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

12. November 2018

Algorithms & Data Structures — Homework 8 — HS 18

Exercise Class (Room & TA): _____

Submitted by: _____

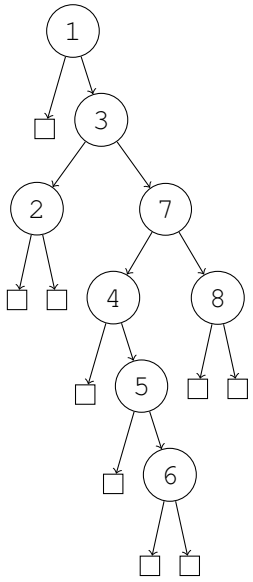
Peer Feedback by: _____

Points: ____

Exercise 8.1 *Search Trees.*

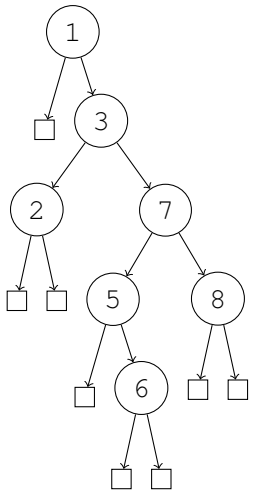
1. Draw the resulting tree when the keys 1,3,7,4,5,8,6,2 in this order are inserted into an initially empty natural search tree.

   Solution:

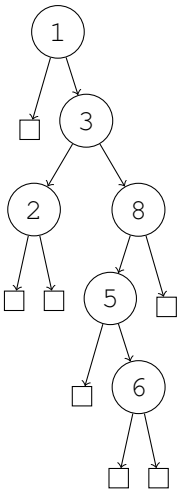


2. Delete key 4 in the above tree, and afterwards key 7 in the resulting tree.

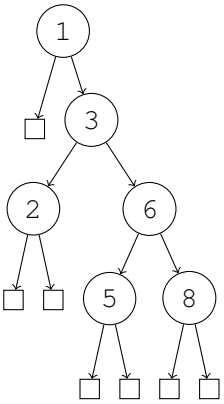
   Solution: Key 4 has one child, so it can just be replaced by 5:



Key 7 must either be replaced by its sucessor key, 8, or its precesessor key, 6. If key 7 is replaced by its sucessor:



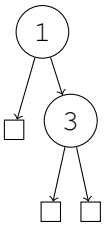
If key 7 is instead replaced by its predecessor:



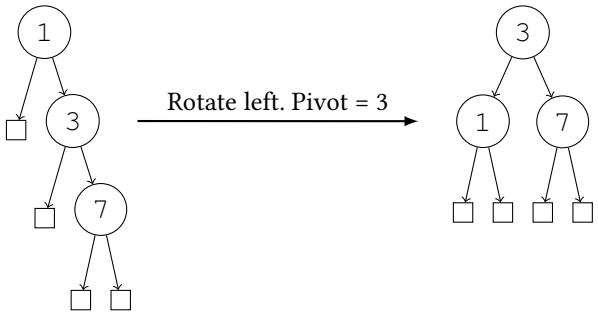
3. Draw the resulting tree when the keys are inserted into an initially empty AVL tree.

   Solution:

