Algorithms race

Fourth laboratory

Algorithms And Programming II

Group No. 1

Author:

Sebastián García Acosta (A00361888)

Tutor:

Juan Manuel Reyes García

Teacher assistant:

Daniel Alejandro Fernández

Santiago de Cali

Sunday, May 14, 2020

Software Systems Engineering

ICESI University

Table of Contents

Problem statement

Due to the quarantine that most of humanity has had to accept due to a highly contagious disease, sports events have been suspended and people who love this type of show are a little disappointed. People cannot go out, but the algorithms are not contagious, so it has been suggested that you organize a tournament in which the algorithms of data structures that store elements can participate.

The great algorithm tournament will be like a kind of mini Olympic computer games in which each data structure is considered as a country, and each type of algorithm as a sport or discipline. Below is the competition scheme:

<table>
<tr><th>Data Structure \ Algorithm</th><th>Add - A</th><th>Search - C</th><th>Delete - D</th></tr>
<tr><td>ArrayList - AL</td><td rowspan="3">This competition consists of adding N randomly generated elements to the data structure.</td><td rowspan="2">This competition consists of consulting N randomly generated elements on a previously created data structure with N also random elements. Only the time for the N queries is taken.

The results of each query will return that the element exists or does not exist (boolean), but that will not be taken into account in the race.</td><td rowspan="2">This competence consists in eliminating N randomly generated elements on a previously created data structure with N random elements. Only the time of the N eliminations is taken.

The results of the deletion may be that the element is deleted or that it is not deleted because it does not exist, but that result will not be taken into account in the race.</td></tr>
<tr><td>Linked List - LL</td></tr>
<tr><td>Binary Search Tree - BST</td></tr>
</table>

The winning data structure is the one that manages to complete the operation before the others.

The previous three races will be carried out in two modalities: iterative and recursive, so the following six (6) races will be finally:

I means Iterative and R means Recursive

# Carrera	Algoritmo		Algoritmo		Algoritmo
1	AL - A - I	vs	LL - A - I	vs	BST - A - I
2	AL - S - I	vs	LL - S - I	vs	BST - S - I
3	AL - E - I	vs	LL - D - I	vs	BST - E - I
4	AL - A - R	vs	LE - A - R	vs	BST - A - R

5	AL - S - R	vs	LE - S - R	vs	BST - S - R
6	AL - E - R	vs	LE - E - R	vs	BST - E - R

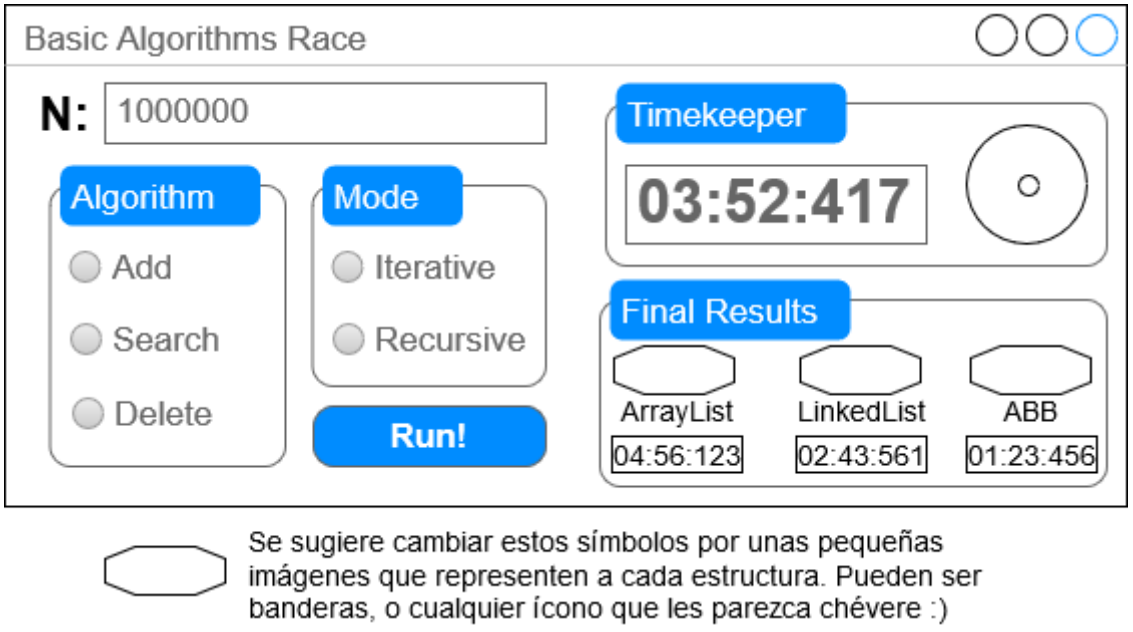
Each of the algorithms will run on its own thread. This is equivalent, in athletics races, to each athlete running in his own lane.

At the time the race starts, it will also start a stopwatch that will count the time in minutes: seconds: milliseconds since the race started. It will also start a small animation of a circle that gradually shrinks to a point and then grows to the original size of the circle and then shrinks again, like this until the race ends and the last competitor finishes. That circle will not be alone, there will be another circle in the same position that will do the same but on the contrary, it decreases when the other increases and vice versa.

