Multiple Parenthesis Matching Using Stack with C Code

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-34

In the last tutorial, we saw the implementation of parentheses matching using stacks in C. One thing you must have observed is that we used only one type of parenthesis throughout the tutorial. But in mathematics, we have expressions consisting of all three types of parenthesis. Today we will be interested in matching parentheses when all three types of parentheses are used in any expression. This is what we called multi-parenthesis matching.

If you remember, parenthesis matching has nothing to do with the validity of the expression. It just tells whether an expression has all the parentheses balanced or not. A balanced parentheses expression has a corresponding closing parenthesis to all of its opening parentheses. When we talk about matching multi parenthesis, our focus is mainly on the three types of an opening parenthesis, [{ (and their corresponding closing parentheses,) }]. So, basically, this tutorial is just an extension of what we learned in the previous two.

Modifying what we did earlier to make it work for multi-matching needs very little attention. Just follow these steps:

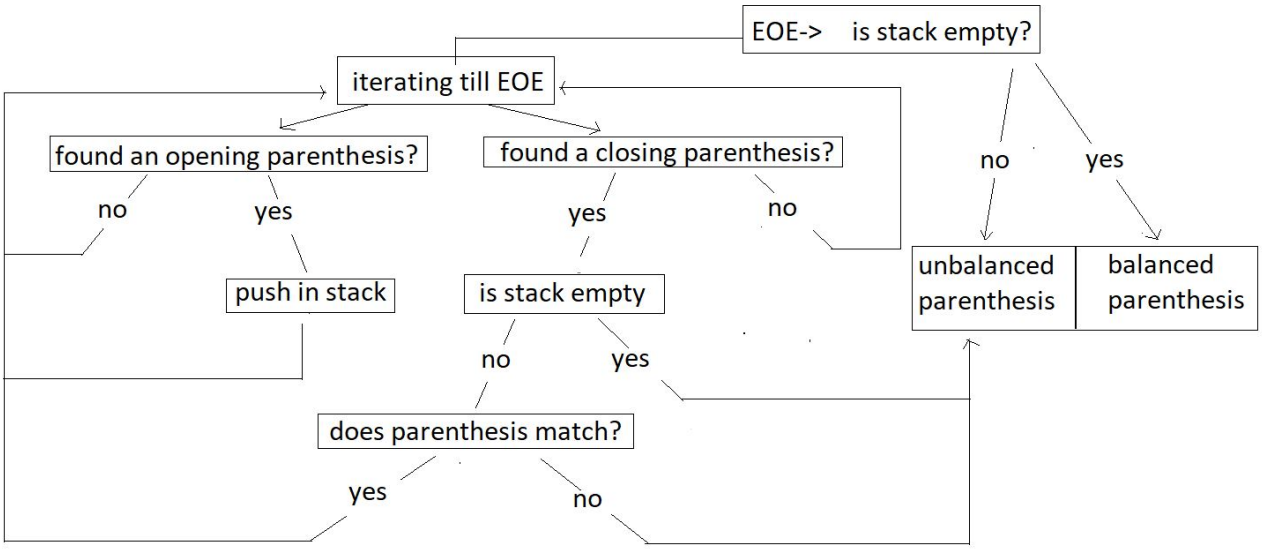
1. Whenever we encounter an opening parenthesis, we simply push it in the stack, similar to what we did earlier.

2. And when we encounter a closing parenthesis, the following conditions should be met to declare its balance:

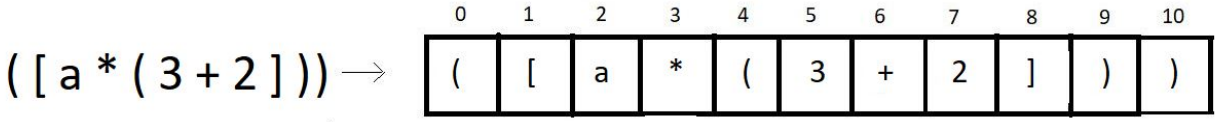
- Before we pop, this size of the stack must not be zero.
- The topmost parenthesis of the stack must match the type of closing parenthesis we encountered.

3. If we find a corresponding opening parenthesis with conditions in point 2 met for every closing parenthesis, and the stack size reduces to zero when we reach EOE, we declare these parentheses, matching or balanced. Otherwise not matching or unbalanced.

So, basically, we modified the pop operation. And that's all. Let's see what additions to the code we would like to make. But before that follow the illustration below to get a better understanding of the algorithm.

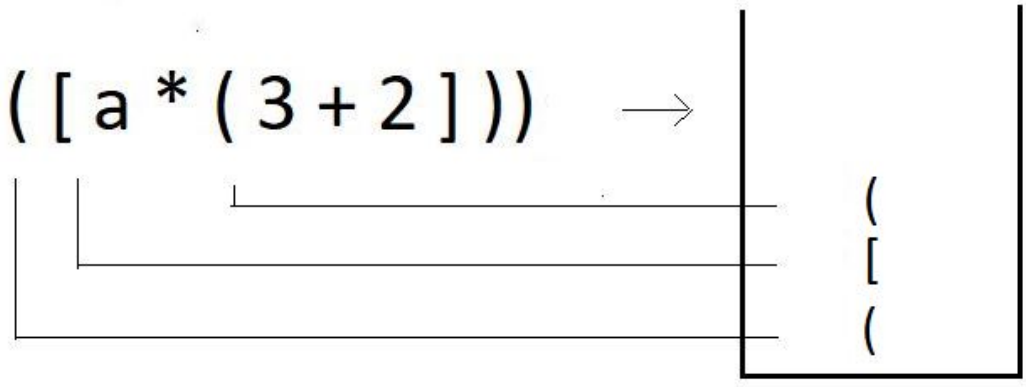


Example:

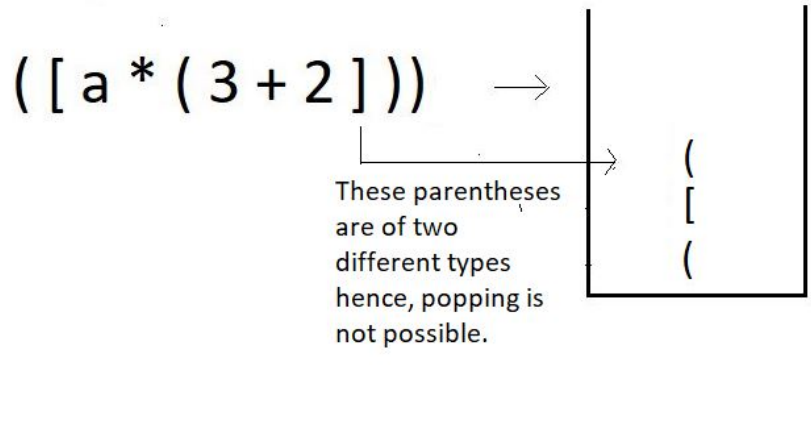


We'll try checking if the above expression has balanced multi-parentheses or not.

Step 1: Iterate through the char array, and push the opening brackets of all types at positions 0, 1, 4 inside the stack.



Step 2: When you encounter a closing bracket of any type in the expression, try checking if the kind of closing bracket you have got matches with the topmost bracket in the stack.



Step 3: Since we couldn't pop an opening bracket corresponding to a closed bracket, we would just end the program here, declaring the parentheses **unbalanced**.

The modified function should follow this algorithm. Let's now move to our editors.

Understanding the code snippet below:

1. Since in this tutorial, our main focus is to modify the code for matching parenthesis of a single type to matching multi parentheses., we'll copy the whole thing from our last tutorial, from creating the function *parenthesisMatch* to the stack inside.

2. It is important to copy everything because a lot of things will remain the same. We make zero changes in the declaration of the stack and its members.

3. Run a loop starting from the beginning of the expression till it reaches EOE.

4. If the current character of the expression is an opening parenthesis, be it of any type,'(', '[', '{', push it into the stack using the push operation.

5. Else if the current character is a closing parenthesis of any type ')', ']', '}', see if the stack is not empty, using isEmpty, and if it is, return 0 there itself, else pop the topmost character using pop operation and store it in another character variable named *popped_ch* declared globally.

