Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación
IC 2001 - Estructuras de Datos
Grupo 40

Proyecto 1
Indización de textos mediante Tries.

Estudiantes:
José Manuel Quesada Carvajal - 2021082885
Christopher Jiménez Gutiérrez - 2021052665

Profesor: Mauricio Avilés Cisneros

Fecha de entrega: 21 de noviembre del 2021

Introducción.

A la hora de leer textos de mucha extensión, en muchas veces se quiere recordar una línea, pero, es muy difícil encontrarla entre las cientos(o más) páginas con las que se está trabajando. O de la misma manera, al estar realizando una investigación, es común buscar en distintos documentos palabras en específico, buscando información relevante para el trabajo, y evitando leer párrafos extensos que no aporten con el objetivo. O también, otra situación que se puede presentar es querer saber las palabras más repetidas en un texto, para así poder llevar un conteo estadístico con distintos fines. Para estos problemas existen algunas herramientas, como el comando ctrl+f en el teclado, el cual funciona en distintos sitios webs y aplicaciones, pero no todas las personas lo manejan de la mejor manera.

Una herramienta eficiente de análisis de las palabras presentes en un escrito, puede mejorar la eficiencia de muchos problemas del ser humano, por ejemplo:

- Buscar una palabra en un escrito, y acceder directamente a la línea en la que aparece.
- Analizar los prefijos más utilizados en un texto.
- Analizar las palabras más usadas en archivos de texto, lo cual puede ayudar a realizar otras aplicaciones.
- Revisiones ortográficas, al saber palabras que se repiten mucho en una redacción.

Por esto, en el presente proyecto se desea desarrollar un software de indización de texto, el cual se manejará mediante una estructura de datos Trie, en la que se irán almacenando las palabras presentes en el texto mediante un TrieNode, el cual maneja una lista que tiene los índices de los números de líneas en las que se encuentra la palabra, para facilitar su acceso posteriormente. De la misma forma, el software se apoyará en otras estructuras de datos, como por ejemplo: DLinkedList, maxheap, UnsortedArrayList, entre otras

Esto se realiza en búsqueda de mejorar los procesos de análisis de palabras en escritos, para facilitar tareas diarias del ser humano. Además, de la misma manera se busca aplicar, y aprender más, sobre distintos conceptos de programación: lenguaje de C++, manejo de archivos en C++, comportamiento de la estructura Trie, búsqueda de eficiencia en programas, entre otros.

Descripción del problema.

Se pretende realizar un sistema de indización de textos. El mayor reto que se presenta es lograr que el software funcione de la manera más eficiente posible. Este tiene que ser capaz de procesar miles de líneas en un par de segundos, ya que un funcionamiento veloz es clave para el aprovechamiento de sus características. Pära lograrlo, surgieron varios subproblemas:

- Se deben decidir las estructuras de datos que se utilizarán para almacenar las palabras, esta decisión tiene un peso grande en el producto final, ya que la eficiencia de las estructuras para trabajar con este tipo de datos, afectará en gran parte en la eficiencia final del proyecto.
- Se debe guardar cada línea del escrito en memoria, para poder accederlas de manera rápida en caso de que se necesite imprimirlas en pantalla, ya que leer el archivo cada vez que se necesita imprimir una línea sería un proceso muy lento de ejecutar.
- Se debe ajustar el código de la estructura de datos a usar(en este caso trie con un TrieNode), para que esta almacene los números de líneas que contienen la palabra añadida.
- Se deben usar métodos eficientes de apertura, lectura y edición de archivos, para que así el programa pueda ser capaz de procesar miles de líneas de forma rápida.
- Se necesita añadir funciones al Trie, para poder filtrar las palabras por su tamaño de forma rápida. En esto se busca evitar recorrer todas las palabras cada vez que se busca un tamaño, ya que podría generar lentitud en el programa.

