

INFORME DE INGENIERÍA
LABORATORIO UNIDAD 3

MARIA CAMILA LENIS RESTREPO
JUAN SEBASTIAN PALMA GARCÍA
JAVIER ANDRÉS TORRES REYES

ALGORITMOS Y ESTRUCTURAS DE DATOS

2018-2

INFORME DE INGENIERÍA

Paso 1: Identificación del problema

Definición del problema

Manejar información de gran tamaño de manera eficiente a partir de árboles binarios balanceados para un programa en cuestión.

Justificación

El manejo de grandes volúmenes de información es de vital importancia para los sistemas de información robustos. De esta manera, resulta evidente que la eficiencia depende mucho de la manera en que estén guardados los datos, su orden y su accesibilidad. Por lo tanto, por medio de árboles binarios de búsqueda se pretende demostrar la eficiencia al tener árboles balanceados y no balanceados para el manejo de una base de datos de más de 200000 jugadores. En este laboratorio se pretende manejar la información de jugadores de basquetbol en específico, pero esta información no es absoluta y puede cambiar, por ende, resulta importante entender el manejo que debe dársele a esta para que, aún con los cambios en la ejecución del programa, su tiempo de respuesta sea el adecuado, teniendo en cuenta que se trata de aplicaciones pertinentes en la vida cotidiana de un desarrollador.

Requerimientos funcionales

1. Ingresar datos de los jugadores ya sea de manera masiva (archivos cvs) o por medio de la interfaz. Estos datos deben incluir el nombre del jugador, edad, equipo y 5 rubros estadísticos sean: puntos por partido, rebotes por partido, asistencias por partido, robos por partido y bloqueos por partido.
2. Modificar los datos de un jugador en específico dado su nombre. Si no existe, el programa mostrará un mensaje de advertencia. Los datos por modificar pueden ser nombre, edad, equipo y los 5 rubros estadísticos. Puede modificar uno o varios datos en una sola consulta.
3. Eliminar un jugador dando su nombre. Si este no existe, en el programa se mostrará un mensaje de advertencia.
4. Realizar consultas de jugadores utilizando una categoría. La búsqueda debe realizarse sobre los atributos del jugador, y puede o no ser una igualdad (igual a, mayor o menor que). Se debe retornar el jugador o grupo de jugadores que cumplan con las condiciones dadas, o un mensaje de advertencia si no existe ninguno.
5. Mostrar el tiempo que toma la consulta, ya sea que se haya realizado de dos maneras (árboles no balanceado y balanceados) o solo de una (árboles binarios balanceados) para todas las consultas.

Requerimientos no funcionales

- Búsqueda eficiente para categorías de 4 rubros estadísticos, por medio de índices.
 - Dos tipos de consulta deben ser con árboles AVL y los otros dos con árboles rojinegros. Donde cada árbol guardará el valor del índice (atributo) y la posición de este dato en el disco.
 - Deben existir dos consultas para árboles binarios no balanceados. Esto quiere decir que una consulta debe ser realizada por árboles AVL y por ABB y la otra por árboles rojinegros y ABB.
 - La complejidad no puede ser lineal.

- Manejo de software de gran tamaño: El programa deberá contener por lo menos 200000 datos válidos sobre jugadores.

Paso 2: Recopilación de la información

Baloncesto

El baloncesto es un deporte que consiste en dos equipos de cinco jugadores cada uno que juegan dentro de una cancha rectangular, donde los jugadores tienen que encestar un balón en una cesta/canasta para anotar puntos. El juego consiste en 4 tiempos de 10 minutos, y cada vez que el balón sale del terreno de juego el tiempo se detiene hasta que se reanude la jugada. La puntuación en el juego varía dependiendo del tipo de lanzamiento, el tiro libre tiene un valor de un punto, el tiro dentro del área demarcada vale por dos puntos y el tirar desde afuera del área demarcada son tres puntos. El baloncesto se origina en E.E.U.U, inventado por James Naismith en Massachusetts durante la temporada de invierno y los jóvenes no podían salir a jugar. El baloncesto empezó como dos canastas elevadas a 10 pies de altura y la pelota utilizada para jugar era un balón de fútbol, ahora el baloncesto es uno de los deportes más reconocidos mundialmente.

Sobre rubros estadísticos del baloncesto

Los rubros estadísticos más usados para definir las habilidades de un jugador de baloncesto son los siguientes:

Abreviación	Significado
#G	Número de partidos jugados
MPG: Minutes per Game	Minutos por partido
PTS: Points	Puntos anotados
FGM-FGA: Field Goals Made- Attempted	Tiros logrados/tiros intentados
FG%: Field Goal Percentage	Porcentaje de anotaciones
3PM-3PA: 3pointer made-Attempted	Triples logrados/intentados
3P%: 3 Pointer Percentage	Porcentaje de triples
FTM-FMA: Free Throws Made-Attempted	Tiros libres anotados/intentados
FT%: Free Throws Percentage	Porcentaje de tiros libres
PPG: Points per game	Puntos por partido
RPG: Rebounds per game	Rebotes por partido
APG: Assists per game	Asistencias por partido
SPG: Steals per game	Robos por partido
BPG: Blocks per game	Bloqueos por partido

Cabe resaltar que dependiendo de la posición del equipo en la que se encuentre el jugador así mismo pueden variar sus estadísticos. Por ejemplo, un poste no tendrá la misma cantidad de triples que un alero o armador, y un armador no recuperará más rebotes que un poste.

Sobre bases de datos de jugadores

Las bases de datos utilizadas como ejemplos en este trabajo fueron tomadas de:

- <https://data.world/jgrosz99/nba-player-data-1978-2016>

Sobre árboles binarios de búsqueda

Es una estructura de datos que se compone de una raíz, nodo inicial del árbol, y dos hijos que se convierten en subárboles a su izquierda y derecha. Para que un árbol binario exista, es necesario implementar el nodo raíz.

Diferencias de un árbol binario lleno y completo

Lleno: hace referencia a cuando los nodos de cada nivel se encuentran con sus dos hijos o con ninguno.

Completo: hace referencia a todas las hojas en el nivel n y $n-1$ tienen un hijo izquierdo y a su vez uno derecho o ninguno.

El árbol lleno es completo, mientras que el completo no es lleno.

Propiedades del Árbol binario

Sea K un nivel cualquiera del árbol binario, el máximo de nodos en el árbol es de 2^k

Sea K la altura del árbol binario, el máximo número de nodos disponibles es de $2^{k+1}-1$

Recorridos

- PreOrden: este recorrido consiste en recopilar la información desde el nodo raíz, luego los nodos de la izquierda y por último, los nodos de la derecha.
 - Toma la raíz, luego izquierda y luego derecha.
- InOrden: este recorrido se encarga de recorrer los nodos de izquierda, centro(raíz), y por último el de la derecha.
- PostOrden: Se recorren los datos de derecha a izquierda.

Tipos de Árboles

- Árbol de Expresión
- Árbol Binario de Búsqueda

Sobre árboles binarios de búsqueda balanceados

Árboles AVL

Un árbol AVL es un árbol binario de búsqueda que tiene la condición de que la altura de los subárboles presentes en él tienen una diferencia de altura menor o igual a 1. El árbol AVL tiene la característica de que todas sus alturas son equivalentes a $O(\log n)$ lo que significa que mantiene una consistencia gracias a su característica de equilibrio, o en otras palabras, su capacidad de auto balancearse.

