VIETNAM NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Reverse Polish Notation

Topic: Abstract data structures

Course: Data structures and Algorithms

Implemented by student:

Huỳnh Thái Hoàng (24127171)

Đặng Hoàng Nam (24127454)

Nguyễn Sơn Hải (24127163)

Lecturer:

B.Sc. Nguyễn Thanh Tình

Ph.D Nguyễn Ngọc Thảo

April 7, 2025

Contents

List of Tables

List of Figures

1 Abstract

Reverse Polish Notation (RPN) [5], also known as reverse Łukasiewicz [8] notation, Polish postfix notation or simply postfix notation, is a mathematical notation in which operators follow their operands, eliminating the need for parentheses to dictate operation precedence. This contrasts with the more common infix notation, where operators are placed between operands and often require parentheses to clarify evaluation order. The concept of RPN was developed to streamline the process of evaluating mathematical expressions, particularly in computing contexts. By placing operators after their respective operands, RPN allows for straightforward, unambiguous expression evaluation using stack-based data structures. This approach not only simplifies the parsing process but also enhances computational efficiency. The adoption of RPN offers several advantages in the realm of computer science and programming. By reducing the need for parentheses and adhering to a consistent evaluation order, RPN reduces the complexity of expression parsing and minimizes the potential for errors. Understanding RPN is integral to the study of *Data structures and Algorithms*, as it emphasizes the practical application of stacks in expression evaluation. Mastery of RPN not only enhances one's comprehension of computational processes but also provides insight into the design and implementation of efficient algorithms.

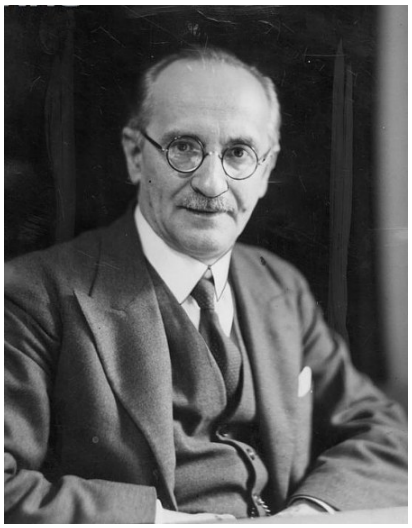


Figure 1: Jan Łukasiewicz

2 Infix, Prefix, Postfix

2.1 Infix expressions

When you write an expression such as:

B * C

, the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator * appears between them in the expression. The infix notation is easily readable by humans. **Infix notation** are mathematical expressions in which the **operator is placed between its operands**. This is the most common mathematical notation used by humans. For example,

1 + 2

is an infix expression, where the operator "+" is placed between the operands "1" and "2".
Infix notation is easy to read and understand for humans, but it can be difficult for computers to evaluate efficiently. This is because the order of operations must be taken into account and parentheses can be used to override the default order of operations.
Consider another infix example,

A + B * C

. The operators + and * still appear between the operands, but there is a problem. Which operands do they work on? Does the + work on A and B or does the * take B and C ? The expression seems ambiguous and difficult for the computer to understand.

2.2 Prefix Expressions (Polish Notation)

Consider the infix expression:

a + b

, what would happen if we moved the operator before the two operands? The resulting expression would be:

+ a b

which is the **prefix notation**.

Prefix notation are also known as **Polish notation**, are a mathematical notation where the operator precedes its operands. This differs from the more common **infix notation**, where the operator is placed between its operands.

In prefix notation, the **operator** is written **first**, followed by its **operands**.

2.3 Postfix Expressions (Reverse Polish Notation)

Postfix expressions are also known as **Reverse Polish Notation (RPN)**, are a mathematical notation where the **operator follows its operands**. This differs from the more common infix notation, where the operator is placed between its operands.

In postfix notation, **operators** are written **first**, followed by the **operator**. For example, the infix expression

10^2

would be written as

10 2 ^

in postfix notation.

Let's look at table 1 to see the comparison of infix, prefix and postfix notation.

Infix Notation	Prefix Notation	Postfix Notation
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Table 1: Examples of Infix, Prefix, and Postfix Notation

2.4 Operator precedence rules

Each operator has a precedence level. Operators of higher precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses.

The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Here is the table 2 summarizing the operator precedence rules for common mathematical operators:

Operators	**Precedence**
Parentheses `()`	Highest
Exponents `^`	High
Multiplication `*`	Medium
Division `/`	Medium
Addition `+`	Low
Subtraction `-`	Low

Table 2: Operator Precedence Table

Also, here is the table 3 showing the associativity for common mathematical operators:

Operators	**Associativity**
Exponents `^`	Right to Left
Multiplication `*`, division `/`	Left to Right
Addition `+`, subtraction `-`	Left to Right

Table 3: Associativity

3 Notation conversion & expression valuation

3.1 Why postfix representation of the expression?

3.1.1 The order in postfix and prefix notation

For postfix notation (Reverse Polish Notation), we scan the expression from **left to right** and for prefix notation (Polish Notation) from **right to left**.
For example,

X + Y * Z

in prefix notation would be written as

+ X * Y Z

. The **multiplication** operator comes immediately before the operands Y and Z, denoting that * has precedence over +. The **addition** operator then appears before the X and the result of the **multiplication**.
Similarly, in postfix, the expression would be

X Y Z * +

. Again, **the order of operations** is preserved since the * appears immediately after the Y and the Z, denoting that * has precedence, with + coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

3.1.2 Why infix notation is not as efficient as postfix and prefix notation

Consider the expression:

a + b * c + d

- The compiler first scans the expression to evaluate the expression $b * c$, then after that scans the expression to add a to it.

- Following that, the compiler scans for d and adds it to the previous result.

The repeated scanning makes it very inefficient. Although infix expressions are easily readable by humans, the computer cannot differentiate the operators and parenthesis fast.
Let's take an example relating to parentheses, considering the infix expression

$$(A + B) * C$$

. In this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when A + B was written in prefix, the addition operator was simply moved **before** the operands, + A B. The result of this operation becomes the first operand for the **multiplication**. The multiplication operator is moved in front of the entire expression, giving us

$$* + A B C$$

. Likewise, in postfix A B + forces the **addition** to happen **first**. The **multiplication** can be done to that result and the remaining operand C. The proper postfix expression is then

$$A B + C *$$

. Therefore, it is better to convert the **infix** expression to **postfix** (or **prefix**) form before evaluation. The postfix expressions can be evaluated easily using a *stack*. [1]

3.2 Converting an Infix notation to a Postfix notation

3.2.1 Shunting-yard algorithm

In the conversion process, we scan the infix expression from left to right so the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their **precedence**. Taking addition and multiplication as an example, since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear **after** the multiplication operator is used in **postfix notation**. Because of this reversal of order, it makes sense to consider using a *stack* to keep the operators until they are needed.

