

simple queue

using array

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

// Function prototypes
int enqueue(int element, int *queueArray, int *frontIndex, int *rearIndex,
int maxSize);
int dequeue(int *queueArray, int *frontIndex, int *rearIndex);
void displayQueue(int *queueArray, int frontIndex, int rearIndex);

int main()
{
    int *queueArray; // Pointer for dynamically allocated queue array
    int choice, result, element;
    int frontIndex, rearIndex, maxSize;

    frontIndex = rearIndex = -1; // Initialize front and rear indices to -1
    (empty queue)

    printf("Enter the size of the queue: ");
    scanf("%d", &maxSize);

    queueArray = (int *)malloc(sizeof(int) * maxSize); // Allocate memory
    for the queue array

    while (1)
    {
        displayQueue(queueArray, frontIndex, rearIndex); // Display current
        queue contents

        printf("\n1..Insert (Enqueue)");
        printf("\n2..Delete (Dequeue)");
        printf("\n3..Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
```

```

switch (choice)
{
    case 1:
        printf("Enter the value to insert: ");
        scanf("%d", &element);
        result = enqueue(element, queueArray, &frontIndex,
&rearIndex, maxSize);
        if (result >= 0)
            printf("Element inserted successfully\n");
        break;

    case 2:
        result = dequeue(queueArray, &frontIndex, &rearIndex);
        if (result >= 0)
            printf("Element deleted = %d\n", result);
        break;

    case 3:
        exit(0);
    }
}

```

```

int enqueue(int element, int *queueArray, int *frontIndex, int *rearIndex,
int maxSize)
{
    // Check if the queue is full
    if (*rearIndex == maxSize - 1)
    {
        printf("Queue full. Cannot insert.\n");
        return -1;
    }

    (*rearIndex)++; // Increment rear index
    queueArray[*rearIndex] = element; // Insert the element at the rear

    if (*frontIndex == -1) // If this is the first element
        *frontIndex = 0; // Set front index to 0
}

```

```

        return 1;
    }

int dequeue(int *queueArray, int *frontIndex, int *rearIndex)
{
    int removedElement;

    // Check if the queue is empty
    if (*frontIndex == -1)
    {
        printf("Queue is empty. Cannot delete.\n");
        return -1;
    }

    removedElement = queueArray[*frontIndex]; // Store the element to be
removed

    if (*frontIndex == *rearIndex)           // If only one element in the
queue
    {
        *frontIndex = *rearIndex = -1; // Reset both front and rear indices
    }
    else
    {
        (*frontIndex)++; // Increment front index after deletion
    }

    return removedElement;
}

void displayQueue(int *queueArray, int frontIndex, int rearIndex)
{
    int i;

    // Check if the queue is empty
    if (frontIndex == -1)
    {
        printf("Queue is empty.\n");
    }
    else

```

```

    {
        printf("Current queue elements: ");
        for (i = frontIndex; i <= rearIndex; i++)
            printf("%d ", queueArray[i]);
        printf("\n");
    }
}

```

using linked list

```

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

typedef struct node {
    int value;           // Element value
    struct node *next;   // Pointer to the next node
} Node;

typedef struct queue {
    Node *front;         // Pointer to the front node
    Node *rear;          // Pointer to the rear node
} Queue;

// Function prototypes
void initQueue(Queue *queue);
void enqueue(Queue *queue, int element);
int dequeue(Queue *queue);
int isEmpty(Queue *queue);
void display(Queue *queue);
void destroyQueue(Queue *queue);

int main() {
    int choice, element;
    Queue queue;
    initQueue(&queue);

    printf("1. Enqueue 2. Dequeue 3. Display 4. Exit\n");
    scanf("%d", &choice);

```

```

do {
    switch (choice) {
        case 1:
            printf("Enter an element: ");
            scanf("%d", &element);
            enqueue(&queue, element);
            break;
        case 2:
            if (!isEmpty(&queue))
                printf("Deleted element is %d\n", dequeue(&queue));
            else
                printf("Queue is already empty\n");
            break;
        case 3:
            display(&queue);
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }

    printf("1. Enqueue 2. Dequeue 3. Display 4. Exit\n");
    scanf("%d", &choice);
} while (choice < 4);

destroyQueue(&queue);
return 0;
}

void initQueue(Queue *queue) {
    queue->front = queue->rear = NULL;
}

void enqueue(Queue *queue, int element) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed!\n");
        return;
    }
}

```

```

    }

    newNode->value = element;
    newNode->next = NULL;

    if (queue->rear == NULL) {
        queue->rear = newNode;
        queue->front = newNode; // First element added
    } else {
        queue->rear->next = newNode; // Link new node at rear
        queue->rear = newNode;      // Update rear to new node
    }
}

int dequeue(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return -1; // Indicating an error
    }

