Analysis of Data Structures in the Java Class Libraries Using Reflection

Matthew Robbins
Second Year, Computer Science
Aberystwyth University
mcr1@aber.ac.uk

ABSTRACT

Built into the java class library are many data structures that one may use to store information for later querying or retrieval. Often, the decision on which structure to use will depend on the speed of the most performed action(s) which can vary dramatically between data structures, especially when using large data sets. Using the Java Reflection API, I have found some evidence to suggest that the time complexity of a data structure may indeed depend on the number of members, but these test were inconclusive and further investigation is needed before any definitive conclusions can be drawn

The number of members in user-implemented data structures (ones not provided in the Java class library) turned out to be much less than in the provided data structures as I hypothesized. I believe this to be because the classes in the Java class library must have been built to cater for a wider variety of tasks due to the large amounts of people and programs that will be using it for different purposes.

Keywords

Data structure, abstract, reflection, package, Java class library, CSV file

1. INTRODUCTION

In the Java class library, there is a collection of Java interfaces and classes that implement fundamental but vital data structures that programmers can use in appropriate situation. These data structures range from lists of elements (Arrays), to lists of linked nodes (Linked list) to structures like a Hash table that stores keys and values.

Due to the diverse nature of the construction of these data structures, the speed for a given task can vary by immense amounts, for example inserting an element could be as easy as tagging it onto the end of a list, or as time consuming as shuffling all the elements along to make room for the new one. This makes choosing the right data structure for a given activity essential, as the fundamental differences between different structures can render some useless.

I am investigating the complexity of the data structures built into Java to determine:

- If there is a link between the complexity of the code of the structures, and its time complexity.
- The difference in code complexity between data structures provided in the Java class library, and ones that users have created.

Primarily, I will be focusing on the most commonly used data structures such as the Hash table, Array List and Linked List – but I will be using other similar structures (for example all that are a sub class of java.util.AbstractList) to get a better idea of each collection and some average values.

I hope to find some distinguishable different between the complexity of the class, that may relate to their big O time complexity.

2. HYPOTHESES

2.1 Data Structures

By analyzing the complexity of the data structures I test, I hope that the data will lend support to a number of hypotheses:

1. The data structures that provide high efficiency for larger data sets (and have a low time complexity) will have a smaller amount of methods and members as I believe their code must have been written very carefully and sparingly. I think this will contribute to how quickly these structures are able to process large amounts of data, as there are no unnecessary instructions and calculations
2. Data structures that possess a worse time complexity will have more methods – methods that provide special functionality that may be essential in some cases – but may slow down the data structure when it is handling vast amounts of data.
3. Data structures not included in the Java class libraries will be smaller, with fewer members in each class, as I believe they will not be as useful to such a wide range of tasks like the Java class library ones must be.

3. METHODOLOGY

To obtain the data I need to analyze the Java classes, I wrote a small Java program that uses the reflection API to first see if a class of a given name exists, and then find information about that class such as what methods, variables and constructors it contains, as well as what is it's superclass, and the package where it is located. Java Reflection is a very powerful tool, and was essential to this project; the ability for Java to introspect upon itself is something many languages do have in such an advanced format.

My program is command line driven, and the user gets a set of options such as:

- Print statistics for a single class
- Read classes from a file and print their stats to the terminal or to a CSV (comma separated value) file.

- Find and print out to a CSV file all the classes that a given classes (or many classes from a text file) refer to.

This allows the user a wide range of freedom, as they are able to use the program as a standalone application, or can have it output CSV files that can be later opened in a spreadsheet application.

I have used this application to generate the data I require to test my hypotheses, adding functionality (such as counting the average number of parameters per method) sometimes where needed.

3.1 Data Structure Collections

I am going to be looking at 4 collections of data structures in particular, grouped by which interface they extend:

- AbstractList (Vector, ArrayList, LinkedList, Stack…)
- AbstractMap (HashMap, EnumMap…)
- AbstractQueue (PriorityQueue, DelayQueue…)
- Dictionary (Hashtable)

These collections contain most of the more interesting and commonly used data structures provided in the Java class library. Other structures such as heaps and trees would be interesting to examine, but I would first have to write and implement them – bringing may in coding skills as another variable that may well affect the end results.

The statistics that I need will be obtained by running each collection of names through my application by sending them in as a text file. My program will then find and generate the statistics for each class and store them in a hash table until they are all ready to be written to a CSV file for easy analysis.

3.2 Scope and Limitations

As stated above, I am just going to me exploring the 4 (main) collections of data structures that are implemented in the Java class library. The statistics that I am going to be generating for each class will include: the total amount of public methods, parameters, fields, members and all methods. I will also be obtaining the average number of parameters for all of the methods in that class.

The main limitations in providing good and accurate data for this report are the time and skills available to me, as I do not have an abundance of time, and I have never examined the Java Reflection API before now.

4. RESULTS

I started off by getting the basic statistics of each class by putting a text file with a collection of classes through my application. This returned CSV files with the following data:

Table 1. AbstractList's output

Class	Public Methods	Parameters	Total Methods	Total Members	Total fields	Avg. Params
Vector	41	40	44	44	0	0
ArrayList	19	22	24	24	0	0
AbstractList	15	17	16	16	0	1
Stack	5	2	5	5	0	0
LinkedList	39	37	48	48	0	0
AbstractSequentialLi	7	9	7	7	0	1
Sum	126	127	144	144	0	2
Average	21	21.1666667	24	24	0	0.33333333

I added an "Average" field to allow easier analysis of the data, and then plotted it in a scatter graph (Figure 1).

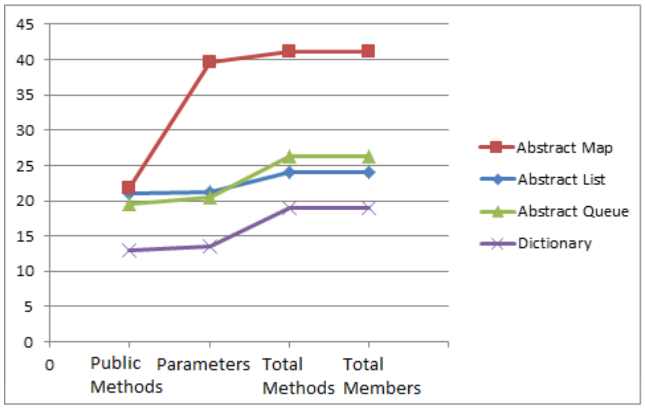
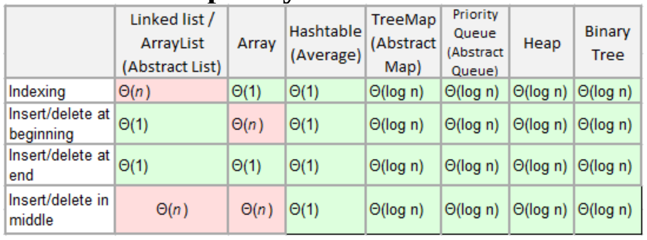


Figure 1. Quantity of each statistic for the 4 different data structure collections

Figure 1 quite definitively shows that the Dictionary collection, comprising of just itself and Hashtable, has the least amount of members out of all of the collections. The AbstractMap collection (containing TreeMap, HashMap, EnumMap etc) has the highest number of members by quite a way, and assuming my hypothesis is true, it will mean that the AbstractMap structures will be quite slow, and the Dictionary structure – Hashtable to be faster in comparison. It also states that AbstractList and AbstractQueue will be slower than the Dictionary structure.

The shape of the graph shows a fairly consistent correlation between a class and the different type of statistics that I have collected about it, partly because some values (such as total methods) rely on others (public methods).

4.1 Time Complexity



	Linked list / ArrayList (Abstract List)	Array	Hashtable (Average)	TreeMap (Abstract Map)	Priority Queue (Abstract Queue)	Heap	Binary Tree
Indexing	Θ(*n*)	Θ(1)	Θ(1)	Θ(log n)	Θ(log n)	Θ(log n)	Θ(log n)
Insert/delete at beginning	Θ(1)	Θ(*n*)	Θ(1)	Θ(log n)	Θ(log n)	Θ(log n)	Θ(log n)
Insert/delete at end	Θ(1)	Θ(1)	Θ(1)	Θ(log n)	Θ(log n)	Θ(log n)	Θ(log n)
Insert/delete in middle	Θ(*n*)	Θ(*n*)	Θ(1)	Θ(log n)	Θ(log n)	Θ(log n)	Θ(log n)

Figure 2. The time complexity values for some data structures

Figure 2 shows the time complexity values for a range of data structures, with at least one from each collection that I am examining.

4.2 Comparing Statistics with Big O

My hypothesis was that the data structures that were generally quicker with large data sets would have fewer members than the others. The data so far supports that statement, as the Hashtable's statistics were the lowest, and its time complexity is better than the rest, as O(1) means that that operation will take a constant time no matter how big the data set is.

My data shows that AbstractMap structures have the most number of members by a considerable amount – almost twice as much as others. The time complexities of structures that extend it are worse than that of the Hashtable, but are not the worst of all, as I was expecting. Their time complexity of O(log n) is relatively good, even for large data sets, but it still grows as the data size increases.

Data structures like Priority Queue that are a subclass of AbstractQueue also have a time complexity of O(log n). The AbstractList structures: LinkedList and ArrayList are both quicker to insert at the beginning/end, but slower to index and insert in the middle than the AbstractQueue structure. This divide may be represented in the graph by each type having 2 higher values than the other. Whilst the data so far seems to suggest this, it is currently inconclusive and more testing would be needed before any definitive conclusions could be drawn.

4.3 Data Structures Not implemented in the Java Class Library

Included in the time complexity chart are the values for “Heap” and “Binary tree” so they can be compared and contrasted with the other values. I was surprised to see that they shared the same time complexity as the structures in AbstractMap and AbstractQueue, and contrary to my hypothesis, imagined that that Heap and Binary Tree may have a similar number of members.

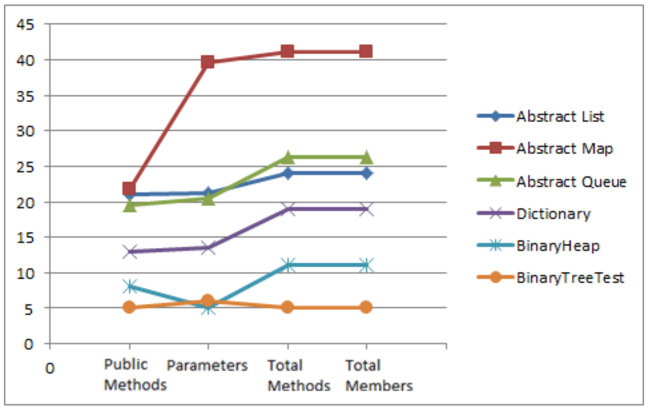


Figure 3. Quantity of Statistics With Binary Heap and Tree

As Figure 3 shows, the Binary Heap and Binary Tree implementations I found online had much less members than the others, proving part 3 of my hypothesis correct. I believe there are more members in the Java class library classes, as they must be very resilient and versatile because vast numbers of people are going to be using them for a huge range of activities. The user created classes can afford to contain fewer features, and therefore not include so many methods and other members.

5. CONCLUSIONS

5.1 Class Complexity vs Time Complexity

My hypotheses enjoyed varying degrees of success, the first two parts about the correlation between complexity of the data structure class and its time complexity was initially supported by the evidence, as the Hashtable had the least amount of members, and was the quickest to access.

This hypothesis was then contradicted by other evidence showing other data structures having varying degrees of complexity with no relation to the speed of access. This experiment was inconclusive, and may warrant further investigation at some point to examine if there is any definite relation.

5.2 Data Structure Classes Will Be Smaller Outside the Java Class Library

Within the limited scope of my tests, I found evidence that supported my hypothesis that classes made by users would not contain such a large amount of members as standard Java files in the Java class library. I believed this would be the case as the Java classes must be built to be very versatile and implement all of the functionality a programmer may need from it, resulting in more code and more members.

Further experiments could go on to test to see if other unimplemented data structures and other classes are still less complex than the in-built classes.

6. REFERENCES

Wikipedia article on ArrayList time complexity: http://en.wikipedia.org/wiki/Arraylist REF

Hexopedia - Article on data structure time complexities: http://essays.hexapodia.net/datastructures/

java-tips.org – Binary Heap Implementation - Mark Allen Weiss: http://www.java-tips.org/java-se-tips/java.lang/priorityqueue-binary-heap-implementation-in-3.html

java2s.com – Binary Tree Implementation: http://www.java2s.com/Code/Java/Collections-Data-Structure/BinaryTree.htm