Algoritmos y Estructuras de Datos
Unidad 3 Laboratorio

Nicolás Martinez, Cristian Molina y Juan Manuel Castillo.

url github: https://github.com/nicolasmr21/NBA_Manager

Paso 1. Identificación del problema

Identificación de necesidades y síntomas

- Debido a la gran cantidad de datos generados por el Baloncesto, la FIBA necesita el desarrollo de una aplicación que permita la consulta de datos de cada uno de los profesionales del baloncesto en el planeta para posterior análisis de estos.
- Se necesita que la aplicación realice las búsquedas necesarias en el menor tiempo posible teniendo en cuenta la gran cantidad de datos.
- Se necesita que la información pueda separarse en criterios de búsqueda con base a 4 de los 5 rubros estadísticos disponibles.

Definición del problema

Una empresa desarrolladora de software requiere de una serie de funcionalidades las cuales son expuestas por la FIBA ante la masiva información proveniente del Baloncesto, permitiendo el análisis de grandes cantidades de información, buscando hacia donde se dirige este deporte en la actualidad.

Requerimientos funcionales

Nombre	**R.F. #1 Ingresar archivos de información con con una cantidad superior o igual a 200000 datos.**
Resumen	**Cargar información de gran tamaño que permita ingresar datos, ya sea de manera masiva (archivos csv, por ejemplo)**
Entrada	**Archivo**
Salida	**Se ha ingresado los archivos de información con una con con una cantidad superior o igual a 200000 datos**

Salida	**Se han clasificado los jugadores por plataforma al buscar una partida.**

Nombre	**R.F. #2 Modificar archivos de información con una cantidad superior o igual a 200000 datos**
Resumen	**Modificar la información cargada previamente mediante un archivo .csv o txt**
Entrada	**Archivo**
Salida	**Se ha modificado los archivos de información con una cantidad superior o igual a 200000 datos.**

Nombre	**R.F. #3 Eliminar archivos de información con una cantidad superior o igual a 200000 datos**
Resumen	**Eliminar la información cargada previamente mediante un archivo .csv o txt**
Entrada	**Archivo**
Salida	**Se ha eliminado los archivos de información con una cantidad superior o igual a 200000 datos.**

Nombre	**R.F. #4 Implementar un buscador de jugadores por criterios de búsqueda con base a 5 rubros.**
Resumen	**Realizar consultas de jugadores utilizando como criterios de búsqueda las categorías estadísticas incluídas en base a los 5 rubros disponibles**
Entrada	**Archivo**
Salida	**Se ha implementado un buscador de jugadores por criterios de búsqueda con base a 5 rubros.**

Nombre	R.F. #4 Implementar índices para 4 de los 5 rubros en el buscador.
Resumen	Realizar consultas de jugadores utilizando índices como criterios de búsqueda para una eficacia mayor en cuanto a tiempo de búsqueda.
Entrada	Archivo
Salida	Se ha implementado índices para 4 de los 5 rubros en el buscador.

Requerimientos no funcionales

Nombre	R.N.F# 1 Implementar el sistema de búsqueda de jugadores en el menor tiempo posible.
Resumen	Con base al sistema implementado para la búsqueda de jugadores, se debe tener en cuenta el menor tiempo de exploración posible.

Nombre	R.N.F# 2 Almacenar la información en memoria secundaria
Resumen	Guardar en memoria secundaria el archivo de información ya que por su tamaño es imposible tenerlo en memoria principal

Nombre	R.N.F# 3 Almacenar la información en memoria secundaria
Resumen	Guardar en memoria secundaria el archivo de información ya que por su tamaño es imposible tenerlo en memoria principal

Nombre	R.N.F# 4 Limitar a 4 de los criterios de búsqueda como atributos estadísticos
Resumen	**Ejecutar consultas de jugadores de acuerdo a todos los criterios, pero solamente cuatro de ellas (sobre atributos estadísticos) deben resultar eficientes.**

Nombre	R.N.F# 5 Utilizar una misma estructura de datos recursiva para 2 de las 4 búsquedas por atributos característicos
Resumen	**Con base a los árboles AVL y Rojinegro, una misma estructura de datos recursiva se utiliza para dos de los cuatro criterios de búsqueda.**

Fase 2. Recopilación de la información necesaria

Definiciones

Fuente:

https://www.wikipedia.org/

https://cupintranet.virtual.uniandes.edu.co/

https://users.dcc.uchile.cl/

https://definicion.de/

FIBA

Es el organismo que se dedica a regular las normas del baloncesto mundialmente, así como de celebrar periódicamente competiciones y eventos en sus dos disciplinas. Fue fundada en 1932 y tiene su sede actual en Mies (Suiza). Cuenta en 2018 con la afiliación de 213 federaciones nacionales, divididas a su vez en 5 federaciones continentales, que son: África, América, Asia, Europa y Oceanía. El argentino Horacio Muratore es el presidente de la FIBA, desde el 2014. (Fuente: Wikipedia)

