PROGRAMMERING UGE 8

OPGAVE 1

Exercises

1)

Text answer Consider the following program for computing factorial numbers:

```
long fact(int n) {
  // precondition:
  assert(n >= 0);
  long f = 1;
  for (long i = 1; i <= n; ++i) {
    f = i * f;
  }
  return f;
}
```

Provide your answers to the following questions in a plain text file:

(a) How many arithmetic operations (`+, -, *, /`) are required to compute `fact(5)`?

Når n = 5 bliver i incremeret 5 gange og derfor går vi også gennem loop 5 gange Så vores operationer er:

I = 5 (+)

F = 5 (*)

(b) How many arithmetic operations (`+, -, *, /`) are required to compute `fact(n)` for any positive integer n?

Fordi vi har ++i og (i<= n) og f *= i, så har vi en funktion som siger

$$operations_{total} = 2n$$

OPGAVE 2

2)

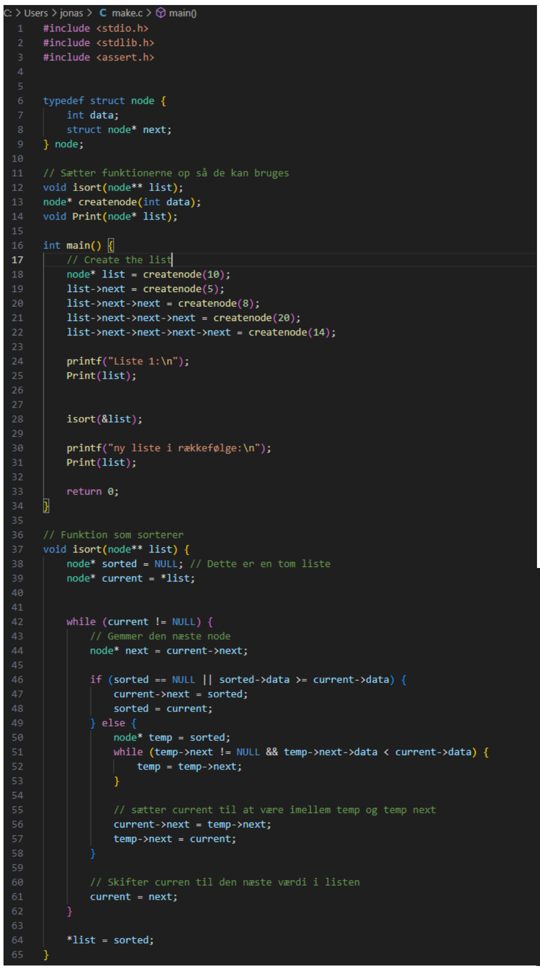
Code answer In the lecture we discussed the *insertion sort algorithm* implemented for sorting an array of integers. Implement an insertion sort function for a *singly linked list* of integers, so that the integers are sorted in the final linked list from smallest to largest. The function has the following signature:

```c
void isort(node* list);
```

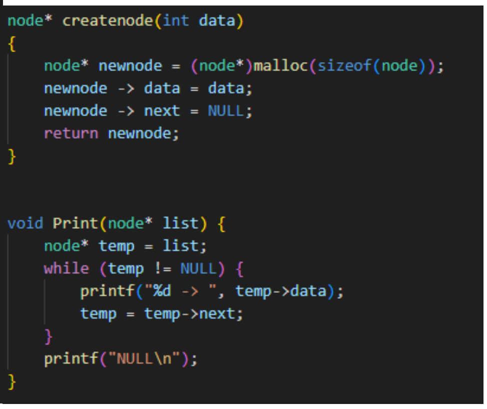
The function should sort the list **in-place**, that is no nodes should be created/allocated, but the pointers linking the nodes together should be changed such that they are in ascending sorted order.

