MÉTODO DE LA INGENIERÍA

LUISA FERNANDA QUINTERO FERNÁNDEZ

JUAN DAVID PELÁEZ VALENCIA

ALGORITMOS Y ESTRUCTURAS DE DATOS

UNIVERSIDAD ICESI

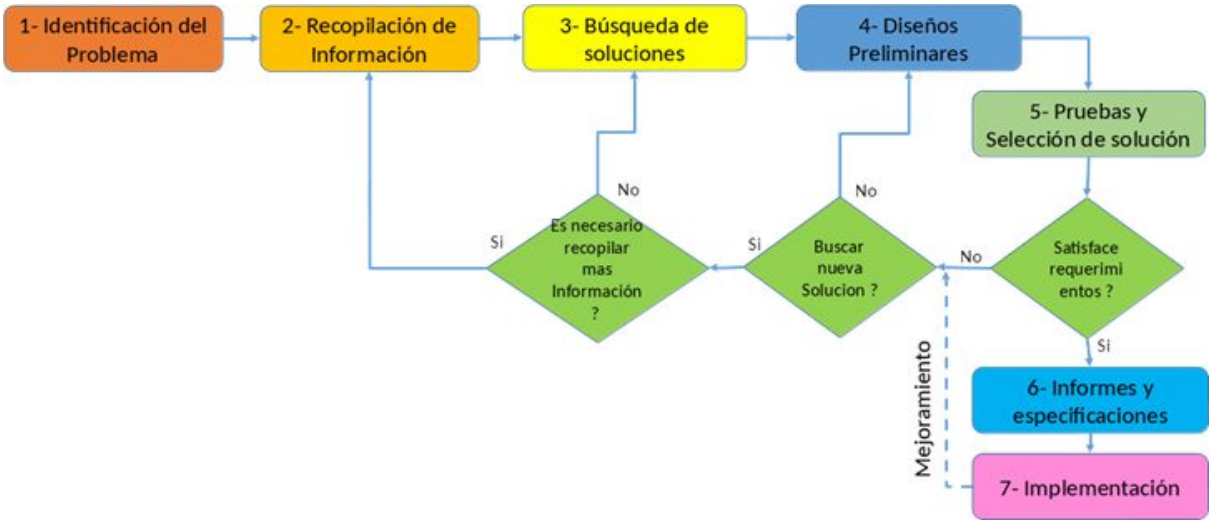
Contexto problemático

Un proyecto de investigación interna dentro de la Universidad desea un software el cual le permita manejar las operaciones CRUD (Create, Read, Update, Delete) en sus bases de datos, en donde su rango de datos comienza desde las mil millones de personas.

Se requiere que se puedan generar dichas personas con sus respectivos datos para poder manipularlos y, además implementar un sistema de búsqueda óptimo en donde mientras se escribe, aparecen sugerencias de búsqueda.

Desarrollo de la Solución

Seguiremos estos pasos mostrados en el siguiente diagrama de flujo para llegar al desarrollo de la solución:



Paso 1: Identificación del problema

Identificación de necesidades y síntomas

1. Se requiere un software capaz de manejar grandes cantidades de registros de personas, y que estas sean persistentes en el sistema.
2. Se desea una búsqueda que muestre sugerencias con base en registros que coincidan con lo digitado dentro del campo de texto.
3. Los registros deben de ser generados con diferentes datasets para cada campo, a excepción del código el cual es autogenerado.

Requerimientos

R1: Generar personas, el usuario puede escoger la cantidad de registros generados. Se generará máximo mil millones de personas con sus respectivos datos, los cuales son: código, nombre, apellido, sexo, fecha de nacimiento, estatura, nacionalidad y fotografía. No deben de haber códigos repetidos por lo cual estos se generarán automáticamente.

R2: Mostrar barra de progreso, se muestra una barra de progreso cuando la generación de los datos tarda más de un segundo, además debe de indicar cuanto tiempo tardo el proceso en generarse

R3: Guardar datos generados, el programa debe de guardar los datos que se generan para después poderlos consultar y utilizar. Los datos deben de ser persistentes.

R4: Actualizar persona, el programa puede actualizar todos los datos de la persona a excepción del código debido a que este es autogenerado y único para cada uno.

R5: Agregar persona, se agrega una nueva persona a la lista de datos con todos sus campos requeridos a excepción del código debido a que este es autogenerado.

R6: Eliminar persona, borra a una persona seleccionada de la base de datos.

