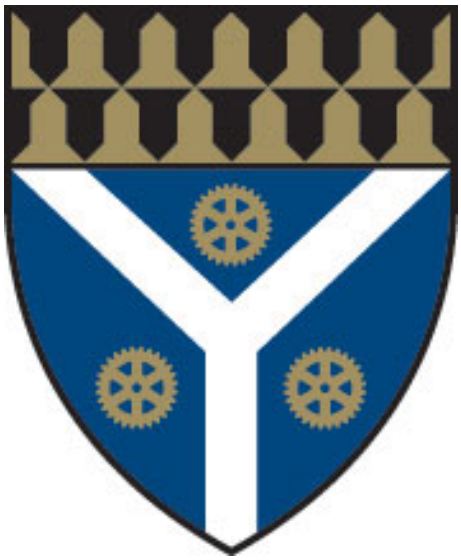
Data Visualization for Debugging and Educational Purposes

Gabriel Dos Santos

Advisor: James Glenn

May 9th, 2024



A Senior Project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

Department of Computer Science
Yale University
Spring 2024

Contents

Abstract **1**

1 Introduction **2**
1.1 Motivation . . . 2
1.2 Related Work . . . 2
1.3 Project Goals . . . 4

2 Implementation **4**
2.1 GUI Design . . . 5
2.2 Graph Design . . . 5

3 Results **6**
3.1 Use Cases . . . 6
3.2 User Study . . . 6

4 Future Work **8**

5 Conclusion **10**

6 Acknowledgements **10**

A Appendix **12**

B DataDebug API Documentation **14**
B.1 GUI Controls . . . 14
B.1.1 `Left arrow: next` . . . 14
B.1.2 `Right arrow: previous` . . . 14
B.1.3 `Up arrow: speed up` . . . 14
B.2 Logger Class Methods . . . 14
B.2.1 `create_node` . . . 14
B.2.2 `delete_node` . . . 14
B.2.3 `add_edge` . . . 15
B.2.4 `delete_edge` . . . 15
B.2.5 `clear_edges` . . . 15
B.2.6 `customize_node` . . . 15
B.2.7 `customize_edge` . . . 15
B.2.8 `add_pointer` . . . 16
B.2.9 `add_pointers` . . . 16
B.2.10 `visualize` . . . 16
B.2.11 `snapshot` . . . 16
B.3 List . . . 16
B.3.1 `__init__` . . . 16

Abstract

The goal of this project is to explore new ways of visualizing data structures, seeking to make a contribution to Computer Science education. In my previous experience as a teaching assistant, many novice programmers often face a gap between what they think their code is doing versus what it actually is doing. With a focus on trees and list-like structures, I developed a Python logging library called Datadebug. The underlying technologies used to develop this library included *pygraphviz* for graph creation and *matplotlib* for the user interface. This library allows users to easily track changes to their data structures during run-time, enabling an intuitive view of their code's results. Various hurdles were faced throughout development such as dealing with ways to handle updates to built-in data structures like lists which are important for stacks and queues. Ultimately, this research compares traditional debugging methods, such as in-line printing, against this new pythonic visual approach to identify how visualization tools can improve the understanding of data structures and facilitate the debugging process. Existing libraries face various barriers such as *GDB* providing low-level views for low-level language or *Vscode Debug Visualizer* requiring lots of effort to format data into a serializable form. This library hopes to be simple and direct for users to adopt and debug their programs. To analyze the effectiveness of this library, a small-scale user study was conducted. The study employs a combination of problem-solving exercises and feedback surveys to access the visualization library's impact on participants' ability to understand and debug code. Overall, participants found that the visualization helped in addressing bugs quicker and enabling a better understanding of the bugs.

1 Introduction

1.1 Motivation

The rapid rise of technology's integration into modern day life necessitates a corresponding advancement in the ways in which we teach and learn these related complex subjects. Consequently, the proportion of Computer Science students at universities have beyond doubled throughout the last decade as more people join the lucrative industry. The understanding of data structures – a foundation of programming and algorithm design – presents a notable challenge to new learners. This is due to their abstract nature and cognitive ability required to visualize and manipulate these structures mentally. Novice programmers are often reliant on in-line printing to gain insight on bugs in their program, however, this may not be the most intuitive method for debugging. Throughout my experience as a data structures undergraduate learning assistant, I've come to realize one of the key challenges people face with data structures is a gap between what one perceives their code is doing in contrast to what is happening in reality. Meanwhile, instructors often do not have time to visually display to students what is happening throughout each and every state update. This gap in pedagogical tools motivates the exploration of a new approach to bridge the divide.

