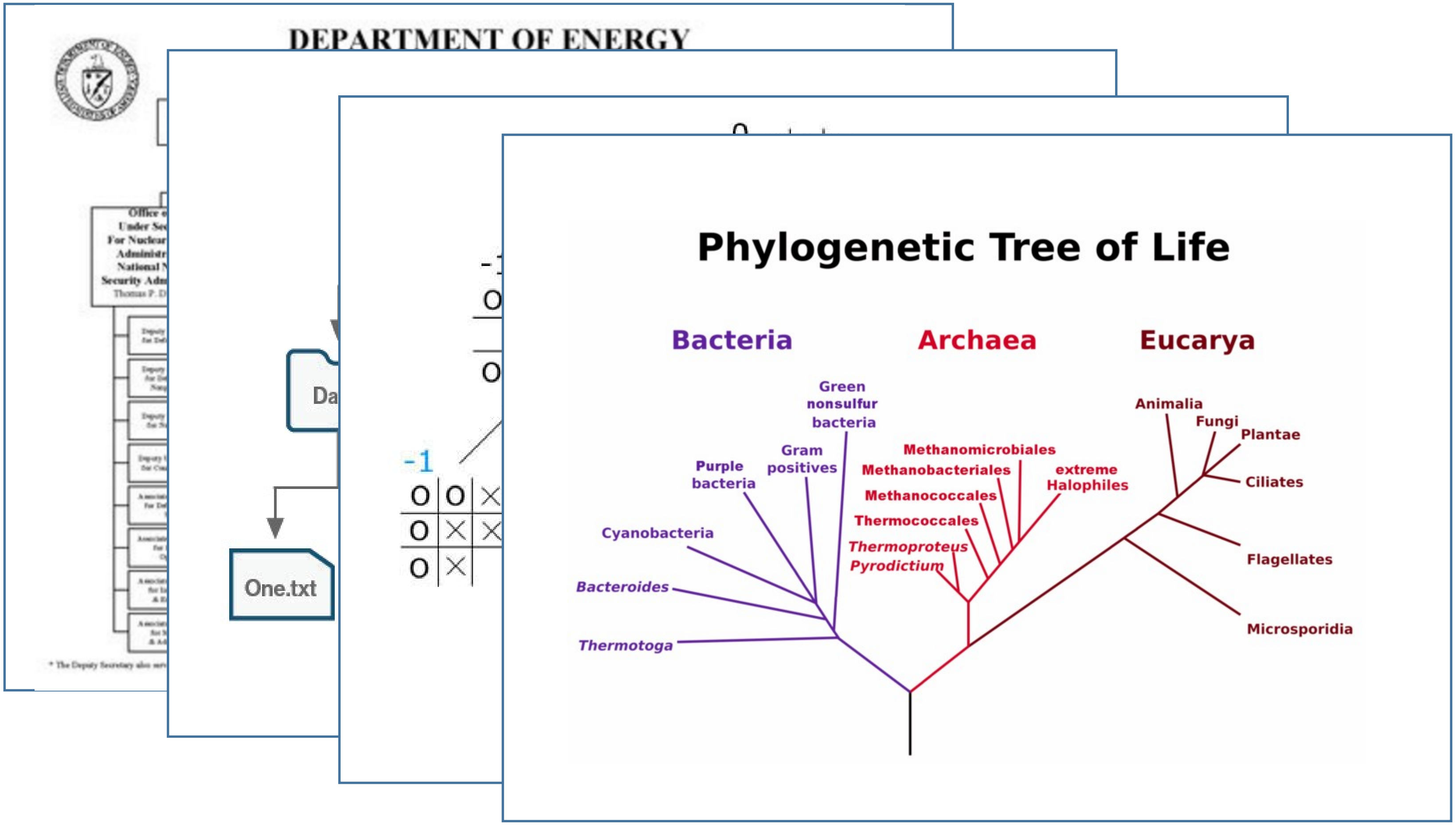
CSc 120
Introduction to Computer Programming II

Trees

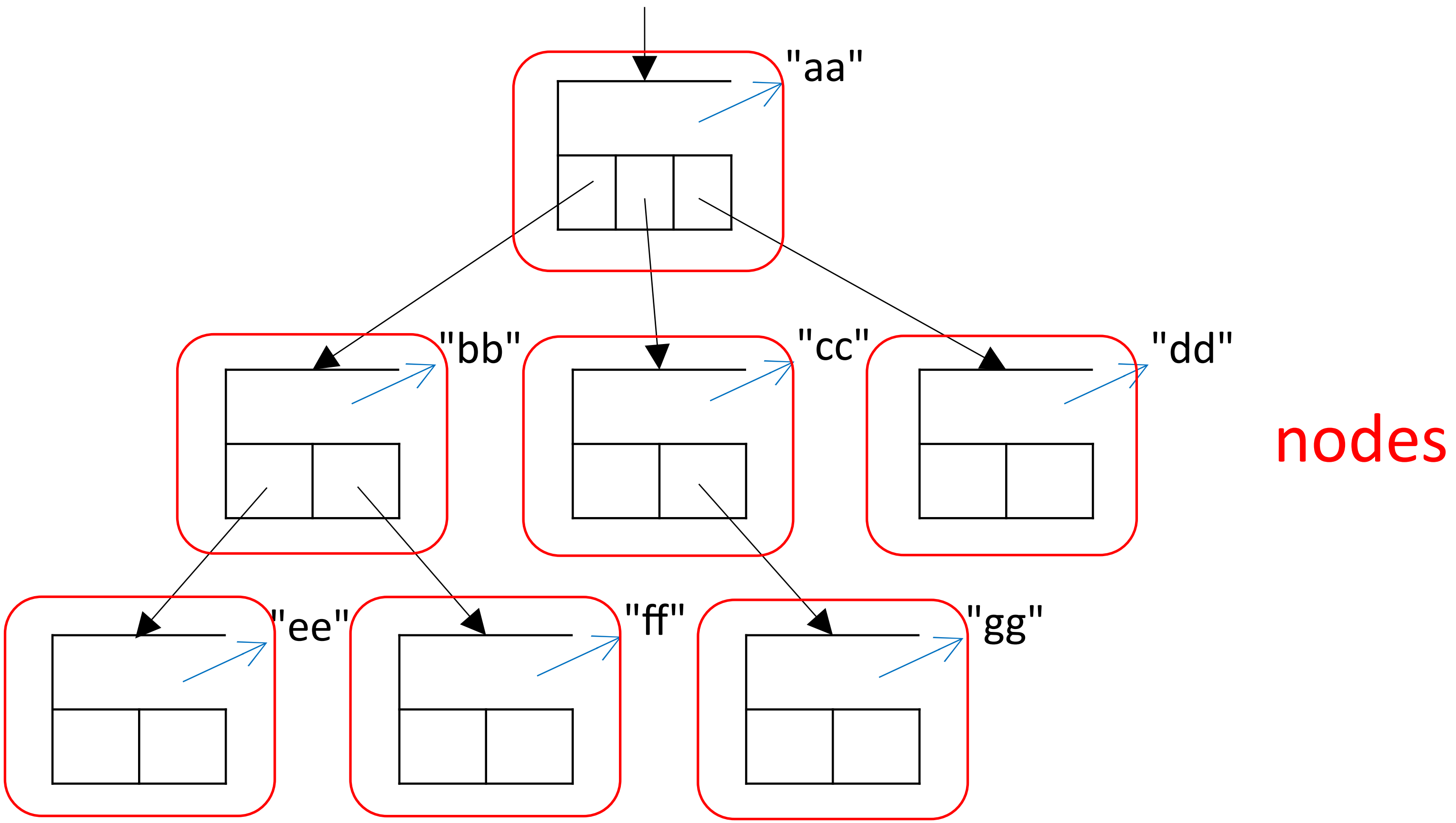
trees: basic concepts

Hierarchies

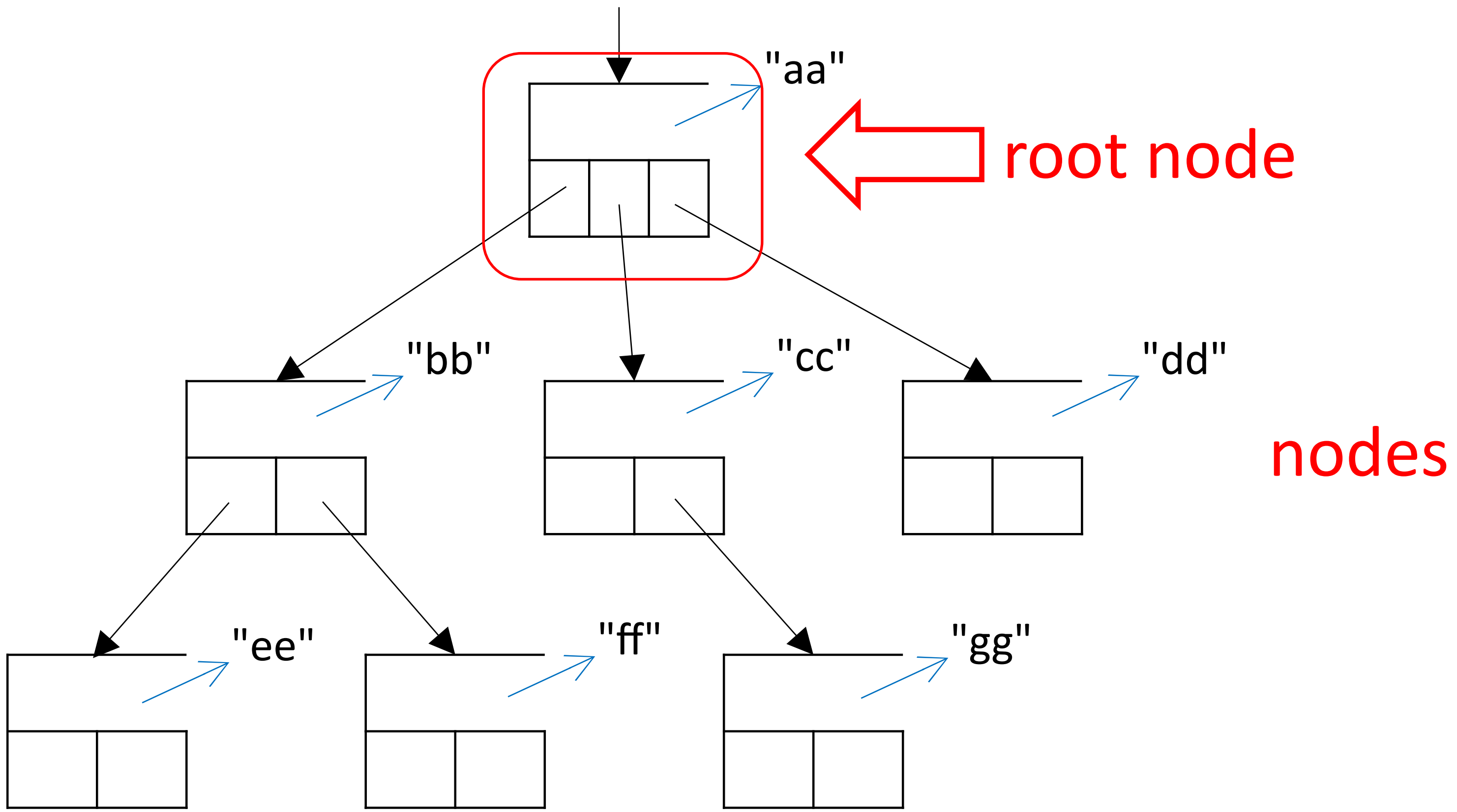
- Hierarchically organized "stuff" are everywhere



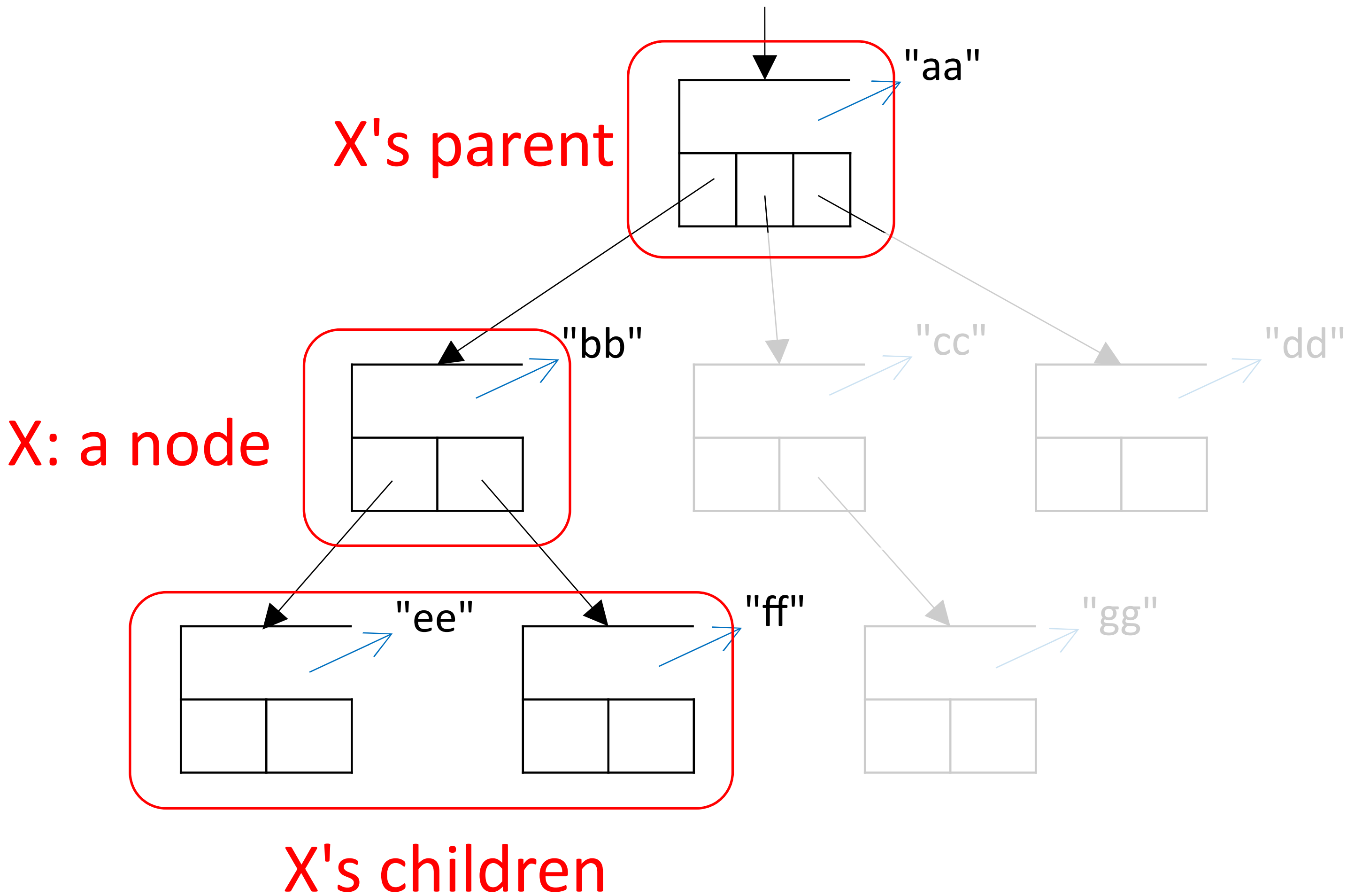
Trees



Trees



Trees



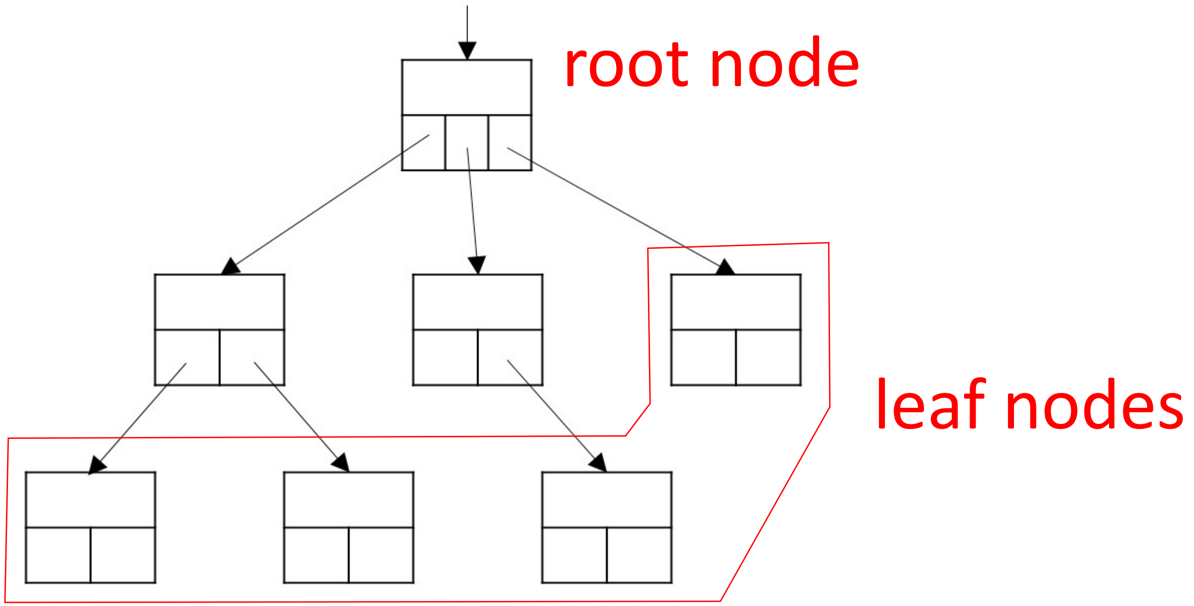
Trees: terminology

- A tree is a collection of nodes
- Each node has:
  - ≥ 0 child nodes
  - 0 or 1 parent nodes

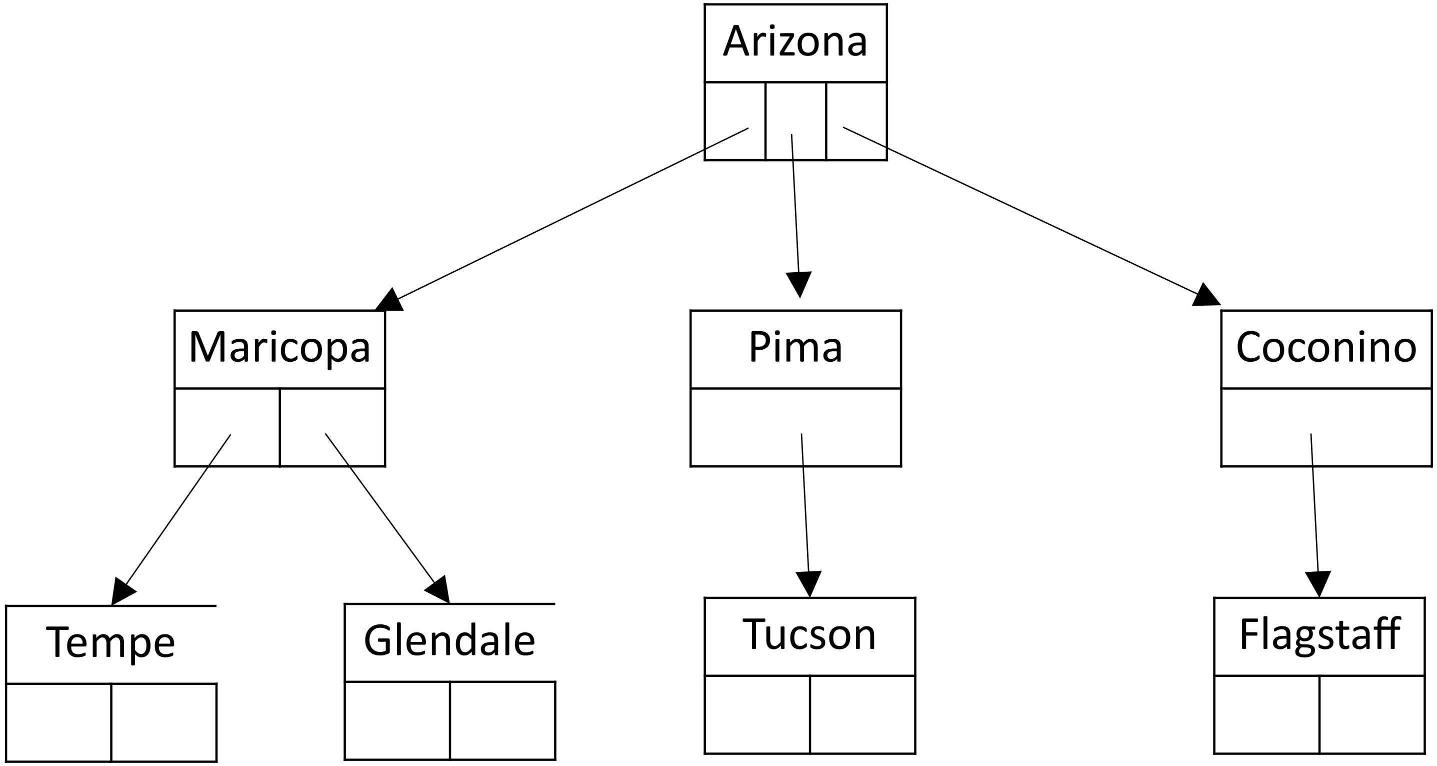
Y is a child of X $\Leftrightarrow$ X is a parent of Y

- A node with 0 children is called a *leaf node*
- A node with 0 parent nodes is called the *root node*
- A tree has:
  - ≥ 1 leaf nodes
  - exactly one root node

