

TAREA INTEGRADORA 2: FIBA REQUESTS

JULIAN CAMILO BOLAÑOS

ARIEL EDUARDO PABÓN BOLAÑOS

JUAN FELIPE BLANCO TIGREROS

2 DE NOVIEMBRE DE 2021

ALGORITMOS Y ESTRUCTURAS DE DATOS

UNIVERSIDAD ICESI

1. INFORME DEL MÉTODO DE LA INGENIERÍA

FASE 1: Identificación del problema y requerimientos funcionales:

Contexto del problema:

La FIBA es el ente regulador a nivel mundial del baloncesto, de sus reglas y de los eventos más importantes que lo involucran. Esta requiere una solución que le permita consolidar la cantidad masiva de datos recopilados anualmente. Esto último con la finalidad de facilitar las consultas y la realización de análisis deportivos.

Definición del problema:

La solución del problema puede ser un software que administre los datos de los jugadores de baloncesto. Debe ser capaz de realizar consultas muy rápidas con criterios relevantes para el deporte.

Especificación de requerimientos:

1. *Requerimientos funcionales:*

RF 1. Manejar los datos de los jugadores. El programa debe estar en capacidad de ingresar, eliminar y modificar los jugadores que existan en el sistema.

- ❖ **RF 1.1. Ingresar jugadores junto con sus datos.** Se debe poder ingresar jugadores al sistema junto con sus datos (nombre, edad, equipo y otras 5 estadísticas) a través de un archivo .csv o de una interfaz gráfica.

- ❖ **RF 1.2. Eliminar jugadores junto con sus datos.** Se debe poder eliminar a los jugadores existentes en el sistema junto con sus datos.
- ❖ **RF 1.3. Modificar datos de un jugador.** Se deben poder modificar los datos de los jugadores existentes en el sistema. Lo anterior implica que se debe ingresar nueva información que la reemplace.

RF 2. Consultar datos de los jugadores. Se deben poder buscar varios jugadores a través del nombre (una consulta que utilice los caracteres digitados por el usuario) o alguna de las 5 estadísticas que tenga. Esto último tiene que tomar en cuenta algún criterio de comparación para hacer la consulta (menor, menor o igual, igual, mayor o igual y mayor). También se debe informar el tiempo que tomó realizar la consulta.

2. *Requerimientos no funcionales:*

RNF 1. Interfaz gráfica. Se debe contar con una interfaz gráfica que le permita al usuario utilizar las funcionalidades del programa.

RNF 2. Consulta eficiente. La consulta tiene que ser eficiente para 4 de los 5 atributos estadísticos del programa.

RNF 3. Cantidad masiva de datos. El programa debe estar en capacidad de funcionar correctamente usando cantidades masivas de datos.

RNF 4. Guardado de datos. El programa debe manejar su información por medio de archivos serializados.

FASE 2: Recopilación de la información necesaria:

Para implementar la solución planteada, es necesario definir las estructuras de datos de árboles y describir los tipos que se van a utilizar. En este caso vamos a darle un vistazo a los árboles binarios de búsqueda, a los árboles AVL y a los árboles rojos y negros.

Concepto de árbol: Un árbol es una estructura que nos permite organizar información de forma jerárquica. Esta tiene un punto de entrada que se denomina raíz, la cual da paso al resto de nodos que contienen la información. Cada nodo tendrá otros nodos como hijos en el caso que sea padre, y en caso que no, se le llamará hoja.

Árboles binarios de búsqueda: Es una estructura de tipo árbol que nos permite organizar la información para hacer búsquedas eficientes. En este caso, cada nodo va a tener hasta 2 hijos que se posicionan a su derecha o a su izquierda. Normalmente los de la izquierda son menores que el valor del nodo padre, mientras que los de la derecha son mayores. El proceso de búsqueda en un árbol balanceado debería tener una complejidad temporal de $O(\log n)$.

Árboles AVL: Por definición, estos son árboles binarios de búsqueda autobalanceados. Su principal diferencia con los comunes es el factor de balanceo de sus nodos, con el cual se puede determinar si el árbol está distribuido equitativamente o si se encuentra “cargado” hacia un lado.

Factor de balanceo: El factor de balanceo de un nodo perteneciente a un árbol AVL se puede calcular restando la altura del subárbol izquierdo con la del subárbol derecho o viceversa. Un ejemplo de ello es la función de balanceo $B(n)$:

$B(n) = H(n.der) - H(n.izq)$ donde:

n = Nodo al que se le calculará el factor de balanceo.

$H(n)$ = Altura del árbol n .

$n.der$ = Subárbol derecho del nodo n .

$n.izq$ = Subárbol izquierdo del nodo n .

Si el factor de balanceo se encuentra entre -1 y 1, el árbol está distribuido equitativamente. En caso contrario, este se encuentra desbalanceado en el subárbol izquierdo (cuando el factor es menor a -1) o en el subárbol derecho (cuando el factor es mayor a 1).

