

Tarea Integradora II

Juan Fernando Martínez

Santiago Rodas

Alejandra Díaz Parra

Facultad de Ingeniería, Universidad ICESI

Algoritmos y Estructuras de Datos

Juan Manuel Reyes

11 de noviembre de 2020

Método de la ingeniería

Fase 1: Identificación del problema

Requerimientos funcionales

1. Generar mil millones de registros con los siguientes campos:
 - A) Código
 - B) Nombre
 - C) Apellido
 - D) Sexo
 - E) Fecha de nacimiento
 - F) Estatura
 - G) Nacionalidad
 - H) Fotografía
2. Mostrar una barra de progreso para indicar cuánto tiempo se demora la operación, en el caso tal que esta tenga un estimado mayor a un 1 segundo.
3. Indicar cuántos registros se quiere generar, a pesar de que el predeterminado sea el máximo posible.
4. Agregar nuevos registros al sistema.
5. Guardar los registros que se agregaron o se generaron en el sistema.
6. Actualizar la información de un registro que el usuario desee cambiar.
7. Eliminar un registro existente en el sistema.
8. Consultar los registros de acuerdo a los siguientes criterios:
 - A) Nombre completo
 - B) Nombre
 - C) Apellido
 - D) Código
9. Mostrar una lista emergente debajo del campo de búsqueda, a medida que el usuario digite la información correspondiente (excepto para código). Esto con la idea de hacer la búsqueda anteriormente mencionada.
10. Calcular y mostrar la cantidad total de elementos que coinciden con el prefijo digitado por el usuario.

Fase 2: Recopilación de la información necesaria

Para solucionar este problema tan importante se requiere de una estructura que almacene eficientemente los datos: desde la lectura hasta la escritura, su velocidad debe de ser rápida. Además, se necesita evitar sobrecribir toda la información antes de que esta sea requerida. Es decir, un almacenamiento eficaz en general.

[Marco teórico esencial](#)

La definición de una estructura eficiente depende mucho del problema. Por ejemplo, con respecto a la búsqueda de información se encontraron diversas estructuras que se pueden utilizar dependiendo del tipo de dato que se almacena y el contexto del problema. Entre otros, para el caso de máximos y mínimos la estructura predilecta son los montículos; para los problemas de búsqueda se usan árboles ABB, AVL, Roji - Negros y Trie (estos últimos se usan para almacenar Strings teniendo en cuenta todos los prefijos de una cadena).

[Información acerca del trie](#)

También es importante entender esta información desde otro ángulo. En pocas palabras, se deben analizar los *requerimientos no funcionales*:

1. Cuando se generen los registros en el sistema, se debe de tomar los siguientes aspectos:
 - A) El código debe ser autogenerado con una cantidad específica de caracteres.
 - B) La estatura debe de estar en un intervalo conceptual (que se asemeje a lo real).
 - C) La fecha de nacimiento debe de estar distribuida según lo indicado.
 - D) Los nombres y apellidos deben ser tomados del dataset.
 - E) Se debe de combinar aleatoriamente cada nombre y/o apellido del dataset.
2. Todos los dataset utilizados en el sistema deben de estar en un directorio llamado “data”.
3. Las fotografías utilizadas para completar el perfil requerido del sistema, deben ser tomadas del siguiente sitio: [oficial numero uno](#) .
4. La proporción de edades debe seguir la siguiente distribución: [oficial numero dos](#) .
5. Las proporciones de cada nacionalidad deben estar acorde a los datos de población del siguiente enlace: [oficial número tres](#) .
6. La interfaz gráfica debe ser realizada con JavaFx.
7. Toda la información existente del sistema debe ser persistente.

8. Si se agrega o se actualiza un registro, el usuario no tendrá la opción de modificar el código del registro, ya que este debe ser autogenerado.
9. La lista emergente debe mostrar máximo 100 nombres parametrizables, los cuales deben empezar con los caracteres digitados hasta el momento.
10. Las búsquedas realizadas se deben manejar en un árbol ABB autobalanceado, el cual debe de tener una implementación propia.
11. Implementar adecuadamente las pruebas unitarias automáticas de las estructuras propias y de las operaciones principales del modelo
12. La interfaz gráfica tendrá un menú superior en la ventana, el cual permite cambiar todos los elementos cuando se esté trabajando en opciones diferentes.

