# A Tool for Data Structure Visualization and User-defined Algorithm Animation

**TAO CHEN AND TAREK SOBH**

*Department of Computer Science and engineering*
*University of Bridgeport*
*Bridgeport, CT 06601, USA*

**Abstract**

In this paper, a software application that features the visualization of commonly used data structures and their associated insertion and deletion operations is introduced. In addition, this software can be used to animate user-defined algorithms.

## 1. Introduction

Data Structures and Algorithms is a fundamental course in Computer Science. However, many students find it difficult because it requires abstract thinking. It would be very helpful if there was a visualization tool of data structures such as arrays, queues, stacks, trees and graphs for students to experiment with. The tool would allow students to see how an element is inserted into or deleted from different data structures, how a tree is traversed in different order (pre-order, in-order, post-order, level-order), etc. Moreover, this tool would provide a simple language, by which students can write their own algorithms so that the execution of the algorithm is animated. This project is intended to create such an exploration environment, in which students can learn through experimentation. This tool can be used as an effective supplement to the traditional classroom education and textbooks for Data Structures and Algorithms courses. The software package presented in this paper has the following functionality.

a. Provides complete visualization for the widely used data structures such as array, stack, queue, tree, heap, graph, etc.
b. Provides the animation of common operations associated with the data structures, such as inserting an element into and deleting an element from array, stack, and queue.
c. Provides animation of simple user-defined algorithms.

## 2. Background

The development of technologies and the evolvement of the World Wide Web have influenced education. Instructional Web sites and courses on the Web have grown dramatically. Web-based courses that consist of the syllabus, assignments and lecture notes are now widely used. Instructional Web sites that are dedicated to Data Structures and algorithms can be easily found by using Search Engines. To name a few:

http://swww.ee.uwa.edu.au/~plsd210/ds/ds_ToC.html [1]
http://www.cee.hw.ac.uk/~alison/ds98/ds98.html [2]
http://www.cs.twsu.edu/~bjowens/cs300/ [3]
http://www.cs.berkeley.edu/~edith/cs270/ [4]

However, The majority of the instructional web sites explored during this project lack interactive multimedia.

One of the best sites found that does contain interactivity is a course site developed for teaching Data Structures and Algorithms in Java by the Computer Science Department of Brown University [5]. This site has a collection of applets that demonstrate some commonly used data structures such as queues, stacks, and some famous algorithms such as merge sort, quick sort, etc. However, these applets are not complete and lack a common Graphical User Interface. Another good site in interactive Data Structure visualizations is developed by Duane J. Jarc in George Washington University [6]. This site provides animations in binary Trees, graphs, and sorting algorithms. But there is no animation available for algorithms that are defined by users.

Algorithm animation is a type of program visualization that is mainly concerned with displaying the executions of computer algorithms. Lots of work has already been done in this field. For example, the XTANGO [7] and POLKA [8] systems developed by the Graphic, Visualization and Usability Center (GUV) at Georgia Tech are general-purpose animation systems, which require the user to write an algorithm in the C language and register the events that the user wants to observe during the execution of the algorithm. However, these systems are implemented on top of Unix and X11 Window system, and are not portable to other platforms. In addition, we feel they are overkill for a basic Data structures and Algorithms course.

Another algorithm animation system found is Zeus[9], which is developed by Digital Equipment Corporation's Systems Research Center. This system is a little complicated, require from the user lots of effort to prepare animations. It is targeted at more advanced application programmers.

Since our software is intended to the aid first year Computer Science students learning Data Structures and Algorithms, ease of use becomes our main consideration. Our approach for the user-defined algorithm animation is that the user codes the algorithm in a simple language called JavaMy, which is very similar to Java. The only effort the user needs to make is to instantiate the data structures he/she wants to observe using the observable data types provided by the software. After parsing the JavaMy algorithm file, an animation frame is created and the observable data structures are added to the frame so that the user can watch the changes made to the data structures when the algorithm is executing.

## 3. Software Package

Before discussing the design of the software package, an overview of the functionality of the package is given here. The screenshots on the following pages should give an idea of how the software runs.

### 3.1 Data Structure Visualization

The observable data structures currently available in this software packages include: array, stack, queue, binary search tree, heap and graph. They will be introduced in subsections 3.1.1-3.1.6.

#### 3.1.1 Array

An Array stores a collection of identically typed objects, which are randomly accessible by a numeric index. The structure and insert, delete operations are shown in Figures1-5.

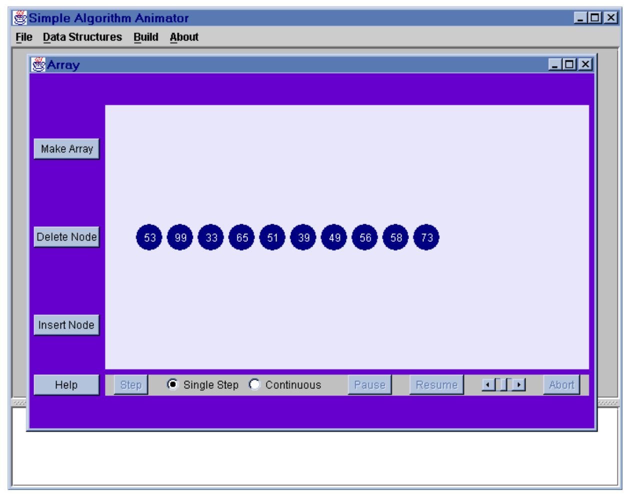


**Figure 1**: Array

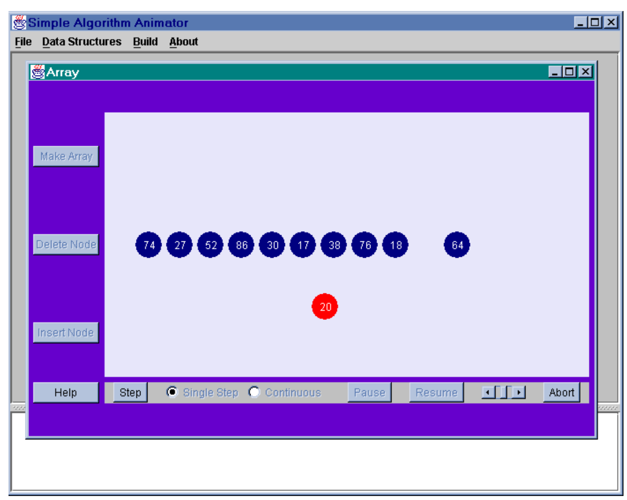


**Figure 2:** Insert a node into array

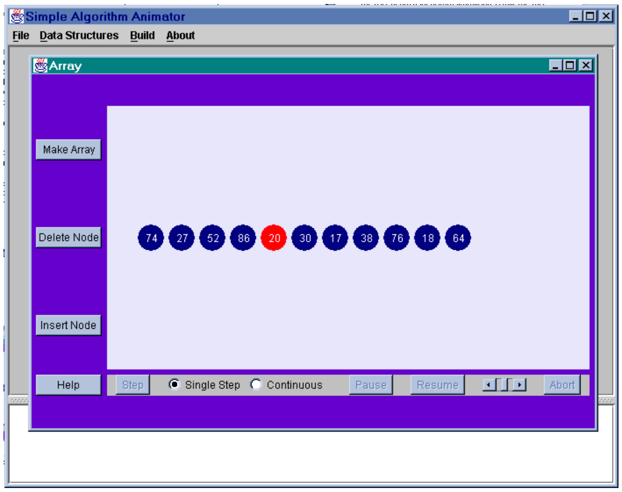


**Figure 3**: Insertion done

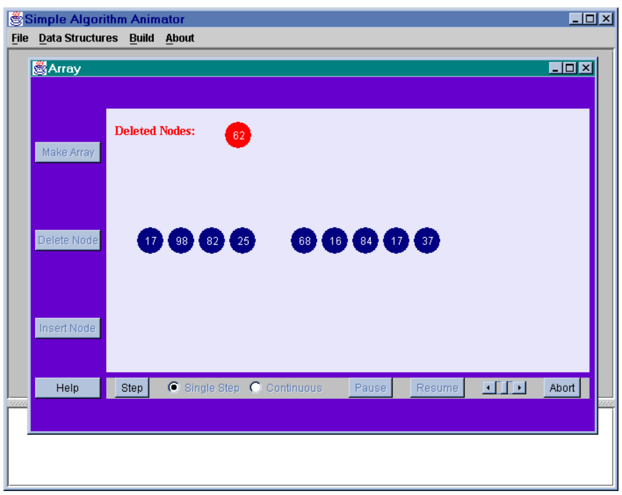


**Figure 4**: Delete a node from array

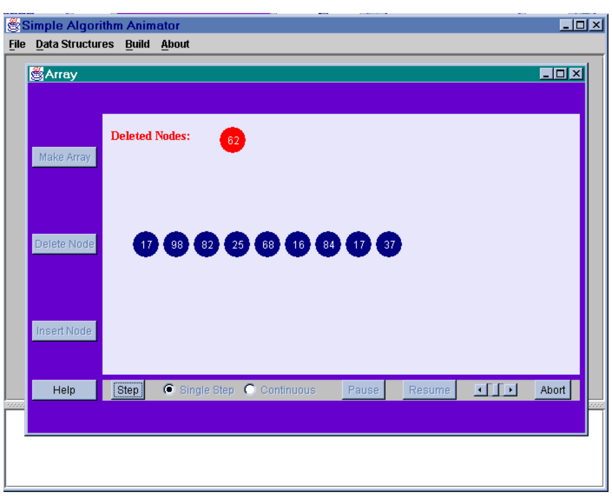


**Figure 5**: Deletion done

### 3.1.2 Stack

A stack is a data structure in which all access is restricted to the most recently inserted element. The basic operations of a stack are push, pop and top. The structure and the push, pop operations are shown in Figures 6-8.

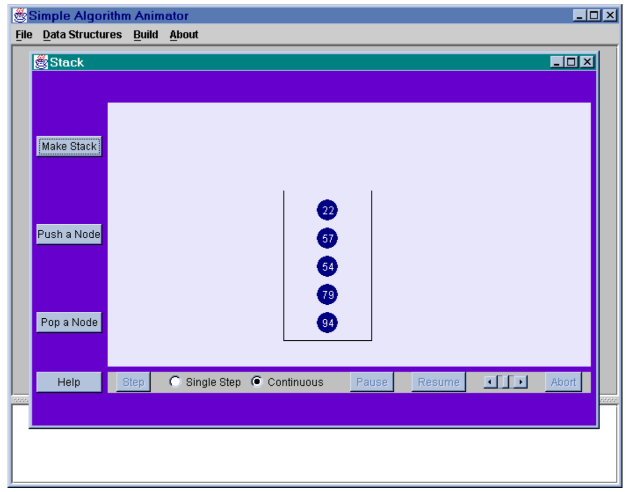


**Figure 6:** Stack

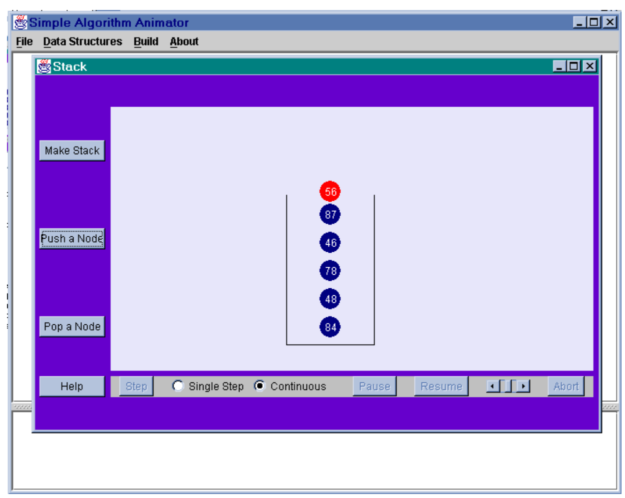


**Figure 7**: Push a node into the stack

### 3.1.3 Queue

A queue is a data structure that restricts the access to the least recently inserted item. The basic operations supported by queues are enqueue and dequeue, representing insertion to the rear (back) and removal of the item at the front. Figures 9-11 demonstrate the array-based queue structure and associated operations.

