Algorithms and Data Structures

Exercise Session 7

https://n.ethz.ch/~ahmala/and

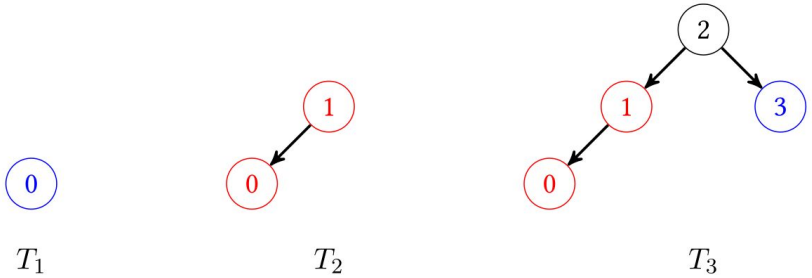
Quiz

Past Exercise Sheet

Exercise 6.1 *Fibonacci trees* **(1 point)**.

For $k \in \mathbb{N}$, we define the *Fibonacci tree* T_k recursively:

- The trees T_1 and T_2 are binary search trees with 1 and 2 nodes respectively, as depicted below.
- For $k \geq 3$, the tree T_k is constructed as follows. We start with a root node with key $\mathrm{Fib}(k+1)-1$. Then, we add as the left subtree of this root node the tree T_{k-1}. Finally, we add as a right subtree the tree T_{k-2}, but with all keys increased by $\mathrm{Fib}(k+1)$.



Note: To achieve full points for this exercise, it is enough to submit parts **(e)** and **(f)**. The other parts are **not worth any points**. When solving a part, you are allowed to use any earlier parts, even if you did not solve them.

(e) Show that, in fact, for any $k \in \mathbb{N}$, and any node u in T_k that is not a leaf, we have $|h_l(u) - h_r(u)| = 1$.

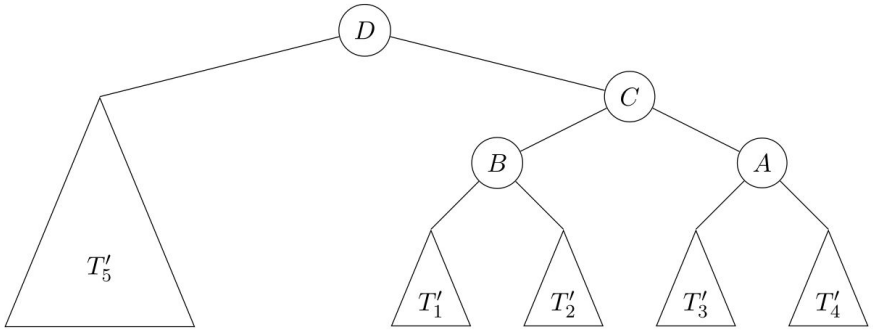
Hint: *Use induction. You will need to show multiple base cases. Be careful how you formulate the induction hypothesis.*

(f) Let $k \geq 3$ and odd. Show that the tree T_k contains a leaf at depth exactly $(k-1)/2$. (Here, the depth of a node is defined as the number of predecessors it has in the tree).

Hint: *Draw the tree* T_5. *Identify which leaf is at depth* $(5-1)/2 = 2$.

Exercise 6.2 *AVL Deduction.*

Let T be an AVL tree and suppose we performed one node insertion to T to get T' (we haven't rebalanced yet). Only partial information about T' is given. The graph of T' looks like:



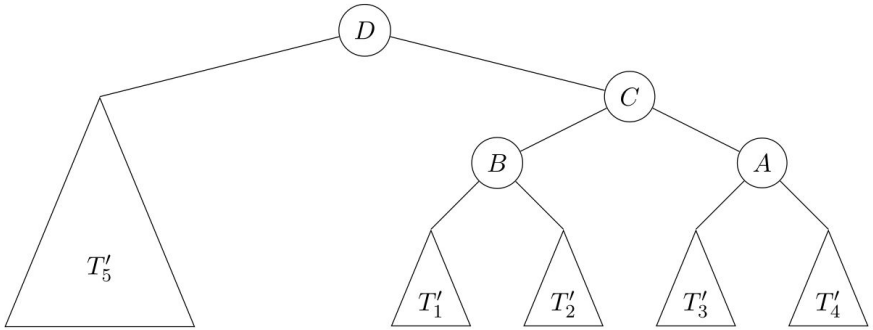
where $T'_1, T'_2, \ldots, T'_5$ are all subtrees whose nodes satisfy the AVL condition. Note that their leaves are not necessarily at the same level.

The left and right subtree heights of B are both 1 and the left and right subtree heights of A are 2 and 3 respectively. Furthermore assume that D does not satisfy the AVL condition.

(a) What are the heights of the left and right subtrees of C?

Exercise 6.2 *AVL Deduction.*

Let T be an AVL tree and suppose we performed one node insertion to T to get T' (we haven't rebalanced yet). Only partial information about T' is given. The graph of T' looks like:



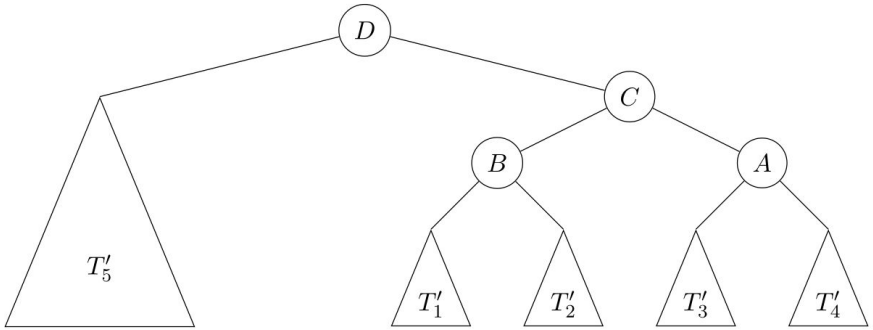
where $T_1', T_2', \ldots, T_5'$ are all subtrees whose nodes satisfy the AVL condition. Note that their leaves are not necessarily at the same level.

The left and right subtree heights of B are both 1 and the left and right subtree heights of A are 2 and 3 respectively. Furthermore assume that D does not satisfy the AVL condition.

(b) Which subtree was the node inserted in?

Exercise 6.2 *AVL Deduction.*

Let T be an AVL tree and suppose we performed one node insertion to T to get T' (we haven't rebalanced yet). Only partial information about T' is given. The graph of T' looks like:



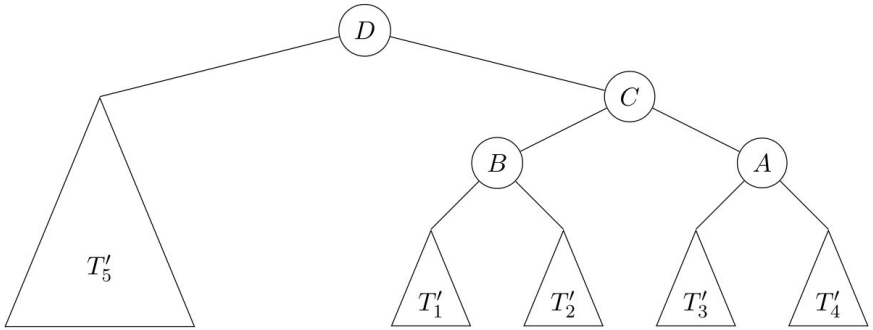
where $T'_1, T'_2, \ldots, T'_5$ are all subtrees whose nodes satisfy the AVL condition. Note that their leaves are not necessarily at the same level.

The left and right subtree heights of B are both 1 and the left and right subtree heights of A are 2 and 3 respectively. Furthermore assume that D does not satisfy the AVL condition.

(c) What are the heights of the left and right subtrees of D?

Exercise 6.2 *AVL Deduction.*

Let T be an AVL tree and suppose we performed one node insertion to T to get T' (we haven't rebalanced yet). Only partial information about T' is given. The graph of T' looks like:

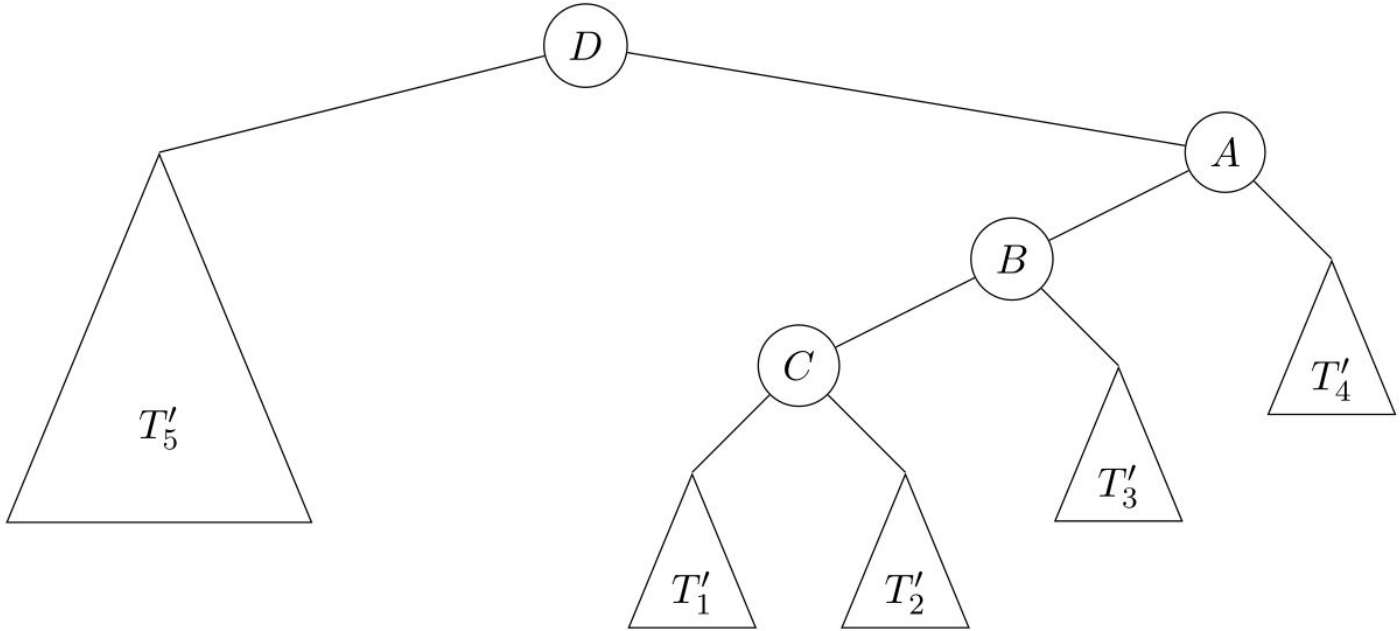


where $T'_1, T'_2, \ldots, T'_5$ are all subtrees whose nodes satisfy the AVL condition. Note that their leaves are not necessarily at the same level.

The left and right subtree heights of B are both 1 and the left and right subtree heights of A are 2 and 3 respectively. Furthermore assume that D does not satisfy the AVL condition.

(d) Draw the tree after the necessary single or double rotation to rebalance T' and restore the AVL condition for all nodes. Also note the new left and right subtree heights for nodes A, B, C and D.

Solution:



For A, left: 3, right: 3.
For B, left: 2, right: 2.
For C, left: 1, right: 1.
For D, left: 3, right: 4.

Exercise 6.3 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$
\begin{aligned}
A_1 &= 1 \\
A_2 &= 2 \\
A_{2n+1} &= \frac{1}{2}(A_{2n} + A_{2n-1}) \\
A_{2n+2} &= 2\left(\frac{1}{A_{2n}} + \frac{1}{A_{2n-1}}\right)^{-1} \quad \text{for } n \geq 1.
\end{aligned}
$$

(a) Provide a recursive function (using pseudo code) that computes A_n for $n \in \mathbb{N}$. You do not have to argue correctness.

Algorithm 1 $A(n)$

if $n \leq 2$ **then**
  return n
else if n is odd **then**
  return $\frac{1}{2}(A(n-1) + A(n-2))$
else
  return $2\left(\frac{1}{A(n-2)} + \frac{1}{A(n-3)}\right)^{-1}$

(b) Lower bound the run time of your recursion from (a) by $\Omega(C^n)$ for some constant $C > 1$.

(c) Improve the run time of your algorithm using memoization. Provide pseudo code of the improved algorithm and analyze its run time.

(d) Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases 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:

- **Dimensions of the DP table:** The DP table is linear, its size is n.
- **Subproblems:** $DP[k]$ contains A_k for $1 \leq k \leq n$.
- **Recursion:** Initialize $DP[1]$ to 1, $DP[2]$ to 2. The entries with $k \geq 3$ are computed by $DP[k] = \frac{1}{2}(DP[k-1]+DP[k-2])$ for k odd and $DP[k] = 2/(1/DP[k-1]+1/DP[k-2])$ for k even (this is correct by the recursive definition of $A[k]$).
- **Calculation order:** We can calculate the entries of DP from smallest to largest.
- **Extracting the solution:** All we have to do is read the value at $DP[n]$.
- **Running time:** Each entry can be computed in time $\Theta(1)$, so the run time is $\Theta(n)$.

Exercise 6.4 *Maximum almost subarray sum* **(1 point)**.

The maximum subarray sum problem from the lecture asks for the sum of the largest *contiguous* subarray in a given array. The maximum almost subarray sum problem asks instead for the sum of the largest contiguous subarray with one element possibly missing from inside this subarray.

We consider the following array of length $n = 10$.

$$A[1..n] = [3, 2, -2, 1, -2, -3, 4, 1, -3, 4]$$

In this exercise we'll take the tools from the solution of maximum subarray sum and extend it to two different methods of computing this maximum almost subarray sum.

(a) The dynamic programming solution for maximum subarray sum given in the lecture revolves around the numbers

$$R[k] := \text{maximum subarray sum of } A[1..k] \text{ which includes index } k$$

for $0 \leq k \leq n$, where we define $R[0] = 0$. Then to get the maximum subarray sum we output $\max_k R[k]$.

In lecture we saw that this array R satisfies the recursive relation

$$R[0] = 0$$
$$R[k] = \max\{A[k], A[k] + R[k-1]\} \text{ for } 1 \leq k \leq n$$

Compute the array for $1 \leq k \leq n = 10$.

(b) Define the following modification of R

$$R'[k] := \text{maximum subarray sum of } A[1..k] \text{ which includes index } k \text{ but skips an entry.}$$

Assume that the following recursive relation is correct

$$\begin{aligned} R'[0] &= 0 \\ R'[1] &= 0 \\ R'[2] &= 0 \\ R'[k] &= \max\{A[k] + R'[k-1], A[k] + R[k-2]\} \text{ for } 3 \leq k \leq n. \end{aligned}$$

Compute $R'[k]$ for $1 \leq k \leq n = 10$. How can the final solution for the maximum almost subarray sum of A be extracted from R and R'? Write down the formula as a function of the entries of these two arrays.

(a) Given are the two arrays

$$A = [7, 6, 3, 2, 8, 4, 5, 1]$$

and

$$B = [3, 9, 10, 8, 7, 1, 2, 6, 4, 5].$$

Use the dynamic programming algorithm from the lecture to find the length of a longest common subsequence and the subsequence itself. Show all necessary tables and information you used to obtain the solution.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1	2	2	2
3	0	1	1	1	1	1	1	1	2	2	2
4	0	1	1	1	1	1	1	2	2	2	2
5	0	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	2	2	2	2	2	3	3
7	0	1	1	1	2	2	2	2	2	3	4
8	0	1	1	1	2	2	3	3	3	3	4

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1	2	2	2
3	0	1	1	1	1	1	1	1	2	2	2
4	0	1	1	1	1	1	1	2	2	2	2
5	0	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	2	2	2	2	2	3	3
7	0	1	1	1	2	2	2	2	2	3	4
8	0	1	1	1	2	2	3	3	3	3	4

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1	2	2	2
3	0	1	1	1	1	1	1	1	2	2	2
4	0	1	1	1	1	1	1	2	2	2	2
5	0	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	2	2	2	2	2	3	3
7	0	1	1	1	2	2	2	2	2	3	4
8	0	1	1	1	2	2	3	3	3	3	4

Extra Exercise

Let A be an AVL tree (as described in the lecture) with n nodes. Let $k_1 < k_2 < \ldots < k_n$ be the keys of A, in ascending order. For a given $1 \leq i \leq n$, our goal is to find k_i, the i-th smallest key of A.

(a) Suppose $i = 1$. Describe an algorithm that finds k_1 in $O(\log n)$ time.

Hint: *An AVL tree is a BST (binary search tree).*

Solution:

Since A is a BST, we know that for each node v with key $\text{key}(v)$, the keys of its left subtree are all smaller than $\text{key}(v)$, while the keys of its right subtree are all greater than $\text{key}(v)$. It follows that k_1 can be found by starting at the root node, and then repeatedly moving to the left child, until we arrive at a node without a left child. As A is an AVL tree, it has depth $O(\log n)$, and so this procedure takes time $O(\log n)$.

(b) Describe an algorithm that finds k_i in $O(i \cdot \log n)$ time.

Hint: *You are allowed to make changes to A while executing your algorithm.*

(b) Describe an algorithm that finds k_i in $O(i \cdot \log n)$ time.

Hint: *You are allowed to make changes to A while executing your algorithm.*

It turns out that we can find k_i in time $O(\log n)$, if we modify the definition of an AVL tree a bit.

(c) Modify the definition of an AVL tree by storing two additional integers $s_l(v), s_r(v) \in \mathbb{N}$ in each node v. Assuming now that A satisfies your modified definition, describe an algorithm that finds k_i in $O(\log n)$ time.

Remark. Your modified definition should still allow for the search, insert and remove operations to be performed in $O(\log n)$ time, but you are not required to prove that this is the case.

The additional integers we store are the sizes $0 \leq s_l(v), s_r(v) \leq n$ of the left and right subtree rooted at the left and right child of v, respectively. (This information can be updated in time $O(1)$ during the rebalancing rotations performed during the insert and remove operations). Assuming A is modified so that each node contains these integers, our algorithm to find k_i proceeds as follows.

Let v_0 be the root node of A. Set $i_0 = i$. We want to find the i_0-th smallest key in the subtree rooted at v_0 (which is just A). We consider the following three cases:

(i) If $s_l(v_0) = i_0 - 1$, we know that there are precisely $i_0 - 1$ keys in the subtree rooted in v_0 that are smaller than $\mathrm{key}(v_0)$; namely the keys in the subtree rooted in the *left child* of v_0. But that means that $\mathrm{key}(v_0)$ is the i_0-th smallest key in the subtree rooted in v_0, and so we output $\mathrm{key}(v_0)$.

(ii) If $s_l(v_0) > i_0 - 1$, the i_0-th smallest key lies in the subtree rooted in the *left* child $\mathrm{lc}(v_0)$ of v_0.

(iii) If $s_l(v_0) < i_0 - 1$, the i_0-th smallest key lies in the subtree rooted in the *right* child $\mathrm{rc}(v_0)$ of v_0.

Assuming we are in case (ii) or (iii), the idea is to proceed recursively by applying the same procedure to the subtree rooted in the left (resp. right) child of v_0.

In case (ii), note that the i_0-th smallest key of the subtree rooted in $\mathrm{lc}(v_0)$ is equal to the i_0-th smallest key of A (since all keys in the right subtree are too large). In this case, we can thus set $v_1 = \mathrm{lc}(v_0)$ and $i_1 = i_0$, and apply the procedure above.

In case (iii), things are slightly more complicated. Note that all $s_l(v_0)$ keys in the subtree rooted at $\mathrm{lc}(v_0)$ are smaller than $\mathrm{key}(v_0)$, and that $\mathrm{key}(\mathrm{rc}(v_0)) > \mathrm{key}(v_0)$. That is to say, if we set

$$i_1 = i_0 - (s_l(v_0) + 1),$$

then the i_1-th smallest key in the subtree rooted in $v_1 = \mathrm{rc}(v_0)$ is precisely the i_0-th smallest key in the subtree rooted in v_0 (which is what we are after).

We now repeat this procedure until we reach case (i). Each repetition takes $O(1)$ time. We move one layer down in each repetition. If we ever reach a leaf, we are certainly in case (i). Therefore, the whole algorithm takes at most $O(\log n)$ time (recall that A has depth $O(\log n)$).

It turns out that we can find k_i in time $O(\log n)$, if we modify the definition of an AVL tree a bit.

(c) Modify the definition of an AVL tree by storing two additional integers $s_l(v), s_r(v) \in \mathbb{N}$ in each node v. Assuming now that A satisfies your modified definition, describe an algorithm that finds k_i in $O(\log n)$ time.

Remark. Your modified definition should still allow for the search, insert and remove operations to be performed in $O(\log n)$ time, but you are not required to prove that this is the case.

```
s_l(v): number of nodes in the left subtree of v
s_r(v): number of nodes in the right subtree of v

g(node, i):
    if s_l(node) + 1 == i
        return node
    if i < s_l(node) + 1
        return g(node.left, i)
    if i > s_l(node) + 1
        return g(node.left, i - (s_l(node) + 1))
```

Dynamic Programming on Trees

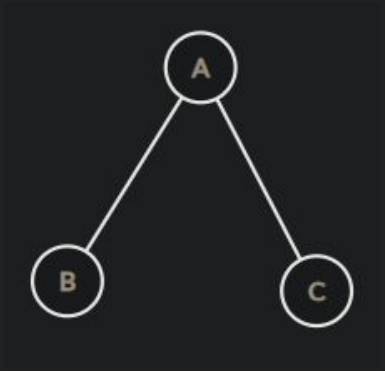
https://atcoder.jp/contests/dp/tasks/dp_p

Tree Coloring

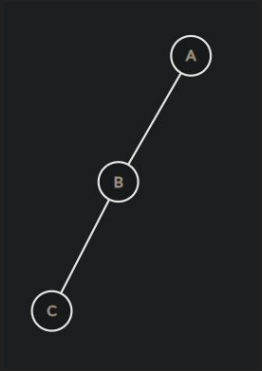
Given a rooted tree with n nodes.

We paint each node in white or black. Here, it is not allowed to paint two adjacent node both in black.

