PROGETTO ASD A.A. 2016/17

Gruppo:

Giulianini Andrea *0000801585*

Lombardi Alessandro *0000789058*

SO e IDE utilizzato:

Windows, Visual Studio 2015

Data consegna:

13/06/2017

Abbiamo scelto di implementare un **Red-black Tree** per la gestione del dizionario in quanto molte operazioni possono essere eseguite in un tempo $O(log(n))$, con *n* uguale al numero di elementi dell'albero. In particolare il Red-black Tree è una struttura dati di tipo Albero binario di ricerca, che però sfrutta determinate proprietà per mantenere l'equilibrio ed evitare la spiacevole situazione in cui l'albero diventi particolarmente sbilanciato. Lo spazio in memoria occupato dal Red-black Tree è pari al numero di nodi di cui è formato.

Nella prima parte del file lib1617.c abbiamo implementato le funzioni standard, seguendo l'implementazione da letteratura, per la gestione dei BST. I costi delle funzioni di ricerca (minimo, massimo, predecessore, successore e nodo generico) hanno tutti un costo $O(h)$, dove *h* è l'altezza dell'albero. Segue poi l'implementazione delle funzioni elementari sui RBT, inserimento e cancellazione che hanno sempre costo logaritmico.

TEOREMA: Un RBTree con *n* nodi ha sempre una altezza $O(log(n))$.

Le funzioni *searchDef()* e *insertDef()* hanno costo $O(log(n))$ con n pari al numero di nodi.

L'algoritmo *searchAdvance()* scorre tutto l'albero partendo dal nodo "più piccolo" aggiornando man mano il risultato, la logica con la quale viene scelta una parola piuttosto che un'altra si basa sulla **distanza di Hamming**. La differenza fra due parole cresce con l'aumentare del numero di lettere diverse nella medesima posizione. Il costo della funzione è $\Theta(nlog(n))$, in quanto viene visitato tutto l'albero a partire dal minimo e muovendosi sui successori. Il costo di *HammingDistance()* dipende dalle lunghezze delle parole, ma avendo un limite superiore di 20 possiamo considerarlo costante.

La funzione *createFromFile()* e *importDictionary()* esegue un numero di inserimenti pari alla lunghezza del file, considerando che l'inserimento ha costo logaritmico e la lunghezza massima di un blocco da inserire è formata da 70 caratteri (parola + definizione) potremmo ignorare la lettura e dire che il costo è $O(mlog(n))$ con *n* pari al numero di nodi presenti nell'albero e *m* pari al numero di nodi nuovi da inserire.

Le funzioni *printDictionary()* e *countWord()* hanno costo $O(n)$, perché scorrono tutto l'albero.

La funzione *getWordAt()* ha costo $O(nlog(n))$, in quanto partendo dal nodo minimo cerco il suo successore e quindi nel caso pessimo (ultimo elemento) eseguo *n* movimenti di costo $O(log(n))$ (successore). Simile approccio è stato usato per *saveDictionary()*.

compressHuffman() riceve l'albero e poi lo scrive su file per gestire meglio la codifica, quindi richiama *saveDictionary()* che ha costo ($O(nlog(n))$). La compressione inoltre legge tutto il file (costo lineare sul numero di caratteri) e poi costruisce tabella e albero di Huffman. L'albero viene costruito prima ordinando le frequenze ($O(nlog(n))$, Quicksort) e poi unendo i vari sottoalberi (tempo costante, alcuni passaggi). La tabella occupa uno spazio esiguo, considerando che i caratteri ammissibili nel file sono pochi (lettere minuscole e pochi simboli), accelera invece i tempi di inserimento e codifica, evitando di usare l'albero per scrivere ogni carattere. I passaggi più impegnativi hanno costo $O(nlog(n))$. *decompressHuffman()* ricostruisce l'albero (nel file sono già presenti le frequenze) necessario per codificare i bit del file compresso (costo logaritmico, altezza albero di Huffman per raggiungere le foglie). Ottenuto il file decodificato lo importa nell'albero usando la funzione *importDictionary()*. I costi delle funzioni sono abbastanza simili, essendo da un certo punto di vista, una la simmetrica dell'altra e richiedendo le stesse costose, ma fondamentali risorse: l'albero di Huffman e un file di appoggio.

Il nodo generico dell'albero è formato da due stringhe contenenti la parola e la relativa definizione, i puntatori al nodo padre, figlio sinistro e destro e infine un flag che indica il colore del nodo (1 = rosso). L'inserimento nell'albero avviene secondo un ordine lessicografico (il figlio destro è "più grande" del padre).

```
typedef struct NODO
{
    char* word;
    char* def;
    int red;
    struct NODO* parent;
    struct NODO* rchild;
    struct NODO* lchild;
}NODO;
```