**Algoritmos y Estructuras de Datos**

**Laboratorio 3 Unidad 1**

Nelson López

Carlos Lizalda.

Santiago Chasqui.

1. **Identificación del problema:** La Federación Internacional de Baloncesto (FIBA) ha solicitado desarrollar un programa que permita administrar la información de sus jugadores.

El sistema debe estar en la capacidad de:

1. **Ingresar Información de los jugadores.**

El sistema debe estar en la capacidad de recibir la información de los jugadores de dos formas diferentes: Como archivos con extensión .csv, en donde se ingresaron grandes cantidades de datos; o a través de una interfaz gráfica en la cual el usuario pueda ingresar individualmente la información de un jugador.

En ambos casos las entradas deben ser las siguientes:

* Nombre (Cadena de texto).
* Edad (Enteros positivos).
* Equipo (Cadena de texto).
* 5 rubros estadísticos (Números de coma flotante).

Esta funcionalidad no tiene ninguna salida.

1. **Consultar información de los jugadores.**

El programa debe estar en la capacidad de hacer consultas de grupos de jugadores que cumplan ciertas condiciones, ya sea de características físicas, de dichos jugadores o estadísticas de los mismos. Las entradas dependerán del criterio de búsqueda elegido por el usuario, pero estarán restringidos a números reales. Finalmente, el sistema retornará el grupo de jugadores, que cumplan las condiciones establecidas por el criterio de búsqueda indicado por el usuario.

1. **Eliminar información de los jugadores.**

El sistema también podrá eliminar los jugadores que ya están almacenados. Para ello el usuario indicará el nombre exacto del jugador que se quiere eliminar.

La única entrada es:

* Nombre del jugador (Cadena de texto).

Esta función no produce ninguna salida.

1. **Modificar la información de los jugadores.**

El programa debe permitir la modificación de la información de los jugadores presentes en el sistema. Los cambios se llevarán a cabo buscando al jugador indicado por el usuario y mostrando la información que ya tiene para que así el usuario pueda seleccionar la información a modificar.

La única entrada es:

* Nombre del jugador (Cadena de texto).

Se avisará al usuario si la información a modificar no/sí se hizo se hizo efectiva.

|  |  |
| --- | --- |
| **R1** | Ingresar información de los jugadores. |
| Resumen | El programa debe estar en la capacidad de recibir la información de los jugadores de dos formas diferentes: Como archivos con extensión .csv, en donde se ingresaron grandes cantidades de datos; o a través de una interfaz gráfica en la cual el usuario pueda ingresar individualmente la información de un jugador. |
| Entrada | * Los datos de los jugadores:   + Nombre.   + Edad.   + Equipo.   + 5 rubros estadísticos. |
| Salida |  |

|  |  |
| --- | --- |
| **R1** | Consultar información de los jugadores. |
| Resumen | Consulta la información de los jugadores de acuerdo a un criterio de búsqueda establecido. |
| Entrada | Jugadores pertenecientes a la FIBA. |
| Salida | Información de los jugadores que coincidieron con el criterio establecido. |

|  |  |
| --- | --- |
| **R2** | Eliminar información de los jugadores |
| Resumen | Elimina la información de cualquier jugador. |
| Entradas | Jugadores |
| Salida | Se eliminó la información indicada. |

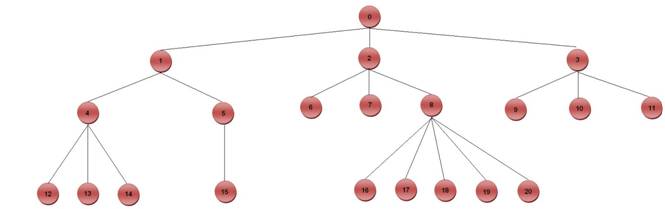
|  |  |
| --- | --- |
| **R3** | Modificar la información de los jugadores. |
| Resumen | Modifica la información de un jugador o grupo de jugadores. |
| Entradas | Jugadores |
| Salidas | Se modificó la información indicada. |

**Requerimientos no funcionales:**

* La información de los jugadores debe guardarse en memoria secundaria.
* La complejidad temporal del algoritmo de búsqueda no puede ser O(n) ni mucho menos mayor.
* Debe ser posible consultar la información de los jugadores bajo diferentes criterios de búsqueda.

