Data Structures & Algorithms (C++)

Graphs + Expression Notations

1. Graphs

Introduction

A graph is a **non-linear data structure** consisting of vertices (nodes) and edges (connections between nodes).

Graphs are used to represent real-world systems such as computer networks, social media connections, maps, and routing systems.

A graph is formally defined as **G = (V, E)** where V is a set of vertices and E is a set of edges.

Types of Graphs

- Directed Graph (Digraph)
- Undirected Graph
- Weighted Graph
- Unweighted Graph

2. Directed and Undirected Graphs

Directed Graph

A directed graph (digraph) is a graph where edges have a direction.

If there is an edge from A → B, it does not necessarily mean B → A exists.

Used in:

- Web pages (links)
- One-way roads
- Task scheduling

Undirected Graph

An undirected graph has edges without direction.

If A is connected to B, then B is also connected to A.

Used in:

- Social networks (friendship)
- Two-way roads
- Communication networks

Example

Directed: A → B → C

Undirected: A — B — C

Key Difference

Feature	Directed	Undirected
Edge Direction	One-way	Two-way
Representation	(A→B)	(A—B)
Use Case	Web links	Social networks

3. Adjacency Matrix

Introduction

An adjacency matrix is a **2D array representation of a graph** where rows and columns represent vertices.

If there is an edge between nodes, the value is 1 (or weight), otherwise 0.

It is very efficient for dense graphs.

Representation

For graph with V vertices:

- matrix size = V × V

Example

```
  A B C
A [ 0 1 1 ]
```

B [1 0 0]
C [1 0 0]

C++ Implementation

```
#include <iostream>
using namespace std;

int main() {
   int v = 3;
   int adj[3][3] = {0};

   // edges
   adj[0][1] = 1;
   adj[0][2] = 1;
   adj[1][0] = 1;
   adj[2][0] = 1;

   cout << "Adjacency Matrix:\n";

   for (int i = 0; i < v; i++) {
      for (int j = 0; j < v; j++) {
         cout << adj[i][j] << " ";
      }
      cout << endl;
   }

   return 0;
}
```

Advantages

- Simple representation
- Fast edge lookup O(1)

Disadvantages

- Uses $O(V^2)$ space
- Inefficient for sparse graphs

4. Infix, Prefix, and Postfix Notations

Introduction

These are ways of writing arithmetic expressions without ambiguity.

They are widely used in compilers and expression evaluation systems.

Types

Infix

Operator between operands:
A + B

Prefix (Polish Notation)

Operator before operands:
+ A B

Postfix (Reverse Polish)

Operator after operands:
A B +

Example

Infix : A + B
Prefix : + A B
Postfix : A B +

5. Use of These Notations

Why They Matter

These notations are used in:

- Compiler design
- Expression evaluation
- Stack-based calculations
- Parsing mathematical expressions

Advantages

- No parentheses needed (prefix/postfix)
- Easier for machines to evaluate
- Faster computation using stacks

6. Conversion Using Tree Method

Idea

Expressions can be converted using an **expression tree**.

- Infix = Inorder traversal
- Prefix = Preorder traversal
- Postfix = Postorder traversal

Example Tree

Conversion Rules

Traversal	Output
Inorder	Infix
Preorder	Prefix
Postorder	Postfix

Example

Expression: A + B

- Infix: A + B
- Prefix: + A B
- Postfix: A B +

7. Direct Method Conversion

Introduction

Direct method converts expressions without building a tree.

It uses operator precedence rules directly.

Key Rules

- Operators have precedence: * / > + -
- Use stack or rule-based scanning
- Scan left to right or right to left

Idea

- Prefix: scan right → left
- Postfix: scan left → right

8. Infix to Postfix using Stack Method

Introduction

This is the most important method used in compilers.

It uses a **stack to store operators** and output queue for result.

Algorithm

1. Scan expression left to right
2. If operand → output
3. If operator → push to stack
4. Pop based on precedence
5. Pop all remaining operators

Pseudocode

```
for each symbol in expression:
   if operand:
      output
   else if operator:
      while stack not empty and precedence(top) >= precedence(current):
         pop stack to output
      push operator

pop remaining stack
```

C++ Implementation

```
#include <iostream>
#include <stack>
using namespace std;
```

```
int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

string infixToPostfix(string expr) {
    stack<char> s;
    string result = "";

    for (char c : expr) {
        if (isalnum(c)) {
            result += c;
        }
        else if (c == '(') {
            s.push(c);
        }
        else if (c == ')') {
            while (!s.empty() && s.top() != '(') {
                result += s.top();
                s.pop();
            }
            s.pop();
        }
        else {
            while (!s.empty() && precedence(s.top()) >= precedence(c)) {
                result += s.top();
                s.pop();
            }
            s.push(c);
        }
    }

    while (!s.empty()) {
        result += s.top();
        s.pop();
    }

    return result;
}

int main() {
    string expr = "A+B*C";
    cout << "Postfix: " << infixToPostfix(expr) << endl;
    return 0;
}
```

Comparison Table

Notations

Type	Form
Infix	A + B
Prefix	+ A B
Postfix	A B +

Graph Representation

Type	Space	Best Use
Adjacency Matrix	$O(V^2)$	Dense graphs
Edge List	O(E)	Sparse graphs

Key Takeaways

- Graphs represent relationships between nodes.
- Directed graphs have one-way edges; undirected are two-way.
- Adjacency matrix is a simple 2D representation.
- Infix, prefix, postfix are expression formats used in compilers.
- Tree traversal connects directly to expression conversion.
- Stack-based conversion is the most important compiler technique.
- Postfix evaluation is faster and used in calculators and compilers.