DATA STRUCTURE

Chapter 4: Queues

Study Notes for Engineering Students

4.1 Introduction

A queue is a linear data structure that stores a collection of elements in a strict order and permits access only at its two ends. Unlike a stack, which follows the Last-In-First-Out (LIFO) principle, a queue follows the First-In-First-Out (FIFO) principle. This means that the element inserted first into the queue is the first one to be removed, exactly like a line of people waiting at a ticket counter — the person who joins the line first is served first.

In a queue, insertion of new elements always takes place at one end, called the REAR (or TAIL), and deletion of elements always takes place at the other end, called the FRONT (or HEAD). Because insertion and deletion happen at opposite ends, a queue is also referred to as a FIFO list.

Queues are widely used in computer science whenever tasks, requests, or data items must be processed in the exact order in which they arrive. Typical real-world and system-level examples include:

- CPU scheduling and disk scheduling in operating systems.
- Handling of requests on a single shared resource, such as a printer or a server.
- Managing calls at a call centre, or customers at a ticket counter.
- Breadth-First Search (BFS) traversal of graphs and trees.
- Buffering of data streams, such as IO buffers, keyboard buffers, and pipes.

4.2 Queue as an Abstract Data Type

An Abstract Data Type (ADT) is a mathematical model of a data structure that specifies the type of data stored, the operations supported on that data, and the behaviour of each operation, without specifying how these operations are actually implemented. The queue ADT defines a FIFO collection along with a fixed set of permitted operations. As an ADT, a queue can be implemented using different underlying structures, such as arrays or linked lists, while the external behaviour visible to the user remains exactly the same.

The Queue ADT is formally defined by the following components:

Data

A finite, ordered collection of elements of the same type, along with two pointers or indices — FRONT, which marks the position of the element to be removed next, and REAR, which marks the position where the next element will be inserted.

Operations

- enqueue(item): Inserts an item at the rear of the queue.

- dequeue(): Removes and returns the item at the front of the queue.
- peek() / front(): Returns the item at the front without removing it.
- isEmpty(): Returns true if the queue contains no elements.
- isFull(): Returns true if the queue has reached its maximum capacity (relevant for array-based implementations).

Because the ADT hides implementation detail, a program that uses a queue only through enqueue, dequeue, peek, isEmpty and isFull will work correctly regardless of whether the queue is actually built using an array or a linked list underneath.

4.3 Representation of Queues

A queue can be represented in memory in two principal ways: using a linear array or using a linked list. Each representation has its own advantages and limitations.

(a) Array Representation

In this representation, a one-dimensional array QUEUE[0..N-1] of fixed size N is used to hold the elements, along with two integer variables FRONT and REAR that keep track of the position of the first and last elements respectively.

- Initially, when the queue is empty, FRONT = REAR = -1 (a common convention).
- On enqueue, REAR is incremented by 1 and the new item is stored at QUEUE[REAR].
- On dequeue, the item at QUEUE[FRONT] is removed and FRONT is incremented by 1.

A significant drawback of this simple linear array representation is that once REAR reaches the last index (N-1), no further insertion is possible even if elements have been deleted from the front and free space exists at the beginning of the array. This limitation is known as the problem of false overflow, and it is overcome by using a circular queue, discussed in Section 4.5.

(b) Linked List Representation

In this representation, each element of the queue is stored in a node that contains the data and a pointer to the next node. Two external pointers, FRONT and REAR, are maintained — FRONT points to the first node of the linked list (the front of the queue) and REAR points to the last node (the rear of the queue).

- On enqueue, a new node is created, attached after the node pointed to by REAR, and REAR is updated to point to this new node.
- On dequeue, the node pointed to by FRONT is removed and FRONT is updated to point to the next node in the list.

The linked list representation does not suffer from the fixed-size limitation of arrays, since memory is allocated dynamically for each new node. However, it requires extra memory for storing pointers and generally has slightly higher overhead per operation compared to arrays.

4.4 Operations on Queue: Searching, Insertion, Deletion

The three fundamental operations performed on a queue are searching, insertion and deletion. Each is described below along with the array-based algorithmic steps.

(a) Searching

Searching in a queue means checking whether a particular element is present in the queue. Since a queue does not permit direct/random access to elements in the middle (only FRONT and REAR are accessible in the pure ADT sense), searching conceptually requires traversing the queue from FRONT to REAR, examining each element in turn until a match is found or the queue is exhausted. Because this violates the restricted-access nature of a pure queue, in practice searching is usually performed only when the underlying array or linked list is inspected directly, and the operation runs in O(n) time, where n is the number of elements currently in the queue.

