EXPLORING THE USE OF SOFTWARE VISUALIZATION IN COMPUTER SCIENCE

EDUCATION AND DEVELOPING A SYSTEM TO FACILITATE THE

CONSTRUCTION OF ALGORITHM VISUALIZATIONS

WITH JSAV AND WEBASSEMBLY

A Project

Presented

to the Faculty of

California State University, Chico

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

© Shelley Wong 2021

Spring 2021
EXPLORING THE USE OF SOFTWARE VISUALIZATION IN COMPUTER SCIENCE
EDUCATION AND DEVELOPING A SYSTEM TO FACILITATE THE
CONSTRUCTION OF ALGORITHM VISUALIZATIONS
WITH JSAV AND WEBASSEMBLY

A Project

by

Shelley Wong

Spring 2021

APPROVED BY THE DEAN OF GRADUATE STUDIES:

Sharon Barrios, Ph.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

Tyson Henry, Ph.D.
Graduate Coordinator

Kevin Buffardi, Ph.D., Chair

Elena Harris, Ph.D.
PUBLICATION RIGHTS

No portion of this thesis may be reprinted or reproduced in any manner unacceptable to the usual copyright restrictions without the written permission of the author.
ACKNOWLEDGEMENTS

I want to thank my committee chair, Dr. Kevin Buffardi, for inspiring me to expand my project concept and consider its academic applications. Thank you for introducing me to the frustrations and joys of LaTeX, asking questions that always led to an improvement in my work, and giving me support and freedom to explore my ideas.

I would also like to thank the faculty and staff of the Chico State Computer Science Department for always being there with assistance and encouragement whenever I needed it. Thank you to Dr. Elena Harris for providing me with my first computer science research experience, pushing me to improve, and always being willing to share your insight. Thank you to Dr. Judy Challinger for giving me guidance on approaching literature review, helping me stay accountable, and giving me space to try new things. Thank you to Dr. Tyson Henry for offering me the opportunity to teach my first classes, keeping me on track, and supporting me throughout the process.

I also want to acknowledge the professors and teachers throughout my academic career who fostered curiosity, passed on their enthusiasm for learning, and provided opportunities for active engagement inside and outside the classroom. Those experiences inspired me, gave me perspective, and made me appreciate a good challenge.

I would also like to thank my loving and understanding family. Even though most of our conversations happened across a video chat, you kept me grounded and gave me the chance to take breaks, relax, and talk to other human beings about things not related to school and work. Last but not least, thank you to my husband Bryan for sticking with me through the late nights and early mornings, offering occasional technological expertise and insight, being my rubber duck.
debugging partner, and helping me find balance, maintain perspective, and make it through a few exhausting yet rewarding years.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publication Rights</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract</td>
<td>x</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>I. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Background</td>
<td>1</td>
</tr>
<tr>
<td>Purpose</td>
<td>3</td>
</tr>
<tr>
<td>Problem Statement</td>
<td>4</td>
</tr>
<tr>
<td>Organization of the Study</td>
<td>7</td>
</tr>
<tr>
<td>II. Literature Review</td>
<td>8</td>
</tr>
<tr>
<td>History of Software Visualization</td>
<td>8</td>
</tr>
<tr>
<td>Examples of AVs</td>
<td>9</td>
</tr>
<tr>
<td>Examples of PVs</td>
<td>10</td>
</tr>
<tr>
<td>Other Software Visualization Systems</td>
<td>11</td>
</tr>
<tr>
<td>Online Textbooks</td>
<td>12</td>
</tr>
<tr>
<td>JSAV</td>
<td>14</td>
</tr>
<tr>
<td>WebAssembly</td>
<td>15</td>
</tr>
<tr>
<td>III. Methodology</td>
<td>17</td>
</tr>
<tr>
<td>Overview</td>
<td>17</td>
</tr>
<tr>
<td>Required Tools</td>
<td>18</td>
</tr>
<tr>
<td>Project Setup</td>
<td>19</td>
</tr>
<tr>
<td>Building the Prototype</td>
<td>20</td>
</tr>
<tr>
<td>Compiling the Code</td>
<td>23</td>
</tr>
<tr>
<td>Array Starter Code</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>WebAssembly Linear Memory</td>
<td>25</td>
</tr>
<tr>
<td>Linked List Starter Code</td>
<td>26</td>
</tr>
<tr>
<td>Binary Tree Starter Code</td>
<td>28</td>
</tr>
<tr>
<td>Providing Documentation and Instructions</td>
<td>30</td>
</tr>
<tr>
<td>Error Checking Overview</td>
<td>31</td>
</tr>
<tr>
<td>Array Error Checking</td>
<td>32</td>
</tr>
<tr>
<td>Linked List Error Checking</td>
<td>36</td>
</tr>
<tr>
<td>Binary Tree Error Checking</td>
<td>43</td>
</tr>
<tr>
<td>Recursion and Infinite Loop Error Checking</td>
<td>50</td>
</tr>
<tr>
<td>Compilation Errors</td>
<td>51</td>
</tr>
<tr>
<td>IV. Discussion, Conclusions, and Recommendations</td>
<td>52</td>
</tr>
<tr>
<td>Limitations</td>
<td>52</td>
</tr>
<tr>
<td>Conclusions</td>
<td>54</td>
</tr>
<tr>
<td>Recommendations and Future Research</td>
<td>56</td>
</tr>
<tr>
<td>Summary</td>
<td>57</td>
</tr>
<tr>
<td>References</td>
<td>60</td>
</tr>
<tr>
<td>Appendix A: Example of an Array Visualizer Slideshow</td>
<td>65</td>
</tr>
<tr>
<td>Appendix B: Example of a Linked List Visualizer Slideshow</td>
<td>68</td>
</tr>
<tr>
<td>Appendix C: Example of a Binary Tree Visualizer Slideshow</td>
<td>72</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. BoxAndArrow Array Visualizer Error Checking</td>
<td>33</td>
</tr>
<tr>
<td>2. BoxAndArrow Linked List Visualizer Error Checking</td>
<td>37</td>
</tr>
<tr>
<td>3. BoxAndArrow Binary Tree Visualizer Error Checking</td>
<td>44</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Array Visualizer GET and POST Requests</td>
<td>22</td>
</tr>
<tr>
<td>2. Linked List Operations on a List Instance</td>
<td>27</td>
</tr>
<tr>
<td>3. Example of a BoxAndArrow Binary Search Tree Visualization</td>
<td>29</td>
</tr>
<tr>
<td>4. Example of Uninitialized Array Iterators Error Response</td>
<td>33</td>
</tr>
<tr>
<td>5. Example of Array Out-Of-Bounds Error Response</td>
<td>34</td>
</tr>
<tr>
<td>6. Example of Array Missing Edge Case Error Response</td>
<td>36</td>
</tr>
<tr>
<td>7. Example of Error Response for Uninitialized Traversal Variable</td>
<td>38</td>
</tr>
<tr>
<td>8. Example of Error Response for Trying to Remove Head of Empty List</td>
<td>40</td>
</tr>
<tr>
<td>9. Example of Error Response for Mishandled Link Between Nodes</td>
<td>41</td>
</tr>
<tr>
<td>10. Example of Slideshow Error Message for Case When Target Not Found</td>
<td>42</td>
</tr>
<tr>
<td>11. Example of Message and Standard Output Display for Failure to Reset Head</td>
<td>43</td>
</tr>
<tr>
<td>12. Example of Error Message for Inaccessible Memory Location in Tree</td>
<td>46</td>
</tr>
<tr>
<td>13. Example of Tree Out-Of-Bounds Error Shown in Slideshow</td>
<td>48</td>
</tr>
<tr>
<td>14. Example of Slideshow Displaying Inaccurate Results: Traversal Mistake</td>
<td>49</td>
</tr>
<tr>
<td>15. Example of Slideshow Displaying Inaccurate Results: Missing Base Case</td>
<td>50</td>
</tr>
<tr>
<td>16. Example Slideshow Sequence: Determine if Array is Sorted</td>
<td>65</td>
</tr>
<tr>
<td>17. Example Slideshow Sequence: Remove Elements From a Linked List</td>
<td>68</td>
</tr>
<tr>
<td>18. Example Slideshow Sequence: Insert Elements Into a Binary Tree</td>
<td>72</td>
</tr>
</tbody>
</table>
ABSTRACT

EXPLORING THE USE OF SOFTWARE VISUALIZATION IN COMPUTER SCIENCE EDUCATION AND DEVELOPING A SYSTEM TO FACILITATE THE CONSTRUCTION OF ALGORITHM VISUALIZATIONS WITH JSAV AND WEBASSEMBLY

by

© Shelley Wong

Master of Science in Computer Science
California State University, Chico
Spring 2021

Many novice programmers struggle to understand data structures and algorithms, a core component of computer science curricula. While computer science education researchers have long considered software visualizations as a critical tool to aid student learning, adopting these tools in the classroom has been inconsistent. It is also vital to consider that not all visualization tools are equal. Older algorithm visualization systems are often incompatible with newer technological standards. Animations that require very little interaction can look impressive without providing significant educational benefits. This paper discusses a prototype application that works with WebAssembly and the JavaScript Algorithm Visualization library, allowing students to write C++ programs that map to a data structure visualization viewable on a personal computer through a web browser. WebAssembly is a new standard that allows languages like
C++ to run on the Web. The JavaScript Algorithm Visualization library is an established open source library used in digital computer science educational material. The result of this research is an open source repository that includes starter code and instructions for implementing arrays, linked lists, and binary trees. As a learning tool, this application provides error-checking features to help students identify mistakes in their code and makes it easy to test programs with various inputs and commands. This project intends to complement, rather than replace, other visualization tools and educational materials. It aims to create a foundation for students to have an active role in constructing, changing, and presenting algorithm visualizations that connect to programs they have written themselves.
I. INTRODUCTION

A. Background

Data Structures and Algorithms (DSA) are a central component of computer science curricula in baccalaureate-level academic degree programs [1]. For students to succeed in computer science, they need a solid comprehension of core computing concepts, the skills to program, and a willingness to put forth the effort necessary to solve complex problems. Yet many novice computer science students have difficulty understanding the data structures and algorithms that form the basis of computing disciplines [2]. According to the Higher Education Statistics Agency, 9.8% of computer science undergraduates drop out before completing their degree, the highest dropout rate out of all college majors [3]. Students have various reasons for failing or dropping out of computer science courses, and sometimes it has nothing to do with how the course material gets presented. However, students who struggle yet are willing to put in the effort may benefit from software visualizations, especially those that incorporate interaction and focus on engagement. Over the years, researchers have tested various software visualization tools in the classroom. In some cases, there were no notable differences in outcomes; however, evidence suggests that visualization tools that required active participation by the student have the most significant positive impact on student learning [4], [5].

Researchers in this field often base their conclusions on student performance for tests and assignments, log data on usage, or student opinion surveys. In an introductory computer science course, visualizations could be integrated into one section of the class and compared against a similar control group taught without the visualizations. Unfortunately, this type of data does not necessarily provide conclusive solutions. Collecting sufficient quantitative data to determine the real impact of algorithm visualizations (AVs) and program visualizations (PVs) for pedagogical purposes is challenging, and opinion surveys can be subjective. Still, student responses tend to be favorable, and reports show that students will spend time using the visualizations if given the opportunity. Section II discusses examples of this kind of research.

Active learning is a concept closely related to constructivism [6], a theory of learning that claims students achieve the most significant educational gains through constructing knowl-
edge and building on concepts they have developed from previous learning. The idea is that students gain knowledge through direct experience, which differs from the traditional philosophy that students learn classroom material by listening to lectures and reading textbooks. They also need a foundational understanding of how data structures and algorithms operate, which algorithm visualizations can help explain. The engagement taxonomy defined by the Working Group on "Improving the Educational Impact of Algorithm Visualization" [7] reflects similar concepts to those described in constructivism. The lowest engagement levels (No Viewing and Viewing) are passive forms of engagement that may not offer any academic advantages. Viewing that does not require active attention may even be detrimental. When an individual views an algorithm animation but no interaction is necessary, viewers are not encouraged to pause and focus on how or why the algorithm works.

