Martin Cifuentes, Gabriel Delgado y Felipe Barreto

Documentación de la tarea integradora 2

Contenido

[Requerimientos 1](#_Toc86261777)

[Requerimientos no funcionales 1](#_Toc86261778)

[Diagrama de clases 2](#_Toc86261779)

[Pruebas unitarias 2](#_Toc86261780)

[Diagrama de pruebas unitarias 2](#_Toc86261781)

[TAD 2](#_Toc86261782)

[TAD de árbol binario 2](#_Toc86261783)

[TAD de Nodo 4](#_Toc86261784)

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

# Pruebas unitarias

# Diagrama de pruebas unitarias

# 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} |