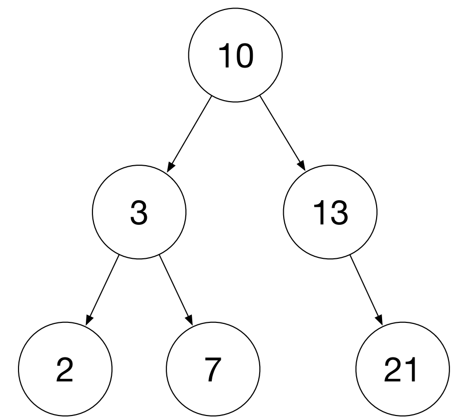
A crash course in binary trees

We'll revisit in MSDS689 but we need binary trees for projects now

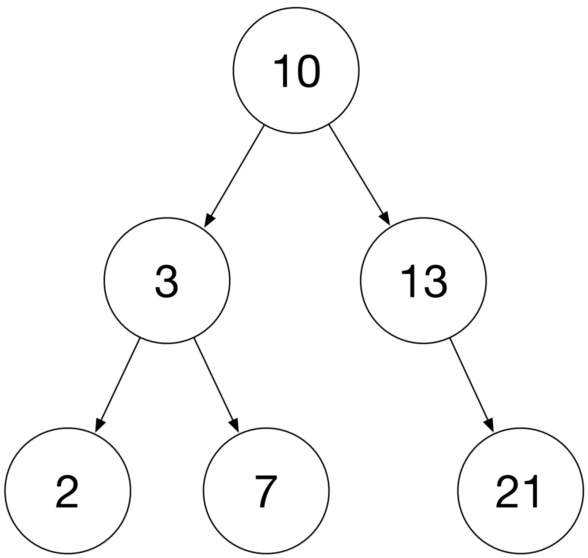
Terence Parr
MSDS program
University of San Francisco

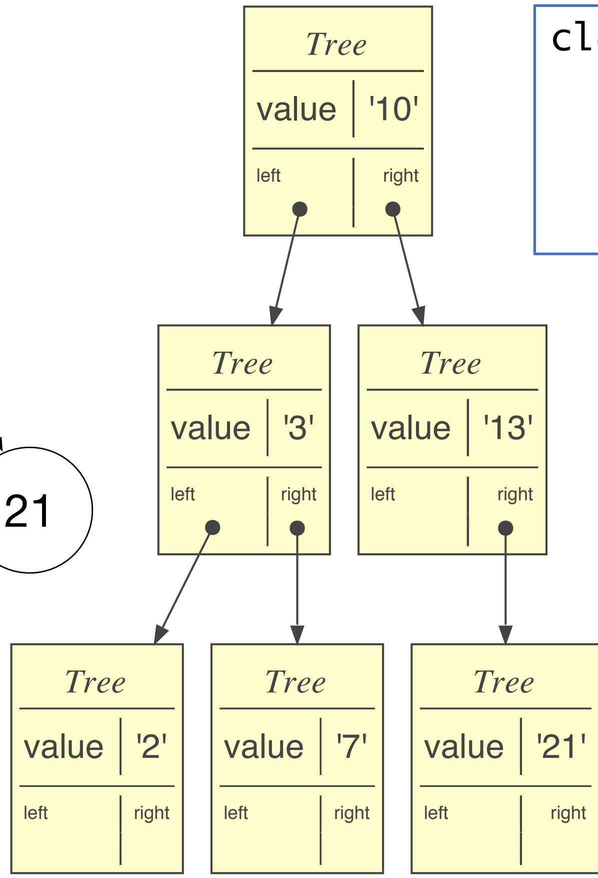
Binary tree abstract data structure

- A directed-graph with internal nodes and leaves
- No cycles and each node has at most one parent
- Each node has at most 2 child nodes
- For n nodes, there are n-1 edges
- Nodes have payloads (values), can be anything
- A *full* binary tree: all internal nodes have 2 children
- Height of full tree with n internal nodes is about log2(n)
- Height defined as number of edges along path root→leaf
- Level 0 is root, level 1, …
- Note: binary tree doesn't imply *binary search tree*



Concrete binary tree using pointers





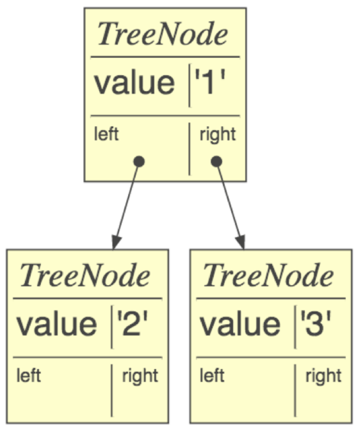

```python
class TreeNode:
  def __init__(self, value, left, right):
    self.value = value
    self.left = left
    self.right = right
```

