

**MÉTODO DE LA INGENIERÍA**

**FASE 1: IDENTIFICACIÓN DEL PROBLEMA**

* **Descripción del contexto problemático (causas y síntomas):**

Debido a la cantidad de datos que las bases de datos deben almacenar, y la complejidad tan alta que podrían llegar a tener la solución a problemas como buscar, modificar o eliminar información, se han desarrollado programas denominados *database management system* o DBMS, SGBD en español.

* **Identificación del problema:**

se solicita almacenar datos de forma estructurada y de una manera que permita el acceso y la modificación de estos, puede llegar a tener una complejidad temporal y/o espacial muy altas. En efecto, como se pueden almacenar millones de datos, algoritmos que usen técnicas iterativas, serían completamente ineficientes a causa de la cantidad tan gran de información que contendría la tabla.   
Es por eso que se necesita encontrar una manera eficiente de realizar las operaciones de consulta, modificación, eliminación y agregación de datos.

* **Requerimientos funcionales y no funcionales**

**Requerimientos funcionales:**

* **RF1: Búsquedas sobre la tabla utilizando criterios de búsqueda.**

El programa debe estar en capacidad de buscar datos utilizando como criterio para la búsqueda uno de los campos que almacena la tabla.   
Por ejemplo, si la base de datos almacena información sobre estudiantes como: nombre, edad, carrera, número de estudiante, cédula, promedio. Entonces, el software debe permitir buscar a un estudiante con respecto a su cédula o su nombre, etc...

* **RF2: Búsquedas muy rápidas para tres de los campos posibles con índices.**

El programa debe estar en capacidad de efectuar búsqueda para cada uno de los índices de manera muy rápida.   
Por ejemplo, si los campos que almacena la base de datos son por ejemplo: nombre, edad, carrera, número de estudiante, cédula, promedio, entonces tres de estos campos deben ser los índices. Siguiendo con el ejemplo, los índices podrían ser: nombre, carrera, cédula.   
Es decir, la complejidad temporal para consultar estos tres índices podría ser O(n log n), mientras que para los otros campos podría ser O(n).

* **RF3: Empleo de árboles binarios de búsqueda balanceados para almacenar los datos.**

El programa debe estar en capacidad de almacenar la información en árboles binarios de búsqueda balanceados (rojinegros o AVL) para cada uno de los tres índices. Esto con el fin de poder buscar con respecto a esta información con un tiempo de O(log n).  
Por ejemplo, si se almacenan los datos de estudiantes como: nombre, edad, carrera, número de estudiante, cédula, promedio, y que los índices son nombre, edad y carrera. Entonces, deben existir tres árboles binarios de búsqueda balanceados que usen nombre, edad y carrera como clave, respectivamente, y como valor el el objeto que representará a una persona.

* **RF4: Posibilidad de escoger los campos de la tabla.**El programa debe estar en capacidad de permitirle al usuario escoger qué campos se quieren almacenar en la base de datos. En efecto, el software no debe solo leer una tabla como la de los ejemplos anteriores, si no que el usuario podría comunicarle al programa qué campos desea. Por ejemplo: nombre, apellido, edad, dirección de residencia, nacionalidad, número de pasaporte, fecha de llegada al país. Estos son datos que, a diferencia del ejemplo usado anteriormente de estudiantes, se podrían querer guardar en una base de datos de todos los extranjeros que residen en una ciudad del país.

**Requerimientos no funcionales:**

* **RNF1: Capacidad de almacenar muchos datos.**El programa debe estar en capacidad de almacenar una capacidad muy grande de datos, millones, por ejemplo. Es decir, se pueden guardar prácticamente cuantos datos se deseen. m
* **RNF2: Realizar consultas de manera eficiente (en cuanto a tiempo):**

**FASE 2: RECOPILACIÓN DE LA INFORMACIÓN NECESARIA**

* **Marco teórico**

**BASES DE DATOS:**

Una base de datos, como cualquier estructura de datos, puede ser clasificada según diferentes criterios.

Se puede clasificar con respecto a la **variabilidad** de la base de datos. Puede ser estática, se usa únicamente para la lectura de datos, o dinámica, su información es modificada a lo largo del tiempo.   
Igualmente se puede clasificar por su **contenido**. Existen bases de datos bibliográficas, contiene información sobre el autor, fecha de publicación, editorial, título, edición, de una determinada publicación, etc. Así mismo hay bases de datos de texto completo, que contienen por ejemplo la totalidad de un libro (a diferencia de la BDD bibliográfica).  
La más usada tal vez es la base de datos como directorio, puede ser personal o empresarial (páginas amarillas), contiene datos básicos como nombre, edad, teléfono, etc.   
  
De la misma forma, existe una variedad de modelos de bases de datos, con los que también se pueden calificar las BDD. Un modelo de datos es en general una descripción de algo conocido como contenedor de datos. Estos modelos se usan para implementar eficiente y eficazmente una BDD, normalmente se refieren a algoritmos o conceptos matemáticos.

Existen las **BDD jerárquicas**, en las que los datos son organizados en forma estructura de árbol binario y permite crear estructuras estables y de gran rendimiento. Su estructura es como se muestra en la figura 1. Cada nodo tiene **un** hijo derecho y **un** izquierdo al igual que **un** padre. Por ejemplo en este caso, el nodo C tiene como hijo izquierda a F y derecho a G, y como padre tiene a A.

El problema más grande al que se enfrenta este tipo de BDD es a la redundancia de los datos.  
  
Además existen las **BDD de red**, en las que la información también es almacenada en árboles binarios, con la diferencia de que en este caso un nodo tiene **al menos** un padre. Este modelo ofrece una solución muy eficiente al problema de la redundancia de los datos. No obstante este es un modelo más usado por programadores que por usuarios finales.