The elements to be stored in the above data structures are long. Remember that the minimum value of long is Long.MIN_VALUE and the maximum value is Long.MAX_VALUE.
You must define a GUI exclusively with JavaFX that allows you to run previous races with an N indicated by the user.

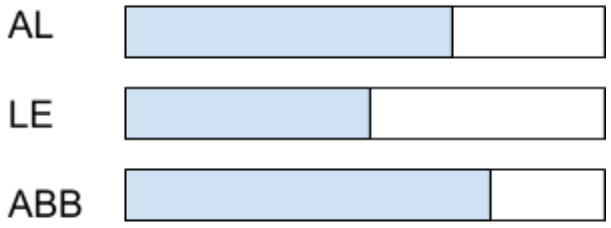
The GUI should have at least the elements described in the following wireframe. It says at least, because you can add additional elements that you consider to improve the user experience.

The colors, position and size are for reference, these characteristics of the GUI elements can be changed from what they are in the example wireframe.



One of the functionalities that can be done extra (additional, optional) is that you can see the status of each algorithm during the race. Since the career of each algorithm consists of doing N operations, seeing the state of each one can consist of showing in the GUI how many operations each algorithm has up to now, that way you can see who is beating who. And so it is much more like sports !! People will be very happy to see how those numbers are being updated. If you can already show the number of operations performed by each algorithm and still want to do something else, you could show the

race in a more visual way. You can do this with three parallel bars that fill as the algorithms carry out more operations. For example like this:



This laboratory is graded with the [Rubric for Laboratory 4 on Algorithm Careers]

Clarifications:

- In the ArrayList data structure does not apply add iterative or add recursive, just do the .add ().
- In the ArrayList data structure when removing, it is not allowed to use remove (Object), of course if you can use remove (int).
- The structure of linked lists can be double but not circular and without a pointer to the last element.
- Note that the generation of the random numbers is not counted within the time of each operation of each data structure.

Functional requirements

The GUI application must be able to:

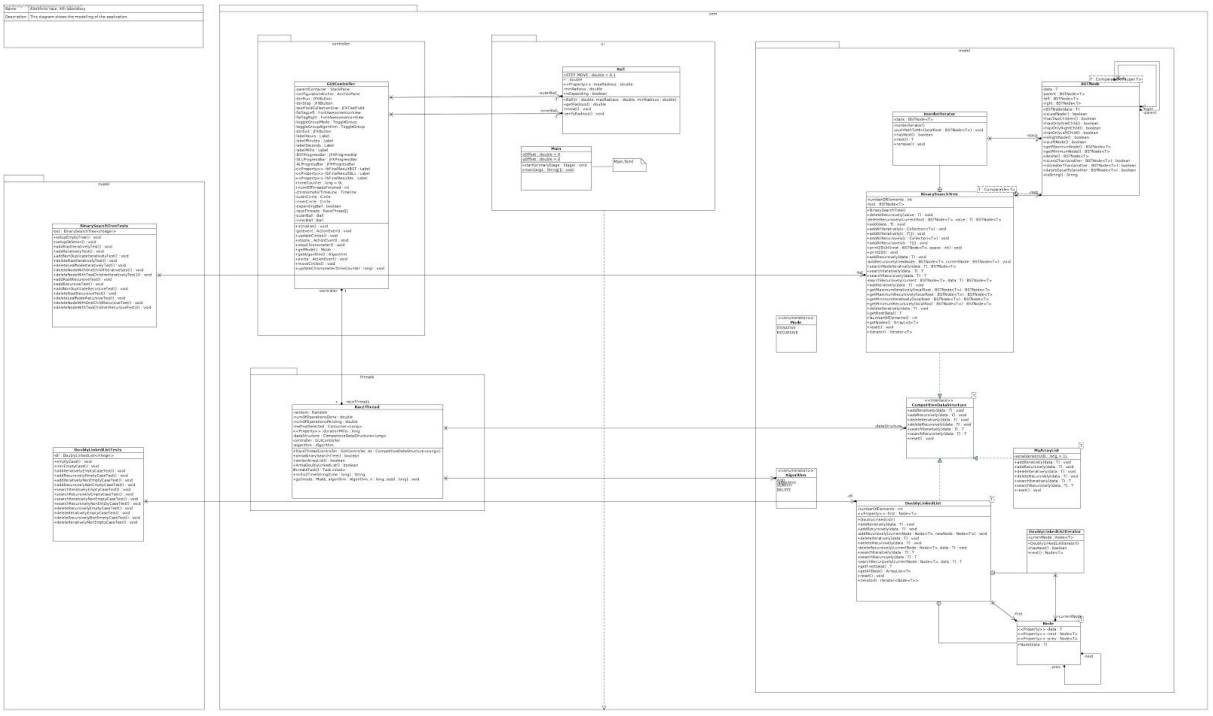
- Execute different algorithms (Add, Search, Delete) in two possible modes (Recursively, Iteratively) on three different Data Structures (Binary Search Tree, Doubly Linked List, ArrayList) concurrently and measure the time that it took for each one to add search, or delete random long numbers, the number of long numbers, the mode, and the algorithm are specified by the user. In the case of Search and Delete, data structures should firstly add the number of random long numbers, and then execute the algorithm.
- Show a chronometer that stops whenever the three competitive data structures finish their process.
- Show an animation that runs during the execution of the three data structures.
- Indicate whenever a data structure cannot complete the execution of the algorithm.

