# Dynamic Data Structures with C

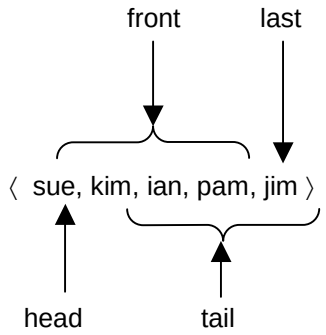
Terry Marris October 2010

## *4 Queues*

In the last chapter we looked at stacks. We noted that a stack is a FILO (first in/last out) data structure. A queue is a first in/first out (FIFO) data structure. Data is added to one end of a queue, and data is removed from the other end. Queues are used to model real-life queues such as aeroplanes queuing up to land, queues at supermarket checkouts, and a queue of jobs scheduled for printing on a computer system.

### 4.1 Queue

A queue is a sequence of data items together with operations to add data to one end, and to remove data from the other.

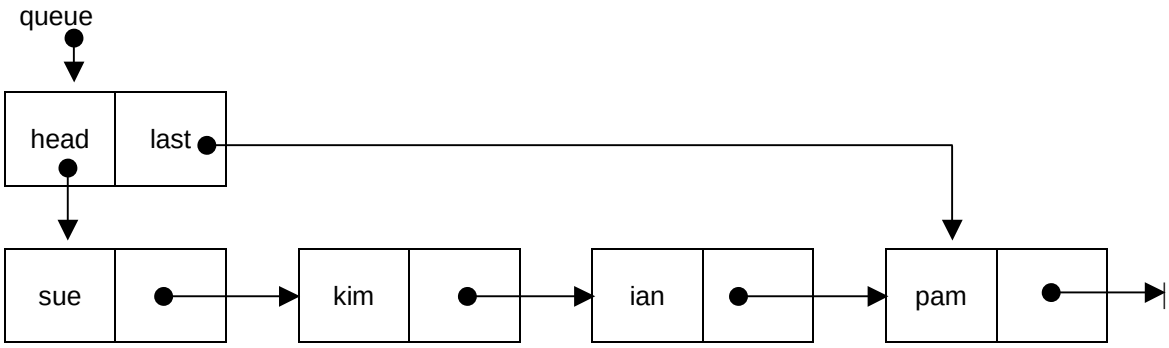



The *head* refers to the first item in the sequence, *tail* is the sequence without the head. *Last* refers to the last item in the sequence, *front* to the sequence without the last item.

Item *sue* has been in the queue for the longest time and is the first to depart from the queue. Item *jim* is the most recent arrival to the queue and will be the last to leave.

### 4.2 Diagrammatic Representation

A diagram representing a queue is shown below.



A *queue* is a pointer to a queue structure. The *head* points to the first node in the queue, *last* points to the last node in the queue.

Each node points to the next in sequence, except the last node.

Item *sue* is at the *head* of the queue, has been in the queue for the longest time, and will be the first to leave. Item *pam* is at the *last* location in the queue, has been in the queue for the shortest time, is the most recent arrival, and will be the last to depart.

## 4.3 Queue Interface

We introduce the interface to our *Queue* type. We specify *Queue* as a pointer to a *QueueHeader* (which is defined in the implementation) and several function prototypes.

```
/* queue.h */

#ifndef QUEUE
#define QUEUE

typedef struct QueueHeader *Queue;
typedef struct QueueIteratorStruct *QueueIterator;
typedef void *(*Duplicate)(void *);

/* queueError: reports queue errors, halts program execution */
int queueError(const char *e);

/* newQueue: returns a new, empty queue */
Queue newQueue(Duplicate);

/* disposeQueue: deallocates memory occupied by Queue */
Queue disposeQueue(Queue *q);

/* joinQueue: adds data item to end of queue */
Queue joinQueue(void * const data, const Queue q);

/* headOfQueue: returns item at head of queue, leaves queue unchanged
*/
void *headOfQueue(const Queue q);

/* leaveQueue: removes data item from front of queue */
Queue leaveQueue(const Queue q);

/* isEmptyQueue: returns 0 if queue is empty */
int isEmptyQueue(const Queue q);

/* reverseQueue: reverses the order of element s in a queue */
Queue reverseQueue(const Queue q);

/* copyQueue: returns a duplicate copy of the given queue */
Queue copyQueue(const Queue q);

/* newQueueIterator: returns a new iterator for the given queue */
QueueIterator newQueueIterator(const Queue q);

/* disposeQueueIterator: deallocates the memory occupied by the
   iterator */
QueueIterator disposeQueueIterator(QueueIterator *iter);
```

```
/* queueIteratorHasNext: returns 1 if there is an element not yet
   visited in the current iteration */
int queueIteratorHasNext(const QueueIterator iter);

/* queueIteratorNext: returns the next element in the queue */
void *queueIteratorNext(QueueIterator iter);

#endif
```

Given this interface we can perform the usual queue operations.

