Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Estructuras de datos - IC – 2001

I semestre del 2022

Integrantes:

Kendall Guzmán Ramírez

Ian Murillo Campos

Andy Porras Romero

I. Introducción:

En este proyecto se pidió organizar dentro de una estructura de datos llamada Trie las palabras que contenga el archivo cuyo nombre es proporcionado por el usuario, esto con el fin de realizar una búsqueda exhaustiva de las palabras dependiendo de algunas de sus características y atributos, ya sea para buscar las palabras que inician con un prefijo en específico, para conocer cuántas veces ha aparecido una palabra en el archivo o para saber en cuáles líneas aparece dicha palabra, todas estas peticiones a decisión del usuario.

Este proyecto consta de varias partes que se mencionan a continuación, algunas de ellas subdivididas en otras más pequeñas:

A. Lectura de archivos

En este punto se leerá el archivo ignorar.txt en el cual se encuentran las palabras que se deben filtrar del archivo proporcionado por el usuario, esto facilita el descarte de algunas palabras menos importantes con respecto al estudio del contenido de cada archivo que se ingrese, ya sean conjunciones o palabras que se repiten mucho por la naturaleza del texto, en este archivo se incluirán todas y cada una de ellas.

Posteriormente, se leerá el archivo principal proporcionado por el usuario, mientras se leen cada una de sus líneas, se ira descartando las palabras leídas del archivo ignorar.txt y caracteres intermedios entre palabras, por lo que se terminaran incluyendo al trie únicamente palabras completas y relevantes.

B. Menú de usuario

Para la confección y testeo de este proyecto se diseñó una interfaz de texto a través de la cual el usuario puede interactuar con el programa, como se indicó anteriormente se solicitará un nombre de archivo de texto incluyendo su respectiva extensión y camino. En el momento en el que el usuario inserte el nombre del archivo se desplegará un mensaje de espera mientras se lee el archivo, posteriormente se desplegará un menú a través del cuál el usuario podrá seleccionar las siguientes opciones:

Salir

Si el usuario selecciona la opción 0 se dará por terminado el programa y para incluir otro archivo deberá volver a correr el programa.

Consultar por prefijo

Si se inserta la opción 1 en la interfaz de texto se pedirá al usuario un prefijo, y se buscará en la estructura trie las palabras que empiecen con ese prefijo, posteriormente se procederá a imprimir la palabra con la cantidad de veces que aparece en el archivo y las líneas en las que aparece la palabra.

Buscar palabra

Si el usuario inserta el número 2, se pedirá una palabra, y se procederá imprimir los números de las filas en las que aparece esa palabra junto a la respectiva fila.

Buscar por cantidad de letras

Si el inserta el número 3, se pide al usuario un número entero positivo y se busca en el Trie todas las palabras que contengan la misma cantidad de letras que el usuario inserte, se realiza un ciclo en el que se imprime la palabra y la cantidad de veces que aparece en el archivo.

Palabras más utilizadas

Si el usuario inserta 4 se desplegará un menú donde tendrá 4 opciones a elegir que se explican a continuación:

- Agregar palabra a ignorar

Se añadirá una palabra insertada por el usuario al archivo ignorar, para ser usada posteriormente ya sea en esta misma corrida del programa (se agrega al trie “ignorar”) o para ser usada posteriormente en otras corridas del programa (se agrega al archivo “ignorar.txt”).

- Borrar palabra a ignorar

Al igual que la opción pasada se modifica tanto el trie ignorar como el archivo ignorar.txt, pero en este caso para borrar una palabra.

- Ver top

Se solicita al usuario un número que indicará la cantidad de elementos que serán incluidos en el top, en este caso el top está dirigido a la cantidad de apariciones de las palabras en el archivo.

- Regresar

Regresa al menú principal, donde puede seguir realizando operaciones de forma normal.