This thesis is motivated by the hypothesis that visualizing data structures can significantly enhance the learning experience by providing a more intuitive grasp of complex concepts and facilitating a deeper understanding of the algorithms which operation on these structures. My newly implemented visual logging library, Datadebug, serves as a testing ground for investigating this claim. This tool has the potential to change the debugging process from a game of guess who to an informative experience by enabling learners to see the real-time updates of their data structures. This research aims to assess the impact of such visualization tools on the efficiency and effectiveness of examining data structures.

1.2 Related Work

The exploration of tools for enhancing debugging of data structures in programming has garnered significant attention in both industry and academic contexts. Among these, several tools have stood out for their popularity and innovation in the field. This section reviews related work focusing on gbd, jGRASP, and VSCode Debug Visualizer. This will help contextualize our contributions within the broader realm of tools.

- **GNU Debugger (gdb):** The GNU Debugger, commonly known as gdb, is staple tool in software development, specifically for low-level languages. Gdb operates through the command-line, offering granular insights on a program by setting up breakpoints and navigating pointers. It is unparalleled in its ability to control program execution and inspect variables, however, it's textual output requires users to mentally map complex structures. Using this tool is also often daunting to programmers. This underscores the need for complementary visual tools that can lower the entry barrier for novices.

- **jGRASP:** This development environment was designed by educators and provides automatic software visualization capabilities that generate representations of data structures at runtime. This Java built tool is extremely flexible, working on Java, C, and

Python programs. This IDE offers one of the best visualizations of structures with all of its handy tools. However, the work must be done on their outdated IDE and carries a learning curve. There are often a multitude of screens that need to be navigated with various buttons.

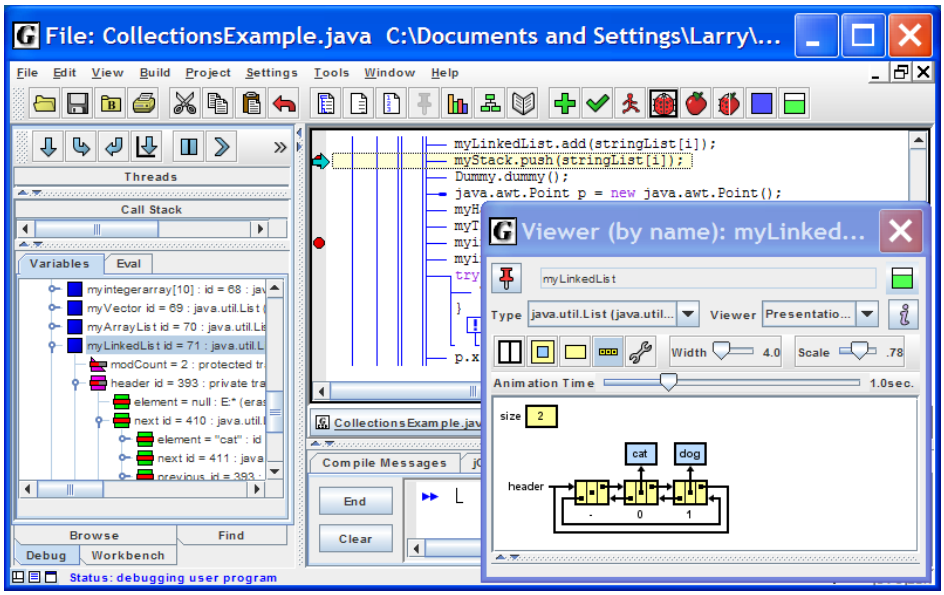


Figure 1: Snapshot of jGRASP development environment.[1]

- **VSCode Debug Visualizer:** The VSCode Debug Visualizer is a recent development. It was designed as an extension for the popular Visual Studio Code editor. The Debug Visualizer allows programmers to visually inspect the state of data structures at any point during execution and makes a significant step forward in increasing accessibility. The main downside of this library is it's high configuration demands to properly visualize the structure which includes setting up a config json file.

While tools like gdb offer classic text-based debugging capabilities, the evolution towards graphical interfaces as seen in jGRASP reflects a growing recognition of the benefits of visual aids. Each tool, with its unique features and intended audience, contributes to a richer ecosystem of resources aimed at enhancing the comprehension and debugging of data structures. This thesis builds upon the foundation laid by these tools and seeks to avoid some of their downsides.

