AlgAE
(Algorithm Animation Engine)
Reference Manual
Version 3.0

Steven J. Zeil
Old Dominion University
Dept. of Computer Science

August 20, 2011

Contents

Chapter 1

Introduction

1.1 What is AlgAE?

ALGAE is a framework for quick construction of algorithm animations. ALGAE is aimed at CS instructors who employ projection screen monitors, LCD overhead pads, or other devices to display computer video output in the classroom or who publish web content in support of their courses. ALGAE allows the instructor to take typical C++ or Java code from a course's textbook or lecture notes and, with relatively little effort, produce an "animated" version of that code. These animations will show the data as it is being manipulated by the code. Data is portrayed as a collection of labeled boxes connected by arrows (denoting pointers). Figure 1.1 shows a typical ALGAE animation in progress.

I have used ALGAE (and an earlier version, written in Pascal) for our CS2 course (which emphasizes program design, the use of ADT's, and some fundamental data structures) and for a course in Advanced Data Structures and Algorithms. I have used it with LCD overhead pads, external video projectors, and scan converters feeding into a broadcast TV signal. ALGAE animations have been an integral part of a data structures course that I offer to distance students via the internet for more than 10 years.

Version 3.0 of ALGAE is a major rewrite of the original system. It is easier to install and to use than the earlier versions.[1]

[1] On the other hand, direct animation of C++ code is no longer supported (though it is relatively easy to use Java to simulate animation of C++). The network socket-based architecture used to support C++ animation by Versions 1 and 2 was unwieldy — many people found it hard to install, and University network administrators are increasingly reluctant to open up ports for software that they themselves do not maintain.

I hope to restore C++ support in the near future.

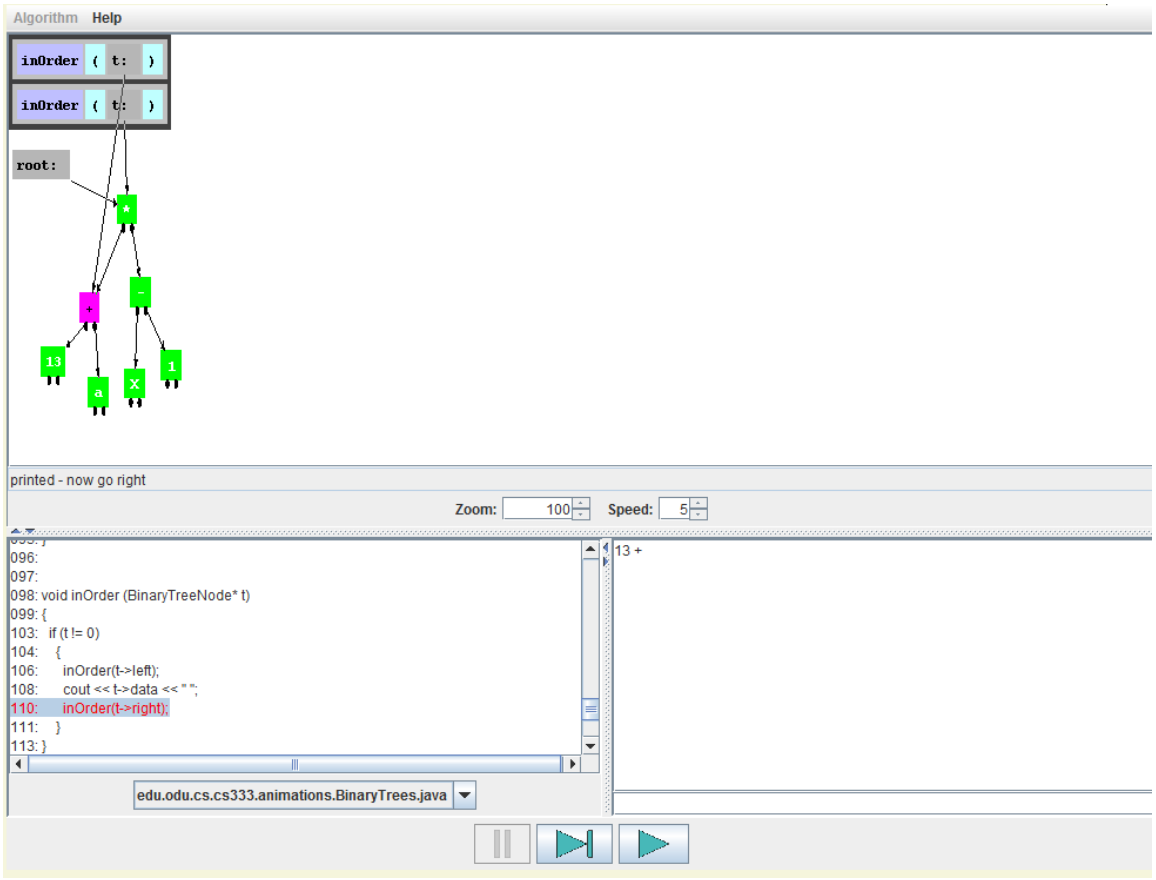

Figure 1.1: A Typical AlgAE Animation in Progress

1.2 Design Principles

ALGAEś design is based on the following principles:

Animations should be tied to real, running code. Yes, there are lots of general-purpose animation packages out there that can produce much prettier pictures. Or you can simply use a conventional drawing tool to present a series of "snapshots", slide by slide. Both of these approaches require you to script out, before the lecture, the exact sequence of operations you want to show. I find this takes more time, in the long run, than building an ALGAE animation. In particular, I found that small mistakes in a "script" often meant having to go back and redraw large numbers of frames.

Perhaps the biggest drawback to these scripted approaches is the lack of flexibility. You can walk nicely through a scripted presentation of the algorithm's execution, then a student raises a hand, and asks "but what if..." and takes you right out of your script. With ALGAE you simply re-run the algorithm with different inputs.

Animations should be tied to code from the course text and lecture notes. Instructors who are facile with graphics-rich languages and packages might consider simply writing a program to draw pictures that simulate the algorithm being discussed. But I suspect that this, also, will prove more time-consuming than using ALGAE. Furthermore, this alternative introduces the problem of keeping the the animated algorithm consistent with the code under discussion. It's counter-productive to have the animation do things in a different order than can be explained by students reading along in their notes on the algorithm.

It must be easy to create an animation from existing code. This is essential, because otherwise it would be more effective for an instructor to fall back on hand-drawn illustrations. Also, texts change frequently, and so, therefore, does the code to be presented to the students. (A year after I finished my Pascal animations, our Department changed our instructional programming language to C++. Since that change, I've used 3 different data structures texts, and many of my other texts have come out with new editions.)

The animations must be easy to run and control. When I'm lecturing, I don't need the distraction of trying to manage a complicated interface while still trying to keep my speech coherent. Also, students frequently ask if they can run the animation programs themselves while studying outside of class.

Some sort of automated layout is essential. In my earlier Pascal algorithm animator, each object had to be manually positioned using the mouse. This turned out to be terribly distracting, especially for data structures like AVL trees where the "natural" positions of objects changes regularly.

The instructor/student needs constant feedback about where the current execution point is in the algorithm. Without this, it's too easy to get confused and totally demolish the instructor's reputation for infallibility! ALGAE provides both a status area for instructor-designed messages, and an ability to synchronize a view of the source code with the current execution location. The ability to trace through the source code is shown in Figure 1.1. If a sufficiently large font size is selected for this trace, students can read the algorithm code off the screen, though I usually prefer to have them follow along on hard copy, using the trace window to keep myself on track.

You don't need to show lots of objects on the screen at once, but you do sometimes need to make them large. For instructional purposes, a couple dozen objects on the screen at once seems to be more than enough. Consequently, ALGAE does not place much emphasis on speedy rendering of large numbers of objects. On the other hand, for the text in these objects and their connecting pointers to be visible in a large lecture room or on a conventional TV monitor (conventional TV bandwidth imposes some pretty severe constraints on text presentation and line drawing), you often need to make everything quite large.

1.3 Platforms Supported

Version 3.0 of ALGAE software is written entirely in Java and should be executable on almost any system. Animations created using ALGAE can be distributed and run as standalone applications or posted as Java web applets.

The current version is compiled using Java 6. The project build is supported using Apache's `ant`[1].