II. Presentación y análisis del problema

A. Descripción del problema

El principal problema que se manifestó en el desarrollo de este proyecto es relacionado a la lectura de archivos, esto debido a que hay que leer tanto el archivo de palabras a ignorar, como el que el usuario deseé y luego almacenar su contenido en estructuras de datos, el de mayor problema es el del usuario, ya que este debe revisarse y cambiarse en casi cada palabra para añadirlo a la estructura Trie. Primero hay que cambiar las palabras a que sean minúsculas, luego eliminar o revisar todos los tipos de separadores que afecten el algoritmo de inserción del programa y finalmente revisar las palabras con respecto al Trie ignorar para añadirlas o no al Trie principal. Otro gran problema que se nos presentó fue la dificultad para encontrar recursivamente las palabras con una cantidad específica de letras.

Esto requiere algunos cambios al TrieNode que almacena las letras de las palabras en el Trie ya que también se necesita almacenar palabras repetidas, el número de letra dentro de la palabra (esto específicamente para la búsqueda por números de letra) y el número de las filas en las que aparece la palabra

Por otro lado, la lectura de archivos, revisión, e inserción de estos debe realizarse de la manera más eficiente posible al igual que las búsquedas y recorridos de las estructuras. Cabe recalcar que la lectura de archivos requiere de la posibilidad de leer cualquier tipo de archivo de texto (archivos tipo “.py”,”.java”, “.cpp” entre otros).

También se tuvo que implementar la clase MaxHeap la cual fue utilizada para ver el top de palabras más utilizadas en el archivo, pero el hecho de tener que mostrar también la cantidad de veces que aparece y que lo que ingrese sean strings puede

presentar un problema, que finalmente solucionamos con el retorno de un KVPair en el MaxHeap. También se realizó la modificación tanto para añadir como para eliminar alguna palabra del archivo de palabras a ignorar, además de añadirlas y eliminarlas en el trie.

Otro gran problema con el que nos topamos fue a la hora de encontrar una estructura lineal adecuada para guardar nuestros datos, para esto tuvimos que conversar bien para que se iba a utilizar estas estructuras.

B. Metodología

Para la lectura de archivos, se utilizó la biblioteca fstream, esta permite la lectura de diversos archivos de texto, lo que solucionó el leer archivos tipo .py, .cpp, entre otros y también el problema de agregar y eliminar palabras al archivo de palabras por ignorar.

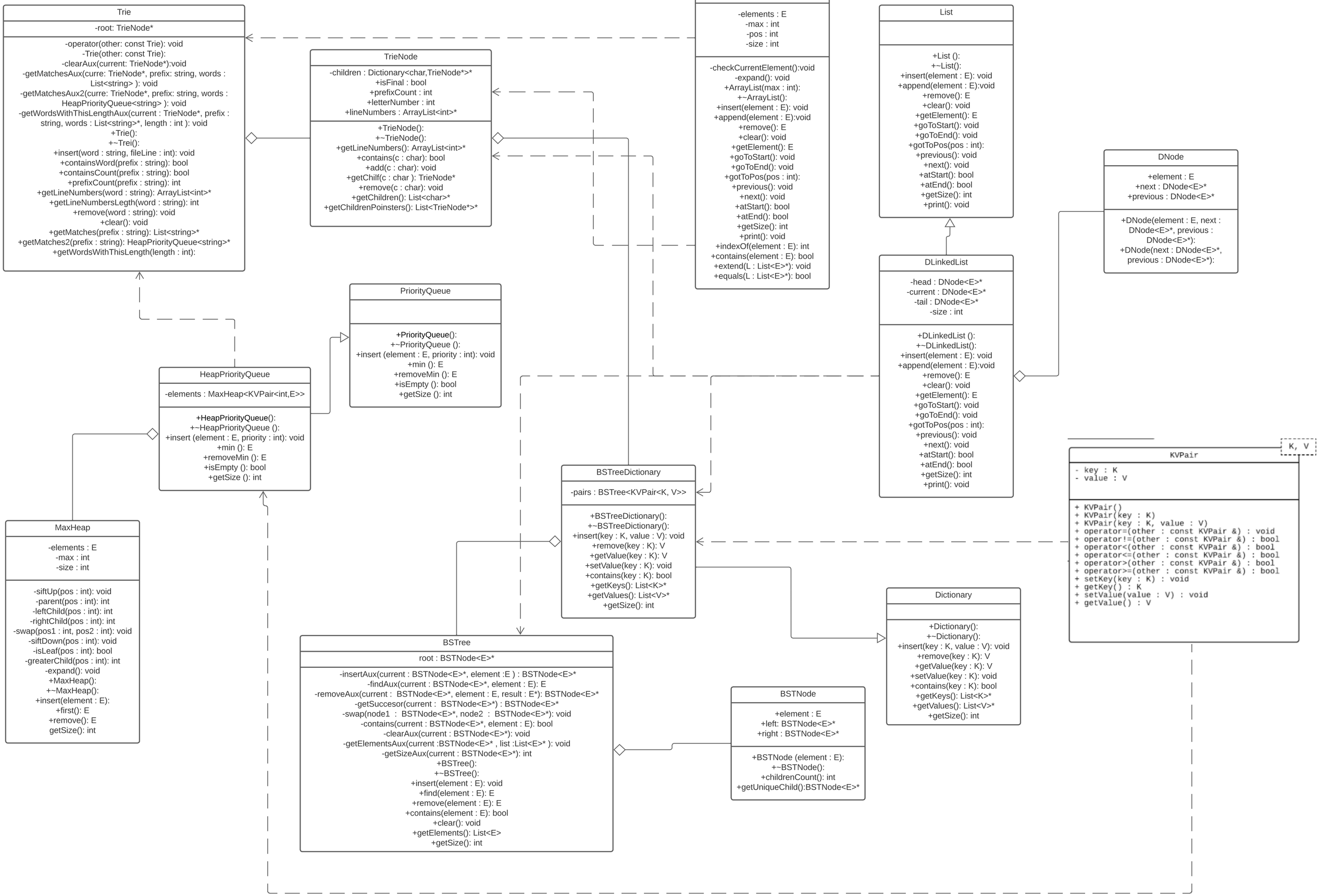
Las bibliotecas iostream y algorithm ayudaron a transformar las palabras leídas del archivo a minúscula para poder compararlas con la lista de palabras por ignorar.

