# Visualization of Memory Reuse in Immutable Stacks and Lists

Evan Olds

*Department of Computer Science*

*University of California, Santa Cruz*

***Abstract* - I propose a visualization tool designed specifically for two immutable data structures: an immutable stack using a singly-linked list and an immutable list using a binary tree. The visualization tool produces a rendering of nodes used by the data structures, along with color-coding to show memory reuse in the form of reference counts. After creating several stacks or lists from the same original source, many nodes have multiple references from different data structures. Higher reference counts are colored differently than lower reference counts, providing a quick visual illustration of the memory efficiency of the immutable data structures.**

***Index Terms - Data visualization, immutable, stack, list***

## I. Motivation

Tools to visualize in-memory data structures are invaluable for debugging and teaching. In practice, efficient, immutable data structures are useful when writing robust code that is easy to maintain. However, teaching students the concepts behind efficient immutable data structure implementations is difficult without visualization tools. A need exists for visualization tools that can be utilized by instructors, when presenting to the class, and students, when learning.

## II. Related Work

Data structure visualization is not new. Many methods exist to render common data structures. Also, a single data structure may have many visualization methods. For example, Herman et. al. surveyed a variety of graph visualization methods in their paper *Graph Visualization and Navigation in Information Visualization: A Survey*. The following is non-exhaustive list of surveyed layout structures or views**[1]**:

- Tree layout
- H-tree layout
- Ballon view
- Radial view (2D)
- Radial view (3D)
- Tree map
- Information cube
- Hierarchical cluster

The University of San Francisco has visualizations for a variety of different data structures and algorithms**[2]**. Binary search trees and linked-list-based stacks are supported, but both are mutable. Furthermore, no binary tree for a list is supported. In addition, no concept of reference count coloring is provided. While the tool is useful for helping students understand data structures and operations, immutable data structures are not an area of focus.

From a rendered-structure standpoint, a rather standard tree view similar to that of existing tree visualizations is expected to suffice for this project. However, a preliminary search for immutable-structure-specific visualizations did not yield any significant results. Similarly, reference-counting-specific visualization methods did not yield any significant results.

This project's work intends to be novel in the area of specifically visualizing reference counts of nodes in immutable data structures.

## III. ETO_ImmStack and ETO_ImmList

Two JavaScript classes, ETO_ImmStack and ETO_ImmList, were written to implement an immutable stack and an immutable list, respectively. Custom implementations were chosen, as opposed to finding existing implementations in open-source code, so that all internal data would be accessible and aspects of internal data structure can be controlled. Also, the ETO_ImmList implementation uses a binary tree for simplicity, while other, real-world implementations may be more likely to use a more complex structure such as an RRB tree**[3]**.

ETO_ImmStack uses a singly-linked list internally. Each node's `next` member references the node below it, or null if the node is at the bottom of the stack. Each node is frozen using JavaScript's Object.freeze method, so as to prevent alterations. ETO_ImmStack exposes the stack's top node through the enumerable `topNode` member.

ETO_ImmList uses a binary tree internally. The lists contents are stored in order in the tree's leaves, all of which are on the same level. Each node is frozen using JavaScript's Object.freeze method, so as to prevent alterations. ETO_ImmList exposes the tree's root node through the enumerable `root` member.

| Class | Member | Description | Worst-case time |
|---|---|---|---|
| ETO_ImmStack | length | Gets the number of items in the stack. | O(1) |
| ETO_ImmStack | pop() | Creates and returns new stack with the item popped off the top. | O(1) |
| ETO_ImmStack | push(item) | Creates and returns new stack with the item pushed on the top. | O(1) |
| ETO_ImmList | add(item) | Creates and returns new list with the item added. | O(log n) |
| ETO_ImmList | length | Gets the number of items in the list. | O(1) |
| ETO_ImmList | removeLast() | Creates and returns new list with the last item removed. | O(log n) |
| ETO_ImmList | replaceAt(index, newItem) | Creates and returns new list with the item at the specified index replaced by the new item. | O(log n) |

**Table 1: ETO_ImmStack and ETO_ImmList members and worst-case time complexities**

The ETO_ImmStack implementation matches the time and space complexities of a mutable stack for each implemented operation. The ETO_ImmList implementation is slower than a mutable list for additions at the end of the list, removals from the end of the list, and indexed read/replace access. However, the mentioned operations all stay within O(log N) time complexity, which is considered practical for real-world use. Table 1 contains a summary of implemented operations and associated worst-case time complexities.

## IV. User Interface

The software's UI contains a code editor where the user can type a snippet of JavaScript code. The open-source Ace code editor is used (https://ace.c9.io/).

The user's code is expected to construct instances of either ETO_ImmStack or ETO_ImmList objects. Multiple instances of either can be constructed, but mixing instances of stacks and lists is currently not supported.

After entering code, the user can click the button below the code editor to refresh the visualization of the stack(s) or list(s). Should the code contain errors, the errors will display in a message box below the refresh button. If the code does not contain errors, the data structures are rendered in a canvas below the message box. Figure 1 shows a sample of the user interface with a single stack being rendered.

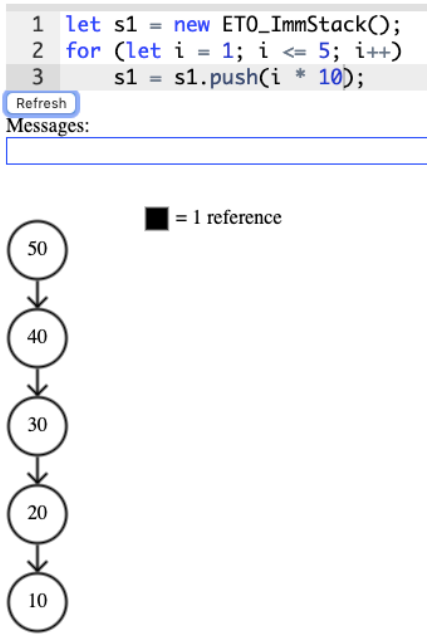


**Figure 1: User interface with a simple stack rendered**

## V. Parsing the User's Code

When the refresh button is clicked, the code string entered by the user is analyzed. Regular expressions are used to recognize lines of code that instantiate an ETO_ImmStack or ETO_ImmList. The declare variable name strings for each are stored in a set named `nameSet`.

A second regular expression is then built to recognize variable declarations that are assigned the result of one of the objects from `nameSet`. If found, the new variable names are added to `nameSet`. The process repeats until no new variable names are found.

A string of JavaScript code is generated from the set of variable names. The code inserts each variable name into an array named "structures". This string is concatenated to the end of the user's JavaScript code, then a JavaScript function is built using the code as the body, and a single argument named "structures". Upon successful construction of the JavaScript function, it is invoked with an empty array. Upon completion, the array will contain each data structure that needs to be rendered, along with a corresponding variable name for that structure.

## VI. Connectivity Groups: Stacks

A single stack may not suffice to properly visualize content created by the user's code. All immutable stack operations are implemented to return a new stack that shares nodes from the previous stack, so new stacks generally branch off an existing stack, as shown in figure 2. But the user may elect to create multiple new stacks that do not all stem from the same original source, and therefore do not share any nodes. In this case, a single stack with branches would not suffice as an accurate rendering.

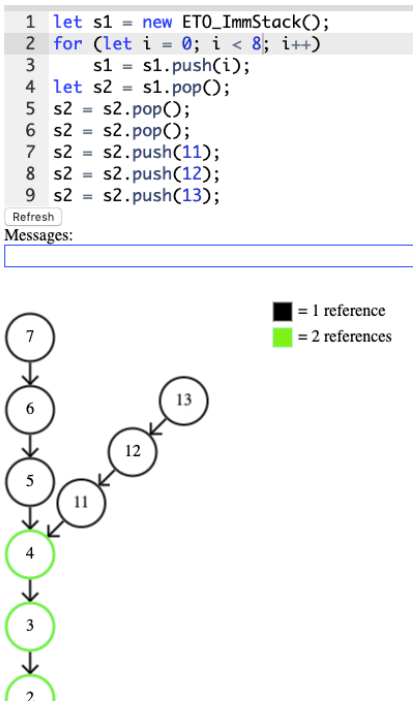


**Figure 2: Stack s2 branches off of s2 above node 4**

For example, the following code produces two stacks with identical content, but no nodes are shared between them.

```
let s1 = new ETO_ImmStack();
let s2 = new ETO_ImmStack();
for (let i = 0; i < 5; i++) {
    s1 = s1.push(i);
    s2 = s2.push(i);
}
```

Rendering disjoint stacks as separate is crucial to properly convey when stacks do or do not share nodes. Stacks must be grouped based on connectivity. Two stacks are connected if they have 1 or more nodes in common.

