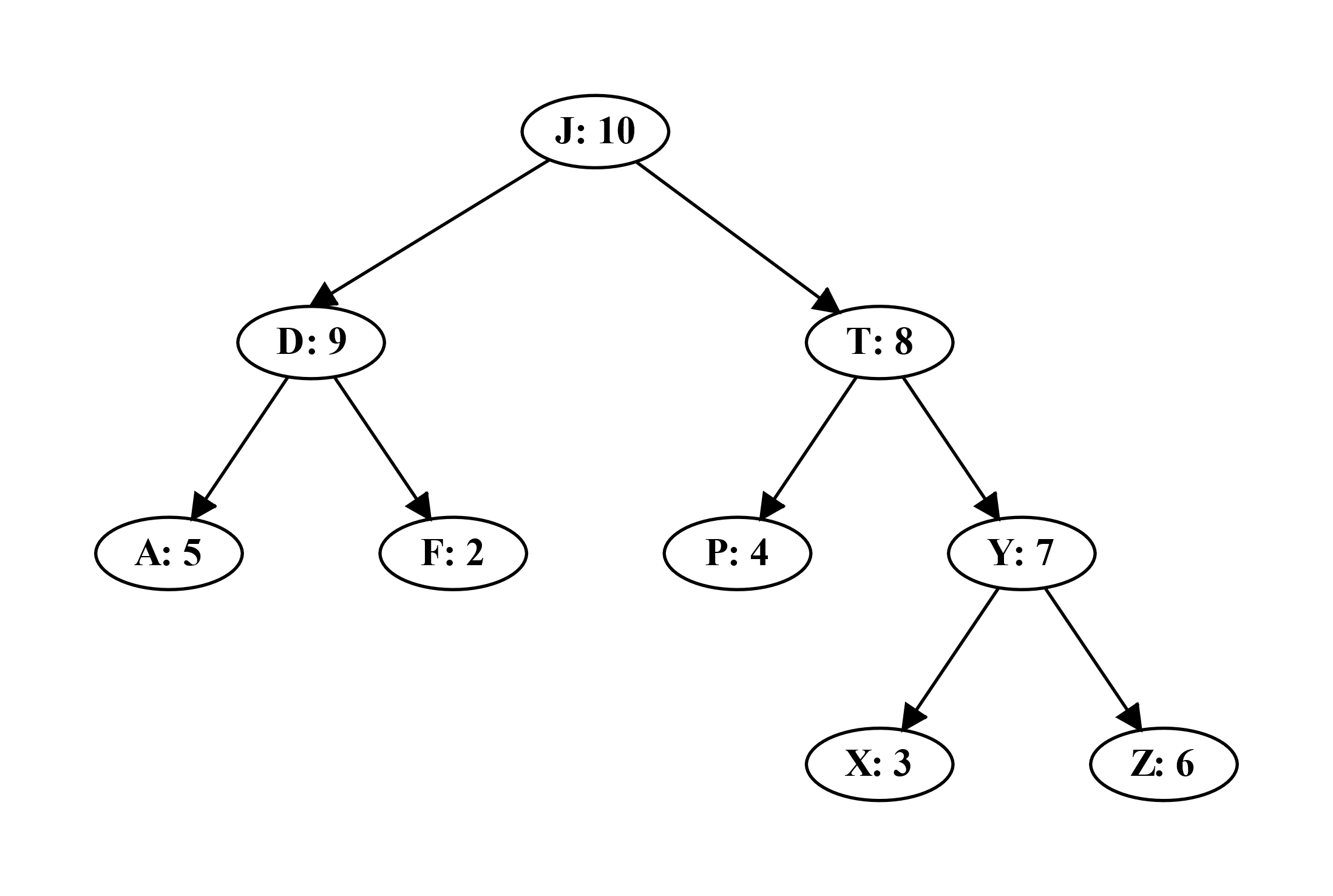
**Advanced Algorithms**

**Exercise for Lecture 8**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name** |  | **Student ID** |  |
| **Problem 1** |  | | |
| **Problem 2** |  | | |
| **Problem 3** |  | | |
| **Problem 4** |  | | |
| **Total Score** |  | | |
| **Notes** | Deadline: **2023-10-03 24:00**  Submission Format: ‘**Lecture8\_Name\_Student ID.docx**’, and please send to: **[algorithms\_23fall@163.com](mailto:algorithms_23fall@163.com)**.  This assignment is meant to be an evaluation of your **individual** understanding coming into the course and should be completed **without collaboration** or outside help. | | |

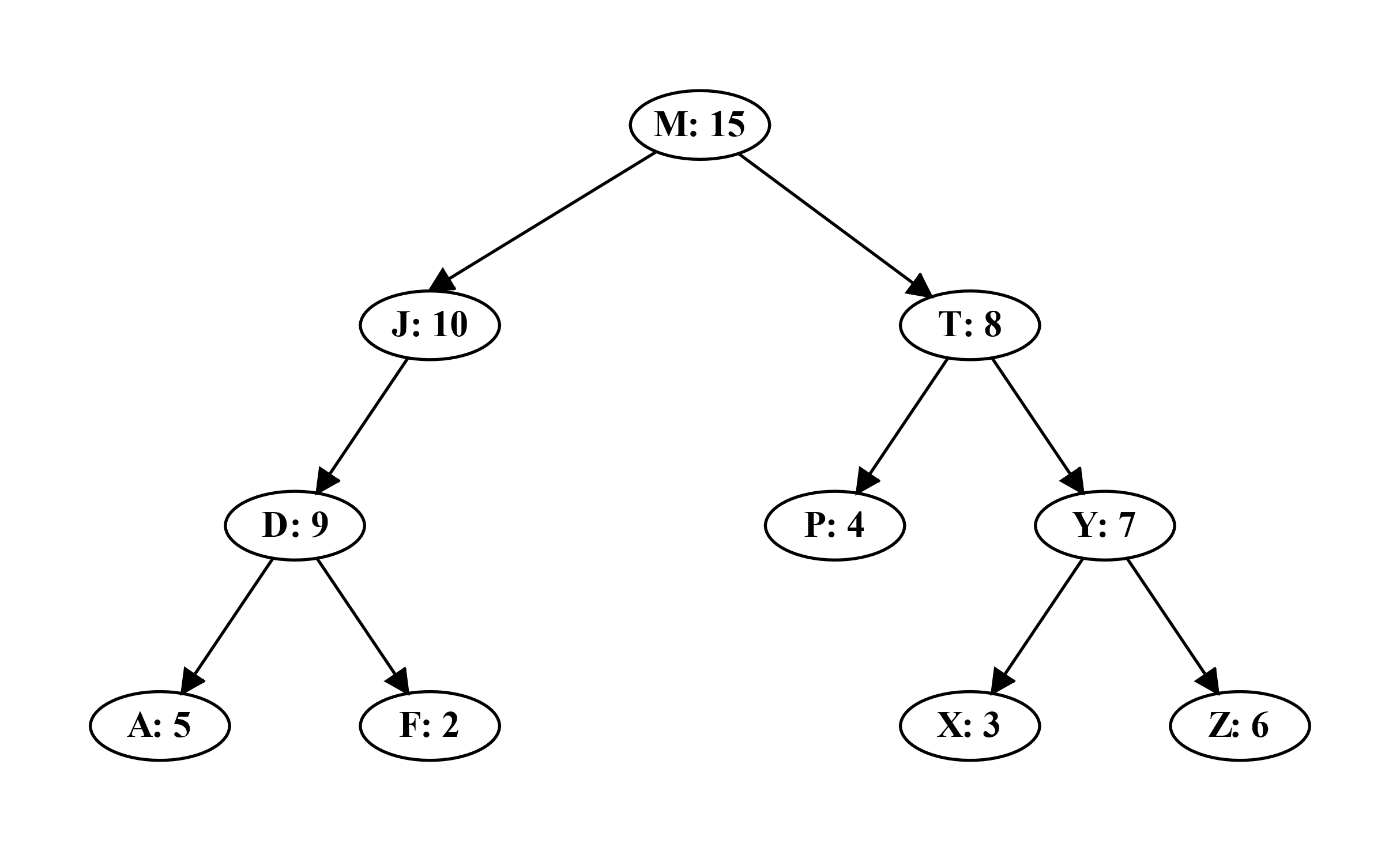
**Problem 1. [20 points]** Consider the following (**max-heap**) treap, where the keys are the letters and the priorities are the integers:



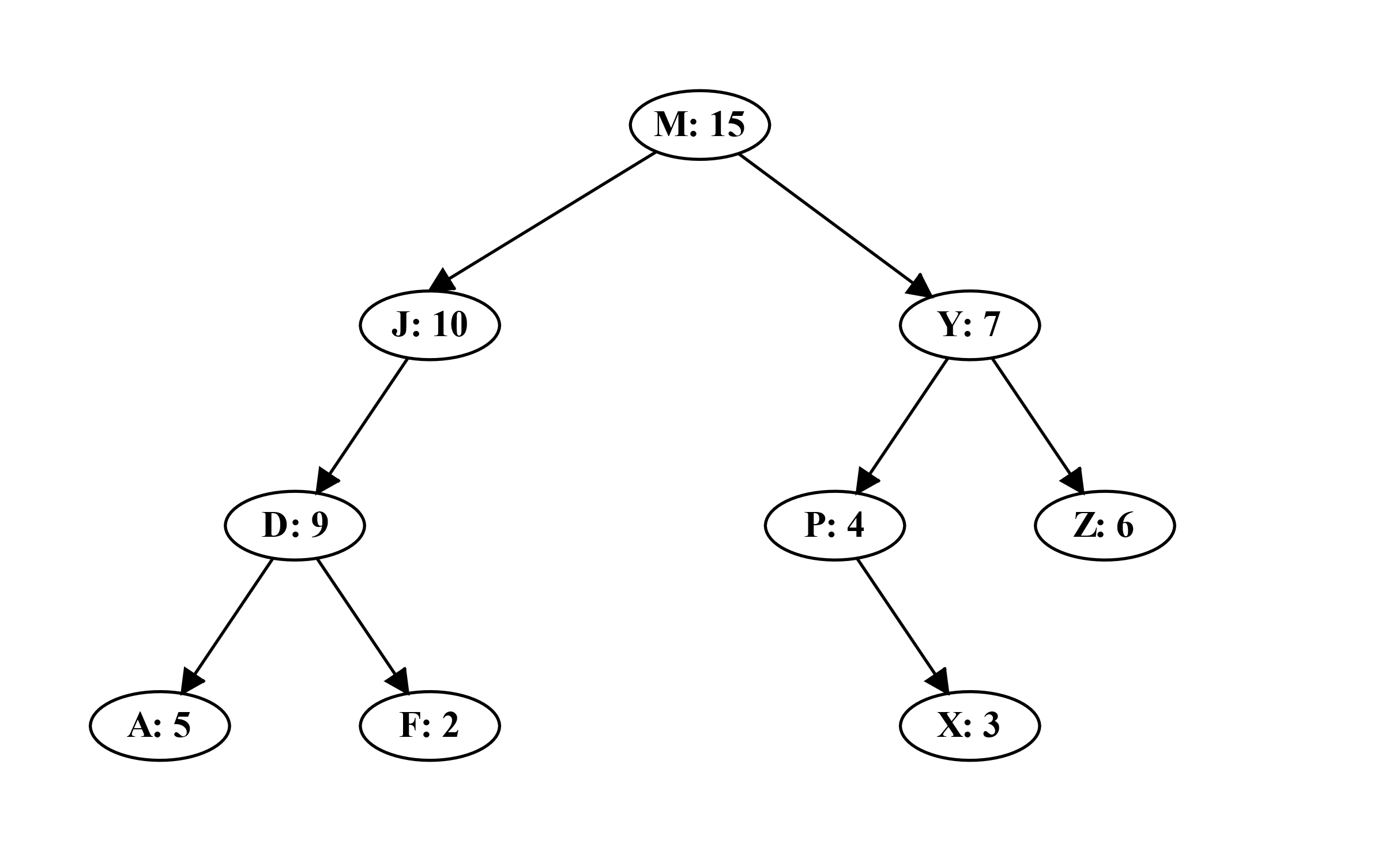
Show the result of inserting the key M, including any necessary rotations. Assume the priority generated for the key M is 15. **Then**, show the result of removing the key T, including any necessary rotations.

