Instituto Tecnológico de Costa Rica

Bachillerato de Ingeniería en Computación

IC-2001 Estructuras de Datos - Prof. Mauricio Avilés

Proyecto Programado – Indización de texto con Tries

Introducción

Los **tries** son una estructura jerárquica o de árbol que se utiliza como arreglo asociativo de llaves tipo *string* y valores de cualquier tipo. Este tipo de estructura permite una búsqueda rápida de las llaves y es especialmente útil cuando se desea encontrar llaves que tienen el mismo prefijo, es decir, llaves que inician con una misma secuencia de caracteres. Los **tries** son utilizados para implementar funcionalidades cotidianas del software de hoy en día como el texto predictivo en teléfonos celulares, sugerencias en buscadores de internet, sugerencias de comandos y revisión de ortografía.

El actual proyecto consiste en la implementación de un árbol trie con palabras obtenidas de un archivo de texto o PDF con el fin de consultar y obtener información relevante sobre el archivo analizado como:

1. Consultar una palabra o prefijo y obtener la lista de palabras que coinciden.
2. Para cada palabra coincidente se muestra una lista con las líneas de texto donde aparece la palabra.
3. Buscar palabras por cantidad de letras.
4. Ordenar las palabras por frecuencia de uso.

Lógica

A continuación, se detalla la estructuración de la lógica del proyecto. El diseño se basa en la construcción de una clase **Trie** que es la estructura de árbol que se encarga de hacer las operaciones principales con *strings*. El **Trie** utiliza nodos tipo **TrieNode** para guardar su información. El modelo que se presenta a continuación no está completo, pero puede usarse como base del proyecto, el grupo de trabajo debe realizar las modificaciones necesarias para lograr el objetivo del proyecto de la forma más eficiente posible.

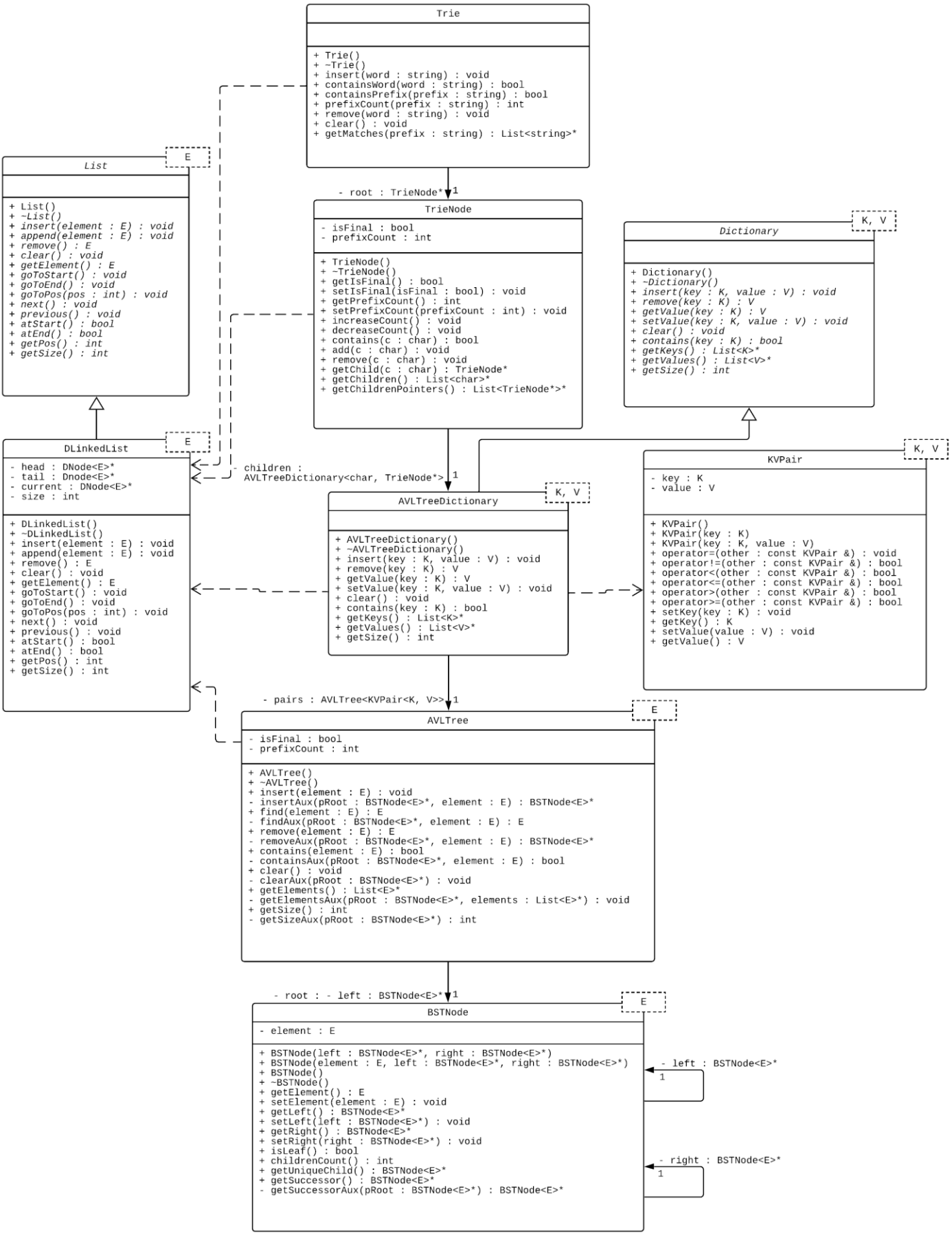
La clase **TrieNode** se encarga de asociar caracteres con punteros a **TrieNode**, esto lo hace utilizando un diccionario. El diccionario a utilizar por parte del **TrieNode** debe ser un **AVLTreeDictionary**, que es una implementación de la clase abstracta **Dictionary** pero utilizando un árbol de búsqueda binaria AVL para almacenar elementos **KVPair**.

