**Método de la Ingeniería**

**Fase 1: Identificación del problema**

El baloncesto es uno de los deportes de mayor relevancia y popularidad a nivel mundial. Ha medida que el tiempo ha transcurrido el deporte ha evolucionado y el volumen de análisis de datos dentro del juego también. En la actualidad se llevan estadísticas en múltiples categorías y aspectos del deporte.

Ante la reciente avalancha de números en donde cualquier cifra proveniente del juego es apta de dar un dato mínimamente útil, la FIBA que es la organización reguladora del basquetbol a nivel mundial ha decidido consolidar los datos de mayor relevancia de cada uno de los profesionales de baloncesto del planeta, de manera que permita efectuar diferentes consultas que permita realizar análisis sobre estos datos.

* **Definición del problema:**

Se ha requerido el desarrollo de un programa que agrupe la información de mayor relevancia como: puntos, rebotes, asistencias, bloqueos, robos, porcentaje de éxito en cada tiro, etc. De esta forma permitir el desarrollo del análisis de los datos con el objetivo de saber la tendencia y los patrones en el deporte, los criterios que tienen más fuerza y en si hacia donde se dirige.

* **REQUERIMIENTOS FUNCIONALES**

|  |  |
| --- | --- |
| Nombre | R1: Manejar la información |
| Resumen | Se requiere manejar la información que es de gran tamaño de cada uno de los jugadores. |
| Entrada | Información de cada uno de los jugadores. |
| Salida | Manejo de toda la información. |

|  |  |
| --- | --- |
| Nombre | R2: Ingresar datos de los jugadores |
| Resumen | Se ingresan los datos de los jugadores de manera masiva (archivos csv. Por ejemplo) o a través de una interfaz. |
| Entrada | Datos de los jugadores para ingresar. |
| Salida | Datos ingresados exitosamente. |

|  |  |
| --- | --- |
| Nombre | R3: Eliminar datos de los jugadores. |
| Resumen | Se eliminan los datos que se requieran o que se prefieran de los jugadores. |
| Entrada | Datos de los jugadores a eliminar. |
| Salida | Datos eliminados exitosamente. |

|  |  |
| --- | --- |
| Nombre | R4: Modificar datos de los jugadores. |
| Resumen | Se debe permitir la modificación de los datos de los jugadores para ingresar información nueva o simplemente cambiarla. |
| Entrada | Datos de los jugadores a ser modificados. |
| Salida | Datos modificados exitosamente. |

|  |  |
| --- | --- |
| Nombre | R5: Realizar consultas de jugadores |
| Resumen | Se debe permitir el consultar los jugadores utilizando como criterio de búsqueda las categorías estadísticas incluidas. |
| Entrada | Categorías estadísticas incluidas para la realización de la búsqueda. |
| Salida | Consulta de jugadores exitosa. |

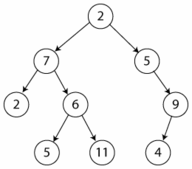
|  |  |
| --- | --- |
| Nombre | R6: Recuperar jugadores de acuerdo a la categoría |
| Resumen | El programa debe estar en la capacidad de recuperar jugadores de acuerdo a la categoría de búsqueda seleccionada y valor dado para ella. |
| Entrada | Jugador a recuperar de acuerdo a la categoría. |
| Salida | Jugador recuperado exitosamente. |

**Fase 2: Recopilación de la información**

Definiciones

* **Arboles binarios**

Un árbol T es un grafo simple que cumple que entre los vértices v y w existe un único camino simple.



Para ir de un vértice cualquiera a otro simple se tiene un camino simple.

Conceptos básicos de árbol:

* Vértice: También llamado nodo y corresponde a cada uno de los elementos del árbol.
* Raíz: Es el vértice principal del árbol.
* Padre: Es el vértice que puede o no tener subárboles
* Hijo: Es el vértice descendiente de una raíz o un padre.
* Hermanos: Son los vértices descendientes de un mismo padre.
* Hoja: Es un vértice que no tiene uno o más hijos. También llamado nodo terminal.

