

# CSc 120

## Introduction to Computer Programming II

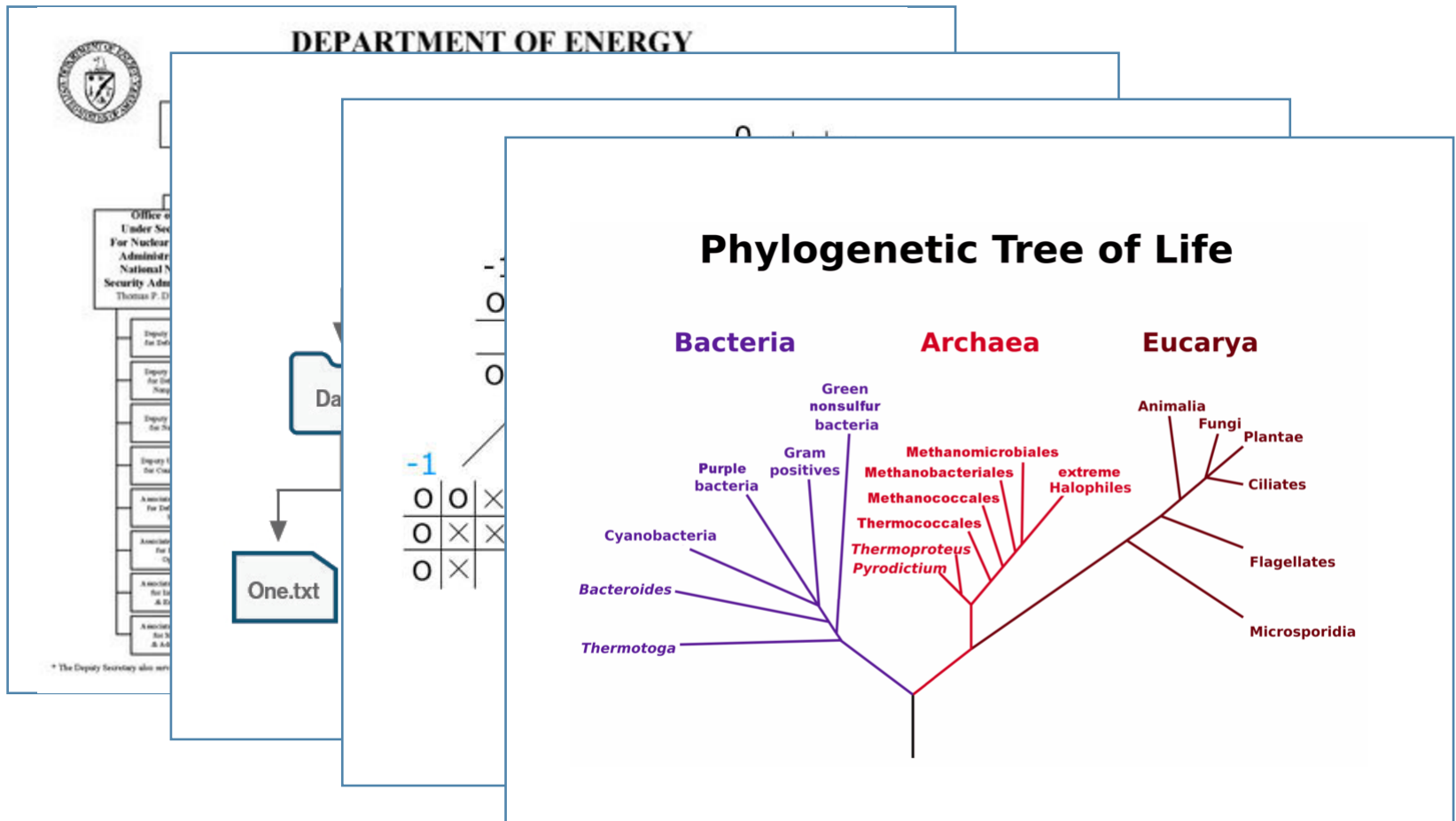
*Adapted from slides  
by Dr. Saumya Debray*

### 14: Trees

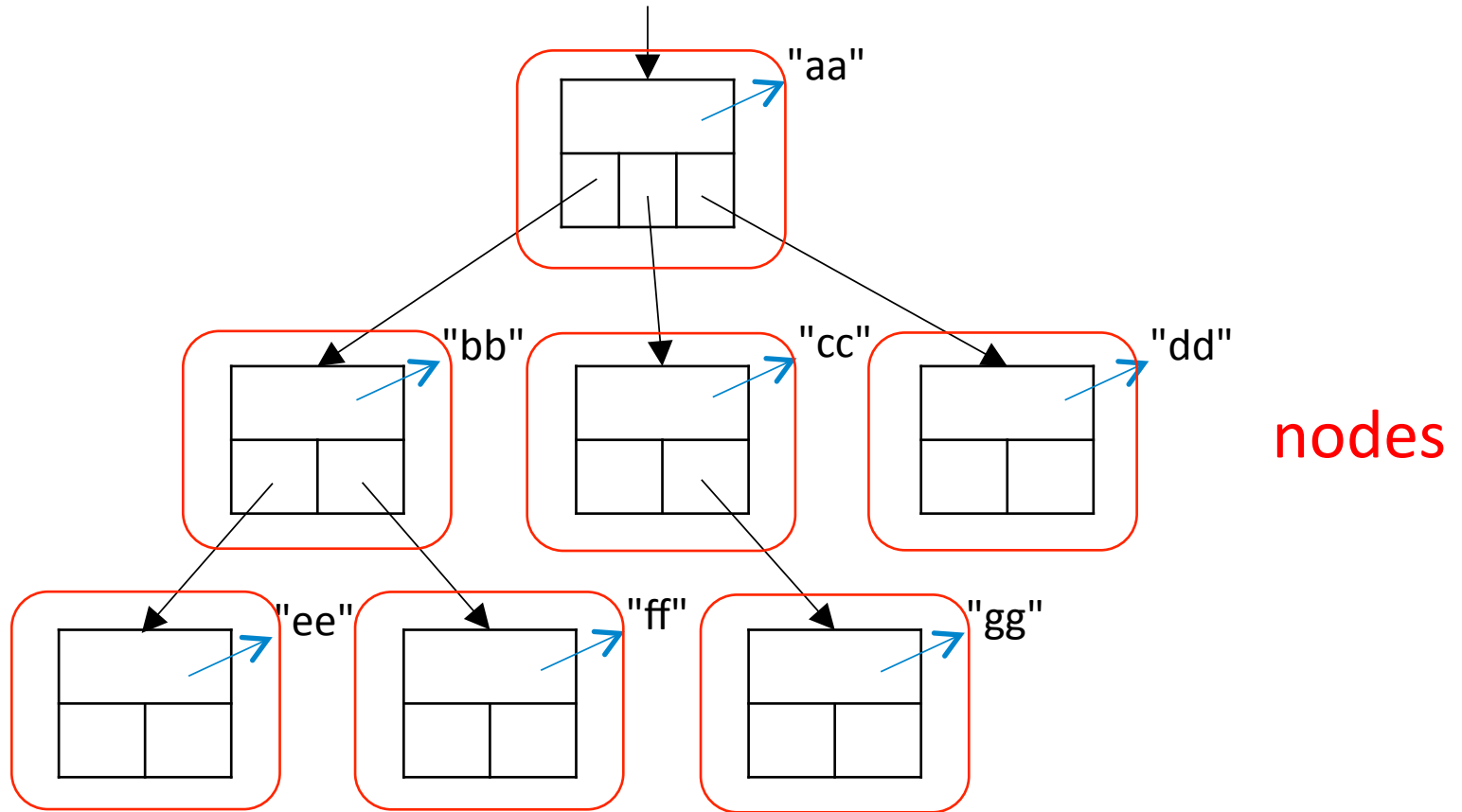
# trees: basic concepts

# Hierarchies

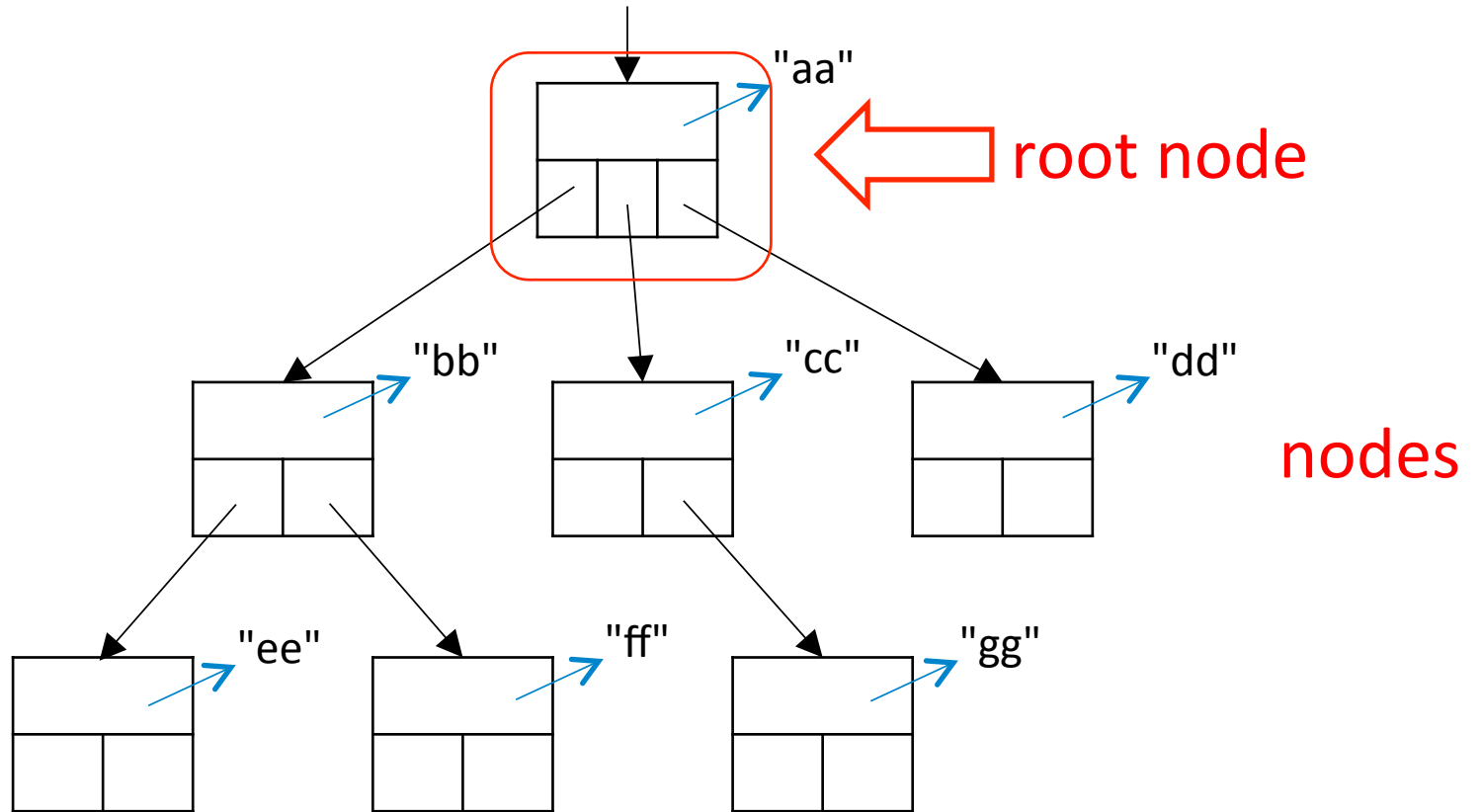
- Hierarchically organized "stuff" are everywhere



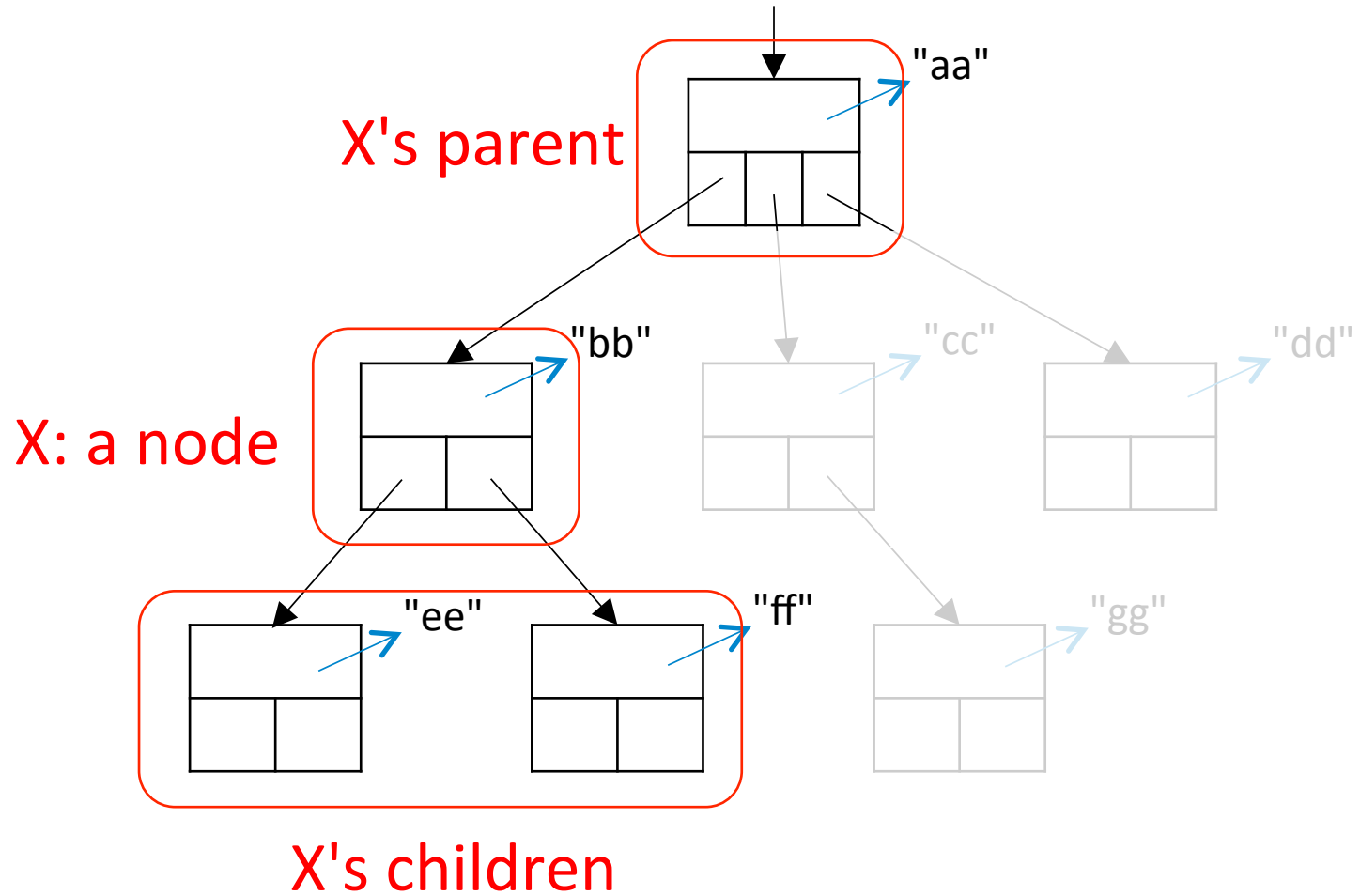
# Trees



# Trees



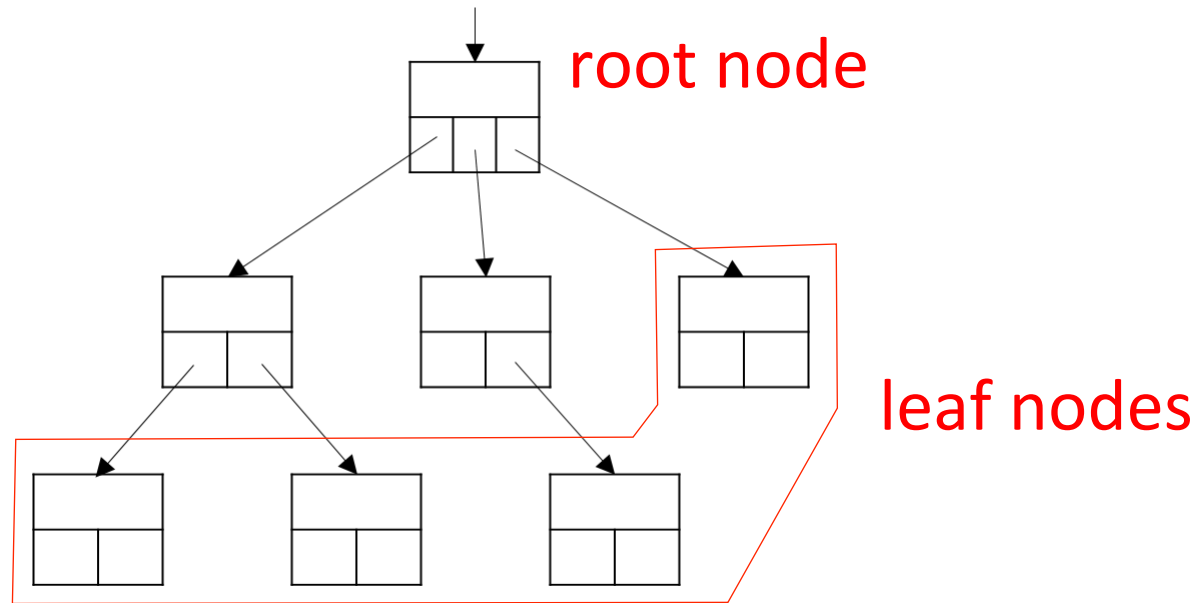
# Trees



# Trees: terminology

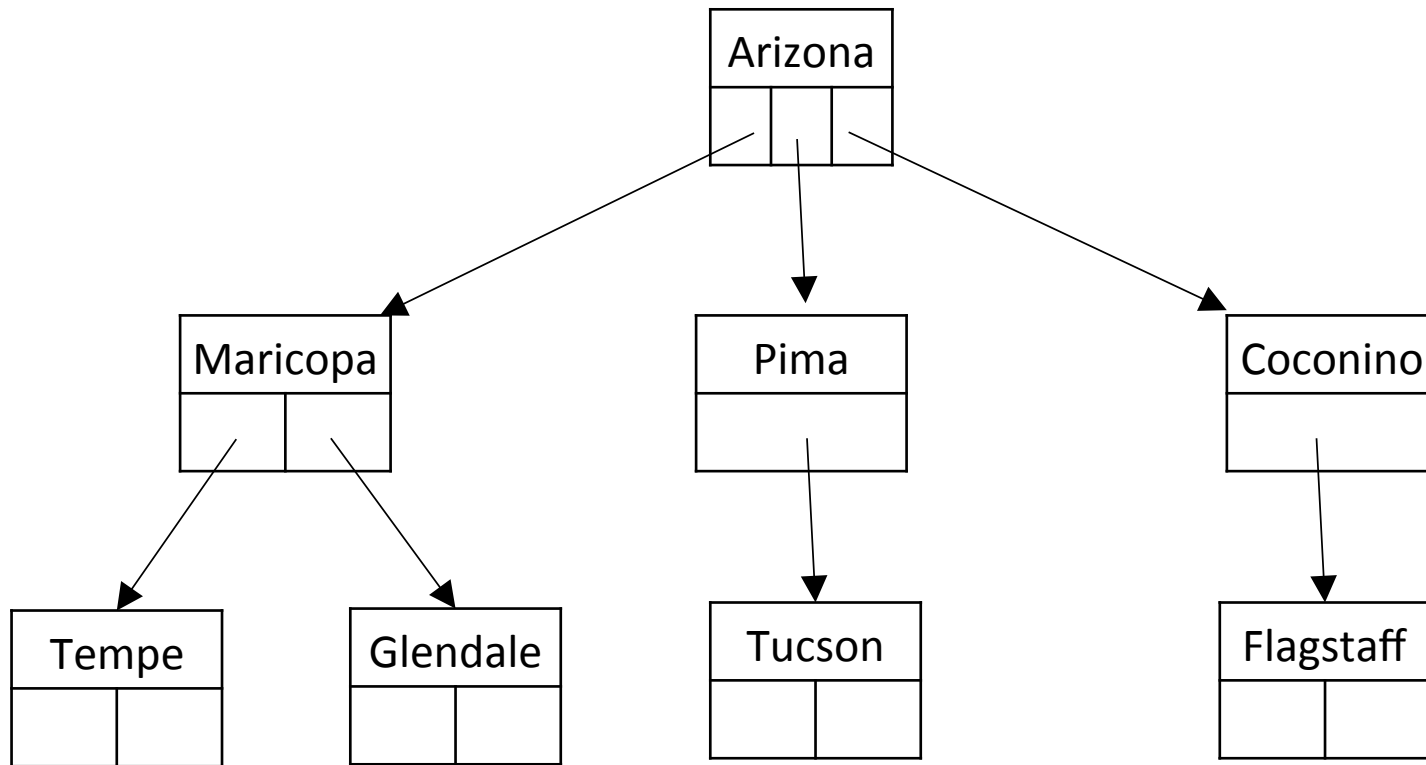
- A tree is a collection of nodes
  - Each node has:
    - $\geq 0$  child nodes
    - 0 or 1 parent nodes
- } Y is a child of X  $\Leftrightarrow$  X is a parent of Y
- A node with 0 children is called a *leaf node*
  - A node with 0 parent nodes is called the *root node*
  - A tree has:
    - $\geq 1$  leaf nodes
    - exactly one root node

