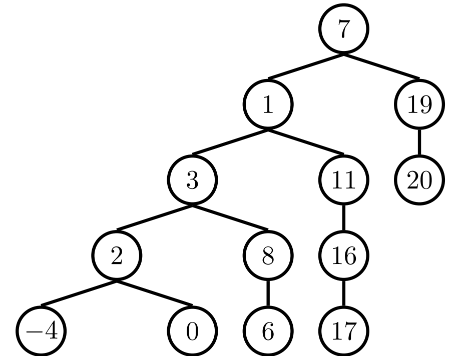
1 Trees



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

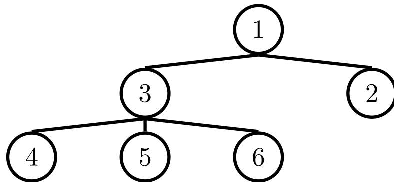
A tree has both a value for the root node and a sequence of branches, which are also trees. In our implementation, we represent the branches as a list of trees. Since a tree is an abstract data type, our choice to use lists is just an implementation detail.

- 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
    """


    if is_leaf(t):
        return 0
    return 1 + max([height(branch) for branch in branches(t)])
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```
    # alternate solution
    return 1 + max([0] + [height(branch) for branch in branches(t)])
```

Video walkthrough

1.2 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
    """


    sq_branches = [square_tree(branch) for branch in branches(t)]
    return tree(label(t)**2, sq_branches)
```

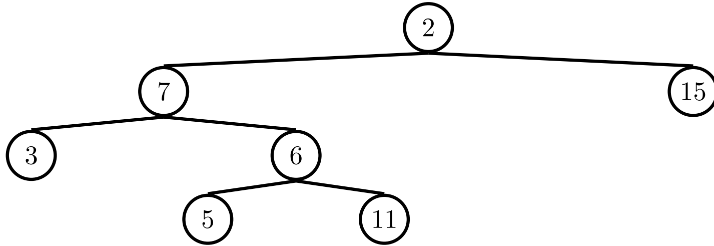
Video walkthrough

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

1.3 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 _____________________________
```

```
def find_path(tree, x):
    if label(tree) == x:
        return [label(tree)]
    for b in branches(tree):
        path = find_path(b, x)
        if path:
            return [label(tree)] + path
```

Video walkthrough

1.4 Consider a tree ADT `t = tree(1, [tree(2), tree(3)])`. For each of the following expressions, answer these two questions:

- What does the expression evaluate to?
- Does the expression violate any abstraction barriers? If so, write an equivalent expression that does not violate abstraction barriers.

1. `label(t)`

   Evaluates to the label of the entire tree, 1. This is simply using a selector to get the label, which is not violating any abstraction barriers.

2. `t[0]`

   This expression evaluates to the label of the entire tree, 1. However, it makes use of the fact that trees are implemented using lists, and violates the abstraction barrier. An equivalent expression is `label(t)`.

3. `label(branches(t)[0])`

   This expression evaluates to the label of the first branch of `t`, `tree(2)`. It is not a violation to index into `branches(t)` because it is given in the description of the ADT that `branches(t)` returns a list of branches.

4. `label(branches(t))`

   This expression evaluates to the first branch of `t`, `tree(2)`. It is a violation of the abstraction barrier as it assumesthat label gets the first element of a list.

   Treating a non-tree list as a tree is an abstraction violation since if `label` or `branches` change it no longer works

5. `is_leaf(t[1:][1])`

   This expression accesses the branches of `t` by slicing `t`, `[tree(2), tree(3)]`. Although this works because this is technically what `branches(t)` returns, this is an abstraction violation because we cannot assume the implementation of `branches(t)`.

   It then accesses the second branch by indexing into the list of branches, which is *not* an abstraction violation because we are allowed to assume that branches is a list. This expression evaluates to `True` because the second branch of `t` is a leaf. An equivalent expression is `is_leaf(branches(t)[1])`.

6. `[label(b) for b in branches(t)]`

   This expression uses the `branches` selector to access the branches of `t` and then iterates through it to construct a new list containing the labels of the branches, `[2, 3]`. It does not violate any abstraction barriers.

7. **Challenge:** `branches(tree(2, tree(t, [])))[0]`

   This expression evaluates to `t`. *Note: not* `tree(t, [])`. This is because the constructor `tree(2, tree(t, []))` passes `tree(t, [])` as the branches object, not as a single branch. Thus, the branches are `[t] + []`, or just `[t]`. This is

a violation of the abstraction barrier since you are passing a tree as a list of branches. An equivalent expression is `branches(tree(2, [t]))[0]`.

2 Mutation

Let's imagine you order a mushroom and cheese pizza from La Val's, and that they represent your order as a list:

```
>>> pizza = ['cheese', mushrooms']
```

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, La Val's would have to build an entirely new list to add onions:

```
>>> pizza = ['cheese', mushrooms']
>>> new_pizza = pizza + ['onions'] # creates a new python list
>>> new_pizza
['cheese', mushrooms', 'onions']
>>> pizza # the original list is unmodified
['cheese', 'mushrooms']
```

This is silly, considering that all La Val's had to do was add onions on top of `pizza` instead of making an entirely new pizza.

We can fix this issue with **list mutation**. In Python, some objects, such as lists and dictionaries, are **mutable**, meaning that their contents or state can be changed over the course of program execution. Other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created.

Therefore, instead of building a new pizza, we can just mutate `pizza` to add some onions!

```
>>> pizza.append('onions')
>>> pizza
['cheese', 'mushrooms', 'onions']
```

`append` is what's known as a method, or a function that belongs to an object, so we have to call it using dot notation. We'll talk more about methods later in the course, but for now, here's a list of useful list mutation methods:

1. `append(el)`: Adds `el` to the end of the list, and returns `None`
2. `extend(lst)`: Extends the list by concatenating it with `lst`, and returns `None`
3. `insert(i, el)`: Insert `el` at index `i` (does not replace element but adds a new one), and returns `None`
4. `remove(el)`: Removes the first occurrence of `el` in list, otherwise errors, and returns `None`
5. `pop(i)`: Removes and returns the element at index `i`

We can also use the familiar indexing operator with an assignment statement to change an existing element in a list. For example, we can change the element at index 1 and to `'tomatoes'` like so:

```
>>> pizza[1] = 'tomatoes'
>>> pizza
```

```
['cheese', 'tomatoes', 'onions']
```

Questions

2.1 What would Python display? In addition to giving the output, draw the box and pointer diagrams for each list to the right.

```
>>> lst1 = [1, 2, 3]
No output
>>> lst2 = lst1
No output
>>> lst1 is lst2
True
>>> lst2.extend([5, 6])
No output
>>> lst1[4]
6
>>> lst1.append([-1, 0, 1])
No output
>>> -1 in lst1
False
>>> lst2[5]
[-1, 0, 1]
>>> lst3 = lst2[:]
No output
>>> lst3.insert(3, lst2.pop(3))
No output
>>> len(lst1)
5
>>> lst1[4] is lst3[6]
True
>>> lst3[lst2[4][1]]
1
>>> lst1[:3] is lst2[:3]
```

```
False

>>> lst1[:3] == lst2[:3]

True

>>> x = (1, 2, [4, 5, 6])

No output

>>> x[2] = [3, 5, 6]

Error

>>> x

(1, 2, [4, 5, 6])

>>> x[2][0] = 3

No output

>>> x

(1, 2, [3, 5, 6])
```

2.2 Write a function that takes in a value `x`, a value `el`, and a list and adds as many `el`'s to the end of the list as there are `x`'s. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """


    count = 0
    for element in lst:
        if element == x:
            count += 1
    while count > 0:
        lst.append(el)
        count -= 1
```

2.3 Write a function that takes in a sequence `s` and a function `fn` and returns a dictionary.

The values of the dictionary are lists of elements from `s`. Each element `e` in a list should be constructed such that `fn(e)` is the same for all elements in that list. Finally, the key for each value should be `fn(e)`.

```
def group_by(s, fn):
    """
    >>> group_by([12, 23, 14, 45], lambda p: p // 10)
    {1: [12, 14], 2: [23], 4: [45]}
    >>> group_by(range(-3, 4), lambda x: x * x)
    {0: [0], 1: [-1, 1], 4: [-2, 2], 9: [-3, 3]}
    """


    grouped = {}
    for x in s:
        key = fn(x)
        if key in grouped:
            grouped[key].append(x)
        else:
            grouped[key] = [x]
    return grouped
```

1. **So Many Options...**

   (a) Implement the following function `partition_options` which outputs all the ways to partition a number `total` using numbers no larger than `biggest`.

```
def partition_options(total, biggest):
    """
    >>> partition_options(2, 2)
    [[2], [1, 1]]
    >>> partition_options(3, 3)
    [[3], [2, 1], [1, 1, 1]]
    >>> partition_options(4, 3)
    [[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
    """
    if total == 0:
        return [[]]
    elif total < 0 or biggest_num == 0:
        return []
    else:
        with_biggest = partition_options(total-biggest_num, biggest_num)
        without_biggest = partition_options(total, biggest_num-1)
        with_biggest = [[biggest_num] + elem for elem in with_biggest]
        return with_biggest + without_biggest
```

   (b) Return the minimum number of elements from the list that need to be summed in order to add up to `T`. The same element can be used multiple times in the sum. For example, for `T = 11` and $lst = [5, 4, 1]$ we should return 3 because at minimum we need to add 3 numbers together (5, 5, and 1). You can assume that there always exists a linear combination of the elements in `lst` that equals `T`.

```
def min_elements(T, lst):
    """
    >>> min_elements(10, [4, 2, 1]) # 4 + 4 + 2
    3
    >>> min_elements(12, [9, 4, 1]) # 4 + 4 + 4
    3
    >>> min_elements(0, [1, 2, 3])
    0
    """

    if T == 0:

        return 0

    return min([1 + min_elements(T - i, lst) for i in lst if T - i >= 0])
```

   (c) Reminder: don't forget to check your quiz solutions @ cs61a.org Quiz solutions can be found on the last page of the discussion solutions, which are posted at the end of each week. If you do not know where to find discussion solutions @ cs61a.org, see links.cs61a.org/quiz-sols-location