Naps et al. hypothesize that the higher levels of engagement, i.e., Responding, Changing, Constructing, and Presenting, can all be combined with Viewing to increase positive learning outcomes. Sorva et al. [8] added a dimension to the engagement taxonomy that emphasized both direct engagement with the visual as well as indirect engagement through Content Ownership. Their idea was that learners who had more involvement with creating and using a visualization would ultimately walk away with a stronger understanding of the material because they had spent more time and cognitive effort on it.

The overwhelming majority of computer science educators have positive views on the use of AVs for pedagogical purposes [7], [9], yet AVs are still rarely integrated into courses. Part of the reason for this is the difficulties educators have trying to find quality educational content online [10]. Efforts to make this process easier have focused on creating agreed-upon standards, using some form of community input, and improving interoperability between systems; however, it is an ongoing process. Crowdsourcing online content is most successful when a large set of people can dedicate time to developing it. Despite the high level of interest, many educators are very limited in the time and resources they can spend actively contributing to online communities [9].
B. Purpose

Over the years, researchers have created software visualization systems that focus on various computing-related topics and offer different levels of engagement [4]. Some widely used AV systems provide pseudocode to connect visualizations to algorithms [11], [12]. Still, others only show what happens in the algorithm without corresponding code [13]. Some PV systems work with user-written C++ code but require that users work within a specific web application [14]. Although many AV and PV tools exist, this research aims to include both AV and PV elements, offer unique ways to increase user interaction, and incorporate a new web standard. This project utilizes WebAssembly, which allows C++ code written in a native environment to execute on the web. It also helps users create software visualizations that change based on user-written C++ programs and are viewable from a web browser.

The traditional purpose of an AV is to provide a high-level explanation of how a data structure or algorithm works. A proper AV graphically represents the different states of a data structure or algorithm, along with transitions between each state [5]. New AV project development often focuses on expanding topic coverage [12]. Given that AV helps explain algorithms, these tools usually are not built from student code; rather, they aid students in understanding algorithms in the first place. The traditional purpose of a PV is to describe low-level program behavior and show what occurs in memory as each line of code executes [8]. PV incorporates user-written code; however, these tools may only work with small code segments or require users to work within a specific web application or integrated development environment [15], [16]. PV tools generally focus on interpreting how a specific program works and creating visual feedback to help debug code. In contrast, AV tools demonstrate the bigger picture of how algorithms naturally behave [17]. Section II discusses some examples of AV and PV. This research project helps students and instructors develop AV-type visualizations that function based on student-written C++ programs. A key focus was to provide users with increased content ownership. Users must write code that determines how the visualization functions and also create tests that ensure their program works in all cases.

New developments in software visualization tend to come about due to changes in technology. The first algorithm animations can be traced back to the 1980s, before the advent
of the World Wide Web [4], [5]. From the mid-1990s to the mid-2000s, practically all AVs used Java and came in the form of Java-based applets and standalone systems. However, these AVs have compatibility issues with various browsers and mobile devices, and interoperability with web platforms has become an essential feature for software visualization systems [18]. Availability on the web improves accessibility and ease of use [15]. Most recent visualization systems have utilized JavaScript, HTML5, and CSS, the primary programming languages of the web that enjoy support by all major web browsers [18].

JavaScript, which allows for dynamic and interactive features, continues to be one of the most widely used programming languages in the world [19]; however, it also has several flaws. In particular, JavaScript’s security features are limited, and its performance can be noticeably inconsistent [20].

In recent years, developers from Google, Microsoft, Mozilla, and Apple joined together to create a new standard that could provide a faster and safer alternative to JavaScript [20]. They developed WebAssembly, a portable, low-level binary instruction format that can efficiently compile and deploy code written in high-level languages (like C, C++, and Rust) to the web. Thus, in addition to offering speed and security benefits, it was also size efficient and independent of any specific language, hardware, or platform [20]. Much of the research on WebAssembly has focused on performance, as WebAssembly provides an improvement in time efficiency that is particularly beneficial for computationally intensive procedures [21], [22], [23]. While software visualizations for educational purposes are not necessarily computationally intensive, it is worthwhile to explore the potential opportunities offered by utilizing WebAssembly in an academic setting. Many university computer science departments use C++ to teach programming courses, so creating a tool that maps C++ programs to a visualization viewable in a browser could have widespread benefits.

C. Problem Statement

Determining the best way to help beginning computer science students better understand data structures and algorithms is a persistent, overarching question of computer science educators. An individual project cannot conclusively determine the best solution for all learners, but it can explore well-regarded theories about how students develop and retain knowledge. This
project explores two research problems, aiming to provide a practical application that incorporates these concepts and promotes increased learner engagement.

1) Can a library built for algorithm visualization development in JavaScript be adapted to work with C++ code compiled to WebAssembly?

2) Can this WebAssembly extension to an existing algorithm visualization library provide users with the ability to construct visualizations from C++ programs they have created themselves?

For Research Problem 1, this project offers a prototype web application that combines WebAssembly and JSAV. It includes C++ programs for arrays, linked lists, and binary trees that have been compiled from C++ to WebAssembly using Emscripten [24]. The implementation includes JavaScript driver code that incorporates the results of the C++ programs into the JSAV-based visualizations in the browser.

C++ consistently ranks as one of the most popular programming languages [25], and many institutions use it to teach CS1, CS2, and DSA courses [26]. Despite other programming languages gaining popularity, there are several reasons why C++ is still widely used in fundamental computer science courses. C++ is a compiled language, which is beneficial for performance [27]. It is strongly typed, which allows programmers to have a lot of control. It is also object-oriented and has solid library support, attributes that are useful for large and complex programs. However, C++ is not as portable as many other languages [28]. Most C++ courses rely on console output, because although various options exist for incorporating C++ into a graphical environment [29], [30], they often include a significant learning curve. While various tools exist for DSA visualization, the examples encountered during research did not provide options for users to compile C++ code and create a web-based visualization that works with the user-written program.

The Constructing level of engagement, described in the Engagement Taxonomy developed by Naps et al. [7], offers the best opportunities for helping students learn data structures
and algorithms. For Research Problem 2, this project provides a foundation for students and instructors to construct data structure visualizations that utilize their native C++ code. Extra steps are required to map C++ to code that is executable on the web, but users do not need to worry about these steps because they are taken care of or accompanied by detailed instructions. Programming exercises provide direction, and users can focus on the algorithms and data structures. Active learning is an integral component of the process. Users are directly involved with developing the visualization, creating tests, and examining the results.

Most AV systems offer some levels of engagement that go beyond viewing. Still, most of these systems rely on engagement via responses (e.g., answering questions related to the visualization) and changing (e.g., modifying the input, adjusting the speed, returning to a previous step, or pausing the animation) [18], [31], [32], [33], [11], [12]. This project promotes many levels of engagement, in addition to viewing. Users must construct algorithms that match a corresponding visualization. If there is an error in the implementation, users must respond by updating the program. The project offers many opportunities to change and test the visualization, with simple methods for updating the input and stepping back and forth through the slideshow. When users are confident in the programs they have written, the web-based format makes it easy to present and share their work.

Many online educational resources exist, but the quality is not always consistent, and content has often been redundant [10]. This project’s primary purpose is to address a feature not traditionally built into AV systems, namely, a foundation for learners to construct and present visualizations they have created themselves. Instead of creating an entirely new system from scratch, this project intends to build on the work of others who have already spent much time and effort improving the quality and availability of existing AV systems. In particular, this project expands the potential applications of a widely used algorithm visualization library, employs a new standard that allows a growing number of programming languages to be compiled and executed on the web, and explores novel uses for each in the process. There are many potential solutions to helping students understand core computer science concepts; however, as online learning becomes increasingly common, active engagement with web-based AVs seems like a particularly relevant field to explore.
This project occupies a space between traditional AVs and PVs. AVs aim to explain algorithms and data structures accurately, and PVs focus on helping debug code line-by-line. In this project, the end product looks like an AV but can also assist in error detection. When there is a bug in the student’s code, the visualization will not behave as expected for every set of inputs. Users can easily compare the differences between the expected and actual results in the visualization. This project cannot identify the specific line where an error exists, but it can identify the function that contains the bug.

Often, the issues students struggle with the most are not compiler errors, but rather edge cases that the student does not realize while reading through the logic of their code [34]. When students believe they know how an algorithm is supposed to work, they are less likely to recognize a mistake in their implementation. They can easily make assumptions that overlook the error while creating a whiteboard drawing or mentally walking through their code. In this project, the algorithm visualization will follow the student’s compiled code precisely as instructed, so the oversight will be easier to spot.

D. Organization of the Study

Chapter I consists of an introduction to the project, an overview of its purpose, and a problem statement. Chapter II provides a review of related works in the area of software visualization in computer science education. Chapter III covers the methods employed to build a system that students can use to create visualizations from code that they have written. It covers the project development process and experiences writing documentation and providing error detection. Chapter IV offers a discussion on the system’s benefits and limitations, considers future research ideas, and summarizes this investigation’s results.
II. LITERATURE REVIEW

A significant amount of literature focuses on software visualization, and this section starts with a discussion on the background of these tools and their use in educational settings. Cordova compiled more comprehensive examinations on the history of AVs, and the state of the field in 1999 [35], Naps et al. in 2002 [7], and Shaffer et al. in 2007 [4] and 2010 [5]. In 2013, Sorva et al. [8] published a thorough history of available PVs. The second and third subsections will discuss some of the many AV and PV systems created since then, developed with knowledge gained from the previous systems. The fourth subsection explores various alternative uses of visualization for computer science education, and the fifth examines eTextbooks that utilize AVs alongside other pedagogical material. The last two subsections explore the JavaScript Algorithm Visualization Library and WebAssembly, which have been essential to this project’s development.

A. History of Software Visualization

The idea of using Algorithm Visualization for pedagogical purposes has been around since the 1980s [4]. Most of the early AVs were built for proprietary operating systems and thus had limited availability [2]. In the 1990s, developers primarily created AVs as Java standalone applications or web applets. While beneficial for increased portability at the time, Java is no longer the best option for development on the web [2]. Today, virtually all AV systems are built with JavaScript, to be available in any major browser [11], [12], [36], [37].

AV systems are not a substitute for other educational material, but they can be a powerful supplement to static presentations in textbooks and verbal descriptions from lectures [4]. In the past, there have been issues with the distribution of AVs, topic coverage, and overall AV quality [5]. While various AV and PV systems have existed over the years, many never grew past the initial prototype stage, and most are no longer available [8]. Currently, a few of the most comprehensive visualization systems accessible on the web include the AV tools Data Structure Visualizations [11], VisuAlgo [12], and Algorithm Visualizer [37], and the PV tools Data Display Debugger [38], and Online Python Tutor [14].
B. Examples of AVs

The JDSL Visualizer (for the Data Structures Library in Java) [39] and JVALL (Java Visual Automated Linked List) [40] were two examples of early data structure visualization systems that were built specifically for the Java application programming interface. Both programs allowed students to generate animations from their code, but while each currently has a website, neither has been updated since 2001 [41], [42].

Many AV prototypes have been built and tested but do not appear to be widely available today. One example was the Data Structure Visualizer, developed with C# and the .NET framework for animating standard manipulations of common data structures [31]. The tool, used for a semester at Webster University, was a practical supplement to lecture content. Students could use the tool; however, they could also see the source code with the tool. Instructors had to expect that students would regulate themselves and complete the programs on their own before checking the model implementation.

Mobile Response System (MRS) [32] was a software environment built for mobile devices and meant to supplement in-class exercises with visualizations and assessment. The developers emphasized the importance of active learning, including interactive elements such as assessment exercises, Back and Next buttons, and parameters that the student or instructor could randomly generate. It is an AV system uniquely created for mobile devices; however, other research has found that students far prefer laptop or desktop computers over mobile devices for this type of activity [17].

Matin et al. developed a customizable AV tool using JavaFX and Kotlin to improve e-learning [33]. In particular, users could utilize the tool to visualize the steps for various searching and sorting algorithms. The ability to modify a visualization is helpful; however, customization options seemed to be limited. Users could initialize random values, search for any value of their choosing, and select a sorting algorithm. Searching and sorting algorithms are essential for students to understand, but many AV tools already include visualizations for these particular procedures.