# Trees: leaves and root



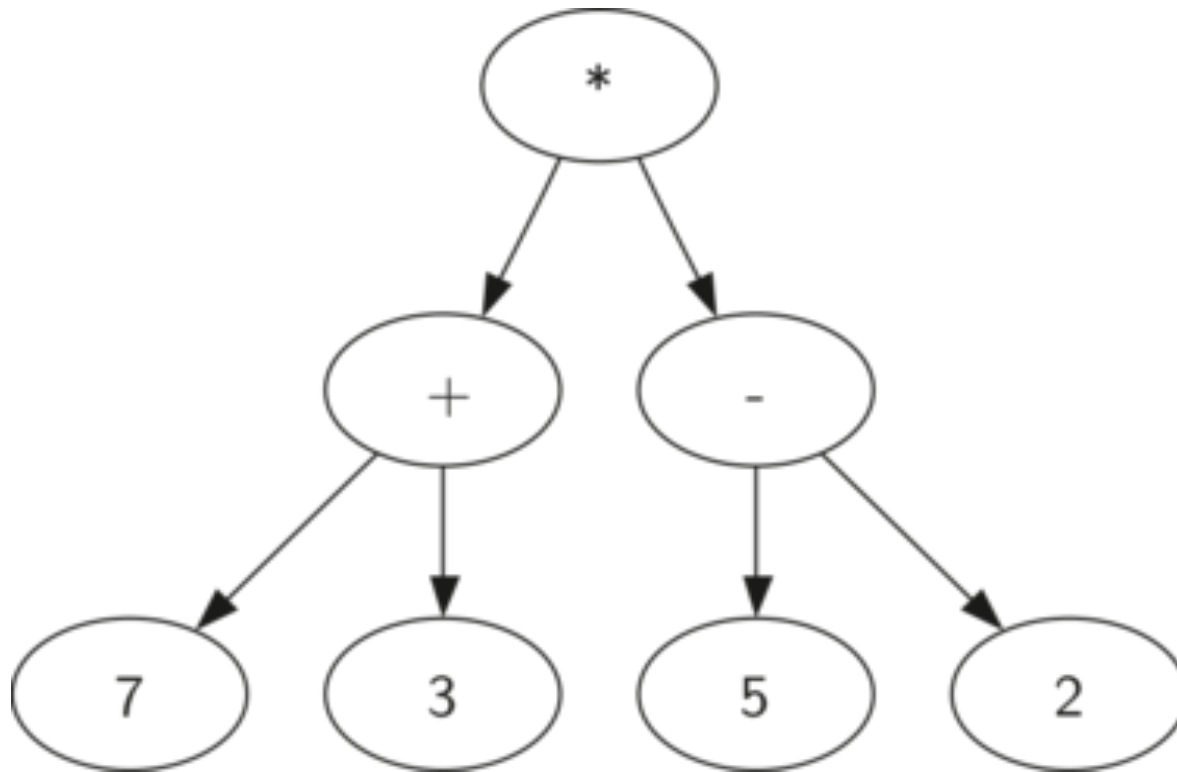


# Tree: Example



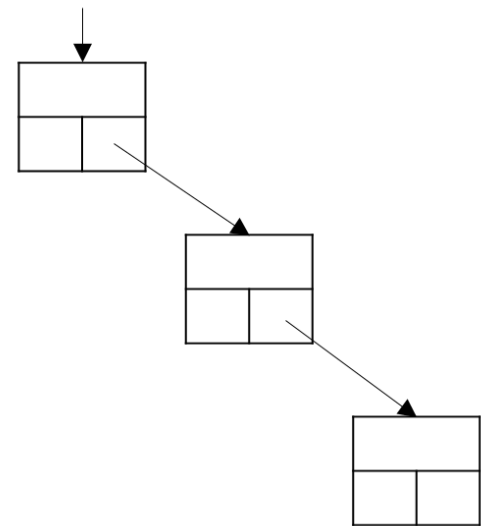
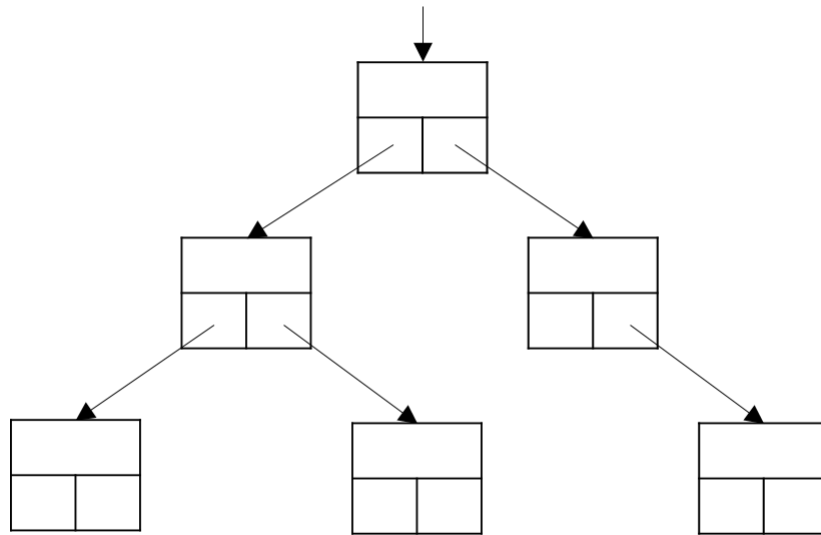
# Tree: Example

$(7+3)*(5-2)$

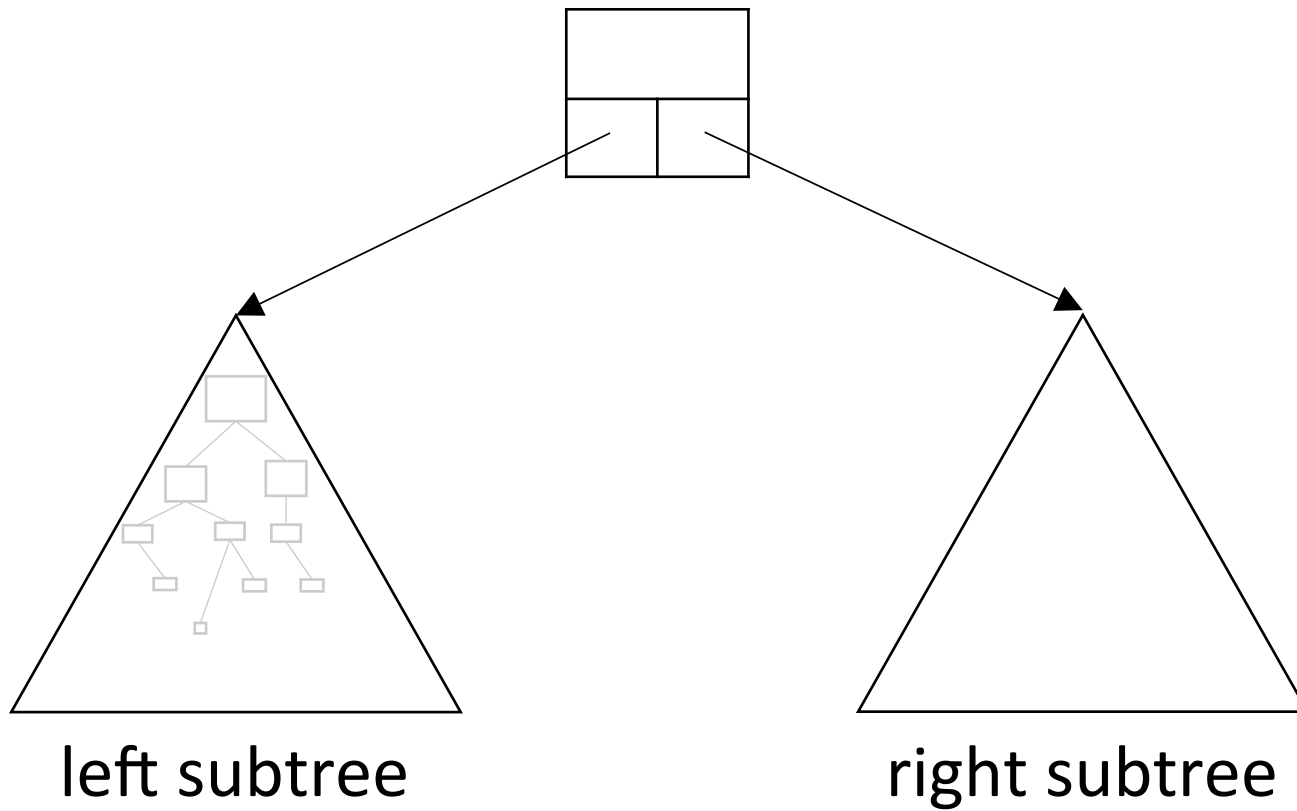


# Binary trees

- 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., an array, 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 BinaryTree:
```

```
    def __init__(self, value):
```

```
        self._value = value           # the value at the node
```

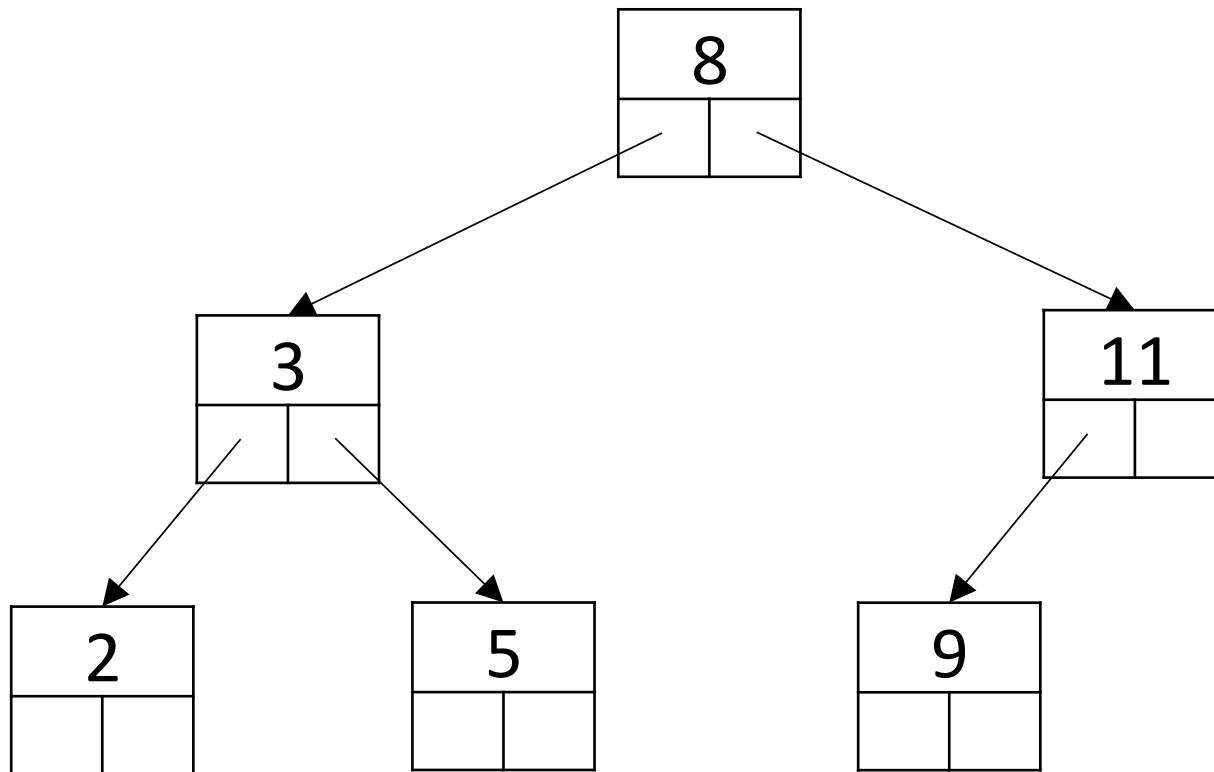
```
        self._lchild = None          # left child
```

```
        self._rchild = None          # right child
```

```
...
```

# binary search trees

# Examine this binary tree:



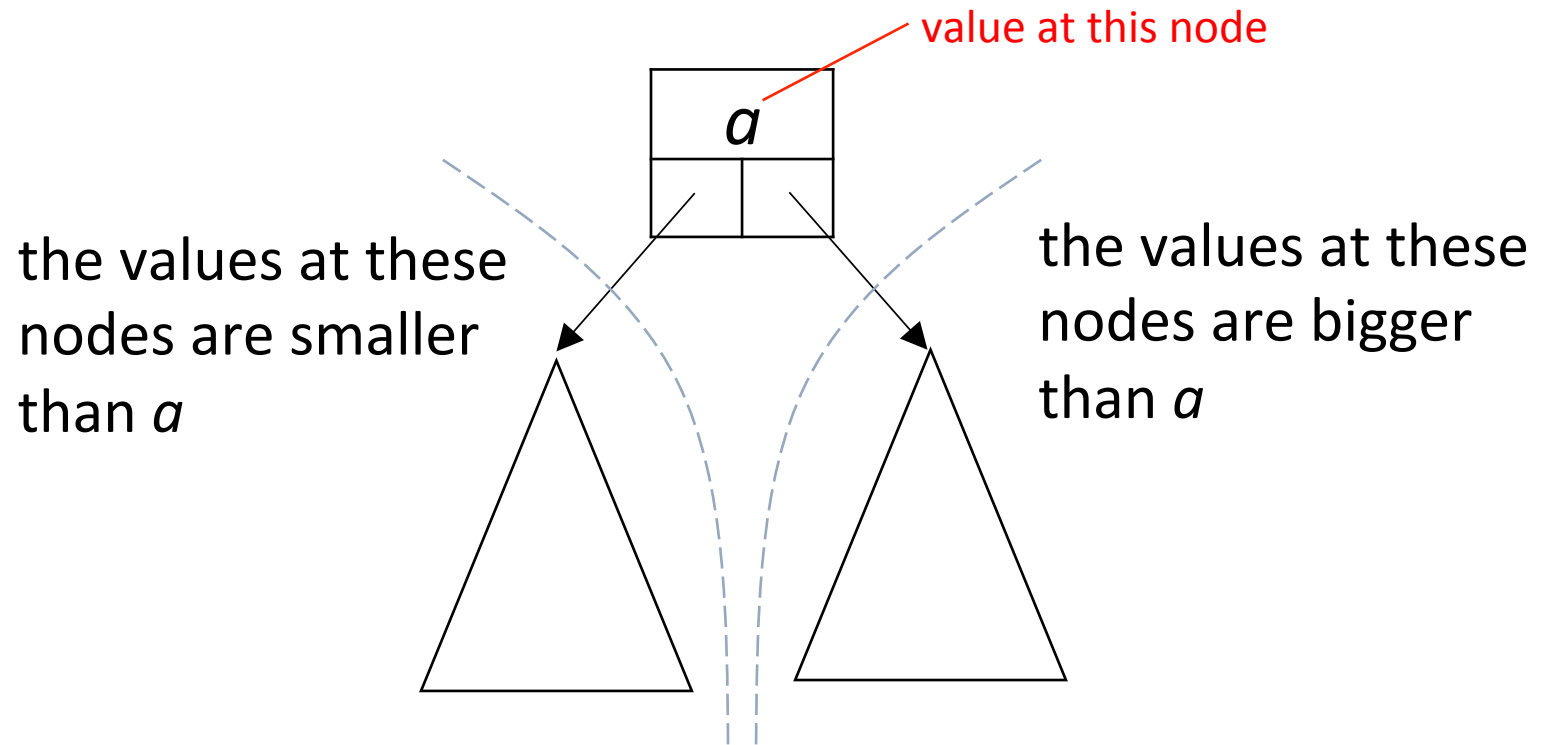
What can we say about the values in the nodes to the left of 8?

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

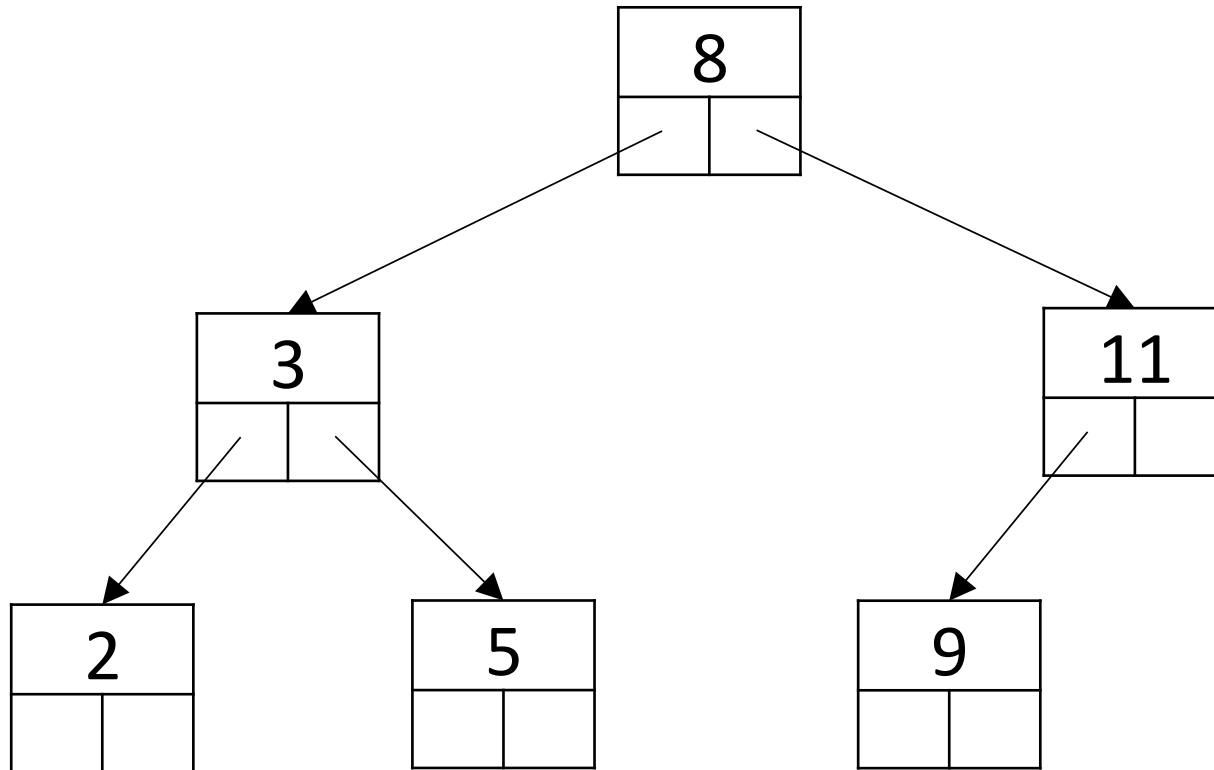


# Binary search tree (BST)

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

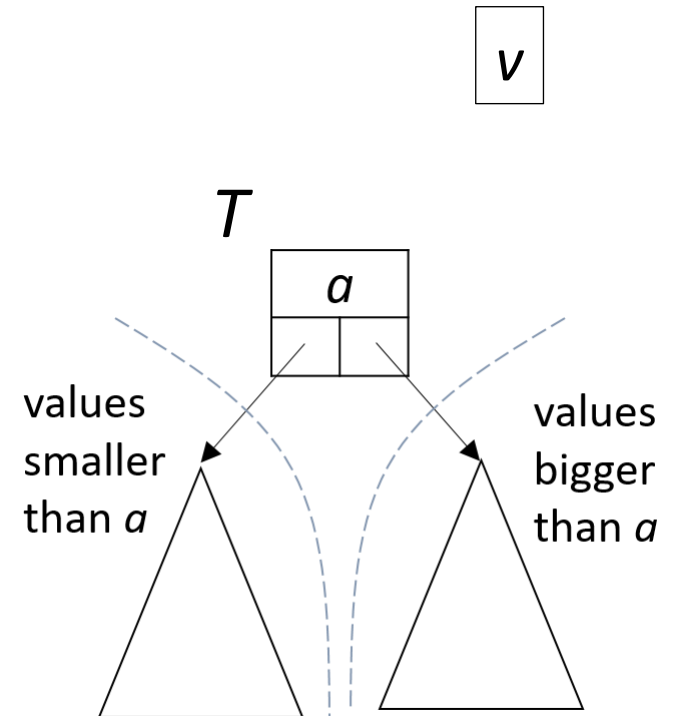


# Binary search tree: Example



# Searching a BST

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

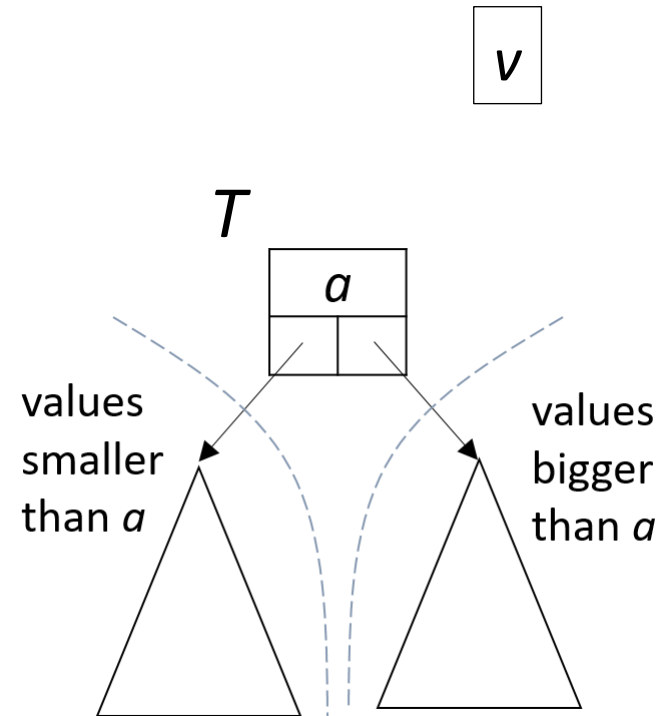


# Searching a BST

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

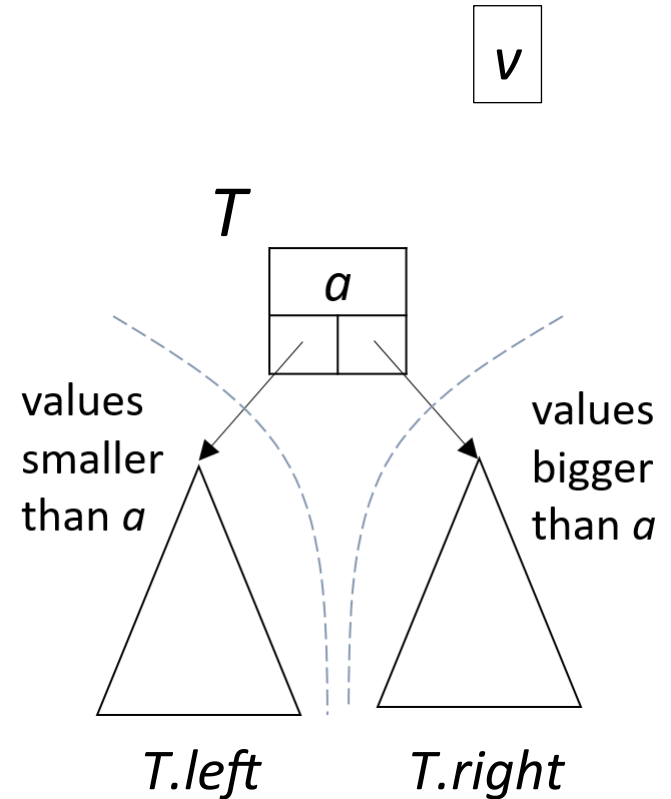
Idea: at each node with value  $a$ :

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



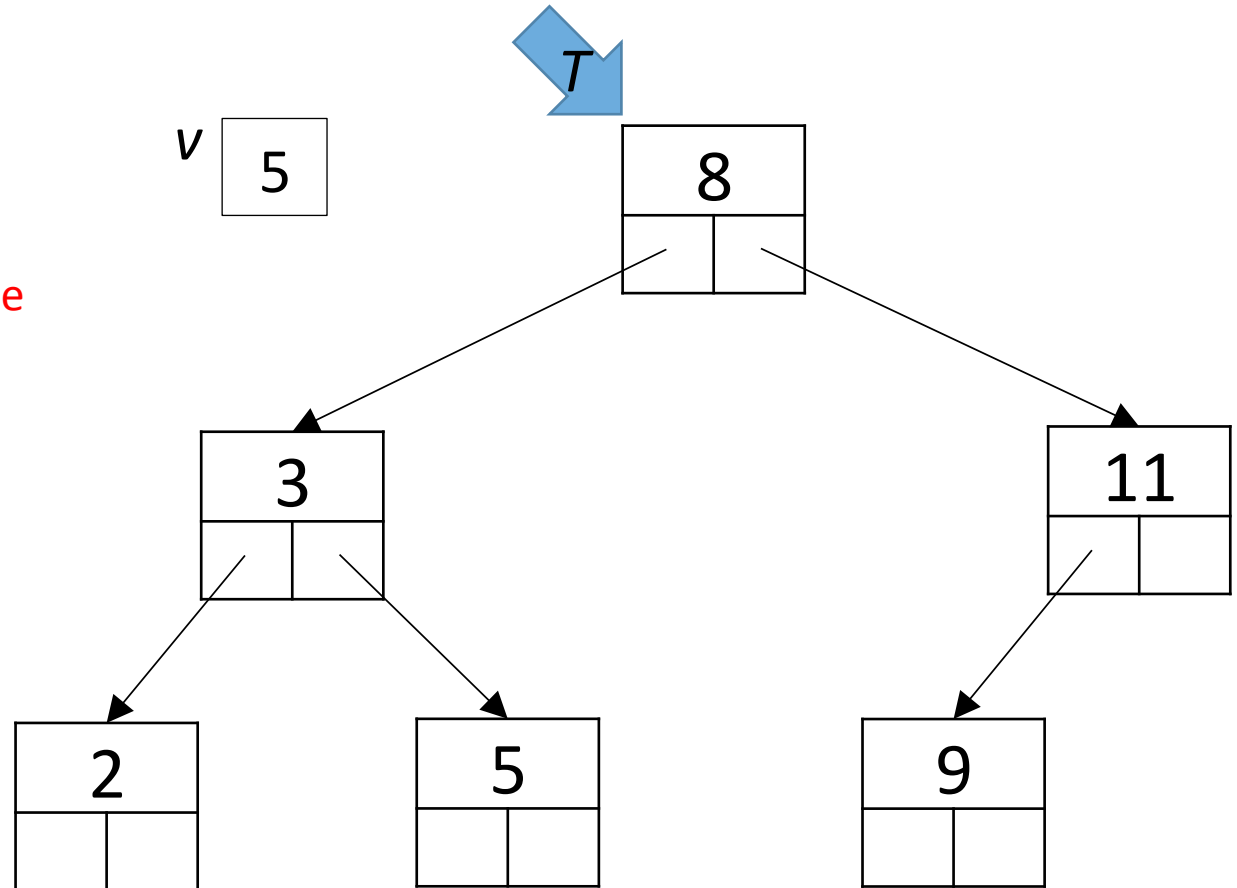
# Searching a BST