Un árbol AVL “cargado” se puede balancear por medio de rotaciones que se realizan a la derecha y a la izquierda. Dependiendo del tipo de desbalance se emplea una u otra rotación, o alguna combinación de estas. Finalmente, estos mantienen las mismas operaciones que un árbol binario de búsqueda, modificando únicamente la inserción y eliminación para mantener siempre el balance del árbol.

Árboles rojos y negros: Son un tipo de árbol binario de búsqueda autobalanceable. Sus nodos cuentan con un atributo adicional que hace referencia a su *color*, rojo o negro (que ocupa un bit de memoria adicional por nodo). Su balanceo consiste en unas normas con respecto al “color” de sus nodos, lo que ahorra pasos al momento de añadir o eliminar elementos a cambio de precisión de balanceo. Es por esto que son preferibles al momento de trabajar con datos que requieren de constante inserción o eliminación. Su altura es $O(\log n)$, donde n es el número de nodos en el árbol y sus operaciones tienen complejidad temporal $O(\log n)$.

FASE 3: Búsqueda de soluciones creativas:

En esta fase tomamos las consultas de jugadores y la modificación de datos de los mismos, como los problemas principales del programa. Luego, formulamos las posibles soluciones con la técnicas de lluvia de ideas o brainstorming y lista de atributos.

Consulta de jugadores:

Se desea consultar en una estructura de datos los jugadores que cumplan con un criterio de búsqueda especificado.

Alternativa 1. Búsqueda binaria:

Se plantea tener una serie de listas de jugadores ordenadas por cada criterio de consulta existente. Para que sea eficiente, se debe implementar un algoritmo de búsqueda binaria que toma en cuenta los parámetros/criterios buscados. Este proceso tiene una complejidad temporal de $O(\log n)$.

Alternativa 2. Tabla hash:

Tener una serie de tablas hash de jugadores que utilicen como llave los criterios de consulta existentes. Estos últimos determinan la tabla en la que se ejecuta el algoritmo de búsqueda. Para cada una se puede utilizar una función hash distinta. Este proceso tiene una complejidad temporal de $O(1)$ en el mejor de los casos, $O(n)$ en el peor de los casos.

Alternativa 3. Árboles binarios de búsqueda autobalanceados:

Utilizar árboles binarios de búsqueda de jugadores que estén ordenados con base a los criterios de consulta. Estos últimos determinan el árbol en el que se ejecuta el algoritmo de búsqueda. Dado que se requiere eficiencia, se debe implementar una forma de autobalanceo. Este proceso tiene una complejidad temporal de $O(\log n)$.

Alternativa 4. Búsqueda lineal:

Utilizar listas enlazadas para almacenar a los jugadores y buscarlos de acuerdo con los criterios de consulta existentes. Este proceso tiene una complejidad temporal de $O(n)$.

Modificación de datos:

Se busca realizar la modificación de los datos de un jugador partiendo de una búsqueda por su nombre y reorganización de la estructura de datos en la que se encuentra (En caso de que esta lo requiera).

Alternativa 1. Búsqueda binaria:

Tener una lista de jugadores ordenados por nombre e implementar un algoritmo de búsqueda binaria para encontrar el jugador que se desea modificar. El añadir y el eliminar del jugador modificado deben tener en cuenta este requisito de ordenamiento y deben insertar ordenadamente al jugador con los datos modificados. Este proceso tiene una complejidad temporal de $O(n)$ en el peor de los casos.

Alternativa 2. Tabla hash:

Tener una tabla hash de jugadores que tome sus nombres como llave, de tal forma que se pueda buscar a un jugador específico para modificar sus datos. Este proceso tiene una complejidad temporal de $O(1)$ en el mejor de los casos, $O(n)$ en el peor de los casos.

Alternativa 3. Árboles binarios de búsqueda autobalanceados:

Tener un árbol binario de jugadores que esté ordenado de acuerdo a sus nombres, permitiendo la búsqueda de estos. Se debe implementar una forma de autobalanceo para garantizar la eficiencia. Este proceso tiene una complejidad temporal de $O(\log n)$.

FASE 4: Transición de la formulación de ideas a diseños preliminares:

En esta fase se buscan descartar las ideas que no resultan factibles para la solución del problema o que son imposibles de implementar. Con esto en mente, descartamos:

A. Las alternativas 2 y 4 para la **consulta de jugadores**, ya que la primera no puede retornar varios de estos y la segunda no es eficiente. A continuación se ofrece una argumentación detallada sobre estas conclusiones:

-La tabla hash se implementa junto con una función que, dado las llaves de los elementos a agregar, se asignan las posiciones del arreglo. Si esta función retorna el mismo índice (es decir, se genera un conflicto), el elemento se añade en el próximo índice disponible. Esto implica que la tabla no necesariamente está ordenada, y tanto, no se pueden retornar todos los jugadores buscados. Por lo anterior, esta opción no cumple con el **RF 2** de **consulta de datos de jugadores**.

-El algoritmo de búsqueda de las listas enlazadas tiene una complejidad temporal de $O(n)$, ya que se tiene que mover por cada uno de los elementos existentes. Esto nos hace descartar la opción, ya que no cumple con el **RNF 2** de **consulta eficiente**.

