Relazione progetto Algoritmi e Strutture Dati

IMPLEMENTAZIONE DIZIONARIO IN LINGUAGGIO C CON RED-BLACK TREE

31 Maggio 2017

Nicolas Farabegoli
Paolo Baldini
Università di Bologna - Ingegneria e Scienze Informatiche Cesena
nicolas.farabegoli@studio.unibo.it
paolo.baldini@studio.unibo.it

Indice

1 Requisiti del progetto

L'obiettivo del progetto è quello di costruire e gestire un vocabolario contenente le voci, ordinate, e le corrispondenti definizioni. Il dizionario può essere inizialmente generato a partire da un file di testo. In questo caso si vogliono memorizzare tutte le parole presenti in un file di testo in ordine lessicografico.

Si utilizzi per rappresentare il dizionario la struttura dati che si ritiene più opportuna, dopo una attenta analisi. Si possono utilizzare le strutture dati astratte affrontate dal corso (alberi, hash table, code, pile, liste, grafi, ...).

Il file contiene un testo le cui parole non sono memorizzate in ordine lessicografico, e sono separate da spazio. I vocaboli sono inizialmente salvati senza la corrispondente definizione; la definizione, una stringa di dimensione sconosciuta, sarà inserita in un secondo momento, tramite la funzione apposita.

Ogni voce del dizionario contiene al massimo 20 caratteri, e minimo 2, e le definizioni inserite ne contengono al massimo 50.

1.1 Vincoli

Nel dizionario non devono essere inserite le parole di un solo carattere o con più di 20 caratteri. Inoltre non devono essere inserite parole che differiscono solo per caratteri minuscoli/maiuscoli. Le parole e le definizioni inserite nel dizionario non devono presentare caratteri maiuscoli; nel caso di inserimento di una parola con un carattere, o più, maiuscoli, convertirli per inserirli nel modo corretto.

Si assume che nel file di testo e nel dizionario non sono presenti nomi propri (di persona) e le voci nel dizionario possono ammettere caratteri accentati (è, à, etc...), ma non ammettono cifre e caratteri di punteggiatura.

1.2 Funzionalità richieste

Sono presenti metodi di gestione del dizionario come il conteggio delle voci del dizionario, l'inserimento o la cancellazione di un nuovo vocabolo, ... Esistono due diverse funzioni di ricerca:

- la prima, di base, dato un termine restituisce, se presente, la sua definizione
- la seconda è una funzione di searchAdvance che verifica la presenza della parola da cercare e restituisce le tre parole più simili a quella che si vuole cercare. Per similarità si intende il minor numero di modifiche (sostituzioni, inserimenti) per trasformare una parola nell'altra.

Le specifiche della ricerca avanzata, e l'algoritmo da utilizzare, sono a scelta dello studente. Inoltre si vuole avere la possibilità di caricare e salvare il dizionario, a partire dalla struttura dati ottenuta, con il seguente formato, in un file testuale (**.txt**):

```
"che":  [(null)]

"classe:  [(null)]

"come":  [nc]

"determinato":  [ulz u]

"di":  [(null)]
```

Oltre al salvataggio classico è richiesto di sviluppare il salvataggio del file compresso, quindi con un ridotto quantitativo di spazio in memoria, utilizzando la tecnica di compressione di Huffman. Il vocabolario è compresso a partire dallo stesso formato del normale salvataggio ("che": [(null)] ...).

Oltre al salvataggio del file compresso, si vuole avere la possibilità di caricare il dizionario compresso da un file di test (decompressione), in modo tale da poterlo visualizzare e/o rielaborare successivamente. Il testo ottenuto dalla decompressione è dello stresso formato del dizionario salvato in memoria. La funzione deve produrre in uscita la struttura dati vocabolario scelta.

2 Aspetti del progetto

2.1 Componenti del gruppo

Il gruppo si compone di due persone: Paolo Baldini (`801207`), Nicolas Farabegoli (`788928`).