```
def search( $T, v$ ):  
    if  $T == \text{None}$ :  
        return False  
    if  $v == T.value$ :  
        return True  
    if  $v < T.value$ :  
        return search( $T.left, v$ )  
    else:  
        return search( $T.right, v$ )
```

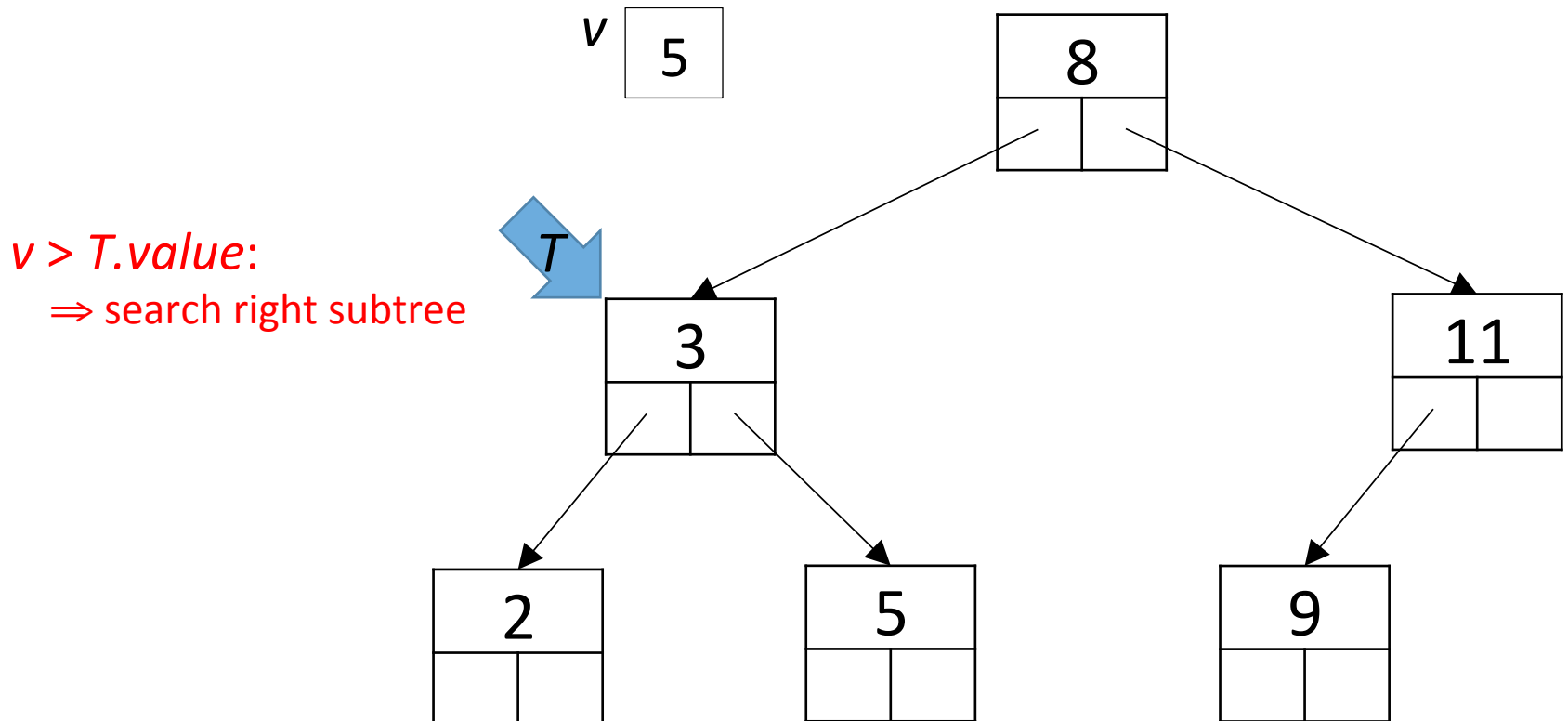


# Searching a BST: Example 1

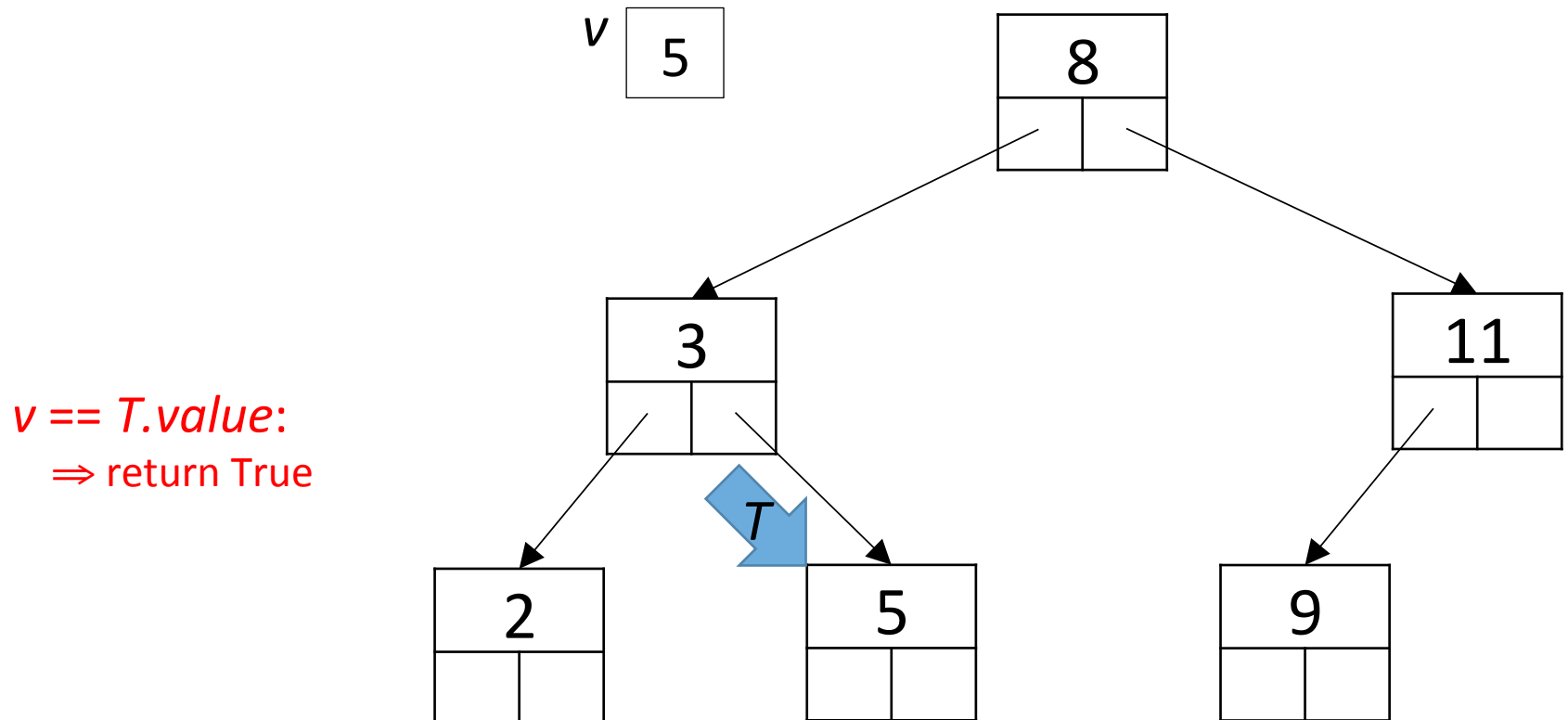
$v < T.value:$   
 $\Rightarrow$  search left subtree



# Searching a BST: Example 1



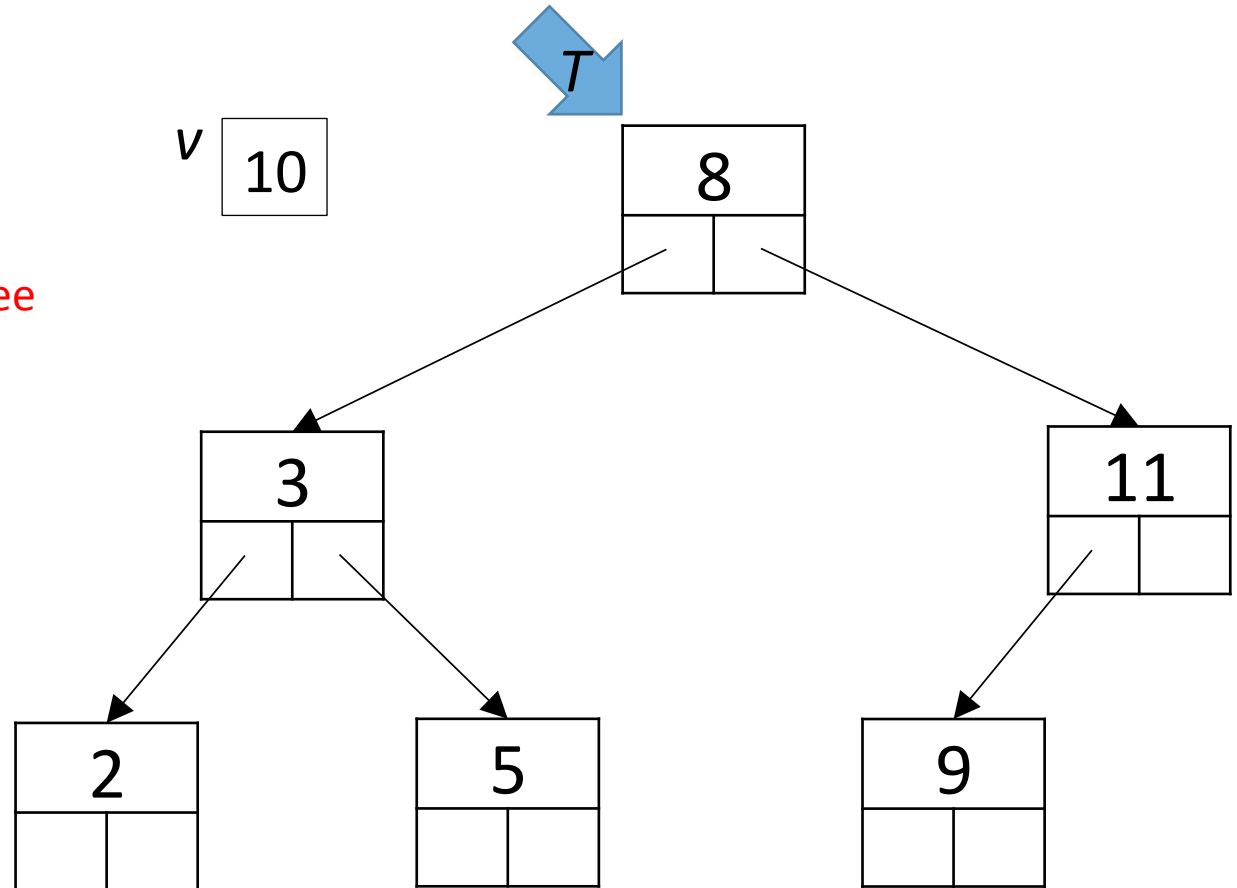
# Searching a BST: Example 1



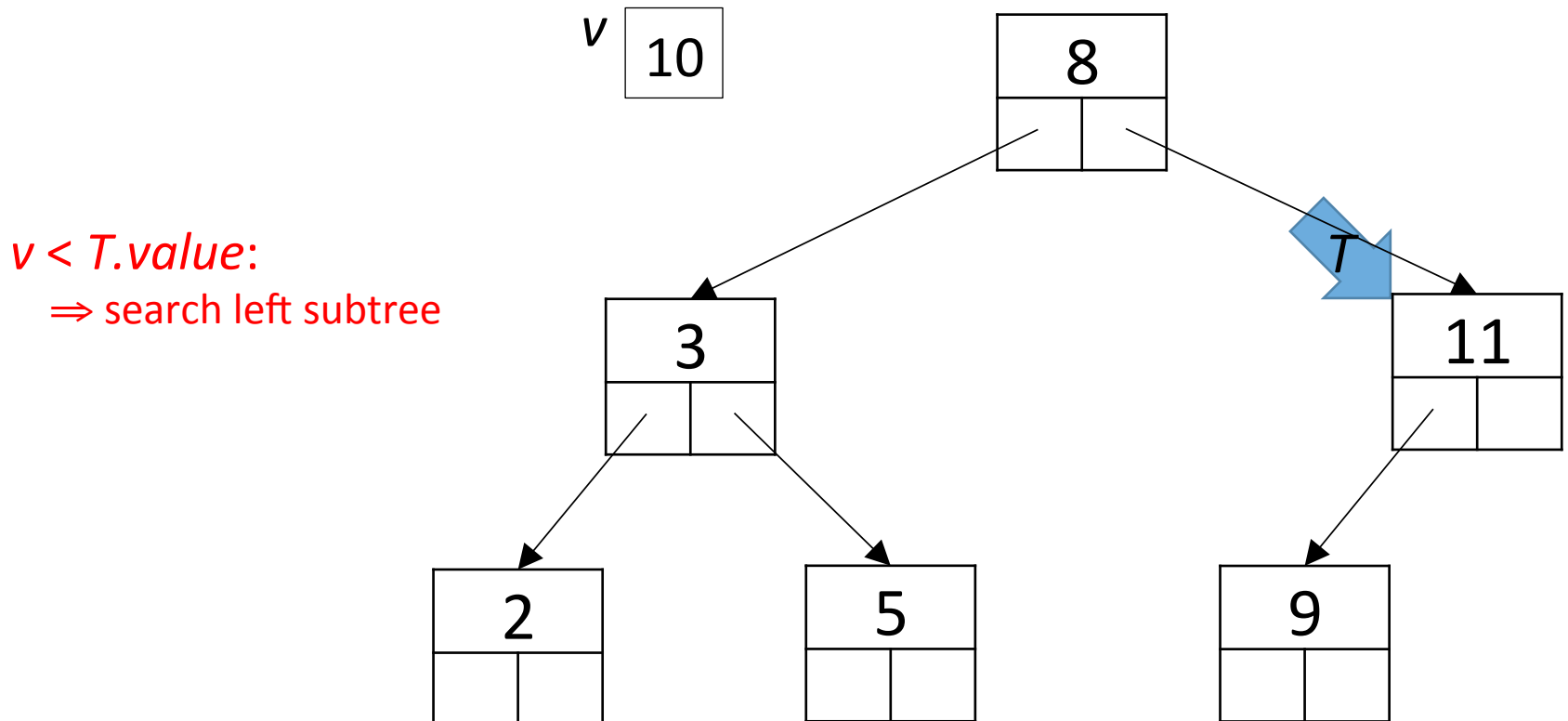


# Searching a BST: Example 2

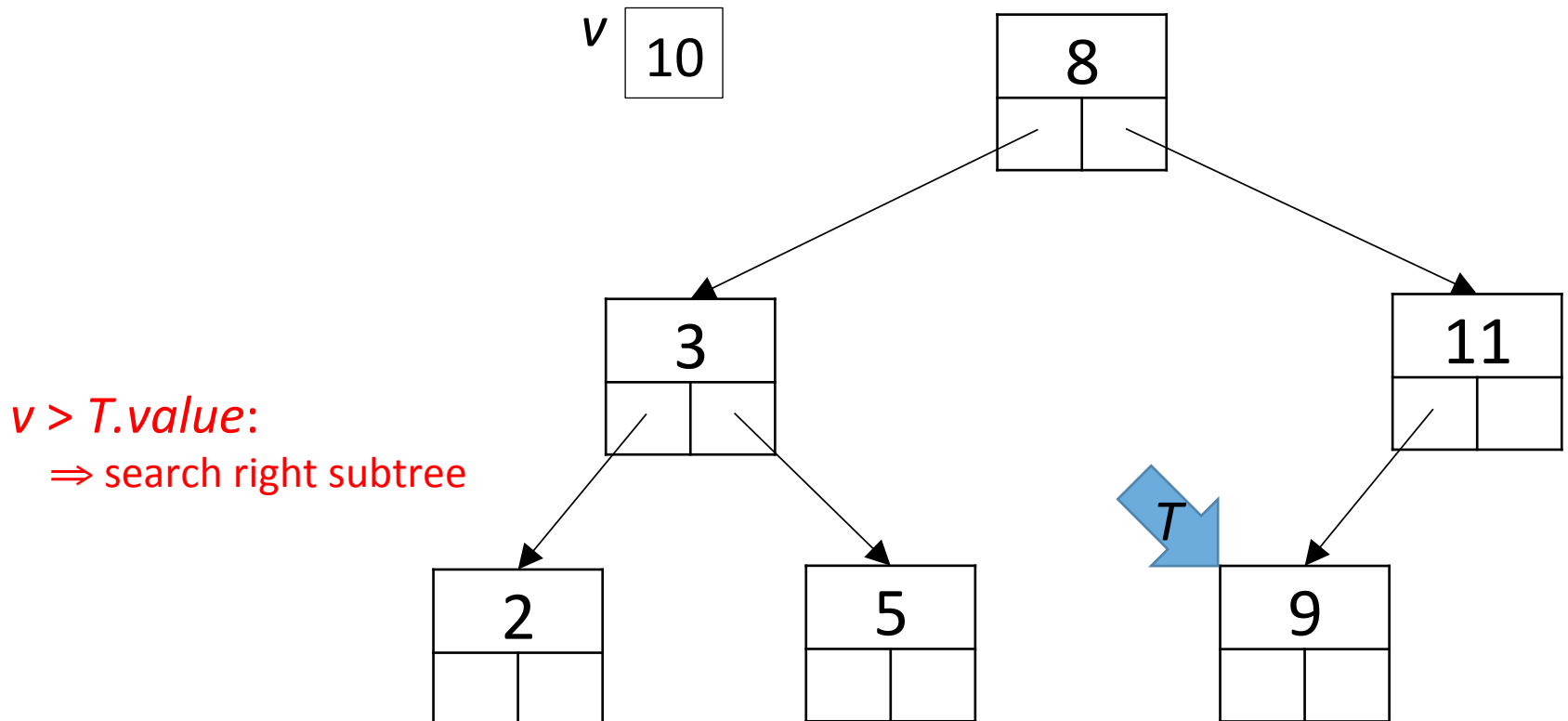
$v > T.value:$   
 $\Rightarrow$  search right subtree



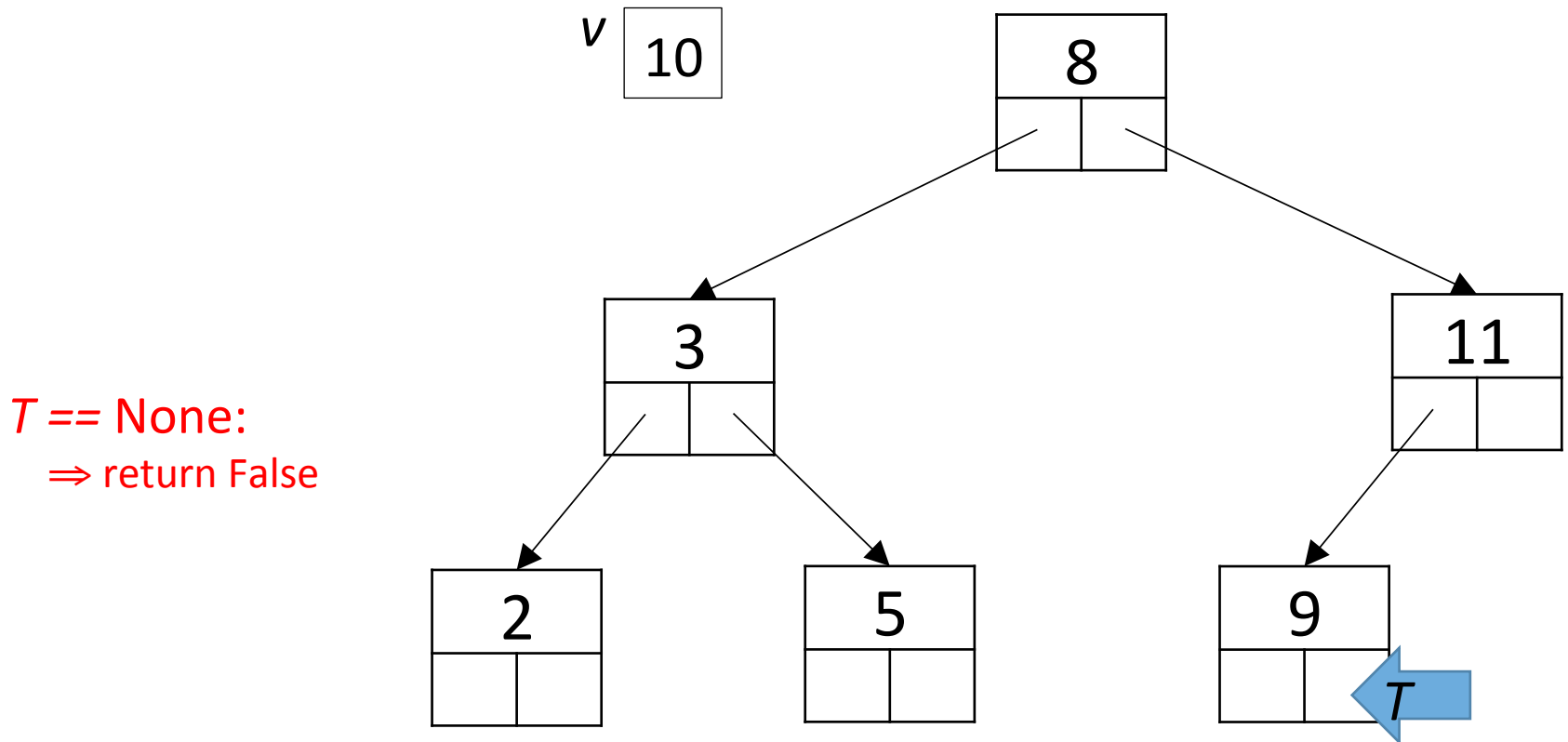
# Searching a BST: Example 2



# Searching a BST: Example 2



# Searching a BST: Example 2



# 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  $v < T.value$  : insert into  $T$ 's left subtree
  - if  $v == T.value$  : done
  - if  $v > T.value$  : insert into  $T$ 's right subtree

# Constructing a BST

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

# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

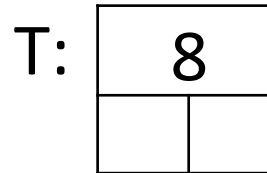
```
def insert(T, v): (v = 8, T = None)  
→ if T == None:  
    return Node(v)  
if v < T.value:  
    T.left = insert(T.left, v)  
elif v > T.value:  
    T.right = insert(T.right, v)  
return T
```

*T*: None

# 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.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```

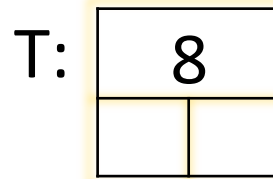




# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

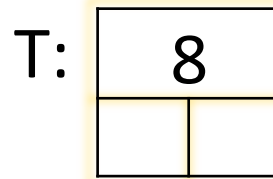
```
def insert(T, v): (v = 3, T.value = 8)
    if T == None:
        return Node(v)
    → if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