R7: Buscar persona, se puede buscar a una persona por cualquiera de los siguientes criterios: Nombre, apellido, nombre completo y código. No se puede buscar por varios criterios al mismo tiempo, es decir, se debe de escoger por cuál de los 4 mencionados se va a realizar la búsqueda. Además, al lado del campo de texto se debe de mostrar un número que indica la cantidad de elementos que coinciden hasta ahora.

R8: Mostrar lista emergente, el programa debe de mostrar una lista emergente a medida que se va realizando la búsqueda de la persona, donde se mostraran máximo 100 nombres que empiecen con las letras ya digitadas en la búsqueda. Cuando queden 20 o menos nombres que coincidan con la búsqueda deberá aparecer una lista con los registros que coinciden y un botón con la opción de editar, en donde se podrá actualizar o eliminar dicho registro.

Definición del Problema

El equipo VIP de simulación requiere un software el cual le permita generar grandes cantidades de datos y poder utilizarlas. A su vez, requiere un sistema de búsqueda que arroje sugerencias de resultados mientras la persona escribe.

Paso 2: Recopilación de la información

Para poder entender y realizar apropiadamente hemos

Definiciones

1. **Autocompletar con JavaFX**

Javafx implementa su propio autocompletar en donde tiene su propio botón para autocompletar, o también tiene un método llamado bindAutoCompletion el cual toma como parámetros el texto ingresado, y como segundo parámetro una lista o un arreglo donde está toda la información de las palabras que se pueden sugerir mientras la persona escribe.

```
String[] autosuggestions = {"test","testing","etc etc"};

TextFields.bindAutoCompletion(inputTextField, autosuggestions);
```

2. **Arbol AVL:**

   Los árboles AVL son árboles BB donde todo nodo cumple la propiedad de equilibrado AVL:

   - La altura del subárbol izquierdo y del derecho no se diferencian en más de uno.

   Se define factor de equilibrio de un nodo como:

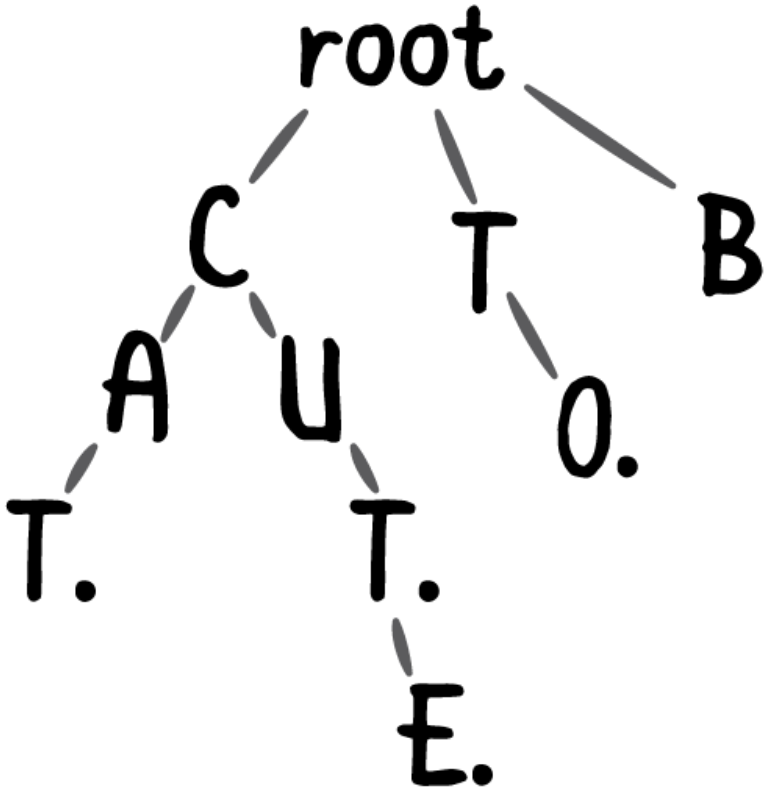
   - FactorEquilibrio(nodo) = altura(derecho) – altura(izquierdo)

   En un árbol AVL el factor de equilibrio de todo nodo es -1, 0 ó +1.

   Tras la inserción o borrado de un elemento, sólo los ascendientes del nodo pueden sufrir un cambio en su factor de equilibrio, y en todo caso sólo en una unidad.