Adicionalmente existen **BDD transaccionales**. Cuyo propósito es, como su nombre lo indica, la transacción lo más rápida posible de datos (recepción y envíos). Un ejemplo claro es la transacción de dinero entre cuentas bancarias.

Continuando, se usan **BDD relacionales**. Estos, se utilizan en la actualidad para representar problemas reales y gestionar datos dinámicamente. Propuesto inicialmente en 1970 por Edgar Frank Codd, fue rápidamente considerado otro tipo de BDD. Como su nombre lo indica, se fundamenta en usar relaciones, que se pueden representar de forma lógica como tuplas. En esta clase de BDD no importa ni el lugar ni la forma en que se almacenan los datos. El lenguaje más habitual para construir las consultas a bases de datos relacionales es SQL (Structured Query Language).

Para crear aplicaciones muy concretas se utilizan a veces las **BDD multidimensionales**. Estas permiten lo mismo que las BDD relacionales, la diferencia está más bien a un nivel conceptual.   
Otro tipo de BDD,son las **documentales** (se usan para almacenar cantidades enormes de datos generalmente con antecedentes históricos), y las **BDD deductivas,** permiten realizar deducciones a través de inferencias, también se les llaman BDD lógicas.

**ARBOLES BINARIOS:**

Un árbol binario es una estructura de datos para almacenar información de forma estructurada y eficiente. Cada nodo del árbol, tiene un hijo derecho y un izquierdo, sólo puede tener máximo dos hijos (de ahí el nombre binario), al igual que solo puede tener un padre. La figura 1 muestra un ejemplo de árbol binario.   
Continuando, existen diferentes tipos de datos, cada uno con propiedades únicas:

* **Árbol binario de búsqueda (ABB):**Además de las propiedades que tiene un árbol binario, el ABB debe cumplir rotundamente la siguiente propiedad: Cada nodo en el árbol es mayor que los vértices de su subárbol izquierdo y menor que los vértices de su subárbol derecho. Como se observa en la figura dos, todos los nodos del subárbol derecho de la raíz (8), son mayores que él, y todos los nodos del subárbol izquierdo son menores que él, y así para todos los nodos. Por esta razón la operación de búsqueda es sumamente eficiente.
* **Árbol rojinegro:**

Esta clase de árbol, cumple todas las condiciones de un ABB, no obstante se considera también como un árbol balanceado debido a que siempre debe cumplir con las siguientes reglas:todo nodo es rojo o negro, la raíz es negra, toda hoja nil es negra, si un nodo es rojo, entonces sus hijos son negros, cada camino de un nodo a sus hojas descendientes contienen el mismo número de nodos negros. Estas condiciones hacen que las operaciones de agregación y eliminación sean más complejas. Para conservar las propiedades que debe cumplir todo árbol rojinegro, en ciertos casos de la agregación y la eliminación será necesario reestructurar el árbol, aunque no debe perderse la ordenación relativa de los nodos. Para ello, se llevan a cabo una o varias rotaciones, que no son más que reestructuraciones en las relaciones padre-hijo-tío-nieto. Además, para estos árboles las hojas son completamente irrelevantes, dicha razón por la cual no contienen nada (de ahí el nombre nil). Lo que los hace también muy valiosos es que ofrecen un peor caso con tiempo garantizado para la inserción, el borrado y la búsqueda.

* **Árbol AVL:**

Los árboles AVL 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.

* **Tipo abstracto de dato (TAD):**

Es la conjunción de variables, operaciones y aserciones Tipo Abstracto de Datos (además de documentación) que modela un dominio de datos.

* + 1. *Grupo de operaciones primitivas de un TAD*

1. Constructoras: Encargadas de crear las estructuras internas del tipo abstracto de datos.
2. Modificadoras: Son aquellas operaciones que alteran el estado de los elementos del TAD.
3. Analizadoras: Operaciones que consultan el estado de los elementos y retornan información (no cambian los estados).
4. Destructoras: Son operaciones que eliminan por completo los elementos del objeto del tipo del TAD. Luego de ejecutar operaciones destructoras los objetos no pueden volver a utilizarse.
5. Persistencia: Con ellas se puede guardar en un dispositivo de memoria secundaria (disco duro, CD/DVD, USB, entre otros) la información de los objetos.

**FASE 3: BÚSQUEDA DE SOLUCIONES CREATIVAS**

* **Generación de ideas**

Por medio de la lluvia de de ideas que consiste en anotar distintas ideas surgidas de manera no sistematizada, para que después puedan analizarse y filtrarse hasta seleccionar una de ellas.

Las ideas presentadas a continuación, son el resultado de la implementación de la técnica de generación de ideas:

Alternativas de estructuras de datos para el almacenamiento de información de gran tamaño:

1. **Estructuras de datos lineales**
   1. Arreglo
   2. ArrayList
   3. LinkedList
2. **Estructuras de datos recursivas**
   1. Árbol Binario de búsqueda
   2. Árbol Rojo-Negro
   3. Árbol AVL

**FASE 4: TRANSICIÓN DE LA FORMULACIÓN DE IDEAS A LOS DISEÑOS PRELIMINARES**

* **Descarte de ideas no viables**

|  |  |
| --- | --- |
| Idea no factible | Justificación |
| Estructura de datos lineal: LinkedList | La búsqueda de datos se haría en un tiempo O(n) por lo que es un tiempo que no nos funciona para el problema. |
| Estructura de datos lineal: Arreglo | La cantidad de datos que se desea manejar es muy grande y se desconoce en su totalidad la cantidad que se necesita. |
| Estructura de datos lineal: ArrayList | La cantidad de datos que se desea manejar es muy grande además la búsqueda no se realizará en tiempo O(1) sino en O(N) |

**FASE 5: EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN**

* **Criterios de evaluación**

Los criterios de evaluación son:

* Complejidad Temporal < O(n) para el método de búsqueda.
* **Evaluación según criterios**

La evaluación de las estructuras recursivas de Arbol AVL, Arbol RojoNegro, Arbol Binario de Búsqueda se tomo en cuenta que

El Arbol RojoNegro todos sus métodos tienen una complejidad temporal de búsqueda, agregar y eliminación son O(log(n)) por lo que tiene una complejidad temporal que funciona perfectamente con la solución.