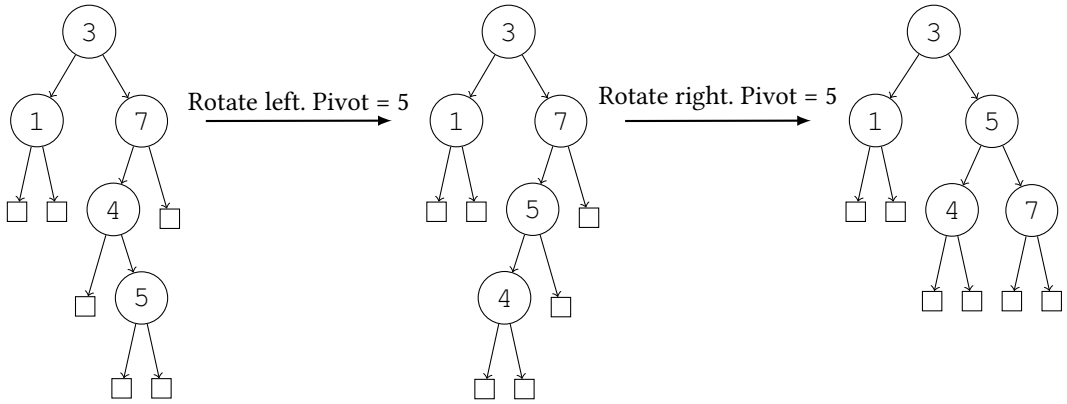
   Insert 1 and then 3:



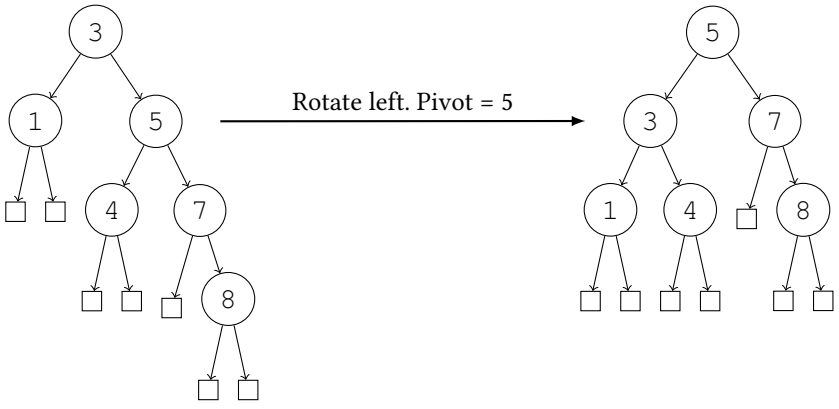
Insert 7:



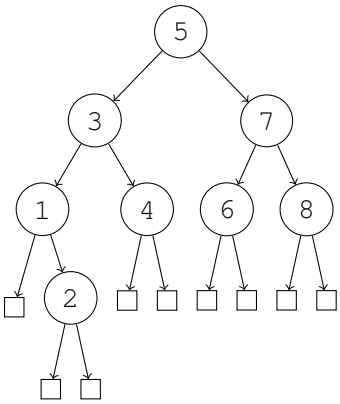
Insert 4 and then 5:



Insert 8:



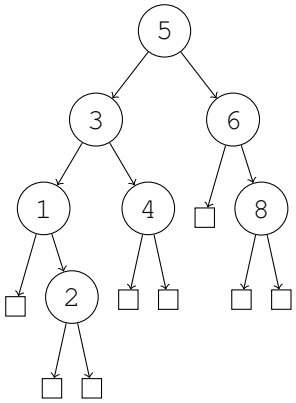
Insert 6 and 2:



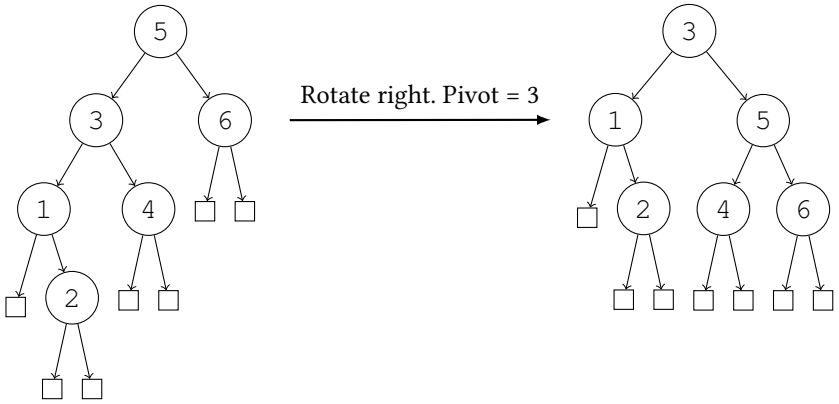
4. Delete key 7 in the above tree, and afterwards key 8 in the resulting tree.

