DVD List Priority Queue

CPS 1000

Edward Fleri Soler

Table of Contents

1) Research

1.1) Arrays

An array is a data structure designed to hold a collection of elements of the same type. This structure is widely implemented in most high-level programming languages, and is highly efficient in storing lists of data in a sequential manner. When an array is declared, the compiler being made use of is informed of the data type to be stored in the array and the number of elements to reserve space for. The memory space required is then made available in memory, with the first index in the element (most often 0) being stored as the base address. Any following indexes may then be accessed by means of an offset. This means that if an array of 5 int elements is declared, with each int data on the specific machine occupying 4 bytes, 20 bytes of memory would be put aside and reserved for the array. If the first index may be found in memory location 1500, then the second index may be accessed at location 1504, and the third at 1508 (and so on) [1].

Arrays are traversed by means of index numbers, whereby each number refers to a separate tuple. Therefore once declared, an array may be traversed sequentially, or randomly, and have data written or read from each index.

```
int i = 0;          //declaration and initialisation
                    //of ordinary integer
int numbers[7];     //int array declaration

for(i = 0; i < 7; i++)
{
    numbers[i] = i;     //initialisation
    printf("%d ", numbers[i]);
}
```

Once ‘int numbers[7]’ is declared, contiguous space for seven integer values is set aside in memory, these locations are then traversed and populated with data through the implementation of the for-loop and the counter ‘i’. The values stored in each index are then output on screen, resulting in the following output:

0 1 2 3 4 5 6

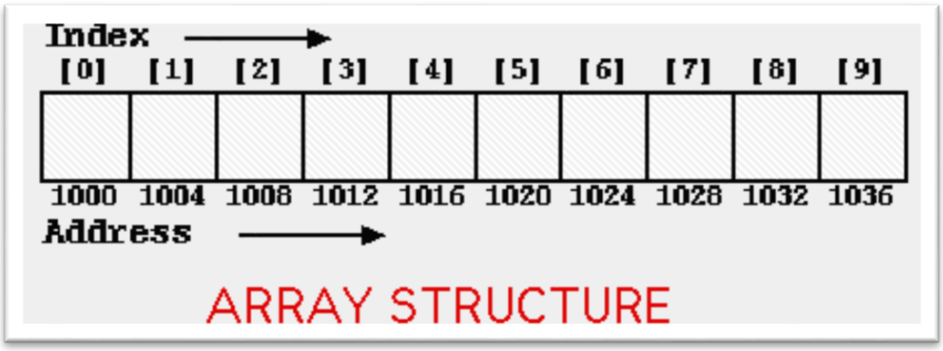


Figure 1 | Source: https://cgi.csc.liv.ac.uk/~frans/OldLectures/COMP101/week9/array.gif89

These data structures may also be implemented in the multi-dimensional domain, up to virtually infinite dimensions. These data structures may be more complex to work with, however they are based on the same principle as the linear array. A two dimensional array merely consists of a one dimensional array in each index of another one dimensional array. This means that two offsets are required to traverse and work with a two dimensional array. The first offset refers to the number of rows, or the first linear array, while the second offset refers to the number of columns, or the second linear present in each index of the first array. Therefore, when declared, each index in the first dimension is provided enough memory space to store an array of the second dimension.

```
 int i, j;        //declaration of ordinary integers
i = j = 0;       //initialisation of ordinary integers
int tdarr[3][4];     //declaration of 3 x 4 array

for(i = 0; i < 3; i++)
{
    for(j = 0; j < 4; j++)
    {
        tdarr[i][j] = i+j;      //initialisation of 3 x 4 array
        printf("%d ", tdarr[i][j]);
    }
    printf("\n")
}
```

The following is the result of this implementation:

```
0 1 2 3
1 2 3 4
2 3 4 5
```

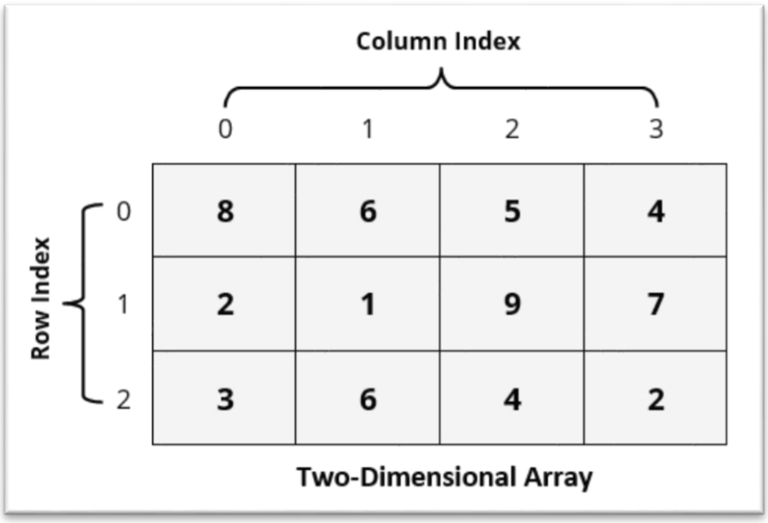


Figure 2 | source: http://i1.wp.com/www.javaforschool.com/wp-content/uploads/2013/11/two-dimensional-array-memory-address-calculation.jpg

The same principle may be applied to arrays of greater dimensions. For example a three-dimensional array may be visualised as cuboid, with the first index traversing the length, the second traversing the breadth and the third traversing the depth. Multi-dimensional arrays are great for handling large quantities of related data and therefore are widely implemented in database construction and data organisation.

One of the limitations of ordinary arrays is that the number of elements to be stored must be listed at compile time, therefore preventing the array from growing or shrinking to the demands of the program. However, a solution to this is variable length arrays, whereby the declaration of an array can replace a fixed size with a variable, whose value will later be determined. This means that the size of the array is determined at run time, as opposed to compile time. This feature is highly efficient and reduces the amount of memory wastage as, for example, if a user was allowed to create a list of up to 100 items, but only wanted to input 6, normal arrays would have to put aside memory storage for up to 100 inputs, but with variable length arrays, the program can request the size of input and assign enough memory to meet the user’s needs.

1.2) Circular Buffer

The circular buffer, also known as the ring buffer, is a First-In-First-Out (FIFO) data structure which is used for the queuing of data. The main aim of a circular buffer is to mimic a physical queue which may be found in daily scenarios, such as queuing at a bank to be seen to or waiting in line for a cup of coffee. Thus, the circular buffer is used for queuing data. It's most efficient criteria for queuing is time of entry, however adaptations allow for data to be queued based on different criteria such as priority.

The name itself allows us to visualise the queue well, however it does not illustrate the true structure of a circular buffer. Although this data type does act as a ring of memory, the actual memory locations are not defined in a circular part. Figure 3 provides a good visualisation of what a circular buffer might look like to the lay man.

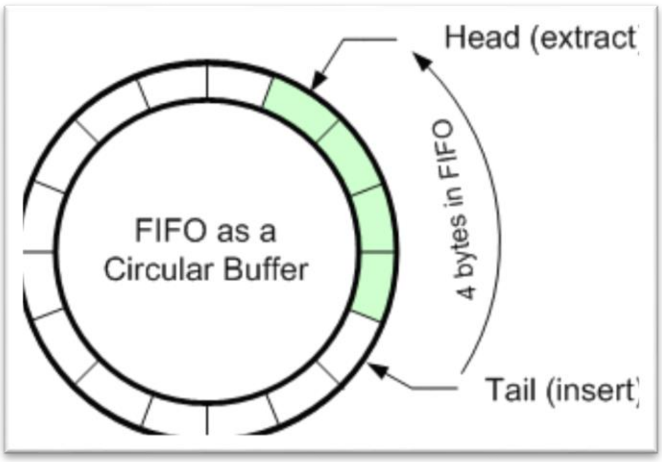


Figure 3 | source: http://polisoftdesign.com/wp-content/uploads/2010/04/Clipboard03.png

As may be seen, two pointers, being the head and tail are required to point to the locations where data must be inserted and extracted. This means that the most recent data element to be entered into the queue will be at the back of the queue, or at the tail, while when the next data element is required to be extracted, the element at the front of the queue, or at the head, is extracted. Each time an element is added or extracted, the tail and head move to point to the next free, and next occupied index in the list respectively.

Although figure 3 provides an easy visualisation of this process, the actual structure of a circular/ring buffer may be understood in figure 4 below.

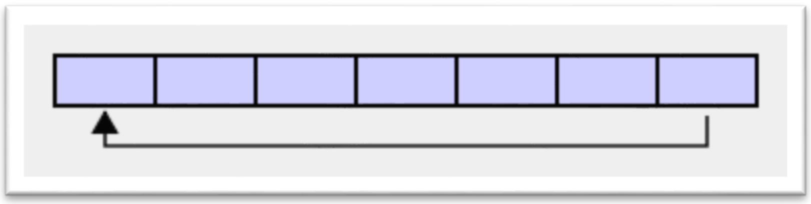


Figure 4 | Source: http://polisoftdesign.com/wp-content/uploads/2010/04/Clipboard03.png

As may be seen, a circular buffer is often implemented as being an array with connected ends or, more efficiently, a linked list, with each individual memory location being connected to the next. The head and tail pointers are still required, and the list is traversed in the same manner, however once the final (right-most) memory space is reached, the pointers must be redirected to the first (left-most) space.

The size and locations of the data elements must be kept track of so as to ensure that data elements aren't overwritten and that requests for data from an empty queue are dealt with.

Dealing with elements on a first-come-first-served basis, the circular/ring buffer is ideal for sequential population and extraction, as direct access to a required data element is not possible as is above the purpose of this data structure. This data structures strong point is that when data extraction from the queue occurs, the elements in the queue do not have to be reshuffled and reordered. However the pointers simply shift to the next elements, proving highly efficient in large queues [2].

The limitation of circular buffers is that in most scenarios, once the memory limit is set, the buffer cannot grow or shrink to the demands of the user (or system). A certain structure known as a variable length buffer may however be implemented to overcome this problem. This data structure can handle varying input or output rates and can deal with case of underflow of data (output faster than input) or overflow of data (input faster than output).

1.3) Linked List

A linked list is a data structure which overcomes the limitations of the previously mentioned data types. A linked list consists of a number of in contiguously stored nodes in memory which are linked together by means of pointers. This advantage of this is, firstly, that the data elements do not require a contiguous block of memory and therefore can be stored separately, and secondly, that the list can grow and shrink indefinitely, as there is no need for a predefined size limit on the structure. This advantage is only possible because of the first characteristic.

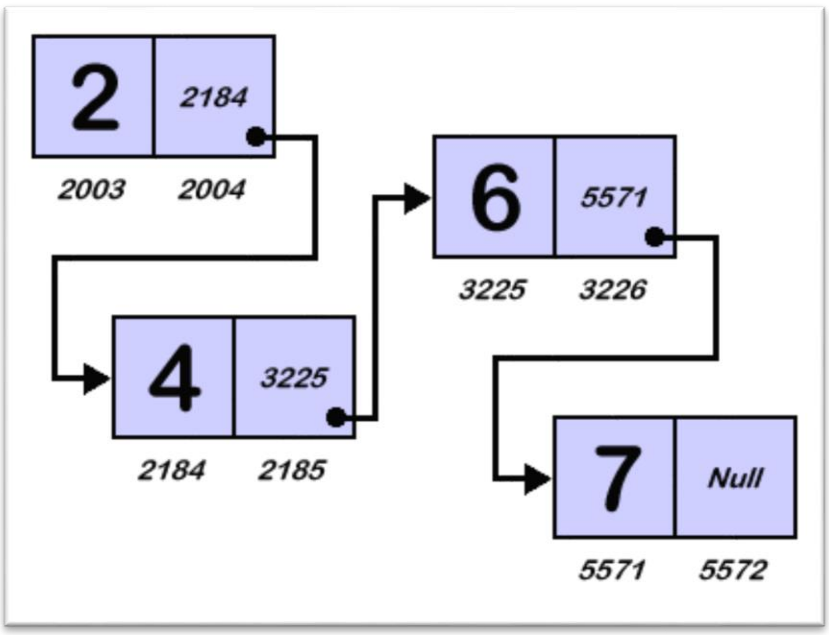


Figure 5 | Source: http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/linked_list.gif

Figure 5 above illustrates the manner in which the individual nodes are stored in a linked list. One may observe that none of the nodes are stored in a contiguous manner, however they are still linked. The compiler is given the freedom to select where the nodes are to be stored. However, for linked purposes, a small portion of memory in each node must be reserved as a pointer to point to the next node in the list, with the last element pointing to NULL to show that it is currently the last element in the list. In this manner, the nodes could be likened to the links in a chain.

The upside of this data structure is that it is highly efficient in memory allocation, as instead of defining the amount of memory required at compile time, the memory can be allocated as the program proceeds. In the C programming language, this is done through the calling of the ‘malloc()’ function. This data structure also bares the characteristic of flexibility, meaning that insertion and deletion of nodes is usually simple and can be adapted to queue data for different scenarios.

However efficient, this data structure is still one of sequential access, and therefore accessing and operating upon a certain node in the list requires traversing the list until that node is found. This inefficiency is heightened by the difficulty of traversing a single linked list in reverse. This is due to the fact that in single linked lists, each node has a pointer to the next node but no pointer to the previous node. A solution to this is the implementation of a double linked list (Figure 6 below), which works exactly like a single linked list, but each node also contains a pointer to the previous node. This therefore allows for easy reverse traversal of the list but requires more data in each node to be assigned to pointers [3].

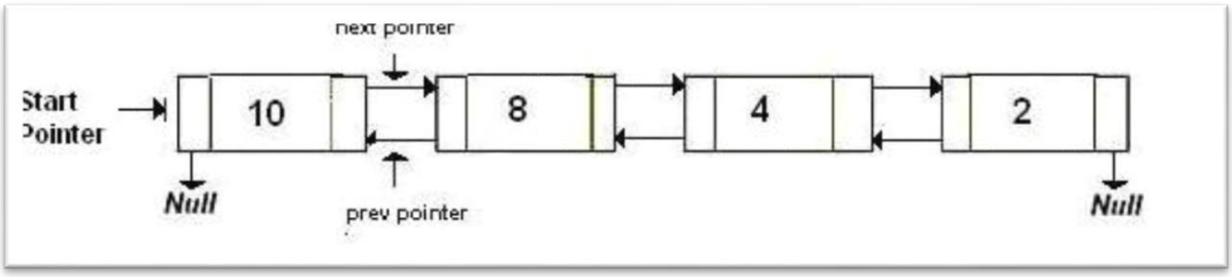


Figure 6 | Source: http://www.geeksforgeeks.org/wp-content/uploads/DLL.jpg

Linked lists are of great use when it comes to dynamically queuing elements for different scenarios. Being so flexible, this data structure could be easily implemented to queue elements based on certain characteristics such as priority. The operations to add or remove a node from the list merely require the pointers of the previous (and possibly next) node to be altered, and therefore each element in the list does not have to be reshuffled and reordered.

1.4) Binary Heap

A binary heap is a data structure based on the principles of binary trees. This means that data elements are stored in an ordered fashion and adhere to the rules governing binary trees. Two main binary heaps may be implemented; the max-heap or the min-heap. The max-heap tree has elements with the greatest deciding value at the top of the tree, while the nodes lower down in the tree will have a smaller deciding value. A min-tree works in the opposite manner.

Binary heaps are highly suitable in prioritising elements, as elements are listed in order of priority. However, when implementing this data structure, nodes are not stored in a linear, list like fashion but are linked together in levels. The main rules which govern binary heaps are that each node can have a maximum of two child nodes and each node can have no more than one parent node, while the root node has no parent nodes and is therefore the highest element in the tree.