Con base en estos requerimientos funcionales y no funcionales, es evidente que se necesita de una estructura eficiente en cuanto a todas las operaciones, principalmente en la operación de búsqueda (será escogida en la etapa de diseño).

Por otro lado, dentro de los requerimientos se establece que la información debe ser persistente. Entonces, se proponen las dos siguientes ideas: La primera funciona por medio de archivos de texto planos, que permite almacenar todos los atributos en forma de texto, y la segunda es usando serialización, que faculta almacenar objetos completos sin necesidad de guardar cada atributo.

Fase 3: Búsqueda de soluciones creativas

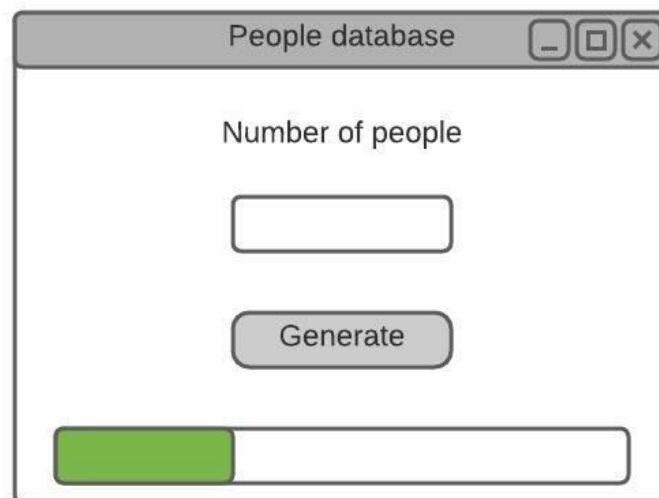
Para realizar esta etapa tan importante para el proyecto, se tomaron dos iniciativas importantes:

- Tomar la lluvia de ideas para la generación de ideas espontáneas que permitan resolver el problema de forma correcta y significativa.
- Dominar la lista de atributos para entender y visualizar todas las características o atributos especiales que puede tener el software en general.

Para la primera parte de estas soluciones planteadas se utilizó la herramienta Google Docs, ya que ésta permite plasmar de forma ordenada, calibrada y sincrónica todas las gestiones individuales y/o grupales que puede tener el proyecto informático. Con base a la plataforma utilizada, se definieron ciertas estrategias y estructuras que van acorde al planteamiento del problema. Es importante aclarar que cada búsqueda planteada va directamente asimilada con una estructura de datos. Éstas pueden ser:

- Árbol Binario de Búsqueda (ABB)
- Árbol AVL
- Trie
- Árbol Roji - Negro
- Lista enlazada
- Lista enlazada ordenada
- Arreglo
- Arreglo ordenado
- HashTable

Tomando en cuenta estas estructuras planteadas, se plantearon de una manera base los diseños de interfaz gráfica. Estos son los que verá el usuario al momento de utilizar el programa y ejecutarlo en su sistema:



People database

Name of the person

Search

People database

Name	<input type="text"/>
Last name	<input type="text"/>
Gender	<input type="text"/>
Height	<input type="text"/>
Nationality	<input type="text"/>

Add

People database

Name of the person

Change information

También es importante destacar que para la generación de edades y del género de la persona, tomando en cuenta la proporción de Estados Unidos, se calculó de la siguiente manera:

- Se usó la clase `SplittableRandom` para obtener un número entre 0 y 1, y así, ser distribuido de acuerdo a los intervalos proporcionales. Por ejemplo: si la proporción de personas entre 0-20 años es del 25% y el método retorna un valor entre 0 y 0.25, se le asigna una edad a las personas que estén en este rango.

Además, para las nacionalidades se sigue un razonamiento similar al anterior, donde se divide en intervalos los números entre 0 y 1, donde cada intervalo le corresponde una nacionalidad. El tamaño de cada intervalo corresponde a la proporción de la población de cada país americano. Ejemplo: Si la proporción de colombianos en América es del 5% el intervalo debe tener un tamaño de 0.05.

Por último, se plantearon ideas para la solución a la funcionalidad de predicción en resultados de búsqueda, éstos tomados por los atributos de nombre, nombre completo y apellido.