B. La alternativa 2 para la **modificación de datos**, ya que no se puede retornar varios jugadores (se toma el mismo razonamiento que el de la tabla hash del punto anterior) y no es posible implementar la segunda de forma eficiente.

-El árbol binario de búsqueda no necesariamente agrupa sus nodos para facilitar el retorno de elementos con características en común. La búsqueda tan sólo permite descartar valores al compararlos con el que se está buscando. Esto implica que la implementación de búsqueda por caracteres sea ineficiente o impráctica. Lo anterior nos lleva a descartar la alternativa, ya que no cumple con el **RNF 2 de consulta eficiente**.

La revisión cuidadosa de las otras alternativas nos conduce a lo siguiente:

Consulta de jugadores:

Alternativa 1. Búsqueda binaria:

-En este caso, se emplea la búsqueda binaria para encontrar el valor límite que satisface el criterio de consulta. Para ello se divide la lista en dos y, dependiendo del valor de la mitad y del tipo de comparación, se busca tomando la parte izquierda o la parte derecha. Al finalizar se obtienen los índices en los que el arreglo cumple la condición dada.

-Para el criterio de “mayor” y “mayor e igual”, se evalúa el valor de la mitad y se verifica la condición. Si la cumple, entonces se busca tomando la izquierda. En caso contrario, se toma la derecha. El proceso termina cuando se encuentra el valor mínimo que satisface la condición.

-Para el criterio de “menor e igual” y “menor”, se evalúa el valor de la mitad y se verifica la condición. Si la cumple, entonces se busca tomando la derecha. En caso

contrario, se toma la izquierda. El proceso termina cuando se encuentra el valor máximo que satisface la condición.

-Para el criterio de “igual”, se evalúa el valor de la mitad y se verifica la condición. Si la cumple, entonces se desplaza por los elementos de la izquierda y de la derecha hasta que encuentre alguno que no sea igual. En caso contrario, escoge alguno de los dos lados y vuelve a buscar la mitad. El proceso termina cuando se encuentra el intervalo en el que el arreglo satisface la condición.

Alternativa 3. Árboles binarios de búsqueda autobalanceados:

-Para esta alternativa, se busca empezar por la raíz y verificar si este nodo cumple el criterio de consulta. En este proceso se tienen dos casos:

- A. Si se cumple que sea “mayor” y “mayor e igual”, entonces se ejecutará el algoritmo de forma recursiva para intentar añadir los nodos con recorrido de inorden. Si lo anterior no se da, entonces se ejecutará el algoritmo para intentar añadir los nodos de la derecha.
- B. Si se cumple el criterio de “menor e igual” y “menor”, entonces se ejecutará el algoritmo de forma recursiva para intentar añadir los nodos con recorrido de inorden inverso. Si lo anterior no se da, entonces se ejecutará el algoritmo para intentar añadir los nodos de la izquierda.
- C. Si se cumple el criterio de “igual”, entonces se añade el nodo y se sigue buscando por la izquierda o por la derecha dependiendo de cómo se haya implementado el árbol (para nuestro caso sería la izquierda). En caso contrario, se verifica si el valor del nodo es mayor o menor que el buscado, para luego irse a la derecha o a la izquierda respectivamente.

Modificación de datos:

Alternativa 1. Búsqueda binaria:

-En este caso, se emplea la búsqueda binaria para encontrar el intervalo en que los nombres inician con los caracteres especificados por el usuario. Para esto se busca el elemento de la mitad y se verifica la condición establecida. Si se cumple, entonces se empiezan a buscar los límites del intervalo (es decir, se busca tanto a la izquierda como a la derecha). En caso contrario, se divide la lista en dos y se busca por la derecha o por la izquierda. Al finalizar se obtienen los índices en los que el arreglo cumple la condición de caracteres.

-El resto del proceso es similar al explicado anteriormente.

Alternativa 3. Árboles binarios de búsqueda autobalanceados:

-En este caso es necesario buscar, eliminar, insertar y rebalancear el árbol por cada modificación que se haga. En este caso, todos los métodos diferentes a la búsqueda presentan un complejidad de $O(\log n)$ en el peor de los casos, sin embargo, dado que el árbol no agrupa los nombres de los jugadores, se tiene que buscar por cada uno de los nodos del árbol a los jugadores que cumplen con el nombre ingresado. Todo esto implica que este proceso tiene una complejidad de $O(n)$.

FASE 5: Evaluación de ideas y selección de la mejor solución:

En esta fase se quieren definir los criterios que nos permitirán evaluar y escoger alguna de las alternativas de solución planteadas. A continuación enumeramos los escogidos, junto con un valor que describe su deseabilidad:

Criterios de Evaluación:

Criterio A. Eficiencia: Se prefiere una solución con mejor eficiencia que las otras consideradas. Esta puede ser:

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Criterio B. Semejanza. Se prefiere una solución que se asemeje más a los requerimientos funcionales y a la problemática. Esta puede ser:

1. Semejante
2. No semejante

Evaluación:

Consulta de jugadores:

	Criterio A	Criterio B
Alternativa 1	$O(\log n)$	No semejante
Alternativa 3	$O(\log n)$	Semejante

Modificación de datos:

	Criterio A	Criterio B
Alternativa 1	$O(\log n)$	Semejante
Alternativa 3	$O(n)$	Semejante

Soluciones decididas:

Para el caso de la **consulta de jugadores** se tiene que ambas alternativas tienen la misma complejidad, sin embargo, la alternativa 3 se asemeja más debido a lo que se busca en el desarrollo de este sistema. Por ende se toma la alternativa 3 como solución al problema de las consultas de jugadores.

