DATA STRUCTURES

Module II
Stacks and Queues

Contents

Module II: Stacks and Queues

This module dives into two fundamental linear data structures: **Stacks** and **Queues**. We'll explore their concepts, operations, applications, and different implementation methods.

1. Stacks: Concepts & Operations

- **What is a Stack?**
- A stack is a linear data structure that follows the **LIFO (Last In, First Out)** principle.
    - This means the element inserted most recently is the first one to be removed.
    - Conversely, the element in the stack for the longest time is the last one to be accessed or removed.
- **Real-life analogy**: Think of a **stack of plates**. You add new plates to the top, and when you need a plate, you take the one from the very top (the last one added).
- All insertions (push operations) and deletions (pop operations) always occur at one end, traditionally called the "**top**" of the stack. This single-point access is a defining characteristic.
- The LIFO nature makes stacks suitable for tasks where the processing order is the reverse of the arrival order.
- **Stack ADT and Use Cases**
- **Abstract Data Type (ADT) for Stacks**

An ADT defines the logical behavior and operations of a data structure, independent of its implementation. For a stack, the key operations are:

- **push()**: Adds an element to the top of the stack.
- **pop()**: Removes and returns the most recently added element (the top element).
- **peek()** (or top()): Views the top element without removing it.
- **isEmpty()**: Checks if the stack contains any elements (returns True if empty, False otherwise).
- **isFull()**: (Primarily for fixed-size implementations) Checks if the stack has reached its capacity.

- **Key Stack Applications**

Stacks are vital in many areas of computer science:

- **Compilers and Interpreters**: Used for **syntax parsing**, especially for checking balanced parentheses, brackets, and braces. Compilers push opening delimiters and pop them when corresponding closing delimiters are found.
- **Function Calls (Call Stack)**: Manages the execution flow of programs. When a function is called, its information (local variables, parameters, return address) is **pushed** onto a **stack frame**. When the function completes, its stack frame is **popped**, allowing the program to return.
- **"Undo/Redo" Features**: In applications, actions are pushed onto an "undo stack." "Undo" pops from this stack, and often pushes the action onto a "redo stack."
- **Web Browser History**: The "back" button often uses a stack to navigate previously visited pages.
- **Graph Traversal Algorithms**: Algorithms like Depth-First Search (DFS) use stacks.

A. Stack Operations – Overview

Here's a detailed look at the five primary stack operations:

1. **push(element)**
    - **Purpose**: Adds a new element to the **top** of the stack.
    - **Mechanism**: The "top" pointer is typically incremented, and the new element is placed at this new position.
    - **Overflow**: If the stack has a fixed capacity and is already full, a "Stack Overflow" condition occurs.
    - **Time Complexity**: O(1) (constant time), as it only involves updating a pointer and assigning a value.
2. **pop()**
    - **Purpose**: Removes and returns the element located at the very **top** of the stack.
    - **Mechanism**: After removing the element, the "top" pointer is decremented.
    - **Underflow**: If an attempt is made to pop from an empty stack, a "Stack Underflow" condition occurs (often returns None or an error).
    - **Time Complexity**: O(1) (constant time).
3. **peek() / top()**
    - **Purpose**: Allows you to inspect (view) the element currently at the top of the stack **without removing it**.
    - **Mechanism**: A non-destructive read operation.

- **Empty Stack**: If the stack is empty, calling peek() typically results in an error or None.
- **Time Complexity**: O(1).

4. **isEmpty()**
    - **Purpose**: A boolean operation that checks if the stack currently contains any elements.
    - **Returns**: True if empty, False otherwise.
    - **Usage**: Crucial before performing pop() or peek() to prevent underflow errors.
    - **Time Complexity**: O(1).
5. **isFull()**
    - **Purpose**: A boolean operation relevant for fixed-capacity stack implementations (e.g., array-based).
    - **Returns**: True if the stack has reached its maximum capacity, False otherwise.
    - **Usage**: Important before performing a push() operation to prevent overflow errors.
    - **Note**: For dynamic implementations (like linked lists), this operation might not be explicitly needed.
    - **Time Complexity**: O(1).