The linked list was discussed in lectures six and seven and is defined as follows:

```c
typedef struct node {
  int data;
  struct node *next;
} node;
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>


typedef struct node {
    int data;
    struct node* next;
} node;

// Sætter funktionerne op så de kan bruges
void isort(node** list);
node* createnode(int data);
void Print(node* list);

int main() {
    // Create the list
    node* list = createnode(10);
    list->next = createnode(5);
    list->next->next = createnode(8);
    list->next->next->next = createnode(20);
    list->next->next->next->next = createnode(14);

    printf("Liste 1:\n");
    Print(list);


    isort(&list);

    printf("ny liste i rækkefølge:\n");
    Print(list);

    return 0;
}

// Funktion som sorterer
void isort(node** list) {
    node* sorted = NULL; // Dette er en tom liste
    node* current = *list;


    while (current != NULL) {
        // Gemmer den næste node
        node* next = current->next;

        if (sorted == NULL || sorted->data >= current->data) {
            current->next = sorted;
            sorted = current;
        } else {
            node* temp = sorted;
            while (temp->next != NULL && temp->next->data < current->data) {
                temp = temp->next;
            }

            // sætter current til at være imellem temp og temp next
            current->next = temp->next;
            temp->next = current;
        }

        // Skifter curren til den næste værdi i listen
        current = next;
    }

    *list = sorted;
}
```



```c
node* createnode(int data)
{
    node* newnode = (node*)malloc(sizeof(node));
    newnode -> data = data;
    newnode -> next = NULL;
    return newnode;
}


void Print(node* list) {
    node* temp = list;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

OPGAVE 3

3)

Code answer Queues maintain *first-in, first-out order (FIFO)*. This appears fairer than last-in, first-out, which is why lines at stores are organised as queues instead of stacks. Decks of playing cards can be modelled by queues since we deal the cards off the top of the deck and add them back in at the bottom. The abstract operations on a queue include:

- `enqueue(x, q)` - Insert item `x` at the back of queue `q`.
- `dequeue(q)` - Return (and remove) the front item from queue `q`.
- `init_queue(q)`, `full(q)`, `empty(q)` - Analogous to their namesake operations on stacks.

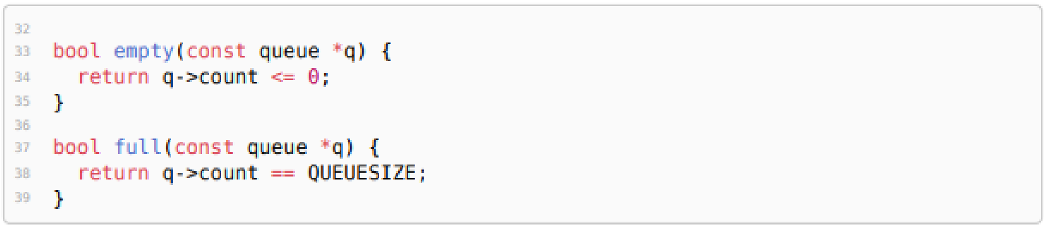
Queues are more difficult to implement than stacks because action happens at both ends. The *simplest* implementation uses an array, inserting new elements at one end and *moving* all remaining elements to fill the empty space created on each dequeue.

However, it is very wasteful to move all the elements on each dequeue. How can we do better? We can maintain indices to the first (head) and last (tail) elements in the array/queue and do all operations locally. There is no reason why we must explicitly clear previously used cells, although we leave a trail of garbage behind the previously dequeued items.

Circular queues let us reuse this empty space. Note that the pointer to the front of the list is always *behind* the back pointer! When the queue is full, the two indices will point to neighbouring or identical elements. There are several possible ways to adjust the indices for circular queues. *All are tricky!* The easiest solution distinguishes full from empty by maintaining a count of how many elements exist in the queue.

Below is an implementation of a circular queue.

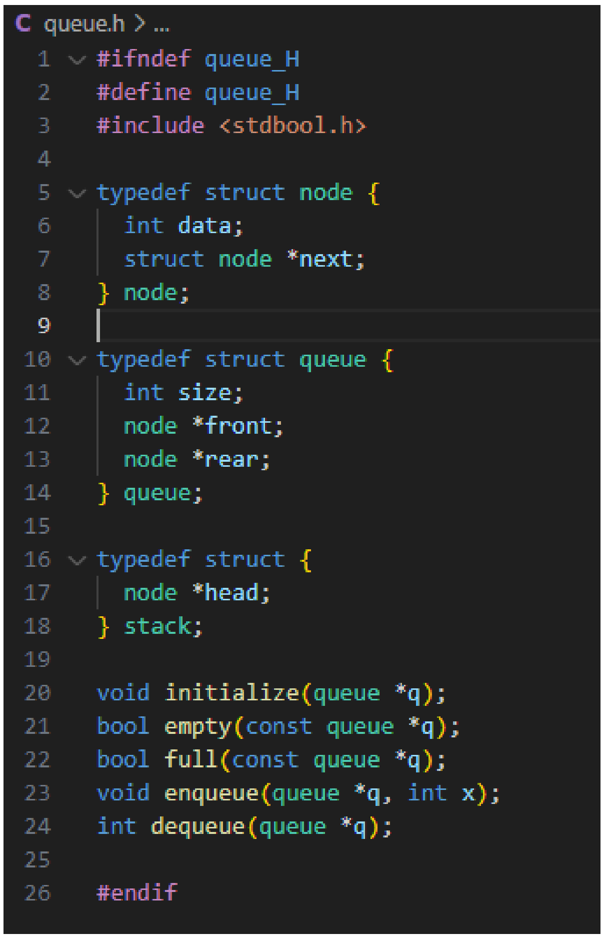
```c
#include <stdbool.h>