Create a new queue:

```
Queue q = newQueue((Duplicate)copyChar);
```

where

```
/* copyChar: returns a duplicate copy of its parameter */
char *copyChar(char *c)
{
  char *t = malloc(sizeof(char));
  t = c;
  return t;
}
```

Add a data item to the end of a queue:

```
char ch = 'X';
q = joinQueue(&ch, q);
```

Retrieve the item at the head of a queue:

```
char ch = *(char *)headOfQueue(q);
```

Determine whether a queue is empty:

```
if (isEmptyQueue(q)) ...
```

Remove the item at the head of a queue:

```
q = leaveQueue(q);
```

Reclaim the memory occupied by a queue:

```
disposeQueue(&q);
```

## 4.4 Queue Implementation

A queue is a sequence of data items. Items are added at one end, and removed from the other.

### 4.4.1 Nodes

We specify the queue node type as a pointer to a C structure named *QueueNodeStruct*.

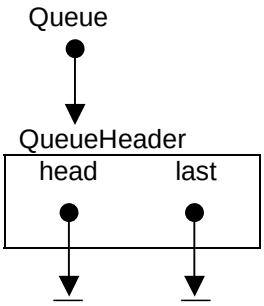
```
typedef struct QueueNodeStruct *QueueNode;

struct QueueNodeStruct {
  void *data;
  QueueNode next;
};
```

*QueueNodeStruct* contains a member named *next*. The *next* member is a pointer to a *QueueNodeStruct*. So we have a recursive definition of a queue node.

### 4.4.2 New Queue

We malloc a hole in memory for the queue header, and set its *head* and *last* members to *NULL*.




```
/* newQueue: returns a new, empty queue */
Queue newQueue(Duplicate dup)
{
  Queue q = malloc(sizeof(struct QueueHeader));
  if (q == NULL)
    queueError("newQueue: out of memory");
  q->head = NULL;
  q->last = NULL;
  q->duplicate = dup;
  return q;
}
```

The *duplicate* member is initialised with the copy function provided by the user.

### 4.4.3 Empty Queue

We have a condition for an empty queue: both *head* and *last* members of the *QueueHeader* are *NULL*.

```
/* isEmptyQueue: returns 0 (true) if queue has no elements */
int isEmptyQueue(const Queue q)
{
  if (q == NULL)
    queueError("isEmptyQueue: queue is null");
  return q->head == NULL && q->last == NULL;
}
```

It is an error to determine whether a *NULL* queue is empty.

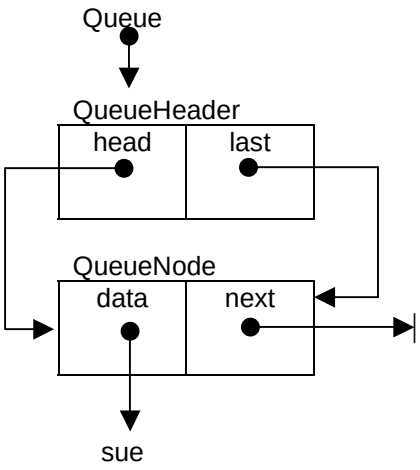
### 4.4.4 New Node

We create a new queue node and set both its members to *NULL*.

```
/* newQueueNode: returns a new, empty queue node */
QueueNode newQueueNode()
{
  QueueNode n = malloc(sizeof(struct QueueNodeStruct));
  if (n == NULL) queueError("newNode: out of memory");
  n->next = NULL;
  n->data = NULL;
  return n;
}
```

