Martin Cifuentes, Gabriel Delgado y Felipe Barreto

Documentación de la tarea integradora 2

Contenido

[Requerimientos 1](#_Toc86616506)

[Requerimientos no funcionales 1](#_Toc86616507)

[Diagrama de clases 2](#_Toc86616508)

[Pruebas unitarias 2](#_Toc86616509)

[Diagrama de pruebas unitarias 2](#_Toc86616510)

[TAD 2](#_Toc86616511)

[TAD de árbol binario 2](#_Toc86616512)

[TAD de Nodo 5](#_Toc86616513)

[TAD de árbol rojo-negro 8](#_Toc86616514)

[JUNIT 13](#_Toc86616515)

[Setup Scenarios del modelo 13](#_Toc86616516)

[Diseño de casos de prueba del modelo 13](#_Toc86616517)

[Diagrama de las pruebas del modelo 17](#_Toc86616518)

[Setup Scenarios de las estructuras de datos 18](#_Toc86616519)

[Diseño de casos de prueba del modelo 19](#_Toc86616520)

Documentación

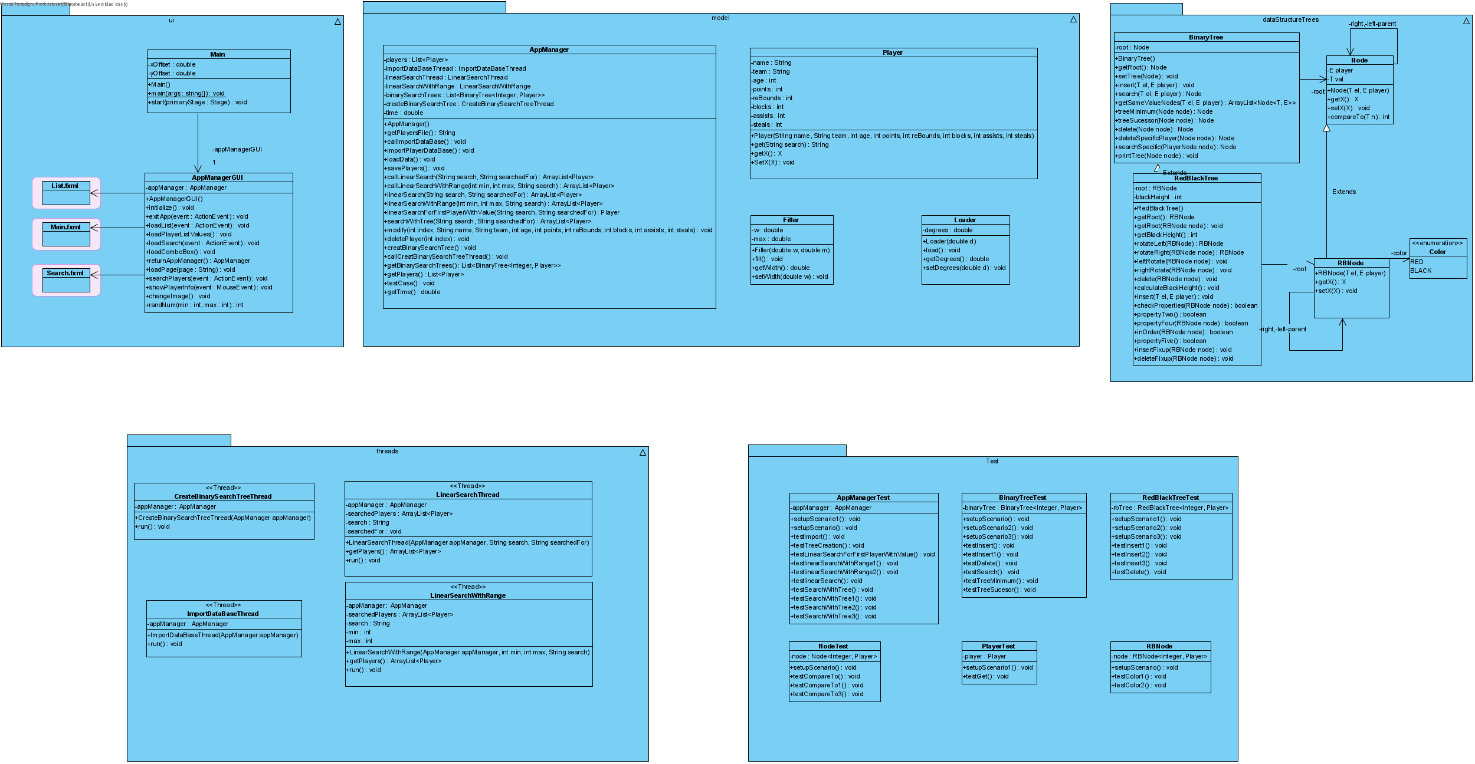
# Requerimientos

* El programa debe poder manejar por lo menos 200,000 jugadores
* El programa debe ser capaz de buscar el dato que el usuario quiera
* El programa debe ser capaz de leer un archivo csv
* El programa debe poder añadir o eliminar jugadores desde la interfaz grafica
* El programa debe poder modificar las estadísticas de los jugadores
* Al menos 4 estadísticas deben tener una búsqueda de complejidad de tiempo de O (log n)
* Todo jugador debe tener nombre, edad, equipo y 5 estadísticas

# Requerimientos no funcionales

* La interfaz gráfica debe ser fácil de entender
* El programa debe estar organizado
* El uso de un árbol rojo, negro
* El uso serialización de los árboles binarios

# Diagrama de clases



# TAD

## TAD de árbol binario

|  |
| --- |
| **TAD** BinaryTree |
| BinaryTree= {root=Node<T,E> } |
| { inv: root.getVal<=root.getRight.getVal && root.getVal>root.getLeft.getVal  } |
| Operaciones primitivas:  BinaryTree  ·         getRoot    null -> Node<T,E>  ·         setRoot    Node<T,E> -> null  ·         insert T,E -> null  ·         search T,E -> Node<T,E>  ·         getSameValueNode T,E -> ArrayList<Node<T,E>>  treeSuccessor    Node<T,E> -> Node<T,E>  delete   T,E -> Node<T,E>  deleteSpecificPlayer    T,E -> Node<T,E>  printTree    Node<T,E> -> void |

