

DATA STRUCTURE VISUALIZATION

by

John A. Costigan, III

A thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science (in Honors)
Computer Science

Virginia Polytechnic Institute and State University

Copyright © 2002 by John Costigan
and Virginia Polytechnic Institute and State University
April 18, 2002

Approved by _____
Chairperson of Supervisory Committee

Program Authorized
to Offer Degree _____

Date _____

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

ABSTRACT

DATA STRUCTURE VISUALIZATION

by John A. Costigan

April 18, 2002

Chairperson of the Supervisory Committee: Professor North
Department of Computer Science

Data structure visualization deals with understandable representations of memory-based data structures. For years, students have learned how to understand data structures by using diagrams that provide a visual rendering of the binary information held in the data structures. These diagrams can serve not only as tools for learning specific classes of data structures (such as binary trees, vectors, or general trees), but also as useful ways of visualizing particular instances of such data structures. Usually, these diagrams are either hand-drawn or designed with general diagramming software. It would be useful if diagrams could be generated automatically, centered on a software project that is currently under development, to assist the developers by providing a human-understandable display of what the data structures in their program look like. This thesis presents guidelines for developing an ideal visual debugging tool and a description of the development progress of the Visual Debugger, a project aimed at implementing the given guidelines.

TABLE OF CONTENTS

Table of Contents	i
List of figures.....	ii
Acknowledgments.....	iii
Glossary.....	iv
Chapter 1: Introduction.....	1
Chapter 2: Background Information	5
2.1 Information Visualization	5
2.2 Graph Theory.....	10
2.3 Data Structures.....	13
2.4 Memory Management	14
Chapter 3: State of the Art in Data Structure Visualization	16
3.1 Graphing Algorithms.....	16
3.1.1 Circular Layout	16
3.1.2 The Layering Method	17
3.1.3 Physical Modeling	19
3.2 Data Structure Visualization Software.....	19
3.2.1 Dotty.....	20
3.2.2 Deet	21
3.2.3 VDBX	23
3.2.4 GNU Visual Debugger.....	24
3.2.5 Data Display Debugger.....	26
Chapter 4: The Data Structure Visualization Problem	28
Chapter 5: The Data Structure Visualization Solution.....	32
5.1 Guidelines	32
5.2 Current Progress	34
5.2.1 Implementation	48
Chapter 6: Conclusions and Future Work.....	50
6.1 Conclusions	50
6.2 Summary of Contributions	51
6.3 Future Research	51
References.....	54
Appendix A: the Visual Debugger DemoTree Code.....	55

LIST OF FIGURES

<i>Number</i>	<i>Page</i>
Figure 1: A diagram of a linked list	2
Figure 2: A vector of vectors	6
Figure 3: A 10x10 matrix	7
Figure 4: A grayscale 10x10 matrix	8
Figure 5: A smaller grayscale 10x10 matrix	9
Figure 6: A higher resolution grayscale matrix	9
Figure 7: A simple directed graph	10
Figure 8: A simple undirected graph	10
Figure 9: A simple tree	12
Figure 10: The same tree, but rooted	12
Figure 11: A circular directed graph layout	17
Figure 12: A layered directed graph layout	18
Figure 13: A graph displayed with Dot	20
Figure 14: A graph displayed with Neato	20
Figure 15: Deet	22
Figure 16: VDBX	23
Figure 17: The GNU Visual Debugger	25
Figure 18: The Data Display Debugger	26
Figure 19: A tree, using DDD's automatic layout algorithm	27
Figure 20: The Visual Debugger Initial Screen	35
Figure 21: Adding <code>tree</code> it to the stack list	36
Figure 22: The "Define Class" dialog box	37
Figure 23: About to enter <code>Node* root</code> as a field	37
Figure 24: The completed definition of <code>Node</code>	38
Figure 25: The completed definition of <code>Tree</code>	39
Figure 26: After adding <code>tree</code> to the stack list	40
Figure 27: The "Classes" dialog box	40
Figure 28: The <code>tree</code> variable has been initialized	41
Figure 29: The tree now has one node.	41
Figure 30: The tree with eight nodes	42
Figure 31: Selecting a node	43
Figure 32: Deletion of a node from the tree	44
Figure 33: After all the nodes have been removed from the tree	45
Figure 34: A tree with nineteen nodes	46

ACKNOWLEDGMENTS

The author wishes to thank: Dr. Chris North for the guidance that only a competent advisor can provide; Ben Wilhite for his assistance in developing the Visual Debugger; Matthew Sample and Sam Stone for the back end of the Visual Debugger; Dr. Cliff Shaffer and Dr. Scott McCrickard for serving on the thesis defense committee; and all of the above for their invaluable input concerning the thesis.

GLOSSARY

call stack. a block of memory that stores local data used for calls to functions. Also called a “frame stack” or just a “stack.”

child. in a rooted tree, the vertex on the sending end of a particular vertex’s incoming edge.

data structure. a collection of related data along with the operations that can be performed with the data.

digraph. in graph theory, a graph whose edges have direction. Also called a “directed graph.”

directed graph. in graph theory, a graph whose edges have direction. Also called a “digraph.”

edge. in graph theory, a relationship (either with or without direction) between two vertices.

frame stack. a block of memory that stores local data used for calls to functions. Also called a “call stack” or just a “stack.”

graph. in graph theory, a set of vertices (usually labeled) and a set of corresponding edges (sometimes labeled) between the vertices.

heap. a large block of memory that is dynamically allocated as necessary to store program data. Also called a “memory heap.”

leaf. in a rooted tree, a vertex with no children or whose children are all empty trees.

memory heap. a large block of memory that is dynamically allocated as necessary to store program data. Also called a “heap.”

memory leak. memory allocated on the heap without a reference with which the owner program can access it.

node. in graph theory, an element of a graph between which edges are connected. Also called a “vertex.”

parent. in a rooted tree, the vertex on the receiving end of a particular vertex's single outgoing edge.

pointer. a piece of data that provides access to another piece of data.

root. in graph theory, a special vertex—with either no outgoing edges or no incoming edges—that is designated to turn a tree into a rooted tree.

rooted. in graph theory, a tree is said to be rooted when there is a single distinguished vertex (often labeled or clearly indicated), called the root, that has no outgoing edges.

stack. in memory allocation, a block of memory that stores local data used for calls to functions. Also called a “frame stack” or a “call stack.”

tree. in graph theory, a connected graph with no cycles.

undirected graph. in graph theory, a graph whose edges have no direction.

vertex. in graph theory, an element of a graph between which edges are connected. Also called a “node.”

CHAPTER 1: INTRODUCTION

Information visualization is a field of computer science that deals with effective means of displaying or otherwise communicating data to human beings. Commonly, this involves displaying data in a way that is somewhat unrelated to its original form in order to enhance human ability to parse information. The organization of data in computer memory is often very different from the way it is displayed to humans, particularly because computer memory is limited by its linearity. In all situations that involve digital data, the simplest way to display the data would be to output the data as a stream of bits, just as the computer sees it. This approach, however, is not very human-friendly, so information visualization techniques are used to present digital data to humans in a more readable fashion.

Data structure visualization deals with understandable representations of memory-based data structures. Data structures are commonly used when implementing a software solution to a problem, and many computer science students learn about data structures relatively early in the programming curriculum. One problem that hampers students that are new to data structures is their physical invisibility. When coding up the answer to a problem, programmers cannot see the actual data structures—physically, they are only streams of bits in a computer’s memory, invisible to the programmer.

For years, students have learned how to understand data structures by using diagrams that provide a visual representation of the binary information held in the data structures. For example, linked lists are often described as a set of blocks connected by arrows (Figure 1). These diagrammatic representations are easier to

understand and facilitate the learning of individual data structures as well as the learning of how to design original data structures.

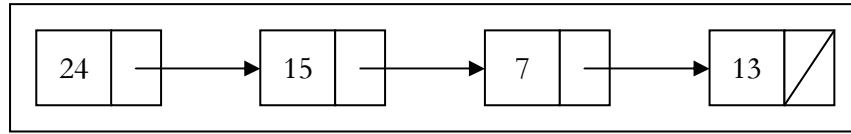


Figure 1: A diagram of a linked list

These diagrams can serve not only as tools for learning the layout of specific classes of data structures (such as binary trees, vectors, or skip lists), but also as useful ways of visualizing particular instances of these types of data structures. For example, the diagram illustrated in Figure 1 may have been concocted from the imagination of an instructor to help describe linked lists as a class of data structures. On the other hand, perhaps it represents actual data in a computer's memory at a particular point in time. Thus, these diagrams are also useful in visualizing instances of data structures during the execution of a program. The diagram in Figure 1 could represent the result of adding the element labeled "15" to the list. Similar diagrams could have been drawn to designate points before and during the insertion, and together the diagrams would help to demonstrate the operation of inserting a new element into a linked list.

Usually, these diagrams are either hand-drawn or designed with general diagramming software. Instructors may design them on the fly, referencing an imaginary list of elements that does not really exist in any computer's memory. Additionally, programmers may draw such diagrams when they are designing and implementing complex data structures, again without reference to actual data. Finally, programmers may also use diagrammatic representations while they are debugging a program that uses complex data structures; in this case, the diagrams represent data as it is stored in a computer's memory.

It would be useful if diagrams could be generated automatically to assist developers by providing a humanly understandable display of what the data structures in their programs look like. These visualizations could be updated dynamically as the debugged program proceeds; in this way, programmers can visually see how the data structures in their program are developing. They can then use that visual information to make decisions to improve the program. For example, if a programmer is having trouble implementing insertion of a new element into a linked list, then the set of automatically generated diagrams might be similar to the diagrams discussed in reference to Figure 1. The programmer can use the diagrams to quickly find the problem and implement a solution.

The goal of the Visual Debugger is to provide visually understandable representations of data structures during runtime, so that developers can better understand the actual layouts of their data structures and respond accordingly to improve the programmatic management of those data structures.

The general solution to the problem of visualizing data structures lies in graphs. Most data structures use a method of referencing to organize pockets of data. For example, in a linked list, each of the “pockets” is an element of the list, and each element contains a reference to the next item in the list. This creates a directional relationship between adjacent elements in a linked list. Each element is related to the next element in the list by the reference that the former holds to the latter. These references can constitute edges of a graph, and the vertices of the graph can represent pockets of data. The problem then reduces to one of properly laying out the graph in a humanly understandable form.

There are several algorithms for laying out directed and undirected graphs in general, but they are not necessarily universally appropriate for data structures. In addition, most classes of data structures have a generally accepted way of being

drawn (i.e. linked lists as a linear row of blocks and arrows); there should be a way to utilize these standards when generating drawings of data structures. This implies advance knowledge of the type of data structures in the program that is being debugged. It may be possible to determine the class of data structure automatically, or at worst it can be queried from the user without excessive trouble.

In addition to being able to design the layout of diagrams, it is important that the diagrams display as much pertinent information as possible about the elements in the data structure. When given more information, the developer can make better conclusions regarding the reliability of the program, and the additional information is also useful in determining the sources of bugs and their resulting solutions. Too much information, however, wastes space and can interfere with a programmer's ability to quickly access the information that he or she wants. A healthy balance must be maintained to provide the programmer with as much information as is needed to solve the problem, but no more or less.

The diagram should also be able to tie the data structure as a whole to the rest of the program. A diagram of a particular instance of data structure is useless unless the programmer knows what part of the program is being examined.

Current visual debugging technology is already advancing toward the goal of providing programmers with visual representations of data structures, but there are still problems that need solving. The Visual Debugger is a software project that is being developed in response to the problems that have been identified in today's visual debugging solutions. It is still in an infant state, but it is making strides toward solving several of the problems that no other visual debugger has attempted to solve.

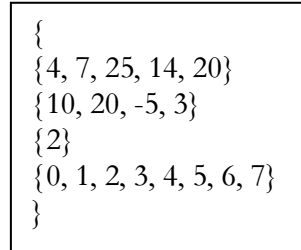
CHAPTER 2: BACKGROUND INFORMATION

2.1 Information Visualization

The field of information visualization is primarily concerned with representing data for interpretation by humans. There are usually many different ways to represent data, and some of them can easily be expressed visually (i.e. on paper or a computer screen). It is the goal of information visualization research to focus on maximizing the efficiency of visual real estate so that as much information is represented in as little space as possible, while still maintaining understandability.

The task of deciding how to organize information is a lot about psychology, and there are countless psychological studies about how much information the human brain can absorb at once and about what sorts of data organizations are better suited for human parsing. With respect to the latter, for example, most languages read from left to right and from top to bottom, so as a result, an ordered list of numbers may best be “visualized” simply as a comma-delimited output of the data values, ordered from left to right: {4, 7, 25, 14, 20}. For something as simple as a vector of numbers, this simple approach may work beautifully. If the vector were of more complex structures, however, such an approach becomes less feasible in terms of human interpretability. Suppose each vector item is itself a vector of numbers. Using the aforementioned visualization approach on such an example would result in the following cryptic representation: { {4, 7, 25, 14, 20}, {10, 20, -5, 3}, {2}, {0, 1, 2, 3, 4, 5, 6, 7} }. This set of characters certainly contains the desired information, and it can be easily parsed by a computer, and while it is interpretable by humans, it is not necessarily the most appropriate and most eye-appealing way to present a vector

of vectors. Perhaps displaying each vector on a separate line (Figure 2) would be more effective.



```
{  
{4, 7, 25, 14, 20}  
{10, 20, -5, 3}  
{2}  
{0, 1, 2, 3, 4, 5, 6, 7}  
}
```

Figure 2: A vector of vectors

Now, each individual vector of numbers in the encompassing vector of vectors is more apparent. Obviously, there are still-better ways to display a vector of vectors (for starters, the braces and commas could be replaced with something more appealing, like tab characters), and information visualization scientists strive to find the best visualization solutions for each problem domain and application.