Building binary trees

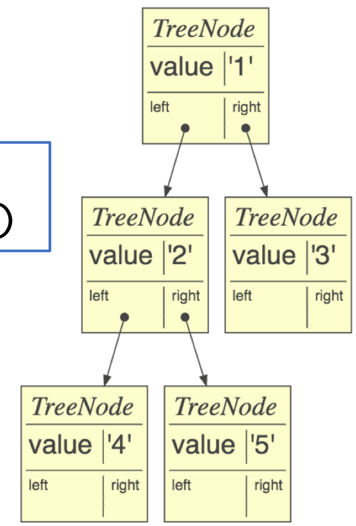
```
class TreeNode:
  def __init__(self, value, left, right):
    self.value = value
    self.left = left
    self.right = right
```

- Manual construction is a simple matter of creating nodes and setting left/right child pointers or passing kids to init

```
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
```




```
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
```



or

```
root = TreeNode(1, TreeNode(2), TreeNode(3))
```

https://github.com/parrt/msds621/blob/master/notebooks/trees/basics.ipynb

Recursion detour

Math recurrence relations => recursion

- Factorial definition:
  - Let 0! = 1
  - Define n! = n * (n-1)! for n>=1
- Recurrent math function calls become recursive function call in Python
- Non-recursive version is harder to understand and less natural

```
def fact(n):
    if n==0: return 1
    return n * fact(n-1)
```

```
def factloop(n):
    r = 1
    for i in range(1,n+1):
        r *= i
    return r
```

Recursion traces out a call graph

- Think of each call to function as node in chain or graph of calls
- Result of each function call is a piece of the result and each call combines subresult(s) to create more complete answer and passes it back

fact(5)

5 * fact(4)

4 * fact(3)

3 * fact(2)

2 * fact(1)

1 * fact(0)

1

Formula for writing recursive functions

def **f**(input):
1. check termination condition
2. process the active input region / current node, etc…
3. invoke f on subregion(s)
4. combine and return results

Steps 2 and 4 are optional

```
def fact(n):
    if n==0: return 1
    return n * fact(n-1)
```

Terminology: *currently-active region* or *element* is what function is currently trying to process. Here, that is argument n (the “region” is the numbers 0..n)

Don't let the recursion scare you

- Just pretend that you are calling a different function
- Or, pretend that you are calling the same function except that it is known to be correct already
- We call this the *recursive leap of faith*
- Follow the “Formula for recursive functions” and all will be well!

Recursive tree procedures

An analogy for recursive tree walking

- Imagine searching for an item in a maze of rooms connected by doors (no cycles)
- Each room has at most 2 doors, some have none
- Search procedure that works in ANY room:

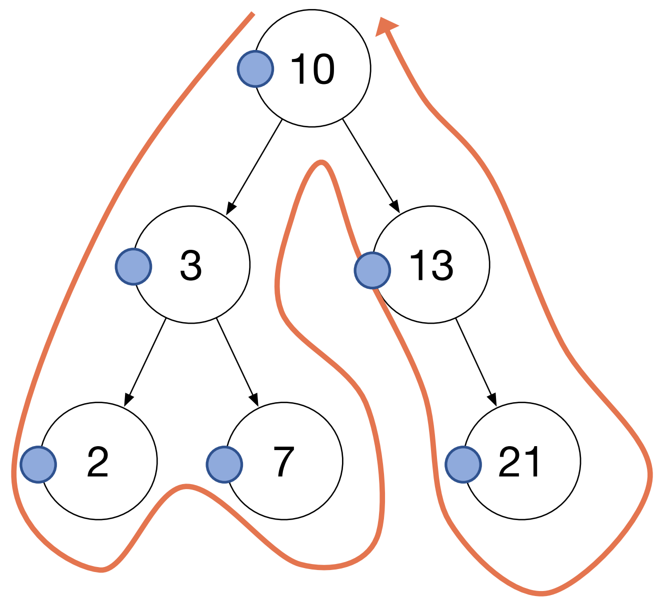
```
def visit(room):
    if item in room: print("rejoice!")
    if room.left exists: visit(room.left)
    if room.right exists: visit(room.right)
```

- This approach is called *backtracking*

Recursive tree walk is natural

- *Depth-first search* is how we walk (visit) through nodes
- *Pre-order traversal*: executing an action at discovery time, before visiting kids

```
def walk(p:TreeNode):
    if p is None: return
    print(p.value) # preorder
    walk(p.left)
    walk(p.right)
```



Think of launching a minion to walk the left subtree and another to walk the right

https://github.com/parrt/msds621/blob/master/notebooks/trees/basics.ipynb

How can walk() remember where it has visited?

