UE18CS390B – Capstone Project Phase – 2

SEMESTER - VII

END SEMESTER ASSESSMENT

Project Title : Data Structure and Algorithm Visualizer
Project ID : PW22KKV01
Project Guide : Prof. Kusuma KV
Project Team : R.Shrenik, Niranjan Bhaskar K, Vidhisha Shankar

Outline

- Abstract
- Team Roles and Responsibilities.
- Summary of Requirements and Design (Capstone Phase - 1)
- Summary of Methodology / Approach (Capstone Phase - 1)
- Design Description
- Modules and Implementation Details
- Project Demonstration and Walkthrough
- Test Plan and Strategy
- Results and Discussion
- Lessons Learnt
- Conclusion and Future Work
- References

Abstract

The project implements linear and nonlinear data structures, graph algorithms and a sorting algorithms visualizer in an interactive, user-operable, animated web user interface to help users understand foundational problem-solving concepts in Computer Science. The system allows user to input their own data to create data structures and see popular algorithms working.

Team Roles and Responsibilities

Team Member	Phase 1 contribution	Phase 2 contribution
R.Shrenik	Literature survey, 3-layer web standards design model -> transition to MVC architecture, implemented linked lists, helped with testing	Worked on animation controls, sorting visualizer v1 and v2, graph algorithm visualizer v1 and v2, binary trees, binary search trees, implementation of code tracing and wrote tests
Niranjan Bhaskar K	Literature survey, 3-layer web standards design model -> transition to MVC architecture, implemented stacks, helped with testing	Worked on animation controls, sorting visualizer v1 and v2, graph algorithm visualizer v1 and v2, balanced binary search tree, conversion of trees, implementation of code tracing and wrote tests
Vidhisha Shankar	Learnt CSS and JavaScript, literature survey, interface diagrams, wrote tests	Worked on sorting visualizer v1, graph algorithm visualizer v1, search algorithms, wrote tests

Summary of requirements and Design - Phase 1

Requirements:

- Creation of a robust project model and file organization
- Design of UI prototypes and creation of linear data structures (linked lists, stacks, queues) and visualizing them
- Implementing animation controls

Review of strengths and weaknesses of existing implementations:

- Strengths: Highly robust implementations, written by engineers with >10 years of software engineering experience
- Weaknesses: Mostly Java-only implementations based on our survey
- Desktop applications => Need to deal with OS-specific problems, lower reach to our target group, our lack of experience building desktop apps
- Low real-world applicability of projects, specific use cases (Eg. Sierpinski triangles, recursion-only implementations)

Summary of requirements and Design - Phase 1

Design considerations

- Interactive UI is required. Can’t display just static content as this is key to understanding the concepts.
- Animation controls need to work real-time. Skipping through frames should not break the visualization.
- Security: We’re still in the process of learning how to write secure code and prevent foreign extensions from manipulating data on the website
- Availability: The project doesn’t use up a lot of resources but availability depends on our hosting plan as well (upon deployment)
- Privacy: We collect no data whatsoever => No privacy issues

Summary of requirements and Design - Phase 1

Other constraints, assumptions, dependencies and risks:

- No mobile-screen support
- Cannot interface with other applications. No API support / 3rd-party access allowed
- We’re still in the process of learning how to write more secure code (how do we not allow foreign extensions to manipulate data and attacks)
- Internet access is required

Summary of Methodology / Approach - Phase 1

Design approach 1:

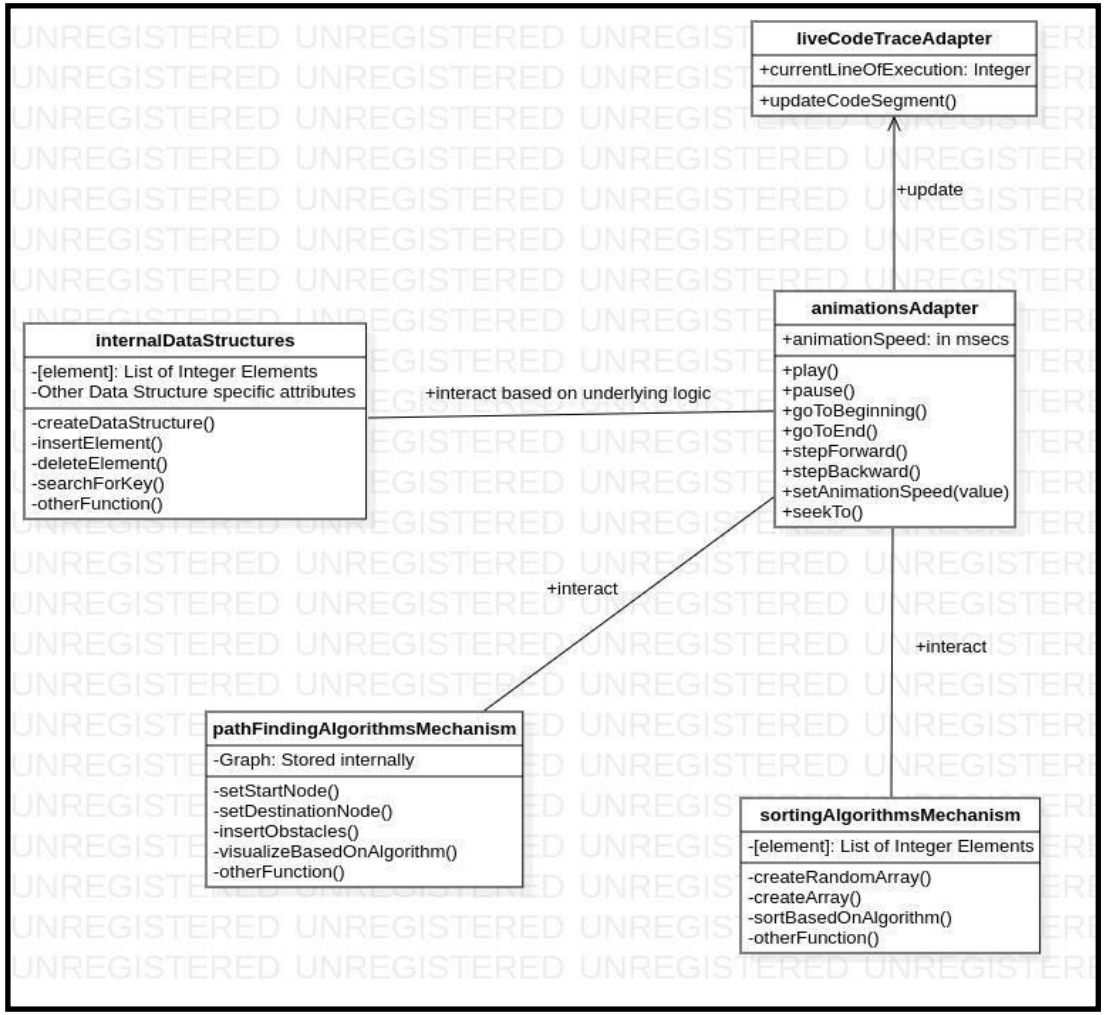
- 3-layer web standards model (UI layer, visualization logic, widgets)
- Straightforward approach (custom HTML and JS files, separate animation logic)
- Drawbacks:
  - Functions (start, end, algo) were doing the rendering and not the widgets => cannot reuse code. Have to reimplement UI every time we want to add another data structure/algo.
  - Bad animation efficiency: library replayed entire animation from the first frame to the requested frame.