|  |
| --- |
| BinaryTree()  “Constructor de la clase, crea un objeto con un Node<T,E> vacio”  { pre: un valor para T y E }    { post:  BinaryTree = {Node<T,E>:null }} |

|  |
| --- |
| getRoot()  “Devuelve el valor del root, es decir, el Node”  { pre: la existencia de root}    { post:  retorna el root del arbol} |

|  |
| --- |
| setRoot(Node<T,E> root)  “Le da un nuevo valor a root”  { pre: la existencia de root}    { post:  cambia el valor del atributo root} |

|  |
| --- |
| search (T el, E Player)  “retorna el primer node que tenga el valor: el”  { pre: La existencia del root }  { post:  retorna el primer jugador con el valor el, sino encuentra a nadie con este valor, retorna nulo} |

|  |
| --- |
| getSameValueNode(T el, E player)  “Devuelve todos los nodos que tengan el mismo valor T que el de los parametros”  { pre: la existencia de root y que este tenga un valor}  { post: Se retorna un arreglo de nodos con el mismo valor T} |

|  |
| --- |
| insert (T el, E player)  “Añade un nodo al árbol binario”  { pre: la existencia de root}  { post:  añade un nodo como hijo a otro nodo que se encuentre en el árbol, si root es nulo, este valor es el nuevo root} |

|  |
| --- |
| treeSuccessor(Node<T,E> node)  “Busca el nodo que este más a la izquierda del nodo que se entregó por parámetros”  { pre: node no puede ser nulo }  { post:  retorna el nodo más a la izquierda de node} |

|  |
| --- |
| delete(T el, E Player)  “Se borra el primer nodo que se encuentre con el valor: el”  { pre: root no puede ser nulo}  { post:  retorna el nodo que se borra} |

|  |
| --- |
| deleteSpecificPlayer (T el, E Player)  “Se borra el primer nodo que se encuentre con el valor: el y Player”  { pre: root no puede ser nulo}  { post:  retorna el nodo que se borra} |

|  |
| --- |
| printTree (Node<T,E> root)  “Imprime todos los valores de todos los nodos de root”  { pre: root no puede ser nulo}  { post: imprime por consola los valores que se encuentran en el árbol que comienza desde root } |

