Sofia Valiante 40191897, Karina Sanchez Duran 40189860
COMP 352
Assignment 3
December 5th, 20222

Assignment 3: Part 2

Question 1

```
Algorithm add(elasticERLObj, key, value)
    Input an instance of the ElasticERL class, the key of the entry as a String, and the value of the
    entry as a String
    Output void
    if elasticERLObj.size < 1000 then
        tree ← elasticERLObj.getAVLTree()
        tree.setRoot(tree.insert(key, tree.getRoot(), value)
    { else }
        elasticERLObj.getHashTable().put(key, value)

Algorithm remove(elasticERLObj, key)
    Input an instance of the ElasticERL class and the key of the entry as a String
    Output void
    if elasticERLObj.size < 1000 then
        tree ← elasticERLObj.getAVLTree()
        tree.setRoot(tree.deleteNode(tree.getRoot(), key))
        sizeOfERL ← sizeOfERL - 1
    { else }
        elasticERLObj.getHashTable().remove(key)

Algorithm getValues(elasticERLObj, key)
    Input an instance of the ElasticERL class and the key of the entry as a String
    Output String of the value of the entry
    if elasticERLObj.size < 1000 then
        tree ← elasticERLObj.getAVLTree()
        if (tree.searchAVL(key, tree.getRoot())) = null then
            return “key does not exist”
        return tree.searchAVL(key, tree.getRoot()).getValue()
    { else }
        return elasticERLObj.getHashTable().get(key)

Algorithm setEINThreshold(size)
    Input integer value of the size
    Output instance of the superclass Object
    return new ElasticERL(size)
```

<u>Question 2</u>
(see zipped files)

<u>Question 3</u>
3. Discuss how both the time and space complexity change for each of the above methods depending on the underlying structure of your ElasticERL (i.e. whether it is an array, linked list, etc.)?

a) A detailed report about your design decisions and specification of your ElasticERL ADT including a rationale and comments about assumptions and semantics.

setEINThreshold(size)
The method creates an instance of the class ElasticERL with the size of the number of entries as the argument passed. If the size is less than 1000, then an AVL of space complexity O(n) is created, and the time complexity is O(1).

If the size is greater than or equal to 1000, then a hash table object is created and has a time complexity of O(1) and a space complexity of O(n).

generate(ElasticERL)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. The time complexity of the operation is O(n) because the function performs an inorder traversal of the tree and at the same time inserts into an array of the size of the ElasticERL object. Once the array has been constructed, the function accesses the array at a randomly generated index and the next index in the array, and generates a random 8 digit number between the keys of both the indexes of the array. As such, the inorder traversal of the tree makes the function's, as a whole, time complexity O(n). The space complexity as well is O(n) because both an array and tree are of size n and are used in parallel.

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Thus, a random number will be created and the function *keyExists(k)* will be called from the *myHashTable* class in order to ensure that the key is not already in the hash table. If the key is not in the hash table, it will return the newly generated number. Otherwise, it will continue to generate random numbers until it does one that is not already in the hash table. The *keyExists(k)* function has a time complexity of O(1) at best and O(n) at worst.

allKeys(ElasticERL)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. The method *allKeys()* makes use of an internal method of the ElasticAVL class that performs an inorder traversal of the tree and inserts each node of the tree into an array in sequential order. As such, the method has a time complexity of O(n) and a space complexity of O(n). The internal functions *inorderTraversal()* returns an array, and *allKeys()* returns an instance of the superclass Object.

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Therefore, the *allKeys(ElasticERL)* function will call the *allKeysInHash(ElasticERL)* which, in turn, will call the *getAllKeys()* function from the *myHashTable* class. The *getAllKeys()* function has a time complexity of O(n) and iterates through the entire hash table in order to obtain all the keys and sort them.

add(ElasticERL, key, value)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. The time complexity for an insert operation into an AVL tree, which is handled with an internal function of the class ElasticERL, is O(logn). Additionally, if the tree needs to rebalance after an insertion, then a single restructure operation will take O(1), and multiple will take O(logn). Thus, *add()* considering only the AVL ADT will therefore have a time complexity of O(logn), and a space complexity of O(n) because an AVL tree has a space complexity of O(n).

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Thus, the *put(key, value)* function is called from the myHashTable class. The *put(key, value)* function checks if the key is already in the hash table and if it is not, it will add the new key-value pair. If the key already exists in the hash table, the value will be overwritten with the new value. At best, the time complexity is O(1). At worst, the time complexity is O(n).

remove(ElasticERL, key)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. Similarly to add, removing from an AVL is O(logn) due to the binary structure of the tree. Any restructuring operations occurring after a deletion from the AVL tree will be either O(1) if only one rebalance occurs, or O(logn) if multiple rebalance operations occur.

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Thus, the *remove(key)* function is called from the myHashTable class. The remove(key) function searches for the key that the user wants to remove and if it exists, it will remove the key-value pair from the hash table and return the value of the key that was removed. Otherwise, it will return null. At best, the time complexity is O(1). At worst, the time complexity is O(n).

getValues(ElasticERL, key)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. The method will then utilize an internal method that will perform a binary search on the AVL tree ADT. Searching a binary tree is O(logn), therefore the time complexity for the method *getValues()* when the ADT is an AVL tree is O(logn) and the space complexity is O(n) (due to the space complexity of the AVL tree ADT).

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Thus, the *get(key)* function is called from the *myHashTable* class which returns the value(s) of the given key if the key exists. Otherwise, get(key) returns null. At best, the time complexity is O(1). At worst, the time complexity is O(n).

nextKey(ElasticERL, key)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. The next key of an AVL tree is the right child of the current node because it is the larger value of the parent and left child nodes. Thus, the method *nextKey()* utilizes an internal method of the ElasticERL class that performs a binary search for the given key, returns said node holding said key value, and

nextKey() returns the right child's key value of said node. The time complexity for a binary search is O(logn) and the time complexity for getting the right child is O(1), thus the time complexity of the function as a whole when only the AVL ADT is considered is O(logn).

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Thus, the *getNextKey(key)* function is called from the *myHashTable* class. The function gets all the keys, sorts them and returns the successor key of the key passed to the function *getNextKey(key)*. The time complexity of this is O(n).

prevKey(ElasticERL, key)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. The previous key to the key passed as the argument to the method is the parent of the node holding the key passed as the argument. Thus, *preKey()* makes use of an internal function of the ElasticAVL class called *getParent()*, which performs similarly to a binary search, but instead the loop breaks when either the left or right child's key value matches that of the key passed as an argument. As such, the method *prevKey()* returns the key returned from *getParent()* if it exists. As such, the method's time complexity is O(logn) and its space complexity is O(n) due to the binary tree.

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Thus, the *getPrevKey(key)* function is called from the *myHashTable* class. The function gets all the keys, sorts them and returns the predecessor key of the key passed to the function *getPrevKey(key)*. The time complexity of this is O(n).

rangeKey(ElasticERL, key1, key2)
If the size of the ElasticERL instance is less than 1000, then the method will consider only the AVL tree ADT. The time complexity of the function is O(n) because the function creates a sorted array by an inorder traversal of the AVL tree, which is a time complexity of O(n), then indexes the array with a loop to search for the first key and ends the loop when the second is found. Given that both the loop and the inorder traversal run in parallel, the entire function has a time complexity of O(n), and a space complexity of O(n) due to both the array and the tree structure.

If the size is greater than or equal to 1000, then the method will only consider the hash table object. Thus, the *getRangeOfKeys(k1, k2)* function is called from the *myHashTable* class. The function gets all the keys, sorts them and returns the number of keys (not inclusive) between the two keys passed to the function *getRangeOfKeys(k1, k2)*. The time complexity of this is O(n).

Design Decisions and Specification of the ElasticERL ADT

An AVL tree was determined for the ElasticERL of a smaller threshold size because the restructuring done when there is an imbalance in the AVL tree could cost more time and is more of a hassle when deleting and inserting into the ADT. Furthermore, it was chosen because its methods, namely for inserting, deleting and searching each have a time complexity of O(logn). Furthermore, it is fairly simple to determine the previous key of a given key, because it would simply be the parent of the given key, as well as the next key would be the largest of the given key, thus the right child of the given key. Both operations would therefore be O(logn) as well. Additionally, performing an inorder traversal of an AVL tree always yields a sorted sequence, and generating and determining a range of keys was also just a means of indexing the sorted sequence created by the inorder traversal and either counting the keys between the two passed keys (key1 and key2, inclusive) or generating a new random key between two keys of the sorted sequence, which at most would be O(n). Thus the AVL tree was chosen as the ADT for the ElasticERL for sizes less than 1000.

The hash table was chosen for files with entries greater than or equal to 1000 because of the possible benefit, in terms of time complexity, for the methods *add(ElasticERL, key, value)* and *remove(ElasticERL, key)*. As mentioned above, the time complexity for both the *add(ElasticERL, key, value)* and *remove(ElasticERL)* is O(1) for a hash table which is better than O(logn) for an AVL tree. At worst, the time complexity for adding and deleting key-value pairs is O(n) for a hash table which is still pretty good. Additionally, the time complexity for the *generate()* and *getValues(ElasticERL, key)* is at best O(1) and at worst O(n). The *SetEINThreshold(Size)* is O(1) for hash tables or AVL trees. The functions *allKeys(ElasticERL)*, *nextKey(ElasticERL, key)*, *prevKey(ElasticERL, key)*, *rangeKey(k1, k2)* have a time complexity of O(n) for a hash table.