Trees: leaves and root

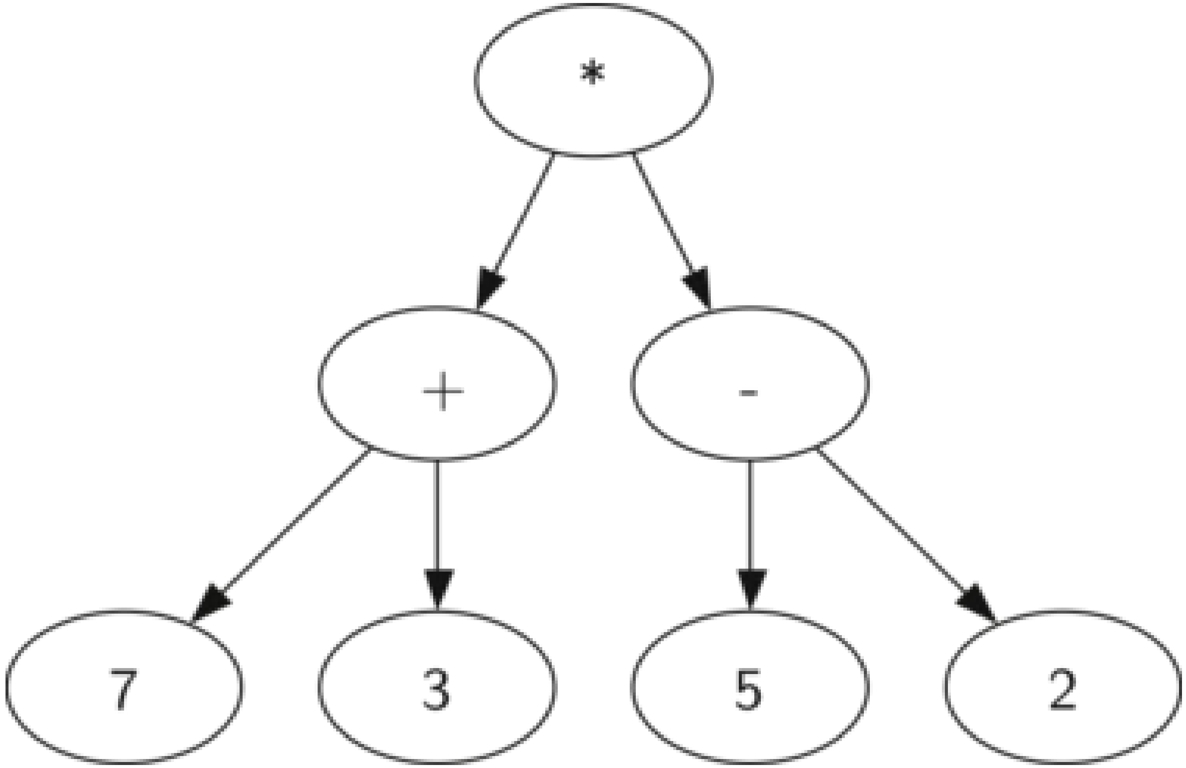


Tree: Example



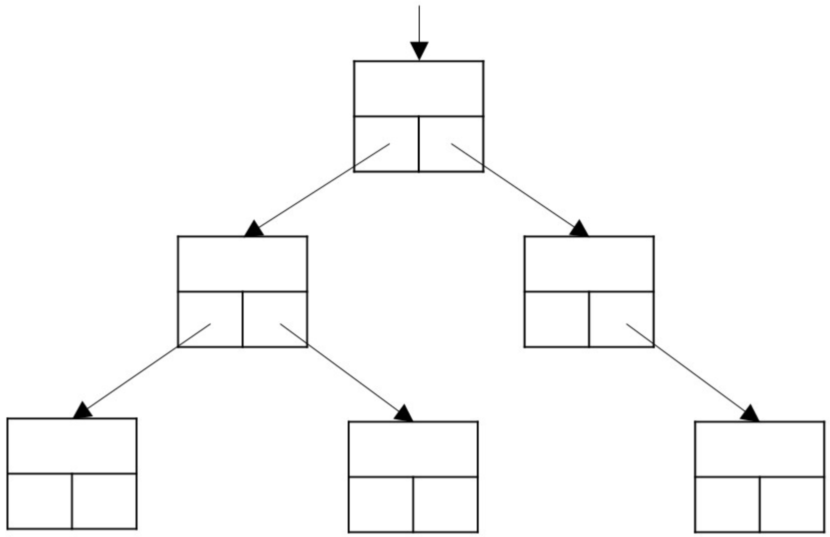
Tree: Example

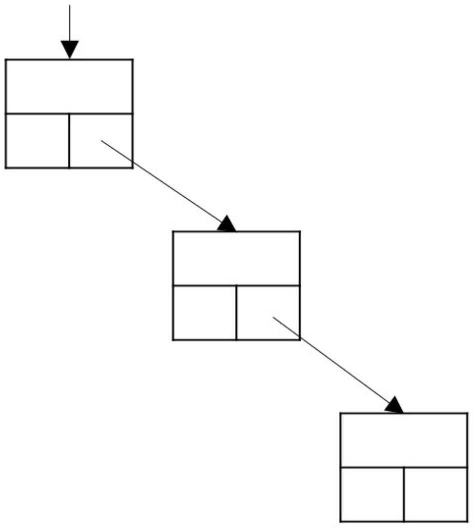
(7+3)*(5-2)



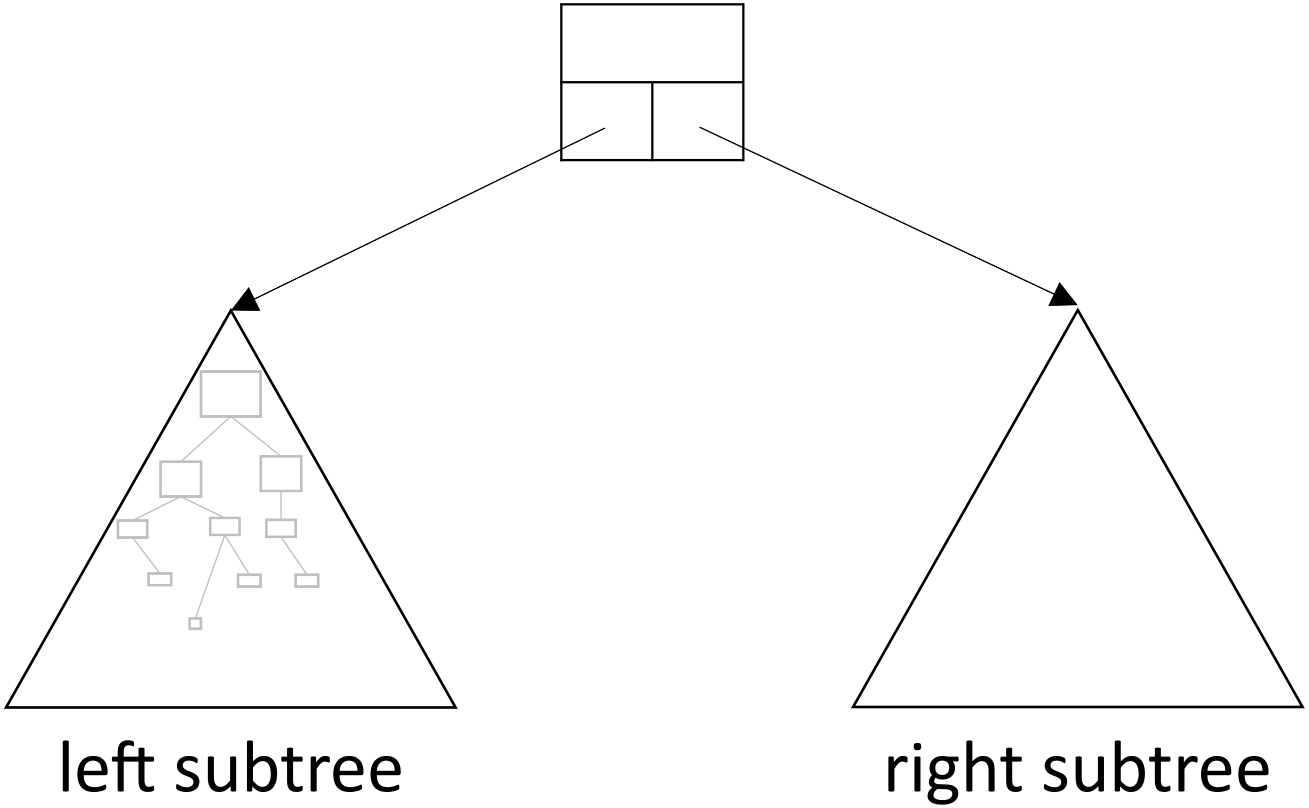
Binary trees

- A tree where each node has at most two children is called a *binary tree*





Binary trees



Trees: node representation

- A node in a general tree:
  - value(s) at the node
  - references to child nodes:
    - an extensible data structure (e.g., a list, a linked list, or dictionary)
  - (infrequently) reference to parent

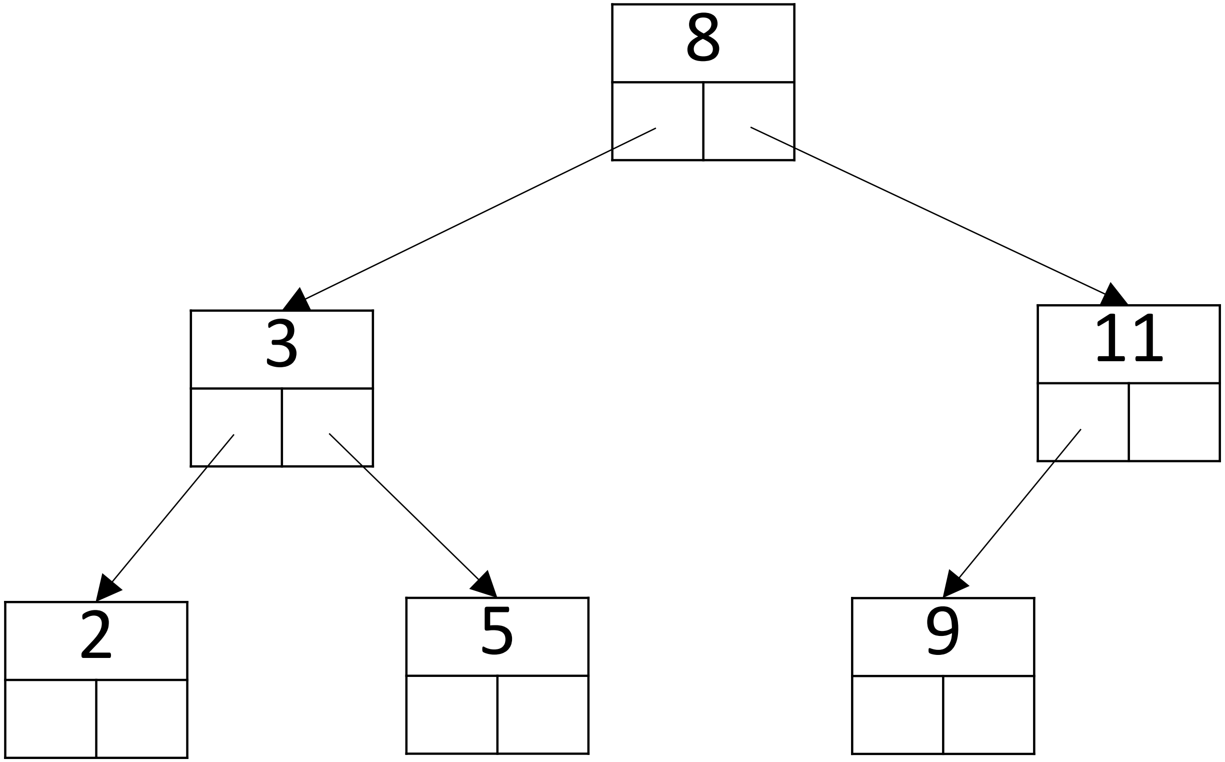
- A node in a binary tree:
  - value(s) at the node
  - a reference to the left subtree
  - a reference to the right subtree
  - (infrequently) reference to parent

Binary trees: node representation

```
class TreeNode:
    def __init__(self, value):
        self.val   = value      # the value at the node
        self.left  = None       # left child
        self.right = None       # right child

    ...
```

binary search trees

Examine this binary tree:



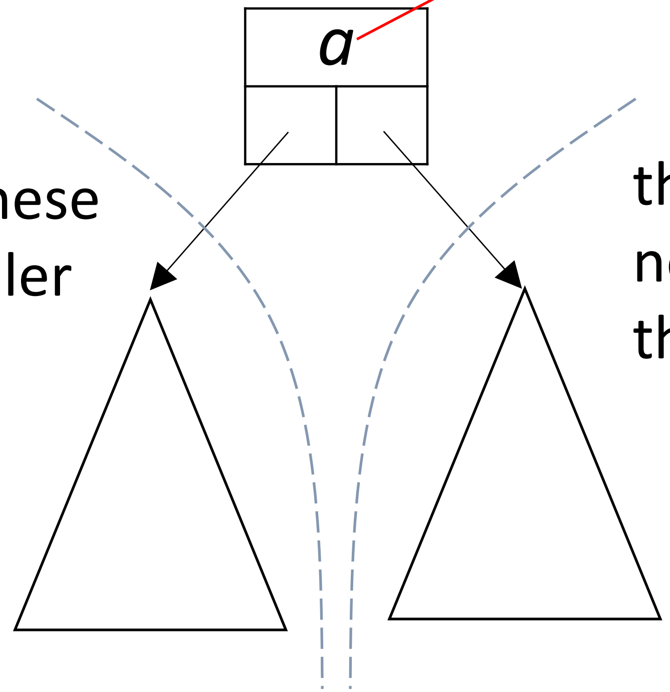
What can we say about the values in the nodes to the left of 8?

What can we say about the values in the nodes to the right of 8?

Binary search tree (BST)

A *binary search tree* is a binary tree where every node satisfies the following:

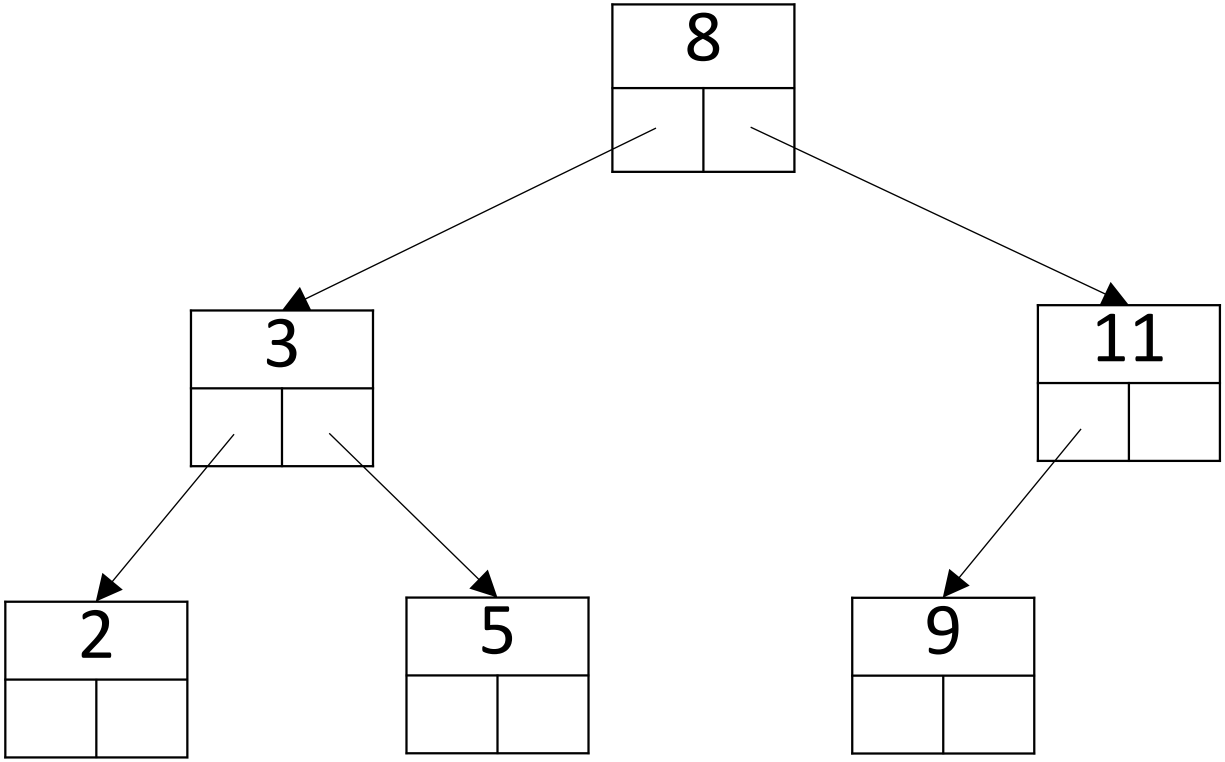
value at this node



the values at these nodes are smaller than *a*

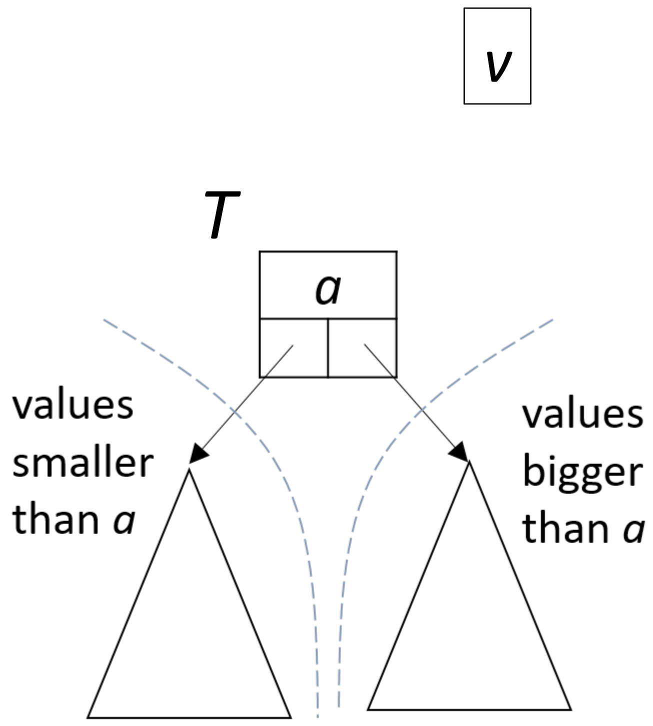
the values at these nodes are bigger than *a*

Binary search tree: Example



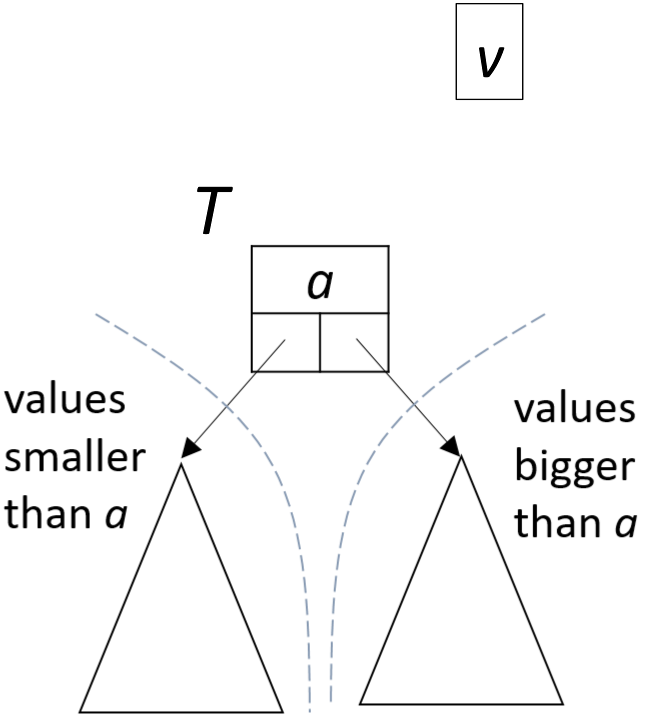
Searching a BST

Given a BST *T* and a value *v*, is there a node in *T* with value *v*?



Searching a BST

Given a BST *T* and a value *v*, is there a node in *T* with value *v*?

