To determine if an expression has redundant parentheses, we can use a **stack**. The core idea is that a pair of parentheses is redundant if it encloses a valid sub-expression that could exist on its own without the outer parentheses. This means the sub-expression within the parentheses does not contain any operators.

We can iterate through the expression character by character.

- When we encounter an opening parenthesis `'('` or an operator (`+`, `-`, `*`, `/`), we push it onto the stack. These are parts of a sub-expression that need to be processed later.
- When we encounter an operand (a letter like 'a' or 'b'), we don't push it onto the stack, as it simply represents a value.
- When we encounter a closing parenthesis `')'`, this is where the main logic happens. We'll check the items on the stack. We pop elements from the stack until we find a matching opening parenthesis `'('`.
    - As we pop elements, we keep track of whether we've encountered any **operators** within this sub-expression. We can use a boolean flag for this, say `hasOperator`, initialized to `false`.
    - If, after popping all elements up to the matching `'('`, the `hasOperator` flag is still `false`, it means the sub-expression enclosed by this pair of parentheses contained only another pair of parentheses or a single operand (e.g., `((a))`, `(a)`). This indicates a redundant pair. We return `1` in this case.
    - If the `hasOperator` flag is `true`, it means the sub-expression contained at least one operator, and the parentheses are necessary. We then pop the matching opening parenthesis `'('` from the stack and continue our iteration.

Step-by-Step Algorithm

1. Initialize an empty **stack** to store characters.
2. Iterate through the input string `s` from left to right.
3. For each character `ch`:
    - If `ch` is `'('` or an **operator** (`+`, `-`, `*`, `/`), push it onto the stack.
    - If `ch` is a closing parenthesis `')'`:
      a. Initialize a boolean flag `isRedundant` to `true`.
      b. While the top of the stack is **not** an opening parenthesis `'('`:
      - Pop the element from the stack.
      - If the popped element is an operator (`+`, `-`, `*`, `/`), set `isRedundant` to `false`. This means the parentheses are **not** redundant.
      c. If `isRedundant` is still `true`, it means we never encountered an operator. This pair of parentheses is redundant, so return `1`.
      d. Finally, pop the opening parenthesis `'('` from the stack to match the current closing one.
4. If the loop completes without finding any redundant parentheses, return `0`.

Example Dry Run: `exp = ((a+b))`

1. Initialize an empty stack: `stack = []`
2. Iterate through `exp`:
    - `ch = '('`: Push `'('`. `stack = ['(']`
    - `ch = '('`: Push `'('`. `stack = ['(', '(']`
    - `ch = 'a'`: Skip (it's an operand). `stack = ['(', '(']`

- `ch = '+'`: Push `'+'`. `stack = ['(', '(', '+']`
- `ch = 'b'`: Skip. `stack = ['(', '(', '+']`
- `ch = ')'`:
  - `isRedundant = true`
  - Pop `'+'`. It's an operator, so `isRedundant = false`. `stack = ['(', '(']`
  - Pop `'('`. The while loop stops.
  - `isRedundant` is `false`, so we don't return.
  - Pop the matching `'('`. `stack = ['(']`
- `ch = ')'`:
  - `isRedundant = true`
  - The top of the stack is `'('`. The while loop condition (`stack.peek() != '('`) is immediately `false`.
  - `isRedundant` is still `true`. This means the parentheses enclose nothing but another sub-expression (which, in this case, is `(a+b)`). The outer parentheses are redundant.
  - Return `1`.

Code Implementation

Java

```java
import java.util.Stack;

class Solution {
    public static int checkRedundancy(String s) {
        Stack<Character> stack = new Stack<>();

        for (char ch : s.toCharArray()) {
            if (ch == '(' || ch == '+' || ch == '-' || ch == '*' || ch == '/') {
                // Push opening parentheses and operators
                stack.push(ch);
            } else if (ch == ')') {
                // When a closing parenthesis is found
                boolean hasOperator = false;

                // Pop elements until a matching opening parenthesis is found
                while (!stack.isEmpty() && stack.peek() != '(') {
                    char top = stack.pop();
                    // Check if there was an operator inside this set of parenthese
                    if (top == '+' || top == '-' || top == '*' || top == '/') {
                        hasOperator = true;
                    }
                }

                // If the stack becomes empty or the top is '(', we check if an ope
                if (!hasOperator) {
                    // This means the parentheses enclosed a sub-expression withou
                    // e.g., ((a)), (a), ((a+b))
                    return 1; // Redundant parentheses found
                }

                // Pop the matching opening parenthesis
                if (!stack.isEmpty() && stack.peek() == '(') {
                    stack.pop();
                }
            }
        }
        return 0; // No redundant parentheses found
    }
}
```

The approach can be simplified slightly. We only need to push operators and opening parentheses. When we encounter a closing parenthesis, we check what's on top of the stack. If it's an opening parenthesis, it's a redundant pair (like `()`). If not, we pop operators until we find the opening one. The core logic remains the same. The code above is a good representation of the full logic.