- “Where to return” is tracked per function **call** not per function **definition**
- Funcion f calls g calls h and Python remembers where it came from
- Just imagine that f, g, and h are the same function and you'll see that recursion also remembers where it came from
- Each function call saves its place and `return` statement uses that as location to resume

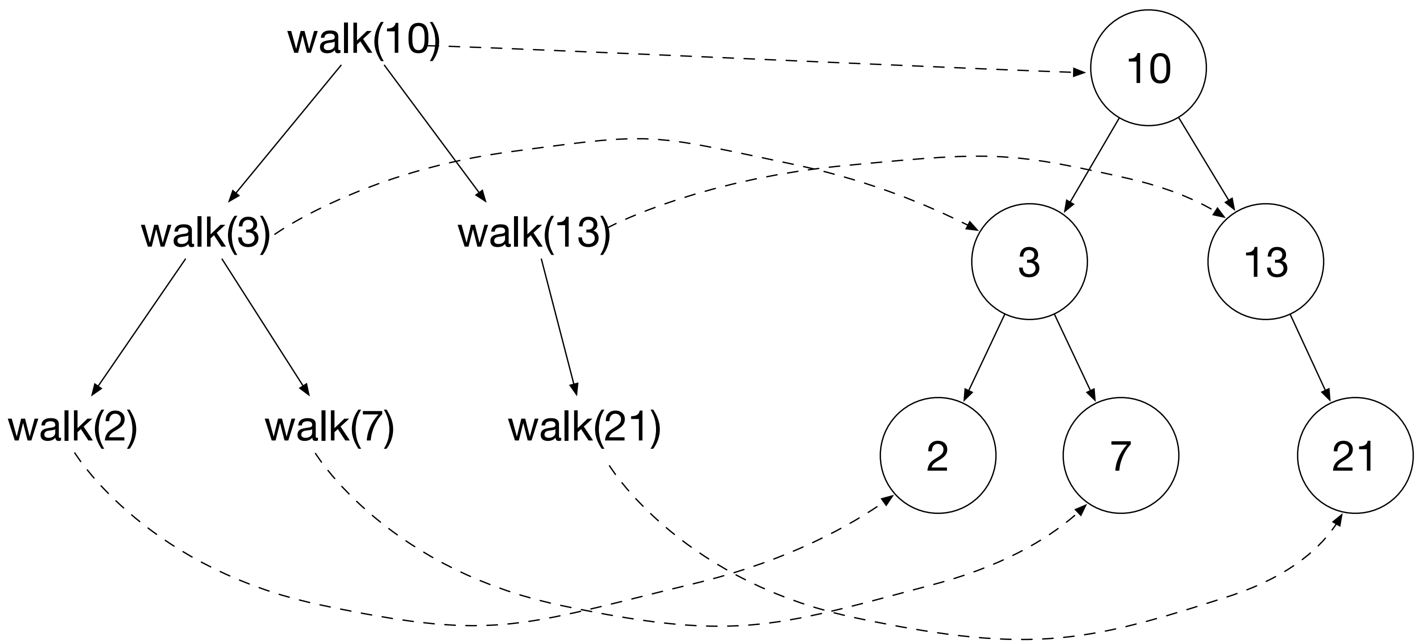
```python
def f():
    g()
    print("back from g()")

def g():
    h()
    print("back from h()")

def h():
    print("hi I'm h!")
```

```python
f()
print("back from h()")
```

```
hi I'm h!
back from h()
back from g()
back from h()
```

Recursion call tree vs tree

```
def walk(p:TreeNode):
    if p is None: return
    walk(p.left)
    walk(p.right)
```



Exhaustive search of all nodes

Searching in binary tree

- Let's modify the tree walker to search for an element and compare to unrestricted depth-first tree walk

```
def walk(p:TreeNode):
    if p is None: return
    walk(p.left)
    walk(p.right)
```

```
def search(p:TreeNode, x:object):
    if p is None: return None
    if x==p.value: return p
    q = search(p.left, x)
    if q is not None: return q
    q = search(p.right, x)
    return q
```

