Program 6: Write a Program for finding the Product of the three largest Distinct Elements. Use a Priority Queue to efficiently find and remove the largest elements

A priority queue is a special type of queue where each element has a priority assigned to it. In a priority queue:

- High-priority elements are served before low-priority ones.
- If two elements have the same priority, they are served based on their order in the queue.

Key Points:

- It can be implemented using a heap (most commonly a max or min heap).
- Example: In a hospital emergency room, critical patients (high priority) are treated before others, regardless of when they arrived.

Uses of Priority Queues:

1. Scheduling tasks (e.g., CPU task scheduling).
2. Pathfinding algorithms (e.g., Dijkstra's algorithm).
3. Data compression (e.g., Huffman coding).
4. Handling real-time systems where tasks must be prioritized.

Simple analogy: Think of a priority queue like a VIP line at an event — VIPs (high priority) get served first!

Relation Between Max Heap and Priority Queue:

- A **Max Heap** is a **common implementation** of a **Priority Queue**:
    - The **highest-priority element** in a **Max Priority Queue** is the largest value, and it is stored at the root of the Max Heap.
    - Operations like **insert** and **extract max** in a Max Priority Queue are implemented using **heap operations** (heapify-up for insertion, heapify-down for extraction).

```c
#include <stdio.h>

#include <stdlib.h>


#define MAX_SIZE 100


// Structure for the Priority Queue (Max Heap)

typedef struct {

    int heap[MAX_SIZE];

    int size;

} PriorityQueue;


// Function to initialize the priority queue

void initialize(PriorityQueue *pq) {

    pq->size = 0;

}


// Function to swap two elements

void swap(int *a, int *b) {

    int temp = *a;

    *a = *b;

    *b = temp;

}


// Function to heapify-up (for insertion)

void heapifyUp(PriorityQueue *pq, int index) {

    int parent = (index - 1) / 2;

    while (index > 0 && pq->heap[parent] < pq->heap[index]) {

        swap(&pq->heap[parent], &pq->heap[index]);

        index = parent;
```

```
        parent = (index - 1) / 2;
    }
}

// Function to heapify-down (for extraction)
void heapifyDown(PriorityQueue *pq, int index) {
    int largest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < pq->size && pq->heap[left] > pq->heap[largest]) {
        largest = left;
    }
    if (right < pq->size && pq->heap[right] > pq->heap[largest]) {
        largest = right;
    }
    if (largest != index) {
        swap(&pq->heap[index], &pq->heap[largest]);
        heapifyDown(pq, largest);
    }
}

// Function to insert an element into the priority queue
void insert(PriorityQueue *pq, int value) {
    if (pq->size == MAX_SIZE) {
        printf("Priority Queue is full.\n");
        return;
    }
    pq->heap[pq->size] = value;
```

```c
    heapifyUp(pq, pq->size);
    pq->size++;
}

// Function to extract the maximum element from the priority queue
int extractMax(PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue is empty.\n");
        return -1;
    }
    int max = pq->heap[0];
    pq->heap[0] = pq->heap[pq->size - 1];
    pq->size--;
    heapifyDown(pq, 0);
    return max;
}

// Function to find the product of the three largest distinct elements
void findProductOfThreeLargest(int arr[], int n) {
    PriorityQueue pq;
    initialize(&pq);

    // Insert elements into the priority queue
    for (int i = 0; i < n; i++) {
        insert(&pq, arr[i]);
    }

    // Extract the three largest distinct elements
    int first = extractMax(&pq);  // Largest element
```

```
    int second, third;

    // Find the second distinct element
    while (pq.size > 0) {
        second = extractMax(&pq);
        if (second != first) break;  // Ensure distinctness
    }

    // Find the third distinct element
    while (pq.size > 0) {
        third = extractMax(&pq);
        if (third != first && third != second) break;  // Ensure distinctness
    }

    // Check if we found three distinct elements
    if (first == second || second == third) {
        printf("Not enough distinct elements to compute the product.\n");
    } else {
        int product = first * second * third;
        printf("The three largest distinct elements are: %d, %d, %d\n", first, second, third);
        printf("Their product is: %d\n", product);
    }
}

// Main function to demonstrate the program
int main() {
    int arr[] = {10, 20, 5, 15, 25, 10, 20};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```c
    findProductOfThreeLargest(arr, n);

    return 0;
}
```

OUTPUT: The three largest distinct elements are: 25, 20, 15

Their product is: 7500

7. Write a Program to Merge two linked lists(sorted)and destination

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



struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to print a linked list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
```

```
        head = head->next;
    }
    printf("NULL\n");
}

// Function to merge two sorted linked lists
struct Node* mergeSortedLists(struct Node* l1, struct Node* l2) {
    // Dummy node to help build the merged list
    struct Node dummy;
    struct Node* tail = &dummy;
    dummy.next = NULL;

    // Traverse both lists and merge them
    while (l1 != NULL && l2 != NULL) {
        if (l1->data < l2->data) {
            tail->next = l1;
            l1 = l1->next;
        } else {
            tail->next = l2;
            l2 = l2->next;
        }
        tail = tail->next;
    }

    // Append the remaining elements of l1 or l2
    if (l1 != NULL) {
        tail->next = l1;
    } else {
        tail->next = l2;
```

```c
    }

    return dummy.next; // Return the merged list
}

// Main function
int main() {
    // Creating the first sorted linked list: 1 -> 3 -> 5
    struct Node* l1 = createNode(1);
    l1->next = createNode(3);
    l1->next->next = createNode(5);