3. **TRIE**

El trie es una estructura de datos similar a un árbol en donde los nodos almacenan el alfabeto, en donde una palabra puede ser recuperada atravesando una rama del árbol. Cada trie tiene un nodo vacío llamado “root” el cual contiene referencias a otros nodos donde cada nodo contiene una letra del alfabeto



4. **Árboles rojinegro:**

Son una estructura de datos de arboles binarios balanceados donde cada nodo puede ser de color rojo o negro. Tiene las siguientes reglas:

1. Todas las hojas son negras y no son relevantes(No contienen datos)
2. Si un nodo es rojo sus hijos deben de ser negros
3. La raíz es negra
4. Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.
5. Todo nodo es rojo o negro (no ambas)

La altura de un arbol es la altura negra de su raiz, que a su vez, es el numero de nodos negros que hay en el camino hasta descender a las hojas.

5. **Treap:**

El treap es una forma de árbol binario de búsqueda que mantiene un conjunto dinámico de llaves ordenadas y permiten búsqueda binaria sobre las llaves almacenadas. Su altura es proporcional al logaritmo del número de llaves, de modo que cada operación de búsqueda, inserción, o borrado toma tiempo logarítmico.

El treap soporta las siguientes operaciones básicas:

1. Para buscar un valor de una llave dada, se aplica el algoritmo de búsqueda estándar sobre un árbol binario de búsqueda ignorando las prioridades.
2. Para insertar una nueva llave x al treap, se genera una prioridad aleatoria y para x. Luego se inserta x aplicando el algoritmo estándar de inserción sobre un árbol binario de búsqueda. Luego, mientras x no sea la raíz del árbol y tenga prioridad mayor que su padre z, realizar una rotación que intercambie la relación padre-hijo entre x y z.
3. Para eliminar un nodo x del treap, si x es una hoja del árbol, sencillamente lo borramos. Si x tiene solamente un hijo z, borramos x del árbol y hacemos z hijo del padre de x (o establecemos z como raíz del árbol si x no tiene padre). Finalmente, si x tiene dos hijos, intercambió su posición en el árbol con la posición de su sucesor inmediato z cuando las llaves son ordenadas, resultando en uno de los casos anteriores. En este caso final, el intercambio puede violar la propiedad de heap para z, así que rotaciones adicionales serán necesarias para restaurar dicha propiedad.

Leer cuando un textfield cambia su texto:

```
TextField.textProperty().addListener(new ChangeListener<String>() {
@Override
public void changed(ObservableValue<? extends String> observable, String oldValue, String newValue) {
System.out.println(" Text Changed to " + newValue + ")\n");
}});
```

Fuentes:

https://stackoverflow.com/questions/36861056/javafx-textfield-auto-suggestions

https://stackoverflow.com/questions/31370478/how-get-an-event-when-text-in-a-textfield-changes-javafx

https://estructurasite.wordpress.com/arbol-avl/

https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014#:~:text=Thus%2C%20the%20worst%20case%20runtime,trie%20is%20O(mn).&text=The%20time%20complexity%20of%20searching,these%20operations%20O(an)

https://stackoverflow.com/questions/14397447/trie-vs-red-black-tree-which-is-better-in-space-and-time

http://webdiis.unizar.es/asignaturas/TAP/material/1.3.rojinegros.pdf

https://cp-algorithms.com/data_structures/treap.html

Paso 3: Búsqueda de soluciones creativas

Utilizamos el método de lluvia de ideas para plantear las siguientes ideas:

1. Utilizar la estructura de datos Trie como solución a la opción de autocompletar.
2. Utilizar el árbol binario de búsqueda balanceados (AVL o Árboles rojinegros) para autocompletar.
3. Utilizar otra estructura de datos de árboles balanceados llamada Treap
4. Utilizar InputReader para la lectura de datos
5. Colocar la lista de sugerencias de autocompletar en un VBox
6. Colocar la lista de sugerencias de autocompletar en un ScrollPane
7. Colocar un task para hacer que la barra de progreso avance progresivamente con la generación de datos.
8. Cada txt contenga mil millones de datos almacenados para generarlos en el software
9. Realizar una combinación de los datos para que estos den mil millones de personas, en ese caso se sugiere 10.000 apellidos y 100.000 nombres para al realizar una combinación de un total de mil millones de personas

Paso 4. Transición de las Ideas a los Diseños Preliminares

El treap es descartado debido a la falta de conocimiento sobre dicha estructura de datos (no fue vista en clase). Igualmente, implementar un bst normal no es factible debido a que el objetivo es conseguir tiempos de búsqueda eficientes.