Baloncesto

Es un deporte de equipo, jugado entre dos conjuntos de cinco jugadores cada uno durante cuatro períodos o cuartos de diez o doce minutos cada uno. El objetivo del equipo es anotar puntos introduciendo un balón por la canasta, un aro a 3,05 metros sobre la superficie de la pista de juego del que cuelga una red. La puntuación por cada canasta o cesta es de dos o tres puntos, dependiendo de la posición desde la que se efectúa el tiro a canasta, o de uno, si se trata de un tiro libre por una falta de un jugador contrario. El equipo ganador es el que obtiene el mayor número de puntos. (Fuente: Wikipedia)

Árbol Rojo-negro

Un árbol rojo negro es un árbol binario donde cada nodo tiene también un atributo de color, cuyo valor puede ser o rojo o negro. Las hojas de un árbol rojo negro son irrelevantes y no tienen datos. (Fuente: UniAndes)

Condiciones Árbol rojo-negro

- La raíz del árbol es negra.
- Los hijos de un nodo rojo son negros.
- Las hojas del árbol son negras.
- Todas las ramas del árbol (caminos desde la raíz hasta una hoja) tienen el mismo número de nodos negros. (Fuente: UniAndes)

Árbol Binario de Búsqueda (ABB)

Es un árbol binario que almacena en cada nodo una llave. El árbol de búsqueda binaria cumple las siguientes propiedades:

- El árbol vacío es un ABB.
- Los árboles izquierdo y derecho son ABBs
- La llave k es mayor que todas las llaves almacenadas en el ABB izquierdo
- La llave k es menor que todas las llaves almacenadas en el ABB derecho.(Fuente: UChile)

Eficiencia Algorítmica

Es usado para describir aquellas propiedades de los algoritmos que están relacionadas con la cantidad de recursos utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza. La eficiencia algorítmica puede ser vista como análogo a la ingeniería de productividad de un proceso repetitivo o continuo. (Fuente: Wikipedia)

Memoria Secundaria

También conocida como almacenamiento secundario, es el conjunto de dispositivos y soportes de almacenamiento de datos que conforman el subsistema de memoria de la computadora, junto con la memoria primaria o principal. La memoria secundaria es un tipo de almacenamiento masivo y permanente (no volátil) con mayor capacidad para almacenar datos e información que la memoria primaria que es volátil, aunque la memoria secundaria es de menor velocidad. (Fuente: Wikipedia.)

Memoria Principal

Es la memoria de la computadora donde se almacenan temporalmente tanto los datos como los programas que la unidad central de procesamiento (CPU) está procesando o va a procesar en un determinado momento. Esta clase de memoria es volátil, es decir que

cuando se corta la energía eléctrica, se borra toda la información que estuviera almacenada en ella. (Fuente: Wikipedia.)

Memoria Caché

Es un búfer especial de memoria que poseen las computadoras, que funciona de manera semejante a la memoria principal, pero es de menor tamaño y de acceso más rápido. Nace cuando las memorias ya no eran capaces de acompañar a la velocidad del procesador, por lo que se puede decir que es una memoria auxiliar, que posee una gran velocidad y eficiencia y es usada por el microprocesador para reducir el tiempo de acceso a datos ubicados en la memoria principal que se utilizan con más frecuencia. (Fuente: Wikipedia.)

Rubro

Es un título, un rótulo o una categoría que permite reunir en un mismo conjunto a entidades que comparten ciertas características. (Fuente: Definición.)

Fase 3. Búsqueda de soluciones creativas

El grupo ha acordado entender el entorno donde se almacena una gran cantidad de información depositada en formato csv , que contiene datos estadísticos de plantillas de jugadores pertenecientes a la FIBA, es decir, de Baloncesto profesional (En nuestro caso tomamos: partidos jugados, puntos por juego, rebotes por juego, robos por juego, bloqueos por juego) Además del nombre, el equipo, el ID y la edad. Las formas de almacenar todos los datos de los jugadores pueden ser:

- Utilizando una lista genérica
- Utilizando un árbol AVL
- Utilizando un árbol Red Black
- Utilizando un árbol ABB
- Utilizando una cola de prioridad.

ALTERNATIVAS PARA BÚSQUEDA

Búsqueda Secuencial: Consiste en introducir un elemento a buscar, el cual será buscado en cada uno de los elementos pertenecientes al vector. Si por ejemplo el contenedor de los datos es un Array de tamaño 10, se comparará con cada uno de los que contiene el arreglo. Es importante reconocer que en caso de que el elemento no se encuentre, las comparaciones se harán igualmente hasta el último elemento del arreglo, lo que se podría interpretar como en vano.

Búsqueda Binaria: Una vez el arreglo está ordenado se puede realizar este tipo de búsqueda. Internamente se selecciona el elemento ubicado en la mitad del arreglo, si el elemento a buscar es el mismo de la mitad la busqueda termina. De no ser así, el arreglo se divide en dos partes. Se evalúa si el elemento se encuentra en la parte izquierda del

arreglo, es decir, menor al valor de la mitad, o a la derecha en caso de que sea mayor a este.El proceso se repite hasta encontrar el elemento buscado en caso de que este pertenezca al arreglo.