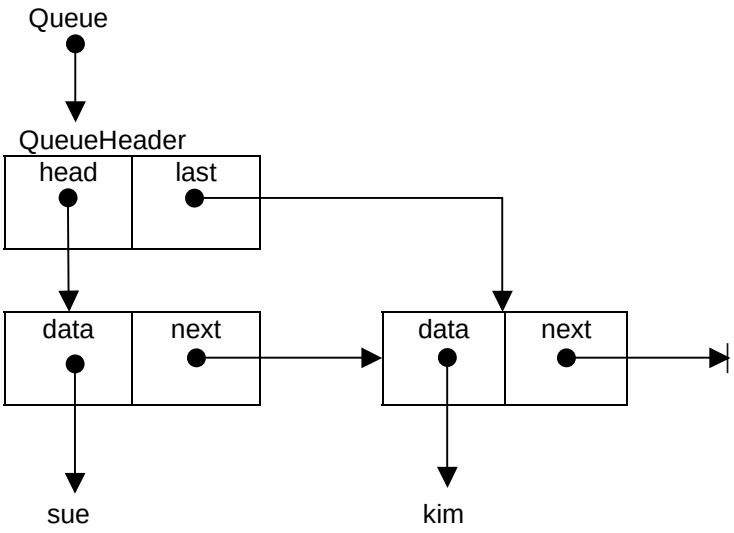
### 4.4.5 Join

We add an item to the end of a queue. We note that it is an error to attempt to add a data item to a null queue. We create a new node and set its data member to the given data item. But assigning values to the pointers *head* and *last* requires a little more thought.




If this is the first node in the queue, we update both *head* and *last* to point to it.

If this is not the first node in the queue we update *last* and the *next* member of the node that was previously pointed to by *last*.




```
/* joinQueue: adds data item to end of queue */
Queue joinQueue(void * const data, const Queue q)
{
  if (q == NULL)
    queueError("joinQueue: queue is null");
  Queue r = newQueue(q->duplicate);
  r->head = q->head;
  r->last = q->last;
  QueueNode n = newQueueNode();
  n->data = r->duplicate(data);
  if (isEmptyQueue(r)) {
    r->head = n;
    r->last = n;
  }
  else {
    r->last->next = n;
    r->last = n;
  }
  return r;
}
```

### 4.4.6 Head

*headOfqueue()* returns the item at the head of the queue. The queue remains unchanged. We rely on the programmer using our queue to know the item's data type and apply an appropriate cast.

```
/* headOfQueue: returns item at head of queue */
void *headOfQueue(const Queue q)
{
  if (q == NULL)
    queueError("headOfQueue: queue is null");
  if (isEmptyQueue(q))
    queueError("headOfQueue: queue is empty");
  return q->duplicate(q->head->data);
}
```

We cannot return a data item if the queue is empty. Nor can we return a data item if the queue does not exist.

### 4.4.7 Dispose Node

Releasing the memory occupied by a queue node is straightforward.

```
/* disposeQueueNode: releases memory occupied by QueueNode */
QueueNode disposeQueueNode(QueueNode *n)
{
  if (*n == NULL)
    queueError("disposeQueueNode: node is NULL");
  free((*n)->data);
  (*n)->data = NULL;
  free(*n);
  *n = NULL;
  return *n;
}
```

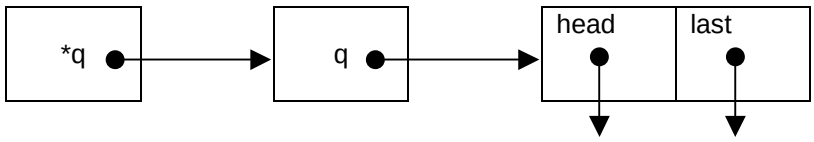
### 4.4.8 Leave

We remove an item from the front of the queue. We anchor the tail of the queue, dispose the node pointed to by *head*, and update *head* with the *tail*. If consequently *head* has the *NULL* value, then we have removed the last node, and so we update *last* with *NULL* also.

```
/* leaveQueue: removes data item from front of queue */
Queue leaveQueue(const Queue q)
{
  if (q == NULL)
    queueError("leaveQueue: queue is null");
  if (isEmptyQueue(q))
    queueError("leaveQueue: queue is empty");
  Queue r = newQueue(q->duplicate);
  QueueNode tail = q->head->next;
  disposeQueueNode(&q->head);
  r->head = tail;
  if (r->head == NULL)
    r->last = NULL;
  return r;
}
```

### 4.4.9 Dispose Queue

We release the memory occupied by queue. First, we ensure all nodes have been disposed of. Then we deallocate the header pointed to by the queue. Finally, we set queue to *NULL*. Since we are updating the queue pointer itself, we are obliged to work with a pointer to the queue.



```
/* disposeQueueNode: releases memory occupied by QueueNode */
QueueNode disposeQueueNode(QueueNode *n)
{
  if (*n == NULL)
    queueError("disposeQueueNode: node is NULL");
  free((*n)->data);
  (*n)->data = NULL;
  free(*n);
  *n = NULL;
  return *n;
}
```

### 4.4.10 Iterator

An iterator visits each element in a collection in turn.

