HIERARCHICAL DATA, ORDERS OF GROWTH 5

COMPUTER SCIENCE 61A

July 8, 2014

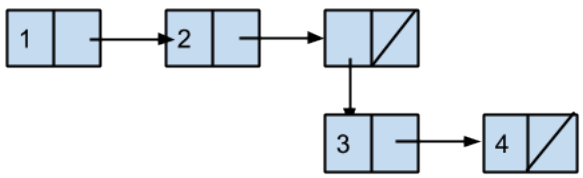
1 Introduction and Warm-Up

What does it mean for data to be structured *hierarchically*? When we talk about hierarchical data structures, we are generally referring to structures which store data in "levels". Descending deeper into a hierarchical data structure is recursive. Each new level is the same "type" of level as the one above it: trees are made of subtrees, deep linked lists are made of, well, other deep linked lists. As you go deeper into the structure, you will eventually reach an end, similar to how we can eventually reach our base case when recursing. Visualizing the levels which compose a hierarchical data structure, and thinking of their recursive nature, can help you tackle functions which traverse and explore these structures in a variety of interesting and useful ways.

1.1 Basics

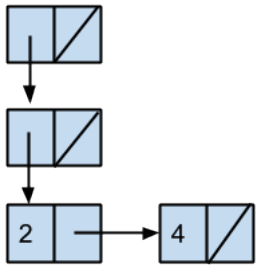
1. Draw a diagram to represent the following deep linked lists. If you need help, theres an example diagram on the following page.
   - `link(1, link(2, link(link(3, link(4, empty)), empty)))`

     Solution:

   - `link(link(link(2, link(4, empty)), empty), empty)`

Solution:



2 Deep Linked Lists

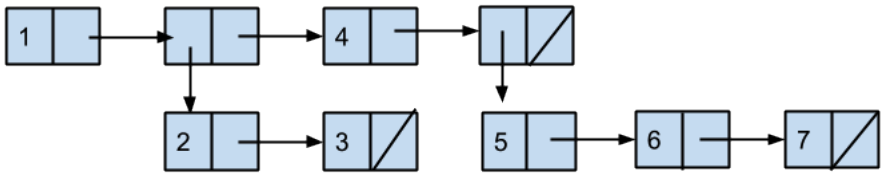
One example of a hierarchical structure which we've already seen briefly are "deep linked lists". As a reminder, linked lists are an ADT which takes in a "first", which can be anything, and a "rest", which must be another linked list.
Constructor: `link(first, rest)`

- `first` can be anything
- `rest` is a linked list, including the empty list value `empty`

"Deep Linked Lists" or recursive linked lists are different from most of the linked lists weve seen in that their `first` can itself also be another linked list, instead of just a single primitive type like a number. For example, the following is a deep list:

```
link(1, link(link(2, link(3, empty)), \
             link(4, link(link(5, link(6, link(7, empty))), empty))))
```

and can be represented in the level-like structure below.



We've already briefly delved into this added layer of complexity. In Discussion 3, you saw the following problem:

Write a function `deep_sum_linked_list` which returns the sum of all the numbers in a recursive linked list. The elements of a recursive linked list are all either numbers or other recursive linked lists. Assume you have `empty` and `is_linked_list` defined.

```
def deep_sum_linked_list(lst):
```

How do we think about this problem? First, we could think about what we'd do without `first` being a recursive list. Thats much simpler: we have our base case for the empty list, then our recursive step where we add `first(lst)` to the recursive sum of the rest of the linked list.

```
def deep_sum_linked_list(lst):
    if lst == empty:
        return 0
    return first(lst) + deep_sum_linked_list(rest(lst))
```

The case when `first` can also be a deep list is only a bit different, with an added recursive condition: we just have to check the case of `first` being a linked list, and if it is, apply `deep_sum_linked_list` to it.

```
def deep_sum_linked_list(lst):
    if lst == empty:
        return 0
    elif is_linked_list(first(lst)):
        return deep_sum_linked_list(first(lst)) + \
               deep_sum_linked_list(rest(lst))
    return first(lst) + deep_sum_linked_list(rest(lst))
```

Again, thinking about the recursive nature of hierarchically structured data is key to implementing functions which handle it.

