Anthony Stem
CS 124 | Fall 2021
Project 3 Report

Dataset

This dataset is a subset of the dataset maintained by the Extrasolar Planets Encyclopaedia as of February 28, 2018. It contains data for 3732 confirmed exoplanets from various sources and missions such as Kepler, CoRoT, K2, Kelt, OGLE, and more.[1] This dataset was found on Kaggle and originally contained 97 different columns of data; however, only five columns from the original dataset were used for this project. In addition, a new column called "row" was introduced to create a unique identifier for each row in the dataset. The six fields used in the project are as follows:

• *row* - a unique integer; indicates the row in the database.
• *name* - a string; indicates the name of the exoplanet.
• *mass* - a double; represents the mass of an exoplanet (in terms of Jovian mass J_M).
• *planetaryRadius* - a double; represents the radius of the exoplanet (in terms of Jovian radii J_R).
• *orbitalPeriod* - a double; represents the orbital period (in Earth days) of the exoplanet around its host star.
• *discoveryYear* - an integer; represents the year in which the exoplanet was discovered.

The entries in this dataset are ordered numerically by *row* in ascending order.

The primary reason I chose this dataset is because I have always been fascinated by space, especially exoplanets and extrasolar objects. I wanted a dataset that would allow me to analyze and explore the various properties and attributes of different exoplanets.

Binary Search Tree

Search for 0, 101, and 102. What return values do you get from the find method? What depths do you get? Why?

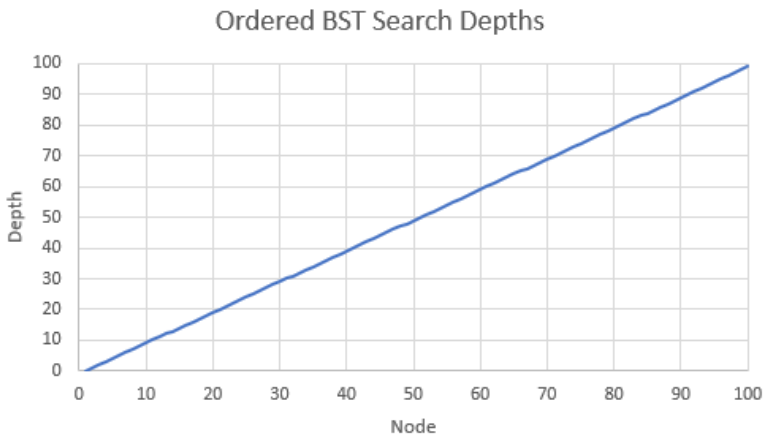
When we search for 0, 101, and 102 (which are not contained in the BST) in the ordered BST of integers 1 through 100, the *find()* method returns the values 1, 100, and 100, respectively. The reason searching for 0 in the binary search tree returns a depth of 1 is because the *find()* function will check the node to the left of the root (the root is 1) since 0 < 1 which will increase the depth of the search from 0 to 1. The reason searching for 101 and 102 returns a depth of 100 is because the *find()* function will go all the way

[1] Original Source: http://exoplanet.eu/
Kaggle: https://www.kaggle.com/eduardowoj/exoplanets-database

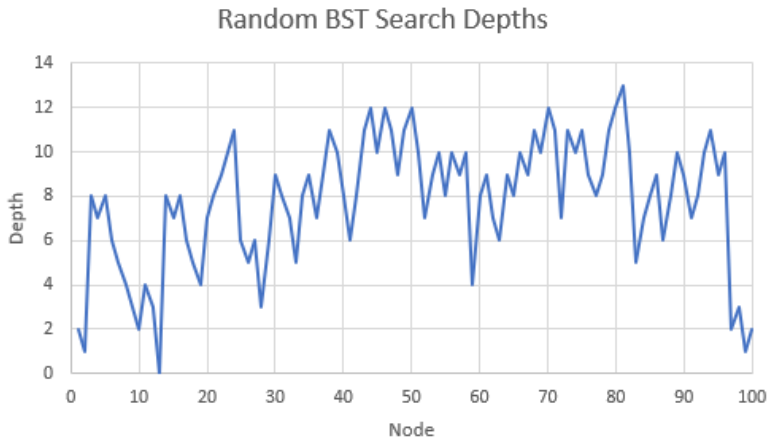
down to the last node in the binary search tree, making the depth 99 (the BST has a height of 100); however, since it doesn't know it is at the final node, it will try to make one last search to the right which will increase the depth to 100, and the find function will return *false* since it will point to *nullptr*.