Researchers have developed some tools that do not fit the traditional definition of an AV system that can visually depict algorithms and data structures. CSTutor was a pen-
based application that acted as an alternative to whiteboards, essentially providing users with an interface for sketching data structures and using gestures to perform operations [43]. Initial assessments found that most students had favorable opinions of the visualizations, but several features could still be improved. Some programming errors could cause the visualization to crash, yet CSTutor did not help locate the error. Also, it was not clear if the tool was definitively easier and faster to use than standard whiteboard drawings.

Another tool that allowed users to draw data structures in a web browser was Dynamic Data Structures (DDS) [44], [45]. Built with HTML5 and JavaScript, it aimed to help instructors teach linked lists and binary trees. Instead of being a straight AV system, DDS provided a platform for interactively building and manipulating a data structure’s graphical model. This tool is currently available online [46], but the instructions for use are lacking. Thus, DDS seems like it would work best as a teaching supplement for someone already familiar with the tool and the data structures.

C. Examples of PVs

SimLIST and SimRECUR [47] were some of the first software packages available for program visualization in Taiwan. Although no longer current, the focus on allowing students to write and manipulate visualizations from their programs, coupled with lab activities, was promising and valuable in its time.

Hoffswell et al. [48] developed a program for embedded visualizations, meant to work as an extension to the Vega visualization grammar [49]. The intention was to limit the necessity of switching back and forth between source code and debugger. The visualizations were helpful in some situations, but other times users found them distracting or irrelevant. Another limitation was the fact that the program required a specific code editor.

Bulmer et al. [50] built a system that highlighted poor coding habits and other common errors in the source code instead of visualizing the program. Bringing attention to these types of issues could help beginning programmers; however, its use was somewhat limited, given that the tool seemed to focus on pointing out errors that a standard compiler, IDE, or linter could find.
Lytle et al. [51] developed a PV to aid students with a pathfinding homework assignment in an Artificial Intelligence class. Most AVs focus on classic algorithms, but since the pathfinding problem has multiple solutions, a traditional AV would not be appropriate in this situation. The assignment had students implement a function called TravelToDestination(), accounting for cases when the robot both did and did not have knowledge of its world. Users could call the visualizer with a separate line in the API. This tool seems helpful, but it was tailored to one specific programming assignment and is not easily accessible to the public.

Willow [52], a pluggable backend for Python and Java, was a web-based PV tool that enabled the creation of visualizations that were similar to the type of visuals found in AVs. The goal of allowing users to manipulate visual elements was a good step. However, the entire program was required to be in a single file, with limited size and restricted access to libraries. The tool still needs more comprehensive testing, but it has many valuable features, including a code editor, sections for input and output, components to show stack history, and a graph of nodes to display the program state.

Online Python Tutor [15] is one of the most extensively used options for program visualization that is available on the web today. It continues to receive updates and currently works with Python, Java, C, C++, JavaScript, and Ruby [14]. Python Tutor is beneficial for stepping through a program and seeing what happens when each line of code executes, but it targets small examples. All code must be in one file, execution stops after 300 lines of code, and all interaction is in the browser. As a tool whose focus is program visualization, it aims to help with debugging, not with students understanding the bigger picture of how an algorithm or data structure functions.

D. Other Software Visualization Systems

Some visualization tools do not fit within the definition of an AV or PV but are still used to help engage students in the learning process.

BRIDGES (Bridging Real-world Infrastructure Designed to Goal-align, Engage, and Stimulate) [53] was a system that created visualizations from data structures based on data from the real world. In this case, the emphasis was on creating a connection with standard data sets, not
helping students understand or debug data structures and algorithms. BRIDGES is open source and currently available on the web [54]. While the graphic representations are visually appealing, the relationship between the visualization and the data structure is not always apparent, so this system’s utility as a teaching tool is limited.

Omnisode [16] was a prototype integrated development environment (IDE) that visualized the entire history of program variables using scatterplot matrices, so users could better understand the runtime behavior of their code. Beginning programmers often have difficulties reading and figuring out how to employ the breakpoints used by traditional debuggers. One of the most straightforward options is to use print statements for debugging, but this can quickly get messy. The idea of Omnicode was promising; however, in practice, users found the scatterplot matrices resulted in visual overload, with too many plots at once, too much unnecessary data, and plots that were difficult to read. The IDE would also be challenging to scale for more extensive programs.

Oka et al. created Kanon [55], a prototype live programming environment that allowed a programmer to observe the results of a program while simultaneously editing the code. The intent was to create an immediate connection between program execution and result without requiring the programmer to visualize how everything runs mentally. This setup removed the separation that usually exists between development and examination environments. Participants in the initial observations generally had favorable views of Kanon; however, the benefits were inconclusive. Results did not significantly improve against the control group that used a simple textual live programming environment.

E. Online Textbooks

Several online textbook options currently exist for computer science education. Some examples include OpenDSA [36], zyBooks [56], and Codio [57]. OpenDSA is free, while zyBooks and Codio have pricing models based on various factors like the number of users and how the instructor uses the content. The widespread use of existing eTextbook options shows that there is a large market for web-based learning tools. Software visualizations are common in eTextbooks, but they are just one component. Often the primary interactions users have with
the visualizations are answering embedded questions and completing other assessment activities. This project aims to complement existing tools and assist educators who do not want to plan an entire course around an online textbook.

Sirkiä and Sorva [58] used log data to investigate how students utilize PVs within an eTextbook. Students seemed to make good use of the PVs, even when students were not required to work with the PVs for their grades. Similar to previous research, they also found that there must be an emphasis on active engagement in order for PVs to be beneficial. In particular, they noticed that continuous animation required less mental effort than interactive media, and students would lose interested in watching videos if they were too long. One way to ensure students engage with the material would be to provide activities that allow them to visualize the code they wrote themselves.

OpenDSA [18] is an open-source, community-based effort to create a complete active eTextbook for undergraduate data structures and algorithms courses. This system integrates text, images, visualizations, simulations, and assessment activities, encompassing much more than a traditional AV system. Researchers developed OpenDSA with JavaScript, HTML5, and CSS, which means that OpenDSA has virtually universal browser support [36]. In general, work on open-source software is never considered complete; thus, there are always opportunities to modify, extend, and improve the project.

Färnqvist et al. [59] discussed their experience using OpenDSA in the classroom. They note that students particularly liked the interactiveness of the eTextbook. The dictionary, visualizations, and exercises gave the students more incentive to spend time with the class content. The log data stored by OpenDSA showed that most students took advantage of the material, particularly around exam time, and that 98.5% of interactions with the system were on a non-mobile operating system. The primary problem with log data is that it shows information regarding the frequency with which the eTextbook is used but does not say anything about how students spend that time.

RecurTutor [60] consisted of two collections of modules within the OpenDSA framework for learning recursion. Students can view examples and visualizations and then practiced with small-scale programming exercises. The exercises were automatically-assessed, so students
got immediate feedback and could engage with the material more constructively.

Mcquague et al. [61] developed interactive teaching modules for data structure and algorithm courses using OpenDSA and Canvas Learning Management Systems. The researchers emphasized that a suitable module should include direct interaction with the visualizations and that variability from auto-generated instances helps with academic integrity. They also suggested that the modules include analytics and data logging so that instructors could assess student performance at the individual, subgroup, or whole class level.

Ellis et al. [62] explored approaches to coordinating eTextbooks with other online tools for programming practice, automated grading, and more. While this paper’s discussion did not center on software visualization tools specifically, the researchers shared some things they had learned from working with online learning tools. They found that it was helpful to introduce new tools in phases and make sure the tool had received thorough testing before use. They also noted that instructors need to pay attention to student feedback and behavior. As an additional benefit, the process of learning and practicing new tools helped build student confidence and initiative, potentially giving students more confidence to explore new tools throughout their careers.

**F. JSAV**

The JavaScript Algorithm Visualization Library (JSAV) [2] is a JavaScript framework for creating engaging AVs with interactive features, intending to facilitate active learning with visual content that is easy to integrate into online tutorials. JSAV is the AV library used in the OpenDSA project. Several engagement levels are available, including static images, slideshows, pop-up questions to which students can respond, and proficiency exercises that require students to simulate steps. The library currently supports standard data structures including arrays, nodes and edges, graphs, linked lists, matrices, and trees [63].

Three groups with extensive experience in AV development had a part in creating JSAV: Aalto University (creators of TRAKLA2), Virginia Tech, and the JHAVÉ community [17]. JSAV uses JavaScript, HTML5, and CSS, offering consistency between browsers and a platform that is easy to integrate with other web technologies and libraries.
G. WebAssembly

JavaScript became the default programming language of the web somewhat by accident. It allowed for dynamic and interactive content on the web; however, its performance has been inconsistent. There continue to be some potential vulnerabilities and security risks with code written in JavaScript. A primary reason for the creation of WebAssembly was to address these problems [20]. WebAssembly has a sandboxed execution environment that allows it to be memory-safe and, when embedded on the web, enforces the same security settings of the browser [64]. The sandboxed environment is not fail-safe, so if memory-safety is a significant concern, developers should take steps to minimize potential issues [65]. The WebAssembly programming language specification was mechanized and verified by Watt in 2018 [66]. This formal process confirms that undefined behavior during execution will not happen. Essentially, this means that the type system of WebAssembly maintains standard soundness properties.

Regarding performance, initial tests showed that C code, compiled to WebAssembly using Emscripten [24], had results comparable to C compiled natively with Clang [67]. Herrera et al. [21] completed a series of experiments to compare the performance of web-based numerical computing options using WebAssembly, JavaScript, and native code on various widely used browsers and devices. Given that JavaScript is an interpreted language, it can be slow for some operations. Developments have been made with JavaScript browser engines to enable improved performance; however, Herrera et al. found that WebAssembly was still 1.5 to 2 times faster in all tested cases.

Researchers have employed WebAssembly for various unique use cases, primarily aimed at improving performance. Reiser and Bläser [22] developed a cross-compiler to translate JavaScript (Typescript) to WebAssembly. They created experiments involving eight computationally intensive algorithms, tested on Chrome and Firefox. TypeScript compiled to WebAssembly ranged from similar speeds (within 2%) to as much as four times faster than standard Typescript.

Matsuo et al. [23] incorporated WebAssembly into a browser-based volunteer computing (BBVC) network. One of the challenges of BBVC, a form of distributed computing, has been that it conventionally relied on JavaScript in the browser; thus, WebAssembly provided an opportunity to speed up computations. In their tests, Matsuo et al. found that, for a highly
complex procedure that required a small amount of data, WebAssembly improved performance by about 49-64%. For a job with low-complexity and a large amount of data, WebAssembly required 4-8% more time than the standard JavaScript method, likely due to the extra overhead required by WebAssembly. Still, WebAssembly performance was more consistent overall compared to JavaScript.
III. Methodology

A. Overview

This project aimed to determine the feasibility of creating web-based visualizations from code written in C++ and compiled to WebAssembly, using the JSAV library. It then explored ways to adapt this application so that novice programmers could use it as a tool for constructing algorithm visualizations. The prototype developed for this project is available in a GitHub repository called BoxAndArrow\(^1\), named after the box and arrow diagrams widely used by computer science instructors to teach data structures and algorithms. This project is published as free and open source software, licensed under the 3-Clause BSD License [68]. It can be adopted, modified, maintained, and integrated into different products with liberal permissions. Open source can be beneficial as it encourages others to contribute, improves the quality of the project, and allows developers to focus on problems that have not yet been solved [69]. The license and contributing guidelines are included in the public repository.

As evident from Section II, many researchers have developed software visualization tools over the years, but most of these systems are no longer available, out of date, or have not grown beyond a prototype. A small proportion of the tools continue to be well-maintained and functional. This project intends to complement the existing systems and test how developers could integrate new web technology into a learning tool accessible by students on their personal computers. WebAssembly allows compiled C/C++ code to run efficiently and safely on the web. This modern standard creates an opportunity to assist students in learning C++ and provides an avenue for students to construct web-based visualizations from their code.