## TAD de Nodo

|  |
| --- |
| **TAD** Node |
| Node= {player = E, val=T, left=Node<T,E>, right=Node <T,E>,parent=Node<T,E> } |
| { inv: val>left.getVal && val<=right.getVal  } |
| Operaciones primitivas:  Node E,T ->null  ·         getVal   null -> T  ·         setVal    T -> null  ·         getLeft null-> Node<T,E>  ·         setLeft Node<T,E> -> null  ·         getRight null -> Node<T,E>  setRight   Node<T,E> -> null  getParent null-> Node<T,E>  setParent    Node<T,E> -> null  getPlayer    null -> E  setPlayer E->null  compareTo T-> int |

|  |
| --- |
| Node(T el, E player)  “Constructor de la clase, le da un valor a val y player, los otros nodos se quedan como nulos”  { pre: }    { post:  val y player ahora tienen un valor y Node es creado} |

|  |
| --- |
| getVal()  “Devuelve el valor de val, es decir, el valor que tiene este nodo”  { pre: la existencia del Node}    { post:  retorna el val del Node} |

|  |
| --- |
| setVal (T val)  “Le da un valor a val”  { pre: La existencia del Node }  { post:  le da un nuevo valor a val} |

|  |
| --- |
| getLeft ()  “retorna el hijo izquierdo de Node, que es otro Node”  {pre: la existencia de Node al cual se le hace .getLeft()}  { post: Se retorna un Node o nulo si no hay un hijo izquierdo} |

|  |
| --- |
| setLeft (Node<T,E> left)  “left se instancia como el nuevo hijo izquierdo del Node, sin importar que ya hubiera un hijo izquierdo”  { pre: la existencia de Node}  { post:  se instancia left como el left del nodo al que se le llamo este metodo} |

|  |
| --- |
| getRight()  “retorna el hijo derecho de Node, que es otro Node”  {pre: la existencia de Node al cual se le hace. getRight()}  { post: Se retorna un Node o nulo si no hay un hijo derecho} |

|  |
| --- |
| setRight (Node<T,E> right)  “right se instancia como el nuevo hijo derecho del Node, sin importar que ya hubiera un hijo derecho”  { pre: la existencia de Node}  { post:  se instancia right como el right del nodo al que se le llamo este metodo} |

|  |
| --- |
| getParent()  “retorna el padre de Node, que es otro Node”  {pre: la existencia de Node al cual se le hace. getParent ()}  { post: Se retorna un Node o nulo si no hay un padre} |

|  |
| --- |
| setParent (Node<T,E> parent)  “parent se instancia como el nuevo padre del Node, sin importar que ya hubiera un padre”  { pre: la existencia de Node}  { post:  se instancia parent como el parent del nodo al que se le llamo este metodo} |

|  |
| --- |
| getPlayer ()  “Retorna el valor de player que tiene el Node”  { pre: la existencia de Node}  { post:  retorna el atributo player de tipo E del Node} |

|  |
| --- |
| setPlayer (E player)  “Se le da un nuevo valor a el atributo player”  { pre: la existencia de Node}  { post:  se cambia el valor del atributo player} |

|  |
| --- |
| compareTo (T n)  “Compara el valor del atributo val con n, si retorna un entero negativo, val<n, si el entero es positivo val>n y si retorna 0: val==n”  { pre: la existencia de Node}  { post:  retorna un entero que indica si val es mayor, menor o igual a n} |

