MÉTODO DE LA INGENIERÍA

**FASE 1: IDENTIFICACIÓN DEL PROBLEMA**

* **Descripción del contexto problemático (causas y síntomas).**

A medida que ha transcurrido el tiempo ha aumentado el interés por conocer patrones y describir el comportamiento de deportes como el baloncesto. Interés en entender cómo se comportan los jugadores en un partido, que patrones tiende a tener un jugador en un partido, cuantas cestas anota en un partido, cuantas asistencias realiza en un partido. Analizar todos estos datos en masa nos puede llevar a predecir comportamientos y ver hacia donde se dirige el baloncesto.

* **Identificación del problema.**

La Federación Internacional de Baloncesto, al ver el crecimiento de los datos que se pueden obtener respecto al comportamiento de un jugador en los partidos, como, por ejemplo, el número de cestas que realiza, el número de asistencias que realiza, etc., desea que se puedan analizar estos datos de manera que sean capaz de describir a un jugador, y de manera más general, obtener patrones acerca del desarrollo del deporte.

Para poder analizar de manera ideal estos datos, el acceso a ellos tiene que ser el más optimo.

* **Requerimientos funcionales**

|  |  |
| --- | --- |
| Nombre | R1 |
| Resumen | Buscar jugador por categoría. |
| Entrada | Categoría. |
| Resultado | Jugador de baloncesto. |

|  |  |
| --- | --- |
| Nombre | R2 |
| Resumen | Buscar jugador por estadística. |
| Entrada | Estadística a buscar. |
| Resultado | Jugador de baloncesto. |

|  |  |
| --- | --- |
| Nombre | R3 |
| Resumen | Buscar jugador por doble estadística. |
| Entrada | 2 Estadísticas a buscar. |
| Resultado | Jugador de baloncesto. |

|  |  |
| --- | --- |
| Nombre | R4 |
| Resumen | Mostrar tiempo de demora de consulta. |
| Entrada | Estadística a buscar. |
| Resultado | Tiempo de demora de la consulta. |

|  |  |
| --- | --- |
| Nombre | R5 |
| Resumen | Ingresar jugador. |
| Entrada | Jugador. |
| Resultado | Se agrega jugador a la estructura. |

|  |  |
| --- | --- |
| Nombre | R6 |
| Resumen | Eliminar jugador. |
| Entrada | Jugador. |
| Resultado | Se elimina jugador de la estructura. |

|  |  |
| --- | --- |
| Nombre | R7 |
| Resumen | Modificar jugador. |
| Entrada | Jugador. |
| Resultado | Se modifica jugador. |

|  |  |
| --- | --- |
| Nombre | R8 |
| Resumen | Mostrar tiempo de demora de consulta. |
| Entrada | Estadística a buscar. |
| Resultado | Tiempo de demora de la consulta. |

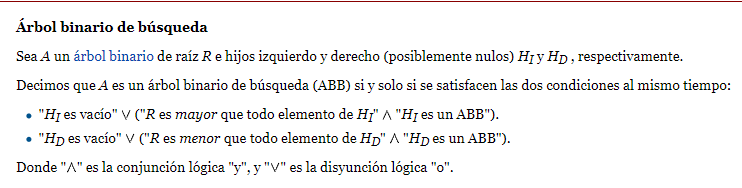
**FASE 2: RECOPILACION DE LA INFORMACION NECESARIA**

Para la resolución de este problema es de primera necesidad conocer y contextualizar los siguientes conceptos:

**Arboles binarios de búsqueda:**

Es un tipo particular de árbol binario que presenta una estructura de datos en forma de árbol usada en informática. Su comportamiento se rige bajo la siguiente propiedad:

* Siempre su hijo derecho va ser mayor o igual que su padre.
* Siempre su hijo izquierdo va ser menor o igual que su padre.