ALTERNATIVAS PARA EL ALMACENAMIENTO DE LOS DATOS

Dado el tamaño de los archivos a manejar, podemos plantear tres tipos de estructuras de datos en la búsqueda de una balanza entre eficiencia y eficacia para así, obtener un rendimiento óptimo en cuanto al funcionamiento en general del programa, en específico en las búsquedas por rubros. Es por esto que desplegamos las posibilidades exponiendo tanto las ventajas como desventajas de cada estructura:

1. ***Árbol ABB***

   Es una opción a considerar debido a que es una estructura de datos que busca mejoras en tiempo en sus métodos (inserción, búsqueda, eliminación, etc) respecto a otras estructuras las cuales lo realizan de manera lineal como arreglos y listas. Sin embargo, esta estructura depende de la altura del árbol para la facilidad de acceso en el cual en el peor de los casos puede llegar a compararse con el de un arreglo , es decir, complejidad O(n).

2. ***Arbol AVL***

   Se podrían tener en cuenta los arboles AVL, al encontrarse siempre balanceados de modo que sus hijos por el lado izquierdo no se diferencien en más de una unidad para el equilibrio del árbol obteniendo una complejidad en el peor caso de $O(\log_2 n)$. Sin embargo a la hora de la implementación puede llegar a ser un poco más complejo debido a las rotaciones necesarias dependiendo del caso que se presente ya sea para los métodos de inserción y borrado los cuales implican, por lo general, un re balanceo del árbol.

3. ***Árbol Red Black (RB)***

   Es la opción más oportuna para el desarrollo del laboratorio al tener tiempos de insercion, eliminacion y búsquedas con una complejidad O(log2n) por siguiente, es esta estructura la recomendada para programas necesitados de eficacia en cuanto a búsquedas se refiere. El Red Black se convierte en una versión mejorada del AVL debido a que el anterior mencionado es más austero y férreo al realizar su balanceo por lo que sus tiempos de búsqueda, inserción y borrado son mayores.

4. ***ArrayList***

   Como última opción, se debe tener en cuenta a una estructura de datos recurrente en los tres cursos de programación. La tomamos en cuenta debido a que es mejor en cuanto a tiempos de búsqueda que, por ejemplo, una LinkedList la cual toma un tiempo proporcional al tamaño de su misma estructura. ArrayList nos ofrece mayor estabilidad en cuanto a tiempos respecto a sus métodos; estando la mayoría en una complejidad promedio de O(n/2) por lo que por tiempo nos beneficia y en cuanto a memoria utilizada es menor que en LinkedList.

Fase 4. Transición de la formulación de ideas a los diseños preliminares

Para la transformación de búsqueda de soluciones creativas a diseños preliminares planteamos una tabla con criterios los cuales nos permiten entender mejor el funcionamiento de cada posible estructura a usar obteniendo un total en base a los criterios cumplidos, seleccionando los que mayor puntaje haya obtenido

- **Criterio 1.** Los métodos (Inserción, eliminación, busqueda) no deben superar la barrera de complejidad de O(n)
- **Criterio 2.** Su implementación es sencilla y entendible en cuanto a revisiones y mejoras futuras.
- **Criterio 3.** Su implementación puede estar basada de forma recursiva.
- **Criterio 4.** La estructura está en la capacidad de auto balancearse para la obtención de menores tiempos.
- **Criterio 5.** La estructura debe contar con todas las operaciones necesarias para el desarrollo del laboratorio (Agregar un elemento, eliminarlo, realizar la búsqueda.)

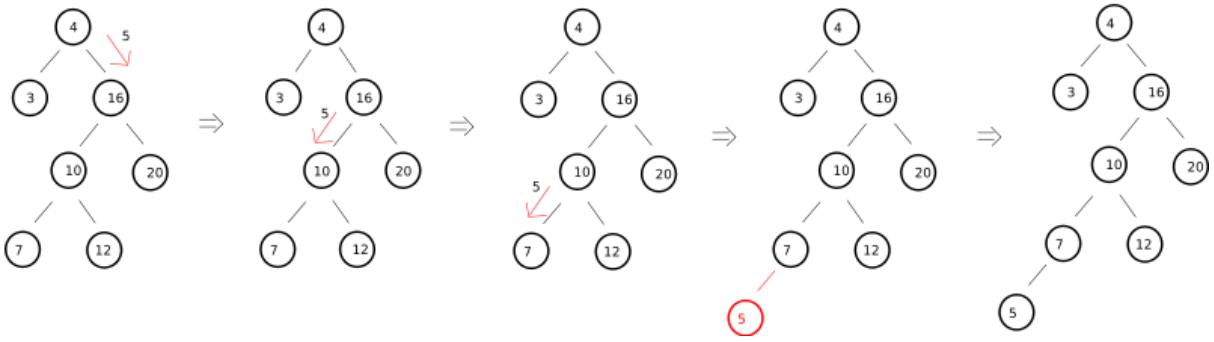
Estructura	Criterio 1	Criterio 2	Criterio 3	Criterio 4	Criterio 5	TOTAL
Árbol ABB	x	x	x		x	4
Árbol AVL	x		x	x	x	4
Árbol RB	x		x	x	x	4
ArrayList		x			x	2

Fase 5. Evaluación y selección de la mejor solución

- Dado el empate en las tres estructuras (árbol ABB, árbol AVL, árbol RB) de datos decidimos incluir las tres en búsqueda de conocer a fondo la implementación de estas; siendo estas el pilar y base fundamental del desarrollo del programa, estos se encargaran de recibir la información almacenada en memoria secundaria por parte de un archivo csv con más de 200.000 datos de jugadores de la FIBA, ente deportivo del Baloncesto.
- Adicional a las estructuras evaluadas anteriormente, se hará uso de una estructura extra (propia) “auxiliar” catalogada List, que nos permitirá mostrar los índices más cercanos al valor buscado en base al rubro indicado.

