

1. Introduction
2. Three Aspects of Abstract Data Types
3. JVALL Overview and Objectives
4. A Simple Example
5. JVALL Features
6. General Interactive Linked List Controller
7. Educational Uses of JVALL
 - 7.1 Tutorials
 - 7.2 Active Student Learning
 - 7.3 Debugging Student Programs
8. Conclusions and Future Work
9. Acknowledgments
10. References

Printer friendly version

A Linked List Prototype for the Visual Representation of Abstract Data Types

Herbert L. Dershem, *Hope College*

Ryan McFall, *Hope College*

Ngozi Uti, *Northern Kentucky University*

About the authors...

Abstract

Visualization is an important tool for learning abstract data types. One approach to this is to expand the class of a data type to include visualization of all operations, thus automatically generating visualizations in programs that use the data type. A prototype of this approach applied to the Java `LinkedList` class is described in this paper. Also described are general purpose controllers for this process, instructions for user generation of more specialized controllers, and ways this tool can be used in an educational setting.

1. Introduction

The use of visualization to enhance learning of data structures and algorithms in computer science has been popular for many years. Because of the number of continuing projects, a progression of packages have developed, prominent among them are Balsa [1], Tango [2], GAIGS [3], and JAWAA [4]. In addition to these projects, many others have developed software to aid learning through visualization and animation of data structures and algorithms. Early work produced scripted algorithm animations. Later these evolved to provide user control. Most recent activity has been hosted on the web.

This paper describes a project to develop a visualization of the algorithms for implementing methods of the Java `LinkedList` class. This visualization will serve as a prototype for visualization of any Java Collection Class. The Java Collection classes consist of a hierarchy of interfaces and classes representing Abstract Data Types (ADTs), originating with the Java Collection interface. These classes each consist of an API containing the methods of the class, but are relatively independent of the implementations. The prototype, called JVALL (Java Visual Automated Linked List), was developed as an extension of the Java `LinkedList` class. The approach used here is similar to that used by Jeliot [5,6], where a user-written Java program is submitted to the Jeliot server, which produces an animation of operations that the submitted code performs on a data type. Jeliot provides animations for all Java primitive types, arrays, stacks, and queues. The JDSL Visualizer [7] also uses this approach by providing the application programmer interface (API) for abstract data types and automatically generating visualizations. The APIs of JDSL, however, do not conform to Java Collection Class APIs.

Many researchers have conducted studies of the effectiveness of visualizations for the learning of algorithms and data structures. These studies have produced mixed results and have led some to question the validity of visualization as an aid to learning in this context. A recent meta-study [8] examines many earlier studies and identifies factors that the earlier studies have empirically agreed lead

to successful visualizations. We have utilized these results in the design of JVALL.

The following are four features that we believe are key to successful visualizations and that we have attempted to provide in this prototype:

1. Ease of Use

In the meta-study of algorithm visualization effectiveness, it is stated that among the reasons visualization software has not been used by faculty beyond its developers are that they do not have time to learn it, that they feel it will take time from other class activities, and that they believe it will require too much time and effort to create visualizations [8]. It was thus a primary objective here to make the use of visualizations as easy as possible for both student and instructor, avoiding the need to learn new syntax or make extensive changes in pedagogy.

2. Flexibility

Just as students have a variety of ways of learning, so too do instructors have a variety of preferred teaching methods. One drawback of many instructional tools is that they are not widely used because they do not match the preferred pedagogical approach of many instructors. JVALL was designed to provide flexibility in its use, not restricting the learner or the instructor in the ways it can be used. The goal is to provide a tool that can easily fit into any learning framework rather than a package that is self-contained for a pre-specified use.

3. Platform Independence

In order to support a wide flexibility in the way a visualization is used, it must be supported in a variety of environments. The platform independence provided by Java minimizes the concerns for availability of an appropriate platform.

4. Interactive

The meta-study concludes that "the most successful educational uses of algorithm visualization technology are those in which the technology is used as a vehicle for actively engaging students in the process of learning algorithms [8]." It is therefore important that JVALL support student interaction, both with the algorithm and with the visualization itself. Student interaction with the execution of the algorithm has been noted as an important factor in successful visualization, not only by the meta-study, but also by others [9] [10]. Effective tools for visualization control include selection of color, user-controlled speed [11], and the ability to reverse the algorithm [12].

2. Three Aspects of Abstract Data Types

There are three aspects of ADTs that are important for students to learn. Textbooks and instructors have differing views on the emphasis and order for presenting these aspects, but there is consensus that it is important for students to gain an understanding of all three. The first is the conceptual view of the ADT. In the Object-Oriented framework, this is represented by the API definition of the methods, including the way each method is called, the specification of the parameters of each call, and the specification of the result of a call, including return values and its impact on the ADT instance itself.

A second important aspect of ADTs is their application. This requires an understanding of the ways an ADT is used, in what situations its use is advisable, and how it is used effectively in those situations.

The final aspect is the implementation. This includes the data structures used to represent the ADT, the algorithms used to implement its methods, and techniques for careful space and time analysis of those algorithms.

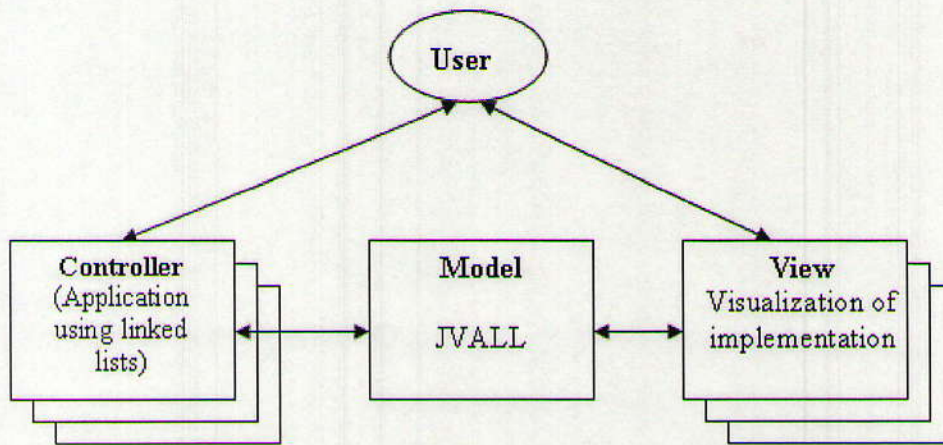
JVALL has been developed to support all three of these aspects of learning an ADT, in particular, the list ADT as represented by the Java *LinkedList* class. The JVALL class has an API that is identical to that of the Java *LinkedList* class, thus supporting the learning of the conceptual view represented by this Java collection class. Since JVALL can be used in any context where the *LinkedList* class is used, it provides an efficient way to understand applications of the ADT through visualization, whether the application is student-written or provided by the instructor. Finally JVALL provides views of multiple implementations of the ADT, giving the student a means to carefully examine the implementation visually. In particular, when used with a source-level debugger, JVALL can enhance the student's ability to perform time and space analysis.



3. JVALL Overview and Objectives

Bergin et al. [13] applied the model-view-controller framework as a design paradigm for visualizations. In Figure 1, we show an overview of JVALL based on this framework.

Figure 1. Model-view-controller



Typical to the MVC framework, JVALL supports multiple controllers and multiple views through a single model, the linked list ADT. A controller may be any class that uses the linked list ADT. It may be a specially designed interactive instructional module, a class implementing another ADT that uses the linked list such as a stack, or any other application, complex or simple, that uses one or more linked lists. The controller may interact with a user, such as the student or the instructor, it may run under file control, or it may be an application with no input at all.

There are also multiple views that represent multiple implementations of the ADT. In the current system, only two implementation views are provided for the linked list ADT, the linear, singly-linked and the circular, singly linked implementations. Others will be provided in the future. The user also interacts with each view, controlling the visualization's color scheme, speed, and direction (through an undo/redo facility).

The JVALL class itself is the *LinkedList* model that implements the underlying Java collection class and interfaces with the multiple controllers and views.

The objective of this project is to produce a linked list visualization tool that provides a prototype for further development and has the following capabilities:

- Provides visualization of multiple ADT implementations. As stated above, two implementations are provided and others can be easily added.
- Shows linked list operations with user-controlled animations. Operations are animated at a speed that is controlled by the user and allows detailed observation of the construction and modification of nodes. In addition, the user determines which implementation is viewed.
- Provides visualization for any Java program that uses the *LinkedList* class. The JVALL class is an extension of the Java *LinkedList* class. Therefore, any program that uses this Java class is very easily modified to enable visualization of its linked list operations.

- Supports visualization of both Java applets and applications. The tool is easily applied in both of these Java run-time environments by a parameter passed when the linked list is constructed.



4. A Simple Example

We introduce the full capabilities of JVALL by providing a simple example of a Java application that constructs a linked list and performs a few very simple operations. The Java source code to this example is found in Figure 2.

```
import jvall.Jvall;
import jvall.JvallListener;
import java.util.ListIterator;

public class JvallTest implements JvallListener {
    public static void main(String[] args) {
        Jvall myList = new Jvall(Jvall.STANDALONE);
        JvallTest myTest = new JvallTest();
        myList.addAnimationListener(myTest);
        myList.addFirst("One");
        myTest.waitForJvall();
        myList.addLast("Three");
        myTest.waitForJvall();
        myList.add(1, "Two");
        myTest.waitForJvall();
        myList.set(myList.indexOf("Two"), "TWO");
        myTest.waitForJvall();
        ListIterator myIterator = myList.listIterator(0);
        while (myIterator.hasNext()) {
            System.out.println(myIterator.next());
        }
    }

    public JvallTest() {
        this.done = true;
        this.status = "Uninitialized";
    }

    public synchronized void waitForJvall() {
        while (!this.done) {
            try {
                wait();
            } catch (Exception e) {}
        }
    }

    public synchronized void animationEvent(int event) {
        if (event == Jvall.ENDED) {
            this.done = true;
            notify();
        }
        else if (event == Jvall.RUNNING)
            this.done = false;
    }

    public void statusUpdate(String strStatus) {
        this.status = strStatus;
    }

    private boolean done=true;
    private String status;
}
```

Figure 2. A simple example program using JVALL.

The *Jvall* class (which implements the JVALL ADT) has the same API as the Java *LinkedList* class except for the inclusion of a constructor that takes a single parameter that specifies whether the JVALL will run as a Java application or an applet. Also, the *addAnimationListener* method enables a *JvallListener* to be attached to the JVALL linked list.

JvallListener is an interface with two methods:

```
public void AnimationEvent(int event);
public void statusUpdate(String strStatus);
```

JvallListener provides communication from the view to the controller with the model serving as an intermediary. This allows the animation to communicate with the controller without knowledge of the details of the controller's implementation. The two possible event parameters that can be sent to *AnimationEvent* are *Jvall.ENDED* and *Jvall.RUNNING*. These two int constants indicate that the animation has finished or begun running. Whenever the status of an animation is changed, the *statusUpdate* method is called and the *String* that appears in the status textfield of the animation will be the *strStatus* parameter. In the case of the program in Figure 2, the instantiation of *statusUpdate* is

```
public void statusUpdate(String strStatus) {
    this.status = strStatus;
}
```

This will set the instance variable *status* to the *String* that is returned to this class by the view. That *String* will correspond to the *String* that appears in the *TextField* labeled "Animation Status" in the animation view window.

In the example in Figure 2, *JvallTest* is also a *JvallListener*. It sets its instance variable *done* whenever the running state of the animation changes. It also keeps the latest status of the animation in instance variable *status*.

The method *waitForJvall* waits for a notification that a visualization that is active has been completed.

```
public synchronized void waitForJvall() {
    while (!this.done) {
        try {
            wait();
        } catch (Exception e) {}
    }
}
```

This is called following every JVALL method call so that the program will not proceed until the animation process is completed.

Three nodes are added to the JVALL linked list, one is changed, and then an iterator is created that iterates through the list, printing each node. This is accomplished in the main program through the calls

```
myList.addFirst("One");
myTest.waitForJvall();
myList.addLast("Three");
myTest.waitForJvall();
myList.add(1, "Two");
myTest.waitForJvall();
myList.set(myList.indexOf("Two"), "TWO");
```



```
myTest.waitForJvall();
```

The interspersed `waitForJvall()` calls require the program to wait for the completion of the visualization before proceeding to the next operation. The output produced by the program is

```
One  
TWO  
Three
```

This is printed by the statements

```
ListIterator myIterator = myList.listIterator(0);  
while (myIterator.hasNext()) {  
    System.out.println(myIterator.next());  
}
```

Figures 3, 4, 5, and 6 show the progression of visualizations of the list after each operation. Not viewable in these figures are the detailed animation of the search through the list and the modification of links. For example, during the execution of the method call

```
myList.set(myList.indexOf("Two"), "TWO");
```

a small arrow will trace the search through the linked list for the node with contents "Two" and, when it is found, will change the contents to "TWO."

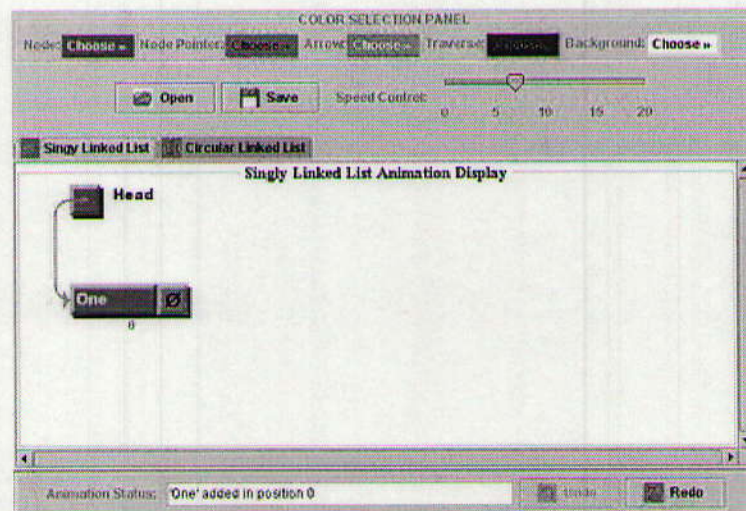




Figure 3. JVALL visualization after first add.

 Full size image
 An external link to the authors' general Linked List Operation applet.

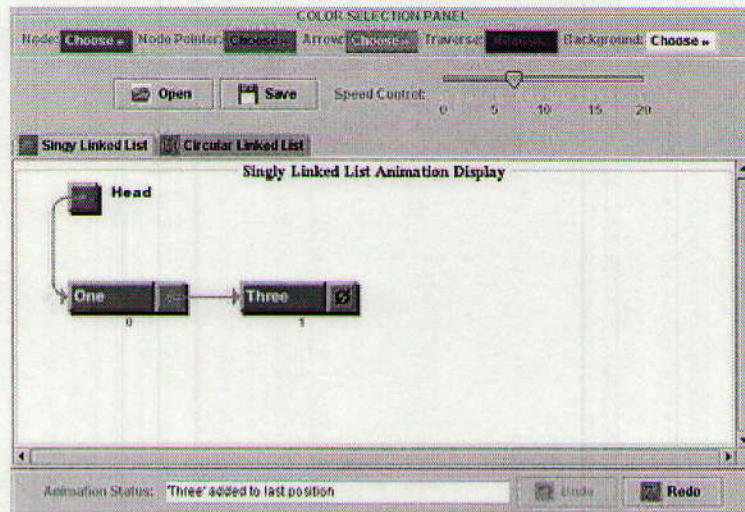


Figure 4. JVALL visualization after second add.

Full size image

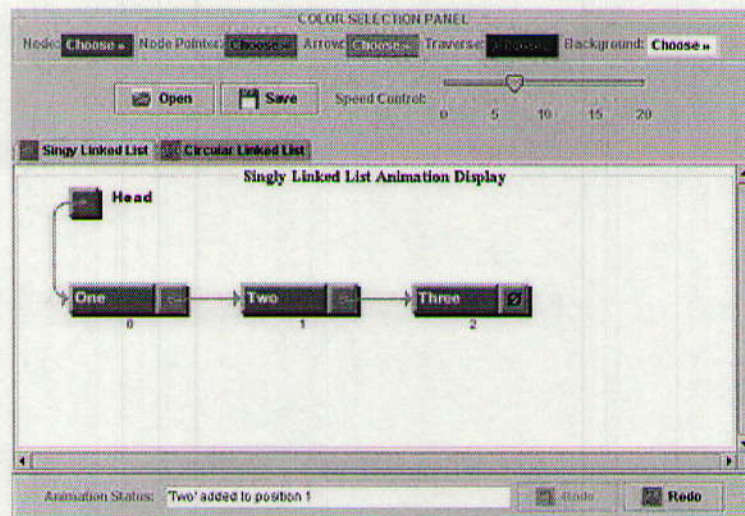


Figure 5. JVALL visualization after third add.

Full size image

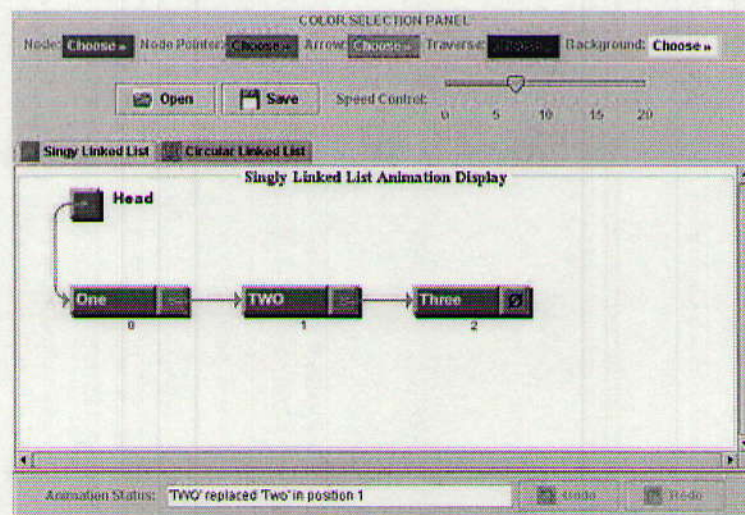


Figure 6. JVALL visualization after change.

Full size image

5. JVALL Features

Figures 7, 8, and 9 illustrate some of the important features of JVALL.

Visualization

The center portion of the display shows the linked list and its operations are animated there. Figure 8 illustrates a snapshot of an animated addition of a node in progress.

Animation Status Report

The current status of the animation is displayed in the area circled in orange in Figure 7.

Undo/Redo Capability

The undo and redo buttons, circled in black in Figure 7, permit the user to rewind and review the visualization steps.

File Load and Save

The buttons circled in green in Figure 7 are used to permit the user to save the current visual linked list and to retrieve a visual linked list that was saved previously. While this feature presently saves and reads the nodes as a text file, a future improvement to this software will permit any serialized objects to be saved and restored.

Color Selection

The user controls the colors of all components of the display through use of the panel circled in blue in Figure 7.

Speed Control

The speed control slide bar allows the user to control the speed of the animation and is circled in red in Figure 7.

Multiple Implementation Views

This selection is circled in yellow in Figure 7. The present implementation includes only two implementations for the linked list, but future enhancements will provide additional options. Figure 9 shows the circular linked list view of the list shown in Figure 6.

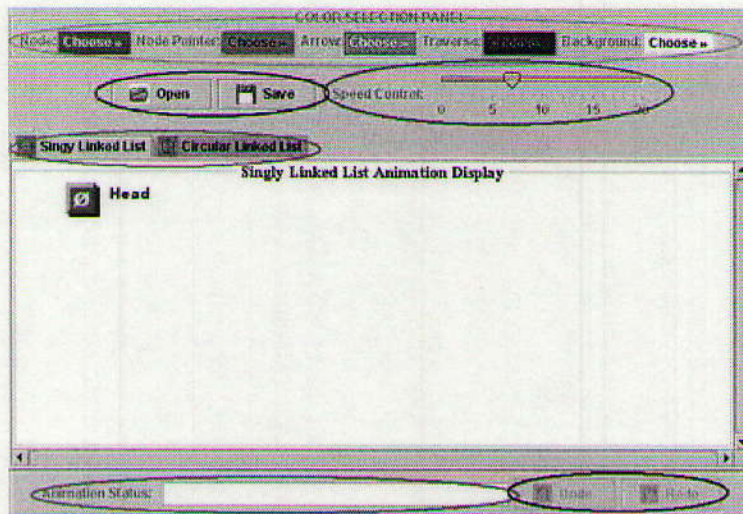


Figure 7. Empty JVALL Window with Feature areas circled.

image Full size image

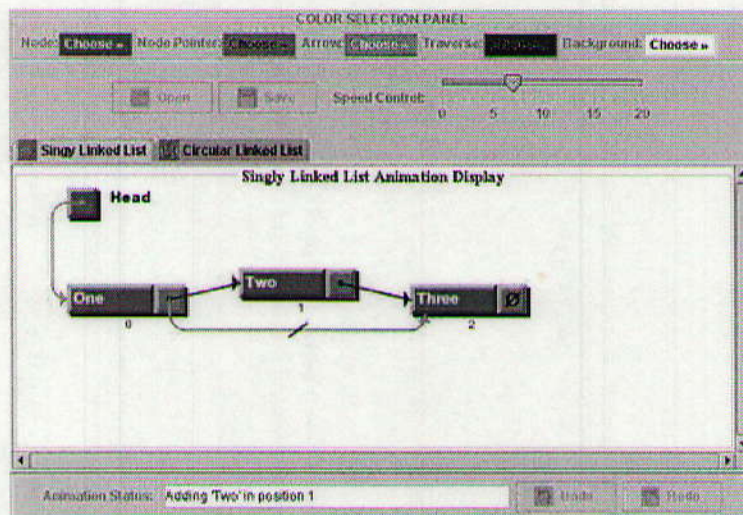


Figure 8. Snapshot during animation of add method execution.

image Full size image

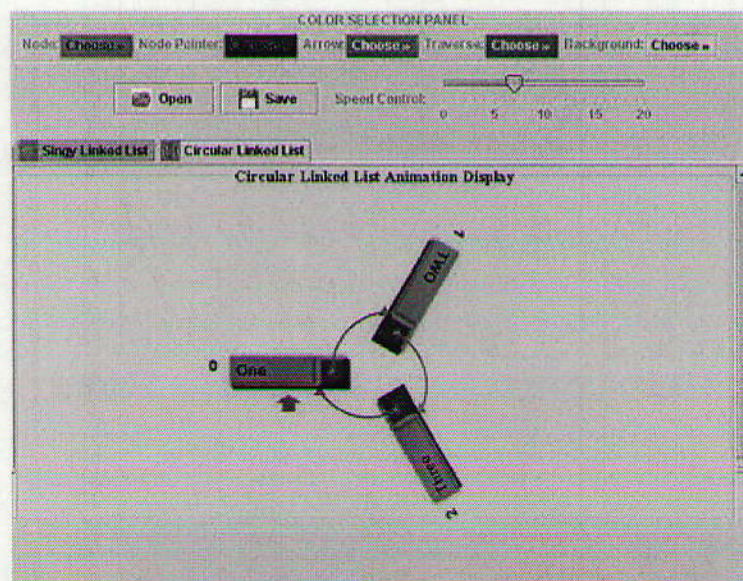


Figure 9. Circular Linked List Implementation view.

image Full size image

6. General Interactive Linked List Controller

The distribution of the JVALL package includes one controller program that is generalized, interactive, and permits the user to perform linked list operations and watch their results through a graphical user interface. The controller window appears directly below the JVALL window as shown in Figure 10.

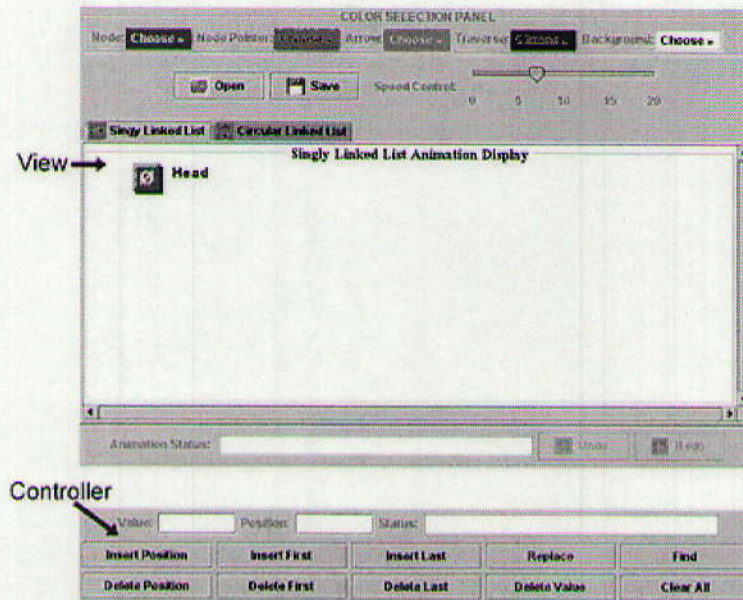



Figure 10. JVALL window with General Interactive Controller.

 Full size image

The user can type into the value and position text boxes of this controller window to insert nodes into the specified position of the linked list. In standalone mode, a user has the option of opening text file, loading it into the linked list and continuing with the normal linked list operations such as insert, delete, and replace. Each time a button is clicked within the controller's window, JVALL receives a request from the controller and displays the animation accordingly. When the user requests an invalid operation such as inserting or deleting from a position that does not exist, JVALL will throw an appropriate exception. The controller will then catch that exception and report the problem, indicating to the user valid position choices.

An example of this is shown in Figure 11. In Figure 11a, the user has selected position 4 to insert a new node. But the only valid positions for insertion would be 0, 1, 2, and 3. Figure 11b shows the resulting view. Here the appropriate message is reported for both the animation status and in the status textbox of the controller. These correspond because the controller program simply reports the status parameter that JVALL returns to it.

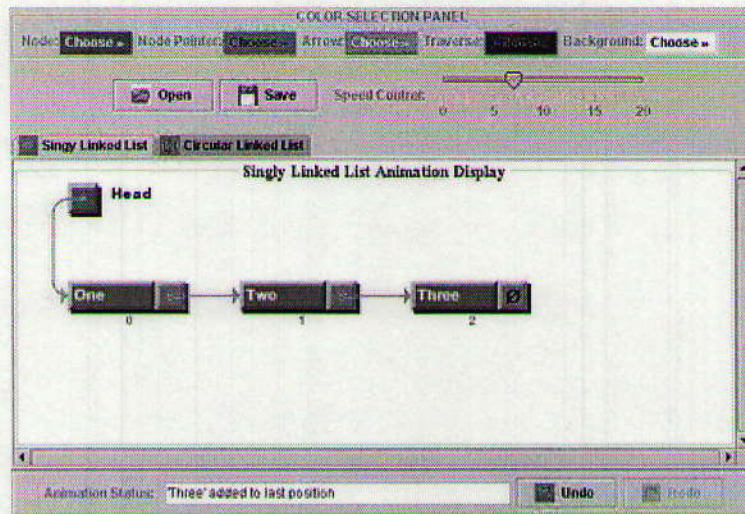


Figure 11a. Before Insert Position button pressed.

[image](#) Full size image

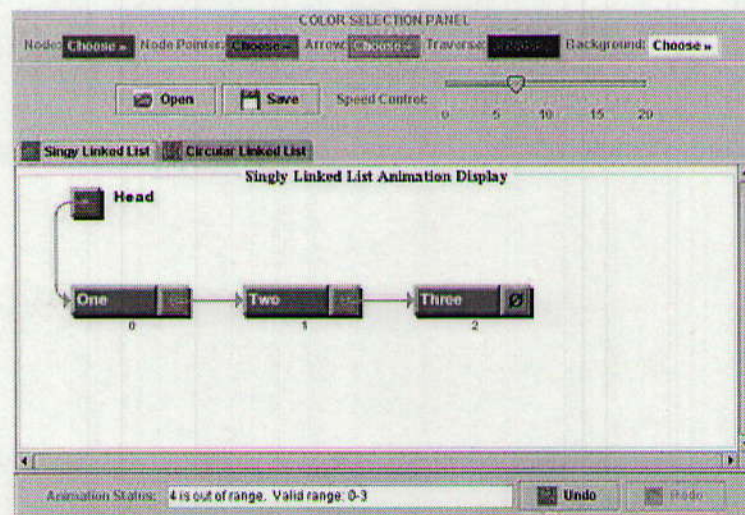
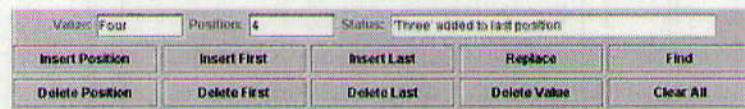
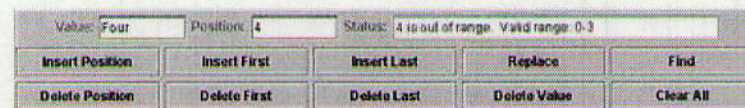


Figure 11b. After Insert Position button pressed.

[image](#) Full size image



7. Educational Uses of JVAL

JVAL can be used in at least three possible ways in an educational environment. The visualization software can be used in tutorial settings, either for classroom demonstrations or for on-line tutorials. JVAL can also be used for active student learning via laboratory and homework activities. Finally, JVAL, when used in concert with a source-level debugger, provides an effective debugging tool for students working on projects.



7.1 Tutorials

The classical application of visualizations for learning data structures and algorithms is through enhancement of instructor-led demonstrations within the classroom. Two features of JVALL make it very appropriate and flexible in such a setting.

First, since any user-provided controller program can drive a JVALL visualization, the instructor can use the General Interactive Linked List Controller or design her own controller as appropriate for the instructional objectives.

The second JVALL feature that supports in-class demonstrations is the ability to load into JVALL a previously saved list from a file. This enables the instructor to use examples on complex list structures without the overhead of spending class time building the list.

In an on-line learning environment, JVALL can be controlled by tutorial software with appropriate visualizations occurring at specified places in the process. Such tutorials may or may not be interactive as the author of the controller determines. When the learning is not only on-line, but also distance learning, the use of the applet mode of JVALL is particularly helpful.



7.2 Active Student Learning

JVALL can be used as a tool to generate student activities that will enhance learning. Laboratory activities might include use of the General Interactive Linked List controller to have students manipulate linked lists and observe each operation, have students count events as they observe them to analyze the run-time for an algorithm, have students make comparisons of the different linked list models, and using JVALL to visualize the operations on a class, either student-written or instructor-provided, that uses the Java *LinkedList* class.

As an example of this, a *Stack* class has been built using the JVALL implementation of the Java *LinkedList* class and an interactive controller has been written to generate any possible operation on that class. A student might learn about the *Stack* class through JVALL visualization of the stack operations. Figure 12 shows a snapshot of this activity.

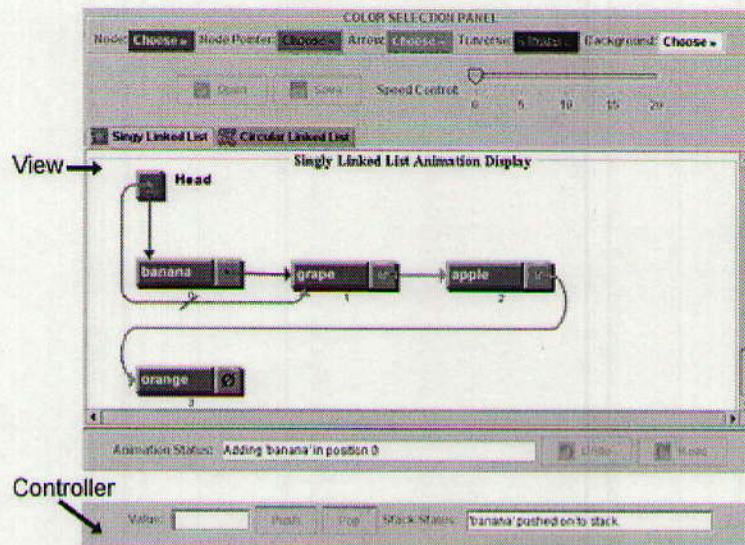




Figure 12. Bottom panel showing Stack Controller.

 Full size image
 An external link to the authors' Linked List Stack applet.

7.3 Debugging Student Programs

JVALL is particularly useful in assisting students while they debug programs that use the Java *LinkedList* class. In addition to the visualizations of the *LinkedList* operations, JVALL, in partnership with a source-level debugger, is an especially powerful tool for discovering logic errors. With this combination, students can step through the code with the debugger and watch the results of operations evolve within the JVALL window. This is a significant improvement over the use of the debugger alone because the student is not forced to follow a chain of references to realize the structure of a list.

8. Conclusions and Future Work

A number of animation strategies have been developed and implemented for use as teaching and learning tools. The qualities that an effective algorithm visualization package should possess have been classified by Cordova in [12]. Below, we apply these five criteria to JVALL.

- **Flexibility.** JVALL is flexible in the sense that it is capable of visualizing the execution of any program that uses the Java *LinkedList* class. JVALL can be used and represented in different ways to meet the needs of the controller program. JVALL can also animate applets as well as standalone applications.
- **Integration of algorithm text and visualization.** JVALL provides dynamic textual updates of the animation stages to the user and to the controller program. Because JVALL was built upon the Java *LinkedList* class and is independent of implementation, it is unable to provide code display of the linked list.
- **Ease of modification.** All the components of JVALL are fully


customizable. JVALL comes with a full color control panel for customizing all aspects of the animation display. The linked list nodes, pointers, arrows, traverse arrow, and background colors can be changed to suit the user's taste.

- **Execution control features.** JVALL's speed control and multiple undo's and redo's give the user the ability to rewind and adjust the speed of the animation to aid learning at a user's pace.

- **Support for animation of algorithms supplied by the user.** As illustrated in the program in Figure 2, a user can generate a visualization using any controller program that utilizes the Java *LinkedList* class.

According to these criteria, JVALL is a very effective tool for enhancing the learning of the algorithms and applications of linked lists.

Future enhancements to JVALL include the implementation of additional linked list models such as doubly linked and header lists. In addition, this same process will be used to provide animated visualizations of other Java Collection classes such as *ArrayList* and *TreeMap*.

 **Link** An external link to the authors' JVALL web site for downloading source code, general controllers, and documentation.

9. Acknowledgments

This work was supported in part by the National Science Foundation Research Experiences for Undergraduates program through grant #EIA-0097464.

10. References

- [1] Brown, M. Exploring algorithms using Balsa-II. *IEEE Computer* 12,5(1988), 14-36.
- [2] Stasko, J.T. TANGO: A framework and system for algorithm animation. *IEEE Computer* 23,9(1990), 27-39.
- [3] Naps, T.L. and Bressler, E. A multi-windowed environment for simultaneous visualization of related algorithms on the World Wide Web. *SIGCSE Bulletin* 30,1(1998), 277-281.
- [4] Pierson, W.C. and Rodger, S.H. Web-based animation of data structures using JAWAA. *SIGCSE Bulletin* 30,1(1998), 267-271.
- [5] Hundhausen, C.D., Douglas, S.A., and Stasko, J.T. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, to appear.
- [6] Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., and Vanninen, P. Animation of user algorithms on the web. *Proceedings of the 13th IEEE International Symposium on Visual Languages*, IEEE Computer Society Press, 1997, 360-367.

- [7] Lattu, M., Tarhio, J., and Meisalo, V. How a visualization tool can be used--evaluating a tool in a research & development project. 12th Workshop of the Psychology of Programming Interest Group, April 2000.
- [8] Baker, R.S., Boilen, M., Goodrich, M.T., Tamassia, R., and Stivel, B.A. Testers and visualizers for teaching data structures. *SIGCSE Bulletin* 31,1(1999), 261-265.
- [9] Rößling, G., Schüler, M., and Freisleben, B. The ANIMAL algorithm animation tool. 5th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2000, 37-40.
- [10] Crescenzi, P., Demetrescu, C., Finocchi, I., and Petreschi, R. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11,2(2000),125-150.
- [11] Bergin, J., Brodlie, K., Goldweber, M., Jiménez-Peris, R., Khuri, S., Patiño-Martínez, M., McNally, M., Naps, T., Rodger, and Wilson, J. An overview of visualization: its use and design. *Proceedings of the ACM SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, 1996, 192-200.
- [12] Cordova, J. A comparative evaluation of web-based algorithm visualization systems for computer science education, *Journal of Computing in Small Colleges* 14,3 (1999), 72-77.

▲ ***** End of Document *****

IMEJ multimedia team member assigned to this paper

Yue-Ling Wong