#define QUEUESIZE 10
typedef struct {
  int q[QUEUESIZE];     // body of queue
  int first;            // position of first element
  int last;             // position of last element
  int count;            // number of queue elements
} queue;

void initialize(queue *q) {
  q->first = 0;
  q->last = QUEUESIZE - 1;
  q->count = 0;
}

void enqueue(queue *q, int x) {
  assert(q->count < QUEUESIZE);
  q->last = (q->last + 1) % QUEUESIZE;
  q->q[q->last] = x;
  q->count = q->count + 1;
}

int dequeue(queue *q) {
  int x;
  assert(q->count > 0);
  x = q->q[q->first];
  q->first = (q->first + 1) % QUEUESIZE;
  q->count = q->count - 1;
  return(x);
}
```



```c
bool empty(const queue *q) {
  return q->count <= 0;
}

bool full(const queue *q) {
  return q->count == QUEUESIZE;
}
```

(a) Implement a queue based on *singly-linked lists* as discussed in the lecture. That is, implement the five functions mentioned above.

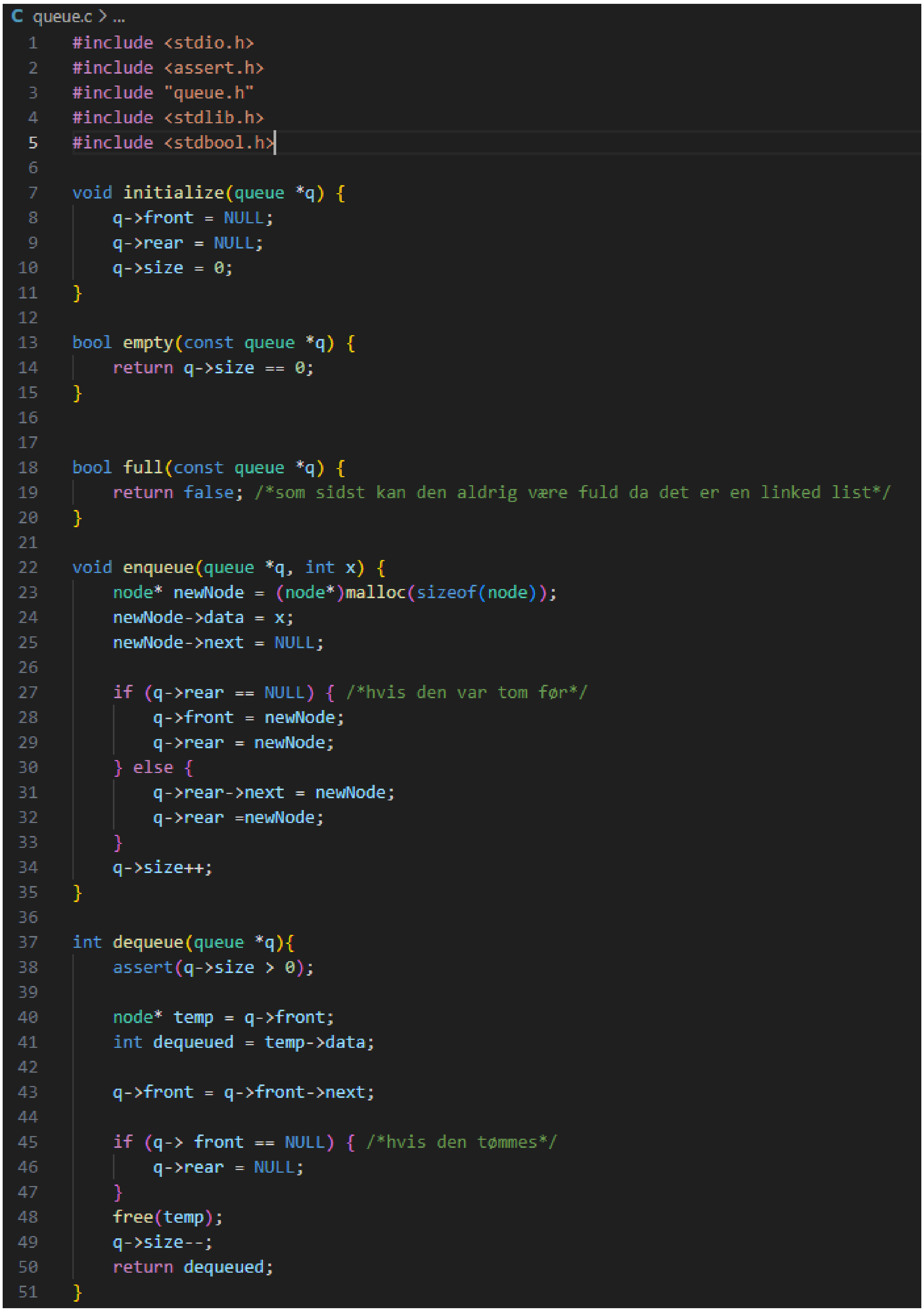
Header filen:



queue.h > ...

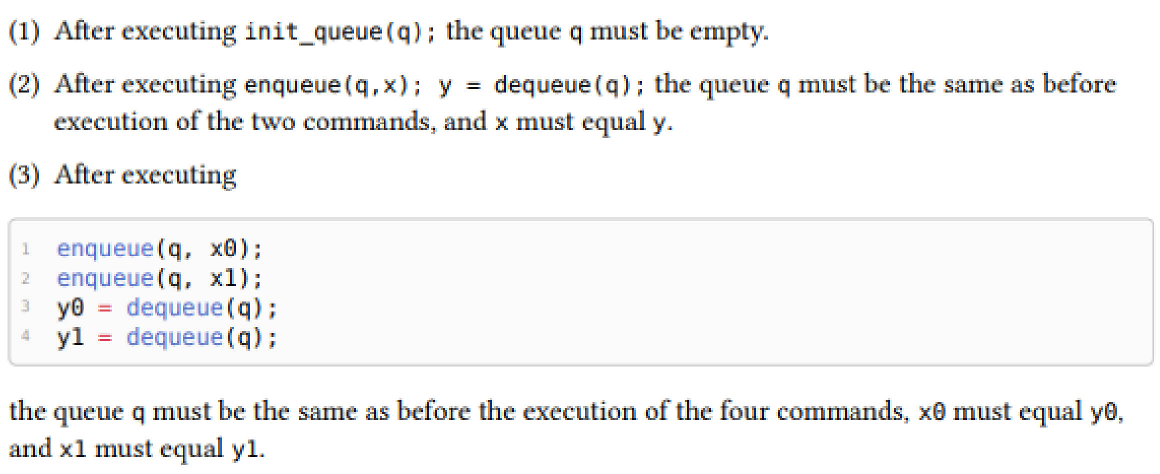
```c
#ifndef queue_H
#define queue_H
#include <stdbool.h>

typedef struct node {
    int data;
    struct node *next;
} node;

typedef struct queue {
    int size;
    node *front;
    node *rear;
} queue;

typedef struct {
    node *head;
} stack;

void initialize(queue *q);
bool empty(const queue *q);
bool full(const queue *q);
void enqueue(queue *q, int x);
int dequeue(queue *q);

#endif
```

C filen med funktioner:


queue.c > ...