Web-based data structure visualizations are similar to those written on a whiteboard but have a cleaner display. Users can review each step as they go and test a variety of inputs quickly and easily. Several options exist for developing AVs in JavaScript. BoxAndArrow employs JSAV [63], the AV library used by the OpenDSA Active e-Book Project [36]. By default, the code for JSAV visualizations does not directly correlate with any specific algorithm. Building slideshows or animations requires extra steps that can obscure any specific algorithm’s

\(^1\)https://github.com/shelleywong/BoxAndArrow
instructions. JSAV researchers wanted a framework that allowed individuals to easily create engaging algorithm visualization and incorporate them into an online textbook or course materials [2]. Sometimes the best way to create a model example for this purpose is to utilize hard-coded values. Thus, JSAV on its own does not offer a simple method for converting student-written code to an associated AV.

Accordingly, this project utilizes WebAssembly (Wasm), a binary instruction format used to compile and deploy code written in high-level programming languages to the web [64]. In particular, the aim is to provide a simple way to convert code written in C++, a programming language commonly taught in college computer science classes, to a web-based visualization. WebAssembly researchers completed initial experiments with C/C++ modules compiled with Emscripten [20]. Although WebAssembly works with a growing list of languages, the Emscripten compiler environment is mature and well-tested.

BoxAndArrow gives students exercises for writing programs in C++. The code required to complete the exercises is nearly identical to the code required to run the same C++ program natively. Any alterations needed in the student-written C++ code are kept to a minimum and accompanied by thorough instructions and documentation to limit the number of new things the student must learn.

The Appendices include examples of step-by-step sequences that walk through algorithm visualizations for an array, a linked list, and a binary tree. Appendix A shows how one could determine if an array is sorted in ascending order. Appendix B shows the removal of two elements from a linked list, one specified by the user and the other removed from the head. Appendix C shows several insertions into a binary tree.

B. Required Tools

This project requires Emscripten (an LLVM-to-web compiler used to compile C++ to WebAssembly), JSAV (the algorithm visualization library), and Flask (the web server). Emscripten uses Python 3, Node.js, Git, Java, GCC, and CMake. JSAV requires Git, Node.js, and Make. Users can reference the requirements.txt file for installing Flask and its dependencies. These resources are all available for free via the web.
The programming languages used for implementing this project are C++, JavaScript, Python 3, HTML5, and CSS. Makefiles and shell scripts in each directory provide instructions for compiling the program and generating targets. Preliminary work included initial tests using C++ compiled with Clang on macOS. The final project employed C++ compiled with the Emscripten compiler frontend (emcc) on a Linux virtual machine. Many data structures and algorithms courses require compiling with GNU. However, the Emscripten compiler toolchain is necessary for compiling C++ to WebAssembly, and most Clang and GCC options work with Emscripten [24].

This project’s development took place on a 2015 MacBook Pro with a virtual machine running the Ubuntu 20.04 LTS operating system. The application was tested with the latest version of Firefox for desktop\(^2\).

### C. Project Setup

The first step is to set up the environment and make sure a straightforward C++ example, compiled with Emscripten into WebAssembly, connects with JavaScript and the JSAV library. The BoxAndArrow repository includes instructions for setting up the project, getting the source code, and running the program\(^3\). A simple Flask application can confirm that code written in C++ is viewable in the browser [70]. While a local testing server could have been sufficient for this project, Flask is a lightweight framework that provides the option of deploying a production-ready product. A simple development or production server allows users to see their project in a browser on their personal computer. Flask requires relatively few dependencies but is extensible if necessary. Integrating the WTForms Python library with Flask improves user input options compared to plain HTML forms [71]. WTForms also provides CSRF protection, offers form validation options, and organizes forms as class instances.

The command to compile C++ with Emscripten and WebAssembly can include many specifications, so creating a Makefile allows the developer to streamline the compilation process. In general, compilers transform human-readable source code into instructions that a machine can

\(^2\) Last tested with Firefox 87.0, released on March 23, 2021.

\(^3\) Instructions for set up are included in the README.md: https://github.com/shelleywong/BoxAndArrow
understand and execute. The target files generated by Emscripten include a .wasm file, .js "glue" code, and an HTML file that works with the .wasm and .js files in the browser. By default, the generated files get created in the directory in which the code compiles. Flask requires a directory named templates for .html files and a directory named static for .js and .css files, so it helps to write a shell script to move the generated code into the appropriate locations. This script could be easily updated to complete other tasks that must execute before the user runs the application.

D. Building the Prototype

This project’s first goal was to confirm that C++, WebAssembly, JSAV, and Flask could work together successfully. Functions written in C++ needed to be callable from the JavaScript driver code, and the corresponding visualization needed to match the expected behavior based on the instructions provided by the code. As a preliminary step, it was beneficial to create separate example projects with JSAV and Emscripten (the WebAssembly compiler). The JSAV documentation [63] provides instructions for getting started with JSAV and creating a slideshow. The Emscripten documentation [24] walks through the steps to download, install, compile, and run a test program. In both cases, users can complete the initial example programs in the JSAV and Emscripten repositories and run them on a local testing server.

The BoxAndArrow directory contains all of the necessary JSAV files, so students would not need to download the JSAV directory. BoxAndArrow utilizes a minified version of the JSAV library to reduce file size and load time. Minifying and obfuscating make the code more difficult to read; however, students would not need to change anything in the JavaScript or understand precisely how the library works to create their C++ programs. On the other hand, developers who want to recreate the project may benefit from downloading the original JSAV files. One minor but significant change in BoxAndArrow is that results which were undefined in the original JSAV library now return "undefined" as a string. The undefined result came up when trying to access out-of-bounds memory. If standard JSAV receives an undefined value, it causes the visualization to fail automatically when the page loads. The console displays an error message, but the cause of the error is not immediately evident to the user. BoxAndArrow manages "undefined" results as a string. This feature allows the code to run like normal until
encountering the error. Then the slideshow displays an error message, and the user can see the specific function in which the error occurred.

Next, development can begin in a separate repository that combines the JSAV and Emscripten. The layout can mirror the Array, List, or Tree directory within the BoxAndArrow directory. The JSAV static files needed for this project include the JSAV library itself, the jQuery and Raphael libraries used by JSAV, and a supporting CSS file. An HTML file should contain JSAV instructions for creating a slideshow and the Emscripten HTML shell file code for launching a WebAssembly program. The directory should include a C++ program either contained in a single file or separated into a header file and an implementation file.

Developers should create a Python virtual environment and install the Python packages used by this project. The requirements.txt file in the BoxAndArrow directory lists packages. Programmers should establish separate isolated virtual environments for different Python projects. A virtual environment allows individual projects to have unique dependencies and makes sure these do not conflict with packages and versions used by other Python projects. The Flask documentation [70] includes instructions for creating and running a minimal Flask application. A Flask application file establishes any URL paths used by the project. All JavaScript and CSS files will need to be in a directory named static, and all HTML files will need to be in a directory named templates. When the C++ code compiles to WebAssembly, it generates several files containing WebAssembly byte code, JavaScript glue code, and HTML necessary for executing the code in a browser. Writing a script to move these files into the subdirectories expected by a Flask application minimizes the steps required to run the program. The commands for compiling to WebAssembly can become quite long, so creating a Makefile simplifies the compilation process and reduces the chances of making mistakes with the program.

This application handles GET and POST HTTP methods. A GET request retrieves the launch page, which holds a form that allows users to determine what happens in the visualization, as shown in Fig. 1a. Users can randomly generate input or enter values of their choosing. The form also lets users select commands from a dropdown menu to test with the input. The commands reflect the functions that the user implements in C++. When the user submits the form, it creates a POST request that uses the form data to create unique AV slideshows, as shown in
Fig. 1b. The setup includes a text area that displays any C++ output printed to standard out below the slideshow, which may assist with debugging. If users want to try a different combination of input and commands, they click a restart button and get directed back to the launch view. Offering a mechanism for saving form submission data was not a priority, as the tool intends to be a replacement for whiteboard and scratch paper drawings. However, the page displays the current commands and input above the visualization, which the user can reference at any time.

![Create an Array Visualization](image)

**Fig. 1a.** GET Request: Array Visualizer Launch Page

![Array Visualizer](image)

**Fig. 1b.** POST Request: BoxAndArrow Array Visualizer

Developers will need to create a JavaScript file to instantiate the Emscripten module, the JSAV container, and instances of the JSAV and C++ data structures. In this .js file, the Emscripten module instantiates a C++ object and calls the C++ functions. The results returned
from these methods direct the visualization. The JavaScript code follows similar algorithms to those implemented in C++ but includes extra steps to direct the slideshow and handle interactions between the JSAXV container and the Emscripten module. When the user correctly implements the program, the visualization works as expected. If there are bugs, the visualization must handle the errors by either producing incorrect results in the visualized data structure or displaying an appropriate error message when the slideshow encounters the issue.

Some of the code in this JavaScript file contains similarities to the expected student C++ implementation. To discourage students from immediately borrowing this code and adapting it for their C++ functions, BoxAndArrow contains a minified and obfuscated version of the file. It is still possible for students to see this code; however, reversing the minified JavaScript and translating the methods to C++ would be more work than just writing the code from scratch. Additionally, the JavaScript implementation contains extra steps that allow the visualization to work correctly, and better examples of pseudocode are readily available online. Ultimately, making it a little more challenging to see the provided application’s solution encourages students to figure out the implementation independently.

The BoxAndArrow directory provides exercises and includes starter code for several algorithms and data structures found in standard data structures courses. The README files include directions for setting up the environment and suggestions for using the visualization to test and debug code. The instructions provide hints about each particular data structure and common mistakes made with each data structure. If the user is unfamiliar with web programming, the files also include tips for working with the web console and clearing the cache. The complete prototype application has everything implemented except the C++ functions. Thus, while some setup is necessary, BoxAndArrow walks the user through the process and provides exercises with a level of difficulty similar to CS2 lab assignments.

E. Compiling the Code

On a Linux operating system, it is common to compile C++ code with the GNU Compiler Collection [72]. The Emscripten Compiler Frontend (emcc) is used in place of GCC to produce WebAssembly output. A Makefile in the Array, List, and Tree directories makes
it easy for a user to compile the program without memorizing several lengthy commands. The
commands each include the -o option, which specifies that Emscripten should generate an HTML
file, a WebAssembly binary code module, and JavaScript ”glue” code that allows WebAssembly
to run in a web browser.

Each compilation command includes the -O3 option to specify build optimizations
and minify the code. Optimizations help the program load and run faster. Developers should
include the –shell-file option to specify an HTML template to launch and run the code. The
shell files used in BoxAndArrow are adapted from the default Emscripten shell file [24]. When
work began on this project, it was necessary to include the -s WASM=1 option to specify that
the compiler should produce WebAssembly output instead of asm.js output, the precursor to
WebAssembly [64]. Emscripten has updated since then to produce WebAssembly by default,
without any additional flags.

There are a couple of differences between the Array Visualizer compilation command
and the List and Tree compilation commands. The Array directory contains a .cpp file with a
set of independent array functions, and a list of EXPORTEDFUNCTIONS in the compilation
command includes each array function called by JavaScript. A couple of Emscripten runtime
functions, named ccall and cwrap, are necessary to call C++ code from JavaScript. While ccall
calls a compiled C++ function and returns the result, cwrap returns a JavaScript function that
lets users call the C++ function as if it were a standard JavaScript function. The BoxAndArrow
directory uses cwrap to allow the compiled function to be easily called in multiple locations if
necessary. The more functions included, the longer the compilation time, so exporting individual
functions allows the user to change the exported functions at any given time.

The List and Tree Visualizers use Embind, an Emscripten feature that provides binding
for entire C++ classes. In these cases, the compilation command includes the –bind option. The
class implementation file includes a block of binding code, and all member functions listed in
the block are callable from JavaScript. The more functions added, the larger the code size, so
the user should only include the public methods called from JavaScript in this block. In the
Tree class, private member functions include an extra character at the end of the function name
to let Emscripten know not to bind these functions and indicate that these are recursive helper
functions only called from within the class.