2.2 Suddivisione del lavoro

Sono state assegnate 13 funzioni per lavorare sulla struttura dati; l'implementazione delle suddette funzioni è stata ripartita nel seguente modo:

- `createFromFile()` - Nicolas Farabegoli
- `printDictionary()` - Paolo Baldini
- `countWord()` - Paolo Baldini
- `insertWord()` - Nicolas Farabegoli
- `cancWord()` - Nicolas Farabegoli
- `getWordAt()` - Paolo Baldini
- `insertDef()` - Nicolas Farabegoli
- `searchDef()` - Nicolas Farabegoli
- `saveDictionary()` - Nicolas Farabegoli
- `importDictionary()` - Nicolas Farabegoli
- `searchAdvance()` - Paolo Baldini
- `compressHuffman()` - Paolo Baldini
- `decompressHuffman()` - Paolo Baldini

2.3 Tools di sviluppo

I tools di sviluppo che sono stati utilizzati sono: Visual Studio Enterprise 2015 (Editor + compilatore) per effettuare il test in ambiente Windows, mentre VIM (Editor) e GCC (compilatore) per il test in ambiente Linux.

I test del software in ambiente Windows sono stati effettuati su Windows 10 Pro, invece i test in ambiente linux sono stati effettuati utilizzando Arch Linux, Kernel 4.11.3-1 e GCC 7.1.1.

2.4 Deadline

Il progetto viene terminato per il primo appello di Giugno, si stima quindi come data di consegna del progetto (in accordo con le scadenze imposte dal docente) il 11 giugno 2017.

3 Analisi del progetto

3.1 Scelta struttura dati

A seguito delle specifiche del progetto ci è stato richiesto di effettuare un'attenta analisi sulla struttura dati da utilizzare per realizzare il dizionario. In un primo momento si era pensato di adottare come struttura dati una tabella Hash; questa però è risultata sin da subito non efficiente all'implementazione di un dizionario: come prima cosa avremo dovuto creare una funzione Hash ad hoc estremamente efficiente e che gestisse le *collision* in maniera intelligente, non ci è sembrata una struttura dati adatta al contesto del progetto proprio per il fatto che le tabelle Hash generano collision. Una possibile soluzione pensata per gestire le collision era quella di creare una lista in corrispondenza di una collision. Un'approfondita analisi circa il problema delle collision ci ha portati a riflettere sui costi computazionali: gestendo le collision con una lista, il *worst case* sia dell'inserimento che della ricerca nel dizionario, avrebbero avuto un costo pari a $\Theta(n)$.

Questo lo si può facilmente intuire dal fatto che: supponendo di avere una funzione Hash che crei un numero di collision molto grande in un unico punto, troveremo la lista corrispondente a quel punto con lunghezza (in termini di ordini di grandezza) pari a n. Quindi sia l'inserimento (in ordine lessicografico) che la ricerca, hanno come costo computazionale pari a $\Theta(n)$.

Ci siamo quindi concentrati su strutture dati che abbiano costi computazionali al più lineari $\Theta(n)$. Ci siamo quindi orientati sugli alberi binari di ricerca (BST). Abbiamo quindi iniziato a stimare i costi computazionali per le funzioni di maggiore rilievo sulla struttura dati, come inserimento, cancellazione, ricerca e visita. Dalle nostre stime è emerso che: per *inserimento*, *cancellazione* e *ricerca* il costo computazionale è $\Theta(log(n))$ (grazie alle proprietà che caratterizzano i BST), mentre per la *visita* il costo è $\Theta(n)$.

Un'ulteriore analisi sulla struttura dati ci ha condotti ad osservare che gli alberi binari di ricerca non erano ancora la soluzione ottimale per il problema: infatti i costi stimati in precedenza fanno riferimento al caso in cui l'albero binario sia bilanciato. Infatti se ciò non fosse, i costi che prima erano stimati a $\Theta(log(n))$, nel caso di un albero fortemente sbilanciato sono $\Theta(n)$. Ritornando alle problematiche precedentemente riscontrate con le tabelle Hash. Non potendo prevedere a posteriori il bilanciamento dell'albero e considerando sempre il caso peggiore, abbiamo appurato che i BST non sono ancora la struttura dati ottimale per implementare un dizionario.

