382. Linked List Random Node

Medium Reservoir Sampling Linked List Math Randomized

Leetcode Link

Problem Description

The problem defines a scenario in which you are given a singly linked list, and you are required to implement a method that returns the value of a random node from that linked list. Importantly, each node in the list should have an equal chance of being chosen. This is achieved by implementing two functions within a `Solution` class:

- `Solution(ListNode head)`: A constructor that initializes an instance of the `Solution` class with the given singly linked list's head.
- `int getRandom()`: A function that, when called, should return the value of a randomly chosen node from the linked list, ensuring each node has the same probability of being selected.

Intuition

To solve this problem, we need a method that not only selects a node at random but also ensures each node has an equal chance of being chosen. This problem is known as "reservoir sampling," which is especially useful when the size of the data is unknown or is too large to fit into memory.

The solution provided follows the reservoir sampling algorithm. The intuition behind reservoir sampling is as follows:

1. Iterate over each node in the given linked list while keeping track of the current node index/counter.
2. On visiting each node, generate a random number between 1 and the current node's index (inclusive).
3. If the generated random number equals the current node's index, we select the current node's value as the candidate answer.
4. Continue this process until the end of the list. Once the end is reached, the value selected will be from one of the nodes, and each node would have had an equal chance of being chosen.

The reason this works is that at each step, the probability of keeping the existing item is equal to the probability of replacing it with the current item, which maintains the uniform random selection of the items. In each iteration, the probability of choosing a node is 1 over the number of nodes encountered so far, thus ensuring the same probability for each node.

Solution Approach

The implementation of the `Solution` class uses the reservoir sampling algorithm to address the challenge of equally likely node selection from the linked list. Let's walk through the steps of the approach:

1. The `__init__` function simply stores the reference to the head of the linked list in the `self.head` variable. No other initial processing or storage is needed, making initialization straightforward.

```
def __init__(self, head: Optional[ListNode]):
    self.head = head
```

2. The `getRandom` function is where the reservoir sampling algorithm is applied. Let's examine it step-by-step:

```
def getRandom(self) -> int:
    n = ans = 0
    head = self.head
    while head:
        n += 1
        if r.random.randint(1, n) == n:
            ans = head.val
        head = head.next
    return ans
```

- A variable `n` is used to keep track of the number of nodes that have been processed (essentially it's the current node's index).
- A variable `ans` is used to store the current answer, which is the value of the randomly chosen node.
- We start a loop that continues until it reaches the end of the linked list (`head` being `None`).
- For each node encountered, we increase `n` by 1.
- We then call `random.randint(1, n)` to generate a random number between `1` and `n` (inclusive).
- If the generated random number `x` equals `n` (which is the current index of the node), we update `ans` with the current node's value. The probability of `x` being equal to `n` is `1 / n`, which corresponds to the probability of selecting any one node when there are `n` nodes.
- We move the pointer `head` to the next node in the list.
- After the entire list has been traversed, the value accumulated in `ans` is returned. Due to the probabilities involved in the selection process, this value is the randomly selected node's value with a uniform distribution.

In summary, the algorithm uses a linear scan of the list with a randomness-driven decision made at each node visited to either replace the current selection or continue with the existing one. The random number determines this replacement in accordance with reservoir sampling, requiring no auxiliary data structure, thus achieving optimal space complexity.

Example Walkthrough

Let's consider a linked list of 5 nodes with values from 1 to 5. Here is how the `Solution` class would select a random node using the reservoir sampling technique:

1. Construct the `Solution` with the head of the linked list.
2. Call `getRandom()` to select a random node. The selection process is as follows:
   - Start with `n = 0` and `ans = 0`.
   - Visit the first node (value = 1): `n=1`. Generate a random number `x` between 1 and 1 (`random.randint(1, n)`). Since `x` will always be 1 (as that's the only possibility), set `ans` to the value of this node (1).
   - Visit the second node (value = 2): `n=2`. Generate a random number `x` between 1 and 2. If `x` is 2, set `ans` to 2. Otherwise, `ans` remains 1.
   - Visit the third node (value = 3): `n=3`. Generate a random number `x` between 1 and 3. If `x` is 3, set `ans` to 3, replacing the older value. If not, `ans` stays the same.
   - Visit the fourth node (value = 4): `n=4`. Generate a random number between 1 and 4. If `x` is 4, `ans` becomes 4. If not, no change.
   - Visit the fifth node (value = 5): `n=5`. Generate a random number between 1 and 5. If `x` is 5, `ans` changes to 5. If not, `ans` does not change.

After visiting all nodes, `ans` will hold the value of a node that has been randomly selected, adhering to the requirement that each node has an equal chance (1/5 in this case) of being chosen. Since complete traversal is necessary, each node's probability of being the final answer is even, ensuring the randomness of the process.