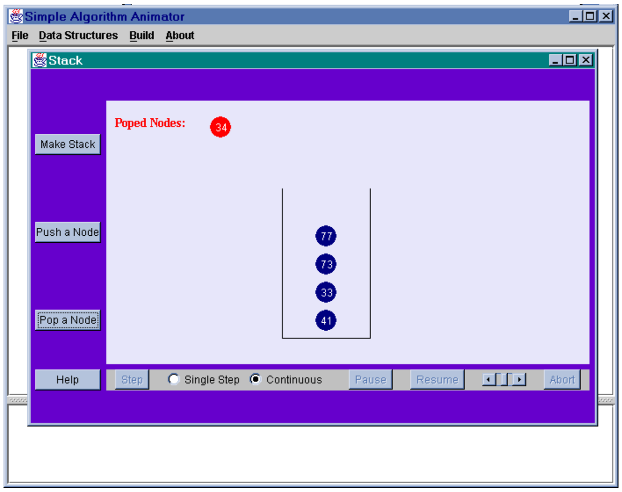


**Figure 8**: Pop a node from the stack

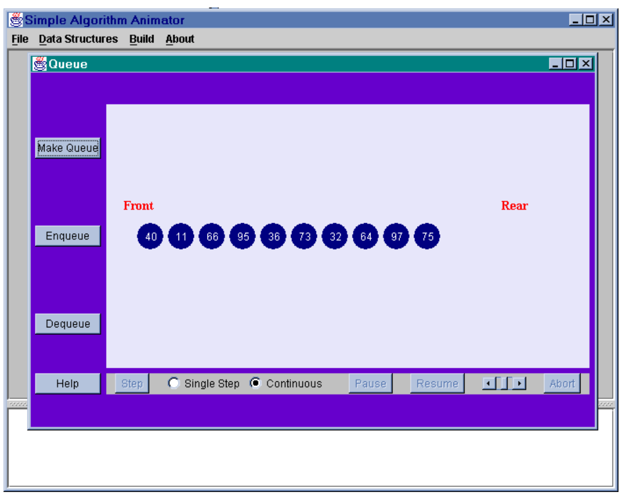


**Figure 9:** Array-based Queue

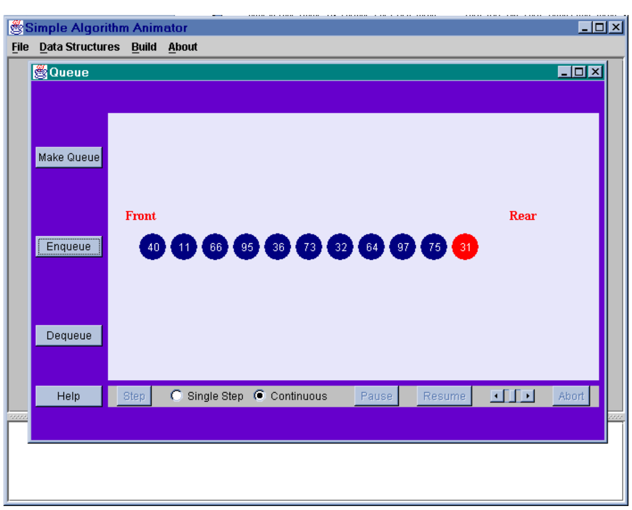


**Figure 10**: Enqueue an element to the queue

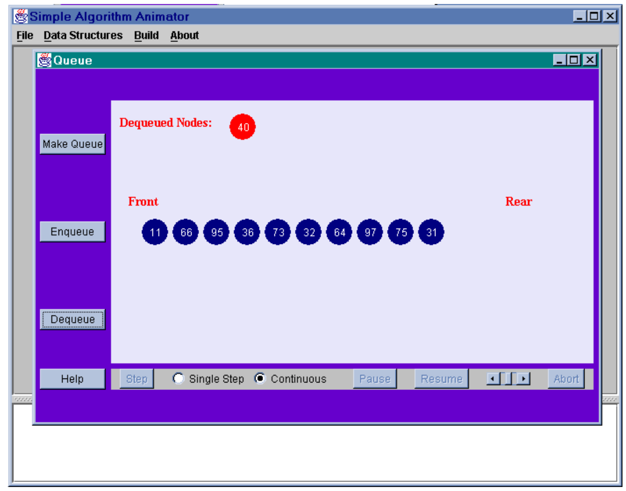


**Figure 11**: Dequeue an element from the queue

### 3.1.4 Binary Search Tree

A binary search tree is a kind of binary tree where every node's left subtree has values less than the node's value, and every right subtree has greater values. The basic operations are delete, insert and find as shown in Figures 12-16.

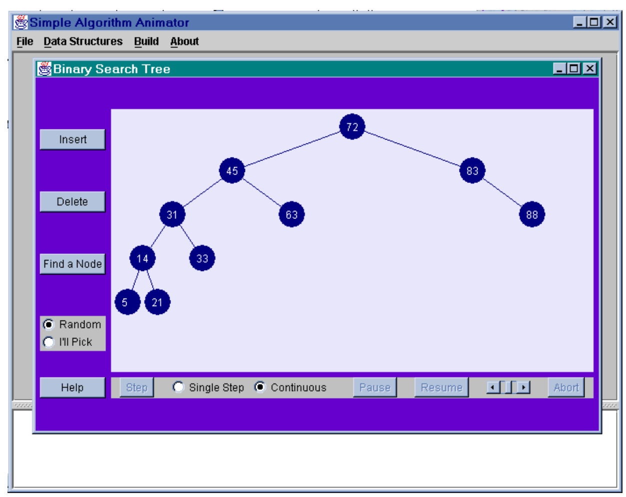


**Figure 12**: Binary Search Tree

### 3.1.5 Binary Heap

A Binary heap is a complete tree where every node has a key more extreme (greater or less) than or equal to the key of its parent. In this paper, a Max Heap is implemented. The allowed operations are deleteMax and insert as shown in Figures 17-19.

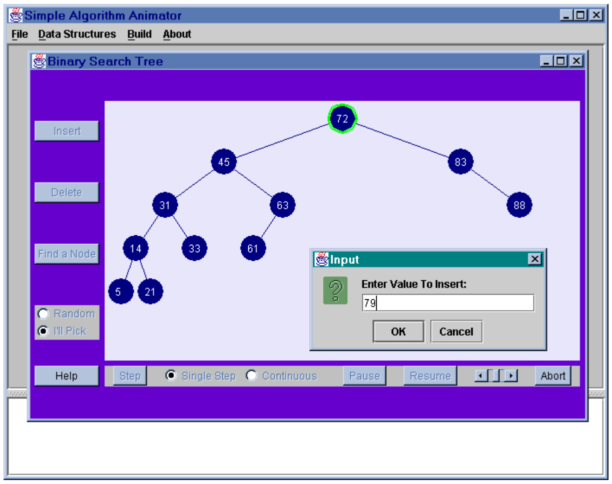


**Figure 13**: Insert an element into the tree

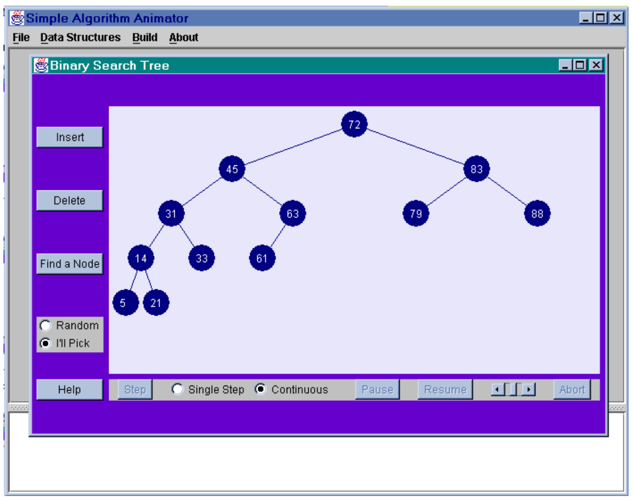


**Figure 14**: Insertion Done

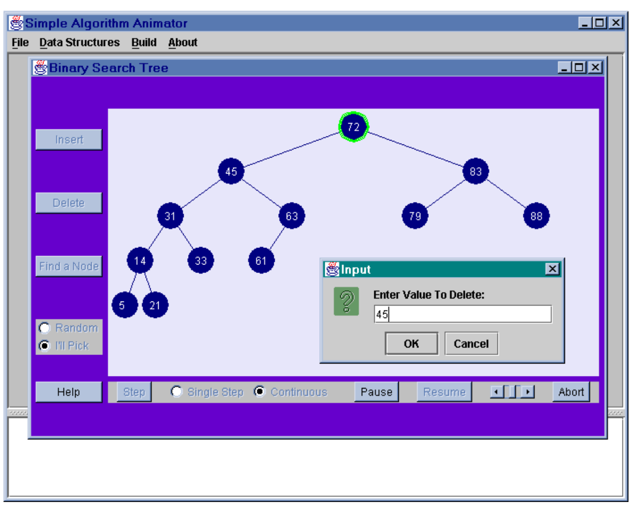


**Figure 15**: Delete an element from the tree

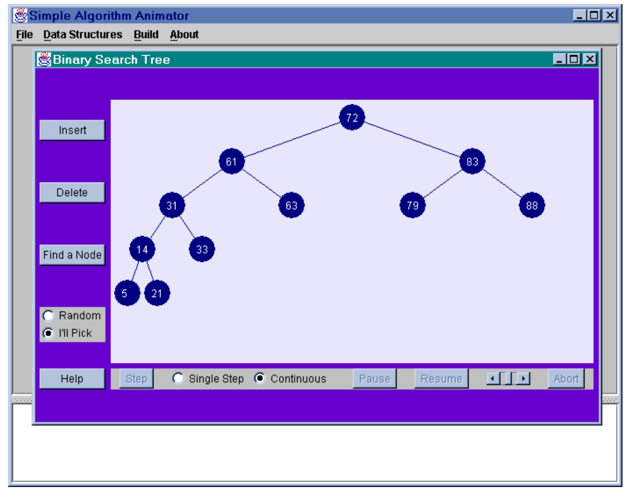


**Figure 16**: Deletion Done

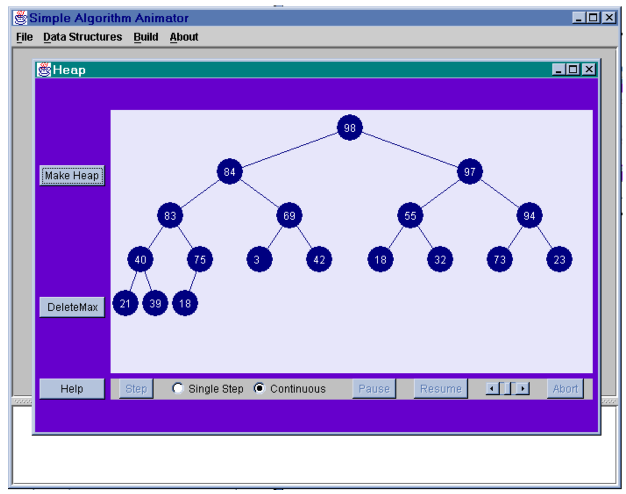


**Figure 17**: Heap

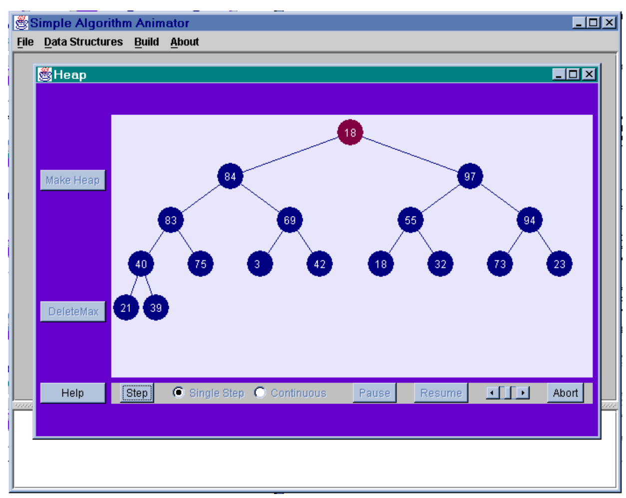


**Figure 18**: Delete the Maximum node

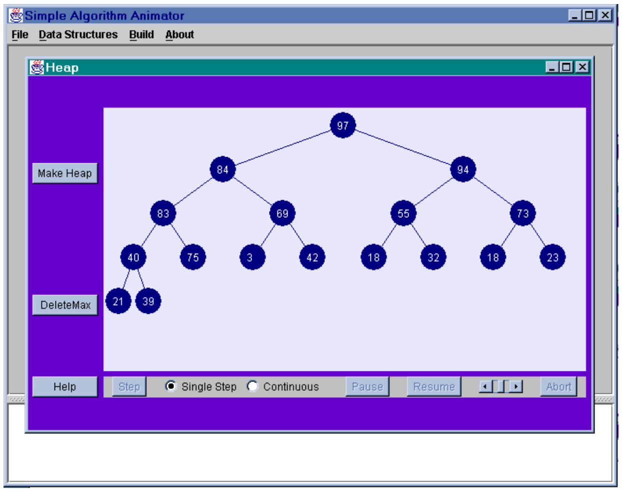


**Figure19**: Deletion done

### 3.1.6 Graph

A graph consists of a set of vertices and a set of edges that connect the vertices as shown in Figure 20.

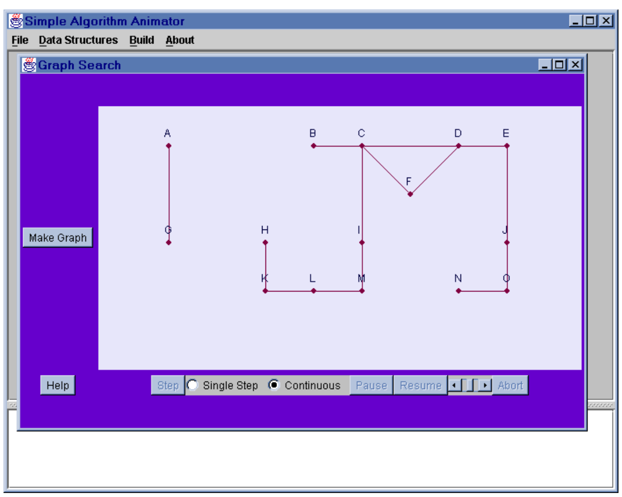


**Figure 20:** Undirected Graph

## 3.2 User-defined Algorithm Animation

In this section we describe the steps of animating an algorithm and the details of the JavaMy language which is provided by the software and is used for visualizing the execution of the algorithm.

### 3.2.1 Steps

To illustrate how the animation of a user-defined algorithm is done we will use a simple sorting algorithm -- bubble sort -- as an example to walk through the steps.

