CS 61A Trees, Data Abstraction, Sequences
Fall 2021 Discussion 5: September 29, 2021 Solutions

Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects. For example, using code to represent cars, chairs, people, and so on. That way, programmers don't have to worry about *how* code is implemented; they just have to know *what* it does.

Data abstraction mimics how we think about the world. If you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of to do so. You just have to know how to use the car for driving itself, such as how to turn the wheel or press the gas pedal.

A data abstraction consists of two types of functions:

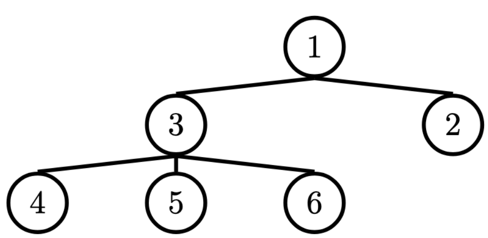
- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

Programmers design data abstractions to abstract away how information is stored and calculated such that the end user does *not* need to know how constructors and selectors are implemented. The nature of *abstraction* allows whoever uses them to assume that the functions have been written correctly and work as described.

Trees

One example of data abstraction is with **trees**.

In computer science, **trees** are recursive data structures that are widely used in various settings and can be implemented in many ways. The diagram below is an example of a tree.



Example Tree

Generally in computer science, you may see trees drawn "upside-down" like so. We say the **root** is the node where the tree begins to branch out at the top, and the **leaves** are the nodes where the tree ends at the bottom.

Some terminology regarding trees:

- **Parent Node**: A node that has at least one branch.
- **Child Node**: A node that has a parent. A child node can only have one parent.
- **Root**: The top node of the tree. In our example, this is the `1` node.
- **Label**: The value at a node. In our example, every node's label is an integer.
- **Leaf**: A node that has no branches. In our example, the `4`, `5`, `6`, `2` nodes are leaves.
- **Branch**: A subtree of the root. 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. We define this as the number of edges between the root to the node. As there are no edges between the root and itself, the root has depth 0. In our example, the `3` node has depth 1 and the `4` node has depth 2.
- **Height**: The depth of the lowest (furthest from the root) leaf. In our example, the `4`, `5`, and `6` nodes are all the lowest leaves with depth 2. Thus, the entire tree has height 2.

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

Tree Data Abstraction

A tree has a root value and a list of branches, where each branch is itself a tree.

The data abstraction specifies that calling `branches` on a tree `t` will give us a **list of branches**. Treating the return value of `branches(t)` as a list is then part of how we define trees.

How the entire tree `t` is implemented is under the abstraction barrier. Rather than assuming an implementation of `label` and `branches`, we will want to use those selector functions directly.

For example, we could choose to implement the tree data abstraction with a dictionary with separate entries for the `label` and the `branches`, or as a list with the first element being `label` and the rest being `branches`.

- The `tree` constructor takes in a value `label` for the root, and an optional list of branches `branches`. If `branches` isn't given, the constructor uses the empty list `[]` as the default.
- The `label` selector returns the value of the root, while the `branches` selector returns the list of branches of the tree.

With this in mind, we can create the tree from earlier using our constructor:

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

Questions

Q1: Tree Abstraction Barrier

Consider a tree `t` constructed by calling `tree(1, [tree(2), tree(4)])`. 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 1, the label of the entire tree. This is simply using a
selector to get the label, which is not violating any abstraction
barriers.
```

2. `t[0]`

```
This expression evaluates to 1, the label of the entire tree.
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`.
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. `is_leaf(t[1:][1])`

```
This expression accesses the branches of `t` by slicing `t`.
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])`.
```

5. `[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. The result list is `[2, 4]`.
It does not violate any abstraction barriers.
```

6. **Challenge:** `branches(tree(5, [t, tree(3)]))[0][0]`