**Arboles rojo-negro:**

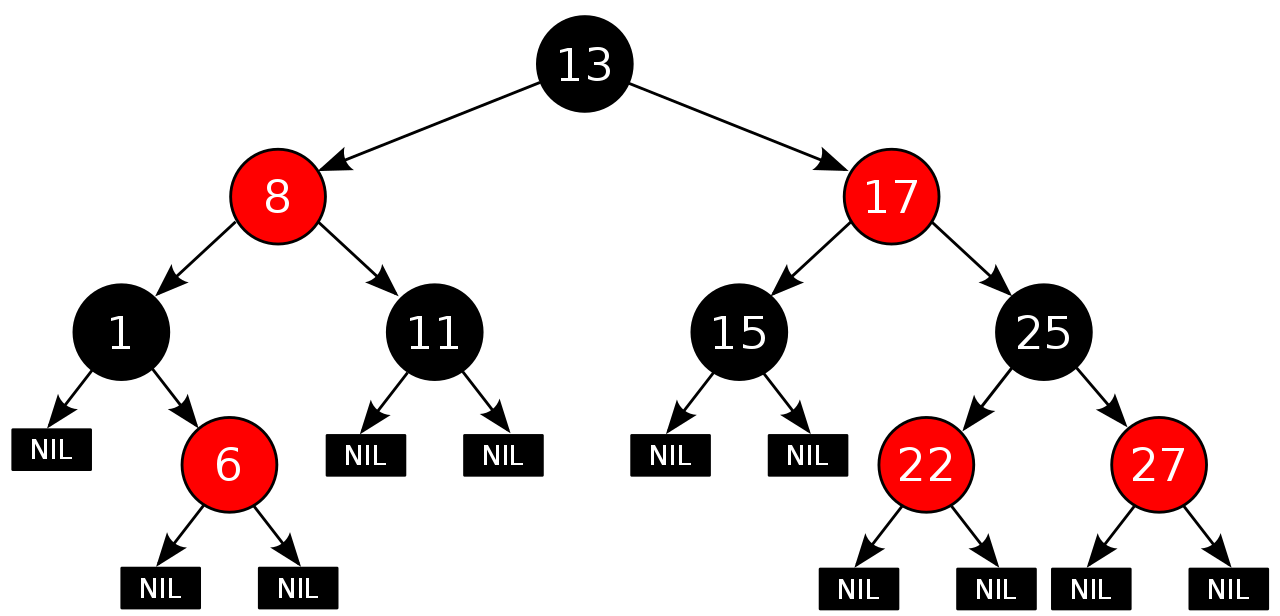
Un árbol rojo-negro es un tipo especial de árbol binario usado en informática para organizar información compuesta por datos comparables (por ejemplo, números). En los árboles rojo-negro las hojas no son relevantes y no contienen datos.

En los árboles rojo-negro, como en todos los árboles binarios de búsqueda, es posible moverse ordenadamente a través de los elementos de forma eficiente si hay forma de localizar el padre de cualquier nodo. El tiempo de desplazarse desde la raíz hasta una hoja a través de un árbol equilibrado que tiene la mínima altura posible es de O(log *n*).

Al implementar esta estructura es posible utilizar un único nodo centinela. Este cumple la función de hoja para todas las ramas del árbol. Así, todos los nodos internos que finalicen en una hoja tienen referencia a este único nodo centinela. Esto no es necesario, ya que puede hacerse una referencia nula (NIL) en el final de cada rama.

**Propiedades:**

* Todo nodo es o bien rojo o bien negro.
* La raíz es negra.
* Todas las hojas (NULL) son negras.
* Todo nodo rojo debe tener dos nodos hijos negros.
* Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.



**Arboles AVL:**

Las arboles AVL son arboles binario de búsqueda que se auto balancean. Los árboles AVL están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa. Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad [O](https://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica)(log n). El factor de equilibrio puede ser almacenado directamente en cada nodo o ser computado a partir de las alturas de los subárboles.

## **Factor de equilibrio**

Cada nodo, además de la información que se pretende almacenar, debe tener los dos punteros a los árboles derecho e izquierdo, igual que los árboles binarios de búsqueda (ABB), y además el dato que controla el factor de equilibrio.

El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo:

FE = altura subárbol derecho - altura subárbol izquierdo

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1.

Si el factor de equilibrio de un nodo es:

0 -> el nodo está equilibrado y sus subárboles tienen exactamente la misma altura.

1 -> el nodo está equilibrado y su subárbol derecho es un nivel más alto.

-1 -> el nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

**FASE 3: BÚSQUEDA DE SOLUCIONES CREATIVAS**

Nuestro problema de encontrar soluciones creativas se redujo drásticamente, dado que la situación especifica claramente que estructuras de datos debemos usar, las cuales son las siguientes:

* Arboles rojo negro
* Arboles AVL
* Arboles binarios de búsqueda

Ya habiendo solucionado el problema respecto a que estructuras íbamos a usar, entramos a la disputa acerca como vamos a cargar y a guardar las respectivas modificaciones que se le hagan a determinado jugador.

