Day 126/180

LinkedList Problems 

---

1. Merge Sort for Linked List

```
/* Structure of the linked list node is as
struct Node
{
   int data;
   struct Node* next;
   Node(int x) { data = x;  next = NULL; }
};
*/

class Solution{
  public:

   Node* merge(Node* firstNode, Node* secondNode)
   {
      Node* merged = new Node(-1);
      Node* temp = new Node(-1);

      // merged is equal to temp so in the end we have the top
      // Node.
      merged = temp;

      // while either firstNode or secondNode becomes NULL
      while (firstNode != NULL && secondNode != NULL) {

         if (firstNode->data <= secondNode->data) {
            temp->next = firstNode;
            firstNode = firstNode->next;
         }

         else {
```

```
            temp->next = secondNode;
            secondNode = secondNode->next;
        }
        temp = temp->next;
    }

    // any remaining Node in firstNode or secondNode gets
    // inserted in the temp List
    while (firstNode != NULL) {
        temp->next = firstNode;
        firstNode = firstNode->next;
        temp = temp->next;
    }

    while (secondNode != NULL) {
        temp->next = secondNode;
        secondNode = secondNode->next;
        temp = temp->next;
    }
    // return the head of the sorted list
    return merged->next;
}

    // function to calculate the middle Element
Node* middle(Node* head)
{
    Node* slow = head;
    Node* fast = head->next;

    while (!slow->next && (!fast && !fast->next)) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}

//Function to sort the given linked list using Merge Sort.
Node* mergeSort(Node* head) {
    // your code here
```

```
        if (head->next == NULL) {
            return head;
        }

        Node* mid = new Node(-1);
        Node* head2 = new Node(-1);
        mid = middle(head);
        head2 = mid->next;
        mid->next = NULL;

        Node* finalhead = merge(mergeSort(head), mergeSort(head2));
        return finalhead;
    }
};
```

Code Explanation and Complexity

Intuition

Split the Linked list in half until it cannot be further divided (i.e the list becomes empty or has only one element left) and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied.

Implementation

mergeSort():

- If the size of the linked list is 1 then return the head.
- Find mid using the Tortoise and The Hare Approach.
- Store the next of mid in head2 i.e. the right sub-linked list.
- Now Make the next midpoint null.
- Recursively call mergeSort() on both left and right sub-linked list and store the new head of the left and right linked list.
- Call merge() given the arguments new heads of left and right sub-linked lists and store the final head returned after merging.
- Return the final head of the merged linkedlist.

merge(head1, head2):

- Take a pointer say merged to store the merged list in it and store a dummy node in it.
- Take a pointer temp and assign merge to it.
- If head1->data < head2.data, then, store head1 in the next of temp & move head1 to the next of head1.
- Else store head2 in next of temp & move head2 to the next of head2.

- temp= temp->next.
- Repeat steps 3, 4 & 5 until head1 and head2 are not equal to null.
- Now add any remaining nodes of the first or the second linked list to the merged linked list.
- Return the next merged(that will ignore the dummy and return the head of the final merged linked list).

Time Complexity: O(n*log n), dividing the linked list into half will take log(n) times and merging two sorted array will take O(n) So Time Complexity will be O(n*log n).

Space Complexity: O(log n), Every merge operation takes O(1) space and in the Recursive tree it can be maximum of logN times, So Space Complexity will be O(logn).

2. Merge Sort on Doubly Linked List

```
//Function to merge two halves of list.
struct Node *merge(struct Node *first, struct Node *second)
{
    //base cases when either of two halves is null.
     if (!first)
          return second;
     if (!second)
          return first;

     //comparing data in both halves and storing the smaller in result and
    //recursively calling the merge function for next node in result.
     if (first->data < second->data)
     {
          first->next = merge(first->next,second);
          first->next->prev = first;
          first->prev = NULL;
          //returning the resultant list.
          return first;
     }
     else
     {
          second->next = merge(first,second->next);
          second->next->prev = second;
          second->prev = NULL;
          //returning the resultant list.
          return second;
     }
}

//Function to split the list into two halves.
struct Node *splitList(struct Node *head)
```

```
{
    //using two pointers to find the midpoint of list
     struct Node *fast = head,*slow = head;

     //first pointer, slow moves 1 node and second pointer, fast moves
    //2 nodes in one iteration.
     while (fast->next && fast->next->next)
     {
          fast = fast->next->next;
          slow = slow->next;
     }
     //slow is before the midpoint in the list, so we split the
    //list in two halves from that point.
     struct Node *temp = slow->next;
     slow->next = NULL;
     return temp;
}

//Function to sort the given doubly linked list using Merge Sort.
struct Node *sortDoubly(struct Node *head)
{
     if (!head || !head->next)
          return head;

     //splitting the list into two halves.
     struct Node *second = splitList(head);

     //calling the sortDoubly function recursively for both parts separately.
     head = sortDoubly(head);
     second = sortDoubly(second);

     //calling the function to merge both halves.
     return merge(head, second);
}
```

Code Explanation and Complexity

Merge Function:

- Merges two halves of a doubly linked list.
- Base cases handle scenarios where either of the halves is null.
- Compares the data in the two halves and recursively calls itself for the next nodes in the result.
- Adjusts the pointers for proper merging.

