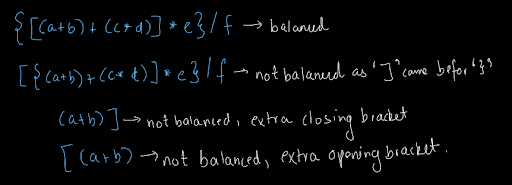
**1. Problem Discussion :-**

The previous problem was where we had to find whether the expression had repeated brackets or not, but the brackets were balanced. Here, we have to find whether the brackets are balanced or not. For that, have a look at the figure given below,

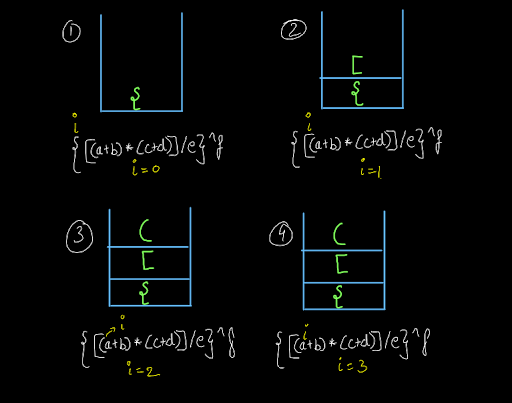
****

An expression will have unbalanced brackets if the order of the brackets does not match, or if the opening brackets are more as compared to the closing brackets or the closing brackets are more as compared to the opening brackets. So, if an expression has balanced brackets we have to print true else we have to print false. Dear reader, we recommend you try this question yourself first and then move to the solution. If you have any doubts regarding the question, you may refer to the question video to clear all your doubts.

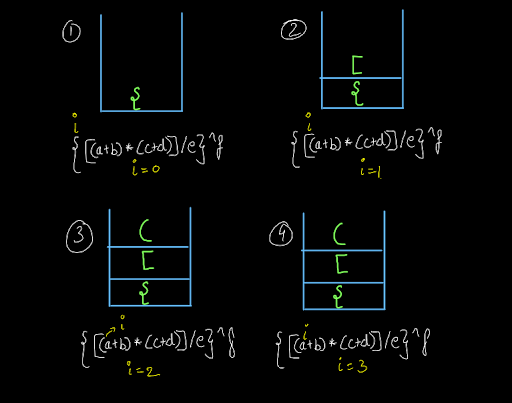
**2. Approach :-**

We will divide this problem into what, and why and try to solve it simply. WHAT: So, here we will see what the problem is and what we can do about it. So, you have already understood the problem that we need to print true if the brackets are balanced and we need to print false if they are not., So, the next question arises, what can we do? We are using the stack data structure right now. It has two primary methods i.e. push and pop. So, we are going to use these two standard operations only Now, the next question is what are we going to push and pop? We are going to push all the opening brackets and pop them whenever we encounter any closing brackets. How and Why? Now, let us discuss the how of the question. Let us take one example of each category and try to find out how our above-mentioned push and pop operations, discussed in what section will take place and why we will get our answer using them.

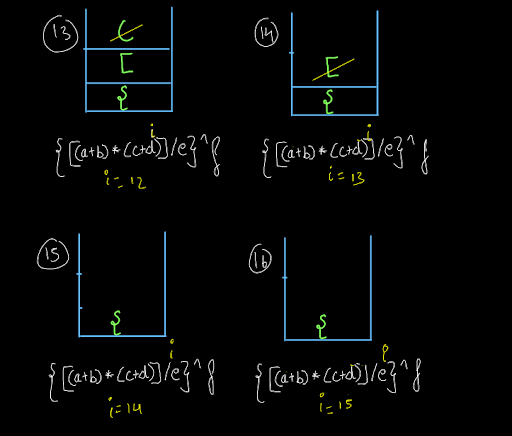
Balanced Parenthesis: Look at the diagram given below and try to analyze each stage of the stack when a character is pushed or popped.

