Final review problem set

During the recitation, we are going to solve: 1, 7, 8, 9, 10, 5, 3.

1. Find missing elements of a range

Given a python list arr of distinct elements and a range [low, high], **print** all numbers that are in range, but not in array. The missing elements should be **printed** in sorted order.

```
Input: arr = [10, 12, 11, 15],
       low = 10, high = 15
       Output: 13, 14


Input: arr = [1, 14, 11, 51, 15],
       low = 50, high = 55
Output: 50, 52, 53, 54
```

Note:

Len(arr) = N

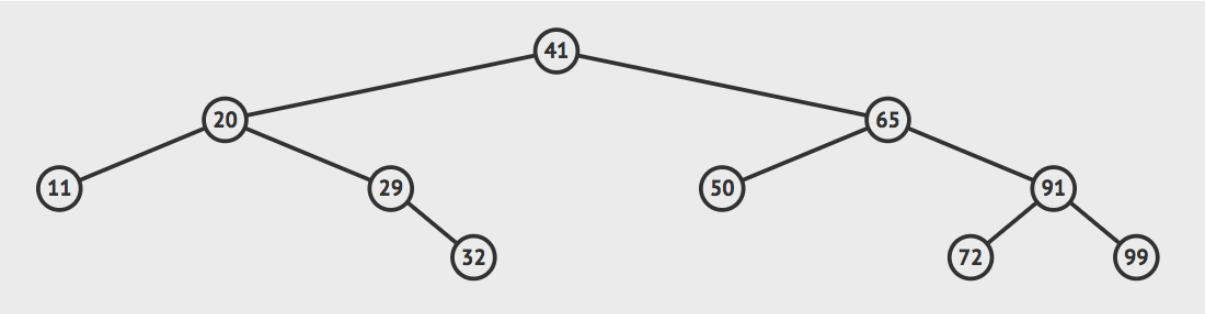
Range(low, high) = K

Required runtime complexity: O(N + K) expected.

```
def print_missing(arr, low, high):
    """ To do for Map & Hashing question 2. """
    # Your code
```

2. AVL Trees.

Suppose I have the AVL Tree below. Draw the result AVL Tree after inserting 73.



3. Implement function, switch_first(self, l2) for doubly linked list with sentinels. Once called, the first node of self and l2 should be swapped.

(to make the question more difficult, you should swap nodes, not elements)

For example, your implementation should provide the following behavior:

L1: head<-->35<-->22<-->10<-->5<-->tail

L2: head<-->“Hi”<-->”MM”<-->”QQ”<-->”WW”<-->tail

L1.switch_first(l2)

L1: head<-->“Hi”<-->22<-->10<-->5<-->tail

L2: head<-->35<-->”MM”<-->”QQ”<-->”WW”<-->tail

```
1.  class DoubleLinkedList:
2.    """A base class providing a doubly linked list representation."""
3.
4.    #-------------------------- nested _Node class --------------------------
5.    # nested _Node class
6.    class _Node:
7.      """Lightweight, nonpublic class for storing a doubly linked node."""
8.      __slots__ = '_element', '_prev', '_next'            # streamline memory
9.
10.     def __init__(self, element, prev, next):      # initialize node's fields
11.       self._element = element                     # user's element
12.       self._prev = prev                           # previous node reference
13.       self._next = next                           # next node reference
14.
15.   #-------------------------- list constructor --------------------------
16.
17.   def __init__(self):
18.     """Create an empty list."""
19.     self._head = self._Node(None, None, None)
20.     self._tail = self._Node(None, None, None)
21.     self._head._next = self._tail
22.     self._tail._prev = self._head
23.     self._size = 0                                  # number of elements
24.
25.   #-------------------------- public accessors --------------------------
26.
27.   def __len__(self):
28.     """Return the number of elements in the list."""
29.     return self._size
30.
31.   def is_empty(self):
32.     """Return True if list is empty."""
33.     return self._size == 0
34.
35.   def switch_first(self, l2):
36.     """Switch first node of self, with first node of l2."""
37.     # To do for question 4
38.
```

```
def switch_first(self, l2):
    """
    :param l2: other doubly linked list, with sentinels

    Switch first node of self list, with first node of l2.

    :return: nothing, modify self list and l2 in place.
    """
    # To do
```

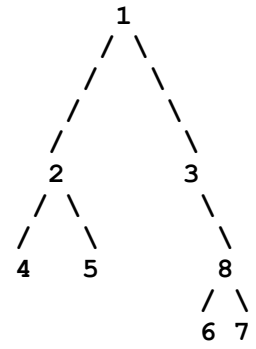
4. maximum tree width

Give a python implementation for the function:

```
def max_width(tree):
```

When called, it returns the **maximum width** of the given binary tree. (Width: how many nodes in a level of the given tree)

For example:



For the above tree, tree1:
width of level 1 is 1,
width of level 2 is 2,
width of level 3 is 3
width of level 4 is 2.

max_width(tree) should return 3.

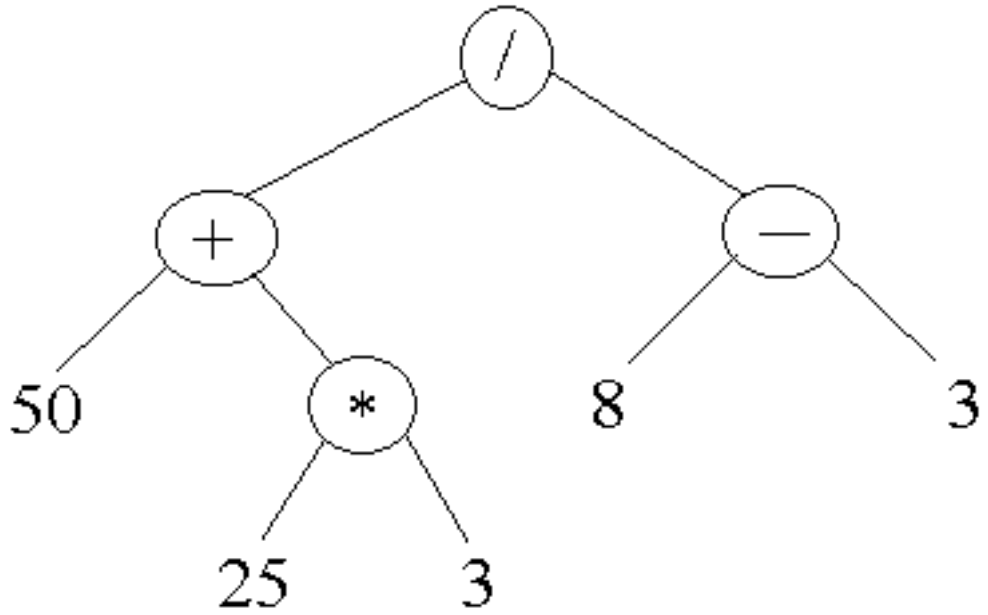
Implementation requirements:

1. Your implementation should base on the SimpleTree class.
2. Your implementation should use O(N) runtime.
3. You can define helper functions if you like.

```
class SimpleTree:
    def __init__(self, element, left=None, right=None, parent=None):
        self.element = element
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.element)


def max_width(tree):
    """
    :param tree: SimpleTree -- Initial call is the root node.
    :return: Int – the maximum tree width
    """
    # to do for question 5
```

```
def max_width(tree):
    """
    :param tree: SimpleTree - Initial call is the root node.
    :return: Int - the maximum tree width.
    """
    # To do
```

5. preorder/inorder/postorder traversal of expression tree, and computing the result



Preorder traversal:
Inorder traversal:
Postorder traversal:
Evaluate result:

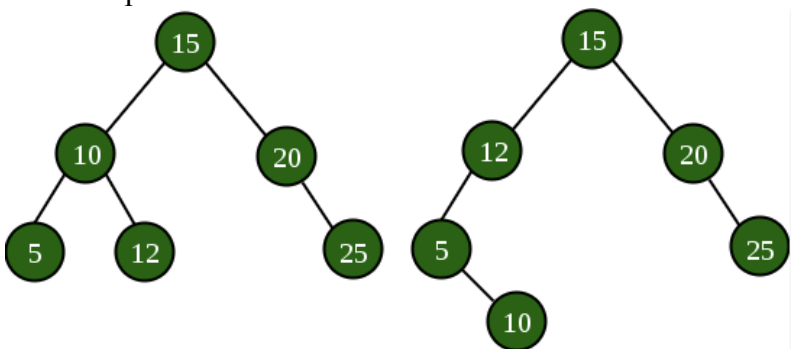
6. Check if two bst contains the same set of elements.

Give a python implementation for the function:

```
def checkBSTs(bst1, bst2):
```

Given two Binary Search Trees, return True if they contain the same set of elements, return False otherwise.

For example:



`check_BSTs(bst1, bst2) should return True.`

Implementation requirements:

1. Your implementation should base on the BinarySearchTree class.
2. Your implementation should use O(N) runtime.
3. You can define helper functions if you like.

```
class Empty(Exception):
    def __init__(self, msg):
        self.msg = msg

class Tree:
    class TreeNode:
        def __init__(self, element, parent = None, left = None, right = None):
            self.parent = parent
            self.element = element
            self.left = left
            self.right = right

    #-------------------------- binary tree constructor -------------------------
    def __init__(self):
        """Create an initially empty binary tree."""
        self.root = None
        self.size = 0

    #-------------------------- public accessors --------------------------------
    def __len__(self):
        """Return the total number of elements in the tree."""
        return self.size

  ... code omitted ...


class BinarySearchTree(Tree):


    def insert(self, v):
```

```
31.         """Insert value v into the Binary Search Tree"""
32.         if self.is_empty():
33.             leaf = self.add_root(v)     # from BinaryTree (class Tree)
34.         else:
35.             node = self._subtree_search(self.root, v)
36.             if node._element < v:
37.                 leaf = self.add_right(node, v)
38.             else:
39.                 leaf = self.add_left(node, v)
40.         self._rebalance_insert(leaf)
41.
42. def check_BSTs(bst1, bst2):
43.     """
44.     :param bst1: BinarySearchTree - the first bst
45.     :param bst2: BinarySearchTree - the second bst
46.     :return: True if bst1 and bst2 contains the same set of elements, False
    otherwise.
47.     """
48.
```

```
def check_BSTs(bst1, bst2):
    """
    :param bst1: BinarySearchTree - the first bst
    :param bst2: BinarySearchTree - the second bst
    :return: True if bst1 and bst2 contains the same set of
elements, False otherwise.
    """
    # To do
```

7. Implement preorder traversal without using recursion (Hard)

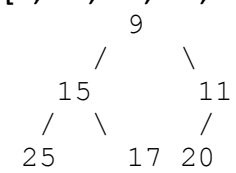
```
class SimpleTree:
    def __init__(self, element, left=None, right=None, parent=None):
        self.element = element
        self.left = left
        self.right = right
        self.parent = parent

    def __str__(self):
        return str(self.element)


def pre_order_print(root):
    """Displays a simple tree in preorder. Can't use recursion."""
    # to do for question 8
```

```
def pre_order_print(root):
    """
    @root: SimpleTree object, you can assume it is the root.

    Displays a simple tree in preorder. Can't use recursion.

    @return: nothing, use print function.
    """
    # To do
```

8. Given an array, give an algorithm to check whether it is representing a min-heap. Return True if the array is representing a min-heap, return False otherwise.

For example:

[9, 15, 11, 25, 17, 20] represents the following binary min-heap.



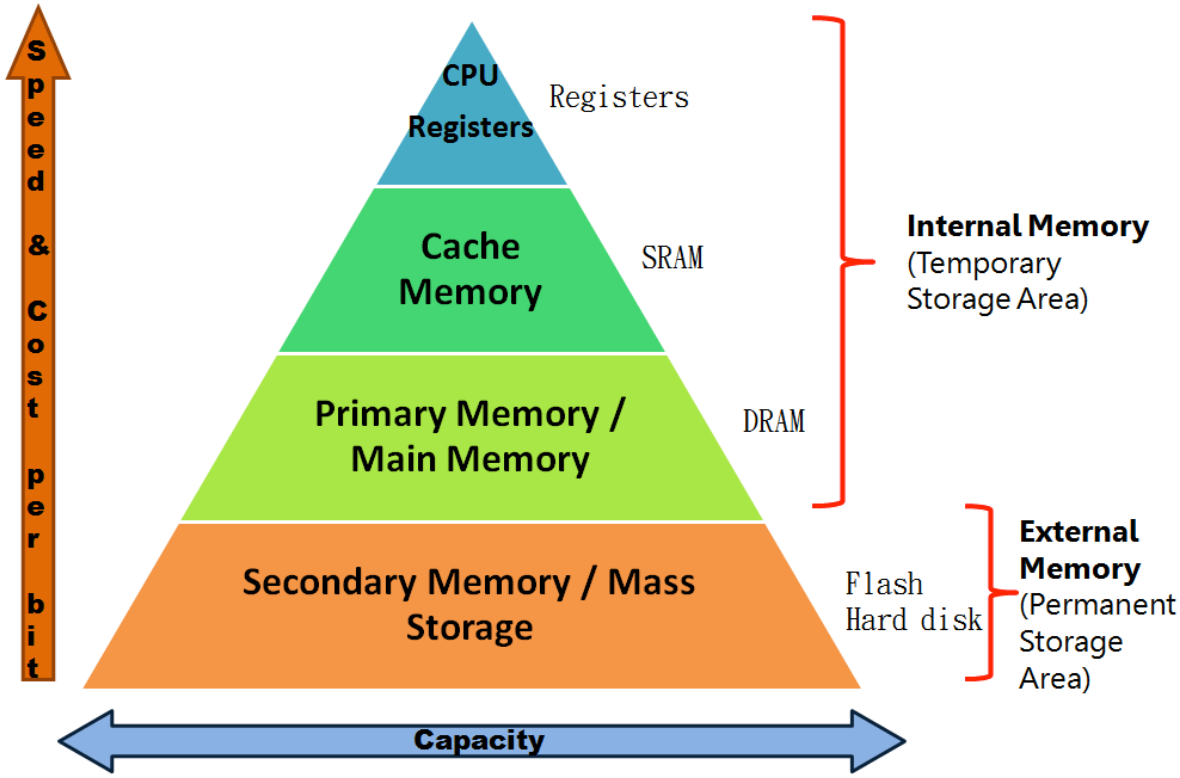

```
>>> is_min_heap([9, 15, 11, 25, 17, 20])  # Min-heap
True
```

Some more examples:

```
>>> is_min_heap([2, 4, 3, 6])  # Min-heap
True
>>> is_min_heap([2, 1, 3, 6])
False
```

```
def is_min_heap(array):
    """
    :param array: List[Int] – the array to check
    :return: True/False
    """
    # Your code
```

9. LRU Cache



In our computer, we have memory hierarchy that runs faster and faster as you go up. This memory hierarchy also gets more and more expensive as you go up. It is important to maintain the recently used data within the fast memory layer. (So you don't have to load that data from hard disk again, loading is slow.)

The LRU cache algorithm is to remove the least recently used memory address when the cache is full and a new memory address is referenced which is not there in cache.

You are given **a sequence of memory addresses to process**.
You are also given **cache size** (or memory size) (Number of memory address numbers that cache can hold at a time).

In this question, you will implement – ***LRU Cache*****.** This class supports:

• *process_next(x)* – process the next memory address x. If x is already in the LRU Cache, LRU Cache should remember x becomes most recently used. If x is not in the LRU Cache, LRU Cache not only remembers x is the most recently used, but also remove the least recently used memory address.
• *pretty_print()* – display (print) all the memory addresses currently being stored in the LRU Cache. The order should start from most recently used, end with the least recently used.

For example, let's demonstrate the following input sequence, with cache size 3:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 2, 5

```
lru = LRUCache(3)      # cache size = 3
lru.process_next(1)    # lru.pretty_print() should display 1
lru.process_next(2)    # lru.pretty_print() should display 2 1
lru.process_next(3)    # lru.pretty_print() should display 3 2 1
lru.process_next(4)    # lru.pretty_print() should display 4 3 2
lru.process_next(1)    # lru.pretty_print() should display 1 4 3
lru.process_next(2)    # lru.pretty_print() should display 2 1 4
lru.process_next(5)    # lru.pretty_print() should display 5 2 1
lru.process_next(1)    # lru.pretty_print() should display 1 5 2       # Note, existing element
lru.process_next(2)    # lru.pretty_print() should display 2 1 5       # Note, existing element
lru.process_next(3)    # lru.pretty_print() should display 3 2 1
lru.process_next(2)    # lru.pretty_print() should display 2 3 1       # Note, existing element
lru.process_next(5)    # lru.pretty_print() should display 5 2 3
```

For example, let's demonstrate the following input sequence, with cache size 5:
1, 5, 4, 6, 6, 2, 7, 5, 1, 2, 2, 2, 3, 3, 4, 7, 3, 1, 2, 1

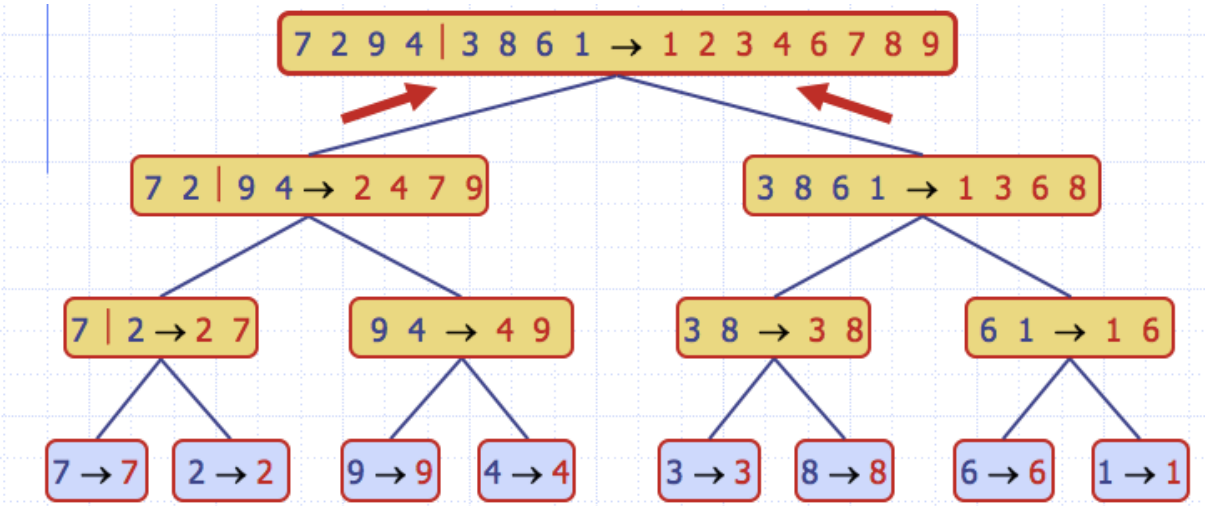
```
lru = LRUCache(5)      # cache size = 5
lru.process_next(1)    # lru.pretty_print() should display 1
lru.process_next(5)    # lru.pretty_print() should display 5 1
lru.process_next(4)    # lru.pretty_print() should display 4 5 1
lru.process_next(6)    # lru.pretty_print() should display 6 4 5 1
lru.process_next(6)    # lru.pretty_print() should display 6 4 5 1    # Note, existing element
lru.process_next(2)    # lru.pretty_print() should display 2 6 4 5 1
lru.process_next(7)    # lru.pretty_print() should display 7 2 6 4 5
lru.process_next(5)    # lru.pretty_print() should display 5 7 2 6 4  # Note, existing element
lru.process_next(1)    # lru.pretty_print() should display 1 5 7 2 6
lru.process_next(2)    # lru.pretty_print() should display 2 1 5 7 6  # Note, existing element
lru.process_next(2)    # lru.pretty_print() should display 2 1 5 7 6  # Note, existing element
lru.process_next(2)    # lru.pretty_print() should display 2 1 5 7 6  # Note, existing element
lru.process_next(3)    # lru.pretty_print() should display 3 2 1 5 7
lru.process_next(3)    # lru.pretty_print() should display 3 2 1 5 7  # Note, existing element
lru.process_next(4)    # lru.pretty_print() should display 4 3 2 1 5
lru.process_next(7)    # lru.pretty_print() should display 7 4 3 2 1
lru.process_next(3)    # lru.pretty_print() should display 3 7 4 2 1  # Note, existing element
lru.process_next(1)    # lru.pretty_print() should display 1 3 7 4 2  # Note, existing element
lru.process_next(2)    # lru.pretty_print() should display 2 1 3 7 4  # Note, existing element
lru.process_next(1)    # lru.pretty_print() should display 1 2 3 7 4  # Note, existing element
```

Implementation requirements:

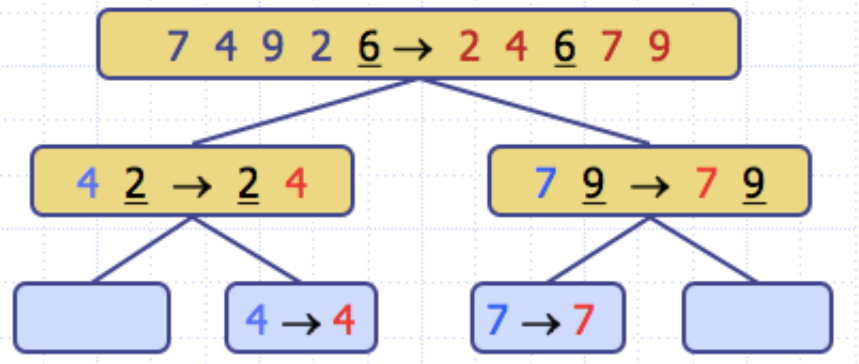
1. You should implement the LRU Cache with two data structures we learned:
    a. Double Linked List
    b. Dictionary (HashTable)
2. Suppose the cache size is N, the required runtime complexities are:
    a. process_next(x) O(1) expected
    b. pretty_print() O(N)
3. **In addition to a Double Linked List, and a Dictionary, you are allowed to use O(1) additional memory.**

```
class LRUCache:

    def __init__(self, capacity):
        self.d = {}
        self.ll = DoubleLinkedList()
        self.cap = capacity


    def process_next(self, x):
        # Your code









    def pretty_print(self):
        # Your code
```

10. Draw the recursion tree for merge sort, like the example below.
The sequence you should be drawing is **8,5,2,0,6,4,5,1**



11. Draw the recursion tree for **inplace_quick_sort** (Textbook version quicksort), looks like the example below. The last element is selected as pivot.



The sequence you should be drawing is **8,5,2,0,6,4,5,1**

12. Show the steps required to do a radix sort on the following set of values when using base 10.

Number set: 346, 22, 31, 212, 157, 102, 568, 435, 8, 14, 5

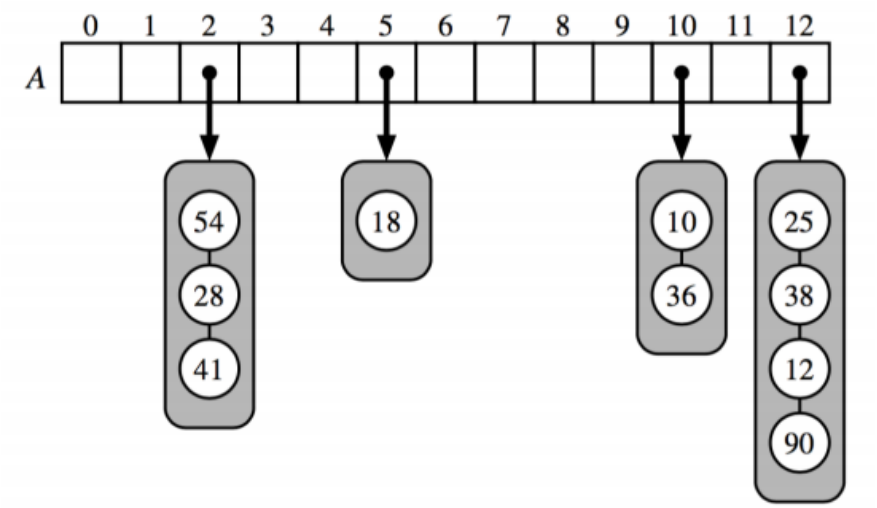
Slots 0 – 9 represents ten queues.

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

14. Consider the following figure from your book, illustrating a hash table, where the hash function used is h(k) = k mod 13:



a. Illustrate in the diagram what will happen if you insert 16. Write “C” next to any changes.

b. Suppose you search for 51 in the has table; it is not there. What items in the hash table will you look when searching for 51? Circle them.