B. Stack Implementation (Array)

- **Concept**: A stack can be implemented using a **static array** where the maximum size is fixed.
- The array stores elements, and a variable (e.g., top or peak) keeps track of the index of the topmost element.
- Initially, top is often set to -1 for an empty stack.
- **push**: Increment top, then place the new element at array[top]. Check for overflow (top reaches capacity - 1).
- **pop**: Retrieve array[top], then decrement top. Check for underflow (top becomes -1).
- **Advantages**:
    - Simplicity.
    - Efficient random access (though not directly used for stack operations).
    - Better cache performance due to contiguous memory allocation.
- **Disadvantages**:

- **Fixed size**: Can lead to wasted space or overflow errors if capacity is exceeded.

- Common in embedded systems or where stack size is predictable.

C.Stack Implementation (Linked List)

- **Concept**: A stack can be implemented using a **dynamic linked list**, offering flexibility.
- Each element is a **node** with data and a pointer to the next node.
- The "top" of the stack is maintained by a pointer to the **head of the linked list**.
- **push**: Create a new node. Its pointer points to the current head. Update the stack's top pointer to the new node. (Effectively adding to the beginning of the list). O(1).
- **pop**: Retrieve data from the node pointed to by top. Update top to point to the next node. Deallocate the old top node if necessary. O(1).
- **isEmpty**: Check if the top pointer is null (None).
- **Advantages**:
    - **Dynamic Size**: Can grow or shrink at runtime, limited only by available memory. Eliminates "Stack Overflow" issues (unless system memory runs out).
    - **Efficient Memory Utilization**: Memory is allocated only as needed.
    - **No Shifting**: Operations at the head of a linked list are always O(1), with no need to shift elements.
- **Minor Overhead**: Extra memory required for pointers in each node.

D.Expression Conversion: Infix → Postfix

- **Infix Notation**: The common mathematical way of writing expressions (e.g., A + B * C), where operators are between operands. Requires rules for operator **precedence** and **associativity**, and often **parentheses** to dictate order.
- **Postfix Notation (Reverse Polish Notation - RPN)**: Operators are placed **after** their operands (e.g., A B C * +).
    - Solves ambiguity as the order of operations is implicitly defined by operator position.
    - Eliminates the need for parentheses and complex precedence rules during evaluation.
- **Role of the Stack**: The stack is crucial for converting infix to postfix.
    - Operands are appended directly to the postfix output.
    - Operators and parentheses use the stack as temporary storage.

- **Precedence Rules**: When an operator is encountered, its precedence is compared with the operator at the stack's top. Higher or equal precedence operators from the stack are popped to the output.
- **Parentheses**: Opening parentheses are pushed. Closing parentheses trigger popping all operators until the matching opening parenthesis is found (which is then discarded).
- After scanning, any remaining operators on the stack are popped to the postfix expression.
- This systematic use of a stack ensures the correct order of operations, respecting precedence and parentheses.

E. Algorithm: Infix to Postfix Conversion

This algorithm systematically uses a stack to manage operator precedence and parentheses:

1. **Initialize**:
    - Create an empty operator_stack.
    - Create an empty postfix_expression list.
    - Define a **precedence** for operators (e.g., ^ highest, *// next, +/- lowest).
2. **Scan Infix Expression**: Iterate through the infix expression character by character from left to right.
3. **Handle Operands**:
    - If the character is an **operand** (letter or digit), append it directly to postfix_expression.
4. **Handle Opening Parenthesis (()**:
    - If the character is (, push it onto the operator_stack.
5. **Handle Closing Parenthesis ())**:
    - If the character is), perform the following:
        - Pop operators from the operator_stack and append them to postfix_expression until an opening parenthesis (is encountered on the stack.
        - Once (is found, pop it from the stack and discard it (do not append to postfix_expression).
        - If the stack becomes empty before finding (, it indicates an error (mismatched parentheses).
6. **Handle Operators (+, -, *, /, ^)**:
    - If the character is an **operator**:

- While the operator_stack is **not empty**, and the top element is **not** an opening parenthesis (, AND the precedence of the current operator is **less than or equal to** the precedence of the operator at the top of the operator_stack (handle right-associative operators like ^ carefully where precedence might be strictly less than), THEN:
    - Pop the operator from the operator_stack and append it to the postfix_expression.
- Finally, push the current operator onto the operator_stack.

