##########################################################################################################  
##########################################################################################################  
##########################################################################################################  
##########################################################################################################  
# adding needed classes  
import math  
  
#all quesiton 1 classes are from notes and given examples in class  
class PriorityQueueBase:  
 *"""Abstract base class for a priority queue."""* # ------------------------------ nested \_Item class ------------------------------  
 class \_Item:  
 *"""Lightweight composite to store priority queue items."""* \_\_slots\_\_ = '\_key', '\_value'  
  
 def \_\_init\_\_(self, k, v):  
 self.\_key = k  
 self.\_value = v  
  
 def \_\_lt\_\_(self, other):  
 return self.\_key < other.\_key # compare items based on their keys  
  
 def \_\_repr\_\_(self):  
 return '({0},{1})'.format(self.\_key, self.\_value)  
  
 # ------------------------------ public behaviors ------------------------------  
 def is\_empty(self): # concrete method assuming abstract len  
 *"""Return True if the priority queue is empty."""* return len(self) == 0  
  
 def \_\_len\_\_(self):  
 *"""Return the number of items in the priority queue."""* raise NotImplementedError('must be implemented by subclass')  
  
 def add(self, key, value):  
 *"""Add a key-value pair."""* raise NotImplementedError('must be implemented by subclass')  
  
 def min(self):  
 *"""Return but do not remove (k,v) tuple with minimum key.  
  
 Raise Empty exception if empty.  
 """* raise NotImplementedError('must be implemented by subclass')  
  
 def remove\_min(self):  
 *"""Remove and return (k,v) tuple with minimum key.  
  
 Raise Empty exception if empty.  
 """* raise NotImplementedError('must be implemented by subclass')  
  
  
class \_DoublyLinkedBase:  
 *"""A base class providing a doubly linked list representation."""* # -------------------------- nested \_Node class --------------------------  
 # nested \_Node class  
 class \_Node:  
 *"""Lightweight, nonpublic class for storing a doubly linked node."""* \_\_slots\_\_ = '\_element', '\_prev', '\_next' # streamline memory  
  
 def \_\_init\_\_(self, element, prev, next): # initialize node's fields  
 self.\_element = element # user's element  
 self.\_prev = prev # previous node reference  
 self.\_next = next # next node reference  
  
 # -------------------------- list constructor --------------------------  
  
 def \_\_init\_\_(self):  
 *"""Create an empty list."""* self.\_header = self.\_Node(None, None, None)  
 self.\_trailer = self.\_Node(None, None, None)  
 self.\_header.\_next = self.\_trailer # trailer is after header  
 self.\_trailer.\_prev = self.\_header # header is before trailer  
 self.\_size = 0 # number of elements  
  
 # -------------------------- public accessors --------------------------  
  
 def \_\_len\_\_(self):  
 *"""Return the number of elements in the list."""* return self.\_size  
  
 def is\_empty(self):  
 *"""Return True if list is empty."""* return self.\_size == 0  
  
 # -------------------------- nonpublic utilities --------------------------  
  
 def \_insert\_between(self, e, predecessor, successor):  
 *"""Add element e between two existing nodes and return new node."""* newest = self.\_Node(e, predecessor, successor) # linked to neighbors  
 predecessor.\_next = newest  
 successor.\_prev = newest  
 self.\_size += 1  
 return newest  
  
 def \_delete\_node(self, node):  
 *"""Delete nonsentinel node from the list and return its element."""* predecessor = node.\_prev  
 successor = node.\_next  
 predecessor.\_next = successor  
 successor.\_prev = predecessor  
 self.\_size -= 1  
 element = node.\_element # record deleted element  
 node.\_prev = node.\_next = node.\_element = None # deprecate node  
 return element  
  
  
class PositionalList(\_DoublyLinkedBase):  
 *"""A sequential container of elements allowing positional access."""* # -------------------------- nested Position class --------------------------  
 class Position:  
 *"""An abstraction representing the location of a single element."""* def \_\_init\_\_(self, container, node):  
 *"""Constructor should not be invoked by user."""* self.\_container = container  
 self.\_node = node  
  
 def element(self):  
 *"""Return the element stored at this Position."""* return self.\_node.\_element  
  
 def \_\_eq\_\_(self, other):  
 *"""Return True if other is a Position representing the same location."""* return type(other) is type(self) and other.\_node is self.\_node  
  
 def \_\_ne\_\_(self, other):  
 *"""Return True if other does not represent the same location."""* return not (self == other) # opposite of \_\_eq\_\_  
  
 # ------------------------------- utility methods -------------------------------  
 def \_validate(self, p):  
 *"""Return position's node, or raise appropriate error if invalid."""* if not isinstance(p, self.Position):  
 raise TypeError('p must be proper Position type')  
 if p.\_container is not self:  
 raise ValueError('p does not belong to this container')  
 if p.\_node.\_next is None: # convention for deprecated nodes  
 raise ValueError('p is no longer valid')  
 return p.\_node  
  
 def \_make\_position(self, node):  
 *"""Return Position instance for given node (or None if sentinel)."""* if node is self.\_header or node is self.\_trailer:  
 return None # boundary violation  
 else:  
 return self.Position(self, node) # legitimate position  
  
 # ------------------------------- accessors -------------------------------  
 def first(self):  
 *"""Return the first Position in the list (or None if list is empty)."""* return self.\_make\_position(self.\_header.\_next)  
  
 def last(self):  
 *"""Return the last Position in the list (or None if list is empty)."""* return self.\_make\_position(self.\_trailer.\_prev)  
  
 def before(self, p):  
 *"""Return the Position just before Position p (or None if p is first)."""* node = self.\_validate(p)  
 return self.\_make\_position(node.\_prev)  
  
 def after(self, p):  
 *"""Return the Position just after Position p (or None if p is last)."""* node = self.\_validate(p)  
 return self.\_make\_position(node.\_next)  
  
 def \_\_iter\_\_(self):  
 *"""Generate a forward iteration of the elements of the list."""* cursor = self.first()  
 while cursor is not None:  
 yield cursor.element()  
 cursor = self.after(cursor)  
  
 # ------------------------------- mutators -------------------------------  
 # override inherited version to return Position, rather than Node  
 def \_insert\_between(self, e, predecessor, successor):  
 *"""Add element between existing nodes and return new Position."""* node = super().\_insert\_between(e, predecessor, successor)  
 return self.\_make\_position(node)  
  
 def add\_first(self, e):  
 *"""Insert element e at the front of the list and return new Position."""* return self.\_insert\_between(e, self.\_header, self.\_header.\_next)  
  
 def add\_last(self, e):  
 *"""Insert element e at the back of the list and return new Position."""* return self.\_insert\_between(e, self.\_trailer.\_prev, self.\_trailer)  
  
 def add\_before(self, p, e):  
 *"""Insert element e into list before Position p and return new Position."""* original = self.\_validate(p)  
 return self.\_insert\_between(e, original.\_prev, original)  
  
 def add\_after(self, p, e):  
 *"""Insert element e into list after Position p and return new Position."""* original = self.\_validate(p)  
 return self.\_insert\_between(e, original, original.\_next)  
  
 def delete(self, p):  
 *"""Remove and return the element at Position p."""* original = self.\_validate(p)  
 return self.\_delete\_node(original) # inherited method returns element  
  
 def replace(self, p, e):  
 *"""Replace the element at Position p with e.  
 Return the element formerly at Position p.  
 """* original = self.\_validate(p)  
 old\_value = original.\_element # temporarily store old element  
 original.\_element = e # replace with new element  
 return old\_value # return the old element value  
  
  
##########################################################################################################  
##########################################################################################################  
##########################################################################################################  
##########################################################################################################  
  
  
class UnsortedPriorityQueue(PriorityQueueBase): # base class defines \_Item  
 *"""A min-oriented priority queue implemented with an unsorted list."""* # ------------------------------ public behaviors ------------------------------  
 def \_\_init\_\_(self):  
 *"""Create a new empty Priority Queue."""* self.\_data = PositionalList()  
  
 def \_\_len\_\_(self):  
 *"""Return the number of items in the priority queue."""* return len(self.\_data)  
  
 def add(self, key, value):  
 newest = self.\_Item(key, value)  
 walk = self.\_data.last()  
 while walk is not None and newest < walk.element():  
 walk = self.\_data.before(walk)  
 if walk is None:  
 self.\_data.add\_first(newest)  
 else:  
 self.\_data.add\_after(walk, newest)  
  
 def min(self):  
 if self.is\_empty():  
 print("priority Queue is empty")  
 return  
 p = self.\_data.first()  
 item = p.element()  
 return (item.\_key, item.\_value)  
  
 def remove\_min(self):  
 if self.is\_empty():  
 print("priority Queue is empty")  
 return  
 item = self.\_data.delete(self.\_data.first())  
 return (item.\_value)  
  