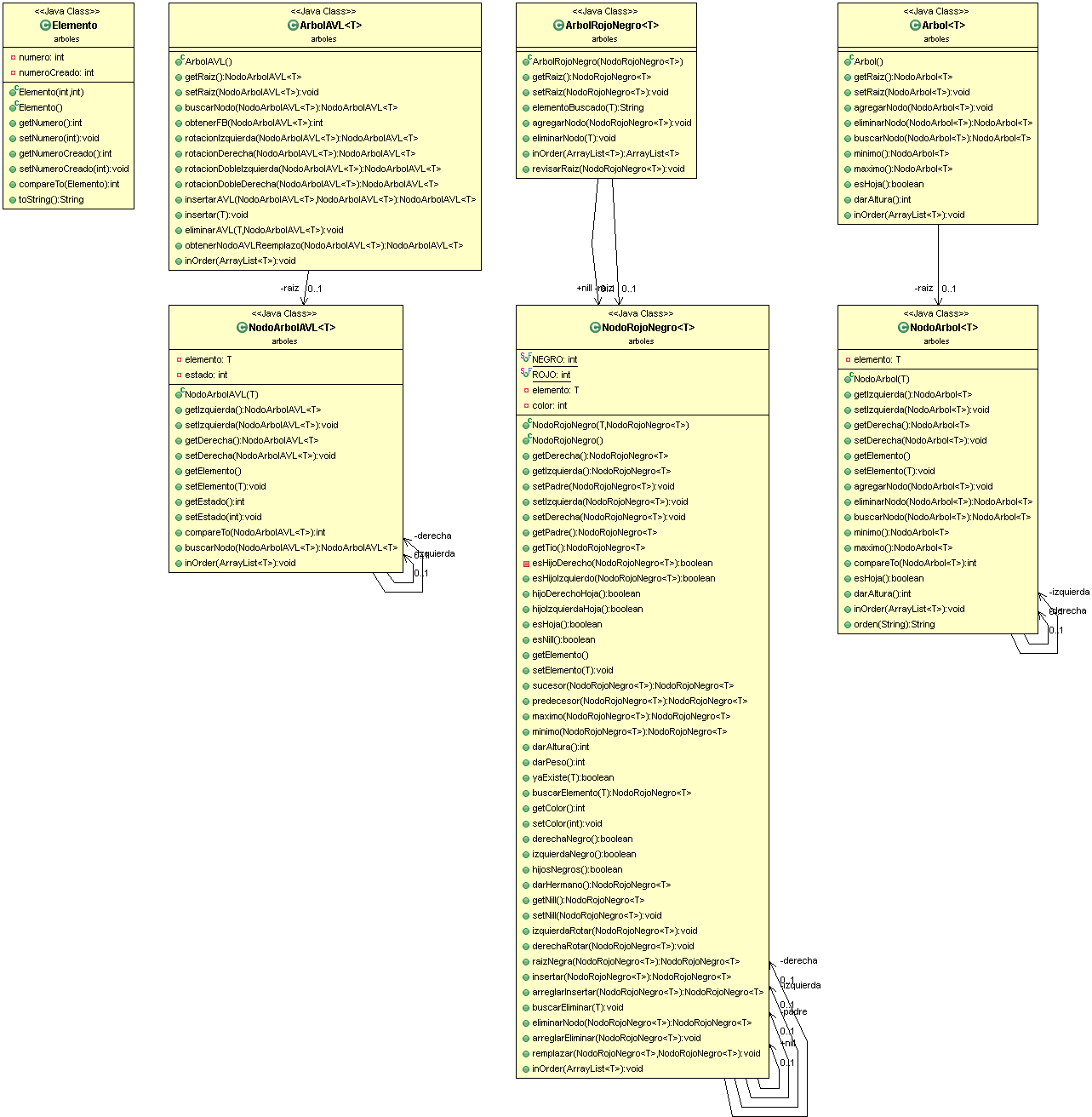
El Arbol AVL todos sus métodos tienen una complejidad temporal de búsqueda, agregar y eliminación son O(log(n)) por lo que tiene una complejidad temporal que funciona perfectamente con la solución.

El Arbol ABB todos sus métodos tienen una complejidad temporal de búsqueda, agregar y eliminación son O(n) pero este puede reducirse dependiendo de cómo se logre acomodar los datos que se ingresen y estos pueden reducirse a una complejidad cercana a O(log(n)) pero todo depende del ingreso de los datos.

