Data Structure

Chapter 5: Linked List

Exam-Oriented Problems and Solutions — Set 3

This third set covers rotation, group reversal, pairwise swapping, digit-wise addition of numbers stored as linked lists, sorted-list duplicate removal, and locating the start of a cycle — all frequently asked variations on the Linked List chapter. Each problem is followed by a complete, step-by-step solution.

Section A: Conceptual and Numerical Problems

Problem 1. In Floyd's Cycle Detection Algorithm, once the slow and fast pointers meet somewhere inside the cycle, describe the second phase used to find the exact starting node of the cycle, and explain briefly why it works. *[5 marks]*

Solution:

1. Keep one pointer (say p1) at the meeting point found by the slow/fast phase.
2. Reset a second pointer, p2, to the head of the list.
3. Move both p1 and p2 forward one node at a time, simultaneously.
4. The node at which p1 and p2 meet again is the starting node of the cycle.

This works because of a distance relationship: if L is the distance from the head to the start of the cycle, and the meeting point is a distance x into the cycle from the cycle's start, it can be shown algebraically (from how far slow and fast travelled before meeting) that L is congruent to (cycle length − x) modulo the cycle length. Advancing a pointer from the head by L steps therefore lands on the cycle's start at exactly the same time as advancing a pointer from the meeting point by the same L steps, which is why moving both one step at a time causes them to meet precisely at the start of the cycle.

Problem 2. A singly linked list has 12 nodes. It is rotated right (counter-clockwise from the end towards the front) by k = 15 positions. How many effective rotation steps actually take place, and why? *[3 marks]*

Solution:

Rotating a list of n nodes by any multiple of n brings it back to its original arrangement, since a full rotation of n positions is equivalent to no rotation at all. The effective number of rotation steps is therefore k mod n.

Here, n = 12 and k = 15, so the effective rotation is 15 mod 12 = 3. This means the algorithm only needs to perform work equivalent to rotating by 3 positions, and any implementation should always reduce k modulo n first, both to avoid unnecessary repeated work and to correctly handle a k value larger than the list length.

Problem 3. Compare the auxiliary space used by an iterative function that finds the length of a singly linked list versus a recursive function that does the same, for a list of n nodes. *[3 marks]*

Solution:

The iterative version uses a single counter variable and a single traversal pointer, both of which occupy constant space regardless of the size of the list, giving O(1) auxiliary space.

The recursive version, by contrast, makes one recursive call per node before any addition is performed (return 1 + length(head->next)), so all n calls remain on the function call stack simultaneously until the base case (an

empty list) is reached and the additions begin to unwind. This gives the recursive version O(n) auxiliary space, purely due to stack frames, even though both versions have the same O(n) time complexity.

Problem 4. To swap two adjacent nodes A -> B in a singly linked list (where P is the node before A, if any), compare the number of pointer/data operations needed if (i) only the data values of A and B are swapped, versus (ii) the nodes themselves are physically relinked. *[4 marks]*

Solution:

(i) Swapping only the data values requires a single temporary variable and 3 assignment operations: temp = A->data; A->data = B->data; B->data = temp; — regardless of where A and B sit in the list, and no pointer is touched at all.

(ii) Physically relinking the nodes (needed, for example, if the nodes contain large or non-copyable records, or if external pointers to the nodes must remain valid) requires updating up to 3 pointers: P->next = B (if P exists), A->next = B->next, and B->next = A — a total of 3 pointer assignments, plus care to save B->next in a temporary pointer before it is overwritten.

Data-swapping is simpler and is preferred whenever the data itself is small and no external reference to the specific node objects needs to be preserved; physical relinking is required when the identity of the node (its address) matters to the rest of the program.