- Se debe definir una estrategia, para poder acceder fácilmente a la cantidad de veces que aparece una palabra en el texto. Además, a la hora de imprimir las palabras más usadas, se tiene que implementar un “Ignorar.txt”, que contendrá palabras que no se deben mostrar en el top de palabras más usadas.
- Se requiere programar una interfaz de consola que muestre todo lo realizada, que permita ingresar un archivo y de sus datos. Mediante esto, se puede medir y mostrar la eficiencia del software.

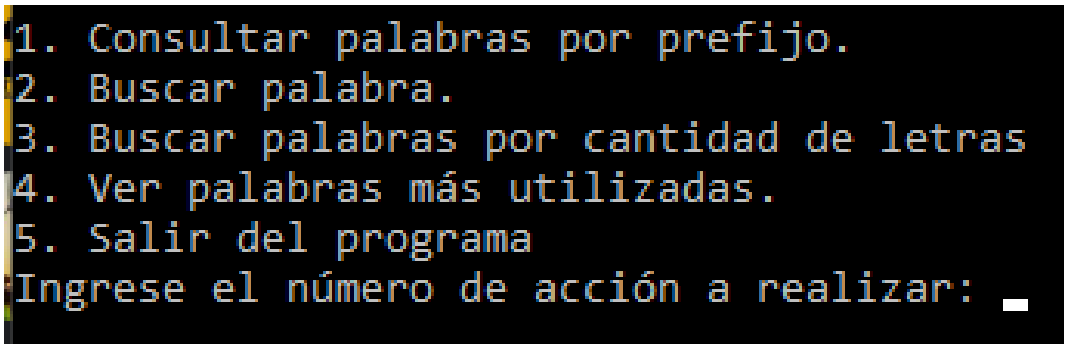
Metodología.

Como se mencionó anteriormente, se decidió usar una estructura Trie para almacenar las palabras. Esta utiliza un TrieNode para almacenar los elementos. Estas estructuras fueron programadas por el profesor Mauricio Avilés, y modificadas en el proyecto para adaptarlas a los objetivos. (comunicación personal). Dentro del TrieNode, usando el nodo final de cada palabra, se implementó una variable que guarda la cantidad de letras y una lista que guarda el número de líneas en las que aparecen las palabras, para así poder acceder a todos estos datos de la maneras más rápida posible.

