Laura Rubio A00346353

Douglas López A00347533

David Obando A00348505

Requerimientos:

**Nombre:** Buscar jugador por características o atributos.

Se encarga de buscar a jugador en base a algunos de los siguientes atributos:

* Rebotes por partido
* Asistencia por partido
* Robos por partido
* Bloques por partido

**Entrada:** Dato a buscar (double o entero).

**Salida:** Devuelve el jugador que tenga el dato buscado. Si son varios se retornan todos los que tengas ese mismo dato.

**Nombre:** Operaciones principales del aplicativo.

El programa debe estar en la capacidad de permitir ingresar, suprimir o modificar un jugador, a través de una interfaz gráfica.

**Entrada:** Jugador

**Salida:** Jugador insertado o jugador suprimido o jugador modificado.

**Nombre:** Guardar información en memoria secundaria.

Puede ocurrir que nuestra memoria principal colapse, debido a que la información que se maneja es exuberante, por eso, para evitar la sobrecarga de flujo de datos en el proceso de nuestro aplicativo, se desea guardar toda esta información en una memoria secundaria.

**Entrada:**

**Salida:**

**Nombre:** Medición de tiempo.

Se requiere medir los cambios de tiempo en que se demora en ejecución una actividad de nuestros métodos resolutivos, esto con el objetivo de ver qué tan eficiente está siendo nuestro aplicativo frente a otras estructuras de datos usadas.

**Entrada:** ------------------

**Salida:** -------------------

**2. Recopilación de información.**

Las estructuras de datos es una forma de organizar un conjunto de datos elementales con el objetivo de facilitar su manipulación. Un dato elemental es la mínima información que se tiene en un sistema.

Tipos de estructuras de datos posibles:

-Listas: Enlazadas simple o doblemente enlazadas.

-Pilas

-Colas

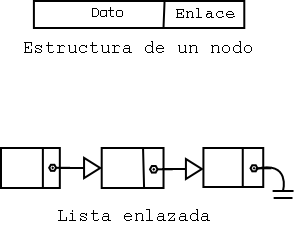
-Árbol ABB

-Árbol AVL

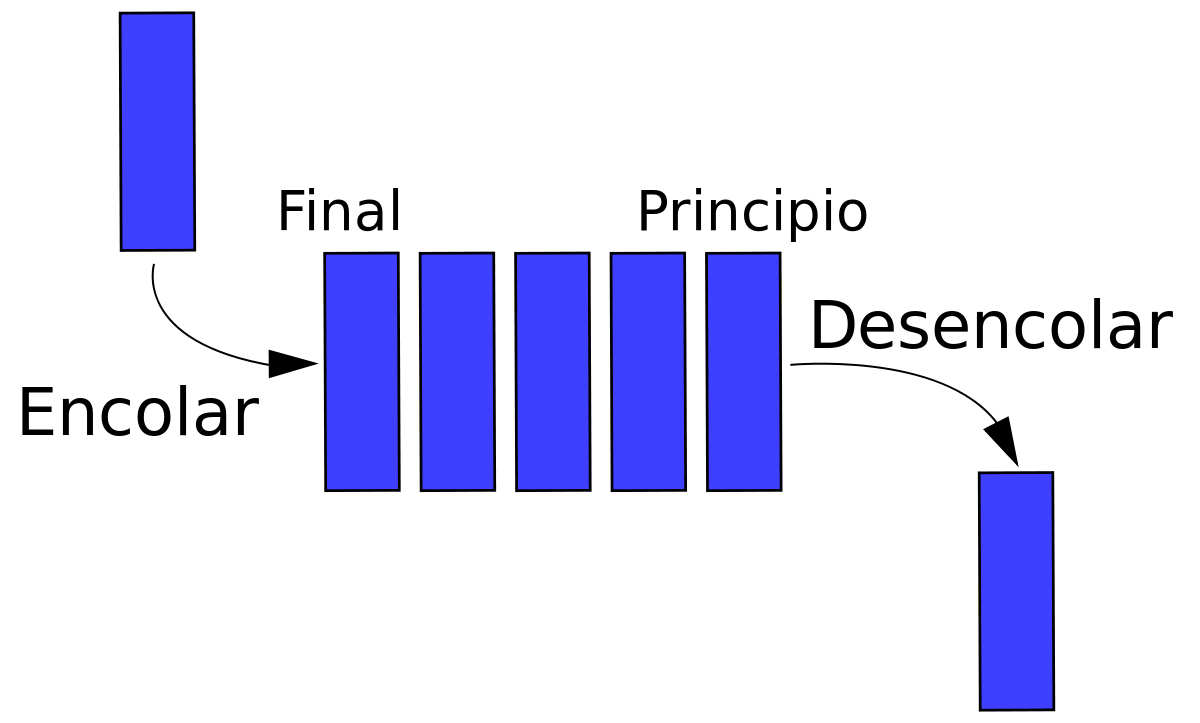
-Árbol N-ario

-Árbol Rojo-Negro

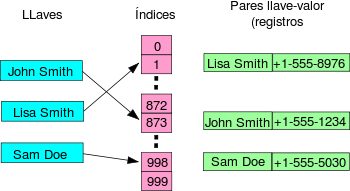
**Lista enlazada:** Son un conjunto de datos que se encuentran en forma secuencial a través de nodos enlazados, en el caso de las doblemente enlazadas cada nodo tiene un enlace a su nodo anterior y a su siguiente.



**Cola:** Es una lista ordenada o estructura de datos en la que el modo de acceso es de tipo FIFO (First In First Out, primero en entrar, primero en salir) que permite almacenar y recuperar datos.



