S.No: 1	Exp. Name: ***Project Module***	Date: 2024-06-14

Aim:

Project Module

Source Code:

hello.c

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

#define MAX_JOBS 100
#define HASH_SIZE 101

typedef struct Job {
    int jobId;
    int priority;
    char description[100];
} Job;

typedef struct Heap {
    Job* jobs[MAX_JOBS];
    int size;
} Heap;

typedef struct HashNode {
    Job* job;
    struct HashNode* next;
} HashNode;

typedef struct HashTable {
    HashNode* table[HASH_SIZE];
} HashTable;

typedef struct QueueNode {
    Job* job;
    struct QueueNode* next;
} QueueNode;

typedef struct Queue {
    QueueNode* front;
    QueueNode* rear;
} Queue;

// Function prototypes
Heap* createHeap();
void insertHeap(Heap* heap, Job* job);
Job* extractMaxHeap(Heap* heap);
void heapifyDown(Heap* heap, int index);

HashTable* createHashTable();
void insertHashTable(HashTable* ht, Job* job);
Job* findHashTable(HashTable* ht, int jobId);

Queue* createQueue();
void enqueue(Queue* queue, Job* job);
Job* dequeue(Queue* queue);

int hash(int jobId);

int main() {
    // Initialize data structures
```

```
    Queue* queue = createQueue();

    int choice;
    int jobIdCounter = 1;

    while (1) {
        printf("\nJob Scheduler Menu:\n");
        printf("1. Add Job\n");
        printf("2. Schedule Jobs\n");
        printf("3. Process Jobs\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        if (choice == 4) {
            break;
        }

        switch (choice) {
            case 1: {
                Job* job = (Job*)malloc(sizeof(Job));
                job->jobId = jobIdCounter++;

                printf("Enter job priority: ");
                scanf("%d", &job->priority);

                printf("Enter job description: ");
                getchar(); // To consume the newline character left by scanf
                fgets(job->description, sizeof(job->description), stdin);
                job->description[strcspn(job->description, "\n")] = 0; // Remove trailing
newline

                insertHeap(heap, job);
                insertHashTable(ht, job);

                printf("Job added with ID %d\n", job->jobId);
                break;
            }
            case 2: {
                Job* scheduledJob = extractMaxHeap(heap);
                while (scheduledJob != NULL) {
                    enqueue(queue, scheduledJob);
                    scheduledJob = extractMaxHeap(heap);
                }
                printf("Jobs scheduled based on priority.\n");
                break;
            }
            case 3: {
                Job* job;
                while ((job = dequeue(queue)) != NULL) {
                    printf("Processing Job ID: %d, Priority: %d, Description: %s\n", job-
>jobId, job->priority, job->description);
                }
                break;
            }
```

```
                break;
            }
        }

        return 0;
}

// Heap functions
Heap* createHeap() {
    Heap* heap = (Heap*)malloc(sizeof(Heap));
    heap->size = 0;
    return heap;
}

void insertHeap(Heap* heap, Job* job) {
    if (heap->size == MAX_JOBS) {
        printf("Heap is full\n");
        return;
    }
    heap->jobs[heap->size] = job;
    int index = heap->size;
    heap->size++;

    while (index != 0 && heap->jobs[(index - 1) / 2]->priority < heap->jobs[index]-
>priority) {
        Job* temp = heap->jobs[(index - 1) / 2];
        heap->jobs[(index - 1) / 2] = heap->jobs[index];
        heap->jobs[index] = temp;
        index = (index - 1) / 2;
    }
}

Job* extractMaxHeap(Heap* heap) {
    if (heap->size == 0) {
        return NULL;
    }
    Job* root = heap->jobs[0];
    heap->jobs[0] = heap->jobs[heap->size - 1];
    heap->size--;
    heapifyDown(heap, 0);
    return root;
}

void heapifyDown(Heap* heap, int index) {
    int largest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < heap->size && heap->jobs[left]->priority > heap->jobs[largest]->priority) {
        largest = left;
    }
    if (right < heap->size && heap->jobs[right]->priority > heap->jobs[largest]->priority) {
        largest = right;
    }
    if (largest != index) {
```

```
        heap->jobs[largest] = temp;
        heapifyDown(heap, largest);
    }
}

// Hash table functions
HashTable* createHashTable() {
    HashTable* ht = (HashTable*)malloc(sizeof(HashTable));
    for (int i = 0; i < HASH_SIZE; i++) {
        ht->table[i] = NULL;
    }
    return ht;
}

void insertHashTable(HashTable* ht, Job* job) {
    int hashIndex = hash(job->jobId);
    HashNode* newNode = (HashNode*)malloc(sizeof(HashNode));
    newNode->job = job;
    newNode->next = ht->table[hashIndex];
    ht->table[hashIndex] = newNode;
}

Job* findHashTable(HashTable* ht, int jobId) {
    int hashIndex = hash(jobId);
    HashNode* current = ht->table[hashIndex];
    while (current != NULL) {
        if (current->job->jobId == jobId) {
            return current->job;
        }
        current = current->next;
    }
    return NULL;
}

int hash(int jobId) {
    return jobId % HASH_SIZE;
}

// Queue functions
Queue* createQueue() {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->front = NULL;
    queue->rear = NULL;
    return queue;
}

void enqueue(Queue* queue, Job* job) {
    QueueNode* newNode = (QueueNode*)malloc(sizeof(QueueNode));
    newNode->job = job;
    newNode->next = NULL;
    if (queue->rear == NULL) {
        queue->front = newNode;
        queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
```

```
}

Job* dequeue(Queue* queue) {
    if (queue->front == NULL) {
        return NULL;
    }
    QueueNode* temp = queue->front;
    Job* job = temp->job;
    queue->front = queue->front->next;
    if (queue->front == NULL) {
        queue->rear = NULL;
    }
    free(temp);
    return job;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
Hello World