La chiave per ottenere una struttura dati che ci consenta di effettuare le principali operazione in tempo sub-lineare e al più lineare è quella di riuscire a bilanciare l'albero. Abbiamo quindi pensato ai Red-Black Tree (RBT), i quali ci garantiscono un buon bilanciamento dell'albero, con conseguente garanzia sui costi computazionali stimati nei casi peggiori. Infatti nel *worst case* abbiamo per *inserimento*, *cancellazione* e *ricerca* un costo pari a $\Theta(log(n))$, mentre per la *visita* rimane sempre $\Theta(n)$. La struttura dati utilizzata nel progetto è il **Red-Black Tree**.

3.2 RBT e BST a confronto

Come accennato alla sezione precedente, i BST sono fortemente inclini a sbilanciarsi, ciò causa un'alterazione nei costi computazionali a tal punto che l'ordine di grandezza si sposta da sub-lineare a lineare. Questo è un problema da tenere in considerazione quando si lavora con un numero di elementi molto elevato come può essere un dizionario. Il problema del bilanciamento è praticamente risolto adottando i Red-Black Tree, poiché la politica di inserimento e cancellazione fa si che l'albero sia fortemente bilanciato, riuscendo, nel caso peggiore, ad avere un costo computazionale pari a $\Theta(2log(n))$.

3.3 Stime costi computazionali: algoritmi a confronto

Algorithm	Average case	Worst Case
Hash Table	$\Theta(xxx)^*$	$\Theta(xxx)^*$
BST	$\Theta(log(n))$	$\Theta(n)$
RBT	$\Theta(log(n))$	$\Theta(2log(n))$

Tabella 1: Inserimento nel dizionario

Algorithm	Average case	Worst Case
Hash Table	$\Theta(xxx)^*$	$\Theta(xxx)^*$
BST	$\Theta(log(n))$	$\Theta(n)$
RBT	$\Theta(log(n))$	$\Theta(2log(n))$

Tabella 2: Cancellazione nel dizionario

Algorithm	Average case	Worst Case
Hash Table	$\Theta(xxx)^*$	$\Theta(xxx)^*$
BST	$\Theta(log(n))$	$\Theta(n)$
RBT	$\Theta(log(n))$	$\Theta(2log(n))$

Tabella 3: Ricerca nel dizionario

(*) I costi computazionali relativi alla tabella hash sono fortemente influenzati dalla funzione hash realizzata.

¡ Come si evince dalle tabelle l'algoritmo che presenta il miglior costo computazionale è il **RBT**.

4 Costi computazionali funzioni implementate

4.1 Funzioni implementate da `Nicolas Farabegoli`

`insertWord(NODO** root, NODO* node)` - Il costo computazionale nel caso peggiore della funzione per l'inserimento di un nodo è piuttosto semplice da calcolare in quanto nel caso peggiore bisognerà arrivare alle foglie e quindi percorrere un numero di nodi pari all'altezza dell'albero. Come detto nei paragrafi precedenti, l'altezza di un albero nei RBT nel caso peggiore, è pari a $2log_2(n)$ dove n sono il numero di nodi presenti nell'albero.

`cancWord(NODO** dictionary, char* word)` - La funzione di cancellazione di una parola all'interno del dizionario effettua due chiamate ad altre due funzioni: `searchWord` per ricercare la parola da cancellare e `rb_delete` per eliminare il nodo contenente la parola cercata. La funzione `searchWord` al suo interno effettua due chiamate ricorsive condizionate: una sul figlio di destra se la parola da cercare ha peso maggiore, oppure a sinistra se la parola ha peso minore. Possiamo quindi impostare un'equazione ricorsiva piuttosto semplice:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(\frac{n}{2}) & n > 1 \end{cases}$$

L'equazione ricorsiva produce come risultato un costo computazionale pari a $\Theta(log(n))+k$. L'altra funzione presente si occupa semplicemente di rimuovere dall'albero il nodo contente la parola, questa funzione non presenta cicli e le uniche operazioni che fa sono qualche rotazione e ricolorazione, quindi operazioni con un costo computazionale pari a $\Theta(k)$. Possiamo quindi affermare che il costo computazionale per la funzione `cancWord(...)` è pari a $\Theta(log(n) + k)$.

