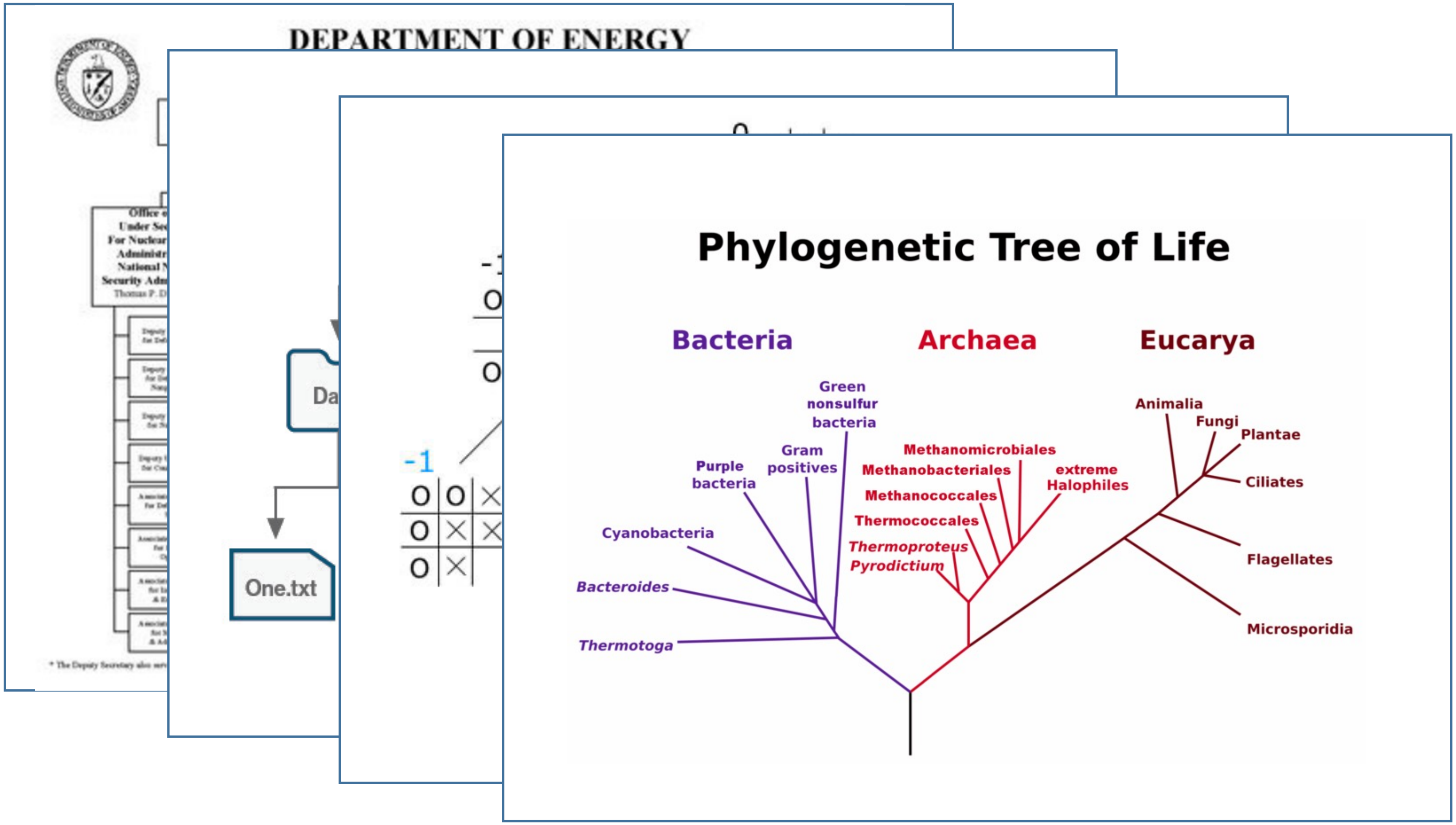
CSc 120
Introduction to Computer Programming II

07: Trees

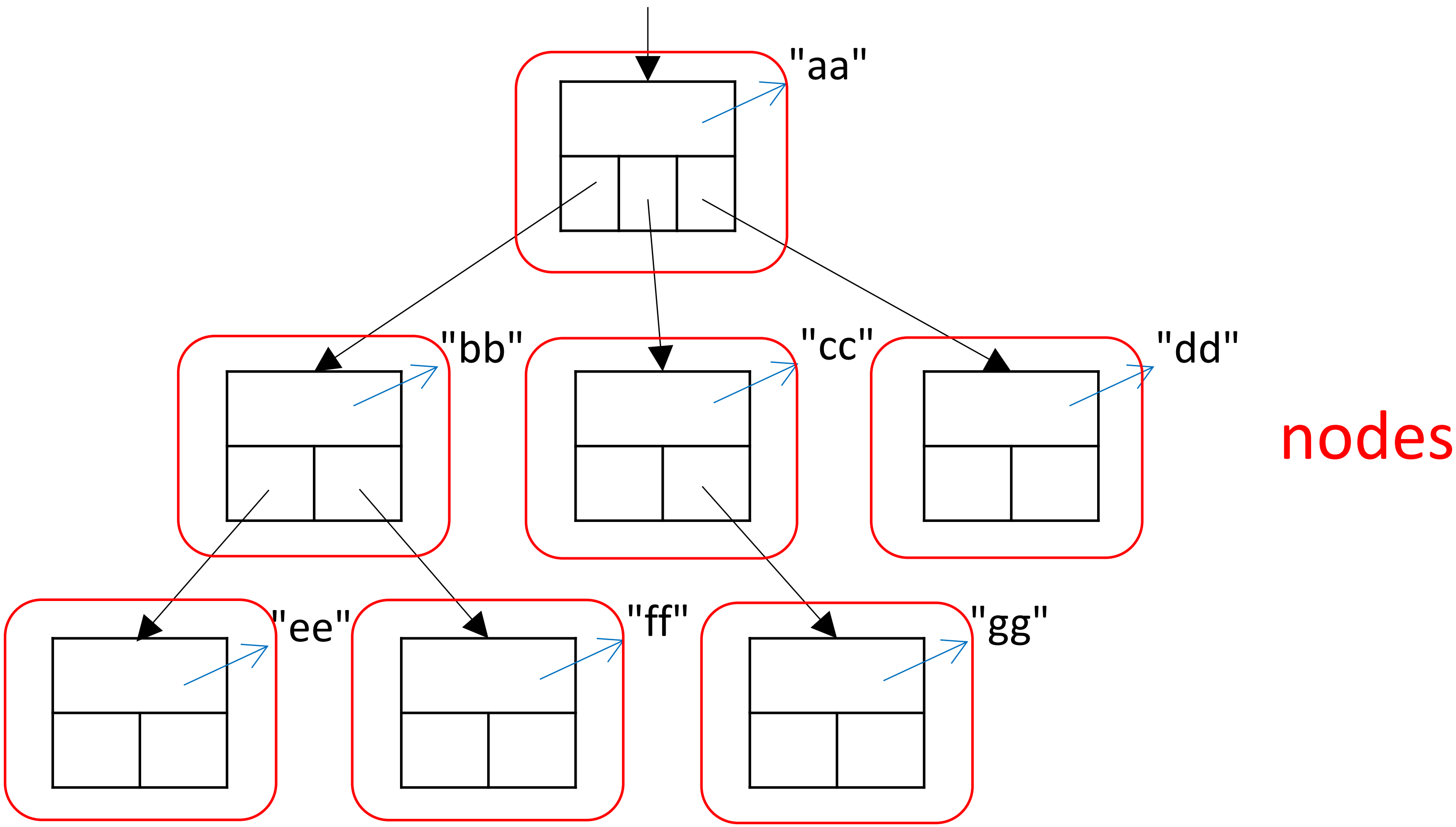
trees: basic concepts

Hierarchies

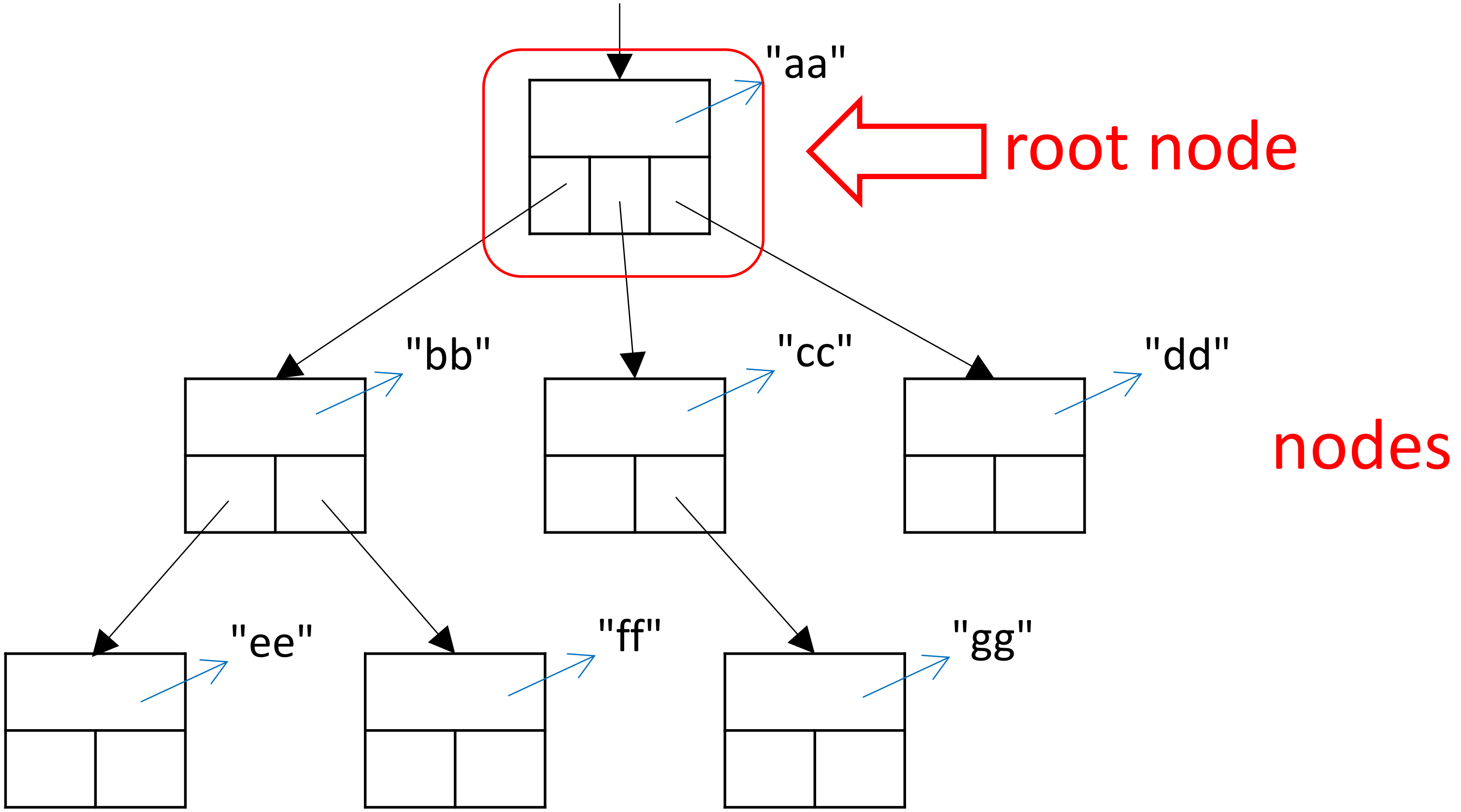
- Hierarchically organized "stuff" are everywhere



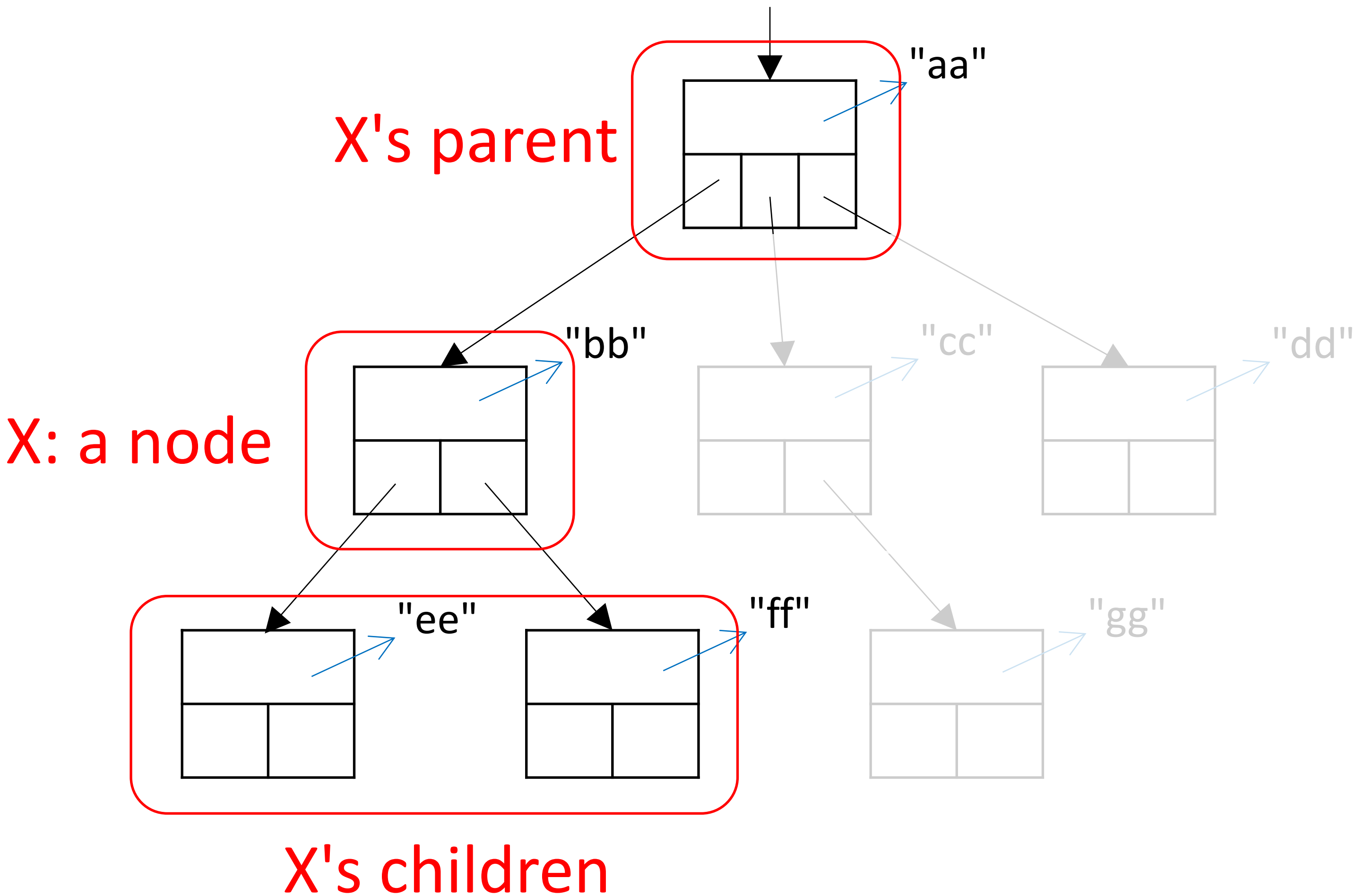
Trees



Trees



Trees

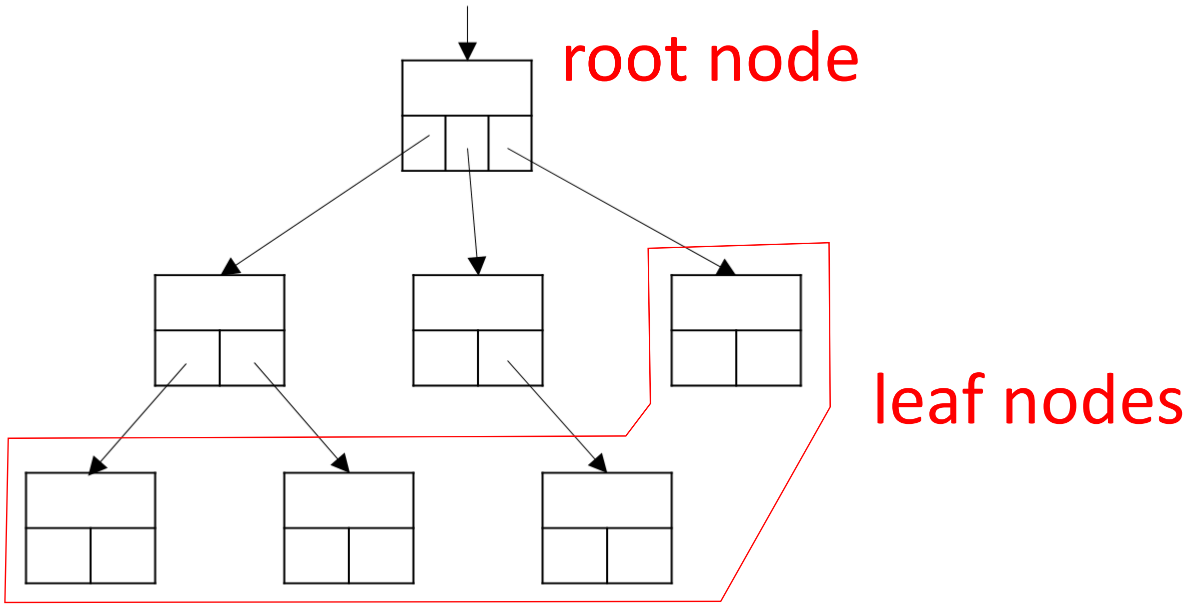


Trees: terminology

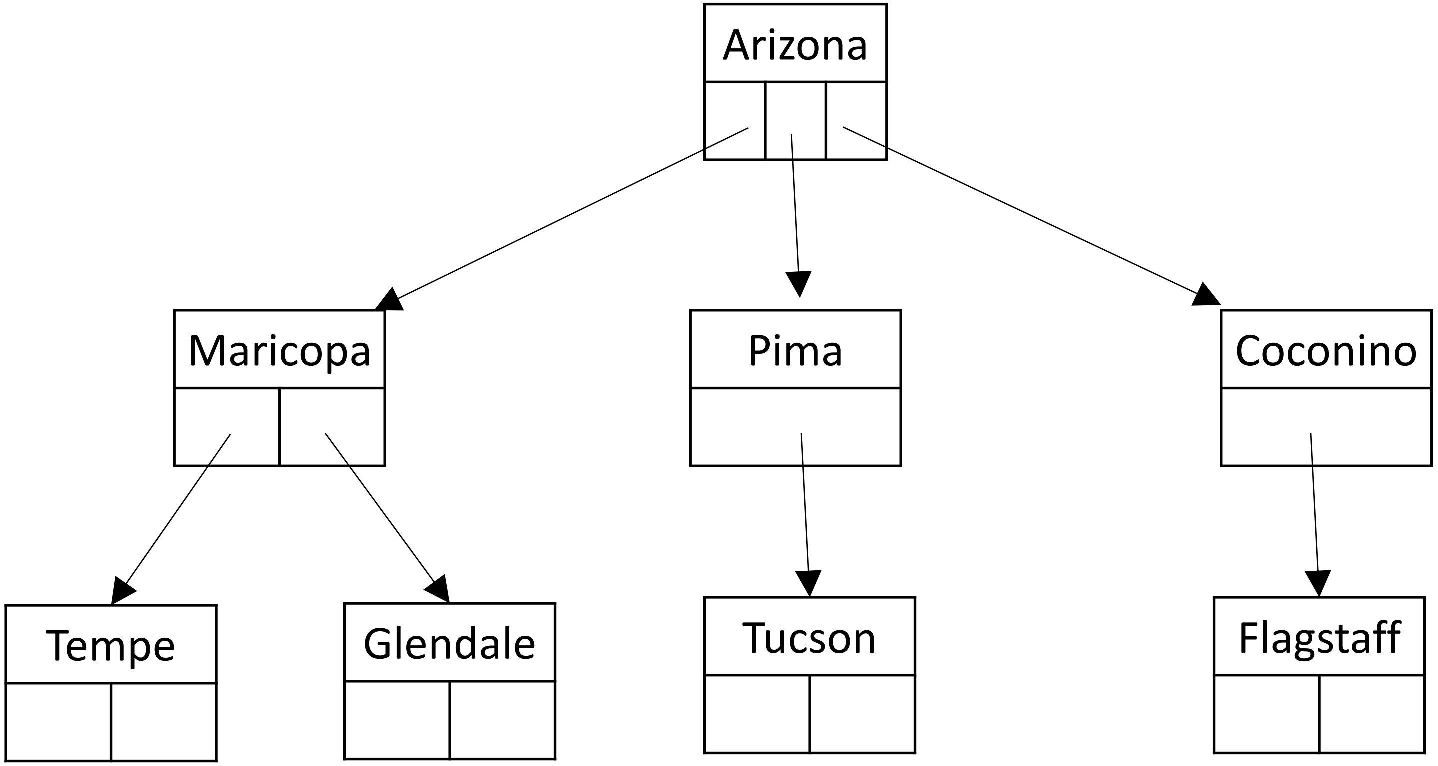
- A tree is a collection of nodes
- Each node has:
  - $\geq$ 0 child nodes
  - 0 or 1 parent nodes