El árbol de búsqueda binaria utilizado por el diccionario se implementa en la clase **AVLTree**, que utiliza nodos de tipo **BSTNode** para guardar su información.

Varias clases mencionadas hacen uso de listas para realizar su trabajo o como valores de retorno, por lo que las clases **Trie**, **TrieNode**, **AVLTreeDictionary** y **AVLTree** tienen dependen de las clases **DLinkedList** y la clase abstracta **List**.

La clase **DLinkedList** hace uso de la clase **DNode** para guardar su información. La clase **DNode** se omitió en el diagrama que se encuentra a continuación y que muestra la estructura descrita.

Varias de las clases necesarias para la implementación de este proyecto son las mismas que se implementaron en las clases del curso y en las tareas cortas asignadas anteriormente, por lo que no debería ser necesaria su implementación completa.


Trie
+ Trie()
+ ~Trie()
+ insert(word : string) : void
+ containsWord(word : string) : bool
+ containsPrefix(prefix : string) : bool
+ prefixCount(prefix : string) : int
+ remove(word : string) : void
+ clear() : void
+ getMatches(prefix : string) : List<string>*
- root : TrieNode*
1
List
E
+ List()
+ ~List()
+ insert(element : E) : void
+ append(element : E) : void
+ remove() : E
+ clear() : void
+ getElement() : E
+ goToStart() : void
+ goToEnd() : void
+ goToPos(pos : int) : void
+ next() : void
+ previous() : void
+ atStart() : bool
+ atEnd() : bool
+ getPos() : int
+ getSize() : int
TrieNode
- isFinal : bool
- prefixCount : int
+ TrieNode()
+ ~TrieNode()
+ getIsFinal() : bool
+ setIsFinal(isFinal : bool) : void
+ getPrefixCount() : int
+ setPrefixCount(prefixCount : int) : void
+ increaseCount() : void
+ decreaseCount() : void
+ contains(c : char) : bool
+ add(c : char) : void
+ remove(c : char) : void
+ getChild(c : char) : TrieNode*
+ getChildren() : List<char>*
+ getChildrenPointers() : List<TrieNode*>*
Dictionary
K, V
+ Dictionary()
+ ~Dictionary()
+ insert(key : K, value : V) : void
+ remove(key : K) : V
+ getValue(key : K) : V
+ setValue(key : K, value : V) : void
+ clear() : void
+ contains(key : K) : bool
+ getKeys() : List<K>*
+ getValues() : List<V>*
+ getSize() : int
DLinkedList
E
- head : DNode<E>*
- tail : Dnode<E>*
- current : DNode<E>*
- size : int
+ DLinkedList()
+ ~DLinkedList()
+ insert(element : E) : void
+ append(element : E) : void
+ remove() : E
+ clear() : void
+ getElement() : E
+ goToStart() : void
+ goToEnd() : void
+ goToPos(pos : int) : void
+ next() : void
+ previous() : void
+ atStart() : bool
+ atEnd() : bool
+ getPos() : int
+ getSize() : int
- children :
AVLTreeDictionary<char, TrieNode*>
1
AVLTreeDictionary
K, V
+ AVLTreeDictionary()
+ ~AVLTreeDictionary()
+ insert(key : K, value : V) : void
+ remove(key : K) : V
+ getValue(key : K) : V
+ setValue(key : K, value : V) : void
+ clear() : void
+ contains(key : K) : bool
+ getKeys() : List<K>*
+ getValues() : List<V>*
+ getSize() : int
KVPair
K, V
- 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
- pairs : AVLTree<KVPair<K, V>>
1
AVLTree
E
- isFinal : bool
- prefixCount : int
+ AVLTree()
+ ~AVLTree()
+ insert(element : E) : void
- insertAux(pRoot : BSTNode<E>*, element : E) : BSTNode<E>*
+ find(element : E) : E
- findAux(pRoot : BSTNode<E>*, element : E) : E
+ remove(element : E) : E
- removeAux(pRoot : BSTNode<E>*, element : E) : BSTNode<E>*
+ contains(element : E) : bool
- containsAux(pRoot : BSTNode<E>*, element : E) : bool
+ clear() : void
- clearAux(pRoot : BSTNode<E>*) : void
+ getElements() : List<E>*
- getElementsAux(pRoot : BSTNode<E>*, elements : List<E>*) : void
+ getSize() : int
- getSizeAux(pRoot : BSTNode<E>*) : int
- root : - left : BSTNode<E>*
1
BSTNode
E
- element : E
+ BSTNode(left : BSTNode<E>*, right : BSTNode<E>*)
+ BSTNode(element : E, left : BSTNode<E>*, right : BSTNode<E>*)
+ BSTNode()
+ ~BSTNode()
+ getElement() : E
+ setElement(element : E) : void
+ getLeft() : BSTNode<E>*
+ setLeft(left : BSTNode<E>*) : void
+ getRight() : BSTNode<E>*
+ setRight(right : BSTNode<E>*) : void
+ isLeaf() : bool
+ childrenCount() : int
+ getUniqueChild() : BSTNode<E>*
+ getSuccessor() : BSTNode<E>*
- getSuccessorAux(pRoot : BSTNode<E>*) : BSTNode<E>*
- left : BSTNode<E>*
1
- right : BSTNode<E>*
1

Trie

Esta es la clase principal que se utiliza en el proyecto. La interfaz de usuario debe instanciar al menos un objeto de este tipo para lograr su trabajo. A continuación, se describe brevemente el objetivo de cada uno de los métodos de la clase.