We initialise a new iterator with the queue to be iterated over. *current*, which refers to the node currently being processed, is set to *NULL*.

```
/* newQueueIterator: returns a new iterator for the given queue */
QueueIterator newQueueIterator(const Queue q)
{
  QueueIterator iter = malloc(sizeof(struct QueueIteratorStruct));
  if (iter == NULL)
    queueError("newQueueIterator: out of memory");
  if (q == NULL)
    queueError("newQueueIterator: queue is NULL");
  iter->queue = q;
  iter->current = NULL;
  return iter;
}
```

We can go on to the next node only if there is one to go on to.

```
/* queueIteratorHasNext: returns 1 if there is an element not yet
   visited in the current iteration */
int queueIteratorHasNext(QueueIterator iter)
{
  if (iter->queue == NULL)
    queueError("queueIteratorHasNext: queue is NULL");
  if (isEmptyQueue(iter->queue))
    return 0;
  if (iter->current == NULL)
    return iter->queue->head != NULL;
  return iter->current->next != NULL;
}
```

We move on to the next node, but only if there is one to go on to, and return it's data iter.

```
/* queueIteratorNext: returns the next element in the queue */
void *queueIteratorNext(QueueIterator iter)
{
  if (!queueIteratorHasNext(iter))
    queueError("queueIteratorNext: no such element");
  if (iter->current == NULL)
    iter->current = iter->queue->head;
  else
    iter->current = iter->current->next;
  return iter->queue->duplicate(iter->current->data);
}
```

### 4.4.11 Implementation

The entire implementation is shown below.

```
/* queue.c */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "queue.h"

typedef struct QueueNodeStruct *QueueNode;

struct QueueNodeStruct {
  void *data;
  QueueNode next;
};

struct QueueHeader {
  QueueNode head;
  QueueNode last;
  Duplicate duplicate;
};

struct QueueIteratorStruct {
  Queue queue;
  QueueNode current;
};

/* queueError: reports queue errors, halts program execution */
int queueError(const char *error)
{
  printf("%s\n", error);
  exit(1);
}

/* newQueue: returns a new, empty queue */
Queue newQueue(Duplicate dup)
{
  Queue q = malloc(sizeof(struct QueueHeader));
  if (q == NULL)
    queueError("newQueue: out of memory");
  q->head = NULL;
  q->last = NULL;
  q->duplicate = dup;
  return q;
}

/* isEmptyQueue: returns 0 (true) if queue has no elements */
int isEmptyQueue(const Queue q)
{
  if (q == NULL)
    queueError("isEmptyQueue: queue is null");
  return q->head == NULL && q->last == NULL;
}
```

```
/* newQueueNode: returns a new, empty queue node */
QueueNode newQueueNode()
{
  QueueNode n = malloc(sizeof(struct QueueNodeStruct));
  if (n == NULL)
    queueError("newNode: out of memory");
  n->next = NULL;
  n->data = NULL;
  return n;
}

/* joinQueue: adds data item to end of queue */
Queue joinQueue(void * const data, const Queue q)
{
  if (q == NULL)
    queueError("joinQueue: queue is null");
  Queue r = newQueue(q->duplicate);
  r->head = q->head;
  r->last = q->last;
  QueueNode n = newQueueNode();
  n->data = r->duplicate(data);
  if (isEmptyQueue(r)) {
    r->head = n;
    r->last = n;
  }
  else {
    r->last->next = n;
    r->last = n;
  }
  return r;
}

/* headOfQueue: returns item at head of queue */
void *headOfQueue(const Queue q)
{
  if (q == NULL)
    queueError("headOfQueue: queue is null");
  if (isEmptyQueue(q))
    queueError("headOfQueue: queue is empty");
  return q->duplicate(q->head->data);
}

/* disposeQueueNode: releases memory occupied by QueueNode */
QueueNode disposeQueueNode(QueueNode *n)
{
  if (*n == NULL)
    queueError("disposeQueueNode: node is NULL");
  free((*n)->data);
  (*n)->data = NULL;
  free(*n);
  *n = NULL;
  return *n;
}
```

```
/* leaveQueue: removes data item from front of queue */
Queue leaveQueue(const Queue q)
{
  if (q == NULL)
    queueError("leaveQueue: queue is null");
  if (isEmptyQueue(q))
    queueError("leaveQueue: queue is empty");
  Queue r = newQueue(q->duplicate);
  QueueNode tail = q->head->next;
  disposeQueueNode(&q->head);
  r->head = tail;
  if (r->head == NULL)
    r->last = NULL;
  return r;
}

/* reverseQueue: reverses the order of element s in a queue */
Queue reverseQueue(const Queue q)
{
  if (q == NULL)
    queueError("copyQueue; queue is NULL");
  Queue r = newQueue(q->duplicate);
  QueueNode n, p;
  for (p = q->head; p != NULL; p = p->next) {
    n = newQueueNode();
    n->data = r->duplicate(p->data);
    n->next = r->head;
    r->head = n;
  }
  r->last = n;
  return r;

}

/* copyQueue: returns a duplicate copy of the given queue */
Queue copyQueue(const Queue q)
{
  Queue r = reverseQueue(q);
  return reverseQueue(r);
}

/* disposeQueue: deallocates memory occupied by queue */
Queue disposeQueue(Queue *q)
{
  if (*q == NULL)
    queueError("disposeQueue: queue is NULL");
  while (!isEmptyQueue(*q))
    *q = leaveQueue(*q);
  free(*q);
  *q = NULL;
  return *q;
}
```

```
/* newQueueIterator: returns a new iterator for the given queue */
QueueIterator newQueueIterator(const Queue q)
{
  QueueIterator iter = malloc(sizeof(struct QueueIteratorStruct));
  if (iter == NULL)
    queueError("newQueueIterator: out of memory");
  if (q == NULL)
    queueError("newQueueIterator: queue is NULL");
  iter->queue = q;
  iter->current = NULL;
  return iter;
}

/* disposeQueueIterator: deallocates the memory occupied by the
   iterator */
QueueIterator disposeQueueIterator(QueueIterator *iter)
{
  free(*iter);
  *iter = NULL;
  return *iter;
}

/* queueIteratorHasNext: returns 1 if there is an element not yet
   visited in the current iteration */
int queueIteratorHasNext(const QueueIterator iter)
{
  if (iter->queue == NULL)
    queueError("queueIteratorHasNext: queue is NULL");
  if (isEmptyQueue(iter->queue))
    return 0;
  if (iter->current == NULL)
    return iter->queue->head != NULL;
  return iter->current->next != NULL;
}

/* queueIteratorNext: returns the next element in the queue */
void *queueIteratorNext(QueueIterator iter)
{
  if (!queueIteratorHasNext(iter))
    queueError("queueIteratorNext: no such element");
  if (iter->current == NULL)
    iter->current = iter->queue->head;
  else
    iter->current = iter->current->next;
  return iter->queue->duplicate(iter->current->data);
}
```

### 4.5 Testing

We test each function looking for errors.

```
/* TestQueue */

#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

/* copyChar: returns a duplicate copy of its parameter */
char *copyChar(char *c)
{
  char *t = malloc(sizeof(char));
  t = c;
  return t;
}

int main()
{
  Queue q = newQueue((Duplicate)copyChar);
  printf("A new queue should be empty: ");
  if (isEmptyQueue(q))
    printf("it is.\n");
  else
    printf("it is not.\n");

  printf("Adding A, B C to the queue ... \n");
  char ch = 'A';
  char ch2 = 'B';
  char ch3 = 'C';
  q = joinQueue(&ch, q);
  q = joinQueue(&ch2, q);
  q = joinQueue(&ch3, q);

  printf("The queue should not be empty: ");
  if (!isEmptyQueue(q))
   printf("it is.\n");
  else
    printf("it is not.\n");

  printf("The item at the head of the queue should be A: ");
  if (*(char *)headOfQueue(q) == 'A')
    printf("it is.\n");
  else
    printf("it is not.\n");
  printf("\n");

  printf("Testing iterator: should see A B C: ");
  QueueIterator iter = newQueueIterator(q);
  while (queueIteratorHasNext(iter))
    printf("%c ", *(char *)queueIteratorNext(iter));
  printf("\n\n");

  printf("Testing copyQueue: should see A B C: ");
  Queue r = copyQueue(q);
  iter = newQueueIterator(r);
  while (queueIteratorHasNext(iter))
    printf("%c ", *(char *)queueIteratorNext(iter));
  printf("\n\n");
```

