Annals of DAAAM for 2012 & Proceedings of the 23rd International DAAAM Symposium, Volume 23, No.1, ISSN 2304-1382
ISBN 978-3-901509-91-9, CDROM version, Ed. B. Katalinic, Published by DAAAM International, Vienna, Austria, EU, 2012
Make Harmony between Technology and Nature, and Your Mind will Fly Free as a Bird
Annals & Proceedings of DAAAM International 2012

AN EMPIRICAL STUDY OF ALGORITHMS PERFORMANCE IN IMPLEMENTATIONS OF SET IN JAVA

KUCAK, D[anijel]; DJAMBIC, G[oran] & FULANOVIC, B[ojan]

Abstract: *Developers very often face the need to use a collection of data that does not allow duplicates. Java programming language for this purpose defines interface Set and several implementation classes of Set. The aim of the experiment was to measure and compare the efficiency of algorithms for containing element, iteration, insertion and removing in/from a set of elements. An experimental case study is done through measuring the performance of following implementations of interface Set: HashSet and LinkedHashSet. The contribution of the study is based on the realized experimental case study analyses.*
Keywords:*Java, Set, Algorithm, Collections*

1. INTRODUCTION

In the majority of software applications executed in computing devices like PC's, tablets, smart-phones and other mobile devices, amounts of processed (stored and retrieved) information are getting larger. Representing information is fundamental to computer science [1].

In algorithmic solution of a problem the choice of data representation is not a simple process and is a reasonably difficult one. Very often, in solving problems we need to work with data that are characterized by not containing duplicates.

Nowadays, one of the most prominent and well-known technology is Java. A part of Java which deals with the structures of data is called the Collection Framework. The Collections framework is the unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. It allows interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse. The framework is based on fourteen collection interfaces. It includes implementations of these interfaces, and algorithms to manipulate them [4].

Different interfaces describe different types of groups. For the most part, once you understand interfaces, you understand the framework. While you always need to create specific implementations of interfaces, the access to the actual collection should be restricted to the use of interface methods, thus allowing you to change the underlying data structure, without altering the rest of your code. The following diagrams show the framework interface hierarchy.

When designing software with the Collections Framework, it is useful to remember the following hierarchical relationships of the four basic interfaces of the framework:

- The Collection interface is a group of objects, with duplicates allowed
- The Set extends Collection but forbids duplicates
- The list extends Collection also, allows duplicates and introduces positional indexing
- The map extends neither the Set nor Collection

Fig 1. describes relations between basic interfaces.

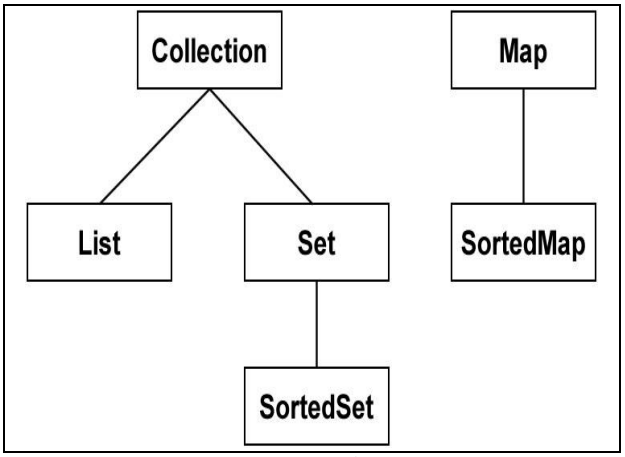

Fig. 1. Collection Framework basic interfaces

Collection framework contains the interface Set with several implementation classes like HashSet and LinkedHashSet (see Table 1). LinkedHashSet is introduced in Java 1.4. The Set is a collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction [4].

Measuring data			
Interface	***Implementation***		
Set	HashSet	LinkedHashSet	TreeSet
List	ArrayList	LinkedList	
Map	HashMap	TreeMap	

Tab. 1. Collection framework interfaces and implementation classes

HashSet is a data structure which is backed by a hash table. It makes no guarantees as to the iteration order of

the set; in particular, it does not guarantee that the order will remain constant over time [6]. LinkedHashSet differs from the HashSet in that it maintains a doubly-linked list running through all of its entries. LinkedHashSet defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order) [7].

The main research focus are implementation classes HashSet and LinkedHashSet and comparison between them. The aim of the experiment was to measure and compare the efficiency of algorithms for checking if the set contains given element, iteration, insertion and removing in/from a set of elements against HashSet and LinkedHashSet.Although the research has reached its aim, there were some unavoidable limitations. First, measurements were carried out on Java SE 7 platform, so there is a chance that result could be slightly different on older Java platforms. Secondly, when measurements were made, it is possible that the processor was busy with other activities, so measured times was not totally correct. For this reason, we made 50000 measurements for each operation to obtain more trustworthy results.