## TAD de árbol rojo-negro

|  |
| --- |
| **TAD** RedBlackTree |
| RedBlackTree= {root=RBNode<T,E>, blackHeight=int=0 } |
| { inv: root.getVal<=root.getRight.getVal && root.getVal>root.getLeft.getVal, blackHeight >=0 } |
| Operaciones primitivas:  RedBlackTree  ·         getRoot    null -> RBNode<T,E>  ·         setRoot    RBNode<T,E> -> null  getBlackHeight null -> int  rotateLeft RBNode<T,E> -> RBNode<T,E>  roateRight RBNode<T,E> -> RBNode<T,E>  leftRotate RBNode<T,E> -> null  rightRotate RBNode<T,E> -> null  delete RBNode<T,E> -> null  calculateBlackHeight null -> null  ·         insert T,E -> null  C checkProperties RBNode<T,E> -> boolean  propertyTwo null -> boolean  propertyFour RBNode<T,E> -> boolean  inOrder null -> boolean  propertyFive RBNode<T,E> -> Boolean  insertFixup(RBNode<T,E> -> null  insertFixup(RBNode<T,E> -> null  search T,E -> Node<T,E>  ·        getSameValueNode T,E -> ArrayList<Node<T,E>>  treeSuccessor    Node<T,E> -> Node<T,E>  deleteSpecificPlayer    T,E -> Node<T,E>  printTree    Node<T,E> -> void |

|  |
| --- |
| RedBlackTree  ()  “Constructor de la clase, crea un objeto con un RBNode<T,E> vacio”  { pre: un valor para T y E }    { post:  RedBlackTree = {RBNode<T,E>:null }} |

|  |
| --- |
| getRoot()  “Devuelve el valor del root, es decir, el RBNode”  { pre: la existencia de root}    { post:  retorna el root del árbol} |

|  |
| --- |
| setRoot(RBNode<T,E> root)  “Le da un nuevo valor a root”  { pre: la existencia de root}    { post:  cambia el valor del atributo root} |

|  |
| --- |
| getBlackHeight()  “Devuelve el valor de la altura negra.”  { pre: blackHeight debe existir}    { post:  retorna el valor de blackHeight} |

|  |
| --- |
| rotateLeft(RBNode<T,E,> node)  “Hace una rotación hacia la izquierda.”  { pre: node.getRight() ¡= null}  { post:  Rotación hacia la izquierda} |

|  |
| --- |
| rotateRight(RBNode<T,E,> node)  “Hace una rotación hacia la derecha.”  { pre: node.getRight() ¡= null}  { post:  Rotación hacia la derecha} |

|  |
| --- |
| leftRotate(RBNode<T,E,> node)  “Hace una rotación hacia la izquierda.”  { pre: node.getRight() ¡= null}  { post:  Rotación hacia la izquierda} |

|  |
| --- |
| rightRotate(RBNode<T,E,> node)  “Hace una rotación hacia la derecha.”  { pre: node.getRight() ¡= null}  { post:  Rotación hacia la derecha} |

|  |
| --- |
| delete(RBNode<T,E,> node)  “Hace una rotación hacia la derecha.”  { pre: node.getRight() ¡= null}  { post:  Rotación hacia la derecha} |

|  |
| --- |
| calculateBlackHeight()  “Calcula la altura negra.”  { pre: blackHeight debe existir}  { post:  blackHeight es asignado un valor} |

|  |
| --- |
| getSameValueNode(T el, E player)  “Devuelve todos los nodos que tengan el mismo valor T que el de los parametros”  { pre: la existencia de root y que este tenga un valor}  { post: Se retorna un arreglo de nodos con el mismo valor T} |

|  |
| --- |
| insert (T el, E player)  “Añade un nodo al árbol rojo-negro”  { pre: la existencia de root}  { post:  añade un nodo como hijo a otro nodo que se encuentre en el árbol, si root es nulo, este valor es el nuevo root} |

|  |
| --- |
| treeSuccessor(Node<T,E> node)  “Busca el nodo que este más a la izquierda del nodo que se entregó por parámetros”  { pre: node no puede ser nulo }  { post:  retorna el nodo más a la izquierda de node} |

|  |
| --- |
| deleteSpecificPlayer (T el, E Player)  “Se borra el primer nodo que se encuentre con el valor: el y Player”  { pre: root no puede ser nulo}  { post:  retorna el nodo que se borra} |

|  |
| --- |
| printTree (Node<T,E> root)  “Imprime todos los valores de todos los nodos de root”  { pre: root no puede ser nulo}  { post: imprime por consola los valores que se encuentran en el árbol que comienza desde root } |

## TAD de Nodo Rojo-Negro

|  |
| --- |
| **TAD** RBNode |
| Node= {color = Color, player = E, val=T, left=RBNode<T,E>, right=RBNode <T,E>,parent=RBNode<T,E> } |
| { inv: val>left.getVal && val<=right.getVal  } |
| Operaciones primitivas:  RBNode T,E ->null  getColor null -> Color  setColor Color -> null  ·         getVal   null -> T  ·         setVal    T -> null  ·         getLeft null-> RBNode<T,E>  ·         setLeft RBNode<T,E> -> null  ·         getRight null -> RBNode<T,E>  setRight   RBNode<T,E> -> null  getParent null->RB Node<T,E>  setParent    RBNode<T,E> -> null  getPlayer    null -> E  setPlayer E->null  compareTo T-> int |

|  |
| --- |
| RBNode(T el, E player)  “Constructor de la clase, le da un valor a val y player, los otros nodos se quedan como nulos”  { pre: }    { post:  val y player ahora tienen un valor y RBNode es creado} |

|  |
| --- |
| getColor()  “Devuelve el valor del color, que es un enum”  {pre: la existencia del RBNode}  {post: retorna el color del RBNode} |

|  |
| --- |
| setColor(Color color)  “Le da un valor al color”  {pre: la existencia del RBNode}  {post: le da un nuevo valor a color} |

|  |
| --- |
| getVal()  “Devuelve el valor de val, es decir, el valor que tiene este nodo”  { pre: la existencia del RBNode}    { post:  retorna el val del RBNode} |

|  |
| --- |
| setVal (T val)  “Le da un valor a val”  { pre: La existencia del Node }  { post:  le da un nuevo valor a val} |

|  |
| --- |
| getLeft ()  “retorna el hijo izquierdo de RBNode, que es otro RBNode”  {pre: la existencia deRB Node al cual se le hace .getLeft()}  { post: Se retorna un RBNode o nulo si no hay un hijo izquierdo} |

|  |
| --- |
| setLeft (RBNode<T,E> left)  “left se instancia como el nuevo hijo izquierdo del RBNode, sin importar que ya hubiera un hijo izquierdo”  { pre: la existencia de RBNode}  { post:  se instancia left como el left del nodo al que se le llamo este metodo} |

|  |
| --- |
| getRight()  “retorna el hijo derecho de RBNode, que es otro RBNode”  {pre: la existencia de Node al cual se le hace. getRight()}  { post: Se retorna un RBNode o nulo si no hay un hijo derecho} |

|  |
| --- |
| setRight (RBNode<T,E> right)  “right se instancia como el nuevo hijo derecho del RBNode, sin importar que ya hubiera un hijo derecho”  { pre: la existencia de RBNode}  { post:  se instancia right como el right del nodo al que se le llamo este metodo} |

|  |
| --- |
| getParent()  “retorna el padre de RBNode, que es otro RBNode”  {pre: la existencia de RBNode al cual se le hace. getParent ()}  { post: Se retorna un RBNode o nulo si no hay un padre} |

|  |
| --- |
| setParent (RBNode<T,E> parent)  “parent se instancia como el nuevo padre del RBNode, sin importar que ya hubiera un padre”  { pre: la existencia de RBNode}  { post:  se instancia parent como el parent del nodo al que se le llamo este metodo} |

|  |
| --- |
| getPlayer ()  “Retorna el valor de player que tiene el Node”  { pre: la existencia de Node}  { post:  retorna el atributo player de tipo E del Node} |

|  |
| --- |
| setPlayer (E player)  “Se le da un nuevo valor a el atributo player”  { pre: la existencia de Node}  { post:  se cambia el valor del atributo player} |

|  |
| --- |
| compareTo (T n)  “Compara el valor del atributo val con n, si retorna un entero negativo, val<n, si el entero es positivo val>n y si retorna 0: val==n”  { pre: la existencia de Node}  { post:  retorna un entero que indica si val es mayor, menor o igual a n} |

