Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Bachillerato en Ingeniería en Computación

IC-2001 Estructuras de Datos

Profesor: Mauricio Avilés Cisneros

Proyecto I

Indización de texto con Tries

Integrantes

Sara Castro Sáenz 2014085332

Ericka Céspedes Moya 2017239557

María Fernanda Niño Ramírez 2018236104

Fecha de Entrega: 28 de octubre

II Semestre, 2018

Introducción

Este proyecto programado consiste en la implementación de un árbol trie con palabras obtenidas de un archivo de texto para su análisis. El objetivo este es la búsqueda de palabras que coincidan con otra palabra o un prefijo. Luego, una vez encontradas las palabras, localizar las líneas de texto en donde se encuentre esta misma palabra. Por último, buscar palabras en el texto con una cierta cantidad de letras. Un trie es una estructura de datos de tipo árbol que permite la recuperación de información. En el árbol se almacenan llaves con una secuencia de caracteres; cada letra del alfabeto está en un nodo interno del árbol. De esta manera, tenemos un nodo raíz con la primera letra y sus nodos hijos representan las diferentes posibilidades con letras distintas para conformar una palabra.

La importancia de un proyecto como el desarrollado o de la estructura jerárquica trie recae en que actualmente se utilizan en funcionalidades cotidianas del software. El árbol trie consiste de llaves de tipo string y de valores de cualquier tipo y se utiliza para buscar llaves rápidamente o encontrar llaves que tengan la misma secuencia de caracteres. Las ventajas de usar una estructura trie consisten en hacer una búsqueda de llaves rápida, requerir menos espacio para almacenar grandes cantidades de pequeñas cadenas, y aprovechar un mejor funcionamiento para buscar prefijos largos. Este tipo de estructura se puede implementar en el texto predictivo en teléfonos celulares, sugerencias en buscadores de internet, sugerencias de comandos y revisiones de ortografía aprovechando la capacidad de los tries de hacer búsquedas, inserciones y borrados rápidos.

La estructuración lógica del proyecto se basa en la construcción de una clase Trie. La clase Trie se encarga de hacer las operaciones principales con strings y utiliza nodos tipo TrieNode. Las clases Trie, TrieNode, BSTreeDictionary y BSTree tienen una dependencia hacia DLinkedList. La clase DLinkedList hereda de la clase abstracta List, así como BSTreeDictionary de la clase abstracta Dictionary. La clase BSTree utiliza nodos de tipo BSTNode y la clase DLinkedList de nodos tipo DNode. La clase Trie

almacenará elementos en nodos de tipo TrieNode. Cada nodo funciona para recorrer el árbol y guiar la búsqueda. La clase BSTreeDictionary almacena elementos KVPair y es utilizada por la clase TrieNode para guardar un caracter y un puntero al siguiente nodo. Esta clase hace uso de la clase abstracta Dictionary que almacena pares llave-valor, para esto se necesita la clase KVPair que se encarga de agrupar los elementos: guarda un elemento llave y un elemento valor. La clase DLinkedList es utilizada a lo largo del proyecto en las clases BSTreeDictionary, BSTree, Trie y TrieNode. Las ventajas que aporta una lista enlazada en este tipo de proyecto es que es independiente de la memoria, no es necesario mover elementos para hacer inserciones, y que el nodo en donde se encuentra guardado el elemento tiene un puntero al siguiente elemento y el anterior. De esta manera, no hay un límite para la cantidad de elementos que pueden ser insertados en las clases BSTreeDictionary y Trie, y se aprovecha la estructura trie para almacenar grandes cantidades de pequeñas cadenas. Además, se optimiza la velocidad de inserción, búsqueda y borrado. La clase DLinkedList también fue utilizada en la clase Text, que fue creada para abrir un archivo de texto y guardar sus palabras en una lista enlazada. La lista enlazada está conformada por elementos KVPair que almacenan la palabra y la línea en la que se encuentra la palabra. Adicionalmente, la clase Text también hace uso de un BSTreeDictionary para guardar las líneas del archivo de texto por línea y número de línea.

Presentación y análisis del problema

I. ¿Qué hay que resolver?

La primera parte del proyecto consiste en la revisión del correcto funcionamiento de las clases BSTreeDictionary, BSTree, Trie y TrieNode elaboradas en clase. Luego de esto se realizan modificaciones a dichas clases según sea necesario para abrir y procesar un archivo de texto, e implementar las funcionalidades necesarias en las clases según sean requeridas en el enunciado del proyecto, estas consisten en:

- Abrir y adquirir los datos de un documento de texto.
- Validar que el archivo se haya podido abrir. De lo contrario, mostrar un mensaje de error y finalizar el programa.
- Obtener el archivo de texto por líneas.
- Separar cada línea en palabras.
- Almacenar cada línea de texto sin modificar para que sean de fácil acceso.
- Obtener el número de línea de cada palabra y asignarlo al final de cada palabra.
- Eliminar caracteres innecesarios que no son letras ni números, como los espacios y puntos antes de ingresar la palabra al trie.
- Consultar palabras del texto según su prefijo.
- Obtener la cantidad de veces que aparece una palabra en el texto.
- Buscar una palabra según su tamaño o la cantidad de letras que contiene.
- En la opción para buscar palabras por tamaño, verificar que el dato ingresado sea un número.

