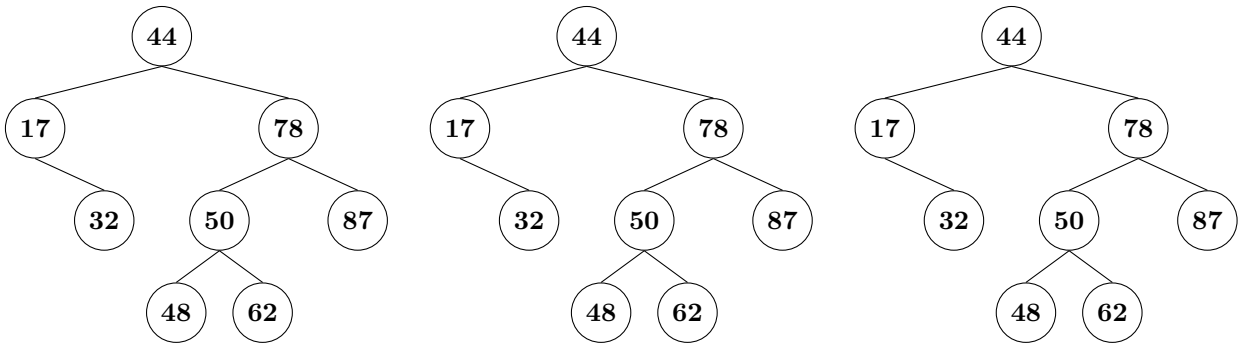
A **rooted tree** T is a set of **node**s storing elements in a **parent-child** relationship with the following properties:

- If T is nonempty, it has a designated root node, which has no parent.
- Every node v other than the root has a unique **parent**
- Every node with parent w is a **child** of w.

A binary search tree is a type of rooted tree. If the tree is non-empty, then the first node is a **root** of the tree (drawn at the top for some reason). You might think of a binary search tree like a linked list with two "next" pointers: one for the list that has only smaller values and one for only larger values. We call those "left" (smaller) and "right" (larger) pointers in this context.

Question 1. Starting with an initially empty binary search tree, insert the keys 50, 25, 75, 60, 55, 90, 65, 37

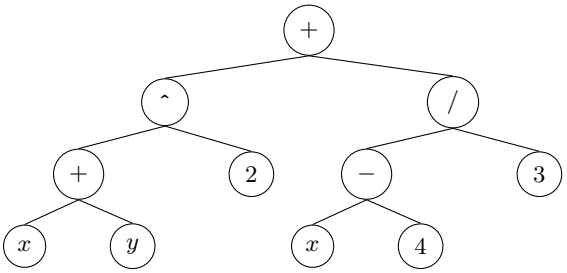
Question 2. From the following tree, what is the result if we delete node 62? What if we instead deleted 17? 78?



Question 3. Write a function that will output each key of a binary search tree from smallest to largest. You may assume you start with a pointer to the root.

Question 4. Write a function that will output each key of a binary search tree. Do so in such an order that if I insert each element of your output into an initially empty binary search tree, the result will be a tree that matches the tree we started with.

Question 5. Write code to evaluate a mathematical expression, represented as a *syntax tree*, such as the one below. A syntax tree will be a binary tree, but not a binary *search* tree. You may assume a function `eval(operator, left, right)` that evaluates any mathematical operator and `lookup(variable)` that looks up the value of a variable. You may also assume that if you call `lookup` with a constant as a parameter, it returns that constant.



Reinforcement

After the first lecture, you should be able to define and use the following terms with respect to a binary search tree: parent, child, sibling nodes, leaf node, internal node, in-order traversal, pre-order traversal, post-order traversal. You should be able to describe the procedure to insert a new key into a binary search tree, how to delete from a binary search tree, and how to search a binary search tree for a particular key. Given a list of keys, you should be able to insert them into an initially empty binary search tree.

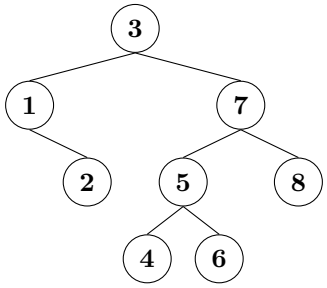
Randomized Binary Search Trees

In this section, we will evaluate what happens when a set of keys are to be added to an initially empty binary search tree and are added in a random order; that is, each permutation of the n keys to be added is equally likely. For convenience, assume the keys to be inserted are $1 \ldots n$. Do you see why that is a reasonable assumption?

Ultimately, we want to know: what is the average depth of a node? That is, how far is it from the root?

Define D_i to be the depth of node i, and X_{ij} to be an indicator random variable representing whether or not i is an *ancestor* of j. For convenience, let $X_{ii} = 1$

Example:



Question 6. In the example above, what are the values of X_{13}, X_{31}, and X_{47}?

