University of Zurich[UZH]

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

Data Structures and Algorithms
Assignment 4

Apr 4, 2016

Abstract Data Types (ADT) [30 points]

Task 1. [20 points] Create a new datatype `stack` of positive integers that is able to store an arbitrary number of elements. A stack is of the following type:

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

Along with the above datatype create the following functions:

- *int isEmpty(struct stack* s)* which checks if stack `s` is empty or not.
- *void push(struct stack** s, int key)* which inserts element `key` into `s`.
- *int pop(struct stack** s)* which removes the last inserted element from `s` and returns its value. In case of an error, -1 should be returned.
- *void printStack(struct stack* s)* which prints the values of the stack `s`.

Test you implementation by performing the following operations:

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

Task 2. **[10 points]** Postfix notation is a mathematical notation used to represent algebraic expressions. In postfix each operator follows their operands; for instance, to add 3 and 4, one would write "3 4 +" rather than "3 + 4" (infix notation). If there are multiple operators, each operator is given immediately after its second operand. So the expression written "3 - 4 + 5", in conventional (infix) notation, is written "3 4 - 5 +" in postfix notation. 4 is first subtracted from 3, then 5 is added to the first result. An advantage of postfix notation is that it removes the need for parentheses that are required by infix. For example "3 - 4 * 5" (infix) can also be written "3 - (4 * 5)", that means something quite different from "(3 - 4) * 5". In postfix the former is written "3 4 5 * -", which unambiguously means "3 (4 5 *) -" and the latter is written "3 4 - 5 *", which unambiguously means "(3 4 -) 5 *".

Here are steps for evaluating a postfix expression:

- Create a stack to store operands.
- Scan the given expression and do the following for every scanned element.
  - If the element is a number, push it into the stack.
  - If the element is an operator, pop two operands from the stack, evaluate the operator, and push the result back to the stack.
- When the whole expression is scanned, the number in the stack is the final answer.

Using the `stack` implementation from Task 1, implement in C the function `int evaluatePostfix(char* exp)` to evaluate a given postfix notation and return its numerical value.

Write a program in C to evaluate these 3 postfix expressions:

- $231 * +9-$
- $128 * +3-$
- $25 * 32 + /$

Notes

- You can assume that all operands in the postfix expression are 1-digit numbers (0..9) only.
- You can assume that operators in the postfix expression are either + or − or ∗ or /
- You can assume that there is no whitespace in the postfix expression.

Binary Search Trees [45 points]

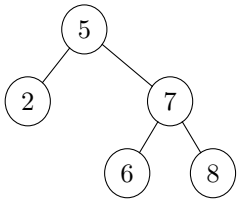
Assume a binary search tree with nodes defined as follows:

```
struct TreeNode {
  int key;
  struct TreeNode* lChild;
  struct TreeNode* rChild;
};
```

Task 3. [25 points] Use the node-definition above to create multiple tree instances. Create a C program including:

- the function *insert* which gets the root of a tree and a value *key* as an input and inserts a node with the value *key* into the proper position of the given binary search tree.
- the function *deleteTree* which gets the root of a tree as an input, removes all the nodes of the tree and sets its root to `NULL`.

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



Test your program by performing the following operations:

- Create an empty tree `root1` and insert the keys 4, 2, 3, 8, 6, 7, 9, 12, 1.
- Create an empty tree `root2` and insert the keys 3, 8, 10, 1, 7.
- Delete the tree root1.
- Delete the tree root2.

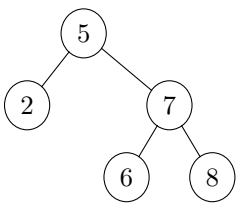
Task 4. [10 points] Write a C program that extends the binary search tree defined in the previous tasks and includes the functions:

- *printTree* which prints a given tree in the console. The printing format should be `graph g {` (all the edges in the form `NodeA -- NodeB`) `}`, where each edge should be printed on a separate line. The ordering of the edges is not relevant.

Output Form

```
graph g {
  5 -- 2
  5 -- 7
  7 -- 6
  7 -- 8
}
```

Alternative Output Form



- *printInOrder* which traverses a given tree *in-order* and prints the keys of all the nodes.

Test your program by performing the following operations:

- Insert the keys 5, 3, 4, 9, 7, 8, 10, 13, 0, 6 into an empty tree.
- Print the resulting tree using the function `printTree`.
- Print the resulting tree using the function `printInOrder`.
- Delete the tree.
- Print the resulting tree using the function `printTree`.

Task 5. [10 points] Given a rooted tree T, the *lowest common ancestor (LCA)* between two nodes *n1* and *n2* is defined as the lowest node in T that has both *n1* and *n2* as descendants (where we allow a node to be a descendant of itself). Consequently, the LCA of *n1* and *n2* in T is the shared ancestor of *n1* and *n2* that is located farthest from the root.

For example, given a tree as in Figure 1, the LCA of nodes *9* and *21* is node *15*.

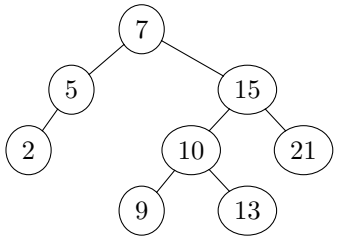


Figure 1: LCA of nodes 9 and 21 is node 15

Given values of two nodes in a Binary Search Tree, implement in C the function `struct TreeNode *lca(TreeNode* root, int n1, int n2)` that finds the *Lowest Common Ancestor (LCA)*. You may assume that both the values exist in the tree.

Write in C a program to test your implementation by performing the following operations:

- Create an empty tree and insert the nodes 7, 5, 6, 1, 9, 10, 8.
- Print the resulting tree using one of the methods described in **Task 4**.
- Print out the LCA of node *8* and node *9*.
- Print out the LCA of node *1* and node *6*.

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 the C-files you created for each of the tasks. Each C-file should be named as *task<task number>.c* and it should also include you personal data in the form of a comment on the top.
Remember to submit a different C-file for each of the tasks.

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.
- Every time you allocate dynamic memory with `malloc`, make sure you `free` it properly.

Deadline: **Sunday, April 17th at 23:59**.