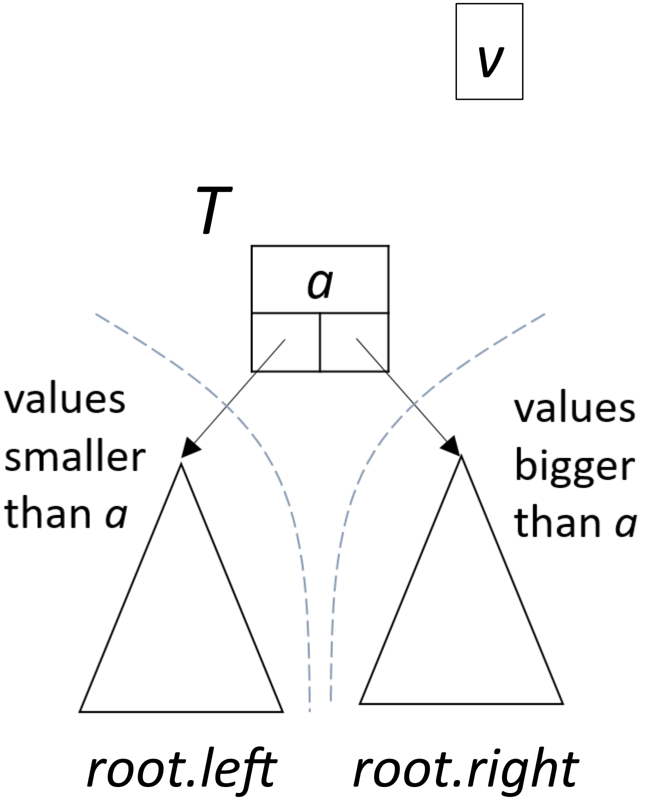


Idea: at each node with value *a*:

- if *a* == *v*: done
- if v < *a*: search left subtree
- if v > *a*: search right subtree

Searching a BST

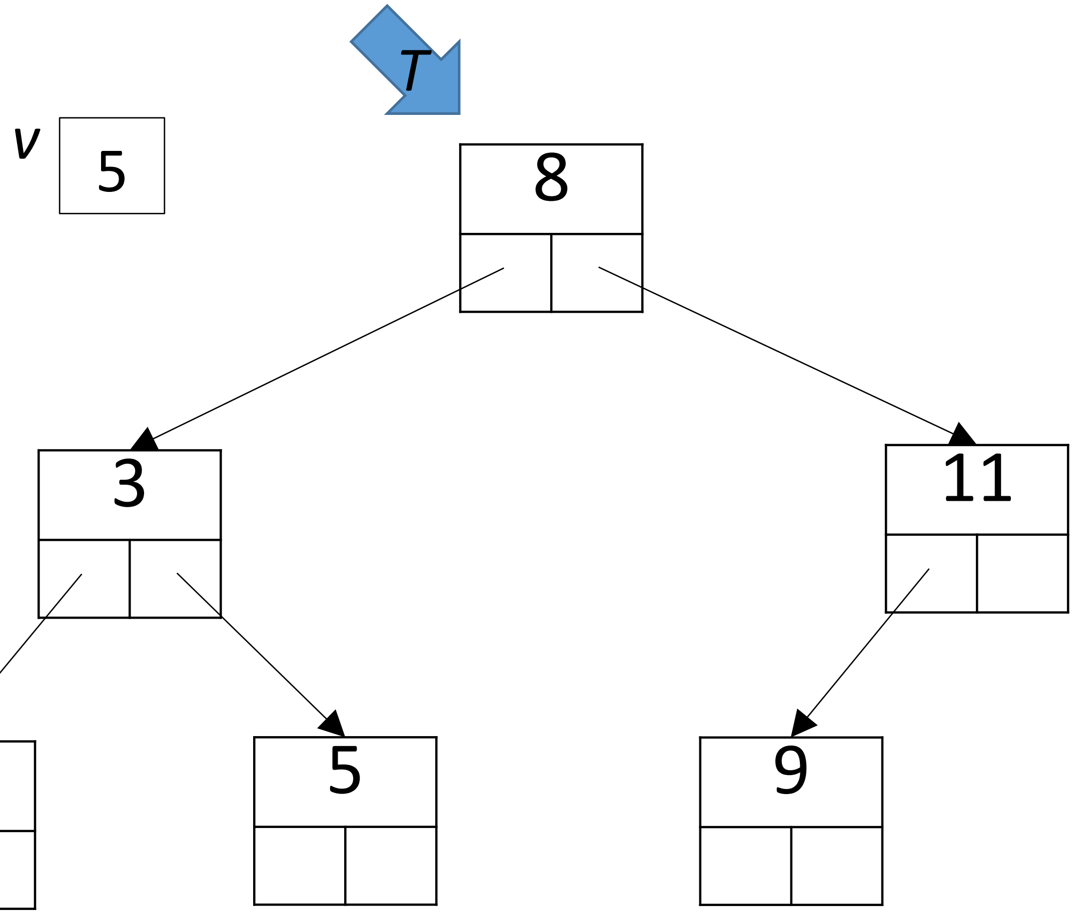
```
def search(T, v):
    if T == None:
        return False
    if v == T.val:
        return True
    if v  < T.val:
        return search(T.left, v)
    else:
        return search(T.right, v)
```



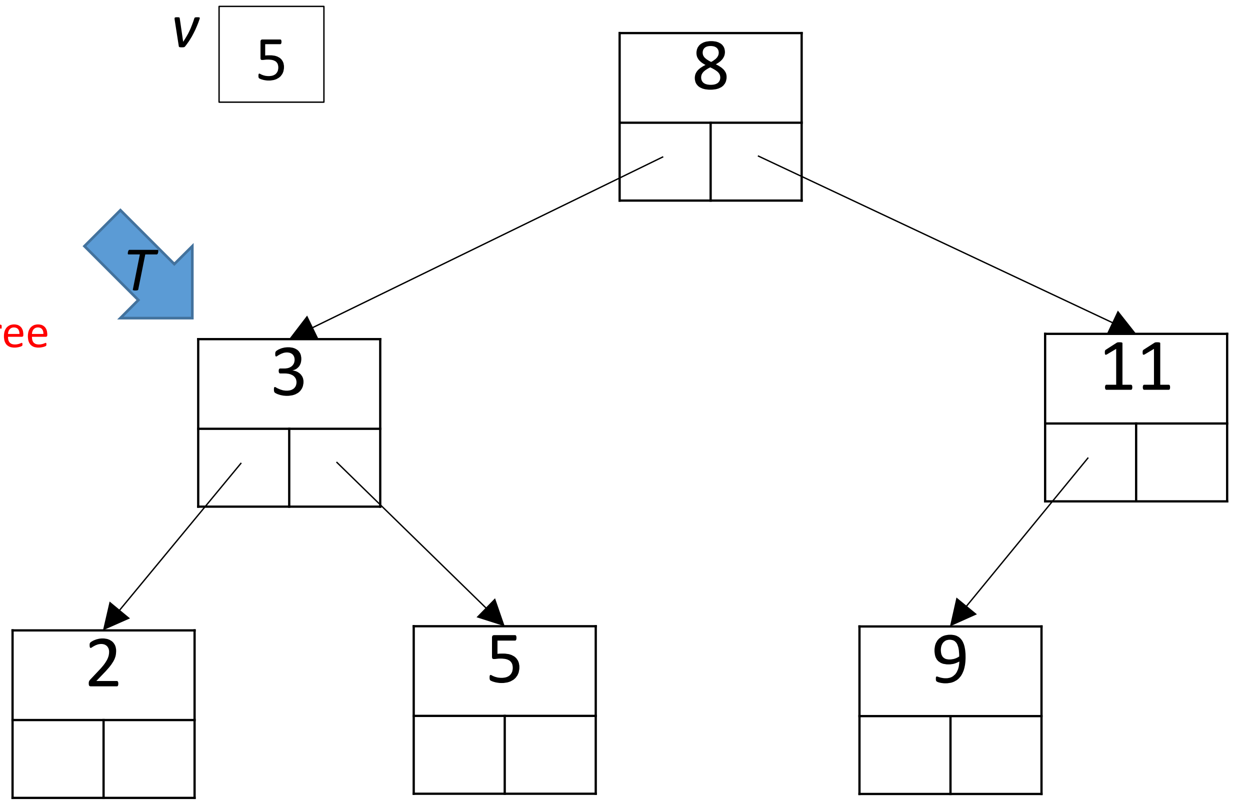
Searching a BST: Example 1

v < T.val:

⇒ search left subtree



Searching a BST: Example 1

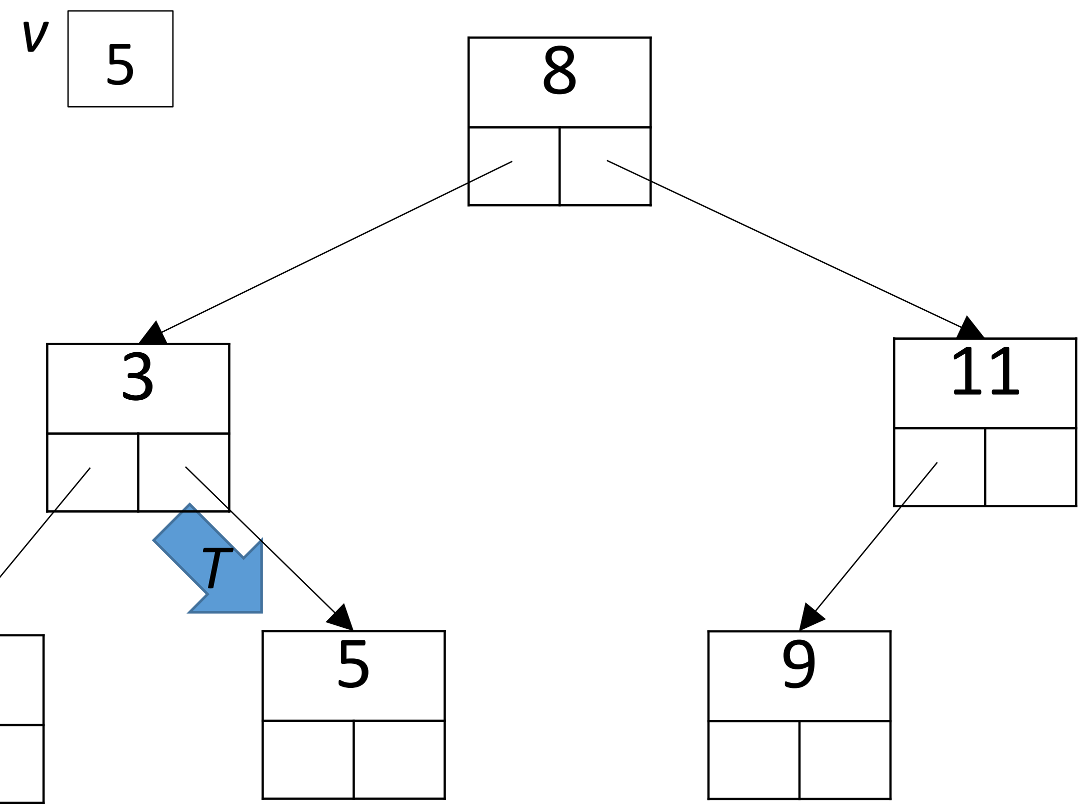


v > T.*val*:
 ⇒ search right subtree

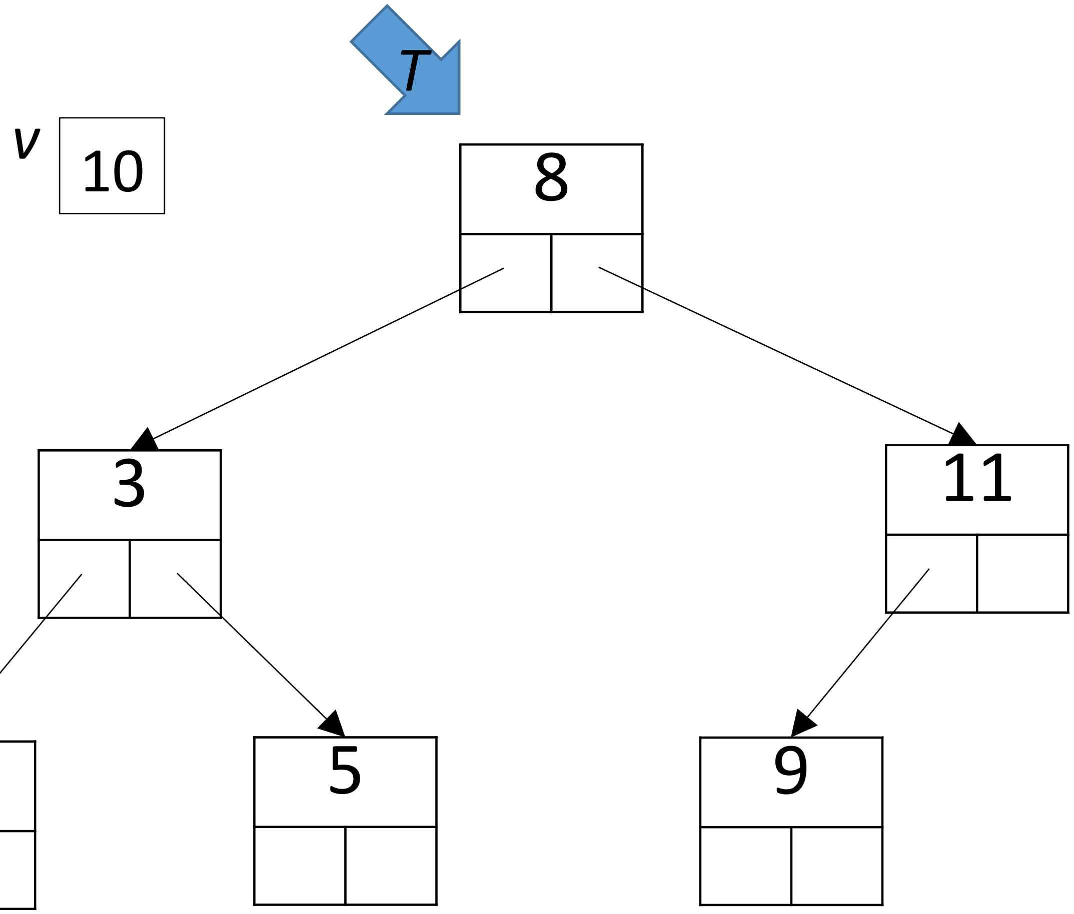
Searching a BST: Example 1

v == T.*val*:

$\Rightarrow$ return True



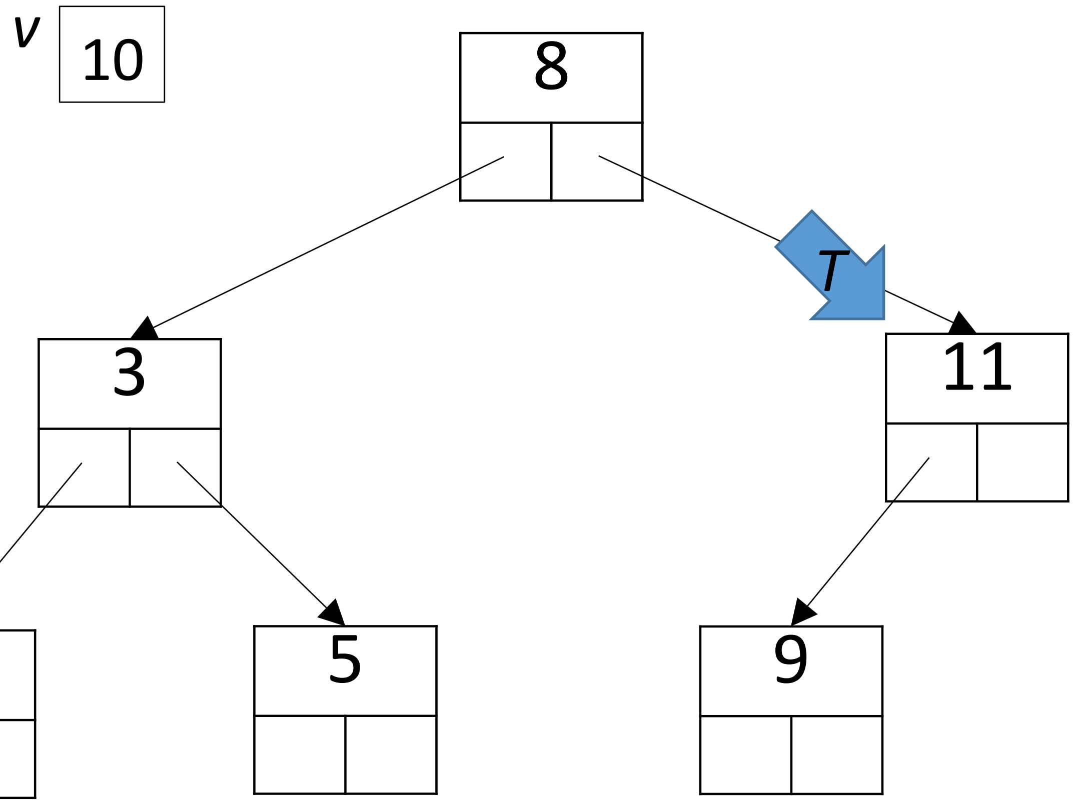
Searching a BST: Example 2



v > *T.val*:

⇒ search right subtree

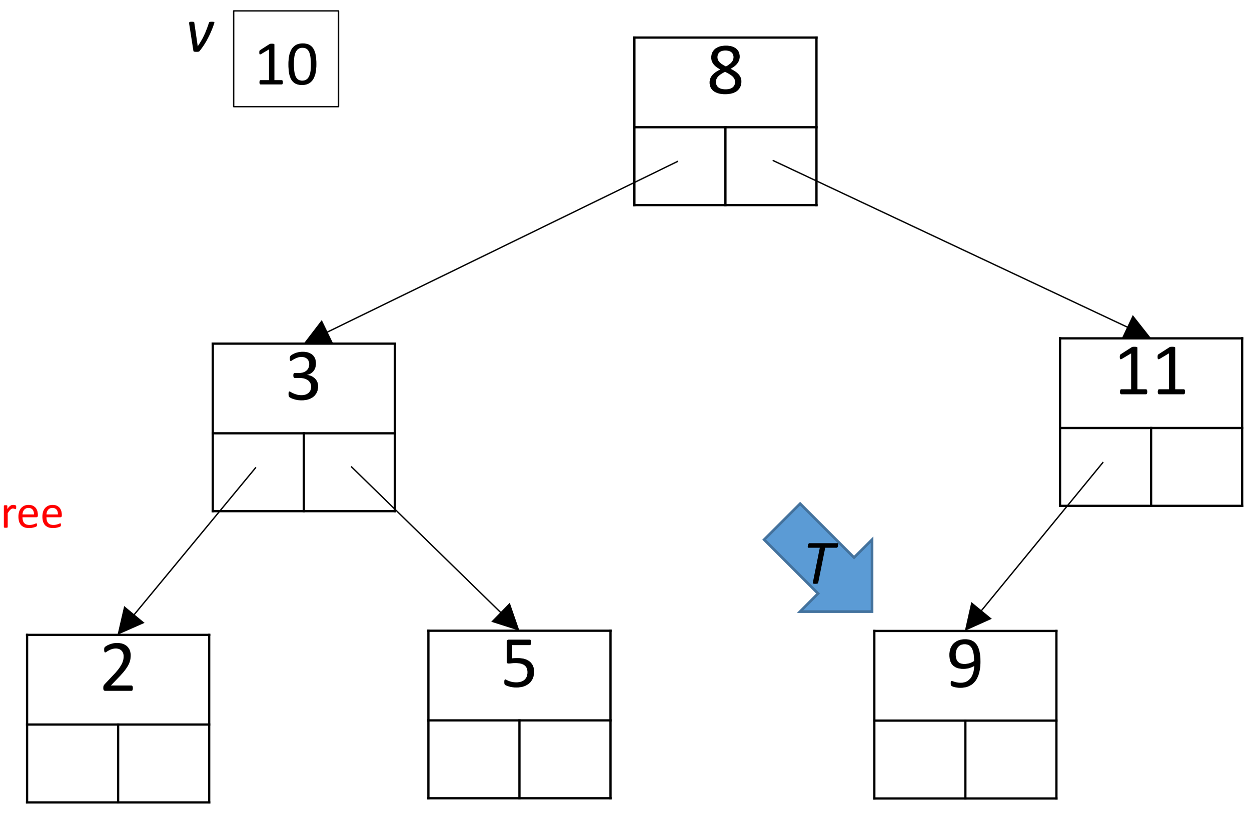
Searching a BST: Example 2



v < *T.val*:

⇒ search left subtree

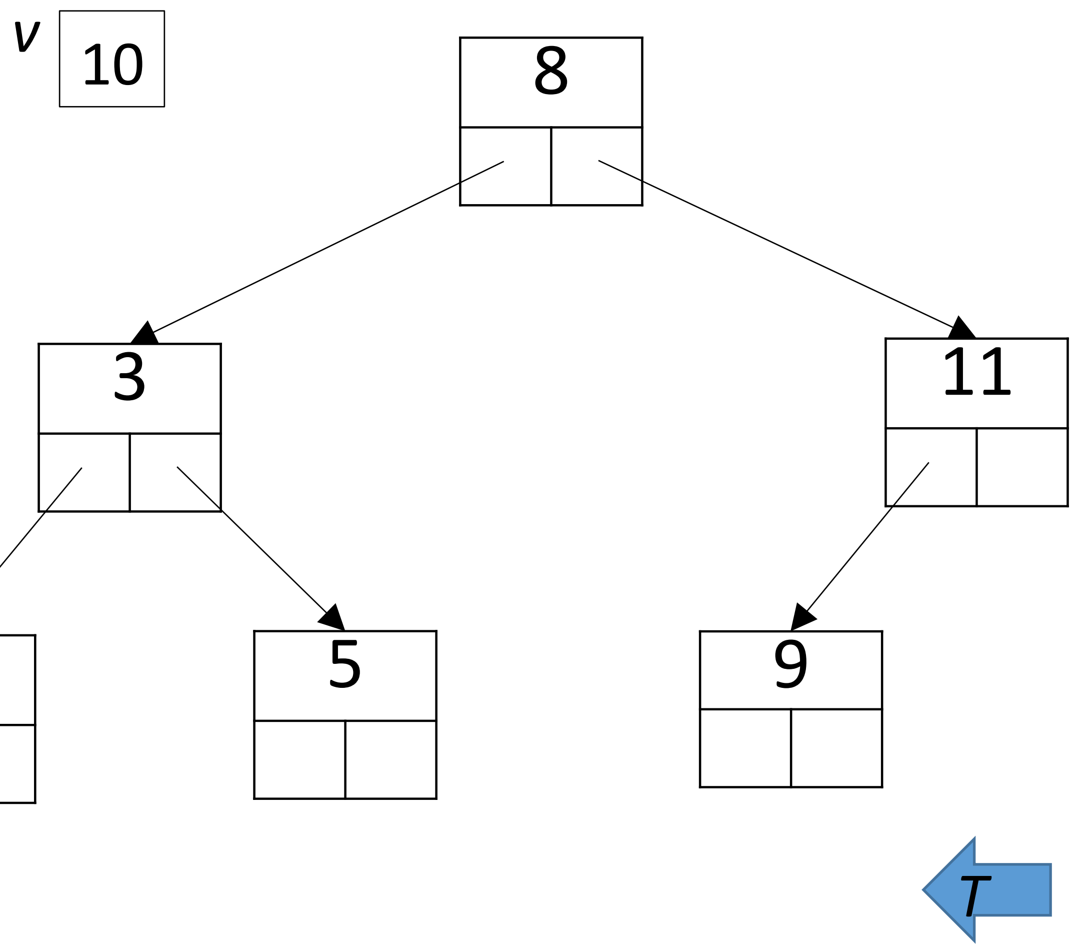
Searching a BST: Example 2



v > *T.val*:

⇒ search right subtree

Searching a BST: Example 2



T == None:

$\Rightarrow$ return False

Constructing a BST

Given a BST *T* and a value *v*, return the tree *T′* obtained by inserting *v* into *T*

- if *T* is empty: return a node with value *v*
- otherwise:
  - if v < T.*val* : insert into *T*'s left subtree
  - if v > T.*val* : insert into *T*'s right subtree

Constructing a BST

```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```

Constructing a BST: Example

Sequence of values: $\boxed{8}$ 3 11 2 9 5

```
def insert(T, v):    (v = 8, T = None)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```

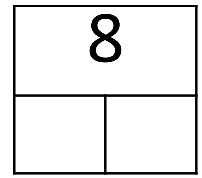
T: None

Constructing a BST: Example

Sequence of values: $\boxed{8}$ 3 11 2 9 5

```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```

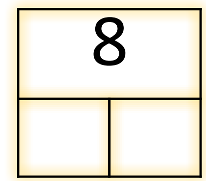
T:



Constructing a BST: Example

Sequence of values: 8 $\boxed{3}$ 11 2 9 5

```
def insert(T, v): (v = 3, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```

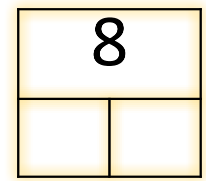
T: 

Constructing a BST: Example

Sequence of values: 8 $\boxed{3}$ 11 2 9 5

```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```

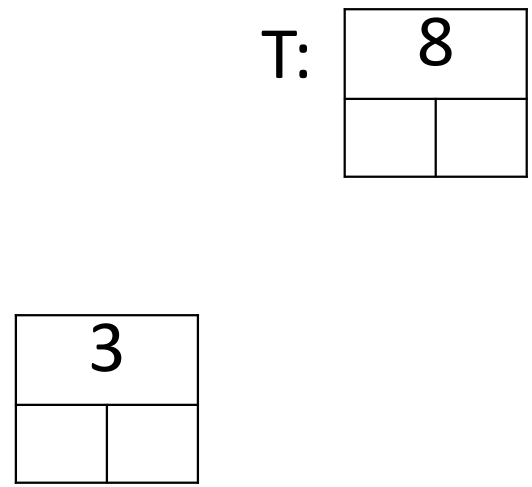
T:



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

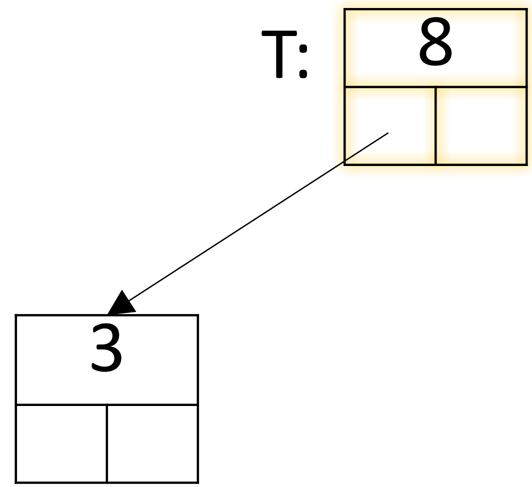
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 $\boxed{3}$ 11 2 9 5

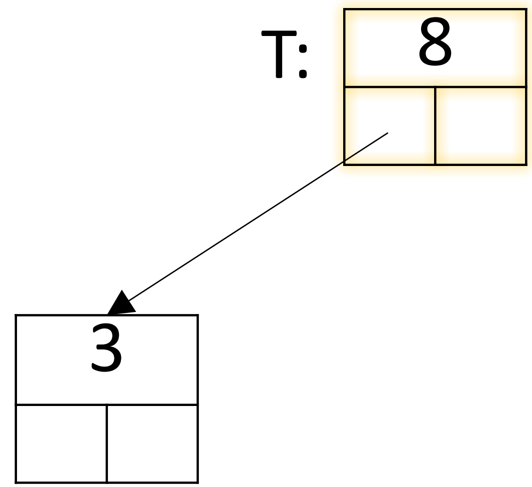
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)
    return T
```



Constructing a BST: Example

Sequence of values: 8 3 $\boxed{11}$ 2 9 5

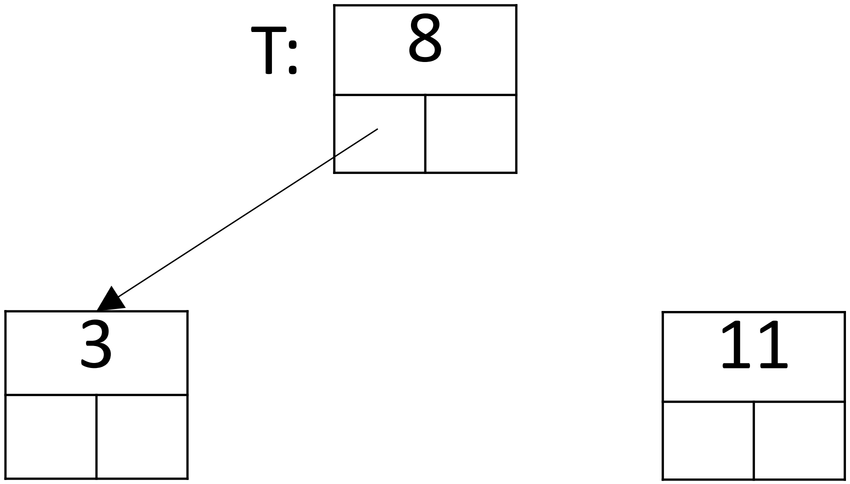
```
def insert(T, v):(v = 11, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

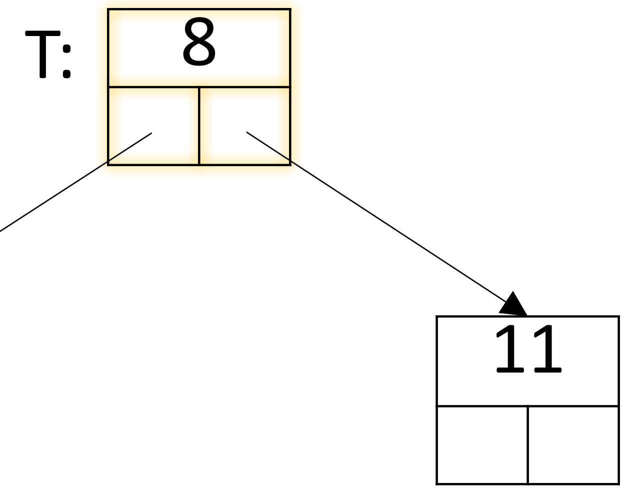
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

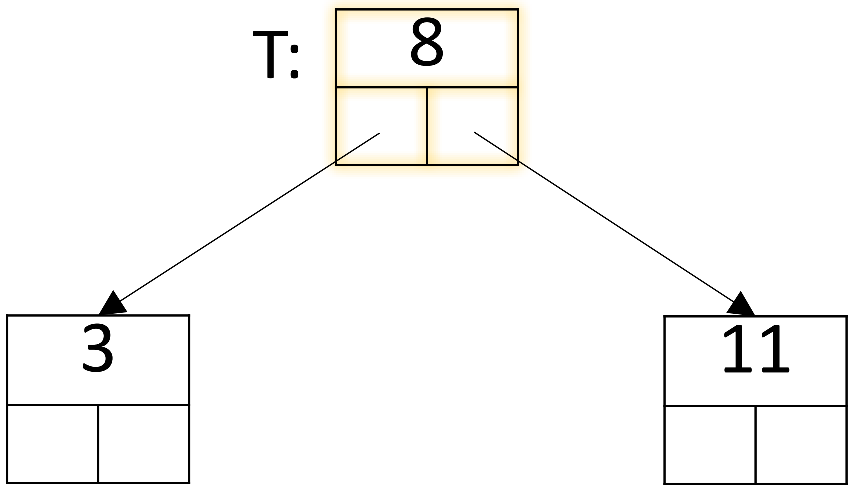
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)
    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 $\boxed{2}$ 9 5

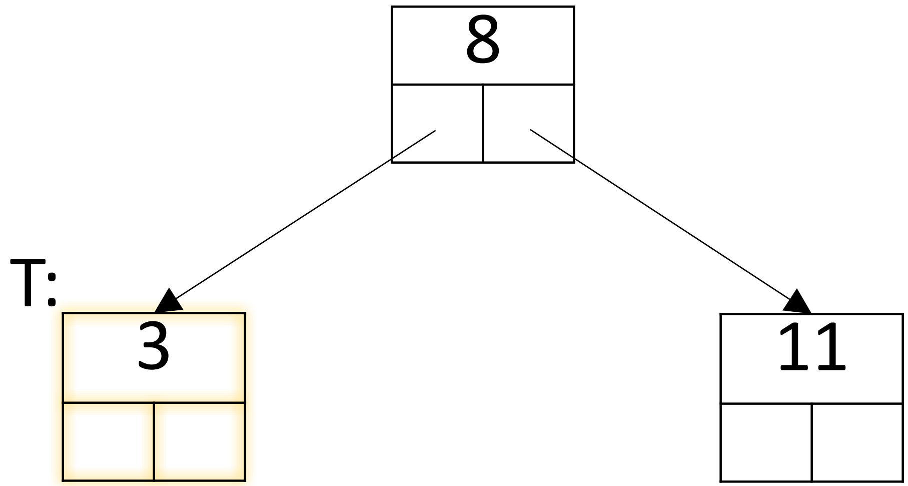
```
def insert(T, v):(v = 2, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)
    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 $\boxed{2}$ 9 5

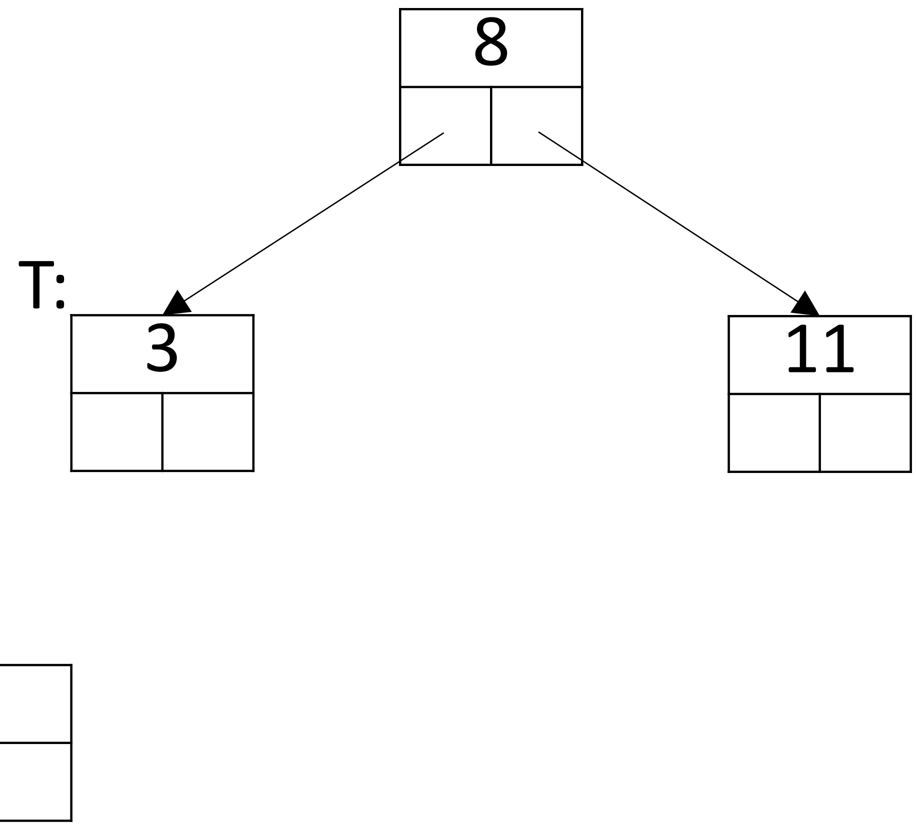
```
def insert(T, v):(v = 2, T.value = 3)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

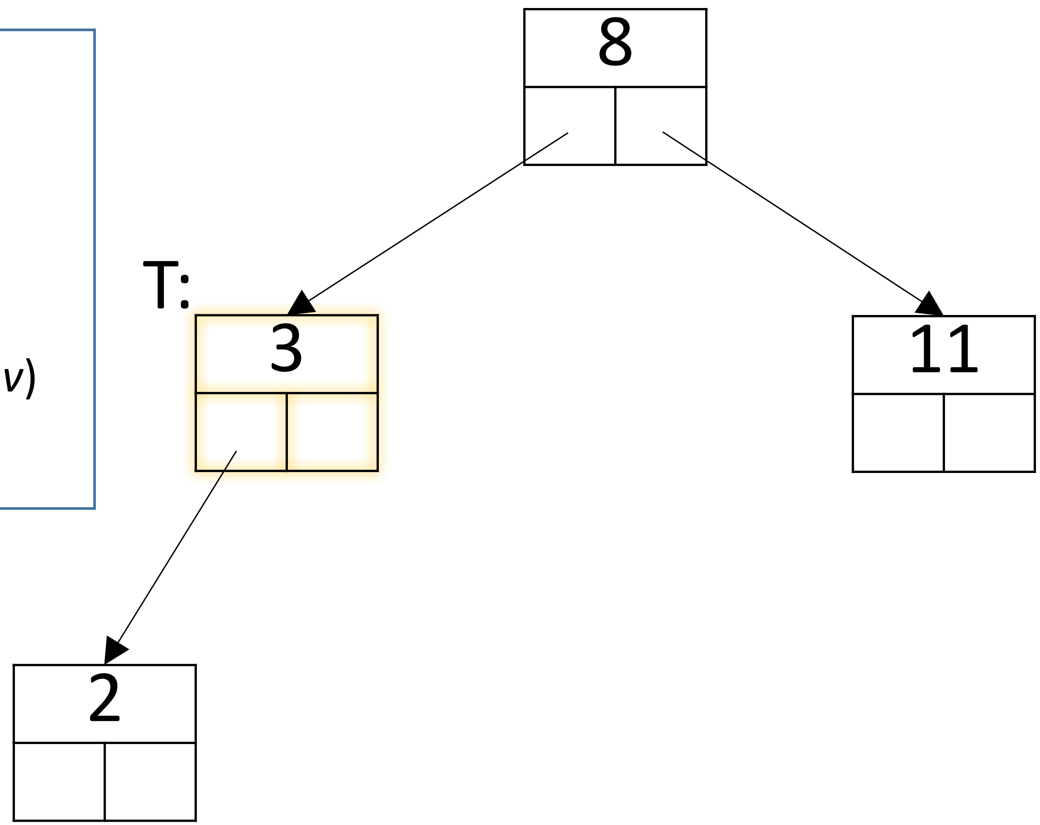
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 $\boxed{2}$ 9 5

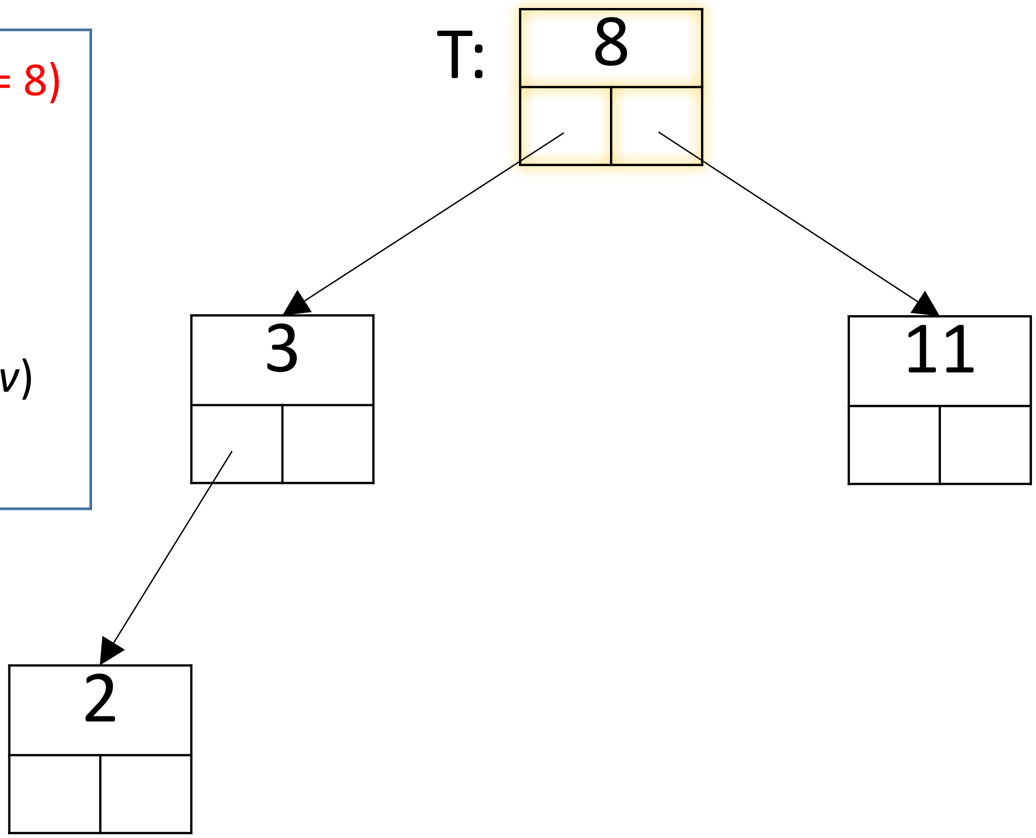
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 $\boxed{9}$ 5

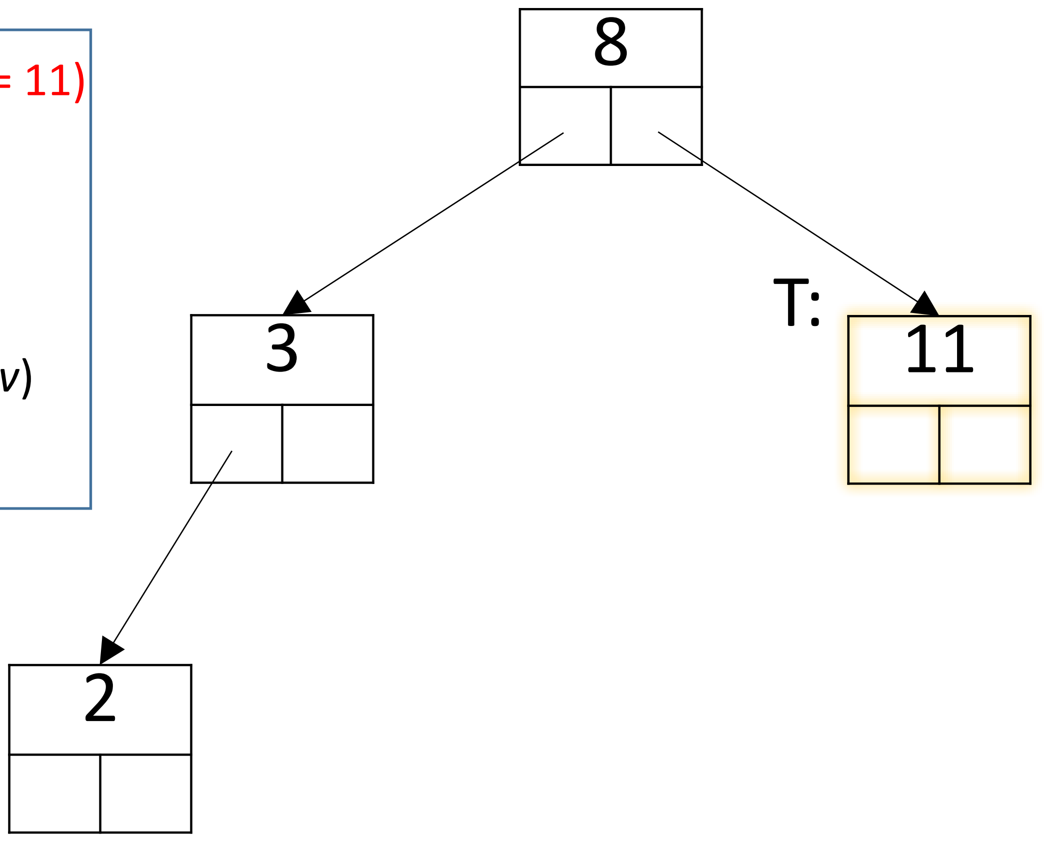
```
def insert(T, v):(v = 9, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 $\boxed{9}$ 5

