Queues A queue is a data structure that can hold many elements.

Think of a queue as people standing in line in a supermarket.

The first person to stand in line is also the first who can pay and leave the supermarket. This way of organizing elements is called FIFO: First In First Out.

The basic operations we can do on a queue are:

- Enqueue: Adds a new element to the queue.
- Dequeue: Removes and returns the first (front) element from the queue.
- Peek: Returns the first element in the queue.
- isEmpty: Checks if the queue is empty.
- Size: Finds the number of elements in the queue.

Queues can be implemented by using arrays or linked lists.

Queues can be used to implement job scheduling for an office printer, order processing for e-tickets, or to create algorithms for breadth-first search in graphs.

Queue Implementation using Arrays

Reasons to implement queues using arrays:

- Memory Efficient: Array elements do not hold the next element's address like linked list nodes do.
- Easier to implement and understand: Using arrays to implement queues require less code than using linked lists, and for this reason it is typically easier to understand as well.

Reasons for not using arrays to implement queues:

- **Fixed-size**: An array occupies a fixed part of the memory. This means that it could take up more memory than needed, or if the array fills up, it cannot hold more elements. And resizing an array can be costly.
- **Shifting cost**: Dequeue causes the first element in a queue to be removed, and the other elements must be shifted to take the removed elements' place. This is inefficient and can cause problems, especially if the queue is long.
- **Alternatives**: Some programming languages have built-in data structures optimized for queue operations that are better than using arrays**.**

This is how a queue can be implemented using a linked list

Queue Implementation using Linked Lists

```java
public void enqueue(int data){
    if(arr.length <= size){
        arr = java.util.Arrays.copyOf(arr,arr.length+arr.length/2);
    }
    arr[size++] = data;
}

public void dequeue(){
    arr = java.util.Arrays.copyOfRange(arr,1,arr.length);
    size--;
}
public int getSize(){
    return size;
}
public int peek(){
    return arr[0];
}
public boolean isEmpty(){
    return (size == 0);
}
```

Reasons for using linked lists to implement queues:

- Dynamic size: The queue can grow and shrink dynamically, unlike with arrays.
- No shifting: The front element of the queue can be removed (enqueue) without having to shift other elements in the memory.

Reasons for not using linked lists to implement queues:

- Extra memory: Each queue element must contain the address to the next element (the next linked list node).
- Readability: The code might be harder to read and write for some because it is longer and more complex.

This is how a queue can be implemented using a linked list.

```java
public void enqueue(int data){
    if(Head == null){
        Head = new Node(data);
        curr = Head;
        size++;
        return;
    }
    curr.next = new Node(data);
    curr = curr.next;
    size++;
}

public void dequeue(){
    Head = Head.next;
    size--;
}

public int peek(){
    return Head.data;
}

public int size(){
    return size;
}

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