**Relazione sul primo progetto per l’esame di Ingegneria degli Algoritmi**

Fabio Buracchi, 0253822

Danilo D’Amico, 0252956

**PREFAZIONE**

Seguendo le indicazioni riportate nella traccia, è stato svolto il Progetto 1.

Il codice è disponibile su GitHub al seguente link <https://github.com/Zaimokuza/EX1IA18>.

Nella realizzazione del progetto è stato utilizzato esclusivamente codice scritto dagli autori o i moduli forniti sul sito GitHub del corso, con l’eccezione dei package e richiesti dal modulo per la generazione dei grafici presenti in questa relazione.

Per riferirsi agli elementi richiesti dalla traccia durante questa relazione si utilizzerà la stessa nomenclatura presente nella traccia del progetto.

Il file costituisce un insieme variegato esempi di casi di utilizzo del codice scritto.

**SCELTE IMPLEMENTATIVE**

Il progetto si articola in due parti: i moduli e presenti nella della repository ed i moduli presenti nella directory , dai quali viene importato tutto il necessario per l’inizializzazione di liste collegate capaci di gestire dizionari ed alberi AVL.

Il progetto è stato finalizzato alla creazione della classe che esegue in dettaglio le operazioni richieste dalla traccia. Le istanze di ricevono come parametri e ed inizializzano un vettore contenente oggetti di tipo .

Tra le classi contenute nella cartella *Librerie*, sono state apportate modifiche solo in , alla quale è stato aggiunto un metodo *size* che restituisce la lunghezza della lista collegata. Tale aggiunta semplifica ed aumenta la leggibilità del codice che costituisce *,* in quanto consente di astrarre l’implementazione dei metodi di e dalla struttura dati sulla quale stanno operando.

La Tabella sottostante riassume i costi dei metodi presenti in *LinkedListDictionaryHybrid.*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Albero AVL** | **Linked List Dictionary** | **Linked List Dictionary Hybrid** |
| **Insert** |  |  | O(nlogn) |
| **Delete** |  |  |  |
| **Search** |  |  |  |
| ***Size*** |  |  | / |

La trasformazione tra albero AVL e Linked List avviene quando la richiesta di un’operazione di tipo o porta il numero di elementi contenuti nella struttura a *r*.

Il costo di *switchToAVL*, che costruisce un albero AVL a partire da una lista collegata, deriva dal dover eseguire l’operazione di inserimento in un albero AVL, di costo *O(logn)*, una volta per ogni elemento presente nella lista che si va a trasformare:

Per quanto riguarda *switchToLinkedList*, in modo analogo, si scorre l’albero AVL mediante *fillLinkedList* (di costo lineare, perché visita una singola volta ogni nodo presente nell’albero AVL) ed inserendolo, con costo *O(1)* per ogni visita, all’interno della nuova lista collegata.

Per quanto riguarda i metodi di *size*, nell’albero AVL questa operazione ha costo *O(1)* perché viene definita in *BinarySearchTree* come il valore di *nodesNumber*, un attributo della classe genitrice *BinaryTree*.

In *LinkedListDictionary*, l’implementazione di *size* effettuata prevede una singola visita ad ogni nodo della lista collegata con il conseguente aumento di un parametro *length*, rendendo di conseguenza il costo del codice lineare.

I costi dei metodi di *insert*, *delete* e *search* nelle due strutture derivano invece dalla teoria del corso.

Si noti tuttavia che il caso peggiore della funzione *insert* si ha quando il numero di elementi presenti all’interno della lista collegata diventa pari al parametro *switchingLength* (6 nella traccia del progetto) e bisogna chiamare la funzione *switchToAVL*:

Tale peggioramento asintotico non avviene invece quando nella funzione di *delete* viene chiamata *switchToLinkedList*, che mantiene lo stesso costo asintotico dell’inserimento peggiore, ovvero *O(n)*. Si noti tuttavia che cambierà la costante moltiplicativa di *O(n)*, dato che è necessario operare prima la cancellazione e poi la trasformazione.

Si è scelto di mantenere disordinati gli elementi all’interno della lista collegata in quanto un loro inserimento ordinato avrebbe aumentato il tempo di *insert* da *O(1)* a *O(logn)* senza un miglioramento dei metodi di *delete* e *search*, data l’impossibilità di eseguire una ricerca binaria.

Per gestire l’array contenente le d istanze di *LinkedListDictionaryHybrid*, è stata scritta una classe *CustomDictionary*. Tale classe, contenuta nel file omonimo, indirizza le coppie di valori (*key*, *value*) in tempo *O(1)*, secondo l’equazione:

nella quale i rappresenta l’i-esimo elemento dell’array *v* nel quale inserire o cercare il valore di *key*.