```c
#include <stdio.h>
#include <assert.h>
#include "queue.h"
#include <stdlib.h>
#include <stdbool.h>

void initialize(queue *q) {
    q->front = NULL;
    q->rear = NULL;
    q->size = 0;
}

bool empty(const queue *q) {
    return q->size == 0;
}


bool full(const queue *q) {
    return false; /*som sidst kan den aldrig være fuld da det er en linked list*/
}

void enqueue(queue *q, int x) {
    node* newNode = (node*)malloc(sizeof(node));
    newNode->data = x;
    newNode->next = NULL;

    if (q->rear == NULL) { /*hvis den var tom før*/
        q->front = newNode;
        q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear =newNode;
    }
    q->size++;
}

int dequeue(queue *q){
    assert(q->size > 0);

    node* temp = q->front;
    int dequeued = temp->data;

    q->front = q->front->next;

    if (q-> front == NULL) { /*hvis den tømmes*/
        q->rear = NULL;
    }
    free(temp);
    q->size--;
    return dequeued;
}
```

(b) Test your implementation. **Create a new file**, where you include your queue library header file. You should expect the following "laws" to hold for any implementation of a queue. **Hint:** you could enforce these conditions using `assert` statements:



(1) After executing `init_queue(q);` the queue `q` must be empty.

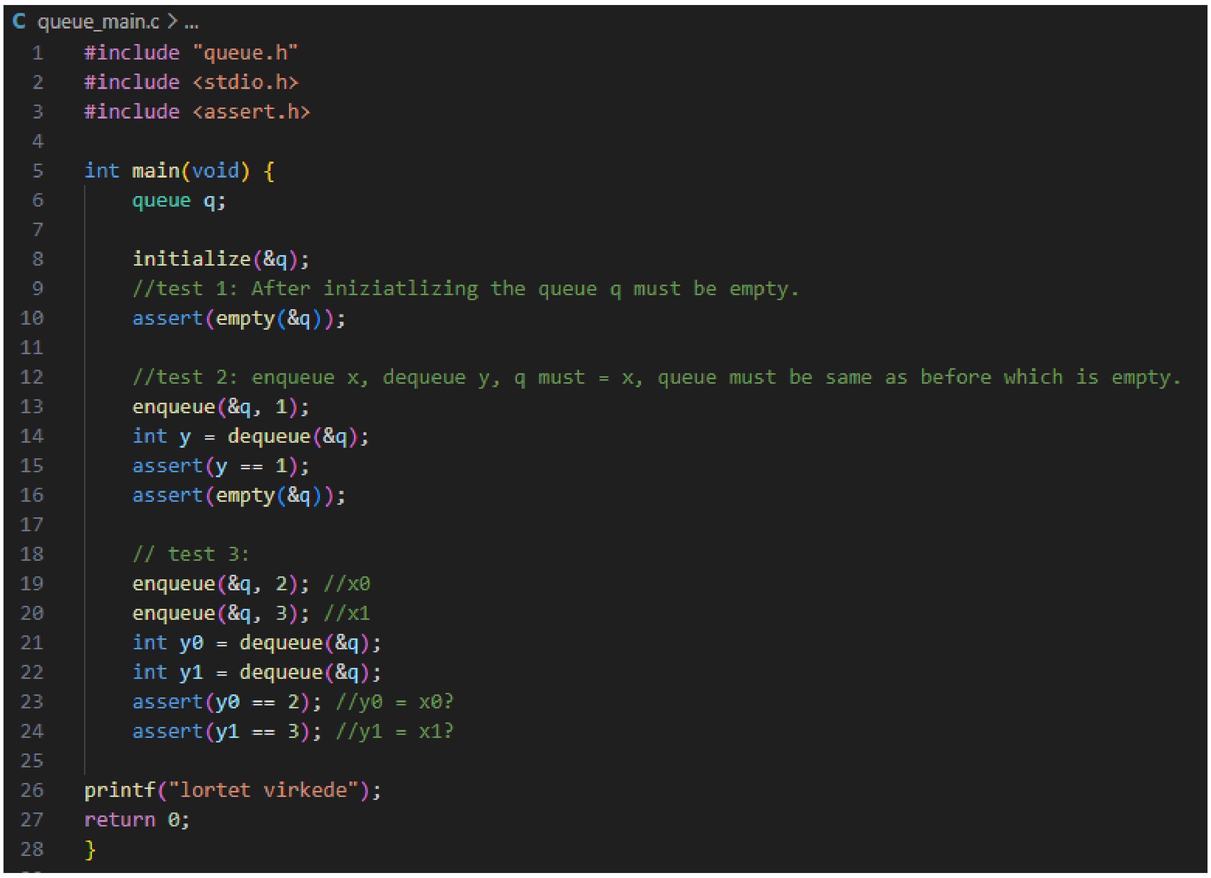
(2) After executing `enqueue(q,x); y = dequeue(q);` the queue `q` must be the same as before execution of the two commands, and `x` must equal `y`.

(3) After executing