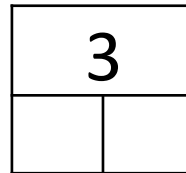
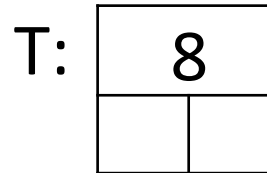
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        → T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

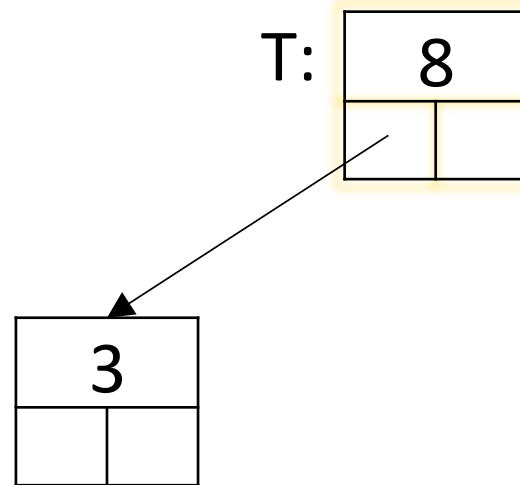
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        → T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

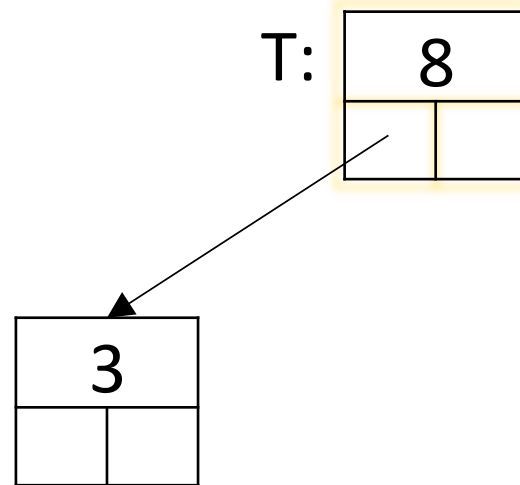
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        → T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example



Sequence of values: 8 3 **11** 2 9 5

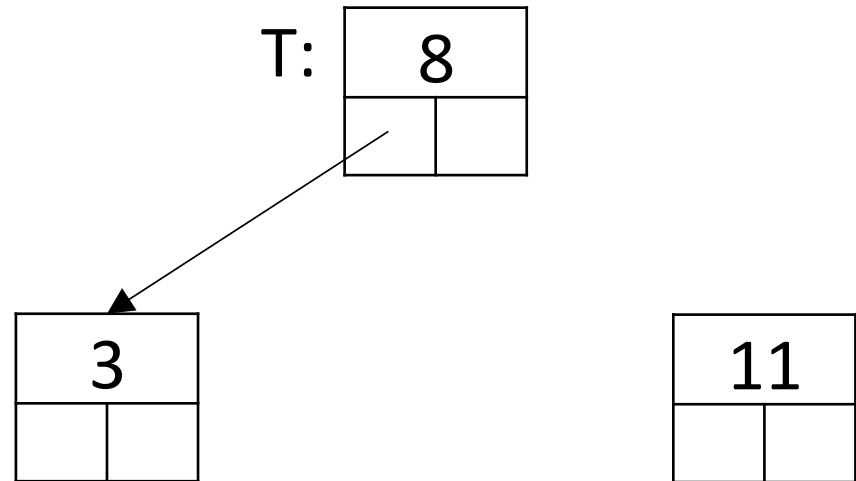
```
def insert(T, v): (v = 11, T.value = 8)  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 **11** 2 9 5

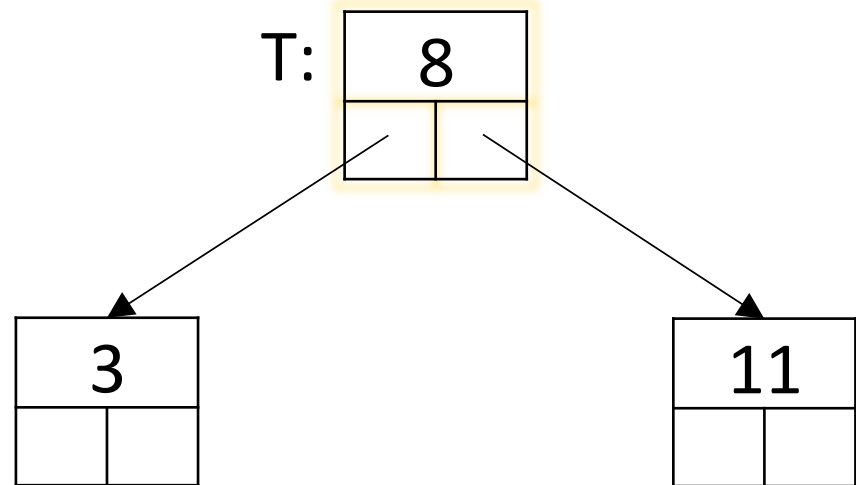
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:   
     T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 **11** 2 9 5

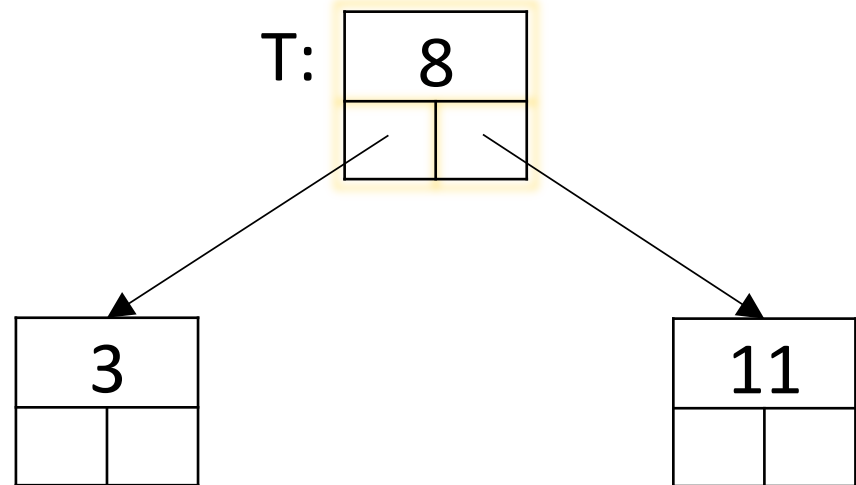
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        → T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 **2** 9 5

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

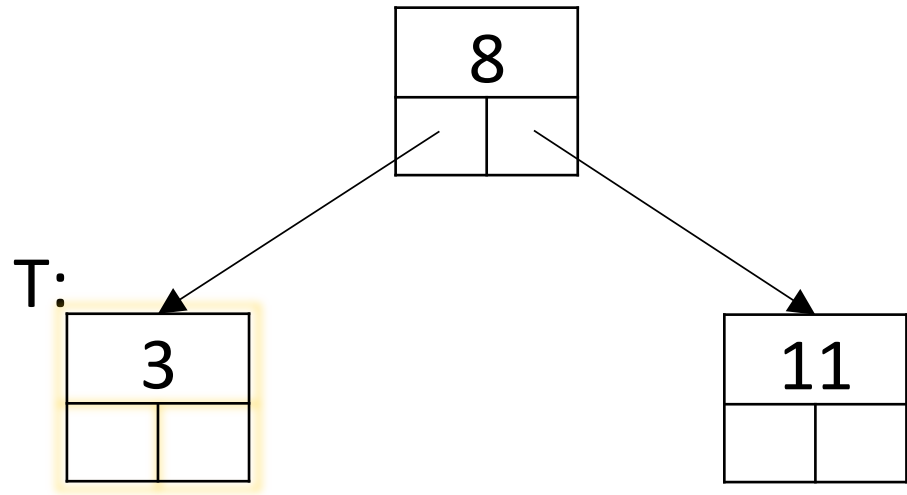




# Constructing a BST: Example

Sequence of values: 8 3 11 **2** 9 5

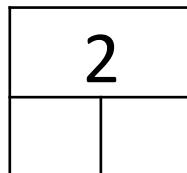
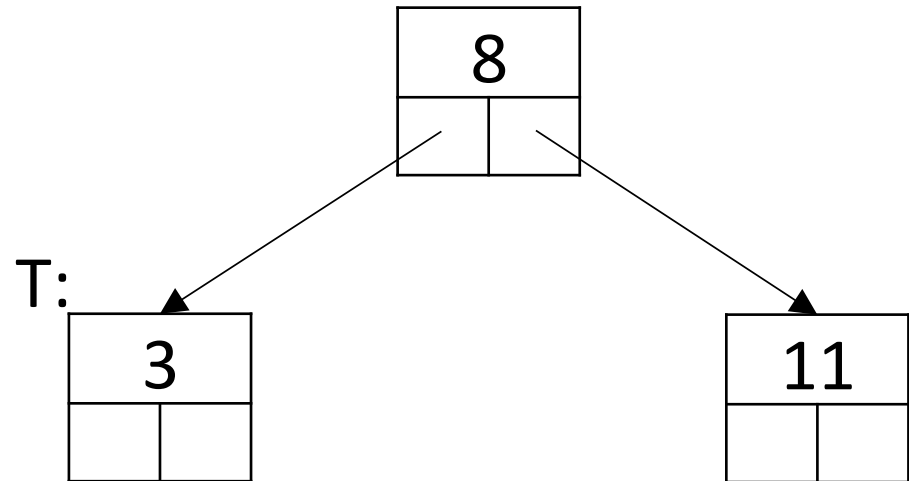
```
def insert(T, v): (v = 2, T.value = 3)  
    if T == None:  
        return Node(v)  
    → if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 **2** 9 5

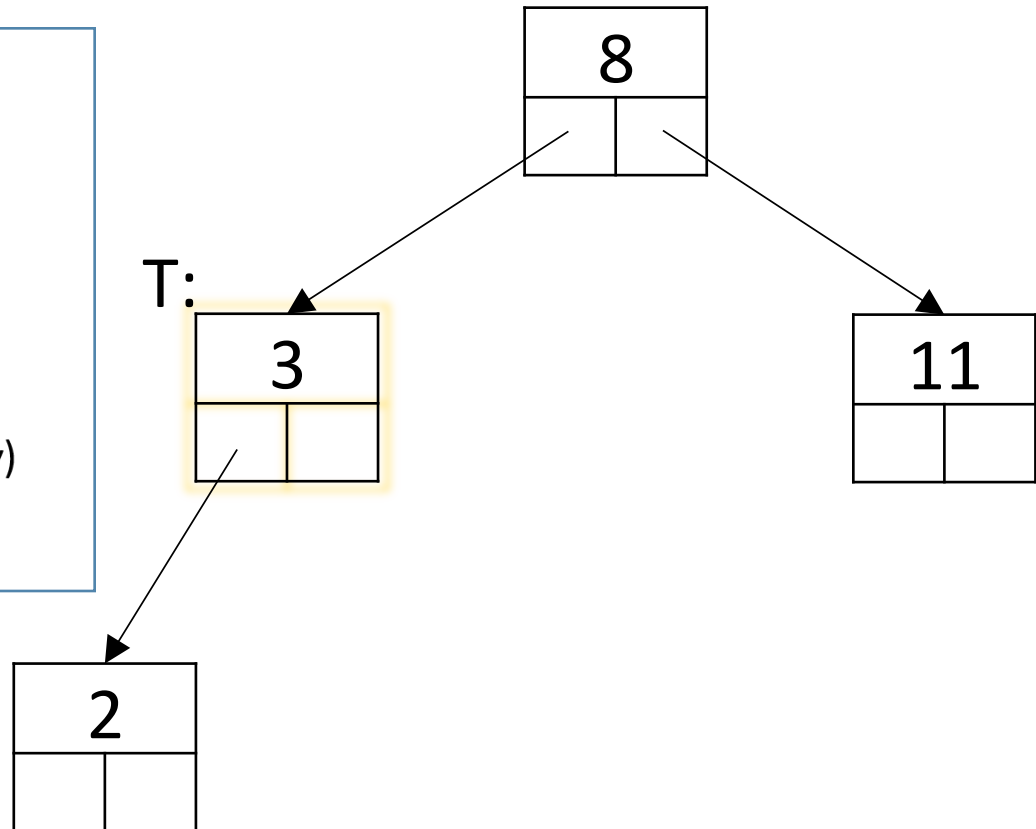
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 **2** 9 5

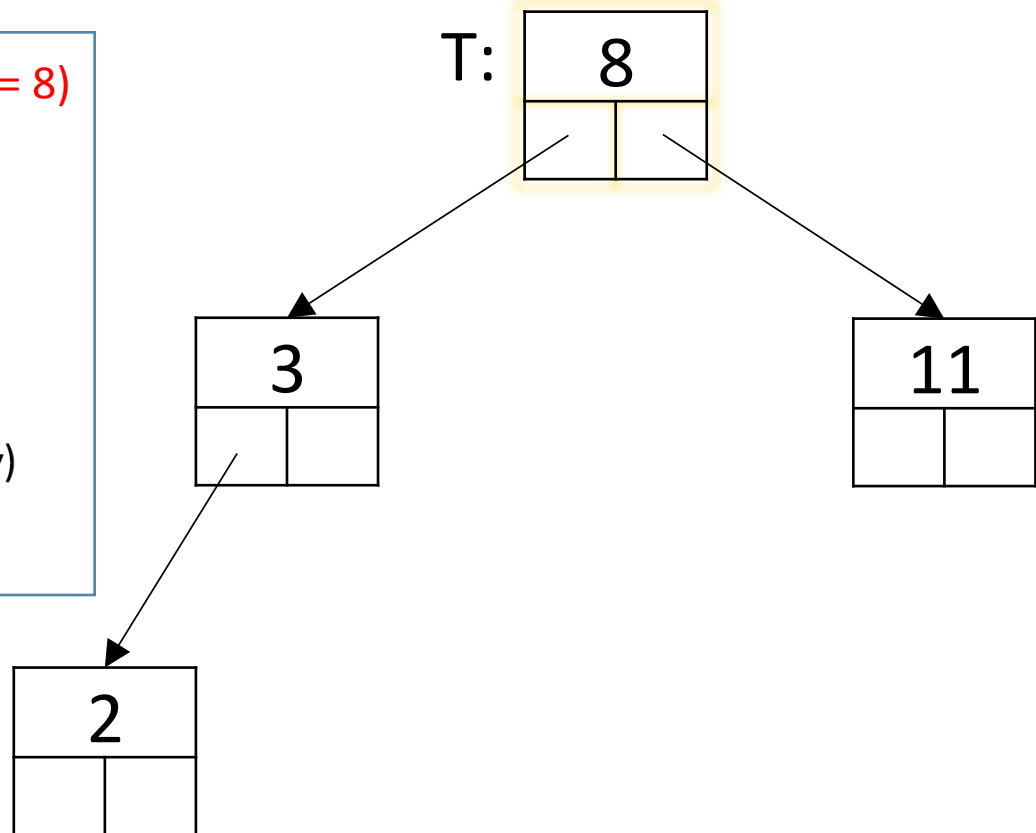
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

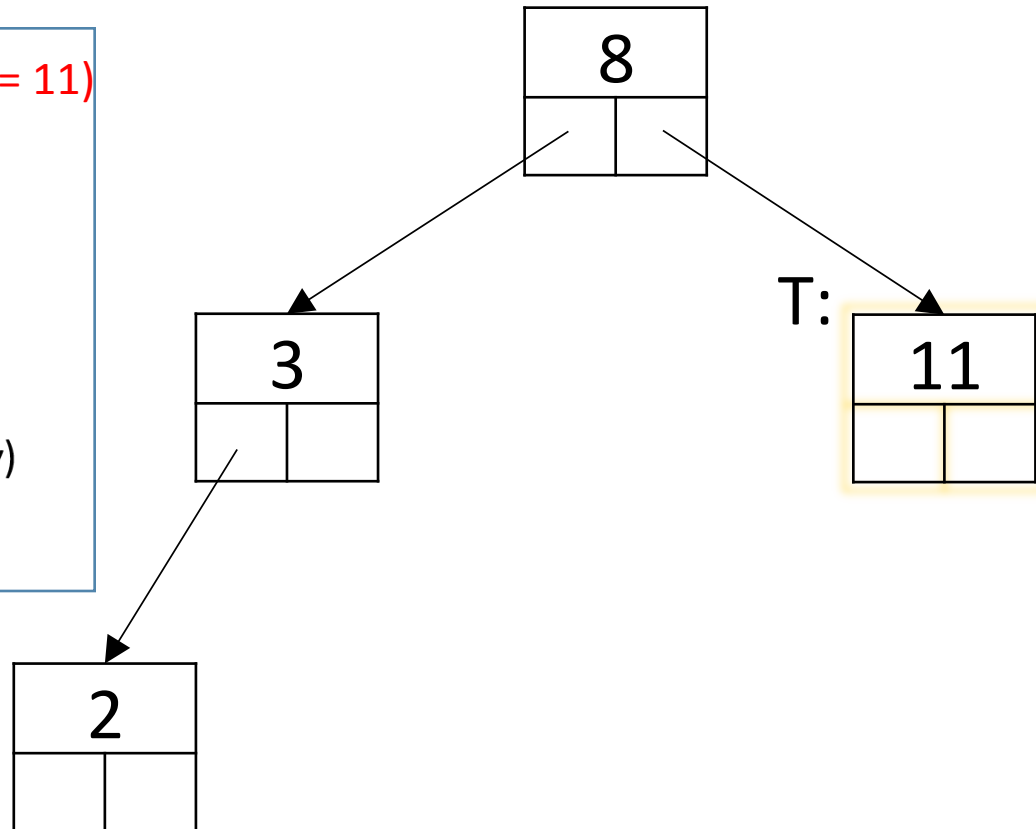
```
def insert(T, v): (v = 9, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 **9** 5

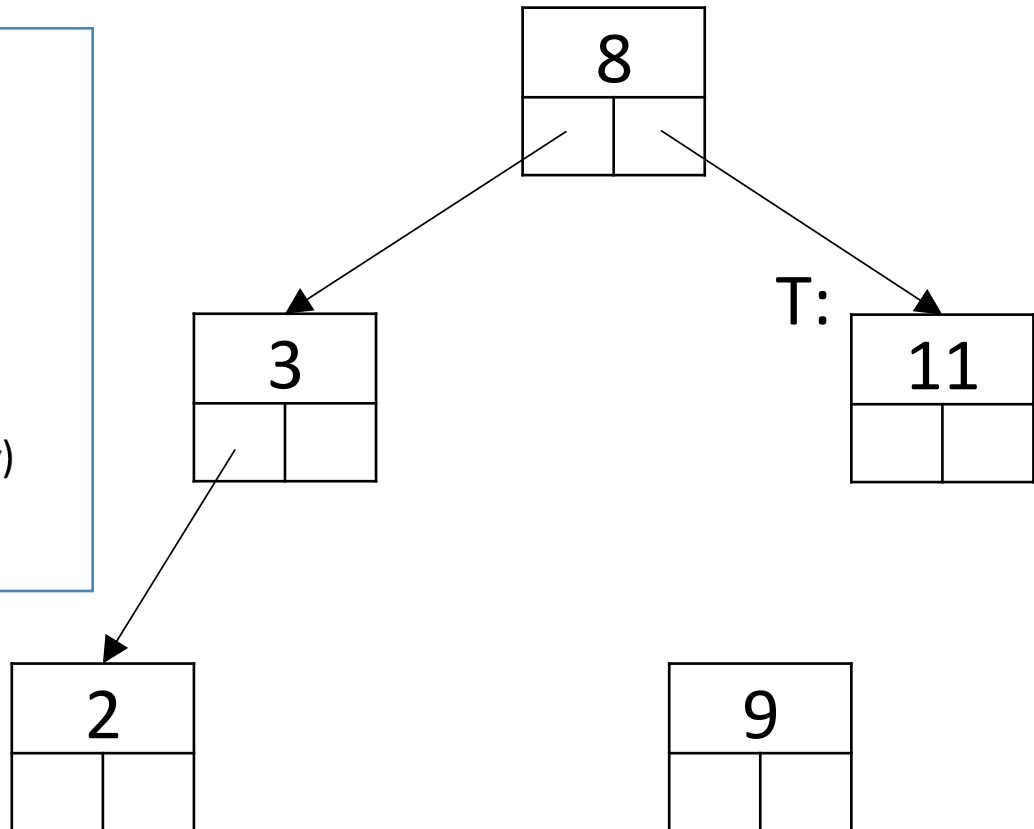
```
def insert(T, v): (v = 9, T.value = 11)  
    if T == None:  
        return Node(v)  
    → if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 **9** 5

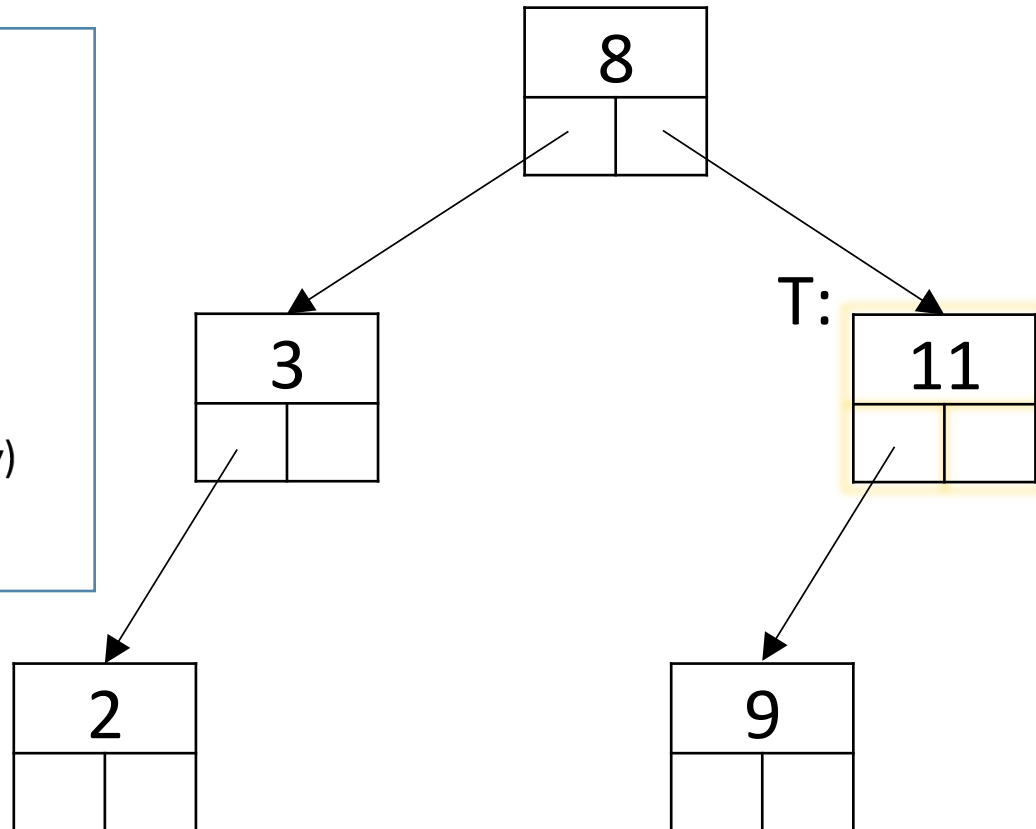
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        → T.left = insert(T.left, v) ↙  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 **9** 5

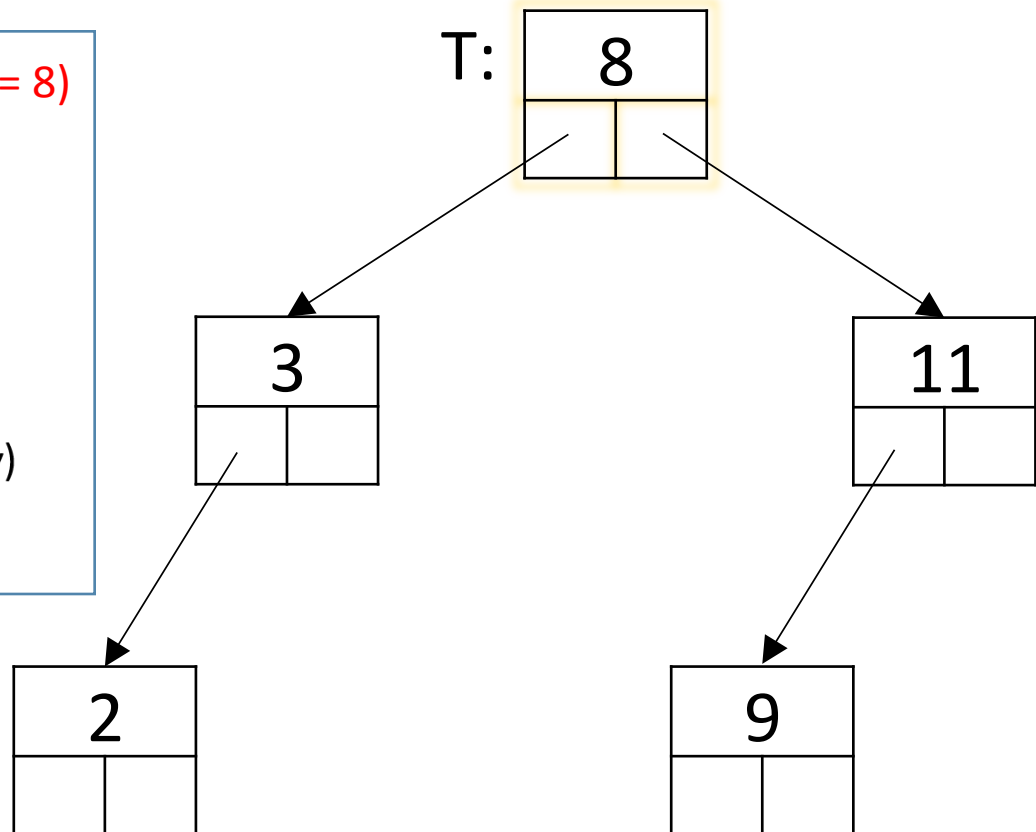
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# 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.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```

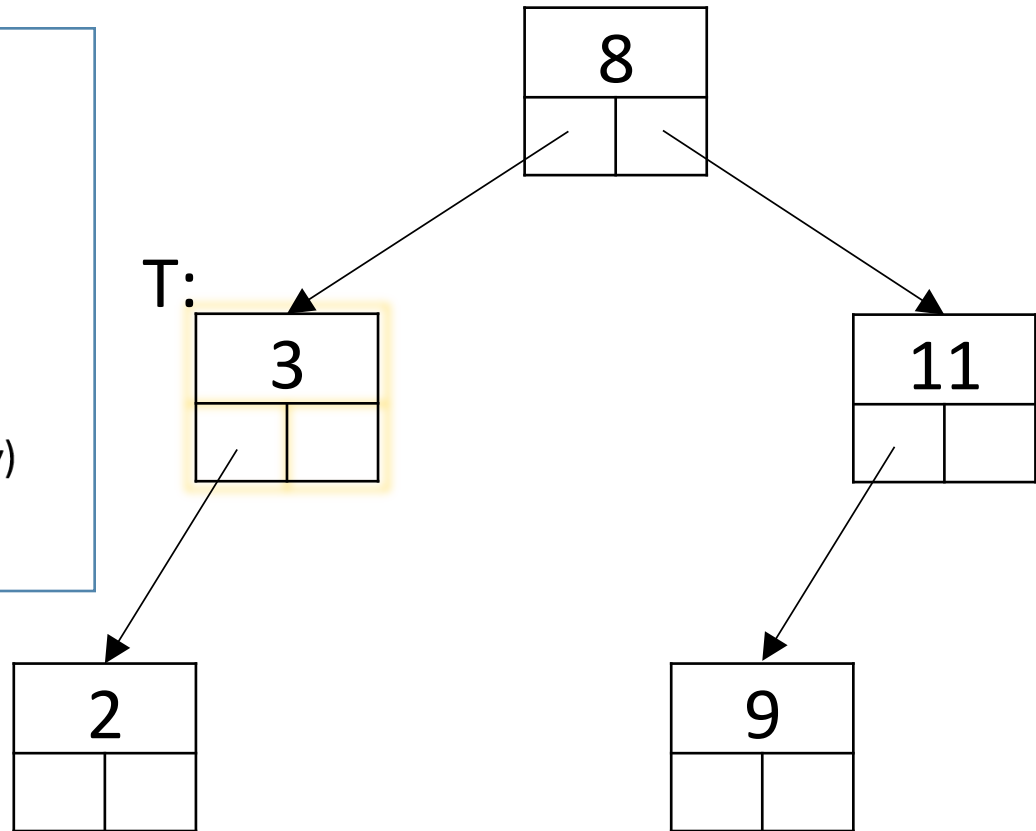




# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 **5**

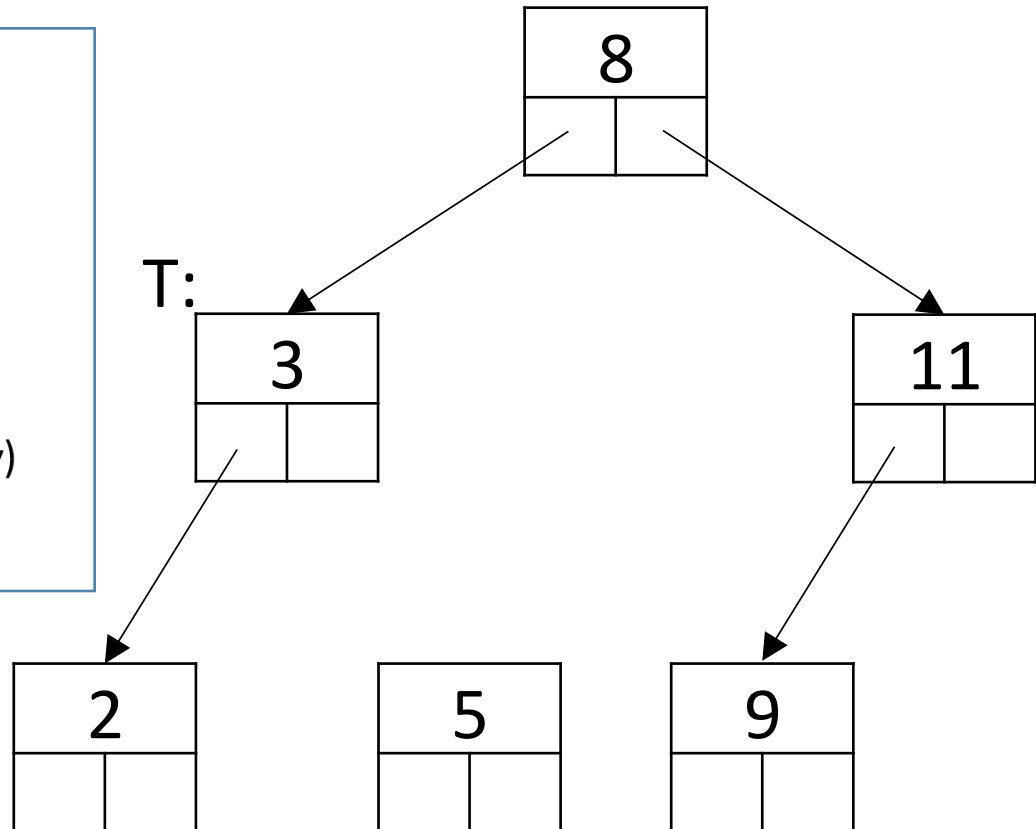
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

Sequence of values: 8 3 11 2 9 **5**

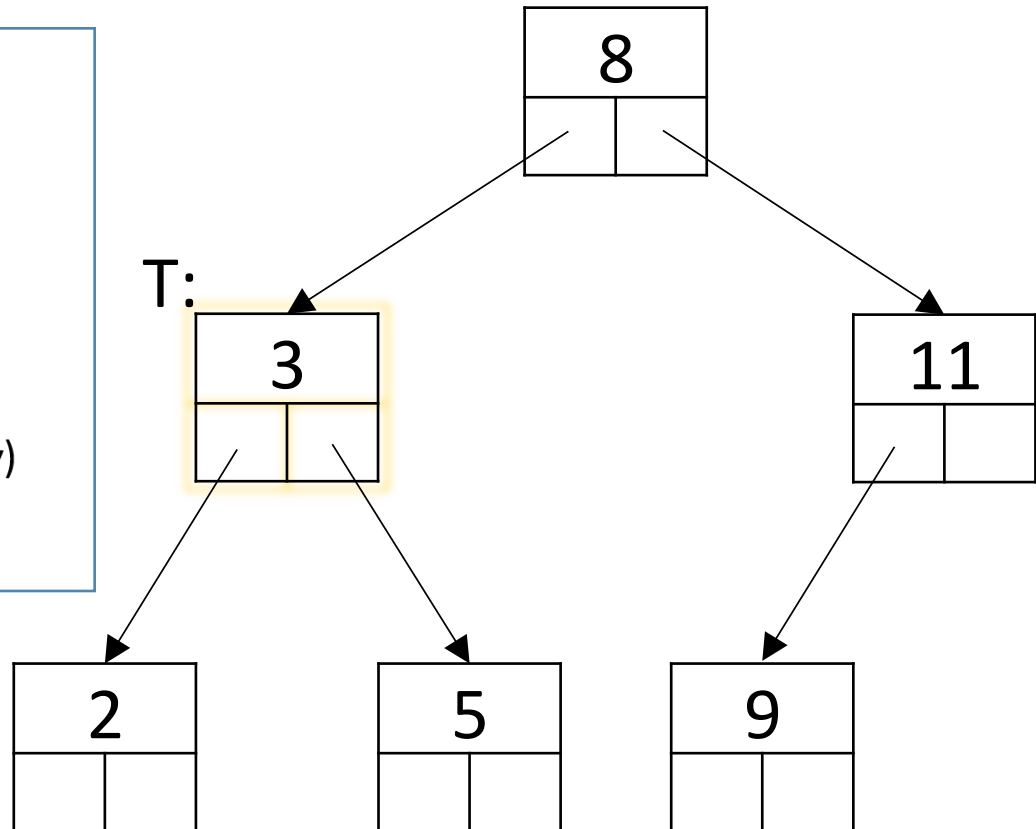
```
def insert(T, v):  
    if T == None:  
        return Node(v)  
    if v < T.value:  
        T.left = insert(T.left, v)  
    elif v > T.value:  
        T.right = insert(T.right, v)  
    return T
```



# Constructing a BST: Example

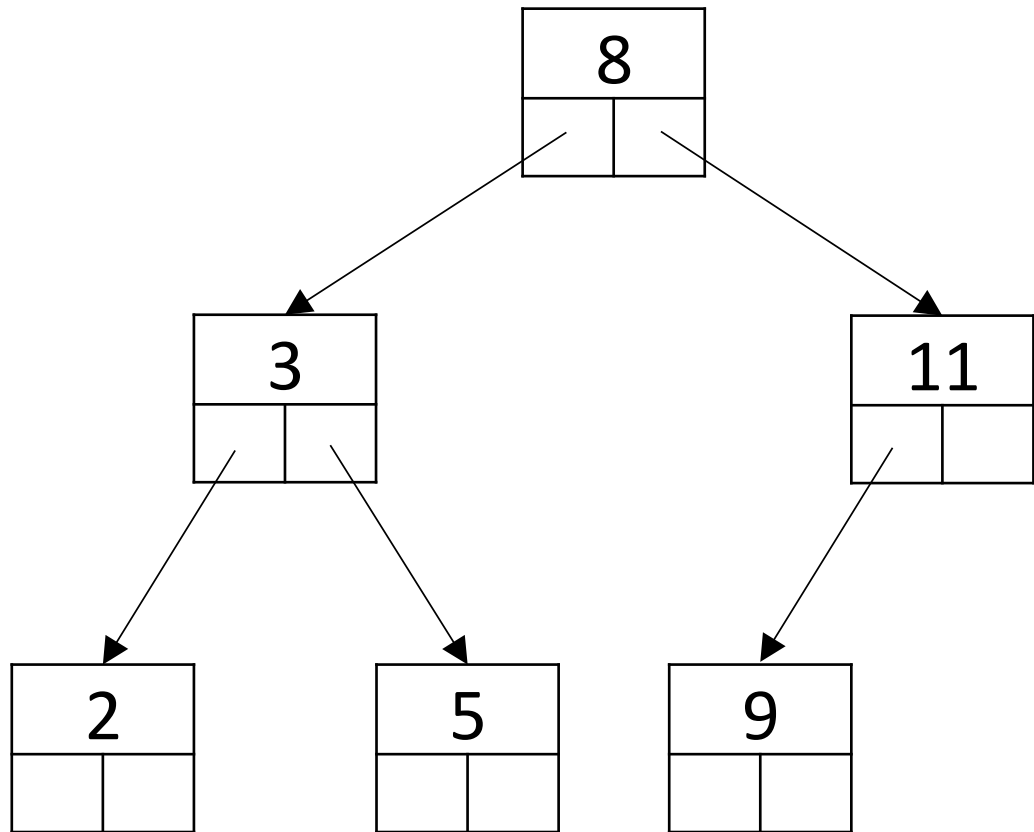
Sequence of values: 8 3 11 2 9 **5**

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

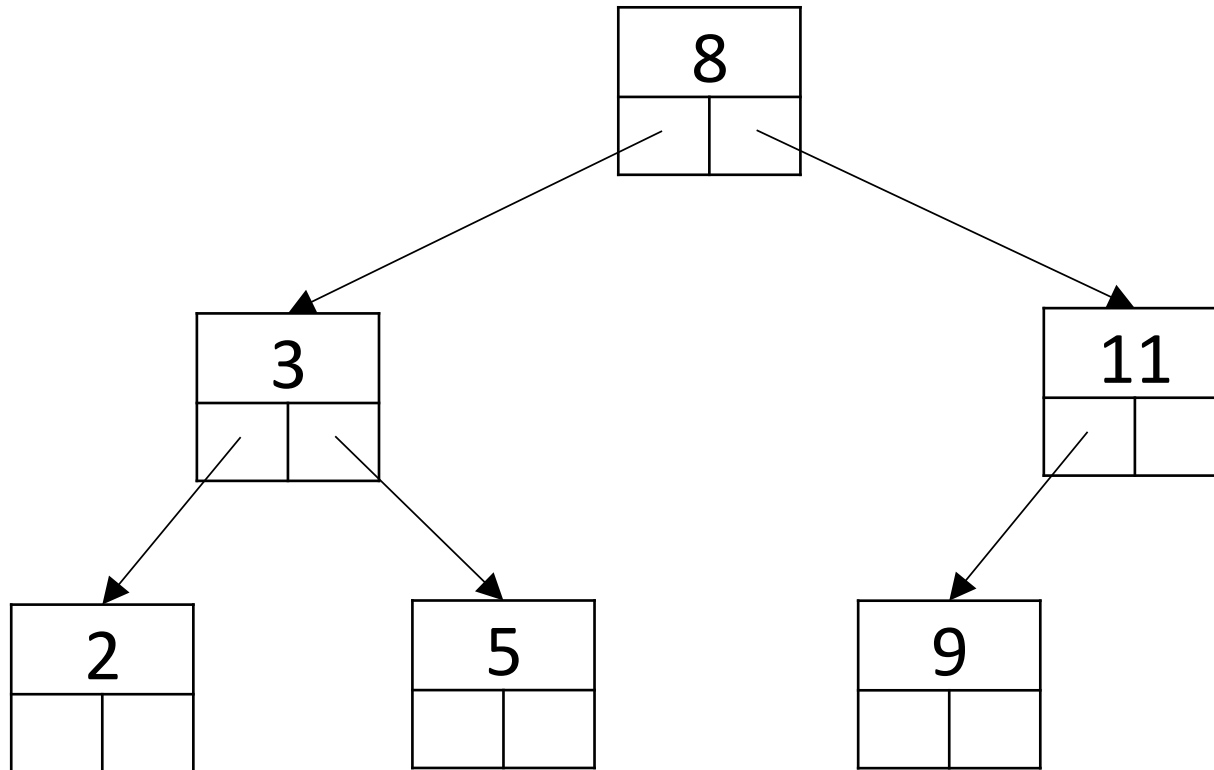


# Constructing a BST: Example

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.value:
```

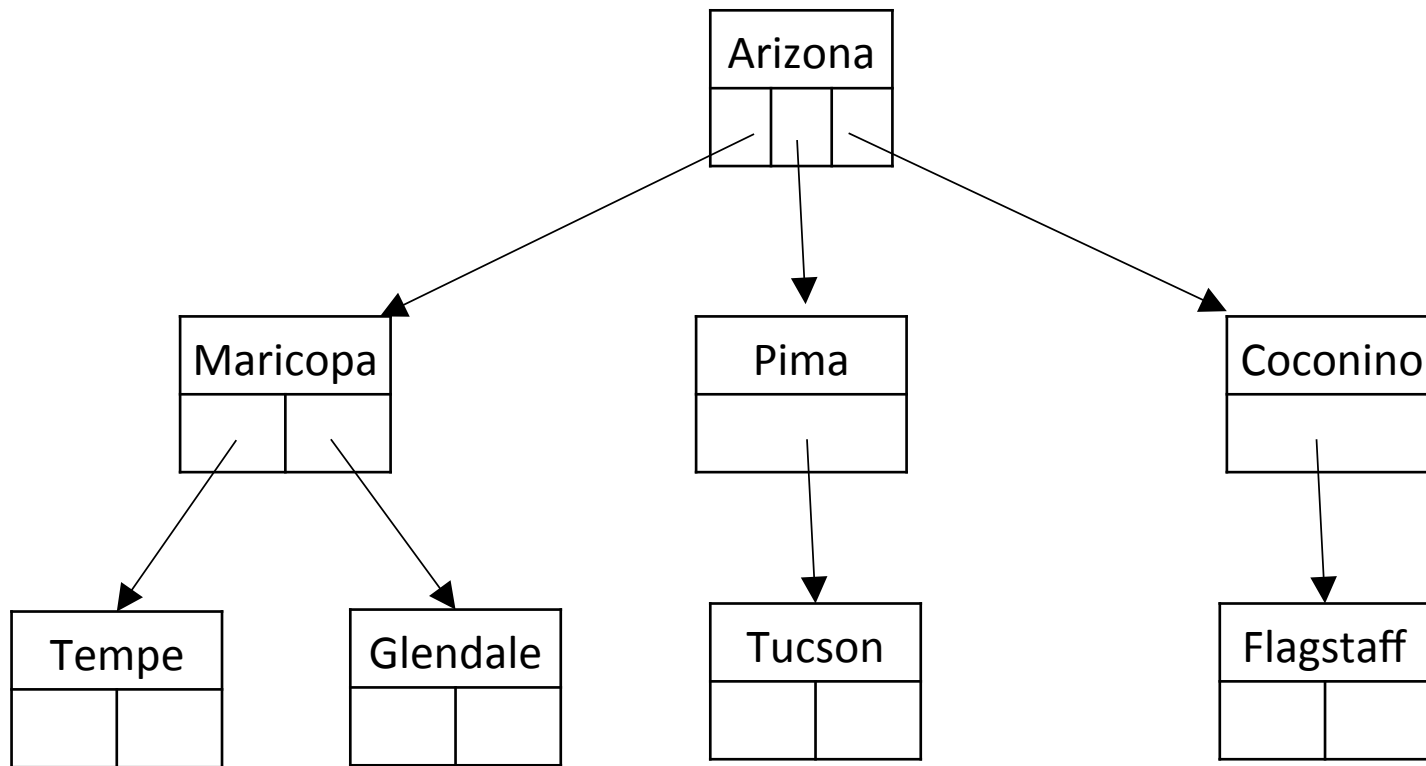
```
        T.left = insert(T.left, v)
```

```
    elif v > T.value:
```

```
        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?

# tree traversals



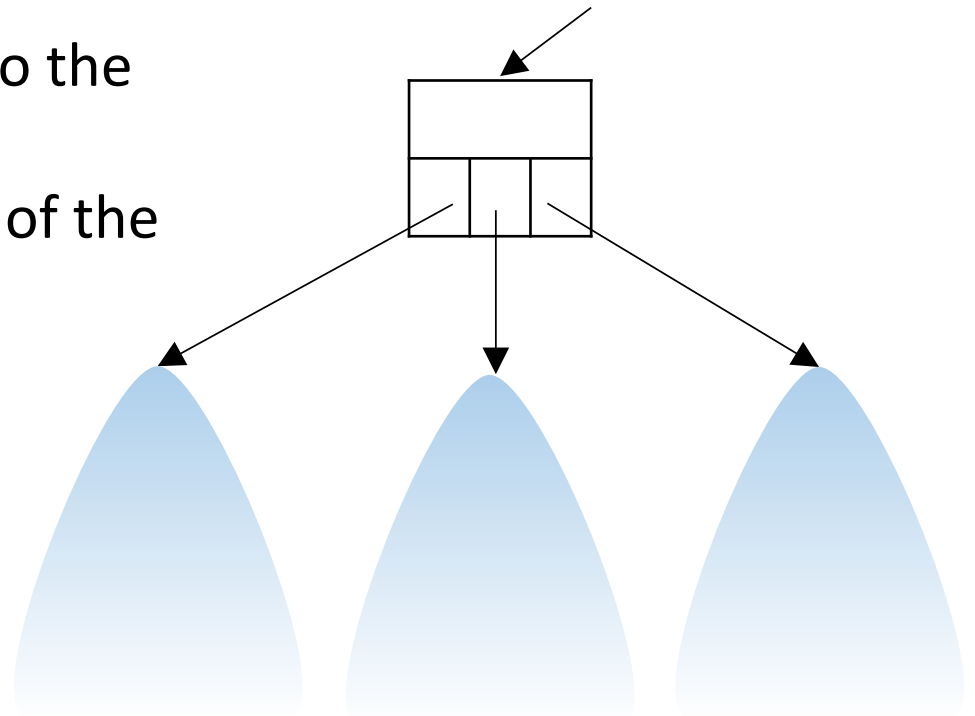
# Tree traversals

- 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 (n-ary)

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

# Tree traversals (binary)

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:

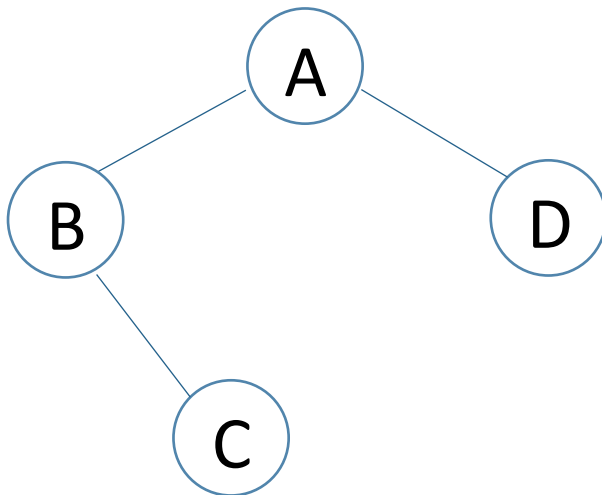
*(where's the base case?)*

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

A

B

C

D

## Trace

Call preorder(A)

Visit

Left (call preorder(B))

Visit

Left – return immediately

Right (call preorder(C))

Visit

Left – return

Right – return

Right(D)

Visit

Left – return

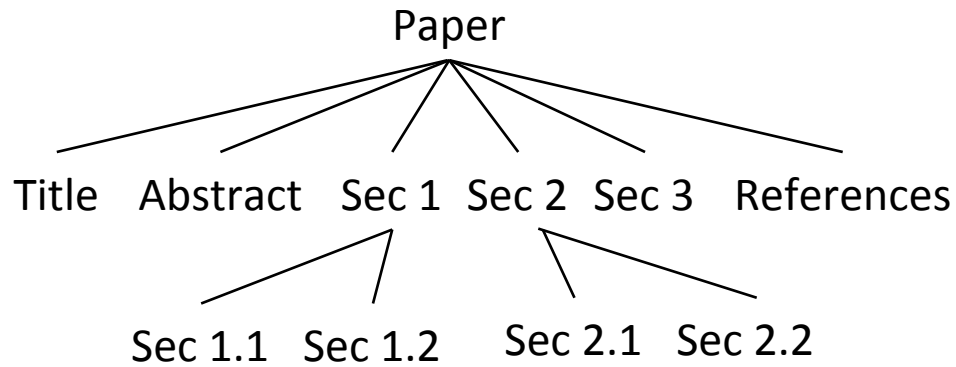
Right – return

# Preorder traversal (n-ary)

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

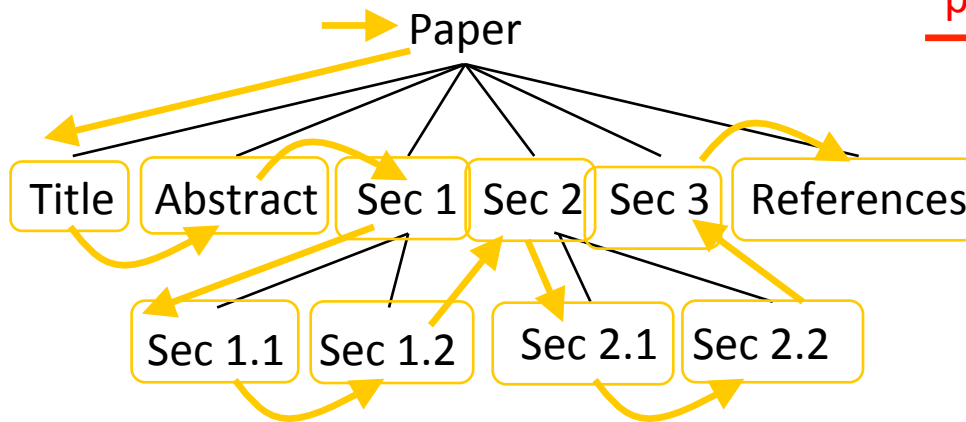
*(where's the base case?)*

# Preorder traversal: Example





# Preorder traversal: Example



preorder →

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

# Inorder traversal

Algorithm:

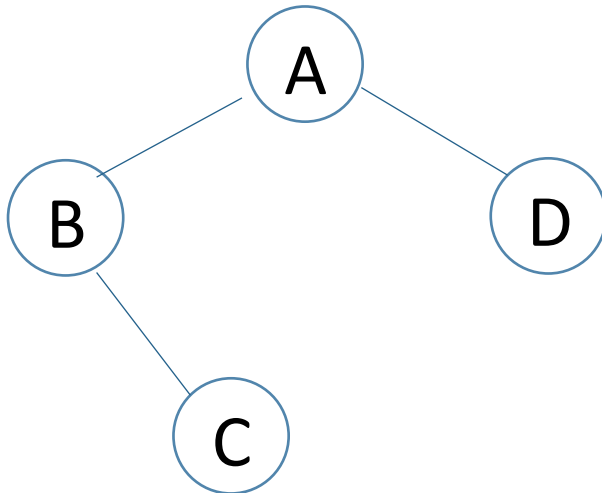
*(where's the base case?)*

Recurse on node's Left subtree

Visit node

Recurse on node's Right subtree

Ex:



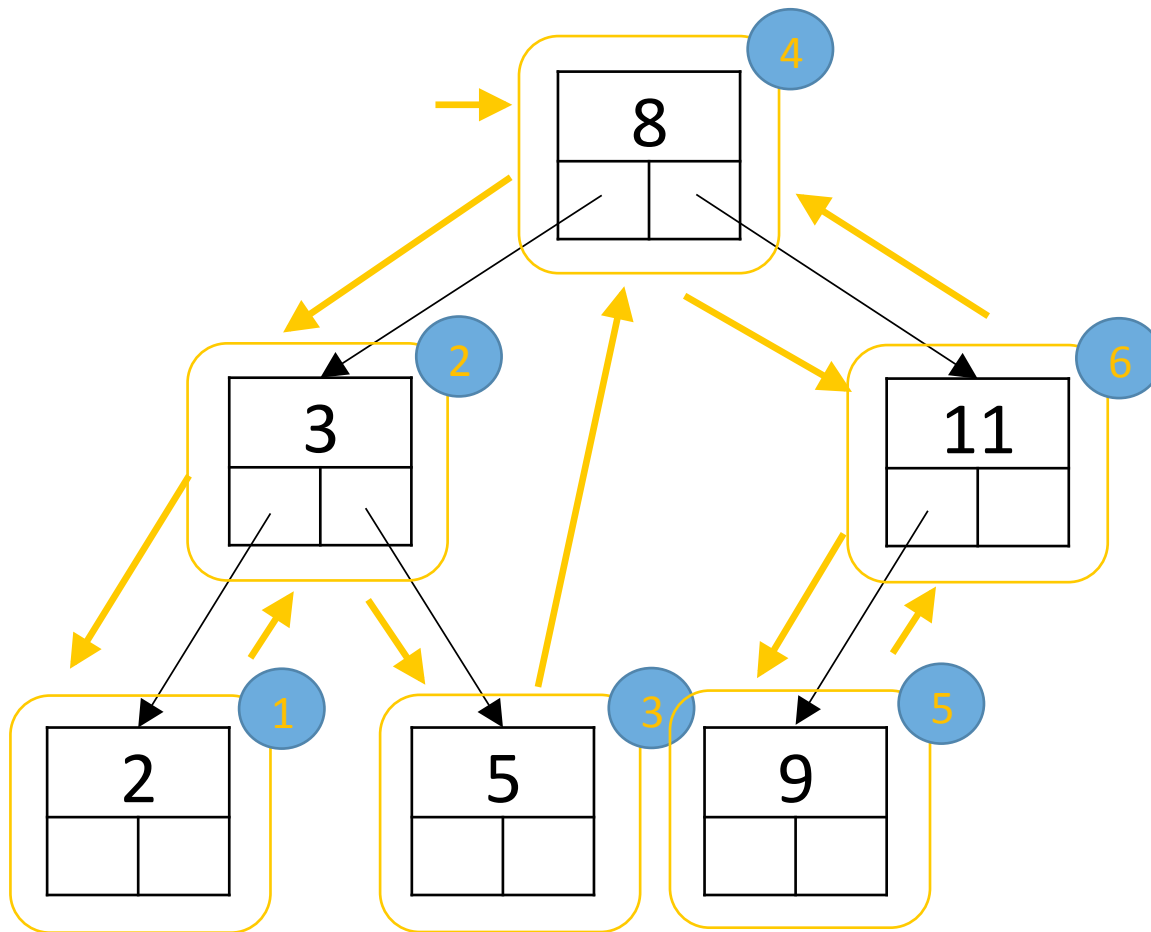
B C A D

# 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



*i* print order

# Postorder traversal

Algorithm:

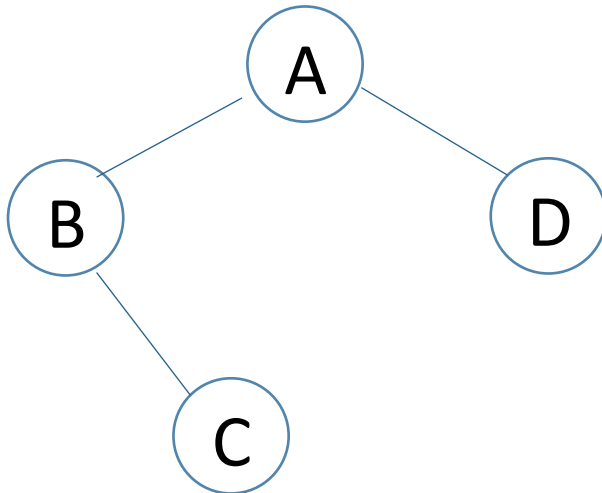
*(where's the base case?)*

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:  $x = 3, y = 2, n = 4$

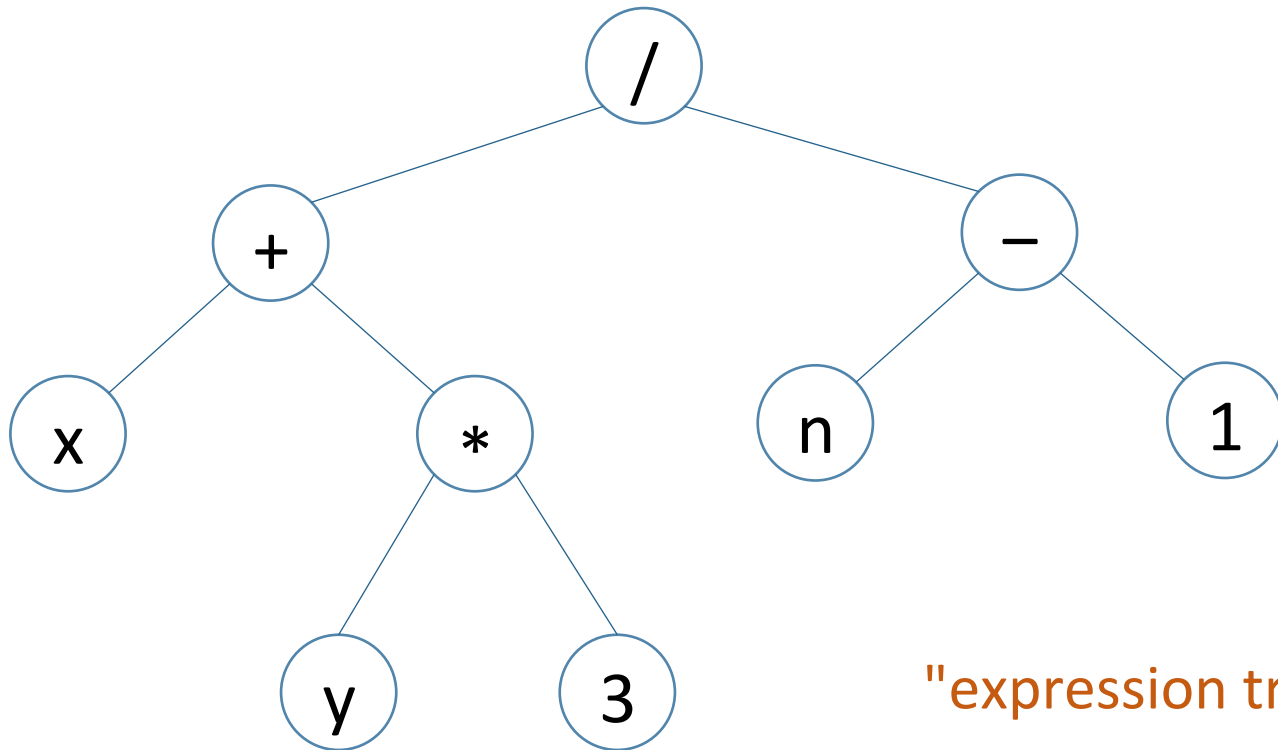
Solution:

- 1) Convert the expression to post-fix notation
  - use an auxiliary stack
  - use the algorithm covered in section
- 2) Represent the expression as a binary tree
  - use a postorder traversal of the tree

