Lab 2 – COMP ENG 2SI4

Abilash Logeswaran

Start Date: Feb/3/2015

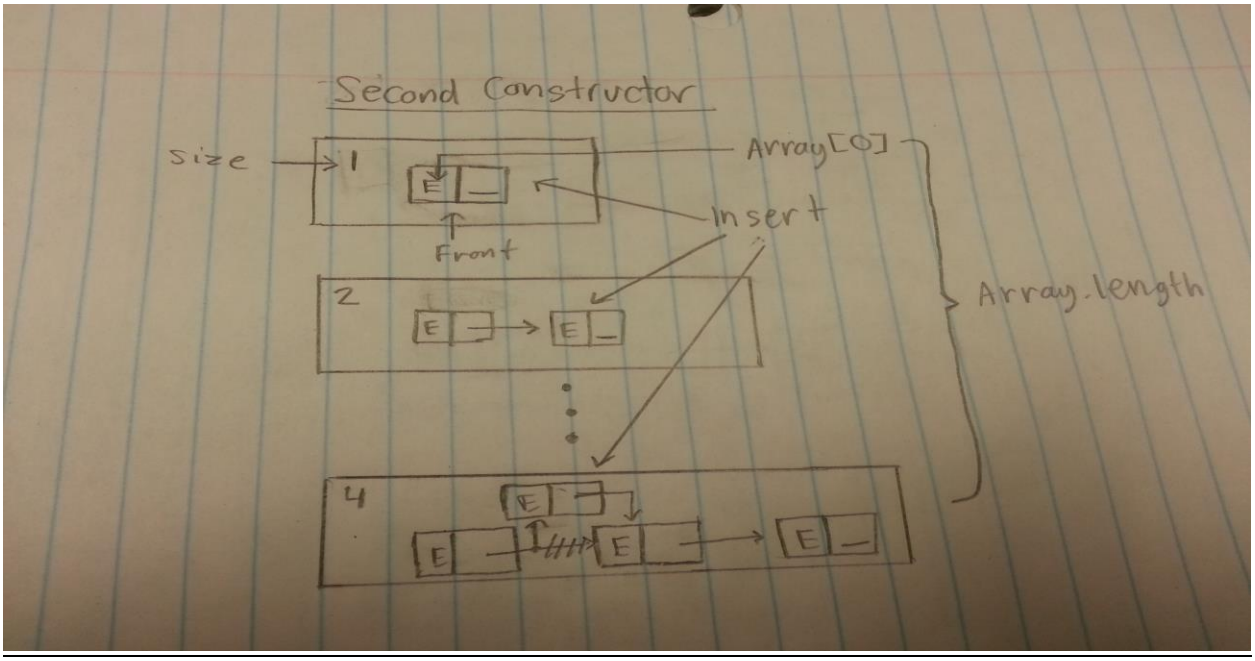
Halfway Point: Feb/4/2015

Completion: Feb/5/2015

Insert():

