TreeViz

Final Report

CS 4624 Multimedia, Hypertext, and Information Access Capstone

Spring 2019

5/8/2019

Instructor: Dr. Edward A. Fox

Virginia Tech, Blacksburg, VA 24061

Authors

Conner Caprio

Moira Pelton

Austin Zensen

Client

Mostafa Mohammed

Table of Contents

Table of Figures 2

Abstract 3

Introduction 5
- A. Goals and Objectives 5
- B. Scope 5
- C. Client Background 6
- D. Major Constraints 6
- E. Organization of Report 6

Requirements 8
- A. Backend Support for Tree Visualizations 8
- B. Frontend Support for Tree Visualizations 8

Design 9
- A. Backend Design 9
- B. Frontend Design 11

Implementation 12

Testing and Evaluation 16

User Manual 17
- A. User Experience 17
- B. Running Tree Visualization 17

Developer's Manual 20
- A. Installation and Setup 20
- B. Project Structure 22
- C. Adding to this Project 24

Lessons Learned 25
- A. Timeline / Schedule 25
- B. Problems and Solutions 26
- C. Future Work 26

Acknowledgements 28

References 29

Appendices 30
- A. VTURCS Symposium Poster 30

Table of Figures

Figure 1: Interaction between CodeWorkout and OpenPOP Server……10
Figure 2: Step-Through of Sample Tree Visualization……..…....13
Figure 3: Step-Through of Sample Tree Iteration Visualization……..……14
Figure 4: Sample Tree Student Code Visualization……..……15
Figure 5: Binary Tree Check Value Exercise in CodeWorkout……..……..18
Figure 6: Test Case Results of Sample CodeWorkout Exercise……..……...19
Figure 7: GitHub Repository for Project Code……20
Figure 8: Main Directory for Project Code……....20
Figure 9: Location of jsavexample.html file……..21
Figure 10: Dual View of Visualization and JavaScript Code in Browser Window…….……22
Figure 11: Execution Trace of Sample Student Code…….……...23

Abstract

The TreeViz project was developed as part of the capstone course CS 4624 Multimedia, Hypertext, and Information Access and was completed over the course of the Spring 2019 semester. The goal of this project is to allow students taking the CS 2114 data structures classes to visualize code written for tree data structures through CodeWorkout homework exercises. Students will be able to make changes to their code to a tree data structure in real time as they click through visualization steps in the existing interface and our changes to accommodate tree data structures. This project is an extension of the existing work of the client, Mostafa Mohamed, to create visualizations for linked list data structures for exercises for the CS 2114 and 3114 data structures classes.

This project was implemented in two main phases, with the first goal of generating a tree visualization from student code and the second goal of being able to visualize changes to the tree data structure as the student code executes. We wrote a set of JavaScript files for the OpenPOP (OPEN Pointer OPerations) server that is used by OpenDSA and CodeWorkout to generate and parse the execution trace of student code for changes to the underlying data structure. Our solution analyzes the execution trace of the student's tree code and determines the pointers created during execution to build a replica of the tree data structure being created in memory in a stack trace. With each step in the execution, our code will determine the pointers that make up the tree and draw the tree using the JSAV library to provide the physical representation of the nodes and branches within the tree data structure. This way, the student can view and step through the execution of their code and visualize the changes to their tree data structure.

Our final visualization can replicate a given tree used to test a student's sample tree exercise code and step through the code's execution showing them the changes made to the tree as a result of their solution. The visualization also provides a pointer to the current location of the root of the tree for exercises that require recursion for tree traversal, like the sample exercise we used to test our visualization. The student will also be able to see their code alongside our visualization so they can step through the visualization to see what each line of code specifically changes to the tree in memory. In the future, our work will be able to handle all tree operations, and can be expanded to provide visualizations for many more data structures taught in these courses and ultimately help to improve students' understanding of the concepts taught to them in class.

Over the course of completing this project, we learned many valuable lessons about working with teams and clients in a professional setting, incremental deliverables for large-scale projects, time management skills, and working with new technologies and systems. We spent a few weeks at the beginning of the semester just learning about the

existing code base from our client in order to gain a full understanding of the existing system such that we could build upon it with this project. We also gained new skills in JavaScript, Ruby on Rails, JSAV, and much more in the course of determining what we needed to add to the existing server code in order to effectively implement our changes. Working on this project with our client over the course of the semester was not just good experience for us in completing real world technology projects with clients, stakes, and deliverables, but also the result of our work this semester will hopefully provide a useful and informative resource for many students to come in the Computer Science department at Virginia Tech.

Introduction

A. Goals and Objectives

The objective of this project is to add support for visualization of tree data structures to be used in homework exercises in CodeWorkout [1] for students in the CS 2114 Data Structures class. Students in this class are learning about how to build data structures in Java, and as a part of learning how to properly write data structures it helps for them to have a visual guide to see a physical representation of the virtual data structure to see how their code will impact the data structure in memory as it executes. Not only will this physical representation increase their understanding of data structures, but it will also help them to find errors in their code and appropriately fix them by seeing their code execute in real time.

The client, Mostafa Mohamed, has already created the appropriate system for visualizing the execution of students' code for this class through the OpenPOP (OPEN Pointer OPerations) server to help visualize basic pointer functions and the JSAV library [2] to create visualizations for data structures. Currently, the server has the ability to visualize code exercises for linked list data structures, and our continuation of the project will build upon the existing system to add functionality for visualizing tree data structures. This will enable students in these classes to utilize this tool to learn more about how tree data structures work and see their tree data structure code execute and find errors within this code.

B. Scope

This is a semester-long group project to be completed in a reasonable time by the end of the spring semester. To keep ourselves on track throughout the project and space out deliverables, we developed a comprehensive timeline alongside our client at one of our first meetings for our work during the semester. We spaced out our project into one to two week periods at the end of which certain deliverables would be due. We also took into account certain events throughout the semester, like purposefully scheduling less group work during Spring Break since we would be out of town and unable to meet. In addition to the project deliverables, we were also required to give three intermediate presentations on the progress of our project to the class, and a final presentation at the end of the semester detailing the work we completed throughout the semester. We will give additional presentations of our project at the VTURCS Spring Symposium and ICAT Day, both at the end of the semester after our project has been completed. For more details of our project timeline and schedule, see the project timeline table in Section A of Lessons Learned later in this report.

C. Client Background

Mostafa Mohamed is a graduate student at Virginia Tech and is the professor for Virginia Tech's Formal Languages course CS 4114, and a former graduate teaching assistant for CS 1054, 2114, 3114, and 4114. Mostafa graduated from Assiut University in Assiut, Egypt with a bachelor's degree in computer science in 2005. He then got his masters degree from Cairo University in Cairo, Egypt in 2011. Mostafa started the visualization project in in the Fall of 2016 with the goal of providing students in data structures courses a tool to help increase their understanding of the course material through visualizations of different data structures and their functionality.

D. Major Constraints

One of the main constraints for this project were related to the languages used in the backend implementation of the existing code that the tree visualization code will be built upon. A couple of our group members had never coded in the JavaScript language, so they spent some time in the first two weeks of the semester to learn the basics of this language before we began editing the code base. Another knowledge constraint was that none of the project group members had ever used Ruby on Rails [3], so the entire group decided to learn the basics of this language over our spring break in case the project implementation required this knowledge.

Another major constraint for this project had to do with the operating system required to run the OpenPOP server. While the OpenPOP server requires a Linux or Mac OS to run, all of the project group members' machines ran on a Windows OS. Furthermore, two of the group members were also enrolled in a class which required them to disable virtualization on their machines, which meant that they could not run a Linux virtual machine on their computers for the entirety of the semester without complications arising. With these challenges in mind, we made a mutual decision with our client that he would run the server on his machine and send us the execution trace of student code generated by the server, and then we would base our implementation off of these execution traces.