  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

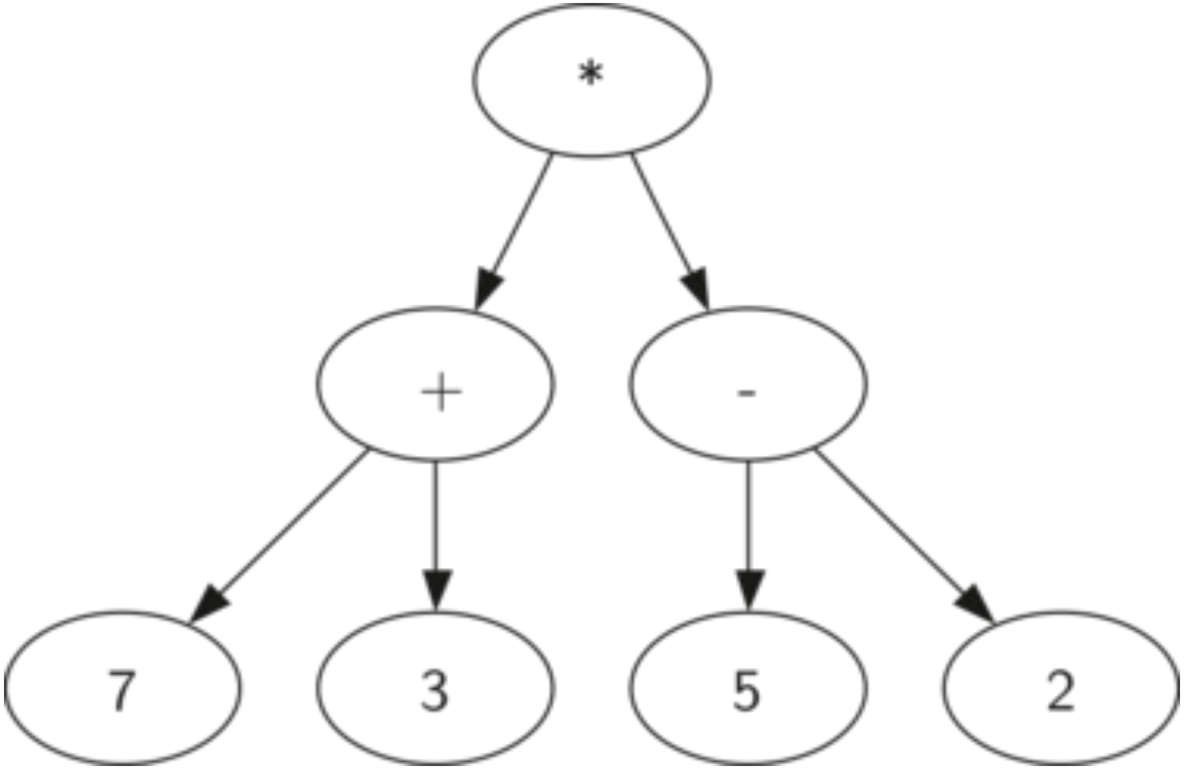


Tree: Example



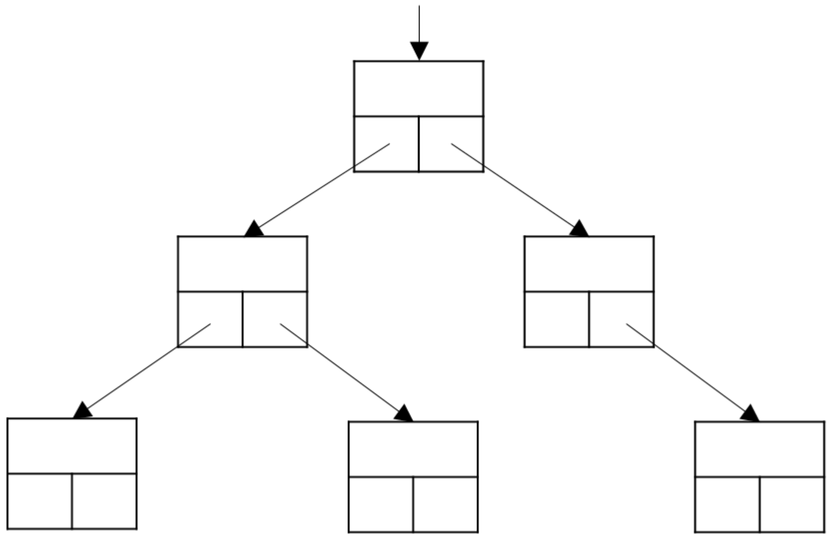
Tree: Example

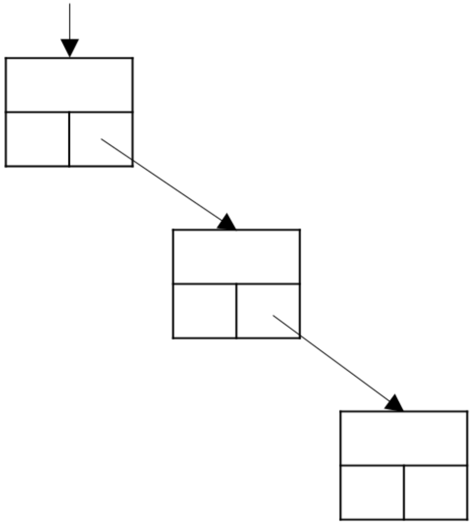
(7+3)*(5-2)



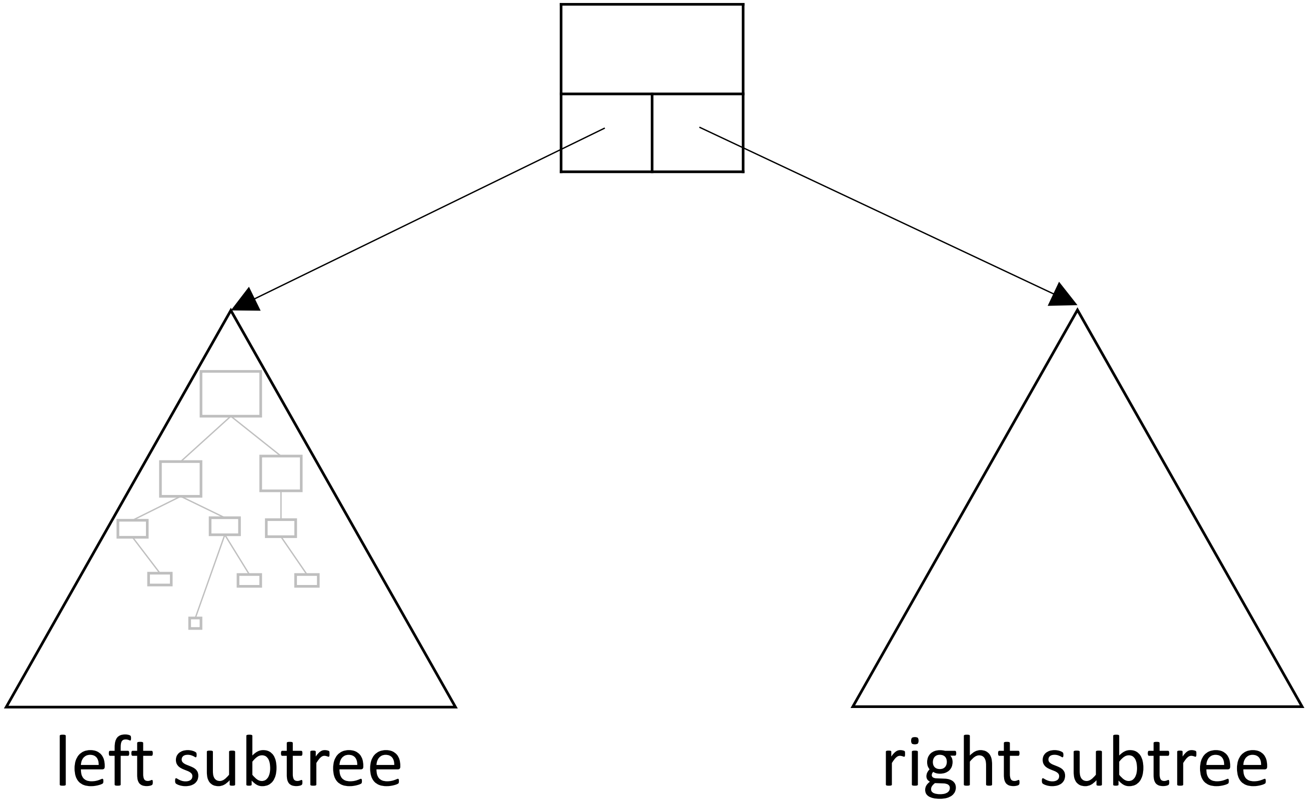
Binary trees

- 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., 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

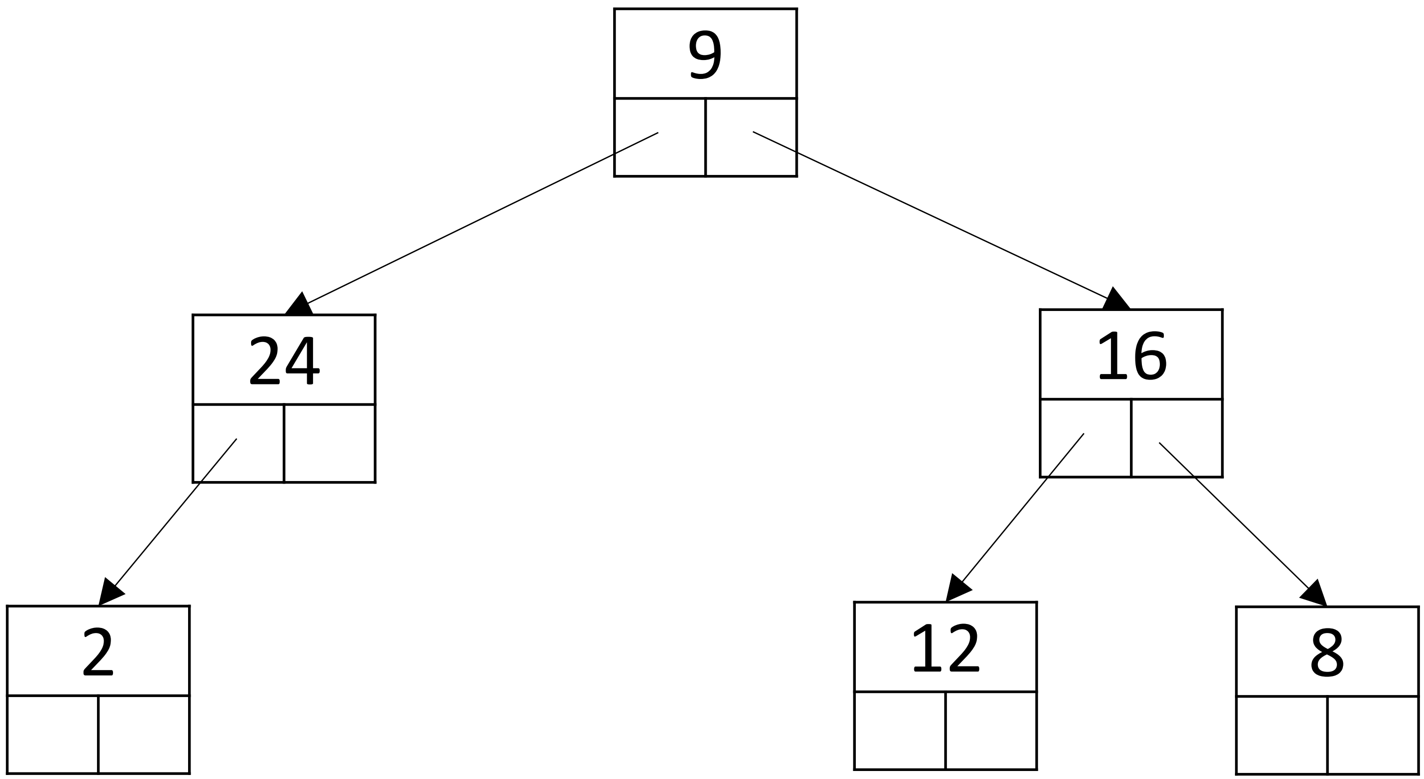
Binary trees: node representation

```
class BinaryTree:
    def __init__(self, value):
        self._value = value      # the value at the node
        self._lchild = None      # left child
        self._rchild = None      # right child

    …
```

Exercise- ICA-22-Prob. 2-6

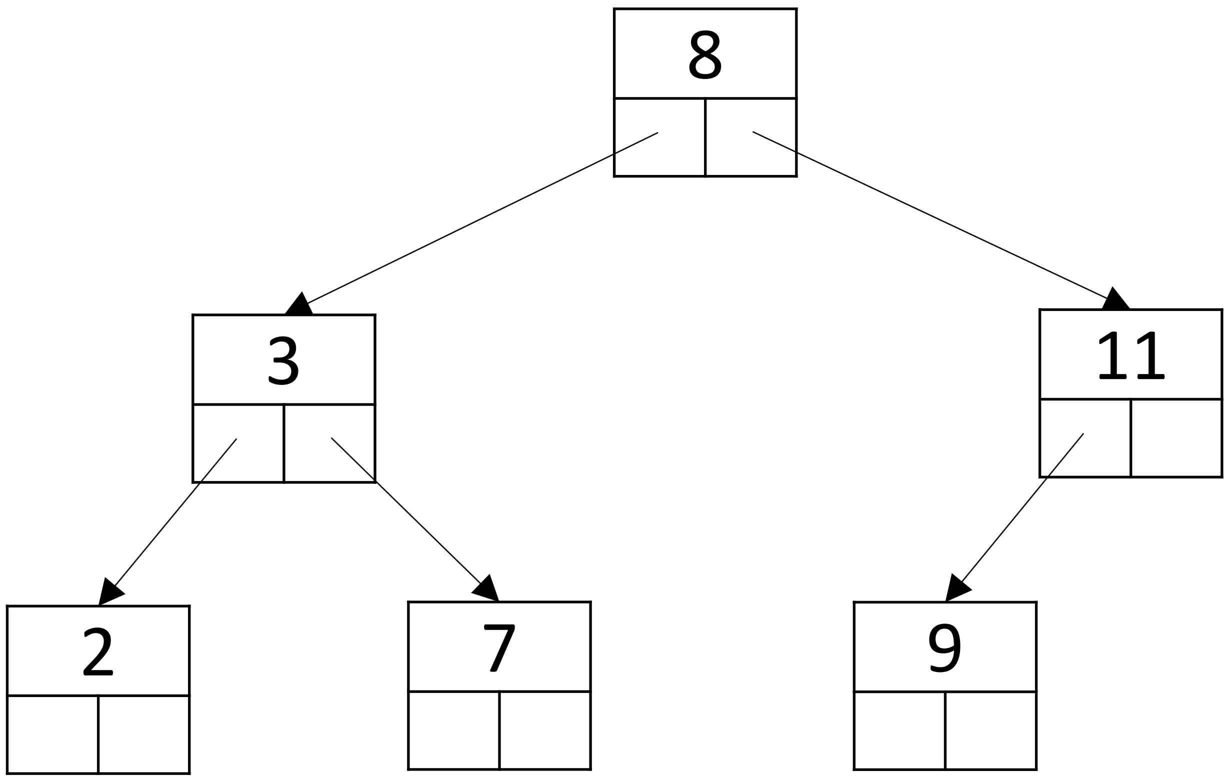
Do all remaining problems.

Whiteboard Exercise



List as many facts about this tree as you can.

Examine this binary tree:



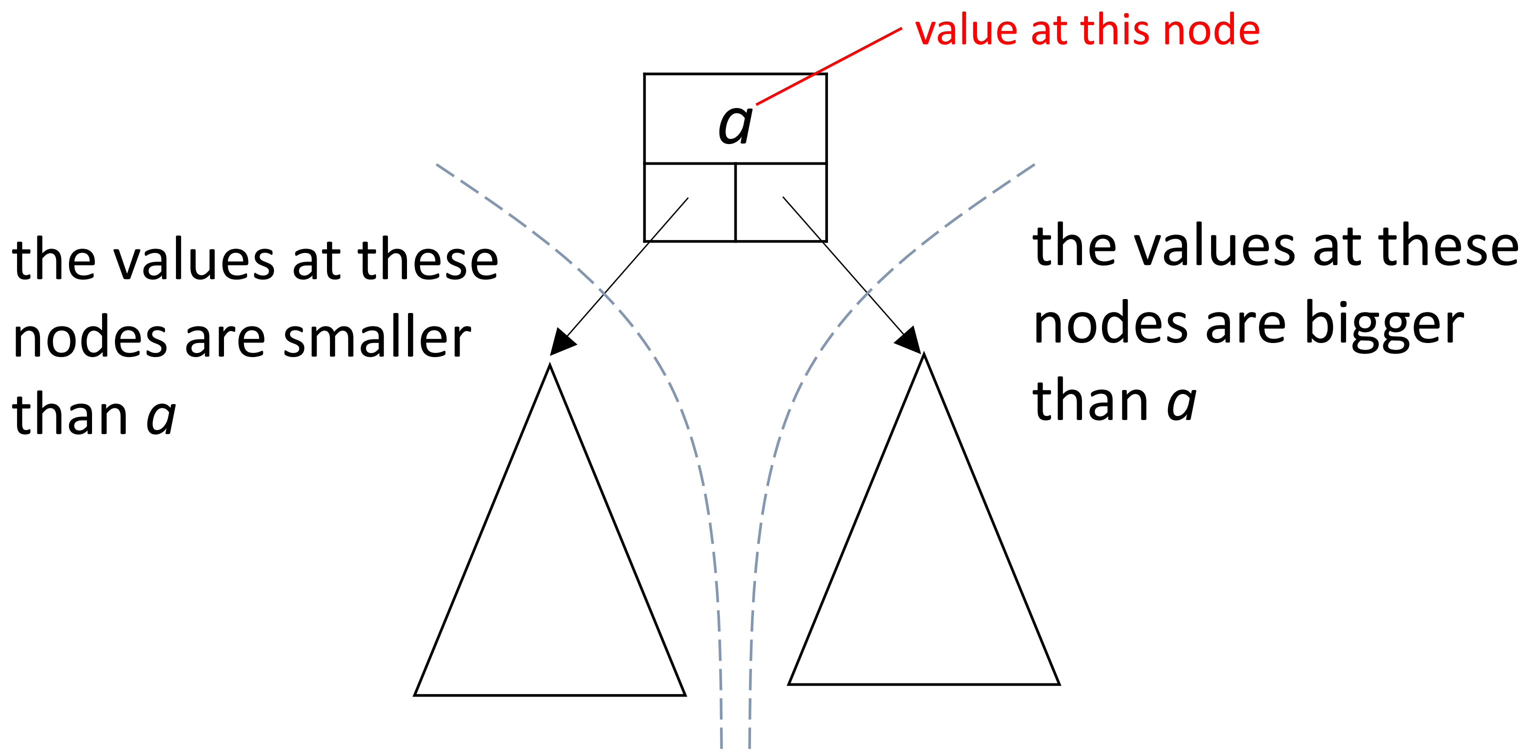
What can we say about the *values* in the nodes in the left subtree of 8?

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

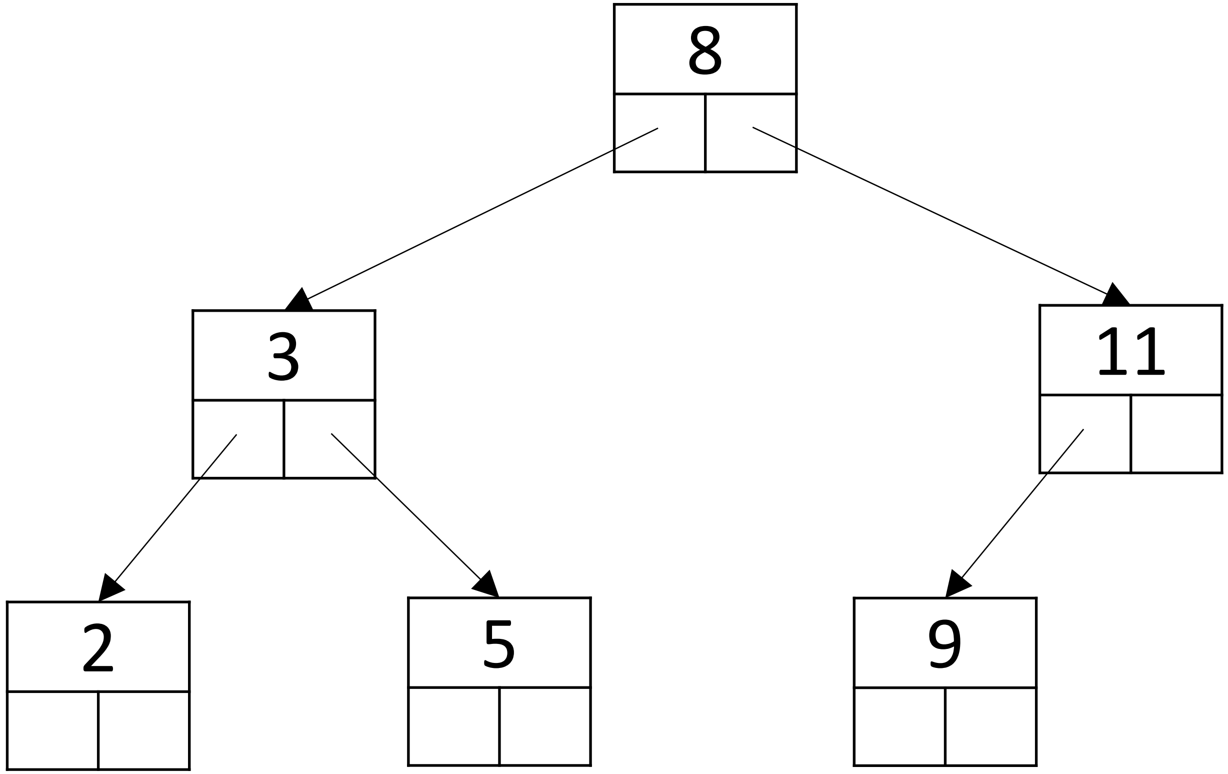
binary search trees

Binary search tree (BST)

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

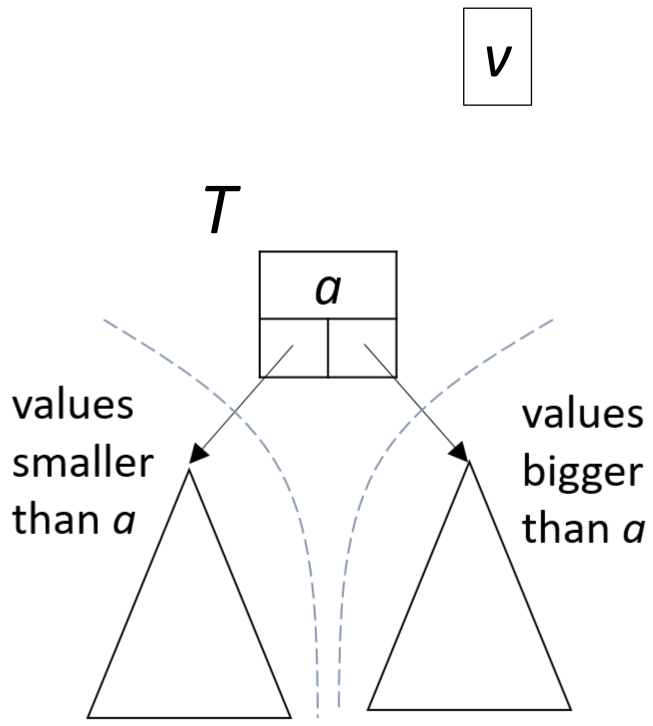


Binary search tree: Example



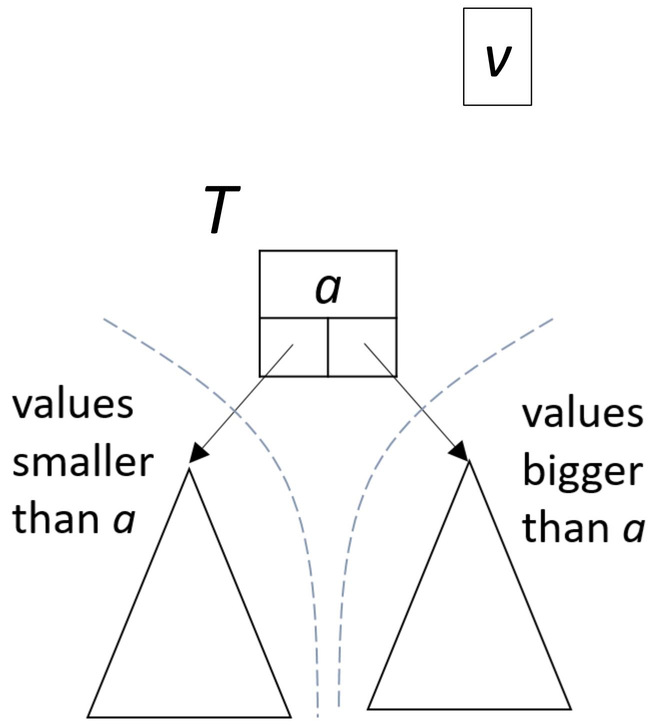
Searching a BST

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*?