```
This expression evaluates to the label of the tree `t`, which is 1.
This is because the expression `tree(5, [t, tree(3)])` evaluates to
a tree whose first branch is the tree `t` that we constructed above!
Howvever, this expression violates the abstraction barrier by
indexing into `t` to get its label. An equivalent expression would
be `label(branches(tree(5, [t, tree(3)]))[0])`.
```

Q2: Height

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
    >>> t = tree(3, [tree(1), tree(2, [tree(5, [tree(6)]), tree(1)])
    ])
    >>> height(t)
    3
    """
    if is_leaf(t):
        return 0
    return 1 + max([height(branch) for branch in branches(t)])
    # alternate solutions
    return 1 + max([-1] + [height(branch) for branch in branches(t)
    ])
    return max([1 + height(b) for b in branches(t)], default=0)
```

See video walkthrough

Q3: Maximum Path Sum

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.

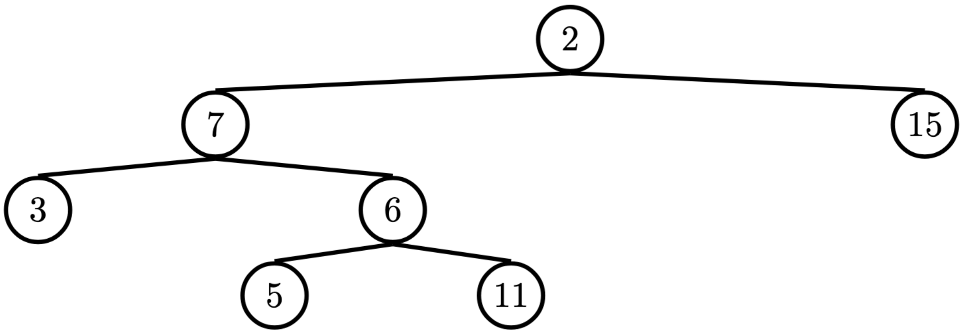
```
def max_path_sum(t):
    """Return the maximum path sum of the tree.

    >>> t = tree(1, [tree(5, [tree(1), tree(3)]), tree(10)])
    >>> max_path_sum(t)
    11
    """
    if is_leaf(t):
      return label(t)
    else:
      return label(t) + max([max_path_sum(b) for b in branches(t)])
```

Q4: Find Path

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]`



Example Tree

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

See video walkthrough

Sequences

Sequences are ordered collections of values that support element-selection and have length. We've worked with lists, but other Python types are also sequences, including strings.

Q5: Map, Filter, Reduce

Many languages provide `map`, `filter`, `reduce` functions for sequences. Python also provides these functions (and we'll formally introduce them later on in the course), but to help you better understand how they work, you'll be implementing these functions in the following problems.

> In Python, the `map` and `filter` built-ins have slightly different behavior than the `my_map` and `my_filter` functions we are defining here.

`my_map` takes in a one argument function `fn` and a sequence `seq` and returns a list containing `fn` applied to each element in `seq`.

```
def my_map(fn, seq):
    """Applies fn onto each element in seq and returns a list.
    >>> my_map(lambda x: x*x, [1, 2, 3])
    [1, 4, 9]
    """
    result = []
    for elem in seq:
        result += [fn(elem)]
    return result
```

`my_filter` takes in a predicate function `pred` and a sequence `seq` and returns a list containing all elements in `seq` for which `pred` returns True.

```
def my_filter(pred, seq):
    """Keeps elements in seq only if they satisfy pred.
    >>> my_filter(lambda x: x % 2 == 0, [1, 2, 3, 4])  # new list
    has only even-valued elements
    [2, 4]
    """
    result = []
    for elem in seq:
        if pred(elem):
            result += [elem]
    return result
```