E. Organization of Report

This report contains twelve sections in total. It starts with the abstract and is followed by an introduction to the project and client. Following the introduction will be the requirements for the project and the design of the project. After this, there will be a summary of the implementation of the project and the testing we performed on our final product. Directly after reviewing the testing of our project the report will contain a user manual to provide a guide for how people can use our project, and a developer manual

containing all information needed to build upon this project in the future. Finally, the report will end with the lessons learned throughout the project, acknowledgements, and any references used throughout this report.

Requirements

A. Backend Support for Tree Visualizations

The main purpose for this project is to generate tree data structure visualizations, which relies on a backend implementation to generate the trees which CodeWorkout can visualize in the frontend to show students the visual execution of their code. This entails writing code to support tree data structures inside the OpenPOP (OPEN Pointer OPerations) server, which is a server that supports the functionality to visualize basic pointer operations for data structures. The OpenPOP server will take the student's code written through CodeWorkout as input, then run this code and generate an execution trace with which to determine the changes performed upon the given data structure. OpenPOP already supports the functionality to create visualizations for linked list data structures, so this project requires similar functionality to be developed for tree data structures.

B. Frontend Support for Tree Visualizations

In addition to the backend support for tree visualizations, we will also need to ensure the front end support so that our backend code can actually be visualized for students using the tool. The linked list visualizations that OpenPOP already supports are supplemented by the JSAV library, which is a JavaScript library for visualizing data structures. Through the use of this library, the visualization tools in CS 2114 homework exercises can step through a student's code and provide a visual depiction of the linked list data structure in memory so the student can see how their code is changing the structure and find any errors within their code. While the backend code in the OpenPOP server actually reads through the execution trace of this code and determines what changes occur between each line of code, the frontend support will actually draw out data structure using the JSAV visual tools so the student can see the data structure in their browser window next to the code generating it. With the addition of support for tree data structures, our client was unsure if the existing frontend support would be sufficient to visualize the trees; however, because the visualization is the key goal of the project, frontend support is a major requirement even if no changes are necessary to the existing code base to support tree visualizations.

Design

A. Backend Design

The backend design of the system is an object-oriented design that is based upon the design of the linked list visualization code originally written by Mostafa Mohamed. A conceptual visual of what the backend system looks like and how it functions is contained in Figure 1. OpenDSA [4] and CodeWorkout both utilize the OpenPOP server, which will run student code passed through to it from either OpenDSA or CodeWorkout, generate an execution trace of that code containing all pointers and variables in memory, and then determining the pointers updated within a data structure in order to draw a visualization through the JSAV library. The execution trace generated by OpenPOP comprises two main components: a heap and a stack that contain information about all of the pointers currently contained in a data structure, the links between those pointers, and any variables in memory. What we will be adding to the OpenPOP server is a set of JavaScript files for trees, similar to the files already developed by our client for linked list data structures, that have the functionality to parse this execution trace, determine all of the pointers and links in the tree, and communicate back to CodeWorkout and the JSAV library what visualization to draw on the student's browser with the homework exercise.

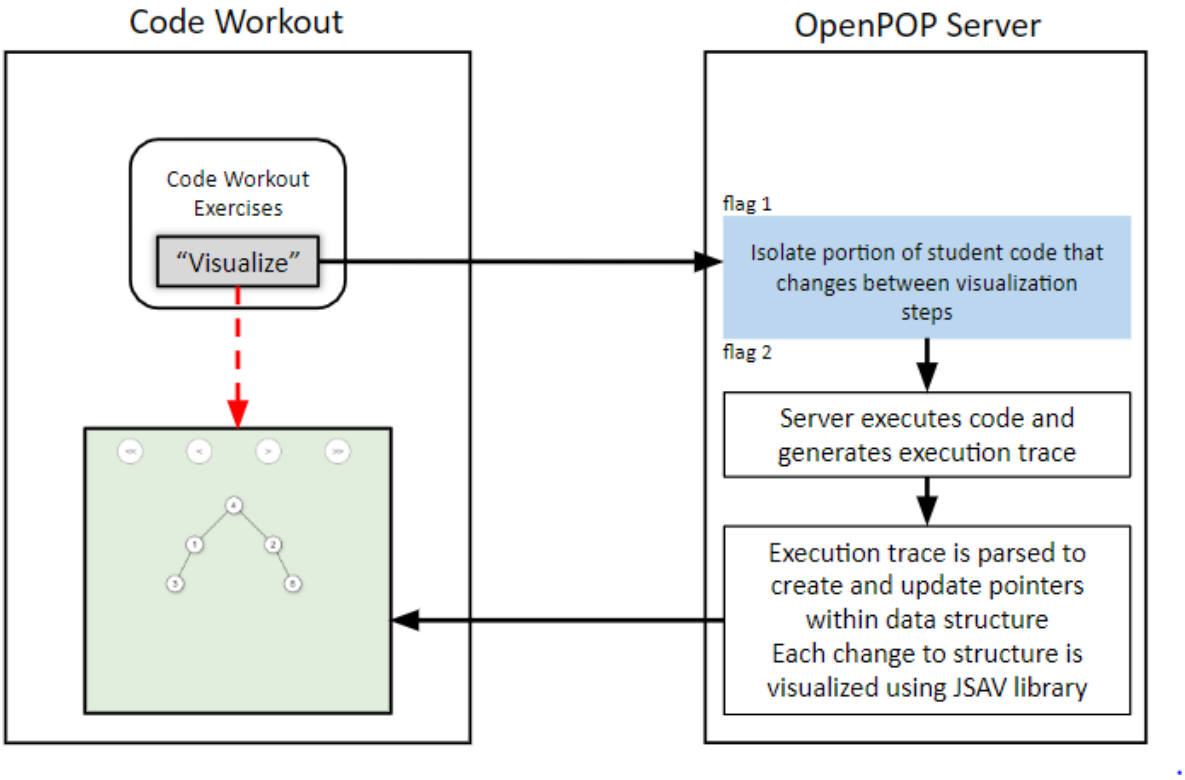


Figure 1: Interaction between CodeWorkout and OpenPOP Server

The most crucial feature of our design for the tree visualization functionality we are adding to OpenPOP is a difference check that is done so that it can detect a change that occured between lines of code when a student is visualizing an action in a tree data structure. This functionality will allow OpenPOP to determine the changes that occur between each line of code submitted by a student by parsing the execution trace generated, and then using the JSAV library to physically draw the nodes and links in the tree and update any node additions, deletions, or tree traversals represented in the student's code. For example, if a student were visualizing adding a node to a tree, this method would take in the trace generated from the student code and calculate the difference between the tree before a new node is added and after it has been added in. The way we determined we would design this method was based on how our client designed a similar method for the linked list data structure visualizations, which is to simply redetermine the pointers in the tree after each new step in the execution trace. While this design is not the most efficient method of calculating this difference, it ensures that all pointers are correct with each new line of code being executed and guarantees that the student is seeing the correct version of the tree visualized in their browser window.

B. Frontend Design

The tree visualizations will be accomplished with the help of the JSAV library, which our client used to help visualize the linked list data structures already supported by CodeWorkout. The frontend aspect of our design will be heavily incorporated with and dependent on the backend of our system in OpenPOP since we will be able to make calls to the JSAV library directly within our JavaScript code to draw the current tree determined through the execution trace. Because our backend design is such that OpenPOP will recalculate all of the tree pointers with each new line in the student's code, JSAV will redraw the entire tree so that we know the tree visualized is correct based on the current step within the code.

After finishing the visualization capability, we also added some extra features to the final visualization by adding in a label and arrow that designates wherever the current root is in the code and points to that location in the tree visualization. We added this visual feature because it will help with exercises that involve tree traversal and search, as they are usually implemented through recursively travelling down a tree by way of passing a child of the current root in the tree through and treating that child node as the temporary root node of its own subtree. This arrow label will update with each step through the submitted code to show the student where the root variable is currently pointing in the tree so they can check whether or not their code is recursing through the tree correctly based on the exercises' intended purpose. We also added the student's code alongside the tree visualization in the browser window so the student can draw the connection between the location in their code currently being represented by the visualization. This feature will be immensely useful to the student as they will be able to follow along line by line in their code and view the changes made to the physical representation of their tree in the visualization.