Decision tree stumps

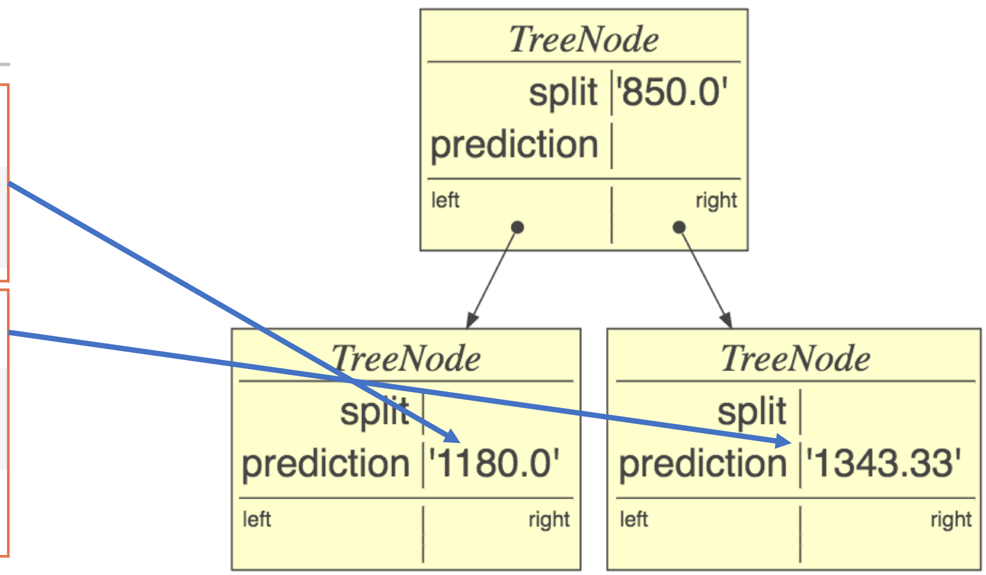
Stumps

- A stump is a 2-level tree w/decision node root & 2 predictor leaves
- Used by gradient boosting machines as the "weak learners"
- If node has split, it's a decision node else it's a leaf

```
# Define a single-node type for simplicity
class TreeNode:
  def __init__(self, split=None, prediction=None,
               left=None, right=None):
    self.split = split
    self.prediction = prediction
    self.left = left
    self.right = right
```

Sample stump that picks midpoint as split

sqfeet	rent
750	1160
800	1200
850	1280
900	1450
950	1300



Creating decision tree stumps

- For demonstration purposes, let's split x always at midpoint between min/max:

```
def stumpfit(x, y):
    if len(x)==1 or len(np.unique(x))==1:
        # if one x value, make leaf
        return TreeNode(prediction=y[0])
    split = (min(x) + max(x)) / 2 # midpoint
    t = TreeNode(split)
    t.left  = TreeNode(prediction=np.mean(y[x<split]))
    t.right = TreeNode(prediction=np.mean(y[x>=split]))
    return t
```

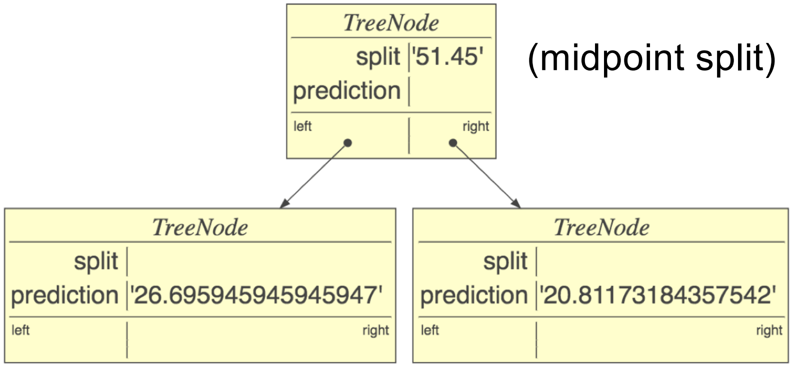
In practice, better to split node type

- See notebook for 1D decision tree implementation
  https://github.com/parrt/msds621/blob/master/notebooks/trees/decision-trees.ipynb

```
class DecisionNode:
    def __init__(self,split): # split chosen from x
        self.split = split

class LeafNode:
    def __init__(self,y):
        self.y = y
```

The magic of recursion

- Demo converting stumpfit() to treefit()
- See "*Regress tree midpoint split for Boston dataset*" in notebook
- In treefit(x,y), convert




```
t.left = TreeNode(prediction=np.mean(y[x<split]))
```

to

```
t.left = treefit(x[x<split], y[x<split])
```

Notebook: https://github.com/parrt/msds621/blob/master/notebooks/trees/decision-trees.ipynb

Key takeaways

- Binary tree: acyclic tree structure with at most two children, constructed by hooking nodes together (`root.left = TreeNode(2)`)
- Self-similar data structures walked and built with recursion
- Each recursive call does a piece of the work and returns its piece combined with results obtained from recursive calls
- Recursion traces out a tree that looks like the data structure
- Recursive call in treefit() returns newly-constructued subtree
- Remember the recursive function template!
- Depth-first-search visits each node through backtracking
- Study these recursive tree functions!