Mejor solución para inserción, eliminación y búsqueda de la información

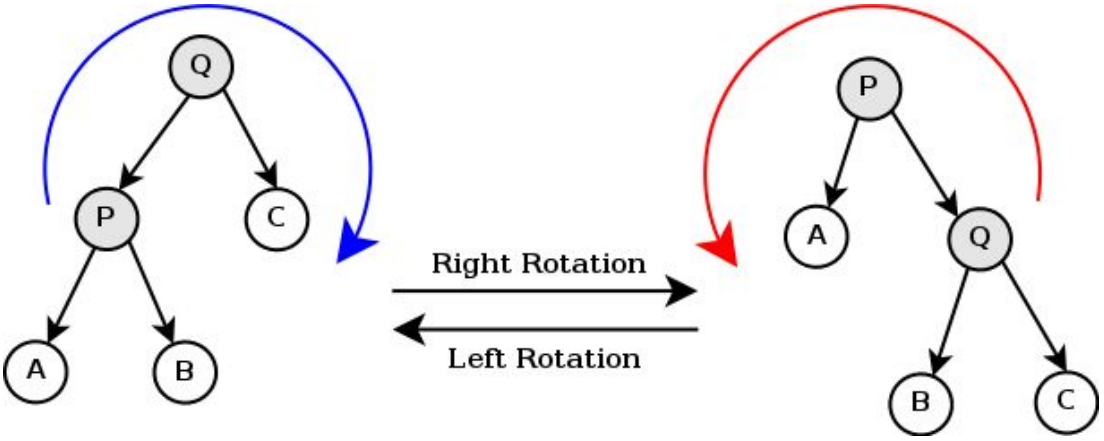
Funcionamiento del Árbol ABB



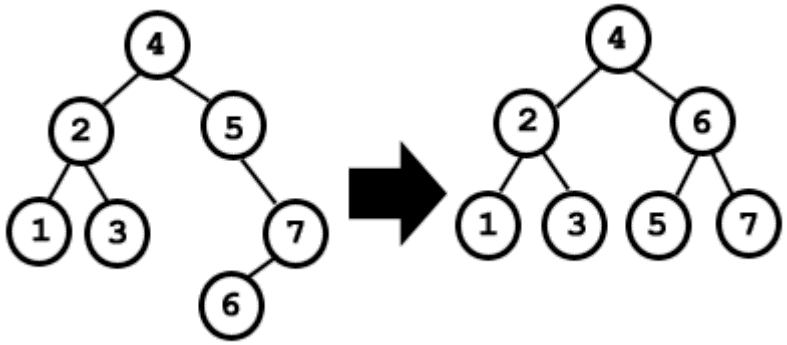
Se basa en la propiedad de que las claves que son menores que el padre se encuentran en el subárbol izquierdo, y las claves que son mayores que el padre se encuentran en el subárbol derecho.

Funcionamiento del Árbol AVL

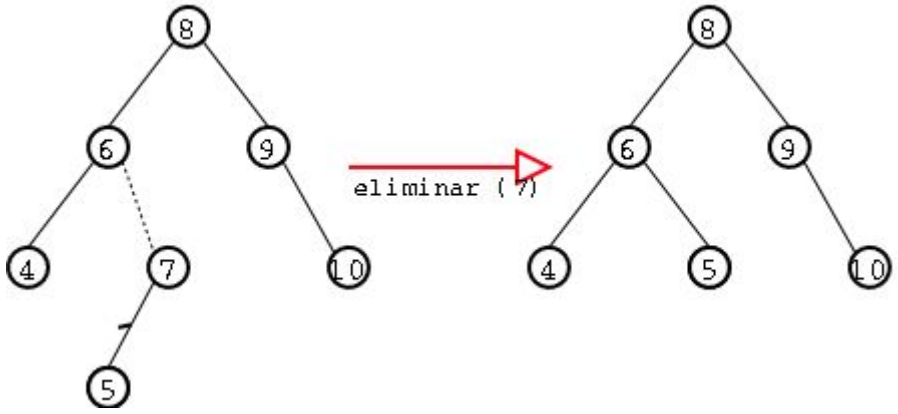
Doble rotación:



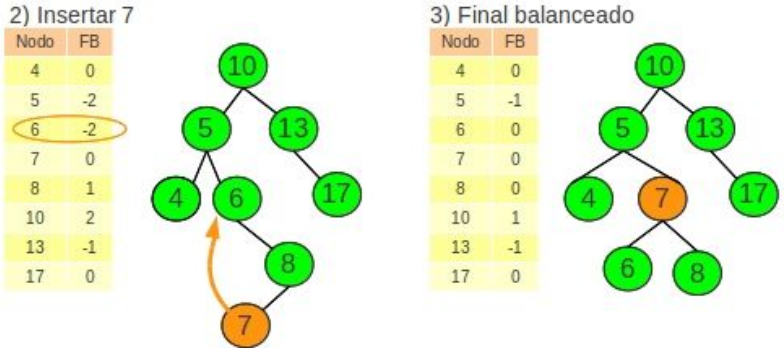
Balanceo:



Eliminación:

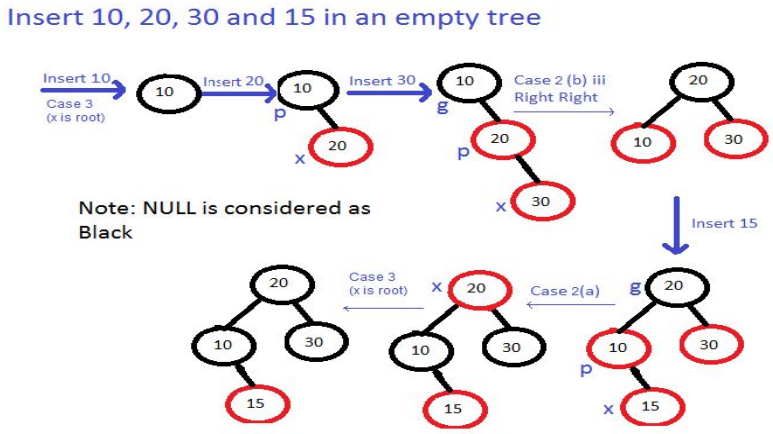


Inserción:



Dado que todas las claves nuevas se insertan en el árbol como nodos hoja y que sabemos que el factor de equilibrio para una hoja nueva es cero, no hay nuevos requisitos para el nodo que se acaba de insertar. Pero una vez que se agrega la hoja nueva, debemos actualizar el factor de equilibrio de su padre. La forma en que esta hoja nueva afecta al factor de equilibrio del padre depende de si el nodo hoja es un hijo izquierdo o un hijo derecho. Si el nuevo nodo es un hijo derecho, el factor de equilibrio del padre se reducirá en uno. Si el nuevo nodo es un hijo izquierdo, entonces el factor de equilibrio del padre se incrementará en uno. Esta relación puede aplicarse recursivamente al abuelo del nuevo nodo, y posiblemente a cada antepasado hasta la raíz del árbol. (Fuente: http://interactivepython.org/)

Funcionamiento de Árbol RB (Red Black)



Basándose en los invariantes:

- Árbol binario estricto (los nodos nulos se tienen en cuenta en la definición de las operaciones
- todo nodo hoja es nulo)
- Cada nodo tiene estado rojo o negro
- Nodos hoja (nulos) son negros
- La raíz es negra (esta condición se impone para simplificar algunas operaciones)

Inserción

La inserción de un nodo se realiza inicialmente igual que en un árbol binario de búsqueda. Al nuevo nodo se le da el color rojo. De ésta manera no se viola la segunda condición (altura negra), aunque se puede violar la primera (el padre del nodo insertado puede ser rojo).

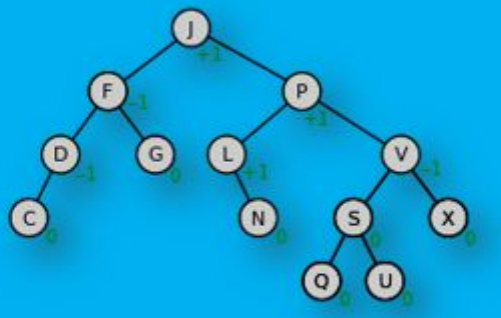
Eliminación

Si es un nodo con dos hijos no nulos, se busca el mayor nodo (el más a la derecha) de su subárbol izquierdo (lo llamamos x), se intercambia sus datos con el nodo a borrar y se pasa a borrar el nodo x. Como es el nodo más a la derecha, su hijo derecho será nulo.

Fase 6. Preparación de informes y especificaciones . Diseño de diagrama de clases.

Diseño de TADs

TAD AVLTree



{inv: ∀ a_1 | $sub_{a1} - sub_{b1} \leq 1$ }

- Constructor: → AVLTree(Constructor)
- getRoot: AVLTree → Elem(Analizador)
- search: AVLTree x key → Elem(Analizador)
- searchAVL: AVLTree x key x node → Elem(Analizador)
- getFe: AVLTree x node → Elem(Modificador)
- leftRotation: AVLTree x node → Elem (Modificador)
- rightRotation: AVLTree x node → Elem (Modificador)
- doubleLeftRotation: AVLTree x node → Elem (Modificador)
- doubleRightRotation: AVLTree x node → Elem (Modificador)
- insertAVL: AVLTree x node → Elem (Modificador)
- insert: AVLTree x node → AVLTree (Modificador)
- getLessThan: AVLTree x key → List(Analizador)
- getBiggerThan: AVLTree x key → List(Analizador)
- searchCloserAVL: AVLTree x key x node → Node(Analizador)
- searchCloser: AVLTree x key → Node(Analizador)

getRoot()

