Programmering aflevering 8

Exercise 1

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)`?

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

a) *i* skal være mindre eller lig *n*, hvilket vil sige, at *i* kører 5 gange, da *n=5.*
Inde i løkken bliver *f* og *i* ganget med hinanden og der bliver lagt 1 til *i* for hver af de 5 runder:
$5 + 5 = 10$
Det kræver 10 “arithmetic operations” at beregne fact(5).

b) Med udgangspunkt i formlen fra (a) beregnes fact(n) ved $n + n$.

Exercise 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 "insertion_sort.h"
#include <stdio.h>
#include "node.h"

node* isort(node *list) {
    node *sorted_list = NULL; //Initialize sorted_list pointer to point to empty sorted list
    node *current = list; //Initialize current pointer to point to head of the list

    while (current != NULL) { //Go through all nodes
        node *nxt = current->next; //Store next node in list before modifying current's next pointer
        //If sorted list is empty or data in sorted list is not lower than current data
        if (sorted_list == NULL || sorted_list->data >= current->data) {
            current->next = sorted_list; //Current's next-pointer points to first node in sorted list
            sorted_list = current; //Update sorted list to point to current
        }
        //Sorted list is not empty or data in sorted list is lower than current data
        else {
            node *temp = sorted_list; //Initialize pointer temp to point to head in list
            while (temp->next != NULL && temp->next->data < current->data) {
                //Find correct position to insert current in sorted list
                temp = temp->next; //Move to the next node
            }
            //Insert current between temp and temp->next
            current->next = temp->next; //Current points to the same node temp points to
            temp->next = current; //Temp points to current node
        }
        current = nxt; //Move on to the next node in list
    }
    return sorted_list;
}
```

Exercise 3

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

(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`.

a)

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

// Opgave 3
void initialize(queue *q) {
    //Initialize queue
    q->front = NULL; //Set front pointer (pointer to first node) to NULL
    q->rear = NULL; //Set rear pointer (pointer to last node) to NULL
    q->size = 0; //Start off with an empty queue
}

bool empty(const queue *q) {
    //Chck if queue is empty
    return q->size == 0; //Return true if size is 0
}

bool full(const queue *q) {
    //Check if queue is empty
    return false; //Queue can't be full as it is based on a singly-linked list
}

void enqueue(queue *q, int x) {
    //Add an element to the queue
    node *new_node = (node *)malloc(sizeof(node)); //Allocate memory for a new node
    if (new_node == NULL) {
        printf("Memory allocation failed\n"); //Print error message if allocation fails
        return;
    }
    new_node->data = x; //Set data of new node
    new_node->next = NULL; //Set the next pointer of new node to NULL (new node is at the end of the queue)

```

b)

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

#include "queue.h"

int main() {
    {
    // Opgave 3 tests
    queue q;
    initialize(&q);

    // Test 1: After init, queue must be empty
    assert(empty(&q));
    assert(q.front == NULL);
    assert(q.rear == NULL);
    assert(q.size == 0);

    // Test 2: Enqueue and dequeue must return the same value
    int x = 10;
    enqueue(&q, x);
    int y = dequeue(&q);
    assert(empty(&q));
    assert(q.front == NULL);
    assert(q.rear == NULL);
    assert(q.size == 0);
    assert(x == y);

    // Test 3: Enqueue multiple elements and ensure they dequeue in the same order
    int x0 = 20, x1 = 30;
    enqueue(&q, x0);
    enqueue(&q, x1);
    int y0 = dequeue(&q);
    int y1 = dequeue(&q);
    assert(empty(&q));
    assert(q.front == NULL);
    assert(q.rear == NULL);
    assert(q.size == 0);
    assert(x0 == y0);
    assert(x1 == y1);

    printf("All tests passed!\n");
    return 0;

}
```

Exercise 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);
```

```c
// Opgave 4
void push(int element, node **head) {
    node *new_node = (node *)malloc(sizeof(node)); // Allocate memory for a new node
    if (new_node == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    new_node->data = element; // Set data of the new node
    new_node->next = *head; // Set the next pointer of the new node to the current head
    *head = new_node; // Update head to point to the new node
}

int pop(node **head) {
    if (*head == NULL) {
        printf("Stack is empty\n");
        return -1; // Return a sentinel value to indicate the stack is empty
    }
    int value = (*head)->data; // Get data from the current head
    node *temp = *head; // Temporarily store the current head
    *head = (*head)->next; // Move head to the next node
    free(temp); // Free memory of the old head node
    return value; // Return the data of the popped node
}

void enqueueStack(queue *q, int x) {
    push(x, &(q->front)); // Push element to the front stack
    q->size++; // Increment size of the queue
}

```

```c
int dequeueStack(queue *q) {
    if (q->rear == NULL) {
        while (q->front != NULL) {
            int element = pop(&(q->front)); // Pop elements from front stack
            push(element, &(q->rear)); // Push them to the rear stack
        }
    }
    int value = pop(&(q->rear)); // Pop the top element from rear stack
    if (value != -1) {
        q->size--; // Decrement size of the queue
    }
    return value; // Return the dequeued value
}
```