Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
François Hublet, Goran Zuzic, Tommaso d'Orsi, Jingqiu Ding

14 November 2022

Algorithms & Data Structures — Exercise sheet 8 — HS 22

The solutions for this sheet are submitted at the beginning of the exercise class on 21 November 2022.

Exercises that are marked by * are "challenge exercises". They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Exercise 8.1 *Exponential bounds for a sequence defined inductively.*

Consider the sequence $(a_n)_{n\in\mathbb{N}}$ defined by

$$\begin{aligned} a_0 &= 1, \\ a_1 &= 1, \\ a_2 &= 2, \\ a_i &= a_{i-1} + 2a_{i-2} + a_{i-3} \quad \forall i \geq 3. \end{aligned}$$

The goal of this exercise is to find exponential lower and upper bounds for a_n.

(a) Find a constant $C > 1$ such that $a_n \leq O(C^n)$ and prove your statement.

(b) Find a constant $c > 1$ such that $a_n \geq \Omega(c^n)$ and prove your statement.

Remark. One can actually show that $a_n = \Theta(\phi^n)$, where $\phi \approx 2.148$ is the unique positive solution of the equation $x^3 = x^2 + 2x + 1$.

Exercise 8.2 *AVL trees* **(1 point)**.

(a) Draw the tree obtained by inserting the keys 1, 6, 8, 0, 3, 2, 9 in this order into an initially empty AVL tree. Give also the intermediate states before and after each rotation that is performed during the process.

(b) Delete 0, 2, and 1 in this tree, and afterwards delete key 6 in the resulting tree. Give also the intermediate states before and after each rotation is performed during the process.

Exercise 8.3 *Augmented Binary Search Tree.*

Consider a variation of a binary search tree, where each node has an additional member variable called SIZE. The purpose of the variable SIZE is to indicate the size of the subtree rooted at this node. An example of an augmented binary search tree (with integer data) can be seen below (Fig. 1).

a) What is the relation between the size of a node and the sizes of its children?

Hint: *Consider the longest simple path in G. Prove that its endpoint is a leaf.*

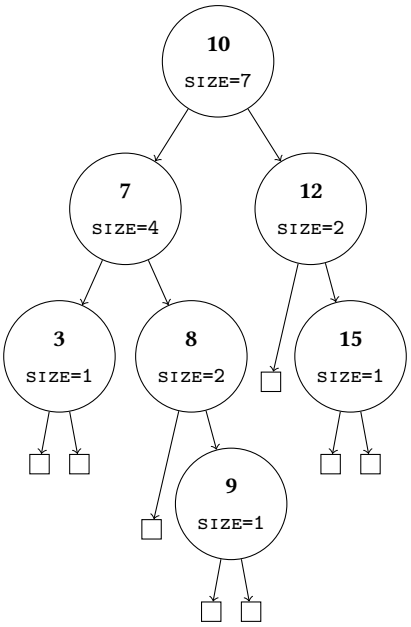


Figure 1: Augmented binary search tree

b) Describe in pseudo-code an algorithm VERIFYSIZES(ROOT) that returns TRUE if all the sizes in the tree are correct, and returns FALSE otherwise. For example, it should return TRUE given the tree in Fig. 1, but FALSE given the tree in Fig. 2.

What is the running time of your algorithm? Justify your answer.

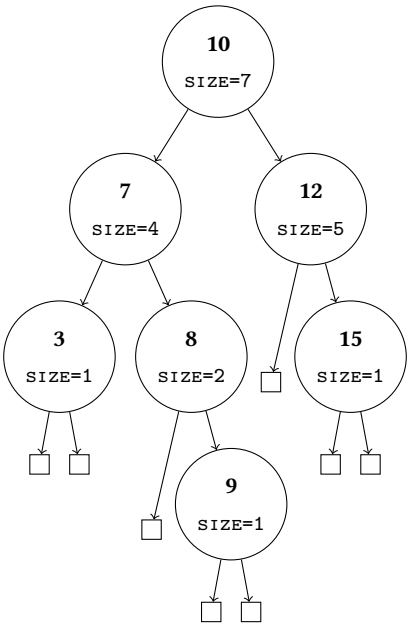


Figure 2: Augmented binary search tree with buggy size: incorrect size for node with data "12"

c) Suppose we have an augmented AVL tree (i.e., as above, each node has a SIZE member variable). Describe in pseudo-code an algorithm SELECT(ROOT, k) which, given an augmented AVL tree and an integer k, returns the k-th smallest element in the tree in $O(\log n)$ time.

Example: Given the tree in Fig. 1, for $k = 3$, SELECT returns 8; for $k = 5$, it returns 10; for $k = 1$, it returns 3; etc.

d)* To maintain the correct sizes for each node, we have to modify the AVL tree operations, insert and remove. For this problem, we will consider only the modifications to the AVL-INSERT method (i.e., you are not responsible for AVL-REMOVE). Recall that AVL-INSERT first uses regular INSERT for binary search trees, and then balances the tree if necessary via rotations.

- How should we update INSERT to maintain correct sizes for nodes?

During the balancing phase, AVL-INSERT performs rotations. Describe what updates need to be made to the sizes of the nodes. (It is sufficient to describe the updates for left rotations, as right rotations can be treated analogously.)

Exercise 8.4 *Round and square brackets.*

A string of characters on the alphabet $\{\texttt{A}, \ldots, \texttt{Z}, \texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}\}$ is called *well-formed* if either

1. It does not contain any brackets, or
2. It can be obtained from an empty string by performing a sequence of the following operations, in any order and with an arbitrary number of repetitions:
   (a) Take two non-empty well-formed strings a and b and concatenate them to obtain ab,
   (b) Take a well-formed string a and add a pair of round brackets around it to obtain $\texttt{(}a\texttt{)}$,
   (c) Take a well-formed string a and add a pair of square brackets around it to obtain $\texttt{[}a\texttt{]}$.

The above reflects the intuitive definition that all brackets in the string are 'matched' by a bracket of the same type. For example, $s = \texttt{FOO(BAR[A])}$, is well-formed, since it is the concatenation of $s_1 = \texttt{FOO}$, which is well-formed by 1., and $s_2 = \texttt{(BAR[A])}$, which is also well-formed. String s_2 is well-formed because it is obtained by operation 2(b) from $s_3 = \texttt{BAR[A]}$, which is well-formed as the concatenation of well-formed strings $s_4 = \texttt{BAR}$ (by 1.) and $s_5 = \texttt{[A]}$ (by 2(c) and 1.). String $t = \texttt{FOO[(BAR])}$ is not well-formed, since there is no way to obtain it from the above rules. Indeed, to be able to insert the only pair of square brackets according to the rules, its content $t_1 = \texttt{(BAR}$ must be well-formed, but this is impossible since t_1 contains only one bracket.

Provide an algorithm that determines whether a string of characters is well-formed. Justify briefly why your algorithm is correct, and provide a precise analysis of its complexity.

Hint: *Use a data structure from the last lecture.*

Exercise 8.5 *Computing with a stack* **(2 points)**.

In many programming languages, e.g., in Python, stacks are commonly used for evaluating arithmetic expressions. Evaluating expressions usually happens in two steps. First, values are loaded into the stack. Then, operations are applied stepwise on the top elements in order to obtain the desired value.

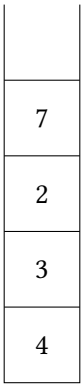


Figure 3: A stack S_0 containing the numbers 4, 3, 2, and 7 (7 is the top of the stack)

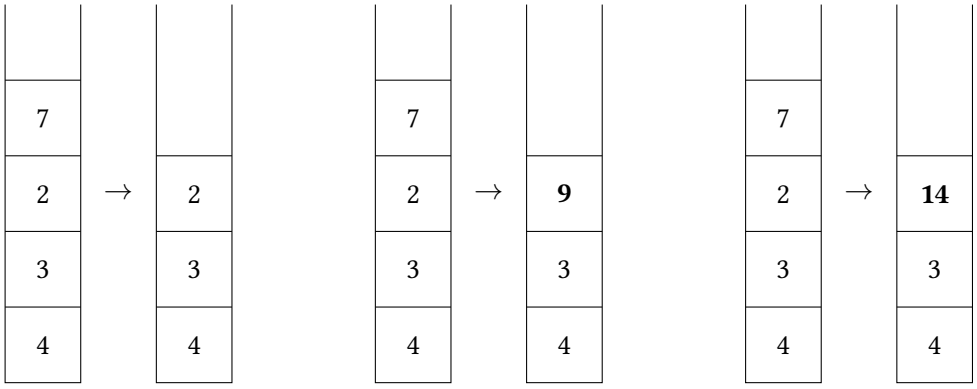
In this exercise, we focus on the second phase, and consider the following three basic operations used to compute with stacks:

POP: If there is at least one element in the stack, remove the top element of the stack. Otherwise, do nothing.

ADD: If there are at least two elements in the stack, remove the top two elements, compute their sum, and push this sum back into the stack. If there is less than two elements in the stack, do nothing.

MUL: If there are at least two elements in the stack, remove the top two elements, compute their product, and push this product back into the stack. If there is less than two elements in the stack, do nothing.

Below are examples of applications of POP, ADD, and MUL.



(a) POP (remove 7) (b) ADD ($7 + 2 = 9$) (c) MUL ($7 \cdot 2 = 14$)

We say that an integer i *can be computed from a stack* S if and only if there exists a sequence of POP, ADD, and MUL operations on S that ends with i on top of the stack. For example, the value $(3 \cdot 2) + 4 = 10$ can be computed from the stack S_0 above through the following sequence of operations:

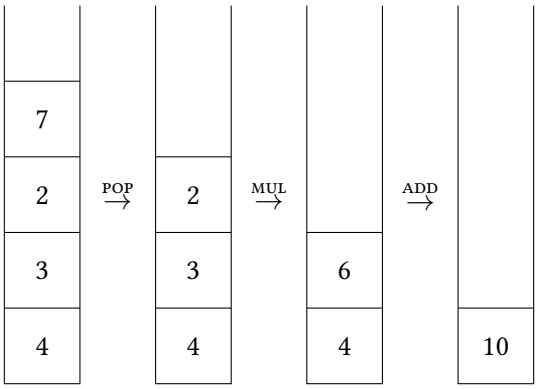


Figure 5: Computing 10 from S_0

Given a stack S containing n integers $S_1, \ldots, S_n \in \{1, \ldots, k\}$ (with S_1 being the top of the stack) and an integer c, you are tasked to design a DP algorithm which determines if c can be computed from S. To obtain full points, your algorithm should have complexity at most $O(c \cdot n)$, but partial points will be awarded for any solution running in time $O(k^n \cdot n)$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Definition of the DP table*: What is the meaning of each entry?
3. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

(*) *Challenge question*: Extend your algorithm to support the following additional operation:

NEG: If there is at least one element in the stack, remove the top element x of the stack, and push $-x$ back into the stack. Otherwise, do nothing.