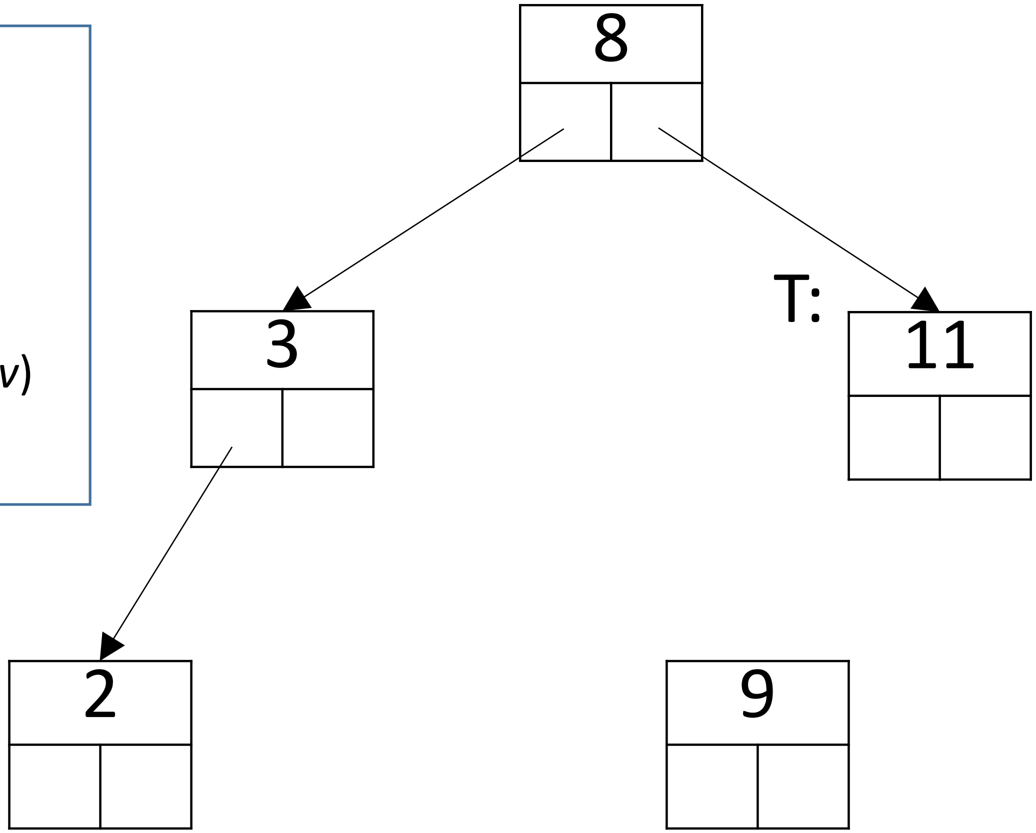
```
def insert(T, v):(v = 9, T.value = 11)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)
    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

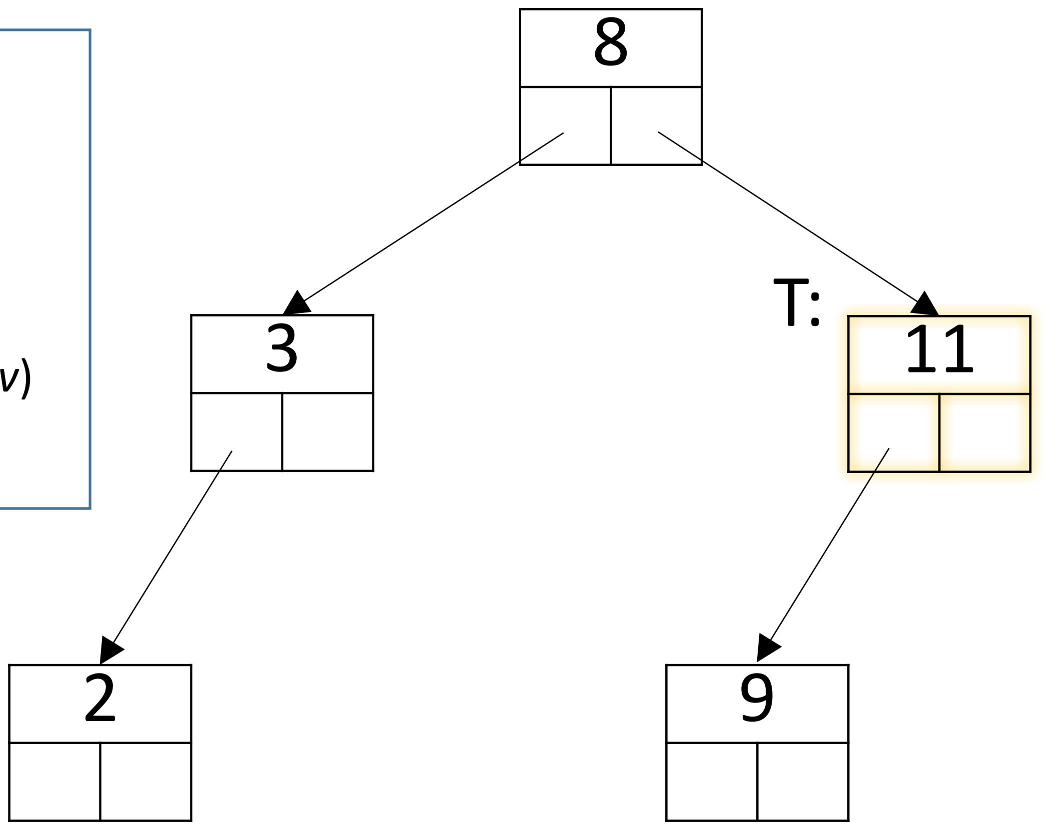
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 $\boxed{9}$ 5

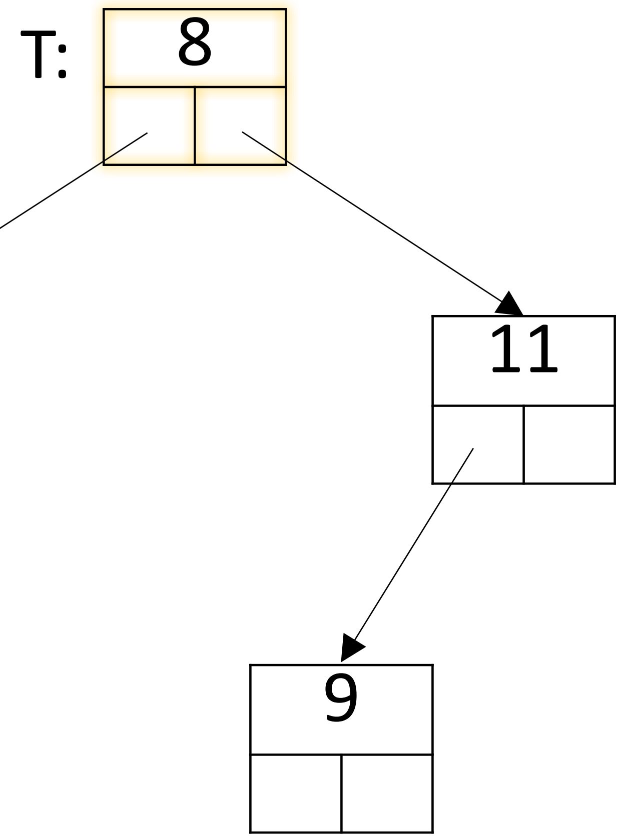
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

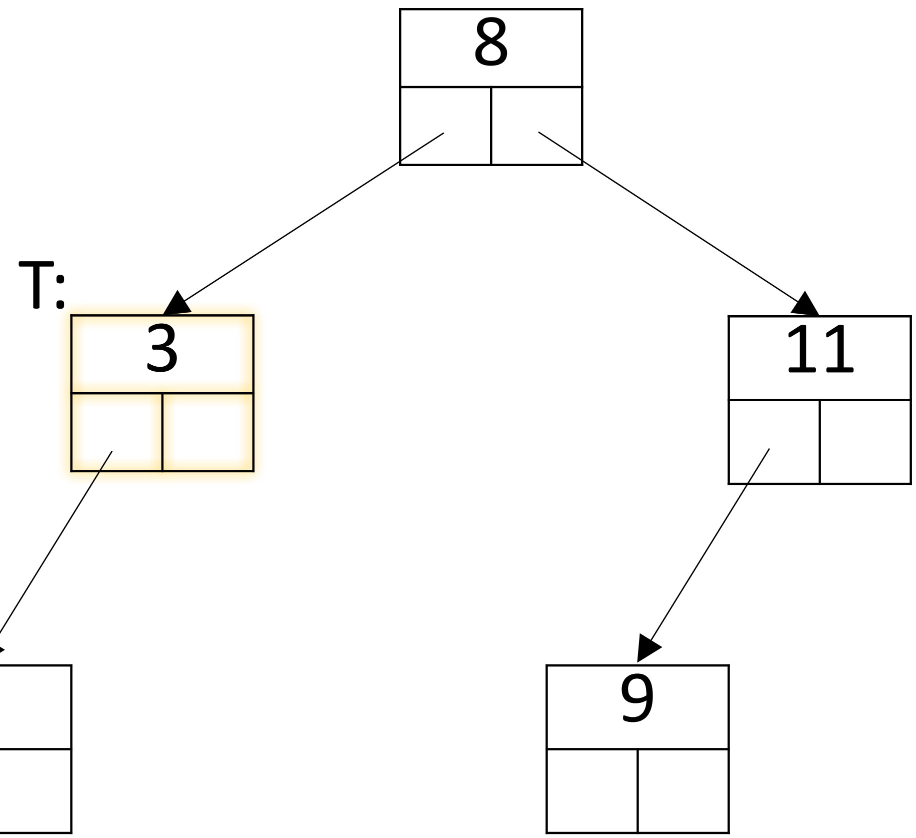
```
def insert(T, v): (v = 5, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 $\boxed{5}$

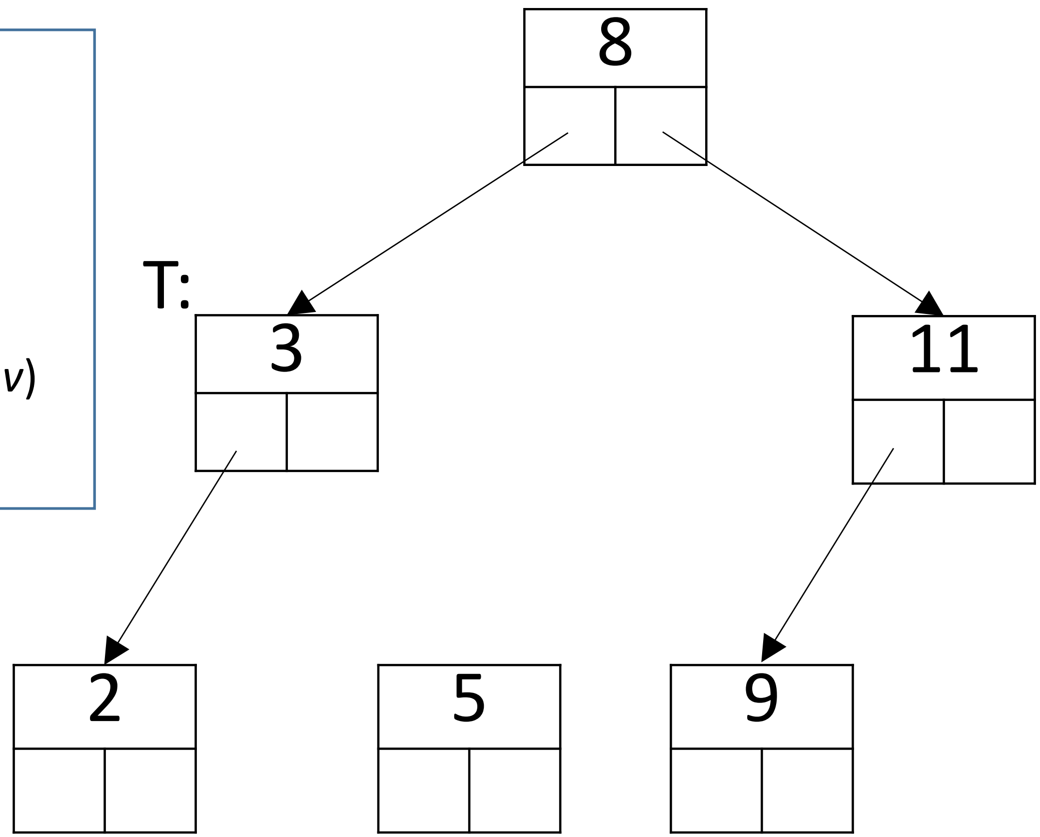
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 $\boxed{5}$

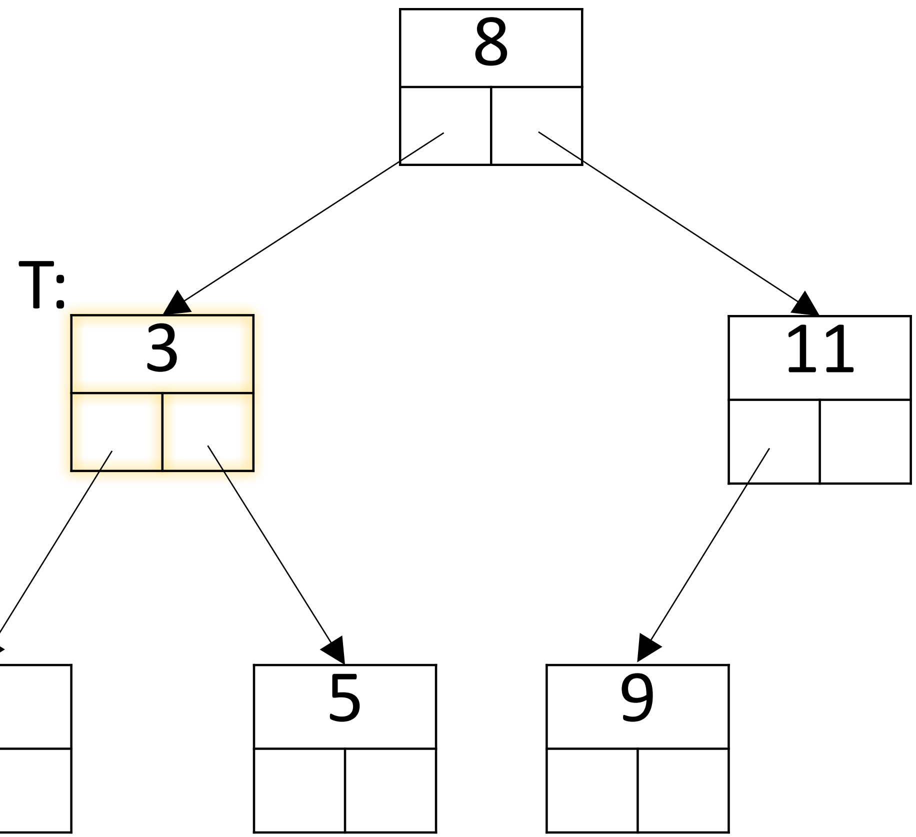
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

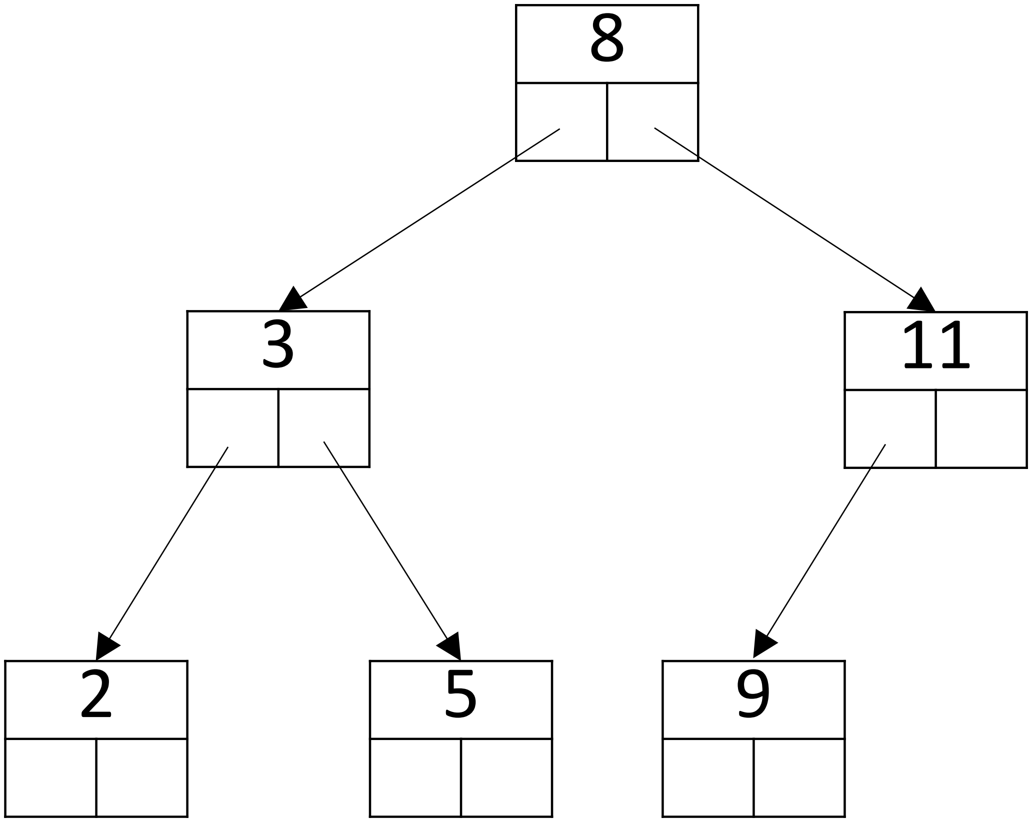
Sequence of values: 8 3 11 2 9 $\boxed{5}$

```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)
    return T
```

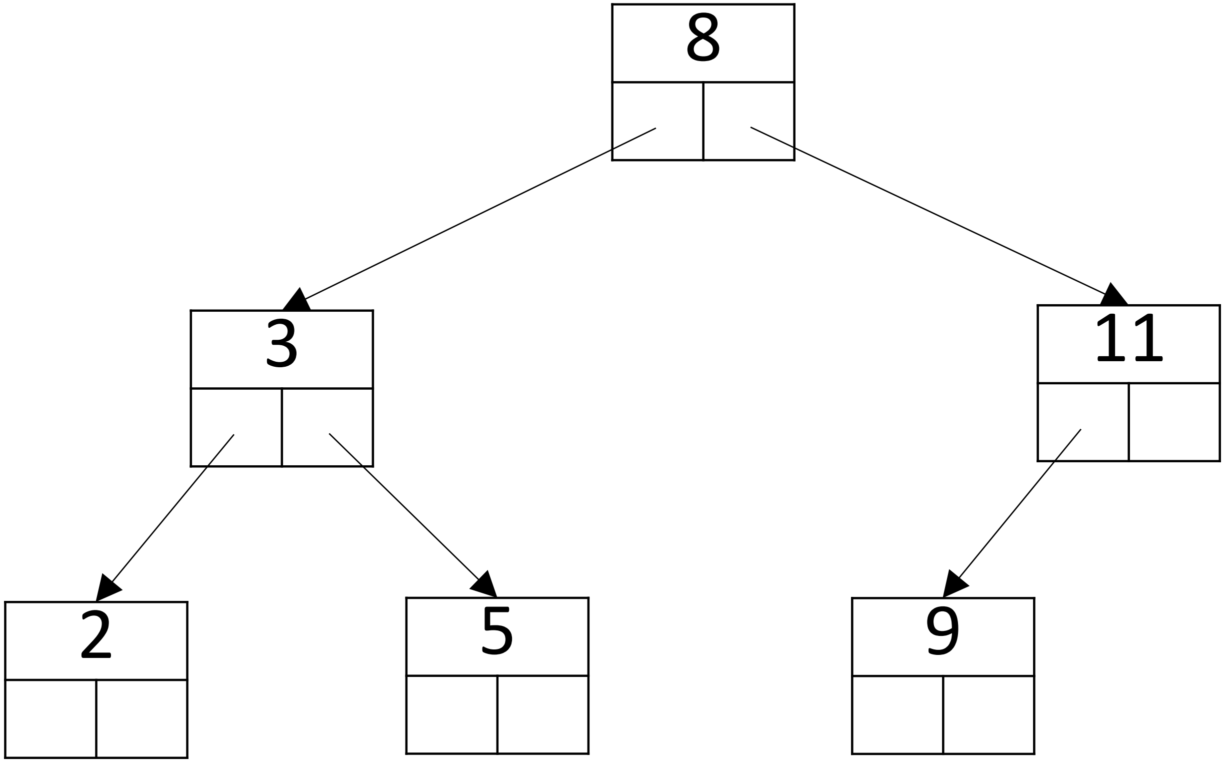


Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5



Exercise



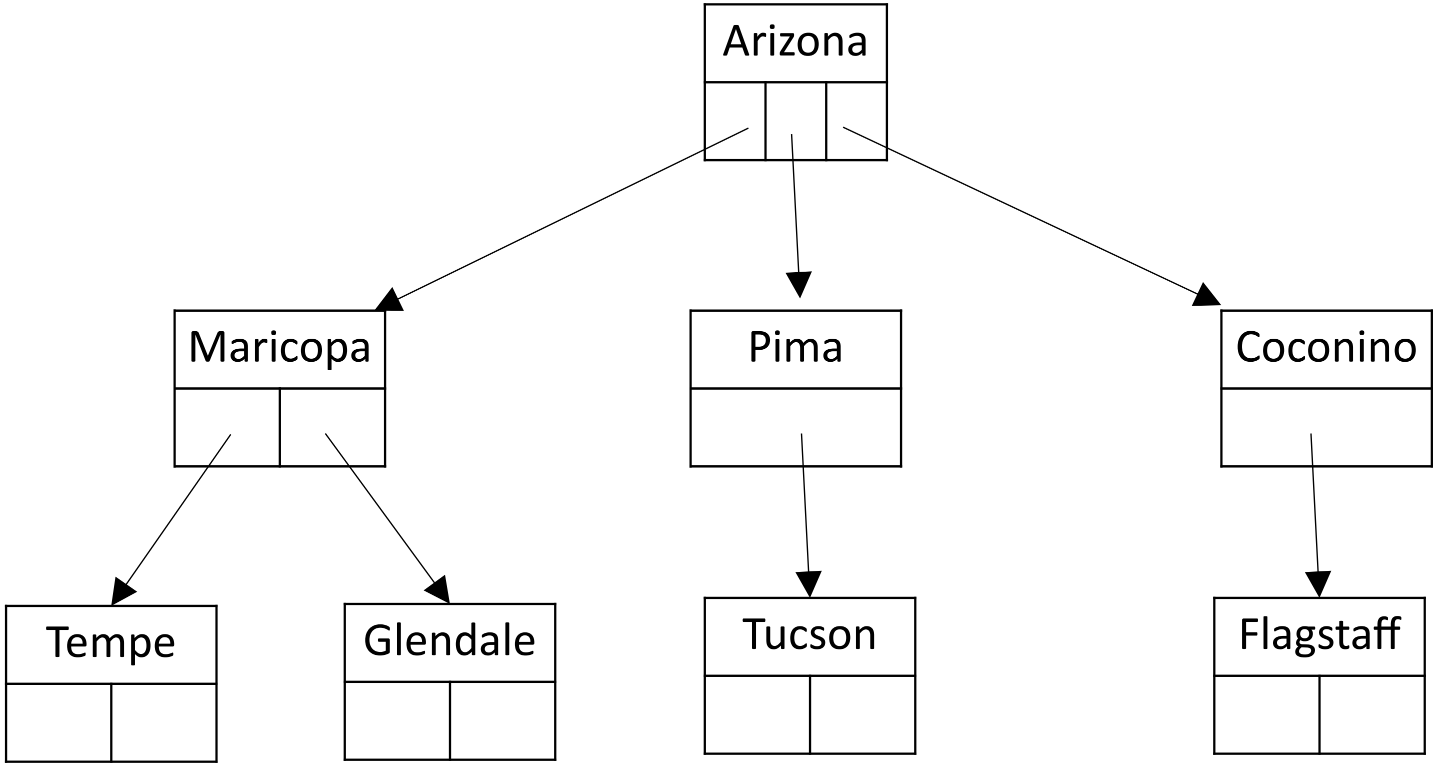
What three things are true of this diagram?

