Lecture 2: Queue Data Structure

Introduction

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle where elements are added at the rear and removed from the front.

Imagine a queue in real life - like a line of people waiting where the first person to join is the first one to be served.

Unlike a stack (which follows Last-In-First-Out), a queue preserves the order of arrival, making it ideal for scenarios where processing order matters.

Characteristics

- First-In-First-Out (FIFO) principle
- Insertions are made at the rear end only
- Deletions are made at the front end only
- Middle access is restricted
- Can be implemented using arrays or linked lists
- Circular implementation possible for better space utilization
- Front and rear pointers track queue boundaries

The FIFO property is what distinguishes queues from other data structures. It ensures fairness in scenarios like request processing, where requests should be handled in the order they were received.

Applications

- Printer queue management: Documents are printed in the order they were sent
- Keystroke data storage in keyboards: Key presses are processed in sequence
- Process scheduling in operating systems: Processes wait their turn for CPU time
- Pipeline data processing: Data moves through processing stages in order
- CPU task scheduling: Tasks are executed based on arrival sequence
- I/O Buffer management: Input/output operations are handled sequentially

These applications highlight why the FIFO property is crucial in many computing scenarios. For example, in a printer queue, it would be unfair if documents submitted later were printed before earlier submissions.

Basic Operations

Insert (Enqueue): Adding an element to the rear end of the queue. If rear reaches the end, check if queue is full.

This operation has O(1) time complexity since we're only manipulating the rear pointer and adding at a known position.

Remove (Dequeue): Removing an element from the front of the queue. Check if queue is empty before removal.

This also has O(1) time complexity as we only need to access the front element and adjust the front pointer.

Peek Front: Reading the front element without removing it. Only the front element is visible to the queue user.

This operation simply returns the value at the front position without modifying the queue structure.

Is Empty: Check if the queue has no elements.

This is typically implemented by checking if the count of elements is zero.

Is Full: Check if the queue has reached its maximum capacity.

In array implementations, this checks if the number of elements equals the maximum size.

Implementation Approach

Queue can be implemented using an array with restricted access to maintain FIFO order. Circular implementation provides better space utilization.

There are two primary approaches to implementing queues:

1. Using arrays (fixed size but faster access)
2. Using linked lists (dynamic size but requires more memory for pointers)

The array implementation is more common for its simplicity and performance, but suffers from size limitations.

Key Implementation Concepts

- Constructor creates queue with specified size
- Front variable tracks the index of front element
- Rear variable tracks the index of last element
- nItems variable tracks total elements in queue
- Circular implementation helps utilize empty spaces
- Array wrapping requires modulo arithmetic
- Proper handling of full and empty conditions

In a linear queue implementation, once elements are dequeued, the space at the beginning of the array becomes unusable. Circular queues solve this problem by "wrapping around" to reuse that space.

Code Examples

Basic Queue Class Structure

```
class QueueX {
    private int maxSize;
    private int[] queArray;
    private int front;
    private int rear;
    private int nItems;

    public QueueX(int s) {
        maxSize = s;
        queArray = new int[maxSize];
        front = 0;
        rear = -1;
        nItems = 0;
    }
}
```

In this class structure:

- `maxSize` defines the capacity of the queue
- `queArray` is the actual storage for elements
- `front` points to the position from where elements will be removed
- `rear` points to the position where the last element was inserted
- `nItems` tracks the current number of elements, which helps with isEmpty() and isFull() operations

Linear Queue Implementation

Insert (Enqueue) Operation

```
public void insert(int j) {
    if (rear == maxSize - 1)
        System.out.println("Queue is full");
    else {
        queArray[++rear] = j;
        nItems++;
    }
}
```

This implementation has a limitation: when `rear` reaches the end of the array, no more insertions are possible even if there's space at the beginning due to previous dequeue operations. This is where circular queues become necessary.

Remove (Dequeue) Operation

```
public int remove() {
    if (nItems == 0) {
        System.out.println("Queue is empty");
        return -99;
    } else {
        nItems--;
        return queArray[front++];
    }
}
```

As elements are removed, `front` moves forward, leaving empty spaces at the beginning of the array. In a linear queue, these spaces are wasted, which is inefficient.

Circular Queue Implementation

Circular queues solve the space utilization problem by treating the array as circular, allowing rear to wrap around to the beginning when it reaches the end.