7. **End of Expression**:
    - After scanning the entire infix expression:
        - Pop any remaining operators from the operator_stack and append them to the postfix_expression. (Any remaining (on the stack indicates an error).

This process ensures that operators are placed in the correct order, respecting their original precedence and grouping.

F. Postfix Evaluation using Stack

- **Concept**: Postfix notation simplifies expression evaluation using a stack because operators follow their operands. No parentheses or explicit precedence rules are needed.
- When an operator is encountered, its operands are readily available on the stack.

- **Algorithm for Postfix Evaluation**

Scan the postfix expression from left to right:

1. **If an operand (number) is encountered**:
    - Immediately **push** it onto the stack. The stack accumulates values.
2. **If an operator is encountered**:
    - **Pop** the top two operands from the stack.
        - The **first pop** gives the **right operand**.
        - The **second pop** gives the **left operand**.
    - Perform the arithmetic operation using these two operands.
    - **Push** the result of this operation back onto the stack. This effectively evaluates sub-expressions.
3. This continues until the entire postfix expression has been scanned.
4. At the end, the stack should contain **only one element**, which is the final result of the evaluated expression.

- **Example with Step-by-Step Evaluation: 2 3 * 5 +**

1. **Scan 2**: 2 is an operand. Push 2 onto the stack.

- Stack: [2]

2. **Scan 3**: 3 is an operand. Push 3 onto the stack.
    - Stack: [2, 3]
3. **Scan ***: * is an operator.
    - Pop 3 (operand2).
    - Pop 2 (operand1).
    - Calculate operand1 * operand2 = 2 * 3 = 6.
    - Push 6 (result) onto the stack.
    - Stack: [6]
4. **Scan 5**: 5 is an operand. Push 5 onto the stack.
    - Stack: [6, 5]
5. **Scan +**: + is an operator.
    - Pop 5 (operand2).
    - Pop 6 (operand1).
    - Calculate operand1 + operand2 = 6 + 5 = 11.
    - Push 11 (result) onto the stack.
    - Stack: [11]

Result: After processing the entire expression, the final result 11 remains as the only element on the stack, which is then popped.

G. Stack Applications Recap

Stacks are fundamental tools in various computing scenarios:

- **Syntax Parsing (Compilers and Interpreters)**: Used to check for balanced symbols (parentheses, braces, brackets) and to build syntax trees.
- **Recursion Implementation (Call Stack)**: The operating system uses an implicit "call stack" to manage function calls (both recursive and non-recursive). Function state is pushed onto the stack, and popped upon completion.
- **Undo/Redo Features**: Actions are pushed onto an "undo stack," allowing users to revert changes. A "redo stack" stores undone actions to reapply them.
- **Expression Evaluation and Conversion**: Central to converting infix expressions to postfix (or prefix) and subsequently evaluating them, crucial for processing mathematical or logical expressions.

- **Backtracking Algorithms**: Used in algorithms that explore multiple paths (e.g., solving mazes, N-Queens problem). States are popped from the stack to backtrack to previous decision points.

---

2. Queues: Concepts & Operations

A. What is a Queue?

- A queue is a linear data structure that operates on the **FIFO (First In, First Out)** principle.
    - This means the element that was added to the queue first will also be the first one to be removed.
- **Real-world analogy**: Think of a **ticket line** at a movie theater or a grocery store. New customers join the end of the line (enqueue), and the person at the front is served first (dequeue).
- **Two Primary Operations**:
    - **enqueue()**: Adds a new element to the "**rear**" (or "tail") of the queue.
    - **dequeue()**: Removes an element from the "**front**" (or "head") of the queue.
- This two-ended access point is crucial for maintaining the FIFO order.
- Queues are ideal for managing tasks or data that need to be processed sequentially in their order of arrival.