A nearly constant companion to the problem of human interpretability is that of packing as much information into as little space without overloading human brain capacity. Some methods of visualization are more suitable than others for displaying large amounts of information. For example, consider a large matrix of integers that all fall within a finite range. For simplicity, let us consider a 10x10 matrix. The most obvious method of displaying the matrix would be to list all the values in a table (Figure 3).

0	1	1	2	4	5	3	2	1	0
1	3	4	5	5	5	4	4	3	2
1	4	5	6	7	7	6	4	3	1
1	3	5	7	8	7	7	5	3	0
0	2	4	6	7	6	5	4	2	0
2	4	6	8	9	9	8	6	4	3
4	5	7	9	9	9	8	7	6	4
5	6	8	9	8	9	8	7	5	3
5	7	8	8	7	8	8	6	4	2
3	6	7	7	5	6	7	5	4	3

Figure 3: A 10x10 matrix

The data itself is rather concise; it occupies a square only two inches on each side. At first glance, however, little can be gathered. It takes a bit of inspection to interpret the data. This is partly due to the lack of relativity between the shapes of the glyphs that represent each value. If you look at the matrix from a distance, you might be able to pick out patches of cells whose values are 7; the shape of the digit “7” is relatively unique when compared to the rest of the digits. The advantage to this tabular representation is that you know the exact values of all of the items in the matrix simply by looking at its position in the table.

In this example, neighboring values are generally close in value, and there should be a way to represent the matrix in order to take advantage of that relationship. Consider using shades of gray to represent values (Figure 4). Pure white could represent 0 (the smallest value in the matrix), and pure black could represent 9 (the largest value in the matrix), while shades of gray in between could represent the values 1 through 8, where lighter grays have smaller value than darker grays.

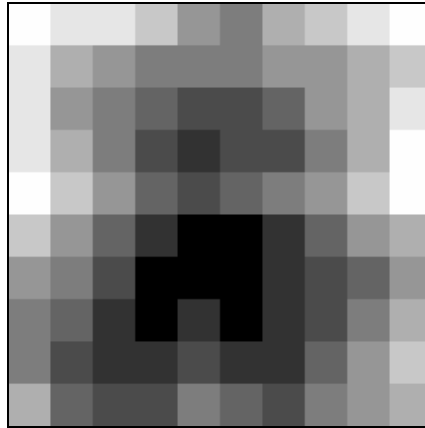


Figure 4: A grayscale 10x10 matrix

The two different representations are roughly the same size, but the grayscale version has some advantages over the simple tabular view. First, it is very easy to see relationships between particular cells; darker cells have higher values than lighter cells. It is generally more efficient for the human eye to distinguish between two shades of gray than to distinguish between two separate glyphs. A quick glance reveals that the cells generally increase in value as they approach the center of the matrix. It takes slightly more than a quick glance to notice the same characteristic in the tabular form. By that same token, patches of equivalent cells are more easily noticeable than they are in the table.

Another distinct advantage of using shades of gray instead of digits involves physical space constraints. Digit glyphs can only get so small before they start becoming unintelligible to the human eye (or unprintable by modern laser printers). Each shaded box of gray, however, can be reduced to a single pixel, so that many more cells can be displayed at once. We can use this property either to decrease the size of the visualization (Figure 5) or to increase resolution by showing more cells or a larger range of values (Figure 6).



Figure 5: A smaller grayscale 10x10 matrix

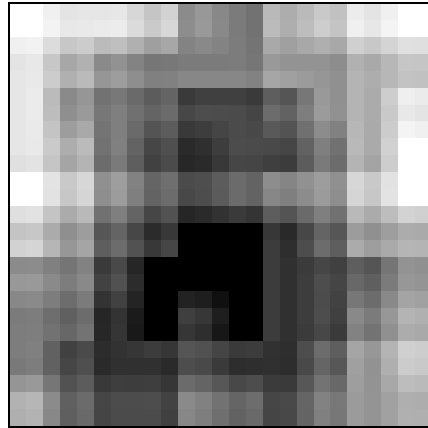


Figure 6: A higher resolution grayscale matrix

Much of the detail that was noticeable in the original 10x10 matrix is still apparent in the smaller version (Figure 5), yet it uses 99% less area of the paper than its relatively sizeable predecessors. The higher resolution version (Figure 6) is the same size as the table and the original grayscale version, but it shows a matrix with many more cells, utilizing a larger range of values.

The disadvantage to the grayscale version is that it is more difficult to ascertain the exact value of a particular cell. In fact, without a corresponding key that indicates which shades of gray relate to which values, it is impossible to discover the values of particular cells, though the presence of a key does not provide much assistance when the size of each cell becomes very small (Figure 5) or when the range of values becomes very large (Figure 6). Only relative values can be judged, and without a key, all judgments are merely assumptions (the assumption, in this case, is that lighter colors have smaller value than darker colors).

Depending on the application, even when using the same data, either the tabular view or the grayscale view may be more beneficial. If exact values of particular cells need to be accessed in as little time as possible, then the table is the best bet. Such tables are quite common in the appendices of mathematics books, where

values of functions at particular inputs can easily be looked up. If, however, being able to quickly see relative values of cells (particularly among neighboring cells) is more important, then the grayscale picture of the matrix can be interpreted much faster. An example of such a representation would be a thermal map that uses colors to illustrate current temperatures in a geographical region.

The goal of information visualization, then, is to maximize the amount of information that can be displayed in a single area (i.e. a computer monitor) while minimizing the amount of effort required by the human beings that need to interpret the data.

2.2 Graph Theory

A **graph** is a set of **vertices** (usually labeled; sometimes called **nodes**) and a set of corresponding **edges** (sometimes labeled) between the vertices [5]. A graph may be **directed** (Figure 7), meaning that its edges have direction, or a graph may be **undirected** (Figure 8), meaning that its edges have no direction.

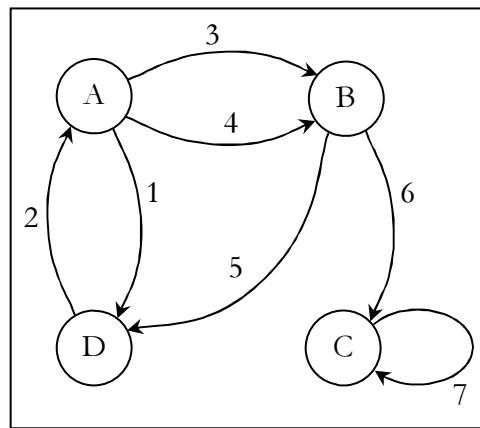


Figure 7: A simple directed graph

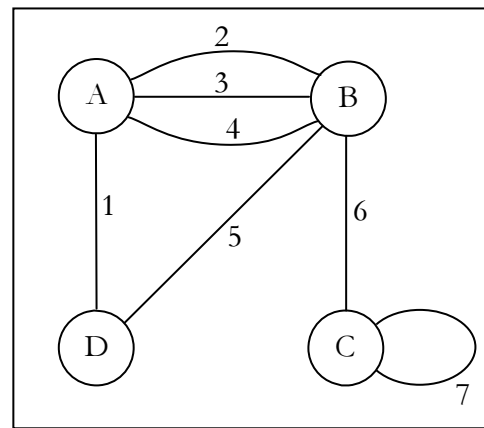


Figure 8: A simple undirected graph

Figure 7 illustrates several important characteristics of graphs (particularly, of directed graphs). In this particular graph, each vertex is represented by a circle

and labeled with a letter, and each edge is represented by an arrow and is labeled with a number. Position of the vertices does not affect the graph; that is, the X-Y coordinates of vertices drawn on a piece of paper do not matter, as long as the edges still connect the same vertices. The actual shape of the vertices and the method of labeling vertices and edges are also unimportant. Squares can be used instead of circles, or the vertices can be labeled “Lucy,” “Ricky,” “Fred,” and “Ethel.” What *is* important is the way that edges are drawn.

In a directed graph (also called a **digraph**), edges flow from one vertex to another vertex and are represented by arrows (Figure 7). There is no restriction on the number of edges that can flow from any vertex to another vertex, so some edges may flow in the same direction between the same two vertices (Figure 7: edges 3 and 4). Also, edges are allowed to flow from a vertex back to itself (Figure 7: edge 7).

In an undirected graph, edges have no direction and are drawn as lines instead of arrows (Figure 8). Again, there are no limitations on the number of edges that can connect any two vertices (Figure 8: edges 2, 3, and 4), and edges can still connect a single vertex to itself (Figure 8: edge 7). The only difference between a directed graph and an undirected graph is that the edges in the former have direction. In all other respects, the two types of graphs behave identically. In fact, any undirected graph can be appropriately represented by a directed graph with twice the number of edges by using a directed edge in each direction in place of each undirected edge. Often, digraphs where all edges are bidirectional (flow from vertex A to vertex B implies flow from vertex B to vertex A, and vice versa) are drawn in the same way as undirected graphs, with lines instead of arrows. Graphs of computer networks, for example, often fall into this category.

There are many specific types of general graphs that are important enough to warrant their own name. In computer science, the most important of those is the **tree**, which is defined as a connected graph without cycles (Figure 1).

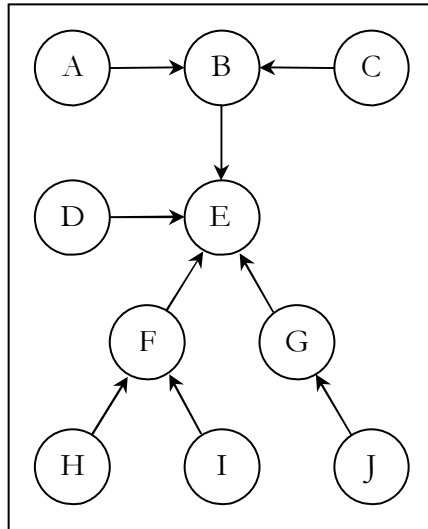


Figure 9: A simple tree

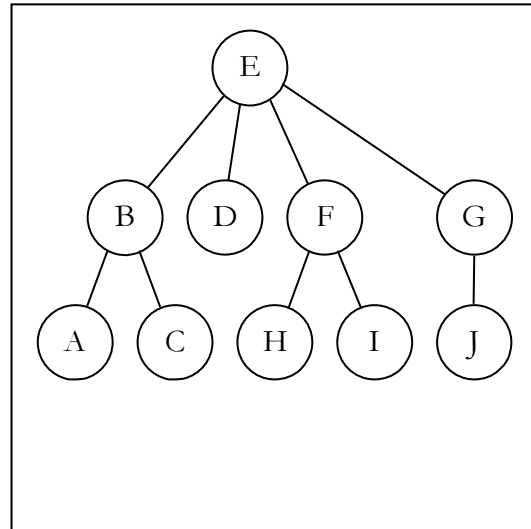


Figure 10: The same tree, but rooted

Trees can be either directed or undirected. When they are directed (and especially when the tree is **rooted**), they are very often drawn in a way such that all the edges point upward (sometimes such that they all point downward), and in this case they may be drawn without arrowheads, with the **root** (if one exists) at the top (Figure 10). In a rooted tree, such as the one in Figure 10, every vertex except for the root (Figure 10: vertex E) has exactly one outgoing edge toward its **parent**. Conversely, each vertex can have any number of incoming edges from its **children**. A vertex with no children is called a **leaf** (Figure 10: vertices A, C, D, H, I, and J).

In addition to special graphs called trees, there are special trees called ***n*-ary trees**. An *n*-ary tree is defined as either an empty tree or a rooted tree with

exactly n children, each of which is an n -ary tree. The tree in Figure 10 fits the definition of a 4-ary tree because every parent—which, by this definition, also includes the leaves—has exactly four 4-ary trees. Each leaf has exactly four empty 4-ary trees. The most common n -ary tree is the binary tree, where each parent has exactly two children.

How are graphs useful for information visualization? Graphs are extremely convenient and understandable ways to represent many real world ideas. For example, road maps can be represented as graphs, where each vertex is an intersection and each edge is a road. Graphs are also the preferred method of displaying computer networks, where each vertex is a computer, and each edge is a network connection between two computers.

Graphs, therefore, are an important method of visualizing certain kinds of data, particularly where certain things are somehow *related* to other things. In the case of computer networks, the “things” are computers, and they are “related” by being connected via hardware. Graphs are by no means limited to physical things and physical relations; they are used in all sorts of applications, such as hierarchical or relational databases, intramolecular dynamics, and even program data flow. In the context of data structures, graphs will be used to visualize the relationships between particular pieces of information.