Python Solution

```
import random

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

class Solution:

    # Constructor which initializes the list head.
    def __init__(self, head: Optional[ListNode]):
        self.head = head

    # Returns a random node's value from the linked list.
    def getRandom(self) -> int:
        # Counter to keep track of the total nodes visited so far
        count_nodes = 0
        # variable to store the randomly selected node's value
        selected_value = None
        # Iterate from the head of the linked list
        current_node = self.head

        # Traverse through the list
        while current_node:
            count_nodes += 1  # Increment the counter for each node
            # Generate a random number between 1 and the current node count
            random_number = random.randint(1, count_nodes)
            # If random number equals the current node index, update the selected value
            if random_number == count_nodes:
                selected_value = current_node.val
            # Move to the next node
            current_node = current_node.next

        # Return the randomly selected node's value
        return selected_value

# The following instantiation and method calls are used to operate on the `Solution` class:
# obj = Solution(head)
# param_1 = obj.getRandom()
```

Java Solution

```
import java.util.Random;

// 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 {
    private ListNode head;
    private Random randomGenerator = new Random();

    // Constructor which initializes the head of the linked list.
    public Solution(ListNode head) {
        this.head = head;
    }

    // Iterate through the linked list to select a random node.
    for (ListNode currentNode = head; currentNode != null; currentNode = currentNode.next) {
        // Increment the count of nodes visited.
        int randomNumber = randomGenerator.nextInt(i); // Generate a random number between 1 and i.

        // If the generated random number equals the number of nodes visited, update the answer.
        if (randomNumber == i) {
            randomValue = currentNode.val;
        }
    }

    // Return the randomly selected node's value.
    return randomValue;
}

// The following class simulates the way the Solution object would be instantiated and called in a client code:
class Example {
    public static void main(String[] args) {
        ListNode listHead = new ListNode(1); // Create a sample linked list.
        listHead.next = new ListNode(2);
        listHead.next.next = new ListNode(3);

        Solution solution = new Solution(listHead); // Instantiate the Solution object with the linked list.
        int randomNodeValue = solution.getRandom(); // Call the getRandom method.
        System.out.println("Random node's value: " + randomNodeValue);
    }
}
```

C++ Solution

```
#include <cstdlib> // For `rand()`

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x = 0, ListNode *next = nullptr) : val(x), next(next) {}
};

class Solution {
public:
    ListNode* head; // Pointer to the head of the linked list.

    // Constructor
    Solution(ListNode* head) {
        this->head = head;
    }

    // Returns a random node's value from the linked list.
    int getRandom() {
        int scope = 0; // Represents the number of nodes seen so far.
        int chosenValue = 0; // Value of the randomly chosen node.

        // Initialize a moving pointer to traverse the linked list.
        ListNode* currentNode = head;

        // Traverse the entire list.
        while (currentNode != nullptr) {
            scope += 1; // Increase the scope since we are seeing a new node.

            // Generate a random number in [1, scope] range.
            int randomNumber = 1 + rand() % scope;

            // With probability 1/scope, choose the current node's value.
            if (randomNumber == scope) {
                chosenValue = currentNode->val;
            }

            // Move to the next node in the list.
            currentNode = currentNode->next;
        }

        // Return the chosen value.
        return chosenValue;
    }
};

/**
 * How to use the Solution class:
 *
 * ListNode* head = new ListNode(1);
 * head->next = new ListNode(2);
 * head->next->next = new ListNode(3);
 * Solution* solution = new Solution(head);
 * int randomValue = solution->getRandom(); // Gets a random value from the list.
 */
```

Typescript Solution

```
class ListNode {
  val: number;
  next: ListNode | null;

  constructor(val: number = 0, next: ListNode | null = null) {
    this.val = val;
    this.next = next;
  }
}

let head: ListNode | null; // Global variable pointing to the head of the linked list

// Function to initiate the list with a head node
function createList(headValue: number): ListNode {
  head = new ListNode(headValue);
  return head;
}

// Function to add a node to the linked list
function addNode(newValue: number) {
  let newNode = new ListNode(newValue);
  if (head === null) {
    head = newNode;
  } else {
    let current = head;
    while (current.next !== null) {
      current = current.next;
    }
    current.next = newNode;
  }
}

// Function that returns a random node's value from the linked list
function getRandom(): number {
  let scope = 0; // Represents the number of nodes seen so far
  let chosenValue = 0; // Value of the randomly chosen node
  let currentNode = head; // Initialize a moving pointer to traverse the linked list

  // Traverse the entire list
  while (currentNode !== null) {
    scope += 1; // Increase the scope as we see a new node

    // Generate a random number in [1, scope] range
    let randomNumber = 1 + Math.floor(Math.random() * scope);

    // With probability 1/scope, choose the current node's value
    if (randomNumber === scope) {
      chosenValue = currentNode.val;
    }

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

  // Return the chosen value
  return chosenValue;
}

/**
 * How to use the getRandom function:
 *
 * head = createList(1);       // Initializes the list with the head node
 * addNode(2);                 // Adds a node with value 2
 * addNode(3);                 // Adds a node with value 3
 * let randomValue = getRandom(); // Gets a random value from the list
 */
```

Time and Space Complexity

The given Python class implements a method to randomly get an element from a singly-linked list.

Time Complexity

The `getRandom()` method has a time complexity of $O(n)$. This is because it processes each node in the singly-linked list exactly once in a sequential manner. The variable n represents the total number of nodes in the list. The while loop iterates over all n nodes, performing a constant amount of work for each node (generating a random number and performing a comparison).

Space Complexity

The `getRandom()` method has a space complexity of $O(1)$. The extra space used by the method does not depend on the size of the input linked list. The variables `n`, `ans`, `head`, and `x` use a constant amount of space regardless of the number of nodes in the list.