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.

**Solution:**

Intuitively, the sequence $(a_n)_{n\in\mathbb{N}}$ seems to be increasing. Assuming so, we would have

$$a_i = a_{i-1} + 2a_{i-2} + a_{i-3} \leq a_{i-1} + 2a_{i-1} + a_{i-1} = 4a_{i-1},$$

which yields

$$a_n \leq 4a_{n-1} \leq \ldots \leq 4^n a_0 = 4^n.$$

This only comes from an intuition, but it is a good way to guess what the upper bound could be. Now let us actually prove (by induction) that $a_n \leq 4^n$ for all $n \in \mathbb{N}$.

**Induction Hypothesis.** We assume that for $k \geq 2$ we have

$$a_k \leq 4^k, \qquad a_{k-1} \leq 4^{k-1}, \qquad a_{k-2} \leq 4^{k-2}. \tag{1}$$

**Base case** $k = 2$**.** Indeed we have $a_0 = 1 \leq 4^0$, $a_1 = 1 \leq 4^1$ and $a_2 = 2 \leq 4^2$.

**Inductive step (**$k \to k+1$**).** Let $k \geq 2$ and assume that the induction hypothesis (1) holds. To show that it also holds for $k+1$, we need to check that $a_{k+1} \leq 4^{k+1}$, $a_k \leq 4^k$ and $a_{k-1} \leq 4^{k-1}$. The two last inequalities clearly hold since they are part of the induction hypothesis, so we only need to check that $a_{k+1} \leq 4^{k+1}$. Indeed,

$$a_{k+1} = a_k + 2a_{k-1} + a_{k-2} \overset{(1)}{\leq} 4^k + 2\cdot 4^{k-1} + 4^{k-2} \leq 4^k + 2\cdot 4^k + 4^k = 4\cdot 4^k = 4^{k+1}.$$

Thus, $a_n \leq 4^n$ for all $n \in \mathbb{N}$. In particular, we have shown that $a_n \leq O(C^n)$ for $C = 4 > 1$.

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

**Solution:**

If we again assume that the sequence is increasing, we would get

$$a_i = a_{i-1} + 2a_{i-2} + a_{i-3} \geq a_{i-3} + 2a_{i-3} + a_{i-3} = 4a_{i-3},$$

which yields

$$a_n \geq 4a_{n-3} \geq \ldots \geq 4^{\lfloor n/3 \rfloor} a_0 = 4^{\lfloor n/3 \rfloor}.$$

So we will aim to prove a lower bound of the form $a_n \geq \varepsilon \cdot 4^{n/3}$ for some constant $\varepsilon > 0$. We see that taking $\varepsilon := \min\{1, 4^{-1/3}, 2 \cdot 4^{-2/3}\} = 4^{-1/3}$ will make the inequality satisfied for the base case, so let's prove by induction that $a_n \geq 4^{-1/3}4^{n/3}$ for all $n \in \mathbb{N}$.

**Induction Hypothesis.** We assume that for $k \geq 2$ we have

$$a_k \geq 4^{-1/3}4^{k/3}, \qquad a_{k-1} \geq 4^{-1/3}4^{(k-1)/3}, \qquad a_{k-2} \geq 4^{-1/3}4^{(k-2)/3}. \tag{2}$$

**Base case $k = 2$.** Indeed we have $a_0 = 1 \geq 4^{-1/3} \cdot 4^0$, $a_1 = 1 \geq 4^{-1/3}4^{1/3}$ and $a_2 = 2 \geq 4^{1/3} = 4^{-1/3}4^{2/3}$.

**Inductive step ($k \to k + 1$).** Let $k \geq 2$ and assume that the induction hypothesis (2) holds. To show that it also holds for $k+1$, we need to check that $a_{k+1} \geq 4^{-1/3}4^{(k+1)/3}$, $a_k \geq 4^{-1/3}4^{k/3}$ and $a_{k-1} \geq 4^{-1/3}4^{(k-1)/3}$. The two last inequalities clearly hold since they are part of the induction hypothesis, so we only need to check that $a_{k+1} \geq 4^{-1/3}4^{(k+1)/3}$. Indeed,

$$\begin{aligned} a_{k+1} = a_k + 2a_{k-1} + a_{k-2} &\overset{(2)}{\geq} 4^{-1/3}\left(4^{k/3} + 2 \cdot 4^{(k-1)/3} + 4^{(k-2)/3}\right) \\ &\geq 4^{-1/3}\left(4^{(k-2)/3} + 2 \cdot 4^{(k-2)/3} + 4^{(k-2)/3}\right) = 4^{-1/3} \cdot 4 \cdot 4^{(k-2)/3} = 4^{-1/3}4^{(k+1)/3}. \end{aligned}$$

Thus, $a_n \geq 4^{-1/3}4^{n/3}$ for all $n \in \mathbb{N}$. In particular, we have shown that $a_n \geq \Omega(c^n)$ for $c = 4^{1/3} > 1$.

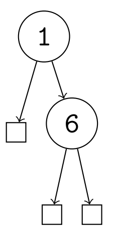
**Remark.** One can actually show that $a_n = \Theta(\phi^n)$, where $\phi \approx 2.148$ is the unique positive solution of the equation $x^3 = x^2 + 2x + 1$.

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

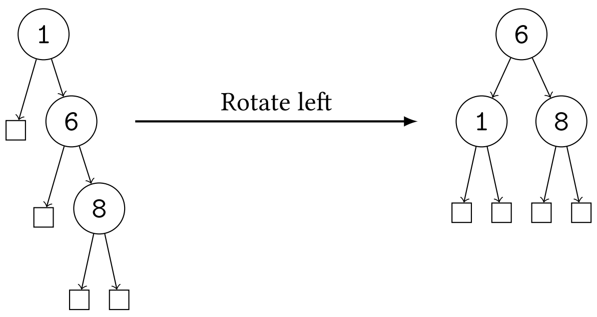
(a) Draw the tree obtained by inserting the keys 1, 6, 8, 0, 3, 2, 9 in this order into an initially empty AVL tree. Give also the intermediate states before and after each rotation that is performed during the process.

**Solution:**

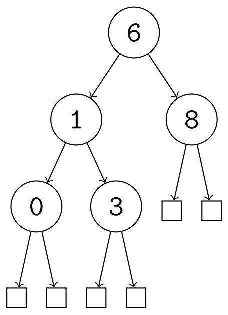
**Insert 1 and then 6:**



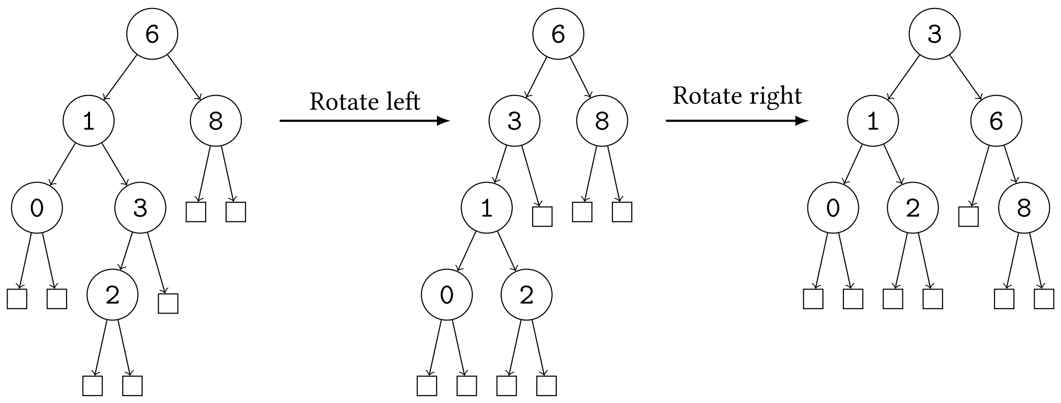
**Insert 8:**



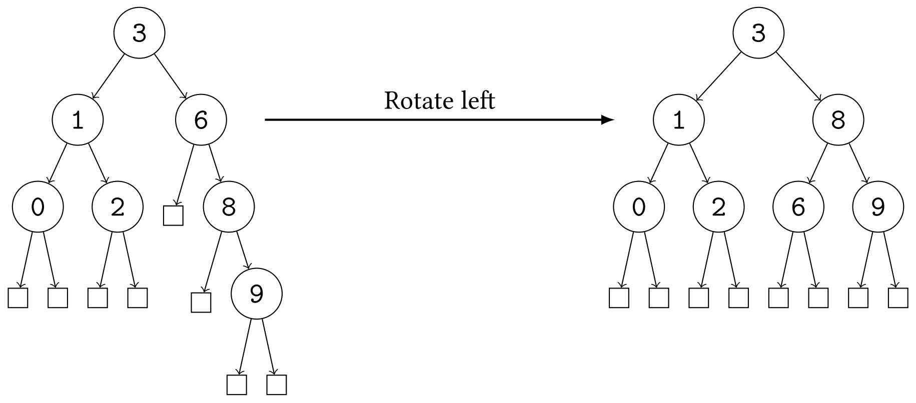
**Insert 0 and 3:**



**Insert 2:**



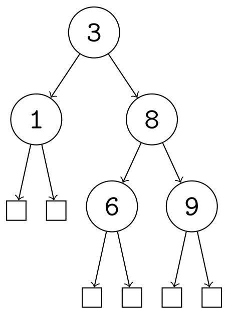
**Insert 9:**



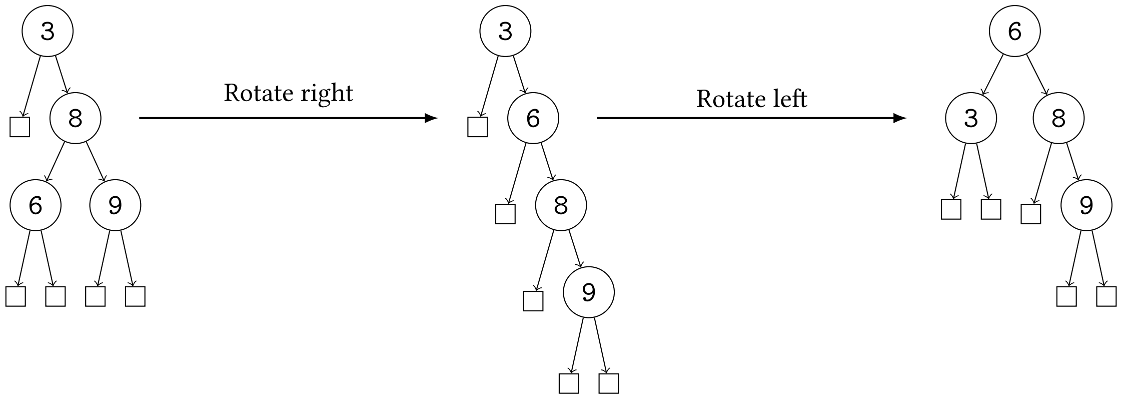
(b) Delete 0, 2, and 1 in this tree, and afterwards delete key 6 in the resulting tree. Give also the intermediate states before and after each rotation is performed during the process.

**Solution:**

**Delete 0 and 2:**

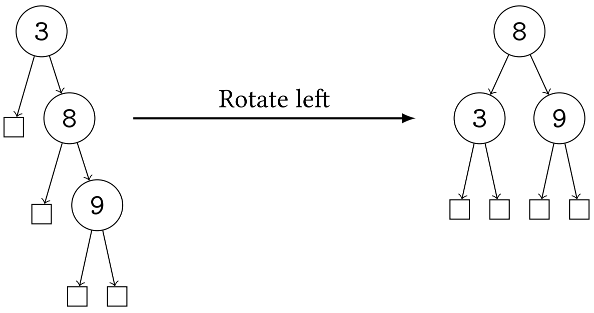


**Delete 1:**

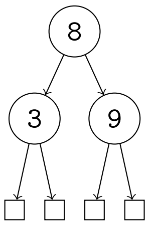


**Delete 6:**

Key 6 can either be replaced by its predecessor key, 3, or its successor key, 8. If key 6 is replaced by its predecessor:



If key 6 is instead replaced by its successor:



**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).

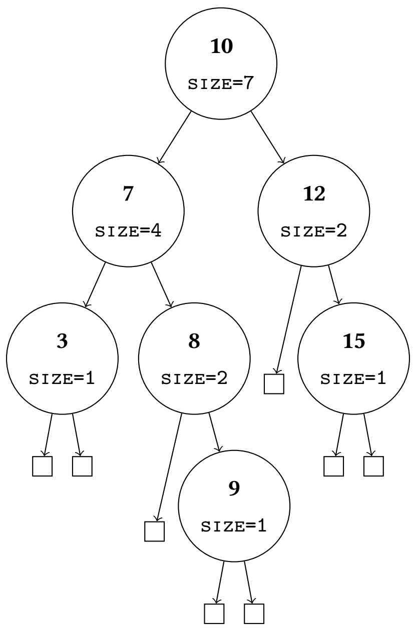


Figure 1: Augmented binary search tree

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

**Solution:**

For every node in the tree, we have

$$\text{NODE.SIZE} = \text{NODE.LEFT.SIZE} + \text{NODE.RIGHT.SIZE} + 1.$$

Note that throughout the solution of this exercise, we adopt the convention that NULL.SIZE $= 0$.

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.

**Solution:**

**Algorithm 1** Verifying the sizes of the tree

```
function VerifySizes(root)
    if root = null then
        return true
    else if VerifySizes(root.left) = false or VerifySizes(root.right) = false then
        return false
    else
        CorrectSize ← 1 + root.left.size + root.right.size
        return CorrectSize = root.size
```

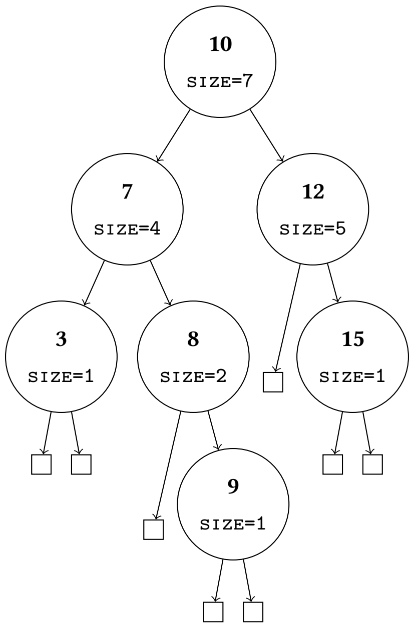


Figure 2: Augmented binary search tree with buggy size: incorrect size for node with data “12”

The above recursive algorithm visits every node of the tree exactly once. Furthermore, it performs a constant number of operations $O(1)$ at each node. Therefore, the runtime is $O(n)$, where $n$ is the number of nodes in the tree.

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.

**Solution:**

**Algorithm 2** Selecting the $k$-th smallest element

```
function SELECT(ROOT, $k$)
    CURRENT $\leftarrow$ ROOT.LEFT.SIZE $+ 1$
    if $k =$ CURRENT then
        return ROOT.DATA
    else if $k <$ CURRENT then
        return SELECT(ROOT.LEFT, $k$)
    else
        return SELECT(ROOT.RIGHT, $k -$ CURRENT)
```

The above algorithm follows a downward path until it finds the correct node. Furthermore, it performs a constant number of operations $O(1)$ at each visited node. Therefore, the runtime of the algorithm is $O(h)$, where $h$ is the height of the tree. Now since the tree is an AVL tree, we have $h = O(\log n)$. We conclude that the runtime of the above algorithm is $O(\log n)$.

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?

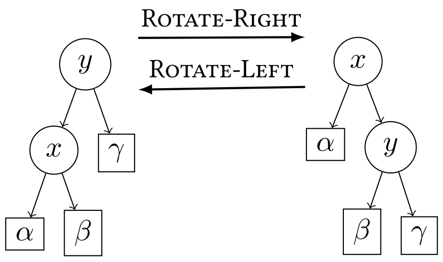
During the balancing phase, AVL-INSERT performs rotations. Describe what updates need to be made to the sizes of the nodes. (It is sufficient to describe the updates for left rotations, as right rotations can be treated analogously.)

**Solution:**

The regular INSERT function follows a downward path and then adds the new node as a leaf at the correct place. We only need to increment the variable SIZE by 1 at each visited node, and set the variable SIZE of the added leaf to 1. The runtime of the modified function remains $O(h)$, where $h$ is the height of the tree. If the tree is an AVL tree, then the runtime is $O(\log n)$.

Regarding AVL-INSERT, after modifying the regular INSERT function as we explained, we need to modify the rotation functions LEFT-ROTATE and RIGHT-ROTATE to maintain the correct SIZE variables.

Suppose we are performing a right-rotation on the node $y$ of the tree that is drawn on the left (or performing a left-rotation on the node $x$ of the tree that is drawn on the right):



In the above diagrams, $\alpha$, $\beta$ and $\gamma$ represent subtrees. As can be easily seen, only $x$.SIZE and $y$.SIZE need to be updated, and we can apply the relation in a) in the correct order:

- At the end of ROTATE-RIGHT, we apply

  $$y.\text{SIZE} \leftarrow y.\text{LEFT}.\text{SIZE} + y.\text{RIGHT}.\text{SIZE} + 1,$$

  and then

  $$x.\text{SIZE} \leftarrow x.\text{LEFT}.\text{SIZE} + x.\text{RIGHT}.\text{SIZE} + 1.$$

- At the end of ROTATE-LEFT, we apply

  $$x.\text{SIZE} \leftarrow x.\text{LEFT}.\text{SIZE} + x.\text{RIGHT}.\text{SIZE} + 1,$$

  and then

  $$y.\text{SIZE} \leftarrow y.\text{LEFT}.\text{SIZE} + y.\text{RIGHT}.\text{SIZE} + 1.$$

As we can see, the runtime of the modified RIGHT-ROTATE (resp. LEFT-ROTATE) function remains $O(1)$. Therefore, the runtime of AVL-INSERT remains $O(\log n)$.

*Remark:* It is also possible to modify AVL-DELETE to maintain the correctness of the SIZE variables while keeping the $O(\log n)$ runtime.

**Exercise 8.4** *Round and square brackets.*

A string of characters on the alphabet {A, ..., Z, (, ), [, ]} 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 ($a$),
   (c) Take a well-formed string $a$ and add a pair of square brackets around it to obtain [$a$].

The above reflects the intuitive definition that all brackets in the string are 'matched' by a bracket of the same type. For example, $s =$ `FOO(BAR[A])`, is well-formed, since it is the concatenation of $s_1 =$ `FOO`, which is well-formed by 1., and $s_2 =$ `(BAR[A])`, which is also well-formed. String $s_2$ is well-formed because it is obtained by operation 2(b) from $s_3 =$ `BAR[A]`, which is well-formed as the concatenation of well-formed strings $s_4 =$ `BAR` (by 1.) and $s_5 =$ `[A]` (by 2(c) and 1.). String $t =$ `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 $t_1 =$ `(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.*

**Solution:**

We use a stack providing standard POP, PUSH, and ISEMPTY operations. Given a stack $S$, $S$.POP() removes and returns the element on top of the stack, if it exists, and a constant `None` otherwise, while $S$.PUSH($x$) pushes $x$ onto the top of the stack, and $S$.ISEMPTY() returns a boolean indicating whether the stack is empty or not. Finally, we assume a function EMPTYSTACK that initializes and returns an empty stack. Our algorithm is as follows:

**Algorithm 3** Detecting well-formed strings

```
function IsWellFormed($s$)
    $S \leftarrow$ emptyStack()
    for $i \in \{0, ..., |s| - 1\}$ do
        if $s[i] =$ "(" then
            $S$.push("(")
        else if $s[i] =$ "[" then
            $S$.push("[")
        else if $s[i] =$ ")" then
            if $S$.pop() $\neq$ "(" then
                return False
        else if $s[i] =$ "]" then
            if $S$.pop() $\neq$ "[" then
                return False
    return $S$.isEmpty()
```

**Correctness.** First, we see that we can completely ignore non-bracket characters to determine well-formedness. The correctness of our algorithm then results from the following invariant: for all $s$, the for loop of $\textsc{IsWellFormed}(s)$ terminates (without returning early) in a configuration with an empty stack if and only if $s$ is well-formed.

We can prove this by induction on the length of $s$.

**Base case:** If $s$ has length `0`, then it is empty. Then $s$ is well-formed and $\textsc{IsWellFormed}(s)$ indeed terminates immediately with an empty stack.

**Induction hypothesis:** Let $n > 0$. Assume that for all $s$ of length $|s| \leq n - 1$, the for loop of $\textsc{IsWellFormed}(s)$ terminates (without returning early) in a configuration with an empty stack if and only if $s$ is well-formed.

**Induction step:** Let $s$ be a well-formed string of length $s$. First, assume that $s$ is well-formed. There are three cases:

- If $s$ is of the form $ab$ with $0 < |a|, |b| < |s|$, then by our induction hypothesis the for loop of $\textsc{IsWellFormed}(a)$ and $\textsc{IsWellFormed}(b)$ terminates in a configuration with an empty stack. When running $\textsc{IsWellFormed}(s)$, the first $|a|$ are exactly the same as in $\textsc{IsWellFormed}(a)$, and we end up with an empty stack after $|a|$ iterations. Then, we run exactly the same $|b|$ steps as in $\textsc{IsWellFormed}(b)$, ending up again with an empty stack. We successfully return `True`.
- If $s$ if of the form `(`$a$`)`, then running $\textsc{IsWellFormed}(s)$ first pushes "`(`" onto the stack, and then runs the same steps (from iterations $1$ to $|s|$) as in $\textsc{IsWellFormed}(a)$, but with the additional "`(`" element remaining at the bottom of the stack. By our induction hypothesis, the stack contains only "`(`" after iteration $|s|$, after which iteration $|s|+1$ removes "`(`" from the stack and terminates successfully.
- The argument is similar for $s =$ `[`$a$`]`.

Conversely, assume that $\textsc{IsWellFormed}(s)$ returns `True`. We distinguish between two cases:

- If $S$ is empty in some intermediate iteration $i \in \{1, \ldots, |s| - 1\}$, consider such an $i$. Then the successful execution of $\textsc{IsWellFormed}(s)$ is exactly the concatenation of two successful executions of $\textsc{IsWellFormed}(s[0..i])$ and $\textsc{IsWellFormed}(s[i + 1..|s| - 1])$. Hence, by our induction hypothesis, $s[0..i]$ and $s[i + 1..|s| - 1]$ are well-formed, and their concatenation $s$ is also well-formed.
- If $S$ is never empty in any intermediate iteration, then we observe that the first element pushed onto the stack is never popped before the very last iteration. For this last pop to be succeed, the first and last character of $s$ must be matching brackets (i.e., `()` or `[]`). Moreover, as the element at the bottom of the stack is never popped and the final stack is empty, iterations $1$ to $|s| - 2$ must be exactly identical to a successful execution of $\textsc{IsWellFormed}(s[1..|s|-2])$. Hence, by our induction hypothesis, $s[1..|s|-2]$ is a well-formed string, and so is $s$ which is either `(`$s[1..|s|-2]$`)` or `[`$s[1..|s| - 2]$`]`.

**Remark.** The above constitutes a formal proof of the correctness of the algorithm, provided for the sake of completeness. A more informal argument shall also be counted as correct.

**Complexity.** Each iteration of the for loop has runtime complexity $O(1)$: stack operations are $O(1)$, and we execute at most one such operation per iteration, along with at most 5 constant-time tests and at most one constant-time return statement. As there are $|s|$ iterations in total and the rest of the operations are constant-time, we get a total runtime complexity in $O(|s|)$.

**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.

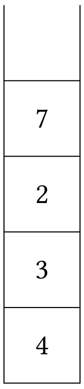


Figure 3: A stack $S_0$ containing the numbers 4, 3, 2, and 7 (7 is the top of the stack)

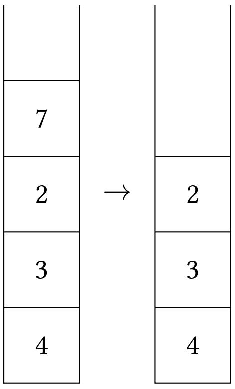
In this exercise, we focus on the second phase, and consider the following three basic operations used to compute with stacks:

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

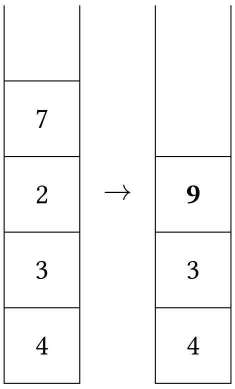
ADD: If there are at least two elements in the stack, remove the top two elements, compute their sum, and push this sum back into the stack. If there is less than two elements in the stack, do nothing.

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.

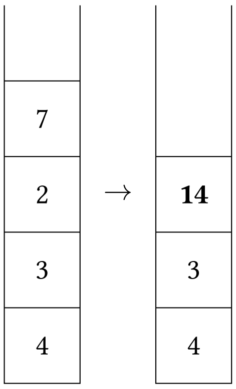
Below are examples of applications of POP, ADD, and MUL.



(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:

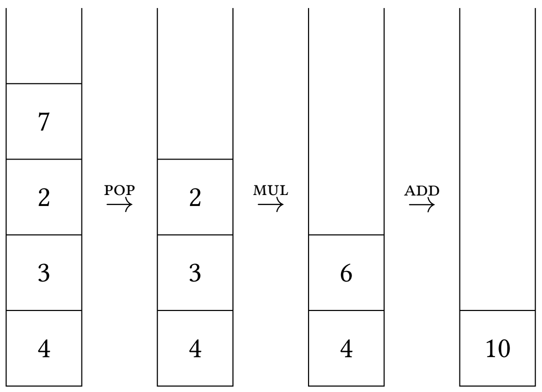


Figure 5: Computing 10 from $S_0$

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?

**Solution:**

For $i \in \{1, \ldots, n\}$, we denote by $S[1..i]$ the stack containing the top $i$ elements of $S$.

1. *Dimensions of the DP table*: $DP[1 \ldots c][1 \ldots n]$
2. *Definition of the DP table*: $DP[i][j]$ is True if, and only if, $i$ can be computed from the stack $S[1..j]$ *and the stack produced by the computation contains only one element in the end*, and False otherwise.
3. *Computation of an entry*: $DP$ can be computed recursively as follows:

$$\begin{aligned}
&DP[i][1] = (i == S_1) && \\
&DP[i][j] = \texttt{False} && j > 1, S_j > i \\
&DP[i][j] = (i == S_j) \texttt{ or } DP[i - S_j, j - 1] && j > 1, S_j \leq i, S_j \nmid i \\
&DP[i][j] = (i == S_j) \texttt{ or } DP[i - S_j, j - 1] \texttt{ or } DP[i/S_j, j - 1] && j > 1, S_j \mid i.
\end{aligned}$$

The three factors of the disjunction in the last equation correspond to the three possible cases in which $i$ can be computed from $S[1..j]$, leaving a singleton stack in the end:

(a) By popping all of $S[1..j-1]$ and returning $S_j = i$ (case $i = S_j$),

(b) By computing $i - S_j$ from $S[1..j-1]$, and then performing ADD (case $DP[i - S_j, j-1]$) between $i - S_j$ (which is now on top of the stack) and $S_j$,

(c) By computing $i/S_j$ from $S[1..j-1]$, and then performing MUL (case $DP[i - S_j, j-1]$) between $i/S_j$ (which is now on top of the stack) and $S_j$.

The second case is only possible if $S_j \leq i$, the last if $S_j$ is a divisor of $i$.

Note that since all numbers in the stack are positive, all intermediate values obtained during the computation of $c$ must be contained in $\{1, \ldots, c\}$. Hence, considering only $i \in \{1, \ldots, c\}$ is sufficient.

4. *Calculation order*: Following the recurrence relations above, we can compute first by order of increasing $j$, and then by order of increasing $i$.
5. *Extracting the solution*: The solution is $DP[c][1]$ `or` ... `or` $DP[c][n]$.
6. *Running time*: The running time of the solution is $O(c \cdot n + n) = O(c \cdot n)$ as there are $c \cdot n$ entries in the table, we process each entry in $O(1)$ time, and the solution is computed in $O(n)$ time.

**Remark.** In practice, a solution based on memoization might be more efficient for this problem, given that many values of $i$ may not be computable from the $S_k$.

(*) *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.

**Solution:**

With the NEG operation at our disposal, a naïve approach would consist in adding a disjunct

$$\ldots \texttt{or}\ DP[-i, j]$$

in the recursion. But this would break the calculation order, since entries with the same value of $j$ (concretely, all $DP[i,j]$ and $DP[-i,j]$) would depend on each other. We observe, however, that $i$ can be computed if and only if $-i$ can be computed (just apply NEG). Hence, we can change the definition of our table to be "$DP[i][j]$ is True if, and only if, $i$ or $-i$ can be computed from the stack $S[1..j]$, and False otherwise", and instead add only a case $DP[i + S_j, j-1]$ to the disjunction, corresponding to an application of NEG followed by an application of ADD. The last thing that needs to be changed is the size of the first dimension of the table, since intermediate results can now be larger than $i$. A safe bound for intermediate results is $k^n$.