2.3 Data Structures

A **data structure** is a collection of related data along with the operations that can be performed with the data. The key word in this definition is *related*—already, that should imply the possibility of using graphs to represent them.

Data structures are rampant in computer science. Since the advent of computers, programmers have needed simple ways to organize complex information. The

linear, binary format of memory does not often facilitate this organization. The concept of a **pointer** was invented to allow more complex relationships between pieces of data. A pointer, in the general sense, is a piece of data that provides access to another piece of data. Specific examples of pointers include memory addresses and indices into arrays.

Pointers allow pieces of data to hold information about related pieces of data (namely, how to access them). For example, in a doubly linked list, each element in the list contains pointers to the next and previous elements in the list. While lists of data can be implemented using linear memory in the form of arrays (which, incidentally, use pointers—array indices—that are external to the data), linked lists have certain advantages over arrays, particularly with respect to insertion and deletion of list elements.

Pointers also facilitate the organization of many other kinds of data structures, such as lists of lists, trees, and graphs. For this reason, pointers are very often used to organize data, so they tend to be a common denominator when it comes to analyzing data structures. In fact, the term “data structure” can be succinctly defined (excluding arrays) as a collection of data elements connected by pointers [3].

2.4 Memory Management

From the viewpoint of a program, operating systems generally divide memory into two allocatable types [1]. The **frame stack** (also abbreviated as the “stack”) holds temporary local variables for all the function calls in a program that have not yet completed. The frame stack occupies a fixed amount of space that is subject to overflow, and hence is unsuitable for storage of large data structures. The **memory heap** (also abbreviated as the “heap”), on the other hand, is a large

block of memory that is dynamically allocated as necessary; it is much more suitable for storing large data structures.

Appropriate program design calls for using the frame stack to store small, temporary pieces of data, such as integers used to iterate through arrays. Larger or dynamically sized data structures, such as large arrays or linked lists, should be allocated on the heap. Data on the heap is referenced via pointers, which can be stored on either the frame stack or the heap. Programs have direct access only to data on the frame stack. As a result, there is a strong relationship between the stack and the heap. Data on the heap must somehow be referenced by at least one piece of data on the frame stack. Any data that is not referenced by a pointer on the frame stack becomes unreachable to the program, so the memory that the data occupies becomes useless, since the operating system still considers the memory as allocated. The result is a **memory leak**.

CHAPTER 3: STATE OF THE ART IN DATA STRUCTURE VISUALIZATION

3.1 Graphing Algorithms

Graph theory is a very important field of study in which countless man-hours of research efforts have been spent. Graphs are incredibly useful constructs, and since they are most understandable as visual entities, they are most commonly seen in a visual format. Some graphs have the potential of becoming very large in size, and the desire to draw such graphs has motivated research into possible ways to generalize the process of drawing graphs, with hope that such a generalization could be realized in an algorithm that could be implemented on a computer, automating the actual process of drawing graphs.

Many great strides have been made toward such an end, and some of the more important classes of algorithms for drawing graphs are worth detailed mention. The circular layout [6] method is perhaps the simplest, and it can be applied to both directed and undirected graphs. The layering method [7], however, focuses more on directed graphs, while the physical modeling class of algorithms [7] works best with undirected graphs.

3.1.1 Circular Layout

The simplest algorithm places all of the vertices on the perimeter of a circle, where each vertex is equally distant from its neighbors, and then connects vertices with lines or arrows as appropriate (Figure 11). This algorithm's strengths are in its simplicity, its speed, and that it can be used for both directed and undirected graphs. Its biggest weakness is that the resulting graphs can

quickly become a spaghetti bowl of nonsense as the number of edges or the number of vertices increases.

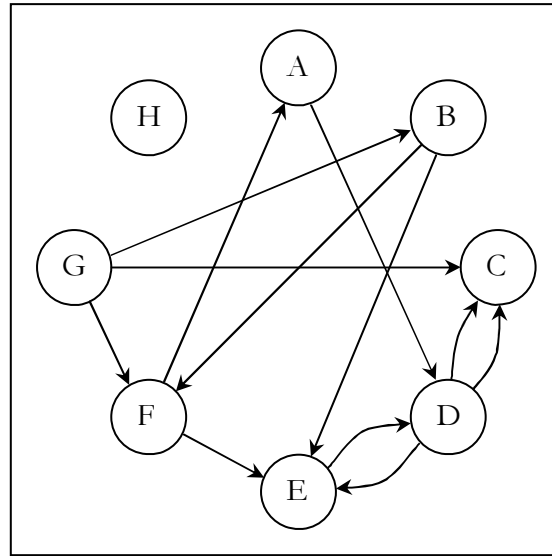


Figure 11: A circular directed graph layout

3.1.2 The Layering Method

A more sophisticated algorithm for drawing directed graphs is called the layering method. The layering method assigns vertices to layers in such a way that as many edges point upward as possible. Edges are only allowed to connect vertices at adjacent layers, so connecting two vertices that are further than one layer apart requires creation of one or more “dummy” vertices through which the algorithm connects the two actual vertices. Figure 12 shows one possible result of applying the layering method to the graph from Figure 11. The gray, unlabeled circles represent dummy vertices, and there is only one edge that points downward (the edge from E to D).

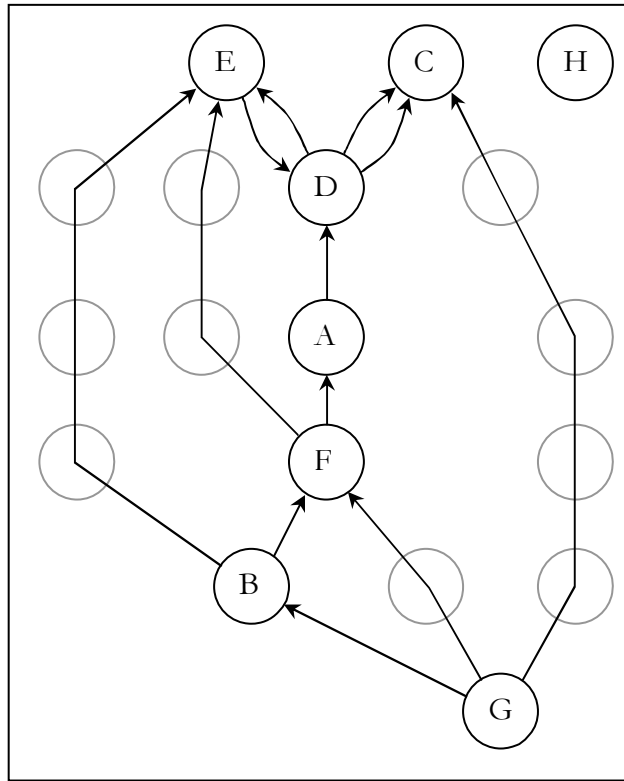


Figure 12: A layered directed graph layout

The layering method is very popular for directed graphs because it creates reasonably aesthetic graphs in an acceptable amount of computer processing time. It can realistically handle graphs with up to a thousands vertices and edges. Problems arise, however, when trying to draw graphs with more than a few cycles. The layering method is rather hierarchical by its very nature, so the algorithm performs quite well on graphs that conform well to a hierarchy. As the number of cycles increases, however, the mostly-upward characteristic of the output becomes a hindrance to aestheticism. This reliance on hierarchy also makes the layering method a poor choice for undirected graphs.

Another complaint against the layering method is its use of bending edges. It is generally agreed that the most aesthetically pleasing drawings of graphs are the ones that rely only on straight edges [7]. Any bending or curving can mislead an

interpretation of the graph, so the layering method's dummy vertices create a problem in that respect.

3.1.3 Physical Modeling

Finally, another class of algorithms seeks to solve the problem of drawing undirected graphs. It uses physics to model the vertices as masses that repel against each other, and the edges are like springs that cause adjacent vertices to be attracted to each other. The repulsive forces of the vertices serve to maintain an appropriate distance between the vertices, so that they are not all bunched together in a pile. The springs bring connected vertices closer together. Repeated optimization results in a relatively aesthetic graph whose vertices try to remain as close as possible to their adjacent neighbors and as far as possible from vertices that are graph-theoretically further.

The advantage of this physical approach is that, eventually, the vertices seem to just place themselves according to the laws of physics into a state of least energy, much in the way that most of nature works. It is very effective and can produce aesthetically appealing representations of some of the most complex graphs. The downside of this approach is that it ignores crossing edges. Edges that cross can be a bothersome property in a drawing of a graph, since it can cause confusion about which vertices an edge actually connects, so the optimal solution would avoid intersecting edges at all costs.

3.2 Data Structure Visualization Software

It is widely admitted that one of the most time-consuming parts of programming is finding bugs and fixing them. This realization has led to practically unlimited interest in reducing the possibility of errors and in expediting the process of debugging. Tools such as those included in Microsoft's Visual Studio suite of

software development environments are a result of this ongoing effort. One thing that most debuggers lack, however, is visualization.

There are, however, many software projects that were or still are being developed with data visualization in mind. A very promising project called Deet [2] uses a common graph-drawing tool called Dotty [4] to provide visual representations of data structures. VDBX [4] is another Dotty-powered visual debugging project. A larger project called the GNU Visual Debugger has a version that compiles on three different platforms. By far, however, the Data Display Debugger (better known as DDD) [9] is the most well-known and largest of the visual debuggers, and has been under active development since 1993.

3.2.1 Dotty

Dotty [4] is a general-purpose graph utility. It can be used to edit and draw general directed or undirected graphs. Though it can run standalone as a graph editor, it is more important to the art of visual debugging as a front-end tool for visual debugging tools such as Deet and VDBX.

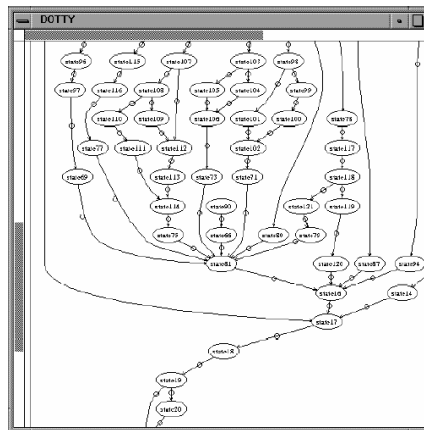


Figure 13: A graph displayed with Dot
(taken, unedited, from [4])

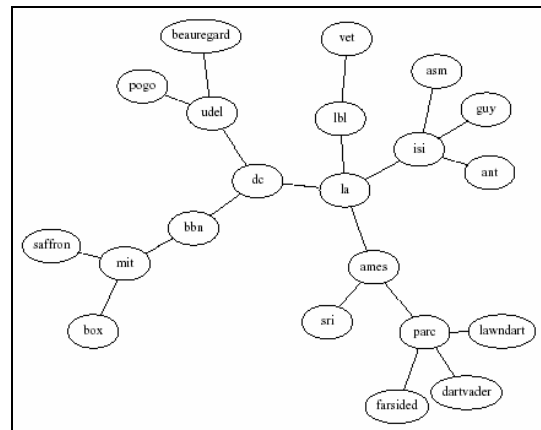


Figure 14: A graph displayed with Neato
(taken, unedited, from [4])

Dotty is partitioned into two sections: a programmable viewer, called Lefty; and graph layout generators, implemented with Dot (Figure 13) and Neato (Figure 14). Dot generates layouts for directed graphs and is based on the layering method described earlier in this chapter. Neato generates layouts for undirected graphs and uses physical modeling as described earlier in this chapter. Together, Dotty provides advanced capabilities to programmers that would otherwise be forced to implement them manually. By using Dotty, visual debugging software need not worry about proper display of the graphs they construct.

3.2.2 *Deet*

Deet [2] stands for Desktop Error Elimination Tool, and its name is a good indicator of its goal. It attempts to address the shortcomings of large, complex debuggers, so it is a very small shell program that does a surprisingly large amount of work in only 1500 lines of shell. Written in tksh, a variant of the Korn shell, it is machine-independent, but it relies on the concept of a “nub,” a small set of machine-dependent functions that must be embedded in the debugged program to allow Deet to collect the necessary data from the program that is being debugged. Thus, Deet requires a slight bit of extra effort on the part of the programmer in order to provide him or her with debugging capabilities.

Figure 15 shows a sample debugging session that uses Deet. The graphical representation of a data structure appears in the lower right corner, in a window labeled “Dotty.”

