Alper İNAN 210316001 *28.12.2021*

CSE 2105 Data Structures Fall Semester Project

Q1: I thought the best approach for this problem was to build an AVL Tree [O(logn)] algorithm. Integrating a linked List [O(n)] or an array would be very costly, as the user (in theory) could enter an infinite number of data. Because with each new number entered, all the elements in the list had to be checked and this would become more costly with each new number. While any hashing approach would still work more effectively than linked lists or arrays, it would be much slower than an AVL Tree. Because the inputs would be infinite, I would not be able to do Linear or Quadratic Probing in an effective way. Since these systems basically work according to the length of the arrays, there would be no space left in the array after a large number of entries. With the separate chaining method, I would basically have built it on the linked list again, which means visiting many elements at every time.

With installing the system as AVL Tree:

- I placed the new entries directly where they should be, I did not visit any element unnecessarily.
- I have prevented unnecessary spaces to occur on the disk with arrays or hashing algorithms.
- Additionally, I were able to keep all entries in order.

The code is complicated so to understand easily, here is the tasks of some important methods:

- **add(int i)** = a recursive function, calls itself if the i is greater than the current node, or vice versa. After completing adding, checks the balance situation with the help of balanceFactor() function which is basically extract the height of the left child and the right child of the node, if there is an unbalancing situation, calls the balance() method.
- **balance(Node node)** = takes the current unbalanced node as an argument. Decides which rotation(s) needed, then rotates. Return the new balanced subroot.
- **rotateRight(Node node)** = makes a right rotation than returns the new subroot.
- **rotateLeft(Node node)** = makes a left rotation than returns the new subroot.
- **traverseInOrderAndPrint()** = recursive function that traverse the tree in order and prints the values of the nodes.
- **assignTraversalToArrayList(ArrayList<Integer> list)** = assign the node values to an arraylist(no need for the problem just to sort with java's own methods)

Q2: In this problem, I realized that I could solve the problem very effectively by making an addition to the existing AVL Tree system. I have placed the elements of the L1 and L2 lists directly in the Tree and I left Tree inactive when collisions occurred. In this way, unique values in both lists remained in our Tree, no values were doubled. Finally, I assigned

the tree of the combination of lists to an Array List with an in-order traversal(so it is made sorted initially) function, then I converted the list to an Array and had the Array List cleaned by the Java Garbage Collector to reclaim the free space that the Array List would occupy on the disk.

Q3: I thought that the best approach to this problem is to hash the incoming strings and assign them to an array with a MyLinkedList object in each index. I created our linked list ourselves because unlike java's own linked list, I wanted the list not to accept it when trying to add the same string. In this way, I both prevented the same string from being repeated and ensured that each unique string was protected in possible collisions. I used the prime number 997(I chose this number because I didn't want to take up too much space on the disk, and I wanted the linked lists in each index not to get too long. For example, in the book Harry Potter and the Philosopher's Stone, 6185 unique words were used, if our function works ideally, the lists in each index should be 6 - 7 objects long.) and 31 (like java) to minimize collisions both in creating the array and in the hash function.

Q4-Q5: I dealt with these two problems together, that is, I solved them in the same main method. I used the same technique as Q3. I created a scanner in our sample file and assigned each number it read to an array with linked lists (MyLinkedList) at each index, using the hash algorithm. To count the unique words, I created a static integer variable that increments with each new object created from the MyLinkedList class and I printed this variable to the user. Then, I assigned the String data in the separate lists in the hashmap and the count variable, which increases when trying to add the same word, to an ArrayList. In order to print the word numbers in alphabetical order, I sorted this ArrayList with java's own method and finally printed it to the user.