Input Values	Expected Output	Actual = Expected?
10 random values in head	Random values ordered ascending (for fun)	Yes
Head and tail NULL, empty lists	Nothing	Yes
Head is 8 3 5 6, tail NULL	3 5 6 8 (baseline test)	Yes
Head is 2 1 0, tail NULL	0 1 2 (testing odd # input)	Yes
Head is 0 2 1 0, tail NULL	0 0 1 2 (testing repeats)	Yes
Head is values -1 through -10, in order	-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 (testing negative numbers)	Yes
Head is 1 2, tail is 30x the value 1	1 2 (ensuring that tail can be non-Null and the sort will function)	Yes
Head is 10 random numbers between -120 and 120	Random values ordered ascending (ensuring random and negative work together)	Yes
Head is random numbers with alternating sign	Random values ordered ascending (testing random and non-random values next to each other of unknowns size)	Yes
Head is the letter a	Some sort of code death (checking for alpha responses)	Yes

Analysis of regMergeSort:

Test Case #1:

We begin with the list head containing 6 5 3 8.

recMergeSort begins by allocating memory for a pointer to *otherHead, then checks if *head or (*head)->next are null, which they are not. Then recMergeSort calls divideList, passing head as 6 5 3 8 and otherHead as empty.

divideList begins by allocating memory for a middle and current list, then checks if the argument first1—in this case head—is NULL; if so it sets first2—otherHead—to be NULL as well. Even if the condition is not met the same thing happens if first1->next is NULL. As neither are true, first2 is not set to be NULL.

Otherwise, middle is set to be first1, containing 6 5 3 8. current is then set to 5 3 8, or first1->next.

As current is not NULL, current is again set to current->next, or 3 8.

Then, while current is not NULL, middle is set to middle->next, which is 5 3 8. current is then set to current->next to give 8. As this is not NULL, current is then set to NULL.

current is iterating through head twice as fast as middle, making middle the second half of head. This process continues, until current becomes NULL and the while loop exits.

After the while loop, *first2 is set to be middle->next; this sets otherHead to be the second half of the list. middle->next is then set to NULL, after which head has been set to 6 5, and otherHead to be 3 8.

We then return to the first call of recMergeSort. At this point, recMergeSort is called with head as an argument.

At this second level of recursion, another otherHead is created, then head and head->next are checked to not be null (which they are not as head is 6 5 3 8). After this point, divideList is called again with *head and &otherHead as arguments.

divideList, through a similar process detailed above, sets head to be 6 5 and otherHead to contain 3 8. We then return to the second level recursion of recMergeSort with 6 5 3 8 as the argument, and call recMergeSort with head (6 5) as the argument.

At a third level of recurion, an otherHead is created, head and head->next are checked to not be NULL and divdeList is called with head (6 5) and otherHead (empty) as arugments.

After divideList is called, head contains 6 and otherHead contains 5. At this point, recMergeSort is called with head (6) as an argument.

In this fourth level of recursion, as head->next == NULL, recMergeSort exits immediately, and the same occurs when it is next called by the third level of recursion with the arugment otherHead (5).

In the third level of recurision, *head is then set to be the return value of mergeList with arguments *head and otherHead.

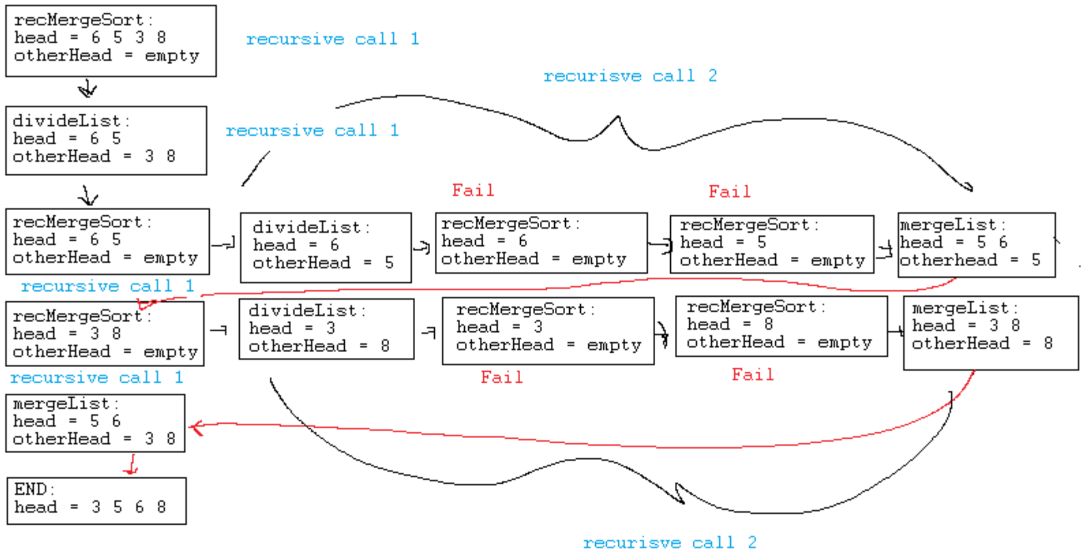
mergeList creates a pointer *lastSmall and *newHead, then checks if either of its arguments are null; if they are, the other list is returned immediately. As this is not the case, the flow proceeds.