- **insert(word : string) : void**
  Inserta en el árbol el *string* enviado por parámetro. Se encarga de crear los nodos necesarios y de ajustar las cuentas de prefijos de cada nodo para reflejar la inserción de la palabra. El *string* no puede ser vacío.
- **containsWord(word : string) : bool**
  Retorna verdadero si el *string* enviado por parámetro se encuentra dentro de la estructura. El nodo en el que termina la palabra debe estar marcado como final de palabra. Si no se encuentra, retorna falso. El *string* enviado no puede ser vacío.
- **containsPrefix(prefix : string) : bool**
  Retorna verdadero si el *string* enviado por parámetro se encuentra dentro de la estructura. El nodo en el que termina el prefijo no debe estar marcado como final de palabra. Si no se encuentra, retorna falso. El *string* enviado no puede ser vacío.
- **prefixCount(prefix : string) : int**
  Retorna un entero con la cantidad de palabras que coinciden con el *string* enviado por parámetro.
- **remove(word : string) : void**
  Elimina del árbol la palabra enviada por parámetro. Se eliminan los nodos necesarios y se ajustan las cuentas de prefijos de cada nodo para reflejar el borrado de la palabra. El *string* no puede ser vacío. El *string* enviado debe existir como palabra dentro de la estructura.
- **clear() : void**
  Limpia por completo los contenidos de la estructura.
- **getMatches(prefix : string) : List<string>***
  Retorna una lista con todas las palabras que coinciden con el prefijo enviado por parámetro. La lista retornada debe estar en memoria dinámica y el código que invoca este método es responsable de eliminar de memoria el objeto retornado. Si no hay coincidencias, la lista retornada es vacía.

TrieNode

Se utiliza para representar la información dentro del **Trie**. Su responsabilidad es asociar caracteres con punteros a nodos de tipo **TrieNode**. Contiene un atributo booleano **isFinal** que indica si el nodo representa el final de una palabra. También cuenta con un contador de prefijos que indica cuántas palabras tienen como prefijo el representado por el nodo actual. El atributo **children** consiste en un diccionario **BSTreeDictionary**, que es donde se asocia cada carácter con un puntero a otro nodo.

Para efectos del proyecto, el nodo del árbol Trie debería contener un atributo adicional que sea una lista de números enteros, que indican los números de línea donde una palabra ha sido encontrada en el archivo analizado.

- **getIsFinal() : bool**
  Retorna el valor del atributo isFinal. Indica si el nodo actual representa el final de una palabra.
- **setIsFinal(isFinal : bool) : void**
  Asigna un valor valor al atributo isFinal.
- **getPrefixCount() : int**
  Retorna el valor del atributo prefixCount. Indica cuántas palabras contienen el prefijo representado por el nodo actual.
- **setPrefixCount(prefixCount : int) : void**

Asigna un valor al atributo prefixCount.

- **increaseCount() : void**
  Aumenta en uno el valor del atributo prefixCount. Se utiliza durante la inserción en el Trie.
- **decreaseCount() : void**
  Decrementa en uno el valor del atributo prefixCount. Se utiliza durante el borrado en el Trie.
- **contains(c : char) : bool**
  Retorna verdadero si el nodo actual contiene un hijo identificado con el carácter enviado por parámetro.
- **add(c : char) : void**
  Agrega un nuevo hijo al diccionario del nodo actual, usando como llave el carácter enviado por parámetro y como valor un nodo nuevo con sus atributos isFinal en falso y prefixCount en cero.
- **remove(c : char) : void**
  Elimina del diccionario el elemento con el carácter enviado por parámetro.
- **getChild(c : char) : TrieNode***
  Retorna un puntero al nodo asociado con el carácter enviado por parámetro. Si no se encuentra el carácter en el diccionario del nodo actual, el diccionario generaría una excepción.
- **getChildren() : List<char>***
  Retorna un puntero a una lista con todos los caracteres hijos del nodo actual. Si no tiene hijos, retorna una lista vacía. La lista debe estar creada en memoria dinámica.
- **getChildrenPointers() : List<TrieNode*>***
  Retorna un puntero a una lista con todos punteros a nodos hijos del nodo actual. Si no tiene hijos, retorna una lista vacía. La lista debe estar creada en memoria dinámica.