2.1 Questions

1. Implement the function `deep_filter`, which takes a recursive linked list `lst` and a condition `pred` and returns a new recursive linked list which only contains elements which satisfy the condition. Assume you have `empty` and `is_linked_list` defined.

```
def deep_filter(pred, lst):
    """
    >>> r1 = link(1, link(link(2, empty), link(3, empty)))
    >>> print_linked_list(r1)
    < 1 < 2 > 3 >
    >>> def even(num):
    ...     return num % 2 == 0
    >>> print_linked_list(deep_filter(even, r1))
    < < 2 > >
    """
```

Solution:

```
if lst == empty:
    return empty
elif is_linked_list(first(lst)):
    return link(deep_filter(pred, first(lst)), deep_filter(pred, re
elif pred(first(lst)):
    return link(first(lst), deep_filter(pred, rest(lst)))
return deep_filter(pred, rest(lst))
```

2. Implement the function `deep_length`, which determines the number of elements in a given deep linked list `lst`.

```
def deep_length(lst):
    """
    >>> deep_length(r1)  # See deep_filter for r1
    3
    >>> deep_length(link(link(1, link(2, empty)), link(3, link(4, empty))))
    4
    """
```

Solution:

```
if lst == empty:
    return 0
elif is_linked_list(first(lst)):
    return deep_length(first(lst)) + deep_length(rest(lst))
return 1 + deep_length(rest(lst))
```

3. Implement the function `present` which returns `True` if the element `el` is in the deep linked list `lst`, and `False` if it is not.

```
def present(lst, el):
    """
    >>> present(r1, 3)  # See deep_filter for r1
    True
    >>> present(r1, 5)
    False
    >>> present(r1, 2)
    True
    """
```

Solution:

```
if lst == empty:
    return False
elif is_linked_list(first(lst)):
    return present(first(lst), el) or present(rest(lst), el)
return first(lst) == el or present(rest(lst), el)
```

4. (Challenging) Implement the function `find_el` which returns the element at the index `ind`. You may use any of your previously defined functions to help you.

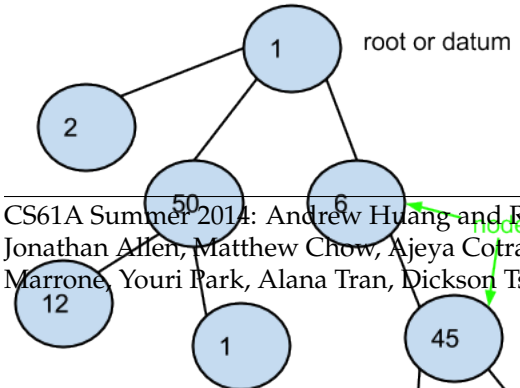
```
def find_el(lst, ind):
    """
    >>> find_el(r1, 0)  # See deep_filter for r1
    1
    >>> find_el(r1, 1)
    2
    """
```

Solution:

```
if is_linked_list(first(lst)):
    first_len = deep_length(first(lst))
    if first_len > ind:
        return find_el(first(lst), ind)
    return find_el(rest(lst), ind - first_len)
elif ind == 0:
    return first(lst)
return find_el(rest(lst), ind - 1)
```

3 Trees

Trees are another hierarchical data type which has a lot of applications, from dissecting syntax to organizing evolutionary lines.
The diagram below shows a tree structure.



```
spruce = tree(1,
  [tree(2),
    tree(50,
      [tree(12),
       tree(1)]),
                                    tree(6,
                                      [tree(45,
                                        [tree(7),
                                         tree(9)])])])
```

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.
Trees consist of two components: the datum and the children.

