Solution: Flatten Nested List Iterator

Let's solve the Flatten Nested List Iterator problem using the Stacks pattern.

We'll cover the following

Statement

You're given a nested list of integers. Each element is either an integer or a list whose elements may also be integers or other integer lists. Your task is to implement an iterator to flatten the nested list.

You will have to implement the **Nested Iterator** class. This class has the following functions:

- **Constructor:** This initializes the iterator with the nested list.
- **Next ():** This returns the next integer in the nested list.
- **Has Next ():** This returns TRUE if there are still some integers in the nested list. Otherwise, it returns FALSE.

Constraints

- The nested list length is between 1 and 500.
- The nested list consists of integers between -10^6 and 10^6.

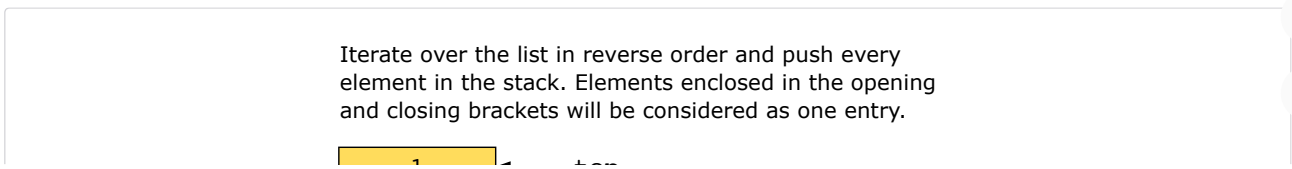
Solution

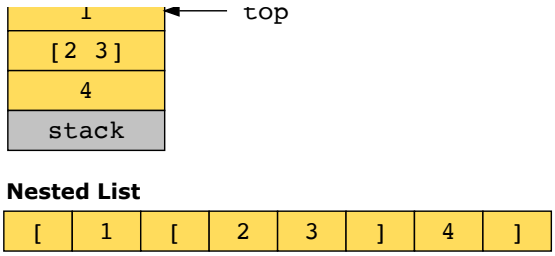
We'll use a stack to solve this problem. The stack will be used to store the integer and list of integers on the iterator object. We'll push all the nested list data in the stack in reverse order in the constructor. The elements are pushed in reverse order because the iterator is implemented using a stack. In order to process the nested list correctly, the elements need to be accessed in the order they appear in the original nested list.

The **Has Next** function performs a set of push and pop operations on the stack in the form of a loop. It checks if the top element of the stack is an integer. If so, it returns TRUE. Otherwise, if the top element is a list of integers, then it pops from the stack and pushes each element of the list in the stack in reverse order. This way, the lists at the top of the stack are converted into individual integers whenever the **Has Next** function is called. If the stack is empty, the function returns FALSE.

The **Next** function first calls the **Has Next** function to check if there is an integer in the stack. If the **Has Next** function returns TRUE, it pops from the stack and returns this popped value.

Here is an example of how this function works:





1 of 7

Let's look at the code for this solution:

Java

main.java

NestedInteger.java

```java
class NestedIterator {
  private Stack<NestedInteger> stack;
  // NestedIterator constructor inializes the stack using the
  // given nestedList list
  public NestedIterator(List<NestedInteger> nestedList) {
    this.stack = new Stack<NestedInteger>();
    for (int i = nestedList.size() - 1; i >= 0; i--) {
      this.stack.push(nestedList.get(i));
    }
  }

  // hasNext() will return True if there are still some integers in the
  // stack (that has nested_list elements) and, otherwise, will return False.
  public boolean hasNext() {
    // Iterate in a stack until stack is not empty
    while (!stack.isEmpty()) {
      // Save the top value of the stack
      NestedInteger top = stack.peek();
      // Check if the top value is integer, if true return True,
      // if not continue
      if (top.isFile()) {
        return true;
      }
      // If the top is not an integer, it must be the list of integers
      // Pop the list from the stack and save it in the top_list
      List<NestedInteger> topList = stack.pop().getList();
      for (int i = topList.size() - 1; i >= 0; i--) {
        // Append the values of the nested list into the stack
```

Flatten Nested List Iterator

Assume n is the number of elements, l is the number of nested lists, and d is the maximum nesting depth (maximum number of lists inside each other).

Constructor: Since the constructor pushes all of the elements from the nested list into the stack, the total time will be the size of that list. Therefore, the time complexity will be $O(n + l)$.

Has Next (): This function will be called several times. During all of these calls, the function will iterate over all the lists exactly once (the while loop) and process every integer exactly once. Thus, a total of $O(n + l)$ effort is spent. The iterator will be progressed for every integer in the nested list. Thus, there will be a total of $O(n)$ calls. Accordingly, the running time for one call to **Has Next** is $O((n + l)/n) = O(l/n)$.

Next (): This function calls the **Has Next** function every time. So, its complexity will be the same as that of the **Has Next** function, that is, $O(l/n)$.

Space complexity

The space complexity will be $O(n + l)$.

In the worst-case scenario, whereby the outermost list contains n integers or l empty sub-lists, it will cost $O(n + l)$ space. Other expensive cases occur when the nesting is very deep. It's useful to remember that $d \leq l$ (because each layer of nesting requires another list), but we don't need to consider this for our case.

Back

Flatten Nested List Ite...

Next

Valid Parentheses

Mark as Completed