Section B: Algorithm and Code-Based Problems

Problem 5. Write a function to rotate a singly linked list to the right by k positions (0 ≤ k < n), where n is the number of nodes. *[7 marks]*

Solution:

```
struct Node* rotateRight(struct Node *head, int k) {
    if (head == NULL || head->next == NULL || k == 0) return head;

    // Step 1: find length and the last node
    int n = 1;
    struct Node *last = head;
    while (last->next != NULL) { last = last->next; n++; }

    k = k % n;
    if (k == 0) return head;

    // Step 2: make the list circular
    last->next = head;

    // Step 3: find the new tail at position (n - k - 1) from head
    struct Node *newTail = head;
    for (int i = 0; i < n - k - 1; i++)
        newTail = newTail->next;

    // Step 4: new head is right after the new tail
    struct Node *newHead = newTail->next;
    newTail->next = NULL;   // break the circular link

    return newHead;
}
```

The list is first joined into a circle by linking the last node back to the head, then the correct point is cut open at the new boundary: the node at position (n − k − 1) from the original head becomes the new last node, and the node right after it becomes the new head. This runs in O(n) time (one pass to find the length/last node, another partial pass to find the new tail) and O(1) extra space.

Problem 6. Two non-negative integers are represented as singly linked lists, with each node storing a single digit and the least significant digit at the head of the list (so 342 is stored as 2 -> 4 -> 3). Write a function that adds the two numbers and returns the sum as a new linked list in the same format. *[8 marks]*

Solution:

```
struct Node* addTwoNumbers(struct Node *l1, struct Node *l2) {
    struct Node dummy; dummy.next = NULL;
    struct Node *tail = &dummy;
    int carry = 0;

    while (l1 != NULL || l2 != NULL || carry != 0) {
        int sum = carry;
        if (l1 != NULL) { sum += l1->data; l1 = l1->next; }
        if (l2 != NULL) { sum += l2->data; l2 = l2->next; }

        carry = sum / 10;
        tail->next = createNode(sum % 10);
        tail = tail->next;
    }
```

```
    return dummy.next;
}
```

This mimics manual column addition of two numbers, starting from the least significant digit, which is conveniently already at the head of each list. At each step the digits from both lists (if present) and any carry from the previous step are added together; the units digit of this sum (sum % 10) becomes the next digit of the result, and the tens digit (sum / 10) becomes the carry passed into the next step. A dummy head node simplifies building the result list, since it avoids a special case for inserting the very first node. The loop continues as long as either list still has digits or a carry remains, correctly handling numbers of different lengths and a possible extra leading digit from a final carry. This takes O(max(m, n)) time, where m and n are the lengths of the two lists.

Problem 7. Write a function to remove duplicate values from a sorted singly linked list, using only O(1) extra space. *[5 marks]*

Solution:

```
void removeDuplicatesSorted(struct Node *head) {
    struct Node *curr = head;
    while (curr != NULL && curr->next != NULL) {
        if (curr->data == curr->next->data) {
            struct Node *dup = curr->next;
            curr->next = curr->next->next;
            free(dup);
        } else {
            curr = curr->next;
        }
    }
}
```

Because the list is already sorted, all occurrences of a duplicate value are guaranteed to sit next to each other, so only each node's immediate next neighbour needs to be checked. If curr and curr->next hold the same value, curr->next is unlinked and freed, and curr is left in place to check the new curr->next as well (in case of three or more repeats); otherwise curr simply advances. Since only one pointer is used and no extra data structure is needed, this achieves O(n) time and O(1) extra space, unlike the unsorted-list version which requires a hash set.

Problem 8. Write a recursive function to print the elements of a singly linked list in reverse order, without physically reversing the list itself. *[5 marks]*

Solution:

```
void printReverse(struct Node *head) {
    if (head == NULL) return;
    printReverse(head->next);   // go to the end first
    printf("%d ", head->data); // print on the way back
}
```

The recursive call is placed before the print statement, so the function keeps descending to the end of the list (the base case, head == NULL) before printing anything at all. As the recursion then unwinds back up the call stack, each node prints its data only after the rest of the list (everything after it) has already been printed — which produces the elements in reverse order naturally, purely from the order in which the stack unwinds. This takes O(n) time and O(n) auxiliary space due to the recursion stack, since the actual list structure is never modified.

