1290. Convert Binary Number in a Linked List to Integer

Easy Linked List Math

Problem Description

In this problem, we're given a singly-linked list that represents a binary number. Each node of the linked list contains a `0` or a `1`, with the head of the list representing the most significant bit (MSB) of the binary number. We need to determine the decimal value represented by this binary number.

Intuition

The intuition behind the solution is that we can initialize an integer to hold our result and iterate over each node of the linked list from head to tail. As we move through each node, we perform two operations on the current result:

1. Left shift the current result by 1 bit. This is similar to multiplying the current decimal number by 2 in a binary system, effectively shifting all the bits to the left and making room for the next bit.
2. Use the bitwise OR operation to include the value of the current node (`0` or `1`) in the least significant bit (LSB) of our current result.

By repeatedly applying these two steps for each node, we can build up the decimal value bit by bit from the binary representation in the linked list.

Solution Approach

The implementation of the solution directly reflects the intuition behind the approach. Here, we use bitwise operations which are a very efficient way to handle binary numbers.

1. Initialize an integer `ans` to `0` which will eventually contain the decimal value of the binary number.
2. Start a while loop that will continue as long as there is a node in the linked list (`while head:`).
3. Within the loop, update `ans` using the left shift (`<<`) operator. Shifting `ans` by one bit to the left (`ans << 1`) is equivalent to multiplying the current `ans` by 2 (in binary representation, this adds a new bit set to `0` at the end).
4. Perform a bitwise OR (`|`) with the value of the current node (`head.val`). Since `head.val` is either `0` or `1`, using `|` will set the new least significant bit of `ans` to the value of the current node.
5. Move on to the next node (`head = head.next`).
6. Once we've iterated over all nodes, the loop ends, and we return `ans`, which now stores the decimal value of the binary number represented by the linked list.

The algorithm uses a single pass to convert the binary number to decimal, making the time complexity O(n) where n is the number of nodes in the linked list. No extra space other than the variable `ans` is used, resulting in a space complexity of O(1).

The data structure used here is the given singly-linked list, and the pattern followed is essentially bit manipulation to convert from binary to decimal efficiently.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Imagine we have a singly-linked list representing the binary number `1011`, which looks like this:

```
1 -> 0 -> 1 -> 1
```

Where the head of the list represents the most significant bit (MSB).

Here are the steps to compute the decimal value of this binary:

1. Initialize an integer `ans` to `0`.
2. The head points to the first node containing `1`, start the while loop.
3. Left shift `ans` by 1 bit, resulting in `ans = 0 << 1 = 0`.
4. Perform a bitwise OR with the value of the current node (`ans | head.val`), resulting in `ans = 0 | 1 = 1`.
5. Move on to the next node, which contains `0`.
6. Left shift `ans` by 1 bit, resulting in `ans = 1 << 1 = 2`.
7. Perform a bitwise OR with the value of the current node (`ans | head.val`), resulting in `ans = 2 | 0 = 2`.
8. Move on to the next node, which contains `1`.
9. Left shift `ans` by 1 bit, resulting in `ans = 2 << 1 = 4`.
10. Perform a bitwise OR with the value of the current node (`ans | head.val`), resulting in `ans = 4 | 1 = 5`.
11. Move on to the last node, which contains `1`.
12. Left shift `ans` by 1 bit, resulting in `ans = 5 << 1 = 10`.
13. Perform a bitwise OR with the value of the current node (`ans | head.val`), resulting in `ans = 10 | 1 = 11`.

After iterating over all the nodes, we conclude that the decimal value of the binary number `1011` is `11`. The `while` loop ends and we return `ans`, which now contains the correct decimal value.

Solution Implementation

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def getDecimalValue(self, head: ListNode) -> int:
        # Initialize the result variable to 0.
        result = 0

        # Traverse the linked list.
        while head:
            # Bitwise left shift result by 1 (equivalent to multiplying by 2)
            # and perform bitwise OR with the current node's value
            # to append it to the binary number we are building.
            result = (result << 1) | head.val

            # Move to the next node in the linked list.
            head = head.next

        # Return the final decimal value of the binary number.
        return result
```

Java

```java
/**
 * Definition for singly-linked list.
 */
class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

class Solution {
    /**
     * Gets the decimal value of a binary number represented as a singly-linked list.
     * where each node contains a binary digit. The most significant bit is at the head of the list.
     *
     * @param head The head of the singly-linked list representing the binary number.
     * @return The integer decimal value of the binary number.
     */
    public int getDecimalValue(ListNode head) {
        int number = 0; // Initialize a variable to store the decimal number.