1. **Datum**: Each tree has one item (datum). The data could be numbers, strings, lists, functions, more trees, etc.
2. **Children**: This is a Python list containing all of its children (each of which are trees).

Some more terminology regarding trees:

- **Parent node**: A node that has children. Parent nodes can have multiple children.
- **Child node**: A node that has a parent. A child node can only have one parent.
- **Root**: The top node. There is only one root. Because every other node branches directly or indirectly from the root, it is possible to start from the root and reach any other node in the tree. The root is the only node that does not have a parent. For example, the node that contains the 1 at the top is the root.
- **Leaf**: Nodes that have no children. For example, the nodes that contain the bottom 2, 12, 1, 9, and 7 are leaves.
- **Subtree**: Notice that each child of a parent is itself the root of a smaller tree (for example, the node containing 45 is the root of another tree which contains the nodes containing 7 and 9). This is why trees are recursive data structures: trees are made up of subtrees, which are trees themselves.
- **Depth**: How far away a node is from the root. In the diagram, the node containing 7 has depth 3; the node containing 6 has depth 1. The root has a depth of 0.
- **Height**: The depth of the lowest leaf. In the diagram, the nodes containing 9 and 7 are all the "lowest leaves," and they have depth 3. Thus, the entire tree has height 3.

The constructor for the tree ADT is:

```
def tree(datum, children=[])
```

- `datum` is the item that is stored in the root of the tree
- `children` is a list of child trees, each of which is itself a tree. The default value of children is the empty list [], which would make the resulting tree a leaf.

The selectors for the tree ADT are:

- `datum(t)` returns the item stored in the root of tree `t`
- `children(t)` returns a list of the children of the tree `t`. If `t` is a leaf, the return value will be an empty list.

Remember that trees are an Abstract Data Type. Use the constructors and selectors to deal with the data held within the tree, or else risk violating an abstraction barrier and having your computer catch on fire out of guilt.

3.1 Questions

1. Implement the function `height` which returns the height of the tree `t`.

```
def height(t):
    """
    >>> height(spruce)  # See previous diagram
    3
    """
```

Solution:

```
    if children(t) == []:
        return 0
    else:
        current_max = 0
        for child in children(t):
            current_max = max(current_max, height(child))
        return 1 + current_max
```

2. Implement the function `count_leaves` which returns the number of leaves in the tree `t`.

```
def count_leaves(t):
    """
    >>> count_leaves(spruce)  # See previous diagram
    5
    """
```

Solution:

```
    if children(t) == []:
```

```
        return 1
    result = 0
    for child in children(t):
        result += count_leaves(child)
    return result
```

3. Implement the function `find_path`, which returns the elements in the path from the root of the tree `t`, to the given element `el`.

```
def find_path(t, el):
    """
    >>> find_path(spruce, 7)  # See previous diagram
    [1, 6, 45, 7]
    """
```

Solution:

```
    if children(t) == [] and datum(t) != el:
        return False
    elif datum(t) == el:
        return [datum(t)]
    else:
        for child in children(t):
            c_path = find_path(child, el)
            if c_path:
                return [datum(t)] + c_path
        return False
```

4. Implement the function `tree_rev` which returns a new tree that is the reverse of the tree `t`. So if I have a tree `t` with the children `A, B, C`, originally appearing in that order, the `tree_rev(t)` would have the children in the order `C, B, A`.

```
def tree_rev(t):
    """
    >>> ecurps = tree_rev(spruce)
    >>> [datum(t) for t in children(ecurps)]
    [6, 50, 2]
    >>> grandchildren = [children(t) for t in children(ecurps)]
    >>> [[datum(t) for t in lst] for lst in grandchildren]
    [[45], [1, 12], []]
```

```
    """
```

Solution:

```
def forest_rev(forest):
    new_forest = []
    for tree in forest[::-1]:
        new_forest += [tree_rev(tree)]
    return new_forest
return tree(datum(t), forest_rev(children(t)))
```

4 Orders of Growth

In CS 61A, were not that concerned with rigorous mathematical proofs. (Youll get the painful details in CS 61B and CS 170!) What we want you to develop in CS 61A is the intuition to figure out the orders of growth without having to delve into mathematical proofs.

