**Problem 1 follows…**  
**Description:** Problem 1 is designed to demonstrate my knowledge of queues. It consists of function descriptions for initialization, dequeue, enqueue, peek, priority and order of content in a queue.

**Answers to Questions:**

1. What is a Queue?

A queue is a linear data structure that has items placed and removed inside it, such that the most recently added item will be the last to be removed. This is called a first in-first out (FIFO) data structure. A FIFO means that the first item (not necessarily most recent) placed inside gets removed first. A queue can add or remove data, but the difference between a FIFO and a LIFO is in which data gets removed when an enque/pop is called.

2. Provide an example of a Queue, name the Queue basic operations and what they are responsible for in your example. Note that you cannot use an example in the text or overview and that you must describe the Queue operations in the context of the example you site.

Anything which is a line of objects of the same type that is first come first serve, is a queue. I will use the example of people buying movie theatre tickets at a ticket booth. The abilities of the queue are to be able to initialize a queue. If you own a movie theatre, initializing the queue is like clearing space for a line of people at your ticket booth outside of the movie theatre so you have space for as many people in line. Also, the queue could help you identify if it is empty. If you exit the ticket booth and see no people in line, then the queue is empty. If you exit the ticket booth and see a single person or more, then the queue is not empty. This helps identify whether you could take a person out of the queue, i.e. call a dequeue. You could also get the number of people in the queue to give you an idea of how many dequeues you can call before the stack is empty. A dequeue requires, in the case of a FIFO queue, to remove the first (not necessarily most recently) placed person and remove it from the queue. Imagine taking the first person in line out of the line because they received a movie ticket. The first person at the front of the line is the first person that is given movie tickets at movie time. A queue also lets you check the highest priority item or the next item, without removing it, called a peek. If you wanted to see if the person at the front end of the line exists, you might call a peek. This function of a queue can tell you if it is full or not, to identify whether another enqueue could occur. An enqueue will update the tail with another person. So, when you enqueue a new person, another person will be added to the tail of the line of people at your ticket booth.

3. What is a Priority Queue?

A priority queue is a queue abstract data type that has a relative (to other entries) priority among its entries. Where a typical queue abstract data type treats content in an enqueued ordered manner, a priority queue takes the first oldest with the highest priority to dequeue or peek. This means when an enqueue occurs, the queue needs to know the priority of content if multiple queues are held, so to place the enqueued content in the queue at the right spot. A priority queue still makes the just enqueued data the last in line, but deques using priority and order.

4. Provide an example of a Priority Queue, naming the Queue basic operations and what they are responsible for in the example. Note that you cannot use an example in the text or overview and that you must describe the Queue operations in the context of the example you site.

A priority queue example is when servers are in line for network access. When a new server joins the network, an enqueue is called. When an old server leaves the network, a dequeue is called. A peek will show the server at the tail end of the line. Each server has information regarding its order, based on when it joins the network. The first (not necessarily most recent) server to join has the highest priority because it became the head of the line before all other servers. This means this original server gets access to the network first. But as more and more servers join the network, network traffic becomes complicated because each server requires a different amount of network access. So, the priority queue (after the first enqueue) will assign a priority along with order to each newly joined server. New servers are required to carry this information so they may dequeue ahead or behind other servers in the queue. If one server joins after every other server already in the queue but requires more network access, it might dequeue before all other servers. This is so that server (with the highest priority but last in order) can get its network access in a fast and reliable manner. Other servers that require less network access can wait longer without impacting their function.

5. Since a Priority Queue needs to keep information as a prioritized FIFO, it needs to be able to always have the highest priority item as the next item that could be "peek"ed or dequeued. As a designer, what could you do to support this? What might change in the Priority Queue supported functions? The initQueue? The enqueue?

Every item in the queue needs information that reflects its relative priority to other entries. This could become a pointer that gets saved in a function to compare priorities of queue objects. This function could compare queued data so to reorder the queue and maintain the priority of queue content. This would fire every time an enqueue is done to place the enqueued data in the prioritized order. A peek could also be made to return the entry’s priority and order. initQueue might not need priority yet, as there is no order to queue objects. But as soon as the queue has more than one entry, each entry must be given priority information alongside its order information so to maintain an order of a priority queue.

**Problem 2 follows…**  
**Description:** Problem 2 is designed to demonstrate the queue abstract data type implementation. It consists of functions and definitions for the queue including initializing, checking if the queue is empty, peek, enqueue/dequeue and deleting the queue. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code (copies of code for functions deleteQueue, enqueue and dequeue in Queue.c):**

/\*

deleteQueue() calls the linked list delete to free up all of its nodes and, on return,

frees up the queue itself. it returns NULL to indicate that there is no longer a

queue.

\*/

Queue deleteQueue(Queue Q)

{

//check if queue is not NULL

assert (Q != NULL);

//free up all linked list nodes and delete linked list

LL\_Delete(Q->LL);

//free up the queue

free (Q);

//decrease AllocationCount to reflect freed up queue

AllocationCount--;

//return NULL to reflect the queue no longer exists

return NULL;

}

/\* enqueue() calls the linked list to place the UserData at the end of

the linked list. Since an enqueue is being done, the queue is no longer empty

\*/

void enqueue (Queue Q, UserData D)

{

//check if queue is not NULL

assert (Q != NULL);

//place UserData at end of the linked list

LL\_AddAtEnd(Q->LL, D);

//set queue to not empty

Q->empty = false;

}

/\*

dequeue() will fetch the UserData at the front of the linked list and return it to

caller. It updates the queue empty status by seeing if the linked list was

holding only a single item before the removal from the linked list occurs.

