

Final Exam

Input Values	Expected Output	Did Actual Meet Expected?
Empty List, Head is Null	Nothing (An Empty List)	Yes
One Integer, One Node in List	The Inputted Integer of the Linked List (One Integer)	Yes
Two Integers, Two Nodes in List	The Sorted Ascending Values of the Linked List (Two Integers)	Yes
Three Integers, Three Nodes in List	The Sorted Ascending Values of the Linked List (Three Integers)	Yes
Four Integers, Four Nodes in List	The Sorted Ascending Values of the Linked List (Four Integers)	Yes
Five Integers, Five Nodes in List	The Sorted Ascending Values of the Linked List (Five Integers)	Yes
Six Integers, Six Nodes in List	The Sorted Ascending Values of the Linked List (Six Integers)	Yes
Seven Integers, Seven Nodes in List	The Sorted Ascending Values of the Linked List (Seven Integers)	Yes
Eight Integers, Eight Nodes in List	The Sorted Ascending Values of the Linked List (Eight Integers)	Yes
Nine Integers, Nine Nodes in List	The Sorted Ascending Values of the Linked List (Nine Integers)	Yes

Test Case #1: (Empty List)

```

1 void recMergeSort(struct node** head){
2   struct node* otherHead;
3   if(*head != NULL) //if the list is not empty
4     if((*head) -> next != NULL){ //if list has more than one
5       divideList(*head, &otherHead);
6       recMergeSort(head);
7       recMergeSort(&otherHead);
8       *head = mergeList(*head, otherHead);
9     }
10 } //end recMergeSort

```

1. Declares the type and name of the function, void recMergeSort, and the parameters of the function, struct node ** head, where head is the head of an empty list where the head pointer points to a null pointer, that has been passed to this function.
2. Declares a new variable, struct node * otherHead.
3. Declares an if statement with the condition when the head pointer of the passed linked list is not equal to a null pointer, which is declared false in this case, because in an empty list, the head pointer is pointing to a null pointer.
4. Declares an if statement with the condition of what the next pointer points to that the head pointer points to of the linked list being pass is not equal to a null pointer, which has been skipped in this case, because the if statement that this line of code is a part of was declared false of line 3.
5. A function call to the function named dividedList, that has been skipped, because the if statement that contains the if statement that this line of code is a part of was declared false of line 3.
6. A function call to the function it is in named recMergeSort, that has been skipped, because the if statement that contains the if statement that this line of code is a part of was declared false of line 3.
7. A function call to the function it is in named recMergeSort, that has been skipped, because the if statement that contains the if statement that this line of code is a part of was declared false of line 3.
8. A function call to the function named mergeList that has been also assigned to the head pointer, that has been skipped, because the if statement that contains the if statement that this line of code is a part of was declared false of line 3.
9. Ends the declarations of the if statement with the condition when the next pointer the head pointer of the linked list being pass is not equal to a null pointer of line 4 in this case.
10. Ends the functionality of the function recMergeSort.

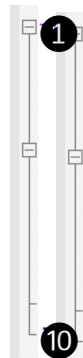
Test Case #2: (One-Node List)

```

1 void recMergeSort(struct node** head){
2     struct node* otherHead;
3     if(*head != NULL) //if the list is not empty
4         if((*head) -> next != NULL){ //if list has more than one
5             divideList(*head, &otherHead);
6             recMergeSort(head);
7             recMergeSort(&otherHead);
8             *head = mergeList(*head, otherHead);
9         }
10 } //end recMergeSort

```

1. Declares the type and name of the function, void recMergeSort, and the parameters of the function, struct node ** head, where head is the head of a linked list containing one node that holds an integer value where the single node points to a null pointer, that has been passed to this function.
2. Declares a new variable, struct node * otherHead.
3. Declares an if statement with the condition when the head pointer of the passed linked list is not equal to a null pointer, which is declared true in this case, because the head pointer of a one-node linked list is not pointing to a null pointer, whereas it is pointing to a node that contains information and a next pointer.
4. Declares an if statement with the condition of what the next pointer points to that the head pointer points to of the linked list being pass is not equal to a null pointer, which has been declared false, because the next pointer points to that the head pointer points to does equal to a null pointer in this case of a one-node linked list.
5. A function call to the function named dividedList, that has been skipped, because the if statement that this line of code is a part of was declared false of line 4.
6. A function call to the function it is in named recMergeSort, that has been skipped, because the if statement that this line of code is a part of was declared false of line 4.
7. A function call to the function it is in named recMergeSort, that has been skipped, because the if statement that this line of code is a part of was declared false of line 4.
8. A function call to the function named mergeList that has been also assigned to the head pointer, that has been skipped, because the if statement that this line of code is a part of was declared false of line 4.
9. Ends the declarations of the if statement with the condition when the next pointer the head pointer of the linked list being pass is not equal to a null pointer of line 4 in this case.
10. Ends the functionality of the function recMergeSort.