**Solution:**

Step 1: [10 points]



Step 2: [10 points]



**Problem 2. [25 points]** Lecture 7 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. Show how to maintain, for every node of a 2-3-4 tree, the height of the subtree rooted at as an attribute . Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.

**Solution:**

Let leaf nodes have a of and internal nodes have . For searching, it does not change the value of any node. A node affected by insertion or deletion will simply recalculate their value by looking at the of their children. In both insertion and deletion, at most nodes positions will be affected and each of their values can be updated in time. Therefore, the asymptotic running times in searching, insertion, and deletion is the same. A slight caveat in using this method is that we must ensure that heights are calculated from the bottom-up, otherwise there could be a case where a parent computes its height from an out-of-date child. Fortunately, both insertion and deletion recursively work from the bottom of the tree upward, so this is not an issue. [25 points]

**Problem 3. [25 points]** Given an interval tree and an interval , write an efficient algorithm MIN-INTERVAL-SEARCH() that returns an interval overlapping that has the minimum low endpoint, or if no such interval exists.

**Solution:**

As it travels down the tree, INTERVAL-SEARCH first checks whether current node overlaps the query interval and, if it does not, goes down to either the left or right child. If node overlaps , and some node in the right subtree overlaps , but no node in the left subtree overlaps , then because the keys are low endpoints, this order of checking (first , then one child) will return the overlapping interval with the minimum low endpoint. On the other hand, if there is an interval that overlaps in the left subtree of , then checking before the left subtree might cause the procedure to return an interval whose low endpoint is not the minimum of those that overlap . Therefore, if there is a possibility that the left subtree might contain an interval that overlaps , we need to check the left subtree first. If there is no overlap in the left subtree but node overlaps , then we return . We check the right subtree under the same conditions as in INTERVAL-SEARCH: the left subtree cannot contain an interval that overlaps , and node does not overlap , either. [15 points]

Because we might search the left subtree first, it is easier to write the pseudocode to use a recursive procedure MIN-INTERVAL-SEARCH-FROM(), which returns the node overlapping with the minimum low endpoint in the subtree rooted at , or if there is no such node. [8 points]

MIN-INTERVAL-SEARCH()

1. **return** MIN-INTERVAL-SEARCH-FROM()

MIN-INTERVAL-SEARCH-FROM()

1. **if** **and**
2. MIN-INTERVAL-SEARCH-FROM()
3. **if**
4. **return**
5. **elseif**  overlaps
6. **return**
7. **else** **return**
8. **elseif**  overlaps
9. **return**
10. **else** **return** MIN-INTERVAL-SEARCH-FROM()