Para la función de autocompletar se descarta la posibilidad de usar el método autocompletar de la clase TextField de JavaFX por razones obvias. Implementar métodos directamente en los árboles AVL o rojinegro resulta redundante. Por ello, implementar el Trie es una solución que resulta más práctica y reutilizable.

Revisando las alternativas restantes, tenemos que:

1. **Árbol AVL**: es una estructura de datos que se ajusta correctamente a las necesidades del problema. Además, se tiene el conocimiento necesario para implementarla.
.
2. **Trie**: es una estructura de datos que permite autocompletar cadenas de caracteres. El trie puede ser implementado en conjunción con cualquier otra estructura de datos para realizar búsquedas.

3. **Árbol Rojinegro**: es una estructura de datos que se ajusta correctamente a las necesidades del problema. Sin embargo, no se tiene el conocimiento necesario para implementarla. Por lo tanto, se necesita realizar un estudio dedicado.

Complejidad temporal de los algoritmos

Arbol AVL

La complejidad temporal de un árbol AVL en el peor de los casos es de O(log n) y su complejidad espacial es de O(n)

Árbol rojinegro

La complejidad temporal de un árbol rojinegro es O(log n) y en el peor caso para búsqueda de prefijos sería de O(L log n)

La complejidad espacial es de 2n.

Trie

La complejidad temporal del trie es de O(n*m) ya que esta depende de cuantas palabras contenga el trie y cuan largas sean estas, se toma el peor caso de n*m donde n es la longitud de la palabra más larga y n es el número de llaves dentro del trie.

La complejidad espacial es de O(n)

Ventajas del trie:

- Posee una búsqueda más eficaz, ya que la complejidad temporal es de O(n)
- El algoritmo de búsqueda del prefijo más largo se resuelve con mayor facilidad con este algoritmo
- No requiere tanto espacio para almacenar muchas cadenas pequeñas de palabras

Paso 5. Evaluación y Selección de la Mejor Solución

Criterio A. Conocimiento. La alternativa ha sido estudiada:
- [2] Sí (se prefiere una solución exacta)
- [1] No

Criterio B. Eficiencia temporal. Se prefiere una solución con mejor eficiencia que las otras consideradas. La eficiencia puede ser:

- [4] Constante
- [3] Mayor a constante
- [2] Logarítmica
- [1] Lineal

Criterio C. Eficiencia espacial. Se prefiere una solución con mejor eficiencia que las otras consideradas. La eficiencia puede ser:

- [4] Constante
- [3] Mayor a constante
- [2] Logarítmica
- [1] Lineal

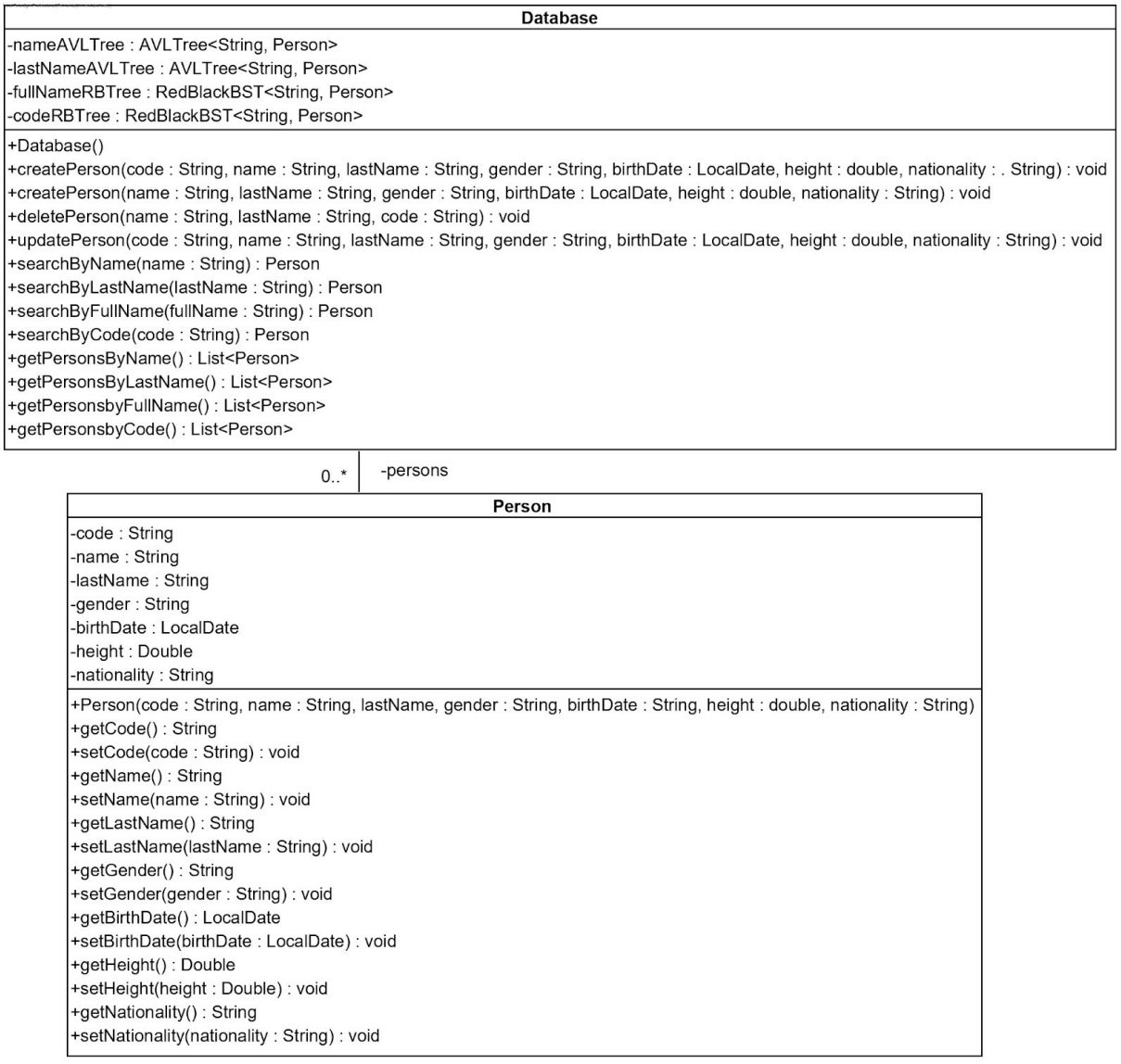
Criterio D. Facilidad en implementación algorítmica:

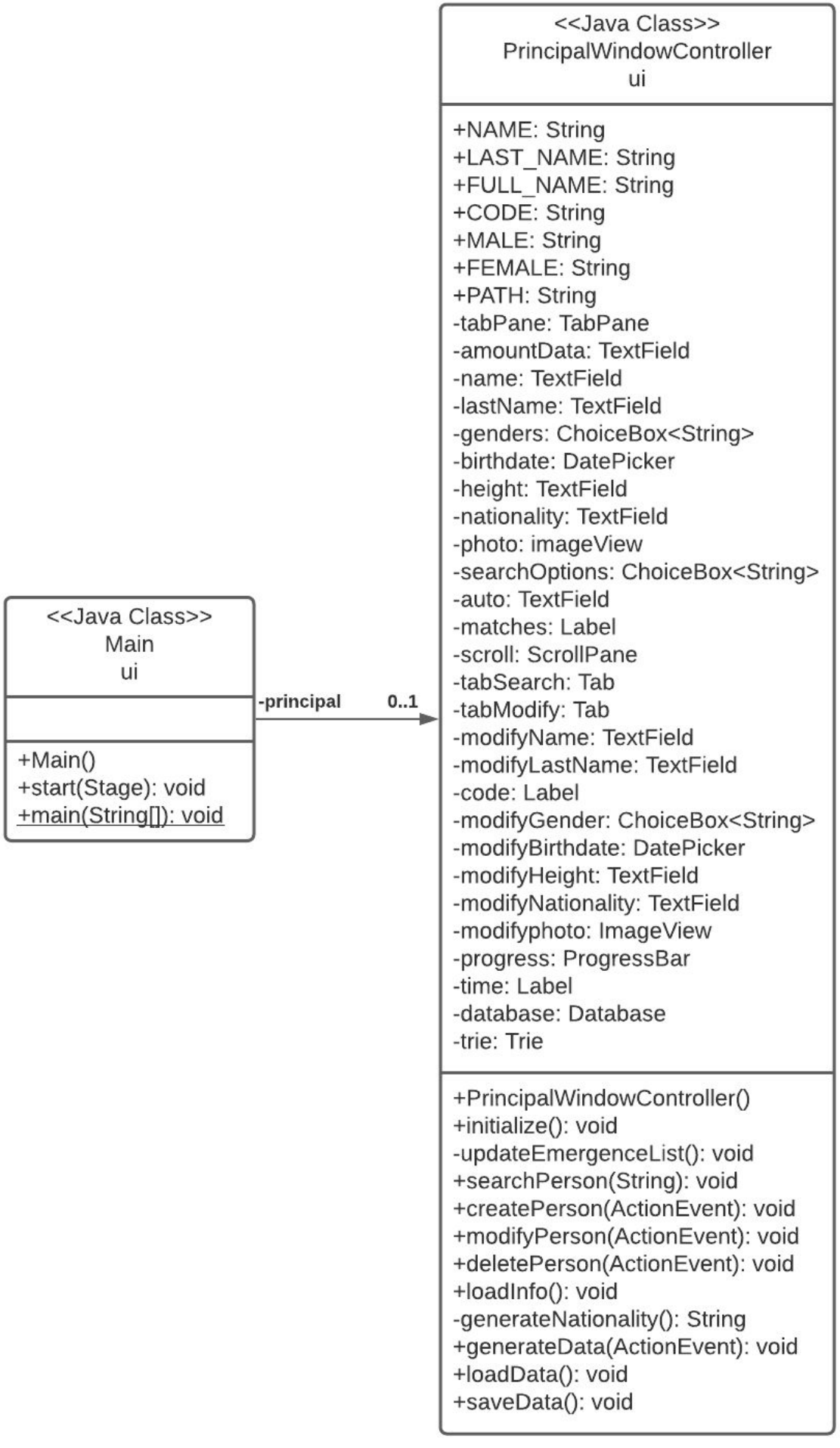
- [2] Compatible con las operaciones aritméticas básicas de un equipo de cómputo moderno
- [1] No compatible completamente con las operaciones aritméticas básicas de un equipo de cómputo moderno.