`my_reduce` takes in a two argument function `combiner` and a non-empty sequence `seq` and combines the elements in `seq` into one value using `combiner`.

```
def my_reduce(combiner, seq):
    """Combines elements in seq using combiner.
    seq will have at least one element.
    >>> my_reduce(lambda x, y: x + y, [1, 2, 3, 4])  # 1 + 2 + 3 + 4
    10
    >>> my_reduce(lambda x, y: x * y, [1, 2, 3, 4])  # 1 * 2 * 3 * 4
    24
    >>> my_reduce(lambda x, y: x * y, [4])
    4
    >>> my_reduce(lambda x, y: x + 2 * y, [1, 2, 3]) # (1 + 2 * 2) +
     2 * 3
    11
    """
    total = seq[0]
    for elem in seq[1:]:
        total = combiner(total, elem)
    return total
```

Q6: Count palindromes

Write a function that counts the number of palindromes (any word that reads the same forwards as it does when read backwards) in a list of words using only `lambda` , string operations, conditional expressions, and the functions we defined above (`my_filter`, `my_map`, `my_reduce`). Specifically, do not use recursion or any kind of loop.

```
def count_palindromes(L):
    """The number of palindromic words in the sequence of strings
    L (ignoring case).

    >>> count_palindromes(("Acme", "Madam", "Pivot", "Pip"))
    2
    """
    return len(my_filter(lambda s: s.lower() == s[::-1].lower(), L))
```

Hint: The easiest way to get the reversed version of a string `s` is to use the Python slicing notation trick `s[::-1]`. Also, the function `lower`, when called on strings, converts all of the characters in the string to lowercase. For instance, if the variable `s` contains the string "PyThoN", the expression `s.lower()` evaluates to "python".

Additional Practice

Q7: Perfectly Balanced

Part A: Implement `sum_tree`, which returns the sum of all the labels in tree `t`.

Part B: Implement `balanced`, which returns whether every branch of `t` has the same total sum and that the branches themselves are also balanced.

Challenge: Solve both of these parts with just 1 line of code each.

```
def sum_tree(t):
    """
    Add all elements in a tree.
    >>> t = tree(4, [tree(2, [tree(3)]), tree(6)])
    >>> sum_tree(t)
    15
    """
    total = 0
    for b in branches(t):
        total += sum_tree(b)
    return label(t) + total
```

```
def balanced(t):
    """
    Checks if each branch has same sum of all elements and
    if each branch is balanced.
    >>> t = tree(1, [tree(3), tree(1, [tree(2)]), tree(1, [tree(1),
    tree(1)])])
    >>> balanced(t)
    True
    >>> t = tree(1, [t, tree(1)])
    >>> balanced(t)
    False
    >>> t = tree(1, [tree(4), tree(1, [tree(2), tree(1)]), tree(1, [
    tree(3)])])
    >>> balanced(t)
    False
    """
    for b in branches(t):
        if sum_tree(branches(t)[0]) != sum_tree(b) or not balanced(b
    ):
            return False
    return True
```

Q8: Hailstone Tree

We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number `n`, continuing to `n/2` if `n` is even or `3n+1` if `n` is odd, ending with 1. Write a function `hailstone_tree(n, h)` which generates a tree of height `h`, containing hailstone numbers that will reach `n`.

> **Hint:** A node of a hailstone tree will always have at least one, and at most two branches (which are also hailstone trees). Under what conditions do you add the second branch?

```python
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will reach N, with height H.
    >>> print_tree(hailstone_tree(1, 0))
    1
    >>> print_tree(hailstone_tree(1, 4))
    1
        2
            4
                8
                    16
    >>> print_tree(hailstone_tree(8, 3))
    8
        16
            32
                64
            5
                10
    """
    if h == 0:
        return tree(n)
    branches = [hailstone_tree(n * 2, h - 1)]
    if (n - 1) % 3 == 0 and ((n - 1) // 3) % 2 == 1 and (n - 1) // 3 > 1:
        branches += [hailstone_tree((n - 1) // 3, h - 1)]
    return tree(n, branches)

def print_tree(t):
    def helper(i, t):
        print("    " * i + str(label(t)))
        for b in branches(t):
            helper(i + 1, b)
    helper(0, t)
```