

Hybrid and Custom Data Structures: Evolution of the Data Structures Course

Daniel J. Ernst, Daniel E. Stevenson, and Paul Wagner

Department of Computer Science
University of Wisconsin – Eau Claire
Eau Claire, Wisconsin, 54702-4004, USA
{ernstdj, stevende, wagnerpj}@uwec.edu

ABSTRACT

The topic of data structures has historically been taught with two major focuses: first, the basic definition and implementation of a small set of basic data structures (e.g. list, stack, queue, tree, graph), and second, the usage of these basic data structures as provided by a data structures framework in solving larger application problems. We see a further evolution of data structures to include new generations of hybrid and custom data structures, implying that our students must not only understand how to use these new data structures but that they continue to understand low-level implementation issues so that they can develop the next generation of data structures needed in the future. We suggest that the data structures course evolve to reflect these new generations of data structures.

Categories and Subject Descriptors

E.1 [Data Structures]; K.3.2 [Computer and Information Science Education]

General Terms: Algorithms, Performance, Design

Keywords: Data Structures

1. INTRODUCTION

The content of data structures courses have changed over the last twenty years. The first major generation of this course focused on describing the basic data structures (e.g. list, stack, queue), implementing these data structures in various ways (e.g. array-based structures or pointer-based structures), and then implementing some simple examples using these structures. The second generation of this course focused on using these basic data structures to solve meaningful problems, either implementing the data structures or using them from libraries such as the Standard Template Library (STL) [11] for C++, the Java Collections Framework (JCF) [6], or the Microsoft .NET Framework [7].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ITiCSE '09, July 6-9, 2009, Paris, France.

Copyright 2009 ACM 978-1-60558-381-5/09/07...\$5.00.

In the last decade, data sets have grown far larger and much more complex. These new data sets, spawned from research areas like biochemistry to customer information data from the World-Wide Web, often encompass a massive scope – an entire chromosome set, or the entirety of Wikipedia. We now see an additional generation of data structures instruction evolving, which adds *hybrid* data structures to the mix in order to properly handle these types of problems.

Our focus in expanding the pool of relevant structures is primarily on usage rather than implementation, given the quick inclusion of these data structures in mainstream frameworks. However, as we consider the new data structures that are significant, we find evidence that we should still be teaching implementation techniques, as software developers are continually confronted with new types of data and access needs for that data, which in turn leads to a need to develop new data structures. Computer science students should be able to customize and implement new data structures to deal with the complexity of new data sets. This evolution of data and data structures affects how we teach the data structures course, and how we strike a balance between time spent on implementation and usage of data structures.

2. BACKGROUND – HISTORICAL AND CURRENT APPROACHES

2.1 First Generation – Focus on Implementation of Basic Structures

The original approach to teaching Data Structures focused on basic data structures and their implementation. Before the existence of frameworks and libraries of canned data structures, students often needed to create their own set of data structures to use in later classes. Thus, a small set of foundational data structures was usually implemented in this course. These included basic array-based lists, various linked lists, stacks, queues, and binary trees. Examples of textbooks following this approach include [2][14]. Applications of these data structures are certainly discussed under this approach, but these were often limited to a small number of examples and/or programming exercises.

In these courses, significant time was spent on details of the inner workings of each data structure. Mid-level concepts, like the runtime impact of various operations on array-based verses link-based lists, were still covered. However, since a full

implementation of each structure was necessary, time had to be spent on the low-level details of each operation.

Additionally, before the rise of object-oriented programming, students would be taught to create abstract data types (ADTs) for these data structures. Without some form of abstraction, one would have to re-implement project specific versions of these data structures. Even with the concept of ADTs, it was sometimes necessary to dig into the details of old implementations in order to modify them slightly for any new needs.

Also, most algorithms that operate on data structures, such as sorting and traversal, were often not packaged with data structures. Therefore, it often required knowledge of the low-level implementation of each structure to properly write these algorithms.

2.2 Second Generation – Focus on Usage of Basic Structures Through Frameworks

With the rise of object-oriented programming and templates/generics, the creation of truly re-useable data structures became a reality. Around this time, the second generation approach to teaching data structures began to emerge.

By using frameworks, such as C++'s STL and Sun's Java JCF, students were able to leverage the considerable time and effort that had been put into creating a re-usable set of core data structures that could store different types of elements. Because of this, the second generation of teaching data structures consists more of instruction on how to use the various data structures rather than on their implementation. Since an implementation of the standard data structures has already been created and tested by others, students can focus more of their time on figuring out how to use these frameworks properly. Examples of textbooks following this approach include [3][4]. This approach has been suggested in previous literature as well [12].

Finally, many of these frameworks include sets of algorithms, such as sorting and traversal, which work on their data structures. This is particularly useful if one needs to perform any of these common operations as few details of the underlying data structures need to be known.

2.3 Early Generation Benefits and Drawbacks

Generations one and two take significantly different approaches to teaching data structures. In this section we analyze the benefits and drawbacks of each generation.

2.3.1 First Generation

It is easy to identify the problems of the first generation. By having to implement all the data structures by hand, one needs to focus on every little detail of each. The big picture of data structure usage is sometimes shortchanged, leaving less time to do more interesting high-level assignments based on data structures.

Additionally, the low-level details can drive home the early illusion that computer science is all about low-level programming. This can lead to a loss of potential majors who might likely enjoy later high-level topics.

While the drawbacks are fairly obvious, the benefits are less so. Certainly one gets lots of practice at developing programming skills by implementing the details of data structures. One can

argue that programming skills can be gained by implementing programs that use data structures as well. However, there are a few key areas where there is a significant advantage gained by low-level implementation.

The first is in pointer manipulation. While this isn't as big an issue for Java or C# based data structure classes, it is for a C++ based classes. There are lots of interesting issues dealing with pointer manipulation, passing/returning by reference, operator overloading, and making copies that are easily demonstrated by implementing data structures. This course material is likely the best test bed for such C++ issues.

A second advantage of the first generation comes in terms of testing. If one wants to emphasize unit testing in the introductory sequence then implementing data structures is a good match. For example, in a linked list one often has to have code to implement the special cases that occur when performing operations at the front, back, and middle of the list. These are just extra test scenarios that need to be written by the students. One can also give out some test cases, which forces the students to consider implementation of all cases. Forcing students to systematically think through all the scenarios that could occur is useful for the development of testing skills.

2.3.2 Second Generation

The benefits of the second generation are the more obvious of the two. For the last number of years, industry has seen widespread use of frameworks, such as the STL and JCF, to speed up their development. Teaching students to use packages that are used in industry is certainly an advantage.

Since there is less time spent on low-level implementation details, more time can be spent on interesting uses for data structures. This can lead to more interesting high-level assignments for students, as well as the opportunity for inclusion of cutting edge technologies.

The drawbacks of this approach can be harder to measure. Students are often taught a specific framework or library of data structures. They may only learn how to use that single library rather than gain a more general understanding of data structures. Related to this is the fact that since a framework only contains a particular set of data structures, students may come away with the idea that those are the only data structures available for them to use for any future project they encounter.

A more obvious disadvantage is that often these libraries contain multiple implementations of data structures that differ in subtle ways. These implementations may all adhere to the same interface and thus are interchangeable in the client code. However, the reason there are multiple implementations is almost always for performance reasons. Selection of the proper data structure implementation for a particular problem is not an easy task. One needs to have a firm understanding of the model the implementation is using. These models can be taught, but it is often harder to drive home the point without having students looking at the details.

2.3.3 Summary – First Two Generations

As with other generational evolutions, the more recent generation in data structures pedagogy is not in every way an improvement on the earlier generation. Some instructors, as well as many students, tend to overlook the advantages of the first generation

and the disadvantages of the second generation. The important point to remember here is that the second generation focuses on different benefits than the first generation, and neither is necessarily better in all ways.

3. EVOLUTION – NEW APPROACHES

3.1 Increase in Data Complexity

As was stated earlier, modern, real-world data sets have grown far larger and much more complex than the examples often used to motivate the basic data-structures used in the first and second generations of coursework. Beyond simply a large size, modern data sets often need to be traversed and examined by multiple access methods. The resulting heterogeneous access patterns for data structures are not easily handled by the standard structures that most students are exposed to.

For an example of these more complex relationships, we can look at the plethora of medical and biological data produced in the last few years. Protein sequence information has often been stored in a flat file, but researchers now want the ability to access the data quickly over multiple different parameters, such as clusters of sequences or taxonomy.

Data access patterns face another emerging complexity: the increased usage of multi-threading in applications as a result of the advances in multi-core processors. This added complexity introduces a new set of access patterns, simultaneous accesses, to the data structure requirements. Understanding how threading affects a data structure requires direct knowledge of the underlying algorithms. This is particularly true if a developer wants to get good performance from thread-safe structures.

3.2 Hybrid Data Structures

To handle larger and more complex data, many languages now provide more complex hybrid data structures. These structures do not consist of just “layered” simple structures (a stack of queues, etc.), but of a more nuanced internal combination.

A linked hash map, for example, consists of a standard hashmap where the map nodes have been connected by a second set of references in a linked-list ordering. This secondary ordering allows for an iterator to access the data elements in a secondary ordering (such as insertion ordering – the default for Java’s `LinkedHashMap`), while still allowing fast hash map access times for individual elements. Implementations of these hybrid structures are becoming common within the mainstream frameworks, including the JCF and the STL, among others.

A hybrid data structure may also consist of a basic structure that has been given some extra capability to handle a different type of abnormal usage pattern, such as for thread-safety or for data sets that do not fit in memory [13]. Frameworks that support this type of application are also becoming more common. Java includes many higher-level parallel structures in their `java.util.concurrent` library (such as blocking queues) and the new Parallel mode STL [10] is now a part of the standard C++ libraries. Another independent library that supports data structures and parallelism is the Parallel Java library [9].

3.3 Hybrid Examples in our Curriculum

We have started to integrate these hybrid structures into our department’s data structures coursework. What follows are a couple of simple examples of how these topics are being

presented to students and the problems that they are asked to solve within this context.

3.3.1 Implementing LRU

The concept of a collection’s Least Recently Used (LRU) item is a very simple one for students to grasp. However, when it comes to implementing this as a policy (for, say, eviction or replacement in a cache) students often struggle with the issue of having two separate goals in play: a basic hashing structure to hold the data but needing to maintain a separate ordering that is not based on the data values themselves. Most students end up with bulky and inefficient implementations, often including extra data fields that must be updated over all elements after each access, or completely replicating the data in a separate list structure.

It turns out that the hybrid *linked hash map* structure is a perfect fit for this problem. The linked hash map is a typical hash map (a hash table for which each table entry is chained using a map), where the map’s items are also linked together in an ordered list. This list ordering often defaults (such as in Java’s `LinkedHashMap`) to being determined by the insertion ordering of the items; however, it is also often possible to order the list based on access order.

With a linked hash map set to keep access ordering, accessing the cache takes the same amount of time (plus a small constant for the accessed item to be re-ordered to the tail of the list) while removing the oldest item is as simple as deleting the item that is at the head of the linked list sub-structure

The students learn that by cleverly merging data structures, they can avoid any duplication of data and keep access and maintenance times to a minimum. In addition, it is now quite easy to elegantly modify an eviction/replacement algorithm – all one needs to do is to modify the list’s ordering parameters.

3.3.2 Exposure to Multi-Threaded Structures

In our C++ data structures class, we have started spending a small portion of our time working with OpenMP [8], a pragma based automated parallelism API that is currently included in most major compilers for C++ or Fortran. OpenMP makes extracting basic parallelism very easy, as students don’t have to work with lower-level threading at all – the compiler takes care of the creation and balancing of threads on its own.

However, OpenMP’s simplicity also makes it extremely easy for students to create multi-threaded race conditions within their code. Students often trip up by making the deceptive assumption that multiple threads can access a data structure (such as an STL List) at the same time with no side-effects. This misconception can easily (and quite enjoyably) be dispelled in class by going through a simple two-thread example where a linked list’s pointers end up in various degrees of disarray.

In both the classroom and the laboratory, we pose this issue to students and work with them to craft solutions. This problem not only teaches them to be aware of these dangerous multi-threading issues (basic solution: lock the structure on any modifying access), but also can be used to illustrate how more intelligent (and more internal) solutions can be used to avoid excessive synchronization and to enhance performance.

4. DISCUSSION / EVALUATION

4.1 A Model for Data Structures Evolution

We have previously considered the evolution of our data structures pedagogy to be linear – that is, that we’ve moved from a state of primarily teaching implementation of basic data structures to a “higher” state where we were primarily teaching usage of these basic structures. However, that model has to be called into question based on our observations when working to teach about the newer hybrid data structures as well. We now think of our data structures pedagogy as being based on a sequential, but multi-level, model – a “sawtooth” model, so to speak.

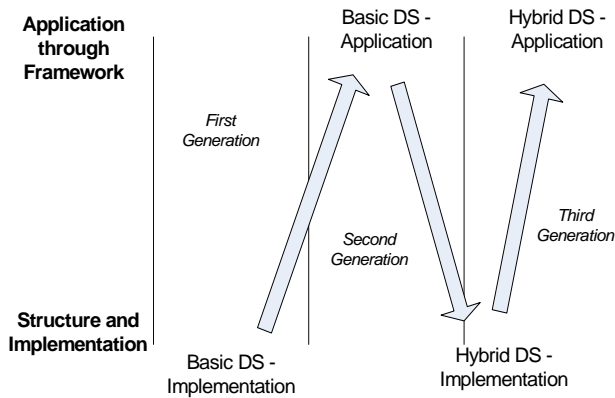


Figure 1: “Sawtooth” Model of Data Structures Evolution

This sawtooth model seems to better fit our progression than a plain linear model, in that it represents the possibility of teaching both implementation and/or usage for each state of the existing pool of data structures that exists over time. For example, an instructor could examine the implementation of the newer hybrid structures, as well as discuss the usage of these structures in order to gain multiple access methods for higher level problem solving.

Now, it could be said that the implementation level of such a model becomes less important over time, as once the frameworks (such as STL or JCF) have been created, new data structures can just be added into those frameworks, negating the need for students to implement the new structures on their own. Thus, only a few data structure developers would be concerned with the implementation, while the rest of us can be content to use the structures as implemented in the frameworks. Instructors might be tempted to skip the implementation techniques and issues for hybrid data structures, since these data structures were added to one or more frameworks very quickly.

However, this view does not match the evolution we see. The fact that a data structure has been included in a framework does not mean that it has finished evolving – for example, the fact that a certain data structure is “thread safe” does not mean it can be used efficiently. There are reasons that old data structures evolve and new data structures are developed, which we discuss in the next sub-section.

4.2 Factors Leading To New Data Structures

The basic data structures (e.g. lists, stacks, queues, sets, trees, and graphs) evolved out of the existing data needs at a certain historical point in time for computer science. These structures allowed computer scientists to store and manipulate the data involved in the problems of that time. Just as relational database systems evolved in the 1970’s and 1980’s to provide a flexible representation of set-oriented data, data structures overall evolved to represent the linear, hierarchical, and graph data of a similar period of time.

However, the problems that computer science is currently trying to solve have become much more complex and the data has changed as well. Today’s data is often less structured and is not always able to be represented as a simple list, tree, or network. For example, protein sequences, while linear, are not just a simple list – the individual elements have both position and a complex structure of their own. Similarly, spatial data takes the need for representation to a third dimension, which necessitates more complex data structures as well. Overall, new problems involve new data, which in turn requires the development of new data structures. Educators have now begun to examine this issue [5]. We see evidence of this data evolution happening right now, possibly leading to further generations of data structures.

4.3 A Fourth Generation of Data Structures?

Computer science is again seeing some significant changes. First, the sheer size of current data sets is leading to the need for new structures. As an example, the original Google paper [1] discussed the need for multi-partition file systems – disk data structures that grew beyond the limits of what in the past contained one structure. Most recently, and as noted above, the rise in multi-core processors has led to renewed interest in and the growth of parallel computation, leading in turn to new structures that better support parallel computation and concurrent access, which finally leads to higher performance. For now, we think of this new generation as “custom” data structures, and see our data structures model evolving to the following:

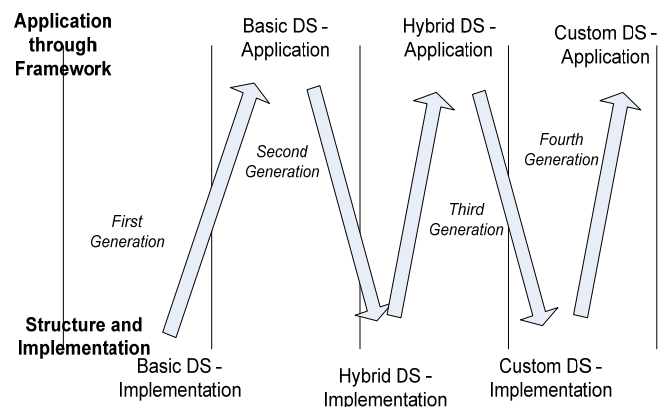


Figure 2 – Expanded Sawtooth Model Including New Custom Data Structures

It is here that we see justification for keeping the implementation model in our curriculum. The growth of new problems and of new forms and sizes of data sets means that developers may have to spend more time in the future developing their own data structures. If this is true, we don't want to see our students only knowing how to use existing data structures, but instead also want them to understand low level data structures issues so that they can customize and develop their own data structures if needed. Thus, we see a continuing need to teach both implementation and usage of data structures in our data structures course.

4.4 Evaluation

We acknowledge that we have not done any formal evaluation of these ideas – we are presenting our analysis of the data structure curriculum as based on our classroom experiences. However, we do see evidence for the ideas presented here. First, colleagues in industry affirm that they are doing work that involves customization of existing data structures in order to solve their real-world problems (unfortunately, such work is proprietary, and cannot be discussed in an open form until it is well established.) Second, the evolution and growth of standard data structure frameworks, such as the JCF and the STL, confirm that the data structures world is not static and that it is expanding and changing. Third, we have found anecdotally that students see data structures as more relevant to the current state of computer science when the data structures we discuss more directly match the problems they currently see as significant in the world. They note the addition of new data structures within the frameworks, and see our changed pedagogical emphasis as reflecting the reality of the changing world around them.

5. CONCLUSION

In conclusion, we see three important messages resulting from this discussion. First, implementation of data structures is not dead or dying. The understanding of low-level data structure implementation issues remains important, as our graduates will continue to have to customize existing data structures to make them usable and efficient for the problems they encounter in practice. While it certainly should not be the sole focus of our instruction in data structures, it still should play an important part.

Second, new data may not fit our old models, and the data structures of the past may not adequately support this new data. Researchers and practitioners alike must be ready to develop new hybrid and custom data structures to support the rapidly growing and evolving bodies of data in our world.

Third, we are in the midst of continued evolution of data structures. We have recently seen the addition of hybrid data structures and data structures that support parallel computation as just two examples of data structures evolution. We see indications of new custom data structures appearing in the future based on the increasing size and heterogeneity of data. Thus, the progression of data structures from the implementation and usage of a basic set of data structures to a world both using and implementing new data structures is already happening. We need to ensure that our pedagogical practice in the area of data structures includes these changes as well.

REFERENCES

- [1] Brin, S., and Page, L., "The Anatomy of a Large-Scale Hypertextual Web Search Engine", <http://infolab.stanford.edu/~backrub/google.html>
- [2] Budd, T., "Classic Data Structures in Java", Addison Wesley, 2001.
- [3] Collins, W., "Data Structures and the Java Collections Framework", McGraw Hill, 2002.
- [4] Ford, W., and Topp, W., "Data Structures with C++ Using STL", 2nd Edition, Prentice Hall, 2002.
- [5] Goldman, K. and Goldman, S., "Real World Case Studies to Support Inquiry-Based Learning for Data Structure Design for Real Applications", Birds-of-a-Feather session at 39th SIGCSE Technical Symposium on Computer Science Education; 40, 1 (March 2008), 561.
- [6] Java Collections Framework, <http://java.sun.com/docs/books/tutorial/collections/>
- [7] Microsoft .NET Framework, <http://msdn.microsoft.com/en-us/netframework/default.aspx>
- [8] OpenMP, <http://openmp.org/>
- [9] Parallel Java Library, <http://www.cs.rit.edu/~ark/pj.shtml>
- [10] Singler, J., Sanders, P., Putze, F., "The Multi-Core Standard Template Library." *Euro-Par 2007*.
- [11] Standard Template Library (STL) for C++, <http://www.sgi.com/tech/stl/>
- [12] Tenenberg, J., "A Framework Approach to Data Structures", in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 35, 1 (March 2003), pp. 210-214.
- [13] Vitter, J.S. "External Memory Algorithm and Data Structures Dealing With Massive Data", *Computing Surveys*, 33:2,(June 2001), pp. 209-271.
- [14] Weiss, M., "Data Structures and Algorithm Analysis in C++", Benjamin Cummings, 1994.