B. Queue ADT and Applications

- ## Abstract Data Type (ADT) for Queues

The key operations for a queue ADT include:

- **enqueue(element)**: Adds an element to the rear of the queue.
- **dequeue()**: Removes and returns the element from the front.
- **peek()** (or front()): Inspects the front element without removing it.
- **isEmpty()**: Checks if the queue is empty.
- isFull(): (For fixed-size implementations) Checks if it's at capacity.

The ADT focuses on the logical behavior, ensuring any implementation adheres to FIFO.

- ## Key Queue Applications

Queues are ubiquitous in computer science, especially for sequential processing:

- **Operating Systems (Scheduling)**:
  - **CPU scheduling**: Processes waiting to be executed are placed in a **ready queue**.
  - **Disk scheduling**: Manages requests for disk I/O.
  - **Printer queues**: Hold print jobs in submission order.
- **Buffers**: Serve as crucial buffers in systems where data is produced at one rate and consumed at another (e.g., streaming media, keyboard input).
- **Network Traffic Management**: Routers use queues to manage incoming and outgoing data packets, ensuring orderly forwarding.
- **Breadth-First Search (BFS)**: A graph traversal algorithm that systematically explores nodes layer by layer. BFS inherently uses a queue to keep track of nodes to visit next.

C. Queue Operations – Overview

Similar to stacks, queues are defined by core operation

Just like stacks, queues are defined by a set of core operations that allow for efficient management of data based on the FIFO principle. These operations enable elements to be added, removed, inspected, and the state of the queue to be checked.

1. **enqueue(element)**
   - **Purpose**: Adds a new element to the **rear** (or tail) of the queue.
   - **Mechanism**: The element is placed at the logical end of the queue.
   - **Overflow**: If the queue is implemented with a fixed-size array and is already full, an "overflow" condition may occur, preventing the addition of new elements.

- **Time Complexity**: Typically **O(1)** (constant time) for both array-based (assuming space) and linked-list based implementations, as it usually involves a simple update of the rear pointer and element assignment.

2. **dequeue()**
    - **Purpose**: Removes and returns the element that is currently at the **front** (or head) of the queue. This is always the element that has been in the queue the longest, strictly adhering to FIFO.
    - **Mechanism**: After removal, the "front" pointer is updated to point to the next element.
    - **Underflow**: If an attempt is made to dequeue from an empty queue, an "underflow" condition occurs, and usually None or an error is returned.
    - **Time Complexity**: Generally **O(1)** for both common implementations.
3. **peek() / front()**
    - **Purpose**: Allows you to inspect, or peek at, the element located at the **front** of the queue **without actually removing it**.
    - **Mechanism**: A non-destructive read operation that simply returns the value of the first element.
    - **Empty Queue**: If the queue is empty, calling peek() typically results in an error or None.
    - **Time Complexity**: **O(1)**.
4. **isEmpty()**
    - **Purpose**: A boolean operation that checks whether the queue currently contains any elements.
    - **Returns**: True if the queue is empty and False otherwise.
    - **Usage**: This check is crucial to prevent "underflow" errors when attempting to dequeue or peek.
    - **Time Complexity**: **O(1)**.
5. **isFull()**
    - **Purpose**: A boolean operation primarily used with queue implementations that have a predefined, fixed capacity (e.g., array-based queues).
    - **Returns**: True if the queue has reached its maximum capacity and cannot accept any more elements, and False otherwise.
    - **Usage**: This check is vital before performing an enqueue() operation to prevent "overflow."
    - **Note**: For dynamic implementations (like those using linked lists), this operation is often not relevant as the queue can expand as long as memory allows.

- **Time Complexity**: **O(1)**.

These five operations collectively define the essential interface of a queue ADT, providing a controlled and efficient mechanism for managing data in a first-in, first-out manner.

