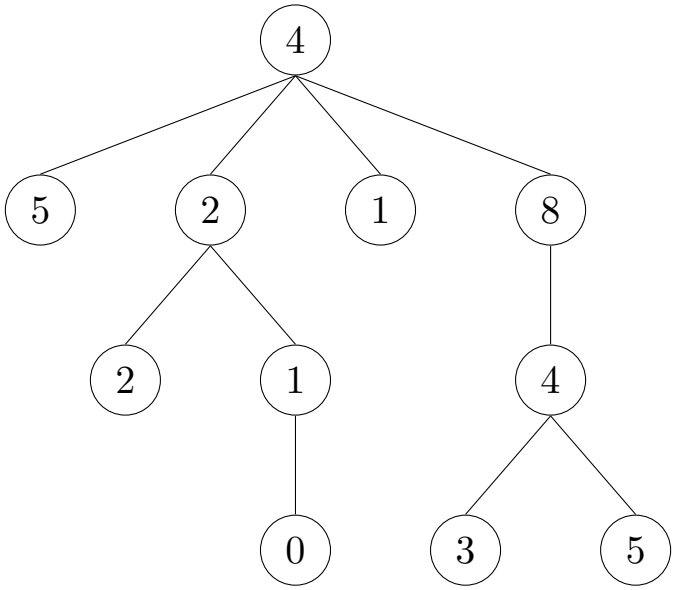
# FUNCTION-BASED TREES, MUTABILITY, & ITERATORS Solutions

COMPUTER SCIENCE MENTORS 61A

March 3 – March 7, 2025

## 1 Function-Based Trees

**Trees** are a kind of recursive data structure. Each tree has a **root label** (which is some value) and a sequence of **branches**. Trees are "recursive" because the branches of a tree are trees themselves! A typical tree might look something like this:




This tree's root label is 4, and it has 4 branches, each of which is a smaller tree. The 6 of the tree's **subtrees** are also **leaves**, which are trees that have no branches.

Trees may also be viewed **relationally**, as a network of nodes with parent-child relationships. Under this scheme, each circle in the tree diagram above is a node. Every non-root node has one parent above it and every non-leaf node has at least one child below it.

Trees are represented by an abstract data type with a `tree` constructor and `label` and `branches` selectors. The `tree` constructor takes in a label and a list of branches and returns a tree. Here's how one would construct the tree shown above with `tree`:

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

```
tree(4,
    [tree(5),
     tree(2,
        [tree(2),
         tree(1,
            [tree(0)])]),
     tree(1),
     tree(8,
        [tree(4,
            [tree(3), tree(5)])])])
```

The implementation of the ADT is provided here, but you shouldn't have to worry about this too much. (Remember the abstraction barrier!)

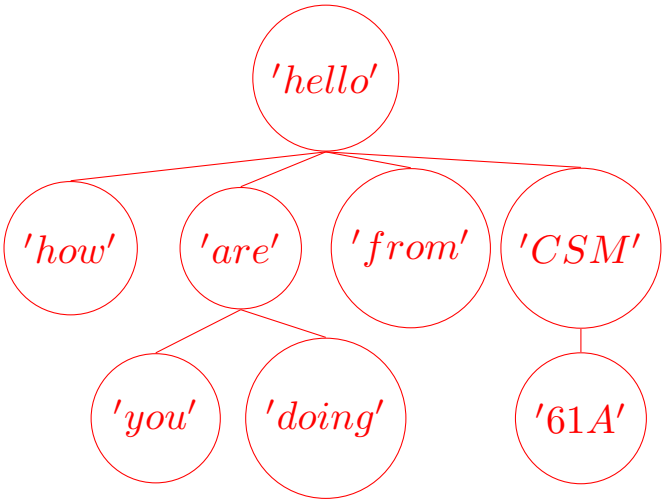
```
def tree(label, branches=[]):
        return [label] + list(branches)

def label(tree):
        return tree[0]

def branches(tree):
        return tree[1:] # returns a list of branches
```

Because trees are recursive data structures, recursion tends to a be a very natural way of solving problems that involve trees.

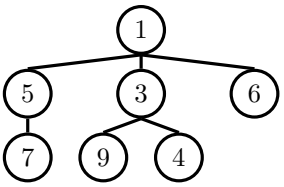
- The **recursive case** for tree problems often involves recursive calls on the branches of a tree.
- The **base case** is often reached when we hit a leaf because there are no more branches to recurse on.