The first node in head is then compared with the first node in otherHead; as 6 > 5, the if statement is not executed and instead the else statement is.

This statement sets newHead to otherHead, sets first2 equal to first2->next (which is NULL), and then sets lastSmall to be newHead.

As first2 is now NULL, the while statement does not execute. The next if statement does not execute as first1 is not NULL. lastSmall then has head appended to it, giving it the value 5 6. newHead is then returned, giving the third level recursion’s head list the contents 5 6.

We then regress to the second level of recursion, where recMergeSort is called with the second level otherHead list 3 8.



divideList fills this third level recursion’s head value with 3 and its otherHead value with 8. recMergeSort is called with both head and otherHead as arguments, failing both times as the next value in both arguments is NULL.

mergeList is then called; in the same process as above it fills head with 3 8, meaning the second level otherHead is set to 3 8 as well.

As both of the second recursion level’s recMergeSort functions have executed, head is then set to the return value of mergeList is then called with the arguments head and otherHead, containing 5 6 and 3 8 respectively.

mergeList creates pointers for lastSmall and newHead, then as first1 and first2 are not null moves into its first else statement.

fist1->info is the value 6 and first2->info is the value 3, and as 5 > 3 the else statement is executed instead of the if statement.

This else fills newHead with otherHead, the values 3 8, then sets first2=first2->next, meaning first2 now has the value 8.

The flow then moves into a while loop; the while loop’s conditions are not met, so its first if statement is checked. This time 5 < 8, so it executes instead of the else statement.

It sets lastSmall->next to be first1, so lastSmall contains 3 5 6. lastSmall is then set to be lastSmall->next, so lastSmall contains 5 6. Finally, first1 is set to be first1->next, so first1 contains 6.

The while loop is checked, and its conditions are not met, so first1->info is compared with first2->info. As 6 < 8, the if statement is executed.

It sets lastSmall->next to be first1, so lastSmall contains 5 6. lastSmall is then set to be lastSmall->next, so lastSmall contains 6. Finally, first1 is set to be first1->next, so first1 contains NULL.

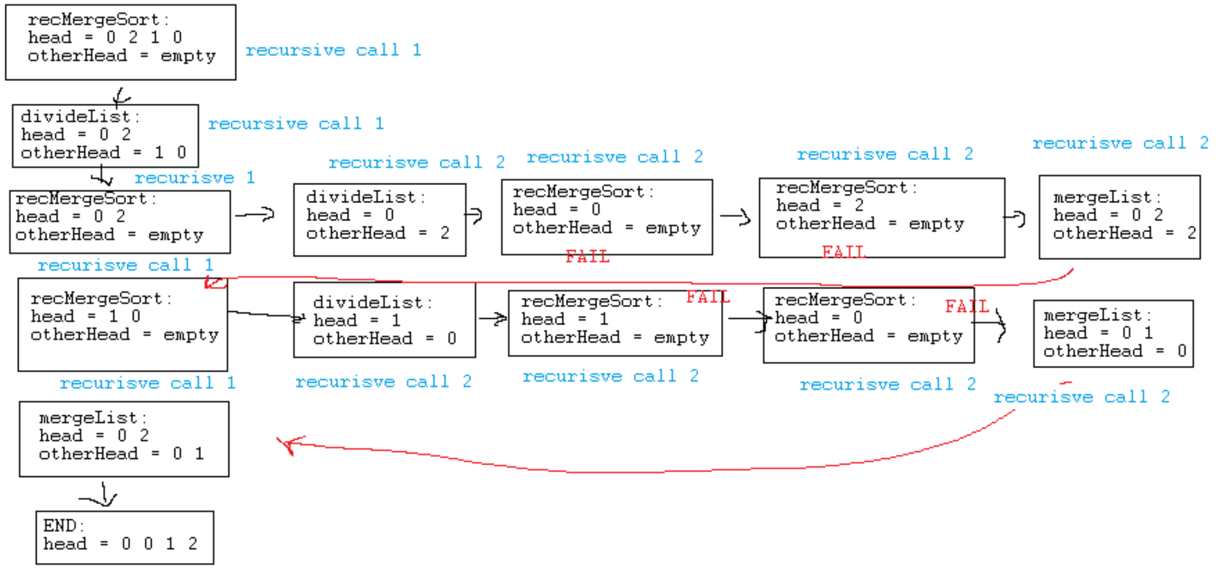
The while loop's conditions are met, so the next if statement is checked; first1 is indeed null so lastSmall->next is set to 8.

Finally, newHead with the values 3 5 6 8 is returned.

Test Case #2:

head is the values 0 2 1 0. otherHead is created empty, then passed to divideList with head. divideList creates middle = 0 2 1 0, and current = 2 1 0. As current is not null, current = 1 0. While current is not null, first middle = 2 1 0, and then current = 0; if current is not null, it is then = current->next = NULL.

The while loop is then checked; as its conditions are met, it ends, and otherHead is set to be 1 0, while head is set to be 0 2.



The program behaves very similarly to test case one; next the list otherHead, containing 1 0, is divided into 1 and 0, then recMergeSort is attempted upon these one node lists and fails. mergeList is called with arguments 1 and 0, and results in 0 1.

The same happens to head, where its values of 0 2 are put into recMergeSort and then divideList, where its otherHead is set to 2 and head is set to 0. recMergeSort is attempted on these and fails, when mergeList is called with arguments 0 and 2, returning 0 2.

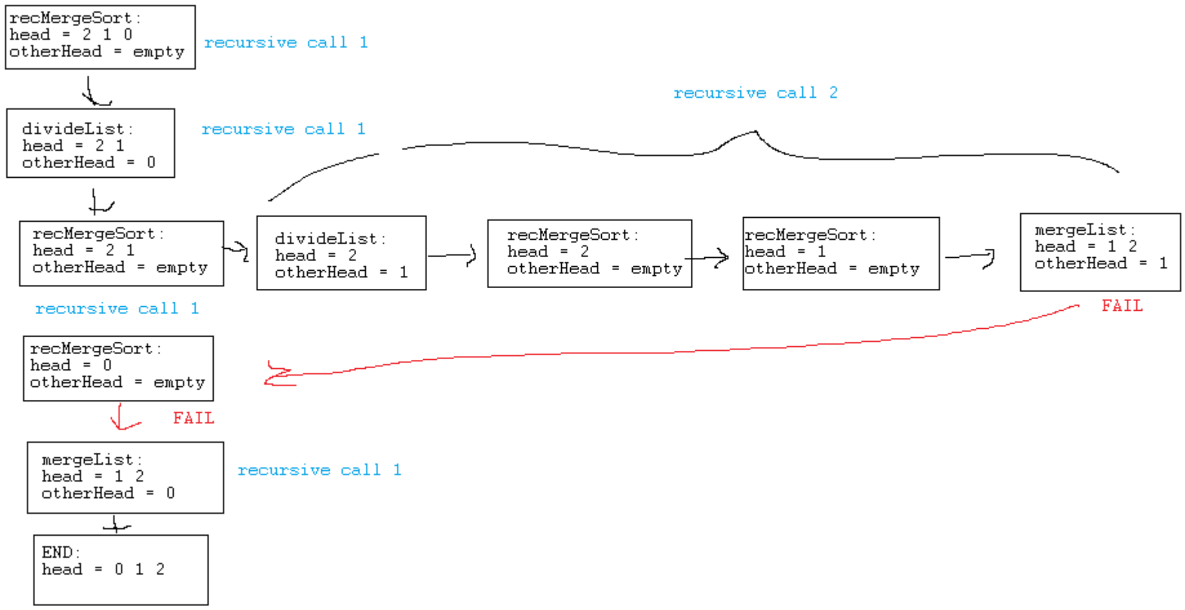
However, the crux of this test case comes at this point, when recMergeSort has been carried out on the original, first-recursion head and otherHead. mergeList is called with head=0 2 and otherHead = 0 1.

At first, first1->info is not less than first2->info as they are equal, so the else loop is executed, with newHead being set to 0 1. first2 is then set to 1, and lastSmall is set to the newHead value of 0 1.

Then, as neither first1 nor first2 are NULL, and first1->info is not less than first2->info, this process repeats until we are left with head equal to 0 0 1 2.

Test Case #3:

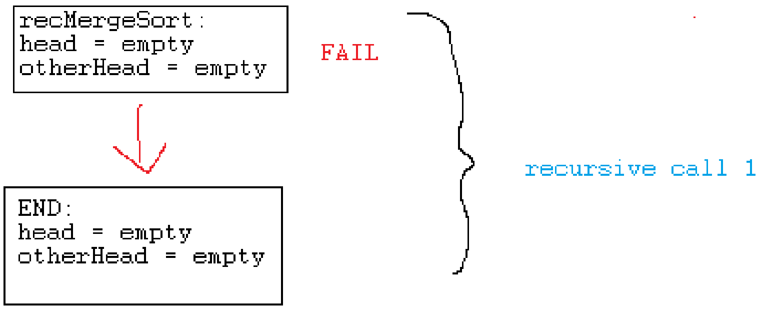
With three nodes, things become slightly more complicated. Head = 2 1 0. The first recursive call behaves normally, however, when asked to split the three node list in half we get an interesting result.



Passed to the function divideList, first1 = 2 1 0 and first2 is empty. Middle = 2 1 0, and current = 1 0. As 1 0 is not NULL, current then becomes 0 and we move into the while loop. However, when we are in the while loop, middle becomes 1 0 and current becomes NULL before the if statement, and first2 becomes middle->next which is 0. We then have two sublists of differing size; one fails immediately as it is only one element, and the other is parsed, sorted and returned as normal. When both these tasks are done, mergeList merges them to get the result 0 1 2.

Test Case #4:

We begin with head = NULL. otherHead is created, but the first if statement is not satisfied, so the function ends without performing any other actions.


recMergeSort:
head = empty
otherHead = empty
FAIL
END:
head = empty
otherHead = empty
recursive call 1