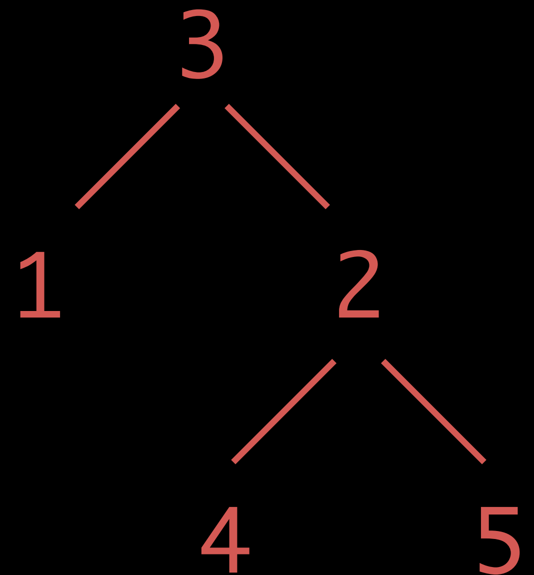
Trees

```
def tree(root_label, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
        return [root_label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not(branches(tree))
```

Trees

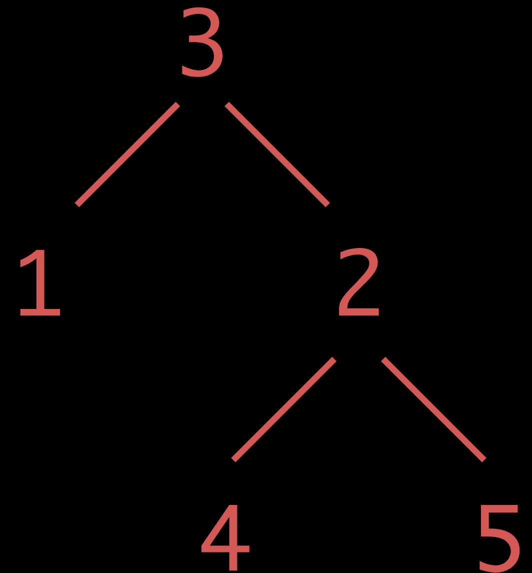
```
>>> t = tree(3, [tree(1), tree(2, [tree(4), tree(5)])])
>>> t -> [3, [1], [2, [4], [5]]]

>>> label(t) -> 3

>>> branches(t) -> [[1], [2, [4], [5]]]

>>> label(branches(t)[0]) -> 1

>>> label(branches(t)[1]) -> 2

>>> r = branches(t)[1]
>>> label(branches(r)[0]) -> 4
>>> label(branches(r)[1]) -> 5
```



Trees (with objects)

```
1  # Trees with objects (manual tree construction)
2
3  class Tree:
4      def __init__(self, label, branches=[]):
5          self.label = label
6          self.branches = branches
7
8      def __repr__(self):
9          if self.branches:
10             return 'T[{0}, {1}]'.format(self.label, repr(self.branches))
11         else:
12             return 'T[{0}]'.format(repr(self.label))
13
14 t = Tree(3)
15 t.branches = [Tree(1),Tree(2)]
16 t.branches[1].branches = [Tree(4),Tree(5)]
17 print(t)
```

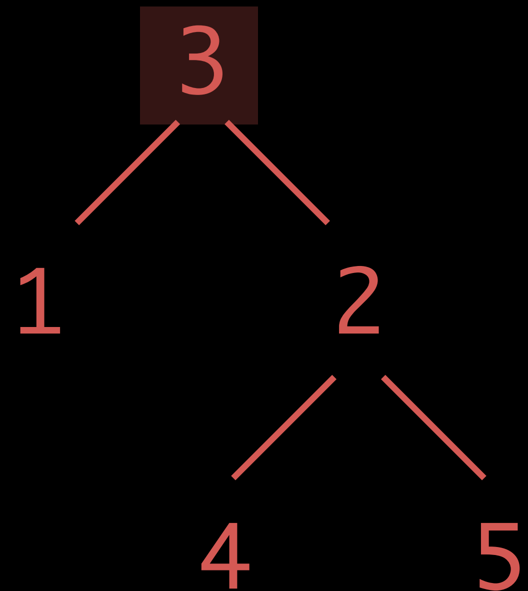
```
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Trees (with objects)

```
1  # Trees with objects (manual tree construction)
2
3  class Tree:
4      def __init__(self, label, branches=[]):
5          self.label = label
6          self.branches = branches
7
8      def __repr__(self):
9          if self.branches:
10             return 'T[{0}, {1}]'.format(self.label, repr(self.branches))
11         else:
12             return 'T[{0}]'.format(repr(self.label))
13
14 t = Tree(3)
15 t.branches = [Tree(1),Tree(2)]
16 t.branches[1].branches = [Tree(4),Tree(5)]
17 print(t)
```

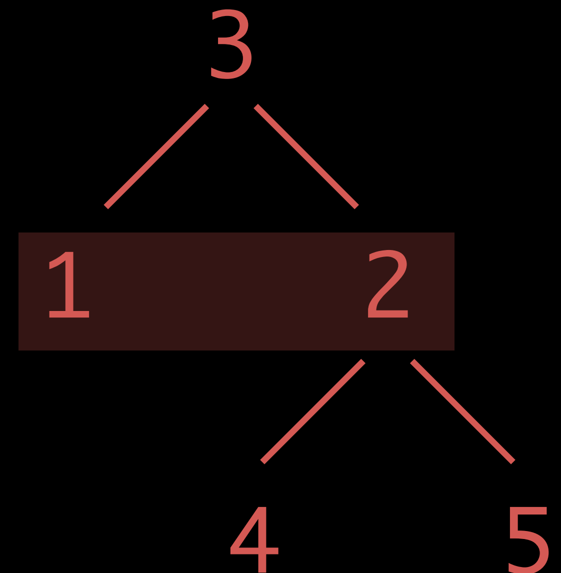
```
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Trees (with objects)

```
 1  # Trees with objects (manual tree construction)
 2
 3  class Tree:
 4      def __init__(self, label, branches=[]):
 5          self.label = label
 6          self.branches = branches
 7
 8      def __repr__(self):
 9          if self.branches:
10              return 'T[{0}, {1}]'.format(self.label, repr(self.branches))
11          else:
12              return 'T[{0}]'.format(repr(self.label))
13
14  t = Tree(3)
15  t.branches = [Tree(1),Tree(2)]
16  t.branches[1].branches = [Tree(4),Tree(5)]
17  print(t)
```

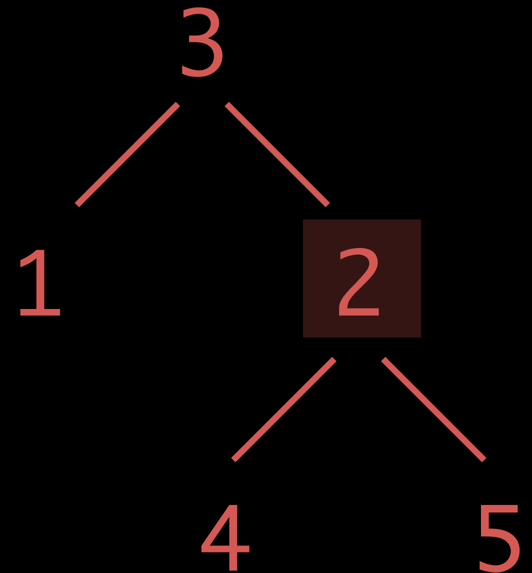
```
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Trees (with objects)

