Handover Documentation

Algorithms in Action – Team 30

Contents

Project Background / Overview

Lists are often used an auxiliary data structure in complex algorithms with defined operations and interfaces that can be pluggable in other algorithms. During the project, the team created a list package and linked list package with initialisation, insertion, deletion, and search operations. Furthermore, we created an example implementation of the linked list package using the merge sort algorithm.

All our changes were made to the existing Algorithms in Action repository, and as such, by loading up the latest version of the repository, you will be able to see our changes. If you want an in-depth understanding of how to use our files, please refer to the 'Understanding Our Changes' section to get and understanding of how to use our packages.

A summary of the changes we have made is listed below. The list of changes has been edited for clarity and brevity, and a detailed description of our changes can be found the 'Changelog' section.

- Created Linked List Package to support the visualisation of linked lists
- Created List Package to support the visualisation of lists
- Visualised merge sort with the Linked List Package

Future improvements to the project that are recommended also include:

- There are still issues with backward movement of the merge sort algorithm for the linked list, and it is recommended for this to be fixed for a fully functional merge sort algorithm.
    - Recursion and depth parameters would need to be used and implemented to keep track of layer.
- Pseudocode conditionals are not fully matched up with algorithm.
    - Would likely need to include the data nodes within the algorithm controller and perform data manipulation to allow for conditionals. Setting conditionals

Demo

To visualise our changes, please use the Algorithms in Action repository linked here and navigate to the main README file. After following the instructions to set up the environment, install dependencies and start a local server, users can navigate to the Linked List Merge Sort page for a demo or our project.

User Stories

User stories have been developed through a combination of official project specification, client requirements and ideas from the team members:

User Story	Description	Priority	Origin
1	As a client, I would like a list package with defined operations and interfaces which are useable within algorithmic animations	Must-Have	Project Specification
2	As a client, I would like a linked list package with defined operations and interfaces which are useable within algorithmic animations	Must-Have	Project Specification
3	As a student, I would like to view the algorithm for merge sorting two linked lists	Must-Have	Project Specification
4	As a student, I would like to view a render of a linked list from an input	Must-Have	Client
5	As a student, I would like to view the instructions before I run the algorithm so I have a clear idea of how the system can be customized	Must-Have	Team
6	As a student, I would like to view background information on linked lists	Must-Have	Team
7	As a student, I would like to see the pseudocode and animation to be synchronized	Must-Have	Team
8	As a programmer, I would like all changes to follow the ethical and security guidelines set out by the team	Must-Have	Team
8	As a student, I would like to search through a linked list for a number	Should-Have	Client
9	As a student, I would like to be able to edit the linked lists at the beginning, middle and end before re-running the algorithm	Should-Have	Client
10	As a student, I would like access to extra-information about the algorithm	Should-Have	Team
11	As a programmer, I want documentation in the repository so I can recreate linked lists	Could-Have	Client
12	As a student, I would like to see colours to help me understand the algorithm	Could-Have	Team

13	As a student, I want the program to be user friendly	Could-Have	Team
14	As a student, I want the program to be reliable and not crash	Could-Have	Team

System Requirements

Minimum System Requirements:

Project deployment requires no additional system requirements beyond standard ones utilised for the basic Algorithms in Action website. Since Algorithms in Action is written in JavaScript, using the React framework, this includes having Node.js installed. Furthermore, you need to ensure you have a node version 18.x or higher, npm version 9.x or higher, and python 3.x or higher for the project.

Further information about how to run the project locally can be found on the main README file.

Installation Guide

Installing the project can also be done through the standard instructions listed on the main README file.

Understanding Our Changes

Linked List Package

As per the project specification, the Linked List package was created to be pluggable into other algorithm visualisations. Down below is documentation to understand, setup and edit the package as needed.

