TASK SCHEDULING WITH PRIORITY QUEUES

A PROJECT REPORT

Submitted by

SRISHTI PANDA [RA2211033010146]
ASRITHA K. [RA2211033010140]
AASTHA SINGH [RA2211033010158]

for the course 21CSC201J Data Structures and Algorithms

Under the Guidance of

Dr. U. SAKTHI

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

NOVEMBER 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that the 21CSC201J Data Structures and Algorithms course project report titled **"TASK SCHEDULING WITH PRIORITY QUEUE"** is the bonafide work done by **SRISHTI PANDA [RA2211033010146], ASRITHA K. [RA2211033010140] and AASTHA SINGH [RA2211033010158]** who carried out under my supervision. Certified further, that to the best of my knowledge, the work reported herein doesnot form part of any other work.

Dr. U. Sakthi,
Faculty In-Charge,
Assistant Professor,
Department of Computational Intelligence,
SRMIST.

Dr. Annie Uthra
Professor and Head,
Department of Computational Intelligence,
SRMIST.

Department of Computing Technologies

SRM Institute of Science and Technology
Own Work Declaration Form

Degree/Course	: B.Tech in Computer Science and Engineering
Student Names	: SRISHTI PANDA , ASRITHA K. and AASTHA SINGH
Registration Number	: RA2211033010146, RA2211033010140, RA2211033010158
Title of Work	: **Task Scheduling with Priority Queues**

We here by certify that this assessment compiles with the University's Rules and Regulations relating to Academic misconduct and plagiarism, as listed in the University Website, Regulations, and the Education Committee guidelines.

We confirm that all the work contained in this assessment is our own except where indicated, and that we have met the following conditions:

- Clearly references / listed all sources as appropriate.
- Referenced and put in inverted commas all quoted text (from books, web,etc.)
- Given the sources of all pictures, data etc that are not my own.
- Not made any use of the report(s) or essay(s) of any other student(s)either past or present.
- Acknowledged in appropriate places any help that I have received from others (e.g fellow students, technicians, statisticians, external sources).
- Compiled with any other plagiarism criteria specified in the Course hand book / University website

I understand that any false claim for this work will be penalized in accordance with the University policies and regulations.

DECLARATION:
I am aware of and understand the University's policy on Academic misconduct and plagiarism and I certify that this assessment is my / our own work, except where indicated by referring, and that I have followed the good academic practices noted above. **Student 1 Signature:** **Student 2 Signature:** **Student 3 Signature:** **Date:**
If you are working in a group, please write your registration numbers and sign with the date for every student in your group.

TABLE OF CONTENT

CHAPTER 1
PROBLEM DEFINITION

The Task Scheduling Program aims to efficiently manage a priority queue of tasks, providing users with a robust platform to control task execution based on specified priorities and task IDs. The program allows users to enqueue tasks into the priority queue, where each task is associated with a priority level and a unique identifier (task ID). This enables the orderly execution of tasks based on their importance.

Users have the flexibility to adjust the execution order dynamically by preempting tasks, temporarily pausing their execution. Preemption ensures that higher-priority tasks can be introduced or prioritized over currently running tasks. This functionality is crucial for adapting to changing requirements and optimizing resource utilization.

Additionally, the program facilitates the dequeuing of tasks from the priority queue. This operation retrieves and removes the highest-priority task, allowing for the orderly execution of tasks according to their designated importance. Dequeuing ensures that tasks are processed efficiently, maintaining the integrity of the priority-based scheduling.

Efficiency is a key focus of the program, ensuring that task management operations are executed in a timely manner. The priority queue data structure is leveraged to guarantee that tasks with higher priorities are processed before lower-priority tasks. This systematic approach maximizes the utility of resources and enhances the overall responsiveness of the task scheduling system.

By providing a comprehensive set of functionalities, including task enqueueing, preemption, and dequeuing, the Task Scheduling Program empowers users to orchestrate task execution based on their priority and importance. The design and implementation of this program consider the dynamic nature of task scheduling, offering a flexible and efficient solution for managing a variety of tasks in diverse environments.

CHAPTER 2
PROBLEM EXPALANTION

The Task Scheduling Program is conceived to address the crucial need for efficient task management in various applications. It revolves around the concept of a priority queue, which is a data structure that maintains tasks in a way that reflects their priority levels. The overarching goal is to enable users to control the execution flow of tasks based on their significance and urgency.

One fundamental aspect of the program is the ability to enqueue tasks into the priority queue. Task enqueuing involves assigning each task a priority level and a unique identifier (task ID) before placing it in the queue. This process ensures that tasks are organized in a manner that reflects their relative importance. By allowing users to specify these attributes, the program accommodates a diverse range of scenarios where certain tasks need to take precedence over others.

The concept of preemption adds another layer of sophistication to the task management system. Preemption allows users to temporarily pause the execution of a task, making it particularly useful in dynamic environments where priorities may shift. This functionality provides flexibility in adapting to changing requirements or unexpected events by allowing higher-priority tasks to be introduced and executed promptly.

Dequeuing tasks from the priority queue is a crucial operation for task execution. The program is designed to efficiently retrieve and remove tasks from the queue in accordance with their priorities. This ensures that tasks are executed in a well-organized manner, addressing higher-priority tasks before moving on to lower-priority ones.

In summary, the Task Scheduling Program is tailored to meet the demands of effective task management by providing a platform for enqueuing, preempting, and dequeuing tasks within a priority queue. Its design considerations take into account the dynamic nature of task scheduling, offering a versatile solution applicable in various contexts.

CHAPTER 3

DATA STRUCTURE USED

Introduction to Priority Queues:

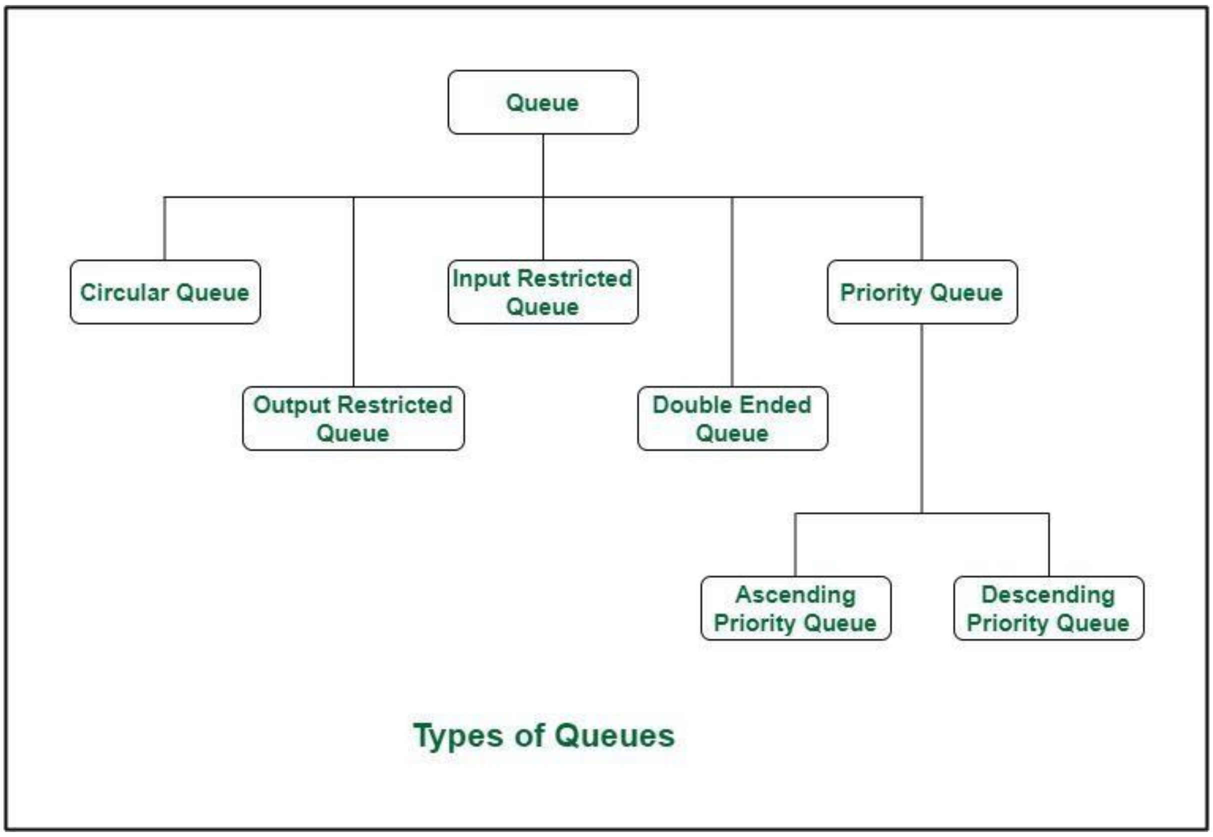
Queues are a fundamental data structure in computer science. They are a linear data structure that follows the first-in-first-out (FIFO) principle. This means that the first element to be added to the queue is also the first element to be removed. Queues are often used to implement buffers, waiting lists, and other systems where items need to be processed in order.

To add an element to the queue, we create a new node and insert it at the tail of the queue. To remove an element from the queue, we remove the head node.

Queues have many different applications in computer science. For example, queues are used in operating systems to schedule tasks for execution, in networking to buffer data packets, and in web browsers to manage the loading of web pages.

There are many types of queues:

1. Linear Queue
2. Circular Queue
3. Priority Queue
4. Double-ended Queue (Deque)
5. Blocking Queue
6. Parallel Queue
7. Counting Queue
8. Delay Queue
9. Priority Blocking Queue
10. Multi-level Queue



Priority queues are fundamental data structures used in various applications, particularly in task scheduling problems. They organize elements based on their priorities, ensuring that higher-priority items are processed first. Commonly found in operating systems, real-time systems, and job schedulers, priority queues play a crucial role in optimizing task execution.

Priority queues are widely used in task scheduling problems, where each task has a priority associated with it. The priority queue allows efficient management of tasks based on their priorities, ensuring that higher priority tasks are executed first. This application can be found in operating systems, real-time systems, and job schedulers.

A priority is a value that marks the “importance” of a queue element. In a priority queue, the order of entry is not of much concern; the order of exit is. Each component of the priority queue is dequeued based on its priority. The priority can be of two types, depending on the situation.

Properties of Priority Queues:

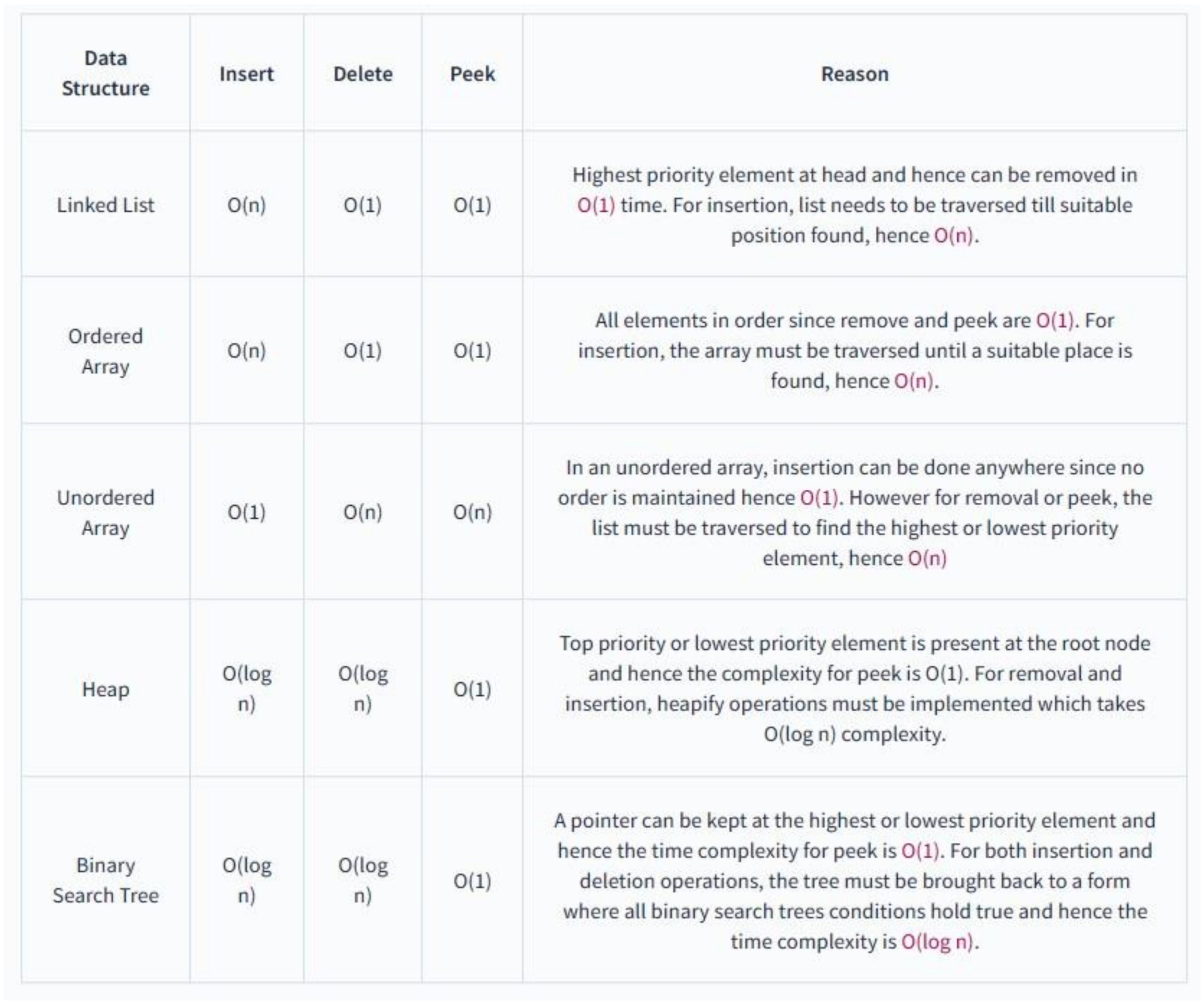
- Elements are stored based on their associated priority values.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.
- Highest and lowest priority elements can be accessed in constant time, irrespective of the number of elements present in the queue.
- After every insertion or deletion, depending on the underlying data structure, the queue is reorganized to maintain the order.
- Only the highest priority element can be removed from a priority queue. You should use multisets to support removal of intermediary elements.