```
1  # Trees with objects (manual tree construction)
2
3  class Tree:
4      def __init__(self, label, branches=[]):
5          self.label = label
6          self.branches = branches
7
8      def __repr__(self):
9          if self.branches:
10             return 'T[{0}, {1}]'.format(self.label, repr(self.branches))
11         else:
12             return 'T[{0}]'.format(repr(self.label))
13
14 t = Tree(3)
15 t.branches = [Tree(1),Tree(2)]
16 t.branches[1].branches = [Tree(4),Tree(5)]
17 print(t)
```

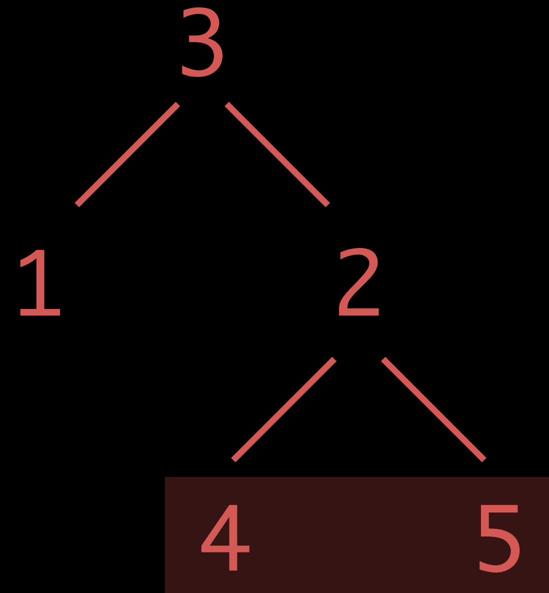
```
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Trees (with objects)

```
1  # Trees with objects (manual tree construction)
2
3  class Tree:
4      def __init__(self, label, branches=[]):
5          self.label = label
6          self.branches = branches
7
8      def __repr__(self):
9          if self.branches:
10             return 'T[{0}, {1}]'.format(self.label, repr(self.branches))
11         else:
12             return 'T[{0}]'.format(repr(self.label))
13
14 t = Tree(3)
15 t.branches = [Tree(1),Tree(2)]
16 t.branches[1].branches = [Tree(4),Tree(5)]
17 print(t)
```

```
T[3, [T[1], T[2, [T[4], T[5]]]]]
```

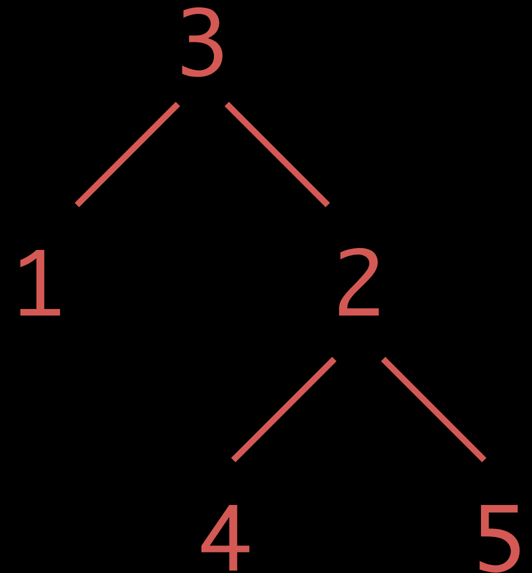


Trees (with objects)

```
1  # Trees with objects (manual tree construction)
2
3  class Tree:
4      def __init__(self, label, branches=[]):
5          self.label = label
6          self.branches = branches
7
8      def __repr__(self):
9          if self.branches:
10             return 'T[{0}, {1}]'.format(self.label, repr(self.branches))
11         else:
12             return 'T[{0}]'.format(repr(self.label))
13
14 t = Tree(3)
15 t.branches = [Tree(1),Tree(2)]
16 t.branches[1].branches = [Tree(4),Tree(5)]
17 print(t)
```

```
T[3, [T[1], T[2, [T[4], T[5]]]]]
```

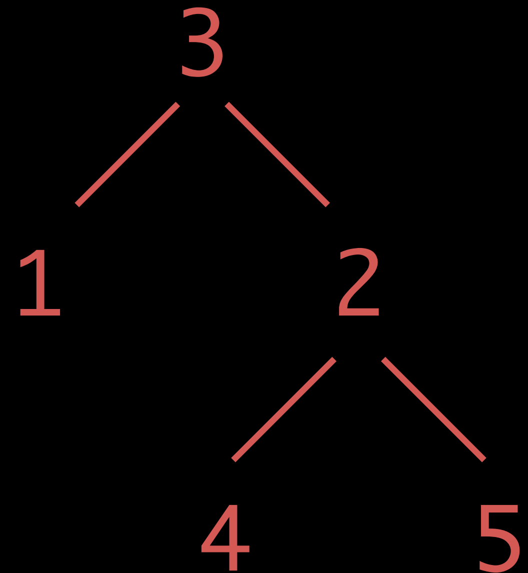
poor data abstraction



Trees (with objects)

```
1  # Trees with objects (functional tree construction)
2
3  class Tree:
4      def __init__(self, label):
5          self.label = label
6          self.branches = []
7
8      # add child in the right-most branch
9      def add_child(self,val):
10         if not self.branches:
11             self.branches = [Tree(val)]
12         else:
13             self.branches.append(Tree(val))
14
15     # return subtree in location num (0-indexed)
16     def get_subtree(self,num):
17         # make sure that there is a child numbered num
18         if self.branches and num < len(self.branches):
19             return self.branches[num]
20         else:
21             return None
22
23 t = Tree(3)
24 t.add_child(1)
25 t.add_child(2)
26 c = t.get_subtree(1)
27 print(c)
28 c.add_child(4)
29 c.add_child(5)
30 print(t)
```

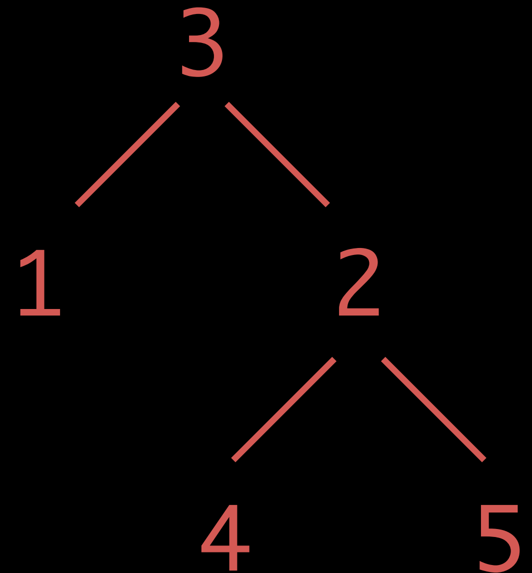
```
T[2]
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Trees (with objects)

```python
# Trees with objects (functional tree construction)

class Tree:
    def __init__(self, label):
        self.label = label
        self.branches = []

    # add child in the right-most branch
    def add_child(self,val):
        if not self.branches:
            self.branches = [Tree(val)]
        else:
            self.branches.append(Tree(val))

    # return subtree in location num (0-indexed)
    def get_subtree(self,num):
        # make sure that there is a child numbered num
        if self.branches and num < len(self.branches):
            return self.branches[num]
        else:
            return None

t = Tree(3)
t.add_child(1)
t.add_child(2)
c = t.get_subtree(1)
print(c)
c.add_child(4)
c.add_child(5)
print(t)
```

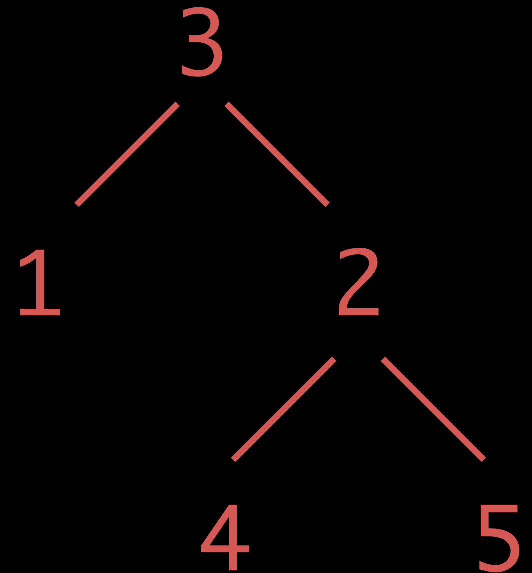
```
T[2]
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Trees (with objects)

```
# Trees with objects (functional tree construction)

class Tree:
    def __init__(self, label):
        self.label = label
        self.branches = []

    # add child in the right-most branch
    def add_child(self,val):
        if not self.branches:
            self.branches = [Tree(val)]
        else:
            self.branches.append(Tree(val))

    # return subtree in location num (0-indexed)
    def get_subtree(self,num):
        # make sure that there is a child numbered num
        if self.branches and num < len(self.branches):
            return self.branches[num]
        else:
            return None

t = Tree(3)
t.add_child(1)
t.add_child(2)
c = t.get_subtree(1)
print(c)
c.add_child(4)
c.add_child(5)
print(t)
```