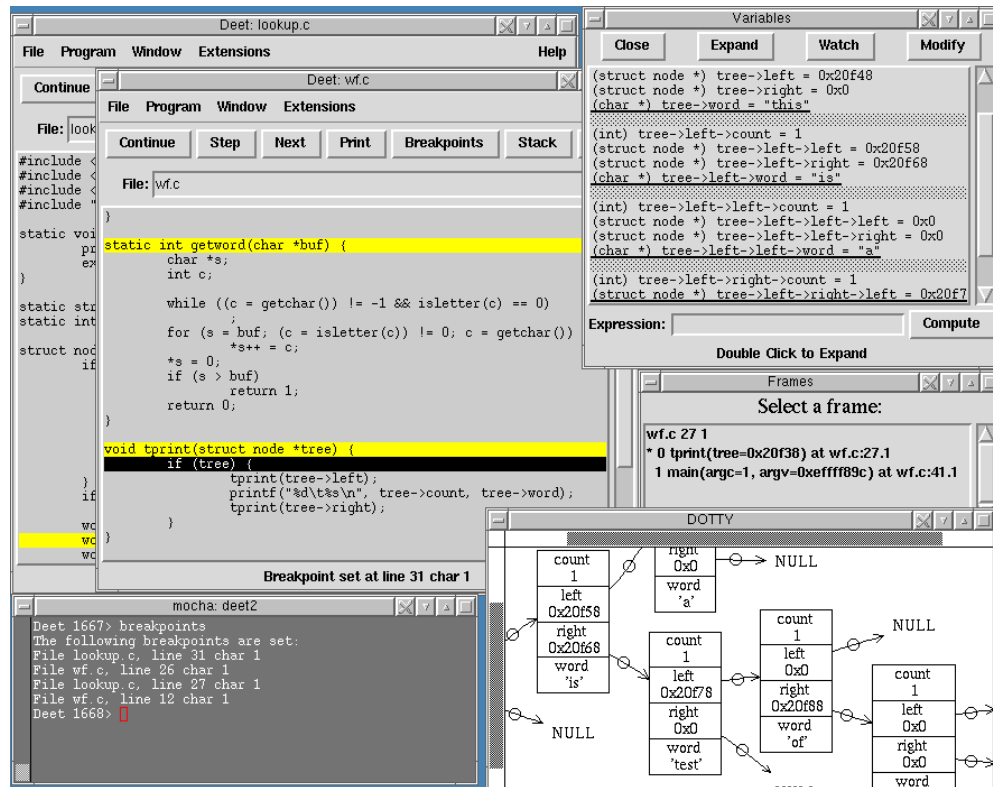


Figure 15: Deet
(taken, unedited, from [2])

Features of Deet include capability for browsing source files, setting breakpoints and watch variables, and visualizing data structures via Dotty, a separate program that relieves Deet of the complex responsibility of drawing graphs. Deet's creators found problems in larger debuggers such as GDB—which is about 150,000 lines of C code. Large programs are inherently buggy themselves, and Deet's conciseness avoids that and other problems that go along with size and complexity. Large debuggers are also not very extendable, and one of Deet's most attractive features is its extensibility. Because it is so small, it is easier to understand, making it easier to modify and extend.

3.2.3 VDBX

VDBX [4] is another small visual debugging project that delegates to Dotty the complex task of drawing graphs in an aesthetically pleasing organization. Unlike Deet, however, VDBX is unintrusive in that it adds nothing to the program being debugged. Instead, it uses DBX, an existing debugging tool, to gather the required information from the debugged program.

Figure 16 shows VDBX in action. A visual representation of a data structure (generated by Dotty) dominates the upper half of the screen.

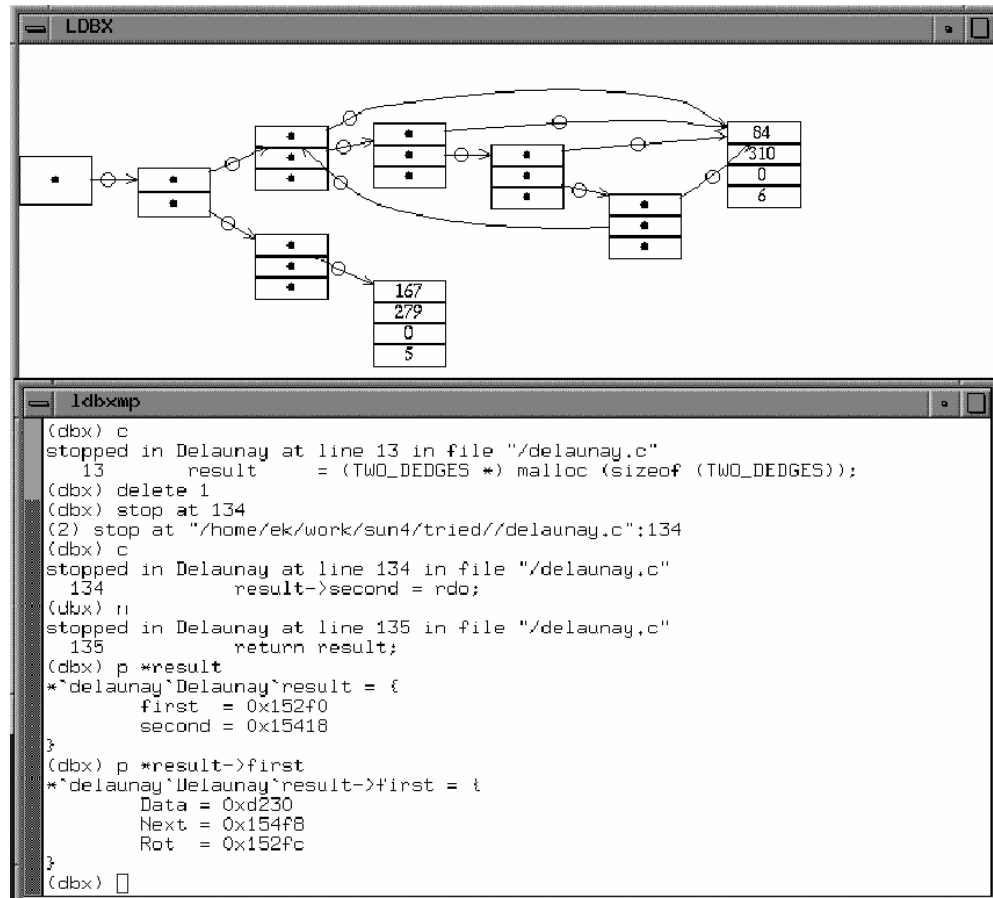


Figure 16: VDBX
(taken, unedited, from [4])

Because it uses an external debugger to provide debugging capabilities, it is tied to that debugger in both positive and negative ways. It inherits all the features of DBX, giving it more capabilities than Deet's smaller nub, but with it comes the limitations of DBX. Developers that wish to use VDBX must first be familiar with DBX in order to use its features. VDBX would be more user-friendly if the developer could use his or her own favorite debugger. Because the developers of VDBX saved themselves the effort of designing their own debugger, however, they were able to focus more on the visualization of data structures, centering more on the use of Dotty to display data structures graphically, contrary to Deet, where Dotty plays a role less significant than Deet's other features.

3.2.4 GNU Visual Debugger

The GNU Visual Debugger (GVD) [10] is an ongoing visual debugging software project, developed by ACT-Europe. It is very robust, implementing many useful debugging features. Like VDBX, it uses a text-based debugger to carry out the actual debugging necessities, and it uses the output to display it on-screen in a user-friendly format. Unlike VDBX, which restricts its users to DBX or debuggers that use the DBX syntax for C structs, it can be configured to use nearly any debugger that the programmer wishes to use. It has native support for GDB, VxWorks, LynxOS, JVM, and others, but if a favorite debugger is not supported natively, GVD can easily be extended to use that particular debugger. GVD even supports using multiple debuggers simultaneously.

Figure 17 shows the main window of the GNU Visual Debugger. The "Canvas" at the top of the screen is of most concern. It displays data nodes as boxes that lists all the fields of the data node, and arrows connect nodes that are related to each other via pointers.

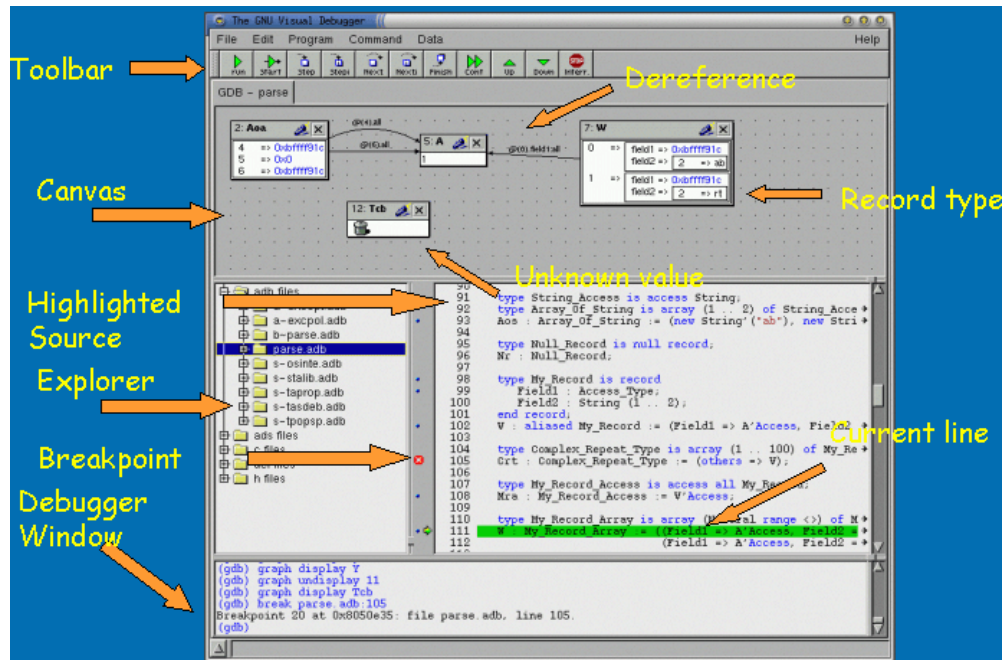


Figure 17: The GNU Visual Debugger
(taken, unedited, from [10])

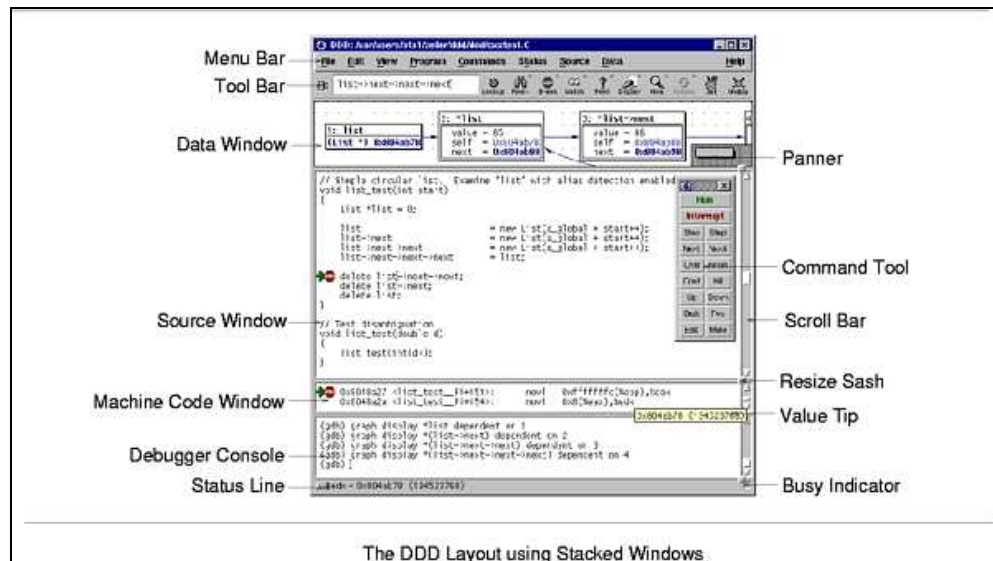
In addition to flexibility with debuggers, it is written in Ada using a highly platform-independent GUI toolkit called GtkAda, so it can be compiled on virtually any modern operating system, including Microsoft Windows, GNU/Linux, and Solaris. As long as the system has a text-based debugger that GVD can use, then the system can use it as a powerful visual debugger.

Among its numerous features, it sports the capability to graphically depict data structures, though this feature is downplayed somewhat in comparison to its high flexibility with respect to debuggers and platform-independence. Still, its visualization of data structures is sufficient enough for simple debugging purposes.

3.2.5 Data Display Debugger

The Data Display Debugger (DDD) [9] is a long-established visual debugging program that serves as a graphical front end to the GNU Debugger (GDB). It was created as a free alternative to commercial debuggers that were bundled with proprietary integrated development environments and compilers.

The project started in 1993 as a simple front end to GDB. It inherited GDB's advanced debugging capabilities and added graphical representation of data structures, much in the same way that GVD does, with boxes representing nodes and arrows representing pointers. Since then, it has evolved to include hundreds of new features, including support for many more command-line debuggers, including DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, and the Python debugger. In addition to standard graph representations of data, it also includes support for multidimensional plots of data, a novel approach to visual debugging.



The DDD Layout using Stacked Windows
Figure 18: The Data Display Debugger
(taken, unedited, from [8])

Figure 18 shows a Data Display Debugger session. Like GVD, its “Data Window” is at the top of the window. DDD takes the graph one step further, however, by providing a method of laying out a Data Window. Figure 19 shows a type of tree data structure after applying one of DDD’s automatic layout algorithms.