- Con el método “on key typed” por parte de la clase `TextField` de JavaFX se puede realizar alguna acción con cada letra digitada por el usuario. Es decir, el sistema reconocerá cada letra escrita sin necesidad de utilizar el botón constantemente.
- A medida que se escribe, se busca un nodo que tenga el nuevo caracter agregado. Esto con la idea de desplegar los elementos que se encuentren por debajo del mismo en el caso de Trie. El uso del trie se limitaría a la predicción de la búsqueda, pero la obtención del objeto con el valor buscado se tomaría de las estructuras donde se almacenan de acuerdo a cada criterio.
- Con la información ingresada por el usuario hasta cierto momento, buscar las personas que contengan su nombre y/o apellido para mostrarlos en la lista de búsqueda implementada para la solución eficaz del software.

Fase 4: Transición de la formulación de ideas a los diseños preliminares

Tomando en cuenta la fase anterior del método de la ingeniería, se obtienen las ideas candidatas para analizarlas y trabajarlas de forma más específica:

- Utilizar la estructura Trie en el sistema, ya que resulta ser eficaz para lo que se está buscando solucionar: la información almacenada aquí es un conjunto de claves, donde una clave es una secuencia de símbolos pertinentes a un alfabeto. El árbol se estructura de forma que cada letra de la clave se sitúa en un nodo, ocasionando que los hijos de un nodo representan las distintas posibilidades de símbolos diferentes que pueden continuar al símbolo representado por el nodo padre.

[Marco teórico utilizado para la herramienta.](#)

- Usar árboles AVL, ya que estos están siempre equilibrados de tal modo que para todos los nodos la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa. Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad $O(\log n)$.

[Información utilizada para mayor entendimiento de la estructura.](#)

- Los árboles Rojo - Negros resultan extremadamente factibles, ya que ofrecen un peor caso con tiempo garantizado para la inserción, el borrado y la búsqueda. No es esto únicamente lo que los hace valiosos en aplicaciones sensibles al tiempo como las aplicaciones en tiempo real, sino que además son apreciados para la construcción de bloques en otras estructuras de datos que garantizan un peor caso.

[En este sitio web se puede visualizar de forma correcta esta estructura.](#)

- El uso de la estructura de datos HashTable no es factible para los criterios de búsqueda que requieran implementar la función de autocompletar, pues la búsqueda de un elemento que se encuentre en ella tiene una complejidad temporal de $O(1)$ y, aunque esto es muy eficiente, no permite predecir los resultados de búsqueda. Sin embargo, la operación de búsqueda por código no requiere esta función, así que puede resultar útil manejarla con una HashTable.
- El uso de arreglos y listas enlazadas se descarta debido a que no son eficientes para las operaciones de buscar ni de eliminar. Las listas enlazadas podrían agregar en $O(1)$ pero no tiene utilidad si el buscar o eliminar tardan mucho más ($O(n)$) en ambas estructuras. Si se mantuvieran ordenadas, la operación de buscar se volvería $O(\log n)$ usando búsqueda binaria, pero agregar e eliminar serían $O(n)$.

Nota: No se analizan las propuestas de solución los demás sub problemas debido a que en todos los casos sólo hay una idea resultante de la lluvia de ideas.

Fase 5: Selección y evaluación de la mejor solución

Siendo M el tamaño de la entrada (String), k el tamaño promedio de una String en el trie, y n el número de datos en la estructura:

Estructura	Complejidad Temporal (caso promedio)			Complejidad Temporal (Peor caso)			Complejidad Espacial (Espacio usado por la estructura)
	Inserción	Eliminación	Búsqueda	Inserción	Eliminación	Búsqueda	
ABB	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Trie	$O(M)$	$O(M)$	$O(M)$	$O(M)$	$O(M)$	$O(M)$	$O(kn)$
Hash Table	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Arbol roji negro	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