Tipos de Rotaciones

- Simples
- Dobles

El Auto Balanceo del árbol consiste en realizar una serie de acciones finitas que tienen la función de modificar la forma de un árbol siempre y cuando se agreguen datos o se eliminen al mismo. Esta acción de auto balanceo consiste en una serie de rotaciones simples o dobles que se aplican dentro de los métodos de inserción y eliminación mediante un llamado.

Árboles Rojinegros

Un árbol rojo y negro consiste en cuatro características especiales, que definen por obligación si es o no un árbol rojo y negro.

- Es un árbol estricto, todo nodo nulo se toma en las operaciones; todas las hojas del árbol tienen que ser nulas.
- Cada nodo tiene que ser rojo o negro, no otro color
- Todos los nodos nulos tienen que ser negros
- La raíz del árbol obligatoriamente es negra, esto ayuda a simplificar las operaciones

Condiciones para la funcionalidad del árbol rojo y negro

- Todo nodo rojo tiene por obligación dos hijos de color negro
- Todo camino de la raíz hasta una hoja debe de tener la misma cantidad de nodos negros en total

La inserción de nodos en este árbol consiste en la creación de un nodo aparte de color rojo con la información que deseamos que contenga, y se inserta de igual manera a un árbol binario. Luego el método al insertar el nuevo nodo llama al método de rebalanceo de inserción y este se encarga de modificar los colores de la rama para que estas sean consistentes en la altura de nodos de color negro. La eliminación sirve de la misma manera que la de los árboles binarios, solo que se le agrega el método de rebalanceo de eliminación.

El método de rebalanceo de inserción tiene tres casos:

- Caso 1: Nodo hermano del padre (Tio) rojo
- Caso 2: Nodo hermano del padre (Tio) negro y el nodo agregado esta a la derecha del padre
- Caso 3: Nodo hermano del padre (Tio) negro y el nodo agregado esta a la izquierda del padre

El método de rebalanceo de eliminación tiene cinco casos y dos casos triviales:

- Caso 1: Nodo borrado tiene hermano rojo y padre negro
- Caso 2: Nodo borrado tiene hermano negro no nulo, sobrinos negros, padre negro
- Caso 3: Nodo borrado tiene hermano negro no nulo, sobrino negro y padre rojo
- Caso 4: Nodo borrado tiene hermano negro no nulo, sobrino rojo/negro, padre de cualquier color
- Caso 5: Nodo borrado tiene hermano negro no nulo, sobrinos cualquiera/rojo, padre cualquier color
- Casos triviales
 - Nodo borrado es rojo: No requiere ningún ajuste
 - El hijo del nodo borrado es rojo, se le cambia el color a negro

Paso 3: Búsqueda de soluciones creativas

Para este problema en cuestión se identificaron dos subproblemas: el manejo de la información de los jugadores y la eficiencia de las consultas.

Para el manejo de información de los jugadores hay que tener en cuenta que se debe contar con las de 200000 jugadores y debe accederse a ellos de manera eficiente. Por lo tanto, se tienen las siguientes alternativas:

- Conectar el aplicativo a una base de datos en SQL para acceder a la información de los jugadores.
- Tomar los datos de los jugadores de un dataset y generar aleatoriamente los restantes para cumplir con el objetivo de más de 200000 jugadores.
- La misma que la anterior, pero guardando la información de cada jugador en un archivo de texto.
- Guardar la información de cada jugador en un árbol binario.

Para el manejo de la eficiencia de las consultas se deben cumplir con las restricciones de los requerimientos que son: 4 consultas sobre rubros estadísticos, 2 con árboles AVL, 2 con árboles rojinegros y adicional 2 con ABB para realizar la comparación con los árboles binarios balanceados. En síntesis, las ideas planteadas no pueden salirse del marco de que las consultas deben ser tratadas con árboles, por lo tanto, las ideas propuestas van en torno a la elección del tipo de árbol a usar para cada criterio.

Cada jugador debe contener 5 rubros estadísticos: puntos por partido, rebotes por partido, asistencias por partido, robos por partido, bloqueos por partido. Los 4 atributos para los cuales la búsqueda debe ser eficiente serán los 4 últimos, debido a que los puntos con enteros y los otros cuatro pueden ser decimales y resulta de mayor pertinencia eficiencia en su búsqueda.

Paso 4: Transición de ideas a los diseños preliminares

Manejo de información de gran tamaño:

Para este problema se ha decidido descartar la primera opción debido a

Entonces quedan las otras tres opciones que serían:

- Tomar los datos de los jugadores de un dataset y generar aleatoriamente los restantes para cumplir con el objetivo de más de 200000 jugadores y manejarlos en un archivo cvs, donde cada línea corresponde a un jugador.
- La misma que la anterior, pero guardando la información de cada jugador en un archivo de texto. Es decir, al inicio del programa se recorre todo el archivo cvs para ir generando los archivos de texto de cada jugador (jugadorx.txt) y guardarlos en un directorio llamado data, y así cada nodo de los árboles tendría el índice a buscar y el valor del jugador para buscarlo directamente en la carpeta data.
- Guardar la información de cada jugador en un árbol binario. Es decir, cada nodo tiene un objeto Jugador que tiene como atributos el nombre, equipo y los 5 rubros estadísticos de jugador.

Eficiencia de las consultas:

Los cuatro datos elegidos son de la misma naturaleza y no interfieren de manera significativa en el tipo de estructura que los almacena. Es decir, no hay restricciones para que la consulta con cierto tipo de dato debe realizarse con un árbol es específico. Por lo tanto, la elección de cuáles son los 2 rubros por consultar con árboles AVL, los dos rubros con rojinegro y los 2 rubros con ABB puede ser de manera aleatoria.

Cabe resaltar que de igual forma se debe informar al usuario del tiempo tomado con cada árbol, y en caso de que la consulta sea realizada por dos árboles (uno balanceado y uno no balanceado) debe

tenerse la comparación de los tiempos, para así tener una idea de la diferencia entre tener garantizado un $O(\log n)$ y tener abierta la posibilidad de que se convierta en $O(n)$.

Paso 5: Evaluación o selección de la mejor solución (Criterios y selección)

Manejo de información de gran tamaño

Criterio A: Organización de la información. Los datos están ordenados de manera que es fácil entender a quién pertenecen y lo que significan (a qué rubro se refiere).

- [3] Los datos se encuentran separados por índices, como en un diccionario
- [2] Los datos se encuentran en los atributos del jugador.
- [1] Los datos se encuentran enlistados

Criterio B: Rapidez de obtención de la información. Que tan accesible es la información por jugador.

- [3] La información puede ser accedida abiertamente, no se necesita de pasar por otro archivo para llegar a los datos.
- [2] Hay que buscar línea por línea en un mismo archivo hasta llegar a la línea que contiene la información del jugador en cuestión.
- [1] Debe realizarse una búsqueda en otro árbol binario de búsqueda.

Criterio C: Persistencia de la información.

- [3] La información sigue estando disponible cuando se vuelva a ejecutar el programa
- [1] La información debe generarse de nuevo cuando se vuelve a ejecutar el programa.