1. Draw the tree that is created by the following statement:

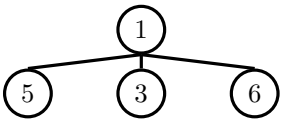
```
tree('hello',
    [tree('how', []),
     tree('are',
        [tree('you', []),
         tree('doing', [])]),
     tree('from', []),
     tree('CSM',
        [tree('61A', [])])])
```


'hello'
'how'
'are'
'from'
'CSM'
'you'
'doing'
'61A'

**BEFORE `prune(1)`:**




**AFTER `prune(1)`:**

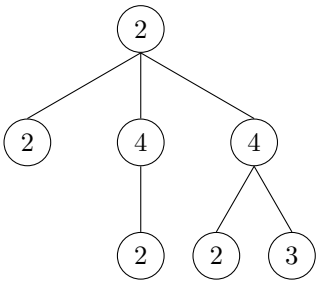



2. Implement `prune`, which takes in a tree `t` and a depth `k`, and should return a new tree that is a copy of only the first `k` levels of `t`. Suppose `t` is the tree shown to the right. Then `prune(t, 1)` returns nodes up to a depth of level 1.

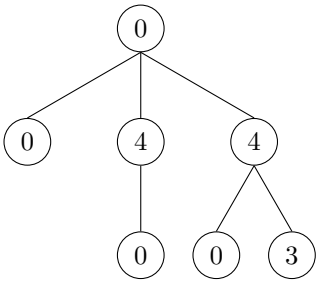
```
def prune(t, k):
    if k == 0:
        return tree(label(t))
    else:
        return tree(label(t), [prune(b, k - 1) for b in branches(t)])
```

3. Write a function, `replace_x` that takes in a tree, `t`, and returns a new tree with all labels `x` replaced with 0.

   For example, if we called `replace_x(t, 2)` on the following tree:




   We would expect it to return




```
def replace_x(t, x):
    """
    >>> t = tree(2, [tree(1), tree(2)])
    >>> replace_x(t, 2)
    tree(0, [tree(1), tree(0)])
    """
    new_branches = []
    for _______ in _____________________________:

        new_branches._____________________________

    if ___________________________________________:

        return ____________________________________

    return _____________________________________
```

```
def replace_x(t, x):
    new_branches = []
    for b in branches(t):
        new_branches.append(replace_x(b, x))
    if label(t) == x:
        return tree(0, new_branches)
    return tree(label(t), new_branches)
```

An alternate solution using list comprehensions is as follows:

```
def replace_x(t, x):
   new_branches = [replace_x(b, x) for b in branches(t)]
   if label(t) == x:
        return tree(0, new_branches)
   return tree(label(t), new_branches)
```

Here, we construct and return a new tree. First, we construct a new list of branches where each branch is the same as the previous branch but all occurrences of x have been replaced with 0 as per our recursive function. The if statement guarantees that if our root node's label is an occurence of x, we replace the subtree we're on starting at its root – keeping all else before it the same while replacing the specific subtree's root node label to be zero.
We do not need a base case here, as if we are at a leaf, the list comprehension we use to create the new branches will evaluate to an empty list. Then we will either return `tree(0, [])` or `tree(label(t), [])` as appropriate.

4. [Exam Level] Write a function that returns `True` if and only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

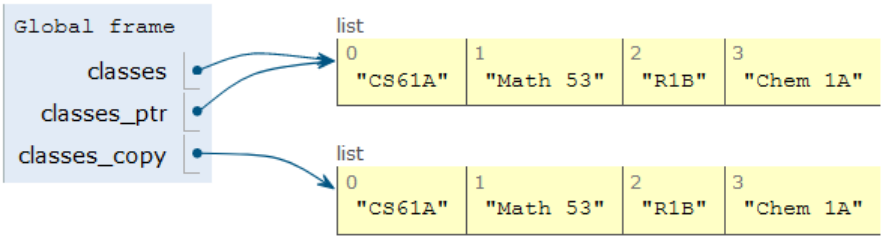
```
def contains_n(elem, n, t):
    """
    >>> t1 = tree(1, [tree(1, [tree(2)])])
    >>> contains_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = tree(1, [tree(2), tree(1, [tree(1), tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif _____________________________________________:

        return _____________________________________

    elif label(t) == elem:

        return _____________________________________

    else:

        return _____________________________________
```

```
def contains_n(elem, n, t):
    if n == 0:
        return True
    elif is_leaf(t):
        return n == 1 and label(t) == elem
    elif label(t) == elem:
        return True in [contains_n(elem, n - 1, b) for b in
          branches(t)]
    else:
        return True in [contains_n(elem, n, b) for b in
          branches(t)]
```

# 2 Mutability

Let's imagine it's your first year at Cal, and you have signed up for your first classes!