**Tabla Hash:** Es una estructura de datos especializada para almacenar grandes cantidades de datos, consiste en un nodo que contiene un índice y una llave, sus operaciones son bastante eficientes en especial su búsqueda, ya que con el índice único para cada elemento se puede acceder fácilmente a él.

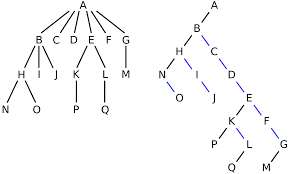


**Árbol Binario de Búsqueda (ABB):**

La búsqueda en árboles binarios es un método de búsqueda simple, dinámico y eficiente considerado como uno de los fundamentales en Ciencia de la Computación. Un árbol binario se define en que cada nodo tiene a lo más un hijo a la izquierda y uno a la derecha, si estos son nulos, se tiene una hoja. Su principal ventaja es su complejidad al momento de búsqueda representada en una función logarítmica (O(log(n+1)).

**Árbol N-ario:**

Es una estructura recursiva en la que cada elemento tiene n árboles n-arios asociados



**Árbol Rojo-Negro:**

Son una clase de árboles binarios de búsqueda aproximadamente equilibrados, en el que cada nodo contiene un bit adicional, su color, el cual puede ser rojo o negro. A su vez como es un árbol de búsqueda binaria balanceado tiene una complejidad de busqueda de O(logn).

**Árbol AVL:**

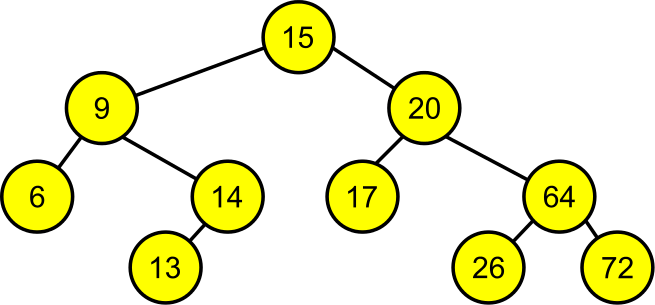
Un árbol AVL es un árbol binario de búsqueda que cumple con la condición de que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como máximo 1. Por lo tanto, garantiza que las operaciones de búsqueda, inserción y eliminación en este árbol binario ordenado tomen en el peor caso O(log n).

**3. Búsqueda de soluciones creativas.**

Se necesita una solución al problema; para esto se basó en la búsqueda de estructuras de datos que me permitieran solucionar las necesidades con mayor eficiencia, debido a que, se van a manejar números de datos bastante grandes . Por ende, optamos por una búsqueda de soluciones exhaustiva a través listas de revisión para examinar los diferentes puntos, áreas y posibilidades de diseño utilizando estructuras de datos.

1. **Árbol Binario de Búsqueda (ABB):**

Árbol en el cual cada uno de sus nodos puede tener no más de 2 hijos. En el cual se cumple con la jerarquización de que todos los nodos a la izquierda de la raíz son menores que ella y todos los nodos a la derecha de la raíz son mayores que ella. Por lo tanto, el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores. Así, es una buena opción para garantizar una búsqueda eficiente de los jugadores guardados.



2. **Árbol Rojo y negro**

Es un Árbol Binario de búsqueda que debe cumplir con las siguientes propiedades:

1 - Cada nodo es rojo o negro.

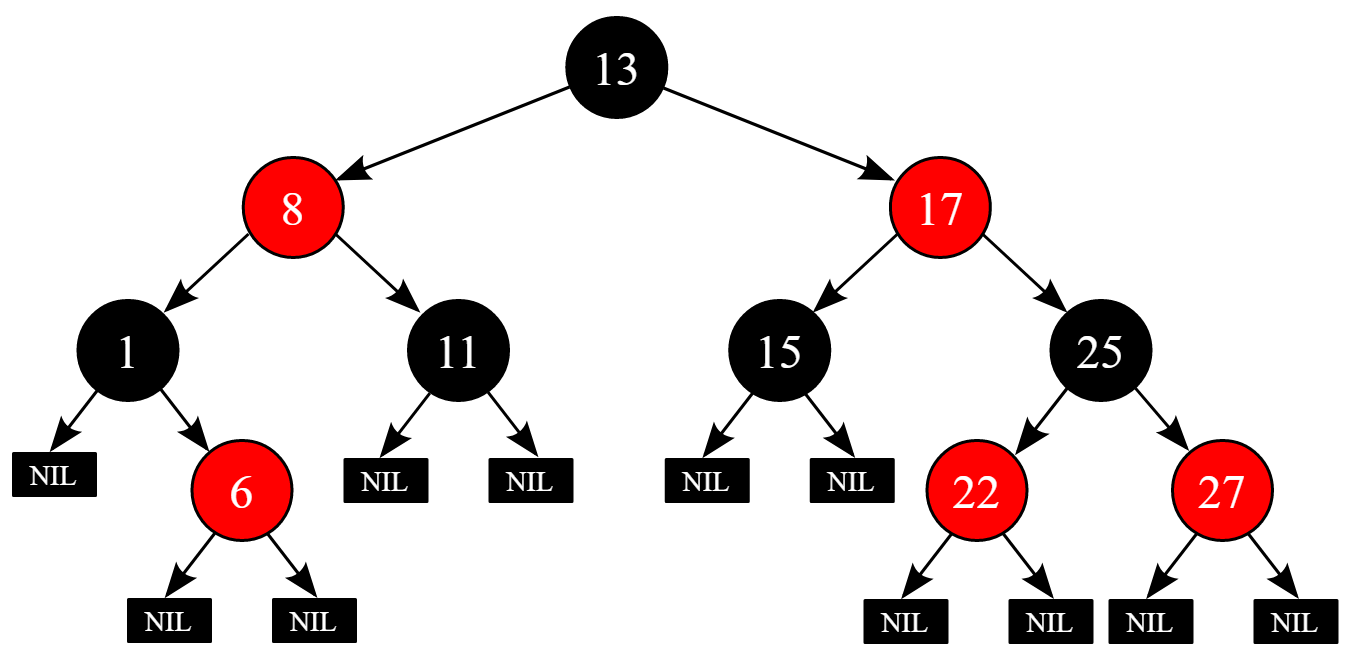
2 - La raíz es negra.