2. MEASURING ALGORITHMS EFFICIENCY

According to [2] an algorithm is a step-by-step well-defined computational procedure for performing some task in a finite amount of time, which takes some values as input, manipulates the data following the prescribed steps and produces some values as output. That information is a collection of data about the actual problem processed by the algorithm, and are organized and accessed in a systematic way in data structures[3]. Algorithm provides the logic or the steps to find the solution; while data provide the needed input values [3]. Programs are executable implementations (in a programming language) of the algorithmic solution of a given problem and the used data structures to store input and output data. Once a correct algorithm is designed, an important step is to determine the efficiency of the algorithm, how much time or space resources the algorithm will require [1].

An algorithm which finds a correct solution only is not sufficient. It might perform long time and be inefficient which becomes more obvious by increasing the size of the input data set. An algorithm that solves a problem but requires a year and/or requires a gigabyte of main memory is not useful [2]. A straightforward way of solving a problem may not be the best one.

There are often more than solutions for a problem. How do we choose between them?Is it one that runs in shortest time or the most efficient one? What is an efficient algorithm? How to measure the efficiency?

A solution is said to be efficient if it solves the problem within the required resource constraints which include the total space available to store the data and the time allowed to perform each subtask or if it requires fewer resources than known alternatives [1]. The mathematical instrument to measure the efficiency (performance) is the analysis of an algorithm. It is used to:

- Estimate resource consumption of an algorithm
- For comparison of relative costs of different algorithms for a given problem.

The choice of the most efficient algorithm among variant solutions is based on the measurements of space complexity (memory resources an algorithm uses) and time complexity (time resources or the time an algorithm runs).

In most cases, only time resources are considered since the techniques used to determine memory requirements are a subset of those used to determine time requirements [1].

The running time becomes an important issue; a natural measure of "goodness," which means computer solutions should run as fast as possible [3]. The running time of an algorithm is affected by many factors:

- The hardware (the processor's speed, the clock rate, memory, disk) and software platform (the operating system, the programming language, the compiler).
- The size of the input data set, etc.

The running time is the most important resource to analyze while considering the size of input data set [2]. To be able to classify some data structures and algorithms as "good," we must have precise ways of analyzing them [3]. There are two ways to estimate or analyze the time complexity of a program:

- Experimental study
- Theoretical study.

The first way to analyze the running time of an algorithm is realized by measuring the time the algorithm runs for differently sized input data sets. Measuring how long the algorithm runs for each input samples with given size, we can visualize the growth rate of the running time (how it increases by increasing the input size).

The second way to estimate the time complexity or the running time T(n) of an algorithm is to determine how many of each primitive instructions (such as arithmetic operations add, multiply, etc., assignment, comparison, invoking a method, etc.,) is executed. Instead of measuring the time execution of primitive operations (each executes in a constant time), assume a one-time unit per operation. We determine the number of primitive operations as a function of the size of the input indicated by n.

This machine model is widely accepted whereas the above factors which influence the running time are not considered. Changing the hardware/ software platform influences the running time T(n) by a constant factor, but does not influence the growth rate of the running time T(n) [3]. That model is a normal computer, in which instructions or primitive operations are executed sequentially and takes exactly one time unit to do each (simple instruction) and infinite memory is assumed [2]. Even though these assumptions about the model could be real problems for computer applications, according to [3]

this is a general way of analyzing the running times of algorithms that:

- Considers all possible inputs.
- Can compare efficiency of any two algorithms independently from the hardware/software platform.
- No need to implement and execute algorithms.

By using this method each algorithm is assigned a function T(n) which determines the dependency of the running time from the input data size n or its growth rate. When analyzing the efficiency of an algorithm, usually the worst case of an algorithm execution is analyzed, when the algorithm performs worst or performs maximum possible number of primitive operations. This methodology is the asymptotic algorithm analysis which determines the running time in big-Oh notation. It gives the upper bound of the growth rate of the running time function.

T(n) is O(f(n)) if T(n) is asymptotically less than or equal to f(n), or lim (T(n) /f(n)) = 0, for n-> . If this function, by increasing the size of the input, has a slow growth rate, then we can say that the algorithm is efficient. Furthermore, if we have to choose among variant algorithms, we' willchoose the one with the running time with the slowest growth rate.

Using theoretical study, algorithms of inserting, deleting and checking if set contains element on HashSet and LinkedHashSet have T(n) = O(1). Iteration on HashSet and LinkedHashSet has T(n) = O(n) [5].

