Visualization Tools for Debugging Simple C and C++ Data Structures

Jonathan Geisler
Taylor University
23 PUBLICATIONS **610** CITATIONS
SEE PROFILE

PROCEEDINGS OF THE
MIDSTATES CONFERENCE ON
UNDERGRADUATE RESEARCH IN
COMPUTER SCIENCE AND MATHEMATICS
Volume 3, Number 1, Pages 29–37

VISUALIZATION TOOLS FOR DEBUGGING SIMPLE C AND C++ DATA STRUCTURES

AUSTIN BEER, LAM TRAN, AND JONATHAN GEISLER

ABSTRACT. This paper details the motivation, process, results, and future work of a project designed to help beginning students succeed in a data structures class. Data structures has been a historically difficult computer science course. Current debuggers are inadequate for such situations, as they are not designed to allow beginning students an easy way to effectively visualize and debug their own data structure implementations. Thus a better debugging tool is needed. This project's aim is to develop a visual debugging tool that supports the ability to automatically display data structures (and continually update them), animate them (i.e. update the data structure display without stopping the debugged program), and display them with user-friendly layouts, all in order to help students understand data structures more efficiently and effectively and gain a higher level of achievement in data structures courses.

1. INTRODUCTION

For this project we are modifying a mature, open source visual debugger (DDD) by adding the ability to automatically display visual representations of varying data structures, using DDD's data display component. We are limiting our scope to just C and C++ programs running on the Unix/Linux platform. The platform limitation is due mainly to our choice of using DDD, which only runs under Unix/Linux.

At this point, several terms need to be defined:

(1) *GDB (GNU's Debugger)*: This is GNU's standard, open source debugger which is found on almost all Linux machines. It is command line based and only works with programs written in non-interpreted languages (i.e., programs compiled to machine level code). However, it is by far the most popular debugger in use within the open source community.
(2) *DDD (Data Display Debugger)*: This is GNU's open source, graphical front end to GDB. It does not actually interface with the program being debugged. Instead it invokes GDB and uses its commands to control the program and to get information about it. The two biggest advantages of DDD are its ability to issue commands to GDB via menus and its ability to graphically display any of the debugged program's data items (via the data display window).
(3) *Node*: This is a single item within a data structure, such as a linked list. It is usually composed of an instance of a single C++ class (e.g., a ListNode class) and usually contains one or more data elements and one or more pointers which connect it with the rest of the data structure. It is displayed in DDD as a single entity.
(4) *Expand*: This paper will often use the verbs "expand," "autoexpand," "expanding," etc. What these are referring to is the act of dereferencing a pointer in one node and then graphically displaying that dereferenced value (which is almost always another node) in the data display window.

2. MOTIVATION AND PURPOSE

The primary motivation behind this project is to increase the effectiveness of teaching data structures to beginning CS students. Data structures has been a historically difficult course. This is because it is a transitional course from basic programming in CS1 to more complex programming and software development in upper level courses. Students are given more freedom, and thus more responsibility, to write their own programs and data structures. This requires a much more solid grasp of the basic concepts as well as an ability to creatively and independently apply them. It is because of this that many students

Received by the editors September 15, 2005 and in revised form October 24, 2005.

struggle in data structures courses and it is at this stage many students decide whether or not they will continue in the CS curriculum.

The specific problem we identified is that students do not have an easy way to visualize and debug their own data structure implementations. There are, of course, tools to visualize generic data structures (see related work below), but often what the student sees on a blackboard or in a demonstration doesn't translate well into their own implementations. The goal of this project is to remedy this problem by enabling the student to see their own implementation while it is running and, by extension, to more quickly and effectively debug errors which frequently arise in such implementations.

The current resources available to students are inadequate for this task. Command line based tools, such as GDB, are not visual and are difficult for beginning students to learn. A data structure, by its very nature as a structure, is best viewed in a visual format. However, even visual debuggers such as DDD and Microsoft's Visual Studio do not meet the needs of students in this respect because they require manual expansion of the data structure. There is no way for students to issue one command and see the entire data structure in its current state in an easy to recognize format.