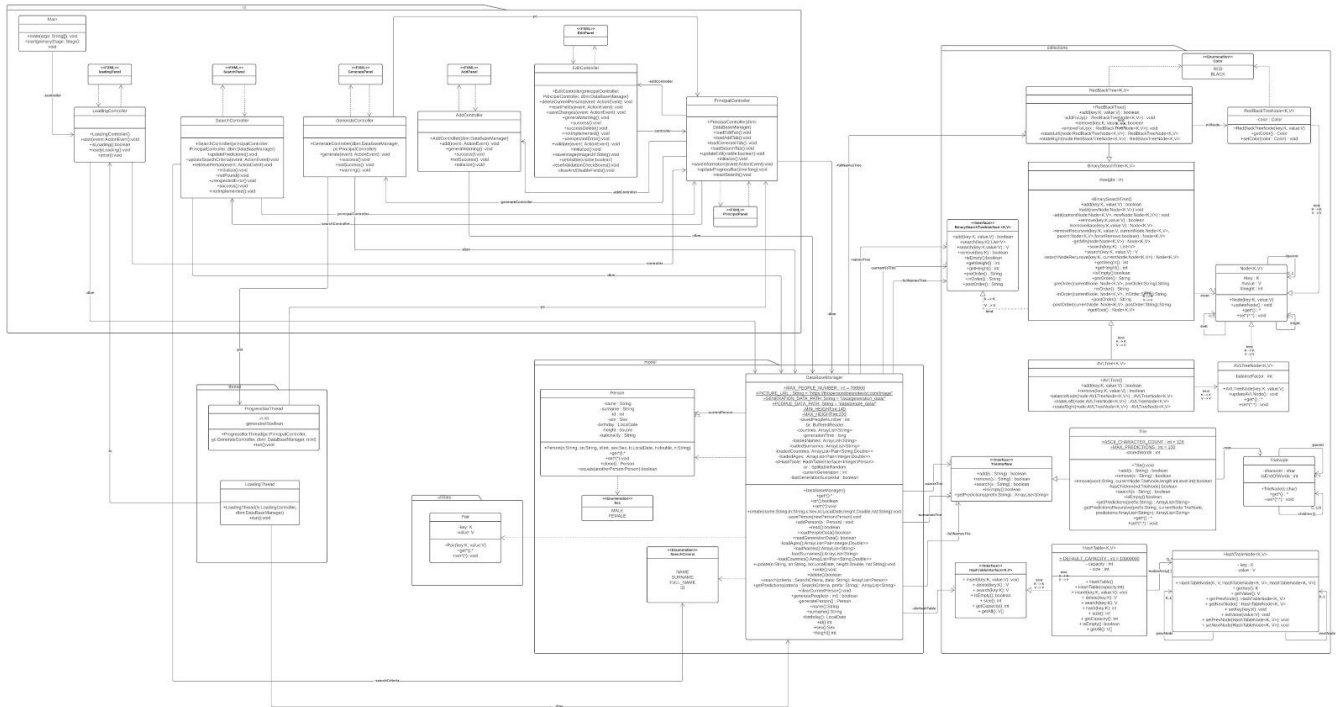
Las evaluaciones que consideramos importantes son:

- Trie para la predicción de resultados de búsqueda: Se inicia en el root del árbol; si el símbolo que se está buscando es A entonces la búsqueda continúa en el subárbol asociado al símbolo A que cuelga de la raíz. Se sigue de forma análoga hasta llegar al nodo hoja. Entonces se compara la cadena asociada a el nodo hoja y si coincide con la cadena de búsqueda entonces la búsqueda ha terminado en éxito, si no entonces el elemento no se encuentra en el árbol.
- AVL para nombre y apellido: El factor de equilibrio puede ser almacenado directamente en cada nodo. Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de rotaciones.
- Roji - Negro para el nombre completo: A pesar de que resulta atractiva y eficaz la idea de utilizar esta estructura para alguna búsqueda, no la se van a implementar por dos razones fundamentales:
 1. El tiempo estimado de entrega, junto a las obligaciones y funcionalidades básicas hacen que no esté en una mayor prioridad.

- Hash Table para el código: La búsqueda es muy rápida en promedio. Muy pocas veces será $O(n)$ porque los objetos se distribuyen de igual forma por todo el arreglo minimizando las colisiones.

Fase 6: Preparación de informes y especificaciones

- Diagrama de clases



- [Diseño de casos de pruebas](#)

Collections

Hash Table

Name	Class	Stage
setUp1()	TestHashTable	Creates an empty hash table whose keys and values are integers.

