Data Structure

Chapter 5: Linked List

Exam-Oriented Problems and Solutions

This set covers numerical, algorithmic, trace-based and code-based questions commonly asked in university and competitive engineering examinations on the Linked List chapter. Each problem is followed by a complete, step-by-step solution.

Section A: Conceptual and Numerical Problems

Problem 1. A singly linked list has n nodes. What is the minimum and maximum number of pointer operations (assignments) needed to delete a node from the middle of the list, given only the head pointer? *[5 marks]*

Solution:

To delete a given node from a singly linked list when only the head pointer is available, the node just before the target node must first be located, because its Next pointer is the one that needs to be updated.

1. Traverse from the head using a pointer, say prev, comparing prev->Next's data (or address) with the target — in the worst case this requires visiting up to (n-1) nodes, i.e. O(n) traversal steps.
2. Once prev is found such that prev->Next is the target node, perform: temp = prev->Next; prev->Next = temp->Next; free(temp); — this is exactly 2 pointer assignments (plus one free).

Minimum case: if the node to be deleted is the second node (right after the head), only 1 comparison is needed before the 2 pointer assignments. Maximum case: if the node to be deleted is the last node, up to (n-1) comparisons are needed, still followed by the same 2 pointer assignments. So pointer-assignment count is always exactly 2, but traversal (comparison) steps range from 1 (minimum) to n-1 (maximum).

Problem 2. A singly linked list contains the elements 10 -> 20 -> 30 -> 40 -> NULL. Show the list after each of the following operations is performed in sequence: (i) insert 5 at the beginning, (ii) insert 25 after the node containing 20, (iii) delete the node containing 40. *[5 marks]*

Solution:

```
Initial list:      10 -> 20 -> 30 -> 40 -> NULL
(i) Insert 5 at beginning:
                   5 -> 10 -> 20 -> 30 -> 40 -> NULL
(ii) Insert 25 after node 20:
                   5 -> 10 -> 20 -> 25 -> 30 -> 40 -> NULL
(iii) Delete node containing 40:
                   5 -> 10 -> 20 -> 25 -> 30 -> NULL
```

Note that each operation only changes the pointer(s) of the node(s) immediately adjacent to the point of change; every other node and its links remain untouched, which is the key efficiency advantage of a linked list over an array for insertion and deletion.

Problem 3. How many NULL pointers are present in (a) a singly linked list with n nodes, and (b) a doubly linked list with n nodes? Justify your answer. *[3 marks]*

Solution:

(a) In a singly linked list of n nodes, every node has exactly one link field (Next). Only the last node's Next field is NULL (assuming n ≥ 1); if n = 0, the head pointer itself is the single NULL. So for n ≥ 1, there is exactly 1 NULL pointer.

(b) In a doubly linked list of n nodes, every node has two link fields, Prev and Next. The Prev field of the first node is NULL, and the Next field of the last node is NULL. So for n ≥ 1, there are exactly 2 NULL pointers, one at each end of the list.

Problem 4. Compare the time complexity of accessing the k-th element in an array versus a singly linked list. Explain why the difference arises. *[3 marks]*

Solution:

In an array, the address of the k-th element can be computed directly using the formula base_address + k × element_size, so accessing it takes O(1) time regardless of k, because arrays are stored in contiguous memory and support random access.

In a singly linked list, there is no such direct address formula, because nodes are scattered across memory and connected only through pointers. To reach the k-th node, the list must be traversed starting from the head, following Next pointers one at a time, which takes O(k) time in the worst case, i.e. O(n) overall. This is the fundamental trade-off: arrays trade flexible size for O(1) access, while linked lists trade O(1) access for flexible, low-cost insertion and deletion.

Problem 5. A circular singly linked list has a single pointer called last, pointing to the last node (whose Next points to the first node). Write the expression that gives the address of the first node using only the last pointer. *[2 marks]*

Solution:

Since the list is circular, the Next field of the last node points directly back to the first node. Therefore the first node's address is simply: last->Next

This is exactly why many circular linked list implementations deliberately store a pointer to the last node rather than the first: it gives O(1) access to both the first node (via last->Next) and the last node (via last itself), which is useful, for example, in queue implementations where both ends need to be reached quickly.

