Introducción

En este proyecto, se creó un corrector ortográfico que funciona de la siguiente manera: a partir de dos archivos de texto (una entrada y un diccionario) creamos un tercer archivo donde se guardará la misma entrada pero corregida según el diccionario (salida). Para llegar a la solución de dicho problema, se utiliza la técnica de diseño de *programación dinámica*.

Objetivo

- Implementación de algoritmos conocidos dándoles un uso real y específico.
- Seleccionar e implementar (sin recurrir a librerías) correctamente las estructuras de datos utilizadas según las operaciones en las cuales operarán teniendo en cuenta la complejidad computacional, elevando la eficiencia algorítmica.

Algoritmos implementados

Wagner-Fischer

Este algoritmo lo utilizamos para calcular la distancia de Levenshtein entre dos palabras, pasando por parámetro las dos cadenas de caracteres. Se genera una matriz de (N+1)x(M+1) donde N y M son la longitud de cada palabra, y se rellena la matriz con la distancia entre cada subcadena de ambas palabras. Luego en la última posición de la matriz se obtendrá la distancia entre esas dos palabras. La complejidad es dependiente de la longitud de las palabras, por lo tanto pertenece al orden O(n*m). Se aplicará este algoritmo cuando no exista la palabra de entrada en el diccionario, pero sí exista en el diccionario un conjunto de palabras que comiencen con la misma letra. Con él se determina cual, de ese conjunto, es la palabra “más cercana” a la palabra de entrada, para saber cuál guardar en la salida.

MergeSort

El mergesort lo utilizamos para el paso posterior a cargar el arreglo de palabras del diccionario. Una vez cargado el arreglo aplicamos el mergesort para ordenarlo, así luego podemos realizar búsquedas binarias las cuales poseen una complejidad de O(log n). Sin embargo, el MergeSort tiene una complejidad que pertenece a O(n log n) la cual se debe al llamado a la función intercalar (O(n)) y los dos llamados recursivos del propio mergesort en los cuales se va dividiendo en dos la longitud del arreglo.

Búsqueda Binaria (iterativa y recursiva)

La búsqueda binaria se realiza sobre el arreglo de strings del diccionario, ordenado anteriormente y se compara la cadena de texto recibida por parámetro con las cadenas del arreglo hasta llegar o no a una coincidencia. Este algoritmo primero se ubica en la mitad del arreglo y compara el string ubicada en dicha posición central con la cadena que recibe por parámetro. Si son iguales se produce un retorno finalizando la ejecución del algoritmo, y en caso contrario si la cadena ubicada en medio del arreglo fuese menor a la cadena pasada por parámetro, se vuelve a ejecutar el algoritmo pero tomando como arreglo en el cual buscar la mitad mayor (índices superiores). Y en caso contrario, es decir que la cadena situada en medio del arreglo es mayor que la pasada por parámetro, el algoritmo se ejecutará nuevamente pero tomando como base la mitad menor del arreglo. Se obtiene una complejidad temporal de O(log n) siendo n la longitud total del arreglo del cual se comienza a buscar.

Implementación

Se desarrollaron las siguientes clases:

Clase Lista

+ Estructuras de datos utilizadas:

Para la implementación de este clase, optamos por una estructura dinámica. Ya que con ellas podemos transportar todos los datos de la entrada desde memoria secundaria a memoria primaria en un solo recorrido. Entre ellas, elegimos a la lista simple vinculada por su fácil utilización y porque nos resultó eficaz debido a que utilizaremos a la lista para implementar los archivos de entrada y salida, donde solo necesitábamos hacer recorridos secuenciales y ninguna búsqueda.

+ Métodos de la clase y sus complejidades:

Lista(): Es el constructor de de la clase, simplemente inicializa a los miembros en cero. Pertenece al orden O(1).

void agregarLinea(string linea): carga un nuevo nodo con el string que recibe por parámetro, y agrega dicho nodo al final de la lista. Su complejidad pertenece al orden O(1).

void setearActual(): Forma parte del mecanismo implementado para iterar a través de la lista. Cuando es llamado, modifica al dato miembro “actual” para que apunte al primer elemento de la lista. Pertenece al orden O(1).
bool esVacio() const: Método observador que verifica si la lista está vacía. Pertenece al orden O(1).

bool devolverActual(string & linea): En conjunto con el método “setearActual”, forman el mecanismo de iteración implementado. Este método devuelve false, si se ha llegado al final de la lista, o si la lista era vacía. Devuelve true en caso contrario y si esto ocurre le asigna al string linea el valor apuntado por el puntero miembro “actual”, y hace que “actual” avance una posición en la lista. Pertenece al orden O(1).

~Lista(): Es el destructor de la clase, tiene sentido haberlo definido ya que hemos pedido memoria dinámicamente. Pertenece al orden O(n) donde n es la longitud de la lista.