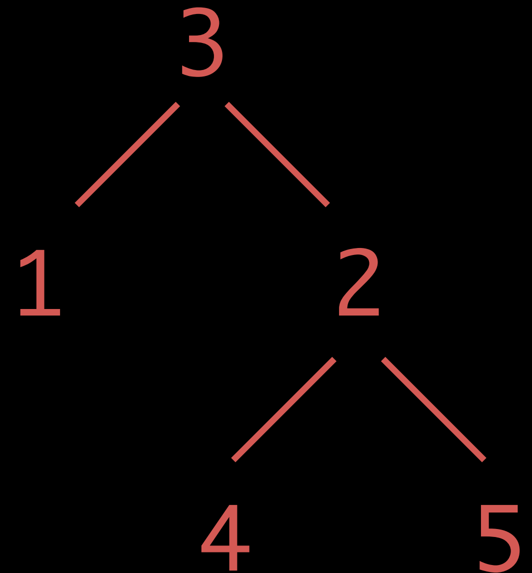
```
T[2]
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Trees (with objects)

```
# Trees with objects (functional tree construction)

class Tree:
    def __init__(self, label):
        self.label = label
        self.branches = []

    # add child in the right-most branch
    def add_child(self,val):
        if not self.branches:
            self.branches = [Tree(val)]
        else:
            self.branches.append(Tree(val))

    # return subtree in location num (0-indexed)
    def get_subtree(self,num):
        # make sure that there is a child numbered num
        if self.branches and num < len(self.branches):
            return self.branches[num]
        else:
            return None

t = Tree(3)
t.add_child(1)
t.add_child(2)
c = t.get_subtree(1)
print(c)
c.add_child(4)
c.add_child(5)
print(t)
```

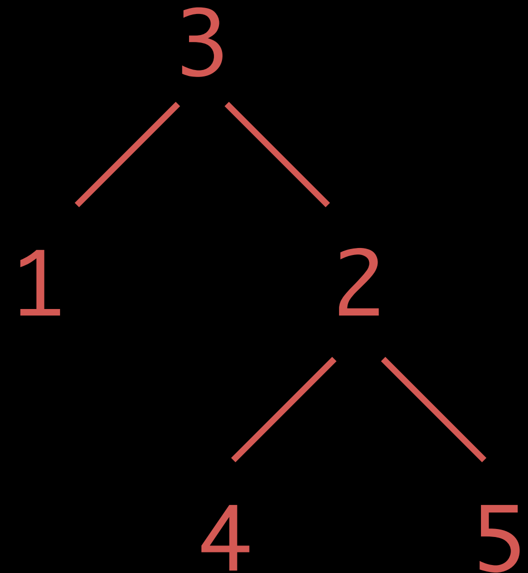
```
T[2]
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



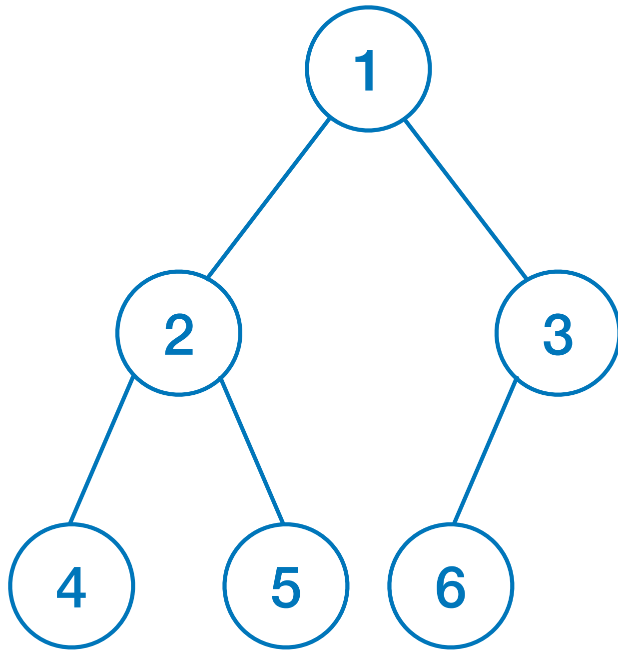
Trees (with objects)

```
1  # Trees with objects (functional tree construction)
2
3  class Tree:
4      def __init__(self, label):
5          self.label = label
6          self.branches = []
7
8      # add child in the right-most branch
9      def add_child(self,val):
10         if not self.branches:
11             self.branches = [Tree(val)]
12         else:
13             self.branches.append(Tree(val))
14
15     # return subtree in location num (0-indexed)
16     def get_subtree(self,num):
17         # make sure that there is a child numbered num
18         if self.branches and num < len(self.branches):
19             return self.branches[num]
20         else:
21             return None
22
23 t = Tree(3)
24 t.add_child(1)
25 t.add_child(2)
26 c = t.get_subtree(1)
27 print(c)
28 c.add_child(4)
29 c.add_child(5)
30 print(t)
```

```
T[2]
T[3, [T[1], T[2, [T[4], T[5]]]]]
```



Binary Trees



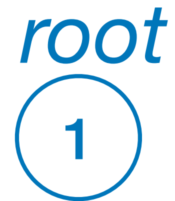
Binary Trees (representation)

```
1  # Binary Tree in Python
2  class Node:
3      def __init__(self, data):
4          self.label = data
5          self.left  = None
6          self.right = None
```

Binary Trees (representation)

```python
# Binary Tree in Python
class Node:
    def __init__(self, data):
        self.label = data
        self.left  = None
        self.right = None
```

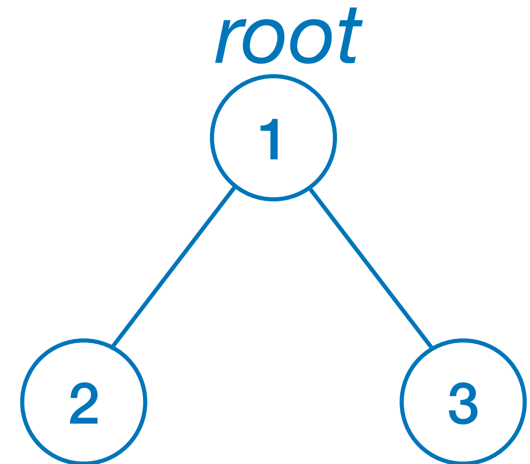
```python
root                    = Node(1)
```



Binary Trees (representation)

```python
# Binary Tree in Python
class Node:
    def __init__(self, data):
        self.label = data
        self.left  = None
        self.right = None
```

