BASKETBALL MANAGER

DIEGO FERNANDO HIDALGO LÓPEZ
LAURA DANIELA MARTÍNEZ ORTIZ

1 NOVEMBER 2021

ALGORITHMS AND DATA STRUCTURES

ICESI UNIVERSITY

Entregables.

1. Informe PSP0. Cada estudiante debe entregar el informe de su desarrollo.
2. Entrega informe del método de la ingeniería.
3. Especificación de Requerimientos y Diseño. Los requerimientos funcionales debe incluirlos en la fase 1.
4. Diseño del TAD para cada estructura de datos requerida.
5. Diseño del diagrama de clases desacoplado y utilizando generics.
6. Diseño de los casos de prueba, para las estructuras de datos y para el programa.
7. Diseño del diagrama de clases de pruebas unitarias automáticas.
8. Implementación del programa en Java.
9. Implementación de las pruebas unitarias automáticas.
10. Usted debe entregar el enlace del repositorio en GitHub o GitLab con los elementos anteriores. El nombre del repositorio debe estar en inglés, en minúsculas y si tiene varias palabras, éstas van separadas por un guión. Su repositorio debe corresponder con un proyecto de eclipse. Debe tener al menos 10 commits con diferencia de 1 hora entre cada uno de ellos. En el repositorio o proyecto de eclipse debe haber un directorio llamado docs/ en el cual deberán ir cada uno de los documentos del diseño.

Engineering method

Problem context

The enormous amount of data generated by basketball makes necessary the development of a tool that facilitates the management and consultation of the information. For this purpose, it's required an application that allows to register and consult the statistics about the activities of this sport.

Solution Development

In order to solve the problem previously presented, it will be used the steps indicated in the Method of the Engineering to develop a solution following a systematic approach and taking into account the proposed problematic situation.

Based on the description of the Engineering Method in the book "Introduction to Engineering" by Paul Wright, the steps to be followed in the development of the solution are presented as follows:

1. Problem identification
2. Compilation of necessary information
3. Search of creative solutions
4. Transition from the formulation of ideas to preliminary designs
5. Evaluation and selection of the best solution
6. Preparation of reports and specifications
7. Design implementation

1. Problem identification

At this stage, the needs of the problematic situation are recognized, as well as its symptoms and conditions under which it must be resolved.

Identification of needs and conditions

- FIBA wants to consolidate, in an application, the most relevant data of each of the professional players on the planet.
- The objective is to be able to make different consults that allow the analysis of this data, to know patterns about the development of the sport, the criteria that are gaining strength or, in general, where the sport is heading nowadays. The solution to the problem must ensure that:
    - It includes data per player for the following items: name, age, team and 5 statistics (e.g. points per game, rebounds per game, assists per game, steals per game, blocks per game).
    - It has a quick access to the data (efficiency in the searches).
    - It shows the search time to justify using balanced trees.
    - The search to filter players is not linear.
    - It works with at least 200,000 valid data.

Problem definition

FIBA requires the development of a program that allows you to register and consult the statistics of each basketball player in an efficient and faster way.

2. Compilation of necessary information

In order to have complete clarity in the concepts involved, a search is made for the definitions of the terms most closely related to the problem presented.

BST Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Source: GeeksforGeeks. (2021). Binary Search Tree. Recovered from: https://www.geeksforgeeks.org/binary-search-tree-data-structure/

AVL Tree

AVL tree is a Binary Search Tree (BST) that is height balanced, where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Each node, in addition to the information to be stored, must have the two pointers to the right and left trees, just like the binary search trees, and also the data that controls the balancing factor.

The balancing factor is the difference between the heights of the right and left trees:

By definition, for an AVL tree, this value must be -1, 0 or 1.

If the balance factor of a node is:

- 0 = the node is balanced and its subtrees have exactly the same height.
- 1 = the node is balanced and its right subtree is one level higher.
- - 1 = the node is balanced and its left subtree is one level higher.

If the balancing factor ≥ 2 it is necessary to rebalance.

Source: GeeksforGeeks. (2021). AVL Tree. Recovered from: https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

Red–black tree

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.