    Node *tempNode = queue->front; // Node to be removed
    int removedValue = tempNode->value;
    queue->front = queue->front->next; // Move front to next node

    if (queue->front == NULL) {
        queue->rear = NULL; // If the queue is now empty
    }

    free(tempNode);
    return removedValue; // Return the value of the removed node
}

int isEmpty(Queue *queue) {
    return queue->front == NULL; // Check if front is NULL
}

void display(Queue *queue) {
    Node *currentNode = queue->front;
    if (currentNode == NULL) {
        printf("Queue is empty\n");
    }
}

```

```

        return;
    }

    printf("Queue elements: ");
    while (currentNode != NULL) {
        printf("%d ", currentNode->value);
        currentNode = currentNode->next; // Move to next node
    }
    printf("\n");
}

void destroyQueue(Queue *queue) {
    Node *currentNode = queue->front;
    while (currentNode != NULL) {
        Node *tempNode = currentNode; // Store current node
        currentNode = currentNode->next; // Move to next node
        free(tempNode); // Free the stored node
    }
    queue->rear = queue->front = NULL; // Reset the queue
}

```

using array of structures

```

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

struct queue {
    int *queueArray;    // Dynamic array to store queue elements
    int frontIndex;     // Index of the front element
    int rearIndex;      // Index of the rear element
    int maxSize;        // Maximum size of the queue
};

typedef struct queue queue_t;

int enqueue(queue_t *, int);
int dequeue(queue_t *);
void displayQueue(queue_t *);
void initializeQueue(queue_t *);

```

```

int main() {
    queue_t q;
    int choice, operationResult, value;

    initializeQueue(&q);

    while(1) {
        printf("\n\n1. Insert (Enqueue)");
        printf("\n2. Delete (Dequeue)");
        printf("\n3. Display");
        printf("\n4. EXIT\n\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                printf("\nEnter the value: ");
                scanf("%d", &value);
                operationResult = enqueue(&q, value);
                if (operationResult >= 0)
                    printf("Element inserted successfully\n");
                break;
            case 2:
                operationResult = dequeue(&q);
                if (operationResult >= 0)
                    printf("Element deleted = %d\n", operationResult);
                break;
            case 3:
                displayQueue(&q);
                break;
            case 4:
                exit(0);
        }
    }
}

```

```

void initializeQueue(queue_t *queuePtr) {
    printf("Enter the maximum size of the queue: ");
    scanf("%d", &queuePtr->maxSize);
}

```



```

    queuePtr->queueArray = (int*)malloc(sizeof(int) * queuePtr->maxSize);
    queuePtr->frontIndex = queuePtr->rearIndex = -1;
}

int enqueue(queue_t *queuePtr, int data) {
    // Check if the queue is full
    if (queuePtr->rearIndex == queuePtr->maxSize - 1) {
        printf("\nQueue is full...cannot insert\n");
        return -1;
    }

    // Increment rearIndex and insert data
    queuePtr->rearIndex++;
    queuePtr->queueArray[queuePtr->rearIndex] = data;

    // Set frontIndex to 0 if inserting the first element
    if (queuePtr->frontIndex == -1)
        queuePtr->frontIndex = 0;

    return 1;
}

int dequeue(queue_t *queuePtr) {
    int removedElement;

    // Check if the queue is empty
    if (queuePtr->frontIndex == -1) {
        printf("\nQueue is empty...cannot delete\n");
        return -1;
    }

    // Retrieve the front element
    removedElement = queuePtr->queueArray[queuePtr->frontIndex];

    // If there is only one element, reset both frontIndex and rearIndex
    if (queuePtr->frontIndex == queuePtr->rearIndex)
        queuePtr->frontIndex = queuePtr->rearIndex = -1;
    else
        queuePtr->frontIndex++;
}

```

```

        return removedElement;
    }

void displayQueue(queue_t *queuePtr) {
    int i;

    if (queuePtr->frontIndex == -1)
        printf("\nQueue is empty...\n");
    else {
        printf("\nElements in the queue:\n");
        for (i = queuePtr->frontIndex; i <= queuePtr->rearIndex; i++)
            printf("%d\t", queuePtr->queueArray[i]);
        printf("\n");
    }
}
}

```

circular queue

using array

```

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

int enqueue(int*, int*, int*, int, int); // Insert element in the queue
int dequeue(int *, int *, int*, int);    // Remove element from the queue
void displayQueue(int *, int, int, int); // Display elements of the queue

int main() {
    int *queueArray; // Dynamic array for queue
    int choice, operationResult, value;
    int frontIndex, rearIndex, maxSize;

    frontIndex = rearIndex = -1;

    printf("Enter the size of the queue: ");
    scanf("%d", &maxSize);

    queueArray = (int*)malloc(sizeof(int) * maxSize); // Allocate memory
    for queue

```