  
#############################################################################  
# QUESTION 2 CODE  
#############################################################################  
#list for the tree values/nodes  
tree = []  
  
#tree node  
class TreeNode(object):  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
#solutions for the orders  
class YourSolution(object):  
 #lists to hold the order results  
 List1 = []  
 List2 = []  
  
 #inorder transversal  
 def inorderTraversal(self, root):  
 #if there is no root, just exit  
 if root is None:  
 return  
 else:  
 #recursively call the left nodes  
 x = (self.inorderTraversal(self, root.left))  
 #add the value to the list if it isn't none  
 if x is not None:  
 self.List1.append(x)  
 #adding in the root if it isnt none  
 y = (root.val)  
 if y is not None:  
 self.List1.append(y)  
 #recursively call the right nodes and add it to the list if it isn't none  
 z = (self.inorderTraversal(self, root.right))  
 if z is not None:  
 self.List1.append(z)  
  
 def preorderTraversal(self, root):  
 #the root is first, unless it is none  
 if(root is None):  
 return  
 y = root.val  
 #if the root is not None, add it to the list  
 if y is not None:  
 self.List2.append(y)  
 #if the left tree exists, pass it recursively  
 if root.left is not None:  
 self.preorderTraversal(self, root.left)  
 # if the right tree exists, pass it recursively  
 if root.right is not None:  
 self.preorderTraversal(self, root.right)  
  
  
# testing a priority queue with insertion sort (for it's insertion of elements)  
test = UnsortedPriorityQueue()  
q1L = [7,4,8,2,3,9]  
#build queue  
test.add(q1L[0], str(q1L[0]))  
test.add(q1L[1], str(q1L[1]))  
test.add(q1L[2], str(q1L[2]))  
test.add(q1L[3], str(q1L[3]))  
test.add(q1L[4], str(q1L[4]))  
test.add(q1L[5], str(q1L[5]))  
  
#queue is sorted with insertion sort as it is inserted inside the queue  
print("Answer for Question #1")  
#printing the list by removal  
print(test.remove\_min(), end=", ")  
print(test.remove\_min(), end=", ")  
print(test.remove\_min(), end=", ")  
print(test.remove\_min(), end=", ")  
print(test.remove\_min(), end=", ")  
print(test.remove\_min())  
#spacing  
print()  
print()  
  
# testing for quesiton 2  
#holder for the place at tree  
place = 0  
#sentinel value and used for user input  
cont = 0  
#n is the break out value  
print("type 'n' to stop inputting")  
while cont != "n":  
 cont = input("Enter an integer:")  
 #exit if the cont is the exit value  
 if cont == "n":  
 break  
 #if null is typed, pass a None  
 if cont == "null":  
 tree.append(TreeNode(None))  
 else:  
 #pass an integer and make it a node  
 tree.append(TreeNode(cont))  
  
#calculate the limit to go into the list; otherwise it creates an error  
#limit is used for connecting the tree nodes in the tree list  
limit = len(tree) - math.pow(2, (math.log2(len(tree) + 1) - 1))  
  
#loop for creating tree from nodes in tree list  
#the value for place, is correlated where its children are  
while place < limit:  
 #assign left and right of place  
 tree[place].left = tree[place + place + 1]  
 tree[place].right = tree[place + place + 2]  
 place += 1  
 #increment place  
  
#instance of YourSolution for testing  
test2 = YourSolution  
test2.inorderTraversal(test2, tree[0])  
test2.preorderTraversal(test2, tree[0])  
#print results  
print("Inorder is:")  
print(test2.List1)  
print("Preorder is:")  
print(test2.List2)