Método de la ingeniería

Identificación del problema

La Federación Internacional de Baloncesto (FIBA) desea consolidar en una aplicación, los datos de mayor relevancia de cada uno de los profesionales del baloncesto en el planeta. La aplicación debe permitir a los usuarios:

- Efectuar diferentes tipos de consultas y operaciones (Insertar, buscar, eliminar y modificar los datos).
- Permitir el manejo de información de gran tamaño, ya sea de manera masiva o a través de una interfaz.
- Lograr que el tiempo de ejecución de las consultas y operaciones sean en el menor tiempo posible a pesar del tamaño masivo de la información que se quiere manejar.

Se debe tener en cuenta que para cada jugador se almacenar su nombre, edad, equipo, puntos por partido, rebotes por partido, asistencias por partido, robos por partido y bloqueos por partido.

Requerimientos funcionales:

Nombre	R.1 Insertar un jugador
Resumen	Permite al usuario agregar un jugador ya sea mediante la carga de archivos de texto plano o a través de la interfaz.
Entrada	- Nombre - Edad - Equipo - Puntos por partido - Rebotes por partido - Asistencias por partido - Robos por partido - Bloqueos por partido
Salida	El jugador se agrega efectivamente con los datos ingresados por el usuario.

Nombre	R.2 Eliminar un jugador
Resumen	Permite al usuario eliminar a un jugador.
Entrada	- Índice

Salida	Se elimina la información del jugador del aplicativo.

Nombre	R.3 Modificar la información de un jugador
Resumen	Permite realizar modificaciones de los datos de un jugador.
Entrada	- Nombre - Característica a modificar - Nuevo valor de la modificación
Salida	Se modifica la información del jugador por el nuevo valor.

Nombre	R.4 Buscar un jugador
Resumen	Permite buscar un jugadores en las estructuras de datos usadas en el aplicativo. Esta puede tener varias entradas, ya sea por puntos por partido, rebotes por partido, asistencias por partido, robos por partido o bloqueos por partido
Entrada	- Tipo de dato a buscar - Dato a buscar
Salida	Retorna a los jugadores que tienen ese dato que se está buscando

Nombre	R.5 Almacenar los datos de los jugadores
Resumen	Almacena los datos de cada jugador en un archivo de texto plano
Entrada	
Salida	Se genera un archivo en formato txt con los datos de los jugadores

Nombre	R.6 Mostrar tiempo de consulta
Resumen	Permite al usuario visualizar cuánto tiempo de demora el programa en consultar un jugador determinado
Entrada	- Consulta
Salida	Imprime en pantalla el tiempo que tardó el programa en realizar la consulta

Nombre	R.7 Cargar jugadores desde un archivo CSV
Resumen	Permite cargar en el programa los datos de los jugadores que se encuentren en un archivo CSV
Entrada	- Archivo CSV
Salida	Se ingresan los datos de los jugadores al aplicativo.

Recopilación de la información

Para llevar a cabo el aplicativo debemos tener en cuenta que hacen y cómo funcionan las estructuras de datos que se tienen como opción para implementar en el aplicativo :

- **Árbol binario de búsqueda (ABB):** Es un tipo de árbol binario en el cual sus nodos cumplen con ciertas reglas las cuales son: El subárbol izquierdo de cualquier nodo contiene valores menores que el que contiene dicho nodo, y el subárbol derecho contiene valores mayores. Si en este tipo árbol hacemos un recorrido *InOrden* proporcionará los elementos de los nodos ordenados de forma ascendente. Adicionalmente, la búsqueda de algún elemento en el ABB es eficiente ya que requiere O(h) de tiempo donde h es la altura del árbol, en el peor de los casos. En las operaciones de insertar y eliminar se tiene un tiempo de O(log n) para ser realizadas.
  A pesar de todo, este árbol puede generar problemas en tiempos de ejecución de determinados casos, pues no se auto-balancea después de cada inserción.

- **Árbol rojo-negro:** Es un tipo de árbol binario de búsqueda balanceado en el cual sus nodos tienen que seguir ciertas reglas, además de cumplir con las reglas de un ABB: La primera regla es que cada nodo debe ser rojo o negro, la segunda es que la raíz del árbol siempre tiene que ser negra, la tercera regla es que los hijos de un nodo rojo deben ser negros y la última regla es que cada camino que haya desde la raíz hasta una hoja debe tener el mismo número de nodos negros. Se usa este tipo de estructura de datos cuando se quiere tener un fácil acceso a los elementos que contiene ya que en este tipo de estructuras las operaciones básicas (buscar, insertar, eliminar) toman un tiempo de O(log n) para ser ejecutadas, mientras en un árbol binario de búsqueda toman O(h) donde h es la altura del árbol.