3 - Toda hoja (NIL) es negra.

4 - Si un nodo es rojo, sus dos hijos son negros. (no puede haber dos rojos consecutivos en un camino)

5 - Todo camino desde un nodo a cualquier hoja descendente contiene el mismo número de nodos negros.

Así pues, de acuerdo a la propiedad 5, todo árbol rojinegro con n nodos internos tiene una altura menor o igual que 2 log(n+1). Por lo tanto, las operaciones de búsqueda pueden implementarse en tiempo O(log n) para árboles rojinegros con n nodos, así como sus funciones para insertar y eliminar.



Además, para ahorrar espacio en memoria, en este árbol se crea un nodo auxiliar T. nill de color negro, al cual van apuntar todos los nodos que apuntan a un nodo NIL. Por lo tanto, se convierte en una de nuestras alternativas para facilitar la organización de jugadores y así mismo su búsqueda.

**3. Árbol AVL:**

Es un tipo especial de árbol binario. Es un árbol de búsqueda binaria autobalanceable, su objetivo es intentar mantener su altura, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento, automáticamente, esto con la intención de tener una complejidad O(log n) en las operaciones de búsqueda, inserción y eliminación.

Su propiedad consiste en 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, para esto cada nodo contiene el dato que controla el factor de equilibrio.

El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo, por definición, para un árbol AVL, este valor debe ser -1,0 ó 1.

Si el factor de equilibrio de un nodo es:

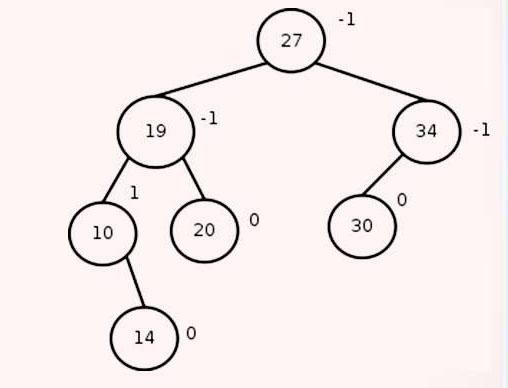
0 : el nodo está equilibrado y sus subárboles tienen exactamente la misma altura.

1 : el nodo está equilibrado y su subárbol derecho es un nivel más alto.

-1: el nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

2 o mayor: es necesario equilibrar.

Para lograr el reequilibrado en un árbol AVL es necesario utilizar las operaciones de rotaciones, existen dos casos: rotación simple o rotación doble.



**4. Transición de las ideas a los Diseños Preliminares.**

En esta fase se va a indagar por qué se descartaron algunas alternativas nombradas en la fase anterior; recapitulando, estas alternativas fueron:

* Listas enlazadas
* Cola
* Tabla Hash
* Pila

Cabe aclarar que las ideas descartadas fueron más que las ideas escogidas, por lo que se llevará a cabo un proceso de justificación en cada caso, empezando por las ideas descartadas, donde se explicará a partir de nuestras percepciones, ideas y requisitos del problema por qué fueron excluidas, y para el caso de las ideas no descartadas se inducirá un diseño preliminar, esto para conocer una mayor información sobre cada una de las ideas.

**Listas enlazadas.**

Tenemos las listas enlazadas, mejor conocidas como LinkedList. Esta es una estructura de datos que consiste en una serie de nodos enlazados que nos permite almacenar una variedad de datos.

Relacionando lo anterior con el sistema de requerimientos que tiene nuestro problema, se pensó que, por ser una estructura lineal, su búsqueda en nuestro aplicativo no será tan eficiente, ya que nuestra institución FIBA maneja una abundante lista de jugadores, y esto para nuestra estructura le tomaría mucho tiempo, por lo que estaríamos gastando muchos recursos en nuestro sistema.

**Cola.**

La cola es una estructura de dato, que permite el almacenamiento y recuperación de datos; caracterizada por usar el método FIFO el cual consiste en sacar el elemento que primero ha ingresado a la estructura, en otras palabras, si tenemos una serie de datos que se han ingresado de la siguiente forma:

**Q** = { e1, e2, e3, e4…, en} donde *e* es igual al tipo de dato que estamos almacenando. Por ende, solo nos permitirá obtener el elemento *e1* de la cola, ya que este ha sido el primer elemento insertado o agregado.

Relacionando lo anterior con nuestro problema, es claro comprender que no es necesario obtener el primer elemento que se ha agregado, ya que el programa necesita agregar una exuberante cantidad de jugadores pertenecientes a la FIBA, y darle la opción de poder buscar alguno de ellos. por medio de uno de sus atributos. En este caso esta estructura no nos ofrece o nos garantiza esta búsqueda, por lo tanto, es una de nuestras alternativas descartadas.

