ADTeach: An Educational ADT Visualization Tool

Sarah LeBlanc, Nick Petru, Colby Seyferth,
Emily Shack, Liz Shank, Brady Soglin

March 15, 2015

Abstract

ADTeach is an ADT visualization tool for students; users enter their code in Viva, a Java-like language, into our programming environment; then, they play or step line-by-line through the code. Through our step-by-step animations, they are able to see how the different ADTs' states change as a result of their code. In its current state, ADTeach can be used as a supplement to course lectures, giving the student a chance to learn about ADTs on their own.

1 Introduction

1.1 Overview

ADTeach is a web application that addresses the educational need to illustrate the functionalities of abstract data types. In order to grasp the applications and benefits of various algorithms, a Computer Science student must begin to understand and utilize the ADTs that go into creating such algorithms. To this end, it can ease the learning process if the student has a way of visualizing ADT structure and behavior. The understanding this promotes is crucial for success in introductory Computer Science courses, such as Data Structures and Algorithms. Certain representations of ADTs have become so foundational in Computer Science that they can be applied not just to these courses at individual institutions, but to a broader shared understanding of these ADTs throughout the field. The primary goals of our project are as follows:

- Allow students to code in a familiar, pseudo-Java language.
- Include a clear visual representation of data structures as they are being searched, sorted, or otherwise manipulated as code runs.
- Provide accessible preloaded "example code" to allow students to immediately view and run algorithms on common data structures/algorithms.
- Maintain a user friendly experience with an easy learning curve.
- Illustrate ADTs in an intuitive way that helps with students' understandings of how they work.

1.2 Timeline

This project was completed over the course of 26 weeks. The first five weeks were spent researching previous work, and using our findings to make big-picture decisions about the structure of our project. The second five weeks were spent continuing research and formally designing a system architecture, as well as beginning work on the interpreter side of the project. We split up over the next six weeks, fleshing out the architecture, continuing more focused work on the interpreter, and beginning work on the visualizations. During the final ten weeks, we completed the visual side and interpreter, and modified the architecture as needed to piece them together. We also formally presented our work during this time, conducted user tests, and compiled this paper documenting our research.

1.3 Use Cases

Although the visualization of ADTs is in theory useful at any level of computer science learning, we imagine that it will be particularly useful to students who are in classes that focus heavily on the construction and manipulation of ADTs. We therefore present two fundamental use cases: Students enrolled in the classes Data Structures and Algorithms at our institution, Carleton College.

1.3.1 Use Case 1

User: Michael is a student in Data Structures fall term.
Goal: Michael wants to understand the different stack methods and how multiple stacks can interact with each other.
Steps:

1. Michael opens ADTeach and clicks on the Help button. Because he is interested in learning about Stacks, he clicks on the Stacks button from the list of ADT help options. On the right a menu of possible Stack options appear, along with information about how to instantiate a Stack (Fig. 1).

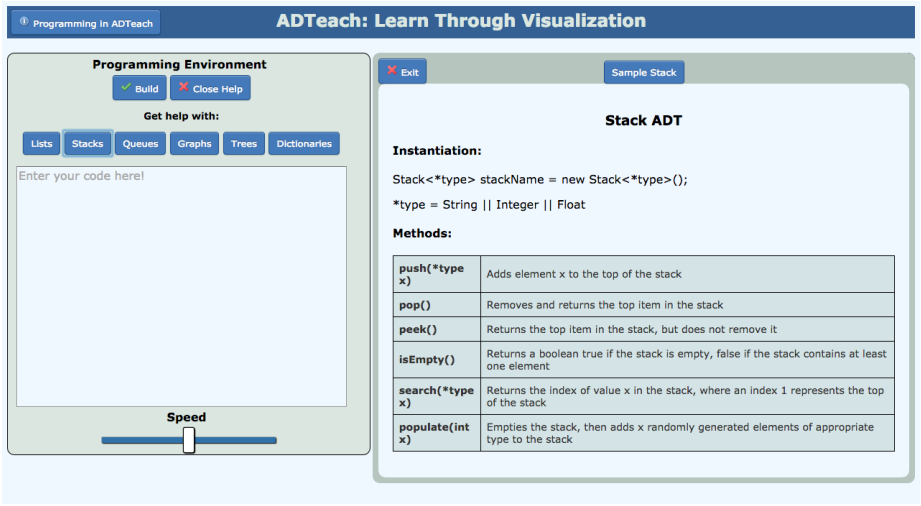

Figure 1: ADTeach Stack Help Menu.

