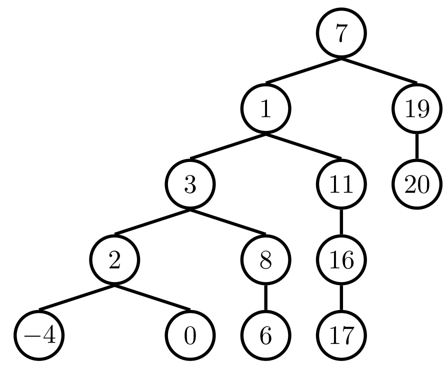
1 Trees



In computer science, **trees** are recursive data structures that are widely used in various settings. The diagram to the right is an example of a tree.

Notice that the tree branches downward. In computer science, the **root** of a tree starts at the top, and the **leaves** are at the bottom.

Some terminology regarding trees:

- **Parent node**: A node that has branches. Parent nodes can have multiple branches.
- **Child node**: A node that has a parent. A child node can only belong to one parent.
- **Root**: The top node of the tree. In our example, the node that contains 7 is the root.
- **Label**: The value at a node. In our example, all of the integers are values.
- **Leaf**: A node that has no branches. In our example, the nodes that contain −4, 0, 6, 17, and 20 are leaves.
- **Branch**: A subtree of the root. Note that trees have branches, which are trees themselves: this is why trees are *recursive* data structures.
- **Depth**: How far away a node is from the root. In other words, the number of edges between the root of the tree to the node. In the diagram, the node containing 19 has depth 1; the node containing 3 has depth 2. Since there are no edges between the root of the tree and itself, the depth of the root is 0.
- **Height**: The depth of the lowest leaf. In the diagram, the nodes containing −4, 0, 6, and 17 are all the “lowest leaves,” and they have depth 4. Thus, the entire tree has height 4.

In computer science, there are many different types of trees. Some vary in the number of branches each node has; others vary in the structure of the tree.

Implementation

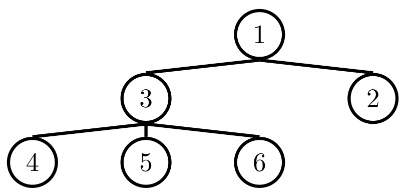
A tree has both a value for the root node and a sequence of branches, which are also trees. In our implementation, we represent the branches as a list of trees. Since a tree is an abstract data type, our choice to use lists is just an implementation detail.

- The arguments to the constructor `tree` are the value for the root node and an optional list of branches. *If no branches parameter is provided, the default value* `[]` *is used.*
- The selectors for these are `label` and `branches`.

Note that `branches` returns a list of trees and not a tree directly. It's important to distinguish between working with a tree and working with a **list of** trees.

We have also provided a convenience function, `is_leaf`.

Let's try to create the tree below.




```
# Example tree construction
t = tree(1,
      [tree(3,
          [tree(4),
           tree(5),
           tree(6)]),
      tree(2)])
```

Questions

1.1 Write a function that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
def height(t):
    """Return the height of a tree.

    >>> t = tree(3, [tree(5, [tree(1)]), tree(2)])
    >>> height(t)
    2
    """
```

1.2 Write a function that takes in a tree and returns the maximum sum of the values along any path in the tree. Recall that a path is from the tree's root to any leaf.

```
def max_path_sum(t):
    """Return the maximum path sum of the tree.

    >>> t = tree(1, [tree(5, [tree(1), tree(3)]), tree(10)])
    >>> max_path_sum(t)
    11
    """
```

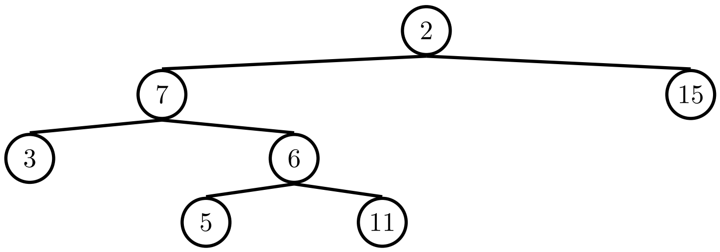
1.3 **Tutorial:** Write a function that takes in a tree and squares every value. It should return a new tree. You can assume that every item is a number.

```
def square_tree(t):
    """Return a tree with the square of every element in t
    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(square_tree(numbers))
    1
      4
        9
        16
      25
        36
          49
        64
    """
```

1.4 **Tutorial:** Write a function that takes in a tree and a value `x` and returns a list containing the nodes along the path required to get from the root of the tree to a node containing `x`.

If `x` is not present in the tree, return `None`. Assume that the entries of the tree are unique.

For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`




```
def find_path(tree, x):
    """
    >>> t = tree(2, [tree(7, [tree(3), tree(6, [tree(5), tree(11)])] ), tree(15)])
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """

    if _____________________________:

        return _____________________________

    _____________________________:

        path = _____________________________

        if _____________________________:

            return _____________________________
```

2 Binary Numbers

In normal life, we think of numbers as being defined in base 10: i.e. We define numbers with digits 0 through 9 and represent numbers as such:

- 11 = 1 * 10 + 1 * 1
- 123 = 1 * 100 + 2 * 10 + 3 * 1
- 9805 = 9 * 1000 + 8 * 100 + 9 * 10 + 5 * 1

But in computer science, we oftentimes look at these numbers in base 2, or **binary** instead. Then, we see numbers represented in 0s and 1s and breakdown their digits in terms of powers of two:

- 11 = 1 * 8 + 0 * 4 + 1 * 2 + 1 * 1 = 1011
- 3 = 1 * 2 + 1 * 1 = 11
- 6 = 1 * 4 + 1 * 2 + 0 * 1 = 110

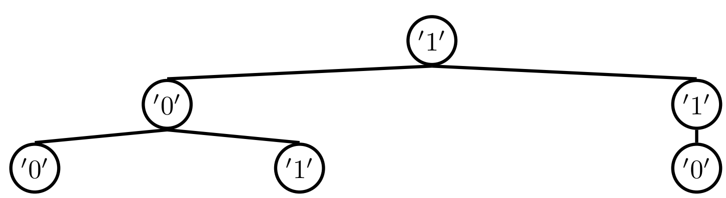
2.1 Fill in the table to convert the following numbers between decimal and binary.

Decimal	Binary (unsigned)
5	
10	
14	
37	
	10
	101010
	1100101

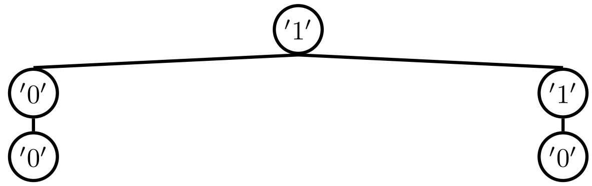
2.2 Write a function that takes in a tree consisting of '0's and '1's `t` and a list of "binary numbers" `nums` and returns a new tree that contains only the numbers in `nums` that exist in `t`. If there are no numbers in `nums` that exist in `t`, return `None`.

Definition: Each binary number is represented as a string. A binary number `n` exists in `t` if there is some path from the root to leaf of `t` whose values are equal to `n`.

For example, if `t` is as follows:



Then `prune_binary(t, ['01', '110', '100'])` should return the following tree.




```
def prune_binary(t, nums):
  if _________________________:
    if _____________:
      return t
    return None
  else:
    next_valid_nums = __________________________
    new_branches = []
    for ____________________:
      pruned_branch = prune_binary(_____, next_valid_nums)
      if pruned_branch is not None:
        new_branches = new_branches + [__________]
    if not new_branches:
      return None
    return ______________________________
```