TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica

Bachillerato en Ingeniería en Computación

Estructura de Datos IC-2001

Proyecto Programado II
Análisis de texto

Profesor: Mauricio Avilés Cisneros

Alumnos: Eric Alpízar Prendas

David González Agüero

Fecha de entrega: 05/8/2020

Tabla de contenido

Resumen

Hoy en día, el análisis de texto es una de las bases de la comunicación moderna, ya que están presentes en básicamente todos los aparatos de comunicación existentes tales como los celulares, los asistentes personales inteligentes, entre otras cosas. Para la creación de estos algoritmos de reconocimiento inteligente y análisis de texto, las estructuras de datos juegan un papel muy importante. En la documentación de este proyecto vamos a estar recorriendo a fondo el proceso de la creación de un programa de análisis de texto de terminal creado en C++. Así mismo, vamos a compartir todas las experiencias que tuvimos al implementar las diferentes estructuras, nuestro enfoque para lograr la mayor eficiencia posible, el análisis de algoritmos que realizamos, recomendaciones, y muchos otros elementos más. Al final de esta documentación se incluyen las conclusiones de aprendizaje a las que llegamos al concluir la elaboración de éste proyecto.

Introducción

Este trabajo se realiza para la segunda evaluación de tipo proyecto del curso de Estructuras de Datos en el Instituto Tecnológico de Costa Rica. Este mismo consiste en la creación de un programa de análisis de texto estableciendo el uso de estructuras de datos vistas en clase, entre ellas principalmente los árboles Trie, que se utilizan específicamente para el uso de *strings* o cadenas de caracteres, pero también se utilizan otras estructuras tales como árboles Splay, AVL binarios, listas de arreglos, listas doblemente enlazadas, entre otras. La entrega también incluye el archivo .exe para la rápida y conveniente ejecución del compilado del programa, también a diferencia de la entrega anterior este programa realiza una lectura de datos a la computadora por medio del API del sistema operativo, así que para el funcionamiento de este mismo se requiere el archivo ignore.txt, este contiene las palabras que van a ser ignoradas al hacer el mostrar de palabras más frecuentemente usadas. Este archivo está incluido, tanto en el proyecto, como en el ejecutable, por defecto.

Específicamente con respecto al código del proyecto, este mismo debe de hacer la lectura del archivo de texto que se desea analizar, a partir de tal, se lee cada línea y se analiza una por una. Al hacer dicha lectura, se analizan las palabras a partir de la separación de espacios ignorando todos los caracteres alfanuméricos, insertando así todas las palabras únicas en el Trie. Además de la validación de la inserción de palabras, la entrega requiere que hagamos validaciones en los menú del programa. Es necesario también guardar todas las líneas de archivo de texto en el disco a memoria de acceso rápido mediante la utilización de una estructura para poderlo accesar de una manera rápida y eficiente al consultar alguna de estas líneas. La frecuencia en la que cada palabra aparece en el texto también es almacenada en el Trie, mediante este elemento se debe calcular el *top* de palabras más utilizadas y las menos utilizadas en el texto.

El programa debe de tener la funcionalidad de consultar una palabra por prefijo, es decir si se consulta el prefijo “la”, el programa debe mostrar palabras como “labio”, “ladrar”, “laboratorio”, estas palabras obviamente se extraen en base a las que estén en el texto que se está analizando. La entrega también requiere de un sistema de indización de palabras o búsqueda por la línea del texto, es decir que, si una palabra es consultada, el programa muestra las líneas en el texto original donde aparece esta palabra. También se debe incluir la implementación de la búsqueda de palabras basadas en la cantidad de letras que estas tengan; por ejemplo, es posible consultar todas las palabras en todo el texto que tengan únicamente dos o cuatro caracteres.

Volviendo al tema de las palabras a ignorar, el programa debe leer del archivo “*ignore.txt”* y cargar todas las palabras contenidas en el mismo en la memoria de acceso rápido, a esta lista se le pueden agregar palabras en dicho programa y se debe actualizar el archivo *ignore.txt* al mismo tiempo. Es necesario que se pueda borrar toda la lista de archivos *ignore.txt* y actualizar el archivo en el disco y la estructura de acceso rápido de contiene dichas palabras. Al mostrar las palabras

menos utilizadas y las más utilizadas, el programa no debe de mostrar las que están en el archivo *ignore.txt*; de esta forma, estas palabras van a ser ignoradas.