Un **Árbol Binario** es un tipo de árbol en que cada vértice máximo puede tener dos hijos; su nodo raíz está enlazado a dos subárboles binarios disjuntos denominados subárbol izquierdo y subárbol derecho. Los árboles binarios no son vacíos ya que como mínimo tienen el nodo raíz.

* **Representación de árboles binarios:**

Los árboles binarios pueden representarse en un vector o en una lista ligada. Nuestro interés se centrará en los vectores.

Para representar a un árbol binario en un vector se escriben por niveles los nodos del árbol de manera ordenada, de izquierda a derecha (hijo izquierdo — hijo derecho).

Esta representación es poco eficiente cuando el árbol no es completo, en vista del gran desperdicio de memoria que podría haber por las posiciones libres que quedarían en el vector.

Si k es la posición en un vector de un nodo de un árbol con n niveles, se tiene:

El vector tendrá 2n+1 -1 posiciones.

El padre estará en la posición k/2 (con k>1)

El hijo izquierdo estará en la posición 2\*I (con 2\*k<n) y el hijo derecho en la posición 2\*k+1.

* **Recorridos en un árbol binario**

Un recorrido en un árbol binario es Una operación que consiste en visitar todos sus vértices o nodos, de tal manera que cada vértice se visite una sola vez.

Se distinguen tres tipos de recorrido: INORDEN, POSORDEN Y PREORDEN.

En cada recorrido se tiene en cuenta la posición de la raíz (de ahí su nombre) y que siempre se debe ejecutar primero el hijo izquierdo y luego el derecho.

**Recorrido INORDEN.** Este recorrido se realiza así: primero recorre el subárbol izquierdo, segundo visita la raíz y por último, va al subárbol derecho. En síntesis:

hijo izquierdo — raíz — hijo derecho

**Recorrido PREORDEN**. Este recorrido se realiza así: primero visita la raíz; segundo recorre el subárbol izquierdo y por último va a subárbol derecho. En síntesis:

raíz — hijo izquierdo — hijo derecho

**Recorrido POSORDEN**. Primero recorre el subárbol izquierdo; segundo, recorre el subárbol derecho y por último, visita la raíz. En síntesis:

hijo izquierdo– hijo derecho — raíz

* **Arboles binarios balanceados**

Cuando hablamos de árboles binarios balanceados nos referimos a arboles binarios con la característica de que siempre están balanceados. Y se dice que un árbol esta balanceado cuando para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad a la altura de la rama derecha o viceversa.

* Factor de equilibrio:

La condición es la comparación entre la altura del subárbol derecho y el izquierdo.

Si es 0, el nodo está perfectamente equilibrado, y sus subárboles derecho e izquierdo tienen la misma altura.

Si es 1, el nodo está equilibrado y su subárbol derecho es un nivel más alto.

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

Si es >=2 es necesario reequilibrar el árbol.

En el caso de que un árbol este desbalanceado, es necesario equilibrarlo, es decir lograr un factor de equilibrio que cumpla con las condiciones previamente establecidas. Para ello se llevan a cabo **rotaciones.**

Existen 4 tipos de rotaciones:

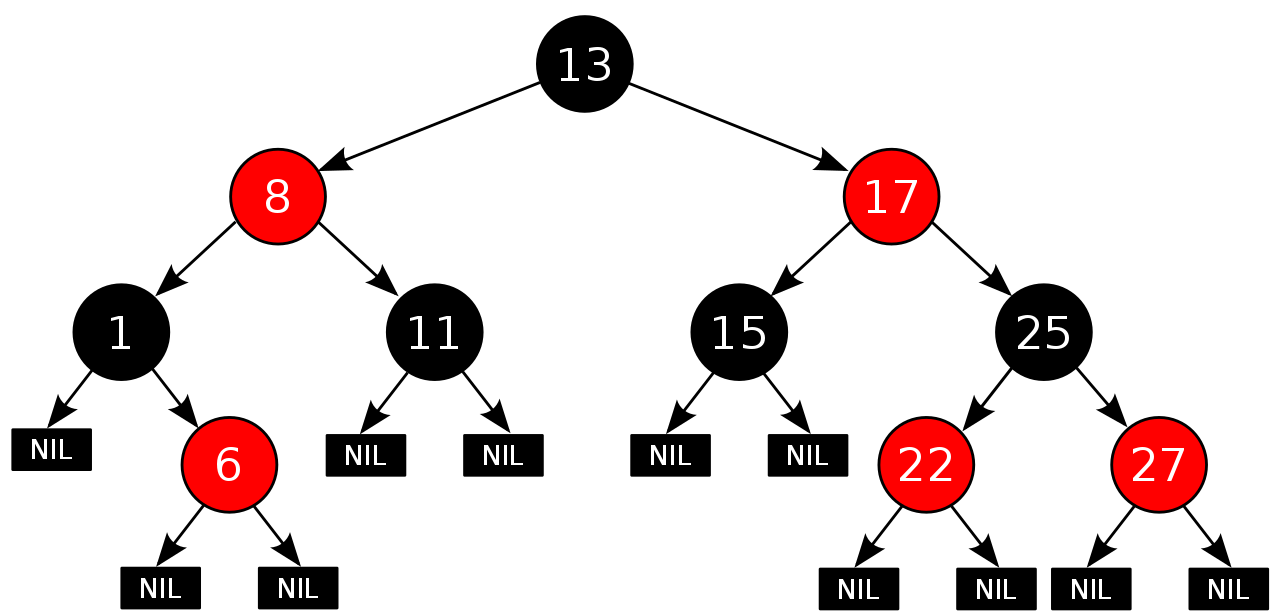
* ROTACIÓN SIMPLE A LA DERECHA: Se usará cuando el subárbol izquierdo de un nodo sea 2 unidades más alto que el derecho.
* ROTACIÓN SIMPLE A LA IZQUIERDA: Se usará cuando el subárbol derecho de un nodo sea 2 unidades más alto que el izquierdo.
* ROTACIÓN DOBLE A LA DERECHA: Si esta desequilibrado su FE>1 y su hijo derecho tiene distinto signo(-).
* ROTACIÓN DOBLE A LA IZQUIERDA: Si esta desequilibrado a la derecha su FE<-1 y su hijo izquierdo tiene distinto signo(+).
* **Arboles Rojo-Negro**

Un árbol rojo-negro es un tipo abstracto de datos. Concretamente, es un árbol binario de búsqueda equilibrado, una estructura de datos utilizada en informática y ciencias de la computación. La estructura original fue creada por Rudolf Bayer en 1972, que le dio el nombre de “árboles-B binarios simétricos”, pero tomó su nombre moderno en un trabajo de Leo J. Guibas y Robert Sedgewick realizado en 1978. Es complejo, pero tiene un buen peor caso de tiempo de ejecución para sus operaciones y es eficiente en la práctica. Puede buscar, insertar y borrar en un tiempo O(log n), donde n es el número de elementos del árbol.

* **Propiedades:**

Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer las siguientes reglas para tener un árbol rojo-negro válido:

* 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.



Estas reglas producen una regla crucial para los árboles rojo-negro: el camino más largo desde la raíz hasta una hoja no es más largo que dos veces el camino más corto desde la raíz a una hoja. El resultado es que dicho árbol está aproximadamente equilibrado.

**Fase 3: Búsqueda de soluciones creativas.**

* Utilizar arboles de manera que nos ayude en el recorrido de cada uno de los basquetbolistas y así mismo poder obtener la información requerida.
* Investigar sobre los tipos de árboles binarios y analizar cuál de ellos nos puede servir en el desarrollo del problema.
* Utilizar un árbol rojo y negro
* Utilizar arboles NArios para la solución del problema
* Utilizar una ArrayList para el almacenamiento de la información
* Utilizar un árbol binario de búsqueda
* Utilizar los arreglos simples de siempre para resolver el problema.
* Utilizar grafos para el desarrollo del problema

