1634. Add Two Polynomials Represented as Linked Lists

Medium | Linked List | Math | Two Pointers | Leetcode Link

Problem Description

The problem defines a special type of linked list where each node represents a term in a polynomial expression. Here are the key points about the linked list structure and the problem requirements:

- Each node contains a `coefficient` (an integer representing the multiplier) and a `power` (an integer representing the exponent).
- The linked list represents a polynomial in which the terms are in strictly descending order by their power.
- Terms with a coefficient of 0 are not included in the list.
- The goal is to add two such polynomial linked lists and return the resulting polynomial linked list that maintains the same properties.

For example, two polynomials `5x^3 + 4x - 7` and `3x^3 - 2x + 8` are represented by their respective linked lists. The task is to sum these polynomials to get a new linked list representing the polynomial `8x^3 + 4x + 1`.

Intuition

To solve the addition of two polynomials represented by linked lists, we follow these steps:

1. Create a dummy head for the result polynomial linked list, which helps in easily returning the result.
2. Traverse both linked lists simultaneously while comparing the powers of the current nodes.
3. If one polynomial has a higher power term than the other, we add that term to the result list and move to the next term in that polynomial.
4. If both have terms with the same power, we add the coefficients. If the sum is not zero, we create a new node with this sum and the corresponding power, then add it to the result list.
5. Continue this process until we reach the end of one or both polynomials.
6. If there are remaining terms in one of the polynomials, append them to the result list.
7. Finally, return the next node of the dummy head as it points to the actual head of the result linked list.

By ensuring terms are added in descending order of power, we construct a valid polynomial linked list representing the sum of the input polynomials.

Solution Approach

To implement the solution for adding two polynomial linked lists, here's the step-by-step approach used:

1. **Initial Setup**: We use a dummy node approach to simplify the process of building a new linked list. We start with a dummy `PolyNode` which acts as a placeholder for the head of the new list, and a `curr` pointer which is used to add new nodes to the list.

   ```
   dummy = curr = PolyNode()
   ```

2. **Traversing Both Lists**: We use a while loop to iterate through both `poly1` and `poly2` as long as there are nodes in both lists.

   ```
   while poly1 and poly2:
   ```

3. **Power Comparison and Node Addition**: Inside the loop, we compare the powers of the current nodes from `poly1` and `poly2`.

   - If `poly1` has a higher power term, we link the current term from `poly1` to `curr` and advance the `poly1` pointer.
   - If `poly2` has a higher power term, we do the equivalent by linking `poly2` to `curr` and advancing `poly2`.
   - If both terms have the same power, we sum their coefficients and only create and link a new node if the resultant coefficient is not zero. We then advance both pointers.

   ```
   if poly1.power > poly2.power:
       curr.next = poly1
       poly1 = poly1.next
   elif poly1.power < poly2.power:
       curr.next = poly2
       poly2 = poly2.next
   else:
       if c := poly1.coefficient + poly2.coefficient:
           curr.next = PolyNode(c, poly1.power)
       poly1 = poly1.next
       poly2 = poly2.next
   ```

   We need to update `curr` to the next node in the list after adding a node:

   ```
   curr = curr.next
   ```

4. **Handling Remaining Terms**: After the loop, there might be remaining nodes in either `poly1` or `poly2` if they were not of equal length. We append the remaining nodes to `curr`.

   ```
   curr.next = poly1 or poly2
   ```

5. **Returning the Result**: Since `dummy` is a placeholder with an unused `0` coefficient and `0` power, we return `dummy.next`, which represents the head of the resultant polynomial linked list.

The overall complexity of the algorithm is linear O(n + m), where `n` and `m` are the number of nodes in `poly1` and `poly2` respectively since each list is only traversed once and nodes are added to the result list in a single pass.

Example Walkthrough

We will walk through a small example to illustrate the solution approach.

Suppose we have two polynomials represented by linked lists:

- `poly1` represents `5x^2 + 3x + 1`
- `poly2` represents `2x^2 + x`

Represented as linked lists, they would look something like this:

poly1: 5 → 3 → 1 (each node has terms decreasing in power from left to right) poly2: 2 → 1

Steps:

1. We initiate a dummy node, which will help us in easily building and returning the result.
2. We begin to iterate through both lists:
   - Comparing the powers, 2 and 2 have the same power for the first nodes.
   - We add the coefficients (5 + 2) = 7.
   - Create a new node with coefficient 7 and power 2 and link it to `curr`.
   - Move both `poly1` and `poly2` to the next nodes (3 and 1 respectively).
3. The next comparison:
   - `poly1` has power `1` and `poly2` also has power `1`.
   - Add coefficients (3 + 1) = 4.
   - Create a new node with coefficient 4 and power 1 and link it to `curr`.
   - Move both pointers forward.
4. Now, `poly2` has been completely traversed, but `poly1` still has one term left:
   - No more comparison is needed.
   - Append the last node of `poly1` (which is `1`) to `curr`.
5. The final polynomial linked list, represented by `dummy.next`, is `7x^2 + 4x + 1`.