Additionally, previous work has analyzed the use of data structure visualization on learning. Chen and Sobh have expressed how "it would be very helpful if there was a visualization tool of data structures such as arrays, queues, stacks, trees and graphs for students to manipulate"[2]. The key structures emphasized in their paper include trees, stacks, and queues and this project focuses on a similar subset of data structures. Osman, Ibrahim, and Elmusharaf similarly hypothesized that "visualization of algorithms help students achieve greater understanding of the programming concepts"[3]. Through a small study, they found a statistically significant performance improvement in those who used their visualization library. We hope to find similar results from our work.

1.3 Project Goals

The goal of this project is to develop an innovative Python library, DataDebug, aimed at enhancing the way data structures are visualized for educational and debugging purposes. This project hopes to fill the gaps in current software visualization tools by offering a more intuitive and hands off solution.

- **Educational Enhancement:** DataDebug is designed to improve the learning experience of programmers and students. By providing a visual representation of various graph-like data structures —ranging from trees to linked lists, stacks, and queues— the library makes abstract concepts more tangible and understandable. This is particularly beneficial in educational settings where learners can visually trace the *evolution* of data structures in *real-time*, thereby deepening their understanding of underlying data structures and algorithms.
- **Debugging Efficiency:** Beyond its educational value, DataDebug serves as a useful debugging tool that allows programmers to visually trace and inspect the state of data structures during code execution. This capability is critical for identifying and resolving bugs that may not be immediately apparent through common debugging methods, such as print statement logging.
- **User-Centric Design:** The library is developed with the end-user in mind, ensuring ease of use without the need for extensive setup or configuration. Unlike the previous tools, we wanted to create a visualization tool that mimicked the flexibility and ease of logging libraries. Instead of printing a state update to a text-based log file, the change is rendered and saved in a graph seamlessly.
- **Community Engagement and Feedback:** An important aspect of this project is engaging with users through small-scale tests to gather feedback and insights. This feedback is invaluable for refining the library, guiding future development, and gaining a better perspective on the users' needs.

In summary, the goals of this project are to provide a robust, intuitive, and versatile tool for visualizing data structures to enhance the learning and debugging experience for developers. This is done by creating a Python logging library which specializes in the visualization of graphs. The key difference in our project from previous work is the logging-based approach to recording state changes and rendering them.

2 Implementation

In this section, we delve into the practical aspects of developing DataDebug. We will be focusing on the implementation and challenges of the core backbone components: the graphical user interface (GUI) and the graph design. These elements are critical in achieving the project's goals of enhancing the user understanding and experience.

2.1 GUI Design

The interface for DataDebug was a key consideration during the development process. We wanted to minimize the work required to build an interface while maintaining a clean sleek look: this led us to matplotlib. This library easily allowed us to load images and display them to the user during each update. In order to facilitate easy viewing of the state, the rendering unit sleeps for a brief period of time, allowing users to grasp the state change before the next one occurs. Additionally, it was important to give users the ability to go back and forth between state snapshots. Therefore, we saved each image in memory, and through the use of event listeners, allowed users to swipe between snapshots using the keyboard arrow keys. For more GUI information, see Appendix B: API Documentation.