`insertDef(NODO* dictionary, char* word, char* def)` - La funzione si occupa di ricercare la parola corretta ed aggiungere la definizione. La funzione al suo interno effettua una chiamata alla funzione `searchWord(...)` che, come accennato nel paragrafo precedente, ha un costo computazionale pari a $\Theta(log(n))$. Le altre operazioni che svolge la funzione sono per l'inserimento della definizione; queste operazioni sono costanti e non dipendono dalla dimensione della struttura dati quindi hanno costo costante. Concludendo, il costo computazionale della funzione è pari a $\Theta(log(n))$.

`searchDef(NODO* dictionary, char* word)` - La funzione si occupa di ricercare la parola passata come argomento e stampare a schermo la sua definizione. La funzione al suo interno richiama `searchWord(...)`. Quindi il costo computazionale è pressoché identico alla funzione illustrata in precedenza, quindi ha costo computazionale pari a $\Theta(log(n))$.

`saveDictionary(NODO* dictionary, chaar* fileOutput)` - La funzione si occupa di salvare il dizionario in memoria nel PC tramite un file. Al suo interno la funzione, oltre a creare il file di destinazione, chiama la funzione `printDictionaryFile(...)` che è strutturata così: vengono effettuate due chiamate ricorsive sul figlio di destra e quello di sinistra del sottoalbero. Questo significa che i nodi dell'albero vengono visitati una ed una

sola volta, in questo caso non si ritiene necessario dover risolvere un'equazione ricorsiva in quanto la visita di ogni nodo è costante per ognuno di essi, il costo computazionale di questa funzione è $\Theta(n)$. Come detto sopra, viene creato un file per il salvataggio, supponiamo che questa operazione sia costante; essendo anche l'unica altra funzione presente il costo computazionale finale della funzione è: $\Theta(n)$.

`importDictionary(char* fileInput)` - La funzione si occupa di leggere il file generato dalla funzione `saveDictionary(...)` e di creare il dizionario. La funzione al suo interno si compone di un ciclo per la lettura del file che termina con la presenza del carattere `EOF`, funzioni per estrapolare la parola e relativa definizione (che consideremo con costo computazionale costante ai fini del costo finale della funzione) e la funzione `insertRBT` che consente di inserire il nodo nell'abero. Quest'ultima funzione presenta costo computazionale pari a $\Theta(2log(n))$ poichè sarà necessario scorrere tutto l'albero fino alle foglie. Siccome questa funzione viene eseguita per ogni nome, il costo computazionale globale della funzione è pari a $\Theta(nlog(n))$.

4.2 Funzioni implementate da `Paolo Baldini`

`countWord(NODO* dictionary)` - La funzione restituisce il numero di elementi presenti all'interno del dizionario. Questa si comporta come una visita in-order, riportando a ogni ciclo ricorsivo il numero delle parole nel sotto-albero controllato. A queste andrà sommato il numero delle parole dell'altro sotto-albero più una rappresentante il nodo controllato. Essendo una visita in-order il costo computazionale della funzione è $\Theta(n)$.

`printDictionary(NODO* dictionary)` - La funzione si occupa di stampare il dizionario. Come nel caso di Count Word, si tratta di una semplice visita in-order, seguita da una stampa; per cui anche qui la complessità sarà $\Theta(n)$.