AVLTreeDictionary

Esta clase se encarga de implementar la clase abstracta **Dictionary** por medio del uso de un árbol de búsqueda binaria AVL balanceado. La clase debe heredar de **Dictionary** y debe tener como atributo un objeto de tipo **AVLTree**. También debe utilizar la clase **KVPair** como elemento dentro del árbol de búsqueda para almacenar pares llave valor.

Los métodos por implementar en esta clase son los mismos que se han implementado con anterioridad en clase en los casos de **UnsortedArrayDictionary**, **SortedArrayDictionary** o **BSTDictionary** por lo que no se explicarán en este enunciado.

MaxHeap

Esta clase se utilizará en el proyecto como medio para producir la lista de las palabras más utilizadas en el archivo. El **MaxHeap** puede implementarse a partir del **MinHeap**. Lo único que debe hacerse es modificar los diferentes métodos para que los elementos con mayor prioridad son los que tengan mayor valor, en lugar de los que tienen un valor menor. Se recomienda usar el **MaxHeap** en conjunto con la clase **KVPair** para guardar como llave la cantidad de apariciones de una palabra y como valor el *string* con la palabra. El método **removeTop(n)** sería el que nos produciría la lista de palabras más utilizadas.

Otras clases

Las clases **List**, **DLinkedList**, **Dictionary**, **KVPair**, **AVLTree** y **BSTNode** son las mismas que se han cubierto durante el curso y que se han implementado en las diferentes tareas cortas, por lo que tampoco se explicarán en este enunciado. Puede hacerse uso de cualquier clase adicional que se haya construido en el curso. No se permite el uso de estructuras de datos de bibliotecas de C++ o de algún tercero. Deben utilizarse única y exclusivamente las estructuras hechas durante el semestre.

Cambios necesarios

Para llevar a cabo el trabajo solicitado, es necesario realizar distintas modificaciones a las estructuras descritas. Para llevar control de las líneas en donde aparece una palabra, es importante modificar el **TrieNode** para que también contenga una lista que le permita accederlas, ya sea de punteros o índices a otra estructura. No se recomienda almacenar el *string* con la línea en cada nodo porque se requeriría demasiada memoria.

También, se solicita imprimir las líneas donde aparece una palabra, por lo que es necesario almacenar las líneas originales del archivo en alguna estructura lineal de acceso rápido para evitar abrir el archivo nuevamente y así mostrarlas lo más eficientemente posible.

A continuación, se describe con mayor detalle lo que debe realizar el programa.

Programa principal

El programa principal puede ser un programa de consola en modo texto.

La ejecución del programa debe cumplir con las siguientes consideraciones:

1. Al iniciar el programa se imprime un mensaje de bienvenida corto que explica el propósito del programa.
2. Se solicita al usuario el nombre de un archivo de texto a analizar. No debe agregarse ningún carácter o extensión al nombre de archivo proporcionado por el usuario. No modifique el *string* escrito por el usuario, la entrada que escribe es la que debe usarse para abrir el archivo. El programa tiene que ser capaz de abrir cualquier extensión (".py", ".cpp", ".h", ".java", por ejemplo).
3. Si el archivo no existe o no es posible abrirlo, se indica un mensaje de error y el programa termina.
4. Si no, se abre el archivo y es procesado línea por línea. La línea original leída debe almacenarse en una estructura de rápido acceso que permita localizar el texto de la línea en caso de que se requiera imprimirla.
5. Cada línea se separa en palabras. Las mayúsculas deben sustituirse por minúscula. Los espacios en blanco y signos de puntuación que separan palabras no deben ser considerados parte de las palabras. Las palabras deben contener sólo letras utilizadas en el idioma español (considere acentos, diéresis y otras). Tome en cuenta que el espacio en blanco no es el único separador de palabras. El string "hombre-lobo", por ejemplo, debe separarse en dos palabras diferentes.
6. Cada palabra leída en el archivo se inserta en el árbol Trie, junto con el número de línea donde apareció o alguna otra referencia a la línea. Como cada palabra puede aparecer en diferentes líneas, cada palabra del Trie debe tener una lista de las líneas en que ha aparecido.
7. Una vez cargados los datos en el árbol, se presenta al usuario las siguientes opciones:
    a. Consulta por prefijo
    Se solicita al usuario un prefijo para buscar en el árbol. Por cada palabra que empiece con dicho prefijo, se muestra la palabra, la cantidad de veces que aparece en el archivo original y el número de línea y la línea de texto completa, tal y como aparece en el archivo original.
    b. Buscar palabra
    Se solicita al usuario una palabra para buscar en el árbol. Si la palabra se encuentra, y por cada una de las veces que aparece, se muestra en pantalla el número de línea y la línea de texto completa, tal y como aparece en el archivo original.
    c. Buscar por cantidad de letras
    Se solicita al usuario una cantidad entera positiva y se utiliza ese valor para buscar en el árbol todas las palabras que tienen esa cantidad de letras. Se muestra la cantidad de veces que se utiliza cada palabra. La lista debe aparecer en orden alfabético. Esta operación debe hacerse mediante un método similar al **getMatches**, utilizando recursión y extrayendo sólo las palabras que contienen la cantidad de caracteres indicada. Se recomienda detener la recursión sobre el árbol cuando ya se alcanzó la cantidad de niveles correspondiente a la cantidad de letras buscada.

Evite obtener la lista completa de palabras y luego filtrarla por tamaño, ya que esto sería excesivamente lento.

d. Ver top de palabras más utilizadas
   Esta opción permite al usuario ver un *top* de las palabras más utilizadas en el archivo analizado. También debe mantenerse una lista de palabras a ignorar, esta lista de palabras se lee de un archivo de texto que lleva por nombre "ignorar.txt", con una palabra por línea.
   Primero se le solicita al usuario un número entero N que indica la cantidad de elementos que se incluirán en el resultado. Se muestra en pantalla el *top* N de palabras más utilizadas, cada palabra y la cantidad de veces que se utiliza en el documento. En el *top* no aparece ninguna de las palabras de la lista a ignorar. Esta operación debe implementarse por medio de un **MaxHeap**, insertando en la estructura pares llave-valor que tenga como llave la cantidad de apariciones de la palabra y como valor asociado la palabra misma. Luego se eliminan de la estructura y se imprimen las primeras N palabras del *heap*.

e. Cargar otro archivo
   Se libera toda la memoria utilizada hasta ahora para el procesamiento del archivo, se limpian todas las estadísticas que se hayan solicitado y se solicita al usuario otro archivo para procesar. Iniciando nuevamente desde el punto 2. El programa debe soportar hacer esta operación repetidas veces con archivos grandes, de forma que se demuestre el manejo correcto de memoria dinámica y la ausencia de *memory leaks*.

8. El usuario puede seguir haciendo consultas a la información procesada hasta que seleccione cargar otro archivo o hasta que elija una opción para salir del programa.

¡Atención!

El programa debe funcionar eficientemente con archivos de gran tamaño. Junto con el enunciado se proveen archivos de texto de tamaño considerable para hacer pruebas. El procesamiento del archivo y las diferentes consultas no deben pasar de unos cuantos segundos. La eficiencia del programa tendrá peso en la nota del proyecto.

Se recomienda utilizar archivos de texto con codificación ANSI, ya que es la codificación más amigable con el tipo char de C++. Convierta manualmente los archivos a este formato antes de utilizarlos en su programa (esto se puede hacer en casi cualquier editor de texto).

Documentación

La documentación interna del código debe ser mínima, puede limitarse únicamente a una descripción breve del objetivo de cada clase y métodos dentro del proyecto. Evite hacer comentarios excesivos. Escriba código claro y conciso, trate de apegarse a los principios de código limpio para el código que escriba. Recuerde que es muy importante que su código sea claro y fácil de entender.

