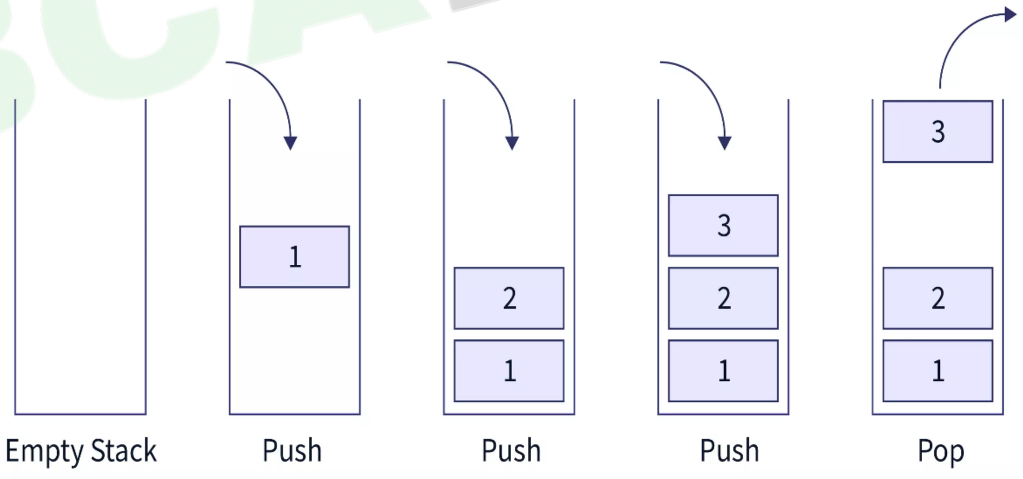
STACKS AND QUEUES

Stacks and queues are fundamental data structures in computer science, widely used for various applications in programming and algorithm design. Understanding their representation, operations, and specific use cases is crucial for efficient data management and processing.

1. Overview of Stacks

A **stack** is a linear data structure that operates on a Last In, First Out (LIFO) basis. This means that the last element added to the stack is the first one to be removed.



Key Operations of Stacks

- **Push**: Adds an element to the top of the stack.

- **Pop**: Removes the element from the top of the stack.
- **Peek**: Retrieves the top element without removing it.
- **IsEmpty**: Checks whether the stack is empty.

2. Representation of Stacks

Stacks can be represented in two primary ways: using **arrays** and **linked lists**.

a) Array Representation

In the array representation, a stack is implemented using a fixed-size array. An integer variable, often called top, is used to keep track of the index of the last element added.

- **Advantages**:
    - Simple to implement.
    - Fast access to elements since array indexing is constant time.
- **Disadvantages**:
    - Fixed size leads to potential overflow if more elements are pushed than the allocated size.
    - Wasted space can occur if many pops are performed.

