Disclaimer

This complementary study package is provided by Easy 4.0 Education Inc. and its affiliated mentors. This study package seeks to support your study process and should be used as a complement, **NOT** substitute to course material, lecture notes, problem sets, past tests and other available resources.

We acknowledge that this package contains some materials provided by professors and staff of the University of Toronto, and the sources of these materials are cited in details wherever they appear.

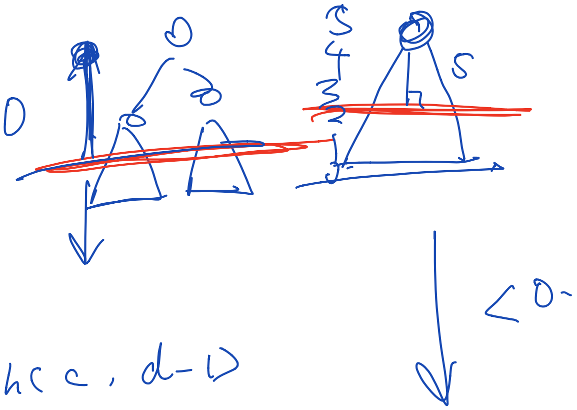
This package is distributed for free to students participating in Easy 4.0's review seminars, and are not for sale or other commercial uses whatsoever. We kindly ask you to refrain from copying or selling in part or in whole any information provided in this package.

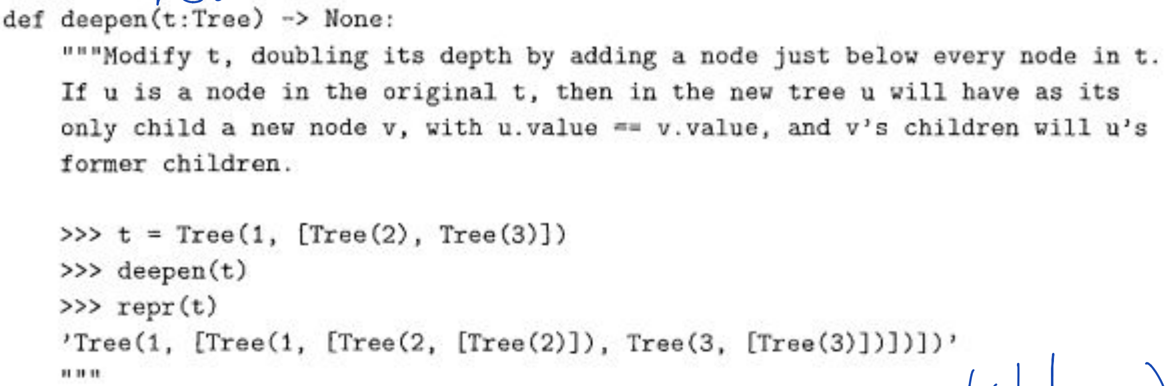
Thank you for choosing Easy 4.0. We sincerely wish you the best of luck in all of your exams.

Easy 4.0 Education Inc.

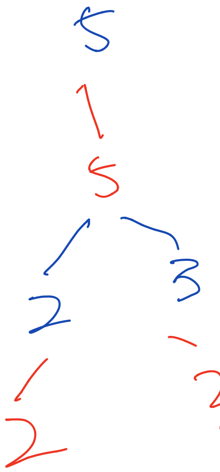
Section 3. Tree

```
def count_at_depth(t: Tree, d) -> int:
    “””Count at the number of node at depth d, assume the root has depth 0.

    >>> t = Tree(‘A’, [Tree(‘B’, [Tree(‘E’)]), Tree(‘C’, [Tree(‘D’)])])
    >>> count_at_depth(t, 1)
    2
    “””
    if d==0:
        return 1

    else:
        acc=0
        for c in t.children:
            acc += count_at_depth(c, d-1)

        return acc
```



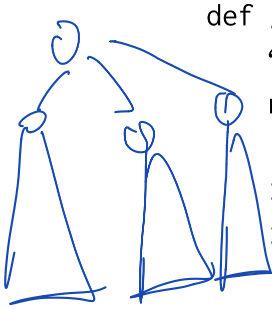


```
def deepen(t:Tree) -> None:
    """Modify t, doubling its depth by adding a node just below every node in t.
    If u is a node in the original t, then in the new tree u will have as its
    only child a new node v, with u.value == v.value, and v's children will u's
    former children.

    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> deepen(t)
    >>> repr(t)
    'Tree(1, [Tree(1, [Tree(2, [Tree(2)]), Tree(3, [Tree(3)])])])'
    """
```