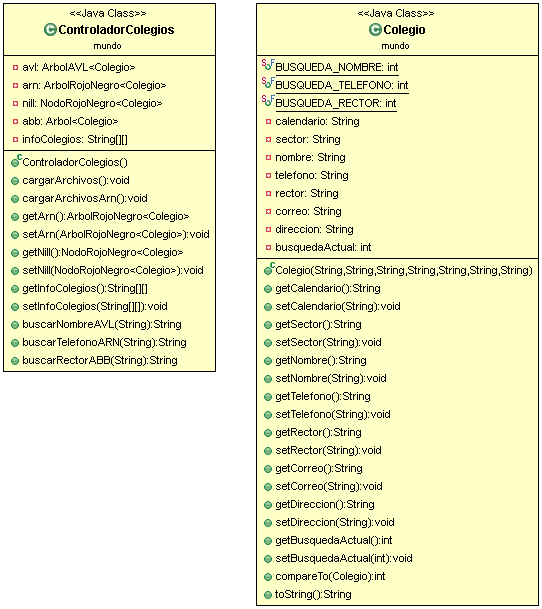
Por lo que para tener la mejor solución se utilizaran los arboles AVL,Rojinegros y se dará la ooportunidad a un Arbol ABB al organizar la forma de ingreso de los datos.

**FASE 6: EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN**

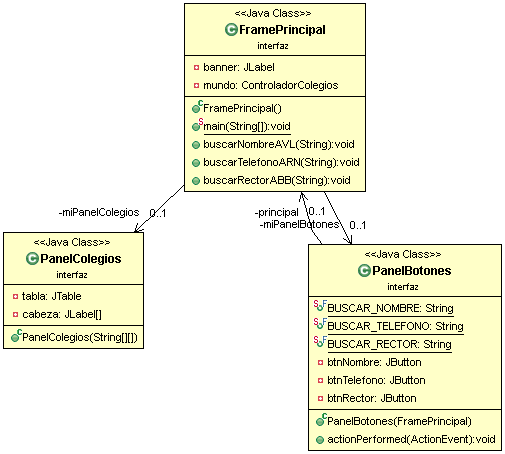
* **Diseño del diagrama de clases de la solución**

** Paquete Estructuras de Arboles**

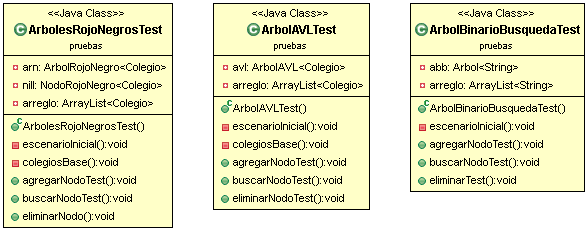
**Paquete mundo**



**Paquete Interfaz**

****

**Paquete Pruebas**



* **Diseño de casos de las pruebas unitarias**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo:** Probar que el método permite agregar nodos al árbol | | | | |
| **Clase:** ABB,AVL,ARN | | **Método:** Agregar() | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | Probar que el método permite agregar a la raiz del arbol. | Escenario 2 | K = 5  V = “daniela” | Permite agregar “daniela” en la raíz, dado que no hay más elementos en el árbol y la raíz ya no es nula |
| **2** | Probar que el método permite agregar al hijo izquierdo, preservando la condición de orden. | Escenario 4 | K = 5  V = “sofia”  K = 2  V = “angelica” | Permite agregar “sofia” como raiz del arbol y a “angelica” como hijo izquierdo de “sofia”, comparandolos con su key y así respetando la condición de orden. |
| **3** | Probar que el método permite agregar al hijo derecho, preservando la condición de orden. | Escenario 3 | K = 5  V = “daybed”  K = 10  V = “laura” | Permite agregar a “daybed” como raiz del arbol y a “laura” como hijo derecho de “Jose” |
| **4** | Probar que el método permite agregar la raíz, el hijo derecho y el hijo izquierdo de ella. | Escenario 5 | K = 5  V = “geral”  K = 2  V = “manuel”  K = 7  V = “carlos” | Permite agregar a “geral” como raiz del arbol, a “manuel” como el hijo izquierdo de “geral” y a “carlos” como hijo derecho de “aleja”. También se respeta la condición de orden. |
| **5** | Probar que el método agregó correctamente la raíz del árbol. | Escenario 5 | V = “aleja” | “Aleja” coincide con el valor de la raíz del árbol |
| **6** | Probar que el método agregó al hijo mayor del árbol. | Escenario 1 | V = “Alejandra” | “Alejandra” coincide con el valor del hijo mayor del árbol. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo:** Probar el método buscar nodos del árbol. | | | | |
| **Clase:** ABB,AVL,ARN | | **Método:** Buscar () | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | Probar que el método permite buscar un nodo, teniendo su clave. | Escenario 3 | Key = 8 | El nodo con valor 8 coincide con un nodo del árbol. |
| **2** | Probar que el método no permite buscar un nodo, donde su clave no existe en el a´rbol. | Escenario 1 | Key = 20 | El nodo con valor 20 no existe en el árbol, por lo tanto el método retorna null. |
| **3** | Probar que el método permite buscar al hijo menor del árbol | Escenario 5 | Key = 1 | Permite buscar el nodo con clave 1 y retorna el nodo. |
| **4** | Probar que el método permite buscar al hijo mayor del árbol. | Escenario 5 | K = 7 | Permite buscar el nodo con clave 7 y retorna el nodo. |
| **5** | Probar que el método permite buscar un nodo, dependiendo de la clave que reciba | Escenario 1  Escenario 2 | K = 1  K = 34  K = 6 | Permite buscar el nodo con las claves recibidas. |
| **6** | Probar que el método permite buscar la raíz del árbol. | Escenario 5 | K = 23 | Busca el nodo con la clave 23. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo:** Probar que el método permite eliminar nodos del árbol | | | | |
| **Clase:** ABB,AVL,ARN | | **Método:** Eliminar () | | |
| **Caso #** | **Descripción de la prueba** | **Escenario** | **Valores de entrada** | **Resultado** |
| **1** | Probar que el método permite eliminar la raíz del árbol | Escenario 6 | K = 5 | El método elimina correctamente la raíz del árbol y en su posición asigna a el sucesor. |
| **2** | Probar que el método permite eliminar el hijo derecho de un nodo | Escenario 6 | K = 9 | El método elimina correctamente el hijo derecho de un nodo y en su lugar ubica a su sucesor. |
| **3** | Probar que el método permite eliminar el hijo izquierdo de un nodo | Escenario 6 | K = 7 | El método elimina correctamente el hijo izquierdo de un nodo y en su lugar ubica a su sucesor. |