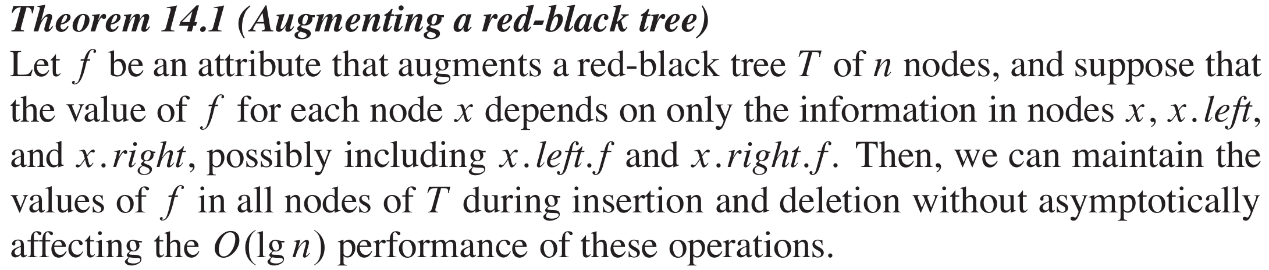
The call MIN-INTERVAL-SEARCH() takes time, since each recursive call of MIN-INTERVAL-SEARCH-FROM goes one node lower in the tree, and the height of the tree is . [2 points]

**Problem 4. [30 points]** Suppose that as a new accountant, you must design a data structure that stores the history of all the transactions on the bank account. In keeping with corporate policy, this data structure should support insertion of past and future transactions, as well as deletion of existing transactions. For this problem, you may assume that no two transactions occur on the same date. The data structure should be able to support the following operations:

* INITIALIZE: Initialize the account. The initial balance in the account is $0. This operation should take time.
* INSTRANS(): Insert a given transaction at a given date. The sum can be either positive or negative, and should be added to the balance in the account starting from the following day. Notice that the date can be arbitrary (not necessarily today’s date). This operation should take time, where is the number of transactions in the database.
* DELTRANS(): Delete the transaction that occurs at the given date, if there is any. When a transaction is deleted, the corresponding sum should be subtracted from the balance in the account starting from the following day. This operation should take steps.
* BALANCE(): Returns the balance in the account at the beginning of the given date. This operation should take time.

Explain how to use a red-black tree to implement this data structure, give clear and concise descriptions of the operations, and argue why they meet the desired running times. No need to write code.

**Solution:**



The data structure will be a red-black tree, ordered by the dates of the transactions, where each node is augmented by a field , which is the total change (increase) in the account balance over all nodes in ’s subtree. This information is merely the sum of all values of nodes in ’s subtree (including ). It is clear that this information can maintained without affecting the running times of the red-black tree operations (by Theorem 14.1): , and . [10 points]

From this description, INITIALIZE is trivial: simply construct an initially- red-black tree. Likewise, INSTRANS and DELTRANS are straightforward by Theorem 14.1: simply use the red-black tree INSERT and DELETE procedures, along with extra code to maintain according to the equations above. [5 points]

BALANCE() is the only non-trivial operation. It is not enough to merely sum the values of all elements preceding , because there may be up to of them. Instead, we consider the path () from to the root, where is the node for date and is the root. Initially set an accumulator variable to . Then follow the path: whenever we add and to the accumulator. The balance is the value of the accumulator at the end of the path. [10 points]

The running time of this is , because the tree is balanced (so the path is short). To see correctness, we need to show that the value of every transaction that occurs before is counted exactly once. Consider any date , and look at its path () to the root (where is the node for , and is the root). Then this path intersects the path from to the root at some node, and is identical to it from that point on. The from the root downward, we have , , , , and (where the paths diverge or one path ends). There are now two cases: if , then the node for (which is ) is on the path from to the root. Because , it must be that is the right child of , so in fact the algorithm adds to the accumulator. If , then the node for must be in the left subtree of (otherwise we would have because the path to diverges or ends at ). Then is at or is in the right subtree of . If it is at , then and the accumulator initially is set to the value of , which includes (exactly once) the value of the transaction on date . If is in the right subtree of , then , so is added to the accumulator by the algorithm, and includes (exactly once) the value of ’s transaction. This completes the proof of correctness. [5 points]