# Shunting yard Algorithm

## *Превод „Маневрена площадка“*

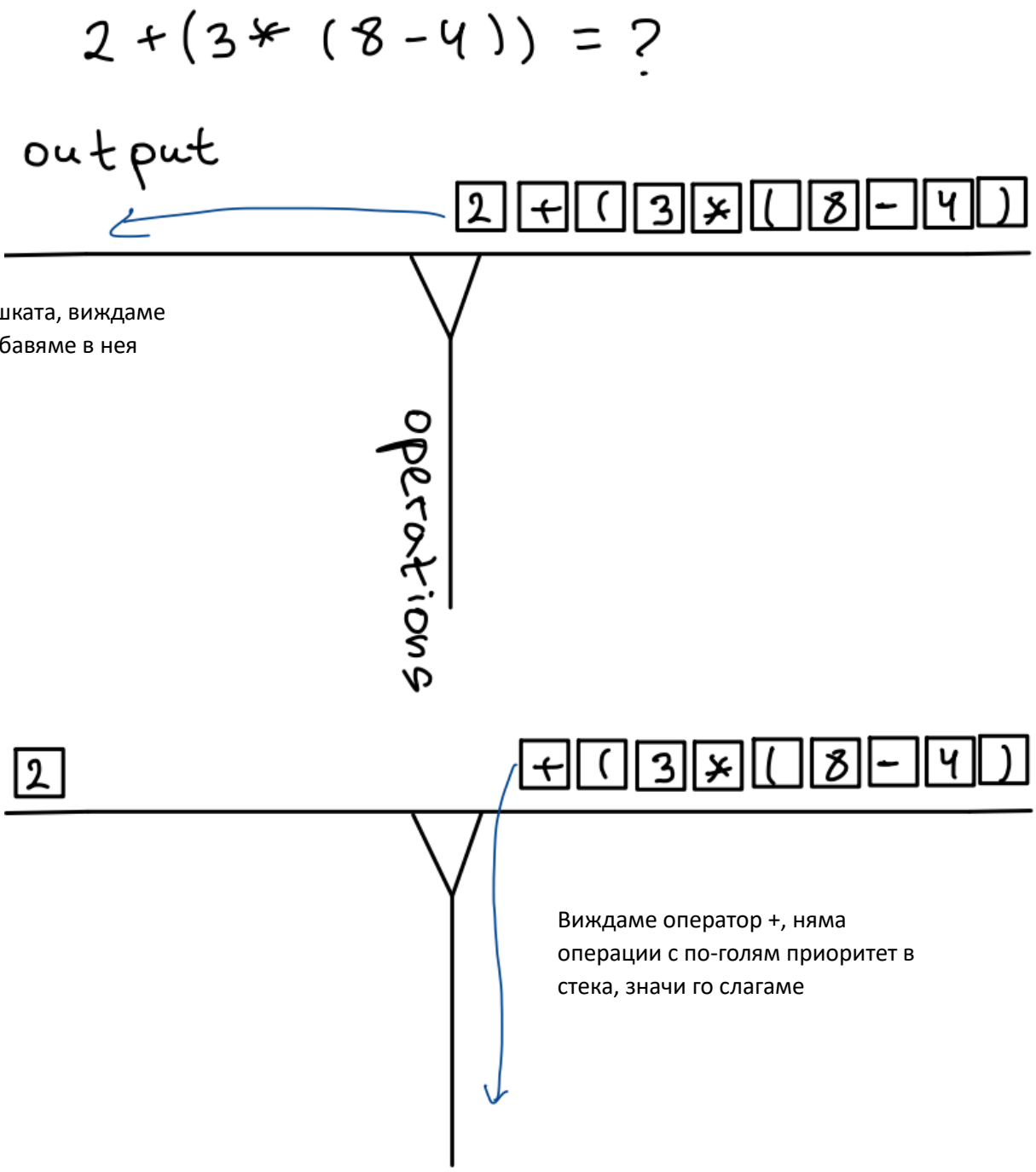
1. **Какво целим с него?**
   - Конвертиране на инфиксен запис към постфиксен (обратен полски) запис
   тоест *3 + 4 * ( 2 - 1 ) -> 3 4 2 1 - * +*
2. **Защо ни е?**
   - елиминира необходимостта от скоби за посочване на реда на операциите.
   - редът им се определя от позицията на оператора спрямо неговите операнди.
   - лесен за оценка с помощта на стек ♡
3. **Как работи?**
   Общи стъпки на алгоритъма:
   Вход: Израз в инфиксен запис (и евентуално списък, дефиниращ приоритет на операторите, но ние си знаем дефолтния)
   Изход: Израз в постфиксен запис.

   Пазим данните си в опашка *output*, стек *operations*.

   Стъпки:
   Докато има символи за четене:
   - Прочитаме символ.
   - Ако е число (операнд), го добавяме в *output*.
   - Ако е оператор, вадим оператори от *operations*, и ги добавяме в *output*, докато оператор с по-нисък приоритет е в горната част на стека, след което добавяме оператора в *operations*.
   - Ако това е лява скоба, добавяме в *operations*.
   - Ако това е дясна скоба, вадим оператори от *operations*, и ги добавяме в *output*, докато лява скоба е в горната част на стека, и я изваждаме.
   - Вадим всички останали оператори от *operations* към *output*.

4. **Хубава визуализация - https://jsfiddle.net/7jh9f/2/**

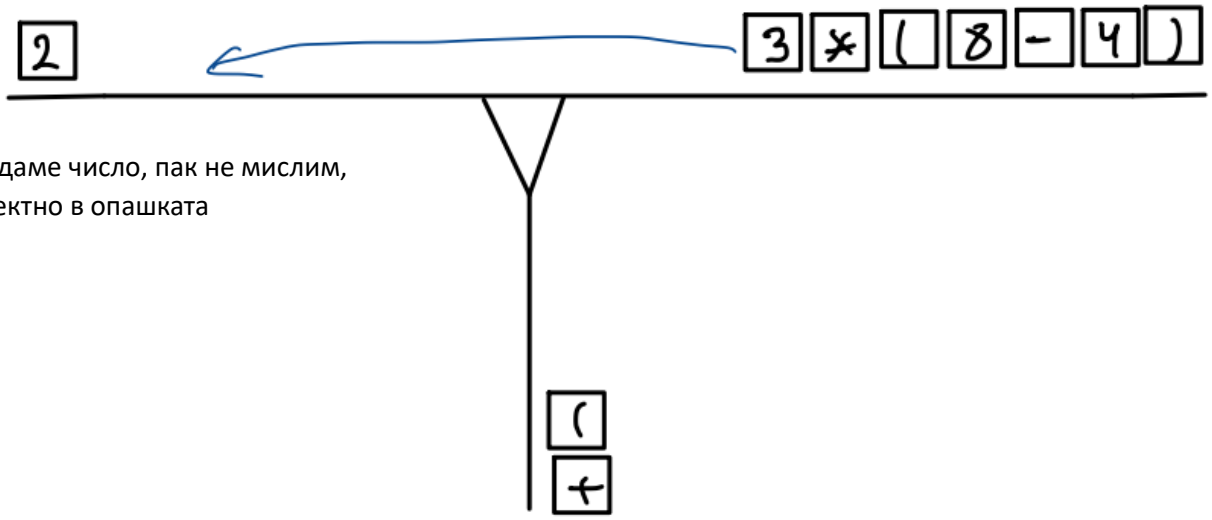
## 5. Нехубава визуализация (от мен)



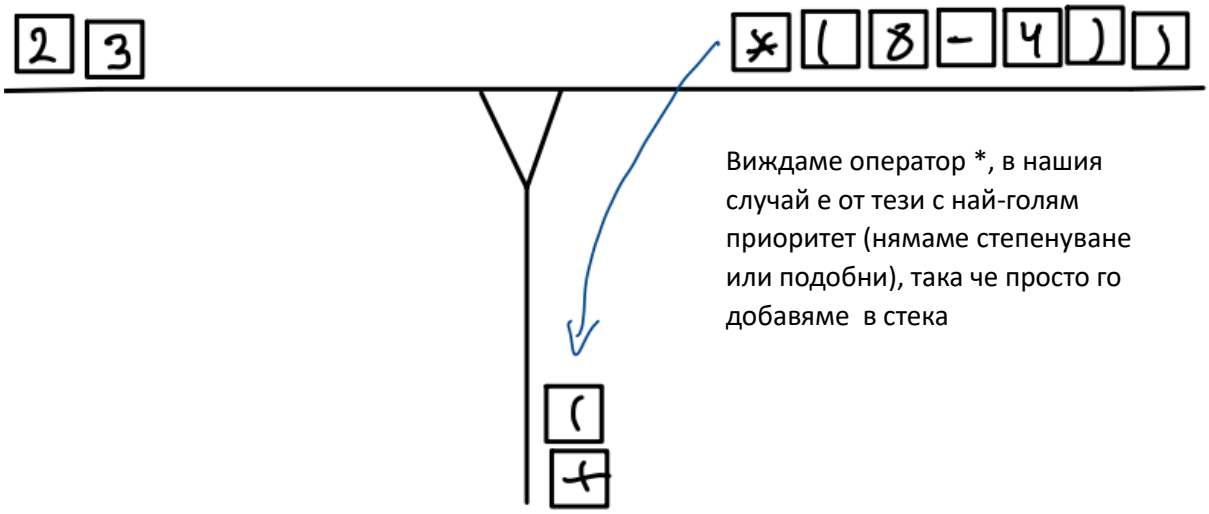

Това ни е опашката, виждаме число, и го добавяме в нея

Виждаме оператор +, няма операции с по-голям приоритет в стека, значи го слагаме

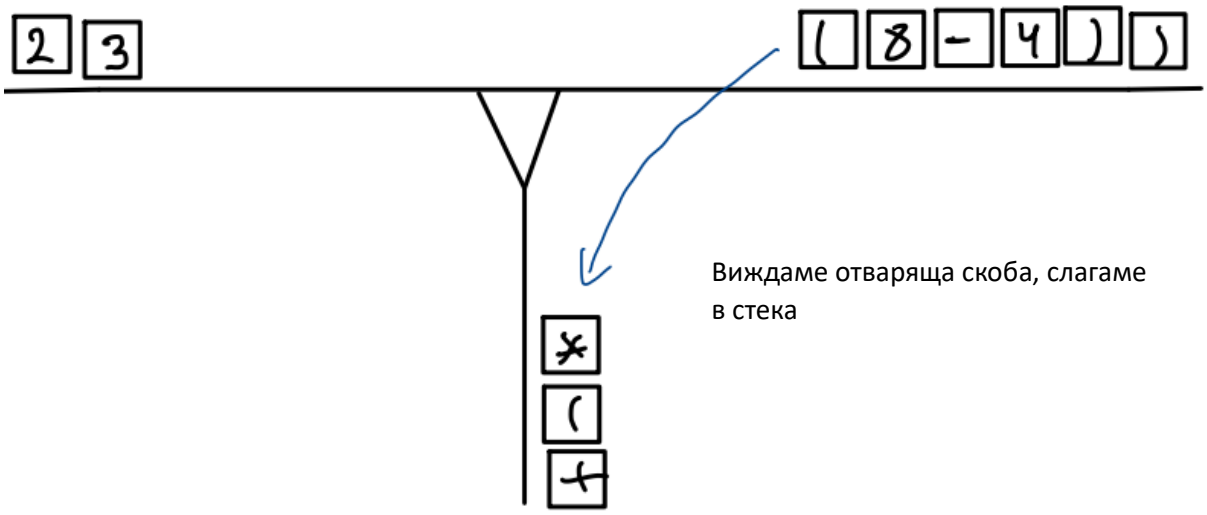

Виждаме отваряща скоба, няма какво да му мислим, слагаме в стека



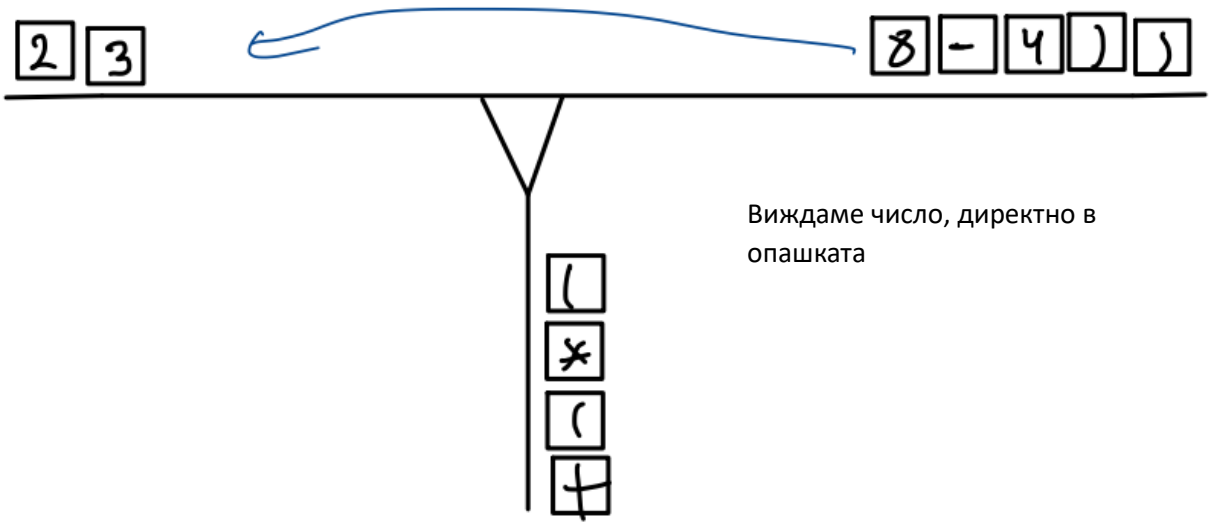

Виждаме число, пак не мислим, директно в опашката



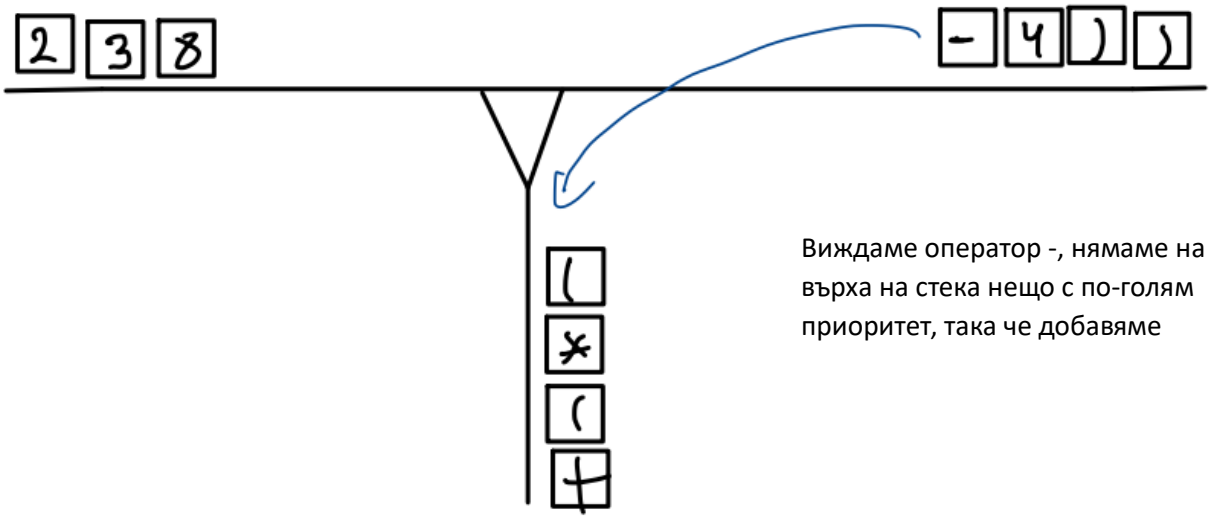

Виждаме оператор *, в нашия случай е от тези с най-голям приоритет (нямаме степенуване или подобни), така че просто го добавяме в стека




Виждаме отваряща скоба, слагаме в стека

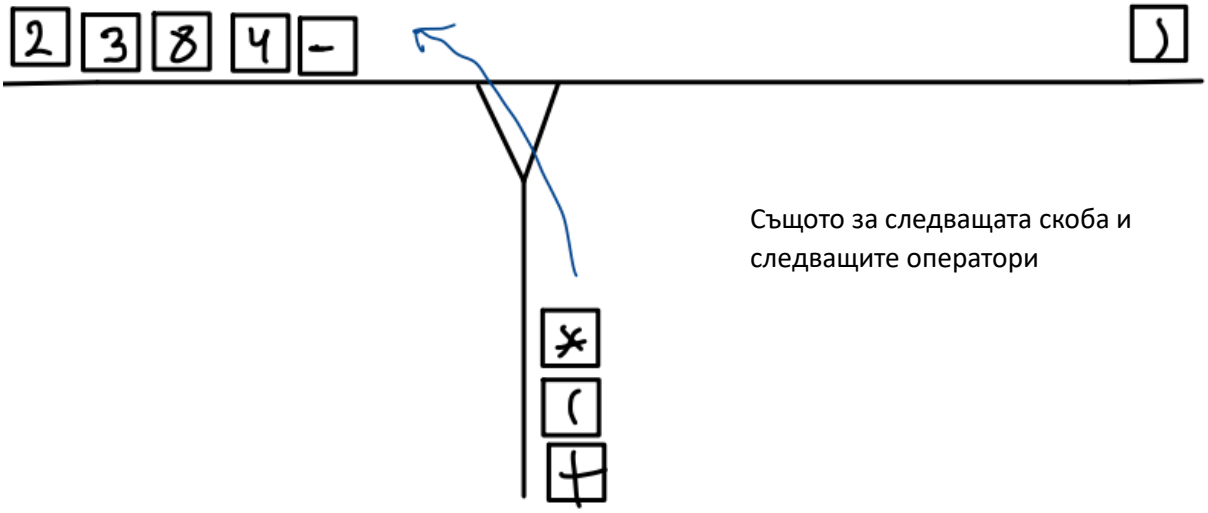



Виждаме число, директно в опашката

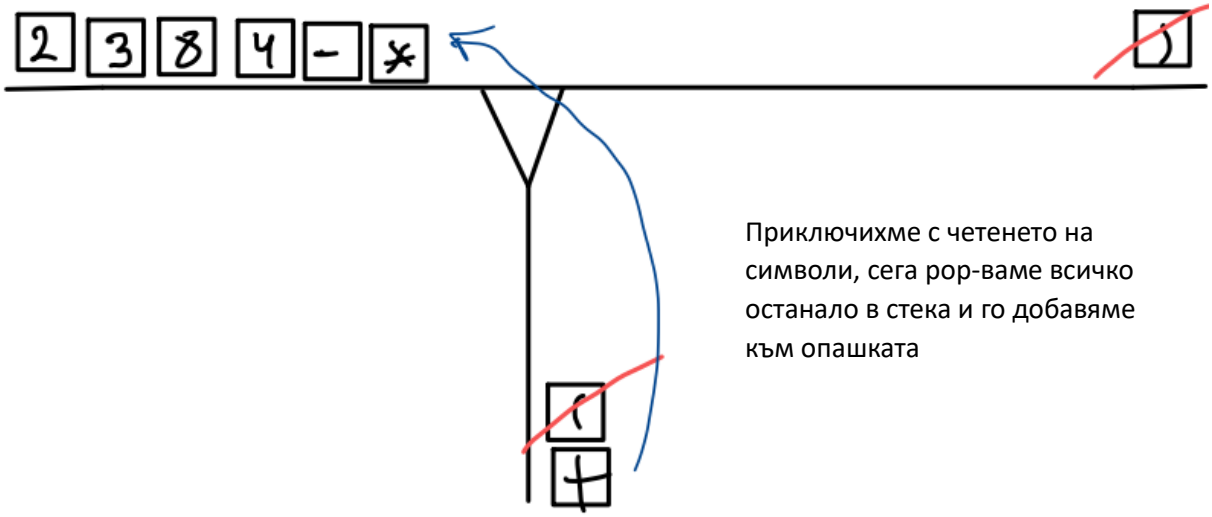



Виждаме оператор -, нямаме на върха на стека нещо с по-голям приоритет, така че добавяме

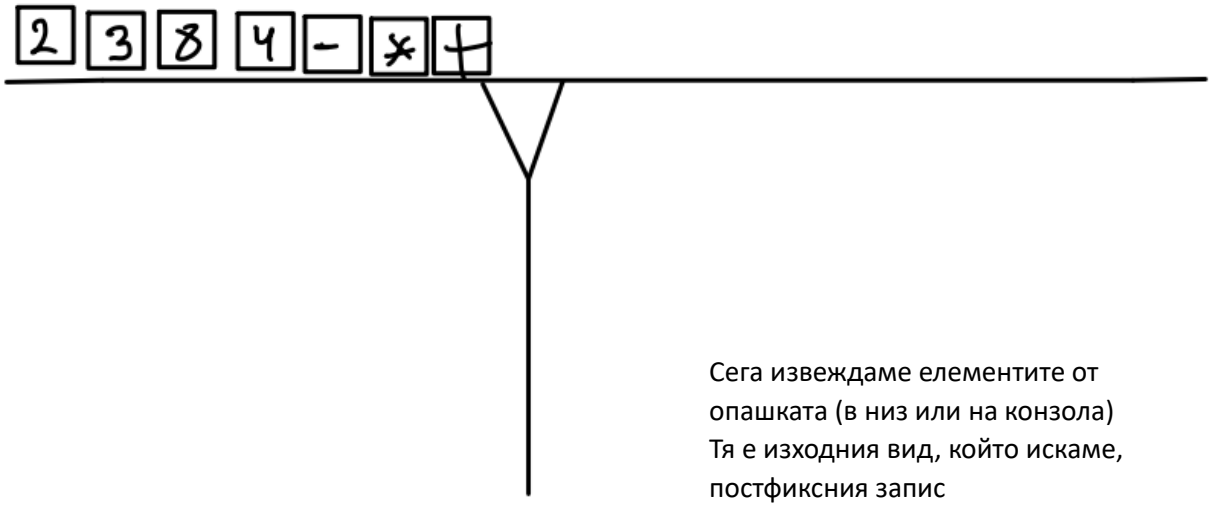

Виждаме затваряща скоба, започваме да местим всички елементи от стека в опашката, докато не срещнем затваряща скоба, махаме я и продължаваме




Същото за следващата скоба и следващите оператори



Приключихме с четенето на символи, сега pop-ваме всичко останало в стека и го добавяме към опашката



Сега извеждаме елементите от опашката (в низ или на конзола)
Тя е изходния вид, който искаме, постфиксния запис

2 3 8 4 - * +

*От тук идва и името на алгоритъма, представяме си, че имаме релси, и символите са вагони, ние контролираме маневрите по площадката, на база на това какъв тип е вагонът.*

### 6. Псевдокод

- 1 stack for operations
- 1 queue of the output
- 1 array (or other list) of token

```
1.  While there are tokens to be read:
2.        Read a token
3.        If it's a number add it to queue
4.        If it's an operator
5.               While there's an operator on the top of the stack with greater precedence:
6.                       Pop operators from the stack onto the output queue
7.               Push the current operator onto the stack
8.        If it's a left bracket push it onto the stack
9.        If it's a right bracket
10.            While there's not a left bracket at the top of the stack:
11.                     Pop operators from the stack onto the output queue.
12.             Pop the left bracket from the stack and discard it
13. While there are operators on the stack, pop them to the queue
```

## 7. Просто код (примерна имплементация на C++)

```
#include <iostream>
#include <stack>
#include <queue>
#include <string>

// Първо две помощни функции, за красота на кода
bool isOperator(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/' ;
}

int getPrecedence(char op) {
    switch (op) {
        case '+':
        case '-': return 1;
        case '*':
        case '/': return 2;
        default: return 0;
    }
}
```

```
std::string infixToPostfix(const std::string& infix) {
std::stack<char> s;
std::queue<char> q;

for (char token : infix) {
    // Число, в опашката
    if (isdigit(token)) {
        q.push(token);
    }
    // Лява скоба, в стека
    else if (token == '(') {
        s.push(token);
    }
    // Затваряща скоба, махаме оператори, докато видим отваряща
    else if (token == ')') {
        while (!s.empty() && s.top() != '(') {
            q.push(s.top());
            s.pop();
        }
        if (!s.empty()) {
            s.pop();  // Махаме отварящата скоба
        }
    }
    // Оператор, махаме оператори с по-висок приоритет (пращаме ги в опашката) и го добавяме
    else if (isOperator(token)) {
        while (!s.empty() && isOperator(s.top()) && getPrecedence(token) <= getPrecedence(s.top())) {
            q.push(s.top());
            s.pop();
        }
        s.push(token);
    }
}
//Премахваме всички оставащи оператори от стека
while (!s.empty()) {
    q.push(s.top());
    s.pop();
}

std::string postfix;
while (!q.empty()) {
    postfix += q.front();
    q.pop();
}

return postfix;
}
```

**8. Ами ако искаме и степенуване?**

- Просто го добавяме при видовете оператори, и слагаме приоритетът му да бъде най-висок.

**9. Ами ако искаме синус, косинус, логаритъм?**

Тук вече ще трябва да променим решението, тъй като дължината на имената на тези функции е повече от 1.

Ще ползваме така наречената токенизация. Разделяме израза на „токени“, като всеки от тях ни носи някакъв смисъл.

Например изразът 3 + sin(45) ще бъде токенизиран в следните токени: 3, +, sin, (, 45, ).

Нататък алгоритъмът ще бъде подобен на предходния, но ще трябва да работим със структури с низове в тях, понеже вече „токените“ не се побират само в един символ.

Разбира се, тъй като сме учили ООП, можем да направим и доста по-красиво решение, като си създадем обекти и функции, като всяка от тях отговаря за част от алгоритъма.