   Solution:

   Delete 7:



   Delete 8:



Exercise 8.2 *Tree Traversals.*

There are three essential ways to traverse binary trees. The first one is Preorder(T), which at first visits the root v, then $T_l(v)$ and then $T_r(v)$, where $T_l(v)$ is the left subtree of v and $T_r(v)$ is the right subtree of v. The second one is Postorder(T), which at first visits $T_l(v)$, then $T_r(v)$ and then v. The third one is Inorder(T), which at first visits $T_l(v)$, then v and then $T_r(v)$.

In each case the left and right subtrees are visited recursively in the same order.

1. Consider this pseudocode for the Preorder procedure:

Algorithm 1: Preorder(T)

```
1 if T is non-empty then
2 |   v ← Root(T);
3 |   Visit(v);
4 |   Preorder(T_l(v));
5 |   Preorder(T_r(v));
6 end
```

Write pseudocodes for Postorder and Inorder procedures.

Solution:

Algorithm 2: Postorder(T)

```
1 if T is non-empty then
2 |   v ← Root(T);
3 |   Postorder(T_l(v));
4 |   Postorder(T_r(v));
5 |   Visit(v);
6 end
```

Algorithm 3: Inorder(T)

```
1 if T is non-empty then
2 |   v ← Root(T);
3 |   Inorder(T_l(v));
4 |   Visit(v);
5 |   Inorder(T_r(v));
6 end
```

2. For the above search trees in 8.1.1 and 8.1.3 give the Preorder, the Postorder, the Inorder of the nodes.

Solution:

For the tree in 8.1.1:

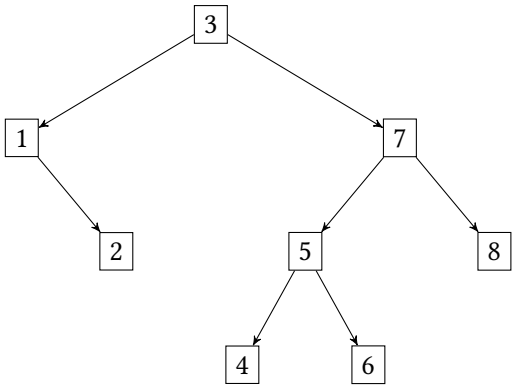
- Preorder: $1, 3, 2, 7, 4, 5, 6, 8$.
- Postorder: $2, 6, 5, 4, 8, 7, 3, 1$.
- Inorder: $1, 2, 3, 4, 5, 6, 7, 8$.

For the tree in 8.1.3:

- Preorder: $5, 3, 1, 2, 4, 7, 6, 8$.
- Postorder: $2, 1, 4, 3, 6, 8, 7, 5$.
- Inorder: $1, 2, 3, 4, 5, 6, 7, 8$.

3. Draw the binary tree with keys $1, 2, 3, 4, 5, 6, 7, 8$ such that the Preorder starts with $3, 1, 2, 7$ and the Postorder ends with $5, 8, 7, 3$.

Solution:



Exercise 8.3 *Advanced Search Trees **(1 Point)**.*

In this exercise, we wish to extend the functionality of a search tree. We consider a binary search tree over integers. In addition to finding a number, we want to be able to answer the following questions:

1. How many elements in the tree are *multiples of 3* and greater than a given number k?
2. How many elements in the tree are *multiples of 3* and (strictly) between two given numbers k_1 and k_2, with $k_1 < k_2$?

Discuss how you can modify the tree so that you can answer these questions efficiently. Describe how the insertion and the removal operation must be changed accordingly. Include a discussion of the running times of the modified algorithms.