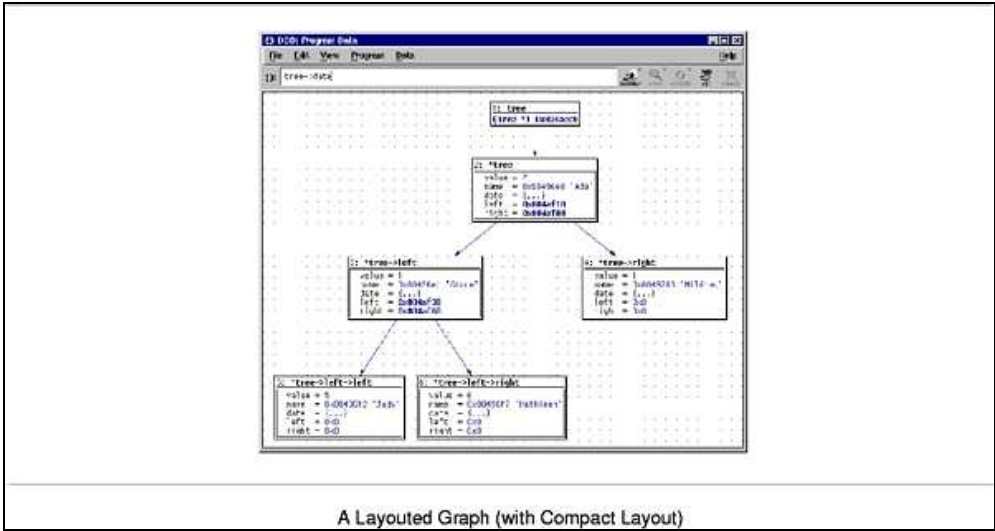


Figure 19: A tree, using DDD’s automatic layout algorithm (taken, unedited, from [8])

CHAPTER 4: THE DATA STRUCTURE VISUALIZATION PROBLEM

The problem is to provide (for purposes of debugging a program) an automatic solution for generating an aesthetic and understandable graphical representation of data structures as they are developing in a running program. The main purpose of the solution should be to debug. Therefore, the targeted data structures will be dynamic, changing constantly in the debugged program as it runs. Also, generic graph-drawing algorithms may suffice for some types of data structures, but in general, computer scientists often visualize certain types of data structures in certain ways.

The optimal solution will strive simply to be the most useful and the most helpful visual debugger, providing the most effective and informative data structure visualizations. It need not concern itself with graph-theoretic or data organizational problems that are of no concern to the debugging programmer. The needs of all developers, however, can be difficult to judge ahead of time.

There are three usage scenarios that professors may encounter during the course of teaching their students about data structures and their implementations. Firstly, students may wish to design and build an implementation of a data structure and see the results of their code. A good visual debugger will provide exactly this capability: visual display of user-defined data structures. Secondly, students often make mistakes with dynamic memory allocation, and a visual debugger should provide feedback with respect to attempting to dereference and/or modify the contents of a pointer's memory before it is actually allocated. Finally, visual debuggers may be able to provide greater feedback about recursion, specifically with information about the program stack and recursive history.

Of the debuggers discussed in Chapter 3, many of them provide visual capabilities that cause them to stand out from other, non-visual debuggers. Each of them fails, however, in several ways.

Deet provides its own debugging facilities without reliance on another debugger (such as GDB or DBX), but at a cost. It requires that the user embed the appropriate nub into the debugged program. This is added effort that the other debuggers do not demand. Deet's strengths are in its small size and its extensibility. Deet's simple nub, however, does not support many of the capabilities inherent in other debuggers [2], such as examining core dumps, evaluating expressions, and assembly-level debugging. Also, due to its simplicity and its reliance on Dotty, it does not provide much flexibility in its data structure visualization. It uses the same algorithm to draw all data structures, and while it is very effective at drawing trees and linked lists, it fails somewhat in conveying the important characteristics of other, more complex data structures like general graphs.

VDBX suffers from the same ailments that are associated with using Dotty to draw its data structures: the inability to adapt graphs to the types of data structures that they are to represent. It draws trees, linked lists, and general graphs in the same way, no matter how inappropriate.

Another problem common to Dotty is that of stability in incremental layouts [4]. Dotty cannot perform simple additions to its graphs; it draws graphs from scratch each time. Graphs that are slightly different may be drawn by Dotty in quite a different manner. In a debugging atmosphere, where data structures are constantly and gradually changing, the most effective graph-drawing algorithm will be stable with respect to slight changes in the data. Unfortunately, Dotty's algorithm does not support this, and both Deet and VDBX suffer as a result.

GVD avoids this problem in a sub-optimal way: it does not employ an algorithm to layout graphs. It places vertices on the drawing area and expects the user to organize the vertices in the way they see fit. The obvious disadvantage is that it requires more of the programmer than if the graph were drawn automatically. The advantages include speed and stability. Not having to worry about layouts means that GVD does not have to spend time figuring out how to appropriately layout the graph. It also means that the graph is not redrawn at every interval. This is a very important property that provides both speed and stability, the latter of which was the biggest problem with Dotty. Still, providing automatic layout would greatly alleviate the duties that the developer would otherwise perform, and less work results in more time to debug and improve a software project.

DDD also implements its own graph-drawing algorithm, but only as specified by the user [8]. The default parallels GVD by simply placing data on the drawing area as needed. Upon request, however, DDD will layout the data structure with one of two algorithms. The default algorithm uses a tree format, while the second algorithm draws graphs in a more compact form. Like Dotty, however, the algorithms do not adapt to the type of data structure, so the layouts are not always meaningful.

There are two other problems that are common to all of these visual debuggers. The first is that none of them provide a means for the programmer to select which fields of a structure should be displayed. Every one of them displays all of the fields, no matter how insignificant any of them may be. This is a disappointment to many programmers, because in practice, most data structures include many fields, some of which are unimportant for all debugging tasks, and some of which are at least unimportant for *some* debugging tasks. None of the current visual debuggers provides for the flexibility to hide fields as desired by the user. DDD offers the ability to compress fields that are themselves data

structures, where they can be expanded as desired by the user, but the field still uses space, no matter how useless it may be to the programmer. The result is an extremely large waste of space in visualizations of data structures that contain many fields.

The second problem that is common to the discussed debuggers is their lack of respect for the relationship between the frame stack and the memory heap. Most debuggers keep the two totally separate, drawing no clear connections between the two. DDD and GDB do perhaps the best job of connecting the two by drawing stack variables on the drawing area as if they were in the same category as heap data. Their solution, however, fails to emphasize the difference between stack and heap data. While it is true that variables are stored in the same way on both the stack and the heap, clarifying the separateness of stack and heap data may assist programmers in finding and solving bugs. All of today's debuggers, then, fail by either separating too much the stack from the heap or by disregarding that separation entirely. The ideal solution would both separate the stack and the heap and illustrate the necessary connections between the two.

Analysis of the state of the art of visual debugging results in no clear-cut solution to the problem of visualizing data structures. The problem may remain unsolved in practice, but in the very least, new standards have been revealed that may help to lead to the optimal visual debugging tool.

CHAPTER 5: THE DATA STRUCTURE VISUALIZATION SOLUTION

5.1 Guidelines

The task of properly visualizing data structures is not a simple one. With it are difficulties in quantifying the aestheticism of a particular drawing of a data structure. The discussion of the state of the art of visual debugging, however, leads to several guidelines that can be taken into account when analyzing how much more or less effective one drawing is when compared to another.

1. It is important that the visual focus of the debugger not interfere with the standard qualities that have come to be expected from a debugger. That is, a visual debugger should at least allow standard debugging tasks (i.e. setting breakpoints, watching values, etc.) if it does not explicitly implement them. For example, both GVD and DDD use an external tool to provide standard debugging tasks, extending these tasks in a graphical manner.
2. A visual debugger should strive to be as unintrusive to the developer's program as possible. In the ideal case, the only thing a programmer needs to do is start the debugging program. Any additional work, such as linking a nub or compiling additional code, should be avoided.
3. It seems most appropriate to use graphs to represent data structures, since there are very fine definitions in this problem domain for vertices (pockets of data, which can be thought of as structures) and edges (pointers from one pocket of data to another pocket of data). Therefore, accurate graph construction is a must.

4. A correct graph representation of data, however, is not enough. The graph must be presented to the user in an understandable and aesthetic form. Since data structure graphs are inherently general directed graphs, an algorithm designed toward drawing these kinds of graphs will most likely be the most appropriate solution. Graph representations of many data structures will have cycles in them; a doubly-linked list, for example, is full of them. A satisfactory solution, therefore, should take cycles into account when drawing graphs.
5. While understandable and aesthetic drawings of graphs are useful in debugging, there are many different ways to draw aesthetic graphs. Some graph layouts are more appropriate than others. For example, a linked list is best drawn in a linear form, emphasizing the linearity of the list. Rooted trees, on the other hand, are most effectively drawn with the root vertex on one side of the graph, with all arrows drawn in the same general direction. The most common format for drawing rooted trees places the root vertex at the top and draws all arrows in the upward direction. The graph should be drawn in the most effective form as is appropriate for the data structure that it represents. This may require asking users for more details about their data structures, but the best debugger would be able to automatically discover the types of the data structures.
6. It is important that a visual debugger provide information about specific values of the data. For example, vertices may represent structures in which there are several fields. Any of those fields may be relevant to the programmer; on the other hand, any of those fields may be useless to the programmer, wasting precious screen real estate. The optimal visual debugger would allow the programmer to customize which fields of any particular data structure should be displayed.

7. Finally, most programming languages and operating systems distinguish between memory allocation to the frame stack and to the heap. For example, in the C programming language, memory is allocated to the heap by calling `malloc()`. Since all memory on the heap should be rooted via a pointer to somewhere in the stack (otherwise, there would be no way for the program to access the heap-allocated memory), there is a clear and distinct relationship between pieces of data on the stack (pointers) and pieces of data on the heap (the data structures). This relationship should be intuitively displayed to the developer, and the ideal visual debugger will provide such a display.

None of these guidelines are strict requirements for a visual debugger, but they help make a visual debugger more useful to its user base of programmers. Many of today's current visual debugging software solutions conform to several of the guidelines, but none of them properly adheres to all of them. The optimal visual debugging solution will do that and probably more.

5.2 Current Progress

The software that is currently being developed as a response to the findings earlier in this chapter is called the Visual Debugger. More information about it can be found at <http://infovis.cs.vt.edu/datastruct/>, the official Visual Debugger website. As of March 2002, it is in a very infant state, operating more as a working prototype than a true software solution, and it implements very few of the qualities that have been deemed necessary for an optimal visual debugger, but it is on its way and hopefully will one day realize optimality as defined earlier in this chapter.

Currently, the Visual Debugger is written mostly in Microsoft Visual Basic and inherits its debugging capabilities from the Microsoft Visual C++ Debugger. As such, it requires that the developer use the Microsoft Visual C++ Integrated

Development Environment, at least for debugging. This requirement provides the standard debugging capability of the Visual C++ Debugger, but—like Deet or VDBX—imposes the limitations of supporting a single debugger. Ideally, the Visual Debugger should either implement its own debugging facilities, or it should be able to support multiple debuggers.

Despite being built on top of the Visual C++ Debugger, there is a small bit of C++ code that must be included in the target program in order for the Visual Debugger to track data allocated on the heap. Ideally, the Visual Debugger should be non-intrusive.

The current capabilities of the Visual Debugger shall be demonstrated through a sample run-through with a very simple binary search tree project (see Appendix A for the code).

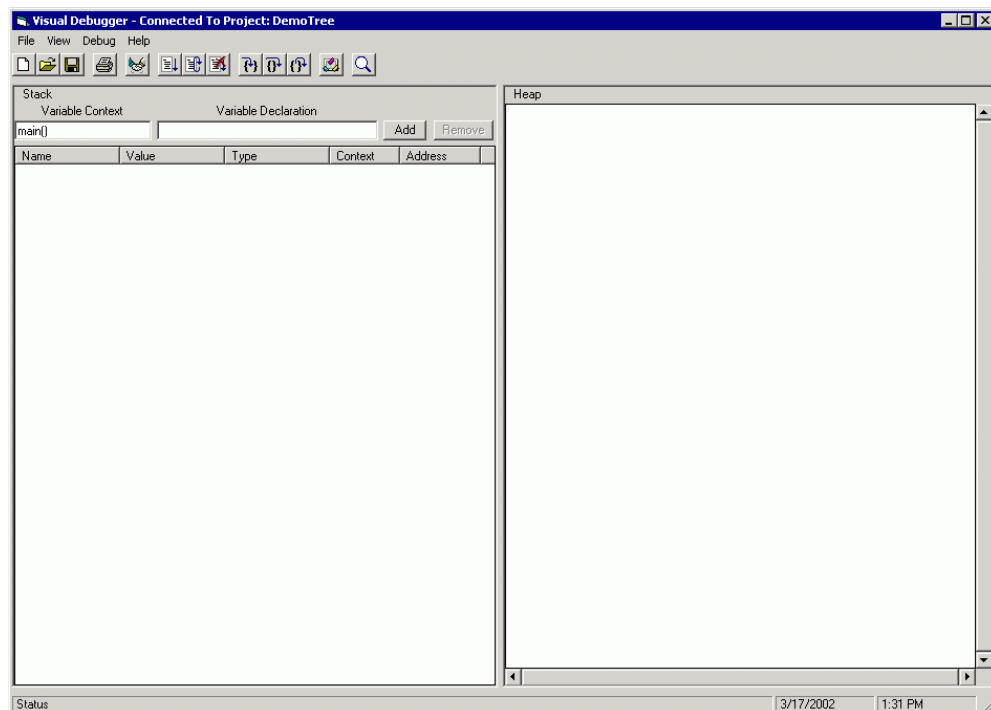


Figure 20: The Visual Debugger Initial Screen

Figure 20 shows a screenshot of the first screen that appears upon program execution. If the Visual C++ project that needs debugging is already open, then the title bar will indicate that the Visual Debugger has connected to the Visual C++ project. Otherwise, the user can simply connect after opening the project in Visual C++. In this case, the user has already opened the DemoTree project, and the Visual Debugger indicates that it has connected to that project.