Section C: Trace-Based Problems

Problem 9. Trace the rotateRight algorithm on the list 1 -> 2 -> 3 -> 4 -> 5 -> NULL with k = 2. *[5 marks]*

Solution:

```
Original list:  1 -> 2 -> 3 -> 4 -> 5 -> NULL      (n = 5, k = 2)

Step 1: join into a circle:  1 -> 2 -> 3 -> 4 -> 5 -> (back to 1)
Step 2: new tail is at position n-k-1 = 2 from head (0-indexed) -> node 3
Step 3: new head = newTail->next = node 4
Step 4: break the circle by setting node 3's next to NULL

Result:  4 -> 5 -> 1 -> 2 -> 3 -> NULL
```

The last k = 2 nodes of the original list (4 and 5) have moved to the front, and the remaining nodes (1, 2, 3) follow in their original relative order — exactly the expected effect of a right rotation by 2 positions.

Problem 10. Trace the digit-wise addition algorithm to add the numbers represented by the lists 2 -> 4 -> 3 -> NULL (representing 342) and 5 -> 6 -> 4 -> NULL (representing 465), least significant digit first. *[6 marks]*

Solution:

```
l1: 2 -> 4 -> 3 -> NULL   (342)     l2: 5 -> 6 -> 4 -> NULL   (465)

Step 1: 2 + 5 + carry(0) = 7  -> digit 7, carry 0
Step 2: 4 + 6 + carry(0) = 10 -> digit 0, carry 1
Step 3: 3 + 4 + carry(1) = 8  -> digit 8, carry 0
Both lists exhausted, carry = 0 -> stop

Result list: 7 -> 0 -> 8 -> NULL   (represents 807)
```

This matches ordinary arithmetic: 342 + 465 = 807, confirming that adding digit by digit from the least significant end, while carrying forward any overflow beyond 9, correctly reproduces standard column addition.

Problem 11. Trace an algorithm that swaps every pair of adjacent nodes (by relinking, not just swapping data) on the list 1 -> 2 -> 3 -> 4 -> NULL. *[5 marks]*

Solution:

```
Original list:      1 -> 2 -> 3 -> 4 -> NULL

Pair 1: nodes (1, 2)
  2->next = 1;  1->next = 3 (points to the next pair for now)
  New head becomes 2.        Partial result: 2 -> 1 -> 3 -> 4 -> NULL

Pair 2: nodes (3, 4)
  4->next = 3;  3->next = NULL
  Node 1's next is redirected to 4 (the new head of this pair)

Final result:       2 -> 1 -> 4 -> 3 -> NULL
```

Each pair of adjacent nodes has its internal order reversed (1,2 becomes 2,1 and 3,4 becomes 4,3), while the pairs themselves remain in their original left-to-right order — node 1 (now second) correctly links forward to node 4 (the new first node of the second pair), keeping the whole list connected.

Problem 12. Using the list from Problem 9 of Set 2 (1 -> 2 -> 3 -> 4 -> 5 -> back to 3, cycle start = node 3), where Floyd's Phase 1 found slow and fast meeting at node 4, trace Phase 2 to confirm the cycle actually starts at node 3. *[5 marks]*

Solution:

```
p1 starts at the meeting point: p1 = 4
p2 starts at the head: p2 = 1

Step 1:  p1: 4 -> 5,     p2: 1 -> 2
Step 2:  p1: 5 -> 3,     p2: 2 -> 3      => p1 == p2 == node 3

Both pointers meet at node 3, confirming node 3 is the start of the cycle.
```

This matches the known structure of the list (node 5's next was defined to point back to node 3), verifying both the correctness of Floyd's two-phase method and the earlier meeting-point trace from Set 2.