Source: Geeks for Geeks. (2021). Red–black tree. Geeks for Geeks. Recovered from: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

Generics

Generics mean parameterized types. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

Source: Geeks for Geeks. (2021). Sorting Algorithms. Geeks for Geeks. Recovered from: https://www.geeksforgeeks.org/generics-in-java/

Abstract Data Type (ADT)
The abstract data type is a special kind of data type, whose behavior is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these datatypes, we can perform different operations. But how those operations are working is totally hidden from the user. The ADT is made of primitive data types, but operation logics are hidden.
Source: Chakraborty, A. (2019). Abstract Data Type in Data Structures. Tutorials Point. Recovered from: https://www.tutorialspoint.com/abstract-data-type-in-data-structures

3. Search of creative solution

At the end of the brainstorming, the following solutions are proposed:

- *Alternative 1:* Implement an application that saves the player's information through an arraylist
- *Alternative 2:* Implement an application that stores the player's information in the ABB trees, AVL trees and the red-black tree, using threads and serialization. (graficos)
- *Alternative 3:* Implement an application that stores the player's information in the ABB trees, AVL trees and the red-black tree, using serialization but all through console.

4. Transition from the formulation of ideas to preliminary designs

First, ideas that are not feasible are discarded. We decided to discard alternative 1 because no cumple con las estructuras de datos solicitadas ni con el tiempo de búsqueda

The review of the other 2 alternatives leads us to the following:

- *Alternative 2.*
  - The player is searched for and when it is found the statistical information is displayed graphically.
  - The search time condition is fulfilled.
  - Allows adding, deleting and searching for players through the graphical interface.
  - Can add data through a file.
  - The statistical attributes are stored in separate trees and then the index is associated with the corresponding player.

- *Alternative 3.*
  - The player is searched and when found the statistical information is displayed by console.
  - It will take time to add and save a large amount of data because it does not handle threads.
  - Puede agregar datos a través de un archivo, pero tendrá que poner la ruta manualmente.
  - You can add data through a file, but you will have to put the path manually.

5. **Evaluation and selection of the best solution**

As the engineering design process evolves, the engineer can evaluate alternative ways to solve the problem in question. Commonly, the engineer abandons the design possibilities that are not promising, thus obtaining a set progressively smaller of options. Feedback, modification and evaluation can occur repeatedly as the device or system evolves from concept to final design.

Criteria

The criteria we chose in this case are those listed below. Next to each criteria, a numerical value has been established with the aim of establishing a weight that indicates which of the possible values of each criteria have more weight and, therefore, are more desirable.

- *Criteria A.* Specified complexity.
  - [2] Satisfied.
  - [1] Not satisfied.
- *Criteria B.* Results.
  - [3] Satisfied.
  - [2] Partially satisfied.
  - [1] Not satisfied.
- *Criteria C.* Waiting time.
  - [3] Little time.
  - [2] Normal time.
  - [1] Long time.
- *Criteria D.*Usability.
  - [3]Easy.
  - [2] Normal.
  - [1] Difficult.

Evaluation

Evaluating the above criteria in the alternatives that are maintained, we obtain the following table:

	Criteria A	**Criteria B**	**Criteria C**	**Criteria D**	**Total**
Alternative 2	2	3	3	3	11
Alternative 3	1	3	1	2	7

Selection

According to the previous evaluation, the Alternative 2 must be selected because it obtained the highest score according to the defined criteria.

6. **Preparation of reports and specifications**

Review the functional and non-functional requirements section and the class diagram.

7. **Design implementation**

Review the implemented project.

Requirement specification

Functional requirements

In order to correctly meet with the needs and functionalities required for this project, the system to be developed must be able to:

FR1: **Manage the information of the players** with a name, age, team and 5 statistics (to define)

- **FR1.1 -** It must be possible to register a new player.
- **FR1.2 -** It must be possible to modify the information of a specific player.
- **FR1.3 -** It must be possible to delete a specific player.

FR2: **Import information from a plain text file** with information of the players.

FR3: **Consult information of the players**

- **FR3.1 -** It must be possible to consult the general information of a specific player.
- **FR3.2 -** It must be possible to consult information using a search criteria.

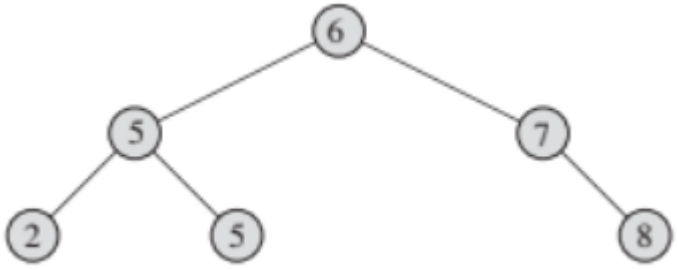
Non-functional requirements

In order to guarantee the correct operation of the system and assure the quality of the software, the system must have the next validations:

NFR1: **Tests.** Implement tests to make sure that the methods work correctly.
NFR2: **Generics.**

Abstract Data Type (ADT) Design

Binary search tree

ADT Binary search tree
BinarySearchTree= {<x_1, x_2, x_3, … ,x_n>, <k_1, k_2, k_3, … ,k_n>, Height = <height >} 
{inv: Each node x contains a key k: $x_1 = k_1$, $x_2 = k_2$, $x_3 = k_3$, … , $x_n = k_n$ Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key ≤ x.key. If y is a node in the right subtree of x, then y.key ≥ x.key. If a child or the parent is missing, the appropriate attribute contains the value NULL. The root node is the only node in the tree whose parent is NULL.}
Primitive operations: BinarySearchTree: →BinarySearchTree IsEmpty BST → Boolean Insert: BST x Key x Value →BinarySearchTree ContainsKey: BST x Key → Boolean ContainsValue: BST x Value → Boolean Remove: BST x Key →BinarySearchTree Minimum: BST →Value Maximum: BST →Value PreOrder: BST →BinarySearchTree InOrder: BST →BinarySearchTree PosOrder: BST →BinarySearchTree

BinarySearchTree() → Constructor
“Creates an empty binary search tree to add new nodes”
{pre: TRUE}
{post: Binary search tree initialized}

IsEmpty()→ Analyzer
“Checks if the binary search tree is empty ”

{pre: Binary search tree initialized}
{post: TRUE if the root == null FALSE if the queue isn't empty}

Insert(K key, V value) → Modifier
"Inserts a new node in the binary search tree"
{pre: BinarySearchTree tree= {<x_1, x_2, x_3, … ,x_n>, <k_1, k_2, k_3, … ,k_n>}}
{post: Added the element and increase the depth of the branch}

ContainsKey(K key)→ Analyzer
"Checks if a specific key is in the binary tree"
{pre: BinarySearchTree tree= {<x_1, x_2, x_3, … ,x_n>, <k_1, k_2, k_3, … ,k_n>}}
{post: TRUE if the key is found FALSE if the key doesn't exist}

ContainsValue(V value)→ Analyzer
"Checks if a specific value is in the binary tree"
{pre: BinarySearchTree tree= {<x_1, x_2, x_3, … ,x_n>, <k_1, k_2, k_3, … ,k_n>}}
{post: TRUE if the value is found FALSE if the value doesn't exist}

Remove(K key)→ Modifier
"Removes a key in the binary search tree"
{pre:BinarySearchTree tree= {<x_1, x_2, x_3, … ,x_n>, <k_1, k_2, k_3, … ,k_n>}}
{post: Deletes the element and decrease the depth of the branch}

Minimum()→ Analyzer
"Search for the minimum element in the binary tree"
{pre:BinarySearchTree tree= {<x_1, x_2, x_3, … ,x_n>, <k_1, k_2, k_3, … ,k_n>}}
{post: Value, returns the minimum value}

Maximum()→ Analyzer
“Search for the maximum element in the binary tree”
{pre:BinarySearchTree tree= {<$x_1, x_2, x_3, \dots, x_n$>, <$k_1, k_2, k_3, \dots, k_n$>}}
{post: Value, returns the maximum value}

Preorder()→ Analyzer
“Shows the binary tree in preorder traversal ”
{pre:BinarySearchTree tree= {<$x_1, x_2, x_3, \dots, x_n$>, <$k_1, k_2, k_3, \dots, k_n$>}}
{post: Preorder (Root, Left, Right)}

InOrder()→ Analyzer
“Shows the binary tree in inorder traversal ”
{pre:BinarySearchTree tree= {<$x_1, x_2, x_3, \dots, x_n$>, <$k_1, k_2, k_3, \dots, k_n$>}}
{post: Inorder (Left, Root, Right)}

posOrder()→ Analyzer
“Shows the binary tree in posorder traversal ”
{pre:BinarySearchTree tree= {<$x_1, x_2, x_3, \dots, x_n$>, <$k_1, k_2, k_3, \dots, k_n$>}}
{post: Postorder (Left, Right, Root)}

AVL Tree

ADT AVL Tree
AVLtree = {}
{inv: }
Primitive operations:

RED-BLACK Tree

ADT RED-BLACK Tree

Unit test design
Configuration of the scenes

Name	Class	Scenes

Design of test cases

Methods to add

Purpose of the test:				
Class	**Method**	**Scenes**	**Inputs**	**Results**