341. Flatten Nested List Iterator

Medium Stack Tree Depth-First Search Design Queue Iterator

Problem Description

The problem presents a data structure that consists of a list where each element can either be a single integer or a nested list of integers. Such nested lists can have multiple levels of inner lists containing integers. The task is to create an iterator that traverses this complex structure sequentially, effectively flattening it, so that all the individual integers get accessed one by one in the order they appear, from left to right and from topmost level to the deepest level without any concern for the nested structure.

This requires designing a class, `NestedIterator`, that keeps track of the integers in these nested lists, allowing the user to repeatedly call `next()` to get the next integer and `hasNext()` to check if more integers are available for retrieval. To check for correctness, the iterator is used to extract all the integers into a flat list `res`, and if `res` matches the expected output (i.e., the list with the same integers in the same order, but without any nested structure), then the implementation is correct.

Intuition

The core of the solution lies in using Depth-First Search (DFS) to traverse the nested list structure before we start iterating. The reason for selecting DFS is that it naturally follows the order and depth in which the integers are stored within the nested lists. It can reach the deepest elements first and backtrack to explore other branches, which is perfect for capturing all elements in the required order.

We start by initializing the iterator with the nested list. During the initialization, we perform a DFS to traverse all elements within the `nestedList`. If the current element is an integer, we append it to the `vals` list, which is a flat list containing all the nested integers in the correct order. If the element is a nested list, we recursively apply the same DFS process to that list.

Once the DFS is complete, `vals` will contain all the integers in a flat structure, and we're ready to iterate over them using our `next()` and `hasNext()` methods. The `next()` method returns the next integer by accessing the current index in `vals` and increments the index for the next call. The `hasNext()` method simply checks if the current index is less than the length of `vals`, indicating that there are more elements to be iterated over.

Solution Approach

The solution is implemented in Python and revolves around the concept of Depth-First Search (DFS) to traverse and flatten the nested list structure. Depth-First Search is an algorithm that starts at the root (in this case, the first list or integer in the nested list) and explores as far as possible along each branch before backtracking.

Data Structure

- A list called `self.vals` is used to store all integers from the nested list in a flattened form after the DFS traversal, and `self.cur` keeps the current index of the next integer to return.

Algorithm

1. A nested function `dfs(nestedList)` is defined within the `__init__` method of the `NestedIterator` class to perform a depth-first traversal of the input `nestedList`. This is a recursive function that:
   - Checks if the current element is an integer by calling `e.isInteger()`. If it is, the integer is appended to `self.vals`.
   - If the element is another list, the function calls itself with `e.getList()`, continuing the DFS on the nested list.
2. The `__init__` method initializes the `self.vals` list to store the flattened integers and sets the `self.cur` index to 0. It then calls the `dfs(nestedList)` to fill `self.vals` using the DFS traversal explained above.
3. The `next()` method is responsible for returning the next integer in the `self.vals` list. It stores the value at the current index (`self.cur`), increments `self.cur` to point to the next integer, and returns the stored value.
4. The `hasNext()` method simply checks if `self.cur` is less than the length of `self.vals`, which determines if there are any more integers to iterate through.

Patterns

- The main pattern used here is the iterator pattern, which provides a way to access the elements of a collection without exposing its underlying representation. The `next()` and `hasNext()` methods are classic examples of this pattern and allow users to iterate over the collection one element at a time.

By using recursive DFS, any nesting of lists within lists is handled elegantly, ensuring that integers are discovered in the correct order. Once the DFS is complete, the resulting `self.vals` becomes a simple flat list that the `next()` and `hasNext()` methods can easily navigate.

Example Walkthrough

Let's say we have the following nested list as an example to illustrate the solution approach:

```
nestedList = [[1, 1], 2, [1, 1]]
```

Our goal is to flatten `nestedList` using the `NestedIterator` class so that we can iterate over all the integers sequentially.

