Alexander Dean

COS 285

11 October 2017

Report: Program 5

Program 5 completes the implementation of the AddenumList started in Program 3. Specifically, three methods and an iterator were implemented. The method mergeAllLevels() will combine all L2 arrays into a singular L2 array at index 0 in the L1 array. Like most other Java data structures, the toArray() method returns an array of the elements in the AddendumList. The subList() method returns an AddendumList with only the elements between two specified elements. Finally, basic iterator methods, such as next() and hasNext() were implemented to satisfy the iterator interface.

The mergeAllLevels() method is quite simple in its design. The code loops as long as there is more than one L2 array and calls the merge1Level() method designed in Program 4. This method, as mentioned in the previous report, merges the last two L2 arrays, therefore reducing the number of L2 arrays used (represented by the member l1numUsed). After this is done, the program checks to see if the newly merged array is full. If it is full, it will create a new L2 array in the next index for future additions. This is not a necessary feature, but was suggested by the comments of this method. Personally, it adds a layer of complexity to the code that is unnecessary. It was noted in the response to Program 4's report that merge1Level() could be performed with a Big-O order of O(N). In the worst case, where the L2 array sizes are N/2, N/4, N/8 and so on, it can be noted that the elements in the last array will be copied l1numUsed – 1 times ($\log_2 N - 1$ approximately), the elements in the second to last array will be copied l1numUsed – 2 times, and so on. This

behavior, which is similar to that of an ArrayList resizing, follows approximately an $O(N^2)$ curve.

The toArray() method is very straightforward in its function. It copies the contents of each of the L2 arrays into the inputted array. It is very effective because it utilizes System.arraycopy() instead of a for loop. When this code was initially implemented, the array was sorted after all of the elements were copied over. This was done using the mergeAllLevels() method on a new AddendumList object to keep the order of the list stable with respect to the original structure. Because it uses this method, the Big-O order of this method is at least $O(N^2)$. However, the System.arraycopy() method should be approximately O(N), so the overall method is around $O(N^2 + N)$.

Sublist() was the most complicated method in this program assignment. It first creates a new AddendumList to store the results in. It copies over all elements of the original list into this new list and merges them into a single L2 array. This portion of the code is approximately O(N) time, due to each element of the array being copied. A constant factor is also present due to the time to allocate memory for the array and other variables, but is dwarfed by the linear nature of copying. Next, the index of fromElement is found in the merged list using the findFirstInArray() method implemented in program 3. The index of toElement() is found using findIndexAfter() (which was not suggested in the program 5 handout, understandably so). The program then checks to make sure the element before this is not the same as toElement, for the sublist requirement called for the toElement to be excluded from the list. It has been noted in past reports that both findFirstInArray() and findIndexAfter() are approximately O(logN) time, and the copying of the sublist elements to

a new array is, in the worst case, O(N) time. All of the methods used in this method give it a Big-O order of approximately ($2N + 2\log N + N^2$). This simplifies to $O(N^2)$.

The iterator was quite simple to implement. It calls mergeAllLevels() on the AddendumList to avoid having to handle multiple tiers of L2 arrays and initializes index to 0. It then behaves identically to an ArrayList iterator in that it returns the element at index and increments index for every next() call, and returns true if the element at index is not null and index is less than the size for every hasNext() call. Iterating through the list is purely O(N) time, but instantiating the iterator is $O(N^2)$ time due to calling mergeAllLevels().

At this time, there are no known bugs with the code, and it has already been tested successfully with program 6 with promising results. The results for program 5, however, can be found below.

```
testing routine for Addendum List
insertion order: qwertyuiopasdfghjklzxcvbnmamz@~
The number of comparison to build the Addendum List = 88
TEST: after adds - data structure dump
[0] -> [a][d][e][f][g][h][i][o][p][q][r][s][t][u][w][y]
[1] -> [b][c][j][k][l][v][x][z]
[2] -> [a][m][m][n]
[3] -> [@][z][~][ ]
STATS:
list size N = 31
level 1 array 4 of 4 used.
level 2 sizes: 16 8 4 3
Successful search: min cmps = 2, avg cmps = 7.1, max cmps 18
TEST: toArray
[@][a][a][b][c][d][e][f][g][h][i][j][k][l][m][m][n][o][p][q][r][s][t][u][v][w][x][y][z][z][~]
TEST: sublist(e,o)
[0] -> [e][f][g][h][i][j][k][l][m][m][n]
[1] -> [ ][ ][ ][ ]
[2] -> null
[3] -> null
TEST: sublist(emu,owl)
[0] -> [f][g][h][i][j][k][l][m][m][n][o]
[1] -> [ ][ ][ ][ ]
[2] -> null
[3] -> null
TEST: iterator
[@][a][a][b][c][d][e][f][g][h][i][j][k][l][m][m][n][o][p][q][r][s][t][u][v][w][x][y][z][z][~]
```