Design approach 2:

- MVC architecture => Modularization + Decoupling. Graph library contains only rendering info, decoupling rendering details from UI visualization logic
- Overall more manageable codebase. Each tool has a specific task, located in a specific folder.
- Drawbacks:
  - Development time
  - Direct DOM manipulation can hinder performance

Design Description

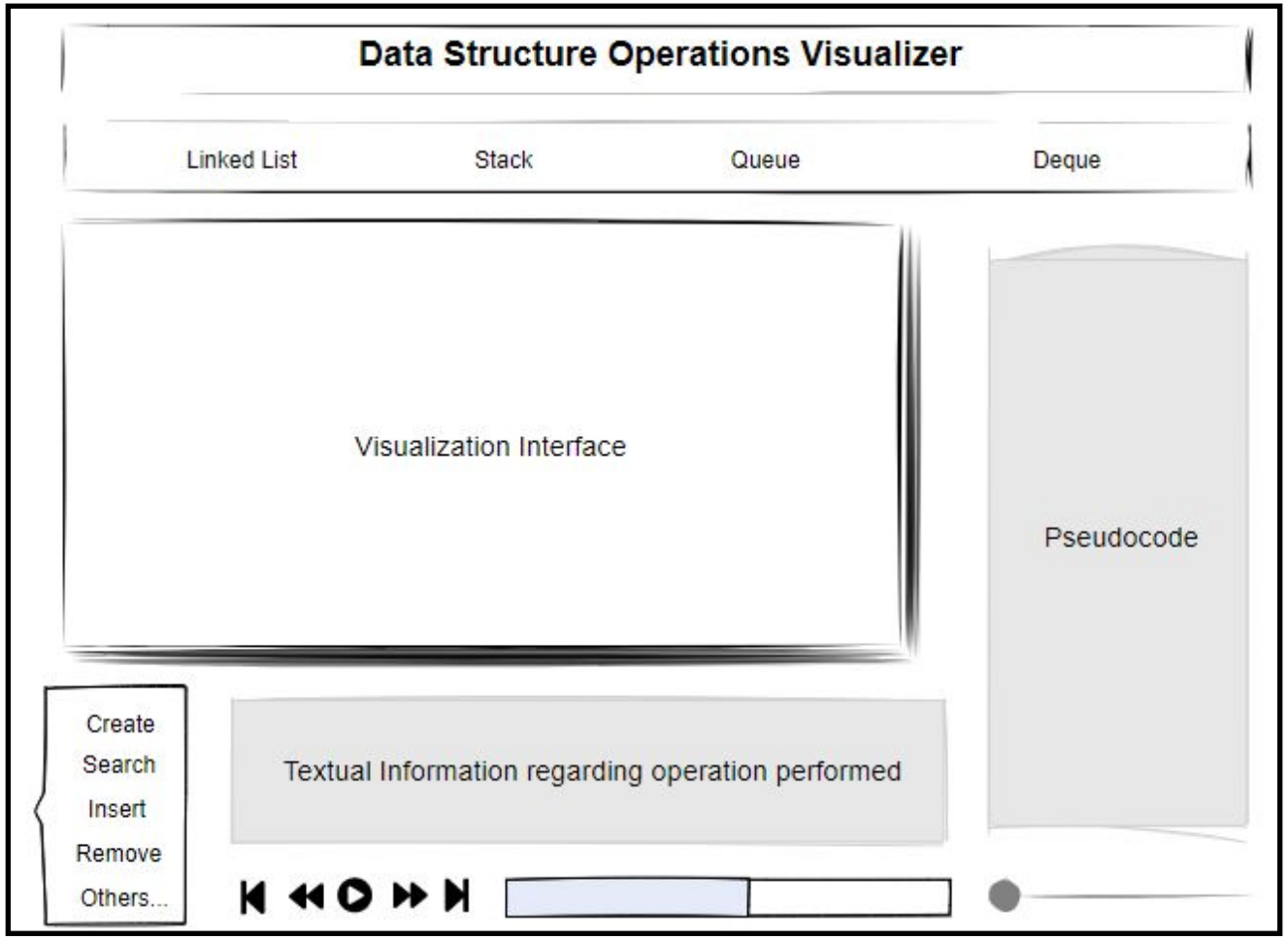
Master class diagram:



Design Description

User interface diagram 1:

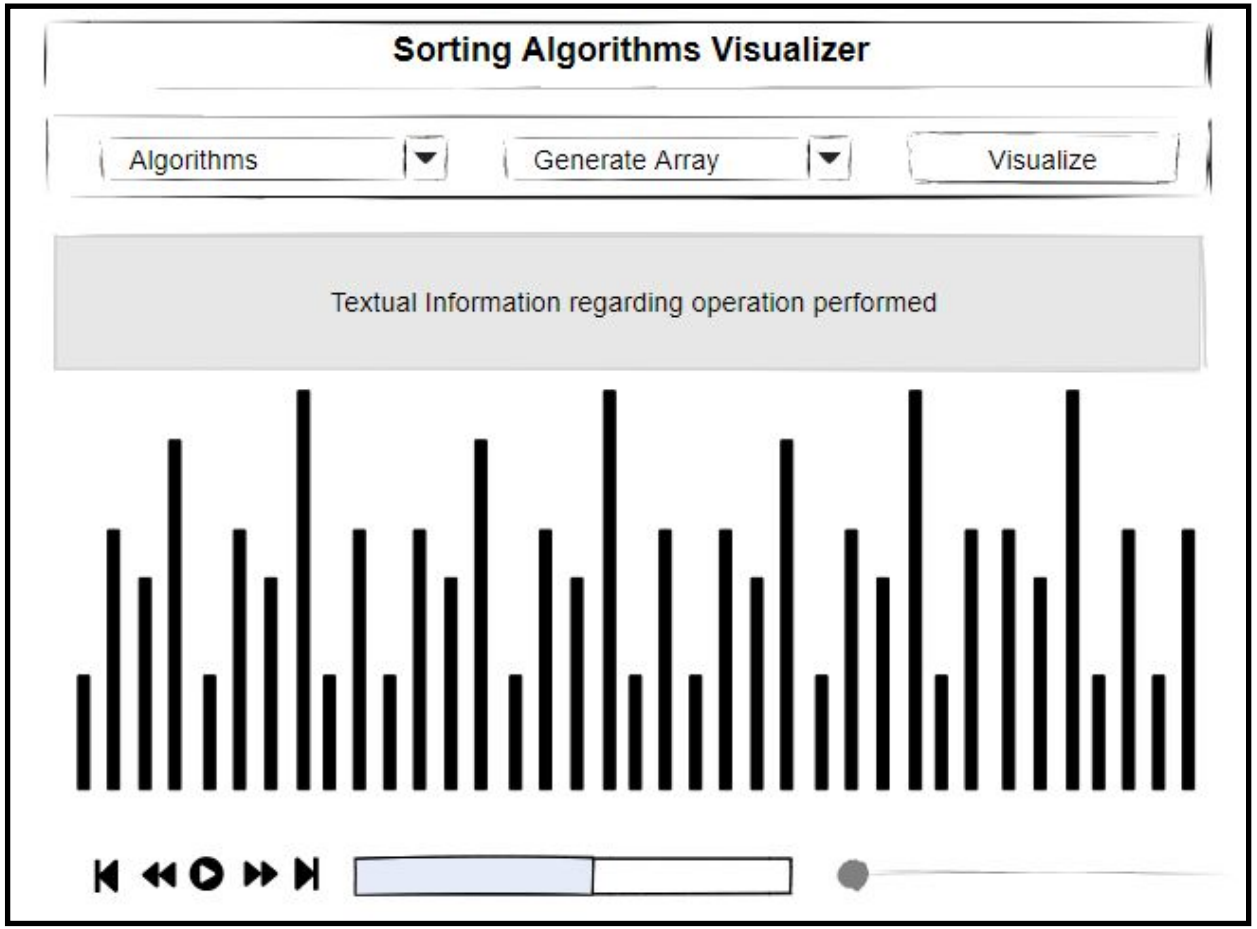
The linear data structure visualizer



Design Description

User interface diagram 2:

The sorting visualizer



Modules and Implementation Details

Features:

- Linear data structures
- Nonlinear data structures
- Sorting visualizer
- Animation Controls