	Criterio A	Criterio B	Criterio C	Total
Archivo cvs	1	2	3	6
Archivos de texto	3	3	3	9
Nodos	2	1	1	4

La manera en que se van a almacenar los datos es por medio de archivos de texto. De esta forma, se pretende que haya una carpeta data donde se encuentren los archivos de texto organizados por índices, y este índice hará parte de los nodos de cada árbol de búsqueda, donde Key va a ser el dato como tal, el rubro estadístico, y value será la posición en los archivos de texto donde se encuentra el resto de la información del jugador (índice).

Eficiencia de las consultas

Para los 4 rubros estadísticos elegidos se tienen las siguientes opciones:

Rubro/Árbol	ABB	AVL	RBTtree
Rebotes por partido	X		X
Asistencias por partido			X
Robos por partido	X	X	
Bloqueos por partido		X	

Por lo tanto, para cumplir con los requerimientos se usarán ABB y árboles rojinegros para el manejo del rubro de rebotes por partido, y así sucesivamente. Para los ítems de nombre, edad y equipo se realizará una búsqueda lineal por los archivos de texto.

Paso 6: Preparación de informes

TAD Árboles Binarios de búsqueda

TAD ABB			
ABB { root, weight }			
inv:			
Operaciones básicas			
• CreateABB			→ ABB
• getRoot	root x ABB		→ root
• getWeight	ABB		→ weight x Integer
• getMax			→ Int
• getMin			→ int
• insert	Node		→ ABB
• modify	Node x ABB		→ ABB
• search	value		→ Node
• isInTree	Key		→ Boolean
• searchEqualTo	Key		→ ArrayList<V>
• searchLowerOrEqualThan	Key		→ ArrayList<V>
• searchLowerThan	Key		→ ArrayList<V>
• searchGreaterOrEqualThan	Key		→ ArrayList<V>
• searchGreaterThan	Key		→ ArrayList<V>
• delete	Node x ABB		→ Node

Operaciones

CreateABB "Este metodo crea el arbol ABB vacio" Pre: Post: Se crea el ABB
getRoot "Este método se encarga de dar la raíz del árbol" Pre: El árbol ABB no es null Post:
GetWeight "Este método se encarga de dar el peso del árbol" Pre: Post:
getMax "Este método retorna la llave mas grande del Arbol" Pre: El árbol no es nulo Post:
getMin "Este método se encarga de retorna la llave de menor valor dentro del árbol" Pre: arbol no puede ser nulo Post:
Insert

<p>“Este método se encarga de insertar un nodo nuevo al árbol”</p> <p>Pre: Nodo nuevo no es nulo</p> <p>Post: Se agrega el nuevo nodo al arbol</p>
<p>Modify</p> <p>“Este método se encarga de modificar un nodo existente”</p> <p>Pre: El nodo modificado debe de existir</p> <p>Post: Se modifica el nodo</p>
<p>Search</p> <p>“Este método tiene la función de buscar una llave y retornar si este nodo existe junto la información que carga”</p> <p>Pre:El nodo debe estar en el árbol, el nodo de búsqueda no puede ser null</p> <p>Post:</p>
<p>isInTree</p> <p>“Este método evalua si el nodo buscado esta o no en el árbol”</p> <p>Pre: nodo distinto de null</p> <p>Post:</p>
<p>searchEqualTo</p> <p>“Este método busca un nodo con el valor de la llave asignada”</p> <p>Pre: Llave no puede ser null, Arbol distinto de null</p> <p>Post:</p>
<p>searchLowerOrEqualThan</p> <p>“Este método busca en el rabol el nodo igual o menor a la llave que se ingresa”</p> <p>Pre:Arbol no nulo y llave no nula</p> <p>Post:</p>
<p>searchLowerThan</p> <p>“Este método busca nodos con llaves menores a la que ha sido ingresada”</p> <p>Pre: Arbol no nulo y llave no nula</p> <p>Post:</p>
<p>searchGreaterOrEqualThan</p> <p>“Este método se encarga de buscar nodos mayores o iguales a la llave ingresada”</p> <p>Pre: Arbol no nulo, llave no nula</p> <p>Post:</p>
<p>searchGreaterThan</p> <p>“Este método se encarga de buscar llaves de nods que sean mayores a la ingresada”</p> <p>Pre: Arbol no nulo, Llave no nula</p> <p>Post:</p>
<p>Delete</p> <p>“Este método se encarga de eliminar el nodo que se busca atraves de una llave”</p> <p>Pre: Llave ingresada no nula, Arbol no nulo</p> <p>Post: Se elimina el nodo del árbol</p>

Diseños de casos de pruebas unitarias

Prueba 1: Verifica que el método insert añade exitosamente y cumple el invariante del árbol binario de búsqueda.				
Clase	Método	Escenario	Entrada	Resultado

BinaryTree	+insert(K key, V value):void	Se ha creado un árbol binario	K=10 V=1	El peso del árbol es 1 y el nodo raíz tiene key 10.
BinaryTree	+insert(K key, V value):void	Se ha creado un árbol binario y se la ha añadido un nodo con value 1 y key 10.	K=5 V=2	El peso del árbol es 2 y el hijo izquierdo del nodo raíz tiene key 5. El padre del nodo con value 2 es 1
BinaryTree	+insert(K key, V value):void	Se ha creado un árbol binario y le se ha añadido lo siguientes nodos: Key=10 y Value= 1 Key=5 y Value=2 En ese orden.	K=20 V=3	El peso del árbol es 3 y el hijo derecho del nodo raíz tiene key 20 y value 3. El padre del nodo 3 es 1.
BinaryTree	+insert(K key, V value):void	Se ha creado un árbol binario y le se han añadido los nodos del escenario anterior más: value=3 key=20	K=6 V=4	El peso del árbol es 4 y el hijo derecho del hijo izquierdo de la raíz tiene value 4. El padre de 4 es 2.
BinaryTree	+insert(K key, V value):void	Se ha creado un árbol binario y se le se han añadido los nodos del escenario anterior más: value=4 key=6	K=5 V=5	El peso del árbol es 5. El hijo izquierdo del hijo izquierdo de la raíz tiene value 5. El padre del hijo izquierdo del hijo izquierdo de la raíz es 2.

Prueba 2: Verifica que el método search busca exitosamente un nodo en el árbol y retorna el valor de su value dado un key.				
Clase	Método	Escenario	Entrada	Resultado
BinaryTree	+search(K): V	Se ha creado un árbol binario	K=10	El método retorna null
BinaryTree	+search(K): V	Se ha creado un árbol binario y se la ha añadido un nodo con key= 10 y value= 1	K=5	El método retorna null
BinaryTree	+search(K): V	El mismo que el anterior	K= 10	Retorna 1
BinaryTree	+search(K): V	El mismo que el anterior y se la añade un nodo con key=5 y value=2	K=6	El método retorna null
BinaryTree	+search(K): V	El mismo que el anterior	K=5	Retorna 2
BinaryTree	+search(K): V	El mismo que el anterior y se le	K=20	Retorna 3

		añade un nodo con key=20 y value 3		
BinaryTree	+search(K): V	El mismo que el anterior y se le añaden dos nodos: Key=6 Value=4 Key=15 Value=5 Key=25 Value=6	K=25	Retorna 6, es el mismo resultado que el método getMax
BinaryTree	+search(K): V	El mismo que el anterior	K=5	Retorna 2, es el mismo resultado que el método getMin