\*/

UserData dequeue (Queue Q)

{

//check if queue is not NULL

assert (Q!= NULL);

//set queue's empty boolean based on if queue had 1 item before removal or not

Q->empty = LL\_Length(Q->LL) == 1 ? true : false;

//return UserData at front of the linked list and delete it

return LL\_GetFront(Q->LL, DELETE\_NODE);

}

**Program Output:**

On startup, #allocations is 0

After initQueue called, #allocations is 2

enqueue called, data is 1000, #allocations is 3

enqueue called, data is 2000, #allocations is 4

enqueue called, data is 3000, #allocations is 5

enqueue called, data is 4000, #allocations is 6

peek called, data is 1000, #allocations is 6

dequeue called, data is 1000, #allocations is 5

peek called, data is 2000, #allocations is 5

dequeue called, data is 2000, #allocations is 4

peek called, data is 3000, #allocations is 4

dequeue called, data is 3000, #allocations is 3

peek called, data is 4000, #allocations is 3

dequeue called, data is 4000, #allocations is 2

Before deleteQueue called, #allocations is 2

After deleteQueue called, #allocations is 0

**Answers to Questions:**

4.

a. After the initQueue has been called, the AllocationCount has gone from 0 to 2. What are the 2 allocations that occurred?

After initQueue has been called, AllocationCount has gone from 0 to 2 to reflect the number of things that are currently dynamically allocated. This includes a linked list and a queue. First, AllocationCount equals zero. Next, initQueue first a allocates a queue, then increases AllocationCount by 1 to equal 1. Then initQueue calls LL\_init() which allocates a linked list, then increases AllocationCount by 1 to equal 2.

b. What did the calls to enque do? How did they use the linked list functionality? Why is the AllocationCount increasing? Explain if this makes sense.

The calls to enqueue check that the queue is not equal to NULL. Then it calls LL\_AddAtEnd to allocate a node at the end of the linked list for UserData to be placed in. Creating a node increases AllocationCount by 1. Then the queue is set to not empty because it contains a newly created data. If there is no data in the linked list, a call to LL\_AddAtEnd will actually just call LL\_AddAtFront so that the new data becomes both the head and the tail of the linked list. Otherwise, the new data just because the new tail.

c. Look at the test code loop that does a peek and dequeue. How did it know that the entire queue had been processed? What in the queue code maintained the boolean named "empty"? Did maintenance need to use a linked list function?

The queue contains a boolean called empty to return whether the stack is empty (true) or not empty (false). The while loop that does a peek and dequeue reads the value of the Boolean called empty to know when the entire queue has been processed. Empty is originally set to true when initQueue is called. Empty is set to false whenever an enqueue is called because a data node is added to the linked list. Dequeue sets empty to true or false based on the condition that the linked list contains 1 data set. Peek checks whether the queue is not NULL and not empty to return a valid front node. Since empty will be false at the beginning of the loop and true at the end of the loop’s condition after dequeue has deleted all nodes in the linked list, the while loop can start and complete fired logic properly. Peek and dequeue need to return the result of a linked list function called LL\_GetFront to display what is next in line to be dequeued and what has been dequeued.

d. During the loop identified in (c), does the order of data peeked and dequeued make sense, given the order it was enqueued? Why didn’t the AllocationCount change as data was peeked, but did change when data was dequeued?

The order of the data values peeked and dequeued make sense because the queue is of first in-first out (FIFA) type. This means the first (not necessarily most recently) added item will be the next to be removed. First, enqueue is called adding 1000 as the tail and head. Then another enqueue is called adding 2000 as the next tail (or end) so that the queue is {1000, 2000}. Then two more enqueues are called to add 3000 then 4000 making the queue {1000, 2000, 3000, 4000}. The sequence of peeks and dequeues starts with the first added, 1000. First peek will display 1000 because it is the front or head of the queue. Then it gets deleted because dequeue is called to remove the front node. This is repeated as many times as needed until the queue is empty. After the first peek and dequeue, the queue will be {2000, 3000, 4000}. Peek will display 2000 because it is the new front or head of the queue. Then it gets deleted because dequeue is called to remove the front node. After which the queue will be {3000, 4000}. This occurs two more times until the queue is empty. AllocationCount changed as data was dequeued but not when peeked because peek will only return a valid front node if there is one. Dequeue is the function responsible for deleting a node which requires the release of dynamically allocated memory for that node. Once the node is deleted, memory is freed and AllocationCount decreases by 1.

e. Does it make sense that the AllocationCount was 2 after all of the queue data was dequeued? If so, why?

It makes sense that AllocationCount was 2 after the queue data was dequeued. Dequeues were called until the queue was empty, decreasing AllocationCount by 1 each time. During which, nodes get deleted hence the decrease of AllocationCount. Once empty, the queue and linked list still exist in memory. Therefore, AllocationCount is still 2 even though the queue data has been dequeued. They have been initialized and will not be freed until the queue is deleted.

f. Explain what your deleteQueue does and why the AllocationCount went from 2 to 0.

deleteQueue checks that the queue is not equal to NULL. Then it gets the queue's linked list and deletes it also by checking that the linked list is not equal to NULL. It does this by getting the front node and it deletes it, removing dynamically allocated memory for each node until there are no longer any nodes in the linked list. Each time, AllocationCount decreases by 1 until there are no nodes and AllocationCount equals 2. Then the information structure (linked list) is deleted, freeing its memory and AllocationCount decreases by 1 to equal 1. It returns NULL to indicate that there is no longer a linked list. After the queue's linked list is deleted, the queue is freed and AllocationCount decreases by 1 to equal 0. It also returns NULL to indicate there is no longer a queue.

**Problem 3 follows…**  
**Description:** Problem 3 is designed to demonstrate priority queue construction using a linked list. It consists of functions and definitions for the queue including initializing, checking if the queue is empty, peek, enqueue/dequeue and deleting the queue. It also contains other system C capabilities such as random number generation and delay. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code (PriorityQueueDemo.c):**

// PriorityQueueTester will demonstrate the init, enqueue, peek, dequeue and delete

// for three different types of queues (a queue without priority, a queue where the

// lowest number is the highest priority and a queue where the highest number is

// the highest priority.