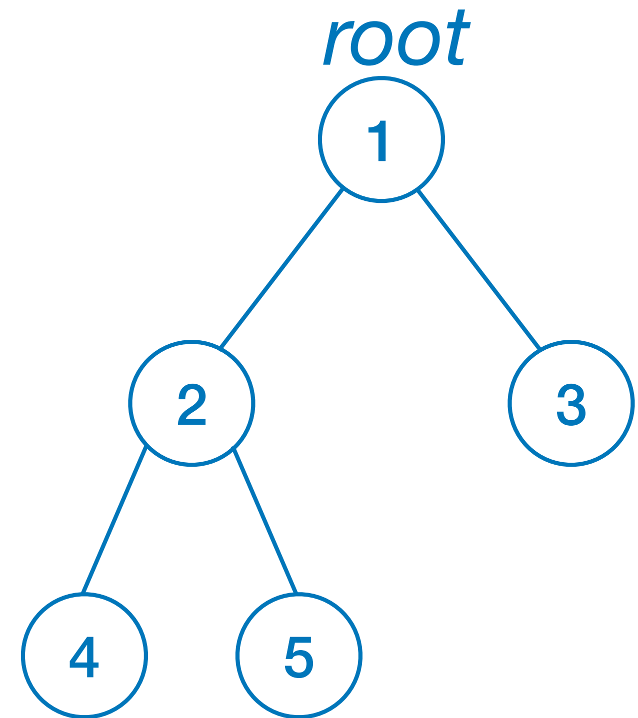
```python
root            = Node(1)
root.left       = Node(2)
root.right      = Node(3)
```



Binary Trees (representation)

```python
# Binary Tree in Python
class Node:
    def __init__(self, data):
        self.label = data
        self.left  = None
        self.right = None
```

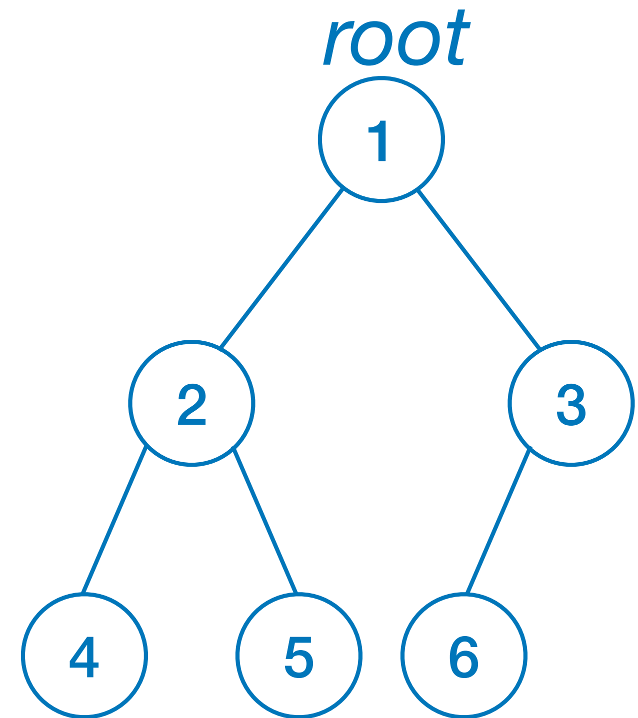
```python
root                = Node(1)
root.left           = Node(2)
root.right          = Node(3)
root.left.left      = Node(4)
root.left.right     = Node(5)
```



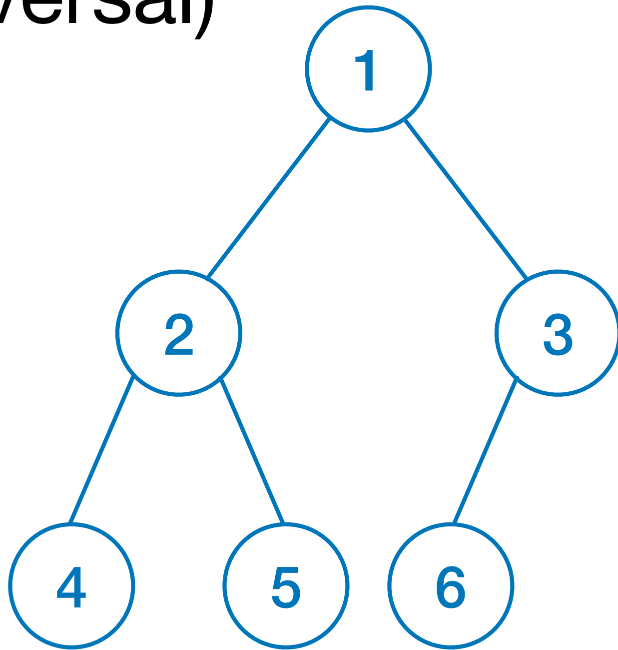
Binary Trees (representation)

```
1  # Binary Tree in Python
2  class Node:
3      def __init__(self, data):
4          self.label = data
5          self.left  = None
6          self.right = None
```

```
1  root              = Node(1)
2  root.left         = Node(2)
3  root.right        = Node(3)
4  root.left.left    = Node(4)
5  root.left.right   = Node(5)
6  root.right.left   = Node(6)
```

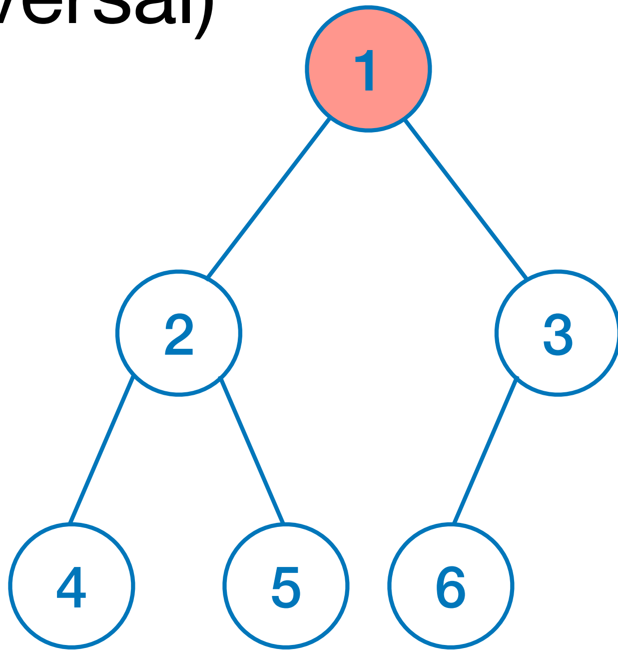


Binary Trees
(traversal)



preorder: root, left subtree, right subtree

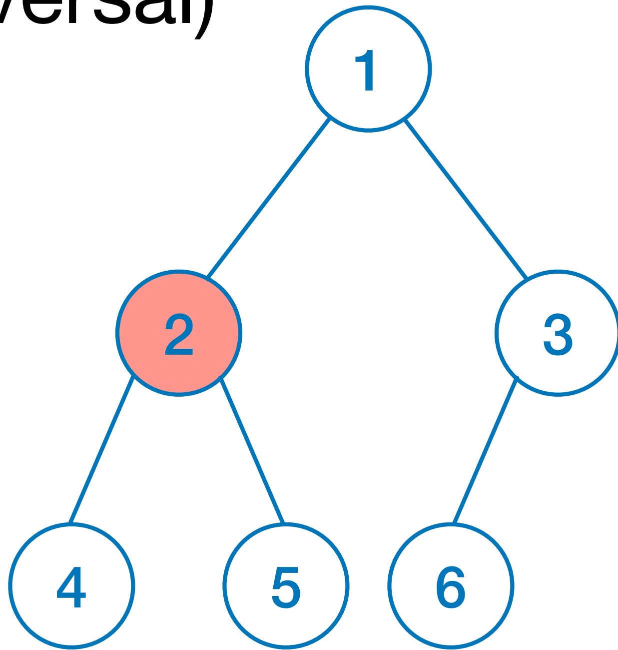
Binary Trees
(traversal)



