Faculty of Information Technology
University of Moratuwa

BSc(Hons) in Information Technology
BSc(Hons) in Artificial Intelligence

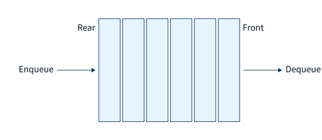
IN 1111 - Data Structures and Algorithm part 1

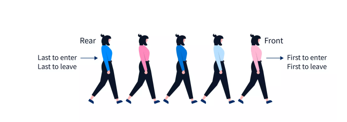
Level 1 - Semester 2 | Lecture Note 05

Queues

1.1 What is a Queue

- A queue is a fundamental data structure in computer science that follows the First In, First Out (FIFO) principle.
- This means that the most recently added element is the last one to be removed.
- Queues are commonly used in scenarios where the order of processing elements is important and where elements need to be processed in the same order they were added.
- Some examples of applications of queues include CPU scheduling in operating systems, print spooling, task scheduling in networking, and asynchronous communication between processes.



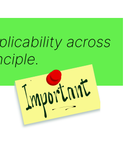


- In C, the Queue data structure is an ordered, linear sequence of items.
- It is a sequential data type, unlike an array. In an array, we can access any of its elements using indexing, but we can only access the right most element in a queue.
- There are two types of Queue data structure: Static and Dynamic
  - **Static Queue**
    - A Static Queue (also known as a bounded queue) has a bounded capacity.
    - It can contain a limited number of elements. If a static queue is full and does not have any space remaining for another element to be queued to it, it is then called to be in an Overflow State.
    - In C, a static queue is implemented using an Array, as arrays are static.
  - **Dynamic Queue**
    - A Dynamic Queue is a queue data structure whose capacity (maximum elements that can be stored) increases or decreases in runtime, based on the operations (enqueue or dequeue) performed on it.
    - In C, a dynamic queue is implemented using a Linked List, as linked lists are dynamic data structures.

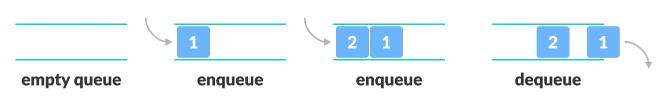
1.2 Why Queue

- The choice of using a queue data structure often depends on the specific problem or task at hand and the requirements of the algorithm or application being developed.
- Here are several reasons why queues are commonly used:
  - **First In, First Out (FIFO) Behaviour:** The FIFO principle enforced by queues makes them suitable for scenarios where the elements need to be processed in the order they were added. This behaviour is essential in many real-world applications such as task scheduling, job processing, and print spooling.
  - **Simple and Efficient Operations:** Queues typically support operations like enqueue and dequeue in constant time complexity (O(1)), assuming a well-designed implementation. This makes them efficient for adding and removing elements from either end of the queue.
  - **Straightforward Interface:** Queues have a simple interface with a limited set of operations (enqueue, dequeue, peek, isEmpty, size), which makes them easy to understand, use, and implement. This simplicity is advantageous for writing clean and maintainable code.
  - **Predictable Performance:** The performance characteristics of queues are well-defined and predictable. Since operations like enqueue and dequeue are always performed in constant time, the behaviour of a queue is consistent and reliable.
  - **Synchronization and Concurrency:** Queues are often used in multi-threaded or concurrent programming environments to manage communication and synchronization between threads or processes. They provide a convenient way to ensure that tasks or messages are processed in a controlled and orderly fashion.
  - **Memory Management:** Queues can be implemented using various underlying data structures, such as arrays or linked lists, allowing flexibility in memory management based on the specific requirements of the application.

Overall, queues are a fundamental and versatile data structure that finds wide applicability across various domains due to their simplicity, efficiency, and adherence to the FIFO principle.



1.3 Queue Operations



- Enqueue: Adds an element to the rear of the queue.
- Dequeue: Removes an element from the front of the queue.
- Peek (or Front): Retrieves the element at the front of the queue without removing it.
- isEmpty: Checks if the queue is empty.
- isFull: This operation checks if the queue is full, although in many implementations, this isn't relevant because queues can resize dynamically.

1.4 Queue Implementation

1.4.1 Array-Based Implementation

- In this approach, a fixed-size array is used to store elements.
- Both the front and the rear of the queue is tracked using index variables.
- When enqueuing an element, it is added to the array at the index corresponding to the rear of the queue.
- When dequeuing an element, the element at the front index is removed, and the front index is incremented.
- This implementation is simple but may have limitations on the maximum capacity of the queue.

Queue Creation

```c
#define MAX_SIZE 100

int queue[MAX_SIZE];
int front = 0;  // Index of the front element
int rear = -1;  // Index of the rear element
int size = 0;   // Current size of the queue

bool isFull();
bool isEmpty();
void enqueue();
int dequeue();
int peek();
```

Enqueue

```c
// Function to add an element to the rear of the queue
void enqueue(int item) {
    if (isFull()) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    rear = (rear + 1) % MAX_SIZE;
    queue[rear] = item;
    size++;
}
```

Dequeue

```c
// Function to remove an element from the front of the queue
int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1; // Alternatively, you can throw an exception
    }
    int removedItem = queue[front];
    front = (front + 1) % MAX_SIZE;
    size--;
    return removedItem;
}
```

Peek

```c
// Function to return the front element of the queue without
    removing it
int peek() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot peek.\n");
        return -1; // Alternatively, you can throw an exception
    }
    return queue[front];
}
```

isEmpty and isFull

```c
// Function to check if the queue is empty
bool isEmpty() {
    return size == 0;
}

// Function to check if the queue is full
bool isFull() {
    return size == MAX_SIZE;
}
```

1.4.2 LinkedList-Based Implementation

- Here, a linked list data structure is used to implement the queue.
- Each node of the linked list contains an element and a reference to the next node.
- The front of the queue corresponds to the first node of the linked list.
- The rear of the queue corresponds to the tail node of the linked list.
- Enqueuing an element involves creating a new node and making it the last/tail node of the linked list.
- Dequeuing an element involves removing the first/head node of the linked list.

Queue Creation

```c
// Structure to represent a node in the queue
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Function to create a new node with given data
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Global pointers to maintain the front and rear of the queue
Node* front = NULL;
Node* rear = NULL;
```

Enqueue

```c
// Function to add an element to the rear of the queue
void enqueue(int data) {
    Node* newNode = createNode(data);
    if (isEmpty()) {
        front = newNode;
    } else {
        rear->next = newNode;
    }
    rear = newNode;
}
```

Dequeue

```c
// Function to remove an element from the front of the queue
int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot dequeue.\n");
        exit(1);
    }
    Node* temp = front;
    int data = temp->data;
    front = temp->next;
    free(temp);
    if (front == NULL) {
        rear = NULL;
    }
    return data;
}
```

Peek

```c
// Function to return the front element of the queue without
    removing it
int peek() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot peek.\n");
        exit(1);
    }
    return front->data;
}
```

isEmpty

```c
// Function to check if the queue is empty
int isEmpty() {
    return front == NULL;
}
```

1.6 Other types of Queues

- In the context of Data Structures and Algorithms (DSA), queues are a fundamental data structure that follows the First In, First Out (FIFO) principle.
- There are several types of queues in DSA, each serving different purposes and offering specific advantages based on the requirements of the application. Here are some common types of queues in DSA:
  - **Circular Queue:**
    - A circular queue is a variation of the linear queue where the last element is connected back to the first element, forming a circular structure.
    - Circular queues efficiently utilize space and avoid the overhead of shifting elements when dequeuing, making them suitable for scenarios where the queue needs to have a fixed size and elements need to be processed cyclically.
  - **Priority Queue:**
    - A priority queue is a queue where each element has an associated priority.
    - Elements are dequeued based on their priority, with higher priority elements dequeued before lower priority ones.
    - Priority queues can be implemented using various data structures such as heaps, balanced binary search trees, or arrays.
  - **Double-ended Queue (Deque):**
    - A deque supports insertion and deletion of elements from both ends of the queue.
    - Elements can be added or removed from the front (head) or the rear (tail) of the queue.
    - Deques are useful in scenarios where elements need to be added or removed from both ends efficiently.

1.7 Complexity Analysis

Data Structure		Complexity	Time Complexity			Space Complexity
		Operations	enqueue	dequeue	peek	
Queue	Array		O(1)	O(1)	O(1)	O(n)
	Linked list		O(1)	O(1)	O(1)	O(n)
Circular queue			O(1)	O(1)	O(1)	O(n)
Deque			O(1)	O(1)	O(1)	O(n)
Priority queue			O(log n)	O(log n)	O(1)	O(n)

1.8 Additional Notes