Las palabras por ignorar las guardamos en un Trie para tener una forma más fácil y eficiente de comparar las palabras que entraban al Trie principal desde la lectura del archivo del usuario.

A la clase TrieNode se le añadió un atributo llamado “lineNumbers”, el cual es de tipo ArrayList y sirve para guardar el número de línea en que aparece la palabra y así poder guardar el número de todas las líneas en las que aparece para su fácil consulta, además del atributo “letterNumber” que retorna el número de letra relacionada al nodo dentro de la palabra.

Para el tema del top de palabras utilizando un MaxHeap, se creó la clase y se utilizó un HeapPriorityQueue para almacenar como llave valor la cantidad de veces que aparece una palabra y la palabra, con la cantidad de veces que aparece como prioridad de esta e imprimiendo el tope del HeapPriorityQueue las veces que solicitara el usuario, esto se logró implementar de manera satisfactoria.

A continuación, se adjunta el diagrama UML a partir del cual desarrollamos el código del proyecto, esto fue muy beneficioso para guiarnos y mejoró nuestro entendimiento del problema:


Trie
-root: TrieNode*
-operator(other: const Trie): void
-Trie(other: const Trie):
-clearAux(current: TrieNode*):void
-getMatchesAux(curre: TrieNode*, prefix: string, words : List<string>): void
-getMatchesAux2(curre: TrieNode*, prefix: string, words : HeapPriorityQueue<string>): void
-getWordsWithThisLengthAux(current : TrieNode*, prefix : string, words : List<string>*, length : int): void
+Trie():
+~Trei():
+insert(word : string, fileLine : int): void
+containsWord(prefix : string): bool
+containsCount(prefix : string): bool
+prefixCount(prefix : string): int
+getLineNumbers(word : string): ArrayList<int>*
+getLineNumbersLegth(word : string): int
+remove(word : string): void
+clear(): void
+getMatches(prefix : string): List<string>*
+getMatches2(prefix : string): HeapPriorityQueue<string>*
+getWordsWithThisLength(length : int):
TrieNode
-children : Dictionary<char,TrieNode*>*
+isFinal : bool
+prefixCount : int
+letterNumber : int
+lineNumbers : ArrayList<int>*
+TrieNode():
+~TrieNode():
+getLineNumbers(): ArrayList<int>*
+contains(c : char): bool
+add(c : char): void
+getChilf(c : char): TrieNode*
+remove(c : char): void
+getChildren(): List<char>*
+getChildrenPoinsters(): List<TrieNode*>*
ArrayList
-elements : E
-max : int
-pos : int
-size : int
-checkCurrentElement():void
-expand(): void
+ArrayList(max : int):
+~ArrayList():
+insert(element : E): void
+append(element : E):void
+remove(): E
+clear(): void
+getElement(): E
+goToStart(): void
+goToEnd(): void
+gotToPos(pos : int):
+previous(): void
+next(): void
+atStart(): bool
+atEnd(): bool
+getSize(): int
+print(): void
+indexOf(element : E): int
+contains(element : E): bool
+extend(L : List<E>*): void
+equals(L : List<E>*): bool
List
+List ():
+~List():
+insert(element : E): void
+append(element : E):void
+remove(): E
+clear(): void
+getElement(): E
+goToStart(): void
+goToEnd(): void
+gotToPos(pos : int):
+previous(): void
+next(): void
+atStart(): bool
+atEnd(): bool
+getSize(): int
+print(): void
DNode
+element : E
+next : DNode<E>*
+previous : DNode<E>*
+DNode(element : E, next : DNode<E>*, previous : DNode<E>*):
+DNode(next : DNode<E>*, previous : DNode<E>*):
DLinkedList
-head : DNode<E>*
-current : DNode<E>*
-tail : DNode<E>*
-size : int
+DLinkedList ():
+~DLinkedList():
+insert(element : E): void
+append(element : E):void
+remove(): E
+clear(): void
+getElement(): E
+goToStart(): void
+goToEnd(): void
+gotToPos(pos : int):
+previous(): void
+next(): void
+atStart(): bool
+atEnd(): bool
+getSize(): int
+print(): void
PriorityQueue
+PriorityQueue():
+~PriorityQueue ():
+insert (element : E, priority : int): void
+min (): E
+removeMin (): E
+isEmpty (): bool
+getSize (): int
HeapPriorityQueue
-elements : MaxHeap<KVPair<int,E>>
+HeapPriorityQueue():
+~HeapPriorityQueue ():
+insert (element : E, priority : int): void
+min (): E
+removeMin (): E
+isEmpty (): bool
+getSize (): int
MaxHeap
-elements : E
-max : int
-size : int
-siftUp(pos : int): void
-parent(pos : int): int
-leftChild(pos : int): int
-rightChild(pos : int): int
-swap(pos1 : int, pos2 : int): void
-siftDown(pos : int): void
-isLeaf(pos : int): bool
-greaterChild(pos : int): int
-expand(): void
+MaxHeap():
+~MaxHeap():
+insert(element : E):
+first(): E
+remove(): E
getSize(): int
BSTreeDictionary
-pairs : BSTree<KVPair<K, V>>
+BSTreeDictionary():
+~BSTreeDictionary():
+insert(key : K, value : V): void
+remove(key : K): V
+getValue(key : K): V
+setValue(key : K): void
+contains(key : K): bool
+getKeys(): List<K>*
+getValues(): List<V>*
+getSize(): int
BSTree
root : BSTNode<E>*
-insertAux(current : BSTNode<E>*, element :E) : BSTNode<E>*
-findAux(current : BSTNode<E>*, element : E): E
-removeAux(current : BSTNode<E>*, element : E, result : E*): BSTNode<E>*
-getSuccesor(current : BSTNode<E>*) : BSTNode<E>*
-swap(node1 : BSTNode<E>*, node2 : BSTNode<E>*): void
-contains(current : BSTNode<E>*, element : E): bool
-clearAux(current : BSTNode<E>*): void
-getElementsAux(current :BSTNode<E>* , list :List<E>*): void
-getSizeAux(current : BSTNode<E>*): int
+BSTree():
+~BSTree():
+insert(element : E): void
+find(element : E): E
+remove(element : E): E
+contains(element : E): bool
+clear(): void
+getElements(): List<E>
+getSize(): int
BSTNode
+element : E
+left: BSTNode<E>*
+right : BSTNode<E>*
+BSTNode (element : E):
+~BSTNode():
+childrenCount(): int
+getUniqueChild():BSTNode<E>*
Dictionary
+Dictionary():
+~Dictionary():
+insert(key : K, value : V): void
+remove(key : K): V
+getValue(key : K): V
+setValue(key : K): void
+contains(key : K): bool
+getKeys(): List<K>*
+getValues(): List<V>*
+getSize(): int
K, V
KVPair
- key : K
- value : V
+ KVPair()
+ KVPair(key : K)
+ KVPair(key : K, value : V)
+ operator=(other : const KVPair &) : void
+ operator!=(other : const KVPair &) : bool
+ operator<(other : const KVPair &) : bool
+ operator<=(other : const KVPair &) : bool
+ operator>(other : const KVPair &) : bool
+ operator>=(other : const KVPair &) : bool
+ setKey(key : K) : void
+ getKey() : K
+ setValue(value : V) : void
+ getValue() : V

C. Análisis crítico

¿Qué se logró implementar?

Se logró implementar todas las estructuras planteadas en el proyecto, entre ellas el Trie actualizado de acuerdo con las necesidades del problema planteado en el proyecto, esta estructura de datos conteniendo otras estructuras dentro de ella que fueron igual de importantes en esta implementación como el ArrayList y el BSTree.

Se consideró que para la utilización de listas lineales la mejor opción en este caso era la implementación de ArrayList, por su facilidad de inserción por medio del método append() y de obtención por medio del método getElement() en comparación con los LinkedList o DLinkedList además de la nula utilización de borrado de elementos que es en lo que falla la eficiencia del ArrayList.

¿Qué faltó?

Se logró implementar satisfactoriamente todo lo solicitado en el proyecto, sin embargo, se pudo implementar nuevas estructuras o algoritmos similares a los presentados en este proyecto para desarrollar código más limpio o reducido.

¿Qué cosas se podrían mejorar de lo que se implementó?

Algo que se pudo hacer mejor fue la reducción del tiempo de ejecución del programa al enviar el menú antes de leer el archivo así mientras el usuario ve el menú y se decide a escoger su opción ya el archivo se ha ido leyendo y el programa ya para su ejecución para pedir la opción del usuario, de esta forma se pierde menor tiempo de ejecución.

Otro aspecto en el cuál pudimos mejorar fue al evitar el uso de separadores entre los despliegues, si lo que se desea es reducción total del tiempo de ejecución, sin embargo, para efectos de este proyecto y para el testeo y la revisión del programa decidimos dejar las impresiones lo más explícitas posibles para facilitar el trabajo de revisión.

Durante la implementación observamos que en las estructuras lineales implementadas únicamente utilizábamos el método append para inserción, por lo cual tal vez pudimos haber hecho una modificación de la estructura Queue agregándole

métodos para recorrer la estructura y obtener cada uno de sus elementos sin borrarlos.

III. Conclusiones

Trie es una excelente estructura para las ocasiones en las que hay que manejar datos de archivos de texto muy grandes y es muy maleable para obtener datos secundarios de estos archivos o para realizar gráficas sobre mayor o menor aparición de palabras en uno u otro archivo y hacer comparaciones entre archivos.

Para estructuras lineales en las cuales no se debe borrar ninguno de sus elementos en tiempo de ejecución la mejor opción es un ArrayList, esto ayuda en sobremanera a reducir el tiempo de ejecución del programa.

Los métodos recursivos son muy útiles para desarrollar Clean Code, pues permiten que se evite la utilización de ciclos en algunas funciones, y en algunos casos permiten que el código sea más agradable a la vista.

Las estructuras de datos en general son herramientas muy importantes para la abstracción, visualización y procesamiento de datos y al utilizarlas dentro de otras estructuras permiten a los programadores desarrollar gran cantidad de aplicaciones.

Los árboles de búsqueda binaria son la mejor opción para búsqueda de elementos con características jerárquicas, mientras que para elementos con una estructura lineal (sin utilización del método borrar) se maneja en mejor manera la estructura ArrayList.