Class	Method	Setup	Input	Output
TestHashTable	testInsert1()	setUp1()	One million of elements in the system.	Equals, that is the number correct of elements in the system.
TestHashTable	testInsert2()	setUp1()	Five elements with the same key.	True, five are the elements in the system with the same key.
TestHashTable	testInsert3()	setUp1()	Two elements with different information.	False, the system inserts 2 elements, and checks if the method is empty.
TestHashTable	testDelete1()	setUp1()	A random key.	Null, the system tried to delete an element that doesn't exist.
TestHashTable	testDelete2()	setUp1()	One element with a random key.	Equals, the final size of the hash table is 0.
TestHashTable	testDelete3()	setUp1()	One million of elements in the system.	True, the system deletes N elements from the M created.
TestHashTable	TestSearch1()	setUp1()	One element with a random key.	True, the object is not null.
TestHashTable	TestSearch2()	setUp1()	Three elements with different information.	True, the information is the same.
TestHashTable	TestSearch3()	setUp1()	A random key.	Null, system tries to search for an object that doesn't exist.

Collections

AVL Tree

Name	Class	Stage
setUp1()	TestAVLTree	Creates an empty AVL tree whose keys and values are integers.
setUp2()	TestAVLTree	Creates an AVL tree with five elements with the following keys and values of type integer: <ul style="list-style-type: none"> • (3,7) • (6,10) • (17,5) • (9,36) • (24,21)

Class	Method	Setup	Input	Output
TestAVLTree	addTest1()	setUp1()	x = 2; y = 6;	True. The node has been added in the root of the tree, the tree is not empty and its height and weight is 1.
TestAVLTree	addTest2()	setUp2()	key = 60; value = 8; key = 14; value = 19; key = 22; value = 30; key = 13; value = 4;	True, every new node has been added, the tree is not empty and its weight is 9.
TestAVLTree	addTest3()	setUp1()	200000 new nodes whose keys and values are Integers less than 2000.	True, every new node has been added and the weight of the tree is 200000.
TestAVLTree	removeTest1()	setUp1()	x = 35; y = 70;	False because there aren't elements in the tree.
TestAVLTree	removeTest2()	setUp2()	key = 3; value = 7; key = 24; value = 21; key = 17;	True. Every element has been removed successfully, and the new height and weight of the tree is 2.

			value = 5	
TestAVLTree	removeTest3()	setup1()	200000 new nodes whose keys go from 0 to 200000 and whose values go from 1 to 200001. All of those nodes are the ones that are going to be erased.	True. Every element has been removed successfully, and the tree is empty.

Collections

Binary Search Tree

Name	Class	Stage
setUp1()	TestBinarySearchTree	Creates an empty binary search tree whose keys and values are integers.
setUp2()	TestBinarySearchTree	Creates a binary search tree with 50000 elements which keys are 1 and which values are 5.
setUp3()	TestBinarySearchTree	<p>Creates a binary search tree with seven elements with the following keys of type integer:</p> <ul style="list-style-type: none"> • 50 • 25 • 75 • 37 • 14 • 63 • 80 <p>All of the values associated with those keys are equal to zero.</p>
setUp4()	TestBinarySearchTree	<p>Creates a binary search tree with five elements with the following keys and values of type integer:</p> <ul style="list-style-type: none"> • (1,15) • (2,30) • (3,45) • (4,60) • (5,75)

Class	Method	Setup	Input	Output
TestBinarySearchTree	testAdd1()	setup1()	key = 1; value =125;	The root is not null, the element was added successfully.
TestBinarySearchTree	testAdd2()	setup1()	One hundred thousand elements with random information.	True, the weight is 100000.
TestBinarySearchTree	testAdd3()	setup1()	Two elements with the key 1. The value is 100 and 200 respectively.	True, the weight of the tree is 2.
TestBinarySearchTree	testRemove1()	setup1()	One element with a random key and value.	True. The element was removed and its weight is 0.
TestBinarySearchTree	testRemove2()	setup2()	Five thousand elements with keys equal to 1 and values equal to 5 .	True, the actual weight is less than the original.
TestBinarySearchTree	testRemove3()	setup3()	key = 50; value = 0; key = 25; value = 0;	True, some of those elements are removed.
TestBinarySearchTree	testSearch1()	setup3()	key = 14; value= 0;	False, the element doesn't exist.
TestBinarySearchTree	testSearch2()	setup3()	key = 105	Null, , the element doesn't exist.
TestBinarySearchTree	testSearch3()	setup3()	key = 75.	Not null, the element does exist.
TestBinarySearchTree	testPreOrder()	setup4()	Five elements.	True, the String is different to null.
TestBinarySearchTree	testInOrder()	setup4()	Five elements.	True, the String is different to null.
TestBinarySearchTree	testPostOrder()	setup4()	Five elements.	True, the String is different to null.