Bubble sort works by repeatedly moving the largest element to the highest index position of the array. It focuses on successive adjacent pairs of elements in the array, either swapping them if they are out of order or leaving them alone. The algorithm can be summarized as following.

1. Step through the array of data.
2. While stepping through, if two adjacent values are not in sorted order, then swap them.
3. When a complete pass of the data has been conducted, if any swaps have been made, then the data may still not be sorted. Goto 1.
4. Otherwise, if no swaps were made on the last pass, then the data is sorted, and the algorithm is finished.

Before we use the software to visualize the execution of bubble sort, we need to translate the above algorithm into JavaMy code. The details of the JavaMy language can be found in next sub-section. The bubble sort algorithm translated into JavaMy reads:

```
/* Bubble Sort Algorithm */
public static void main(String args[])
{
  final int SIZE = 8;

  MyArray intArray = new MyArray(
       AnimatorFrame.ARRAY_POSITION,SIZE);

  for (int i=0; i<SIZE; i++)
  {
       intArray.setValue(
       ScreenPanel.getRandom(10, 100), i);
  }

  for (int i=SIZE; i>1; i--)
  {
     for (int j=0; j<i-1; j++)
     {
```

```
                if (intArray.getInt(j) >
                        intArray.getInt(j+1)) {
                        intArray.swap(j, j+1);
                }
        }
    }
}
```

The first step is to write the algorithm. The user can use any text editor to enter and edit the algorithm. To make it easy to use, the software provides a simple built-in code editor. By clicking File->New menu item, the user can enter the code into the text field as shown in Figure 21.

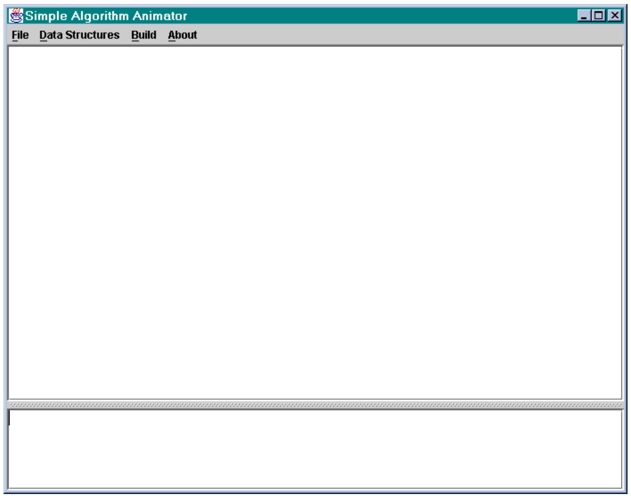


**Figure 21**: Algorithm Animator

The second step is to save the entered algorithm into a file by clicking File->Save as menu item as shown in Figure 22.

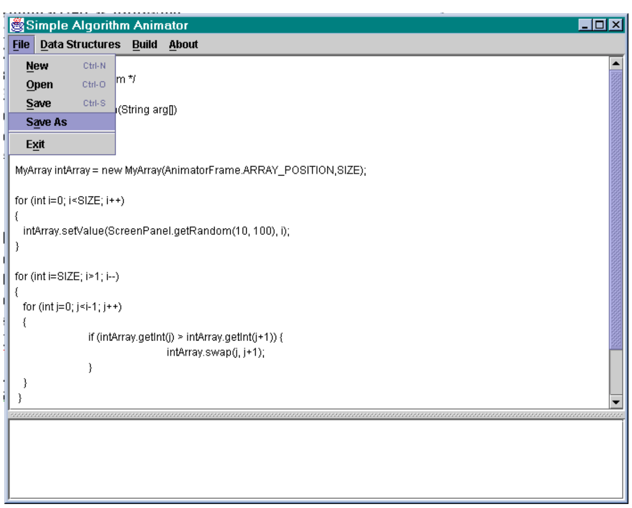


**Figure 22**: Save Algorithm File

After saving the file, the user can parse it by clicking the Build->Compile menu item as in Figure 23.

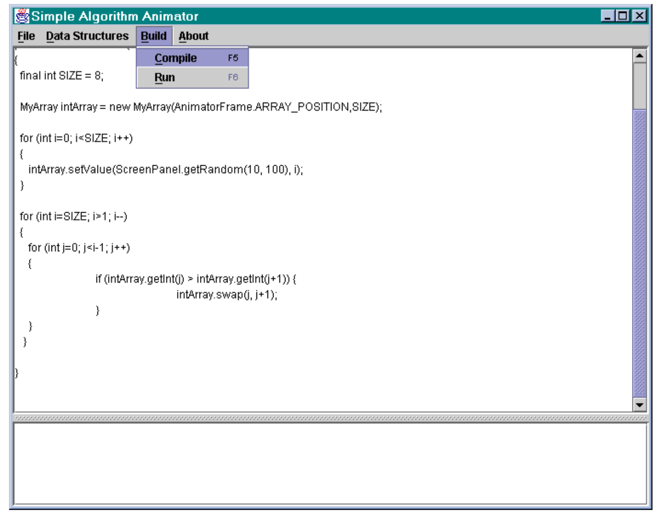


**Figure 23**: Parse and Compile Algorithm File

If the file is edited using another editor, the user can load the algorithm file by first clicking the File -> Open menu item, then finding the file name in the File Open dialogue as shown in Figure 24.

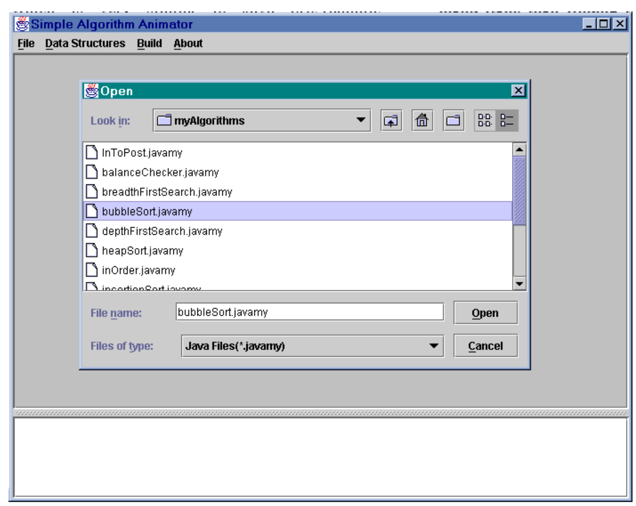


**Figure 24**: Open an Algorithm File

If no error occurs during the parsing process, the resulting Java file will be compiled. If errors occur during the parsing or compilation process, the errors will be displayed on the text area on the bottom of the window. The user can then go back to the algorithm file and make necessary corrections. If the file is parsed and compiled successfully, a corresponding message will be displayed as shown in Figure 25.

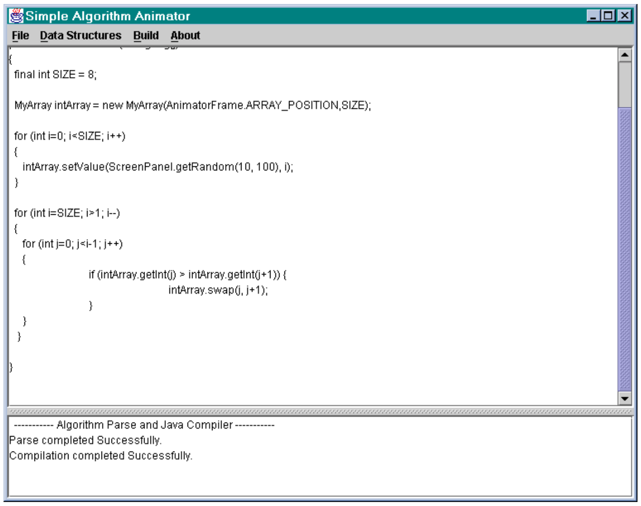


**Figure 25**: Parsed and Compiled Successfully

The user can then click Build->Run menu item to watch the animation. The animation frame is shown in Figure 26.

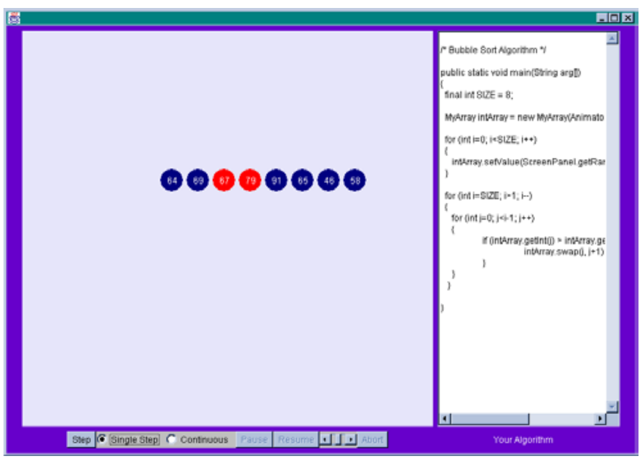


**Figure 26**: Animation Frame of Bubble Sort

The animation frame consists of the animation canvas, the user algorithm text field, and the animation control panel. The animation canvas is where the data structures used in the algorithm are displayed. The user-defined algorithm coded in JavaMy language is displayed in the right hand text field. The control panel can be used to control the animation. Users can choose to run the algorithm animation either continuously or step by step by clicking the radio button labeled "Continuous" or "Single Step". The animation speed can be changed by clicking the slider bar.

### 3.2.2 JavaMy and Algorithm Coding

As mentioned earlier, JavaMy is the language used to code the user-defined algorithm. The syntax of JavaMy is similar to Java. The difference is in the program constructs. Every program in Java consists of at least one class definition. When the class definition is saved in a file, the file name must be the class name followed by the ".java" file name extension. However, in JavaMy the user does not need to define a class, the coded algorithm is put into the main() method. That is, the user algorithm file always starts with

```
public static void main(String args[])
```

Moreover, the algorithm file can be named in anyway the user wants. However, the file name extension ".javamy" is recommended to separate the algorithm file coded in JavaMy from other files.

When coding the algorithm, the user is allowed to make a decision regarding which of the data structures used in the algorithm he/she wants to observe, and use the set of the observable data types provided by the software to define these data structures. All the observable data types are named by adding the prefix "My" to the corresponding normal data types. For instance, the observable array is named MyArray, and the queue is named MyQueue, etc. The data structure objects that do not interest the user can be instantiated by the data types provided by Java API. The software also provides some helper Java classes such as DrawableString, which can be used to add labels, explanations and other useful information to the Animation Frame. All the available observable data type classes, helper classes and their usage can be found in the Javadoc documentation that comes with the software.

### 3.2.3 Examples

In this subsection a few more examples will be presented on how the algorithm is coded in JavaMy language and what the final animation looks like. This will help the user to have a good understanding of algorithm coding.

#### 3.2.3.1 Balanced Symbol Checking

A balanced symbol checker is a tool to help debug compiler errors, which checks whether symbols are balanced. In other words, whether every "{" corresponds to a "}", every "[" to a "]", every "(" to a “)”, and so on. The basic algorithm is stated as follows:

Make an empty stack. Read tokens until the end of the input file. If the token is an opening symbol, push it onto the stack. If it is a closing symbol and if the stack is empty, report an error. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error. At the end of the file, if the stack is not empty, then report an error.

The above algorithm coded in JavaMy is shown in the following program:.

```
/* Balanced Symbol checker is used to check
whether every { corresponds to a }, every [ to
a ], every ( to a ). And the squence [()] is
legal, but [(]) is wrong.
 */

public static void main(String arg[])
{
  String input = "{[([([()]}";
  char c, match;
  String errmsg;

  MyArray in = new
       MyArray(AnimatorFrame.ARRAY_POSITION,
           input.length());
  MyStack pendingTokens = new
       MyStack(AnimatorFrame.STACK_POSITION,
                 0);

  for (int i=0; i<input.length(); i++)
  {
     in.setValue(input.charAt(i),i);
  }

  for (int i=0; i<input.length(); i++)
  {
       c = in.getChar(i);
       switch(c)
       {
          case '(':
          case '{':
          case '[':
              pendingTokens.push(c);
              break;

          case ')':
          case '}':
          case ']':
              if (pendingTokens.isEmpty())
              {
               System.out.println("Extraneous "
               + c +" found");
              }
              else
              {
               match = pendingTokens.topChar();
              pendingTokens.pop();
              if (match == '(' && c != ')' ||
                  match == '{' && c != '}' ||
                  match == '[' && c != ']' )
              {
                errmsg = "Found \"" + c + "\"
                 does not match \""+match+"\"";
                JOptionPane.showMessageDialog(
                new JPanel() , errmsg,"Error",
                 JOptionPane.ERROR_MESSAGE);
                }
              }
              break;

              default:
                 break;
       }
     }

     while (!pendingTokens.isEmpty())
     {
       match = pendingTokens.topChar();
       pendingTokens.pop();
       errmsg = "Unmatched \"" +
                   match +"\"";
       JOptionPane.showMessageDialog( new
             JPanel(), errmsg, "Error",
            JOptionPane.ERROR_MESSAGE);
     }
  }
```

The program starts with multiple-line comments, which document programs and improve program readability. The comment notation in JavaMy is the same as Java. Multiple-line comments are delimited with /* and */, and single-line comments are delimited with //. Following comments is simply a blank line. Blank lines, space characters and tab characters are known as white-space. Such characters are used to make the program easier to read. They are ignored by the parser. The bold line indicates the beginning of the real code of the algorithm. The three lines following the opening parentheses declare normal variables as in Java, using the data type provided by the Java programming language. The next six bold lines instantiated two observable data structures that will show on the animation frame. The first one is an array, which is used to hold the input string, that is, the string to be checked. The second one is a stack, which is used to hold the opening symbols. Here, MyArray and MyStack are used. Both of the constructors of MyArray and MyStack take two parameters. One is the Position parameter, which is used to decide the location of the data structure on the animation frame. Another parameter is the size of the array or stack. The rest of the code is the same as Java. Class MyArray and MyStack provides most of the commonly used methods, for example, setters and getters for setting and getting the values of the elements in the array, respectively, push(), pop() and methods for peeking the top element on the stack, etc. Details of those methods are described in the documentation generated by Javadoc.