**Tabla Hash:**

Tabla Hash es una estructura de datos muy útil para las búsquedas eficientes, su tiempo promedio es O(1), sin importar la cantidad de elementos que contenga la tabla. Sin embargo, su principal desventaja es que los datos son almacenados de una forma “aleatoria”, lo que nos conlleva a una desventaja para el problema planteado, ya que en ocasiones es necesario realizar una búsqueda de manera ordenada, porque así las búsquedas son mucho más eficientes.

**Pila:**

La pila es una estructura de dato muy similar a la Cola, ya que nos permite almacenar y recuperar datos, pero en este caso la Pila nos ofrece el sistema LIFO, el cual consiste en poder recuperar o extraer el dato que se ha ingresado de último, en otras palabras, si tenemos una serie de datos que se han ingresado de la siguiente manera:

**S** = { e1, e2, e3, …., en} donde *e* es igual al tipo de dato que estamos almacenando.

Como se mencionó anteriormente, solo nos permite tomar o extraer el último elemento que se ha ingresado a la pila, entonces solo nos dejará obtener el elemento *en* de la pila **S.**

Teniendo claro lo anterior, cabe resaltar que no es una estructura que supla con los requerimientos de nuestro problema, esto debido a que la pila nos ofrece un sistema muy restringido en cuanto a la obtención de datos, esto quiere decir que solo nos basaremos en el último jugador perteneciente a la FIBA que se ha ingresado a nuestra estructura de dato, y esto no es lo que nos pide solucionar el problema, si nos enfocamos o hacemos énfasis en lo que se requiere, es evidente que la obtención o la búsqueda de un jugador o de un dato es libre y no tan específico como lo ofrece una Pila. En otras palabras, necesitamos un dato dado por un usuario para poder encontrar un jugador, cosa que no nos ofrece la estructura Pila.

En contextualización, todo lo dicho anteriormente pertenece a las ideas descartadas, donde fueron aclaradas las razones por las cuales no se tuvieron en cuenta para la solución de nuestro problema. Ahora, seguimos con un diseño preliminar, que consiste en conocer u obtener más información de las ideas que no descartamos, o sea de las alternativas que tomamos como referencia para la solución de nuestro problema. Para recordar, estas alternativas fueron las siguientes estructuras de datos:

* Árbol Binario de Búsqueda (ABB)
* Árbol Rojo y Negro (ARN)
* Árbol AVL.

**Diseño preliminar.**

Aún nos queda lo más relevante, y es llevar a cabo las razones de nuestras alternativas que servirán para la resolución de nuestro problema (ideas no descartadas), sin embargo, estas razones se plantearon a través de ciertas ideas concretas, por esto, se formuló un diseño preliminar básico y sencillo para comprender la relación que existe entre las alternativas y el problema que se nos plantea.

Antes de analizar el diseño preliminar, cabe recalcar que se inducirá una estructura de diseño, que a continuación se plantea:

* Ideas
* FODA
* Análisis
* Conclusión

En primer lugar, las ideas se trataron un poco dispersas, por eso es un proceso que no es organizado, pero es importante para una buena disidencia de alternativas, aunque esto no quiere decir que es un mal comienzo, ya que es una base para el seguimiento de nuestras escogencias. Algunas de nuestras ideas fueron:

* Es indudable que la lista de jugadores pertenecientes a la FIBA es extensa, y sus atributos son variados, por ejemplo, un jugador tiene nombre, apellido, edad, posición de juego, puntaje, altura, entre otras. Vemos que son varias características que componen a un solo jugador, ahora imaginemos esto para una cantidad bastante grande de jugadores, claramente sería algo tedioso y pesado para la ejecución de un aplicativo.
* Otra idea relevante, fue la búsqueda eficiente para una grande cantidad de datos, por lo que llegamos a que sería mejor buscar de una manera donde los datos estén ordenados.
* Por último, una idea de recurso y tiempo, es el cómo está funcionando el programa y cuánto tiempo está tardando para hacer una operación. Básicamente es la comparación de tiempo en las diferentes alternativas escogidas para la resolución al problema.

En segundo lugar, para hacer un énfasis en cada idea, se clasificó las fortalezas, oportunidades, debilidades y amenazas de nuestras alternativas para así verificar la potencialidad de nuestra resolución.

Para nuestra primera y segunda idea se tomó como base la estructura de datos **Árbol Binario de Búsqueda**(ABB).

|  |  |  |
| --- | --- | --- |
|  | Factores internos | Factores externos |
| Puntos positivos | Fortalezas   * Organiza los datos (Izquierda, datos menores y derecha, datos mayores) | Oportunidades   * Almacena grande cantidades de datos |
| Puntos negativos | Debilidades   * Casi siempre está desbalanceado | Amenazas   * Búsqueda lineal. |

*tabla 1*

Para la **tabla 1** tenemos que los puntos positivos se asemeja más a lo planteado en el problema, porque nos ofrece un sistema de orden en el cual al lado izquierdo de la raíz están los datos menores y a la derecha los datos mayores, sin embargo, también tenemos puntos negativos, en el que dependiendo de cómo están organizados los datos, la búsqueda puede ser eficiente, en el peor de los casos podemos llegar a una búsqueda lineal, esto quiere decir que tomaría mucho tiempo para una cantidad de datos extensa.

