Problem 213: Data Forest

Difficulty: Medium

Author: Vedant Patel, Stratford, Connecticut, United States

Originally Published: Code Quest 2023

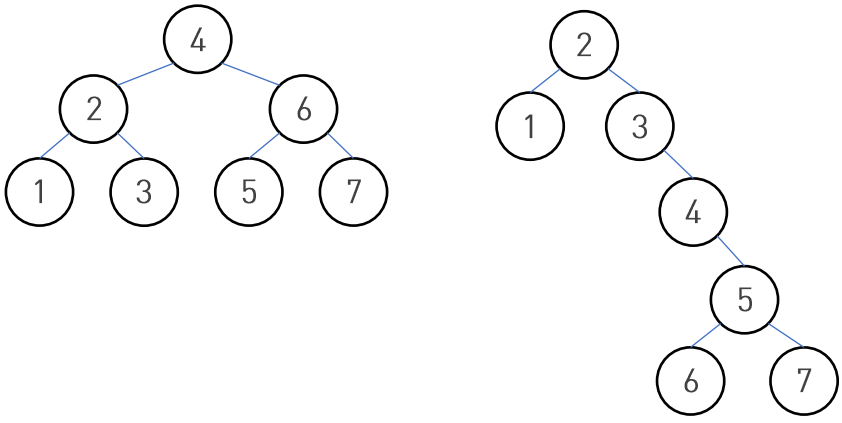
Problem Background

Being able to find data quickly is an important task for any software application. Huge amounts of research have gone into developing different data structures, and the specific data structure used by an application depends on a wide range of factors. For simple data that can be easily sorted, a binary search tree allows that data to be retrieved quickly and efficiently.

A binary search tree is simple to set up. Each node within the tree is associated with a value, and can have up to two child nodes: a left child and a right child. The left child (and its descendants) must have values less than the parent node's value; the right child (and its descendants) have values greater than that of the parent. Searching for a value therefore requires comparing your desired value against the root node of the tree, and moving down the appropriate branch until your target is found.

Problem Description

A binary tree must be built correctly for it to be as efficient as possible. For example, consider the two binary search trees below, which both include the integers from 1 to 7 inclusive:



The tree on the right is poorly built; if searching for 6 or 7, you'll have to do four comparisons before you find them. The tree on the left is much shorter; it will only take two comparisons at most to locate any within the tree. It's what we call "balanced." A balanced binary tree optimizes retrieval time by ensuring that no data is added to a new layer of the tree until the layer before it is filled. Most trees follow a set of rules to ensure that they remain balanced, even as data is added or removed.

You'll need to read in a list of numbers and build a balanced binary search tree from those numbers. For this problem, you'll be guaranteed to get enough data to completely fill the tree; that is, the

bottom-most layer of the tree will be completely filled, as shown in the balanced tree diagram above. Once you've determined how the tree should be laid out, you'll need to print out a visual representation of the tree similar to the above diagram. To ensure consistent indentation, each number within the tree should be printed using three characters, adding leading spaces as necessary. Spaces should be printed at the beginning of each line and between each number, in multiples of three, to ensure that each parent node is positioned directly between both of its children.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include a single line containing a list of unique integers with values between 0 and 999 inclusive, separated by spaces. The number of integers in each line will equal $2^N - 1$ for an undisclosed integer value of N between 1 and 9 inclusive.

```
2
1 2 3 4 5 6 7
106 115 103 111 108 101 104 113 105 110 109 112 102 107 114
```

Sample Output

For each test case, your program must print the structure of the tree, laid out across N lines, indented and spaced as described above. **Do not use tab characters to indent lines or separate numbers, and do not print trailing spaces on any line.**

```
            4
        2           6
  1     3     5     7
                    108
          104                   112
    102         106         110         114
101   103   105   107   109   111   113   115
```