preorder: root, left subtree, right subtree

1

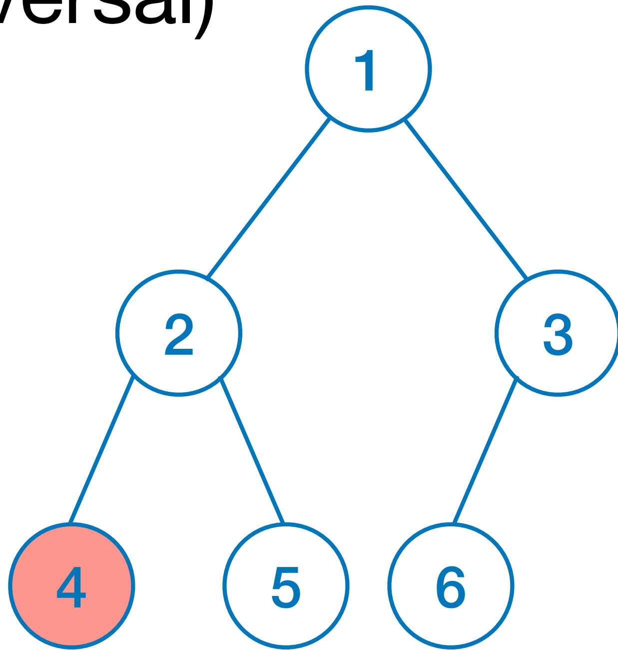
Binary Trees
(traversal)



preorder: root, left subtree, right subtree

1 2

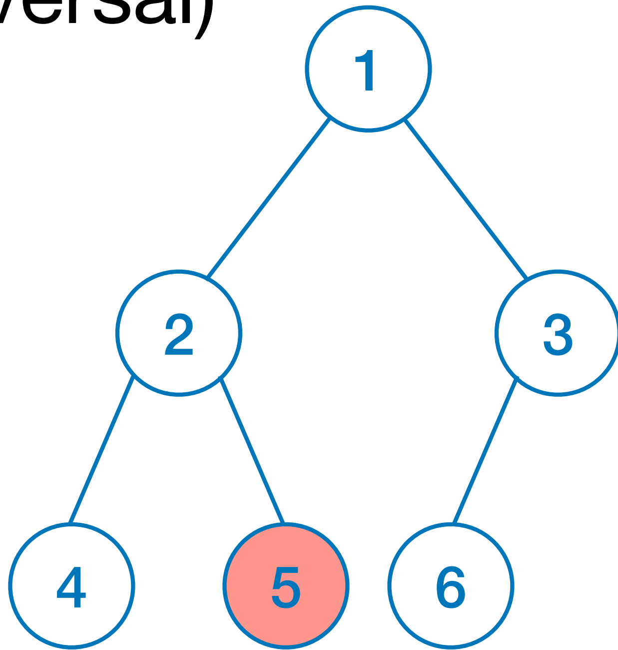
Binary Trees (traversal)



preorder: root, left subtree, right subtree

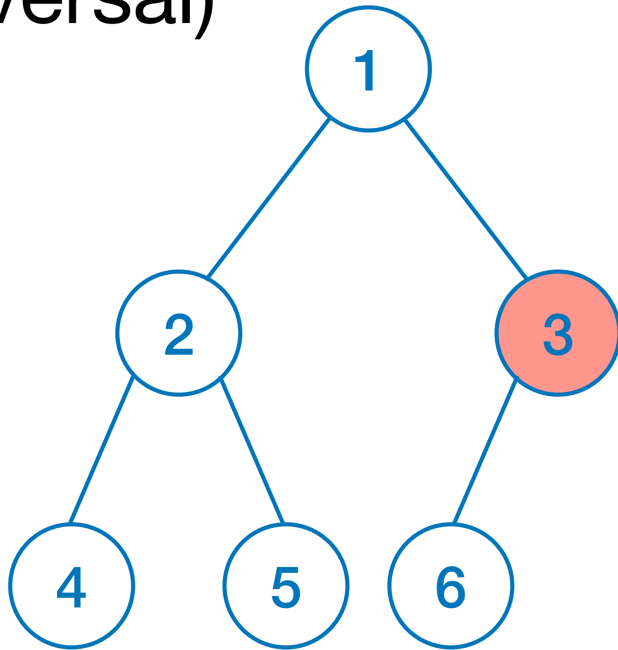
1 2 4

Binary Trees
(traversal)



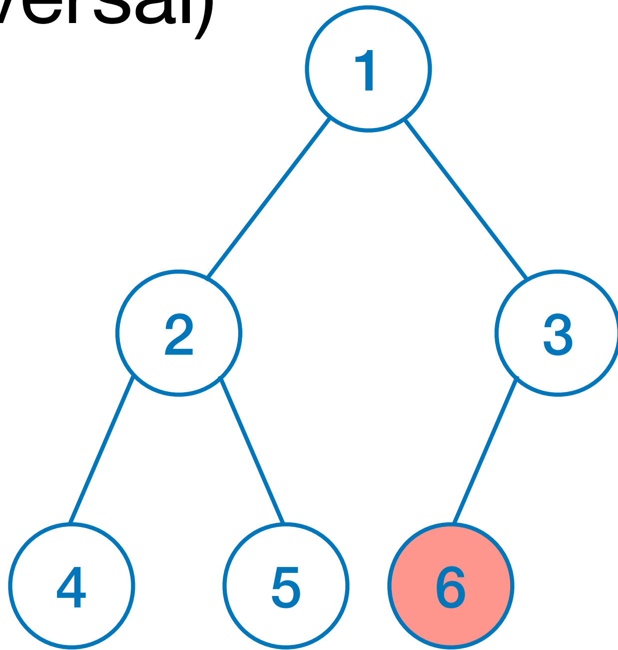
preorder: root, left subtree, right subtree
1 2 4 5

Binary Trees
(traversal)



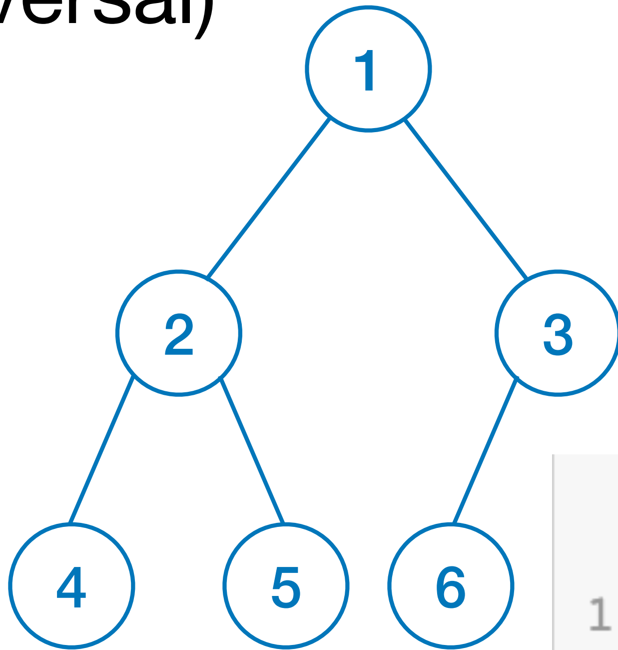
preorder: root, left subtree, right subtree
1 2 4 5 3

Binary Trees
(traversal)



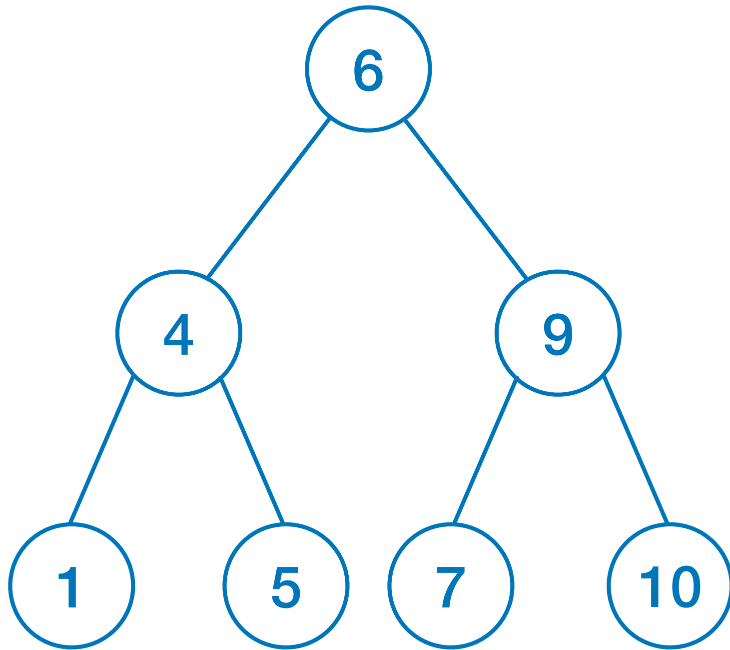
preorder: root, left subtree, right subtree
1 2 4 5 3 6

Binary Trees (traversal)

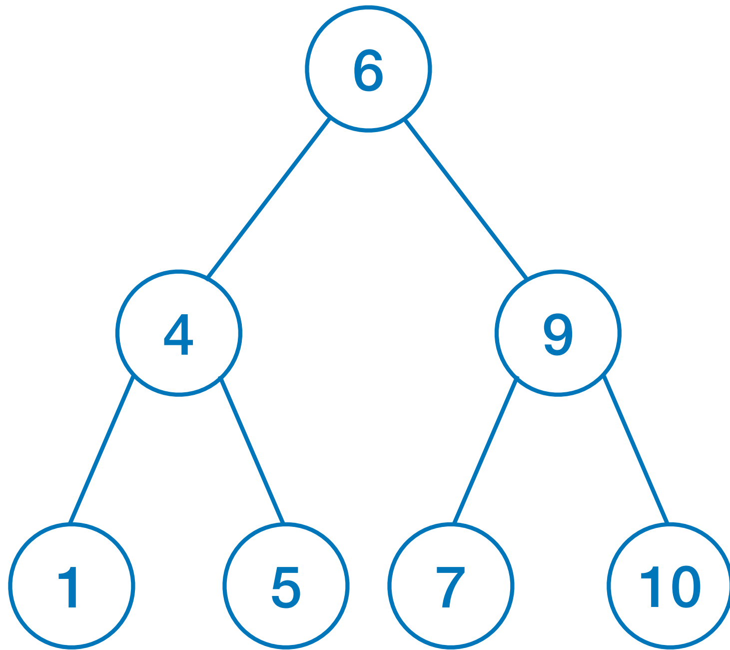



```
 8     # Traverse preorder
 9     def traversePreOrder(self):
10         print(self.label, end=' ')
11         if self.left:
12             self.left.traversePreOrder()
13         if self.right:
14             self.right.traversePreOrder()
```

Binary Search Trees

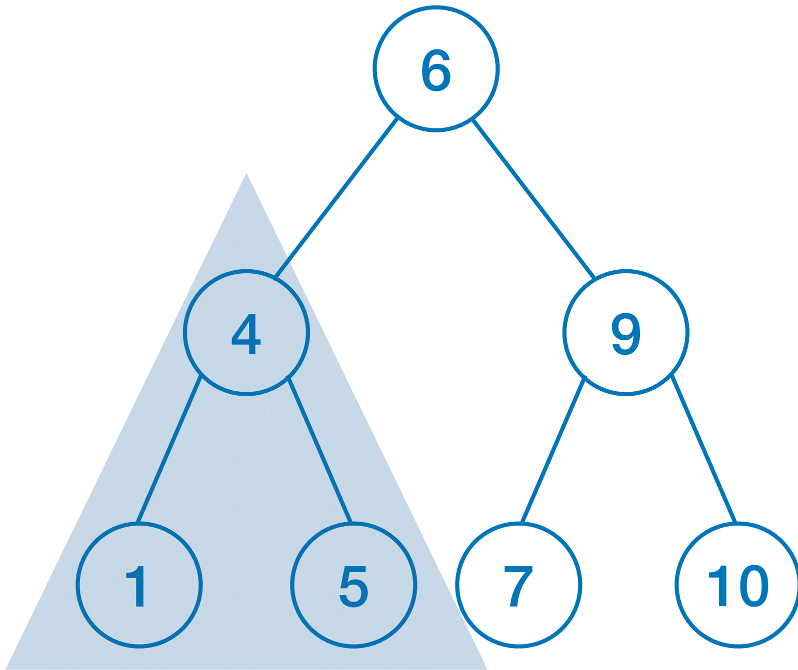


Binary Search Trees

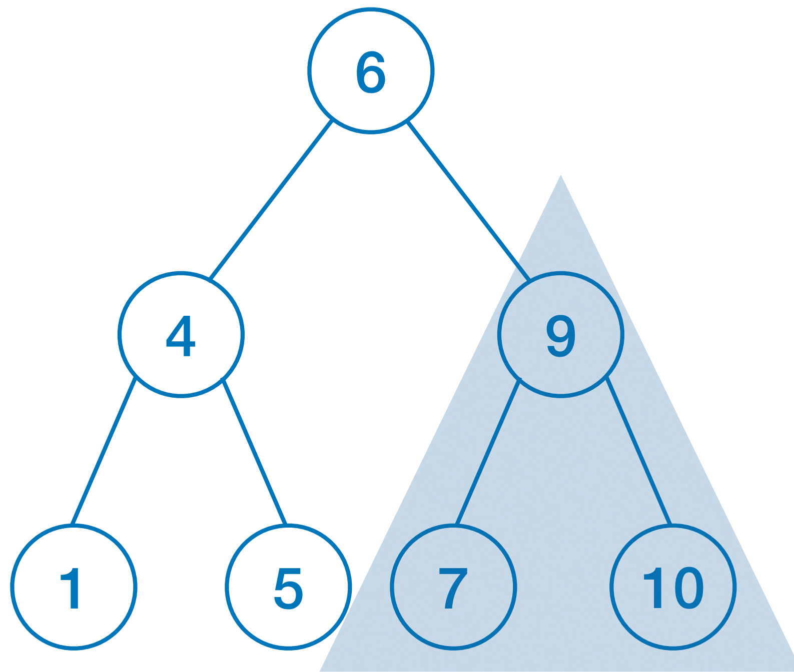


all nodes in left subtree are less than root
all nodes in right subtree are larger than root

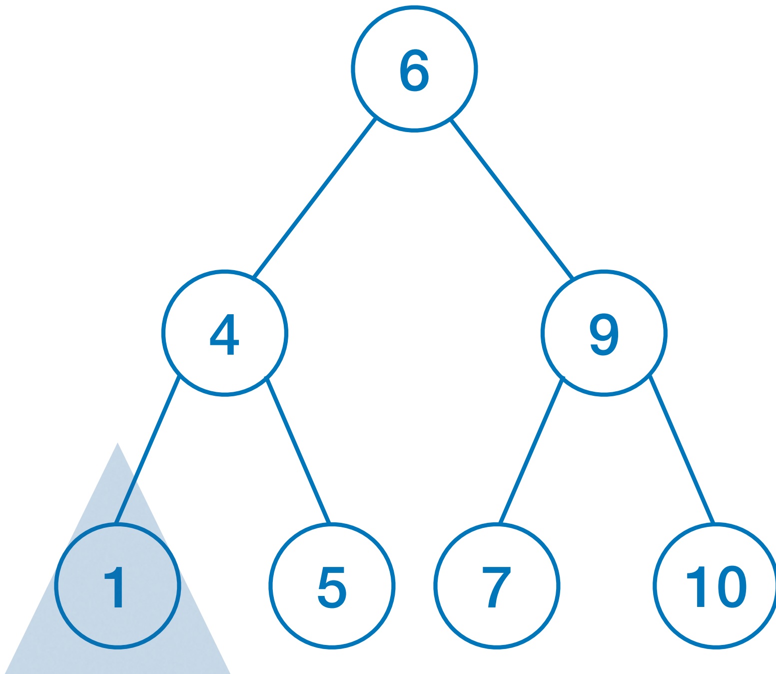
Binary Search Trees



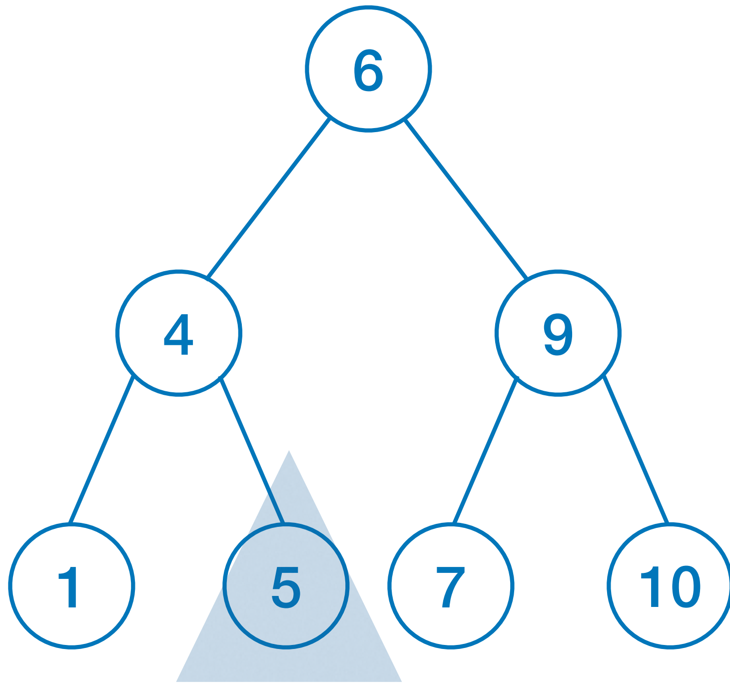
Binary Search Trees



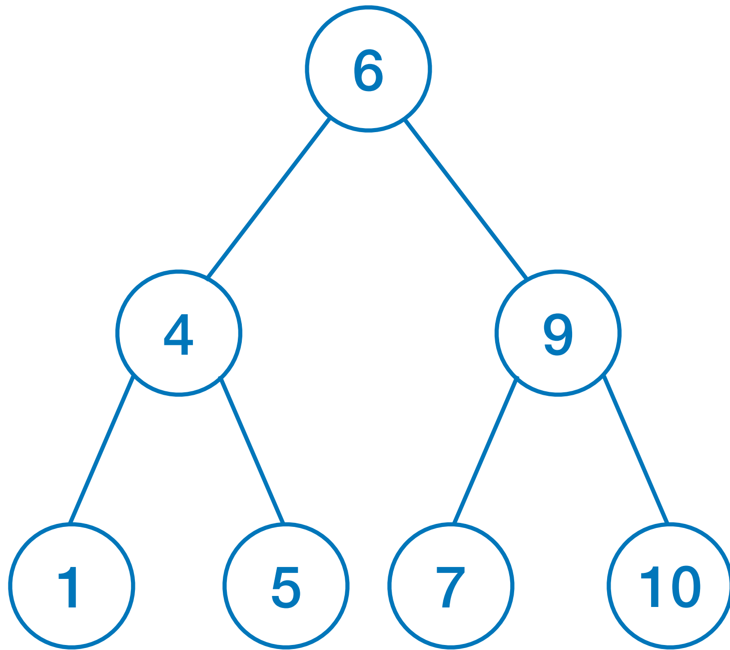
Binary Search Trees



Binary Search Trees

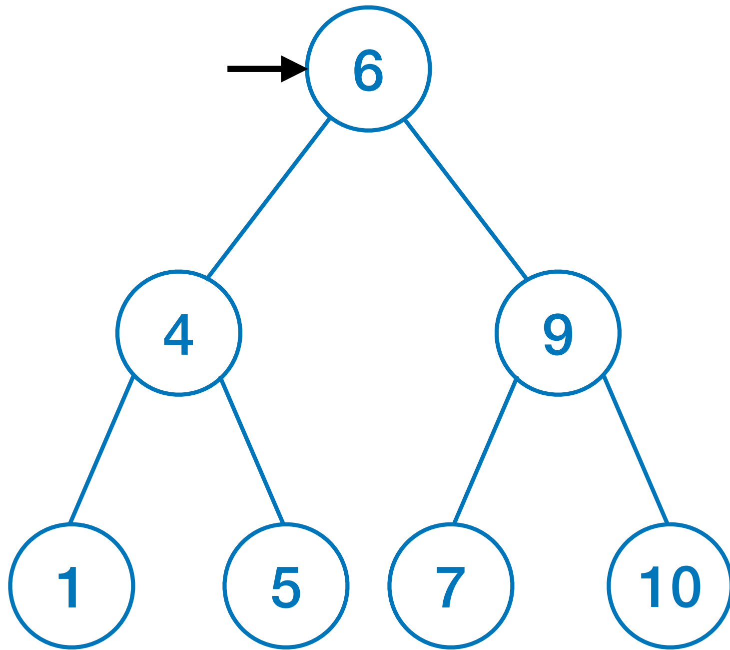


Binary Search Trees



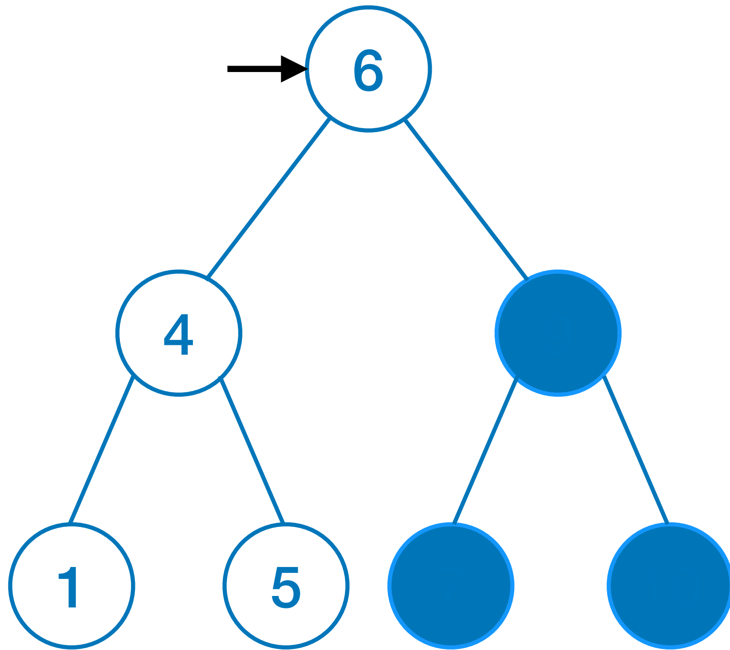
is 5 in this tree?
search in preorder (root, left, right)
n comparisons

Binary Search Trees



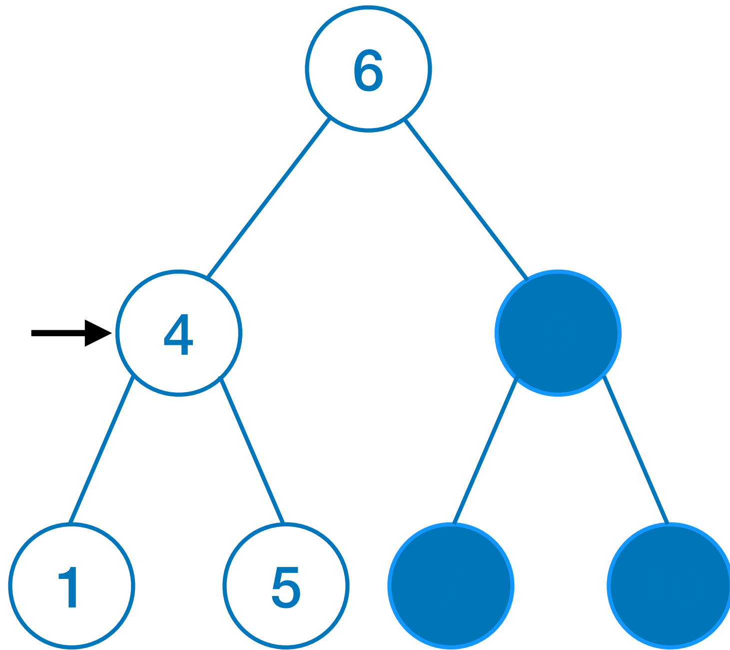
is 5 in this tree?

Binary Search Trees



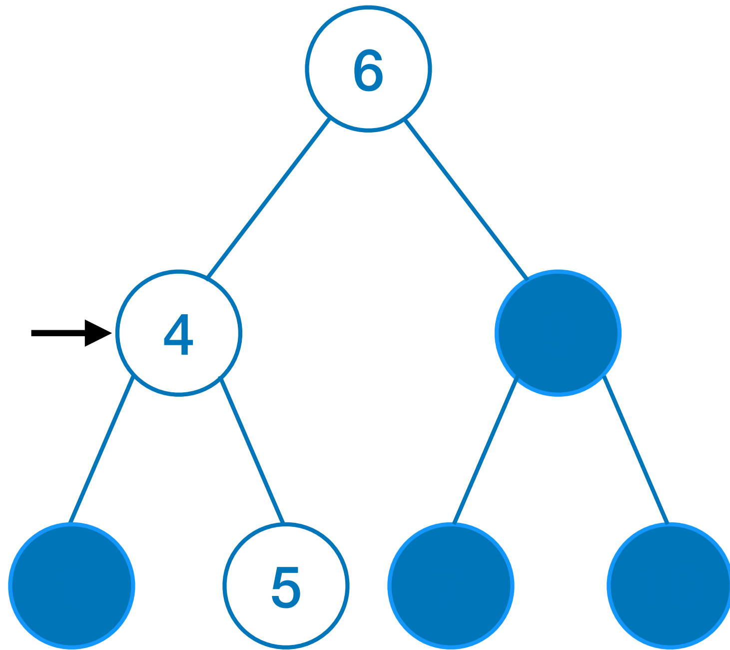
is 5 in this tree?
5 cannot be in right subtree

Binary Search Trees



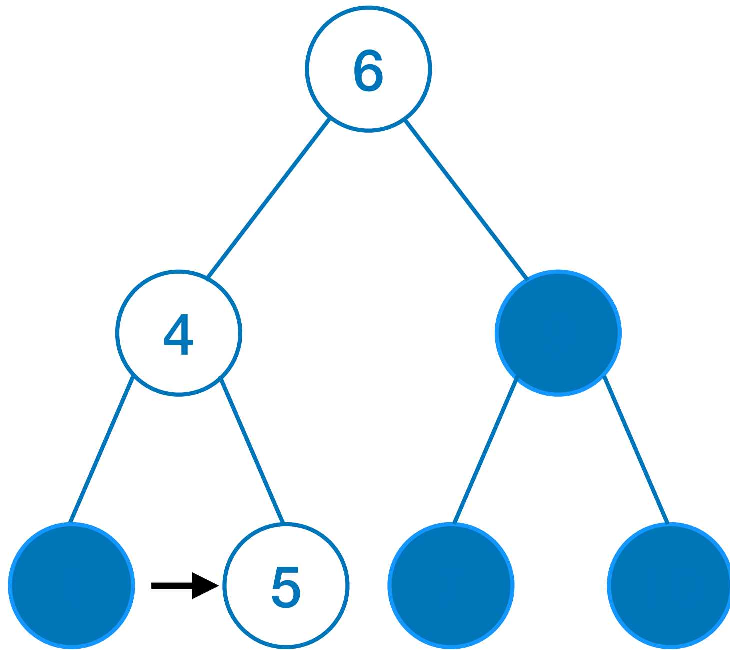
is 5 in this subtree?

Binary Search Trees



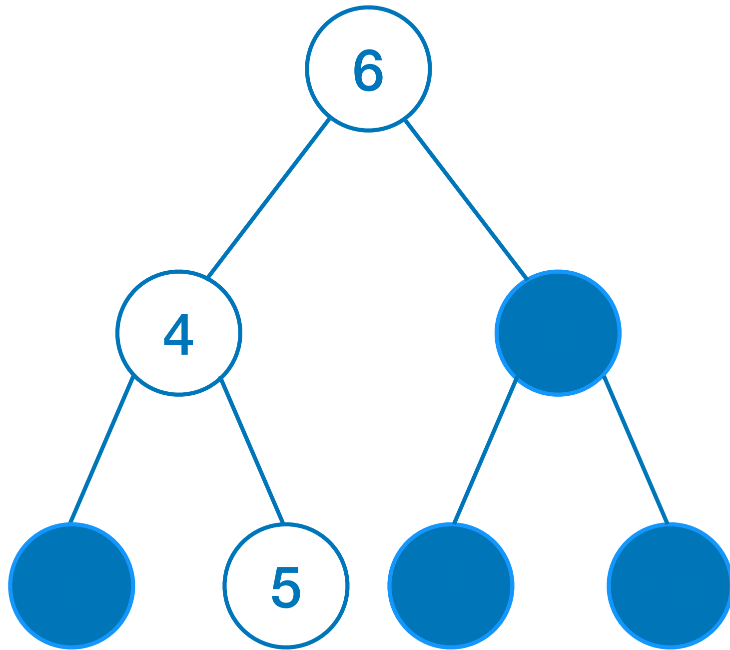
is 5 in this subtree?
5 cannot be in left subtree

Binary Search Trees



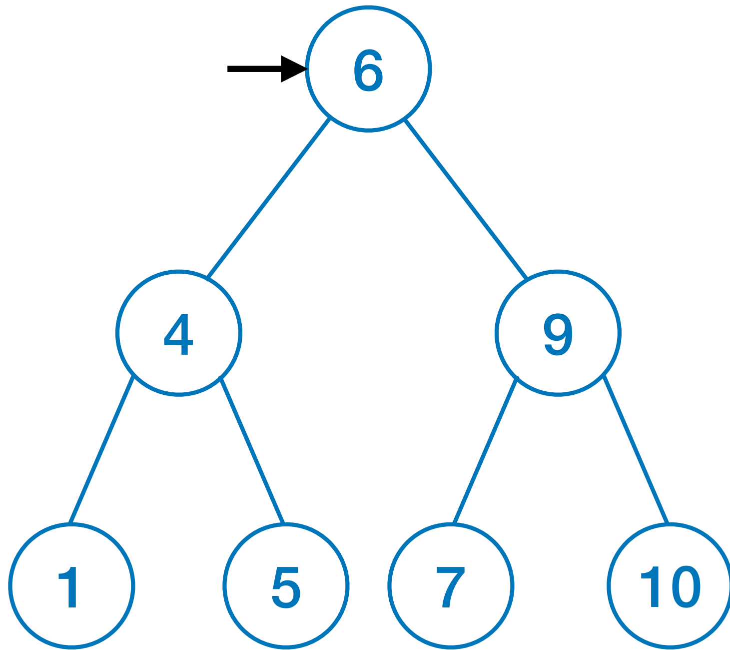
is 5 in this subtree?

Binary Search Trees



is 5 in this subtree?
height of tree is at most *log(n)*

Binary Search Trees




```
1   def bst(node, val):
2       if node == None:
3           return False
4       else:
5           if val == node.label:
6               return True
7           elif val < node.label:
8               return bst(node.left,val)
9           elif val > node.label:
10              return bst(node.right,val)
11
12  bst(root,5)
```