Para nuestra primera y segunda idea, se tomó otra estructura de datos llamada **Árbol Rojo y Negro** (ARN).

|  |  |  |
| --- | --- | --- |
|  | Factores internos | Factores externos |
| Puntos positivos | Fortalezas   * Eficiencia * Organiza los datos | Oportunidades   * Está aproximadamente balanceado * Es dinámico |
| Puntos negativos | Debilidades   * La inserción y supresión es muy tardía. | Amenazas   * Datos erróneos |

*tabla 1.1*

Para la **tabla 1.1** nos encontramos que nuestros puntos positivos mejoraron en comparación al árbol anterior, ya que el problema que tenía el anterior de desbalanceo en este se reduce porque conduce a un sistema de balanceo por colores. Por un lado, también organiza los datos: izquierda los menores y derecha los mayores. Por otro lado, los puntos negativos se relacionan con el tiempo de ejecución, ya que, para suplir un árbol aproximadamente balanceado, se toma mucho más tiempo verificando cada caso para insertar y eliminar, esto simplemente para mantener siempre nuestra invariante activa.

Por último, como otra alternativa para resolución de nuestra idea uno y dos aparece la última estructura de datos escogida: **Árbol AVL.**

|  |  |  |
| --- | --- | --- |
|  | Factores internos | Factores externos |
| Puntos positivos | Fortalezas   * Eficiencia * Organiza datos | Oportunidades   * Está siempre balanceado * Es dinámico |
| Puntos negativos | Debilidades   * Inserción para datos menores. | Amenazas   * Datos erróneos |

*tabla 1.2*

Para la **tabla 1.2** nos encontramos con el árbol más productivo y eficiente de todas las 3 alternativas, ya que, si analizamos los puntos positivos, esto nos ofrece un árbol 100% balanceado, eficiente en búsqueda donde en el peor caso es de O(log2n). Cuando nos referimos a dinámico es en el sentido de que aun así agreguemos secuencialmente datos, el árbol está en la capacidad de estar balanceado. Por otro lado, solamente se tiene una debilidad y es que para un proceso de inserción para datos menores se toma un poquito más de tiempo que cuando insertamos una grande cantidad, pero la diferencia no es determinante.

Para la última idea, solamente se tomaría el tiempo de ejecución de cada estructura con la misma cantidad de datos, para así corroborar cuál es más eficiente. Por esto, se basará una evaluación en la siguiente fase para determinar la mejor alternativa para nuestro programa.

Para resumir, todas las alternativas seleccionadas para la resolución de nuestro problema tienen puntos negativos, sin embargo, estos puntos no influyen mucho en nuestro aplicativo, porque, por una parte pueden repararse gracias a las fuerzas y oportunidades de cada alternativa, y por otra parte, la codificación nos ofrece una ciertas posibilidades para evitar muchos errores, por ejemplo en el caso de que el usuario quiera ingresar o almacenar un dato que no soporte o sea el indicado para el árbol.

**Fase 5. Evaluación y selección de la mejor opción.**

En la evaluación vamos a tener en cuenta varios criterios:

**Criterio A:** Es dinámico

[2] Dinámico

[1]Fijo

Este criterio se evalúa si la alternativa presenta un comportamiento activo o cambiante para el aplicativo.

Se califica 1 si no varía dentro del programa.

Se califica 2 si varía dentro del programa.

**Criterio B:** Suple con la solución planteada en el problema.

[3] Sí

[2] Suple, pero no del todo.

[1] No

Este segundo criterio, evalúa si la alternativa satisface totalmente el requerimiento que se nos presentó en el problema.

Se califica 3, si lo satisface totalmente.

Se califica 2, si lo satisface de vez en cuando.

Se califica 1, si no lo satisface.

**Criterio C**: Mejora el rendimiento (en tiempo de ejecución) para el aplicativo

[3] Sí

[2] Muy poco

[1] No varía

Para este tercer criterio, se evalúa si la alternativa aumenta el rendimiento del aplicativo.

Se califica 3, si en realidad sirve para mejorar la ejecución del aplicativo.

Se califica 2, si mejora en tiempo de ejecución, pero muy insignificante.

Se califica 1, no mejora en lo absoluto.

**Criterio D:**  El árbol está balanceado

[3] Totalmente balanceado

[2] Aproximadamente balanceado

[1] No está balanceado

Para este último criterio, se evalúa si la estructura de dato nos asegura un balanceo relevante en todos los casos.

Se califica 3, si está siempre balanceado.

Se califica 2, si se acerca a un balanceo.

Se califica 1, no está balanceado.

***Evaluación:***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Métodos / Criterios | **Criterio A** | **Criterio B** | **Criterio C** | **Criterio D** | **Total** |
| ABB | 2 | 3 | 2 | 1 | 8 |
| Árbol Rojo y Negro | 2 | 3 | 3 | 2 | 10 |
| Árbol AVL | 2 | 3 | 3 | 3 | 11 |

*tabla 2*

En la **tabla 2** se puede ver claramente una calificación objetiva para cada una de las estructuras seleccionadas, por lo que empezaremos analizar cada una de estas dependiendo de su puntaje.

En primer lugar, el árbol binario de búsqueda procede a tener un puntaje total de 8, para el criterio A, se dice que es dinámico, ya que no tiene un límite o se le asigna un tamaño para agregar cierta cantidad de datos. Para el criterio B, si suple con la necesidad del problema, por lo que almacena de manera ordenada una grande cantidad de datos. El criterio C tiene relación con el criterio D, porque el rendimiento de un árbol se basa en sí está balanceado, esto quiere decir que la altura de los dos subárboles de todo nodo difiere a lo sumo en 1. Por ende, el ABB no nos garantiza un balanceo y el rendimiento mejorará muy poco dentro del aplicativo.