The constructed linked list will have the following structure, representing the sum of the input polynomials: 7 → 4 → 1

Python Solution

```python
# Definition for polynomial singly-linked list.
class PolyNode:
    def __init__(self, coefficient=0, power=0, next_node=None):
        self.coefficient = coefficient
        self.power = power
        self.next = next_node


class Solution:
    def addPoly(self, poly1: "PolyNode", poly2: "PolyNode") -> "PolyNode":
        # Create a dummy node to serve as the head of the result list
        dummy = current = PolyNode()

        # Iterate while neither of the polynomial lists is exhausted
        while poly1 and poly2:
            # Compare powers of both poly nodes
            if poly1.power > poly2.power:
                # If poly1 has the higher power, append it to the result list
                current.next = poly1
                poly1 = poly1.next
            elif poly1.power < poly2.power:
                # If poly2 has the higher power, append it to the result list
                current.next = poly2
                poly2 = poly2.next
            else:
                # Both powers are equal, add coefficients if they don't cancel out
                sum_coefficient = poly1.coefficient + poly2.coefficient
                if sum_coefficient != 0:
                    current.next = PolyNode(sum_coefficient, poly1.power)
                # Move to the next nodes in both lists
                poly1 = poly1.next
                poly2 = poly2.next

            # Move to the next node in the result list if we appended a node
            if current.next:
                current = current.next

        # Append the remaining nodes of the polynomial list that is not yet exhausted
        current.next = poly1 or poly2

        # Skip the dummy node and return the result list
        return dummy.next

# Please note that the main method signature of addPoly cannot be changed as it's probably a constraint given by the problem statemer
```

Java Solution

```java
/**
 * Definition for a polynomial singly-linked list node.
 */
class PolyNode {
    int coefficient, power;
    PolyNode next = null;

    // Constructors
    PolyNode() {}
    PolyNode(int x, int y) { this.coefficient = x; this.power = y; }
    PolyNode(int x, int y, PolyNode next) { this.coefficient = x; this.power = y; this.next = next; }
}

class Solution {
    /**
     * Adds two polynomials represented as linked lists.
     * @param poly1 First polynomial represented as a linked list.
     * @param poly2 Second polynomial represented as a linked list.
     * @return The result of the polynomial addition, also as a linked list.
     */
    public PolyNode addPoly(PolyNode poly1, PolyNode poly2) {
        // dummy node to simplify list operations
        PolyNode dummyNode = new PolyNode();
        // iterator for the resultant list
        PolyNode current = dummyNode;

        // Iterate over both polynomial linked lists
        while (poly1 != null && poly2 != null) {
            if (poly1.power > poly2.power) {
                // The term from poly1 should come next in the result list
                current.next = poly1;
                poly1 = poly1.next;
            } else if (poly1.power < poly2.power) {
                // The term from poly2 should come next in the result list
                current.next = poly2;
                poly2 = poly2.next;
            } else {
                // Powers are equal, sum the coefficients
                int sumCoefficient = poly1.coefficient + poly2.coefficient;
                if (sumCoefficient != 0) { // Only add non-zero terms to the result
                    current.next = new PolyNode(sumCoefficient, poly1.power);
                }
                // Move forward in both lists
                poly1 = poly1.next;
                poly2 = poly2.next;
            }
            // Don't move current in the case where sumCoefficient is zero to avoid zero coefficient terms in the list
            if (current.next != null) {
                current = current.next;
            }
        }

        // If there's remaining terms in poly1 or poly2, append them to the result
        current.next = (poly1 != null) ? poly1 : poly2;

        // The first node is dummy, so return the next node which is the head of the actual result
        return dummyNode.next;
    }
}
```

