`

UNIVERSITY INSTITUTE OF COMPUTING

CHANDIGARH UNIVERSITY

GHARUAN, MOHALI (PUNJAB)

SUBMITTED TO:	SUBMITTED BY:
Name: Jyoti Rani	Student Name: Harshdeep Kaur
Desgination: Assistant Professor	UID: 24BCA10370
Chandigarh University	Sub Name: Data Structure
	Sub Code: 24CAT-151
Faculty Signature:_____	Section:24BCA6-B

📑 Table of Contents

🛒E-Commerce Order Processing System (Using Queue - Heap and Linked List in C)

🧠1. Abstract

In today's fast-paced digital world, e-commerce platforms are required to process thousands of orders in real time, prioritizing critical shipments while ensuring fairness and efficiency. Efficient order management is key to customer satisfaction and business success.

This project simulates a simplified **E-Commerce Order Processing System** using basic data structures in C. It uses a **Linked List** to represent normal customer orders that are processed in **First-In-First-Out (FIFO)** order. Meanwhile, urgent or high-value orders are processed using a **Priority Queue implemented with Max Heap**, ensuring orders with higher priority (like same-day delivery or VIP customers) are processed before others.

By combining these two core data structures, the system demonstrates how real-world order processing logic can be built from foundational algorithmic principles. This implementation allows users to add, process, and display both normal and priority orders, replicating how backend systems in retail and logistics prioritize work queues based on urgency and order time.

~📘 2. Introduction

As e-commerce continues to expand globally, order volume and complexity have increased dramatically. From electronics to groceries, customers now expect faster

deliveries and seamless order tracking. Companies must adapt by implementing automated systems that can queue and fulfill orders efficiently based on predefined rules.

Order processing involves receiving customer orders, storing them for dispatch, and fulfilling them within a committed timeframe. A well-designed processing system ensures:

- No order is lost or missed
- High-priority orders are fulfilled faster
- Resources are utilized efficiently

In this project, we simulate a basic backend system where:

- **Normal Orders** are queued using a **Linked List** (FIFO).
- **Priority Orders** are queued using a **Max Heap** (processed based on priority value).

The implementation provides insight into how even the simplest data structures can solve complex real-world problems when designed thoughtfully.

🎯 3. Objectives

- Simulate order entry and processing using fundamental C data structures.

- Demonstrate priority-based scheduling using Max Heap.
- Manage FIFO orders using a Linked List.
- Reinforce the practical use of queues and heaps in system design.

🔄 4. Algorithms Used

📌 A. Linked List for Normal Orders

Normal orders follow the FIFO principle, i.e., the first order added will be processed first.

Algorithm (Insert Normal Order):

1. Create a new node.
2. If the list is empty, point head to the new node.
3. Else, traverse to the end and add the node there.

Algorithm (Process Normal Order):

1. If head is NULL, no order to process.
2. Else, process head, then move head to head->next.

⚡ B. Max Heap for Priority Orders

Orders with higher priority (larger integer value) should be processed before others.

Algorithm (Insert into Max Heap):

1. Insert order at the end of the heap array.
2. Compare it with its parent:
    - If it has a higher priority than parent, swap.
3. Repeat until the heap property is restored (heapify up).

Algorithm (Process Max Priority Order):

1. Return the element at index 0.
2. Move the last element to root.
3. Perform heapify down:
    - Swap root with the largest child if necessary.
4. Repeat until the heap is valid.

✅ C Program: E-Commerce Order Processing System

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define MAX 100

// ------------------- Linked List for Normal Orders -------------------
typedef struct Node {
    int orderId;
    char item[50];
    struct Node* next;
} Node;

Node* head = NULL;

void addNormalOrder(int id, char* item) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->orderId = id;
    strcpy(temp->item, item);
    temp->next = NULL;

    if (head == NULL)
        head = temp;
```

```
    else {
        Node* p = head;
        while (p->next) p = p->next;
        p->next = temp;
    }

    printf("Normal order #%d added.\n", id);
}

void processNormalOrder() {
    if (head == NULL) {
        printf("No normal orders to process.\n");
        return;
    }
    Node* temp = head;
    printf("Processing normal order #%d: %s\n", temp
->orderId, temp->item);
    head = head->next;
    free(temp);
}
```

```
void displayNormalOrders() {
    Node* temp = head;
    printf("\nNormal Orders:\n");
    while (temp) {
        printf("Order #%d: %s\n", temp->orderId, temp->item);
        temp = temp->next;
    }
}
```

```
// ------------------- Heap for Priority Orders -------------------
typedef struct {
    int orderId;
    int priority;
    char item[50];
} Order;
```

```
Order heap[MAX];
int size = 0;

void swap(Order* a, Order* b) {
    Order temp = *a;
    *a = *b;
    *b = temp;
}

void heapifyUp(int index) {
    if (index <= 0) return;
    int parent = (index - 1) / 2;
    if (heap[index].priority > heap[parent].priority) {
        swap(&heap[index], &heap[parent]);
        heapifyUp(parent);
    }
}

void heapifyDown(int index) {
```

```
    int left = 2 * index + 1, right = 2 * index + 2, largest =
index;
    if (left < size && heap[left].priority > heap[largest].priority)
largest = left;
    if (right < size && heap[right].priority >
heap[largest].priority) largest = right;


    if (largest != index) {
        swap(&heap[index], &heap[largest]);
        heapifyDown(largest);
    }
}

void addPriorityOrder(int id, char* item, int priority) {
    Order o;
    o.orderId = id;
    o.priority = priority;
    strcpy(o.item, item);
    heap[size] = o;
    heapifyUp(size);
    size++;
```

```
    printf("Priority order #%d added with priority %d.\n", id,
priority);
}

void processPriorityOrder() {
    if (size == 0) {
        printf("No priority orders to process.\n");
        return;
    }
    printf("Processing priority order #%d (%s) with priority
 %d\n",
           heap[0].orderId, heap[0].item, heap[0].priority);
    heap[0] = heap[--size];
    heapifyDown(0);
}

void displayPriorityOrders() {
    printf("\nPriority Orders:\n");
    for (int i = 0; i < size; i++) {
        printf("Order #%d: %s [Priority: %d]\n", heap[i].orderId,
heap[i].item, heap[i].priority);
```

```
    }
}

// ------------------- Main Menu -------------------
int main() {
    int choice, id, priority;
    char item[50];

    while (1) {
        printf("\n--- E-Commerce Order Processing System ---
\n");
        printf("1. Add Normal Order\n2. Process Normal
Order\n3. Add Priority Order\n");
        printf("4. Process Priority Order\n5. Display All
Orders\n6. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter Order ID and Item: ");
```

```
            scanf("%d %s", &id, item);
            addNormalOrder(id, item);
            break;
        case 2:
            processNormalOrder();
            break;
        case 3:
            printf("Enter Order ID, Item, and Priority: ");
            scanf("%d %s %d", &id, item, &priority);
            addPriorityOrder(id, item, priority);
            break;
        case 4:
            processPriorityOrder();
            break;
        case 5:
            displayPriorityOrders();
            displayNormalOrders();
            break;
        case 6:
            exit(0);
```

```
        }
    }
    return 0;
}
```

🧪Sample Output

```
--- E-Commerce Order Processing System ---
1. Add Normal Order
2. Process Normal Order
3. Add Priority Order
```

4. Process Priority Order

5. Display All Orders

6. Exit

Enter choice: 1

Enter Order ID and Item: 101 Keyboard

Normal order #101 added.

Enter choice: 3

Enter Order ID, Item, and Priority: 201 iPhone 5

Priority order #201 added with priority 5.

Enter choice: 3

Enter Order ID, Item, and Priority: 202 Macbook 8

Priority order #202 added with priority 8.

Enter choice: 4

Processing priority order #202 (Macbook) with priority 8

Enter choice: 2

Processing normal order #101: Keyboard

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100

// -------------------- Linked List for Normal Orders --------------------
typedef struct Node {
    int orderId;
    char item[50];
    struct Node* next;
} Node;

Node* head = NULL;

void addNormalOrder(int id, char* item) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->orderId = id;
    strcpy(temp->item, item);
    temp->next = NULL;

    if (head == NULL)
        head = temp;
    else {
        Node* p = head;
        while (p->next) p = p->next;
        p->next = temp;
    }

    printf("Normal order #%d added.\n", id);
```

```
--- E-Commerce Order Processing System ---
1. Add Normal Order
2. Process Normal Order
3. Add Priority Order
4. Process Priority Order
5. Display All Orders
6. Exit
Enter choice: 1
Enter Order ID and Item: 101 Keyboard
Normal order #101 added.

--- E-Commerce Order Processing System ---
1. Add Normal Order
2. Process Normal Order
3. Add Priority Order
4. Process Priority Order
5. Display All Orders
6. Exit
Enter choice: 3
Enter Order ID, Item, and Priority: 201 iphone
Priority order #201 added with priority 5.

--- E-Commerce Order Processing System ---
1. Add Normal Order
2. Process Normal Order
3. Add Priority Order
4. Process Priority Order
5. Display All Orders
6. Exit
```

🧾 7. Conclusion

This project demonstrates how a simple yet effective **Order Processing System** can be implemented using core C data structures — **Linked List** and **Max Heap**. By using a linked list, we ensure fairness and simplicity for regular orders. The max heap brings efficiency by dynamically handling priority-based scheduling, a common necessity in e-commerce systems today.

Although basic, this simulation mirrors how real backend systems prioritize VIP users, high-value items, or urgent shipments, ensuring that the most critical operations are handled first. Such systems are extendable, and additional layers like multi-threading, concurrency control, and database integration can further bring this simulation closer to real-life applications.

This project not only strengthens the understanding of queues and heaps but also offers a glimpse into real-world applications of data structures beyond the academic realm.