```
enqueue(q, x0);
enqueue(q, x1);
y0 = dequeue(q);
y1 = dequeue(q);
```

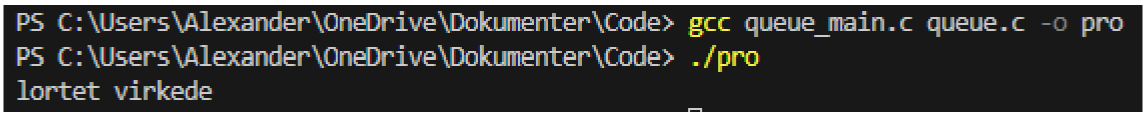
the queue `q` must be the same as before the execution of the four commands, `x0` must equal `y0`, and `x1` must equal `y1`.

Main filen:



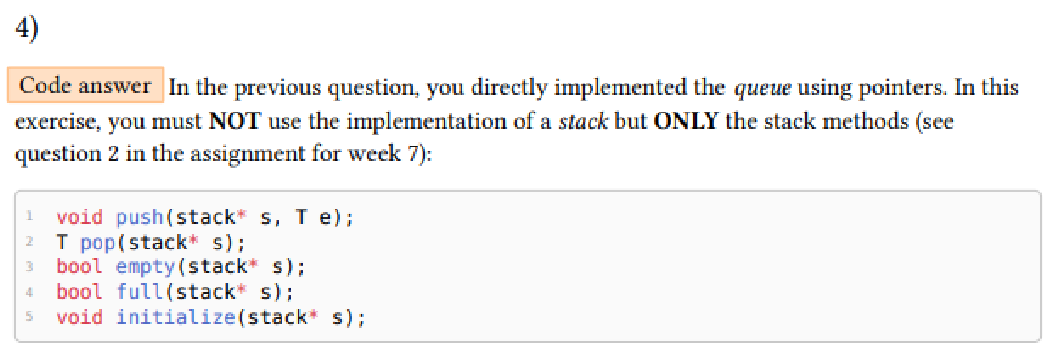
```c
queue_main.c > ...
#include "queue.h"
#include <stdio.h>
#include <assert.h>

int main(void) {
    queue q;

    initialize(&q);
    //test 1: After iniziatlizing the queue q must be empty.
    assert(empty(&q));

    //test 2: enqueue x, dequeue y, q must = x, queue must be same as before which is empty.
    enqueue(&q, 1);
    int y = dequeue(&q);
    assert(y == 1);
    assert(empty(&q));

    // test 3:
    enqueue(&q, 2); //x0
    enqueue(&q, 3); //x1
    int y0 = dequeue(&q);
    int y1 = dequeue(&q);
    assert(y0 == 2); //y0 = x0?
    assert(y1 == 3); //y1 = x1?

printf("lortet virkede");
return 0;
}
```

Kører man det ser man at den kører hele vejen til bunds uden at blive fanget af