En segundo lugar, el árbol rojo y negro obtuvo un puntaje total de 10, a diferencia del árbol binario de búsqueda, este nos ofrece un árbol aproximadamente balanceado, y por tanto, nos mejorará el rendimiento dentro de nuestro aplicativo. Por esto, en los casos donde varía con respecto al anterior fue en el criterio C y D, donde obtuvo un puntaje de 3 y 2, respectivamente.

Por último, el árbol AVL obtuvo un puntaje total de 11 puntos, la diferencia contra el árbol rojo y negro fue en solo un punto, y esto debido a que el árbol AVL está siempre balanceado, esto quiere decir que en el criterio D, obtuvo un puntaje de 3 puntos, y por tanto, la secuencia de rendimiento en un árbol AVL va ser más determinante que cualquiera de las otras estructuras.

**6. Preparación de Informes y Especificaciones**

En esta fase se llevará a cabo la parte del diseño de nuestro método de ingeniería para la solución del problema.

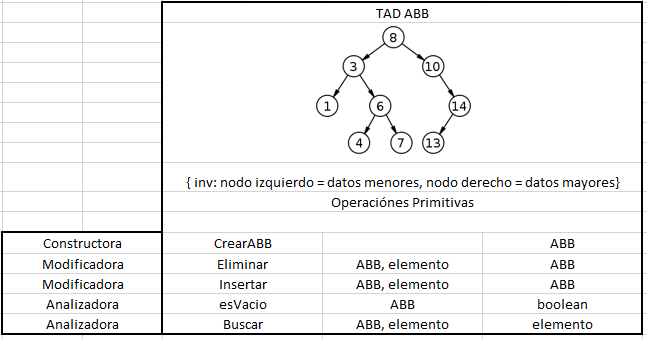
En primer lugar, se presentará los Tipos de Datos Abstractos de nuestras estructuras de datos que se implementarán en la codificación, para así llevar una comprensión detallada de las operaciones que nos ofrece cada una de estas.

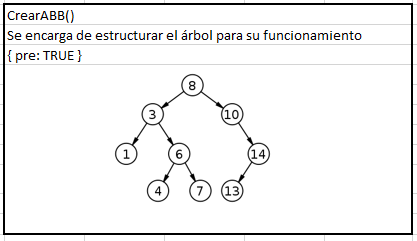
En segundo lugar, también se incluirá el diagrama de clases respectivo para la descripción de nuestra codificación de manera que cualquier programador o persona relacionada con el tema pueda entender las clases, los atributos, las operaciones(métodos) y las relaciones entre objetos.

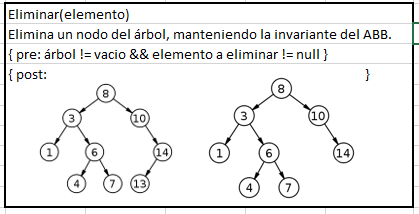
Por último, para comprobar y verificar el comportamiento de los métodos o las salidas que esperamos en cada operación, se usará el diseño de casos de pruebas unitarias, para detallar las entradas, y las salidas que se deben dar para un buen funcionamiento del aplicativo.

