CS 124C/Fall 2021
Project 3
Lars Jensen

Data Set

The data is taken from the website baseball-reference.com and contains data regarding every Major League Baseball (MLB) Team from 1962 to 2012, most recently updated in 2017. While this data originated from baseball reference, I discovered the data set, titled "Moneyball", on the website Kaggle. The data contains 15 fields, and they are as follows:

- **Team:** *string:* The abbreviation of the team that the rest of the data in the row is for.
- **League:** *string:* The league that the team plays in, either the National League (NL) or the American League (AL). This is important as the two leagues have differing rules that can affect a team's statistics depending on the league they play in.
- **Year:** *integer:* The season that the data is from.
- **Runs Scored (RS):** *integer:* The number of runs scored by a team over the entire season.
- **Runs Allowed (RA):** *integer:* The number of runs allowed by a team over the entire season.
- **Wins (W):** *integer:* Number of games a team won that season.
- **On-Base Percentage (OBP):** *float:* The average rate a team gets on base per at-bat.
- **Slugging Percentage (SLG):** *float:* A stat to signify the number of bases each hit results in, and the frequency that they occur. (Similar to batting average but a double is worth 2x, triple 3x, and home run 4x).
- **Batting Average (AVG):** *float*: The average rate at which a team will get a hit per at-bat.
- **Playoffs:** *integer:* 1 if the team made the playoffs that year, 0 if they did not.
- **Season Rank:** *integer:* If the team made the playoffs, this signifies what their season rank was.
- **Playoff Rank:** *integer:* In what place they finished in the playoffs, so 1 would mean that they won the world series.
- **Games Played (G):** *integer:* The number of games a team played in that season.
- **Opponent On-Base Percentage (oOBP):** *float:* The rate at which their opponents got on base per at bat.
- **Opponent Slugging Percentage (oSLG):** *float:* The rate at which their opponents got hits, and how many bases each one resulted in.
- **Team ID (teamID):** *string*: The unique field for each team, created by combing the team's abbreviation and the year of the statistics.

Entries are ordered by year, descending, and in each year, they are ordered alphabetically.

Peculiarities: Season Rank and Playoff Rank are empty if the team did not make the playoffs that year. Also, opponent on-base percentage and opponent slugging percentage were not tracked prior to the 1999 season, so those columns are empty until then.

Binary Search Tree

Integers were inserted in order from 1-100 into a binary search tree. Every number was then searched for, and its depth recorded to a file. *Search for 0, 101, and 102. What return values do you get from the find method? What depths do you get? Why?*

All the searches yield a return value of false, which makes sense as the 3 numbers weren't inserted into the binary search tree. What does vary among them is the depths of the searches. Searching for 0 returns a depth of 1, while 101 and 102 return depths of 100. This is because 1-100 were inserted in order, meaning each number was greater than the one inserted before it. Therefore, each node in the tree only has 1 right child (except for the leaf node), and no left child. When searching for 0, it's less than 1, so it checks the left child, finds nothing there, so it only must go one generation down to figure out it isn't in the tree. For 101 and 102, they are greater than every value in the tree, so the find method traverses all 99 generations until it checks the final node of 100 and finds no right child, returning a depth of 100.

