INSTITUTO TECNOLOGICO DE COSTA RICA

INGENIERÍA EN COMPUTADORES

ALGORITMOS Y ESTRUCTURAS DE DATOS I

Proyecto 2

PROFESOR ING. LUIS DIEGO NOGUERA MENA

MARCO ANDRÉS VILLATORO CHACÓN - 2021057692

Octubre 28 2022

Contenidos:

Descripción del problema

El objetivo es el desarrollo de la aplicación de escritorio Text Finder. Text Finder es una aplicación de escritorio desarrollada en java con la funcionalidad principal de realizar búsquedas de texto en documentos de tipo .txt; .pdf y .docx mediante un cliente por el cual el usuario puede interactuar directamente con la aplicación y un servidor encargado de realizar la búsqueda de texto mediante el uso de un árbol binario de búsqueda[1] , la comunicación de ambos se da por medio de sockets. Al realizar la búsqueda, la aplicación muestra el nombre de los archivos donde se encontró la palabra, los resultados pueden ser ordenados según la instrucción del usuario por el nombre del archivo a través de un algoritmo de ordenamiento Quicksort[2].

Las funcionalidades principales de la aplicación son la administración la biblioteca de documentos donde se va a efectuar la búsqueda, la indización de la biblioteca en un árbol binario de búsqueda[1], la búsqueda de texto dentro de la librería de documentos y la capacidad de abrir el documento donde se encontraron ocurrencias de la búsqueda.

Adicionalmente se documenta el código mediante el uso de la herramienta Javadoc generado su respectivo archivo HTML y se redacta la presente documentación externa como parte de un documento entregable el cual adicionalmente contine los enlaces para acceder al repositorio de GitHub utilizado para el manejo del código y el repositorio de Microsoft Azure DevOps utilizado para la planificación y administración del proyecto.

1. ^ En la especificación de la aplicación se indica el uso de un árbol binario de búsqueda así como un árbol AVL, sin embargo por situaciones externas este contexto en particular solo presenta el árbol binario de búsqueda.
2. ^ En la especificación de la aplicación se indica el uso de tres algoritmos de ordenamiento (Quicksort, Bubblesort, Radixsort) sin embargo por situaciones externas este contexto en particular solo presenta el algoritmo Quicksort.

Diagramas de clases

⇒ Server

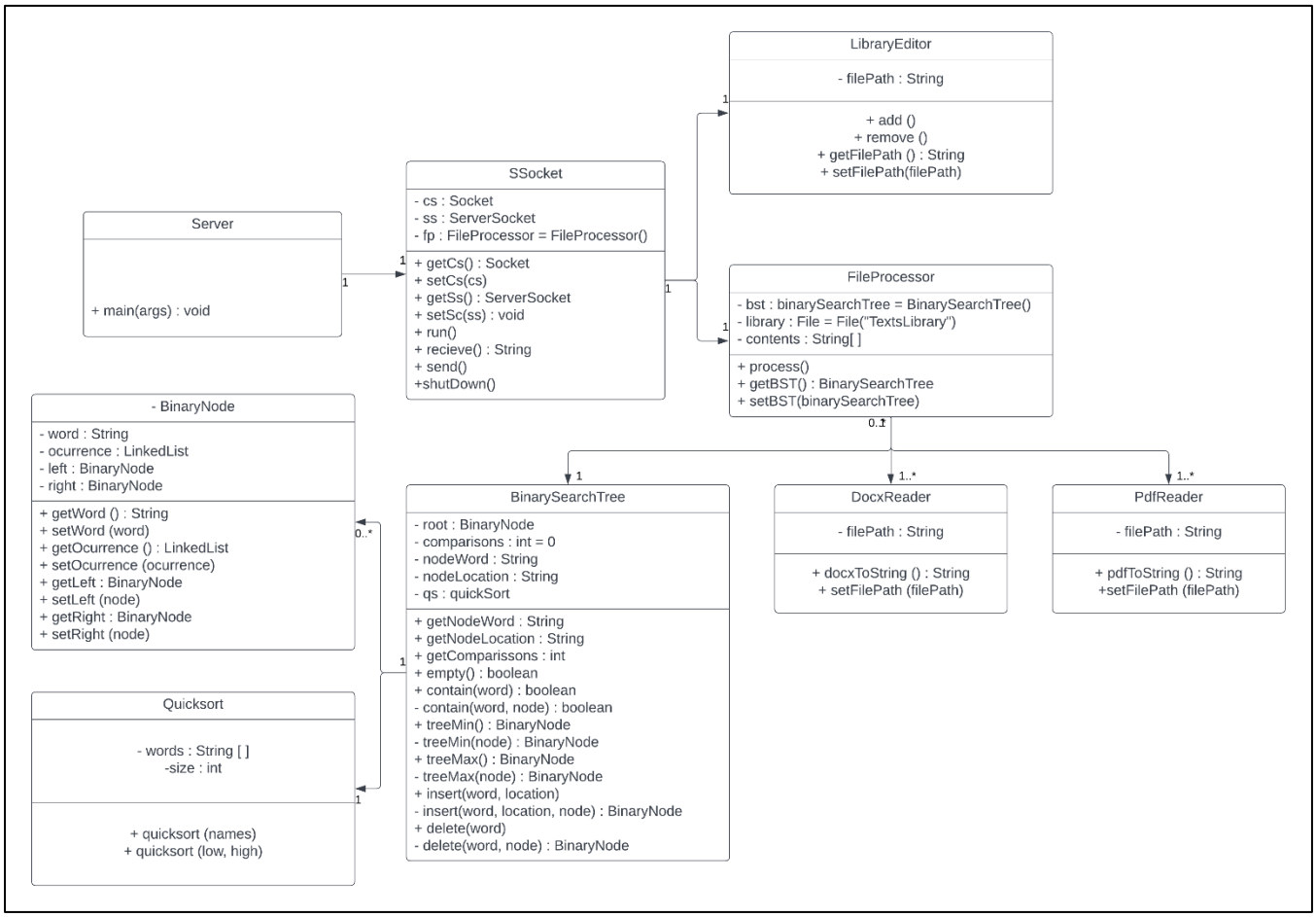

Figura 1. Diagrama de clases para la aplicación *Server*

⇒ Client

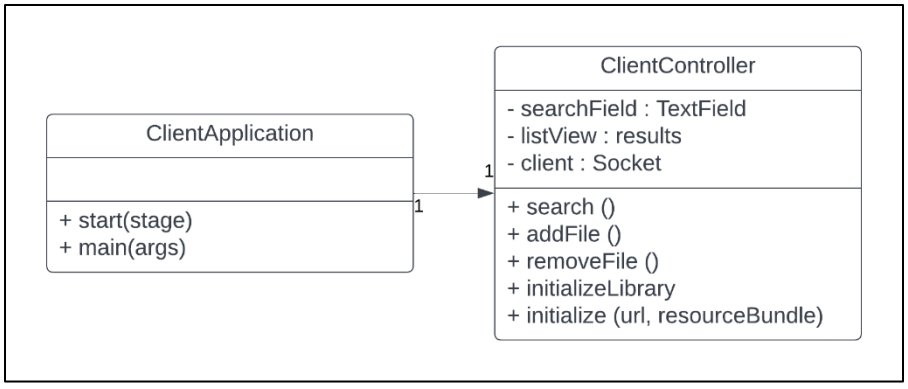

Figura 2. Diagrama de clases para la aplicación *Client*

Descripción de las estructuras de datos

BinarySearchTree corresponde a un árbol binario de búsqueda estándar. Principalmente se utilizan los métodos de inserción y verificación de contenido los cuales se muestran en las Figura 3 y 4 respectivamente.

```
/** Calls the insert() method */
public void insert (String word, String location) {
    this.root = this.insert(word, location, this.root);
}

/** Recursively inserts a node to the tree in its corresponding location */
private BinaryNode insert (String word, String location, BinaryNode node) {
    if ( node == null){
        return new BinaryNode(word, location,null, null);
    }

    int compare = word.compareTo(node.getWord());

    if (compare < 0){
        node.setLeft(this.insert(word, location, node.getLeft()));
    } else if (compare > 0){
        node.setRight(this.insert(word, location, node.getRight()));
    } else {
        node.occurrence.add(location);
    }
    return node;
}
```

Figura 3. Definición de los métodos para la inserción de nodos

```
/** Calls the contain method */

public boolean contain(String word){
    return this.contain(word, this.root);
}

/** Recursively verifies if a node with a given element exists within the tree */

private boolean contain (String word, BinaryNode node) {
    if (node == null){
        return false;
    } else {
        int compare = word.compareTo(node.getWord());

        if (compare < 0){
            return contain(word, node.getLeft());
        } else if (compare > 0) {
            return contain(word, node.getRight());
        } else {
            nodeWord = word;
            nodeLocation = node.getOccurrence().toString();
            return true;
        }
    }
}
```

Figura 4. Definición de los métodos para la verificación de contenido

También se definen el método *empty()* para la verificación del estado de un árbol como uno con contenido o vacío, este método se utiliza en la definición de los métodos de utilidad interna *treeMin()* y *treeMax()*, estos obtienen el contenido del nodo con el menor y mayor valor respectivamente los cuales se requieren para la definición del método *delete()* encargado de eliminar nodos del árbol. El método *delete()* se incluye por formalidad, su definición se muestra en la figura 5.

```
/** Calls the delete method */
public void delete (String word) {
    this.root = this.delete(word, this.root);
}

/** Recursively deletes a node with a given element */
private BinaryNode delete (String word, BinaryNode node) {

    if (node == null){
        return node;}

    int compare = word.compareTo(node.getWord());

    if (compare < 0) {
        node.setLeft(delete(word, node.getLeft()));
    } else if (compare > 0) {
        node.setRight(delete(word, node.getRight()));
    } else if (node.getLeft() != null && node.getRight() != null) {
        node.setWord(treeMin(node.getRight()).getWord());
        node.setRight(delete(node.getWord(), node.getRight()));
    } else {
        node = node.getLeft() != null ? node.getLeft() : node.getRight();
    }
    return node;
}
```

Figura 5. Definición de los métodos para eliminar nodos

En la implementación como clase, se define el atributo básico *root* siendo un nodo binario de la clase *BinaryNode* con función de facilitar el acceso a la información almacenada en el árbol y un atributo *comparissons* se utiliza para almacenar la cantidad de comparaciones hechas en las búsquedas, importantemente se definen los atributos *nodeWord* y *nodeLocation* para almacenar los contenidos de un nodo en los casos que se obtenga un resultado de *true* del método *contain()*. En la figura 6 se muestran los atributos *nodeWord* y *nodeLocation* como se definen dentro de la clase.

```
/** Attribute 3, a String */
private String nodeWord;

/** Attribute 3, a String */
private String nodeLocation;
```

Figura 6. Atributos tipo string encargados de almacenar resultados de una búsqueda

BinaryNode corresponde a la clase para crear los nodos que conforman a *BinarySearchTree*, contiene los métodos para obtener y modificar su información. Como atributos se definen *word* y *ocurrence* para almacenar la información que contiene el nodo en momento de ser insertado a un árbol y los atributos *left* y *right* para indicar sus nodos hijos. Como se muestra en la figura 7, al construirse un nodo se modifica ocurrence con fin de almacenar toda aparición de una palabra dentro de una librería de documentos.

```
/** Class constructor 1 */
public BinaryNode (String word, String occurrence){
    this(word, occurrence, null, null);
}

/** Class constructor 2 */
public BinaryNode (String word, String location, BinaryNode left, BinaryNode
right){

    this.word = word;

    if (this.occurrence == null) {
        this.occurrence = new LinkedList();
        occurrence.add(location);
    } else {
        this.occurrence.add(location);
    }

    this.left = left;
    this.right = right;
}
```

Figura 7. Constructores de la clase *BinaryNode*

Descripción de los algoritmos desarrollados

Quicksort() corresponde a un algoritmo de tipo *Quicksort* adaptado para ordenar un arreglo de datos clase *String* mediante el método *string.compareTo(string)* como se muestra en la figura 8. Se definen dos métodos *quicksort()* de manera sobrecargada, uno con la función de reordenar arreglos de manera recursiva y otro encargado de hacer la llamada al otro método indicando los extremos del arreglo a ordenar

```java
public void quicksort(String names[]){
    if (names == null || names.length == 0){
        return;
    }
    this.words = names;
    size = names.length;
    quicksort(0, size - 1);
}

/** Sorts a string array using a Quicksort algorithm */
private void quicksort(int low, int high) {
    int i = low;
    int j = high;
    String pivot = words[low + (high - low) / 2];

    while (i <= j) {
        while (words[i].compareTo(pivot) < 0) {
            i++;
        }
        while (words[j].compareTo(pivot) > 0) {
            j--;
        }
        if ( i <= j) {
            String temp = words[i];
            words[i] = words[j];
            words[j] = temp;
            i++;
            j--;
        }
    }
    if (low < j) {
        quicksort(low, j);
    }
    if (i < high) {
        quicksort(i, high);
    }
}
```

Figura 8. Definición del algoritmo adaptado al ordenamiento strings

Enlaces

Repositorio de GitHub correspondiente al lado del servidor:

- https://github.com/Specktic/TextFinderServer.git

Repositorio de GitHub correspondiente al lado del cliente

- https://github.com/Specktic/TextFinderClient.git

Repositorio de Azure DevOps:

- https://dev.azure.com/Specktic/Project%202%20Text%20Finder