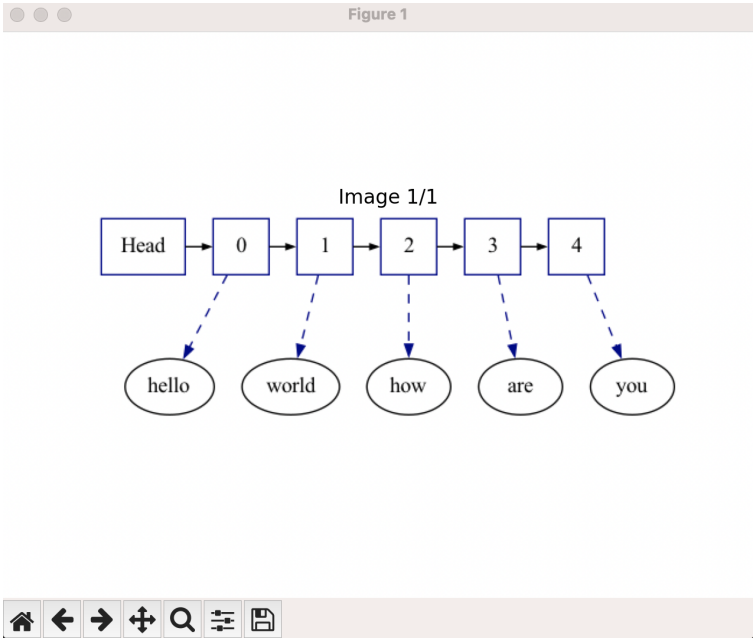


Figure 2: DataDebug graphical user interface

2.2 Graph Design

An equally important segment of DataDebug was determining how to store the data structure. Quickly, pygraphviz shined as the best tool to accomplish this task due to it's robustness and rendering capabilities. Pygraphviz supported basic graph creation and image processing so early development was quick for trees. It was mainly a matter of integrating pygraphviz and matplotlib properly to set up the foundations.

However, pygraphviz lacked in it's ability to store and display lists, especially ones that would be dynamically updating throughout run-time. Having lists as a feature would be critical due to it's power of being a stack or queue, and allowing for visualization of key algorithms on a tree. Initially, we tried to wrap the default list's magic methods which deal with state changes (__setitem__, __delitem__, etc), however, we were unable to find an accurate implementation that would capture all the updates and maintain high flexibility for users

to determine how objects are rendered. Ultimately, we decided not to track list updates in the graph during regular execution, but rather, convert all logged lists into sub-graphs and merge them into the main graph during rendering. This meant that for an O(1) list update to be visualized, O(n) work needed to be done, where n is the length of the list.

Another issue that arose from list rendering was object collision. That is, if we had z in a graph and wanted to include the same object z in a list as well, unexpected behavior arose from the nodes and pointers. To address this issue, we had to analyze how pygraphviz was mapping keys to nodes. Ultimately, we had to wrap all of the items in the list inside of another object with a *str* method returning a unique identifier. This allowed us to maintain different nodes and edges for the same object value.

3 Results

This section presents the key findings of the DataDebug tool's implementation and its results in improving the understanding of data structures through visualization. The results are separated into two sections, providing insights into the tool's practical application and the impact on users.

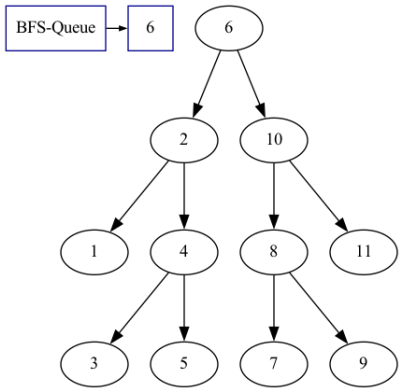
3.1 Use Cases

The most central use case for DataDebug was visualizing graph exploration algorithms like breadth-first-search. In Figure 3, we have outlined the beginning steps of a BFS on a small graph. This was accomplished by first registering all the graph edges into the logger. Then, our custom List class was used as a queue to keep track of nodes to explore. Ultimately, this visualization allows users to quickly notice that a breadth-first-search is not properly occurring, but rather a depth-first-search has been implemented. The library is very flexible and can be adapted to display weights on edges and demonstrate more complex algorithms like Dijkstra's or A* search.

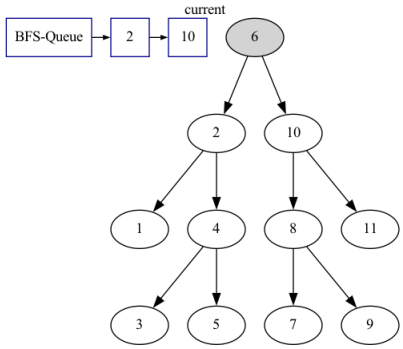
The library also has high flexibility for representing items in list-like structures such as stacks, queues, linked lists, and regular lists. As previously seen, stacks and queues can be easily rendered through our List class. We've also tested visualizing sorting algorithms on lists as well. As seen in Appendix Figure 5, DataDebug can help users conceptualize pointers and the movement of nodes during the merger of two sorted linked lists.