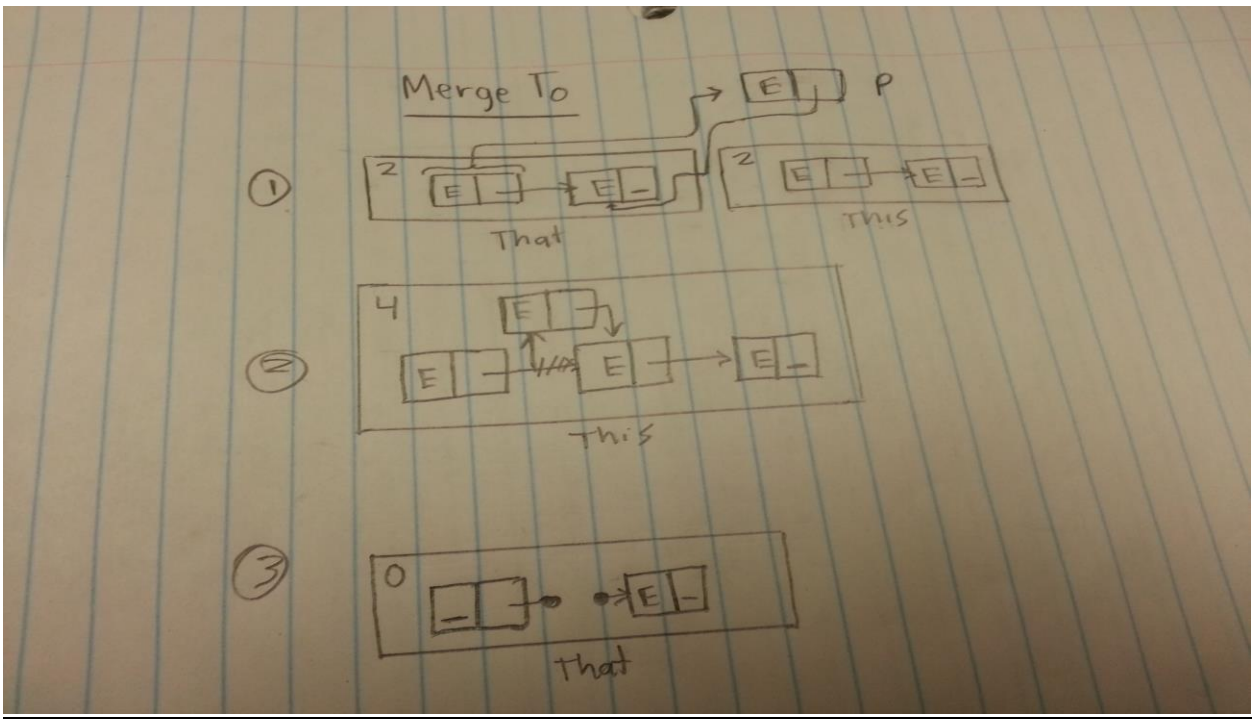
The insert method works according to the following algorithm. First the entry being inserted cannot be present in the list already, so using the find() method this is evaluated. The second condition that splits the algorithm into 2 paths is whether the array being inserted into has any other entries; if not, then the word is simply inserted as the first entry. This sets the front node to contain the word as well as its own address as it's the only word in the linked list, and also size is incremented by 1. If size>0 and the word does not exist in the linked list, it must be compared to the other words in order to determine its alphabetic placement in the list. In order to compare the word values at each node, a temporary variable is set to equal the front node, which by extension contains the addresses of all other nodes, and each iteration of the loop sets the value of this temp node to its "next" field so it is now the next node and this loop ends when the next value is null indicating the end of the list. Using this observation, a for loop is set up wherein the word at each node is compared to newword with the compareTo() method. This again forks the path. If newword comes before the first word in the list, it becomes the new head node and takes the address of the old head as the next address. If newword comes before another word in the list, that node reference is reassigned to a new node that contains newword and its prior node value so no information is lost. Lastly if newword is alphabetically lower than all the words in the list, it must be added at the end, and this is known because the check variable maintains its original value of 0 (thus newword has not been inserted yet). Adding this word to the end of the list is simple: the temp variable references the last word in the list, and now it is simply assigned as a new node containing newword and its next field as null. The running time for this method is simply Big Theta (n) where n is the size of the linked list. This is because there are 2 for loops (with running time n) evaluated in succession, which is the find() method which loops through the list looking for an entry and the for loop that compares the alphabetic order. When summed they equal 2n, from which the constant can be dropped and thus a final worst case running time of n.

Second Constructor:



The second constructor works according to the following algorithm. It takes as input an array of words. It runs through a for loop, which runs according to the size (number of words) in the array and at each iteration it uses insert() to insert the corresponding word in the array into the Object. The mechanisms in insert() ensure that duplicates are not inserted.

MergeTo():



The mergeTo method works according to the following algorithm. Its purpose is to insert words from that object into this object provided no duplicate exists and in the end remove all words from that object. The insert works as expected, wherein a temporary variable is set to reference the head node of that object (and by extension all nodes) and through a while loop (which follows the same condition as the for loop mentioned in the insert() method) each word in that object is inserted into this object using the insert() method. The insert() method once again ensures no duplicates exist and places the words in the correct alphabetical order. The removal of the words, I chose to approach in a novel way. Instead of using a for loop with the remove method (which would increase running time), I chose to set the head node to null. Since this is a singly linked list if a node is deleted (set to null) all other nodes after it are lost because the chain of references is gone since they only flow in one direction. Since size is not updated automatically after this deletion I had to manually set the size of the object to 0, so that the toString() method does not display a null pointer error. I also followed up with a manual call of java's garbage collector so that the lost chain of nodes can be de-allocated from memory. The additional memory usage for this method is Big Theta (1) because only the temporary node is declared each time. The run time is Big Theta (n1*n2) because the loop runs n2 times for each word in that object, whereas the body contains the insert() method which has a worst case run of n1. There is no possible further simplification because in order to insert words into this object, the insert() method must be called which itself is Big Theta (n1) and since it must add every word in that object, it has to run n1*n2 times in a worst case scenario. An alternate way to reduce run time would have been to add another instance field which contained a regular string array of the words in each object and then using the ArrayUtils.addAll(..) API method to concatenate them into a single array and then finally create a new object with this array as input to constructor 2, and then reference this object to equal the new object. This was not allowed because no other Java API methods/classes were allowed and this would not have resulted in a constant additional memory usage.