Por último, una vez que todas los puntos mencionados anteriormente funcionen correctamente, se podrá crear el programa de consola para que el usuario lo pueda utilizar. Si se implementa correctamente, el programa de consola será llevado a cabo rápidamente, sólo llamará a métodos de las clase Trie y no necesitará tratar directamente con el archivo de texto o el contenido de este.

II. ¿Cómo se resolvió?

Para procesar el archivo de texto se creó una clase llamada Text que lee el archivo y guarda el número de línea y la línea en un *BSTreeDictionary*. La clase tiene dos métodos públicos, uno con el que se obtiene una línea del texto en específico retornando el valor segun la llave que se le especifique y otro que retorna una lista creada en memoria dinámica con todas las palabras que hay en el documento de texto, para ello se procesa cada línea que hay en el diccionario y se separa en palabras usando los espacios para delimitar. En la clase Text se utiliza un método privado que es llamado en el constructor de la clase que intenta abrir el archivo y si falla tira una excepción, si no falla lee cada línea de texto usando <*fstream*> y la agrega al diccionario junto con el número de línea, para saber el número de línea, se utiliza un contador dentro del método. También, utiliza otro método que quita cualquier carácter no deseado de una tira de texto, usa *remove_if(ForwardIt first, ForwardIt last, UnaryPredicate p)* del paquete <*algorithm*> que itera sobre la tira de caracteres y los remueve si cumplen una condición, la condición en este caso es que sea signo de puntuación y usamos una expresión lambda que retorna true si lo es, *std::string* ya tiene un método para saber si eso pero da problemas cuando se le envía un carácter español por parámetro por lo que se tuvo que solucionar con la expresión lambda. Este método solo es llamado en el método Text::*getWords()*.

```
class Text {
private:
    BSTDictionary<int, std::string> *lines;
    void parseFile (std::string fileName);
    std::string cleanWord (std::string word);

public:
    Text (std::string fileName);
    ~Text ();
    std::string getLine (int line);
    List< KVPair<int, std::string> >* getWords ();
};
```

Figura 1. La clase Text

Las clases Trie y TrieNode implementadas en clases fueron modificadas para poder cumplir con las especificaciones del proyecto. En la clase TrieNode se agregó una lista de números para poder almacenar el número de línea en el que se encuentra una palabra y se agregaron métodos para poder modificar la lista. En la clase Trie se modificó el método *insert()* para poder insertar un par llave/valor en vez de una sola tira de caracteres con la idea de poder pasar el número de línea en el que se encuentra la palabra y así insertarlo en el arreglo de números que se agregó en el *TrieNode*, también se se eliminó la excepción que tiraba si la palabra ya estaba en el trie, ahora solo marca el nodo como palabra si esta no se encuentra en el Trie, e inserta el número de línea al final de la palabra para ambos casos. Se agregó la posibilidad de obtener una lista de palabras con n cantidad de letras por medio de un método recursivo y un auxiliar, el método funciona parecido al método *getMatches()* pero este sólo agrega la palabra a la lista si esta es del tamaño que se paso por parámetro. Los métodos *getMatches()* y *getMatchesAux()* fueron modificados para que retornen una lista de pares <string, List<int>*> y así tener acceso tanto a la palabra como la lista que contiene los números de línea, con esto podemos saber cuántas veces se repetía una palabra en el texto utilizando el método *getSize()* de la lista de números y saber en qué líneas se usó la palabra.

```
List<std::string>* Trie::getSizeMatches (int size) {
    List<std::string> *words = new DLinkedList<std::string> ();
    getSizeMatchesAux (root, "", size, words);
    return words;
}

void Trie::getSizeMatchesAux (TrieNode * current, std::string prefix, int size, List<std::string>* words) {
    if (prefix.size () == (unsigned) size && current->getIsWord ())
        words->append (prefix);

    List<char> *children = current->getChildren ();
    for (children->goToStart (); !children->atEnd (); children->next ()) {
        char c = children->getElement ();
        std::string newPrefix = prefix;
        newPrefix.append (1, c);
        getSizeMatchesAux (current->getChild (c), newPrefix, size, words);
    }
    delete children;
}
```

Figura 2. Método para obtener las palabras por tamaño

En el main se implementaron todos los métodos necesarios para realizar lo solicitado en la especificación. Se implementó un método que imprime todas las palabras que tienen el prefijo que ingresó el usuario. Se obtiene la lista de pares llave/valor con el método Trie*::getMatches(std::string prefix)* y se itera sobre ella para imprimir la llave junto con el tamaño de la lista que está en el valor. El método para imprimir las palabras que tienen n cantidad de letras usa una lista que genera Trie*::getSizeMatches(int size)* e imprime sus contenidos. Para imprimir las líneas del texto en las que se usa la palabra especificada por el usuario, se usa Trie*::getMatches(string prefix)* e itera sobre la lista que se obtiene hasta que encuentra la palabra, luego itera sobre la lista de los números de línea para imprimir las líneas donde está la palabra usando Text*::getLine(int line)*.

```
void printMatchesbyPrefix (std::string prefix) {
    List< KVPair<std::string, List<int>*> > *matches = trie.getMatches (prefix);
    for (matches->goToStart (); !matches->atEnd (); matches->next ())
        std::cout << matches->getElement ().getKey () << "\t" << matches->getElement ().getValue ()->getSize () << "\n";
    std::cout << std::endl;
    delete matches;
}

void printMatchesBySize (int size) {
    List<std::string> *list = trie.getSizeMatches (size);
    for (list->goToStart (); !list->atEnd (); list->next ())
        std::cout << list->getElement () << std::endl;
    std::cout << std::endl;
    delete list;
}

void printWord (std::string word) {
    List< KVPair<std::string, List<int>*> > *matches = trie.getMatches (word);
    List<int> *lineNumbers;
    for (matches->goToStart (); !matches->atEnd (); matches->next ()) {
        if (matches->getElement ().getKey () == word) {
            lineNumbers = matches->getElement ().getValue ();
            for (lineNumbers->goToStart (); !lineNumbers->atEnd (); lineNumbers->next ())
                std::cout << lineNumbers->getElement () << "\t" << text->getLine (lineNumbers->getElement ()) << "\n\n";
            std::cout << std::endl;
        }
    }
    delete matches, lineNumbers;
}
```