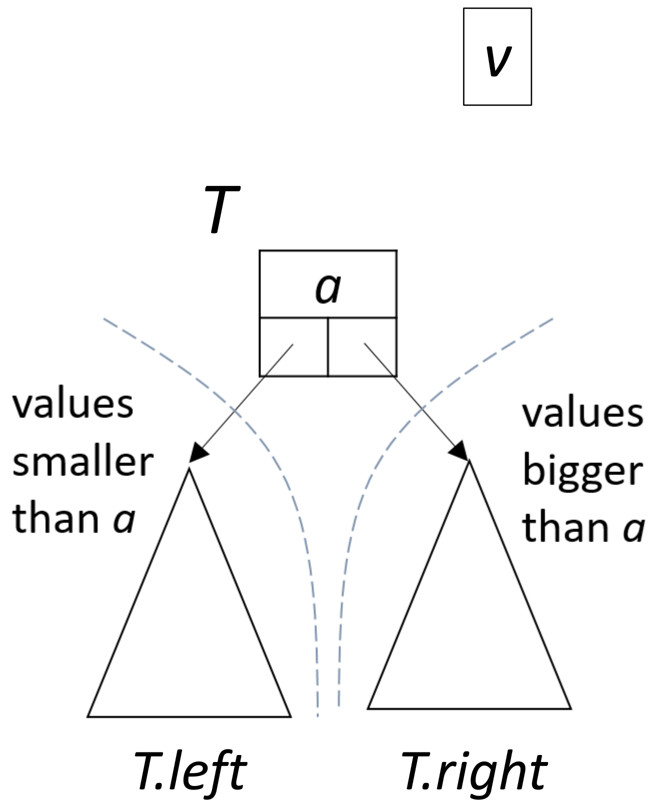


Idea: at each node with value *a*:

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

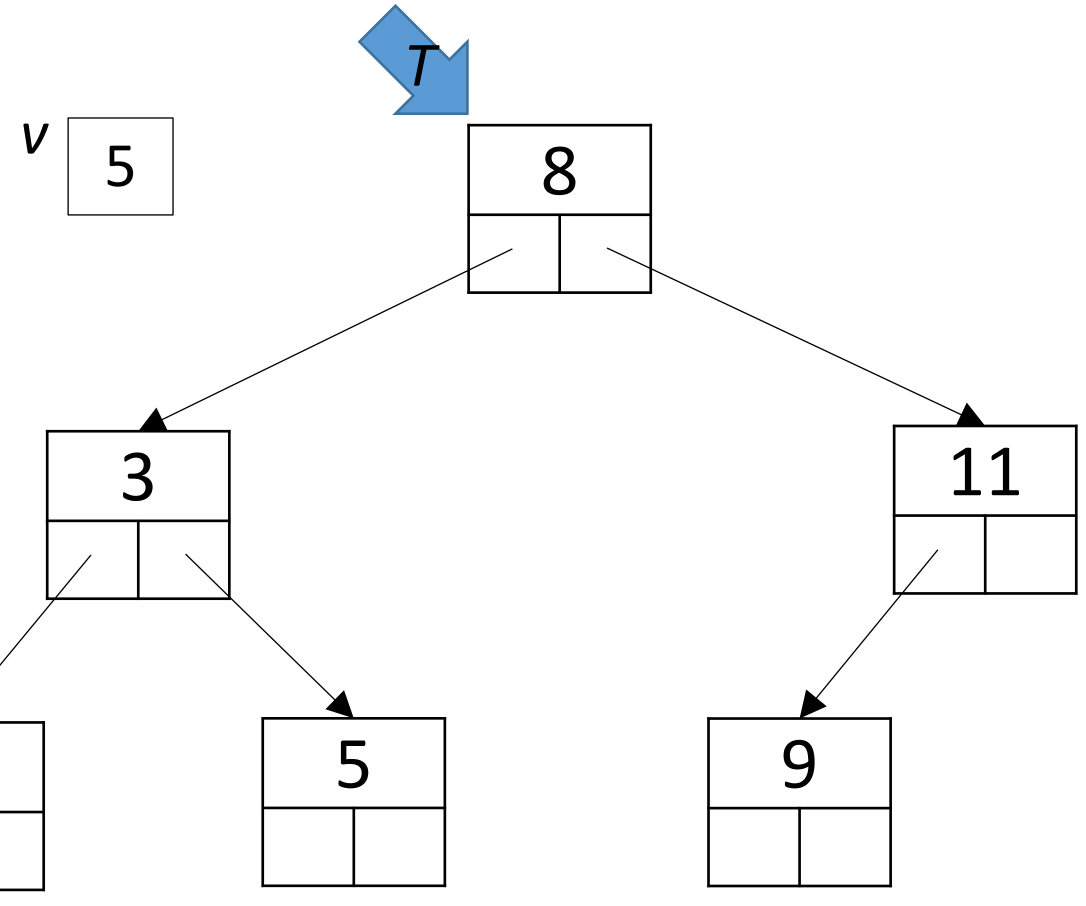
Searching a BST

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



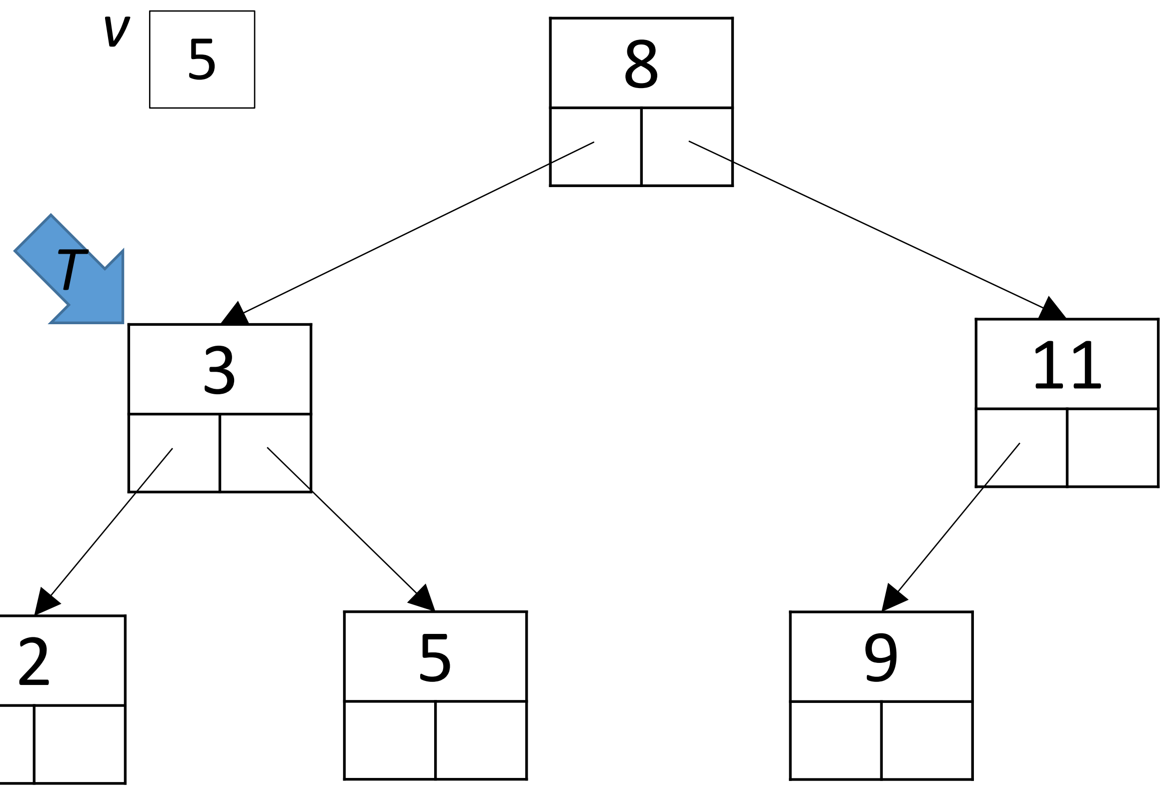
Searching a BST: Example 1

v < *T.value*:
⇒ search left subtree

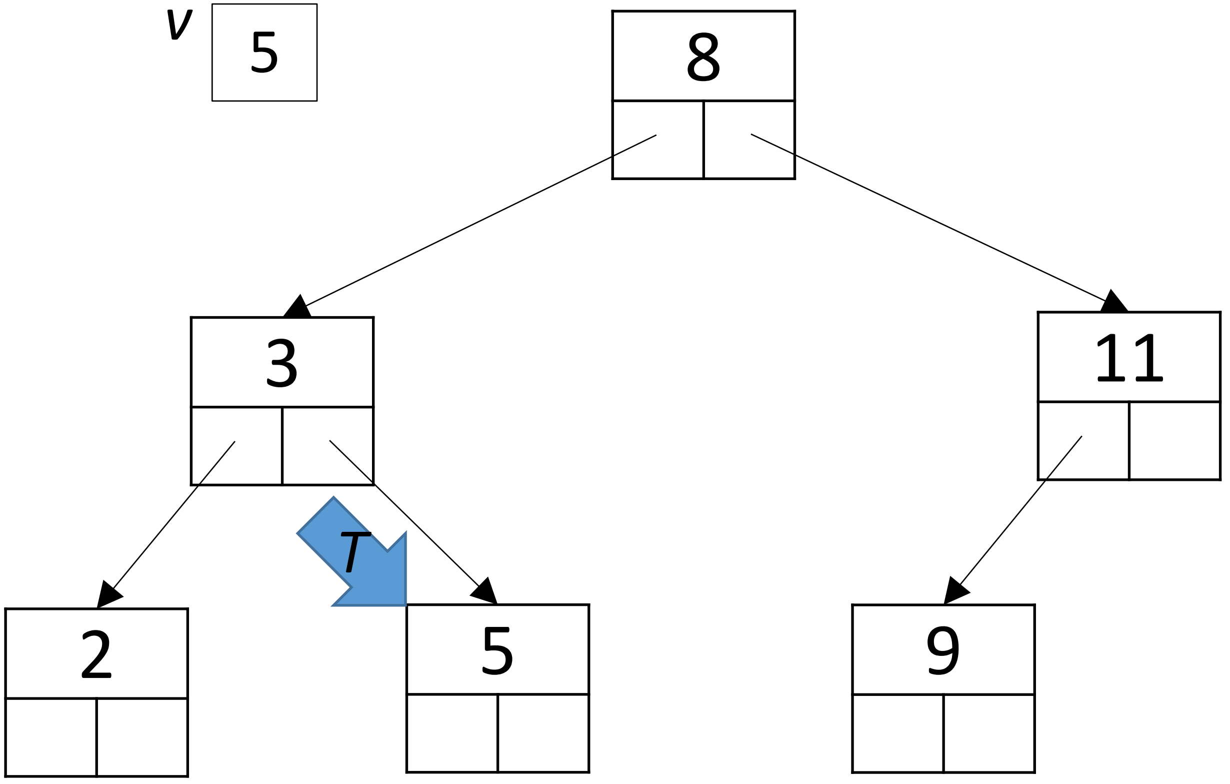


Searching a BST: Example 1

v > *T.value*:
⇒ search right subtree



Searching a BST: Example 1



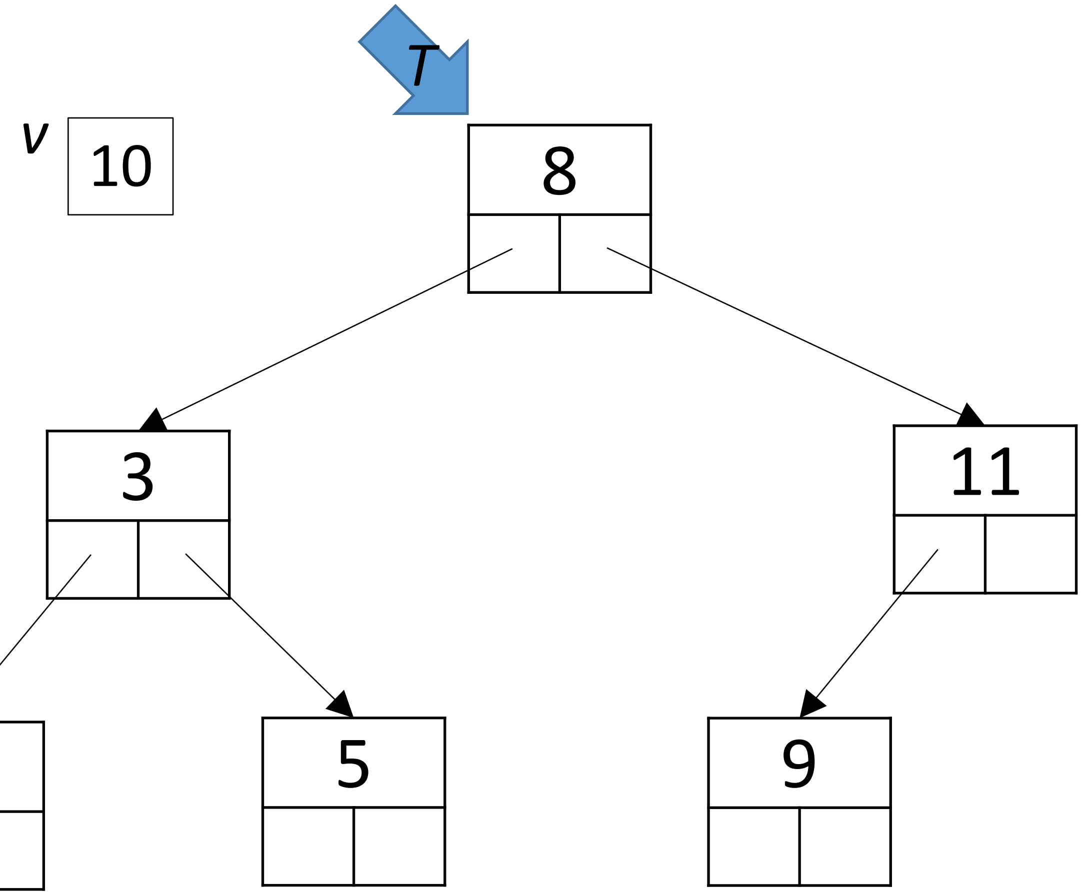
v == *T.value*:

$\Rightarrow$ return True

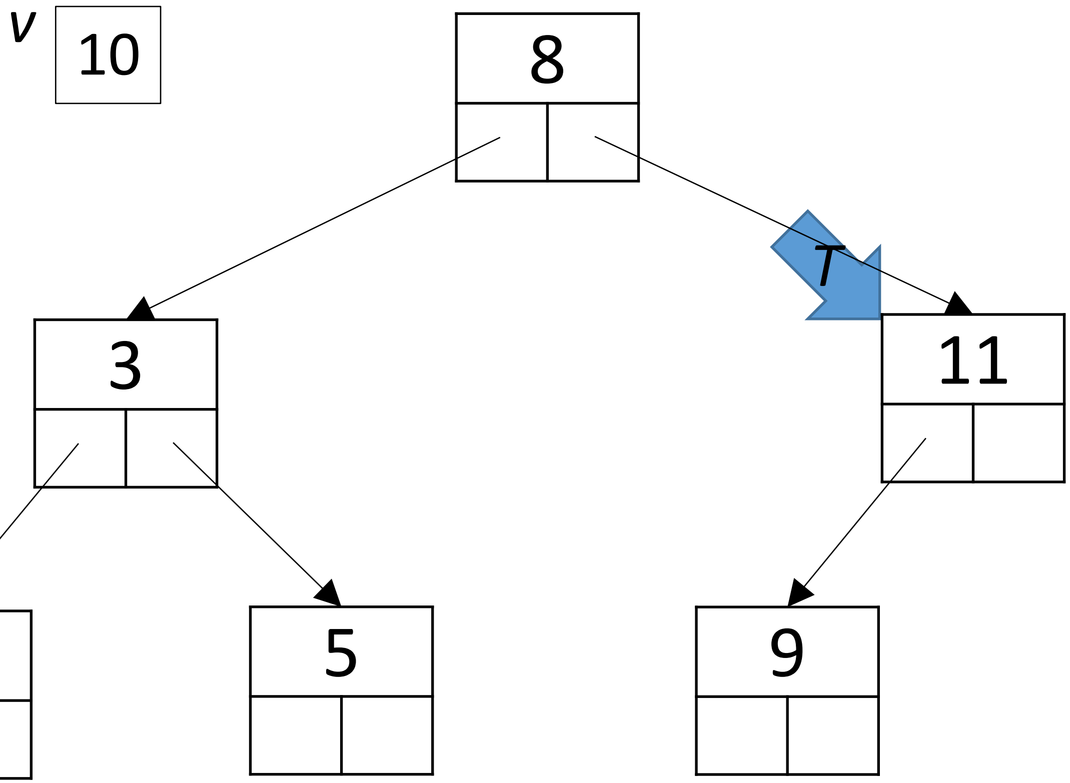
Searching a BST: Example 2

v > *T.value*:

⇒ search righ subtree

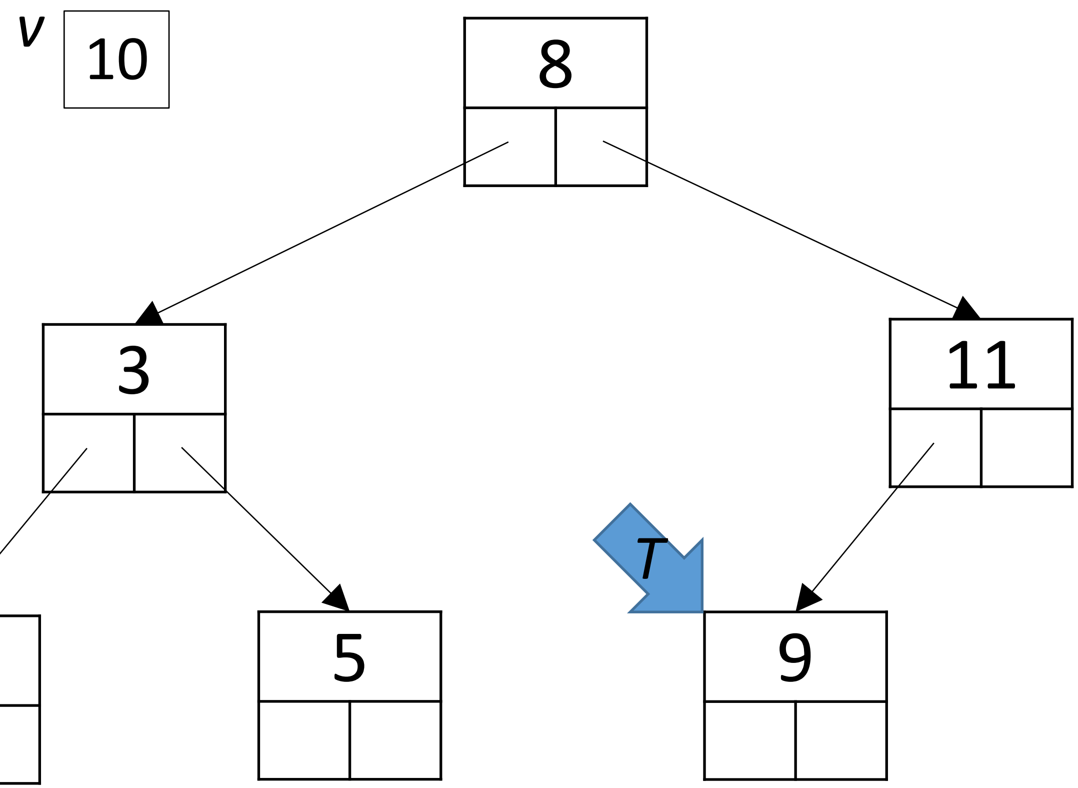


Searching a BST: Example 2



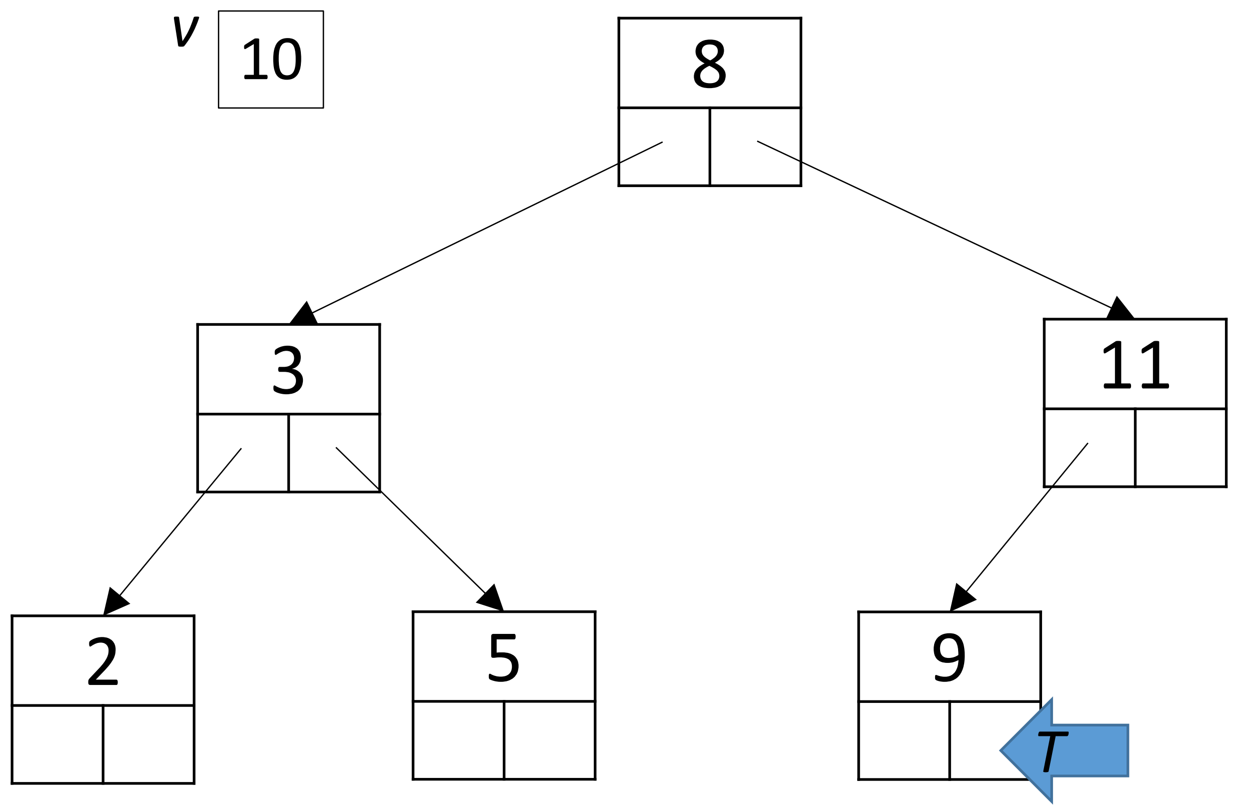
v < *T.value*:
⇒ search left subtree

Searching a BST: Example 2



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

Searching a BST: Example 2



T == None:

⇒ 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.*value* : insert into T's left subtree
  - if v == T.*value* : done (no duplicates)
  - if v > T.*value* : insert into T's right subtree

Constructing a BST

```
def insert(T, v):
    if T == None:
        return Node(v)
    # assumes no duplicates
    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: $\boxed{8}$ 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
```

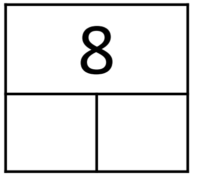
T: None

Constructing a BST: Example

Sequence of values: $\boxed{8}$ 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
```

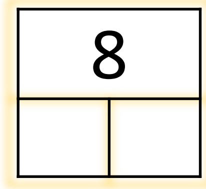
T:



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

    return T
```

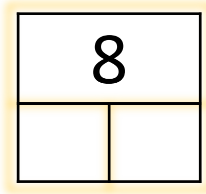
T: 

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

    return T
```

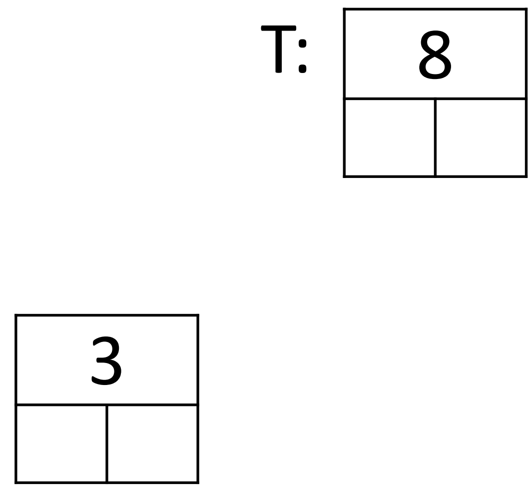
T:



Constructing a BST: Example

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

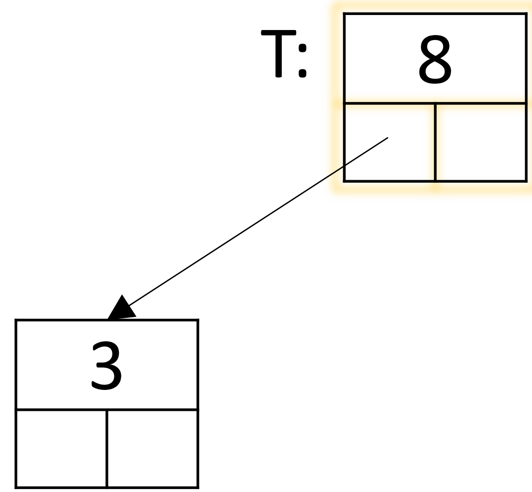
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

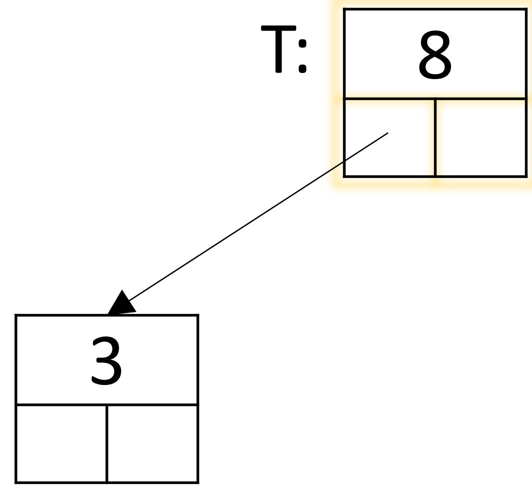
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

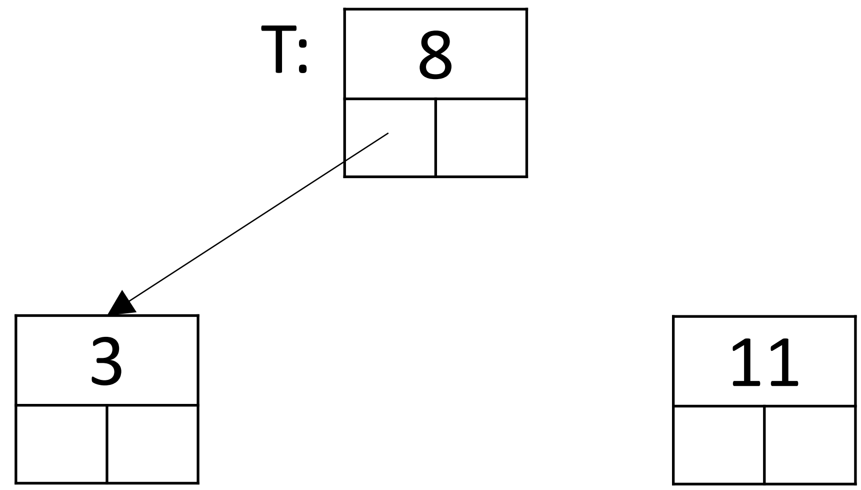
```
def insert(T, v):(v = 11, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

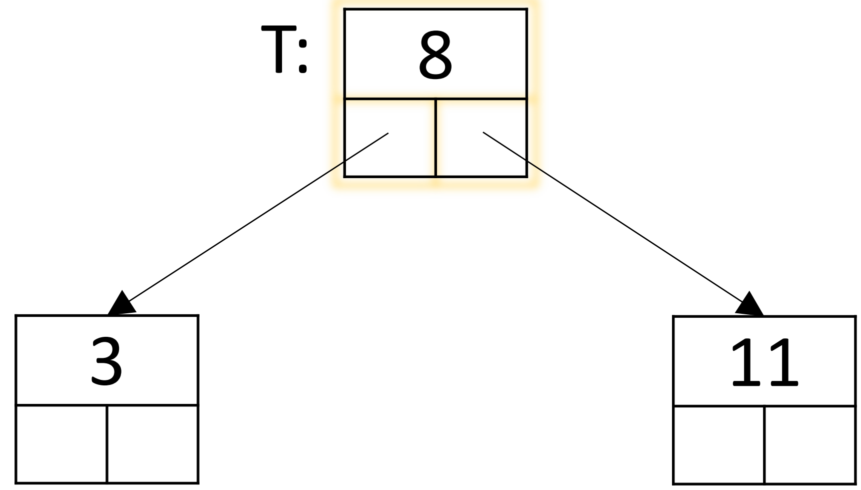
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

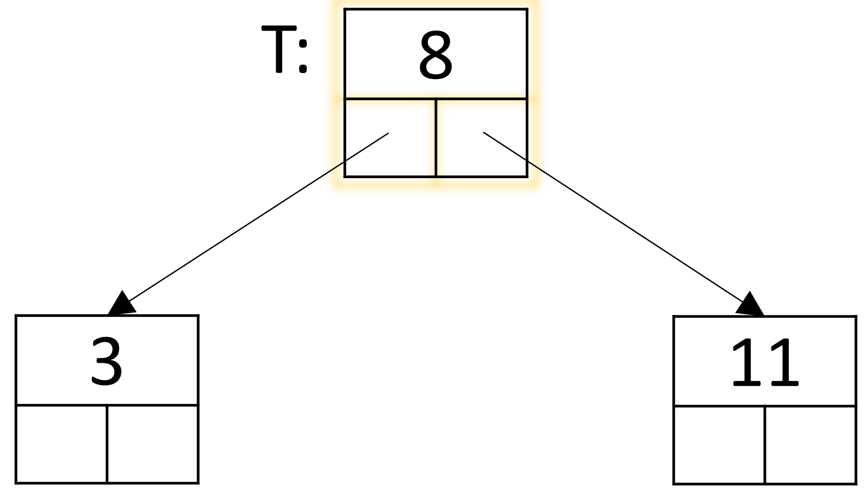
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

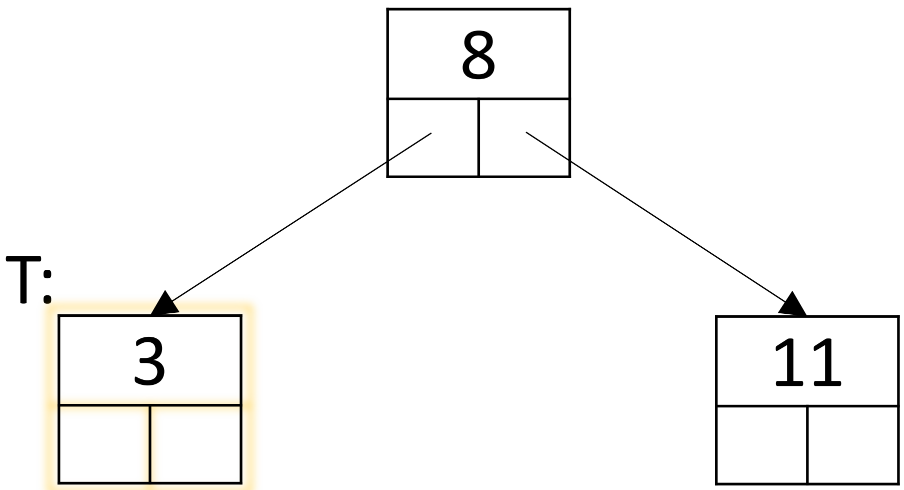
```
def insert(T, v): (v = 2, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

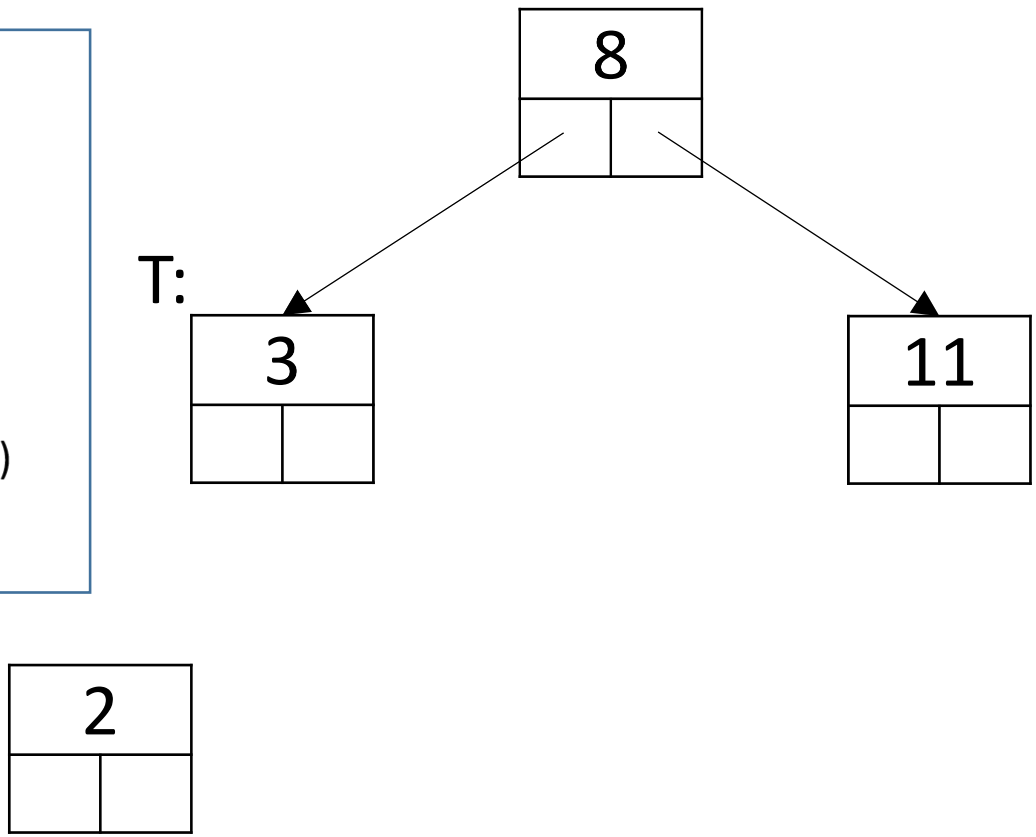
```
def insert(T, v):(v = 2, T.value = 3)
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

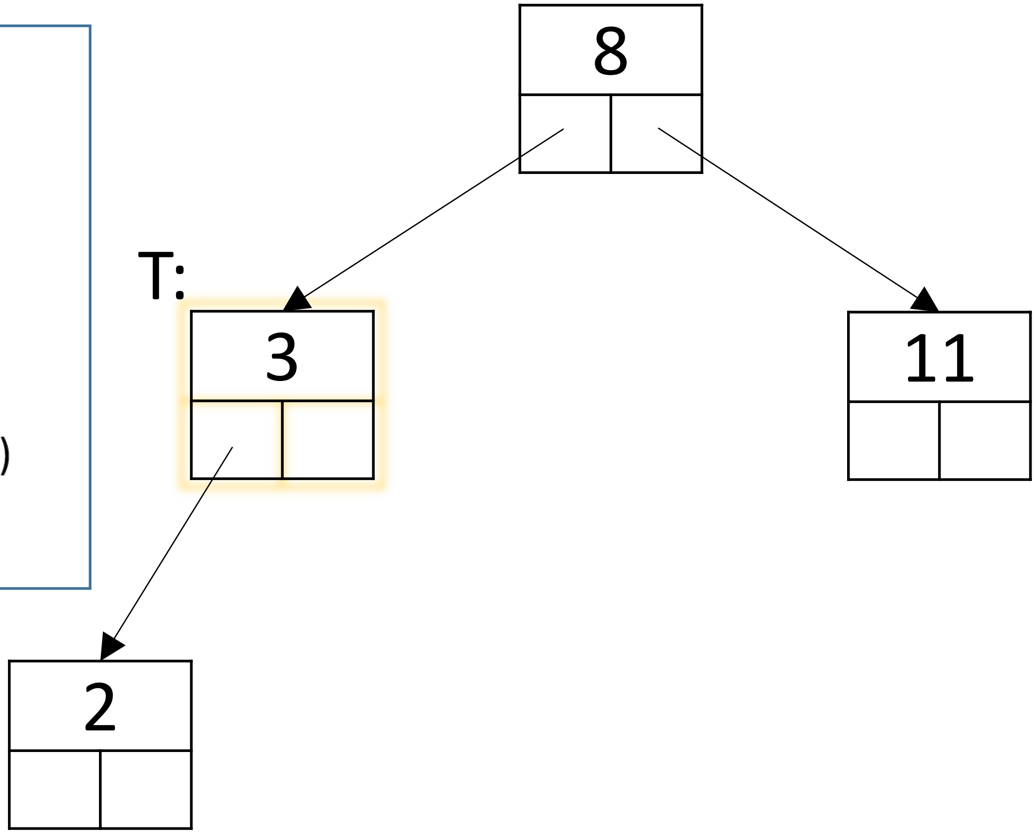
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

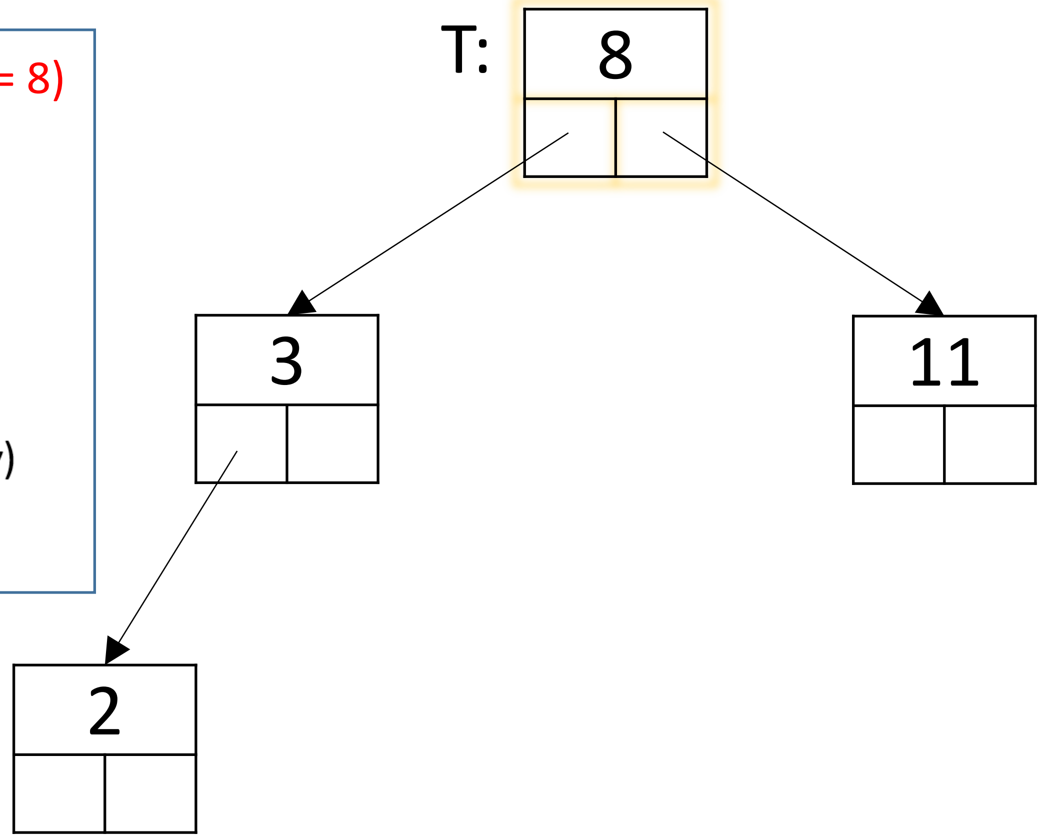
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

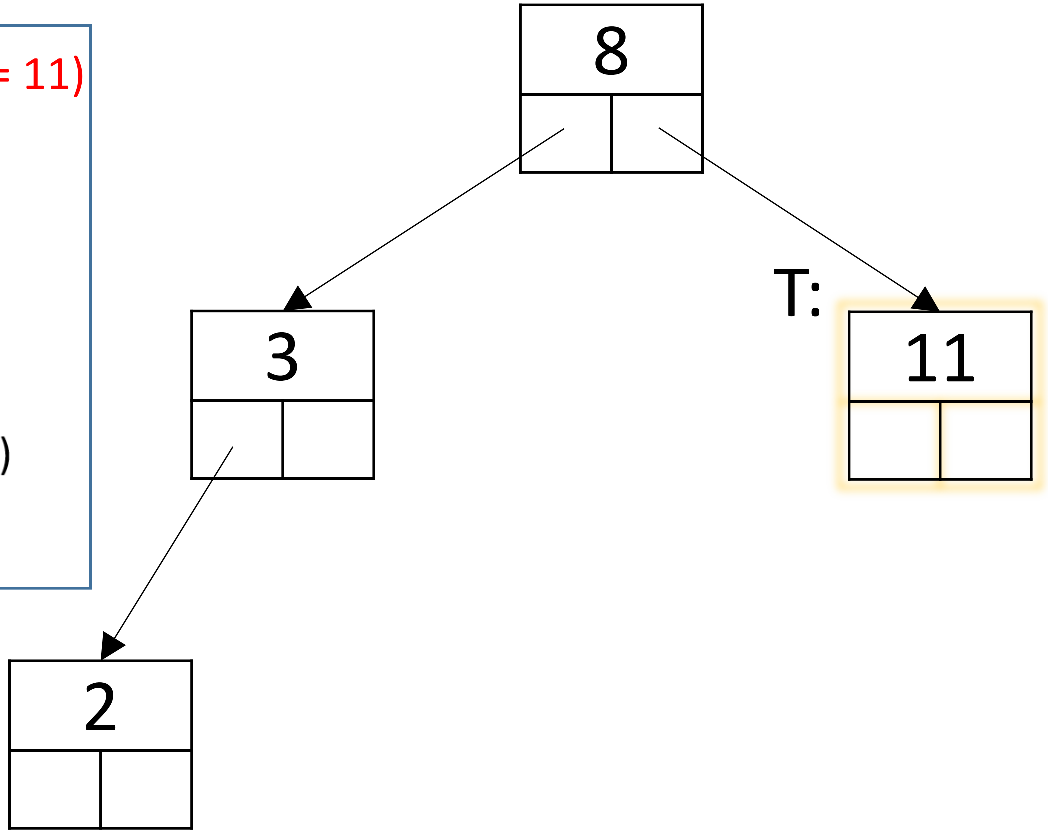
```
def insert(T, v): (v = 9, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

Sequence of values: 8 3 11 2 9 5

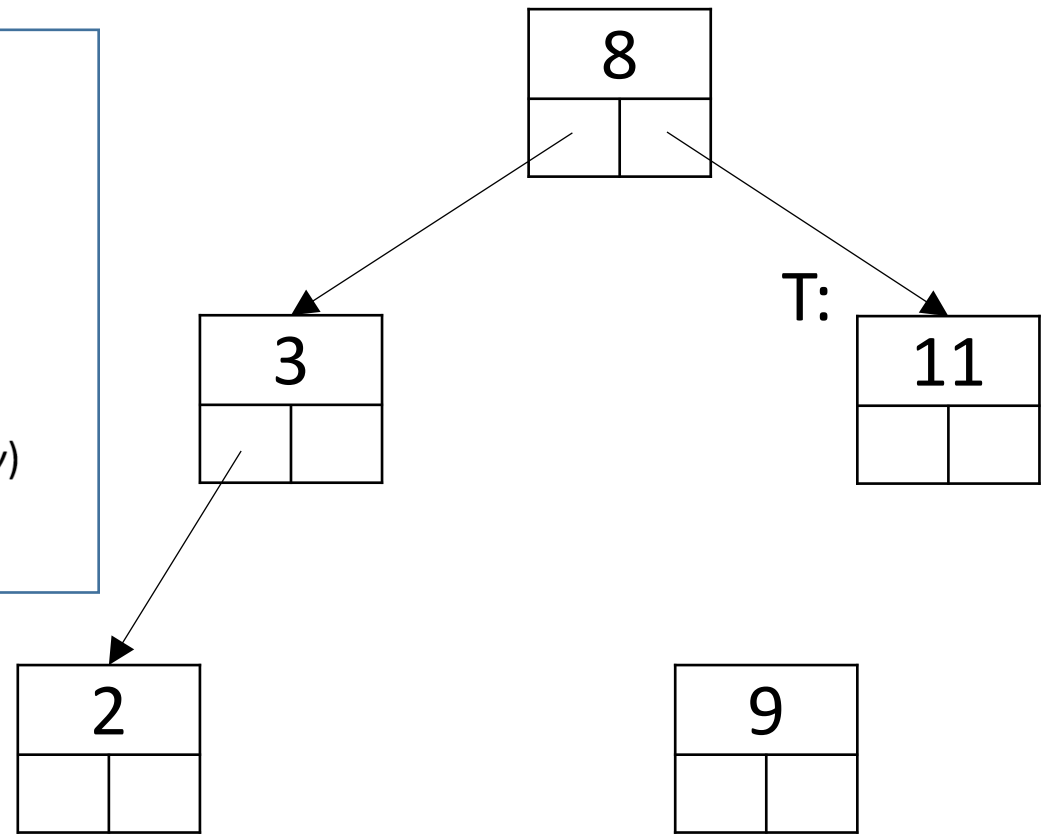
```
def insert(T, v):(v = 9, T.value = 11)
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

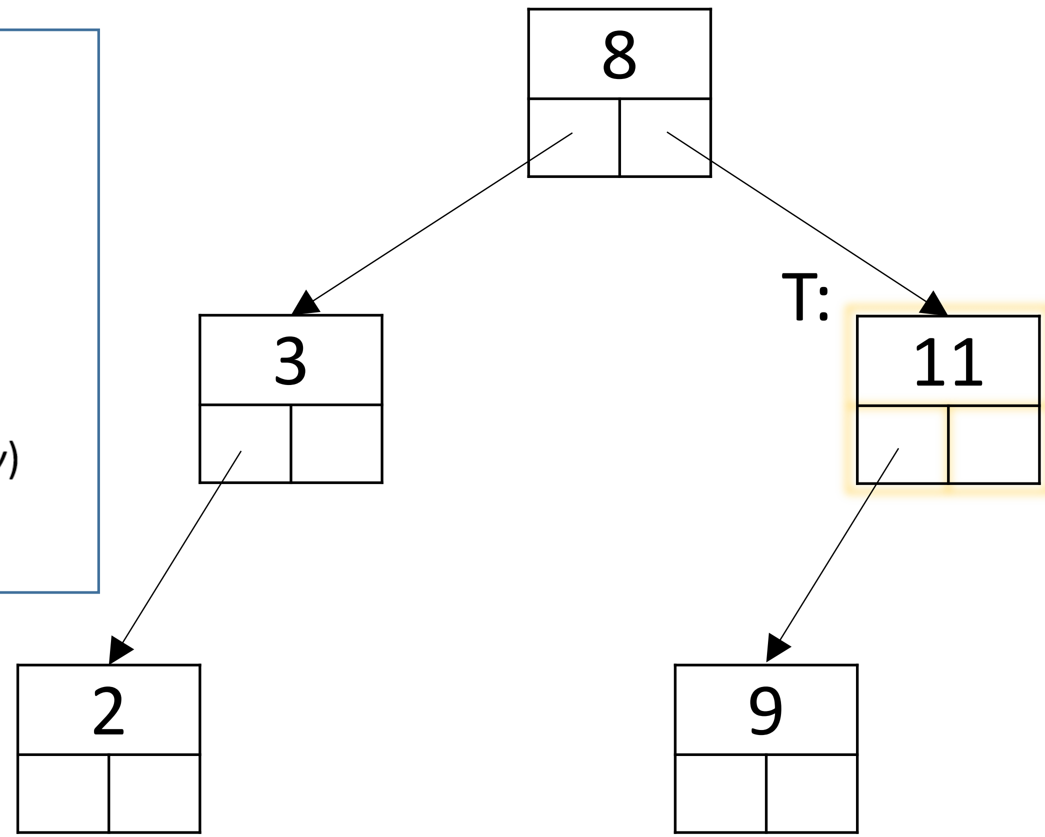
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

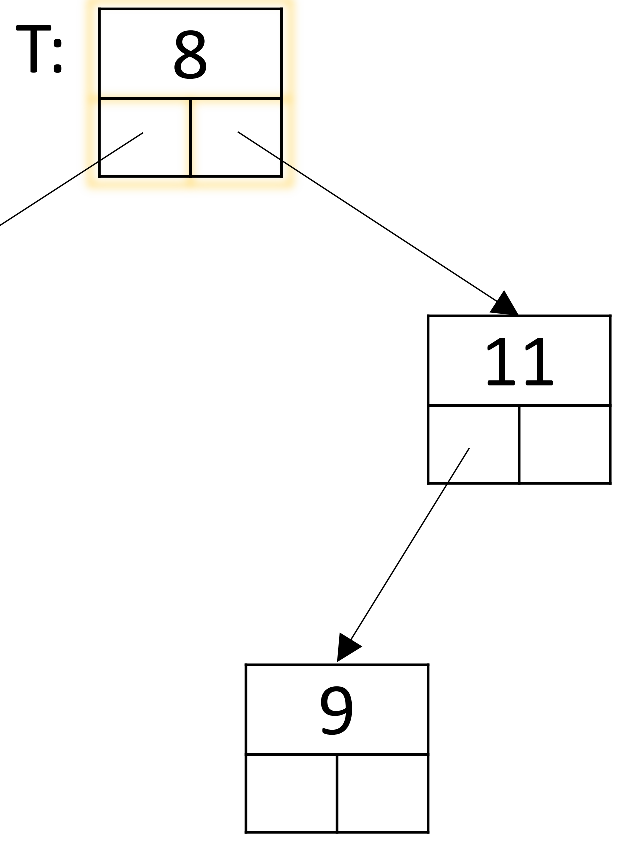
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

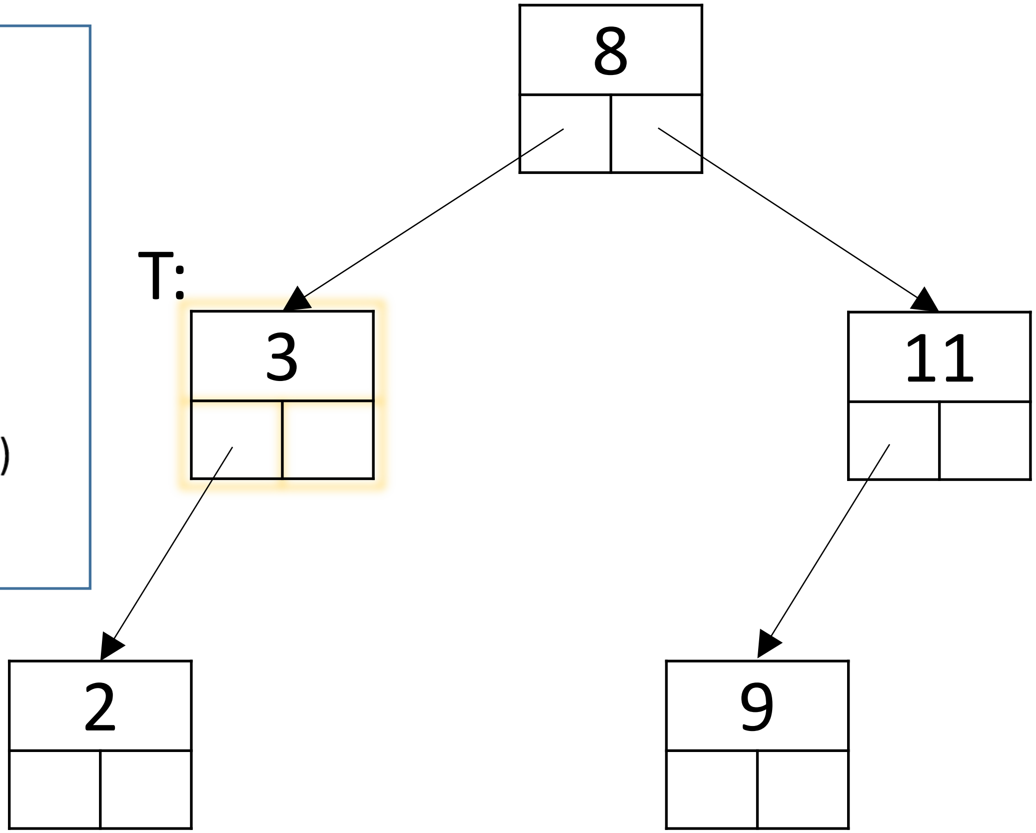
```
def insert(T, v):(v = 5, T.value = 8)
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

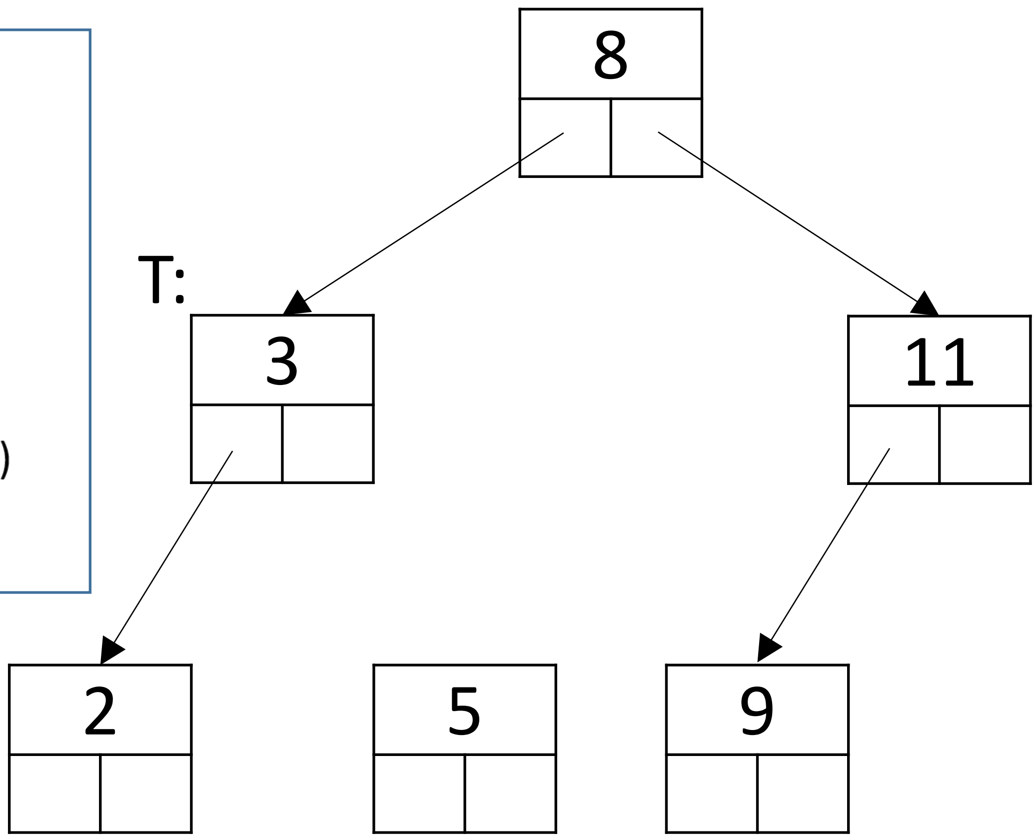
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

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

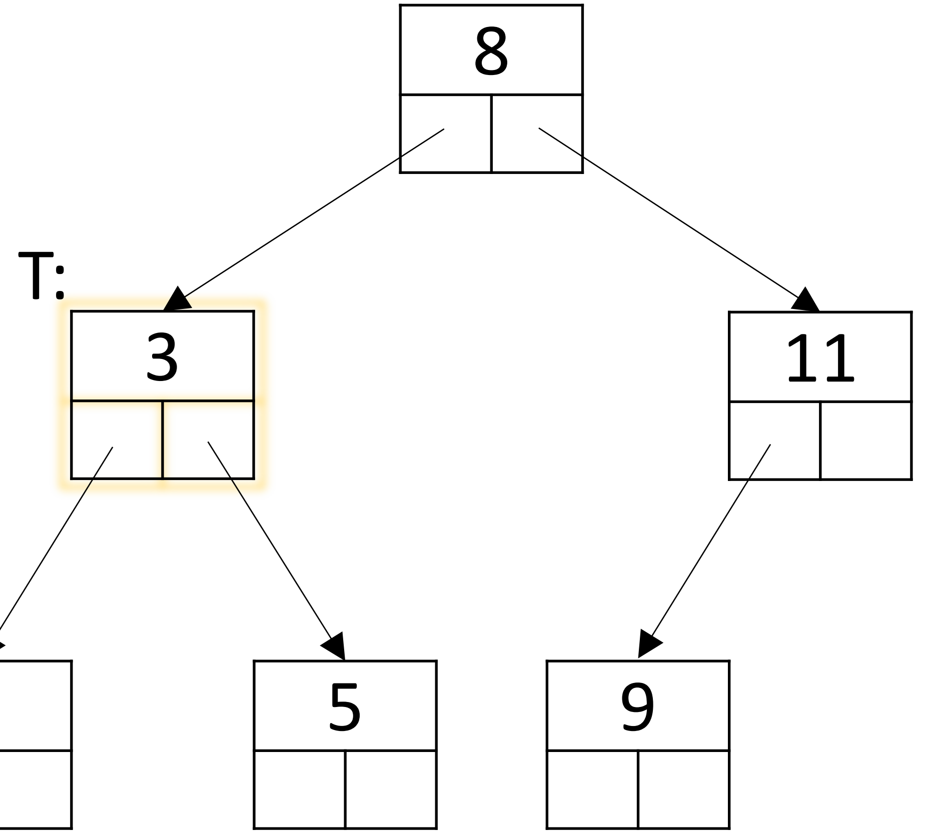
```
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```



Constructing a BST: Example

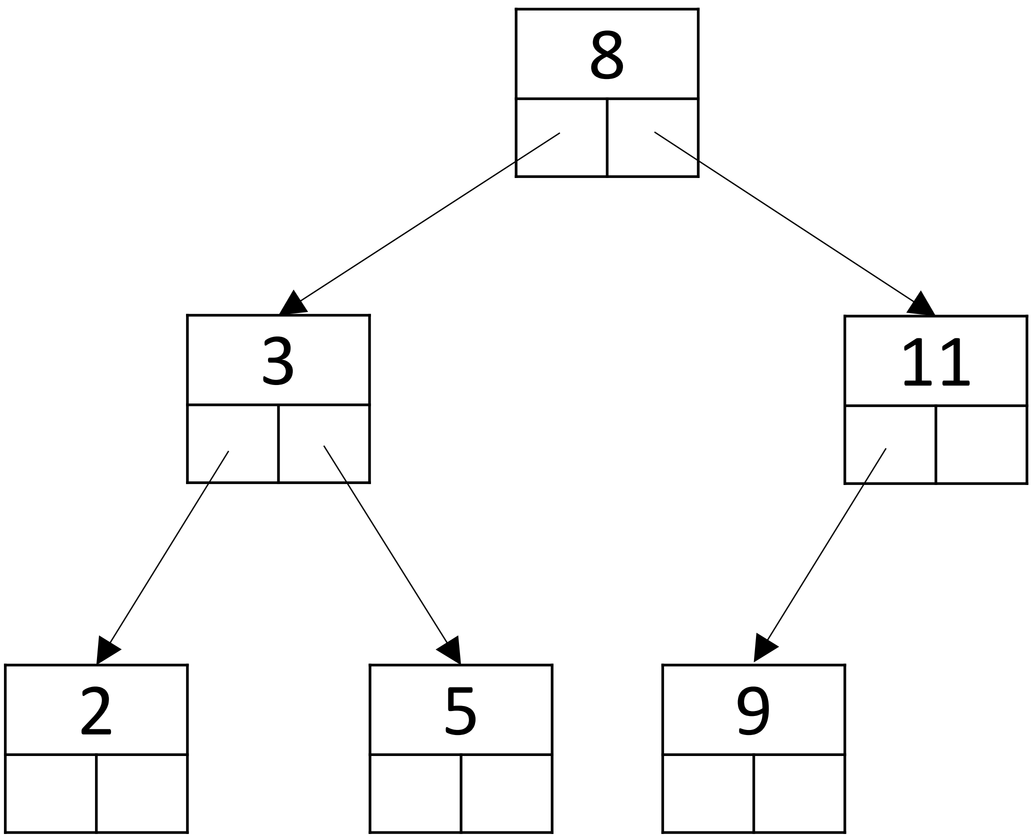
Sequence of values: 8 3 11 2 9 5

```
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



Whiteboard Exercise

Create a BST from this sequence: 10, 7, 23, 4, 14, 5

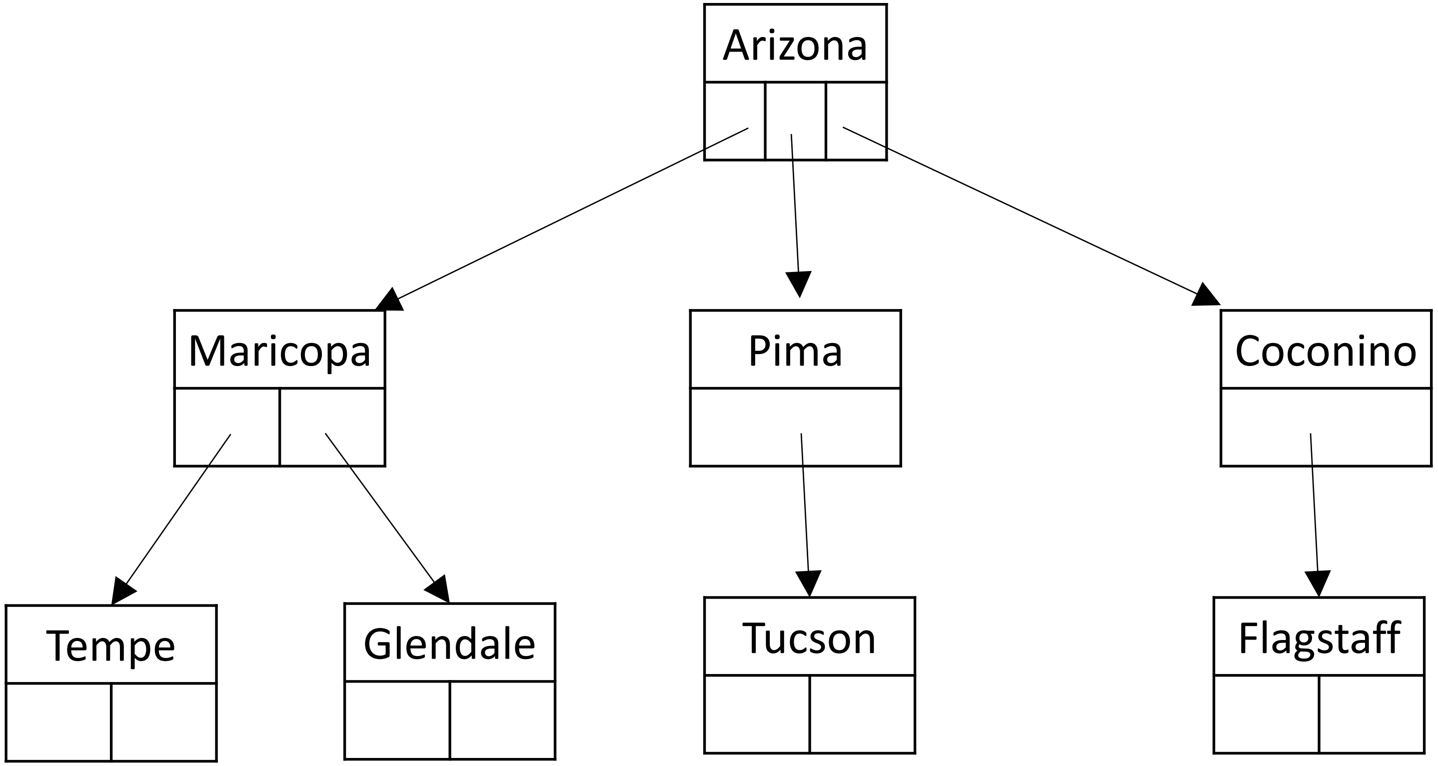
```
# Algo for creating a  BST
def insert(T, v):
    if T == None:
        return Node(v)
    if v < T.value:
        T.left = insert(T.left, v)
    elif v > T.value:
        T.right = insert(T.right, v)

    return T
```

Exercise – ICA23

Do all problems.

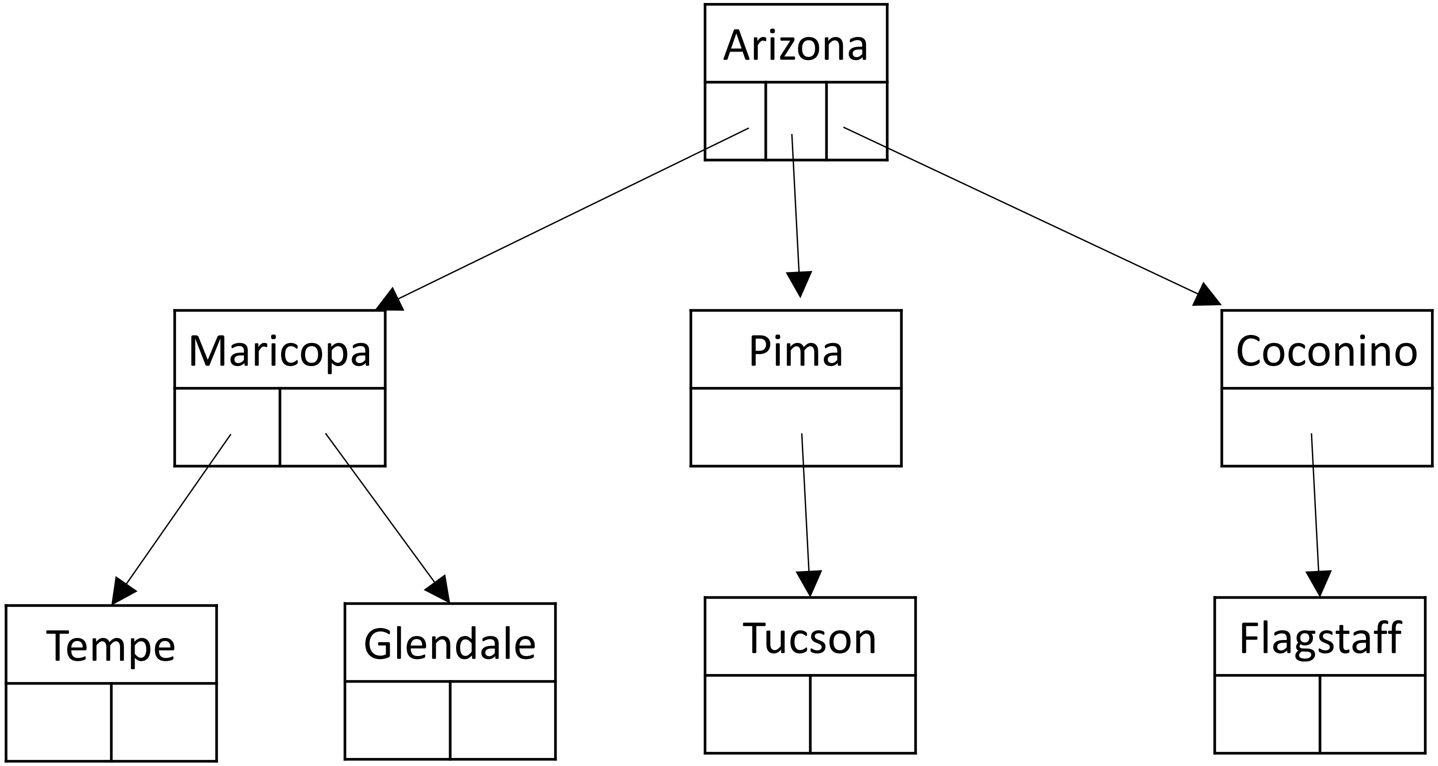
We’ll then continue with the lecture.

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?

Whiteboard Exercise



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

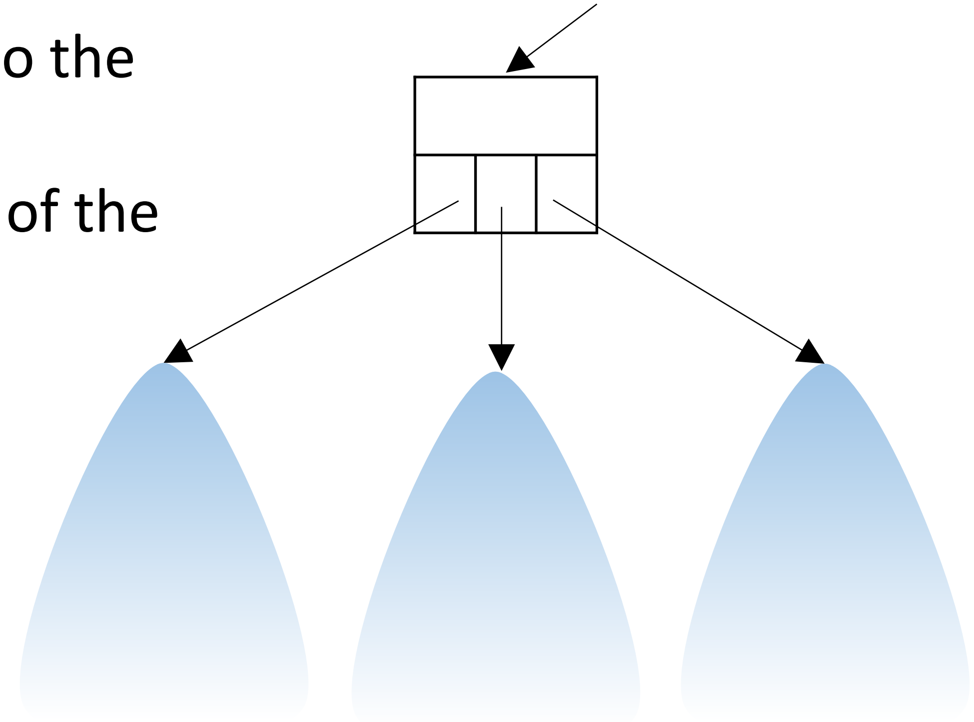
tree traversals

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 (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

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 all but last child
  - then process the node
  - then traverse the last child

"in" – visit node in between (somewhere)

- *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:

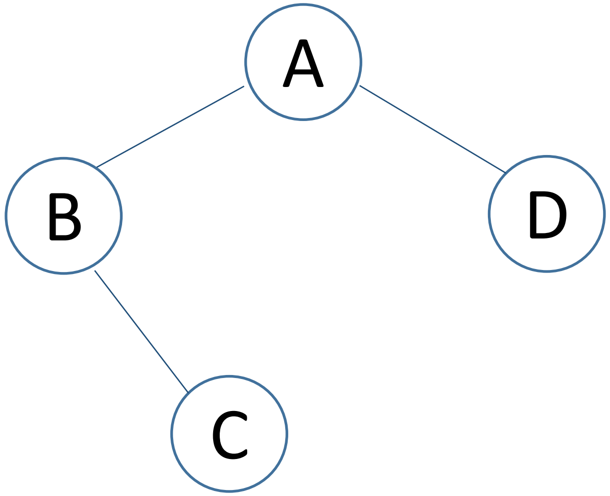
(where's the base case?)

Visit the node

Recurse on node's Left subtree

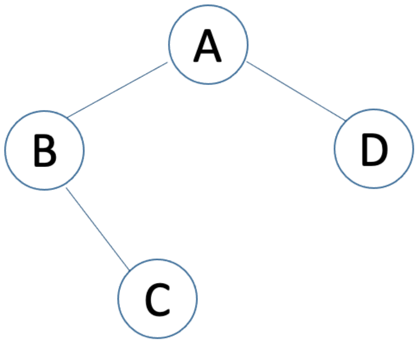
Recurse on node's Right subtree

Ex:



A B C D

Trace of preorder traversal

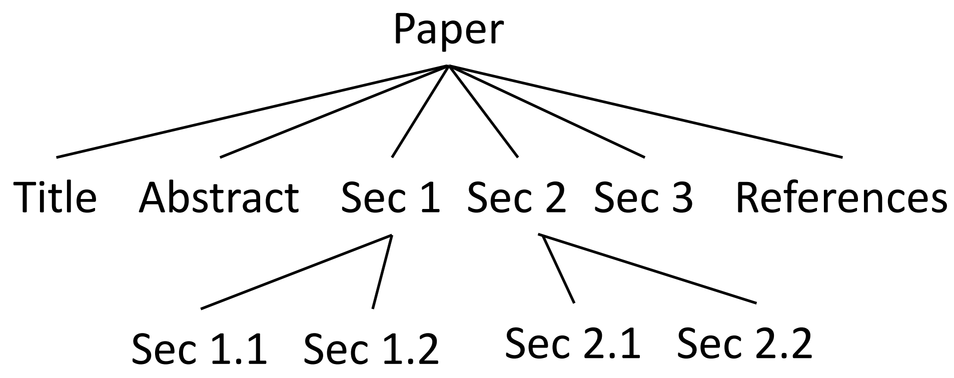


Output	Trace
A	Call preorder(A)
	Visit (print)
	Left (call preorder(B))
B	Visit (print)
	Left – return immediately
	Right (call preorder(C))
C	Visit (print)
	Left – return
	Right – return
	Right(call preorder(D))
D	Visit (print)
	Left – return
	Right – return

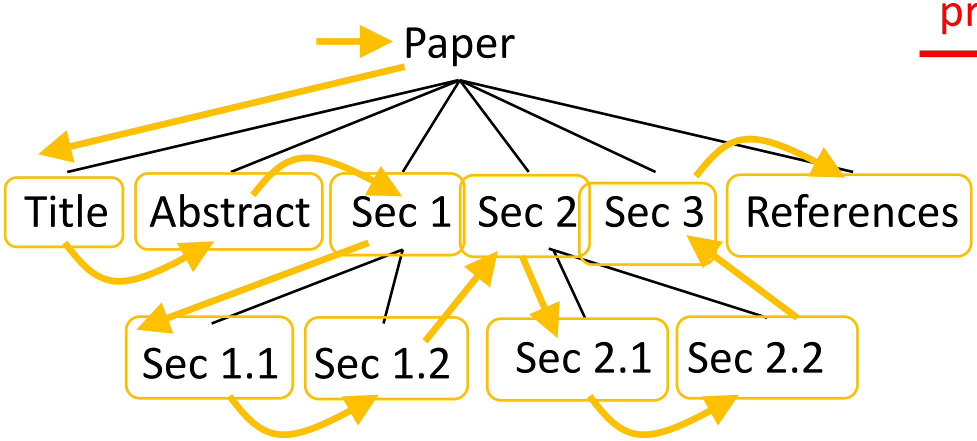
Preorder traversal (n-ary)

```
def preorder(T):
    if T == None:
        return
    process(T._value)
    for i in range(len(T._children)):
        preorder(T._children[i])
```

Preorder traversal: Example



Preorder traversal: Example



preorder

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

Inorder traversal

Algorithm:

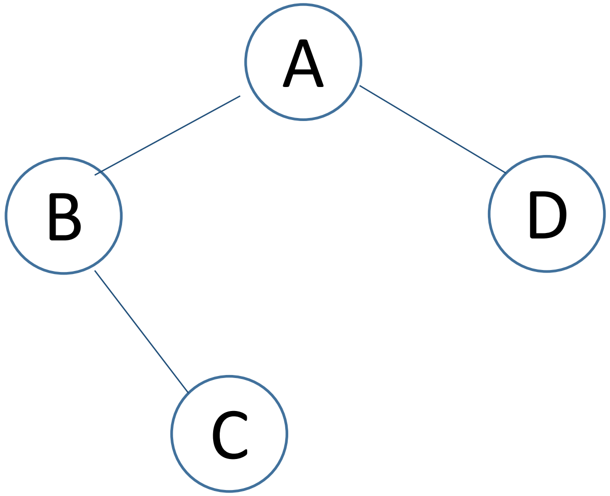
(where's the base case?)

Recurse on node's Left subtree

Visit node

Recurse on node's Right subtree

Ex:



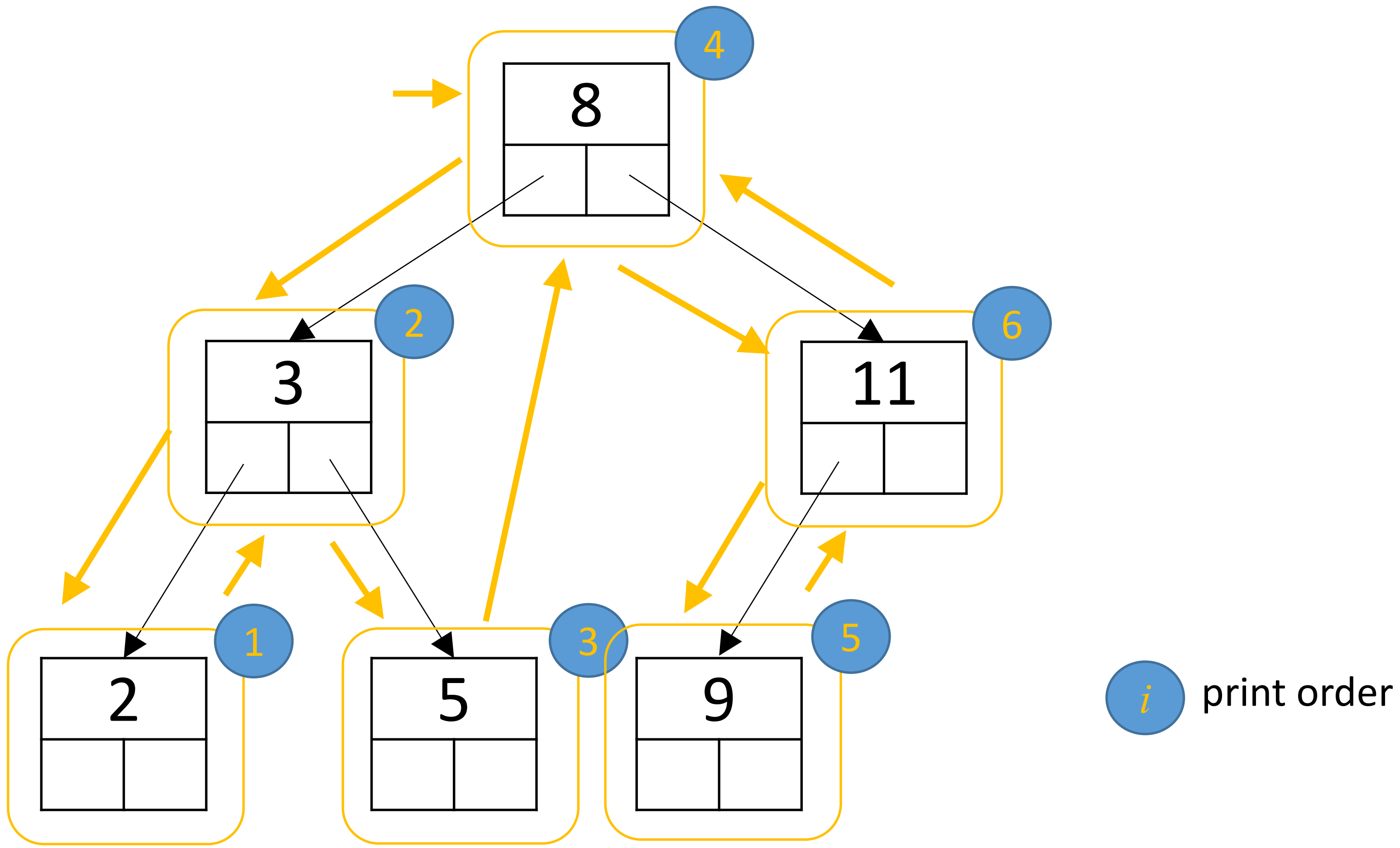
B C A D

Inorder traversal (binary trees)

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

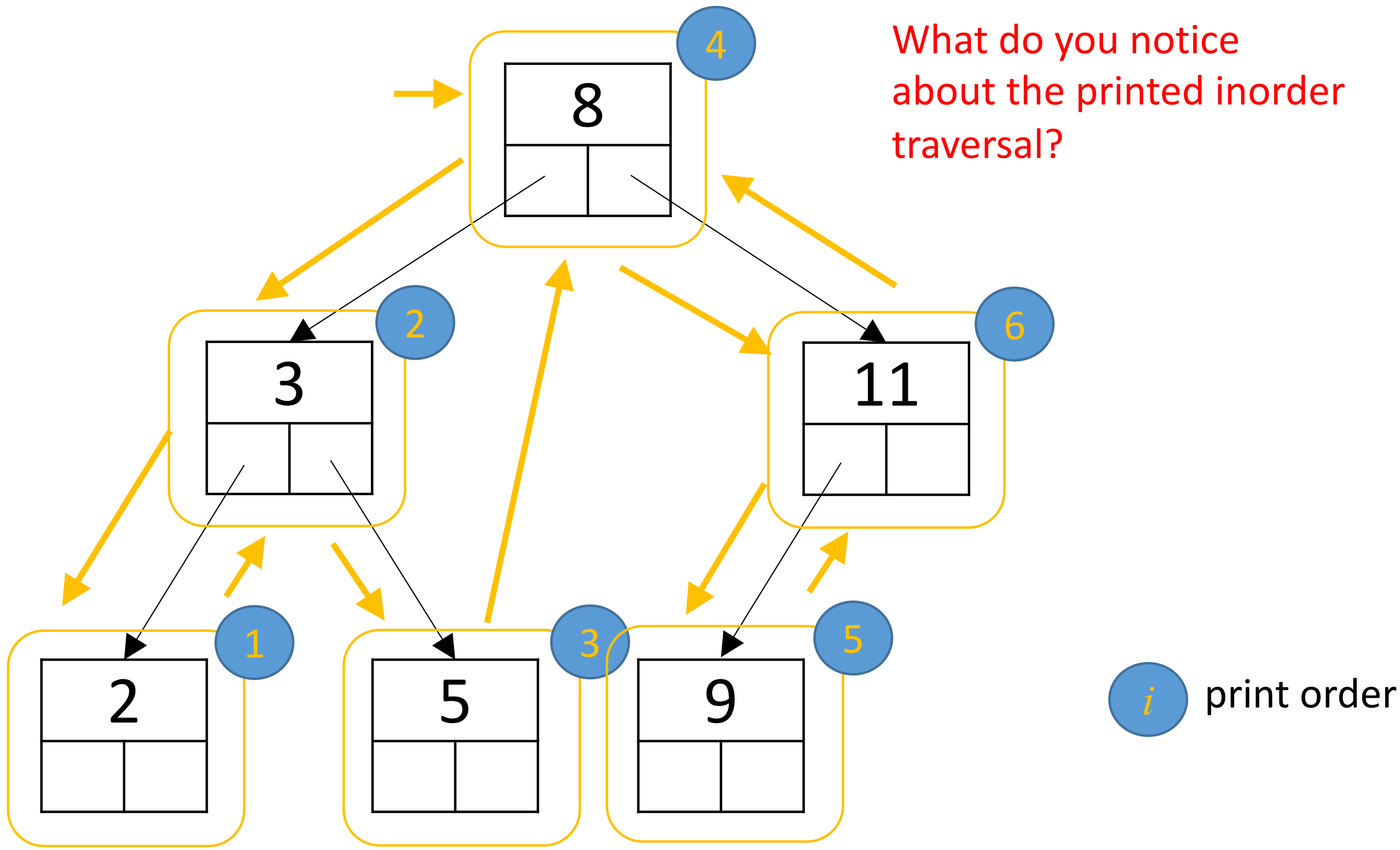
Inorder traversal: Example

Print out the values in a BST in sorted order



Inorder traversal: Example

Print out the values in a BST in sorted order



What do you notice about the printed inorder traversal?

Postorder traversal

Algorithm:

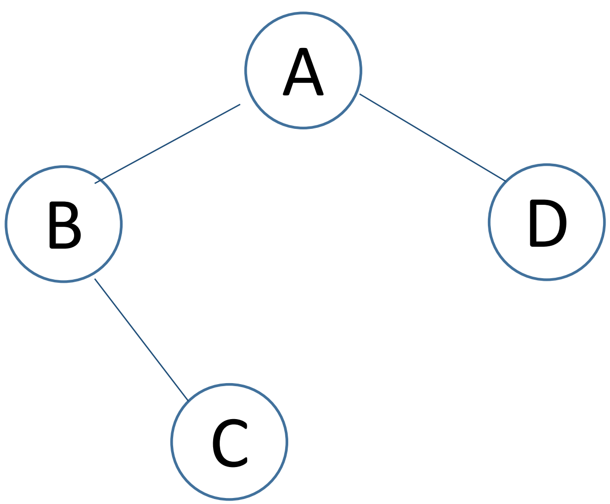
(where's the base case?)

Recurse on node's Left subtree

Recurse on node's Right subtree

Visit node

Ex:



C B D A

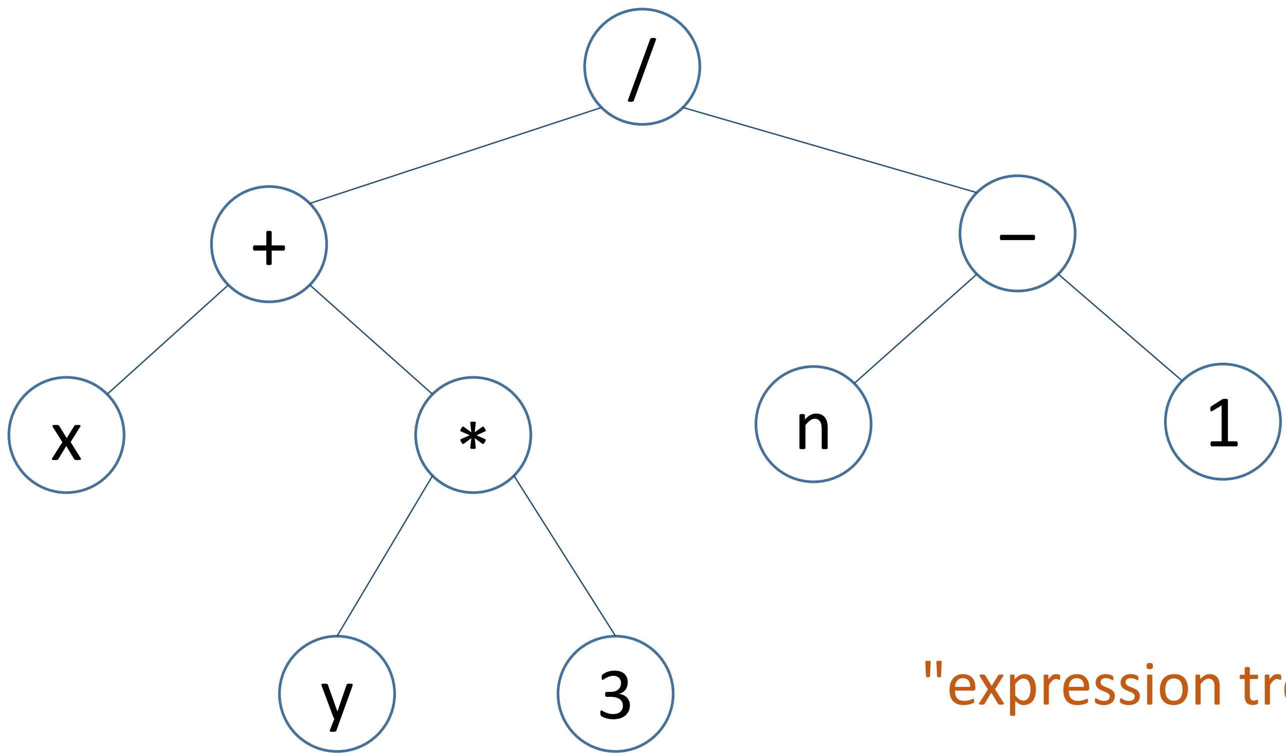
Postorder traversal (n-ary)

```
def postorder(T):
    if T == None:
        return
    for i in range(len(T._children)):
        postorder(T._children[i])   # visit all children first
    process(T._value)
```

Postorder traversal: Example

Evaluate: (x + y * 3) / (n – 1)

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

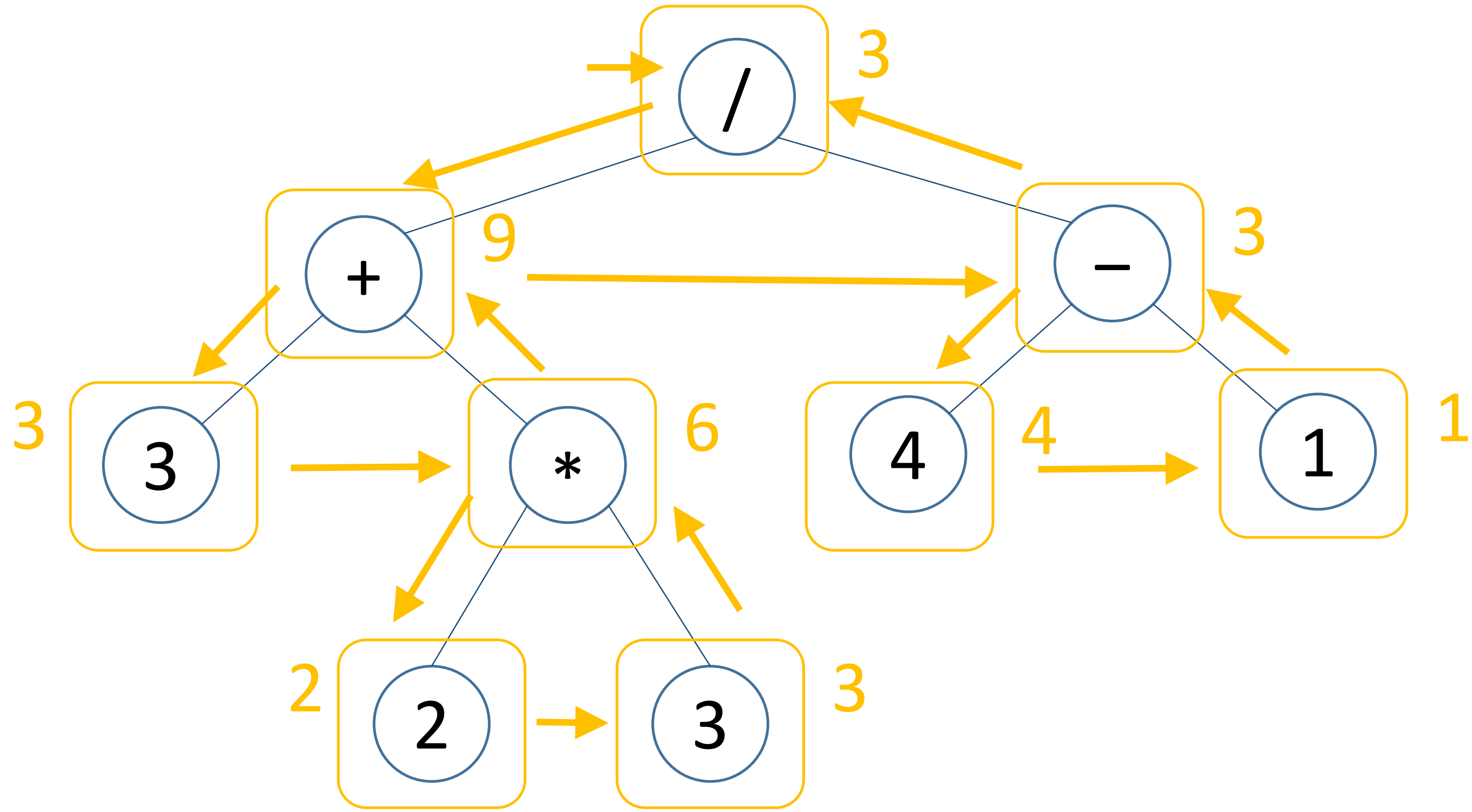


"expression tree"

Postorder traversal: Example

Evaluate: (x + y * 3) / (n – 1)

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



Exercise- ICA-24

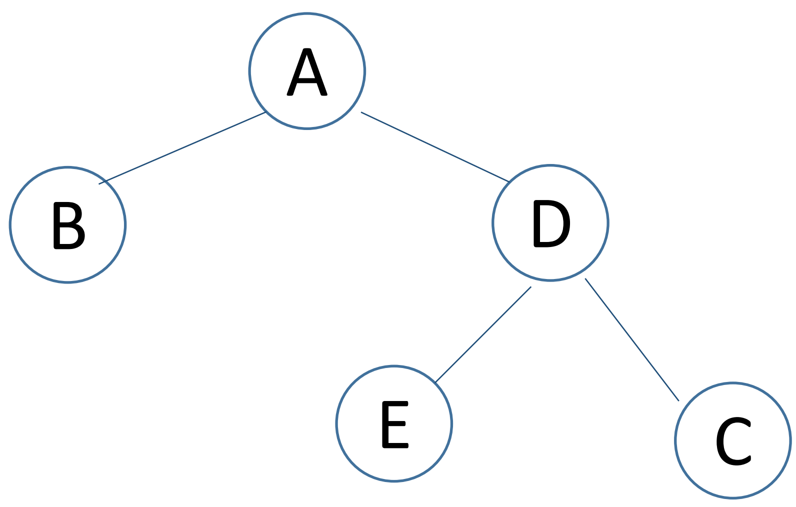
Do all problems.

Trees $\leftrightarrow$ traversals

What is the preorder traversal of this tree?

Inorder?

Postorder?

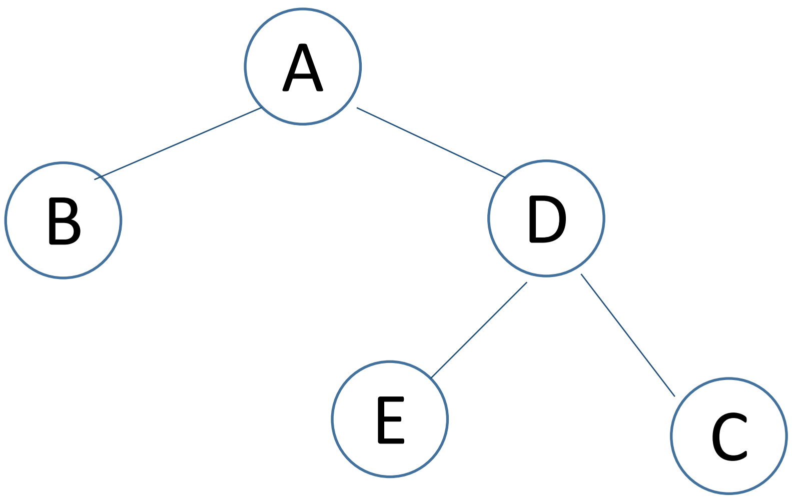


Whiteboard Exercise

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?

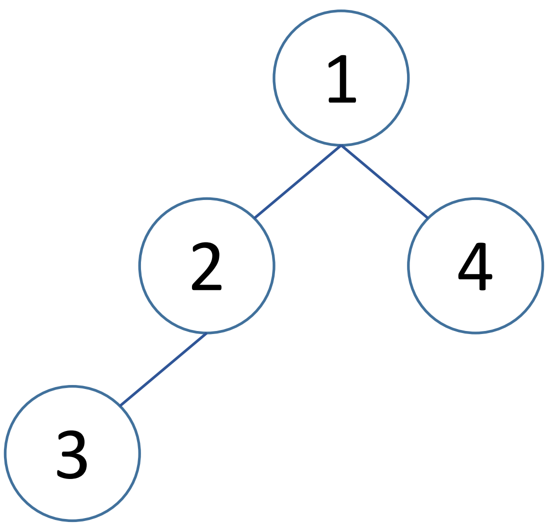
  I.e., given a traversal, can we figure out the tree?

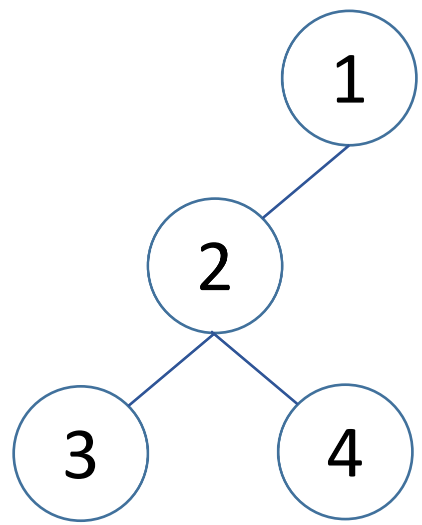
    preorder: 1 2 3 4

Trees ↔ traversals

- The two trees below have the same preorder traversal.

Preorder traversal = 1 2 3 4





Trees ↔ 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 figure out an algo to do it

Trees <-> Traversals

- Create a unique tree given these traversals:
  - Preorder: 5 3 9 7 4
  - Inorder: 3 5 7 9 4
- Algorithm

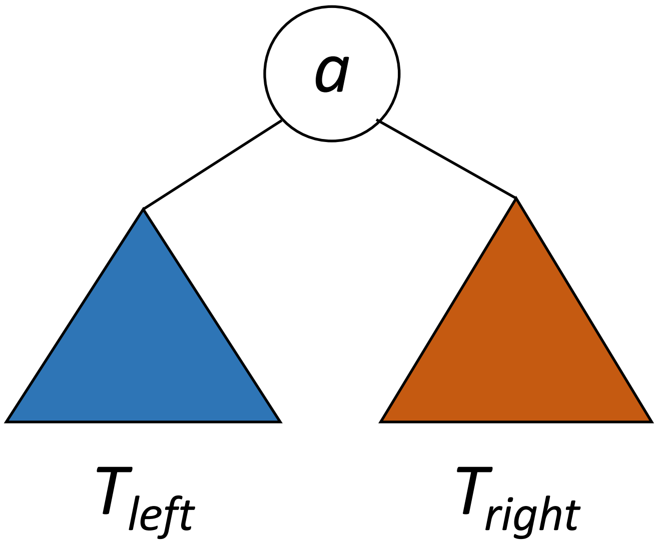
1. Identify the root
2. Identify the nodes that go to the left and right of the root
3. Add the node and create the *new* pre- and inorder traversals for each side
4. Do steps 1 and 2 on each subtree

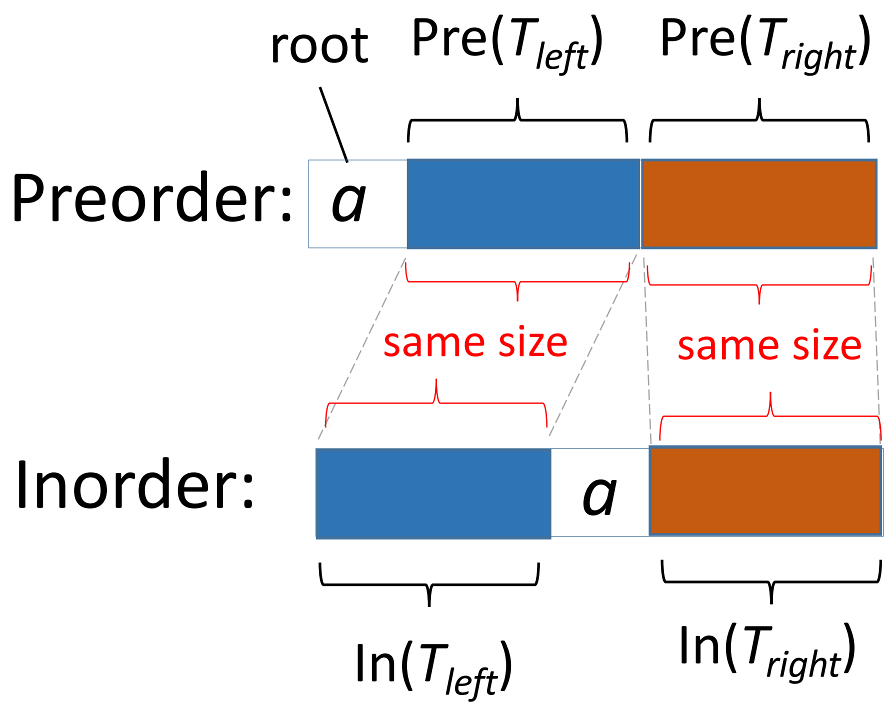
Let’s do this together.

ICA-25

Do problems 1 through 3.

Trees ↔ traversals





Trees ↔ 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 ↔ 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:

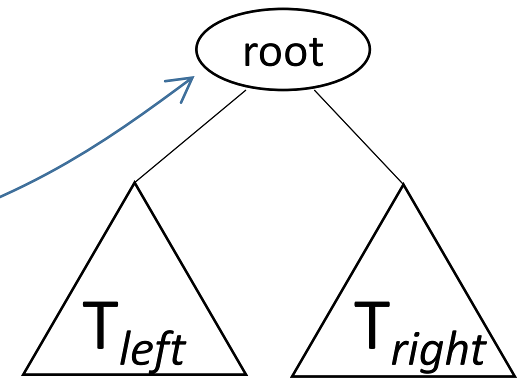
traversals_to_tree(preorder_left, inorder_left) $\rightarrow$ T *left*

traversals_to_tree(preorder_right, inorder_right) $\rightarrow$ T *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) → T_{left}

traversals_to_tree(preorder_right, inorder_right) → T_{right}

ICA-25

Do problems 4 through 6.

more traversals

Consider: game playing

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

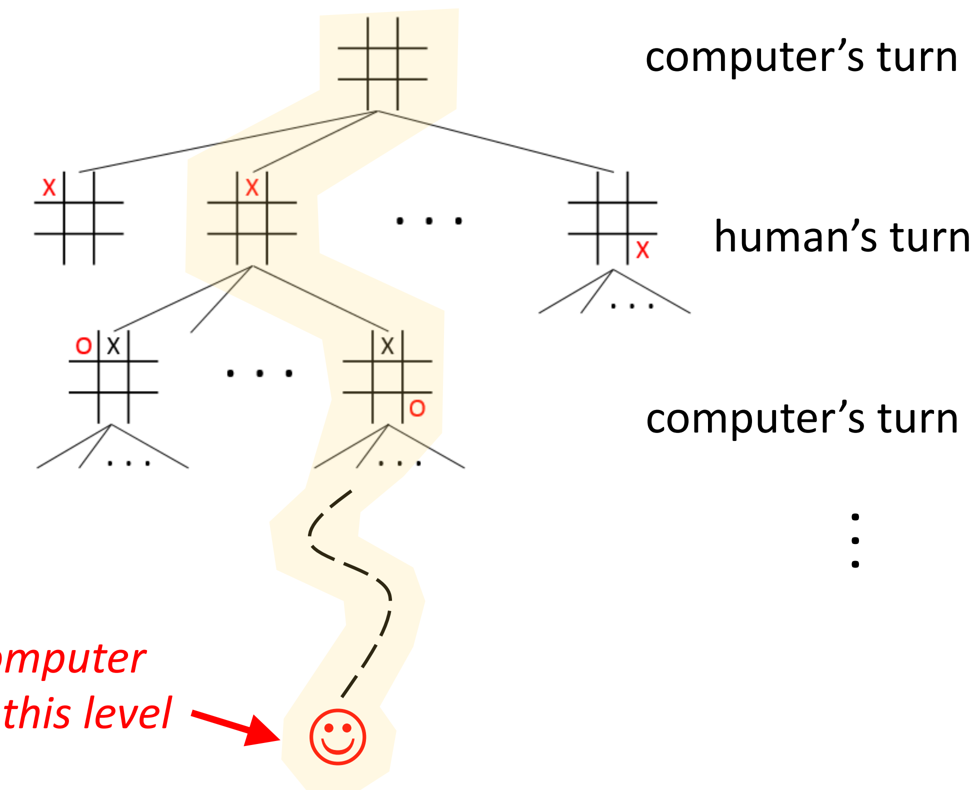
How does this work?

Consider: game playing

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