Estrategia utilizada: Lluvia de ideas.

Ideas:

* Utilizar una base de datos online.
* Utilizar un archivo de texto plano como base de datos.
* Utilizar un archivo CSV como base de datos.
* Utilizar un archivo EXCEL como base de datos.
* Tener la base de datos como un tipo de variable y serializarla junto a los cambios que se le hagan.
* Cada vez que se realice un cambio en los datos (Guardar, eliminar, modificar) registrarlo en la base de datos.
* Cargar nuestras estructuras de datos solo una vez y hacer persistir nuestro modelo a través de serialización.

**Situación 1)**

**¿Como vamos a cargar nuestros datos?**

**Idea 1)**

**Utilizar una base de datos online**

* Nuestra primera idea fue una usar una base de datos online, como forma de usar la memoria secundaria.

**Idea 2)**

**Cargar los datos desde un archivo de texto:**

* Fue la segunda idea de solución que tuvimos en cuenta, sin embargo, nos damos cuenta que visualmente va ser mejor ver un archivo de Excel que un archivo de texto, que es menos legible.

**Idea 3)**

**Cargar los datos desde un archivo de texto**

* La tercera idea fue cargar los datos desde un archivo Excel ya que este estructura datos en forma de filas y columnas, de manera que va serán más legibles.

**Idea 4)**

**Cargar los datos desde un archivo CSV**

* Nuestra cuarta idea fue cargar los datos desde un archivo CSV, este tipo de archivos está separado por comas, y especifica que significa lo que va después de la coma.

**Situación 2)**

**¿Cómo vamos a guardar nuestros archivos?**

**Idea 1)**

**Tener la base de datos como un tipo de variable y serializarla junto a los cambios que se efectúen.**

* Guardar con un tipo de variable todo mi estructura que almacena los jugadores, de este modo cuando la modifique la vuelvo a serializar.

**Idea 2)**

**Los cambios que se realicen se vuelven a registrar en la base de datos:**

* Cada vez que se realice algún tipo de modificación se vuelve a acceder a la base de datos y se vuelve a modificar la base de datos.

**Idea 3)**

**Hacer persistir el modelo a través de serialización:**

* Se agregaran los datos a nuestras estructuras solo una vez. Los cambios que se realicen a través del modelo (Agregar, eliminar y modificar) se guardaran por medio de serialización.

**FASE 4: TRANSICIÓN DE LA FORMULACIÓN DE IDEAS A LOS DISEÑOS PRELIMINARES**

**Situación 1)**

|  |  |
| --- | --- |
| **Idea no factible** | **Justificación** |
| Idea 1 | La complejidad de implementar el algoritmo en nuestro caso es alta ya que no tenemos los conocimientos en bases de datos para poder llevar a cabo esta implementación. |
| Idea 2 | Esta idea se descarta ya que existe una mejor manera de manejar datos con texto plano y es a través de archivos CSV. |

**Situación 2)**

|  |  |
| --- | --- |
| **Idea no factible** | **Justificación** |
| Idea 1 | La complejidad espacial al implementar esta idea aumentaría demasiado al tener nuestra base de datos en una variable y no en memoria secundaria. |

* Diseños preliminares.

**PENDIENTE**

**FASE 5: EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN**

**CRITERIOS**

* **Criterios situación 1**
  + **Legibilidad**
    - **[1]** Son poco entendibles los datos ahí almacenados**.**
    - **[2]** Son entendibles los datos ahí almacenados**.**
* **Criterios situación 2**
  + **Complejidad temporal**
    - **[5]** La mayoría de sus métodos tienen complejidad temporal constante**.**
    - **[4]** La mayoría de sus métodos tienen complejidad temporal mayor a constante**.**
    - **[3]** La mayoría de sus métodos tienen complejidad temporal logarítmica**.**
    - **[2]** La mayoría de sus métodos tienen complejidad temporal lineal.
    - [1] La mayoría de sus métodos tienen complejidad temporal mayor a lineal.
  + **Complejidad Espacial** 
    - **[5]** La mayoría de sus métodos tienen complejidad espacial constante**.**
    - **[4]** La mayoría de sus métodos tienen complejidad espacial mayor a constante**.**
    - **[3]** La mayoría de sus métodos tienen complejidad espacial logarítmica**.**
    - **[2]** La mayoría de sus métodos tienen complejidad espacial lineal.
    - [1] La mayoría de sus métodos tienen complejidad espacial mayor a lineal.