"Este método se encarga de dar la raíz el árbol"
Pre: Debe existir el árbol para la implementación del método
Post: Se devuelve un elemento del árbol catalogado como raíz.

search(*K key*)

"Este método se encarga de llamar al método searchAVL para la búsqueda de manera pública por índice"

Pre: Debe existir el árbol para la implementación del método
Post: Se devuelve un nodo en forma de lista dada la búsqueda por índice.

searchAVL(*K key*)

"Este método se encarga de realizar la búsqueda de manera privada por índice"
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se devuelve un nodo en forma de lista dada la búsqueda por índice.

getFe(*K*)

"Este método se encarga de indicar valor de balanceo del nodo"
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se devuelve un valor int con el valor corregido del balanceo del nodo.

leftRotation(*Node<K, V> node*)

"Este método se encarga de realizar una rotación hacia la izquierda con base al caso presentado"
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se ha realizado una rotación a la izquierda en bien del balanceo del árbol.

rightRotation(*Node<K, V> node*)

"Este método se encarga de realizar una rotación hacia la derecha con base al caso presentado"
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se ha realizado una rotación a la derecha en bien del balanceo del árbol.

doubleLeftRotation(*Node<K, V> node*)

“Este método se encarga de realizar una rotación simple hacia la derecha y otra a rotación simple hacia la izquierda con base al caso presentado”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se ha realizado las dos rotaciones en bien del balanceo del árbol.

doubleRightRotation(*Node<K, V> node*)

“Este método se encarga de realizar una rotación simple hacia la izquierda y otra a rotación simple hacia la derecha con base al caso presentado”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se ha realizado las dos rotaciones en bien del balanceo del árbol.

insertAVL(*Node<K, V> novo, Node<K, V> sub)*

“Este método se encarga de realizar la inserción de un nodo buscando la posición correcta para ser agregado”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se ha realizado la inserción de un nodo en el árbol AVL.

insert(*Node<K, V> novo)*

“Este método se encarga de llamar el método insertAVL haciendo de este una inserción de manera pública de un nodo buscando la posición correcta para ser agregado”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se ha realizado la inserción de un nodo en el árbol AVL.

getBiggerThan*(K k)*

“Este método se encarga de dar los valores de los índices más altos dependiendo de la búsqueda de determinado rubro”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se devuelve una lista con los índices más altos con base a la búsqueda realizada.

getBiggerThan*(K k)*

“Este método se encarga de dar los valores de los índices más bajos dependiendo de la búsqueda de determinado rubro”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se devuelve una lista con los índices más bajos con base a la búsqueda realizada.

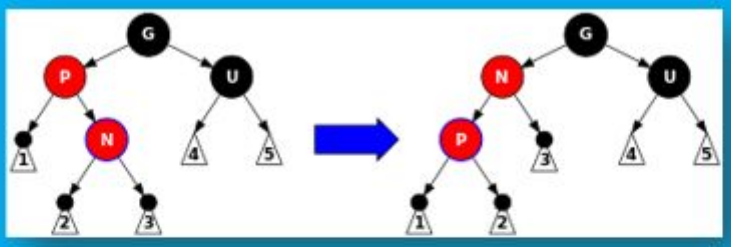
searchCloserAVL*(K key, Node<K, V> node)*

“Este método se encarga de dar los nodos más cercanos al índice buscado”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se devuelve una lista de nodos con los resultados más cercanos a la búsqueda realizada.

searchCloser(*K key*)

“Este método se encarga de llamar el método searchCloserAVL para la realización de la búsqueda más cercana con base al índice dado.”
Pre: Debe existir el árbol AVL para la implementación del método
Post: Se devuelve una lista de nodos con los resultados más cercanos a la búsqueda realizada.

TAD RBTree



{inv1: ∀ *a* | key$_{a1}$ > key$_{a2}$ > key$_{a3}$ > ... > key$_{an}$}
{inv2: ∀ *a* | root$_a$ = black }
{inv3: ∀ *a* | (sub$_{a1}$ = (black ⋎ red)) ≠ (sub$_{b1}$ = (black ⋎ red))}

- Constructor: — RBTree(Constructor)
- getRoot: RBTree — Elem(Analizador)
- search: RBTree x key — Elem(Analizador)
- searchRB: RBTree x key x node — Elem(Analizador)
- leftRotation: RBTree x node — Elem (Modificador)
- rightRotation: RBTree x node — Elem (Modificador)
- insertRB: RBTree x node — RBTree (Modificador)
- correct: RBTree x node — RBTree (Modificador)
- getLessThan: RBTree x key — List(Analizador)
- getBiggerThan: RBTree x key — List(Analizador)
- searchCloserRB: RBTree x key x node — Node(Analizador)
- searchCloser: RBTree x key — Node(Analizador)

getRoot()

"Este método se encarga de dar la raíz el árbol"
Pre: Debe existir el árbol RB para la implementación del método
Post: Se devuelve un elemento del árbol catalogado como raíz.

search(*K key*)

"Este método se encarga de llamar al método searchRB para la búsqueda de manera pública por índice"
Pre: Debe existir el árbol para la implementación del método
Post: Se devuelve un nodo en forma de lista dada la búsqueda por índice.