# Postorder traversal: Example

Evaluate:  $(x + y * 3) / (n - 1)$

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



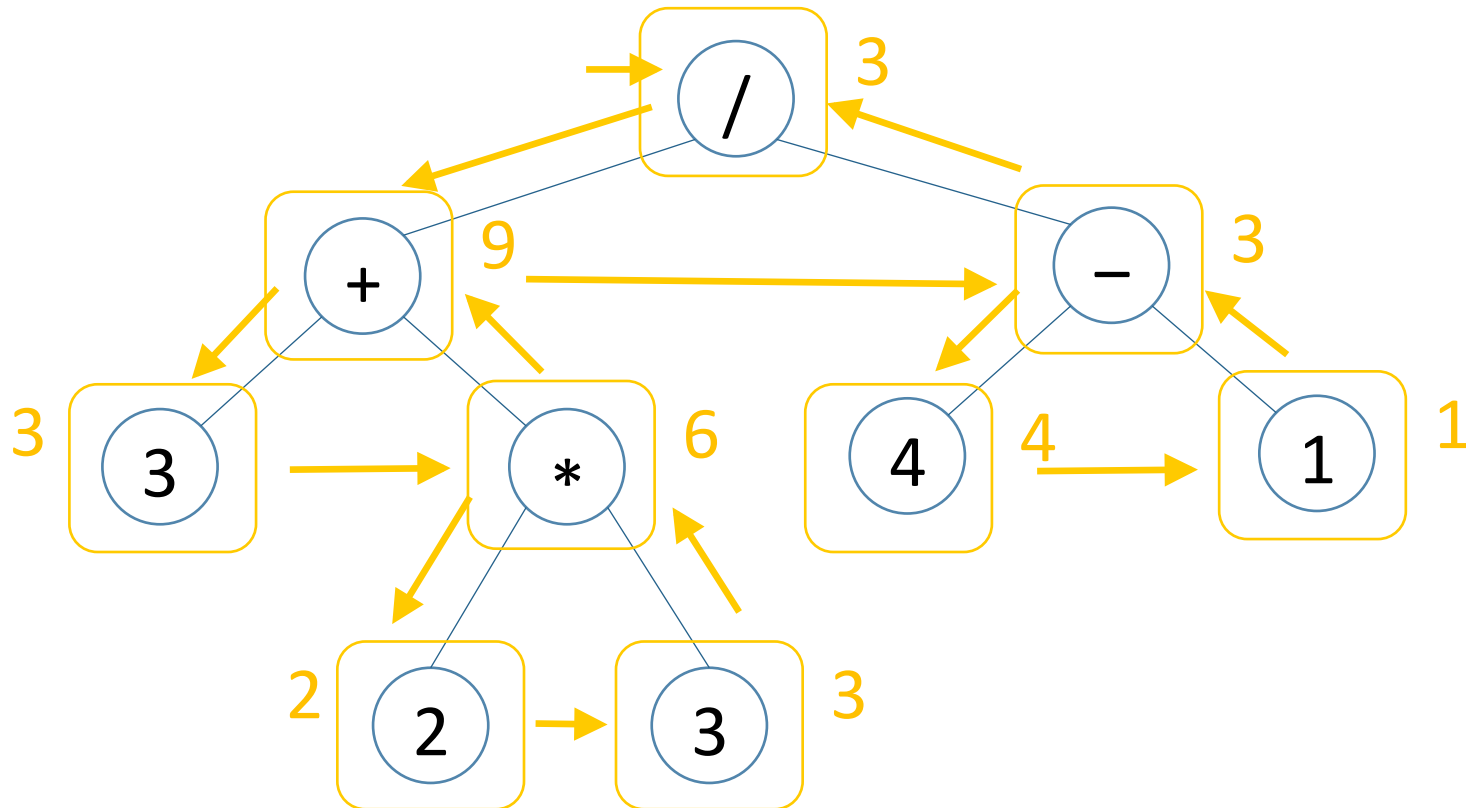
"expression tree"