**SITUACION 1**

|  |  |  |
| --- | --- | --- |
|  | Legibilidad | Total evaluación |
| Idea 3 | **1** | **1** |
| Idea 4 | **2** | **2** |

**SITUACION 2**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Complejidad Temporal | Complejidad Espacial | Total evaluación |
| Idea 2 | 1 | 3 | 4 |
| Idea 3 |  | 3 |  |

**SELECCIÓN**

* **Situación 1:**

Dado que la idea 4 tuvo una evaluación mayor respecto a su legibilidad, será elegida como la forma en la cual vamos a cargar los datos.

* **Situación 2:**

Dado que la idea 2 tuvo una evaluación mayor respecto a su complejidad espacial y temporal, será la que elegiremos como método para guardar los datos.

**FASE 6: PREPARACIÓN DE INFORME Y ESPECIFICACIONES:**

* Diseño del diagrama de clases de la solución
* Diseño del diagrama de objetos
* Diseño de casos de las pruebas unitarias

**FASE 7: IMPLEMENTACIÓN**:

La solución se encuentra en el siguiente enlace:

* https://github.com/KingSatur/Laboratorio-3

**TAD’s**

|  |
| --- |
| **TAD ABB** |
|  |
| {*inv.: Se deben cumplir la siguiente propiedad de orden. (1) Todos los datos de su subárbol izquierdo son menos o iguales que el dato que ocupa la raíz. (2) Todos los datos de su subárbol derecho son mayores que el dato que ocupa la raíz. (3) Los subárboles izquierdo y derecho también son ABB*} |
| Operaciones Primitivas:   * CrearABB: ->ABB * DarRaiz: *ABB* ->Nodo * ModificarRaiz: *ABBxNodo* ->ABB * Buscar: *ABBxLlave* ->Nodo * Eliminar: *ABBxLlave* ->ABB * Insertar: *ABBxNodo* ->ABB * Sucesor: *ABBxLlave* ->Nodo * Predecesor: *ABBxLlave* ->Nodo * EsHoja: *ABBxLlave* ->Booleano * ArbolMinimo: *ABB* ->Nodo * ArbolMaximo: *ABB* ->Nodo |

|  |
| --- |
| **CrearABB()**  “Crea un árbol binario de búsqueda en el que la raíz es el primer elemento y no existe, por lo tanto, es nula”  {*pre:* TRUE}  {post*: Crea un árbol binario de búsqueda vacío*} |

|  |
| --- |
| **DarRaiz()**  “Da la raíz del árbol binario de búsqueda”  {*pre:* raíz!=nulo}  {*post: Se da la raíz del ABB*} |

|  |
| --- |
| **ModificarRaiz(**Nodo**)**  “Modificar el nodo que se encuentra como raíz del ABB”  {*pre:* ABB!=nulo}  {*post: Se modifica el nodo raíz del ABB por un nuevo nodo*} |

|  |
| --- |
| **Buscar(**Llave**)**  “Busca un nodo específico de acuerdo con la llave dada”  {*pre:* TRUE}  {*post: Si algún nodo.llave = Llave -> Se da este nodo,*  *Si no -> se devuelve nulo*} |

|  |
| --- |
| **Eliminar(**Llave**)**  “Elimina el nodo del ABB de acuerdo con la llave dada”  {*pre:* ABB!=nulo}  {*post: Si algún nodo.llave = llave -> Se elimina este nodo,*  *Si no -> no se hace ninguna modificación*} |

|  |
| --- |
| **Insertar(**Nodo**)**  “Se inserta un nuevo en el árbol binario de búsqueda”  {*pre:* ABB!=nulo}  {*post: El ABB tiene un nuevo nodo en su árbol*} |

|  |
| --- |
| **Sucesor(**Llave**)**  “Se da el nodo sucesor del nodo al que le pertenece la Llave”  {*pre:* ABB!=nulo}  {*post: Si la Llave ∈ algún nodo en el árbol y este nodo != hoja -> Se da el nodo mínimo del hijo derecho de este nodo (sucesor),*  *Si la llave ∉ algún nodo en el árbol -> Se devuelve un valor nulo}* |