Figura 3. Métodos para imprimir lo solicitado por el usuario

En el main también se implementó un método que agrega .txt al final del nombre del documento que ingresó el usuario si este no tiene la extensión para evitar errores a la hora de abrir el archivo. Se utilizaron otros métodos para poder solucionar el

problema de ingresar caracteres españoles en la consola e imprimirlos, *SetConsoleCP (1252)* y *setlocale (LC_ALL, "spanish")* específicamente, 1252 es la codificación de los caracteres latinos, la consola de Windows usa 850 por defecto y esta no tiene los caracteres españoles. La implementación del menú se hizo usando *do* y *switch* para poder verificar si los datos ingresados por el usuario eran los correctos y para navegar por las opciones del menú. Para verificar que el dato sea un numero, se utilizo el metodo en std::*cin.fail()* para verificar que el dato fuese correcto, en caso contrario se muestra un mensaje de error y se devuelve al menú principal.

III. Análisis crítico de la implementación

Finalmente, todas las funcionalidades requeridas en el enunciado del proyecto pudieron ser llevadas a cabo. La aplicación de consola puede consultar palabras por prefijo y mostrar cuántas veces aparecen en el archivo de texto. También puede buscar palabras en el archivo de texto, mostrar cuántas veces aparece, la línea en la cual aparece y mostrar en pantalla la línea completa. Por último, se pueden buscar palabras según la cantidad de letras ingresada. Satisfactoriamente se puede abrir un archivo de texto, analizar su contenido y mostrar un mensaje de error si este no puede ser abierto. Además de que se pueden leer y escribir caracteres especiales del idioma español en la aplicación.

El método para buscar una palabra e imprimir las líneas que la contienen podría mejorarse implementando un método para buscar en el trie que retornara el par llave/valor que contiene la palabra y la lista de los números en vez de utilizar el método *getMatches* y preguntar si el elemento de la lista es igual a la palabra. Al utilizar *getMatches* se está utilizando más memoria de la necesaria y realizando operaciones extra que podrían simplificarse con un método de búsqueda de palabras.

```
void printWord (std::string word) {
    List< KVPair<std::string, List<int>*> > *matches = trie.getMatches (word);
    List<int> *lineNumbers;
    for (matches->goToStart (); !matches->atEnd (); matches->next ()) {
        if (matches->getElement ().getKey () == word) {
            lineNumbers = matches->getElement ().getValue ();
            for (lineNumbers->goToStart (); !lineNumbers->atEnd (); lineNumbers->next ())
                std::cout << lineNumbers->getElement () << "\t" << text->getLine (lineNumbers->getElement ()) << "\n\n";
            std::cout << std::endl;
        }
    }
    delete matches, lineNumbers;
}
```

Figura 4. Método actual para imprimir las líneas en donde se usa la palabra

```
void printWord (std::string word) {
    List<int> *lineNumbers = trie.find (word).getValue();
    for (lineNumbers->goToStart (); !lineNumbers->atEnd (); lineNumbers->next ())
        std::cout << lineNumbers->getElement () << "\t" << text->getLine (lineNumbers->getElement ()) << "\n\n";
    std::cout << std::endl;
    delete lineNumbers;
}
```

Figura 5. Cómo podría mejorarse el método

El método Text::*cleanWord(string word)* debería llamarse Text::*cleanLine(string line)* y en Text::*getWords()* solo se debería ser llamado por cada línea de texto y no por cada palabra, se obtiene el mismo resultado y se reduce el *overhead*.

Text*::parseFile(string fileName)* debería dividirse en dos métodos ya que está realizando más de una tarea, está abriendo el archivo de texto y comprobando que este está abierto y además está iterando sobre cada línea del texto y agregandola al diccionario si no es una línea vacía. Los métodos deberían ser *openFile(string fileName)* que abre el archivo y comprueba si esta abierto, si no tira una excepcion y otro metodo *parseText(std::fstream stream)* que recibe el stream y opera sobre este para guardar cada línea en el diccionario.