Implementation

A. Visualization of One Execution Step

We implemented the code to create the tree visualizations for this project in a set of JavaScript files, beginning the the files our client had previously written to create the linked list visualizations in CodeWorkout. While all of the files are necessary to run the visualizations, most of our changes are contained in the JsavWrapper.js file contained in our Github repository for the project. We followed the structure and design of how our client designed the linked list visualizations closely for our implementation; this will make it so that both implementations for the visualizations are similarly designed so that it will be easier for someone to continue to build upon the project in the future.

To begin the implementation of the visualization, we wrote the code for a generic tree data structure and its nodes in JavaScript. This goes in hand with our client's object oriented design for the linked list visualizations, which uses a linked list object to actually create a linked list object from the nodes and pointers contained in the execution trace. From the generic tree object, we will be able to create a tree in JavaScript from the pointers in the execution trace, and then use the JSAV library to draw the tree object in the CodeWorkout browser window.

After creating the generic tree object, we wrote the main loop that parses the execution trace, builds a JavaScript tree from the pointers it finds in the trace, then calls the JSAV library to draw the tree in the front end. The execution trace of student code contains a stack and several heaps, where each heap indicates a change that has occured in the tree data structure in memory. Within each heap is a list of nodes in the tree, each with a number label that acts as the identifier of that node in memory, and the left and right pointers of that node. Since each node has a unique number attached to it, we can create tree node objects for each node and determine where it exists physically in the tree based on whether or not any other nodes point to that specific number node. The program will build up a tree object with all of these nodes based on this information contained in the heap, and when it has parsed through one heap, it can call the JSAV library on the tree object to draw it in the browser. The JSAV library already has the functionality to create the visualization if we provide it the object to draw, so we did not need to add any other specific functionality to create the front end visualization other than making a call to JSAV.

B. Visualization of Multiple Execution Steps

After implementing a basic tree visualization based on one state of the student's code execution, we needed to provide functionality for visualization of the whole execution of

the code. This meant that we needed to create a loop that would parse through each heap contained within the execution trace of the student's code, and continue to visualize each heap until the end of the trace. To provide for this functionality, we had to alter our initial code by providing for visualizations where the tree did not actually change between visualization steps. That is, for exercises where the tree remains static but the visualization is showing the traversal of the tree, our code would be much more efficient if it simply checked for whether or not the current heap within the trace was the same as the previous heap state visualized. By checking whether or not the visualization has actually changed between visualization steps, our code can run faster by just keeping the previous visualization up and simply changing the pointer to wherever the current root is for these traversal exercises. Figure 2 depicts the visualization of code that adds nodes to a tree in Java, with each step labeled to clearly show the progression of the code.

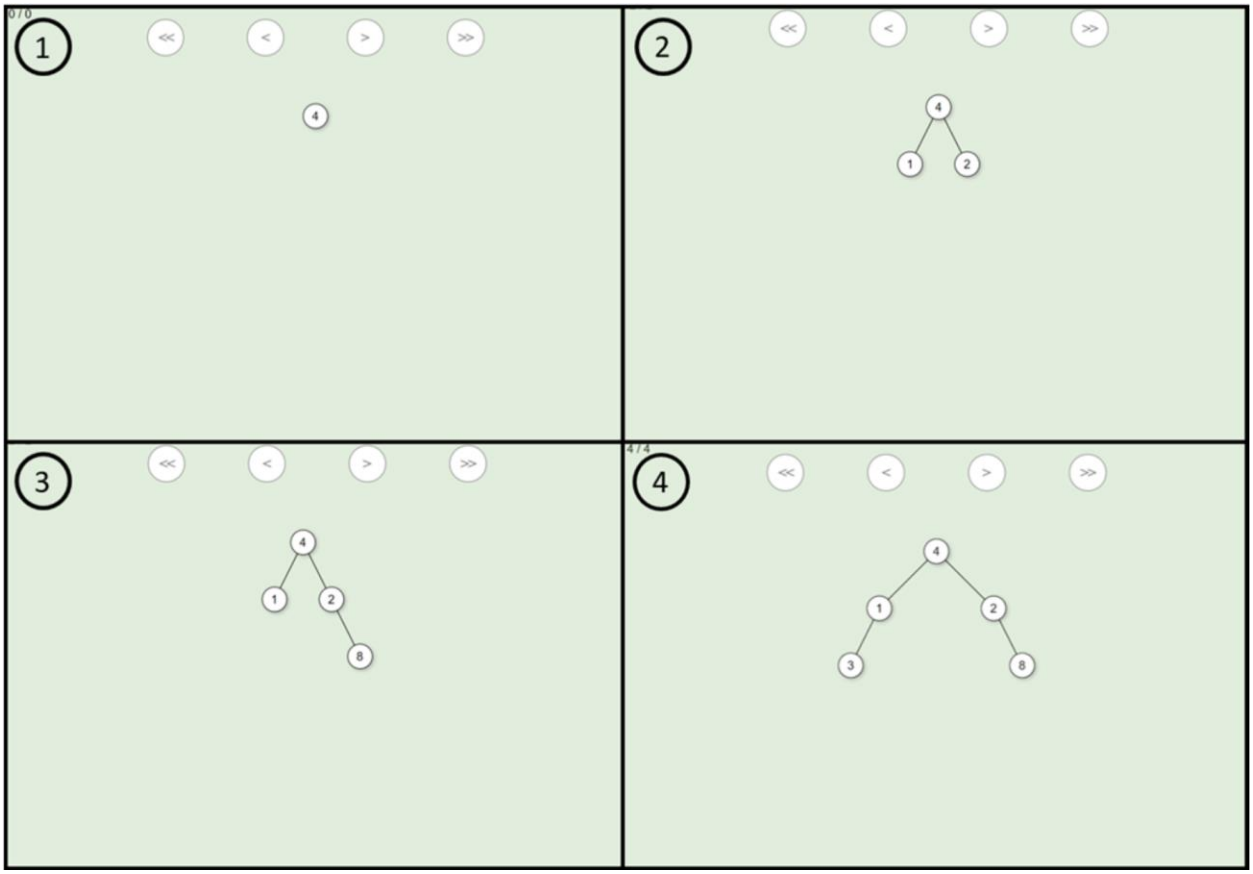


Figure 2: Step-Through of Sample Tree Visualization

With the functionality for visualizing each execution step in the execution trace, we also provided the root pointer through JSAV that will draw an arrow to and label the current root at each step in the student's code. This functionality is helpful with tree traversal

exercises where the student needs to see the location of the current root in the tree, and the sample execution trace we based our testing off of was one of these exercises. Since the execution trace keeps track of the root in each execution heap, we accomplished drawing this label onto the final visualization just through making a call to the JSAV library to draw the label to the right location. A visual of this root pointer label in our final product is shown in Figure 3.