Setting up a Basic Linked List Algorithm

The merge sort for linked lists is an example of how to use the linked list package. However, you can also follow the steps below to use the linked list package:

- Setup a new algorithm by following the readMe file in src/algorithms/README
- Import the linked list package in the controller file

```
import LinkedListTracer from "../../components/DataStructures/LinkedList/LinkedListTracer";
```

- Setup an instance of the linked list by initialising a LinkedListTracer

```
initVisualisers() : {list: {...}} {
    return {
        list: {
            instance: new LinkedListTracer( key: 'list', getObject: null, title: 'List Prototype',
                options: { arrayItemMagnitudes: true }),
            order: 0,
        }
    };
},
```

- Use the methods defined in the Tracer file to add frames to the visualisation. I.e.

```
run (chunker, {nodes}) : void {
    const A : any[] = [...nodes];
    let n = nodes.length;

    // Initialise
    chunker.add(
        1,
        (vis, list) : void => {
            vis.list.addList(list);},
        [nodes]
    );
}
```

To understand what other functionality is available, refer down below to documentation on the Tracer file in Understanding the Linked List Package

Understanding the Linked List Package:

There are three main files in the Linked List package:

- LinkedListRenderer.module.scss (CSS file)
- Index.js (Renderer file)
- LinkedListTracer.js (Tracer file)

CSS File:

This file is a SCSS stylesheet that defines the styles and layout for visualising linked lists. The file applies various layout, styling and colour properties to different elements of the linked list visualisation. Developers should edit this file if they want to change stylistic elements such as the ones listed before. Below is a diagram which shows how a linked list is coded.

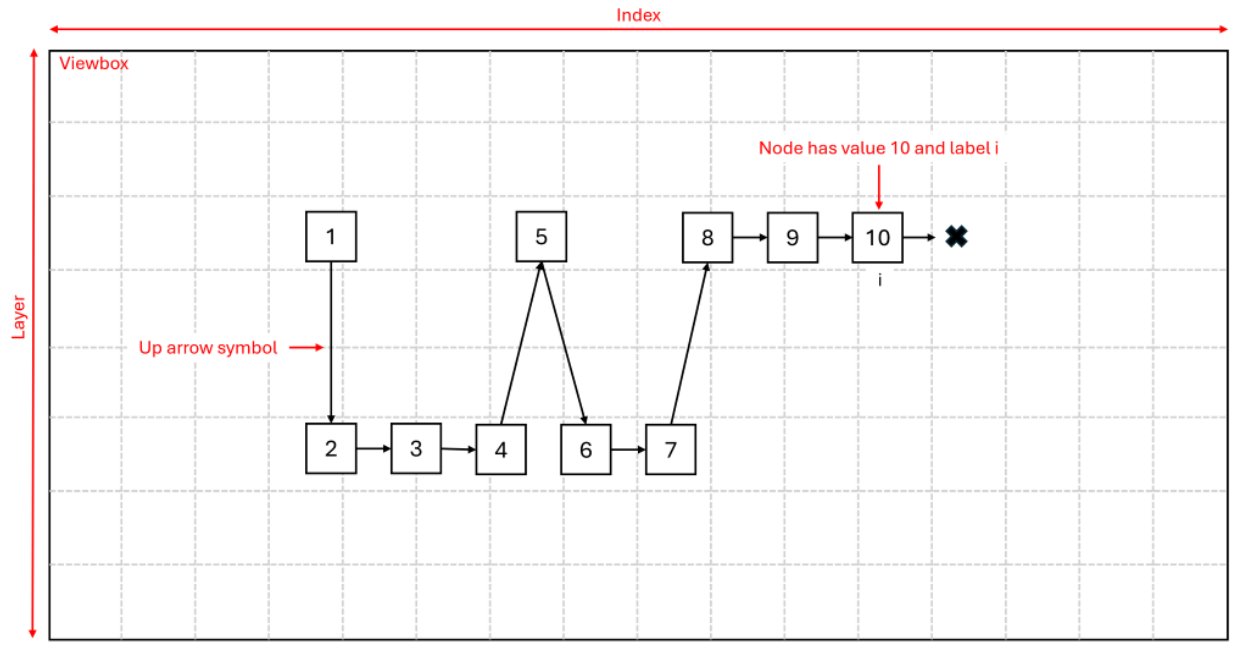

Diagram: Visualisation of Linked List in Algorithms in Action.