After parsing and compiling the algorithm successfully, we can run the animation as described in subsection 3.2.1. The resulting animation frame is shown in Figure 27.

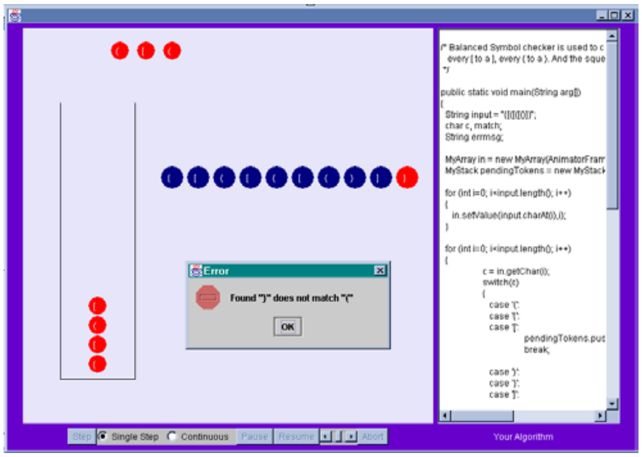


**Figure 27**: Animation Frame of Balanced Symbol Checking

3.2.3.2 Operator Precedence Parsing algorithm

The operator precedence parsing algorithm converts an infix expression to a postfix expression. It works as follows:

Make an empty stack. Go through the infix expression. If the token read is an operand, we immediately output it. If it is a close parenthesis, we pop the stack until an open parenthesis is seen. If it is an operator, pop all stack symbols until we see a symbol of lower precedence or a right associative symbol of equal precedence, then push the operator. When we reach the end of infix expression, pop all remaining stack symbols. Everything that is output and popped from the stack is the converted postfix expression.

```
// Filename: InToPost.javamy
// Convert an infix expression into postfix
// expression

public static void main (String args[])
{
    String infix = "1+2*7-9*5";
    DrawableString label1 = new
       DrawableString(new Position(200, 380),
                    "The infix expression:");
    DrawableString in = new DrawableString(new
                    Position(200, 400), infix);
DrawableString label2 = new
    DrawableString(new Position(200, 180),
        "The output postfix expression:");
MyQueue outQue = new  MyQueue(
   AnimatorFrame.ARRAY_POSITION,0);
MyStack opStack = new MyStack(
   AnimatorFrame.STACK_POSITION, 0);

int topOp;
int token;
int i=0;
String operand = new String();
StringBuffer value = new StringBuffer();

Precedence.initPrecTable();

while ( i< infix.length() )
{
    switch( infix.charAt(i) )
    {
      case '^':
         token = Precedence.EXP;
         i++;
         break;
      case '/':
         token = Precedence.DIV;
         i++;
         break;
      case '*':
         token = Precedence.MULT;
         i++;
         break;
      case '(':
         token = Precedence.OPAREN;
         i++;
         break;
      case ')':
         token = Precedence.CPAREN;
         i++;
         break;
      case '+':
         token = Precedence.PLUS;
         i++;
         break;
      case '-':
         token = Precedence.MINUS;
         i++;
         break;
      default: //operand
        value.delete(0, value.length());
        char c = infix.charAt(i);
       while ( c!='^' && c!='/' && c!='*'
         && c!='(' && c!=')' && c!='+'
         && c!='-' && c!=' ')
      {
         value.append(c);
         if (++i>=infix.length())
                 break;
         c = infix.charAt(i);
        }

        token = Precedence.VALUE;
       operand = new String(value);
  }

  switch( token )
  {
      case Precedence.VALUE:
```

```
              outQue.enqueue(operand);
              break;
          case Precedence.CPAREN:
            while((topOp = opStack.topInt())
            != Precedence.OPAREN && topOp !=
            Precedence.EOL )
    { //pop and output operators on the
    //stack until meet the open parenthesis
      outQue.enqueue(Precedence.token2char(
            topOp));
            opStack.pop();
    }
    if( topOp == Precedence.OPAREN )
    {//Get rid of opening parentheseis
       opStack.pop();           }
    else
    {    System.out.println(
              "Missing open parenthesis" );
         token = Precedence.ERR;
    }
    break;
    default:    // General operator case
       if (!opStack.isEmpty())
       {
            topOp = Precedence.char2token(
                   opStack.topChar());
            while(((Precedence)Precedence.
             precTable.elementAt(token)).
             inputSymbol <=((Precedence)
             Precedence.precTable.elementAt(
                   topOp)).topOfStack )
            {
              if (topOp==Precedence.OPAREN )
              {
                System.out.println(
                 "Unbalanced parentheses" );
                token = Precedence.ERR;
                break;
              }
              outQue.enqueue(
               Precedence.token2char(topOp));
              opStack.pop();
              if (opStack.isEmpty())
                   break;
              topOp = Precedence.char2token(
                          opStack.topChar());
            }
        }
        if (token != Precedence.EOL)
        {
            opStack.push(
              Precedence.token2char(token) );
        }
        else
                   break;
    }//end switch

 if (token == Precedence.ERR)
    break;
 } // end for

 while (!opStack.isEmpty())
 {// pop up all remaining stack symbols
    outQue.enqueue(opStack.topChar());
    opStack.pop();
 }
}
```

In this example, two observable data structures and a helper class are used, namely, MyQueue, MyStack and DrawableString. The queue is used to store the resulting postfix expression, and the stack is used to hold the operators. The DrawableStrings add some nice labels in the animation frame as shown in Figure 28 and 29, which make the animation clearer. The constructor of the MyQueue class also takes position and size as the parameters. The DrawableString takes two parameters of type Position and String, respectively. The String parameter passed is the string that will show on the animation frame.

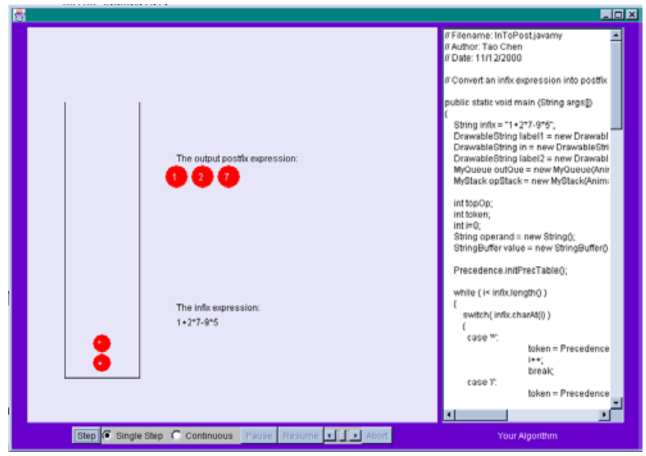


**Figure 28:** Infix to Postfix animation in progress

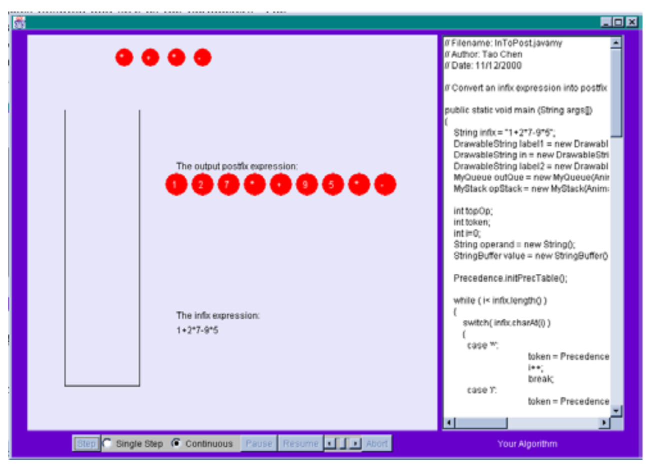


**Figure 29:** Infix to Postfix animation done

3.2.3.3 Heap Sort Algorithm

Heap sort is an algorithm to sort by building a heap, then repeatedly extracting the minimum item. An example heap sort written in JavaMy is shown as follows:

```
  public static void main(String args[])
  {
    final int MIN_NODES = 10;
    final int MAX_NODES = 32;
    final Position ROOT_POSITION = new
                     Position(273, 80);
    int numberOfNodes;
    Random random = new Random();
    MyHeap myheap = new MyHeap(ROOT_POSITION);
    DrawableString nodeLabel1 = new
              DrawableString(ROOT_POSITION);
    DrawableString nodeLabel2 = new
             DrawableString(ROOT_POSITION);
    DrawableString label1 = new
       DrawableString(new Position(20, 370),
       "Before Sort:");
    DrawableString b4Sort = new
       DrawableString(new Position(20, 390));
    DrawableString label2 = new DrawableString
       (new Position(20, 440), "After Sort:");
    DrawableString afterSort = new
       DrawableString(new Position(20, 460));

    int values[] = new int[MAX_NODES];
    numberOfNodes = Math.abs(random.nextInt()
              % (MAX_NODES / 2)) + MIN_NODES;
    String temp = "";

    // get the node values
    for (int i = 1; i <= numberOfNodes; i++)
       values[i] = Math.abs(
             random.nextInt()%100);

    for (int i=1; i<=numberOfNodes; i++)
       temp = temp + values[i]+"  ";

    b4Sort.setString(temp);
    myheap.makeHeap(values, numberOfNodes,
                nodeLabel1, nodeLabel2);

    // Performing Sort
    int i = numberOfNodes;
    while (i > 1)
    {
       myheap.root.swapNodes(myheap.heap[i],
              nodeLabel1, nodeLabel2);
       myheap.heap[i].changeColor();
       myheap.root.reheapDown(--i, nodeLabel1,
                     nodeLabel2);
    }
    myheap.root.changeColor();

    //output the sort result
    temp = "";
    for (int j=1; j<=numberOfNodes; j++)
       temp = temp+myheap.heap[j].getString()
              +"  ";
    afterSort.setString(temp);
}
```

This example demonstrates the usage of the observable data type MyHeap. The resulting animation frames are shown in Figures 30-32.

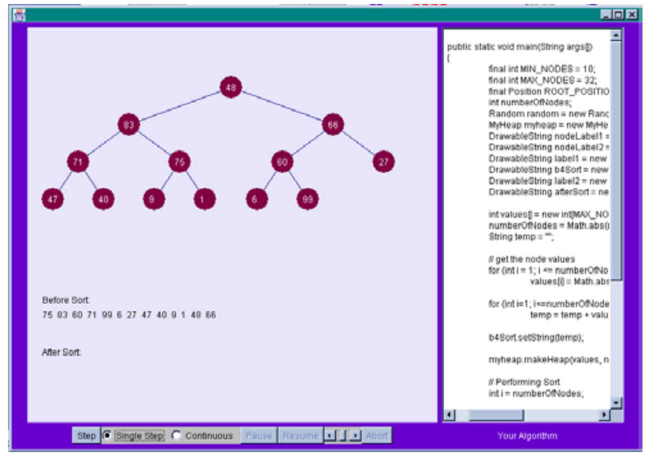

**Figure 30 :** Before Heap Sort

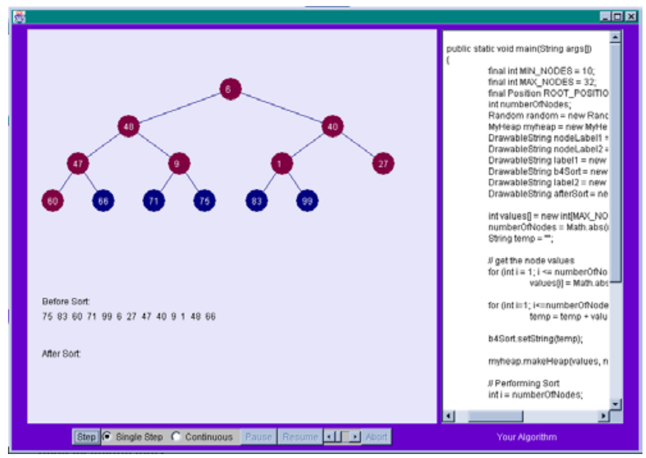

**Figure 31**: Heap Building in progress

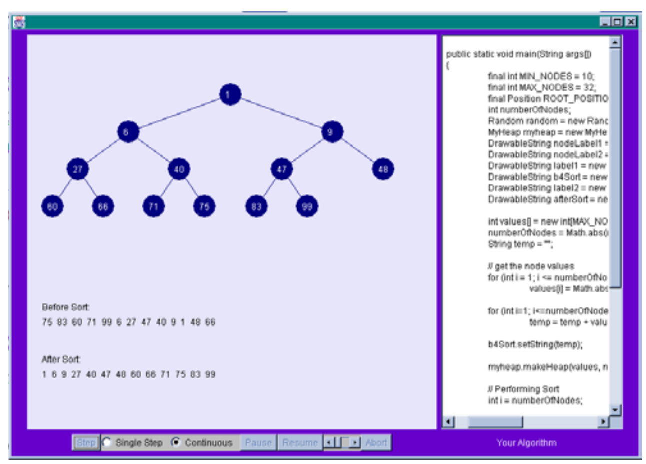

**Figure 32**: Heap Sort Done

### 3.2.3.4 Breadth-first Search Algorithm

The breadth-first search algorithm takes a graph and a vertex in the graph known as the source, and visits (performs functions on) each node that can be reached from the source by traversing the edges. In doing so, it is easy to determine which vertices can be reached from the source. The algorithm for breadth-first search from a source vertex s in a graph g is as follows:

```
enqueue the source vertex;
repeat
  dequeue u;
  perform any relevant operations on u;
  enqueue all the neighbors of u;
until the queue is empty
```