searchRB(*K key, RBNode<K, V> node*)

"Este método se encarga de realizar la búsqueda de manera privada por índice"
Pre: Debe existir el árbol RB para la implementación del método
Post: Se devuelve un nodo en forma de lista dada la búsqueda por índice.

correct(*RBNode<K, V> z*)

"Este método se encarga de balancear el árbol con base a los casos que se vayan presentando."
Pre: Debe existir el árbol RB para la implementación del método
Post: Se devuelve el árbol de manera balanceada.

leftRotation(*Node<K, V> node*)

"Este método se encarga de realizar una rotación hacia la izquierda con base al caso presentado"
Pre: Debe existir el árbol RB para la implementación del método
Post: Se ha realizado una rotación a la izquierda en bien del balanceo del árbol.

rightRotation(*Node<K, V> node*)

"Este método se encarga de realizar una rotación hacia la derecha con base al caso presentado"
Pre: Debe existir el árbol RB para la implementación del método
Post: Se ha realizado una rotación a la derecha en bien del balanceo del árbol.

insertRB(RBNode<K, V> node)

"Este método se encarga de realizar la inserción de un nodo buscando la posición correcta para ser agregado"
Pre: Debe existir el árbol RB para la implementación del método
Post: Se ha realizado la inserción de un nodo en el árbol RB.

getBiggerThan*(K k)*

“Este método se encarga de dar los valores de los índices más altos dependiendo de la búsqueda de determinado rubro”
Pre: Debe existir el árbol RB para la implementación del método
Post: Se devuelve una lista con los índices más altos con base a la búsqueda realizada.

getBiggerThan*(K k)*

“Este método se encarga de dar los valores de los índices más bajos dependiendo de la búsqueda de determinado rubro”
Pre: Debe existir el árbol RB para la implementación del método
Post: Se devuelve una lista con los índices más bajos con base a la búsqueda realizada.

searchCloserRB*(K key, Node<K, V> node)*

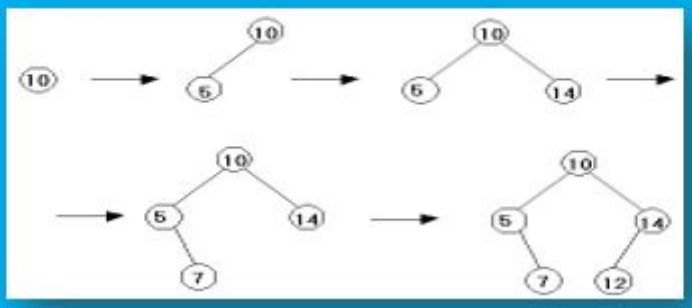
“Este método se encarga de dar los nodos más cercanos al índice buscado”
Pre: Debe existir el árbol RB para la implementación del método
Post: Se devuelve una lista de nodos con los resultados más cercanos a la búsqueda realizada.

searchCloser(*K key*)

“Este método se encarga de llamar el método searchCloserRB para la realización de la búsqueda más cercana con base al índice dado.”
Pre: Debe existir el árbol RB para la implementación del método
Post: Se devuelve una lista de nodos con los resultados más cercanos a la búsqueda realizada.

searchRB(*K key*)
"Este método realizar la búsqueda de manera privada por índice"
Pre: Debe existir el árbol para la implementación del método
Post: Se devuelve un nodo en forma de lista dada la búsqueda por índice.

TAD BSTree



{inv1: ∀ *a* | suba1 > suba2 > suba3 > ... > suban}

- Constructor: __ BSTree (Constructor)
- getRoot: BSTree __ Elem(Analizador)
- search: BSTree x key __ Elem(Analizador)
- insert: BSTree x node __ BSTree (Modificador)
- getLessThan: BSTree x key __ List(Analizador)
- getBiggerThan: BSTree x key __ List(Analizador)
- searchCloserBS: BSTree x key x node __ Node(Analizador)
- searchCloser: BSTree x key __ Node(Analizador)

getRoot()

"Este método se encarga de dar la raíz el árbol"
Pre: Debe existir el árbol BS para la implementación del método
Post: Se devuelve un elemento del árbol catalogado como raíz.

insert(Node<K, V> novo)

"Este método se encarga de realizar la inserción de un nodo buscando la posición correcta para ser agregado"

Pre: Debe existir el árbol BS para la implementación del método
Post: Se ha realizado la inserción de un nodo en el árbol BS.

getBiggerThan*(K k)*

"Este método se encarga de dar los valores de los índices más altos dependiendo de la búsqueda de determinado rubro"
Pre: Debe existir el árbol BS para la implementación del método
Post: Se devuelve una lista con los índices más altos con base a la búsqueda realizada.

getBiggerThan*(K k)*

"Este método se encarga de dar los valores de los índices más bajos dependiendo de la búsqueda de determinado rubro"
Pre: Debe existir el árbol BS para la implementación del método
Post: Se devuelve una lista con los índices más bajos con base a la búsqueda realizada.

searchCloserBS*(K key, Node<K, V> node)*

"Este método se encarga de dar los nodos más cercanos al índice buscado"
Pre: Debe existir el árbol BS para la implementación del método
Post: Se devuelve una lista de nodos con los resultados más cercanos a la búsqueda realizada.

searchCloser(*K key*)

"Este método se encarga de llamar el método searchCloserBS para la realización de la búsqueda más cercana con base al índice dado."
Pre: Debe existir el árbol BS para la implementación del método
Post: Se devuelve una lista de nodos con los resultados más cercanos a la búsqueda realizada.

