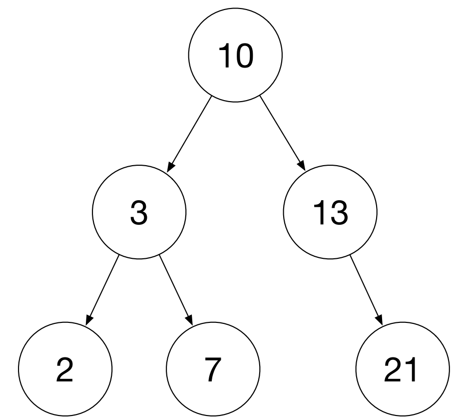
A crash course in binary trees

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

Terence Parr
MSDS program
University of San Francisco

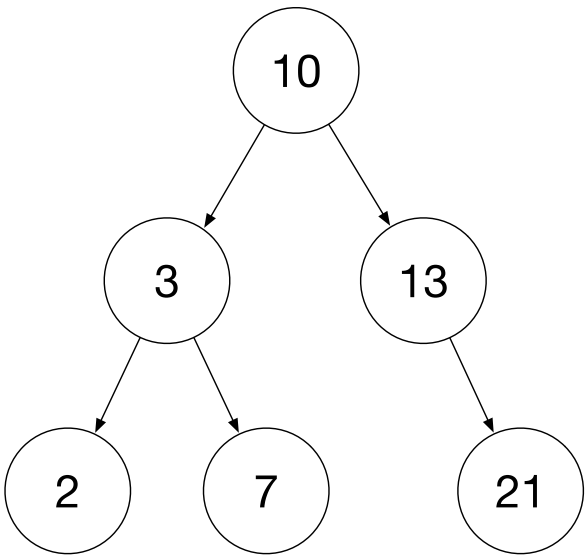
UNIVERSITY OF SAN FRANCISCO

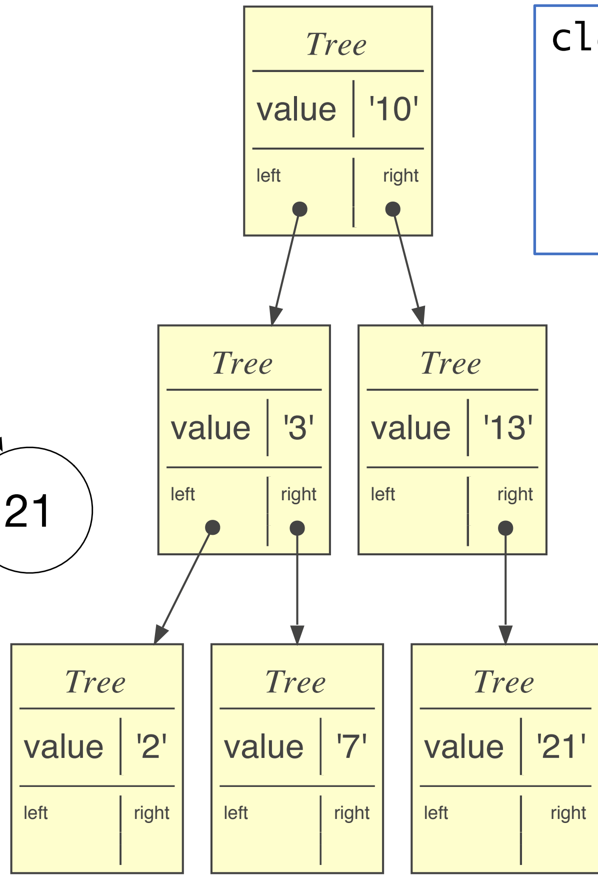
Binary tree abstract data structure



- 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
- 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

