DEQUEUE DATA STRUCTURE

EC/2022/053 - GALKOTUWA K.S.B.

BECS 21223

Table of Contents

DEQUEUE DATA STRUCTURE 0

1. DEFINITION **2**

2. TYPES OF DEQUEUE **3**

Input-Restricted Dequeue *3*

Output-Restricted Dequeue *3*

3. SPECIFICATION – DEQUEUE OPERATIONS **4**

4. CONTIGUOUS IMPLEMENTATION **6**

5. LINKED IMPLEMENTATION **9**

6. CONCLUSION **12**

REFERENCES **13**

1. Definition

A dequeue (also called a double-ended queue) is a linear data structure that allows insertion and deletion operations to be performed at both ends, front and rear, of the queue. Unlike a traditional queue that follows a First-In-First-Out (FIFO) principle, a dequeue offers greater flexibility by permitting operations from both ends. This versatility makes it useful in various applications where elements need to be added or removed from either end of the sequence.

The dequeue can be visualized as a linear array or a linked list, where elements can be added or removed from both ends. This bidirectional functionality distinguishes it from both stacks (Last-In-First-Out) and queues (First-In-First-Out), effectively combining the features of both data structures

Dequeues perform all their basic operations super-fast, regardless of how many items they contain.

- **Adding items**: The dequeue always knows exactly where its front and back are, so it can place new items instantly without searching.
- **Removing items**: Since it keeps track of both ends, it can immediately grab and remove items from either end without checking other items.
- **Viewing items**: Looking at the front or back item happens instantly because the dequeue maintains direct references to both ends.

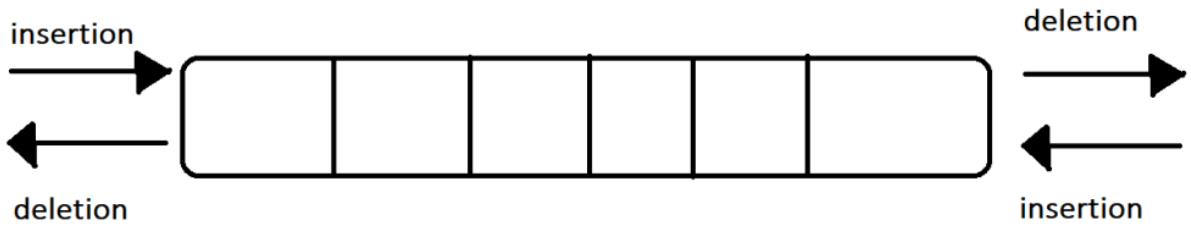

Figure 1 - DeQueue Structure

2. Types of Dequeue

There are two primary types of dequeues, each with specific operational constraints

Input-Restricted Dequeue

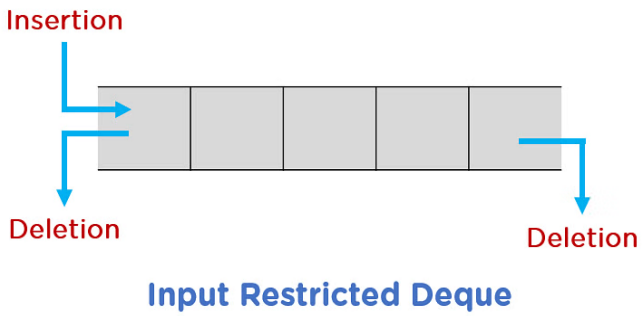

Figure 2 Input Restricted Dequeue

In an input-restricted dequeue, insertions can only be performed at one end (mostly the rear), and deletions can be performed from both ends (front and rear).

This restriction creates a hybrid between a standard queue and a more flexible data structure.

Output-Restricted Dequeue

In an output-restricted dequeue, deletions can only be performed at one end (mostly the front), and insertions can be performed at both ends (front and rear) .

This configuration provides flexibility in adding elements while maintaining orderly removal.

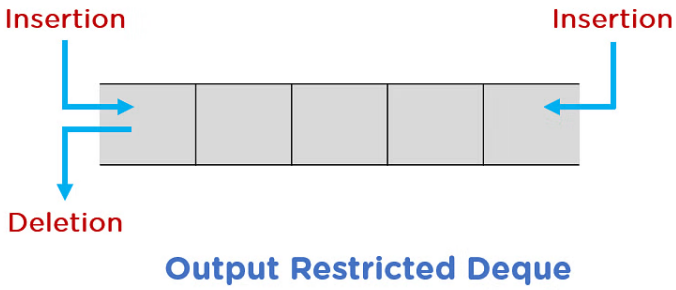

Figure 3 Output Restricted Dequeue

3. Specification – Dequeue Operations

A dequeue supports the following fundamental operations.

1. insertFront(element)

- **Inputs:** DQ (a Dequeue) and element (E)
- **Output:** DQ' (the updated Dequeue)
- **Preconditions:** DQ is created and not full. E is of appropriate type for an element of DQ
- **Postconditions: DQ' is DQ with E added as the first element (at the front)**.

2. insertRear(element)

- **Inputs:** DQ (a Dequeue) and element (E)
- **Output:** DQ' (the updated Dequeue)
- **Preconditions:** DQ is created and not full. E is of appropriate type for an element of DQ
- **Postconditions: DQ' is DQ with E added as the last element (at the rear)**.

3. deleteFront()

- **Inputs:** DQ (a Dequeue)
- **Output:** DQ' (the updated Dequeue) and E (the removed element)
- **Preconditions:** DQ is created and not empty
- **Postconditions: The element at the front of DQ has been removed and returned as E**.

4. deleteRear()

- **Inputs:** DQ (a Dequeue)
- **Output:** DQ' (the updated Dequeue) and E (the removed element)
- **Preconditions:** DQ is created and not empty
- **Postconditions: The element at the rear of DQ has been removed and returned as E**.

5. getFront()

- **Inputs:** DQ (a Dequeue)
- **Output:** E (the element at the front)
- **Preconditions:** DQ is created and not empty
- **Postconditions: The element at the front of DQ has been returned as E and DQ remains unchanged**.

6. getRear()

- **Inputs:** DQ (a Dequeue)
- **Output:** E (the element at the rear)
- **Preconditions:** DQ is created and not empty
- **Postconditions: The element at the rear of DQ has been returned as E and DQ remains unchanged**.

7. isEmpty()

- **Inputs:** DQ (a Dequeue)
- **Outputs:** boolean (true or false)
- **Preconditions:** DQ is created [Inferred based on Queue/List specifications, e.g., IsQueueEmpty preconditions].
- **Postconditions: Returns true if DQ is empty, false otherwise**

8. isFull()

- **Inputs:** DQ (a Dequeue)
- **Outputs:** boolean (true or false)
- **Preconditions:** DQ is created
- **Postconditions: Returns true if DQ is full, false otherwise**

9. size()

- **Inputs:** DQ (a Dequeue)
- **Output:** S (non-negative integer)
- **Preconditions:** DQ is created
- **Postconditions: S = the number of elements in DQ**

4. Contiguous Implementation

The contiguous implementation of a dequeue utilizes an array-based structure with fixed capacity.

In this implementation, the dequeue is represented as a circular array to efficiently manage space and avoid unnecessary shifting of elements.

```
public class ArrayDequeue<T> {
    private T[] array;
    private int front;
    private int rear;
    private int size;
    private int capacity;

    public ArrayDequeue(int capacity) {
        this.capacity = capacity;
        this.array = (T[]) new Object[capacity];
        this.front = 0;
        this.rear = capacity - 1;
        this.size = 0;
    }

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

    public boolean isFull() {
        return (size == capacity);
    }

    public int size() {
        return size;
    }

    public void insertFront(T element) {
        if (isFull()) {
            System.out.println("Dequeue is full");
        }

        // If first element is being inserted
        if (isEmpty()) {
            front = 0;
            rear = 0;
        } else {
            // Circular array concept - decrement front with circular wrap
            front = (front - 1 + capacity) % capacity;
        }

        array[front] = element;
        size++;
    }
```

```
public void insertRear(T element) {
    if (isFull()) {
        System.out.println("Dequeue is full");
    }

    // If first element is being inserted
    if (isEmpty()) {
        front = 0;
        rear = 0;
    } else {
        // Circular array concept - increment rear with circular wrap
        rear = (rear + 1) % capacity;
    }

    array[rear] = element;
    size++;
}

public T deleteFront() {
    if (isEmpty()) {
        System.out.println("Dequeue is empty");
    }

    T removed = array[front];
    array[front] = null; // Help with garbage collection

    // If this was the last element
    if (front == rear) {
        front = -1;
        rear = -1;
    } else {
        // Circular array concept - increment front with circular wrap
        front = (front + 1) % capacity;
    }

    size--;
    return removed;
}

public T deleteRear() {
    if (isEmpty()) {
        System.out.println("Dequeue is empty");
    }

    T removed = array[rear];
    array[rear] = null; // Help with garbage collection

    // If this was the last element
    if (front == rear) {
        front = -1;
        rear = -1;
    } else {
        // Circular array concept - decrement rear with circular wrap
        rear = (rear - 1 + capacity) % capacity;
    }

    size--;
    return removed;
}
```

```
    public T getFront() {
        if (isEmpty()) {
            System.out.println("Dequeue is empty");
        }
        return array[front];
    }

    public T getRear() {
        if (isEmpty()) {
            System.out.println("Dequeue is empty");
        }
        return array[rear];
    }
}
```

The circular array implementation prevents wasted space and unnecessary shifting operations

One challenge with the contiguous implementation is handling the wrap-around at the ends of the array.

5. Linked Implementation

The linked implementation of a dequeue uses a doubly linked list, where each node contains a reference to both the next and previous nodes.

This allows for efficient insertions and deletions at both ends of the dequeue without the size limitations of the array-based implementation.

```
public class LinkedDequeue<T> {
    private class Node {
        T data;
        Node prev;
        Node next;

        Node(T data) {
            this.data = data;
            this.prev = null;
            this.next = null;
        }
    }

    private Node front;
    private Node rear;
    private int size;

    public LinkedDequeue() {
        this.front = null;
        this.rear = null;
        this.size = 0;
    }

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

    public int size() {
        return size;
    }

    public void insertFront(T element) {
        Node newNode = new Node(element);

        // If dequeue is empty
        if (isEmpty()) {
            front = newNode;
            rear = newNode;
        } else {
            // Link the new node with current front
            newNode.next = front;
            front.prev = newNode;
            front = newNode;
        }

        size++;
    }
```

```
public void insertRear(T element) {
    Node newNode = new Node(element);

    // If dequeue is empty
    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {
        // Link the new node with current rear
        newNode.prev = rear;
        rear.next = newNode;
        rear = newNode;
    }

    size++;
}

public T deleteFront() {
    if (isEmpty()) {
        System.out.println("Dequeue is empty");
    }

    T removed = front.data;

    // If this is the only element
    if (front == rear) {
        front = null;
        rear = null;
    } else {
        front = front.next;
        front.prev = null;
    }

    size--;
    return removed;
}

public T deleteRear() {
    if (isEmpty()) {
        System.out.println("Dequeue is empty");
    }

    T removed = rear.data;

    // If this is the only element
    if (front == rear) {
        front = null;
        rear = null;
    } else {
        rear = rear.prev;
        rear.next = null;
    }

    size--;
    return removed;
}
```

```
    public T getFront() {
        if (isEmpty()) {
            System.out.println("Dequeue is empty");
        }
        return front.data;
    }

    public T getRear() {
        if (isEmpty()) {
            System.out.println("Dequeue is empty");
        }
        return rear.data;
    }
}
```

The linked implementation offers several advantages over the contiguous implementation

- **Dynamic size**: No fixed capacity, allowing the dequeue to grow as needed
- **Efficient operations**: All insertions and deletions are O (1) operations, like the array implementation
- **No wasted space**: The linked list only uses memory for the actual elements stored, plus the overhead of the node references

However, the linked implementation does have some downsides:

- **Memory overhead**: Each element requires additional memory for storing the next and previous references
- **No random access**: Elements can only be accessed sequentially, starting from either end of the dequeue

6. Conclusion

The dequeue data structure provides a versatile solution for scenarios requiring efficient insertion and deletion operations at both ends of a collection. Both the contiguous (array-based) and linked implementations offer same time for all basic operations, with different trade-offs in terms of memory usage and flexibility.

The choice between contiguous and linked implementations depends on the specific requirements of the application

- Choose the contiguous implementation when the maximum size is known in advance and memory efficiency is important.
- Choose the linked implementation when flexibility in size is needed and the overhead of node references is acceptable.

The dequeue's bidirectional nature makes it suitable for a wide range of applications, including

- Task scheduling algorithms
  - Priority-based job scheduling
  - Process management in operating systems
  - Event handling in real-time systems
- Stealing algorithms in work-stealing schedulers
  - Load balancing across multiple threads
  - Parallel task execution frameworks
  - Distributed computing workload distribution
- Implementation of various algorithms requiring operations at both ends of a data structure
  - Sliding window algorithms
  - BFS traversals with level tracking
  - Expression evaluation
  - Graph algorithms with frontier management
- Palindrome checking
  - String validation
  - Sequence symmetry verification
  - Pattern matching
- Implementations of other data structures like stacks and queues
  - Stack implementation (restricted to one end)
  - Queue implementation (using front and rear)
  - LRU cache implementation
  - Buffer management systems

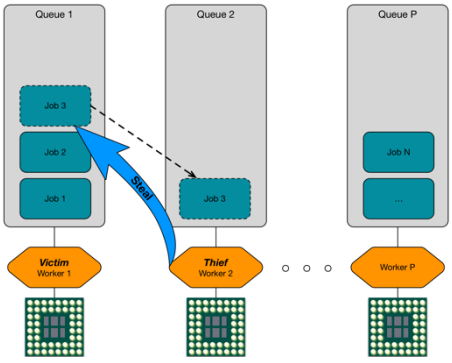

Figure 5 - Work Stealing

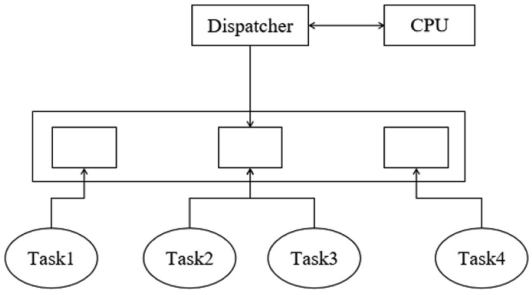

Figure 4 - Task Scheduling

References

1. https://actor-framework.readthedocs.io/en/0.17.5/Scheduler.html
2. https://er.yuvayana.org/double-ended-queue-deque-in-data-structures/
3. https://www.simplilearn.com/tutorials/data-structure-tutorial/dequeue-in-data-structure
4. https://www.geeksforgeeks.org/deque-interface-java-example/
5. https://www.scaler.com/topics/java/deque-in-java/
6. https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html