```

while(1) {
    printf("\n1. Insert (Enqueue)");
    printf("\n2. Delete (Dequeue)");
    printf("\n3. Display");
    printf("\n4. EXIT");
    printf("\nEnter your choice: ");
    scanf("%d", &choice);

    switch(choice) {
        case 1:
            printf("Enter the value to insert: ");
            scanf("%d", &value);
            operationResult = enqueue(queueArray, &frontIndex,
&rearIndex, maxSize, value);
            if (operationResult >= 0)
                printf("Element inserted successfully\n");
            break;
        case 2:
            operationResult = dequeue(queueArray, &frontIndex,
&rearIndex, maxSize);
            if (operationResult >= 0)
                printf("Element deleted = %d\n", operationResult);
            break;
        case 3:
            displayQueue(queueArray, frontIndex, rearIndex, maxSize);
            break;
        case 4:
            exit(0);
    }
}

```

```

int enqueue(int *queueArray, int *frontIndex, int *rearIndex, int maxSize,
int value) {
    // Check if the queue is full
    if ((*rearIndex + 1) % maxSize == *frontIndex) {
        printf("Queue is full... cannot insert\n");
        return -1;
    }
}

```

```

// Increment rearIndex and insert value
*rearIndex = (*rearIndex + 1) % maxSize;
queueArray[*rearIndex] = value;

// Set frontIndex to 0 if inserting the first element
if (*frontIndex == -1)
    *frontIndex = 0;

return 1;
}

int dequeue(int *queueArray, int *frontIndex, int *rearIndex, int maxSize) {
    int removedValue;

    // Check if the queue is empty
    if (*frontIndex == -1) {
        printf("Queue is empty... cannot delete\n");
        return -1;
    }

    // Retrieve the front element
    removedValue = queueArray[*frontIndex];

    // If there is only one element, reset both frontIndex and rearIndex
    if (*frontIndex == *rearIndex)
        *frontIndex = *rearIndex = -1;
    else
        *frontIndex = (*frontIndex + 1) % maxSize;

    return removedValue;
}

void displayQueue(int *queueArray, int frontIndex, int rearIndex, int
maxSize) {
    if (frontIndex == -1) {
        printf("Queue is empty...\n");
        return;
    }

```

```

    printf("Elements in the queue: ");
    while (frontIndex != rearIndex) {
        printf("%d ", queueArray[frontIndex]);
        frontIndex = (frontIndex + 1) % maxSize;
    }
    printf("%d\n", queueArray[rearIndex]);
}

```

using array of structures

```

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

struct queue {
    int *circularQueue; // Dynamic array to store queue elements
    int frontIndex, rearIndex, maxSize; // Indices for front, rear, and
    maximum size of the queue
};

typedef struct queue queue_t;

void initQueue(queue_t*);
int enqueue(queue_t*, int);
int dequeue(queue_t*);
void displayQueue(queue_t*);
int isEmpty(queue_t*);

int main() {
    int choice, result, element;
    queue_t q;
    initQueue(&q);

    while (1) {
        printf("\n1. Enqueue (Insert)");
        printf("\n2. Dequeue (Delete)");
        printf("\n3. Display");
        printf("\n4. EXIT");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
    }
}

```

```

switch (choice) {
    case 1:
        printf("Enter the value to insert: ");
        scanf("%d", &element);
        result = enqueue(&q, element);
        if (result >= 0)
            printf("Element inserted successfully\n");
        break;

    case 2:
        result = dequeue(&q);
        if (result >= 0)
            printf("Element deleted = %d\n", result);
        break;

    case 3:
        displayQueue(&q);
        break;

    case 4:
        exit(0);
    }
}

```

```

int isQueueEmpty(queue_t *queuePtr) {
    if (queuePtr->frontIndex == -1)
        return 1;
    return 0;
}

```

```

void initQueue(queue_t *queuePtr) {
    printf("Enter the size of the queue: ");
    scanf("%d", &queuePtr->maxSize);
    queuePtr->frontIndex = queuePtr->rearIndex = -1;
    queuePtr->circularQueue = (int*)malloc(sizeof(int) * (queuePtr->maxSize));
}

```

```

int enqueue(queue_t *queuePtr, int element) {
    // Check if the queue is full
    if ((queuePtr->rearIndex + 1) % queuePtr->maxSize == queuePtr->frontIndex) {
        printf("Queue is full. Cannot insert.\n");
        return -1;
    }
    // Insert element into the queue
    if (queuePtr->frontIndex == -1) {
        queuePtr->frontIndex = 0;
    }
    queuePtr->rearIndex = (queuePtr->rearIndex + 1) % queuePtr->maxSize;
    queuePtr->circularQueue[queuePtr->rearIndex] = element;
    return 1;
}

```

```

int dequeue(queue_t *queuePtr) {
    int removedValue;

    // Check if the queue is empty
    if (queuePtr->frontIndex == -1) {
        printf("Queue is empty.\n");
        return -1;
    }

    // Remove the front element
    removedValue = queuePtr->circularQueue[queuePtr->frontIndex];