Solution In addition to the value of the key, we store at each node v of the tree the number g_v of elements that are multiples of 3 and are in the right subtree of v.

When asked for the number of elements that are multiples of 3 and greater than k, we look for the element k in the tree. Every time we go to the left subtree of a node v (in the case where k is smaller than the key in v), we increment a counter by g_v (initially the counter is zero), or by $g_v + 1$ if v is a multiple of 3 and greater than k. If we go to the right subtree, we keep the counter unchanged. If we find k in a node v, then we add one last time g_v to the counter. Otherwise, we end up in a leaf v of the tree. Then, if the key of v is smaller or equal to k, the counter is already set to the correct number. Otherwise, we increase the counter by one if the key in v is a multiple of 3.

To answer how many numbers are multiple of 3 and between k_1 and k_2 in the tree, we need to determine only the values L_1, L_2 of elements that are multiple of 3 and larger than k_1 and k_2 respectively. We have just seen how to find these values. Subtracting L_2 from L_1, we obtain how many such numbers x such that $k_1 < x \leq k_2$ are in the tree. If k_2 exists in the tree and is a multiple of 3, we thus have to remove one from this number. We can easily check for this case when computing L_2.

The insertion of an element i works as follows. We first search for i. If it is already in the tree, we do nothing. If i is not a multiple of 3, we insert it in the leaf where the search ended and we set $g_i \leftarrow 0$. If i is a multiple of 3, we additionally traverse the tree again (i.e., search for i again) and whenever we meet a node v containing a smaller key (i.e., we "turn right"), we increase g_v by one since i is inserted in its right subtree.

The removal of an element i works as follows. Again, we first search for i. If it is not in the tree, we do nothing. Otherwise we continue in i to find its symmetrical predecessor n. We will later replace i by n. First, we need to update all counters. If i is a multiple of 3, we traverse the tree again from the root to i (i.e., search for i), and whenever we "turn right" in a node v, we decrement the value g_v. Similarly,

if n is a multiple of 3, we decrement the counters on the path from i to n at every „right turn“. Due to the properties of the symmetric predecessor, it holds that $g_n = 0$. We now set n in the old position of i. Furthermore, we set $g_n \leftarrow g_i$, since n was originally in the left subtree of i.

All new tree operations require an additional effort that is constant in every visited node, therefore there is no impact on the asymptotic running time of Insert, Search and Remove.

Exercise 8.4 *Maximum Depth Difference of two Leaves.*

Consider an AVL tree of height h. What is the maximum possible difference of the depths of two leaves? Imaging which structure such trees need to have, and draw examples of corresponding trees for every $h \in \{2, 3, 4\}$. Derive a recursive formula (depending on h), solve it and use induction to prove the correctness of your solution. Provide a detailed explanation of your considerations.

Note: For the proof the principle of *complete induction* can be used. Let $\mathcal{A}(n)$ be a statement for a number $n \in \mathbb{N}$. If, for every $n \in \mathbb{N}$, the validity of *all* statements $\mathcal{A}(m)$ for $m \in \{1, \ldots, n-1\}$ implies the validity of $\mathcal{A}(n)$, then $\mathcal{A}(n)$ is true for every $n \in \mathbb{N}$.

$$\Big(\forall n \in \mathbb{N} : \big(\forall m \in \{1, \ldots, n-1\} : \mathcal{A}(m)\big) \Rightarrow \mathcal{A}(n)\Big) \Rightarrow \forall n \in \mathbb{N} : \mathcal{A}(n). \tag{1}$$

Thus, complete induction allows multiple base cases and inductive hypotheses.

