Stack, Infix, Prefix, and Postfix Notations: A Comprehensive Cheatsheet

This cheatsheet provides a detailed explanation of stack data structures and their application in understanding and converting between infix, prefix, and postfix expressions. These notations are fundamental concepts in computer science, particularly in compiler design, expression evaluation, and understanding the order of operations.

1. Introduction to Expressions and Notations

In mathematics and computer science, an expression is a combination of operands (values or variables) and operators (symbols representing operations). The way these operands and operators are arranged defines the notation of the expression. While humans are most familiar with infix notation, computers often find it easier to process expressions in prefix or postfix forms due to their unambiguous nature and straightforward evaluation using stack-based algorithms.

This document will cover: * **Infix Notation**: The standard mathematical way of writing expressions. * **Prefix Notation (Polish Notation)**: Operators precede their operands. * **Postfix Notation (Reverse Polish Notation)**: Operators follow their operands. * **Stack Data Structure**: A crucial data structure for converting and evaluating these expressions. * **Conversion Algorithms**: Step-by-step processes for converting between different notations. * **Evaluation Algorithms**: How to evaluate expressions in prefix and postfix forms. * **C++ Code Examples**: Practical implementations of conversion and evaluation algorithms.

Let's dive into the details of each notation and how they relate to the stack data structure.

2. Infix Notation

Infix notation is the most common way of writing mathematical expressions, where the operator is placed *between* its operands [1]. For example, `A + B` is an infix expression, where `+` is the operator and `A` and `B` are the operands.

Characteristics of Infix Notation:

- **Operator Placement**: Operators are positioned between the operands they operate on.
- **Readability**: It is highly human-readable and intuitive, mirroring how we naturally write mathematical equations.
- **Parentheses**: Parentheses are often used to define the order of operations, overriding default operator precedence rules. For instance, in `(A + B) * C`, the addition `A + B` is performed before multiplication by `C`.
- **Operator Precedence**: Infix expressions adhere to operator precedence rules (e.g., multiplication and division have higher precedence than addition and subtraction) and associativity (e.g., left-to-right for addition and subtraction).

Example:

Consider the expression: `(A + B) * C - D / E`

Here, `+`, `-`, `*`, `/` are operators, and `A`, `B`, `C`, `D`, `E` are operands. The parentheses dictate that `A + B` is evaluated first.

Advantages of Infix Notation:

- **Natural and Familiar**: It's the notation most people are accustomed to, making it easy to write and understand for humans.
- **Widespread Use**: Supported by almost all programming languages and calculators, making it universally applicable.

Disadvantages of Infix Notation:

- **Parsing Complexity**: For computers, parsing and evaluating infix expressions can be complex due to the need to consider operator precedence, associativity, and parentheses. This often requires converting them to other notations (like postfix) for efficient processing.
- **Ambiguity**: Without proper use of parentheses or strict adherence to precedence rules, infix expressions can be ambiguous.

Operator Precedence Rules (Standard Mathematical Operators):

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

To evaluate an infix expression programmatically, it is often first converted into postfix notation, which simplifies the evaluation process significantly [1].

3. Prefix Notation (Polish Notation)

Prefix notation, also known as Polish Notation, places the operator *before* its operands [1]. This notation was developed by Polish logician Jan Łukasiewicz in 1924. For example, the infix expression `A + B` becomes `+ A B` in prefix notation.

Characteristics of Prefix Notation:

- **Operator Placement**: The operator always precedes its operands.

- **No Parentheses Needed**: One of the key advantages of prefix notation is that it inherently defines the order of operations, eliminating the need for parentheses. The position of the operator unambiguously indicates its scope.
- **Readability**: Less human-readable than infix notation, as it requires a different way of thinking about expressions.

Example:

Consider the infix expression: `(A + B) * C - D / E`

In prefix notation, this would be: `- * + A B C / D E`

Let's break down the conversion: 1. `(A + B)` becomes `+ A B` 2. `(+ A B) * C` becomes `* + A B C` 3. `D / E` becomes `/ D E` 4. `(* + A B C) - (/ D E)` becomes `- * + A B C / D E`

Advantages of Prefix Notation:

- **Unambiguous**: The order of operations is clear without the need for parentheses, simplifying parsing for computers.
- **Stack-Based Evaluation**: Easily evaluated using a stack-based algorithm, making it efficient for computer processing.

Disadvantages of Prefix Notation:

- **Less Intuitive for Humans**: It can be challenging for humans to read and write compared to infix notation, as it deviates from traditional mathematical representation.
- **Not Widely Used in Everyday Math**: Primarily used in computer science and specific programming contexts (e.g., Lisp).