3.2 User Study

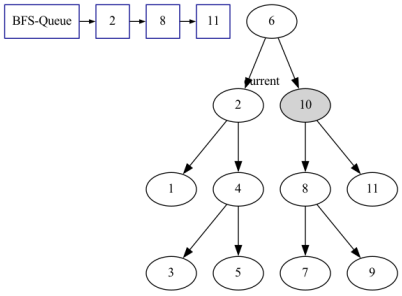
Finally, to validate the effectiveness of this library, user tests were conducted. A small number of Yale Computer Science students with at least some experience with data structures were surveyed for this study. Initially, the subjects filled out basic pre-study questions regarding their previous experience with data structures and debugging. Following, they were introduced to the DataDebug library through a light tutorial of all the key classes and methods. The two code scenarios for this study involved merging listed lists and ordering tree traversals. First, the subjects were prompted to debug one of the subsequent scenarios using simply in-line printing. Thereafter, they completed the debugging challenge on the other

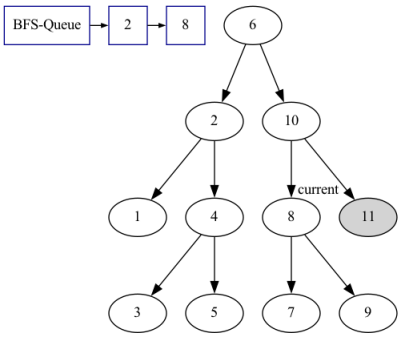


(a) Initializing of graph search



(b) Visit first node and add neighbors






```
while bfs_stack:
    current :int = bfs_stack.pop() #BUG
    visit_order.append(current)

    for neighbour in self.graph[current]:
        if neighbour not in visited:
            bfs_stack.append(neighbour)

    logger.add_pointer(current, "current",
       display_once=True)
```

Figure 3: Depiction of DataDebug on tree traversal

scenario using the DataDebug visual logger. Finally, they completed a variety of post-study questions prompting subjects to compare and contrast their experiences with and without the new logger tool. The results from the study are as follows:

- **Background**: During the preliminary study survey, participants expressed a common theme of finding debugging a challenging experience. Some described it as "frustrating" and others said it was "time-consuming". Nonetheless, some highlighted that it was an essential process for programming. The pool of participants contained an approximately even range of people who scaled themselves between 3 and 8 on a scale of how comfortable they are with data structures. Generally, participants claimed they somewhat rarely use data structures in their regular programming tasks.

- **Comparison**: After the 2 tests, participants were asked questions regarding both their experiences using printing versus the visual debugger. All participants found the experience using the visualization library better than traditional in-line printing. They provided various reasons why. One participant said "tree structures aren't necessarily intuitive" so the picture helped them picture the traversal easier. Similarly, another participant said "the main problem with debugging data structures...is visualizing them" and that "printing is hard". A common theme in responses was how the visualization made bugs more obvious. This proved valuable in terms of saving time as one participant put it: "[I could] quickly see the logical error/where the algorithm went wrong". Overall, the subjects found it more enjoyable to use the visual library.

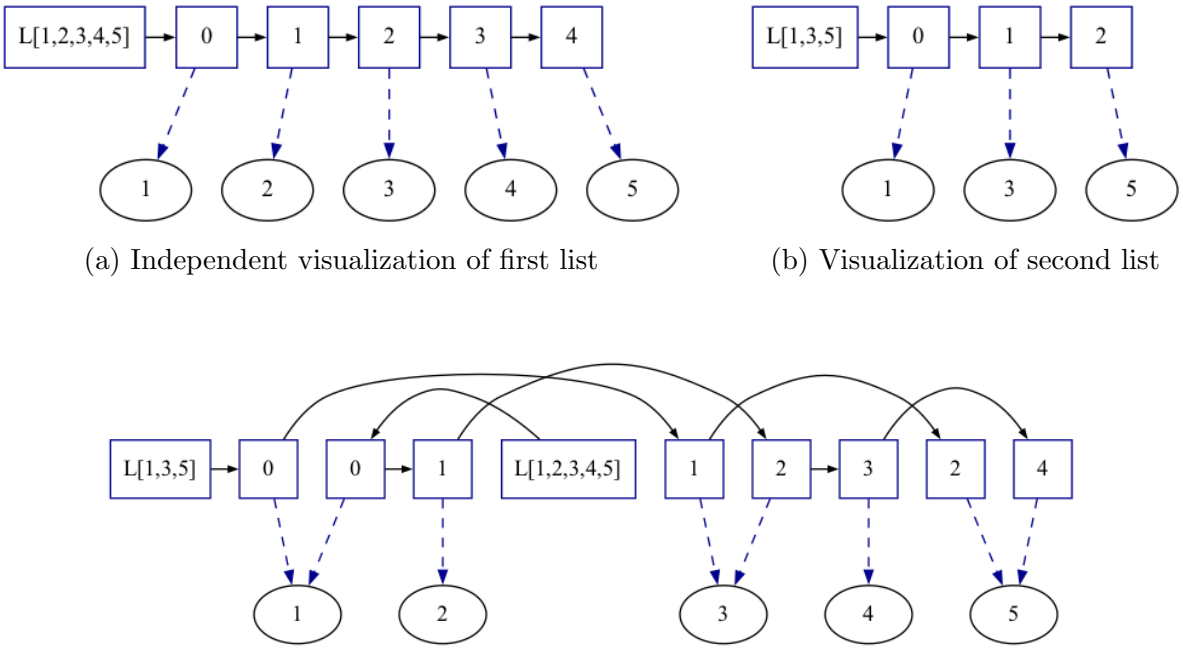
- **Feedback**: There was limited brief feedback from users. Nonetheless, a vast majority of participants suggested that a pause feature would be useful. Currently, the library configurations cause the next snapshot to be displayed after x seconds. During the study, participants were often still analyzing an image and would need to back-track when an automatic update occurred. This made it more difficult for them to analyze the images. A few participants also noted that they liked the label feature because it "made it very clear where things were flowing". In terms of more niche comments, one subject suggested the implementation that allowed for users to click into a node for a more in-depth view, perhaps revealing all of the objects attributes instead of just its simplified string form. Another participant suggested to update the GUI so that the arrows on screen behaved like the keyboard arrows.

