3. ADT Queue

A Queue is a linear data structure that follows the First-In, First-Out (FIFO) principle. Imagine a line of people waiting for a service; the first person in line is the first to be served.

3.1. Types of Queues and Operations

3.1.1. Simple Queue

A basic queue where elements are added at the rear and removed from the front.

operation	description	algorithm	time complexity	space complexity
`enqueue(item)`	Adds an element to the `rear` of the queue.	1. If `rear` reaches `MAX_SIZE - 1`, report "Queue Full". 2. Increment `rear`. 3. Assign item to `queue[rear]`. 4. If `front` is -1 (first element), set `front = 0`.	O(1)	O(1)
`dequeue()`	Removes and returns the front element from the queue.	1. If `front` is -1 or `front > rear`, report "Queue Empty". 2. 2. Store `queue[front]` in a temporary variable. 3. Increment `front`. 4. If `front > rear`, reset `front = -1`, `rear = -1` (queue becomes empty). 5. Return the stored element.	O(1)	O(1)
`front()` / `peek()`	Returns the front element without removing it.	1. If queue is empty, report "Queue Empty". 2. Return `queue[front]`.	O(1)	O(1)

operation	description	algorithm	time complexity	space complexity
`isEmpty()`	Checks if the queue is empty.	1. Return true if top is -1, false otherwise.	O(1)	O(1)
`isFull()`	Checks if the queue is full (if implemented with a fixed-size array).	1. Return true if top is MAX_SIZE - 1, false otherwise.	O(1)	O(1)

3.1.2. Circular Queue

A circular queue addresses the limitation of a simple queue where, even if there are empty slots at the beginning of the array, `rear` might reach `MAX_SIZE - 1` and declare the queue full. In a circular queue, the last position is connected to the first.

Operation	Description	Algorithm (Key Difference)	Time Complexity	Space Complexity
`enqueue(item)`	Adds an element to the rear.	`rear = (rear + 1) % Q_MAX_SIZE;` Check: `(rear + 1) % Q_MAX_SIZE == front` (queue full)	O(1)	O(1)
`dequeue()`	Removes an element from the front.	`front = (front + 1) % Q_MAX_SIZE;` Check: `front == rear` (queue empty)	O(1)	O(1)
`front()` / `peek()`	Returns the front element.	Same as simple queue	O(1)	O(1)
`isEmpty()`	Checks if the queue is empty.	`front == -1` (initial state) or `front == rear` (in some implementations)	O(1)	O(1)
`isFull()`	Checks if the queue is full.	`(rear + 1) % Q_MAX_SIZE == front` (One slot left empty to distinguish from	O(1)	O(1)

Operation	Description	Algorithm (Key Difference)	Time Complexity	Space Complexity
		empty)		

3.1.3. Priority Queue

A priority queue is a special type of queue where each element has a *priority*. Elements with higher priority are served before elements with lower priority. If two elements have the same priority, they are served according to their order in the queue (FIFO).

Implementation: Priority queues are typically implemented using *heaps* (binary heaps are common), which provide efficient insertion and extraction of the maximum/minimum element. Other implementations include unsorted arrays/linked lists (slow enqueue/dequeue), or sorted arrays/linked lists (fast dequeue, slow enqueue).

Operation	Description	Algorithm (Using Heap)	Time Complexity	Space Complexity
`insert(item, priority)`	Adds an element with a given priority.	Insert at end, then heapify-up.	O(log N)	O(1)
`extractMin()` / `extractMax()`	Removes and returns the element with the highest/lowest priority.	Remove root, move last element to root, then heapify-down.	O(log N)	O(1)
`peekMin()` / `peekMax()`	Returns the highest/lowest priority element without removing it.	Return root.	O(1)	O(1)
`isEmpty()`	Checks if the priority queue is empty.	Check if heap size is 0.	O(1)	O(1)

C Code for Priority Queue Implementation (Min-Heap based)

For brevity, a full heap implementation is complex. Here's a conceptual outline and a simple array-based (less efficient for general use) approach to illustrate the idea. For production, a binary heap is recommended.

```c
#include <stdio.h>
#include <stdlib.h>

#define PQ_MAX_SIZE 100

// Structure to represent an element in the priority queue
typedef struct {
    int value;
    int priority; // Lower value means higher priority
} PQElement;

PQElement priorityQueue[PQ_MAX_SIZE];
int pqSize = 0; // Current number of elements in the priority queue

// Function to swap two elements in the array
void swap(PQElement *a, PQElement *b) {
    PQElement temp = *a;
    *a = *b;
    *b = temp;
}

// Helper function to maintain min-heap property (heapify-up)
void heapifyUp(int index) {
    int parent = (index - 1) / 2;
    while (index > 0 && priorityQueue[index].priority <
priorityQueue[parent].priority) {
        swap(&priorityQueue[index], &priorityQueue[parent]);
        index = parent;
        parent = (index - 1) / 2;
    }
}

// Helper function to maintain min-heap property (heapify-down)
void heapifyDown(int index) {
    int leftChild = 2 * index + 1;
    int rightChild = 2 * index + 2;
    int smallest = index;

    if (leftChild < pqSize && priorityQueue[leftChild].priority <
priorityQueue[smallest].priority) {
        smallest = leftChild;
    }
    if (rightChild < pqSize && priorityQueue[rightChild].priority <
priorityQueue[smallest].priority) {
        smallest = rightChild;
    }

    if (smallest != index) {
        swap(&priorityQueue[index], &priorityQueue[smallest]);
```

```
        heapifyDown(smallest);
    }
}

// Function to insert an element into the priority queue
void pqInsert(int value, int priority) {
    if (pqSize == PQ_MAX_SIZE) {
        printf("Priority Queue is Full! Cannot insert value %d with priority
%d\n", value, priority);
        return;
    }
    priorityQueue[pqSize].value = value;
    priorityQueue[pqSize].priority = priority;
    heapifyUp(pqSize); // Maintain heap property
    pqSize++;
    printf("Inserted value %d with priority %d.\n", value, priority);
}

// Function to extract the element with the highest priority (minimum
priority value)
PQElement pqExtractMin() {
    PQElement emptyElement = {-1, -1}; // Indicate error/empty
    if (pqSize == 0) {
        printf("Priority Queue is Empty! Cannot extract.\n");
        return emptyElement;
    }
    PQElement minElement = priorityQueue[0]; // Element with highest priority
(root)
    priorityQueue[0] = priorityQueue[pqSize - 1]; // Move last element to
root
    pqSize--;
    heapifyDown(0); // Maintain heap property
    return minElement;
}

// Function to peek at the element with the highest priority
PQElement pqPeekMin() {
    PQElement emptyElement = {-1, -1};
    if (pqSize == 0) {
        printf("Priority Queue is Empty!\n");
        return emptyElement;
    }
    return priorityQueue[0];
}

// Function to check if the priority queue is empty
int pqIsEmpty() {
    return (pqSize == 0);
}
```

```c
// Function to display the priority queue (not necessarily sorted visually)
void displayPriorityQueue() {
    if (pqIsEmpty()) {
        printf("Priority Queue is empty.\n");
        return;
    }
    printf("Priority Queue elements (value, priority): ");
    for (int i = 0; i < pqSize; i++) {
        printf("(%d, %d) ", priorityQueue[i].value,
priorityQueue[i].priority);
    }
    printf("\n");
}

int main() {
    pqInsert(10, 2);
    pqInsert(20, 1);
    pqInsert(30, 3);
    pqInsert(5, 1); // Same priority as 20

    displayPriorityQueue(); // Order might not be visually sorted due to heap
property

    PQElement extracted = pqExtractMin();
    if (extracted.value != -1) {
        printf("Extracted (value, priority): (%d, %d)\n", extracted.value,
extracted.priority);
    }
    displayPriorityQueue();

    extracted = pqExtractMin();
    if (extracted.value != -1) {
        printf("Extracted (value, priority): (%d, %d)\n", extracted.value,
extracted.priority);
    }
    displayPriorityQueue();

    printf("Peek min (value, priority): (%d, %d)\n", pqPeekMin().value,
pqPeekMin().priority);
    printf("Is PQ empty? %s\n", pqIsEmpty() ? "Yes" : "No");

    return 0;
}
```

4. Exercises

Here are some exercises to practice your understanding of Stacks and Queues.

Stacks

1. **Parentheses Checker:** Write a C program that uses a stack to check if an expression has balanced parentheses (e.g., `([]{})` is balanced, `([)]` is not).
    - Hint: Push opening brackets onto the stack. When a closing bracket is encountered, pop from the stack and check for a match.
2. **Reverse a String:** Implement a function that reverses a given string using a stack.
3. **Evaluate Prefix Expression:** Write a C function to evaluate a prefix expression. (Hint: Scan from right to left).
4. **Implement Stack using Linked List:** Re-implement the Stack ADT using a linked list instead of an array. Analyze its complexity.

Queues

1. **Hot Potato Game:** Simulate the "Hot Potato" game using a simple queue. In this game, a group of players stands in a circle, and a "hot potato" is passed around. When the music stops, the person holding the potato is out. The game continues until only one person remains.
2. **Implement Queue using Linked List:** Re-implement the Queue ADT (simple or circular) using a linked list. Analyze its complexity.
3. **Deque (Double-Ended Queue):** Research and implement a Deque, which allows insertions and deletions from both ends.
4. **BFS (Breadth-First Search):** Understand how a queue is used in the Breadth-First Search algorithm for traversing graphs/trees. (Requires knowledge of graphs/trees).