Reference counts alone do not determine stack groups. For example, among 4 stack objects a particular node could be shared by only 2 of the 4. Such a node would have a reference count of 2, but the 4 stacks could potentially be connected or disjoint.

Since nodes are immutable, any shared node eventually leads to (or is) a shared node at the bottom of the stack. Therefore, stacks can be grouped by a simple criteria: Two stacks are in the same group if and only if they share a bottom node.

## VII. Connectivity Groups: List Trees

Much like the stacks, a single binary tree may not suffice to properly visualize content created by the user's code. Any two trees sharing one or more nodes should be rendered as a connected tree. Any two trees not sharing any nodes should be rendered as separate trees.

Determination of tree groups is more complex than stack groups. Stack groups are determined based on a single node at the bottom of the stack. Tree groups are determined by shared leaf nodes.

The NamedListGroup class implements a static build method, that returns an array of NamedListGroup objects. Each object represents a collection of lists whose trees are connected by 1 or more shared nodes. The build algorithm does the following:

1. Build and populate JavaScript Map object, `nodeMap`, that maps a node to a descriptive structure containing a set of trees that contain the node.
2. Create a collection of Set objects, initially 1 per list. Store them in a Map object, `listGroupMap`, that maps a list to the set containing all lists with connected trees.
3. Iterate through nodes in `nodeMap`. If a node's list set contains more than 1 referencing list, merge the sets associated with each list together into 1 set, then update `listGroupMap` for each list in the merged set.
4. Find the distinct sets in `listGroupMap`, build a NamedListGroup for each one, and return the groups in an array.

## VIII. Node Positioning: Stacks

When rendering a collection of stacks, node positions in the render view must be computed. Since stacks are typically rendered vertically, the tallest stack amongst the collection, denoted as T, is rendered vertically first, then shorter stacks branch off at appropriate points.

A JavaScript Map object is used to map a node to a descriptive structure that includes an (x, y) position. Nodes in stack T are iterated through first. The top node is arbitrarily placed at (0, 0). Each next node is placed below the previous, such that the distance between the nodes' centers is a bit more than a node's diameter.

After T's nodes are positioned vertically, each additional stack is analyzed. For some stack S, node N is found such that N is the first/highest node from S that is also in T. Each node

below N has already been positioned, so only nodes above N need positions computed.

An arbitrary set of branching directional vectors is used: (1, 1), (1, 0),(1, -1), (-1, 1), (-1, 0), and (-1, -1). A node at a branching point will head off in one of these directions. More than 6 branches from a single node is currently not supported.

## IX. Color Schemes

Two different color schemes were tested during development. An initial assumption was that a gradient scheme would suffice if the starting and ending colors were sufficiently different. Green and red were chosen as the two colors. The scheme chooses red for nodes with one reference and red for nodes with the maximum number of references. Colors for reference counts in between 1 and the max were computed based on linearly interpolating. The figure below shows the results on a sample tree.

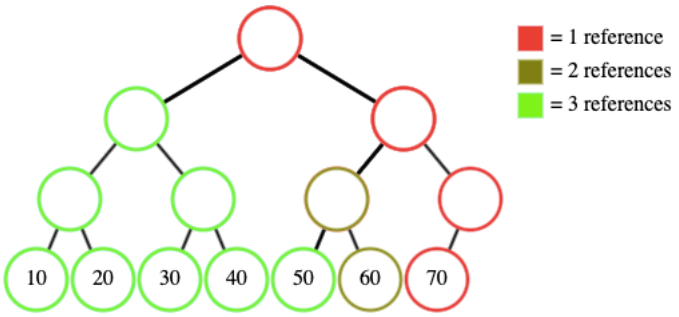


**Figure 3: Green/red color scheme used on a list's binary tree**

Although discernible, the color scheme didn't seem to provide as clear of a picture as desired. The color used for 2 references doesn't stand out from the other colors, especially if the display's color quality is poor. Therefore, a new color scheme was developed and tested in similar situations.

Choosing colors arbitrarily, such that colors have high visual contrast between each other, produced better results. The same tree is shown with the arbitrary color scheme in figure 4. Note that the two green nodes with a reference count of 2 have a more pronounced difference from the blue and black nodes.