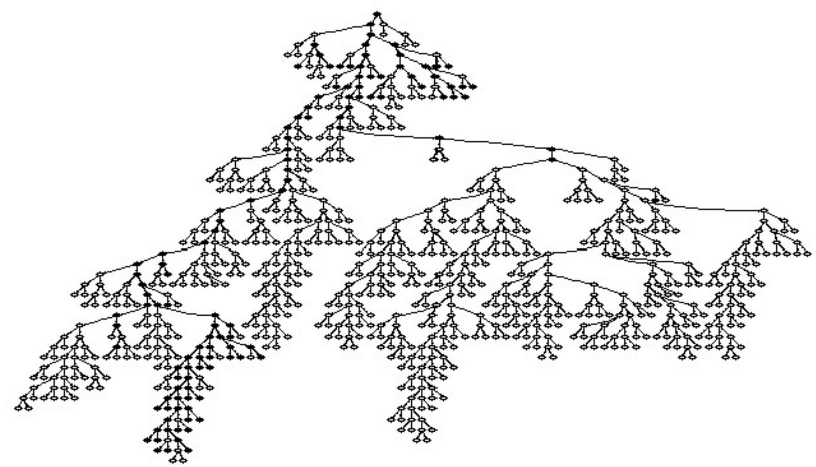
Generate successive levels of board positions

- 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



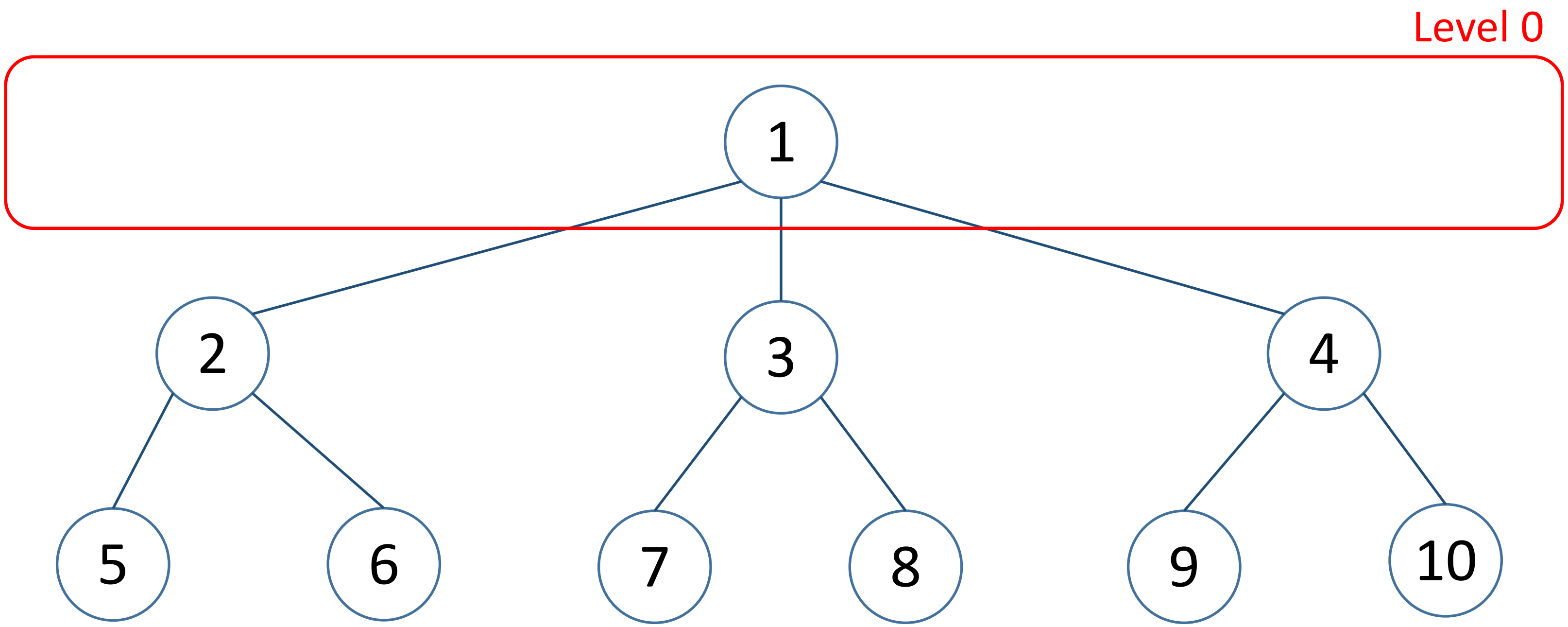
Consider: game playing

- 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.

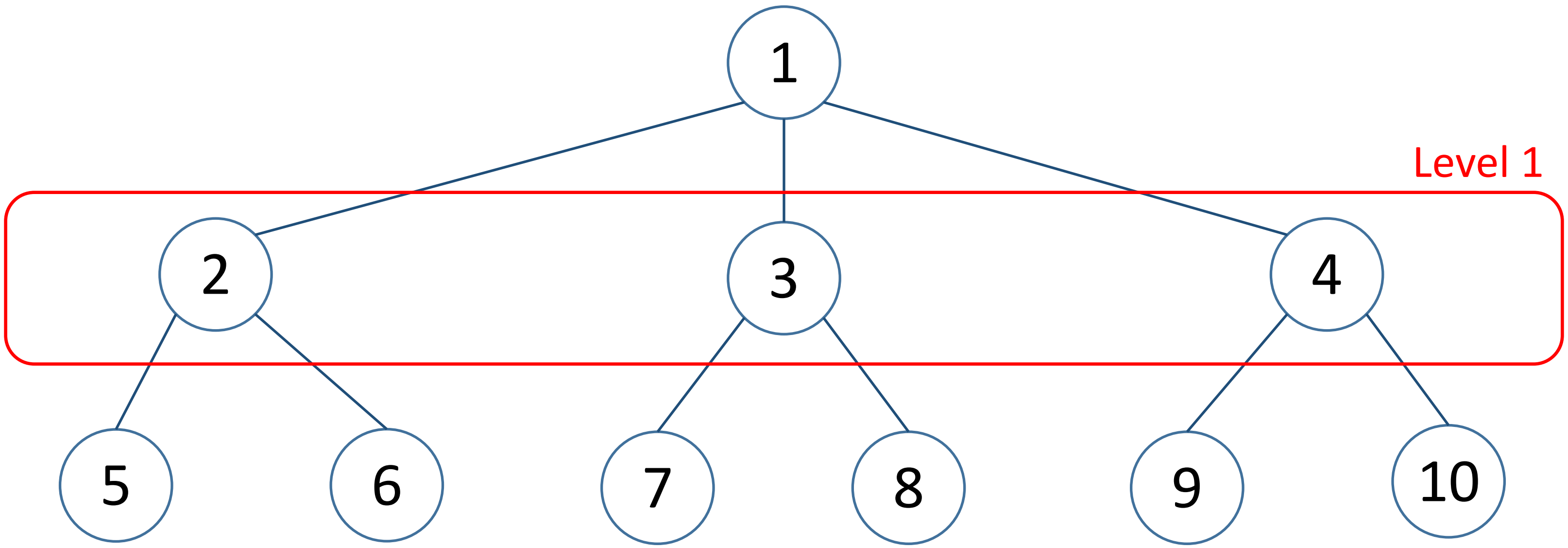


A game tree

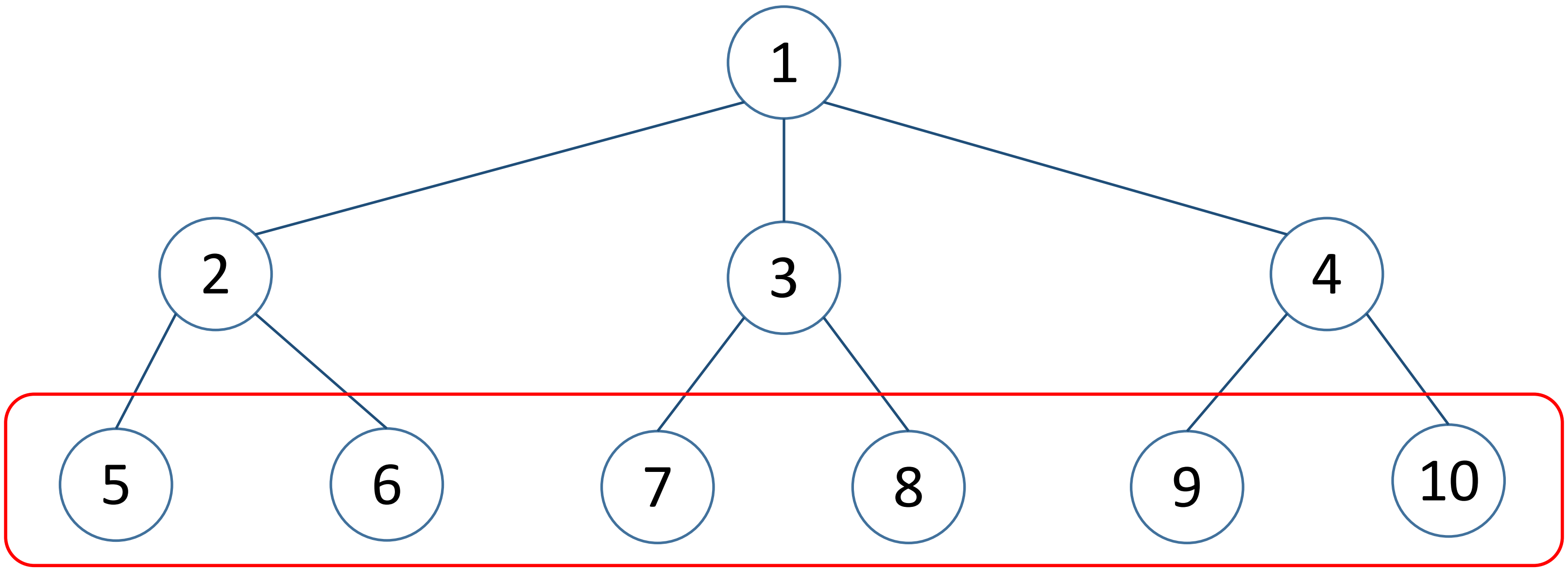
Level-by-level tree traversal



Level-by-level tree traversal

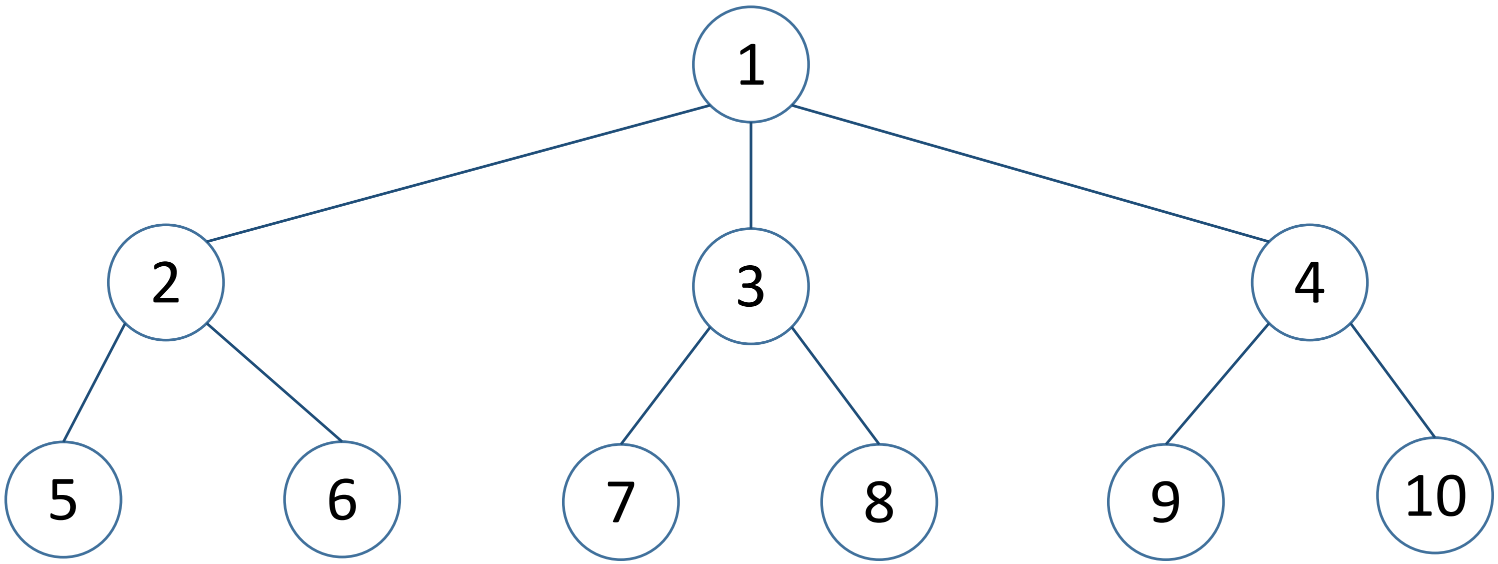


Level-by-level tree traversal



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

Breadth-first tree traversal



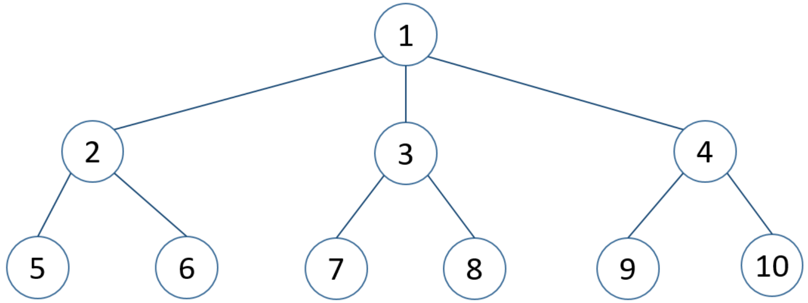
Breadth-first traversal order:

1 2 3 4 5 6 7 8 9 10

Breadth-first tree traversal

- Previous traversals used recursion.

- We will use a Queue for breadth-first search.

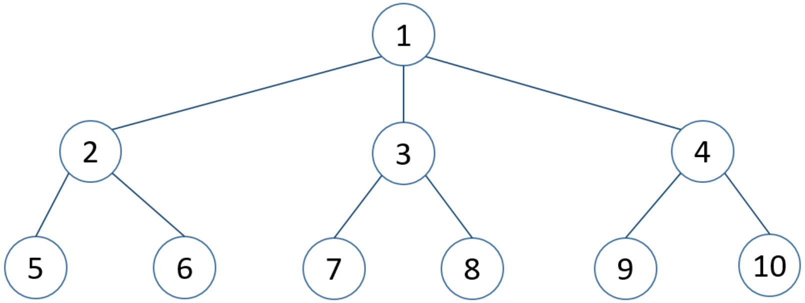


Breadth-first tree traversal

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 ”stack” in such traversals is most often the implicit stack used in recursion (the runtime stack frames built by Python when running a program)
  - the deeper levels of the tree are explored first
  - this is referred to as *depth-first traversal*
    - *Preorder, inorder, postorder*

Trees: summary

- An n-ary tree represents a hierarchy

- They show up in all kinds of contexts
  - nature (evolutionary trees)
  - organizations (org charts)
  - your folder/file structure
  - math expressions

- 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

Exercise- ICA-26

Do all problems.

ASCII encodings

- ASCII uses fixed length encodings:

char	ASCII dec value	ASCII binary value
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

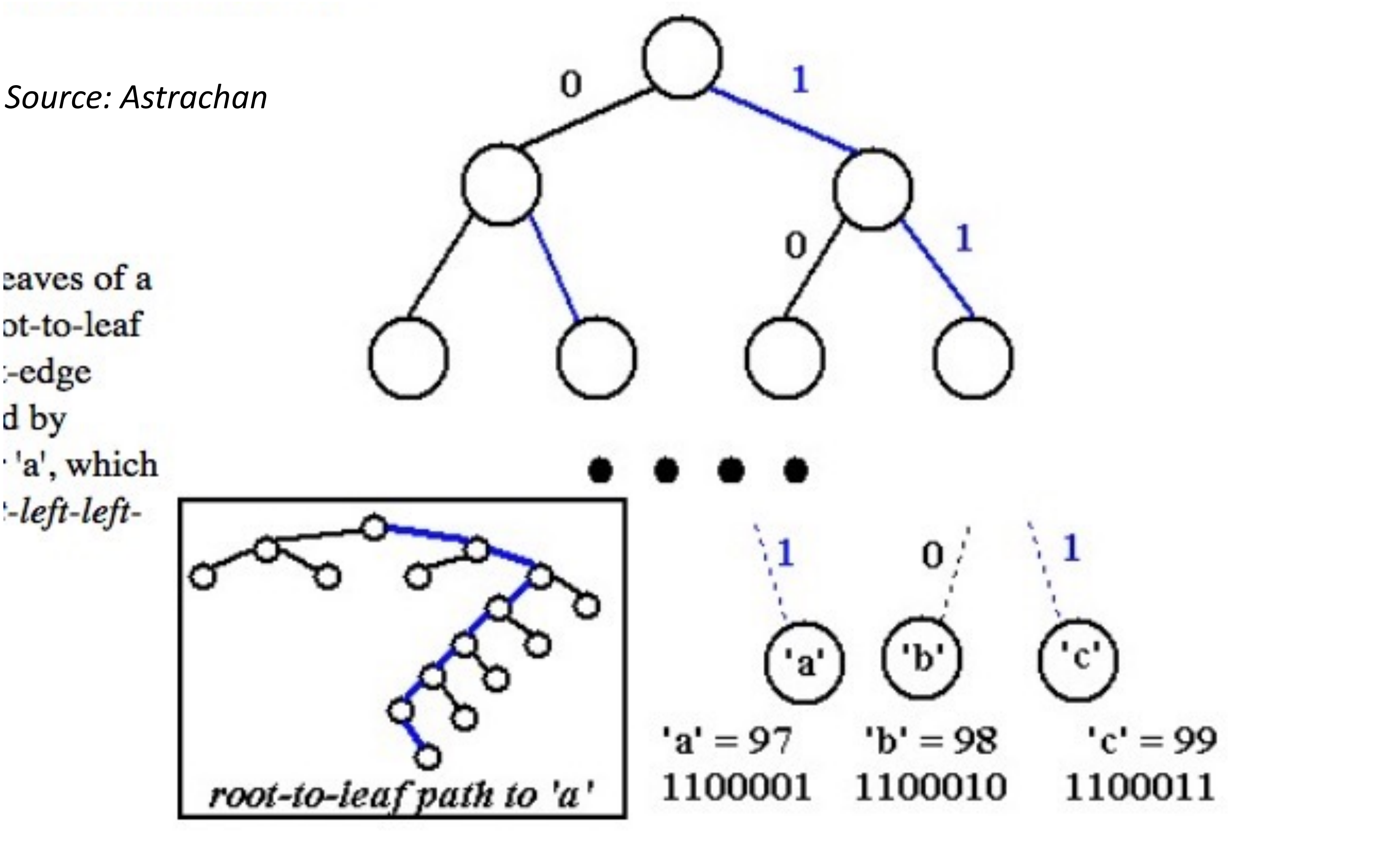
binary: 1101000 1101111 1110010 1110011 1100101 (group by 7's)
ASCI: 104 111 114 115 101

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

Huffman Coding

Source: Astrachan



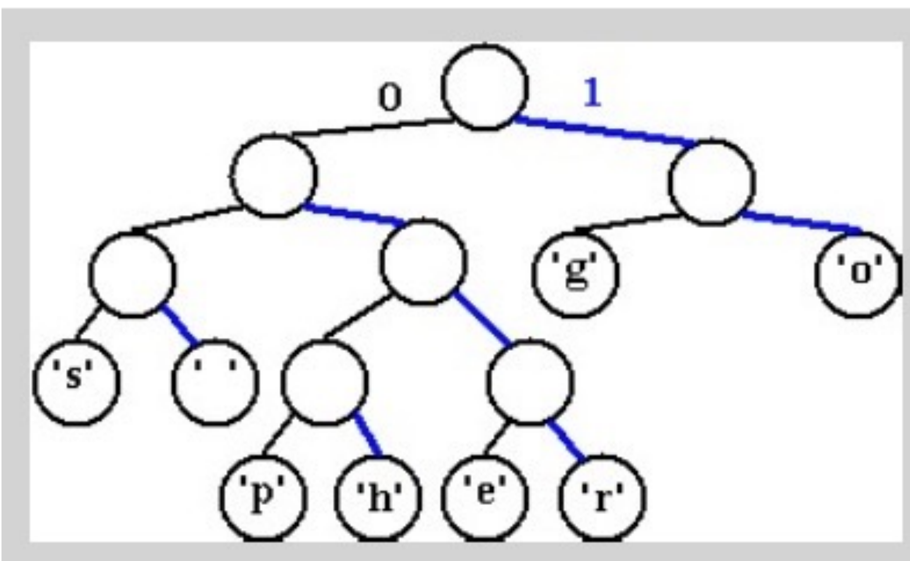
path: *right-right-left-left-left-left-right* *using the 0/1 convention*

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

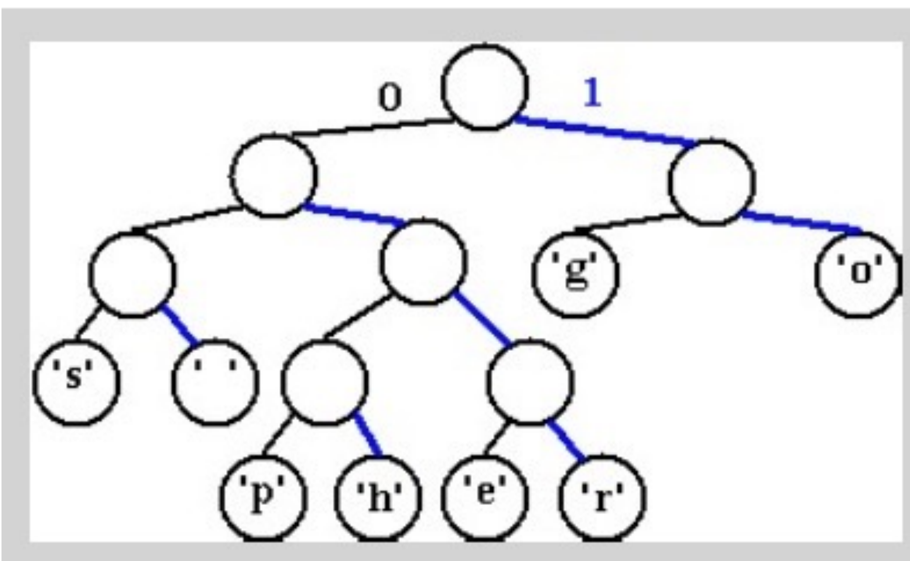
What would the encoding be for hope?

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?
0101 11 0100 0110

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

Breadth-first tree traversal

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