	Criterio A	**Criterio B**	**Criterio C**	**Criterio D**	**Total**
Árbol AVL	Sí(2)	Logarítmica(2)	Lineal(1)	Compatible (2)	7
Árbol rojinegro	No(1)	Logarítmica(2)	Lineal(1)	Compatible (2)	6
Trie	Sí(2)	Lineal(1)	Lineal(1)	Compatible (2)	6

Como todas las soluciones son prácticamente igual de útiles para resolver el problema es posible implementarlas de manera indistinta. También se puede utilizar todas a la vez.

Paso 6: Preparación de informes y especificaciones


Database
-nameAVLTree : AVLTree<String, Person>
-lastNameAVLTree : AVLTree<String, Person>
-fullNameRBTree : RedBlackBST<String, Person>
-codeRBTree : RedBlackBST<String, Person>
+Database()
+createPerson(code : String, name : String, lastName : String, gender : String, birthDate : LocalDate, height : double, nationality : . String) : void
+createPerson(name : String, lastName : String, gender : String, birthDate : LocalDate, height : double, nationality : String) : void
+deletePerson(name : String, lastName : String, code : String) : void
+updatePerson(code : String, name : String, lastName : String, gender : String, birthDate : LocalDate, height : double, nationality : String) : void
+searchByName(name : String) : Person
+searchByLastName(lastName : String) : Person
+searchByFullName(fullName : String) : Person
+searchByCode(code : String) : Person
+getPersonsByName() : List<Person>
+getPersonsByLastName() : List<Person>
+getPersonsbyFullName() : List<Person>
+getPersonsbyCode() : List<Person>
0..*
-persons
Person
-code : String
-name : String
-lastName : String
-gender : String
-birthDate : LocalDate
-height : Double
-nationality : String
+Person(code : String, name : String, lastName, gender : String, birthDate : String, height : double, nationality : String)
+getCode() : String
+setCode(code : String) : void
+getName() : String
+setName(name : String) : void
+getLastName() : String
+setLastName(lastName : String) : void
+getGender() : String
+setGender(gender : String) : void
+getBirthDate() : LocalDate
+setBirthDate(birthDate : LocalDate) : void
+getHeight() : Double
+setHeight(height : Double) : void
+getNationality() : String
+setNationality(nationality : String) : void