Insert the numbers from 1 to 100 in random order. Then search for all 100 numbers in the tree and record their depths in a file. How do these depths compare to the first file of depths? Why?

The graph of the search depths for the binary search tree contained the integers 1 to 100 in order (graph of the first file of depths) is shown below:



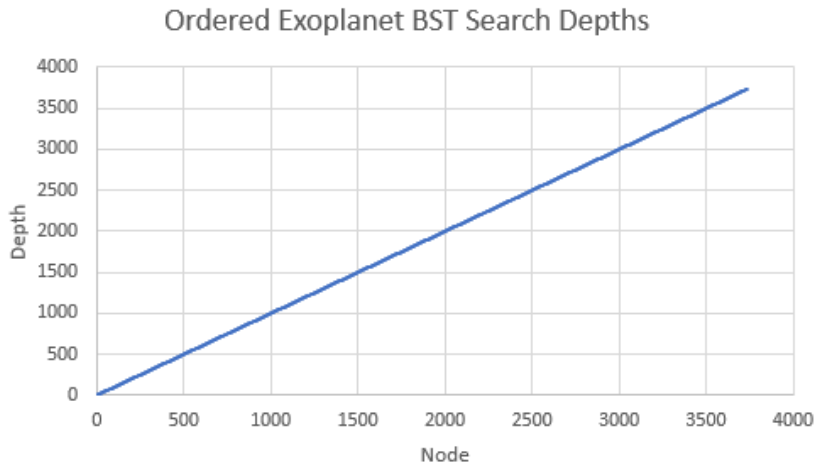
As shown in the graph, the search depths increase linearly from 0 to 99. The maximum depth was 99 and the minimum depth was 0. The average search depth was 49.5. Alternatively, the graph of the search depths for the binary search tree containing the integers 1 to 100 in a random order is as follows:



This graph shows varying depths ranging from 0 to 13 with the average search depth being 7.78. Unlike the second file containing the depths of the randomly ordered binary search tree of integers, the first file of depths (the depths of the binary search tree containing the integers 1 to 100 in order) increases in a linear fashion since the tree is a linked list, meaning searching in the ordered tree will have a complexity of O(n) while searching in the randomly ordered tree will be, on average, O(log(n)).

Create a BinarySearchTree of your custom data type from Project 1. Insert all 1000+ objects from the vector into the tree, then search for all the objects and write their depths to a file. How do these depths compare to the integer BST depths? Why?

Searching for all 3732 exoplanet objects after storing each object in the binary search tree in order gives us the depths shown in the graph below:

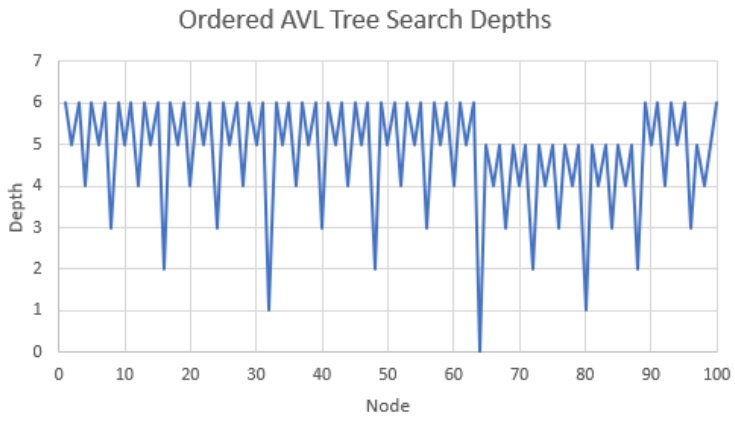


These depths increase linearly from a minimum of 0 to a maximum of 3731. Similarly to the binary search tree with ordered insertion and unlike the binary search tree with random insertion, this binary search tree took on a linked list formation. So, the search complexity for this tree was O(n).

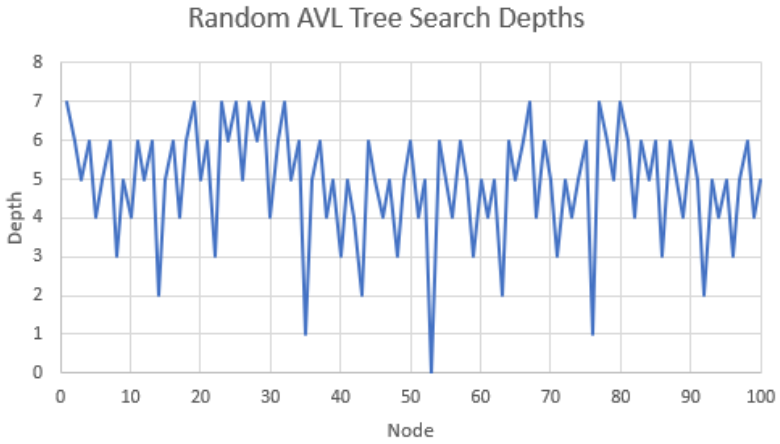
AVL Tree

Create two integer AVL Trees. Insert the numbers from 1 to 100 in order into one tree and in a random sequence in the other tree (in a similar fashion to the BST). Search for all the integers in both trees and record their depths in one or more files. How do these depths compare to the BST depths? Why?

The graph for the search depths for nodes in an AVL tree containing the integers 1 through 100 inserted in order is shown below:



As seen in the graph, the maximum depth searched was 6 and the minimum was 0. The average depth search was 4.8. Next, the graph of the search depths of AVL tree containing the integers 1 to 100 in a random order is as follows:



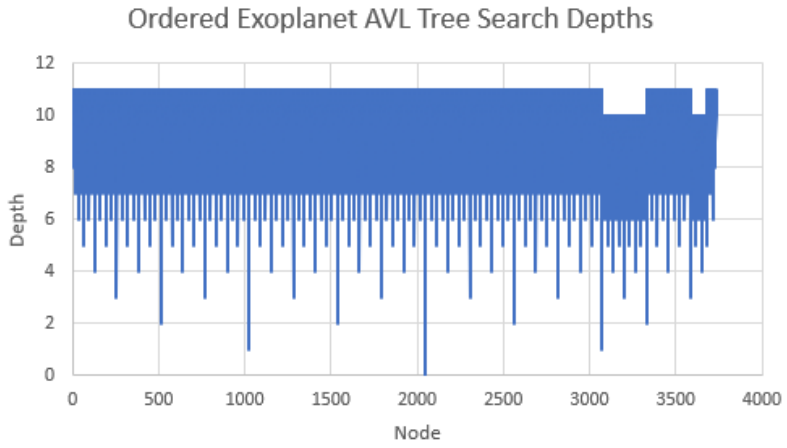
For this graph, we can see the maximum depth was 7 and the minimum depths was also 0. The average search depth was 4.91.

Overall, the average search depths for the AVL trees were lower than the average search depths for the binary search tree. The average search depths for the AVL trees were 4.8 and 4.91 for the ordered insertion and random insertion AVL trees, respectively. On the other hand, the average search depths were 49.5 and 7.78 for the ordered insertion and random insertion binary search trees, respectively. Both searches for the AVL trees followed the expected average O(log(n)) complexity while the searches for the binary search tree had a worst-case complexity of O(n) and an average complexity of O(log(n)), telling us that the AVL tree is more efficient for searching. This is due to AVL trees being a self-balancing tree while the binary search tree is not. Because the AVL trees are balanced, this makes the average and worst-case complexity both O(log(n)); however, as we saw with the binary search tree, the worst case complexity for the binary search

tree is O(n) and the search times were slower on average because the trees were not balanced.

Create an AVL Tree of your custom data type from Project 1. Insert all 1000+ objects from the vector into the tree, then search for all the objects and write their depths to a file. - How do these depths compare to the BST? Why?

Searching for all 3732 exoplanet objects in the AVL tree gives us the depths shown in the graph below:



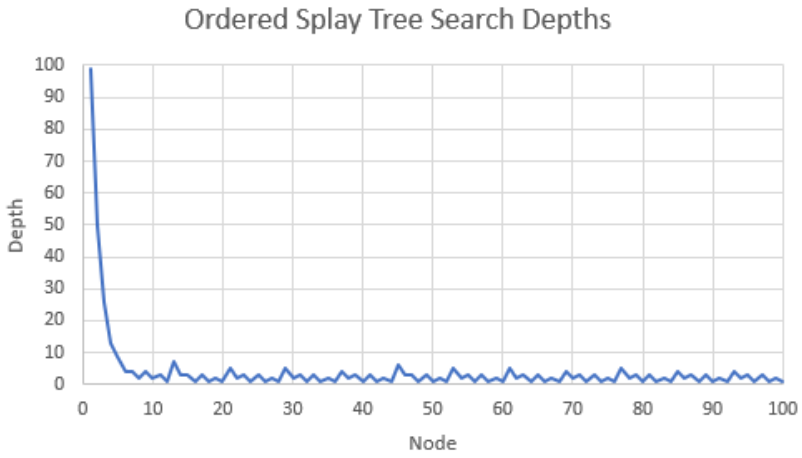
As seen in the graph, the maximum depth of the AVL tree is 11, the minimum is 0, and the average search depth is 9.91 (rounded to 2 decimal places). The search depths for the AVL tree are much smaller than the search depths for the binary search tree as evident by the maximum depth of the AVL tree being 11 with an average depth of 9.91 and the maximum depth of the binary search tree being 3731 with an average of 1865.5. This is due to the nature of AVL trees and how they function compared to binary search trees. The objects in each tree were added in order, so the AVL tree ensured that it remained balanced making the average and maximum depth lower while the binary search tree, on the other hand, ended up making a very long linked list since the row number for each object added into the tree is greater than the previous, which has a much higher average and maximum depth compared to the AVL tree.