During debugging, the left half of the main window will contain information about data on the stack, while the right half of the main window will contain information about data on the heap. Currently, the programmer must manually specify which stack variables he or she would like to watch, as well as the specifics of the user-defined data types in the program. The latter requirement, though a nuisance to the programmer, has the desired advantage of being able to specify only those fields that are of importance to him or her.

There is only one stack variable in this project that absolutely needs to be defined in order to be able to watch anything useful: `tree`, which is a pointer to an instance of type `Tree`. Figure 21 shows that the user can add a stack variable to be watched using standard C declaration syntax. Tabs and semi-colons can be included as desired, so the user can simply copy and paste the declaration line.

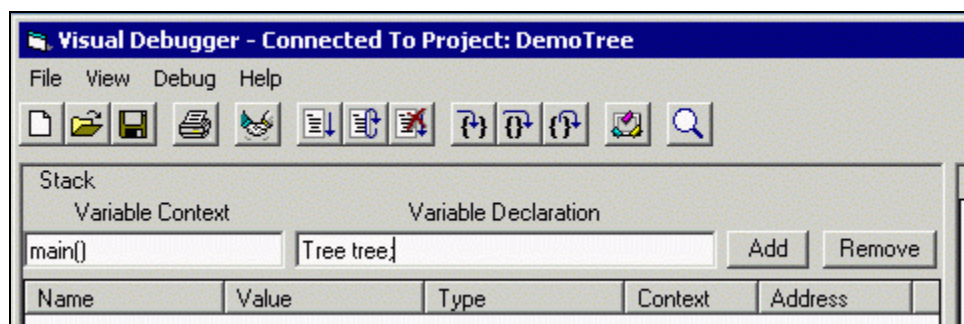


Figure 21: Adding `tree` it to the stack list

When the user presses the Enter key or pushes the Add button, the Visual Debugger realizes that the data type `Tree` has not yet been defined, so it brings up the “Define Class” dialog box (Figure 22). The user enters the fields that he or she wishes to watch, disregarding the extraneous fields that he or she does not want to watch. There are three fields in the `Tree` class (`Node* root`, `int treeCount`, and `int nodeCount`), but for the sake of demonstrating the ability to ignore certain fields, the user decides that the number of trees is unimportant for this debugging session (it can be added at any time, should the user so desire). Figure 23 shows the “Define Class” dialog box after having added `nodeCount` and just before the user adds `root`.

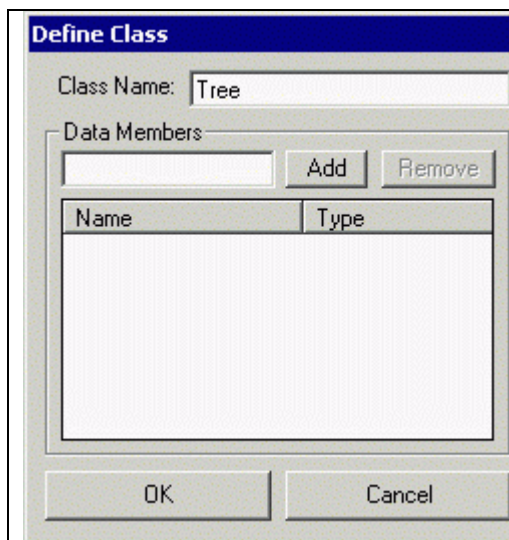


Figure 22: The “Define Class” dialog box

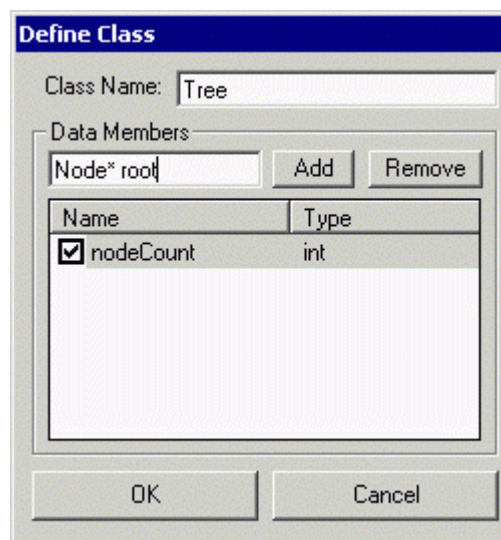


Figure 23: About to enter `Node* root` as a field

The `Node` class has not yet been defined, so another “Define Class” dialog box is brought up to allow the user to define it (Figure 24). There are three fields in the `Node` class (`int value`, `Node* left`, and `Node* right`), and all of them are of relevance to the user, so he or she adds them all to the definition of `Node`.

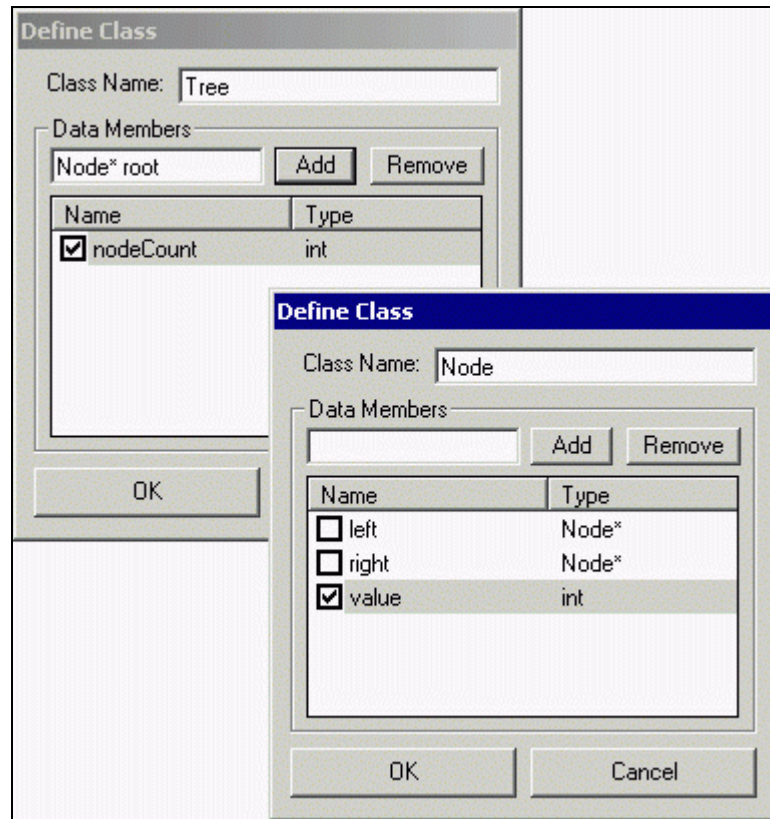


Figure 24: The completed definition of Node

In the Visual Debugger, there are two visual modes that the user can dynamically switch between. The first is “Zoomed In” mode. In this mode, all of the defined fields—in addition to the address in memory—for every instance of every class are displayed in the drawing area. Because the programmer can select which fields are important enough to display, this is often the display mode of choice. Should the display begin to get excessively crowded, however, the Visual Debugger offers an alternative mode, called “Zoomed Out” mode. In this mode, only one field from each instance is displayed. This field is chosen by the user in the “Define Class” dialog box, and can be changed at any time. The user selects this important field by putting a check next to it in the list of fields. The user can select no field to indicate that the object’s address in memory be displayed in “Zoomed Out” mode. In this example, the user has selected value as the most

significant field in the Node class (Figure 24). Therefore, when the user chooses “Zoomed Out” mode, every instance of Node shall be denoted by that field, value.

After closing the “Define Class” dialog associated with the Node class, the user resumes defining the Tree class. For the Tree class, the user has specified that nodeCount be the field displayed in “Zoomed Out” mode (Figure 25).

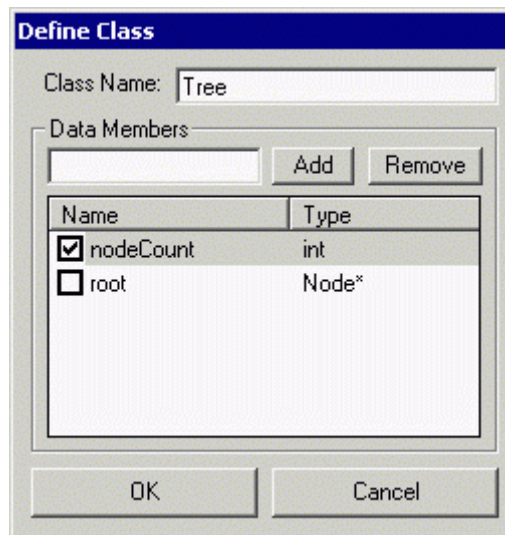


Figure 25: The completed definition of Tree

After confirming the definition of Tree by pressing the OK button, the user has completed defining the necessary classes for this simple software project (Figure 26). The in-depth explanation might imply that class definition, even for this trivial example, is a long and arduous process; after a little experience, however, the process is very quick (about one minute for this example). To prevent this step in future DemoTree debugging sessions, however, the user may choose to save the current workspace definitions to a file. All class definitions and stack variables are saved so that they can be retrieved later.

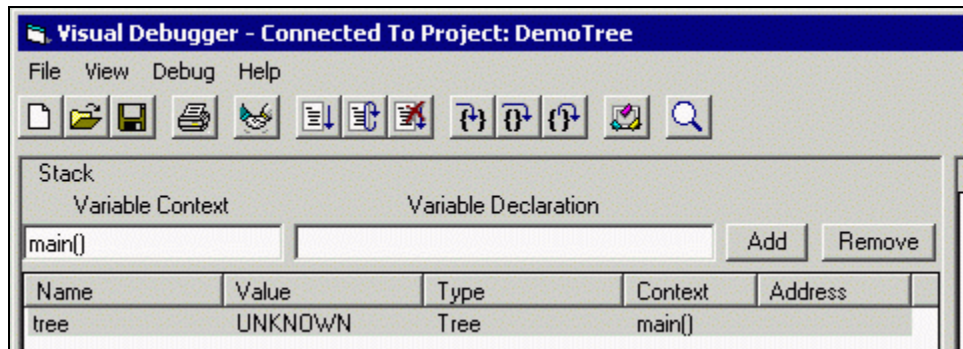


Figure 26: After adding `tree` to the stack list

The user can modify any of the defined classes via the “Classes” dialog box (Figure 27). All of the currently defined classes are listed, and selecting any of them produces the same “Define Class” dialog that was used for defining unknown stack variable classes. The user may also add new or remove old classes with the “Classes” dialog box.

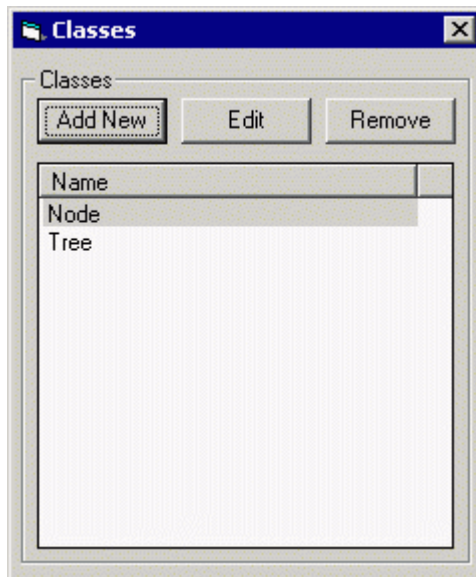


Figure 27: The “Classes” dialog box

When the user is ready to start debugging, he or she can run the program in debug mode by using the Debug menu or the toolbar, in the same way that it is done in Visual C++. Users may set up breakpoints in Visual C++ to stop

execution at desired intervals, or they may step through their program line by line. In either case, the Visual Debugger's display is updated at each stop point.

As elements are allocated onto the heap, they appear as nodes in the heap window (Figure 28). Newly allocated nodes are highlighted in green. An arrow leads from `tree` to the node to which it points. A blue arrow means that the arrow points from a pointer on the stack. Instances of type `Tree` initialize as an empty tree, therefore the `root` field of `tree` is null.