Diseño de pruebas

AVLTreeTest

Scene1(): *Un árbol AVL de tamaño 8, instanciando sus índices con su respectivo elemento contenido.*
Scene2(): *Un árbol AVL con índice desde 0 hasta 500 y con elementos basados en un carácter por el índice a medida que avanza la distribución del índice.*
Scene3(): *Un árbol AVL vacío.*

Método	Escenario	Entrada	Salida
searchTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna.	Se prueba que el árbol AVL si busca la clave buscada para el elemento correspondiente y en caso de no encontrarse se retorna un null
insertTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol AVL inserte de manera correcta un nodo dependiendo del caso en el que se presente
getBiggerThanTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol AVL encuentre los índices más altos para x parámetro característico de la búsqueda.
getLessThanTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol AVL encuentre los índices más bajos para x parámetro característico de la búsqueda.
searchCloserTest()	*Scene1()* *Scene2()*	Ninguna	Se prueba que el árbol AVL

	Scene3()		encuentre los valores más cercanos al valor buscado.

BSTreeTest

Scene1(): *Un arbol de busqueda binaria de tamaño 8, instanciando sus índices con su respectivo elemento contenido.*
Scene2(): *Un árbol de búsqueda binaria con índice desde 0 hasta 500 y con elementos basados en un carácter por el índice a medida que avanza la distribución del índice.*
Scene3(): *Un árbol de búsqueda binaria vacío.*

Método	Escenario	Entrada	Salida
searchTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna.	Se prueba que el árbol BS si busca la clave buscada para el elemento correspondiente y en caso de no encontrarse se retorna un null
insertTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol BS inserte de manera correcta un nodo dependiendo del caso en el que se presente
getBiggerThanTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol BS encuentre los índices más altos para x parámetro característico de la búsqueda.
getLessThanTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol BS encuentre los

			índices más bajos para x parámetro característico de la búsqueda.
searchCloserTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol BS encuentre los valores más cercanos al valor buscado.

RBTreeTest

Scene1(): *Un árbol RB de tamaño 8, instanciando sus índices con su respectivo elemento contenido.*
Scene2(): *Un árbol RB con índice desde 0 hasta 500 y con elementos basados en un carácter por el índice a medida que avanza la distribución del índice.*
Scene3(): *Un árbol RB vacío.*

Método	Escenario	Entrada	Salida
searchTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna.	Se prueba que el árbol RB si busca la clave buscada para el elemento correspondiente y en caso de no encontrarse se retorna un null
insertTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol RB inserte de manera correcta un nodo dependiendo del caso en el que se presente

getBiggerThanTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol RB encuentre los índices más altos para x parámetro característico de la búsqueda.
getLessThanTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol RB encuentre los índices más bajos para x parámetro característico de la búsqueda.
searchCloserTest()	*Scene1()* *Scene2()* *Scene3()*	Ninguna	Se prueba que el árbol RB encuentre los valores más cercanos al valor buscado.

Diagrama de pruebas unitarias:

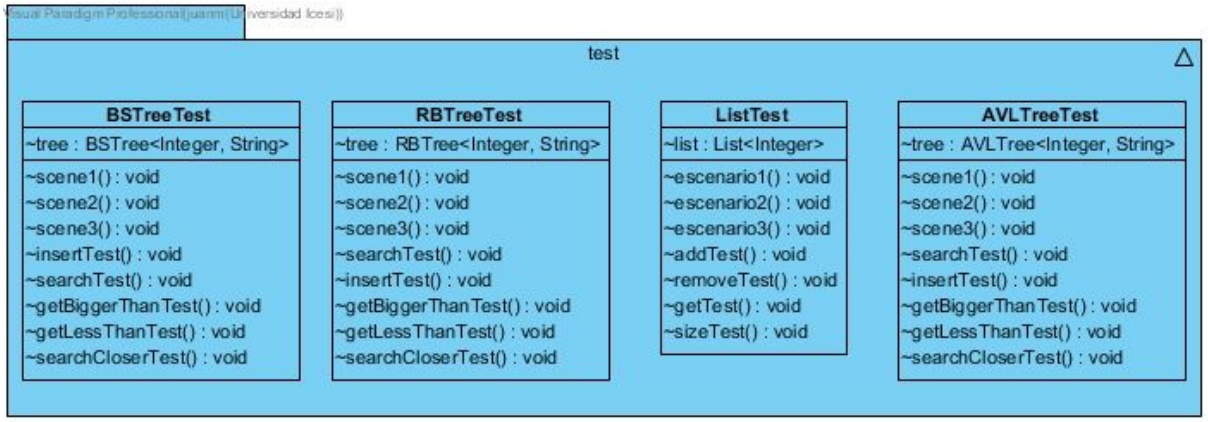


Diagrama del paquete NBAUtil