Section B: Algorithm and Code-Based Problems

Problem 6. Write a C structure definition for a node of a singly linked list that stores an integer, and write a function to create a new node with a given value. *[5 marks]*

Solution:

```
struct Node {
    int data;
    struct Node *next;
};

struct Node* createNode(int value) {
    struct Node *newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

The structure has two fields: data, which holds the information part of the node, and next, a self-referential pointer of type struct Node* that will store the address of the following node. The createNode function allocates memory dynamically using malloc, fills in the given value, initializes next to NULL (since a freshly created node is not yet linked to anything), and returns the address of the new node.

Problem 7. Write an algorithm/function to insert a new node at the end of a singly linked list, given only the head pointer. *[6 marks]*

Solution:

```
struct Node* insertAtEnd(struct Node *head, int value) {
    struct Node *newNode = createNode(value);
    if (head == NULL) {
        return newNode;         // list was empty
    }
    struct Node *temp = head;
    while (temp->next != NULL) {
        temp = temp->next;      // traverse to the last node
    }
    temp->next = newNode;       // link last node to new node
    return head;
}
```

If the list is empty, the new node itself becomes the head. Otherwise, a temporary pointer traverses the list until it reaches the node whose next field is NULL — the current last node — and that node's next field is then updated to point to the newly created node, whose own next remains NULL. This operation takes O(n) time because the entire list must be traversed to locate the last node; it can be reduced to O(1) if a separate tail pointer is maintained.

Problem 8. Write a recursive function to count the number of nodes in a singly linked list, and derive its time and space complexity. *[5 marks]*

Solution:

```
int countNodes(struct Node *head) {
    if (head == NULL)
        return 0;
    return 1 + countNodes(head->next);
}
```

The base case returns 0 for an empty list (or when the recursion reaches past the last node). For a non-empty list, the function counts the current node as 1 and adds the count of the rest of the list, obtained by recursing on head->next.

Time complexity: O(n), since exactly one recursive call is made per node, visiting each node once. Space complexity: O(n) as well, not O(1), because each recursive call adds a new frame to the call stack, and n such frames exist simultaneously at the deepest point of the recursion — this is worse than the O(1) space of an iterative counting loop.

Problem 9. Write an algorithm to delete the first node of a singly linked list that contains a given value X (assume the value is present). Handle the case where the node to be deleted is the head node separately. *[6 marks]*

Solution:

```
struct Node* deleteValue(struct Node *head, int X) {
    struct Node *curr = head, *prev = NULL;

    if (curr != NULL && curr->data == X) {  // Case 1: X is at the head
        head = curr->next;
        free(curr);
        return head;
    }

    while (curr != NULL && curr->data != X) { // Case 2: search elsewhere
        prev = curr;
        curr = curr->next;
    }

    prev->next = curr->next;   // unlink curr
    free(curr);
    return head;
}
```

Case 1 is handled first: if the head node itself contains X, the head pointer is simply advanced to the second node and the old head is freed — no prev pointer is needed here since there is nothing before the head. Case 2 handles X occurring elsewhere: two pointers, prev and curr, move together through the list until curr reaches the node containing X; then prev's next field is set to curr->next, unlinking curr from the chain, after which curr is freed. Separating these two cases is essential because deleting the head requires updating the external head pointer, whereas deleting an internal node only requires updating another node's next field.

Problem 10. Write a function that checks whether a given singly linked list is a palindrome (reads the same forwards and backwards), using O(n) time and O(1) extra space (besides the list itself). *[8 marks]*

Solution:

```
int isPalindrome(struct Node *head) {
    if (head == NULL || head->next == NULL) return 1;

    // Step 1: find the middle using slow/fast pointers
    struct Node *slow = head, *fast = head;
    while (fast->next != NULL && fast->next->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Step 2: reverse the second half (from slow->next onward)
    struct Node *prev = NULL, *curr = slow->next;
```

```
    while (curr != NULL) {
        struct Node *next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }

    // Step 3: compare first half with reversed second half
    struct Node *p1 = head, *p2 = prev;
    int result = 1;
    while (p2 != NULL) {
        if (p1->data != p2->data) { result = 0; break; }
        p1 = p1->next;
        p2 = p2->next;
    }
    return result;
}
```

The slow/fast pointer technique locates the middle of the list in a single pass: fast moves two steps for every one step of slow, so when fast reaches the end, slow is at the midpoint. The second half of the list, starting just after the midpoint, is then reversed in place using the standard iterative reversal technique. Finally, the first half and the reversed second half are compared node by node from their respective starts; if all corresponding values match, the list is a palindrome. This approach uses each of the three passes in O(n) time and only a constant number of extra pointers, giving overall O(n) time and O(1) extra space.

Section C: Trace-Based Problems (Stacks, Queues, Reversal, Merging)

Problem 11. A stack is implemented using a singly linked list, with the head of the list acting as the top of the stack. Starting from an empty stack, trace the list after each of the following operations: push(10), push(20), push(30), pop(), push(40). *[6 marks]*

Solution:

```
Start:              top -> NULL                       (empty stack)
push(10):           top -> 10 -> NULL
push(20):           top -> 20 -> 10 -> NULL
push(30):           top -> 30 -> 20 -> 10 -> NULL
pop()  [removes 30]:  top -> 20 -> 10 -> NULL      (30 is returned/discarded)
push(40):           top -> 40 -> 20 -> 10 -> NULL
```

Every push inserts a new node at the head, and every pop removes the node currently at the head, which is exactly the beginning-insertion and beginning-deletion operation on a linked list. Because both operations happen only at the head, each push and pop here takes O(1) time, with no traversal required.

Problem 12. A queue is implemented using a singly linked list with both front and rear pointers. Starting from an empty queue, trace the values of front and rear after each operation: enqueue(5), enqueue(15), enqueue(25), dequeue(), enqueue(35). *[6 marks]*

Solution:

```
Start:      front = NULL, rear = NULL              (empty queue)
enqueue(5):  list: 5->NULL          front=5,  rear=5
enqueue(15): list: 5->15->NULL       front=5,  rear=15
enqueue(25): list: 5->15->25->NULL   front=5,  rear=25
dequeue():   removes 5;  list: 15->25->NULL   front=15, rear=25
enqueue(35): list: 15->25->35->NULL  front=15, rear=35
```

Every enqueue links a new node after the current rear and then moves rear to the new node; every dequeue advances front to the next node and discards the old front node. Because rear is maintained explicitly, enqueue never needs to traverse the list to find the last node, so both enqueue and dequeue run in O(1) time here.

Problem 13. Trace the iterative reversal algorithm (using prev, curr, next pointers) on the list 1 -> 2 -> 3 -> NULL, showing the values of prev, curr and next at the start of each loop iteration. *[6 marks]*

Solution:

```
Initial:        prev = NULL,  curr = 1,  head list: 1 -> 2 -> 3 -> NULL

Iteration 1:    next = curr->next = 2
                curr->next = prev   =>  1 -> NULL
                prev = curr = 1,   curr = next = 2

Iteration 2:    next = curr->next = 3
                curr->next = prev   =>  2 -> 1 -> NULL
                prev = curr = 2,   curr = next = 3

Iteration 3:    next = curr->next = NULL
                curr->next = prev   =>  3 -> 2 -> 1 -> NULL
                prev = curr = 3,   curr = next = NULL
```

```
Loop ends (curr == NULL). New head = prev = 3.
Final reversed list:   3 -> 2 -> 1 -> NULL
```

At every iteration exactly three pointer assignments occur (save next, redirect curr's link, advance prev and curr), so the algorithm performs 3n pointer operations in total for a list of n nodes, giving O(n) time and O(1) extra space, as no additional list or array is used.

Problem 14. Merge the two sorted linked lists A: 2 -> 5 -> 8 -> NULL and B: 3 -> 4 -> 9 -> NULL into a single sorted linked list. Show the merged list built step by step. *[6 marks]*

Solution:

```
List A: 2 -> 5 -> 8 -> NULL         List B: 3 -> 4 -> 9 -> NULL

Compare 2, 3  -> pick 2 (from A)     Result: 2
Compare 5, 3  -> pick 3 (from B)     Result: 2 -> 3
Compare 5, 4  -> pick 4 (from B)     Result: 2 -> 3 -> 4
Compare 5, 9  -> pick 5 (from A)     Result: 2 -> 3 -> 4 -> 5
Compare 8, 9  -> pick 8 (from A)     Result: 2 -> 3 -> 4 -> 5 -> 8
A exhausted   -> attach remaining B (9)  Result: 2 -> 3 -> 4 -> 5 -> 8 -> 9

Final merged list:  2 -> 3 -> 4 -> 5 -> 8 -> 9 -> NULL
```

At each step the smaller of the two current front values is appended to the result and that list's pointer is advanced; once one list is fully consumed, the entire remaining (already sorted) portion of the other list is attached directly, since it is already known to be sorted and, being all greater than or equal to the last attached value, requires no further comparison. This merges m and n elements in O(m + n) time using only re-linking, with no extra list allocated.

Problem 15. A singly circular linked list stores the values 10 -> 20 -> 30 and loops back to 10. A pointer p starts at the node containing 10. If p is advanced using p = p->next exactly 7 times, at which node does p end up? Show your working. *[4 marks]*

Solution:

Since the list is circular with 3 nodes, following next repeatedly cycles through the nodes with a period of 3: position 0 = 10, position 1 = 20, position 2 = 30, position 3 = 10 again, and so on.

After k advances, p is at position (k mod 3) counted from the start. Here k = 7, and 7 mod 3 = 1, so p ends up at the node at position 1 from the start, which is the node containing 20.

```
Advance 1: 10 -> 20       Advance 2: 20 -> 30       Advance 3: 30 -> 10
Advance 4: 10 -> 20       Advance 5: 20 -> 30       Advance 6: 30 -> 10
Advance 7: 10 -> 20   =>   p now points to the node containing 20
```