2. Michael doesn't know exactly what he wants to code on his own yet, so he selects some sample code using the Sample Stack button on the top of the help screen.

3. Michael watches as his code runs and can see the data being moved from one stack to another (Fig. 2).

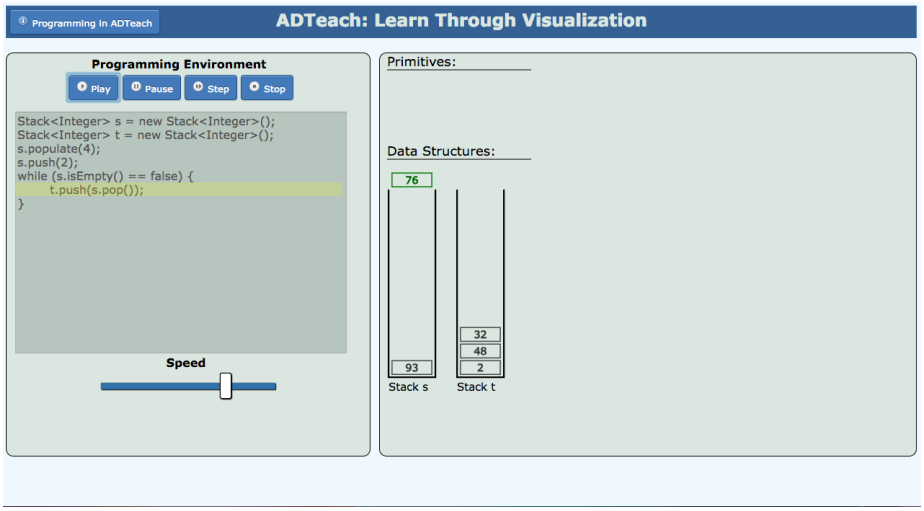

Figure 2: ADTeach Stack Visualization.

1.3.2 Use Case 2

User: Anders is a Junior Computer Science major in an 8:30 AM Algorithms class. He is not the definition of a morning person, so he spends time after class reviewing and recreating algorithms from class on his own.
Goal: Anders wants to compare and contrast Breadth First Search and Depth First Search.
Steps:

1. Anders opens ADTeach. He translates the BFS pseudocode from his class notes into Viva in the programming environment (Fig. 3).

2. He steps through the code, noticing how the Queue and Graph work together in BFS, finding nodes and adding them to the list (Fig. 4).

3. He repeats the process for DFS (Fig. 5).

4. After stepping through each of the algorithms a few times, Anders is able to have a better understanding of how search algorithms work, and how BFS and DFS are similar, yet different.

2 Background

ADTeach is a useful tool that brings visualizations to learning about ADTs. Without ADTeach, many students learn to use print statements in the console/terminal window to see the states of their ADTs. They can also use a pen

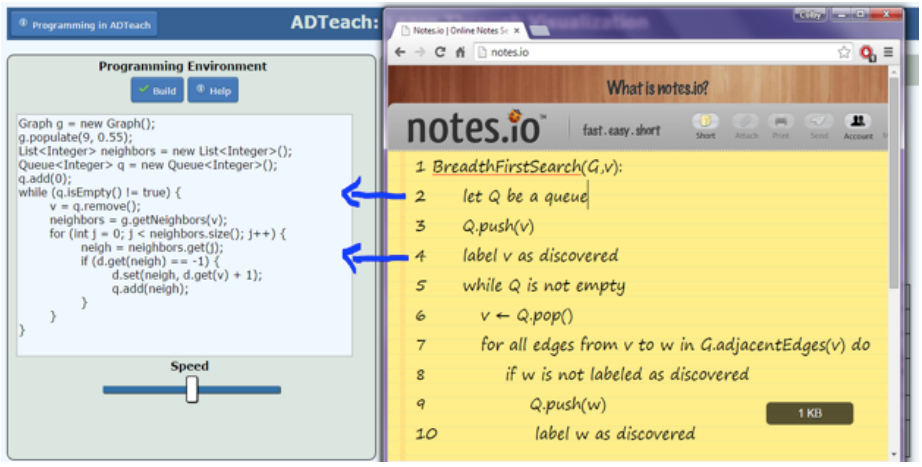

Figure 3: Converting Pseudocode into Viva.

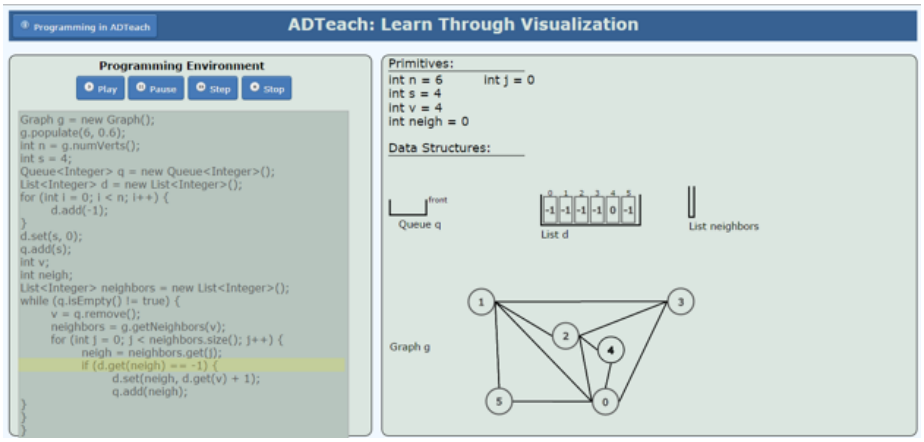

Figure 4: BFS with ADTeach.

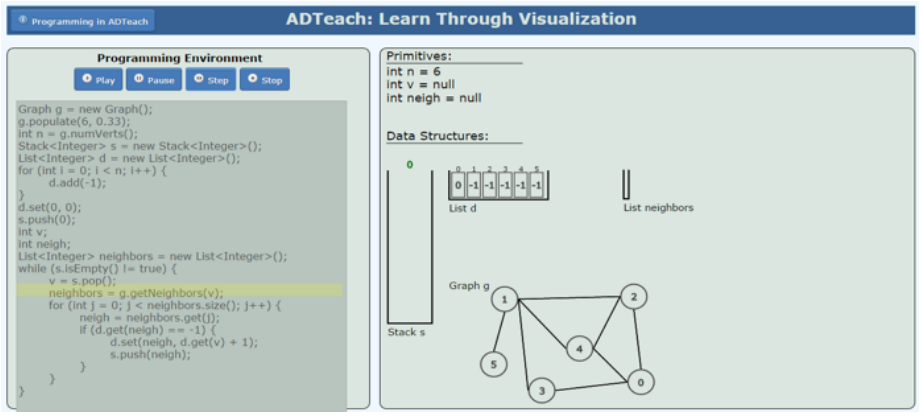

Figure 5: DFS with ADTeach.

and paper to sketch out their ADTs, but that is a long and tedious process. One tool we found which allows users to input their own code and see it rendered visually on the same screen is the website w3schools [4]. This is a site that teaches the user how to program in HTML, CSS and Javascript to create their own websites. Many demos on this site display user-modified code as it would appear on a website. However, it does not interpret Java or Python, or any other language where stepping-through of code is supported. A more similar instrument to ADTeach, Python Tutor, markets itself as a visualization tool for students that are learning a new programming language (specifically Python, Java, or Javascript). Python Tutor does not have any animations, and their ADTs look similar to each other [5]. Thus, one of our goals with ADTeach was to illustrate the ADTs in a more varied way, in line with the ADT visualization practices used by Carleton professors. We were also hoping the added animations would make it easy to follow the ADTs' interactions with data.

3 Methods and Implementation

The target audience of ADTeach is the students of Data Structures and Algorithms and so we kept them in mind when deciding which language to support in the Programming Environment. Because Data Structures students are learning Java at the same time as learning the ADTs, we decided to support a simplified form of Java, which we named Viva. The main differences between Java and Viva are that Viva does not support user creation of Classes or functions, and Viva does not support pointers. The syntax of Java and Viva are virtually identical.

We used a Model-View-Controller system architecture for ADTeach (Fig. 6). In the View we placed the code box, the buttons and the visualizer. The Model contains the symbol table and sample code, and the Controller contains the interpreter and event handler. When a user is finished writing their code in the code box, it gets sent through the event handler to the interpreter. As the interpreter is working on the code, it is updating the symbol table with updated values or new variables. The visualizer handler then looks at the sequence of events in the symbol table and animates them with the given information.

The tokenizer is based on sample code in Eli Bandersky's blog post comparing the run time of regex tokenizers vs handwritten tokenizers in javascript [2]. The parser is a modified version of Douglas Crockford's parser written for JavaScript in JavaScript [3]. This parser utilizes Vaugh Pratt's flexible operator precedence technique, allowing us to fine-tune it to our language [6]. We changed the parser slightly so that instead of parsing for JavaScript it parses Viva, and then built the interpreter from scratch to work with Crockford's parse tree structure. When Build is pressed, the code is sent through the Event Handler from the Code Box to the Interpreter. The interpreter takes the string of code and sends it through the tokenizer to break the string into a list of tokens. This token list is sent through the parser, which organizes the tokens into a parse tree. The parse tree is linearly structured in its highest level. The highest level of the parse tree is a list of blocks with possible types of Semi-colon, For, If, and While. The different types of blocks hold different information. They all have

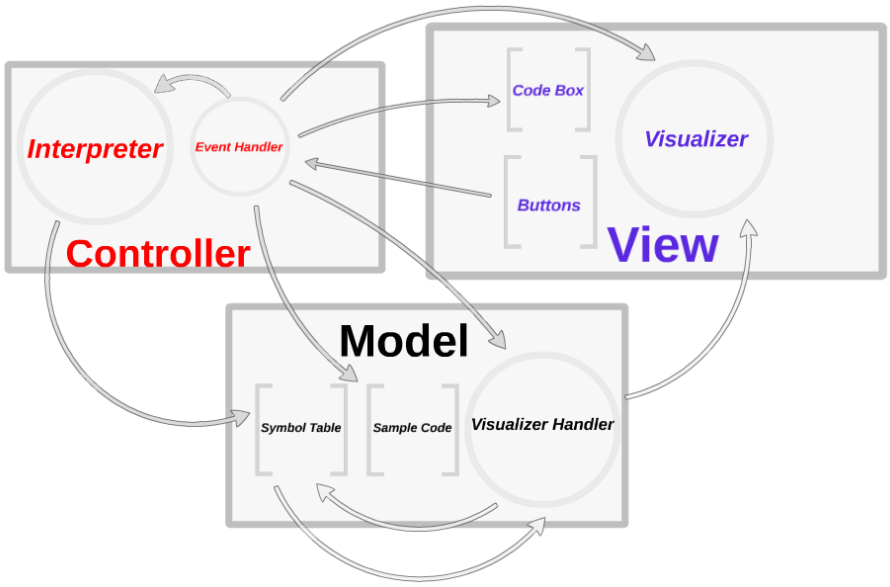

Figure 6: Software Architecture.

a type, which helps tell the interpreter how to evaluate it. If and While blocks both have condition trees, For blocks have initialization and step trees, and all blocks have Body trees. The next level of the parse tree, when a block is being evaluated, is formatted as trees. A tree is a smaller quanitity of code, usually just a single line or condition. The different types of trees are Assignment, Condition, Method, Step, and Math (Fig. 7).

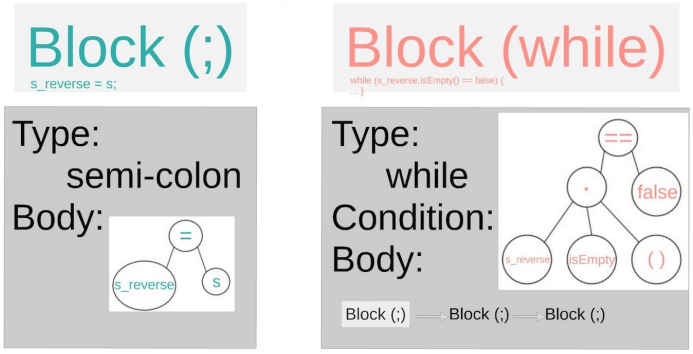

Figure 7: Sample parse blocks.

The parse tree gets passed to the interpreter, which takes each block in order and evaluates it. While the interpreter is working through the parse tree, the values of the variables are stored in the Environment. The Environment lives inside the interpreter and stores the current variables, their types, and their current value. In Viva, information in the environment only changes during an

assignment statement, when incrementing a variable or when a method is called. Because of this, the only time the environment is updated is when the interpreter is evaluating an step tree, assignment tree or a method tree. The interpreter itself is structured as a type of tree, where different blocks will have different paths through the evaluation tree. All blocks start at the master evaluation method, which sends the block to the next evaluation level depending on the type of block. Each tree type has its own evaluation method (Fig. 8, Fig. 9).

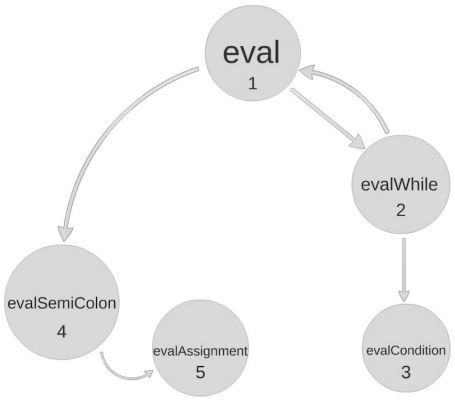

Figure 8: Evaluation path for while block.

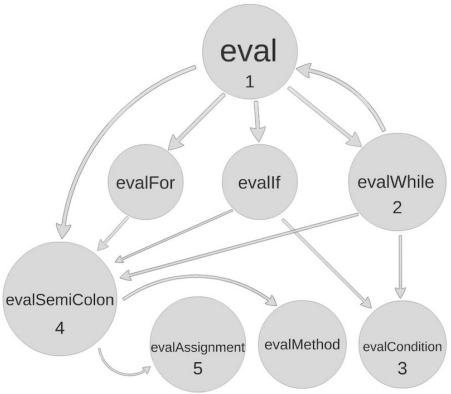

Figure 9: Additional evaluation paths in interpreter.

The interpreter's sole job is to take the code as a string and gather meaning from it in the form of variables and their values. Each time the environment gets updated - when a new variable is created, a variable's value is changed, or a variable is removed - the environment alerts the Symbol Table of these changes, which is the connection between the Controller and the Model.

The animations of ADTeach rely on Raphaël, a small JavaScript graphics library based on the SVG W3C Recommendation and VML. Every graphical object created with Raphaël is also a DOM object, making it simple to track,

modify, and add JavaScript event handlers to any such object [1]. We chose Raphaël over more powerful data visualization libraries such as D3 and Infovis in part because of Raphaël's reliable browser compatability, rich documentation, and ease of use. More importantly, although the layout algorithms built in to D3 and Infovis make them both tempting options, we wanted to animate primitive values moving between different kinds of ADTs, and to demonstrate visually the differences between the structures of various ADTs. Raphaël made our job easier without getting in the way by simplifying the creation and tracking of graphical objects without imposing any design constraints.

The symbol table stores an environment (a series of variables and corresponding values), and, as this environment changes over time, can communicate the state of the environment one frame at a time to the visualizer handler. The visualizer handler is responsible for animating this step– representing the environment at one state and then showing the transition from this state to the next (Fig. 10).

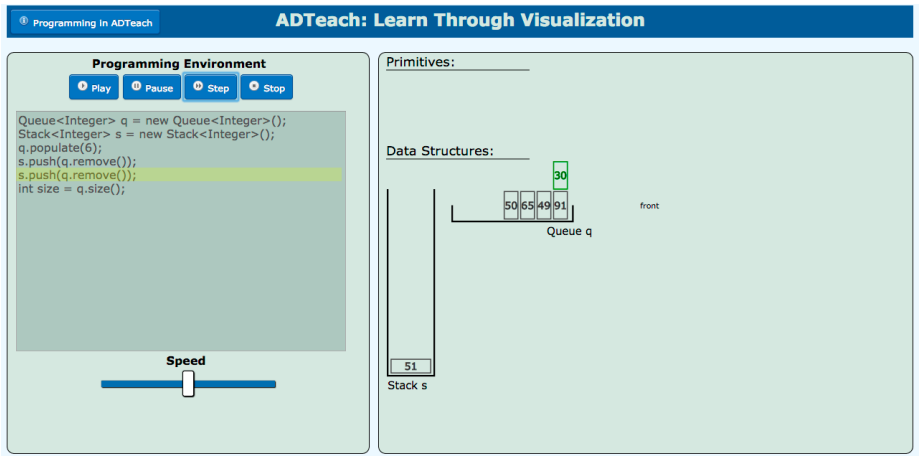

Figure 10: Adding data unit from queue to stack.

Data detailing the new state of the environment flows from the symbol table to the visualizer handler as an events, and each event takes the form of one operation that can be performed on a single variable. The visualizer handler may receive a variable's corresponding event (be it a "create," "update," or "destroy") and then will prepare to animate the neccesary change in state. Variables are stored in the visualizer handler as "Entitites." An Entity can be an ADT or a primitive value (String, int, float, etc.) Each entity is responsible for animating the variable it represents, and Entity objects contain preloaded Raphael animations that it can call upon as it is created, updated, or destroyed. The process of animating a single change in the state of the visualizer handler's environment is called a step.

These steps will always be handled in sequence, though the frequency in which the visuals are created depends on the user interaction. A user can either watch

each step individually or play through them in sequence, sit back, and watch as the visualizer handler animates the interpreted code.

When the visualizer handler encounters a "new" event, it creates a new entity and pushes it onto a list of entities as an instantiation of the appropriate type. An entity represents a variable visually, and could be a Primitive, List, Queue, Priority Queue, Stack, Dictionary, Graph, Weighted Graph, or Tree. Entitities have all of the animations a variable of their type could need preloaded in Raphael, and they can prompted to be animated by an "update" or "delete" event if needed. The entities can then be arranged and animated.

Within an Entity, animations are handled by Raphael, and queued at the start of a given step using an incremental delay system. A simple event may require only a single animation (fading in a variable when it's created), but a more complicated one may require many (inserting a node into the middle of a list.) The delay system allows Entities to coordinate the animation of more complicated, multi-step animations. By adjusting the delay, Entities can communicate to each other when they will be animating, and so avoid playing multiple animations simultaneously and confusing the user.

In addition to carefully implementing delays to facilitate comprehension of code, we including a system of entity color change and movement to communicate updates and removal. These features fit in with our educationally oriented design goals, as they aid the user's ability to connect animations with lines of code. For instance, if an primitive or data unit moves or if a graph edge is established, the item will turn green temporarily. Similarly, entities will turn red before fading out. Finally, we implemented a slight shaking of primitives when their value changes. That way, the user can expect a certain type of change to be made in the visuals. This brings the user's attention to the appropriate entity before it actually changes (Fig. 11).

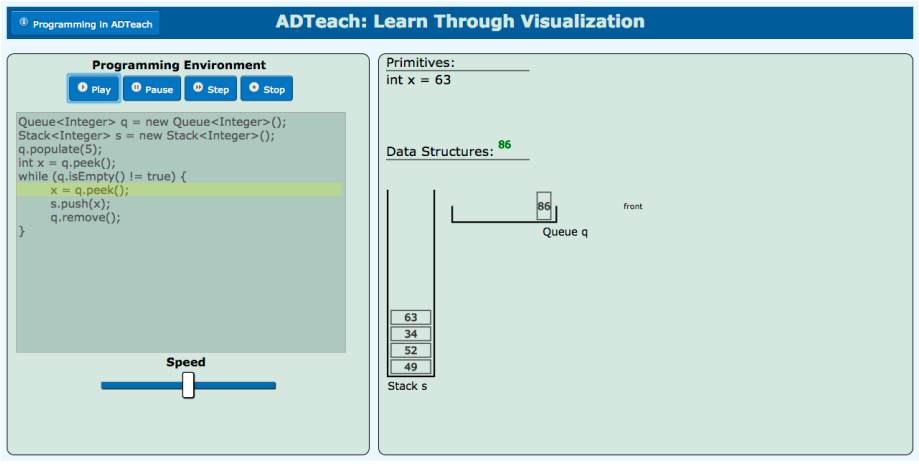

Figure 11: Demonstrating primitive change based on ADT data unit.

Another important portion of animating a step is the passage of data between entities. When an "update" event includes a source Entity, we know that the new value used in the update originated with another instantiated Entity. If this occurs, the Entity being updated will request an 'anonymous variable' from the source Entity, which will then show the data moving from the source to the updated Entity (Fig. 12).

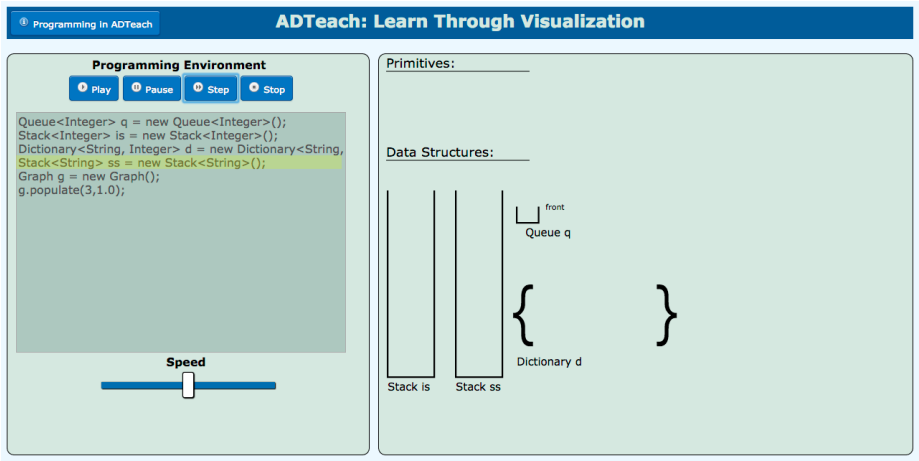

Figure 12: Creating new ADTs and arranging/shifting them.

The visualizer handler has two systems for arranging entities: one for primitives and one for ADTs. For primitives, it stores all such variables in a matrix whose cells correspond to columns of the primitive visual section. This system seemingly organizes the primitives in a queue; for instance, if a primitive is removed, the subsequent primitives will shift, ensuring that all the first cells are all filled. ADTs follow a much more naive approach. They are broken down into three categories: vertical, horizontal, and "blob" shaped ADTs. The vertical ADTs, namely Stacks, will appear on the left and populate to the right as more are added. Horizontal ADTs (Queues, Priority Queues, and Lists) will always appear to the right of Stacks and populate downards. "Blob" ADTs (Graphs, Weighted Graphs, and Trees) will also appear to the right of Stacks but also below the horizontal ADTs. If certain ADTs are visualized in such a way that a previously drawn ADTs conflict with the expected layout, the preexisting ADTs will shift to make room for the new data structures. By categorizing the ADTs, the orientation can help enable a predictable, automated layout so that the user can gain a higher level of familiarity with the program.

3.1 Evaluation

A qualitative way to evaluate ADTeach is through user testing. We asked for volunteers from the Winter 2015 Data Structures (CS 201) and Algorithms (CS 252) classes at Carleton College. We user-tested ADTeach on five Algorithms students and one Data Structures student. The volunteers used ADTeach for approximately 30 minutes on average. They took the first half of the time to

explore ADTeach on their own and gain an understanding of it without our interference. They were also given a sheet of paper with some sample problems should they run out of ideas of things to try. They were then asked to fill out a form with general feedback about usability, readability, and suggestions, as well as being asked if they would use ADTeach in their class.

In addition, we evaluated the time it takes for the interpreter to compile code on various inputs. This gave us a quantitative measure for the performance of ADTeach.