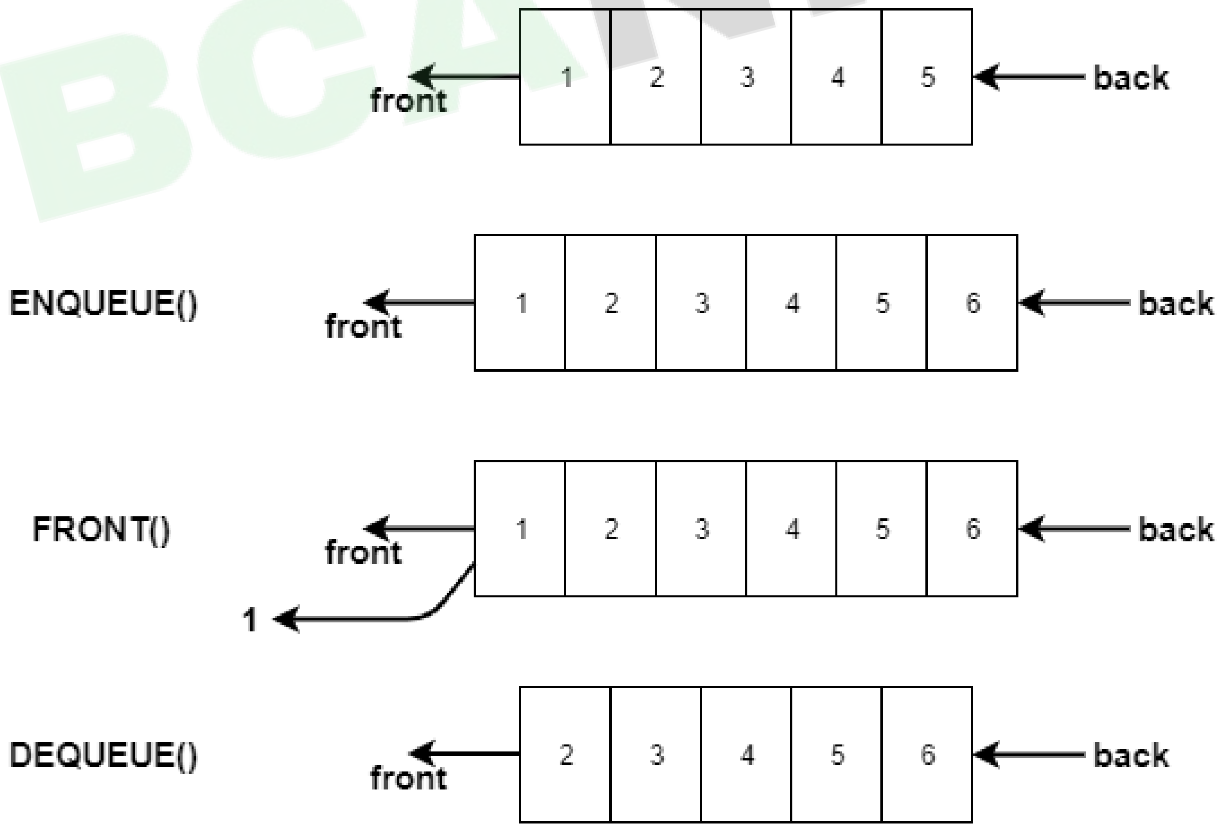
b) Linked List Representation

In the linked list representation, each element is stored in a node that contains data and a pointer to the next node. The top of the stack points to the head of the linked list.

- **Advantages**:
  - Dynamic size allows for more flexibility; the stack can grow or shrink as needed.
  - No overflow, as long as there is available memory.
- **Disadvantages**:
  - More complex to implement.
  - Extra memory is required for storing pointers.

3. Overview of Queues

A **queue** is a linear data structure that follows the First In, First Out (FIFO) principle. The first element added is the first one to be removed.



Key Operations of Queues

- **Enqueue**: Adds an element to the end of the queue.
- **Dequeue**: Removes the element from the front of the queue.
- **Front**: Retrieves the front element without removing it.
- **IsEmpty**: Checks whether the queue is empty.

4. Representation of Queues

Queues can also be represented using arrays and linked lists.

a) Array Representation

In an array-based queue, elements are stored in a fixed-size array, with two pointers (front and rear) tracking the start and end of the queue.

- **Advantages**:
  - Simple to implement and fast access to elements.
- **Disadvantages**:
  - Fixed size leads to overflow when the queue is full.
  - Wasted space if many elements are dequeued.