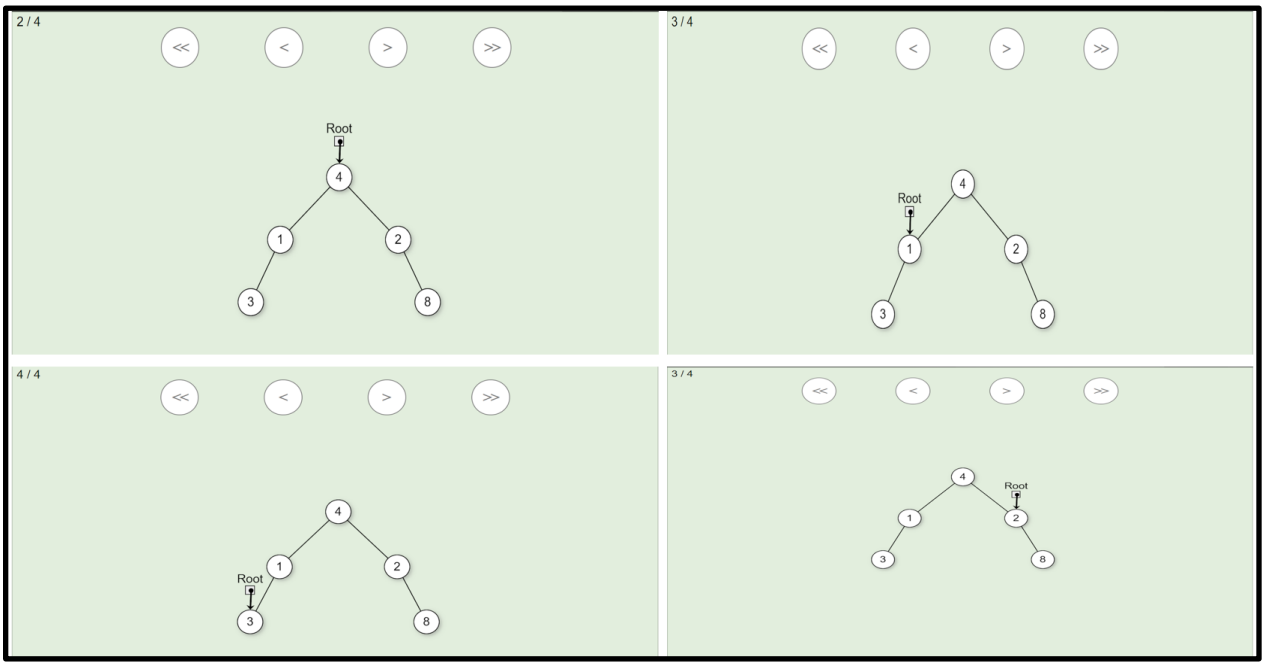


Figure 3: Step-Through of Sample Tree Iteration Visualization

We also included the student code alongside the visualization at this page, through the StudentCode.js file which contains methods to represent the student code solution provided through CodeWorkout. Our final visualization, along with student code shown by its side, is shown through the picture in Figure 4. Since the functionality of these methods were not specific to the linked list visualization, we were able to use it without much change in order to visualize the student code alongside the tree visualizations. This functionality will enable students to step through the visualization and see which line of code is currently being visualized in the tree diagram, which will help them to find errors in their code more easily than just looking at the tree visualization on its own.

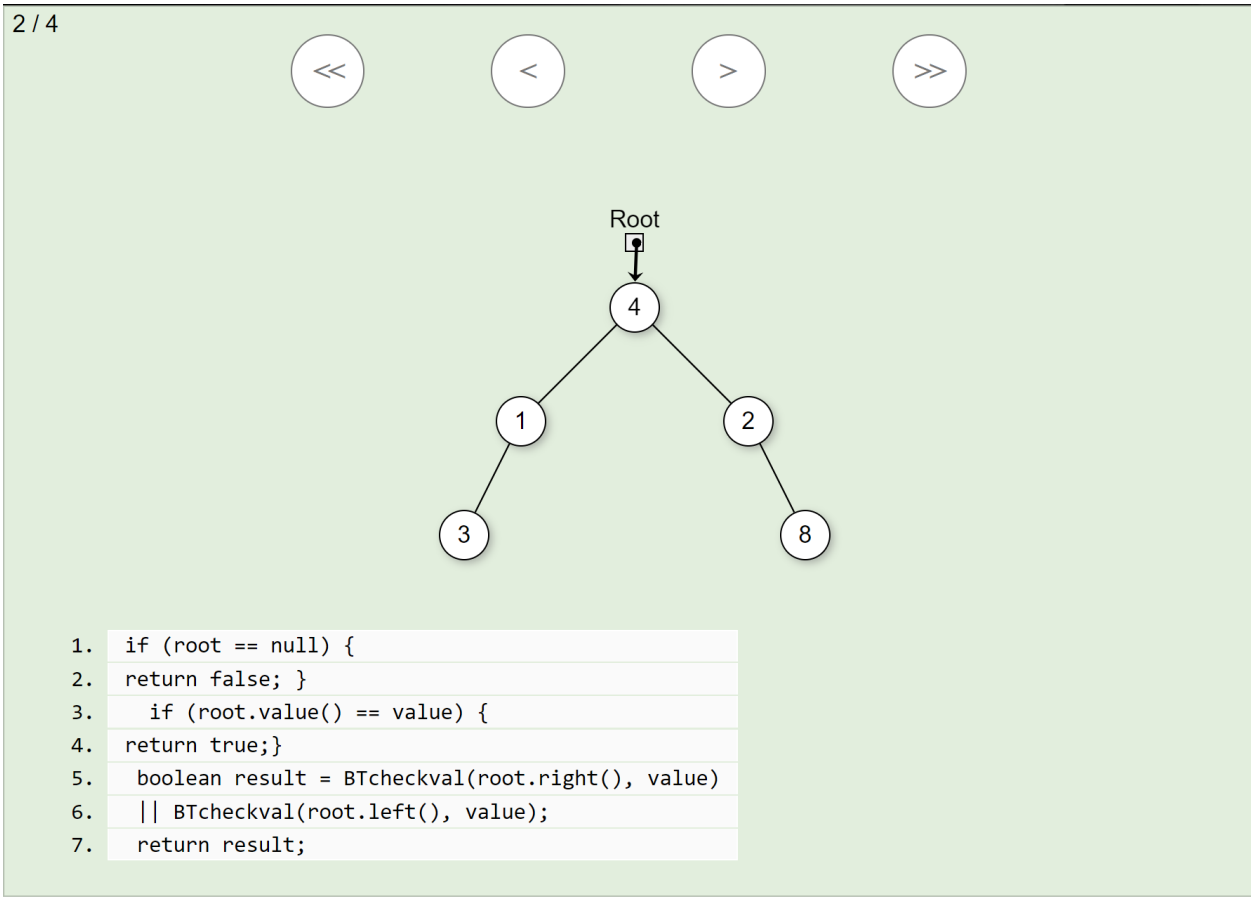


Figure 4: Sample Tree Student Code Visualization

Testing and Evaluation

To test our implementation of tree visualizations, we did some testing on our machines using a sample exercise from CodeWorkout and a sample student solution to that exercise. The sample exercise we used to test our visualization was the Binary Tree Check Value Exercise [5], which takes a sample value in as input and the student is required to write a solution to search through the tree looking for a node with an equal value as the input. The exercise will return true if the value is found, and false if the tree does not contain a node with that input value. In order to grade the effectiveness of the student's solution, CodeWorkout runs several test cases with the student's code and the exercise is complete when the student's code passes every test case provided.

We ran our visualization against the first three test cases of this exercise, which required three separate execution traces of the sample solution running in each of the three cases. The goal of this testing was not to test the effectiveness of the student's solution, as the sample provided to us was a correct solution. Rather, the purpose of running three different test cases was to see if our visualization properly displayed the test tree, and the traversal through the tree based on the sample solution code. With three different test cases we could check several different execution traces with different traversals of the same code, so we could check to see if our visualization provided the correct tree with the correct nodes and pointers, the correct tree root based on a given execution step of the code, and the correct visual of the student code alongside the tree diagram.

User Manual

A. User Experience

This section will go over how this will help the users, why it helps, and then go into detail on how a user can use this.

When someone is learning how to code especially for data structures it helps to be able to see how their code is working. This way the student can know why a program is not working or even to visualize how the code should work in the first place. For example, it is much easier to code an algorithm to traverse a binary tree when you see how the code will execute the traversal. Our product will do this by taking the students code and visualizing it. After that the student will clearly be able to see why the code failed or why it ran correctly. This should help in the debugging process, especially for a website that has no sort of debugger, as well as help with lesson retention.

B. Running Tree Visualization

1. Go to the OpenDSA exercises on CodeWorkout
    a. CodeWorkout
2. Scroll to the binary tree exercise.
    a. X280: Binary Tree Check Value Exercise (Figure 5)
        i. For future implementations the visualizations will work for all exercises.

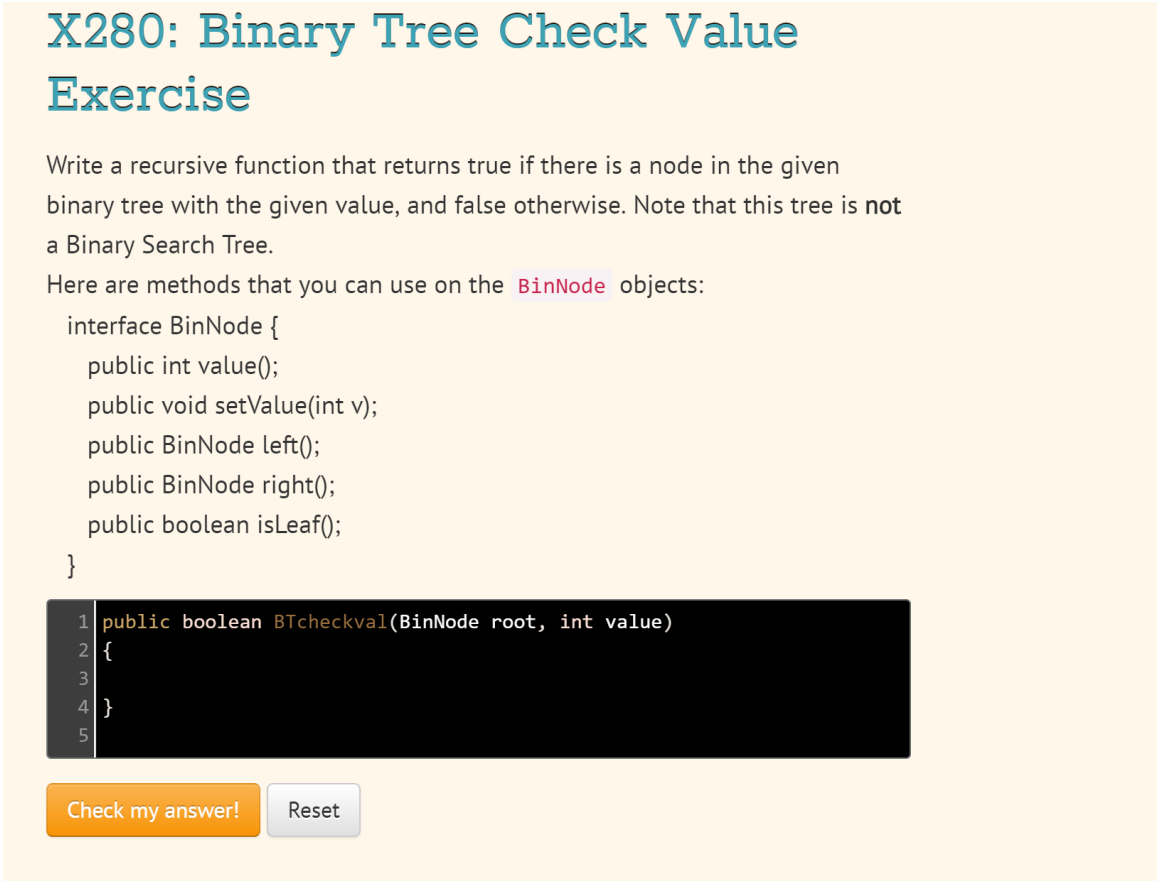


Figure 5: Binary Tree Check Value Exercise in CodeWorkout

3. Write code for exercise between the curly brackets
4. Click Check my answer!
    a. Figure 6 shows the resulting page after the user clicks “Check my answer!”
    b. Then there will be a visualization to show you how your code went and the results from the tests will be shown.

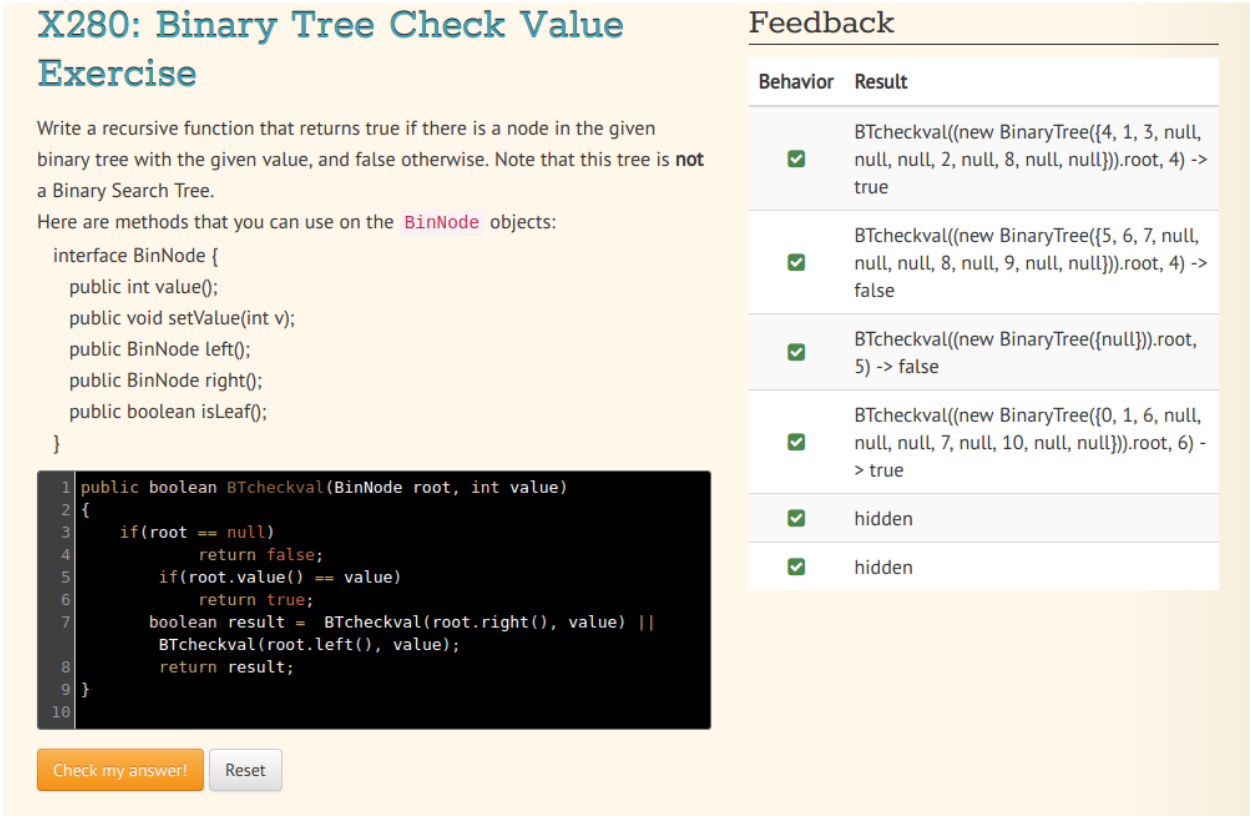


Figure 6: Test Case Results of Sample CodeWorkout Exercise

Developer's Manual

A. Installation and Setup

1. Go to GitHub and clone the repository by following the link below
    a. GitHub Repository
    b. If the link does not work it can be found at the Capstone repository under the user auze217
    c. The homepage for the repository is shown in Figure 7

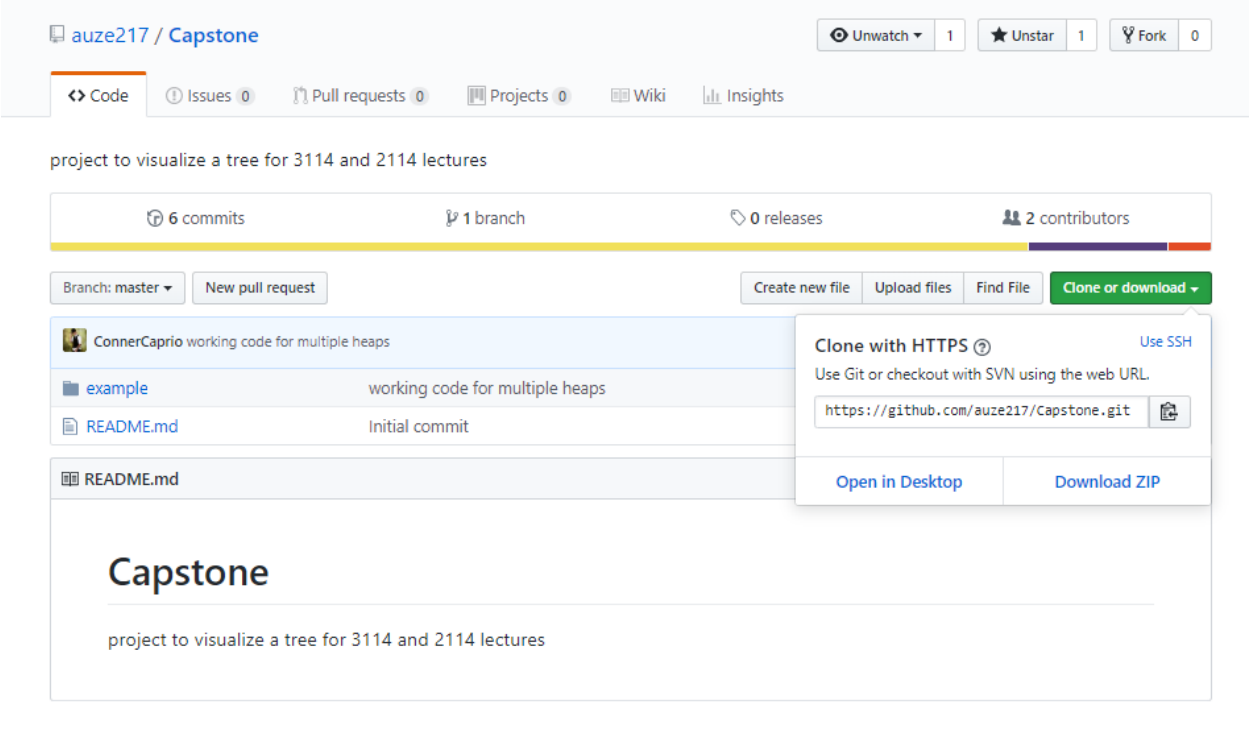


Figure 7: GitHub Repository for Project Code

2. Navigate inside the cloned repository to the example folder, shown in Figure 8

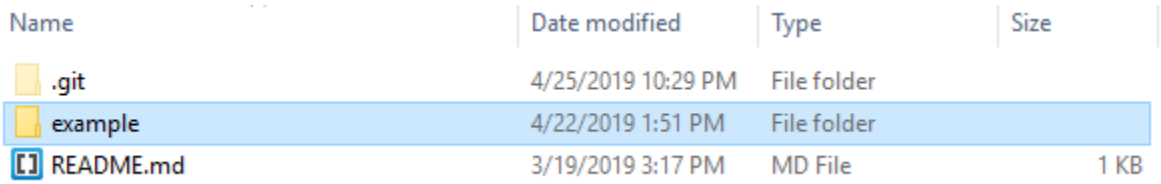
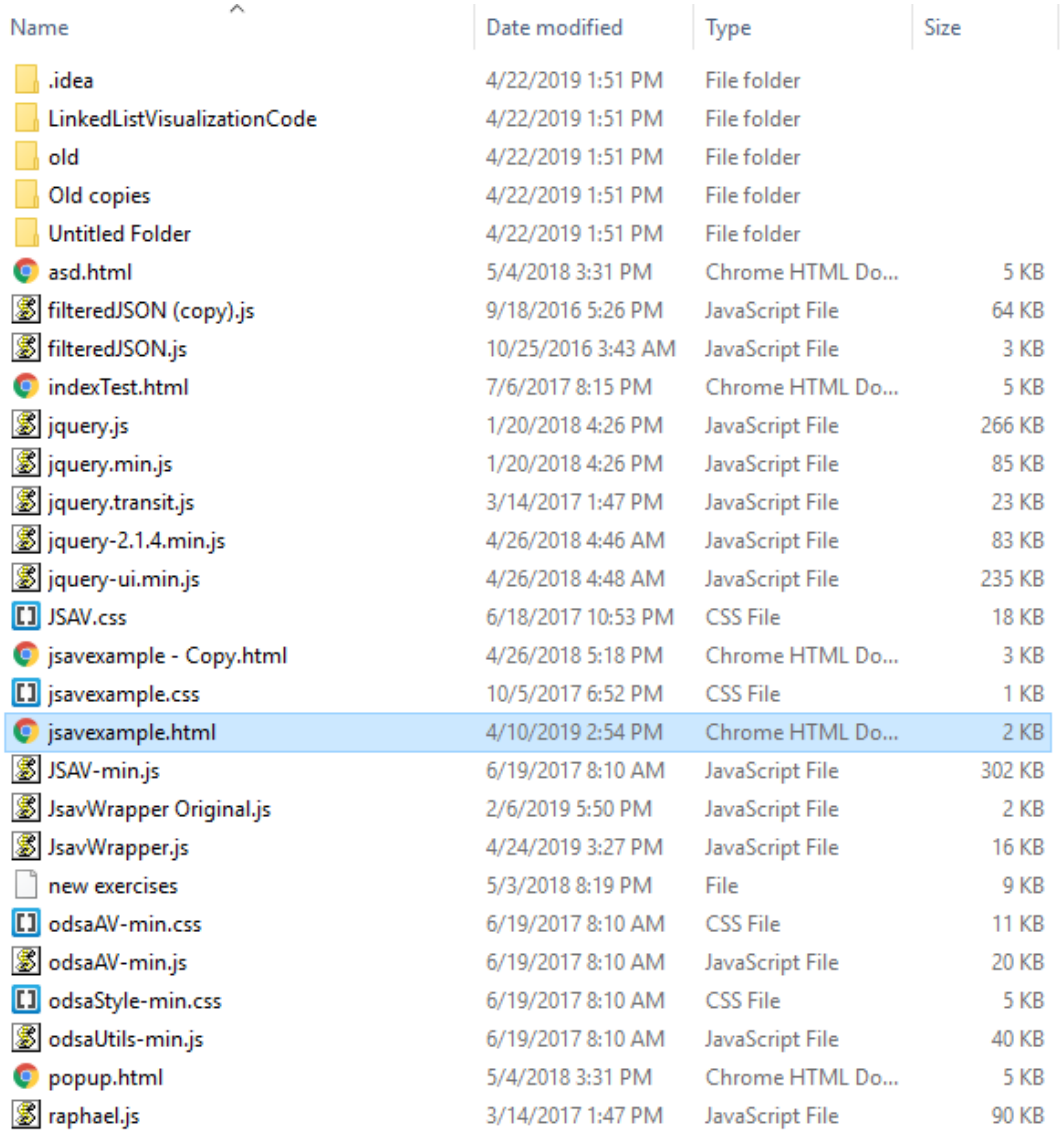


