Write a c program code for priority queue.

Code:

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

// Define the structure for a priority queue
typedef struct {
    int *arr;      // Array to store heap elements
    int size;      // Current number of elements
    int capacity;  // Maximum capacity of the heap
} PriorityQueue;

// Function prototypes
PriorityQueue* createPriorityQueue(int capacity);
void insert(PriorityQueue *pq, int value);
int extractMax(PriorityQueue *pq);
void maxHeapify(PriorityQueue *pq, int index);
void swap(int *a, int *b);
void printPriorityQueue(PriorityQueue *pq);
void freePriorityQueue(PriorityQueue *pq);

int main() {
    PriorityQueue *pq = createPriorityQueue(10);

    // Insert some elements into the priority queue
    insert(pq, 10);
    insert(pq, 20);
    insert(pq, 15);
    insert(pq, 30);
    insert(pq, 40);

    printf("Priority Queue elements:\n");
    printPriorityQueue(pq);

    // Extract elements from the priority queue
    printf("\nExtracted max: %d\n", extractMax(pq));
    printf("Priority Queue elements after extraction:\n");
    printPriorityQueue(pq);

    // Clean up
    freePriorityQueue(pq);

    return 0;
}
```

```
// Create a priority queue with the given capacity
PriorityQueue* createPriorityQueue(int capacity) {
   PriorityQueue *pq = (PriorityQueue *)malloc(sizeof(PriorityQueue));
   pq->capacity = capacity;
   pq->size = 0;
   pq->arr = (int *)malloc(capacity * sizeof(int));
   return pq;
}

// Swap two elements in the array
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

// Insert a new value into the priority queue
void insert(PriorityQueue *pq, int value) {
   if (pq->size >= pq->capacity) {
      printf("Priority Queue is full\n");
      return;
   }

   // Insert the new value at the end of the heap
   pq->arr[pq->size] = value;
   int index = pq->size;
   pq->size++;

   // Heapify up to maintain the max-heap property
   while (index != 0 && pq->arr[index] > pq->arr[(index - 1) / 2]) {
      swap(&pq->arr[index], &pq->arr[(index - 1) / 2]);
      index = (index - 1) / 2;
   }
}

// Heapify down to maintain the max-heap property
void maxHeapify(PriorityQueue *pq, int index) {
   int largest = index;
   int left = 2 * index + 1;
   int right = 2 * index + 2;

   if (left < pq->size && pq->arr[left] > pq->arr[largest]) {
      largest = left;
   }
```

```
    if (right < pq->size && pq->arr[right] > pq->arr[largest]) {
        largest = right;
    }

    if (largest != index) {
        swap(&pq->arr[index], &pq->arr[largest]);
        maxHeapify(pq, largest);
    }
}

// Extract the maximum element from the priority queue
int extractMax(PriorityQueue *pq) {
    if (pq->size <= 0) {
        printf("Priority Queue is empty\n");
        return -1;
    }

    if (pq->size == 1) {
        pq->size--;
        return pq->arr[0];
    }

    // Store the maximum value, and remove it from the heap
    int root = pq->arr[0];
    pq->arr[0] = pq->arr[pq->size - 1];
    pq->size--;

    // Heapify down from the root
    maxHeapify(pq, 0);

    return root;
}

// Print all elements in the priority queue
void printPriorityQueue(PriorityQueue *pq) {
    for (int i = 0; i < pq->size; i++) {
        printf("%d ", pq->arr[i]);
    }
    printf("\n");
}

// Free the memory allocated for the priority queue
void freePriorityQueue(PriorityQueue *pq) {
```

```
    free(pq->arr);
    free(pq);
}
```

Output: Priority Queue elements:
40 30 15 10 20

Extracted max: 40
Priority Queue elements after extraction:
30 20 15 10

2.Write a c program code for heap sorting.

Code:

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

// Function prototypes
void heapify(int arr[], int n, int i);
void heapSort(int arr[], int n);
void swap(int *a, int *b);
void printArray(int arr[], int size);

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    printArray(arr, n);

    heapSort(arr, n);

    printf("\nSorted array:\n");
    printArray(arr, n);

    return 0;
}

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to heapify a subtree rooted at index i
```

```
void heapify(int arr[], int n, int i) {
    int largest = i;  // Initialize largest as root
    int left = 2 * i + 1;  // left = 2*i + 1
    int right = 2 * i + 2; // right = 2*i + 2

    // Check if left child exists and is greater than root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // Check if right child exists and is greater than root
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // Swap and continue heapifying if root is not the largest
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

// Main function to perform heap sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // One by one extract elements
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // Call heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
```

```
    printf("\n");
}
```

Output: **Original array:**
12 11 13 5 6 7

Sorted array:
5 6 7 11 12 13