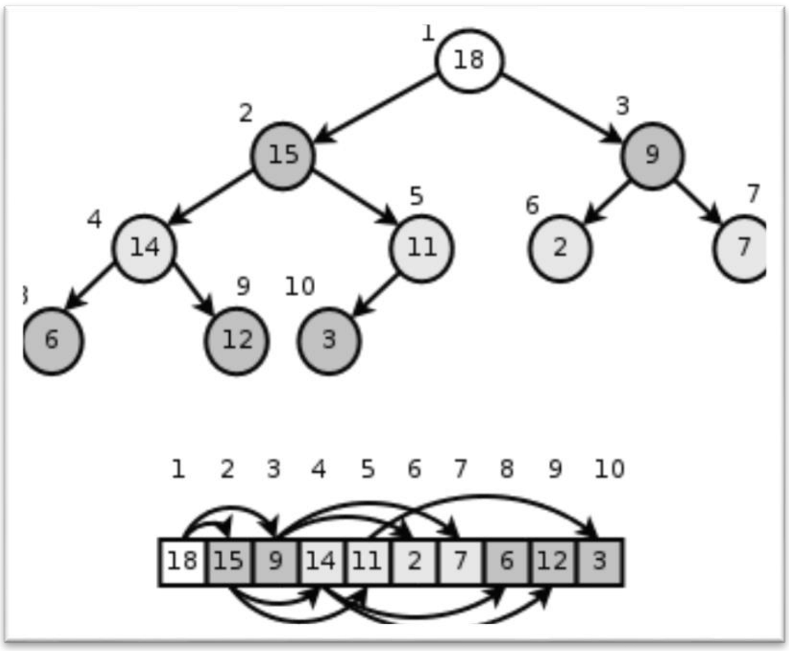


Figure 7 | Source: http://cs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/module2/images/Heaps_HeapStructure.png

Figure 7 above illustrates the structure of the binary heap. This is a max-heap, and as may be seen, each child node is smaller than (but can be equal to) their respective parent node. One may also notice the manner in which a binary heap may be implemented through the use of an array data structure.

Each element in the tree is stored within the array, with the root node being stored at index 1 (index 0 is left void for practical purposes) and subsequent nodes being stored after it. As the tree is traversed, one may notice that each level contains twice as many nodes as the previous level (if each level is completely full). Therefore, within the array, each parent will have two pointers to their child nodes.

The manner in which the tree is structured allows for highly efficient traversal of the tree. For example, for a parent to access its left child nodes index, if the parent is stored in index n, the left child node would be found in index 2n and the right child node will be found in index 2n+1. Also, accessing the parent of a left child node n would be possible by accessing index n/2 while accessing the parent of a right child node n would be possible by accessing (n-1)/2. This structures layout makes the binary heap highly efficient to traverse. Furthermore, being readily sorted allows for efficient adding and deleting of nodes to/from the heap [4].

Adding a node to the tree involves adding it to the lowest level of the tree, comparing it to its parent node and swapping if the order is broken. If the order is not broken then the node is in its correct place. This operation has a time complexity of O(log(n)), making it highly efficient when compared to other sorting algorithms.

Since the binary heaps biggest adaptation is in priority queues, removal of the highest priority node in the tree (the root) is fairly simple. The root is first removed and replaced with the lowest priority element. The new root is compared with its children and is rearranged if the order is broken. This is repeated until the root is in the correct place and order is restored. This operation also boasts a time complexity of O(log(n)).

Binary heaps are highly efficient in ordering nodes according to a certain characteristic and therefore are widely adopted in the creation of priority queues. The manner in which new nodes are sorted in the list makes this data structure the best of the four mentioned for the implementation of a priority queue.

2) Program Source Code

2.1.1) Linked List Header File

The following lines of code were implemented in the header file of the linked list project. This file contains the prototypes for each of the functions to be defined in the function file. The priority_queue and node types were also defined in this file to be used in the function file and client application.

The node type was defined as a structure which holds a number of data items ranging from film rating, to title and release date which would be filled in by the user. Noteworthy is the fact that these data items could easily be altered so as to use this application for a different scenario, such as an emergency room priority queue, or any other priority queue as a matter of fact. The 'next' pointer within the node type was defined to perform a recursive role and point to the next node in the queue. We will cover this in greater detail later on.

The priority_queue type was defined as a structure that holds a queue name, queue size and queue maximum size, among other data items. These are all characteristics which will be determined by the user. The final two data items, being the head and tail pointer serve a structural purpose. Both declared as pointers to type Node, these pointers will point to the first and last elements in the linked list respectively.

The prototypes following the type definitions will all be declared in the function file, and will serve the purpose of informing the compiler that a function with certain parameters and return types will be declared.

```
#ifndef LINKED_LIST_H_INCLUDED
#define LINKED_LIST_H_INCLUDED

#include <stdbool.h>
#define MAXSIZE 40
#define SIZE 81

typedef struct node{     double priority;
                         char title[MAXSIZE];
```

```
                            char genre[MAXSIZE];
                            int length;
                            char release_date[MAXSIZE];
                            char director[MAXSIZE];
                            char actors[MAXSIZE];
                            struct node * next;   //pointer to next node in queue
                        } Node;

typedef struct Q{    char qname[MAXSIZE];
                     int size;
                     int max_size;
                     Node * head;  //pointer to first node in queue
                     Node * tail;  //pointer to last node in queue
                }priority_queue;

/* operation: create a new empty priority queue with maximum number of
elements max_size*/
/* postcondition: new queue with a maximum size of 'max_size' created*/
/*                returns NULL if creation fails*/
priority_queue * create_q(int max_size);

/* operation: enqueue 'node' in priority queue 'queue' with priority
'priority'*/
/* precondition: Node 'node' and possibly empty or non-empty queue 'queue'
*/
/* postcondition: placement of 'node' in 'queue' at point of priority if
return true*/
/*                else failed to enqueue item*/
bool enqueue(priority_queue * queue, Node * node, double priority);

/* operation: dequeue element with highest priority from 'queue'*/
/* precondition: non-empty priority_queue 'queue' */
/* postcondition: return of pointer to highest priority 'node' in 'queue'
and removal of node from 'queue'*/
/*                returns NULL if dequeue was unsuccessful (queue is empty)
*/
Node * dequeue(priority_queue * queue);

/* operation: return highest priority 'node' in 'queue' without removal*/
/* precondition: non-empty priority queue 'queue'*/
/* postcondition: return of pointer to highest priority 'node' in 'queue'*/
/*               without removal of 'node' - returns NULL if queue is empty
*/
Node * peek(const priority_queue * queue);

/* operation: returns true if 'queue' is empty*/
/* precondition: possibly empty queue 'queue' */
/* postcondition: returns true if 'queue' contains 0 elements, false
otherwise*/
bool is_empty(const priority_queue * queue);

/* operation: returns the number of elements in 'queue'*/
/* precondition: possibly empty priority_queue 'queue' */
/* postcondition: returns the number of nodes in 'queue'*/
int size(const priority_queue * queue);

/* operation: creates new priority_queue from elements in queues q1 and q2*/
/*            without altering their contents*/
/* precondition: two queues 'q1' and 'q2'*/
/* postcondition: return of pointer to priority_queue containing elements 
from 'q1' and 'q2'*/
/*                without effecting 'q1' and 'q2'*/
priority_queue * merge(const priority_queue * q1, const priority_queue * q2);

/* operation: remove all elements in priority_queue 'queue'*/
/* precondition: non-empty priority_queue 'queue' */
/* postcondition: returns true if queue is successfully cleared, false 
otherwise*/
bool clear(priority_queue * queue);

/* operation: store priority_queue 'queue' in file 'f'*/
/* precondition: non-empty priority_queue 'queue'*/
/* postcondition: returns true if queue is successfully written to file, 
false otherwise*/
bool store(priority_queue * queue, char * f);

/* operation: load priority_queue 'queue' from file 'f'*/
/* precondition: file 'f' containing priority_queue 'queue'*/
/* postcondition: returns true if queue is successfully read from file, false 
otherwise*/
bool load(priority_queue * queue, char * f);

/*operation: Read string input*/
char * s_gets(char * st, int n);

#endif // LINKED_LIST_H_INCLUDED
```

2.1.2.1) Linked List Function File

The following lines of code were implemented in the Linked List function file. Here the functions to be used in the client application were defined and built. The header file containing the prototypes and type definitions was called through the '#include "linked_list.h" ' so that the prototypes and the actual functions are linked in the pre-compilation stage. 'node', of type 'Node', and 'queue', 'q1', 'q2', and 'q3', of type 'priority_queue' were declared as static variables with a global scope. This was due to the fact that these variable and the data they hold would be used in both this file and in the client application.

Also two functions, being the 'dequeue_no_return' and the 'create_q_no_name' have been defined as being local functions. This is due to the fact that their use was limited to this file as they simply acted as functions to be used inside other functions. Therefore they have been assigned file scope to prevent their use in other files.

```
#include<stdio.h>
#include<time.h>
#include<stdbool.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
#include "linked_list.h"

//-------------------- DECLERATION OF STATIC VARIABLES --------------------
extern Node node;
extern priority_queue queue, q1, q2, q3;

//----------------------- LOCAL FUNCTION PROTOTYPES ----------------------
bool dequeue_no_return(priority_queue * queue);
bool create_q_no_name(int max_size, char * qname, priority_queue * queue);

//---------------------------- LOCAL FUNCTIONS ----------------------------
bool create_q_no_name(int max_size, char * qname, priority_queue * queue)
//creates a queue without asking for user input for name
{
    strcpy(queue->qname, qname);

    queue->size = 0;
    queue->max_size = max_size;
    queue->head = queue->tail = NULL;

    return true;
}

bool dequeue_no_return(priority_queue * queue)
```

```
//removes greatest priority element from the queue without outputting the
data
{
    Node * tempn;

    if(is_empty(queue))
    {
        printf("Queue is empty\n\n");
        return false;
    }

    tempn = queue->head;
//point tempn to front of queue

    if(size(queue) == 1)
    {
        queue->head = queue->tail = NULL;
//point head to NULL as queue is empty
    }
    else
    {
        queue->head = queue->head->next;
//point head to next element in queue
    }

    queue->size--;
    free(tempn);
//free highest priority element in queue

    return true;
}

//-------------------------- EXTERNAL FUNCTIONS -------------------------
priority_queue * create_q(int max_size)
//creates a new priority queue
{
    priority_queue * pq;
    pq = malloc(sizeof(priority_queue));
//allocate memory for new priority queue

    fflush(stdin);
    printf("Please enter the name of the queue that you would like to
create:\n\n");

    s_gets(pq->qname, MAXSIZE);

    if(pq->qname[0] == '\0')
    {
        printf("Invalid input\n\n");
        s_gets(pq->qname, MAXSIZE);
        fflush(stdout);
    }


    pq->size = 0;
    pq->max_size = max_size;
    pq->head = pq->tail = NULL;
    return pq;
}

int size(const priority_queue * queue)
//returns number of elements currently in queue
{
    return queue->size;
}

bool is_empty(const priority_queue * queue)
//returns true if queue is empty
{
    return queue->size == 0;
}

bool clear(priority_queue * queue)
//clears elements in queue
{
    while(!is_empty(queue))
    {
        dequeue_no_return(queue);
    }

    if(is_empty(queue))
    {
        return true;
    }
    else
    {
        return false;
    }
}

Node * dequeue(priority_queue * queue)
//removes highest priority element in queue and returns data
{
    Node * tnode, * temp;
    Node rnode;

    if(is_empty(queue))
    {
        printf("Queue is empty\n\n");
        return NULL;
    }

    tnode = queue->head;        //point tnode to highest priority element
    rnode = *tnode;       //copy data of highest priority element for return

    if(size(queue) == 1)
    {
        queue->head = queue->tail = NULL;
//point head to NULL as queue is empty
    }
    else
    {
        queue->head = queue->head->next;        //point head to next element
    }

    queue->size--;

    free(tnode);            //free highest priority element in queue

    temp = &rnode;

    return temp;           //return copy of data of highest priority element
}

priority_queue * merge(const priority_queue * q1, const priority_queue * q2)
//merge contents of two queues into a new queue
{
    int newmax, i;
    newmax = i = 0;

    priority_queue newq;
    priority_queue * tqueue;   //memory allocation for new queue to be created

    Node * tempn;

    newmax = q1->max_size + q2->max_size;
//determine maximum size of new queue

    tqueue = create_q(newmax);
//create queue to hold maximum of newmax elements
    newq = *tqueue;

    tempn = q1->head; //point tempn to head of q1

    for(i = 0; i < q1->size; i++)
//N.B: If both queues are empty, the new queue will be empty but will have
a max_size of newmax
    {
        enqueue(&newq, tempn, tempn->priority);
//enqueue newq with elements of q1
        tempn = tempn->next;
    }

    tempn = q2->head; //point tempn to head of q2

    for(i = 0; i < q2->size; i++)
    {
        enqueue(&newq, tempn, tempn->priority);
//enqueue newq with elements of q1
        tempn = tempn->next;
    }

    if(size(&newq) == (size(q1) + size(q2)))
//test that all elements have been copied
    {
        tqueue = &newq;
        return tqueue;
    }

    printf("Merge operation failed\n\n");
    return NULL;
}

bool enqueue(priority_queue * queue, Node * node, double priority)
//enqueue elements in queue
{
    Node *prev, *pnew;
    int i = 0;

    prev = NULL;

    pnew = malloc(sizeof(Node));
//allocate memory for new node to be queued

    *pnew = *node;

    if(size(queue) == queue->max_size)  //catch if queue is full
    {
        printf("Queue is full\n\n");
        return false;
    }

    if(is_empty(queue))                  //catch if queue is empty
    {
        pnew->next = NULL;                  //as new node is final element
        queue->head = queue->tail = pnew;   //point to new node
        queue->size++;                      //increment queue size

        return true;
    }

    if(size(queue) == 1) //special case
    {
        if((queue->head->priority) >= priority)
//if priorities are equal FCFS basis due to >=
        {
            pnew->next = NULL;              //as new node is final element
            queue->head->next = pnew;       //as new node is second node
            queue->tail = pnew;             //as new node is final element
            queue->size++;                  //increment queue size

            return true;
        }
        else
        {
            pnew->next = queue->head; //as new node is first element
            queue->head = pnew;       //as new node is the head of the queue
            queue->size++;            //increment queue size

            return true;
        }
    }

    //more than 1 element in queue

    if(pnew->priority > queue->head->priority)
//new node has highest priority
```

```
    {
        pnew->next = queue->head;
        queue->head = pnew;
        queue->size++;

        return true;
    }

    if(priority < queue->tail->priority)
//new node has lowest priority
    {
        queue->tail->next = pnew;
        queue->tail = pnew;
        pnew->next = NULL;
        queue->size++;

        return true;
    }

    pnew->next = queue->head->next;         //make node 2nd element in queue
    queue->head->next = pnew;

    if(queue->size == 2)
//by elimination if there are 3 elements and the new one is not the head or
the tail - it is the middle
    {
        queue->size++;
        return true;
    }

    prev = queue->head;                      //for operation to come

    while(i < ((queue->size) - 2))           //traverse queue
    {
        if(priority <= pnew->next->priority)    //sorting algorithm
        {
            prev->next = pnew->next;
            pnew->next = pnew->next->next;
            prev->next->next = pnew;
            prev = prev->next;
        }
        else
        {
            queue->size++
            return true;
        }
        prev = prev->next;
        i++;                      //increment i
     }

    printf("Something went wrong - element not queued\n\n");

    return false;
}

Node * peek(const priority_queue * queue)
//return data of highest priority element without removing it from queue
```

```
{
    if(is_empty(queue))
    {
        printf("Queue is empty\n\n");
        return NULL;
    }

    return queue->head;         //highest priority element
}

bool store(priority_queue * queue, char * fname)
//write queue contents to file
{
    FILE * fp;
    Node tempn;
    Node * temp;
    int i, tsize;
    i = tsize = 0;

    if((fp = fopen(fname, "w+b")) == NULL)
//attempt to open 'fname' in write plus binary mode
    {
        printf("Unable to open file '%s'\n\n", fname);
        return false;
    }

    rewind(fp);         //rewind to start of file

    fwrite(queue, sizeof(priority_queue), 1, fp);
//write queue details such as max_size and name to file

    tsize = queue->size;        //temporarily store queue size

    for(i = 0; i <tsize; i++)                   //traverse queue
    {
        temp = dequeue(queue);
//dequeue nodes from queue for saving
        tempn = *temp;
//required as dequeue returns a node pointer
        fwrite(&tempn, sizeof(Node), 1, fp);
//write each node in the list to file
    }

    if(fclose(fp) != 0)
    {
        printf("Unable to close file '%s'\n\n", fname);
        return false;
    }

    return true;
}

bool load(priority_queue * queue, char * fname)
//read queue contents from file
{
    FILE * fp;
    Node ptr;
    int i, temps;
    i = temps = 0;

    if((fp = fopen(fname, “rb”)) == NULL)
//attempt to open file in read binary mode
    {
        printf(“Unable to open file ‘%s’\n\n”, fname);
        return NULL;
    }

    rewind(fp);         //rewind to start of file

    fread(queue, sizeof(priority_queue), 1, fp);  //read queue data from file

    temps = queue->size;
//temporarily store queue size

    for(i = 0; i <temps; i++)                   //traverse queue
    {
        fread(&ptr, sizeof(Node), 1, fp);
//read individual nodes from file
        queue->size = i;
//for enqueueing – simulates new nodes being added to a queue
        enqueue(queue, &ptr, ptr.priority);
//enqueue nodes in queue
    }

    queue->size = temps;                        //restore correct queue size

    if(fclose(fp) != 0)
    {
        printf(“Unable to close file ‘%s’\n\n”, fname);
        return false;;
    }

    return true;
}

char * s_gets(char * st, int n)             //use for string input
{
      char * ret_val;
      int i = 0;

      ret_val = fgets(st, n, stdin);

      if (ret_val)
      {
            while (st[i] != ‘\n’ && st[i] != ‘\0’)
            {
                  i++;
            }

            if (st[i] == ‘\n’)
            {
                  st[i] = ‘\0’;
            }
            else
```

```
		{
			while (getchar() != ‘\n’)
			{
				continue;
			}
		}
	}
	return ret_val;
}
```

2.1.2.2) Source Code Explanation

- bool create_q_no_name(int max_size, char * qname, priority_queue * queue)

The purpose of this local function is to generate a new queue with a maximum size of 'max_size' name 'qname'. The queue to be created is passed as a parameter and already has memory allocated to it therefore there is no need to call 'malloc()'. This functions main use is within the load function, and therefore it will be better understood later on. The function returns true if it runs to completion.