SplitList Function:

- Finds the midpoint of the doubly linked list using two pointers (fast and slow).
- fast moves two nodes at a time, and slow moves one node at a time.
- Splits the list into two halves from the node next to slow.

SortDoubly Function:

- Sorts the doubly linked list using the Merge Sort algorithm.
- Base cases handle scenarios where the list is empty or has only one node.
- Splits the list into two halves using the splitList function.
- Recursively calls itself for both halves.
- Merges the sorted halves using the merge function.
- Returns the head of the sorted list.

Overall Logic:

- The sortDoubly function is the entry point for sorting the doubly linked list.
- The sorting process involves recursively dividing the list into halves until base cases are reached, and then merging the sorted halves.
- The Merge Sort algorithm guarantees a time complexity of O(n log n) for sorting.

Time Complexity:

- The sortDoubly function recursively splits the linked list into halves until there are sub-lists of only one element.
- Each split operation takes linear time in the size of the sublist because we have to traverse the list to find the middle (which takes O(n) time for each level of recursion).
- The merge operation, on the other hand, takes O(n) time for each merge since it involves going through every element of the list exactly once at each level of recursion.
- Merge sort follows a divide and conquer strategy that can be represented by a recurrence relation:
- $T(n) = 2 * T(n/2) + O(n)$
- Using the Master Theorem or simple recursion tree methods, this solves to $T(n) = O(n \log n)$.
- So, the overall time complexity of the sortDoubly function (and thus for the entire sort process) is $O(n \log n)$.

Space Complexity:

- The merge function itself uses constant space because it only creates a few pointers and recursively calls itself; it does not allocate any non-constant space for data structures.
- However, the sortDoubly function uses space due to the recursion stack. In the worst case, the maximum depth of the recursive call stack will be O(log n) because the list is being divided in half each time.
- There are no additional data structures being allocated, and the given merge sort algorithm is doing in-place merging for the doubly linked list.

- Thus, the space complexity of this merge sort implementation is O(log n) due to the recursive calls.

In summary:

Time Complexity: O(n log n)

Space Complexity: O(log n)

3. Reorder List

```
class Solution {
public:

   void reorderList(struct Node* head) {
      int r = 0, c;
      struct Node* temp, *temp1, *temp2, *p;
      temp = head;

      // if list contains 2 or lesser nodes, no change needed
      if (head == NULL || head->next == NULL || head->next->next == NULL) return;

      while (temp != NULL) {
         // finding number of nodes in list
         r++;
         temp = temp->next;
      }

      c = (r + 1) / 2;
      // c represents mid point

      temp = head;
      while (c--) {
         p = temp;
         temp = temp->next;
      }

      p->next = NULL;
      // dividing the list into 2
      // temp holds the address to head of second half
```

```
    temp1 = NULL;
    temp2 = temp;

    // reversing the second half
    while (temp2 != NULL) {
       p = temp2->next;
       temp2->next = temp1;
       temp1 = temp2;
       temp2 = p;
    }
    // second half is now reversed

    p = temp1;
    // merging the 2 halves by selecting nodes alternatively
    Node *a, *b;
    while (p) {
      a = head->next;
      b = p->next;

      head->next=p;
      p->next=a;

      head=a;
      p=b;
    }
  }
};
```

Code Explanation and Complexity

Intuition

- Idea is to split the vector into two halves, then reverse the second half and then merge two halves while picking elements alternatively from each half.

Implementation

- Step 1: If the size of the list is less or equal to two, just return the control.
- Step 2: Calculate the size of the list.
- Step 3: Move a pointer to the (n/2)'th (ceil) node.

//reverse the half list

Now we have 3 pointers, p, temp1 and temp2. temp2 is pointing to next to (n/2)'th (ceil) node.

- Step 4: while temp2 is not equal to null run a loop

  inside the loop

- Step 5: p = temp2 -> next;
- Step 6: temp2 -> next = temp1;
- Step 7: temp1 = temp2;
- Step 8: temp2 = p;

//merge the two list

Now p is pointing to the reversed middle half.

- Step 9: Run a loop until p becomes null.

  inside the loop

- Step 10: a = head->next;
- Step 11: b = p->next;
- Step 12: head->next=p;
- Step 13: p->next=a;
- Step 14: head=a;
- Step 15: p=b;

Time Complexity: As we are traversing the list roughly 2N times so the time complexity is O(N), N is length of the string.

Space Complexity: As we are not using any extra space so the space complexity is O(1).

_____ END _____