The insert(String newword) method is used to insert the passed word, newword, into the list at the correct alphabetical position. This first part of this algorithm calls the find function to see if the word is currently in the list, and if it is, will exit the method. If the word is not found in the list then the word will be inserted into the list. The first condition that is checked is if the list is empty. If this is true then a new node is created that points to null and stores the passed word. The head is then assigned to point to the new node, the list size is increased and the method is exited. If the list is not empty then the second part of the main algorithm is executed. This starts by creating two reference nodes, one points to the first node in the list, p, and the second points to the node after, q. There are three possible cases that can happen, the first being that the word will be inserted at the beginning of the list. The second being the word will be inserted between two words in the list. The third being the word is inserted at the end of the list. Since the words are stored using linked list, these three are executed from first to third. The first word of the list is compared to the passed word using the compareTo() function, and if it is found to be alphabetically less than the first word of the list, it is to be inserted at the beginning of the list. This is done be creating a new node that points to the node p, which is currently the first node in the list, and then setting the head to point to the new node. The size of the list is incremented and then the method exits. The second possibility is if the word needs to be inserted between two words. This is done by comparing the newword, with two words in the list which are next to each other. The way this is done is by looping through the list until the second reference node is equal to null. In this loop the compareTo() function is used to compare the two. If it is found that the return values of these two function calls differ by sign, the word will be inserted between these two nodes. This is done be creating a new node that points to the second reference node, q, which holds the passed word, newword. The reference node p is then set to point to the new node. The list size is then incremented and the method exits. If either of these two possibilities did not occur this means that the word is to be inserted at the end of the list. Since the two reference nodes p and q are now pointing to the end of the list and null respectively, a new node is created that points to null, and contains the passed word, newword, and the node p is set to point to this new node. The list is incremented and the method exits.

The mergeTo(WordLinkedList that) function is used to merge two lists together. Specifically will merge the words from that list, to this list. This method has four different possible scenarios that can happen depending on the size of the lists. Out of the four, three are easily handled. Two of the three are if this.listsize is zero and that.listsize can either be zero or non-zero. When this happens this lists head node is set to equal the node of that list, as well as this list size being set equal to that list size, which does not depend on that lists' size. The other easy case is if that list size is zero. Nothing will be executed because there are no words to be merged into this list, and that list is empty, which has a size of zero. The fourth possibility is that both lists contain words. The main idea with this case is that both lists are sorted alphabetically, which helps to decrease the time to insert the words in the correct position in the list. When this fourth case is the case that is executed the first part of the algorithm creates three nodes: the first node, p, which is set to the initial node of the passes list, and is used to loop through said list, the second and third nodes, a and b, respectively are set to equal the first and second node in this list, and are used to compare the current word in that list with the words in this list. After these nodes are created three different conditions are checked. The first is if the first word in that list needs to be inserted at the beginning of this list. If so, a new node is created which points to the node a, and contains the word being merged into this list. After this, the head of this list points to the new node, node b is set to a, and a is set to the new node. This condition is only checked once as a result of the lists being sorted. The second condition checked is if this list has a size of one. This needs to be checked

because if the list has a size of one, the node b is null which causes error further in the algorithm. Three cases can happen. The first being that the word needs to be inserted at the beginning of this list. This case is handled by checking to see if the word needs to be inserted at the beginning of the list. The second and third case are related to each other. These happen if the first case did not meaning that the word that is being inserted equals the word in the list or it needs to be inserted after the list. If the word that is being inserted in the list equals the word in the list, then since the lists are sorted and there are no duplicates of the words in the list, the node that is point to the current node which word is already in the list is set to point to the next node in the list, and if the node is not null, the second word in that list will be inserted into the list. The last condition of the three is if the first to words are equal to each other. This is checked here because the only node that node b will not point to is the first node, this check will make it easier when looping through that list. If this condition is true, then node p is set to equal the next node in the list. After these three conditions are checked, the main loop to go through that list is executed only if there are words left in the list, or p is not null. This loop has three main cases it handles. The first is if the word is to be inserted at the end of the list. If node b is the last node in the list, and when comparing the two words using the compareTo() it is found that, that word is to be inserted at the end of this list, a new node is created that points to null, and contains the word to be inserted. Node b is set to point to the new node, the list size is increased, and node p is set to its next node. The second case is if the word is to be inserted between two nodes a and b. If this is found to be true then a new node is created which points to b and contains the word to be inserted in the list. Node a points to the new node, the list size is increased, and node p points to the next node in the list. The third case is if either of the two cases did not execute. This means that the word is not to be inserted by the current node positions of a and b. When this happens the word at node p in that list is compared to the word at node b in this list, to see if they are equal. If this true, then the word at node p will not be inserted in the list, by setting node p to equal the next node in the list. After this check is completed, node a is set to b, and node b is set to the next in the list. This loop is executed until all the words in that list are either merged into this list or ignored if they are already in this list. After this loop is executed, the head node in that list is set to point to null, and the list size of that list is set to zero.
Consider the two lists, this: bear, dog, fun and list that: apple, bear, great.

S**tage 1: Intital call to merge()** this.listsize = 3, that.listsize = 3

$$\begin{tikzcd}
\boxed{\text{This.head}} \arrow[r] & \boxed{\text{a = bear}} \arrow[r] & \boxed{\text{b = dog}} \arrow[r] & \boxed{\text{fun}} \arrow[r] & \boxed{\text{null}} \\
\boxed{\text{That.head}} \arrow[r] & \boxed{\text{p = apple}} \arrow[r] & \boxed{\text{bear}} \arrow[r] & \boxed{\text{great}} \arrow[r] & \boxed{\text{null}}
\end{tikzcd}$$

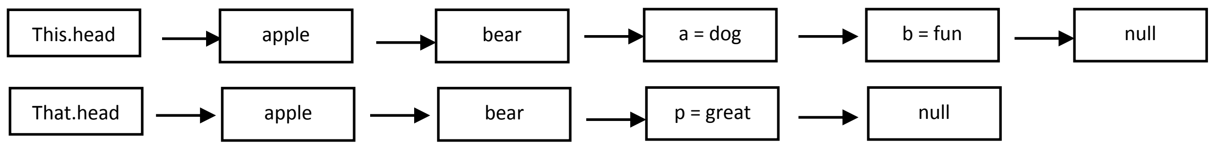
Stage 2: After first word in that gets inserted into this this.listsize = 4, that.listsize = 3

$$\begin{tikzcd}
\boxed{\text{This.head}} \arrow[r] & \boxed{\text{a = apple}} \arrow[r] & \boxed{\text{b = bear}} \arrow[r] & \boxed{\text{dog}} \arrow[r] & \boxed{\text{fun}} \arrow[r] & \boxed{\text{null}} \\
\boxed{\text{That.head}} \arrow[r] & \boxed{\text{apple}} \arrow[r] & \boxed{\text{p = bear}} \arrow[r] & \boxed{\text{great}} \arrow[r] & \boxed{\text{null}}
\end{tikzcd}$$

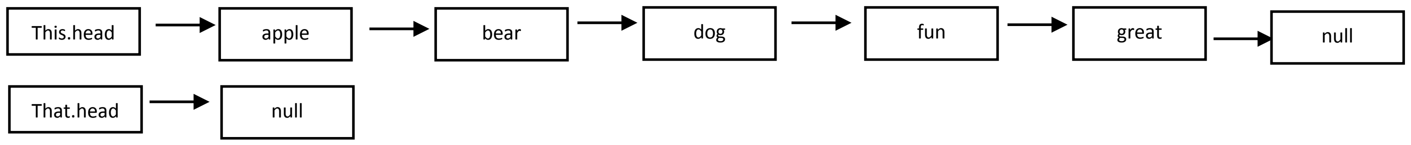
Stage 3: After second word gets checked this.listsize = 4, that.listsize = 3

$$\begin{tikzcd}
\boxed{\text{This.head}} \arrow[r] & \boxed{\text{apple}} \arrow[r] & \boxed{\text{a = bear}} \arrow[r] & \boxed{\text{b = dog}} \arrow[r] & \boxed{\text{fun}} \arrow[r] & \boxed{\text{null}} \\
\boxed{\text{That.head}} \arrow[r] & \boxed{\text{apple}} \arrow[r] & \boxed{\text{bear}} \arrow[r] & \boxed{\text{p = great}} \arrow[r] & \boxed{\text{null}}
\end{tikzcd}$$

<u>Stage 4: After third word is compared to a and b</u> this.listsize = 4, that.listsize = 3;



<u>Stage 5: After third word is inserted at the end of list</u> this.listsize = 5, that.listsize = 0



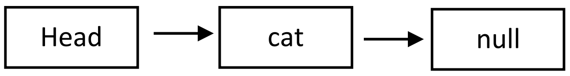
When analyzing the algorithm, the worst case of additional memory used is $\Theta(n_2)$, where n_2 is the size of that list. This is reached if every word in that list is inserted into this list.

When analyzing the algorithm, the worst case run time is $\Theta(n_1 + n_2)$, where n_1 is the size of this list and n_2 is the size of that list. The run time of the algorithm before the while loop is a constant, which does not affect the time with Θ notaion. The while loop will be executed at max n_2 times and inside the loop, the if statements will only execute once, for a maximum of n_1 throughout the whole execution. Then the total worst case run time is $\Theta(n_1 + n_2)$.

The second constructor is called if an array of words is passed. When this is called an integer that stores the length of the array is created. This is used to loop through the array to insert the words into the list. A new node is created that contains the first word in the list and which points to null, since it is the only word in the list. The list size is increased and the head node points to the new node. After this a loop is executed which loops from the second word in the array through until the end of the array. This will only be executed if the array size is greater than one. In the main body of the loop the insert() method is called and passes the current string in the array, which will insert the word into the correct position in the array, and if inserted will increase the list size. After all the words have been passed to the insert method the constructor exits. Consider calling the constructor with the string, cat, apple, bear.

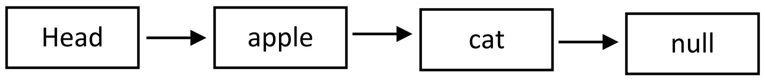
Stage 1: Initial call to constructor

listsize = 1 ;



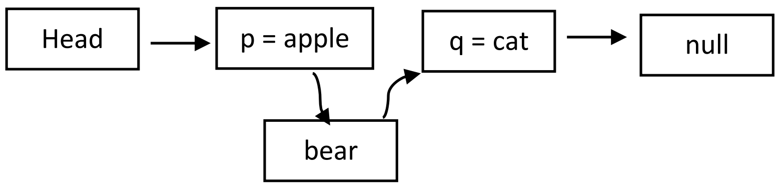
Stage 2: After first completion of the for loop

listsize = 2; j = 1;



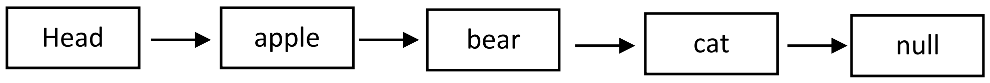
Stage 3: Through second iteration of the for loop, in the insert method

listSize = 2; j = 2;



Stage 4: After constructor exits

listSize = 3;


Head
apple
bear
cat
null