Analysis of Binary, AVL, and Splay Trees

by Isaac Lee

Abstract:
An analysis of the properties of binary search trees, AVL trees, and splay trees using a chosen data set. The data set contains the information on 16,599 different video games, including their names, release dates, sales per region, and sales worldwide. The data is then shuffled and inserted into trees using varying indices.

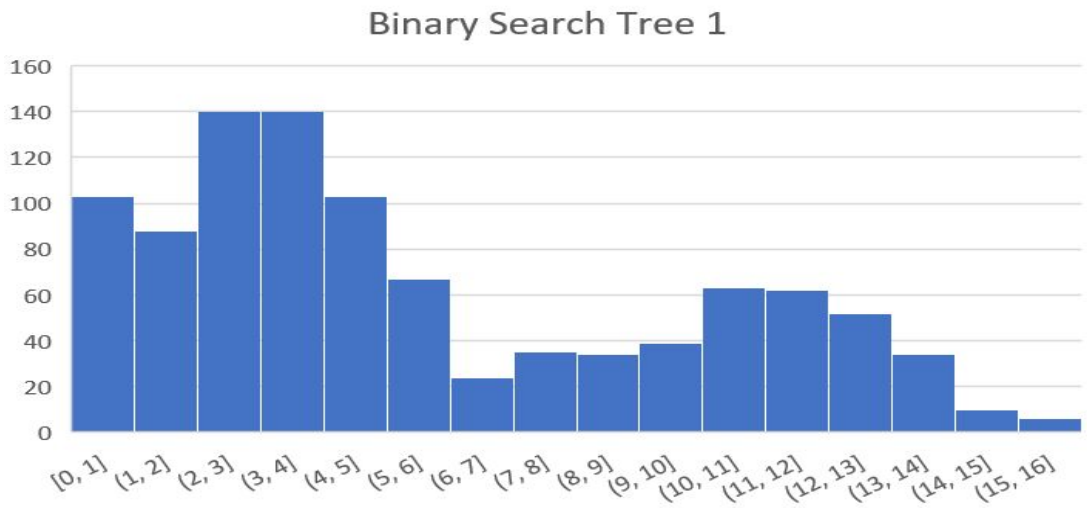
Data Set Attributes:

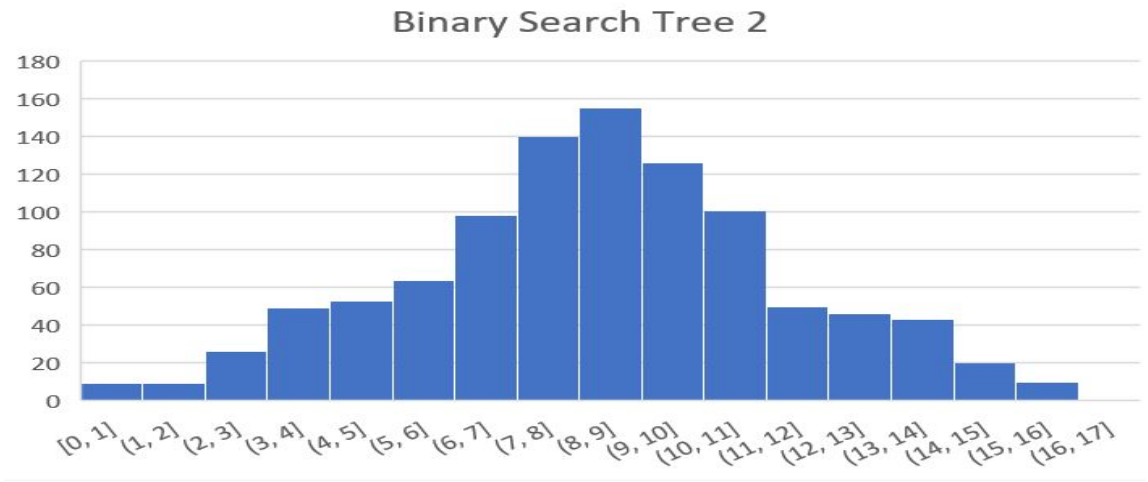
1. Rank - The rank of the video game based on sales number (Rank 1 ==> most sales)
2. Name - The name of the video game
3. Platform - The platform/console the video game was released on
4. Year - The year of the video game's release
5. Genre - The genre of the video game (Platformer, Simulation, Racing, Fighting, etc.)
6. Publisher - The publisher of the video game (Nintendo, Sony, Microsoft, etc.)
7. NA Sales - The total number of sales in North America
8. EU Sales - The total number of sales in Europe
9. JP Sales - The total number of sales in Japan
10. Other Sales - The total number of sales in all other regions
11. Global Sales - The total number of sales worldwide

