Programmering afl. 8

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) Den lægger til 5 gange og ganger 5 gange så der er 10 arithmetic operations.

b) Der er n + n arithmetic operations for alle positive integers.

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;
```

```
#include "insertion_sort.h"

node* isort(node *list) {
if(!list || !list->next) {
    return list; //If theres no element or only one element the list is already sorted
}

    node *sorted = NULL; //Keeps track of sorted part of the list
    node *current = list; //Used to move around the unsorted part of the list
    while (current != NULL) { //Stops when current becomes NULL which happens at the end of the list
        node *next = current->next; //current->next is stored in next to not lose track of the list
        if (!sorted || current->data <= sorted->data) {
            current->next = sorted;//If the current is smaller than all previously sorted elements the current becomes the new head
            sorted = current;
        } else {
            node *temp = sorted;//Temps is used to move around the sorted part of the list
            while (temp->next != NULL && temp->next->data < current->data) { /*Temp is moved until temp is NULL or
            when the next node has a greater value current*/
                temp = temp->next;
            }
            current->next = temp->next;//Current points to the node temps was pointing to
            temp->next = current;//temp points to current. Inserting current into the sorted part of the list
        }
        current = next;//Moves to the next node in the unsorted part of the list
    }
    return sorted;
}
```

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.

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

```
4   // Opgave 3
5   void initialize(queue *q) {
6   q->front = NULL;
7   q->rear = NULL;
8   q->QUEUESIZE = 0; //No elements in the queue
9   }
10
11  bool empty(const queue *q) {
12      return q->QUEUESIZE == 0;//Returns true if queue is empty
13  }
14
15  bool full(const queue *q) {
16      node *temp = (node *)malloc(sizeof(node));
17      if(temp == NULL){ //Like in the linked list, if malloc returns NULL no more memory can be allocated
18      return true;
19  }
20  free(temp);
21  return false;
22  }
23
24  void enqueue(queue *q, int x) {
25  node *new = (node *)malloc(sizeof(node));
26  if (new == NULL){
27      printf("Queue is full. \n")
28      return;
29  }
30  new->data = x;
31  new->next = NULL; //new's data is x and new is pointing to the end of the queue
32
33  if(q->rear == NULL){ //If queue is empty both front and rear become new
34      q->front = new;
35      q->rear = new;
36  } else {
37      q->rear->next = new;
```

```
    q->rear = new; //New is added to the back
}
q->size++; //The queue becomes one bigger
}


int dequeue(queue *q) {
   if (q-front == NULL) {
    printf("Queue is empty\n")
    return -1;
   }
   node *temp = q->front;
   int value = temp->data;

   q->front = q->front->next; //Front is moved to the next node in line
    if (q->front == NULL){
        q->rear = NULL; //If front is NULL the queue is empty and rear is set to NULL
    }
    free(temp);
    q->size--;
    return value;
}

```

(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`.

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

#include "queue.h"
#include "queue.c"
int main() {
queue q;
initialize(&q);
assert(empty(&q) == true); //After initializing the queue must be empty
assert(q.front == NULL);
assert(q.rear == NULL);
assert(q.size == 0);
printf("Test one passed.\n");

int x = 10;
enqueue(&q,x);
int y = dequeue(&q);

assert(empty(&q) == true);
assert(q.front == NULL);
assert(q.rear == NULL);
assert(q.size == 0);
assert(x==y);

printf("Test two passed.\n");
//x = 10 and y = 10. The queue also looks the same as before the test.

```

```c
    int x0 = 20;
    int x1 = 30;

    enqueue(&q, x0);
    enqueue(&q, x1);
    int y0 = dequeue(&q);
    int y1 = dequeue(&q);
    assert(empty(&q) == true);
    assert(q.front == NULL);
    assert(q.rear == NULL);
    assert(q.size == 0);
    assert(x0==y0);
    assert(x1==y1);

    //x0 = 20, x1 = 30, y0 = 20 and y1 = 30. The queue looks the same as before all four commands.
    printf("All tests passed.\n");

    return 0;

    }
```

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):

Push og pop funktioner

```c
// Opgave 4
void push(int element, node **head) {//tilføjer element til toppen af vores stak
    node *newNode = (node *)malloc(sizeof(node));
    if (newNode == NULL)
        {
            printf("Memory allocation failed\n");
            return;
        }
    newNode->data = element;
    newNode->next = *head;//sætter det nye element på toppen af stakken
    *head = newNode;

}

int pop(node **head) {//fjerner/returnerer element fra toppen af stak
    if (*head == NULL)
    {
        printf("stack is empty\n");
        return -1;//returner fejl hvis stak er tom
    }
    node *temp = *head;
    int value = temp->data;
    *head = (*head)->next;
    free(temp);
    return value;
}
```

Funktioner tilsvarende til enqueue og dequeue fra opg 3 bare med brug af stack metoder.

```c
//Tilføjer element til rear, som er den stak vi bruger
void enqueueStack(queue *q, int x) {//som enqueue men bruger push til at tilføje element
    push(x, &q->rear);
}

//fjerner første tilføjede element fra køen
int dequeueStack(queue *q) {//som dequeue men bruger pop til at fjerne element
        if (q->front == NULL)
        {
            while (q->rear != NULL)//flytter elementer fra stack1 til stack2 hvis den er tom
            {
                int element = pop(&q->rear);
                push(element, &q->front);
            }
        }

    if (q->front == NULL)
    {
        printf("queue is empty\n");
        return -1;
    }

    return pop(&q->front);//fjerner første tilføjede element
}
```

Vi tester koden med:

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

#include "queue.h"
#include "queue.c"
int main() {
    queue q;
    initialize(&q);
    assert(q.front == NULL);
    assert(q.rear == NULL);
    printf("Test one passed.\n");//stack/queue is empty after initialization


    int x = 10;
    enqueueStack(&q,x);
    int y = dequeueStack(&q);

    assert(q.front == NULL);
    assert(q.rear == NULL);
    assert(x==y);

    printf("Test two passed.\n");
//x = 10 and y = 10. The stack/queue also looks the same as before the test.

    int x0 = 20;
    int x1 = 30;

    enqueueStack(&q, x0);
    enqueueStack(&q, x1);
    int y0 = dequeueStack(&q);
    int y1 = dequeueStack(&q);

    assert(q.front == NULL);
    assert(q.rear == NULL);
    assert(x0==y0);
    assert(x1==y1);

//x0 = 20, x1 = 30, y0 = 20 and y1 = 30. The queue looks the same as before all four commands.
printf("All tests passed.\n");

return 0;
```