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** (27-10-2025).

The beginning of each question describes what kind of answer you are expected to submit. If Text answer 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). Make sure that you have committed your code solutions to your GitHub Repository.

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:

```
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?

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;
```

Info

The tests for this exercise expect that a linked list does not have a **sentinel** element in the front or the back of the list. That is if the sequence

$$5, 1, 3, 7, 2$$

Is turned into a linked list, it would look like the one below

If you prefer to solve the exercise with a linked list having a sentinel node you are free to do so. In that case, the tests will most likely not work.

Tip

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 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.

(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

```c
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`.

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);
```

Tip

If the elements are added 1,2,3 (using *enqueue*)

3
2
1

where is the element that you should *dequeue* ? You can get to that element using *pop* and store the other element (2, 3) in an additional stack and then restore the original stack (without 1 and in the same order as before). Try - before programming - to draw it on paper and then implement it. **REMEMBER** you cannot use anything else than the stack methods in this implementation of *queue*

Tip

Make your solution to exercise 2 in the assignment from week seven into a library. Use that library for your solution

Remark

Not accessing the implementation but only using the functions is called *information hiding* and is a very important concept in computer science. This allows an easy change of implementation from e.g. a pointer-based stack to an array-based stack.

Challenge

In Lecture 7 we discussed how we can use a stack to elegantly evaluate an arithmetic expression in post-fix format. Write a program that takes a single post-fix expression (as a string) from the input, parses the string into a queue of symbols, and then evaluates the symbols in the queue as an arithmetic expression using a stack. Write a number of tests to verify that your implementation is correct (as presented in Lecture 3). Assume that no brackets are used, and that spaces separate the symbols, e.g. your parser should be able to evaluate:

```
"4 6 * 8 * 98 -"
```

Separate the task of parsing the string into a queue of symbols into one function, and the task of evaluating a queue of symbols into another function:

```c
void parse(char* expr, queue *q);
int eval(queue *q);
```