```
def min_length_subarr(s, A):
    w_sum, w_st = 0, 0
    min_l = math.inf

    for w_end in range(len(A)):
        w_sum += A[w_end]
        while w_sum >= s:
            min_l = min(min_l, w_end - w_st + 1)
            w_sum -= A[w_st]
            w_st += 1

    return min_l
```

```
def max_subarray_of_size_k(k, arr):
    max_sum, window_sum = 0, 0
    window_start = 0    #index val.
    for window_end in range(len(arr)):
        window_sum += arr[window_end]
        if window_end >= k-1
            max_sum = max(max_sum, window_sum)
            window_sum -= arr[window_start]
            window_start +=1

    return max_sum
```

Implement an inorder traversal w/ constant space

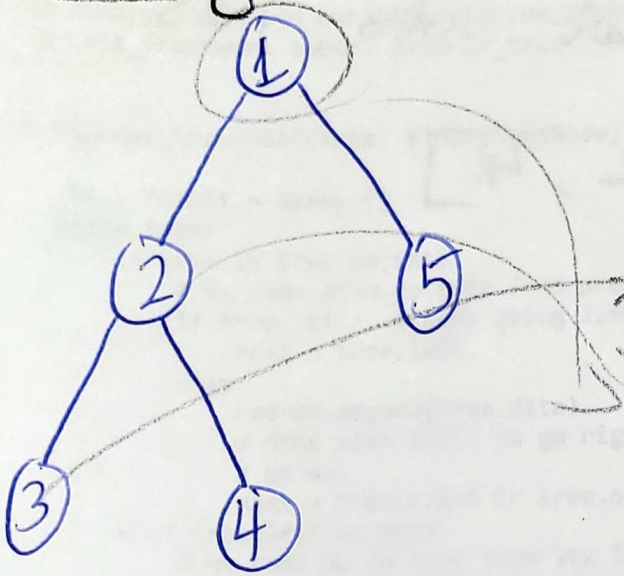
9.10 7/14

Input: Bin. Tree, Output: list

Recursive Approach

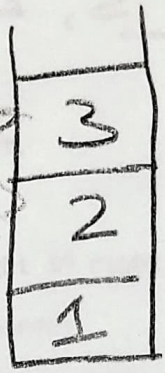
```
def f(root: Bin.Tree) -> List[int]:
    res = []
    def inorder(root):
        if not root:
            return
        inorder(root.left)
        res.append(root.data)
        inorder(root.right)
    inorder(root)
    return res
```

Using stack



res = [ ]

Stack

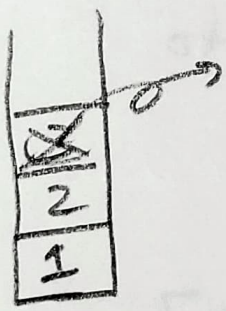


at node 3

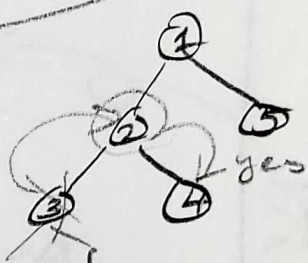
At this point, there is no left subtree.
Now we pop() and add to the res. list.

function call stack

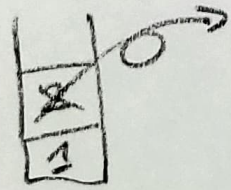
When calling func. inside a func. it has to remember to get back to the orig. func.



res = [3]

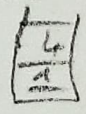


Now we check if the node has a right subtree.
Since there is no right subtree, we pop again to the res.

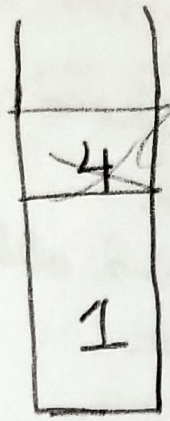


res = [3, 2]

From node 2, we have right child so push that node to the call stack and check it's children. →

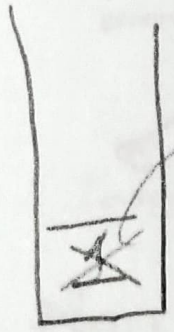


Since at node w/ value 4 has no left children, we pop that into res.



→ res = [3, 2, 4]

Since node w/value 4 does not have any right child we pop the last elem. on the stack to res.