`getWordAt(NODO* dictionary, int index)` - Se presente, la funzione restituisce il puntatore alla parola cercata, altrimenti restituisce NULL. La funzione inizializza le variabili e si appoggia a una sotto-funzione per la ricerca della parola all'interno del dizionario. La sotto-funzione è stata inserita per poter lavorare su una variabile inizializzata dalla funzione "madre", senza dover quindi restituire indici di sorta attraverso il return, cosa che avrebbe potuto creare problemi di identificazione fra puntatore/indice in caso d'errore. Non era neanche possibile utilizzare una variabile statica in quanto avrebbe creato problemi in una successiva chiamata alla funzione. Computazionalmente parlando, la sotto-funzione si comporta, nel caso peggiore possibile, come una visita in-order, mentre la funzione "madre" ha semplicemente un ruolo di inizializzazione, quindi costo k, che può essere trascurato. Il costo complessivo della funzione è quindi pari a $\Theta(n)$ nel caso peggiore.

`searchAdvance(NODO* dictionary, char * word, char ** first, char ** second, char ** third)` - Questa funzione restituisce 0 nel caso trovi la parola cercata e -1 nel caso non la trovi. Internamente modifica anche tre puntatori alle parole più "simili" a quella cercata. La similitudine è data dalla funzione di Levenshtein. Il codice di quest'ultima funzione è stato ispirato da varie ricerche ma adattato e ri-progettato secondo le necessità.

La funzione lavora mantenendo in memoria le varie distanze e parole e aggiornandole nel caso trovi una nuova parola più "vicina" a quella cercata. Nel complesso anche questa funzione si comporta come una visita in-order. Il costo è dato dalla visita $\Theta(n)$ più il costo della funzione di Levenshtein. Questa lavora su due stringhe m ed n e il suo costo è pari length(m)*length(n). Considerando le due stringhe di lunghezza massima di 20 caratteri (specifiche del progetto) pari ad m, avremmo un costo pari a m^2, e quindi un costo totale di Search Advance pari a $\Theta(n(m^2))$. C'è però da considerare che m non può eccedere i 20 caratteri, il che la porta a poter essere considerata una costante.

`compressHuffman(NODO* dictionary, char* file_name)` - Il compito di questa funzione è di comprimere il file per fargli occupare meno spazio. La funzione crea un albero delle codifiche basato sulla frequenza standard dei caratteri nel dizionario italiano (reperita su internet). Questa scelta è stata effettuata per permettere la decodifica anche dopo la chiusura/riapertura del file. Se l'albero fosse stato creato secondo la frequenza all'interno del dizionario, si sarebbe richiesto il salvataggio di un file aggiuntivo per la memorizzazione delle codifiche. Considerando che la frequenza media delle lettere italiane in un testo di grandi dimensioni si può considerare accurata, abbiamo optato per utilizzare quella. Per evitare errori di sorta le lettere accentate sono state convertite in lettere normali. La codifica di ogni lettera generata dall'albero viene poi salvata all'interno di una "tabella". Ogni lettera dispone di una codifica lunga fino a 18 caratteri. La scelta di limitare la lunghezza dei caratteri è stata fatta per evitare di snaturare l'essenza stessa della compressione di Huffman, che consiste appunto nel ridurre al minimo le dimensioni delle codifiche. L'algoritmo di creazione della tabella è stato costruito su quest'assunto. Una volta costruita la tabella delle codifiche, l'algoritmo si preoccupa di trasformare ogni lettera di ogni stringa di ogni nodo del dizionario in una sequenza di bit, che verranno scritti a coppie di 8 (quindi 1 byte) nel file di salvataggio. Il costo della funzione di Huffman è dato dalla somma del costo per la creazione dell'albero (considerando un numero di caratteri limitato è approssimabile a una costante), al costo del riempimento della tabella (anche qui costante) e al costo della scrittura su file di ogni nodo. La visita di ogni nodo ci costerà $\Theta(n)$, e per ognuno di questi dovremo andare a codificare la stringa della parola e quella della definizione. Considerando la lunghezza massima di parola più definizione k = 20 + 50, e considerando che per ognuno dei caratteri dovremo scrivere la codifica su file, otteniamo che il costo computazionale generale della funzione di compressione di Huffman sarà $\Theta(n * k)$. Con k limitato superiormente a 70, il costo si può scrivere come $\Theta(70 * n) = \Theta(n)$, quindi lineare, anche se con costante moltiplicativa molto grande.