Por su parte, para la **modificación de datos** nos resulta más factible la alternativa 1 debido a que es la de menor complejidad, por ello la solución a la modificación de datos es la alternativa 1.

2. DISEÑOS TAD

TAD BinaryTree		
BinaryTree = {root = <rt>}		
{inv: $root.left.parameter \leq root.parameter \leq root.right.parameter \wedge root \in Node$ }		
Operaciones primitivas:		
-createBinaryTree		→ BinaryTree
-searchParameter	BinaryTree x Value	→ Node
-searchList	BinaryTree x Value x Criterion	→ List
-min	BinaryTree	→ List
-max	BinaryTree	→ List
-isEmpty	BinaryTree	→ Boolean
-add	BinaryTree x Value	→ AVLTree
-delete	BinaryTree x Value	→ Node

createBinaryTree()

Crea un árbol binario.

{pre: True}

{post: binaryTree: {root = <null>}}

searchParameter(tree, param)

Busca un elemento en el árbol binario dado un parámetro específico. Se retorna el nodo *node* que tenga a *parameter* como elemento.

{pre: $tree \neq \text{null} \wedge \text{parameter} \neq \text{null}$ }

{post: Si se encuentra el elemento: $node = \{\text{element} = \langle e \rangle, \text{left} = \langle l \rangle, \text{right} = \langle r \rangle, \text{parent} = \langle p \rangle, \text{parameter} \langle \text{param} \rangle\}$

Si no se encuentra: null }

searchList(tree, value, criterion)

Busca un listado de nodos que cumplan con un criterio asociado a un valor. Se retorna el listado *list* que contiene todos los nodos con valores que cumplen con la condición.

{pre: $tree \neq \text{null} \wedge \text{value} \neq \text{null} \wedge \text{criterion} \neq \text{null}$ }

{post: $\langle \text{list} \rangle$ }

min(tree)

Retorna el listado de elementos que tiene el mínimo parámetro *min* del árbol binario.

{pre: $tree \neq \text{null} \wedge \text{tree.root} \neq \text{null}$ }

{post: $\langle e \rangle$ }

max(tree)

Retorna el listado de elementos que tiene el máximo parámetro *max* del árbol binario.

{pre: $tree \neq \text{null} \wedge \text{tree.root} \neq \text{null}$ }

{post: $\langle e \rangle$ }

isEmpty(tree)

Verifica si el árbol está vacío o no.

{pre: $tree \neq \text{null}$ }

{post: True si $\text{tree.root} == \text{null}$ }

add(tree, element, parameter)

Añade un nodo con valor *element* y parámetro *parameter* al árbol *tree*.

{pre: $tree \neq \text{null} \wedge \text{element} \neq \text{null} \wedge \text{parameter} \neq \text{nul}\}$

{post: $tree: \{\text{root} = \langle r \rangle\}$ }

delete(tree, element, parameter)

Elimina el primer elemento con parámetro *parameter* del árbol *tree*. Se retorna el nodo donde se eliminó el elemento en caso de que el listado de elementos del nodo no esté vacío, caso contrario, se retorna el padre del nodo eliminado o el sucesor del mismo.

{pre: $tree \neq \text{null}$ }

{post: $\text{node: \{element = \langle e \rangle, left = \langle l \rangle, right = \langle r \rangle, parent = \langle p \rangle, parameter \langle parameter \rangle\}}$ }

TAD AVLTree

$\text{AVLTree} = \{\text{root} = \langle \text{rt} \rangle\}$

{inv: $\text{root.left.element} \leq \text{root.element} \leq \text{root.right.element} \wedge \text{root} \in \text{Node} ; \text{AVLTree} \subset \text{BinaryTree}\}$

Operaciones primitivas:

-createAVLTree		→ AVLTree
-searchParameter	AVLTree x Value	→ Node
-searchList	AVLTree x Value x Criterion	→ List
-min	AVLTree	→ Element
-max	AVLTree	→ Element
-isEmpty	AVLTree	→ Boolean
-add	AVLTree x Value	→ AVLTree
-delete	AVLTree x Value	→ Node
-leftRotate	AVLTree x Node	→ AVLTree
-rightRotate	AVLTree x Node	→ AVLTree
-balance	AVLTree x Node	→ AVLTree

createAVLTree()

Crea un árbol AVL *tree* vacío.

{pre: True}

{post: avlTree: {root = null} }

searchParameter(avlTree, parameter)

Busca un elemento en el árbol AVL dado un parámetro específico. Se retorna el nodo *node* que tenga a *parameter* como elemento.