Implementation of Priority Queues:

There are several ways to implement a priority queue:

- array
- linked list
- heap
- binary search tree

Each method has its own advantages and disadvantages, and the best choice will depend on the specific needs of your application.



Data Structure	Insert	Delete	Peek	Reason
Linked List	O(n)	O(1)	O(1)	Highest priority element at head and hence can be removed in O(1) time. For insertion, list needs to be traversed till suitable position found, hence O(n).
Ordered Array	O(n)	O(1)	O(1)	All elements in order since remove and peek are O(1). For insertion, the array must be traversed until a suitable place is found, hence O(n).
Unordered Array	O(1)	O(n)	O(n)	In an unordered array, insertion can be done anywhere since no order is maintained hence O(1). However for removal or peek, the list must be traversed to find the highest or lowest priority element, hence O(n)
Heap	O(log n)	O(log n)	O(1)	Top priority or lowest priority element is present at the root node and hence the complexity for peek is O(1). For removal and insertion, heapify operations must be implemented which takes O(log n) complexity.
Binary Search Tree	O(log n)	O(log n)	O(1)	A pointer can be kept at the highest or lowest priority element and hence the time complexity for peek is O(1). For both insertion and deletion operations, the tree must be brought back to a form where all binary search trees conditions hold true and hence the time complexity is O(log n).

Operations of a Priority Queue:

1) Insertion in a Priority Queue

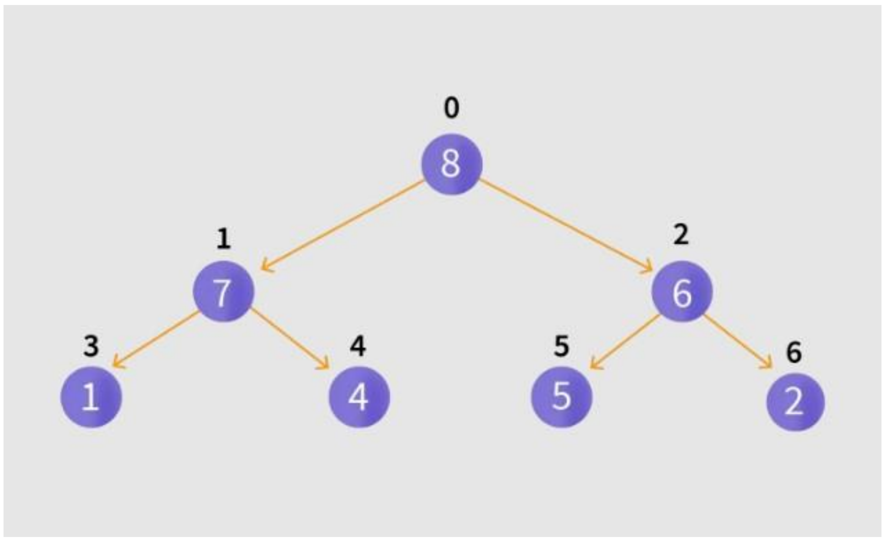
When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct position, it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.

2) Deletion in a Priority Queue

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.

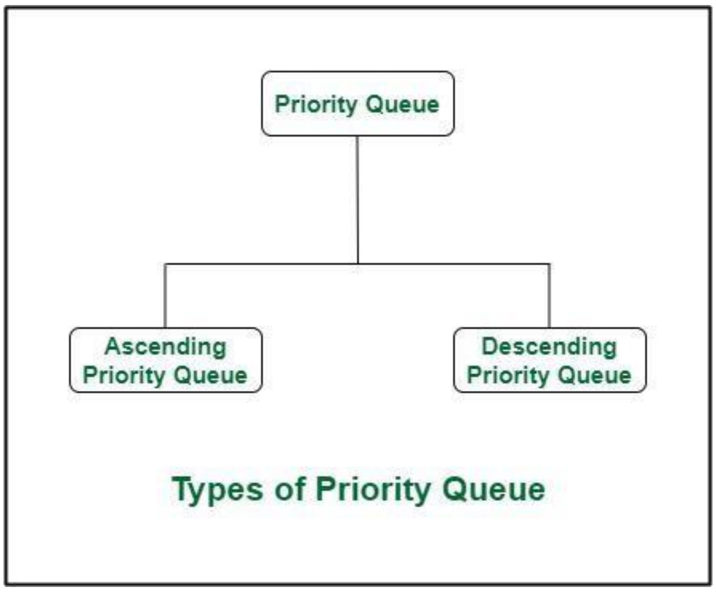
3) Peek in a Priority Queue

This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.



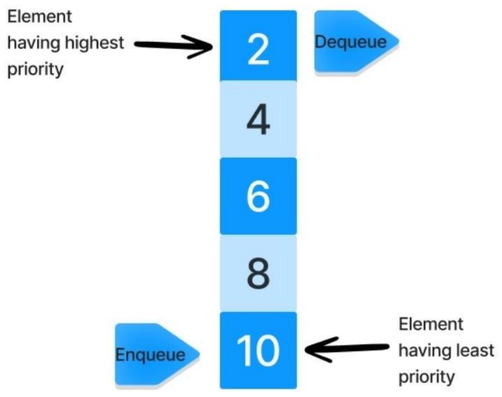
Types of Priority Queue:

Two common types of priority queues are ascending order and descending order.



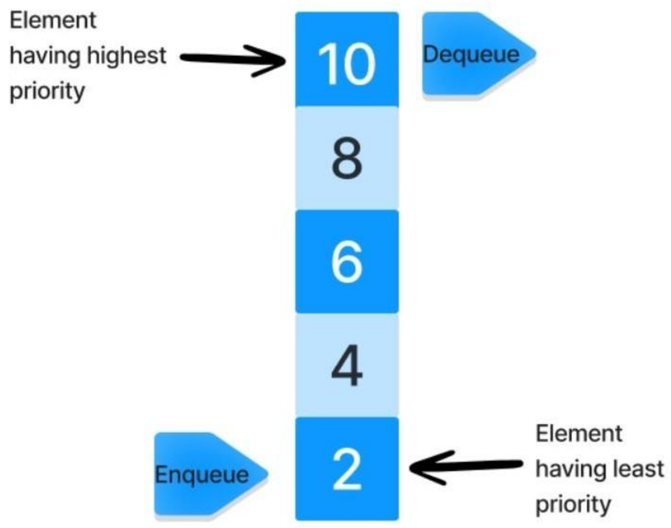
1) Ascending Order Priority Queue (Lower value, higher priority)

As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 2,4,6,8,10. Here, 2 is the smallest number, therefore, it will get the highest priority in a priority queue and so when we dequeue from this type of priority queue, 2 will remove from the queue and dequeue returns 2.