F. Array Starter Code

BoxAndArrow users should begin in the Array directory. Each function included within this file is independent, so once a single function is complete, the user can test it in the visualization. A simple example using an array data structure requires a function that accepts two parameters, a pointer to an integer array and an integer array length. The client-side array input gets passed to the C++ function, and the server-side function returns a result that the JavaScript code can use to create the visualization. The user can step through the slideshow in the browser and determine whether the resulting visualization matches the expected results of the C++ program.

The Array directory includes function prototypes for find_largest, calc_sum, double_vals, rotate_left, and is_sorted. When the user includes these functions in the set of exported functions, a global Module object can call them from JavaScript. The JavaScript implementation sees the return values from these functions and can create visualizations based on the results. The function to find the largest value in the array returns the maximum value as an integer. The function to calculate the sum of array values returns the array sum as an integer. The function to determine if the array values are in ascending sorted order returns true if the array is sorted and false if it is not. Like using diff in the command line, the program checks the user’s result and compares it to the expected result, providing a message if anything does not match. The functions to double the array’s values and rotate all values in the array to the left both return a pointer to an integer array. The visualization checks each value in the returned array and compares it with the model solution. If any value does not match, the visualization shows a message indicating this discrepancy.

G. WebAssembly Linear Memory

By default, WebAssembly works with 32-bit and 64-bit integer and floating-point numbers. Passing arrays from JavaScript to C++ and vice versa requires the use of WebAssembly’s linear memory [73], which represents memory as a contiguous array of untyped bytes. The
programmer needs to allocate the amount of memory required by the array, assign data to this portion of memory, and then free the memory after the program calls the JavaScript-wrapped C++ function and returns a value. WebAssembly modules work in a sandboxed environment for security and memory safety reasons [64]. Thus, linear memory is separate from the execution environment, and a compiled program cannot jump outside of its designated memory space [20]. A C++ program could still produce undefined results within its allocated memory area, but it cannot access memory in the same direct way as pointers in native C++ code. Additionally, a JavaScript typed array is necessary to match the types used natively in C++ [74], as standard JavaScript arrays do not specify a type. A few extra steps are required to pass array parameters and return array values, but all of the additions happen in the JavaScript implementation, and the C++ code remains relatively unchanged.

Each array function gets passed two parameters, a pointer to an integer representing the first element in the array and an integer to keep track of the array length. If the user’s program steps outside of the array bounds, the program grabs the value previously stored in that memory address. In some cases, the program may look like it works as expected, but other times visiting that location produces undefined results. When the code contains an error, the programmer will run into an example where the visualization produces an incorrect result with sufficient testing. Results that do not match the model result get flagged, indicating that the person who wrote the program should review the code. Methodology subsection III-K contains details on error detection provided by this project.

H. Linked List Starter Code

The List and Tree directories rely on Embind, an Emscripten feature that allows C++ classes to compile and run similar to native code. An `EMSCRIPTEN_BINDINGS` block binds the class and all public methods that the user may call from JavaScript, allowing an instance of the C++ class to exist in JavaScript. Although JavaScript has a garbage collector, the programmer must still manage any dynamically allocated memory. When the JavaScript object is no longer needed, Emscripten does not automatically call the destructor of a C++ object [24], so the
programmer needs to make an explicit call to the delete method on the instance. Then the delete method calls the C++ destructor, thereby freeing the memory used by the object.

Unlike the Array Visualizer, all linked list methods are executed on the same linked list data structure. The commands shown in Fig. 2a are all standard operations performed on linked lists. Fig. 2b shows how operations happen in sequence for the visualizations.

![Create a Linked List Visualization](image)

Fig. 2a. Example of the launch page where users can test standard linked list operations.

![Linked List Visualizer](image)

Fig. 2b. Example of a sequence of linked list operations execute on the same list instance.

The List class includes a private nested Node class. Each Node contains an integer
value and a pointer to the next linked list Node. Member functions of the List allow for insertion at the head or tail, resetting the head or a specific value, printing all values, looking up target values, and finding the linked list length. Many students have no prior experience with linked lists before encountering them in a data structures class; thus, the Linked List Visualizer provides some starter code that can familiarize them with the basic syntax. The example code shows how to use constructors to initialize List and Node classes, dynamically allocate memory for a Node, and insert a Node into the List. A standard print function shows how to traverse a linked list. The List destructor shows how to dynamically free all memory used by the linked List.

For BoxAndArrow, a provided print function serves a couple of tasks. The function prints each List item’s value to the standard output stream. The program calls the C++ List instance print after running each command, so the user can see what happens after each insertion and removal. Everything printed to standard output appears in the browser below the visualization. The print function also returns a space-separated string of values representing the C++ List. Code in JavaScript uses this string to compare the C++ List instance with the JSav linked list visualization. If there are any differences between the lists, the visualization posts a message indicating that the code contains errors that the programmer should review.

1. Binary Tree Starter Code

The Tree class includes a private nested Node class. Each Node represents a binary tree root that can have zero, one, or two children. A Node contains an integer value and pointers to the root’s left and right subtrees. A standard JSav binary tree visualization would not include null nodes, but BoxAndArrow includes them so that the visualization can react based on user-written code. Fig. 3 shows a BoxAndArrow binary tree with null nodes conceptualized in the visualization. If the left and right nodes are null, the current root is a leaf node. Member functions of the Tree provide the ability to insert and find nodes, determine the number of nodes in the Tree, check whether the Tree is balanced, and print the Tree using in-order, pre-order, and post-order traversals. All of the public member functions have a corresponding private recursive function. While a student would generally have some experience with recursive functions before
being assigned a project to create a binary tree, the Binary Tree Visualizer README includes hints for working with recursive functions as reminders in case the student gets stuck.

![Binary Search Tree Visualizer](image)

Fig. 3. Example of a BoxAndArrow binary search tree visualization.

The starter code includes a helper function that follows a sequence of instructions similar to the algorithms used for inserting and finding nodes in a tree. The insert and find functions do not traverse the entire Tree; instead, they take the most direct path possible to a target location in the Tree. This attribute allows these functions to have $O(\log n)$ runtime complexity. The helper function’s job is to keep track of the path taken to a particular node in the Tree. The character "1" indicates that the path goes to the left subtree, the character "2" indicates that the path goes to the right subtree, and the character "0" indicates that the function found the target value. If the current root is null, it means the function has reached a base case and should return. The helper returns a string of digits representing the path. The user could accomplish this task within the insert and find functions; however, there is no clear reason to increase the number of tasks these functions must complete. A standard binary tree insert or find function would not track its path; thus, a separate function takes care of this task in BoxAndArrow.
This project’s development required implementing the C++ programs for widely-used data structures, writing JavaScript to call the C++ functions, connecting the function results to a slideshow visualization, and creating easy-to-understand options for users to test the methods and various input values. BoxAndArrow uses slideshows instead of animations because these allow users to move through the visualization at their own pace and quickly return to a step if they want to review anything. While an animation requires little to no interaction, a slideshow encourages pausing and thinking about what happens at each stage within the data structure.

J. Providing Documentation and Instructions

After developing BoxAndArrow to work with correct C++ implementations, the project needed to address Research Problem 2, namely adapting BoxAndArrow into a tool that could be beneficial to students and instructors in an educational setting. The goal was to create a foundation for students to construct visualizations from their code. The first step to achieve this was to create a separate repository and remove any code that would give away the solutions. This directory needed to include thorough documentation for setting up the project and guidelines for walking users through the programming exercises. It was also essential to add error checking features and messages to help catch bugs. This project requires several tools that a beginning programmer might be unfamiliar with, but the documentation walks through each step to make the process as smooth as possible.

The BoxAndArrow repository includes a general README that outlines how to set up, compile, and run the application on a simple server. The goal is to provide a way for students and instructors to view data structure visualizations through a web browser on their personal computers, so localhost is sufficient for this purpose. The Array, List, and Tree subdirectories include README files with instructions specific to each visualization. These README files include exercises that work with the provided starter code. Function prototypes and class definitions specify all necessary methods, parameters, and returned values. To complete the exercises, the user must follow the specifications and implement the C++ functions. If the student runs into errors, the README files include hints that address common problems and assist with debugging. Since some problems may be unique to web development, the files include
information about working with code that runs in the browser. Students with little to no web programming experience can review the tips included for using the console and clearing the browser cache. The provided JavaScript and HTML driver code allows the students to focus on the algorithms and not spend all of their time figuring out how to get the AV to work. To complete the exercises, the user only needs to implement the functions in the .cpp file.

BoxAndArrow is available as a public starter repository with an open-source license on GitHub [75]. The repository contains documentation, including instructions for setting up the environment, instructions for using the project, and a file for contribution guidelines.

K. Error Checking Overview

To get the most out of BoxAndArrow, users must thoroughly test their code with various inputs and confirm that the visualization matches their expectations for each test. BoxAndArrow provides error messages for incorrect return values and user results that do not match the expected results. While the error messages cannot always identify the specific problem, they can bring attention to any discrepancies and direct users to the function that contains the error. Sometimes, there will also be a mistake visible in the slideshow. Then the programmer can narrow down the problem by reviewing the input and commands that the program executed before the error occurred. A few bugs cause JavaScript internal errors, namely infinite loops and infinite recursion. In these cases, the entire visualization fails, and an ”Uncaught Internal Error” message appears in the web console.

BoxAndArrow focuses on addressing runtime errors that arise during execution, not compilation errors that prevent a student’s code from successfully compiling. Many compile-time errors correspond to incorrect syntax or semantics. This project, similar to a standard C++ program, requires that students pay attention to compiler warnings and error messages. The Emscripten compiler produces messages similar to those provided by Clang and GNU compilers, so working with this tool should feel similar to working with other projects through the command line. While novice programmers must learn to address compilation errors, figuring out how to identify runtime errors is also an important skill. Often student programmer mistakes are due to sloppiness, a misunderstanding or misinterpretation of an assignment, or an issue in the logic that
the student uses to solve the problem [34]. This program aims to encourage thorough testing, help users better visualize data structures and algorithms, and provide visual cues indicating where logic errors may have occurred. Sometimes the best way to avoid careless errors is to spend more time working with the code. A visualization tool that changes based on the student’s work gives the student a level of ownership and encourages spending more time with the material [8].

L. Array Error Checking

C++ source code is ready to load and execute after it runs through a compiler. The Array Visualizer works by calling the compiled C++ functions from JavaScript and updating the visualization based on the function return values. While the tool cannot see what happens at each step in the student-written code, it can still identify many common errors by comparing the actual and expected results. There is more than one way to implement a function, but the results should always match in the end when the implementation is correct. If there are any differences, the slideshow displays an appropriate message and gives the student a visual representation to consider. Table 1 lists common programming mistakes addressed by the Array Visualizer.

Uninitialized variables are common mistakes for beginning programmers. In the array find_largest function, the user might make the mistake of not initializing a local variable to hold the current largest value. Another error might be to initialize the variable to zero and ignore cases when all array values are negative. In these cases, the value returned by the C++ function will not match the value that was determined to be the largest in the JSav array.

This mistake could also happen with the is_sorted function. For example, one solution involves using two iterators to check if neighboring values in an array are in ascending order. The first time this conditional statement was not true, the program would immediately return false. The function returns true if it has traversed the entire loop and found no pairs to be unsorted. If the user forgets to initialize at least one of the iterators to zero, the program does not enter the for-loop and always returns true, as shown in Fig. 4. If the student only tests sorted arrays, the program may appear to work, but testing an unsorted array will produce an incorrect result.
<table>
<thead>
<tr>
<th>Array Error</th>
<th>Array Visualizer Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overstepping array boundaries</td>
<td>Undefined results</td>
</tr>
<tr>
<td></td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Uninitialized iterator variables</td>
<td>Incorrect results shown in the visualization</td>
</tr>
<tr>
<td></td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Setting a variable to an uninitialized value</td>
<td>Incorrect result shown in the visualization</td>
</tr>
<tr>
<td></td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Forgetting that C++ arrays are zero-indexed</td>
<td>Incorrect results shown in the visualization</td>
</tr>
<tr>
<td></td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Using a single = sign to check equality</td>
<td>Emscripten compiler warning</td>
</tr>
<tr>
<td>Including an extra semicolon</td>
<td>Emscripten compiler error</td>
</tr>
<tr>
<td>Undeclared variables</td>
<td>Emscripten compiler error</td>
</tr>
<tr>
<td>Uninitialized variables that should be initialized to zero</td>
<td>Currently no error checking available</td>
</tr>
</tbody>
</table>

Fig. 4. Example of an uninitialized array iterators error response.
Another frequent error is overstepping the array boundaries. Sometimes students forget that C++ arrays are zero-indexed, so they initialize an iterator to 1 and iterate through the array until it equals the array size. The memory location immediately after the space allocated for the array will hold an undefined value. When a student makes these mistakes in BoxAndArrow, a message appears in the slideshow indicating that the user’s result did not match the expected result, as shown in Fig. 5. Occasionally it will appear that a set of inputs produces the correct results. However, in a thoroughly tested program, the results produced by incorrect code will not match the expected result every time.

