PES UNIVERSITY

Data Structures

Dilip Maripuri
Associate Professor
Department of Computer Applications

Data Structures

Session Outline

PES UNIVERSITY

Queue

Queue as an ADT

Linear Queue

Queue - What is a Queue?

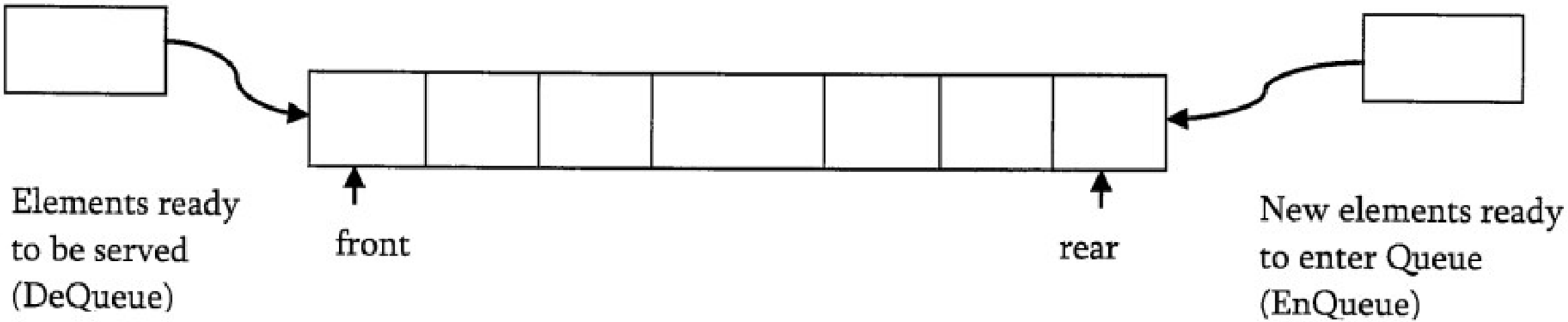
PES UNIVERSITY

- A queue, in computer science, is a linear data structure that operates on the First In, First Out (FIFO) principle.
- This principle dictates that the elements are added and removed in a sequential order, specifically ensuring that the first element added to the queue will be the first one to be removed.
- This structure is fundamentally similar to queues observed in daily life, such as people lining up at a checkout counter or vehicles in a traffic line.

Queue - Characteristics

- **Addition at One End (Rear)**
  - In a queue, new elements are added at one end, commonly referred to as the 'rear' or 'tail' of the queue.
  - This process is analogous to a new person joining the end of a line.
- **Removal from the Other End (Front)**
  - The removal of elements occurs at the opposite end, known as the 'front' or 'head' of the queue.
  - This resembles the person at the front of a line being served and then leaving.

Queue - Characteristics



Queue - Characteristics

PES UNIVERSITY

- **Linear Structure**
  - A queue maintains a linear formation, with each element having a specific position relative to the others, typically ordered by the time of addition.
- **One-Way Traffic**
  - Operations in a queue typically move in one direction from rear to front, reflecting a structured process flow.

Queue - How is a Queue Used?

PES UNIVERSITY

- **Real-world Analogy**
  - The concept of a queue is easily relatable to everyday scenarios.
  - For instance, a ticket counter where people line up and are served in the order they arrived.
    - This orderly process ensures fairness and organization.

Queue - How is a Queue Used?

PES UNIVERSITY

- **Computing Analogy**
  - **Process and Task Management**
    - In operating systems, queues are instrumental in process scheduling.
    - Processes are lined up in a queue and are allocated CPU time in the order they arrive.
    - This ensures a fair and efficient management of CPU resources.

Queue - How is a Queue Used?

PES UNIVERSITY

- **Computing Analogy**
  - **Data Stream Processing**
    - In applications dealing with streaming data, such as video streaming or real-time data analytics, queues help manage the data packets.
    - They ensure that packets are processed in the order they are received, maintaining data integrity and flow consistency.

Queue - How is a Queue Used?

PES UNIVERSITY

- **Computing Analogy – Data Handling**
  - **Sequential Data Processing**
    - In algorithms and applications, queues are used to handle data that must be processed in the order it was received or generated.
    - This is crucial in applications like simulations and event-driven programming.