3. Related Work

(1) Canned animations in Java [1] and Ada [2]. Canned animations are useful for demonstrating how data structures work in the classroom or online, but not for developing code. With canned animations students cannot experiment with their own data structure implementations.
(2) Data structures libraries that generate their own visualizations (e.g., JSAVE (Figure 1) [4]). These must be implemented for each type of data structure for which the user wants to have graphical visualizations. The user does not need to write the implementation. They are usually limited to basic data structures and do not work with novel or custom data structures.
(3) DDD [6]. DDD allows users to graphically display and debug data structures without adding additional code. It is able to display any type of data structure. However, the user must manually walk through and expand each of their data structures. They also have to manually arrange nodes in order to have the display layout they want.

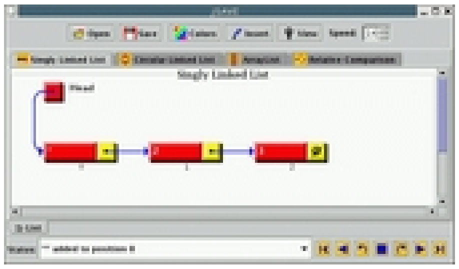
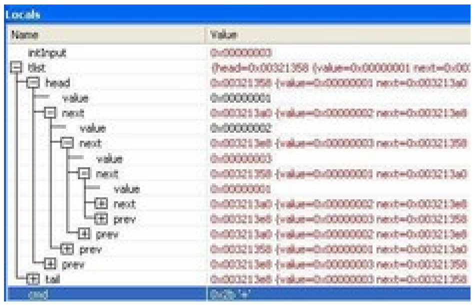


Figure 1. JSAVE screenshot



Locals

Name	Value
intInput	0x00000003
tlist	{head=0x00321358 {value=0x00000001 next=0x00
head	0x00321358 {value=0x00000001 next=0x003213a0
value	0x00000001
next	0x003213a0 {value=0x00000002 next=0x003213e8
value	0x00000002
next	0x003213e8 {value=0x00000003 next=0x00321358
value	0x00000003
next	0x00321358 {value=0x00000001 next=0x003213a0
value	0x00000001
next	0x003213a0 {value=0x00000002 next=0x003213e8
prev	0x003213e8 {value=0x00000003 next=0x00321358
prev	0x003213a0 {value=0x00000002 next=0x003213e8
prev	0x00321358 {value=0x00000001 next=0x003213a0
prev	0x003213e8 {value=0x00000003 next=0x00321358
tail	0x003213e8 {value=0x00000003 next=0x00321358
cmd	0x2b '+'

Figure 2. MS Visual Studio.Net screenshot

(4) Microsoft Visual Studio .NET (Figure 2 [7]. Users' data structures are displayed in a tree format. Users must manually expand their data structures as well. It is hard for users to debug circular data structures with this tool. They can only detect circular references by manually comparing pointers and finding identical memory addresses.

4. Problems Solved

There were several significant problems which we have had to overcome during the course of the project.

4.1. **Modifying a mature program.** DDD is a larger program than anything we have ever worked on before, with almost 120,000 lines of code [5]. It also comes with very little in the way of documentation. Most of what of does exist consists of one or two line descriptions at the top of various functions. What is lacking, though, is documentation giving a detailed description of the general architecture of DDD and how all of the classes and functions work together. There is also very little documentation within functions describing why things are coded a certain way and what the consequences are of coding them a different way. This makes it especially difficult for us to avoid creating unintentional side effects when implementing our own modifications.

We have had to learn the code simply by reading through it and experimenting with the effects of making changes to this or that part or adding small things here and there. However, due to the size of the code base and the time constraints of the project, there is no way we could learn the function and purpose of every single line of code, so in the process of learning the code we have had to determine which parts are relevant to our project and which are not, and then focus only on those parts.

4.2. **Cyclical Data Structures.** Our algorithm checks for any pointers which have not been expanded and then automatically expands them. This works fine on any tree-like data structure where you don't have to worry about a node referencing one of its parent or ancestor nodes. However, with circular linked lists and other cyclical data structures, this becomes a huge problem that, if left unresolved, will result in infinite recursion or an infinite loop.

To solve this, we have developed an algorithm which checks the entire displayed data structure for duplicates before expanding any unexpanded pointer. We then use DDD's aliasing feature to make these data structures appear circular, instead of appearing as a tree. See Figures 5 and 6 for examples of DDD's aliasing feature in action.