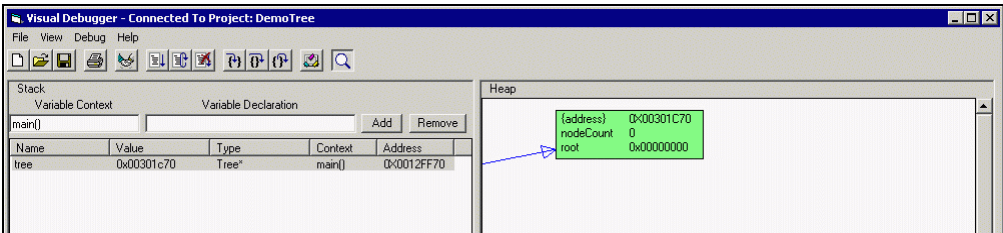


Figure 28: The `tree` variable has been initialized

Figure 29 shows the Visual Debugger's depiction of the tree after the first insert. By default, new nodes are placed next to their parents, but the user is free to move nodes to any location they wish. Scrollbars are provided to reach non-visible areas of the drawing canvas. The size of the canvas is, for all intents and purposes, unlimited.

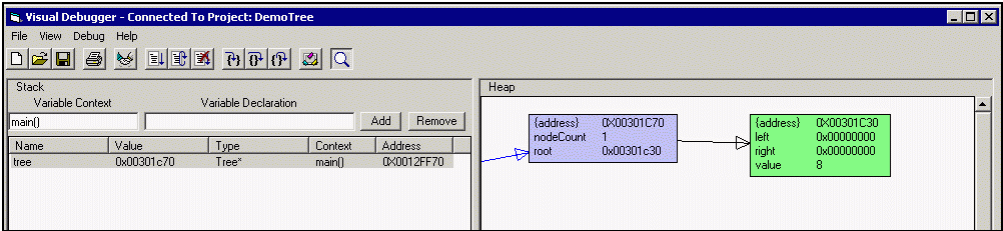


Figure 29: The tree now has one node.

The new node is green to indicate that it is a newly allocated node. The random number generator, which feeds the binary search tree values between 0 and 99, has selected 8 for the first node. Nodes that have changed since the last stopping

point are highlighted in purple. The tree object is purple because both nodeCount and root have changed with the addition of the first number in the tree.

Figure 30 shows the Visual Debugger's depiction of the tree after all eight of the random numbers have been inserted into the tree, after some organizing by the user. The node with the value 90 was the last to be inserted, so it is green. The node with the value 98 had its left pointer changed from null, and the tree object's nodeCount increased from 7 to 8, so they are both purple.

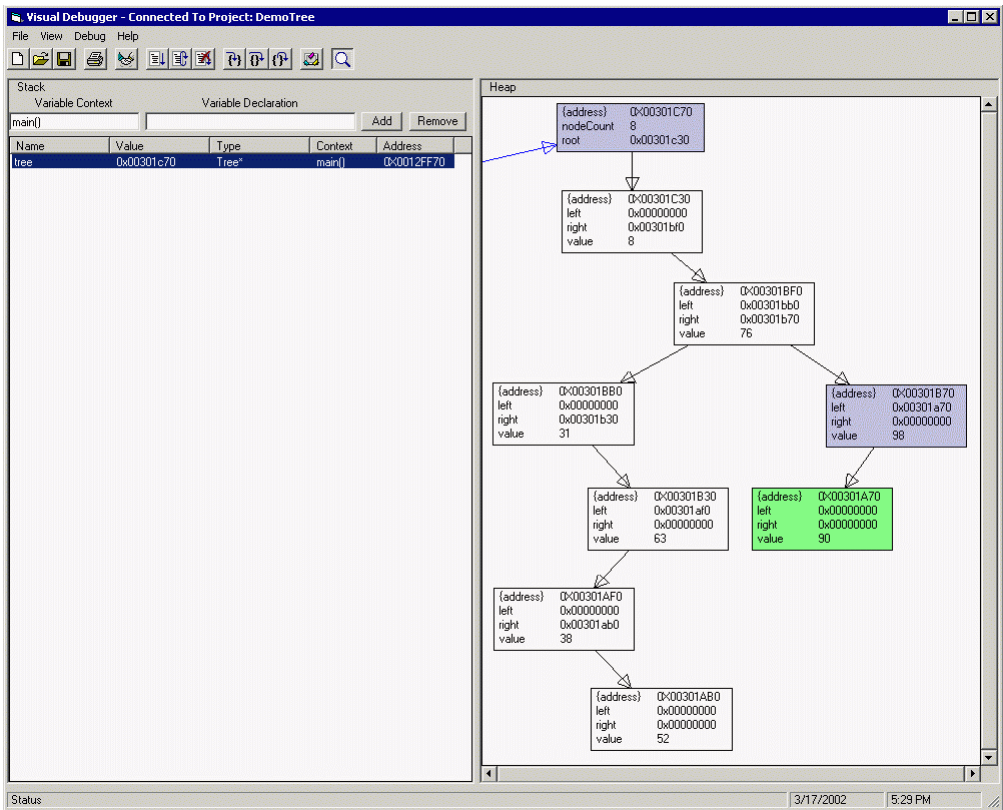


Figure 30: The tree with eight nodes

A user can select a node by clicking on it (Figure 31). When a node is selected, it is highlighted yellow so that the user knows which node has been selected. All nodes that either point to the selected node or are pointed to by the selected node

memory, but it still persists in the Visual Debugger as a “dead” node, highlighted red. Both its parent pointers and the tree object are purple due to the deletion.

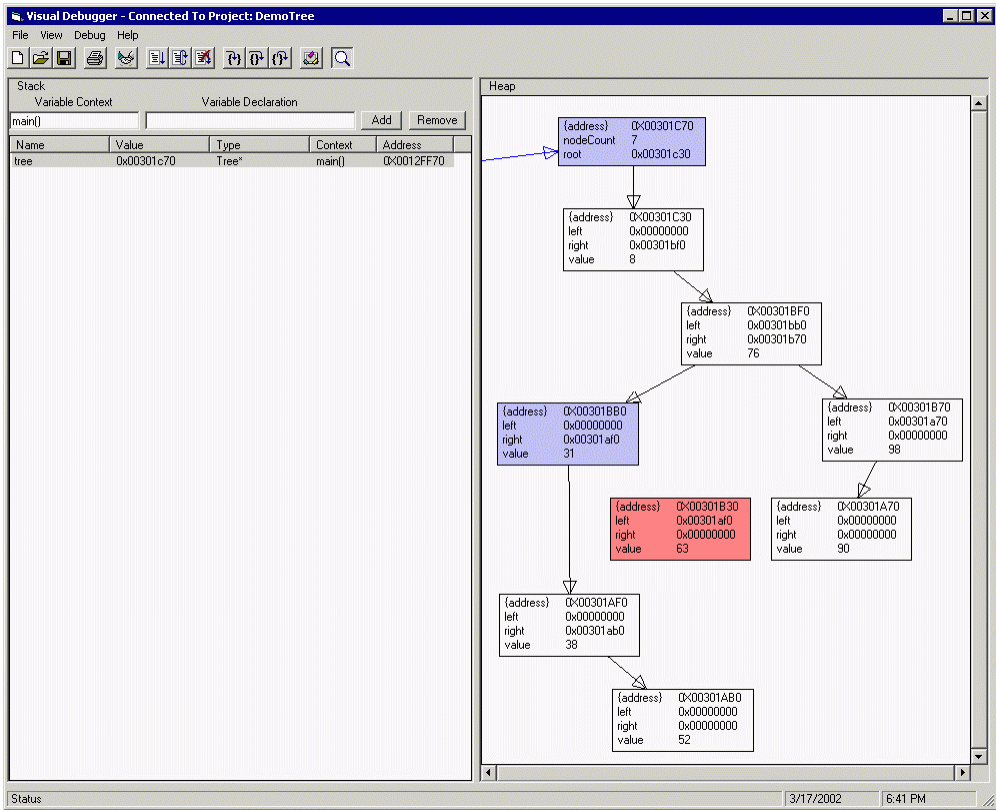


Figure 32: Deletion of a node from the tree

Eventually, the code deletes all the nodes from the tree and leaves the tree object alone as the only living object left on the heap (Figure 33). The dead nodes are still visible to the user, allowing him or her to check on their last known values before they were deleted from memory. The code, however, calls for the creation of a second, larger binary search tree, so the user will probably want to make room by moving the dead nodes out of the way.

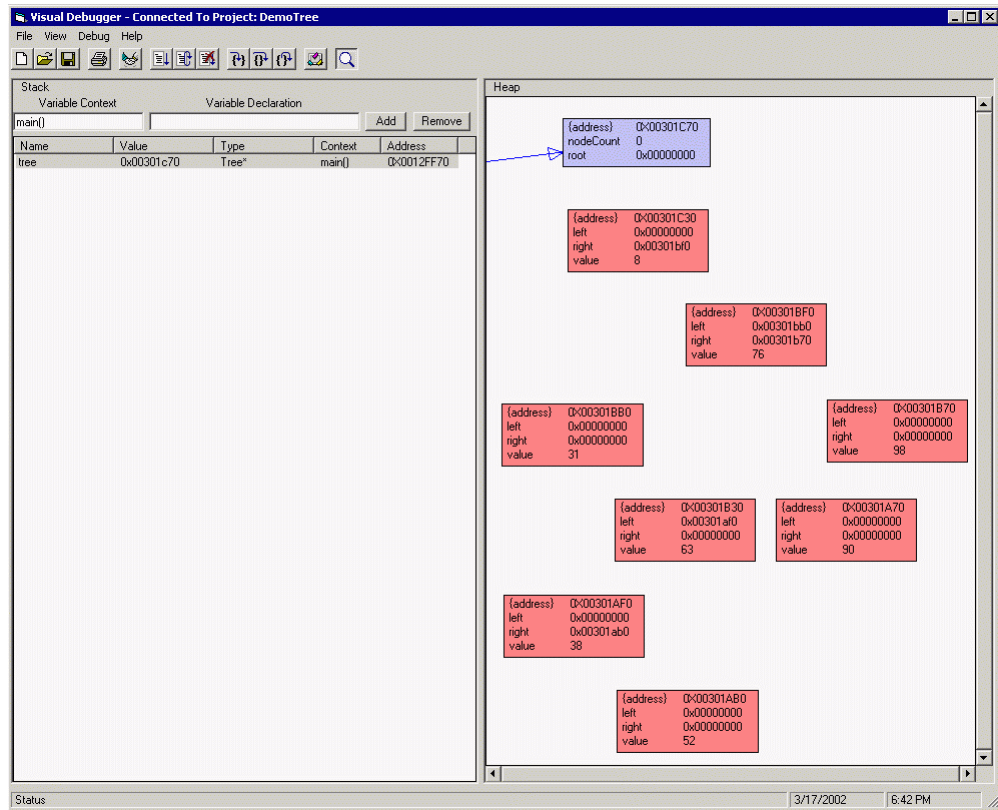


Figure 33: After all the nodes have been removed from the tree

The next segment of the user's code creates a new tree, called `tree2`. The user did not add `tree2` as a stack variable before running the program, but this is not a problem, since the user can simply add it now. Its values will be updated to reflect the new addition without requiring the user to restart the debugged program.

Eight nodes filled a medium-sized drawing canvas relatively quickly, providing motivation for the "Zoomed Out" mode. The `tree2` object will contain nineteen nodes, so the user may wish to switch to the "Zoomed Out" mode via either the menu or the toolbar to better examine the structure of the tree.

In "Zoomed Out" mode, only one field is included in a node's displayed box, and it is not labeled. This saves a lot of space on the drawing canvas, allowing for

many more nodes to be packed into a smaller space. Figure 34 shows how nineteen nodes in “Zoomed Out” mode can be packed into a smaller space than the original eight nodes in “Zoomed In” mode.

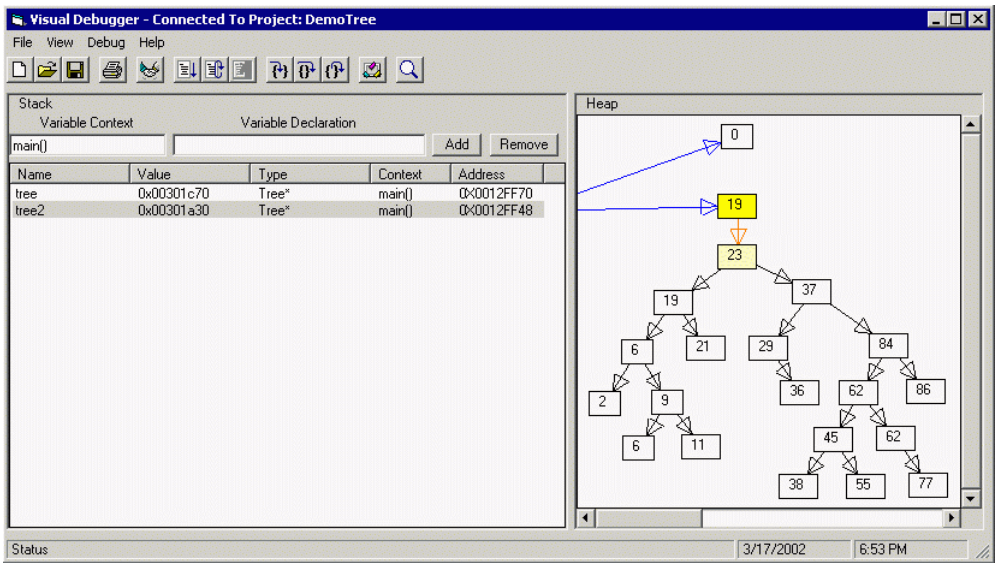


Figure 34: A tree with nineteen nodes

The two objects of type Tree are both pointed to by their respective stack pointer variables, and they are each represented by a single integer. Incidentally, `tree2` is the selected object, which causes its single child (the root of the tree) to highlight yellow and its parent (a stack variable) to highlight gray.