|  |
| --- |
| **Predecesor(**Llave**)**  “Se da el nodo predecesor del nodo al que pertenece la Llave”  {*pre:* ABB!=nulo}  {*post: Si la Llave ∈ algún nodo en el árbol y este nodo != hoja -> Se da el nodo máximo del hijo izquierdo de este nodo (predecesor),*  *Si la llave ∉ algún nodo en el árbol -> Se devuelve un valor nulo}* |

|  |
| --- |
| **EsHoja(**Llave**)**  “Da a conocer si el nodo al que le pertenece la llave es hoja”  {*pre:* ABB!=nulo}  {*post: Si la Llave ∈ algún nodo en el árbol y los hijos de este son nulos ->* True,  *Si la Llave ∈ algún nodo en el árbol y algún hijo de este no es nulo -> False*} |

|  |
| --- |
| **ArbolMinimo()**  “Da el nodo con el valor menor en el ABB”  {*pre:* ABB!=nulo}  {*post: Si la raíz != nulo -> Se da el nodo con el valor mínimo en el ABB,*  *De lo contrario -> Se devuelve un valor nulo*} |

|  |
| --- |
| **ArbolMaximo()**  “Da el nodo con el valor máximo en el ABB”  {*pre:* ABB!=nulo}  {*pre: Si la raíz != nulo -> Se da el nodo con el valor máximo en el ABB,*  *De lo contrario -> Se devuelve un valor nulo*} |

|  |
| --- |
| **TAD** **ARN** |
| Imagen que contiene colgante  Descripción generada con confianza alta |
| {*inv.: Se deben respetar las siguientes propiedades. (1) Cada nodo del árbol es rojo o negro. (2) Cada hoja o el siguiente apuntador -de sus últimos hijos- va a ser nulo. (3) Si un nodo es rojo entonces sus dos hijos son negros. (4) Todos los caminos desde el nodo en que este ubicado actualmente va a tener la misma cantidad de nodos negro -altura negra- (5) El padre es menor o igual que su hijo derecho y mayor que su hijo izquierdo.*} |
| Operaciones Primitivas:   * CrearARN: ->ARN * InsertarEnARN: *ARNxLlavexValor* ->ARN * ArreglarInsertar: *ARNxNodo* ->ARN * EliminarEnARN: *ARNxLlave* ->ARN * ArreglarEliminar: *ARNxNodo* ->ARN * RotarIzquierda: *ARNxNodo* ->ARN * RotarDerecha: *ARNxNodo* ->ARN |

|  |
| --- |
| **CrearARN()**  “Crea un árbol binario de búsqueda rojo y negro con raíz nula”  {*pre:* TRUE}  {*post: Se crea un árbol binario de búsqueda rojo y negro vacío*} |

|  |
| --- |
| **InsertarEnARN(**Llave, Valor)  “Se inserta un nuevo nodo en el ARN. Este nodo tendrá Llave y Valor”  {*pre:* ARN!=nulo}  {*post: Se agrega un nuevo nodo al árbol rojo y negro*} |

|  |
| --- |
| **ArreglarInsertar(**Nodo**)**  “Se evalúan y confirman las propiedades del árbol ARN. Después, se aplican las rotaciones necesarias para balancear el árbol siempre y cuando sea necesario. Esto se hace cada vez que se insertar un nuevo nodo”  {*pre:* ARN!=nulo}  {*post: Si una de las propiedades no se cumple -> Se aplica el balanceo apropiado al ARN*  *De lo contrario -> No se hace ninguna modificación*} |

|  |
| --- |
| **EliminarEnARN(**Llave**)**  “Eliminar un nodo del árbol ARN de acuerdo con la Llave dada”  {*pre:* ARN!=nulo}  {*post: Si la Llave ∈ a algún nodo en el ARN -> Se elimina este nodo y se llevan a cabo las rotaciones necesarias,*  *De lo contrario -> No se hace ninguna modificación*} |

|  |
| --- |
| **ArreglarEliminar(**Nodo**)**  “Se evalúan y confirman las propiedades del árbol ARN. Después, se aplican las rotaciones necesarias para balancear el árbol siempre y cuando sea necesario. Esto se hace cada vez que se elimina un nodo del árbol”  {*pre:* ARN!=nulo}  {*post: Si una de las propiedades no se cumple -> Se aplica el balanceo apropiado al ARN*  *De lo contrario -> No se hace ninguna modificación*} |