Prueba 3: Verifica que el método delete elimina exitosamente un Nodo dado el key a eliminar.				
Clase	Método	Escenario	Entrada	Resultado
BinaryTree	+delete(K): V	Se ha creado un árbol binario	K=10	Retorna null
BinaryTree	+delete(K): V	Se ha creado un árbol binario y se la ha añadido un nodo con Value=1 Key=10	K=5	Retorna null
BinaryTree	+delete(K): V	El mismo que el anterior	K = 10	Retorna 1 El árbol está vacío.
BinaryTree	+delete(K): V	Se ha creado un árbol binario y le se ha añadido los siguientes nodos: Key=10 value=1 Key=5 value=2	K=5	Retorna 2 La raíz es una hoja.
BinaryTree	+delete(K): V	El mismo que el anterior	K=10	Retorna 1 La raíz tiene key 5 y es una hoja.
BinaryTree	+delete(K): V	El mismo que el anterior más: Key=20 value=3	K=20	Retorna 3 El hijo derecho de la raíz es null.
BinaryTree	+delete(K): V	El mismo que el anterior y se le añaden dos nodos: Key=6 Value=4 Key=15 Value=5 Key=25 Value=6	K=20	Retorna 3 El hijo derecho de la raíz tiene key 15 y su hijo derecho tiene key 25.
BinaryTree	+delete(K): V	El mismo que el anterior	K=5	Retorna 2 El hijo izquierdo de la raíz es 6 y es una hoja.

Prueba 4: Verifica que el método searchEqual busca exitosamente los nodos del árbol que sean iguales a un key dado.				
Clase	Método	Escenario	Entrada	Resultado
BinaryTree	+searchEqualTo(K): V	Se ha creado un árbol binario	K=10	El método retorna un ArrayList vacío.
BinaryTree	+searchEqualTo(K): V	Se ha creado un árbol binario y se la ha añadido un nodo con key= 10 y value= 1	K=10	El método retorna arrayList de size 1 que tiene en su posición 0 el value 1
BinaryTree	+searchEqualTo(K): V	El mismo que el anterior y se le añaden los siguientes nodos: Key=10 value=2 Key=10 value=3	K= 10	Retorna un arrayList de tamaño 3 donde: Posición 0=1 Posición 1=2 Posición 2=3
BinaryTree	+searchEqualTo(K): V	El mismo que el anterior y se la añade un nodo con key=5 y value=4	K=5	El método retorna un arraylist de size 1 que tiene en su posición 0 el value 4
BinaryTree	+searchEqualTo(K): V	El mismo que el anterior	K=6	Retorna null
BinaryTree	+searchEqualTo(K): V	El mismo que el anterior y se le añade un nodo con key=20 y value=5 Key=6 Value=6 Key=15 Value=7 Key=25 Value=8 Key=15 value=9	K=20	Retorna un arrayList de size 1 que tiene en su posición 0 el value 5
BinaryTree	+searchEqualTo(K): V	El mismo que el anterior.	K=15	Retorna un arrayList de size 2 donde: Posición 0=7 Posición 1=9
BinaryTree	+searchEqualTo(K): V	El mismo que el anterior.	K=25	Retorna un arrayList de size 1 que tiene en su posición 0 el value 8

Prueba 5: Verifica que el método searchLowerOrEqualThan busca exitosamente los nodos del árbol que sean iguales o menores a un key dado.				
Clase	Método	Escenario	Entrada	Resultado

BinaryTree	+searchLowerOrEqualTo(K): V	Se ha creado un árbol binario	K=10	El método retorna un ArrayList vacío.
BinaryTree	+searchLowerOrEqualTo(K): V	Se ha creado un árbol binario y se la ha añadido un nodo con key= 10 y value= 1	K=10	El método retorna de size 1 que tiene en su posición 0 el value 1
BinaryTree	+searchLowerOrEqualTo(K): V	El mismo que el anterior y se le añaden los siguientes nodos: Key=10 value=2 Key=10 value=3	K= 10	Retorna un arrayList de tamaño 3 donde: Posición 0=1 Posición 1=2 Posición 2=3
BinaryTree	+searchLowerOrEqualTo(K): V	El mismo que el anterior y se la añade un nodo con key=5 y value=4	K=10	El método retorna un arraylist de size 4 donde: Posición 0=1 Posición 1=2 Posición 2=3 Posición 3=4
BinaryTree	+searchLowerOrEqualTo(K): V	El mismo que el anterior	K=6	Retorna un arrayList de size 1 que tiene en su posición 0 el value 4
BinaryTree	+searchLowerOrEqualTo(K): V	El mismo que el anterior	K=4	Retorna un arrayList vacío
BinaryTree	+searchLowerOrEqualTo(K): V	El mismo que el anterior y se le añade un nodo con key=20 y value=5 Key=6 Value=6 Key=15 Value=7 Key=25 Value=8 Key=15 value=9	K=20	Retorna un arrayList de size 6 donde: Posición 0=5 Posición 1=1 Posición 2=2 Posición 3=3 Posición 4=4 Posición 5=6
BinaryTree	+searchLowerOrEqualTo(K): V	El mismo que el anterior.	K=15	Retorna un arrayList de size 7 donde: Posición 0=7 Posición 1=9 Posición 2=1 Posición 3=2 Posición 4=3 Posición 5=4 Posición 6=6

BinaryTree	+searchLowerOrEqualTo(K): V	El mismo que el anterior.	K=25	Retorna un arrayList de size 9 donde las posiciones corresponden al recorrido inorden en reversa del árbol.
------------	--------------------------------	---------------------------	------	---

Prueba 5: Verifica que el método searchLowerThan busca exitosamente los nodos del árbol que sean menores a un key dado.				
Clase	Método	Escenario	Entrada	Resultado
BinaryTree	+searchLowerThan(K): V	Se ha creado un árbol binario	K=10	El método retorna un ArrayList vacío.
BinaryTree	+searchLowerThan(K): V	Se ha creado un árbol binario y se la ha añadido un nodo con key= 10 y value= 1	K=10	El método retorna un arrayList vacío.
BinaryTree	+searchLowerThan(K): V	El mismo que el anterior y se le añaden los siguientes nodos: Key=10 value=2 Key=10 value=3	K= 10	Retorna un arrayList vacío.
BinaryTree	+searchLowerThan(K): V	El mismo que el anterior y se la añade un nodo con key=5 y value=4	K=10	Retorna un arraylist de size 1 donde en su posición 0 tiene el value 4.
BinaryTree	+searchLowerThan(K): V	El mismo que el anterior	K=6	Retorna un arrayList de size 1 que tiene en su posición 0 el value 4
BinaryTree	+searchLowerThan(K): V	El mismo que el anterior	K=4	Retorna un arrayList vacío
BinaryTree	+searchLowerThan(K): V	El mismo que el anterior y se le añaden nodos key=20 y value=5 Key=6 Value=6 Key=15 Value=7 Key=25 Value=8 Key=15 value=9	K=10	Retorna un arrayList de size 2 donde: Posición 0=4 Posición 1=6
BinaryTree	+searchLowerThan(K): V	El mismo que el anterior.	K=15	Retorna un arrayList de size 5 donde: Posición 0=1 Posición 1=2 Posición 2=3 Posición 3=4

				Posición 4=6
BinaryTree	+searchLowerThan(K): V	El mismo que el anterior.	K=25	Retorna un arrayList de size 8 donde: Posición 0=5 Posición 1=7 Posición 2=9 Posición 3=1 Posición 4=2 Posición 5=3 Posición 6=4 Posición 7=6