D.Linear Queue using Array

- **Concept**: A straightforward way to implement a queue is by using a **fixed-size array**. This is often called a "**linear queue**."
- It uses two pointers (or indices): front and rear.
    - The front pointer tracks the index of the first element (the one to be dequeued next).
    - The rear pointer tracks the index of the last element (where new elements are enqueued).
    - Initially, both front and rear can be set to -1 or 0, with specific logic for the first enqueue.
- **enqueue**: The rear pointer is incremented, and the element is placed at array[rear].
- **dequeue**: The element at array[front] is retrieved, and the front pointer is incremented.
- **Limitation: Space Wastage**:
    - As elements are dequeued, the front pointer continuously moves forward, leaving **unused, empty slots at the beginning of the array**.
    - Even if there's enough physical space, if the rear pointer reaches the end of the array, the queue is considered "full," preventing further enqueue operations, despite empty slots at the start.
    - This happens because the pointers don't "wrap around" to reuse space. Reusing space would require shifting elements (inefficient O(N)) or resizing (costly).
    - This limitation makes linear array-based queues inefficient for long-running processes with frequent operations, as they can prematurely run out of usable space.

E. Circular Queue – Concept

- The **Circular Queue** is a clever solution designed to overcome the space wastage limitation of linear array-based queues.
- **Concept**: It treats the underlying array as if its ends are connected, forming a **circle**. This allows the rear pointer to "wrap around" to the beginning of the array once it reaches the physical end, provided there's empty space.
- **Visual**: Imagine array indices 0, 1, ..., N-1 arranged in a circle.
    - When rear reaches N-1, the next enqueue position is 0 (if empty).
    - Similarly, if front reaches N-1, the next dequeue position is 0.

- **Modulo Operator (%)**: This wrapping behavior is achieved using the modulo operator. For example, (index + 1) % capacity calculates the next logical index, effectively cycling back to 0 after capacity - 1.

- **Benefits over Linear Queue**

1. **Efficient Space Utilization**: The most prominent advantage is that it effectively reuses the empty slots at the beginning of the array. The entire array capacity can be utilized, maximizing memory efficiency.
2. **No Shifting of Elements**: Both enqueue and dequeue operations involve simple pointer manipulations and element assignments, making them constant time operations (**O(1)**). This leads to much better performance.
3. **Continuous Operation**: A circular queue can operate continuously without needing recreation or costly data shifting, as long as its capacity is not exceeded.

F. Circular Queue Operations

The operations in a circular queue involve a critical modification to handle the wrapping-around behavior: the use of the **modulo operator (%)**.

- **enqueue(element)**:
    1. First, check if the queue is **full**. (Condition is more complex; see "Front and Rear Management" below).
    2. If not full, update rear: rear = (rear + 1) % capacity.
    3. Place the element at queue[rear].
    4. If this is the very first element, front must also be initialized to 0.
    5. Increment a size counter.
- **dequeue()**:
    1. First, check if the queue is **empty**. (Typically when size counter is 0).
    2. If not empty, retrieve the element at queue[front].
    3. Update front: front = (front + 1) % capacity.
    4. If the queue becomes empty after this dequeue (e.g., size becomes 0), both front and rear are often reset to -1.
    5. Decrement the size counter.
- **peek() / front()**:
    1. Returns the element at queue[front] without modifying front or rear.
    2. Requires checking for an empty queue first.

- **Front and Rear Management in Circular Queues**

The key challenge is accurately distinguishing between a **full** and an **empty** queue, as the condition front == rear can mean both (e.g., after the queue wraps around). Common strategies to resolve this include:

1. **Using a size (or count) variable**:
    - Track the number of elements in the queue.
    - size == 0 for empty, size == capacity for full. This is robust.
2. **Leaving one empty slot**:
    - The queue is considered full when (rear + 1) % capacity == front. This means the actual usable capacity is capacity - 1.

Modulo arithmetic ensures the queue pointers correctly cycle through the array, effectively utilizing all available memory and providing constant-time (**O(1)**) operations for both enqueue and dequeue.

G. Code Example: Circular Queue

This section would typically feature a practical code example, often in Python or C, demonstrating the implementation of a circular queue.

A CircularQueue class would likely include:

- An __init__ method to set up the fixed-size array (e.g., a Python list initialized with Nones), capacity, front, rear, and size.
- is_empty() and is_full() methods that check the size variable to accurately determine the queue's state.
- An enqueue(item) method:
    - Checks is_full().
    - If the queue is_empty(), self.front is set to 0.
    - self.rear is updated using (self.rear + 1) % self.capacity.
    - The item is placed at self.queue[self.rear].
    - self.size is incremented.
- A dequeue() method:
    - Checks is_empty().
    - The dequeued_item is retrieved from self.queue[self.front].
    - Optionally, self.queue[self.front] can be set to None to clear the slot.
    - self.size is decremented.
    - If self.size becomes 0 after dequeue, both self.front and self.rear are reset to -1.

    - Otherwise, self.front is updated using (self.front + 1) % self.capacity.
- A peek() method to return self.queue[self.front].

A demonstration with enqueue and dequeue operations, especially when rear wraps around, would visually confirm the circular behavior and efficient space reuse. For instance, showing how an element is enqueued into a slot previously occupied and dequeued, proving that the fixed-size array is used optimally.

H. Queue using Linked List

- **Concept**: Implementing a queue using a **linked list** provides a highly flexible and dynamic solution, overcoming the fixed-size constraint of array-based implementations.
- Each element is a **node** (containing data and a pointer to the next node).
- To maintain FIFO efficiently, the linked list implementation typically keeps two pointers:
    - front (or head), pointing to the first node in the queue.
    - rear (or tail), pointing to the last node.
- **enqueue(element)**:
    - A new node is created for the element.
    - This new node is added to the **rear** of the queue.
    - If the queue is empty, the new node becomes both front and rear.
    - Otherwise, the next pointer of the current rear node is updated to point to the new node, and then the rear pointer itself is updated to point to this new node.
    - **Time Complexity**: **O(1)**.
- **dequeue()**:
    - The element is removed from the **front** of the queue.
    - The data from the front node is retrieved.
    - The front pointer is moved to the next node in the list.
    - If, after dequeue, the queue becomes empty (front becomes null), the rear pointer should also be set to null.
    - **Time Complexity**: **O(1)**.
- **Advantages of Linked List-based Queue**

1. **Dynamic Size**: The queue can grow or shrink infinitely, limited only by the available system memory. There's no need to pre-specify a maximum capacity.
2. **No Overflow (Memory-Permitting)**: Unlike array-based queues, there's no fixed-capacity "overflow" issue. The queue will only run out of space if the entire system's memory is exhausted.

3. **No Space Wastage or Shifting**: Memory is allocated for nodes only as needed, leading to efficient memory usage. Elements never need to be shifted, which contributes to the consistent **O(1)** performance of enqueue and dequeue.

The main trade-off is the slight overhead of storing pointers within each node, but this is a small price for increased flexibility and consistent performance.

I. Dequeue (Double-Ended Queue)

- **Concept**: A **Deque** (pronounced "deck," short for **Double-Ended Queue**) is a more versatile form of a queue.
- It allows for **insertions and deletions from both ends** – the front and the rear.
- It combines the functionalities of both a standard queue (FIFO) and a stack (LIFO), offering greater flexibility.

• Deque Operations

A general deque supports:

- **addFront(element)**: Inserts an element at the front.
- **addRear(element)**: Inserts an element at the rear.
- **removeFront()**: Removes and returns the element from the front.
- **removeRear()**: Removes and returns the element from the rear.
- **peekFront()**: Returns the front element without removing it.
- **peekRear()**: Returns the rear element without removing it.
- **isEmpty()**: Checks if the deque is empty.

Dequeues can be implemented using either a **circular array** or a **doubly linked list**. A doubly linked list is often preferred due to its inherent flexibility in adding/removing nodes from both ends efficiently (**O(1)** time complexity for all operations).

• Specialized Deque Types

Based on restricted operations, deques can be categorized:

1. **Input-Restricted Deque**:
    - Insertions are allowed only at **one end** (either front or rear).
    - Deletions can be performed from **both ends**.
    - Example: A system where new tasks enter from one point, but administrators can expedite/cancel tasks from either end.
2. **Output-Restricted Deque**:
    - Deletions are allowed only from **one end**.
    - Insertions can be made from **both ends**.