The algorithm coded in JavaMy is:

```
// Breadth First Search
public static void main(String args[])
{
  final Position GRAPH_POSITION = new
               Position(80, 80);
  MyGraph myGraph = new MyGraph(
               GRAPH_POSITION, 4, 8, true);
  DrawableString label = new DrawableString(
       new Position(20, 390),
       "Visited Nodes(Breadth First):");
  DrawableString traversalList = new
       DrawableString(new Position(20, 420));
  label.setColor(Color.blue);
  traversalList.setColor(Color.red);
  myGraph.makeGraph(2,2);
  myGraph.init();
  // search the graph
  int depth = 0;
  int current = myGraph.initSearch(false);
  Vector nextQueues[] = new
       Vector[myGraph.getNumOfNodes()];
  nextQueues[depth] = new Vector();
  Position positions[] = new Position[1];
  positions[0] =
               myGraph.nodePosition(current);
  myGraph.circle.moveTo(positions);
  myGraph.traceAndMark(current,
                      traversalList);
  myGraph.setNexts(current,
                      nextQueues[depth]);
  nextQueues[++depth] = new Vector();
  while (!myGraph.empty(
               nextQueues[depth - 1]))
  {
    current = myGraph.getNext(
               nextQueues[depth - 1]);
    positions[0] =
               myGraph.nodePosition(current);
    myGraph.circle.moveTo(positions);
    myGraph.traceAndMark(current,
                      traversalList);
    myGraph.setNexts(current,
                      nextQueues[depth]);
    if (nextQueues[depth - 1].size() == 0)
       nextQueues[++depth] = new Vector();
  }
  myGraph.circle.hide();
}
```

This example demonstrates usage of the observable data structure MyGraph. Some snapshots of the animation are shown in Figures 33-35.

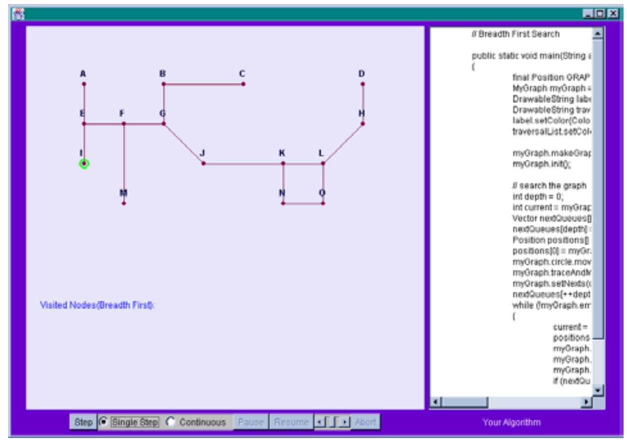


**Figure 33**: Graph to be searched

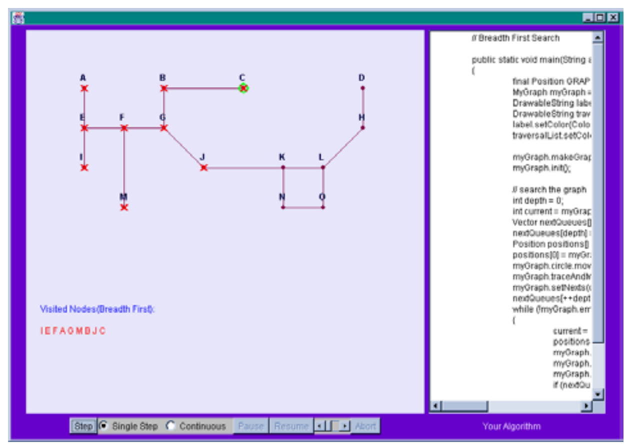


**Figure 34**: Breadth-first Search in progress

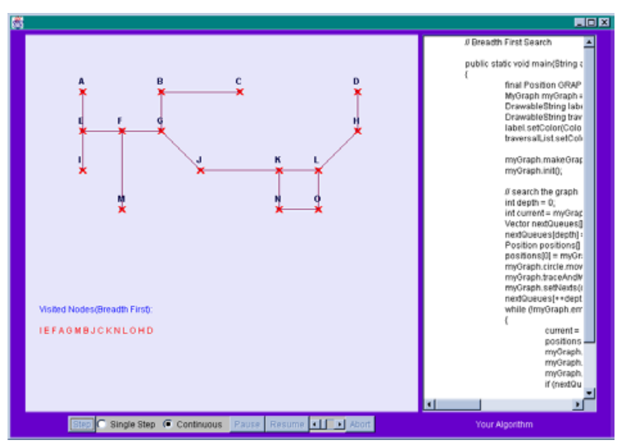


**Figure 35**: Breadth-first Search done

3.2.3.5 Depth-first Search Algorithm

Depth first search is another way of traversing graphs, which is closely related to a preorder traversal of a tree.

The algorithm coded in JavaMy:

```
// Depth First Search
public static void main(String args[])
{
  final Position GRAPH_POSITION = new
               Position(80, 80);
  MyGraph myGraph = new MyGraph(
               GRAPH_POSITION, 4, 8, true);
  DrawableString label = new
        DrawableString(new Position(20, 390),
               "Visited Nodes(Depth First):");
  DrawableString traversalList = new
         DrawableString(new Position(20, 420));
  label.setColor(Color.blue);
  traversalList.setColor(Color.red);
  myGraph.makeGraph(2,2);
  myGraph.init();
  int current = myGraph.initSearch(true);
  // search the graph
  depthFirstSearch(myGraph,
                current,traversalList);
  myGraph.arrow.hide();
}

private static void depthFirstSearch(MyGraph
myGraph, int current,DrawableString
traversalList)
{
  if (!myGraph.marked[current])
  {
    myGraph.arrow.setDirection(
       myGraph.nodePosition(current), true);
    Position positions[] = new Position[1];
    positions[0]=
              myGraph.nodePosition(current);
    myGraph.arrow.moveTo(positions);
    myGraph.traceAndMark(current,
                     traversalList);
    Vector nextQueue = new Vector();
    myGraph.setNexts(current, nextQueue);
    while (!myGraph.empty(nextQueue))
    {
       int next = myGraph.getNext(nextQueue);
       depthFirstSearch(myGraph, next,
                            traversalList);

       myGraph.arrow.setDirection(
        myGraph.nodePosition(current), false);
       positions[0] =
              myGraph.nodePosition(current);

       myGraph.arrow.moveTo(positions);
       myGraph.traceAndMark(current,
                     traversalList);
    }
  }//end if
}
```

This example uses the observable data structure MyGraph and the helper class DrawableString. The resulting animation is shown in Figures 36-38.

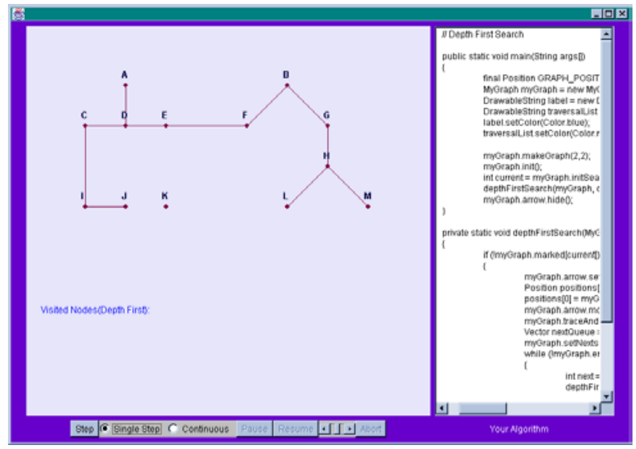


**Figure 36**: Graph to be Depth-first searched

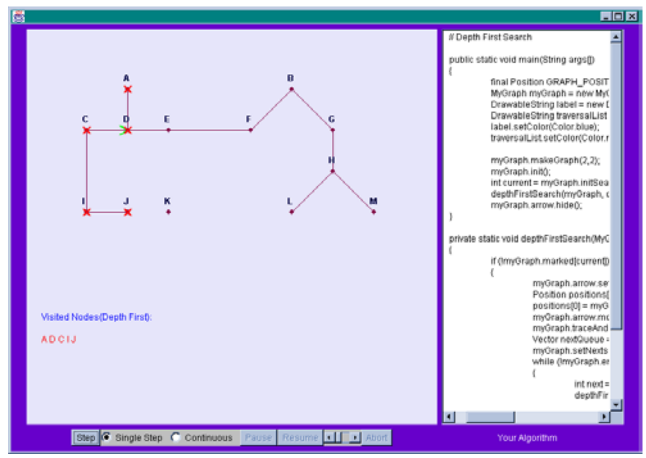


**Figure 37**: Depth-first Search in progress

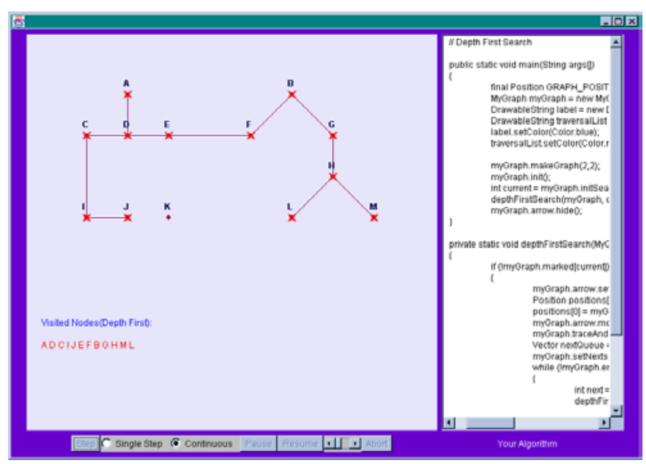


**Figure 38**: Depth-first Search Done

3.2.3.6 In-order Tree Traversal

In-order traversal is a technique for recursively processing the node of a tree in which the left subtree is processed first, then the root, and finally the right subtree. The pseudocode of in-order traversing a binary tree is:

*inorder(tree)*
*begin*
*if tree is null, return;*
*inorder(tree.left_subtree);*
*print(tree.root);*
*inorder(tree.right_subtree);*
*end*

The pseudocode coded in JavaMy is:

```
// In-order Tree Traversal
static String travlListString;

public static void main(String args[])
{
  final int MIN_NODES = 10;
  final int MAX_NODES = 32;
  final Position ROOT_POSITION = new
              Position(273, 80);
  Random random = new Random();
  MyTreeNode myTree = new MyTreeNode(
              1,1,ROOT_POSITION, null);
  DrawableString label = new
       DrawableString(new Position(20, 390),
              "Visited Nodes(In-order):");
  DrawableString traversalList = new
       DrawableString(new Position(20, 420));
  label.setColor(Color.blue);
  traversalList.setColor(Color.red);
  travlListString = new String();

  int numberOfNodes = Math.abs(
              random.nextInt() % MAX_NODES);
  numberOfNodes = Math.max(numberOfNodes,
                     MIN_NODES);
  numberOfNodes = myTree.randomizeShape(
                     numberOfNodes);
  // make the binary tree
  myTree.randomizeValues(2);
  inOrderTraversal(myTree, traversalList);
       // in-order traversal the binary tree
}

// in-order traversal
private static void inOrderTraversal(
MyTreeNode node, DrawableString traversalList)
{
  if (!node.isHidden())
  {
    inOrderTraversal((MyTreeNode)node.left(),
                          traversalList);
    node.changeColor(Color.red);
    travlListString = travlListString +
                     node.getValue()+ " ";
    traversalList.setString(travlListString);
    inOrderTraversal((MyTreeNode)node.right(),
                          traversalList);
  }
}
```

This example uses the observable data structure MyTreeNode. It also demonstrates how to code a recursive function in JavaMy language. The recursive function must be coded as a static function since it is invoked in the static main function. The algorithm animation is shown in Figures 39-41.