Chapter 2

Installation

2.1 Obtaining ALGAE

ALGAE is available via anonymous ftp at `ftp.cs.odu.edu` in the directory `/pub/zeil/algae`.

2.2 Configuration and Installation

1. Unpack the ALGAE `.zip` file into a directory of your choice. For the sake of exposition, I will assume throughout this section that you have chosen to place this in a directory named *algae-root*.

2. You will need a copy of the Sun Java Help library, `jhbasic.jar` [6]. If one is not included in the *algae-root*`/AlgAE` directory, find a copy on the net and place it there.

3. In the *algae-root*`/AlgAE` directory, run

   ```
   ant
   ```

   This should build the `AlgAE.`*versionnumber*`.jar` file containing the Java engine.

2.3 Testing your installation

The installation includes various sample animations of both Java and `C++` code. The Java animations will be found in the *algae-root* directories other than the main *algae-root*`/AlgAE` directory.

You should be able to compile any of these by giving the command

```
ant
```

and to execute them by giving the command

```
ant run
```

Chapter 3

Running an Animation

When an ALGAE animation is launched, you will be presented with a blank animation window. From the `Algorithms` menu, you can select any of several operations you would like to perform, e.g., inserting an item into a binary search tree, removing an item, do a post-order traversal, etc.

At selected points in the algorithm, the program will pause and draw a picture of the current data state. The picture in Figure 1.1 is an example of this.

Boxes in the picture that represent data allocated on the heap can be repositioned by dragging them with the mouse, if you are unhappy with the automatic layout.

The controls below the main drawing area should be reminiscent of a CD/DVD player. You can move forward one step at a time, or move continuously without pausing. You can also "rewind" an animation to re-examine details that you might have missed.

That's basically all there is to it. More details on the available commands and options may be obtained via the "Help" menu.

3.1 Deploying an Animation

The most common way to deploy an animation is as an applet.

For this purpose, you will need 4 things:

1. The compiler ALGAE engine, `AlgAE.`*version*`.jar`
2. The Sun Java Help library, `jhbasic.jar`
3. A jar file (or collection of class files) representing your specific animation, and
4. An HTML file that invokes the animation as an applet.

   Typically, this will contain an applet element similar to the following

```
<applet
    code="yourAnimationMainClassName"
    archive="yourAnimationName.jar,
                ../AlgAE/AlgAE.3.0.jar,../AlgAE/jhbasic.jar"
    width=10 height=10
    alt="Is Java disabled on this browser?">
Oops.
This browser does not display Java applets.
  </applet>
```

Of course, you will need to adjust the paths to the ALGAE and `jhbasic` jars according to your own preference.

By default, ALGAE applets will pop up as a separate window when the page is visited. If you prefer to have the animation display within the page, add the `inline` parameter and give a sufficiently large height and width:

```
<applet
    code="yourAnimationMainClassName"
    archive="yourAnimationName.jar,
                ../AlgAE/AlgAE.3.0.jar,../AlgAE/jhbasic.jar"
    width=800 height=700
    alt="Is Java disabled on this browser?">
    <param name="inline" value="1"/>
Oops.
This browser does not display Java applets.
  </applet>
```

Place these in a webserver-accessible directory and publish the URL of the HTML page.

Chapter 4

Animating a Java Algorithm

This section presents the major capabilities of ALGAE via a series of examples.

4.1 Animation with Arrays

The first example will be relatively simple, because most of the rendering decisions will correspond to ALGAE's built-in defaults.

Our starting point will be the Java class in Listing 4.1.1, which contains three well-known sorting algorithms.

Listing 4.1.1: An ADT for Disjoint Sets

```
package edu.odu.cs.AlgAE.Demos;


public class Sorting0 {


//
//  Based on Malik, "C++ Programming [From Problem Analysis to Program Design]"
//       chapter 10
//

public static
void bubbleSort(int list[], int length)
{
    int temp;
    int iteration;
    int index;
```

```
    for (iteration = 1; iteration < length; iteration++)
    {
        for (index = 0; index < length - iteration; index++)
            if (list[index] > list[index + 1])
            {
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
            }
    }
}

public static
void selectionSort(int list[], int length)
{
    int index;
    int smallestIndex;
    int location;
    int temp;

    for (index = 0; index < length - 1; index++)
    {
            //Step a
        smallestIndex = index;

        for (location = index + 1; location < length; location++)
            if (list[location] < list[smallestIndex])
                smallestIndex = location;

            //Step b
        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}


public static
void insertionSort (int list[], int listLength)
{
    int firstOutOfOrder, location;
    int temp;

    for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                              firstOutOfOrder++)
```

```
        if (list[firstOutOfOrder] < list[firstOutOfOrder-1])
        {
            temp = list[firstOutOfOrder];
            location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
}

}
```

- You can place your code in any package you wish, or forgo a package entirely. I place my copy in `edu.odu.cs.AlgAE.Demos` because I wish to make it part of the ALGAE distribution. You should probably not use a more appropriate designation for your own work.

4.1.1 Setting Up An ALGAE Project

The first step is to set up a project structure. I recommend using `ant`[1] as the main project manager, though it's often convenient to do the bulk of the work in an `ant`-friendly IDE such as Eclipse [2].

1. Create a directory for the project. Within that project directory create a `src` directory to hold your Java code.

2. Then, within the project directory, create a `build.xml` file for `ant`. Listing 4.1.2 shows a possible structure.

Listing 4.1.2: Ant Build File, build.xml

```
<project name="AlgAE Reference Manual" default="buildJar"
         basedir="." xmlns:ac="antlib:net.sf.antcontrib">
  <description>
    AlgAE reference manual and sample code
  </description>



  <!-- set global properties for this build -->
```

```
<property name="project" value="SortingExample" />
<property name="mainClass"
          value="edu.odu.cs.AlgAE.Demos.SortingDriver" />

<property name="version" value="3.0" />
<property name="AlgAE.dir" location="../AlgAE"/>

<property name="compile.dest" location="src"/>


<path id="compile.classpath">
  <pathelement location="${AlgAE.dir}/AlgAE.${version}.jar"/>
  <pathelement location="${AlgAE.dir}/jhbasic.jar"/>
</path>

<path id="execute.classpath">
       <pathelement location="${project}.jar"/>
       <pathelement location="${AlgAE.dir}/AlgAE.${version}.jar"/>
       <pathelement location="${AlgAE.dir}/jhbasic.jar"/>
</path>


<target name="compile">
  <mkdir dir="${compile.dest}" />
  <javac srcdir="src" destdir="${compile.dest}"
         classpathref="compile.classpath"
         excludes="**/Test*.java" debug="true"
         includeantruntime="false">
    <compilerarg value="-Xlint:deprecation"/>
  </javac>
</target>


<target name="buildJar" depends="compile">
  <jar destfile="${project}.jar">
    <fileset dir="${compile.dest}"></fileset>
    <manifest>
      <attribute name="Main-Class"
                 value="${mainClass}"/>
    </manifest>
  </jar>
</target>

<target name="run" depends="buildJar">
```

```
    <java classname="${mainClass}"
          fork="true"
          classpathref="execute.classpath">
    </java>
  </target>

</project>
```

- Change the project name and description as desired.
- The `project` property is used to name the final Jar file produced as the output.
- The `mainClass` property names a Java class that will be the default animation driver to be executed in the produced Jar file. It will also be run by the

  ```
  ant run
  ```

  command.

  Note that a project (and hence the project's Jar file) can actually have several animations. This property establishes only a default selection.
- The `version` and `AlgAE.dir` properties are sued to locate the already compiled ALGAE library and the accompanying `jhbasic.jar` file.
- The remainder of the file can, for most simple animations, be left unchanged.

If setting up a project directory with Eclipse [2] or other project managers, be aware that a deliberate design decision is to place the compiled class files into the same directories as the source code. Although this practice is often discouraged in large projects, individual ALGAE animations are seldom large. More importantly, the feature of displaying the code being animated exploits this decision to locate the relevant source code file.

3. Next, create an HTML file (or several files) to invoke the animation(s) in the project, as per the guidelines in section 3.1. Listing 4.1.3 shows an example of such a file. I typically include basic instructions to the students telling them what kinds of activities they might employ to explore the animation.