- Example: A buffer where items can be added from two sources (e.g., high/low priority streams), but consumption always happens from a single output point.

J. Applications of Queue and Dequeue

Queues and Dequeues are incredibly versatile data structures with a wide array of applications across various domains.

- **Applications of Queues**
- **Operating System Scheduling**: Foundational for managing processes and resources.
    - **CPU scheduling**: Processes wait in a **ready queue** (e.g., FCFS, Round Robin).
    - **I/O buffering**: Keyboard input, printer queues.
    - **Network traffic management**: Packet queues in routers.
    - Ensures fairness and order in resource allocation.
- **Breadth-First Search (BFS)**: A fundamental graph traversal algorithm that explores a graph layer by layer. A queue stores discovered but unvisited nodes, ensuring all neighbors at the current depth are explored before moving deeper. Perfect for finding shortest paths in unweighted graphs.
- **Simulation**: Used to model real-world waiting lines (e.g., customer service, call centers, manufacturing assembly lines) to analyze and optimize flow.
- **Asynchronous Data Transfer**: Act as buffers in asynchronous communication, smoothing data flow between components operating at different speeds.
- **Applications of Dequeues**
- **Work Stealing Queues (Parallel Computing)**: In parallel programming, one processor can "steal" tasks from the back of another processor's deque when its own is empty, optimizing load balancing.
- **Cache Systems (e.g., LRU Cache)**: Excellent for implementing cache replacement policies like **Least Recently Used (LRU)**. When an item is accessed, it's moved to the front. If the cache is full, the item at the rear (least recently used) is removed.
- **Palindrome Checking**: A common application to check if a string reads the same forwards and backward. Characters are added to a deque, then removed from both the front and rear simultaneously and compared.
- **Web Browser History**: While often using stacks for "back," a combination of a deque can manage both forward and backward history more flexibly.

K. Summary & Key Takeaways

This module has covered the core distinctions and applications of stacks and queues. Understanding when to use each data structure is paramount for efficient algorithm design and problem-solving.

- ## Stack vs. Queue: The Fundamental Difference

The fundamental difference lies in their **access principles**:

- **Stack**: Operates on a **LIFO (Last In, First Out)** principle.
    - Imagine a physical stack of items where the last one placed on top is the first one removed.
    - All operations (push and pop) occur at one end, the "**top**."
    - Suitable for situations where items need to be processed in the **reverse order of their arrival**.
- **Queue**: Operates on a **FIFO (First In, First Out)** principle.
    - Think of a traditional waiting line where the first person to join the line is the first person to be served.
    - Operations occur at two ends: enqueue (add) at the "**rear**" and dequeue (remove) at the "**front**."
    - Ideal for situations where items must be processed in the **exact order they arrived**.

- ## Use Cases of Each Data Structure

Stack Use Cases:

- **Function Call Management**: The call stack manages function invocations and returns, crucial for program execution and recursion.
- **Expression Evaluation and Conversion**: Essential for parsing and evaluating arithmetic expressions (e.g., infix to postfix conversion, postfix evaluation).
- **Undo/Redo Functionality**: Software applications use stacks to track user actions, allowing them to be reversed.
- **Backtracking Algorithms**: Algorithms like maze solvers or game AI use stacks to keep track of previous states and backtrack when a dead end is reached.
- **Syntax Checking**: Compilers use stacks to verify balanced parentheses, braces, and brackets in code.

Queue Use Cases:

- **Task Scheduling**: Operating systems use queues to manage processes, I/O requests, and print jobs, ensuring fair execution order.
- **Buffering**: Used to temporarily hold data between components that operate at different speeds (e.g., keyboard input, network packet handling, streaming media).
- **Breadth-First Search (BFS)**: A primary algorithm for traversing graphs level by level, finding shortest paths in unweighted graphs.

- **Data Synchronization**: In multi-threaded environments, queues can synchronize data access between producer and consumer threads.

This recap highlights that while both are linear structures, their distinct access patterns dictate their suitability for different computational problems, making them indispensable tools in a programmer's arsenal.