**Fase 4: Transición de la formulación de ideas a los diseños preliminares**

Se descartaron las siguientes ideas:

|  |  |
| --- | --- |
| Idea no factible | Justificación |
| Utilizar una ArrayList | Porque el uso de el ArrayList para este problema donde se tienen que almacenar muchos jugadores y en donde cada uno contiene una información, y además esta se va a comparar es muy ineficiente. |
| Utilizar arboles NArios | Cómo no hay una jerarquía en el problema para su desarrollo hemos decidido descartar esta opción. |
| Utilizar Grafos | Porque el uso de grafos recién lo estamos aprendiendo, y el uso que le hemos dado es muy básico como para emplearlo en el desarrollo del programa. |
| Utilizar arboles binarios | Utilizaremos estos como bases a través de árboles rojo y negros y arboles balanceados, pero su uso único como arboles binarios no lo utilizaremos. |
| Utilizar arreglos | El uso de este tipo de arreglos es obsoleto para la solución en este laboratorio ya que no sirve de mucho guardar la información porque es muy grande. |

* **Diseños preliminares**

**TAD: ABB**

|  |
| --- |
| TAD : ABB |
| Representación: |
| Invariantes   * <Datn < Datn+1> * <Cuando se cree un nuevo nodo izquierdo y derecho = NIL> * <Dat1 va ser la raíz> * <Si un nodo no tiene nodo izquierdo o derecho se llamará hijo> |
| Operaciones   * <ContruirArbol>: Árbol * <AgregarNodo>: Árbol, Nodo, Dato Árbol * <BuscarDato>: Árbol, Dato Árbol * <EliminarNodo>: Árbol, Nodo Árbol * <ModificarDato>: Árbol, Nodo, Dato Árbol * <Vacio>: Árbol Árbol |

**TAD Árbol Binario Rojinegro**

|  |
| --- |
| TAD : ABB Rojinegro. |
| Representación: |
| Invariantes   * <Datn < Datn+1 por la rama derecha y Datn > Datn+1 por la rama izquierda> * <Cuando se cree un nuevo nodo izquierdo y derecho = NIL> * <Dat1 va ser la raíz> * <Si un nodo no tiene nodo izquierdo o derecho se llamará hijo> * <Cada nodo es rojo o negro> * <Toda hoja (NIL) es negra> * <Si un nodo es rojo, sus dos hijos son negros> * <Todo camino desde un nodo a cualquier hoja descendente contiene el mismo número de nodos negros> |
| Operaciones   * <ContruirArbol>: Árbol * <AgregarNodo>: Árbol, llave, Valor Árbol * <Buscarllave>: Árbol, llave Árbol * <EliminarNodo>: Árbol, llave Árbol * <ModificarDato>: Árbol,llaveAn,llave, Valor Árbol * <Vacio>: Árbol Árbol |

|  |
| --- |
| ConstruirArbol() |
| “Crea un nuevo árbol rojiNegro sin ningún nodo ”  {Pre: True}  {Post: BST\_RB = {raiz, raiz = dato} } |

|  |
| --- |
| AgregarNodo() |
| “Agrega un nuevo nodo al árbol, teniendo en cuenta el criterio de ordenamiento”  {Pre: BST\_RB = {raiz, raiz = dato or nill}, llave; llave = dato, valor; valor = dato}  {Post: BST\_RB = Árbol con un nodo nuevo agregado} |

|  |
| --- |
| BuscarLlave() |
| “Busca la llave que se especifica la retorna”  {Pre: BST\_RB = {raiz, raiz = dato or nill}, llave; llave = dato}  {Post: Llave} |
|  |

|  |
| --- |
| EliminarNodo() |
| “Elimina el nodo que se requiere de acuerdo a la llave especificada”  {Pre: BST\_RB = {raiz, raiz = dato or nill}, llave; llave = dato}  {Post: BST\_RB; Sin el nodo con la llave} |

|  |
| --- |
| ModificarNodo() |
| “Modifica el nodo con la llave anterior, y lo modifica con la nueva llave o el nuevo valor”  {Pre: BST\_RB = {raiz, raiz = dato or nill}, llaveAn; llave = dato,ll,ve; llave = dato, valor; valor = dato}  {Post: BST\_RB con el nodo modificado} |

|  |
| --- |
| Vacio() |
| “Retorna un valor de verdad si el árbol está vacío o tiene algún elemento”  {Pre: BST\_RB = {raiz, raiz = dato or nill}}  {Post: Si el árbol está vacío o esta con al menos un elemento retorna verdadero o falso respectivamente } |