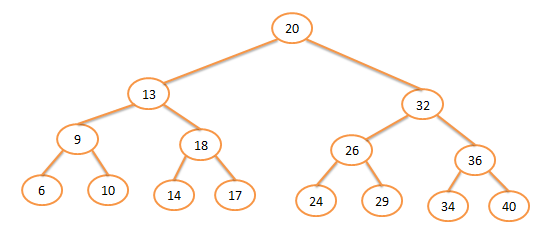
**2. Recopilación de la información:** A continuación se describirán los conceptos formales necesarios para abordar el problema de manera pertinente:

* **Estructura de Datos:** En términos sencillos, una estructura de datos una forma de organizar un conjunto de datos con el objetivo de facilitar su manipulación. Entre las estructuras de datos más conocidas se encuentran:
* **Arbole n-arios:** “Un árbol n-ario es una estructura recursiva en la que cada elemento puede tener cualquier número de subárboles n-arios asociados. En este caso el orden de los subárboles no es importante, en el sentido de que no es necesario saber cuál es el primero o el último, sino simplemente saber que es un subárbol.” (S., 2006). A continuación se muestra una representación gráfica de un árbol n-ario.



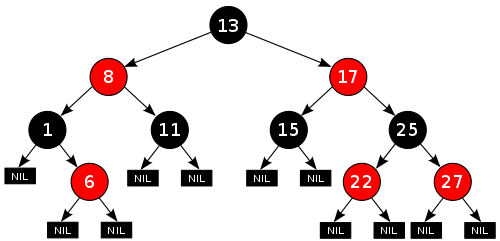
(Asencio, 2011)

* + **Arboles binarios:** Un árbol binario es una estructura de datos no lineal en la que cada nodo puede apuntar a uno o máximo dos nodos. Usualmente se suelen utilizar los arboles binarios ordenados o también denominados arboles binarios de búsqueda que son estructuras que presentan un orden definido. A continuación se muestra una representación gráfica de este tipo de estructura:

(Castrillon, 2015)

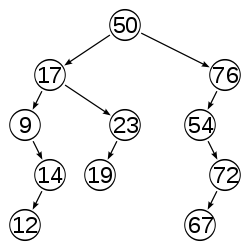
* + **Arboles Rojinegros:** Un árbol rojinegro es una estructura de datos que similar a un árbol ABB pero que presenta una característica adicional relacionada con el “color” de los nodos, los cuales deben cumplir con las siguientes características:
    - Un nodo solo puede ser rojo o negro.
    - El nodo raíz del árbol siempre debe ser negro.
    - Cada posible camino desde la raíz del árbol a sus hojas debe tener tanto el mismo número de nodos rojos como negros.
    - Un nodo solo puede tener sus hijos de color negro.

A continuación se muestra una representación gráfica de este tipo de árbol:



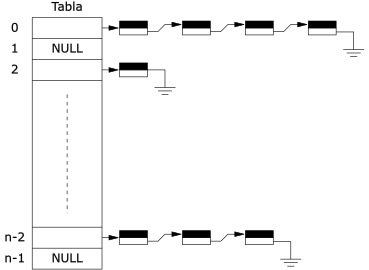
(Serrano, 2013)

* + **Arboles AVL:** Un árbol AVL conserva las mismas características de un árbol binario pero con la condición de que la longitud de sus ramas solo pueden ser mayores o menores en una unidad. Esta condición hace que este tipo de árbol binario se mantenga balanceado. A continuación se muestra una representación gráfica de este tipo de estructura:

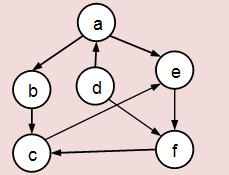


(Wikipedia, 2013)

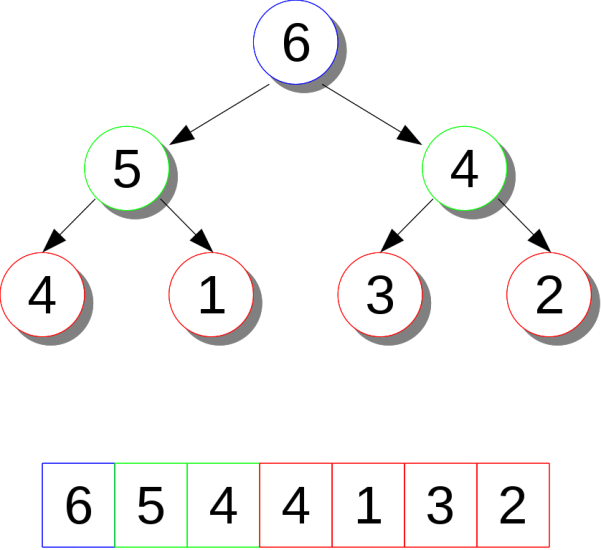
* + **Tablas Hash:** “Es una estructura de datos no lineal cuyo propósito final se centra en llevar a cabo las acciones básicas (inserción, eliminación y búsqueda de elementos) en el menor tiempo posible, mejorando las cotas de rendimiento respecto a un gran número de estructuras.” (Heileman, 1994). La siguiente es una representación gráfica de esta estructura de datos:

 (uc3m, 2017)

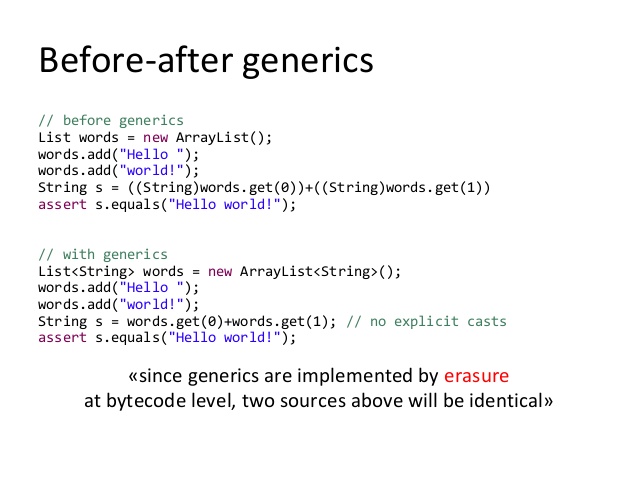
* + **Grafos:** Un grafo es una estructura de datos compuesta por un conjunto de nodos (también llamados vértices) y un conjunto de arcos (aristas) que establecen relaciones entre nodos y que permiten modelar situaciones de la vida real que implican numerosos aspectos relacionados entre sí. La grafica siguiente representa dicha estructura de datos:

 (Serrano, 2013)

* + **Montículo:** Un montículo es una estructura de datos que cumple con las condiciones de un árbol binario balanceado, pero que se implementa en un array que permite conservar estas propiedades. La siguiente es una representación gráfica de un heap:

 (Wikipedia, 2013)

* **Interfaz:** Se compone de un conjunto de declaraciones de signaturas de **métodos sin implementar** que especifican un **comportamiento**  específico para una o varias clases con la ventaja de que una clase puede implementar diferentes interfaces.
* **Generics:** Permiten asignar parámetros a clases, interfaces o métodos de forma que sólo admitan tipos de objetos ya especificados. El siguiente es un ejemplo de implementación de generics:



*(slideshare, 2012)*

* **Tipos de Datos Abstractos:** Se diferencian de los tipos de datos en el sentido de que los datos abstractos son especificados de manera precisas y diseñados independientemente de cualquier implementación, es decir que pueden ser implementados en cualquier lenguaje de programación ya que su forma de definirlos proporciona la información necesaria para hacerlo. El siguiente es un ejemplo de cómo se representa un tipo de dato abstracto de manera formal:

****

*(Villalobos, 1996).*

* **Pruebas Unitarias:** Una prueba unitaria es un mecanismo que permite comprobar la eficacia de determinado software. Esto se lleva a cabo creando diferentes casos de prueba que, al ejecutarlos en el programa, corroboran el correcto funcionamiento del mismo. El siguiente es un diseño formal de pruebas unitarias:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase** | **Método** | **Escenario** | **Valores de Entrada** | **Resultado** |
| Tablero | testTraducirNotacion() | escenario1 | jugada =a7a6  convertida =6858 | Verdadero. La notación se tradujo en coordenada de fila y columna |
| Tablero | testMoverFicha() | escenario1 | jugada= g8h6 | Verdadero. La ficha se movió a la casilla de llegada. |
| Tablero | testNotacionInvalidaExcepcion() | escenario1 | jugada= sdfsdfsd | Verdadero. El mensaje de excepción corresponde al de notación invalida |

*Basado en el Libro J. Villalobos en [Villalobos, 1996].*

**3. Búsqueda de soluciones creativas:**

Existen numerosas estructuras de datos que se encargan de almacenar información de una manera particular, haciendo que cada una de ellas tenga diferentes características que pueden adaptarse de mejor manera a cierto tipo de problemas dependiendo de las condiciones y necesidades que proporcione el mismo.

Teniendo en cuenta lo anterior se procederá a hacer uso de una técnica conocida como lluvia de ideas para enlistar algunas de las opciones que a primera vista son factibles para resolver de manera pertinente el problema abordado:

**Alternativa 1: Tabla Hash**

Esta es una estructura de datos que se caracteriza porque funciones como buscar o insertar son notablemente rápidos, haciendo que sea una excelente alternativa para solucionar algunos aspectos del problema ya que se requiere un nivel de eficiencia aceptable en los algoritmos y estructuras usadas en la solución del problema.