DDD's aliasing feature takes identical nodes and joins them together in the data display window so that they appear to the user as one node. The arrows which represent pointers and which join nodes together remain in place, so the user can clearly see when two pointers point to the same node. This is necessary because DDD never expands a pointer by just drawing an arrow to an already-displayed node, even if an already-displayed node is what the pointer is pointing to. Instead, DDD instead creates a new display node and draws a new arrow pointing to it. Thus, it sees all pointer-based structures as trees with parent and child nodes. Aliasing is necessary to make what the user sees on the screen match his or her mental picture of what the data structure should look like. Because of the way aliasing works, then, our algorithm has to be designed to sometimes create duplicate nodes in order to insure that one (and only one) arrow is visible for each pointer in the data structure.

4.3. **Data Structure Types.** Introductory data structures courses usually just introduce students to basic data structure types such as lists, doubly linked lists, binary search trees, m-array trees, and graphs. Our project is limited to simple data structure layouts which introductory data structures students need. Our algorithm displays the data structures in different display layouts, depending on its type (see future work below). If no specific display layout is selected when the user enters an autodisplay command, the program examines the data structure and tries to determine what the data structure type is based on the names of the variables. This is necessary because the data structure type is needed in order for the layout algorithm to determine which layout to use.

If the data structure has only one pointer, the data structure can be laid out as a singly linked list because it can only be expanded in one direction. But what if there is more than one pointer in the data structure? It could be a doubly linked list, a tree, or even a graph. The solution is to use the "ptype" command on the data structure to obtain the names of the member variables ("ptype" is a GDB command that returns the member variables and functions of a given class). Each member variable name is then compared with lists of popular names normally used with specific data structures. Some popular names for each data structure are:

- *List*: next, previous, backward, forward
- *Tree*: left, right, parent, root
- *Graph*: edges, nodes

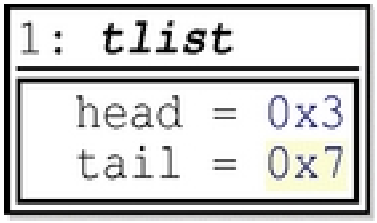
Points are given to each data structure type for each member variable name that is found in the lists of popular names, and points are awarded by the number of characters that match and additional points are given when a variable name matches exactly. It then recursively steps into each nested data structure that has not been encountered before and calculates points for each of these data structures as well for a more accurate prediction. When the calculations are done the data structure that has the most points is selected as the display layout. This scheme works as long as the user follows good naming conventions and does not make up variable names randomly. A future version upgrade is planned to allow the user to select another display layout and redraw the data structures with the new layout after an initial layout has been selected.

4.4. **Fast User Input.** After implementing our autoexpand algorithm, we have encountered an unexpected issue with errors sometimes arising if users enter new commands before the algorithm has finished running. This is especially noticeable when the user holds down the enter key, which causes DDD to simply repeat the last entered command over and over again as fast as possible.

DDD communicates with GDB over a text-based interface with no direct connection between the two program's threads. Thus when DDD sends a command to GDB it expects a very specific type of response in order to be able to correctly parse it. Consequently the two programs must always be completely in sync, otherwise errors can arise. Our issue arises because the autoexpand function is slowing down DDD enough to allow user input entered while that function is running to throw the communication channel between the two programs out of sync. The user commands are sent straight to GDB while DDD is still processing the autoexpand function and expecting data from GDB related to that command. The result is that DDD receives data from GDB different from what it is expecting.

The reason our algorithm is slow is because it relies on DDD's own display functions to display nodes and update the data display, and these functions are relatively slow. The other aspect is that these functions, while slow, are constant time, whereas our algorithm is linear time. It has to traverse the whole data structure every time, looking both for new pointers to expand and making sure it doesn't unnecessarily expand duplicate pointers. It often then calls multiple display commands simultaneously. This whole process is repeated iteratively until no new pointers are found to expand. DDD was originally only designed for single display commands to be issued–one at a time–by the user, so this has never been a problem before.

The best solution we have found is to simply ignore user input while the autoexpand function is running. This isn't an ideal solution but it does work and it is the best we can come up with given the limitations of DDD's communication protocol with GDB and the time constraints of the project.



```
1: tlist
  head = 0x3
  tail = 0x7
```

FIGURE 3. Invalid pointers

4.5. **Invalid Pointers.** Since the autoexpand algorithm only expands valid pointers, it is necessary to determine which pointers are pointing to a valid memory address and which are not. Unfortunately, invalid pointers are not limited to null (0x0). Valid pointers for a program should be in the range of the memory addresses that the program has been allocated. See Figure 3 for an example node that contains invalid pointers which should not be automatically followed by the final algorithm.

