Fondamenti di Informatica II

Esercitazione del 15 Aprile 2025 - Prof. Becchetti

Scopo dell' esercitazione è quello di realizzare una tabella hash per la memorizzazione di coppie di tipo `(String, int)`, esplorando due techniche per la risoluzione delle collisioni: liste di trabocco (chaining) e indirizzamento aperto. Per il secondo useremo scansione quadratica.

Nota bene: Le chiavi sono `String`, mentre per semplicità si assume che i valori siano `int` e *che non possano essere negativi*. La soluzione proposta può tuttavia essere estesa al caso in cui i valori siano oggetti qualsiasi, modificando leggermente il codice.

Attenzione: la cartella contiene lo scheletro delle classi da completare e un programma di prova (`Driver.java`). Non occorre creare nuovi file, ma soltanto completare le classi fornite. Si consiglia agli studenti di implementare i metodi nell'ordine suggerito di seguito.

Si tenga presente che i commenti presenti nel codice descrivono i) le funzionalità dei metodi che si chiede di implementare e ii) le loro interfacce (valori da restituire ecc.). Ulteriori commenti ai metodi più importanti sono dati di seguito.

Organizzazione del codice

Molte funzioni sono comuni alla soluzione con liste di trabocco e a quella con scansione quadratica. Per tale motivo, è stata definita una classe `AbstractHashTable.java`, contenente le implementazioni dei metodi che realizzano funzionalità comuni ai due approcci. Le classi `ChainHashTable.java` e `OpenHashTable.java` estendono `AbstractHashTable.java` con le implementazioni dei metodi che realizzano funzionalità che dipendono dal modo in cui le collisioni sono gestite. Infine, la classe `Entry` è una inner class di `AbstractHashTable.java` e rappresenta le coppie (chiave, valore) memorizzate nella tabella hash.

Task 1. Implementazione con liste di trabocco

Si vuole implementare una tabella hash usando liste di trabocco per la risoluzione delle collisioni. Per le funzioni hash: i) usare il metodo dei polinomi per generare l'hash code (implementato dal metodo `String.hashCode()` in Java); ii) usare una funzione hash universale per la compressione:

$$h(k) = ((ak + b) mod\ p) mod\ N,$$

dove N è la dimensione della tabella e $p > N$ è un primo, mentre a e b sono interi scelti uniformemente a caso in $[1, p-1],$ e $[0, p-1]$ rispettivamente. Nella nostra implementazione scegliamo 109345121 come valore di default per p.

Implementazione. Completare `AbstractHashTable.java` e `ChainHashTable.java`. Al fine di facilitare il test, si consiglia di procedere in questo modo:

1. Implementare il metodo `createTable()` di `ChainHashTable.java`, che permette di inizializzare una tabella vuota della dimensione desiderata. A tale scopo implementare prima il metodo `getCapacity()` di `AbstractHashTable.java`, che restituisce la dimensione (capacità) della tabella;

2. Implementare poi il metodo `hashFunction(String k)` di `AbstractHashTable`. Tale metodo implementa la funzione hash: data una chiave (`String`) restituisce un indice; per implementare tale metodo si consiglia di i) convertire la chiave in un intero usando il metodo `hashCode` della classe `String` e ii) usare una funzione hash universale come descritto sopra;
3. Implementare quindi il metodo `put(String k, int value)` di `ChainHashTable.java`, che gestisce l'inserimento della coppia (k, v) nella tabella (o l'aggiornamento del valore associato a k qualora quest'ultima sia già presente nella tabella);
4. implementare il metodo `entrySet()` di `ChainHashTable.java`. Tale metodo restituisce un oggetto che implementa l'interfaccia `Iterable` (ad esempio un oggetto di classe `LinkedList`) e che contiene una sequenza (ad esempio una lista) di tutte le coppie (chiave, valore) presenti nella tabella; per implementare tale metodo ci si serva anche del metodo `toString()` della classe `Entry`;
5. Implementare infine il metodo `print()` della classe `AbstractHashTable.java`. Ciò consentirà il test delle funzionalità di inserimento con piccoli insiemi di dati di input (teoricamente, fino alla massima capacità della tabella).

Si possono a questo punto implementare i restanti metodi pubblici comuni della classe `AbstractHashTable.java` e completare la classe `ChainHashTable.java`. Per il metodo `resize()` di `AbstractHashTable` (che si consiglia di implementare per ultimo), ci si può giovare dei metodi `entrySet()` e `put(k, v)`, che andranno implementati sia in `ChainHashTable.java` che in `OpenHashTable.java`.

Nota: a seconda della vostra implementazione, il metodo `resize()` potrebbe essere invocato all'interno di un'istanza della metodo `put`. Ciò non è necessariamente un problema.

Lo scheletro commentato delle classi `AbstractHashTable.java` e `ChainHashTable.java` segue per completezza. Il codice delle classi distribuite contiene commenti più dettagliati.

Classe `AbstractHashTable`

```
import java.util.Random;
import java.util.ArrayList;

public abstract class AbstractHashTable {
    private int capacity; // dim. tabella (N)
    private int n = 0; // numero di entry (coppie) nella tabella
    private int prime; // numero primo
    private long a, b; // coefficienti per MAD (Multiply Add and Divide)
    private double maxLambda; // fattore di carico massimo oltre il quale si ha resize

    // La classe Entry --> coppie (chiave, valore)
    class Entry {
        private String key;
        private int value;
        public Entry(String k, int v) {
            key = k;
            value = v;
        }
        // Metodi pubblici per accedere ai campi privati della classe Entry
        public String getKey() {
            return key;
        }
```

```
        public int getValue() {
            return value;
        }
        public void setValue(int v) {
            value = v;
            return;
        }
        public String toString() {
            return "(" + getKey() + ", " + Integer.toString(getValue()) + ")";
        }
    }

    // Costruttori della classe AbstractHashTable
    // Invocando AbstractHashTable() si assegnano valori di default ai parametri
    public AbstractHashTable(int cap, int p, double lambda) {
        capacity = cap;
        prime = p;
        maxLambda = lambda;
        Random gen = new Random();
        a = gen.nextInt(prime) + 1;
        b = gen.nextInt(prime);
        createTable();
    }
    public AbstractHashTable(int cap, int p) {
        this(cap, p, 0.5); // massimo fattore di carico predefinito
    }
    public AbstractHashTable(int cap) {
        this(cap, 109345121); // primo predefinito
    }
    public AbstractHashTable() {
        this(5); // capacità predefinita
    }

    // Metodi ausiliari comuni a tutte le classi

    // metodo che implementa la funzione hash (hash code + compressione)
    protected int hashFunction(String k) {
        return  0;
    }

    // metodo che aggiorna la dimensione della tabella hash (N)
    protected void resize(int newCap) {
        return;
    }

    // Metodi pubblici comuni a tutte le le classi

    // restituisce true se tabella vuota
    public boolean isEmpty() {
        return false;
    }

    // restituisce il numero di chiavi presenti nella tabella
```

```
    public int size() {
        return 0;
    }

    // restituisce la capacità della tabella
    public int getCapacity() {
        return 0;
    }

    // incrementa il numero n di chiavi presenti
    public void incSize() {
        return;
    }

    // decrementa il numero n di chiavi presenti
    public void decSize() {
        return;
    }

    // restituisce valore del max. fattore di carico
    public double getMaxLambda() {
        return 0.0;
    }

    // Stampa una rappresentazione delle coppire presenti secondo
    // il formato [(k1, v1),(k2,v2), ... ,(kn, vn)]
    public void print() {
        System.out.println("[]");
    }

    // Metodi astratti da implementare nelle sottoclassi
    protected abstract void createTable(); // inizializza la tabella hash
    public abstract int get(String k); // restituisce il valore associato alla chiave k
    public abstract int put(String k, int value); // inserisce/modifica un coppia
    public abstract int remove(String k); // rimuove la coppia con chiave k
    public abstract Iterable<Entry> entrySet(); // restituisce un Iterable contentente
// tutte le coppie presenti

}
```

Classe `ChainHashTable.java`

```
import java.util.LinkedList;
import java.util.ArrayList;

public class ChainHashTable extends AbstractHashTable {
    // Un array di LinkedList per le liste di trabocco
    private LinkedList<Entry> [] table; // table è l'array che implementa la tabella

    // Costruttori
```

```java
    public ChainHashTable(int cap, int p, double lambda) {
        super(cap, p, lambda);
    }
    public ChainHashTable(int cap, int p) {
        super(cap, p); // massimo fattore di carico predefinito
    }
    public ChainHashTable(int cap) {
        super(cap); // primo predefinito
    }
    public ChainHashTable() {
        super(); // capacità predefinita
    }

    // Metodi non implementati in AbstractHashTable

    // Inizializza una tabella vuota della dimensione desiderata
    protected void createTable() {
        return;
    }

    // restituisce il valore associato a una chiave se o -1 se la chiave non è presente
    public int get(String k) {
        return -1;
    }

    // inserisce la coppia nella tabella o modifica il valore della coppia con chiave k
    // Restituisce il vecchio valore se chiave già presente
    // Restituisce -1 se la chiave non è presente
    public int put(String k, int value) {
        return;
    }

    // elimina la coppia con chiave k, se presente
    public void remove(String k) {
        return;
    }

    // restituisce un oggetto Iterable sull'insieme delle coppie, ad esempio LinkedList
    public Iterable<Entry> entrySet() {
        return null;
    }

}
```

Task 2. Implementazione con scansione quadratica

Si vuole implementare una tabella hash, gestendo le collisioni mediante scansione quadratica (classe `OpenHashTable.java`). A tale scopo: i) il fattore di carico non verrà fatto crescere oltre il valore 0.5, onde garantire che un'eventuale scansione restituisca almeno una cella vuota nella quale inserire una nuova coppia; ii) si usa uno speciale oggetto `DEFUNCT` per contrassegnare celle vuote assegnate in precedenza, ma che vanno comunque esaminate durante la scansione quadratica (v. libro di testo/slide).

Nota: potrebbe essere utile implementare i) un metodo che trova la posizione in cui si trova la chiave k se presente e ii) un metodo che trova la posizione del prossimo slot disponibile per l'inserimento di una chiave k (se assente).

Classe `OpenHashTable.java`

```
import java.util.LinkedList;

public class OpenHashTable extends AbstractHashTable {
    // Un array di Entry per la tabella
    private Entry[] table;
    // marcatore di cella vuota ma da esaminare durante probing
    private final Entry DEFUNCT = new Entry(null, 0);

    // Costruttori
    public OpenHashTable(int cap, int p, double lambda) {
        super(cap, p, lambda);
    }
    public OpenHashTable(int cap, int p) {
        super(cap, p); // massimo fattore di carico predefinito
    }
    public OpenHashTable(int cap) {
        super(cap); // primo predefinito
    }
    public OpenHashTable() {
        super(); // capacità predefinita
    }


    // Metodi non implementati in AbstractHashTable
    protected void createTable() {
        return;
    }

    public int get(String k) { // restituisce -1 se chiave non trovata
        return -1
    }

    public void put(String k, int value) {
        return;
    }

    public int remove(String k) {
        return;
```

```
    }

    // Restituisce un Iterable sulle coppie, ad esempio di classe LinkedList
    public Iterable<Entry> entrySet() {
        return null;
    }

}
```

Task 3 (per casa)

Nell'attuale implementazione della scansione quadratica, il rehashing viene effettuato raddoppiando all'incirca la dimensione della tabella (se x è la vecchia dimensione, la nuova è $2x + 1$). Mentre ciò garantisce che la dimensione della tabella sia un numero dispari, questo non sarà necessariamente un numero primo. Tale condizione è tuttavia necessaria per essere certi che la scansione quadratica esplori almeno $N/2$ celle della tabella, garantendo la disponibilità di una cella per un nuovo inserimento, qualora il fattore di carico sia inferiore a 0.5. In pratica è improbabile che sorgano problemi, soprattutto al crescere delle dimensioni della tabella. Tuttavia, in linea di principio è possibile che la strategia di raddoppio adottata in questa implementazione dia luogo a errori (si cerca di effettuare un inserimento ma non si trova una cella libera).

Si risolva definitivamente il problema, modificando l'implementazione del metodo `put` in modo che la nuova dimensione della tabella non sia $2N + 1$, ma il primo intero p maggiore o uguale a tale valore (si consiglia di definire allo scopo un metodo privato). Ciò può essere effettuato in due modi:

1. Poiché la capacità è un intero rappresentato a 32 bit, si possono memorizzare in un array tutti i primi più grandi rappresentabili con i bit, per $i = 1, \ldots, 32$.
2. Alternativamente, si può implementare un algoritmo per la ricerca di numeri primi, ad esempio il crivello di Eratostene o algoritmi probabilistici più recenti e assai efficienti.

Probabilmente il primo metodo suggerito è sufficiente per i nostri bisogni.