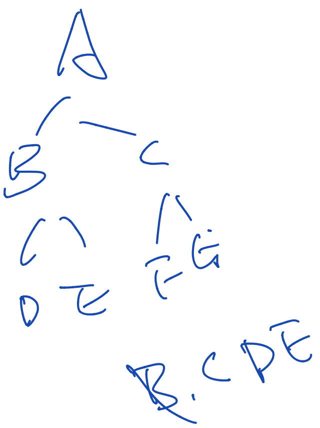
```
t.children=[Tree(t.value, t.children)]
                copy of t.

for c in t.children[0].children:
    deepen(c).
```

EZ4 CSC 148 TT2 REVIEW PART 2



```
def get_longest_path(t: Tree) -> List:
    “”” Return a list contains the values in the right most longest pathfrom
    root to leaf.

    >>> t = Tree(‘A’, [Tree(‘B’, [Tree(‘E’)]), Tree(‘C’, [Tree(‘D’)])])
    >>> get_longest_path(t)
    [‘A’, ‘C’, ‘D’]
    “””
    if not t.children:
        return [t.value]

    else: result=[].
        for c in t.children:
            path= get_longest_path(c).
            if len(path) >=. len(result).
                result = path[:]
        return [t.value] + result.
```

```
def level_order_number_nodes(t):
    “”” Replace the value of all nodes by the sequence of their occurrence in
    the level order

    >>> t = Tree(‘A’, [Tree(‘B’, [Tree(‘E’)]), Tree(‘C’, [Tree(‘D’)])])
    >>> level_order_number_nodes(t)
    >>> repr(t)
    Tree(1, [Tree(2, [Tree(4)]), Tree(3, [Tree(5)])])
    “””
    n=1.
    q=[]
    q.append(t).
    while. len(q) > 0:  # not queue.is_empty().

        temp= q.pop(0)
        temp.value = n.
        n+=1.
        q.extend(temp.children).
```

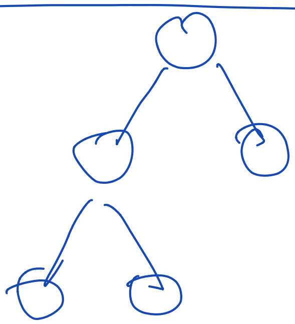


Section 4. Binary Tree

None

Binary Tree, 顾名思义, arity最大只有2的Tree, 但是, 我们的implementation有改变，不再是node, 变成了 left和right.（虽然我们还是可以 for child in [t.left, t.right]).
注意：binary tree并没有规定顺序!
所以说在这里我们可以分成三项.

1. Value
2. left child
3. right child

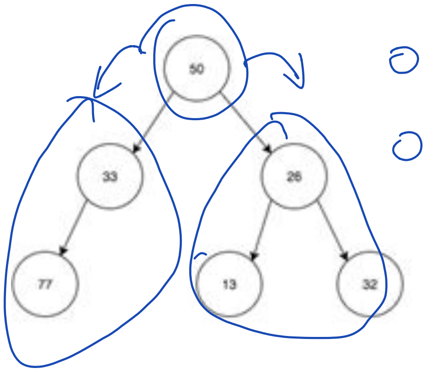


```
class BinaryTree:
    """
    A Binary Tree, i.e. arity 2.
    """
    def __init__(self, value, left=None, right=None):
        """
        Create BinaryTree self with value and children left and right.
        @param BinaryTree self: this binary tree
        @param object value: value of this node
        @param BinaryTree|None left: left child
        @param BinaryTree|None right: right child
        @rtype: None
        """
        self.value, self.left, self.right = value, left, right
```

Pre order: root先. children后.

Traversals:

我们一共接触了三种traversal, pre-order,　in-order和post-order. 这里pre, in和post可以理解为 root放的位置. Pre-order也就是先root再children, in-order为root在左右children中间, post-order为root在最后. 注意,



item　recursion.

○ Pre-order: 中 左右.

50, 33, 77, 26. 13, 32

○ Post-order: 左右中 77, 33, 13, 32, 26, 50

In-order: 左中右

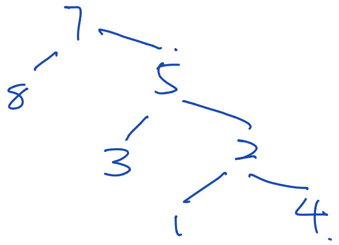
77, 33, 50, 13, 26, 32.