En cuanto a la documentación externa, debe entregarse un documento en formato PDF:

1. Portada
2. Introducción. ¿Por qué se hace el proyecto y qué partes incluye este? (1-2 páginas.)
3. Presentación y análisis del problema.
   a. Descripción del problema. ¿Qué es lo que hay que resolver? Identificar subproblemas que deben resolverse en el proyecto y explicar cada uno de ellos. Por favor, no copiar y pegar partes de este enunciado (1 o 2 páginas).
   b. Metodología. ¿Cómo resolvió el problema? Explicar la forma en que se resolvieron los diferentes subproblemas. Describir de forma técnica y detallada. Utilice diagramas para representar la estructura del proyecto (1 o 2 páginas).

    c. Análisis crítico. ¿Qué se logró implementar? ¿Qué faltó? ¿Qué cosas se podrían mejorar de lo que se implementó? Desarrollar cada uno de estos aspectos (1 o 2 páginas).
4. Conclusiones: resoluciones puntuales tras el proyecto. Estas deben ser relacionadas con los aspectos técnicos del trabajo únicamente. Mínimo 10 conclusiones. (1-2 págs.)
5. Recomendaciones: consejos o advertencias que se derivan de las conclusiones. Lecciones aprendidas durante el desarrollo del proyecto. Recomendaciones para personas que tengan que hacer el mismo trabajo. También deben estar orientadas con aspectos técnicos de la tarea programada. Hacer una o más recomendaciones por cada conclusión. Mínimo 10 recomendaciones. (1-2 págs.)
6. Referencias. Deben incluirse en formato APA.

Forma de trabajo

El proyecto se desarrollará en grupos de 3 o 4 integrantes. No se permitirá la entrega de proyectos desarrollados de forma individual a menos que exista autorización previa del profesor.

No dude en consultar cualquier asunto tanto de programación como de elaboración de la documentación.

Los miembros del grupo deben distribuir el trabajo uniformemente, cualquier situación en la que un miembro no esté realizando su parte del trabajo debe ser notificada al profesor inmediatamente y dicha persona puede ser separada del grupo para realizar el trabajo de forma individual.

Evaluación

La tarea tiene un valor de 20% de la nota final, en el rubro de Proyectos Programados.

Desglose de la evaluación de la tarea programada:

Documentación:	20%
Programación:	80%

Recomendaciones adicionales

Pruebe cada funcionalidad individualmente. No implemente grandes secciones del programa sin verificar el funcionamiento por separado de cada una de sus partes. Esto dirige a errores que son más difíciles de encontrar.

Recuerde que el trabajo es en equipos, es indispensable la comunicación y la coordinación entre los miembros del subgrupo.

Comparta el conocimiento con los demás compañeros de grupo y de la carrera, la ciencia de la computación es una disciplina que requiere el traspaso libre de conocimientos. Se logran mejores resultados con la colaboración de todos que con el esfuerzo separado de diferentes personas.

No dude en consultar diferentes fuentes para satisfacer las dudas. Aparte de las búsquedas en internet, asegúrese de exponer sus dudas a sus compañeros, profesor y conocidos que estudien la carrera; en la mayoría de las ocasiones es más provechosa conversación de 10 minutos entre personas que están trabajando en lo mismo que pasar horas buscando la respuesta a una duda de forma individual.

No deje la documentación para el final, es buena práctica ir desarrollándola durante todo el transcurso del proyecto. Recuerde que la documentación debe ser concisa y puntual, por lo que en realidad no toma mucho tiempo al realizarla de esta forma.

Plagios no serán tolerados en ninguna circunstancia. Cualquier intento de fraude será evaluado con una nota de cero y se enviará una carta al expediente del estudiante. Siempre escriba su propio código.

Referencias

Wikipedia contributors. (2018, August 16). Trie. In *Wikipedia, The Free Encyclopedia*. Retrieved 01:55, October 3, 2018, from https://en.wikipedia.org/w/index.php?title=Trie&oldid=855171582