|  |
| --- |
| **RotarIzquierda(**Nodo**)**  “Se rota a la izquierda el sub-árbol a partir del nodo dado, teniendo en cuenta las condiciones necesarias del ARN”  {*pre:* ARN!=nulo}  {*post: Se rota el sub-árbol a la izquierda teniendo como punto de partida el nodo dado*} |

|  |
| --- |
| **RotarDerecha(**Nodo**)**  “Se rota a la derecha el sub-árbol a partir del nodo dado, teniendo en cuenta las condiciones necesarias del ARN”  {*pre:* ARN!=nulo}  {*post: Se rota el sub-árbol a la derecha a teniendo como punto de partida el nodo dado*} |

|  |
| --- |
| **TAD** **AVL** |
|  |
| {*inv.: Se deben cumplir las siguientes propiedades: (1)El factor de balanceo en un árbol AVL no debe diferir de 1. (2) El factor de balanceo debe estar entre -1 y 1, teniendo en cuenta el valor 0*} |
| Operaciones Primitivas:   * CrearAVL: ->AVL * InsertarEnAVL: *AVLxLlavexValor* ->AVL * BalancearInsertar: *AVLxNodo* ->AVL * EliminarEnAVL: *AVLxLlave* ->AVL * BalancearEliminar: *AVLxLlave* ->AVL * RotarIzquierda: *AVLxNodo* ->AVL * RotarDerecha: *AVLxNodo* ->AVL |

|  |
| --- |
| **CrearAVL()**  “Crea un árbol AVL con raíz nula”  {*pre:* TRUE}  {*post: Se crea un árbol AVL vacío*} |

|  |
| --- |
| **InsertarEnAVL(**Llave, Valor**)**  “Se insertar un nuevo nodo al árbol AVL. Este nodo va a tendrá Llave y Valor”  {*pre:* AVL!=nulo}  {*post: Se agrega un nuevo nuevo nodo al árbol AVL*} |

|  |
| --- |
| **BalancearInsertar(**Nodo**)**  “Se corrige el factor de balanceo de cada nodo del sub-árbol AVL de acuerdo con el nodo dado, llevando a cabo la rotaciones necesarias siempre y cuando sea necesario. Esto se hace cada vez que se agrega un nuevo nodo”  {*pre:* AVL!=nulo}  {*post: Si algún factor del sub-árbol se excede de 1 -> Se aplica el balanceo apropiado al AVL*  *De lo contrario -> No se hace ninguna modificación*} |

|  |
| --- |
| **EliminarEnAVL(**Llave**)**  “Elimina un nodo del árbol AVL de acuerdo con la llave dada”  {*pre:* AVL!=nulo}  {*post: Si la Llave ∈ a algún nodo en el AVL -> Se elimina este nodo y se llevan a cabo las rotaciones necesarias,*  *De lo contario -> No se hace ninguna modificación*} |

|  |
| --- |
| **BalancearEliminar(**Nodo**)**  “Se corrige el factor de balanceo de cada nodo del sub-árbol AVL de acuerdo con el nodo dado, llevando a cabo la rotaciones necesarias siempre y cuando sea necesario. Esto se hace cada vez que se elimina un nodo”  {*pre:* AVL!=nulo}  {*post: Si algún factor del sub-árbol se excede de 1 -> Se aplica el balanceo apropiado al AVL*  *De lo contrario -> No se hace ninguna modificación*} |

|  |
| --- |
| **RotarIzquierda(**Nodo**)**  “Se rota a la izquierda el sub-árbol a partir del nodo dado, teniendo en cuenta las condiciones necesarias del AVL”  {*pre:* AVL!=nulo}  {*post: Se rota el sub-árbol a la izquierda teniendo como punto de partida el nodo dado*} |

|  |
| --- |
| **RotarDerecha(**Nodo**)**  “Se rota a la derecha el sub-árbol a partir del nodo dado, teniendo en cuenta las condiciones necesarias del AVL”  {*pre:* AVL!=nulo}  {*post: Se rota el sub-árbol a la derecha a teniendo como punto de partida el nodo dado*} |

**Diagrama de clases**

****