Prueba 6: Verifica que el método searchGreaterOrEqualThan busca exitosamente los nodos del árbol que sean mayores o iguales a un key dado.				
Clase	Método	Escenario	Entrada	Resultado
BinaryTree	+searchGreaterOrEqualThan (K): V	Se ha creado un árbol binario	K=10	El método retorna un ArrayList vacío.
BinaryTree	+searchGreaterOrEqualThan (K): V	Se ha creado un árbol binario y se la ha añadido un nodo con key= 10 y value= 1	K=10	El método retorna un arrayList de size 1 donde su posición 0 es 1.
BinaryTree	+searchGreaterOrEqualThan (K): V	El mismo que el anterior y se le añaden los siguientes nodos: Key=10 value=2 Key=10 value=3	K= 10	Retorna un arrayList de size 3 donde sus posiciones son 3, 2 y 1 respectivamente.
BinaryTree	+searchGreaterOrEqualThan (K): V	El mismo que el anterior y se la añade un nodo con key=5 y value=4	K=10	Retorna un arrayList de size 3 donde sus posiciones son 3, 2 y 1 respectivamente.
BinaryTree	+searchGreaterOrEqualThan (K): V	El mismo que el anterior	K=6	Retorna un arrayList de size 3 donde sus posiciones son 3, 2 y 1 respectivamente.
BinaryTree	+searchGreaterOrEqualThan (K): V	El mismo que el anterior	K=4	Retorna un arrayList de tamaño 4 donde: Posición 0=4 Posición 1=3 Posición 2=2 Posición 3=1

BinaryTree	+searchGreaterOrEqualThan (K): V	El mismo que el anterior y se le añaden nodos: key=20 value=5 Key=6 Value=6 Key=15 Value=7 Key=25 Value=8 Key=15 value=9	K=15	Retorna un arrayList de size 4 donde: Posición 0=9 Posición 1=7 Posición 2=5 Posición 3=8
BinaryTree	+searchGreaterOrEqualThan (K): V	El mismo que el anterior.	K=20	Retorna un arrayList de size 2 donde: Posición 0=5 Posición 1=8
BinaryTree	+searchGreaterOrEqualThan (K): V	El mismo que el anterior.	K=25	Retorna un arrayList de size 1 donde su posición 0 es 8

Prueba 7: Verifica que el método searchGreaterThan busca exitosamente los nodos del árbol que sean mayores a un key dado.				
Clase	Método	Escenario	Entrada	Resultado
BinaryTree	+searchGreaterThan(K): V	Se ha creado un árbol binario	K=10	El método retorna un ArrayList vacío.
BinaryTree	+searchGreaterThan(K): V	Se ha creado un árbol binario y se la ha añadido un nodo con key= 10 y value= 1	K=10	Retorna un ArrayList vacío.
BinaryTree	+searchGreaterThan(K): V	El mismo que el anterior y se le añaden los siguientes nodos: Key=10 value=2 Key=10 value=3	K= 10	Retorna un ArrayList vacío.
BinaryTree	+searchGreaterThan(K): V	El mismo que el anterior y se la añade un nodo con key=5 y value=4	K=10	Retorna un ArrayList vacío.
BinaryTree	+searchGreaterThan(K): V	El mismo que el anterior	K=6	Retorna un arrayList de size 3 donde sus posiciones son 3, 2 y 1 respectivamente.
BinaryTree	+searchGreaterThan(K): V	El mismo que el anterior	K=4	Retorna un arrayList de tamaño 4 donde: Posición 0=4 Posición 1=3 Posición 2=2 Posición 3=1

BinaryTree	+searchGreaterThan(K): V	El mismo que el anterior y se le añaden nodos: key=20 value=5 Key=6 Value=6 Key=15 Value=7 Key=25 Value=8 Key=15 value=9	K=15	Retorna un arrayList de size 2 donde: Posición 0=5 Posición 1=8
BinaryTree	+searchGreaterThan(K): V	El mismo que el anterior.	K=20	Retorna un arrayList de size 1 donde su posición 0 es 8
BinaryTree	+searchGreaterThan(K): V	El mismo que el anterior.	K=25	Retorna un arrayList vacío.

TAD TreeNode	
TreeNode { value, parent, left, right }	
inv: left.value < value right.value > value	
Operaciones básicas <ul style="list-style-type: none"> • TreeNode value → TreeNode • isLeaf TreeNode → Boolean • getParent parent x TreeNode → parent x TreeNode • getLeft left x TreeNode → left x TreeNode • getRight left x TreeNode → right x TreeNode • setValue value x TreeNode → TreeNode • getValue TreeNode → value x TreeNode • addNode TreeNode → TreeNode • searchNode value x TreeNode → TreeNode • deleteNode value x TreeNode → TreeNode 	

Operaciones

TreeNode “Este es el metodo constructor de la clase TreeNode” Pre:value no es null Post:Se crea el nodo
isLeaf “Este método tiene la función de revisar si el nodo evaluado es una hoja del árbol” Pre:Arbol no nulo, Nodo ingresado no es nulo Post:
getParent “Este método retorna el nodo padre” Pre:El nodo evaluado se encuentra dentro del árbol, El nodo no es nulo Post:
getLeft “Este método retorna el nodo de la izquierda”

Pre:Nodo no nulo,
getRight “Este método retorna el nodo a la derecha “ Pre: El nodo no es nulo Post:
setValue “Este método modifica el valor de la variable value con otro” Pre: El nodo no es nulo Post:
getValue “Este método retorna el valor de la variable value en el nodo” Pre:Nodo no es nulo Post:
addNode “este método tiene la función de agregar un nodo a la izquierda o derecha del actual en base al valor de su llave” Pre: Post:
searchNode “Este método tiene la función de buscar el nodo que se ha ingresado” Pre: Post:
deleteNode “Este método se encarga de eliminar el nodo que se ingresó” Pre: Post:

TAD Árboles AVL

TAD AVL Tree
AVL es una extensión del arbol ABB solo que utiliza el metodo de rebalanceo cada vez que se agregue o elimine un nodo
Inv: Cada rama debe tener una distancia de h o h+/-1
Operaciones básicas <ul style="list-style-type: none"> Operacions Basicas <ul style="list-style-type: none"> AVLTree →AVL Height NodeBinaryTree→ int Left rotate NodeBinaryTree→AVL Right rotate NodeBinaryTree→AVL Rebalance NodeBinaryTree→AVL Insert NodeBinaryTree→AVL Delete NodeBinaryTree→NodeBinaryTree

Operaciones

AVL Tree “Este metodo se encargad e crear un árbol AVL” Pre: Post: Se creo un nuevo árbol AVL
leftRotate(x) “Este método se encarga de rotar cierto nodo hacia la izquierda” Pre: AVLTree!=null, AVLTree desbalanceado Post:Nodos rotados hacia la izquierda
RightRotate(x) “ Este método se encarga de rotar cierto nodo haia la derecha” Pre: AVLTree!=null, AVLTree desbalanceado Post: Nodo rotado a la derecha
Rebalance(n) “Este metodo se encarga de rebalancear el árbol siempre y cuando se agregue o se elimine un nodo del mismo” Pre: AVLTree j= null, AVLTree Weight>2, AVL Tree desbalanceado Post: El arbol es balanceado
Insert(z) “Este método se encarga de insertar un nodo dentro del árbol AVL y llama al metodo rebalancear si es necesario” Pre: Post: Nodo agregado
Delete(z) “Este método se encarga de eliminar el nodo z del árbol y llama al método rebalancear si es necesario” Pre:AVLTree!= null, z esta el árbol, Post:Se elimina el nodo z