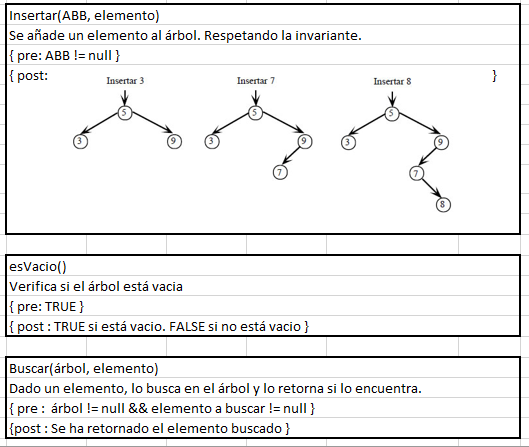
**TADs**

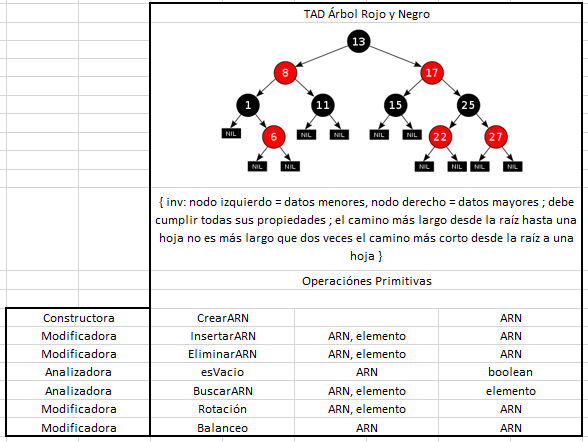
**ABB**

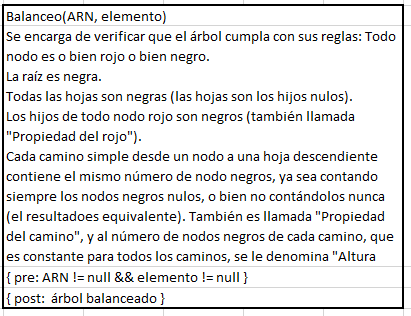
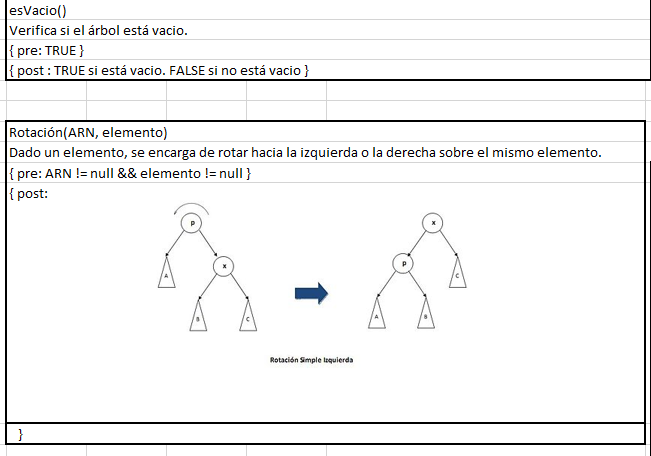
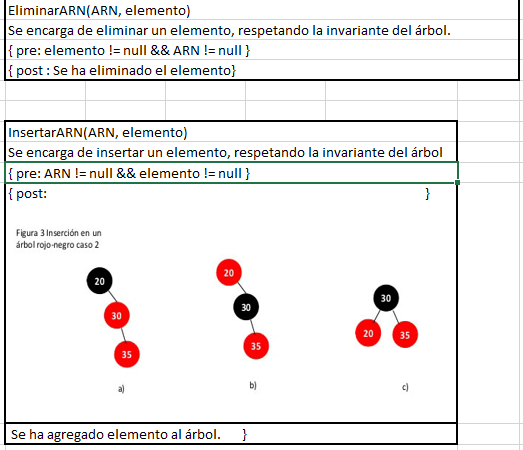
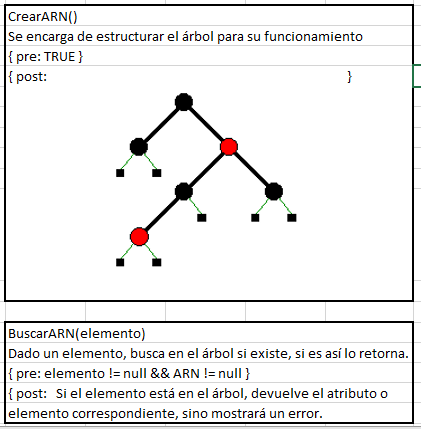
****

****

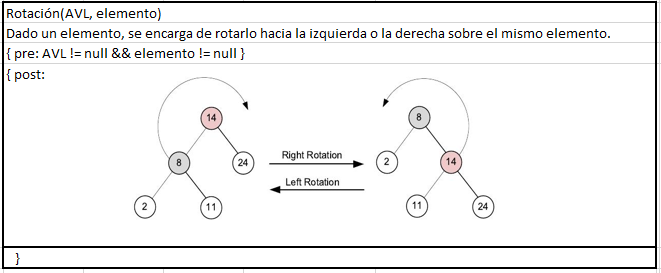
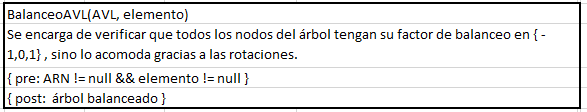
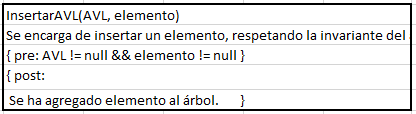
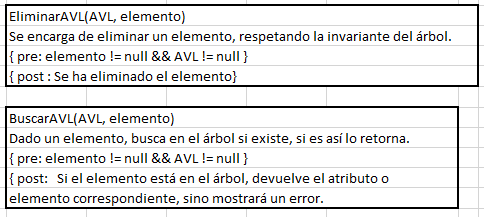
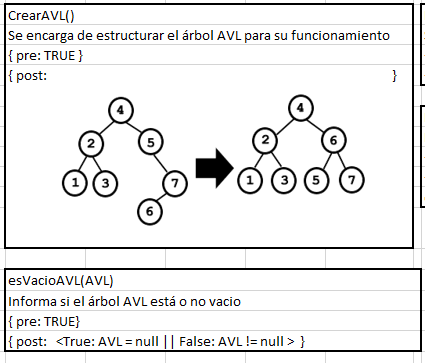
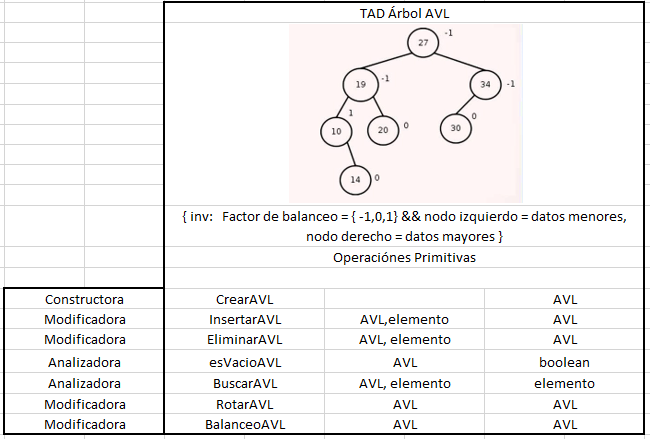