6. Create an integer function, *match* which will get the characters, *popped_ch,* and the current character of the expression as two parameters. Inside this function, check if these two characters are the same. If they are the same, return 1, else 0.

```
int match(char a, char b){
    if(a=='{' && b=='}'){
        return 1;
    }
    if(a=='(' && b==')'){
        return 1;
    }
    if(a=='[' && b==']'){
        return 1;
    }
  return 0;
}
```

Code Snippet 1: Creating the match function

6. If the *match* function returns 1, our pop operation is successful, and we can continue checking further characters; else, if it returns 0, end the program here itself and return 0 to the main.

7. And if things went well throughout, and in the end, if the stack becomes empty, return 1, else 0.

Code for multi parentheses matching:

```
int parenthesisMatch(char * exp){
    // Create and initialize the stack
    struct stack* sp;
    sp->size = 100;
    sp->top = -1;
    sp->arr = (char *)malloc(sp->size * sizeof(char));
    char popped_ch;

    for (int i = 0; exp[i]!='\0'; i++)
    {
        if(exp[i]=='(' || exp[i]=='{' || exp[i]=='['){
            push(sp, exp[i]);
        }
        else if(exp[i]==')'|| exp[i]=='}' || exp[i]==']'){
            if(isEmpty(sp)){
                return 0;
            }
            popped_ch = pop(sp);
            if(!match(popped_ch, exp[i])){
              return 0;
            }
        }
    }

    if(isEmpty(sp)){
        return 1;
    }
    else{
        return 0;
    }
}
```

Code Snippet 2: Creating the modified parenthesisMatch function

Here is the whole source code:

```
#include <stdio.h>
#include <stdlib.h>

struct stack
{
    int size;
    int top;
    char *arr;
};

int isEmpty(struct stack *ptr)
{
    if (ptr->top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int isFull(struct stack *ptr)
{
    if (ptr->top == ptr->size - 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void push(struct stack* ptr, char val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}

char pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}

char stackTop(struct stack* sp){
    return sp->arr[sp->top];
}
```

```
int match(char a, char b){
    if(a=='{' && b=='}'){
        return 1;
    }
    if(a=='(' && b==')'){
        return 1;
    }
    if(a=='[' && b==']'){
        return 1;
    }
  return 0;
}

int parenthesisMatch(char * exp){
    // Create and initialize the stack
    struct stack* sp;
    sp->size = 100;
    sp->top = -1;
    sp->arr = (char *)malloc(sp->size * sizeof(char));
    char popped_ch;

    for (int i = 0; exp[i]!='\0'; i++)
    {
        if(exp[i]=='(' || exp[i]=='{' || exp[i]=='['){
            push(sp, exp[i]);
        }
        else if(exp[i]==')'|| exp[i]=='}' || exp[i]==']'){
            if(isEmpty(sp)){
                return 0;
            }
            popped_ch = pop(sp);
            if(!match(popped_ch, exp[i])){
              return 0;
            }
        }
    }

    if(isEmpty(sp)){
        return 1;
    }
    else{
        return 0;
    }

}

int main()
{
    char * exp = "[4-6]((8){(9-8)})";

    if(parenthesisMatch(exp)){
        printf("The parenthesis is balanced");
    }
    else{
        printf("The parenthesis is not balanced");
    }
    return 0;
}
```

Code Snippet 3: A program to check for balanced multi-parentheses.

Let's try the functions now and see if they work. We will give it some random expressions of our choice.

```
    char * exp = "((8){(9-8)})";
    // Check if stack is empty
    if(parenthesisMatch(exp)){
        printf("The parenthesis is matching");
    }
    else{
        printf("The parenthesis is not matching");
```

Code Snippet 4: Calling the parenthesisMatch function

The output we received was:

```
The parenthesis is matching
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

Let’s see for some another expression:

```
    char * exp = "[[4-6]((8){(9-8])})";

    if(parenthesisMatch(exp)){
        printf("The parenthesis is balanced");
    }
    else{
        printf("The parenthesis is not balanced");
    }
```

Code Snippet 5: Calling the parenthesisMatch function for another expression

The output we received was:

```
The parenthesis is not matching
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output of the above program

So, now we are capable of handling the parenthesis of more than one type in an expression. This was matching multi-parentheses. In order to accommodate any no. of types of parentheses, these codes can be modified further. We should now move to our next topic. I am excited, and so should you be!

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven’t

checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll learn another great topic, infix, prefix, and postfix expressions. Till then, keep coding.