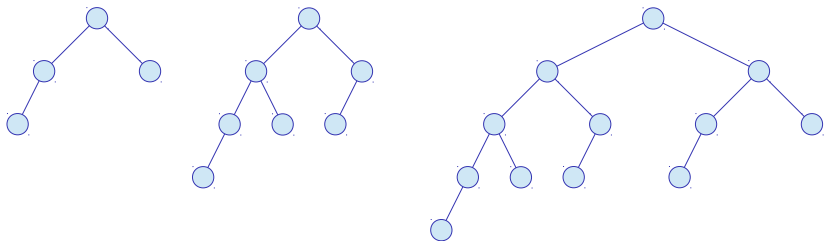
Solution:

For an AVL-tree T with a root node v and height h, we can distinguish the following 3 cases:

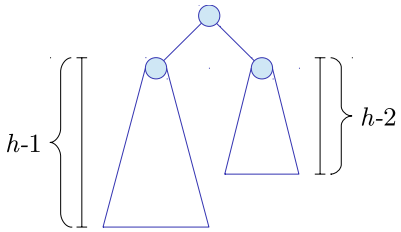
- Both sub-trees $T_l(v)$ and $T_r(v)$ have height $h-1$
- $T_l(v)$ has height $h-1$ and $T_r(v)$ has height $h-2$, or
- $T_l(v)$ has height $h-2$ and $T_r(v)$ has height $h-1$

As we are interested in the *maximum* depth difference of two leaves, we can disregard the first case, and focus on sub-trees that have heights that differ by 1. Without loss of generality, we can take the second case, assuming that the left sub-tree will have height of $h-1$, while the right sub-tree will have height of $h-2$. If the left sub-tree is an AVL-tree of height $h-1$, then the right tree must be an AVL-tree of height $h-2$. This comes from the properties of an AVL tree, because if at any time they differ by more than one, rebalancing is done to restore this property. As a result of this, the entire tree T will have a height of h and as such there will be a leaf on the left-subtree with this depth.

The figure below illustrates the AVL trees of height $h \in \{1, 2, 3\}$:



In general, we consider trees with the following structure:



The left subtree $T_l(v)$ contains a leaf of depth h (while $T_l(v)$ has height of $h-1$), the right subtree $T_r(v)$ contains a leaf of depth $h-1$ (while $T_r(v)$ has height $h-2$). The maximum possible difference of the depths of two leaves in the tree (with height h) is therefore 1 greater than the maximum difference of the depths of two leaves in the right subtree (with height $h-2$). For $h=2$ and $h=3$, the maximum depth difference is exactly 1.

As a result, we have the following recursive formula for the maximum difference of the depths of two leaves in a tree of height h:

$$D(2)=1,\ D(3)=1,\ D(h)=1+D(h-2) \text{ for all } h\geq 4. \tag{2}$$

From the above, we can assume that $D(h)=\lfloor h/2\rfloor$. We prove this assumption using induction over h.

Base case I ($h=2$): $D(2)=1=\lfloor 2/2\rfloor$.

Base case II ($h=3$): $D(3)=1=\lfloor 3/2\rfloor$.

Induction hypothesis: Assume that the property holds for some h: $D(h-2)=\lfloor (h-2)/2\rfloor$.

Inductive step: $((h-2)\to h)$: From the recursive definition of $D(h)$, we have:

$$D(h)=1+D(h-2)=1+\lfloor (h-2)/2\rfloor=1+\lfloor h/2-1\rfloor=1+\lfloor h/2\rfloor-1=\lfloor h/2\rfloor. \tag{3}$$

Exercise 8.5 *One-Column Candy Crush* ***(2 Points.)***.

Consider a column of n candies, and assume that if three or more *adjacent* candies in this column are equal, then these candies can be removed, and the column shrinks. Removing the candies is done with the following tie breaking rule: The candies are removed from top to bottom, i.e., when there is more than one group that can be removed at the same time, the upper group is removed first. We are interested in the number of candies that can not be removed.

You get this column of candies stored on a stack S, such that the first (topmost) candy of the column is on the top of this stack. Recall that the operations on a stack are `top()`, `pop()`, and `push(`v`)` where v is a candy. Moreover, you can access the number of candies in the stack. Assume that all these operations require constant time.