**TAD Árbol AVL**

|  |
| --- |
| TAD : Árbol AVL |
| Representación:  fe= -1  fe= 0 fe=0    fe=0 fe=0  fe=0 fe=0  fe=0 fe=0 |
| Invariantes   * <Datn < Datn+1 por la rama derecha y Datn > Datn+1 por la rama izquierda> * <Dat1 va ser la raíz> * <Se tiene un factor de balanceo> * <Para cada nodo del árbol, las alturas de sus dos hijos difieren por mucho en 1> |
| Operaciones   * <ContruirArbol>: Árbol * <AgregarNodo>: Árbol, llave, Valor Árbol * <Buscarllave>: Árbol, llave Árbol * <EliminarNodo>: Árbol, llave Árbol * <ModificarDato>: Árbol,llaveAn,llave, Valor Árbol * <Vacio>: Árbol Árbol |

|  |
| --- |
| ConstruirArbol() |
| “Crea un nuevo árbol AVL sin ningún nodo ”  {Pre: True}  {Post: BST\_AVL = {raiz, raiz = dato} } |

|  |
| --- |
| AgregarNodo() |
| “Agrega un nuevo nodo al árbol, teniendo en cuenta el criterio de ordenamiento”  {Pre: BST\_AVL= {raiz, raiz = dato or nill}, llave; llave = dato, valor; valor = dato}  {Post: BST\_AVL = Árbol con un nodo nuevo agregado} |

|  |
| --- |
| BuscarLlave() |
| “Busca la llave que se especifica la retorna”  {Pre: BST\_RB = {raiz, raiz = dato or nill}, llave; llave = dato}  {Post: Llave} |

|  |
| --- |
| EliminarNodo() |
| “Elimina el nodo que se requiere deacuerdo a la llave especificada”  {Pre: BST\_AVL = {raiz, raiz = dato or nill}, llave; llave = dato}  {Post: BST\_AVL; Sin el nodo con la llave} |

|  |
| --- |
| ModificarNodo() |
| “Modifica el nodo con la llave anterior, y lo modifica con la nueva llave o el nuevo valor”  {Pre: BST\_AVL = {raiz, raiz = dato or nill}, llaveAn; llave = dato,ll,ve; llave = dato, valor; valor = dato}  {Post: BST\_AVL con el nodo modificado} |

|  |
| --- |
| Vacio() |
| “Retorna un valor de verdad si el árbol está vacío o tiene algún elemento”  {Pre: BST\_AVL = {raiz, raiz = dato or nill}}  {Post: Si el árbol está vacío o esta con al menos un elemento retorna verdadero o falso respectivamente } |

**Pseudocódigos**

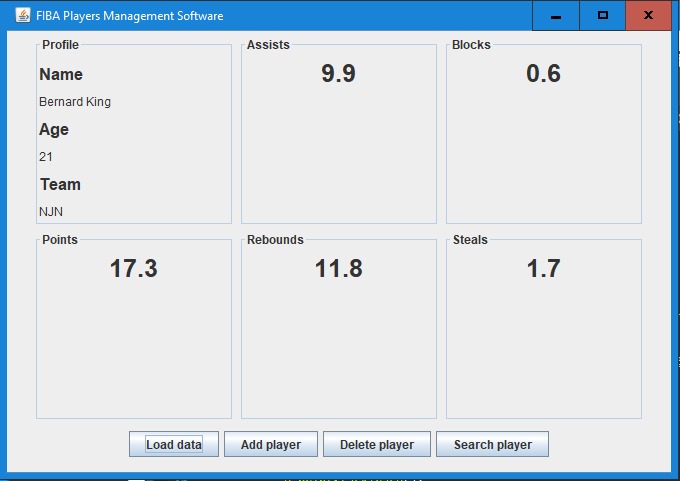
|  |  |  |  |
| --- | --- | --- | --- |
| Rotaciones Árbol Rojo  y Negro    Left-Rotate (T, x)  y = x.right  x.right = y.left  if y.left ≠ T.nil  y.left.p = x  y.p = x.p  if x.p == T.nil  T.root = y  elseif x==x.p.left  x.p.left = y  else x.p.right = y  y.left = y  x.p = y | Insertion  R-B-INSERT(T,z)  y=T.nil  x=T.root  while x ≠T.nil  y=x  if z.key<x.key  x = x.left  elseif x = x.right  z.p = y  if y==T.nil  T.root = z  elseif z.key < y.key  y.left = z  else y.right = z  z.left = T.nil  z.right = T.nil  z.color = RED  RB-INSERT FIXUP(T,z) | RB-INSERT-FIXUP(T,z)  while z.p.color == RED  if z.p = = z.p.p.left  y = z.p.p.right  if y.color == RED  z.p.color = BLACK // case 1  y.color = BLACK // case 1  z.p.p.color = RED // case 1  z = z.p.p // case 1  else if z == z.p.right  z = z.p // case 2  LEFT-ROTATE(T,z) // case 2  z.p.color = BLACK // case 3  zp.p.color = RED // case 3  RIGHT-ROTATE(T, z.p.p) // case 3  else (same as then clause  with “right” and “left” exchanged)  T.root.color = BLACK | RB-TRANSPLANT(T,u,v)  if u.p == T.nil  T.root = v  elseif u == u.p.left  u.p.left = v else u.p.right = v  v.p = u.p |

