# Data Structure

## Chapter 5: Linked List

*Exam-Oriented Problems and Solutions — Set 4*

This fourth set covers sorting a linked list (merge sort and insertion sort), comparing two lists for equality, polynomial representation and addition using linked lists, and reversing a doubly linked list — all common exam 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. A singly linked list needs to be completely destroyed (every node freed). Compare an iterative deletion approach with a naive recursive deletion approach (free(head) after recursing on head->next) for a list of 100,000 nodes, and explain which is safer in practice.** *[4 marks]*

***Solution:***

An iterative approach walks through the list with a temporary pointer, saving the next node's address, freeing the current node, and moving on — using only O(1) auxiliary space regardless of how many nodes there are.

A naive recursive approach (recurse to the end of the list first, then free nodes while unwinding) places one stack frame per node before any freeing happens, giving O(n) auxiliary space. For n = 100,000 nodes, this can exceed the typical call-stack size limit and cause a stack overflow, crashing the program. The iterative approach is therefore the safer and standard choice for destroying long lists, even though both approaches are O(n) in time.

**Problem 2. A programmer converts a singly linked list into a circular linked list by writing last->next = head, but forgets to keep a variable tracking the last node, relying instead on traversing from head every time the last node is needed. What is the time complexity cost of this omission for each subsequent end-insertion, and how could it be avoided?** *[3 marks]*

***Solution:***

Without a stored pointer to the last node, finding it again requires traversing the entire circular list starting from head until a node whose next pointer equals head is found — an O(n) traversal for every single end-insertion, even though the actual pointer update itself is O(1).

This is avoided by maintaining an explicit external pointer, commonly called last (or tail), that always points directly to the last node of the circular list. Then the first node is reached in O(1) via last->next, and a new end-insertion only needs to update last->next and then move last to the new node, giving O(1) time per insertion instead of O(n).

**Problem 3. A singly linked list has n nodes, each storing an integer. A function multiplies every node's data value by a constant c by traversing the list once. State its time and space complexity, and explain why no extra list needs to be created.** *[3 marks]*

***Solution:***

The function only needs a single traversal pointer that visits each of the n nodes exactly once, updating curr->data = curr->data * c in place at each node, and then advancing to curr->next. This gives O(n) time complexity.

Because each node's data field is modified directly rather than being copied into a new structure, no additional list or array needs to be allocated; only the single traversal pointer is extra, giving O(1) auxiliary space. This in-

place modification pattern is common whenever an operation transforms existing values without changing the list's structure (its length or node order).

**Problem 4. For insertion sort applied to a singly linked list of n nodes, state the best-case and worst-case number of comparisons (in Big-O terms), and describe the input arrangement that produces each case.** *[4 marks]*

***Solution:***

Best case: O(n) comparisons, which occurs when the input list is already sorted in ascending order. Each new node needs to be compared with only the last node of the already-sorted portion once before it is confirmed to belong at the end, so the total work across all n insertions is linear.

Worst case: $O(n^2)$ comparisons, which occurs when the input list is sorted in strictly descending order. Every new node then has to be compared against every node already placed in the sorted portion before finding its correct position (at the very front each time), giving roughly $1 + 2 + ... + (n-1) = O(n^2)$ total comparisons — the same worst-case behaviour as insertion sort on an array.

## Section B: Algorithm and Code-Based Problems

**Problem 5. Write a function to sort a singly linked list using merge sort (divide the list at the middle using slow/fast pointers, recursively sort each half, then merge the two sorted halves).** *[8 marks]*

***Solution:***

```
struct Node* mergeTwoSorted(struct Node *a, struct Node *b) {
    struct Node dummy; dummy.next = NULL;
    struct Node *tail = &dummy;
    while (a != NULL && b != NULL) {
        if (a->data <= b->data) { tail->next = a; a = a->next; }
        else                    { tail->next = b; b = b->next; }
        tail = tail->next;
    }
    tail->next = (a != NULL) ? a : b;
    return dummy.next;
}

struct Node* mergeSort(struct Node *head) {
    if (head == NULL || head->next == NULL) return head;   // base case

    struct Node *slow = head, *fast = head->next;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    struct Node *mid = slow->next;
    slow->next = NULL;              // split into two halves

    struct Node *left  = mergeSort(head);
    struct Node *right = mergeSort(mid);
    return mergeTwoSorted(left, right);
}
```

The slow/fast pointers locate the midpoint of the current sublist in one pass, after which the list is split into two independent halves by cutting the link at slow. Each half is sorted recursively by the same function, down to the base case of a list with 0 or 1 nodes (which is trivially sorted), and the two sorted halves are then combined using the standard sorted-merge routine. Because splitting takes O(n) total per level and there are O(log n) levels of recursion, and merging at each level also takes O(n) total, the overall time complexity is O(n log n) — better than insertion sort's $O(n^2)$ worst case, and, notably, merge sort on a linked list needs no extra O(n) array unlike merge sort on an array, since nodes are simply re-linked.