Tale indirizzamento è compiuto dal metodo *getIndexFromKey*, mentre i metodi di *insert, search* e *delete* vengono ereditati dalla classe *LinkedListDictionaryHybrid*, leggermente modificati in modo da aggiungere l’indirizzamento sopra descritto.

Per analizzare il costo delle singole operazioni all’interno di , è utile trovare la cardinalità massima di ogni elemento di :

Data la conformazione dell’array, possiamo individuare un caso migliore ed uno peggiore di scelta dei parametri della classe .

Identifichiamo come caso peggiore una scelta di massimo e minimo tale che tutti gli input siano maggiori del massimo (o minori del minimo). In tal caso elementi dell’array non vengono utilizzati, portando l’analisi del costo asintotico dei metodi del a ricalcare quelli di .

Il caso auspicabile, invece, è quello in cui . In tale situazione, infatti, sappiamo, per averlo dimostrato in precedenza, che il massimo numero di elementi dell’-esima è . Il costo asintotico dei metodi sarà allora:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

È possibile ottimizzare l’andamento asintotico della classe mediante una scelta opportuna del parametro b.

La creazione di un’istanza richiede l’inizializzazione di liste, in modo da non gravare con continui controlli sulle operazioni di e .

La scelta del parametro corrisponde a determinare il massimo numero di elementi che le prime liste del vettore possono contenere, di conseguenza ha un grande peso sull’efficienza del codice.

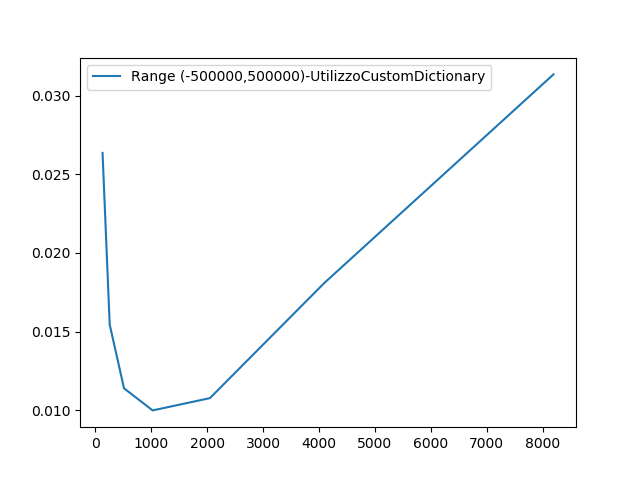
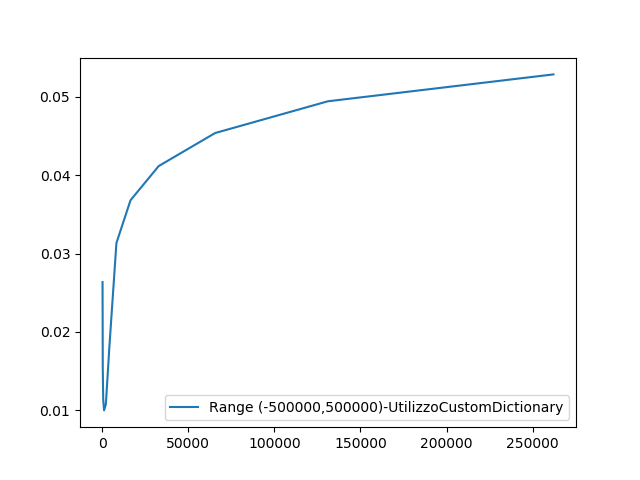
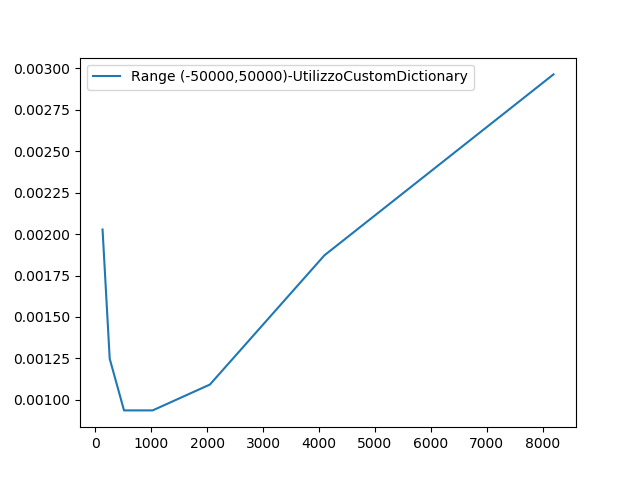
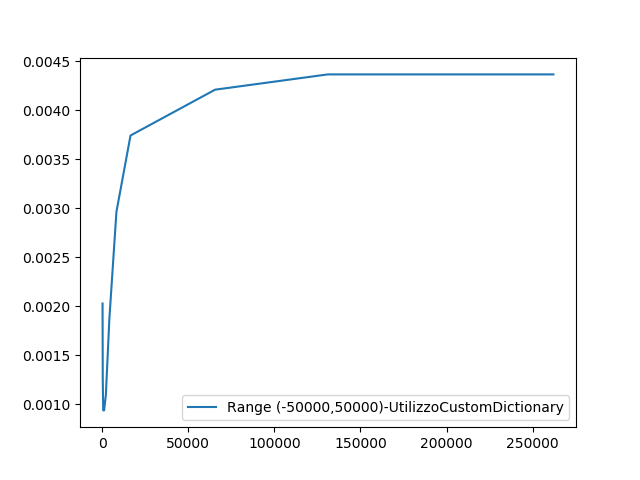
Scegliere un valore per troppo piccolo, seppur aumentando l’efficienza della struttura dati, aumenterebbe tuttavia il tempo di caricamento del .