When defining the Tree class, the user selected `nodeCount` to be the field displayed during “Zoomed Out” mode, so the integers refer to the number of nodes in the tree. The Node objects, however, are represented by the `value` field.

“Zoomed Out” mode is very effective in conserving screen real estate. It is most effective when there is a single field in a class that can be singled out as the most important. Should there be more than one important field, however, it is not difficult to edit the class to display only those two fields in “Zoomed In” mode.

With respect to the guidelines discussed earlier in this chapter, the Visual Debugger outdoes its predecessors with two of the properties that they all overlooked. The Visual Debugger clearly draws a line between data that is on the stack and data that is on the heap, but simultaneously illustrates appropriate relationships between the two types of data. Also, by offering the programmer the ability to select which fields of a data structure are important, and by implementing a second, more compact viewing mode (“Zoomed Out”) the Visual Debugger eliminates a lot of unnecessarily wasted space and is able to display much larger data structures than its predecessors.

The Visual Debugger is still very early in its development, however, and is lacking most of the guidelines that the other debuggers implemented. The Visual Debugger is intrusive to the programmer, requiring him to compile code into his program to help monitor the heap, while all but one of the other visual debuggers (Deet) use separate debugging programs that are unintrusive. The Visual Debugger models data structures as graphs, but uses a very primitive layout scheme that is inferior even to DDD’s simple layout algorithms and that does not adapt to the kind of data structure being drawn. All of these inadequacies must be approached and resolved before the Visual Debugger can even begin to approach optimality.

In the very least, the Visual Debugger has demonstrated the possibility of improving visual debugging effectiveness through field-selection and illustration of the relationship between the stack and the heap. This shows that all of the guidelines that were given earlier in this chapter are, in fact, possible, except for one: data-structure-dependent graph layout.

5.2.1 *Implementation*

The Microsoft Visual Studio 6 Debugger does not provide a lot of assistance in its API for collecting information about data variables in a user's program. There is no way to retrieve the names of the classes in a Visual C++ project or any of its data members or methods. This was not too large an obstacle, for it had been decided to give the user flexibility with defining classes as they want them to be viewed in the debugger (hence the Class Editor), so the Visual C++ Debugger's inabilities in that respect were simply ignored. More importantly, the debugger provides no accessible information regarding data on the heap, posing a serious problem in trying to map and display the heap in the Visual Debugger.

To solve the heap problem, a single file must be included by the user's source code before compilation. The file contains overloaded versions of the `new` and `delete` operators. These overloaded operators keep track of all data that is allocated to the heap via `new`. Unfortunately, the Visual Debugger is unable to track allocations made using `malloc()`.

Tracked allocations are stored as strings in a data structure with known, hard-coded variable names. This allows the Visual Debugger to query the Visual C++ Debugger for the values stored in the variables referenced by these known variable names. The Visual Debugger then decodes the returned strings into information regarding all of the data stored onto the heap with the user's program. The Visual Debugger analyzes the data in order to manipulate the representative graph.

At the beginning of the debugging session, the Visual Debugger contains only an empty graph, but as the debugged program proceeds, modifications are made to the graph to correspond to changes in the user's data structures. Currently, nodes are not automatically moved around to make room for new nodes. This provides

the stability that is absent in the debuggers that use Doty to layout their graphs, but it requires the user to manually move objects around the drawing area if they are expecting many more nodes to be created.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

1. There now exist guidelines for a solution to the problem stated in Chapter 4, as enumerated in Chapter 5, for providing a visual debugging tool that provides optimal usability to debugging programmers. These guidelines should not be considered as sufficient conditions, but rather as *necessary* conditions, for an ideal solution.
2. Implementation of all but one of the aforementioned guidelines has been shown to be possible. The Visual Debugger implements those guidelines that have not been implemented by other debuggers, except for data-structure-dependent graph layout.
3. Today's finest visual debugging tools are still lacking in three areas: allowing the programmer to select which fields of a data structure are important; providing graph layouts that are specific to the type of data structure that the graph represents; and illustrating the relationship between data on the stack and data on the heap.
4. The Visual Debugger, though early in its development, has made strides toward resolving two of the three problems common to its visual debugging predecessors, and it can be used as a model in some respects for a more ideal data structure visualization tool, but it needs much more work before it can be considered as a competitive visual debugging utility.

6.2 Summary of Contributions

1. Guidelines for a solution to the problem stated in Chapter 4 have been enumerated for providing a visual debugging tool that, while perhaps not optimal, is at least better than today's visual debugging solutions.
2. A visual debugging tool has been created that demonstrates the possibility of implementing two of the three guidelines that have yet to be realized.

6.3 Future Research

1. It still remains to be proven that it is possible for a visual debugger to create graph layouts as appropriate with respect to the type of data structure the graph represents. There are two known suggestions for implementing this feature in the Visual Debugger. The first asks the user for the type of data structure that should be depicted in the drawing area. The second suggestion is to automatically infer the data type from the definition of the user data types. The latter would be exceedingly more difficult to implement, but it may be worth the effort in order to relieve work from the user. The bulk of current research for the Visual Debugger is focused on this most important aspect of visual debugging: proper layout of the representative graph.
2. The Visual Debugger maintains graph stability by not automatically moving nodes at each update, even when there is very little room for inserting newly created nodes. It would be convenient if a graph layout algorithm could automatically shuffle nodes to make room for new nodes, while still maintaining a decent amount of stability. Animation may prove useful in explicitly indicating to the user when and how such shuffling occurs.

3. The Visual Debugger currently employs an intrusive method for monitoring the heap. Ideally, the Visual Debugger should not be intrusive, and research is currently underway to realize this goal.
4. The Visual Debugger is very platform- and language-dependent. It is compiled in Visual Basic, so it can run only on a Windows operating system, and it targets only the Visual C++ Integrated Development Environment. Programmers wishing to use the Visual Debugger must use the Visual C++ Debugger. This is a huge drawback that must be remedied before the Visual Debugger can be released to the masses.
5. The ability to select which fields are important carries with it the inconvenience of requiring the programmer to manually define all the pertinent data types in his or her project. A proposed alternative is to parse the user's source files (or possibly the binary executable) and automatically collect data type information, presenting afterwards to the user a list from which they can select the appropriate fields. This eliminates the inconvenience of manually defining data types while still providing the flexibility of displaying only the fields that are important to the programmer.
6. The stack list in the Visual Debugger is too simple. Structures defined on the stack are displayed in a single, uninformative line. Development is under way to provide a stack variable list similar to the one implemented in the Visual C++ Debugger, using a hierarchical representation for stack data.
7. The stack list in the Visual Debugger also uses a disproportionately large amount of screen space compared to the amount of information that is displayed. Moving the stack list to the top of the window is not a viable solution, because it disrupts the ability to draw arrows between individual stack variables and their heap targets. One possible solution allocates the

lower part of the stack list, the part not used to display information, to the heap window, creating an L-shaped heap window.

8. Currently, deleted nodes remain on the Visual Debugger's screen as "dead" nodes until the debugged program is restarted. The Visual Debugger can better handle deleted nodes by allowing the user to remove them from the display after they are no longer needed.

REFERENCES

- [1] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*, Third Edition. Addison Wesley Longman, Reading, MA, Apr. 2000.
- [2] D. R. Hanson and J. L. Korn. A Simple and Extensible Graphical Debugger. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 174-183, Anaheim, CA, Jan. 1997.
- [3] M. C. Kasten. Data Structures in COBOL. Mar. 18, 2002. <<http://home.swbell.net/mck9/cobol/tech/struct.html>>
- [4] E. Koutsofios and S. C. North. Applications of Graph Visualization. In *Proceedings of Graphics Interface*, Banff, Alberta, Canada, pages 235-245, 1994.
- [5] S. C. Locke. Graph Theory. Sept. 1, 2001. Mar. 18, 2002. <<http://www.math.fau.edu/locke/graphthe.htm>>
- [6] W. Pierson and S. H. Rodger. Web-based Animations of Data Structures Using JAWAA. In *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, pages 267-271, 1998.
- [7] D. Tunkelang. A Practical Approach to Drawing Undirected Graphs. *Technical Report CMU-CS-94-161*, Carnegie Mellon University, School of Computer Science, 1994.
- [8] A. Zeller. *Debugging with DDD*. Jan. 3, 2000. Free Software Foundation, Inc. Mar. 18, 2002. <http://www.gnu.org/manual/ddd/html_mono/ddd.html>
- [9] A. Zeller and D. Lütkehaus. DDD — A Free Graphical Frontend for UNIX Debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, Jan. 1996.
- [10] Using the GNU Visual Debugger. Jan. 24, 2002. ACT-Europe. Mar. 18, 2002. <<http://libre.act-europe.fr/gvd/gvd.html>>

APPENDIX A: THE VISUAL DEBUGGER DEMOTREE CODE

```
#include "newVD.h" // the only line of code added to use VD
#include <stdlib.h>
#include <time.h>

class Node {
public:
    Node() { value = 0; left = NULL; right = NULL; }

    Node(int v, Node* l = 0, Node* r = 0)
        { value = v; left = l; right = r; }

    int value;
    Node* left;
    Node* right;
};

class Tree {
public:
    Node* root;
    int nodeCount;

    Tree() { root = NULL; nodeCount = 0; treeCount++; }

    Tree(Node* r) { root = r; nodeCount = 1; }

    ~Tree() { treeCount--; }

    void insert(int newVal) {
        if(root == NULL) root = new Node(newVal);
        else {
            Node* curr = root;
            int depth = 0;
            while(true) {
                depth++;
                if(newVal >= curr->value) {
                    if(curr->right == NULL) {
                        curr->right = new Node(newVal);
                        break;
                    }
                    else
                        curr = curr->right;
                }
                else {
                    if(curr->left == NULL) {
                        curr->left = new Node(newVal);
                    }
                }
            }
        }
    }
};
```

```

        break;
    }
    else
        curr = curr->left;
    }
}
}
nodeCount++;
}

void remove(int oldVal)
{
    if(root == NULL) return;
    Node* curr = root;
    if(root->value == oldVal) {
        nodeCount--;
        root = deleteNode(root);
    }
    else {
        while(curr != NULL) {
            if(oldVal > curr->value && curr->right != NULL)
                if(curr->right->value == oldVal) {
                    curr->right = deleteNode(curr->right);
                    nodeCount--;
                    return;
                }
            else
                curr = curr->right;
            else if(oldVal < curr->value && curr->left != NULL)
                if(curr->left->value == oldVal) {
                    curr->left = deleteNode(curr->left);
                    nodeCount--;
                    return;
                }
            else
                curr = curr->left;
        }
        break;
    }
}

int getRandVal()
{
    if(root == NULL)
        return -1;
    srand(time(NULL));
    Node* curr = root;
    while(curr->left != NULL || curr->right != NULL)
    {

```

```

        int num = rand() % 5;
        if(num == 0) return curr->value;
        if(curr->left == NULL)
            curr = curr->right;
        else if(curr->right == NULL)
            curr = curr->left;
        else if(num % 2 == 0)
            curr = curr->left;
        else
            curr = curr->right;
    }
    return curr->value;
}

private:
    static int treeCount;

    Node* deleteNode(Node* toDelete)
    {
        Node* delNode;
        if(toDelete->left == NULL && toDelete->right == NULL)
        {
            delete toDelete;
            return NULL;
        }
        else if(toDelete->right == NULL) {
            delNode = toDelete;
            toDelete = toDelete->left;
            delete delNode;
        }
        else if(toDelete->left == NULL) {
            delNode = toDelete;
            toDelete = toDelete->right;
            delete delNode;
        }
        else {
            for(delNode = toDelete->right;
                delNode->left;
                delNode = delNode->left);
            delNode->left = toDelete->left;
            delNode = toDelete;
            toDelete = toDelete->right;
            delete delNode;
        }
        return toDelete;
    }
};

int Tree::treeCount = 0;

```

```

int main()
{
    Tree* tree = new Tree;

    srand(time(NULL));
    int i;
    for(i = 0; i < 8; i++)
    {
        tree->insert(rand() % 100);
    }

    while(tree->nodeCount > 0)
        tree->remove(tree->getRandVal());

    Tree* tree2 = new Tree;
    for(i = 1; i < 20; i++)
    {
        tree2->insert(rand() % 100);
    }

    while(tree2->nodeCount > 0)
        tree2->remove(tree2->getRandVal());

    delete tree;
    delete tree2;

    return 0;
}

```