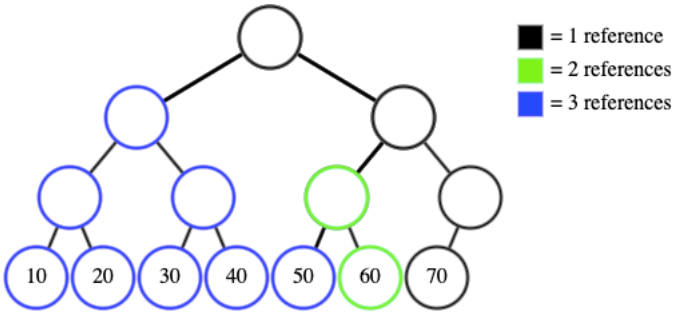


**Figure 4: Arbitrary color scheme used on a list's binary tree**

The arbitrary color scheme uses the following colors in this order:

1. Black
2. Green
3. Blue
4. Orange
5. Magenta
6. Yellow
7. Red
8. Gray

The colors may be reordered and/or extended in a future version, but the color scheme seemed to suffice for most situations.

## X. Rendering Connected Groups

Each connected stack group is rendered as 1 unit. Each distinct group simply gets rendered to the right of the previous group, as shown below in figure 6.

```
1  let s1 = new ETO_ImmStack();
2  s1 = s1.push(10);
3  s1 = s1.push(20);
4  let s2 = s1.push(30);
5  let s3 = new ETO_ImmStack();
6  for (let i = 1; i <= 4; i++)
7      s3 = s3.push(i * 10 + i);
```

**Figure 5: Code that produces 2 stack groups**

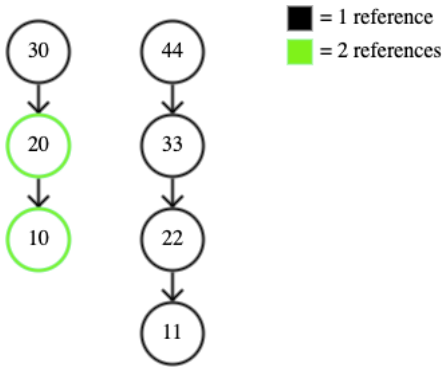


**Figure 6: Rendering of the 2 stack groups made by figure 5's code**

Each connected tree group is currently rendered as 1 tree. Only nodes from an arbitrarily chosen tree are shown, but reference count coloring is applied to represent all trees in the group. Future work may change this to render all nodes from

all trees, although this would introduce complexities with node spacing/locations.

## XI. Root Node External Labels

In order to allow the viewer to match a JavaScript variable with a corresponding rendered list, external node labels are added to each root node in the rendered tree view. For example, figure 7 shows code that creates two lists. `list1` is a list containing the integers 1 through 8. `list2` is built by adding the integer 100 to `list1`. Since `list1`'s tree was perfect, `list2`'s tree has a height one greater. This means that `list1`'s root is `list2`'s root's left child.

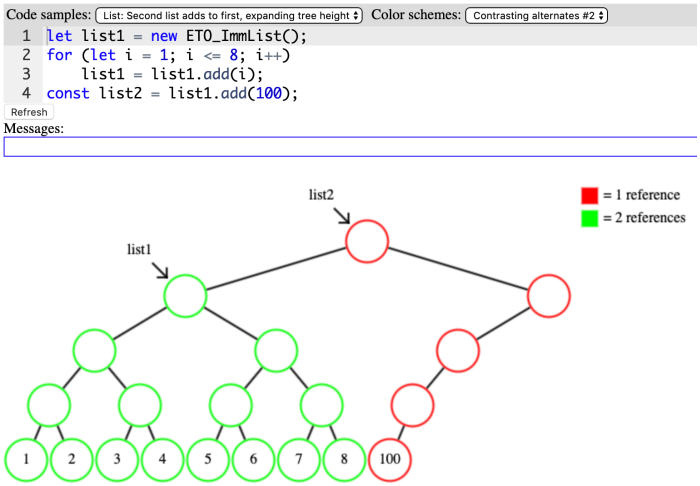


**Figure 7: Root node labels for 2 connected trees**

Positioning of root labels and arrows is arbitrarily chosen as up and to the left. This generally suffices, since roots are often along the leftmost side of the tree. However, it is acknowledged that should certain operations be added to the immutable list, such as sublist or concatenation, then roots may appear more towards the center of the tree. The labeling position would not suffice in such cases, as the label and/or arrow may overlap another node.

