# Assignment 8 Programmering - Frederik

## 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)`?

a)

The short answer is running the loop 5 times, so 5 multiplications

We arrive at that answer when you think about the loop. The for loop run “i = 1” until it becomes “i = 5”. Since we have the operation “f = i * f” and ++i for each loop. It will have to go through the loop 5 times. To compute fact(5) we need 5 multiplications.

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

b)

If we look at the loop again it starts with “i = 1” and goes until “i <= n”. Which also means it runs n times. So, to compute fact(n) it must do, n multiplications.

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:

```
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:

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

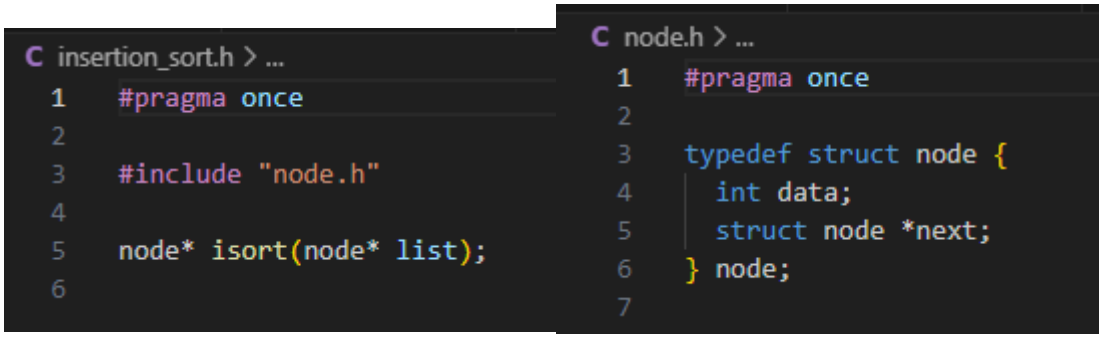
ass2.c > isort(node *)

```c
#include "insertion_sort.h"

void isort(node* list) {
node* sorted = NULL; // When pointer is sorted = NULL

node* current = list; // Goes through the list
while (current != NULL) {
    node* next = current->next; // Saves next node

    if (sorted == NULL || current->data < sorted->data) { // Current gets put into sorted list
        current->next = sorted; // Gets puts in the beginning of the sorted list
        sorted = current; // Says list is currently sorted
    } else {
        node* temp = sorted; // Finds a temporary position to store current node
        while (temp->next != NULL && temp->next->data < current->data) {
            temp = temp->next; // The temporary node becomes the next node
        }
        // Current node gets put after the temporary node
        current->next = temp->next;
        temp->next = current;
    }
    current = next; // Moves to the ne
  }
    *list = *sorted; // Now the start of the list points to the sorted list
}
```

a)

Let's break down some code

Comments in the program says most of it, but I digress. We include "insertion_sort.h" since it includes where it includes "node.h" and has these following lines of code;



insertion_sort.h > ...

```
#pragma once

#include "node.h"

node* isort(node* list);

```

node.h > ...

