Solution: Minimum Remove to Make Valid Parentheses

Let's solve the Minimum Remove to Make Valid Parentheses problem using the Stacks pattern.

We'll cover the following

Statement

Given a string, `s`, that may have matched and unmatched parentheses, remove the minimum number of parentheses so that the resulting string represents a valid parenthesization. For example, the string "a(b)" represents a valid parenthesization while the string "a(b" doesn't, since the opening parenthesis doesn't have any corresponding closing parenthesis.

Constraints:

- $1 \leq$ `s.length` $\leq 10^5$
- `s[i]` is either an opening parenthesis (, a closing parenthesis), or a lowercase English letter.

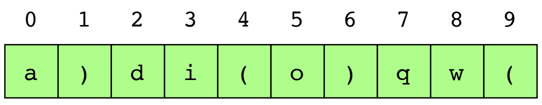
Solution

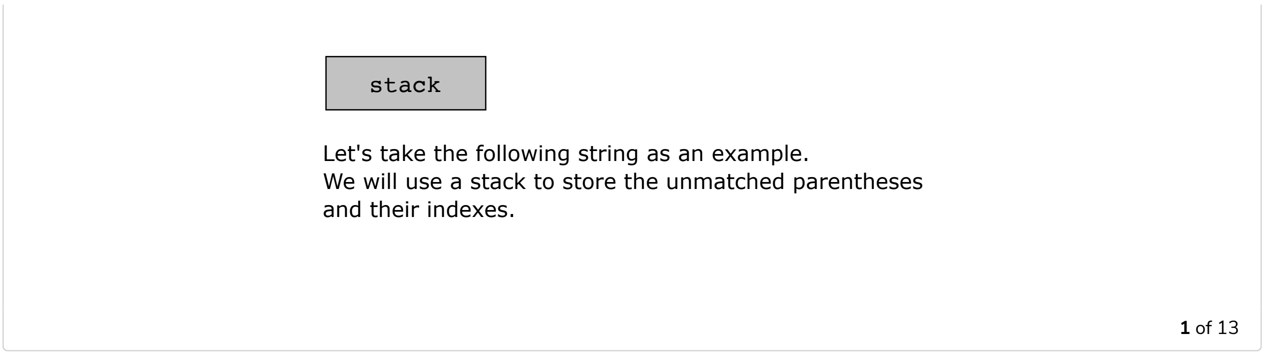
In this solution, we will use a stack with its method, LIFO, to remove all the extra parentheses from the input string. We traverse the input string, and every time we encounter an opening parenthesis, we push it, along with its index, onto the stack and keep traversing. Meanwhile, whenever we find a closing parenthesis, we decide whether to push it onto the stack or not. For this, we check the stack as follows:

- If the stack is not empty and the top stack element is an opening parenthesis, we pop it off. This represents that the recently removed opening parenthesis corresponds to the current closing parenthesis making a valid set of parenthesis in the input string.
- If the stack is empty or the top stack element is a closing parenthesis, we push the current closing parenthesis, along with its index, onto the stack.

After traversing the complete string, all the parentheses left in the stack are invalid. Since we have stored each parenthesis index as well, we can now use these index values to remove the instances of these parentheses in the input string. Return the updated string as the output, which should represent the valid parenthesization.

The slides below illustrate the steps of the solution in detail:





Let's take the following string as an example.
We will use a stack to store the unmatched parentheses
and their indexes.

1 of 13

Let’s look at the code for this solution:

Java

```java
import java.util.*;

// Utility class for Pair
class Pair<F, S> {
        F first;
        S second;

        Pair(F first, S second) {
            this.first = first;
            this.second = second;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) {
                return true;
            }
            if (obj == null || getClass() != obj.getClass()) {
                return false;
            }
            Pair<?, ?> pair = (Pair<?, ?>) obj;
            return first.equals(pair.first) && second.equals(pair.second);
        }
```

```java
        public int hashCode() {
            return 31 * first.hashCode() + second.hashCode();
        }
```

Time complexity

In the worst case, we might need to traverse the entire string once, which takes $O(n)$ time, where n is the length of the input string. For each character, the operations performed are constant time $O(1)$, such as pushing and popping elements from the stack, as well as accessing the top element of the stack. Therefore, the overall time complexity of the solution above is $O(n)$.

Space complexity

In the worst case, all the characters in the string can be unmatched delimiters, for example: “)))(((”. This makes the stack’s depth equal to the string’s length. Therefore, the space complexity is $O(n)$.

Back
Minimum Remove to ...

Next
Exclusive Execution Ti...

✓ Mark as Completed