|  |  |
| --- | --- |
| RB-DELETE-FIXUP(T,x)  while x ≠ T.root and x.color == BLACK  if x == x.p.left  w = x.p.right if w.color = = RED  w.color = BLACK // case 1  x.p.color = RED // case 1  LEFT-ROTATE(T, x.p) // case 1  w = x.p.right // case 1 if w.left.color == BLACK and w.right.color = = BLACK  w.color = RED // case 2  x = x.p // case 2  else if w.right.color == BLACK  w.left.color = BLACK // case 3  w.color = RED // case 3  RIGHT-ROTATE(T,w) // case 3  w = x.p.right // case 3  w.color = x.p.color // case 4  x.p.color = BLACK // case 4  w.right.color = BLACK // case 4  LEFT-ROTATE(T, x.p) // case 4  x = T.root // case 4  else (same as then clause with “right” and “left” exchanged)  x.color = BLACK | RB-DELETE(T, z)  y = z y-original-color = y.color if z.left == T.nil  x = z.right  RB-TRANSPLANT(T,z,z.right) elseif z.right == T:nil  x = z.left  RB-TRANSPLANT(T,z,z.left)  else y = TREE-MINIMUM(z.right)  y-original-color = y.color  x = y.right  if y.p == z  x.p = y else RB-TRANSPLANT(T, y, y.right)  y.right = z.right  y.right.p = y RB-TRANSPLANT(T,z, y)  y.left = z.left  y.left.p = y  y.color = z.color if y-original-color == BLACK RB-DELETE-FIXUP(T, x) |

**Complejidad**

* Como la altura del árbol rojo y negro es O(log n) y hay O(log n) iteraciones entonces en el caso general de la inserción en ARN tiene complejidad O(log n).
* La complejidad temporal para cada uno de los métodos utilizados en la implementación de arboles AVL es O(log n) como lo son: insertar, eliminar, etc.