        // Iterate through each node of the list until the end is reached.
        while (head != null) {
            // Left-shift 'number' by 1 bit to make space for the new bit, and then
            // combine it with the current node's value using bitwise OR operation.
            number = (number << 1) | head.val;

            // Move to the next node in the list.
            head = head.next;
        }

        // Return the decimal number that is represented by the binary list.
        return number;
    }
}
```

C++

```cpp
// Definition for singly-linked list.
struct ListNode {
  int val; // Value of the node
  ListNode *next; // Pointer to the next node
  ListNode() : val(0), next(nullptr) {} // Constructor initializes node with 0
  ListNode(int x) : val(x), next(nullptr) {} // Constructor initializes node with given value
  ListNode(int x, ListNode *next) : val(x), next(next) {} // Constructor initializes node with given value and next pointer
};

class Solution {
public:
    // Function to convert a binary number in a linked list to an integer.
    int getDecimalValue(ListNode* head) {
        int result = 0; // This will hold the decimal value of the binary number

        // Iterate through the linked list
        while (head != nullptr) {
            // Left shift the result by 1 bit to make space for the next bit
            // Then perform a bitwise OR with the current node's value to add the bit to the number
            result = (result << 1) | head->val;

            // Move to the next node
            head = head->next;
        }
        // Return the computed decimal value
        return result;
    }
};
```

TypeScript

```typescript
// TypeScript definition for a node in a singly-linked list.
interface ListNode {
    val: number;
    next: ListNode | null;
}

/**
 * This function calculates the decimal value of a binary number represented
 * as a singly-linked list. where each node contains a single digit.
 * The digits are stored in reverse order, meaning the head of the list
 * represents the least significant bit.
 *
 * @param {ListNode | null} head - The head of the singly-linked list.
 * @return {number} - The decimal value of the binary number.
 */
function getDecimalValue(head: ListNode | null): number {
    // Initialize answer as 0 to prepare for binary to decimal conversion.
    let decimalValue = 0;

    // Traverse the linked list starting from the head until the end.
    while (head !== null) {
        // Left shift the current decimal value by 1 bit (equivalent to multiplying by 2)
        // to make space for the next binary digit, and then use bitwise OR to add
        // the value of the current node (either 0 or 1) to the least significant bit of the answer.
        decimalValue = (decimalValue << 1) | head.val;

        // Move to the next node in the list.
        head = head.next;
    }

    // Return the computed decimal value after traversing the entire list.
    return decimalValue;
}
```

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def getDecimalValue(self, head: ListNode) -> int:
        # Initialize the result variable to 0.
        result = 0

        # Traverse the linked list.
        while head:
            # Bitwise left shift result by 1 (equivalent to multiplying by 2)
            # and perform bitwise OR with the current node's value
            # to append it to the binary number we are building.
            result = (result << 1) | head.val

            # Move to the next node in the linked list.
            head = head.next

        # Return the final decimal value of the binary number.
        return result
```

Time and Space Complexity

The provided code snippet defines a method `getDecimalValue` that converts a binary number represented by a linked list to its decimal value. Let's analyze the time complexity and space complexity.

Time Complexity:

The time complexity of the function is determined by the while loop that iterates once for each element in the linked list. Inside the while loop, the operations performed (bit shift and bitwise OR) are constant time operations, which means they take `O(1)` time. If there are `n` elements in the linked list, the loop runs `n` times. Therefore, the time complexity is `O(n)`.

Space Complexity:

As for the space complexity, the function only uses a fixed number of variables (`ans` and `head`) regardless of the input size. There are no additional data structures used that grow with the size of the input. Hence, the space complexity is `O(1)`, which is constant space complexity.