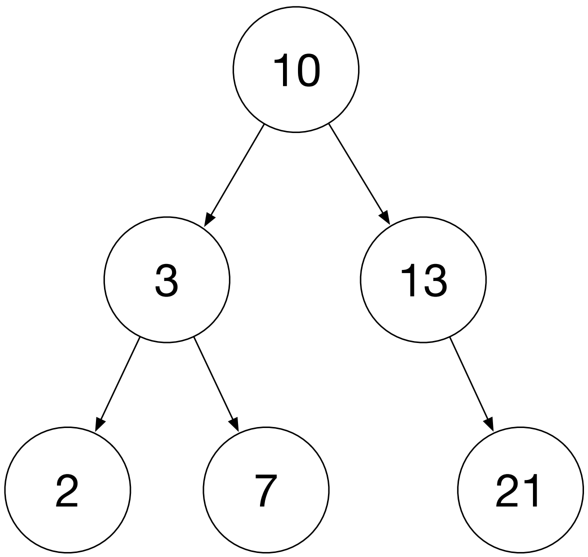





```python
class TreeNode:
  def __init__(self, value, left, right):
    self.value = value
    self.left = left
    self.right = right
```

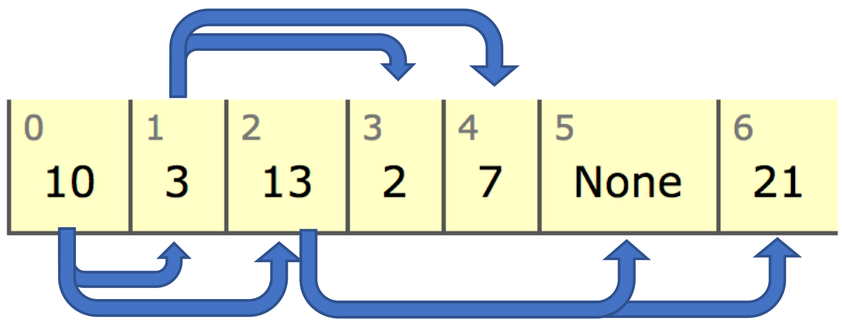
UNIVERSITY OF SAN FRANCISCO

Concrete binary tree using contiguous array



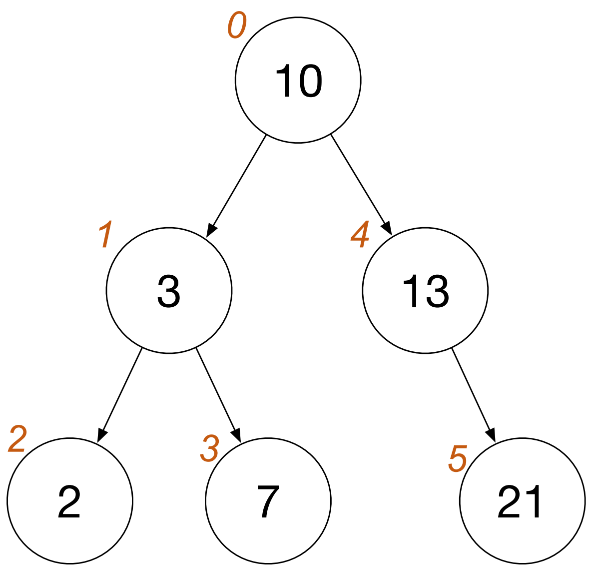
We don't need child pointers!

left child is 2i+1
right child is 2i+2



Indexes as pointers

- sklearn doesn't use nodes with pointers
- Uses node ID's and parallel arrays like *left*, *right*, *value*



left[0] = 1	right[0] = 4	value[0] = 10
left[1] = 2	right[1] = 3	value[1] = 3
left[2] = -1	right[2] = -1	value[2] = 2
left[3] = -1	right[3] = -1	value[3] = 7
left[4] = -1	right[4] = 5	value[4] = 13
left[5] = -1	right[5] = -1	value[5] = 21

