

Day-22

Problem 1: Binary Tree to Double Linked List

class Node:

```
def __init__(self, data):
```

```
    self.data = data
```

```
    self.left = None
```

```
    self.right = None
```

```
def binary_tree_to_dll(root):
```

```
    if root is None:
```

```
        return None
```

```
    left_head = binary_tree_to_dll(root.left)
```

```
    while left_head is not None and left_head.right is not None:
```

```
        left_head = left_head.right
```

```
    if left_head is not None:
```

```
        left_head.right = root
```

```
        root.left = left_head
```

```
    right_head = binary_tree_to_dll(root.right)
```

```
    if right_head is not None:
```

```
        right_head.left = root
```

```
        root.right = right_head
```

```
    return root if root.left is None else root.left
```

```
def print_dll(head):
```

```
    while head is not None:
```

```
        print(head.data, end=" ")
```

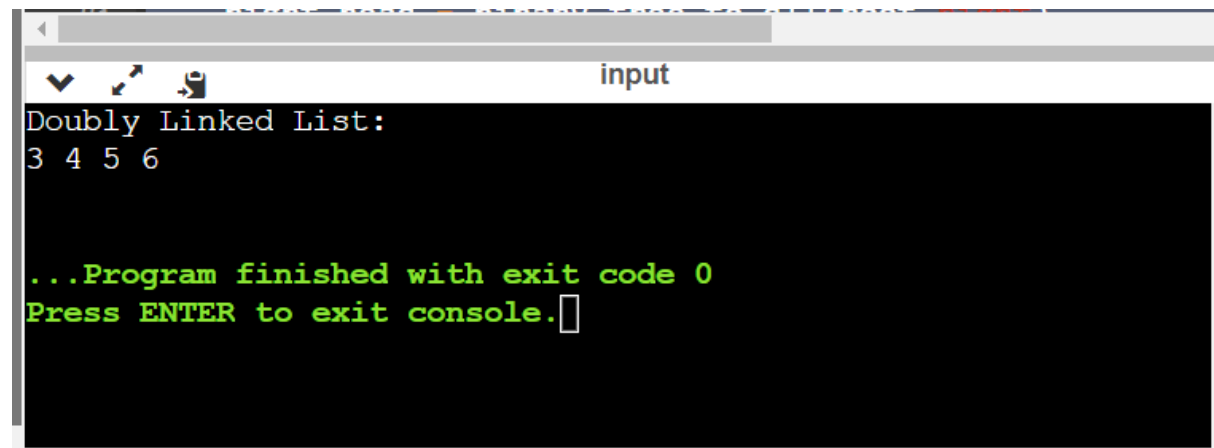
```
    head = head.right  
    print()
```

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

```
dll_head = binary_tree_to_dll(root)
```

```
print("Doubly Linked List:")
```

```
print_dll(dll_head)
```



```
input  
Doubly Linked List:  
3 4 5 6  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Problem 2: Find median in a stream of running integer

```
class TreeNode:
```

```
def __init__(self, val):
```

```
    self.val = val
```

```
    self.left = None
```

```
    self.right = None
```

```
class MedianFinder:
```

```
def __init__(self):
```

```
    self.root = None
```

```
    self.count = 0
```

```
def addNum(self, num):
```

```
    if self.root is None:
```

```
        self.root = TreeNode(num)
```

```
    else:
```

```
        self._insert(self.root, num)
```

```
    self.count += 1
```

```
def _insert(self, node, num):
```

```
    if num < node.val:
```

```
        if node.left is None:
```

```
            node.left = TreeNode(num)
```

```
        else:
```

```
            self._insert(node.left, num)
```

```
    else:
```

```
        if node.right is None:
```

```
            node.right = TreeNode(num)
```

```
        else:
```

```
self._insert(node.right, num)
```

```
def findMedian(self):
```

```
    if self.count == 0:
```

```
        return None
```

```
    if self.count % 2 == 1:
```

```
        return self._find_kth_smallest(self.root, (self.count + 1) // 2)
```

```
    else:
```

```
        left = self._find_kth_smallest(self.root, self.count // 2)
```

```
        right = self._find_kth_smallest(self.root, self.count // 2 + 1)
```

```
        return (left + right) / 2
```

```
def _find_kth_smallest(self, node, k):
```

```
    left_count = self._count_nodes(node.left)
```

```
    if k == left_count + 1:
```

```
        return node.val
```

```
    elif k <= left_count:
```

```
        return self._find_kth_smallest(node.left, k)
```

```
    else:
```

```
        return self._find_kth_smallest(node.right, k - left_count - 1)
```

```
def _count_nodes(self, node):
```

```
    if node is None:
```

```
        return 0
```

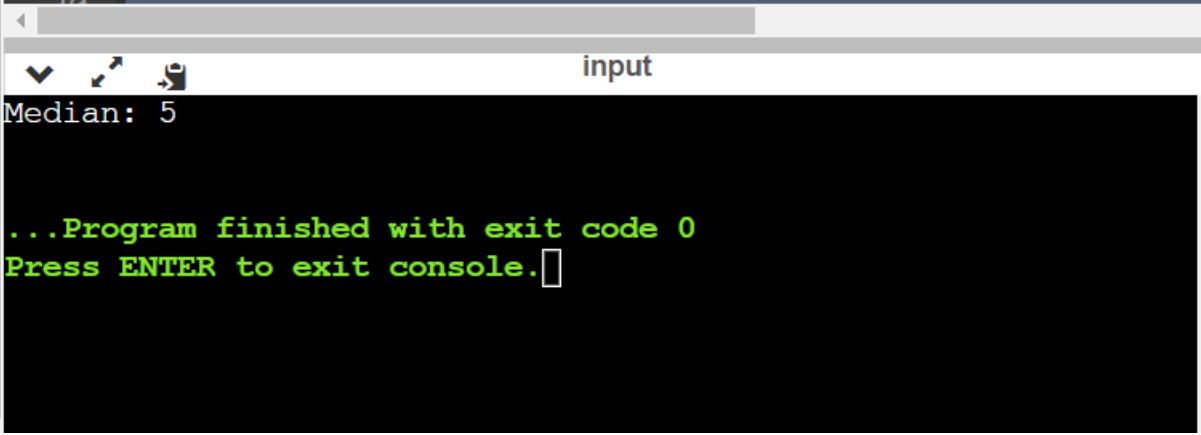
```
    return 1 + self._count_nodes(node.left) + self._count_nodes(node.right)
```

```
mf = MedianFinder()
```

```
mf.addNum(5)
```

```
mf.addNum(10)
```

```
mf.addNum(1)
mf.addNum(7)
mf.addNum(3)
median = mf.findMedian()
print("Median:", median)
```



The screenshot shows a terminal window with a title bar that includes standard window controls and the title 'input'. The terminal output is as follows:

```
Median: 5

...Program finished with exit code 0
Press ENTER to exit console.
```

The output indicates that the program successfully calculated the median as 5 and has finished execution.

Problem 3: Find K-th largest element in a stream

```
import heapq
```

```
class TreeNode:
```

```
    def __init__(self, val):
```

```
        self.val = val
```

```
        self.left = None
```

```
        self.right = None
```

```
class KthLargest:
```

```
    def __init__(self, k, nums):
```

```
        self.k = k
```

```
        self.heap = []
```

```
        for num in nums:
```

```
            self.add(num)
```

```
    def add(self, val):
```

```
        if len(self.heap) < self.k:
```

```
            heapq.heappush(self.heap, val)
```

```
        else:
```

```
            heapq.heappushpop(self.heap, val)
```

```
    def find_kth_largest(self):
```

```
        return self.heap[0]
```

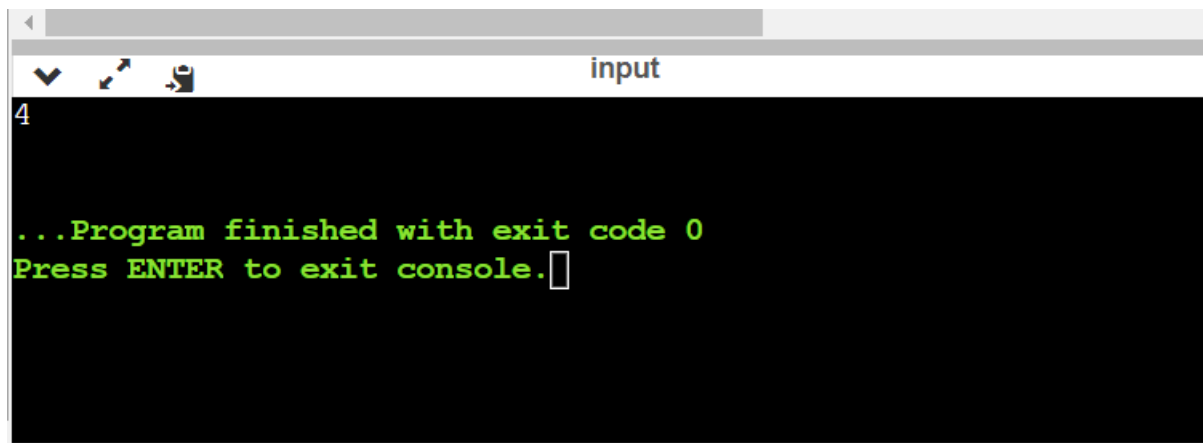
```
# Test case
```

```
nums = [4, 5, 8, 2]
```

```
k = 3
```

```
kth_largest = KthLargest(k, nums)
```

```
print(kth_largest.find_kth_largest())
```



Problem 4: Distinct numbers in Window

```
class TreeNode:
```

```
    def __init__(self, value):
```

```
        self.value = value
```

```
        self.left = None
```

```
        self.right = None
```

```
def distinctNumbersInWindow(root, window_size):
```

```
    distinct_numbers = set()
```

```
    window_numbers = set()
```

```
    def traverse(node):
```

```
        if not node:
```

```
            return
```

```
        window_numbers.add(node.value)
```

```
        distinct_numbers.add(node.value)
```

```
        if len(window_numbers) > window_size:
```

```
            window_numbers.remove(node.value)
```

```
        traverse(node.left)
```

```
        traverse(node.right)
```

```
    traverse(root)
```

```
    return len(distinct_numbers)
```

```
root = TreeNode(1)
```

```
root.left = TreeNode(2)
```

```
root.right = TreeNode(3)
```

```
root.left.left = TreeNode(2)
```



```
root.left.right = TreeNode(4)
```

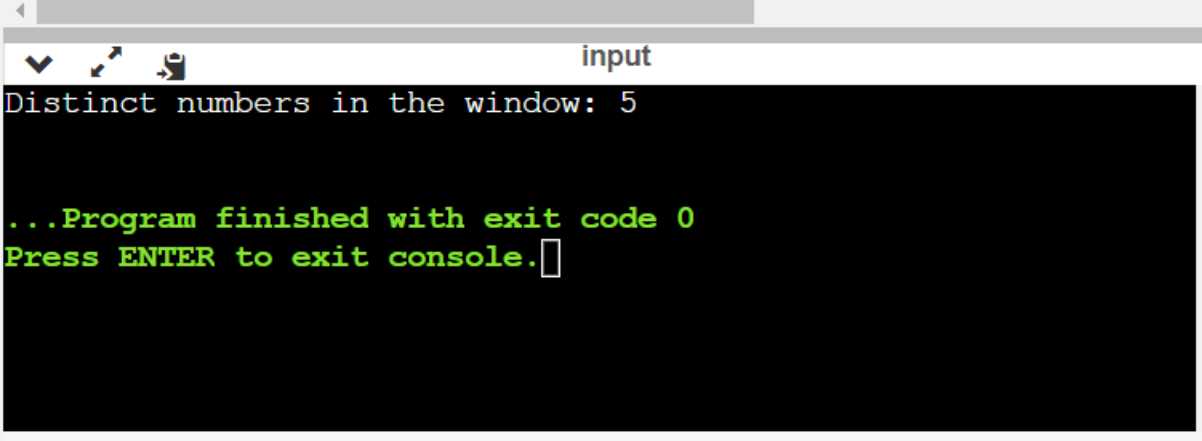
```
root.right.left = TreeNode(3)
```

```
root.right.right = TreeNode(5)
```

```
window_size = 3
```

```
distinct_count = distinctNumbersInWindow(root, window_size)
```

```
print("Distinct numbers in the window:", distinct_count)
```



```
input
Distinct numbers in the window: 5

...Program finished with exit code 0
Press ENTER to exit console.
```

Problem 5: K-th largest element in an unsorted array.