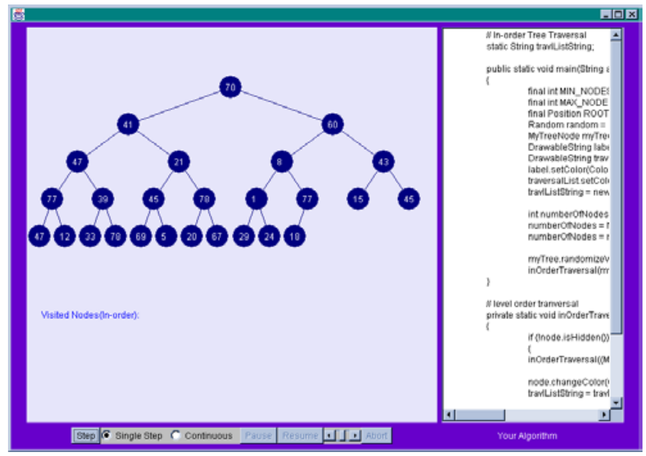


Figure 39: In-order traversal of a binary tree

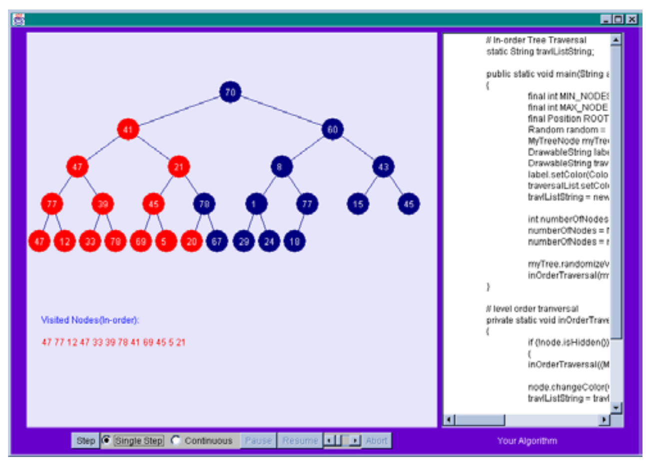


Figure 40: In-order traversal in progress

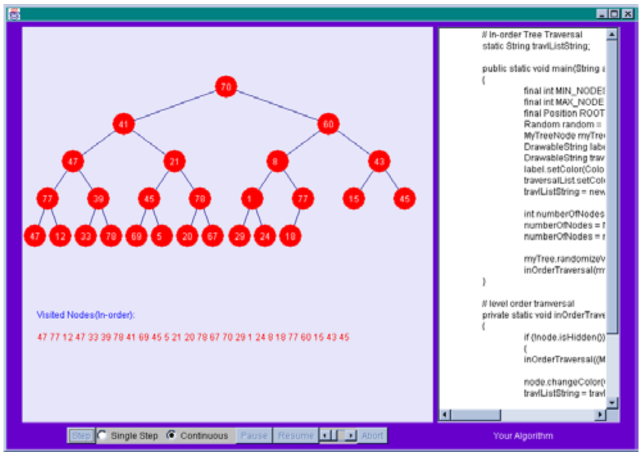

**Figure 41**: In-order traversal done

### 3.2.3.7 Pre-order Tree Traversal

Pre-order is another technique for recursively processing the nodes of a tree. Pre-order is similar to in-order except that, the root is processed first, then the left and right subtrees. The pseudo code is:

*preorder(tree)*
*begin*
*if tree is null, return;*

*print(tree.root);*
*preorder(tree.left_subtree);*
*preorder(tree.right_subtree);*
*end*

The corresponding JavaMy code is:

```
// Pre-order Tree Traversal
static String travlListString;

public static void main(String args[])
{
  final int MIN_NODES = 10;
  final int MAX_NODES = 32;
  final Position ROOT_POSITION = new
             Position(273, 80);
  Random random = new Random();
  MyTreeNode myTree = new MyTreeNode(
             1,1,ROOT_POSITION, null);
  DrawableString label = new
      DrawableString(new Position(20, 390),
             "Visited Nodes(Pre-order):");
  DrawableString traversalList = new
      DrawableString(new Position(20, 420));
  label.setColor(Color.blue);
  traversalList.setColor(Color.red);
  travlListString = new String();

  int numberOfNodes = Math.abs(
      random.nextInt() % MAX_NODES);
  numberOfNodes = Math.max(numberOfNodes,
             MIN_NODES);
  numberOfNodes = myTree.randomizeShape
             (numberOfNodes);
  // make the binary tree
  myTree.randomizeValues(2);
  preOrderTraversal(myTree, traversalList);
  // traversal the binary tree
}

// pre-order tranversal
private static void preOrderTraversal(
MyTreeNode node, DrawableString traversalList)
{
  if (!node.isHidden())
  {
    node.changeColor(Color.red);
    travlListString = travlListString +
             node.getValue()+ " ";
    traversalList.setString(travlListString);
    preOrderTraversal((MyTreeNode)node.left(),
                 traversalList);
    preOrderTraversal((MyTreeNode)node.right()
             , traversalList);
  }
}
```

The animation is shown in Figures 42-44.

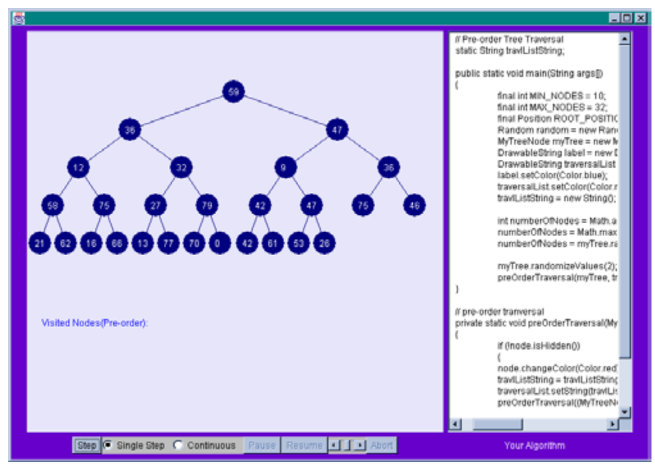

**Figure 42**: Pre-order traversal a binary tree

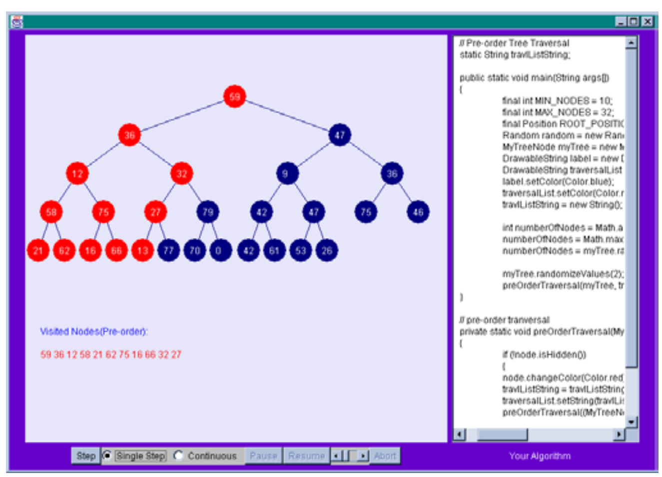

**Figure 43**: Pre-order traversal in progress

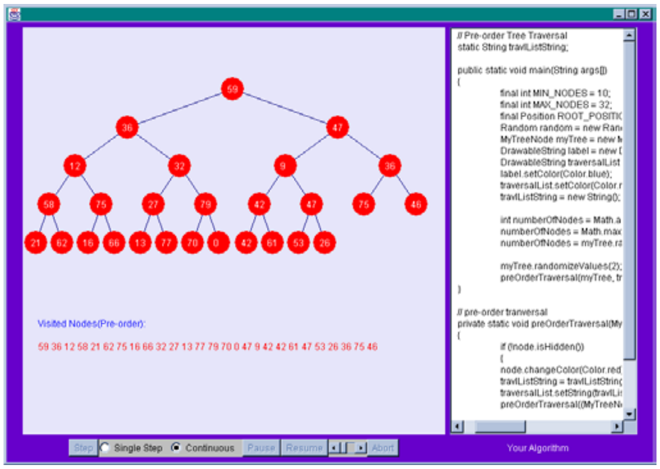


**Figure 44**: Pre-order traversal done

### 3.2.3.8 Post-order Tree Traversal

In Post-order traversal the left and right subtrees are processed first, then the root is processed. The pseudo code is:

*postorder(tree)*
*begin*
*if tree is null, return;*

*postorder(tree.left_subtree);*
*postorder(tree.right_subtree);*
*print(tree.root);*
*end*

Translated in JavaMy we have:

```
// Post-order Tree Traversal
static String travlListString;

public static void main(String args[])
{
  final int MIN_NODES = 10;
  final int MAX_NODES = 32;
  final Position ROOT_POSITION = new
              Position(273, 80);
  Random random = new Random();
  MyTreeNode myTree = new MyTreeNode(
              1,1,ROOT_POSITION, null);
  DrawableString label = new
       DrawableString(new Position(20, 390),
       "Visited Nodes(Post-order):");
  DrawableString traversalList = new
       DrawableString(new Position(20, 420));
  label.setColor(Color.blue);
  traversalList.setColor(Color.red);
  travlListString = new String();

  int numberOfNodes = Math.abs(
              random.nextInt() % MAX_NODES);
  numberOfNodes = Math.max(numberOfNodes,
                     MIN_NODES);
  numberOfNodes = myTree.randomizeShape(
                     numberOfNodes);
  // make the binary tree
  myTree.randomizeValues(2);
  postOrderTraversal(myTree, traversalList);
       // traversal the binary tree
}

// post-order tranversal
private static void postOrderTraversal(
 MyTreeNode node,DrawableString traversalList)
{
  if (!node.isHidden())
  {
    postOrderTraversal((MyTreeNode)node.left()
                  , traversalList);
   postOrderTraversal((MyTreeNode)node.right()
                  , traversalList);
    node.changeColor(Color.red);
    travlListString = travlListString +
                  node.getValue()+ " ";
    traversalList.setString(travlListString);
  }
}
```

The animation of the post-order traversal is shown in Figure 45-47.

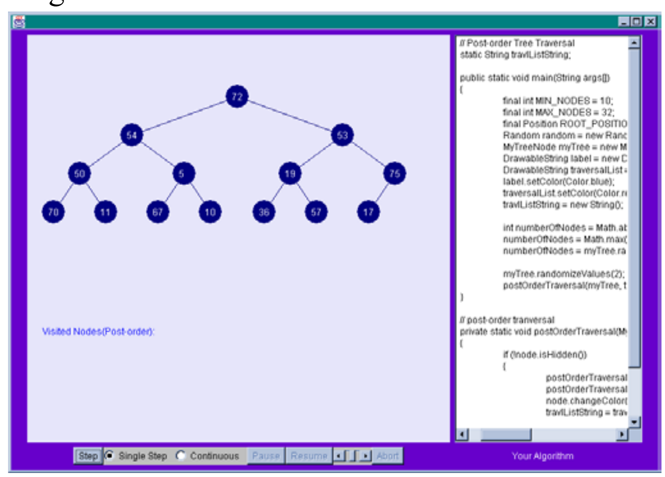


**Figure 45:** Post-order traversal

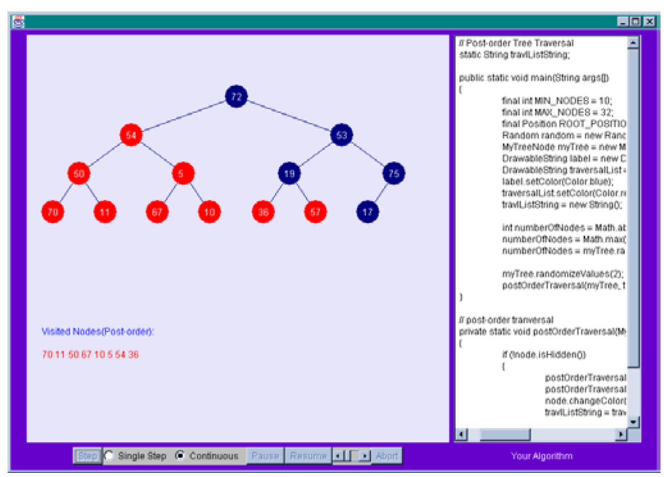


**Figure 46**: Post-order traversal in progress

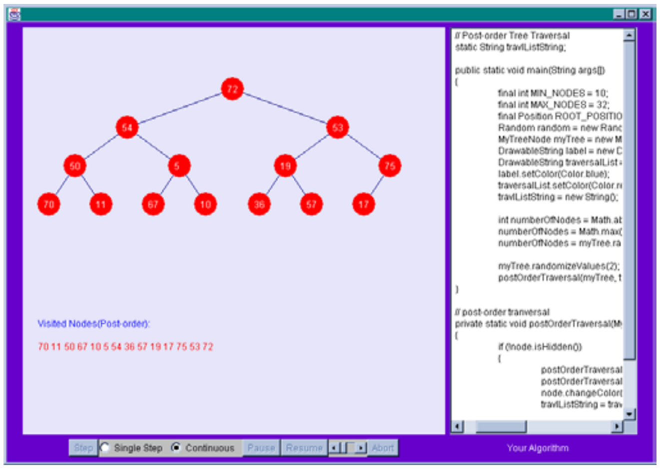


**Figure 47**: Post-order traversal done

3.2.3.9 Level-order Tree Traversal

In a level-order traversal, nodes are processed from top to bottom, left to right. It is implemented by using a queue. The JavaMy code of the algorithm is shown as follows.

```
// Level-order Tree Traversal
static String travlListString;

public static void main(String args[])
{
  final int MIN_NODES = 10;
  final int MAX_NODES = 32;
  final Position ROOT_POSITION = new
              Position(273, 80);
  Random random = new Random();
  MyTreeNode myTree = new MyTreeNode(
              1,1,ROOT_POSITION, null);
  DrawableString label = new
       DrawableString(new Position(20, 390),
       "Visited Nodes(Level Order):");
  DrawableString traversalList = new
       DrawableString(new Position(20, 420));
  label.setColor(Color.blue);
  traversalList.setColor(Color.red);
  travlListString = new String();

  Vector queue = new Vector();
  int numberOfNodes = Math.abs(
       random.nextInt() % MAX_NODES);
  numberOfNodes = Math.max(numberOfNodes,
              MIN_NODES);
  numberOfNodes = myTree.randomizeShape(
                     numberOfNodes);

  myTree.randomizeValues(2);
  // make the binary tree
  levelTraversal(myTree, queue,
              traversalList);
  // level traversal the binary tree
}
```

```
// level order traversal
private static void levelTraversal(MyTreeNode
       node, Vector queue, DrawableString
       traversalList)
{
  MyTreeNode next;
  if (!node.left().isHidden())
    queue.addElement(node.left());
  if (!node.right().isHidden())
    queue.addElement(node.right());
  node.changeColor(Color.red);
  travlListString = travlListString +
              node.getValue()+ " ";
  traversalList.setString(travlListString);

  if (queue.size() > 0)
  {
    next = (MyTreeNode) queue.firstElement();
    queue.removeElement(next);
    levelTraversal(next, queue,
              traversalList);
  }
}
```

The queue in level-order traversal is used to store nodes that are yet to be visited. When a node is visited, its children are placed at the end of the queue, to be visited after the nodes that are already in the queue. In this example, we choose not to view the content of the queue, therefore, a class Vector provided by Java is used instead of the observable data structure MyQueue. This further demonstrates that the user can determine which data structures he/she wants to observe, then chooses the classes accordingly. The level-order tree traversal animation is shown in Figure 48-50.