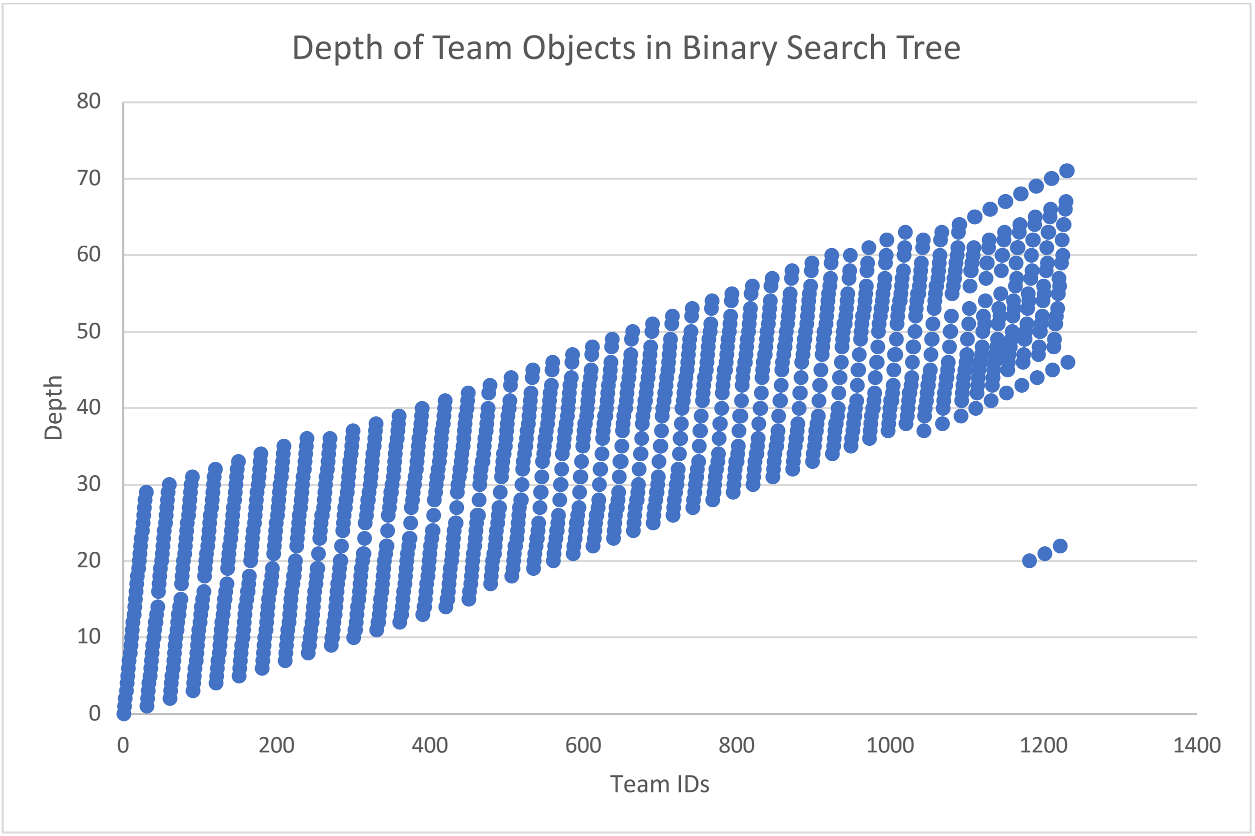
Integers from 1-100 were inserted in random order into a binary search tree. Every number was then searched for, and its depth recorded to a file. *How do these depths compare to the first file of depths? Why?*

These depths were both much more random than those of the in-order ones, as well as smaller. They were more random because the numbers were inserted randomly, so there was no specific pattern to the tree like there was for the ordered insertion (all right children). The depths were smaller because the tree was much more balanced: there were left and right children of nodes, which means more nodes could be in any given generation, unlike the ordered one where there was only one node per generation.

All 1000+ team objects were inserted in order into a binary search tree, and sorted by their Team ID, which is their abbreviation and the year the data is from. Every team was then searched for, and its depth recorded to a file. *How do these depths compare to the integer BST depths? Why?*

The depths of the team objects in the BST look more like the ordered integer BST because there is a pattern to the depths of the teams. The pattern is the teams are in alphabetic order in each year, and then each group of teams per year is in descending order by year. That means every team was just inserted as a right child of the previous team, and then the same team next year would be the left child of that team as their year was smaller. This order can be seen in the trend in the graph, with the outliers being caused by the changing of the teams in the league and their abbreviations (three dots on the far end having depths much lower than the rest is when the Angels changed their abbreviation to LAA from 1962 to 1964). The depths were like the random BST ones in that there was more than object at each depth, and both were more balanced than the ordered integer BST.

Graph of Binary Search Tree:



*Team IDs are in order alphabetically in each year, and in descending order per year.