Diseños de casos de pruebas unitarias

Prueba : Se va a probar si el método insertar, agrega los valores de la manera adecuada y rebalancea el arbol				
Clase	Método	Escenario	Entrada	Resultado
AVLTree	Insert()	Hay un árbol AVL vacio	(30,1)	GetRoot().getKey()==30
AVLTree	Insert()	Hay un árbol AVL con las llaves: 30	(15,2) (40,3)	GetRoot().getLeft().getKey()==15 GetRoot().getRight().getKey()==40
AVLTree	Insert()	Hay un árbol AVL con las llaves: 30,15,40	(45,19) (46,19)	Search(45).getLeft().getKey()==40 Search(45).getRight().getKey()==46
AVLTree	Insert()	Hay un árbol AVL con las llaves: 30,15,40,45,46	(12,1) (7,1)	Search(12).getLeft().getKey()==7 Search(12).getRight().getKey()==15

Prueba 2: Verifica que el método delete elimina un nodo del árbol dado su Key y mantiene el invariante para que el árbol este balanceado.				
Clase	Método	Escenario	Entrada	Resultado

AVLTree	Delete()	Hay un árbol AVL con las llaves: 30,15,40,45,46,12,7	30	GetRoot().getKey()==40
AVLTree	Delete()	Hay un árbol AVL con las llaves: 15,40,45,46,12,7	45	getRoot().getRight().getKey()==46
AVLTree	Delete()	Hay un árbol AVL con las llaves: 15,40,46,12,7	46	getRoot().getKey()==15 getRoot().getRight().getKey()==40

TAD Árboles rojinegros

TAD RB TREE	
RBTree{ root, NIL}	
{inv: Raíz ==black; todas las hojas == centinela (NIL); color de Nodo es rojo o negro;	
<p>Operaciones básicas</p> <ul style="list-style-type: none"> • RBTree → RBTree • getMin → K • getMax → K • insert K, V → RBTree • insertFixup RBTree → RBTree • delete RBNode → RBTree • deleteFixup RBNode → RBTree • search K → V • rightRotate RBNode → RBTree • leftRotate RBNode → RBTree • getPredessor RBNode → RBNode • getSucesor RBNode → RBNode • getNIL → RBNode • getRoot → RBNode • isInTree K → Boolean • searchBiggerThan K → ArrayList<V> • searchBiggerOrEqualThan K → ArrayList<V> • searchEqualTo K → ArrayList<V> • searchLowerTo K → ArrayList<V> • searchLowerOrEqualTo K → ArrayList<V> • setRoot RBNode → RBTree • transplant RBNode x RBNode → RBTree 	

Operaciones

RBTree “ Este es el método constructor de la clase RBTree” Pre: Post: Se crea la clase RBTree
getMin

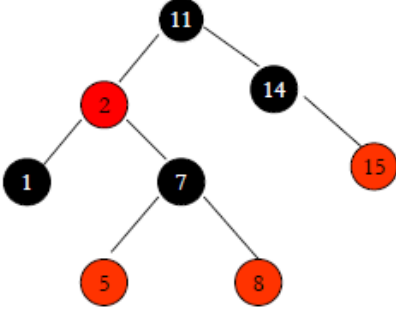
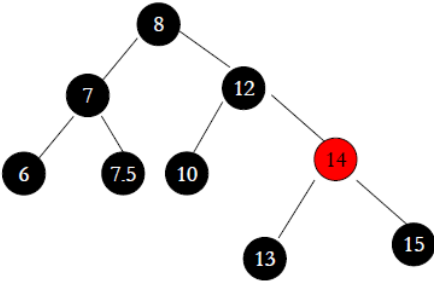
<p>“Este método se encarga de buscar la llave que corresponde al valor más bajo dentro de todo el árbol Rojo y Negro”</p> <p>Pre: El árbol Rojo y Negro no debe de ser nulo,</p> <p>Post:</p>
<p>getMax</p> <p>“Este método se encarga de encontrar la llave dentro del árbol con el valor más alto”</p> <p>Pre: El árbol Rojo y Negro no debe ser Nulo</p> <p>Post:</p>
<p>Insert</p> <p>“Este método permite insertar un nodo dentro del árbol Rojo y Negro, donde este pertenece”</p> <p>Pre: El nodo insertado de debe de ser rojo, El nodo no puede ser nulo,</p> <p>Post: Se ha agregado un nuevo nodo y el árbol va a ser rebalanceado</p>
<p>insertFixup</p> <p>“Este método se encarga de balancear el árbol Rojo y negro después de agregar un nodo”</p> <p>Pre: UN árbol Rojo y negro desbalanceado</p> <p>Post: El Arbol R y N esta blanceado</p>
<p>Delete</p> <p>“Este método permite eliminar un nodo dentro del árbol Rojo y Negro con respecto a una llave que el usuario ingresa”</p> <p>Pre: La llave debe pertenecer a los nodos del árbol, La llave no puede ser Nil</p> <p>Post: Se elimina el nodo que se buscó</p>
<p>deleteFixup</p> <p>“Este método balancea el árbol R y N después de eliminar algun nodo en el mismo”</p> <p>Pre: Arbol R y N desbalanceado</p> <p>Post: Arbol balanceado</p>
<p>Modify</p> <p>“Este método permite modificar el nodo que se buscó con una llave que el usuario ingreso y esta modifica cualquier otro dato”</p> <p>Pre: El árbol no es nulo, La llave no es nula, alguno de los valores nuevos no modifica a nulo alguno de los otros valores</p> <p>Post: Se modifica el nodo</p>
<p>Search</p> <p>“Este método tiene la función de buscar un nodo en base a una llave ingresada por el usuario”</p> <p>Pre: La llave no es null, la llave existe dentro del árbol,</p> <p>Post:</p>
<p>rightRotate</p> <p>“ Este método tiene la funicion de rotar el nodo especificado hacia la derecha para reorganizar el árbol “</p> <p>Pre: El nodo existe,</p> <p>Post: los nodos son reorganizados en base al ingresado</p>
<p>leftRotate</p> <p>“Este método se encarga de rotar el nodo hacia la izquierda en base al especificado por el usuario”</p> <p>Pre: La llave existe, el nodo no es null</p> <p>Post: El nodo es rotado hacia la izquierda</p>
<p>getPredecessor</p> <p>“Este método busca el nodo que le precede al que el usuario busca”</p>