The solution to determining whether or not a pointer is valid is to use "info proc" (a GDB command for getting the debugged program's process number) and then accessing the debugged program's memory file in the Unix /proc directory and parsing out the allocated memory address ranges. A valid pointer is determined by comparing to see if it is the range of the allocated memory addresses.

5. Final Solution

The final solution consists of the following features and modifications:

- It automatically and recursively dereferences and displays each valid pointer in a user-specified data structure, starting with a single, base node.
- It accurately displays circular linked lists and other cyclical data structures and avoids the infinite recursion which might be caused by expanding such data structures.
- It checks all pointers against the debugged program's allocated memory range and only expands those pointers which are valid.

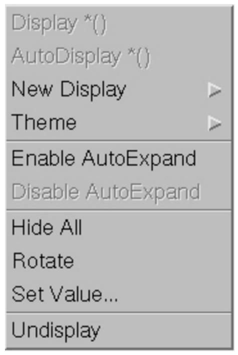


FIGURE 4. Example popup menu

- It includes an "animate" command, which enables a data structure to autoexpand without the debugged program stopping at breakpoints during execution. Breakpoints are set internally by the animate function and the data display is updated at each of those breakpoints, but the debugged program is automatically continued so the user never realizes that it is actually repeatedly stopping and starting. Thus the user can interact with the program just as if the debugger were not running (no need for break or continue commands), yet he or she can still see the full data structure being displayed and updated in the data display window.
- Using DDD's display history, it allows the user to view past states of an animated data structure after the debugged program has completed execution (see future work below). This is what makes the animate function valuable. The user can undo the data display to the point where the data structure went wrong and see exactly in the program code where the error occurred.
- It allows an entire autoexpanded data structure to be undisplayed with a single user command. If the user undisplays an autoexpanded parent node DDD will ask if the child nodes descended from that parent should be deleted as well.
- Autoexpand can be enabled or disabled at any time for any part of a displayed data structure.
- GUI menu access (Figure 4) is available in DDD for all of these autoexpand features.

Figure 7 illustrates an example session using the final solution. The user is trying to view a doubly linked list during execution. The first step tells DDD to autoexpand the data structure. The next step continues until a user-defined breakpoint. During the program execution, the user enters a value (1) to be entered on the list. At the breakpoint, DDD now shows the list populated with a single entry. The final two steps repeat the process where the user adds 2 and 3 to the list. Note that no extra commands have to be entered into DDD to update the list with the new values–they are automatically updated to add the new nodes that the user added.

6. Current Limitations

The modified version of DDD has only been developed and tested with C and C++ programs and with GDB as back-end debugger to DDD. Since DDD can be extended to use other debuggers and other programming languages, this modified version might work with other programming languages and debuggers as well without additional modifications, but there is no guarantee. Additional testing is necessary.

This program also does not scale well to large data structures even though it has been modified a few times and has improved from a polynomial running time to a linear running time. It is slightly slower than the original DDD when debugging large data structures. Also, large data structures do not work well simply because a large amount of space is required to display them. Beyond a certain size, viewing such data structures becomes impractical.

The autoexpand algorithm does not autoexpand composite data structures. An example of a composite data structure is a data structure nested within another a data structure, such as a data structure containing a pointer to a list of pointers. The autoexpand function only autoexpands the pointer to the list, not the pointers in the list, because it has been implemented to only expand the top level pointers.

7. Future Work

There are multiple avenues for continued development and research with this project:

- While the animate function works and works well, in certain situations DDD's undo buffer doesn't always fully show the past states of the displayed data structure. This is due to a design limitation of the undo buffer and so fixing it requires modifying that part of DDD, described in the code comments as “a nasty and obfusticating [sic] piece of DDD.” Without this ability, the animate function loses much of its usefulness to students, and so this should be a high priority for any future development.