<<Java Class>>
PrincipalWindowController
ui
+NAME: String
+LAST_NAME: String
+FULL_NAME: String
+CODE: String
+MALE: String
+FEMALE: String
+PATH: String
-tabPane: TabPane
-amountData: TextField
-name: TextField
-lastName: TextField
-genders: ChoiceBox<String>
-birthdate: DatePicker
-height: TextField
-nationality: TextField
-photo: imageView
-searchOptions: ChoiceBox<String>
-auto: TextField
-matches: Label
-scroll: ScrollPane
-tabSearch: Tab
-tabModify: Tab
-modifyName: TextField
-modifyLastName: TextField
-code: Label
-modifyGender: ChoiceBox<String>
-modifyBirthdate: DatePicker
-modifyHeight: TextField
-modifyNationality: TextField
-modifyphoto: ImageView
-progress: ProgressBar
-time: Label
-database: Database
-trie: Trie
+PrincipalWindowController()
+initialize(): void
-updateEmergenceList(): void
+searchPerson(String): void
+createPerson(ActionEvent): void
+modifyPerson(ActionEvent): void
+deletePerson(ActionEvent): void
+loadInfo(): void
-generateNationality(): String
+generateData(ActionEvent): void
+loadData(): void
+saveData(): void
-principal
0..1
<<Java Class>>
Main
ui
+Main()
+start(Stage): void
+main(String[]): void

Paso 7: Implementación del diseño

Se implementará con Java orientada a objetos.

Tareas a implementar:

a. Generar personas
b. Mostrar barra de progreso
c. Guardar datos generados
d. Actualizar persona
e. Agregar persona
f. Eliminar persona
g. Buscar persona
h. Mostrar lista emergente

SUBRUTINAS:

Nombre	Generar personas
Resumen	Se genera una cantidad máxima de mil millones de personas de manera aleatoria para la base de datos.
Entradas	código, nombre, apellido, sexo, fecha de nacimiento, estatura, nacionalidad y fotografía
Salidas	La cantidad deseada de personas generadas.

```
public void loadInfo() throws IOException{

		File file = new File("data\\name.txt");
		BufferedReader br = new BufferedReader(new FileReader(file));
		String name = br.readLine();
		int amount = Integer.parseInt(amountData.getText());
		int total = 0;

		while(name!=null) {

			File file2 = new File("data\\lastname.txt");
			BufferedReader br2 = new BufferedReader(new FileReader(file2));
			String lastName = br2.readLine();

			while(lastName!=null && total <= amount) {
				if(total%1000==0) System.out.println(total+" "+name+"
"+lastName);
```

```
                    database.createPerson(name, lastName,
generateNationality());
                    lastName = br2.readLine();
                    total++;
                }
                br2.close();
                name = br.readLine();
            }
            br.close();
        }
```

Nombre	Mostrar barra de progreso
Resumen	Se muestra una barra de progreso con la duración de la generación de datos dentro del programa
Entradas	
Salidas	Muestra en pantalla la barra de progreso y al finalizar muestra el tiempo tardado en generar los datos

```
@FXML
        public void generateData(ActionEvent event) {
                time.setText(null);
                double timePassed = System.currentTimeMillis();
                Task<Void> task = new Task<Void>() {
                        @Override
                        public Void call() throws ClassNotFoundException, IOException {
                                progress.setVisible(true);
                                try {
                                        loadInfo();
                                }
                                catch (Exception e) {
                                        e.printStackTrace();
                                }

progress.setMaxHeight(System.currentTimeMillis()-timePassed);
                                return null;
                        }
                };
                progress.progressProperty().bind(task.progressProperty());
                task.setOnSucceeded(new EventHandler<WorkerStateEvent>() {

                        @Override
```

```
                public void handle(WorkerStateEvent arg0) {
                        progress.setVisible(false);
                        time.setText(progress.getMaxHeight()/1000+" sec");
                }
        });
        Thread loadingThread = new Thread(task);
        loadingThread.start();
    }
```

Nombre	Guardar datos generados
Resumen	Guarda los datos generados por el programa en un archivo serializable
Entradas	
Salidas	Un archivo .txt que contiene los datos generados previamente

```
@FXML
    public void saveData() {
        try {
                FileOutputStream fileOut = new
FileOutputStream("data\\database.txt", false);
                ObjectOutputStream out = new ObjectOutputStream(fileOut);
                out.writeObject(database);
                out.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
    }
```