Insert (Enqueue) Operation

```
public void insert(int j) {
    if (nItems == maxSize)
        System.out.println("Queue is full");
    else {
        if(rear == maxSize - 1)
            rear = -1;
        queArray[++rear] = j;
        nItems++;
    }
}
```

Notice how we reset `rear` to -1 when it reaches the end of the array. After incrementing to 0, this allows insertions to continue from the beginning of the array, effectively reusing the space.

Remove (Dequeue) Operation

```
public int remove() {
    if (nItems == 0) {
        System.out.println("Queue is empty");
        return -99;
    } else {
        int temp = queArray[front++];
        if (front == maxSize)
            front = 0;
        nItems--;
        return temp;
    }
}
```

Similarly, when `front` reaches the end of the array, we reset it to 0 to continue removals from the beginning. This creates the circular behavior that gives this implementation its name.

Peek Front Operation

```
public int peekFront() {
    if (nItems == 0) {
        System.out.println("Queue is empty");
        return -99;
    } else {
        return queArray[front];
    }
}
```

Peeking simply returns the value at the front position without changing any state. It's useful for examining the next element to be processed without actually removing it.

Exercises

Exercise 1: Basic Queue Operations

Description: Implement a queue with maximum size 10 and insert the values: 10, 25, 55, 65, 85. Then remove all items and display them.

Solution:

```
QueueX theQueue = new QueueX(10);
theQueue.insert(10);
theQueue.insert(25);
theQueue.insert(55);
theQueue.insert(65);
theQueue.insert(85);

while(!theQueue.isEmpty()) {
    System.out.print(theQueue.remove() + " ");
}
```

This exercise demonstrates the basic workflow of queue operations. The output will be: `10 25 55 65 85` – notice how elements come out in the same order they went in, demonstrating the FIFO principle.

Exercise 2: Queue Status Methods

Description: Implement isEmpty() and isFull() methods for the Queue class

Solution:

```
public boolean isEmpty() {
    return (nItems == 0);
}

public boolean isFull() {
    return (nItems == maxSize);
}
```

These helper methods improve code readability and encapsulate the internal state checks. Using `nItems` for both checks is more efficient than trying to compare front and rear positions directly.

Exercise 3: Circular Queue Implementation

Description: Convert a linear queue to a circular queue by modifying the insert and remove operations

Solution:

```
public void insert(int j) {
    if (nItems == maxSize)
        System.out.println("Queue is full");
    else {
        if(rear == maxSize - 1)
            rear = -1;
        queArray[++rear] = j;
        nItems++;
    }
}

public int remove() {
    if (nItems == 0) {
        System.out.println("Queue is empty");
        return -99;
    } else {
        int temp = queArray[front++];
        if (front == maxSize)
            front = 0;
        nItems--;
        return temp;
    }
}
```

The key to circular implementation is the wrapping logic. For both `front` and `rear`, when they reach the end of the array, they're reset to the beginning. This allows for continuous operation without wasting space.

Exercise 4: Queue Peek Implementation

Description: Implement peekFront() method that returns the front element without removing it

Solution:

```
public int peekFront() {
    if (nItems == 0) {
        System.out.println("Queue is empty");
        return -99;
    } else {
        return queArray[front];
    }
}
```

The peek operation is useful in scenarios where you need to examine the next element before deciding whether to process it. In complex applications, this allows for logic like "look at the next task, but only process it if certain conditions are met."

Exercise 5: Array Wrapping

Description: Draw the Queue frame after performing insert and remove operations in a circular queue

Solution:

```
// Example operations
QueueX queue = new QueueX(5);
queue.insert(1);  // [1, _, _, _, _] front=0, rear=0
queue.insert(2);  // [1, 2, _, _, _] front=0, rear=1
queue.remove();   // [_, 2, _, _, _] front=1, rear=1
queue.insert(3);  // [_, 2, 3, _, _] front=1, rear=2
queue.insert(4);  // [_, 2, 3, 4, _] front=1, rear=3
queue.insert(5);  // [_, 2, 3, 4, 5] front=1, rear=4
queue.remove();   // [_, _, 3, 4, 5] front=2, rear=4
queue.insert(6);  // [6, _, 3, 4, 5] front=2, rear=0
```

This exercise illustrates the circular behavior in action. Notice how after removing elements and reaching the end of the array with insertions (rear = 4), the next insert wraps around to position 0. This demonstrates how circular queues efficiently utilize the entire array space.