Technology used in all modules: HTML, CSS, JavaScript, jquery, bootstrap

Modules and Implementation Details

- **Linear data structures:**
  - Canvas is drawn using the HTML <canvas> element
  - “list/singleList.html” uses multiple elements with input fields
  - controlBar: The animation speed control slider
  - LinkedList.js (Steps to implement linked lists):
    - initialize the linked list, attributes
    - Set colors, coordinates, head and tail pointers
    - Parse inputs given to the insert function
    - Set background, foreground colors, position and length (max length = 6)
    - insertNode() handles the insertion logic
  - So at each step of the process you parse the input, make changes to the structure using “this”, set a state(color, drawing etc), highlight the step in progress and write algorithm comments (that go in the top right)
  - The same steps apply to stacks and queues

Example: Linked List

```
// Initialize attributes
LinkList.prototype.initAttributes = function () {
  // Logical part
  this.head = null;
  this.tail = null;
  this.length = 0;
  // Graphics part
  this.objectID = 1; // Sequence number of graphics
  this.width = 50; // Rectangular width
  this.height = 50; // Rectangular height
  this.interval = 120; // gap
  this.foregroundColor = "#1E90FF"; // Foreground
  this.backgroundColor = "#B0E0E6"; // Background color
  this.tomato = "#FF6347"; // tomatocolour
  this.palegreen = "#32CD32"; // palegreencolour
  this.startX = 150; // New nodexcoordinate
  this.startY = 100; // New nodeycoordinate
  this.startHeadY = 200; // Head nodeycoordinate
  this.startheadArrowY = 250; // Head pointerycoordinate
  this.starttailArrowY = 310; // Tail pointerycoordinate
  this.arrowLength = 30; // Arrow length
  this.implementAction(this.initHeadNode.bind(this), 0);
};
```

```
LinkList.prototype.insertCallBack = function (seq, value) {
  if (seq != "" && value != "") {
    seq = parseInt(seq);
    value = parseInt(value);
    if (this.head.value <= 5) {
      this.implementAction(this.insertNode.bind(this), [seq, value]);
    } else {
      alert("Max length allowed for LL = 6");
    }
  }
};
```

- initialize attributes and parse input
- Explain implementAction() in next slide

Example: Linked List

```
Algorithm.prototype.implementAction = function (func, val) {
  var retVal = func(val);
  this.animationManager.startNewAnimation(retVal);
};

// Helper method to create a command string from a bunch of arguments
Algorithm.prototype.cmd = function () {
  var command = arguments[0];
  for (i = 1; i < arguments.length; i++) {
    command = command + "<;>" + String(arguments[i]);
  }
  this.commands.push(command);
};
```

```
LinkList.prototype.insertCallBack = function (seq, value) {
  if (seq != "" && value != "") {
    seq = parseInt(seq);
    value = parseInt(value);
    if (this.head.value <= 5) {
      this.implementAction(this.insertNode.bind(this), [seq, value]);
    } else {
      alert("Max length allowed for LL = 6");
    }
  }
};
```

- implementAction takes as parameters a function “func” and an argument “val”, then calls that function using that argument i.e “func(val)”
- functions that are called by implementAction like insertNode need to :
  - create an array of strings that represent commands to give to the animation manager
  - return the array of commands
- **this.cmd** is a helper function that appends commands to the commands list

insertNode(pt1)

```
LinkList.prototype.insertNode = function (valueArr) {
  var pos = valueArr[0];
  var value = valueArr[1];
  var point = this.head;
  if (pos > this.length || pos <= 0) {
    alert(
      "Location error! The location is out of range.\nCurrent range [1-" +
        (this.head.value + 1).toString() + "]"
    );
    // alert('Position error! The position is out of range.\nCurrent range ' + 1 +'
    //to '+this.head.value) ;
  } else {
    var newNode = new ListNode(
      this.objectID,
      value,
      this.startX,
      this.startY,
      null
    );
    this.objectID++;
    this.length++;
    // Draw a new node
    {
      this.cmd("SetState", "Create a new node: " + value);
      this.cmd("Step");
      this.cmd(
        "CreateRectangle",
        newNode.objectID,
        newNode.value,
        this.width,
        this.height,
        "center",
        "center",
        newNode.x,
        newNode.y
      );
      this.cmd("SetForegroundColor", newNode.objectID, this.foregroundColor);
      // node will have red outline if commented out
      this.cmd("SetBackgroundColor", newNode.objectID, this.backgroundColor);
      // node will turn green if commented out
      this.cmd("Step");
    }
```