Here are some common orders of growth, ranked from best to worst:

- $\Theta(1)$ constant time. Takes the same amount of time regardless of input size
- $\Theta(\log(n))$ logarithmic time
- $\Theta(n)$ linear time
- $\Theta(n^3)$, etc. polynomial time
- $\Theta(2^n)$ exponential time ("intractable"; these are really, really slow)

4.1 Orders of Growth in Time

"Time," for us, basically refers to the number of recursive calls or the number of times the suite of a `while` loop executes. Intuitively, the more recursive calls we make, the more time it takes to execute the function.

- If the function contains only primitive procedures like $+$ or $*$, then it is constant time, or $\Theta(1)$.
- If the function is recursive, you need to:
  - Count the number of recursive calls that will be made given input n
  - Count how much time it takes to process the input per recursive call.

The answer is usually the product of the above two. For example, lets try to sum all of the items in a linked list recursively. Each addition takes us 3 seconds, and there are 10 items. Therefore, we would take 3 * 10 = 30 seconds to calculate the summation. The same principle applies when we are dealing with n and $log(n)$ and things like that.

- If the function contains calls of helper functions that are not constant-time, then you need to take orders of growth of the helper functions into consideration as well. In general, how much time the helper function takes would be included.
- When we talk about orders of growth, we dont really care about constant factors. So if you get something like $\Theta(1000000 * n)$, this is really $\Theta(n)$. We can also usually ignore lower-order terms. For example, if we get something like $\Theta(n^3 + n^2 + 4 * n + 399)$, we can take it to be $\Theta(n^3)$. This is because the general shape of the plot for $n^3 + n^2 + 4 * n + 399$ is very similar to n^3 as we look towards where n approaches infinity. However, note that in practice, these extra terms and constant factors are important. If a program f takes $1000000 * n$ time whereas another function g takes n time, both functions will be $\Theta(n)$, even though function g is clearly faster (and better).

4.2 Questions

What is the order of growth in time for the following functions?

```
def calculate(n):
    return n*2
```

Solution: $\Theta(1)$

```
def cry(n):
    for i in range(n):
        print("boo")
        for i in range(n):
            print("hoo")
```

Solution: $\Theta(n^2)$

```
def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)
```

Solution: $\Theta(n^2)$

```
def bar(n):
    return n + 1 if n % 2 == 1 else n

def foo(n):
    if n < 1:
        return 2
    if n % 2 == 0:
        return foo(n - 1) + foo(n - 2)
    else:
        return 1 + foo(n - 2)
```

What is the order of growth of `foo(bar(n))`

Solution: $\Theta(n^2)$

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Solution: $\Theta(1)$

```
def bonk(n):
   sum = 0
   while n >= 2:
       sum += n
       n = n / 2
   return sum
```

Solution: $\Theta(log(n))$

4.3 Harder Question

1. Given the following function `f`, give the running times of the functions `g` and `h` in terms of `n`. If you really want a challenge, try to find the running time of the function

`f` in terms of `x` and `y`.

```
def f(x, y):
    if x == 0 or y == 0:
        return 1
    if x < y:
        return f(x, y-1)
    if x > y:
        return f(x-1, y)
    else:
        return f(x-1, y) + f(x, y-1)

def g(n):
    return f(n, n)

def h(n):
    return f(n, 1)
```

Solution: Run time of `g`: $\Theta(2^n)$ Run time of `h`: $\Theta(n)$ Run time of `f`: $\Theta(2^{min(x,y)} + |x - y|)$