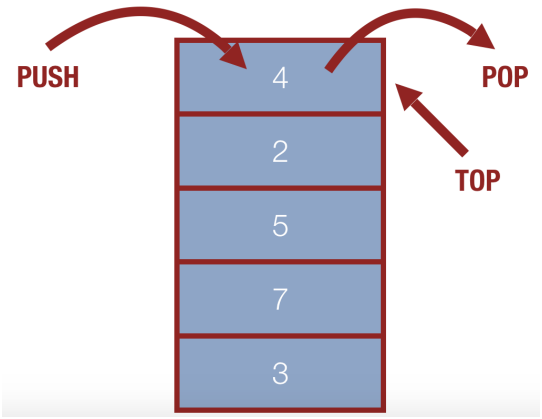


Figure 2: A stack

As we scan the infix expression from left to right, we will use a stack to keep the operators. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in **precedence** with the operators, if any, already on the stack.

The following steps will produce a string of tokens (an output list) in postfix order (so-called **Shunting-yard algorithm** [6]).

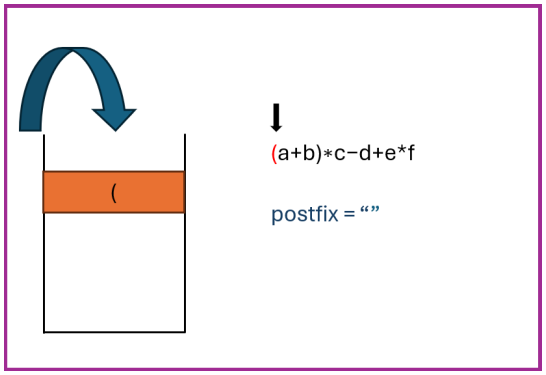
1. Initialize an empty operator stack and an empty output list.
2. Scan the infix expression from **left to right**.
   (a) If an operand is encountered, add it directly to the output list.
   (b) If a left parenthesis '(' is encountered, push it onto the stack.
   (c) If a right parenthesis ')' is encountered:
      i. Pop operators from the stack to the output until a left parenthesis is encountered.
      ii. Discard the pair of parentheses.
   (d) If an operator ('+,-,*,/,^') is encountered:
      i. While there is an operator at the top of the stack with **greater or equal precedence** (and for left-associative operators, equal precedence as well), pop operators from the stack to the output.
      ii. Then push the current operator into the stack.
3. End of Expression: Pop any remaining operators from the stack to the output.

3.2.2 Illustrative examples step by step

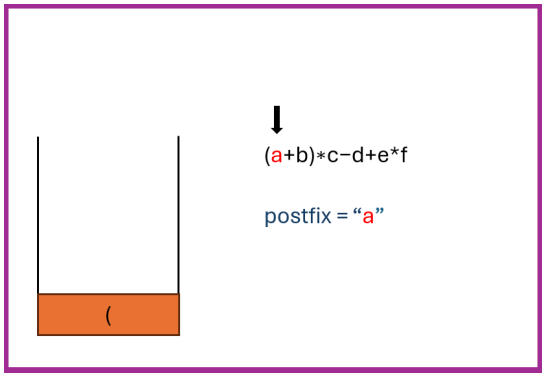
For example, let's consider an infix notation *infix* = (assuming this is a string)

$$(a + b) * c - d + e * f$$

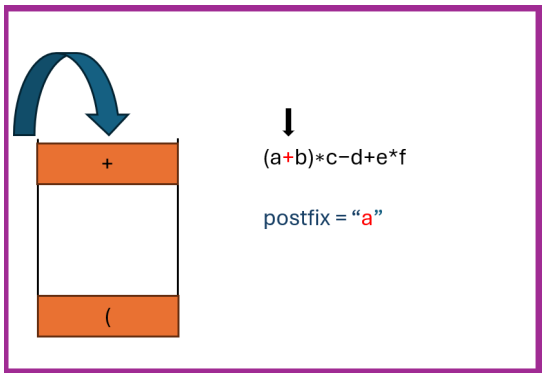
Step 1: i = 0, *infix[i]* = '(', it is a open parenthesis. Stack is empty so push it into the stack. So *postfix* = "a" and stack = { '('}.



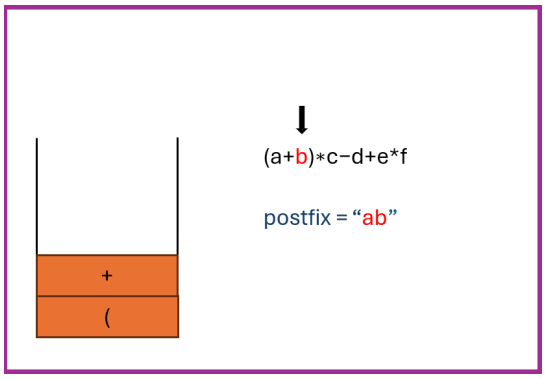
Step 2: i = 1, *infix[i]* = 'a', it is an operand. Therefore, we add this to *postfix* (our output list). So *postfix* = "a".



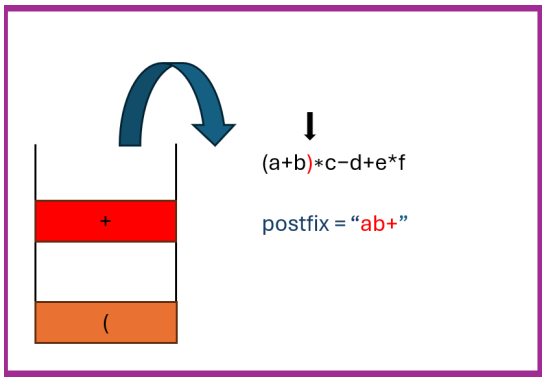
Step 3: i = 2, *infix[i]* = '+', it is an operator. Top stack is '(' so we push it into the stack normally. Therefore, *postfix* = "a" and stack = { '(', '+'}.

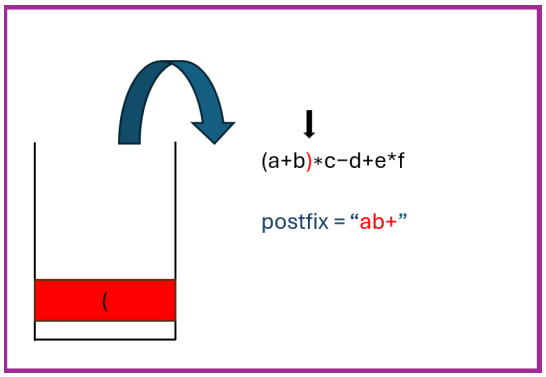


Step 4: i = 3, *infix[i]* = 'b', it is an operand. Therefore, we add this to *postfix*. So *postfix* = "ab".