- Get position and value to be inserted and do a range check
- Create a new node ListNode
- increment length and object ID
- Draw the new node
- Set foreground and background color
- “this.cmd” appends the commands to the commands list that are sent to the animation manager
- **Ex:** Create a rectangle with objectID 5 containing value 10 of width 100 and height 200 to (100,150) would be "CreateRectangle<;>5<;>10<;>100<;>200<;>100<;>150"

insertNode(pt2)

```
for (var i = 0; i < pos - 1; i++) {
  // Highlight
  {
    this.cmd("SetState", "Search" + i);
    this.cmd("Step");
    this.cmd("SetHighlight", point.objectID, true);
    this.cmd("Step");
    this.cmd("SetHighlight", point.objectID, false);
    this.cmd("Step");
  }
  point = point.linked;
}
// Highlight
{
  this.cmd("SetState", "Search" + parseInt(pos - 1));
  this.cmd("Step");
  this.cmd("SetHighlight", point.objectID, true);
  this.cmd("Step");
  this.cmd("SetHighlight", point.objectID, false);
  this.cmd("Step");
}
// If inserted into the tail node
if (point == this.tail) {
  newNode.x = parseInt(point.x + this.interval);
  newNode.y = parseInt(point.y);
  point.linked = newNode;
  this.tail = newNode;
  this.tailArrow.x = newNode.x;
  this.tailArrow.y = this.startheadArrowY;
  // connect
  {
    this.cmd(
      "SetState",
      "This location is the end of the node, insert directly"
    );
    this.cmd("Step");
    this.cmd(
      "Connect",
      point.objectID,
      newNode.objectID,
      this.forgroundColor
    );
    this.cmd("Step");
    this.cmd("Move", newNode.objectID, newNode.x, newNode.y);
    this.cmd("Step");
    this.cmd(
      "Move",
      this.tailArrow.objectID,
      this.tailArrow.x,
      this.tailArrow.y
    );
    this.cmd("Step");
  }
```

- Move until insertion position
- Highlight nodes while moving (point = point.linked)
- Highlight node at which you stopped
- If you stopped at the tail node, point.linked = newNode will connect tail node to newNode
- move the tail arrows to the new last node
- set the state and add the “insert directly” comment
- Highlight the connection of the new node

insertNode(pt3)

```
} else {
  // If it is not a tailpoint
  newNode.x = parseInt(point.x + this.interval);
  newNode.y = parseInt(point.y);
  newNode.linked = point.linked;
  point.linked = newNode;
  // connect
  {
    this.cmd("SetState", "disconnect" + point.value + "Pointer domain");
    this.cmd("Step");
    this.cmd("Disconnect", point.objectID, newNode.linked.objectID);
    this.cmd("Step");
    this.cmd(
      "SetState",
      "Set the insert node " +
        value +
        "Pointer points to subsequent nodes " +
        newNode.linked.value
    );
    this.cmd("Step");
    this.cmd(
      "Connect",
      newNode.objectID,
      newNode.linked.objectID,
      this.foregroundColor
    );
    this.cmd("Step");
    this.cmd(
      "SetState",
      "set up " +
        point.value +
        "Pointer pointing to the insertion node: " +
        value
    );
    this.cmd("Step");
    this.cmd(
      "Connect",
      point.objectID,
      newNode.objectID,
      this.foregroundColor
    );
    this.cmd("Step");
    this.shiftBack(newNode.linked);
    this.cmd("Move", newNode.objectID, newNode.x, newNode.y);
    this.cmd("Step");
  }
}
```

- If insertion point is not at tail node
- newNode.linked = point.linked (newNode points to insertion position's next node)
- point.linked will be linked to newNode
- A -> C
  B is to be inserted between A and C
  B->C
  A disconnects from C
  B points to subsequent nodes
  A connects to B ("pointer pointing to insertion node)
  A -> B -> C
- Set new objectID and foreground color after insertion

Modules and Implementation Details

- **Nonlinear data structures:**
  - The nonlinear structures have also been implemented the same way as the linear ones.
  - Creation of a canvas, parsing of input, initialization of the structure, the attributes and the colors
  - Data structure-specific functions handle the core logic each of which contains steps that involve highlighting the necessary item on the canvas and displaying comments on the comment box visible on the page which indicates what's happening.

- **Sorting visualizer:**
  - The implementation for the sorting algorithms visualizer is pretty much the same as the implementation of the previously mentioned two visualizers.
  - Creation of the HTML canvas, parsing of input, structure initialization, and other relevant attributes.
  - Additionally this uses d3.js which is an animation library for producing dynamic, interactive data visualizations to ease the integration of code tracing into the sorting visualizer's system.

Modules and Implementation Details

- **Elaborating on the code tracing and how it is implemented within the sorting visualizer:**
  - We have got each of the sorting algorithm's lines from the pseudocode stored within an array of strings, these strings are mapped with their subsequent line numbers (it is to take note that these line numbers can be an array of numbers and not just numbers).
  - This array is populated as and when the concerned sorting algorithm component is initialized.
  - The same array of strings also has a mapping to another array of strings which essentially hold the messages that are displayed from within the helper box.
  - Going further in a very basic sense we have got the subsequent line numbers for the concerned algorithm be passed into the highlightLine() method.
  - The next slide elaborates on this highlightLine() function.

Modules and Implementation Details

```
function highlightLine(lineNumbers) {
    $('#codetrace p').css('background-color',
colourTheThird).css('color', codetraceColor);
    if (lineNumbers instanceof Array) {
        for (var i = 0; i < lineNumbers.length; i++)
            if (lineNumbers[i] != 0)
                $('#code' + lineNumbers[i]).css('background-color',
'black').css('color', 'white');
    } else
        $('#code' + lineNumbers).css('background-color',
'black').css('color', 'white');
```

- The above is the method responsible for highlighting the concerned lines that are mapped internally by the sorting visualizer.
- It is to note that each of the lines have an DOM id equivalent to “#code<lineNumber>”.

Modules and Implementation Details

- **String operations:**
  - The miscellaneous string operations have also been implemented the same way as the linear/non-linear data structure operations.
  - Features:
    - Pattern matching
      - Simple Matching
      - KMP Matching
    - Find
      - Binary Search
      - Linear Search
  - Creation of a canvas, parsing of input, initialization of the structure, the attributes and the colors.
  - Data structure-specific functions handle the core logic each of which contains steps that involve highlighting the necessary item on the canvas and displaying comments on the comment box visible on the page which indicates what's happening.

Project Demonstration and walkthrough

Demo.

Tests conducted

Feature	Tests conducted	Timeline
Linked Lists, Stacks and Queues	Unit tests confirming Input sanity, range checks, checks confirming if insert, delete, create, push, pop, enqueue and dequeue were performed	Apr 2021
BST, Heap, AVL Tree	Unit tests confirming input sanity, range checks, checks confirming if insert, find, delete work	Oct 2021
Graph algorithms	Unit tests for generation of graph, addition/deletion of edges, algorithm logic, placement and alignment of nodes, show/hide weights and directed edges	Oct 2021
Pattern Match and Search	Unit tests for simple and KMP match algorithm checks, input sanity	1st Nov 2021 - 7th Nov 2021
Sorting visualizer	All algorithm logic, input sanity, range, code<->execution mapping, correct highlights, animation frames	Late Nov 2021 - Dec 10th 2021

Tests conducted

- Benefits of unit testing each module before each review is that each data structure performed the way it should despite invalid/unexpected inputs
- Ensured that the addition of the code tracer to the existing project didn’t affect existing modules (Regression test)
- Non-functional tests:
  - All animations, visualizations and events on the website were unit tested. They perform well and can be tweaked to further suit users’ needs (good performance)
  - Returning to prev. frame reqd. that the animation be played from the first frame to the requested frame => slow animation. Now, the visualizations can jump b/w frames in O(1) time since all that needs to be done is to load the state object for that frame. This functionality was also tested.
- Test environment: Tested on Chrome v90 on Windows 10 and Ubuntu 20.04
- Role played by each team member was to unit test the component they were working on at the time.

Results and discussion:

Initial requirements:

- Visualize popular linear, nonlinear data structures, graph algorithms and sorting algorithms in one web-based interface.
- Provide an animated, bug-free experience which is highly interactive.

Have we met the requirements?

- To a large extent, yes. We even scrapped the sorting visualizer v1 (Phase-2 Review 1) and the grid-based graph visualization (Phase-2 Review 2) because v1 of the sorting visualizer was a separate app and we got rid of the graph viz. because even though it was fun to use, it didn’t help users understand how each step of the algo works.

Schedule

Task/Module	Planned timeline	Actual timeline	Reason for deviation (if any)
Linear data structures	Mar 2021 - May 2021	Mar 2021 - May 2021	
Trees	Aug 2021 - Oct 29th 2021	Aug 2021 - Oct 29th 2021	
Graph algorithms and misc.	Aug 2021 - Sept 2021	Sept 2021 - Oct 29th 2021	Based on feedback from our guide and using the module, we felt it could be more effective if we changed how graph visuals worked.
Sorting visualizer	Oct 1st - Oct 29th 2021	Nov 2021 - Dec 2021	Addition of code tracing

Documentation

- Project report
- IEEE document
- Video
- Github repository
- Poster
- Files will be uploaded to the repository post review.

Lessons Learnt

- Learnt that it isn’t easy to create an actionable 1-year plan:
  - time taken to pick up new technologies and be effective with them
  - estimation of time taken to develop software is hard because of bugs, integration issues
- Learnt the importance of writing unit, integration and system tests
- Validating ideas/modules for a project on a timely basis is important to assess success/failure and project delivery status.

Conclusion and Future work

- Conclusion:
  - Successfully implemented linear, non linear data structures, graph algorithms and a sorting visualizer that supports a robust code tracing system on a bug-free, web-based user interface that is easy-to-use, user-operable and highly interactive to support the learning of fundamental concepts in Computer Science that hopefully aid in problem-solving behaviour.

- Future work:
  - Refactor to reduce redundant code and make it easier to understand and debug
  - Add more data structures
  - Implement code tracing for all data structures
  - Incorporate secure coding practices

References

- Tao Chen and T. Sobh, "A tool for data structure visualization and user-defined algorithm animation," 31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No.01CH37193), Reno, NV, USA, 2001, pp. TID-2, doi: 10.1109/FIE.2001.963845.
- Osman, Waleed & Elmusharaf, Mudawi. (2014). Effectiveness of Combining Algorithm and Program Animation: A Case Study with Data Structure Course. Issues in Informing Science and Information Technology. 11. 155-168. 10.28945/1986.
- Ryan S. Baker, Michael Boilen, Michael T. Goodrich, Roberto Tamassia, and B. Aaron Stibel. 1999. Testers and visualizers for teaching data structures. SIGCSE Bull. 31, 1 (March 1999), 261–265. DOI:https://doi.org/10.1145/384266.299779
- Teaching Support for the Visualization of Selected Recursive Algorithms Baraník, Róbert and Steingartner, William. ipsitransactions.org/journals/papers/tar/2021jan/p3.pdf
- https://google.github.io/styleguide/jsguide.html
- https://material.io/design/guidelines-overview

Thank You