// - It prints the type of queue being tested

// - It builds a queue that holds UserData generated based on user's

// current time across an interval

// - It peeks at the queue, returning UserData

// - It dequeues UserData from the queue, returning UserData

// - As it dequeues, every so often it enqueues other

// generated time data to prove the queue is still

// intact, to get removed just like other queue data

// - it uses empty() to determine if the queue holds any data that

// can be dequeued or peeked

// - when done, it deletes the queue

// For demonstration purposes, it shows the number of allocations for

// everything it does (NOTE: it prints only once after generating time data).

#include <stdio.h>

// we will use rand() and srand() from stdlib.h

// we will be getting system data and time from time.h

#include <time.h>

#include <stdlib.h>

// we will use strcpy() and strlen() from string.h

#include <string.h>

// we use a bool from stdbool.h

#include <stdbool.h>

// we use Queue functions from Queue.h

#include "Queue.h"

// we use UserData for the queue

#include "UserData.h"

#define MAXPRIO 4

#define DEQUEUESPERENQUEUE 3

#define INITIALENQUEUES 15

// this local function receives a queue

// as an argument to populates, peek

// and dequeue it, while printing

// messages and AllocationCount

// along the way

static void Runtest (Queue Q);

// this local function receives a queue

// and a number of items as arguments

// so to fill the queue with that data

static void buildQueue (Queue Q, int numItems);

// this local function creates a data

// and assign it a random priority number

// between 1 and MAXPRIO

static UserData genTimePriorityUserData ();

// this local function takes two UserData

// objects in order to update a queue

// by comparing priority numbers of those

// two UserData objects. It returns true or false

// so that queue knows whether data with lower

// priority numbers are listed as higher priority

// in the queue

static bool LowestNumIsHighestPriority (UserData first, UserData second);

// this local function takes two UserData

// objects in order to update a queue

// by comparing priority numbers of those

// two UserData objects. It returns true or false

// so that queue knows whether data with higher

// priority numbers are listed as higher priority

// in the queue

static bool HighestNumIsHighestPriority (UserData first, UserData second);

// this is an int variable that is defined elsewhere

// in the source code. It is listed here as extern

// so PriorityQueueDemo.c can declare that same

// variable which is already defined

extern int AllocationCount;

// buildQueue takes a queue and a number

// in order to fill the queue with

// data and print that data along with

// its priority number

void buildQueue (Queue Q, int numItems)

{

for (; numItems > 0; numItems--)

{

UserData D = genTimePriorityUserData();

enqueue (Q, D);

printf ("Time = %s queued at priority %d\n", D.time, D.priority);

}

return;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Function: genTimePriorityUserData

// This function fills a UserData structure with the current

// system time and a random priority from 1 to MAXPRIO.

// Before exiting, this routine spins for 1 second so

// that another, immediate call will have a different

// time stamp. It then returns the populated UserData.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

UserData genTimePriorityUserData ()

{

// get the current time

time\_t current\_time = time(NULL);

// convert it to ASCII and point to it

char\* theTime = ctime(&current\_time);

// trim off the \n for prettier printing

for (int loop = 0; loop < strlen(theTime); loop++)

if (theTime[loop] == '\n')

theTime[loop] = 0;

// fill in time and a priority (1 to MAXPRIO)

UserData D;

strcpy (D.time, theTime);

D.priority = 1 + rand() % MAXPRIO;

// spin for at least 1 second so that the time field will

// be different if we are called again immediately

time\_t time\_now;

do

{

// get the current time (seconds)

time\_now= time(NULL);

// spin here if it hasn't changed by at least 1 second

}

while ((time\_now - current\_time) == 0);

return D;

}

// LowestNumIsHighestPriority is called by the queue

// whenever the queue needs to be updated to maintain priority.

// It performs a comparison in support of the sort used by

// the PriorityQueue implementation

bool LowestNumIsHighestPriority (UserData first, UserData second)

{

// treat the lower priority number as more important

if (first.priority <= second.priority)

return true;

else

return false;

}

// HighestNumIsHighestPriority is called by the queue

// whenever the queue needs to be updated to maintain priority.

// It performs a comparison in support of the sort used by

// the PriorityQueue implementation

bool HighestNumIsHighestPriority (UserData first, UserData second)

{

// treat the lower priority number as more important

if (first.priority > second.priority)

return true;

else

return false;

}

void Runtest (Queue Q)

{

// seed the random number generator so it doesn't always

// start with the same value

time\_t t;

srand((unsigned) time(&t));

// start with a queue filled with INITIALENQUEUES items of random priority

// buildQueue will insure that the timestamps are separated by 1

// second so we can see that priority queue behavior is done correctly

buildQueue (Q, INITIALENQUEUES);

printf ("Total allocations is %d after buildQueue\n", AllocationCount);

printf ("Starting to dequeue and queue information\n");

// we will enqueue 1 item after dequeueing at most DEQUESPERENQUEUE

// items by priority unless the queue is exhausted.

// if that happens, we are done

int NumToDequeue = DEQUEUESPERENQUEUE;

// peek data, print it, then dequeue it and print it

// until the queue no longer contains any data.

// data is added along the way to prove the queue is

// still functional or intact and this new data

// gets dequeued as well.

while (empty(Q) != true)

{

UserData D = dequeue (Q);

printf (" Allocation = %2d, dequeued data: ", AllocationCount);

printf ("Priority %-3d Time = %s\n", D.priority, D.time);

// if we have dequeued DEQUEUESPERENQUEUE items, generate

// and queue another one to show that the queue is intact

// even if we have begun dequeueing items

if (--NumToDequeue == 0)

{

// generate random data and a time stamp

UserData D = genTimePriorityUserData();

// place previously randomly generated

// data into the queue at the end of

// the linked list

enqueue (Q, D);

printf ("Time = %s queued at priority %d\n", D.time, D.priority);

// reestablish how many items to dequeue before the next enqueue

NumToDequeue = DEQUEUESPERENQUEUE;

}

}

// only the allocation of the queue itself remains to be freed

printf ("Total allocations is %2d after all the dequeue calls\n", AllocationCount);

return;

}

// main takes no arguments and is responsible for testing three different kinds

// of queues including a queue without priority, a queue where the lowest number

// is the highest priority and a queue where the highest number is the highest

// priority. Each test allocates a queue and its linked list but also

// prints descriptions of steps taken to test the queue as well as

// AllocationCount along the way. This includes the generation of 15

// sequential times, each a second apart, based on user's current time.