Element having highest priority
2
Dequeue
4
6
8
Enqueue
10
Element having least priority

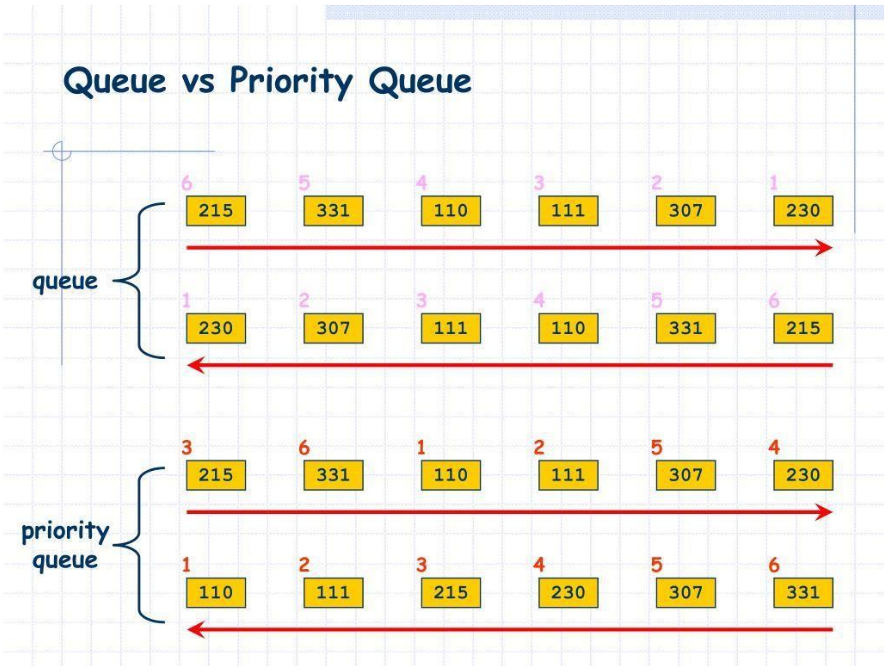
2) Descending order Priority Queue (Higher value, higher priority)

The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.

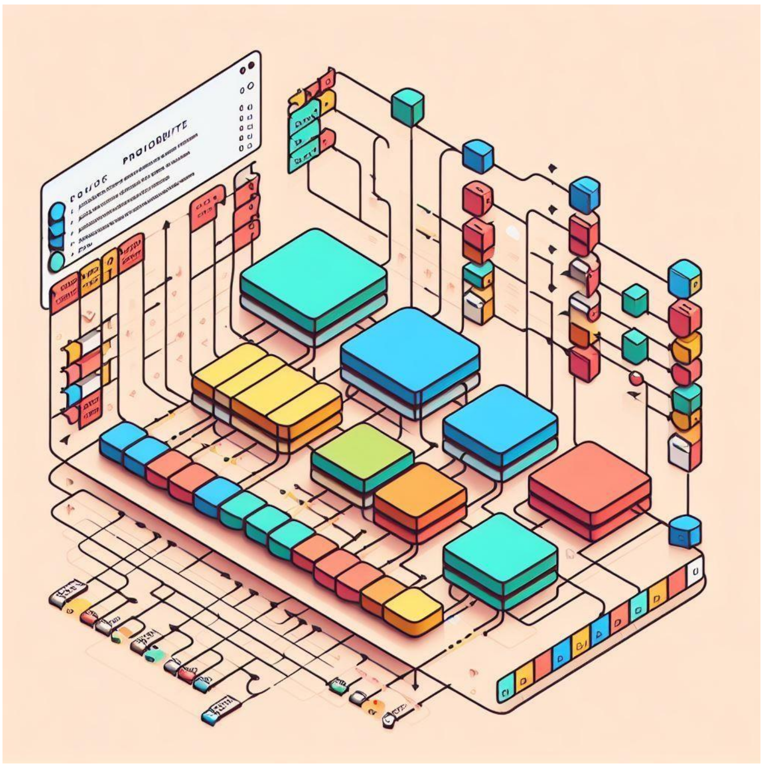

Element having highest priority
10
Dequeue
8
6
4
Enqueue
2
Element having least priority

Difference Between Priority Queue and Normal Queue:

The fundamental distinction lies in the absence of priority in a normal queue, where the first-in-first-out (FIFO) rule is followed. Priority queues, on the other hand, serve elements based on their assigned priorities.



A normal queue where everyone has to wait till the first person’s checking is done. Prime example of first come first serve (FSFS).



The circuit system of any microcontroller has a brain that arranges tasks by priority.

Application of Priority Queue:

Priority queues find applications in diverse fields:

1. Algorithms:
   - ➢ Dijkstra's Algorithm: Finds the shortest path in a graph by storing the graph in an adjacency list then use a priority queue to extract the minimum path.
   - ➢ Prim's Algorithm: Determines the Minimum Spanning Tree in a weighted undirected graph by using a priority queue to store the keys of nodes. Then we extract minimum key nodes at every step.

2. Data Compression:
   - ➢ Huffman Coding: Utilizes priority queues to compress data efficiently.

3. Operating Systems:
    - Priority Scheduling: Schedules processes based on their priority.
    - Load Balancing: Balances network or application traffic across multiple servers.
    - Interrupt Handling: Manages interruptions by prioritizing processes.

4. In Artificial Intelligence:
    - A priority queue is used to implement the AI search algorithm.
    - The AI search algorithm first attempts the most promising routes to discover the shortest path between two vertices in a weighted graph.
    - The priority queue is used to keep track of the unexplored routes, with the one with the shortest lower bound on the total path length receiving the most priority.

5. Real-Life Examples:
    - Hospitals: Emergency cases receive priority in queues.
    - Bandwidth Management: Prioritizes essential data packets in networks.

In this project, the implementation of a priority queue revolves around the use of a linked list. Specifically, a descending order priority queue has been employed to develop a task scheduling program in the C programming language.

The project provides a user-friendly interface, presenting four distinct options upon program execution: enqueue a task, preempt a task, dequeue a task, or exit. To enqueue a task, the user inputs both the task ID and its associated priority level. Once a task is enqueued, the user is given the flexibility to either dequeue it or preempt (interrupt) its execution. Importantly, if the queue is empty, the user is informed that dequeueing is not possible, preventing potential errors.

The core mechanism of dequeuing tasks operates on the principle of descending order priority. For instance, when there are tasks with priorities 3 and 4 in the queue, the dequeue operation will prioritize the task with priority 4, ensuring that tasks are processed in order of decreasing priority.