    // If there is only one element, reset both frontIndex and rearIndex
    if (queuePtr->frontIndex == queuePtr->rearIndex) {
        queuePtr->frontIndex = queuePtr->rearIndex = -1;
    } else {
        queuePtr->frontIndex = (queuePtr->frontIndex + 1) % queuePtr->maxSize;
    }

    return removedValue;
}

```

```

void displayQueue(queue_t *queuePtr) {

```

```

    if (queuePtr->frontIndex == -1) {
        printf("Queue is empty.\n");
        return;
    }

    int currentIndex = queuePtr->frontIndex;
    int rearIndex = queuePtr->rearIndex;
    int size = queuePtr->maxSize;

    printf("Queue elements: ");
    while (currentIndex != rearIndex) {
        printf("%d ", queuePtr->circularQueue[currentIndex]);
        currentIndex = (currentIndex + 1) % size;
    }
    printf("%d\n", queuePtr->circularQueue[rearIndex]);
}

```

using ll

```

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

typedef struct node {
    int value;          // Element value
    struct node *next; // Pointer to the next node
} Node;

typedef struct queue {
    Node *front; // Pointer to the front node
    Node *rear;  // Pointer to the rear node
} Queue;

void initQueue(Queue *queue);
void enqueue(Queue *queue, int element);
int dequeue(Queue *queue);
int isEmpty(Queue *queue);
void display(Queue *queue);
void destroyQueue(Queue *queue);

```



```
int main() {
    int choice, element;
    Queue queue;
    initQueue(&queue);

    printf("1. Enqueue 2. Dequeue 3. Display 4. Exit\n");
    scanf("%d", &choice);

    do {
        switch (choice) {
            case 1:
                printf("Enter an element: ");
                scanf("%d", &element);
                enqueue(&queue, element);
                break;

            case 2:
                if (!isEmpty(&queue))
                    printf("Deleted element is %d\n", dequeue(&queue));
                else
                    printf("Queue is already empty\n");
                break;

            case 3:
                display(&queue);
                break;

            case 4:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }

        printf("1. Enqueue 2. Dequeue 3. Display 4. Exit\n");
        scanf("%d", &choice);
    } while (choice < 4);

    destroyQueue(&queue);
}
```

```

        return 0;
    }

void initQueue(Queue *queue) {
    queue->front = queue->rear = NULL;
}

void enqueue(Queue *queue, int element) {
    Node *newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed!\n");
        return;
    }

    newNode->value = element;

    if (queue->rear == NULL) {
        queue->rear = newNode;
        queue->front = newNode;
        newNode->next = newNode; // Circular link
    } else {
        newNode->next = queue->front; // New node points to front
        queue->rear->next = newNode; // Rear's next points to new node
        queue->rear = newNode;      // Update rear to new node
    }
}

int dequeue(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return -1; // Indicating an error
    }

    Node *tempNode = queue->front;
    int removedValue = tempNode->value;

    if (queue->front == queue->rear) { // Only one node
        queue->front = queue->rear = NULL;
    } else {
        queue->rear->next = queue->front->next; // Update rear's next
    }
}

```

```

        queue->front = queue->front->next;        // Move front to next
    }

    free(tempNode);
    return removedValue;
}

int isEmpty(Queue *queue) {
    return queue->front == NULL;
}

void display(Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return;
    }

    Node *currentNode = queue->front;
    do {
        printf("%d ", currentNode->value);
        currentNode = currentNode->next;
    } while (currentNode != queue->front);

    printf("\n");
}

void destroyQueue(Queue *queue) {
    while (!isEmpty(queue)) {
        dequeue(queue); // Dequeue until empty
    }
}

```

priority queue

using array

```

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

```

```

// Program to implement a priority queue using an array

// Structure to represent an element in the priority queue
struct element {
    int priority; // Priority of the element (lower values indicate higher
priority)
    int value;    // Value of the element
};

typedef struct element element_t;

// Function prototypes
void pq_insert(element_t *queue, int value, int priority, int *count);
element_t pq_delete(element_t *queue, int *count);
void pq_display(element_t *queue, int count);

int main() {
    int choice, priority, value;
    element_t priority_queue[100], deleted_element;
    int queue_count = 0; // Keeps track of the number of elements in the
queue

    while (1) {
        // Display menu options
        printf("\n1. Insert");
        printf("\n2. Remove");
        printf("\n3. Display");
        printf("\n4. EXIT");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Insert an element into the priority queue
                printf("Enter the priority: ");
                scanf("%d", &priority);
                printf("Enter the value: ");
                scanf("%d", &value);
                pq_insert(priority_queue, value, priority, &queue_count);
                break;

```

```

        case 2:
            // Remove the highest-priority element
            deleted_element = pq_delete(priority_queue, &queue_count);
            if (deleted_element.value > 0) {
                printf("Deleted element: Value = %d, Priority = %d\n",
deleted_element.value, deleted_element.priority);
            }
            break;

        case 3:
            // Display the elements in the priority queue
            pq_display(priority_queue, queue_count);
            break;

        case 4:
            // Exit the program
            printf("Exiting...\n");
            exit(0);

        default:
            printf("Invalid choice. Please try again.\n");
    }
}

return 0;
}

```

```

// Function to insert an element into the priority queue
void pq_insert(element_t *queue, int value, int priority, int *count) {
    int index;
    element_t new_element;

