439. Ternary Expression Parser

Medium Stack Recursion String

Problem Description

The problem requires us to evaluate a ternary expression given as a string. A ternary expression is of the form "X ? Y : Z" which means "if X is true, then Y, otherwise Z". The expression provided will be a nested ternary expression where the result of one ternary can be a condition for other ternaries. The challenge is to process the expression which may be heavily nested and produce a single output.

An expression can only contain one-digit numbers, 'T' for true, 'F' for false, '?' as a delimiter between condition and true case, and ':' as a delimiter between true case and false case. Expressions are evaluated from right to left, and we are ensured the expression is always valid.

For example, given an expression "T?T?F:5:3", the result of evaluating this expression should be "F", as we evaluate "T?F:5" first which yields "F", then "T?F:3" which yields "F".

Intuition

The intuition behind the solution comes from understanding how a stack can help in evaluating nested expressions. A stack is a simple data structure that allows us to add elements to the top and remove elements from the top; Last-In, First-Out (LIFO). We reverse the expression and iterate over it, using a stack to keep track of operands and operators. When we encounter a number or 'T'/'F', we push it onto the stack. When we encounter a '?', we know the top two elements on the stack are operands for the ternary operation, and we evaluate them based on the condition just before the '?'. We throw away the false case and keep only the true case result on the stack right away as the problem simplifies to evaluating the rightmost expression first due to the right-to-left grouping.

By continuously evaluating the ternary operations as we encounter them and keeping track of results on the stack, by the time we reach the beginning of the expression (since we iterate from the end), we should have the final result on top of the stack.

The provided solution algorithm correctly implements this intuition. When a ':' is encountered, it is simply ignored because our stack already contains the operands it needs. The '?' indicates that an evaluation must be performed. If the condition is 'T', we pop once from the stack to discard the false case and the top of the stack now becomes the true case (which is our result). If the condition is 'F', we pop twice to discard the true case and then the false case becomes our result. Either way, we reset the `cond` flag to False until the next '?' is encountered.

Solution Approach

The solution uses a stack data structure to evaluate the expression in a right-to-left manner, which is suitable for the right-to-left grouping nature of ternary expressions. The algorithm proceeds as follows:

1. Initialize an empty stack called `stk`.
2. Initialize a boolean variable `cond` as False. This variable will be used as a flag to know when we're ready to evaluate a ternary expression.
3. Reverse the input `expression` and iterate over it character by character using a for-loop. Reversing is crucial because the conditions and their corresponding true-false values are pushed onto the stack in such a way that they can be popped off in the correct order when evaluating.
4. Ignore the case when the current character is a `':'`. This character simply separates the true and false case, which we don't need to keep track of since the stack already contains the necessary cases for evaluation.
5. When the current character is a `'?'`, set the `cond` flag to True. This indicates that the next non-':' character will be a condition whose result determines which of the two values on top of the stack to keep.
6. For any other character (which could be a digit, `'T'`, or `'F'`):
   - If `cond` is True, this means the current character is a condition (either `'T'` or `'F'`). Based on its value:
     - If the condition is `'T'`, pop once from the stack to remove the false case (as it is not needed) and the remaining top is the true case, which becomes our result for the current ternary.
     - If the condition is `'F'`, pop twice from the stack; once to remove the true case and once to remove the false case and then append this false case back onto the stack, as it is the result of the ternary.
   - If `cond` is False, append the current character onto the stack because it's a value, not a condition.
   - Reset the `cond` flag to False after the conditional check, since the ternary expression has been evaluated and the result is on the stack.
7. Continue the above process until the entire string has been iterated over. Since the evaluation is always complete right after a `'?'` or when a value is encountered (and the stack is not ready for evaluation), the top of the stack at the end of the iteration will hold the final result.
8. Return the top of the stack, which is the evaluated result of the entire expression.

Through this implementation, we effectively mimic the parsing and evaluating of a nested ternary expression while using the stack to keep track of pending operations. Each character in the reversed string contributes to the formation of the expression as the ternary operations are resolved.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the ternary expression "F?1:T?4:5".

