Algoritmos e Estruturas de Dados

2022-2023

Projeto 2 – Coleções

Introduction

In the 2nd Project we aim to explore the creation and use of data structures used to represent collections of objects. We will explore the implementation of Smart Lists and queues implemented through arrays.

It is not necessary to have a fully implemented problem to get the corresponding grade on the project. Even if you don't have time to implement some of the requested methods, you should at least write its header, so that *Mooshak* can compile the code and run tests on the methods you implemented.

Problem A: Implementing Smart Lists (12 valores)

A smart list is a list designed to optimize linear access to a list. A linear access corresponds to searching for an element in the list (for example through a key). In traditional lists, in the worst case a linear search involves going through the entire list and comparing the object we are looking for with all the objects, and on average we go through half of the list.

A smart list tries to improve the average linear lookup time using a few simple techniques. The main idea is that if the accesses/searches to a list have some characteristics such as locality of reference, or follow the Pareto principle (80% of the searches are carried out on 20% of the items), we can improve efficiency by bringing to the front of the list the items that will be searched more frequently.

In this project we will implement and test 2 techniques. The first is the *MoveToFront* technique which, as the name implies, corresponds to searching for an element, and when this element is found, moving it to the beginning of the list (with the aim of making the search for this element more efficient in the future). The second technique, called Transpose, is similar to the previous one, but instead of moving the found element to the front of the list, it is simply swapped with its predecessor. The following image shows an example of the application of both techniques.

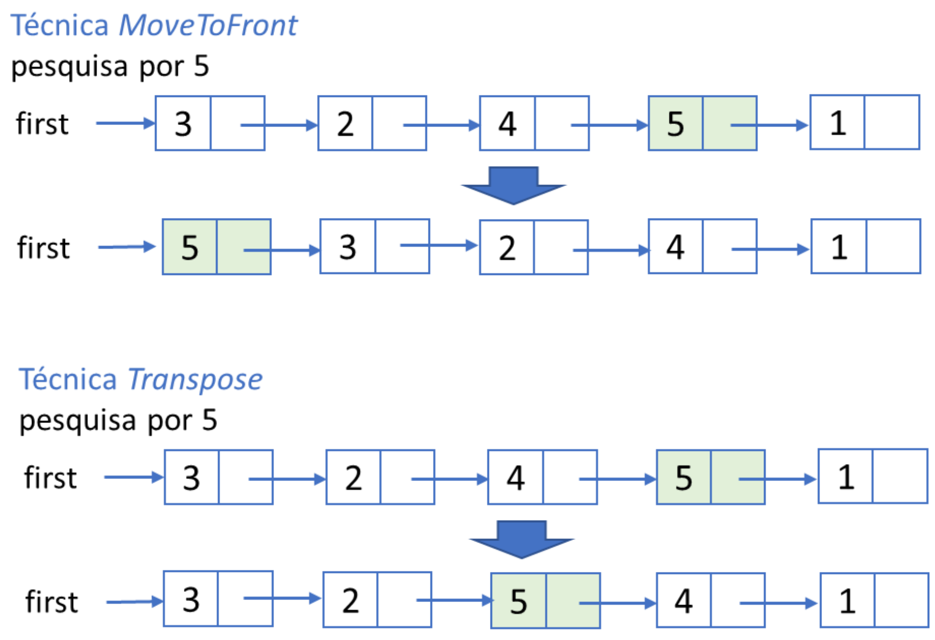

Figura 1 – Illustration of MoveToFront and Transpose techniques when searching for element 5

Implement the SmartList class, which represents a list where elements are stored and searched in a way that makes the most frequently searched items closer to the beginning. The class must comply with the following specification:

SmartList<Item> – represents a smart list of elements of type Item. Most frequently searched items tend to be brought to the top of the list to speed up future searches.	
	`SmartList()`
Creates a new empty *SmartList*	
`void`	`add(Item item)`
Adds an item at the beginning of the list. The item cannot be null. If null, an IllegalArgumentException will be thrown.	
`Item`	`searchMTF(Item item)`
Given an item, checks if the item exists in the list, and if there is an item considered equal (the equals method returns true), the item stored in the list is returned. If the list is empty, or the item is not found, it must return null. This method applies the *MoveToFront* technique to the searched item if it exists.	
`Item`	`searchTrans(Item item)`
Similar to the previous method. However, this method applies the Transpose technique to the searched item, if it exists.	
`Item`	`search(Item item)`
Similar to the previous ones. However, in this method you should choose the technique (*MTF*, *Transpose*, or other) with the best results according to the empirical tests performed.	
`Item`	`search(Item item, Comparator<Item> c)`
This method should implement the same functionality as the previous one. However, instead of using the *equals* method to find the item, it uses the received comparator.	
`Item`	`remove(Item item)`
Receives an item, and searches for the item in the list. If there is an item considered the same, that item is removed from the list and returned. If the list is empty, or the item is not found, it should return *null* and nothing is removed.	
`float`	`getAvgMTFCompares()`
For the current list, returns the average number of comparisons made by the *searchMTF* method in previous calls for this list. For example, in a list with the elements {1,2,3,4,5}, if we search for 1, there will be 1 comparison. If we look for 4, there will be 4 comparisons. If we look for 7, there will be 5 comparisons.	
`float`	`getAvgTransCompares`