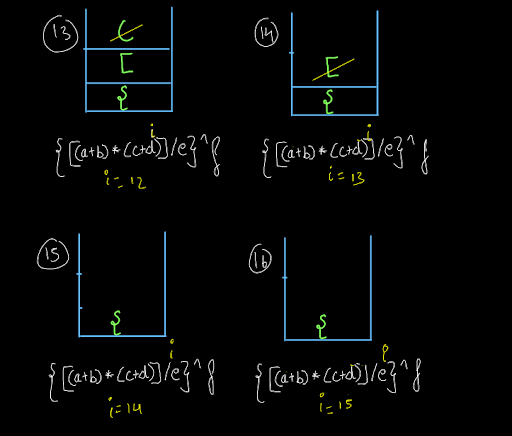
****

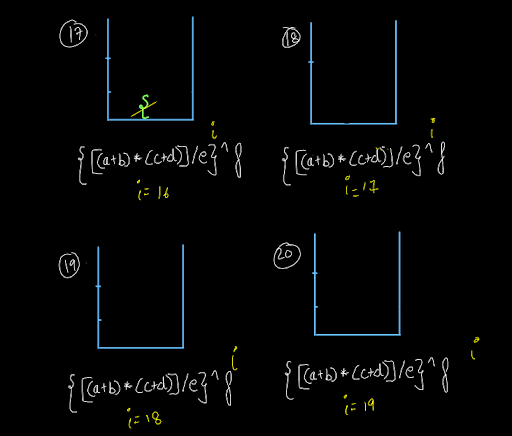
The input string was : { [ ( a + b ) \* ( c + d ) ] / e } ^ f Initially, the variable 'i' was at index 0. It encountered the first opening bracket '{' at i=0. So, it pushed it into the stack. Then in the next two steps also opening brackets were encountered and they were pushed into the stack. Then we came across alphabets and operator symbols which we kept ignoring and moved forward till 'i' became equal to 6.

****

When i=6, we encountered a closing bracket. Here, we checked whether we got the corresponding opening bracket from the stack or not. The brackets matched and we popped out the opening bracket '(' from the stack. Similarly, we kept on moving and scanned the entire string as shown below:

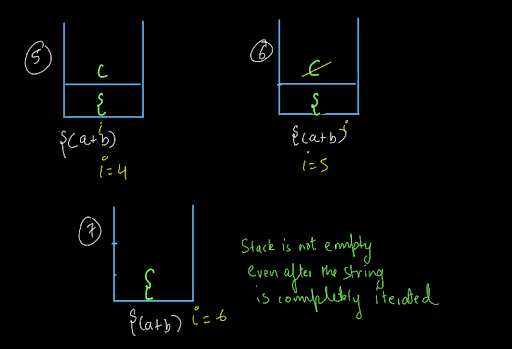
****

****

****

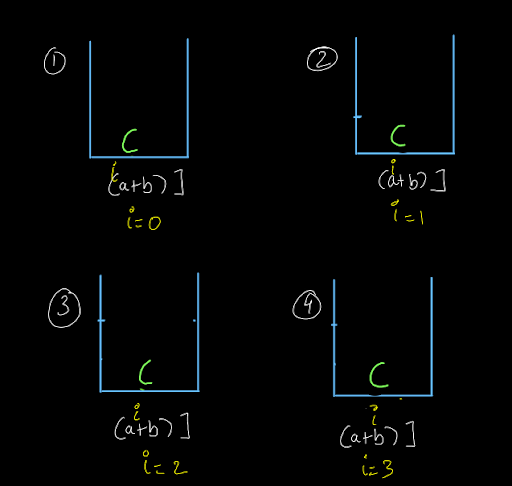
Whenever we encountered a closing bracket, we checked the stack and we got the corresponding opening bracket. So, we popped and we iterated through the entire string and when the string was iterated completely, we got the stack empty meaning there were no extra opening brackets. There were no extra closing brackets also as we never got stack empty whenever we wanted to pop from it. So, the brackets were balanced. Now, let us see how this process will work when the brackets are not balanced.

Extra Opening Brackets: Consider the following expression: { ( a + b ) There is an extra opening bracket in the expression. Now, let us try to find out how we will come to know of this situation using our method.

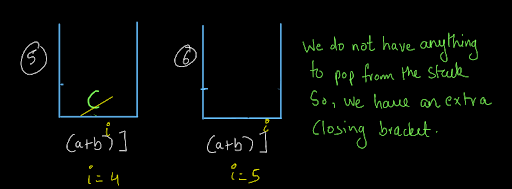
****

When i=5, we encounter a closing bracket. We check whether we have the corresponding opening bracket into the stack or not. Since we had the matching opening bracket, we popped out an element from the stack. Now, we have iterated through the entire string but, our stack is not empty. This means that there was an extra opening bracket for which we could not find a closing bracket and hence it was not popped. So, the brackets are not balanced.