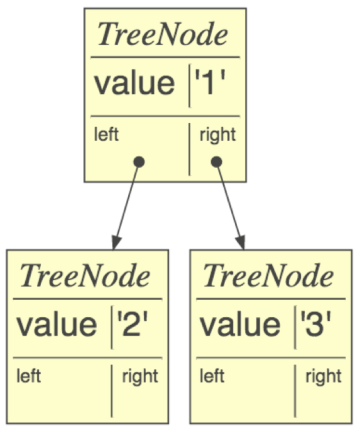
https://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_structure.html

Building binary trees

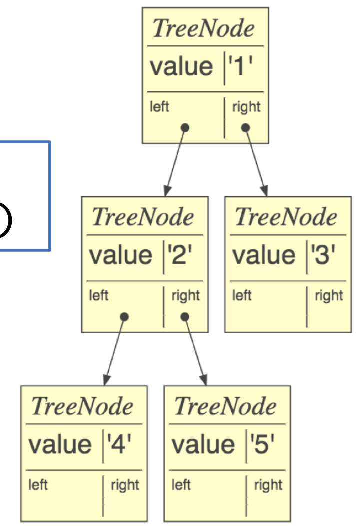
```
class TreeNode:
  def __init__(self, value, left, right):
    self.value = value
    self.left = left
    self.right = right
```

- Manual construction is a simple matter of creating nodes and setting left/right child pointers
- **Exercise**: work through notebook given below