* **Tablas TAD**

|  |
| --- |
| **TAD ABB** |
| X  Y  Z |
| *{inv: Y ≤ X ≤ Z}* |
| **Operaciones primitivas:**   * crearABB→ - →ABB * agregar→ABB x Elemento x Elemento → Elemento * eliminar→ABB x Elemento →boolean * buscar→ABB x Clave → Valor * buscarN → ABB x Clave → Elemento * sucesor → ABB x Elemento → Elemento |

|  |
| --- |
| **crearABB()** |
| *“Crear un nuevo árbol vacío ”* |
| {pre: - } |
| {pos: ABB ≠ Ø} |

|  |
| --- |
| **agregar()** |
| *“Agregar un elemento a un árbol ABB ”* |
| {pre: Debe de existir un ABB y el elemento a agregar no debe pertenecer al ABB } |
| {pos: Se agregó el elemento en el ABB con el orden establecido} |

|  |
| --- |
| **eliminar()** |
| *“Eliminar un elemento de un árbol ABB ”* |
| {pre: Debe de existir un ABB y el elemento a eliminar debe pertenecer al ABB } |
| {pos: Se eliminó el elemento en el ABB } |

|  |
| --- |
| **Buscar()** |
| *“Buscar un elemento en un árbol ABB ”* |
| {pre: Debe de existir un ABB } |
| {pos: Se encontró el elemento buscado y retorna el valor} |

|  |
| --- |
| **BuscarN()** |
| *“Buscar un elemento en un árbol ABB ”* |
| {pre: Debe de existir un ABB } |
| {pos: Se encontró el elemento buscado con su clave y retorna el elemento } |

|  |
| --- |
| **Sucesor()** |
| *“Encontrar el sucesor de un elemento del ABB ”* |
| {pre: Debe de existir un ABB } |
| {pos: Se encontró el sucesor del elemento } |

|  |
| --- |
| **TAD ABB AVL** |
| X  Y  Z |
| *{inv: Y ≤ X ≤ Z*  → |EL Factor de Balanceo| <= 1*}* |
| **Operaciones primitivas:**   * crearAVL→ - →AVL * agregar→AVLx Elemento x Elemento → AVL * eliminar→AVL x Elemento →boolean * buscarN → AVL x Clave → Elemento * reemplazo → AVL x Elemento → Elemento * obtenerFactorBalanceo → - → int * rotaciónIzquierda → AVL → AVL * rotacionDobleIzquierda → AVL → AVL * rotacionDerecha → AVL → AVL * rotacionDobleDerecha → AVL → AVL |

|  |
| --- |
| **crearAVL()** |
| *“Crear un nuevo árbol AVL vacío ”* |
| {pre: - } |
| {pos: AVL ≠ Ø} |

|  |
| --- |
| **agregar()** |
| *“Agregar un elemento a un árbol AVL ”* |
| {pre: Debe de existir un AVL y el elemento a agregar no debe pertenecer al AVL } |
| {pos: Se agregó el elemento en el AVL con el orden establecido, se actualizaron los factores de balanceo del arbol} |

|  |
| --- |
| **eliminar()** |
| *“Eliminar un elemento de un árbol AVL ”* |
| {pre: Debe de existir un AVL y el elemento a eliminar debe pertenecer al AVL } |
| {pos: Se eliminó el elemento en el AVL } |

|  |
| --- |
| **BuscarN()** |
| *“Buscar un elemento en un árbol AVL ”* |
| {pre: Debe de existir un AVL } |
| {pos: Se encontró el elemento buscado y retorna el valor} |

|  |
| --- |
| **Reemplazo()** |
| *“Encontrar el sucesor de un elemento del AVL ”* |
| {pre: Debe de existir un AVL } |
| {pos: Se encontró el sucesor del elemento } |

|  |
| --- |
| **obtenerFactorBalanceo()** |
| *“retorna el factor de balanceo de AVL ”* |
| {pre: Debe de existir un AVL } |
| {pos: Se retorna el valor que da la resta de la altura del sub árbol derecho – la altura del sub árbol izquierdo } |

|  |
| --- |
| **rotarIzquierda ()** |
| *“Rota el nodo que se encuentra desbalanceado hacia la izquierda para ceder ‘peso’ y reducir el factor de balanceo ”* |
| {pre: Debe de existir un AVL, el nodo a rotar debe estar desbalanceado (fb = 2) } |
| {pos: Se reestructura el sub árbol en base a la rotación efectuada en el nodo } |

|  |
| --- |
| **rotarDobleIzquierda ()** |
| *“Rota el nodo que se encuentra desbalanceado hacia la derecha y después a la izquierda para ceder ‘peso’ y reducir el factor de balanceo ”* |
| {pre: Debe de existir un AVL, el nodo a rotar debe estar desbalanceado (fb = 2) } |
| {pos: Se reestructura el sub árbol en base a la rotación efectuada en el nodo } |

|  |
| --- |
| **rotarDerecha ()** |
| *“Rota el nodo que se encuentra desbalanceado hacia la derecha para ceder ‘peso’ y reducir el factor de balanceo ”* |
| {pre: Debe de existir un AVL, el nodo a rotar debe estar desbalanceado (fb = -2) } |
| {pos: Se reestructura el sub árbol en base a la rotación efectuada en el nodo } |