    // Initialize the new element
    new_element.value = value;
    new_element.priority = priority;

    // Find the correct position to insert the new element based on priority
    index = *count - 1;
    while ((index >= 0) && (queue[index].priority > new_element.priority)) {

```

```

        queue[index + 1] = queue[index]; // Shift elements with lower
priority
        index--;
    }

    // Insert the new element at the correct position
    queue[index + 1] = new_element;

    // Increment the count of elements in the queue
    (*count)++;
}

// Function to delete the highest-priority element from the priority queue
element_t pq_delete(element_t *queue, int *count) {
    int i;
    element_t removed_element;

    // Check if the queue is empty
    if ((*count) == 0) {
        printf("\nEmpty Queue.\n");
        removed_element.value = -1; // Indicate error
        removed_element.priority = -1;
    } else {
        // Remove the element with the highest priority (first element)
        removed_element = queue[0];

        // Shift the remaining elements to the left
        for (i = 1; i < *count; i++) {
            queue[i - 1] = queue[i];
        }

        // Decrement the count of elements in the queue
        (*count)--;
    }

    return removed_element;
}

// Function to display the elements in the priority queue
void pq_display(element_t *queue, int count) {

```

```

    int i;

    // Check if the queue is empty
    if (count == 0) {
        printf("\nEmpty Queue.\n");
    } else {
        // Display each element in the queue along with its priority
        printf("\nPriority Queue:");
        for (i = 0; i < count; i++) {
            printf("\nValue = %d, Priority = %d", queue[i].value,
queue[i].priority);
        }
        printf("\n");
    }
}

```

using array of structures

```

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

// Define the structure for a node in the priority queue
typedef struct node {
    int data;           // The value stored in the node
    int priority;       // Priority of the node (higher value indicates
higher priority)
    struct node *next; // Pointer to the next node in the queue
} NODE;

// Define the structure for the priority queue
typedef struct queue {
    NODE *rear; // Pointer to the rear (lowest priority) node
    NODE *front; // Pointer to the front (highest priority) node
} QUEUE;

// Function to initialize a new priority queue
QUEUE* init(QUEUE *queue) {
    queue = (QUEUE *)malloc(sizeof(QUEUE)); // Allocate memory for the queue
    queue->front = NULL; // Initialize front as NULL (empty queue)
}

```

```

    queue->rear = NULL;    // Initialize rear as NULL (empty queue)
    return queue;
}

// Function to check if the priority queue is empty
int isEmpty(Queue *queue) {
    return queue->front == NULL; // Return 1 if the queue is empty, 0
    otherwise
}

// Function to check if the priority queue has exactly one element
int isOneElement(Queue *queue) {
    return (queue->rear == queue->front) && (queue->front != NULL); //
    Return 1 if only one element exists
}

// Function to create a new node with given data and priority
Node* createNode(int data, int priority) {
    Node *newNode = (Node *)malloc(sizeof(Node)); // Allocate memory for the
    new node
    newNode->data = data;           // Assign the data
    newNode->priority = priority; // Assign the priority
    newNode->next = NULL;          // Initialize the next pointer to NULL
    return newNode;
}

// Function to insert an element into the priority queue
void enqueue(Queue *queue, int data, int priority) {
    Node *newNode;           // New node to be added
    Node *current = queue->front; // Pointer to traverse the queue
    Node *previous = NULL; // Pointer to keep track of the previous node

    newNode = createNode(data, priority); // Create the new node

    // Case 1: Queue is empty
    if (isEmpty(queue)) {
        queue->front = newNode; // New node becomes both the front and rear
        queue->rear = newNode;
    }

    // Case 2: New node has higher or equal priority than the front

```



```

else if (queue->front->priority <= priority) {
    NODE *temp = queue->front; // Temporarily store the current front
    queue->front = newNode;    // New node becomes the new front
    newNode->next = temp;      // Point the new front to the old front
}
// Case 3: New node has lower priority than the rear
else if (queue->rear->priority >= priority) {
    NODE *temp = queue->rear; // Temporarily store the current rear
    queue->rear = newNode;    // New node becomes the new rear
    newNode->next = NULL;     // Rear always points to NULL
    temp->next = newNode;     // Link the old rear to the new rear
}
// Case 4: New node has priority between two existing nodes
else {
    while (current->next != NULL && current->priority >= priority) {
        previous = current; // Move the previous pointer
        current = current->next; // Move to the next node
    }
    // Insert the new node in the correct position
    previous->next = newNode;
    newNode->next = current;
}
}

// Function to display the elements in the priority queue
void display(Queue *queue) {
    NODE *current = queue->front; // Start from the front of the queue

    // If the queue is not empty, traverse and print all elements
    if (!isEmpty(queue)) {
        printf("\nPriority Queue Elements (Value, Priority):\n");
        while (current->next != NULL) {
            printf("%d %d\n", current->data, current->priority);
            current = current->next;
        }
        // Print the last node
        printf("%d %d\n", current->data, current->priority);
    } else {
        printf("\nThe queue is empty.\n");
    }
}

```

