Sorting in Linked Lists

We have the problem of sorting a Linked List. We will discuss 2 standard algorithms to sort the linked List - Bubble sort and Merge Sort

Bubble Sort Algorithm

We swap the adjacent nodes of the linked list until the list becomes sorted. We do this in the standard bubble sort way repeating the swapping procedure n - 1 times, wherein each iteration we go through the list once and swap every pair of adjacent elements that do not follow the sorting order.

Algorithm:

1. Repeat the step n - 1 times.
2. Make a pass through the list and keep swapping the adjacent elements if node.data > node.next.data.

```
function bubbleSort(head)
    //n is the size of the linked list
    n = head.length
    for each i from 0 to n - 1
        h = head
        for each j from 0 to n  - i - 1
            p1 = h
            p2 = p1->next
            if (p1->data > p2->data)
                // update the link after swapping
                swap(p1, p2)
            h = h->next
    return head
```

Time Complexity: O(N*N), Where N is the number of nodes in the linked list. We make a pass over the linked list n - 1 times and in each of the iterations we make n comparisons.

Space Complexity: O(1). We use constant additional space for swapping the elements.

Merge Sort Algorithm

Merge Sort is a Divide and Conquer algorithm. It divides the input into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.

Algorithm:

1. If the list contains only one node, return the head of the list.
2. Else, divide the list into two sublists. For this, we will take pointers 'mid' and 'tail' which will initially point to the head node. We will change 'mid' and 'tail' as follows until 'tail' becomes NULL:
   mid = mid->next
   tail = tail->next->next
3. The above operation will ensure that mid will point to the middle node of the list. 'mid' will be the head of the second sublist, so we will change the 'next' value of the node which is before mid to NULL.
4. Call the same algorithm for the two sublists.
5. Merge the two sublists and return the head of the merged list. For this, we will take a pointer to a node, 'mergeList', which will be initially pointing to NULL. If the head of the first sublist has a value less than the head of the second sublist then we will point the 'mergeList' to the head of the first sublist and change the head to its next value. Else, we will point the 'mergeList' to the head of the second sub

list. In the end, if any of the sublists becomes empty and the other sublist is non-empty, then we will add all nodes of the non-empty sublist in the ‘mergeList’.

The code looks as follows:

```
function merge(firstHead, secondHead)

    mergeList = NULL, cur = NULL
    // Placing the nodes in sorted order and simultaneously
    // merging the two lists
    while (firstHead is not null && secondHead is not null)
        if (firstHead->data < secondHead->data)
            if (mergeList is null)
                mergeList = firstHead
                cur = firstHead
            else
                cur->next = firstHead
                cur = cur->next
                firstHead = firstHead->next
        else
            if (mergeList is null)
                mergeList = secondHead
                cur = secondHead
            else
                cur->next = secondHead
                cur = cur->next
        secondHead = secondHead->next

    // If any of the list is left, append it at the end of
    // currently merged list
    if (firstHead is not null)
        cur->next = firstHead
    if (secondHead is not null)
        cur->next = secondHead
    return mergeList

function MergeSort(head)
    if (head is null or head->next is null)
        return head
```

```
// dividing list into two sublists
mid = head , tail = head
while (tail is not null && tail->next is not null)
        prev = mid
        mid = mid->next
        tail = tail->next->next

prev->next = NULL
firstHead = head
secondHead = mid
// calling the recursive function on sub-lists
firstHead = MergeSort(firstHead)
secondHead = MergeSort(secondHead)
head = merge(firstHead, secondHead)
return head
```

Time Complexity: O(N*log2(N)), Where N is the number of nodes in the linked list.

The algorithm used (Merge Sort) is divide and conquer. So, for each traversal, we divide the list into 2 parts, thus there are log2(N) levels and the final complexity is O(N * log2(N)).

Space Complexity: O(log2(N)), where N is the number of nodes in the linked list

Since the algorithm is recursive and there are log2(N) levels, it requires O(log2(N)) stack space