```
def find_kth_largest(nums, k):
```

```
    nums.sort(reverse=True)
```

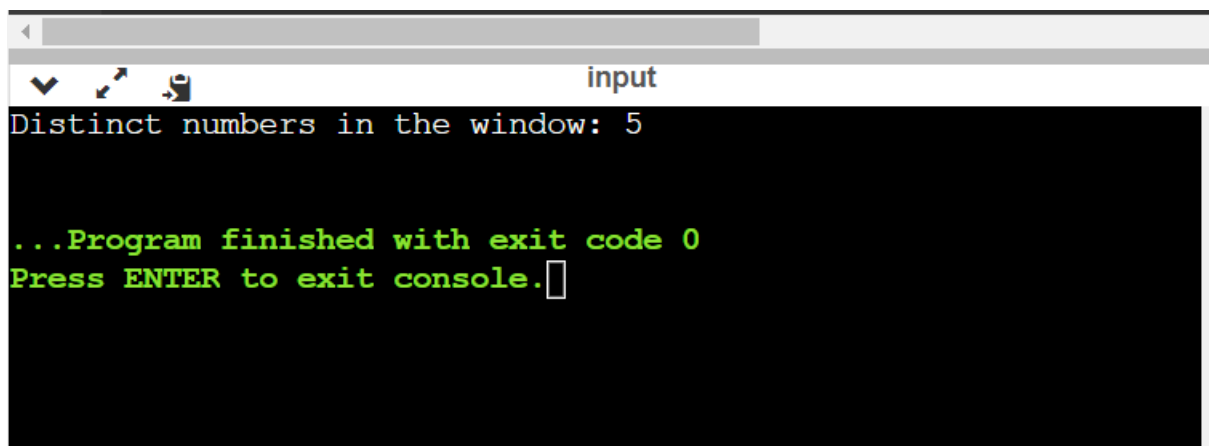
```
    return nums[k - 1]
```

```
arr = [3, 1, 5, 2, 4]
```

```
k_value = 2
```

```
kth_largest = find_kth_largest(arr, k_value)
```

```
print(f"The {k_value}th largest element is: {kth_largest}")
```



```
Distinct numbers in the window: 5

...Program finished with exit code 0
Press ENTER to exit console.
```

Problem 6: Flood-fill Algorithm

```
def flood_fill(image, sr, sc, new_color):  
    if image[sr][sc] == new_color:  
        return image  
  
    rows, cols = len(image), len(image[0])  
    original_color = image[sr][sc]  
  
    def dfs(row, col):  
        if (  
            row < 0  
            or row >= rows  
            or col < 0  
            or col >= cols  
            or image[row][col] != original_color  
        ):  
            return  
  
        image[row][col] = new_color  
        dfs(row + 1, col)  
        dfs(row - 1, col)  
        dfs(row, col + 1)  
        dfs(row, col - 1)
```

```
dfs(sr, sc)
```

```
return image
```

```
image = [
```

```
    [1, 1, 1, 1, 1],
```

```
    [1, 1, 1, 1, 1],
```

```
    [1, 1, 0, 0, 1],
```

```
    [1, 0, 1, 0, 1],
```

```
    [1, 0, 0, 1, 1],
```

```
]
```

```
sr = 2
```

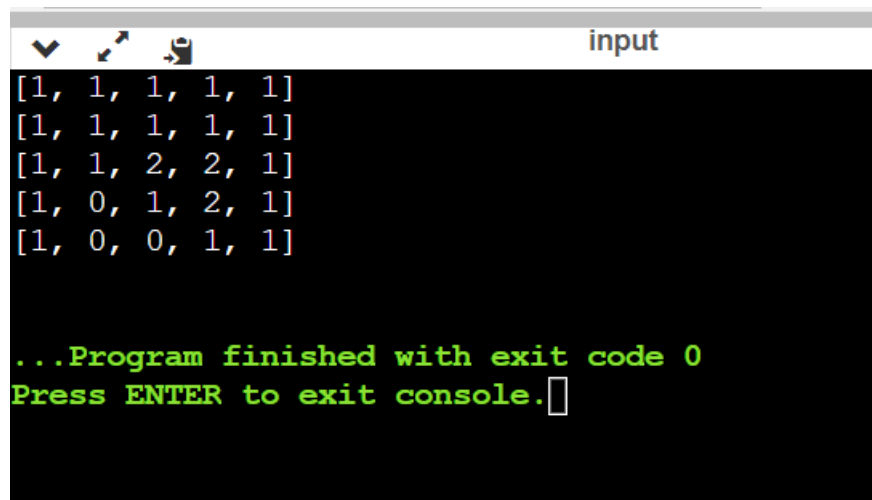
```
sc = 2
```

```
new_color = 2
```

```
filled_image = flood_fill(image, sr, sc, new_color)
```

```
for row in filled_image:
```

```
    print(row)
```



```
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 2, 2, 1]
[1, 0, 1, 2, 1]
[1, 0, 0, 1, 1]

...Program finished with exit code 0
Press ENTER to exit console.
```