- bool dequeue_no_return(priority_queue * queue)

This function is also a local function and serves an internal purpose. The queue pointer, passed as a parameter will be accessed and will have the element with the highest priority removed. The queue is first tested to be empty and a warning is displayed if it is. The queue is then dealt with according to its present size. If the queue has a size of 1 before being dequeued, then the head and tail pointers are set to point to NULL. Otherwise the head pointer is set to point to the next element in the list through the statement:

```
queue->head = queue->head->next;
```

The node being dequeued then has the memory it was allocated set free. The function returns true if the dequeue was successful and returns false if any problems where encountered.

- priority_queue * create_q(int max_size)

This function is designed to create a new queue to be queued and manipulated by the user. The parameter being passed is the maximum number of elements that the queue should hold. The following lines of code first declare a queue pointer named 'pq' and then allocate it memory manually:

```
priority_queue * pq;
pq = malloc(sizeof(priority_queue));
```

The user is then asked to enter the name that they would like to call the queue. User input is tested to ensure that the data entered is valid.

The maximum size limit of the queue is then set to hold the value passed by the parameter ‘max_size’ and the current size of the queue is set to 0 as it is currently empty. The head and tail pointers are also set to point to NULL to signify that there are currently no elements in the queue. The pointer ‘pq’ is then returned to be made use of in the client application.

- int size(const priority_queue * queue)

This function takes a constant priority_queue pointer as its parameter. The queue pointer is declared as being constant to ensure that data will simply be read from the parameter and that nothing will be altered. This function simply accesses the size pointer defined within the priority_queue type and returns the value.

- bool is_empty(const priority_queue * queue)

Similarly to the previous function, this function takes a constant priority_queue pointer as its parameter and accesses the queue size. If the queue contains a size value of 0 it returns true, otherwise it returns false.

- bool clear(priority_queue * queue)

This functions purpose is to receive a pointer to a queue to which it must clear all nodes. Therefore the queue data such as size and name must be left untouched but the elements queued must be removed. The first while loop keeps running until the loop is empty. This is tested by calling the ‘is_empty’ function explained above. If the loop is not empty the ‘dequeue_n_return’ function (explained earlier) is called. This function removes the highest priority element from the queue without returning any node data. Therefore each time the loop is run, the node at the front of the queue is removed.

This is run till the queue is empty. The queue is then tested again to check whether it is empty by means of the 'is_empty' function. If the queue is now empty, the return will be true, otherwise the function will return false, signifying that an error has occurred.

- Node * dequeue(priority_queue * queue)

This function is implemented to dequeue the highest rated node from the queue and return it to be used in the client application. It accepts a priority_queue pointer as its parameter, which it will then access and dequeue. The queue is first tested to be empty, in which case the function will return a NULL pointer, which will be tested for in the client application and which will show that the dequeue did not succeed. If the queue is not empty, a Node pointer called tnode is declared and is set to point to the head of the queue. The value pointed to by tnode is then copied to a temporary node which we shall later use to return the node to return the node data.

```
If(size(queue) == 1)
{
     queue->head = queue->tail = NULL;
}
else
{
    queue->head = queue->head->next;
}
```

These lines of code test whether the queue shall be empty following the dequeue. If this is the case, then the head and tail pointer are set to point to NULL, signifying that the queue is empty. If not, the head pointer is set to point to the next element in the queue, now the element with the highest priority. The size of the queue is then decremented and the node pointer previously set to point to the head is freed, releasing the data which was previously allocated. The data which was copied from the head pointer is then returned, to be used within the client application.

- bool enqueue(priority_queue * queue, Node * node, double priority)

This function is designed to enqueue a queue with a node. The parameters include a pointer to the queue to be enqueued, a pointer to the node containing the data which shall be enqueued, and the priority of the element to be enqueued. The priority will be needed to position the node in the right place within the queue.

```
Pnew = malloc(sizeof(Node));
*pnew = *node;
```

The first step in enqueueing the node is to allocate memory for the new node in the queue. ‘pnew’, declared as a node pointer, is allocated the memory space of a Node. A copy of the data passed as a parameter is made and is stored in the newly allocated memory space pointed to be ‘pnew’. ‘pnew’ must now be enqueued within the queue.

The first case is one where the queue is empty. This case would result in ‘pnew’ being the only node in the list. Therefore the head and tail of the queue must both point to ‘pnew’, and ‘pnew’ must point to NULL, as it is the last element in the list. The queue size must then be incremented, showing that there is now one element in the list. The function then returns true so as to show that it has successfully queued ‘node’ in ‘queue’:

```
if(is_empty(queue))
    {
        pnew->next = NULL;
        queue->head = queue->tail = pnew;
        queue->size++;
```

```
            return true;
        }
```

The second case is if there is one element in the queue already. In this case, the single element in the list could be compared to the node to be added and they can be reshuffled according to preceding priority:

```
if(size(queue) == 1) //special case
    {
        if((queue->head->priority) >= priority)
         {
            pnew->next = NULL;
            queue->head->next = pnew;
            queue->tail = pnew;
            queue->size++;

            return true;
        }
        else
        {
            pnew->next = queue->head;
            queue->head = pnew;
            queue->size++;

            return true;
        }
    }
```

The element at the head of the queue (being the only element) is compared to the new node in terms of priority. If the new node has a priority greater or equal to that of the new node, then the new node shall be placed behind it. If the node is of the same priority, it is anyway placed behind the current node as the second determining factor is that of First-Come-First-Served.

Thus, if the new node is to be placed at the back of the queue, its next pointer is set to NULL, the next pointer of the head is set to point to the new node ‘pnew’, the tail is also set to point to ‘pnew’ as it is the last element, and the size of the queue is incremented. The function then returns true.

If, however, the current size of the queue is 1 and the new node has a greater priority, the new node is placed in front of the current head. Therefore, the next pointer in ‘pnew’ is set to point to the current head, the head pointer is altered to point to pnew (the new head) and the size of the queue is incremented, resulting in a successful routine, thus the return of the function is true.

The next case to consider is when there is more than one element within the queue. Firstly, the new node ‘pnew’ will be compared to the head of the queue. If it has a greater priority it will be placed at the head, in front of the previous head. If not it will be compared to the tail of the queue. If ‘pnew’ holds a priority lower than or equal to the tail (due to first-come-first-served), then pnew will be placed at the tail of the queue. If neither of these cases result as true, then the new node must be placed at some point within the queue between the head and the tail.

Therefore the penultimate test is to check whether the current queue size is 2. If this results as true, then the new node must definitely be behind the head but in front of the tail, meaning that it will be placed as the second element in a queue of 3. These mentioned tests are tested through the following lines of code:

```
if(pnew->priority > queue->head->priority) //test against head
{
    pnew->next = queue->head;              //place at head of queue
    queue->head = pnew;
    queue->size++;
    return true;
}

if(priority <= queue->tail->priority)  //test against tail
{
    queue->tail->next = pnew;         //place at tail of queue
    queue->tail = pnew;
    pnew->next = NULL;
    queue->size++;
    return true;
}

pnew->next = queue->head->next;    //place as second element in the queue
```

```
queue->head->next = pnew;

if(queue->size == 2)  //if queue size is 2 then it is sorted
{
    queue->size++;
    return true;
}                   //else, continue...
```

If the size of the queue is greater than 2, and the new node is of less priority than the head but of greater priority than the tail, then it is left as the second element in the queue before its final position is found.

```
Prev = queue->head;
while(i < ((queue->size) - 1))
{
    if(priority <= pnew->next->priority)
    {
        temp = pnew->next;
        pnew->next = temp->next;
        temp->next = pnew;
        prev->next = temp;
    }
    else
    {
        queue->size++
        return true;
    }
    prev = prev->next;
    i++;                      //increment i
}
```

This final test will find the nodes place in the queue if the other tests all failed. Firstly, a previously declared Node pointer ‘prev’ is set to point to the head of the queue. The while loop will loop for one time less than the number of elements in the queue. The priority of the new node is compared to the priority of the next node in the list. If the priority of the node to be enqueued is less than or equal to that of the next node then they are swapped. ‘prev’ is then set to point to the next pointer (currently second node in the list), ‘i’ is incremented and the cycle repeats.

This repeats until ‘pnew’ is placed in front of a node with a smaller priority, and thus ‘pnew’ is in the correct place. Once this occurs, the queue size is incremented and the function returns true.

It is solely due to the fact that all previously queued nodes are in order, that only ‘pnew’ needs to be compared to the other nodes in the list. This means that each time a node is added to the list, its correct place is located and it is put there. This idea of queuing the nodes in order of priority is much more efficient than queuing them in a first-come-first-served basis and then sorting them in order of priority each time an operation on the list must be executed.

- Node * peek(const priority_queue * queue)

  This function is designed to return a pointer to the node at the head of the priority queue. It differs from the dequeue function as the node itself is not deleted. Firstly, the queue is tested to be empty through the ‘is_empty’ function. If this turns out to be true, then a NULL pointer is returned, signifying failure of the function. Otherwise, a pointer to the node at the head of the queue is returned.

- priority_queue * merge(const priority_queue * q1, const priority_queue * q2)

  This function is designed to receive pointers to two queues which will be merged into one. Firstly a pointer to a new queue called ‘tqueue’ is created, and will later point to the newly merged queue.

```
Newmax = q1->max_size + q2->max_size;
tqueue = create_q(newmax);
```

  ‘newmax’, a previously declared int, is set to store the sum of the sizes of the queues to be merged. ‘newmax’ is then passed as an argument to the create_q function, which will be used to create the new queue. ‘tqueue’, the previously defined pointer stores the return pointer from create_q, and therefore is now pointing to the newly merged, yet empty, queue.

The new queue must now be populated with the nodes from the queues ‘q1’ and ‘q2’. Firstly, newq, of type priority_queue, stores a copy of the data in tqueue. And the Node pointer tempn is set to point to the head of q1.

A for loop is then set to run the number of times equal to the size of q1. Each time the loop is run, the value at the head of q1 is enqueued in newq. Therefore, the enqueue function is made use of to populate the new queue with data:

```
     newq = *tqueue;
    tempn = q1->head; //point tempn to head of q1

//N.B: If both queues are empty, the new queue will be empty but will have
a max_size of newmax
    for(i = 0; i < q1->size; i++)
    {
        enqueue(&newq, tempn, tempn->priority);
        tempn = tempn->next;
    }
```

Once the end of the while loop has been reached, all the elements from q1 have been queued and sorted in ‘newq’. The exact same procedure is repeated again, this time populating ‘newq’ with the nodes from q2. In this manner, modularity is being exercised, as the enqueue function is handling the enqueueing and sorting of the nodes within the newq.

Finally the newq size is compared to the sum of the sizes of queues ‘q1’ and ‘q2’:

```
        if(size(&newq) == (size(q1) + size(q2)))
        {
                tqueue = &newq;
                return tqueue;
        }
```

If the sizes match, then the queue has been successfully created and populated. Therefore newq is pointed to be tqueue again and tqueue is returned. Otherwise the function returns NULL, signifying that it has failed to successfully merge the two queues.

- bool store(priority_queue * queue, char * fname)

The purpose of this function is to write queue data to file to be stored. A pointer to the queue to be stored, as well as the name of the file to store the queue in, are passed as parameters.

Firstly, the file is opened in ‘w+b’ mode being write plus in binary. This allows for data from the file to read as well as written in binary form. Binary mode is preferred as it allows for more portable code which could run on different machines.

```
If((fp = fopen(fname, “w+b”)) == NULL)
{
      printf(“Unable to open file ‘%s’\n\n”, fname);
      return false;
}
rewind(fp);
```

These lines of code attempt to open the file mentioned by the user in ‘w+b’ mode and store its location in the previously declared FILE pointer ‘fp’. If this was unsuccessful the return from the fopen function would be NULL, and therefore if ‘fp’ was found to point to NULL, then the function is exited with a return of false. Otherwise, the file pointer ‘fp’ is set to point to the start of the file using the ‘rewind’ function. This will result in any previously written data to be overwritten. This is due to the fact that since the positions of nodes in the queue may have shifted, it is problematic to append data to the end of the file.

The queue data, including the size, name and max_size, are then written to file by implementing the ‘fwrite’ function:

```
fwrite(queue, sizeof(priority_queue), 1, fp);
```

This writes the whole queue data to file in one chunk. Although the file data has been written, including the head and tail pointer, the actual nodes must also be written to file. This is due to the fact that even though the head and tail pointer may be stored in file, when they are later reloaded, the addresses to which they point will now most likely not be those of the nodes at the head and tail of the queue, but some other data which has been allocated the memory which was previously that of these nodes. Therefore, although tedious, it is required to write each and every individual node to file.

```
tsize = queue->size;
for(i = 0; i <tsize; i++)
{
      temp = dequeue(queue);
      tempn = *temp;
      fwrite(&tempn, sizeof(Node), 1, fp);
}
```

Firstly, the size of the queue is stored in a temporary variable. The queue is then traversed by means of a for loop which runs a number of times equal to the number of nodes in the queue. The dequeue function is used to remove and return the highest priority element from the list. The node being returned is pointed to by the Node pointer ‘temp’. The value of temp is then copied into ‘tempn’, which is then written to file in one block. This cycle is continued, until the queue is emptied out and all nodes have been written to file.

```
If(fclose(fp) != 0)
{
        printf("Unable to close file '%s'\n\n", fname);
        return false;
}
```

An attempt to close the file pointed to by ‘fp’ is then made. If this is unsuccessful, then the function will return false. Otherwise the file will be closed and the function will return true.

- bool load(priority_queue * queue, char * fname)

The purpose of this function is to read the stored data from file and load it into a queue to be used by the user. The function accepts a pointer to the queue in which the data shall be loaded, as well as the file name of the file containing the required data.

The file is first attempted to be opened, as was explained above, but this time in ‘rb’, or read in binary, mode. ‘rb’ mode allows for data to be read from the file, but does not permit data to be written. This protects the files contents from being corrupted while it is open. Also, since the file was written to in binary, it must be read from in binary. The compiler will handle the translation of the data from binary into understandable form.