Nombre	Actualizar persona
Resumen	Actualiza a una persona seleccionada con los nuevos datos deseados por la persona
Entradas	código, nombre, apellido, sexo, fecha de nacimiento, estatura, nacionalidad
Salidas	La persona modificada con los nuevos datos ingresados.

```
public void updatePerson(String code, String name, String lastName, String gender,
LocalDate birthDate, Double height, String nationality) {
```

```
                Person person = codeRBTree.search(code).getV();
                fullNameRBTree.search(person.getName()+"
"+person.getLastName()).setK(name+" "+lastName);;
                person.setName(name);
                person.setLastName(lastName);
                person.setGender(gender);
                person.setBirthDate(birthDate);
                person.setHeight(height);
                person.setNationality(nationality);
        }
```

Nombre	Agregar persona
Resumen	Se agrega a una persona manualmente con los respectivos datos a ingresar para ser creado y guardado en la base de datos
Entradas	nombre, apellido, sexo, fecha de nacimiento, estatura, nacionalidad
Salidas	La persona se agrega y se guarda en cada uno de los árboles para manejar su información

```
public void createPerson(String name, String lastName, String gender, LocalDate birthDate,
Double height,
                        String nationality) throws IllegalArgumentException {
                String code = UUID.randomUUID().toString();
                Person person = new Person(code, name, lastName, gender, birthDate,
height, nationality);
                nameAVLTree.addNode(name, person);
                lastNameAVLTree.addNode(lastName, person);
                fullNameRBTree.insertRB(name+" "+lastName, person);
                codeRBTree.insertRB(code, person);
        }
```

Nombre	Eliminar persona
Resumen	Se elimina una persona dentro de la base de datos.
Entradas	código, nombre, apellido

Salidas	La persona indicada se elimina de la base de datos de personas.

```
public void deletePerson(String name, String lastName, String code) {
		nameAVLTree.deleteNode(name);
		lastNameAVLTree.deleteNode(lastName);
		fullNameRBTree.deleteRB(name+" "+lastName);
		codeRBTree.deleteRB(code);
	}
```

Nombre	Buscar persona
Resumen	Se puede buscar a una persona de acuerdo a alguno de los 4 criterios de búsqueda: nombre, apellido, nombre completo o código
Entradas	criterio de búsqueda
Salidas	Se busca a la persona deseada y en caso de encontrarla se muestran los datos de esta de acuerdo al criterio seleccionado.

```
public Person searchByName(String name) {
		return nameAVLTree.searchNode(name).getV();
	}

	public Person searchByLastName(String lastName) {
		return lastNameAVLTree.searchNode(lastName).getV();
	}

	public Person searchByFullName(String fullName) {
		return fullNameRBTree.search(fullName).getV();
	}

	public Person searchByCode(String code) {
		return codeRBTree.search(code).getV();
	}
```

Nombre	Mostrar lista emergente
Resumen	Se muestra una lista emergente con un máximo de 100 personas con el mismo carácter ingresado por el usuario.

Entradas	criterio de búsqueda
Salidas	Un menú con la lista de personas que coinciden actualmente con las letras ingresadas.

```
public List<String> autocomplete(String word){
		TrieNode aux= root;
		for (int i = 0; i < word.length(); i++) {
			aux= aux.getChild(word.charAt(i));
			if(aux== null) {
				return new ArrayList<>();
			}
		}
		return aux.getWords();
	}

auto.textProperty().addListener(new ChangeListener<String>() {
			@Override
			public void changed(ObservableValue<? extends String> observable,
String oldValue, String newValue) {
				scroll.setVisible(true);
				String entry = auto.getText();
				GridPane gridPane = new GridPane();
				if (entry.length()==0) {
					gridPane.getChildren().clear();
					scroll.setContent(gridPane);
					scroll.setVisible(false);
				} else {
					List<String> data= trie.autocomplete(entry);
					gridPane.getChildren().clear();
					scroll.setContent(gridPane);
					matches.setText("("+data.size()+") results");
					if(data.size()<=100) {
						for (int i = 0; i < data.size(); i++) {
							Label label = new Label(data.get(i));
							gridPane.add(label, 1, i);
							if(data.size()<=20) {
								Button edit = new Button("edit");
								gridPane.add(edit, 2, i);
								edit.setOnAction(new
EventHandler<ActionEvent>() {
@Override
public void
handle(ActionEvent arg0) {
auto.setText(null);
```

```
gridPane.getChildren().clear();

scroll.setContent(gridPane);

matches.setText(null);

tabPane.getSelectionModel().select(tabModify);

tabModify.setDisable(false);

searchPerson(label.getText());
                                                                    }
                                                                });
                                                            }
                                                            scroll.setContent(gridPane);
                                                        }
                                                    }
                                                }
                                            }
                                        });
```