Exercise

Create a BST from this sequence: 7, -2, 10, 0, 13, 14, 3

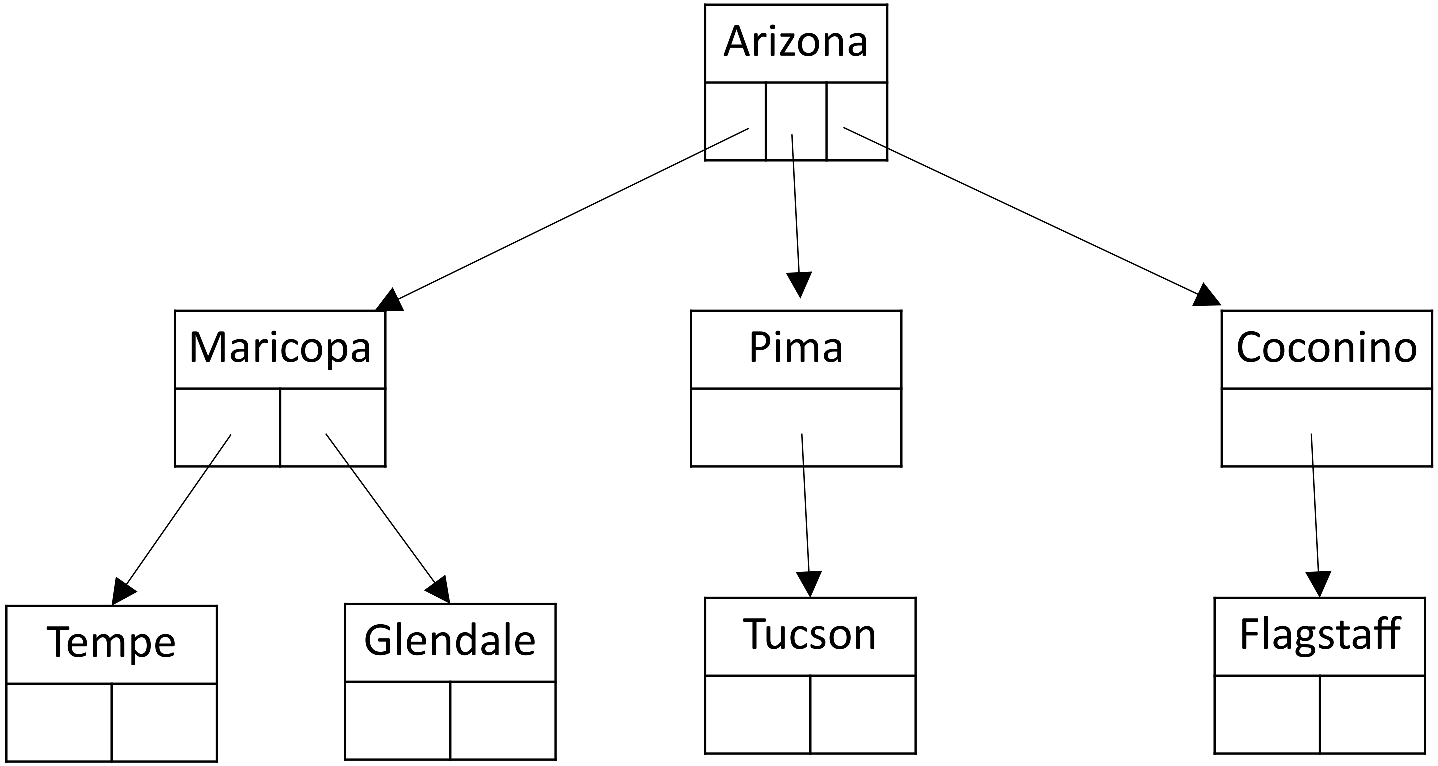
```
# Algo for creating a  BST
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.val:
        T.left = insert(T.left, v)
    elif v > T.val:
        T.right = insert(T.right, v)

    return T
```

Traversing Trees- Questions



Does Maricopa come before Coconino? If so, why?
Are the leaves more important than the nodes with children?
What does "order" mean here?

Exercise



Chose a method of "visiting" the nodes in the tree.
Write the nodes in the order you've chosen.

tree traversals

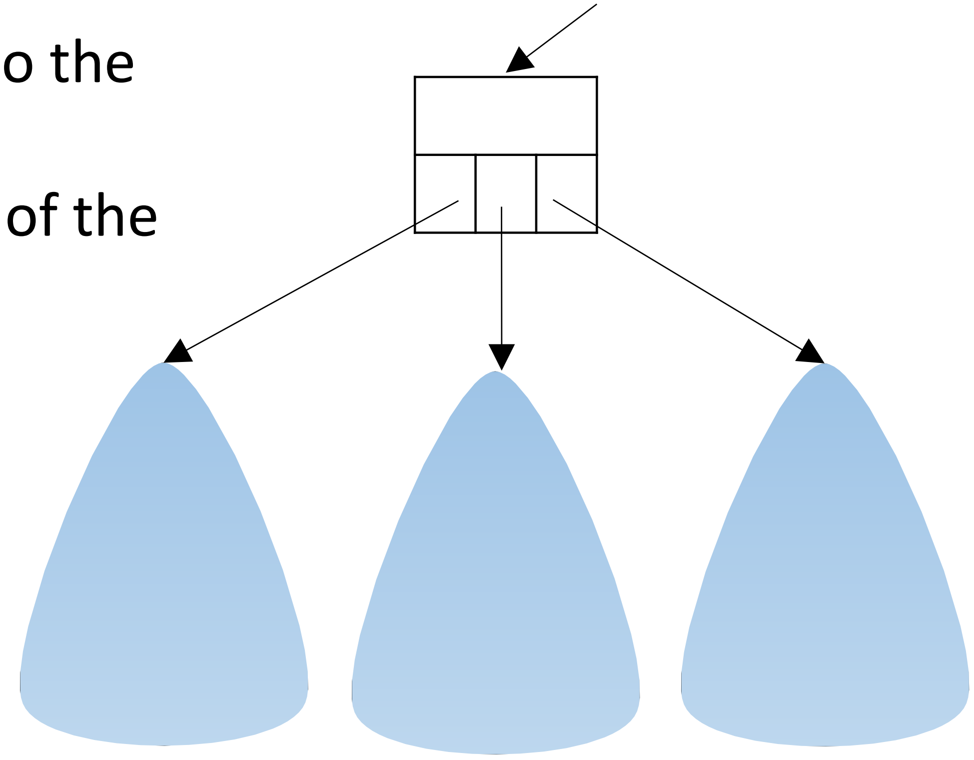
Tree traversals

- A *traversal* of a tree is a systematic way of visiting and processing the nodes of the tree

This usually comes down to the relative order between:

- traversing the subtrees of the node's children; and
- processing the node

"*Doing something with the value at the node*" – e.g., printing it out



Tree traversals

There are three widely used traversals:

- *Preorder traversal*
  - process the node first
  - then traverse (and process) its children

"pre" – visit node first

- *Inorder traversal*
  - traverse left subtree children
  - then process the node
  - then traverse right subtree

"in" – visit node in between

- *Postorder traversal*
  - traverse (and process) the children
  - then process the node

"post" – visit node last

BinaryTree Traversals

3 Traversals

	1	2	3
preorder:	Visit	-----	-----
inorder:	-----	Visit	-----
postorder:	-----	-----	Visit

Preorder traversal

Algorithm:

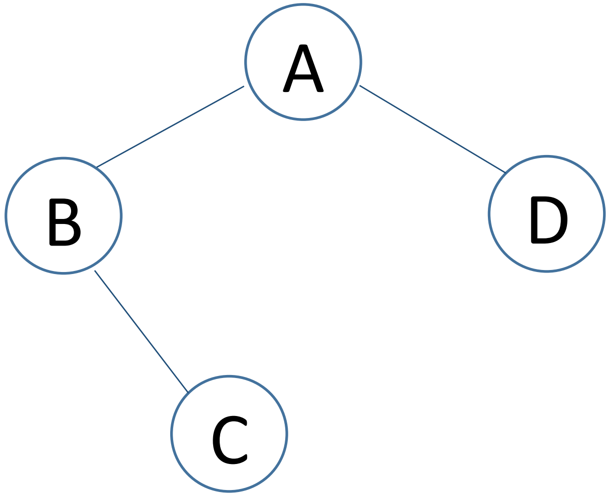
(where's the base case?)

Visit the node

Recurse on node's Left subtree

Recurse on node's Right subtree

Ex:



A B C D

Trace of preorder traversal

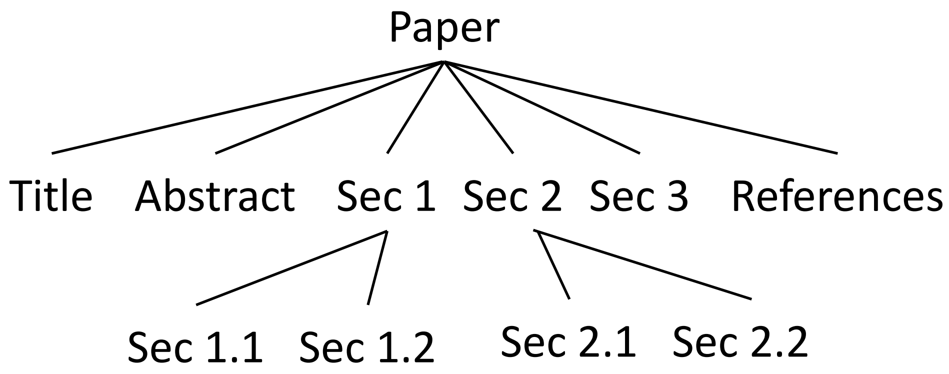
Output	Trace
A	Call preorder(A)
	Visit
B	Left (call preorder(B))
	Visit
	Left – return immediately
	Right (call preorder(C))
C	Visit
	Left – return
	Right – return
D	Right(D)
	Visit
	Left – return
	Right – return

Preorder traversal (n-ary)

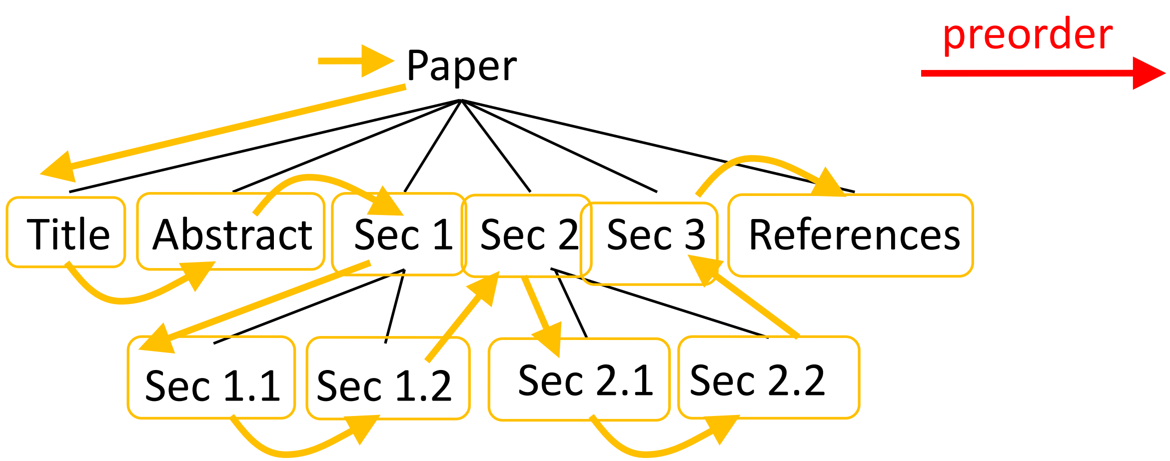
```
def preorder(T):
    process( T.value )
    for i in range(len(T.children)):
        preorder(T.children[i])
```

(where's the base case?)

Preorder traversal: Example



Preorder traversal: Example



Title
Abstract
…
Sec 1
…
Sec 1.1
…
Sec 1.2
…
Sec 2
…
Sec 2.1
…
Sec 2.2
…
Sec 3
…
References

Inorder traversal

Algorithm:

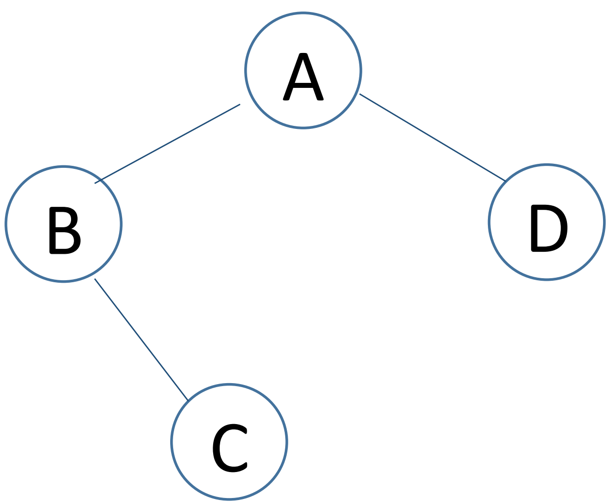
(where's the base case?)

Recurse on node's Left subtree

Visit node

Recurse on node's Right subtree

Ex:



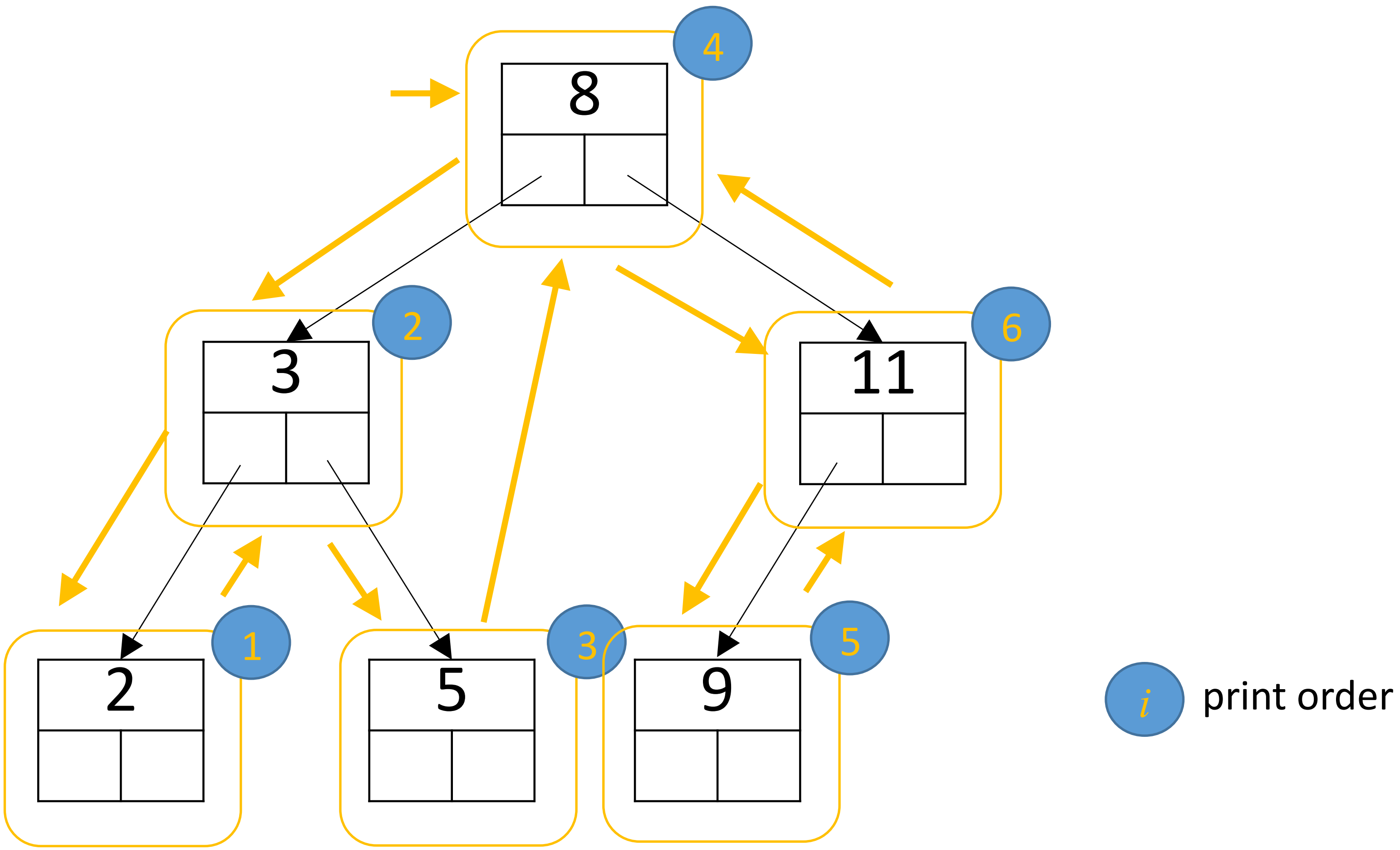
B C A D

Inorder traversal (binary trees)

```
def inorder(T):
    if T == None:
        return
    else:
        inorder(T.left())
        process(T.value)
        inorder(T.right())
```

Inorder traversal: Example

Print out the values in a BST in sorted order



Postorder traversal

Algorithm:

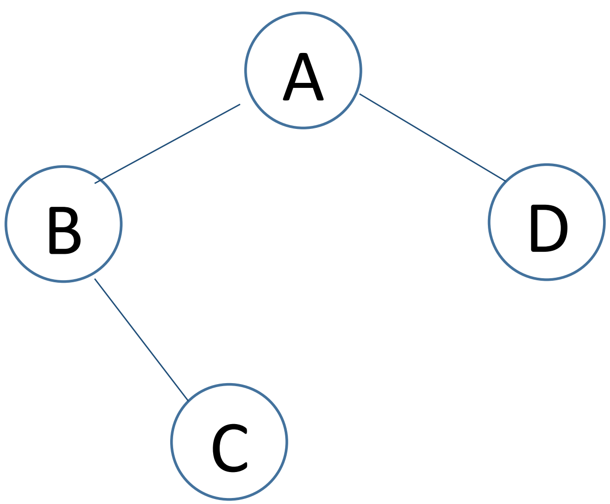
(where's the base case?)