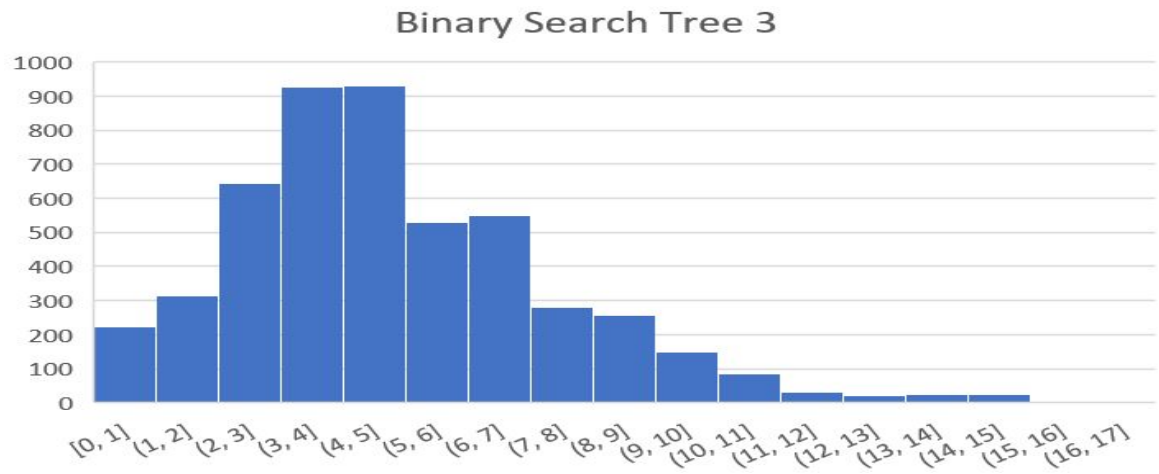
Data Source:
This data was downloaded from https://www.kaggle.com/gregorut/videogamesales. I chose this data because I enjoy video games far more than I should, to the extent where I now run a gaming club and am considering a career in game development. This just seemed like it made sense to use.

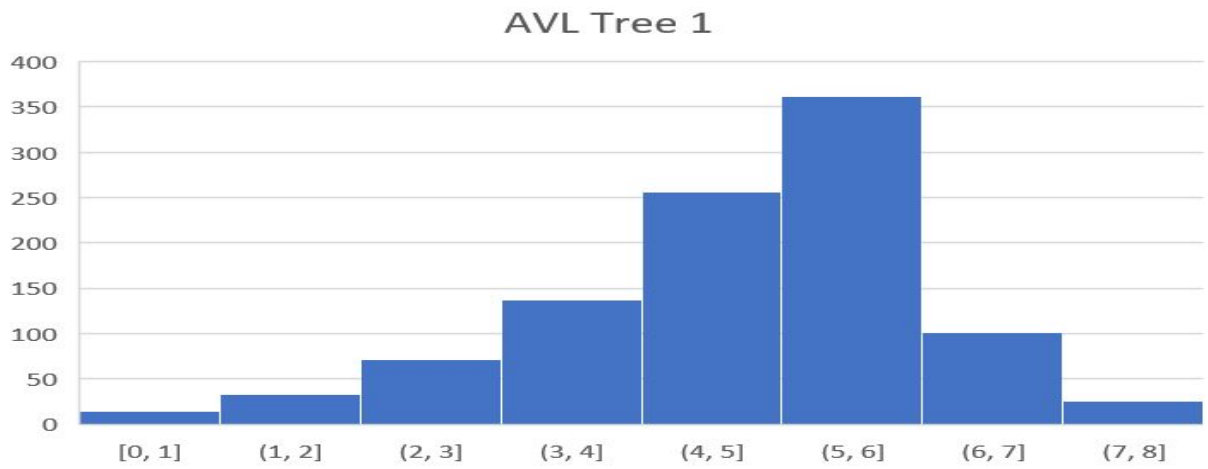
Entry Order by Default:
Entries are ordered from the video game with the highest global sales (Rank 1), to the video game with the lowest global sales (Rank 16599).

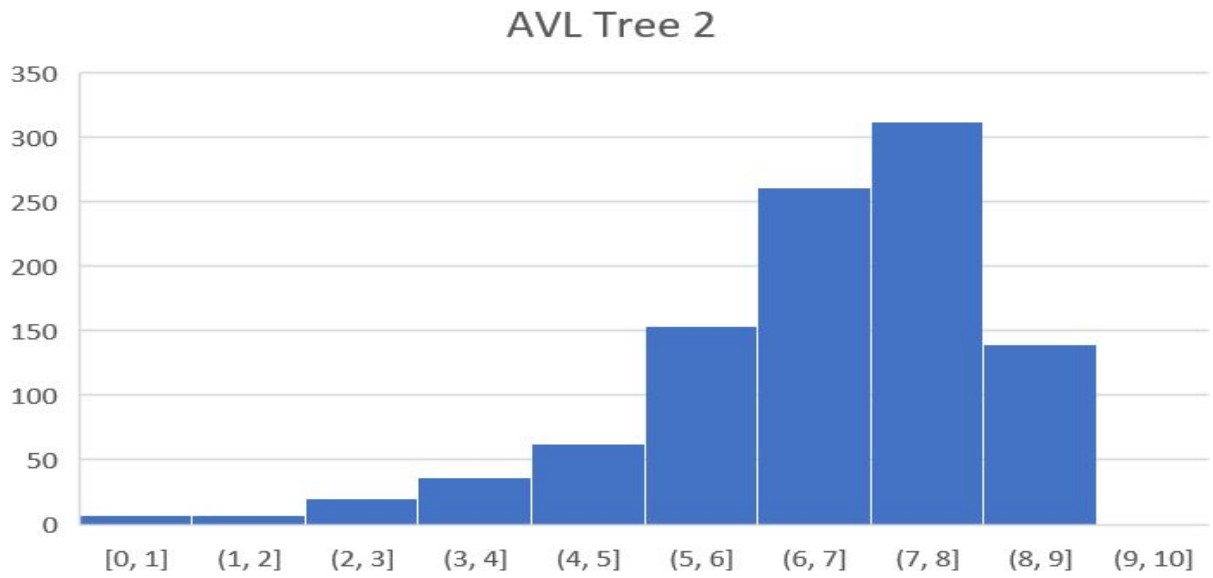
Data/Graphs:
The histograms on the following pages show how many times a node of a certain depth appears. That is, the generational distance from the root of an item found through searching through the respective tree.

