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** (13-11-2023).

In the beginning of each question, it is described what kind of answer that 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)

See `linked_list.{h,c}`

Code answer Write a recursive function that prints out a linked list of integers, with the following signature:

```
void print_list(node *p);
```

2)

See `linked_list.{h,c}`

Code answer Write a recursive function that takes a linked list of integers, and returns the sum of the *squares* of the integers in the list. E.g

```
node* list =
  make_node(1,
    make_node(2,
      make_node(3,
        make_node(4,
          make_node(5, NULL)
        )
      )
    )
  );

int sum = sum_squares(list);
assert(sum == 55);
```

The function has the following signature:

```c
int sum_squares(node *p);
```

3)

See `linked_list.{h,c}`

Code answer 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:

```c
node* X =
  make_node(1,
    make_node(2,
      make_node(3,
        make_node(4,
          make_node(5, NULL)
        )
      )
    )
  );
```

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

```c
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.

The function has the following signature:

```c
node *map(node *p, int (*f)(int));
```

> 🔥 **Tip**
>
> 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:
>
> ```c
> int (*sf)(int);
> sf = square;
> int x;
> x = sf(2);
> ```

4)

Code answer 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 search 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.
- `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. **Create a new file**, where you include your binary tree library header file. You should expect the following "laws" to hold for any implementation of a binary tree. **Hint:** you could enforce these conditions using `assert` statements:

(1) After executing `insert(x, t); remove(x, t);` the tree *t* must be the same as before execution of the two commands.

(2) After executing `insert(x, t); y = contains(x, t);` the value of *y* must be true.

(3) After executing
```
insert(x, t); insert(y, t);
remove(x, t); z = contains(y, t);
```
the value of *z* must be `true`.

(4) 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.

[1] There are more variants of binary trees that differ from the one discussed here.

🔥 Tip

In `btree.c` a function called `pp_btree_node()` is implemented that you can use to pretty print a binary tree. It might be helpful to you, if you need to debug your code and see the state of a binary tree 🤓

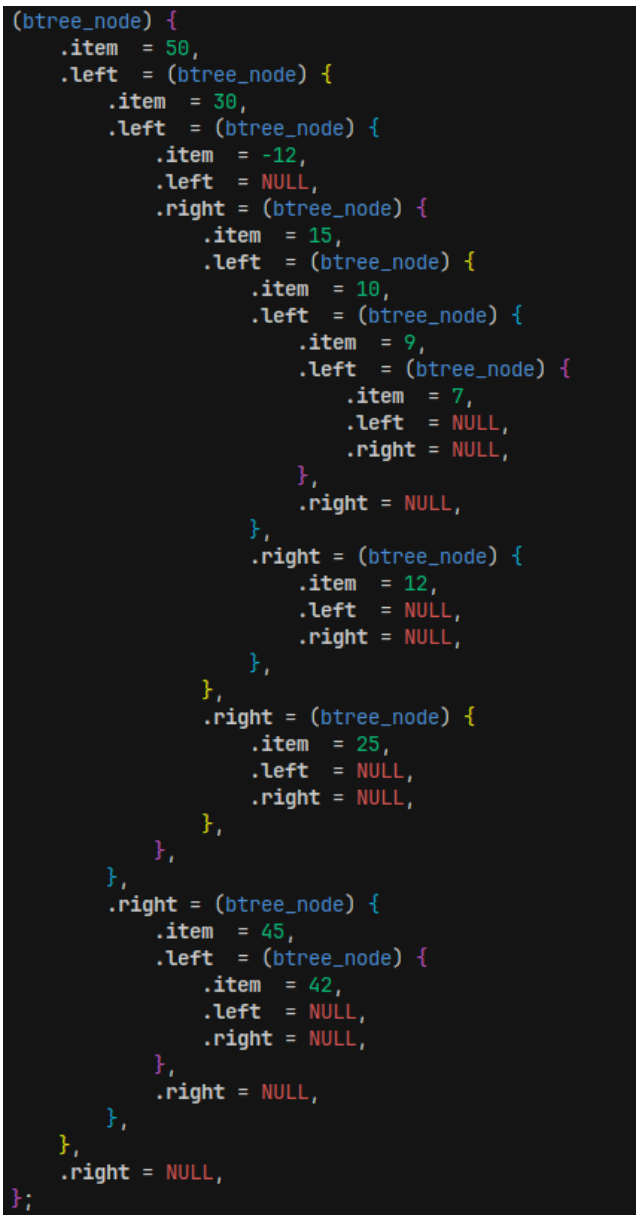
Here is how you can use it:

```c
void pp_btree_node(const btree_node *node, const int indent_by, const int depth);

btree_node* root = NULL;
// create the binary tree ...

pp_btree_node(root, 4, 0);
```

Example output:



Challenge

The eight queens puzzle is a classic puzzle where the task is to place eight queens on a chess board of size 8×8 such that no two queens can attack each other. In chess, a queen piece can attack any other piece in a straight line horizontally, vertically or diagonally:

Queen Q can attack pieces in position *

```
* - - * - - * -
- * - * - * - -
- - * * * - - -
* * * Q * * * *
- - * * * - - -
- * - * - * - -
* - - * - - * -
- - - * - - - *
```

(a) Write a recursive program that solves the eight queens puzzle, i.e. placing eight queens on a chess board of size 8×8. You can represent the pieces and the board in any way that you deem appropriate (arrays, using structs, etc.). Print the solution to the console (e.g. as above when illustrating queen attack positions).

(b) Modify your program so that, instead of queens, the task is to place as many *knight* pieces as possible on an 8×8 board such that no two knights can attack each other.

Knight K can attack pieces in position *

```
- - - - - - - -
- - * - * - - -
- * - - - * - -
- - - K - - - -
- * - - - * - -
- - * - * - - -
- - - - - - - -
- - - - - - - -
```

What is the maximum number of knights your program can place? Display your solution to the console.

(c) An interesting variation to the eight queens puzzle is to consider that the board can *wrap* around horizontally, as if it was drawn on a cylinder. The horizontal and vertical attack positions are the same, but now the diagonal attack positions are more complicated, e.g. here are the attack positions that only consider the up-left (or down right) diagonal directions:

Queen Q can attack pieces in diagonal up-left / down-right positions *

```
- - - - * - - -
- - - - - Q - -
- - - - - - * -
* - - - - - - *
- * - - - - - -
- - * - - - - -
- - - * - - - -
- - - - * - - -
```

The queen can of course also attack diagonal up-right (down-left), and horizontally and vertically. Modify your recursive program so that it places as many queens as possible on a *cyclic* 8×8 board.