Evaluation of Prefix Expressions:

To evaluate a prefix expression, you typically scan the expression from right to left. When an operand is encountered, push it onto a stack. When an operator is encountered, pop the required number of operands from the stack, perform the

operation, and push the result back onto the stack. This process continues until the entire expression is scanned, and the final result remains on the stack.

4. Postfix Notation (Reverse Polish Notation)

Postfix notation, also known as Reverse Polish Notation (RPN), places the operator *after* its operands [1]. This notation is widely used in stack-based calculators and programming languages. For example, the infix expression `A + B` becomes `A B +` in postfix notation.

Characteristics of Postfix Notation:

- **Operator Placement**: The operator always follows its operands.
- **No Parentheses Needed**: Similar to prefix notation, postfix notation is unambiguous and does not require parentheses to define the order of operations. The order of evaluation is determined solely by the position of operators and operands.
- **Readability**: While not as intuitive as infix for humans, it is generally considered more readable than prefix notation.

Example:

Consider the infix expression: `(A + B) * C - D / E`

In postfix notation, this would be: `A B + C * D E / -`

Let's break down the conversion: 1. `(A + B)` becomes `A B +` 2. `(A B +) * C` becomes `A B + C *` 3. `D / E` becomes `D E /` 4. `(A B + C *) - (D E /)` becomes `A B + C * D E / -`

Advantages of Postfix Notation:

- **Unambiguous**: Like prefix notation, it eliminates ambiguity and the need for parentheses, simplifying parsing.
- **Efficient Evaluation**: Extremely well-suited for evaluation using a stack. This makes it very efficient for computers to process.

- **Simpler Algorithms**: Conversion from infix to postfix and evaluation of postfix expressions are relatively straightforward using stack-based algorithms.

Disadvantages of Postfix Notation:

- **Less Intuitive for Humans**: Requires a different way of thinking about expressions compared to the familiar infix notation.

Evaluation of Postfix Expressions:

To evaluate a postfix expression, you typically scan the expression from left to right. When an operand is encountered, push it onto a stack. When an operator is encountered, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack. This process continues until the entire expression is scanned, and the final result remains on the stack.

5. Stack Data Structure

A stack is a fundamental linear data structure that follows the **Last In, First Out (LIFO)** principle. Imagine a stack of plates: you can only add a new plate to the top, and you can only remove the top plate. The last plate added is always the first one to be removed.

Key Operations:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element from the stack.
- **Peek (or Top)**: Returns the top element of the stack without removing it.
- **isEmpty**: Checks if the stack is empty.
- **isFull**: Checks if the stack is full (relevant for fixed-size stacks).

How Stacks Work:

When an element is `pushed` onto the stack, it becomes the new top element. When an element is `popped`, the element that was previously below it becomes the new top.

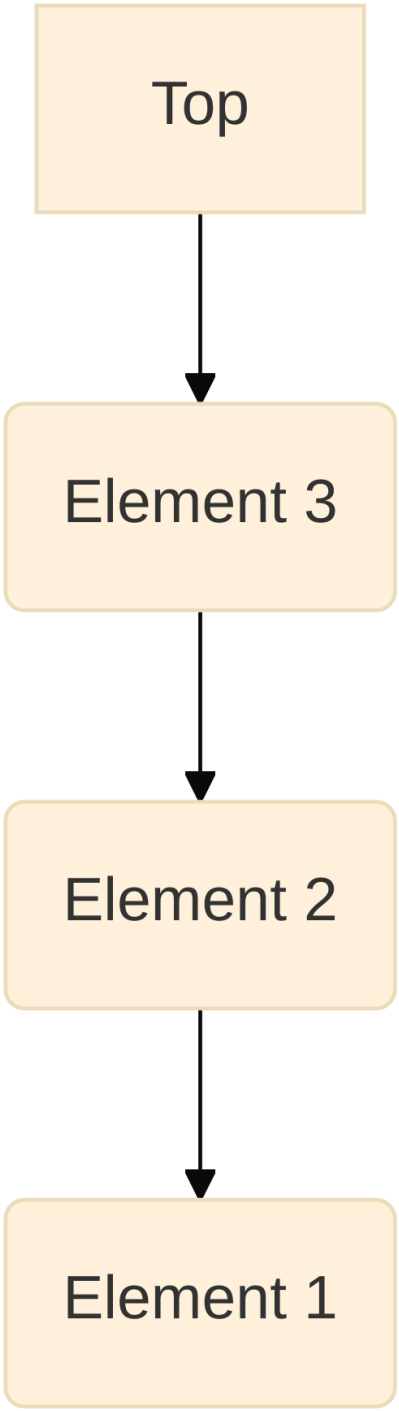
This LIFO behavior makes stacks ideal for tasks where the most recently added item needs to be processed first.