![Array Visualizer]

Fig. 5. Example of an array out-of-bounds error response.

Learning to test code is a valuable programming skill. While BoxAndArrow does not teach students how to test code, it provides a quick and straightforward procedure for students to try different input values and analyze their results. Like `find_largest`, the `calc_sum` function also returns an integer value, so the mechanism for highlighting errors is similar. When the user’s program includes an iterator that goes past the array bounds, it calculates an additional value from memory into the sum. The return value will not match the expected result, and the slideshow will highlight the discrepancy.

The `double_vals` function returns an array that is the same size as the original array, with all values doubled. BoxAndArrow uses the returned array to update the array in the
visualization. A static copy of the original array appears below the altered array so that the user can confirm that each value changed as expected. A conditional statement within the visualization code checks if the value at each index matches the expected result. Any difference between the arrays tells the slideshow to display a message that directs the user to compare the updated array with the original array. For example, if the user made an indexing error and started the array at index one instead of index zero, the array’s first value would not be changed in the visualization. Students could print values to standard output to identify these problems, but this tool makes it easy to see the output as an array data structure and not just numbers in a line.

In the rotate_left function, the user needs to make special considerations for the values at the ends of the array. Changing the value at index \( i \) to the value at index \( i+1 \) works for all elements except the last one, which should update to the value that had been at the front of the array. When the user forgets to consider this edge case or updates the value in the array at the length index, the rotated array appears correct except for the last index value, as shown in Fig. 6. A slideshow message will indicate that something in the array does not match the expected results, but the error will also be clear from the visualization. When a program goes outside the array bounds, it grabs whatever value was previously in that memory address, often a garbage value with no apparent relation to the other values in the array.

There are various ways for students to write programs that unintentionally overstep array boundaries. For instance, in the rotate_left function, one solution is to create a local variable to hold the value at index zero, then assign the value to the element at the length-1 index after iterating through the rest of the array. Novice programmers might try to access a variable outside of the array bounds by assigning the local variable to the length index, the memory location immediately after the array. In this case, a location in memory will update, but the last element in the array will not. The visualization will see that the returned array does not match the expected results, so a slideshow message will indicate that the user should evaluate the two arrays in the visualization and consider what situation might have led to the error.

There are extras challenges involved with creating a program that both executes correctly and catches programming errors. The original visualization setup did not catch every instance of overstepping array bounds in functions that return a pointer to an array. Going one
index past the array would change a memory location’s value, but the visualization would only check the designated array indices. The array itself would appear correct even though issues in the code made the implementation incorrect. Addressing this problem required updating the JavaScript array to include a placeholder integer immediately after the last index value. Now, any change to the value at this location in WebAssembly memory produces a warning message. Thus, even if the returned array matched the visualization array, the user is provided with a message to review their code.

M. Linked List Error Checking

Many novice programmers run into errors while working with linked lists that they would not run into while working with arrays. Both are linear data structures, but arrays have a set size, while linked lists use dynamically allocated memory and can grow or shrink as needed. Some common linked list mistakes produce compiler errors, such as using the dot operator instead of the arrow operator with dynamically allocated objects. However, many linked list bugs result in segmentation faults and other errors that are only identifiable at runtime. This project encountered some challenges due to the differences between standard C++ memory

Fig. 6. Example of a missing edge case error response.
and WebAssembly memory, and therefore it cannot catch all possible programming errors. Still, several features have been added to BoxAndArrow to handle programs that try to access unallocated memory. Table 2 lists common programming mistakes addressed by the Linked List Visualizer.

<table>
<thead>
<tr>
<th>List Error</th>
<th>Linked List Visualizer Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out of bounds: Trying to access next node of null pointer</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td></td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Uninitialized variables</td>
<td>Incorrect results shown in the visualization</td>
</tr>
<tr>
<td></td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Not resetting list head after removal</td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td></td>
<td>Future list commands may not work</td>
</tr>
<tr>
<td>Not resetting previous node’s next pointer after removal</td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td></td>
<td>Future list commands may not work</td>
</tr>
<tr>
<td>Forgetting edge case: insert at head</td>
<td>List never created</td>
</tr>
<tr>
<td></td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Forgetting edge case: remove from empty list</td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Forgetting special case: no duplicates</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>Forgetting special case: target not in list</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>Confusing the . and - &gt; operators</td>
<td>Emscripten compiler error</td>
</tr>
<tr>
<td>Undeclared variables</td>
<td>Emscripten compiler error</td>
</tr>
<tr>
<td>Out of bounds: Trying to access value of null pointer</td>
<td>Currently no error checking available</td>
</tr>
</tbody>
</table>

A common beginner mistake is to write code that accesses a location in memory that
the code is not supposed to access. This situation comes up frequently in programs that utilize pointers. A program may assign a Node pointer to null, but attempting to access a null pointer’s value or next pointer results in a segmentation fault. Novice programmers also frequently fail to consider special cases, such as inserting or removing from the beginning or end of a linked list, forgetting to handle duplicates, or not addressing cases when the list does not contain a target element.

BoxAndArrow implements some starter code to offer an example of linked list syntax to novice programmers, including a function that inserts a node at the head of the linked list. The first BoxAndArrow exercise asks students to find the list length. Any returned value that does not match the number of nodes in the visualization triggers an error message in the slideshow. Like with arrays, uninitialized counter variables do not produce errors. However, uninitialized traversal variables result in error messages displayed in the slideshow. In the `length` function, forgetting to assign a traversal pointer to the head node makes the visualization think that the list is always empty, as shown in Fig. 7. The user can also look at the visualization itself. If the function returns empty, but more than zero nodes appear in the visualized data structure, users know they should check their code.

![Linked List Visualizer](image)

Fig. 7. Example of an error response for an uninitialized traversal variable in a list function.

A version of the overstepping array bounds error often happens with linked lists. Suppose the user tries to access the head node’s next pointer without first checking that the list contains one or more nodes. When the list is empty, this results in a segmentation fault in
standard C++. While the memory is still technically accessible in WebAssembly, BoxAndArrow sees this as going outside of the memory allocated to the List object. The slideshow provides a message indicating that the function produced an unexpected result, and the program may have tried to access memory that it is not supposed to access.

The purpose of the lookup function is to find a target value in the list, returning the node’s list location or -1 if the lookup was unsuccessful. The return value is similar to an index value, and it allows the visualization to find the node at the specified index. If the result is a location within the linked list, the user can confirm that the visualization highlights the node at the correct location. If the returned location is not within the list, the slideshow indicates that the code produced unexpected results. BoxAndArrow catches all cases when the user’s program tries to access a node outside of the set of designated linked list nodes. The lookup function uses a standard traversal, so features are included to catch mistakes similar to those encountered with the length function.

One difference with the lookup function is that a user needs to check the list node object’s value. Suppose the list is empty and the program has a condition to check the value at the head. This situation causes a segmentation fault in standard C++. The WebAssembly and JavaScript setup allows programs to access this memory location, so the operation does not produce any errors by default. During testing, it appeared that a null pointer always had a value of 0, but 0 is a valid integer. If the program did not have any other errors, the result returned to JavaScript was valid. BoxAndArrow can check if the program is trying to access a Node that does not exist, and it can see if a value returned from the function does not match expected results. However, it cannot see if C++ code tries to check a null pointer’s value at any point in the implementation.

The third exercise is to remove the head of the linked list. The starter code includes a linked list destructor that removes all nodes from the list, so the student can see the basic syntax for removing a node. The destructor code can be adapted to only remove the node at the head of the linked list. When the program removes the head node, the function returns true, but if the list is empty, there were no nodes to remove, and the function should return false. There is no need for the code to traverse the list in this case, but the student may forget the special case of
returning false when the list is empty, as shown in Fig. 8. If this happens, BoxAndArrow sees that the program is trying to remove a node that does not exist, and it displays a message that the user was trying to remove a node from an empty list.

The last two linked list exercises are a little more complicated than the first three. The function to insert a node at the end of the list has several special cases. If the list is empty, the program needs to insert the node at the head; otherwise, it needs to traverse the list until it finds the last node. The program should insert the new node after the previous last node. In this exercise, the instructions also ask that the student not insert any duplicate values, so the program must check if the head or any other values match the value designated for insertion. One potential error would be to traverse the list until the current pointer equals null. Instead, the program should stop when the current pointer’s next node is null so that the programmer can create a link from the existing list to the new node. If the code then tries to access a null node’s next pointer, it results in a segmentation fault in standard C++. Even though this application does not produce a segmentation fault in this situation, the slideshow identifies the problem, displays an error message, and does not insert the value into the list. Fig. 9, shows the case where the first insertion at the head gets handled correctly, but subsequent insertions produce an error. The program uses the C++ length function to determine the number of nodes in the list after a node insertion, and if there is any difference, the slideshow knows something is wrong and sends a signal to the user. The C++ list will not include the node that the code was trying to insert, so
the length will not match the visualization list length.

![Linked List Visualizer](image)

**Fig. 9.** Example of an error response when a link between list nodes is mishandled.

BoxAndArrow handles common insertion edge cases. An *insert_at_end* function must handle the case when the list is empty, and insertion at the end of the list is also an insertion at the head of the list. The program never initializes the linked list if it omits this case, so the slideshow displays a message indicating that the user should check that the code handles insertion when the list is empty. A function that is not supposed to accept duplicate values should include a condition to check if a value is already in the list. If the programmer forgets to account for this case, a node with a duplicate value will appear in the linked list visualization and indicate to the user that they have forgotten to address a special case for this function.

The last exercise is for a function that removes a specified value from the list. The value could be at the head, in the middle, or at the end of the list. The function should also handle the case when the list is empty. A typical student mistake would be to make the program traverse the list and check the current pointer’s node value. If this program finds and removes the current node, the list does not remain intact because the previous node no longer connects to the list nodes that come after the removed node. A traversal pointer should stop immediately before the target node so that the node before the target can be assigned a link to the node that comes after the removed node. The visualization initially appears to remove the node successfully; however, the visualization checks if the C++ list object matches the JSAV list before the subsequent command runs. An error message will display in the visualization because the lists will not
match when the `remove` function gets implemented incorrectly. In this case, the string returned by the provided `print` function allows the program to check the C++ list against the JSAV list.

Students are likely to make the mistake of not accounting for special cases in a linked list `remove` function. For example, the programmer may forget to return 0 after traversing the list and not finding the target value. As shown in Fig. 10, the code automatically removes the last node even when the value does not match the target value, the visualization will not print an error. However, a node that was supposed to remain in the list will be removed from the visualization. If the error is not caught by the user immediately, any subsequent function calls could result in an error because the C++ List object and the visualization list would not match.

![Linked List Visualizer](image)

**Fig. 10.** Example of a slideshow error message for a program that fails to check for the case when a target node is not found in the list.

Another edge case error involves forgetting to check if the node targeted for removal is at the list head. The program may successfully remove the node at the head, but if the programmer forgets to reassign the head node pointer, any future operations on the list will fail. Fig. 11a, shows how the visualization will see that the C++ List object does not match the visualization list and print an error message immediately after the operation. The user can also look at the standard out results in the text box below the visualization, as shown in Fig. 11b. This output will show garbage values printed in place of the expected list values because the head no longer has access to the remaining nodes in the list.
Fig. 11a. Example of a slideshow error message after failure to reset the head.