**Alternativa 2: Montículos**

**“**La estructura de datos conocida como montículo es un arreglo de objetos que podemos ver como un árbol binario casi completo. Cada nodo del árbol corresponde a un elemento del arreglo” (Thomas H. Cormen, 2009)**.** Esta estructura de datos también podría incluirse en el grupo de candidatos a suplir la funcionalidad de almacenar los jugadores de la liga, aunque dada la gran cantidad de jugadores a almacenar, puede resultar ineficiente el uso de esta estructura.

**Alternativa 3: Árbol Binario**

“Un árbol binario es un conjunto finito de elementos, el cual está vacío o dividido en tres subconjuntos separados: El primer subconjunto contiene un elemento único llamado raíz del árbol. El segundo subconjunto es en sí mismo un árbol binario y se le conoce como subárbol izquierdo del árbol original. El tercer subconjunto es también un árbol binario y se le conoce como subárbol derecho del árbol original“ (Serrano, 2013). Este tipo de estructura es una fuerte candidata debido a que sus operaciones de búsqueda, eliminación e inserción son notablemente rápidas. La desventaja de esta estructura de datos es que no es auto-balanceada y ello puede traer problemas en la eficiencia de las operaciones elementales.

**Alternativa 4: Árbol n-ario**

“Un árbol n-ario es una estructura recursiva, en la cual cada elemento tiene un número cualquiera de árboles n-arios asociados. Estos árboles corresponden a la generalización de un árbol binario. La diferencia radica en que esta estructura puede manejar múltiples subárboles asociados a cada elemento, y no solamente 2, como en el caso de los árboles binarios.” (Blanco, 2017). Esta estructura de datos es poco viable para el problema que se desea resolver, debido a que esta estructura no permite mantener un criterio de orden determinado y esa es la principal condición del software a crear.

**Alternativa 5: Grafos**

“Los grafos no son más que la versión general de un árbol, es decir, cualquier nodo de un grafo puede apuntar a cualquier otro nodo de éste (incluso a él mismo).” (Asencio, 2011). Esta estructura de datos puede usarse para almacenar los jugadores de la liga, aunque esta alternativa puede resultar poco viable debido a que los grafos se caracterizan por tener múltiples relaciones entre los elementos que almacena y este es un aspecto totalmente innecesario para la resolución de este problema en particular.

**Alternativa 6: Arboles Rojinegros.**

“Un árbol rojo-negro es un árbol de búsqueda binario con un bit extra de almacenamiento por nodo: Su color, que puede ser rojo o negro” (Thomas H. Cormen, 2009). Esta es una estructura de datos idéntica a un árbol binario pero con ciertas características adicionales que le permiten tener la facultad de auto-balancearse por sí solos. Esta es una alternativa muy viable debido a que puede soportar las dos principales condiciones del problema en cuestión: Un criterio especifico de orden y una notable eficiencia en las operaciones de los elementos de la estructura.

**Alternativa 7: Arboles AVL.**

“Un árbol AVL es un árbol binario de búsqueda con altura equilibrada: Para cada nodo x, las alturas de los subárboles izquierdos y derechos de x difieren como máximo 1.” (Thomas H. Cormen, 2009). Esta también resulta una alternativa a tener en cuenta de cara a la resolución del problema en cuestión debido a su condición de auto-balanceo, permitiendo que la estructura se mantenga equilibrada y que sea posible mantener una notoria eficiencia en las operaciones principales que manipulan los elementos almacenados.

**4. Transición de las ideas a los diseños preliminares**

A continuación se evaluarán individualmente cada una de las alternativas planteadas en el inciso anterior y se procederá a descartar las propuestas menos factibles teniendo en cuenta los requerimientos del problema abordado, es decir, sus funcionalidades y condiciones de eficiencia. Para ello se procederá a aplicar la técnica de revisión selectiva que consiste en describir los aspectos relevantes de la información abordada dividiéndolos en dos secciones conocidas como pros y contras:

|  |  |  |
| --- | --- | --- |
| **Alternativa** | **Ventajas** | **Desventajas** |
| 1. **Tabla Hash** | Los algoritmos que manipulan este tipo de estructura son muy eficientes dado que tienen una complejidad temporal O(1). | El proceso de redimensionamiento de la tabla resulta una maniobra costosa de implementar. Además, no es posible establecer un criterio de ordenamiento de los elementos almacenados. |
| 1. **Montículo** | Los objetos guardados pueden ser organizados de manera ascendente o descendente. La ordenación se realiza con una complejidad temporal relativamente óptima. | La complejidad temporal puede llegar a ser ineficiente si se trata de un número grande de elementos, hecho que hace imposible la implementación de esta estructura en el problema en cuestión. |
| 1. **Árbol Binario** | Su funcionamiento puede emparejarse directamente con situaciones de la vida real. La información puede guardarse de manera ordenada. Puede redimensionarse sin mayor problema | Si un árbol no está equilibrado, sus funcionalidades pueden no ser eficientes. No es posible usarlo para datos que no requieran un orden. Además no cumple con la propiedad de auto-balanceo. |
| 1. **Árbol n-ario** | Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Puede redimensionarse sin mayor problema. | No es posible guardar información de manera ordenada, aunque si de manera jerárquica. |
| 1. **Grafos** | Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Todos sus nodos pueden apuntar a otro nodo entre sí, incluso puede apuntar al mismo. Puede redimensionarse fácilmente**.** | Puede ser una estructura demasiado robusta en cuanto a especificaciones espaciales y temporales. No es posible establecer un criterio de ordenamiento entre los elementos almacenados. |
| 1. **Árbol Rojinegro** | Su principal ventaja es que puede auto-balancearse, además de que es posible establecer un criterio de orden entre los elementos. | Las operaciones de eliminación e inserción suelen ser difíciles de entender y por ende de implementar. |
| 1. **Árbol AVL** | Su principal ventaja es que puede auto-balancearse, además de que es posible establecer un criterio de orden entre los elementos. | Las operaciones de eliminación e inserción suelen ser difíciles de entender y por ende de implementar. |

Teniendo en cuenta el análisis anterior, resulta necesario descartar las estructuras de datos cuyo funcionamiento no proporciona una solución óptima al problema planteado. Este es el caso de los grafos y el montículo debido a que la complejidad temporal de sus algoritmos es elevada comparada con otras estructuras, además de que este problema puede tornarse más grave cuando se manipulan grandes cantidades de datos.

**5. Evaluación y Selección de la Mejor Solución:**

A continuación se expondrán los criterios mediante los cuales se elegirán las estructuras de datos a utilizar en la resolución del problema en cuestión:

**Criterio 1:** Complejidad temporal de los algoritmos que manipulan la estructura de datos.

|  |  |
| --- | --- |
| **Complejidad Temporal** | **Puntuación** |
| O (logn) | 4 |
| O (n+k) | 2 |
| O (1) | 6 |

**Criterio 2:** Criterio de ordenamiento entre sus elementos.

|  |  |
| --- | --- |
| **Orden entre sus elementos** | **Puntuación** |
| Si | 10 |
| No | 5 |

**Criterio 3:** Propiedad de auto-balanceo de la estructura de datos.

|  |  |
| --- | --- |
| **Propiedad de auto-balanceado** | **Puntuación** |
| Auto-balanceado | 10 |
| No auto-balanceado | 2 |

En el siguiente recuadro se mostraran las diferentes alternativas que, luego de evaluarlas con los criterios descritos, se elegirán las que cumplan con las condiciones establecidas con mayor rigurosidad.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Criterio 1** | **Criterio 2** | **Criterio 3** | **Total** |
| Alternativa 1. Tabla Hash | 6 | 5 | 2 | 13 |
| Alternativa 2. Árbol Binario | 4 | 10 | 2 | 16 |
| Alternativa 3: Árbol n-ario | 4 | 5 | 2 | 11 |
| Alternativa 4. Árbol rojinegro | 4 | 10 | 10 | 24 |
| Alternativa 5. Árbol AVL | 4 | 10 | 10 | 24 |

**Selección definitiva:** Teniendo en cuenta el método de selección utilizado anteriormente, se llegó a la conclusión de que las estructuras que deberán ser usadas en la creación del software son las siguientes: Árbol ARN y árbol AVL. La principal razón por la que se eligieron estas estructuras es porque su complejidad temporal es baja y además porque es posible mantener un criterio de ordenamiento entre sus elementos.