Once the file is opened and pointed to by ‘fp’, as was implemented in the store function, fp is set to point to the start of the file so that the reading process may begin. Data is read from the file into the queue pointer by means of the fread function:

```
fread(queue, sizeof(priority_queue), 1, fp);
```

This reads one chunk of data equal to the size of a priority_queue data type from the file to the location pointed to by queue. The size, max_size and name values have now been successfully read, but as was previously mentioned, the head and queue pointers must be updated, as they currently point to garbage.

```
Temps = queue->size;
for(i = 0; i <temps; i++)
{
      fread(&ptr, sizeof(Node), 1, fp);
      queue->size = i;
      enqueue(queue, &ptr, ptr.priority);
}
queue->size = temps;
```

Here, the queue size is temporarily stored in a variable labelled ‘temps’, which is used to run a for loop for a number of cycles equal to the queue size. With each cycle, a chunk of memory equal to the size of a Node data type is read from file into a Node variable which was defined earlier. The size of the queue is then incremented and the enqueue function is then called to enqueue the node read into ‘ptr’ within the queue pointed to by ‘queue’.

This is repeated until all nodes are read from file to the queue, at which point the queue has now been successfully loaded. The original size of the queue is then restored. The file is now attempted to be closed, as was previously seen. If this attempt fails then the function will return false, otherwise the function will return true.

- char * s_gets(char * st, int n)

This function is implemented when string input is requested from the user. The character pointer ‘st’ and the integer ‘n’ are passed as arguments. Data input by the user is then stored in pointer ‘st’ up to the ‘n’th character. This is performed through the ‘fgets’ function which reads data from the keyboard buffer. The downside of the ‘fgets’ function is that the newline character pressed by the user, which is needed to flush the buffer, is also stored in ‘st’. Therefore, the ‘s_gets’ function traverses the character input until it finds the newline character, this newline character is then replaced by the end of line character ‘\0’, resulting in a string without the newline character at the end. This is often convenient, especially when storing data within databases or other storage systems such as this.

2.2.1) Circular Buffer Header File

As was the case with the linked list header file, the circular buffer implementation also requires a header file. Within this file, two data types, namely the ‘Node’ data type and the ‘priority_queue’ data type, were defined. The ‘Node’ data type consisted of a struct containing numerous data items related to the objects in hand; being films. Similarly to the previously covered header file, a number of data items ranging from movie rating to film director, were included within this struct. Once again, this program boasts flexibility, as the same code could be adapted for a different scenario by simply altering the data items within the ‘Node’ struct.

One may note a slight difference between this header file and the previous. The ‘Node’ data type in the circular buffer header file does not include a ‘Node’ pointer called ‘next’. This is due to the fact that the implementation is dissimilar, and therefore the ‘next’ pointer would serve no use in this implementation. Another slight difference which is easily noted is that the ‘priority_queue’ data type has been defined to hold two extra pointers; ‘taken’ and ‘free’. Once again, due to the different technique of queueing, these pointers will be of use later on, in keeping track of the occupied, and void memory units in the queue. Other than this, the other data items within the ‘priority_queue’ structure have been left untouched.

Identical to the linked list header file, the function prototypes to be implemented in the client application have been listed. These, as was previously stated, will inform the compiler of the functions to be expected. All listed functions perform the exact same operations as was the case for the linked list; however the manner in which these operations will be performed will differ slightly.

```
#ifndef CIRCULAR_BUFF_H_INCLUDED
#define CIRCULAR_BUFF_H_INCLUDED

#include <stdbool.h>

#define MAXSIZE 40
#define SIZE 81

typedef struct node{    double priority;
                        char title[MAXSIZE];
                        char genre[MAXSIZE];
                        int length;
                        char release_date[MAXSIZE];
                        char director[MAXSIZE];
                        char actors[MAXSIZE];
                    } Node;

typedef struct Q{   char qname[MAXSIZE];
                    int size;
                    int max_size;
                    Node * head;
                    Node * tail;
                    Node * taken;
                    Node * free;
                }priority_queue;


/* operation: create a new empty priority queue with maximum number of
elements max_size*/
/* postcondition: new queue with a maximum size of ‘max_size’ created if
return is true */
/*                  else queue creation failed */
priority_queue * create_q(int max_size);

/* operation: enqueue ‘item’ in priority queue ‘queue’ with priority
‘priority’*/
/* precondition: Item ‘item’ and possibly empty or full queue ‘queue’ */
/* postcondition: placement of ‘item’ in ‘queue’ at point of priority if
return true*/
/*                  else failed to enqueue item*/
bool enqueue(priority_queue * queue, Node * node, double priority);

/* operation: dequeue element with highest priority from ‘queue’*/
/* precondition: non-empty priority_queue ‘queue’ */
/* postcondition: return of pointer to highest priority ‘item’ in ‘queue’
and removal of ‘item from ‘queue’*/
/*                  if ‘queue’ is emptied it is reset to ‘empty’ */
Node * dequeue(priority_queue * queue);

/* operation: return highest priority ‘item’ in ‘queue’ without removal*/
/* precondition: non empty priority queue ‘queue’*/
/* postcondition: return of pointer to highest priority ‘item’ in ‘queue’*/
/*                  without removal of ‘item’ from queue */
Node * peek(const priority_queue * queue);

/* operation: returns true if ‘queue’ is empty*/
/* precondition: possibly empty queue ‘queue’ */
/* postcondition: returns true if ‘queue’ contains 0 elements*/
bool is_empty(const priority_queue * queue);

/* operation: returns the number of elements in ‘queue’*/
/* precondition: possibly empty priority_queue ‘queue’ */
/* postcondition: returns the number of ‘items’ in ‘queue’*/
int size(const priority_queue * queue);

/* operation: creates new priority_queue from elements in q1 and q2*/
/*              without altering their contents*/
/* precondition: two queues ‘q1’and ‘q2’*/
/* postcondition: return of pointer to priority_queue containing elements
from ‘q1’ and ‘q2’*/
```

```
/*                without effecting ‘q1’ and ‘q2’*/
priority_queue * merge(const priority_queue * q1, const priority_queue * q2);

/* operation: remove all elements in priority_queue ‘queue’*/
/* precondition: non-empty priority_queue ‘queue’ */
/* postcondition: priority_queue ‘queue’ is emptied*/
bool clear(priority_queue * queue);

/* operation: store priority_queue ‘queue’ in file ‘f’*/
/* precondition: non-empty priority_queue ‘queue’*/
/* postcondition: priority_queue ‘queue’ is stored in file ‘f’ */
bool store(priority_queue * queue, char * f);

/* operation: load priority_queue ‘queue’ from file ‘f’*/
/* precondition: file ‘f’ containing priority_queue ‘queue’*/
/* postcondition: priority_queue ‘queue’ is loaded from file ‘f’ */
bool load(priority_queue * queue, char * f);

/*operation: Read string input*/
char * s_gets(char * st, int n);

#endif // CIRCULAR_BUFF_H_INCLUDED
```

2.2.2.1) Circular Buffer Function File

The following lines of code were implemented within the circular buffer function file. In this file, the functions to be used in the client application are defined. Implementing the circular buffer data structure, these functions will differ from the ones declared in the linked list function file, however their scope and purpose will remain unchanged.

As was the case with the linked list implementation, a handful of static global variables are required. Apart from the Node and priority_queue variables, an additional Node pointer titles 'buff' has been declared. This variable will be used to store and maintain the node data in the form of an array, and will be better understood in the coming section.

Both the 'dequeue_no_return' and the 'create_q_no_name' functions have once again been locally declared for use inside other functions.

```
#include<stdio.h>
#include<time.h>
#include<stdbool.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
#include "circular_buff.h"

//-------------------- DECLERATION OF STATIC VARIABLES ---------------------
extern Node *buff;
extern Node node;
extern priority_queue queue, q1, q2, q3;

//------------------------ LOCAL FUNCTION PROTOTYPES -----------------------
bool dequeue_no_return(priority_queue * queue);
bool create_q_no_name(int max_size, char * qname, priority_queue * queue);

//---------------------------- LOCAL FUNCTIONS ----------------------------
bool create_q_no_name(int max_size, char * qname, priority_queue * queue)
//create queue with out asking user to input name
{
    strcpy(queue->qname, qname);

    queue->size = 0;
    queue->max_size = max_size;
    queue->head = queue->tail = NULL;

    return true;
}
```

```
bool dequeue_no_return(priority_queue * queue)
//remove highest priority element from queue without returning the data
{
    if(is_empty(queue))
    {
        printf(“Queue is empty\n\n”);
        return false;
    }

    if(size(queue) == 1)
    {
        queue->taken = NULL;
//point taken pointer to NULL as no indexes are taken
    }
    else
    {
        if(queue->taken == queue->tail)
        {
            queue->taken = queue->head;
//point taken pointer to head as end of queue has been reached
        }
        else
        {
            queue->taken = queue->taken + 1;
//point taken pointer to next element
        }
    }

    queue->size--;          //decrement queue size

    return true;
}

//--------------------------- EXTERNAL FUNCTIONS --------------------------
priority_queue * create_q(int max_size)     //create new priority queue
{
    priority_queue * pq;
    pq = malloc(sizeof(priority_queue));
//memory allocation for queue to be created

    fflush(stdin);
    printf(“Please enter the name of the queue that you would like to
create:\n\n”);
    s_gets(pq->qname, MAXSIZE);

    if(pq->qname[0] == ‘\0’)
    {
        printf(“Invalid input\n\n”);
        s_gets(pq->qname, MAXSIZE);
        fflush(stdout);
    }

    pq->size = 0;
    pq->max_size = max_size;

    buff = malloc(sizeof(Node) * max_size);
//allocation of memory to global array variable
```

```
    pq->head = &buff[0];            //point head to first index in the array
    pq->tail = &buff[max_size - 1];  //point tail to last index in the array
    pq->taken = NULL;
//initialise first taken index pointer to 0 as no indexes are taken
    pq->free = &buff[0];    //point first empty index pointer to first index

    return pq;
}

int size(const priority_queue * queue)
//return number of elements in queue
{
    return queue->size;
}

bool is_empty(const priority_queue * queue) //return true if queue is empty
{
    return queue->size == 0;
}

bool clear(priority_queue * queue)          //clear queue contents
{
    while(!is_empty(queue))
    {
        dequeue_no_return(queue);
//call dequeue_no_return function to dequeue elements until empty
    }

    if(is_empty(queue))
    {
        return true;
    }
    else
    {
        return false;
    }
}

Node * dequeue(priority_queue * queue)
//remove element with highest priority from queue and return node data
{
    Node tnode;
    Node * rnode;

    if(is_empty(queue))
    {
        printf("Queue is empty\n\n");
        return NULL;
    }

    tnode = *(queue->taken);
//store contents of item at head of queue in tnode

    if(size(queue) == 1)
    {
        queue->taken = NULL;            //queue is now empty
    }
    else
    {
        if(queue->taken == queue->tail)
        {
            queue->taken = queue->head;
//if queue->taken points to the end of the array, direct it to start (circular
motion)
        }
        else
        {
            queue->taken = queue->taken + 1;
//make the first taken pointer point to the next index
        }
    }

    queue->size--;      //decrement size

    rnode = &tnode;
//required as return is of type pointer and tnode is a local variable

    return rnode;
}

priority_queue * merge(const priority_queue * q1, const priority_queue * q2)
//create new queue from contents of q1 and q2
{
    int newmax, i;
    newmax = i = 0;

    priority_queue newq;
    priority_queue * tqueue;            //create queue for new queue

    Node * tempn;

    newmax = q1->max_size + q2->max_size;
//compute the maximum for the new queue

    tqueue = create_q(newmax);
//create queue to hold maximum of newmax elements
    newq = *tqueue;

    tempn = q1->head; //point tempn to head of q1

    for(i = 0; i < q1->size; i++)
//N.B: If both queues are empty, the new queue will be empty but will have
a max_size of newmax
    {
        enqueue(&newq, tempn, tempn->priority);
//enqueue newq with elements of q1
        if(tempn == q1->tail)
        {
            tempn = q1->head;
        }
        else
        {
            tempn++;
        }
    }

    tempn = q2->head; //point tempn to head of q2

    for(i = 0; i < q2->size; i++)
    {
        enqueue(&newq, tempn, tempn->priority);
//enqueue newq with elements of q2
        if(tempn == q2->tail)
        {
            tempn = q2->head;
        }
        else
        {
            tempn++;
        }
    }

    if(size(&newq) == (size(q1) + size(q2)))
//test that all elements have been copied
    {
        tqueue = &newq;
        return tqueue;
    }

    printf("Merge operation failed\n\n");
    return NULL;
}

bool enqueue(priority_queue * queue, Node * node, double priority)
//enqueue node in queue
{
    int i = 0;
    bool change = false;
    Node cur, temp;
    Node * ptr1, * ptr2;

    if(size(queue) == queue->max_size)      //queue is full
    {
        printf("Queue is full\n\n");
        return false;
    }

    if(is_empty(queue)) //special case
    {
        queue->taken = queue->free;

        *(queue->free) = *node;
//take copy of node data and store in location pointed to by free

        if(queue->free == queue->tail)
        {
            queue->free = queue->head;
        }
        else
        {
            queue->free = queue->free + 1;
//point free pointer to next index;
        }

        queue->size++;

        return true;
    }
    else
    {
        *(queue->free) = *node;
//take copy of node data and store in location pointed to by free

        if(queue->free == queue->tail)
        {
            queue->free = queue->head;
        }
        else
        {
            queue->free = queue->free + 1;
// point queue->free to next free element
        }

        queue->size++;
    }

    //sort array priorities

    ptr1 = malloc(sizeof(Node));
//memory allocation for Node pointers
    ptr2 = malloc(sizeof(Node));           //to be used in sorting operation

    change  = false;

    ptr1 = queue->taken;
//point ptr1 to element at front of queue

    if(queue->free == queue->head)
    {
        ptr2 = queue->tail;
    }
    else
    {
        ptr2 = queue->free - 1;
    }

/* every time a node is enqueued in the queue it will be placed at the
back of the queue therefore we shall sort the queue out each time a new node
is added, therefore everytime a new node is added, any other nodes in the
queue are already sorted, therefore once the new node is in place the queue
is sorted and we may return*/

    for(i = 0; i < ((queue->size) - 1); i++)        //sorting algorithm
    {
        cur = *ptr1;
//take copy of data pointed to by ptr1 and store in cur
        temp = *ptr2;   //same with ptr2 and temp
        if((cur.priority    <    temp.priority)    ||    ((cur.priority    ==
temp.priority) && (temp.priority != node->priority)))
//test priority of new node against priority of other nodes before it
        {
            *ptr2 = cur;
            *ptr1 = temp;
            change = true;
        }
        else
        {
            if(change)
            {
                break;
//if there was a change made, and now there isn't, the queue must be sorted
therefore break
            }
        }

        if(ptr1 == queue->tail)
        {
            ptr1 = queue->head;
        }
        else
        {
            ptr1++
        }//traverse list - with each cycle the highest priority element is
in place
    }

    free(ptr1);         //free previously allocated memory

    ptr2 = queue->taken;    //for testing purposes

    for(i = 0; i < ((queue->size) - 1); i++)       //test for sorting success
    {
        if(ptr2->priority < (ptr2+1)->priority)
        {
            printf("Something went wrong - element not queued\n\n");
            return false;
        }
        ptr2++; //traverse list
    }

    free(ptr2); //free previously allocated memory

    return true;
}

Node * peek(const priority_queue * queue)
//return data of highest priority element without removing it from queue
{
    if(is_empty(queue))
    {
        printf("Queue is empty\n\n");
        return NULL;
    }
    return queue->taken;
}

bool store(priority_queue * queue, char * fname)
//write queue data to file
{
    FILE * fp;
    Node tempn;
    Node * temp;
    int i, tsize;
    i = tsize = 0;

    if((fp = fopen(fname, “w+b”)) == NULL)
//attempt to open ‘fname’ in write plus binary mode
    {
        printf(“Unable to open file ‘%s’\n\n”, fname);
        return false;
    }

    rewind(fp); //rewind file to the beginning

    fwrite(queue, sizeof(priority_queue), 1, fp);
//write queue data such as name and size to file

    tsize = queue->size;            //temporarily store the queue size

    for(i = 0; i <tsize; i++)       //sequentially read in node data
    {
        temp = dequeue(queue);
//dequeue highest priority element from list
        tempn = *temp;
//required as dequeue returns a node pointer
        fwrite(&tempn, sizeof(Node), 1, fp);
//write highest priority element to file
    }

    if(fclose(fp) != 0)
    {
        printf(“Unable to close file ‘%s’\n\n”, fname);
        return false;
    }

    return true;
}

bool load(priority_queue * queue, char * fname)
//read queue data from file
{
    FILE * fp;
    Node * ptr;
    int i, temps, highest, num;
    i = temps = highest = num = 0;

    if((fp = fopen(fname, “rb”)) == NULL)
//attempt to open file in read binary mode
    {
        printf("Unable to open file '%s'\n\n", fname);
        return NULL;
    }

    rewind(fp); //rewind file to the beginning

    fread(queue, sizeof(priority_queue), 1, fp);
//read queue data from file

    temps = queue->size;        //temporarily store queue size

    fread(buff, sizeof(Node), queue->max_size, fp);
//read max_size nodes from memory in node sized chunks and write to buff
array

    queue->head = &buff[0];
//point head of queue to first index in array
    queue->tail = &buff[queue->max_size - 1];
//point tail to last index in array

    ptr = queue->head;

    highest = 0;

    for(i = 0; i < queue->max_size; i++)
//algorithm determines which node the taken pointer must point too
    {
        if(ptr != NULL)
        {
            if(ptr->priority > highest)
            {
                highest = ptr->priority;
//node with highest priority is found
                num = i;
            }
        }
      ptr = ptr + 1;
    }

    queue->taken = &buff[num];
//node with highest priority is in the front of the queue and is therefore
pointed to by taken

   queue->free = queue->taken;
    for(i = 0; i < temps; i++)
    {
            if(queue->free == queue->tail)
            {
                queue->tail = queue->head;
            }
            else
            {
                queue->free = queue->free + 1;
            }
        }
        queue->size = temps;        //queue size is restored

        if(fclose(fp) != 0)
        {
            printf("Unable to close file '%s'\n\n", fname);
            return false;;
        }
        return true;
}

char * s_gets(char * st, int n)     //implemented for string input
{
      char * ret_val;
      int i = 0;

      ret_val = fgets(st, n, stdin);

      if (ret_val)
      {
            while (st[i] != '\n' && st[i] != '\0')
            {
                  i++;
            }

            if (st[i] == '\n')
            {
                  st[i] = '\0';
            }
            else // must have words[i] == '\0'
            {
                  while (getchar() != '\n')
                  {
                        continue;
                  }
            }
      }
      return ret_val;
}
```

2.2.2.2) Source Code Explained

- bool create_q_no_name(int max_size, char * qname, priority_queue * queue)

This function serves an internal purpose, and is defined to create a new queue without asking the user to input data, as is the case with the other function 'create_q'. 'queue', the priority_queue pointer passed as a parameter is made to point to a new queue data item with maximum size set to 'max_size', current size set to 0, name set to 'qname', and with the head and tail pointers point to NULL. This signifies that the queue currently has no memory to hold nodes assigned to it. The function will return true if queue creation succeeds.

- bool dequeue_no_return(priority_queue * queue)

This function accepts a pointer to a queue as a parameter. The queue is first tested to be empty. If this test comes back positive, then the operation is halted and the function returns false, showing that a node has not been dequeued. Otherwise the function continues on.

```
if(size(queue) == 1)
    {
        queue->taken = NULL;
    }
    else
    {
        if(queue->taken == queue->tail)
        {
            queue->taken = queue->head;
        }
        else
        {
            queue->taken = queue->taken + 1;
        }
    }
```

These lines of code perform a test on the queue passed to the function. If the queue currently contains one node, then the taken pointer, which points to the first data element in the queue, is set to point to NULL, as, once the node is removed from the queue, there would be no more data elements in the queue.

If more than one data element is currently queued, then a second test is made. If the taken pointer is currently pointing to the same place as the tail pointer, which points to the last memory location in the queue (array), then the taken pointer is set to point to the head, being the first memory location in the queue (array). This performs a circular motion. As was explained in section 1.2, a circular buffer, in fact is not physically circular, therefore once the end of the array is reached, the pointer must be diverted to the front of the pointer. This operation is handled at this point.

If the taken pointer is currently not pointing to the last index in the array, then it is set to point to the next element in line. This is done by incrementing its memory location. By setting it to point to itself 'plus one index', the compiler sets it to point to the next index in the array, as was explained in section 1.1.

Finally the queue size is decremented, as the highest priority element has been dequeued, and the function returns true to signify that the dequeue operation has been successfully executed. The only difference between this function and the 'dequeue' function is that, unline the 'dequeue' function, this function does not return any node data.

- priority_queue * create_q(int max_size)

This function, declared to be used within the client application, is designed to create a new circular buffer priority queue. The return type of this function is a pointer to a priority_queue, which will be the newly created queue, and the parameter is an integer called 'max_size' which will be the maximum number of nodes to permit the created queue to enqueue.

```
priority_queue * pq;
pq = malloc(sizeof(priority_queue));
```

The first step in creating a new queue is to create a priority_queue pointer, in this case 'pq' and then allocate memory to it. The amount of memory allocated is that equal to the size of a priority_queue, and 'ps' is then set to point to the start of this memory location.

The user is then asked to input the name that they would like to call the queue. The size of the new queue ‘pq’ was then set to 0 and the maximum number of nodes to be enqueued was set to the amount specified by the parameter ‘max_size’. The next step is to allocate memory to the global variable ‘buff’, which will act as an array of nodes which will house the actual enqueuement of the individual nodes:

```
buff = malloc(sizeof(Node) * max_size);

pq->head = &buff[0];
pq->tail = &buff[max_size - 1];
pq->taken = NULL;
pq->free = &buff[0];
```

Here, the ‘buff’ pointer previously mentioned is allocated a memory space equal to size of ‘max_size’ number of nodes. Therefore, when compared to the linked list implementation, it is already apparent that this implementation is inefficient, as even when the queue is empty, the memory for the maximum number of nodes must be made available.

The queue->head pointer is then made to point to the first index of the array ‘buff’, being index 0, while the tail pointer is made to point to the last index, being ‘max_size’ – 1 (as array indexing starts at 0). The taken pointer is then set to point to NULL, to signify that there are currently no elements in the queue, while the free pointer is set to point to the first index in the array. This pointer can be set anywhere in the queue, but for purposes of visualisation, it has been set to the first index. This pointer will point to the next location for a node to be queued.

The priority queue has now successfully been allocated memory and initialised, therefore the function returns the pointer ‘pq’ which is now pointing to the new queue.

- int size(const priority_queue * queue)

This function serves the purpose of returning the number of nodes currently queued in parameter ‘queue’. The parameter has been declared as being a constant, which ensures that the data from the priority_queue pointer will simply be read, and will not be altered. This function accesses the queue size and returns this value.

- bool is_empty(const priority_queue * queue)

As was seen in the previous function, this function accepts a constant priority_queue pointer as a parameter. The purpose of this function is to test whether the queue passed as a parameter is empty or not. The size of the queue is compared to zero, and the return is true if it is equal, or false if it is not.

- bool clear(priority_queue * queue)

The purpose of the function is to truncate the contents of the queue pointed to by the parameter.

```
while(!is_empty(queue))
{
     dequeue_no_return(queue);
}
```

This loop will run until the queue is no longer empty. While the queue is not empty, the 'dequeue_no_return' function, which was explained above, is called to remove the node at the front of the list. Therefore, the loop will run for *n* times if there are *n* nodes queued.

Once the loop is exited, the queue is tested to be empty. Is this is found to be true then the return of the function will be true, else the return will be false, showing that the operation has failed. Noteworthy is the fact that the queue name and queue maximum size have not been altered, simply the contents of the queue, and obviously the size, taken and free pointers, have been altered.

- Node * dequeue(priority_queue * queue)

The dequeue function is to serve the same purpose as the 'dequeue_no_return' function, but is to return the data of the node which was dequeued. Therefore the only added steps are to first store a copy of the node data at the front of the queue and then return it.

```
tnode = *(queue->taken);
```

'tnode', a Node variable previously declared, stores a copy of the data of the node which is about to be dequeued. As was previously implemented, the 'taken' pointer is then set to point to NULL, if the queue is now empty, or to the next element if it is not. The size of the queue is then decremented. The final step is to copy the data of the node to a new Node pointer which will be returned by the function. This step is required because 'tnode' is a variable with function scope and therefore returning its address runs the risk of returning corrupted data. Therefore the following step is required:

```
rnode = &tnode;
return rnode;
```

- bool enqueue(priority_queue * queue, Node * node, double priority)

The purpose of this function is to add nodes to the priority queue provided. The three parameters include a pointer to the queue to be enqueued, a pointer to the node to be queued and the priority of that node. The function then returns true or false, based on the final success of the operation.

The queue is first tested to be full. If this results as true, then the operation is halted and the function returns false, as the queue has no more available space to queue nodes. Otherwise the operation proceeds. We must now enqueue the node. First, a test is conducted to see whether the queue is empty, and therefore the new element will be the only element in the queue. If this is the case, the following operation occurs:

```
queue->taken = queue->free;
*(queue->free) = *node;
if(queue->free == queue->tail)
{
        queue->free = queue->head;
}
else
{
        queue->free = queue->free + 1;
}
queue->size++;
return true;
```

First, the taken pointer is set to point to the location where the free pointer is currently pointing. This is going to be the location where the new node is going to be added. The data from the node is then copied to the location pointed to be free (which is alos pointed to by taken). The node has now been added to the queue, however the free pointer must still be shifted.

The same test which was previously conducted on the taken pointer is now conducted on the free pointer. If the free pointer is currently at the tail of the queue, it is moved to the head, otherwise it is simply shifted one space. This ensures that if the end of the array has been reached, the circular motion is maintained and the pointer is redirected. Once this has been accomplished, the queue size is incremented and the function returns true, signifying that the operation was successful.

The second possible case, is that the queue is not empty prior to the addition of the new node. If this scenario is present, the following operation is undertaken:

```
*(queue->free) = *node;
if(queue->free == queue->tail)
{
        queue->free = queue->head;
}
else
{
        queue->free = queue->free + 1;
}
queue->size++;
```

Here, the node data is copied to the next free location at the back of the queue (pointed to by ‘free’). The same test on free is conducted, ensuring that the circular path is maintained if the pointer has reached the end of the array. The queue size is then incremented. However, the node has now simple been placed at the back of the queue, regardless of its priority. Therefore, this process is not over, as the queue now has to be sorted.

In order to sort the queue, two Node pointers must be declared and allocated memory. These will be used to traverse the queue and reshuffle the nodes. They have been named ‘ptr1’ and ‘ptr2’. A Boolean variable called ‘change’ is also declared and initialised to false, while ‘ptr1’ is to point to the first node in the queue:

```
ptr1 = malloc(sizeof(Node));
ptr2 = malloc(sizeof(Node));
change  = false;

ptr1 = queue->taken;
if(queue->free == queue->head)
{
    ptr2 = queue->tail;
}
else
{
    ptr2 = queue->free - 1;
}
```

Here, ‘ptr2’ is set to point to the last node in the queue. This means that it must point to the index prior to the free pointer. Therefore, due to the circular properties of the queue, the location of the free pointer must be tested, as if the free pointer is currently pointing to the head of the queue, ‘ptr2’ must point to the tail, as it is the final occupied index. The sorting of the queue may now commence:

```
for(i = 0; i < ((queue->size) - 1); i++)
    {
        cur = *ptr1;
        temp = *ptr2;
if((cur.priority < temp.priority) || ((cur.priority == temp.priority) && (temp.priority != node->priority)))
        {
            *ptr2 = cur;
            *ptr1 = temp;
            change = true;
        }
        else
        {
            if(change)
            {
                break;
            }
        }
        if(ptr1 == queue->tail)
        {
            ptr1 = queue->head;
        }
        else
        {
            ptr1++;
        }
    }
```

Firstly, the for-loop is set to run the number of times equal to the queue size, or until the ‘break’ command is called. The values of the node pointed to by ‘ptr1’ are copied to a Node variable named ‘cur’, while the data of the node pointed to by ‘ptr2’ is copied to a Node variable named ‘temp’.

Next, the priority of the ‘cur’ variable to compared to that of the ‘temp’ variable. If the ‘cur’ variable, pointed to by ‘ptr1’ is less than that of the temp, *OR* their priorities are the same *AND* the variable pointed to by ‘ptr2’ does not have the same priority as that of the node, the if statement is entered. This test may seem confusing at first, but when it is broken down, it isn’t too hard to understand.

The first case is if the ‘cur’ priority is less than the ‘temp’ priority, the statement is entered. However, if their priorities are the same, the statement is only entered if ‘temp’ does not hold the same priority as the node being added, and, hence, is not the node currently being added. This whole test is required, as it ensures that if the priorities of the nodes being compared are the same, they are treated on a first-come-first-served basis.

If the if-statement is entered, the two nodea pointed to by ‘ptr2’ and ‘ptr1’ have their values swapped and the changed Boolean is true. The location pointed to by ‘ptr1’ is then set to point to the next taken element, with the usual circular motion being maintained with the if-statement test.

This cycle is repeated until the first if-statement is not entered. This will occur when the node at the end of the queue is in its correct place and thus the queue is sorted. Therefore, when this occurs, following the shifting around of nodes, the loop is exited by means of the ‘break’ reserved word. However, this statement is only called if a change has previously been made, ensuring that the nodes are now in place.

The fact that each element is correctly positioned in the queue when it is enqueued means that every time the enqueue function is called, only one node must be correctly position, as the rest of the nodes are already in place.

Once this operation has completed, the Node pointer ‘ptr1’ is freed while ‘ptr2’ is kept to perform a final check. Starting from the first node in the queue, each node is compared to the node following it, and if the order of the priorities is correct, then the function has succeeded in enqueueing the node and it returns true. If one node is out of position, then the operation has failed and the return is false.

- priority_queue * merge(const priority_queue * q1, const priority_queue * q2)

This function takes two priority_queue pointers as its parameters and is designed to merge the two queues and form a new queue containing the nodes from each one. The first task to be performed is to declare a priority_queue pointer to point to the new queue, and to declare a Node pointer to transfer the nodes from the original queues to the new queues.

The maximum of the new queue to be created is computed by summing the maximum sizes of the queues provided. This value, known as ‘newmax’ is then passed to the create_q function, previously defined, so as to create a new queue. This new queue is then pointed to by the priority_queue pointer ‘tqueue’:

```
newmax = q1->max_size + q2->max_size;
tqueue = create_q(newmax);
```

The next step is to traverse the first queue ‘q1’ and enqueue each of the nodes in ‘tqueue’:

```
newq = *tqueue;
tempn = q1->taken;
for(i = 0; i < q1->size; i++)
{
    enqueue(&newq, tempn, tempn->priority);
    if(tempn == q1->tail)
    {
        tempn = q1->head;
    }
    else
    {
        tempn++;
    }
}
```

The values of the queue pointed to by ‘tqueue’ are copied to a priority_queue variable newq, which is then used as an argument in the enqueue function. ‘tempn’ is first set to point to first node in ‘q1’ and is then enqueued within ‘newq’ through the enqueue function. ‘tempn’ is then set to point to the next node in the list, by implementing the usual if-statement to ensure a circular motion. This process repeats until all the nodes in ‘q1’ have been enqueued in ‘newq’.

The exact same process is repeated with the elements from ‘q2’, until ‘newq’ contains the complete set of nodes from both queues. A final test is then conducted to ensure that all the values have been enqueued. This is done by comparing the size of the merged queue ‘newq’ to the size of the parameter queues ‘q1’ and ‘q2’. If ‘newq’ contains a number of elements equal to the sum of those in ‘q1’ and ‘q2’, then the merge operation has succeeded. If this is so, ‘tqueue’ is set to point to the new queue ‘newq’ and is returned. Otherwise the NULL pointer is returned, signifying a failed attempt to merge the new queues.

Noteworthy is the fact that the two queues ‘q1’ and ‘q2’ have not been altered in anyway, and stay true to their definition of being constant within the merge function.

- Node * peek(const priority_queue * queue)

The purpose of this function is to return the data of the highest priority node in the queue without dequeueing it. Since the data is not to be altered, the queue pointer passed as a parameter is listed as being constant. The peek function first checks if the queue is empty by means of the 'is_empty' function. If this is the case then the function is terminated by returning a NULL pointer, otherwise the taken pointer, pointing to the node at the front of the queue, is returned.

- bool store(priority_queue * queue, char * fname)

This function is designed to write the queue data to file to be stored until further use. The queue to be written to file is pointed to by the parameter 'queue', while the file name to write 'queue' to is listed in the character pointer 'fname'.

Firstly a FILE pointer is declared as 'fp' and is used to point to the start of the file.

```
if((fp = fopen(fname, "w+b")) == NULL)
{
        printf("Unable to open file '%s'\n\n", fname);
        return false;
}
rewind(fp);
```

Here an attempt to open the file in 'w+b' mode (write plus in binary mode), will return NULL if unsuccessful. 'w+' mode allows data to be read from the file as well as written, while the file is opened in binary mode for purposes of portability. If 'fp' is pointing to NULL, the attempt to open has failed and the operation halts, returning false. Otherwise the file pointer is set to point to the start of the file, in preparation to write the queue data (Note: any previously written data will be over written by the new data. This is possible as any data which was previously present has already been read and therefore is no longer needed).

The next step is to begin writting the data.

```
fwrite(queue, sizeof(priority_queue), 1, fp);
tsize = queue->size;
for(i = 0; i <tsize; i++)
{
     temp = dequeue(queue);
     tempn = *temp;
     fwrite(&tempn, sizeof(Node), 1, fp);
```

```
			}
			free(buff);
```

Here, the queue data is first written to file, this includes the queue size, maximum size and name. The head, tail, taken and free pointers are also written, however these may be disregarded as the addresses they point to will possible point to garbage when they are reloaded, and therefore must be updated at load time anyway.

Next, a variable 'tsize' temporarily stores the size of the queue, and is used to loop through the contents of the queue. Within the loop, the highest priority element is dequeued by means of the dequeue function. This function returns a pointer to the highest priority node, which is then copied to a temporary node variable and written to file by means of the 'fwrite' function. This is repeated until all nodes have been dequeued and written to file. Once all data has been written, the memory which had been allocated to the global variable 'buff', when the queue was created, is now freed, so as to prevent any memory leaks.

An attempt is then made to close the file, as all data has been read. If this is successful, the operation is over and the function returns true, otherwise the operation has failed and the function returns false.

- bool load(priority_queue * queue, char * fname)

The load function is required as it reads the data which has previously been written to file, allowing for previosuly created and populated queues to be operated upon. Similarly to the store function, the first parameter is a priority_queue pointer which will point to the queue which shall be loaded, while the second parameter is a character pointer which holds the name of the file to be opened.

The first step is to attempt to open the file, and store the location of the file in a file pointer 'fp'. As was the case with the store function, the 'fopen' function is used. This function attempts to open the file in 'rb' mode, meaning 'read in binary'. This time round, this mode has been selected, as since data is being loaded, the contents of the file should not be altered. Therefore, in this way, any attempt to alter or corrupt the data in the file will be blocked by the compiler. If the attempt to open the said file is successful, the file pointer is set to point to the start of the file, so that reading may proceed.

```
fread(queue, sizeof(priority_queue), 1, fp);
temps = queue->size;
buff = malloc(sizeof(Node) * queue->max_size);
fread(buff, sizeof(Node), queue->max_size, fp);
queue->head = &buff[0];
queue->tail = &buff[queue->max_size - 1];
```

Here, the queue data, including the size, max_size, name and pointers, is read. As was previously mentioned, the pointers, including head, tail, free and taken, must be updated, as the newly allocated buffer memory is most likely in a different location. Therefore, the global variable pointer, 'buff' is allocated enough memory to hold the maximum number of elements and is then populated with the node data which was previously written to file. The fread function is reading a number of chunks of data, equal to a Node in size, from the file to the 'buff' variable. The number of chunks read is equal to the number of nodes present. Therefore all the nodes which were previously written to file (in order of priority), are now read into buff. The head is then set to point to the first index in the 'buff' array, while the tail is set to point to the final index, as was implemented in the create_q function.

The final step is to assign the taken and free pointers a location to point too, as their previous values are no longer valid.

```
ptr = queue->head;
highest = 0;
for(i = 0; i < queue->max_size; i++)
{
    if(ptr != NULL)
    {
        if(ptr->priority > highest)
```

```
                {
                    highest = ptr->priority
                    num = i;
                }
            }
            ptr = ptr + 1;
        }
        queue->taken = &buff[num];
```

Node pointer ‘ptr’ is first set to point to the head of the queue, be it a void or occupied element. The highest int variable is then set to 0. The array is then traversed, from the head to the tail, with the operation keeping track of the node encountered with the highest priority. Once the loop has finished, ‘num’ will store the index number of the node with the highest priority. This node must be at the front of the queue and should therefore be pointed too by the taken pointer. Therefore the taken pointer is finally set to point to that location.

Assigning the free pointer requires a different approach:

```
        queue->free = queue->taken;
        for(i = 0; i < queue->size; i++)
        {
                if(queue->free == queue->tail)
                {
                    queue->tail = queue->head;
                }
                else
                {
                    queue->free = queue->free + 1;
                }
        }
    queue->size = temps;
```

Here, the free pointer is first set to point to the same location as the head. A loop is then set to run the same amount of times as the size of the queue. Each cycle of the loop will result in the free pointer moving one node down. The usual operation is executed, where if the free is at the end of the array, it is redirected to the start. Once all the cycles of the loop have finished, the free pointer will be one index behind the node with the lowest priority, and will therefore be in its place.

The correct queue size is then restored to the queue pointer and an attempt to close the file is made. If the file is successfully closed, then the function returns true, otherwise the function returns false, signifying a failed attempt to load from file.

- char * s_gets(char * st, int n)

This function is implemented when string input is requested from the user. The character pointer ‘st’ and the integer ‘n’ are passed as arguments. Data input by the user is then stored in pointer ‘st’ up to the ‘n’th character. This is performed through the ‘fgets’ function which reads data from the keyboard buffer. The downside of the ‘fgets’ function is that the newline character pressed by the user, which is needed to flush the buffer, is also stored in ‘st’. Therefore, the ‘s_gets’ function traverses the character input until it finds the newline character, this newline character is then replaced by the end of line character ‘\0’, resulting in a string without the newline character at the end. This is often convenient, especially when storing data within databases or other storage systems such as this.

2.3.1) Client Application Source Code

A client application capable of handling both data structures was created to test out the capabilities and the efficiency of the two priority queue approaches. A simple menu was designed to provide full functionality to users while testing the whole domain of functions and operations. As was stated with regards to the priority queue approaches, the client application could be adapted in a short amount of time to serve a different priority queue purpose. The implementation of modular structured and robust code allows for this program to run on most systems without any alterations necessary. This is a key step towards code portability and is a must when designing generalised software, with no specific end user.

The following lines of code make up the client application:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<stdbool.h>
#include "circular_buff.h" (alternatively: #include "linked_list.h")

//----------------------- LOCAL FUNCTION PROTOTYPES -----------------------
int menu_fun(void);
void create_new_queue_fun(void);
void enqueue_queue_fun(void);
void dequeue_queue_fun(void);
void peek_fun(void);
void size_fun(void);
void merge_fun(void);
void truncate_fun(void);

//------------------- DECLERATION OF STATIC VARIABLES ---------------------
priority_queue queue, q1, q2, q3;

Node *buff;

Node node;

//---------------------------- MAIN FUNCTION ------------------------------
```

```
int main(void)
{
    int choice = 0;
    bool quit = false;

    do
    {
        choice = menu_fun();                  //menu call

        switch(choice)
        {
        case 1:

           create_new_queue_fun();           //queue creation function call

            break;
        case 2:

            enqueue_queue_fun();          //queue enqueuement function call

            break;
        case 3:

           dequeue_queue_fun();           //queue dequeuement function call

            break;
        case 4:

            peek_fun();                   //peek element function call

            break;
        case 5:

            size_fun();                  //size function call

            break;
        case 6:

            merge_fun();                 //merge function call

            break;
        case 7:

            truncate_fun();                 //truncate function call

            break;
        case 8:

            quit  = true;                   //quit program

            break;
        default:

                printf("Invalid input\n");

            break;
        }

    }while(!quit);

    return 0;
}

//--------------------------- LOCAL FUNCTIONS ---------------------------
int menu_fun(void)                         //menu function
{
    int choice = 0;

  printf("                       ***** MENU *****\n\n");
  printf("1) Create new queue                  2) Enqueue element\n");
  printf("3) Dequeue highest rated element      4) View highest rated
                                                 element\n");
  printf("5) Queue size                        6) Merge two queues\n");
  printf("7) Truncate queue                    8) Quit\n\n");

        while((scanf("%d", &choice) != 1) || (choice < 1) || (choice > 8))
        {
            fflush(stdin);
            printf("Invalid entry\n\n");
        }

        return choice;
```

```
}

void create_new_queue_fun(void)           //function to create new queue
{
    int tempcheck, size2;
    tempcheck = size2 = 0;

     printf("                   ***** CREATE NEW QUEUE *****\n\n");

printf("Please enter the maximum number of elements\nto be stored in the
queue to be created: \n\n");

            tempcheck = scanf("%d", &size2);

            while((tempcheck != 1) || (size2 < 1) || (size2 > 100))
            {
                fflush(stdin);
                printf("\nInvalid Input\n\n");
                tempcheck = scanf("%d", &size2);
            }

                priority_queue * qu;
//required to store priority_queue pointer returned by create_q
                qu = create_q(size2);
                queue = *qu;

            printf("\nQueue successfully generated\n\n");

            if(!store(&queue, queue.qname))
            {
                printf("\nUnable to save '%s' to file\n\n",queue.qname);
            }
}

void enqueue_queue_fun(void)               //function to enqueue elements
{
    int test = 0;

    printf("                    ***** ENQUEUE ELEMENT *****\n\n");

            do
            {
                    fflush(stdin);
printf("Please enter the name of the queue you would like to enqueue:\n\n");
                    s_gets(queue.qname, SIZE);

            }while(queue.qname[0] == '\0');

           if(!load(&queue, queue.qname))
            {
                  printf("Unable to load queue\n\n");
                  return;
            }

            if(size(&queue) == queue.max_size)
            {
                printf("Queue is full\n\n");
                return;
            }

           printf("We will now enqueue '%s' with elements\n\n",queue.qname);
            printf("Please enter the Movie Title: ");
            s_gets(node.title, SIZE);

            if(node.title[0] == '\0')
            {
                strcpy(node.title, "N/A");
            }

            printf("\nPlease enter the Movie Genre: ");
            s_gets(node.genre, SIZE);

            if(node.genre[0] == '\0')
            {
                strcpy(node.genre, "N/A");
            }

            printf("\nPlease enter the Movie Length: ");
            test = scanf("%d",&node.length);
            fflush(stdin);

            if(test == 0)
            {
                node.length = '~';
            }

            printf("\nPlease enter the Movie Release Date: ");
            s_gets(node.release_date, SIZE);

            if(node.release_date[0] == '\0')
            {
                strcpy(node.release_date, "N/A");
            }

            printf("\nPlease enter the Movie's director: ");
            s_gets(node.director, SIZE);

            if(node.director[0] == '\0')
            {
                strcpy(node.director, "N/A");
            }

            printf("\nPlease enter the Movie's Main Actors: ");
            s_gets(node.actors, SIZE);

            if(node.actors[0] == '\0')
            {
                strcpy(node.actors, "N/A");
            }

            printf("\nPlease enter the Movie Rating (0 - 10): ");

while(((scanf("%lf",&node.priority)) != 1) || (node.priority < 0.0) ||
(node.priority > 10.0))
            {
                printf("\nInvalid input!\n\n");
                fflush(stdout);
            }

            if(enqueue(&queue, &node, node.priority))
            {
                printf("Element successfully queued!\n\n");
                if(!store(&queue, queue.qname))
                {
              printf("Unable to store '%s' at the moment\n\n", queue.qname);
                }

                return;
            }

            printf("Element was not successfully enqueued\n\n");
}

void dequeue_queue_fun(void)                //function to dequeue elements
{
    Node * tempn;

     printf("    ***** DEQUEUE ELEMENT *****\n\n");

            do
            {
                fflush(stdin);
 printf("Please enter the name of the queue you would like to dequeue:\n\n");
                s_gets(queue.qname, SIZE);

            }while(queue.qname[0] == '\0');

            if(!load(&queue, queue.qname))
            {
                return;
            }

            if(size(&queue) == 0)
            {
                printf("Queue is empty\n\n");
                return;
            }

            printf("We will now dequeue '%s'\n\n",queue.qname);

            tempn = dequeue(&queue);

            if(tempn == NULL)
            {
                return;
            }
            node = *tempn;

 printf("The following Movie has been removed from '%s': \n\n", queue.qname);
            printf("Title: %s\n", node.title);
            printf("Genre: %s\n", node.genre);
            printf("Director: %s\n", node.director);
            printf("Actors: %s\n", node.actors);
            printf("Length: %d\n", node.length);
            printf("Release date: %s\n", node.release_date);
            printf("Rating: %0.1lf\n\n", node.priority);

            if(!store(&queue, queue.qname))
            {
                printf("Unable to store '%s' to file\n\n", queue.qname);
            }
}

void peek_fun(void)            //function returns highest priority element
{
    printf("    ***** HIGHEST RATED ELEMENT *****\n\n");

           do
            {
                fflush(stdin);
printf("Please enter the name of the queue you would like to operate upon:\n\n");
                s_gets(queue.qname, SIZE);

            }while(queue.qname[0] == '\0');

            if(!load(&queue, queue.qname))
            {
               printf("Unable to load queue\n\n");
               return;
            }

            Node * no;
            no = peek(&queue);

            if(no == NULL)
            {
                return;
            }
            node = *no;

            printf("The highest rated Movie in '%s' is:\n\n", queue.qname);
            printf("Title: %s\n", node.title);
            printf("Genre: %s\n", node.genre);
            printf("Director: %s\n", node.director);
            printf("Actors: %s\n", node.actors);
            printf("Length: %d\n", node.length);
            printf("Release date: %s\n", node.release_date);
            printf("Rating: %0.1lf\n\n", node.priority);
}

void size_fun(void)                   //function returns size of queue
{
    printf("    ***** QUEUE SIZE *****\n\n");

            do
            {
                fflush(stdin);
                printf("Please enter the name of the queue you would like to
operate upon:\n\n");
                s_gets(queue.qname, SIZE);

            }while(queue.qname[0] == '\0');

            if(!load(&queue, queue.qname))
            {
               printf("Unable to load queue\n\n");
               return;
            }

printf("'%s' currently contains %d elements out of a maximum of %d\n\n",
queue.qname, size(&queue), queue.max_size);
}
```

```
void merge_fun(void)             //merge function
{
    printf("    ***** QUEUE MERGE *****\n\n");

            do
            {
                fflush(stdin);
printf("Please enter the name of the first queue that you would like to merge:\n\n");
                s_gets(q1.qname, SIZE);

            }while(q1.qname[0] == '\0');

            if(!load(&q1, q1.qname))
            {
                printf("Unable to load queue\n\n");
                return;
            }

            do
            {
printf("Please enter the name of the queue you would like to operate upon:\n\n");
                s_gets(q2.qname, SIZE);

            }while(q2.qname[0] == '\0');

            if(!load(&q2, q2.qname))
           {
               printf("Unable to load queue\n\n");
               return;
           }

            priority_queue * qu2;
                  //required as merge return priority_queue pointer
            qu2 = merge(&q1, &q2);

            if(qu2 == NULL)
            {
                return;
            }
            q3 = *qu2;

            if(store(&q3, q3.qname))
            {
                printf("Merge successful\n\n");
                return;
            }

                printf("Unable to store '%s' to file\n\n", q3.qname);
}

void truncate_fun(void)                                //truncate function
{
    printf("    ***** TRUNCATE QUEUE *****\n\n");

            do
            {
                fflush(stdin);
printf("Please enter the name of the queue you would like to truncate:\n\n");
                s_gets(queue.qname, SIZE);

            }while(queue.qname[0] == '\0');

            if(!load(&queue, queue.qname))
            {
               printf("Unable to load queue\n\n");
               return;
            }

            if(clear(&queue))
            {
printf("'%s' has successfully been truncated\n\n", queue.qname);
                if(!store(&queue, queue.qname))
                {
              printf("Unable to store queue '%s' to file \n\n", queue.qname);
                }
                return;
            }

            printf("Clear unsuccessful\n");
```

```
}
```

2.3.1) Source Code Explained

When designing the client application, one main objective was to implement modularity in defining multiple functions and calling the functions when needed. This not only made the code neater, but it was of great help when testing and debugging the program.

The menu, together with the choices were defined as seperate functions and were then called in the main function. Therefore, the first few lines of code included defining the function prototypes. Next, the external variables were defined, including the Node pointer 'buff', the Node variable 'node' and the four priority_queue variables, 'queue', 'q1, 'q2', and 'q3'. These variable will be used time and time again throughout the client application to store return values as well as pass user data to various functions.

The main function included a do-while loop which ran the program until the user decided to quit. The menu function was initially called, and based on the users selection, a switch statement was used to select the valid choice function.

The 'menu_fun' function displayed a list of possible menu choices and took user input. The input was tested to be valid and within range, and once a valid choice was input, the function returned this value. Based on this value, one of the multiple paths was chosen through the switch statement.

The 'create_new_queue_fun' is called if option choice 1 is selected. The user is immediately asked to input the maximum number of elements to be queued in their desired queue. Once validated, this value was passed as an argument to the 'create_q' function, which generated a new queue. The return of this function was stored in a temporary priority_queue pointer name 'qu', whose contents were then copied to the global variable 'queue'. 'queue' together with the queue name ('queue.qname') were then passed as arguments to the save function to save progress.

```
 priority_queue * qu;
qu = create_q(size2);
queue = *qu;

printf("\nQueue successfully generated\n\n");
if(!store(&queue, queue.qname))
{
 printf("\nUnable to save '%s' to file\n\n",queue.qname);
 }
```

Being a menu function, this function had a return of void.

The next choice, being choice 2, called the ‘enqueue_queue_fun’. This function first asked the user for the name of the queue that they would like to enqueue. This input was stored directly in ‘queue.qname’. Following the appropriate validation, the queue name was then passed on to the load function so as to load the queue to be used.

If the load function returns false, the function will return and will loop back to the menu. Otherwise, the queue has been successfully loaded and the function will commence. Once loaded, the size of the queue is tested, and if the queue size is equal to its max size, the function will return back to the main menu. Otherwise, the user is prompted to input the node data, including title, data of release, genre and so on. The appropriate input validation is carried out to ensure that the node stores valid data. If the user decides to press enter when prompted to enter a certain input related to a node, the data item in question will store “N/A” as its input. This has been implemented for purposes of flexibility, as not all knowledge of the film may be available.

Once all data input has been completed, including a valid, in-range, input for the film rating, the ‘node’ variable holding the data is passed as an argument to the enqueue function together with the queue which has been loaded. If the enqueue function returns false, the operation is halted and the user is returned to the main menu, otherwise, the function goes on to store the queue to file.

Similarly to the ‘enqueue_queue_fun’ function, the ‘dequeue_queue_fun’ function prompts the user to input the name of the queue to be operated upon. Once again the size of the queue is tested, as if the queue is empty, the function will halt and return to the main menu. Otherwise, the function will proceed to dequeue the highest priority element through the ‘dequeue’ function, passing the address of the loaded queue as an argument. The return of this function is stored in a previously declared Node * variable named tempn, which will halt the function if the return address if NULL, signifying that the dequeue operation has failed. Otherwise the function proceeds to output the data of the dequeued node on screen:

```
tempn = dequeue(&queue);
if(tempn == NULL)
{
    return;
}
node = *tempn;
printf("The following Movie has been removed from '%s': \n\n",
queue.qname);
printf("Title: %s\n", node.title);
printf("Genre: %s\n", node.genre);
printf("Director: %s\n", node.director);
printf("Actors: %s\n", node.actors);
printf("Length: %d\n", node.length);
printf("Release date: %s\n", node.release_date);
printf("Rating: %0.1lf\n\n", node.priority);
```

An attempt to store the newly dequeued queue is then made.

The ‘peek_fun’ function is very similar to the ‘dequeue_queue_fun’ function. As usual the user is prompted to input the name of the queue that they would like to operate upon, and the same validations are performed. The queue is then loaded from file and once again tested to contain valid nodes, and progress is halted if the queue is empty. The peek function is then called, passing the address of the loaded queue as an argument, and the return from this function is stored in a temporary node pointer, which is later output on screen if the return is not equal to NULL.

```
Node * no;
```

```
        no = peek(&queue);
        if(no == NULL)
        {
            return;
        }
        node = *no;
        printf("The highest rated Movie in '%s' is:\n\n",
       queue.qname);
        printf("Title: %s\n", node.title);
        printf("Genre: %s\n", node.genre);
        printf("Director: %s\n", node.director);
        printf("Actors: %s\n", node.actors);
        printf("Length: %d\n", node.length);
        printf("Release date: %s\n", node.release_date);
        printf("Rating: %0.1lf\n\n", node.priority);
```

Since the queue data has not been manipulated in any way, there is no need to store the data, therefore the function immediately returns to display the main menu again. Similarly, the 'size_fun' function does not require the data to be saved at the end of its execution, however the user must once again specify which queue they would like to operate upon. Once the usual validation and loading operations are completed, the size of the queue, together with the maximum size of the queue are output on screen.

```
printf("'%s' currently contains %d elements out of a maximum of %d\n\n",
queue.qname, size(&queue), queue.max_size);
```

The function 'merge_fun', is the 6th choice in the menu, and allows the user to merge the contents of two previously created queues and store the outcome in a third queue.

The user is first prompted to input the name of the first of two queues which they would like to merge. Standard validation checks are made prior to loading the queue. The same procedure is then followed after asking the user to input the second queue to be merged. (Note: these queues must both be previously created, however they can both be bare).

The merge function is then called and ‘q1’ and ‘q2’ are passed as parameters, bearing the address of the first and second loaded queues respectively. The return from the merge function is stored in a temporary priority_queue pointer named ‘qu2’. If ‘qu2’ is found to hold the NULL pointer then the operation is halted and the user is returned to the main menu, otherwise the data from ‘qu2’ is copied to the priority_queue variable ‘q3’:

```
priority_queue * qu2;
qu2 = merge(&q1, &q2);

if(qu2 == NULL)
{
        return;
}
q3 = *qu2;
```

‘q3’ is then stored to file.

The final functional choice is that of queue truncation. The ‘truncate_fun’ function prompts the user to select which queue they would like to truncate. Following validation the queue is loaded from memory. The clear function is then called, with the address of the loaded queue being passed as its argument. If this function has succeeded in clearing the queue then the queue is saved to file, otherwise the user is returned to the main menu.

```
if(clear(&queue))
 {
 printf("'%s' has successfully been truncated\n\n", queue.qname);
 if(!store(&queue, queue.qname))
 {
  printf("Unable to store queue '%s' to file \n\n", queue.qname);
 }
 return;
 }
  printf("Clear unsuccessful\n");
```

If the user decides to select the 8th choice, the test value for the do-while loop is altered, resulting in the termination of the program. Since each queue is stored after each individual operation, there is no need to call the store function when exiting the program. Any value outside the range 1-8 will result in the default case within the switch statement to be selected, warning the user of the invalid selection.

3) Testing

Testing is an integral part of software design and code optimisation. A number of debug trials were carried out so as to understand the cause of a number of problems and to solve these problems. Once these internal problems were solved, further console testing was required to ensure that the end product works and that the program runs in an easily understandable and relatively user friendly manner. We shall start by testing the main client application for validation errors, and we will then ensure that all functions within the code are working.

3.1) Data Validation and Client Application Testing

Data validation is a vital characteristic of a well-planned program. It ensures that all entered data is valid and relative to the subject. This ensures that the data input by the user is what they really wanted to input, meaning that error correction is allowed, and that the data input can be processed by the program.

We shall start by testing general data validation by entering out of range values in menus and when prompted for input.

```
                         ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

9
Invalid entry

0
Invalid entry

-1
Invalid entry

h
Invalid entry
```

```
1
                   ***** CREATE NEW QUEUE *****

Please enter the maximum number of elements
to be stored in the queue to be created:
```

As may be observed, the main menu passes the input validation test, with any input outside of the range, or out of context such as a letter, is disregarded and the user is informed of their invalid input

```
                   ***** CREATE NEW QUEUE *****
Please enter the maximum number of elements
to be stored in the queue to be created:

h
Invalid Input

0
Invalid Input

-1
Invalid Input

12
Please enter the name of the queue that you would like to create:

* enter *
Invalid input

myq
Queue successfully generated
```

The same may be said for the create queue function, with any invalid data inputs being rejected.

```
                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

2
                    ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:

my
Unable to open file 'my'
Unable to load queue

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

2
                    ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:

myq
We will now enqueue 'myq' with elements
Please enter the Movie Title: * enter *
Please enter the Movie Genre: * enter *
Please enter the Movie Length: * enter *
* enter *
2
Please enter the Movie Release Date: * enter *
Please enter the Movie's director: * enter *
Please enter the Movie's Main Actors: * enter *
Please enter the Movie Rating (0 - 10): * enter *
12
Invalid input!
10.1
Invalid input!
-1
Invalid input!
```

```
h
Invalid input!
7
Element successfully queued!
```

As for the enqueue menu, when the name of a file which does not exist is provided, the program informs the user that the file was not located and returns back to the main menu. Once a valid file name is provided, the user is asked to input the relevant movie data. As was explained in the previous section, the input for movie data was implemented in a way that allows the user to leave out some data, such as the title or genre. These data elements will be replaced with the string "N/A". However, other vital data elements such as the priority must be input, as they dictate the position of the movie in the queue. Therefore, any invalid data, be it out of range or irrelevant, is disregarded. The user is repeatedly prompted to input a value between 0 and 10, both inclusive.

```
                        ***** MENU *****
1) Create new queue                  2) Enqueue element
3) Dequeue highest rated element     4) View highest rated element
5) Queue size                        6) Merge two queues
7) Truncate queue                    8) Quit

3
                   ***** DEQUEUE ELEMENT *****
Please enter the name of the queue you would like to dequeue:
my
Unable to open file 'my'

                        ***** MENU *****
1) Create new queue                  2) Enqueue element
3) Dequeue highest rated element     4) View highest rated element
5) Queue size                        6) Merge two queues
7) Truncate queue                    8) Quit

3
                   ***** DEQUEUE ELEMENT *****
Please enter the name of the queue you would like to dequeue:
```

```
myq
We will now dequeue 'myq'

The following Movie has been removed from 'myq':

Title: N/A
Genre: N/A
Director: N/A
Actors: N/A
Length: 2
Release date: N/A
Rating: 7.0
```

As was previously explained, the fields left blank have now been replaced by the string “N/A”. Since the remainder of the functions rely on the same input validation methods, due to the fact that the load function is common to all functions, testing of the above samples is proof enough that the remainder of the program passes the input validation test. The next step is to test the functionality of the different operations.

3.2.1) Program Functionality – Circular Buffer

Over and above general program presentation and data validation, the inner workings of the program must be sound, and must perform their tasks without error. Therefore it is of paramount importance to rigorously test all the different functions with data of different levels of validity. This will ensure that the software is fully functional and can perform the job that it is supposed to do.

We shall start by testing the functions to create a queue and enqueue elements.

```
                    ***** CREATE NEW QUEUE *****
Please enter the maximum number of elements
to be stored in the queue to be created:
5
Please enter the name of the queue that you would like to create:
myq
Queue successfully generated

                         ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

2
                     ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:
myq
We will now enqueue 'myq' with elements
Please enter the Movie Title: Titanic
Please enter the Movie Genre: Drama
Please enter the Movie Length: 120
Please enter the Movie Release Date: * enter *
Please enter the Movie's director: James Cameron
Please enter the Movie's Main Actors: Leonardo DiCaprio
Please enter the Movie Rating (0 - 10): 8
Element successfully queued!
```

```
                    ***** MENU *****
1) Create new queue                  2) Enqueue element
3) Dequeue highest rated element     4) View highest rated element
5) Queue size                        6) Merge two queues
7) Truncate queue                    8) Quit

2
                ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:
myq
We will now enqueue 'myq' with elements
Please enter the Movie Title: The Guest
Please enter the Movie Genre: Action
Please enter the Movie Length: 98
Please enter the Movie Release Date: 12/4/14
Please enter the Movie's director: * enter *
Please enter the Movie's Main Actors: * enter *
Please enter the Movie Rating (0 - 10): 4.9
Element successfully queued!
```

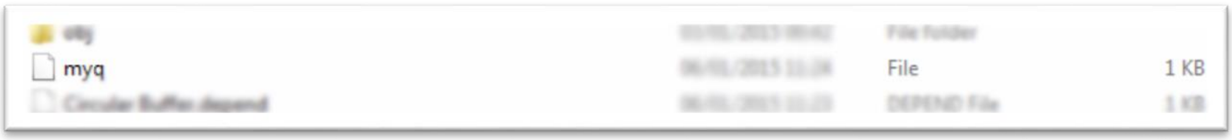
The queue was further populated with another 2 elements, bringing the total number of elements up to 4. The ‘Queue Size’ functionality was then tested. The following were the results:

```
                  ***** QUEUE SIZE *****

Please enter the name of the queue you would like to operate upon:
myq
'myq' currently contains 4 elements out of a maximum of 5

                    ***** MENU *****
1) Create new queue                  2) Enqueue element
3) Dequeue highest rated element     4) View highest rated element
5) Queue size                        6) Merge two queues
7) Truncate queue                    8) Quit

2
                ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:
myq
```

```
We will now enqueue 'myq' with elements
Please enter the Movie Title: Pulp Fiction
Please enter the Movie Genre: Action
Please enter the Movie Length: 162
Please enter the Movie Release Date: 10/9/94
Please enter the Movie's director: Quintin Tarantino
Please enter the Movie's Main Actors: John Travolta & Samuel L. Jackson
Please enter the Movie Rating (0 - 10): 7.7
Element successfully queued!

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

2
                    ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:
myq
Queue is full

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                     ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:

myq
'myq' currently contains 5 elements out of a maximum of 5
```

As may be seen, a queue named ‘myq’ was successfully created to hold a maximum of 5 elements. This queue was successfully populated with nodes, having data input by the user. The size function was also tested, succeeding in returning the correct number of nodes presently queued as well as the correct maximum queue size. Once the queue was populated with 5 elements, its maximum amount, attempts to further populate the queue were futile, as the ‘enqueue’ function recognised that the queue had reached its set limit.

As may have been observed in previous sections. The load function is called at the start of every function, while the save function is called at the end of most of the functions. Therefore, with 3 functions being called and executed, it may be concluded that both the load and save functions perform the respective jobs. This conclusion may be reinforced further by the presence of the text file named ‘myq’ within the same folder as the executable.



```
obj                      [illegible]   File folder
myq                      [illegible]   File            1 KB
Circular Buffer.depend   [illegible]   DEPEND File     1 KB
```

Figure 8

Now that we have a fully populated queue, we can test the functionality of the peek, dequeue and truncate operations.

```
                      ***** MENU *****
1) Create new queue                 2) Enqueue element
3) Dequeue highest rated element    4) View highest rated element
5) Queue size                       6) Merge two queues
7) Truncate queue                   8) Quit

4
            ***** HIGHEST RATED ELEMENT *****
Please enter the name of the queue you would like to operate upon:
myq
The highest rated Movie in 'myq' is:
Title: Titanic
Genre: Drama
Director: James Cameron
Actors: Leonardo DiCaprio
Length: 120
Release date: N/A
Rating: 8.0
```

```
                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                     ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
myq
'myq' currently contains 5 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

3
                  ***** DEQUEUE ELEMENT *****
Please enter the name of the queue you would like to dequeue:
myq
We will now dequeue 'myq'
The following Movie has been removed from 'myq':

Title: Titanic
Genre: Drama
Director: James Cameron
Actors: Leonardo DiCaprio
Length: 120
Release date: N/A
Rating: 8.0

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                      ***** QUEUE SIZE *****
```

```
Please enter the name of the queue you would like to operate upon:
myq
'myq' currently contains 4 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                     2) Enqueue element
3) Dequeue highest rated element        4) View highest rated element
5) Queue size                           6) Merge two queues
7) Truncate queue                       8) Quit

3
                  ***** DEQUEUE ELEMENT *****
Please enter the name of the queue you would like to dequeue:
myq
We will now dequeue 'myq'
The following Movie has been removed from 'myq':

Title: The Grudge
Genre: Horror
Director: N/A
Actors: N/A
Length: 118
Release date: N/A
Rating: 8.0

                        ***** MENU *****
1) Create new queue                     2) Enqueue element
3) Dequeue highest rated element        4) View highest rated element
5) Queue size                           6) Merge two queues
7) Truncate queue                       8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
myq
'myq' currently contains 3 elements out of a maximum of 5
```

A couple of things may be noted from this test. First of all, when the peek function was called to return the data of the highest priority element, it did so without removing it from the queue, as supposed to. This was observed as after the queue function was called, the size function still returned a current size of 5.

Next when the dequeue function was called, the same element which was shown by the peek function, therefore correctly the element with the highest priority, was output on screen. This time, it was removed from the queue, as the size function now returned a reading of 4 nodes in the queue.

Another noteworthy observation is that when the dequeue function was called again, another different film was dequeued, however this film, ‘The Grudge’, had the same rating as the previously dequeued film, ‘Titanic’. However, since ‘Titanic’ had been enqueued before ‘The Grudge’, ‘Titanic’ was placed before it in the queue, proving that the enqueueing function is indeed functional.

We will now go on to test the dequeue function one more time, followed by the truncate function.

```
                  ***** DEQUEUE ELEMENT *****
Please enter the name of the queue you would like to dequeue:
myq
We will now dequeue 'myq'
The following Movie has been removed from 'myq':

Title: Pulp Fiction
Genre: Action
Director: Qeuintin Tarantino
Actors: John Travolta & Samuel L. Jackson
Length: 162
Release date: 10/9/94
Rating: 7.7

                        ***** MENU *****
1) Create new queue                  2) Enqueue element
3) Dequeue highest rated element     4) View highest rated element
5) Queue size                        6) Merge two queues
```

```
7) Truncate queue                8) Quit

5
                 ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
myq
'myq' currently contains 2 elements out of a maximum of 5
```

As you may appreciate, the node with the next highest priority was dequeued and the size of the queue was decreased by one. The truncate function is designed to empty the contents of the queue:

```
                    ***** MENU *****
1) Create new queue              2) Enqueue element
3) Dequeue highest rated element 4) View highest rated element
5) Queue size                    6) Merge two queues
7) Truncate queue                8) Quit

7
                 ***** TRUNCATE QUEUE *****
Please enter the name of the queue you would like to truncate:
myq
'myq' has successfully been truncated

                    ***** MENU *****
1) Create new queue              2) Enqueue element
3) Dequeue highest rated element 4) View highest rated element
5) Queue size                    6) Merge two queues
7) Truncate queue                8) Quit

5
                 ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
myq
'myq' currently contains 0 elements out of a maximum of 5
```

As may be seen, the contents of the queue ‘myq’ have successfully been truncating. The queue, which previously held two elements, is now empty, yet it may still be used to enqueue and operate upon more films.

We will now go on to test the merge function and its ability to merge the contents of two queues into one single queue.

```
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
myq
'myq' currently contains 4 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

4
                    ***** HIGHEST RATED ELEMENT *****
Please enter the name of the queue you would like to operate upon:
myq
The highest rated Movie in 'myq' is:

Title: The Shawshank Redemption
Genre: Crime
Director: Frank Darabont
Actors: Morgan Freeman, Tim Robbins
Length: 142
Release date: 14/10/1994
Rating: 9.3

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
```

```
                  ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
yourq
'yourq' currently contains 3 elements out of a maximum of 10

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

4
                    ***** HIGHEST RATED ELEMENT *****
Please enter the name of the queue you would like to operate upon:
yourq
The highest rated Movie in 'yourq' is:

Title: Fury
Genre: War
Director: David Ayer
Actors: Brad Pitt & Logan Lerman
Length: 142
Release date: 6/10/14
Rating: 9.7
```

In order to test the merge function, two queues named ‘myq’ and ‘yourq’ have been created. ‘myq’ currently holds 4 elements out of a maximum 5 while ‘yourq’ currently holds 3 elements out of a maximum 10. The highest rated film in ‘myq’ is ‘The Shawshank Redemption’ with a rating of 9.3, while the highest rated film in ‘yourq’ is ‘Fury’ with a rating of 9.7. We shall now merge the two queues.

```
                    ***** QUEUE MERGE *****
Please enter the name of the first queue that you would like to merge:
yourq
Please enter the name of the queue you would like to operate upon:
myq
Please enter the name of the queue that you would like to create:
ourq
Merge successful
```

```
                         ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                     ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
ourq
'ourq' currently contains 7 elements out of a maximum of 15

                         ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

4
                     ***** HIGHEST RATED ELEMENT *****

Please enter the name of the queue you would like to operate upon:
ourq
The highest rated Movie in 'ourq' is:

Title: Fury
Genre: War
Director: David Ayer
Actors: Brad Pitt & Logan Lerman
Length: 142
Release date: 6/10/14
Rating: 9.7
```

With this, testing of the circular buffer functions is complete, with all functions performing their assigned tasks without error. We must now test the functions used in the linked list data structure.

3.2.2) Program Functionality – Linked List

As was previously explained, code functionality is a must, and it is important that each and every function is rigorously tested before being packaged as a whole program. The linked list data stricture will be implementing the same client application, therefore, data validation tests from section 3.1 apply to this implementation. However, the testing of the individual functions is still required, and therefore we shall now begin by testing the functions designed to create and enqueue a linked list.

```
                         ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

1
                    ***** CREATE NEW QUEUE *****
Please enter the maximum number of elements
to be stored in the queue to be created:
5
Please enter the name of the queue that you would like to create:
Films
Queue successfully generated

                         ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

2
                     ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:
Films
We will now enqueue 'Films' with elements
Please enter the Movie Title: Django Unchained
Please enter the Movie Genre: Action
Please enter the Movie Length: 165
Please enter the Movie Release Date: 25/12/12
Please enter the Movie's director: Quentin Tarantino
Please enter the Movie's Main Actors: Jamie Foxx & Leonardo DiCaprio
Please enter the Movie Rating (0 - 10): 8.5
Element successfully queued!

                     ***** MENU *****
1) Create new queue                   2) Enqueue element
3) Dequeue highest rated element      4) View highest rated element
5) Queue size                         6) Merge two queues
7) Truncate queue                     8) Quit

5
                 ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 1 elements out of a maximum of 5

                     ***** MENU *****
1) Create new queue                   2) Enqueue element
3) Dequeue highest rated element      4) View highest rated element
5) Queue size                         6) Merge two queues
7) Truncate queue                     8) Quit

2
                 ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:
Films
We will now enqueue 'Films' with elements
Please enter the Movie Title: Slumdog Millionaire
Please enter the Movie Genre: Drama
Please enter the Movie Length: 120
Please enter the Movie Release Date: 25/12/08
Please enter the Movie's director: * enter *
Please enter the Movie's Main Actors: Dev Patel
Please enter the Movie Rating (0 - 10): 8
Element successfully queued!

                     ***** MENU *****
1) Create new queue                   2) Enqueue element
3) Dequeue highest rated element      4) View highest rated element
5) Queue size                         6) Merge two queues
```

```
7) Truncate queue                 8) Quit

5
                 ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 2 elements out of a maximum of 5

                     ***** MENU *****
1) Create new queue               2) Enqueue element
3) Dequeue highest rated element  4) View highest rated element
5) Queue size                     6) Merge two queues
7) Truncate queue                 8) Quit

2
                 ***** ENQUEUE ELEMENT *****
Please enter the name of the queue you would like to enqueue:
Films
We will now enqueue 'Films' with elements
Please enter the Movie Title: Transformers
Please enter the Movie Genre: Action
Please enter the Movie Length: 144
Please enter the Movie Release Date: 3/07/07
Please enter the Movie's director: Michael Bay
Please enter the Movie's Main Actors: * enter *
Please enter the Movie Rating (0 - 10): 8
Element successfully queued!

                     ***** MENU *****
1) Create new queue               2) Enqueue element
3) Dequeue highest rated element  4) View highest rated element
5) Queue size                     6) Merge two queues
7) Truncate queue                 8) Quit

5
                 ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 3 elements out of a maximum of 5
```

As the testing has proven, the queue creation, node enqueuement and size function are all working as was planned. A new queue called ‘Films’ was created to hold a maximum of 5 elements. The queue was then populated with data entered by the user. As one may observe, each time a film was added to the queue, the queue size increased. It is now required to test the dequeue function and the peek function.

```
                    ***** HIGHEST RATED ELEMENT *****
Please enter the name of the queue you would like to operate upon:
Films
The highest rated Movie in 'Films' is:
Title: Django Unchained
Genre: Action
Director: Quentin Tarantino
Actors: Jamie Foxx & Leonardo DiCaprio
Length: 165
Release date: 25/12/12
Rating: 8.5

                        ***** MENU *****
1) Create new queue                   2) Enqueue element
3) Dequeue highest rated element      4) View highest rated element
5) Queue size                         6) Merge two queues
7) Truncate queue                     8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 3 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                   2) Enqueue element
3) Dequeue highest rated element      4) View highest rated element
5) Queue size                         6) Merge two queues
7) Truncate queue                     8) Quit

3
                   ***** DEQUEUE ELEMENT *****
Please enter the name of the queue you would like to dequeue:
Films
```

```
We will now dequeue 'Films'
The following Movie has been removed from 'Films':

Title: Django Unchained
Genre: Action
Director: Quentin Tarantino
Actors: Jamie Foxx & Leonardo DiCaprio
Length: 165
Release date: 25/12/12
Rating: 8.5

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 2 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

3
                   ***** DEQUEUE ELEMENT *****

Please enter the name of the queue you would like to dequeue:
Films
We will now dequeue 'Films'
The following Movie has been removed from 'Films':

Title: Slumdog Millionaire
Genre: Drama
Director: N/A
Actors: Dev Patel
```

```
Length: 120
Release date: 25/12/08
Rating: 8.0

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 1 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

3
                   ***** DEQUEUE ELEMENT *****
Please enter the name of the queue you would like to dequeue:
Films
We will now dequeue 'Films'
The following Movie has been removed from 'Films':

Title: Transformers
Genre: Action
Director: Michael Bay
Actors: N/A
Length: 144
Release date: 3/07/07
Rating: 8.0


                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
```

```
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 0 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

3
                   ***** DEQUEUE ELEMENT *****

Please enter the name of the queue you would like to dequeue:
Films
Queue is empty
```

In the above command line, the dequeue and peek function were both tested. When the peek option was initially chosen, the Bode containing the film ‘Django Unchained’ with a rating of 8.5 was output on screen. Of the three films in the queue, this was indeed the highest rated one and therefore was rightly output. Also when the size function was called, following the call of the peek function, the size was observed to remain the same, signifying that the element was not removed from the queue. From this it can be concluded that the peek function is working fine.

The next function to be tested was the dequeue function. This function was called and the same output as that of the peek function was observed. The highest rated movie was still ‘Django Unchained’ with a rating of 8.5, and thus it was rightly output on screen. The size function was then called again, and it was noticed that the queue now had one less element, which is expected as the dequeue function is designed to remove the highest rated element from the list.

These steps were then repeated. The second film to be output on screen was 'Slumdog Millionaire' with a rating of 8.0. 'Transformers', the remaining element in the queue, also had a rating of 8.0, however since it was input after the film 'Slumdog Millionaire', it was rightfully placed behind it in the queue, respecting the first-come-first-served principle. These therefore truly concludes that the dequeue function is working as designed. Also, one may note that once the queue was empty, any request to further dequeue data resulted in a warning being output on screen.

The final two functions which are yet to be tested are the merge function and the truncation function. The merge function is used to merge the contents of two queues and store them in a third named queue.

```
                        ***** MENU *****
1) Create new queue                     2) Enqueue element
3) Dequeue highest rated element        4) View highest rated element
5) Queue size                           6) Merge two queues
7) Truncate queue                       8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Films
'Films' currently contains 4 elements out of a maximum of 5

                        ***** MENU *****
1) Create new queue                     2) Enqueue element
3) Dequeue highest rated element        4) View highest rated element
5) Queue size                           6) Merge two queues
7) Truncate queue                       8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
Movies
'Movies' currently contains 3 elements out of a maximum of 10
```

```
                    ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

6
                    ***** QUEUE MERGE *****
Please enter the name of the first queue that you would like to merge:
Movies
Please enter the name of the queue you would like to operate upon:
Films
Please enter the name of the queue that you would like to create:
what to watch
Merge successful

                       ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
what to watch
'what to watch' currently contains 7 elements out of a maximum of 15

                       ***** MENU *****
1) Create new queue                    2) Enqueue element
3) Dequeue highest rated element       4) View highest rated element
5) Queue size                          6) Merge two queues
7) Truncate queue                      8) Quit

4
                    ***** HIGHEST RATED ELEMENT *****
Please enter the name of the queue you would like to operate upon:
what to watch
The highest rated Movie in 'what to watch' is:

Title: Gone Girl
```

```
Genre: Drama
Director: David Finch
Actors: Ben Affleck
Length: 149
Release date: 3/10/14
Rating: 9.5
                        ***** MENU *****
1) Create new queue                     2) Enqueue element
3) Dequeue highest rated element        4) View highest rated element
5) Queue size                           6) Merge two queues
7) Truncate queue                       8) Quit

7
                    ***** TRUNCATE QUEUE *****
Please enter the name of the queue you would like to truncate:
what to watch
'what to watch' has successfully been truncated

                        ***** MENU *****
1) Create new queue                     2) Enqueue element
3) Dequeue highest rated element        4) View highest rated element
5) Queue size                           6) Merge two queues
7) Truncate queue                       8) Quit

5
                    ***** QUEUE SIZE *****
Please enter the name of the queue you would like to operate upon:
what to watch
'what to watch' currently contains 0 elements out of a maximum of

                        ***** MENU *****
1) Create new queue                     2) Enqueue element
3) Dequeue highest rated element        4) View highest rated element
5) Queue size                           6) Merge two queues
7) Truncate queue                       8) Quit

4
                    ***** HIGHEST RATED ELEMENT *****
Please enter the name of the queue you would like to operate upon:
what to watch
Queue is empty
```

Above, one may observe the effects of the merge function. First, two queues were populated. The first ‘films’ held 4 movies with a maximum of 5, while the second queue ‘movies’ held 3 movies with a maximum of 10. The merge function was called and the two queue names were inputted. The new queue to be formed was named ‘what to watch’. Once merged, it was observed that the new queue ‘what to watch’ now held 7 movies with a maximum capacity for 15. The peek function was then called and it was observed that the film with the greatest rating was called ‘Gone Girl’ with a rating of 9.5.

The final function that was tested was the queue truncation. The queue called ‘what to watch’ was inputted to the truncation function and the result was that it was returned with a total size of 0 out of a maximum capacity for 15 elements. Therefore, it may be concluded that both the merge and truncation functions performed the tasks which they were supposed to.

The program as a whole performs as supposed to, with each function executing its own specific task successfully. The program as a whole implements both data validation and functionality, and therefore, in general, functions effectively.

4) Conclusion

After designing, building, altering, debugging and testing the programs, it may be concluded, that although both data structures perform the job of a priority queue, the linked list is much more efficient in doing so. This is mainly due to the fact that data is allocated to each node as the program executes, and does not need to be determined by the user. This is a big bonus, as the queue can grow or shrink indefinitely to the demands of the user.

When planning and building the program, modularity and code reusability was given great importance. The ability to use one function multiple times is both time efficient, as the same lines of code do not need to be retyped, but also makes testing an easier process. The issue of portability must also be kept in mind, as certain features may not be portable to different systems. This was especially so when it came to loading and writing the data to file.

In conclusion, the program runs smoothly and meets the reuirements of a priority queue, offering basic operations to the user with minimal knowledge in Computer Science required.

References

[1] (). *Array Data Structure*. Available: http://www.cs.cmu.edu/~adamchik/15-121/lectures/Arrays/arrays.html

[2] (7 Aug 2013). *Ring buffer basics*. Available: http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer

[3] S. Prata, *C Primer Plus.* 2013

[4] (). *Heaps*. Available: http://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html