Non-functional requirements

The application must be able to:

- In the ArrayList data structure does not apply add iterative or add recursive, just do the .add ().
- In the ArrayList data structure when removing, it is not allowed to use remove (Object), of course if you can use remove (int).
- The structure of linked lists can be double but not circular and without a pointer to the last element.
- Note that the generation of the random numbers is not counted within the time of each operation of each data structure.

Classes Diagram



Test cases design

(non-circular) Doubly Linked List

Scenarios configuration

Name	**Class**	**Stage**
emptyCase	LinkedListTest	An initialized object of the LinkedList class.
nonEmptyCase	LinkedListTest	An initialized object of the LinkedList class with five objects of the class Integer added: (4, 666, 1879, 1912, 1643)

unsortedSetup	LinkedListTest	An initialized object of the LinkedList class 6 objects of the class Double added: (-666.314, 100, 2.718, 2, 0, 1)

Tests cases design:

Test objective: Test that the class correctly adds a new Integer.				
Class	**Method**	**Stage**	**Input**	**Result**
LinkedList	add<I & R>[1]	emptyCase	666	A new Integer (666) has been added to the LinkedList and is the first node.
LinkedList	add<I & R>*	nonEmptyCase	1	A new Integer (1) has been appended to the LinkedList and can be found

Test objective: Test that the class correctly deletes an element.				
Class	**Method**	**Stage**	**Input**	**Result**
LinkedList	delete<I & R>*	emptyCase	4	Integer(4) cannot be found within

[1] * <method>”<I & R>” Notation means that <method> will be tested in its recursive and iterative version. This is done in this way in order to ensure that both programming paradigms lead to the same exact result regardless of their logic.

				the elements of the linked list.
LinkedList	delete<I & R>*	nonEmptyCase	666	Integer(66) was successfully deleted

Test objective: Test that the class correctly searches a new object. If found, return it, else, return null.				
Class	**Method**	**Stage**	**Input**	**Result**
LinkedList	search<I & R>*	nonEmptyCase	1879	1879 has been returned
LinkedList	search<I & R>*	emtpyCase	666	returns null
LinkedList	search<I & R>*	nonEmptyCase	364	returns null

Binary Search Tree

Stages configuration:

Name	**Class**	**Stage**
setupEmptyTree	BinarySearchTreeTest	An object of the BinarySearchTree class empty.

setupBinaryTreeForDeletion	BinarySearchTreeTest	An object of the BinarySearchTree class with 17 Integers added (63, 70, 66, 65, 64, 50, 40, 35, 30, 36, 45, 46, 51, 55, 56, 60, 37).

Test Cases design:

Test objective: Test that the class correctly adds a new Integer at the root.				
Class	**Method**	**Stage**	**Input**	**Result**
BinarySearchTree	add<I & R>**	setupEmptyTree	3	The given Integer has been added at the root.

Test objective: Test that the class correctly adds a given array of numbers.				
Class	**Method**	**Stage**	**Input**	**Result**
BinarySearchTree	addAll<I & R>*	setupEmptyTree	Integer[] {1, 4, 53, 3, 2}	The given numbers are correctly added to the tree.

Test objective: Test that the class correctly adds a given array of numbers without repetitions.				
Class	**Method**	**Stage**	**Input**	**Result**

BinarySearchTree	addAll<I & R>*	setupEmptyTree	Integer[] {12, 1, 3, 3, 3, 3, 3, 3, 3, 3}	The repeated numbers are added only once.

Test objective: Test that the class correctly deletes a given Integer when it is the root.				
Class	**Method**	**Stage**	**Input**	**Result**
BinarySearchTree	delete<I & R>*	setupBinaryTreeForDeletion	50	The given Integer of the Tree has been deleted.

Test objective: Test that the class correctly deletes the given Integers (nodes) when they are leafs.				
Class	**Method**	**Stage**	**Input**	**Result**
BinarySearchTree	delete<I & R>*	setupBinaryTreeForDeletion	30, 63, 70	The given Integers of the Tree have been deleted.

Test objective: Test that the class correctly deletes the given Integers (nodes) when they have one children.				
Class	**Method**	**Stage**	**Input**	**Result**

BinarySearchTree	delete<I & R>*	setupBinaryTreeForDeletion	36, 45, 66	The given Integers of the Tree have been deleted.

Test objective: Test that the class correctly deletes the given Integers (nodes) when they have two children.				
Class	**Method**	**Stage**	**Input**	**Result**
BinarySearchTree	delete<I & R>*	setupBinaryTreeForDeletion	40, 55, 35	The given Integers of the Tree have been deleted.