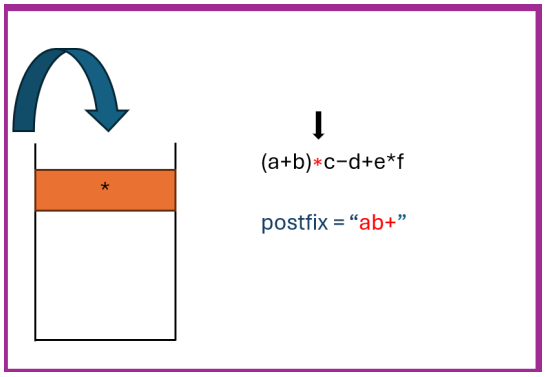


Step 5: i = 4, *infix*[i] = ')', it is a close parenthesis so we pop all the operators in the stack into the *postfix* until we successfully pop out a corresponding open parenthesis (we do not add any parenthesis into the postfix). So now stack is empty and *postfix* = "ab+".

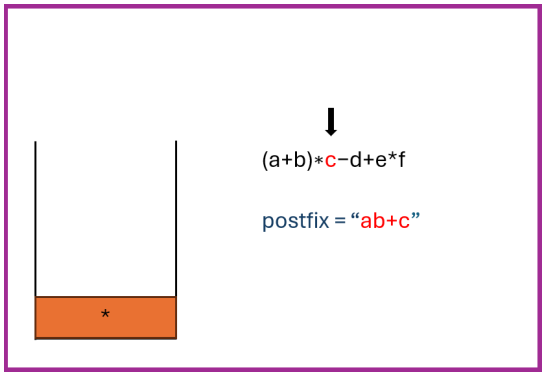




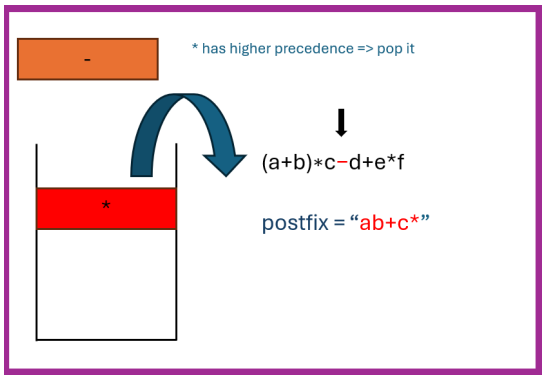
Step 6: i = 5, *infix[i]* = '*', it is an operator. Stack is empty so we push it into the stack. Therefore, *postfix* = "ab+" and stack = { '*'}.



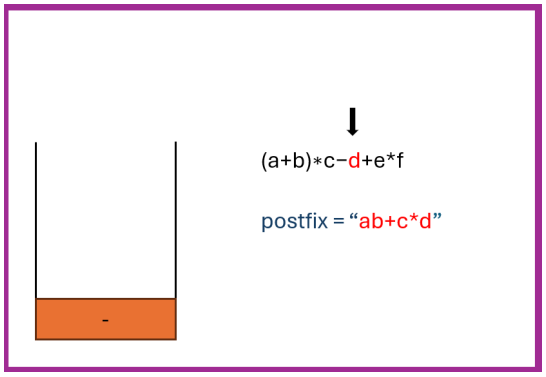
Step 7: i = 6, *infix[i]* = 'c', it is an operand. Therefore, we add this to *postfix*. So *postfix* = "ab+c".



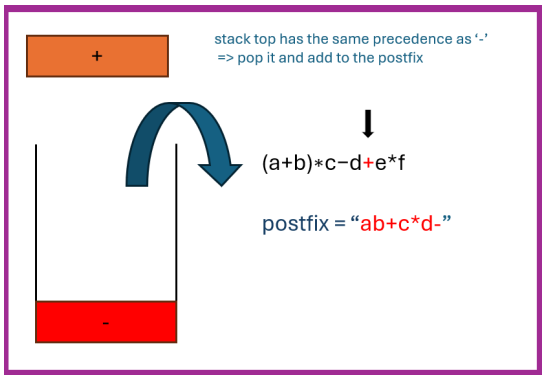
Step 8: i = 7, *infix[i]* = '-', it is an operator. The top stack is '*' which has higher precedence than '-' so we pop the '*' into *postfix*. Therefore, *postfix* = "ab+c*" and stack is empty. After that, '-' is pushed into the stack and now stack = { '-'}.



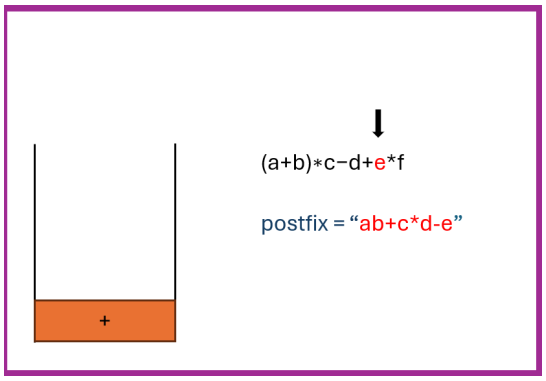
Step 9: i = 8, *infix[i]* = 'd', it is an operand. Therefore, we add this to *postfix*. So *postfix* = "ab+c*d".



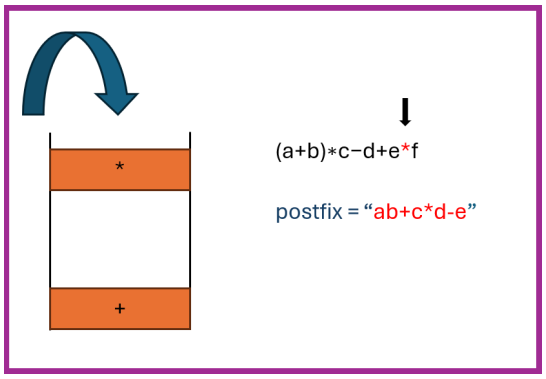
Step 10: i = 9, *infix[i]* = '+', it is an operator. The top stack is '-' which has the same precedence as '+' so we pop the top stack into the *postfix* and the stack now is empty. After that, *postfix* = "ab+c*d-" and the stack = { '+'}.



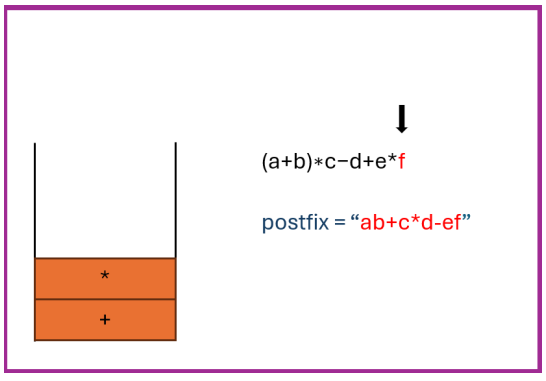
Step 11: i = 10, *infix[i]* = 'e', it is an operand. Therefore, we add this to *postfix*. So *postfix* = "ab+c*d-e".



Step 12: i = 11, *infix[i]* = '*', it is an operator. The top stack is '+' which has lower precedence than '*' so we push the '*' into the stack. After that, *postfix* = "ab+c*d-e" and the stack = { '+', '*' }.

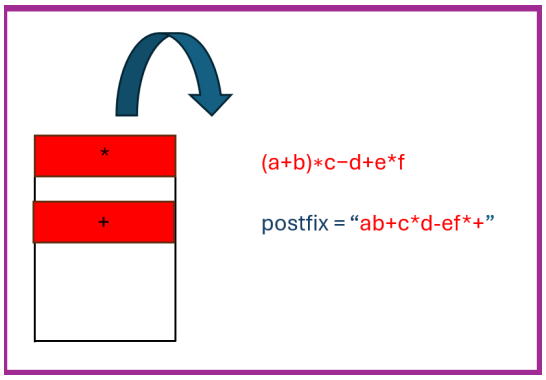


Step 13: i = 12, *infix[i]* = 'f', it is an operand. Therefore, we add this to *postfix*. So *postfix* = "ab+c*d-ef".



Step 14: i = 13 = length(infix) which means we have fully scanned the *infix*, now any remaining operators from the stack will be popped out to the *postfix*. As stack = { '+', '*' }, '+' and '*' are popped out and added to *postfix*, finally we have our result:

"a b + c * d - e f * +"



Here is the code for implementing the above algorithm:

- First, we need to have a function to find the precendence of two operators.

```
// function to find precedence of operators.
int precedence(char op)
{
    // if op is + or - (same precedence)
    if (op == '+' || op == '-') return 1;
    // if op is * or / (same precedence) higher than + and -
    if (op == '*' || op == '/') return 2;
    // remaining cases
    return 0;
}
```

- Here is the code for the *Infix to Postfix* conversion by using a *stack*.

```
// Function to convert infix expression to postfix.
string convertToPostfix(string infix) {
    // size of infix.
    int n = infix.length();
    string postfix = "";
    // stack to store operators.
    stack<char> ops;
```

```
8       // convert infix to postfix.
9       for (int i = 0; i < n; i++)
10      {
11          // if brace, push to 'ops'.
12          if (infix[i] == '(')
13          {
14              ops.push(infix[i]);
15          }
16          else if (isdigit(infix[i]))
17          {
18              string val = "";
19              // find the value.
20              while (i < n && isdigit(infix[i]))
21              {
22                  val += infix[i];
23                  i++;
24              }
25              postfix += val;
26              postfix += " ";
27              // go back one digit because now we are at the next digit after
    value.
28              i--;
29          }
30          // if closing brace.
31          else if (infix[i] == ')')
32          {
33              // handle operations in parentheses
34              while (!ops.empty() && ops.top() != '(')
35              {
36                  // take operator.
37                  char op = ops.top();
38                  ops.pop();
39                  postfix += op;
40                  postfix += " ";
41              }
42              ops.pop();
43          }
```

```
44          else
45          {
46              // while top of 'ops' has same or greater precedence, apply
    operator on top of 'ops' to top two elements in values stack.
47              while (!ops.empty() && precedence(ops.top()) >= precedence(
    infix[i]))
48              {
49                  char op = ops.top();
50                  ops.pop();
51                  postfix += op;
52                  postfix += " ";
53              }
54              // push new op to 'ops'.
55              ops.push(infix[i]);
56          }
57      }
58      // process remaining data in stack
59      while (!ops.empty())
60      {
61          // treat data as if it were in parentheses
62          char op = ops.top();
63          ops.pop();
64          postfix += op;
65          postfix += " ";
66      }
67      return postfix;
68  }
```

3.3 Converting an infix notation to a prefix notation

The idea for this algorithm is simple, it generally relies on the infix notation to posfix method, these are the following steps:

1. **Reverse** the infix notation including swapping '(' with ')' and vice versa
2. **Convert** this modified notation to *postfix notation* using **The Shunting-yard algorithm** in 3.2.1.

3. **Reverse** the obtained prefix notation to get the final prefix notation.

Here is an illustrative of the algorithm: let's consider an infix notation *infix* = (assuming this is a string)

$$(a + b) * c - d + e * f$$

First, we reverse the infix notaion:

$$f * e + d - c * (b + a)$$

Then we implement the **The Shunting-yard algorithm** in the table 3.3 below.

Step	**Infix expression**	**Stack**	**Temporary prefix expression**
1	f	[]	f
2	*	["*"]	f
3	e	["*"]	f e
4	+	["+"]	f e *
5	d	["+"]	f e * d
6	-	["-"]	f e * d +
7	c	["-"]	f e * d + c
8	*	["-", "*"]	f e * d + c
9	(	["-", "*", "("]	f e * d + c
10	b	["-", "*", "("]	f e * d + c b
11	+	["-", "*", "(", "+"]	f e * d + c b
12	a	["-", "*", "(", "+"]	f e * d + c b a
13	)	["-", "*"]	f e * d + c b a +
14	(pop the remaining elements)	[]	f e * d + c b a + * -

Table 4: Infix to prefix by using stack (table illustration)

Finally, we reverse the **temporary prefix notation** and then we have the actual prefix expression:

$$- * + a\ b\ c + d * e\ f$$

Here is the code for implementing the *infix to prefix* conversion by using a *stack*.

```
string convertToPrefix(string infix)
{
    reverse(infix.begin(), infix.end());
    for (int i = 0; i < infix.length(); i++)
    {
        if (infix[i] == '(')
            infix[i] = ')';
        else if (infix[i] == ')')
            infix[i] = '(';
    }
    int n = infix.length();
    string prefix = "";
    stack<char> ops;
    for (int i = 0; i < n; i++)
    {
        if (infix[i] == '(')
        {
            ops.push(infix[i]);
        }
        else if (isdigit(infix[i]) || isalpha(infix[i]))
        {
            string val = "";
            while (i < n &&
                (isdigit(infix[i]) || isalpha(infix[i])))
            {
                val += infix[i];
                i++;
            }
            reverse(val.begin(), val.end());
            prefix += val;
            prefix += " ";
            i--;
        }
```

```
34          else if (infix[i] == ')')
35          {
36              while (!ops.empty() && ops.top() != '(')
37              {
38                  char op = ops.top();
39                  ops.pop();
40                  prefix += op;
41                  prefix += " ";
42              }
43              ops.pop();
44          }
45          else
46          {
47              while (!ops.empty() &&
48               precedence(ops.top()) >= precedence(infix[i]))
49              {
50                  char op = ops.top();
51                  ops.pop();
52
53                  prefix += op;
54                  prefix += " ";
55              }
56              ops.push(infix[i]);
57          }
58      }
59      while (!ops.empty())
60      {
61          char op = ops.top();
62          ops.pop();
63          prefix += op;
64          prefix += " ";
65      }
66      reverse(prefix.begin(), prefix.end());
67      return prefix;
68  }
```

3.4 More ways to convert between notations

3.4.1 Prefix notation to infix notation

To convert the **prefix notation** to **infix one**, we do following steps:

1. Initialize an empty stack.
2. Scan the infix expression from **right to left**.
3. If an operand is encountered, push it onto stack
4. If an operator ("+,-,*,/,^") is encountered:
   (a) Pop the **top two values** of stack **(op1, op2)**.
   (b) Form an infix expression as "**(op1 operator op2)**".
   (c) Push infix expression onto stack.
5. After processing all tokens, the remaining value in stack is infix expression.