**6. Preparación de informes y especificaciones:**

**Especificación del problema:**

**· Problema:** Desarrollar un programa que permita administrar la información concerniente a la liga de baloncesto NBA.

**· Entrada:** Información de los jugadores de baloncesto y de cada uno de los partidos jugados.

**· Salidas:** Información detallada de cada jugador de la liga, incluyendo sus datos y estadísticas deportivas.

Como parte de las especificaciones de las estructuras de datos utilizadas para resolver el problema se definieron los TADs de cada una de ellas:

|  |
| --- |
| **TAD Árbol Rojo-Negro** |
|  |
| X es el conjunto domino de los valores con un tipo de dato particular  U es el universo de llaves |
| * CrearARN() -> <Arbol Rojo-Negro> * Insertar (U, X) * Buscar (U) -> <X> * Eliminar (U) |

|  |
| --- |
| **TAD Árbol AVL** |
|  |
| X es el conjunto domino de los valores con un tipo de dato particular  U es el universo de llaves |
| * CrearAVL() -> <AVL> * Insertar (U, X) * Buscar (U) -> <X> * Eliminar (U) |

|  |
| --- |
| **Insertar (U, X)** |
| \*Agrega un nuevo elemento al árbol, lo ubica de acuerdo a un criterio de orden específico y se balancea la estructura si es necesario.  {pre: La clave debe existir dentro del universo de llaves}  {post: Se agregó un elemento a al árbol de acuerdo al criterio especificado} |

|  |
| --- |
| **Buscar (U)** |
| \*Busca el elemento dentro del árbol de acuerdo a un criterio de búsqueda especifico o usando la clave asociada al elemento en cuestión.  {pre: root!=null}  {post: Se retornó un elemento con la clave o el criterio de búsqueda ingresado como parámetro. } |

|  |
| --- |
| **Eliminar (U)** |
| \*Elimina el elemento del árbol que corresponda a la clave o el criterio U de búsqueda.  {pre: existe }  {post: Se eliminó el objeto con el criterio o clave ingresada como } |

|  |
| --- |
| **Obtener (criterio)** |
| \*Retorna el elemento al que le pertenece el criterio de búsqueda ingresado como parámetro.  {pre: }  {post: Se ha encontrado el elemento con el criterio ingresado como parámetro} |

|  |
| --- |
| **Balancear (subárbol)** |
| \*Retorna el subárbol ingresado ya balanceado.  {pre: null}  {post: Se balanceó el subárbol ingresado como parámetro} |
| **ObtenerColor ()** |
| \*Retorna el color de un nodo determinado.  {pre: null}  {post: Se balanceó el subárbol ingresado como parámetro} |

|  |
| --- |
| **estaBalanceado ()** |
| \*Retorna verdadero om.  {pre: null}  {post: Se balanceó el subárbol ingresado como parámetro} |

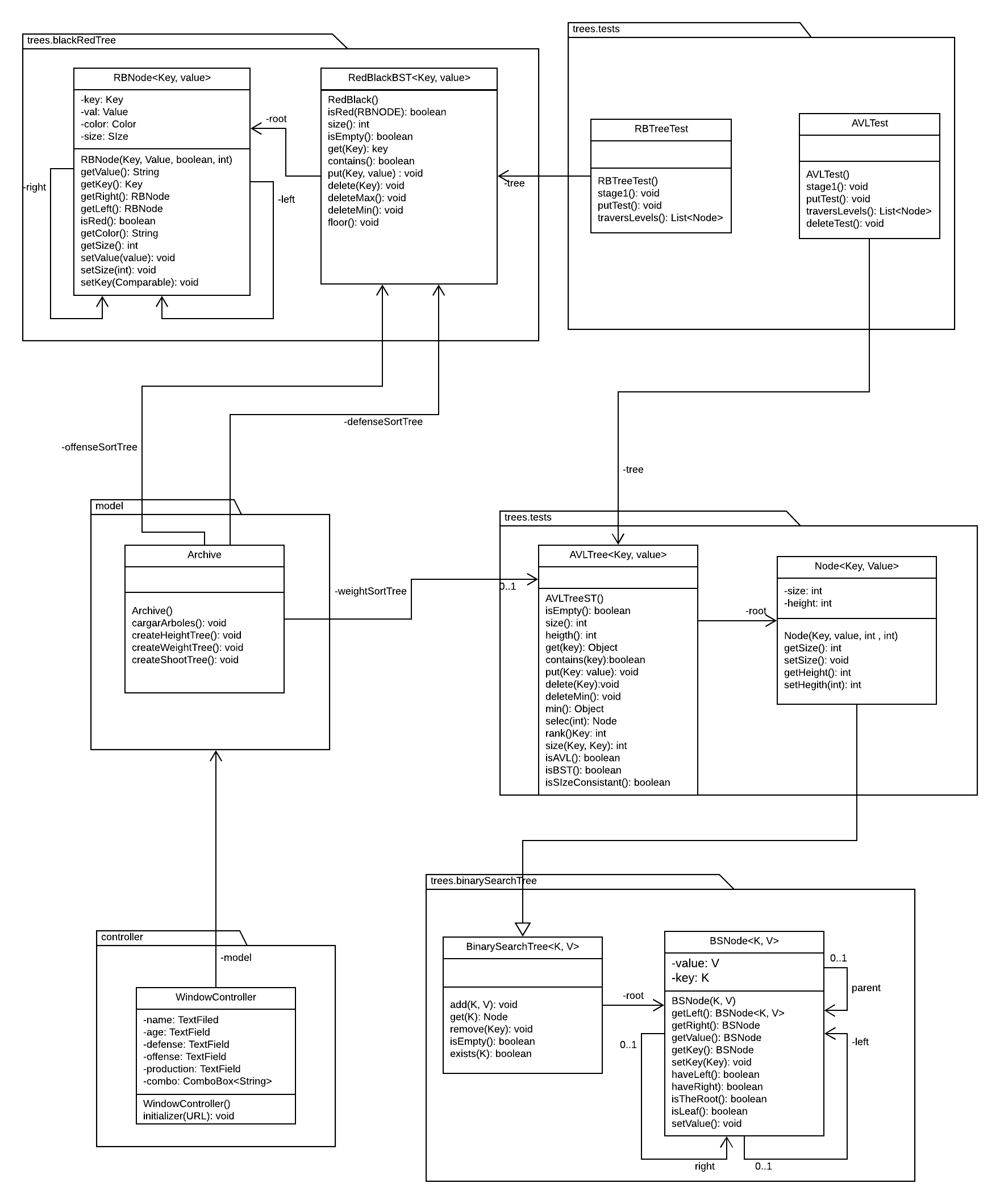
|  |
| --- |
| **estaBalanceado ()** |
| \*Retorna verdadero om.  {pre: null}  {post: Se balanceó el subárbol ingresado como parámetro} |