Recurse on node's Left subtree

Recurse on node's Right subtree

Visit node

Ex:



C B D A

Postorder traversal (n-ary)

```
def postorder(T):
    for i in range(len(T.children)):
        postorder(T.children[i])   # visit all children first
    process(T.value)
```

Postorder traversal- Example

Problem: evaluate this expression

(x + y * 3) / (n – 1)

suppose that: x = 3, y = 2, n = 4

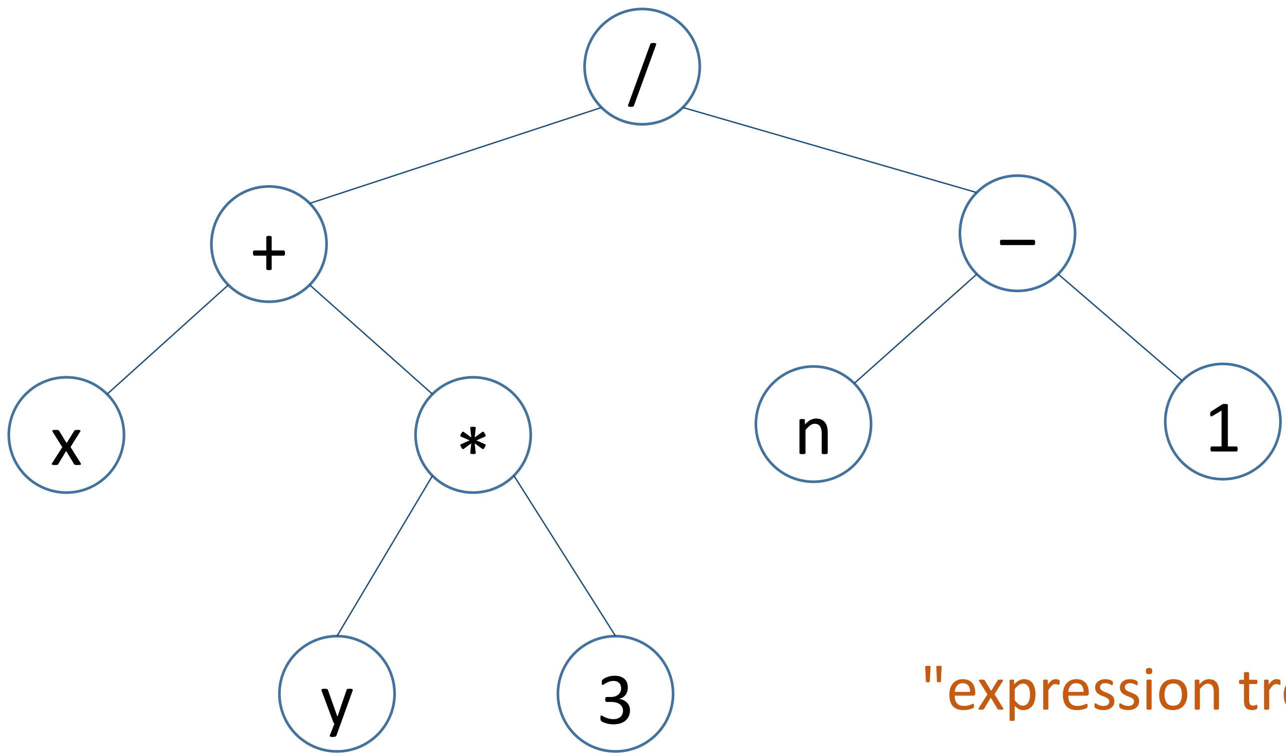
Solution:

1) Convert the fully parenthesized expression into a binary tree
   - use an auxiliary stack
   - use the algorithm covered in section
2) Evaluate the tree binary tree
   - use a postorder traversal of the tree

Postorder traversal: Example

Evaluate: (x + y * 3) / (n – 1)

suppose that: x = 3, y = 2, n = 4

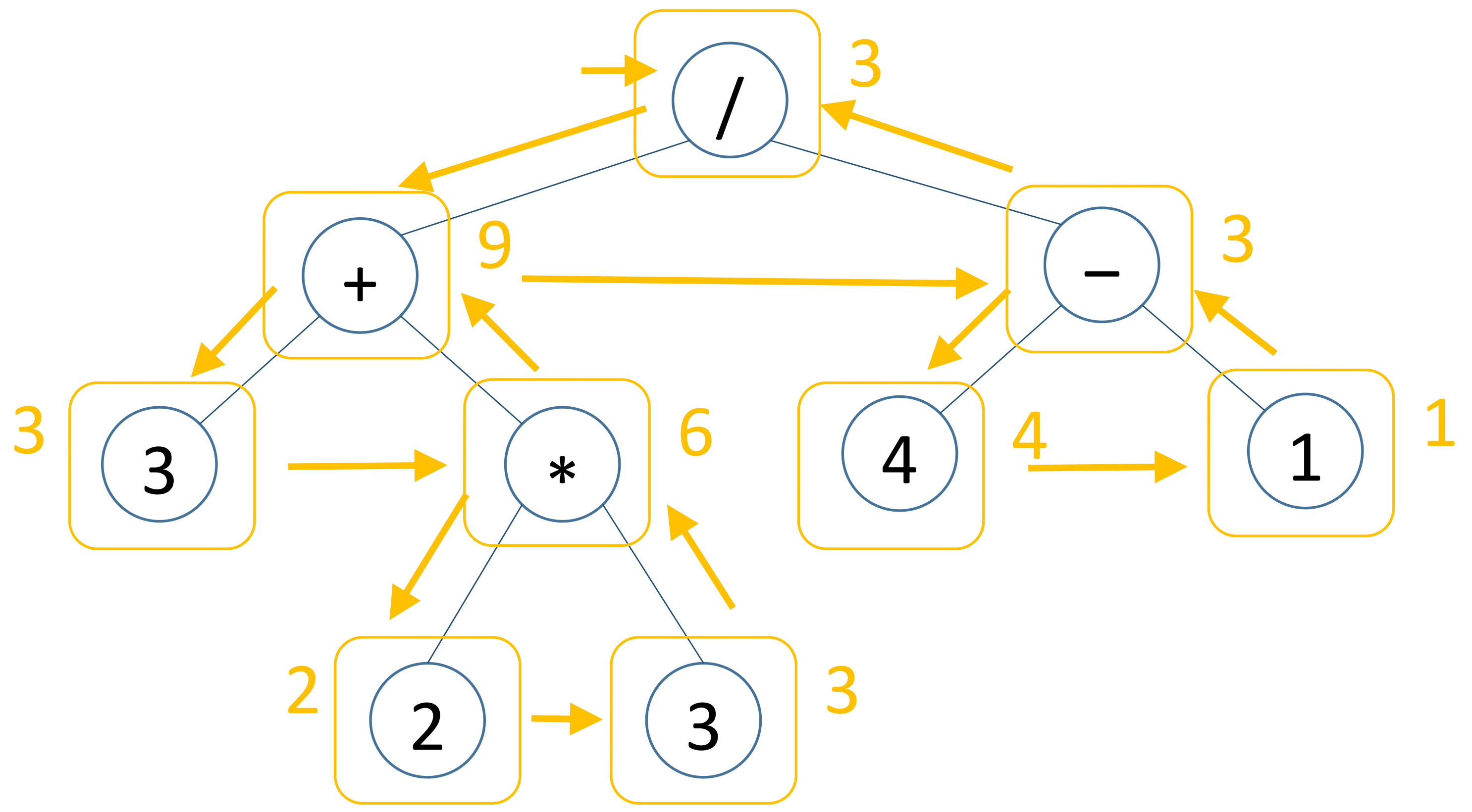


"expression tree"

Postorder traversal: Example

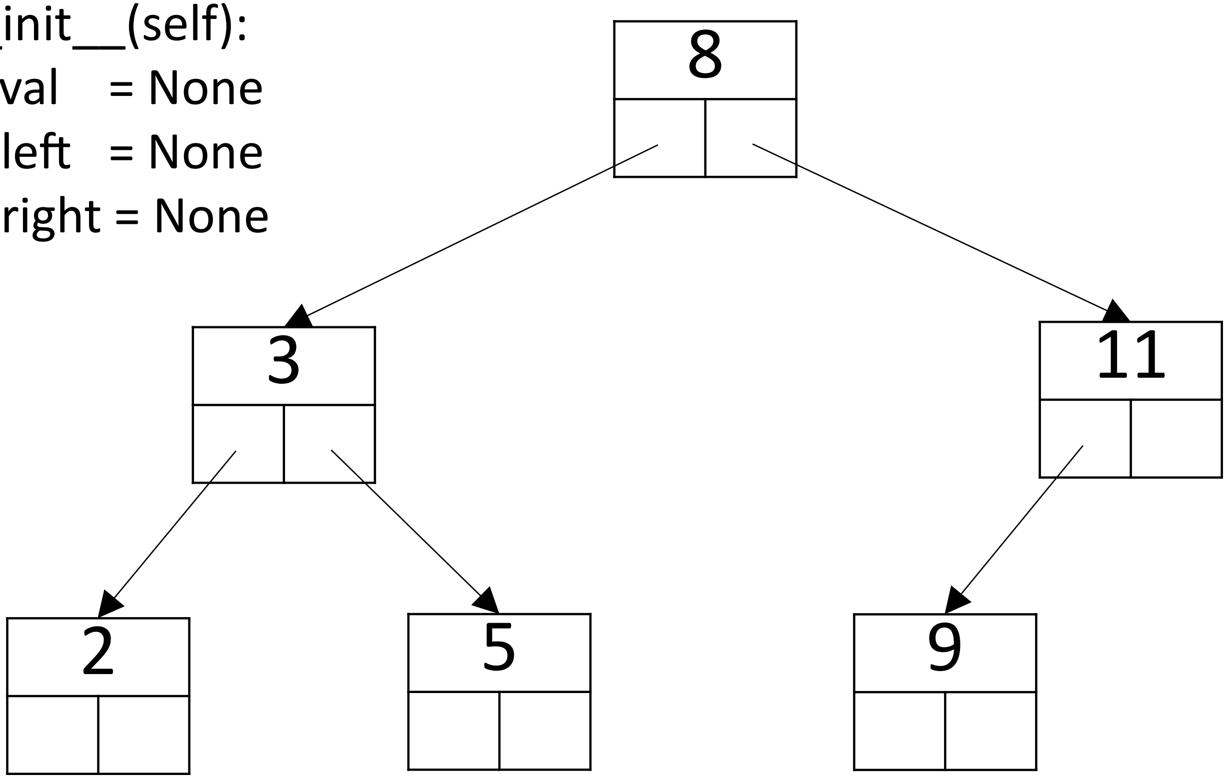
Evaluate: (x + y * 3) / (n – 1)

suppose that: x = 3, y = 2, n = 4



EXERCISE

```
class BinarySearchTree:
    def __init__(self):
        self.val   = None
        self.left  = None
        self.right = None
```



Write a traversal for a BST that prints the values in this order: 2, 3, 5, 8, 9, 11 (sorted order)

Answer

traverse a BST and print the nodes in sorted order

```
def traverse(t):
    if t == None:
        return
    else:
        traverse(t.left)
        print(t.val)
        traverse(t.right)
```

EXERCISE

Given a binary tree, write a function `count_leaves(t)` that counts the number of leaf nodes.

- What is the base case? (Is there more than one?)
- What is the smaller amount of computation for the next round of recursion?

Answer

count the leaves of a tree

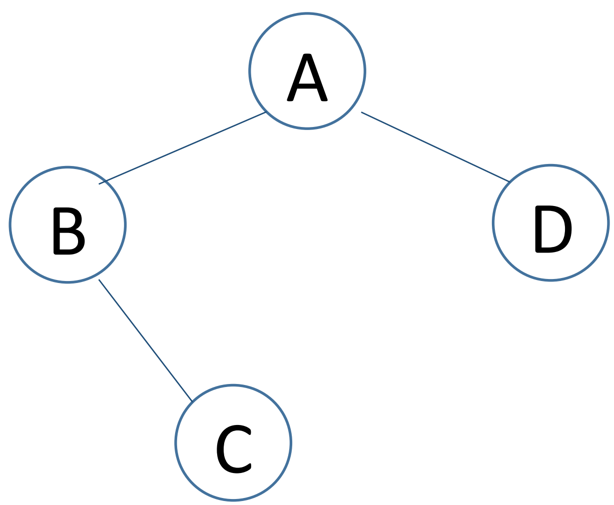
```
def count_leaves(t):
    if t == None:
        return 0
    if t.left == None and t.right == None:
        return 1
    else:
        return count_leaves(t.left) + \
               count_leaves(t.right)
```

Trees ↔ traversals

What is the preorder traversal of this tree?

Inorder?

Postorder?



Trees ↔ traversals

- Given a tree, we can figure out its traversals
- Does the converse hold?

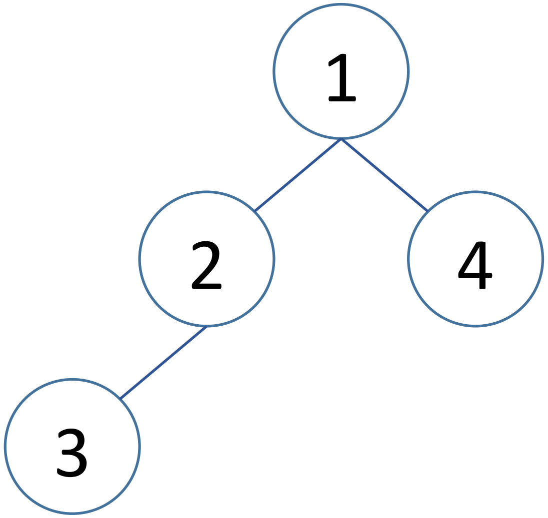
  I.e., given a traversal, can we figure out the tree?

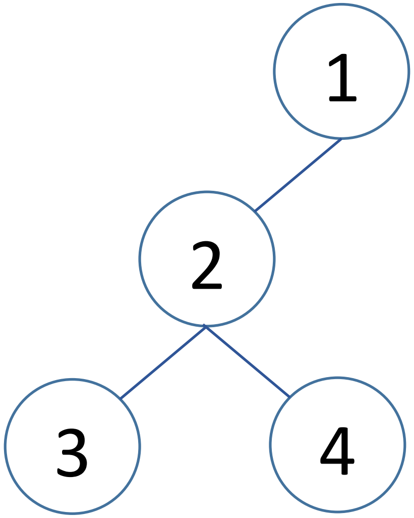
    preorder: 1 2 3 4

Trees ↔ traversals

- The two trees below have the same preorder traversal.

Preorder traversal = 1 2 3 4





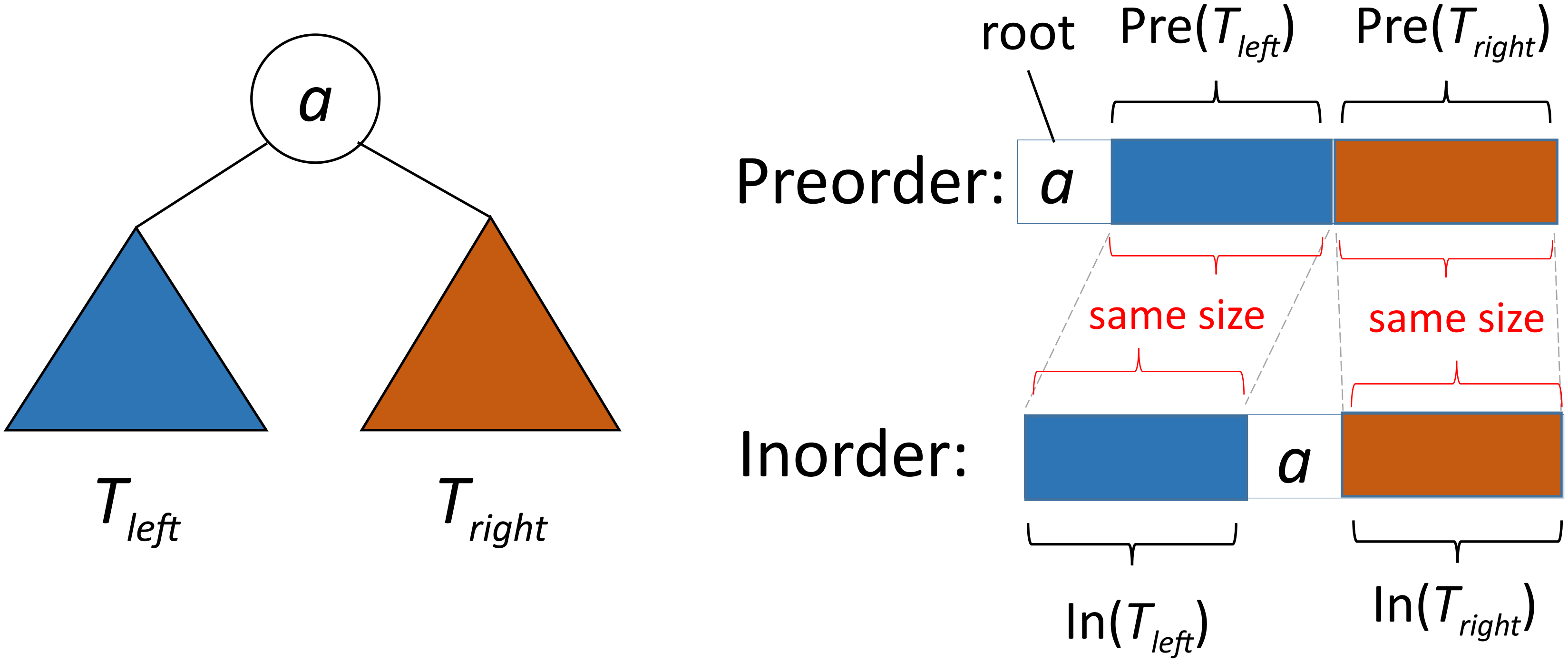
Trees $\leftrightarrow$ traversals

- We cannot derive a *unique* tree from a single traversal

- What if we have two traversals?
  - Inorder: 3 5 7 9 4
    - hard to tell where the root is
  - Preorder: 5 3 9 7 4
    - now we know

- Let's draw the tree & figure out an algo to do it

Trees ↔ traversals



Trees $\leftrightarrow$ traversals

- Given a preorder *<u>and</u>* an inorder traversal, create the tree

- Given:
  - preorder_list
  - inorder_list

node sequences from traversals of a tree

Need to do: build a function:

traversals_to_tree(preorder_list, inorder_list)

that will return the tree for the given traversals.

Trees ↔ traversals

- Given:
  - preorder_list + inorder_list
- Suppose we can figure out:
  - root
  - preorder_left + preorder_right
  - inorder_left + inorder_right

Trees ↔ traversals

- Given:
  - preorder_list + inorder_list
- Suppose we can figure out:
  - root
  - preorder_left + preorder_right
  - inorder_left + inorder_right
- Then:

  traversals_to_tree(preorder_left, inorder_left)

  traversals_to_tree(preorder_right, inorder_right)

Trees ↔ traversals

- Given:
  - preorder_list + inorder_list
- Suppose we can figure out:
  - root
  - preorder_left + preorder_right
  - inorder_left + inorder_right
- Then:

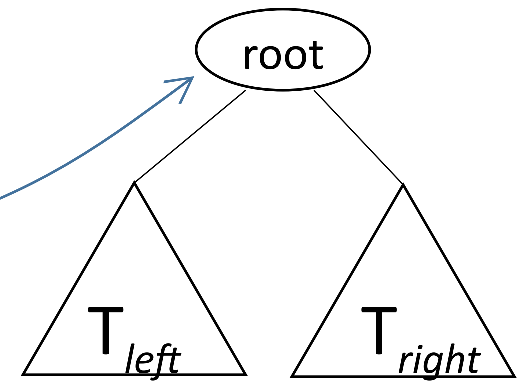
traversals_to_tree(preorder_left, inorder_left) $\rightarrow T_{\mathit{left}}$

traversals_to_tree(preorder_right, inorder_right) $\rightarrow T_{\mathit{right}}$

recursion

Trees ↔ traversals

- Given:
  - preorder_list + inorder_list
- Suppose we can figure out:
  - root
  - preorder_left + preorder_right
  - inorder_left + inorder_right



- Then:

traversals_to_tree(preorder_left, inorder_left) $\rightarrow$ T_{left}

traversals_to_tree(preorder_right, inorder_right) $\rightarrow$ T_{right}

more traversals

Consider: game playing

Goal: to write a program to play a 2-person game (e.g., tic-tac-toe, chess, go, ...)