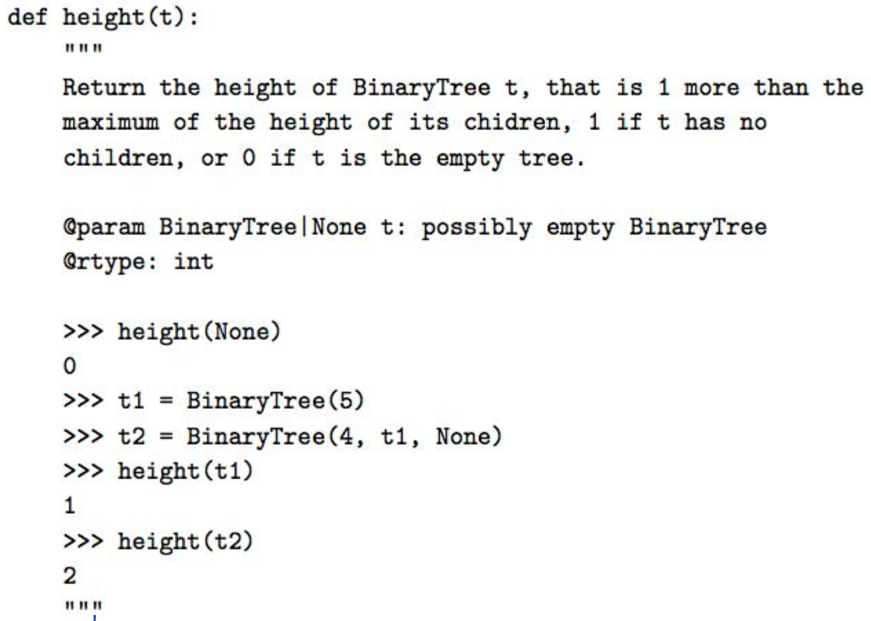
```
def inorder(node):
    if not node :          # if node is None.
        return []

    else:  return.  inorder( node.left) +
                     [node. value.]  +
                    inorder ( node . right).
```

常见题型： 给两种order要求画出原本的binary tree

pre-order: 7 8 5 3 2 1 4

in-order : 8 7 3 5 1 2 4

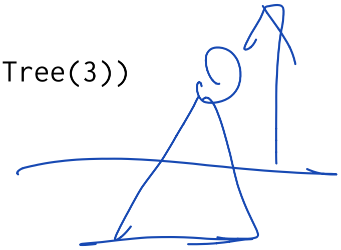




```
def height(t):
    """
    Return the height of BinaryTree t, that is 1 more than the
    maximum of the height of its chidren, 1 if t has no
    children, or 0 if t is the empty tree.

    @param BinaryTree|None t: possibly empty BinaryTree
    @rtype: int

    >>> height(None)
    0
    >>> t1 = BinaryTree(5)
    >>> t2 = BinaryTree(4, t1, None)
    >>> height(t1)
    1
    >>> height(t2)
    2
    """
```

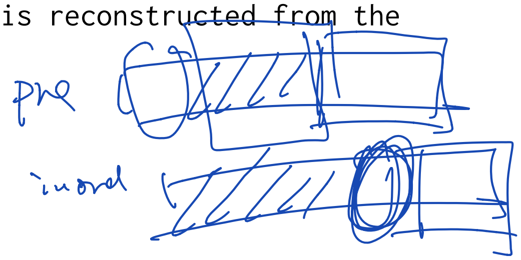
```
    if not t:
        return 0.
    else:
        return max( height(t.left), height(t.right)) + 1.
```

```
def count(node, item) -> int:
    “”” Count the occurrence of item in the tree rooted at node.
    “””
    if not node:
        return 0.

    else:
        if node.value == item:
            return 1 + count(node.left, item) +
                       count(node.right, item).
        else:
            return count(node.left, item) +
                   count(node.right, item).

def get_above(node: BinaryTree, d: int) -> List:
    “”” Get all the value in the tree rooted at node and above the depth d,
    Assume root has depth 0.

    >>> t = BinaryTree(1, BinaryTree(2, BianryTree(5)), BinaryTree(3))
    >>> get_above(t, 2)
    [1, 2, 3]
    “””
    if not node:
        return []
    if d == 0:
        return []

    else:
        return [node.value] + get_above(node.left, d - 1) +
               get_above(node.right, d - 1).
```



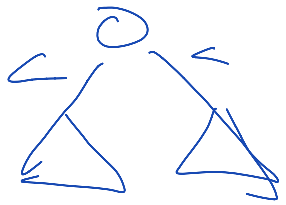
```
def swap_node(node: BinaryTree) -> None:
    """ Swap all the left and child node in the binary tree rooted at node.

    >>> t = BinaryTree(1, BinaryTree(2, None, None), BinaryTree(3, None, None))
    >>> swap_node(t)
    >>> t
    ~~t =~~ BinaryTree(1, BinaryTree(3, None, None), BinaryTree(2, None, None))
    """
    if node:
        node.left, node.right = node.right, node.left.
        swap_node(node.left)
        swap_node(node.right).
```

```
def get_all_path(node: BinaryTree) -> ~~None~~ list:
    """ Get all path from root to leaf in the Binary Tree rooted at node.

    >>> t = BinaryTree(1, BinaryTree(2), inaryTree(3))
    >>> get_all_path(t)
    [[1, 2], [1, 3]]
    """
    if not node:
         return []

    else:
        if node.left is None and node.right is None:
              return [[node.value]]

        else: acc=[]
              for c in [node.left, node.right]:
                   for path in get_all_path(c).
                        acc.append([node.value] + path).

        return acc.
    def reconstruct_tree(preorder: list, inorder: list) -> BinaryTree:
        """ Return the root of new binary tree that is reconstructed from the
given                           inorder and preorder list.
        """
        if len(preorder) == 0:
            return None
        root = preorder[0]
        index = inorder.index(root).
        left_subtree = reconstruct_tree(preorder[1: index+1],
                                        inorder[0: index]).
        right_subtree = reconstruct_tree(preorder[index+1: ],
                                         inorder[index+1: ])
        return BinaryTree(root, left_subtree,
                          right_subtree).
```

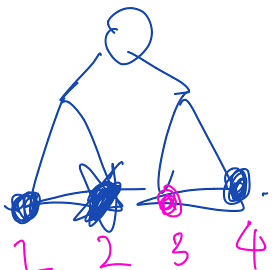


Section 5. Binary Search Tree

Binary Search Tree 简单来说是在 binary tree的基础上增加大小关系, 在binary search tree当中遵循着左边subtree一定小于root, root一定小于右边subtree的定式。所以在binary search tree当中, 如果我们需要找到一个node, 根据性质, 我们只需要检查单边就可以, 这样可以大大缩减running time.



✓ 1) 在BST中, inorder traversal可以返回一个sorted list.
2) 在常规BST中, 一般不存在duplicate value.
3) 对BST的任何操作都不能违反BST左小右大的原则.
4) 左边subtree 的最右是左边subtree中仅小于root的node.
5) 右边subtree的最左是右边subtree中仅大于root的node.



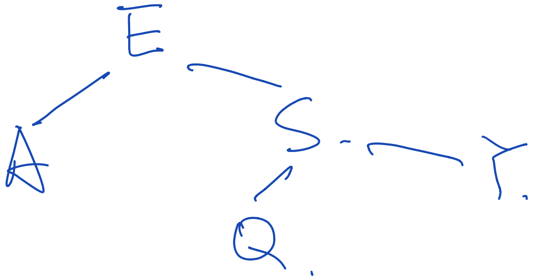
BST的insert

会从root开始按照左小右大原则向下寻找，加到leaf. 如果出现等于, 那么什么都不发生.

Draw the BST that results when you insert items with keys

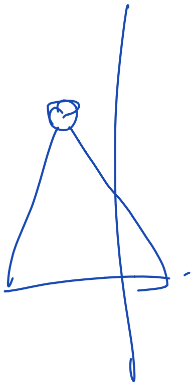
E A S Y Q U E S T I O N

in that order into an initially empty tree.

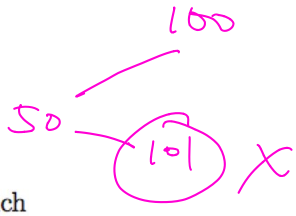


BST的search

def BST_contains(node: BTnode, value:object) -> bool:



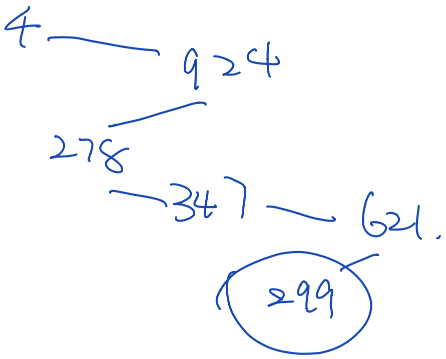
```
  if not node:
      return False
  if node.value == value:
      return True.
  elif: node.value < value:
      return BST_contains(node.right, value).
  else: return BST_contains(node.left.value).
```



Suppose we have int values between 1 and 1000 in a BST and search for 363. Which of the following cannot be the sequence of keys examined.

d

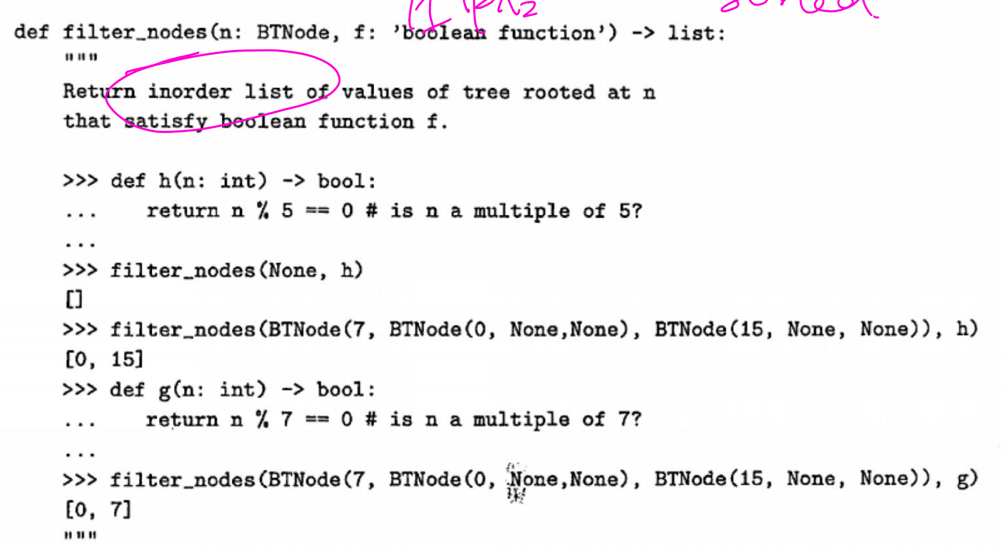
(a) 2 252 401 398 330 363
(b) 399 387 219 266 382 381 278 363
(c) 3 923 220 911 244 898 258 362 363
(d) 4 924 278 347 621 299 392 358 363
(e) 5 925 202 910 245 363



BST的Traversal

在BST中, 做题一定要使用到左小右大的性质, 并且记住 inorder 永远会得到一个sorted list, 做题过程中一旦遇到要求返回sorted list, 一定要使用inorder的顺序.

左中右 → sorted



```
def filter_nodes(n: BTNode, f: 'boolean function') -> list:
    """
    Return inorder list of values of tree rooted at n
    that satisfy boolean function f.

    >>> def h(n: int) -> bool:
    ...     return n % 5 == 0 # is n a multiple of 5?
    ...
    >>> filter_nodes(None, h)
    []
    >>> filter_nodes(BTNode(7, BTNode(0, None,None), BTNode(15, None, None)), h)
    [0, 15]
    >>> def g(n: int) -> bool:
    ...     return n % 7 == 0 # is n a multiple of 7?
    ...
    >>> filter_nodes(BTNode(7, BTNode(0, None,None), BTNode(15, None, None)), g)
    [0, 7]
    """
```

```
if not n:
    return [].

acc = filter_nodes(n.left, f)

if f(n.value):
    acc.append(n.value)
acc += filter_nodes(n.right, f)
return acc
```

```
def get_greater_than(n: BTnode, item: int) -> List[int]:
    “”” Return all items in the BST rooted at n that are greater than item in a sorted
order.
    “””
    if not n:
        return []
    if n.value > item:
        return getgreater_than(n.left, item) + [n.value] +
            get_greater_then (n.right, item)
    else: # n.value <= item
        return getgreater_than (n.right, item)
```