Overall, the user study was successful at highlighting the strengths and weaknesses of DataDebug. Through this study, we were able to better understand programmers' sentiments towards debugging and evaluate methods to improve their experiences. On a final note, this user study was small, containing only 8 participants. Nonetheless, these results provide us direction for future work and incentives to conduct larger studies to better understand the target population.

4 Future Work

This project encounters certain constraints inherent to its design and the technologies it employs. These limitations are outlined as follows:

- **Rendering**: The underlying rendering dot notation of pygraphviz directs the graphical representation uni-directionally (either from left to right or top to bottom). This feature can lead to non-intuitive visual results, particularly when the elements of a data structure, such as a binary tree, are not input in a sequence that aligns with the uni-directional rendering. Since pygraphviz abstracts the positioning algorithm, direct manipulation of the node placement was unsuccessful. Furthermore, the addition of nodes or labels may trigger an unexpected rearrangement of the graph making it difficult to track changes throughout snapshots. As pictured in Figure 4, the structure may choose unnatural node placement. A potential solution to these issues could involve developing one's own visualization library that allows us developers to have more control in regards to node placement, however, this is an incredibly time intensive task.



(a) Independent visualization of first list

(b) Visualization of second list

(c) Unintuitive display of both lists combined

Figure 4: Unexpected and uncontrollable rendering results

- **Collisions**: Currently, the system creates only 1 instance of an object which renders to the same string value. In other words, if you wanted to render two lists, one with the original objects and the second with deep-copies, the system would only display one instance of the list's values because they would have the same string representation. A potential solution to differentiate between elements with identical values could involve encapsulating each entity within a distance object furnished with a universally unique identifier (UUID). This was successfully done to allow different lists to render the same index value as seen in Figure 5, so the same technique could be applied there as well.

- **Scalability and Size Limitations:** DataDebug demonstrates good performance with small-scale structures. However, as list lengths increase into the hundreds and as

graphs grow bigger, it becomes harder to view everything. The scale decreases and makes nodes really small. This appears to be a problem that would plague every sort of visualization interface.

5 Conclusion

This thesis explores the potential of visualization tools in enhancing both the understanding of data structures and the debugging experience. This is accomplished through the development and testing of the DataDebug library. The project highlights the significant benefits visualization tools have to offer, particularly in educational contexts where new learners struggle to grasp these abstract concepts. The DataDebug library is able to bridge this crucial gap between theoretical knowledge and practical application by providing real-time visuals on data structures. It allows programmers to see beyond the code and into the dynamic behavior of their data. The user study conducted as part of this thesis further supports the idea that visualization methods can enhance learning and debugging efficiency by providing unique insights into data manipulation and algorithm behaviors.

While the DataDebug library has shown promising results, the feedback and findings from this thesis also highlight areas for further development. Notably, there are many challenges when scaling the visualizations for larger and more complex structures. There is also plenty of room for improvement in the user interface: both API and GUI. These enhancements will not only make the library more robust and versatile but also extend its applicability in different scenarios and different kinds of users. Thus, extended research and development is necessary to continue advancement in the field. This project lays a step towards a more visually integrated approach in programming education and practice, advocating for a shift from traditional text printing methods to more engaging visual solutions.

6 Acknowledgements

I extend my deepest gratitude to all those who have made the completion of this thesis possible. I would like to thank my advisor, James Glenn, for his invaluable guidance and support throughout this project. He helped me understand the preexisting tools, guided me through the Yale University Institutional Review Board for my study, and provided valuable feedback. Special thanks goes to my peers in the Computer Science department, whose camaraderie and intellectual curiosity have been a constant source of motivation.

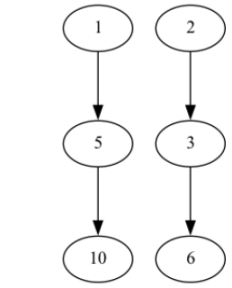
References

[1] jGRASP Group, Auburn University. (2009, April 23). JGRASP on WindowsXP. Wikipedia. https://upload.wikimedia.org/wikipedia/commons/d/d9/JGRASP_screenshot.png

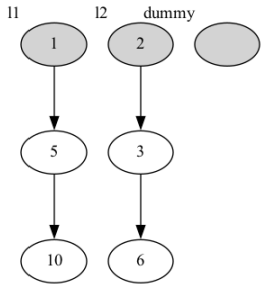
[2] Tao Chen and T. Sobh, "A tool for data structure visualization and user-defined algorithm animation," *31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No.01CH37193)*, Reno, NV, USA, 2001, pp. TID-2, doi: 10.1109/FIE.2001.963845

[3] Osman, Waleed Ibrahim, and Mudawi M. Elmusharaf. "Effectiveness of combining algorithm and program animation: A case study with data structure course." Issues in Informing Science and Information Technology 11 (2014): 155-168.

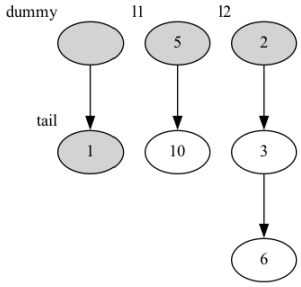
A Appendix



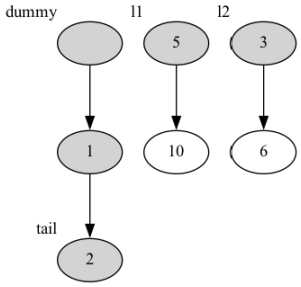
(a) Initializing the linked lists



(b) Creating new merged pointer



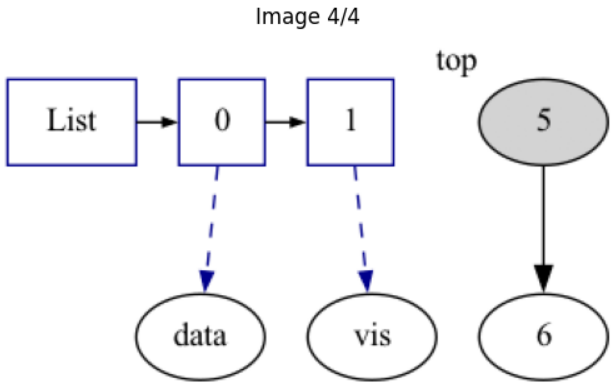
(c) Taking lowest value 1 from *l1*



(d) Taking lowest value 2 from *l2*

```
while l1 and l2:
    logger.add_pointers(
        [tail, dummy, l2, l1],
        ["tail", "dummy", "l2", "l1"],
    )
    if l1.value < l2.value:
        tail.next = l1
        logger.add_edge(tail, l1, display=False)
        l1 = l1.next
    else:
        tail.next = l2
        logger.add_edge(tail, l2, display=False)
        l2 = l2.next
    tail = tail.next
    // Graphically cut off the tail from the l1/l2
        head
    logger.delete_edge(tail, tail.next, display=False)
```

Figure 5: Merge two sorted linked lists



(a) Initializing of graph search

```
# create node
logger.create_node(5)
logger.create_node(6)

# create edge
logger.add_edge(5, 6)

# create label
logger.add_pointer(5, "top", display_once=False)

# create list
l = List(["data", "vis"], display_edges=True,
    display_index=True, label="List")

# visualize (useful for updating list visuals)
logger.visualize()
```

Figure 6: Depiction of DataDebug basic visualization operations

B DataDebug API Documentation

`DataDebug` is a Python package designed for visual debugging of data structures. Leveraging `matplotlib` and `pygraphviz`, it provides an intuitive interface for visualizing and interacting with data structures as graphs. To use this library, import the *logger* from src/datadebug and start debugging your data structures.

B.1 GUI Controls

B.1.1 `Left arrow: next`

Advances to previous snapshot of the underlying data structures. Pauses GUI updates for default of 3 seconds.

B.1.2 `Right arrow: previous`

Advances to next pre-recorded snapshot of the underlying data structures. Pauses GUI updates for default of 3 seconds.

B.1.3 `Up arrow: speed up`

Cancels the GUI pause, allowing for quick rendering of the next state.

B.2 Logger Class Methods

B.2.1 `create_node`

Creates a new node in the graph.

- **Parameters:**
- `node`: Identifier for the new node.
- `display` (optional): Whether to display the graph after adding the node. Defaults to `True`.

B.2.2 `delete_node`

Deletes a node from the graph.

- **Parameters:**
- `node`: Identifier for the node to delete.
- `display` (optional): Whether to display the graph after deleting the node. Defaults to `True`.

B.2.3 `add_edge`

Adds an edge between two nodes in the graph.

- **Parameters:**
- `node_from`: Identifier for the source node.
- `node_to`: Identifier for the destination node.
- `display` (optional): Whether to display the graph after adding the edge. Defaults to `True`.

B.2.4 `delete_edge`

Deletes an edge between two nodes in the graph.

- **Parameters:**
- `node_from`: Identifier for the source node.
- `node_to`: Identifier for the destination node.
- `display` (optional): Whether to display the graph after deleting the edge. Defaults to `True`.

B.2.5 `clear_edges`

Clears all edges connected to a specific node.

- **Parameters:**
- `node`: Identifier for the node whose edges will be cleared.
- `display` (optional): Whether to display the graph after clearing the edges. Defaults to `True`.

B.2.6 `customize_node`

Customizes the appearance of a node.

- **Parameters:**
- `node`: Identifier for the node to customize.
- `display_once` (optional): Whether to apply customization for the current state only. Defaults to `True`.
- `**kwargs`: Additional styling parameters (e.g., color, shape).

B.2.7 `customize_edge`

Customizes the appearance of an edge.

- **Parameters:**
- `node_from`: Identifier for the source node.
- `node_to`: Identifier for the destination node.
- `display` (optional): Whether to display the graph after customizing the edge. Defaults to `False`.
- `**kwargs`: Additional styling parameters (e.g., color, label).

B.2.8 `add_pointer`

Adds a pointer label to a node.

- **Parameters:**
- `node`: The node to add the pointer to.
- `pointer_name`: The name of the pointer (e.g., head, tail).
- `display_once` (optional): Whether to display the pointer for the current state only. Defaults to `True`.
- `clear_repeated` (optional): Whether to clear repeated pointers before adding the new one. Defaults to `True`.

B.2.9 `add_pointers`

Adds multiple pointer labels to nodes at the same time.

- **Parameters:**
- `nodes`: The nodes to add pointers to.
- `pointers`: The names of the respective pointers (e.g., head, tail).
- `display_once` (optional): Whether to display the pointers for the current state only. Defaults to `True`.
- `clear_repeated` (optional): Whether to clear repeated pointers before adding new ones. Defaults to `True`.

B.2.10 `visualize`

Renders and displays the current graph.

B.2.11 `snapshot`

Takes a snapshot of the current graph state. Saves in specified folder (default: ./log_snapshot)

B.3 List

To directly visualize a list, the List class can be leveraged by users. It automatically connects to the underlying graph and captures all changes onto the graph. However, the user must manually direct the library to re-visualize the data structure when updates to the List want to be rendered & seen on the interface.

B.3.1 `__init__`

Creates a list using the data and records structure in graph.

- **Parameters:**
- `data`: The data to initialize the list.
- `label`: The name of the list pointer (e.g., head, tail).

- `display_edges` (optional): Whether to display edges from the nodes in the list to pre-existing nodes in the graph. Defaults to `True`.
- `display_index` (optional): Whether to display the index of the items in the list or their true values. Defaults to `False`.

This documentation aims to provide a clear overview of how to interact with the `DataDebug` package. For further details, including examples and advanced usage, please refer to the package's official documentation or source code.