4 Results

4.1 Quantitative Results

Algorithm	Time
Reverse a stack of 20 objects using a Queue	9 ms
Empty a stack of 15 items onto another stack 1,000 times	67 ms
Performing bubble sort on a randomly populated list of 50 items	95 ms
Performing bubble sort on a randomly populated list of 300 items	3.372 s

Table 1: Results of the timed interpreter.

4.2 Qualitative Results

The user testing feedback can be divided up based on topic:

Ease of Use

- "Help pages are clear and informative."
- "It's really easy to use, I like the help button and how there's sample code so I can watch how it's supposed to work."
- "Looks great! Super happy about the visualizations and transitions. The help was easy enough to find and use."
- "Good. Easy access to instructions and instructions were clear. Coding was easy despite slight differences in language from Java. I found myself navigating the various options very easily and quickly within five minutes of playing around with them."
- "Wasn't immediately clear to me that ADTs were already implemented"
- "Should play by default once a build works"
- "Maybe have a clear screen button"

Would it be useful for your class?

- "Definitely! Especially with stacks and queues, where it can be hard to visualize on a chalkboard."
- "For a lot of people, absolutely. This would certainly help visualizing examples of more difficult algorithms in order to understand them."
- "Yes. I am much more of a visual learner, and I would enjoy playing around with this program much more than having to draw the ADTs on paper and erase/redraw all the steps myself. I think this is a fun and interactive way to learn about ADTs."

Could you picture yourself using ADTeach as a resource?

- "I would use it if I'm having trouble with an assignment and want to make sure that my fundamental understanding of how it should be working is right."
- "Yes, I'd use it as a reference for making sure I actually got a given algorithm. I think maybe for someone learning data structures it could be useful to visualize as well."
- "I would. There are lots of problems in classes like Math of CS and Algorithms that use graphs and trees and I have trouble visualizing them and end up wasting time drawing and redrawing them."

5 Analysis

We performed the user testing before we had completed the final version of ADTeach and therefore had time to take into consideration some of the productive feedback not listed above, such as more descriptive error messages and some issues with the visualization layout when there were multiple ADTs. The overall results were very positive, and people felt that they would use this in a class setting. Our goal was to help Data Structures students cement their fundamental understanding of ADTs and to help Algorithms students visualize complicated algorithms that prove difficult to visualize on a chalkboard. Based on the feedback form the user testing, these goals have been met with ADTeach.

As far as the timing of the interpreter goes, for most algorithms being performed on a reasonable size ADT, the interpreter will take less than one second to interpret the code and allow it to be played. Therefore, with more complicated code, the time the interpreter takes is almost unnoticeable and therefore not something that our users will be concerned with.

6 Conclusions

ADTeach is a functional educational tool applicable to Data Structures and Algorithms students. It supports all the ADTs introduced to students in Data Structures and the ADTs that are the backbone of many of the complicated

algorithms taught in Algorithms. ADTeach can be used by individual students who want to explore the different methods of an ADT or by professors who don't want to fill an entire chalkboard and redraw if students are confused. In the future, with more time to work on ADTeach, students could be able to paste in their actual code to ADTeach without filtering it to fit Viva's rules, they could make even larger ADTs and professors could monitor what sample code their students have access to. Visualizing abstract data types can be hard, and that's why ADTeach is a necessary and very helpful tool.

6.1 Future Work

A lot of ground has been made on ADTeach, but there are still some things that we would accomplish if we had more time.

- As ADTs get too large, they come off the screen. This could be solved using an animation with ellipses to compress the ADTs.
- The creation of graph edges is slightly delayed
- User created functions are not supported

References

[1] Dmitry Baranovskiy. Raphaël–javascript library. Web, 2013.

[2] Eli Bendersky. Hand-written lexer in javascript compared to the regex-based ones. Web, July 2013.

[3] Douglas Crockford. Top down operator precedence. Web, February 2007.

[4] Refsnes Data. w3schools online web tutorials. Web., 2015.

[5] Philip Guo. Online python tutor: Embeddable web-based program visualization for cs education. *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2013.

[6] Vaughan R. Pratt. Top down operator precedence. *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1973.