Conclusiones

- GitLab y Visual Live Share son aplicaciones muy útiles para el desarrollo de un trabajo grupal de programación.
- La creación de una clase Text para procesar un archivo de texto es conveniente si se necesita abrir varios archivos de texto en un programa o si es necesario procesar el contenido del texto. En este caso se utilizó para obtener las líneas y las palabras del texto.
- El uso de la clase KVPair es práctico para guardar los números de líneas de las mismas líneas o de la línea en la cual aparece cada palabra. En el caso de las líneas se utilizó un diccionario que implementa esta misma clase y para las palabras se hizo uso de una lista doblemente enlazada de elementos KVPair con el número de línea como llave y la palabra como valor. Estos mismos elementos KVPair con el número de línea y la palabra fueron insertados en el árbol trie.
- Se evitó la repetición de código uniendo los métodos *containsWord(string word)* y *containsPrefix(string prefix)*.
- Para leer y escribir tildes y caracteres especiales del español como la ñ en una aplicación de consola C se puede utilizar el paquete *locale* y establecer el idioma a español con *setlocale(LC_ALL, "spanish")*. También, se hace uso de la instrucción *SetConsoleCP(1252)* para poder leer caracteres del idioma español. Sin embargo, en versiones desactualizadas de Windows no soluciona el problema.
- Los métodos *getMatches* y *getMatchesAux* fueron modificados para que retornen una lista de pares <*string, List<int>**> y así tener acceso tanto a la palabra como la lista que contiene los números de línea.
- Los métodos *getSizeMatches* y *getSizeMatchesAux* fueron implementados para obtener las palabras que tienen n cantidad de letras.

Recomendaciones

Uso de GitLab para el desarrollo de trabajos en grupo

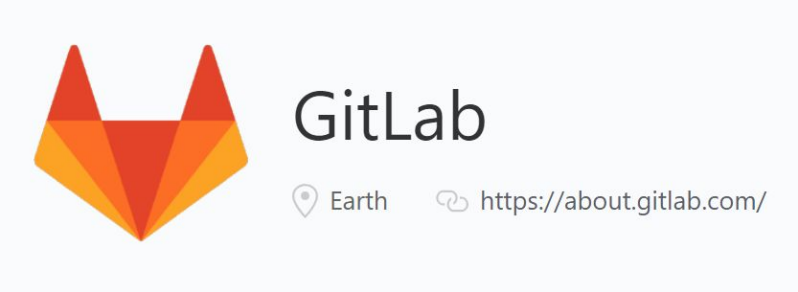

Figura 6. GitLab

Gitlab es un servicio web de control de versiones y desarrollo de software colaborativo basado en Git. Según GitLab (2018), fue escrito por los programadores ucranianos Dmitriy Zaporozhets y Valery Sizov en el lenguaje de programación Ruby.

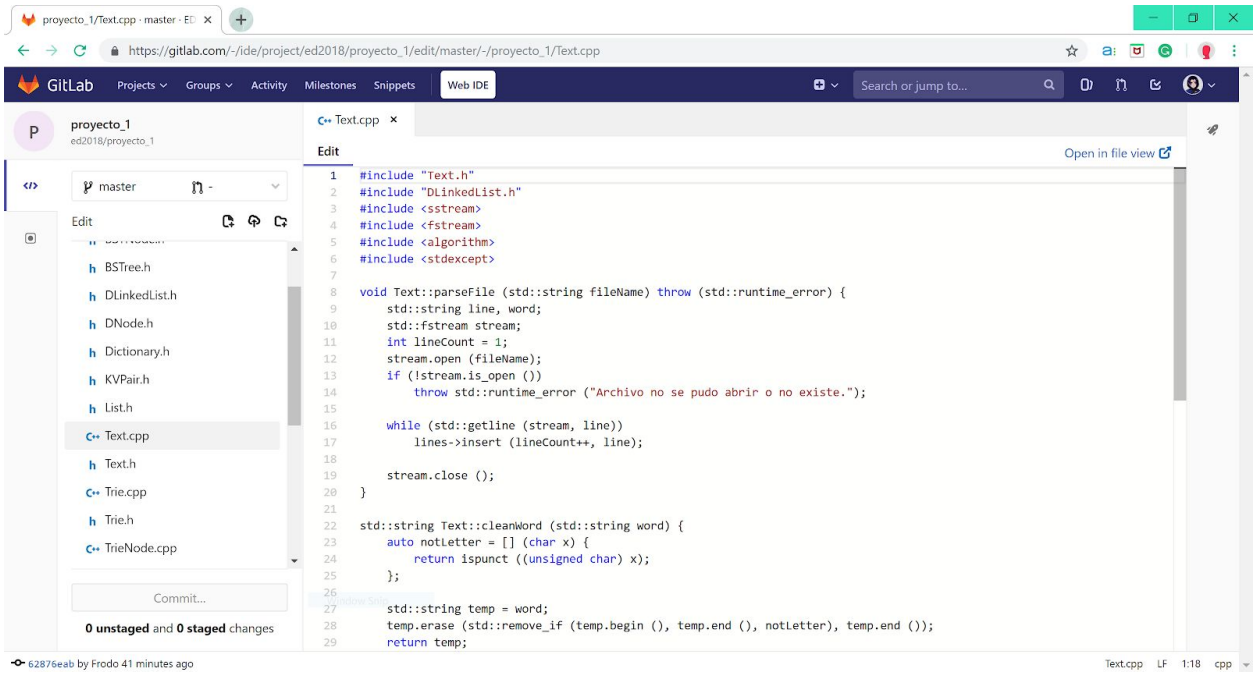

Figura 7. Ejemplo del archivo de la clase Text en GitLab

GitLab ofrece muchas funcionalidades y una de estas es el desarrollo de software colaborativo. Esta página permite subir archivos para ser compartidos con los demás miembros del grupo. Además, se muestran las versiones de cada documento y los cambios hechos cada vez que se guarda una nueva versión del documento.