```
PS C:\Users\Alexander\OneDrive\Dokumenter\Code> gcc queue_main.c queue.c -o pro
PS C:\Users\Alexander\OneDrive\Dokumenter\Code> ./pro
lortet virkede
```

OPGAVE 4

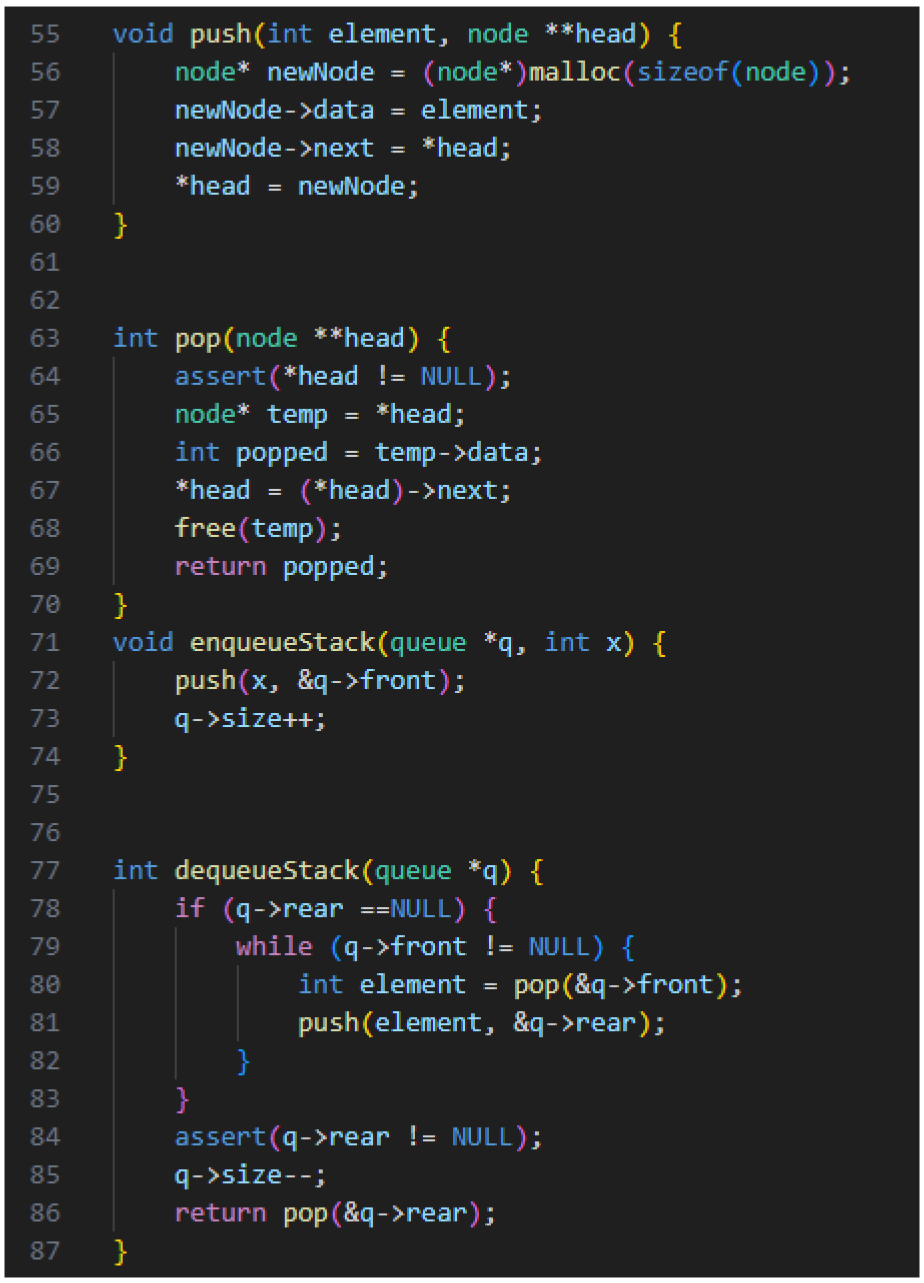


4)

Code answer In the previous question, you directly implemented the *queue* using pointers. In this exercise, you must **NOT** use the implementation of a *stack* but **ONLY** the stack methods (see question 2 in the assignment for week 7):

```c
void push(stack* s, T e);
T pop(stack* s);
bool empty(stack* s);
bool full(stack* s);
void initialize(stack* s);
```

The functions:



```c
void push(int element, node **head) {
    node* newNode = (node*)malloc(sizeof(node));
    newNode->data = element;
    newNode->next = *head;
    *head = newNode;
}


int pop(node **head) {
    assert(*head != NULL);
    node* temp = *head;
    int popped = temp->data;
    *head = (*head)->next;
    free(temp);
    return popped;
}
void enqueueStack(queue *q, int x) {
    push(x, &q->front);
    q->size++;
}


int dequeueStack(queue *q) {
    if (q->rear ==NULL) {
        while (q->front != NULL) {
            int element = pop(&q->front);
            push(element, &q->rear);
        }
    }
    assert(q->rear != NULL);
    q->size--;
    return pop(&q->rear);
}
```