{pre: avlTree \neq null \wedge parameter \neq null }

{post: Si se encuentra el elemento: *node* = {element = <e>, left = <l>, right = <r>, parent = <p>}

Si no se encuentra: null}

searchList(avlTree, value, criterion)

Busca un listado de nodos que cumplan con un criterio asociado a un valor. Se retorna el listado *list* que contiene todos los nodos con valores que cumplen con la condición.

{pre: tree \neq null \wedge value \neq null \wedge criterion \neq null}

{post: <list>}

min(avlTree)

Retorna el listado e de elementos que tiene el mínimo parámetro *min* del árbol AVL.

{pre: avlTree \neq null \wedge avlTree.root \neq null}

{post: <e>}

max(avlTree)

Retorna el listado e de elementos que tiene el máximo parámetro *max* del árbol AVL.

{pre: avlTree \neq null \wedge avlTree.root \neq null}

{post: <e>}

isEmpty(avlTree)

Verifica si el árbol está vacío o no.

{pre: avlTree \neq null }

{post: True si avlTree.root == null}

add(avlTree, value)

Añade un nodo con valor *value* al árbol AVL *avlTree*. Posteriormente, se rebalancea el árbol.

{pre: avlTree \neq null \wedge value \neq null}

{post: avlTree : {root = <r>}}

delete(avlTree, element, parameter)

Elimina el primer elemento con parámetro *parameter* del árbol AVL *avlTree*. Se retorna el nodo donde se eliminó el elemento en caso de que el listado de elementos del nodo no esté vacío, caso contrario, se retorna el padre del nodo eliminado o el sucesor del mismo. Posteriormente, se rebalancea el árbol.

{pre: avlTree \neq null }

{post: node: {element = <e>, left = <l>, right = <r>, parent = <p>, parameter<parameter>}}

leftRotate(avlTree, node)

Hace una rotación a la izquierda en el nodo *node*.

{pre: avlTree \neq null \wedge node \neq null }

{post: avlTree: {root = <r>}}

rightRotate(avlTree, node)

Hace una rotación a la derecha en el nodo *node*.

{pre: avlTree \neq null \wedge node \neq null}

{post: avlTree: {root = <r>}}

TAD Node

Node= {element = <e>, left = <l>, right = <r>, parent=<p>, parameter = <par>}

{inv: $element \neq null \wedge left, right, parent \in Node ; element \in List$ }

Operaciones primitivas:

-createNode	Value	→ Node
-getElement	Node	→ Value
-getLeft	Node	→ Value
-getRight	Node	→ Value
-getParameter	Node	→ Value
-setElement	Node x Value	→ Node
-setLeft	Node x Node	→ Node
-setRight	Node x Node	→ Node
-setParent	Node x Node	→ Node
-setParameter	Node x Value	→ Node
-getSuccessor	Node	→ Node
-getPredecessor	Node	→ Node
-getBalance	Node	→ Value
-getHeight	Node	→ Value
-addElement	Node x Value	→ Node

createNode(e)

Crea un nodo *node* que tiene una lista *element* que contiene el valor e.

{pre: True}

{post: node: {element = <e>, left = null, right = null, parent=null, parameter = null} }

getElement(node)

Retornar el listado *element* que almacena el nodo

{pre: node≠ null}

{post: *element* }

getLeft(node)

Retornar el hijo izquierdo *left* del nodo

{pre: node ≠ null}

{post: *left*}

getRight(node)

Retornar el hijo derecho *right* del nodo

{pre: node ≠ null}

{post: *right*}

getParameter(node)

Retornar el parámetro *par* del nodo

{pre: node ≠ null}

{post: *par*}

getParent(node)

Retornar el padre *parent* del nodo

{pre: node ≠ null}

{post: *parent*}

setElement(node, v)

Establece el listado *v* como el nuevo *element* de *node*.

{pre: node ≠ null ∧ *v* ≠ null}

{post: *node*: {*element* = <*v*>, *left* = <*l*>, *right* = <*r*>, *parent* = <*p*>, *parameter* = <*par*>}}

setLeft(node, leftNode)

Establece *leftNode* como el nuevo hijo izquierdo *left* de *node*.

{pre: node ≠ null ∧ *leftNode* ≠ null}

```
{post: node:{element =<v>, left=<leftNode>, right=<r>, parent=<p>,
parameter = <par>}}
```

setRight(node, rightNode)

Establece *rightNode* como el nuevo hijo derecho *right* de *node*.

```
{pre: node≠ null ∧ rightNode ≠ null}
```

```
{post: node:{element =<v>, left=<l>, right=<rightNode>, parent=<p>,
parameter = <par>}}
```

setParent(node, parentNode)

Establece *parentNode* como el nuevo padre *parent* de *node*.

```
{pre: node≠ null ∧ parentNode ≠ null}
```

```
{post: node:{element =<v>, left=<l>, right=<rightNode>,
parent=<parentNode>, parameter = <par>}}
```

setParameter(node, param)

Establece *param* como el nuevo parámetro *par* de *node*.

```
{pre: node≠ null ∧ param ≠ null}
```

```
{post: node:{element =<v>, left=<l>, right=<rightNode>, parent=<p>,
parameter = <param>}}
```

getSuccessor(node)

Busca el mayor elemento *successor* entre los hijos del hijo izquierdo de *node*.

```
{pre: node≠ null}
```

```
{post: succesor:{element =<e>, left=<l>, right=<rightNode>, parent=<p>,
parameter = <par>}}
```

getPredecessor(node)

Busca el menor elemento *predecessor* entre los hijos del hijo derecho de *node*.