- **Árbol AVL:** Es un tipo de árbol binario de bùsqueda donde sus operaciones básicas (insertar, eliminar y buscar) requieren un tiempo de O(log n) para ser ejecutadas en el peor de los casos. Se diferencia del àrbol rojo-negro por el criterio de comparación que se ejerce en este tipo de àrbol, ya que este requiere un *factor de AVL* el cual se da

por la resta del nùmero de nodos en la sub-rama màs grande de la izquierda y el nùmero de nodos de la sub-rama màs grande la derecha debe dar entre 1, -1 o 0 para que este balanceado y todos sus nodos deben cumplir este criterio, de lo contrario las operaciones tomaran un tiempo mayor a O(log n).

Búsqueda de soluciones creativas

Para la solución del problema identificado se propone guardar a los jugadores ingresados o cargados mediante el archivo CSV en una estructura de datos para así poder manipular fácilmente los datos de los jugadores, esta estructura debe permitir realizar las operaciones que requiere el aplicativo (Insertar, modificar, buscar y eliminar).

Para el desarrollo de la aplicación se tiene como opción implementar las siguientes estructuras de datos:

1. **ArrayList <Player>:** Esta estructura de datos tiene como ventaja que facilita la implementación del programa y permite de igual manera realizar las operaciones necesarias requeridas en el aplicativo. Esta estructura de datos nos permite: Insertar en O(1), buscar en O(N) y eliminar en O(N), teniendo en cuenta que N es el tamaño de los datos almacenados en la estructura.

2. **Árbol binario de búsqueda (BST):** Esta estructura aunque dificulta la implementación tiene como ventaja que permite ordenar los datos según los criterios de comparación que se decidan. Adicionalmente nos permite igual realizar las operaciones en el siguiente tiempo: Insertar en O(log h), buscar en O(log h) y eliminar en O(log h) en caso de estar balanceado; si el árbol no está balanceado estas operaciones tardarían O(h), siendo h la altura del árbol .

3. **Árbol rojo-negro y árbol AVL:** Estas estructuras cuentan con la capacidad de que siempre están balanceadas permitiendo así realizar todas las operaciones básicas como Insertar, buscar y eliminar en O(log h) donde h es la altura del árbol.

Diseños preliminares

Para analizar cuál de las soluciones propuestas en el apartado anterior se debe implementar se plantean algunos de criterios de comparación que permitan determinar cuál/cuáles son las mejores soluciones, los criterios de comparación son los siguientes:

- Criterio 1:La estructura de datos debe permitir realizar todas las operaciones necesarias (Insertar,buscar y eliminar).

- Criterio 2:La complejidad temporal de las operaciones deber ser menor a O(N) en su peor caso
- Criterio 3:Fácil implementación y comprensión de la estructura de datos.
- Criterio 4:Utilizar estructuras de datos recursivas.
- Criterio 5:Las estructuras deben ser capaces de auto balancearse.

	Criterio 1	Criterio 2	Criterio 3	Criterio 4	Criterio 5	Total
ArrayList <Player>	X		X			2
Árbol binario de búsqueda (BST)	X	X		X		3
Árbol rojo-negro y árbol AVL	X	X		X	X	4

En estas estructuras no se guardaran los jugadores, sino un índice que lleva a un archivo de texto con la información necesaria para crear un objeto de tipo jugador. Lo anterior con el fin de usar memoria secundaria a cambio de memoria RAM, pues es mucho más costosa en cuestión de recursos la memoria RAM.

Selección de la mejor solución

Teniendo en cuenta los criterios y los resultados obtenidos a partir de estos para cada solución de las que se plantearon en el apartado anterior ahora se debe elegir cuáles serán las que se implementaran y cuáles se deben descartar.

Con base a los resultados se descarta la solución número 1 y 2 de implementar una ArrayList <Player> o un BST porque no cumplieron los criterios 3 y 5 para el caso del BST y los criterios 2, 4 y 5 para el caso de la ArrayList.

Por otra parte, debido a que los árboles rojo-negro y AVL obtuvieron una mayor calificación total se decide implementar estas estructuras para lo que se refiere a la parte de realizar operaciones con los índices para insertar, buscar y eliminar jugadores. Sin embargo, se toma la decisión de implementar el BST debido a que esta estructura permitirá comparar los tiempos de búsqueda de los árboles para los casos en los cuales se encuentren balanceados y donde no lo estén.

En conclusión, se implementaran las estructuras de árboles rojo-negro, AVL y BST, excluyendo así únicamente la ArrayList que aunque se consideró como una posible alternativa para la solución del problema no brinda ninguna ventaja o superioridad sobre las demás estructuras propuestas.

Diseño de pruebas unitarias

- **BinaryTree**

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
BinaryTree	BinaryTree()	setup()		Crea un objeto de tipo BinaryTree con todos sus nodos inicializados en nil.

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
BinaryTree	insert()	setup()	Utiliza el árbol creado por el método setup()	Agrega un nuevo nodo al árbol en la raíz

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
BinaryTree	insert()	setup()	Utiliza el árbol creado por el método setup()	Agrega un nuevo nodo al árbol el cual queda en la raiz tambien, se crea un nodo raiz de tipo ArrayList y se almacena los 2 objetos en el mismo nodo (la raíz).

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
BinaryTree	insert()	setup()	Utiliza el árbol creado por el método setup()	Agregan 2 nodos nuevos al árbol con

				llaves de valor (5 y 6) dejando de tal forma en la raíz el nodo con llave (5) y a su derecha el nodo de llave (6)

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
BinaryTree	insert()	setup()	Utiliza el árbol creado por el método setup()	Agregan 2 nodos nuevos al árbol con llaves de valor (5 y 1) dejando de tal forma en la raíz el nodo con llave (5) y a su izquierda el nodo de llave (1)

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
BinaryTree	insert()	setup()	Utiliza el árbol creado por el método setup()	Agrega 5 nodos nuevos al árbol siguiendo la invariante del BST

- **RBBinaryTree**

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	RBBinaryTree()	setup()		Crea un objeto de tipo RBBinaryTree con todos sus nodos inicializados en nil.

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**

RBBinaryTree	insert()	setup()	Utiliza el árbol creado por el método setup() e inserta un elemento en el	Agrega el nodo de forma eficaz en la raíz del árbol con valores iniciales para la llave de 5.0 (Double)y valor del nodo como 1 (String)

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	insert()	setup2()	Utiliza el árbol creado por el método setup2() con la raíz que ya se encuentra inicializada	Agrega el nodoa ingresar al árbol en la rama izquierda y de color rojo

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	insert()	setup2()	Utiliza el árbol creado por el método setup2() con la raíz que ya se encuentra inicializada	Agrega el nodoa ingresar al árbol en la rama derecha y de color rojo

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	insert()	setup2()	Utiliza el árbol creado por el método setup2() e inserta dos elementos en el	Agrega ambos nodos al árbol, uno en la rama derecha y otro en la rama izquierda de la raíz

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	insert()	setup()	Utiliza el árbol creado por el método setup() e inserta 10	Agrega los nodos al árbol, de tal manera que se obtiene

			elementos en el	un árbol balanceado con 11 nodos en el.

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	search(K key)	setup3()	Utiliza el árbol creado por el método setup3() que es un árbol con 3 nodos y balanceado	Retorna el nodo buscado mediante la llave (9.2) que se ingresa por parámetro, el cual se encuentra al lado derecho de la raíz

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	search(K key)	setup3()	Utiliza el árbol creado por el método setup3() que es un árbol con 3 nodos y balanceado	Retorna el nodo buscado mediante la llave (4.5) que se ingresa por parámetro, el cual se encuentra al lado izquierdo de la raíz

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	min(RBTreeNode <K, V> node)	setup4()	Utiliza el árbol creado por el método setup4() que es un árbol con 11 nodos y balanceado	Retorna el nodo que tenga la llave con menor valor (0), es decir el nodo que se encuentra más a la izquierda

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**

RBBinaryTree	max(RBTreeNode<K, V> node)	setup4()	Utiliza el árbol creado por el método setup4() que es un árbol con 11 nodos y balanceado	Retorna el nodo que tenga la llave con mayor valor (9), es decir el nodo que se encuentra más a la derecha

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	getRoot()	setup4()	Utiliza el árbol creado por el método setup4() que es un árbol con 11 nodos y balanceado	Retorna la raíz del arbol

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
RBBinaryTree	delete()	setup4()	Utiliza el árbol creado por el método setup4() que es un árbol con 11 nodos y balanceado	Se elimina el nodo raiz del arbol y ejecutan los casos para que el árbol continúa estando balanceado, cambiando su raíz a (5.0)

- **AVLTree**

Clase	**Método**	**Escenario**	**Entradas**	**Resultado**
AVLTree	AVLTree()	setup()		Crea un objeto de tipo AVLTree e inserta un nodo en su raíz con llave (1.0) y su factor de carga es 0

Clase	Método	Escenario	Entradas	Resultado
AVLTree	insert()	setup()	Utiliza el árbol creado por el método setup()	Se comprueba que la raíz no sea nula y que se haya agregado correctamente

Clase	Método	Escenario	Entradas	Resultado
AVLTree	search(K key)	setup()	Utiliza el árbol creado por el método setup()	Busca en el árbol un nodo con llave (1.0), el cual se encuentra en su raíz.

Clase	Método	Escenario	Entradas	Resultado
AVLTree	AVLTree()	setup()	Utiliza el árbol creado por el método setup()	Agrega 8 nodos nuevos al árbol de forma que queda un árbol balanceado con 9 nodos

Clase	Método	Escenario	Entradas	Resultado
AVLTree	getRoot()	setup2()	Utiliza el árbol creado por el método setup2()	Se obtiene la raiz del arbol, el cual es un nodo con llave (4)

Clase	Método	Escenario	Entradas	Resultado
AVLTree	search(K key)	setup2()	Utiliza el árbol creado por el método setup2()	Busca en el árbol un nodo con llave (6.0), el cual se encuentra a la derecha del arbol.