Testers and Visualizers for Teaching Data Structures*

Ryan S. Baker
Dept. Comput. Sci.
Brown Univ.
rsb@cs.brown.edu

Michael Boilen
Dept. Comput. Sci.
Brown Univ.
mgb@cs.brown.edu

Michael T. Goodrich
Dept. Comput. Sci.
Johns Hopkins Univ.
goodrich@cs.jhu.edu

Roberto Tamassia
Dept. Comput. Sci.
Brown Univ.
rt@cs.brown.edu

B. Aaron Stibel
Dept. Comput. Sci.
Johns Hopkins Univ.
astibel@cs.jhu.edu

Abstract

We present two tools to support the teaching of data structures and algorithms: *Visualizers*, which provide interactive visualizations of user-written data structures, and *Testers*, which check the functionality of user-written data structures. We outline a prototype implementation of visualizers and testers for data structures written in Java, and report on classroom use of testers and visualizers in an introductory Data Structures and Algorithms (CS2) course.

1 Introduction

Much recent work in Computer Science education has studied the development of computer-aided tools, such as algorithm animation, for pedagogical purposes (e.g., see the 1998 SIGCSE proceedings for no fewer than eight papers on this topic). Some lessons learned from this work include observations that demonstration animations are most effective when they are accompanied by supporting text or audio explanation, and the use of animation is even more effective if students can produce animations from their own code.

In this paper, we describe computer tools for teaching introductory data structures (CS2), focusing on methods for the visualization and testing of student-written code. We present prototype visualizers and testers, and report on claassroom experience that shows how they were used by students learning data structures in CS2. Unlike some previous animation schemes, our approach does not require any modification to the student's code in order to facilitate visualization and testing. Rather, we simply require that the student's code conform to a simple application programmer interface (API).

*Work supported by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grants CCR–9625289 and CCR–9732327.

CS2 at Brown and Johns Hopkins is taught in Java within the object-oriented paradigm [6]. Instead of writing single-use, throw-away implementations data structures for a specific task, students write generic, reusable implementations that conform to given APIs. Students then implement algorithms assuming the availability of data structures that realize these APIs. To emphasize coherent design, the APIs are given for a simplified version of those in JDSL, the Library of Data Structures in Java [1, 5], a research effort currently in progress at Brown and Johns Hopkins.

Although our visualizers and testers have been developed and experimented with in Java, we believe that they can be adapted and succesfully used with any object-oriented programming language.

2 Visualizers

2.1 Previous work

In the last fifteen years, a wide variety of systems for visualizing algorithms and data structures have been developed. Although there is enough diversity and complexity among them to allow the construction of an extensive taxonomy [11], most of these visualizers can be separated into two broad groups in relation to their use for teaching data structures.

The first group of visualizers, from the pioneering systems BALSA [3] and TANGO [15], to recent efforts such as JAWAA [10] and GAIGS [9], are powerful tools for animating and comparing different algorithms. They are also useful for teaching data structures, because they provide a high-level, conceptual view of how the data structures function. One limitation of these visualizers, however, is that they are either intended to give a non-interactive demonstration of the structure performing some operation, or they allow access only to a limited subset of a data structure's potential functionality. We believe that in a CS2 class that emphasizes generic data structures, it is more useful to have a tool that students can plug their data structures directly into and see if all of its methods functioned as intended.