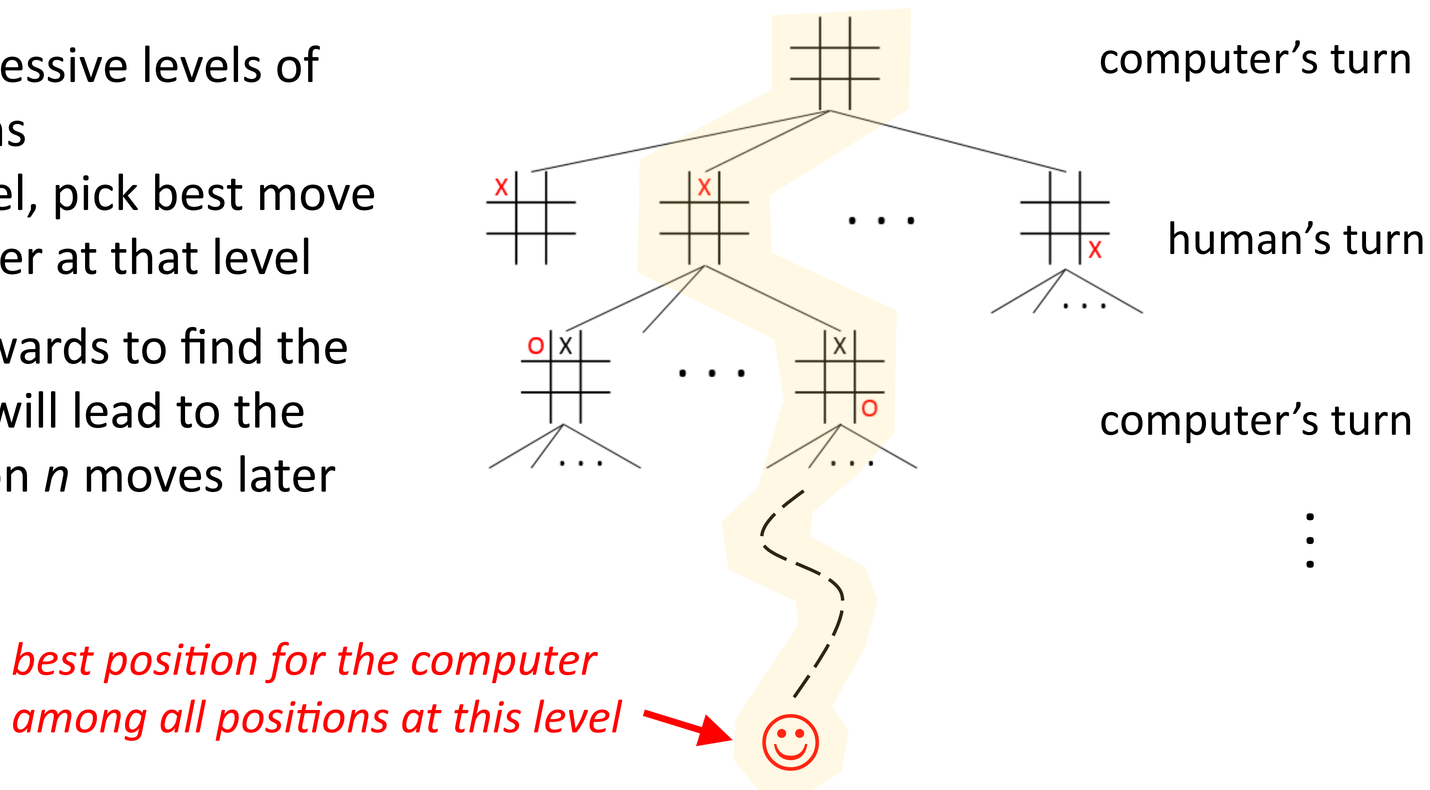
How does this work?

Consider: game playing

Goal: to write a program to play a 2-person game (e.g., tic-tac-toe, chess, go, …)

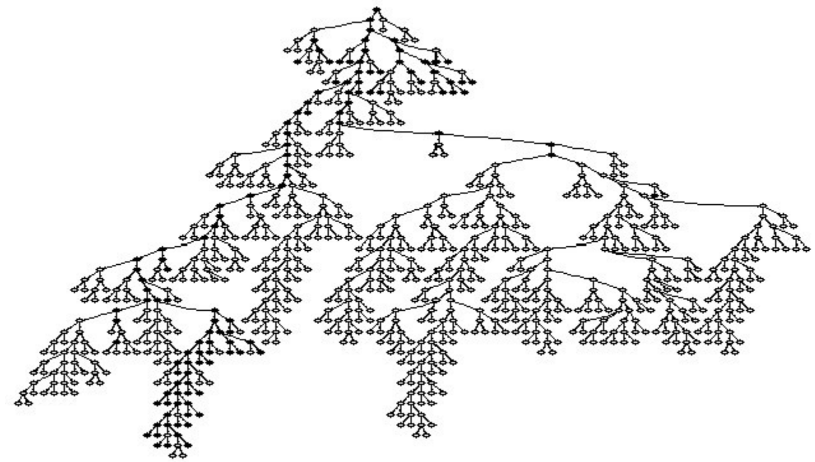
Generate successive levels of board positions

- At each level, pick best move for the player at that level
- Work backwards to find the move that will lead to the best position *n* moves later



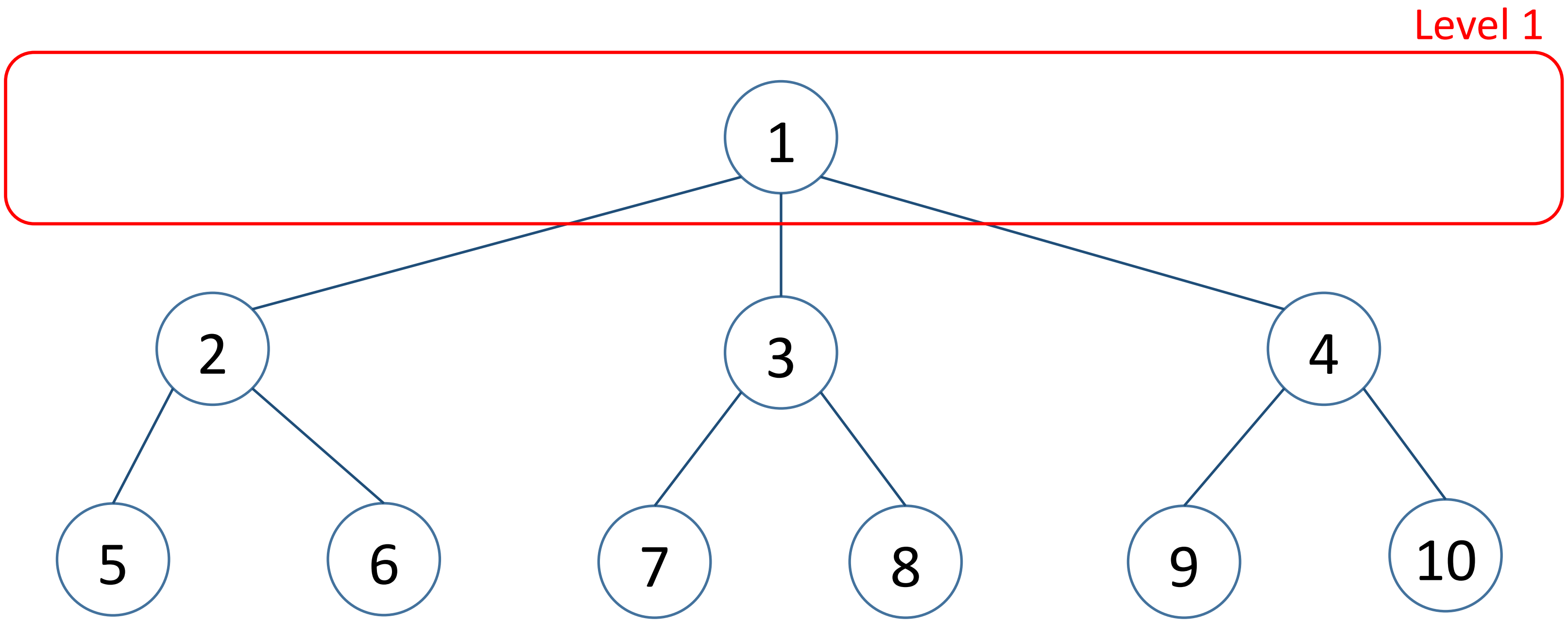
Consider: game playing

- For a nontrivial game (e.g., chess, go) the tree is usually too large to build or explore fully
  - also, usually there are time constraints on play
  - our previous tree traversal algorithms don't work
- Game-playing algorithms typically explore the tree level by level
  - consider the nodes at depth 1, then depth 2, etc.

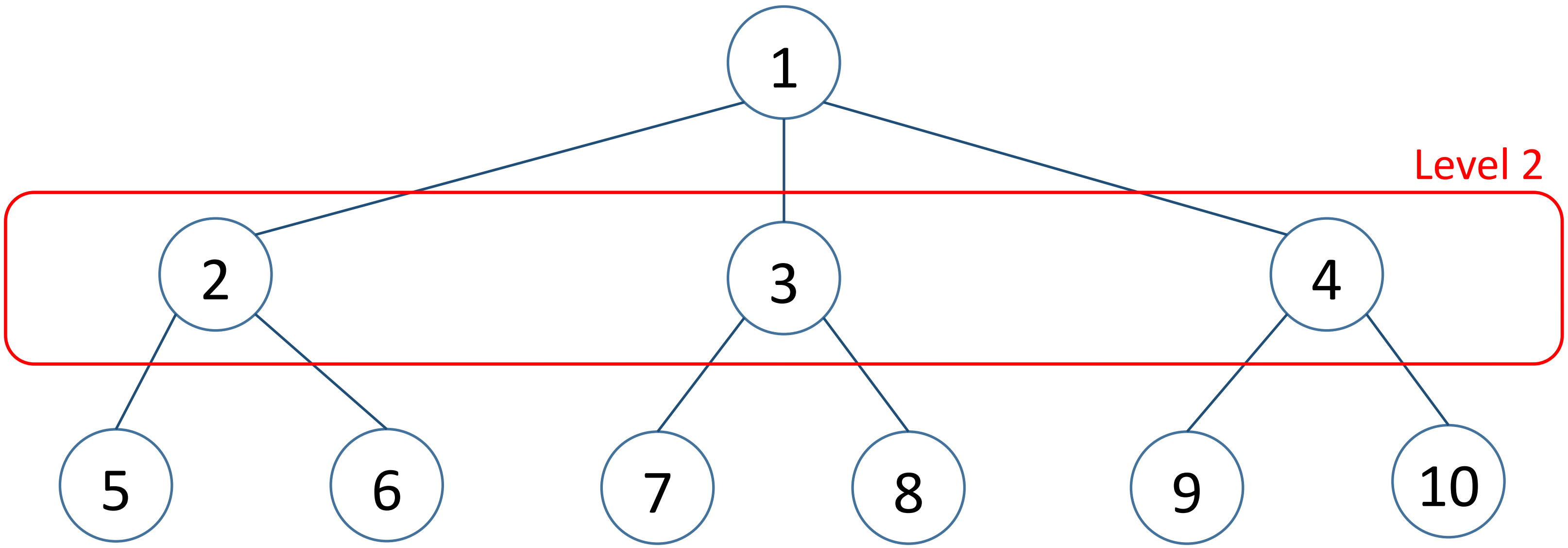


A game tree

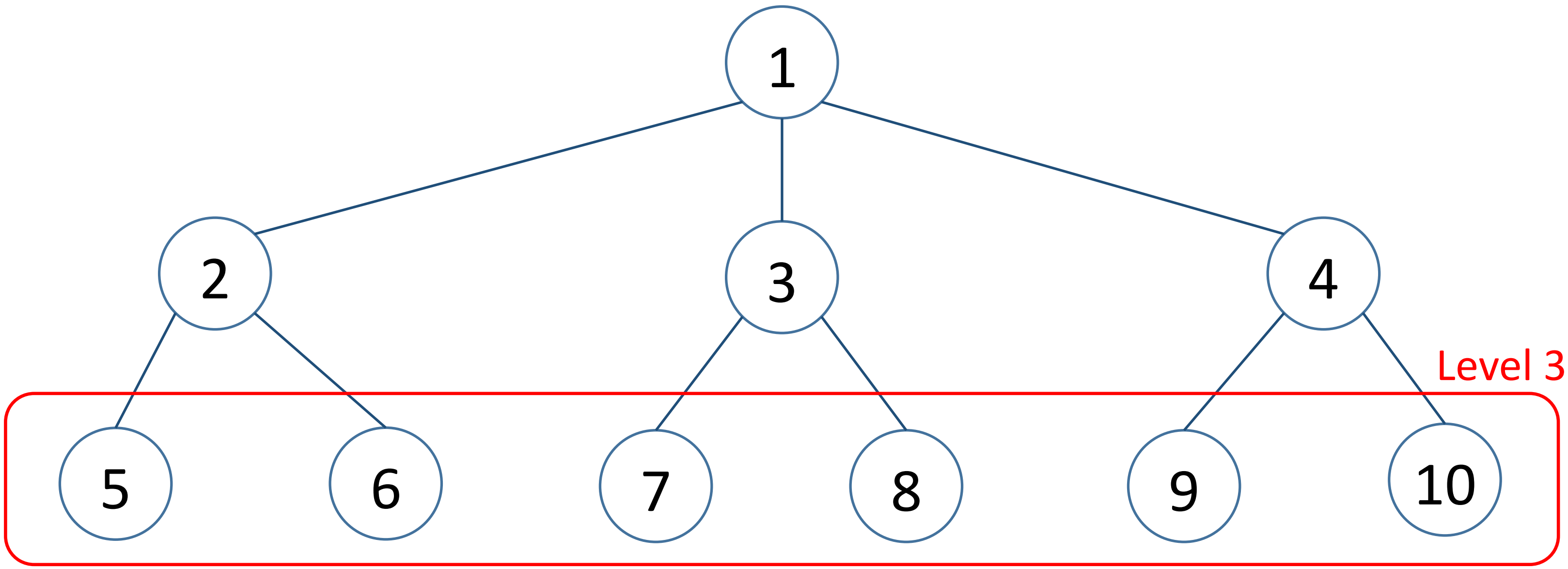
Level-by-level tree traversal



Level-by-level tree traversal

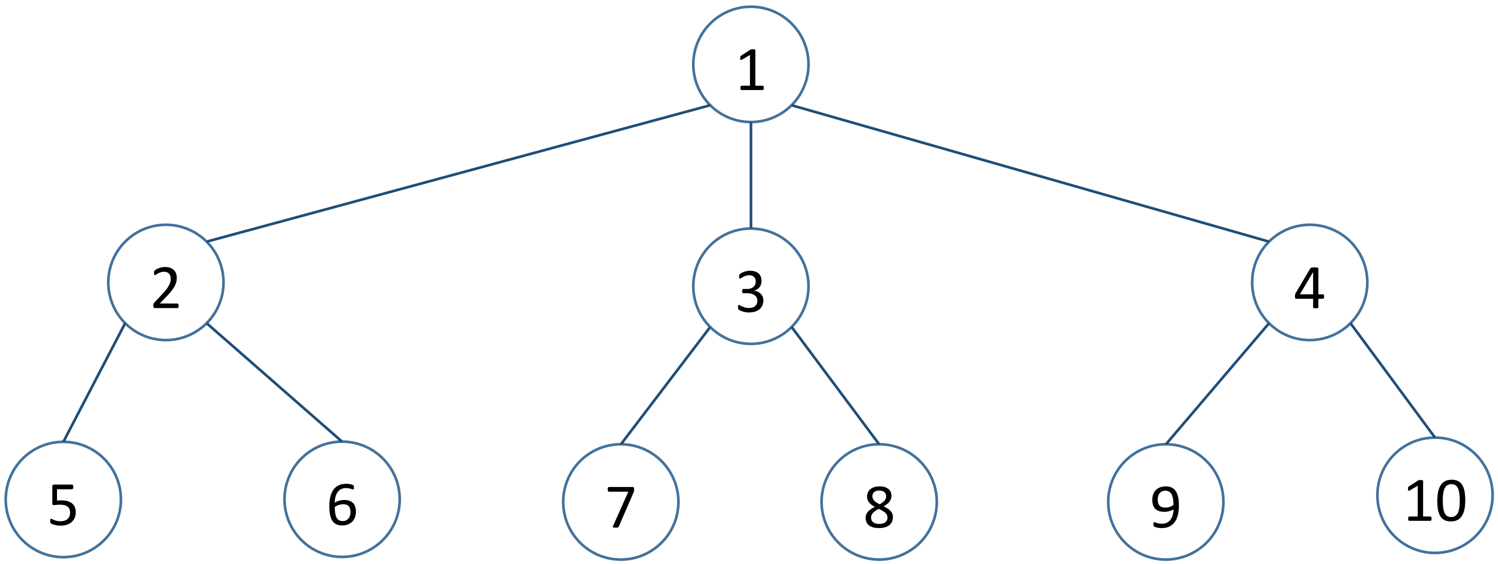


Level-by-level tree traversal



This order of traversal is called *breadth-first traversal*

Breadth-first tree traversal



Breadth-first traversal order:

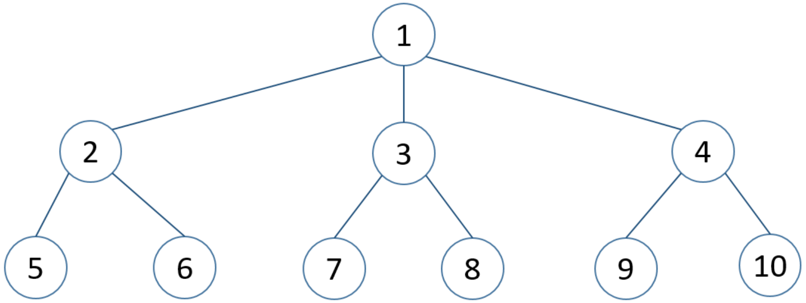
1 2 3 4 5 6 7 8 9 10

Breadth-first tree traversal

Data structure: use a queue *q*

Algorithm:

- Create a queue *q*
- Put the root in *q*
- While *q* not empty
  - node = *q*.dequeue()
  - process node
  - enqueue its children



Breadth-first vs. Depth-first

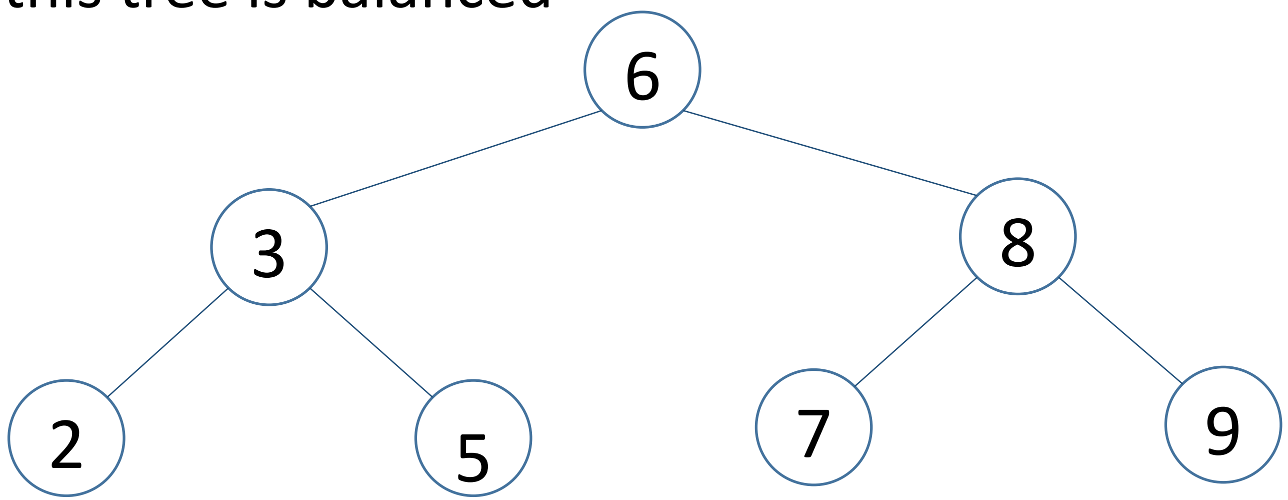
- Stacks and queues are closely related structures
- What if we use a stack in our tree traversal?
  - the deeper levels of the tree are explored first
  - this is referred to as *depth-first traversal*

BST / Complexity

Binary Search tree: complexity

Searching: O(log n), where n is the number of elements in the tree

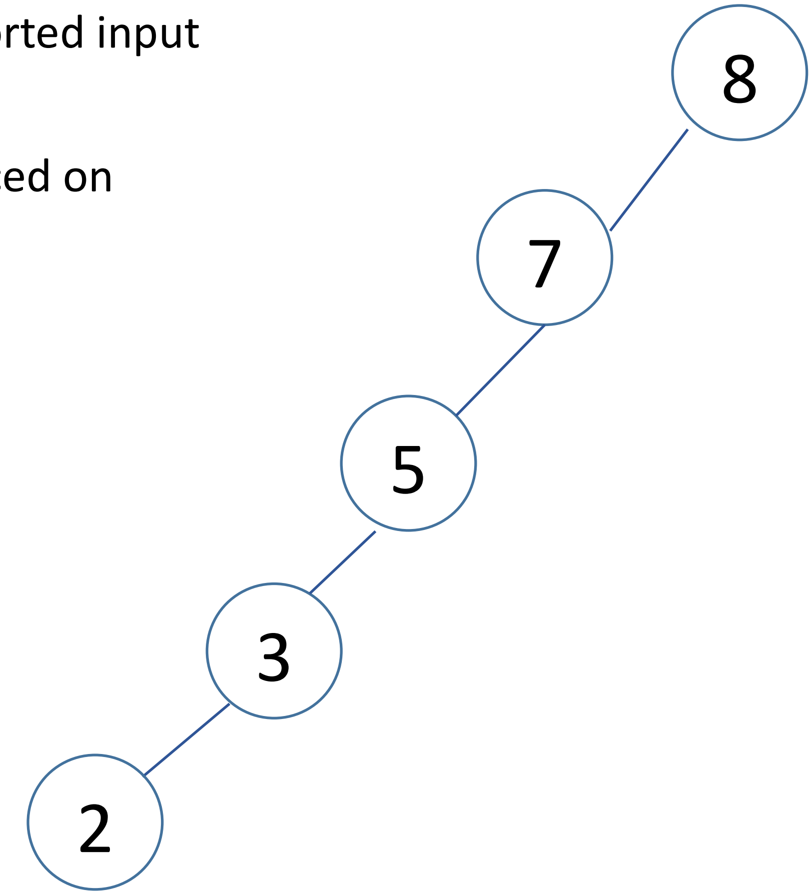
Note: this tree is balanced



What if the tree is not balanced?

BST / Complexity

- Unbalanced BST
  - How many comparisons does it take to find 2?
  - Worst case, complexity can be O(n)
  - Skewed trees can result from sorted input
- Balanced trees
  - AVL Trees: trees are kept balanced on insertion, deletion, etc.



Trees: summary

- An n-ary tree represents a hierarchy
- They show up in all kinds of contexts
  - including many in computer science
- Various kinds of tree traversals reflect different ways of processing the information and structure of trees
- Recursion is often the simplest way to process trees