AVL Tree

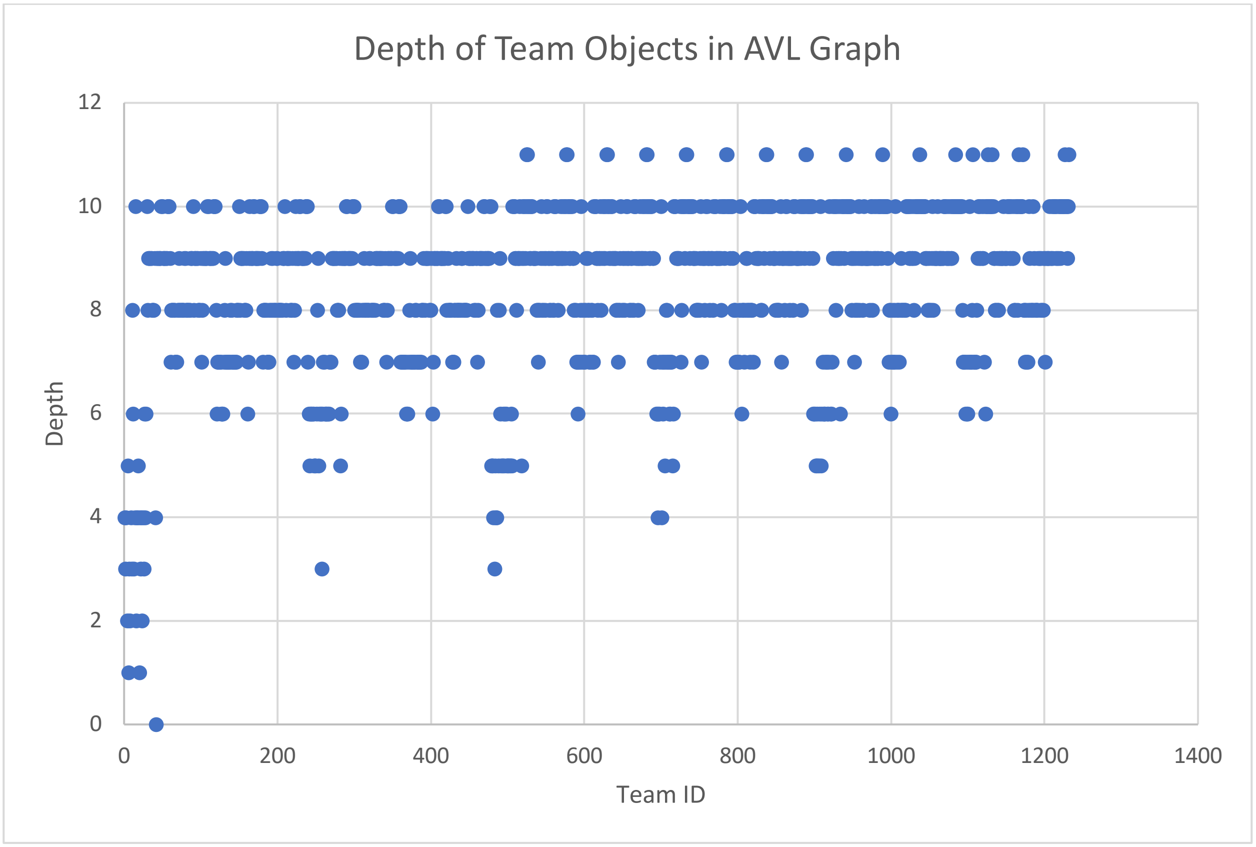
Two integer AVL trees were created, one had 1-100 inserted in order, the other randomly. Every number was then searched for, and its depth recorded to a file. *How do these depths compare to the BST depths? Why?*

These depths are much smaller than the depths recorded for the BSTs. That is because AVL trees must be balanced, meaning that the left and right subtrees of any given node cannot vary by more than one generation. The random and ordered trees had very similar depths then as they were reordered to preserve depth as the numbers were inserted. In fact, the random tree only went one generation deeper than the ordered tree.

All 1000+ team objects were inserted in order into an AVL tree, and sorted by their Team ID. Every team was then searched for, and its depth recorded to a file. *How do these depths compare to the BST? Why?*

These depths are much smaller in general, as the maximum depth for the BST tree is 71, while the maximum depth for the AVL is 11. This is because of the balancing that occurs in AVL trees, so each generation was much fuller than in the team BST, where it is not required to be balanced.

Graph of AVL Tree:



Splay Trees

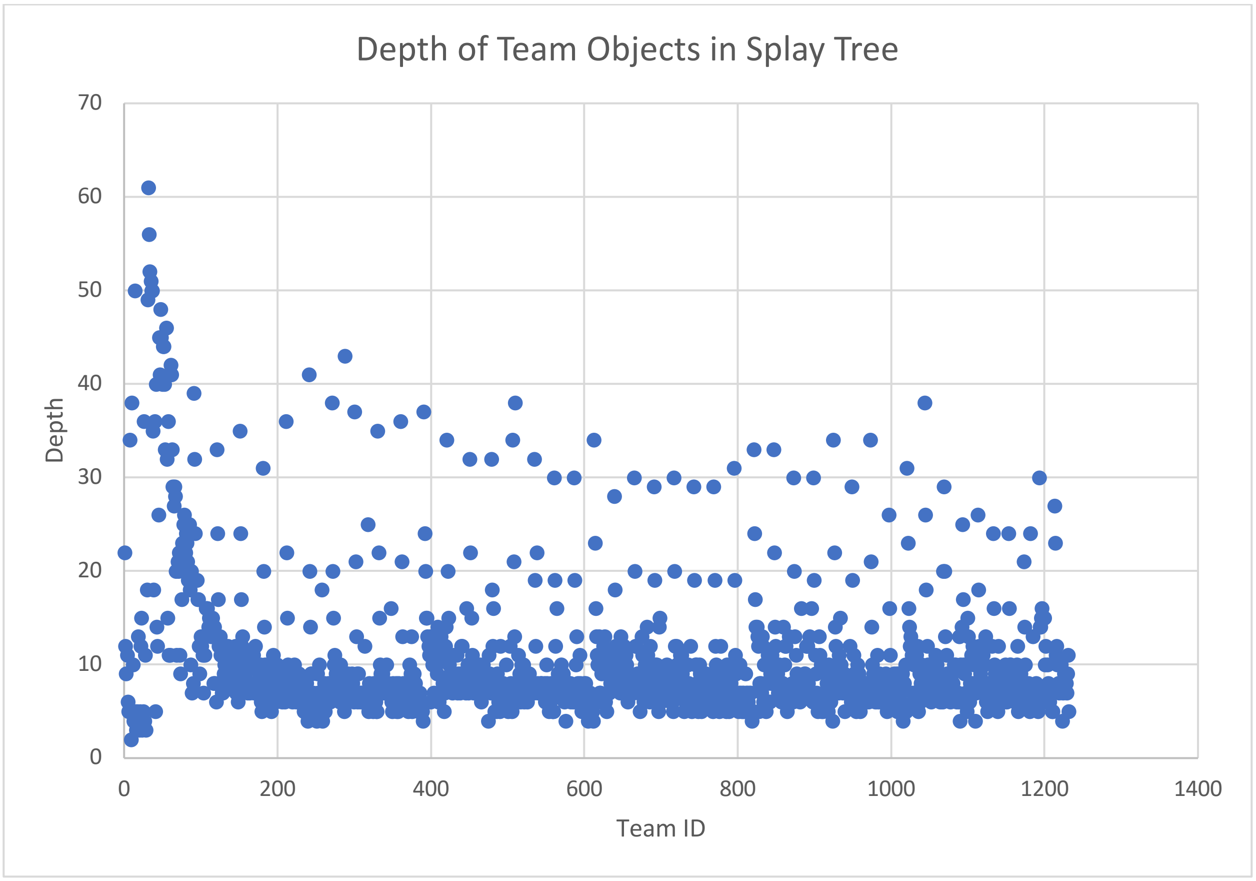
Two integer splay trees were created, one had 1-100 inserted in order, the other randomly. Every number was then searched for, and its depth recorded to a file. *How do these depths compare to the depths from the other two types of trees? Why?*

The ordered splay tree is very interesting, as you can see how the add method works. The first search for 1 returns a depth of 99, meaning 1 is at the very bottom of the tree, the one leaf node. This is because every time an element is added to the tree, it's placed at the root and the rest of the tree is splayed. Because the insertion is in order, it creates a tree that is like the ordered BST tree in that there is one node per generation, just in reverse order. However, the find method demonstrates how the splay tree improves efficiency, as the depths of the searches are essentially halved from number to number for the first 4 searches, while in the BST, the depths of the searches increment by 1. That's because the find method doesn't change the structure of the tree at all in a BST. The random splay and random BST trees had very similar depths, this because the random insertion meant that the trees were relatively balanced to begin with, so the splaying didn't improve efficiency all that much.

When comparing the splay trees to the depths of the AVL trees, the AVL trees in general returned smaller depths for the searches in both ordered and the random trees, and this is due to the balancing requirement of the AVL trees. While the splaying of the trees does make the tree more balanced, it doesn't make it quite as balanced as the AVL tree. The only place where the depths are smaller for the splay tree is towards the end of the searches for the ordered ones, as the constant splaying moves the following numbers closer to the root. This is somewhat offset though by the fact that the depths are so much larger at the start of the ordered splay tree searches.

Two splay trees of team objects were then created, one of which was searched for in order of how they were inserted. The depths of these objects were recorded and graphed.

Graph of Splay Tree:

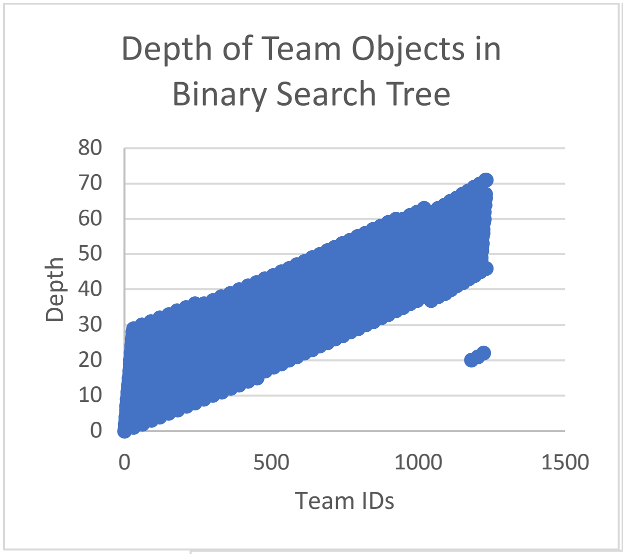


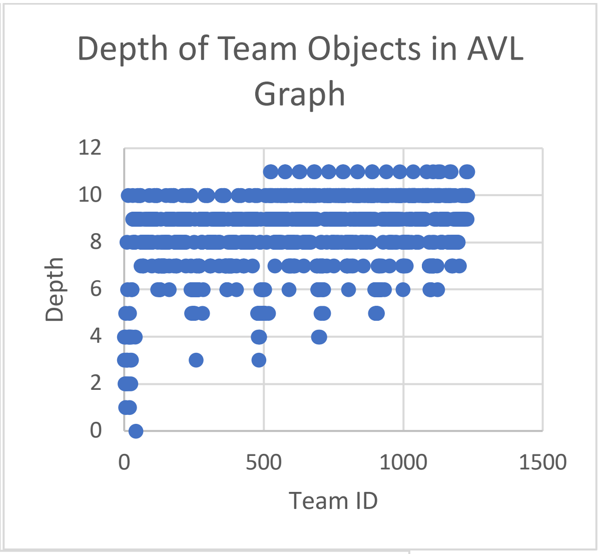
It can be seen clearly in this graph how the splaying of the tree reduces the depths of the searches over time as more objects are searched for. As you move from left to right on the graph, the dots become more condensed towards the bottom, where the depths are lower.

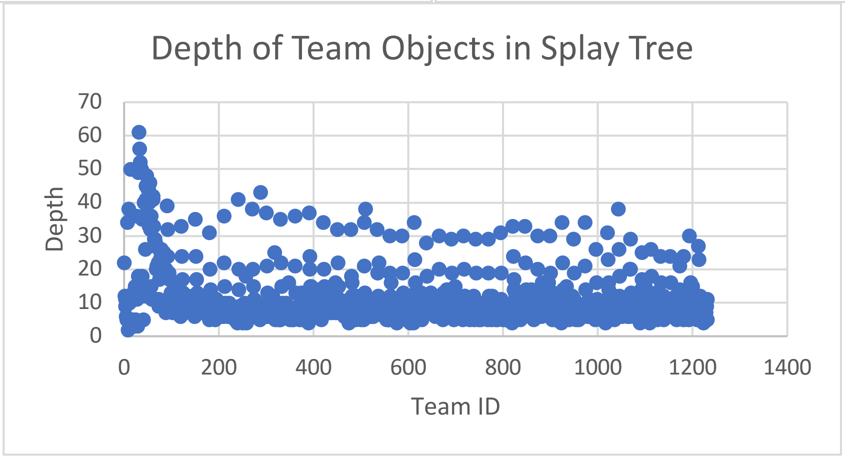
In the other tree, the objects were searched for in a random order, and for each item it was searched 5 times in a row before moving onto the next item. The depths of these searches were recorded to a file. *Why do these depths make sense?*

For all the objects, the first search returned a depth of a positive integer. This makes sense as it couldn't be zero (the root node) as the root node would be the previous node that was searched for due to how splay trees work. The next four searches all returned values of zero, which made a lot of sense as that object had just been searched for, which meant it was splayed and moved to be the root of the tree, making its depth zero.

Comparing the Three Graphs







Compare and contrast the graphs and explain the differences based on what you know about the structure and behavior of the trees. Justify the complexity of searching the trees based on the results.

When looking at all three graphs, it's clear they all at least follow some sort of pattern due to the teams being ordered before they were inserted. The BST and splay trees are almost opposites of each other in terms of their depths, with the BST having lower depths at the start and high ones later, while the splay tree had some high depths at first and then the depths shifted lower as more searches were performed. This is because they have the same structure, it's just that when a new team is added to the splay tree, it becomes the root node, while for a BST it's added as a leaf node somewhere. That explains why the first search of a splay is so deep, as the first team added got pushed all the way down to the bottom as every team after it was placed above it in the tree. Meanwhile in the BST, nodes just keep being added to the bottom, so the later the search, the deeper it had to go to find the team. The searches also got to smaller and smaller depths for the splay tree as each team was splayed after it was found, meaning it was made the root, which balanced the tree and moved the nodes after it closer to the root. The AVL has much different depths than the splay and BST trees, as its balancing requirement meant the depths of the teams were much smaller as almost all generations were full or entirely full for the AVL tree. It is like the BST as the depths of the later teams inserted are deeper because the teams are also inserted as leaf nodes.

Complexities of insertion:
BST: Between O(logN) and O(N)
AVL: O(logN)
Splay: Between O(logN) and (MlogN)

The BSTs time complexity has a worst case of linear and a best case of logarithmic, and this is true and could be seen in the data. It did not follow a linear trend as there were 1200+ objects and a maximum depth of 71, so it was increasing not quite as quickly as O(N). It was also not increasing in a logarithmic fashion, as the time complexity is still increasing over time as you add to it, and this can be seen in the graph as the rate that the depths are increasing doesn't decrease as you add more elements to it. In the AVL tree, the graph shows the logarithmic nature of the tree. This is due to the fact that the tree is so balanced and is almost full, so as you double the number of items added, it still only needs to go one generation deeper to reach the items added. The graph reflects this as if you squint, looking especially at the bottom, you can fit a logarithmic curve to the data. Finally, for the splay tree, the best case is logarithmic, and the worst case is logarithmic multiplied by M, the number of operations performed. This is harder to see in the graph as there are outliers at the start that are super deep due to inserting and searching for the first item first and doing an ordered search. However, there is a clump at the bottom near 0 as well, and from there it follows a logarithmic pattern, just on a larger scale due to the number of items that were searched for.

Outside sources used:
https://www.techiedelight.com/shuffle-vector-cpp/ - Used to learn how to shuffle a vector
https://en.wikipedia.org/wiki/Los_Angeles_Angels - Used to explain outliers on graph due to change in teams name over history.