```
#pragma once

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

```

“node.h” has defined a struct for node, which we use in “insertion_sort.h” with the isort function… which is sort for “import sort” so it’s just a sorting function.

We then start by saying the node pointer “node*” is sorted when = NULL

We set the pointer node for current to = list so we can look through the list. We have the first while loop which says when current ISN’T = NULL then the pointer for current updates to next.

The if statement says when sorted = NULL OR the pointer current to data is smaller “<” than the pointer for sorted to data THEN the next node becomes sorted and sorted becomes current.

ELSE, we make a temporary node pointer set to = sorted

In the second while loop when the temporary node ISN’T = NULL AND current data must be > than the temporary nodes next data so we can say the temp node is the pointer to next.

Outside the second while loop, current points to next, becomes temp pointing to next so next becomes current.

At the bottom of the while loop current becomes next and therefore, we can say the pointer for list becomes the pointer for sorted. I.e. now the list should be sorted.

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

queue.c > ...

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

// Opgave 3
typedef struct node {  // Node structure for the lists
    int data;
    struct node* next;
} node;

typedef struct { // Queue structure
    node* front;
    node* rear;
    int count;
} queue;

void initialize(queue *q) { // Starts or initizlizes queue
    q->front = NULL;
    q->rear = NULL;
    q->count = 0;
}


bool empty(const queue *q) { // Checks if queue is empty
    return q->count == 0;
}

bool full(const queue *q) { // Checks if queue is full
    return false;
}


void enqueue(queue *q, int x) { // Adds element at the end of queue, for last
    node* new = (node*)malloc(sizeof(node));
    new->data = x;
    new->next = NULL;

    if (q->rear == NULL) {
        q->front = new; // If queue is empty first and last becomes the same
        q->rear = new;
    } else {
        q->rear->next = new; // Else new node is updated to the last as new
        q->rear = new;
    }
    q->count++; // count++ because we add an element
}
```

```c
int dequeue(queue *q) { // Removes the first element in queue
    if (empty(q)) {
        printf("Queue is empty\n");
        exit(1); // If queue is empty, close program
    }

    int x = q->front->data; // First nodes data gets stored in x
    node* temp = q->front; // Saves a temporary pointer "q->first", temporary because this is the one to be removed
    q->front = q->front->next; // Moves first node to next node, here it is practically already removed from the list
    free(temp); // Removes the memory of the temporary pointer

    if (q->front == NULL) { // If queue is empty after removal we update to NULL
        q->rear = NULL;
    }

    q->count--; // count-- because we remove an element
    return x; // we return x, because the temporary pointer is stored there, the one we want to remove
}
```

Here we include “queue.h” which looks like this (is used for opgave 3 then 4)

queue.h > dequeueStack(queue2 *)

```c
#pragma once

#include <stdbool.h>

#include "node.h"

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);
void push(int element, node **head);
int pop(node **head);
void enqueueStack(queue2 *q, int x);
int dequeueStack(queue2 *q);
```

We set up 2 structs one for queue and one for node for a baseline. Looking back we didn’t need to make these 2 structs, since they are already in “queue.h” and may cause syntax errors or worse.

Since we are asked to implement the 5 functions we start with “void initialize”, we have queue and the pointer for queue “*q”. We start by making the pointers for front and rear = NULL and count 0, since we should start from 0 and anything else would sense… in this case.

Bool empty checks to see if the queue is empty, since we say count = 0, it should return 0.

We keep the already set bool full line, since anything else wouldn’t make sense. There are in general no size limits in linked lists (if there was something to do with the code, please make a note so I can attempt something else).-

For void enqueue we want to add a new node at the end of the queue, which we call “new”, for new node. We set the node pointer for new to = a malloc size so we can say the pointer for new data = x, and that the pointer for new next = NULL.

We make an if statement that says if rear = NULL then front = new and rear = new. So that if the queue is empty the first and last value is the same. ELSE the rear pointer for next becomes new i.e. rear = new, so the value for last becomes new. We then add 1 to count, since we have 1 value in the queue.

The last function we want implemented is int dequeue which removes the first element in the queue. We start by making a fail state for IF the queue is empty and the value for count already is 0. If it’s empty the program says, “Queue is empty” and exits the program.

We stored the first nodes data in x, then make a temporary node for the front pointer. We then set the front node to be the next node, effectively removing it from the list, so when we use the free temp function and remove the temporary pointer, nothing is lost, essentially.

We also add a if loop that says if the front pointer = NULL the rear pointer should also = NULL.

When we remove an element we say count-- to make count go from 1 to 0. And return x.

(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`.

queuetest.c > main()

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

#include "queue.h"

int main() {
    queue q;
    int x0 = 5, x1 = 10;
    int y0, y1;

// Test 1, when initialize is executed queue should be empty or else it wont run
    initialize(&q);
    assert(empty(&q) == true); // Asserts queue should be empty

// Test 2, when executed q should be the same as before executed, hence the x = y
    enqueue(&q, x0); // First element enqueued
    y0 = dequeue(&q); // y0 set to be value of dequeue or x0
    assert(empty(&q) == true); // Same as test 1
    assert(x0 == y0); // x must be the same as y

// Test 3, like Test 2. Though here we have, x0 = y0 AND x1 = y1
    enqueue(&q, x0); // First element enqueued
    enqueue(&q, x1); // Second element enqueued
    y0 = dequeue(&q); // y0 set to be value of first dequeue or x0
    y1 = dequeue(&q); // y1 set to be value of second dequeue or x1