Question 7. We insert keys $1 \ldots n$ in some random order. What event has to happen for i to be an ancestor of j? If all permutations of the input keys are equally likely, what is the probability that i is an ancestor of j? That is, $P(X_{ij} = 1)$?

We can now find the expected depth of a node.

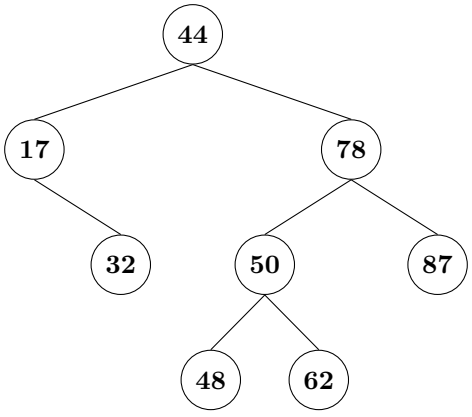
$$\begin{aligned} E[D_j] &= \sum_{i=1}^{n} E[X_{ij}] - 1 \\ &= \sum_{i=1}^{j} E[X_{ij}] + \sum_{i=j+1}^{n} E[X_{ij}] - 1 \\ &= \end{aligned}$$

AVL Trees

We say that a tree is an **AVL Tree** if the following two conditions both hold:

- The **binary search tree** property holds for all nodes.
- For every node v of T, the heights of the children of v differ by at most 1. This is referred to as the **height-balance property**.

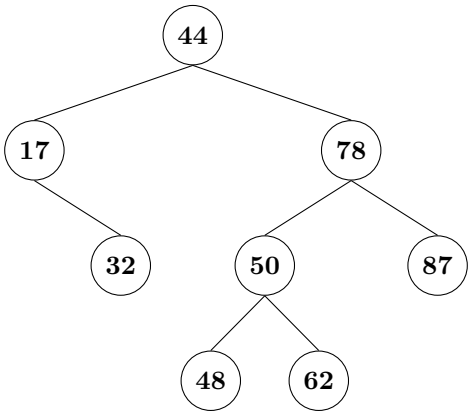
Note that this means that any subtree of an AVL tree is itself an AVL tree.



Question 8. What is the *height* of each node? Is each node height balanced?

Insert Operations

Question 9. Consider the example AVL tree from earlier, and consider what would happen if we were to insert the key 14 into it. Is it still an AVL tree? If it is not, what if we **instead** insert key 54?



Question 10. Suppose we insert 99 to the result of the previous operation. What should happen now?

Question 11. Suppose we insert 30 to the result of the previous operation. What should happen now?

Maintaining the Height-Balance Property

After inserting a node to a binary search tree, we apply the following procedure:

Start at the newly-inserted node and walk up to the root, checking if each node is balanced (the height-balance rule applies to this node). If a node is unbalanced, rotate the subtree rooted at that node. If all are balanced, we're done. Otherwise, rotate the following three nodes:

1. Let z be (a pointer to) the first unbalanced node on the way up.
2. Let y be the child of z with greater height (hint: this is always an ancestor of the node you inserted. Why?).
3. Let x be the child of y with greater height. In the event of a tie, choose x to be an ancestor of the node you inserted.

When x, y, z form a zig-zag pattern, we do a **double rotation**. Otherwise we do a **single rotation**. The rotations are pictured on the last page of this packet. Each of these is considered a *single update operation.*

Question 12. Draw the smallest (fewest node) AVL Tree you can that has height 1.

Question 13. Draw the smallest (fewest node) AVL Tree you can that has height 2.

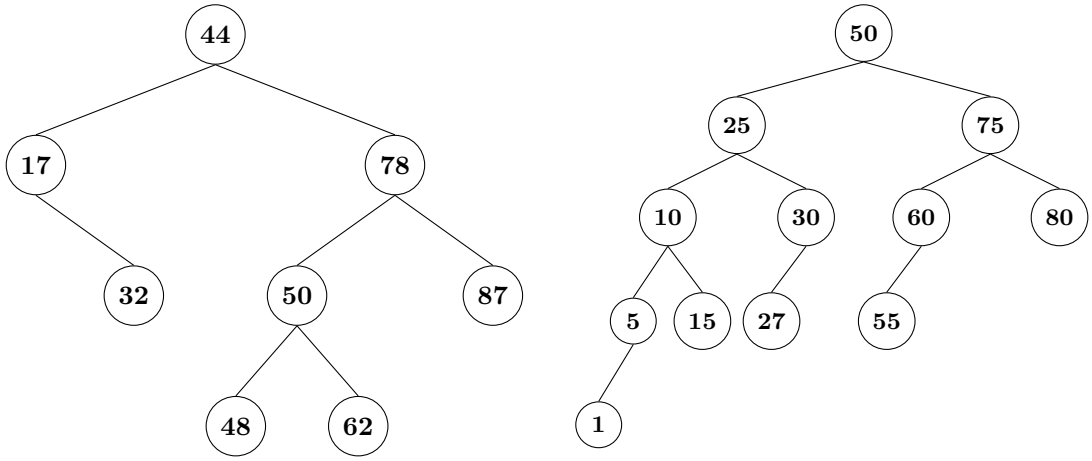
Question 14. What is the minimum number of internal nodes an AVL tree can have if it has height h?

Let n_h be the minimum number of tree nodes for a height-balanced binary search tree with height h. What are n_0 and n_1?

Question 15. What does this tell you about the running time needed for a `find` operation on an AVL Tree? For an `insert` operation?

Deleting from an AVL Tree

Question 16. What if we delete 32 from the left AVL Tree below?



Question 17. What if we delete 80 from the AVL tree on the right of the two above?

Question 18. What is the running time for a `delete` operation on an AVL Tree?

Question 19. Starting with an empty AVL tree, insert the following keys into the tree, in sequence: $1, 2, 3, 12, 9, 4, 7, 5, 13, 15, 6, 16, 14, 17$. Note that there is not enough room on this paper to complete this here.

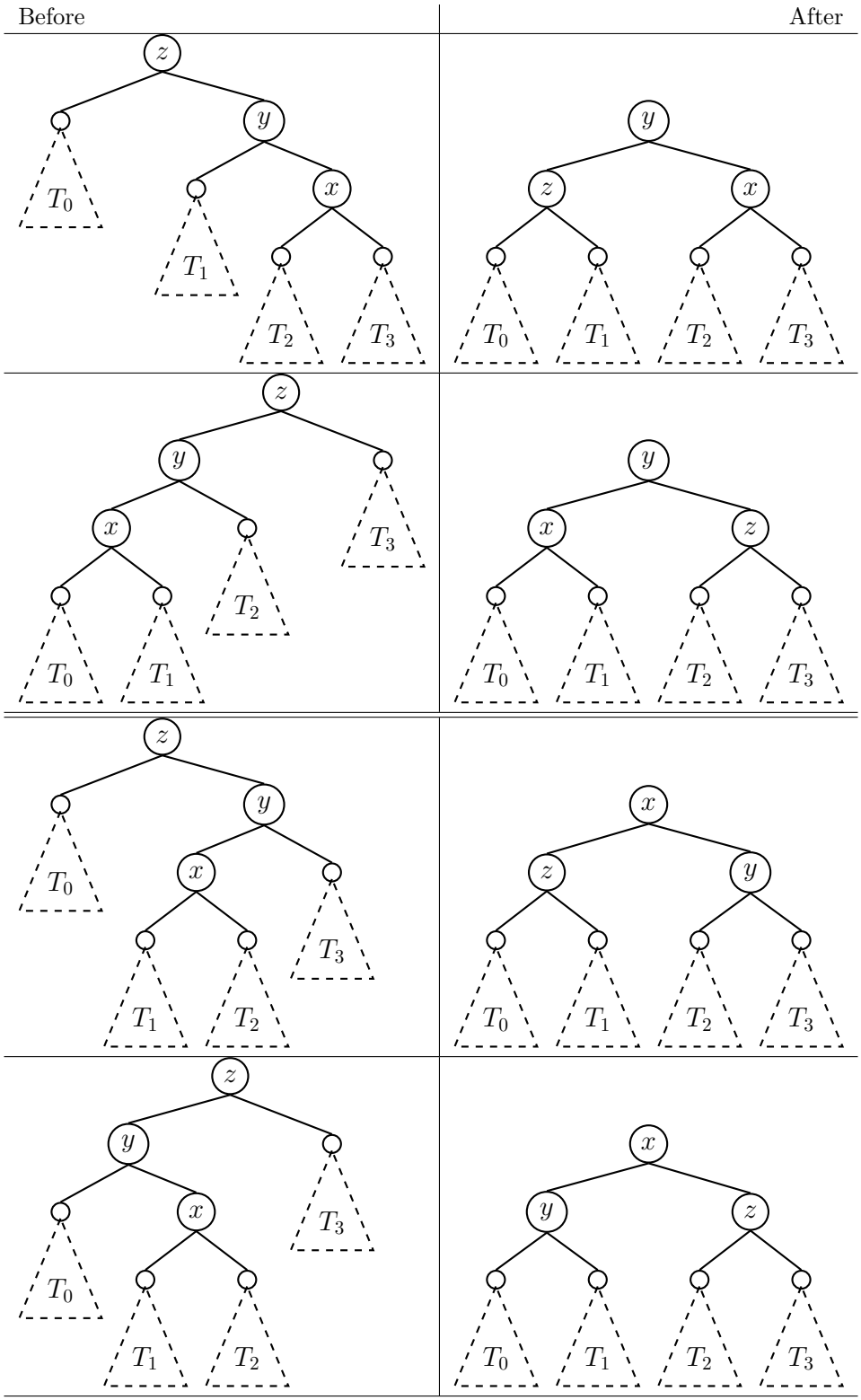


Table 1: Figure 10.10 from the textbook of Goodrich/Tamassia