// That data is printed and added into the queue. Then that data is removed from

// the queue differently based on what kind of queue is used including first in-first

// out, first in-first out with lowest number as dequeue priority and first in-first

// out with highest number as dequeue priority. Dequeues are printed alongside

// the AllocationCount. Along the way, new data is added to the queue to prove

// the queue is still intact. This data gets removed too so that after all data has

// been removed, the queue is deleted and the next queue is tested.

int main()

{

printf ("Demonstrating how the queue works WITHOUT a priority application\n");

// provide NULL so that no priority check will be done

Queue Q = initQueue(NULL);

printf ("Total allocations is %d after initQueue\n", AllocationCount);

Runtest(Q);

Q = deleteQueue (Q);

printf ("After deleteQueue, remaining allocations is %d \n", AllocationCount);

printf ("\n\nDemonstrating how the queue works WITH a priority application\n");

printf ("The lowest priority number should be the highest priority to dequeue\n");

// provide a routine to test priorities so that priority will be maintained

Q = initQueue(LowestNumIsHighestPriority);

printf ("Total allocations is %d after initQueue\n", AllocationCount);

Runtest(Q);

Q = deleteQueue (Q);

printf ("\n\nDemonstrating how the queue works WITH a priority application\n");

printf ("The highest priority number should be the highest priority to dequeue\n");

// provide a routine to test priorities so that priority will be maintained

Q = initQueue(HighestNumIsHighestPriority);

printf ("Total allocations is %d after initQueue\n", AllocationCount);

Runtest(Q);

Q = deleteQueue (Q);

printf ("After deleteQueue, remaining allocations is %d \n", AllocationCount);

return 0;

}**Output File:**

Demonstrating how the queue works WITHOUT a priority application

Total allocations is 2 after initQueue

Time = Sat Jun 20 13:00:43 2020 queued at priority 3

Time = Sat Jun 20 13:00:44 2020 queued at priority 4

Time = Sat Jun 20 13:00:45 2020 queued at priority 2

Time = Sat Jun 20 13:00:46 2020 queued at priority 4

Time = Sat Jun 20 13:00:47 2020 queued at priority 2

Time = Sat Jun 20 13:00:48 2020 queued at priority 1

Time = Sat Jun 20 13:00:49 2020 queued at priority 2

Time = Sat Jun 20 13:00:50 2020 queued at priority 4

Time = Sat Jun 20 13:00:51 2020 queued at priority 4

Time = Sat Jun 20 13:00:52 2020 queued at priority 1

Time = Sat Jun 20 13:00:53 2020 queued at priority 3

Time = Sat Jun 20 13:00:54 2020 queued at priority 4

Time = Sat Jun 20 13:00:55 2020 queued at priority 2

Time = Sat Jun 20 13:00:56 2020 queued at priority 1

Time = Sat Jun 20 13:00:57 2020 queued at priority 4

Total allocations is 17 after buildQueue

Starting to dequeue and queue information

Allocation = 16, dequeued data: Priority 3 Time = Sat Jun 20 13:00:43 2020

Allocation = 15, dequeued data: Priority 4 Time = Sat Jun 20 13:00:44 2020

Allocation = 14, dequeued data: Priority 2 Time = Sat Jun 20 13:00:45 2020

Time = Sat Jun 20 13:00:58 2020 queued at priority 3

Allocation = 14, dequeued data: Priority 4 Time = Sat Jun 20 13:00:46 2020

Allocation = 13, dequeued data: Priority 2 Time = Sat Jun 20 13:00:47 2020

Allocation = 12, dequeued data: Priority 1 Time = Sat Jun 20 13:00:48 2020

Time = Sat Jun 20 13:00:59 2020 queued at priority 4

Allocation = 12, dequeued data: Priority 2 Time = Sat Jun 20 13:00:49 2020

Allocation = 11, dequeued data: Priority 4 Time = Sat Jun 20 13:00:50 2020

Allocation = 10, dequeued data: Priority 4 Time = Sat Jun 20 13:00:51 2020

Time = Sat Jun 20 13:01:00 2020 queued at priority 3

Allocation = 10, dequeued data: Priority 1 Time = Sat Jun 20 13:00:52 2020

Allocation = 9, dequeued data: Priority 3 Time = Sat Jun 20 13:00:53 2020

Allocation = 8, dequeued data: Priority 4 Time = Sat Jun 20 13:00:54 2020

Time = Sat Jun 20 13:01:01 2020 queued at priority 1

Allocation = 8, dequeued data: Priority 2 Time = Sat Jun 20 13:00:55 2020

Allocation = 7, dequeued data: Priority 1 Time = Sat Jun 20 13:00:56 2020

Allocation = 6, dequeued data: Priority 4 Time = Sat Jun 20 13:00:57 2020

Time = Sat Jun 20 13:01:02 2020 queued at priority 4

Allocation = 6, dequeued data: Priority 3 Time = Sat Jun 20 13:00:58 2020

Allocation = 5, dequeued data: Priority 4 Time = Sat Jun 20 13:00:59 2020

Allocation = 4, dequeued data: Priority 3 Time = Sat Jun 20 13:01:00 2020

Time = Sat Jun 20 13:01:03 2020 queued at priority 2

Allocation = 4, dequeued data: Priority 1 Time = Sat Jun 20 13:01:01 2020

Allocation = 3, dequeued data: Priority 4 Time = Sat Jun 20 13:01:02 2020

Allocation = 2, dequeued data: Priority 2 Time = Sat Jun 20 13:01:03 2020

Time = Sat Jun 20 13:01:04 2020 queued at priority 4

Allocation = 2, dequeued data: Priority 4 Time = Sat Jun 20 13:01:04 2020

Total allocations is 2 after all the dequeue calls

After deleteQueue, remaining allocations is 0

Demonstrating how the queue works WITH a priority application

The lowest priority number should be the highest priority to dequeue

Total allocations is 2 after initQueue

Time = Sat Jun 20 13:01:05 2020 queued at priority 3

Time = Sat Jun 20 13:01:06 2020 queued at priority 1

Time = Sat Jun 20 13:01:07 2020 queued at priority 2

Time = Sat Jun 20 13:01:08 2020 queued at priority 4

Time = Sat Jun 20 13:01:09 2020 queued at priority 3

Time = Sat Jun 20 13:01:10 2020 queued at priority 1

Time = Sat Jun 20 13:01:11 2020 queued at priority 4

Time = Sat Jun 20 13:01:12 2020 queued at priority 4

Time = Sat Jun 20 13:01:13 2020 queued at priority 3

Time = Sat Jun 20 13:01:14 2020 queued at priority 3

Time = Sat Jun 20 13:01:15 2020 queued at priority 1

Time = Sat Jun 20 13:01:16 2020 queued at priority 1

Time = Sat Jun 20 13:01:17 2020 queued at priority 2

Time = Sat Jun 20 13:01:18 2020 queued at priority 4

Time = Sat Jun 20 13:01:19 2020 queued at priority 3

Total allocations is 17 after buildQueue

Starting to dequeue and queue information

Allocation = 16, dequeued data: Priority 1 Time = Sat Jun 20 13:01:06 2020

Allocation = 15, dequeued data: Priority 1 Time = Sat Jun 20 13:01:16 2020

Allocation = 14, dequeued data: Priority 1 Time = Sat Jun 20 13:01:10 2020

Time = Sat Jun 20 13:01:20 2020 queued at priority 1

Allocation = 14, dequeued data: Priority 1 Time = Sat Jun 20 13:01:20 2020

Allocation = 13, dequeued data: Priority 1 Time = Sat Jun 20 13:01:15 2020

Allocation = 12, dequeued data: Priority 2 Time = Sat Jun 20 13:01:07 2020

Time = Sat Jun 20 13:01:21 2020 queued at priority 3

Allocation = 12, dequeued data: Priority 2 Time = Sat Jun 20 13:01:17 2020

Allocation = 11, dequeued data: Priority 3 Time = Sat Jun 20 13:01:21 2020

Allocation = 10, dequeued data: Priority 3 Time = Sat Jun 20 13:01:19 2020

Time = Sat Jun 20 13:01:22 2020 queued at priority 2

Allocation = 10, dequeued data: Priority 2 Time = Sat Jun 20 13:01:22 2020

Allocation = 9, dequeued data: Priority 3 Time = Sat Jun 20 13:01:13 2020

Allocation = 8, dequeued data: Priority 3 Time = Sat Jun 20 13:01:14 2020

Time = Sat Jun 20 13:01:23 2020 queued at priority 1

Allocation = 8, dequeued data: Priority 1 Time = Sat Jun 20 13:01:23 2020

Allocation = 7, dequeued data: Priority 3 Time = Sat Jun 20 13:01:05 2020

Allocation = 6, dequeued data: Priority 3 Time = Sat Jun 20 13:01:09 2020

Time = Sat Jun 20 13:01:24 2020 queued at priority 3

Allocation = 6, dequeued data: Priority 3 Time = Sat Jun 20 13:01:24 2020

Allocation = 5, dequeued data: Priority 4 Time = Sat Jun 20 13:01:18 2020

Allocation = 4, dequeued data: Priority 4 Time = Sat Jun 20 13:01:12 2020

Time = Sat Jun 20 13:01:25 2020 queued at priority 3

Allocation = 4, dequeued data: Priority 3 Time = Sat Jun 20 13:01:25 2020

Allocation = 3, dequeued data: Priority 4 Time = Sat Jun 20 13:01:11 2020

Allocation = 2, dequeued data: Priority 4 Time = Sat Jun 20 13:01:08 2020