Your task is to design a linear time algorithm that returns the number of the remaining candies. Your algorithm is allowed to use stack S plus one additional stack T, for which you can assume that it is initially empty. On top of that, you are allowed to use only $O(1)$ extra memory space. Provide an analysis of the actual running time of your algorithm.

Solution: We assume that $n>2$, otherwise the answer is trivial. We start with an observation: whenever the topmost group of candies $e,e,\ldots,e$ is removed from the column, then either this removal

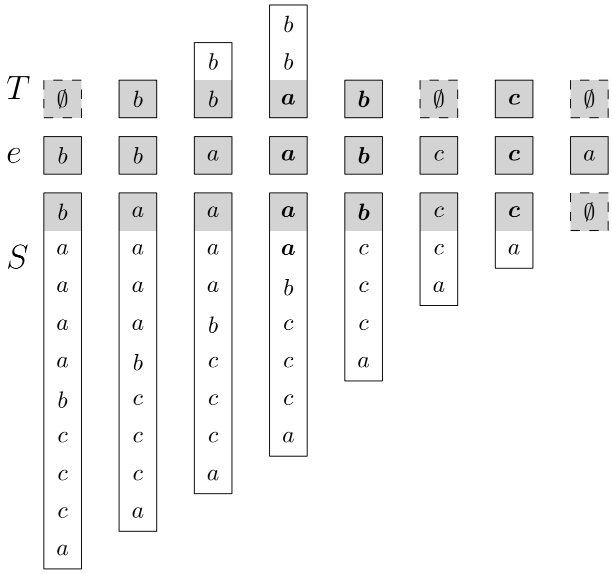


Figure 1: Example for the execution of the algorithm.

leads to three or more other equal candies $e', e', \dots, e'$ that are now adjacent but were split before into two groups by the es, or the next topmost group to be removed is further down in the column.

The idea of our algorithm is as follows: Split the column into tree parts, S, e, and T, where e is a single candy and S and T are stacks. We will keep the following invariants:

- No group of candies in T plus e can be removed
- The remaining column is given by the candies in T backwards, e, and the candies in S forwards
- $|S| > 0$

Initially, the first candy of S is popped into e. Then, intuitively the following step is repeated until S is empty.

1. Check whether the top candies in S and T (if they exists) are both equal to e.
2. If no, then e does not belong to the next group to remove. Therefore, push e to T and pop the the top candy of S into e.
3. If yes, then you have found the topmost group of candies to remove. Remove thus all equal candies on the top of T and S. Replace the candy in e with the top candy of T, or if no such candy exists with the top candy of S.

When the algorithm terminates, all candies that cannot be removed are in T or e. Consider the pseudocode in Algorithm 4. An example is given in Figure 1.

The running time of the algorithm is as follows: in each step (each iteration in the outer while loop), the algorithm does $x \geq 1$ pops from S (and no pushes), and a constant number of other operations. So the total number of pops from S is n and the number of other operations is $O(\#\text{of steps})$ which is also at most n (since when $|S| = 0$ the algorithm terminates). Therefore, the running time of the algorithm is in $\Theta(n)$.

Submission: On Monday, 19.11.2018, hand in your solution to your TA *before* the exercise class starts.

Algorithm 4: One-Column_Candy_Crush(S)

```
e = S.pop();
while |S| > 0 do
    if (|T| > 0 and e == T.top() and e == S.top()) then
        //Remove the matching candy from T (There can only be one!!! WHY?)
        T.pop();
        //Remove the matching candies from S
        while (|S| > 0 and S.top() == e) do
            S.pop();
        end
        //Find new middle candy e
        if (|T| > 0) then
            e = T.pop();
        else if (|S| > 0) then
            e = S.pop();
        else
            //Out of candies
            return 0;
        end
    else
        //e cannot be removed now, move it to T
        T.push(e);
        e = S.pop();
    end
end
//No more candies can be removed
return 1 + |T|;
```