Week#8-Friday
Outline

• Binary Tree
  • Find – tracing
  • Traversal

• Binary Search Tree
def find(node: Union[BinaryTree, None], data: object) -> Union[BinaryTree, None]:
    """
    Return BinaryTree containing data or else None.
    """
    if node is None:
        return None
    else:
        if node.value == data:
            return node
        elif find(node.left, data) is not None:
            return find(node.left, data)
        else:
            return find(node.right, data)
def find(node: Union[BinaryTree, None], data: object) -> Union[BinaryTree, None]:
    ""
    Return BinaryTree containing data or else None.
    ""
    if node is None:
        return None
    else:
        if node.value == data:
            return node
        elif find(node.left, data) is not None:
            return find(node.left, data)
        elif find(node.right, data) is not None:
            return find(node.right, data)
        else:
            return None

Find – another solution

if node is None:
    return None
else:
    if node.value == data:
        return node
    elif find(node.left, data) is not None:
        return find(node.left, data)
    elif find(node.right, data) is not None:
        return find(node.right, data)
    else:
        return None
If we add the following print in find:

```python
print('node:', node.value if node is not None else "None")
```

What will be the output of this code?

```python
from binary_tree_wed import *

bt3 = BinaryTree(3, None, BinaryTree(6))
btt2 = BinaryTree(4, BinaryTree(0), BinaryTree(8))
btt1 = BinaryTree(5, bt2, bt3)

print (find(btt1, 7))
```
The output is:

node: 5
node: 4
node: 0
node: None
node: None
node: None
node: 8
node: None
node: None
node: None
node: 3
node: None
node: 6
node: None
node: None
None
Tree traversal

- Preorder: visit parents first then children
- Postorder: visit children then parents
- Levelorder: visit node in each level
Tree traversal

- **Preorder**: visit parents first then children
  - One Use: can be used to make a copy of tree
  - order: 2,7,2,6,5,11,5,9,4

- **Postorder**: visit children then parents
  - One Use: can be used to delete tree
  - order: 2,5,11,6,7,4,9,5,2

- **Levelorder**
  - One Use: can be used to print a sorted tree
  - order: 2,7,5,2,6,9,5,11,4

- **Inorder**
  - One Use: in Binary Search Trees to print nodes in order
  - 2,7,5,6,11,5,4,9
Binary Tree Traversal

- **We will implement the following functions:**
  - Preorder
  - Postorder
  - Inorder
preorder

- visit this node itself
- visit the left subtree in preorder
- visit the right subtree in preorder
```python
def preorder_visit(t: BinaryTree, act: Callable[[BinaryTree], Any]) -> None:
    """
    Visit BinaryTree t in preorder and act on nodes as you visit.
    """
    if t is None:
        pass
    else:
        act(t)
        preorder_visit(t.left, act)
        preorder_visit(t.right, act)

>>> bt = BinaryTree(5, BinaryTree(7), BinaryTree(9))
>>> def f(node): print(node.value)
>>> preorder_visit(bt, f)
5
7
9
"""
```
```python
def preorder_visit(t: BinaryTree, act: Callable[[BinaryTree], Any]) -> None:
    """
    Visit BinaryTree t in preorder and act on nodes as you visit.
    """

    if t is None:
        pass
    else:
        act(t)
        preorder_visit(t.left, act)
        preorder_visit(t.right, act)
```

```python
>>> bt = BinaryTree(5, BinaryTree(7), BinaryTree(9))
>>> def f(node): print(node.value)
>>> preorder_visit(bt, f)
5
7
9
"""
```
postorder

• visit the left subtree in postorder
• visit the right subtree in postorder
• visit this node itself
def postorder_visit(t: BinaryTree, act: Callable[[BinaryTree], Any]) -> None:
    ""
    Visit BinaryTree t in postorder and act on nodes as you visit.
    ""
    if t is None:
        pass
    else:
        postorder_visit(t.left, act)
        postorder_visit(t.right, act)
        act(t)

>>> bt = BinaryTree(5, BinaryTree(7), BinaryTree(9))
>>> def f(node): print(node.value)
>>> postorder_visit(bt, f)
7
9
5
"""
```python
def postorder_visit(t: BinaryTree, act: Callable[[BinaryTree], Any]) -> None:
    
    Visit BinaryTree t in postorder and act on nodes as you visit.

    >>> bt = BinaryTree(5, BinaryTree(7), BinaryTree(9))
    >>> def f(node): print(node.value)
    >>> postorder_visit(bt, f)
    7
    9
    5
    
    if t is None:
        pass
    else:
        postorder_visit(t.left, act)
        postorder_visit(t.right, act)
        act(t)
```
inorder

• A recursive definition:
  • visit the left subtree \textit{inorder}
  • visit this node itself
  • visit the right subtree \textit{inorder}

• The code is almost identical to the definition.
def inorder_visit(t: BinaryTree, act: Callable[[BinaryTree], Any]) -> None:
    """
    Visit each node of binary tree rooted at root in order and act.
    """
    if t is None:
        pass
    else:
        inorder_visit(t.left, act)
        act(t)
        inorder_visit(t.right, act)

>>> bt = BinaryTree(5, BinaryTree(7), BinaryTree(9))
>>> def f(node): print(node.value)
>>> inorder_visit(bt, f)
7
5
9
"""
def inorder_visit(t: BinaryTree, act: Callable[[BinaryTree], Any]) -> None:
    """
    Visit each node of binary tree rooted at root in order and act.
    """
    if t is None:
        pass
    else:
        inorder_visit(t.left, act)
        act(t)
        inorder_visit(t.right, act)
Binary Search Tree

- Add **ordering** conditions to a binary tree:
  - data are comparable: int, float, etc
  - data in left subtree are less than node.data
  - data in right subtree are more than node.data
Binary Search Tree

• Draw a Binary Search Tree if the following numbers are inserted in the given order:
  10, 6, 4, 8, 18, 21, 15
Binary Search Tree

• Draw a Binary Search Tree if the following numbers are inserted in the given order:

10, 6, 4, 8, 18, 21, 15
why binary search trees?

• Searches that are directed along a single path are **efficient**:  
  • a BST with 1 one has height 1  
  • a BST with 3 nodes may have height 2  
  • a BST with 7 nodes may have height 3  
  • a BST with 15 nodes may have height 4  
  • a BST with $n$ nodes may have height: $\lceil \log_2 n \rceil$
bst contains

• If node is the root of a “balanced” BST, then we can check whether an element is present in about $\lceil \log_2 n \rceil$ node accesses.
```python
def bst_contains(node: BinaryTree, value: object) -> bool:
    
    Return whether tree rooted at node contains value.

    Assume node is the root of a Binary Search Tree

    >>> bst_contains(None, 5)
    False
    >>> bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(9)), 5)
    True
    
    
```
```python
def bst_contains(node: BinaryTree, value: object) -> bool:
    """
    Return whether tree rooted at node contains value.
    Assume node is the root of a Binary Search Tree
    
    >>> bst_contains(None, 5)
    False
    >>> bst_contains(BinaryTree(7, BinaryTree(5), BinaryTree(9)), 5)
    True
    """
    if node is None:
        return False
    elif node.value > value:
        return bst_contains(node.left, value)
    elif node.value < value:
        return bst_contains(node.right, value)
    else:
        return True
```