C++ Solution

```cpp
/**
 * Definition for polynomial singly-linked list.
 * This structure is used for representing terms in a polynomial expression
 * where 'coefficient' represents the term's coefficient and 'power' represents
 * the term's exponent. The linked list is sorted in decreasing order of powers.
 */
struct PolyNode {
    int coefficient, power;
    PolyNode *next;
    PolyNode() : coefficient(0), power(0), next(nullptr) {};
    PolyNode(int x, int y) : coefficient(x), power(y), next(nullptr) {};
    PolyNode(int x, int y, PolyNode* next) : coefficient(x), power(y), next(next) {};
};

class Solution {
public:
    /**
     * Adds two polynomial expressions represented by linked lists.
     * @param poly1 First polynomial linked list
     * @param poly2 Second polynomial linked list
     * @return A new polynomial linked list representing the sum of poly1 and poly2.
     */
    PolyNode* addPoly(PolyNode* poly1, PolyNode* poly2) {
        // Dummy node to simplify edge cases and to keep a reference to list start
        PolyNode* dummyHead = new PolyNode();
        PolyNode* current = dummyHead;

        while (poly1 && poly2) {
            if (poly1->power > poly2->power) {
                // If poly1 has a term with greater exponent, append it next in the result
                current->next = poly1;
                poly1 = poly1->next;
            } else if (poly1->power < poly2->power) {
                // If poly2 has a term with greater exponent, append it next in the result
                current->next = poly2;
                poly2 = poly2->next;
            } else {
                // If both have the same exponent, sum the coefficients
                int sumCoefficient = poly1->coefficient + poly2->coefficient;
                if (sumCoefficient != 0) {
                    // Append the new term if the sum is non-zero
                    current->next = new PolyNode(sumCoefficient, poly1->power);
                    current = current->next;
                }
                // Move forward in both lists
                poly1 = poly1->next;
                poly2 = poly2->next;
            }
            // Move forward in the result list if we appended a term
            if (current->next) {
                current = current->next;
            }
        }

        // Append any remaining terms from either polynomial to the result
        current->next = (poly1 != nullptr) ? poly1 : poly2;

        // Return the next of dummyHead which points to the start of the sum list
        PolyNode* headOfSum = dummyHead->next;
        delete dummyHead; // Clear the memory occupied by the dummy node
        return headOfSum;
    }
};
```

Typescript Solution

```typescript
// Define a class for a polynomial node.
class PolyNode {
    coefficient: number;
    power: number;
    next: PolyNode | null;

    constructor(x: number = 0, y: number = 0, next: PolyNode | null = null) {
        this.coefficient = x;
        this.power = y;
        this.next = next;
    }
}

/**
 * Adds two polynomial singly-linked lists and returns the sum as a new polynomial singly-linked list.
 * @param poly1 The first polynomial singly-linked list.
 * @param poly2 The second polynomial singly-linked list.
 * @return The sum of poly1 and poly2 as a new polynomial singly-linked list.
 */
const addPoly = (poly1: PolyNode | null, poly2: PolyNode | null): PolyNode | null => {
    // Create a dummy node to serve as the anchor for the result list.
    const dummy = new PolyNode();
    // This will be used to iterate over the new polynomial linked list.
    let current = dummy;

    // Iterate over both polynomials as long as there are nodes in each list.
    while (poly1 && poly2) {
        if (poly1.power > poly2.power) {
            // If the current term of poly1 has a greater power, include it in the result.
            current.next = poly1;
            current = current.next;
            poly1 = poly1.next; // Move to the next term in poly1.
        } else if (poly1.power < poly2.power) {
            // If the current term of poly2 has a greater power, include it in the result.
            current.next = poly2;
            current = current.next;
            poly2 = poly2.next; // Move to the next term in poly2.
        } else {
            // If the powers are equal, add the coefficients if non-zero and include in the result.
            const sumOfCoefficients = poly1.coefficient + poly2.coefficient;
            if (sumOfCoefficients != 0) {
                current.next = new PolyNode(sumOfCoefficients, poly1.power);
                current = current.next;
            }
            // Move to the next terms in both poly1 and poly2.
            poly1 = poly1.next;
            poly2 = poly2.next;
        }
    }

    // Attaching the remaining elements of poly1 or poly2 in case one is longer than the other.
    current.next = poly1 || poly2;

    // The first node in our resulting linked list is a dummy node,
    // so we return the next node which starts the resulting polynomial.
    return dummy.next;
};
```

Time and Space Complexity

The given code defines a function `addPoly` that takes two polynomial singly-linked lists as input, where each node represents a term in the polynomial (with a coefficient and power), and returns a new polynomial linked list representing the sum of the two input polynomials.

Time Complexity

The time complexity of this function is `O(n + m)`, where `n` is the number of terms in `poly1` and `m` is the number of terms in `poly2`. This is because the function iterates over both lists at most once, advancing one node at a time in whichever list has the term with the higher power, or both if the powers are equal and need to be summed.

Each operation inside the while loop—comparing powers, creating a new node (when coefficients sum to a non-zero value), and linking nodes—occurs in constant time, `O(1)`. Since these operations are not nested and we only iterate through each list once without revisiting any nodes, the time complexity is linear with respect to the total number of nodes in both input lists.

Space Complexity

The space complexity of the function is `O(max(n, m))`. In the worst-case scenario every term from both polynomials needs to be included in the resulting polynomial. However, since the input polynomials' nodes are reused when possible (i.e., when one term's power is greater than the other or when the sum of coefficients is non-zero), additional space is only required when two terms have the same power and their coefficients sum to a non-zero value, which requires allocating a new node. Therefore, the space complexity depends on the number of such term pairs, which is at most the length of the shorter list, resulting in a space complexity that is the maximum of `n` or `m`, excluding any space that may be required for the output.