Time = Sat Jun 20 13:01:26 2020 queued at priority 4

Allocation = 2, dequeued data: Priority 4 Time = Sat Jun 20 13:01:26 2020

Total allocations is 2 after all the dequeue calls

Demonstrating how the queue works WITH a priority application

The highest priority number should be the highest priority to dequeue

Total allocations is 2 after initQueue

Time = Sat Jun 20 13:01:27 2020 queued at priority 3

Time = Sat Jun 20 13:01:28 2020 queued at priority 2

Time = Sat Jun 20 13:01:29 2020 queued at priority 2

Time = Sat Jun 20 13:01:30 2020 queued at priority 1

Time = Sat Jun 20 13:01:31 2020 queued at priority 4

Time = Sat Jun 20 13:01:32 2020 queued at priority 1

Time = Sat Jun 20 13:01:33 2020 queued at priority 2

Time = Sat Jun 20 13:01:34 2020 queued at priority 4

Time = Sat Jun 20 13:01:35 2020 queued at priority 2

Time = Sat Jun 20 13:01:36 2020 queued at priority 2

Time = Sat Jun 20 13:01:37 2020 queued at priority 3

Time = Sat Jun 20 13:01:38 2020 queued at priority 3

Time = Sat Jun 20 13:01:39 2020 queued at priority 1

Time = Sat Jun 20 13:01:40 2020 queued at priority 3

Time = Sat Jun 20 13:01:41 2020 queued at priority 3

Total allocations is 17 after buildQueue

Starting to dequeue and queue information

Allocation = 16, dequeued data: Priority 4 Time = Sat Jun 20 13:01:31 2020

Allocation = 15, dequeued data: Priority 4 Time = Sat Jun 20 13:01:34 2020

Allocation = 14, dequeued data: Priority 3 Time = Sat Jun 20 13:01:27 2020

Time = Sat Jun 20 13:01:42 2020 queued at priority 3

Allocation = 14, dequeued data: Priority 3 Time = Sat Jun 20 13:01:37 2020

Allocation = 13, dequeued data: Priority 3 Time = Sat Jun 20 13:01:38 2020

Allocation = 12, dequeued data: Priority 3 Time = Sat Jun 20 13:01:40 2020

Time = Sat Jun 20 13:01:43 2020 queued at priority 3

Allocation = 12, dequeued data: Priority 3 Time = Sat Jun 20 13:01:41 2020

Allocation = 11, dequeued data: Priority 3 Time = Sat Jun 20 13:01:42 2020

Allocation = 10, dequeued data: Priority 3 Time = Sat Jun 20 13:01:43 2020

Time = Sat Jun 20 13:01:44 2020 queued at priority 1

Allocation = 10, dequeued data: Priority 2 Time = Sat Jun 20 13:01:28 2020

Allocation = 9, dequeued data: Priority 2 Time = Sat Jun 20 13:01:29 2020

Allocation = 8, dequeued data: Priority 2 Time = Sat Jun 20 13:01:33 2020

Time = Sat Jun 20 13:01:45 2020 queued at priority 1

Allocation = 8, dequeued data: Priority 2 Time = Sat Jun 20 13:01:35 2020

Allocation = 7, dequeued data: Priority 2 Time = Sat Jun 20 13:01:36 2020

Allocation = 6, dequeued data: Priority 1 Time = Sat Jun 20 13:01:30 2020

Time = Sat Jun 20 13:01:46 2020 queued at priority 2

Allocation = 6, dequeued data: Priority 2 Time = Sat Jun 20 13:01:46 2020

Allocation = 5, dequeued data: Priority 1 Time = Sat Jun 20 13:01:32 2020

Allocation = 4, dequeued data: Priority 1 Time = Sat Jun 20 13:01:39 2020

Time = Sat Jun 20 13:01:47 2020 queued at priority 4

Allocation = 4, dequeued data: Priority 4 Time = Sat Jun 20 13:01:47 2020

Allocation = 3, dequeued data: Priority 1 Time = Sat Jun 20 13:01:44 2020

Allocation = 2, dequeued data: Priority 1 Time = Sat Jun 20 13:01:45 2020

Time = Sat Jun 20 13:01:48 2020 queued at priority 1

Allocation = 2, dequeued data: Priority 1 Time = Sat Jun 20 13:01:48 2020

Total allocations is 2 after all the dequeue calls

After deleteQueue, remaining allocations is 0

**Output Explanation:**

The order of the data values peeked and dequeued for the queue without a priority application make sense because the queue is of first in-first out (FIFA) type. This means the first (not necessarily most recently) added item will be the next to be removed. AllocationCount shows 2 after initialization of the queue and linked list. First, enqueue is called adding:

Time = Sat Jun 20 13:00:43 2020 queued at priority 3

as the tail and head. Then another enqueue is called adding:

Time = Sat Jun 20 13:00:44 2020 queued at priority 4

as the next tail (or end) so that the queue is:

{Time = Sat Jun 20 13:00:43 2020 queued at priority 3, //front

Time = Sat Jun 20 13:00:44 2020 queued at priority 4} //end

Then the rest of enqueues occur making the queue:

{Time = Sat Jun 20 13:00:43 2020 queued at priority 3, //front

Time = Sat Jun 20 13:00:44 2020 queued at priority 4,

Time = Sat Jun 20 13:00:45 2020 queued at priority 2,

Time = Sat Jun 20 13:00:46 2020 queued at priority 4,

Time = Sat Jun 20 13:00:47 2020 queued at priority 2,

Time = Sat Jun 20 13:00:48 2020 queued at priority 1,

Time = Sat Jun 20 13:00:49 2020 queued at priority 2,

Time = Sat Jun 20 13:00:50 2020 queued at priority 4,

Time = Sat Jun 20 13:00:51 2020 queued at priority 4,

Time = Sat Jun 20 13:00:52 2020 queued at priority 1,

Time = Sat Jun 20 13:00:53 2020 queued at priority 3,

Time = Sat Jun 20 13:00:54 2020 queued at priority 4,

Time = Sat Jun 20 13:00:55 2020 queued at priority 2,

Time = Sat Jun 20 13:00:56 2020 queued at priority 1,

Time = Sat Jun 20 13:00:57 2020 queued at priority 4} //end

AllocationCount increases to 17 to show memory is allocated for these new 15 data nodes in addition to the already initialized queue and linked list. The sequence of peeks and dequeues starts with the first added:

Time = Sat Jun 20 13:00:43 2020 queued at priority 3

This gets displayed first by peek because it is the front or head of the queue. Then it gets deleted because dequeue is called to remove the front node. This is repeated two more times until an enqueue happens to prove that the queue is still in tact and that new data can be added to the queue properly. The first example of this is:

Time = Sat Jun 20 13:00:58 2020 queued at priority 3

This enqueue adds data to the tail (end) of the linked list. This repeats itself until all 15 dequeues have occurred to remove all data from the queue that was generated by the function genTimePriorityUserData(). But since enqueue was occurring during the dequeue process, there are still leftover newly generated time data in the queue. These also get removed by the dequeue process until the queue is empty. So, the order in which data gets removed is the following:

Time = Sat Jun 20 13:00:43 2020 queued at priority 3

Time = Sat Jun 20 13:00:44 2020 queued at priority 4

Time = Sat Jun 20 13:00:45 2020 queued at priority 2

Time = Sat Jun 20 13:00:46 2020 queued at priority 4

Time = Sat Jun 20 13:00:47 2020 queued at priority 2

Time = Sat Jun 20 13:00:48 2020 queued at priority 1

Time = Sat Jun 20 13:00:49 2020 queued at priority 2

Time = Sat Jun 20 13:00:50 2020 queued at priority 4

Time = Sat Jun 20 13:00:51 2020 queued at priority 4

Time = Sat Jun 20 13:00:52 2020 queued at priority 1

Time = Sat Jun 20 13:00:53 2020 queued at priority 3

Time = Sat Jun 20 13:00:54 2020 queued at priority 4

Time = Sat Jun 20 13:00:55 2020 queued at priority 2

Time = Sat Jun 20 13:00:56 2020 queued at priority 1

Time = Sat Jun 20 13:00:57 2020 queued at priority 4

//data below was added after original data generation during dequeue process

Time = Sat Jun 20 13:00:58 2020 queued at priority 3

Time = Sat Jun 20 13:00:59 2020 queued at priority 4

Time = Sat Jun 20 13:01:00 2020 queued at priority 3

Time = Sat Jun 20 13:01:01 2020 queued at priority 1

Time = Sat Jun 20 13:01:02 2020 queued at priority 4

Time = Sat Jun 20 13:01:03 2020 queued at priority 2

Time = Sat Jun 20 13:01:04 2020 queued at priority 4

AllocationCount changed as data was dequeued but not when peeked because peek will only return a valid front node if there is one. Dequeue is the function responsible for deleting a node which requires the release of dynamically allocated memory for that node. Once the node is deleted, memory is freed and AllocationCount decreases by 1. However, after a data is enqueued during the dequeue process, you will notice AllocationCount remains the same, i.e. at AllocationCount of 14, 12, 10, 8, 6, 4 and 2. This is because before one node is deleted, an enqueue creates a new node leaving the AllocationCount "unchanged" when the next print fires to print AllocationCount. AllocationCount shows 0 after deletion of the queue and linked list.

The order of the data values peeked and dequeued for the queue with a priority application (where the lowest priority number should be the highest priority to dequeue) make sense because the queue is of first in-first out (FIFA) type with data that gets sorted by its priority number. This means the closest to first (not necessarily most recently) added item that has the lowest priority number will be the next to be removed. AllocationCount shows 2 after initialization of the queue and linked list. First, enqueue is called adding:

Time = Sat Jun 20 13:01:05 2020 queued at priority 3

as the tail and head. Then another enqueue is called adding:

Time = Sat Jun 20 13:01:06 2020 queued at priority 1

in this case eventually to the next head (or front) because the enqueue swaps it with the closest to first data with the next highest priority. So then the queue is:

{Time = Sat Jun 20 13:01:06 2020 queued at priority 1, //front

Time = Sat Jun 20 13:01:05 2020 queued at priority 3} //end

Then the rest of enqueues occur making the queue:

{Time = Sat Jun 20 13:01:06 2020 queued at Priority 1, //start

Time = Sat Jun 20 13:01:16 2020 queued at Priority 1,

Time = Sat Jun 20 13:01:10 2020 queued at Priority 1,

Time = Sat Jun 20 13:01:15 2020 queued at Priority 1,

Time = Sat Jun 20 13:01:07 2020 queued at Priority 2,

Time = Sat Jun 20 13:01:17 2020 queued at Priority 2,

Time = Sat Jun 20 13:01:19 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:13 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:14 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:05 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:09 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:18 2020 queued at Priority 4,

Time = Sat Jun 20 13:01:12 2020 queued at Priority 4,

Time = Sat Jun 20 13:01:11 2020 queued at Priority 4,

Time = Sat Jun 20 13:01:08 2020 queued at Priority 4} //end

AllocationCount increases to 17 to show memory is allocated for these new 15 data nodes in addition to the already initialized queue and linked list. The sequence of peeks and dequeues starts with the data closest to first added which happens to be in sorted order of lowest priority number:

Time = Sat Jun 20 13:01:06 2020 queued at priority 1

This gets displayed first by peek because it is the front or head of the queue after being sorted by lower priority number during the enqueue process. The lower priority number in this test means it more important to dequeue next than those with a higher priority number. Then it gets deleted because dequeue is called to remove the head (or front) of the linked list. This is repeated two more times until an enqueue happens to prove that the queue is still in tact and that new data can be added to the queue properly. The first example of this is:

Time = Sat Jun 20 13:01:20 2020 queued at priority 1

This enqueue adds data to the tail (end) of the linked list then gets swapped (based on lower priority and closeness to the front) with other existing entries so the queue is in sorted order for when the next dequeue occurs. This repeats itself until all 22 (since enqueue added 7 more entries into the queue) dequeues have occurred to remove all data from the queue. So, the order in which data gets removed is the following:

Time = Sat Jun 20 13:01:06 2020 queued at Priority 1

Time = Sat Jun 20 13:01:16 2020 queued at Priority 1

Time = Sat Jun 20 13:01:10 2020 queued at Priority 1

Time = Sat Jun 20 13:01:20 2020 queued at Priority 1

Time = Sat Jun 20 13:01:15 2020 queued at Priority 1

Time = Sat Jun 20 13:01:07 2020 queued at Priority 2

Time = Sat Jun 20 13:01:17 2020 queued at Priority 2

Time = Sat Jun 20 13:01:21 2020 queued at Priority 3

Time = Sat Jun 20 13:01:19 2020 queued at Priority 3

Time = Sat Jun 20 13:01:22 2020 queued at Priority 2

Time = Sat Jun 20 13:01:13 2020 queued at Priority 3

Time = Sat Jun 20 13:01:14 2020 queued at Priority 3

Time = Sat Jun 20 13:01:23 2020 queued at Priority 1

Time = Sat Jun 20 13:01:05 2020 queued at Priority 3

Time = Sat Jun 20 13:01:09 2020 queued at Priority 3

Time = Sat Jun 20 13:01:24 2020 queued at Priority 3

Time = Sat Jun 20 13:01:18 2020 queued at Priority 4

Time = Sat Jun 20 13:01:12 2020 queued at Priority 4

Time = Sat Jun 20 13:01:25 2020 queued at Priority 3

Time = Sat Jun 20 13:01:11 2020 queued at Priority 4

Time = Sat Jun 20 13:01:08 2020 queued at Priority 4

Time = Sat Jun 20 13:01:26 2020 queued at Priority 4

AllocationCount changed as data was dequeued but not when peeked because peek will only return a valid front node if there is one. Dequeue is the function responsible for deleting a node which requires the release of dynamically allocated memory for that node. Once the node is deleted, memory is freed and AllocationCount decreases by 1. However, after a data is enqueued during the dequeue process, you will notice AllocationCount remains the same, i.e. at AllocationCount of 14, 12, 10, 8, 6, 4 and 2. This is because before one node is deleted, an enqueue creates a new node leaving the AllocationCount "unchanged" when the next print fires to print AllocationCount. AllocationCount shows 0 after deletion of the queue and linked list.