Queue - How is a Queue Used?

PES UNIVERSITY

- **Computing Analogy – Data Handling**
  - **Buffering**
    - Queues act as buffers in systems where there is a disparity between the rate of data input and output.
    - For instance, in a print spooler, print jobs are queued up and executed one by one, accommodating the printer's limited processing capacity.

Queue - How is a Queue Used?

PES UNIVERSITY

- **Computing Analogy – Data Handling**
  - **Load Balancing**
    - In distributed computing and web server management, queues are used for load balancing.
    - Incoming requests are queued and then distributed amongst multiple servers or processes to optimize response time and system utilization.

Queue - How is a Queue Used?

PES UNIVERSITY

- **Computing Analogy – Data Handling**
  - **Asynchronous Data Transfer**
    - In scenarios involving asynchronous data transfer, such as between different software components or networked systems, queues play a pivotal role.
    - They allow data to be stored and transmitted when the receiving end is ready, ensuring that no data is lost or overwritten.

Queue ADT

PES UNIVERSITY

```
ADT Queue
The Queue operates on a FIFO (First-In-First-Out) basis.

Data:
        A collection of elements in a specific order.


Operations:
        InitializeQueue:
                Preconditions: None.
                Postconditions: An empty queue is created.
                Description: Initializes a new, empty queue.
        IsEmpty:
                Preconditions: None.
                Postconditions: Returns true if the queue is empty, false otherwise.
                Description: Checks if the queue has no elements.
```

Queue ADT

Enqueue:

Preconditions: The queue is not full (if capacity is limited).
Postconditions: The new element is added at the end of the queue.
Description: Adds a new element to the rear of the queue.

Dequeue:

Preconditions: The queue is not empty.
Postconditions: The front element is removed from the queue and returned.
Description: Removes and returns the front element of the queue.

Queue ADT


```
        Peek:
                Preconditions: The queue is not empty.
                Postconditions: Returns the front element without removing it.
                Description: Retrieves the front element of the queue without removing it.
        Size:
                Preconditions: None.
                Postconditions: Returns the number of elements in the queue.
                Description: Counts the number of elements in the queue.
        Clear:
                Preconditions: None.
                Postconditions: The queue is empty.
                Description: Removes all elements from the queue, making it empty.
End ADT
```

Queue ADT: Exceptions - Empty Queue Exception

- **When it occurs**
  - This exception is thrown when an operation that requires elements to be present in the queue is attempted on an empty queue.
  - The most common operation that would cause this exception is `Dequeue`, which removes and returns the front element of the queue.
- **What it signifies**
  - An Empty Queue Exception indicates that the queue does not contain any elements to perform the operation and thus the operation is invalid in the current state.
- **How it is handled**
  - When this exception occurs, the program must handle it gracefully, perhaps by informing the user that the queue is empty or by performing some other action that does not involve queue manipulation.

Queue ADT: Exceptions - Full Queue Exception

- **When it occurs**
  - This exception is thrown when an attempt is made to add an element to a queue that has reached its maximum capacity.
  - This is common in queues implemented with a static array where the size of the queue is fixed.
- **What it signifies**
  - A Full Queue Exception signals that there is no more room in the queue to add additional elements.
- **How it is handled**
  - Handling this exception may involve informing the user that no more items can be added or implementing some logic to increase the capacity of the queue, if possible.

Queue ADT : Implementation using Static Arrays

PES UNIVERSITY

```c
#define QUEUE_MAX_SIZE 100

typedef struct {
    int items[QUEUE_MAX_SIZE];
    int front, rear;
} Queue;

// Function to initialize the queue
void initializeQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}
```

```c
// Check if the queue is empty
int isEmpty(Queue *q) {
    return q->rear == -1;
}

// Check if the queue is full
int isFull(Queue *q) {
    return q->rear == QUEUE_MAX_SIZE - 1;
}
```

Queue ADT : Implementation using Static Arrays

PES UNIVERSITY

```
// Add an element to the queue
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot insert %d\n", value);
    } else {
        if (q->front == -1) { // First element to be inserted
            q->front = 0;
        }
        q->rear++;
        q->items[q->rear] = value;
    }
}
```

