|  |
| --- |
| Algorithms and Datastructure 1 Year 2 - Mathematical Engineering InHolland University of Applied Science |
| Assignment Hat |
|  |

|  |
| --- |
| Project by:  572481 - Adu, Stephen and 570027- Andreicha, Semida  31-3-2017 |

Contents

[1. Introduction 2](#_Toc478738404)

[1.1 Problem Description Assignment Focus 2](#_Toc478738405)

[1.2 Assignment Focus 2](#_Toc478738406)

[2. Methodology 2](#_Toc478738407)

[3. Results 3](#_Toc478738408)

[4. Conclusion 4](#_Toc478738409)

[5. Bibliography 4](#_Toc478738410)

[6. Appendices 4](#_Toc478738411)

[6.1 Implementations 4](#_Toc478738412)

[6.2 Table 4](#_Toc478738413)

# 1. Introduction

## 1.1 Problem Description Assignment Focus

A Hat is a data structure that can be used to retrieve random elements. For instance, it can be used to draw names. It supports the following API[[1]](#footnote-1):

Public class Hat<Item>

|  |  |  |
| --- | --- | --- |
|  | Hat() | *create a new empty hat* |
| Boolean | isEmpty() | *is the hat empty?* |
| int | size() | *number of items in the hat* |
| void | give(Item item) | *add an item to the hat* |
| Item | take() | *delete a random item from the hat and return it* |

It is needed to find the best implementation for a program with the class Hat[[2]](#footnote-2), and create an efficient implementations of the required methods. As well the determinacies of the average case time complexity [[3]](#footnote-3)of each of the methods. In addition experiments with doubling ratio should be used to verify the time complexity of the implementation.

## 1.2 Assignment Focus

The *main focus* for this assignment is the implementation of a program, that has the most efficient time complexity for each method implemented in class Hat. As well as the process of testing the implementation.

The main steps that are going to be considered as guidelines are the following:

* 1. An efficient implementations of the required methods (Section 1.1).
  2. A simple test, to test all of the program’s implementations.
  3. The average case time complexity of each of the methods.
  4. Using the doubling ratio experiments[[4]](#footnote-4) determine the time complexity of the program’s implementation.

# 2. Methodology

The methodology explored in this assignment reflects the class Hat and its complexity of implementations. By means of collecting data the research will then attempt to develop an program which has the best average time complexity during runtime. The process of development has several phases that are being implemented to get a final result.

The first step into development begins with data collection, which is based on the information presented during the university lectures of Algorithms and Datastructures 1. During the lectures the basics of implementing basic data structures were thought. As well some other resources used to gather information regarding the complexity levels were from an Algorithms textbook (Sedgewick & Wayne, 2011).

Then the process of development shifts towards the second step which is the program development, where the process of implementation of the main classes is happening.

The tertiary step into development is the tests implementation, which are to help find the average time complexity for the methods in the developed program, by the use of the doubling ratio experiment.

# 3. Results

The results of the development of the assignment Hat, which results in a data structure that can be used to retrieve random elements, has been developed in sequential steps which involved: data collection, which gathered information for the development of the program. Then the tests implementation to find the average time complexity for each of the methods found in the developed program.

During the data collection session a lot of information has been collected, which was to be further used in the development of the program. The time complexity is scalable from 𝑂(1) [[5]](#footnote-5)to 𝑂(2𝑛) (Preez, B. D. ,n.d.), where N is the number of items in the Hat.

The program development began with the creation of the methods in the class Hat, that have to be implemented for the program to be functional ( Section 6 Appendices). Firstly a TreeMap is taken, with an Integer as key and generic T as a value, and made an object in the class Hat. Then the method isEmpty checks and returns true if empty, and false if the number of elements is greater than 0. The method size() returns the size/length of the array. While the method give(T item), takes as a parameter an item, of type T(generic), and places it into the collection. For removing the method take() removes a random item from the collection, as well it uses counter to keep count of size.

The tests were done using the doubling ratio experiment, results showing the complexity of the methods used in the program( Section 6 Appendices). The implementation of the test included an object of type hatTreeMap, storing integers; a r which is for the random object to be taken out of the Hat; a timer t; and a double which calculates the elapsed time.

The first test is done on the execution of the Give(), where it loops until the size of the collection is dine. Then the timer is started, followed by the Give() method of the hatTreeMap which is called to give a random object. Following the elapsed time is calculated and the method formatTimeTaken() prints the size of the hatTreeMap and the elapsedTime.

The second test is done on the take(), by starting the timer t.start(). Then calling the take() method from the hatTreeMap, and then calculating the elapsed time. Afterwards printing the size of the hatTreeMap and the elapsedTime, using formatTimeTaken().

Thirdly the isEmpty() method was tested, by starting the timer and calling the method isEmpty() from the hatTreeMap. Then the elapsed time is calculated, and printed out by the method formatTimeTaken() together with the size of the hatTreeMap.

Lastly the size() method was tested,starting the timer, calling the method form hatTreeMap, calculating elapsed time and printing out the size of the hatTreeMap as well as the elapsed time.

After running, the implemented tests, the time complexity for each of the methods was determined:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Give() | take() | isEmpty() | size() |
| Time complexity | *O(Log(n))* | *O(Log(n))* | *O(1)* | *O(1)* |

# 4. Conclusion

In the class Hat there are implementations of the required methods from Section 1.1. By the conduction of a simple test the average case time complexity of each of the methods was found(Section 3). Since a TreeMap is self-balancing, this means that when executing no object takes longer than any other. Thus the time complexity of the methods give() and take() was determined to be *O(Log(n)).* While the average case time complexity for the methods isEmpty() and Size() is considered to be O(1). The reason for this is because they are bounded by a value that does not depend on the size of the input, they help at reading variables.

Using the doubling ratio experiments determine the time complexity of the program’s implementation.

# 5. Bibliography

Sedgewick, R., & Wayne, K., (2011). Algorithms (Fourth ed.) Princeton University.

Preez, B. D. (n.d.). The simple Big-O Notation Post. Retrieved February 26, 2017, from [https://www.javacodegeeks.com/2011/04/simple- big-o-notation-post.html](https://www.javacodegeeks.com/2011/04/simple-%20big-o-notation-post.html)

# 6. Appendices

## 6.1 Implementations

## 6.2 Table

1. An *Program Programming Interface* is a set of routines, protocols and tools for building program. [↑](#footnote-ref-1)
2. A *class* is the blueprint from which individual objects are created. Class Hat is a data structure that can be used to retrieve random elements (such as String [words], Integers [numbers], etc.). [↑](#footnote-ref-2)
3. Of an algorithm is the amount of some computational resource (time) used by an

   algorithm, averaged over all possible inputs. [↑](#footnote-ref-3)
4. An experiment that allows a user to analyze time taken for an algorithm to run, predicting how long it takes for the algorithm to do the task given the amount of data it has to handle. [↑](#footnote-ref-4)
5. Order of growth [↑](#footnote-ref-5)