Uso de Visual Studio Live Share para el desarrollo de trabajos en grupo

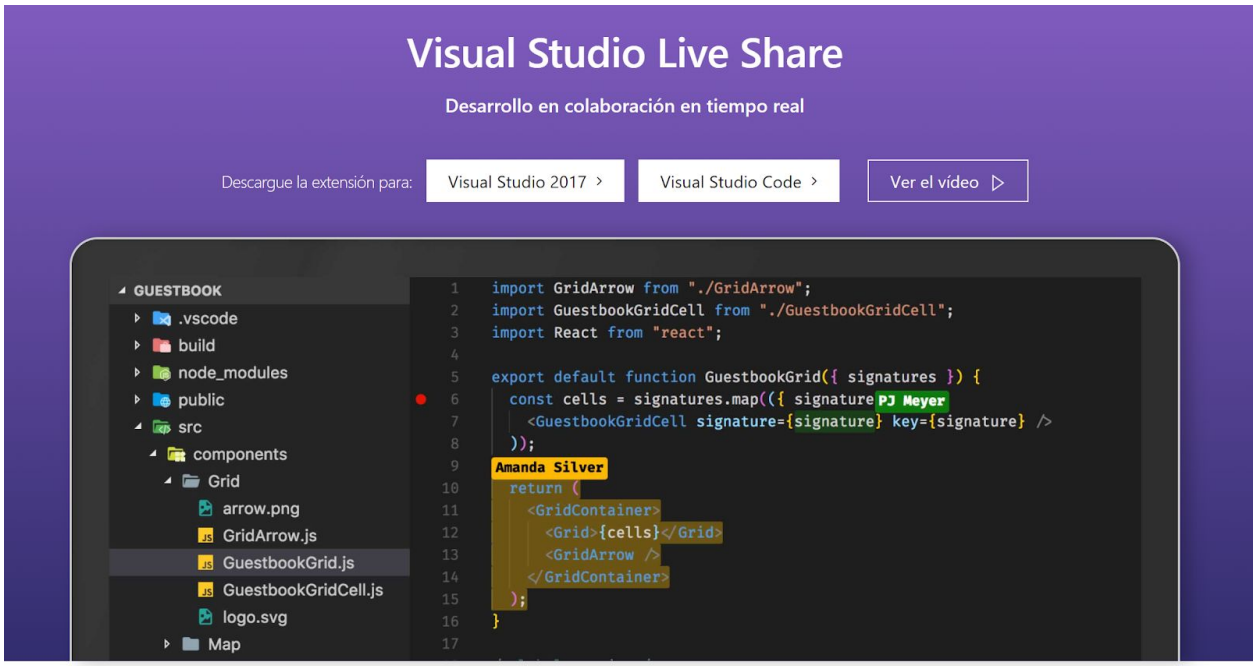

Figura 8. Página web de Visual Studio Live Share

Visual Studio Live Share es una herramienta de desarrollo colaborativo en tiempo real desarrollada por Microsoft. La ventaja de esta aplicación es que permite la colaboración grupal en tiempo real. Las desventajas radican en el peso y la velocidad de la aplicación.

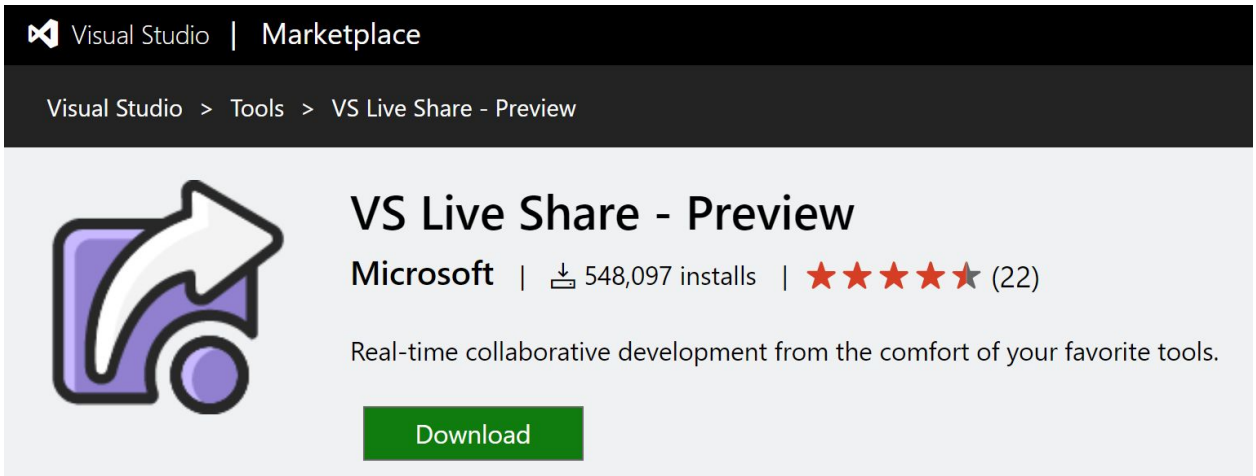

Figura 9. Página de descarga de Virtual Live Share Studio

La aplicación puede ser descargada en el siguiente link https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsls-vs.