Following the solution approach:

1. A stack `stk` is initialized as empty.
2. A boolean variable `cond` is initialized as False.
3. Reverse the expression to "5:4?T:1?F".
4. Iterating over the reversed expression:
   - '5' is encountered: `cond` is False, so '5' is pushed onto `stk`.
   - ':' is a separator, so we do nothing.
   - '4' is encountered next: `cond` is False, push '4' onto `stk`.
   - '?' is encountered: `cond` is set to True, indicating that the next non-':' character will be a condition.
   - 'T' is a condition: Since `cond` is True and the condition is 'T', pop once from `stk` (remove '5'), so '4' is the result for this ternary operation. Reset `cond` to False.
   - ':' is another separator, do nothing.
   - '1' is encountered: `cond` is False, push '1' onto `stk`.
   - '?' is encountered: Set `cond` to True.
   - 'F' is a condition: Now, `cond` is True and the condition is 'F'. Pop twice from `stk` to remove '4' (true case) and '1' (false case), then push '1' back onto `stk`, as it is the result of this ternary. Reset `cond` to False.
5. At the end of the iteration, the stack `stk` has one element, which is '1', the evaluated result of the entire expression.

Therefore, the given ternary expression "F?1:T?4:5" evaluates to '1'.

Solution Implementation

Python

```python
class Solution:
    def parseTernary(self, expression: str) -> str:
        # Initialize a stack to keep track of characters and results.
        stack = []
        # This variable is used as a flag to know when a condition (T or F) is encountered
        is_condition = False

        # Loop through the expression in reverse order.
        for char in expression[::-1]:
            # Skip over the ':' character as it's just a separator.
            if char == ':':
                continue
            # If we encounter a '?', we know the next character is a condition (T or F).
            if char == '?':
                is_condition = True
            else:
                # If we're dealing with a condition, resolve the ternary operation.
                if is_condition:
                    # If the condition is 'T' (true), pop and use the first result (left-hand operand).
                    if char == 'T':
                        true_result = stack.pop()
                        # Pop and discard the second result (right-hand operand) as it's not needed.
                        stack.pop()
                        stack.append(true_result)
                    # If condition is 'F' (false), skip the first result and pop the second (right-hand operand).
                    else:
                        stack.pop()
                        false_result = stack.pop()
                        stack.append(false_result)
                    # Reset the condition flag as we've resolved this ternary operation.
                    is_condition = False
                else:
                    # If we're not handling a condition, push the character onto the stack.
                    stack.append(char)
        # The final result will be at the top of the stack, so return it.
        return stack[0]
```

Java

```java
import java.util.Deque;
import java.util.ArrayDeque;

class Solution {

    // Method to parse the given ternary expression and return the result as a string
    public String parseTernary(String expression) {
        // Initialize a stack to help evaluate the ternary expressions from right to left
        Deque<Character> stack = new ArrayDeque<>();
        // Variable to keep track of when the '?' symbol is encountered, indicating a condition
        boolean isCondition = false;

        // Iterate over the expression from end to start
        for (int i = expression.length() - 1; i >= 0; i--) {
            char currentChar = expression.charAt(i);
            if (currentChar == ':') {
                continue; // Skip ':' as it does not affect stack operations
            }
            if (currentChar == '?') {
                isCondition = true; // Set the condition flag upon finding '?'
            } else {
                if (isCondition) {
                    // Evaluate the ternary condition based on the current character 'T' or 'F'
                    if (currentChar == 'T') {
                        // If condition is true, pop and keep the first value
                        char trueValue = stack.pop();
                        stack.pop(); // Discard the false value
                        stack.push(trueValue); // Push the true value back onto the stack
                    } else {
                        // If condition is false, simply discard the true value
                        stack.pop(); // Discard the true value
                        // The top now contains the false value, which is the desired result
                    }
                    isCondition = false; // Reset condition flag for the next evaluation
                } else {
                    // Push the current character onto the stack if not part of a condition
                    stack.push(currentChar);
                }
            }
        }
        // The top of the stack now contains the result of the ternary expression
        return String.valueOf(stack.peek());
    }
}
```

C++

```cpp
#include <algorithm>
#include <string>

class Solution {
public:
    // Function to parse a ternary expression and return the resulting string.
    string parseTernary(string expression) {
        // Use a string as a stack to keep track of the characters.
        string stack;

        // A flag to know when we're evaluating a condition.
        bool isCondition = false;

        // Reverse the expression to evaluate it from right to left.
        std::reverse(expression.begin(), expression.end());

        // Iterate through each character of the reversed expression.
        for (char& currentChar : expression) {
            if (currentChar == ':') {
                // Ignore colons as they only separate expressions, not required in evaluation.
                continue;
            }
            if (currentChar == '?') {
                // If we find a '?', the next character is a condition.
                isCondition = true;
            } else {
                if (isCondition) {
                    // If we're at a condition, check if it is 'T' (True).
                    if (currentChar == 'T') {
                        // If True, pop the false result and keep the true result.
                        char trueResult = stack.back();
                        stack.pop_back();
                        stack.pop_back(); // Remove the false result.
                        stack.push_back(trueResult); // Keep the true result.
                    } else {
                        // If False ('F'), remove the true result, keeping the false one.
                        stack.pop_back(); // The true result is removed.
                    }
                    isCondition = false; // Reset the flag as we've resolved the condition.
                } else {
                    // If it's a regular character, add it to the stack.
                    stack.push_back(currentChar);
                }
            }
        }

        // The final result will be at the top of the stack.
        return {stack.back()};
    }
};
```

TypeScript

```typescript
function parseTernary(expression: string): string {
    // A stack to hold the characters for processing the expression
    let stack: string[] = [];

    // Variable to mark whether we are evaluating a condition
    let evaluatingCondition: boolean = false;

    // Reverse the expression to evaluate from right to left
    let reversedExpression: string = expression.split('').reverse().join('');

    // Iterate through each character of the reversed expression
    for (let i = 0; i < reversedExpression.length; i++) {
        let currentChar: string = reversedExpression[i];

        if (currentChar === ':') {
            // Colons are separators and do not affect the evaluation
            continue;
        } else if (currentChar === '?') {
            // The next character is a condition
            evaluatingCondition = true;
        } else {
            if (evaluatingCondition) {
                // If evaluating a condition, check if it's 'T' for True
                if (currentChar === 'T') {
                    // If True, the top of the stack is the true result, pop and discard the next (false) result
                    let trueResult: string = stack.pop()!;
                    stack.pop(); // Remove the false result
                    stack.push(trueResult); // Push the true result back onto the stack
                } else {
                    // If False ('F'), simply pop the true result leaving the false one at the top of the stack
                    stack.pop(); // The true result is discarded
                }
                // Reset evaluating condition flag after processing
                evaluatingCondition = false;
            } else {
                // If it's not a condition, push the character onto the stack for later evaluation
                stack.push(currentChar);
            }
        }
    }

    // The final result will be at the top of the stack
    return stack.pop()!;
}

// Example usage
// const result = parseTernary("T?2:3"); // Should return "2"
```


```python
class Solution:
    def parseTernary(self, expression: str) -> str:
        # Initialize a stack to keep track of characters and results.
        stack = []
        # This variable is used as a flag to know when a condition (T or F) is encountered
        is_condition = False

        # Loop through the expression in reverse order.
        for char in expression[::-1]:
            # Skip over the ':' character as it's just a separator.
            if char == ':':
                continue
            # If we encounter a '?', we know the next character is a condition (T or F).
            if char == '?':
                is_condition = True
            else:
                # If we're dealing with a condition, resolve the ternary operation.
                if is_condition:
                    # If the condition is 'T' (true), pop and use the first result (left-hand operand).
                    if char == 'T':
                        true_result = stack.pop()
                        # Pop and discard the second result (right-hand operand) as it's not needed.
                        stack.pop()
                        stack.append(true_result)
                    # If condition is 'F' (false), skip the first result and pop the second (right-hand operand).
                    else:
                        stack.pop()
                        false_result = stack.pop()
                        stack.append(false_result)
                    # Reset the condition flag as we've resolved this ternary operation.
                    is_condition = False
                else:
                    # If we're not handling a condition, push the character onto the stack.
                    stack.append(char)
        # The final result will be at the top of the stack, so return it.
        return stack[0]
```


Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input string `expression`. This complexity arises because the code traverses the entire input in reverse once, performing a constant amount of work for each character. Each character is processed only once and the operations inside the loop (like pop and append on the stack) are `O(1)`.

The space complexity of the code is also `O(n)`. In the worst case, all characters besides `":"` and `"?"` might end up on the stack, which means the stack can grow up to the size of the input string when the conditional expressions are nested but always evaluating to the false branch (e.g., "F?F?F?...:a:a:a"). However, in most cases, the stack will contain fewer elements than `n`.