Here is an illustrative of the algorithm: Convert the prefix notation "/ * 2 + 5 3 4" to infix notation:

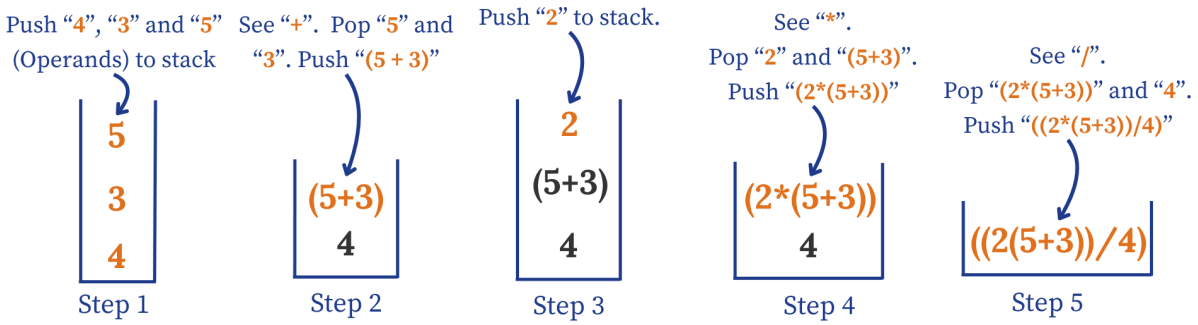


Figure 3: An example of prefix notation to infix notation

3.4.2 Postfix notation to infix notation

To convert the **postfix notation** to **infix** one, we also do following steps:

1. Initialize an empty stack.
2. Scan the infix expression from **left to right**.

3. If an operand is encountered, push it onto stack
4. If an operator ("+,-,*,/,^") is encountered:
   (a) Pop the **top two values** of stack **(op1, op2)**.
   (b) Form an infix expression as "**(op2 operator op1)**".
   (c) Push infix expression onto stack.
5. After processing all tokens, the remaining value in stack is infix expression.

Here is an illustrative of the algorithm: convert the postfix notation " 7 3 + 2 * " to infix notation:

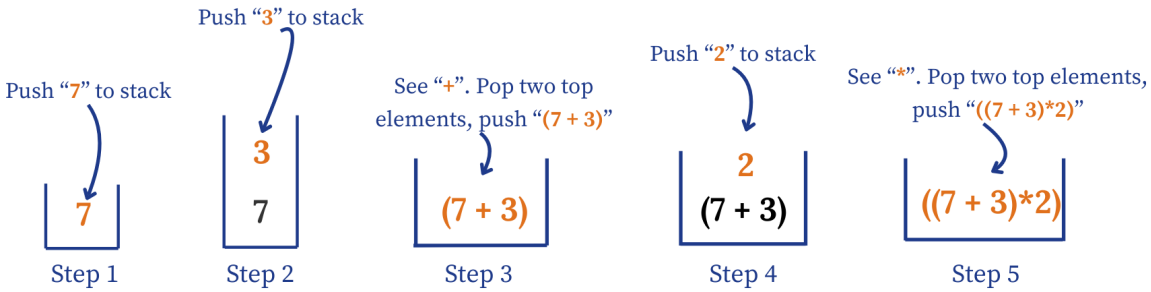


Figure 4: An example of postfix notation to infix notation

3.4.3 Prefix notation to postfix notation

To convert the **prefix notation** to **postfix** one, we also do following steps:

1. Initialize an empty stack.
2. Scan the prefix expression from **right to left**.
3. If an operand is encountered, push it onto stack.
4. If an operator ("+,-,*,/,^") is encountered:
   (a) Pop the **top two values** of stack **(op1, op2)**.
   (b) Form a postfix expression as "**(op1 op2 operator)**".
   (c) Push postfix expression onto stack.

5. After processing all tokens, the remaining value in stack is postfix expression.

Here is an illustrative of the algorithm: convert the prefix notation " - * + 5 6 7 8 " to postfix notation:

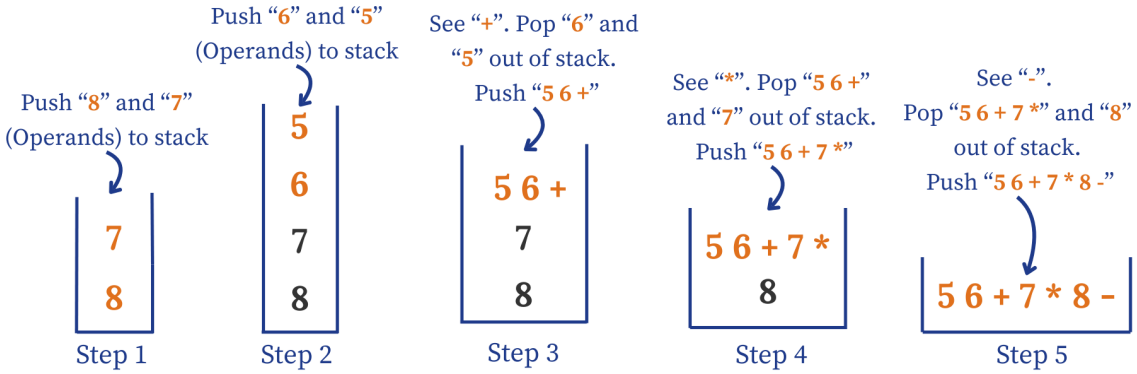


Figure 5: An example of prefix notation to postfix notation

3.4.4 Postfix notation to prefix notation

To convert the **postfix notation** to **prefix** one, we also do following steps:

1. Initialize an empty stack.
2. Scan the prefix expression from **left to right**.
3. If an operand is encountered, push it onto stack
4. If an operator ("+,-,*,/,^") is encountered:
   (a) Pop the **top two values** of stack **(op1, op2)**.
   (b) Form a prefix expression as "**(operator op2 op1)**".
   (c) Push prefix expression onto stack.
5. After processing all tokens, the remaining value in stack is prefix expression.

Here is an illustrative of the algorithm: convert the postfix notation " 6 4 - 4 7 + * " to prefix notation:

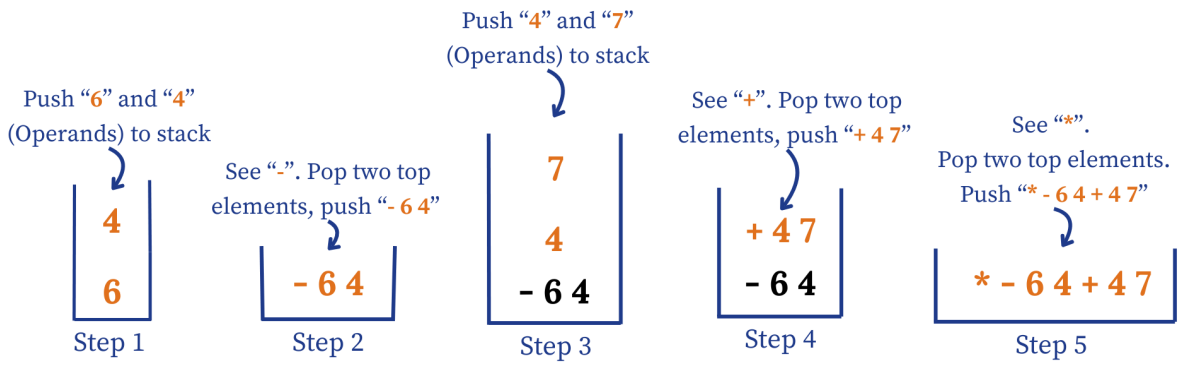


Figure 6: An example of postfix notation to prefix notation

4 Expression valuation

4.1 Postfix expression valuation

To evaluate a postfix expression we also use a stack [2]. The idea is to scan the from left to right and push the operands into the stack. Once an operator is encountered, pop the two topmost elements and evaluate them and push the result in the stack again. It is noted that in postfix notation, there is no parenthesis. Here is the algorithm:

1. Initialize an empty stack only to push operands.
2. Scan from **left to right** and do the following steps:
   (a) If an operands is encountered, push it into the stack.
   (b) If an operator is encountered, pop **two topmost operands** for the operator from the stack. Evaluate the operator and push the result back to the stack.
3. When the scanning is ended, the final element in the stack is the final result.

Here is an illustrative of the algorithm:

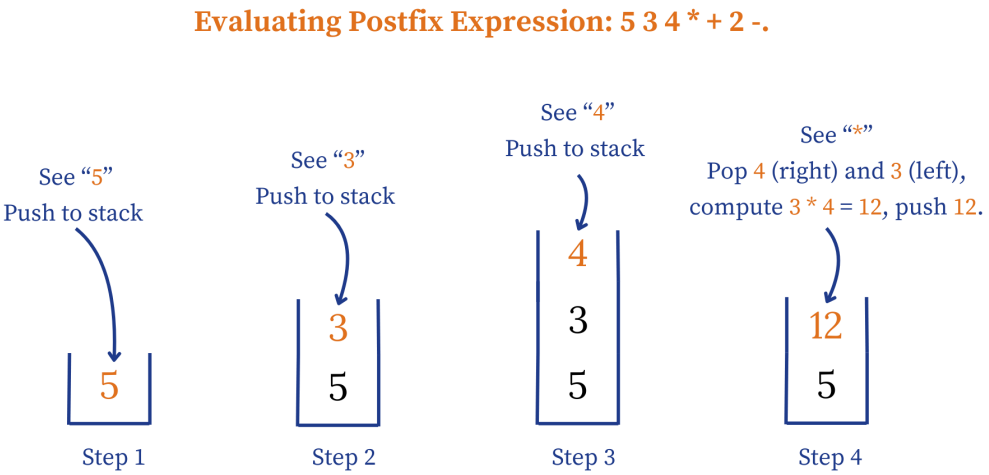


Figure 7: An example of prefix expression evaluation (step 1 - 4)

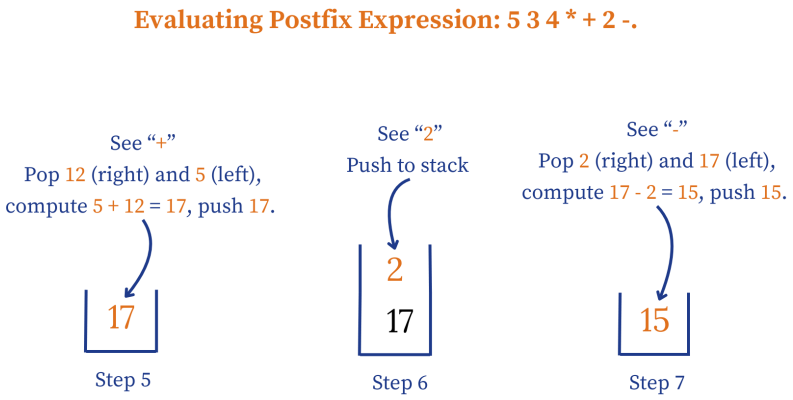


Figure 8: An example of prefix expression evaluation (step 5 - 7)

Here is the code for implementing the valuation:

- First, we need to have a function to apply two operations which returns to a double.

```
double applyOp(double a, double b, char op){
    switch(op){
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
}
```

- Here is the code of the function for the main process by using a *stack*.

```
double evaluatePostfix(string postfix) {
    // size of postfix (different from infix
    // because postfix has spaces to recognize numbers together)
    int m = postfix.length();

    // stack to store numbers
    // (using double because division may produce non-integer values)
    stack<double> numbers;

    // evaluate the postfix expression
```

```
    for (int i = 0; i < m; i++) {
        if (postfix[i] == ' ') continue; //skip spaces
        else if (isdigit(postfix[i]))
        {
            double number = 0;
            // extract the number
            while (i < m && isdigit(postfix[i]))
            {
                number = (number * 10) + (postfix[i] - '0');
                i++;
            }
            // push the number to 'numbers'
            numbers.push(number);
            // go back one digit because now we are at the next digit after number.
            i--;
        }
        else
        {
            // take two numbers from top of stack.
            // first top is number2 because it comes after number1.
            double number_2 = numbers.top();
            numbers.pop();

            double number_1 = numbers.top();
            numbers.pop();

            // push the result of applying the operator to the two numbers.
            numbers.push(applyOp(number_1, number_2, postfix[i]));
        }
    }

    // the final number is the result of the expression.
    return numbers.top();
}
```

It is simple for estimating the time complexity and the space complexity as they are both $O(n)$.

4.2 Prefix expression valuation

The idea of this algorithm is similiar to the *Postfix expression valuation* in 4.1. However, we scan the prefix **from right to left**.

Here is an illustrative example of the algorithm:

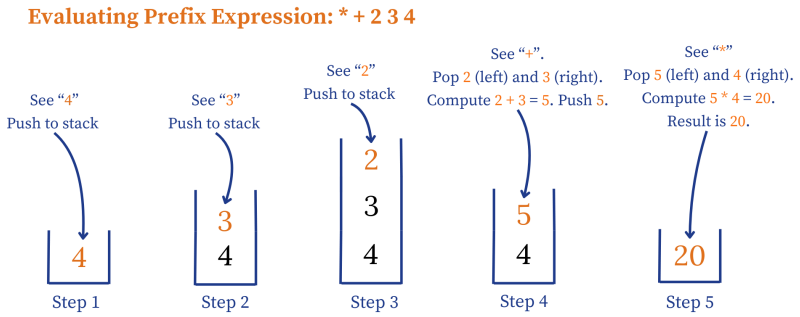


Figure 9: An example of prefix expression evaluation

Here is the code of how it is implemented:

```
double evaluatePrefix(string prefix)
{
    int m = prefix.length();
    stack<double> numbers;
    for (int i = m - 1; i >= 0; i--)
    {
        if (prefix[i] == ' ') {continue;
        }
        else if (isdigit(prefix[i]))
        {
            double number = 0, multiplier = 1;
            while (i >= 0 && isdigit(prefix[i]))
            {
                number = (number * 10) + (prefix[i] - '0') * multiplier;
                i--;
            }
            numbers.push(number); i++;
        }
```

```
        else
        {
            double number_1 = numbers.top();
            numbers.pop();
            double number_2 = numbers.top();
            numbers.pop();
            numbers.push(applyOp(number_1, number_2, prefix[i]));
        }
    }

    return numbers.top();
}
```

It is also quite simple for estimating the time complexity and the space complexity as they are both $O(n)$.

4.3 Infix expression valuation

To implement this algorithm, it is quite more complex because we have to use *two stacks*, one for operands and one for operators:

1. Use two stacks: one for **operands** and one for **operators**.
2. Scan the infix expression from left to right.
   (a) If an operand is encountered, push to the **operand** stack.
   (b) If a left parenthesis "(" is encountered, push it onto the operator stack.
   (c) If a right parenthesis ")" is encountered:
      i. Pop and apply operators until a left parenthesis is encountered.
      ii. Discard the pair of parentheses.
   (d) If an operator (+,-,*,/,^) is encountered:
      i. While there is an operator at the top of the **operator** stack with greater or equal precedence (and for left-associative operators, equal precedence as well), pop operators from the stack and apply them.
      ii. Then push the current operator into the **operator** stack.

3. End of Expression: Pop any remaining operators from the stack to the output.

Here is an illustrative example of the algorithm: let's take an infix expression " 3 + (9 - 1) / 2 " as an example.

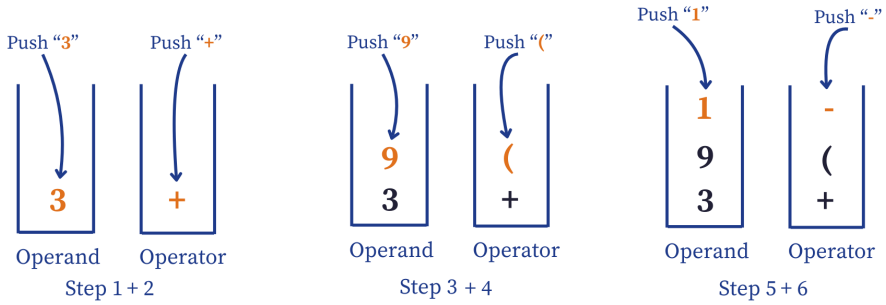


Figure 10: An example of infix expression evaluation (step 1 - 6)

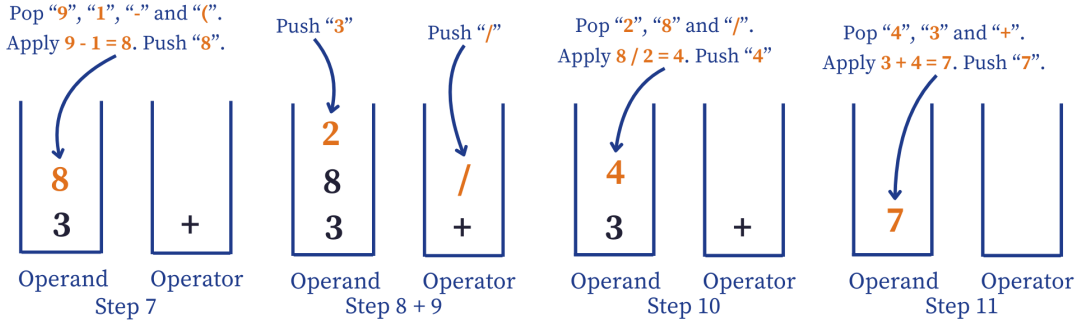


Figure 11: An example of infix expression evaluation (step 7 - 11)

While it requires two stacks to implement the algorithm, the time complexity and the space complexity are still $O(n)$.

Here is the code of how it is implemented:

```
double evaluateInfix(string infix)
{
    stack<double> numbers; // Stack to store numbers
    stack<char> ops;       // Stack to store operators

    int n = infix.length();

```

```
    for (int i = 0; i < n; i++)
    {
        char c = infix[i];

        if (c == ' ')
        {
            continue;
        }
        else if (isdigit(c))
        {
            double number = 0;
            while (i < n && isdigit(infix[i]))
            {
                number = number * 10 + (infix[i] - '0');
                i++;
            }
            numbers.push(number);
            i--;
        }
        else if (c == '(')
        {
            ops.push(c);
        }
        else if (c == ')')
        {
            while (!ops.empty() && ops.top() != '(')
            {
                double number_1 = numbers.top();
                numbers.pop();
                double number_2 = numbers.top();
                numbers.pop();
                char op = ops.top();
                ops.pop();
                numbers.push(applyOp(number_2, number_1, op));
            }
            ops.pop();
        }
```

```
45          else if (c == '+' || c == '-' || c == '*' || c == '/')
46          {
47              while (!ops.empty() && precedence(ops.top()) >= precedence(c))
48              {
49                  double b = numbers.top();
50                  numbers.pop();
51                  double a = numbers.top();
52                  numbers.pop();
53                  char op = ops.top();
54                  ops.pop();
55                  numbers.push(applyOp(a, b, op));
56              }
57              ops.push(c);
58          }
59      }
60      while (!ops.empty())
61      {
62          double b = numbers.top();
63          numbers.pop();
64          double a = numbers.top();
65          numbers.pop();
66          char op = ops.top();
67          ops.pop();
68          numbers.push(applyOp(a, b, op));
69      }
70      return numbers.top();
71  }
```

5 Stack Application

5.1 Parentheses Matching

In some programming problems, we have to check if an expression has balanced parentheses. These expressions consist of opening and closing brackets. The opening brackets can be any one of the following – '(' , '{' or '[' . We must have figured out the corresponding closing brackets, it could be any one of – ')' , '}' or ']'. This can be used to validate user input, check mathematics notation,... To solve this problem, we use a stack to check if a string or an expression contains balanced parentheses.

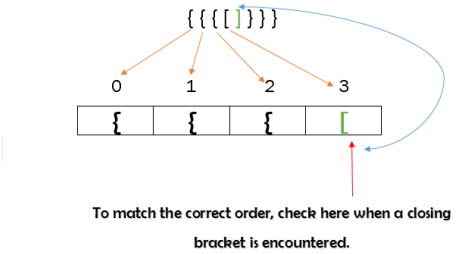


Figure 12: A stack to check parentheses matching

For simplicity, we use a stack to store the opening brackets. When we iterate the string from left to right, if we encounter an open bracket, we push it onto the stack. If we encounter a closing bracket, we check whether the top of the stack contains the corresponding opening bracket. If it does, we pop the opening bracket from the stack and continue processing the rest of the string. If it does not match or the stack is empty, the expression is unbalanced. This method allows us to efficiently verify the correctness of nested structures and ensure the proper order of brackets. [3]

5.2 Depth-First Search (Graph Traversals)

In graph traversals, Depth-first search (DFS) [7] is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. This algorithm underlies the stack concept to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

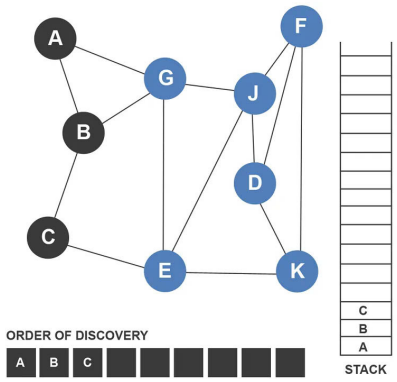


Figure 13: A stack to perform DFS

5.3 The Web browser History

In today's web browser, stacks are used to control the browser history [4]. When you use the browser, each visited page is then pushed onto a stack. This stack underlies the concept of **Last-In-First-Out (LIFO)** for navigation, for efficiency, there are **two stacks** including the **backward stack** and the **forward stack**. There are common functions of those stacks:

- **Pushing New Pages**: Every time a user navigates to a new webpage, the current page is pushed onto the history stack.
- **Navigating Back**: When the user clicks the Back button, the browser pops the top item from the history stack to retrieve the last visited page.
- **Navigating Forward**: When a page is popped from the history stack (via a Back operation), it is pushed onto the forward stack. Clicking the Forward button then pops from the forward stack, allowing users to return to the more recent page they backed out of.

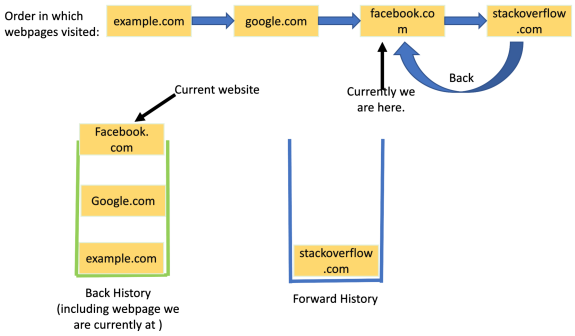


Figure 14: Two stacks to control browser history

6 Group work

6.1 Presentation video

Click here to watch our group presentation video.

6.2 Work assignment table

Here's our work assignment table:

Group member	Student ID	Task assigned	Completion Rate
Huỳnh Thái Hoàng	24127171	• Leading, supervising the project and assisting group members. • Researching and collecting data for the project (including converting algorithms, making test cases,...). • Designing and drawing pictures. (Shunting-yard algorithm). • Writing report for the group project.	100%
Đặng Hoàng Nam	24127454	• Researching and collecting data for the project (including converting between notations algorithms, making test cases,...). • Designing and drawing diagrams, tables, pictures for the topics. • Making slides for group presentation.	100%
Nguyễn Sơn Hải	24127163	• Researching and coding the main source code of the project (including converting infix notation to postfix and evaluating the postfix notation). • Developing and fixing the main source code of the project during the project.	100%

Table 5: Group member's task assignment

6.3 Self-evaluation

6.3.1 Key contributions to the project

Hai's source code is effective and understandable. Therefore, other members can rely on that source code to write for other cases. (For example: from infix notation to postfix, infix expression evaluation,...).
Nam's illustrative diagrams are designed to be simple and understandable. Making the report and the presentation more imaginative.
With the assistance of AI, our group members could research and collect data for the project quickly and precisely so we could implement our tasks easier and more productively.

6.3.2 Challenges faced

Some members in the groups have the restriction of English knowledge, so these caused some difficulties in researching the project and also doing the presentation in English.
The report writer has some difficulties in fixing errors of the report's Latex source code because of not getting familiar with the new programming language.

6.3.3 Areas for improvement

Improving English skill, teamwork skills, communication skills, presentation skills.
Improving technical skills.

6.3.4 Conclusion

More importantly, with the efforts and dedications of each group member, the project was completely done on time and no assigned tasks were due late. Hence, each completion rate of the tasks is 100%.

References

[1] Brad Miller and David Ranum. Infix, prefix and postfix expressions. https://runestone.academy/ns/books/published/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html, 2013. Accessed: 2025-04-07.

[2] GeeksforGeeks. Expression evaluation by using stack. https://www.geeksforgeeks.org/expression-evaluation/, 2024. Accessed April 6, 2025.

[3] Ajay Iyengar. How to solve the valid parentheses problem in java? https://ajayiyengar.com/2020/08/16/how-to-solve-the-valid-parentheses-problem-in-java/, 2020. Accessed: 2025-04-07.

[4] The Algorists. Web Browser History by using two stacks. https://www.thealgorists.com/LLD/WebBrowserHistory.

[5] Wikipedia contributors. Reverse Polish Notation. https://en.wikipedia.org/wiki/Reverse_Polish_notation, 2024. Accessed April 6, 2025.

[6] Wikipedia contributors. The Shunting-yard algorithm. https://en.wikipedia.org/wiki/Shunting_yard_algorithm, 2024. Accessed: 2025-04-07.

[7] Wikipedia contributors. Depth-First Search — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Depth-first_search, 2025.

[8] Wikipedia contributors. Jan Łukasiewicz. https://en.wikipedia.org/wiki/Jan_%C5%81ukasiewicz, 2025. Accessed: 2025-04-07.