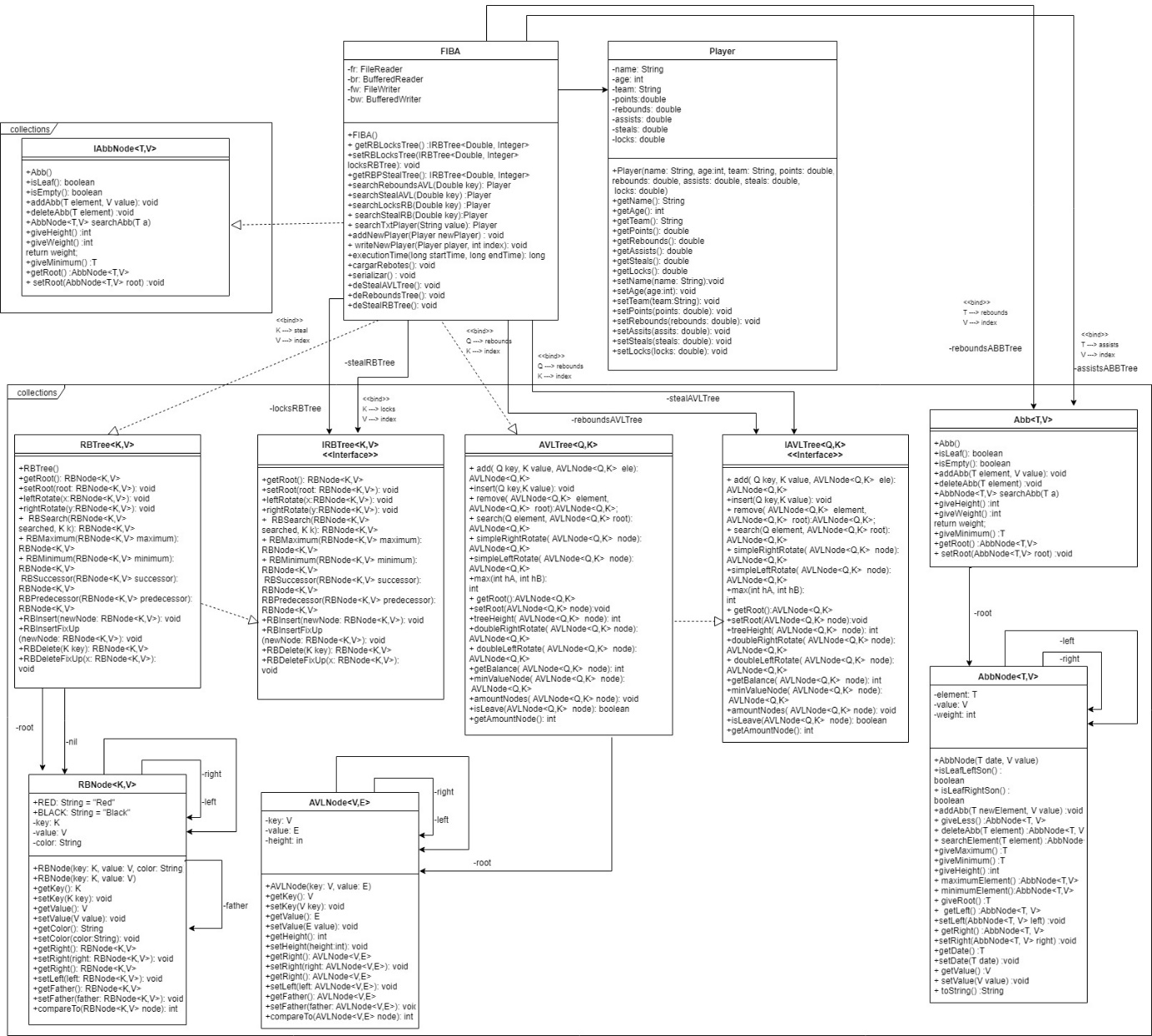
**Árbol Rojo y Negro**

****

**Árbol AVL**

****

**Diagrama de clases**



Para ver mejor el diagrama, se encuentra adjunto al GitHub

**Diseño de pruebas**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Clase | Método | Escenario | Valores | Resultado |
| FIBATest | insertTestRB() | Se instancio un objeto de tipo FIBA y se inicializan un grupo de integer | P1=100.0,1  P2=92.0, 2  P3=93.0, 3  P4=105.0, 4  P5=102.0, 5  P6=91.0, 6 | Los objetos han sido ingresados correctamente y el árbol rojo y negro se encuentra balanceado |
| FIBATest | deleteTestRB | Se instancio un objeto de tipo FIBA, se inicializa un grupo de integers y se añaden al árbol 6 objetos | P1=102.0 | El objeto ingresado ya no hace parte del árbol y este se encuentra balanceado |
| FIBATest | SearchTestRB() | Se instancio un objeto de tipo FIBA, se inicializa un grupo de integers y se añaden al árbol 6 objetos | P1=102.0  P2=93.0 | Los objetos que se buscan fueron encontrados correctamente. |
| FIBATest() | insertTestAVL() | Se instancio un objeto de tipo FIBA y se inicializan un grupo de integer | P1=100.0,1  P2=92.0, 2  P3=93.0, 3  P4=105.0, 4  P5=102.0, 5  P6=91.0, 6 | Los objetos han sido ingresados correctamente y el árbol avl se encuentra balanceado |
| FIBATest() | DeleteTestAVL() | Se instancio un objeto de tipo FIBA y se inicializan un grupo de integer | P=10.0 | El objeto ingresado ya no hace parte del árbol y este se encuentra balanceado |

**Bibliografía**

<https://prezi.com/ael64ckh47wd/aplicacion-de-la-estructura-de-datos/>

<https://es.wikipedia.org/wiki/%C3%81rbol_AVL>

<https://es.wikipedia.org/wiki/%C3%81rbol_rojo-negro>

<https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda>