```
from math import log2

from ArrayQueue import ArrayQueue
from ArrayStack import ArrayStack
from ChainingHashTableMap import ChainingHashTableMap
from LinkedBinaryTree import LinkedBinaryTree








Node = LinkedBinaryTree.Node

#THis is just for help
def construct_tree(nodes):
   n = len(nodes)
   levels = int(log2(n + 1))

   if n == 0:
       return LinkedBinaryTree()
   if pow(2, levels) != n + 1:
       raise Exception("Invalid node list")

   node_list = [Node(val) if val is not None else None for val in
nodes]

   for level in range(levels - 1):
       for i in range(2 ** level - 1, 2 ** (level + 1) - 1):
           if node_list[i] is None and (node_list[2 * i + 1] is not
None or node_list[2 * i + 2] is not None):
               raise Exception("Invalid node list")
           if node_list[2 * i + 1] is not None:
               node_list[i].left = node_list[2 * i + 1]
               node_list[2 * i + 1].parent = node_list[i]
           if node_list[2 * i + 2] is not None:
               node_list[i].right = node_list[2 * i + 2]
               node_list[2 * i + 2].parent = node_list[i]

   tree = LinkedBinaryTree(node_list[0])
```

```python
    return tree


# https://leetcode.com/problems/longest-palindrome/
def longest_palindrome(s: str) -> int:
    counter = ChainingHashTableMap()
    for letter in s:
        if letter not in counter:
            counter[letter] = 0
        counter[letter] += 1

    s = 0
    has_odd = False
    for letter in counter:
        if counter[letter] % 2 == 0:
            s += counter[letter]
        else:
            s += counter[letter] // 2 * 2
            has_odd = True


    return s + has_odd


# https://leetcode.com/problems/least-number-of-unique-integers-
after-k-removals/
def find_least_num_of_unique_ints(arr: list[int], k: int) -> int:
    counter = ChainingHashTableMap()
    for num in arr:
        if num not in counter:
            counter[num] = 0
        counter[num] += 1

    sorted_freq = sorted([counter[num] for num in counter])
    max_num = len(sorted_freq)
    for i in range(len(sorted_freq)):
        if k < sorted_freq[i]:
            max_num = i
            break
        else:
            k -= sorted_freq[i]
    return len(sorted_freq) - max_num
```

```python
#
https://leetcode.com/problems/tuple-with-same-product/description/
def tuple_same_product(nums: list[int]) -> int:
   prods = ChainingHashTableMap()
   for num1 in nums:
       for num2 in nums:
           if num1 != num2:
               p = num1 * num2
               if p not in prods:
                   prods[p] = []
               prods[p].append((num1, num2))
   s = 0
   for p in prods:
       n = len(prods[p])
       s += n * (n - 2)
   return s




# Two sum for a BST in nlogn.
def has_val(root, val):
   if not root:
       return False

   if root.data == val:
       return True

   if root.data > val:
       return has_val(root.left, val)

   return has_val(root.right, val)


def findTarget(self, root: Node, k: int) -> bool:
   def preorder(node):
       if not node:
           return
       yield node.data
       yield from preorder(node.left)
       yield from preorder(node.right)
```

```python
    for node in preorder(root):
        if node != k - node and has_val(root, k - node):
            return True

    return False


# Write a function that returns a list of lists, where every list
is a level of the binary tree from left to right
def tree_levels(root: LinkedBinaryTree.Node) -> list:
    if not root:
        return []
    ans = []
    nodes = []
    current = 0
    q = ArrayQueue()
    q.enqueue((root, 0))
    while not q.is_empty():
        node, level = q.dequeue()
        if level == current:
            nodes.append(node.data)
        else:
            ans.append(nodes)
            nodes = [node.data]
            current += 1
        if node.left:
            q.enqueue((node.left, level + 1))
        if node.right:
            q.enqueue((node.right, level + 1))
    ans.append(nodes)
    return ans


def maxSumBST(root):
    max_val = 0
    def helper(root):
        if not root:
            return 0, True, None, None
        (sum_left, is_bst_left, left_min, left_max) = helper(root.left)
        (sum_right, is_bst_right, right_min, right_max) = helper(root.right)

        left_val = left_max or -math.inf
        right_val = right_min or math.inf
        min_tree = left_min or root.val
        max_tree = right_max or root.val

        is_bst = is_bst_left and is_bst_right and (left_val < root.val <
right_val)
        new_sum = sum_left + sum_right + root.val

        if (is_bst):
            nonlocal max_val
            max_val = max(max_val, new_sum)

        return new_sum, is_bst, min_tree, max_tree

    helper(root)
    return max_val



# https://leetcode.com/problems/check-if-word-is-valid-after-
substitutions/
def is_valid(s: str) -> bool:
    stack = ArrayStack()
    for c in s:
        if c == "c":
            if len(stack) >= 2:
                c1 = stack.pop()
                if stack.top() + c1 + c != "abc":
                    stack.push(c1)
                    stack.push(c)
                else:
                    stack.pop()
        else:
            stack.push(c)

    return len(stack) == 0

from DoublyLinkedList import DoublyLinkedList
from ArrayStack import ArrayStack
from HashMap import ChainingHashTableMap
from ArrayQueue import ArrayQueue
```

```
#  Should do this and the one with hashmap


class Company:
   def __init__(self, prirority_dept):
       self.dll = DoublyLinkedList()
       self.stack = ArrayStack()
       self.priority = priority_dept

   def __len__(self):
       return len(self.dll)

   def addEmployee(self, employee_name, dept_name):
       new_node = self.dll.add_first((employee_name, dept_name))
       if dept_name == self.priority:
           self.stack.push(new_node)

   def fire(self):
       if len(self.dll) == 0:
           raise Exception("No employees to fire")
       emp_data = self.dll.delete_first()
       if emp_data[1] == self.priority:
           self.stack.pop()
       return emp_data[0]

   def fireFromPriorityDept(self):
       if self.stack.is_empty():
           return self.fire()
       node_to_delete = self.stack.pop()
       emp_data = self.dll.delete_node(node_to_delete)
       return emp_data[0]

   def displayEmployees(self):
       for emp_data in self.dll:
           print(emp_data[0])





# Must do

class AlbumCatalog:
   def __init__(self):
       self.artists = ChainingHashTableMap()
       self.all_albums = ArrayQueue()

   def __len__(self):
       return len(self.all_albums)
```

```
    def add_album(self, artist_name, album_title):
        if artist_name not in self.artists:
            self.artists[artist_name] = DoublyLinkedList()
        album_node = self.artists[artist_name].add_first(album_title)
        self.all_albums.enqueue(album_node)

    def displayArtistAlbums(self, artist_name):
        for album in self.artists[artist_name]:
            print(album)

    def displayAlbums(self):
        for i in range(len(self.all_albums)):
            album_node = self.all_albums.dequeue()
            print(album_node.data)
            self.all_albums.enqueue(album_node)
```

● Write a class of a company that lays off employees, prioritizing the most recently joined employees

○ company = Company(priority_dept) – initializes company and the priority_dept to lay off

○ len(company) – returns number of employees currently working

○ company.addEmployee(employee_name, dept_name) – hires new employee with their assigned department

○ company.fire() – fires the last employee added, return name of the fired employee

○ company.fireFromPriorityDept() – fires the last employee added that works in a priority department,
return name of the fired employee, if there are no employees in the priority department it should work like a regular fire

○ company.displayEmployees() – displays the list of all company employees

- Write a class of one’s album inventory
  - catalog = AlbumCatalog();
  - len(catalog) – returns how many albums are in the catalog -> O(1)
  - catalog.add_album(artist_name, album_title) – adds album by artist -> Average O(1)
  - catalog.displayArtistAlbums(artist_name) – returns a list of all artist’s albums from oldest to newest
->average O(k) where k is number of albums of artist
  - catalog.deleteArtist(artist_name) - removes all albums belonging to artist-> average O(k)
  - catalog.displayAlbums() – return list of all albums from oldest to newest -> O(n)

```
class AlbumCatalog():
    def __init__(self):
        self.artists = ChainingHashTableMap()
        self.albums = DoublyLinkedList()

    def __len__(self):
        return len(self.albums)

    def is_empty(self):
        return len(self.albums) == 0

    def add_album(self, artist, title):
        node = self.albums.add_last((artist, title))
        if artist not in self.artists:
            self.artists[artist] = DoublyLinkedList()
        self.artists[artist].add_last(node)

    def  display_artist(self, artist):
        if artist not in self.artists:
            raise Exception("Artist does not exist in catalog")

        return [album.data[1] for album in self.artists[artist]]

    def delete_artist(self, artist):
        if artist not in self.artists:
            raise Exception("Artist does not exist in catalog")
        for node in self.artists[artist]:
            self.albums.delete_node(node)

        del self.artists[artist]

    def display_albums(self):
        return [album[1] for album in self.albums]
```

a