Clase Archivo

+ Estructuras de datos utilizadas:

En la implementación, definimos la clase Archivo, la cual posee como atributo una clase Lista. Creamos una instancia de esta clase para la entrada, y otra para la salida. Se escogió así, ya que como mencionamos anteriormente, con los datos de entrada y salida solo necesitamos hacer recorridos secuenciales, y ninguna búsqueda. Por otro parte, no se conoce con anticipación el tamaño de la entrada ni de la salida, lo cual no es un problema con esta implementación. Además, nos interesa conservar el orden en el que vienen dados los datos de entrada, para poder guardarlos en una salida corregida en el mismo orden. Para dicho objetivo, nos beneficia elegir a la lista, ya que la misma, agrega los elementos siempre al final.

+ Métodos de la implementación y sus complejidades:

Archivo(): Es el constructor de la clase. No tiene nada implementado ya que el único atributo, es del tipo lista, cuyo constructor tampoco tiene parámetros. Por lo tanto, pertenece al orden O(1).

void getPalabras(string ruta): Carga la lista de la clase, con todas las palabras que se encuentren en el archivo de memoria secundaria que está en el directorio dado por “ruta”. Pertenece al orden O(n) donde n representa la cantidad de palabras en el archivo físico.

void agregarLinea(string linea): carga el string que recibe por parámetro a la lista de la clase. Pertenece al orden O(1).

void guardar(string ruta): crea (de ser necesario) un archivo en memoria secundario ubicado en el directorio dado por “ruta”. En el cual almacena todas las palabras que haya en la lista de la clase. Su complejidad pertenece al orden O(n) donde n representa la cantidad de palabras de la lista.

void setearActual(): Simplemente llama al método “setearActual” de la lista miembro de la clase archivo. Pertenece al orden O(1).

bool esVacio() const: Método observador que devuelve si es o no vacío el archivo. Pertenece al orden O(1).

bool devolverActual(string & linea): Llama al método “devolverActual” de la lista que pertenece a la clase archivo. Pertenece al orden O(1).

~Archivo(): Es el destructor de la clase archivo. No tiene implementación ya que aquí no se pidió memoria dinámicamente. Por lo tanto, pertenece al orden O(1).

Clase Diccionario

+ Estructuras de datos utilizadas:

Realizamos la implementación de la clase Diccionario con un arreglo dinámico de strings, el cual una vez cargado, es ordenado ascendentemente. Elegimos al arreglo (obligatoriamente dinámico ya que se desconoce el tamaño del archivo físico a partir del cual debe crearse) ya que por sus características, nos permite hacer búsquedas de elementos con baja complejidad. Opinamos que es mayor la eficiencia lograda de esta manera, que dejando desordenado el arreglo y haciendo búsquedas secuenciales, ya que deben hacerse demasiadas búsquedas.

+ Métodos de la implementación y sus complejidades:

Diccionario(): Constructor de la clase. Inicializa los atributos en cero. Pertenece al orden O(1).

unsigned int filesize(ifstream & dicc): Método privado. Devuelve la cantidad total de líneas que posee el archivo en memoria secundaria “dicc”. El dato devuelto es utilizado para establecer el tamaño del arreglo. Pertenece al orden O(n) donde n representa la cantidad de lineas de “dicc”.

void intercalar(int inicio, int medio, int fin): Método privado. Intercala ordenadamente los elementos de ambas mitades del arreglo. Su complejidad pertenece al orden O(n) donde n es el tamaño del arreglo
(n = fin - inicio + 1).

void ordenar(unsigned int inicio,unsigned int fin): método privado usado por getPalabras. Es la implementación del algoritmo de ordenamiento ***MergeSort***. Su complejidad pertenece al orden O(n . log n) donde n representa la longitud del arreglo.

bool busquedaBinaria(int inicio, int fin, string dato) const: es una función privada recursiva que devuelve si encuentra o no el string dato en el arreglo. Su complejidad pertenece al orden O(log n) donde n es la dimensión del arreglo.

bool buscarIndices(int & inicio,int & fin,char letra) const: Busca los índices en el arreglo entre los cuales se encuentran las palabras que comienzan con determinada letra. Guarda esa información en los int que llegan por parámetros. Devuelve true o false dependiendo de si existe o no , al menos una palabra con dicha letra. Pertenece al orden O(log n) ya que realiza una búsqueda binaria iterativa, siendo n el tamaño del arreglo.

void getPalabras(string ruta): Carga el arreglo completamente, con todas las palabras que se encuentran almacenadas en el archivo en memoria secundaria ubicado en el directorio dado por “ruta”. Luego de cargarlo, lo ordena. Su complejidad pertenece al orden O(n. log n) donde n es el tamaño del arreglo, y resulta de dicha complejidad por llamar al algoritmo Merge Sort.

bool existe(string linea) const: método observador que devuelve si existe o no una palabra en el arreglo. Su complejidad pertenece al orden O(log n) donde n es el tamaño del arreglo, y este orden viene dado por llamar a la función “busquedaBinaria”.

void comienzaCon(char letra, Lista & resultado) const: carga en la lista que le llega por parámetro todas las palabras del arreglo, que comienzan con una letra determinada. Su complejidad pertenece al orden O(log n) siendo n el tamaño del arreglo. Esta complejidad viene dada por llamar a la funcion “buscarIndices”.

bool esVacio() const: método observador que verifica si el diccionario es vacío o no. Pertenece al orden O(1).

~Diccionario(): Destructor de la clase. En el se elimina el arreglo. Su complejidad pertenece al orden O(1).