For the current list, returns the average number of comparisons made by the *searchTrans* method in previous calls for this list. For example, in a list with the elements {1,2,3,4,5}, if we search for 1, there will be 1 comparison. If we look for 4, there will be 4 comparisons. If we look for 7, there will be 5 comparisons.	
`void`	`clear()`
Removes all items from the list, leaving it empty.	
`boolean`	`isEmpty()`
Returns true if the list is empty and false otherwise	
`int`	`size()`
Returns the size (number of elements) of the list	
`Object[]`	`toArray()`
This method returns an array with all the objects in the list, using the order in which the elements are stored in the list (eg the 1st item will be stored at index 0).	
`SmartList<Item>`	`shallowCopy()`
Returns a shallow copy of the list. A shallow copy copies the list structure without copying each item individually	
`Iterator<T>`	`Iterator()`
Returns a stateful iterator to iterate over the elements of the list, in the order in which the elements are arranged. This iterator must implement the *hasNext()* and *next()* methods.	
`void`	`main(String[] args)`
Main method, which should be used to test the above methods.	

Technical requirements: The list must be implemented using a linked list. You cannot use any of Java's native collections (*ArrayList*,*LinkedList*,*Vector*,etc) in your implementation. Implement the *SmartList* class in the *SmartList.java* file. The class must be defined in the *aed.collections* package. Other auxiliary classes must be defined in the same file.

Submit only the file ***SmartList.java*** in problem A.

Empirical tests

Use empirical tests, including doubled ratio tests, to determine the average running time and order of temporal growth of the *searchMTF* and *searchTrans* methods. Also look at the average number of comparisons. Compare several different situations, such as completely random accesses to elements, and accesses to elements using a Pareto distribution. Based on your analysis choose the best technique for the search method. Write a summary of the tests and results, including the r values (doubled ratio), the averages of comparisons, and a brief analysis of the results as comments in the Java file to be submitted

The main method must implement the tests and experiments used. Although this method is not automatically validated by *Mooshak*, it will be taken into account in the project validation and will count towards the final grade of the project.

Problema B: Implementing a Queue using arrays (8 valores)

Implemente a classe *QueueArray*, que representa uma fila, implementada através de um array. A implementação deverá seguir a seguinte especificação:

QueueArray<Item> – Implement the QueueArray class, which represents a queue, implemented through an array. The implementation should follow the following specification:	
	`QueueArray()`
Creates an empty queue.	
`void`	`queue(Item item)`
Puts an item at the tail of the queue. The item cannot be null. If null, an IllegalArgumentException will be thrown.	
`Item`	`dequeue()`

Removes and returns the item at the head of the queue. If the queue is empty, return null	
`Item`	`peek()`
Returns the item at the head of the queue, but does not remove it. Returns null if the queue is empty	
`boolean`	`isEmpty()`
Retorna *true* se a fila estiver vazia e *false* caso contrário	
`Int`	`size()`
Returns the size (number of elements) of the queue	
`QueueArray<Item>`	`shallowCopy()`
Returns a shallow copy of the queue. A shallow copy copies the structure of the queue without copying each item individually	
`Iterator<T>`	`Iterator()`
Returns a stateful iterator to iterate over elements in the queue. This iterator must implement the *hasNext()* and *next()* methods.	
`void`	`main(String[] args)`
Main method, which should be used to test the above methods.	

Technical requirements: This queue must be implemented using an *Array* to store the elements. For the implementation to be as efficient as possible both in terms of memory and in terms of time, you should minimize the number of resizes, and you should make the most of the available space before a potential resize. It is also not allowed to shift elements (move to a position next to it) of the array (unless during resizes). You can't use any of Java's native collections in your implementation (*ArrayList*,*LinkedList*,*Vector*,etc), but you can use *Arrays*.

Implement the *QueueArray* class in the *QueueArray.java* file. The class must be defined in the *aed.collections* package. Other auxiliary classes you need must be defined in the same file.

Submit only **file QueueArray.java** in Problem B.

Empirical tests

Use empirical tests, including doubled-ratio tests, to determine the average running time and order of temporal growth of the queue and dequeue methods. Write a summary of the tests and results, including the values of r (doubled ratio), and a brief analysis of the results as comments in the Java file to be submitted

The main method must implement the tests and trials used. Although this method is not automatically validated by *Mooshak*, it will be taken into account in the project validation, and will count towards the final grade of the project.

Conditions

The project must be carried out individually. The same or very similar projects will lead to failure in this course. The faculty will be the only judge of what is considered or not to be copied in a project.

The project's source code must be delivered electronically, through the *Mooshak* system, until 23:59 on the 28th of October. Validations will take place the following week, from 31 October to 4 November. Students will have to validate the code together with the teacher during laboratory hours corresponding to the shift in which they are enrolled. The evaluation and corresponding grade of the project will only take effect after the validation of the code by the teacher.

The evaluation of code execution is done automatically through the *Mooshak* system, using various tests configured in the system. The execution time of each test is limited, as well as the memory used. It is not necessary to register for those who have already registered for the 1st project, and you can use the same username and password.