|  |
| --- |
| **rotarDobleDerecha()** |
| *“Rota el nodo que se encuentra desbalanceado hacia la izquierda y después a la derecha para ceder ‘peso’ y reducir el factor de balanceo ”* |
| {pre: Debe de existir un AVL, el nodo a rotar debe estar desbalanceado (fb = -2) } |
| {pos: Se reestructura el sub árbol en base a la rotación efectuada en el nodo } |

|  |
| --- |
| **TAD ABB RojoNegro** |
| X  Z  Y |
| *{inv: Y ≤ X ≤ Z*  SI X es Rojo → Y es Negro y Z es Negro  Cada camino de un nodo n, a sus hojas descendientes tiene el mismo números de nodos negros*}* |
| **Operaciones primitivas:**   * crearARN→ - →ARN * agregar→ARNx Elemento x Elemento → ARN * eliminar→ARN x Elemento →ARN * buscarN → ARN x Clave → Elemento * Sucesor → ARN x Elemento → Elemento * darHermano → NodoARN→ NodoARN * arreglarInsertar → NodoARN → ARN * arreglarEliminar → NodoARN → ARN * rotaciónIzquierda → NodoARN → ARN * rotacionDerecha → NodoARN → ARN |

|  |
| --- |
| **crearARN()** |
| *“Crear un nuevo árbol ARN vacío ”* |
| {pre: - } |
| {pos: ARN ≠ Ø} |

|  |
| --- |
| **agregar()** |
| *“Agregar un elemento a un árbol ARN ”* |
| {pre: Debe de existir un ARN y el elemento a agregar no debe pertenecer al ARN } |
| {pos: Se agregó el elemento en el ARN con el orden establecido, se actualizaron los colores de los nodos del arbol} |

|  |
| --- |
| **eliminar()** |
| *“Eliminar un elemento de un árbol ARN ”* |
| {pre: Debe de existir un ARN y el elemento a eliminar debe pertenecer al ARN } |
| {pos: Se eliminó el elemento en el ARN } |

|  |
| --- |
| **BuscarN()** |
| *“Buscar un elemento en un árbol ARN ”* |
| {pre: Debe de existir un ARN } |
| {pos: Se encontró el elemento buscado y retorna el valor} |

|  |
| --- |
| **Sucesor()** |
| *“Encontrar el sucesor de un elemento del AVL ”* |
| {pre: Debe de existir un ARN } |
| {pos: Se encontró el sucesor del elemento } |

|  |
| --- |
| **darHermano()** |
| *“retorna el nodo Hermano del nodo actual”* |
| {pre: Debe de existir un ARN } |
| {pos: Se retorna el nodo izquierdo del padre del nodo actual si estamos en el nodo derecho o el nodo derecho del padre del nodo actual si estamos en un nodo izquierdo } |

|  |
| --- |
| **rotarIzquierda ()** |
| *“Rota el nodo que se encuentra desbalanceado hacia la izquierda para acomodar el árbol y este cumpla con las propiedades”* |
| {pre: Debe de existir un ARN, y el nodo a rotar debe estar ocasionando que se incumplas las propiedades } |
| {pos: Se reestructura el sub árbol en base a la rotación efectuada en el nodo } |

|  |
| --- |
| **rotarDerecha()** |
| *“Rota el nodo que se encuentra desbalanceado hacia la derecha para acomodar el árbol y este cumpla con las propiedades”* |
| {pre: Debe de existir un ARN, y el nodo a rotar debe estar ocasionando que se incumplas las propiedades } |
| {pos: Se reestructura el sub árbol en base a la rotación efectuada en el nodo } |

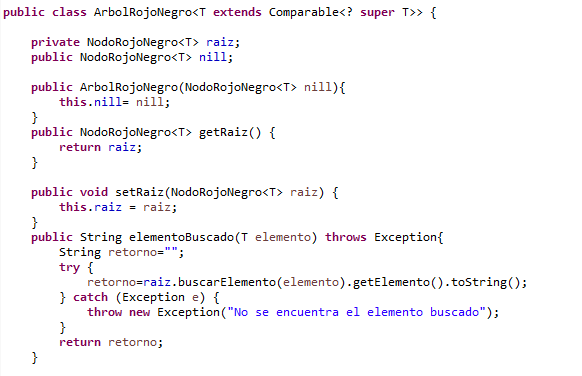
|  |
| --- |
| **ArreglarInsertar ()** |
| *“Ajusta el árbol ARN para que este después de una inserción se vuelva a balancear y siga cumpliendo con las propiedades de la estructura de datos”* |
| {pre: Debe de existir un ARN que no cumple con alguna de las propiedades} |
| {pos: Se reestructura el árbol ARN ajustando sus nodos para que cumplan las condiciones } |

|  |
| --- |
| **ArreglarEliminar()** |
| *“Ajusta el árbol ARN para que este después de una inserción se vuelva a balancear y siga cumpliendo con las propiedades de la estructura de datos”* |
| {pre: Debe de existir un ARN que no cumple con alguna de las propiedades} |
| {pos: Se reestructura el sub árbol en base a la rotación efectuada en el nodo } |

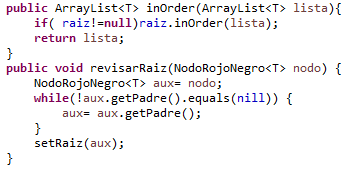
Fase 7: Implementación del Diseño

Al ya haber escogido las estructuras que se utilizarían y definir el diseño de estas en los modelos correspondientes se muestra la implementación del diseño en forma de código ya funcional.

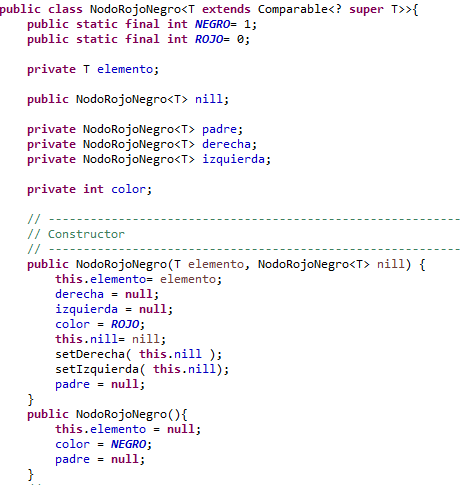
**ARBOL ROJONEGRO**







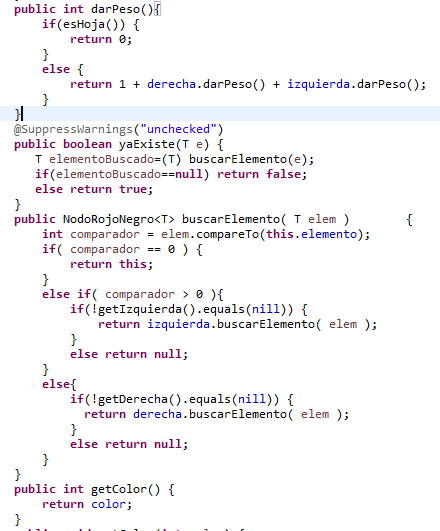
**Nodo RojoNegro**









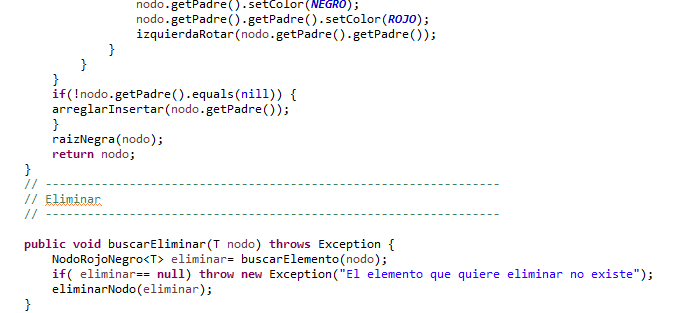








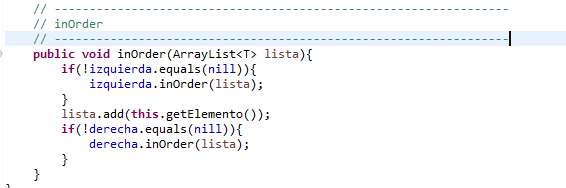






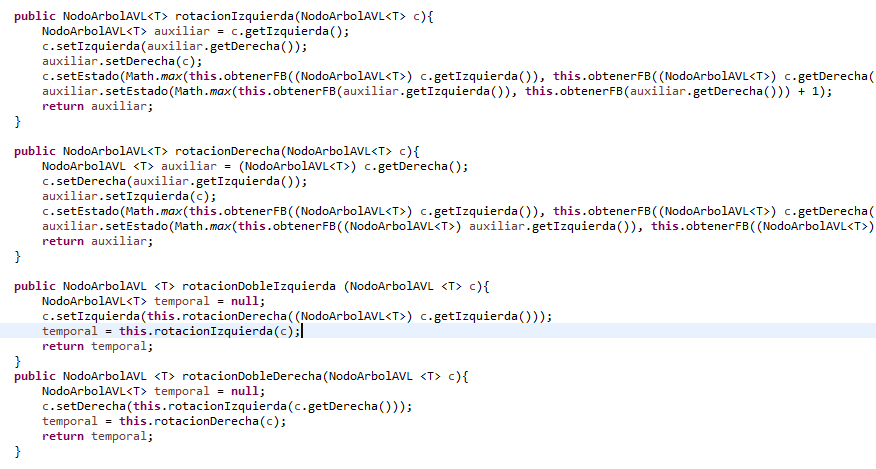




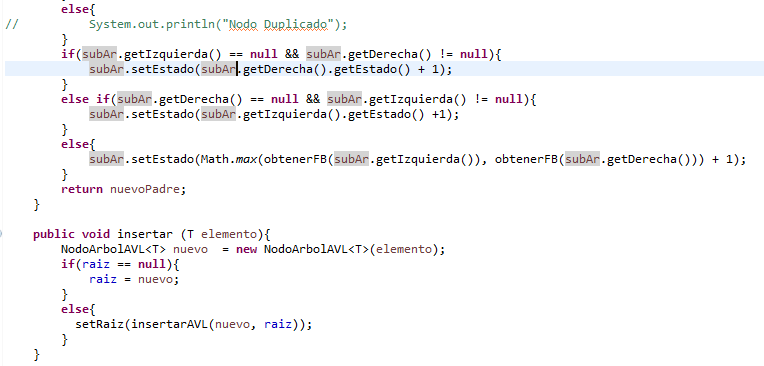


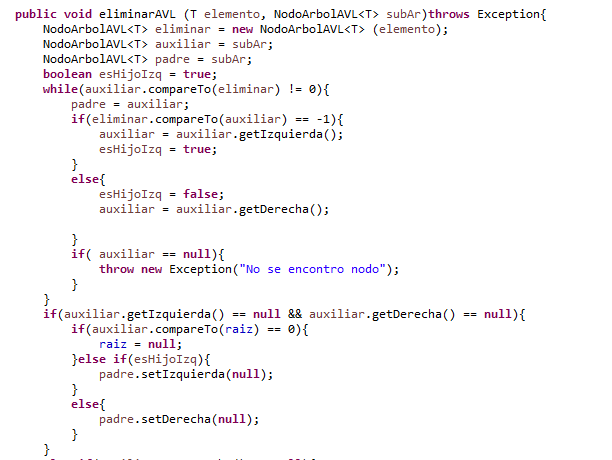