|  |
| --- |
| **cambiarColor ()** |
| \* Se modificaron los colores de los nodos de un árbol Rojo-Negro de acuerdo a las condiciones propias de este tipo de estructura.  {pre: null}  {post: Se modificaron los colores de los nodos de un árbol Rojo-Negro} |

|  |
| --- |
| **rotarIzq (Nodo)** |
| \*Se rota hacia la izquierda el subárbol cuya raíz es el nodo ingresado como parámetro.  {pre: null}  {post: Se roto el subárbol hacia la izquierda} |

|  |
| --- |
| **rotarDer (Nodo)** |
| \*Se rota hacia la izquierda el subárbol cuya raíz es el nodo ingresado como parámetro.  {pre: null}  {post: Se roto el subárbol hacia la derecha} |

Adicional a ello se elaboró una representación gráfica del software a desarrollar denominado diagrama de clases, con el fin de tener una visión más clara del programa y cómo será la lógica de su funcionamiento.



**7. Implementación del Diseño:**

A continuación se presentara una lista de las tareas a implementar por el programa:

1. Ingresar información de los jugadores de la liga.
2. Eliminar información de los jugadores de la liga.
3. Consultar información de un jugador específico.

|  |  |
| --- | --- |
| **Especificación de subrutinas** | **Construcción** |
| |  |  | | --- | --- | | **Nombre** | add | | **Descripción** | Ese método se encarga de añadir un nuevo jugador al árbol de jugadores de la liga NBA. | | **Entrada** | Información del nuevo jugador a añadir. | | **Retorno** | Sin salida. | | |  | | --- | | **public** **void** add(K key, V value) {  **if** (!isEmpty())  {  add(key, value, root);  }  **else**  {  root = **new** BSNode<K, V>(key, value);  }  } | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |
| |  |  | | --- | --- | | **Nombre** | get | | **Descripción** | Este método se encarga de devolver un jugador con la clave ingresada como parámetro | | **Entrada** | Clave del jugador a buscar | | **Retorno** | Jugador correspondiente a la clave asociada como parámetro. | | **public** V get(K key) {  **if** (root != **null**)  {  BSNode<K, V> temp = get(key,root);  **if** (temp != **null**)  {  **return** temp.getValue();  }  **else**  {  **return** **null**;  }  }  **return** **null**;  } |
| |  |  | | --- | --- | | **Nombre** | remove | | **Descripción** | Este método se encarga de eliminar el jugador cuya clave es ingresada como parámetro. | | **Entrada** | Clave del jugador a eliminar. | | **Retorno** | Sin salida. | | **public** V remove(K key){  **return** remove(key, root);  }  **private** V remove(K key, BSNode<K, V> startNode) {  BSNode<K, V> node = get(key,startNode);  **if**(node != **null**)  {  V value = node.getValue();  **if** (node.isLeaf())  {  value = node.getValue();  **if** (node.isleftSon())  {  node.getParent().setLeft(**null**);  }  **else**  {  node.getParent().setRigth(**null**);  }  **return** value;  }  BSNode<K, V> aux = GetMOL(node);  aux.setLeft(node.getLeft());  aux.setRigth(node.getRigth());  aux.setParent(node.getParent());  **if** (node.isTheRoot())  {  root = aux;  }  **else** **if** (node.isRightSon())  {  aux.getParent().setRigth(aux);  }  **else**  {  aux.getParent().setLeft(aux);  }  **return** value;  }  **else**  {  **return** **null**;  }  } |
| |  |  | | --- | --- | | **Nombre** | balance | | **Descripción** | Este método se encarga de balancear el subárbol ingresado como parámetro. | | **Entrada** | Subárbol. | | **Retorno** | Subárbol balanceado. | | **private** Node<Key, Value> balance(Node<Key, Value> x) {  **if** (balanceFactor(x) < -1) {  **if** (balanceFactor((Node<Key, Value>) x.getRigth()) > 0) {  x.setRigth(rotaterigth((Node<Key, Value>) x.getRigth()));  }  x = rotateLeft(x);  }  **else** **if** (balanceFactor(x) > 1) {  **if** (balanceFactor((Node<Key, Value>) x.getLeft()) < 0) {  x.setLeft(rotateLeft((Node<Key, Value>) x.getLeft()));  }  x = rotaterigth(x);  }  **return** x;  } |
| |  |  | | --- | --- | | **Nombre** | getColorString | | **Descripción** | Este método se encarga de devolver el color de un nodo determinado. | | **Entrada** | Subárbol. | | **Retorno** | Subárbol balanceado. | | **public** String getColorString() {  **if**(color == ***RED***) **return** "R";  **return** "B";  } |
| |  |  | | --- | --- | | **Nombre** | flipColor | | **Descripción** | Este método se encarga de cambiar el color de los nodos de un árbol Rojo-Negro | | **Entrada** | Árbol Rojo-Negro. | | **Retorno** | Árbol Rojo-Negro con el color de los nodos modificado. | | **private** **void** flipColors(Node h) {  h.color = !h.color;  h.left.color = !h.left.color;  h.right.color = !h.right.color;  } |
| |  |  | | --- | --- | | **Nombre** | rotateLeft | | **Descripción** | Este método se encarga de rotar hacia la izquierda el subárbol ingresado como parámetro. | | **Entrada** | Raíz del subárbol a rotar | | **Retorno** | Sin salida | | // make a right-leaning link lean to the left  **private** Node rotateLeft(Node h) {  // assert (h != null) && isRed(h.right);  Node x = h.right;  h.right = x.left;  x.left = h;  x.color = x.left.color;  x.left.color = ***RED***;  x.size = h.size;  h.size = size(h.left) + size(h.right) + 1;  **return** x;  } |
| |  |  | | --- | --- | | **Nombre** | rotateRight | | **Descripción** | Este método se encarga de rotar hacia la derecha el subárbol ingresado como parámetro. | | **Entrada** | Raíz del subárbol a rotar. | | **Retorno** | Sin salida | | **private** Node rotateRight(Node h) {  // assert (h != null) && isRed(h.left);  Node x = h.left;  h.left = x.right;  x.right = h;  x.color = x.right.color;  x.right.color = ***RED***;  x.size = h.size;  h.size = size(h.left) + size(h.right) + 1;  **return** x;  } |

# Bibliografía

Asencio, A. (2011). *iuma.* Obtenido de http://www.iuma.ulpgc.es

Blanco, O. (2 de 12 de 2017). *oscarblancarteblog.* Obtenido de https://www.oscarblancarteblog.com/

Castrillon, C. (2015). *ocw.* Obtenido de ocw.opm.es

Heileman, G. L. (1994). *HeEstructuras de Datos, Algoritmos y Programación Orientada a Objetos.* Madrid: McGraw-Hill.

Serrano, M. (2013). Obtenido de https://www.infor.uva.es/~mserrano/EDI/cap5.pdf

Thomas H. Cormen, C. E. (2009). *Introduction to Alogrithms.* Londres: Massachusetts Institute of Technology.

*uc3m*. (02 de 06 de 2017). Obtenido de http://www.it.uc3m.es/