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




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



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

Recursion detour

Math recurrence relations => recursion

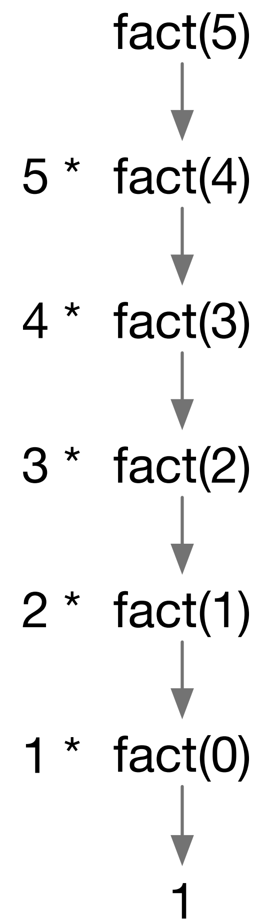
- Factorial definition:
  - Let 0! = 1
  - Define n! = n * (n-1)! for n>=1
- Recurrent math function calls become recursive function call in Python
- Non-recursive version is harder to understand and less natural

```
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 function call is a piece of the result and each call combines subresult(s) to create more complete answer and passes it back



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
- 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:

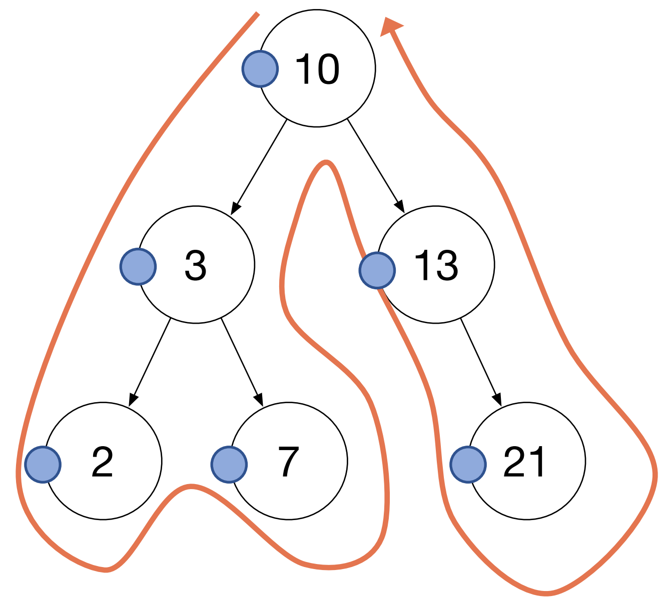
```
def visit(room):
    if item in room: print("rejoice!")
    if room.left exists: visit(room.left)
    if room.right exists: visit(room.right)
```

- This approach is called *backtracking*

Recursive tree walk is natural

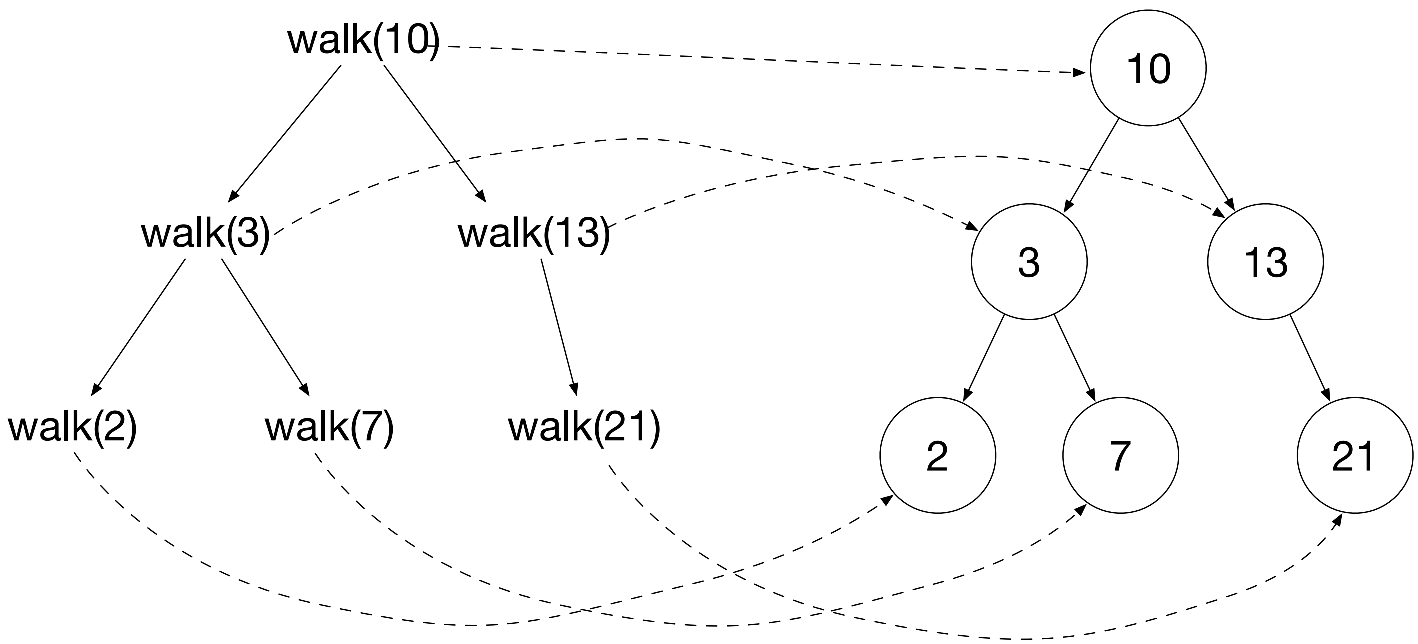
- *Depth-first search* is how we walk (visit) through nodes
- *Pre-order traversal*: executing an action at discovery time, before visiting kids

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



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

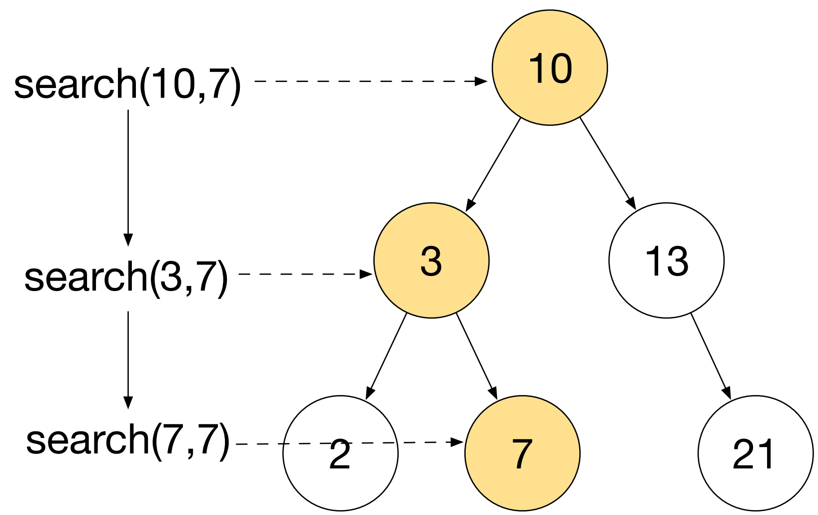
```
def walk(p:TreeNode):
    if p is None: return
    walk(p.left)
    walk(p.right)
```