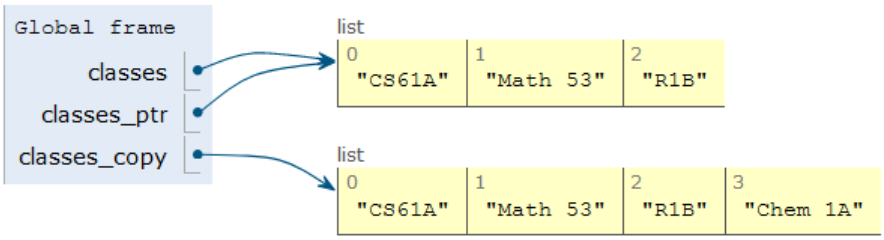
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```




After a few weeks, you realize that you cannot keep up with the workload and you need to drop a class. You've chosen to drop Chem 1A. Based on what we know so far, to change our classes list, we would have to create a new list with all the same elements as the original list except for Chem 1A. But that is silly, since all we really need to do is remove the Chem 1A element from our list.

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 creating a new list, we can just call `classes.pop()`, which removes the last element from the list.

```
>>> classes.pop() # pop returns whatever item it removed
"Chem 1A"
```



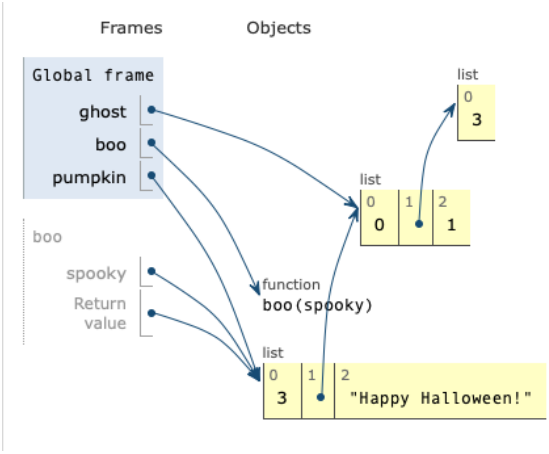

Here are more list methods that mutate:

**Mutability in Lists**

| Function | Create/Mutate | Action | Return Value |
|---|---|---|---|
| lst.**append**(element) | mutate | attaches element to end of the list | None |
| lst.**extend**(iterable) | mutate | attaches each element in iterable to end of the list | None |
| lst **+=** lst2 | mutate | attaches lst2 to the end of lst; same as lst.extend(lst2) | None |
| lst.**pop**() | mutate | removes last element from the list | Removed element |
| lst.**pop**(index) | mutate | removes element at index | Removed element |
| lst.**remove**(element) | mutate | removes element from the list | None |
| lst.**insert**(index, element) | mutate | inserts element at index and pushes rest of elements down | None |
| lst ***=** n | mutate | attaches lst to the end of lst (n - 1) times; same as calling lst.extend(lst) (n - 1) times | None |
| lst[start:end:step size] | create | creates a new list that includes elements from *start* index to *end* index - 1 while incrementing *step size* | new list |
| lst = lst **+** [1, 23] | create | creates a new list with elements from lst and [1, 23] | new list |
| lst = lst ***** n | create | creates a new list with elements from lst repeated n times | new list |
| **list**(iterable) | create | create new list with elements of iterable | new list |

1. Draw the environment diagram that results from running the following code.

```
ghost = [1, 0,[3], 1]
def boo(spooky):
  ghost.append(spooky.append(ghost))
  spooky = spooky[ghost[2][1][1]]
  ghost[:].extend([spooky])
  spooky = [spooky] + [ghost[spooky - 1].pop()]
  ghost.remove(ghost.remove(1))
  spooky += ["Happy Halloween!"]
  return spooky
pumpkin = boo(ghost[2])
```




PythonTutor Link

2. Given some list `lst` of numbers, mutate `lst` to have the accumulated sum of all elements so far in the list. If `lst` is a deep list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list. Your function should return an integer representing your "accumulated" sum (sum of all numbers in your list). You may not need all lines provided.

   *Hint:* The **`isinstance`** function returns True for **`isinstance`**`(l, list)` if `l` is a list and False otherwise.

```
def accumulate(lst):
    """
    >>> l = [1, 5, 13, 4]
    >>> accumulate(l)
    23
    >>> l
    [1, 6, 19, 23]
    >>> deep_l = [3, 7, [2, 5, 6], 9]
    >>> accumulate(deep_l)
    32
    >>> deep_l
    [3, 10, [2, 7, 13], 32]
    """
    sum_so_far = 0
    for ____________________________________________:

        ____________________________________________

        if isinstance(____________________, list):

            inside = ____________________________

            ____________________________________

        else:
            ____________________________________

            ____________________________________

    ________________________________
```

```
def accumulate(lst):
    sum_so_far = 0
    for i in range(len(lst)):
        item = lst[i]
        if isinstance(item, list):
            inside = accumulate(item)
            sum_so_far += inside
        else:
            sum_so_far += item
            lst[i] = sum_so_far
    return sum_so_far
```

3. **Scrabble!** [Exam Level - Adapted from CS61A Su19 Midterm Q7(b)]

   Implement `scrabbler` which takes in a string `chars`, a list of strings `words`, and a dictionary `values` which maps letters to point values. It should return a dictionary where each key is a word in `words` which can be formed from the letters in `chars` and each value is the point value of that word.

   For this problem, we will only consider words we can form using letters in `chars` in the same order they appear in the string. Assume values contains all the letters across valid words as keys.

   Furthermore, you have access to the function `is_subseq` which returns `true` if a string is a subsequence of another string. A string is a sequence of another string if all the letters in the first string appear in the second string in the same order (but they do not need to be next to each other).

```
def scrabbler(chars, words, values):
    """ Given a list of words and point values for letters, returns a
    dictionary mapping each word that can be formed from letters in chars
    to their point value. You may not need all lines
    >>> words = ["easy", "as", "pie"]
    >>> values = {"e": 2, "a": 2, "s": 1, "p": 3, "i": 2, "y": 4}
    >>> scrabbler("heuaiosby", words, values)
    {'easy': 9, 'as': 3}
    >>> scrabbler("piayse", words, values)
    {'pie': 7, 'as': 3}
    """
    result = ____________________________________

    for ____________________________________________:

        if ____________________________________________:

            ____________________________________

            ____________________________________

    return result
```

```
def scrabbler(chars, words, values):
    result = {}
    for w in words:
        if is_subseq(w, chars):
            total = sum([values[c] for c in w])
            result[w] = total
    return result
```

# 3 Iterators

On a conceptual level, **iterables** are simply objects whose elements can be iterated over. Think of an iterable as anything you can use in a **for** loop, such as ranges, lists, strings, or dictionaries.

On a technical level, iterables are a bit more complicated. An **iterator** is an object on which you can (repeatedly) call **next**, which will return the next element of a sequence. For example, if `it` is an iterator representing the sequence $1, 2, 3$, then we could do the following:

```
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
StopIteration
```

`StopIteration` is an exception that is raised when an iterator has no more elements to produce; it's how we know we've reached the end of an iterator. Iterators that will never produce a `StopIteration` exception are called *infinite*.

Under this regime, an iterable is formally defined as an object that can be turned into an iterator by passing it into the **iter** function. When you iterate over an iterable, Python first uses **iter** to create an iterator from the iterable and then iterates over the iterator. The simple **for** loop syntax abstracts away this fact.

There are a few useful functions that act on iterables that are particularly useful:

- **map**`(f, it)`: Returns an iterator that produces each element of `it` with the function `f` applied to it.
- **filter**`(pred, it)`: Returns an iterator that includes only the elements of `it` where the predicate function `pred` returns true.
- **reduce**`(f, it, init)`: Reduces `it` to a single value by repeatedly calling the two-argument function `f` on the elements of `it`: **reduce**`(add, [1, 2, 3])` $\rightarrow$ `6`. Optionally, an initializer may be provided: **reduce**`(add, [1], 5)` $\rightarrow$ `6`.
- **zip**`(it1, it2, ...)`: Returns an iterator of tuples where the first tuple has the first element of each iterable, second tuple has the second element of each iterable, etc.

1. What Would Python Display? If you think nothing will be displayed, write N/A.

```
>>> a = [1, 2, 3]
>>> x = iter(a)
```

```
N/A
```

```
>>> next(x)
```

```
1
```

```
>>> next(x)
```

```
2
```

```
>>> y = iter(a)
>>> next(y)
```

```
1
```

```
>>> next(x)
```

```
3
```

```
>>> next(x)
```

```
StopIteration Error
```

```
>>> z = iter(y) [2, 3]
>>> next(z)
```

```
2
```

```
>>> [next(y), next(y), next(z)]
```

```
[2, 3, 3]
```

```
>>> a = iter(filter(lambda x: x % 2, map(lambda x: x - 1, range(10))))
>>> next(a)
```

```
-1
```

```
>>> reduce(lambda x, y: x + y, a)
```

```
16
```