Splay Tree

Create two integer Splay Trees. Insert the numbers from 1 to 100 in order into one tree and in a random sequence in the other tree (in a similar fashion to the other two types of trees). Search for all the integers in both trees and record their depths in one or more

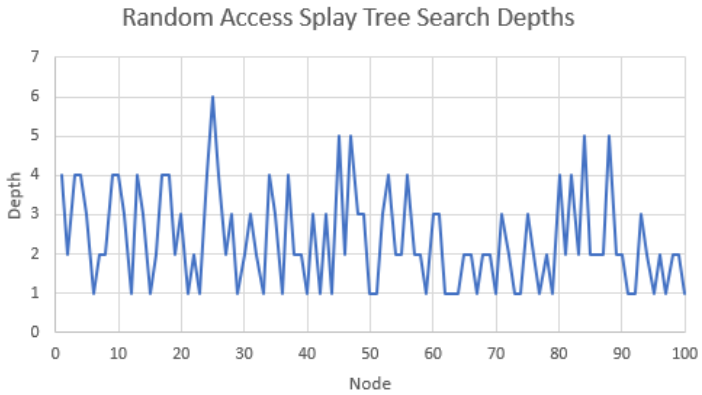
files. How do these depths compare to the depths from the other two types of trees? Why?

The resulting graph for searching for each value sequentially for the splay tree where the numbers were inserted in order from 1 to 100 is shown below:



As shown in the graph, the maximum depth was 99, and the minimum depth was 1. The average search depth for this tree was 4.16; however, the first five values being search for (1, 2, 3, 4, and 5) were all significantly different from the remaining 95 values. The first value, for example, had a depth of 99, which is because the splay tree originally started out as a linked list due to the items being inserted in order, meaning the complexity for searching for the first item was O(n). However, the search depths quickly decreased because of how splaying or searching for an item in a splay tree makes the tree more balanced.

This idea is supported by the next graph which is the graph of accessing each item in a splay tree containing the integers from 1 to 100 that were inserted in a random order:



As seen in this graph, there are no major outliers like there were in the previous one. This is because splay trees are a type of binary search tree so the nodes being added to the tree will be more balanced compared to items being added in order which, as we saw previously, makes a linked list to start out. For this graph, the maximum depth was 6 and the minimum was 1. Also, the average search depth was 2.37.

As mentioned before, the average search depths for the ordered and random splay trees were 4.16 and 2.37, respectively. However, the average search depths for the AVL tree were 4.8 for the ordered insertion tree and 4.91 for the random insertion tree, and the average search depths for the binary search tree were 49.5 for the ordered insertion tree and 7.78 for the random insertion tree. The average depths for the splay tree searches are significantly less than the depths for the AVL tree and binary search tree, overall. However, the splay tree with items inserted in order was only faster than the AVL trees and the binary search trees after repeated searches, which is evident by its initial search depth being very high at 99 but decreasing dramatically (about 50% each time) for the next 4 or so searches (depths decreased as follows: 99, 50, 26, 13, and 9). Overall, however, the splay tree, on average, required the least search depths, the AVL tree required the second most search depths, and the binary search tree required the most search depths for each type of tree overall. The splay tree was likely faster, on average, than the AVL tree and the binary search tree because the splay trees

In the other tree, search for the objects in a random order, and repeat each search five times (i.e. search for the first object five times in a row before searching for the second object, etc.). Record the depths to a file. Why do these depths make sense?

The depths for searching for each node five times created a pattern in the form

$$D_{N_1},\ 0,\ 0,\ 0,\ 0,\ D_{N_2},\ 0,\ 0,\ 0,\ 0,\ldots,\ D_{N_i},\ 0,\ 0,\ 0,\ 0,$$

where D_{N_i} is some non-zero depth for some node N_i. This pattern of depths is logical because of the way splay trees function. When a node is accessed in a splay tree, it becomes the new root of the tree. Therefore, when we search for the same node an additional 4 times, we will find the node at the root every time, which will give us a depth of 0 for each search after the initial search for that specific node.

Complexity, Comparisons & Final Thoughts

Overall, the results from these tests support the expected complexities for searching for searching each type of tree.

The Binary Search Tree's Search Complexity

The graphs and results for the binary search tree tests clearly support the following complexities for searching in a binary search tree:

Case	Complexity
Worst Case	O(n)
Average Case	O(log(n))

We can see the worst case of O(n) in both the graph of depths for the tree with integers 1-100 inserted in order, as well as with the graph of depths for the tree with the 3732 exoplanet objects. Both exhibited a linear increase in depths as the value of the node being searched for increase due to being linked lists, showing us the worst case is O(n). However, the graph of the randomly inserted integers 1-100 had an average depth of 7.78 which is a bit more than the expected average search depth of $log_2(100) \approx 6.68$ but it is still close, and it is still less than the worst case.

The AVL Tree's Search Complexity
The graphs and results for the AVL tree tests support the following complexities for searching in an AVL tree:

Case	Complexity
Worst Case	O(log(n))
Average Case	O(log(n))

We can see that both the worst case and average case for searching in an AVL tree is O(log(n)). This is evident by the results for the searches for each AVL tree which yielded an average depth of 4.8, 4.91, and 9.91 for the ordered insertion tree, the random insertion tree, and the Exoplanet tree, respectively. In addition, the maximum depths were 6, 7, and 11 for the ordered insertion tree, the random insertion tree, and the Exoplanet tree, respectively. These maximum depth values support O(log(n)) as being the worst case because each maximum depth does not exceed their respective value of $log_2(n)$: $6 < log_2(100) \approx 6.68, 11 < log_2(3732) \approx 11.87$ (7 technically exceeds $log_2(100)$, but the max depth technically cannot exceed $1.44 \cdot log_2(n)$, so it is still acceptable).[2] Each average search depth is below the expected worst-case depths $4.8 < log_2(100) \approx 6.68, 4.91 < log_2(100) \approx 6.68$, and $9.91 < log_2(3732) \approx 11.87$, but are still relatively close, which supports the average case being O(log(n)). Not only do the results support an average case of O(log(n)) but the structure of an AVL tree does as well. Because AVL trees are balanced, the nodes are split almost equally

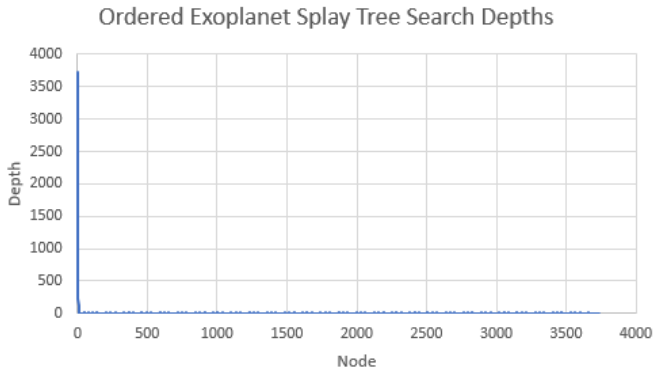
[2] Maximum Depth of AVL Tree: http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html

between the left and right subtrees, which means the number of nodes we have to search through is halved each time we move down the tree, which suggests a complexity of O(log(n)).

The Splay Tree's Search Complexity
The graphs and results for the splay tree tests support the following complexities for searching in a splay tree:

Case	Complexity
Worst Case[3]	O(n)
Average Case	O(log(n))

We can see that the worst case for searching is O(n) when we look at the ordered insertion tree depth graph and the exoplanet depth graph (the exoplanet graph is shown below).



For both of these trees, the first search had a search complexity of O(n). The tree with the integers 1 through 100, for example, reached a depth of 99 when searching for the first item. The first search in the exoplanet reach a depth of 3731 (there were 3732 nodes in the tree). Overall, the depths required to search for a node in the splay tree decreased over time, and remained in line with the average case of O(log(n)).

Final Thoughts
All in all, each tree had their own strengths and weaknesses, and they all yielded varying results. The binary search tree had the worst average search depths, but it . The AVL tree is faster than the binary search tree and it required less searches, on average, to find a node. So, it may be a good choice if a person is looking for fast search times. Finally, the splay tree was especially efficient over many uses, and it would be a great choice for when nodes may need to be accessed multiple times quickly and efficiently.

[3] Worst-Case Complexity for Splay Tree: https://people.eecs.berkeley.edu/~jrs/61b/lec/36