3. MEASUREMENTS

Experimental studies have been done focused on analyzing the efficiency of algorithms for storing and retrieving data respectively: contains, insert, delete and iterate operations on the HashSet and LinkedHashSet. It is realized by measuring the running time for the same input data set for respective algorithms using both implementations. The running time of an algorithm can be observed by running the algorithm on differently sized input data set and storing the values of the time the algorithm starts and ends. Measurements can be taken in a precise way in Java by using the method called System.nanoTime(), registering the start time and end time of the algorithm, performing it 50000 times with same data for both implementations and then finding the difference (finish - start))/50000, which is the time an algorithm runs in milliseconds.

Then in 2D coordinate system, we can visually represent the dependency of the running time (the y coordinate) on the selected set implementation (the x coordinate).

The measured running times are presented in the Table 2 and visualized in Fig 2 to Fig5.

Measuring data		
	HashSet	***LinkedHashSet***
Insert	36,42	39,59
Remove	5,1	9,2
Contains	490	520
Iterate	59	46

Tab. 2. Measuring experiment data

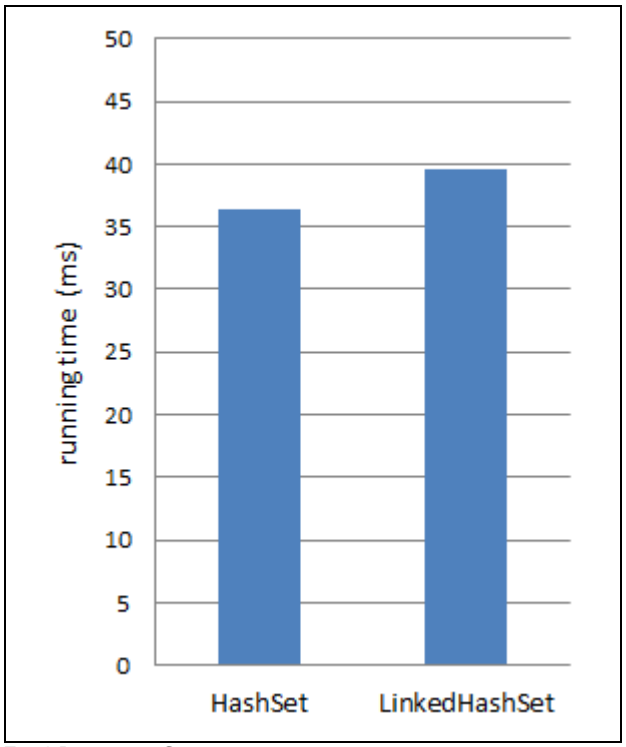

Fig. 2.Insertion in Set

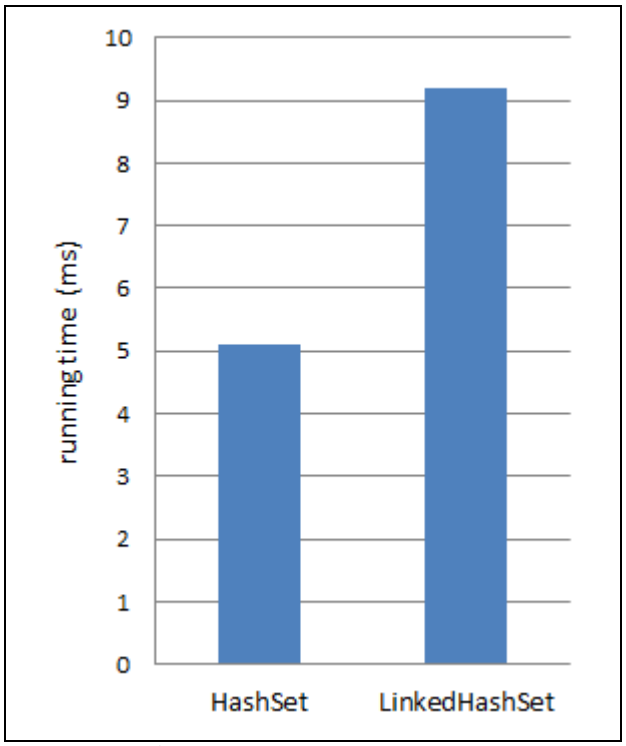

Fig. 3.Removing from Set

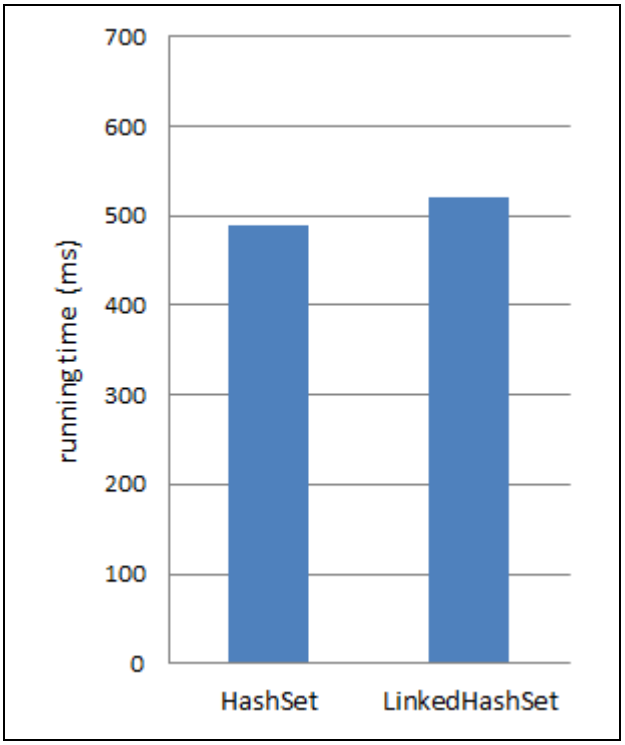

Fig. 4.Set contains element

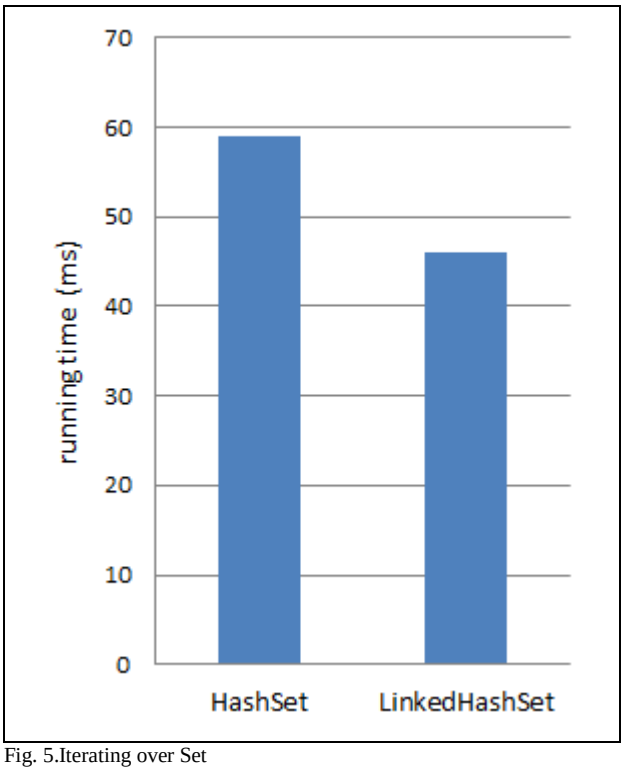

Fig. 5.Iterating over Set

4. CONCLUSION

The experiments show that algorithms of inserting, removing and checking if element is contained in the set are faster over the HashSet than overLinkedHashSet (see Fig.2 – Fig.4), but a difference is relatively small. Iterating over LinkedHashSet is faster than iterating over HashSet, but there is no significant difference (see Fig.5). Conclusion is that the selection of appropriate Set implementation will depend greatly upon how you intend on using it in your production environment. Performance is less a concern in choosing among Set implementation class HashSet and LinkedHashSet. The main concern is the functionality you require. If your only requirement is fast lookup, then you can use the HashSet. If you need to be able to retrieve your items in the same order that you added them to the Set, then you would use the LinkedHashSet without fear of significant performance impact. If you need sorted data, than neither the HashSet nor LinkedHashSet is appropriate; you should consider using class TreeSet.