→ res = [3, 2, 4, 1]

And then we move to node w/ 5.

... ⇒ res = [3, 2, 4, 1, 5]

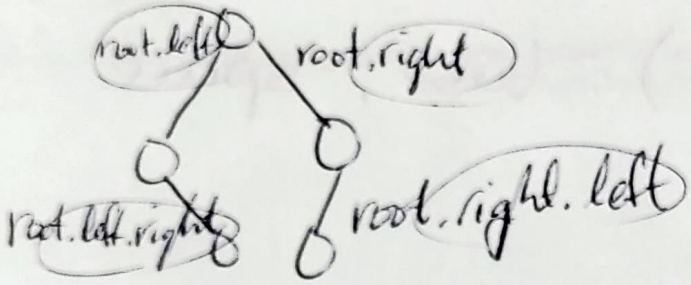
9.10

```
from typing import List

from binary_tree_with_parent_prototype import BinaryTreeNode
from test_framework import generic_test


def inorder_traversal(tree: BinaryTreeNode) -> List[int]:

    prev, result = None, []
    while tree:
        if prev is tree.parent:
            # We came down to tree from prev.
            if tree.left:  # Keep going left.
                next = tree.left
            else:
                result.append(tree.data)
                # Done with left, so go right if right is not empty. Otherwise,
                # go up.
                next = tree.right or tree.parent
        elif tree.left is prev:
            # We came up to tree from its left child.
            result.append(tree.data)
            # Done with left, so go right if right is not empty. Otherwise, go
            # up.
            next = tree.right or tree.parent
        else:  # Done with both children, so move up.
            next = tree.parent

        prev, tree = tree, next
    return result


if __name__ == '__main__':
    exit(
        generic_test.generic_test_main('tree_with_parent_inorder.py',
                                       'tree_with_parent_inorder.tsv',
                                       inorder_traversal))
```

9.2 7/14

Input: Bin. Tree
Output: boolean

- Need to check if search path is same from both sides.
- Need to have nodes points to opposite?

(Ex. 

)

How to check?

↳ Not sure if I need to check this.

- First need to check if depth is same
  ↳ return False, if no

- Starting from the leaves, we need to check if nodes value are the same for both sides.

- Empty ⇒ return False
- Only root ⇒ return True

→ Do I need to check levels?

- Naive Approach
  - ↳ Use hash map.
    - ↳ Store all nodes into hash except root of one side
      - ↳ Check other side

$O(h-1) = O(h)$ time, space

11 min quit

```
from binary_tree_node import BinaryTreeNode
from test_framework import generic_test


def is_symmetric(tree: BinaryTreeNode) -> bool:
    def check_symmetric(subtree_0, subtree_1):
        if not subtree_0 and not subtree_1:
            return True
        elif subtree_0 and subtree_1:
            return (subtree_0.data == subtree_1.data
                    and check_symmetric(subtree_0.left, subtree_1.right)
                    and check_symmetric(subtree_0.right, subtree_1.left))
        # One subtree is empty, and the other is not.
        return False

    return not tree or check_symmetric(tree.left, tree.right)

if __name__ == '__main__':
    exit(
        generic_test.generic_test_main('is_tree_symmetric.py',
                                       'is_tree_symmetric.tsv', is_symmetric))
```

```
def is_symmetric(tree):
    def check_symm(subtree0, subtree1):
        if not subtree0 and not subtree1:
            return True

        elif subtree0 and subtree1:
            return (subtree0.data == subtree1.data
                and check_symm(subtree0.left, subtree1.right)
                and check_symm(subtree0.right, subtree1.left))
        return False

    return check_symm(tree.left, tree.right)
        or not tree
```

Compute LCA when nodes have parent pointers

A brute force approach is to traverse one node and store them in a hash table and then traverse the other node and check for the first common node by using the hash table. O(h) time, space.

Better approach.