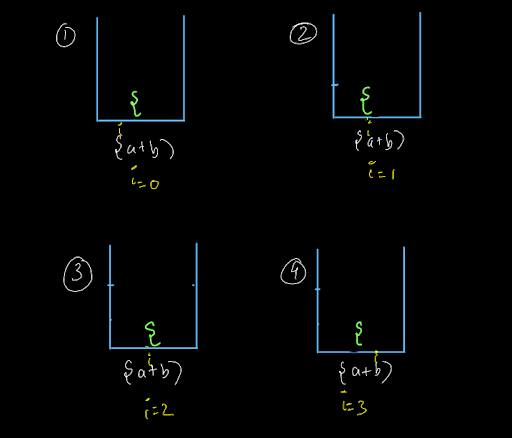
Extra Closing Bracket: Consider the following expression: ( a + b ) ] There is an extra opening bracket in the expression. Now, let us try to find out how we will come to know of this situation using our method.

****

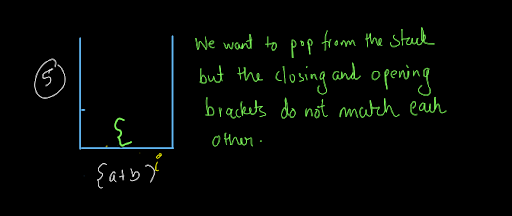
We started with i=0 and since we got an opening bracket we pushed it into the stack. Then, we kept on moving as we got alphabets and operator characters.

****

At i=4, we got a closing bracket and we checked whether it was the corresponding opening bracket in the stack or not. Since it was the matching opening bracket, we popped it out of the stack. Then, we move to i=5 and we encounter a closing bracket. We have to now check for the opening bracket in the stack but the stack is empty. This means there is an extra closing bracket for which no opening bracket is present. So, we will print false as the brackets are not balanced. Brackets Mismatch: Well, till now, you would have already understood how this is going to work. Still, let's have a look at it. Consider the following input string: { a + b ). Here the opening and closing brackets are equal in number, but they are not of the same type.

****

We started with i=0 and since it was an opening bracket we pushed it into the stack. Now, we kept on moving further and ignoring the alphabet and operator characters.

****

Now, when i=5, we get a closing bracket. When we try to pop from the stack, we find that the opening and closing brackets are of different types. So, the brackets are not balanced here and we will print false.

**3. CODE -:**

ConsoleCpp

#include <bits/stdc++.h>

using namespace std;

int main(int argc, char\*\* argv)

{

string s;

getline(cin, s);

stack<char>st;

for (int i = 0; i < s.length(); i++)

{

char ch = s[i];

if (ch == '(' || ch == '{' || ch == '[')

{

st.push(ch);

} else if (ch == ')')

{

if (st.size() == 0 || st.top() != '(')

{

cout << "false";

return 0;

} else

{

st.pop();

}

} else if (ch == '}')

{

if (st.size() == 0 || st.top() != '{')

{

cout << "false";

return 0;

} else

{

st.pop();

}

} else if (ch == ']')

{

if (st.size() == 0 || st.top() != '[')

{

cout << "false";

return 0;

} else

{

st.pop();

}

}

}

if (st.size() == 0)

{

cout << "true";

} else

{

cout << "false";

}

return 0;

}

**4. Code Discussion -:**

Iterate through the string and : Push any opening bracket whether '(' or '[' or '{' whenever you encounter it. Whenever you encounter any closing bracket whether ')' or ']' or '}', see if the bracket at the top of the stack matches with the corresponding closing bracket. If it matches, pop the opening bracket from the stack, else print false and break.

Ignore all other symbols in the string and keep moving forward If at some point, you encounter a closing bracket and there is no element to pop i.e. the stack is empty, print false, and break as this means that the number of closing brackets is more as compared to the number of opening brackets.

If you have scanned through the entire string and there was no mismatch found, but the stack is not empty, it means the number of opening brackets is more than the number of closing brackets. Print false and break.

If you have iterated the entire string and there was no mismatch and the stack is also empty, this means that the brackets were matching and the number of opening and closing brackets was equal. So, print true.

**5. Analysis -:**

Time Complexity : O(n)

The time complexity of the above algorithm is O(n) as we are traversing a string of length n once. What about the push and pop operations then? Come on, you already know they take O(1) time, right? So, the only thing affecting the time complexity is the traversing of the string.

Space Complexity : O(1)

The space complexity is O(n). Why? We are continuously popping and pushing the elements from the stack. So why O(n)? Well, let me ask you a question. What can be the maximum size of the stack? Yes, it can be equal to the length of the string if we can input the string with all opening brackets. It is after the string gets completely scanned that we will realize that the brackets are not balanced. Otherwise, we keep on pushing all the opening brackets into the stack.

That was easy. Wasn't it? Our desire to make you learn will remain unsatisfactory if you still have doubts. We strongly recommend you to watch our video lecture on Balanced Brackets for clearing any type of doubts.

Suggestions and feedback are always welcomed. You can contact us via our website. All the best for a bright future! Happy Coding!