# JUNIT

## Setup Scenarios del modelo

|  |  |  |
| --- | --- | --- |
| Nombre | Clase | Escenario |
| setupScenario1 | PlayerTest |  |
| setupScenario1 | AppMangerTest |  |
| setupScenario2 | AppMangerTest |  |

## Diseño de casos de prueba del modelo

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que el método get(String search) funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| PlayerTest | testGet | setupScenario1 | String search = “name”;  String search = “age”;  String search = “points”;  String search = “team”;  String search = “reBounds”;  String search = “blocks”;  String search = “assists”;  String search = “steals”; | Devuelve correctamente cada uno de los valores que se espera |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la importación funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testImport | setupScenario1 |  | Se añade correctamente los 200,000 jugadores |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que se creen los 4 arboles | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testTreeCreation | setupScenario1 |  | Se crean los 4 arboles binarios |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda lineal funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testLinearSearchForFirstPlayerWithValue | setupScenario2 | String search=”name”  String search=”Tori” | Retorna correctamente el jugador que se llama Tori |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda lineal con rango funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testlinearSearchWithRange1 | setupScenario2 | String search=”age”  Int max=40  Int min= 20 | Retorna un arreglo con los 5 jugadores |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda lineal con rango funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testlinearSearchWithRange2 | setupScenario2 | String search=”age”  Int max=40  Int min= 30 | Retorna un arreglo con 2 jugadores |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda lineal de varios jugadores con mismo valor funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testlinearSearch | setupScenario2 | String search=”age”  String searchedFor=”29” | Retorna un arreglo con 3 jugadores |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda con los árboles funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testSearchWithTree | setupScenario2 | String search=”age”  String searchedFor=”29” | Retorna un arreglo con 3 jugadores |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda con los árboles funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testSearchWithTree1 | setupScenario2 | String search=”points”  String searchedFor=”33” | Retorna un arreglo con 1 jugador, en este caso, Aimil |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda con los árboles funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testSearchWithTree2 | setupScenario2 | String search=”reBounds”  String searchedFor=”2” | Retorna un arreglo con 2 jugadores |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la búsqueda con los árboles funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| AppMangerTest | testSearchWithTree3 | setupScenario2 | String search=”blocks”  String searchedFor=”34” | Retorna un arreglo vació |

## Diagrama de las pruebas del modelo

Diagrama

Descripción generada automáticamente

## Setup Scenarios de las estructuras de datos

|  |  |  |
| --- | --- | --- |
| Nombre | Clase | Escenario |
| setupScenario1 | NodeTest |  |
| setupScenario1 | BinaryTreeTest |  |
| setupScenario2 | BinaryTreeTest |  |

## Diseño de casos de prueba del modelo

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la comparación de nodos funcione correctamente, es decir, que el nodo derecho este donde debería estar, es decir, que el método retorne 1 | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| NodeTest | testCompareTo | setupScenario1 |  | Debería retornar 1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la comparación de nodos funcione correctamente, es decir, que el nodo izquierdo este donde debería estar, es decir, que el método retorne -1 | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| NodeTest | testCompareTo2 | setupScenario1 |  | Debería retornar -1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la comparación de nodos funcione correctamente, en este caso, se comparar el nodo con el mismo | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| NodeTest | testCompareTo3 | setupScenario1 |  | Debería retornar 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la inserción de nodos funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| BinaryTreeTest | testInsert | setupScenario1 | Int=40  Player=null | El nuevo nodo se debería añadir como hijo derecho |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo de la prueba: Verificar que la inserción de nodos funcione correctamente | | | | |
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| BinaryTreeTest | testInsert1 | setupScenario1 | Int= 20  Player=null | El nuevo nodo se debería añadir como hijo izquierdo |