1. First, let's visualize the depth-first traversal:
   - Start with the first element, which is a nested list `[1, 1]`.
   - The DFS will go into this nested list and append both `1`s to `self.vals`.
   - Backtrack to the next element which is `2`, append it to `self.vals`.
   - Move to the last element, which is another nested list `[1, 1]`, and append both `1`s to `self.vals`.
2. After initialization and DFS traversal, our `self.vals` list inside the `NestedIterator` will look like this:

   ```
   self.vals = [1, 1, 2, 1, 1]
   ```

   And the `self.cur` index used to keep track of the current position will be initialized to `0`.
3. The `next()` method is called. Since `self.cur` is `0`, the first element in `self.vals` is returned which is `1`, and `self.cur` is incremented to `1`.
4. The `next()` method is called again. Now `self.cur` is `1`, the second element in `self.vals` is returned which is also `1`, and `self.cur` is incremented to `2`.
5. This process continues each time `next()` is called. When `next()` is called for the third time, `self.cur` is at `2`, and `self.vals[2]` which is `2`, is returned, with `self.cur` incremented to `3`.
6. When the `hasNext()` method is called at any point, it checks if `self.cur` is less than the length of `self.vals`. As long as `self.cur` is less than `5` in this case, `hasNext()` will return `true`, indicating that there are more elements to be iterated over.
7. The process continues until `self.cur` equals the length of `self.vals` (in this case, `5`), at which point `hasNext()` will return `false`, signifying that we have reached the end of our flattened list.

By following this approach, the `NestedIterator` class effectively flattens the nested structure of the input list and allows for an easy sequential iteration over the integers using the DFS technique.

Python Solution

```python
class NestedIterator:
    def __init__(self, nestedList: [NestedInteger]):
        # A depth-first search function to flatten the nested list
        def flatten(nested_list):
            for element in nested_list:
                if element.isInteger():
                    # If the element is an integer, append it directly to flat_list
                    self.flat_list.append(element.getInteger())
                else:
                    # If the element is a list, recursively call flatten on it
                    flatten(element.getList())

        self.flat_list = []  # A list to store the flattened elements
        self.index = 0  # An index to track the current position in flat_list
        flatten(nestedList)  # Initialize by starting the flattening process

    def next(self) -> int:
        # Returns the next integer in the flat_list and increments the index
        value = self.flat_list[self.index]
        self.index += 1
        return value

    def hasNext(self) -> bool:
        # Check if there are more integers to iterate over
        return self.index < len(self.flat_list)


# Your NestedIterator object will be instantiated and called as such:
# i, v = NestedIterator(nestedList), []
# while i.hasNext(): v.append(i.next())

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class NestedIterator implements Iterator<Integer> {

    // A list to hold all integers gathered from the nested list.
    private List<Integer> flattenedList;

    // An iterator to iterate through the flattened list of integers.
    private Iterator<Integer> flatListIterator;

    /**
     * Constructor which takes a list of NestedInteger objects and
     * initializes the iterator after flattening the list.
     * @param nestedList a list of NestedInteger objects to be flattened.
     */
    public NestedIterator(List<NestedInteger> nestedList) {
        flattenedList = new ArrayList<>();
        // Flatten the nested list using depth-first search.
        flattenList(nestedList);
        // Initialize iterator for the flattened list.
        flatListIterator = flattenedList.iterator();
    }

    /**
     * Returns the next integer in the nested list.
     * @return the next integer.
     */
    @Override
    public Integer next() {
        return flatListIterator.next();
    }

    /**
     * Determines if there are more integers to return from the nested list.
     * @return true if there are more integers to return, false otherwise.
     */
    @Override
    public boolean hasNext() {
        return flatListIterator.hasNext();
    }

    /**
     * Helper method to flatten a list of NestedInteger objects using a depth-first search approach.
     * @param nestedList a list of NestedInteger to be flattened.
     */
    private void flattenList(List<NestedInteger> nestedList) {
        for (NestedInteger element : nestedList) {
            // Check if the NestedInteger is a single integer.
            if (element.isInteger()) {
                // Add integer to flattened list.
                flattenedList.add(element.getInteger());
            } else {
                // If it is a nested list, then recur.
                flattenList(element.getList());
            }
        }
    }
}

/**
 * Examples of how the NestedIterator class could be used:
 * NestedIterator iterator = new NestedIterator(nestedList);
 * while (iterator.hasNext()) {
 *     v[f()] = iterator.next();
 * }
 */

