```
h <- s_0 //h a 32 bit
for i <- 1 to k-1 do{
     h <- (h << 5) ! (h >> 27); //cyclic shift di 5 posizioni a sinistra
     h <- h + s_i;
}
return h;
```

6.4.3. Compression function

6.4.3.1. Division Method

Dato i l'intero prodotto dall'hashcode: $i \to i \bmod N$, dove N è la 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 due*.

Delle scelte di N poco adeguate sarebbero:

- $N = 2^p \implies i \bmod N$ equivale a prendere i p bit meno significativi di i, mentre è meglio che $i \bmod N$ dipenda da tutti i bit di i.
- $N = 10^p \implies i \bmod N$ equivale a prendere le p cifre meno significative di i in base 10. Come prima no dipende da tutta la rappresentazione di i in base 10.

6.4.3.2. Multiply-Add-Divide (MAD) Method

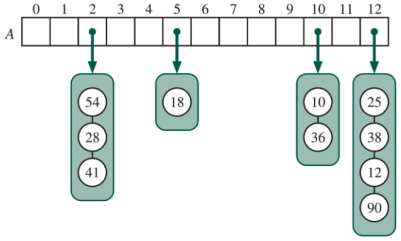
In questo caso si ha $i \to [(ai+b) \bmod p] \bmod N$, dove $p > N$ con p primo, $a, b \in [0, p-1]$ scelti a caso, $a > 0$.

> **Osservazione**
>
> Il metodo MAD, *leggermente più costoso dal punto di vista computazionale*, assicura una *migliore distribuzione degli hash code tra gli indici del Bucket Array*.

6.4.4. Risoluzione delle collisioni

> Una *collisione* avviene quando si hanno `<k1, v1>`, `<k2, v2>` con $k_1 \neq k_2$, ma $h(k_1) = h(k_2)$.

Nel *separate chaining* ogni bucket è visto come una `Map` più piccola, implementata tramite lista.



Nell'*open addressing* non si fa ricorso a strutture ausiliaria ma vengono memorizzate le entry direttamente nelle celle del Bucket Array. In questo modo si risparmia spazio, ma si complica la gestione delle collisioni (in caso di collisione, si usa una legge per cercare un'altra cella, occupando la prima cella libera). Non studieremo questo approccio.