Colby Wirth

COS 285

Assignment 6

Dr. Behrooz Mansouri

5 November 2024

Analysis for Program 6 sub-tasks

Note about Heap Memory errors

With each run, I kept encountering the error java.lang.OutOfMemoryError: Java heap space: failed reallocation of scalar replaced objects. I attempted to manually call the garbage collector to clear unused memory, but the problem persisted. Because of this, I limited the Search Engine to just one genre of songs: rock. However the code to run the program with all songs is still available in program6.java.

File Input and Insert Functions

The input function, MyDataReader.readFileToArrayList, is an extension of the function MyDataReader.readfileToBST as it takes the outputted BST and converts it to a sorted ArrayList¡song¿.readFileToBST has a best case time complexity of $O(log(n)$, and worse case of $O(n)$. The BST is then converted to an ArrayList using the function BinarySearchTree.ToSortedArrayList. This function operates with an in order traversal, which executes in $O(n)$. Thus, the best case time complexity for the file input and insertion is $O(n+log)$ and worse case of $O(n+n)$ which simplify to $O(n)$ in both cases.

Constructing a MySearchEngine object

Term Frequency

When the constructor is called, three functions are executed to build a MySearchEngine object. The first function, MySearchEngine.calculateTF calculates the term frequency for every lyric with respect to its song. Each song is stored as a key in a TreeMap that encloses another TreeMap as its value. The key set of each inner TreeMap represents each lyric in its respective song. The value is the term frequency, stored as Double. The outer TreeMap is populated in $O(nlog(n)$, as the insertion method 'TreeMap.put' inserts each element in the average case of $O(log(n))$ time. Each inner TreeMap is populated in $O(mlog(m))$ (where m is the number of lyrics per song), time as the 'TreeMap.put' method is used again. Finally, for each song, its inner TreeMap is iterated on a second time to properly set the term frequencies. This is again done in $O(mlog(m))$

time as 'TreeMap.put' is used. The overall time complexity of this algorithm is $O(n(log(n)) * (m(log(m)) + m(log(m))$. This simplifies to $O(nm(log(nm))$.

Inverse Document Frequency

The second function called by the constructor, MySearchEngine.calculateIDF, computes the Inverse Document Frequency (IDF) for each lyric for all inputted songs. For each of the n input songs, the lyrics are split into an array of size m, this is done in $O(nm)$ time. A TreeSet is used to store all k unique lyrics (where $k \leq m$). This is also done in $O(nm)$ time.

Each unique lyric is inserted into the idf TreeMap using TreeMap.put, with a complexity of $O(mlog(m))$ per song. This is $O(nm(log(m)))$ for all songs. After building the map, the values are updated with the calculated IDF and reinserted into the map. This is done in $O(mlog(m))$ time. By combining each component, the overall complexity is $O((nm \cdot (mlog(m))) + m(log(m)))$. This simplifies to $O(nm(log(m))$.

Average Length

The third and final method called by the constructor is MySearchEngine.calculateAvgLength. This function iterates on all songs and inserts each song with its length as the key to the 'avgLength' TreeMap with TreeMap.put. After the first iteration, each song is iterated on a second time. The value of each pair is divided by the average song length and then put back to the TreeMap. Both of these iterations with puts occur in $O(nlog(n))$ time complexity so the overall time complexity is $O(2nlog(n))$ which simplifies to $O(nlog(n))$.

Total Time Complexity for Indexing

For the overall time complexity, each of the subroutines is added together because each function operates independently. $O(nm(log(nm) + nm^2(log(m) + nlog(n))$. This simplifies to $O(nm^2(log(m)))$.

Searching Algorithm

Calculating Relevance

The Searching algorithm has two made sub-routines: (1) Calculating Relevance and (2) Sorting. The first sub-routine, MySearchEngine.calculateRelevance, calculates a relevance score for each term t in a query, and for every song n. Thus this time complexity is $O(nt)$.

Sorting

The second sub-routine, MySearchEngine.sortedByValue sorts all relevance scores

for n songs. This is done with Collections.sort which sorts the songs in $nlog(n)$.

Total Searching and Sorting
The total time complexity is $O(n(log(n) + nt)$ as each sub-routine is executed independently. This simplifies to $O(nlog(n))$.

Output Results for Queries

11714 milliseconds to build the index

Results for we are the champions
1: Where Shadows Dance by Anameth 0.21
2: King Tut v2 by Steve Martin and the Toot Uncommons 0.19
3: To Live Is to Die by Metallica 0.13
4: Digestive system by Angesof darkness 0.12
5: I Need You by M83 0.11
533 milliseconds to search for we are the champions

Results for i will always love you
1: Closedown by The Cure 0.31
2: Red Like Roses Red Trailer by Jeff Williams 0.29
3: Hands Away by Interpol 0.27
4: Put Your Back N 2 It by Perfume Genius 0.25
5: All Alone by Saturnus 0.23
720 milliseconds to search for i will always love you

Results for walking on sunshine
1: Pancake by Swirlies 0.46
2: Loomer by my bloody valentine 0.46
3: Palisade by Mineral 0.30
4: Big Money by Big Black 0.22
5: Nightclubbing by Iggy Pop 0.19
232 milliseconds to search for walking on sunshine

Results for dancing in the rain
1: Palisade by Mineral 0.39
2: Pigs on the Wing Part One by Pink Floyd 0.30
3: Alerion by Asking Alexandria 0.29
4: Tea for the Tillerman by Cat Stevens 0.29
5: Reckoner by Radiohead 0.24
521 milliseconds to search for dancing in the rain

Results for put your hands in the jupitersky
1: Hands Away by Interpol 0.73

2: Put Your Back N 2 It by Perfume Genius 0.34
3: No Need for Introductions Ive Read About Girls Like You on the Backs of Toilet Doors by Bring Me The Horizon 0.26
4: Beach Baby by Bon Iver 0.22
5: All Alone by Saturnus 0.19
721 milliseconds to search for put your hands in the jupitersky