```

C++ Solution

```cpp
class NestedIterator {
public:
    // Constructor initializes the iterator with the given nested list
    NestedIterator(vector<NestedInteger> &nestedList) {
        flattenList(nestedList);
    }

    // Returns the next integer in the nested list
    int next() {
        return flattenedList[currentIndex++];
    }

    // Returns true if there are more integers to be iterated over
    bool hasNext() const {
        return currentIndex < flattenedList.size();
    }

private:
    vector<int> flattenedList; // Flattened list of integers
    size_t currentIndex = 0;   // Current index in the flattened list

    // Helper function to flatten a nested list into a single list of integers
    void flattenList(const vector<NestedInteger> &nestedList) {
        for (const auto &element : nestedList) {
            if (element.isInteger()) {
                flattenedList.push_back(element.getInteger());
            } else {
                flattenList(element.getList());
            }
        }
    }
};

// Usage:
// NestedIterator i(nestedList);
// while (i.hasNext()) cout << i.next();

```

Typescript Solution

```typescript
/** This is the given interface for NestedInteger with explanations */
interface NestedInteger {
    // Constructor may hold a single integer
    constructor(value?: number): void;

    // Returns true if this NestedInteger holds a single integer
    isInteger(): boolean;

    // Returns the single integer this NestedInteger holds, or null if it holds a nested list
    getInteger(): number | null;

    // Sets this NestedInteger to hold a single integer
    setInteger(value: number): void;

    // Sets this NestedInteger to hold a nested list and adds a nested integer to it
    add(elem: NestedInteger): void;

    // Returns the nested list this NestedInteger holds, or an empty list if it holds a single integer
    getList(): NestedInteger[];
}

// Array to hold the flattened list of integers
let flatList: number[] = [];

// Index to track the current position in the flat list
let currentIndex: number = 0;

/**
 * Constructor that takes a NestedInteger list and flattens it.
 */
function nestedIteratorConstructor(nestedList: NestedInteger[]): void {
    currentIndex = 0;
    flatList = [];
    flattenList(nestedList);
}

/**
 * Helper function to flatten a nested list.
 * Recursively traverses the input and stores integers in flatList.
 */
function flattenList(nestedList: NestedInteger[]): void {
    for (const item of nestedList) {
        if (item.isInteger()) {
            flatList.push(item.getInteger());
        } else {
            flattenList(item.getList());
        }
    }
}

/**
 * Returns true if the iterator has more elements, false otherwise.
 */
function hasNext(): boolean {
    return currentIndex < flatList.length;
}

/**
 * Returns the next element in the iteration and advances the iterator.
 */
function next(): number {
    return flatList[currentIndex++];
}

// Example usage:
// nestedIteratorConstructor(nestedList);
// const result: number[] = [];
// while (hasNext()) result.push(next());

```

Time and Space Complexity

Time Complexity

The constructor of the `NestedIterator` class involves a Depth-First Search (DFS) through the entire nested list. The time complexity for this operation is `O(N)`, where `N` is the total number of integers within the nested structure. This is because we need to visit every nested element and integer once to flatten the structure into the `vals` list.

The `next()` method has a time complexity of `O(1)` for each call, as it simply accesses the next element in the flattened `vals` list and increments the `cur` pointer.

The `hasNext()` method also works in `O(1)` time as it only checks if the `cur` pointer is less than the length of the `vals` list.

Therefore, considering all method calls, the time complexity is `O(N)` for the entire iteration over the nested structure due to the initial DFS.

Space Complexity

The space complexity for the DFS in the constructor is `O(L)`, where `L` is the maximum depth of nesting in the input, due to the stack space used by the recursive calls of the `dfs()` function.

Additionally, the space complexity for storing the flattened list of integers is `O(N)`, where `N` is the total number of integers.

Thus the overall space complexity of the `NestedIterator` is `O(N + L)`. In the case where there is no nesting, `L` would be `O(1)`, making the space complexity purely `O(N)`.