La mejor manera de utilizar ciclos es utilizarlos solo cuando sea necesario, en el caso de este proyecto utilizamos un único ciclo para generar todas las estructuras y restricciones necesarias para introducir los elementos en el Trie desde el archivo y además incluir las filas sin modificar desde el archivo.

El archivo sys/stat.h es sumamente útil para búsqueda de archivos y su método stat se presta para búsqueda ya sea en un archivo en específico o en el archivo en donde se encuentra el programa.

La estructura MaxHeap es perfecta para ordenamiento de datos y su inclusión en la estructura HeapPriorityQueue con KVPair genera un excelente método para ordenar strings ya no solo dependiendo de la cantidad de apariciones si no dependiendo del orden de aparición o de cualquier otro atributo entero de los strings.

Los diccionarios permiten enlazar listas de datos en pares de manera excepcional y el uso de KVPair ya no solo en estructuras lineales si no en estructuras jerárquicas permite que este sea unas de las estructuras más utilizadas.

Cada una de las líneas de un archivo funciona como un string, esto es un conocimiento sumamente útil principalmente para la búsqueda de palabras o caracteres dentro de un archivo.

La encapsulación de código en métodos secundarios permite procesar muchísima información con en pocas líneas de código, y permite que el código sea más entendible.

IV. Recomendaciones

No tener miedo a utilizar atributos en las estructuras. En ocasiones la solución más simple a un problema con estructuras jerárquicas con nodos es incluir atributos para los nodos que guarden información en lugar de realizar métodos sumamente complejos y que no mejoran en sobremanera la eficiencia

Utilizar métodos recursivos cuando realmente haga más sencillo y eficiente el código, de esta manera se genera un código más limpio y fácil de leer.

Realizar todos los procesos de un string a la hora de leer cada una de sus líneas, no realizar otro ciclo pues esto consumiría más tiempo de ejecución y sería menos eficiente, los ciclos no son malos cuando se utilizan únicamente los necesarios, simplemente se deben usar sabiamente.

No hay que tener miedo a reutilizar código de unas estructuras para introducirlas como atributos dentro de otras estructuras, pues esto simplifica ampliamente el trabajo del programador al dividir el código en bloque y de cierta manera “encapsularlo”.

Cuando el código se haga demasiado largo o poco entendible siempre encapsularlo en métodos secundarios, esto permitirá generar algoritmos más sencillos que ya probados pueden servir para algoritmos más grandes y con menor capacidad de ser entendibles, ni para el creador del código ni para los compañeros de este que deseen entender el código.

Las listas son una maravilla, pero se deben utilizar sabiamente y de manera específica se debe saber para que se van a utilizar porque en ocasiones pueden ser

sustituidas por otras estructuras lineales que se utilizan de manera similar pero que son más eficientes para una función en específico.

Realizar más de una prueba y utilizar otros tipos de estructura, no solo usar una si no varias, dependiendo de lo que se necesite para mejorar la eficiencia del programa, además de hacer pruebas con las diferentes estructuras, para verificar la eficiencia que tiene y así usar la que trae mayor beneficio en eficiencia.

Realizar comentario de cada función realizada, aunque sea un comentario corto, pero que mencione que realiza, para así cuando un compañero vea el código pueda comprender para que sirve y así poder comprender en que caso se podría usar ese método o cuando se podría reutilizar.

V. Referencias

Cabrera, L. (s.f.). *replit*. Obtenido de https://replit.com/@parzibyte/LeerArchivoEnC#main.cpp

DelftStack. (17 de Diciembre de 2020). Obtenido de https://www.delftstack.com/es/howto/c/c-check-if-file-exists/#:~:text=completa%20del%20archivo.-,stat()%20Funci%C3%B3n%20para%20comprobar%20si%20un%20archivo%20existe%20en,si%20el%20archivo%20no%20existe.&text=El%20programa%20imprimir%C3%A1%20file%20exist

DelftStack. (21 de Febrero de 2021). Obtenido de https://www.delftstack.com/es/howto/cpp/how-to-convert-string-to-lower-case-in-cpp/