This ensures a systematic and efficient execution of tasks, where higher-priority tasks are addressed before lower-priority ones, aligning with the priority queue's inherent characteristics. Overall, the project provides a versatile and interactive platform for users to manage tasks based on their specified priorities within a descending order priority queue.

The utilization of a linked list for the implementation of the priority queue adds a dynamic and adaptable dimension to the project. Linked lists allow for efficient memory management and flexibility in accommodating varying numbers of tasks. As opposed to fixed-size arrays, linked lists enable the queue to dynamically adjust to the number of enqueued tasks, making the program more scalable.

Additionally, the descending order priority queue ensures that the tasks are processed in a manner that aligns with their significance. The linked list structure allows for seamless insertion and removal operations, enhancing the overall responsiveness and effectiveness of the task scheduling program. Overall, the combination of linked list implementation and descending order priority queue design offers a robust and adaptable solution for task management in the C program.

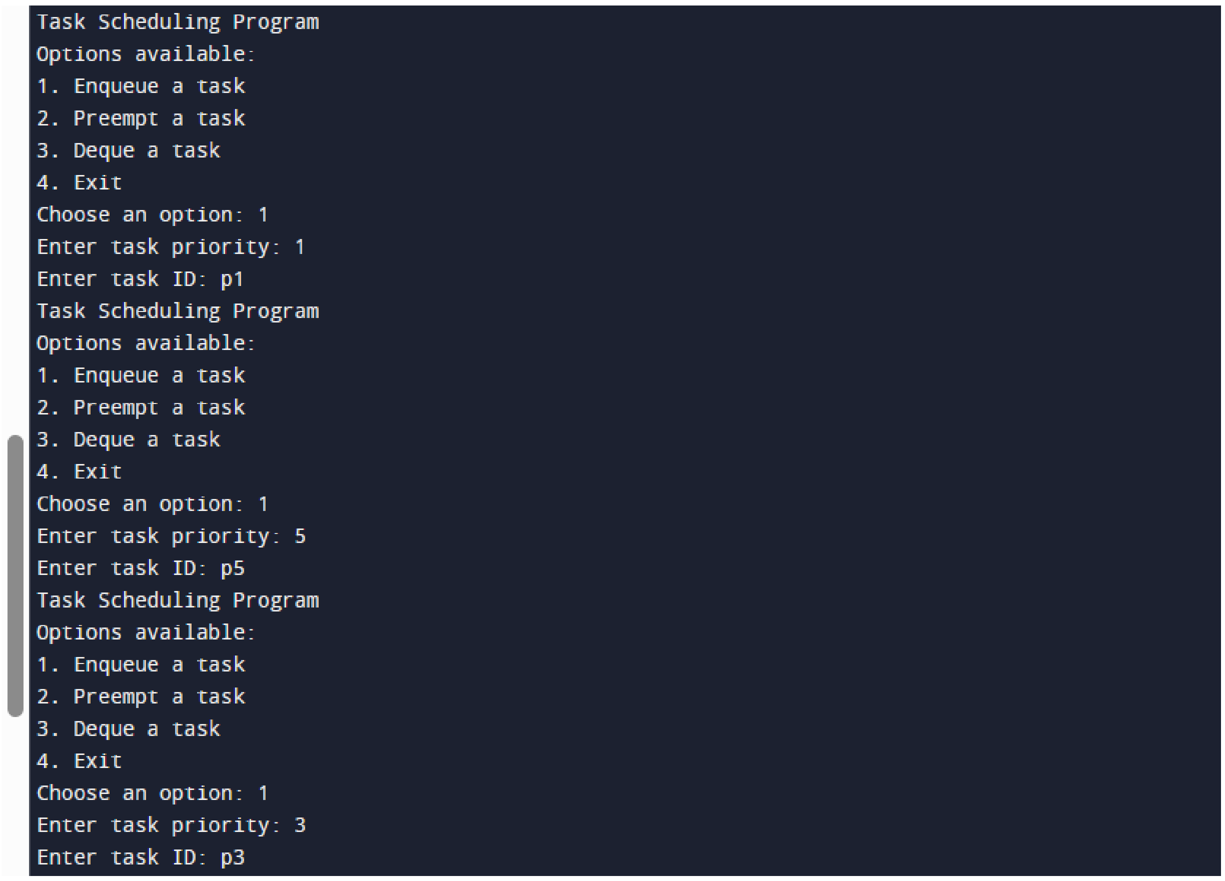
CHAPTER 4
TASK SCHEDULER SIMULATION

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

#define MAX_SIZE 100

// Task structure to represent a task with priority and task ID
typedef struct {
   int priority;
   char taskID[20];
} Task;

// Priority Queue structure to hold tasks
typedef struct {
   Task tasks[MAX_SIZE];
   int size;
} PriorityQueue;

// Function to initialize a priority queue
void init(PriorityQueue *pq) {
   pq->size = 0;
}

// Function to enqueue a task with a given priority and task ID
void enqueue(PriorityQueue *pq, int priority, const char *taskID) {
   if (pq->size >= MAX_SIZE) {
      printf("Queue is full, cannot enqueue more tasks.\n");
      return;
   }

   Task  newTask;
   newTask.priority = priority;
   strcpy(newTask.taskID, taskID);

   // Find the correct position to insert the new task based on priority
```

```
    int i = pq->size - 1;
    while (i >= 0 && pq->tasks[i].priority < newTask.priority) {
        pq->tasks[i + 1] = pq->tasks[i];
        i--;
    }
    pq->tasks[i + 1] = newTask;
    pq->size++;
}

// Function to dequeue a task (remove and display)
void dequeue(PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Queue is empty, cannot dequeue.\n");
        return;
    }

    printf("Dequeued task with ID: %s and priority: %d\n", pq->tasks[0].taskID, pq-
>tasks[0].priority);
    // Shift remaining tasks to fill the gap
    for (int i = 1; i < pq->size; i++) {
        pq->tasks[i - 1] = pq->tasks[i];
    }
    pq->size--;
}

// Function to preempt the task with the lowest priority
void preemptTask(PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Queue is empty, cannot preempt any task.\n");
        return;
    }

    // Find the task with the lowest priority
    int lowestPriorityIndex = 0;
    for (int i = 1; i < pq->size; i++) {
        if (pq->tasks[i].priority < pq->tasks[lowestPriorityIndex].priority) {
            lowestPriorityIndex = i;
        }
    }
    // Preempt the task with the lowest priority
    printf("Preempted task with ID: %s and priority: %d. Task remains in the queue.\n",
           pq->tasks[lowestPriorityIndex].taskID, pq-
>tasks[lowestPriorityIndex].priority);
}