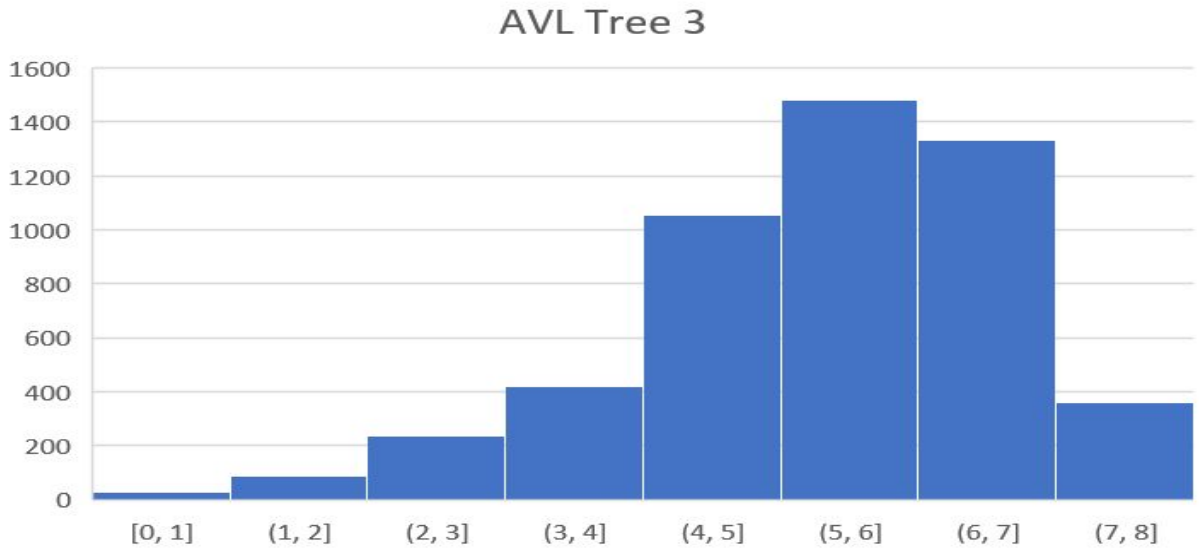

Binary Search Tree 1
160
140
120
100
80
60
40
20
0
[0, 1]
[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]
[6, 7]
[7, 8]
[8, 9]
[9, 10]
[10, 11]
[11, 12]
[12, 13]
[13, 14]
[14, 15]
[15, 16]

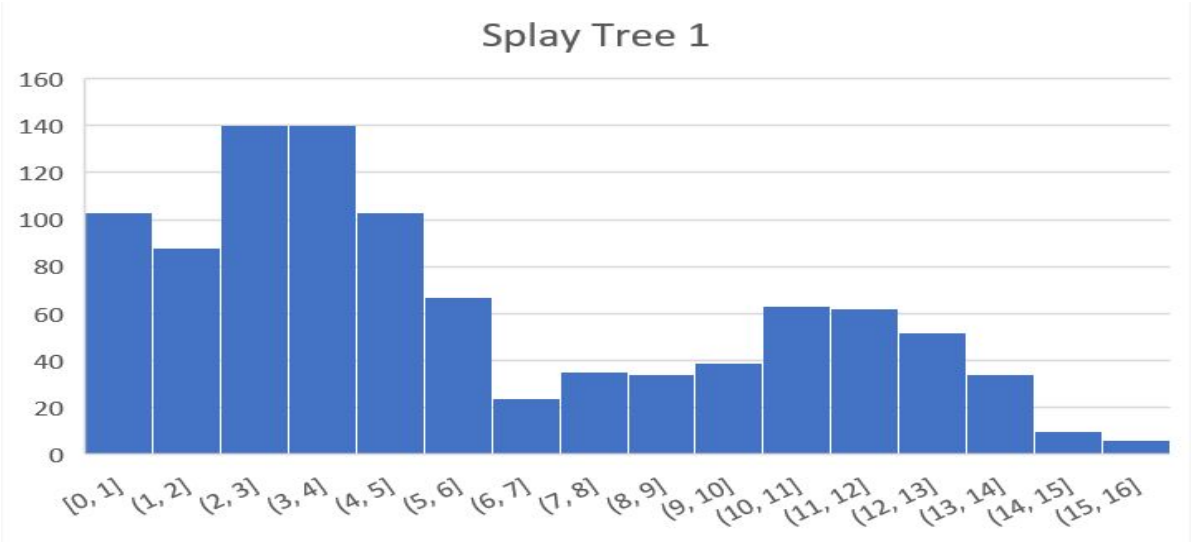

Binary Search Tree 2
180
160
140
120
100
80
60
40
20
0
[0, 1]
[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]
[6, 7]
[7, 8]
[8, 9]
[9, 10]
[10, 11]
[11, 12]
[12, 13]
[13, 14]
[14, 15]
[15, 16]
[16, 17]

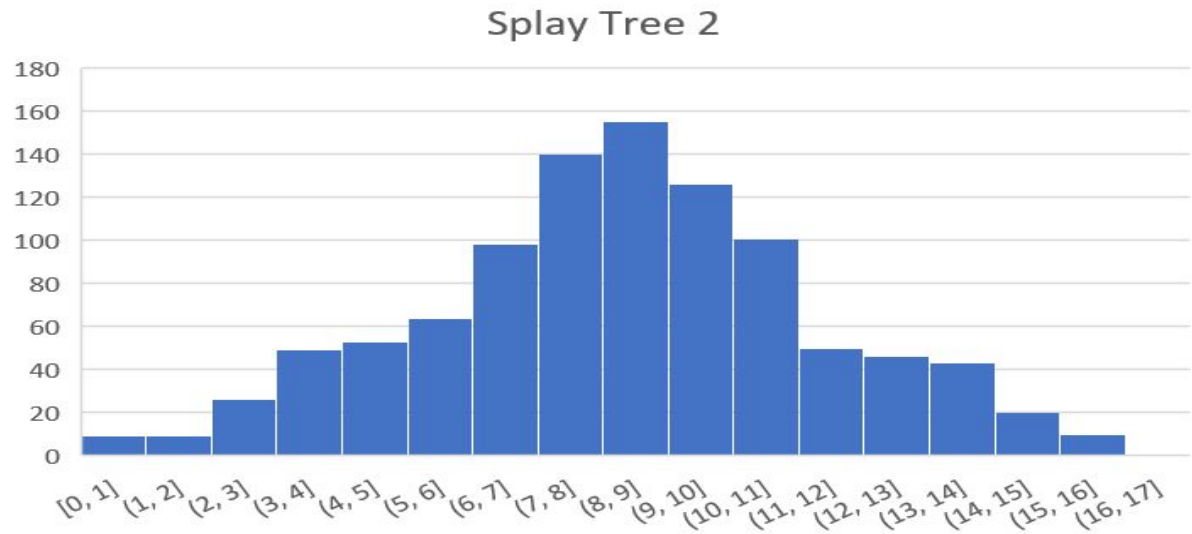

Binary Search Tree 3
1000
900
800
700
600
500
400
300
200
100
0
[0, 1]
[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]
[6, 7]
[7, 8]
[8, 9]
[9, 10]
[10, 11]
[11, 12]
[12, 13]
[13, 14]
[14, 15]
[15, 16]
[16, 17]

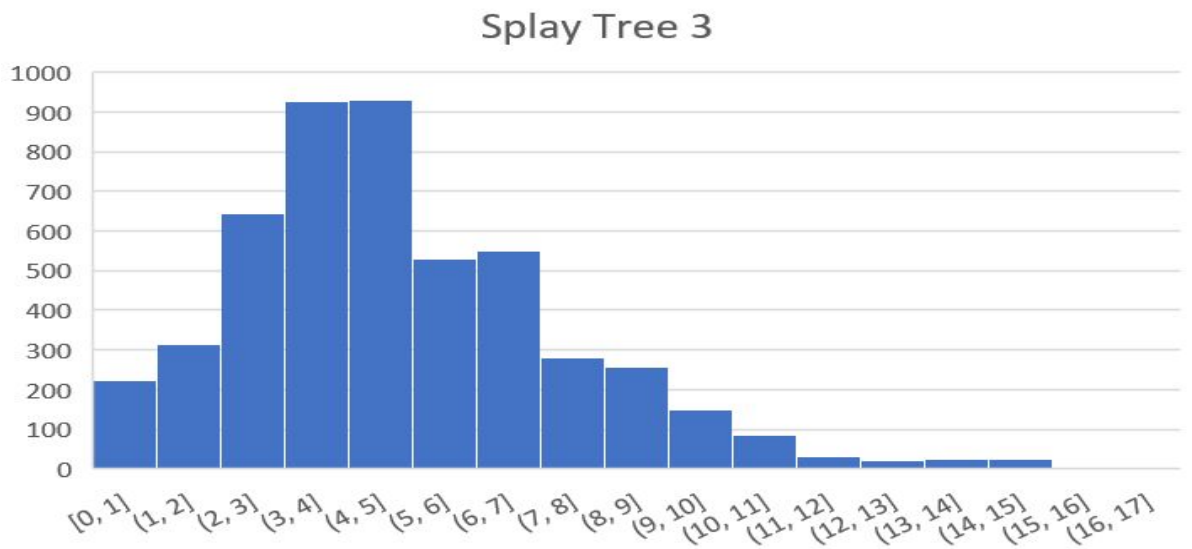

AVL Tree 1
400
350
300
250
200
150
100
50
0
[0, 1]
(1, 2]
(2, 3]
(3, 4]
(4, 5]
(5, 6]
(6, 7]
(7, 8]


AVL Tree 2
350
300
250
200
150
100
50
0
[0, 1]
(1, 2]
(2, 3]
(3, 4]
(4, 5]
(5, 6]
(6, 7]
(7, 8]
(8, 9]
(9, 10]


AVL Tree 3
1600
1400
1200
1000
800
600
400
200
0
[0, 1]
(1, 2]
(2, 3]
(3, 4]
(4, 5]
(5, 6]
(6, 7]
(7, 8]


Splay Tree 1
160
140
120
100
80
60
40
20
0
[0, 1]
[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]
[6, 7]
[7, 8]
[8, 9]
[9, 10]
[10, 11]
[11, 12]
[12, 13]
[13, 14]
[14, 15]
[15, 16]


Splay Tree 2
180
160
140
120
100
80
60
40
20
0
[0, 1]
[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]
[6, 7]
[7, 8]
[8, 9]
[9, 10]
[10, 11]
[11, 12]
[12, 13]
[13, 14]
[14, 15]
[15, 16]
[16, 17]


Splay Tree 3
1000
900
800
700
600
500
400
300
200
100
0
[0, 1]
[1, 2]
[2, 3]
[3, 4]
[4, 5]
[5, 6]
[6, 7]
[7, 8]
[8, 9]
[9, 10]
[10, 11]
[11, 12]
[12, 13]
[13, 14]
[14, 15]
[15, 16]
[16, 17]

Analyze the data. Plot graphs of the depths for the 1000+ searches in each of the three tree structures. Explain if your data makes sense based on what you know about the trees and how they behave. Justify the complexity of searching the trees based on the results:

Both the Binary and Splay trees follow the exact same pattern. They both have "two humps" for the first set of indices, having a very normal distribution in for the second, and being skewed right for the third. If you look closely, the values are almost exactly the same.

This is likely because there was no actual splaying done with the splay tree, meaning that the "order" of the tree is exactly the same as in the binary tree. This means that searches will end at same or similar depths.

The AVL Tree is different, mostly in that it is highly consistent when compared with the data from the binary and splay trees. All of the graphs of depths are skewed left, with most of the depths being held in the third-to-last or fourth-to-last values.

This demonstrates AVL Trees' "rule" that every node's subtrees must not differ in maximum depth by more than 1. In other words, because AVL Trees are highly balanced, the depths found when searching are likely to be consistent.

What happens when you search for something that isn't in the trees? What depth value(s) return from the find methods? Experiment by creating at least three new objects that aren't inthe trees and searching for them. Write about your findings in your report:

When you search for something that isn't in the tree, the LAST stored depth will be whatever depth the last node searched was at before the algorithm stopped once it "realized" that the searched-for node was not present.

When I attempted to search for a value that doesn't exist in any of the vectors (i.e. the item at index 2000 in the vector of 1000 Video_Game objects), each search goes through each node and it's children looking for an item with an equal value, until it stops at node with no children.

In the case of index 2000, the search for each tree will keep going to the right child, as it will always be greater than all the other indices in the tree, until it hits the largest value present (1000) and stops when it finds 1000 has no right children.

Side Note: The depths "found" for the item of index 200 were as follows:

Binary Tree: 5 AVL Tree: 10 Splay Tree: 5

References

Dion, Lisa. “Project 3 Starter Code .” *Blackboard Learn*, https://bb.uvm.edu/webapps/blackboard/content/listContent.jsp?course_id=_136454_1&content_id=_3066750_1.

“How to Make a Histogram in Excel (Step-by-Step Guide).” *Trump Excel*, 3 Sept. 2018, https://trumpexcel.com/histogram-in-excel/.

“Std::random_shuffle.” *Cplusplus.com*, http://www.cplusplus.com/reference/algorithm/random_shuffle/.

“Std::Vector::Clear.” *Cplusplus.com*, http://www.cplusplus.com/reference/vector/vector/clear/.