Fig. 11b. Example of standard output display after failure to reset the head.

N. Binary Tree Error Checking

The BoxAndArrow Tree Visualizer provides exercises for implementing a recursive binary search tree. Some errors encountered while working with binary trees are similar to those found while working with linked lists. Given that both implementations utilize dynamically allocated memory, it is common in both cases for beginning programmers to run into segmentation faults while writing traditional C++ programs for these data structures. Developing this program required additional considerations for errors caused by incorrect implemented binary tree recursion. Table 3 lists common programming mistakes addressed by the Binary Tree Visualizer.
<table>
<thead>
<tr>
<th>Tree Error</th>
<th>Binary Tree Visualizer Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swapping &lt; and &gt; signs</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>Forgetting an optional terminating condition</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td></td>
<td>Future Tree commands may not work</td>
</tr>
<tr>
<td>Out of bounds: insert</td>
<td>Error message displayed in slideshow</td>
</tr>
<tr>
<td>Out of bounds: find</td>
<td>Error in visualization (tree node not found)</td>
</tr>
<tr>
<td>Off by one: traversal</td>
<td>Printed results do not match visualization</td>
</tr>
<tr>
<td>Not visiting both subtrees</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>Visiting subtrees in wrong order</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>Not accounting for empty list: size</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>Not incrementing count</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>Incorrectly implemented recursive calls: size</td>
<td>Error message displayed in the slideshow</td>
</tr>
<tr>
<td>Not accounting for all cases: balanced</td>
<td>Incorrect results shown in visualization</td>
</tr>
<tr>
<td>No base case</td>
<td>JavaScript internal error: message in console</td>
</tr>
<tr>
<td>Missing base case (multiple base cases needed)</td>
<td>JavaScript internal error: message in console</td>
</tr>
<tr>
<td>Recursive step does not reduce problem size</td>
<td>JavaScript internal error: message in console</td>
</tr>
</tbody>
</table>

The standard binary tree structure is relatively straightforward; however, it is more complicated than a linear data structure like an array or linked list. Linear data structures all have a single entry point and a single exit point. This organization makes it easy to print the data
structure to a terminal in a standard traversal order. A binary tree still has one entry point, but each node can point to zero, one, or two other nodes instead of a single pointer to the next node. There are several logical ways to traverse and print a tree. The programmer can print each binary tree node value using a linear format, but this makes it challenging to see the tree structure. It is possible to print the tree to a terminal and show the tree structure, but it is more complicated to implement. Thus, a visualization tool for binary trees is helpful simply for showing the structure of the tree. Even without errors printed to the screen, a web-based visualization for trees allows the user to see the intrinsic data structure format and understand how their code impacts the nodes in the tree.

In a standard binary tree, the programmer should always insert nodes as leaf nodes with left and right pointers to null. If the tree is empty, the node gets inserted at the root of the tree. Then, to maintain the binary search tree properties, nodes with values less than the current root value are inserted in the left subtree, and nodes with values greater than the current root’s value get inserted in the right subtree. Some mistakes do not trigger error messages in the visualization; however, the visualization itself provides a clear picture of the data structure, making it easier for the user to see if they implemented something incorrectly.

BoxAndArrow provides a Tree constructor that initializes the root to null. The user must create a constructor for Node objects and destructors for Tree and Node objects. These special member functions are difficult to visualize. The Node constructor initializes the member variables of a Node object. However, as a private member function, it is only called by the Tree insertion method, not from outside of the function. In standard C++, the program never explicitly calls the destructors. Instead, the destructor destroys the object instances when the object gets deleted or the object goes out of scope. Emscripten requires that the programmer explicitly delete any class included in an EMSCRIPTEN_BINDINGS block [24], but this is taken care of in the JavaScript code of BoxAndArrow. There is no straightforward way to test destructors in the visualization. However, extensive instructions are provided in the Tree directory README to walk students through the steps to create these member functions.

After the constructors and destructors are complete, the student should implement the insert and helper functions. Both follow the same basic algorithm, but the helper function’s job
is to track the path taken to the insertion location. This path helps the visualization reflect the user’s implementation. Without a functioning helper, the slideshow displays an error message on the first insertion attempt because the C++ tree instance does not match the tree visualization. If the programmer implements the insert and helper functions and both have the same error, an error message will not print. However, it will be easy to see from the visualization that the programmer made a mistake. For instance, if the less-than sign gets swapped with the greater-than sign, the visualized binary tree values will be swapped. Suppose the helper function has this error and the insert function does not, or vice versa. As shown in Fig. 12, the visualization would display an error message immediately after the first insertion because the path returned from the helper would not match the actual path to the insertion location.

![Binary Search Tree Visualizer](image)

Fig. 12. Example of a slideshow error message for a program that tries to access the memory location of a node that does not exist due to mismatched paths.

Novice programmers may make a similar mistake with the find function, as its algorithm has a similar structure to the algorithms for insert and helper. Suppose the student correctly implements the insert and helper functions. In the case where the find function swaps the less than and greater than signs, find will always return false because it will take the wrong path direction to find the value at any node. The only exception to this is if the target value is at
the root, but as long as the user thoroughly tests the program, the bug should be easy to identify in the visualization.

When writing recursive functions, students frequently make the error of forgetting to include a terminating condition. In many cases, this results in an infinite loop type of error. Methodology subsection III-O discusses infinite recursion errors. When a particular base case is optional, missing the base case results in an incorrect solution, but the code does not produce an infinite loop. Suppose the program instructions indicate that there should be no duplicate values in the tree. In that case, if an insert function does not return false when a value already exists in the tree, the visualization results are incorrect even if the slideshow does not display any immediate errors. However, subsequent C++ function calls may return unexpected results because the visualization expects a tree with no duplicate nodes.

Another commonly repeated error, similar to errors encountered with linked lists and arrays, is code that tries to access memory beyond the data structure boundaries. In a tree implementation, this may manifest itself as a request for the left node’s value instead of the current node’s value. If the programmer makes this error with the insert function and the current node does not have a left node, the program is trying to access memory to which it does not have access. If the programmer implements insert correctly but makes this error in the find function, the code may not find a node that exists in the tree, as shown in Fig. 13. The program will check memory locations beyond the current node and potentially reach null before finding the target.

After implementing the insert, helper, and find functions, the student should implement the print functions that visit each node in the tree using in-order, pre-order, and post-order methods of traversal. A recursive binary tree function often contains two recursive calls, one to visit the left subtree of the current node and one to visit the right subtree of the current node. Beginning programmers may make the mistake of copying and pasting the line of code that makes the recursive call but forget to update the copied line of code to check the other subtree. When the user calls the print function, the slideshow displays the tree values based on the order in which the given C++ traversal implementation says to visit the nodes. The slideshow highlights the tree nodes in the correct order according to the specified traversal. If the user-written code
only visits one side of the tree, it should be clear from the visualization and provided messages that there is an error within the code.

Other potential traversal mistakes will also be visible based on the results shown in the slideshow. Suppose the program visits nodes in the opposite order that they are supposed to be visited. This code does not produce an error, but the result printed from the C++ code will not match the visualization since the JSAV tree nodes get highlighted in the specified traversal order. An off-by-one error could occur in a traversal if the program checks if the left subtree exists but then visits the right subtree or vice versa. As shown in Fig. 14, if the current node only has a left child and the right child is null, the result printed from the C++ code will include zeros, representing the null sibling, in place of the actual subtree values.

The next BoxAndArrow Tree exercise is to count the nodes to determine the size of the tree. One solution starts by checking if the tree is empty. If not, the function returns one plus the result of a recursive call to the left subtree plus a recursive call to the right subtree. The code is concise, but it can be challenging for students to understand the method and why it works. Similar to other data structures that utilize dynamic memory, a frequent error is forgetting to
account for the case when the tree is empty. If the tree is empty and the function does not return immediately, it may return a size greater than 0. Sometimes incorrectly implemented recursive calls may return the correct result. However, if there is a problem with the implementation, the user will run into a case when the count returned from the C++ size function does not match the number of nodes found in the visualization. As long as users test various inputs, they will eventually come across an error message indicating that the user’s results do not match the expected results.

The last Binary Tree Visualizer exercise is a function that checks if the tree is balanced. The tree created in BoxAndArrow is not required to be a balanced binary tree, so the balanced function checks if the tree is height-balanced. For a binary tree to be height-balanced, the heights of the left and right subtrees of every root must not differ by more than one. Some calculations of height count the maximum number of edges from the root to a leaf; this implementation counts the maximum number of nodes from the root to a leaf. Each root is itself a tiny tree with one left subtree and one right subtree. This exercise specifies that the function should return the tree height if it is balanced and -1 if the tree is not balanced. One of
the tricks is to make sure to check that all subtrees are balanced. If any recursive call returns -1 or the absolute value is greater than one between the height of a node’s left and right trees, the tree is unbalanced. If the code misses any case, the slideshow will not print an error message, but the error will be visible. As shown in Fig. 15, if the tree is unbalanced yet the code returns the height, a message appears in the slideshow that includes the height returned from the function, indicating a balanced tree. The user can examine the tree displayed in the visualization and determine if the result looks accurate.

**Fig. 15.** Example of the slideshow displaying inaccurate results due to a missing base case for a recursive function.

**O. Recursion and Infinite Loop Error Checking**

Some common recursion mistakes are forgetting to include a base case, not covering all possible base cases, or including a recursive step that does not reduce the problem size. All of these errors result in infinite recursion or behavior that is similar to an infinite loop. In native C++, executing a program with an infinite loop requires the programmer to terminate the program with an interrupt signal in the terminal. In the browser, JavaScript produces an exception, and a message is printed to the web console [76]. The Firefox error message is "Internal Error: too
much recursion.” The Chrome error message is ”RangeError: Maximum call stack size exceeded.” The user never gets to run the program because the error is determined almost immediately after the user clicks the button to start the visualization. All of the tree member functions call recursive helper functions, except for the constructors and destructors. Thus, the programmer runs a high risk of encountering a recursion error at some point while creating their program. There is no simple way to catch this error in the JavaScript code; however, a message about these errors is included in the Binary Tree Visualizer README file to provide users with as much information as possible.

P. Compilation Errors

The primary tool used to compile C++ to WebAssembly is the LLVM-based Emscripten compiler toolchain [24]. Novice students sometimes struggle with compilation errors and compiler messages that are challenging to interpret. However, other times compilers offer handy information and direct the user to the specific line of code where the error took place. While compiler message overall could be improved, addressing this issue was not the goal of this research. Emscripten provides error and warning messages similar to those provided by the well-established GCC and Clang compilers. Instead, this project focused on catching runtime errors and helping students identify mistakes that do not produce error messages.
IV. DISCUSSION, CONCLUSIONS, AND RECOMMENDATIONS

A. Limitations

As discussed in Section III, BoxAndArrow has some limitations as a debugging tool, partly due to the underlying structure of the tools used to create it. Several common mistakes made by students writing native C++ code do not appear as errors in an application built using WebAssembly and JavaScript. In some cases, these errors may be impossible to catch due to the differing ways that these languages allow programmers to access memory. Research and development included consideration for frequent student mistakes made with all included data structures [77], [78], [79]. However, it is impossible to account for all potential bugs that a person may encounter while programming, so it is very likely that future work on this project will uncover new errors to address.

When working with dynamic memory in C++, segmentation faults are a regularly seen error. BoxAndArrow attempts to catch these types of errors wherever possible. This project can identify segmentation fault errors that arise when a program tries to access memory beyond the bounds of the data structure or when it tries to access a node that does not exist. When a program accesses memory not allocated to a node in a linked list or binary tree, it results in an error message displayed in the slideshow. However, code that tries to access a null pointer’s value produces a segmentation fault in standard C++ but no clear error in this application. This inconsistency appears because the 0x0 address in WebAssembly memory is a valid memory location, and the code that runs the visualization can only see the value returned from the function. As long as the rest of the code is correct, this error goes unnoticed.