The order of the data values peeked and dequeued for the queue with a priority application (where the highest priority number should be the highest priority to dequeue) make sense because the queue is of first in-first out (FIFA) type with data that gets sorted by its priority number. This means the closest to first (not necessarily most recently) added item that has the highest priority number will be the next to be removed. AllocationCount shows 2 after initialization of the queue and linked list. First, enqueue is called adding:

Time = Sat Jun 20 13:01:27 2020 queued at priority 3

as the tail and head. Then another enqueue is called adding:

Time = Sat Jun 20 13:01:28 2020 queued at priority 2

in this case to the tail (or end) without actually sawpping itself for another entry because its priority (2) is lower than that of the other entry (3) in the queue. So then the queue is:

{Time = Sat Jun 20 13:01:27 2020 queued at priority 3, //start

Time = Sat Jun 20 13:01:28 2020 queued at priority 2} //end

Then the rest of enqueues occur making the queue:

{Time = Sat Jun 20 13:01:31 2020 queued at Priority 4, //start

Time = Sat Jun 20 13:01:34 2020 queued at Priority 4,

Time = Sat Jun 20 13:01:27 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:37 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:38 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:40 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:41 2020 queued at Priority 3,

Time = Sat Jun 20 13:01:28 2020 queued at Priority 2,

Time = Sat Jun 20 13:01:29 2020 queued at Priority 2,

Time = Sat Jun 20 13:01:33 2020 queued at Priority 2,

Time = Sat Jun 20 13:01:35 2020 queued at Priority 2,

Time = Sat Jun 20 13:01:36 2020 queued at Priority 2,

Time = Sat Jun 20 13:01:30 2020 queued at Priority 1,

Time = Sat Jun 20 13:01:32 2020 queued at Priority 1,

Time = Sat Jun 20 13:01:39 2020 queued at Priority 1} //end

AllocationCount increases to 17 to show memory is allocated for these new 15 data nodes in addition to the already initialized queue and linked list. The sequence of peeks and dequeues starts with the data closest to first added which happens to be in sorted order of highest priority number:

Time = Sat Jun 20 13:01:31 2020 queued at Priority 4

This gets displayed first by peek because it is the front or head of the queue after being sorted by higher priority number during the enqueue process. The higher priority number in this test means it more important to dequeue next than those with a lower priority number. Then it gets deleted because dequeue is called to remove the head (or front) of the linked list. This is repeated two more times until an enqueue happens to prove that the queue is still in tact and that new data can be added to the queue properly. The first example of this is:

Time = Sat Jun 20 13:01:42 2020 queued at priority 3

This enqueue adds data to the tail (end) of the linked list then gets swapped (based on higher priority and closeness to the front) with other existing entries so the queue is in sorted order for when the next dequeue occurs. This repeats itself until all 22 (since enqueue added 7 more entries into the queue) dequeues have occurred to remove all data from the queue. So, the order in which data gets removed is the following:

Time = Sat Jun 20 13:01:31 2020 queued at Priority 4

Time = Sat Jun 20 13:01:34 2020 queued at Priority 4

Time = Sat Jun 20 13:01:27 2020 queued at Priority 3

Time = Sat Jun 20 13:01:37 2020 queued at Priority 3

Time = Sat Jun 20 13:01:38 2020 queued at Priority 3

Time = Sat Jun 20 13:01:40 2020 queued at Priority 3

Time = Sat Jun 20 13:01:41 2020 queued at Priority 3

Time = Sat Jun 20 13:01:42 2020 queued at Priority 3

Time = Sat Jun 20 13:01:43 2020 queued at Priority 3

Time = Sat Jun 20 13:01:28 2020 queued at Priority 2

Time = Sat Jun 20 13:01:29 2020 queued at Priority 2

Time = Sat Jun 20 13:01:33 2020 queued at Priority 2

Time = Sat Jun 20 13:01:35 2020 queued at Priority 2

Time = Sat Jun 20 13:01:36 2020 queued at Priority 2

Time = Sat Jun 20 13:01:30 2020 queued at Priority 1

Time = Sat Jun 20 13:01:46 2020 queued at Priority 2

Time = Sat Jun 20 13:01:32 2020 queued at Priority 1

Time = Sat Jun 20 13:01:39 2020 queued at Priority 1

Time = Sat Jun 20 13:01:47 2020 queued at Priority 4

Time = Sat Jun 20 13:01:44 2020 queued at Priority 1

Time = Sat Jun 20 13:01:45 2020 queued at Priority 1

Time = Sat Jun 20 13:01:48 2020 queued at Priority 1

AllocationCount changed as data was dequeued but not when peeked because peek will only return a valid front node if there is one. Dequeue is the function responsible for deleting a node which requires the release of dynamically allocated memory for that node. Once the node is deleted, memory is freed and AllocationCount decreases by 1. However, after a data is enqueued during the dequeue process, you will notice AllocationCount remains the same, i.e. at AllocationCount of 14, 12, 10, 8, 6, 4 and 2. This is because before one node is deleted, an enqueue creates a new node leaving the AllocationCount "unchanged" when the next print fires to print AllocationCount. AllocationCount shows 0 after deletion of the queue and linked list.