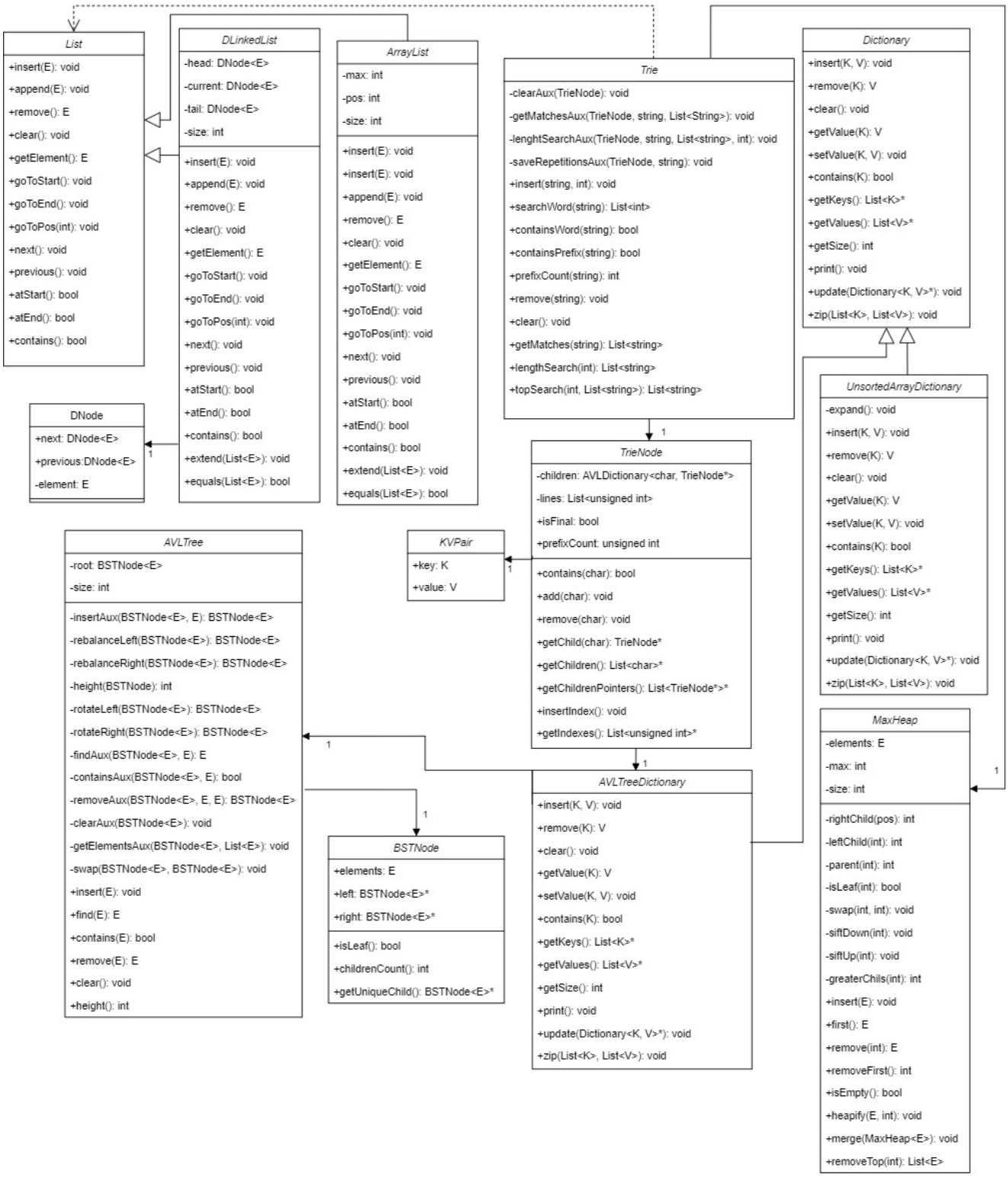
Para guardar las líneas se decidió utilizar una lista doblemente enlazada, la cual almacena y puede ser accedida de una manera súper sencilla, en procesos que no duran ni un segundo, sin importar que tenga que manejar miles de datos. Las líneas se almacenan abriendo el archivo, leyéndolo mediante la clase std::if, y agregando cada una de las líneas del archivo a la lista. Posteriormente, se recorre toda la lista palabra por palabra, y, mediante la ayuda de la función isAlpha() de C++, se inserta cada palabra en la estructura Trie. Además, anteriormente la estructura Trie lanzaba un error si se intentaba insertar la palabra varias veces. En este caso, se editó el comportamiento para que si la palabra era repetida, no diera error, sino que almacene el índice de la línea en la lista del TrieNode. Además, se agregó que en caso de que la palabra sea vacía no ejecute el insert, para evitar errores en casos en los que el texto tenía varios espacios seguidos.

Para la cantidad de veces que aparece una palabra, se agregó una función topSearch(), que recorre el Trie al final de las inserciones y utiliza un diccionario UnsortedArrayDictionary.h para guardar palabra-valor, lo cual es posteriormente ordenado con la ayuda de un MaxHeap.

Para probar la eficiencia, se creó un pequeño programa de consola que pide al usuario un texto, lo intenta abrir y procesa todas sus líneas y palabras. Posteriormente, le da la opción al usuario de consultar por prefijo, por palabra, por cantidad de letras y por cantidad de apariciones; de la siguiente manera:



En el siguiente diagrama de clases se presenta la estructura del proyecto:


List
+insert(E): void
+append(E): void
+remove(): E
+clear(): void
+getElement(): E
+goToStart(): void
+goToEnd(): void
+goToPos(int): void
+next(): void
+previous(): void
+atStart(): bool
+atEnd(): bool
+contains(): bool
DLinkedList
-head: DNode<E>
-current: DNode<E>
-tail: DNode<E>
-size: int
+insert(E): void
+append(E): void
+remove(): E
+clear(): void
+getElement(): E
+goToStart(): void
+goToEnd(): void
+goToPos(int): void
+next(): void
+previous(): void
+atStart(): bool
+atEnd(): bool
+contains(): bool
+extend(List<E>): void
+equals(List<E>): bool
ArrayList
-max: int
-pos: int
-size: int
+insert(E): void
+insert(E): void
+append(E): void
+remove(): E
+clear(): void
+getElement(): E
+goToStart(): void
+goToEnd(): void
+goToPos(int): void
+next(): void
+previous(): void
+atStart(): bool
+atEnd(): bool
+contains(): bool
+extend(List<E>): void
+equals(List<E>): bool
Trie
-clearAux(TrieNode): void
-getMatchesAux(TrieNode, string, List<String>): void
-lenghtSearchAux(TrieNode, string, List<string>, int): void
-saveRepetitionsAux(TrieNode, string): void
+insert(string, int): void
+searchWord(string): List<int>
+containsWord(string): bool
+containsPrefix(string): bool
+prefixCount(string): int
+remove(string): void
+clear(): void
+getMatches(string): List<string>
+lengthSearch(int): List<string>
+topSearch(int, List<string>): List<string>
Dictionary
+insert(K, V): void
+remove(K): V
+clear(): void
+getValue(K): V
+setValue(K, V): void
+contains(K): bool
+getKeys(): List<K>*
+getValues(): List<V>*
+getSize(): int
+print(): void
+update(Dictionary<K, V>*): void
+zip(List<K>, List<V>): void
DNode
+next: DNode<E>
+previous:DNode<E>
-element: E
TrieNode
-children: AVLDictionary<char, TrieNode*>
-lines: List<unsigned int>
+isFinal: bool
+prefixCount: unsigned int
+contains(char): bool
+add(char): void
+remove(char): void
+getChild(char): TrieNode*
+getChildren(): List<char>*
+getChildrenPointers(): List<TrieNode*>*
+insertIndex(): void
+getIndexes(): List<unsigned int>*
UnsortedArrayDictionary
-expand(): void
+insert(K, V): void
+remove(K): V
+clear(): void
+getValue(K): V
+setValue(K, V): void
+contains(K): bool
+getKeys(): List<K>*
+getValues(): List<V>*
+getSize(): int
+print(): void
+update(Dictionary<K, V>*): void
+zip(List<K>, List<V>): void
AVLTree
-root: BSTNode<E>
-size: int
-insertAux(BSTNode<E>, E): BSTNode<E>
-rebalanceLeft(BSTNode<E>): BSTNode<E>
-rebalanceRight(BSTNode<E>): BSTNode<E>
-height(BSTNode): int
-rotateLeft(BSTNode<E>): BSTNode<E>
-rotateRight(BSTNode<E>): BSTNode<E>
-findAux(BSTNode<E>, E): E
-containsAux(BSTNode<E>, E): bool
-removeAux(BSTNode<E>, E, E): BSTNode<E>
-clearAux(BSTNode<E>): void
-getElementsAux(BSTNode<E>, List<E>): void
-swap(BSTNode<E>, BSTNode<E>): void
+insert(E): void
+find(E): E
+contains(E): bool
+remove(E): E
+clear(): void
+height(): int
KVPair
+key: K
+value: V
BSTNode
+elements: E
+left: BSTNode<E>*
+right: BSTNode<E>*
+isLeaf(): bool
+childrenCount(): int
+getUniqueChild(): BSTNode<E>*
AVLTreeDictionary
+insert(K, V): void
+remove(K): V
+clear(): void
+getValue(K): V
+setValue(K, V): void
+contains(K): bool
+getKeys(): List<K>*
+getValues(): List<V>*
+getSize(): int
+print(): void
+update(Dictionary<K, V>*): void
+zip(List<K>, List<V>): void
MaxHeap
-elements: E
-max: int
-size: int
-rightChild(pos): int
-leftChild(int): int
-parent(int): int
-isLeaf(int): bool
-swap(int, int): void
-siftDown(int): void
-siftUp(int): void
-greaterChils(int): int
+insert(E): void
+first(): E
+remove(int): E
+removeFirst(): int
+isEmpty(): bool
+heapify(E, int): void
+merge(MaxHeap<E>): void
+removeTop(int): List<E>
1

Análisis crítico.