A second group of visualizers, inspired in part by Myers' work on Incense [8], have been used with some

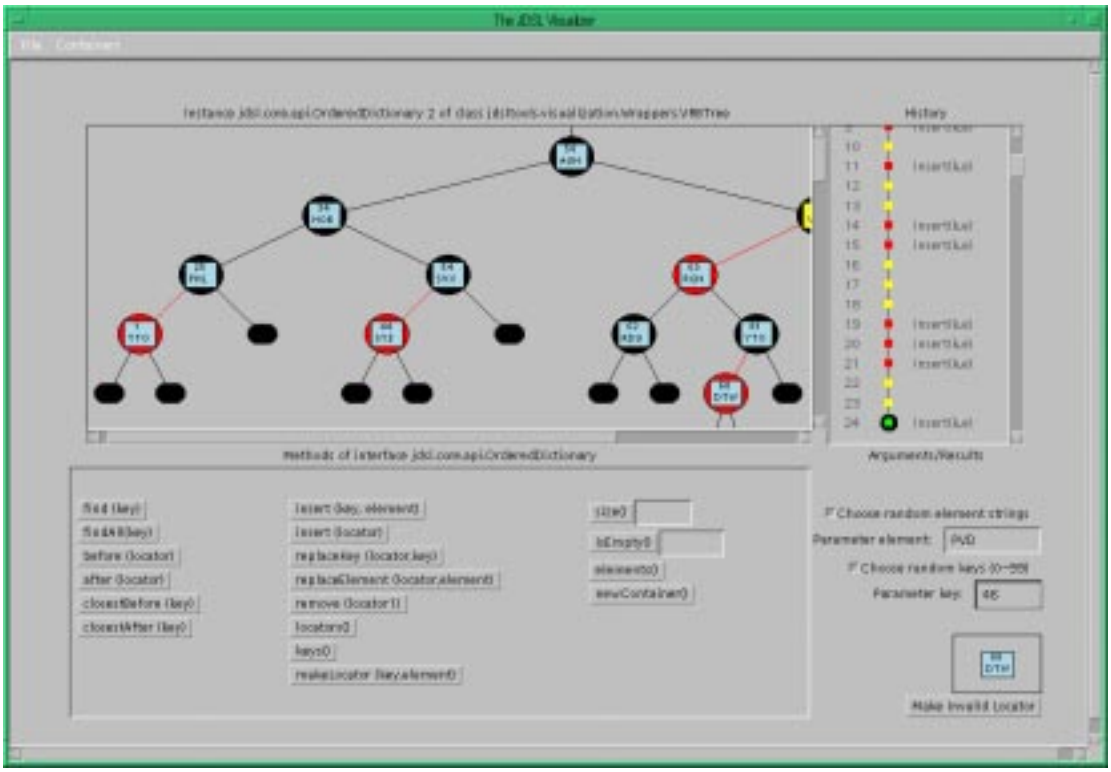

Figure 1: The JDSL Visualizer, seen here displaying a Red-Black Tree.

success in introductory CS courses, from Amethyst [11] and Field [13] to more recent visualization systems such as the one constructed by Sangwan et al. [14]. These visualizers allow the user considerable interactivity with their data structures as well as a display of the different data structures in memory, by integrating visualization with a source-level debugger. Nevertheless, they accomplish this task by displaying the physical organization of the data in memory rather than the underlying structure it represents, which is undesirable from the point of view of an introductory data structures course.

2.2 Goals

We advocate the development of a visualizer of user-implemented data structures with the following goals:

- Produce visualizations of data structures while interfering minimally with their design.
- Allow users to interact at runtime with all the methods of a data structure to verify its operation.
- Display data structures at a conceptual level instead of simply visualizing the contents of memory.

Previous work addressing the above requirements includes a system by Augenstein and Langsham [2] that provides a high-level, abstract display of data structures written in Pascal. This system is capable of simultaneous visualizations of multiple data structures. Unfortunately, its interaction with user-written programs is accomplished by requiring the user to modify her program to insert statements that produce output in a specified format to a text file.

2.3 The JDSL Visualizer

We describe in this paper the *JDSL Visualizer*, which is a tool for visualizing data structures in Java. The system interfaces with a user-written implementation of a data structure through the API of that data structure, as specified in the JDSL library [1, 5]. It can display more than one type of structure at once, showing interactions between structures (for example, a binary tree and the sequence of its nodes visited in preorder). It keeps an extensive history of the active data structures by recording for any point in time their state, the methods invoked on them (including parameters), and the return values. Unlike many previous visualization systems, which only allow the user to move along the history, it also supports direct jumps to specific points in time.

By default, the JDSL Visualizer displays a data structure before and after the execution of API methods. In order to allow users to examine the inner workings of their implementation, it additionally supports the display of snapshots based on reaching "interesting

events"[4], designated by "hooks" within the user's code, such as:

```
viz.VisualizationController.snapshot
    (java.awt.Color.yellow,"Recoloring");
```

Such hooks are entirely optional, however, for the novice student may simply wish to visualize his data structure without making any code modifications.

As shown in Figure 1, the JDSL Visualizer's window is split into several different panels. The top-left panel displays the structure at a specific moment in time; the history panel is at the top-right (note that user-defined snapshots are colored lighter in the figure). On the bottom level there is a panel of buttons, where each button corresponds to a method of the data structure, and a panel that allows the user to set the parameters of method calls. Not shown are a window displaying the exceptions thrown and a window for on-line help.

A prototype of the JDSL Visualizer has been implemented. It supports six fundamental data structures: Enumerations, Sequences, Binary Trees, Restructurable Binary Trees (i.e., binary trees with rotations), Heaps, and Red-Black Trees.

2.4 How the Visualizer works

The visualizer has a framework of different components that can be swapped in and out depending on what structure is currently being visualized. Administrative components keep track of the different structures in memory, the history of each structure, and which structure is currently on screen. When the user wants to display a different type of structure than the current one, the administrative components swap in different components for displaying the structure, and for its buttons and parameters. All of the components are designed to conform to interfaces, so that if a different sort of functionality is needed — for example, the ability to visualize a structure in a different way or the ability to input a new sort of parameter — all that is necessary is writing a new class (or set of classes) to conform to the interfaces. This modular design should make it easy to expand the current prototype.

Whenever an interesting event occurs, either user-defined or predefined (e.g., when an API method is called from the visualizer), the visualizer updates its history of all of the data structures in memory, reading the data out of them into data structures known to be reliable. The histories of each of the data structures are synchronized to allow the user to examine what each looked like at one point in time. After all of the histories are up to date, the visualizer displays the on-screen structure in its current state by iterating through each of its elements, starting, for example, at the root of a tree or first element of a sequence, and continuing recursively until all leaves or the last element have been explored. The visualization algorithms are designed to accommodate any implementation that includes a small subset of the structure's full functionality—for example, visualizing a Binary Tree requires that we can get the root, get the children of an arbitrary node, and test if a node has children.

3 Testers

3.1 Previous work

Program testers have been previously used as aids to students and instructors. However, unlike visualizers, testers have received little attention in the computer science education literature. One example of a tester package is ASSYST [7], which is designed to automate the grading of student programs. It checks correctness by parsing the text output of a program, using Lex and Yacc, and compares it with the correct output. ASSYST also measures execution time and computes several source code metrics to check for appropriate commenting and style.

The TRY system [12] is another package to test student programs. Unlike ASSYST, this package was designed to be used by both the students and the instructor. TRY also compares text output from a program to the saved output from a correct program. Its main feature is that it allows students to run a TRY tester without seeing the correct output. Unfortunately, this feature causes many security problems.

3.2 The JDSL Testers

Our *JDSL Testers* were designed to be thorough, robust, and easy to use. By being thorough, our approach provides students with accurate reports on the functionality of their programs, which in turn increases grading efficiency. We do not achieve this goal by parsing text output as previous testers have done, for such an approach would be unwieldy for testing large programs. Moreover, output-parsing approaches usually have difficulty recovering from error conditions and often do not provide accurate results. Instead, we interact directly with the data structures, handling and storing responses, and comparing the behavior of the student's code with a "reference" implementation. This approach gives our testers considerable flexibility and functionality, and at the same time allows the student to use them without having to modify her code.

Another goal for us was to make the tester program easy to adapt, so that students could modify the testers for their own needs. To meet this goal, we provided the students with templates that they could use to write their own tester programs. By using the templates, documentation, and sample code, student could write their

own testers.

Recently, the testers have been modified to take advantage of a new and powerful feature of the Java language, the Reflection API. By using introspection, the testers have been reduced to a clean "scripting" language.

3.3 How the testers work

The JDSL Tester API consists of four major components, the parser, the generic tester, the generic factory, and the comparator interface. The generic tester execute methods on data structures. Since it utilizes the Reflection API, it takes parameters as strings (see the example code below), and then uses the parser to interpret these strings. The generic factory contains methods to make creating and initializing data structures easier. The comparator interface provides a framework to test two structures against each other. Commonly, the tester will be run on a demonstration implementation and the student's implementation simultaneously. The demonstration implementation is known to be correct, so by comparing the two implementations, one can judge the correctness of the student's implementation.

In order to write a tester, one must specialize the generic tester, the generic factory, and, if desired, a comparator. The tester will use the factory to produce data structures, and then execute methods on those structures.

Below is a snippet of a tester that illustrates some of the features that the JDSL Tester API has:

```
for(int i=0;i<10;i++) {
    execute("insert","Integer("+i+")",
        "Integer("+(10-i)+")" );
    executeCheckReturn("size","void","int "+i);
}
```

This snippet of code inserts ten elements into a data structure, and checks to see if the size is correct after each insertion. The *executeCheckReturn* tests the return of the specified method against the provided value. If the test fails, a useful error message will be automatically produced. Also, if the code being tested throws an unexpected exception, it will be caught, an error message will be produced, and depending on how the tester is configured, it will either continue or abort. In addition, if the student's *size* method is incorrect, and enters an infinite loop, the tester will terminate the method's execution after a user specified timeout period has passed, produce an error message, and continue. The tester can also check if a method throws an exception, and produce an error message if the exception is not thrown.

4 Classroom Experiences

4.1 Integration of Visualizers and Testers in CS2

In the CS2 class at Brown, the JDSL Visualizer is used by the students in assignments where they program generic data structures such as sequences, binary trees, heaps, and dictionaries. Once a student has correctly implemented a small subset of the structure's methods, she will be able to see what her structure looks like and interact with it. From this point, students will be able to use the visualizer to help them design the data structure, testing each method in a simple and intuitive fashion after writing it. For more complicated data structures (for example, red-black trees) the student will be able to see exactly how the structure is performing a complex operation.

Once a student has written all the code required to complete the assignment, she can then use a JDSL tester to check her data structure for correctness. At first we considered providing the students with a comprehensive tester to check many special cases. However, the availability of a complete tester would not encourage a student to think critically about the different special cases that must be handled by her code. Hence, we decided to provide the students with incomplete testers that only test some methods and some cases. The students are then encouraged to think about their programs, and identify special cases, and then write testers for them.

Through the combination of these two methods of verification, the student gets immediate feedback at the level of abstraction that their problem needs: conceptual problems will become obvious in the visualizer while more subtle special-case errors will be pointed out by the tester. This feedback is more useful than only providing standard debugging tools and then giving feedback during grading. Feedback through the visualizer and testers is immediate, and allows the student to spend her time improving her code rather than writing the code to test her program.

4.2 Positive Experiences

The JDSL testers and visualizer have been used in CS2 at Brown for one and two years respectively. Thanks to questionnaires given to the 120 students after the third week of class this spring and at the end of semester, as well as the suggestions of former students, we have ample and specific feedback to use in evaluating the usefulness of visualizers and testers in CS2.

Many students found testers and the visualizer to be valuable debugging tools, and by semester's end, only 4% of the class reported that they had not found the visualizer and testers useful. Teaching assistants observed that the students who used the visualizer and testers

regularly produced programs with fewer errors.

One concern we had was that if students had to learn how to use the visualizer and testers on top of the CS2 material, they might find the class' initial learning curve unacceptably steep. However, we were pleased to find that this was not a major problem. In the week-three questionnaire, 86% of the class said that understanding the visualizer and testers was not the trickiest part of the class thus far.

4.3 Limitations and Drawbacks

Although the majority of our experiences with the visualizer and testers have been positive, we have found some drawbacks to using them in CS2. Some students had difficulty seeing the visualizer and testers as complementary tools, and therefore did not use them in the most effective manner. Particularly, in the situation where the tester revealed a subtle bug that the visualizer did not display, many students perceived that as an indication that the tester was faulty rather than as evidence that their own program was incorrect.

5 Future Directions

In changing the visualizer from a prototype to a finished product, a few steps need to be taken. One essential modification is to switch the visualizer to an introspection-based architecture for creating buttons corresponding to the various methods of the structure (as opposed to the current customized-button approach) in order to speed its adaptation to different interfaces. Another change is the previously mentioned expansion of the set of data structures supported by the visualizer.

Currently, the testers only check whether an implementation is correct. We would like to expand the testers' capabilities to determine time and space complexity, since these are both qualities of a good implementation, and are commonly stressed in CS2.

A final future direction for the visualizer and testers is to complete their adaptation to the Internet. Using a browser that supports the full capabilities of Java 1.1, the JDSL visualizer can be run within the browser and can load structures that are at remote sites by their URL. The testers are currently designed to run within a shell, but would benefit from being able to load structures from remote sites.

6 Availability

To read more information about the JDSL Visualizer and JDSL Testers, point your browser at the JDSL Web page `http://www.cs.brown.edu/cgc/jdsl`. The visualizer can be run over the Web using a browser, and screen shots are available for perusal. Additionally, the visualizer and example testers are available for download at that site.

References

[1] The Library of Data Structures for JAVA Webpage. http://www.cs.brown.edu/cgc/jdsl.

[2] M. Augenstein and Y. Langsham. Graphic Displays of Data Structures on the IBM PC. *Proceedings of the 17th SIGCSE Technical Symposium on Computer Science Education*, 1986.

[3] M. H. Brown. *Algorithm Animation*. MIT Press, 1988.

[4] M.H. Brown and R. Sedgwick. Interesting Events. In J. Stasko, J. Domingue, M.H Brown, and B.A. Price, editors, *Software Visualization:Programming as a Multimedia Experience*, chapter 12, pages 155–171. MIT Press, 1997.

[5] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia. Abstracting Positional Information in Data Structures: Locators and Positions in JDSL. Manuscript, 1998.

[6] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in JAVA*. John Wiley & Sons, Inc., 1998.

[7] David Jackson and Michelle Usher. Grading Student Programs Using ASSYST. *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, 1997.

[8] B. Meyers. A System For Displaying Data Structures. *Computer Graphics*, 17, 1983.

[9] T. L. Naps and E. Bressler. A multi-windowed environment for simultaneous visualization of related algorithms on the World Wide Web. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 1998.

[10] W. C. Pierson and S. H. Rodger. Web-based Animation of Data Structures Using JAWAA. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 1998.

[11] B. A. Price, R. M. Baecker, and I. S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 3:211–264, 1993.

[12] K.A. Reek. The TRY System or How to Avoid Testing Student Programs. *Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education*, 1989.

[13] S. Reiss. Visualization for Software Engineering — Programming Environments. In J. Stasko, J. Domingue, M.H Brown, and B.A. Price, editors, *Software Visualization:Programming as a Multimedia Experience*, chapter 18, pages 259–276. MIT Press, 1997.

[14] R. Sangwan, J. Korsh, and P. LaFolette. A System for Program Visualization in the Classroom. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 1998.

[15] J. T. Stasko. TANGO: a Framework and System for Algorithm Animation. *Computer*, 23:27–39, 1990.