```

}

// Main function to demonstrate the priority queue operations
int main() {
    QUEUE *priorityQueue; // Pointer to the priority queue
    priorityQueue = init(priorityQueue); // Initialize the queue

    // Enqueue some elements with data and priority
    enqueue(priorityQueue, 20, 1);
    enqueue(priorityQueue, 30, 5);
    enqueue(priorityQueue, 40, 6);

    // Display the elements in the queue
    display(priorityQueue);

    return 0;
}

```

using II

```

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

#define SIZE 5 // Maximum size of the queue

// Define the structure for a node in the priority queue
typedef struct node {
    int data;        // Data stored in the node
    int priority;    // Priority of the node (higher value = higher priority)
} NODE;

// Define the structure for the priority queue
typedef struct queue {
    int front;       // Index of the front element
    int rear;       // Index of the rear element
    NODE array[SIZE]; // Array to store the queue elements
} QUEUE;

// Function to initialize a priority queue

```

```

QUEUE* init(QUEUE *queue) {
    queue = (QUEUE *)malloc(sizeof(QUEUE)); // Allocate memory for the queue
    queue->front = -1; // Initialize front index to -1 (empty queue)
    queue->rear = -1; // Initialize rear index to -1 (empty queue)
    return queue;
}

// Function to check if the queue is empty
int isEmpty(QUEUE* queue) {
    return (queue->front == -1 && queue->rear == -1); // True if both
indices are -1
}

// Function to check if the queue is full
int isFull(QUEUE* queue) {
    return (queue->rear >= SIZE - 1); // True if rear index reaches max size
}

// Function to check if the queue has exactly one element
int isOneElement(QUEUE* queue) {
    return (queue->front == queue->rear && queue->front != -1); // True if
front and rear are equal and not -1
}

// Function to add an element to the priority queue
void enqueue(QUEUE *queue, int data, int priority) {
    int insertionIndex = queue->front; // Index to find the insertion point
    int shiftingIndex = queue->rear; // Index to shift elements for
insertion

    if (isFull(queue)) {
        printf("\nThe queue is full.");
        return;
    }

    // Case 1: Queue is empty
    if (isEmpty(queue)) {
        queue->front = 0; // Update front index
        queue->rear = 0; // Update rear index
        queue->array[queue->rear].data = data;
    }
}

```

```

        queue->array[queue->rear].priority = priority;
    }
    // Case 2: Insert based on priority
    else {
        // Find the position where the new element should be inserted
        while (queue->array[insertionIndex].priority >= priority &&
insertionIndex <= queue->rear) {
            insertionIndex++;
        }

        // Shift all elements with lower priority one position to the right
        while (shiftingIndex >= insertionIndex) {
            queue->array[shiftingIndex + 1] = queue->array[shiftingIndex];
            shiftingIndex--;
        }

        // Insert the new element at the correct position
        queue->array[insertionIndex].data = data;
        queue->array[insertionIndex].priority = priority;
        queue->rear++; // Increment rear index
    }
}

// Function to display the elements in the priority queue
void display(Queue *queue) {
    if (!isEmpty(queue)) {
        printf("\nPriority Queue Elements (Index: Data, Priority):");
        for (int i = queue->front; i <= queue->rear; i++) {
            printf("\n[%d] %d, %d", i, queue->array[i].data, queue-
>array[i].priority);
        }
        printf("\n*****");
    } else {
        printf("\nThe queue is empty.");
    }
}

// Function to remove the front element from the priority queue
void dequeue(Queue *queue) {
    if (isEmpty(queue)) {

```

```

        printf("\nThe queue is empty.");
        return;
    }

    // Case 1: Only one element in the queue
    if (isOneElement(queue)) {
        queue->front = -1; // Reset front index
        queue->rear = -1;  // Reset rear index
    }
    // Case 2: Remove the front element
    else {
        queue->front++; // Move front index to the next element
    }
}

// Main function to demonstrate priority queue operations
int main() {
    QUEUE *priorityQueue;
    priorityQueue = init(priorityQueue); // Initialize the priority queue

    // Insert elements into the queue
    enqueue(priorityQueue, 10, 2);
    display(priorityQueue);
    enqueue(priorityQueue, 15, 1);
    display(priorityQueue);
    enqueue(priorityQueue, 20, 4);
    display(priorityQueue);
    enqueue(priorityQueue, 30, 3);
    display(priorityQueue);

    // Remove elements from the queue
    dequeue(priorityQueue);
    display(priorityQueue);
    dequeue(priorityQueue);
    display(priorityQueue);
    dequeue(priorityQueue);
    display(priorityQueue);
    dequeue(priorityQueue);
    display(priorityQueue);
}

```

```
    return 0;
}
```

deque (double linked queue)

linkedList

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

// Define a node structure for the doubly linked list
struct node {
    int value;                // Stores the value of the node
    struct node *next;        // Pointer to the next node
    struct node *previous;    // Pointer to the previous node
};

typedef struct node node_t;

// Define the structure for the double-ended queue (deque)
struct deque {
    node_t *front; // Points to the first node in the deque
    node_t *rear;  // Points to the last node in the deque
};

typedef struct deque deque_t;

// Function prototypes
void qdisplay(dequeue_t *deque);           // Display all elements in the deque
void qinsert_head(dequeue_t *deque, int value); // Insert an element at the front of the deque
void qinsert_tail(dequeue_t *deque, int value); // Insert an element at the rear of the deque
int qdelete_head(dequeue_t *deque);         // Remove and return the element from the front
int qdelete_tail(dequeue_t *deque);         // Remove and return the element from the rear
void init(dequeue_t *deque);               // Initialize the deque to be
```

empty

```
int main() {
    int deleted_value, input_value, choice;
    dequeue_t deque;
    init(&deque); // Initialize the deque

    while (1) {
        // Menu for deque operations
        printf("\n1. Insert at Head");
        printf("\n2. Insert at Tail");
        printf("\n3. Delete from Head");
        printf("\n4. Delete from Tail");
        printf("\n5. Display Deque");
        printf("\n6. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to insert at head: ");
                scanf("%d", &input_value);
                qinsert_head(&deque, input_value);
                break;
            case 2:
                printf("Enter the value to insert at tail: ");
                scanf("%d", &input_value);
                qinsert_tail(&deque, input_value);
                break;
            case 3:
                deleted_value = qdelete_head(&deque);
                if (deleted_value >= 0)
                    printf("Element deleted from head = %d\n",
deleted_value);
                break;
            case 4:
                deleted_value = qdelete_tail(&deque);
                if (deleted_value >= 0)
                    printf("Element deleted from tail = %d\n",
deleted_value);
```

```

        break;
    case 5:
        qdisplay(&deque);
        break;
    case 6:
        exit(0); // Exit the program
    default:
        printf("Invalid choice. Please try again.\n");
    }
}

void init(dequeue_t *deque) {
    deque->front = deque->rear = NULL; // Both front and rear are initially
    NULL
}

// Delete a node from the rear of the deque
int qdelete_tail(dequeue_t *deque) {
    if (deque->rear == NULL) {
        printf("Deque is empty. Cannot delete.\n");
        return -1; // Indicates failure
    }

    node_t *current = deque->rear;
    int value = current->value; // Retrieve the value to return

    if (deque->front == deque->rear) {
        // Only one node in the deque
        deque->front = deque->rear = NULL;
    } else {
        // Update rear to the previous node and disconnect the last node
        deque->rear = current->previous;
        deque->rear->next = NULL;
    }
    free(current); // Deallocate memory
    return value;
}

// Delete a node from the front of the deque

```



```

int qdelete_head(dequeue_t *deque) {
    if (deque->front == NULL) {
        printf("Deque is empty. Cannot delete.\n");
        return -1; // Indicates failure
    }

    node_t *current = deque->front;
    int value = current->value; // Retrieve the value to return

    if (deque->front == deque->rear) {
        // Only one node in the deque
        deque->front = deque->rear = NULL;
    } else {
        // Update front to the next node and disconnect the first node
        deque->front = current->next;
        deque->front->previous = NULL;
    }
    free(current); // Deallocate memory
    return value;
}

```

```

// Insert a node at the rear of the deque
void qinsert_tail(dequeue_t *deque, int value) {
    node_t *new_node = (node_t *)malloc(sizeof(node_t));
    new_node->value = value;
    new_node->next = NULL;
    new_node->previous = NULL;

    if (deque->front == NULL) {
        // Deque is empty
        deque->front = deque->rear = new_node;
    } else {
        // Add the new node at the rear
        new_node->previous = deque->rear;
        deque->rear->next = new_node;
        deque->rear = new_node;
    }
}

```

```

// Insert a node at the front of the deque

```

```

void qinsert_head(dequeue_t *deque, int value) {
    node_t *new_node = (node_t *)malloc(sizeof(node_t));
    new_node->value = value;
    new_node->next = NULL;
    new_node->previous = NULL;

    if (deque->front == NULL) {
        // Deque is empty
        deque->front = deque->rear = new_node;
    } else {
        // Add the new node at the front
        new_node->next = deque->front;
        deque->front->previous = new_node;
        deque->front = new_node;
    }
}

// Display the contents of the deque
void qdisplay(dequeue_t *deque) {
    if (deque->front == NULL) {
        printf("Deque is empty.\n");
        return;
    }

    node_t *current = deque->front;
    printf("Deque contents: ");
    while (current != NULL) {
        printf("%d", current->value);
        if (current->next != NULL) {
            printf(" <-> ");
        }
        current = current->next;
    }
    printf("\n");
}

```

josephus problem

-
- Data structure used is a circular list where each node represents one soldier
 - To represent the removal of a soldier from the circle, a node is deleted from the circular list.
 - Finally one node remains on the list and the result is determined

