CSC148 winter 2018

recursive structures
week 7

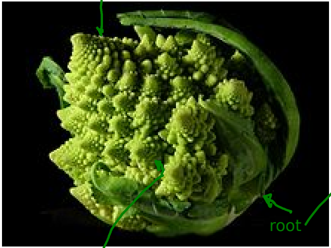
Danny Heap
heap@cs.toronto.edu / BA4270 (behind elevators)
http://www.teach.cs.toronto.edu/~csc148h/winter/
416-978-5899

February 26, 2018

Computer Science
UNIVERSITY OF TORONTO

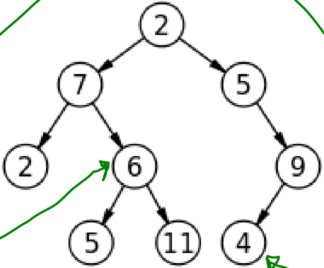
recursion, natural and otherwise

leaf



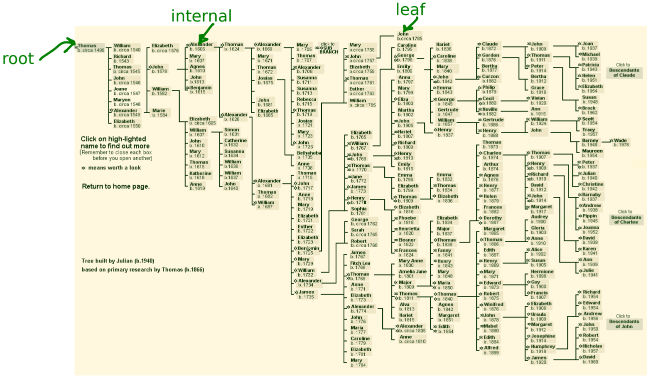
root

internal node



structure to organize information

patriarchal family tree...



terminology

- set of **nodes** (possibly with values or labels), with directed **edges** between some pairs of nodes
- One node is distinguished as **root** edges are directed away from root
- Each non-root node has exactly one parent.

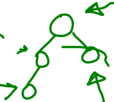
  ...root has zero!
- A **path** is a sequence of nodes $n_1, n_2, \ldots, n_k$, where there is an edge from n_i to n_{i+1}. The **length** of a path is the number of edges in it

  paths lead away from the root because edges do...
  there is a path of length 0 from a node to itself...
- There is a unique path from the root to each node. In the case of the root itself this is just n_1, if the root is node n_1.

  unique: one and only one, aka exactly one...
- There are no **cycles** no paths that form loops.

  compare with csc165 definition of tree...

more terminology



- **leaf:** node with no children
- **internal node:** node with one or more children
- **subtree:** tree formed by any tree node together with its descendants and the edges leading to them.

- **height:** 1 + the maximum path length from the root to some leaf.

  NB: Some texts use a different definition. Be sure you know which!

- **depth:** length of a path from root to a node is the node's depth.

  root — depth = 0
  children of root - depth = 1

- **arity, branching factor:** maximum number of children for any node.

general tree implementation

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree
    """
    def __init__(self, value: object=None,
                 children: List[Tree]=None) -> None:
        """
        Create Tree self with content value and 0 or more children
        """
        self.value = value
        # copy children if not None
        self.children = children[:] if children are not None else []
```

This approach avoids
the dreaded
mutable default parameter...

shallow copy...

general form of recursion:

if ⟨condition to detect a base case⟩:

Figure out 1 or more base cases and how to deal with them...

⟨do something without recursion⟩

else: # ⟨general case⟩

Anything that isn't a base case involves 1 or more recursive calls. Assume they work correctly and figure out how to combine them...

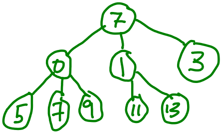
⟨do something that involves recursive call(s)⟩

how many leaves?

```
def leaf_count(t: Tree) -> int:
    """
    Return the number of leaves in Tree t.

    >>> t = Tree(7)
    >>> leaf_count(t)
    1
    >>> t = descendants_from_list(Tree(7),
                                  [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> leaf_count(t)
    6
    """
```

descendants_from_list used to build example (below)



Base case when t is a leaf... how many leaves is that?

General case: sum up the leaves in t's children...

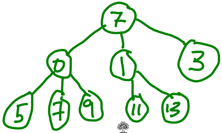
height of this tree?

```
def height(t: Tree):
    """
    Return 1 + length of longest path of t.

    >>> t = Tree(13)
    >>> height(t)
    1
    >>> t = descendants_from_list(Tree(13),
                                  [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> height(t)
    3
    """
    # 1 more edge than the maximum height of a child, except
    # what do we do if there are no children?
```

base case when t is a leaf... how tall is a leaf?

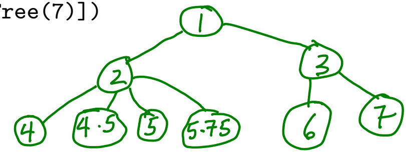
general case: maximum height of children + 1...



arity, or branching factor

```
def arity(t: Tree) -> int:
    """
    Return the maximum branching factor (arity) of Tree t.

    >>> t = Tree(23)
    >>> arity(t)
    0
    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(7)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> arity(tn1)
    4
    """
```



filesystem example

```
from os import scandir, path
from tree import Tree


def path_to_tree(path_name: str) -> Tree:
    """
    Return a Tree representing filesystem starting from pathname.
    """
    return Tree((path_name, [f.name for f in scandir(path_name)]),
                [path_to_tree(path.join(path_name, f.name))
                 for f in scandir(path_name)
                 if f.is_dir()])
```

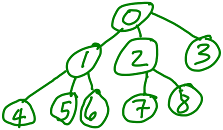
Try this on your computer...

pass in a function

```
def count_if(t: Tree, p: Callable[[object], bool]) -> int:
    """
    Return number of values in Tree t that satisfy predicate p(value).

    Assume predicate p is defined on t’s values

    >>> def p(v): return v > 4
    >>> t = descendants_from_list(Tree(0),
                                  [1, 2, 3, 4, 5, 6, 7, 8], 3)
    >>> count_if(t, p)
    4
    >>> def p(v): return v % 2 == 0
    >>> count_if(t, p)
    5
    """
```



base case: t is a leaf... return 0 if its value doesn't satisfy the predicate, otherwise 1...

general case: t has some children... return all the counts for t's children plus 1 if t's value satisfies the predicate...

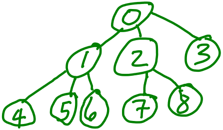
list the leaves

```
def list_leaves(t: Tree) -> int:
    """
    Return list of values in leaves of t.

    >>> t = Tree(0)
    >>> list_leaves(t)
    [0]
    >>> t = descendants_from_list(Tree(0),
                                          [1, 2, 3, 4, 5, 6, 7, 8], 3)
    >>> list_ = list_leaves(t)
    >>> list_.sort() # so list_ is predictable to compare
    >>> list_
    [3, 4, 5, 6, 7, 8]
    """
```

base case, t is a leaf: list of t's value

general case: concatenate lists of t's childrens leaves...



traversal

The functions and methods we have seen get information from every node of the tree   in some sense they traverse the tree.

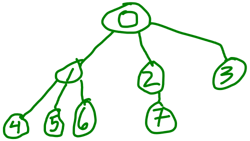
Sometimes the **order** of processing tree nodes is important: do we process the root of the tree (and the root of each subtree...) **before** or **after** its children? Or, perhaps, we process along **levels** that are the same distance from the root?

pre-order visit

```
def preorder_visit(t: Tree, act: Callable[[Tree], Any]) -> None:
    """
    Visit each node of Tree t in preorder, and act on the nodes
    as they are visited.

    >>> t = descendants_from_list(Tree(0),
                                  [1, 2, 3, 4, 5, 6, 7], 3)
    >>> def act(node): print(node.value)
    >>> preorder_visit(t, act)
    0
    1
    4
    5
    6
    2
    7
    3
    """
    act(t)
    for c in t.children:
        preorder_visit(c, act)
```

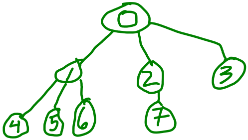
see page 19 for a diagram...



postorder

```
def postorder_visit(t: Tree, act: Callable[[Tree], Any) -> None:
    """
    Visit each node of t in postorder, and act on it when it is visited

    >>> t = descendants_from_list(Tree(0),
                                  [1, 2, 3, 4, 5, 6, 7], 3)
    >>> def act(node): print(node.value)
    >>> postorder_visit(t, act)
    4
    5
    6
    1
    7
    2
    3
    0
    """
```

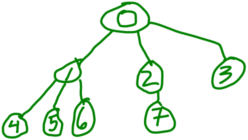


See code on page 15, and change the order slightly...

levelorder

```
def levelorder_visit(t: Tree, act: Callable[[Tree], Any]) -> None:
    """
    Visit every node in Tree t in level order and act on the node
    as you visit it.

    >>> t = descendants_from_list(Tree(0),
                                  [1, 2, 3, 4, 5, 6, 7], 3)
    >>> def act(node): print(node.value)
    >>> levelorder_visit(t, act)
    0
    1
    2
    3
    4
    5
    6
    7
    """
```



Use either a queue, or a helper method that visits all the nodes at a given depth...

queues, stacks, recursion

You may have noticed in the last slide there were no recursive calls, and a **queue** was used to process a recursive structure in level order.

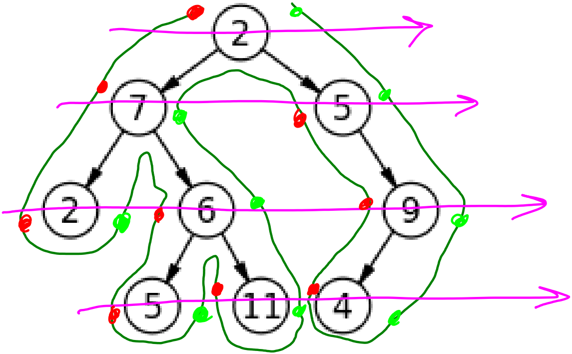
Careful use of a **stack** allows you to process a tree in preorder

preorder tracing...

preorder

postorder

levelorder



notes...