// After this queue goes back to being empty, how it started, or its original state
    assert(empty(&q) == true);
    assert(x0 == y0);
    assert (x1 == y1);

    printf("Test have passed, lawful abiding citizen\n"); // Congratulation prompt for passing all tests

    return 0;
}
```

In test 1 we make sure to assert the queue should be empty.

In test 2 we assert that x0 = y0, but first we add x0 (5) then remove y0 which should also be 5, therefore we get 5-5 = 0 and the queue is empty again.

In test 3 we also assert that x1 = y1

If everything went right and the tests have passed we should get the message “Test have passed”

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
// Opgave 4

typedef struct node { // Same struct as in opgave 3
    int data;
    struct node* next;
} node;

typedef struct queue2 { // Here we define our nonstacks a node pointers
    node* notastack1;
    node* notastack2;
} queue2;

void initializeQueue(queue2 *q) { // Function to start a queue by setting our values to NULL
    q->notastack1 = NULL;
    q->notastack2 = NULL;
}

void push(int element, node **head) { // Function that pushes an element on our stack
    node* new = (node*)malloc(sizeof(node));
    if (!new) {
        printf("Malloc problems, exiting...\n"); // If there are memory problems with the malloc function the program should exit
        exit(1);
    }
    new->data = element; // The pused node becomes the integer i.e. the element
    new->next = *head; // New node becomes the pointer to head
    *head = new; // The pointer points to the new element we have pushed
}

int pop(node **head) { // Function that pops an element from the stack
    if (*head == NULL) {
        printf("Stack problems, exiting...\n"); // If there are problems with the stack the program should exit
        exit(1);
    }
    int result = (*head)->data; // The popped element gets stored as result
    node* temp = *head; // A temporary node is set as the pointer for head
    free(temp); // The temp node is removed
    return result;
}
```

```c
void enqueueStack(queue2 *q, int x) { // Function to enqueue a stack
    push(x, &q->notastack1); // Pushes pointer to notastack1
}

int dequeueStack(queue2 *q) { // Function to dequeue a stack
    if (q->notastack2 == NULL && q->notastack1 == NULL) {
        printf("There is something wrong with the nonstacks\n"); // Error code if both values are NULL
        exit(1);
    }

    if (q->notastack2 == NULL) {
        while (q->notastack1 != NULL) { // If notastack 2 is NULL but notastack1 isnt we pop notastack1 and push notastack2
            int element = pop(&q->notastack1);
            push(element, &q->notastack2);
        }
    }
    return pop(&q->notastack2); // We get returned the popped value
}

int Check_If_Empty(const queue2 *q) { // Function to check if queue is empty
    return q->notastack1 == NULL && q->notastack2 == NULL;
}
```

We again make some definitions on struct that may not be necessary in the great picture, but works in a vacuum. Hence queue is named queue2 here, since queue was used for opgave 3.

We again set the pointer values for notastack1,2 = NULL when we initialize the queue

We then in void push (push, pushes and element in the stack to the front/top) again use the same malloc line from earlier, IF new is 0 then we get an error message and the program exists.

The new pointer to data becomes the element, and the new pointer to next becomes the head pointer, *head, and that pointer becomes = new

Int pop removes an element from the stack, by POPping it like a balloon. Here we make a fail safe for if the pointer for head, *head = NULL then we exit the program. Since we can't pop nothing.

Here result becomes an integer and result = *head->data, the popped elements gets stored as result here. Again we make a temporary node and make it = *head, then frees  the temporary node so we only have the stored value of data left which is returned in result.

Void enqueueStack just adds the x value to the pushed pointer for notastack1.

Int dequeueStack removes an element. But IF both notastack1,2 = NULL then we get a failstate and exit once again. The second if loop says I notastack1 ISN'T = NULL then we pop the pointer to notastack1 and pushes the element and pointer to notastack2 to the top, we returned the popped value.

Then lastly we make the function int Check_If_Empty, which should return notastack1,2 = NULL

(In a vacuum all opgaver should work, but in the way its all put together in visual studio code idk how well it works and how to make it work properly...)