**Problem 6. Write a function to sort a singly linked list using insertion sort, building a new sorted list by repeatedly inserting each original node into its correct position.** *[6 marks]*

***Solution:***

```
struct Node* insertionSortList(struct Node *head) {
    struct Node *sorted = NULL;
    struct Node *curr = head;

    while (curr != NULL) {
        struct Node *next = curr->next;   // save before relinking curr

        if (sorted == NULL || curr->data <= sorted->data) {
            curr->next = sorted;
            sorted = curr;
        } else {
```

```
            struct Node *temp = sorted;
            while (temp->next != NULL && temp->next->data < curr->data)
                temp = temp->next;
            curr->next = temp->next;
            temp->next = curr;
        }
        curr = next;
    }
    return sorted;
}
```

A separate sorted list is built up node by node, reusing the original nodes rather than copying data. For each node taken from the original list, it is either placed at the front of the sorted list (if it is empty or the new node's value is smaller than the current sorted head) or inserted at the correct position found by scanning the sorted list until a node with a larger value is found. This gives O(n) time in the best case (already sorted input) and O($n^2$) time in the worst case (reverse-sorted input), with O(1) extra space since nodes are relinked in place rather than copied.

**Problem 7. Write a function to check whether two singly linked lists are identical, meaning they have the same length and the same sequence of data values.** *[5 marks]*

***Solution:***

```
int areIdentical(struct Node *a, struct Node *b) {
    while (a != NULL && b != NULL) {
        if (a->data != b->data)
            return 0;      // mismatch found
        a = a->next;
        b = b->next;
    }
    return (a == NULL && b == NULL);  // both must end together
}
```

The two lists are traversed together, one pointer per list, comparing data values at each corresponding position. If any pair of values differs, the lists are immediately known to be different and the function returns false without checking the rest. If the loop finishes because one (or both) pointers become NULL, the final check (a == NULL && b == NULL) ensures that both lists actually ended at the same position — otherwise one list is a proper prefix of the other and they cannot be identical, even though every value that was compared did match. This runs in O(min(m, n)) time, since it can stop as soon as either list ends or a mismatch is found.

**Problem 8. A polynomial is represented as a singly linked list, where each node stores three fields: coeff (coefficient), exp (exponent), and next, with terms kept in strictly decreasing order of exponent. Write a function to add two such polynomials and return the resulting polynomial as a new linked list.** *[8 marks]*

***Solution:***

```
struct PolyNode { int coeff, exp; struct PolyNode *next; };

struct PolyNode* addPoly(struct PolyNode *p1, struct PolyNode *p2) {
    struct PolyNode dummy; dummy.next = NULL;
    struct PolyNode *tail = &dummy;

    while (p1 != NULL && p2 != NULL) {
        if (p1->exp == p2->exp) {
            int sum = p1->coeff + p2->coeff;
            if (sum != 0) {
                tail->next = createPolyNode(sum, p1->exp);
                tail = tail->next;
```

```
            }
            p1 = p1->next; p2 = p2->next;
        } else if (p1->exp > p2->exp) {
            tail->next = createPolyNode(p1->coeff, p1->exp);
            tail = tail->next; p1 = p1->next;
        } else {
            tail->next = createPolyNode(p2->coeff, p2->exp);
            tail = tail->next; p2 = p2->next;
        }
    }
    while (p1 != NULL) { tail->next = createPolyNode(p1->coeff, p1->exp); tail = tail-
>next; p1 = p1->next; }
    while (p2 != NULL) { tail->next = createPolyNode(p2->coeff, p2->exp); tail = tail-
>next; p2 = p2->next; }
    return dummy.next;
}
```

Since both polynomials are stored with exponents in decreasing order, they can be merged much like two sorted lists: at each step, the terms with the higher exponent are copied through unchanged, while terms with equal exponents have their coefficients added together (and are dropped entirely if the sum is zero, to keep the result in proper polynomial form). Once one polynomial is exhausted, the remaining terms of the other are copied through as-is, since there is nothing left to combine them with. This takes O(m + n) time for polynomials with m and n terms respectively, exactly analogous to merging two sorted linked lists.