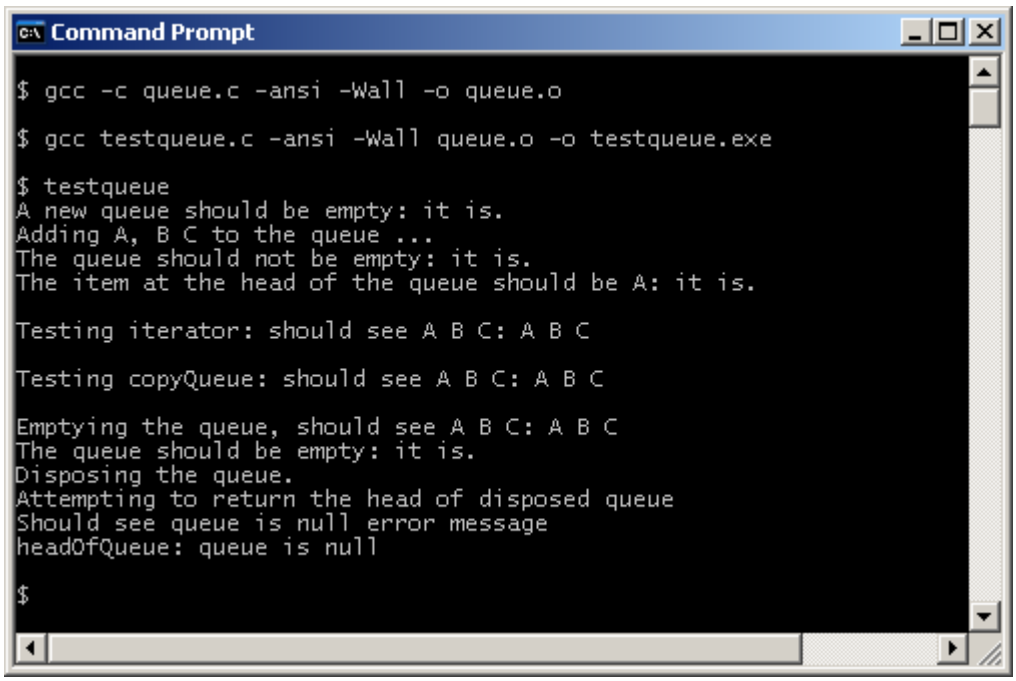
```
  printf("Emptying the queue, should see A B C: ");
  while (!isEmptyQueue(q)) {
    printf("%c ", *(char *)headOfQueue(q));
    q = leaveQueue(q);
  }
  printf("\n");

  printf("The queue should be empty: ");
  if (isEmptyQueue(q))
    printf("it is.\n");
  else
    printf("it is not\n");

  printf("Disposing the queue.\n");
  disposeQueue(&q);

  printf("Attempting to return the head of disposed queue\n");
  printf("Should see queue is null error message\n");
  printf("%c ", *(char *)headOfQueue(q));

  return 0;
}
```



## Exercise 4.1

1. Design, write and test a function that returns the length of a queue.

2. A priority queue might be found in a hospital's emergency department where patients are queued according to their medical need and not on their time of arrival.
    a. Draw a sequence of diagrams to show how data may be added to a priority queue. You could have a priority queue header with just one member, *head*. and you could consider three cases:
        i. when the queue is empty
        ii. when the given priority is less than the priority of the first node
        iii. when the given priority is not less than the priority of the first node.

    You might find it helpful to have two pointers that march along the queue, one pointing to the current node being visited, and one to the previous node just visited.
    b. Design, implement and test a priority queue.

## Bibliography

Kernighan B, Ritchie D, *The C Programming language*, Prentice Hall, 1988
Mark Williams Company, *ANSI C - A lexical Guide*, Prentice Hall 1988
Stubbs D and Webre N *Data Structures with Abstract Data Types and Modula-2*, Brooks/Cole 1987