## 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

dict1 = {}

dict2 = {}

for i in arr:

if dict1.get(i) == None:

dict1[i] = 0

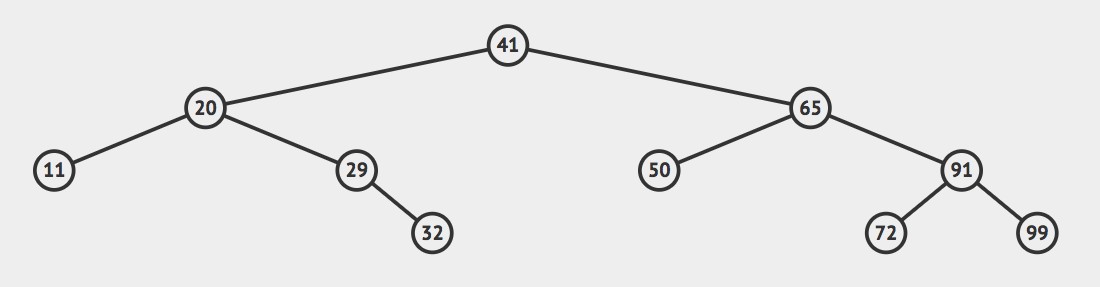
for i in range(low, high+1):

if dict1.get(i) == None:

print(i)

**AVL Trees.**

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



1. **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 --------------------------

1. # nested \_Node class
2. **class** \_Node:
3. """Lightweight, nonpublic class for storing a doubly linked node."""
4. slots = '\_element', '\_prev', '\_next' # streamline memory 9.
5. **def** init\_\_(self, element, prev, next): # initialize node's fields
6. self.\_element = element # user's element
7. self.\_prev = prev # previous node reference
8. self.\_next = next # next node reference 14.

15. #-------------------------- list constructor --------------------------

16.

1. **def** init (self):
2. """Create an empty list."""
3. self.\_head = self.\_Node(None, None, None)
4. self.\_tail = self.\_Node(None, None, None)
5. self.\_head.\_next = self.\_tail
6. self.\_tail.\_prev = self.\_head
7. self.\_size = 0 # number of elements 24.

25. #-------------------------- public accessors --------------------------

26.

1. **def** len (self):
2. """Return the number of elements in the list."""
3. **return** self.\_size 30.
4. **def** is\_empty(self):
5. """Return True if list is empty."""
6. **return** self.\_size == 0 34.
7. **def** switch\_first(self, l2):
8. """Switch first node of self, with first node of l2."""
9. # To do for question 3 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

head1 = self.\_head.\_next #first node

head2 = l2.\_head.\_next

#cut original connection

self.\_head.\_next = head1.\_next

head1.\_next.\_prev = self.\_head

head1.\_prev = None

head1.\_next = None

l2.\_head.\_next = head2.\_next

head2.\_next.\_prev = l2.\_head

head2.\_prev = None

head2.\_next = None

#insert new head

self.\_head.\_next.\_prev = head2

head2.\_next = self.\_head.\_next

head2.\_prev = self.\_head

self.\_head.\_next = head2

l2.\_head.\_next.\_prev = head1

head1.\_next = l2.\_head.\_next

head1.\_prev = l2.\_head

l2.\_head.\_next = head1

1. **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:

**1**

**/ \**

**/ \**

**/ \**

**/ \**

**2 3**

**/ \ \**

**/ \ \**

**4 5 8**

**/ \**

**6 7**

## 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.

4. **class** SimpleTree:

6.

self.element = element

8.

9.

self.right = right

11.

12.

13.

**return** str(self.element)

15.

16.

17.

18.

19.

"""

:param tree: SimpleTree -- Initial call is the root node.

:return: Int – the maximum tree width """

# to do for question 5

14. **def** max\_width(tree):

10. **def** str (self):

self.left = left

7.

**def** init\_\_(self, element, left=None, right=None, parent=None):

5.

## def max\_width(tree): ”””

**:param tree: SimpleTree – Initial call is the root node.**

**:return: Int – the maximum tree width. ”””**

# To do

curr\_level = []

next\_level = []

curr\_Node = tree.\_element

curr\_level.append(curr\_Node)

max\_width = len(curr\_level)

while len(curr\_level) != 0:

for i in range(len(curr\_level):

curr\_Node = curr\_level[i]

if curr\_Node.\_left is not None:

next\_level.append(curr\_Node.\_left)

if curr\_Node.\_right is not None:

next\_level.append(curr\_Node.\_right)

#curr\_level is done

while len(curr\_level) != 0:

curr\_level.pop()

for i in next\_level:

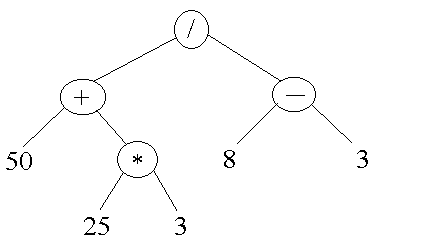
curr\_level.append(i)

if len(curr\_level) > max\_width:

max\_width = len(curr\_level)

return max\_width

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



Preorder Traversal: / + 50 \* 25 3 - 8 3

Inorder Traversal: 50 + 25 \* 3 / 8 - 3

Post order Traversal: 50 25 3 \* + 8 3 - /

Result: (50+ 25\*3 )/ (8-3) = 125/5 = 25

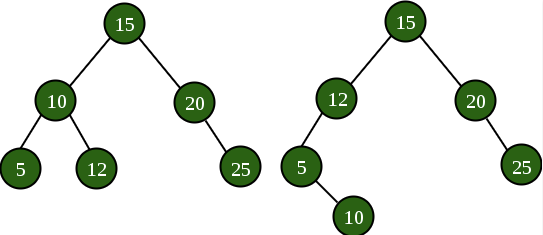
1. **Check if two bst contains the same set of elements.**

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.

1. **class** Empty(Exception):

3.

self.msg = msg

5. **class** Tree:

7.

**def** init (self, element, parent = None, left = None, right = None):

9.

self.element = element

11.

self.right = right

13.

#-------------------------- binary tree constructor --------------------------

15.

"""Create an initially empty binary tree."""

17.

self.size = 0

19.

#-------------------------- public accessors ---------------------------------

21.

"""Return the total number of elements in the tree."""

23.

24. ... code omitted ...

26.

28.

29.

30.

**def** insert(self, v):

27. **class** BinarySearchTree(Tree):

25.

**return** self.size

22.

20. **def** len (self):

18.

self.root = None

16.

14. **def** init\_\_(self):

12.

self.left = left

10.

self.parent = parent

8.

**class** TreeNode:

6.

4.

**def** init\_\_(self, msg):

2.

|  |  |  |  |
| --- | --- | --- | --- |
| 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. """   1. :param bst1: BinarySearchTree – the first bst 2. :param bst2: BinarySearchTree – the second bst 3. :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

#traverse two BST

dic1 = check\_BST\_helper(bst1.\_root, {})

dic2 = check\_BST\_helper(bst2.\_root, {})

if len(dic1) != len(dic2):

return False

for i in dic1:

if dic2.get(i) == None:

return False

return True

def check\_BST\_helper(node,dic):

if node == None:

return dic

else:

if dic.get(node.\_element) == None:

dic[node.\_element] = 0

left = check\_BST\_helper(node.\_left, dic)

right = check\_BST\_helper(node.\_right, left)

return right

1. **Implement preorder traversal without using recursion (Hard)**

1. **class** SimpleTree:

3.

self.element = element

5.

self.right = right

7.

9.

10.

11.

**return** str(self.element)

13.

14.

"""Displays a simple tree in preorder. Can’t use recursion."""

# to do for question 6

12. **def** pre\_order\_print(root):

**def** str (self):

8.

self.parent = parent

6.

self.left = left

4.

**def** init\_\_(self, element, left=None, right=None, parent=None):

2.

**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

print self.\_element

go left

go right

prev = []

curr\_Node = root

prev.append(curr\_Node)

checked= {}

while len(prev) != 0：

if checked[curr\_Node] != ‘Marked’:

checked[curr\_Node] = ‘Marked’

print(curr\_Node.\_element)

if curr\_Node.\_left != None:

prev.append(curr\_Node)

curr\_Node = curr\_Node.\_left

elif curr\_Node.\_right != None:

prev.append(curr\_Node)

curr\_Node = curr\_Node.\_right

else:

curr\_Node = prev.pop()

else:

#back again, check right

if curr\_Node.\_right != None:

curr\_Node = curr\_Node.\_right

1. 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.

9

/ \

15 11

/ \ /

25 17 20

# >>> 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

n = 0

while n <= len(array)//2:

#check its left child and right child

if 2\*n+1 < len(array):

if array[n] > array[2n+1]:

return False

if 2\*n+2 < len(array):

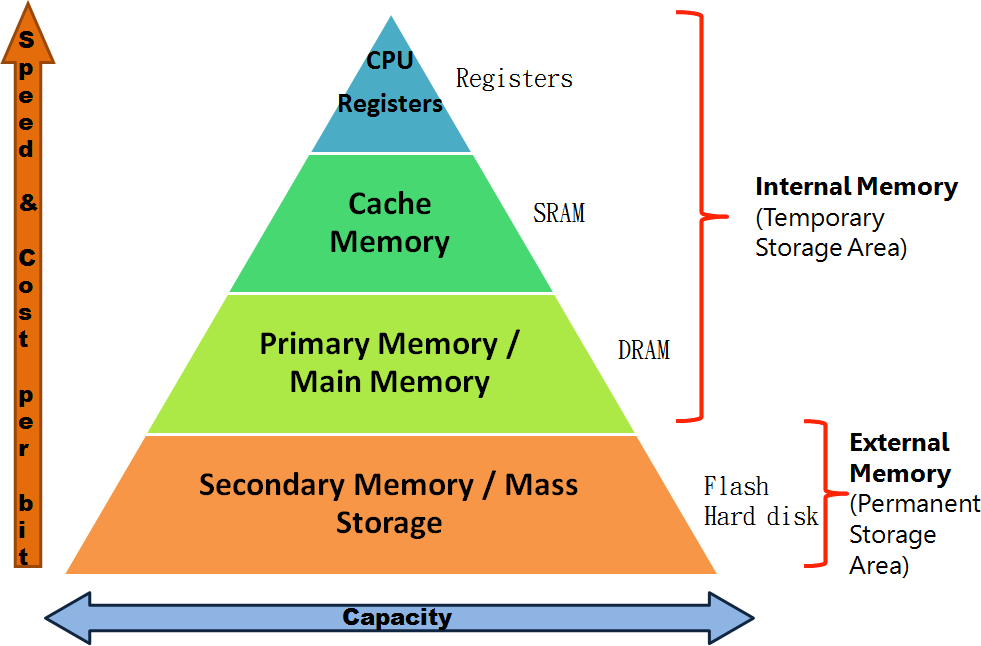
if array[n] > array[2n+2]:

return False

n += 1

return True

1. 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:
   1. Double Linked List
   2. Dictionary (HashTable)
2. Suppose the cache size is N, the required runtime complexities are:
   1. process\_next(x) O(1) expected
   2. 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**

**#check if x already exists**

**if self.d.get(x) != None:**

**#put it to the front**

**#cut off old connection**

**x.\_prev.\_next = x.\_next**

**x.\_next.\_prev = x.\_prev**

**x.\_next = self.ll.\_head.\_next**

**self.ll.\_head.\_next.\_prev = x**

**self.ll.\_head.\_next = x**

**x.\_prev = self.ll.\_head**

**self.ll.\_size += 1**

**else:**

**#add it to the dict**

**self.d[x] = 0**

**#insert into list, release the least used element if #capacity reached**

**x.\_next = self.ll.\_head.\_next**

**self.ll.\_head.\_next.\_prev = x**

**self.ll.\_head.\_next = x**

**x.\_prev = self.ll.\_head**

**self.ll.\_size += 1**

**if self.ll.\_size > self.cap:**

**#remove last**

**self.ll.\_tail.\_prev.\_prev.\_next=self.ll.\_tail**

**self.ll.\_tail.\_prev = self.ll.\_tail.\_prev.\_prev**

**self.ll.\_size -= 1**

# **def** pretty\_print(self):

**# Your code**

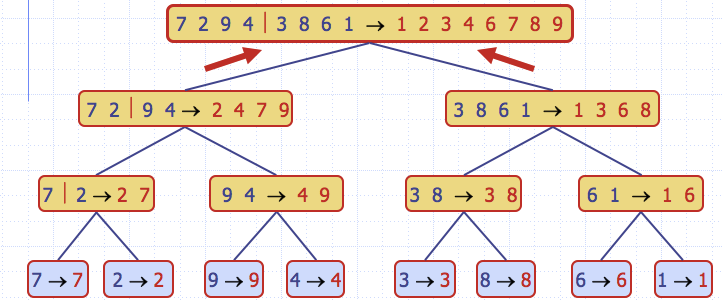
curr\_Node = self.ll.\_head.\_next

while curr\_Node != self.ll.\_tail:

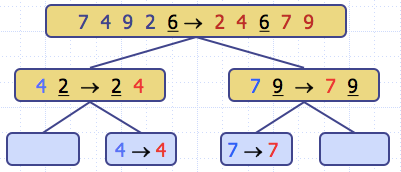
print(curr\_Node.\_element)

curr\_Node = curr\_Node.\_next

1. 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**



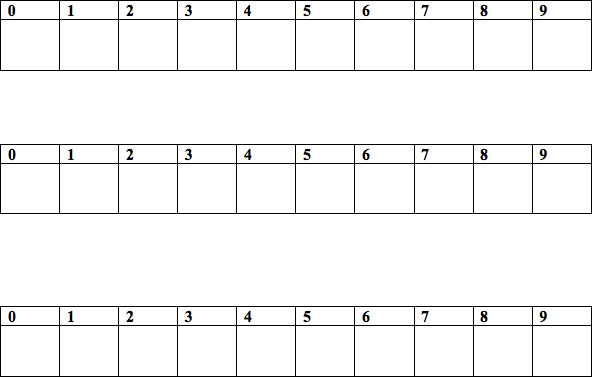
1. Draw the recursion tree for **inplace\_quick\_sort** (Textbook version quicksort), looks like the example below. The last element is selected as pivot.



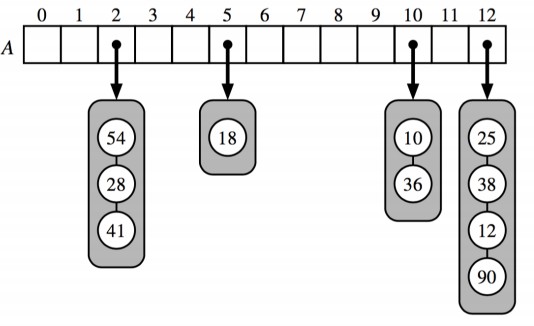
The sequence you should be drawing is **8,5,2,0,6,4,5,1**

1. 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.

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



1. Illustrate in the diagram what will happen if you insert 16. Write “C” next to any changes.
2. 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.