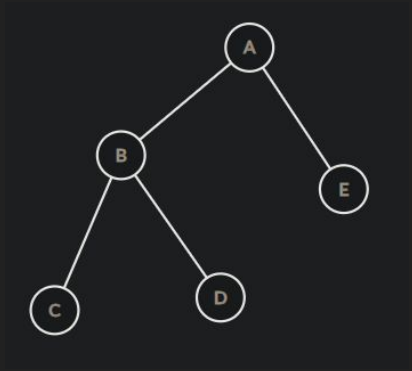
Find the number of ways in which the nodes can be painted



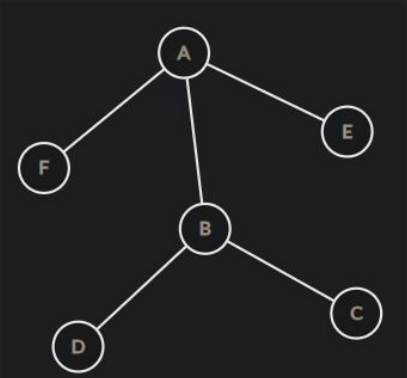
Answer: 5



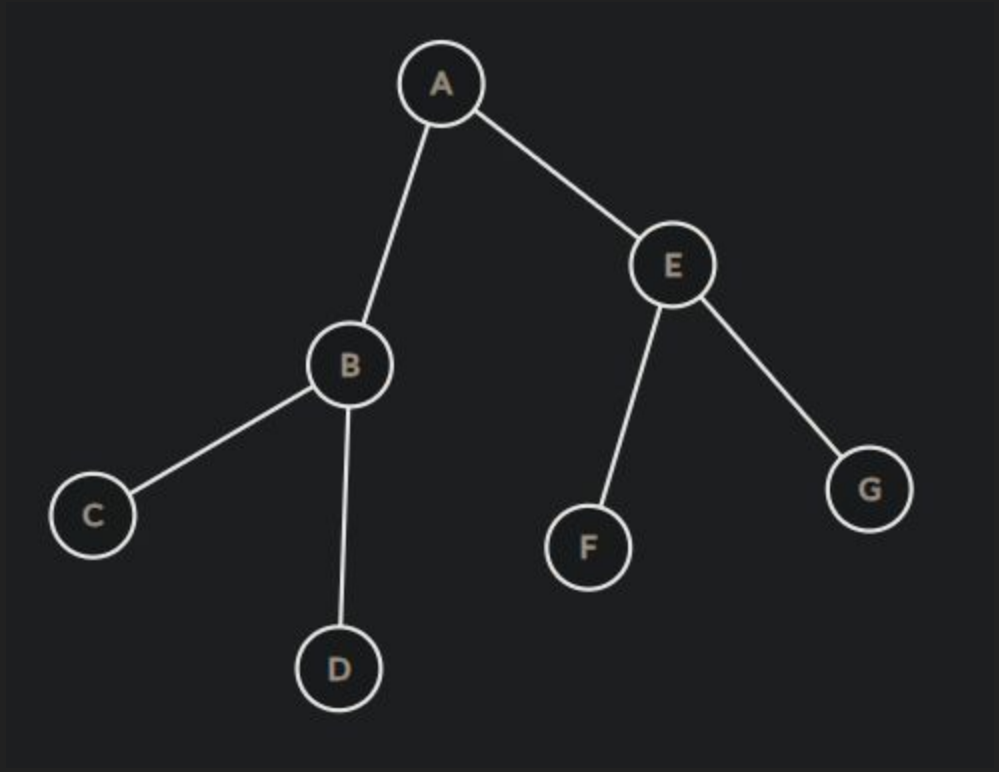
Answer: 5



Answer: 14



Answer: 24



Answer: 41

Time Complexity: $\mathcal{O}(N)$

First, lets root the tree arbitrary. Let $\mathbf{dp}[i][j]$ represent the number of ways to color subtree i, coloring node i color j.

Here, $j = 0$ is black and $j = 1$ is white. If we color node i black, then it's children must be all white. So, we have

$$\mathbf{dp}[i][0] = \prod \mathbf{dp}[c][1]$$

such that c is i's child, and if we color node i white, then it's children can be either white or black:

$$\mathbf{dp}[i][1] = \prod (\mathbf{dp}[c][0] + \mathbf{dp}[c][1])$$

Peer Grading

Exercise 6.1

Peer grading must be done with former groups

Sheet 7 must be submitted with new groups