Work in Progress – Analyzing the Gap Between Diagrams and Code in Computer Science

Stephen Davies
University of Mary Washington, stephen@umw.edu

Abstract - **Students in sophomore computer science ("CS 2") are required to study the properties of a number of standard data structures; that is, common patterns of organizing data in a computer program. Typically, students are first presented with diagrams that graphically depict the data structure, and then shown sample code that actually implements it. We have observed, however, that there is a sizable gap between these two representations, and that many students who master the former have great difficulty translating that knowledge into the latter. We suspect that our pedagogy could be made more effective by treating diagrams themselves as formal entities, and providing students with a way of mapping operations on the "easy" (pictorial) domain into the "hard" (programmatic) domain. To help develop this technique, we carried out a semester-long experiment in which students demonstrated their understanding of the material both in diagrams and in code. The goal was to ascertain the kinds of mistakes that are often made, and how a technique like this could be most effective.**

Index Terms – Computational Thinking, Pedagogy

INTRODUCTION

Data structures students are required to study the properties of a number of standard data structures; that is, common patterns of organizing data in a computer program. There are two aspects to the commonly practiced pedagogy for these topics: (1) to present pictorial diagrams of each type of data structure, in order to illustrate the fundamentals of how it operates; and (2) to show the actual code that implements it. Research findings about the efficacy of diagrams is mixed (see [1]-[5]), but our own impression in teaching this course has been that many students find it much easier to reason about the diagrams than the code.

This leads to a problem. Some students, while demonstrating perfect understanding of even advanced data structure operations in terms of diagrams, have a great deal of trouble converting these notions into a running program. A cognitive gap exists which renders insights obtained from the diagrams, though noteworthy, ultimately of limited practical value. We suspect that our pedagogy could be improved by treating diagrams themselves as formal entities, and providing students with a way of mapping operations on the "easy" domain into the "hard" domain.

Our first step in developing this pedagogy is to examine samples of students' work in which they operate in both domains.

EXPERIMENTAL WORKSHEETS

Over the course of a semester, twenty CS 2 students were given worksheets, each of which focused on a particular data structure. Each worksheet described a particular hypothetical sequence of operations on an instance of the data structure. Students were then asked to complete two tasks: (1) to draw a pictorial representation of the state of the data structure after each of the operations, and (2) to write code that actually carried out those operations. (See Figure 1 for an example worksheet.) Each of these experiments was carried out about three weeks after the associated data structure was covered in lecture. Hence students had been exposed to the material, but a significant period of time had passed before they were asked, without warning, to demonstrate their understanding of it.

Part I – Suppose a developer needs to use a Stack of integers in the course of his or her implementation. The developer writes code to perform the following operations:

1. Instantiate a stack, called "s".
2. Push the value "5" onto the stack.
3. Push the value "19" onto the stack.
4. Pop the top value from the stack.
5. Push the value "43" onto the stack.

Suppose that the stack is implemented internally by means of a *dynamic array*. Draw a series of five diagrams, indicating what the stack would look like pictorially after *each* of the above operations. Be sure to indicate what the value of each private member variable would be at each step.

Part II – Write some code to implement the push() and pop() methods of the Stack class in C++.

FIGURE 1
A SAMPLE EXPERIMENTAL WORKSHEET.

For brevity, we present here only our findings from the "stack" worksheets, but we believe these observations are generalizable to other data structures based on common principles.

DYNAMIC ARRAYS

Students were asked to work with the stack data structure twice: first, assuming a dynamic array implementation (as in Figure 1), and then, assuming a linked-list implementation.

On the dynamic array worksheets, most students demonstrated mastery over the *conceptual* operation of the data structure. Fifteen out of 20 diagrams were perfect in their depiction of the contents of the stack (the elements' position in memory) after each operation. There was a great deal of variation, however, in exactly *how* the students drew the diagrams. To represent the array, some drew a sequence of contiguous boxes whose elements went from bottom to top; others, from top to bottom; others, from left to right. Some included memory addresses, others did not. Some drew *only* the boxes that currently had legitimate contents at each stage (e.g., one box in the array after the second operation, two after the third) while others always drew an array of the same size, partially filled.

Two kinds of mistakes were common. (1) Eleven students failed to include one or more instance variables in the depiction of the stack – namely, a "top" or "numItems" variable to keep track of how many elements in the array hold legitimate stack contents at any point in time, and a "capacity" variable to indicate how much room the dynamic array has. (2) Seven students evidently confused the stack object ("s") itself with its pointer to the dynamic array. Of the former, this omission in the diagram apparently spilled over into problems with writing the code for seven of the eleven students. Three of them demonstrated they did not know how to acquire the information that these variables were meant to contain, and four were unable to make even a reasonable attempt at writing the code.

LINKED LISTS

On the stack-as-a-linked-list worksheet, too, most students (17/20) were able to draw a correctly functioning stack, although over half of these (9) mistakenly had the stack itself pointing to the bottom element rather than the top. (The result is a functionally correct, but non-optimally performing product.) This error, too, seemed to stem from a confusion between the stack object itself and its pointer to its elements. Students normally drew the elements in the correct order, with the pointers connecting them as required, but sometimes struggled with how to depict the actual stack object itself with its "top" pointer. The overall diagrams themselves, however, convincingly represented valid linked lists that properly reflected the contents of the stack.

Despite this success, however, over half of the students (13 out of 20) were unable to successfully translate their linked list diagrams into code. These thirteen each had significant errors, mostly involving pointer concepts and syntax, or misuse of dynamic memory allocation. Six could muster no reasonable attempt at the code at all.

Interestingly, three students even wrote explanations of why they could not attempt the linked list implementation! These three comments all claimed that the ability to do so had been forgotten. One wrote "can't remember"; another said "I would have to refresh myself in the book first"; and a third wrote, "I always need some sort of reference when working on linked lists." And these three students had all provided a mostly *correct* diagram of the linked list. This indicates an enormous gap between diagrams and code in these students' minds. Despite being able to draw exactly what memory looks like, pointer for pointer, these students view the ability to write code to effect those operations a matter of memorization rather than translation. Their diagrams, in effect, are of no help at all.

DISCUSSION

In the dynamic array case, we believe that the informal nature of the diagrams is an impediment to students. It proves too easy for a student to grab a pencil and draw *something* that indicates they understand what the contents of the stack are. But because there are no formal rules about what constitutes a correct vs. incorrect diagram, this activity fails to force students to be precise enough in their diagrams to help when writing the code. The missing instance variables, in particular, are indicative of incomplete understanding. Students are most concerned with drawing the most complex part of the diagram correctly – the actual dynamic array, with its contents – that they appear to forget that there are other important variables that participate in the "state" of the object. Thus when it comes time to translate their diagrams into code, they are confused about how to determine things such as the actual height of the stack, and the capacity of the array.

We believe that students would benefit from a more formal diagrammatic approach, in which there are firm right-and-wrong rules about exactly what a pictorial representation of a data structure must look like. Forcing students to make their diagrams adhere to this formal pictorial language means that they would be guaranteed to be working with a sound blueprint before they consider how to translate boxes into pointer operations.

As for the linked lists, the discontinuity between diagrams and code is so large that the diagrams appear to serve no practical value. Students have a difficult time recognizing that each change to a diagram must logically correspond to a line of code affecting the contents of memory. We suspect that by providing students with a set of "turn-the-crank" rules that can transform diagram changes into code changes, we can significantly aid them in turning their conceptual ideas into realities.

REFERENCES

[1] Blackwell, A., et al., "Cognitive factors in programming with diagrams," *Artificial Intelligence Review*, 15(1-2), pp.95-114.

[2] Denis, M. Imagery and thinking. In: Cornoldi, C. and McDaniel, M. (eds.), *Imagery and Cognition*. New York: Springer-Verlag.

[3] Petre, M., "Why looking isn't always seeing: readership skills and graphical programming," *Communications of the ACM*, 38(6), pp.33-44.

[4] Scanlan, D., "Data structures students may prefer to learn algorithms using graphical methods," Proceedings of SIGCSE 1987, pp.302-307.

[5] Stasko, J., et al., "Do algorithm animations assist learning?: an empirical study and analysis," Proceedings of SIGCHI 1993, pp.61-66.