Emmanuel BANDOLA SAYOTO – 40211918

**1.**

**a)**

Algorithm add(ElasticERL, key, value)

Input: ElasticERL object to add to, the int key, string value

Output: If the size reaches the threshold of the next ADT, output the swap

If (size is less than 100)

Add object array using implemented LinkedList

Increment size

Else if (size between 100 and 500000)

Add object array using implemented BST

Increment size

Else if (size is greater than 500000)

Add object array using implemented HashTable

Increment size

Algorithm SetEINThreshold(size)

Input: The integer size

Output: Depending on the input size, the ADT will be a certain type.

If (size between 0 and 100)

List is an implemented LinkedList

If (size between 100 and 500000)

List is an implemented BST

If (size more than 500000)

List is an implemented HashTable

Algorithm remove(ElasticERL, key)

Input: ElasticERL object to remove from, the int key if the entry

Output: If the size reaches the threshold of the previous ADT, output the swap

If (size is less than 100)

Remove object array using implemented LinkedList

Decrement size

Else if (size between 100 and 500000)

Remove object array using implemented BST

Decrement size

Else if (size is greater than 500000)

Remove object array using implemented HashTable

Decrement size

Algorithm nextKey(ElasticERL, key)

Input: ElasticERL to search through, the int key to find the following key

Create int k that will hold the following key regardless of the ADT type

If (size is less than 100)

Find following key in the LinkedList

k <- following key

Else if (size between 100 and 500000)

Find following key in the BST

k <- following key

Else if (size is greater than 500000)

Find following key in the HashTable

k <- following key

Return k

**3.**

**Time:**

LinkedList:

* Add: O(1), adding to the end of the LinkedList is constant time due to tail pointers.
* SetEINThreshold: O(n), as if we reach a threshold, we must copy all the values to the next ADT one by one
* Remove: O(n), we must traverse the LinkedList node by node til we find the one to remove.
* NextKey: O(n) we must traverse the LinkedList node by node til we find the node that we want to find to following node key.

Binary Search Tree:

* Add: O(n) as you may have to traverse all the way down the height of the tree to add the new value.
* SetEINThreshold: O(n), as if we reach a threshold, we must copy all the values to the next ADT one by one
* Remove: O(n), in the case of a skewed tree where you want to remove the largest/smallest element, you need to traverse the entire tree height.
* NextKey: O(n), in the worse case, we need to traverse all the way down the tree height-1 and get the following node element.

HashTable (chaining):

* Add: O(n), in the worst case, all the elements are chained in the same bucket and have to be traversed. But since we use the modulo (key % size) operation to determine the bucket, and the key is generated randomly, then having many values in the same bucket will not be as common, and on average will be O(1).
* SetEINThreshold: O(n), as we have to traverse through every bucket.
* Remove: O(n), Similar to add(), in the worst case all elements are in the same bucket and you must traverse them to find the element to remove. On average O(1).
* NextKey: O(n), Average O(1), for same reasons as Add() and Remove().

**Space:**

Small ADT: I chose LinkedList as insertion and deletion are very fast (O(1)). While the size of the LinkedList is O(n), the size of the LinkedList will always be small (<100), therefore it isn’t too costly.

Medium ADT: I chose Binary Search Trees due to its fast traversal speed (on average) at O(log n).

Large ADT: I chose HashTable, as it is very fast at accessing, inserting, and deleting elements, all averaging at O(log n). Space complexity is also O(n) for the number of entries, which is decent given the performance of the ADT.