// Main function for the task scheduling program
int main() {
    PriorityQueue pq;
    init(&pq);

    int priority, option;
    char taskID[20];

    do {
        printf("\nTask Scheduling Program\n");
        printf("Options available:\n1. Enqueue a task\n2. Preempt a task\n3. Dequeue a
task\n4. Exit\nChoose an option: ");
        scanf("%d", &option);

        switch (option) {
            case 1:
                printf("Enter task priority: ");
                scanf("%d", &priority);
                printf("Enter task ID: ");
                scanf("%s", taskID);
                enqueue(&pq, priority, taskID);
                break;
            case 2:
                preemptTask(&pq);
                break;
            case 3:
                dequeue(&pq);
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid option. Please try again.\n");
        }
    } while (option != 4);

    return 0;
}
```

CHAPTER 5
OUTPUT

The following are the output screenshots obtained after running the above code



```
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 1
Enter task priority: 1
Enter task ID: p1
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 1
Enter task priority: 5
Enter task ID: p5
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 1
Enter task priority: 3
Enter task ID: p3
```

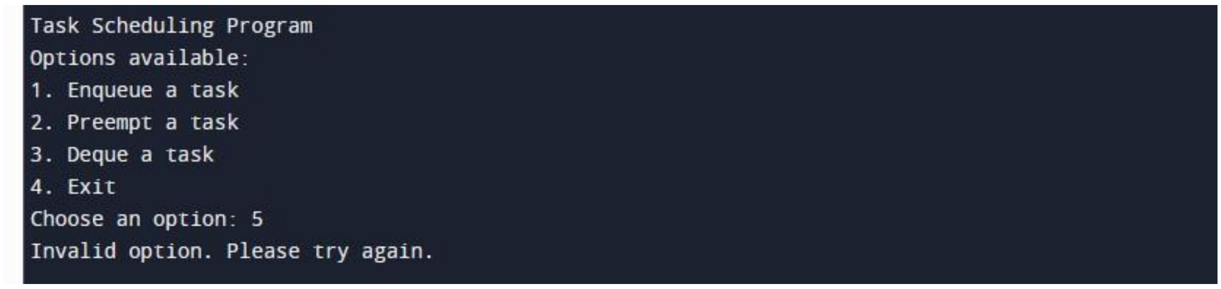
Here, the we enqueue 3 tasks, one at a time after giving the necessary details:

P1 of priority 1

P5 of priority 5

P3 of priority 3

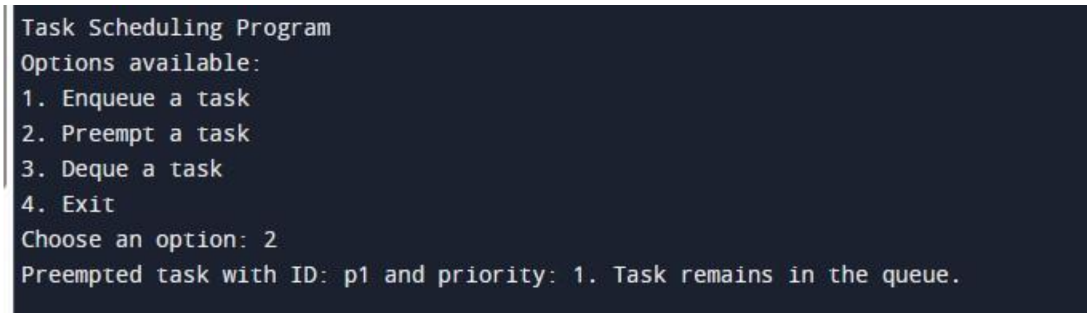
If suppose the user inputs any invalid choice, the system will return a message to input only from the given oprtion like so:



```
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 5
Invalid option. Please try again.
```

In priority-based scheduling systems, tasks with higher priority are given preference over lower-priority tasks. Preemption in such systems typically involves interrupting the execution of a lower-priority task to allow a higher-priority task to run.

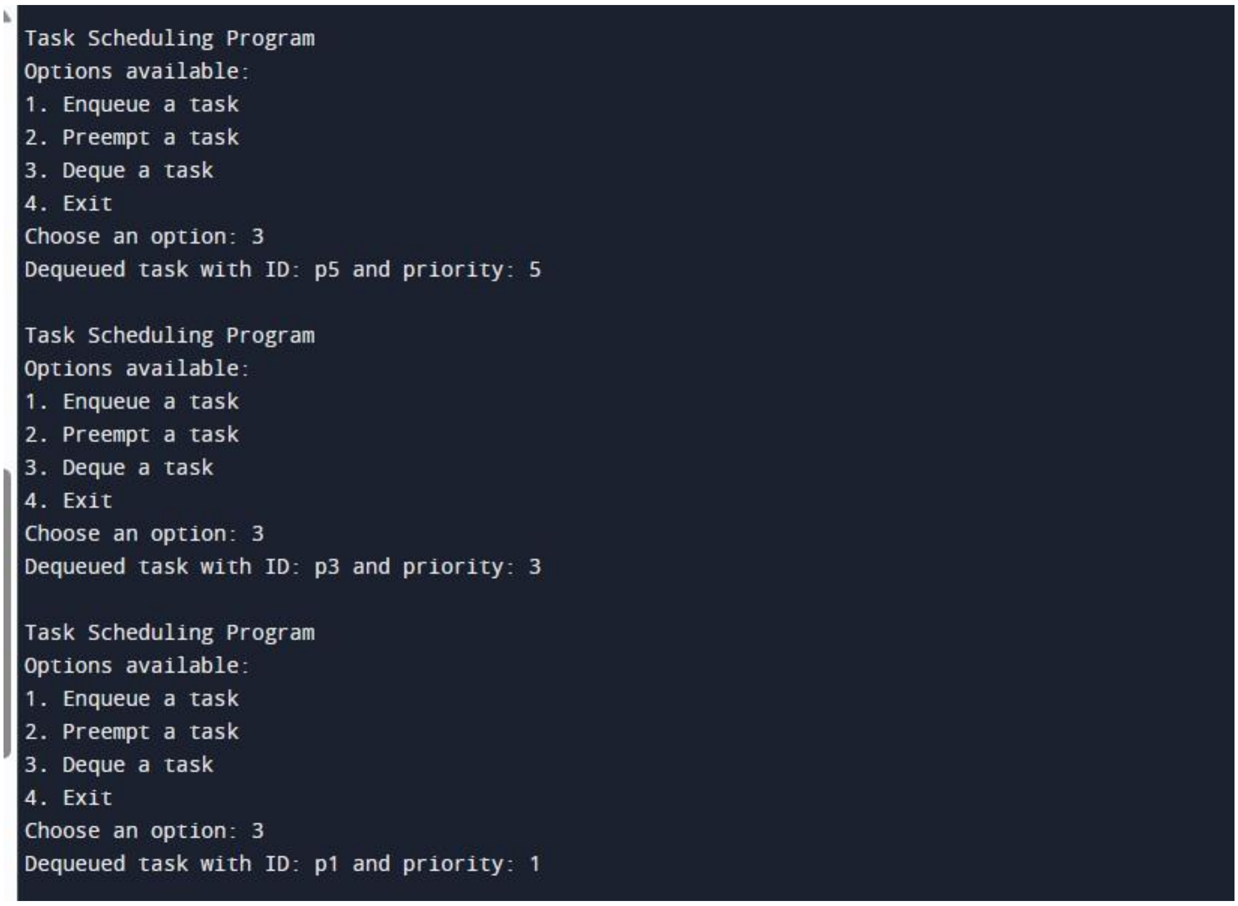
As we tried to replicate a program close to how it operates in real-life, the program only preempts the lowest priority task alone