- The autoexpand function does not yet have a corresponding layout function. At this point, the display layout is being handled by DDD’s default autolayout algorithms, which are not bad but which are designed to accommodate anything the user might wish to display. Thus it does not cater to the specific needs of displaying simple data structures for beginning students. The two biggest requirements for such are displaying data structures with a layout similar to what the user is expecting (especially true for tree data structures) and minimizing the amount of scrolling necessary to see the entire data structure (a list shouldn’t just expand down or to the left, as anything bigger than three or four nodes would require scrolling in one or the other of those directions). Some progress has already been made on this front and a rough version of a list layout function has already been demonstrated, but there is much that still needs to be done (especially with trees) before the full layout function can be included with the other modifications.
- Right now the autoexpand algorithm only supports programs written in C and C++. It would be nice to see this functionality expanded to other languages, such as Java. There would be challenges, though, as many languages do not contain explicit pointers like C and C++ do.
- DDD supports several back-end debuggers other than GDB, but at this point the autoexpand function has only been developed to work with that specific debugger. It would be nice to allow users to use other available debuggers, such as DBX and JDB.
- Right now this functionality is only available on the Unix/Linux platform. As such, there is an excellent opportunity for investigating the possibility of implementing something similar on the Windows platform, which is much more pervasive. One of the main challenges of this, obviously, would be finding a quality visual debugger which was also open source.
- The main reason for developing this autoexpand feature is to help students and their professors in data structures courses. Now that that has been accomplished, the question becomes whether or not it actually fulfills that purpose? A study of students in a real-world classroom setting needs to be conducted to answer questions like:
  - Does this improve students’ comprehension? grades? coding speed?
  - Does it decrease student frustration and/or lack of motivation?
  - Does it decrease students’ drop-out rate?
  - How does this tool compare with other options available to professors?

8. Conclusion

The goal of this project is to develop a better way to visualize and debug data structures implemented by beginning CS students. We have achieved this by adding an autoexpand feature to the visual debugger DDD as well as an animate feature. The next step is to determine how useful this tool is to beginning students in real-world classroom settings. We anticipate that the results will show the benefit of this tool and hope that it will be incorporated into the next release of DDD.

References

1. Tao Chen and Tarek Sobh. *A Tool for Data Structure Visualization and User-defined Algorithm Animation,* in proceedings of the thirty-first ASEE/IEEE Frontiers in Education, October 2001.
2. Duane J. Jarc and Michael B. Feldman. *A [sic] Empirical Study of Web-based Algorithm Animation Courseware in an Ada Data Structure Course,* in proceedings of SigADA, 1998.
3. Matt Roddan, *The Determinants of Student Failure and Attrition in First Year Computing Science,* August 2, 2005, http://www.psy.gla.ac.uk/~steve/localed/roddanpsy.pdf.
4. Nick Sumner, Daniela Banu, and Herb Dershem. *JSAVE: Simple and Automated Algorithm Visualization Using the Java Collection Framework,* in proceedings of the tenth annual Consortium for Computing Sciences in Colleges, October 2003.
5. David A. Wheeler. *SLOCCount User’s Guide,* http://www.dwheeler.com/sloccount/sloccount.html, August 2004.

6. Andreas Zeller. *Animating data structures in DDD*, in proceedings of the SIGCSE/SIGCUE Program Vizualization Workshop, July 2000.
7. Microsoft Corporation. http://msdn.microsoft.com.