- If two nodes are in the same depth, we can move (traverse) the two nodes at the same time and return the first common node.
- The problem lies when the two input node have different depths, then we need to move the node w/ greater depth to the same depth as the other node.

```
def lca(n0, n1):
    def get_depth(node):
        depth = 0
        while node.parent:
            depth += 1
            node = node.parent
        return depth

    depth0, depth1 = map(get_depth, (n0, n1))
    if depth1 > depth0:
        n0, n1 = n1, n0
    depth_diff = abs(n1 - n0)
    while depth_diff:
        n0 = n0.parent
        depth_diff -= 1

    while n0 is not n1:
        n0, n1 = n0.parent, n1.parent
    return n0
```

9.4

```
import functools
from typing import Optional

 om binary_tree_with_parent_prototype import BinaryTreeNode
from test_framework import generic_test
from test_framework.binary_tree_utils import must_find_node
from test_framework.test_failure import TestFailure
from test_framework.test_utils import enable_executor_hook


def lca(node0: BinaryTreeNode,
        node1: BinaryTreeNode) -> Optional[BinaryTreeNode]:
    def get_depth(node):
        depth = 0
        while node.parent:
            depth += 1
            node = node.parent
        return depth

    depth0, depth1 = map(get_depth, (node0, node1))
    # Makes node0 as the deeper node in order to simplify the code.
    if depth1 > depth0:
        node0, node1 = node1, node0

    # Ascends from the deeper node.
    depth_diff = abs(depth0 - depth1)
    while depth_diff:
        node0 = node0.parent
        depth_diff -= 1

    # Now ascends both nodes until we reach the LCA.
    while node0 is not node1:
        node0, node1 = node0.parent, node1.parent
    return node0
```

This is given

→ getdepth(node0), ... (node1)

} when two nodes meet during traversal.

```
@enable_executor_hook
def lca_wrapper(executor, tree, node0, node1):
    result = executor.run(
        functools.partial(lca, must_find_node(tree, node0),
                          must_find_node(tree, node1)))

    if result is None:
        raise TestFailure('Result can\'t be None')
    return result.data


if __name__ == '__main__':
    exit(
        generic_test.generic_test_main('lowest_common_ancestor_with_parent.py',
                                       'lowest_common_ancestor.tsv',
                                       lca_wrapper))
```

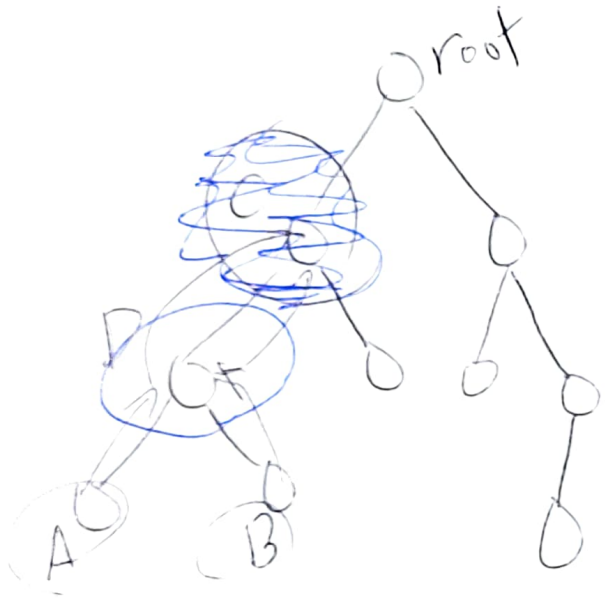
9.4 7/14

- Parent Pointer?

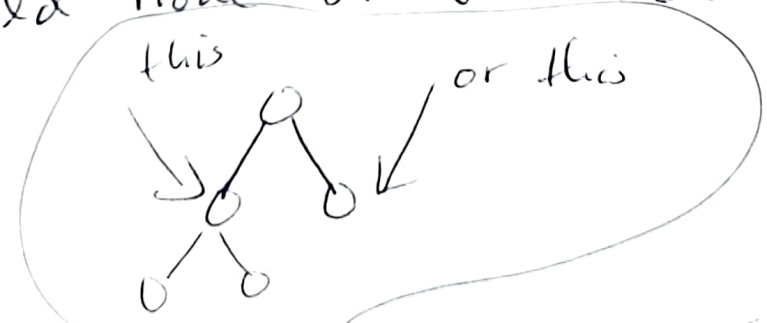
Input: Tree nodes (n1, n2)

Output: Node (tree?) or some value

- Probably has to do w/ traversal.
- Maybe like 9.1 and start from the two leaves and traversal upto root.



- Wouldn't the LCA always be one of the child node of the root?



/0min
Quit

In this case we just need to know if the input nodes are in the left or right (or left & right)