Programmering for computerteknologi
Hand-in Assignment Exercises

Week 8: Designing Sequences of Program Instructions for Solving Problems

Please make sure to submit your solutions **by next Monday**.

In the beginning of each question, it is described what kind of answer that you are expected to submit. If *Text and code answer* is stated, then you need to submit BOTH some argumentation/description and some code; if just (*Text answer*) or (*Code answer*) then just some argumentation/description OR code. The final answer to the answers requiring text should be **one pdf document** with one answer for each text question (or text and code question). When you hand-in, add a link to your GitHub reposetory in the beginning of your pdf file. Make sure that you have committed your code solutions to that reposetory.

Note: the **Challenge** exercises are *optional*, the others mandatory (i.e. you **have** to hand them in).

Exercises

(1) (Text answer) Consider the following program for computing factorial numbers:

```
/* Factorial function definition */
int fact(int n)
{
  int i; /* counter variable */
  int f; /* factorial */

  /* pre-condition */
  assert (n >= 0);

  /* post-condition */
  f = 1;
  for(i = 1; i <= n; i = i + 1)
  {
    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)`?

2*5=10

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

2*n

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

The linked list was discussed in lecture six and seven and used the following `typedef`:

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

Hint: each node in the linked list only has a pointer to the next node, so when finding the position to insert the i^{th} element, rather than starting from the largest element you will need to start from the smallest (i.e. the first node in your linked list).

Remark: a linked list in which each node has a pointer to *both* the next node and the previous node is called a *Doubly Linked List*. We have been using *Singly Linked Lists*.

(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 these operation 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

```
typedef struct
{
    int q[QUEUESIZE+1]; /* body of queue */
    int first;          /* position of first element */
    int last;           /* position of last element */
    int count;          /* number of queue elements */
} queue;

void init_queue(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);
}

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

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

(b) Write tests to verify your implementation as presented in the lectures (please review Lecture 3, Section "Testability of a progam" and "Testing functions with functions"). Your tests should ensure the following "laws" hold for any implementation of a queue (variable named `q`:

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

(B) 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`.

(C) After executing
```
enqueue(q, x0); enqueue(q, x1);
y0 = dequeue(q); y1 = dequeue(q);
```
the queue `q` must be the same as before execution of the two commands, `x0` must equal `y0`, and `x1` must equal `y1`.