Relazione Primo Progetto Intermedio Programmazione II [Corso A]

Raffaele Apetino - N.M. 549220

Istruzioni per l'utilizzo del programma

Compilare sulla propria macchina i file sorgenti ed eseguire il comando "java Main". Da terminale vi verrà chiesto di inserire la *Path* dove è situato il file, ad esempio /Users/nomeutente/Desktop/Progetto/file.estensione. Per l'utilizzo dei metodi si faccia riferimento alle due interfacce DataCounter e FileOpenerInterface.

Hashtable o TreeMap?

Prima esporre le mie scelte è giusto sottolineare alcune osservazioni riguardanti le due strutture dati richieste e che ho implementato. Quando si utilizza una struttura di tipo albero come TreeMap avrà sicuramente tempistiche diverse da una struttura dati di tipo Hashtable. Questo perché sono implementate in modo diverso, entrambe però hanno pro e contro.
Infatti per eseguire la maggior parte delle operazioni (come l'inserimento di un nodo) su TreeMap impiegherà un tempo di $\boldsymbol{O(h)}$ dove h è l'altezza dell'albero riconducibile a $\boldsymbol{O(log_2 n)}$ quando tutte le chiavi sono differenti. A differenza di Hashtable, che riuscirà ad accedere all'indice in tempo costante $\boldsymbol{O(1)}$. Se esaminiamo file di testo molto ampi, con migliaia di parole sarebbe ottimo utilizzare una HashTable poiché l'inserimento, come dicevo prima, ha tempo costante. Osservando però che il metodo keySet di TreeMap restituisce le chiavi ordinate attraverso una visita dell'albero, dovendo noi lavorare sulle chiavi ordinate, possiamo trovare più utile l'utilizzo di TreeMap. Personalmente ho scelto di utilizzare HashTable poiché oltre ad essere *thread-checking* (poiché i metodi *put, get, contains* etc sono sincronizzati, mentre su TreeMap due thread possono accedere contemporaneamente alla struttura dati) l'ho trovato più performante nell'utilizzo nella nostra lingua. In cui molte parole (attraverso lo *stemming*) si ripetono numerose volte. Quindi andando a ridurre di molto il tempo di inserimento e il conseguente aumento del contatore. Essendoci quindi molte parole simili, il numero totale di parole complessive può diminuire e il costo dell'ordinamento delle chiavi HashTable sarà molto basso nell'ordine degli O-grandi. Più precisamente: il metodo $\boldsymbol{compare(Est1, Estr2)}$ implementato dall'interfaccia *Comparator* va eseguito sia sulla struttura dati ad albero che quella tabella hash, a differenza che sulla prima non va controllata la chiave in caso di contatore con valore uguale. Nel metodo compare di HashTable invece aggiungiamo una istruzione nel caso in cui i contatori siano uguali, siamo obbligati a controllare con il metodo *compareTo*, le chiavi. Il metodo *compareTo* ha una complessità relativamente bassa. Esso non ha bisogno di allocare in memoria i due elementi da comparare, perché ha accesso diretto alle due variabili impiegando nel caso peggiore $\boldsymbol{O(n)}$.

Interfaccia FileOpener

Ho preferito inoltre creare una interfaccia per l'apertura e la lettura del file in ingresso, così da rendere l'utente facilitato senza dover andare a cercare le istruzioni per la lettura di un file. Ci possono essere vari metodi per leggere un file di testo scelto dall'utente, ho preferito utilizzare *BufferReader* che è un potente lettore di stringhe che è in grado di leggere i file di testo riga per riga. A questo punto avevo disponibili due scelte: lavorare su una stringa alla volta ed elaborarla, oppure salvare l'intero file in una stringa e avere una singola chiamata di funzione. Ho optato per utilizzare maggiore memoria e meno chiamate di funzioni. La complessità asintotica è la stessa in entrambi i casi.

Batterie di test

Infine i vari test case che ho applicato al mio programma. Prima di tutto ho voluto testare se la mia precedente scelta di salvare l'intero file di testo in una singola stringa si fosse rivelata una scelta di progetto errata, quindi ho preso un file molto grande contenente l'Inferno della Divina Commedia (*divina_commedia.txt*). Altro test case è il file vuoto, non contenente alcuna parola (*empity.txt*). Seguono i test case che comprendono numerose ripetizioni di parole (*aroundtheworld.txt*) e numerosi simboli come la home page di una pagina HTML (*index.html*). *file.txt* contiene l'esempio contenuto nella consegna mentre *password.txt* contiene stringhe alfanumeriche.

Conclusioni

Grazie alla struttura di java e l'astrazione sui tipi è possibile utilizzare DataCounterImpl su qualsiasi tipo di oggetto. L'utente quindi definirà l'oggetto che sarà la chiave mentre per il contatore è vincolato ad usate un oggetto di tipo intero. Inoltre utilizzando la classe FileOpener è facilitato nell'interfacciarsi con file di testo preso dall'esterno. Chiamando quindi il metodo readFile() in automatico verrà richiesta la *Path* del file ed elaborato così da ottenere una struttura dati che contenga tutte le parole contenute normalizzate (eliminata punteggiatura e caratteri speciali).