As shown above the viewbox is split into different indices and layers. The indices and are the 'x' and 'y' coordinate respectively. Each layer stacks vertically in layers.

Elements:

The linked list is made up of a variety of elements; nodes and symbols. Each node can have a value and label. Whereas each symbol can be an arrow or null pointer. The CSS file contains the code required to change the background colour, border size, size, font and alignment of each of the elements. The CSS file can also control the margin between linked lists in the elementContainer class.

The CSS file applies dynamic styles for different node states. For example when they are '.visited' and '.selected', they have different background and border colours.

Centering:

The file organises elements and containers using flexbox. When a linked list is rendered, it automatically centers the nodes horizontally through the 'justify-content' line and 'display' lines. The 'display' for indices, layers and nodes is set to flex which means it automatically calculates the distance of the viewbox and the number of nodes to 'center' the linked list horizontally.

However, the location of symbols, are determined relatively to location of the nodes. Its position is determined based on the center of the node coordinates.

The linked list can also be moved manually. This is made possible through the 'display' lines in the indexContainer, layerContainer and elementContainer.

Colour Variables:

The stylesheet also uses CSS variables such as '--linked-list-node-bg' to allow for easy customisation of colours. These can be edited at the bottom of the CSS file.

Renderer File:

This file defines the React component, 'LinkedListRenderer' that visualises linked lists using animation and SVG graphics. Developers should edit this file if they want to change how the linked lists are animated or change stylistic components such as the size between nodes. For information about animations, refer to documentation from the framer motion website.

Rendering Data:

In the diagram, the 'renderData' method retrieves data from the const { lists } and organises them into layers based on their listIndex to handle rendering in a structured manner; the file iterates over layers and then lists. Each layer has a set of linked lists which has the nodes.

In the method, the return() method renders the layout. There are a few key components which work together. Namely:

<AnimateSharedLayout> which is a component from framer-motion which enables nodes to be added, removed or rearranged

<motion.div> which is the container for the entire linked list, and allows it to be draggable due to the 'drag' property

this.renderSymbols() which lets the 'renderData' method to access the reuseable SVG symbols such as arrows and null markers from the 'renderSymbols' method

Rendering Nodes and Variables:

In the 'renderData' method, the 'React.Fragment' is responsible for rendering the nodes and variables. The 'whileHover' line causes each node to grow slightly when hovered. The nodes also have values and labels with the stylistic components being stored in the CSS file. Note for the nodes, there are dynamic styles applied for when the node is in a 'selected' or 'patched' state.

Rendering Arrows and Null Markers

The arrows and null markers are created in the 'renderSymbols' method. These symbols are created once in the and then referenced multiple times using the <use> tag in 'renderData'.

Tracer File:

The tracer file provides a comprehensive interface for managing and visualising linked lists, with support for:

- Adding, deleting and modifying lists and nodes
- Sorting, merging, and splitting lists

- Managing visual attributes like selection, arrows and variables
- Synchronising the internal state with an external chart tracer

Developers should change this file if they want to edit or add methods to update the linked lists values and labels, sorting functionality etc. Below is a short description of the purpose of the methods.

Core Renderer Integration

- getRendererClass():
    - This returns the LinkedListRenderer class to be used as the renderer for visualising linked lists.
- init():
    - Initializes the tracer by setting up its properties:
        - key: Tracks unique IDs for lists.
        - nodeKey: Tracks unique IDs for nodes.
        - chartTracer: Placeholder for chart synchronization (if any).
        - lists: Array to store all linked lists.

Linked List Management:

- addList():
    - Adds a new linked list and pushes the constructed list into a list of lists called this.lists
- deleteList():
    - Removes a list identified by its unique key.
- findList():
    - Locates a list based on its listIndex and layerIndex.
- findListbyNode():
    - Finds the list containing a node identified by key.
- shiftRight():
    - Shifts the list to the right visually by a unitShift.
- moveList():
    - Moves a list to a new position at newIndex. This has two methods: stack and insert. Stack adds it to the next available layer and insert shifts the lists to the right before inserting.
- updateIndices:
    - Rearranges list indices to fill gaps caused by deletions.
- getMaxIndex & getMaxSize:
    - Returns the highest listIndex or the largest size among the lists.

Node Management:

- createNode(value)
    - Creates a new node of a specific value.
- appendToList():
    - Appends a new node to a specified list and updates its size.
- addToList():

    - Adds a value to a specific list by index.
- removeAt():
    - Removes a node at a given index in a specified list, adjusting connections and size.
- swapNodes():
    - Swaps the values of two nodes in the same list.

List Operations:

- sortList():
    - Sorts a list (containing node1) based on node values.
- mergeLists():
    - Merges two lists containing the specified nodes into one.
- splitList():
    - Splits a list at the specified node into two separate lists.

Visual Operations:

- setArrow() and resetArrows()
    - setArrow() sets the directional arrow for a node as either up, down or diagonal, whereas resetArrows() resets all arrow directions in the list at the specified node.
- patch() and depath()
    - patch() highlights nodes between startIndex and endIndex in a list, whereas depatch() removes the highlight from a specific node.
- select() and deselect() and clear select:
    - select() marks nodes in a range as selected. deselect() deselects nodes in a range. clearSelect deselects all nodes in all lists.
- addVariable() and removeVariable():
    - The addVariable() adds a label to the node whereas removeVariable() removes the label from the node.
- clearVariables() and assignVariable():
    - The clearVariables() method removes all labels from the node whereas assignVariable() assigns a label to a specific node, ensuring no other node has that variable.

Synchronisation and Output:

- SyncChartTracer:
    - Updates the chartTracer to reflect the current state of the linked lists.
- StringTheContent:
    - Retusns a string representation of all linked lists, showing their values in order.

List Package

As per the project specification, the List package was created to be pluggable into other algorithm visualisations. Down below is documentation to understand, setup and edit the package as needed.

Setting up a Basic List Algorithm

Setting up the List package is the exact same as the Linked List package. As such, please refer to documentation from the Linked List package. To understand what other functionality is available, refer down below to documentation on the Tracer file in Understanding the List Package

Understanding the List Package:

There are three main files in the List package:

- ListRenderer.module.scss (CSS file)
- Index.js (Renderer file)
- ListTracer.js (Tracer file)

As the CSS file and Renderer file for the Linked List Package and List Package is very similar. Please refer to documentation on the Linked List Package to understand how to edit stylistic elements (CSS file) or change how they want lists to be visualised (Renderer file).

Tracer File:

The tracer file provides a comprehensive interface for managing and visualising lists, with support for:

- Adding, deleting and modifying lists and nodes
- Sorting, merging, and splitting lists
- Managing visual attributes like selection, arrows and variables

Developers should change this file if they want to edit or add methods to update the linked lists values and labels, sorting functionality etc. Below is a short description of the purpose of the methods.

Core Renderer Integration

- getRendererClass():
    - This returns the ListRenderer class to be used as the renderer for visualising lists.
- init():
    - Initializes the tracer by setting up its properties.

List Management:

- addList():
    - Adds a new list and pushes the constructed list into a list of lists called this.lists
- clear() and clearAll()
    - clear() clears all nodes and labels in a specific list whereas clearAll() clears all lists and resets the nextListIndex.

Node Management:

- addNode() and removeNode():
    - Creates a new node of a specific value and removes a node at a specified position if valid.
- set():
    - Replaces all nodes in a specific list with the provided values.
- searchNode():
    - Searches for a node with a specified value in a given list and returns it if found.

List Operations:

- sortList():
    - Sorts a list (containing node1) based on node values.
- mergeLists():
    - Merges two lists containing the specified nodes into one.
- splitList():
    - Splits a list at the specified node into two separate lists.
- reverse():
    - Reverses the order of elements in a given list.
- swapElements():
    - Swaps the elements at indices i and j in the specific list.

Visual Operations:

- patch() and depath()
    - patch() highlights nodes between startIndex and endIndex in a list, whereas depatch() removes the highlight from a specific node.
- select() and deselect():
    - select() marks nodes in a range as selected. deselect() deselects nodes in a range.
- addLabel() and removeLabel():
    - The addLabel() adds a label to the node whereas removeLabel() removes the label from the node.
- clearLabels():
    - The clearLabels() method removes all labels from the node.

Changelog

Sprint 1:

1. Setup the files for the new LinkedListTracer, LinkedListRenderer and LinkedListModule files as per User Story #2.
2. Setup the files for the new ListTracer, ListRenderer and List Module files as per User Story #1.
3. Setup new files in the controller, pseudocode, parameters are other files as per User Story #3.
4. Implemented pseudocode for the linked list mergesort (Removed after revision from client)
5. Changed the controller tab to have the main logic for the mergesort module (Removed after revision from client)

Sprint 2:

After consulting with the client in our second client meeting, we realised that many of our changes to the controller and other files were redundant as the client wanted the linked list package and merge sort visualisation in a specific way. As such the progress for our initial renders had to be scraped. The full list of changes made during this sprint include:

1. Implemented prototype files from client for User Story #1, #2 and #3. This also enabled functionality from User Story #4, #5 and #6 automatically.
2. Changed the Linked List data structure to include the same functionality as the Array2DTracer including changing functions such as append, prepend, select and swapping nodes as per User Story #1.
3. LinkedListTracer file was changed to support animation, patching, filling and other visual aspects including functions assignVariable and syncChartTracer() as per User Story #7.
4. Changed the LinkedListRenderer and ListRender to support multiple linked lists and multiple lists respectively as per User Story #3.

Sprint 3:

1. Added right shift functionality for lists in the tracer file to move the lists after being split to support User Story #3.
2. Implemented functionality to split the lists as per User Story #3.
3. Implemented functionality to have lists layered vertically to support User Story #13.
4. Fixed issues around linked lists initiation in tracer and renderer, amongst other issues with multiple lists, class separation, split functionality, and label rendering.
5. Added more functionality into the ListTracer from the 2DArrayTracer to make the code more consistent as per User Story #1.
6. Added arrows on the nodes and added code to ensure the arrows could be displayed in different directions as per User Story #13.
7. Implemented testing for simple cases including sorting an empty list, sorting negative numbers, sorting invalid inputs, etc.

Testing

To test the list package, linked list package as well as the algorithm, A testing suite were created to assess the functionality, reliability and accuracy of our modules. A testing controller "msort_linkedlist.js.test" were added under the controller folder, and the following inputs were tested on linked list merge sort module:

- Empty set
- Single element
- Ascending/ Descending list
- Odd/ Even numbers list
- List with duplicates
- List of negative/ positive and negative numbers
- List of identical elements invalid inputs (elements of other types)
- Decimal numbers

Upon running the tests, it was observed that the module failed to produce correct results when handling edge cases such as empty sets, single elements, and invalid inputs. Necessary changes were made to the algorithm controller accordingly and the issues have been addressed

Traceability Matrix

Requirement description	**Test Case ID**	**Status**
No additional system requirements beyond standard setup for Algorithms in Action website.	Setup verification (not in code)	Completed
Requires Node.js version 18.x or higher, npm version 9.x or higher, and Python 3.x or higher for project deployment.	Setup verification (not in code)	Completed
Sorts an empty list	it('sorts empty list')	Passed
Sorts a single-element list	it('sorts single element')	Passed
Sorts an already sorted list	it('sorts sorted list')	Passed
Sorts a reverse sorted list	it('sorts reverse sorted list')	Passed
Sorts a list of odd numbers	it('sorts odd numbers list')	Passed
Sorts a list of even numbers	it('sorts even numbers list')	Passed
Sorts a list with duplicate elements	it('sorts list with duplicates')	Passed
Sorts a list with negative numbers	it('sorts negatives')	Passed
Sorts a list with both negative and positive numbers	it('sorts negatives and positives')	Passed
Sorts a list where all elements are identical	it('sorts all identical list')	Passed

Exits upon encountering invalid inputs	it('exits with invalid inputs')	Passed
Handles decimal numbers	it('handles decimals')	Passed