Department of Informatics
Database Technology Group
Prof. Dr. M. Böhlen

# Informatik II
# Assignment 4

Apr 9, 2018

## Abstract Data Types (ADT) [30 points]

**Task 1.** **[20 points]** Create a `stack` of positive integers that is able to store an arbitrary number of elements. Each node in stack is of the following type:

```
struct Node {
    int key;
    struct Node* next;
};
```

Along with the above datatype create the following functions:

a) *int isEmpty(struct Node* stackTop)* which checks if stack is empty or not.

b) *void push(struct Node** stackTop, int key)* which inserts element `key` into stack.

c) *int pop(struct Node** stackTop)* which removes the last inserted element from stack and returns its value. In case of an empty stack, -1 should be returned.

d) *void printStack(struct Node* stackTop)* which prints the values of the stack.

Test you implementation by performing the following operations:

- Create a node `stackTop`;
- Push 3, 4, 6, 9, 10 into stack while updating `stackTop` node.
- Pop two elements and print their values.
- Push 4, 17, 30.
- Print stack.

**Task 2.** [**10 points**] The FIFO behaviour of a queue can be simulated using two LIFO stacks. The elements of the queue are moved from one stack to another to properly perform the enqueue and dequeue operations. After the completion of a enqueue and dequeue operations, all elements are stored in only one of the two stacks.
Based on the `stack` implementation from the previous task, create a new datatype `Queue` of positive integers that is able to store an arbitrary number of elements. The FIFO `Queue` datatype created from two stacks is as follows.

```
struct Queue {
    struct Node* stack1;
    struct Node* stack2;
};
```

Along with the above datatype create the following functions:

a) *void enQueue(struct Queue *q, int key)* which inserts element `key` into `q`.

b) *int deQueue(struct Queue *q)* which removes the first inserted element from `q` and returns its value. In case of an empty queue, -1 should be returned.

c) *void printQueue(struct Queue *q)* which prints the values of the queue `q`.

Test you implementation by performing the following operations:

- Create Queue `q`;
- Enqueue 2, 8, 11, 15 into `q`.
- Dequeue two elements and print their values.
- Enqueue 1, 5.
- Print q.

## Binary Search Trees [30 points]

```
struct TreeNode {
  int val;
  struct node* left;
  struct node* right;
};
```

**Task 3.** [**15 points**] Given the above definition of a binary search tree with positive integer values, create a C program that contains the following functions:

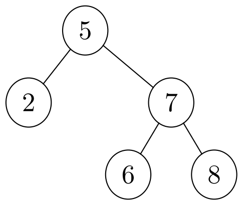
a) the function *void insert(struct TreeNode** root, int val)*, which gets a root node and a value `val` as an input and inserts a node with the value `val` into the proper position of the binary search tree.

b) the function *struct TreeNode* search(struct TreeNode* root, int val)*, which finds the tree node with value `val` and returns the node.

c) the function *void delete(struct TreeNode** root, int val)*, which deletes the node with value `val` from the tree.

d) the function *void print(struct TreeNode* root)*, which prints the elements of a tree in order.

For example, if the values 5, 2, 7, 6 and 8 are inserted into an empty tree, your program should produce the binary tree shown below.



And it should print its elements in order: $2, 5, 6, 7, 8$. Test your program by performing the following operations:

- Create a root node `root` and insert the values 4, 2, 3, 8, 6, 7, 9, 12, 1.
- Print tree to the console.
- Delete the values 4, 7, 2 from the tree.
- Print tree to the console.

**Task 4. [15 points]** Extend the program of the previous task with the following functions:

a) *struct TreeNode* maximum(struct TreeNode* n)*, which returns the node with the largest value in the subtree with root node `n`.

b) *struct TreeNode* minimum(struct TreeNode* n)*, which returns the node with the smallest value in the subtree with root node `n`.

c) *struct TreeNode* successor(struct TreeNode* root, struct TreeNode* n)*, which takes a root node and any other node of the tree as an argument and returns the node with the next largest value. If the node given as the argument has the largest value in the tree, your function should return `NULL`.

d) *struct node* ith_smallest(struct TreeNode* root, int i)* that returns the node with the $i^{th}$-smallest element of the tree and is implemented using the functions *successor* and *minimum*. If the $i^{th}$-smallest element does not exist, your function must return `NULL`.

e) *int distanceToRoot(struct TreeNode* root, int x)* that returns the distance of the node with value `x` from the root node `root`.

f) *int distance(struct TreeNode* n, int a, int b)* that returns the distance between the two nodes with values `a` and `b` in a tree with root node `root`. Use the function *distanceToRoot* and assume that $a < b$.

## Submission

For this exercise, you need to submit a zipped folder *a<exercise number>_<family name>_<matriculation number>.zip* where `family name` and `matriculation number` correspond to your personal data. This folder should include:

a) the C-files you created for each of the tasks. Each C-file should be named as *task<task number>.c*

b) a pdf named *a<exercise number>.pdf* with the solutions for Tasks 3 and 5a.

Make sure that both in the C-files as well as in the pdf file you submit, your personal data is included (in the form of comments or a note).

## Notes

- You are allowed to create as many additional functions as you need in all exercises and you can also reuse material from previous exercises if needed.

Deadline: **Sunday, April 22nd at 23:59**.