## Section C: Trace-Based Problems

**Problem 9. Trace the merge sort algorithm on the list 6 -> 2 -> 9 -> 4 -> NULL, showing the split at each level and the merges that produce the final sorted list.** *[6 marks]*

***Solution:***

```
Original list:            6 -> 2 -> 9 -> 4

Split (slow/fast on full list):
   Left half:  6 -> 2          Right half: 9 -> 4

Split further (each half has 2 nodes):
   Left half splits into:   [6]  and  [2]
   Right half splits into:  [9]  and  [4]
   (single-node lists are the base case – already 'sorted')

Merge [6] and [2]  ->  2 -> 6
Merge [9] and [4]  ->  4 -> 9

Final merge of (2 -> 6) and (4 -> 9):
   compare 2,4 -> take 2      Result: 2
   compare 6,4 -> take 4      Result: 2 -> 4
   compare 6,9 -> take 6      Result: 2 -> 4 -> 6
   right exhausted after 6 is taken from left -> attach remaining 9
   Result: 2 -> 4 -> 6 -> 9

Final sorted list:   2 -> 4 -> 6 -> 9 -> NULL
```

Each level of splitting takes the list closer to single-node sublists (the base case), and each level of merging combines sorted sublists into larger sorted sublists, until the single fully sorted list remains — the hallmark divide-and-conquer pattern of merge sort, taking O(n log n) time overall for n nodes.

**Problem 10. Trace the insertion sort algorithm on the list 5 -> 1 -> 4 -> 2 -> NULL, showing the sorted portion after each node is processed.** *[6 marks]*

***Solution:***

```
Original list:  5 -> 1 -> 4 -> 2 -> NULL

Process 5:  sorted is empty -> sorted = 5
Process 1:  1 <= 5 -> insert at front       sorted = 1 -> 5
Process 4:  4 > 1, and 4 < 5 -> insert between 1 and 5   sorted = 1 -> 4 -> 5
Process 2:  2 > 1, and 2 < 4 -> insert between 1 and 4   sorted = 1 -> 2 -> 4 -> 5

Final sorted list:   1 -> 2 -> 4 -> 5 -> NULL
```

Each original node is removed from the input one at a time and placed directly into its correct position within the growing sorted portion by scanning from the sorted list's head; no separate array is used, and the same node objects are reused, only their next pointers being changed.

**Problem 11. Two polynomials are given as linked lists: P1 = $5x^2 + 3x + 1$ and P2 = $4x^2 + 2$. Trace the polynomial addition algorithm to compute P1 + P2.** *[5 marks]*

***Solution:***

```
P1: (5,2) -> (3,1) -> (1,0) -> NULL      [5x^2 + 3x + 1]
P2: (4,2) -> (2,0) -> NULL               [4x^2 + 2]

Compare exp 2 vs exp 2 (equal): coeff 5+4=9  -> add term (9,2);  advance both
Compare exp 1 (P1) vs exp 0 (P2): P1 exp larger -> copy (3,1);  advance P1
Compare exp 0 (P1) vs exp 0 (P2) (equal): coeff 1+2=3 -> add term (3,0); advance both
Both lists exhausted -> stop

Result: (9,2) -> (3,1) -> (3,0) -> NULL   =  9x^2 + 3x + 3
```

This matches direct algebraic addition: $(5x^2 + 3x + 1) + (4x^2 + 0x + 2) = 9x^2 + 3x + 3$, confirming that comparing exponents term by term (since both polynomials are kept sorted by decreasing exponent) correctly combines like terms and carries through unmatched terms unchanged.

**Problem 12. A doubly linked list is NULL <- 10 <-> 20 <-> 30 -> NULL. Trace the in-place reversal algorithm (swap each node's prev and next, then move to the new next) and show the final list along with the new head.** *[6 marks]*

***Solution:***

```
Nodes: A(10), B(20), C(30).  Initially: A.prev=NULL,A.next=B; B.prev=A,B.next=C;
C.prev=B,C.next=NULL
current = A, temp = NULL

Iteration 1 (current = A):
   temp = A.prev (NULL);  A.prev = A.next (B);  A.next = temp (NULL)
   => A: prev=B, next=NULL.     current = A.prev = B

Iteration 2 (current = B):
   temp = B.prev (A);  B.prev = B.next (C);  B.next = temp (A)
   => B: prev=C, next=A.        current = B.prev = C

Iteration 3 (current = C):
   temp = C.prev (B);  C.prev = C.next (NULL);  C.next = temp (B)
   => C: prev=NULL, next=B.     current = C.prev = NULL  (loop ends)

New head = temp->prev = B.prev = C

Final list:   NULL <- 30 <-> 20 <-> 10 -> NULL
```

Every node's prev and next fields are swapped in place, which by itself reverses the direction of traversal at each node; the final adjustment (new head = temp->prev) simply relocates the external head pointer to node C (30), which is now correctly the first node of the reversed list, matching the expected result of reversing 10 <-> 20 <-> 30 into 30 <-> 20 <-> 10.