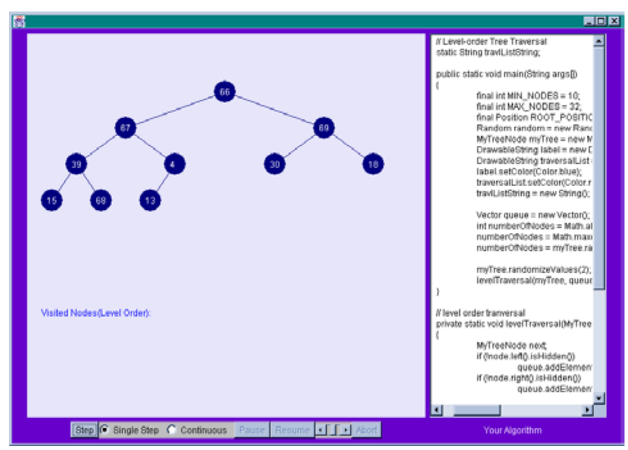


**Figure 48**: Level-order Tree Traversal

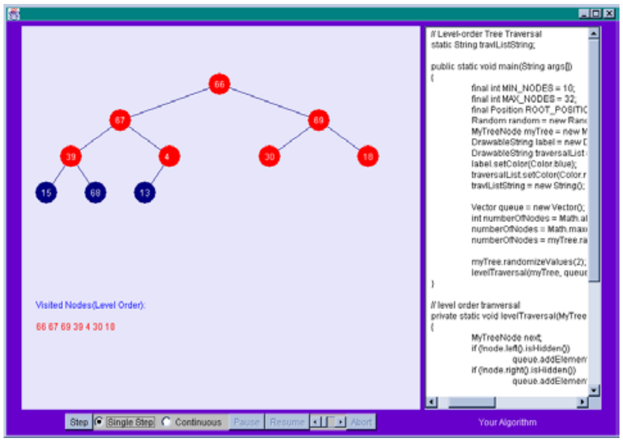


**Figure 49:** Level-order tree traversal in progress

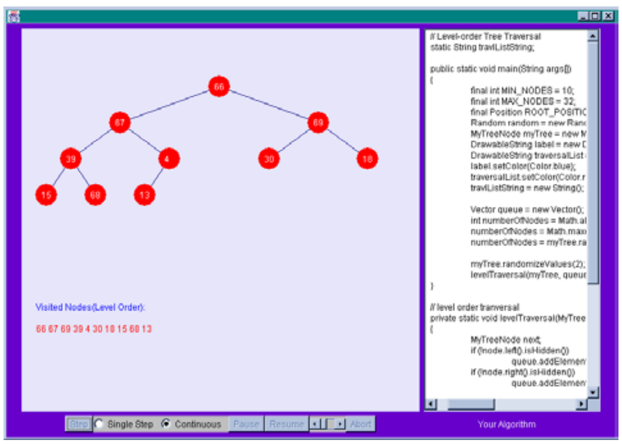


**Figure 50:** Level-order tree traversal done

## 4. Implementation

This software package is implemented using Java. Java is a general-purpose object-oriented language. The AWT and Swing packages of Java provide extensive components for creating Graphic User Interfaces. Moreover, its graphics capabilities are platform independent and hence portable, which makes it our natural choice for implementation.

To animate a user-defined algorithm, a lexical analyzer and parser are needed. A lexical analyzer breaks an input stream of characters into tokens. A parser reads the input tokens and converts the tokens to a Java program. There are several ways to build a lexer and parser. One possibility would be to code the lexical analyzer and parser completely from scratch, implementing all string handling and checking functions, which is a very tedious and error prone process. Another method is to find a Java parser generator, which reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. After intensive search, we found that JavaCC [10], a product of Sun Microsystems is currently the most popular parser generator for use with Java applications. Consequently, it was our choice. The parser is generated by two steps: (1) Run JavaCC on the grammar input file to generate a set of Java files that implement the parser and the lexer. (2) Compile all the Java files obtained in step (1). The grammar file for JavaMy language is shown in Appendix A.

## 5. Conclusions and future works

In this paper, we present a visualization tool designed to aid first-year computer science students learn Data Structures and Algorithms. This tool not only lets students visualize the commonly used data structures, but also allows students to write their own algorithms in a Java similar language - JavaMy, and observe the execution of the algorithms. We believe this tool will be an effective supplement to traditional instruction.

Because of the time limitation, only the most commonly used data structures are implemented in this version of the software package, which include arrays, stacks, queues, binary search tree, binary heap, priority queue and undirected graph. There are two ways to add more observable data structures to this software such as directed graph, weighted graph, AVL tree, Red Black Tree, AA- tree, splay tree, hash table, etc. One way is to implement these data structures in the software. Another approach would be to develop and implement a mechanism for the software package to recognize the user-defined observable data structures, and leave the implementation to the user. This approach will allow users to use their own observable data structures, hence add more flexibility to the software.

Another possible future enhancement for the software is to highlight the executing command line of the user-defined algorithm file. This would help the user to better follow the execution of the algorithm.

## References

[1] Morris, John, "Programming Languages and Data Structures", http://swww.ee.uwa.edu.au/~plsd210/ds/ds_ToC.html

[2] Cawsey, Alison, "Data Structures and Algorithms", http://www.cee.hw.ac.uk/~alison/ds98/ds98.html

[3] Owens, Brad "CS300 Data Structures and Algorithms I", http://www.cs.twsu.edu/~bjowens/cs300/

[4] Cohen, Edith "CS270: Combinatorial Algorithms and Data Structures", http://www.cs.berkeley.edu/~edith/cs270/

[5] Goodrich, Michael T. and Tamassia, Roberto, "Data Structures and Algorithms in Java", http://www.cs.brown.edu/courses/cs016/book/

[6] Jarc, Duane J., "Interactive Data Structure Visualizations", http://www.seas.gwu.edu/~idsv/idsv.html

[7] The Graphics, Visualization & Usability (GVU) Center at Georgia Tech, "XTango", http://www.cc.gatech.edu/gvu/softviz/algoanim/xtango.html

[8] The Graphics, Visualization & Usability (GVU) Center at Georgia Tech, "Polka", http://www.cc.gatech.edu/gvu/softviz/algoanim/xtango.html

[9] System Research Centers (SRC) at Compaq Computer Corporation, "Algorithm Animation at SRC", http://www.research.compaq.com/SRC/zeus/home.html

[10] Sun Microsystems, "JavaCC – The Java Parser Generator", http://www.metamata.com/javacc/