Applications of Stacks:

Stacks have numerous applications in computer science, including:

- **Expression Evaluation and Conversion**: As we will see, stacks are crucial for converting infix expressions to postfix or prefix, and for evaluating postfix and prefix expressions.
- **Function Call Management**: Compilers use stacks to manage function calls and local variables (the call stack).
- **Undo/Redo Functionality**: Many applications use stacks to implement undo and redo features.
- **Backtracking Algorithms**: Used in algorithms like depth-first search (DFS) and solving mazes.
- **Browser History**: Storing the history of visited web pages.

Visual Representation of a Stack:



In the above diagram, `Element 3` is at the top of the stack. A `push` operation would add a new element above `Element 3`, and a `pop` operation would remove `Element 3`.

6. Infix to Postfix Conversion using Stack (C++)

Converting an infix expression to a postfix expression is a common application of stacks. The algorithm involves scanning the infix expression from left to right and using a stack to temporarily store operators and parentheses. The general rules are as follows:

1. **Operands**: If the scanned character is an operand, append it to the postfix expression.
2. **Opening Parenthesis `(`**: Push it onto the stack.
3. **Closing Parenthesis `)`**: Pop operators from the stack and append them to the postfix expression until an opening parenthesis `(` is encountered. Pop and discard the opening parenthesis.
4. **Operators**: If the scanned character is an operator:
    - Pop operators from the stack and append them to the postfix expression as long as the stack is not empty, the top of the stack is not an opening parenthesis, and the operator at the top of the stack has higher or equal precedence than the scanned operator.
    - Push the scanned operator onto the stack.
5. **End of Expression**: After scanning the entire infix expression, pop any remaining operators from the stack and append them to the postfix expression.

Operator Precedence Function:

To implement the conversion, we need a function that defines the precedence of operators. A common approach is to assign integer values to operators, where higher values indicate higher precedence.

```
// Function to return precedence of operators
int prec(char c) {
    if (c ==
'^
')
        return 3;
    else if (c ==
'/'
 || c ==
'*
')
        return 2;
    else if (c ==
'+
' || c ==
'-'
)
        return 1;
    else
        return -1;
}
```

C++ Implementation for Infix to Postfix Conversion:

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

// Function to convert infix expression to postfix
std::string infixToPostfix(std::string s) {
    std::stack<char> st; // For operators
    std::string result;  // To store postfix expression

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        // If the scanned character is an operand, add it to the output string.
        if ((c >=
'a'
 && c <=
'z'
) || (c >=
'A'
 && c <=
'Z'
) || (c >=
'0'
 && c <=
'9'
)) {
            result += c;
        }
        // If the scanned character is an
'('
, push it to the stack.
        else if (c ==
'('
) {
            st.push(
'('
);
        }
        // If the scanned character is an
')'
, pop and add to the output string from the stack
        // until an
'('
 is encountered.
        else if (c ==
')'
) {
            while (!st.empty() && st.top() !=
'('
) {
                result += st.top();
                st.pop();
            }
            if (!st.empty() && st.top() ==
'('
) {
                st.pop(); // Pop
'('
```

```
            }
        }
        // If the scanned character is an operator
        else {
            while (!st.empty() && prec(s[i]) <= prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

    // Pop all the remaining elements from the stack
    while (!st.empty()) {
        result += st.top();
        st.pop();
    }

    return result;
}

int main() {
    std::string exp =
"a+b*(c^d-e)"
;
    std::cout <<
"Infix expression: "
 << exp << std::endl;
    std::cout <<
"Postfix expression: "
 << infixToPostfix(exp) << std::endl;

    exp =
"(A+B)*C-D/E"
;
    std::cout <<
"Infix expression: "
 << exp << std::endl;
    std::cout <<
"Postfix expression: "
 << infixToPostfix(exp) << std::endl;

    return 0;
}
```

Output for `a+b*(c^d-e)`:

```
 Infix expression: a+b*(c^d-e)
Postfix expression: abcd^e-*+
```

Output for `(A+B)*C-D/E`:

```
 Infix expression: (A+B)*C-D/E
Postfix expression: AB+C*DE/-
```

This code snippet demonstrates the conversion process, showcasing how operators are handled based on their precedence and how parentheses influence the order of operations. This implementation is based on the algorithm described in [2].

7. Postfix Expression Evaluation using Stack (C++)

Evaluating a postfix expression is a straightforward process that also utilizes a stack. The algorithm involves scanning the postfix expression from left to right:

1. **Operand**: If the scanned character is an operand (a number), push it onto the stack.
2. **Operator**: If the scanned character is an operator, pop the top two operands from the stack. Perform the operation with these two operands (the first popped operand is the second operand in the operation, and the second popped operand is the first). Push the result of the operation back onto the stack.
3. **End of Expression**: After scanning the entire expression, the final result will be the only element remaining on the stack. Pop it and return it.

C++ Implementation for Postfix Evaluation:

```
 #include <iostream>
#include <stack>
#include <string>
#include <cctype> // For isdigit()

// Function to evaluate a postfix expression
int evaluatePostfix(std::string exp) {
    std::stack<int> st;

    for (int i = 0; i < exp.length(); i++) {
        // If the scanned character is an operand (number), push it to the
stack.
        if (isdigit(exp[i])) {
            st.push(exp[i] -
'0'
); // Convert char to int
        }
        // If the scanned character is an operator, pop two elements from the
stack and apply the operator.
        else {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();

            switch (exp[i]) {
            case
'+
':
                st.push(val2 + val1);
                break;
            case
'-
':
                st.push(val2 - val1);
                break;
            case
'*
':
                st.push(val2 * val1);
                break;
            case
'/'
:
                st.push(val2 / val1);
                break;
            case
'^
': // Exponentiation
                {
                    int res = 1;
                    for (int j = 0; j < val1; j++) {
                        res *= val2;
                    }
                    st.push(res);
                }
                break;
            }
```

```
        }
    }
    return st.top();
}

int main() {
    std::string exp =
"231*+9-"
;
    std::cout <<
"Postfix expression: "
 << exp << std::endl;
    std::cout <<
"Result: "
 << evaluatePostfix(exp) << std::endl;

    exp =
"82/3-32*+"
;
    std::cout <<
"Postfix expression: "
 << exp << std::endl;
    std::cout <<
"Result: "
 << evaluatePostfix(exp) << std::endl;

    return 0;
}
```

Output for `231*+9-`:

```
Postfix expression: 231*+9-
Result: -4
```

Explanation: 1. `2` is pushed. Stack: `[2]` 2. `3` is pushed. Stack: `[2, 3]` 3. `1` is pushed. Stack: `[2, 3, 1]` 4. `*` is encountered. Pop `1` and `3`. `3 * 1 = 3`. Push `3`. Stack: `[2, 3]` 5. `+` is encountered. Pop `3` and `2`. `2 + 3 = 5`. Push `5`. Stack: `[5]` 6. `9` is pushed. Stack: `[5, 9]` 7. `-` is encountered. Pop `9` and `5`. `5 - 9 = -4`. Push `-4`. Stack: `[-4]`

Output for `82/3-32*+`:

```
Postfix expression: 82/3-32*+
Result: 7
```

Explanation: 1. `8` is pushed. Stack: `[8]` 2. `2` is pushed. Stack: `[8, 2]` 3. `/` is encountered. Pop `2` and `8`. `8 / 2 = 4`. Push `4`. Stack: `[4]` 4. `3` is pushed. Stack: `[4, 3]` 5. `-` is encountered. Pop `3` and `4`. `4 - 3 = 1`. Push `1`. Stack: `[1]` 6. `3` is pushed. Stack: `[1, 3]` 7. `2` is pushed. Stack: `[1, 3, 2]` 8. `*` is encountered. Pop `2`

and `3`. `3 * 2 = 6`. Push `6`. Stack: `[1, 6]` 9. `+` is encountered. Pop `6` and `1`. `1 + 6 = 7`. Push `7`. Stack: `[7]`

This code provides a simple implementation for evaluating postfix expressions containing single-digit numbers. For more complex scenarios involving multi-digit numbers or floating-point values, the parsing logic would need to be extended. This implementation is based on the algorithm described in [3].

8. References

[1] GeeksforGeeks. (2024, March 21). *Infix, Postfix and Prefix Expressions/Notations*. GeeksforGeeks. https://www.geeksforgeeks.org/infix-postfix-prefix-notation/

[2] GeeksforGeeks. (2024, March 18). *Infix to Postfix Conversion using Stack in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp/infix-to-postfix-conversion-using-stack-in-cpp/

[3] GeeksforGeeks. (2024, March 18). *How to Evaluate a Postfix Expression using Stack in C++?*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp/how-to-evaluate-a-postfix-expression-using-stack-in-cpp/

Visual Comparison of Notations: