## Significance of Expression Conversion and Evaluation:

1. **Ease of Evaluation for Computers:**
   - **Infix expressions** (e.g., a + b * c) are natural for humans to read, but they are harder for computers to evaluate directly due to the need to follow operator precedence and associativity rules.
   - **Postfix (Reverse Polish Notation)** and **Prefix (Polish Notation)** are more suitable for computers because they eliminate the need for parentheses and the complexities of operator precedence. The expressions can be evaluated directly using stacks, making them easier and faster for machines to process.
2. **Stack-based Evaluation:**
   - Both prefix and postfix expressions can be easily evaluated using stacks, which are a fundamental data structure in computer science. This makes the evaluation process efficient in terms of both time and space.
   - **Postfix evaluation** involves a simple left-to-right scan where operands are pushed to a stack and operations are performed whenever an operator is encountered. Similarly, **prefix evaluation** involves a right-to-left scan.
3. **Compilers and Interpreters:**
   - In the field of **compilers**, expression conversion is crucial. When a high-level programming language is compiled into machine code, infix expressions are often converted into postfix or prefix forms for easier generation of assembly or machine instructions.
   - **Expression conversion** helps compilers evaluate arithmetic expressions during the translation from source code to target code, optimizing the computational resources.
4. **Simplification of Parentheses Handling:**
   - In infix notation, parentheses are required to indicate the order of operations. However, both postfix and prefix notations inherently define the order of operations without the need for parentheses. This simplifies the expression, especially for nested or complex operations.
5. **Improved Efficiency in Algorithms:**
   - Many algorithms and systems (e.g., calculators, interpreters, and parsers) rely on the conversion of expressions into postfix or prefix forms. For instance, the **Shunting Yard algorithm** is commonly used to convert infix to postfix for efficient parsing and evaluation.
   - Postfix and prefix notations enable **faster computations** as they reduce the number of steps required to determine the correct order of operations.
6. **Multiple Applications in Mathematical Systems:**
   - These conversions are also useful in mathematical systems such as symbolic computation tools (e.g., MATLAB, Mathematica), where expressions are simplified and transformed to optimize calculations.
   - **Postfix expressions** are particularly useful in stack-based CPU architectures, such as the Java Virtual Machine (JVM), where instructions are executed in a stack-based manner.

## Key Takeaways:

- **Conversion from infix to postfix or prefix** helps in simplifying the evaluation process by eliminating parentheses and precedence rules.
- **Evaluation** of postfix and prefix expressions is efficient and direct, making them ideal for use in computer systems and compilers.
- **Expression conversion** is essential for optimizing compilers, interpreters, and certain hardware architectures (e.g., stack-based processors).

Pseudo code to convert an infix expression into a postfix expression using the stack data structure:

## Working principle:

1. **Operands** are added directly to the postfix expression.
2. **Operators** are pushed to the stack but only after popping higher or equal precedence operators from the stack.
3. **Parentheses** are used to enforce precedence in infix expressions, so they are handled separately.
4. Finally, any remaining operators in the stack are appended to the postfix expression.

```
Function infixToPostfix(expression):
    Initialize an empty stack called operatorStack
    Initialize an empty string called postfixExpression
Define a function precedence(operator): // to get precedence of op
    if operator is '+' or '-':
        return 1
    else if operator is '*' or '/':
        return 2
    else if operator is '^':
        return 3
    return 0  // for non-operators
For each character ch in the expression: // to handle token while conversion
    if ch is an operand (number/variable):
        Append ch to postfixExpression
    else if ch is '(':
        Push ch onto operatorStack
    else if ch is ')':
        While operatorStack is not empty and top of operatorStack is not '(':
            Pop from operatorStack and append to postfixExpression
            Pop '(' from operatorStack
    else if ch is an operator ('+', '-', '*', '/', '^'):
        While operatorStack is not empty and precedence of top of operatorStack is
        greater than or equal to precedence of ch:
            Pop from operatorStack and append to postfixExpression
            Push ch onto operatorStack
While operatorStack is not empty: // to handle end of input string
    Pop from operatorStack and append to postfixExpression
Return postfixExpression
```

# Queue Data Structure

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. This means that the first element added to the queue will be the first one to be removed. Think of it like a line of people waiting to buy tickets: the person who arrives first is the one who gets served first.

## Key Characteristics of a Queue:

1. **FIFO Order**: Elements are processed in the order they were added.
2. **Two Main Operations**:
    - **Enqueue**: Adding an element to the back of the queue.
    - **Dequeue**: Removing an element from the front of the queue.
3. **Peek/Front**: Accessing the element at the front without removing it.
4. **Size**: Tracking the number of elements in the queue.

## Types of Queues:

1. **Simple Queue**: A basic queue with standard enqueue and dequeue operations.
2. **Circular Queue**: The last position is connected back to the first position to make it circular, optimizing space utilization.
3. **Priority Queue**: Elements are dequeued based on priority rather than the order they were enqueued.
4. **Double-ended Queue (Deque)**: Elements can be added or removed from both ends.

## Queue Implementation:

Queues can be implemented using various data structures:

1. **Array-based Implementation**:
    - A fixed-size array is used to store elements.
    - Two pointers (front and rear) track the positions for dequeuing and enqueuing.
    - Queue Operations:
        1. **Enqueue (Insert an element)**
        2. **Dequeue (Remove an element)**
        3. **Peek/Front (Access the front element)**
        4. **isFull (Check if the queue is full)**
        5. **isEmpty (Check if the queue is empty)**

    - Pseudo code for Queue ADT using Sequential Organization:

```
Initialize:
   size = maximum size of the queue
   queue[size] = an empty array of size 'size'
   front = -1   // points to the front of the queue
   rear = -1    // points to the rear of the queue

Function isEmpty():
   If front == -1:
      return True
   Else:
      return False

Function isFull():
        If rear == size - 1:
            return True
        Else:
            return False

    Function enqueue(element):
        If isFull() is True:
            Print "Queue is full"
        Else:
            If front == -1:      // Queue is empty, initialize front and rear
                front = 0
            rear = rear + 1
            queue[rear] = element
            Print "Enqueued:", element

    Function dequeue():
        If isEmpty() is True:
            Print "Queue is empty"
        Else:
            dequeued_element = queue[front]
            If front == rear:     // Only one element was present
                front = -1        // Reset queue to empty state
                rear = -1
            Else:
                front = front + 1
            Print "Dequeued:", dequeued_element

    Function peek():
        If isEmpty() is True:
            Print "Queue is empty"
        Else:
            Print "Front element:", queue[front]

    Function displayQueue():
        If isEmpty() is True:
            Print "Queue is empty"
        Else:
            For i = front to rear:
                Print queue[i]
```

- ## Pictorial Demonstration:

Let’s consider a queue with a size of 5.

The array representing the queue has five slots, and front and rear pointers keep track of the elements' positions in the queue.

Initially:

Queue: [__, __, __, __, __]     // Queue is empty

Front = -1, Rear = -1

*1. Enqueue 10*

- enqueue(10) adds 10 to the queue.

Queue: [10, __, __, __, __]     // 10 is enqueued
Front = 0, Rear = 0

*2. Enqueue 20*

- enqueue(20) adds 20 to the queue.

Queue: [10, 20, __, __, __]     // 20 is enqueued
Front = 0, Rear = 1

2. **Linked List-based Implementation**:
   - A linked list is used where each node points to the next node.
   - The head of the list represents the front, and a pointer to the tail represents the rear.

- Queues are used in various applications like:
  - Managing requests in servers (e.g., print queues).
  - Breadth-first search (BFS) in graph algorithms.
  - Task scheduling in operating systems.

# Circular queue

A **circular queue** is a variation of a linear queue where the last position is connected back to the first position, creating a circular structure. This design allows for efficient use of space, as it prevents the "wasted" empty spaces that can occur in a linear queue when elements are dequeued.

## Key Characteristics of a Circular Queue:

1. **Circular Structure**: When the end of the queue is reached, the next element is added at the beginning if there is space.
2. **Two Pointers**:
   - **Front**: Points to the first element of the queue.
   - **Rear**: Points to the last element of the queue.
3. **Full and Empty Conditions**:
   - A circular queue is considered **full** if moving the rear pointer one step forward would lead it to the front pointer.
   - It is considered **empty** when the front and rear pointers are at the same position.

## Circular Queue Operations:

1. **Enqueue (Insert an element)**
2. **Dequeue (Remove an element)**
3. **Peek/Front (Access the front element)**
4. **isFull (Check if the queue is full)**
5. **isEmpty (Check if the queue is empty)**

## Explanation of Circular Queue Operations:

1. **Enqueue**:
   - When an element is added, the rear pointer moves forward circularly using the formula (rear + 1) % size.
   - If the queue is initially empty, both front and rear are set to 0 when the first element is added.
2. **Dequeue**:
   - When an element is removed, the front pointer moves forward circularly using the formula (front + 1) % size.
   - If the queue becomes empty after dequeuing the last element, both front and rear are reset to -1.
3. **isEmpty**:
   - The queue is empty if front == -1.
4. **isFull**:
   - The queue is full if (rear + 1) % size == front, meaning the next position for rear would overlap with front.

## Pseudocode for Circular Queue ADT using Sequential Organization:

```
Initialize:
  size = maximum size of the queue
  queue[size] = an empty array of size 'size'
  front = -1   // points to the front of the queue
  rear = -1    // points to the rear of the queue
```

```
Function isEmpty():
    If front == -1:
        return True
    Else:
        return False

Function isFull():
    If (rear + 1) % size == front:
        return True
    Else:
        return False

Function enqueue(element):
    If isFull() is True:
        Print "Queue is full"
    Else:
        If front == -1:      // Queue is empty, initialize front and rear
            front = 0
        rear = (rear + 1) % size
        queue[rear] = element
        Print "Enqueued:", element

Function dequeue():
    If isEmpty() is True:
        Print "Queue is empty"
    Else:
        dequeued_element = queue[front]
        If front == rear:     // Only one element was present
            front = -1        // Reset queue to empty state
            rear = -1
        Else:
            front = (front + 1) % size
        Print "Dequeued:", dequeued_element

Function peek():
    If isEmpty() is True:
        Print "Queue is empty"
    Else:
        Print "Front element:", queue[front]

Function displayQueue():
    If isEmpty() is True:
        Print "Queue is empty"
    Else:
        index = front
        While index != rear:
            Print queue[index]
            index = (index + 1) % size
        Print queue[rear]  // Print the last element
```

## Advantages of Circular Queue:

1. **Efficient Use of Space**: It avoids wasted space compared to a simple queue where the front pointer moves ahead but never reuses the space behind it.
2. **Effective for Fixed-size Buffers**: Ideal for situations where the queue has a fixed size, such as memory buffers or task scheduling.