Crear una clase Text para procesar el archivo de texto

La clase Text tiene la funcionalidad de abrir y procesar el archivo de texto. Primero se crea una instancia de la clase Text, *text = new Text("nombre del archivo.txt")*. Inmediatamente se crean el atributo de clase *lines*, que se refiere a las líneas del texto, y llama al método *parseFile(string fileName)* para procesar contenido del archivo de texto. Si no se logra abrir el archivo, se muestra un mensaje de error. La ventaja de crear esta clase es obtener las líneas y las palabras de un archivo de texto para ser analizadas en el programa de consola.

```
Text::Text (std::string fileName) {
    lines = new BSTDictionary<int, std::string> ();
    Text::parseFile (fileName);
}

Text::~Text () {
    delete lines;
}

std::string Text::getLine (int line) {
    return lines->getValue (line);
}

List< KVPair<int, std::string> >* Text::getWords () {
    List< KVPair<int, std::string> > *words = new DLinkedList< KVPair<int, std::string> > ();
    List<int>* lines = this->lines->getKeys ();
    for (lines->goToStart (); !lines->atEnd (); lines->next ()) {
        std::stringstream stream (getLine (lines->getElement ()));
        std::string word;
        while (std::getline (stream, word, ' ')) {
            KVPair<int, std::string> p (lines->getElement (), cleanWord (word));
            words->append (p);
        }
    }
    delete lines;
    return words;
}
```

Figura 10. Obtención de las líneas y palabras de un archivo de texto

La clase Text tiene como atributo a lines. Se crearon los métodos *getWords()* para obtener las palabras del documento de texto en una lista y *getLine(int line)* para obtener una línea de texto según el número de esta. El método *getWords()* devuelve una lista doblemente enlazada de palabras al recorrer el diccionario de líneas ya creado en la clase.

```
std::string Text::cleanWord (std::string word) {
    auto notLetter = [] (char x) {
        return ispunct ((unsigned char) x);
    };

    std::string temp = word;
    temp.erase (std::remove_if (temp.begin (), temp.end (), notLetter), temp.end ());
    return temp;
}
```

Figura 11. Obtención de una cadena limpia de caracteres

Se creó un método llamado *cleanWord(string word)* para eliminar los caracteres que no corresponden a una letra del alfabeto o a un número. Este método es utilizado por *getWords()* para insertar las palabras sin caracteres especiales en la lista de palabras.

```
void Text::parseFile (std::string fileName) throw (std::runtime_error) {
    std::string line, word;
    std::fstream stream;
    int lineCount = 1;
    stream.open (fileName);
    if (!stream.is_open ())
        throw std::runtime_error ("Archivo no se pudo abrir o no existe.");

    while (std::getline (stream, line)) {
        if(!line.empty())
            lines->insert (lineCount, line);
        lineCount++;
    }

    stream.close ();
}
```

Figura 12. Abrir y procesar un archivo de texto

El método *parseFile(string fileName)* procesa el archivo de texto. Primero se abre y luego se van insertando las líneas con texto en el atributo correspondiente de la clase. Por último, se cierra el archivo con las líneas del texto ya almacenadas en el diccionario de líneas.

Utilizar la clase KVPair para guardar los números de línea como llaves

En el programa de consola se requiere buscar una palabra, y si esta es encontrada, se debe imprimir cuántas veces aparece, en cuál número de línea aparece y la línea de texto. La clase KVPair se utilizó para guardar cada línea y palabra con su número de línea correspondiente.

```
Text::Text (std::string fileName) {
    lines = new BSTDictionary<int, std::string> ();
    Text::parseFile (fileName);
}
```

Figura 13. Se utiliza un BSTreeDictionary para guardar las líneas

Para las líneas del archivo de texto se hace uso de un BSTreeDictionary que guarda pares de elementos KVPair. Esto facilita el guardado de palabras según el número de línea y la búsqueda de líneas según su número. El uso de un BSTreeDictionary es conveniente por sus métodos *getKeys()* y *getValues()* para obtener tanto el número como la línea.

```
List< KVPair<int, std::string> >* words = text->getWords ();
for (words->goToStart (); !words->atEnd (); words->next ())
    trie.insert (words->getElement ());
```

Figura 14. Se usa una lista de elementos KVPair para guardar las palabras

Una lista doblemente enlazada que contiene elementos KVPair fue utilizada para almacenar las palabras del archivo de texto. Cada elemento KVPair contiene una llave con el número de línea en donde aparece la palabra y un valor con la palabra sin caracteres especiales. De esta manera, se agregaron los mismos elementos KVPair a la estructura de datos trie.

```
void Trie::insert (KVPair<int, std::string> word) {
    TrieNode *current = root;
    for (unsigned int i = 0; i < word.getValue ().size (); i++) {
        current->increaseCount ();
        if (!current->contains (word.getValue ()[i]))
            current->add (word.getValue ()[i]);
        current = current->getChild (word.getValue ()[i]);
    }

    if (!contains (word.getValue ())) {
        current->increaseCount ();
        current->setIsWord (true);
    }

    current->insertLineNumber (word.getKey ());
}
```