**Visualización de la interfaz**

****

**Fase 5: Evaluación y Selección de la mejor Solución**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 4 | 3 | 2 | 1 | Total |
| Complejidad temporal de la implementación de cada método | El 100% de sus  métodos  presenta  complejidad  constante de  O(1) | El 80% o  menos de sus  métodos  presenta  complejidad de  O(log n) y el  resto igual a  O(1) | El 50% o  menos de sus  métodos  presenta  complejidad de  O(n) y el resto  ≤O(log n) | El 50% o menos  de sus métodos  presentan  complejidad de  O(n log n) y el  resto ≤ O(n) | Alguno de sus  métodos  presenta  complejidad ≥  O(n2) |  |
|  | Complejidad espacial de la implementación del método. | Restricciones de capacidad de la estructura de datos utilizada. | Facilidad de  la  implementación | Nivel de  aprendizaje  en la  implementación | Complejidad temporal de la implementación de cada método |  |
|  | 15% | 25% | 15% | 30% | 15% | 100% |
| Árbol balanceado | 4 | 5 | 4 | 5 | 4 | 4.55 |
| Árbol Rojo y Negro | 4 | 5 | 4 | 4 | 4 | 4.25 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Clase | Método | Escenario | Valores de entrada | Salida |
| BST\_RBT | addNode() | setup1() = Àrbol creado sin ningún nodo | key = 2; value = 6 | Se agrega el nodo con llave 2 y valor 6 al árbol y se pinta de color negro ya que es la raiz |
| BST\_RBT | addNode() | setup1() = Àrbol creado con 3 nodos la raíz con valor 2, y un hijo derecho con llave 6 y una hijo izquierdo con llave 1 | key = 3; value = 4 | Se agrega el nodo con llave 3 y valor 6 al árbol y se cumplen las propiedades del arbol rojiNegro |
| BST\_RBT | addNode() | setup1() = Àrbol creado con más de 10 nodos | key = 50, value = 9 | Se agrega el nodo con llave 50 y valor 9 al árbol y se cumplen las propiedades del arbol rojiNegro |
| BST\_RBT | deleteNode() | setup1() = Àrbol creado con un solo nodo | key = 1 | Se elimina el único nodo y el arbol queda vacio |
| BST\_RBT | deleteNode() | setup1() = Àrbol creado con 3 nodos la raíz con valor 2, y un hijo derecho con llave 6 y una hijo izquierdo con llave 1 | key = 6 | Se elimina el nodo con llave 6 y el arbol sigue cumpliendo las propiedades de un rojinegro |
| BST\_RBT | deleteNode() | setup1() = Àrbol creado con más de 10 nodos | key = 64 | Se elimina el nodo con llave 64 y el arbol sigue cumpliendo las propiedades de un rojinegro |
| BST\_AVL | addNode() | setup2() = Àrbol AVL creado sin ningún nodo | key = 1; value = 1 | Se agrega el nodo con llave 1 y valor 1 al arbol AVL y queda como la raiz |
| BST\_AVL | addNode() | setup2() = Àrbol AVL creado con 3 nodos | key = 50; value = 4 | Se agrega el nodo con llave 50 y valor 4 al arbol AVL y se cumple con el factor de balanceo |
| BST\_AVL | addNode() | setup2() = Àrbol AVL creado con más de 10 nodos | key = 60; value = 5 | Se agregar el nodo con llave 60 y valor 5 al arbol AVL y se cumple con el factor de balanceo |
| BST\_AVL | deleteNode() | setup2() = Àrbol AVL creado con un solo nodo | key = 10 | Se elimina el nodo con llave 10 y el arbol queda vacio |
| BST\_AVL | deleteNode() | setup2() = Àrbol AVL creado con tres nodos | key = 20 | Se elimina el nodo con llave 20 y se siguen cumpliendo las propiedades del arbol AVL |
| BST\_AVL | deleteNode() | setup2() = Àrbol AVL creado con más de 10 nodos | key = 21 | Se elimina el nodo con llave 21 y se siguen cumpliendo las propiedades del arbol AVL |
|  |  | Como el método Buscar es igual en ambos casos se harán solo 3 pruebas para este método |  |  |
| BST | searchNode() | setup3() = Àrbol creado con un solo nodo | key = 2 | Retorna el nodo con valor 2 que encontró en el arbol |
| BST | searchNode() | setup3() = Àrbol creado con 10 nodos | key = 100 | Retorna el nodo con valor 100 que encontró en el arbol |
| BST | searchNode() | setup3() = Àrbol creado con un solo nodo | key = 80 | Retorna el nodo con el valor 80 que encontró en el arbol |

* **FASE 7: IMPLEMENTACIÓN**

La implementación de la solución se encuentra en el siguiente repositorio de github:

Nota: Tuvimos un inconveniento con el antiguo repositorio usado para la solucion de este programa el cual es este: <https://github.com/nicolass03/fibaLab_AEDII>

Pero el proyecto terminado se encuentra en: <https://github.com/nicolass03/fibaLab_AED>

* **Bibliografía**

<https://medium.com/@matematicasdiscretaslibro/cap%C3%ADtulo-12-teoria-de-arboles-binarios-f731baf470c0>

<https://es.wikipedia.org/wiki/%C3%81rbol_rojo-negro>

**[1] Cormen, T., Leiserson, C., Rivest, R., & Stein, C. Introduction to algorithms**