`decompressHuffman(char* file_name, NODO** dictionary)` - La funzione si occupa di decomprimere il file compresso dalla funzione di compressione di Huffman. Anche la decompressione richiede la costruzione dell'albero delle codifiche, necessario per riconoscere i vari caratteri. La funzione si concentra sul leggere i byte (e quindi i bit della codifica) dal file e a scorrere l'albero di Huffman fino alle foglie per identificare la lettera; successivamente compone la stringa e alloca un nodo, contenente le parole e le definizioni del dizionario, che andrà poi inserito nel RBT. Nei vari test si è evinto come alcuni caratteri particolari dessero alcuni problemi in lettura. Per questo, ad esempio, non è

stato usato un while ($!EOF$) ma è stato creato un carattere terminatore personalizzato. I caratteri di controllo, nei vari test, non hanno presentato anomalie di sorta. La funzione, dovendo gestire l'importazione di un dizionario prestabilito, non accetta in input un dizionario che già contenga parole, per questo si preoccupa di svuotarlo ad ogni chiamata. Questa funzione è stata implementata con lo scopo di evitare di mischiare più dizionari diversi in memoria. Sarebbe stato anche possibile aggiungere semplicemente le parole "in più", ma abbiamo valutato che importare il dizionario così come era stato salvato fosse una procedura più "pulita" e corretta. In ogni caso, volendo aggiungere le parole in più, basterebbe eliminare la funzione che si preoccupa di svuotare il dizionario (5 righe) e aggiungere una condizione che controlli (ad esempio con la searchAdvance) che la parola non sia già presente nel dizionario prima di inserirla. Il costo della funzione dipende dal numero di caratteri codificati nel file. Il costo di questa funzione si può dire quindi linearеal numero di caratteri. Essendo ogni stringa formata da minimo 20 caratteri, il costo pessimo sarebbe quindi $\frac{n}{20}$. Le stringhe saranno poi inserite nel nodo che a sua volta sarà inserito nel RBT. Questo inserimento nell'albero comporta un costo computazionale pari a $log(n)$. Il costo complessivo della funzione sarà quindi $\Theta((\frac{n}{20})nlog(n)) = \Theta(nlog(n))$.

5 Riflessioni e considerazioni sul progetto

5.1 Difficoltà incontrate

Tra le difficoltà più grandi incontrate nel progetto troviamo la creazione delle funzioni relative alla manipolazione della struttura dati principale (RBT); per esempio la funzione per l'inserimento di un nodo nell'albero ha creato non poche difficoltà per il fatto che non riusciva a rispettare le politiche dei RBT. La funzione di cancellazione di un nodo (probabilmente per la sua difficoltà intrinseca) è stata piuttosto complicata da implementare e farla funzionare correttamente. In molte funzioni, la stesura del codice non si è rivelata di per se complicata, piuttosto si è rivelata a momenti problematica la risoluzione e l'individuazione di piccoli bug che creavano qualche problema in alcuni sottoprogrammi. Grazie comunque a lunghe sessioni di debuging sono stati per la maggior parte eliminati.

5.2 Considerazioni finali

Grazie al progetto abbiamo potuto consolidare le nostre conoscenze circa il funzionamento e le proprietà legate ai RBT. Abbiamo messo in campo le nostre migliori competenze per poter affrontare e terminare nel migliore dei modi il compito assegnato e poter confrontarci per migliorare o talvolta sistemare/segnalare le problematiche incontrate durante la stesura del codice. Gli strumenti utilizzati per la realizzazione del progetto hanno indubbiamente favorito l'identificazione e di conseguenza la correzione di bugs che impedivano il corretto flusso e coerenza del programma. Grazie a questo progetto siamo riusciti ad analizzare tutti gli aspetti (o quasi) del problema e abbiamo sempre cercato le soluzioni maggiormente efficienti per implementare le funzioni date.