Queue ADT : Implementation using Static Arrays

PES UNIVERSITY

```
// Remove and return the front element from the queue
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue\n");
        return -1;
    } else {
        int item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) { // Queue has become empty
            initializeQueue(q);
        }
        return item;
    }
}
```

Queue ADT : Implementation using Static Arrays


```c
// Get the first element of the queue
int peek(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. No front element\n");
        return -1;
    } else
        return q->items[q->front];
}

// A function to display the queue
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements are:\n");
        for (int i = q->front; i <= q->rear; i++)
            printf("%d ", q->items[i]);
    }
}
```

Queue ADT : Implementation using Static Arrays

PES UNIVERSITY

```
int main() {
    Queue q;
    initializeQueue(&q);

    // Demonstration of enqueue operation
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);

    // Display the queue
    displayQueue(&q);

    // Demonstration of dequeue operation
    int item = dequeue(&q);
    printf("Dequeued item: %d\n", item);

    // Display the queue after dequeue    displayQueue(&q);
```

Queue ADT : Implementation using Dynamic Arrays


```
// Define Queue Data Type
typedef struct QueueADT {
    int *queue;
    int front, rear;
    int capacity;
} *Queue;
```

Queue ADT : Implementation using Dynamic Arrays

PES UNIVERSITY

```
// Function to create a queue of given capacity
Queue CreateQueue(int capacity) {
    Queue newQueue = (Queue)malloc(sizeof(struct QueueADT));
    if (newQueue == NULL) {
        printf("\n\t\t Insufficient Memory !!! Exiting !!!");
        exit(EXIT_FAILURE);
    }
    newQueue->queue = (int *)malloc(sizeof(int) * capacity);
    if (newQueue->queue == NULL) {
        printf("\n\t\t Insufficient Memory !!! Exiting !!!");
        exit(EXIT_FAILURE);
    }
    newQueue->front = -1;
    newQueue->rear = -1;
    newQueue->capacity = capacity;
    printf("\n\t\t Successfully Created Queue ADT\n");
    return newQueue;
}
```

Queue ADT : Implementation using Dynamic Arrays

PES UNIVERSITY

```
// Function to create a queue of given capacity
Queue CreateQueue(int capacity) {
    Queue newQueue = (Queue)malloc(sizeof(struct QueueADT));
    if (newQueue == NULL) {
        printf("\n\t\t Insufficient Memory !!! Exiting !!!");
        exit(EXIT_FAILURE);
    }
    newQueue->queue = (int *)malloc(sizeof(int) * capacity);
    if (newQueue->queue == NULL) {
        printf("\n\t\t Insufficient Memory !!! Exiting !!!");
        exit(EXIT_FAILURE);
    }
    newQueue->front = -1;
    newQueue->rear = -1;
    newQueue->capacity = capacity;
    printf("\n\t\t Successfully Created Queue ADT\n");
    return newQueue;
}
```

Queue ADT : Implementation using Dynamic Arrays

PES UNIVERSITY

```c
// Function to resize the queue
int ResizeQueue(Queue myQueue, int sizeChange) {
    int newSize = myQueue->capacity + sizeChange;
    int *newArray = (int *)realloc(myQueue->queue, sizeof(int) * newSize);

    if (newArray == NULL) {
        printf("\n\t\t Insufficient Memory !!! Exiting !!!");
        return 0;
    }
    myQueue->capacity = newSize;
    myQueue->queue = newArray;
    return 1;
}
```

Queue ADT : Implementation using Dynamic Arrays

PES UNIVERSITY

```
// Function to add an element to the queue
void Enqueue(Queue myQueue, int item) {
    if (myQueue->rear == myQueue->capacity - 1) {
    // Resize if the queue is full
        if (!ResizeQueue(myQueue, 1)) {
            printf("\n\t\t Queue is Full !!!\n");
            return;
        }
    }
    if (myQueue->front == -1) // If queue is empty
        myQueue->front = 0;
    myQueue->queue[++myQueue->rear] = item;
    printf("\n\t\t Enqueued %d\n", item);
}
```

Queue ADT : Implementation using Dynamic Arrays

PES UNIVERSITY

```
// Function to remove an element from the queue
void Dequeue(Queue myQueue) {
    if (myQueue->front == -1) {
        printf("\n\t\t Queue is Empty !!!\n");
        return;
    }
    printf("\n\t\t Dequeued %d\n", myQueue->queue[myQueue->front]);
    if (myQueue->front == myQueue->rear)
        myQueue->front = myQueue->rear = -1;
    else
        myQueue->front++;
    ResizeQueue(myQueue, -1)}
}
```

Queue ADT : Implementation using Dynamic Arrays

PES UNIVERSITY

```c
// Function to display the queue contents
void DisplayQueue(Queue myQueue) {
    if (myQueue->front == -1) {
        printf("\n\t\t Queue is empty\n");
        return;
    }
    printf("\n\t\t Queue: ");
    for (int i = myQueue->front; i <= myQueue->rear; i++)
        printf("%d ", myQueue->queue[i]);
    printf("\n");
}

// Function to free the allocated memory
void DestroyQueue(Queue myQueue) {
    free(myQueue->queue);
    free(myQueue);
}
```

Queue ADT : Implementation using Dynamic Arrays

PES UNIVERSITY

```
Queue myQueue = CreateQueue(1);
Enqueue(myQueue, item);
Dequeue(myQueue);
DisplayQueue(myQueue);
DestroyQueue(myQueue);
```

Queue ADT : Implementation using Linked Lists

PES UNIVERSITY

- **Linked List Implementation**
  - **Enqueue operation**
    - Insert_at_End Operation
  - **Dequeue operation**
    - Delete_at_Front Operation
  - **Display operation**
    - Traverse and Display Operation

Queue ADT : Implementation using Linked Lists

PES UNIVERSITY

```
typedef struct node {
    int data;
    struct node *next;
} *NODE;

typedef struct {
    NODE front;
    NODE rear;
} *Queue;
```

```
NODE createNode(int value) {
    NODE newNode = (NODE)malloc(sizeof(struct node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

Queue ADT : Implementation using Linked Lists


```
Queue createQueue() {
    Queue q = (Queue)malloc(sizeof(struct queue));
    if (q == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    q->front = q->rear = NULL;
    return q;
}
```

```
int isEmpty(Queue q) {
    return q == NULL || q->front == NULL;
}

void enqueue(Queue q, int value) {
    if (q == NULL) {
        printf("Queue not initialized!\n");
        return;
    }
```

Queue ADT : Implementation using Linked Lists

PES UNIVERSITY

```c
void enqueue(Queue q, int value) {
    if (q == NULL) {
        printf("Queue not initialized!\n")
        return;
    }

    NODE newNode = createNode(value);
    if (newNode == NULL) return;

    if (isEmpty(q)) {
        q->front = q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
}
```

```c
int dequeue(Queue q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    }

    int value = q->front->data;
    NODE temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    free(temp);
    return value;
}
```

Queue ADT : Implementation using Linked Lists

PES UNIVERSITY

```
void displayQueue(Queue q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }

    NODE temp = q->front;
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

```
enqueue(q, 30);
displayQueue(q);
printf("Dequeued: %d\n", dequeue(q));
free(q);
return 0;
```

Queue ADT : Circular Queues

- **Drawbacks of Linear Queues**
  - **Fixed Size**
    - Traditional linear queues have a fixed size, meaning the number of elements it can hold is defined at the time of its creation.
    - This can lead to inefficiency in memory usage, as we might allocate more memory than actually needed, or face a situation where the queue runs out of space.

PES UNIVERSITY

- **Drawbacks of Linear Queues**
  - **Inefficient Memory**
    - In a linear queue, even if elements are dequeued from the front, the space they occupied is not reused.
    - Over time, as elements are added to the rear and removed from the front, the queue might reach a state where it cannot accept new elements despite having free space, because this space is non-contiguous and at the front.

Queue ADT : Circular Queues

- The concept of a circular queue is an extension and improvement over the linear queue model.
- In a circular queue, the last position is connected back to the first position to make a circle, effectively utilizing the queue's storage capacity more efficiently.
- This design addresses several limitations of the linear queue.

Queue ADT : Circular Queues

PES UNIVERSITY

- **Key Characteristics of Circular Queues**
  - **Circular Structure**
    - In a circular queue, when an element is removed from the front, the space it occupied becomes available for new elements.
    - This way, the queue makes the most of its allocated memory, avoiding the wastage seen in linear queues.

PES UNIVERSITY

- **Key Characteristics of Circular Queues**
  - **Efficient Memory Utilization**
    - Unlike linear queues, the circular queue does not suffer from the problem of "false overflow."
    - This is where a linear queue might appear to be full even though there are unused memory spaces, due to the linear arrangement of elements.
    - In a circular queue, as soon as the rear reaches the end of the queue, it can circle back to the beginning (if there is space available).

Queue ADT : Circular Queues

PES UNIVERSITY

- **Key Characteristics of Circular Queues**
  - **Fixed Size but Flexible Utilization**
    - While a circular queue still has a fixed size, it utilizes its capacity more efficiently.
    - This is because the positions freed by dequeued elements can be reused for new elements.

Queue ADT : Circular Queues

PES UNIVERSITY

- **Operations**
  - **Enqueue**
    - This operation adds an element to the rear of the queue.
    - If the rear of the queue is at the last position and there is space at the beginning, the rear is moved to the beginning to add the element.
  - **Dequeue**
    - This removes an element from the front of the queue.
    - If the queue is not empty, the front pointer is moved to the next element in the queue.

Queue ADT : Circular Queues

- **Handling Overflow and Underflow**
  - A circular queue needs to handle underflow and overflow.
  - Due to its circular nature, checking for overflow is slightly different, as it needs to consider both the front and rear positions.

Queue ADT : Circular Queues- Array Implementation

PES UNIVERSITY

```
#define MAXSIZE 10
typedef struct {
    int items[MAXSIZE];
    int front, rear;
} CircularQueue;

void InitializeQueue(CircularQueue *q)
{    q->front = q->rear = -1;   }

int IsEmpty(CircularQueue *q)
{    return (q->front == -1);   }

int IsFull(CircularQueue *q) {
    return ((q->rear + 1) % MAXSIZE == q->front);
}
```

Queue ADT : Circular Queues- Array Implementation

PES UNIVERSITY

```
// Function to add an element to the queue
void Enqueue(CircularQueue *q, int element) {
    if (IsFull(q)) {
        printf("\n\t\t Queue is full!\n");
    } else {
        if (q->front == -1) q->front = 0;
        q->rear = (q->rear + 1) % MAXSIZE;
        q->items[q->rear] = element;
        printf("\n\t\t Enqueued %d\n", element);
    }
}
```

Queue ADT : Circular Queues- Array Implementation

PES UNIVERSITY

```
// Function to remove an element from the queue
int Dequeue(CircularQueue *q) {
    int element;
    if (IsEmpty(q)) {
        printf("\n\t\t Queue is empty!\n");
        return -1;
    } else {
        element = q->items[q->front];
        if (q->front == q->rear)
            q->front = q->rear = -1;
        else
            q->front = (q->front + 1) % MAXSIZE;
        printf("\n\t\t Dequeued %d\n", element);
        return element;
    }
}
```

Queue ADT : Circular Queues- Array Implementation

PES UNIVERSITY

```
// Function to display the queue
void DisplayQueue(CircularQueue *q) {
    int i;
    if (IsEmpty(q)) {
        printf("\n\t\t Queue is empty!\n");
    } else {
        printf("\n\t\t Queue: ");
        for (i = q->front; i != q->rear; i = (i + 1) % MAXSIZE)
            printf("%d ", q->items[i]);
        // Print the last element
        printf("%d", q->items[i]);
        printf("\n");
    }
}
```

Queue ADT : Circular Queues- Array Implementation

PES UNIVERSITY

```
// Function to display the queue
void DisplayQueue(CircularQueue *q) {
    int i;
    if (IsEmpty(q)) {
        printf("\n\t\t Queue is empty!\n");
    } else {
        printf("\n\t\t Queue: ");
        for (i = q->front; i != q->rear; i = (i + 1) % MAXSIZE)
            printf("%d ", q->items[i]);
        // Print the last element
        printf("%d", q->items[i]);
        printf("\n");
    }
}
```

Queue ADT : Circular Queues- Array Implementation

PES UNIVERSITY

```
CircularQueue q;
InitializeQueue(&q);
Enqueue(&q, element);
Dequeue(&q);
DisplayQueue(&q);
```

Queue ADT : Circular Queue Implementation using SLL

PES UNIVERSITY

- **Structures Used**
  - **Node Structure (NODE)**
    - data: Integer to store the value.
    - next: Pointer to the next NODE.
  - **Circular Queue Structure (CircularQueue)**
    - front: Pointer to the front node of the queue.
    - rear: Pointer to the rear node of the queue.

```
typedef struct node {
    int data;
    struct node *next;
} *NODE;

typedef struct {
    NODE front;
    NODE rear;
} *CircularQueue;
```

Queue ADT : Circular Queue Implementation using SLL

PES UNIVERSITY

- **Function createNode(int value)**
  - **Purpose :** Create a new node with a given value.
  - **Input :** value - The integer value for the new node.
  - **Output :** Pointer to the newly created node (NODE).
  - **Steps**
    - Allocate memory for a new NODE.
    - If memory allocation fails, display an error message and return NULL.
    - Set the data field of the new node to value.
    - Set the next field of the new node to point to itself.
    - Return the new node.

Queue ADT : Circular Queue Implementation using SLL

PES UNIVERSITY

```
NODE createNode(int value) {
    NODE newNode = (NODE)malloc(sizeof(struct node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    newNode->data = value;
    newNode->next = newNode;
    return newNode;
}
```

Queue ADT : Circular Queue Implementation using SLL

- **Function createQueue()**
  - **Purpose :** Initialize a new circular queue.
  - **Input :** None
  - **Output :** Pointer to the newly created queue (CircularQueue).
  - **Steps**
    - Allocate memory for a CircularQueue.
    - If memory allocation fails, display an error message and return NULL.
    - Initialize front and rear of the queue to NULL.
    - Return the pointer to the queue.

Queue ADT : Circular Queue Implementation using SLL

PES UNIVERSITY

```
CircularQueue createQueue() {
    CircularQueue q = (CircularQueue)malloc(sizeof(struct queue));
    if (q == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }
    q->front = q->rear = NULL;
    return q;
}
```

Queue ADT : Circular Queue Implementation using SLL

PES UNIVERSITY

- **Function isEmpty(CircularQueue q)**
  - **Purpose :** Check if the queue is empty.
  - **Input :** q - The circular queue.
  - **Output :** Boolean value indicating whether the queue is empty.
  - **Steps**
    - If q is NULL or q->front is NULL, return TRUE.
    - Otherwise, return FALSE.

Queue ADT : Circular Queue Implementation using SLL


```
int isEmpty(CircularQueue q)
{    return q == NULL || q->front == NULL;      }
```

Queue ADT : Circular Queue Implementation using SLL

PES UNIVERSITY

- **Function enqueue(CircularQueue q, int value)**
  - **Purpose :** Add a new element to the rear of the queue.
  - **Input :** q - The circular queue, value - The value to be enqueued.
  - **Output :** None
  - **Steps**
    - If q is NULL, print an error message and exit the function.
    - Create a new node with value.
    - If the queue is empty, set both front and rear to the new node.
    - Else, set the next of rear to the new node and update rear to this new node.
    - Set the next of rear to front to maintain circularity.

Queue ADT : Circular Queue Implementation using SLL


```
void enqueue(CircularQueue q, int value) {
    if (q == NULL) {
        printf("Queue not initialized!\n");
        return;
    }

    NODE newNode = createNode(value);
    if (newNode == NULL) return;

    if (isEmpty(q))
        q->front = q->rear = newNode;
    else {
        newNode->next = q->front;
        q->rear->next = newNode;
        q->rear = newNode;
    }
}
```

Queue ADT : Circular Queue Implementation using SLL

- **Function dequeue(CircularQueue q)**
  - **Purpose :** Remove and return the front element from the queue.
  - **Input :** q - The circular queue.
  - **Output :** The value of the front element.
  - **Steps**
    - If the queue is empty (isEmpty(q) is TRUE), print an error message and return a sentinel value.
    - Store the value of the front node.
    - If front is the same as rear (only one element in the queue), set both to NULL.
    - Else, set front to the next node and update rear->next to the new front.
    - Free the memory of the old front node.
    - Return the stored value.

Queue ADT : Circular Queue Implementation using SLL


```
int dequeue(CircularQueue q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    }

    int value = q->front->data;
    if (q->front == q->rear) {
        free(q->front);
        q->front = q->rear = NULL;
    } else {
        NODE temp = q->front;
        q->front = q->front->next;
        q->rear->next = q->front;
        free(temp);
    }
    return value;
}
```

Queue ADT : Circular Queue Implementation using SLL

PES UNIVERSITY

- **Function displayQueue(CircularQueue q)**
  - **Purpose :** Display all elements of the queue.
  - **Input :** q - The circular queue.
  - **Output :** None
  - **Steps**
    - If the queue is empty, print an appropriate message.
    - Else, iterate from front to rear (following next pointers) and print each element's value.
    - Continue looping until reaching front again.

Queue ADT : Circular Queue Implementation using SLL


```
void displayQueue(CircularQueue q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }

    NODE temp = q->front;
    printf("Queue: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != q->front);
    printf("\n");
}
```

Queue ADT : Circular Queue Implementation using SLL


```
enqueue(q, 30);
displayQueue(q);
printf("Dequeued: %d\n", dequeue(q));

free(q);
```

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **What is DeQue**
  - A Doubly Ended Queue, commonly known as a Deque.
  - It is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front or the rear.
  - This is in contrast to a simple queue, where insertion happens at the rear and removal happens at the front.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **Characteristics of a Deque**
  - **Two Ends for Operations**
    - Elements can be added or removed from both the front and the rear.
  - **No FIFO or LIFO Constraints**
    - Unlike regular queues (FIFO - First In First Out) or stacks (LIFO - Last In First Out), deques don't follow a strict linear order for accessing elements.
  - **Random Access**
    - Deques can allow random access to elements, although this is not always implemented.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - Data Elements
    - **Deque:** A linear list of elements, with a designated front and rear.
    - **Front:** A pointer or index to the front end of the deque.
    - **Rear:** A pointer or index to the rear end of the deque.
    - **Size:** An optional attribute representing the maximum capacity of the deque.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **createDeque(int maxSize)**
      - **Description:** Initializes the deque with a maximum size of maxSize.
      - **Input:** maxSize The maximum number of elements that the deque can hold.
      - **Output:** A new instance of an Input-Restricted Deque.
      - **Post-Condition:** An empty deque is created with a specified maximum size.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **isEmpty(Deque d)**
      - **Description:** Checks if the deque is empty.
      - **Input:** d The deque to check.
      - **Output:** true if the deque is empty, false otherwise.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **isFull(Deque d)**
      - **Description:** Checks if the deque is full.
      - **Input:** d The deque to check.
      - **Output:** true if the deque is full, false otherwise.

Queue ADT : Doubly Ended Queue

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **insertRear(Deque d, ElementType item)**
      - **Description:** Inserts an element at the rear of the deque.
      - **Input:** d The deque, item The element to be inserted.
      - **Pre-Condition:** The deque must not be full.
      - **Post-Condition:** The element is added to the rear of the deque.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **deleteFront(Deque d)**
      - **Description:** Deletes an element from the front of the deque.
      - **Input:** d The deque.
      - **Pre-Condition:** The deque must not be empty.
      - **Output:** The element removed from the front.
      - **Post-Condition:** The front element is removed from the deque.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **deleteRear(Deque d)**
      - **Description:** Deletes an element from the rear of the deque.
      - **Input:** d The deque.
      - **Pre-Condition:** The deque must not be empty.
      - **Output:** The element removed from the rear.
      - **Post-Condition:** The rear element is removed from the deque.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **getFront(Deque d)**
      - **Description:** Retrieves the element at the front of the deque without removing it.
      - **Input:** d The deque.
      - **Pre-Condition:** The deque must not be empty.
      - **Output:** The element at the front of the deque.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **getRear(Deque d)**
      - **Description:** Retrieves the element at the rear of the deque without removing it.
      - **Input:** d The deque.
      - **Pre-Condition:** The deque must not be empty.
      - **Output:** The element at the rear of the deque.

Queue ADT : Doubly Ended Queue

PES UNIVERSITY

- **ADT: Input-Restricted Deque**
  - **Operations**
    - **size(Deque d)**
      - **Description:** Returns the number of elements in the deque.
      - **Input:** d The deque.
      - **Output:** The number of elements in the deque.

Queue ADT : Doubly Ended Queue Implementation using Static Arrays

PES UNIVERSITY

```
typedef struct {
    int arr[MAXSIZE];
    int front,rear, size;
} *Deque;

Deque createDeque();
int isFull(Deque dq);
int isEmpty(Deque dq);
void insertRear(Deque dq, int item);
int deleteFront(Deque dq);
int deleteRear(Deque dq);
int getFront(Deque dq);
int getRear(Deque dq);
void displayDeque(Deque dq);
```

Queue ADT : Doubly Ended Queue Implementation using Static Arrays

PES UNIVERSITY

```
Deque createDeque() {
    Deque dq = (Deque)malloc(sizeof(*dq));
    if (dq == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    dq->front = dq->rear = -1;
    dq->size = 0;
    return dq;
}
```

```
int isFull(Deque dq)
{    return dq->size == MAXSIZE;          }

int isEmpty(Deque dq)
{    return dq->size == 0;          }
```

Queue ADT : Doubly Ended Queue Implementation using Static Arrays

PES UNIVERSITY

```
void insertRear(Deque dq, int item) {
    if (isFull(dq)) {
        printf("Deque is full!\n");
        return;
    }
    dq->arr[++dq->rear] = item;
    if (dq->front == -1) dq->front = 0;
    dq->size++;
}
```

```
int isFull(Deque dq)
{    return dq->size == MAXSIZE;}

int isEmpty(Deque dq)
{    return dq->size == 0;      }
```

Queue ADT : Doubly Ended Queue Implementation using Static Arrays

PES UNIVERSITY

```
int deleteFront(Deque dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty!\n");
        return -1;
    }
    int item = dq->arr[(dq->front)++];
    dq->size--;
    if (isEmpty(dq)) dq->front = dq->rear = -1;
    return item;
}
```

Queue ADT : Doubly Ended Queue Implementation using Static Arrays

PES UNIVERSITY

```
int deleteRear(Deque dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty!\n");
        return -1;
    }
    dq->rear = (dq->rear - 1 + MAXSIZE) % MAXSIZE;
    int item = dq->arr[dq->rear];
    dq->size--;
    if (isEmpty(dq)) dq->front = -1;
    return item;
}
```

Queue ADT : Doubly Ended Queue Implementation using Static Arrays

PES UNIVERSITY

```
int getFront(Deque dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty!\n");
        return -1;
    }
    return dq->arr[dq->front];
}
```

```
int getRear(Deque dq) {
    if (isEmpty(dq)) {
        printf("Deque is empty!\n");
        return -1;
    }
    return dq->arr[dq->rear];
}
```

Queue ADT : Doubly Ended Queue Implementation using Static Arrays

PES UNIVERSITY

```
void displayDeque(Deque dq) {
    int i;
    if (isEmpty(dq)) {
        printf("Deque is empty!\n");
        return;
    }
    printf("Deque: ");
    for(i = dq->front;i <= dq->rear;i++)
        printf("%d ", dq->arr[i]);
    printf("\n");
}
```

Queue ADT : Doubly Ended Queue Implementation using SLL / DLL

PES UNIVERSITY

- **Linked List Operations that are applicable for implementing both the variants of Deque**
  - Insert_at_Front
  - Delete_at_Front
  - Insert_at_Rear
  - Delete_at_Rear
  - Display The Queue Elements

THANK YOU

Dilip Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu