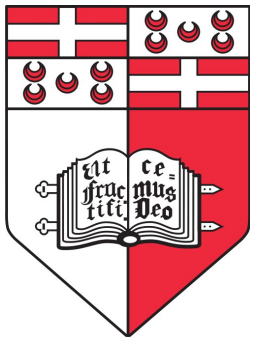
Programming Assignment
The 'priority queue' abstract data type

CPS1000
Programming Principles in C
Instructor: Mark J. Vella

Linux Mint Debian Edition

Stefan Mallia
0128591m
Bachelor of Science (Honours)
Computing Science
and
Statistics and Operations Research



University of Malta
Department of Computer Science

Table of Contents

1.0 Data Structures

1.1 The Array

An array is a data structure which stores multiple elements in a single variable. The elements in an array are usually of the same type but some languages allow the user to store different data types in a single array.

The concept of an array is that of a linear storage of information with a specified length. Often the way an array is stored in hardware is also linear.

Arrays can be multidimensional with no limit on the number of dimensions possible. Figure 1 is an illustration of a simple one-dimensional array and also of a two-dimensional array. Also shown in Figure 1 are the index numbers used to access a particular element from an array. Programming languages usually start their index from 0 and continue to n-1 as shown; this is the case for C where array index are appended to the variable within square brackets [ ]. To represent more than one dimension multiple square brackets can be used next to each other [ ][ ].

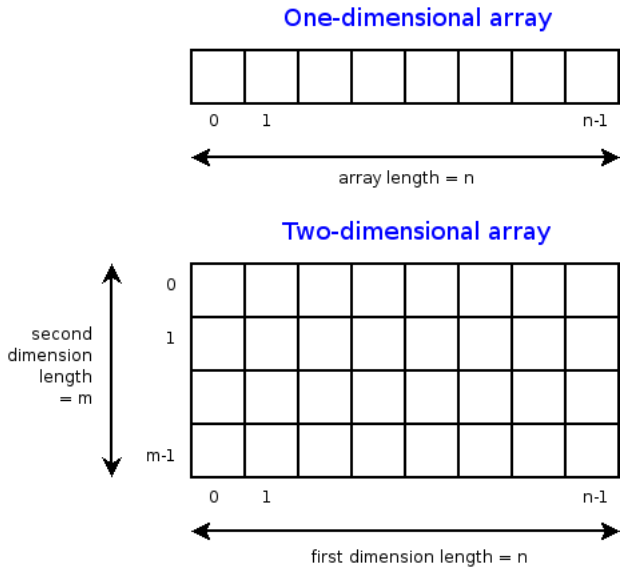


Figure 1[1]

1 Source: http://ycpcs.github.io/cs201-summer2014/notes/javaArrays.html

1.2 Circular/Ring Buffer

A circular buffer is a memory allocation approach where memeory is recycled when an incremental index reaches a limit. It is conceptually similar to a simple array but with the difference that both ends of the array are connected to each other to form a circle.

This assignment's implemented ring buffer reserves n elements but only allows n-1 to be used. This is done so that the recognition between an empty queue and a full queue is made easier. In this manner an empty queue can be identified when the start and end indicators point to the same node and a full queue can be identified when the start indicator points to node 0 and the end indicator points to node n-1.

The end indicator points to the next node where data is to be written. Both the start and end indicators are shifted forward by one when the node at index n-1 (last node, meaning that queue is already full) is written over. This means the information at node n-1 is never used. An illustration of a cicular buffer is shown in Figure 2.

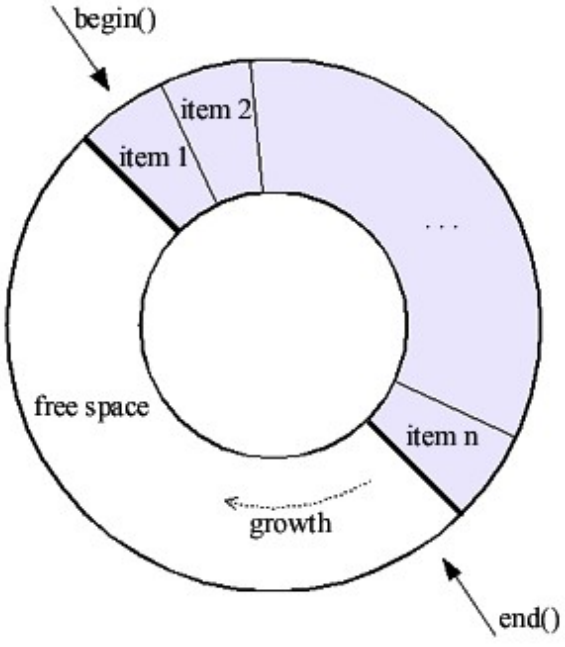


Figure 2[2]

2 Source: http://www.boost.org/doc/libs/1_39_0/libs/circular_buffer/doc/circular_buffer.html

1.3 Linked List

A linked list is a data structure which consists of nodes which form a sequence. Each node is split into two parts: the element data that the node contains and a pointer to the address of the next node in memory.

Unlike the circular buffer this is a dynamically allocated data structure, meaning that the number of nodes allocated to memory is equal to the amount of elements stored. The list can then shrink or grow as needed during run time. While this is an efficient way of storing information in memory (especially when the list has the potential to grow very large), it comes with the draw back of being less efficient than the circular buffer when it comes to accessing elements from the queue. To access an element in a linked list, each element must be iterated through until the desired element is found. Figure 3 shows an illustration of a linked list.

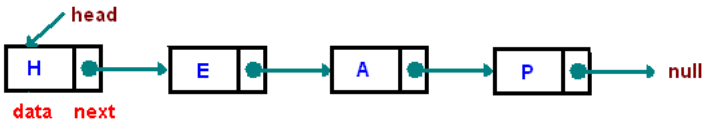


Figure 3[3]

3 Source: http://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

1.4 Binary Heap

A data structure that stores data in the form of a binary tree. A representation of a binary heap is shown in Figure 4.

The root is the first term of the binary head and is illustrated at the top of the tree. It can either be a minimum for a min-heap implementation or a maximum for a max-heap implementation. In the case of a priority queue where the highest priority is dequeued first a maximum binary heap would be appropriate. In such a case each node's children will either be smaller than or equal to their parent.

Considering a k^{th} element and an array with a starting index of 1:

- the left child of a parent node is located at 2k indexed element
- the right child of a parent node is located at 2k+1 indexed element
- the parent is located at k/2 index (rounding down)

An insert is added to the end of the heap at the end of the array, which is represented by the lowest layer and furthest to the right of the tree. Each parent can have only two children nodes so in such a case the parent node to the right would be assigned new child nodes. The respective parent is then compared with the inserted node and if the parent is smaller than the inserted node they would be switched. In the case of a removal, the root node is removed and in its place is put the last element of the array. This element is then switched with its largest child multiple times until its correct place is reached.

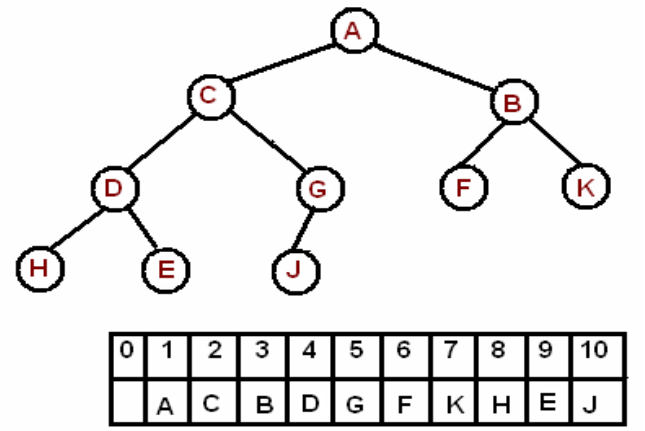


Figure 4[4]

4 Source: http://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html

2.0.0 Software Testing

Two types of data structures were used to implement a priority queue. This section of the report gives a demonstration of each using a simple client.

The client prompts the user with a menu selection where the user can choose to either enter the linked list menu or to enter the ring buffer menu.

```
stefan@stefanlinux:~/Cprogramming/Principles Assignment$ ./Client

Welcome to the queueing application

a. Linked List
b. Ring Buffer

Please make a selection (a or b or enter 'q' to quit):
```

In the main c file, Client.c, there are two structs with the purpose of holding a list of queues. The two structs are similar as one holds a list of linked list queues and the other holds a list of ring buffers. Both structs use a linked list approach so both structs have a pointer which is intended to point to the next struct.

```
//A node for a linked list of linked list priority queues
typedef struct LLarray
{
        linkedlists * ll;
        char * name;
        struct LLarray * next;
}LLarray;

//A node for a linked list of ringbuffer priority queues
typedef struct RBarray
{
        RingBuffer * rb;
        char * name;
        struct RBarray * next;
}RBarray;
```

Both of these structs have their first node allocated to memory at the start of the program with all their values being set to NULL being that each element is a pointer.

The menu client is split into two main parts using switch statements. At the start of the program the user is prompted with two choices, either to use the linked list menu or to use the ring buffer menu. Making a choice of either 'a' or 'b' gives the program the input to a switch statement that then uses a 'goto' operator which directs the code to either the linked list menu code or the ring buffer menu code.

These two code sections are what make up most of the code are are very similar. However, despite being similar the difference in requirements for the linked list and ring buffer necessitated the need to divide their code into two, including separate functions.

2.1.0 LinkedList

Selecting the linked list menu prompts the user with the following choices:

```
Actions:

a. Create a Linked List Queue
b. Make an entry into a queue
c. Dequeue highest priority element
d. Highest priority element
e. Check whether queue is empty
f. Size of queue
g. Merge two queue's
h. Clear queue of contents
i. List existing queues
j. Store existing queue to file
k. Load queue from file

Please make a selection (enter 'q' to return to main menu):
```

2.1.1 Creating a new linked list

Creating a new queue prompts the user for a name and maximum size. Confirmation that queue is created is given by the last line in the below console snippet.

```
Name of Linked List:
Queue1

Maximum size of Linked List:
5

Your queue name is: Queue1
```

The a linked list is created by using a malloc function to create a struct to hold the linked list nodes. The first struct shown holds the max size value and the pointer to the head node. The second struct contains the element data and the pointer to the next node.

```
typedef struct linkedlists
{
        int max_size;
        struct pqnode * headnode;
```

```
}linkedlists;



typedef struct pqnode
{
        int element;
        int priority;
        struct pqnode * next;
}pqnode;
```

2.1.2 Making an entry into linked list

Entering an element into a queue requires an identifying queuename and the element data.

```
Enter the queue name:
Queue1

Enter an integer element:
1

Enter an integer priority:
7
```

All other functions have the same pattern of usability in the console.

This is handled by the enqueue() function which first checks for any existing nodes in the queue and then assigns the data to a newly created node.

If the headnode is unassigned and equal to NULL, the function goes through the first code block and assigns the relevant values. Element and priority are assigned their values from the function arguments. The 'next' pointer is always set to null until the enqueue function is called again where it would be assigned a new memory allocation as a new node.

```
linkedlist->headnode = (pqnode *) malloc(sizeof(pqnode));

linkedlist->headnode->element = x;
linkedlist->headnode->priority = p;
linkedlist->headnode->next = NULL;
```

The function also contains a counter, which is initially set to 2, to count the size of the linked list. This is to ensure that it does not exceed the maximum number of nodes specified by the linkedlists struct.

2.1.3 Dequeueing

The dequeue function iterates through the whole linked list and notes the highest priority element. Once the highest priority is recorded in a temporary variable, the function iterates throught the linked list once more until the node is found. It is then removed using a clear() function and its neighboring queues are connected.

2.1.4 Peeking at highest priority

Similar to the dequeue() function, the peek function iterates through the linked list nodes until it finds the highest priority which it assigns to a temporary variable. It then iterates again until the highest priority is found but instead of clearing the node t returns it and is printed to console.

2.1.5 Empty queue check

This function simply checks whether the head node of the linked list is NULL

2.1.6 Size of queue

This function iterates and counts the number of nodes in the linked list.

2.1.7 Queue merge

The max size of both input queues are combined for the new merged queue and stores it in the first linkedlists struct. The second inputted linkedlists struct is freed at the end, leaving only the first input linkedlist.

The function takes the first input queue and iterates to the end. Once at the end, the 'next' pointer is set to the head node of the second input queue. The linkedlists struct is freed but the individual nodes are not. This has the effect of 'attaching' the second linked list to the

end of the first.

2.1.8 Clearing queue

This function iterates through the linked list and clears each node. The client also gives the option of deleting the whole queue.

2.1.9 Storing queue to file

A binary write was used as an approach to storing the linked lists. For linked list queues, the user inputs the name of an existing queue and the function appends 'll.bin' to the end of the name to be used as a filename. The 'll' is used to distinguish from the ring buffer files, which are appended by 'rb.bin'.

```
Name of Linked List:
Queue1

Queue1ll.bin
Write Successful
```

Firstly, the linkedlists struct was written to binary file with the headnode set to NULL since this is a pointer and would have to be memory allocated again on load. Next, the size of the linked list is written for reference during load. Lastly, the linkedlist headnode is restored and iterated through so that each node is written to file. Each node's 'next' pointer was also set to NULL and restored.

```
while(Current != NULL)
{
	holdnext = Current->next;
	Current->next = NULL;

	fseek(pFile, 0, SEEK_END);
	fwrite(Current, sizeof(pqnode), 1, pFile);


	Current->next = holdnext;

	Current = Current->next;
}
```

2.1.10 Loading queue from file

An approach similar to the storing process. The user does not need to enter the filename 'Queue1ll.bin' but rather needs only input 'Queue1', the name of the queue.

```
Name of Linked List:
Queue1

Queue1ll.bin
Read Successful
```

While the function iterates and reads through the binary file it also allocates memory for each node and for the linked list struct.

```
linkedlist->headnode = (pqnode *) malloc(sizeof(pqnode));
current = linkedlist->headnode;



for(i = 1; i < numnodes; i++)
{
        current->next = (pqnode *) malloc(sizeof(pqnode));
        current = current->next;

        fread(current, sizeof(pqnode), 1, pFile);

        current->next = NULL;
}
```

2.2.0 RingBuffer

Selecting the ring buffer menu prompts the user with the following choices:

```
Actions:

a. Create a RingBuffer Queue
b. Make an entry into a queue
c. Dequeue highest priority element
d. Highest priority element
e. Check whether queue is empty
f. Size of queue
g. Merge two queue's
h. Clear queue of contents
i. List existing queues
j. Store existing queue to file
k. Load queue from file
Please make a selection (enter 'q' to return to main menu):

(enter 'q' to return to main menu):
```

The usability of this menu is identical to that of the linked list.

2.2.1 Creating a new ring buffer

The ring buffer also requires two structs to implement.

```
typedef struct    //element struct
{
        int elem;
        int priority;
}
Element;

typedef struct    //struct that contains elements
{
        int max_size;
        int start;
        int end;
        Element * elems;
}
RingBuffer;
```

The first struct contains the information of a single node. The second struct contains meta data on the ring buffer, such as the max size allowable, and it also contains the data itself

which comes in the form of an array of node structs.

Creating a new ring buffer requires that memory is first allocated to the Ringbuffer struct and second to all the nodes simultaneously to form an array.

```
RingBuffer * create_qR(int max_size)
{
	//using calloc to create an array of structs
	RingBuffer * ringbuffer = calloc(1, sizeof(RingBuffer));
	ringbuffer->max_size = max_size+1;//add one because one of the elements is not
used
	ringbuffer->start = 0;
	ringbuffer->end = 0;
	ringbuffer->elems = calloc(ringbuffer->max_size, sizeof(Element));

	return ringbuffer;

}
```

2.2.2 Making an entry into ring buffer

An entry is made into the array by assigning the inputted values to the end node (initially equal to the start node) and then incrementing by 1. After incrementing, if the end node is equal to the start node, then that means that the queue has been filled and the start node must also start being incremented unless data is dequeued.

The end nodes location is only a placeholder for incoming data and is not considered part of the queue.

2.2.3 Dequeueing

Dequeueing is simple as the ring buffer is sorted. A dequeue only requires the start indicator to be incremented.

2.2.4 Peeking at highest priority

Since the queue is sorted, returns the start node.

2.2.5 Empty queue check

If the start indicator is equal to the end indicator the queue is empty.

2.2.6 Size of queue

Iterates and counts through the ring buffer.

2.2.7 Queue merge

A new queue is created to contain the previous two queues. Since one node is always unused in a ring buffer, if two are combined then that leaves us with two unused nodes. Therefore one is removed when combining max size.

```
RingBuffer * rbnew = calloc(1, sizeof(RingBuffer));
rbnew->max_size = rb1->max_size + rb2->max_size - 1;
        //adding two queues together leaves two unused elements, one is removed
rbnew->start = 0;
rbnew->end = 0;
rbnew->elems = calloc(rbnew->max_size, sizeof(Element));
```

2.2.8 Clearing queue

Sets the start indicator equal to the end indicator to empty the queue. The client also gives the option of deleting the queue.

2.2.9 Storing queue to file

Since the ringbuffer is one chunk of memory (in contrast to the linked list), storing to file can be done at once. Files are saved with 'rb.bin' appended.

```
fseek(pFile, 0, SEEK_END);
fwrite(rb, sizeof(RingBuffer), 1, pFile);

printf("\nWrite Successful\n");
fclose(pFile);
```

2.2.10 Loading queue from file

The process is similar to that of the writing process but a memory allocation is made before loading.

3.0 Appendices

3.1 Appendix 1: Client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "LINKEDLIST.h"
#include "RINGBUFFER.h"

//A node for a linked list of linked list priority queues
typedef struct LLarray
{
	linkedlists * ll;
	char * name;
	struct LLarray * next;
}LLarray;

//A node for a linked list of ringbuffer priority queues
typedef struct RBarray
{
	RingBuffer * rb;
	char * name;
	struct RBarray * next;
}RBarray;

//ll linkedlist
//rb ringbuffer

void printqueuesll(LLarray * llheadnode); //will print existing ll queues to console
void printqueuesrb(RBarray * rbheadnode); //will print existing rb queues to console
char * getstring(); //a function for user string input
void appendllqueue(linkedlists * linkedlist, char * queuename, LLarray *
llheadnode);//adding an element to the ll for ll priority queues
LLarray * accessllqueue(char * queuename, LLarray * llheadnode);//accessing a particular
node of the ll for the ll priority queues
LLarray * clearllqueue(char * queuename, LLarray * llheadnode);//clearing a particular
node after having cleared its contained ll
void appendrbqueue(RingBuffer * RBqueue, char * queuename, RBarray *
rbheadnode);//adding an element to the ll for rb priority queues
RBarray * accessrbqueue(char * queuename, RBarray * rbheadnode);//accessing a
particular node of the ll for the rb priority queues
RBarray * clearrbqueue(char * queuename, RBarray * rbheadnode);//clearing a particular
node after having cleared its contained rb
```

```
int main()
{
	//malloc on the first node of the ll containing the ll priority queues
	//everything is set to NULL to be set on the first create_q() function call
	LLarray * llheadnode = (LLarray *) malloc(sizeof(LLarray));
	llheadnode->ll = NULL;
	llheadnode->name = NULL;
	llheadnode->next = NULL;

	//malloc on the first node of the ll containing the rb priority queues
	//everything is set to NULL to be set on the first create_q() function call
	RBarray * rbheadnode = (RBarray *) malloc(sizeof(RBarray));
	rbheadnode->rb = NULL;
	rbheadnode->name = NULL;
	rbheadnode->next = NULL;

	char ch;//to be used to switch statements

	printf("\nWelcome to the queueing application");

	BEGINNING://used heavily to return to "main" menu
	printf("\n\na. Linked List\nb. Ring Buffer\n\n");
	printf("Please make a selection (a or b or enter 'q' to quit):\n\n");


	scanf("%c%*c", &ch);//%*c is used to exclude newline
	{

		if('q' == ch)
			return 0;

		switch(ch)
		{
			case 'a':
				goto CASE1;//ll menu

			case 'b':
				goto CASE2;//rb menu

			default:
				printf("\nThat is not a valid choice\n");
				goto BEGINNING;
		}
	}

	CASE1:
	{
		char * queuename;
		char ch2 = NULL;
```

```
		printf("\n\nActions:\n\n");
		printf("a. Create a Linked List Queue\nb. Make an entry into a queue\nc. Dequeue highest priority element\nd. Highest priority element\ne. Check whether queue is empty\nf. Size of queue\ng. Merge two queue's\nh. Clear queue of contents\ni. List existing queues\nj. Store existing queue to file\nk. Load queue from file\n");
		printf("Please make a selection (enter 'q' to return to main menu):\n\n");

		scanf("%c%*c", &ch2);
		{

			if('q' == ch2)
				goto BEGINNING;

			switch(ch2)
			{
				case 'a':
				{//create a new ll
					printf("\nName of Linked List:\n");
					queuename = getstring();

					printf("\nMaximum size of Linked List:\n");
					int max_size;
					scanf("%d%*c", &max_size);

					appendllqueue(create_q(max_size), queuename, llheadnode);

					goto BEGINNING;
				}

				case 'b':
				{
					printf("\nEnter the queue name:\n");
					queuename = getstring();

					printf("\nEnter an integer element:\n");
					int element;
					scanf("%d%*c", &element);

					printf("\nEnter an integer priority:\n");
					int priority;
					scanf("%d%*c", &priority);

					if(accessllqueue(queuename, llheadnode) == NULL)//accessllqueue returns a NULL if it does not find specified name
						printf("\nThis queue does not exist\n");
					else
						enqueue(accessllqueue(queuename, llheadnode)->ll, element, priority);
```

```
                goto BEGINNING;
            }

            case 'c':
            {
                printf("\nName of Linked List:\n");
                queuename = getstring();

                if(accessllqueue(queuename, llheadnode) == NULL)
                    printf("\nThis queue does not exist\n");
                else
                    dequeue(accessllqueue(queuename, llheadnode)-
>ll);

                goto BEGINNING;
            }

            case 'd':
            {
                printf("\nName of Linked List:\n");
                queuename = getstring();

                if(accessllqueue(queuename, llheadnode) == NULL)
                    printf("\nThis queue does not exist\n");
                else
                    printf("\nHighest priority element is: %d\nwith a
priority of: %d", peek(accessllqueue(queuename, llheadnode)->ll)->element,
peek(accessllqueue(queuename, llheadnode)->ll)->priority);

                goto BEGINNING;
            }

            case 'e':
            {
                printf("\nName of Linked List:\n");
                queuename = getstring();

                if(accessllqueue(queuename, llheadnode) == NULL)
                    printf("\nThis queue does not exist\n");
                else
                {
                    if(is_empty(accessllqueue(queuename,
llheadnode)->ll))
                        printf("Queue is empty\n");
                    else
                        printf("Queue is not empty\n");
                }

                goto BEGINNING;
            }
```

```
                        case 'f':
                        {
                                printf("\nName of Linked List:\n");
                                queuename = getstring();

                                if(accessllqueue(queuename, llheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                        printf("Size of queue is
%d\n",size(accessllqueue(queuename, llheadnode)->ll));

                                goto BEGINNING;
                        }

                        case 'g':
                        {
                                printf("\nName of Linked List 1:\n");
                                char queuename1[20];
                                strncpy(queuename1, getstring(),20);


                                printf("\nName of Linked List 2:\n");
                                char queuename2[20];
                                strncpy(queuename2, getstring(),20);

                                if((accessllqueue(queuename1, llheadnode) == NULL) ||
(accessllqueue(queuename2, llheadnode) == NULL))
                                        printf("\nSelection does not exist\n");
                                else
                                {//also deletes the two input queues (queuename1 and
queuename2)
                                        printf("\nName of new Linked List:\n");
                                        char queuename3[20];
                                        strncpy(queuename3, getstring(),20);


        appendllqueue(merge(accessllqueue(queuename1, llheadnode)->ll,
accessllqueue(queuename2, llheadnode)->ll), queuename3, llheadnode);
                                        clear(accessllqueue(queuename1, llheadnode)-
>ll);
                                        llheadnode = clearllqueue(queuename1,
llheadnode);

                                        clear(accessllqueue(queuename2, llheadnode)-
>ll);
                                        llheadnode = clearllqueue(queuename2,
llheadnode);

                                }
```

```
                goto BEGINNING;
        }

        case 'h':
        {
                printf("\nName of Linked List:\n");
                queuename = getstring();

                if(accessllqueue(queuename, llheadnode) == NULL)
                        printf("\nThis queue does not exist\n");
                else
                {
                        clear(accessllqueue(queuename, llheadnode)->ll);
                        llheadnode = clearllqueue(queuename,
llheadnode);
                        //this function returns headnode in  case it was
deleted and changed
                }

                goto BEGINNING;
        }

        case 'i':
        {
                printqueuesll(llheadnode);

                goto BEGINNING;
        }

        case 'j':
        {
                printf("\nName of Linked List:\n");
                queuename = getstring();

                if(accessllqueue(queuename, llheadnode) == NULL)
                        printf("\nThis queue does not exist\n");
                else
                        store(accessllqueue(queuename, llheadnode)->ll,
queuename);

                goto BEGINNING;
        }

        case 'k':
        {
                printf("\nName of Linked List:\n");
                queuename = getstring();

                appendllqueue(load(queuename), queuename,
```

```
llheadnode);

                                        goto BEGINNING;
                                }

                                default:
                                {
                                        printf("\nThat is not a valid choice\n");

                                        goto BEGINNING;
                                }
                        }
                }
        }

        CASE2:
        {
                char * queuename;
                char ch2 = NULL;

                printf("\n\nActions:\n\n");
                printf("a. Create a RingBuffer Queue\nb. Make an entry into a queue\nc.
Dequeue highest priority element\nd. Highest priority element\ne. Check whether queue is
empty\nf. Size of queue\ng. Merge two queue's\nh. Clear queue of contents\ni. List
existing queues\nj. Store existing queue to file\nk. Load queue from file\n");
                printf("Please make a selection (enter 'q' to return to main menu):\n\n");

                scanf("%c%*c", &ch2);
                {
                        if('q' == ch2)
                                goto BEGINNING;

                        switch(ch2)
                        {
                                case 'a':
                                {
                                        printf("\nName of Ring Buffer:\n");
                                        queuename = getstring();

                                        printf("\nMaximum size of Ring Buffer:\n");
                                        int max_size;
                                        scanf("%d%*c", &max_size);

                                        appendrbqueue(create_qR(max_size), queuename,
rbheadnode);

                                        goto BEGINNING;
                                }

                                case 'b':
```

```
                        {
                                printf("\nEnter the queue name:\n");
                                queuename = getstring();

                                printf("\nEnter an integer element:\n");
                                int element;
                                scanf("%d%*c", &element);

                                printf("\nEnter an integer priority:\n");
                                int priority;
                                scanf("%d%*c", &priority);

                                if(accessrbqueue(queuename, rbheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                        enqueueR(accessrbqueue(queuename,
rbheadnode)->rb, element, priority);

                                goto BEGINNING;
                        }

                        case 'c':
                        {
                                printf("\nName of Ring Buffer:\n");
                                queuename = getstring();

                                if(accessrbqueue(queuename, rbheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                        dequeueR(accessrbqueue(queuename,
rbheadnode)->rb);

                                goto BEGINNING;
                        }

                        case 'd':
                        {
                                printf("\nName of Ring Buffer:\n");

                                queuename = getstring();
                                if(accessrbqueue(queuename, rbheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                        printf("\nHighest priority element is: %d\nwith a
priority of: %d", peekR(accessrbqueue(queuename, rbheadnode)->rb).elem,
peekR(accessrbqueue(queuename, rbheadnode)->rb).priority);

                                goto BEGINNING;
                        }
```

```
                        case 'e':
                        {
                                printf("\nName of Ring Buffer:\n");
                                queuename = getstring();

                                if(accessrbqueue(queuename, rbheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                {
                                        if(is_emptyR(accessrbqueue(queuename,
rbheadnode)->rb))
                                                printf("Queue is empty\n");
                                        else
                                                printf("Queue is not empty\n");
                                }

                                goto BEGINNING;
                        }

                        case 'f':
                        {
                                printf("\nName of Ring Buffer:\n");
                                queuename = getstring();

                                if(accessrbqueue(queuename, rbheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                        printf("Size of queue is
%d\n",sizeR(accessrbqueue(queuename, rbheadnode)->rb));

                                goto BEGINNING;
                        }

                        case 'g':
                        {
                                //Takes 3 string inputs, deletes 2 input queues, creates
one merged queues
                                printf("\nName of Ring Buffer 1:\n");
                                char queuename1[20];
                                strncpy(queuename1, getstring(), 20);


                                printf("\nName of Ring Buffer 2:\n");
                                char queuename2[20];
                                strncpy(queuename2, getstring(), 20);

                                if(accessrbqueue(queuename1, rbheadnode) == NULL ||
accessrbqueue(queuename2, rbheadnode) == NULL)
                                        printf("\nSelection does not exist\n");
                                else
```

```
                                {
                                        printf("\nName of new Ring Buffer:\n");
                                        char queuename3[20];
                                        strncpy(queuename3, getstring(), 20);


        appendrbqueue(mergeR(accessrbqueue(queuename1, rbheadnode)->rb,
accessrbqueue(queuename2, rbheadnode)->rb), queuename3, rbheadnode);
                                        //clearing input queues
                                        clearR(accessrbqueue(queuename1,
rbheadnode)->rb);
                                        rbheadnode = clearrbqueue(queuename1,
rbheadnode);

                                        clearR(accessrbqueue(queuename2,
rbheadnode)->rb);
                                        rbheadnode = clearrbqueue(queuename2,
rbheadnode);
                                }

                                goto BEGINNING;
                        }

                        case 'h':
                        {
                                printf("\nName of Ring Buffer:\n");
                                queuename = getstring();

                                if(accessrbqueue(queuename, rbheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                {
                                        clearR(accessrbqueue(queuename, rbheadnode)-
>rb);
                                        rbheadnode = clearrbqueue(queuename,
rbheadnode);
                                }

                                goto BEGINNING;
                        }

                        case 'i':
                        {
                                printqueuesrb(rbheadnode);
                                goto BEGINNING;
                        }

                        case 'j':
                        {
                                printf("\nName of Ring Buffer:\n");
```

```
                                queuename = getstring();

                                if(accessrbqueue(queuename, rbheadnode) == NULL)
                                        printf("\nThis queue does not exist\n");
                                else
                                        storeR(accessrbqueue(queuename, rbheadnode)-
>rb, queuename);

                                goto BEGINNING;
                        }

                        case 'k':
                        {
                                printf("\nName of Ring Buffer:\n");
                                queuename = getstring();

                                appendrbqueue(loadR(queuename), queuename,
rbheadnode);

                                goto BEGINNING;

                        }

                        default:
                        {
                                printf("\nThat is not a valid choice\n");

                                goto BEGINNING;
                        }
                }
        }
        }
}


////////////////////////////////////////////////////////////////////////////

void printqueuesll(LLarray * llheadnode)
{       //will print existing ll queues to console
        LLarray * current = llheadnode;// Queue array struct
        int i = 1; //Queue number

        while(current->name != NULL) //Iterate through queue struct array to print  one by
one
        {
                printf("Queue:%d %s with %d elements \n", i, current->name, size(current-
>ll));
                current = current->next;
                i++;
        }
```

```
}
void printqueuesrb(RBarray * rbheadnode)
{	//will print existing rb queues to console
	RBarray * current = rbheadnode;// Queue array struct
	int i = 1; //Queue number

	while(current->name != NULL) //Iterate through queue struct array to print  one by
one
	{
		printf("Queue:%d %s with %d elements \n", i, current->name, sizeR(current-
>rb));
		current = current->next;
		i++;
	}
}

char * getstring()
{	//a function for user string input
	char * string = malloc(sizeof(char)*20);

	scanf("%s%*c", string);

	return string;
}

////////////////////////////////////////////////////////////////////////////

void appendllqueue(linkedlists * linkedlist, char * queuename, LLarray * llheadnode)
{	//adding an element to the ll for ll priority queues
	if(accessllqueue(queuename, llheadnode) == NULL)//if name not found in list of
queues
	{
		LLarray * current  = llheadnode;

		while(current->next != NULL)
			current = current->next;

		current->name = (char *)calloc(strlen(queuename)+1, sizeof(char));
		strncpy(current->name, queuename, strlen(queuename));

		current->ll = linkedlist;

		printf("\nYour queue name is: %s", current->name);

		current->next = (LLarray*) malloc(sizeof(LLarray));
		current->next->ll = NULL;
		current->next->name = NULL;
		current->next->next = NULL;

	}
```

```
	else
		printf("\nThis name already exists");


}


LLarray * accessllqueue(char * queuename, LLarray * llheadnode)
{	//accessing a particular node of the ll for the ll priority queues
	LLarray * current = llheadnode;//starting from first node

	while(current->next != NULL && strcmp(current->name, queuename) != 0)
	//if next node is unallocated and current node's name does not match user input
		current = current->next;

	if(current->name == NULL && current->next == NULL)
	//if this is true then whole list was iterated through with no results
		return NULL;
	else
	//this is true if result is found, current node is returned (list of linked lists)
		return current;
}

LLarray * clearllqueue(char * queuename, LLarray * llheadnode)
{	//clearing a particular node after having cleared its contained ll
	LLarray * previous;//used to fix previous node's pointer
	LLarray * current = llheadnode;

	while(strcmp(current->name, queuename) != 0)//iterates until match
	{
		previous = current;
		current = current->next;
	}

	if(current == llheadnode)//if current happens to be headnode
	{
		llheadnode = current->next;
		if(llheadnode->next == NULL)//if headnode is not followed by additional
nodes, allocates new memory
		{
			llheadnode->next = (LLarray *)malloc(sizeof(LLarray));
			llheadnode->next->name = NULL;
			llheadnode->next->ll = NULL;
			llheadnode->next->next = NULL;
		}

		free(current->name); // ll already freed and next should not be freed
		free(current); // free what was previously headnode
		printf("\n%s cleared\n", queuename);
	}
```

```
	else//if headnode is not node to be cleared
	{
		previous->next = current->next;
		free(current->name);
		free(current);
		printf("\n%s cleared\n", queuename);


	}


	return llheadnode;
}
////////////////////////////////////////////////////////////////////////////


void appendrbqueue(RingBuffer * RBqueue, char * queuename, RBarray * rbheadnode)
{	//adding an element to the ll for ll priority queues
	if(accessrbqueue(queuename, rbheadnode) == NULL)//if name not found in list of
queues
	{
		RBarray * current  = rbheadnode;

		while(current->next != NULL)
			current = current->next;

		current->name = (char *)calloc(strlen(queuename)+1, sizeof(char));
		strncpy(current->name, queuename, strlen(queuename));

		current->rb = RBqueue;

		printf("\nYour queue name is: %s", current->name);

		current->next = (RBarray*) malloc(sizeof(LLarray));
		current->next->rb = NULL;
		current->next->name = NULL;
		current->next->next = NULL;

	}
	else
		printf("\nThis name already exists");

}


RBarray * accessrbqueue(char * queuename, RBarray * rbheadnode)
{	//accessing a particular node of the ll for the rb priority queues
	RBarray * current = rbheadnode;//starting from first node

	while(current->next != NULL && strcmp(current->name, queuename) != 0)
```

```
	//if next node is unallocated and current node's name does not match user input
		current = current->next;

	if(current->name == NULL && current->next == NULL)
	//if this is true then whole list was iterated through with no results
		return NULL;
	else
	//this is true if result is found, current node is returned (list of linked lists)
		return current;
}

RBarray * clearrbqueue(char * queuename, RBarray * rbheadnode)
{	//clearing a particular node after having cleared its contained rb
	RBarray * previous;
	RBarray * current = rbheadnode;


	while(strcmp(current->name, queuename) != 0)
	{
		previous = current;
		current = current->next;
	}

	if(current == rbheadnode)
	{
		rbheadnode = current->next;
		if(rbheadnode->next == NULL)
		{
			rbheadnode->next = (RBarray *)malloc(sizeof(RBarray));
			rbheadnode->next->name = NULL;
			rbheadnode->next->rb = NULL;
			rbheadnode->next->next = NULL;
		}

		free(current->name); // ll already freed and next should not be freed
		free(current); // free struct
		printf("\n%s cleared\n", queuename);
	}
	else
	{
		previous->next = current->next;
		free(current->name);
		free(current);
		printf("\n%s cleared\n", queuename);


	}
	return rbheadnode;
}
```

3.2 Appendix 2: LINKEDLIST.c

```
#include <stdbool.h>
#include <string.h> /*has the strcpy prototype*/
#include <stdlib.h> /* has the malloc prototype */
#include <stdio.h>
#include "LINKEDLIST.h"



linkedlists * create_q(int max_size)
{	//first malloc the Linkedlist (linkedlists contains all nodes and one int for max, but for now it is empty except for max int)
	linkedlists * linkedlist = (linkedlists *)malloc(sizeof(linkedlists));
	linkedlist->max_size = max_size;
	linkedlist->headnode = NULL;//headnode set to NULL until next

	return linkedlist;
}


void enqueue(linkedlists * linkedlist, int x, int p)
{


	if(linkedlist->headnode == NULL)//if linkedlist contains no nodes
	{	//malloc and assign first node
		linkedlist->headnode = (pqnode *) malloc(sizeof(pqnode));

		linkedlist->headnode->element = x;
		linkedlist->headnode->priority = p;
		linkedlist->headnode->next = NULL;
	}
	else
	{
		pqnode * current = linkedlist->headnode;//starting from first node to iterate

		int max_size = linkedlist->max_size;
		int i = 2;//counter

		while(current->next != NULL)
		{
			current = current->next;
			i++;
		}


		if(i <= max_size)
		{//for example if max size is 2 and only one node exists then the while loop
```

```
above does not execute and i is left at 2
                        current->next = (pqnode *) malloc(sizeof(pqnode));
                        current = current->next;

                        current->element = x;
                        current->priority = p;
                        current->next = NULL;
                }
                else
                        printf("\nLinked list is full with %d elements.\n", i-1);
        }

}

void dequeue(linkedlists * linkedlist)
{
/*FIFO*/
        pqnode * current;
        pqnode * prev;//prev required to prevent connect previous to next

        current = linkedlist->headnode;
        int temppriority = 0;

        while(current != NULL)
        {//iterate and record highest priority
                if(current->priority > temppriority)
                        temppriority = current->priority;

                current = current->next;
        }

        current = linkedlist->headnode;

        while(current->priority != temppriority && current != NULL)
        {//iterate until largest priority is found
                prev = current;
                current=current->next;
        }

        if(temppriority == current->priority)
        {
                prev->next = current->next;
                free(current);
        }
}


pqnode * peek(linkedlists * linkedlist)
{
        pqnode * current;
```

```
	current = linkedlist->headnode;
	int temppriority = 0;

	while(current != NULL)
	{//iterate and record highest priority
		if(current->priority > temppriority)
			temppriority = current->priority;

		current = current->next;
	}

	current = linkedlist->headnode;

	while(temppriority != current->priority && current != NULL)
	//iterate until highest priority is found
		current=current->next;


	if(temppriority == current->priority)
	{
		return current;
	}
	else
		return NULL;
}


bool is_empty(linkedlists * linkedlist)
{
	pqnode * current;
	current = linkedlist->headnode;


	if(current == NULL)
		return true;
	else
		return false;
}


int size(linkedlists * linkedlist)
{
	pqnode * current = linkedlist->headnode;
	int count = 0;

	while(current != NULL)
	{//iterate and count
		count++;
		current=current->next;
```

```
	}
	return count;
}


linkedlists * merge(linkedlists *linkedlistQ, linkedlists* linkedlistR)
{	//new queue has combined max size.
	linkedlistQ->max_size = linkedlistQ->max_size + linkedlistR->max_size;

	pqnode * current = linkedlistQ->headnode;

	while(current->next != NULL)
		current = current->next;

	current->next = linkedlistR->headnode;
	free(linkedlistR);

	return linkedlistQ;
}


void clear(linkedlists * linkedlist)
{
	pqnode * current = linkedlist->headnode;
	pqnode * prev;

	while(current != NULL)
	{//iterate and clear
		prev = current;
		current = current->next;
		free(prev);
	}
}


void store(linkedlists * linkedlist, char * filename)
{	//append ll.bin to distinguish from ringbuffer and to save as binary filie
	char filetype[6] = "ll.bin";
	char filename2[20] = "\0";
	strncpy(filename2, filename, strlen(filename));
	strncat(filename2,filetype,6);
	printf("%s", filename2);

	FILE * pFile = fopen(filename2, "wb");


	if (pFile != NULL)
	{	//first fwrite the linkedlist struct, then the maxsize, then each individual node
		pqnode * Current = linkedlist->headnode;
		pqnode * holdnext = NULL;
```

```
                linkedlist->headnode = NULL;

                fseek(pFile, 0, SEEK_END);
                fwrite(linkedlist, sizeof(linkedlists), 1, pFile);

                linkedlist->headnode = Current;

                int numnodes = size(linkedlist);
                fseek(pFile, 0, SEEK_END);
                fwrite(&numnodes, sizeof(int), 1, pFile);



                while(Current != NULL)
                {
                        holdnext = Current->next;
                        Current->next = NULL;

                        fseek(pFile, 0, SEEK_END);
                        fwrite(Current, sizeof(pqnode), 1, pFile);


                        Current->next = holdnext;

                        Current = Current->next;
                }
                printf("\nWrite Successful\n");
                fclose(pFile);
        }
        else
                printf("\nError Writing\n");
}



linkedlists * load(char * filename)
{       //same process as store but using also using malloc

        char filetype[6] = "ll.bin";
        char filename2[20] = "\0";
        strncpy(filename2, filename, strlen(filename));
        strncat(filename2,filetype,6);
        printf("%s", filename2);

        FILE * pFile = fopen(filename2, "rb");
        linkedlists * linkedlist;


        if(pFile != NULL)
```

```
	{
		linkedlist = (linkedlists *) malloc(sizeof(linkedlists));
		fseek(pFile, 0, SEEK_SET);
		fread(linkedlist, sizeof(linkedlist), 1, pFile);

		int numnodes;
		fseek(pFile, sizeof(linkedlists), SEEK_SET);
		fread(&numnodes, sizeof(int), 1, pFile);

		if(numnodes > 0)
		{
			pqnode * current = NULL;

			linkedlist->headnode = (pqnode *) malloc(sizeof(pqnode));
			current = linkedlist->headnode;

			fseek(pFile, sizeof(linkedlists)+sizeof(int), SEEK_SET);
			fread(current, sizeof(pqnode), 1, pFile);
			current->next = NULL;

			int i;
			for(i = 1; i < numnodes; i++)
			{
				current->next = (pqnode *) malloc(sizeof(pqnode));
				current = current->next;

				fread(current, sizeof(pqnode), 1, pFile);

				current->next = NULL;
			}
		}
		printf("\nRead successful\n");
		fclose(pFile);
	}
	else
	{
		printf("\nError with read\n");
		return NULL;
	}

	return linkedlist;
}
```

3.3 Appendix 3: LINKEDLIST.h

```
#ifndef _LINKEDLIST_PRIORITYQUEUE_h
#define _LINKEDLIST_PRIORITYQUEUE_h
#include <stdbool.h>

//single node for the linked list priority queue
typedef struct pqnode
{
	int element;
	int priority;
	struct pqnode * next;
}pqnode;

//struct that contains all nodes
typedef struct linkedlists
{
	int max_size;
	struct pqnode * headnode;
}linkedlists;


linkedlists * create_q(int max_size);

void enqueue(linkedlists * linkedlist, int x, int p);

void dequeue(linkedlists * linkedlist);

pqnode * peek(linkedlists * linkedlist);

bool is_empty(linkedlists * linkedlist);

int size(linkedlists * linkedlist);

linkedlists * merge(linkedlists *linkedlistQ, linkedlists* linkedlistR);

void clear(linkedlists * linkedlist);

void store(linkedlists * linkedlist, char * filename);

linkedlists * load(char * filename);

#endif
```

3.4 Appendix 4: RINGBUFFER.c

```
#include <stdbool.h>
#include <string.h> /*has the strcpy prototype*/
#include <stdlib.h> /* has the malloc prototype */
#include <stdio.h>
#include "RINGBUFFER.h"



RingBuffer * create_qR(int max_size)
{
	//using calloc to create an array of structs
	RingBuffer * ringbuffer = calloc(1, sizeof(RingBuffer));
	ringbuffer->max_size = max_size+1;//add one because one of the elements is not
used
	ringbuffer->start = 0;
	ringbuffer->end = 0;
	ringbuffer->elems = calloc(ringbuffer->max_size, sizeof(Element));

	return ringbuffer;

}


void enqueueR(RingBuffer * rb, int x, int p)
{
	rb->elems[rb->end].elem = x;
	rb->elems[rb->end].priority = p;
	rb->end = (rb->end + 1) % rb->max_size;//iterate end indicator by one
	if (rb->end == rb->start)
		rb->start = (rb->start + 1) % rb->max_size;//iterate start indicator by one
(modulus to stop from exceeding max size)

	sortR(rb);

}

void sortR(RingBuffer * rb)
{	//create temporary copy to sort
	Element * temp = (Element *) malloc(sizeof(Element));
	Element * rbnew = calloc(rb->max_size, sizeof(Element));
	Element empty = {.elem = 0, .priority = 0};
	int indexmemory;
	int i;
	int j;

	for(i = 0; i < sizeR(rb); i++)
	{
```

```
                *temp = rb->elems[rb->start];
                indexmemory = rb->start;
                for(j = 1; j < sizeR(rb); j++)
                {
                        if(rb->elems[(rb->start + j) % rb->max_size].priority > temp->priority)
                        {
                                *temp = rb->elems[(rb->start + j) % rb->max_size];
                                indexmemory = (rb->start + j) % rb->max_size;
                        }
                }

                rb->elems[indexmemory] = empty;

                rbnew[(rb->start+i) % rb->max_size] = *temp;
        }

        memcpy(rb->elems, rbnew, sizeof(Element)*rb->max_size);

        free(rbnew);
}

int sizeR(RingBuffer * rb)
{
        int count = 0;
        int i = 0;
        while((rb->start+i) %rb->max_size != rb->end)
        {
                i++;
                count++;
        }

        return count;
}


void dequeueR(RingBuffer * rb)
{       //iterating start by one to change start location
        if(rb->start % rb->max_size != rb->end % rb->max_size)
                rb->start++;
        else
                printf("\nQueue is empty\n");
}


Element peekR(RingBuffer * rb)
{
        return rb->elems[rb->start];
}
```

```
bool is_emptyR(RingBuffer * rb)
{
	if(rb->start == rb->end)
		return true;
	else
		return false;
}




RingBuffer * mergeR(RingBuffer * rb1, RingBuffer * rb2)
{
	RingBuffer * rbnew = calloc(1, sizeof(RingBuffer));
	rbnew->max_size = rb1->max_size + rb2->max_size - 1; //adding two queues
together leaves two unused elements, one is removed
	rbnew->start = 0;
	rbnew->end = 0;
	rbnew->elems = calloc(rbnew->max_size, sizeof(Element));

	int i;
	for(i = 0; i < sizeR(rb1); i++)//first write to new queue from queue1
	{
		rbnew->elems[i] = rb1->elems[(rb1->start + i) % rb1->max_size];
		rbnew->end = (rbnew->end + 1) % rbnew->max_size;
	}

	for(i = 0; i < sizeR(rb2); i++)//write to new queue from queue2
	{
		rbnew->elems[(sizeR(rb1) + i) % rb2->max_size] = rb2->elems[(rb2->start +
i) % rb2->max_size];
		rbnew->end = (rbnew->end + 1) % rbnew->max_size;
	}

	sortR(rbnew);//new queue is sorted


	return rbnew;
}


void clearR(RingBuffer * rb)
{
	rb->start = 0;
	rb->end = 0;
}


void storeR(RingBuffer * rb, char * filename)
```

```
{       //frwite as a whole due to no pointers present
        char filetype[6] = "rb.bin";
        char filename2[20] = "\0";
        strncpy(filename2, filename, strlen(filename));
        strncat(filename2,filetype,6);
        printf("%s", filename2);

        FILE * pFile = fopen(filename2, "wb");


        if (pFile != NULL)
        {

                fseek(pFile, 0, SEEK_END);
                fwrite(rb, sizeof(RingBuffer), 1, pFile);

                printf("\nWrite Successful\n");
                fclose(pFile);
        }
        else
                printf("Error Writing/n");
}

RingBuffer * loadR(char * filename)
{       //same process as store but using also using calloc

        char filetype[6] = "rb.bin";
        char filename2[20] = "\0";
        strncpy(filename2, filename, strlen(filename));
        strncat(filename2,filetype,6);
        printf("%s", filename2);

        FILE * pFile = fopen(filename2, "rb");
        RingBuffer * rb;


        if(pFile != NULL)
        {
                rb = calloc(1, sizeof(RingBuffer));
                fseek(pFile, 0, SEEK_SET);
                fread(rb, sizeof(RingBuffer), 1, pFile);

                printf("\nRead successful\n");
                fclose(pFile);
        }
        else
                printf("Error with read");

        return rb;
}
```

3.5 Appendix 5: RINGBUFFER.h

```
#ifndef _RINGBUFFER_PRIORITYQUEUE_h
#define _RINGBUFFER_PRIORITYQUEUE_h

#include <stdbool.h>

typedef struct//element struct
{
	int elem;
	int priority;
}
Element;

typedef struct//struct that contains elements
{
	int max_size;
	int start;
	int end;
	Element * elems;
}
RingBuffer;

void sortR(RingBuffer * rb);
int sizeR(RingBuffer * rb);

RingBuffer * create_qR(int max_size);

void enqueueR(RingBuffer * rb, int x, int p);

void sortR(RingBuffer * rb);

int sizeR(RingBuffer * rb);

void dequeueR(RingBuffer * rb);

Element peekR(RingBuffer * rb);

bool is_emptyR(RingBuffer * rb);

RingBuffer * mergeR(RingBuffer * rb1, RingBuffer * rb2);

void clearR(RingBuffer * rb);

void storeR(RingBuffer * rb, char * filename);

RingBuffer * loadR(char filename[]);

#endif
```

4.0 References

[1] *Data Structures/Arrays*. Available: http://en.wikibooks.org/wiki/Data_Structures/Arrays.

[2] (10 February 2007). *Circular Buffer*. Available: http://c2.com/cgi/wiki?CircularBuffer.

[3] *Introduction to linked list: C Programming*. Available: http://www.c4learn.com/data-structure/introduction-to-linked-list-c-programming/.

[4] (2009). *Linked Lists*. Available: http://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html.

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

[6] *Arrays in Java*. Available: http://ycpcs.github.io/cs201-summer2014/notes/javaArrays.html.

[7] *Linked List Basics*. Available: http://cslibrary.stanford.edu/103/LinkedListBasics.pdf.