Data Structures and Algorithms

Pierre Collet/ Manal ElZant-Karray

Practical work n°4 : Singly Linked List

Exercise 1: Given a list, split it into two sub-lists: one for the front half, and one for the back half. If the number of elements is odd, the extra element should go in the front list. So ***FrontBackSplit()*** on the list {1, 4, 3, 11, 8} should yield the two lists {1, 4, 3} and {11,8}. Check your solution against a few cases (length = 2, length = 3, length=4) to make sure that the list gets split correctly! You will probably need special case code to deal with the (length <2) cases.

Approaches:

1. The first strategy is to compute the length of the list, then use a for loop to hop over the right number of nodes to find the last node of the front half, and then cut the list at that point.
2. The second technique uses two pointers to traverse the list. A "slow" pointer advances one nodes at a time, while the "fast" pointer goes two nodes at a time. When the fast pointer reaches the end, the slow pointer will be about half way.

For either strategy, care is required to split the list at the right point. Write the two ways functions to Split the nodes of the given list into front and back halves, and return the two lists using the reference parameters.

Exercise 2: Write a ***SortedMerge***() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list.

Exercise 3: Given ***FrontBackSplit***() and ***SortedMerge***(), it's easy to write a classic recursive ***MergeSort***(): split the list into two smaller lists, recursively sort those lists, and finally merge the two sorted lists together into a single sorted list.

Example:

```
                              4->1->5->3->NULL
              4->1->NULL                                5->3->NULL
      4->NULL           1->NULL               5->NULL          3->NULL
              1->4->NULL                                3->5->NULL
                              1->3->4->5->NULL
```

Walk through:

Step 1: If head is NULL or there is only one element in the Linked List
then return the Linked List

Step 2: Divide the linked list into two equal halves. (use ***FrontBackSplit*** function)
FrontBackSplit (head, &first_half, &second_half);

Step 3: Sort the two halves first_half and second_half.
MergeSort(first_half);
MergeSort(second_half);

Step 4: Merge the sorted first_half and second_half (using MergeSort() recursively)
and update the head pointer using head_reference.
*head_reference = MergeSort(first_half, second_half);

Exercise 4: Write a ***RemoveDuplicates***() function which takes a list sorted in increasing order and deletes any duplicate nodes from the list.

Exercise 5: Write a program for reverse a given linked list and return the reversed linked list.

Reverse linked list is a linked list created to form a linked list by reversing the links of the list. The head node of the linked list will be the last node of the linked list and the last one will be the head node.

Walk through

To reverse the given linked list you will need three pointers: *previous*, *current* and *following*.

Initialize *previous* and *following* to *NULL* and *current* to *head* of the linked list.

Iterate until you reach the NULL of the initial linked list. And do the following:

```
following = current ->next
current -> next = previous
previous = current
current = following
```

There can be two ways to create the program, one is iterative and the other one is recursive. Try both solutions!