**2 Priority queues**

**1. Design a complete binary tree ADT based on a singly linked list, assign the first element’s index as 0. Given an index i, please implement three operations to get its parent, left child, and right child, respectively**

class Node:  
 def \_\_init\_\_(self, key=None, next=None):  
 self.key = key  
 self.next = next  
  
class ADT:  
 def \_\_init\_\_(self):  
 self.head = None  
 self.tail = None  
 self.size = 0  
  
 def insert(self, key):  
 newNode = Node(key, None)  
 if self.tail is None:  
 self.tail = newNode  
 self.head = newNode  
 else:  
 self.tail.next = newNode  
 self.tail = newNode  
 self.size += 1  
  
 def get\_parent(self, i):  
 if i == 0:  
 return None  
 else:  
 return self.find((i - 1) // 2)  
  
 def get\_left\_child(self, i):  
 return self.find(2 \* i + 1)  
  
 def get\_right\_child(self, i):  
 return self.find(2 \* i + 2)  
  
 def find\_node(self, i):  
 if self.head is None:  
 return None  
 current = self.head  
 if i == 0:  
 return current  
 pos = 0  
 while current.next:  
 current = current.next  
 pos += 1  
 if pos == i:  
 return current  
 return None  
  
 def find(self, i):  
 res = self.find\_node(i)  
 if res:  
 return res.key  
 else:  
 return None

**2. Based on the complete binary tree above, design a minimum priority queue. Specifically, you should implement and explain insert() and delMin()**

*#based on the code in question 1*

def del\_min(self):  
 if self.size == 0:  
 return None  
 retval = self.find(0)  
 self.swap(self.head, self.tail)  
 self.del\_tail()  
 self.size -= 1  
 self.perc\_down(0)  
 return retval  
  
def perc\_up(self):  
 i = self.size - 1  
 while (i - 1) // 2 >= 0:  
 node\_i = self.find\_node(i)  
 node\_p = self.find\_node((i - 1) // 2)  
 if node\_i.key < node\_p.key:  
 self.swap(node\_p, node\_i)  
 i = (i - 1) // 2  
  
def perc\_down(self, i):  
 while (i \* 2 + 1) <= self.size - 1:  
 mc\_pos = self.find\_min\_child(i)  
 node\_mc = self.find\_node(mc\_pos)  
 node\_i = self.find\_node(i)  
 if node\_i.key > node\_mc.key:  
 self.swap(node\_i, node\_mc)  
 i = mc\_pos  
  
def del\_tail(self):  
 if self.size == 1:  
 self.tail = None  
 return  
 new\_tail = self.find\_node(self.size - 2)  
 new\_tail.next = None  
 self.tail = new\_tail  
  
def find\_min\_child(self, i):  
 if i \* 2 + 2 > self.size - 1:  
 return i \* 2 + 1  
 else:  
 if self.get\_left\_child(i) < self.get\_right\_child(i):  
 return i \* 2 + 1  
 else:  
 return i \* 2 + 2  
  
def swap(ni, nj):  
 tmp = nj.key  
 nj.key = ni.key  
 ni.key = tmp

*# a minimum priority queue based on ADT*class PriorityQueue:  
 def \_\_init\_\_(self):  
 self.heap = ADT()  
  
 def insert(self, k):  
 self.heap.insert(k)  
 self.heap.perc\_up()  
  
 def delMin(self):  
 return self.heap.del\_min()

The minimum priority queue is actually a minimum heap.

The characteristics of the minimum heap:

1) is a complete binary tree

2) The value of each node in the heap must be less than or equal to the value of each node in its subtree.

* 1. **insert() description**

Insert elements into the min-heap:

1) Insert data to the tail node of the linked list

2) Heaping from bottom to top to meet the characteristics of the minimum heap

* 1. **delMin() description**

Delete the minimum value of the priority queue, that is, delete the first element of the linked list. After deletion, it is also necessary to ensure that the value of each node in the heap must be less than or equal to the value of each node in its subtree.

Steps to remove the minimum element:

1) Exchange the head and tail nodes of the linked list

2) Delete the tail node

3) Heaping from top to bottom to meet the characteristics of the minimum heap

**3. Analyze the time complexity of the methods above in a minimum priority queue (10 marks).**

A complete binary tree of n nodes whose height does not exceed log2n. The process of heaping is compared and exchanged along the path where the nodes are located, so the time complexity of heaping is proportional to the height of the tree, which is O(logn). The main logic of inserting data and deleting the top element of the heap is heapization, so the time complexity of inserting an element into the heap and deleting the top element of the heap is O(logn).

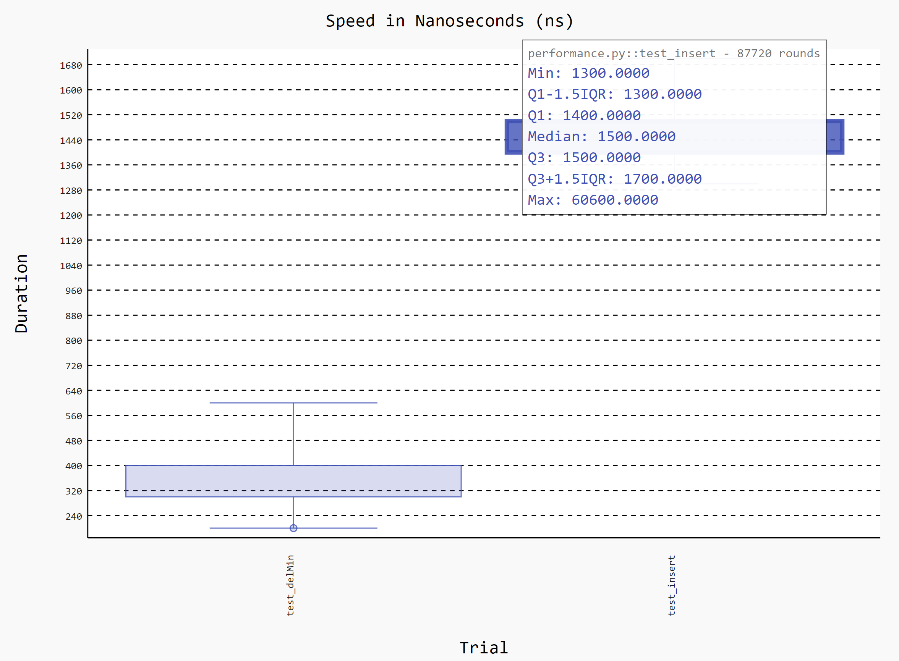
The exchange of tree nodes is performed on a single linked list, and the time complexity of linked list query is O(n).

Thus, the total time complexity is O(nlogn)

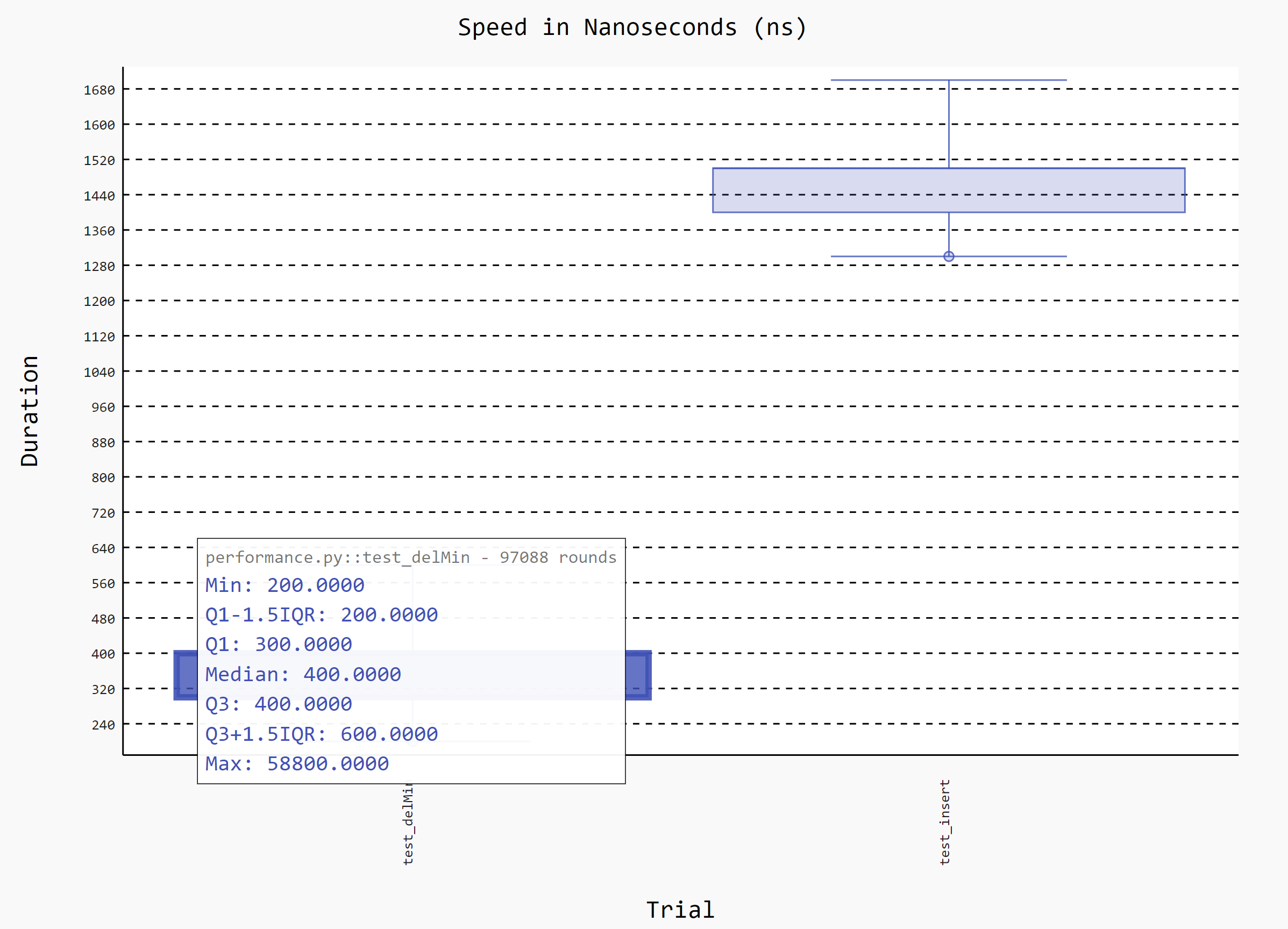
**4. Conduct a simple performance benchmark for your linked list based heap, and visualize the results (5 marks).**

import random  
from priority\_queue import PriorityQueue  
  
def insert(k):  
 PriorityQueue().insert(k)  
  
def delMin(q):  
 q.delMin()  
  
def test\_insert(benchmark):  
 key = random.randint(1, 100)  
 benchmark(insert, key)  
  
def test\_delMin(benchmark):  
 q = PriorityQueue()  
 q.insert(random.randint(1, 100))  
 benchmark(delMin, q)

1. **insert()**



1. **delMin()**



**5. (Bonus) Based on your linked list based heap, draw its corresponding tree structure using graphviz**

import os  
from priority\_queue import PriorityQueue  
from graphviz import Digraph  
  
  
def convert\_to\_digraph(tree):  
 dot = Digraph(comment='Tree Structure')  
 edges = []  
 for i in range(tree.size):  
 dot.node(f'{i}', f'{tree.find(i)}')  
 lc\_pos = 2 \* i + 1  
 rc\_pos = 2 \* i + 2  
 if lc\_pos < tree.size:  
 dot.node(f'{lc\_pos}', f'{tree.find(lc\_pos)}')  
 edges.append(f"{i}{lc\_pos}")  
 if rc\_pos < tree.size:  
 dot.node(f'{rc\_pos}', f'{tree.find(rc\_pos)}')  
 edges.append(f"{i}{rc\_pos}")  
 dot.edges(edges)  
 return dot  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 os.environ['PATH'] = os.pathsep + r'D:\soft\Graphviz\bin'  
 q = PriorityQueue()  
 data = [9, 5, 6, 2, 3]  
 for ele in data:  
 q.insert(ele)  
 dot\_data = convert\_to\_digraph(q.heap)  
 dot\_data.render('./Tree-Structure.gv', view=True)

**Tree Structure**