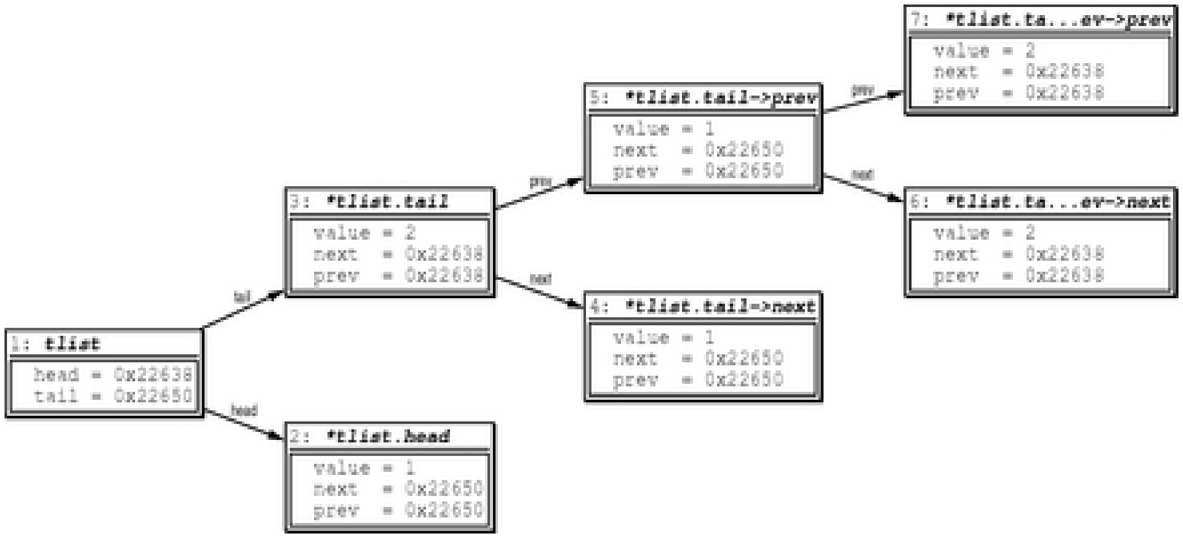


Figure 5. Unaliased circular list

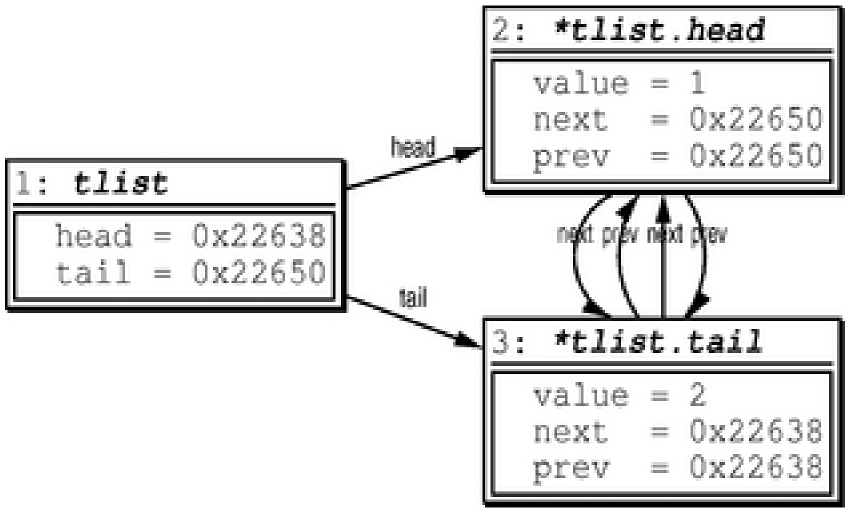


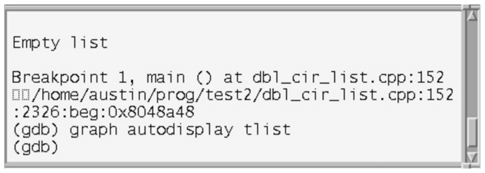
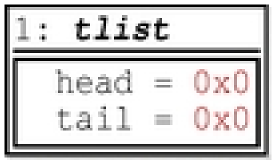
Figure 6. Aliased circular list

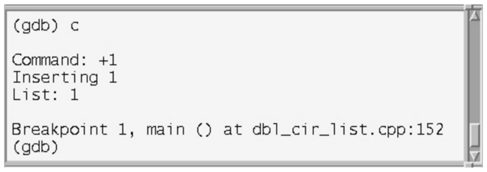
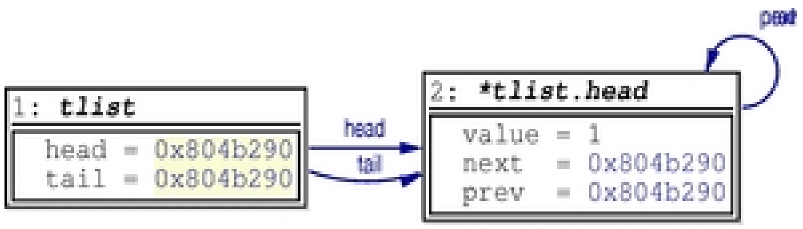
Computing & System Sciences Department, Taylor University, Upland, IN 46989
E-mail address: `abeer@css.taylor.edu`

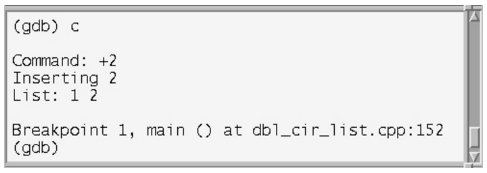
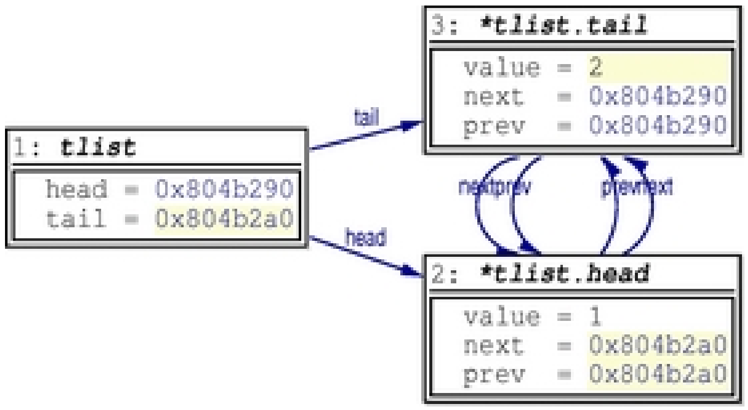
Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180
E-mail address: `lamt@rpi.edu`

Computing & System Sciences Department, Taylor University, Upland, IN 46989
E-mail address: `jgeisler@css.taylor.edu`

DDD commands	Graphic output

 Empty list
 	
Breakpoint 1, main () at dbl_cir_list.cpp:152
□□/home/austin/prog/test2/dbl_cir_list.cpp:152
:2326:beg:0x8048a48
(gdb) graph autodisplay tlist
(gdb)
	 1: tlist
	head = 0x0
	tail = 0x0

 (gdb) c
 	
Command: +1
Inserting 1
List: 1
 	
Breakpoint 1, main () at dbl_cir_list.cpp:152
(gdb)
	 1: tlist
	head = 0x804b290
	tail = 0x804b290
	2: *tlist.head
	value = 1
	next = 0x804b290
	prev = 0x804b290

 (gdb) c
 	
Command: +2
Inserting 2
List: 1 2
 	
Breakpoint 1, main () at dbl_cir_list.cpp:152
(gdb)
	 3: *tlist.tail
	value = 2
	next = 0x804b290
	prev = 0x804b290
	1: tlist
	head = 0x804b290
	tail = 0x804b2a0
	2: *tlist.head
	value = 1
	next = 0x804b2a0
	prev = 0x804b2a0

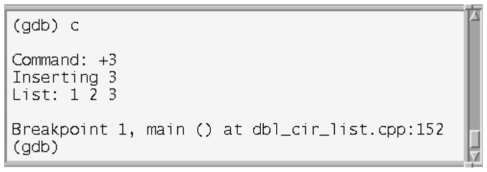
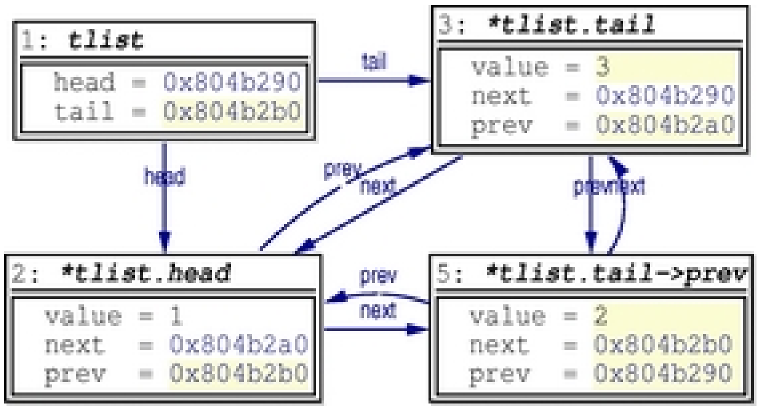
 (gdb) c
 	
Command: +3
Inserting 3
List: 1 2 3
 	
Breakpoint 1, main () at dbl_cir_list.cpp:152
(gdb)
	 1: tlist
	head = 0x804b290
	tail = 0x804b2b0
	3: *tlist.tail
	value = 3
	next = 0x804b290
	prev = 0x804b2a0
	2: *tlist.head
	value = 1
	next = 0x804b2a0
	prev = 0x804b2b0
	5: *tlist.tail->prev
	value = 2
	next = 0x804b2b0
	prev = 0x804b290

FIGURE 7. Example debugging session with new functionality