b) Linked List Representation

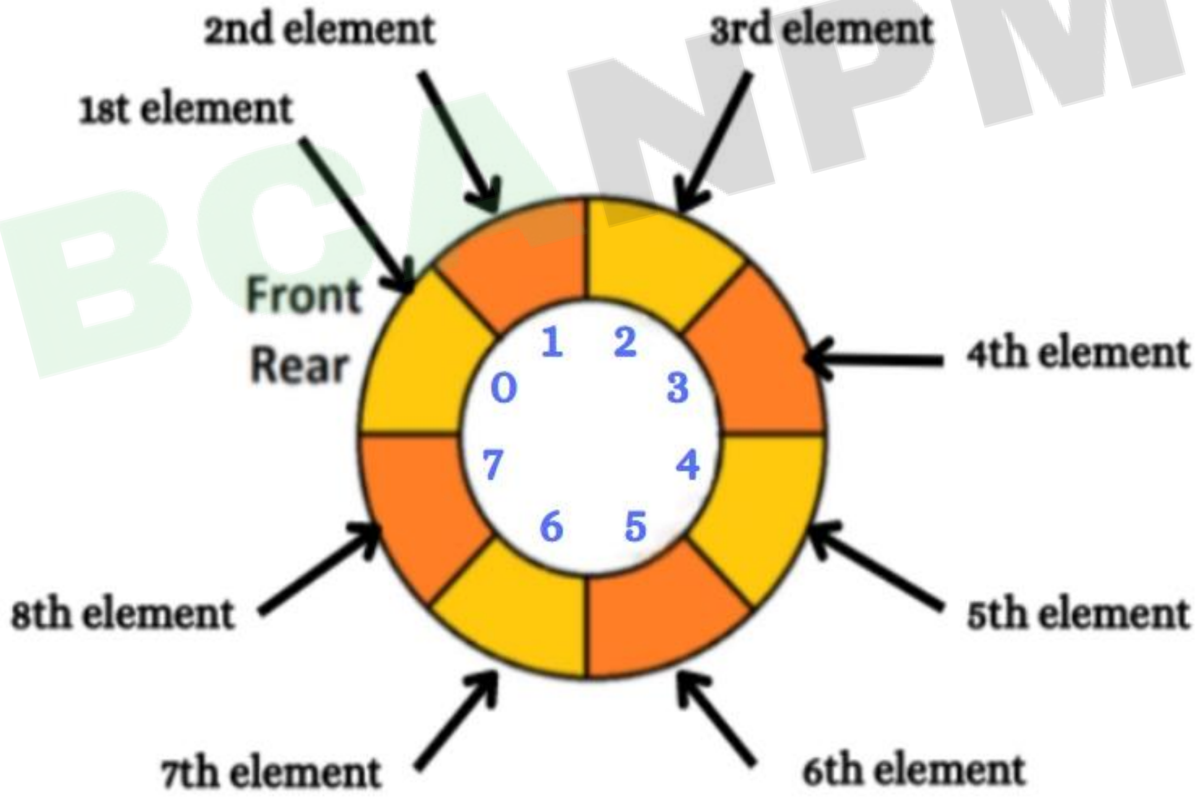
A queue implemented using a linked list consists of nodes where each node has data and a pointer to the next node.

- **Advantages**:
  - Dynamic size accommodates growing data needs.

    - No overflow until memory is exhausted.
- **Disadvantages**:
    - More complex to manage.
    - Extra memory needed for pointers.

5. Circular Queues

A **circular queue** improves upon the standard queue by connecting the last position back to the first position. This effectively uses available space and prevents the wastage seen in linear queues.



Key Features:

- When adding an element, if the rear pointer reaches the end of the array, it wraps around to the beginning.
- This circular structure allows for continuous use of the queue without needing to shift elements.

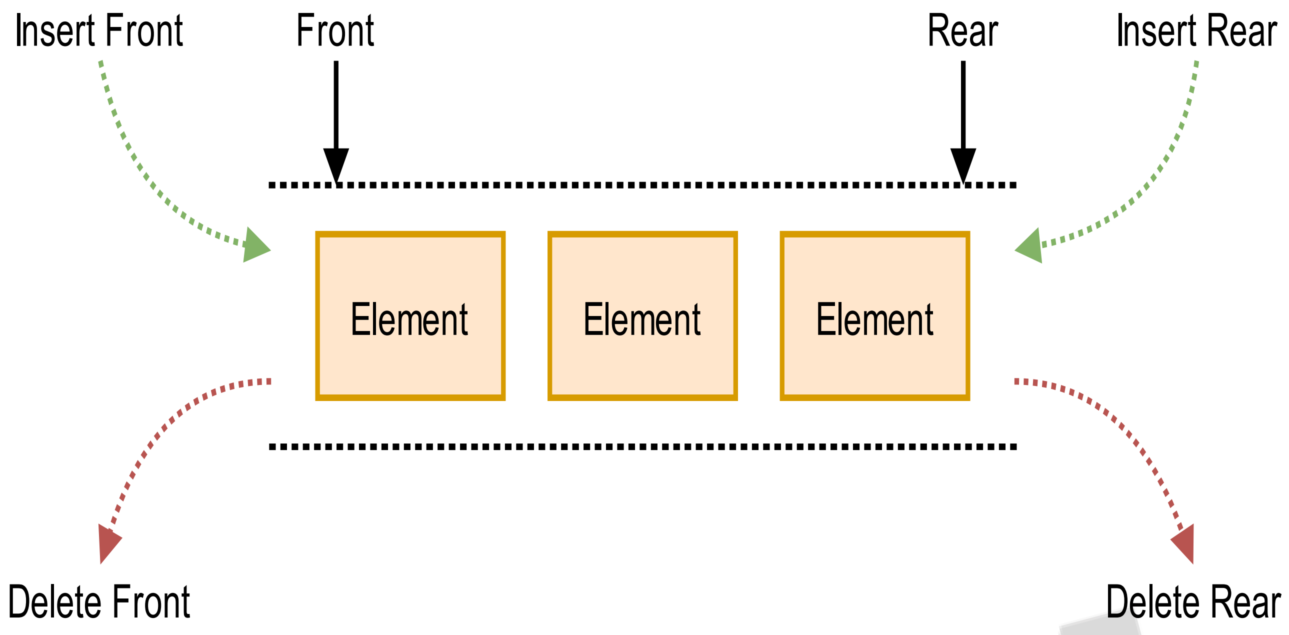
6. Priority Queues

A **priority queue** is a specialized queue where each element is associated with a priority. Elements are dequeued based on their priority rather than their order in the queue.

- **Implementation**: Typically implemented using heaps (binary heaps), which allow for efficient insertion and removal based on priority.
- **Applications**: Commonly used in scheduling algorithms, such as in operating systems, for task management and resource allocation.

7. Double-Ended Queue (Deque)

A **deque** (double-ended queue) allows insertion and deletion of elements from both ends. This flexibility makes it a powerful data structure.

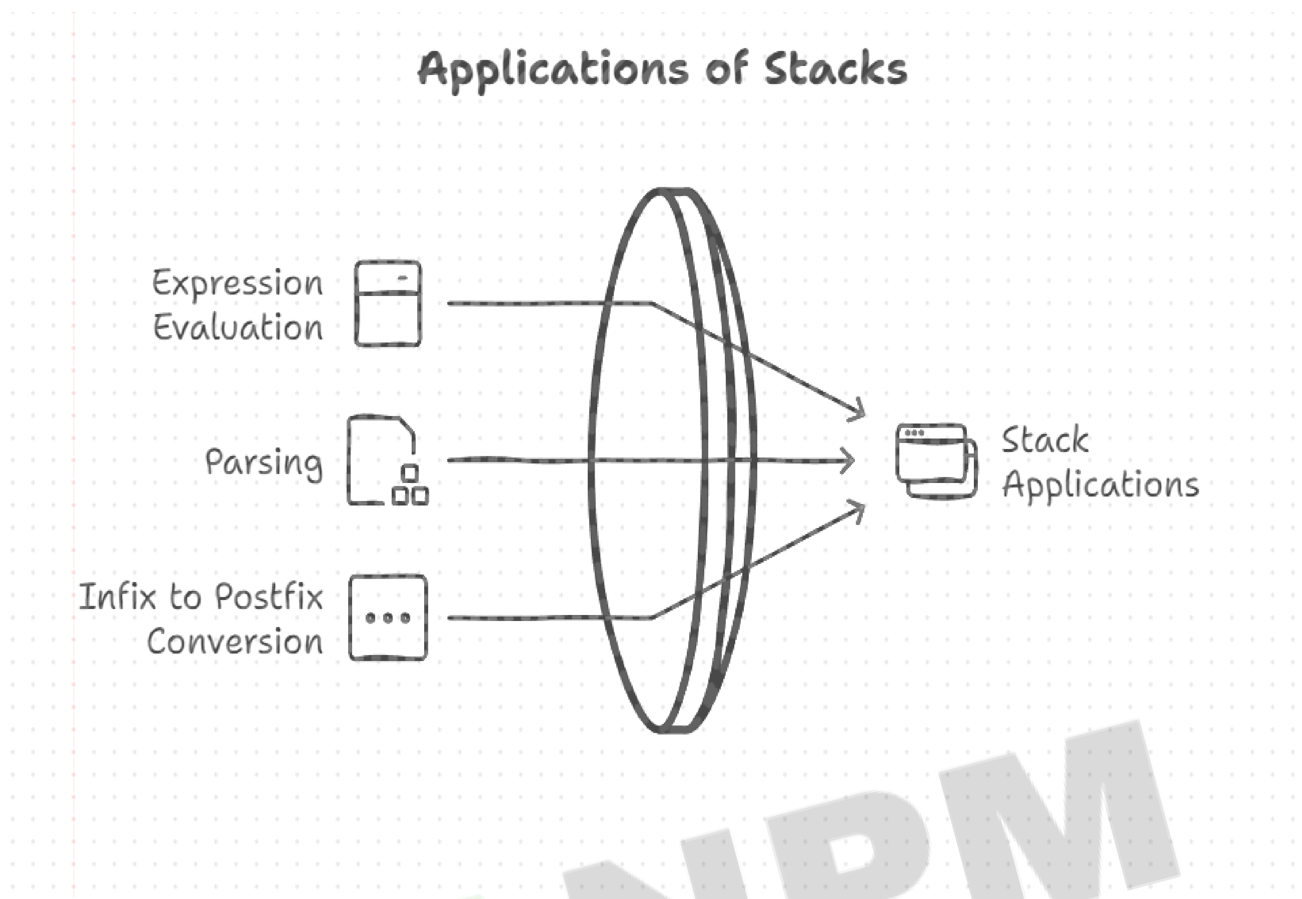


Key Operations:

- **AddFront**: Adds an element to the front.
- **AddRear**: Adds an element to the back.
- **RemoveFront**: Removes an element from the front.
- **RemoveRear**: Removes an element from the back.

8. Applications of Stacks

Stacks have several applications, particularly in expression evaluation and parsing.



a) Conversion from Infix to Postfix Expression

Infix notation (e.g., A + B) is commonly used in arithmetic expressions. However, it can be ambiguous due to operator precedence and parentheses. **Postfix notation** (also known as Reverse Polish Notation) removes this ambiguity and is easier for computers to evaluate.

- **Process**:
    - Use a stack to hold operators while scanning the infix expression.
    - Operands are output immediately, while operators are pushed onto the stack based on their precedence.

b) Evaluation of Postfix Expression

Postfix expressions can be evaluated efficiently using a stack:

- **Process**:
  - Scan the postfix expression from left to right.
  - Push operands onto the stack.
  - When an operator is encountered, pop the required number of operands, perform the operation, and push the result back onto the stack.

9. Conclusion

Stacks and queues are essential data structures that underpin many algorithms and applications in computer science. Their various representations (array and linked list) allow for flexibility in implementation, while advanced forms like circular queues, priority queues, and deques expand their utility.

Mastering these data structures and their applications—such as expression conversion and evaluation—enhances a programmer's ability to solve complex problems efficiently. As you continue your studies in computer science, understanding and implementing stacks and queues will provide a solid foundation for more advanced topics and data structures.

Ultimately, the principles behind these structures—LIFO for stacks and FIFO for queues—are critical for effective data management in software development, algorithm design, and systems programming.