Here are the answers except for the additional questions 以下为除附加题之外的答案

```
class Node:
    def __init__(self, key = None, next = None):
        self.key = key
        self.next = next


class CBT():
    def __init__(self):
        self._root = None
        self._size = 0                  #Number of nodes（节点数量）

    #Get the number of nodes（获取节点数量）
    def size(self):
        return self._size

    #Print node（打印节点）
    def visit(self):
        currentNode = self._root
        while currentNode != None:
            print(currentNode.key)
            currentNode = currentNode.next


    #Gets the node whose index is specified as i（取指定索引为 i 的节点）
    def getNode(self, pos):
        #Define the current node to be equal to the list head node（定义当前节点等于链表头节点）
        currentNode = self._root
        #Start at the head of the list until you find the location of the parent, or the current node is None（从链表头开始找，直到找到父节点的位置，或者当前节点为 None）
        while pos > 0 and currentNode != None:
            #Move the current node to the next node（把当前节点置为下一节点）
            currentNode = currentNode.next
            pos -= 1
        return currentNode


    #Gets the parent node whose index is i（取得索引为 i 节点的父节点）
    def getParent(self, i):
        #Gets the location of the parent of the node you want to find（取要查找节点的父节点的位置）
        pos = (i + 1) // 2 - 1
        return self.getNode(pos)

    #Gets the left child node whose index is node i（取得索引为 i 节点的左孩子节点）
    def getChildLeft(self, i):
```

```
        #Gets the position of the left child node of the node you want to find（取要查找节点的左孩子节点的位置）
        pos = (i + 1) * 2 - 1
        return self.getNode(pos)

    #Gets the right child node whose index is node i（取得索引为 i 节点的右孩子节点）
    def getChildRight(self, i):
        #Gets the position of the right child node of the node you want to find（取要查找节点的右孩子节点的位置）
        pos = (i + 1) * 2
        return self.getNode(pos)



class MinPQ(CBT):
    def __init__(self):
        super().__init__()


    #Compare whether the second node is less than or equal to the first node（比较第二个节点是否小于等于第一节点）
    def less(self,first, second):
        return self.getNode(first).key >= self.getNode(second).key

    #Swap the positions of the first and second nodes（交换第一个节点和第二个节点的位置）
    def swap(self,first, second):
        nodeFirst = self.getNode(first)
        nodeSecond = self.getNode(second)

        key = nodeFirst.key
        nodeFirst.key = nodeSecond.key
        nodeSecond.key = key

    #Upper float node（上浮子节点）
    def swim(self,k):
        step = 0                #Number of executions（执行次数）
        #If the child node is less than or equal to the value of the parent, the positions of the two are swapped until the child node is larger than the parent（如果孩子节点小于等于父节点的值，就交换两者的位置，直到孩子节点大于父节点为止）
        while k > 0 and self.less((k + 1) // 2 -1, k):
            self.swap((k + 1) // 2 - 1, k)
            k = (k + 1) // 2 - 1
            step += 1

        return step

    #Sinking parent（下沉父节点）
```

```
    def sink(self, k):
        step = 0                #Number of executions (执行次数)
        while 2 * (k + 1) - 1 < self.size():
            j = 2 * (k + 1) -1
            if j < self.size() - 1 and  self.less(j, j + 1):
                j = j + 1
            #j is the youngest of the children  (j 是孩子中最小的)
            if not self.less(k, j):
                break

            self.swap(k, j)

            k = j

            step += 1

        return step

    #Insert the node whose priority is key (插入优先级为 key 的节点)
    #The insertion strategy is a bubbling process from one leaf node to the root node. (插入策略是从一个叶子节点到根节点的一趟起泡过程。)
    #Each layer takes @ (1), so the time complexity is O(height)=O(log n). (每一层的操作需耗时@ (1)，因此，时间复杂度为 O (height)=O(log n))
    def insert(self, key):

        step = 0
        #Define a new node with priority key (定义新节点，优先级为 key)
        newNode = Node(key)

        #If the root node is None, insert a new node directly at the root node (如果根节点是 None,直接在根节点插入新节点)
        if self._root == None:
            self._root = newNode
        else:
            #Adds the new node to the end of the list (把新节点加入到链表尾部)
            i = 1
            currentNode = self._root
            while currentNode.next != None:
                currentNode = currentNode.next
                i += 1

            currentNode.next = newNode

            #Float the newly inserted node up to the correct position (把新插入的节点上浮到正确的位置)
            step = self.swim(i)
        #Increase the number of nodes by 1 (节点数量加 1)
```

```
            self._size += 1

            return step

        #Delete the node with the lowest priority (删除优先级最小的节点)
        #The deletion strategy is a sinking process from the root node to a leaf node. (删除策略是从根节点到一个叶子节点的下沉过程。)
        #Each layer takes @ (1), so the time complexity is O(height)=O(log n). (每一层的操作需耗时@ (1)，因此，时间复杂度为 O (height)=O(log n))
        def delMin(self):
            key = self._root

            step = 0
            if self.size() > 1 :
                currentNode = self._root
                while currentNode.next.next != None:
                    currentNode = currentNode.next

                #Move the tail node to the root node (把尾结点移到根节点)
                self._root.key = currentNode.next.key
                currentNode.next = None

                #The number of nodes is reduced by  (1 节点数量减 1)
                self._size -= 1

                #Sinking node (下沉要节点)
                step = self.sink(0)
            else:
                self._root = None
                self._size = 0

            return step


#Test insert performance routine (测试插入性能程序)
import matplotlib.pyplot as plt
def testInsert():
    priorityArray = input("Please enter priority queue (space separated)：")
    listPriority = priorityArray.split(" ")

    listCount = []
    listStep = []

    minPQ = MinPQ()
    i = 0
    for priority in listPriority:
        step = minPQ.insert(priority)
```

```
            listCount.append(i)
            listStep.append(step)

            i += 1

            #print('---', i, '---')
            #minPQ.visit()

        minPQ.visit()


        print(listCount)
        print(listStep)


        plt.title("Priority Queue Insert() Test", fontsize=18)
        plt.scatter(listCount, listStep, s=40)
        # Set the chart title and label the axes (设置图表标题并给坐标轴加上标签)

        # Sets the range of values for each axis (设置每个坐标轴的取值范围)
        plt.axis([0, 10, 0, 10])

        plt.plot(listCount, listStep, linewidth=3)

        plt.show()



#Test Delete performance program Test delete performance program (测试删除性能程序)
def testDelMin():
        priorityArray = input("Please enter priority queue (space separated): ")
        listPriority = priorityArray.split(" ")

        listCount = []
        listStep = []

        minPQ = MinPQ()
        i = 0
        for priority in listPriority:
            minPQ.insert(priority)

        minPQ.visit()

        for n in range(minPQ.size()):

            listCount.append(minPQ.size())
```

```
            step = minPQ.delMin()

            listStep.append(step)

            #print('---', n, '---')
            #minPQ.visit()

        print(listCount)
        print(listStep)


        plt.title("Priority Queue DelMin() Test", fontsize=18)
        plt.scatter(listCount, listStep, s=40)
        # Set the chart title and label the axes（设置图表标题并给坐标轴加上标签）

        # Sets the range of values for each axis（设置每个坐标轴的取值范围）
        plt.axis([0, 10, 0, 10])

        plt.plot(listCount, listStep, linewidth=3)

        plt.show()


testInsert()
testDelMin()
```