Collections

Red Black Tree

Name	Class	Scenario
setUp1()	RedBlackTree	Creates an empty tree whose keys and values are integers.
setUp2()	RedBlackTree	Creates an empty tree whose keys and values are strings.
setUp3()	RedBlackTree	Creates a tree with five elements with the following keys and values of type integer: <ul style="list-style-type: none"> • (81,9) • (72,18) • (63,27) • (54,36) • (45,45)

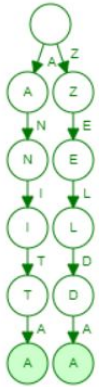
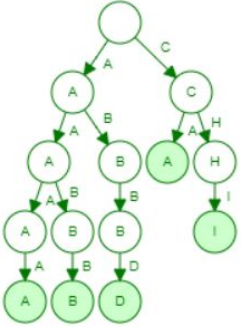
Class	Method	Setup	Input	Output
TestRedBlackTree	testAdd1()	setUp1()	key = 13; value = 78;	True. The node has been added in the root of the tree, the tree is not empty and its height and weight is 1.
TestRedBlackTree	testAdd2()	setUp1()	key = 16 value = 5 key = 2 value = 93 key = 44 value = 8 key = 31 value = 60 key = 7 value = 59 key = 25 value = 1	True. Every node has been added.
TestRedBlackTree	testAdd3()	setUp1()	We add three elements with the same value.	True, all the elements are in the system.
TestRedBlackTree	testAddFixUp1()	setUp1()	We add three elements with the idea to	Equals, the weight is correct.

			rotate every one.	
TestRedBlackTree	testAddFixUp2()	setup1()	We add five elements with the idea to rotate every one.	True, every information it's in the system.
TestRedBlackTree	testAddFixUp3()	setup1()	We add one hundred elements with the idea to rotate every one.	Equals, the weight is correct.
TestRedBlackTree	testRemove1()	setup1()	key = 65; value = 22;	False because the tree does not have elements to be erased.
TestRedBlackTree	testRemove2()	setup3()	The info of the nodes to be removed: <ul style="list-style-type: none"> • key = 72, value=18 • key = 63, value=27 • key = 54, value=36 • key = 45, value=45 	True. All of the input nodes have been erased and the tree is now empty.
TestRedBlackTree	testRemove3()	setup1()	Five elements in the system.	True, we remove all the elements and the weight is 0.
TestRedBlackTree	testRemoveFixUp1()	setup1()	Ten elements in the system.	True, all the actions it's correct.
TestRedBlackTree	testRemoveFixUp2()	setup1()	One hundred elements in the system.	True, all the actions it's correct.
TestRedBlackTree	testRemoveFixUp3()	setup1()	Only one element.	True, all the actions it's correct.

Collections

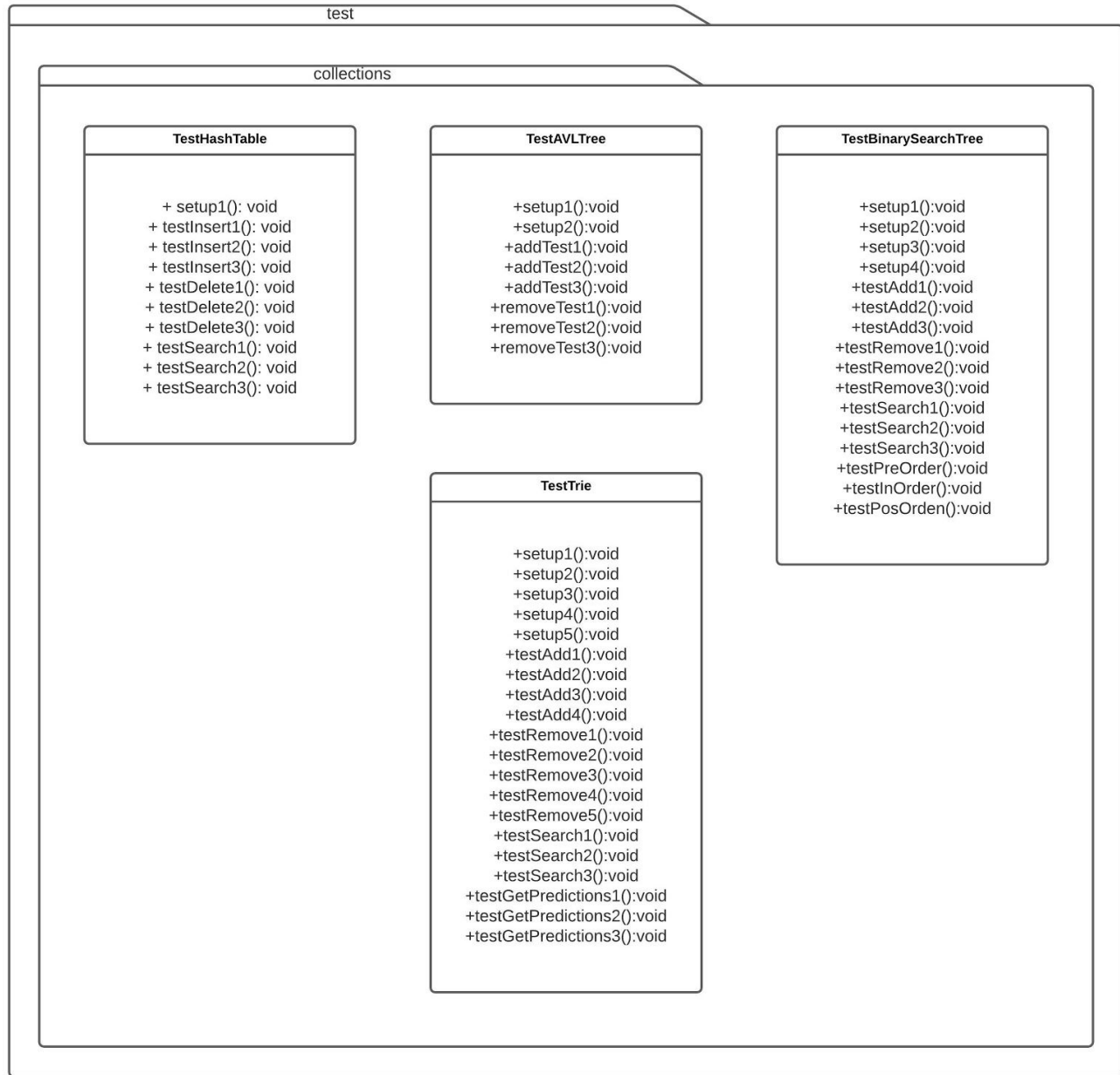
Trie

TestTrie	testAdd1()	setUp1()	"Ana" "Anita" "Felix" "Zelda"	
TestTrie	testAdd2()	setUp2()	"Feliza" "Analu" "Camilo"	
TestTrie	testAdd3()	setUp3()	"AA" "A" "ABA"	
TestTrie	testAdd4()	setUp3()	"AAAA" "AAA" "CAI" "AABB" "AB" "CHI" "ABBD" "CA"	
TestTrie	testRemove1()	setUp1()	"AAA"	False because there are no words in the trie so its structure remains the

				same.
TestTrie	testRemove2()	setUp2()	"Felix" "Ana" "Camilo"	
TestTrie	testRemove3()	setUp3()	"AAA" "AB" "CAI" "CASA"	
TestTrie	testRemov4()	setUp3()	"AAAA" "AAA" "CAI" "AABB"	All the asserts return true, because the trie is empty.
TestTrie	testRemove5()	setUp4()	2x "AAAA" "AAA" "CAI" "AABB" "AB" "CHI" "ABBD" "CA"	All return true, Empty trie
TestTrie	testSearch1()	setUp1()	"Pedro" "Camilo" "Felipe"	3x False
TestTrie	testSearch2()	setUp2()	"Anita" "Felix"	true true

			"Felipe" "Sara" "Carolina:	false false false
TestTrie	testSearch3()	setUp3()	"AA" "CA" "CAI" "ABBD" "AABC"	false true true true false
TestTrie	testGetPredictions1()	setUp1()	"AAA" "IDK"	Empty(null) list for both as there are no words in trie
TestTrie	testGetPredictions2()	setUp3()	"" "A" "AA" "ZEE"	List of words: all words List of words: <"AAAA", "AABB"> Null list
TestTrie	testGetPredictions3()	setUp5()	"" "A"	List Containing the first 100 words in the trie List containing the first 100 words or less that start with "A"

- [Diagrama de pruebas](#)



Fase 7: Implementación del diseño

[Ver repositorio en GitHub](#)