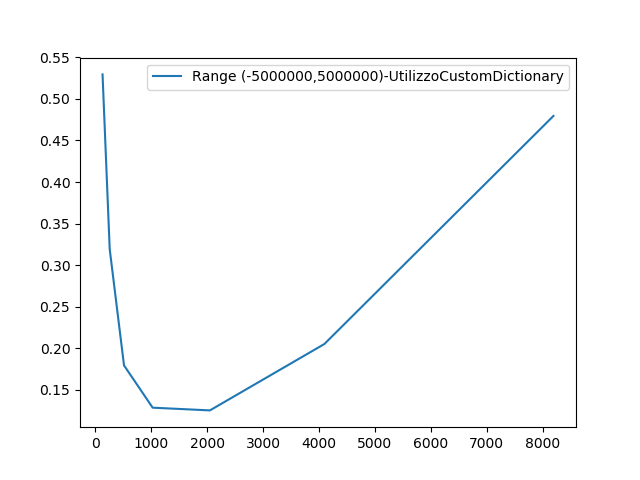
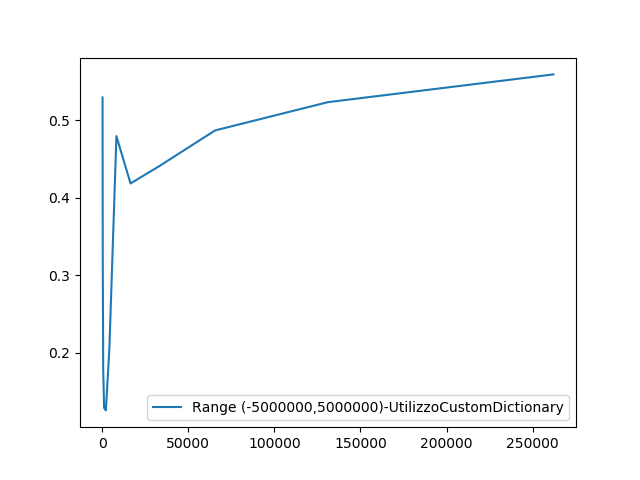
Per trovare un compromesso ottimale sono stati utilizzati test sperimentali.

Da questi è emerso che per la scelta di un qualsiasi non compromette in maniera rilevabile l’efficienza del dizionario.

I seguenti grafici rappresentano i risultati dei test eseguiti di operazioni effettuate sul dizionario con differenti.

I grafici sono alternati tra grafici comprendenti tutti gli esperimenti fatti e quelli contenenti solo in dettaglio il rilevante. Sulle ordinate viene rappresentato il tempo in secondi mentre sulle ordinate i valori di .





Da un’analisi sperimentale emerge che impostare contribuisca a rendere più efficiente la struttura dati.

**RISULTATI SPERIMENTALI**

Gli esperimenti sono stati condotti utilizzando le ultime versioni disponibili di *PyCharm* (versione 2018.3.1) e dell’interprete di *Python* (versione 3.7.1)

|  |  |
| --- | --- |
| Inserimento  Insert | Ricerca  Search |
| Eliminazione  Deleate | Caso medio d’utilizzo  Utilizzo |

Dai test finali si può notare l’importanza della scelta corretta dei parametri è fondamentale per avere un’efficienza adeguata del dizionario.

Sia la lista di tutte le chiavi che s’intendono utilizzare durante l’esecuzione del dizionario.

Data l’evidenza sperimentale, i parametri che si sono meglio prestati all’esecuzione del progetto sono stati:

A livello asintotico i grafici assumono un andamento lineare, con un c