-Vikas Anant Pawar

MCS – 208 Important Topics.

---

1. Postfix Expression Conversion

Postfix Expression, also known as Reverse Polish Notation (RPN), is a way of writing arithmetic expressions without the use of parentheses and while maintaining the correct order of operations. In postfix notation, the operator comes after the operands, unlike infix notation, where the operator is between the operands.

Why Use Postfix?

Postfix expressions are easier for computers to evaluate because they eliminate the need for parentheses and reduce the complexity of operator precedence. They are commonly used in stack-based expression evaluations and compilers.

Basic Idea:

In an infix expression like A + B * C, the operator * has higher precedence than +, so B * C is evaluated first. But in postfix notation, this is written as A B C * + which automatically reflects the correct order of evaluation without needing precedence rules or parentheses.

Operator Precedence and Associativity:

When converting from infix to postfix, operator precedence and associativity play a crucial role. The common precedence rules are:

- * and / have higher precedence than + and -
- Operators with equal precedence are evaluated left to right (left associativity), except exponentiation ^, which is right associative.

Steps to Convert Infix to Postfix:

1. **Scan** the expression from left to right.
2. **Operands** (e.g., A, B, 1, 2) are added directly to the output (postfix) string.
3. **Operators** are pushed onto a stack.
4. **Parentheses**:
    - If an opening parenthesis (is encountered, it is pushed onto the stack.

- If a closing parenthesis) is encountered, pop operators from the stack to the output until an opening parenthesis is encountered.

Stack Role:

A stack is used to temporarily hold operators until they are needed. When an operator with lower precedence is encountered, higher precedence operators from the stack are popped and added to the output.

Example:

Convert (A + B) * C to postfix:

- Read (→ push to stack
- Read A → add to output → A
- Read + → push to stack
- Read B → add to output → A B
- Read) → pop + from stack → A B +
- Read * → push to stack
- Read C → add to output → A B + C
- End → pop * → final: A B + C *

Application:

Postfix is useful in:

- Expression evaluation by compilers and interpreters.
- Stack machine design (e.g., JVM, HP calculators).
- Converting infix expressions programmatically in parsers.

Algorithm (Pseudocode):

```
Initialize an empty stack and empty postfix list
For each token in the infix expression:
  If operand → add to postfix
  If '(' → push to stack
  If ‘)’ → pop until ‘(’ and append to postfix
  If operator:
    While precedence(top of stack) ≥ precedence(current):
      Pop and add to postfix
    Push current operator
Pop remaining operators and append to postfix
```

Conclusion:

Postfix notation is essential in computer science for parsing and evaluating expressions efficiently. Its use of stack simplifies expression evaluation without relying on complex operator precedence rules or parenthesis matching. Mastering postfix conversion also helps in understanding the internal working of compilers and interpreters.

---

2. Linear Search (Algorithm)

Linear Search, also known as **sequential search**, is the simplest method of searching an element in a list. It checks each element of the list one by one until the target element is found or the end of the list is reached.

How It Works:

Suppose we have an array arr[] = {10, 25, 31, 48, 50} and we are searching for 31. A linear search begins from the first index:

- Compare 10 → not equal
- Compare 25 → not equal
- Compare 31 → found (at index 2)

Algorithm Steps:

1. Start from the first element of the array.
2. Compare each element with the target key.
3. If the element matches the key, return its position/index.
4. If not found after traversing the list, return -1 or indicate “not found”.

Pseudocode:

```
LinearSearch(arr, n, key):
 for i = 0 to n-1:
  if arr[i] == key:
   return i
 return -1
```

Time Complexity:

- **Best Case**: O(1) → if the element is found at the first index.
- **Worst Case**: O(n) → if the element is at the end or not present.
- **Average Case**: O(n/2) ≈ O(n)

Space Complexity:

- O(1) as no additional space is required.

Characteristics:

- Simple and easy to implement.
- Does not require a sorted array.
- Suitable for small data sets.

Limitations:

- Inefficient for large datasets.
- Time-consuming compared to binary search (for sorted data).

Real-Life Applications:

- Searching names in an unsorted attendance list.
- Finding a specific entry in an unsorted log file.