Figura 15. La clase Trie guarda strings

Algunos métodos de la clase Trie fueron modificados para pasar como parametro elementos tipo KVPair con los valores del número de línea y la palabra. De esta manera, los métodos *getKey()* y *getValue()* pueden ser aprovechados en la clase Trie para hacer uso tanto del número de línea como de la palabra.

```
List< KVPair<std::string, List<int>*> >* Trie::getMatches (std::string prefix) {
    List< KVPair<std::string, List<int>*> > *words = new DLinkedList< KVPair<std::string, List<int>*> > ();
    getMatchesAux (root, prefix, "", words);
    return words;
}
```

Figura 16. La clase Trie retorna elementos KVPair en el método *getMatches(string prefix)*

Unir los métodos *containsWord(string word)* y *containsPrefix(string prefix)*

Originalmente se tenían los métodos *containsWord(string word)* y *containsPrefix(string prefix)* en la clase Trie que retornaba un valor booleano según se encontraba o no una palabra en el árbol. Se puede simplificar el código uniendo ambos métodos en uno solo. Se creó un método *contains(string word, bool prefix = false)* que

busca una palabra o un prefijo según el valor ingresado de prefix. Si prefix es falso, se busca una palabra, y si prefix es verdadero, se busca un prefijo.

```
bool containsWord(string word) {
    TrieNode*current = root;
    for (unsigned int i = 0; i <word.size(); i++) {
        if (!current->contains(word[i])) {
            return false;
        }
        current = current->getChild(word[i]);
    }
    return current->getIsWord();
}
bool containsPrefix(string prefix) {
    TrieNode*current = root;
    for (unsigned int i = 0; i <word.size(); i++) {
        if (!current->contains(word[i])) {
            return false;
        }
        current = current->getChild(word[i]);
    }
    return true;
}
```

Figura 17. Código original de *containsWord(string word)* y *containsPrefix(string prefix)*

```
bool Trie::contains (std::string word, bool prefix) {
    TrieNode *current = root;
    for (unsigned int i = 0; i < word.size (); i++) {
        if (!current->contains (word[i]))
            return false;
        current = current->getChild (word[i]);
    }
    return (prefix ? true : current->getIsWord ());
}
```

Figura 18. Unión de los métodos *containsWord(string word)* y *containsPrefix(string prefix)* en *contains(string word, bool prefix = false)*

Leer y escribir acentos y caracteres especiales del español

Uno de los requisitos del programa es que las palabras del archivo de texto a analizar deben contener letras utilizadas en el idioma español: acentos, diéresis y eñes. Para leer y escribir caracteres especiales del idioma español en una aplicación de consola C se usó el paquete *locale* y se estableció el idioma a español con *setlocale(LC_ALL, "spanish")*. También se usa la instrucción *SetConsoleCP(1252)*, 1252 es la codificación de los caracteres latinos, la consola de Windows usa 850 por defecto y esta no tiene los caracteres españoles.

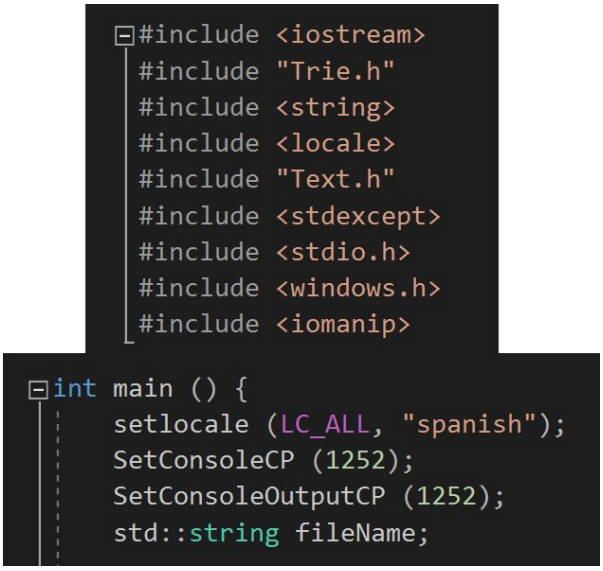

Figura 19. Establecimiento del idioma español con el uso de las bibliotecas *locale* y *windows*

Modificación de los métodos *getMatches* y *getMatchesAux*

```
List< KVPair<std::string, List<int>*> >* Trie::getMatches (std::string prefix) {
    List< KVPair<std::string, List<int>*> > *words = new DLinkedList< KVPair<std::string, List<int>*> > ();
    getMatchesAux (root, prefix, "", words);
    return words;
}

void Trie::getMatchesAux (TrieNode* current, std::string rest, std::string prefix, List< KVPair<std::string, List<int>*> >* words) {
    if (rest.size () == 0 && current->getIsWord ()) {
        KVPair<std::string, List<int>*> p (prefix, current->getLineNumbers ());
        words->append (p);
    }

    List<char> *children = current->getChildren ();
    for (children->goToStart (); !children->atEnd (); children->next ()) {
        char c = children->getElement ();
        if (rest.size () == 0) {
            std::string newPrefix = prefix;
            newPrefix.append (1, c);
            getMatchesAux (current->getChild (c), rest, newPrefix, words);
        } else if (c == rest[0]) {
            std::string newPrefix = prefix;
            newPrefix.append (1, c);
            rest.erase (0, 1);
            getMatchesAux (current->getChild (c), rest, newPrefix, words);
            break;
        }
    }
    delete children;
}
```

Figura 20. Métodos *getMatches* y *getMatchesAux* de la clase Trie

Modificando los métodos *getMatches* y *getMatchesAux* se pudo obtener una lista de palabras con n cantidad de letras. Estos retornan una lista de pares *<string, List<int>*>* y así se pudo tener acceso a la palabra y al números de línea de la palabra correspondiente. De esta manera, el método *getMatches* es utilizado para resolver la consulta por prefijo y la búsqueda de una palabra en el archivo de texto.

```
void printMatchesbyPrefix (std::string prefix) {
    List< KVPair<std::string, List<int>*> > *matches = trie.getMatches (prefix);
    for (matches->goToStart (); !matches->atEnd (); matches->next ())
        std::cout << matches->getElement ().getKey () << "\t" << matches->getElement ().getValue ()->getSize () << "\n";
    std::cout << std::endl;
    delete matches;
}
```

Figura 21. Método *printMatchesbyPrefix(string prefix)* que se utiliza para consultar por prefijos en la consola

```
void printWord (std::string word) {
    List< KVPair<std::string, List<int>*> > *matches = trie.getMatches (word);
    List<int> *lineNumbers;
    for (matches->goToStart (); !matches->atEnd (); matches->next ()) {
        if (matches->getElement ().getKey () == word) {
            lineNumbers = matches->getElement ().getValue ();
            for (lineNumbers->goToStart (); !lineNumbers->atEnd (); lineNumbers->next ())
                std::cout << lineNumbers->getElement () << "\t" << text->getLine (lineNumbers->getElement ()) << "\n\n";
            std::cout << std::endl;
        }
    }
    delete matches, lineNumbers;
}
```

Figura 22. Método *printWord(string word)* que se utiliza para buscar una palabra en la consola

La consulta por prefijo necesita saber la cantidad de veces que aparece cada palabra en el archivo original y la búsqueda de una palabra requiere el número de línea de la palabra para imprimir la línea de texto. Para la búsqueda de una palabra también se necesita conocer cuántas veces se repite una palabra en el texto; esto se llevó acabo utilizando el método *getSize()* de la lista de números.

Implementación de los métodos *getSizeMatches* y *getSizeMatchesAux*

```
List<std::string>* Trie::getSizeMatches (int size) {
    List<std::string> *words = new DLinkedList<std::string> ();
    getSizeMatchesAux (root, "", size, words);
    return words;
}

void Trie::getSizeMatchesAux (TrieNode * current, std::string prefix, int size, List<std::string>* words) {
    if (prefix.size () == (unsigned) size && current->getIsWord ())
        words->append (prefix);

    List<char> *children = current->getChildren ();
    for (children->goToStart (); !children->atEnd (); children->next ()) {
        char c = children->getElement ();
        std::string newPrefix = prefix;
        newPrefix.append (1, c);
        getSizeMatchesAux (current->getChild (c), newPrefix, size, words);
    }
    delete children;
}
```

Figura 23. Métodos *getSizeMatches* y *getSizeMatchesAux* de la clase Trie

Se implementaron los métodos *getSizeMatches* y *getSizeMatchesAux* en la clase Trie para realizar la búsqueda de palabras por cantidad de letras. El algoritmo

implementado utiliza el método *getChildren()* de la clase TrieNode que consiste en obtener todos los caracteres hijos del nodo actual. Cada vez que se encuentra una prefijo que sea una palabra y que contenga la cantidad de letras necesarias se agrega a la lista de palabras que retorna *getSizeMatches*. Estos métodos son requeridos en la búsqueda de una palabra por cantidad de letras.

```
void printMatchesBySize (int size) {
    List<std::string> *list = trie.getSizeMatches (size);
    for (list->goToStart (); !list->atEnd (); list->next ())
        std::cout << list->getElement () << std::endl;
    std::cout << std::endl;
    delete list;
}
```

Figura 24. Método *printMatchesBySize(int size)* que se utiliza para buscar una palabra por cantidad de letras en la consola

Referencias Bibliográficas

Cppreference.com. (2018, Junio 15). *Std::remove, std::remove_if*. Recuperado de https://en.cppreference.com/w/cpp/algorithm/remove

Cppreference.com. (2018, Octubre 23). *Lambda expressions (since C++11)*. Recuperado de https://en.cppreference.com/w/cpp/language/lambda

Cygwin authors. (n.d.). *Internationalization*. Recuperado de https://cygwin.com/cygwin-ug-net/setup-locale.html

Git-scm.com. (2018, Enero 17). *Documentation*. Recuperado de https://git-scm.com/docs/user-manual.html

GitLab. (2018). The first single application for the entire DevOps lifecycle. Recuperado el 28 de octubre del 2018 de https://gitlab.com/

GitLab. (2018). GitLab Team Page. Recuperado el 28 de octubre del 2018 de https://about.gitlab.com/company/team/

Microsoft. (2018). Visual Studio Live Share | Visual Studio. Recuperado el 28 de octubre del 2018 de https://visualstudio.microsoft.com/es/services/live-share/

Microsoft. (2018). VS Live Share - Preview - Visual Studio Marketplace. Recuperado el 28 de octubre del 2018 de https://marketplace.visualstudio.com/items?itemName=MS-vsliveshare.vsls-vs

Tutorialspoint.com. (2018). *C Library <locale.h>*. Recuperado de https://www.tutorialspoint.com/c_standard_library/locale_h.htm