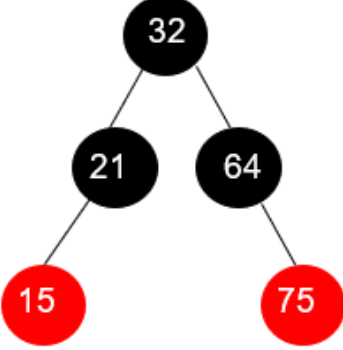
<p>Pre: Existe un numero menor a ese, Arbol no vacio</p> <p>Post:</p>
<p>getSuccesor</p> <p>“Este método se encarga de buscar el nodo que le sigue al que se esta buscando”</p> <p>Pre: El árbol no esta vacio, existe un valor mayor al buscado</p> <p>Post:</p>
<p>getNIL</p> <p>“Este método se encarga de retornar el nodo centinela del árbol Rojo y Negro”</p> <p>Pre:</p> <p>Post:</p>
<p>getRoot</p> <p>“Este método se encarga de retornar la raíz del árbol Rojo y Negro”</p> <p>Pre: El árbol Rojo y Negro no es Nulo</p> <p>Post:</p>
<p>isInTree</p> <p>“Este método verifica si la llave se encuentra o no dentro del árbol”</p> <p>Pre:</p> <p>Post:</p>
<p>searchBiggerThan</p> <p>“Este método se encarga de buscar todos los nodos mayores a la llave y retorna un arreglo con los archivos”</p> <p>Pre: Llave no es null</p> <p>Post:</p>
<p>searchBiggerOrEqualThan</p> <p>“Este método se encarga de buscar todos los nodos mayores o iguales a la llave y retorna un arreglo con los archivos”</p> <p>Pre: Llave no es null</p> <p>Post:</p>
<p>searchEqualTo</p> <p>“Este método busca el nodo que es igual a la llave y retorna un arreglo con los archivos”</p> <p>Pre: La llave no es null</p> <p>Post:</p>
<p>searchLowerOrEqualTo</p> <p>“Este método busca los nodos menores o iguales a la llave y retorna un arreglo con los archivos”</p> <p>Pre: La llave no es null</p> <p>Post:</p>
<p>searchLowerTo</p> <p>“Este método busca los nodos menores a la llave y retorna un arreglo con los archivos”</p> <p>Pre: La llave no es null</p> <p>Post:</p>
<p>Transplant</p> <p>“Este método se encarga de modificar el nodo hermano de el primer nodo “</p> <p>Pre: el primer nodo debe existir</p> <p>Post: Se ha modificado la ubicación del segundo nodo</p>

Diseños de casos de pruebas unitarias

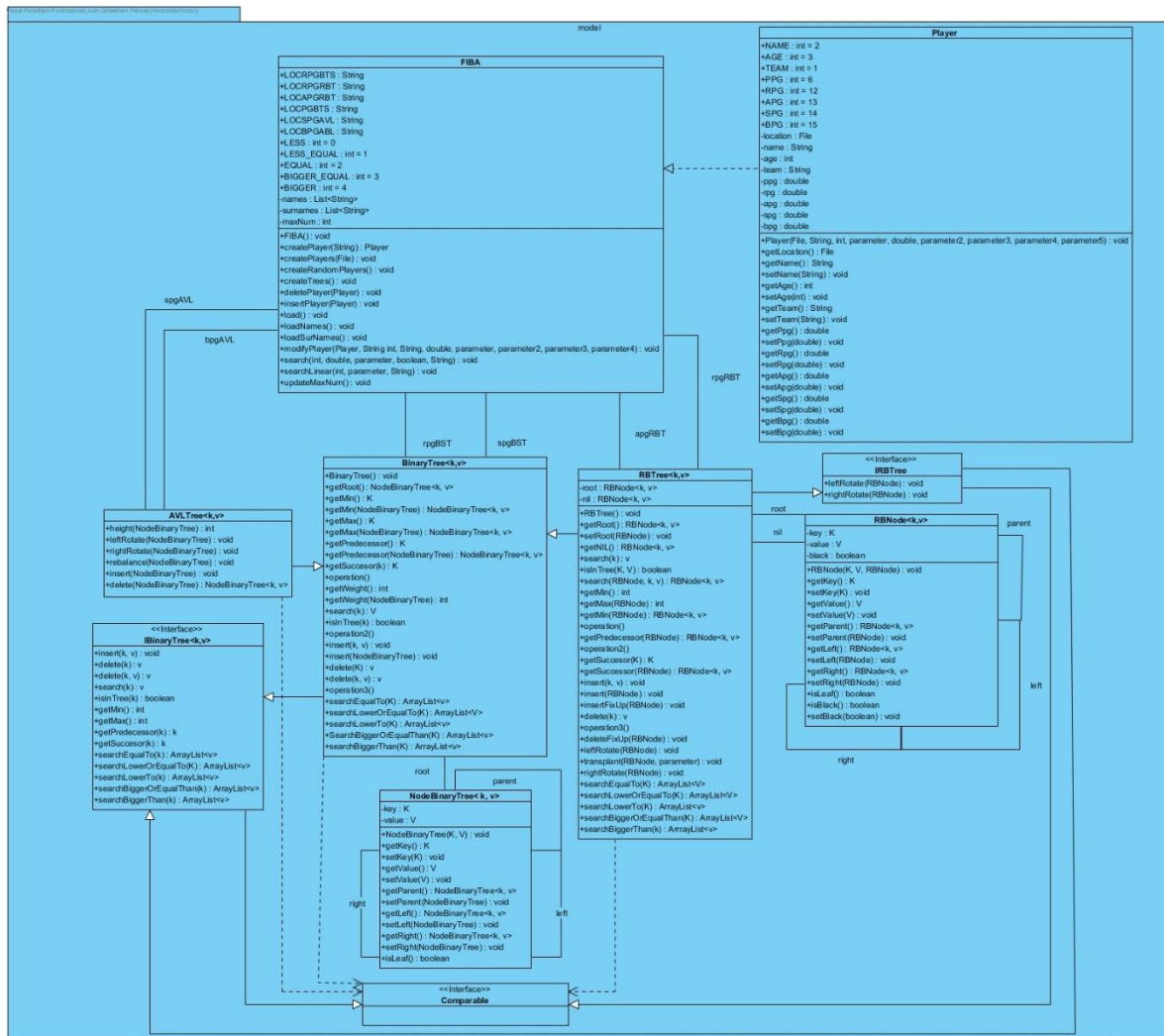
Prueba 1: Verifica que el método añade inserta correctamente un nodo al árbol manteniendo las propiedades.				
Clase	Método	Escenario	Entrada	Resultado
RBTree	+insert (K, V): void	Hay un árbol rojinegro vacío	K=10 V=1	La raíz es negra
RBTree	+insert (K, V): void	Hay un árbol rojinegro con un nodo: Key=10, value=1	K=5 V=2	La raíz es negra El hijo izquierdo de la raíz es rojo.
RBTree	+insert (K, V): void	Hay un árbol rojinegro con los siguientes nodos: Key=10 value=1 Key=5 value=2	K=15 V=3	La raíz es negra y sus dos hijos son rojos.
RBTree	+insert (K, V): void	El mismo que el anterior más un nodo con key=15 y value=3	K=4 V=4	La raíz es negra y sus dos hijos son negros. El hijo izquierdo el hijo izquierdo de la raíz es rojo.
RBTree	+insert (K, V): void	El mismo que el anterior más un nodo con key=4 y value=4	K=4 V=5	La raíz es negra. Sus dos hijos son negros y tiene key 4 y 15 respectivamente. Los dos hijos del hijo izquierdo de la raíz son rojos y tienen key 4 y 5 respectivamente.

Prueba 2: Verifica que el método delete elimina correctamente un nodo al árbol manteniendo las propiedades.				
Clase	Método	Escenario	Entrada	Resultado
RBTree	+delete(K): V	Hay un árbol rojinegro vacío	K=10	Retorna null. El árbol sigue estando vacío
RBTree	+delete(K): V	Hay un árbol rojinegro con un nodo: Key=10, value=1	K=10	Retorna 1 El árbol está vacío.
RBTree	+delete(K): V	Hay un el siguiente árbol rojinegro:	K=5	Retorna 5 No ocurre ninguna rotación, los colores permanecen estables. El hijo izquierdo de 7 es NIL.

