Using Real Execution Timings to Enliven a Data Structures Course

B. Boothe
Computer Science Department, University of Southern Maine, Portland, ME, USA

Abstract - *Which is faster, an array list or a linked list? In our data structures course we increase the engagement of our students by asking them to predict the outcome of a future race between competing data structures. We record their predictions for future validation, and subsequently as we analyze the data structures, students are more engaged in the analysis because it either lends credence or doubt towards the accuracy of their predictions. Some of their expectations get radically broken by the analysis, and in the end everyone gets surprised by the actual runtime results. Everyone benefits because the class is more engaging and the results are more memorable.*

Keywords: Real Execution Timings, Data Structures

1 The Starting Line

At the University of Southern Maine our initial programming curriculum is taught in Java. Our Data Structures course is the third course in our sequence of programming courses. In our second course, students use data structures such as Java's ArrayList and LinkedList and learn a bit about the internal mechanisms of how they work. They know that array lists have to periodically grow the internal array by creating a new one and copying everything over, and they know that it is painfully slow to insert at the start of an array list and move everything up. They have also been convinced linked lists are slick because of how efficiently they can insert new values between others without needing to copy anything. Accurate or not, these are their typical preconceived notions coming into data structures.

The first day of class I propose that we use the online ordered collection problem as a basis for analyzing and comparing the variety of data structures that we will encounter in this class. The online ordered collection problem is to take N items and insert them into the data structure while maintaining the data structure in sorted order at every insert. The online aspect refers to the possibility of using this data structure to perform a search at any point while it is being built. Since building the data structure involves searching for the positions to insert new members, this is a concise benchmark that incorporates both the cost of creating and growing the data structure as well as the cost of searching it.

After I have introduced the ordered collection benchmark, I then have students participate in a small group activity and make their predictions on which will be faster, array lists or linked lists. I pose this as a race between the best real programs we can write, running on a real computer. Which data structure will win?

I am not asking them to perform an order analysis, but simply to discuss it in their groups and use their intuition about these two data structures to predict the relative runtimes. Will one data structure be a little faster, a lot faster, or about the same as the other? The reader might pause to consider this question themselves.

I have used this activity for several years. The majority of groups always predict that linked lists will be either *a little faster* or *a lot faster* than array lists. For justifications they cite the high cost of inserting into the array and the cost of repeatedly growing the array to make it larger. In contrast, usually at least one group notes that the sorted array can be binary searched much faster than the linear search required of the linked list, and thus they predict that the array list will be faster. Finally, there is usually another group that remembers and utilizes order analysis to predict that both data structures will perform about the same. I do a lot of group activities and discussions in my class, and this disparity in opinion is perfect. All the better to grab their attention and later dispel their misconceptions.

With all of their predictions and justifications recorded on a transparency, I tuck this away to be brought out in a couple weeks after we have thoroughly studied and analyzed both of these data structures. Students are always disgruntled to learn that they will have to wait a couple weeks before they see the outcome of the race. They are eager to know the winner and whether their predictions are correct. Like a good writer, I know the value of foreshadowing.

2 Handicapping

Over the next couple weeks we discuss and analyze the implementation choices and asymptotic analyses of Java's ArrayList and LinkedList. The final analyses for building the ordered collection are shown in Tables 1 & 2. Overall both data structures have an expected execution time of $O(N^2)$.

Once they have seen that both the ArrayList and LinkedList will be $O(N^2)$, I give the students a chance to revise their opinions. Most students, having placed their bets on the LinkedList, stick with it beating the ArrayList, although maybe not by such a wide margin as some had predicted.

Table 1. Using an ArrayList to build an ordered collection.

building and growing	$O(N)$
searching	$O(N \log N)$
inserting	$O(N^2)$
total:	$O(N^2)$

Table 2. Using a LinkedList to build an ordered collection.

building	$O(N)$
searching	$O(N^2)$
inserting	$O(N)$
total:	$O(N^2)$

In addition to the traditional runtime analysis, I also do memory usage analysis of these data structures. For analyzing memory usage we simply count the array locations and/or visible object fields. These are summarized in Table 3. These memory footprints will play a role in explaining the real execution time results that the class will be seeing soon.

Table 3. Memory Footprints (as explained to students)

ArrayList	1N active, plus up to N/2 reserved for future growth
LinkedList	3N (data reference, previous node and next node references)

For memory usage they have seen that the object references stored in Java's array list are compactly stored in N adjacent memory locations, with up to an additional N/2 unused memory locations allocated as space for future growth.

For linked lists they have seen that each node in a doubly linked list has a data object reference as well as references to the next and previous nodes. This means that its memory footprint is 3N to hold N data objects.

At this point student attitudes are generally not overly interested in memory footprints since modern computers have enormous memories. In fact, in earlier classes I sometimes encourage students to be profligate with memory, if it can simplify their algorithms.

Over the duration of the semester we build a summary table of the data structures as we study them. For each new data structure, we note its key strengths and weaknesses, and we record its memory footprint and our analysis of its runtime on the online ordered collection benchmark (if appropriate). This table serves as a backbone for the class. As each new data structure is studied, we can quickly see its benefits and tradeoffs compared to the previously studied data structures.

3 The Race

Everyone takes pride in making a fast program, and thus students are always interested and eager to see the results from the race. The results are shown in Figure 1. This is a log-log plot of the execution times versus the problem sizes for 4 different data structures. At this point in the class, however, I show them a simpler version of this graph with only the performance curves for the ArrayList and the LinkedList. The lines for the ChunkList and TreeSet are added later as we analyze these data structures. I present only the single graph in this paper to save space.

These execution times are from a 2.4 GHz Intel Core 2 Duo with a 2MB L2 cache and 3.25 GB of RAM running Java 6. The testing code was written to be as efficient as possible. For array lists I used binary search to find the insertion point. For linked lists I used iterators. The objects inserted into the data structures are random Integer objects so as to minimize the amount of time spent comparing the objects and thus focusing on the time spent by the data structure manipulations themselves.

Students generally are quite startled to see the 60 times difference between the performances of these two data structures, especially since the majority had bet on the loser of the race! When I first created this activity I was myself surprised by the magnitude of the difference. I had expected the array list to win, but not by such a large margin.

4 Explanation

Why has our order analysis led us astray? Our expectation was that they were both $O(N^2)$, and thus they would perform similarly. Order analysis ignores constant factors, but a factor of 60 is startlingly large! It even appears from the increasing gap as the problem size grows, that measured in real time the LinkedList may in fact be of a higher order than the ArrayList.

I offer the class two explanations for the performance difference. I have not tried to quantify the importance of one factor over the other.

1. **A method call versus a memory access.** Focus on the $O(N^2)$ factors. For the linked list this was the searching time. Each insertion must linearly search down the list to find the insertion point. In total this

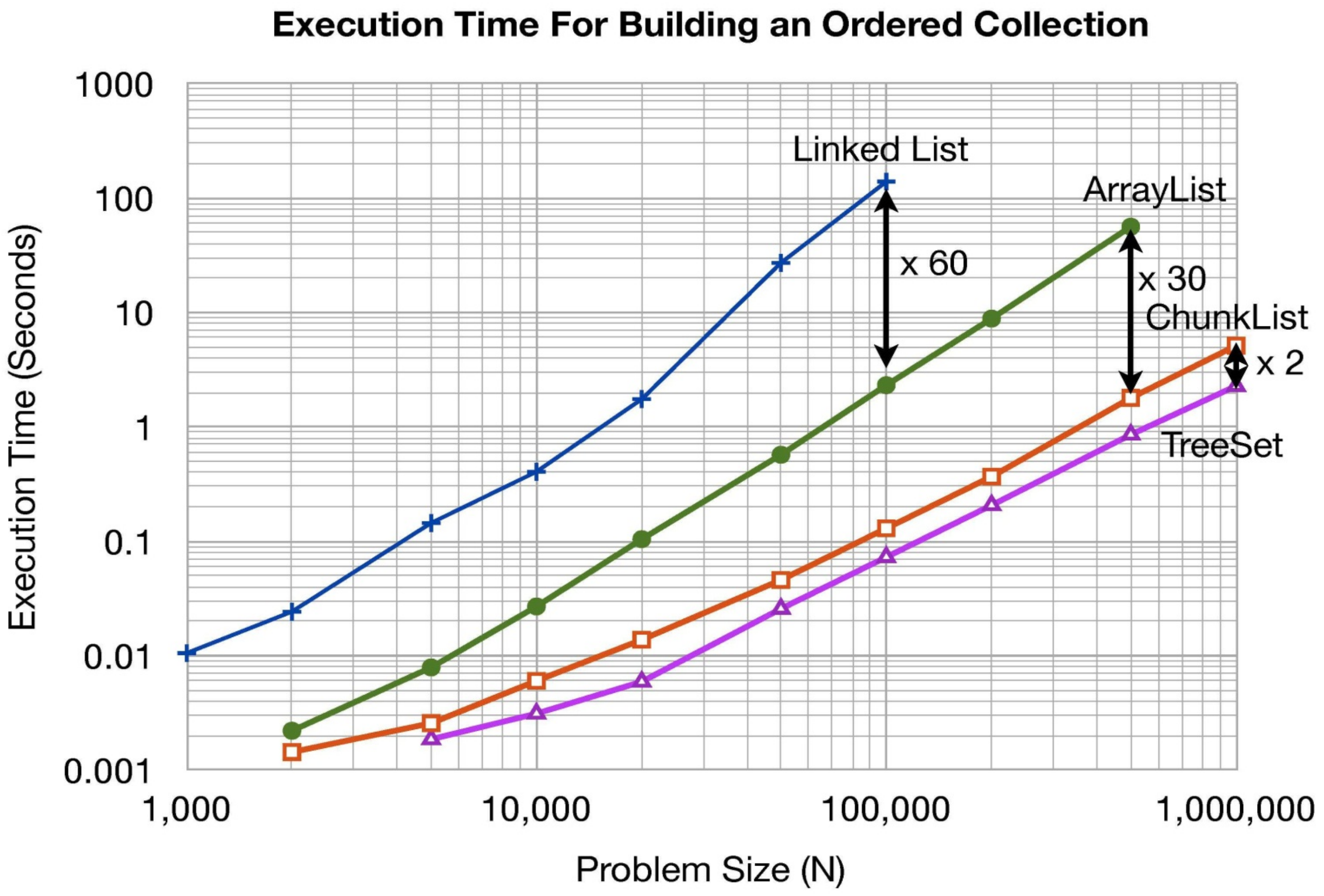

Figure 1. Execution times for building an ordered collection for 4 data structures.

involves $O(N^2)$ method calls to the comparator. In contrast, for the array list, the $O(N^2)$ factor was the insertion time incurred as each item was inserted somewhere into the array and everything in the array above the insertion point was moved up to make room for the new item. This involves N calls to System.arraycopy(), with each call moving $O(N)$ memory locations for a total of $O(N^2)$ memory locations moved. The difference is that arraycopy() is a tuned system routine very efficiently moving a contiguous block of memory [5]. When compared to the $O(N^2)$ individual comparator calls for the linked list, this is a big speed advantage for the array list.

2. **Memory Footprint.** In modern computers the time for a level 1 cache hit is more than 100 times faster than access to actual memory [1,2]. As the memory footprint of the data structure grows, more memory accesses are pushed to lower and slower levels of the memory hierarchy. In class we analyzed only the user visible fields and came up with an active memory footprint of 3 times larger for Java's LinkedList than its ArrayList. This means that the linked list is being pushed into lower levels of the memory hierarchy before the array list is. Furthermore the array copies are nice sequential accesses that work well with caches and pre-fetching compared to the linked list accesses which are jumping all over memory. We expect that this contributes to the increasing performance gap between the two data structures as the problem size gets larger. (At this point in class we do not complicate the picture by pointing out that the true memory footprint for Java's LinkedList is actually 7 times larger than that of an ArrayList due to hidden fields used in the representation of objects and the indirection table used for incremental garbage collection. [6])

Considering these two explanations, the factor of 60 difference in performance is more plausible, and those students who care strongly about fast execution times suddenly grant their old professor a little more respect when they realize he might be able to teach them more about the Zen of programming than they had expected.

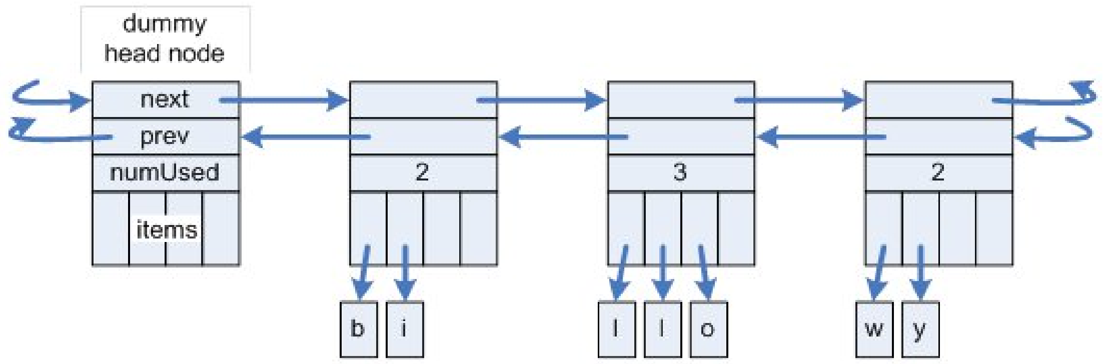

Figure 2. A ChunkLlist. A hybrid data structure that is a linked list of arrays. It is used to store an ordered list.

5 Hybrid Data Structures

Back in my days as a student, programming in C, we often needed to build a variety of simple data structures. These days programming languages provide excellent libraries of well-designed generic data structures. Data structures course have thus transformed into using these data structures and understanding their performance implications [4,7]. I, nevertheless, strongly feel that students ought to experience creating their own data structure from scratch. At some point in their careers they are likely to work on a complex software project that warrants creation of a new or custom data structure. I address this by having one assignment in which students implement a hybrid data structure, something incorporating aspects of two traditional data structures.

Several years ago I invented, for pedagogical purposes, a hybrid data structure that I called a ChunkList. A small example is shown in Figure 2. (I am not the first person to explore this hybrid data structure [3].) It is a linked list of nodes, but with each node containing an array of keys. When a node becomes full, it is split into 2 nodes. The purpose of this ChunkList is to implement an ordered list. Its add() method inserts a key in sorted order into the collection of keys. It is thus exactly what we need to implement the online ordered collection. Later students will see that its interface is in fact a subset of Java's TreeSet interface.

It is presented to students as a hybrid of a linked list and an array list having the best of both worlds. It has quicker searching than a linked list by jumping a chunk at a time through the list, but it also has quicker insertion than an array list by only inserting into an individual chunk, rather than inserting into a huge monolithic array. If chunks each hold 100 keys, then we have the possibility of searching 100 times faster than with a simple linked list. (All of that knowledge of linked lists has not gone to waste after all!)

One of my motivations for using a hybrid data structure is that having something more unique than what is found in textbooks decreases the chance that the less honest students will be able to find a pre-existing implementation to copy from. A quick Google search for ChunkList finds matches for the name, but I examined the top ranked matches and currently they provide nothing useful for the dishonest student who is trolling for a working program online. I would encourage anyone using this idea to think of a variation on the name. I have also seen this called an unrolled-linked list[3]. We might also call it a list of arrays or a blocked list.

Students are asked to implement this with a tunable chunk size and test it out with chunk sizes of 10 and 100. For extra credit students are asked to determine the optimal chunk size. I perform this analysis in class on the day they turn it in. It is an excellent opportunity to use a little calculus to minimize the execution time formula and demonstrate that all those math requirements that they grumble about weren't entirely immaterial to computer science. The analysis shows that the optimal chunk size is $\sqrt{N}$ and that the overall runtime for the online ordered collection problem using a ChunkList will then be $O(N\sqrt{N})$.

I now present to the class the earlier performance graph with the ChunkList added on. (In this paper the chunk list was already shown in Figure 1.) My ChunkList is a more sophisticated implementation than that of the students. It grows the chunk size in proportion to $\sqrt{N}$. As the total list size grows, so does the chunk size. From the performance graph this is clearly much better than the $O(N^2)$ of the ArrayList. The real execution time is 30 times faster than an array list, as the problem size reaches the maximum comparable size shown.

6 The Shoo-in

The eventual winner of course will be the binary search tree. Maintaining an ordered and quickly searchable collection is exactly what binary trees are intended for. Java's TreeSet is implemented with red black trees. It is the last line plotted in Figure 1. Its time for building the online ordered collection is $O(N \log N)$ which is asymptotically superior to the ChunkList's $O(N\sqrt{N})$.

The graph surprisingly shows only a factor of 2 advantage in real execution time of the TreeSet over the ChunkList. I was again surprised when I saw the actual performance data. At a size of N=1,000,000, the last data points shown on the graph, one might expect a performance difference of a factor of 50 for a $O(N \log N)$ algorithm compared to one that takes $O(N\sqrt{N})$. This is hard to explain. The contributing factors are:

1. **Red-black trees are complex.** Undoubtedly some of that factor of 50 is used up by the frequent internal rotations used in maintaining balance within the red black tree.

2. **ChunkList tuning.** The chunk list was tuned for optimal execution time. Through empirical tests I found that a chunk sizes of $13\sqrt{N}$ gave the fastest execution times. This larger chunk size gives more weighting to the faster array copy used in inserting into a chunk over the slower search of the linked list of chunks.

3. **TreeSet has a richer implementation.** For example the TreeSet provides both the Comparator and Comparable interfaces, whereas I asked the students only to implement the Comparator interface. Every option adds complexity and slows things down a tad.

4. **Java's TreeSet is just a wrapper for its TreeMap class.** In order to reuse code, Java's TreeSet is in fact a TreeMap with its data fields ignored. Every call to a TreeSet method is in reality an indirect call to a TreeMap method.

5. **Memory footprint.** Again the memory footprint size undoubtedly plays a role. The active memory footprint for the ChunkList is slightly over 1N, but the visible memory footprint for a TreeSet is 6N and the true footprint is 9N. A larger memory footprint pushes it into lower and slower levels in the memory hierarchy.

7 Conclusions

I have found using real execution timings and analysis of memory usage to be valuable tools in teaching data structures and for increasing student interest levels in the comparative analysis of these data structures. Students have great respect for real performance data and seem to trust it more than our mathematical analyses. Combining real execution times along with order analysis, and explaining the reasons for the discrepancies, has increased understanding of the value of asymptotic analysis and its limitations.

8 References

[1] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus, "Cache-Conscious Structure Layout," In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI),* May 1999.

[2] David A. Patterson and John L. Hennessy, "*Computer Organization and Design, The Hardware / Software Interface, 4th ed.",* Morgan Kaufmann 2009.

[3] "Unrolled Linked List", *http://end.wikipedia/wiki/Unrolld_linked_list*

[4] Elliot B. Koffman and Paul A. T. Wolfgang, "Data Structures, Abstraction and Design Using Java, 2nd ed.", Wiley 2010.

[5] Oracle Sun Developer Network, "JAVA SE 6 Performance White Paper", http://java.sun.com/performance/reference/whitepapers/6_performance.html

[6] Neil Coffey, "Memory usage in Java", http://www.javamex.com/tutorials/memory.

[7] R. Lister, I. Box, B. Morrison, J. Tenenberg, and D. S. Westbrook "*The Dimensions of Variation in the Teaching of Data Structures",* In Proceedings of the 9th annual SIGCSE conference on Innovation and Technology in computer science education (pp. 92-96)