La adaptación de la estructura Trie dio buen desempeño, funcionando correctamente y dando los resultados esperados. En cuestión de eficiencia, el proceso no pasa de pocos segundos, lo cual cumple las expectativas que se tenían sobre el programa. Además, la interfaz en consola muestra, con creces, la funcionalidad y eficacia del programa, permitiéndole a la gente procesar cualquier archivo de texto que esté en formato ANSI.

Refiriéndose al formato ANSI, este representó una dificultad en el trabajo. Por el tipo de manejo de archivos que se usó(y que en general se usa en C++), se generaban errores al usar archivos de texto con formato UTF-8, u cualquier otro formato que no sea ANSI. Por esto, es importante aclarar, que el programa requiere documentos en formato ANSI para funcionar bien. De lo contrario, se podrían generar fallos. Esto, en parte, es una limitación del software, que se podría buscar solucionar utilizando otras bibliotecas para manejar los archivos, o desarrollando el programa en otro lenguaje de programación.

La implementación de un diccionario y un MaxHeap para ordenar las palabras por su cantidad de operaciones salió de manera exitosa. Todas las palabras se ordenan bien, y el proceso se da en un par de segundos, lo cual no afecta mucho en la eficiencia del programa en general, ya que aunque requiere un poco más de tiempo, genera que los datos sean muy fáciles de acceder a la hora de que el usuario los requiera.

La aplicación de consola logra su función de buena forma, el usuario puede configurar el texto que quiere trabajar, la cantidad de palabras que quiere ver en el top de palabras más usadas, la palabra o prefijo que quiere buscar y las palabras que desea ignorar en la lista de palabras más usadas. Tal vez, se podría mejorar el programa implementando una interfaz gráfica, para que sea más amigable para el

usuario, mediante un menú en el que se puedan acceder a todas las funciones del software.

En general, todas las estructuras utilizadas en el proyecto aportaron a que la carga de textos sea eficiente. Tal vez, una mejora para ordenar de mejor forma el código, sería programar un método getElement en la clase list, que reciba un número entero de índice. De esta forma, la lista se encarga de llamar a la función goToPos() y obtiene el elemento en la posición deseada. Además, se podría agregar una función que mida el tiempo exacto de carga de los archivos, para medir su eficiencia.

Conclusiones.

- La estructura Trie utilizada es bastante eficiente para guardar gran cantidad de palabras, permitiendo acceder de forma muy veloz a la palabra, sus índices y la cantidad de veces que aparece.
- La utilización de la DLinkedList fue un acierto, ya que no tiene problemas de velocidad al manejar miles y miles de strings.
- Para lograr la eficacia deseada, fue vital cargar los datos en memoria, ya que leer el documento cada vez que el usuario pide algo podría ser un proceso bastante lento.
- A la hora de manejar archivos con la biblioteca std en C++, se encontraron problemas al utilizar codificaciones como UTF-8, por lo que el programa solo funciona al 100% con documentos de texto en codificación ANSI.
- A la hora de programar estructuras de datos propias, es importante desarrollarlas con tipos de datos genéricos, para así poder asignarle cualquier tipo de dato cuando se necesite usar.
- Al crear un TrieNode para almacenar palabras, se puede utilizar el último nodo de cada string para almacenar datos relacionados a una palabra, como la lista con índices utilizada en el presente trabajo.
- La bandera std::ofstream::trunc es útil para rehacer desde 0 archivos que ya existen, esta funcionó a la perfección para editar la lista de las palabras a ignorar.

- Las clases usadas para manejo de archivos(std::ifstream y std::ofstream) ayudan bastante a manejar errores. Ya que el programa no se cae si el archivo que se intenta abrir no existe, solo se asigna falso a la función isOpen(), con la cual se puede consultar si hubo un error abriendo el archivo.
- Para buscar la mayor eficiencia es importante programar contra clases abstractas, ya que mediante la clase “List.h”, se pueden probar todos los tipos de listas que heredan de esta clase, buscando cúal tipo de lista logra mayor eficiencia con los datos dados.
- A la hora de programar métodos grandes, es mejor dividirlos en pequeñas tareas, y crear subrutinas para cada tarea. Esto genera que el código principal sea más corto y legible, asignando funciones con nombres significativos.
- Se concluye que al realizar copias de objetos, hay que tener cuidado con evitar referencias, si no se desea alterar el objeto original. Para esto, se requiere realizar una deep copy.

Recomendaciones.

- Se recomienda programar una estructura Trie si se desea llevar a cabo un proyecto similar al presente, para poder manejar las palabras y los datos de forma eficiente.
- Se recomienda utilizar listas doblemente enlazadas para trabajar con grandes cantidades que pueden variar, ya que tiene una capacidad más flexible y dependiendo de su uso, puede utilizar menos memoria que otras listas.
- Se sugiere no recorrer un archivo de texto cada vez que se ocupa información, sino cargarlo todo en memoria para poder accederlo de una manera más sencilla.
- Se aconseja utilizar la codificación ANSI si se va a trabajar con la biblioteca de std de C++, si se requiere otra codificación, se sugiere usar otra biblioteca u otro lenguaje de programación.

- Al programar estructuras de datos, se aconseja utilizar tipos de datos genéricos, para poder utilizarlas con distintos objetos. Esto se logra con la siguiente línea:

- Si se requiere almacenar datos de las palabras insertadas en tries, se recomienda almacenarlos en el último nodo de la palabra en el trienode, ya que este se ´puede acceder fácilmente con el método isFinal.
- Si se requiere escribir de 0 en c++ un archivo que ya existe, se recomienda usar la bandera trunc, la cual se puede llamar mediante la siguiente línea:

```
newFile.open("Ignorar.txt", ofstream::out | ofstream::trunc);
```

- Se recomienda utilizar la función isOpen a la hora de leer archivos mediante de la biblioteca std::ifstream, ya que si esta devuelve el valor false, se sabe que ocurrió un error abriendo el archivo.
- Se recomienda programar contra clases abstractas, para así poder crear distintas implementaciones de una sola estructura, lo cual permite tener varias opciones para usar una estructura, pudiendo probar varias buscando cuál es la que da mejor resultado.
- Al desarrollar un programa similar, u otro tipo de programa, se recomienda no escribir todo el código en el main. Es mejor escribir métodos que realicen tareas, y tengan nombres que describan estas tareas. De esta manera, se dividen los problemas en subproblemas, y el código principal es más corto y legible.
- Se sugiere realizar copias del tipo “deep copy” cuando se necesite respaldar o duplicar un objeto, para así evitar modificar el objeto original. Una forma común de realizar esto, es recorrer todo el objeto e ir copiando todos sus elementos.

Referencias.

Cplusplus.com. (2021). Ifstream. Recuperado de: https://www.cplusplus.com/reference/fstream/ifstream/

Meza, J. Condicional switch en C++. (Enero 2021). Declaración uso y sintaxis de switch en C++. Recuperado de: https://www.programarya.com/Cursos/C++/Condicionales/Condicional-switch

Vaca, A. (Marzo 20, 2011). Manejo de archivos en c++. Recuperado de: http://www.programacionenc.net/index.php?option=com_content&view=article&id=69:manejo-de-archivos-en-c&catid=37:programacion-cc&Itemid=55