Programmering for computerteknologi
Hand-in Assignment Exercises

Week 10: Passing Functions As Arguments To Other Functions

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).

Link to repository:

https://github.com/Aarhus-University-ECE/assignment-10-Teun0n

Exercises

(1) Write a recursive function that prints out a linked list of integers.

(2) Write a recursive function that accepts a linked list of integers, and returns the sum of the *squares* of the integers in the list. E.g.

```
int sum;

node* list =
  make_node(1,
    make_node(2,
      make_node(3,
        make_node(4,
          make_node(5, &SENTINEL)
        )
      )
    )
  );

sum = sum_squares(list); /* sum should equal 55 */
```

(3) In the lecture we discussed the *fold* (or *reduce*) function that takes a list and a *binary* operator (i.e. an operator that takes two inputs) and returns an integer. Another useful general higher order function is *map* that takes a list and a unary operator

(i.e. an operator that accepts just one input), applies the operator to each element in the list, and returns a new list of the resulting values. For example, if X is a list:

```
node* X =
  make_node(1,
    make_node(2,
      make_node(3,
        make_node(4,
          make_node(5, &SENTINEL)
        )
      )
    )
  );
```

We also have a *square* function that multiplies a value by itself:

```
int square(int x) {
  return x * x;
}
```

Your task in this exercise is to implement a recursive *map* function. When you pass the list X and the function *square* to your *map* function, the result should be a linked list with the following elements: 1, 4, 9, 16, 25.

Hint: you should assign the function *square* to a variable, and pass this variable to your *map* function, e.g. consider the following code that assigns the *square* function to a variable:

```
int (*sf)(int);
sf = square;
int x;
x = sf(2);
```

(4) A *binary tree*[1] is a container that stores items for potentially faster retrieval than a linked list. It is equipped with a search operation (called `Contains(x,t)` below). A binary tree is composed from nodes and leafs that store an item. Each node has maximally two children (a *left* child and a *right* child) each of which is either a node or a leaf. The node is called the parent node of the children. A leaf does not have children. The node without a parent is called the root node of the tree. A tree has exactly one root node. The nodes (and leafs) in a binary tree are ordered such that, the left child of a node is smaller or equal than the item of the node and the right child is larger.
The abstract operations on a binary tree include —

- `Insert(x,t)` – Insert item `x` into the tree `t`.
- `Remove(x,t)` – Remove one item `x` from tree `t`. Do nothing if `x` is not contained in the tree.
- `Contains(x,t)` – Return `true` if the tree `t` contains item `x`. Return `false` otherwise.
- `Initialize(t)` – Create an empty tree.
- `Full(t)`, `Empty(t)` – Test whether the tree can accept more insertions or removals, respectively.

(a) Implement a binary tree following the implementation of the singly-linked lists as discussed in the lecture.

(b) Test your implementation. You should expect the following "laws" to hold for any implementation of a queue:

(A) After executing `Initialize(t);` the tree `t` must be empty.

(B) After executing `Insert(x,t); Remove(x,t);` the tree `t` must be the same as before execution of the two commands.

(C) After executing `Insert(x,t); y = Contains(x,t);` the value of `y` must be `true`.

(D) After executing
`Insert(x,t); Insert(y,t); Remove(x,t); z = Contains(y,t);`
the value of `z` must be `true`.

(E) After executing
```
Insert(x,t); Insert(x,t);
Remove(x,t); y = Contains(x,t);
Remove(x,t); z = Contains(x,t);
```
the value of `y` must be `true` and the value of `z` must be `false`.

Implement a binary tree following the implementation of the singly-linked lists as discussed in the lecture.