Animation Environment for Linked Data Structures

Kiu Man Yeung

Final-Year Project- BSc Computer Science
Supervisor: Dr Kieran Herley

Department of Computer Science
University College Cork

April 2022

Abstract

The basis of this project is to create a data structures visualisation tool for educational purpose, which students can inspect their code in execution to achieve better learning result. The main goal of the project is to create a visualiser that shows step-by-step linked data structure manipulations in Python. The visualiser provides a graphical interface that allows users to input their implementation and display the result as stepped animation of the execution beside.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award. I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: Kiu Man Yeung
Date: 25th April, 2022

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor Dr Kieran Herley for his time, patience, advice, and guidance throughout the project.

Contents

1 Introduction

1.1 Motivation

Data structures and algorithms is a must-learn topic in a Computer Science or Software Engineering education. The main focus of learning data structures and algorithms is correctness and efficiently. Being able to organise data in an appropriate structure enhances the algorithms' efficiency and showcases one's ability in problem solving.

Many students struggle with understanding, choosing, or implementing data structures. Students learning data structures and algorithms after introductory level courses. Data structures and memory management concepts is quite a foreign idea to beginning programmers. Students may not understand how pointers in a data linking mechanism works. They may already have experience with canned data structures without know what is inside the black box.

Instead of just reading the textbooks, drawing of data structures is much more effective to visual learners. Students can see how data structures are constructed and linked. Graphing every step in the execution allows them to understand how the operation has affected the data structure.

However, the drawings may not be the correct representations of the data structures in execution. It is rather an expectation projection. The graphs and the implementation are not technically connected and therefore may not reflect problems in broken data structures and algorithms. This project aims to help students better understand the topic by visualising the structures they have created.

1.2 Goal

This project aims to build a visualiser that bridges the gap between the source algorithm and the graphical visualization. The tool is specialised to visualise linked data structures nicely and present the effect of the algorithm in an intuitive animated environment.

2 Analysis

2.1 Objective

The visualisation tool targets students who are studying data structures and algorithms. It is developed as a teaching aid hoping to assist students master the topic and enhance their ability in problem solving.

The visualiser allows entering and editing Python source code. It generates a graph for every step once the source code is executed. The graph presents the call stack [1] which shows all objects created by source code currently in the memory. While stepping through the source code, it highlights the current line of execution and display the corresponding instant of call stack.

2.2 Specification

Source Language	Python 3
Length Restriction	Length of the source input is not limited by the characters count. Instead is limited by the complexity. See Section 5.2.
Visualisation Capability	Linked data structures Includes, but not limited to: • Stack • Queue • Linked List • Double Linked List • Binary Tree
Platform	Windows
Implemented Language	Python 3.7+
Dependencies	• Graphviz • Lolviz

Table 1: Specification

2.3 Features and Requirements

Must haves:

- User must be able to interact with the tool via graphical user interface.
- User must be able to enter, edit, and execute the source code.
- The tool must be able to display the error message when there is a syntax error or a runtime error.
- The tool must be able to display graphs of all objects created during the execution of source code.
- The tool must be able to display graphs of the call stack in all steps of the execution.
- The tool must provide access to earlier and later steps of execution if available.

Should haves:

- The tool should highlight the current line of the execution.
- The tool should display the current state of call stack in the execution.
- The tool should automatically scroll to and show the current line.
- The tool should display the status of the tool in Ready, Executing, and Complete.
- The tool should have Left and Right key bindings to Previous and Next buttons.
- The tool should freeze the textbox when the user clicks Run and unfreeze it when the user clicks Edit
- The tool should remove any line highlight when the textbox is unfrozen.

Could haves:

- The tool could have display for the current number and the count of steps of the execution.
- The tool could have line number display next to the textbox.
- The tool could have cross-platform support.
- The tool could be packaged as an executable and bundled with its dependences.
- The tool could have syntax highlights in the textbox.

Not haves:

- The tool will not support any language other than Python 3.
- The tool will not be tested on platforms other than Windows.

2.4 User Interface

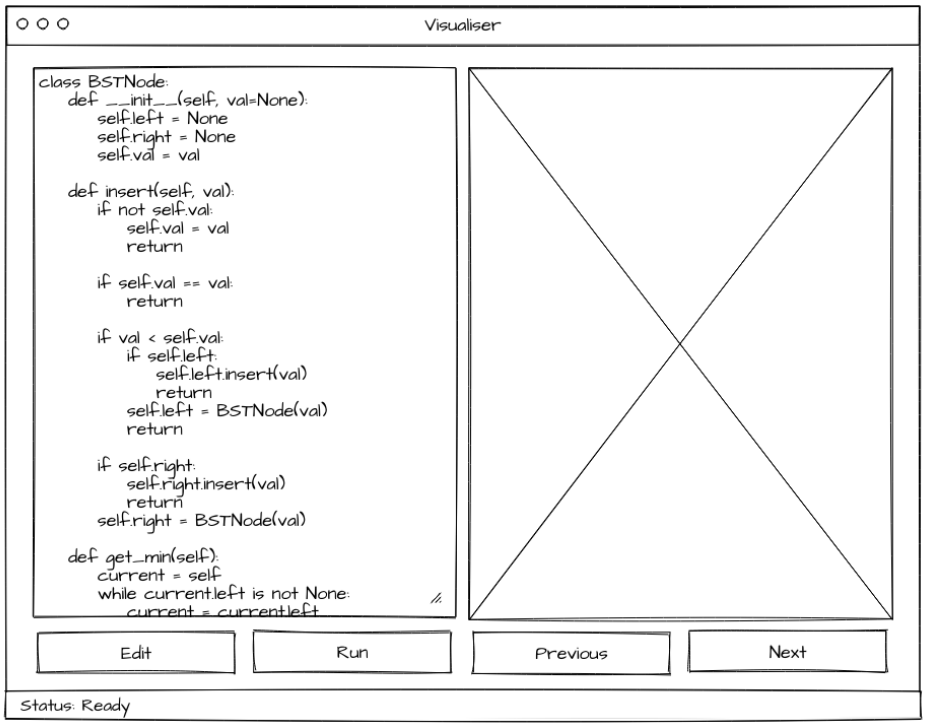

Figure 1: User Interface Design

On the left, there is a textbox allowing users to enter their source code.

To the right of the textbox is a canvas to display graphs and error messages.

The status bar on the bottom reflects what the visualiser is working on, where Ready means it is idle for code editing, Executing means it is working on graph generation and Complete means the generation is complete and the result is displayed on the canvas.

The Run button triggers the graph generation process and freezes the textbox. Once the graphing is completed, if there is a syntax error or runtime error, the error is displayed on the canvas, otherwise the corresponding line of the

first step in the execution is highlighted and the initial state of all the objects related to the user's source code.

The Previous and Next buttons travel between steps of the execution when the graphing completes, with the current line highlighted and the canvas showing the current state of call stack.

The Edit button unfreezes the textbox, clears the highlight, and turn the state of the visualiser to Ready waiting for the next execution.

2.5 Existing Work

Python Tutor [2] is a popular online visualiser that is a well-developed and mature tool. This project as well as the Lolviz package was inspired by this service. Python Tutor trustful and stable visualisation tool that supports 5 languages with known limitations. It can only handle up to 5600 bytes of code input and does not display data structures in a friendly arrangement. The project aims to fill in the gaps and offer a different solution that is offline and linked data structures oriented.

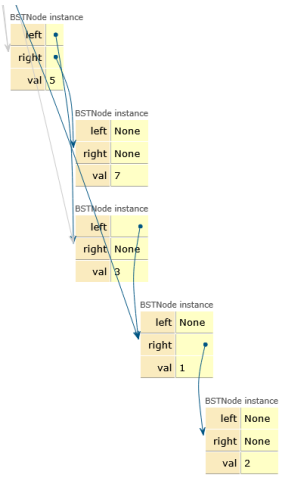

Figure 2: Python Tutor Presentation on Binary Search Tree

3 Design

3.1 Dependencies

This section discuses software, modules or packages that supports the ability to generate graphs of data structures.

3.1.1 Abstract Syntax Tree

Abstract Syntax Tree [3], AST, is a module comes with the standard installation of Python. It has functions that parse source code into a parse tree called AST that would then allow code alteration and insertion.

Python parse source code into an AST based on Python grammar, which nodes are syntactically linked to remain the structure of the source. A node in the AST typically represents a statement or an expression. These nodes have an attribute that retained the line number for reference. Statement nodes that have other statements in scope, for example Function Definition statements, store the children in a built-in Python list. The AST is essentially composed of nested list. On that account, AST can be easily altered by simple list manipulation.

3.1.2 Graphviz

Graphviz [4] is an open–source graphing software that is powerful in rendering and manipulating structural diagrams. It is capable of turning a graph written in DOT language [5] into useful formats like SVG, PNG, and PDF. The DOT language describes relations between objects. The graphs, nodes and edges can have attributes that specifies aspects such as colour, shapes, and lines. However, other than ranking objects, it is not specific in the layout. The Graphviz layout engine determines where to place the elements.

The Graphviz Python package [6] is an API to the Graphviz software. It provides `Graph` and `Digraph` class in Python that represents undirected graphs and directed graphs in DOT language respectively. The package can generate DOT source code from graph objects. The API provides the functionality to render graph objects into images using Graphviz.

3.1.3 Lolviz

Lolviz [7] is a Python package that utilise the Graphviz Python package. It generates graphs objects from a wide range of objects from built-in Python data types, arbitrary objects created by users, to call stack of the process. It also specifies ranks, shapes, colours, and styles to make Graphviz display objects nicely.

3.2 Parse and Graph Mechanism

The idea is to alter the source so that when it is executed, on top of doing what the code was intended, it also generates a graph for every step and add the corresponding line number to a list where it keeps track of the order of execution.

After taking the source input from the user, it is parsed into a parse tree with the AST module. The added code must also be parsed into AST so that it can be inserted to the source AST correctly. Recursively stepping through the AST to insert statements where necessary. Not all statements benefit from being followed by a graphing statement. While order of execution is needed for every line, graphing statements are desirable only after statements that have effects on the call stack. AST allows performing different alteration according to the type of each statement easily. This will be further discussed in detail in Section 4.3. Using the AST module also allows less error prone alteration to the source. The reason being that a successful AST parsing requires the source to be syntactically error free.

Then the altered AST is executed. The graph statements call the call stack graphing function in Lolviz to generate Graphviz graph objects. The objects are then rendered using Graphviz into PNG. These images will be stored into a folder and later displayed in the visualiser.

3.3 High-Level Architecture Overview

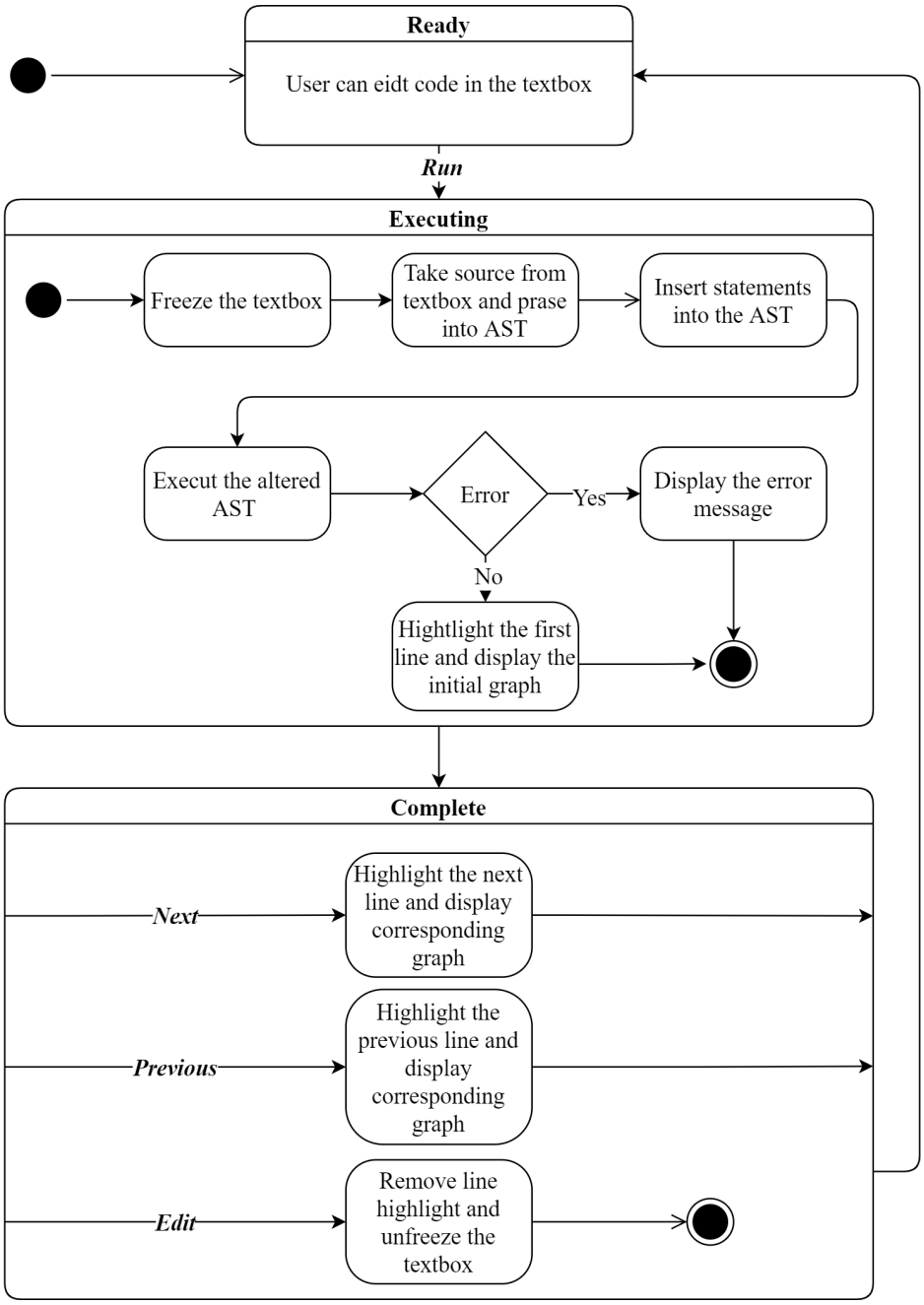

Figure 3: Event-Driven and State Design

The visualiser is designed to be an event-driven system where the GUI handles events by binding them to a function. Button click events are labelled italic in Figure 2. The machine can be described in three states: Ready, Executing, and Complete.

Ready Ready state is the initial state of the system. In this state, the user can freely edit their source code in the textbox. The system idles until the user clicks the Run button which triggers the machine to transition into the Executing state.

Executing Once the system enters the executing state, the system freezes the textbox to avoid incoherence if the text has been modified since execution started. The source code is processed to insert statements as described in Section 4.3. After the insertion is over, the processed code is then executed. On success, the execution is finished, it should have generated and stored all the call graphs in a folder, as well as returning a list of the execution order. With the list of the order of execution and the list of graphs, the visualiser highlights the first line in the execution order and displays the graph that corresponds to the line. On failure, the error is captured and displayed on the canvas instead of graphs. Either way, the system automatically transitions into the Complete state.

Complete The complete state waits for a button click event on either Next, Previous or Edit. The Next and Previous buttons would update the line highlight to the next or previous line in the execution order list respectively and display the corresponding graph. Both Next and Previous button events end with returning back to waiting. The only way to exit Complete state is the Edit button. It unfreezes the textbox, remove any line highlight and take the machine back to the Ready state.

3.4 Alternatives

The method explained in this Section is easy to implement. While it is tolerable, the amount of file saving to the secondary storage slows down the performance and there is a risk of other processes locking the file writing. There are also many other ways to accomplish the same task, but each of them comes

with different pros and cons.

One way is to generate graphs on previous and next button click events. The executing altered algorithm stop at a certain number of steps, then a snapshot of the call graph is taken and displayed. It can energy squandering to recreate if the user going back and forth, which is quite a common learn pattern.

4 Implementation

4.1 UML

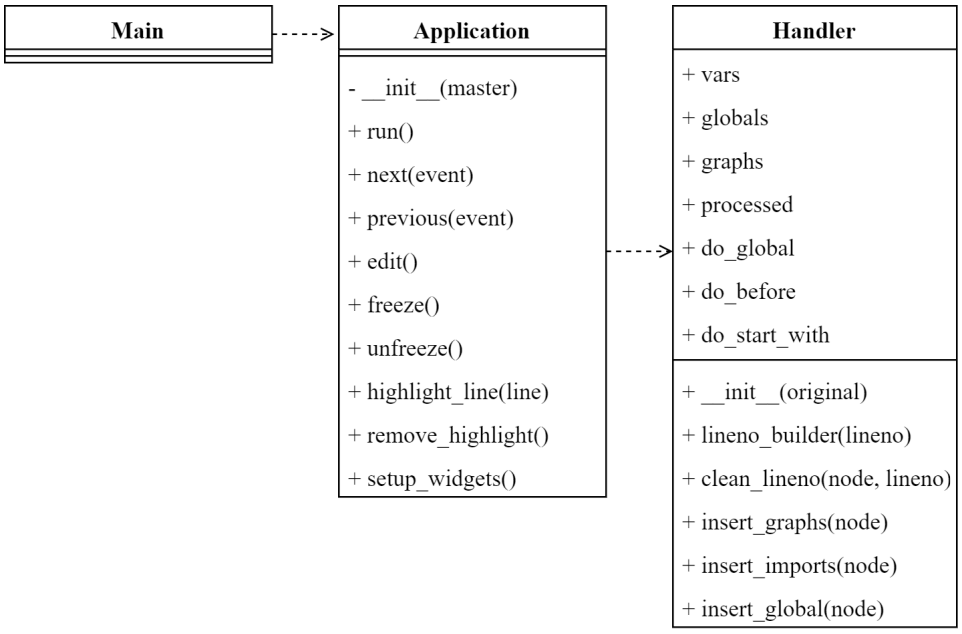

Figure 4: Class Diagram

The structure of the software shown in Figure 4 is fairly simple. It includes two class: Application and Handler. Application class is responsible for event handling in the graphical user interface (GUI). Handler class handles everything about code alteration. The Main is solely for initiating an `Application`.

4.2 Graphical User Interface: Application

The GUI framework of choice for the visualiser is Tkinter [8]. The creation of `Application` initialises all class variables, set up key bindings and calls `setup_widgets()` which creates and places all the widgets in the GUI according to Section 2.4.

The `run()` function is responsible all tasks during the Executing state described in Section 3.3. Whenever it catches an exception, it displays the error message and transition to Complete state. It first reset the variables and remove any file in the image directory. Then it creates a `Handler` and pass in the source code from the textbox. The `Handler` will process the source code as described in Section 4.3. If the error message in `Handler` is set, it is a syntax error raised by AST parsing. Otherwise, the `Handler` would have prepared the altered AST. Try execute the altered AST and catch any runtime error will be thrown during the execution. On success read all the PNG filenames in the in the image directory into the list. An empty dictionary that was passed into the `exec()` now stores the end state of variables from the execution. The list of execution order, `execution_sequence`, is then retrieved from the dictionary. The elements of the `execution_sequence` are tuples, where the first element is a line number and the second element indicates if there is an graph associated with the line. Finally, the `run()` function highlights the first line from the `execution_sequence`, displays if there is a corresponding graph, and update the state to Complete.

The `Application` keeps tracking the index of the current line in highlight and the current image in display with `index_line` and `index_img` respectively. The `next()` and `previous()` functions update the `index_line` and update the highlight. They also update `index_img` if there is a graph associated with the line and change the image of display.

After in-depth research and trials there does not seem to be a good way to have line number labelling to the textbox and this featured was not implemented.

4.3 Parse and Insert: Handler

The `Handler` manages the alteration to the source code by parsing and inserting statements into the AST. For this purpose all, code that is inserted has to be parsed. The two main goal of `Handler` is to build a list for the order of execution and graph wherever needed. Getting the order of execution can be achieved by using a list. For every line from the source, add a line that append the source line number into the list.To reflect the effect of a line, add a graphing statement after the line.

AST module has defined node classes to represent the grammar in Python. AST is a tree of node classes where children of the node is a list of nodes. Since the intension is to insert extra statements in the source code, nodes with child statements are subject to alteration. A Function Definition statement for example has a list that contains all statements under its scope. Some statements on the other hand have more than one list to separate its child statements. For instance, Try statement has one list for the statements under the try body, one for the else body, and one for the final body. Insertion has to be applied to all children recursively.

One might easily assume that inserting the extra statements after every statement will do the job. However in AST, a statement node follows the Function Definition executed after the entire function. A statement node comes after a return node is never executed. It turns out there are statements that do not benefit to have inserted statements as a sibling node. There are also statements that would be problematic if the line number append statement comes after it. Therefore, statement classes are labelled into four types.

Type1 Statements that have their local namespaces hence global variables has to be declared before use. The global variable declarations are inserted as their first child.

Type2 Statements that do not execute statements after the statement or mess up the execution order record if inserted after.

Type3 Statements that require the line number append statement as their first child instead of as their following sibling.

Type4 Any statement that is neither Type 2 nor Type 3.

Table 2 explains the position relationship of each type of nodes with global declarations, append statements, and graphing statement. Note that a statement can be labelled with multiple types yet Type 4 is mutually exclusive to the union of Type 2 and 3. As an example function declaration is of Type 1 and 3.

Statement Type	First in children	Before	After
1	global declarations		
2		append	
3	append, graphing		
4		append	graphing

Table 2: Location of Insertions for Different Types of Statement

If there is not any error from parsing the source code, the `Handler` first stepped through the source AST once to collect all the variable names that is declared into a list called `vars`. This collection is essential because calling `exec()` to execute the AST is considered the same process. Using `callsviz()`, a call chart graphing function in Lolviz, without passing in the variable list will lead to graphing all variables existing in visualizer including those imported form packages.

At the first level of the tree, the handler inserts the import statement of the Lolviz package, initialises a global graph counter, and initialises the global list `execution_squence` for execution order at the beginning of the list. The graph counter is used for naming the files when Graphviz renders the PNGs.

While stepping through the AST, Global declaration statements, append statements, and graphing statements are inserted to the positions as presented in Table 2. The line number of a node can be accessed to build the line number append statement. The graphing statements are made up of three step: graph generation, set the image formats, and rendering the graphs into images. Graph are generated by passing `vars` into `callsviz()` to control which variables are included in the call graph. The filenames of the images are numbered in chronological order by keeping a `png_counter` updated and are kept in same length by filling 0s.

After the `Handler` processes the following simple source code Algorithm

1, the altered AST is unparsed to exhibit the effect of the alteration as shown in Algorithm 2. The final state of `execution_sequence` should be `[(1, 0), (4, 1), (1, 1), (2, 0)]`.

Algorithm 1 Example Source Code

```
1 def foo():
2     pass
3
4 foo()
```

Algorithm 2 Example Altered Code

```
1  from lolviz import *
2  import os
3  global png_counter
4  png_counter = 0
5  global execution_sequence
6  execution_sequence = []
7  execution_sequence.append((1, 0))
8  def foo():
9      global png_counter
10     global execution_sequence
11     execution_sequence.append((1, 1))
12     mySpecialGraph = callsviz(varnames=['foo'])
13     mySpecialGraph.format = 'png'
14     png_counter += 1
15     mySpecialGraph.render(os.path.join(os.path.
           curdir, 'png', str(png_counter).zfill(4)))
16     execution_sequence.append((2, 0))
17     pass
18 execution_sequence.append((4, 1))
19 foo()
20 mySpecialGraph = callsviz(varnames=['foo'])
21 mySpecialGraph.format = 'png' png_counter += 1
22 mySpecialGraph.render(os.path.join(os.path.curdir
     , 'png', str(png_counter).zfill(4)))
```

5 Evaluation

5.1 Testing

In the design stage, packages and dependencies has been tested thoroughly to verify that they fit the purpose. Figure 5 shows that Lolviz has proven its ability on custom and arbitrary objects. With little worry on the crossing arrows, the project settled on its call graph generating feature since it has provided colour coordination with different type of objects.

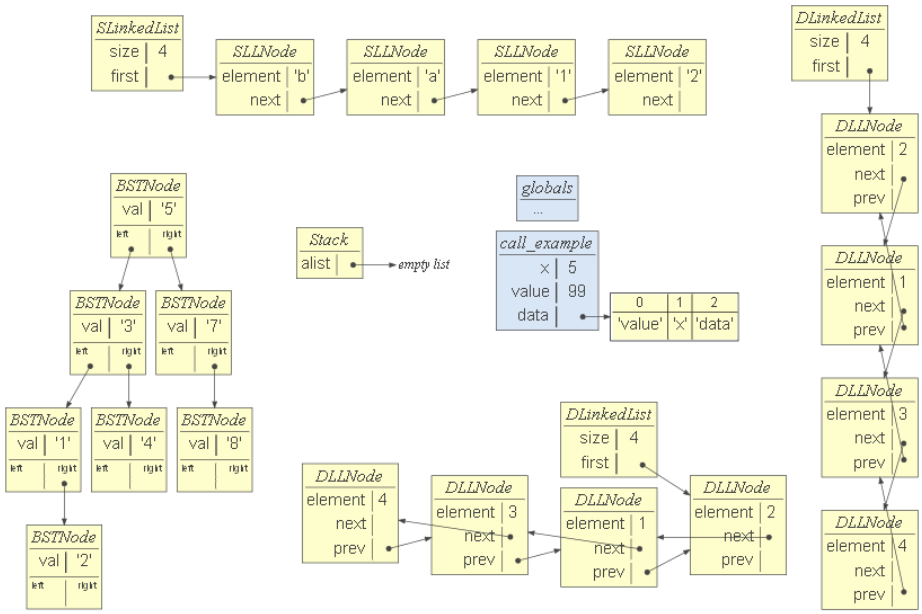

Figure 5: Tests Performed on Lolviz

Throughout the developing stage, the parse and graph mechanism was tested countless times. Other than basic statements such as Algorithm 1, data structures mentioned in the specification table in Section 2.2 were all implemented for testing purpose. The parsing tests were done by both printing the altered AST, unparsed version of the AST and the `execution_squence`. The doubly linked list is purposefully implemented wrong to prove the graph illustrated in Figure 6 can show the problem.

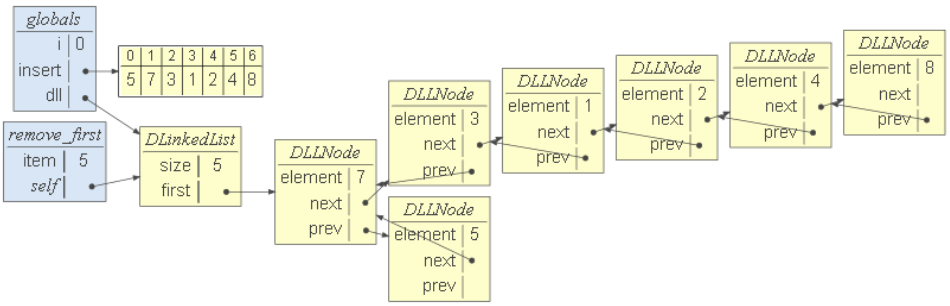

Figure 6: Testing Handler on Doubly Linked List that Fails to Remove a Node

The mentioned data structured was all tests on the final product for GUI testing. Syntax error and runtime error examples was also tested to prove that error massage is shown in the canvas.

There is one unsolved issue that was founded during testing and it is be described in Section 5.3.

5.2 Limitations

The visualiser can handle almost any arbitrary code with some trade-offs. As reported in Section 2.2, there are limitations on the complexity of the source code for several reasons. Python limits the recursion depth of 1000. The visualiser would not be able to handle AST of height exceeding 1000 since the insertion requires recursively stepping through the entire tree. The source code is also limited to 9999 graphs generated. PNG filenames will be read in correct order if the are all 0s filled to the same length. The decision made in this case is 4 digits. Source code written by students rarely generates thousands of graphs. Generate huge number of graphs can cause the visualiser to freeze for a period of time. However, the visualiser lacks of a mechanism to avoid lengthy input.

The user should avoid using `if __name__ == "__main__"` because the alternated code is not being run as top-level environment of the programme. There are also limitations on the naming of variables and functions. Although trying the best to avoid collisions with comprehensive names, `png_counter`, `execution_sequence`, `mySpecialGraph`, and functions names that

present in the Lolviz package are reserved. Naming variable `node` causes unexpected issues and should be avoided until it is solved (see Section 5.3).

Standard output displayed in the command line that executed the application.

Never ending algorithm will not exit the Executing state and therefore not recommanded.

It is not recommanded to be running from a folder that is access by synchronization service, like Dropbox or OneDrive, to avoid files used by another process.

5.3 Issues

There is one issue that was discovered on variable naming. Naming a variable `node` in the source input sometimes causes syntax error raised by Graphviz during execution. It appears to only happen when two conditions are both met. One being that the source code has once parsed by AST and the other one being that there was multiple declaration on `node` in different namespace. Algorithm 3 is one example that results in throwing an error. The reason is remained unknown.

Algorithm 3 Example Source Code with Naming Issue

```
1 class Foo:
2    def __init__(self):
3        self.node = 1
4 node = Foo()
```

5.4 Final Product

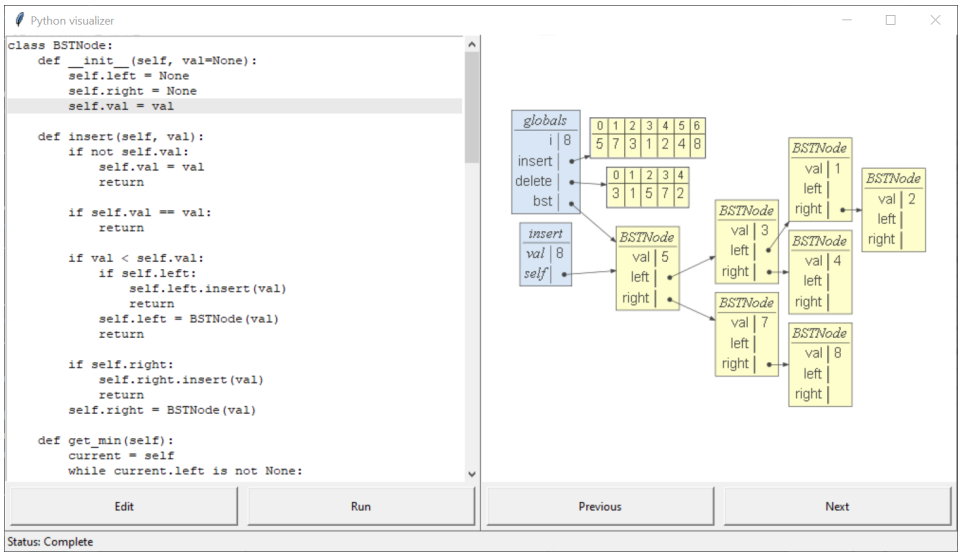

Figure 7: Python Visualiser in Use with Binary Search Tree

The final work shown in Figure 7 has completed all must haves and should haves features listed in Section 2.3. The visualiser was only intended to show the transformation of linked data structures in execution. It went from try to just display the most basic data structures to now that it is capable of handling algorithms such as recursions, classes object-oriented programming, which is believed to be sufficient for educational uses.

The UI design is quite clean and straight forward. Students nowadays are tech-natives may already have experienced similar application. Users should be familiarized with it fairly soon since there are not many options. The elements in the graphs are very well placed as seen in the Binary Search Tree example in Figure 7. Though it is not the scope of this project, modernising the look of the GUI would make it much more appealing.

Standard output displayed in the command line that executed the application becomes the shortcoming of the system. It was short-sighted not planning this into the design.

5.5 Possibilities

This is a vastly expendable project that can become a full-fledged practical software. There could be some basic text editor features, syntax highlights, 21st century aesthetic themes, and colours. It is also nice to see that it execution steps displayed. It would be splendid if it is packaged into an executable bundled with all the dependencies, and keep all the image files in the temporary files directory. Ultimately it can be expended to do the same for code in other languages on other operating systems.

6 Conclusions

6.1 Self Reflection

I realised most time-consuming part of conducting a project is the preparation and researches. There are infinite libraries and frameworks available, but one cannot afford to try all of them to determine what is the best option within the project time frame. For trying out each of the new technology it comes with a huge price of spending hours to learn that it is not capable for my purpose. Once all the components are tested and ready, it does not take much time to piece them together.

Prior to the project, I had no knowledge about GUI and AST. This project has proven that I am capable to self-educate on new topics and materials. I have acquired the skill to build software using technologies that I am not familiar with.

I found myself struggle a lot with structuring long sentences that explains complicated logics and relationships in technologies. I believed myself am not a writer, but writing turns out to be very helpful of ordering ones thought and found details and logics that I would not otherwise have discovered.

The project could be much better documented and commented. There was no straight development methodology that was followed, but logs, ideas and findings were jotted weekly. I find thing these notes are quite important. It saved me from having to warm up and get familiar with the progress I have

made every time.

6.2 Timeline

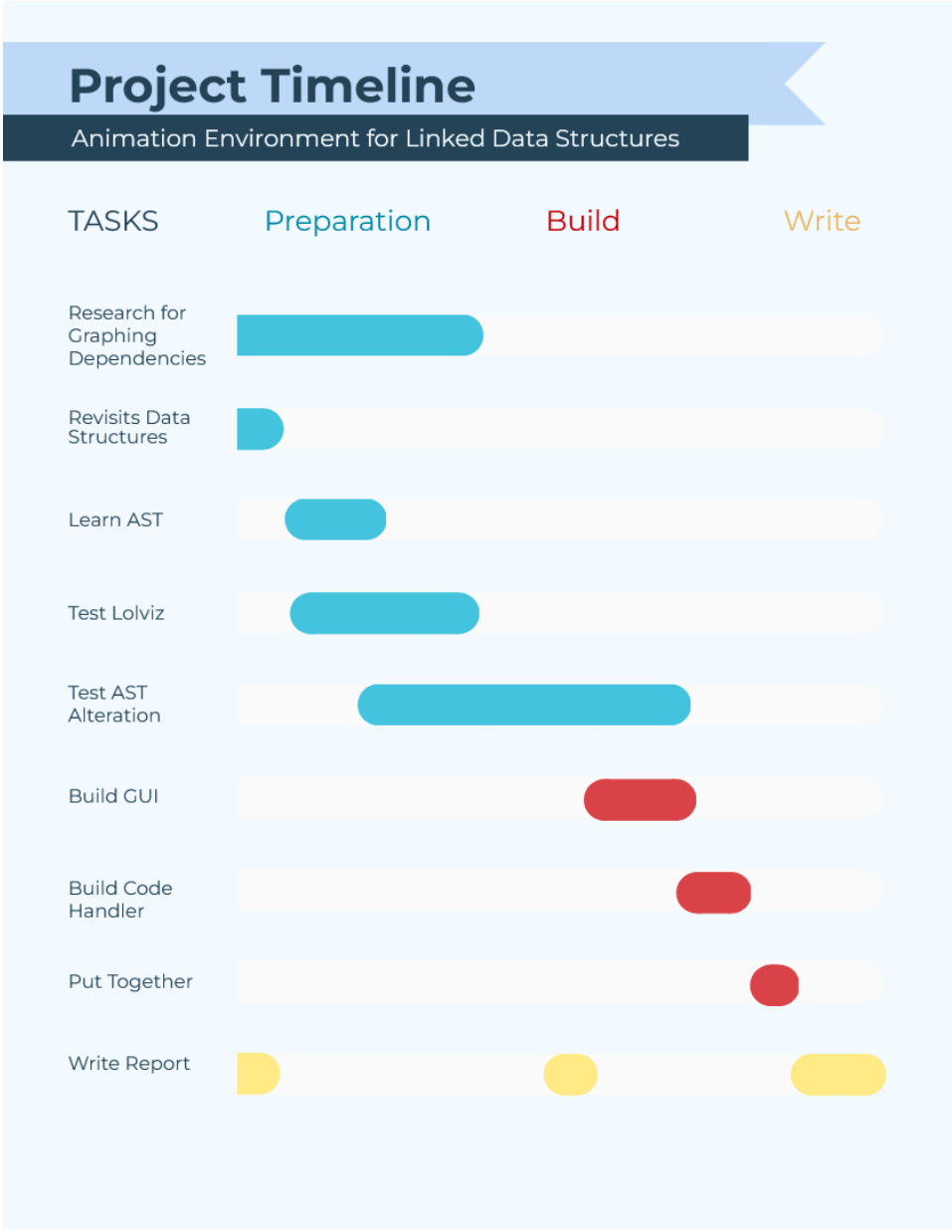

Figure 8: Gantt Chart

There was no date precise documentation of what task has been achieved through out the project. The Figure 8 is roughly proportion to how the year was spent.

References

[1] "Call stack," Mar. 2022, page Version ID: 1078532188. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Call_stack&oldid=1078532188

[2] P. Guo, "Python Tutor - Visualize Python, Java, C, C++, JavaScript, TypeScript, and Ruby code execution." [Online]. Available: https://pythontutor.com/

[3] Python Software Foundation, "ast — Abstract Syntax Trees — Python 3.10.4 documentation." [Online]. Available: https://docs.python.org/3/library/ast.html

[4] AT&T Research and Lucent Bell Labs, "Graphviz." [Online]. Available: https://graphviz.org/

[5] "DOT (graph description language)," Feb. 2022, page Version ID: 1072201904. [Online]. Available: https://en.wikipedia.org/w/index.php?title=DOT_(graph_description_language)&oldid=1072201904

[6] S. Bank, "Graphviz," Apr. 2022, original-date: 2014-01-12T17:49:29Z. [Online]. Available: https://github.com/xflr6/graphviz

[7] T. Parr, "parrt/lolviz: A simple Python data-structure visualization tool for lists of lists, lists, dictionaries; primarily for use in Jupyter notebooks / presentations." [Online]. Available: https://github.com/parrt/lolviz

[8] Python Software Foundation, "tkinter — Python interface to Tcl/Tk — Python 3.10.4 documentation." [Online]. Available: https://docs.python.org/3/library/tkinter.html