There are several possible directions for future research.

First, we would like to include class the TreeSet in the empirical study to investigate its impact on performance of algorithms described in this paper. Next step could be an empirical study of Map implementations and correlation between Map implementations classes and Set implementation classes (for example HashSet and HashMap) in order to recommend usage of the appropriate collection. In this paper we mentioned the hash table which is underlying data structure of HashSet, LinkedHashSet, HashMap and some other implementation classes. The main concept of the hash table is hashing function and resolving element collisions. One of our aims in future work is the investigation and empirical study of different hashfunctions used in mentioned implementation classes and possibly new class implementation which uses a hash function with better characteristics than existing hash functions have. Finally, we would like to compare Java implementations of collection classes mentioned here and .NET Framework version of them in order to find possible differences from performance aspect of view.

5. REFERENCES

[1] Shaffer, C. (2011).*A practical introduction to data structures and algorithm analysis*, Prentice Hall, ISBN 13: 978-0136609117, USA

[2] Weiss M. A.(1996). *Data structures and algorithm analysis in C*, Addison Wesley, ISBN-13: 978-0201498400, USA

[3] Goodrich M. T. & Tamassia R (2006). *Data structures and algorithms in Java*, John Wiley & Sons, Inc., ISBN: 9780471738848, USA

[4] http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html (2012) *Accessed on: 2012-07-30*

[5] Shirazi, J (2003). *Java performance tuning*, O'Riley, ISBN:0596003773, USA

[6] http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html*Accessed on: 2012-07-30*

[7] http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html*Accessed on: 2012-07-30*