Figure 8: Main Directory for Project Code

3. Click on the jsavexample.html to run the code in your default browser, as shown in Figure 9



Name	Date modified	Type	Size
.idea	4/22/2019 1:51 PM	File folder	
LinkedListVisualizationCode	4/22/2019 1:51 PM	File folder	
old	4/22/2019 1:51 PM	File folder	
Old copies	4/22/2019 1:51 PM	File folder	
Untitled Folder	4/22/2019 1:51 PM	File folder	
asd.html	5/4/2018 3:31 PM	Chrome HTML Do...	5 KB
filteredJSON (copy).js	9/18/2016 5:26 PM	JavaScript File	64 KB
filteredJSON.js	10/25/2016 3:43 AM	JavaScript File	3 KB
indexTest.html	7/6/2017 8:15 PM	Chrome HTML Do...	5 KB
jquery.js	1/20/2018 4:26 PM	JavaScript File	266 KB
jquery.min.js	1/20/2018 4:26 PM	JavaScript File	85 KB
jquery.transit.js	3/14/2017 1:47 PM	JavaScript File	23 KB
jquery-2.1.4.min.js	4/26/2018 4:46 AM	JavaScript File	83 KB
jquery-ui.min.js	4/26/2018 4:48 AM	JavaScript File	235 KB
JSAV.css	6/18/2017 10:53 PM	CSS File	18 KB
jsavexample - Copy.html	4/26/2018 5:18 PM	Chrome HTML Do...	3 KB
jsavexample.css	10/5/2017 6:52 PM	CSS File	1 KB
jsavexample.html	4/10/2019 2:54 PM	Chrome HTML Do...	2 KB
JSAV-min.js	6/19/2017 8:10 AM	JavaScript File	302 KB
JsavWrapper Original.js	2/6/2019 5:50 PM	JavaScript File	2 KB
JsavWrapper.js	4/24/2019 3:27 PM	JavaScript File	16 KB
new exercises	5/3/2018 8:19 PM	File	9 KB
odsaAV-min.css	6/19/2017 8:10 AM	CSS File	11 KB
odsaAV-min.js	6/19/2017 8:10 AM	JavaScript File	20 KB
odsaStyle-min.css	6/19/2017 8:10 AM	CSS File	5 KB
odsaUtils-min.js	6/19/2017 8:10 AM	JavaScript File	40 KB
popup.html	5/4/2018 3:31 PM	Chrome HTML Do...	5 KB
raphael.js	3/14/2017 1:47 PM	JavaScript File	90 KB

Figure 9: Location of jsavexample.html file

4. Press F12 to open up developer tools on the browser (preferably google chrome)
5. Refresh the page and go to the sources tab on the developer tools
6. Open up the jsavwrapper.js file to see the bulk of the code for visualizing binary trees.
    a. Figure 10 shows the dual view of the visualization and our code in a browser window
    b. The code will be explained in a later section of the document

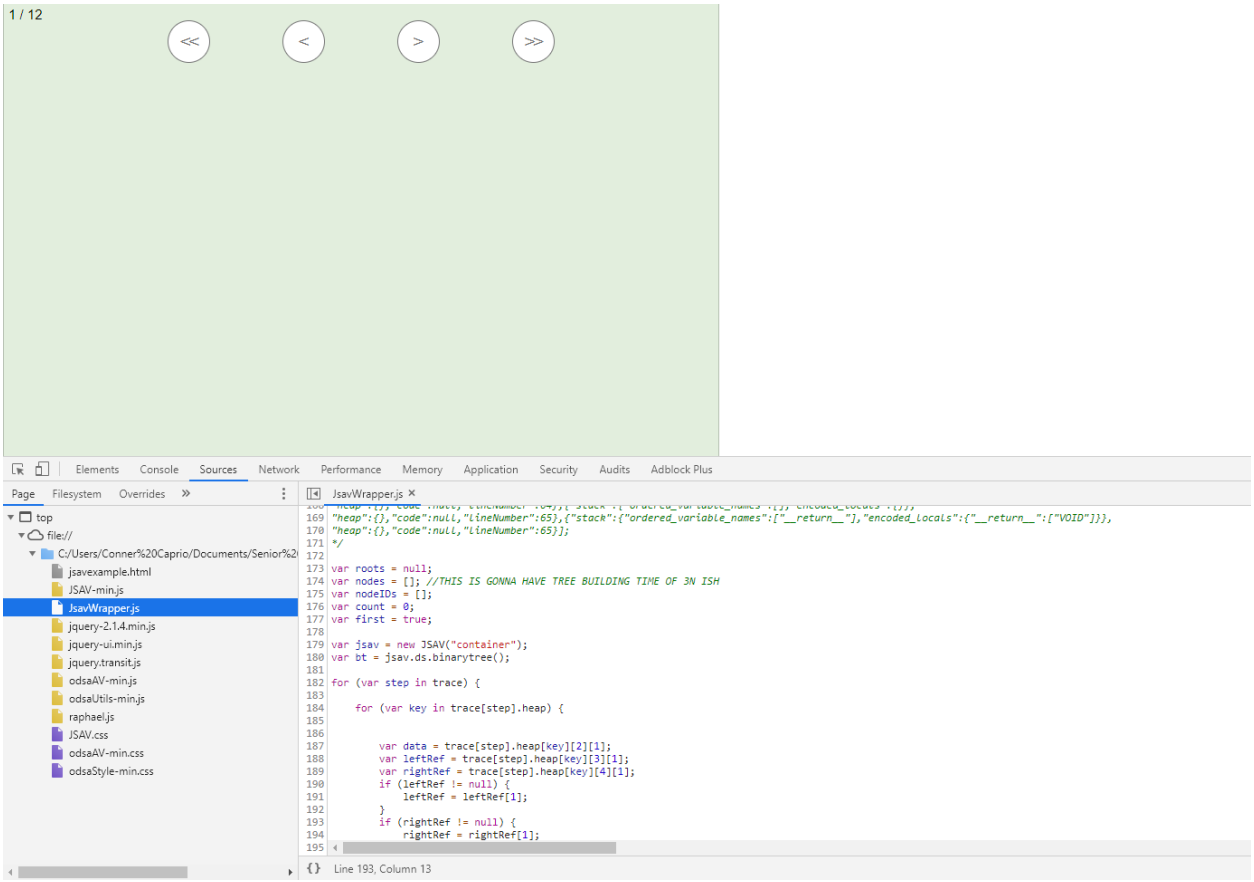


Figure 10: Dual View of Visualization and JavaScript Code in Browser Window

B. Project Structure

Code Execution:

The overview on how this whole project works is as follows. First code will be written by students in code workout based on the exercise . This code will be taken and input into an OpenPop server. This will analyze all of the pointer operations. With this a “trace” will be generated. This trace will be the main topic used and talked about later. This trace is then sent to the javascript files for reading and to be transformed into a JavaScript tree and used by the JSAV library for visualizing.

Code Explanation:

The code for this project can be found in the JsavWrapper.js file. The code calls the testTree method from the visualize function which for testing purposes had a hardcoded trace within it. This was done because of a problem with our machines not being able to run the OpenPop and Ruby servers. The servers require a linux machine which went

against other classwork. For the implemented solution the trace will be passed in as a parameter and that will be the only change.

The code that analyzes the trace and then makes the visualization logically executes as follows: first it starts by making some arrays to hold the made nodes later as well as the jsav object to create the visuals later. There is a main for loop that loops through all the heaps in the trace. Within that there is a nested for loop that goes through all the nodes within a single heap. Depending on whether or not the node has already been made before it updates the data for it or creates a new node and adds it to the array of nodes. After the nodes are created or updated a new iteration is done through all the existing nodes to assign the actual pointers. This is done after because the trace because the trace is only a json object essentially and there is no guaranteed order. So in order to avoid the case where a node is assigning a pointer to a node that hasn't been made yet all pointers are assigned after. Lastly, with the root of the tree set during the first pass of the trace a recursive function is called to set the JSAV objects tree. This is done by simply setting a nodes' left and right but on the JSAV object rather than with the nodes themselves because the JSAV library only takes the values and handles the rest on its own backend. After the trace has been completely iterated through and the JSAV object has made all of its snapshots of the tree the JSAV.recorded method is called to bring the front end back to the first step for the user to see and go through the steps.

Trace Explanation:

The trace is a stack trace of all of the operations with pointers that happen in the students code. A picture of a specific tree trace is shown in Figure 11.

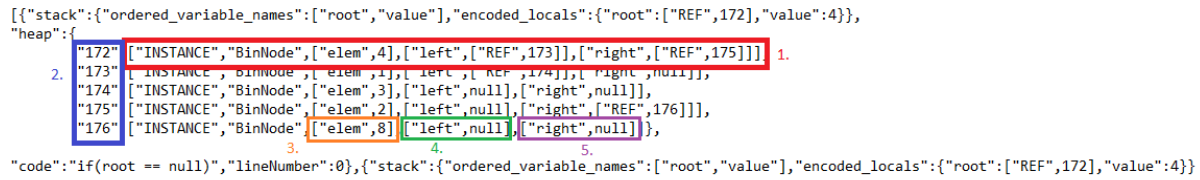


Figure 11: Execution Trace of Sample Student Code

This snippet of a tree trace shows a lot of information. Box number 1 in red shows one of the nodes in the trees. The numbers in box number two, the blue, are the memory addresses of each of the nodes. The string following the INSTANCE is the type of data structure. For linked lists this will say link. For trees it says BinNode for binary node tree (this "BinNode" is what is added to traceHeap.js for accepting new data structures and will be added to for new structures). Next, the information in boxes 3, 4, and 5 (orange,

green, and purple) respectively are the pointers to other nodes in the tree with elem being the data for that node, left being the left child and right being the right child.

In order to generate a trace you must have the servers running for OpenPop and codeworkout. Due to technological constraints we were unable to generate a trace ourselves for this project. Instead, our client Mostafa Mohamed was able to generate a trace and send it to us so we could write our code around it.

C. Adding to this Project

In order to start adding to this project after learning the base code would be to add a new section in the traceHeap file to accommodate the new data structure. Also there will need to be a new trace generated in order to analyze the steps and write the code for looking making the tree and visualizations in JavaScript. For information on traces refer to the section above. After this, all files that start with LinkedList will need to be remade for this new data structure. Any tips or suggestions to adding code will go in the lessons learned section of this document.

Lessons Learned

A. Timeline / Schedule

Date	Milestone
2/1/19 - 2/8/19	First meeting with client, discussion of requirements and timeline for project
2/8/19 - 2/22/19	Install and run OpenPOP server on local machines, ramp up on base code, research all tool required for project implementation
2/21/19	Presentation 1
2/22/19 - 3/8/19	Continue to review base code in preparation for implementation, generate execution trace from OpenPOP for sample tree data structure exercise in CodeWorkout, determine from base code if there is need to update json files
3/18/19 - 3/17/19	Spring Break, catch up on individual coding and learn Ruby on Rails in case needed in implementation
3/18/19 - 4/5/19	Use sample execution trace to write JavaScript code for basic tree visualization functionality
3/20/19	Presentation 2
3/28/19	Interim Report due
4/5/19 - 4/22/19	Add functionality for visualizing multiple heap states within execution trace, which will add the capability to visualize all tree operations and traversals (including recursion) in the visualizations
4/10/19	Presentation 3
4/22/19 - 4/29/19	Find students (focusing on 2114, 3114 students) to test our system and give feedback, write up final report for project
4/26/19	Final Report due

4/29/19 - 5/8/19	Revise final report and create final presentation, wrap up project implementation and write documentation for future development and improvement
4/30/19	Present at VTURCS Spring Symposium
5/2/19	Final Presentation
5/6/19	Present at ICAT Day

B. Problems and Solutions

A major problem we encountered throughout the semester was that we got off track of our initial timeline when we were delayed by the issue of not being able to run the OpenPOP server on our machines. When we discovered that we needed to have a Linux OS in order to run the server and therefore generate the execution trace used to create the visualization, we met up with our client to discuss possible solutions since none of the group members owned a laptop running Linux. We determined that the easiest solution to this problem was that our client would generate the execution trace for us since he already had the server up and running on his computer and it wouldn't take very long for him to generate a new trace for us when we required it.

Because of this delay in figuring out how to resolve the issue of getting the execution trace, we found ourselves about a week behind in our project schedule. We were able to make up the time to finish the implementation of the visualizations by pushing back our implementation timeframe into April rather than starting in late March as we had initially planned. This way, we finished the implementation for the visualizations and we were able to do local testing by testing three different test cases of a tree exercise in CodeWorkout, and confirming that they were visualized correctly. This pushback did mean that we did not have enough time to conduct our planned student testing at the end of April and start of May since we also had to finish the final presentation, final report, and our research poster for the VTURCS Spring Symposium. This testing would need to be conducted in the future as this project is developed further to ensure that our final product is a good interface and learning tool for the students in these classes using the visualization tool.

C. Future Work

The TreeViz project as it stands right now has a lot of potential to be expanded upon in the future. First, the code we have written is currently not integrated into the OpenPOP server to run in conjunction with tree data structure exercises in CodeWorkout, so the

first future goal of this project should be to properly integrate our work into CodeWorkout. This will require more testing of our base code to ensure that it works for all types of tree exercises, specifically exercises involving node addition and deletion since the sample exercise we used to test on our local machines was a tree search method. Once the visualizations have been tested on multiple different sample CodeWorkout exercises, our solution files should be added into the OpenPOP server alongside the linked list visualization JavaScript files. Likewise, the CodeWorkout homework exercises involving trees should be updated to include our visualization code so that the visualizations will appear alongside the exercises.

Another future goal is to expand the OpenPOP server to allow functionality for more data structures that are taught in CS2114 and CS3114 courses. These visualizations would be very helpful for all data structures exercises since the concept of data structures are so visual in nature, and it would help student learning significantly if they had the ability to debug their code through another medium by way of these visualizations. Since the design already in place for the linked list and tree visualizations is very similar, it could be replicated without lots of challenges for other kinds of data structures. This would also be accomplished very easily due to the fact that the JSAV library already has the resources to visualize all kinds of data structures taught in these courses, as they are already present and used by in the OpenDSA textbook modules that students read for the courses. With the addition of more visualizations for all kind of data structures, it would make CodeWorkout a more helpful tool for students to use in conjunction with these data structures courses.

Additionally, a future goal for this project is to optimize the entire code base to make all of the code cleaner and more efficient. Right now, our code for tree visualizations is significantly slower than it could be because with each new heap in the execution trace our code rebuilds the entire tree from the pointers within the trace. Rather than taking the time to rebuild the entire tree, which has not changed much between execution states, it would be much more efficient to simply change the one part of the tree that was impacted from that execution step and only draw that change. This would eliminate the time needed to redetermine the tree pointers each time the code loops through the heaps within the execution trace, and also the need to rebuild the tree each time around this loop. This would decrease execution time of our solution by at least 2N time. For the time being, this change won't impact the execution of our visualizations since the sample trees used in the CodeWorkout exercises are quite small and will not feel the effect of a 2N execution time difference as much as a larger tree. However, in keeping with good coding practice, our solution should be optimized in the future to increase the effectiveness and longevity of our design.

Acknowledgements

Mostafa Mohammed, profmdn@vt.edu

Clifford A. Shaffer, shaffer@cs.vt.edu

References

[1] “CodeWorkout.” CodeWorkout, https://codeworkout.cs.vt.edu/. Accessed 26 April 2019

[2] “JSAV: The JavaScript Algorithm Visualization Library.” JSAV, http://jsav.io/. Accessed 26 April 2019.

[3] “Welcome to Rails.” Ruby on Rails 5.2.3, Ruby on Rails, https://api.rubyonrails.org/. Accessed 26 April 2019

[4] “OpenDSA.” OpenDSA, https://opendsa-server.cs.vt.edu/. Accessed 26 April 2019

[5] “12.7. Information Flow in Recursive Functions.” OpenDSA Data Structures and Algorithms Modules Collection, OpenDSA, https://opendsa.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeInfFlw.html. Accessed 26 April 2019

Appendices

A. VTURCS Symposium Poster