		 <p>Con los siguientes Values: $V(11)=1$ $V(2)=2$ $V(1)=3$ $V(7)=4$ $V(5)=5$ $V(8)=6$ $V(14)=7$ $V(15)=8$</p>		
RBTree	+delete(K): V	El mismo que el anterior	K=7	Retorna 4 8 pasa a ser el hijo derecho de 2 y es negro, además su hijo derecho es 5 y sigue siendo rojo.
RBTree	+delete(K): V	El mismo del anterior	K=20	Retorna null.
RBTree	+delete(K): V	<p>Se tiene el siguiente árbol rojinegro</p>  <p>Con los siguiente values: $V(8)=1$ $V(7)=2$ y su key ahora va a ser 6 $V(12)=3$ $V(6)=4$ y su key ahora va a ser 5 $V(7.5)=5$ y su key ahora va a ser 7 $V(10)=6$ $V(14)=7$ $V(15)=8$</p>	K=10	Retorna 6. La raíz es negra y es 8 Su hijo derecho es 14 y es negro. El hijo izquierdo de 14 es 12 y es negro y su hijo derecho es 13 y es rojo. El hijo derecho de 14 es 15 y es negro.

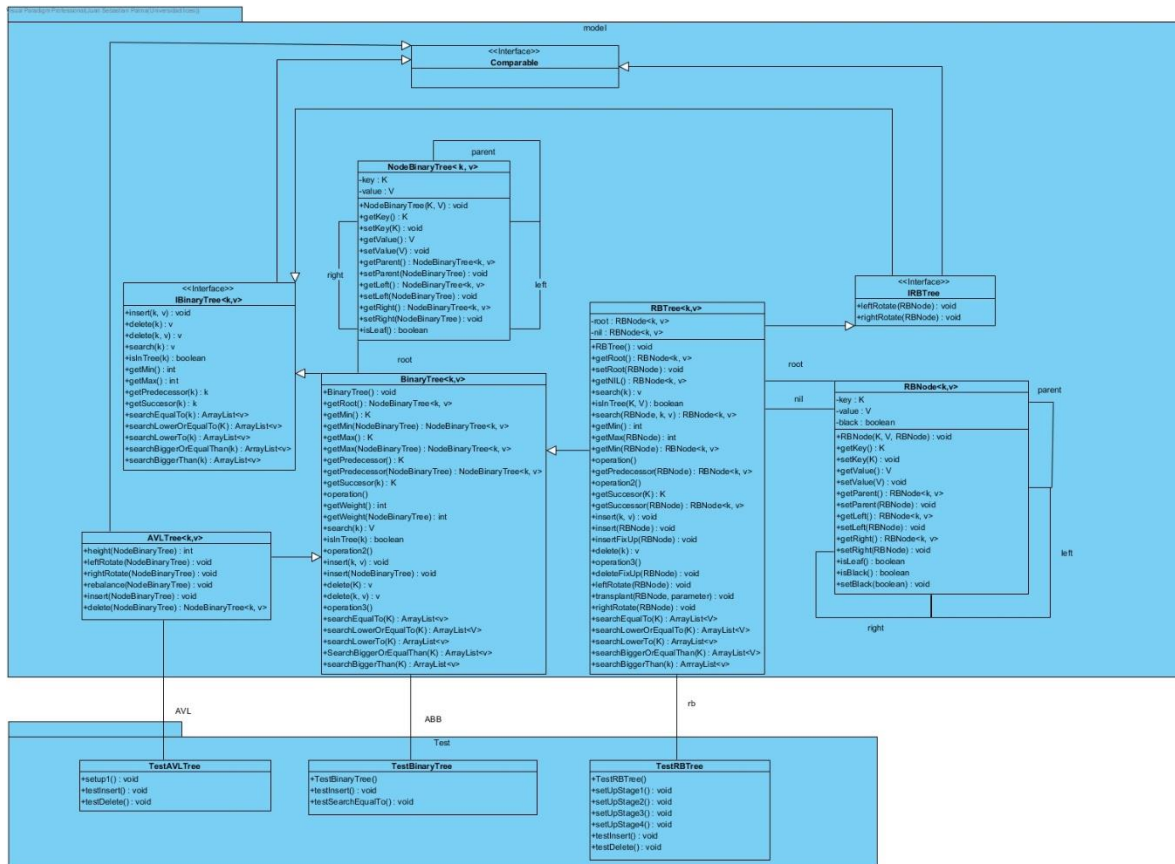
		V(13)=9		
RBTree	+delete(K): V	<p>Existe el siguiente árbol rojinegro:</p>  <pre> graph TD 32((32)) --- 21((21)) 32 --- 64((64)) 21 --- 15((15)) 64 --- 75((75)) style 32 fill:#000,color:#fff style 21 fill:#000,color:#fff style 64 fill:#000,color:#fff style 15 fill:#f00,color:#fff style 75 fill:#f00,color:#fff </pre> <p>Con los siguiente Values: V(32)=1 V(21)=2 V(64)=3 V(15)=4 V(75)=5</p>	K=21	<p>Retorna 2</p> <p>La raíz es 32 y sus dos hijos son negros y son 15 y 64 respectivamente. El hijo derecho de 64 es 75 y es rojo.</p>
RBTree	+delete(K): V	El mismo que el anterior	K=64	<p>Retorna 3</p> <p>La raíz es 32 y sus dos hijos son negros y son 21 y 75 respectivamente. El hijo izquierdo de 21 es 15 y es rojo.</p>

Diseño del diagrama de clases de la solución



El archivo se encuentra en la carpeta Bibliografía/Diagramas/UML.jpg

Diseño del diagrama de pruebas unitarias



El archivo se encuentra en la carpeta Bibliografía/Diagramas/Test.vpp

Bibliografía

Anónimo. (Desconocido) “2017-18 Golden State Warriors Roster and Stats”. Tomado de: <https://www.basketball-reference.com/teams/GSW/2018.html>

Anónimo. (Desconocido). “Definición de árbol Rojinegro”. Tomado de: <https://www.infor.uva.es/~cvaca/asigs/doceda/rojinegro.pdf>

Carrillo, A. (2017, marzo 6). “Estadísticas del baloncesto: los 4 factores”. Tomado de: <http://sportsmadeinusa.com/baloncesto/nba/estadisticas-baloncesto-4-factores/>

ESPN. (Desconocido). “NBA Player Scoring Statistics - 2018-19”. Tomado de: http://www.espn.com/nba/statistics/player/_/stat/scoring/sort/points

ESPN. (Desconocido). “Basquetbol Estadísticas -2019”. Tomado de: <http://www.espn.com.co/basquetbol/nba/estadisticas>

Guarín, S. (2004). “Árboles AVL”. Tomado de: <http://es.tldp.org/Tutoriales/doc-programacion-arboles-avl/avl-trees.pdf>

Matemáticas Discretas (2017, noviembre 23). “Capítulo 12: Teoría sobre árboles binarios”. Tomado de: <https://medium.com/@matematicasdiscretaslibro/capitulo-12-teoria-de-arboles-binarios-f731baf470c0>