**ARBOL AVL**



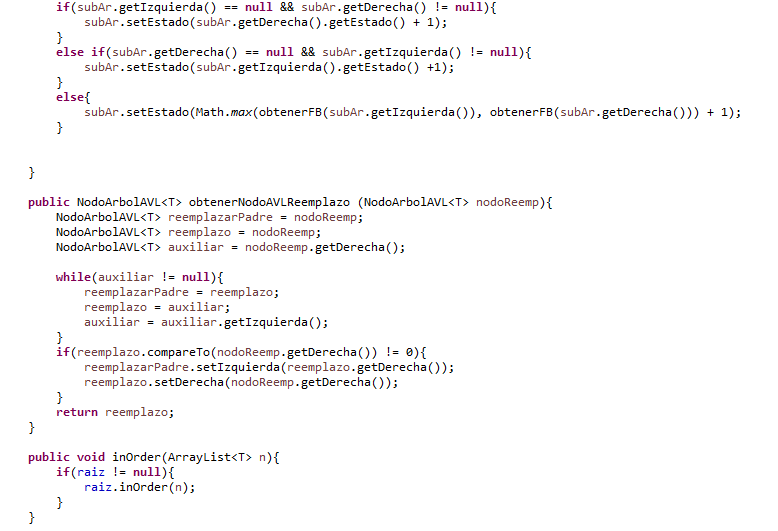




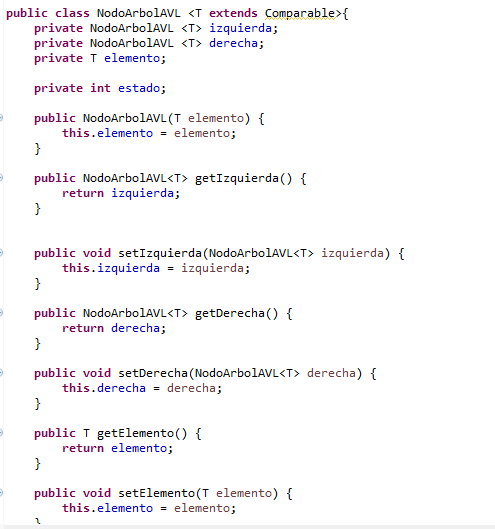


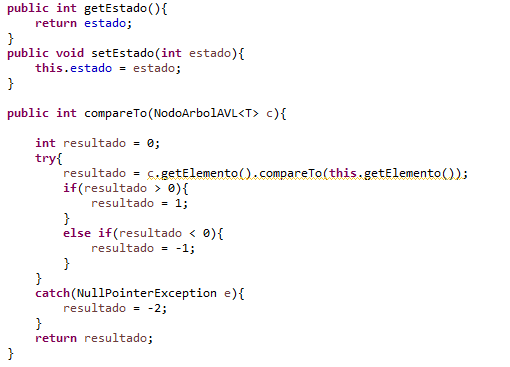






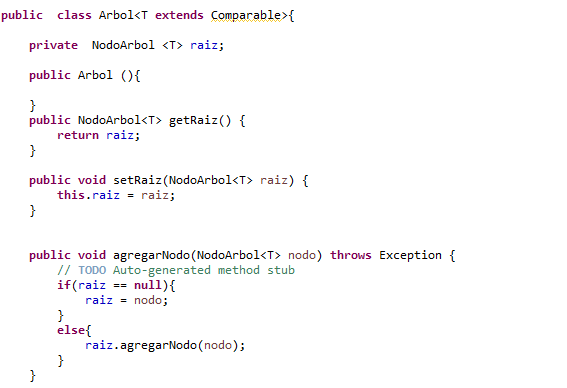
**NodoArbolAVL**

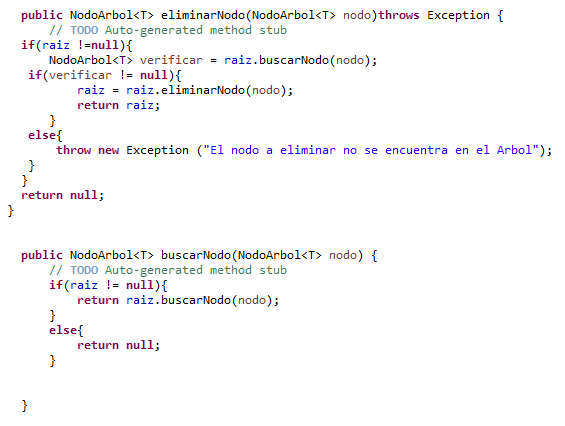


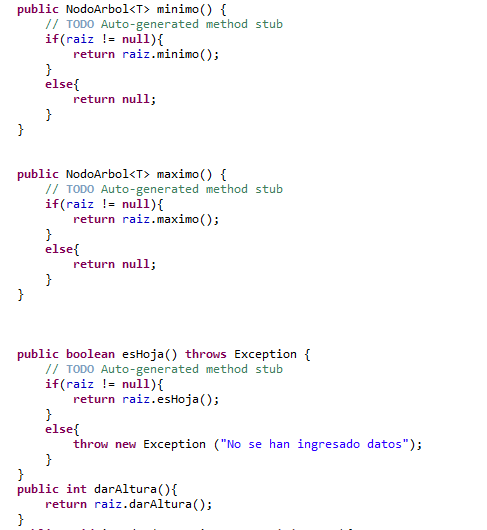




**Arbol Binario Busqueda**









**NODO ARBOL BINARIO BUSQUEDA**