{pre: node ≠ null}

{post: *predecessor* : {element = <e>, left = <l>, right = <rightNode>, parent = <p>, parameter = <par>}}

getBalance(node)

Calcula el balance de un *node*. Se le resta a la altura del subárbol derecho (hijo derecho de *node*) la altura del subárbol izquierdo (hijo izquierdo de *node*).

{pre: node ≠ null}

{post: balance = }

getHeight(node)

Calcula la altura *height* de un *node*. En caso de que no tenga hijos, su altura es -1.

{pre: node ≠ null}

{post: <height> si tiene hijos
-1 si no tiene hijos}

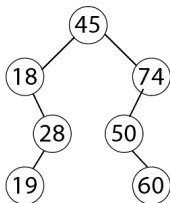
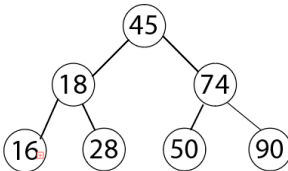
3. DISEÑO DEL DIAGRAMA DE CLASES

El diagrama de clases se encuentra en la carpeta docs del repositorio, junto con este documento.

4. DISEÑO DE LOS CASOS DE PRUEBA

//Al hablar de valor nos referimos al parámetro de los nodos "SearchParameter".

Nombre	Clase	Escenario
BTTSC1	BinaryTreeTest	La raíz del árbol se encuentra vacía
BTTSC2	BinaryTreeTest	La raíz tiene un valor igual a 7, mas no tiene hijos
BTTSC3	BinaryTreeTest	La raíz cuenta con valor igual a 7 y un hijo derecho con valor igual a 9
BTTSC4	BinaryTreeTest	La raíz cuenta con un valor igual a 7 y un hijo izquierdo con valor igual a 5
BTTSC5	BinaryTreeTest	El árbol cuenta con 7 nodos. La raíz cuenta con un valor igual a 9, un hijo izquierdo igual a 5, que a su vez posee un hijo izquierdo igual a 3 y uno derecho igual a 7. Por el lado derecho de la raíz hay un nodo con valor 13, nodo izquierdo 11 y derecho igual a 15.
BTTSC6	BinaryTreeTest	La raíz cuenta con un valor igual a 7 y un hijo izquierdo con valor igual a 5, que a su vez tiene otro hijo izquierdo igual a 3

Nombre	Clase	Escenario
AVLTSC1	AVLTreeTest	La raíz del árbol se encuentra vacía
AVLTSC2	AVLTreeTest	La raíz con un valor igual a 9, un hijo izquierdo igual a 5 y un hijo derecho igual a 11
AVLTSC3	AVLTreeTest	 <pre> graph TD 45((45)) --- 18((18)) 45 --- 74((74)) 18 --- 28((28)) 18 --- 19((19)) 74 --- 50((50)) 74 --- 60((60)) </pre>
AVLTSC4	AVLTreeTest	 <pre> graph TD 45((45)) --- 18((18)) 45 --- 74((74)) 18 --- 16((16)) 18 --- 28((28)) 74 --- 50((50)) 74 --- 90((90)) </pre>

AVLTSC5	AVLTreeTest	<pre> graph TD 45((45)) --> 18((18)) 45 --> 74((74)) 18 --> 16((16)) 74 --> 60((60)) 74 --> 90((90)) 90 --> 95((95)) </pre>
AVLTSC6	AVLTreeTest	<pre> graph TD 45((45)) --> 18((18)) 45 --> 74((74)) 18 --> 16((16)) 74 --> 60((60)) 74 --> 90((90)) 60 --> 55((55)) 60 --> 65((65)) </pre>
AVLTSC7	AVLTreeTest	<pre> graph TD 45((45)) --> 18((18)) 45 --> 90((90)) 18 --> 16((16)) 16 --> 10((10)) 90 --> 65((65)) 65 --> 60((60)) 60 --> 55((55)) 60 --> 65((65)) </pre>
AVLTSC8	AVLTreeTest	<pre> graph TD 45((45)) --> 18((18)) 45 --> 74((74)) 18 --> 16((16)) 74 --> 65((65)) 74 --> 90((90)) 65 --> 60((60)) 60 --> 55((55)) 60 --> 64((64)) </pre>
AVLTSC9	AVLTreeTest	root = 9. right = 15

Nombre	Clase	Escenario
FIBASC1	FIBASRequestsTest	Un elemento de tipo FIBAResquest, con todas sus listas y árboles vacíos.

FIBASC2	FIBASRequestsTest	Un elemento de tipo FIBARequest, con todas sus listas y árboles con un elemento igual a Player(Juancaire,23,NOCHU,5.5,4.3,2.8,7.5,6)
---------	-------------------	--

Clase	Método	Escenario	Valores de entrada	Resultado
-------	--------	-----------	--------------------	-----------

Objetivo de la Prueba: Verificar que el método add, agrega nodos en las posiciones adecuadas dentro del árbol.

Clase	Método	Escenario	Valores de Entrada	Resultado
Binary Tree	add	BTTSC1	valor = 7	La raíz del árbol no es nula
Binary Tree	add	BTTSC2	valor = 9	El hijo derecho del nodo raíz tiene un valor igual a 9
Binary Tree	add	BTTSC2	valor = 5	El hijo izquierdo del nodo raíz tiene un valor igual a 5
Binary Tree	add	BTTSC2	valor = 7	El hijo izquierdo del nodo raíz tiene un valor igual a 7
Binary Tree	add	BTTSC3	valor = 8	El hijo izquierdo del hijo derecho de la raíz posee un valor igual a 8
Binary Tree	add	BTTSC4	valor = 6	El hijo derecho del hijo izquierdo de la raíz posee un valor igual a 6
Binary Tree	add	BTTSC5	valor = 15	El hijo izquierdo, del hijo derecho del hijo derecho de la raíz posee un valor igual a 15

Objetivo de la Prueba: Verificar que el método search devuelve el valor apropiado del árbol. En caso de que el valor no se encuentre, se retornará nulo.

Clase	Método	Escenario	Valores de Entrada	Resultado
Binary Tree	search	BTTSC1	valor = 5	null
Binary Tree	search	BTTSC2	valor = 7	nodo con valor 7
Binary Tree	search	BTTSC2	valor = 5	null
Binary Tree	search	BTTSC5	valor = 10	null

Objetivo de la Prueba: Verificar que el método delete borre apropiadamente un nodo del árbol, teniendo en cuenta los nodos hijos del mismo.

Clase	Método	Escenario	Valores de Entrada	Resultado
Binary Tree	delete	BTTSC2	valor = 7	El nodo raíz tiene un valor nulo
Binary Tree	delete	BTTSC3	valor = 9	El nodo raíz ahora tiene su hijo derecho apuntando a un valor nulo.
Binary Tree	delete	BTTSC4	valor = 5	El nodo raíz ahora tiene su hijo izquierdo apuntando a un valor nulo.
Binary Tree	delete	BTTSC5	valor = 9	El nodo raíz tiene valor = 7. El lado derecho del árbol se mantiene igual, pero el izquierdo ahora cuenta con un hijo cuyo valor sigue siendo 5 y un solo hijo (izquierdo) con valor = 3

Objetivo de la Prueba: Verificar que el método searchList retorne los elementos que cumplan con el criterio de búsqueda asignado.

Clase	Método	Escenario	Valores de Entrada	Resultado
Binary Tree	searchList	BTTSC3	parámetro = 10. criterio = Menor	{9,7}
Binary Tree	searchList	BTTSC4	parámetro = 10. Criterio = Menor	{7,5}
Binary Tree	searchList	BTTSC5	parámetro = 12. Criterio = menor o igual	{11, 9, 7, 5, 3}
Binary Tree	searchList	BTTSC5	parámetro = 10. Criterio= igual	null
Binary Tree	searchList	BTTSC5	parámetro = 7. Criterio = mayor	{9, 11, 13, 15}
Binary Tree	searchList	BTTSC5	Parámetro = 7. Criterio = mayor o igual	{7, 9, 11, 13, 15}

Objetivo de la Prueba: Verificar que el método getMinimun retorna el elemento con el menor valor asociado en el árbol.

Clase	Método	Escenario	Valores de Entrada	Resultado
Binary Tree	getMinimun	BTTSC1	Ninguno	null
Binary Tree	getMinimun	BTTSC2	Ninguno	7
Binary Tree	getMinimun	BTTSC4	Ninguno	5

Binary Tree	getMinimum	BTTSC5	Ninguno	3
-------------	------------	--------	---------	---

Objetivo de la Prueba: Verificar que el método getMaximun retorna el elemento con el mayor valor asociado en el árbol.

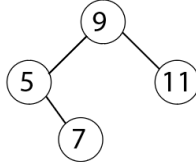
Clase	Método	Escenario	Valores de Entrada	Resultado
Binary Tree	getMaximun	BTTSC1	Ninguno	null
Binary Tree	getMaximun	BTTSC2	Ninguno	7
Binary Tree	getMaximun	BTTSC4	Ninguno	7
Binary Tree	getMaximun	BTTSC5	Ninguno	15

Objetivo de la Prueba: Verificar que el método isEmpty retorna falso si el árbol se encuentra lleno y verdadero en caso de estar vacío.

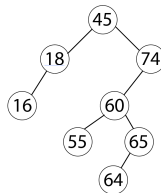
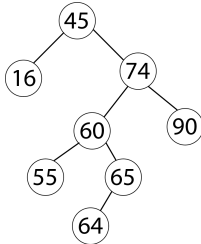
Clase	Método	Escenario	Valores de Entrada	Resultado
Binary Tree	isEmpty	BTTSC1	Ninguno	true
Binary Tree	isEmpty	BTTSC2	Ninguno	false
Binary Tree	isEmpty	BTTSC5	Ninguno	false

AVL

Objetivo de la Prueba: Verificar que el método add agregue los nuevos elementos al árbol y se asegure de balancear correctamente de ser necesario.

Clase	Método	Escenario	Valores de Entrada	Resultado
AVLTree	add	AVLTSC1	Valor = 7	Árbol con raíz igual a 7
AVLTree	add	AVLTSC2	valor = 7	
AVLTree	add	AVLTSC4	valor = 55	el nodo con valor 50 del árbol ahora tiene hijo derecho con valor 55. El resto del árbol permanece igual
AVLTree	add	AVLTSC9	valor = 30	9, 15, 30

Objetivo de la Prueba: Verificar que el método delete, remueve el nodo especificado (si lo hay) y deja el árbol balanceado.

Clase	Método	Escenario	Valores de Entrada	Resultado
AVLTree	delete	AVLTSC1	valor = 7	null
AVLTree	delete	AVLTSC6	valor = 90	parent = 74 
AVLTree	delete	AVLTSC6	valor = 18	

AVLTree	delete	AVLTSC6	valor = 74	<pre> graph TD 45((45)) --- 18((18)) 45 --- 90((90)) 18 --- 16((16)) 90 --- 60((60)) 60 --- 55((55)) 60 --- 65((65)) 65 --- 64((64)) </pre>
---------	--------	---------	------------	---

Objetivo de la Prueba: verificar que el left rotate asigne nueva adecuadamente los nodos según las normas de los AVL

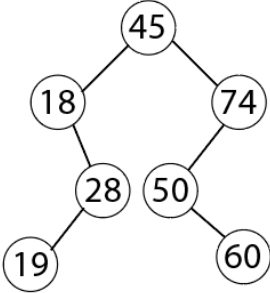
Clase	Método	Escenario	Valores de Entrada	Resultado
AVLTree	leftRotate	AVLTSC7	node = 65	<pre> graph TD 45((45)) --- 18((18)) 45 --- 74((74)) 18 --- 16((16)) 74 --- 65((65)) 74 --- 90((90)) 65 --- 60((60)) 60 --- 55((55)) 60 --- 64((64)) </pre>

Objetivo de la Prueba: verificar que el right rotate asigne nueva adecuadamente los nodos según las normas de los AVL

Clase	Método	Escenario	Valores de Entrada	Resultado
AVLTree	rightRotate	AVLTSC7	node = 90	<pre> graph TD 45((45)) --- 18((18)) 45 --- 65((65)) 18 --- 16((16)) 65 --- 60((60)) 65 --- 90((90)) 16 --- 10((10)) 60 --- 55((55)) 60 --- 65((65)) </pre>

Objetivo de la Prueba: Verificar que el árbol se balancee apropiadamente

Clase	Método	Escenario	Valores de Entrada	Resultado
-------	--------	-----------	--------------------	-----------

AVLTree	balance	AVLTSC3	Balance por su creación	 <pre> graph TD 45((45)) --- 18((18)) 45 --- 74((74)) 18 --- 19((19)) 18 --- 28((28)) 28 --- 50((50)) 50 --- 60((60)) </pre>
---------	---------	---------	-------------------------	--

Objetivo de la Prueba: Verificar que los valores de los factores de balanceo de cada nodo sea correcto

Clase	Método	Escenario	Valores de Entrada	Resultado
AVLTree	getBalance	AVLTSC6	nodo con valor 45	1
AVLTree	getBalance	AVLTSC6	nodo con valor 74	-1
AVLTree	getBalance	AVLTSC6	nodo con valor 65	0

Objetivo de la Prueba: Verificar que las alturas de los nodos es la apropiada

Clase	Método	Escenario	Valores de Entrada	Resultado
AVLTree	getHeight	AVLTSC6	nodo con valor 45	3
AVLTree	getHeight	AVLTSC6	nodo con valor 74	2
AVLTree	getHeight	AVLTSC6	nodo con valor 65	0

Objetivo de la Prueba: Verificar que los jugadores se están añadiendo apropiadamente a los árboles

Clase	Método	Escenario	Valores de Entrada	Resultado
FIBARerequests	addPlayer	FIBASC1	Player = ("Jose", 34, "MECHU", 6, 3, 5, 8, 9)	Todos los árboles cuentan con una raíz igual al elemento añadido
FIBARerequests	addPlayer	FIBASC2	Player = ("Jose", 34, "MECHU", 6, 3, 5, 8, 9)	La lista de jugadores tiene ahora dos elementos, siendo el elemento recién agregado el segundo. Todos los árboles cuentan con dos elementos, con player estando como: en score, hijo derecho; en assist, como hijo izquierdo; en blocks como derecho; en steals como derecho y en bounces como derecho