```
// Name: Rodrigo Ignacio Rojas Garcia
// Course Number: ECE 2230
// Section: 001
// Semester: Spring 2017
// Assignment Number: 3
// Â© Rodrigo Rojas. All Rights Reserved.

// Library Declaration Section
#include <stdio.h>
#include <stdlib.h>
#include "disk_queue.h"
#include "event_queue.h"
#include "structures.h"

// Defines structure of event_queue_s
struct event_queue_s
{
    event_t *data;
    int size;
    int num_of_events;
};

// Function event_queue_t will create an event_queue which then will be allocated w
ith dynamic memory depending of the size that is passed on the
// function creating an array of event_queue_t pointers
event_queue_t event_queue_init(int size)
{
    event_queue_t event_queue;
    event_queue = (event_queue_t)calloc(1, sizeof(struct event_queue_s));
    event_queue->data = (event_t *)calloc(size + 1, sizeof(event_t));
    event_queue->size = size;
    event_queue->num_of_events = 0;
    return event_queue;
}

// Function event_queue_insert  will first check tha the number of events put in th
e even_queue do not surpass the number of events
// if true it will add one to number of events, the the current to the number of ev
ents so parent can be calculated as well as where
// the event needs to be stored in the array. Then, it enters a while loop which ch
ecks that the parent does not go into
// slot 0 of the event_queue since it is the end beginnign fo the array and nothing
 is stored there. If is not true, it goes into
// the loop and checks that the event passed in the function has a smaller event_ti
me in it, and if true it will switch the new
// event passed with the current parent and will not stop until the new event passe
d finds a parent that is smaller than it.
int event_queue_insert(event_queue_t event_queue, event_t event)
{
    // Variable Declaration Section
    int current = event_queue->num_of_events;
    int parent;

    // If statement will only run if the index if is trying to access a slot of the
 array less than the size of the array created
    if (current <= event_queue->size)
    {
        // Stores the event passed in function into the last slot
        event_queue->num_of_events += 1;
        current = event_queue->num_of_events;
        parent = current / 2;
        event_queue->data[current] = event;
        // While the parent does not equal slot 0, this runs
        while (parent != 0)
        {
            // If the parent has an event greater than the child, then they will be
 swapped and child becomes new parent, parent becomes new child
            // Process will continue until either it goes to beggining of array or
child is greater than the parent
            if (event_queue->data[parent]->event_time > event_queue->data[current]-
>event_time)
            {
                event_t temp_event;
                temp_event = event_queue->data[parent];
                event_queue->data[parent] = event;
                event_queue->data[current] = temp_event;
                current = parent;
                parent = current / 2;
            }
            else
            {
                break;
            }
        }
        return 0;
    }
    else
    {
        return -1;
    }
}

// Function event_queue_remove will first create a temporary event which will store
 the event at the head of the heap, which will later
// be returned at the end of the function. Then the head of the event_queue is set
to the last event on the array. After this, it enters
// a while loop which will check that the parent does not access memory not allocat
ed yet. If true, it will create another temporaroy event
// called temp_event2. Then it will check that the event_queue has been allocated w
ith an if statement and then will check if the
// event_time of the aprent is greater than the event_time of the left_child, if tr
ue, it will switch both of them. If not true, it will
// check if the right_child event_time is smaller than the new parent and if true i
t will switch them. The parent will be increased
// to check the next childs in the array and so on to keep the heap true at in all
the tree
event_t event_queue_remove(event_queue_t event_queue)
{
    // Variable Declaration Section
    event_t temp_event;
    int parent = 1;
    int current = event_queue->num_of_events;
    temp_event = event_queue->data[parent];
    event_queue->data[parent] = event_queue->data[current];
    event_queue->data[current] = NULL;
    event_queue->num_of_events -= 1;

    while (parent < event_queue->num_of_events / 2)
    {
        event_t temp_event2;
        int left_child = 2 * parent;
        int right_child = (2 * parent) + 1;
        if (event_queue != NULL)
        {
            if (event_queue->data[parent] != NULL && event_queue->data[left_child]
!= NULL)
            {
                if (event_queue->data[parent]->event_time > event_queue->data[left_
child]->event_time)
                {
                    temp_event2 = event_queue->data[parent];
                    event_queue->data[parent] = event_queue->data[left_child];
                    event_queue->data[left_child] = temp_event2;
                }
```

```
            }
            if (event_queue->data[parent] != NULL && event_queue->data[right_child]
 != NULL)
            {
                if (event_queue->data[parent]->event_time > event_queue->data[right
_child]->event_time)
                {
                    temp_event2 = event_queue->data[parent];
                    event_queue->data[parent] = event_queue->data[right_child];
                    event_queue->data[left_child] = temp_event2;
                }
            }
        }
        parent++;
    }
    return temp_event;
}

// Function event_Queue_empty will be passed an event_queue which will be checked i
f it has been allocated with memory and then will check if there has been any data
// stored in the first slot of the array, if true, it returns 0, if false it return
s -1
int event_queue_empty(event_queue_t event_queue)
{
    if (event_queue != NULL)
    {
        if (event_queue->data[1] != NULL)
        {
            return 0;
        }
        else
        {
            return -1;
        }
    }
    else
    {
        return -1;
    }
}

// Function event_queue_full will be passed an event_queue and will first chec if t
he event_queue has been allocated with memory, if true, it wil check if the number
of event equals
// the size if the array, if true it means that the data is full and cannot store m
ore items, and returns a 0, if not there is space avaialble it will return a -1
int event_queue_full(event_queue_t event_queue)
{
    if (event_queue != NULL)
    {
        if (event_queue->num_of_events == event_queue->size)
        {
            return 0;
        }
        else
        {
            return -1;
        }
    }
    else
    {
        return -1;
    }
}

// Function event_queue_finalize will enter a for loop which will free each event o
n the data and will set them to NULL until it reaches value of NULL in
// the array meaning the end. After this it will free the data of the even_queue an
d frees the event_queue
void event_queue_finalize(event_queue_t event_queue)
{
    // Variable Declaration Section
    int c1;
    for (c1 = 1; c1 <= event_queue->size; c1++)
    {
        free(event_queue->data[c1]);
        event_queue->data[c1]= NULL;
    }
    free(event_queue->data);
    event_queue->num_of_events = 0;
    event_queue->size = 0;
    free(event_queue);
}
```