Todo lo mencionado anteriormente debe implementarse de la manera más eficiente posible sin sacrificar la calidad del programa o sus funcionalidades. Legando así, de una manera objetiva y resumida, la presente entrega.

Presentación de Análisis y Análisis del problema

I. ¿Qué hay que resolver?

A la hora de idear y analizar el código que necesitábamos hacer, formulamos una lista de lo que debíamos cumplir para que el programa tuviera el funcionamiento y manejo que estábamos buscando, en esta lista se muestran los principales problemas y retos que tuvimos que analizar al empezar nuestro programa de análisis de texto "NeuraText":

- Idear la mejor manera para realizar la validación de la lectura de palabras de las líneas originales en el texto, esto debía ser eficiente y exacto.
- Encontrar la mejor manera para leer archivos de una manera rápida donde podamos extraer información del número de línea donde estemos leyendo.
- Encontrar la manera de leer archivos en *Unicode* ya que algunos según el tipo de *encoding* pesan más de un *byte*.
- Buscar y hacer un análisis de la mejor estructura de datos para almacenar todas las líneas del archivo de texto original realizando una inserción y búsqueda rápida para que el programa tenga un buen rendimiento y eficiencia.
- Modificar algunas estructuras de datos hechas en clases para adaptarlas a las funcionalidades del proyecto.
- Formular un algoritmo que nos permita acceso a las palabras por su profundidad en el trie, evidentemente haciendo uso de la recursión.
- Hallar la interfaz de texto más simple, eficaz y agradable para el usuario.
- Utilizar una consola de salida que nos funcione y que no se caiga al imprimir cantidades masivas de texto, que al mismo tiempo muestre todos los caracteres de una manera correcta.
- Idear una manera para el manejo de caracteres especiales del idioma español, como las tildes, diéresis, entre otros. Debido a que tomaban diferentes valores en el texto en memoria al texto que lee la consola digitado por el usuario.
- Buscar una manera eficiente para guardar la frecuencia en la que aparece cada palabra.

- Encontrar la mejor estructura de datos vista en clase para guardar las palabras a ignorar ya que también se deben de almacenar en memoria de acceso rápido
- Idear una manera bastante eficiente de imprimir las líneas originales de texto de una palabra consultada por el usuario, sin que la búsqueda tome mucho tiempo y sin agotar la paciencia de éste.
- Crear una implementación para imprimir la frecuencia en la que una palabra es utilizada en las funciones que lo requieren, de la manera más practica y eficiente posible.
- Implementar eficientemente el algoritmo que calcula las palabras más y menos frecuentes

II. ¿Cómo se va a resolver el problema?

Teniendo una idea de los problemas a solucionar, se comenzó a planear la forma de resolverlos uno a uno. A continuación, se recalcarán cada uno de los problemas identificados junto con su propuesta para poder resolverlo:

1. *Problema: Idear la mejor manera para realizar la validación de la lectura de palabras de las líneas originales en el texto, esto debía ser eficiente y exacto.*

   Propuesta: Para hacer la validación de lectura lo que hicimos fue investigar en C++ la manera más eficiente y exacta para separar las palabras en una oración. Para esto encontramos la función isalpha() la cual se encarga de retornar un valor booleano si el carácter que estamos leyendo es alfabético, con la ayuda de esta función en conjunto de isspace() que hace lo mismo para caracteres en blanco logramos obtener un resultado bastante eficiente y preciso. Estas dos funciones están contenidas en la biblioteca standard de C++.

2. *Problema: Encontrar la mejor manera para leer archivos de una manera rápida donde podamos extraer información del número de línea donde estamos leyendo.*

   Propuesta: Para realizar esto decidimos tomar la misma variable de la iteración que hace la lectura secuencial del archivo línea por línea y tomarla como numero de línea, esto nos generó una implementación bastante eficiente.

3. *Problema: Encontrar la manera de leer archivos en Unicode ya que algunos según el tipo de encoding pesan más de un byte.*

   Propuesta: Investigar en internet las maneras de poder abrir en C++ archivos de encodings específicos para así almacenarlos correctamente dentro del Trie y dentro de la Estructura de Datos utilizada para guardar el contenido de cada línea del texto.