(b) Insertion (Enqueue)

Insertion adds a new element at the REAR of the queue. Using a linear array representation, the algorithm is:

```
Algorithm ENQUEUE(QUEUE, N, FRONT, REAR, ITEM)
Step 1: If REAR = N - 1, then
            Print "QUEUE OVERFLOW" and exit.
Step 2: If FRONT = -1, then
            Set FRONT = 0.
Step 3: Set REAR = REAR + 1.
Step 4: Set QUEUE[REAR] = ITEM.
Step 5: Exit.
```

The insertion operation always executes in constant time, O(1), since the new element is placed directly at the position indicated by REAR without shifting any other elements.

(c) Deletion (Dequeue)

Deletion removes the element present at the FRONT of the queue. Using a linear array representation, the algorithm is:

```
Algorithm DEQUEUE(QUEUE, FRONT, REAR, ITEM)
Step 1: If FRONT = -1 or FRONT > REAR, then
            Print "QUEUE UNDERFLOW" and exit.
Step 2: Set ITEM = QUEUE[FRONT].
Step 3: Set FRONT = FRONT + 1.
Step 4: If FRONT > REAR, then
            Set FRONT = REAR = -1.   (queue becomes empty)
Step 5: Exit.
```

Like insertion, deletion also runs in constant time, O(1). Both enqueue and dequeue must always check for the boundary conditions of overflow (queue full) and underflow (queue empty) before proceeding, to avoid invalid memory access or incorrect results.

4.5 Circular Queues

A circular queue is an improved, more memory-efficient version of the linear array-based queue, in which the array is treated as circular — that is, the position after the last index (N-1) wraps around to index 0. This design eliminates the false overflow problem of linear queues, because the free space created at the front of the array (after elements are dequeued) can be reused for new insertions.

In a circular queue, the positions of FRONT and REAR are updated using modulo arithmetic:

```
REAR = (REAR + 1) MOD N        // on insertion
FRONT = (FRONT + 1) MOD N      // on deletion
```

Detecting Full and Empty Conditions

A subtle difficulty in circular queues is that when FRONT = REAR, the queue could be either completely empty or completely full, and the two situations must be told apart. Two common solutions are used:

- Maintain a separate COUNT variable that tracks the current number of elements; the queue is empty when COUNT = 0 and full when COUNT = N.
- Deliberately leave one array slot always empty, so the queue is considered full when (REAR + 1) MOD N = FRONT; this sacrifices one storage slot but avoids the need for an extra counter variable.

Algorithm: Insertion in a Circular Queue

```
Step 1: If (REAR + 1) MOD N = FRONT, then
            Print "QUEUE OVERFLOW" and exit.
Step 2: If FRONT = -1, then set FRONT = 0.
Step 3: Set REAR = (REAR + 1) MOD N.
Step 4: Set QUEUE[REAR] = ITEM.
Step 5: Exit.
```

Circular queues form the conceptual basis for circular buffers used extensively in real-time and embedded systems, such as producer-consumer buffering, streaming media buffers, and keyboard input buffers.

4.6 Priority Queue

A priority queue is a special type of queue in which every element is associated with a priority value, and elements are removed from the queue based on their priority rather than strictly on the order in which they were inserted. The element with the highest priority is served first; if two elements share the same priority, they are typically served according to their order in the queue (FIFO among equal priorities).

Priority queues are generally classified into two types:

- Ascending Priority Queue: Elements are removed in increasing order of value, i.e., the smallest element has the highest priority and is deleted first.
- Descending Priority Queue: Elements are removed in decreasing order of value, i.e., the largest element has the highest priority and is deleted first.

Implementation Techniques

A priority queue can be implemented in several ways, each with different performance trade-offs:

- Unordered array/list: Insertion is O(1), but deletion requires scanning the entire structure to find the highest-priority element, giving O(n) deletion.
- Ordered array/list: Elements are kept sorted by priority at all times, giving O(1) deletion of the highest-priority element, but O(n) insertion because the correct sorted position must be located and later elements shifted.
- Heap (Binary Heap): The most efficient general-purpose implementation; both insertion and deletion of the highest-priority element take O(log n) time, since a heap maintains a partially ordered, balanced tree structure stored in an array.

Because of its favourable O(log n) performance for both operations, the binary heap is the standard implementation used for priority queues in most programming language libraries and textbooks.

4.7 Application of Queues

Owing to its FIFO nature, the queue data structure is applied extensively across operating systems, networking, and everyday computing systems. Some of the most important applications are:

- CPU Scheduling: Operating systems maintain a ready queue of processes waiting for CPU time; scheduling algorithms such as First-Come-First-Served (FCFS) and Round Robin are built directly on queue behaviour.
- Disk Scheduling: Requests to read or write data from a disk are queued and served in an order determined by the disk scheduling algorithm in use.
- Handling of Interrupts: In real-time and embedded systems, interrupts are placed in a queue in the order they occur and are handled in that same order, often combined with priority queues for urgent interrupts.
- Breadth-First Search (BFS): Queues are the core data structure used to explore graphs and trees level by level, since a queue naturally enforces exploring nodes in the order they were discovered.
- Spooling: Print spoolers, and similar spooling systems for shared devices, queue print jobs or tasks so a single device can process them one at a time in the order submitted.
- Buffering of Data Streams: I/O buffers, network routers (packet queues), and asynchronous data transfer between processes running at different speeds all use queues to temporarily hold data.
- Call Centre and Customer Service Systems: Incoming calls or service requests are placed in a queue and answered strictly in the order of arrival, ensuring fairness.
- Simulation of Real-World Waiting Lines: Queues are used to model and simulate systems such as traffic, bank counters, and ticket booking systems for analysis and optimisation.

Multiple Choice Questions (MCQs)

Answer the following 15 multiple choice questions. The correct answer key is provided at the end of this section.

1. A queue is based on which of the following principles?

(A) LIFO
(B) FIFO
(C) Random access
(D) LILO (opposite order)

2. In a queue, insertion of a new element is done at the:

(A) Front
(B) Middle
(C) Rear
(D) Any position

3. In a queue, deletion of an element is done at the:

(A) Rear
(B) Front
(C) Middle
(D) Top

4. Which of the following best describes a real-life example of a queue?

(A) A stack of plates
(B) A pile of books
(C) A line of people at a ticket counter
(D) A deck of cards

5. The condition FRONT = -1 (initial state) in a simple array-based queue represents:

(A) Queue overflow
(B) Queue is full
(C) Queue is empty
(D) Invalid queue

6. In a simple (linear) array queue, the problem where REAR reaches N-1 but free space exists at the front is called:

(A) Deadlock
(B) False overflow
(C) Stack overflow
(D) Underflow

7. Which data structure eliminates the false overflow problem of a linear queue?

(A) Linked list stack
(B) Circular queue
(C) Binary tree
(D) Doubly linked stack

8. In a circular queue of size N, the REAR pointer is updated using which formula on insertion?

(A) REAR = REAR + 1
(B) REAR = (REAR + 1) MOD N
(C) REAR = REAR - 1
(D) REAR = N MOD REAR

9. In a circular queue, the condition FRONT = REAR can indicate:

(A) Only that the queue is full
(B) Only that the queue is empty
(C) Either the queue is empty or full
(D) An invalid queue state

10. What is the time complexity of the enqueue operation in a simple array-based queue?

(A) O(n) (B) O(log n)

(C) O(1) (D) O(n^2)

11. In a priority queue, elements are removed based on:

(A) Order of insertion only (B) Their priority value

(C) Random selection (D) Memory address

12. Which implementation of a priority queue gives O(log n) time for both insertion and deletion?

(A) Unordered array (B) Ordered array

(C) Binary heap (D) Simple linked list

13. Which traversal algorithm on graphs/trees uses a queue as its core data structure?

(A) Depth-First Search (DFS) (B) Breadth-First Search (BFS)

(C) In-order traversal (D) Binary search

14. In the linked list representation of a queue, the REAR pointer points to:

(A) The first node (B) The last node

(C) The middle node (D) A null node always

15. Which of the following is NOT a typical application of a queue?

(A) CPU scheduling (B) Printer spooling

(C) Undo operation in a text editor (D) Handling of interrupts

Answer Key

1. B 2. C 3. B 4. C 5. C 6. B 7. B 8. B 9. C 10. C 11. B 12. C 13. B 14. B 15. C

Broad Questions

Answer the following 10 descriptive/broad questions in detail.

1. Define a queue. Explain the FIFO principle with a suitable real-life example, and distinguish a queue from a stack.

2. What is meant by 'Queue as an Abstract Data Type'? List and briefly explain the standard operations defined on the Queue ADT.

3. Describe the array representation of a queue. Write the algorithms for the enqueue and dequeue operations, and mention the overflow and underflow conditions.

4. Describe the linked list representation of a queue. What are the advantages of this representation over the array representation?

5. What is a circular queue? Explain why it is needed, and describe how it overcomes the false overflow problem of a linear queue.

6. Write the algorithm for insertion and deletion in a circular queue. Explain how the queue-full and queue-empty conditions are distinguished when FRONT equals REAR.

7. What is a priority queue? Differentiate between an ascending priority queue and a descending priority queue with examples.

8. Discuss the different ways of implementing a priority queue (unordered list, ordered list, and heap), comparing their time complexities for insertion and deletion.

9. Explain, with suitable examples, at least five real-world applications of the queue data structure.

10. Explain how a queue is used in Breadth-First Search (BFS) traversal of a graph. Illustrate with a short example or pseudocode.