Memory leaks occur in a computer program when the programmer does not correctly manage dynamically allocated memory. It can be challenging to identify memory leaks even in native C++ programs because memory leaks do not cause the program to function incorrectly for most test cases. The memory available on the computer diminishes and could eventually cause applications to slow down or fail; however, none of these problems are easily identifiable by running the code with small examples. JavaScript utilizes a garbage collector to manage memory automatically. Nevertheless, in a program that binds C++ objects to JavaScript, an object must be
explicitly deleted to ensure that the Emscripten heap does not grow indefinitely. BoxAndArrow calls delete on all C++ modules allocated in JavaScript, but freeing this memory relies on C++ destructors functioning as expected. If the programmer has not correctly used the delete operator to free all dynamically allocated objects, this could cause a memory leak. Several tools exist for detecting memory leaks; however, these options are not perfect, as they require extra memory and cannot find all memory-related errors [80].

Another challenge with this project is that it requires a fair amount of setup but intends to be used by novice programmers. In an academic setting, a standard programming course may already require students to set up several technologies, so instructors need to be mindful of how much extra work is required of students just so that they can complete their assignments. One way to address this problem is by providing thorough documentation and instructions to walk users through the steps to download and install various dependencies. Even if there are many steps to understand and implement, most students can follow detailed instructions, and they will ultimately gain confidence from successfully setting up their environment. Another solution could be to write an installation script that would automatically perform any actions necessary to install dependencies.

This program does not require students to write a full-fledged web application. Still, anything that incorporates web technology includes extra steps and challenges that a novice programmer would not run into while working with a language like C++ in a native or integrated development environment. The process to compile and run code differs when the user intends to execute it on the web, and sometimes changes made to the code are not immediately apparent in the browser. This issue is often due to the web cache, which stores files, images, and other browsing data to improve performance. When programmers update their code, the page may reload with a cached version, so the changes will not be visible. A couple of options exist to address this problem and ensure that the browser visualization uses the latest version of the code. The user may need to leave the web console open with the cache disabled or go into the settings and manually clear the cached data. Once programmers become comfortable with these requirements, the issue is easy to handle; however, if the programmer does not identify this issue right away, it can cause significant frustration. Future work could reconcile this issue by
integrating the tool into a hosted web service. Students could submit their code to this service rather than set everything up locally. Despite these extra challenges, a program that introduces students to web programming and some of the issues encountered while developing for the web could benefit students in introductory programming courses. Learning new tools can give students confidence and encourage them to try new technologies [62]. On the other hand, incorporating web technologies into a course where the web is not a primary focus may prove to be too much additional work and information for students, thereby potentially limiting the benefits of this application.

Due to limited time and resources, this project did not incorporate human subjects for testing the prototype application. Incorporating human subjects in research requires several extra steps for approval and monitoring, and these steps were beyond the scope of this project. Additionally, the timeframe for completing this project did not allow for adequately engaging in the necessary formal processes. However, gathering user input and evaluating the results is a goal of future research. This project focused on developing an application that utilizes WebAssembly and JSAV. It also determined ways to improve the usability of this type of application for novice programmers. Testing this application with human subjects requires the project to be at an appropriate completion stage before the first day of a school semester. It would also be necessary to create a plan for integrating the material into a data structures course and gathering data throughout the semester.

B. Conclusions

This paper described a prototype application for an algorithm visualization system and its process of development. The application provides users with instructions and starter code to create C++ programs for commonly utilized data structures. Using WebAssembly, the project maps user-written code to a JSAV-based algorithm visualization that students and instructors can view in a web browser. The visualization provides students with a straightforward process for testing their programs, allows them to use various combinations of input and commands, and provides error messages and other visual components to help identify bugs in a C++ program. Although this research did not involve human subjects, having a hands-on role in constructing
visualizations provides a significant benefit to the user, based on literature review and personal experience.

As described in Section I, educators tend to agree that AVs and PVs provide positive benefits to novice programmers learning data structures and algorithms. However, passively viewing AVs has fewer benefits and could even be detrimental. The most crucial factor is that students play an active and engaged role in the process. It is particularly beneficial to incorporate various levels of increased user engagement beyond simply viewing. An interactive visualization can serve as a valuable tool for helping novice programmers genuinely understand how data structures and algorithms work.

In BoxAndArrow, students write programs that directly impact a data structure visualization and an associated slideshow. The program is usable for anyone with a personal computer and a browser. Students can easily create tests with various combinations of inputs and commands. The web-based visualization provides a clean canvas on which students can present their visualization and explain to others how their code works. When students have written code that interacts with the visualization, it increases content ownership as well. Many AVs and PVs are currently available on the web. However, AVs do not generally work with student-written code, and most PVs require students to write code in a web-based code runner, not in the student’s typical programming environment.

BoxAndArrow provides users with a role in creating algorithm visualizations while incorporating some features of program visualizations. While students write and test their code, slideshow messages and visual cues from the data structure visualization help identify programming errors. The debugging information is not as granular as it is for a standard program visualization, but it does help direct students to the function where the problem occurred. The visual can also highlight mistakes that students might overlook if they were running the program in a terminal and only looking at command line output. After students have completed their code and determined it is working as expected, the visualization shows what happens in the data structure at a high-level, similar to an algorithm visualization. Stepping through the slideshow reinforces the steps of the algorithm, helping students confirm their understanding.

Various eTextbook options provide a more comprehensive set of material for educators
than this project; however, many instructors do not want to switch over to all-online content completely. BoxAndArrow aims to work as a supplement to other instruction. Its exercises would work well as lab assignments, giving students more experience working with data structures before applying their knowledge to more substantial programming assignments. Students could complete most of the work with a text editor, similar to the environment that most professional developers use to write code. Students receive experience compiling and running code from a terminal but can see results visualized in a data structure that uses their program.

This project includes several small exercises that an instructor could utilize as lab assignments in a CS2 data structures course. It could also serve as optional extra credit or a good side project for students after completing a CS2 course. Sometimes students can pass a programming course even if they do not have a strong understanding of every topic covered in the class. Only later, after many hours spent coding, assisting other students, and working on projects for school or jobs, do they feel like they have a firm understanding of the material. A project like BoxAndArrow could reinforce essential concepts about data structures and algorithms outside of the classroom or alongside other content while introducing students to WebAssembly and Flask. These web technologies could come in handy for future projects or help them become more comfortable with different development tools.

C. Recommendations and Future Research

Instructors and other individuals interested in using this project for academic applications or personal development should start by downloading BoxAndArrow from GitHub and creating a private repository for their implementation. Several other dependencies will need to be downloaded and installed for Emscripten, JSAV, and Flask. The BoxAndArrow README files provide instructions for setting up the environment, compiling and running the code, completing the exercises, and using the browser-based visualization tool. BoxAndArrow is an open-source project utilizing the 3-Clause BSD License. Potential contributors can fork the repository and create an issue or pull request.

An essential step for future research would be to incorporate the BoxAndArrow application into a programming and data structures course, monitor student progress and results, gather
feedback from surveys, and evaluate the outcomes. Researchers could also gather experimental data by offering the application as an optional project that students complete alongside their coursework or after the semester. Sometimes students want opportunities to work on projects outside of their regular classwork to improve their understanding of the material. The more experience students have coding and working with new tools, the more confidence they will gain and the more prepared they will be for job interviews and future experience in the workforce. This option could utilize the same evaluation strategies but allow students to experience the project in a lower-pressure setting. It could be helpful to have a trial run with a small set of participants and ask them to provide user experience feedback, identify bugs, or provide information about how the instructions could be improved.

This application could also be beneficial for faculty interested in presenting algorithm visualizations during lectures for DSA courses. Instruction often involves diagrams written on a whiteboard, but these can get messy or include mistakes if not careful. BoxAndArrow allows instructors to effortlessly move back and forth within a visualization displayed in a web browser. It also provides the ability to test many combinations of inputs and commands, even when only a small time frame is available.

Additionally, it would be beneficial to account for more potential student errors and expand the topic coverage provided by this application. While arrays, linked lists, and binary trees are relatively uncomplicated data structures, novice programmers still have difficulty understanding the concepts. A project covering these topics benefits students beginning their programmer experience, and adapting it to provide visualizations for more advanced data structures could make it a more comprehensive educational tool. JSAV already supports data structures such as graphs and matrices. Alternatively, more examples could be created with the existing data structures so students could gain a better understanding of the wide variety of uses for these foundational concepts.

D. Summary

Many students struggle with introductory programming courses, which often results in high failure and dropout rates for computer science degree programs. Incorporating software
visualization tools will not solve this problem for all students; however, it can help those who begin CS1 or CS2 without a solid understanding of how data structures and algorithms work but are interested and willing to put in the effort necessary to learn the material. AV, PV, and other visualization tools can help novice programmers learn and build knowledge, especially when incorporated into a system that offers many opportunities for active learning.

BoxAndArrow takes a novel approach to software visualization by providing an environment for students to engage directly with an AV. The application gives students the responsibility to write programs, build tests with different inputs and commands, and either confirm that their code works or identify potential problems within their code. The application combines WebAssembly, a new web technology standard, and JSAV, a widely used, open source JavaScript algorithm visualization library. Students write C++ code that compiles to WebAssembly, enabling C++ programs to execute in a browser. The program then interacts with a corresponding JSAV slideshow. The visualization functions as expected if the program is written correctly and provides error messages or other visual cues if the code contains a bug. As students test their code, they are encouraged to pay attention to what happens in the visualization and figure out how to solve problems as they arise.

Educators generally have favorable views on AV and PV, but many do not use them regularly in the classroom. Full-service eTextbook options exist; still, some instructors prefer to teach with course material they have already created, and others do not have the time to switch to an entirely new program. BoxAndArrow is an application that could supplement existing course content and provide the basis for several lab exercises or other small assignments that students could complete throughout a semester. Many higher education institutions teach C++ in CS1, CS2, and other DSA courses, so a system that uses WebAssembly to compile C++ into a program that can run on the web could have extensive applications.

BoxAndArrow is an open source repository available for free via GitHub. Instructors, students, and anyone interested in improving their understanding of data structures can use, adapt, or expanded this project. The repository includes instructions, exercises, hints, and starter code for arrays, linked lists, and binary trees. There are many opportunities to build on this application by developing visualizations for more complex data structures, improving error detection, and
creating additional commands for existing algorithms. Computer science education aims to produce graduates who have the skills to program, an interest in solving complex problems, and a solid knowledge of core computing concepts. BoxAndArrow works toward these goals. The project uses algorithm visualization to strengthen understanding of computational foundations, provides opportunities for students to write code and develop programs, and promotes content ownership and engagement with class material.
REFERENCES


APPENDIX A

EXAMPLE OF AN ARRAY VISUALIZER SLIDESHOW

The following figures provide a step-by-step example of a BoxAndArrow Array Visualizer slideshow for determining if an array is sorted in ascending order. The visualization uses two pointers to compare neighboring values, highlighting the values if the current pair is not in sorted order.

Fig. 16a.

Fig. 16b.
Fig. 16c.

Fig. 16d.
Fig. 16e.

Fig. 16f.
APPENDIX B

EXAMPLE OF A LINKED LIST VISUALIZER SLIDESHOW

The following figures provide a step-by-step example of a BoxAndArrow Linked List Visualizer slideshow with two elements getting removed from the list, one specified by the user and one removed from the head. A pointer keeps track of the head of the list while a highlighted node steps through the list.

Fig. 17a.

Fig. 17b.
Fig. 17c.

Fig. 17d.
| Fig. 17e. | Fig. 17f. |
APPENDIX C

EXAMPLE OF A BINARY TREE VISUALIZER SLIDESHOW

The following figures provide a step-by-step example of a BoxAndArrow Binary Tree Visualizer slideshow in which six elements get inserted into the tree. A highlighted node steps through the tree, showing how the user can find the correct insertion location without visiting every node in the tree.

Fig. 18a.

Fig. 18b.
Fig. 18c.

Fig. 18d.
Fig. 18e.

Fig. 18f.
Fig. 18g.

Fig. 18h.
Fig. 18i.

Fig. 18j.
Fig. 18k.

Fig. 18l.
Fig. 18m.

Fig. 18n.
Fig. 18o.

Fig. 18p.
Fig. 18q.

Fig. 18r.
Fig. 18s.

Fig. 18t.
Fig. 18u.

Fig. 18v.
Fig. 18w.

Fig. 18x.