---

3. Matrix Multiplication (Algorithm)

Matrix multiplication is a binary operation that produces a matrix from two matrices. It is a fundamental concept in mathematics, used widely in computer graphics, neural networks, and linear algebra applications.

Rules for Matrix Multiplication:

If A is of size (m × n) and B is of size (n × p), then the resulting matrix C will be of size (m × p). The number of columns in A must equal the number of rows in B.

Algorithm:

Each element of the resulting matrix C[i][j] is calculated as:

$C[i][j] = \Sigma (A[i][k] * B[k][j])$ for $k = 0$ to $n-1$

Example:

A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]

C = [[(1×5 + 2×7), (1×6 + 2×8)],
[(3×5 + 4×7), (3×6 + 4×8)]]
= [[19, 22], [43, 50]]

Time Complexity:

- $O(m \times n \times p)$
- For square matrices (n × n), it's $O(n^3)$

Applications:

- Transformations in computer graphics
- Solving systems of linear equations
- Representing data in machine learning

Optimized Methods:

- Strassen's Algorithm ($O(n^{2.81})$)
- Coppersmith–Winograd (theoretical)

---

4. Stack – Implementation / Operations

A **stack** is a linear data structure following the **LIFO** (Last-In, First-Out) principle. It allows insertion (push) and deletion (pop) operations only at the top of the stack.

Basic Operations:

- **Push(x)**: Inserts element x at the top

- **Pop()**: Removes and returns top element
- **Peek()**: Returns the top element without removing
- **isEmpty()**: Checks if stack is empty

Stack Implementation (Array):

```
#define SIZE 100
int stack[SIZE], top = -1;

void push(int x) {
  if (top == SIZE - 1) printf("Stack Overflow");
  else stack[++top] = x;
}

int pop() {
  if (top == -1) printf("Stack Underflow");
  else return stack[top--];
}
```

Time Complexity:

- Push/Pop/Peek: O(1)

Applications:

- Undo functionality in editors
- Expression evaluation and conversion
- Syntax parsing (e.g., HTML tags)
- Recursion stack in programming languages

---

5. Queue – Implementation / Deletion / Operations

A **Queue** is a linear data structure that follows the **FIFO** (First-In, First-Out) principle. This means the element inserted first will be removed first, similar to a line in real life—people who enter the queue first are served first.

Basic Queue Operations:

1. **Enqueue(x)**: Adds element x to the rear (end).
2. **Dequeue()**: Removes and returns the front element.
3. **Front()**: Returns the front item without removing it.
4. **isEmpty()**: Checks if the queue has no elements.
5. **isFull()**: (Only for static arrays) Checks if the queue is full.

Implementation:

Using Array (Static Queue):

```
#define SIZE 100

int queue[SIZE];

int front = -1, rear = -1;


void enqueue(int x) {

  if (rear == SIZE - 1) printf("Overflow");

  else {

    if (front == -1) front = 0;

    queue[++rear] = x;

  }

}


int dequeue() {

  if (front == -1 || front > rear) printf("Underflow");

  else return queue[front++];

}
```

Using Linked List (Dynamic Queue):

No size limitation; we use nodes with pointers.

Types of Queues:

- **Simple Queue**: Basic FIFO structure.
- **Circular Queue**: Last position is connected to the first to reuse space.
- **Double-Ended Queue (Deque)**: Insertion and deletion at both ends.
- **Priority Queue**: Elements have priorities; high-priority items dequeued first.

Time Complexity:

- Enqueue: O(1)
- Dequeue: O(1)

Applications:

- CPU Scheduling
- Printer queue
- Call center systems
- BFS in Graphs

---

6. Bubble Sort (Algorithm)

Bubble Sort is a simple sorting algorithm that compares each pair of adjacent elements and swaps them if they are in the wrong order. It is one of the easiest sorting methods to understand, but not efficient for large data sets.

Working:

- Iterate through the array.
- Compare adjacent elements.
- Swap if left element > right element.
- Repeat until the array is sorted.

Pseudocode:

```
for i = 0 to n-1
```

```
 for j = 0 to n-i-1
   if arr[j] > arr[j+1]
     swap(arr[j], arr[j+1])
```

Example:

Given array: [5, 1, 4, 2, 8]
After 1st pass: [1, 4, 2, 5, 8]
After 2nd pass: [1, 2, 4, 5, 8]
...and so on.