```
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 2
Preempted task with ID: p1 and priority: 1. Task remains in the queue.
```

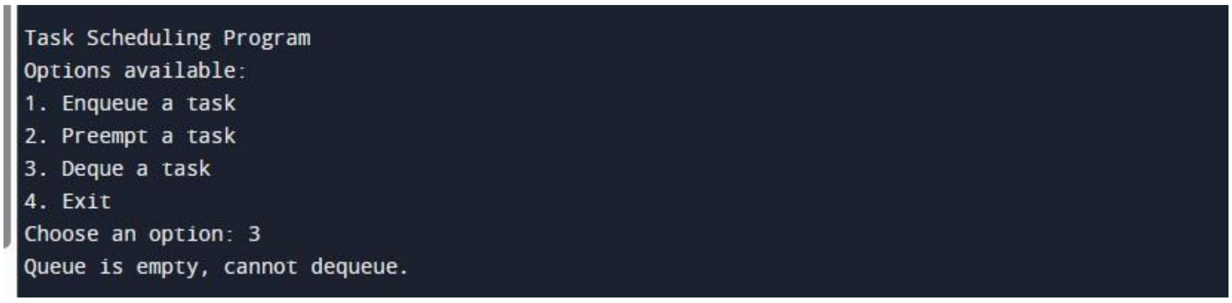
Since we implemented the task scheduler using descending order priority queues by linked lists, the tasks will be deqeued in the descending order of priority, regardless of the order it was enqueued in.

P5 with priority 5 is dequeued first, followed by P3 with priority 3 and finally P1 with priority 1.



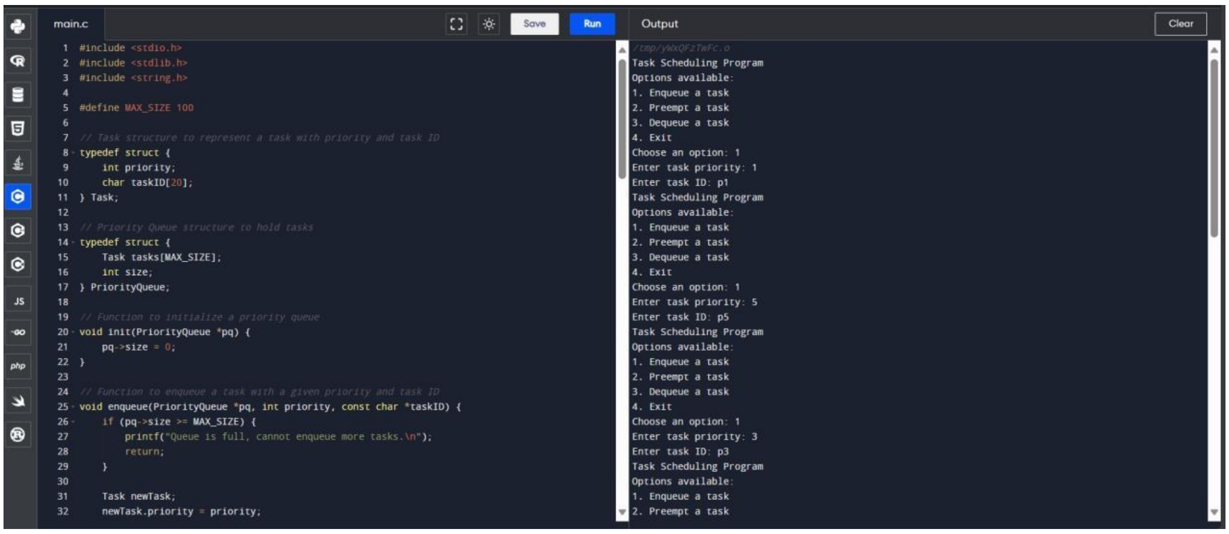
```
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 3
Dequeued task with ID: p5 and priority: 5

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 3
Dequeued task with ID: p3 and priority: 3