4. *Problema: Buscar y hacer un análisis de la mejor estructura de datos para almacenar todas las líneas del archivo de texto original realizando una inserción y búsqueda rápida para que el programa tenga un buen rendimiento y eficiencia.*

   Propuesta: Para esto decidimos inicialmente hacerlo mediante un diccionario implementado con un árbol binario normal debido a las ventajas que nos trae un árbol binario al tener una inserción relativamente rápida y una búsqueda más rápida que la secuencial, sin embargo al meditarlo más profundamente nos dimos cuenta que si utilizábamos este tipo de árbol tendríamos un árbol resultante donde prácticamente seria secuencial debido a que el orden de la inserción era ascendente; es decir, como estábamos insertando el número de líneas como llave del diccionario de una manera estrictamente ascendente 0, 1 ,2,…, n entonces el árbol resultante tomaría una forma secuencial y esto al hacer búsquedas nos tomaría demasiado tiempo, para solucionar este problema decidimos hacer el uso de un diccionario tipo AVLTree donde guardábamos el numero línea que estábamos leyendo como llave y todos los caracteres de esa línea como valor, igual a como lo hubiéramos hecho con un árbol binario pero esta vez haciendo rotaciones para que el árbol resultante fuera un árbol balanceado donde sacrificaríamos un poco en el tiempo de inserción pero tendríamos una búsqueda más rápida y eficiente. Esto nos funcionó bien para archivos de un tamaño medianos o pequeños, sin embargo cuando decidimos probar con archivos más grandes el tiempo de inserción era mucho, en nuestras computadoras la inserción tardaba de 23 – 28 segundos en libros como don Quijote, esto es porque al hacer la inserción con llaves estrictamente ascendentes las rotaciones del AVL requerían muchas operaciones aritméticas que cargaban al CPU, donde en algunas pruebas durante el proceso de inserción el programa utilizaba aproximadamente un rango de 50% - 60% del CPU en la computadora, esto nos obligó a pensar en alguna otra estructura que se desempeñara mejor para realizar este trabajo, al meditar un poco más la naturaleza de la inserción de nuestros archivos nos dimos cuenta que el orden ascendente al hacer la inserción más bien podría ser de gran ayuda ya que nos permitiría usar una estructura con índices donde nos permitiría hacer una inserción muy rápida y una búsqueda prácticamente instantánea. Este pensamiento fue el que nos llevó a usar un arrayList en memoria dinámica y esto hizo que nuestro programa se comportara como una belleza.

   Al implementar esta estructura el único problema que tuvimos fue definir el tamaño del arreglo ya que no queríamos malgastar nada de memoria, pero tampoco definir un tamaño que no soportara los libros más grandes, esto lo solucionamos con una función que calcula el tamaño del archivo .txt a leer y le asigna este tamaño al arreglo de arrayList. Esta implementación nos generó los tiempos que estábamos buscando, con una búsqueda prácticamente instantánea y los tiempos de inserción pasaron de 28 segundos a 1 – 2 segundos para libros como don Quijote.

   Para llevar esta prueba un poco más al extremo, ilustrando un poco mejor la diferencia, corrimos el programa en una computadora con un procesador más básico como el Intel

Celeron de 1.5GHz y obtuvimos resultados bastante interesantes, con la versión de diccionario AVL analizando el libro WarPeace la inserción tardo 201112 microsegundos, ósea dos minutos enteros, con la versión final utilizando arrayList la misma inserción tardo 4383 microsegundos, es decir 4 segundos.

5. *Modificar algunas estructuras de datos hechas en clases para adaptarlas a las funcionalidades del proyecto.*

   Propuesta: Crear un diccionario que sea tipo AVL, en la clase TrieNode agregar parámetros que controlen la frecuencia de uso de las palabras, también otro parámetro que guarde dentro de una lista las líneas en donde aparece alguna palabra. Agregar métodos nuevos que hagan uso de estos parámetros. Dentro de la clase Trie, crear métodos que puedan interactuar con los nuevos métodos de la clase TrieNode, dando nuevas funcionalidades a la clase Trie que cumplen con las especificaciones del proyecto.

6. *Problema: Formular un algoritmo que nos permita acceso a las palabras por su profundidad en el trie, evidentemente haciendo uso de la recursión.*

   Propuesta: Para realizar este algoritmo hicimos un análisis en papel de lo que queríamos hacer y terminamos con un método recursivo bastante eficiente y preciso. Nos apoyamos mucho en la función que consulta por prefijo con de la profundidad.

7. *Problema: Hallar la interfaz de texto más simple, eficaz y agradable para el usuario.*

   Propuesta: Para hacer esto intentamos realizar la una interfaz de texto fácil e intuitiva con todas las validaciones de entrada necesarias para que el usuario no tuviera problemas al interactuar con el programa. También hicimos la adición de ASCII art para darle un poco de estética.

8. *Problema: Utilizar una consola de salida que nos funcione y que no se caiga al imprimir cantidades masivas de texto, que al mismo tiempo muestre todos los caracteres de una manera correcta.*

   Propuesta: Con la experiencia de aprendizaje que se tuvo en el primer proyecto programado, se pensó en tomar parte del código encargado en mostrar los símbolos de las cartas, utilizando esta parte de código en el proyecto actual se solucionaría el problema de mostrar en la consola los caracteres especiales de la lengua española.

9. *Problema: Idear una manera para el manejo de palabras especiales al idioma español como las tildes, diéresis, entre otros. Debido a que tomaban diferentes valores en el texto en memoria a el texto que lee la consola digitado por el usuario.*

   Propuesta: Conseguir los valores de la tabla ASCII de cada carácter especial y crear una función que los convierta al código correcto de los caracteres especiales guardados dentro del árbol Trie.

10. *Problema: Buscar una manera eficiente para guardar la frecuencia en la que aparece cada palabra.*

    Propuesta: Lo que realizamos para solucionar esto fue crear un atributo *frequency* en el TrieNode para que cada vez que en la inserción se intente insertar una palabra que ya esta en el Trie, entonces simplemente incrementar la frecuencia de esa palabra en el nodo

final. De esta manera no hay que hacer ningún paso adicional en la inserción, ni tampoco hacer una búsqueda o análisis adicional para determinar este elemento.

11. *Problema: Encontrar la mejor estructura de datos vista en clase para guardar las palabras a ignorar ya que también se deben de almacenar en memoria de acceso rápido*

    Propuesta: Para esto dedicamos tiempo a hacer un análisis de lo que ocupábamos y la naturaleza de la estructura que requeríamos. Después de hacer esto decidimos que el árbol Splay sería una buena opción para lograr hacer una implementación eficiente. Esta decisión es porque notamos que algunas palabras a ignorar en el archivo .txt cargadas en el árbol eran mucho más propensas a ser buscadas que otras, entonces el árbol Splay ayudaría a optimizar un poco el tiempo de búsqueda al hacer la consulta si una palabra esta contenida en el árbol. Esto funcionó bastante bien y nos generó los tiempos de búsqueda que buscábamos, la inserción debido a que este archivo no era tan grande como los libros a analizar, también se efectúa de una manera rápida.

12. *Problema: Idear una manera bastante eficiente de imprimir las líneas originales de texto de una palabra consultada por el usuario, sin que la búsqueda tome mucho tiempo, de esta manera no agotar la paciencia de este.*

    Propuesta: Para imprimir las líneas en donde la palabra aparecía, la propuesta fue crear una estructura aparte que almacena las líneas y todo el texto que contiene; luego, al tener guardado los datos, se crearía un método encargado de buscar en esa estructura la línea mandada como parámetro y este retornaría todas las oraciones relacionadas a la línea para luego imprimirla en pantalla.

13. *Problema: Crear una implementación para imprimir la frecuencia en la que una palabra es utilizada en las funciones que lo requieren, de la manera más practica y eficiente posible.*

    Propuesta: En resolución a dicha propuesta, creamos algunos métodos que devuelven la frecuencia del nodo final de una palabra, entonces se busca la palabra en el Trie y se devuelve la frecuencia para que pueda ser utilizada en los otros métodos que la ocupemo

14. ***Problema:*** *Implementar eficientemente el algoritmo que calcula las palabras más frecuentes y menos frecuentes*

    Propuesta: Para solucionar este algoritmo, pensamos en numerosas formas de hacerlo; sin embargo, algunas no prometían ser muy eficientes. Al final, utilizamos una estructura que nos ayudara a hacer esto de manera tal que nos ordenara las palabras más utilizadas a las menos utilizadas, descendentemente, basado en el criterio de la frecuencia de cada palabra. De esta forma, obtenemos las palabras más utilizadas en el texto y las palabras menos utilizadas también. Para lograr esto, teníamos muy claro que el *heap* era la estructura indicada; por lo tanto, hicimos una implementación donde utilizamos un *heapNode,* que tiene una llave y un valor, el cual resulta similar a la composición del *KVPair*.

    Una vez obtenido lo anterior, insertamos todas las palabras del Trie con su frecuencia como llave para el *heap,* y, la misma estructura se encargaba de poner las palabras más utilizadas en los niveles más altos del árbol y las palabras menos utilizadas en sus niveles más bajos, realizando un tipo de *heapSort*.

    Posteriormente, sacamos todas las palabras de la estructura y las introducimos en una lista, la cual devolvemos y asignamos como atributo de la clase principal como la lista de las palabras más utilizadas, la cual trae las menos utilizadas al mismo tiempo al final de la lista.

    Todo esto es realizado al principio del programa, es decir, los tiempos de inserción que mencionamos anteriormente incluyen este proceso también.

III. Análisis de la implementación.

De todas las especificaciones que se piden en el proyecto se logró cumplir con la mayoría de las funciones que se piden; sin embargo, hay algunos aspectos que se pueden mejorar para que el funcionamiento sea mucho mejor.

Dentro de las implementaciones que se lograron realizar, está el poder leer archivos de tipo plano sin problema, leer el contenido de éste y almacenar su contenido dentro de una estructura de rápido acceso en el caso de las oraciones; además, un árbol Trie para las palabras del texto.

La estructura del programa fue implementada correctamente al hacer uso de un diccionario creado con un árbol AVL, también la adaptación de algunas clases como Trie, TrieNode, entre otras. Se elaboró una clase que se encarga de manejar las interacciones directamente con el usuario. Asimismo, se logró que uno de los principales enfoques de este proyecto fuera posible: la eficiencia demostrada anteriormente con los tiempos e implementaciones. Las palabras se almacenan correctamente dentro del Trie, reconociendo bien cada palabra y diferenciando cuando existe algún separador o divisor como un espacio, una coma, un punto entre otros; así

evitando que se escape algún carácter que no compone una palabra, a su vez, gracias a las modificaciones en las clases TrieNode y Trie, se logra hacer la inserción del número de línea donde la palabra aparece escrita dentro del archivo.

Se hizo la selección de una muy buena estructura que se encarga en el manejo de las líneas del archivo, haciendo que su búsqueda sea bastante eficiente evitando largos tiempos. Mostrar la cantidad de veces que una palabra aparece dentro del texto también se pudo implementar, la cual se muestra en funcionamiento en las siguientes aplicaciones del programa: Buscar por cantidad de letras, Ver top de palabras más utilizadas, Ver palabras menos utilizadas.

Las implementaciones de estas funcionalidades fueron realizadas con éxito, en el caso de Buscar una palabra, primero se verifica que la palabra en caso de tener algún carácter especial lo pueda reconocer y así no perder su valor. Buscar por profundidad se logró implementar gracias al uso de la recursión, a su vez cargar la lista de palabras a ignorar se implementó realizando un uso de la estructura árbol Splay. Agregar y eliminar alguna de las palabras a ignorar también se logró realizar exitosamente.

Mostrar las palabras más utilizadas, menos utilizadas y el top de cada una de éstas, se logra haciendo uso de la frecuencia, atributo creado dentro del TrieNode, específicamente para poder tener el control de la aparición de cada palabra. Otra implementación fue el saber manejar caracteres especiales gracias a una función que intercambia los valores de los caracteres escritos por el usuario a los caracteres que contiene el Trie. Esto ayuda a que el programa sepa reconocer y encontrar alguna palabra que contenga algún tipo de carácter del lenguaje español, sea tildes, o diéresis.

Por último, la implementación de un menú donde se pueda interactuar sin problemas entre escoger alguna opción, regresar y salir del programa se realizó en su totalidad.

Dentro de las cosas que faltaron, o más bien que se pueden mejorar dentro del proyecto, es el poder abrir cualquier tipo de archivo de texto sin importar el tipo de *encoding* con el que fue guardado. Solamente se pueden leer archivos con el *encoding* tipo ANSI; sin embargo, nos hubiera encantado el poder leer otros archivos. El anterior punto fue uno de los más duros ya que en el lenguaje de programación C++, lidiar con este tipo de problemas no es tan simple, sino que nos tomó días el tratar de hacer el programa *multi-encoding*, mas no lo logramos; razón la cual optamos por utilizar el tipo de encoding que pudimos leer y procesar sin problemas.

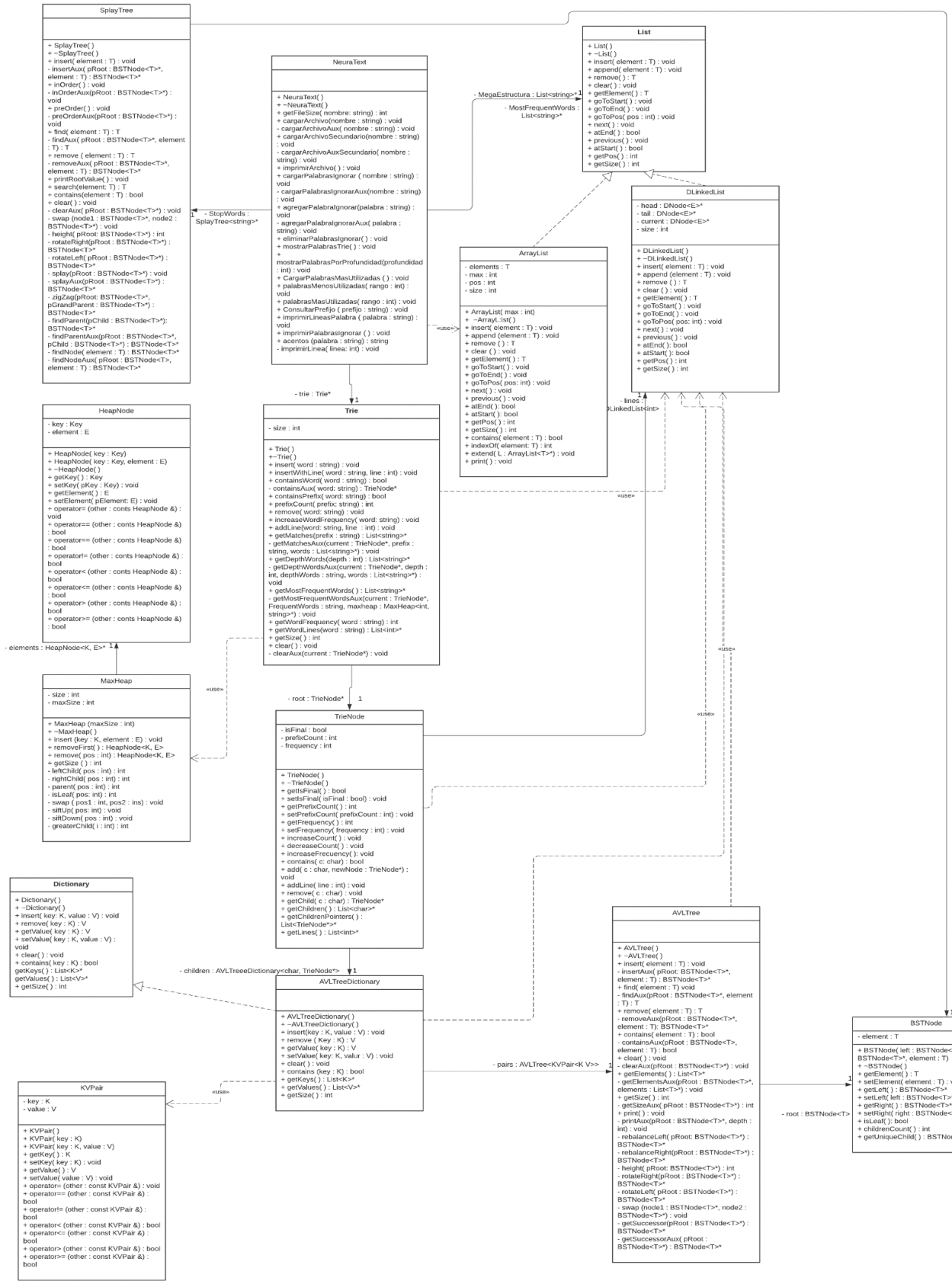
Otro punto que nos tomó tiempo en resolver fue el poder trabajar con caracteres especiales, ya que suponíamos que los caracteres del archivo iban a ser leídos por el programa exactamente igual a los caracteres escritos por el usuario desde el teclado; pero no fue así: teníamos problemas a la hora de buscar alguna palabra ya que no reconocía correctamente estos caracteres del teclado. Para esto pensamos en buscar alguna biblioteca de C++ e importarla para solucionar la situación, pero también nos tomó mucho tiempo, por lo que mejor decidimos hacer nuestra propia función de convertir los caracteres con acentos del teclado a los caracteres con acento del