A problem occurs when two list's root nodes exist at the same position. In fact, a plethora of problems beyond just root node labeling arise because of this. Solutions to other aspects of this problem are discussed in later sections. For now, just the labeling problem is addressed.

If two root nodes occupy the same position, one option is to have cascade positions, as will be discussed more in the next section. In this case, adding cascaded labels generally results in overlap and is visually unpleasant, as shown in figure 8.

This problem was addressed by producing a single label for all root nodes that cascade from a shared position. A comma-separated list of all JavaScript variable names corresponding to the root is used as a single label, as shown in figure 9.

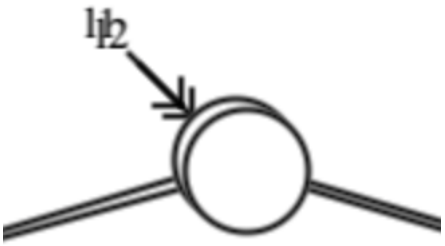


**Figure 8: Overlapping labels for l1 and l2's root nodes**

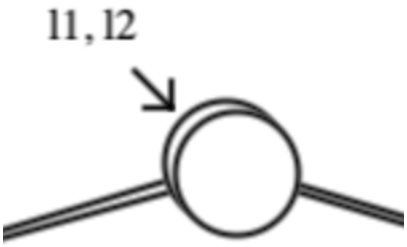


**Figure 9: Comma-separated node label**

The algorithm to determine how to group labels for a cascaded node group is as follows:

1. Create Map object `rootToNamedList`, which maps a list tree's root node to the named structure object.
2. Initialize `handledShared`, a Set object that tracks sets of shared nodes that already have had a label produce for them.
3. For each named list that's being rendered:
    a. Let `desc` = the descriptor for the named list's root node, which includes a list of all nodes that shared the same position.
    b. If `handledShared` contains desc's list of nodes sharing the position, skip this list.
    c. Else iterate through the lists whose roots share the position, producing a comma-separated string of all list names.
    d. Add the list of nodes sharing the position to `handledShared` to prevent the label from being built/rendered more than once.

## XII. Overlapped, Non-Shared Node Positioning

When two distinct lists have the same height, their root nodes are positioned at the same location. However, the 2 nodes are NOT shared, since distinct each list has a distinct

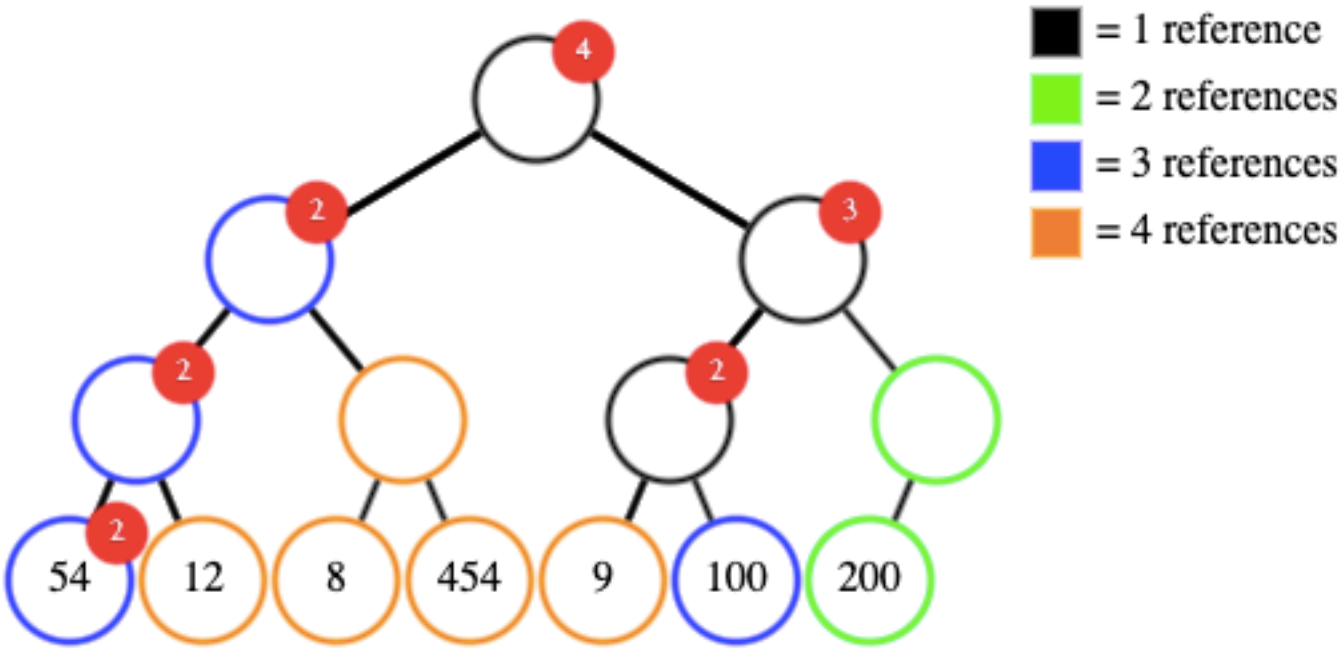


**Figure 10: Overlapping node counts rendered in the upper-right of nodes. Trees are produced by figure 13's code.**

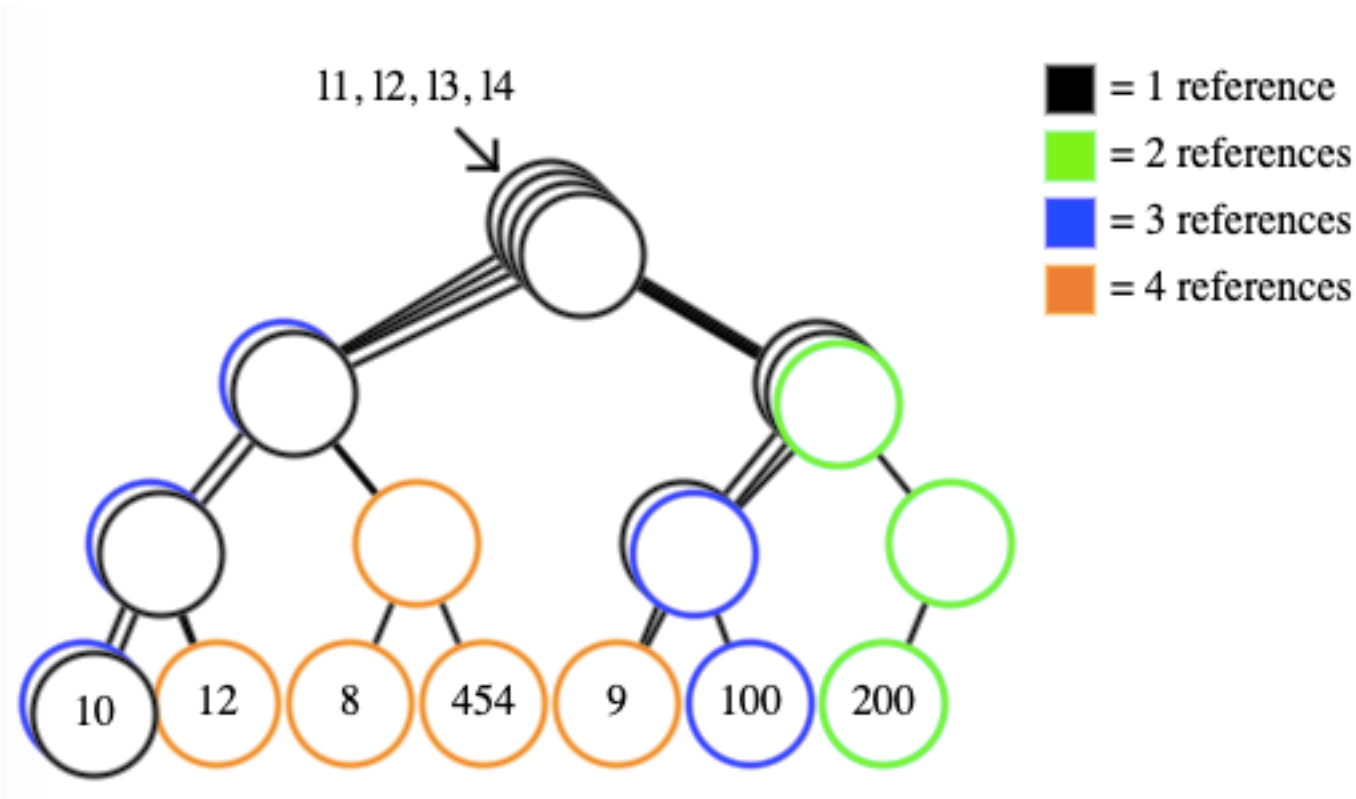


**Figure 11: Cascaded positioning for non-shared nodes at the same position. Trees are produced by figure 13's code.**

Code samples: (default) Color schemes: Contrasting alternates #1

```
1  let list1 = new ETO_ImmList();
2  for (let i = 1; i <= 13; i++)
3      list1 = list1.add(i);
4  const list2 = list1.add(100);
```

Refresh

Messages:

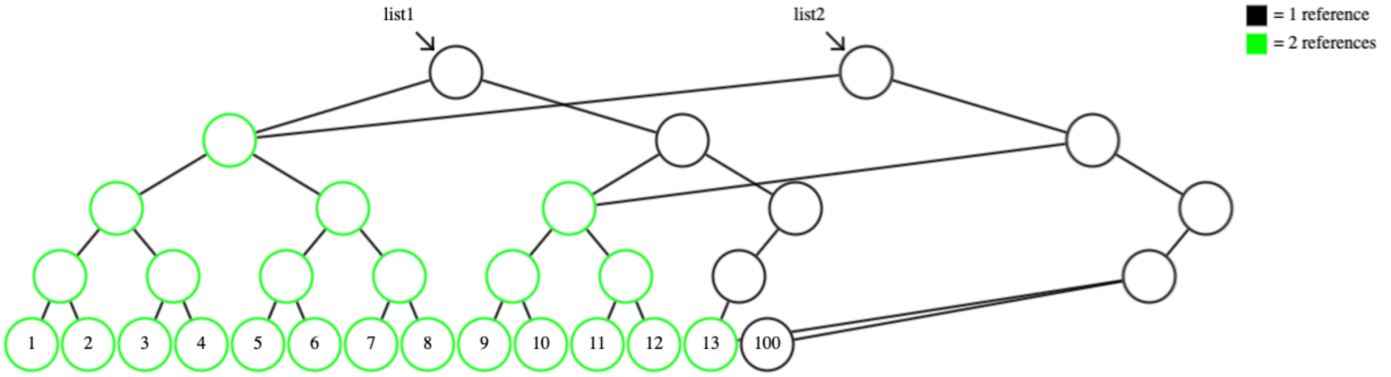


**Figure 12: Right shifting for overlapping, non-shared nodes from two list's trees.**

tree root node. A few methods were tried to deal with the issue of 2 or more distinct nodes occupying the same position.

The code in figure 13 creates 4 lists and accompanying trees. Six instances of overlapping, non-shared nodes exist in the produced trees.

```
let l1 = new ETO_ImmList([54, 12, 8, 454, 9]);
let l2 = l1.add(100);
let l3 = l2.add(200);
let l4 = l3.replaceAt(0, 10);
```

**Figure 13: Code that creates 6 overlapping, non-shared nodes**

The first method implemented for visualizing overlapping, non-shared nodes was a small circle containing the number of overlapping nodes at a particular node. Figure 10, on the previous page, shows this method used with trees produced by the code from figure 13.

It was quickly decided the that the circular numerical labels were not a good indication of what was going on. This lead to the creation of method 2, which involved shifting / repositioning nodes that overlapping.

The "cascading" method method is shown in figure 11, also on the previous page. This method more clearly indicates when multiple distinct nodes exist at the same position. Each node is shifted a small amount up and to the right, so the exact number of nodes at that particular location can be determined.

The cascading method uses a small shift, but a larger shift to the right was also implemented. The goal was to try to visually separate out parts that were specific to new lists after the first. The results are shown in figure 12. While the results work to visually distinguish 2 trees, 3 or more trees become visually confusing quickly, as shown in figure 14.

## XIII. Analysis of Results

The rendered structures produced a clear picture of memory reuse for both the immutable stacks and the immutable lists. The remaining and future work sections that follow mentions aspects of the results that need improvement.

## XIV. Remaining Work

The top nodes of stacks currently only contain their data values. An external label for top nodes should be implemented, similar to the labels for tree roots.

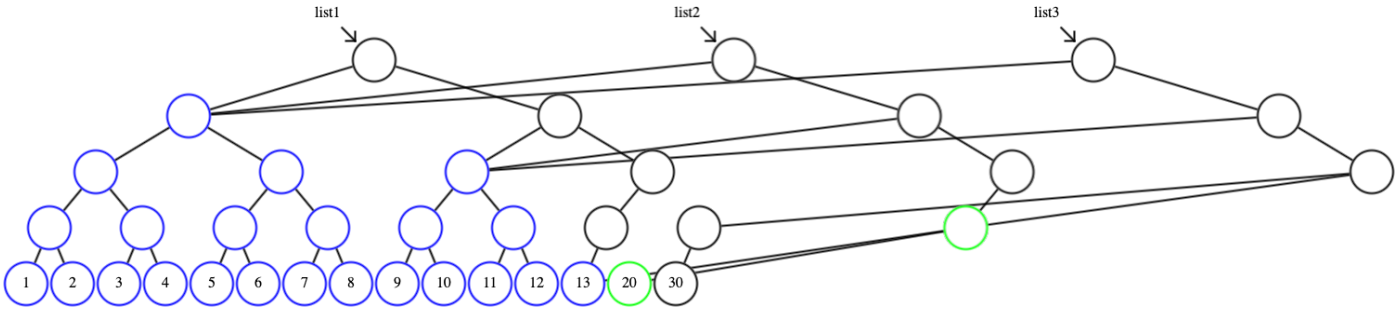


**Figure 14: Multiple shifted lists**

Although list tree grouping is implemented, rendering separate tree groups is not yet implemented. Rendering tree groups side by side should suffice. This was not implemented simply due to time constraints and may be added if the project continues after this quarter.

## XV. Future Work

This project could be extended in many ways. The first, and perhaps most obvious, is that additional immutable data structures can be rendered. The following data structures are of interest:

- Immutable binary search trees
- Immutable k-ary heaps
- Immutable binomial heaps
- Immutable queues
- Immutable dequeues
- Immutable sets
- Immutable maps

Some of the above immutable objects may be implemented with another immutable object as the underlying storage. For example, if an immutable binary search tree is implemented, this could potentially be used as the storage for an immutable set or map. So perhaps not all of the above need to be implemented with custom immutable data structures, but it is desired to have efficient implementations of each. Ideally, operations with O(1) time complexity in the mutable version should have no worse than O(log n) time complexity in the immutable version.

A second way to extend the project would be to visualize additional aspects beside position and reference count of nodes. Perhaps the ability to single out a specific list's tree from a tree with shared nodes by using color coding would be helpful to the viewer.

Interactivity is another area for extension. Any of the following could be implemented:

- Add the ability for the user to choose which of multiple trees is the frontmost, since the cascading scheme still has only 1 node's data value visible in a cascaded collection.
- Add the ability for the user to hover over nodes and get a list of all lists or stacks containing that node.
- Add visibility toggles for all rendered structures, so that desired structures can be hidden/shown.

Animation is yet another potential area for extension. Showing how a tree or stack is built by executing the code line-by-line may help the viewer understand which steps in code correspond to the creation of various constructs. Also, animating between views, such as between the cascading vs. shifted views for trees, may help with understanding of what's being shown.

One additional note pertains to the importance of generalizing the code if the project continues. In theory, generic positioning and rendering logic could be produced for a k-ary tree, allowing list binary trees, BSTs, AVL trees, red-black trees, k-ary heaps, and any other k-ary tree to be positioned/rendered using the same code. Abstracting out the positioning and rendering logic as separate, reusable pieces may be worth the effort to increase the flexibility of the tool.

If made more flexible, the tool could potentially even become an integrated browser debugging / visualization tool. Such an alteration would require a significant amount of work, but may be well worth the effort if this project continues.

## References


1. I. Herman, G. Melancon, and M. S. Marshall, "Graph Visualization and Navigation in Information Visualization: A Survey", *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 1, pp. 24-43, January-March 2000.
2. Galles, D. (2019). Data Structure Visualization. [online] cs.usfca.edu. Available at: https://www.cs.usfca.edu/~galles/visualization/Algorithms.html [Accessed 11 Mar. 2019].
3. Bagwell, Philip, and Rompf, Tiark. *RRB-Trees: Efficient Immutable Vectors*, N.p., 2011. Print.