# Postorder traversal: Example

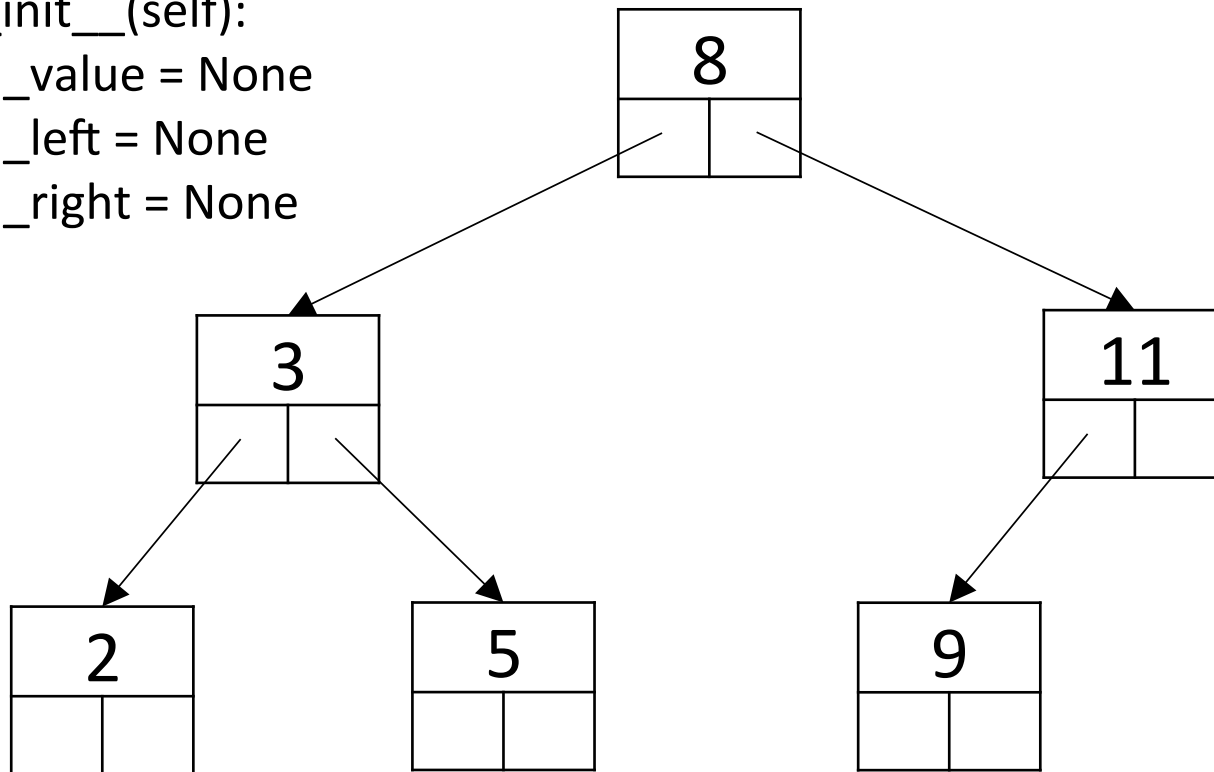
Evaluate:  $(x + y * 3) / (n - 1)$

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



# Exercise

```
class BinarySearchTree:  
    def __init__(self):  
        self._value = 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.

# 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?

# Exercise (Answer)

# count the leaves of a tree

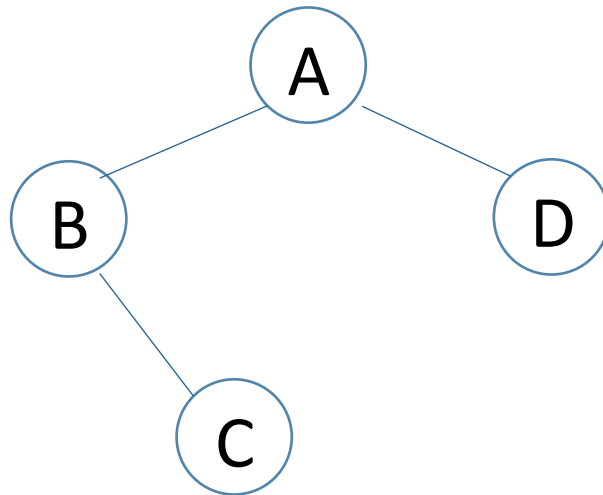
```
def count_leaves(t):  
    if t == None:  
        return 0  
    if t._left == None and t._right == None:  
        return 1  
    else:  
        return count_leaves(t._left) + \  
                count_leaves(t._right)
```

# Trees $\leftrightarrow$ traversals

What is the preorder traversal of this tree?

Inorder?

Postorder?



# Trees $\leftrightarrow$ traversals

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

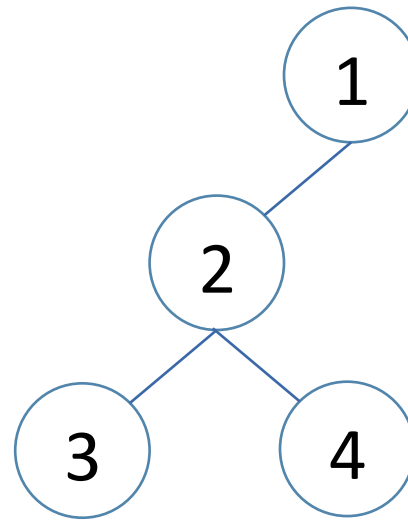
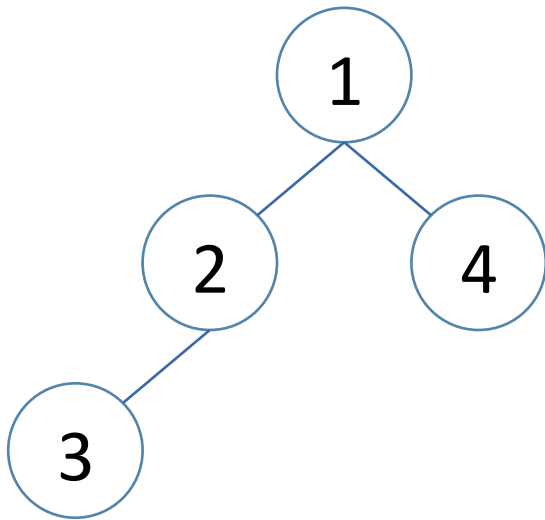
I.e., given a traversal, can we figure out the tree?

preorder: 1 2 3 4

# Trees $\leftrightarrow$ traversals

- The two trees below have the same preorder traversal.

Preorder traversal = 1 2 3 4

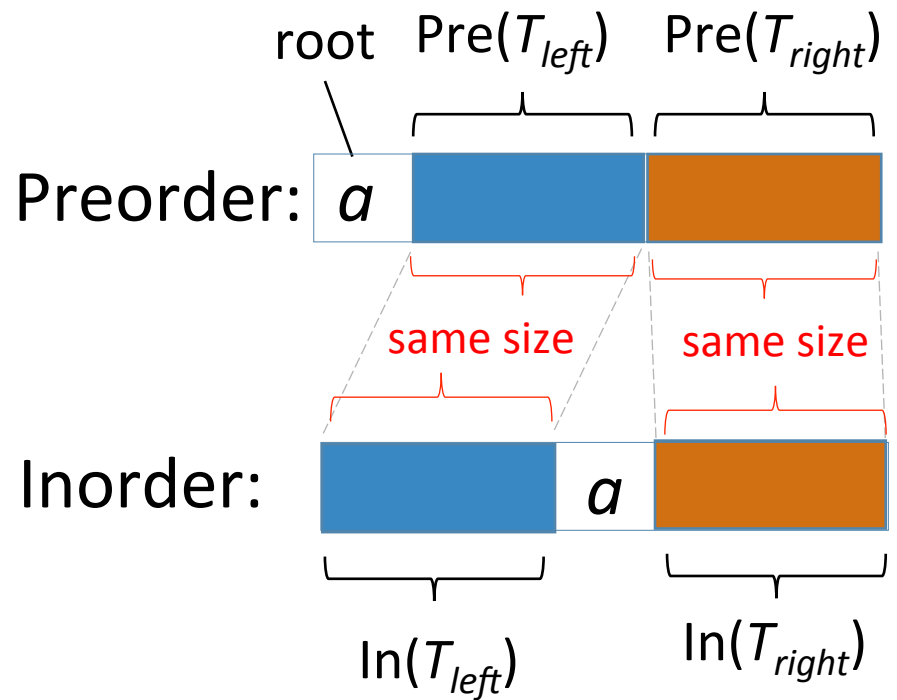
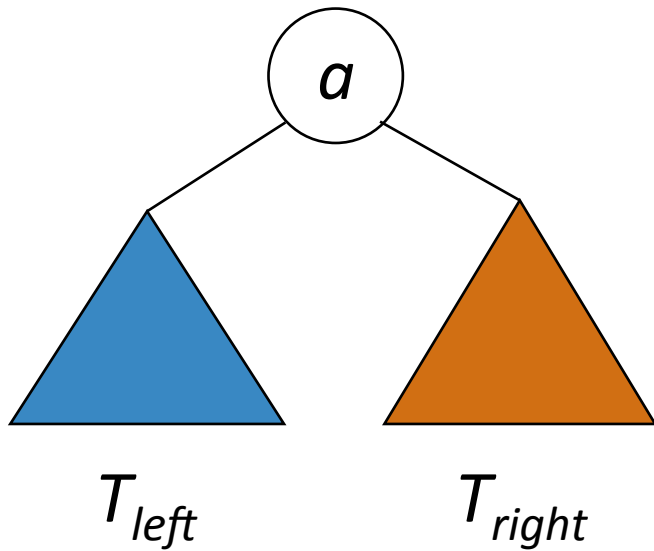


# Trees $\leftrightarrow$ traversals

- 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 $\leftrightarrow$ traversals



# Trees $\leftrightarrow$ traversals

- Given a preorder and 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 $\leftrightarrow$ traversals

- Given:

- preorder\_list + inorder\_list

- Suppose we can figure out:

- root

- preorder\_left + preorder\_right

- inorder\_left + inorder\_right

# Trees $\leftrightarrow$ 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 $\leftrightarrow$ traversals

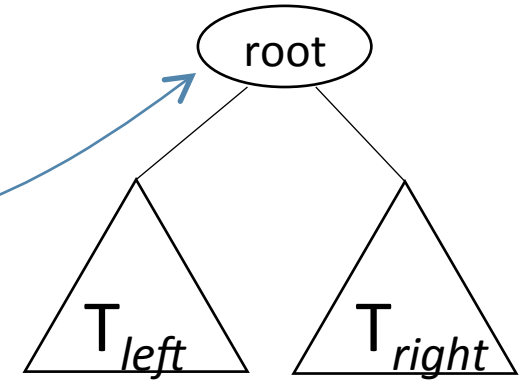
- Given:
  - preorder\_list + inorder\_list
- Suppose we can figure out:
  - root
  - preorder\_left + preorder\_right
  - inorder\_left + inorder\_right

- Then:

traversals\_to\_tree(preorder\_left, inorder\_left)  $\rightarrow T_{left}$   
traversals\_to\_tree(preorder\_right, inorder\_right)  $\rightarrow T_{right}$   
*recursion*

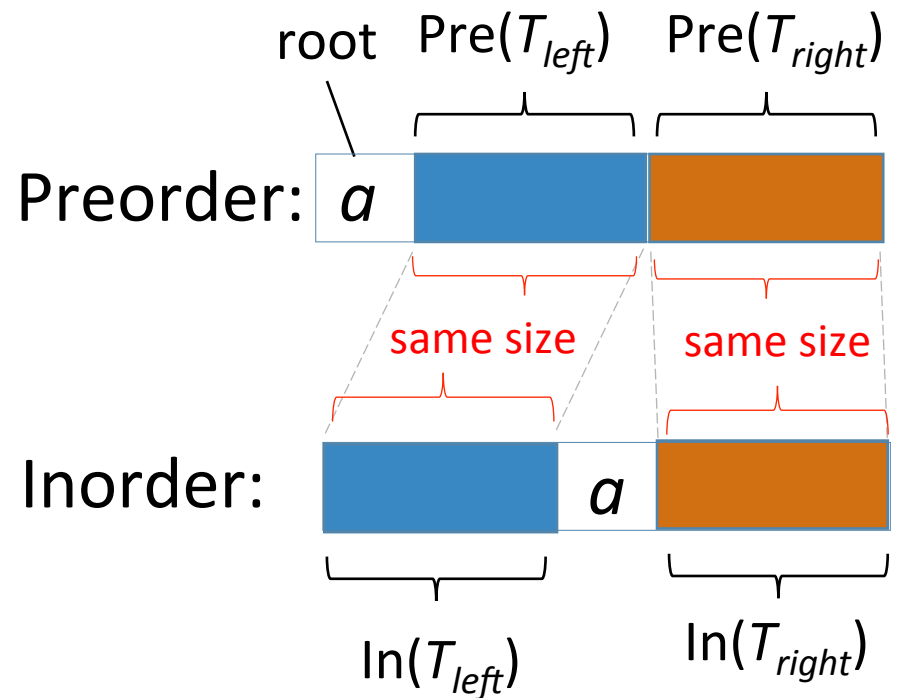
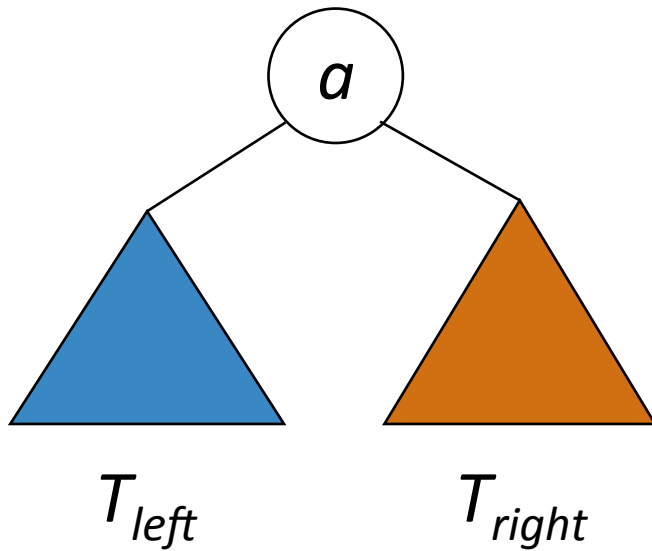
# Trees $\leftrightarrow$ traversals

- 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)  $\rightarrow$   $T_{right}$

# Trees $\leftrightarrow$ traversals



more traversals



# Consider: game playing

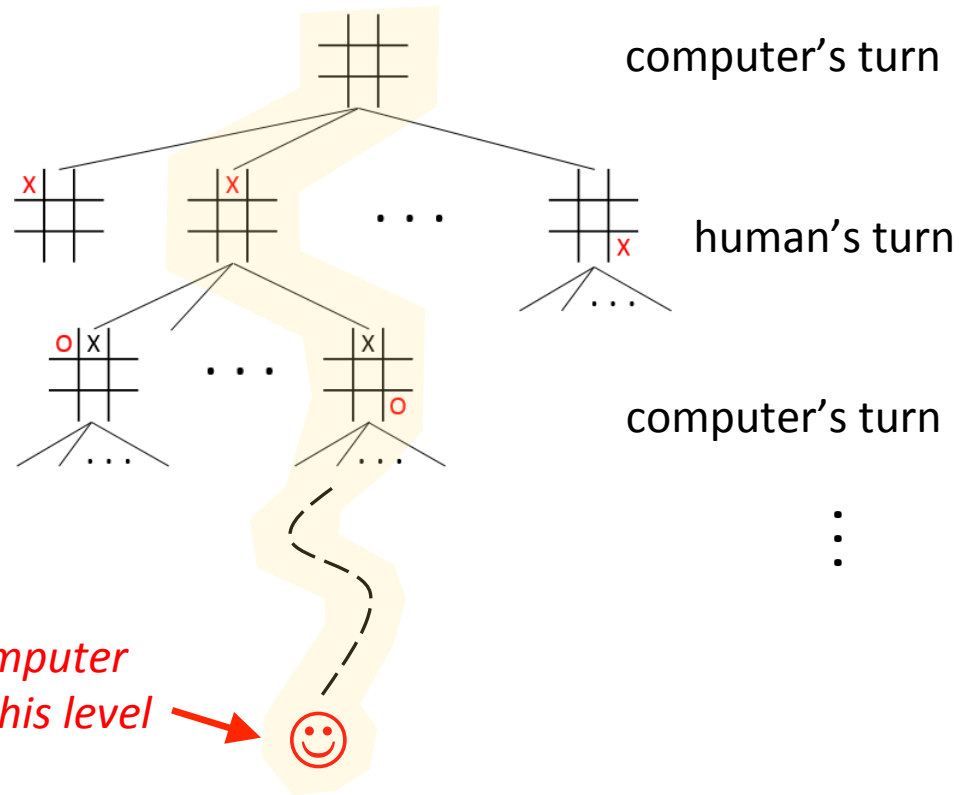
Goal: to write a program to play a 2-person game (e.g., tic-tac-toe, chess, go, ...)

How does this work?

# Consider: game playing

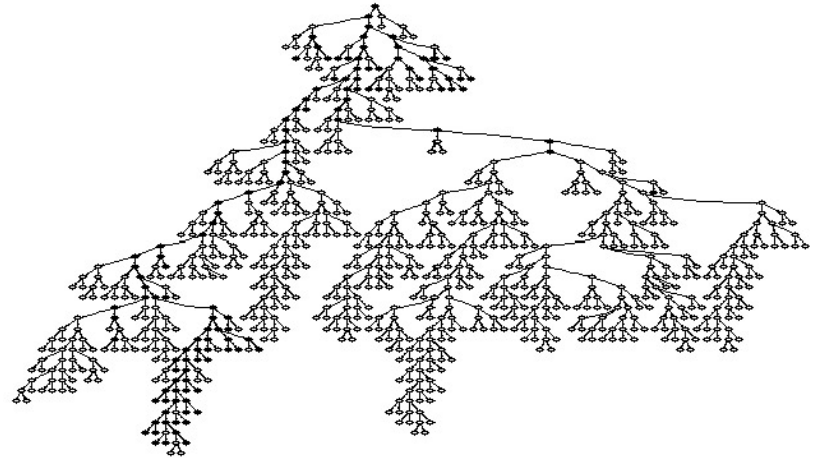
Goal: to write a program to play a 2-person game  
(e.g., tic-tac-toe, chess, go, ...)

- 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

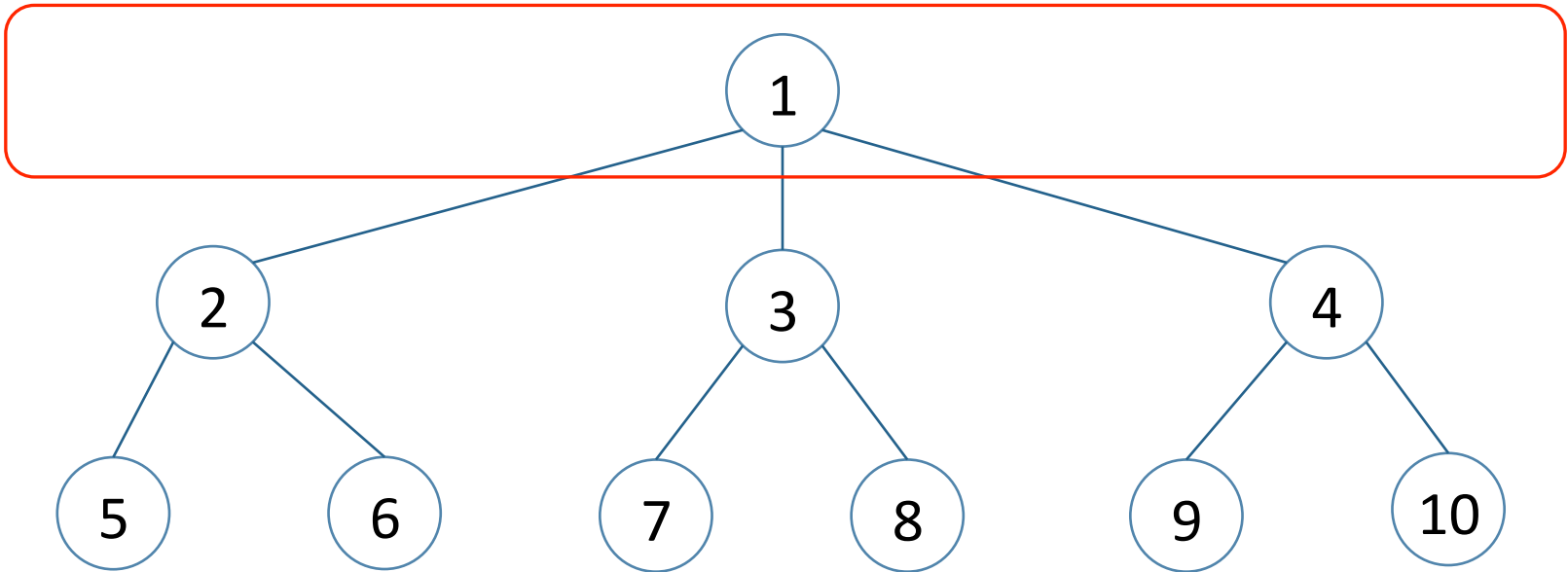
- For a nontrivial game (e.g., chess, go) the tree is usually too large to build or explore fully
  - also, usually there are time constraints on play
  - our previous tree traversal algorithms don't work
- Game-playing algorithms typically explore the tree level by level
  - consider the nodes at depth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



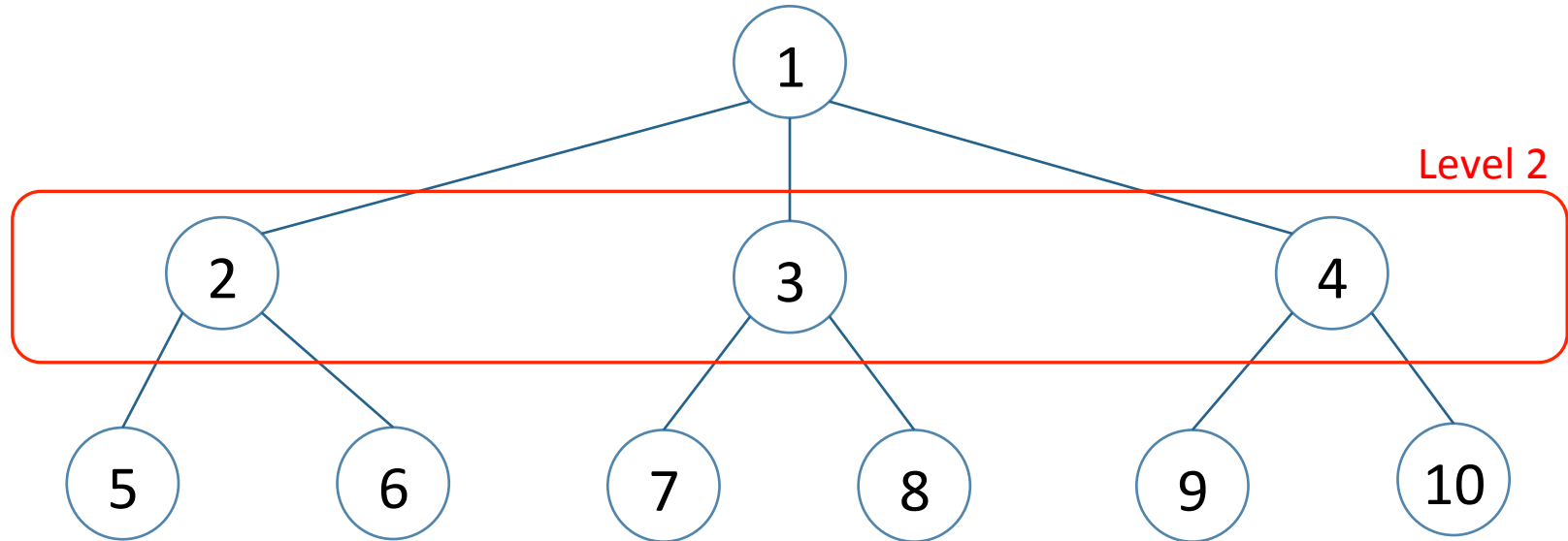
A game tree

# Level-by-level tree traversal

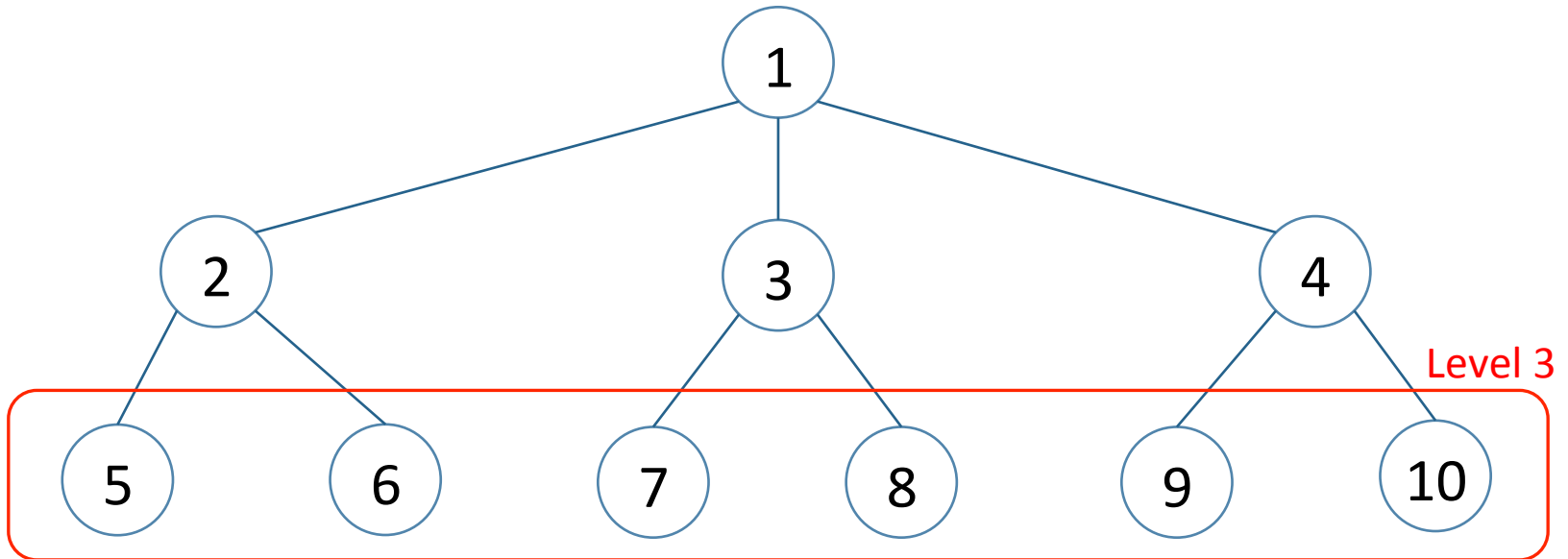
Level 1



# Level-by-level tree traversal

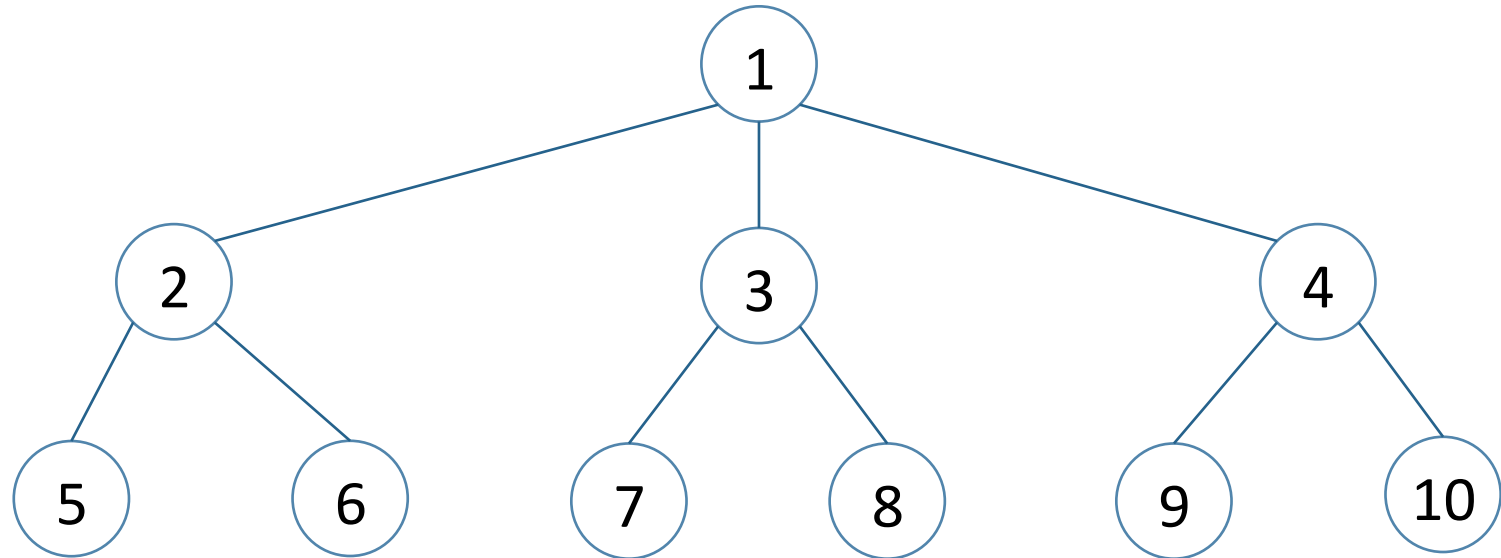


# Level-by-level tree traversal



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

# Breadth-first tree traversal



Breadth-first traversal order:

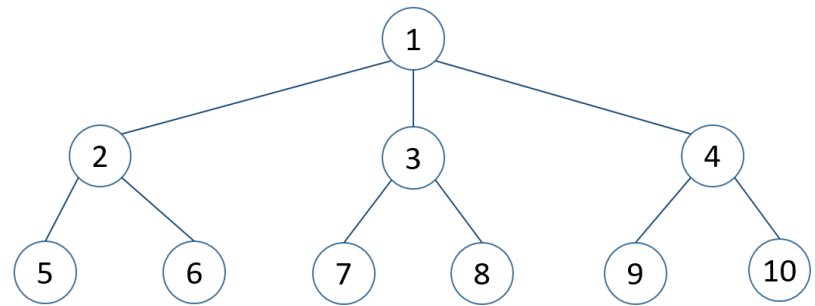
1 2 3 4 5 6 7 8 9 10

# 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

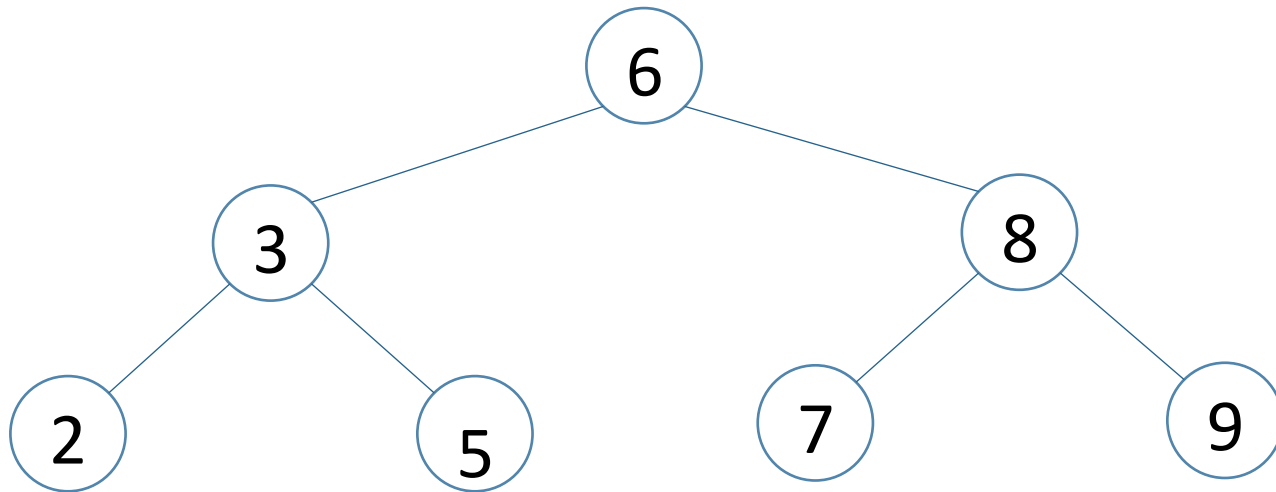
- Stacks and queues are closely related structures
- What if we use a stack in our tree traversal?
  - the deeper levels of the tree are explored first
  - this is referred to as *depth-first traversal*

# BST / Complexity

# Binary Search tree: complexity

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

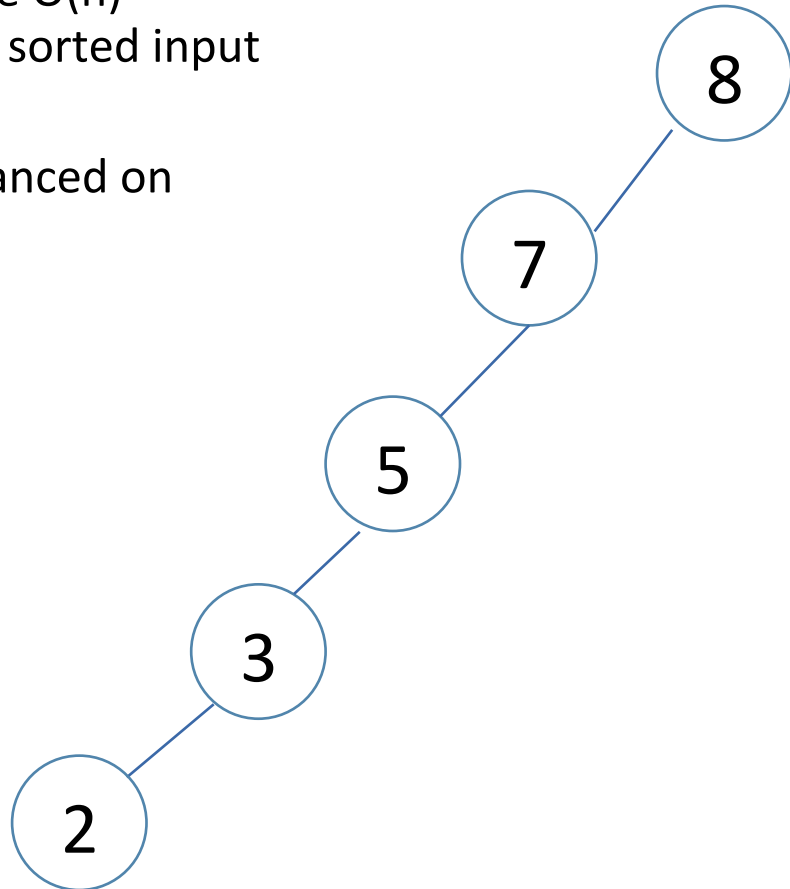
Note: this tree is balanced



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
- A binary tree represents a sequence
- 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

# Assgn 12 - Huffman Coding

- ASCII uses fixed length encodings:

char	ASCII	binary
g	103	1100111
o	111	1101111
p	112	1110000
h	104	1101000
e	101	1100101
r	114	1110010
s	115	1110011

What is this word?

11010001101111111001011100111100101

binary: 1101000 1101111 1110010 1110011 1100101 (group by 7's)

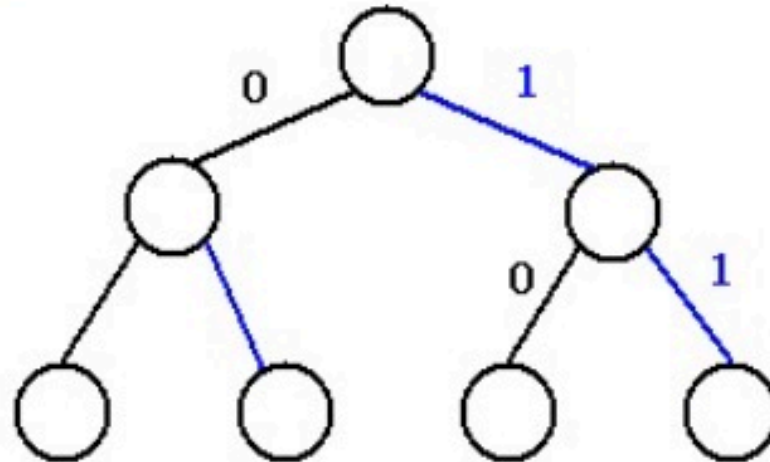
ASCII: 104 111 114 115 101

# Assign 12 – Huffman Coding

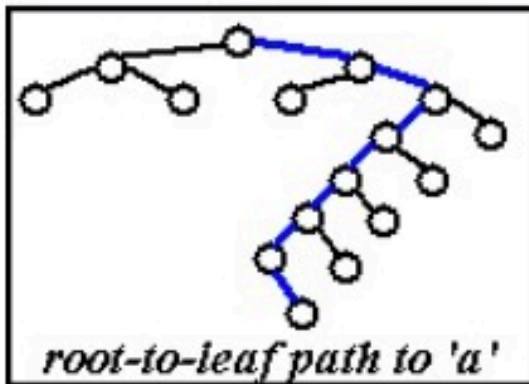
- Huffman coding:
- Use fewer bits (not 7) for more frequently occurring characters
- Do this by using a tree that stores characters at the leaves
- root-to-leaf paths provide the bit sequence used to encode the characters

# Assign 12 – Huffman Coding

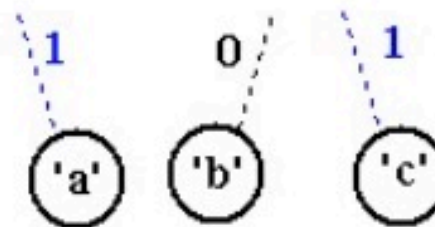
Source: Astrachan



leaves of a  
not-to-leaf  
edge  
d by  
'a', which  
left-left-



• • • •



'a' = 97	'b' = 98	'c' = 99
1100001	1100010	1100011

path: *right-right-left-left-left-right*

*using the 0/1 convention*

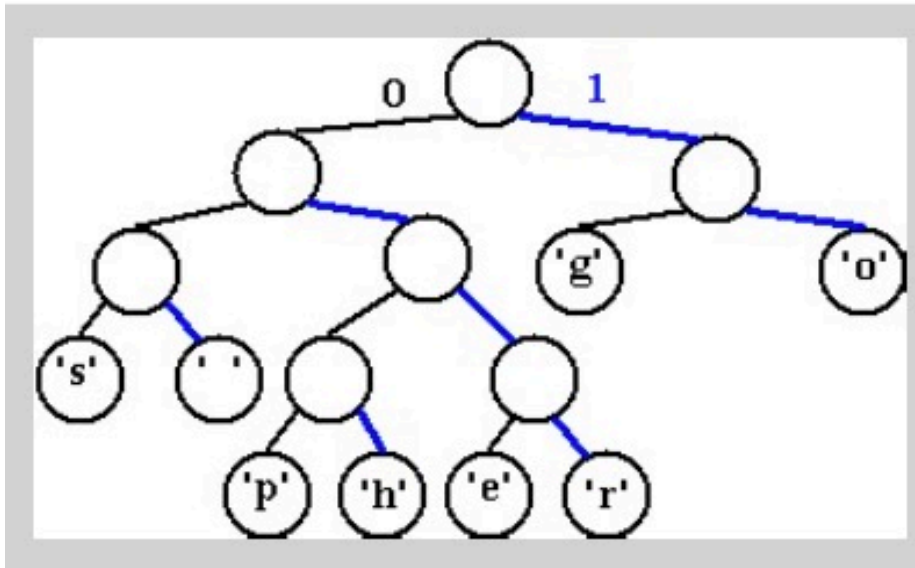


# Assign 12 – Huffman Coding

The structure of the tree can be used to determine the coding of any leaf by using the 0/1 edge convention. A different tree gives a different coding.

The tree below gives the coding on the right.

*Source: Astrachan*



char	binary
'g'	10
'o'	11
'p'	0100
'h'	0101
'e'	0110
'r'	0111
's'	000
' '	001

What would the encoding be for hope?

# Assign 12 – Huffman Coding

Prefix codes/Huffman codes

*prefix property*: no bit-sequence encoding of a character is the prefix of any other bit-sequence encoding.

When all characters are stored in leaves and every non-leaf node has two children, the coding produced by the 0/1 convention has the prefix property

*invented by Huffman 1952*