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 3
Dequeued task with ID: p1 and priority: 1
```

After all the tasks are executed, the queue is empty. If the user tries to dequeue now, the system will send a message.



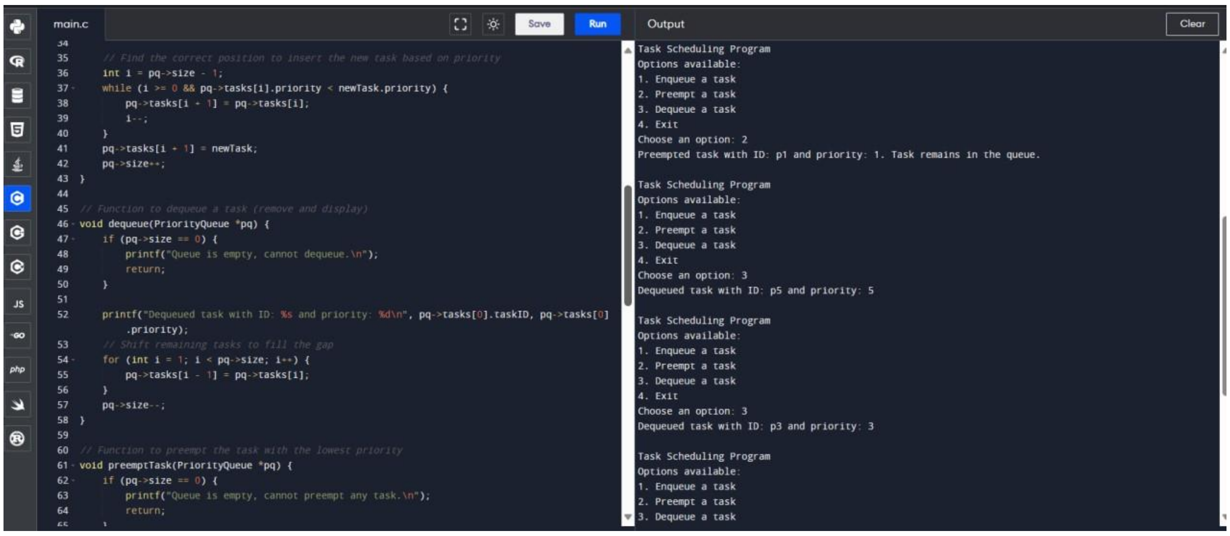
```
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Deque a task
4. Exit
Choose an option: 3
Queue is empty, cannot dequeue.
```

Now, the user can again start the cycle by enqueuing, preempting and dequeuing tasks.



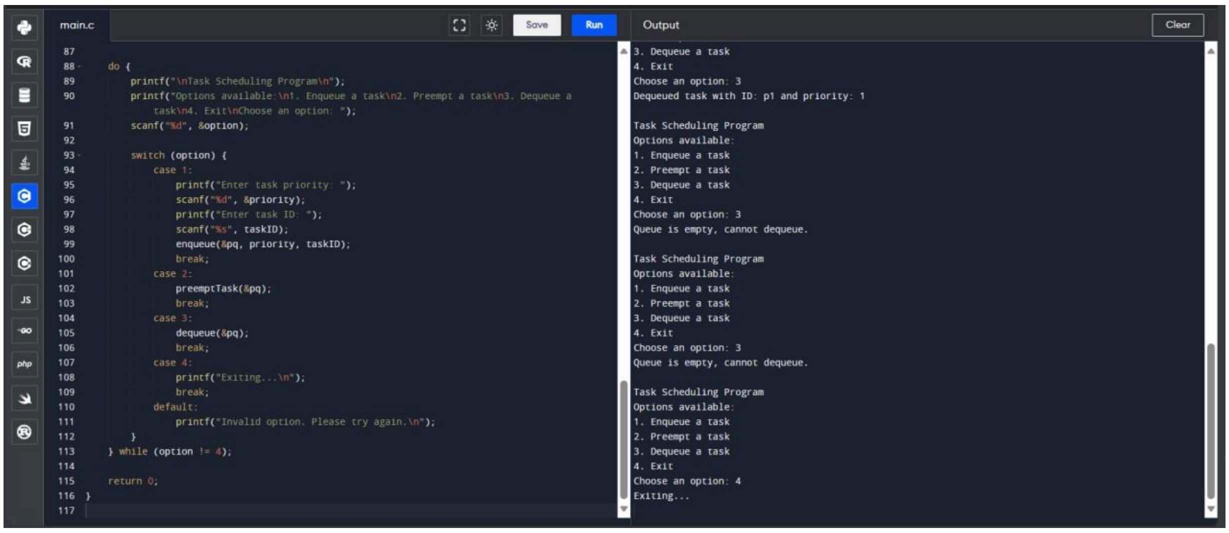
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 100

// Task structure to represent a task with priority and task ID
typedef struct {
    int priority;
    char taskID[20];
} Task;

// Priority Queue structure to hold tasks
typedef struct {
    Task tasks[MAX_SIZE];
    int size;
} PriorityQueue;

// Function to initialize a priority queue
void init(PriorityQueue *pq) {
    pq->size = 0;
}

// Function to enqueue a task with a given priority and task ID
void enqueue(PriorityQueue *pq, int priority, const char *taskID) {
    if (pq->size >= MAX_SIZE) {
        printf("Queue is full, cannot enqueue more tasks.\n");
        return;
    }

    Task newTask;
    newTask.priority = priority;
```

```
/tmp/yWxQFzTwFc.o
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 1
Enter task priority: 1
Enter task ID: p1
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 1
Enter task priority: 5
Enter task ID: p5
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 1
Enter task priority: 3
Enter task ID: p3
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
```



```c

    // Find the correct position to insert the new task based on priority
    int i = pq->size - 1;
    while (i >= 0 && pq->tasks[i].priority < newTask.priority) {
        pq->tasks[i + 1] = pq->tasks[i];
        i--;
    }
    pq->tasks[i + 1] = newTask;
    pq->size++;
}

// Function to dequeue a task (remove and display)
void dequeue(PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Queue is empty, cannot dequeue.\n");
        return;
    }

    printf("Dequeued task with ID: %s and priority: %d\n", pq->tasks[0].taskID, pq->tasks[0]
        .priority);
    // Shift remaining tasks to fill the gap
    for (int i = 1; i < pq->size; i++) {
        pq->tasks[i - 1] = pq->tasks[i];
    }
    pq->size--;
}

// Function to preempt the task with the lowest priority
void preemptTask(PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Queue is empty, cannot preempt any task.\n");
        return;
    }
```

```
Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 2
Preempted task with ID: p1 and priority: 1. Task remains in the queue.

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 3
Dequeued task with ID: p5 and priority: 5

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 3
Dequeued task with ID: p3 and priority: 3

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
```



```c

    do {
        printf("\nTask Scheduling Program\n");
        printf("Options available:\n1. Enqueue a task\n2. Preempt a task\n3. Dequeue a
            task\n4. Exit\nChoose an option: ");
        scanf("%d", &option);

        switch (option) {
            case 1:
                printf("Enter task priority: ");
                scanf("%d", &priority);
                printf("Enter task ID: ");
                scanf("%s", taskID);
                enqueue(&pq, priority, taskID);
                break;
            case 2:
                preemptTask(&pq);
                break;
            case 3:
                dequeue(&pq);
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid option. Please try again.\n");
        }
    } while (option != 4);

    return 0;
}

```

```
3. Dequeue a task
4. Exit
Choose an option: 3
Dequeued task with ID: p1 and priority: 1

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 3
Queue is empty, cannot dequeue.

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 3
Queue is empty, cannot dequeue.

Task Scheduling Program
Options available:
1. Enqueue a task
2. Preempt a task
3. Dequeue a task
4. Exit
Choose an option: 4
Exiting...
```

CHAPTER 6
CONCLUSION

In conclusion, priority queues stand as essential data structures with wide-ranging applications, serving both algorithmic optimization and real-world scenarios. Their inherent ability to efficiently manage elements based on priority values positions them as valuable tools across diverse fields. Despite the comparatively slower time complexity associated with enqueue and dequeue operations, the distinct advantage lies in the rapid access to high-priority items. This attribute makes priority queues a powerful and indispensable solution for addressing a variety of problems, offering an effective means of handling elements with differing levels of importance.

The significance of the priority queue becomes apparent when tasked with managing items of varying priorities. Its major strength lies in the ability to access the highest priority item swiftly, boasting a time complexity of just O(1). However, it's crucial to acknowledge that the enqueue and dequeue operations exhibit a relatively slower time complexity of O(log n). Despite this drawback, priority queues remain pivotal in solving coding challenges, with notable applications like optimizing the connection of ropes with minimum cost or identifying the top k frequent keywords.

In practical terms, the utility of priority queues transcends theoretical limitations. They excel in efficiently handling tasks where prioritization is essential, making them indispensable tools in scenarios ranging from algorithmic optimizations to real-world problem-solving. While acknowledging the slower time complexity for certain operations, the unparalleled advantage of quick access to high-priority items solidifies the priority queue as a robust and versatile solution across a spectrum of challenges.

CHAPTER 7
REFERENCES

1. https://www.codingninjas.com/studio/library/applications-of-priority-queue

2. https://www.scaler.com/topics/data-structures/priority-queue-in-data-structure/

3. https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

4. https://www.prepbytes.com/blog/queues/applications-of-priority-queue/#:~:text=Task%20Scheduling&text=The%20priority%20queue%20allows%20efficient,time%20systems%2C%20and%20job%20schedulers