## Appendix A JavaMy grammar file

```
options {
  MULTI = true;
  NODE_DEFAULT_VOID = true;
  JAVA_UNICODE_ESCAPE = true;
}

PARSER_BEGIN(AlgorithmParser)

import java.io.*;
import project.*;

public class AlgorithmParser
{
  public static void main(String args[])
       throws Exception {
    AlgorithmParser parser;
    ASTCompilationUnit node;

    if (args.length == 2) {
      System.out.println("Algorithm
Preprocessor:  Reading from file " + args[0] +
" . . .");
      try {
        parser = new AlgorithmParser(new
FileInputStream(args[0]));
      } catch (FileNotFoundException e) {
        System.out.println("Algorithm
Preprocessor:  File " + args[0] + " not
found.");
        return;
      }
    } else {
      System.out.println("Algorithm
Preprocessor:  Usage is \"java AlgorihtmParser
inputfile outputfile\"");
      return;
    }
    try {
      node = parser.CompilationUnit();
      PrintWriter ostr = new PrintWriter(new
FileWriter(args[1]));
      node.process(ostr, args[0]);
      ostr.close();
      System.out.println("Algorithm
Preprocessor:  Transformation completed
successfully.");
    } catch (ParseException e) {
      System.out.println("Algorithm
Preprocessor:  Encountered errors during
parse.");
      System.out.println(""+e);
    } catch (IOException e) {
      System.out.println("Algorithm
Preprocessor:  Could not create file " +
args[1]);
    }
  }
}

PARSER_END(AlgorithmParser)

SPECIAL_TOKEN : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
```

```
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])*
("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" |
(~["*","/"] (~["*"])* "*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*"
| (~["*","/"] (~["*"])* "*"))* "/">
}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
| < BREAK: "break" >
| < BYTE: "byte" >
| < CASE: "case" >
| < CATCH: "catch" >
| < CHAR: "char" >
| < CLASS: "class" >
| < CONST: "const" >
| < CONTINUE: "continue" >
| < _DEFAULT: "default" >
| < DO: "do" >
| < DOUBLE: "double" >
| < ELSE: "else" >
| < EXTENDS: "extends" >
| < FALSE: "false" >
| < FINAL: "final" >
| < FINALLY: "finally" >
| < FLOAT: "float" >
| < FOR: "for" >
| < GOTO: "goto" >
| < IF: "if" >
| < IMPLEMENTS: "implements" >
| < IMPORT: "import" >
| < INSTANCEOF: "instanceof" >
| < INT: "int" >
| < INTERFACE: "interface" >
| < LONG: "long" >
| < NATIVE: "native" >
| < NEW: "new" >
| < NULL: "null" >
| < PACKAGE: "package">
| < PRIVATE: "private" >
| < PROTECTED: "protected" >
| < PUBLIC: "public" >
| < RETURN: "return" >
| < SHORT: "short" >
| < STATIC: "static" >
| < SUPER: "super" >
| < SWITCH: "switch" >
| < SYNCHRONIZED: "synchronized" >
| < THIS: "this" >
| < THROW: "throw" >
| < THROWS: "throws" >
| < TRANSIENT: "transient" >
| < TRUE: "true" >
| < TRY: "try" >
| < VOID: "void" >
| < VOLATILE: "volatile" >
| < WHILE: "while" >
| < MYARRAY: "MyArray" >
| < MYSTACK: "MyStack">
| < MYQUEUE: "MyQueue" >
| < MYHEAP: "MyHeap" >
| < MYPQHEAP: "MyPQHeap" >
| < MYGRAPH: "MyGraph" >
| < MYTREENODE: "MyTreeNode">
| < DRAWABLESTRING: "DrawableString" >
}

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
| < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
| < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-
"f","A"-"F"])+ >
| < #OCTAL_LITERAL: "0" (["0"-"7"])* >
| < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])*
(<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)?
(["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT>
(["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)?
["f","F","d","D"]
  >
| < #EXPONENT: ["e","E"] (["+","-"])? (["0"-
"9"])+ >
| < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            (
["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
| < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            (
["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
```

```
        "\u00f8"-"\u00ff",
        "\u0100"-"\u1fff",
        "\u3040"-"\u318f",
        "\u3300"-"\u337f",
        "\u3400"-"\u3d2d",
        "\u4e00"-"\u9fff",
        "\uf900"-"\ufaff"
       ]
  >
|  < #DIGIT:
       [
        "\u0030"-"\u0039",
        "\u0660"-"\u0669",
        "\u06f0"-"\u06f9",
        "\u0966"-"\u096f",
        "\u09e6"-"\u09ef",
        "\u0a66"-"\u0a6f",
        "\u0ae6"-"\u0aef",
        "\u0b66"-"\u0b6f",
        "\u0be7"-"\u0bef",
        "\u0c66"-"\u0c6f",
        "\u0ce6"-"\u0cef",
        "\u0d66"-"\u0d6f",
        "\u0e50"-"\u0e59",
        "\u0ed0"-"\u0ed9",
        "\u1040"-"\u1049"
       ]
  >
}

TOKEN : /* SEPARATORS */
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
}

TOKEN : /* OPERATORS */
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < BANG: "!" >
| < TILDE: "~" >
| < HOOK: "?" >
| < COLON: ":" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < SC_OR: "||" >
| < SC_AND: "&&" >
| < INCR: "++" >
| < DECR: "--" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < BIT_AND: "&" >
| < BIT_OR: "|" >
| < XOR: "^" >
| < REM: "%" >
| < LSHIFT: "<<" >
| < RSIGNEDSHIFT: ">>" >
| < RUNSIGNEDSHIFT: ">>>" >
| < PLUSASSIGN: "+=" >
| < MINUSASSIGN: "-=" >
| < STARASSIGN: "*=" >
| < SLASHASSIGN: "/=" >
| < ANDASSIGN: "&=" >
| < ORASSIGN: "|=" >
| < XORASSIGN: "^=" >
| < REMASSIGN: "%=" >
| < LSHIFTASSIGN: "<<=" >
| < RSIGNEDSHIFTASSIGN: ">>=" >
| < RUNSIGNEDSHIFTASSIGN: ">>>=" >
}

/*********************************************
* THE ALGORITHM LANGUAGE GRAMMAR STARTS HERE *
*********************************************/
/* Program structuring syntax follows. */
ASTCompilationUnit CompilationUnit()
#CompilationUnit :
{}
{
       {
       jjtThis.setFirstToken(getToken(1));
       }
  ( ImportDeclaration() )*
  ( BodyDeclaration() )*
  <EOF>
              {
                return jjtThis;
              }
}

void ImportDeclaration() :
{}
{
  "import" Name() [ "." "*" ] ";"
}

/* Declaration syntax follows. */
void BodyDeclaration() :
{}
{
  LOOKAHEAD( MethodDeclarationLookahead() )
  MethodDeclaration()
|
  FieldDeclaration()
}

//This production is to determine lookahead
//only.
void MethodDeclarationLookahead() :
{}
{
  ( "public" | "protected" | "private" |
"static" | "abstract" | "final" | "native" |
"synchronized" )*
  ResultType() <IDENTIFIER> "("
}

void FieldDeclaration() :
{ }
{
  ( "public" | "protected" | "private" |
"static" | "final" | "transient" | "volatile"
)*
  (    ShowVariableDeclaration()
```

```
    |  Type() VariableDeclarator() ( ","
VariableDeclarator() )*) ";"
}

void ShowVariableDeclaration() :
{      Token t;
}
{      { t = getToken(1); }
  ( (  "MyArray" <IDENTIFIER> "=" "new"
"MyArray" Arguments()
    | "MyStack" <IDENTIFIER> "=" "new"
"MyStack" Arguments()
    | "MyQueue" <IDENTIFIER> "=" "new"
"MyQueue" Arguments()
    | "MyHeap" <IDENTIFIER> "=" "new" "MyHeap"
Arguments()
    | "MyPQHeap" <IDENTIFIER> "=" "new"
"MyPQHeap" Arguments()
    | "MyGraph" <IDENTIFIER> "=" "new"
"MyGraph" Arguments()
    | "MyTreeNode" <IDENTIFIER> "=" "new"
"MyTreeNode" Arguments()
    | "DrawableString" <IDENTIFIER> "=" "new"
"DrawableString" Arguments())
       {      jjtThis.setFirstToken(t);

       jjtThis.setLastToken(getToken(0));
       }
  )#ShowBlock
}

void Comma() :
{
       Token t;
}
{
  ";"
  (          {
               t = getToken(1);
               jjtThis.setFirstToken(t);
         jjtThis.setLastToken(getToken(0));
             }
    ) #SpecialBlock

}

void VariableDeclarator() :
{}
{
  VariableDeclaratorId() [ "="
VariableInitializer() ]
}

void VariableDeclaratorId() :
{}
{
  <IDENTIFIER> ( "[" "]" )*
}

void VariableInitializer() :
{}
{
  ArrayInitializer()
|
  Expression()
}

void ArrayInitializer() :
{}
{
  "{" [ VariableInitializer() ( LOOKAHEAD(2)
"," VariableInitializer() )* ] [ "," ] "}"
}

void MethodDeclaration() :
{}
{
  ( "public" | "protected" | "private" |
"static" | "abstract" | "final" | "native" |
"synchronized" )*
  ResultType() MethodDeclarator() [ "throws"
NameList() ]
  ( Block() | ";"  )
}

void MethodDeclarator() :
{}
{
  <IDENTIFIER> FormalParameters() ( "[" "]" )*
}

void FormalParameters() :
{}
{
  "(" [ FormalParameter() ( ","
FormalParameter() )* ] ")"
}

void FormalParameter() :
{}
{
  [ "final" ] Type() VariableDeclaratorId()
}

void Initializer() :
{}
{
  [ "static" ] Block()
}


/* Type, name and expression syntax follows.*/
void Type() :
{}
{
  ( PrimitiveType() | Name() | "MyArray"
|"MyQueue" |"MyStack" |"MyHeap"|"MyPQHeap"
|"MyGraph"|"MyTreeNode"|"DrawableString") (
"[" "]" )*
}

void PrimitiveType() :
{}
{
  "boolean"
|  "char"
|  "byte"
|  "short"
|  "int"
|  "long"
|  "float"
|  "double"
}

void ResultType() :
{}
{
  "void"
```

```
|  Type()
}

void Name() :
/* A lookahead of 2 is required below since
"Name" can be followed
 * by a ".*" when used in the context of an
"ImportDeclaration".
 */
{}
{
  <IDENTIFIER>
  ( LOOKAHEAD(2) "." <IDENTIFIER>  )*
}

void NameList() :
{}
{
  Name()
  ( "," Name() )*
}


/* Expression syntax follows. */
void Expression() :
{}
{
  LOOKAHEAD( PrimaryExpression()
AssignmentOperator() )
  Assignment()
|
  ConditionalExpression()
}

void Assignment() :
{}
{
  PrimaryExpression() AssignmentOperator()
Expression()
}

void AssignmentOperator() :
{}
{
  "=" | "*=" | "/=" | "%=" | "+=" | "-=" |
"<<=" | ">>=" | ">>>=" | "&=" | "^=" | "|="
}

void ConditionalExpression() :
{}
{
  ConditionalOrExpression() [ "?" Expression()
":" ConditionalExpression() ]
}

void ConditionalOrExpression() :
{}
{
  ConditionalAndExpression() ( "||"
  ConditionalAndExpression() )*
}

void ConditionalAndExpression() :
{}
{
  InclusiveOrExpression() ( "&&"
  InclusiveOrExpression() )*
}
```

```
void InclusiveOrExpression() :
{}
{
  ExclusiveOrExpression() ( "|"
ExclusiveOrExpression() )*
}

void ExclusiveOrExpression() :
{}
{
  AndExpression() ( "^" AndExpression() )*
}

void AndExpression() :
{}
{
  EqualityExpression() ( "&"
EqualityExpression() )*
}

void EqualityExpression() :
{}
{
  InstanceOfExpression() ( ( "==" | "!=" )
InstanceOfExpression() )*
}

void InstanceOfExpression() :
{}
{
  RelationalExpression() [ "instanceof" Type()
]
}

void RelationalExpression() :
{}
{
  ShiftExpression() ( ( "<" | ">" | "<=" |
">=" ) ShiftExpression() )*
}

void ShiftExpression() :
{}
{
  AdditiveExpression() ( ( "<<" | ">>" | ">>>"
) AdditiveExpression() )*
}

void AdditiveExpression() :
{}
{
  MultiplicativeExpression() ( ( "+" | "-" )
MultiplicativeExpression() )*
}

void MultiplicativeExpression() :
{}
{
  UnaryExpression() ( ( "*" | "/" | "%" )
UnaryExpression() )*
}

void UnaryExpression() :
{}
{
  ( "+" | "-" ) UnaryExpression()
|  PreIncrementExpression()
|  PreDecrementExpression()
|  UnaryExpressionNotPlusMinus()
```

```
}

void PreIncrementExpression() :
{}
{
  "++" PrimaryExpression()
}

void PreDecrementExpression() :
{}
{
  "--" PrimaryExpression()
}

void UnaryExpressionNotPlusMinus() :
{}
{
  ( "~" | "!" ) UnaryExpression()
|  LOOKAHEAD( CastLookahead() )
  CastExpression()
|  PostfixExpression()
}

// This production is to determine lookahead
//only.  The LOOKAHEAD specifications
// below are not used, but they are there just
//to indicate that we know about
// this.
void CastLookahead() :
{}
{
  LOOKAHEAD(2)
  "(" PrimitiveType()
|  LOOKAHEAD("(" Name() "[")
  "(" Name() "[" "]"
|  "(" Name() ")" ( "~" | "!" | "(" |
<IDENTIFIER> | "this" | "super" | "new" |
Literal() )
}

void PostfixExpression() :
{}
{
  PrimaryExpression() [ "++" | "--" ]
}

void CastExpression() :
{}
{
  LOOKAHEAD("(" PrimitiveType())
  "(" Type() ")" UnaryExpression()
|  LOOKAHEAD("(" Name())
  "(" Type() ")" UnaryExpressionNotPlusMinus()
}

void PrimaryExpression() :
{}
{
  PrimaryPrefix() ( LOOKAHEAD(2)
PrimarySuffix() )*
}

void PrimaryPrefix() :
{}
{
  Literal()
|  Name()
|  "this"
|  "super" "." <IDENTIFIER>
|  "(" Expression() ")"
|  AllocationExpression()
}

void PrimarySuffix() :
{}
{
  LOOKAHEAD(2)
  "." "this"
|  LOOKAHEAD(2)
  "." "class"
|  LOOKAHEAD(2)
  "." AllocationExpression()
|  "[" Expression() "]"
|  "." <IDENTIFIER>
|  Arguments()
}

void Literal() :
{}
{
  <INTEGER_LITERAL>
|  <FLOATING_POINT_LITERAL>
|  <CHARACTER_LITERAL>
|  <STRING_LITERAL>
|  BooleanLiteral()
|  NullLiteral()
}

void BooleanLiteral() :
{}
{
  "true" |  "false"
}

void NullLiteral() :
{}
{
  "null"
}

void Arguments() :
{}
{
  "(" [ ArgumentList() ] ")"
}

void ArgumentList() :
{}
{
  Expression() ( "," Expression() )*
}

void AllocationExpression() :
{}
{
  LOOKAHEAD(2)
/*  SpecialAllocation()
| LOOKAHEAD(2) */
  "new" PrimitiveType() ArrayDimensions() [
ArrayInitializer() ]
|  "new" Name()
    (
      ArrayDimensions() [ ArrayInitializer() ]
    |      Arguments()
    )
}
```

```
/* The second LOOKAHEAD specification below is
to parse to PrimarySuffix
 * if there is an expression between the
"[...]".
 */
void ArrayDimensions() :
{}
{
  ( LOOKAHEAD(2) "[" Expression() "]" )+ (
LOOKAHEAD(2) "[" "]" )*
}

/* Statement syntax follows. */
void Statement() :
{}
{
  LOOKAHEAD(2)
  LabeledStatement()
|  Block()
|  EmptyStatement()
|  StatementExpression() Comma()
|  SwitchStatement()
|  IfStatement()
|  WhileStatement()
|  DoStatement()
|  ForStatement()
|  BreakStatement()
|  ContinueStatement()
|  ReturnStatement()
|  ThrowStatement()
|  SynchronizedStatement()
|  TryStatement()
}

void LabeledStatement() :
{}
{
  <IDENTIFIER> ":" Statement()
}

void Block() :
{}
{
  "{" ( BlockStatement() )* "}"
}

void BlockStatement() :
{}
{
  LOOKAHEAD([ "final" ] (Type() <IDENTIFIER>)
| "MyArray"|"MyStack"|"MyQueue"|"MyHeap")
  LocalVariableDeclaration() ";"
|  Statement()

}

void LocalVariableDeclaration() :
{}
{  ShowVariableDeclaration()
 | [ "final" ] Type() VariableDeclarator() (
"," VariableDeclarator() )*
}

void EmptyStatement() :
{}
{
  ";"
}
```

```
void StatementExpression() :
/* The last expansion of this production
accepts more than the legal
 * Java expansions for StatementExpression.
 */
{}
{
  PreIncrementExpression()
|  PreDecrementExpression()
|  LOOKAHEAD( PrimaryExpression()
AssignmentOperator() )
  Assignment()
|  PostfixExpression()
}

void SwitchStatement() :
{}
{
  "switch" "(" Expression() ")" "{"
    ( SwitchLabel() ( BlockStatement() )* )*
  "}"
}

void SwitchLabel() :
{}
{
  "case" Expression() ":"
|  "default" ":"
}

void IfStatement() :
/*
 * The disambiguating algorithm of JavaCC
automatically binds dangling
 * else's to the innermost if statement.  The
LOOKAHEAD specification
 * is to tell JavaCC that we know what we are
doing.
 */
{}
{
  "if" "(" Expression() ")" Statement() [
LOOKAHEAD(1) "else" Statement() ]
}

void WhileStatement() :
{}
{
  "while" "(" Expression() ")" Statement()
}

void DoStatement() :
{}
{
  "do" Statement() "while" "(" Expression()
")" Comma()
}

void ForStatement() :
{}
{
  "for" "(" [ ForInit() ] ";" [ Expression() ]
";" [ ForUpdate() ] ")" Statement()
}

void ForInit() :
{}
{
  LOOKAHEAD( [ "final" ] Type() <IDENTIFIER> )
```

```
  LocalVariableDeclaration()
|  StatementExpressionList()
}

void StatementExpressionList() :
{}
{
  StatementExpression() ( ","
StatementExpression() )*
}

void ForUpdate() :
{}
{
  StatementExpressionList()
}

void BreakStatement() :
{}
{
  "break" [ <IDENTIFIER> ] ";"
}

void ContinueStatement() :
{}
{
  "continue" [ <IDENTIFIER> ] Comma()
}

void ReturnStatement() :
{}
{
  "return" [ Expression() ] ";"
}

void ThrowStatement() :
{}
{
  "throw" Expression() ";"
}

void SynchronizedStatement() :
{}
{
  "synchronized" "(" Expression() ")" Block()
}

void TryStatement() :
/* Semantic check required here to make sure
that at least one
 * finally/catch is present.
 */
{}
{
  "try" Block()
  ( "catch" "(" FormalParameter() ")" Block()
)*
  [ "finally" Block() ]
}
```