Trie, ayudando al programa a reconocer correctamente las palabras con algún carácter especial del lenguaje español.

Uno de los grandes avances con respecto a eficiencia que se consiguió en el proceso de creación, fue el cambio de estructura utilizada para almacenar el contenido de una línea, ya que al inicio se optó por el uso de un diccionario tipo AVLTree. Todo parecía bastante bien hasta que probamos la estructura con un libro de tamaño considerablemente grande, Don Quijote de la Mancha, y ahí fue cuando se notó que había algo que no estaba bien dentro del programa ya que su duración en segundos era muy alta. Por lo tanto, se replanteó la idea de guardar las líneas a otra estructura la cuál fue un ArrayList; haciendo este cambio, la duración se redujo bastante ya que la ventaja de los ArrayList es que, su forma de acceso mediante índices, es muy rápida; solamente se necesitaba saber el tamaño de líneas que contiene el archivo para poder crear esta estructura.

Sobre las cosas que se pueden mejorar, como anteriormente se comentó, está el uso de diferentes tipos de encoding, pues nos hubiera encantado poder implementar sin problemas esta funcionalidad ya que haría el programa compatible a cualquier archivo de texto. También pensamos que la manera de convertir caracteres especiales puede mejorarse, ya sea por medio de alguna librería que contenga el lenguaje o una externa.

Diagrama UML



Conclusiones

Dentro de las conclusiones podemos afirmar lo siguiente:

- Procesar archivos de texto en C++ puede resultar un poco difícil debido a que el manejo con el encoding resulta complicado.

- Los caracteres pueden leerse de diferente forma debido a su formato, por lo que hay que tener claro cuál es el que se está usando.

- Si es posible utilizar una estructura con índices basado en la información de la inserción, el tamaño que va a tomar la estructura y qué cosas se ocupan accesar. En este caso lo mejor es utilizarla ya que ofrece una rápida inserción y acceso instantáneo si los elementos están ordenados.

- El árbol Trie es bastante eficiente a la hora de realizar búsquedas, su forma de guardar elementos es diferente ya que no guarda todos los elementos que se le dan si la palabra tiene una misma secuencia de caracteres; éstos no los duplica, por lo que para datos muy grandes toma un buen comportamiento.

  La utilización de la estructura heap es muy conveniente para ordenar ascendente o descendentemente los elementos en otra estructura de una manera eficiente haciendo un tipo de HeapSort.

- El proceso de lectura de texto en C++ para archivos de texto es bastante simple utilizando rutas relativas.

- La importación de bibliotecas grandes para alguna función muy específica, pueden requerir de una investigación extensa en la documentación y código de la biblioteca.

- El árbol Splay es una buena estructura cuando se ocupan accesar muchas veces en una lista grande de elementos algunos de estos en especifico principalmente. Pueden llegar a agregarle bastante eficiencia a lo que se este haciendo si se sabe implementar bien.

- Para archivos muy grandes o donde el orden de inserción sea muy ordenado hacia alguna dirección, la utilización de árboles AVL puede llegar a degradar la eficiencia del programa. Esto debido a que la cantidad de rotaciones que se realizan para balancear el árbol. Estas

rotaciones requieren bastantes operaciones aritméticas que puede saturan al CPU y hacer el programa pesado.

- Es conveniente hacer la menor cantidad de búsquedas para realizar las funciones que se tengan que implementar, de esta manera es posible optimizar la eficiencia del programa que se desee hacer.

- Las validaciones en la interfaz de texto de las entradas del usuario son necesarias para que el programa tenga un funcionamiento óptimo y no se caiga por cualquier cosa.

- El uso de ASCII *art* para la interfaz de texto puede darle un poco mas de personalidad a lo que se esté realizando en consola.

Recomendaciones

A continuación, se mostrará una lista de recomendaciones basadas en las conclusiones que se sacaron:

- Hay que tener mucho cuidado a la hora de procesar archivos de texto con algún tipo de encoding en C++ ya que resulta un poco complicado encontrar información relevante que se aplique a las necesidades que se buscan. Por lo que primero hay que tener claro que es lo que se quiere hacer para luego a la hora de buscar por ejemplo en internet ver qué tipo de artículos funcionan y cuáles no, evitando confusiones.

- Tener en cuenta que el formato de las palabras con caracteres especiales ya sean escritas en el teclado por medio del comando >>cin por ejemplo, escritas dentro de una variable string o palabras dentro de un texto cambian en su representación interna, por ejemplo algunos caracteres se guardan dentro de un byte pero en otras ocasiones depende del encoding un carácter puede requerir más de un byte, por lo que hay que saber cómo hacer que todas estas letras tengan la misma codificación.

- Los ArrayList o arreglos son sumamente recomendados a la hora de acceder a algún valor, ya que su acceso es directo gracias al número de índice, pero si piensa usar el ArrayList para ingresar o eliminar valores haciendo búsquedas o movimientos secuenciales de toda la estructura no es recomendado ya que al crecer esta estructura se va volviendo más ineficiente, por lo que en ese caso es recomendable buscar otro tipo de estructura.

- Recomendamos la utilización del árbol Splay si se ocupa accesar un conjunto de elementos en específico más frecuentemente sobre otro conjunto mucho mas grande. Esto podría agregarle mucha mas eficiencia a lo que se este programando si se implementa bien, sin embargo, es necesario hacer un análisis primero para determinar si resulta conveniente.

- Recomendamos fuertemente usar un heap si se necesitan ordenar elementos de una estructura original para retornarlos en otra estructura sin modificar la estructura original. Esto resulta bastante eficiente para ordenar elementos ascendente y descendentemente ya que se aprovecha la naturaleza de la inserción en los árboles heap además como se puede implementar con un arreglo la extracción de los elementos es bastante rápida también. Esto también se puede describir como un HeapSort.

- Recomendamos optimizar los tiempos de las funciones reduciendo las búsquedas al máximo posible y evitando copiar la misma información de una estructura mediante el uso de punteros.

- Recomendamos siempre el uso de ASCII art para darle un poco mas de personalidad y estética a la interfaz de consola.

- La validación de entradas del usuario es esencial para el funcionamiento óptimo del programa así que es una recomendación que hacemos al elaborar la implementación de la interfaz.

Bibliografía

ICU. (s.f.). *ICU - International Components for Unicode*. Obtenido de ICU - International Components for Unicode: http://site.icu-project.org/

McNellis, J. (2016). Unicode in C++. *Meeting C++ 2016 Unicode in C++*. The United States.

Spolsky, J. (8 de October de 2003). *JOEL ON SOFTWARE*. Obtenido de JOEL ON SOFTWARE: https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/

Goodrich, M. T., Tamassia, R., & Mount, D. M. (2001). Data structures and algorithms in C++" (2nd ed.). Hoboken, NJ: Wiley.

Avilés, M., Ing. (n.d.). *Árboles Avanzados*. Lecture. In *Presentaciones IC-2001* (pp. 1-100). (2020). San Jose, Costa Rica: TEC.

Stack Overflow. (2020, May 8). Stack Overflow. https://stackoverflow.com/

TylerMSFT. (n.d.). Support for Unicode. Retrieved August 05, 2020, from https://docs.microsoft.com/en-us/cpp/text/support-for-unicode?view=vs-2019

Adelyar, S. H. (2008). *The complexity of splay trees and skip lists*. Western Cape, South Africa: University of the Western Cape.

Miyaki, K. (2020, February 18). Basic Algorithms - Heapsort. Retrieved August 05, 2020, from https://towardsdatascience.com/basic-algorithms-heapsort-31d64d6919a1

Std::isalpha. (n.d.). Retrieved August 05, 2020, from http://www.cplusplus.com/reference/locale/isalpha/