```
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
```

Now restrict to *binary search tree* structure

- *BST* does a restricted walk using node values
- *binary search tree*: all elements < current node live in left subtree & all elements > current node live in right subtree

```
def search(p:TreeNode, x:object):
    if p is None: return None
    if x<p.value:
        return search(p.left, x)
    if x>p.value:
        return search(p.right, x)
    return p
```



Compare BST search to tree walk

- Conditional recursion; we only recurse to ONE child not both

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

O(n) complexity

```
def search(p:TreeNode, x:object):
    if p is None: return None
    if x<p.value:
        return search(p.left, x)
    if x>p.value:
        return search(p.right, x)
    return p
```

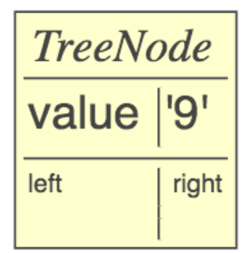
O(log(n)) complexity

Constructing binary search trees

- Result of add() function is the modified tree

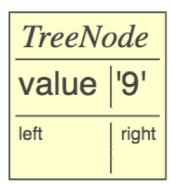
```
def add(p:TreeNode, value) -> TreeNode :
    if p is None:          return TreeNode(value)
    if value < p.value:    p.left = add(p.left, value)
    elif value > p.value: p.right = add(p.right, value)
    return p # do nothing if equal (already there)
```

- Initial condition: add to None: `root = add(None, 9)`



- If node.value==value, return that node:

`root = add(root, 9)`



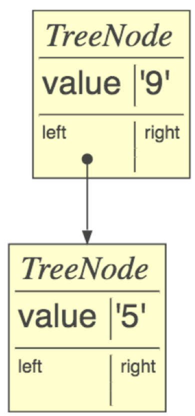
Constructing binary search trees cont'd

- If value < current node, add to the left

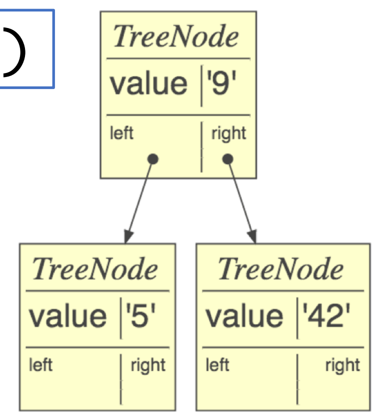
```
if value < p.value: p.left = add(p.left, value)
```

- If value > current node, add to the right

```
root = add(root, 5)
```




```
root = add(root, 42)
```



Consider similarity of search / build

```
def search(p:TreeNode, x:object):
    if p is None: return None
    if x < p.value:
        return search(p.left, x)
    if x > p.value:
        return search(p.right, x)
    return p
```

```
def add(p:TreeNode, v):
    if p is None: return TreeNode(v)
    if v < p.value:
        p.left = add(p.left, v)
    elif v > p.value:
        p.right = add(p.right, v)
    return p
```

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
- Remember the recursive function template
- Depth-first-search visits each node through backtracking
- Binary search tree constrains node values