    // Creating the second sorted linked list: 2 -> 4 -> 6
    struct Node* l2 = createNode(2);
    l2->next = createNode(4);
    l2->next->next = createNode(6);

    printf("List 1: ");
    printList(l1);

    printf("List 2: ");
    printList(l2);

    // Merging the two sorted linked lists
    struct Node* mergedList = mergeSortedLists(l1, l2);

    printf("Merged List: ");
    printList(mergedList);
```

```c
    return 0;
}
```

Ouput:

List 1: 1 -> 3 -> 5 -> NULL

List 2: 2 -> 4 -> 6 -> NULL

Merged List: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> NULL

8. Write a Program to find the Merge point of two linked lists(sorted)

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

struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to find the length of a linked list
int getLength(struct Node* head) {
    int length = 0;
    while (head != NULL) {
```

```c
        length++;
        head = head->next;
    }
    return length;
}

// Function to find the merge point of two linked lists
struct Node* findMergePoint(struct Node* head1, struct Node* head2) {
    int length1 = getLength(head1);
    int length2 = getLength(head2);

    int diff = abs(length1 - length2);

    // Move the pointer of the longer list by `diff` steps
    if (length1 > length2) {
        for (int i = 0; i < diff; i++)
            head1 = head1->next;
    } else {
        for (int i = 0; i < diff; i++)
            head2 = head2->next;
    }

    // Traverse both lists simultaneously to find the merge point
    while (head1 != NULL && head2 != NULL) {
        if (head1 == head2) // Merge point found
            return head1;
        head1 = head1->next;
        head2 = head2->next;
    }
```

```c
    return NULL; // No merge point
}

// Function to print a linked list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// Main function to test the program
int main() {
    // Create two linked lists
    struct Node* head1 = createNode(1);
    head1->next = createNode(2);
    head1->next->next = createNode(3);

    struct Node* head2 = createNode(4);
    head2->next = createNode(5);

    // Create a merge point (common node)
    struct Node* mergePoint = createNode(6);
    head1->next->next->next = mergePoint;
    head2->next->next = mergePoint;

    mergePoint->next = createNode(7);
```

```
    mergePoint->next->next = createNode(8);

    printf("List 1: ");
    printList(head1);

    printf("List 2: ");
    printList(head2);

    // Find the merge point
    struct Node* result = findMergePoint(head1, head2);
    if (result != NULL)
        printf("Merge point found at node with data: %d\n", result->data);
    else
        printf("No merge point found.\n");

    return 0;
}
```

Output:

```
 List 1: 1 -> 2 -> 3 -> 6 -> 7 -> 8 -> NULL
List 2: 4 -> 5 -> 6 -> 7 -> 8 -> NULL
Merge point found at node with data: 6
```

9. Write a Program to Swap Nodes pairwise

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

struct Node {
```

```
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to print a linked list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

// Function to swap nodes pairwise
void swapPairs(struct Node* head) {
    struct Node* temp = head;

    while (temp != NULL && temp->next != NULL) {
        // Swap the data of the current node and the next node
        int tempData = temp->data;
        temp->data = temp->next->data;
```

```
        temp->next->data = tempData;

        // Move to the next pair
        temp = temp->next->next;
    }
}

// Main function to test the program
int main() {
    // Create a linked list: 1 -> 2 -> 3 -> 4 -> 5 -> NULL
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    head->next->next->next = createNode(4);
    head->next->next->next->next = createNode(5);

    printf("Original List: ");
    printList(head);

    // Swap nodes pairwise
    swapPairs(head);

    printf("List After Pairwise Swap: ");
    printList(head);

    return 0;
}
```

Output: List After Pairwise Swap: 2 -> 1 -> 4 -> 3 -> 5 -> NULL

10. Write a Program to Understand and implement Tree traversals i.e. Pre-Order Post-Order, In-Order

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

// Definition of a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Pre-Order Traversal: Root -> Left -> Right
void preOrder(struct Node* root) {
    if (root == NULL)
        return;
    printf("%d ", root->data);
    preOrder(root->left);
    preOrder(root->right);
}
```

```
// In-Order Traversal: Left -> Root -> Right
void inOrder(struct Node* root) {
    if (root == NULL)
        return;
    inOrder(root->left);
    printf("%d ", root->data);
    inOrder(root->right);
}

// Post-Order Traversal: Left -> Right -> Root
void postOrder(struct Node* root) {
    if (root == NULL)
        return;
    postOrder(root->left);
    postOrder(root->right);
    printf("%d ", root->data);
}

// Main function to demonstrate the tree traversals
int main() {
    // Create a simple binary tree
    //         1
    //        / \
    //       2   3
    //      / \ / \
    //     4  5 6  7

    struct Node* root = createNode(1);
    root->left = createNode(2);
```

```
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);

    printf("Pre-Order Traversal: ");
    preOrder(root);
    printf("\n");

    printf("In-Order Traversal: ");
    inOrder(root);
    printf("\n");

    printf("Post-Order Traversal: ");
    postOrder(root);
    printf("\n");

    return 0;
}
```

Output:

```
      1
    /   \
   2     3
  / \   / \
 4   5 6   7
```

Pre-Order Traversal: 1 2 4 5 3 6 7

In-Order Traversal: 4 2 5 1 6 3 7

Post-Order Traversal: 4 5 2 6 7 3 1