Relazione primo progetto PR2 2017/2018 – Michele Zoncheddu

1. Obiettivo

Progettare e realizzare un'applicazione che determini la frequenza delle parole in un documento di testo, utilizzando due implementazioni dell'interfaccia DataCounter<E>: una basata su Hashtable<K, V> e una basata su TreeMap<K, V>.

2. Analisi del problema

L'applicazione dovrà produrre come output una tabella contenente le singole parole del documento, senza ripetizioni, ordinate per numero di occorrenze non crescente e, a parità di occorrenze, in ordine lessicografico.
Ho scelto di creare un'applicazione che consenta solamente l'analisi dei risultati, quindi non sarà consentita alcuna modifica delle strutture dati dopo l'elaborazione. È possibile tuttavia analizzare più documenti e affiancare le tabelle dei risultati.
La principale struttura astratta osservabile sarà una funzione che associa ad ogni parola il proprio numero di occorrenze all'interno del documento, con ulteriori informazioni come il numero di caratteri con e senza spazi, il numero di righe, di parole e il tempo di elaborazione del documento.

3. Sviluppo dell'applicazione

L'applicazione si basa su 6 classi e un'interfaccia:
HashDataCounter e ***TreeDataCounter*** sono le due implementazioni di DataCounter; possono essere istanziate con un qualsiasi tipo che estenda Comparable, condizione necessaria per permettere l'ordinamento: in questo caso specifico, il tipo utilizzato è String. Le stringhe in Java sono oggetti immutabili, quindi non ci sono problemi di aliasing. Il tipo dei valori legati alle chiavi sarebbe dovuto essere un qualunque tipo numerico, quindi la scelta iniziale è ricaduta su un generico che estendesse Integer, ma essendo Integer una classe posta alla fine della gerarchia di Number, è poco probabile che venga creato un sottotipo di Integer; quindi il tipo scelto è semplicemente Integer.
Oltre ai metodi richiesti dall'interfaccia, è stato aggiunto un metodo che restituisce il numero di parole senza contare i duplicati, in modo da evitare lo spreco di risorse che si avrebbe utilizzando strutture grandi quanto il numero totale di parole.
Come anticipato nella sezione 2, non sono stati implementati metodi modificatori, ad eccezione di incCount, richiesto dall'interfaccia. Ogni oggetto DataCounter sarà inizializzato nel costruttore, e rappresenterà un preciso documento di testo.

Le restanti quattro classi sono:

1. ***DataIterator***: un iteratore su DataCounter che restituisce gli elementi ordinati come richiesto. L'ordinamento è effettuato da una versione randomizzata del Quicksort, adattata per lavorare su due array (chiavi e relativi valori). DataIterator è in grado di lavorare sia su HashDataCounter che su TreeDataCounter;
2. ***TextAnalyzer***, che estrae le parole da un file, rimuovendo la punteggiatura, contando il numero di spazi, di righe, di caratteri, di parole, e calcolando il tempo di elaborazione. Sceglie inoltre quale implementazione di DataCounter utilizzare, attraverso un parametro gestito dai pulsanti dell'interfaccia grafica;
3. ***TextAnalyzerGUI***, che crea e gestisce l'intera interfaccia grafica (framework Swing), consentendo la scelta tra hash table e tree map, la scelta del file da analizzare e visualizzando una finestra con la tabella delle parole e relative occorrenze, e i risultati aggiuntivi dell'elaborazione;

4. ***Stats***, una classe wrapper di supporto, che consente il trasferimento dei risultati verso la GUI.

La maggior parte dei try-catch non sono stati scritti perché avrebbero cercato di catturare eccezioni che non sarebbero mai state lanciate. Sono presenti solo quelli che catturano eccezioni dovute a una cattiva interazione da parte dell'utente, come FileNotFoundException.

Nota teorica: l'applicazione sarà complessivamente più performante nel caso venga scelta la hash table, dato l'elevato numero di ricerche, tuttavia sarà richiesto un maggior quantitativo di RAM per via del load factor di 0.5, che garantisce il miglior compromesso tra prestazioni e costo in spazio.

4. Punti ostici e miglioramenti possibili

- Un punto molto impegnativo nello sviluppo dell'applicazione è stato la scelta di un invariante di rappresentazione e di pre/post condizioni dei metodi che fossero il più restrittivi e al contempo più chiari e funzionali possibili.

- Nella classe DataIterator, l'ordinamento sarebbe potuto avvenire in due modi alternativi a quello scelto:
  1. si sarebbe potuta utilizzare una classe wrapper contenente una chiave e il relativo valore, per sfruttare il metodo sort() di Java, utilizzando i campi di ogni singolo oggetto "contenitore" all'interno del metodo compareTo(Oggetto_1, Oggetto_2);
  2. se la classe chiamante fosse stata TreeDataCounter, avrei potuto sfruttare l'ordinamento parziale fornito dalla struttura ad albero per utilizzare un algoritmo di ordinamento stabile, in modo tale da ordinare solamente sul numero di occorrenze, senza perdere l'ordinamento alfabetico.

- Data la grande varietà di messaggi di errore, come eccezioni specifiche o scelta di file vuoti, non verranno visualizzati in apposite finestre di dialogo, ma sulla console, sono quindi visibili se si esegue il programma da riga di comando. Tuttavia questa si può considerare una versione ancora primitiva di un analizzatore di testi.

- Una funzionalità che renderebbe più semplice l'analisi di grandi documenti di testo è la ricerca di una parola: si potrebbe implementare attraverso una casella di testo nella finestra delle statistiche, evidenziando la riga corrispondente alla parola cercata, se si trova nella tabella, visualizzando un messaggio di errore altrimenti.

5. Compilazione e file di test

Il main-file da compilare è TextAnalyzerGUI.java.
Sono allegati 4 libri su cui testare l'applicazione, 3 dei quali tratti dalla pagina web del Progetto Gutenberg.