Abagail Ponsetto

Data 219

Final Project

I chose a data set from Kaggle that focuses on an in-depth analysis of Taylor Swift’s discography. This dataset encompasses both the original versions of her songs and the newer Taylor’s Versions, including comprehensive attributes such as song titles, albums, musical characteristics, and popularity metrics, spanning over 500 songs.

I used a binary search tree structure to navigate and compare the songs based on the key attributes for easy comparison. I set it up for easy user input to compare any songs within the data set and different musical attributes. I used a stack to store and organize my findings using a stack data structure, facilitating the systematic recording and retrieval of comparative analyses. I chose these two because I was most familiar with them and I wanted a way to easily search from the dataset and compare.

My question for this data set is to discern whether Taylor Swift’s new rerecorded songs significantly differ in musical attributes and if that affects popularity. If Taylor’s Versions of songs are more popular than the original records, not because they are drastically different but because the fan base would want to support the artist’s work after the masters were sold out from under her.

Upon reviewing the dataset, I undertook data cleansing procedures to ensure its integrity and relevance. This involved the removal of extraneous columns such as identifiers and track numbers, as well as excluding live concert albums to streamline the analysis towards studio recordings. I have decided to keep the date for release in case I decide to use it later.

In processing Taylor Swift's Spotify data, I began by importing the pandas library to manage the CSV dataset. This dataset includes essential song details like name, album, release date, and various musical attributes such as danceability, energy, and popularity. After reading the CSV file into a DataFrame, I sorted the data by popularity in ascending order, creating a new DataFrame for the sorted data.

To facilitate efficient data management and comparisons, I designed a Node class to represent each song's attributes and a Binary Search Tree (BST) class to organize and search the songs based on their names. The BST allows quick insertion and retrieval of songs, making it suitable for comparisons based on specific attributes.

Additionally, I implemented a Stack class to store song names sorted by popularity. Using this stack, I populated it with song names sorted by popularity from the sorted DataFrame. This process ensured that I could easily access and compare songs based on their popularity rankings.

The program prompts user input for two song names and an attribute for comparison, such as energy, tempo, or popularity. Leveraging the BST's compare\_attributes method, the program conducts the requested comparison and provides the result, indicating which song has a higher value for the specified attribute or if they are equal.

Finally, the program retrieves and displays the comparison results stack, showcasing the details of all comparisons made during the session. This comprehensive approach not only organizes and compares the songs effectively but also ensures a user-friendly interaction for exploring and understanding the dataset's musical attributes and rankings.

In comparing the performance of Binary Search Trees (BSTs) and Stacks for managing and retrieving song data from Taylor Swift's Spotify dataset, several factors come into play. BSTs excel in searching and retrieving data based on specific criteria, such as song names or attributes. Their average time complexity for insertion and retrieval operations is O(log n), where n is the number of elements in the tree. This logarithmic time complexity implies that as the number of songs increases, BSTs maintain efficient search and retrieval speeds. Stacks primarily support operations like push (insertion) and pop (removal) in constant time O(1). Stacks are not inherently designed for searching or retrieving specific elements based on attributes like popularity or release date. While they can store items in a last-in-first-out (LIFO) manner, searching for specific songs or attributes would require iterating through the entire stack, resulting in linear time complexity O(n), where n is the number of elements in the stack.

Both BSTs and Stacks consume memory proportional to the number of elements stored. BSTs require additional memory for node structures and pointers, but this overhead is generally manageable compared to the benefits they offer in efficient data retrieval. Stacks, being simpler in structure, have lower memory overhead but lack the sophisticated search capabilities of BSTs.

In terms of scalability and flexibility, BSTs shine in scenarios where precise search and retrieval operations are crucial, such as comparing songs based on attributes like popularity or release date. They maintain logarithmic time complexity even as the dataset grows, making them suitable for handling large datasets efficiently. Stacks, while efficient for basic push and pop operations, are less suitable for complex search and comparison tasks. They are better suited for scenarios where the order of items is essential, such as maintaining a sorted list of song names based on popularity.

In conclusion, for tasks involving detailed comparisons and retrieval based on specific attributes like popularity or release date, Binary Search Trees offer superior performance and scalability compared to Stacks. Stacks are more appropriate for simpler operations where LIFO behavior is sufficient and precise searching or comparison is not a primary requirement.