Priority Queue

A queue is a data structure where we add at the rear (this operation is called enqueue) and remove from the front (this operation is called deque). We say that the queue is FIFO – First In First Out or FCFS – First Come First served.

In the real world, not all are equal – some are more equal. Those who are more equal get the services before those who are less equal.

A priority queue has nodes with an additional information which decides about the deque operation. Let us call this field as priority. Let us assume that the higher the priority, faster the service.

In a priority queue, the enque operation stores the priority information.
The deque operation will remove or service the one with the higher priority. If more than one has the same priority, the deque will select that which joined the queue earlier.

The priority queue is a data structure with the interface enque and deque. We can implement this in number of ways. We shall discuss a few possible implementations.

Various implementations of priority queue:

1. priority queue using an unordered array.
    enqueue: add at the end
    dequeue : search; find the leftmost node with the highest priority
        remove it. Shift the nodes following this, to the left.
2. priority queue using an ordered array in non-decreasing order.
    Enqueue : insert at the right position based on priority. If there are more
        than one with the same priority, insert to the left of this bunch.
    dequeue : remove the last node
3. priority queue using an unordered linked list.
    Enqueue : add in the beginning. That is the fastest way of adding.
    Dequeue : search; find the leftmost node with the highest priority

remove it.

4. priority queue using an ordered linked list.

    Enqueue : insert at the right position based on priority. If there are more than one with the same priority, insert to the right of this bunch.

    Dequeue : remove the first node.

5. priority queue using heap.

    This is beyond the scope of this course. We shall not discuss this even though this is the most common way of implementation.

Priority queue implemented using an unordered linked list:

Our priority queue has a structure with a field called head which points to a node. The node contains a component and a link field. The component has fields priority and some string. The client shall know about the component. The rest shall be opaque to him.

```
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H
struct component
{
	char detail[20];
	int priority;
};
typedef struct component component_t;
struct node
{
	component_t c;
	struct node *link;
};
typedef struct node node_t;

struct priority_queue
{
	node_t *head;
};
typedef struct priority_queue priority_queue_t;

void init_queue(priority_queue_t *p_queue);
void enqueue(priority_queue_t *p_queue, const component_t* p_comp);
void dequeue(priority_queue_t *p_queue, component_t* p_comp);
void clean_queue(priority_queue_t *p_queue);

#endif
```

The client creates a priority queue, initialized it, calls enqueue and dequeue repeatedly in a menu driven looping structure. The enqueue takes priority queue, priority and detail as arguments. The dequeue takes priority queue and two other arguments to fetch the detail and the priority. Then the client calls a routine to clean up the remaining nodes.

```
#include <stdio.h>
#include <stdlib.h>
#include "1_priority_queue.h"
// does not check for q empty on dequeue and q full on enqueue
int main()
{
	priority_queue_t q;
	init_queue(&q);
	component_t c;
	int option;
	printf("enter an option : 0 : quit 1 :enqueue 2 : dequeue\n");
	scanf("%d", &option);
	while(option != 0)
	{
		switch(option)
		{
			case 1:
					printf("enter a string and the priority : \n");
					scanf("%s %d", c.detail, &c.priority);
					enqueue(&q, &c);
					break;
			case 2 :
					dequeue(&q, &c);
					printf("%s %d\n", c.detail, c.priority);
					break;
		}

		printf("enter an option : 0 : quit 1 :enqueue 2 : dequeue\n");
		scanf("%d", &option);
	}
	clean_queue(&q);
}
```

Let us examine the implementation.
We have not checked for queue empty and queue full conditions.
The function init_queue makes the head of the queue NULL.

The function enqueue creates a node, populates it and adds that in the beginning of the queue.

The function dequeue is a bit more complicated. We have to do three things.
a) find the rightmost node with the highest priority.

```
        We can assume that the highest priority is 0.
        int max = 0;
        We have to traverse the whole list.
        We keep two pointers prev and pres. The pointer prev is required for
                removal of the pres node.
        We initialize the two pointers.
        node_t* prev = NULL;
        node_t* pres = p_queue→head;
        We now loop.
        while(pres != NULL)
        {
                // some code
                prev = pres;
                pres = pres→link;
        }
        We always keep a pointer to the node before the one which has the
                highest priority so far. This is to facilitate removal later.

        if(pres->c.priority >= max)
                {
                        max = pres->c.priority;
                        prev_max = prev;
                }
```

b) copy relevant information to the parameters.

```
        strcpy(p_comp->detail, prev_max->link->c.detail);
        p_comp->priority = prev_max->link->c.priority;
```

c) remove it.

```
        There are two cases.
        1) Remove in the middle or the end of the list. The head will not change.
                temp = prev_max->link;
                prev_max->link = prev_max->link->link;
                free(temp);

        2) Remove in the beginning. The head will change.
                temp = p_queue->head;
                p_queue->head = p_queue->head->link;
                free(temp);
```

Thats all.

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

#include "1_priority_queue.h"

void init_queue(priority_queue_t *p_queue)
```

```
{
	p_queue->head = NULL;
}
void enqueue(priority_queue_t *p_queue, const component_t* p_comp)
{
	// create a  node
	node_t* temp = (node_t*)malloc(sizeof(node_t));
	// populate the component in the node from the parameter
	temp->c.priority = p_comp->priority;
	strcpy(temp->c.detail, p_comp->detail);
	// insert the node into the linked list
	temp->link = p_queue->head;
	p_queue->head = temp;
}
void dequeue(priority_queue_t *p_queue, component_t* p_comp)
{
	// find the node with the highest priority
	node_t* prev = NULL;
	node_t* pres = p_queue->head;
	int max = 0; // priorities are non zero positive values
	node_t* prev_max;
	while(pres != NULL)
	{
		if(pres->c.priority >= max)
		{
			max = pres->c.priority;
			prev_max = prev;
		}
		prev = pres;
		pres = pres->link;
	}
	node_t* temp; // node to remove
	// middle of list
	if(prev_max != NULL )
	{
		// copy to the parameter
		strcpy(p_comp->detail, prev_max->link->c.detail);
		p_comp->priority = prev_max->link->c.priority;
		// remove it
		// short ckt
		temp = prev_max->link;
		prev_max->link = prev_max->link->link;
		free(temp);
	}
	else // beginning of list; head will change
	{
		strcpy(p_comp->detail, p_queue->head->c.detail);
		p_comp->priority = p_queue->head->c.priority;
		// remove it
		// short ckt
		temp = p_queue->head;
		p_queue->head = p_queue->head->link;
```

```
                free(temp);
        }
}


void clean_queue(priority_queue_t *p_queue)
{
        // TODO
}
```