Priority Queue using Array

Explanation

1. Enqueue Operation:
   - New elements are inserted at the end of the array.
   - After inserting, elements are sorted in descending order of priority. This ensures that the highest-priority element is always at the front of the queue.
2. Dequeue Operation:
   - Removes the element at the front (highest priority) by shifting all elements one position to the left.
3. Display Operation:
   - Displays the elements of the priority queue along with their priorities.

This code demonstrates the core operations of a priority queue using an array, maintaining efficient ordering for dequeueing based on priority.

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

#define MAX 100

// Structure to represent a priority queue
typedef struct {
    int data;
    int priority;
} Element;

// Priority Queue with array
Element priorityQueue[MAX];
int size = 0;

// Function to enqueue an element based on priority
void enqueue(int data, int priority) {
    if (size == MAX) {
        printf("Queue overflow\n");
        return;
    }

    // Insert the new element at the end
    priorityQueue[size].data = data;
    priorityQueue[size].priority = priority;
    size++;

    // Sort elements by priority (higher priority at the beginning)
    for (int i = size - 1; i > 0; i--) {
        if (priorityQueue[i].priority > priorityQueue[i - 1].priority) {
            Element temp = priorityQueue[i];
            priorityQueue[i] = priorityQueue[i - 1];
            priorityQueue[i - 1] = temp;
        } else {
            break;
        }
    }
}

// Function to dequeue the highest priority element
void dequeue() {
    if (size == 0) {
        printf("Queue underflow\n");
        return;
    }

    printf("Dequeued element: %d with priority: %d\n", priorityQueue[0].data,
priorityQueue[0].priority);

    // Shift elements to the left
    for (int i = 0; i < size - 1; i++) {
        priorityQueue[i] = priorityQueue[i + 1];
    }
    size--;
}

// Function to display the priority queue
```

```
void display() {
    if (size == 0) {
        printf("Queue is empty\n");
        return;
    }

    printf("Priority Queue:\n");
    for (int i = 0; i < size; i++) {
        printf("Data: %d | Priority: %d\n", priorityQueue[i].data,
priorityQueue[i].priority);
    }
}

int main() {
    // Sample operations
    enqueue(10, 1);
    enqueue(20, 3);
    enqueue(15, 2);
    display();

    dequeue();
    display();

    enqueue(30, 5);
    display();

    return 0;
}
```

Output:

```
Priority Queue:
Data: 20 | Priority: 3
Data: 15 | Priority: 2
Data: 10 | Priority: 1

Dequeued element: 20 with priority: 3
```

```
Priority Queue:
Data: 15 | Priority: 2
Data: 10 | Priority: 1

Priority Queue(After Enqueue):
Data: 30 | Priority: 5
Data: 15 | Priority: 2
Data: 10 | Priority: 1
```

```
//C program to Demonstrate Priority Queue
#include<stdio.h>
#include<limits.h>

#define MAX 100


// denotes where the last item in priority queue is
// initialized to -1 since no item is in queue
int idx = -1;

// pqVal holds data for each index item
// pqPriority holds priority for each index item
int pqVal[MAX];
int pqPriority[MAX];



int isEmpty ()
{
  return idx == -1;
}

int
isFull ()
{
  return idx == MAX - 1;
```

```
}

// enqueue just adds item to the end of the priority queue | O(1)
void enqueue (int data, int priority)
{
  if (!isFull ())
    {

      // Increase the index
      idx++;

      // Insert the element in priority queue
      pqVal[idx] = data;
      pqPriority[idx] = priority;
    }
}

// returns item with highest priority
// NOTE: Max Priority Queue High priority number means higher priority | O(N)
int peek ()
{
  // Note : Max Priority, so assigned min value as initial value
  int maxPriority = INT_MIN;
  int indexPos = -1;

  // Linear search for highest priority
  for (int i = 0; i <= idx; i++)
    {
      // If two items have same priority choose the one with
      // higher data value
      if (maxPriority == pqPriority[i] && indexPos > -1
              && pqVal[indexPos] < pqVal[i])
            {
              maxPriority = pqPriority[i];
              indexPos = i;
            }

      // note: using MAX Priority so higher priority number
      // means higher priority
      else if (maxPriority < pqPriority[i])
            {
              maxPriority = pqPriority[i];
              indexPos = i;
            }
    }
  // Return index of the element where
  return indexPos;
}

// This removes the element with highest priority
// from the priority queue | O(N)
void dequeue ()
{
  if (!isEmpty ())
    {
      // Get element with highest priority
      int indexPos = peek ();

      // reduce size of priority queue by first
      // shifting all elements one position left
      // from index where the highest priority item was found
      for (int i = indexPos; i < idx; i++)
            {
              pqVal[i] = pqVal[i + 1];
              pqPriority[i] = pqPriority[i + 1];
            }

      // reduce size of priority queue by 1
      idx--;
    }
}

void display ()
{
  for (int i = 0; i <= idx; i++)
    {
      printf ("(%d, %d)\n", pqVal[i], pqPriority[i]);
    }
}

// Driver Code
int main ()
{
  // To enqueue items as per priority
  enqueue (5, 1);
  enqueue (10, 3);
  enqueue (15, 4);
  enqueue (20, 5);
  enqueue (500, 2);

  printf ("Before Dequeue : \n");
  display ();

  // Dequeue the top element
  dequeue ();                          // 20 dequeued
  dequeue ();                          // 15 dequeued

  printf ("\nAfter Dequeue : \n");
  display ();

  return 0;
}
```

Output

```
Before Dequeue :
(5, 1)
(10, 3)
(15, 4)
(20, 5)
(500, 2)

After Dequeue :
(5, 1)
(10, 3)
(500, 2)
```