Pseudo code of implementation using circular list

```
read(n)
read(name)
while(all the names are read)
{
    insert name on the circular list
    read(name)
}
while(there is more than one node on the list)
{
    count through n-1 nodes on the list
    print name in the nth node
    delete the nth node
}
print the name of the only node on the list
```

Code of implementation using circular list

```
int survivor(struct node **head, int n)
{
    // head is pointer to first node

    struct node *p, *q;
    int i;
    q = p = *head;
    while (p->next != p)
    {
        for (i = 0; i < n - 1; i++)
        {
            q = p;
            p = p->next;
        }
        q->next = p->next;
        printf("%d has been killed.\n", p->num);
        free(p);
        p = q->next;
    }
    *head = p;
    return (p->num);
}
```

Pseudo code of implementation using circular queue

Enter n

while(all the names are read)

{

 insert name into the queue

 read(name)

}

while(q has one element)

{

 dequeue n-1 names from the queue and enqueue it.

 dequeue the n^{th} name

 print the n^{th} name

}

dequeue the only name of the queue

print the name

cpu scheduling

1. First Come First Serve (FCFS) Scheduling

- **Simplest algorithm:** Schedules processes based on arrival time.
 - **Non-preemptive:** The first process to request CPU gets executed until completion.
 - **Drawbacks:** Can lead to **convoy effect** (long waiting times for shorter jobs behind longer ones).
-

2. Shortest Job First (SJF) Scheduling

- **Non-Preemptive:**
 - The process with the **shortest burst time** is selected first.
 - **Tiebreaker:** If two processes have the same burst time, FCFS is used to break the tie.
 - **Simple queue implementation:** Processes are added to the rear of the queue.
 - The process at the front is executed until completion.
- **Preemptive (Shortest Remaining Time First - SRTF):**
 - Jobs are added to the ready queue as they arrive.
 - If a new process has a **shorter burst time** than the current one, the current process is preempted and the shorter job is executed.

3. Longest Job First (LJF) Scheduling

- **Non-Preemptive:**
 - **Opposite of SJF:** The process with the **longest burst time** is selected first.
 - Non-preemptive, meaning once a process starts, it cannot be interrupted until it finishes.
 - **Preemptive:**
 - Similar to SRTF but prioritizes the process with the **largest remaining burst time**.
 - If a new process with a longer burst time arrives, it preempts the current process.
-


4. Round Robin (RR) Scheduling

- **Time-slice based:** Processes are kept in a queue and given **equal time quantum** to execute.
 - **Preemptive:** After one time quantum, if the process hasn't completed, it's preempted and moved to the rear of the queue.
 - **Efficient for time-sharing systems:** Ensures no process starves, but context switching overhead can be high.
 - **Fairness:** All processes are treated equally with no prioritization.
-

5. Priority-Based Scheduling

- **Non-Preemptive:**
 - Processes are scheduled based on their **priority number**.
 - Once a process is scheduled, it runs until completion regardless of other processes arriving.
 - **Preemptive:**
 - **At the time of process arrival**, its priority is compared with others in the ready queue and the one currently running.
 - If the new process has a **higher priority**, the current process is preempted, and the higher priority process is executed.
-

Key Comparisons:

- **Preemptive Algorithms:** SJF (Preemptive), Round Robin, Priority-Based (Preemptive) – these algorithms can interrupt a running process.
 - **Non-Preemptive Algorithms:** FCFS, SJF (Non-Preemptive), LJF (Non-Preemptive), Priority-Based (Non-Preemptive) – once a process starts execution, it runs until completion without interruption.
 - **Efficiency and Context Switching:** Preemptive algorithms generally have more **context switches**, leading to overhead but ensuring responsiveness.
- 

1. Arrival Time:

- The time at which a process arrives in the **ready queue**, waiting for CPU execution.

2. Burst Time:

- The total time required by a process to complete its execution on the CPU.

3. Preemptive:

- **Preemption** occurs when a running process is interrupted and moved back to the ready queue to allow another process to execute. This is usually based on priority or time quantum (in the case of Round Robin).

4. Non-Preemptive:

- Once a process starts execution, it continues until it finishes without being interrupted, even if a higher-priority process arrives in the queue.

5. Time Quantum:

- The fixed time slice assigned to each process in **Round Robin scheduling**. After the time quantum expires, the process is preempted if it hasn't finished and the CPU is given to the next process.

6. Convoy Effect:

- A situation in **FCFS scheduling** where a short job has to wait for a long job to finish, causing inefficient use of CPU and longer wait times for the short jobs.

7. Context Switch:

- The process of storing the state of a currently running process so that another process can be executed. This happens during **preemptive scheduling** and incurs overhead due to saving and restoring states.

8. Priority:

- A value assigned to each process indicating its importance or urgency. In **priority-based scheduling**, processes with higher priority get CPU time before those with lower priority.

9. Ready Queue:

- A data structure (usually a queue) where processes wait until the CPU is free to execute them.

10. Process Control Block (PCB):

- A data structure that stores information about a process, including its state, priority, burst

time, and other execution details.