Listing 4.1.3: HTML Applet Support

```
<html>
<head>
   <title>
     CS333 Sorting Algorithms
   </title>
</head>
<body text="#330000" bgcolor="#F5F5DC">
```

```
<center>
  <h1>
    CS333 Sorting Algorithms
  </h1>
</center>

<p>Quick instructions:
<ul>
  <li>
    Select a desired algorithm from the
    <b>
      Algorithm
    </b> menu.
    <ul>
      <li>
        Start by generating an array. Keep it small (5-10 elements) -
        you'll be examining the sorting algorithm behavior in fine
        detail.
      </li>
      <li>
        Once the array is generated, choose a sorting algorithm.
      </li>
    </ul>
  </li>

  <li>
    When execution pauses inside of an algorithm, press the "step" or
    "play" buttons to resume.
  </li>

  <li>
    Observe how each algorithm behaves on random arrays, reversed
    arrays, and arrays that you have already sorted.
  </li>

</ul>
</p>
<p>
  More detailed help is available from the <b> Help</b>Help menu.
</p>

<center>
  <applet
     code="edu.odu.cs.cs333.animations.CS333SortingAnimation"
```

```
      archive="CS333Sorting.jar,
               ../AlgAE/AlgAE.3.0.jar,../AlgAE/jhbasic.jar"
      width=10 height=10
      alt="Is Java disabled on this browser?">
     Oops.
     This browser does not display Java applets.
  </applet>
</center>

<hr/>
<center>
  <hr width="75%"></center>

<center>Email me at
  <i>
    <a href="mailto:zeil@cs.odu.edu">
      zeil@cs.odu.edu
    </a>
  </i>
</center>

</body>
</html>
```

4. Finally, add (or build) your animation source code within the `src` directory. As with all Java code, if your classes are declared inside packages, you will need to construct the directory structure corresponding to the package structure.

4.1.2 Creating the Driver

Each ALGAE animation starts its execution from a *driver*. The driver must be a subclass of `edu.odu.cs.zeil.AlgAE.Animation`. `Animation` itself is a subclass of `JApplet` and therefore can be run as a web applet. It also, however, provides support for being run as a standalone application. This is often more convenient during development and debugging.

The basic structure of a driver is shown below:

```
package edu.odu.cs.AlgAE.Demos;


import edu.odu.cs.zeil.AlgAE.Animation;
import edu.odu.cs.zeil.AlgAE.Server.MenuFunction;

public class SortingDriver extends Animation {
```

```
    public SortingDriver() {
        super("Sorting Algorithms", true);
    }

    @Override
    public String about() {
        return "Demonstration of Sorting Algorithms,\n" +
               "prepared for CS 333, Advanced Data Structures\n" +
               "and Algorithms, Old Dominion University\n" +
               "Summer 2011";
    }


    ⋮
    global data structures
    ⋮


    @Override
    public void buildMenu() {
        ⋮
        menu function registrations
        ⋮
    }


    public static void main (String[] args) {
        SortingDriver demo = new SortingDriver();
        demo.runAsMain();
    }
}
```

Several things are worth noting about the above code.

- You can place your code in any package you wish, or forgo a package entirely. You can name this class anything you wish.
- The imports shown here will probably be required. Others may be needed as well.
- The argument of the `super` call within the constructor is used to supply a title for any windows opened by the animation.

- The `about()` function supplies a short message that can be accessed from the Help menu in the running animation.
- The `buildMenu()` function is the heart of the driver. it is discussed below in section 4.1.2.
- As noted earlier, animations can be run as applets or, with the `main()` function shown here, as a standalone application.

Planning the Menu

.

The "Algorithms" menu for an animation is where the user goes to launch the ADT functions. Typically, this menu will contain one item for each of the major ADT operations, plus one or more items for creating new instances of or reinitializing an instance of the ADT.

Looking again at Listing 4.1.1, we can see that this ADT is fairly simple. We can expect to have three menu items, one to invoke each of the the three sorting algorithms. In addition, we will want to add menu items to set up an array for sorting. I have opted for one menu item to create an randomly arranged array, and one to create a reversed array.

These menu items are created in the driver's `buildMenu()` function (Listing 4.1.4).

Listing 4.1.4: Creating the Algorithms Menu

```
package edu.odu.cs.AlgAE.Demos;


import edu.odu.cs.zeil.AlgAE.Animation;
import edu.odu.cs.zeil.AlgAE.Server.MenuFunction;

public class SortingDriver0 extends Animation {

        public SortingDriver0() {
                super("Sorting Algorithms", true);
        }

        @Override
        public String about() {
                return "Demonstration of Sorting Algorithms,\n" +
                                "prepared for CS 333, Advanced Data Strcutures\n" +
                                "and Algorithms, Old Dominion University\n" +
                                "Summer 2011";
        }


        private int[] array = new int[8];
```

```
@Override
public void buildMenu() {

        registerStartingAction(new MenuFunction() {
                @Override
                public void selected() {
                        generateRandomArray(array.length);
                }
        });

        register ("Generate a random array", new MenuFunction() {
                @Override
                public void selected() {
                        generateRandomArray(array.length);
                }
        });

        register ("Generate a reversed array", new MenuFunction() {
                @Override
                public void selected() {
                        generateReverseArray(array.length);
                }
        });

        register ("Bubble Sort", new MenuFunction() {
                @Override
                public void selected() {
                        Sorting.bubbleSort (array, array.length);
                }
        });

        register ("Selection Sort", new MenuFunction() {
                @Override
                public void selected() {
                        Sorting.selectionSort (array, array.length);
                }
        });

        register ("Insertion Sort", new MenuFunction() {
                @Override
                public void selected() {
                        Sorting.insertionSort (array, array.length);
```

```
                }
            });

    }


    private void generateRandomArray(int n)
    {
        if (n != array.length) {
            array = new int[n];
        }
        for (int i = 0; i < n; ++i) {
            array[i] = (int)((double)(2*n) * Math.random());
        }

    }

    private void generateReverseArray(int n)
    {
        if (n != array.length) {
            array = new int[n];
        }
        array[n-1] = (int)(3.0 * Math.random());
        for (int i = n-2; i >= 0; --i) {
            array[i] = array[i+1] + (int)(3.0 * Math.random());
        }

    }




    public static void main (String[] args) {
        SortingDriver0 demo = new SortingDriver0();
        demo.runAsMain();
    }

}
```

- Each Algorithm menu item is created by calling `register` with a string to appear in the menu and a `MenuFunction` object for which a `selected()` function carries the actions we want performed when that item is chosen.
  - Three of these simply invoke the three sorting algorithms on the private array `array`.

  - The other two invoke a pair of functions to fill the array with different patterns of random numbers.
  - The order in which these `MenuFunction` objects are registered determines the order in which they will appear in the Algorithms menu.

- A special registration function, `registerStartingAction`, is used to designate a function to be executed before the user is allowed to select anything from the Algorithms menu.

4.1.3 Displaying Global Data

Although it would be possible to run the animation as already developed, doing so would be profoundly boring because, although the desired menu items would be available, there's no way yet to see the effect of these.

We would like to be sure that the array we have created will be visible at all times while the animation is running.

The rules for determining exactly what data objects will be shown during a running animation are:

1. Objects that are registered as "global" will be shown.
2. Objects that are registered as parameters to a function or as local "variables" to a function will be shown during any breakpoints within that function.
3. Objects will be shown if they are registered as "components" of another object that is already being shown.
4. Objects will be shown if they are registered as "connected" from another object that is already being shown.

At the moment, we have not done any of these things, so no data objects will be shown.

Because we want `array` to be shown at all times, no matter what function is executing, we will register it as global:

```
    public void buildMenu() {

        registerStartingAction(new MenuFunction() {
            @Override
            public void selected() {
                generateRandomArray(array.length);
                globalVar("list", array);
            }
        });
```

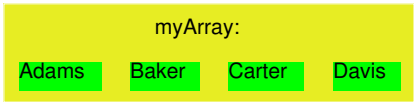

Figure 4.1: A Sorted Array of Strings

.

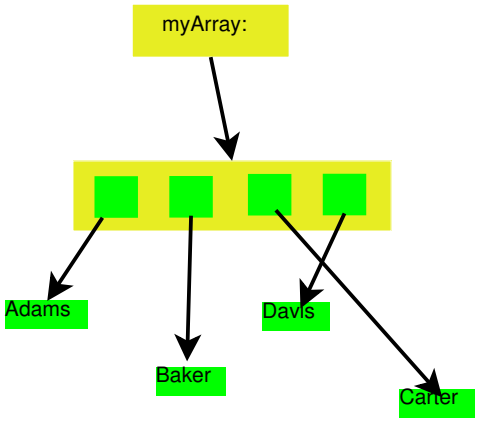

Figure 4.2: A More Honest Sorted Array of Strings

.

Global variables can be registered from within any `MenuFunction` (or within any function called by a `MenuFunction`). `globalVar()` requests that an object be shown directly. A variant, `globalRefVar()` requests that a pointer/arrow be drawn that connects to the object. Each of these functions receives both a string to use as a name for the object when it is drawn and the reference to the actual object itself.

This driver can be executed (within the reference manual project directory) as `SortingDriver1`.

4.1.4 The Fine Art of Fibbing

One of the practical considerations in preparing an animation for students is to realize that too much honesty is not always the best policy. Suppose, for example, that instead of sorting integers, we had decided to sort strings. We might envision a picture of a sorted array as shown in Figure 4.1.

But any good Java programmer knows that Figure 4.2 is a much more accurate portrayal of an array of strings.

Nonetheless, if I wanted to explain a sorting algorithm I would prefer the simpler picture in Figure 4.1. The extra pointers are a distraction. Trying to visualize the movement of elements in

the array as shifting pointers rather than as direct rearrangements of the data is simply not as clear.

ALGAE provides several mechanisms by which an animation designer may opt to "fib" to viewers in support of better instructional clarity. One of the most pervasive of these is in allowing the animation designer to choose whether to present variables and data members of objects "inline" or as connections/references.

Another way in which a designer may "fib" to viewers is to selectively rewrite the source code displayed during the animation, as we will see shortly.

4.1.5 Inserting Breakpoints

The animation at the end of subsection 4.1.3 shows only the final outcome of executing each function. We can see, for example, that each of the sorting algorithms does indeed sort the array, but we are not able to follow the intermediate steps of each sorting algorithm.

The solution is to introduce breakpoints into the sorting algorithms. More specifically, there are two steps:

1. Create a (simulated) activation record for each function
2. Use that activation record to create breakpoints.

An activation record is created by the call `Animation.activate(...)`. A breakpoint is created by the activation record's member function `breakHere(...)`. Here, for example, we can see the process of creating an activation record for the bubble sort function and setting a breakpoint at the very start of that function:

```
public static //!
void bubbleSort(int list[], int length)
{
  ActivationRecord arec = Animation.activate(Sorting2.class); //!
  arec.breakHere("starting bubble sort"); //!
```

- The parameter to the `activate` function fulfills two purposes:
  1. It helps the source code viewer to locate the appropriate file of source code.
  2. If it is supplied with a `this` reference (in a non-static member function), then that parameter is also displayed as part of the visible rendering of the activation stack.

     In this case, the function being animated is static, so we supply the class object instead of a `this` reference.
- The parameter to the `breakHere()` function is a descriptive string that is displayed on the status line when the breakpoint is encountered. Use these strings to establish a running commentary on the progress of the algorithm:

```
public static//!
void bubbleSort(int list[], int length)
{
    ActivationRecord arec = Animation.activate(Sorting2.class);//!
    arec.breakHere("starting bubble sort").show();//!
    int temp;
    int iteration;
    int index;

    for (iteration = 1; iteration < length; iteration++)
    {
        arec.breakHere("start a pass over the array");//!
        for (index = 0; index < length - iteration; index++)
        {//!
            arec.breakHere("compare list[index] to list[index+1]");//!
            if (list[index] > list[index + 1])
            {
                arec.breakHere("list[index] and list[index+1] are out of order - swap th
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
            }
        }//!
        arec.breakHere("completed this pass over the array");//!

    }
    arec.breakHere("completed bubble sort");//!
}
```

Many of the statements in the above listing end with the comment marker `//!`. This marker is a signal for selective rewriting by the source code viewer.

- For any source code line that contains a `//!`, the source code viewer will not display the part of that line from the start of the line up to and including the `//!`.
- If the remaining part of the line after a `//!` has no non-whitespace characters, then the entire line is omitted from the source code viewer.

The statements that create the activation records and the breakpoints are terminated with a `//!` marker so that they are never shown to the students running the animation. This is an example of pnambic (Pay No Attention to the Man BehInd the Curtain) behavior [4], or, as I have described it earlier, the fine art of fibbing to the students for instructional purposes.

This version of the system (including breakpoints in all three sorting algorithms) can be executed (within the reference manual project directory) as `SortingDriver2`. You can see that the individual changes to the array are visible at each step.

4.1.6 Displaying Local Data

The most recent version of the animation still does not provide enough information to allow students to understand what is happening. The primary problem now is that the array is the *only* data being shown. But understanding the behavior of these algorithms depends on knowledge of the values of the various indices, loop counters, and temporary variables used within the algorithm.

To add local data to the display, we register desired variables or expressions with the breakpoint. There are two kinds of local data to be displayed: parameters and variables.

For example, this code

```
public static //!
    void bubbleSort(int list[], int length)
    {
     ActivationRecord arec = Animation.activate(Sorting3.class); //!
     arec.refParam("list", list).param("length", length).breakHere("starting␣bubble
```

adds two parameters to the display of the breakpoint.

- In this case, we have opted to display the `list` parameter as a pointer (because the array itself is already being displayed as a global variable. The `length` parameter, however, is displayed inline (Figure 4.3).
- The two parameters are labeled “list” and “length” (matching the formal parameter names of the function) and the actual values of these parameters are also passed in each call.
- It’s convenient, but not necessary to combine all the breakpoint manipulation into one line. It could be broken up:

```
public static //!
    void bubbleSort(int list[], int length)
    {
     ActivationRecord arec = Animation.activate(Sorting3.class); //!
     arec.refParam("list", list); //!
     arec.param("length", length)//!
     arec.breakHere("starting␣bubble␣sort"); //!
```

As we advance further into the algorithm, we add the second type of local display – variables:

```
public static //!
    void bubbleSort(int list[], int length)
```

Figure 4.3: Displaying Parameters of a Call

```
{
    ActivationRecord arec = Animation.activate(Sorting3.class);//!
    arec.refParam("list", list).param("length", length).breakHere("starting␣bubble␣sort")
    int temp;
    int iteration;
    int index;

    for (iteration = 1; iteration < length; iteration++)
    {
        arec.var("iteration", iteration);//!
        arec.breakHere("start␣a␣pass␣over␣the␣array");//!
        for (index = 0; index < length - iteration; index++)
        {//!
            arec.var("index", index);//!
            arec.breakHere("compare␣list[index]␣to␣list[index+1]");//!
            if (list[index] > list[index + 1])
            {
                arec.breakHere("list[index]␣and␣list[index+1]␣are␣out␣of␣order␣-␣swap␣them
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
             }
        }//!
        arec.breakHere("completed␣this␣pass␣over␣the␣array");//!

    }
    arec.breakHere("completed␣bubble␣sort");//!
}
```

- The `var()` calls are similar to those for registering parameters, but variables will be displayed outside of the call stack.
- Again, the two parameters for each variable registration consist of a name for the variable to display and a value to be displayed (Figure 4.4).
- Although not used here, we also have the option of displaying a local variable as a reference

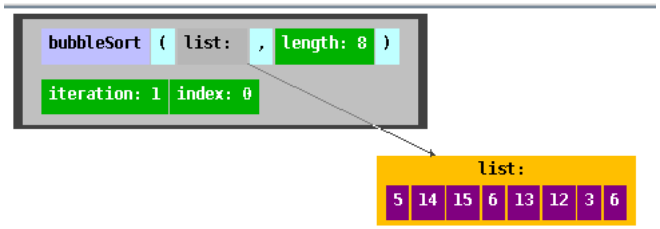

Figure 4.4: Displaying Local Variables

arrow to a separately allocated object. The call for doing so is called `refVar()`.

What, precisely, does the ALGAE syetem "remember" when a parameter or variable is registered? It remembers the reference value stored in the registered value. So, when we say, for example,

```
public static //!
    void bubbleSort(int list[], int length)
    {
        ActivationRecord arec = Animation.activate(Sorting3.class); //!
        arec.refParam("list", list).param("length", length).breakHere("starting␣bubb
```

what is registered for the parameter "list" is the reference (address) stored in the variable `list`. For the parameter "length", things are a touch more complicated. `length` is a primitive (int) variable. It does not contain a refernece. However, when `param(...)` or the other registration functions are called, Java will automatically convert the `int` value to an `Integer` object, and it will be the address of that temporary variable that is remembered.

All this is fine as long as the value of the primitive does not change (or, in general, as long as the reference stored in a parameter or variable does not change). So `list` is fine – its value changes but its address does not. `length` is also fine – the sorting algorithm does not change the value of this primitive.

Looking down a little further, though we see the local variables `iteration` and `index`, which *do* change value repeatedly.

```
public static //!
    void bubbleSort(int list[], int length)
    {
        ActivationRecord arec = Animation.activate(Sorting3.class); //!
        arec.refParam("list", list).param("length", length).breakHere("starting␣bub
        int temp;
        int iteration;
```

```
        int index;

        for (iteration = 1; iteration < length; iteration++)
        {
            arec.var("iteration", iteration);//!
            arec.breakHere("start a pass over the array");//!
            for (index = 0; index < length - iteration; index++)
            {//!
                arec.var("index", index);//!
                arec.breakHere("compare list[index] to list[index+1]");//!
                if (list[index] > list[index + 1])
                {
                    arec.breakHere("list[index] and list[index+1] are out of order - swap th
                    temp = list[index];
                    list[index] = list[index + 1];
                    list[index + 1] = temp;
                }
            }//!
            arec.breakHere("completed this pass over the array");//!

        }
        arec.breakHere("completed bubble sort");//!
    }
```

When a primitive changes value or an object variable changes its reference value, then we must re-register it using the same param/var function and the same label. The new value then replaces the old one. In this case, `iteration` and `index` are re-registered at the start of each iteration of their respective loops, which corresponds to the only times at which their values change.

One last detail remains to be considered. Once registered for display, a parameter or variable will normally be displayed until execution returns from the function. That's not always appropriate. For example, the last registered values of `iteration` and `index` would be displayed at the final, "completed bubble sort" although, technically, neither variable exists outside of their respective loops. You can designate a more limited lifetime for displayed variables by creating and destroying a "scope" over the relevant portion of the program.

```
public static//!
    void bubbleSort(int list[], int length)
    {
        ActivationRecord arec = Animation.activate(Sorting3.class);//!
        arec.refParam("list", list).param("length", length).breakHere("starting bubble sort"
        int temp;
        int iteration;
        int index;
```

```
        for (iteration = 1; iteration < length; iteration++)
        {
            arec.pushScope(); //!
            arec.var("iteration", iteration); //!
            arec.breakHere("start a pass over the array"); //!
            for (index = 0; index < length - iteration; index++)
            {//!
                arec.pushScope(); //!
                arec.var("index", index); //!
                arec.breakHere("compare list[index] to list[index+1]"); //!
                if (list[index] > list[index + 1])
                {
                    arec.breakHere("list[index] and list[index+1] are out of order
                    temp = list[index];
                    list[index] = list[index + 1];
                    list[index + 1] = temp;
                }
                arec.popScope(); //!
            }//!
            arec.breakHere("completed this pass over the array"); //!

            arec.popScope(); //!
        }
        arec.breakHere("completed bubble sort"); //!
    }
```

Any new variables introduced after a `pushScope` call are forgotten upon execution of the matching `popScope` call.

This version of the system (including parameters and local variables in all three sorting algorithms) can be executed (within the reference manual project directory) as `SortingDriver3`. This is now an animation that could be useful to students trying to learn these algorithms.

4.1.7 Decoration

Although potentially useful, the prior version of the sorting demo is still not quite as clear as we might wish. Understanding what is going on with array manipulation requires mentally correlating the values of various indices against the data in corresponding positions in the array. Even with relatively small arrays, it can be easy to slip up.

Sometimes it is useful to add explicit *decoration* to a breakpoint that can focus the viewer's attention on specific data elements.

Two useful ways to add decorations are by adding connections to the diagram or by exploiting color changes.

Decorative Connections

The package `edu.odu.cs.zeil.AlgAE.Utilities` contains, among other things, a number of *wrapper* (a.k.a. *adapter*) classes [3] for data that "indexes" into other collections. These include wrappers for Java iterators and, the one that we will use, an `Index` wrapper for integers that are used to index into one or possibly two different arrays or `Iterable` structures.

An index is created via a constructor that takes two parameters. The first is the integer value of the index, and the second is the array or `java.util.Iterable` container that it indexes into. An `Index` is rendered like an integer value but with a colored arrow pointing to the component of the array/container indicated by that position value.

Here I have decorated both the `index` and `smallestIndex` variables in the bubble sort:

```
    public static//!
    void bubbleSort(int list[], int length)
    {
        ActivationRecord arec = Animation.activate(Sorting3.class);//!
        arec.refParam("list", list).param("length", length).breakHere("starting␣bubble␣sort'
        int temp;
        int iteration;
        int index;

        for (iteration = 1; iteration < length; iteration++)
        {
            arec.pushScope();//!
            arec.var("iteration", new Index(iteration, list));//!
            arec.breakHere("start␣a␣pass␣over␣the␣array");//!
            for (index = 0; index < length - iteration; index++)
            {//!
                arec.pushScope();//!
                arec.var("index", new Index(index, list));//!
                arec.breakHere("compare␣list[index]␣to␣list[index+1]");//!
                if (list[index] > list[index + 1])
                {
                    arec.breakHere("list[index]␣and␣list[index+1]␣are␣out␣of␣order␣-␣swap␣th
                    temp = list[index];
                    list[index] = list[index + 1];
                    list[index + 1] = temp;
                }
                arec.popScope();//!
            }//!
            arec.breakHere("completed␣this␣pass␣over␣the␣array");//!

            arec.popScope();//!
        }
```

```
        arec.breakHere("completed bubble sort"); //!
    }
```

so that each will be shown pointing to the specific element of the `list` array that is being referenced.

This version of the sorting demo can be executed (within the reference manual project directory) as `SortingDriver4`. This is, by and large, the version that I actually distributed to my class.

Decorative Color

Color changes can also be an effective way to draw a viewer's attention to a specific group of objects. The `ActivationRecord` class supports operations `highlight(object)` and `highlight(object color)` to create temporary color changes. Their effects will end upon returning from the current function and will be suspended during calls from that function. (Permanent color changes are discussed later in section 4.2.)

For example, in the `insertionSort`, we might consider highlighting the element being compared against the temp value and also using a fixed color to distinguish the already sorted portion of the array from the portion that has not been checked:

```
public static //!
    void insertionSort (int list[], int listLength)
    {
        ActivationRecord arec = Animation.activate(Sorting3.class); //!
        arec.refParam("list", null).param("listLength", listLength); //!
        arec.breakHere("starting insertion sort"); //!
        int firstOutOfOrder, location;
        int temp;

        for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                                  firstOutOfOrder++)
        {//!
            arec.var("firstOutOfOrder", new Index(firstOutOfOrder, list)); //!
            arec.breakHere("move list[firstOutOfOrder] into place"); //!
            if (list[firstOutOfOrder] < list[firstOutOfOrder-1])
            {
                temp = list[firstOutOfOrder];
                location = firstOutOfOrder;
                arec.var("temp",temp).var("location", new Index(location, list)); //
                arec.breakHere("temp holds the value we want to insert"); //!

                do
                {
                    arec.highlight(list[location-1]); //!
                    arec.breakHere("shift an element up to make room"); //!
```

```
                    list[location] = list[location - 1];
                    arec.breakHere("then move to the next lower element"); //!
                    arec.highlight(list[location-1]); //!
                    location--;
                    arec.var("location", new Index(location, list)); //!
                    }
                while (location > 0 && list[location - 1] > temp);

                arec.breakHere("Now we know where to insert temp"); //!
                list[location] = temp;
                arec.highlight(list[location], Color.lightGray); //!
            }
            arec.breakHere("Move to the next unordered element"); //!
        }//!
        arec.breakHere("Completed insertion sort"); //!
    }
```

- If no color parameter is specified, the `highlight` function "inverts" the color of the object: i.e., if the RBG colors are expressed on a $0.0 \ldots 1.0$ scale, then an object of color (r, g, b) is changed to color $(1 - r, 1 - g, 1 - b)$.
  - Consequently, highlighting the same object a second time actually returns it to its original color.
- Alternatively, an explicit color can be selected in a highlight parameter.

Unfortunately, the change shown here will actually have no visible effect. That's because all renderable objects actually need to **be** objects. When we pass an `int` or other non-class primitive, they are converted to a corresponding `Integer` or other container object. That converted object is what is actually being shown, and because we don't have a reference to that object, we can't pass it to a `highlight` call.

A possible solution is to actually change the array supplied by the driver from an array of `int` to and array of `Integer`:

```
public class SortingDriver5 extends Animation {

    public SortingDriver5() {
        super("Sorting Algorithms", true);
    }

    @Override
    public String about() {
        return "Demonstration of Sorting Algorithms,\n" +
```

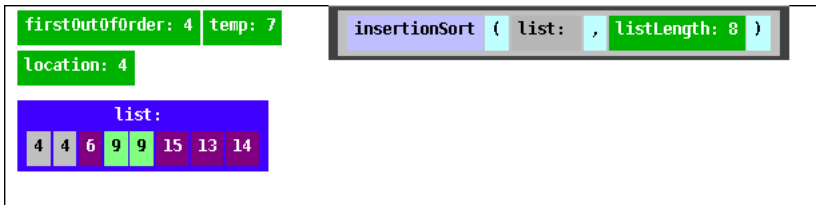

Figure 4.5: Unintended Aliasing

```
                "prepared for CS 333, Advanced Data Structures\n" +
                "and Algorithms, Old Dominion University\n" +
                "Summer 2011";
    }

    private Integer[] array = new Integer[8];
      ⋮
```

This, however, requires that the sorting functions be rewritten to work with an array of `Integer` as well. The resulting changes are probably not something we want the viewers to see, so we must engage in some selective rewriting of the source code:

```
public class Sorting5 {//!
    ⋮
void bubbleSort(Integer list[], int length)//!  void bubbleSort(int list[], int len
    ⋮
void selectionSort(Integer list[], int length)//!      void selectionSort(int list
    ⋮
void insertionSort (Integer list[], int listLength)//!  void insertionSort (int lis
    ⋮
}//!
```

This is easy enough, but it also runs into problems on some(?) Java engines. You will almost certainly discover that whenever one element in the array is highlighted, any elements containing the same value will also be highlighted (See Figure 4.5).

The problem is that, despite being rendered as distinct components of the array, the array is actually holding references to the actual values on the heap, and, in this case, some pairs of those are actually shared references to the same object. Consequently, highlighting the one object shared by two different array elements causes all of those array elements to appear highlighted.

Now that is a common problem that can arise when designing an animation, but, if you examine the Sorting Demo code closely, you may be at a loss to find the reason for this sharing. In fact, there's some rather strange behavior associated with the Java `Integer` class. The program in Listing 4.1.5 reveals that some sort of caching is used when `int` values are converted implicitly to `Integer`, with the result that `Integer` containers holding equal `int` values will often occupy the same address.

Listing 4.1.5: Integers Are Shared Unexpectedly

```
public class IntegerAliasing {

        /**
         * A check to see how often the Integer class
         * actually returns distinct objects
         *
         * @param args
         */
        public static void main(String[] args) {
                Integer[] a = new Integer[100];
                Integer[] b = new Integer[100];
                for (int i = 0; i < 100; ++i)
                        a[i] = i;
                for (int i = 0; i < 100; ++i)
                        b[i] = i;

                int count = 0;
                for (int i = 0; i < 100; ++i)
                        if (a[i] == b[i]) {
                                // a[i] and b[i] contain the same address/reference
                                ++count;
                        }
                System.out.println ("The two arrays share " + count + " elements.");
        }

        // Output is typically: The two arrays share 100 elements.
}
```

When run with Oracle Java 1.6, this program produces the output

```
The two arrays share 100 elements.
```

To counter this problem, the `edu.odu.cs.zeil.AlgAE.Utilities` contains a `DiscreteInteger` container in which no such sharing will occur. `DiscreteIntegers` are also mutable, making them a better fit for many algorithms. On the other hand, they do not get the benefit of implicit

conversion to and from `int`, so the use of `DiscreteInteger` would require more rewriting of the sorting algorithms and still more selective rewriting of the source code.

In practice, I generally find that decoration of algorithms takes far more time and effort than does getting a basic animation with breakpoints and rendering of data. You therefore want to be careful in pursuing decorations, restricting their use to situations where they truly enhance the instructional role of the animation.

4.2 Rendering Your Own Structures

The sorting demo that formed the basis of the previous section had a major simplification that won't always be available. The only data being drawn were primitives such as `int` values and arrays. ALGAE "knows" how to render these types (and a good handful of other types including `Strings` and anything that implements `Iterable`.

In this section, we will consider the problem of portraying a new class type for which no acceptable set of rendering rules has yet been provided.

The starting point for this discussion will be code for inserting an element into a singly linked list:

```
package edu.odu.cs.AlgAE.Demos;//!

public class SinglyLinkedLists {


  class nodeType
  {
      int info;
      nodeType link;

      public nodeType()
      {
          link = null;
      }
  }


  private nodeType head = null;


  public void addToFront (int k)
  {
     nodeType newNode = new nodeType();
     newNode.link = head;
     newNode.info = k;
```

```
      head = newNode;
   }

   public void insert (nodeType p, int value)
   {
      nodeType newNode = new nodeType();
      newNode.info = value;
      newNode.link = p.link;
      p.link = newNode;
   }

}
```

In this example, we will be interested only in demonstrating the process of inserting a node into a linked list, so the driver will provide only a single menu item:

Listing 4.2.6: Driver For Singly-Linked List Insertion

```
package edu.odu.cs.AlgAE.Demos;


import edu.odu.cs.zeil.AlgAE.Animation;
import edu.odu.cs.zeil.AlgAE.Server.MenuFunction;

public class SLLInsertionDriver0 extends Animation {

        public SLLInsertionDriver0() {
                super("Linked List Insertions", true);
        }

        @Override
        public String about() {
                return "Demonstration of Linked Lists Algorithms,\n" +
                                "prepared for CS 333, Programming and Problem\n" +
                                "Solving in C++, Old Dominion University\n" +
                                "Fall 2010";
        }

        SinglyLinkedLists0 sll = new SinglyLinkedLists0();




        @Override
        public void buildMenu() {
```

```
                register ("Insert a node", new MenuFunction() {

                        @Override
                        public void selected() {
                                generateLL();
                                sll.insert(sll.head.link, 50);
                        }

                });

        }

        public void generateLL()
        {
                sll.head = null;
                sll.addToFront(76);
                sll.addToFront(34);
                sll.addToFront(65);
                sll.addToFront(45);
        }




        public static void main (String[] args) {
                SLLInsertionDriver0 demo = new SLLInsertionDriver0();
                demo.runAsMain();
        }

}
```

We can also start with a set of breakpoints in the insertion function.

Listing 4.2.7: Singly-Linked List Insertion Breakpoints

```
package edu.odu.cs.AlgAE.Demos;//!

import edu.odu.cs.zeil.AlgAE.ActivationRecord;
import edu.odu.cs.zeil.AlgAE.Animation;

public class SinglyLinkedLists0  {


        class nodeType
        {
```

```
            int info;
            nodeType link;

            public nodeType()
            {
                    info = -999;//!
                    link = null;
            }

    }

    public nodeType head = null;


    public void addToFront (int k)
    {
            nodeType newNode = new nodeType();
            newNode.link = head;
            newNode.info = k;
            head = newNode;
    }



    public void insert (nodeType p, int value)
    {
            ActivationRecord arec = Animation.activate(this);//!
            arec.refParam("p", p).param("value", value);//!
            arec.breakHere("starting insertion");//!
            nodeType newNode = new nodeType();//!     nodeType *newNode = new nodeType;
            arec.refVar("newNode", newNode);//!
            arec.breakHere("allocated new node");//!
            newNode.info = value;
            arec.breakHere("inserted data into new node");//!
            newNode.link = p.link;//!     newNode->link = p->link;
            arec.breakHere("make new node point to p's successor");//!
            p.link = newNode;//!     p->link = newNode;
            arec.breakHere("make p point to the new node");//!
            arec.breakHere("Insertion has been completed");//!
            arec.breakHere("Trace the 'next' links and see for yourself that ...");//!
            arec.breakHere("... the new node was inserted right after p.");//!
    }


}
```

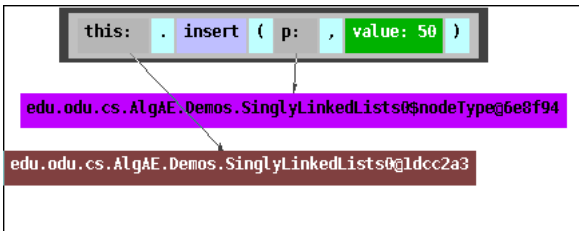

Figure 4.6: Singly-Linked Lists: Default Rendering

Running this animation (`SLLInsertionDriver0`) yields disappointing results (Figure 4.6). That's because we have supplied no rendering options for the two data types (`SinglyLinkedLists` and `SinglyLinkedLists.nodeType`) being displayed. Consequently, the default rendering rules are being applied.

4.2.1 Rendering

ALGAE *renders* objects according to five criteria:

1. What color should be used for the enclosing box?
2. What are the components of the object?
3. How many components should be laid out per row?
4. To what other objects does this one connect?
5. What string represents the value of this object?

For each object, ALGAE consults a chain of *renderers*, each of which either provides an answer to these questions or responds with a "don't know" value, in which case the question is referred further on down the chain.

Each renderer must implement this interface:

```
package edu.odu.cs.zeil.AlgAE.Snapshot.Rendering;

import java.awt.Color;

/**
 * Determines how a given object or class of objects should be drawn.
 *
 * @author zeil
```

```
 *
 */
public interface Renderer<T> {

        /**
         * What string will be used as the value of this object?
         *
         * @param obj: object to be drawn
         * @return a string or null to yield to other renderers
         */
        public String getValue(T obj);

        /**
         * What color will be used to draw this object?
         *
         * @param obj: object to be drawn
         * @return a color or null to yield to other renderers
         */
        public Color getColor(T obj);

        /**
         * Get a list of other objects to be drawn inside the
         * box portraying this one.
         *
         * @param obj: object to be drawn
         *
         * @return an array of contained objects or null to yield to other renderers
         */
        public List<Component> getComponents(T obj);

        /**
         * Get a list of other objects to which we will draw
         * pointers from this one.
         *
         * @param obj: object to be drawn
         *
         * @return an array of referenced objects or null to yield to other renderers
         */
        public List<Connection> getConnections(T obj);


        /**
         * Indicates how components will be laid out within the box
         * representing this object. A return value of 1 will force all
         * components to be laid out in a single vertical column. Larger
```

```
     * return values will permit a more horizontal layout.
     *
     * A zero value requests that components be laid out in a (more or less) mi
     *
     * @param obj
     * @return max #components per row or a negative value to yield to other re
     */

    public int getMaxComponentsPerRow(T obj);

}
```

Each of the functions in this interface directly pertains to one of the 5 rendering criteria listed above.

There are several ways to associate a renderer with a particular object:

- An *object renderer* is associated with a specific object reference. Object renderers have a limited lifetime. They are created via an `ActivationRecord` and last only as long as that function activation is in effect.

  Color highlighting is actually a shorthand for creating a renderer that specifies a color but answers "don't know" to every other rendering question.

  Object renderers have the highest priority of any kind of renderer.

- If a rendering question cannot be resolved by an object renderer, then a search is made for a *type renderer* that is bound to the data type of the object or to to one of its base classes.

  Type renderers can be bound in one of two ways. The most general way is for an animation driver to call `algae().render(`*`classObject, renderer`*`);`. A convenient shorthand, however, is for a type to implement the *CanBeRendered* interface:

  ```
  package edu.odu.cs.zeil.AlgAE.Snapshot.Rendering;

  public interface CanBeRendered<T> {
     public Renderer<T> getRenderer();
  }
  ```

  in which case each object of that type supplies a reference to its own type renderer. Some classes may implement both `CanBeRendered` and `Renderer`, in which case `getRenderer()` will simply return `this`.

- If no object renderer nor type renderer responds to a particular rendering question, then the final fallback is the *default renderer*, which behaves as follows:

  1. Color: A color is selected based upon a hash of the type name. In essence, the color is chosen randomly, but all objects of the same type will have the same color.

2. Components: If the object is an array or implements `Iterable`, its elements are shown as internal components. Otherwise, the object has no components.
3. Layout: Most objects are arranged vertically (1 component per row). However, arrays alternate horizontal and vertical depending on dimensionality. A one-dimensional array is laid out horizontally (100 elements per row), a two dimensional array is laid out as a vertical list of one-dimensional rows, and so on.
4. Connections: none
5. Value: Determined by calling `toString()`

   Not that this provides a useful default for primitives, Strings, and many other classes. Figure 4.6, however, shows that a class that provides no renderer of its own *and* that fails to implement `toString()` will yield rather ugly results.

4.2.2 Rendering Linked List Nodes

Let's start, then, by creating a renderer for the inner `nodeType` class from Listing 4.2.7.

To make this easy, we will use the `CanBeRendered` approach (although this increases the number of lines of code we will need to hide with `//!`).

We will start by having the node type declare that it will provide its own renderer:

```
class nodeType
implements CanBeRendered<nodeType>, Renderer<nodeType>//!
{
    int info;
    nodeType link;

    public nodeType()
    {
        info = -999;//!
        link = null;
    }

    public Renderer<nodeType> getRenderer() {//!
        return this;//!
    }//!
      ⋮
```

Now, how shall we actually render this?

1. We'll choose any color that we like:

```
      ⋮
```

```
public Color getColor(nodeType obj) {//!
    return Color.green.darker();//!
}//!
  ⋮
```

2. Each node will have a single internal component, its `info` field.

```
  ⋮
public List<Component> getComponents(nodeType obj) {//!
    LinkedList<Component> data = new LinkedList<Component>();//!
    data.add (new Component(info, "info"));//!
    return data;//!
}//!
  ⋮
```

Actually, we could just as easily treat this as the "value" string of the object, but treating it as a separate component is more flexible in case we later want to adapt this code to lists of other kinds of data.

Components can be given optional labels (e.g., "info" in this case). in which case those labels will appear within the rendering. One might commonly use such labels for components representing a single data member while eschewing any such labels for components of a more array-like structure.

3. The `link` field will supply the only connection leaving this object.

```
  ⋮
public List<Connection> getConnections(nodeType obj) {//!
    LinkedList<Connection> links = new LinkedList<Connection>();//!
    Connection c =  new Connection(link, 85.0, 95.0);//!
    c.setLabel("link");//!
    links.add(c);//!
    return links;//!
}//!
  ⋮
```

When creating connections, we give a range of angles from which the connection may emerge. In this case, I use the range from 85 to 95 degrees. Zero degrees represents straight up vertically, and 90 degrees points directly to the right.

These angles have a tremendous effect upon the rendering of connected entities. Consider, for example, that a typical doubly-linked list node and a typically binary tree node are topo-

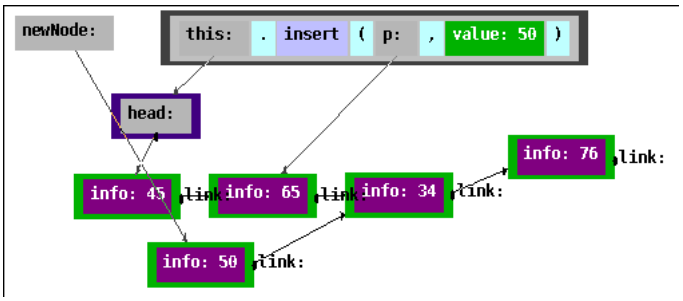

Figure 4.7: Singly-Linked Lists: Rendering

logically identical - each has a data field and a pair of emerging pointers. But if those pointers emerge at opposite ends of the node (90 and 270 degrees), then the nodes will tend to arrange themselves into a straight line. If those pointers emerge from the bottom corners (135 and 225 degrees), then even a fairly degenerate tree will try to arrange itself into the traditional "inverted-Vs" drawing.

4. Since we only have one internal component, the number of components per row is largely irrelevant:

```
  ⋮
public int getMaxComponentsPerRow(nodeType obj) {//!
    return 100;//!
}//!
  ⋮
```

5. And we won't need an explicit value string, because we have opted to show the `info` as a nested component.

```
  ⋮
public String getValue(nodeType obj) {//!
    return "";//!
}//!
```

With these rendering functions, and a similar but even simpler set for the `SinglyLinkedLists` class itself, we arrive at an animation (SLLInsertionDriver) that produces images such as in Figure 4.7.

4.3 Continuous Animations

The linked list insertion demo is rather limited, as it runs only a single algorithm. The reason for this is that it was actually originally intended as an applet that would be embedded (inline) into a course web page devoted to that specific algorithm.

For that purpose, it might be useful if the algorithm ran itself immediately and continuously as soon as the web page was visited.

This can be accomplished by making calls to the animated function from the starting action, the `MenuFunction` called before any choices are offered to the viewer. In the driver, we add a new registration:

```
    ⋮
@Override
public void buildMenu() {

    registerStartingAction(new MenuFunction() {
        @Override
        public void selected() {
            getAnimator().setSpeed(30);
            for (int k = 0; k < 1000; ++k) {
                generateLL();
                sll.insert(sll.head.link, 50);
                try {
                    Thread.sleep(2500);
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    });


    register ("Insert a node", new MenuFunction() {

        @Override
        public void selected() {
            generateLL();
            sll.insert(sll.head.link, 50);
        }

    });
⋮
```

- Normal menu selections start with the viewer in pausing mode - it will stop at each breakpoint. The starting action, however, runs by default in continuous mode. The optional `setSpeed` call is used to slow down the default speed with which it moves from breakpoint to breakpoint.

- Similarly, the `sleep` call at the end helps inject a slight pause between each restart of the algorithm.

- Rather than looping forever, I prefer to use a loop like the for loop shown here to repeat the demonstration a large but finite number of times. The regular Algorithm menu function can be kept available if someone were to stay on the page long enough for this loop to complete.

4.4 User Interaction

Because ALGAE is actually executing the code being animated, rather than simply displaying a pre-determined sequence of pictures, an ALGAE animation can accept user inputs and demonstrate how the algorithm would react to those.

For example, we might alter the driver for the (non-continuous) linked list insertion function as follows:

```
    ⋮
  @Override
  public void buildMenu() {
      register("Insert a node", new MenuFunction() {

          @Override
          public void selected() {
              generateLL();
              String val = promptForInput("What number to insert?", "[0-9]+");
              int value = Integer.parseInt(val);
              sll.insert(sll.head.link, value);
          }

      });
    ⋮
```

- The `promptForInput` function takes two parameters. The first is a prompt string that is displayed in a pop-up activation box. The second is a regular expression describing a valid input string. If the user responds with a string that does not match this pattern, he or she will be prompted again for a new input.

- Another mechanism for interaction is provided by the I/O streams `Animation.in` and `Animation.out`, which can be thought of as analogues of the more conventional `System.in` and `System.out`. These streams, however, read and write from the I/O pane located in the lower right of the animation (Figure 1.1).

Chapter 5

Animating a C++ Algorithm

Version 3.0 of ALGAE no longer supports direct animation of C++ code. I hope to restore this ability in the near future.

However, by using the selective rewriting facility of the source code viewer, one can easily portray a set of animated Java algorithms as C++, as pseudo-code, or as almost any other language. C++ is a particularly easy case because so much code in that language is also valid Java code.

For example, for one of my classes I wished to animate the following sorting algorithms taken from the course textbook [5]:

```
void bubbleSort(int list[], int length)
{
    int temp;
    int iteration;
    int index;

    for (iteration = 1; iteration < length; iteration++)
    {
        for (index = 0; index < length - iteration; index++)
            if (list[index] > list[index + 1])
            {
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
            }
    }
}

void selectionSort(int list[], int length)
{
    int index;
```

```
    int smallestIndex;
    int location;
    int temp;

    for (index = 0; index < length - 1; index++)
    {
            //Step a
        smallestIndex = index;

        for (location = index + 1; location < length; location++)
            if (list[location] < list[smallestIndex])
                smallestIndex = location;

            //Step b
        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}


void insertionSort (int list[], int listLength)
{
    int firstOutOfOrder, location;
    int temp;

    for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                              firstOutOfOrder++)
        if (list[firstOutOfOrder] < list[firstOutOfOrder-1])
        {
            temp = list[firstOutOfOrder];
            location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
}
```

To do this, I simply created a basic Java package structure:

```
package edu.odu.cs.cs333.animations;//!

import edu.odu.cs.zeil.AlgAE.ActivationRecord;//!
import edu.odu.cs.zeil.AlgAE.Animation;//!

public class Sorting {//!


//
//  Based on Malik, "C++ Programming [From Problem Analysis to Program Design]"
//      chapter 10
//

   ⋮
}//!
```

marking the new Java statements with a trailing `//!` so that they would not be displayed by the source code viewer.

I then pasted the `C++` code into the middle of the package and added "`public static`" to each function.

```
package edu.odu.cs.cs333.animations;//!

import edu.odu.cs.zeil.AlgAE.ActivationRecord;//!
import edu.odu.cs.zeil.AlgAE.Animation;//!

public class Sorting {//!


//
//  Based on Malik, "C++ Programming [From Problem Analysis to Program Design]"
//      chapter 10
//

public static//!
void bubbleSort(int list[], int length)
{
    int temp;
    int iteration;
    int index;

    for (iteration = 1; iteration < length; iteration++)
    {
        for (index = 0; index < length - iteration; index++)
```

```
            if (list[index] > list[index + 1])
            {
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
            }
    }
}

public static//!
void selectionSort(int list[], int length)
{
    int index;
    int smallestIndex;
    int location;
    int temp;

    for (index = 0; index < length - 1; index++)
    {
            //Step a
        smallestIndex = index;

        for (location = index + 1; location < length; location++)
            if (list[location] < list[smallestIndex])
                smallestIndex = location;

            //Step b
        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}


public static//!
void insertionSort (int list[], int listLength)
{
    int firstOutOfOrder, location;
    int temp;

    for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                              firstOutOfOrder++)
        if (list[firstOutOfOrder] < list[firstOutOfOrder-1])
        {
            temp = list[firstOutOfOrder];
```

```
            location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
}

}//!
```

The resulting Java code is, in fact, the starting point for the example given earlier in section 4.1.

In many cases, the process is a little more involved than that, but the principle remains the same. For example, a block of C++ code to walk a linked list

```
void traverse (Node* head)
{
  Node* current = head;
  while (current != 0)
    {
      cout << current->data << endl;
      current = current->next;
    }
}
```

will need to be rewritten almost entirely. Even this is not as bad as it seems, however, because the equivalent Java code still exhibits a one-to-one correlation among the statements:

```
void traverse (Node head)//!void traverse (Node* head)
{
  Node current = head;//!  Node* current = head;
  while (current != null)//!  while (current != 0)
    {
      System.out.println(current.data);//!       cout << current->data << endl;
      current = current.next;//!       current = current->next;
    }
}
```

I generally start this process by using an editor with a keyboard macro capability to put a `//!` at the beginning of each line, then move down line by line, copying everything to the right of the `//!`, pasting it at the start of the line, and making whatever small changes are actually necessary.

Bibliography

[1] Apache Software Foundation. *The Apache Ant Project*. 2011. URL: `http://ant.apache.org/`.

[2] Eclipse Foundation. *Eclipse*. 2011. URL: `http://www.eclipse.org/`.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 2008.

[4] D. Klein. The open channel. *IEEE Computer*, page 112, November 1981.

[5] D. S. Malik. *C++ Programming: From problem analysis to program design*. Course Technology Ptr, 2010.

[6] Oracle Corp. *JavaHelp System*. 2011. URL: `http://javahelp.java.net/`.