Time Complexity:

- Best Case (Already Sorted): O(n)
- Average Case: O(n^2)
- Worst Case: O(n^2)

Space Complexity:

- O(1) (In-place sorting)

Applications:

- Teaching basic sorting principles
- Simple applications with small data sets

---

7. Dijkstra's Algorithm

Dijkstra's Algorithm is a **greedy algorithm** used to find the **shortest path** from a single source vertex to all other vertices in a weighted graph with **non-negative weights**.

Steps:

1. Mark all vertices with infinity distance.
2. Set distance of source to 0.
3. Pick the unvisited node with the smallest distance.
4. Update distance for adjacent vertices.
5. Mark current node as visited.

6. Repeat until all nodes are visited.

Pseudocode:

```
Initialize dist[] with ∞, dist[src] = 0
While Q is not empty:
  u = extract-min(Q)
  For each neighbor v of u:
    if dist[u] + weight(u, v) < dist[v]
       dist[v] = dist[u] + weight(u, v)
```

Time Complexity:

- With array: $O(V^2)$
- With min-heap: $O((V + E) \log V)$

Applications:

- GPS systems
- Network routing (e.g., OSPF)
- Flight and road map systems

---

8. Kruskal's Algorithm

Kruskal's Algorithm is used to find the **Minimum Spanning Tree (MST)** of a connected, undirected graph. It uses the **greedy approach** by selecting the shortest edge that doesn't form a cycle.

Steps:

1. Sort all edges by weight.
2. Initialize MST as empty.
3. For each edge in sorted list:
    - If adding the edge does not form a cycle, include it.
4. Stop when MST has (V – 1) edges.

Cycle Detection:

Use **Disjoint Set (Union-Find)** to detect cycles efficiently.

Time Complexity:

- Sorting edges: O(E log E)
- Union-Find: Nearly constant with path compression

Applications:

- Network design (electrical grids, roads, telecommunication)
- Clustering algorithms

---

9. Traversal of Binary Tree

Binary Tree Traversal is the process of visiting each node in a binary tree exactly once in a systematic way. There are two main categories:

- **Depth-First Traversals**: Inorder, Preorder, Postorder
- **Breadth-First Traversal**: Level Order

A. Inorder Traversal (Left → Root → Right)

Example for tree:

```
   1
  / \
 2   3
```

Inorder: 2, 1, 3

B. Preorder Traversal (Root → Left → Right)

Same tree: Preorder: 1, 2, 3

C. Postorder Traversal (Left → Right → Root)

Same tree: Postorder: 2, 3, 1

Recursive Inorder Code:

```
void inorder(Node* root) {
  if (root != NULL) {
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
  }
}
```

Application of Traversals:

- **Inorder**: For BST, gives sorted order
- **Preorder**: Used in expression trees, serialization
- **Postorder**: Used in file systems, deleting trees

Time & Space:

- Time: O(n)
- Space: O(h) (recursive stack, h = height)

---

10. Tree vs Binary Tree

Aspect	**General Tree**	**Binary Tree**
Definition	Each node can have many children	Each node has max two children
Structure	N-ary	Left & Right child
Use Cases	Hierarchical data, XML, file systems	Expression trees, BSTs
Memory Usage	Uses child lists or arrays	Uses two pointers per node
Traversal	Varies	Preorder, Inorder, Postorder

Summary:

All binary trees are trees, but not all trees are binary trees. Binary trees have a well-defined structure, making them ideal for efficient searching and manipulation, especially in BSTs and heaps.

---

11. Doubly Linked List

A **Doubly Linked List** (DLL) is a linear data structure where each node contains:

- **Data**
- A pointer to the **next node**
- A pointer to the **previous node**

Node Structure:

```
struct Node {
  int data;
  struct Node* prev;
  struct Node* next;
};
```

Operations:

- **Insert at Head**:
    - Create new node
    - Point new node's next to current head
    - Update current head's prev to new node
- **Delete Node**:
    - Update previous node's next to current's next
    - Update next node's prev to current's prev
- **Traverse forward and backward**

Advantages:

- Bidirectional traversal
- Easy insertion/deletion from both ends

Disadvantages:

- Extra memory for prev pointer
- Complex implementation than singly list

Applications:

- Navigation systems (forward/backward)
- Undo-redo functionality
- Music/video players

---

12. Quick Sort (Algorithm + Trace)

Quick Sort is a **divide-and-conquer** sorting algorithm known for its high performance on average.

Idea:

1. Pick a **pivot**
2. **Partition** the array: elements < pivot to left, > pivot to right
3. **Recursively sort** left and right partitions

Example:

Array: [10, 80, 30, 90, 40, 50, 70], pivot = 70
Partition: [10, 30, 40, 50] [70] [80, 90]

Pseudocode:

```
QuickSort(arr, low, high):
  if (low < high):
    pi = partition(arr, low, high)
    QuickSort(arr, low, pi - 1)
    QuickSort(arr, pi + 1, high)
```

Time Complexity:

- Best: O(n log n)
- Worst: $O(n^2)$
- Average: O(n log n)

Space Complexity:

- O(log n) due to recursion

Applications:

- Search engines
- Efficient sorting of large datasets

---

13. Time Complexity vs Storage (Space) Complexity

In algorithm analysis, two key metrics are used to evaluate performance: **time complexity** and **space (storage) complexity**.

Time Complexity:

- Refers to the **amount of time** an algorithm takes to run as a function of the size of its input.
- Expressed using **Big-O notation** like O(1), O(n), O(n^2), etc.
- Time complexity helps predict **scalability**—how fast or slow a program becomes as data size increases.

Examples:

- Linear search: O(n)
- Binary search: O(log n)
- Nested loops: O(n^2)
- Merge sort: O(n log n)

Space Complexity:

- Refers to the **amount of memory** an algorithm uses during execution.
- Includes:
  - **Input space**: memory for input data
  - **Auxiliary space**: temporary memory for variables, recursion stack, etc.

Examples:

- Storing an array of size n: O(n)
- Using recursion in quicksort: O(log n)

Key Differences:

Aspect	**Time Complexity**	**Space Complexity**
Measures	Execution time	Memory usage
Focus	Speed/performance	RAM/disk usage
Affected by	Algorithm design, loops	Data structures, recursion
Optimization goal	Reduce time taken	Reduce memory use

Why Both Matter:

- On low-end systems, space matters more.
- On real-time systems, time may be critical.
- A **tradeoff** often exists—faster algorithms may use more memory and vice versa.

---

14. Tries and Characteristics

A **Trie** (pronounced "try") is a **tree-like data structure** used to store dynamic sets or associative arrays where the keys are usually strings.

Structure:

- Each node represents a **character**
- Paths from root to node form **prefixes**
- Each node may have **multiple children**
- Final nodes may have a **flag** indicating a complete word

Example:

Inserting "cat", "cap", and "can" into a trie creates branching on shared prefix "ca".

Advantages:

- Efficient prefix-based searching
- Lookup time proportional to length of string (O(k))
- Auto-completion and dictionary lookup

Operations:

- **Insert**: Add new word by traversing and creating nodes as needed

- **Search**: Traverse nodes matching each character
- **Delete**: Remove word and prune unused branches

Applications:

- **Autocomplete** in search engines
- **Spell checkers**
- **IP routing**
- **DNA sequence storage**

Time Complexity:

- Insert/Search/Delete: O(k), where k is length of the key

---

15. Hashing and Load Factor

Hashing is a technique used to map data of arbitrary size to fixed-size values, usually for efficient data retrieval in **hash tables**.

Components:

- **Hash Function**: Converts a key into a valid array index
- **Hash Table**: Array where data is stored using the index returned by hash function

Example Hash Function:

index = key % table_size

Load Factor:

- Denoted by α = (Number of elements) / (Size of table)
- Measures how full the table is
- High load factor → more collisions → degraded performance

Collision Resolution:

1. **Chaining**: Store multiple elements in a linked list at the same index
2. **Open Addressing**:
    - Linear Probing

    - Quadratic Probing
    - Double Hashing

Rehashing:

- When load factor exceeds threshold, create a new larger table and reinsert all existing elements.
- Helps maintain O(1) performance.

Time Complexity:

- Average: O(1) for insert/search/delete
- Worst: O(n) if all elements collide

Applications:

- Symbol tables in compilers
- Databases
- Password storage (secure hashes)
- Caches