Test Case #3: (Two-Node List)


```

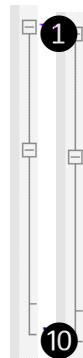
1 void recMergeSort(struct node** head){
2     struct node* otherHead;
3     if(*head != NULL) //if the list is not empty
4         if((*head) -> next != NULL){ //if list has more than one
5             divideList(*head, &otherHead);
6             recMergeSort(head);
7             recMergeSort(&otherHead);
8             *head = mergeList(*head, otherHead);
9         }
10 } //end recMergeSort

```

1. Declares the type and name of the function, void recMergeSort, and the parameters of the function, struct node ** head, where head is the head of a linked list containing two nodes which each hold an integer value where the second node of the list points to a null pointer, that has been passed to this function.
2. Declares a new variable, struct node * otherHead.
3. Declares an if statement with the condition when the head pointer of the passed linked list is not equal to a null pointer, which is declared true in this case, because the head pointer of a two-node linked list is not pointing to a null pointer, whereas it is pointing to a node that contains information and a next pointer.
4. Declares an if statement with the condition of what the next pointer points to that the head pointer points to of the linked list being pass is not equal to a null pointer, which has been declared true, because the next pointer points to that the head pointer points to does not equal to a null pointer, whereas in a two-node linked list, it points to another node that contains information and a next pointer.
5. A function call to the function named dividedList, where the information of the head pointer that creates a linked list and the address of the otherHead that is currently equal to nothing is passed. The function dividedList also creates two more struct node* variables named middle and current along with the parameters of the function in order to use as a part of its algorithm. The current variable is programmed to go through the list by being set to every other node in the linked list until it gets to the end where it points to a null pointer, and the variable middle goes through with the same number of iterations but goes through the list without skipping nodes of the linked list. Once the current pointer get to the end of the list, the middle pointer should be at somewhat of the middle of the list, where the list will be then divided of that location; giving a divided list. The otherHead pointer is then updated to contain the information of the second half of the linked list, and the head pointer contains the first half. In this case with a two-node linked list, the two nodes are processed as the first half and a null pointer is the second half.
6. A function call to the function it is in named recMergeSort, where the information that remains of the head is passed to the function which is the same function that this line of code is a part of in order to update the information being used again.
7. A function call to the function it is in named recMergeSort, which has been technically skipped, because the address of otherHead points to a null pointer, where the function the suppose information is being pass to would have nothing to work with.
8. A function call to the function named mergeList that has been also assigned to the head pointer, where the information of the head pointer and the otherHead are compared, and are merged back

to a linked list with the initial length but sorted in ascending order. The function creates two struct node * variables named lastSmall and newHead along with the parameters of the function in order to use as a part of its algorithm. The values of the first nodes are first compared depending if the first value is larger than the second, and if so, the nodes are switched as they merge, but if not, they are left to be as they are merged into back as a linked list. But since the linked list in this case only contains two nodes, it doesn't run through all the loops and statements stated in the function in order to create the sorted linked list in this case.

9. Ends the declarations of the if statement with the condition when the next pointer the head pointer of the linked list being pass is not equal to a null pointer of line 4 in this case.
10. Ends the functionality of the function recMergeSort.

Test Case #4: (Three-Node List)


```

1 void recMergeSort(struct node** head){
2     struct node* otherHead;
3     if(*head != NULL) //if the list is not empty
4         if((*head) -> next != NULL){ //if list has more than one
5             divideList(*head, &otherHead);
6             recMergeSort(head);
7             recMergeSort(&otherHead);
8             *head = mergeList(*head, otherHead);
9         }
10 } //end recMergeSort

```

1. Declares the type and name of the function, void recMergeSort, and the parameters of the function, struct node ** head, where head is the head of a linked list that contains three nodes that each hold an integer value where the third node of the linked list points to a null pointer, that is passed to this function.
2. Declares a new variable, struct node * otherHead.
3. Declares an if statement with the condition when the head pointer of the passed linked list is not equal to a null pointer, which is declared true in this case, because the head pointer of a three-node linked list is not pointing to a null pointer, whereas it is pointing to a node that contains information and a next pointer.
4. Declares an if statement with the condition of what the next pointer points to that the head pointer points to of the linked list being pass is not equal to a null pointer, which has been declared true, because the next pointer points to that the head pointer points to does not equal to a null pointer, whereas in a three-node linked list, it points to another node that contains information and a next pointer.
5. A function call to the function named dividedList, where the information of the head pointer that creates a linked list and the address of the otherHead that is currently equal to nothing is passed. The function dividedList also creates two more struct node* variables named middle and current along with the parameters of the function in order to use as a part of its algorithm. The current variable is programmed to go through the list by being set to every other node in the linked list until it gets to the end where it points to a null pointer, and the variable middle goes through with the same number of iterations but goes through the list without skipping nodes of the linked list. Once the current pointer get to the end of the list, the middle pointer should be at somewhat of the middle of the list, where the list will be then divided of that location; giving a divided list. The otherHead pointer is then updated to contain the information of the second half of the linked list, and the head pointer contains the first half. In this case with a three-node linked list, the first two nodes are processed as the first half and the third node is the second half. With this initial three-node linked list, the first half will go through the process again, in order for the lengths of each piece of information is a part of its own list with a length one, until the program will move pass the recursive calls of this function recMergeSort in order to merge the list back together again.
6. A function call to the function it is in named recMergeSort, where the information that remains of the head is passed to the function which is the same function that this line of code is a part of in order to update the information being used again.
7. A function call to the function it is in named recMergeSort, where the information by the address of the otherHead is passed to the function which is the same function that this line of code is a

part of, in order to recall the function of line 5 to do again until it is unneeded depending on this update.

8. A function call to the function named mergeList that has been also assigned to the head pointer, where the information of the head pointer and the otherHead are compared, and are merged back to a linked list with the initial length but sorted in ascending order. The function creates two struct node * variables named lastSmall and newHead along with the parameters of the function in order to use as a part of its algorithm. The values of the first nodes are first compared depending if the first value is larger than the second, and if so, the nodes are switched as they merge, but if not, they are left to be as they are merged into back as a linked list. The variation of passing a two-node linked list from passing a three-node list in this case, the program will run through the whole code of this function, or the other parts of the algorithm that with passing a two-node list didn't have to go through, in order for the linked list to be completely sorted and merged back to the same length of final linked list.
9. Ends the declarations of the if statement with the condition when the next pointer the head pointer of the linked list being pass is not equal to a null pointer of line 4 in this case.
10. Ends the functionality of the function recMergeSort.