Priority queues

1. class CompleteBinaryTree:

```
    def __init__(self):
        self.root = None
    def parent(self, i):
        if i == 0:
            return None
        else:
            return (i-1)//2
    def left_child(self, i):
        return 2*i + 1
    def right_child(self, i):
        return 2*i + 2
```

1. parent(i): Given the index i of a node, the parent index is calculated by (i-1)//2. If i is 0, which is the root of the tree, the function will return None.
2. left_child(i): Given the index i of a node, the left child index is calculated by 2*i + 1.
3. right_child(i): Given the index i of a node, the right child index is calculated by 2*i + 2.

To get the value of each node at an index i, you can traverse the linked list from the root node to the i-th node by using the next pointer stored in each node.

Note that, this implementation assumes that the binary tree is complete, and the number of nodes in the tree is a perfect power of 2, and it is not a general implementation of binary tree, it is only a specialization of binary tree with specific properties.

2. class MinPriorityQueue:

```
    def __init__(self):
        self.tree = CompleteBinaryTree()
        self.size = 0
    def insert(self, key):
        """
        Inserts a new key into the priority queue.
        """
        # create a new node and add it to the end of the list
        new_node = Node(key)
        if self.tree.root is None:
            self.tree.root = new_node
        else:
            current = self.tree.root
```

```
            while current.next is not None:
                current = current.next
            current.next = new_node
        self.size += 1
        # bubble the new node up to its correct position
        i = self.size - 1
        while i > 0 and self.tree.get_node_value(self.tree.parent(i)) > self.tree.get_node_value(i):
            self.tree.swap(i, self.tree.parent(i))
            i = self.tree.parent(i)
    def delMin(self):
        """
        Removes and returns the minimum key from the priority queue.
        """
        if self.size == 0:
            return None
        # remove the minimum key from the root
        min_key = self.tree.get_node_value(0)
        self.tree.set_node_value(0, self.tree.get_node_value(self.size - 1))
        self.size -= 1
        # bubble the new root down to its correct position
        i = 0
        while i < self.size:
            left_child = self.tree.left_child(i)
            right_child = self.tree.right_child(i)
            min_child = i
            if left_child < self.size and self.tree.get_node_value(left_child) < self.tree.get_node_value(min_child):
                min_child = left_child
            if right_child < self.size and self.tree.get_node_value(right_child) <
self.tree.get_node_value(min_child):
                min_child = right_child
            if min_child != i:
                self.tree.swap(i, min_child)
                i = min_child
            else:
```

```
            break

        return min_key
```

- insert(key): This method takes a key as an argument and creates a new node with the given key. It adds this node to the end of the linked list and increment the size of the tree. Then it uses a while loop to compare the value of the new node with its parent and swap them if the new node has a smaller value. This process is called "bubbling up", this way the new node will be placed in the correct position based on its value.
- delMin(): This method removes and returns the minimum key from the priority queue. If the size of the tree is 0, it returns None. Otherwise, it removes the minimum key from the root, which is the first node in the linked list, and set the value of the last node to the root. Then, it uses a while loop to compare the value of the new root with its left and right children, and swap it with the smallest child if necessary. This process is called "bubbling down", this way the new root will be placed in the correct position based on its value. The method continues this process until the new root is in the correct position. This ensures that the minimum key is always at the root of the tree and can be easily removed. At the end, the method returns the minimum key that was removed from the root.
  It is important to note that the tree size is decremented in the first step, so that the last element in the tree can be used as a replacement for the root. This ensures that the tree remains a complete binary tree and the property of a min-heap is maintained.
  It is also important to note that, this implementation of priority queue is based on a complete binary tree which is not a general implementation and it is only a specialization of binary tree with specific properties.

3. insert(): The insert method first adds a new node to the end of the linked list, which takes O(n) time, where n is the number of nodes in the tree. Then, it performs the "bubbling up" operation, which is O(log n) in the average case and O(n) in the worst case. The worst case occurs when the tree is not a complete binary tree, and the new node needs to be moved all the way up to the root. In this case, the overall time complexity of the insert method is O(n + log n) = O(n).

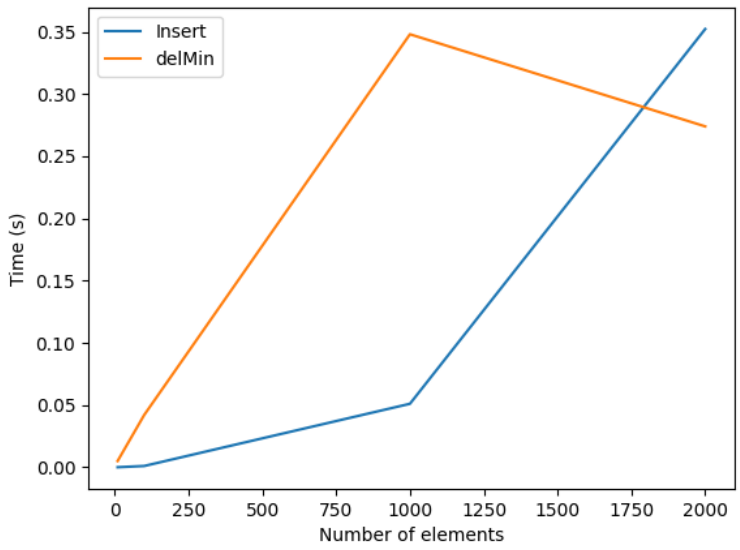
delMin(): The delMin method first removes the root node from the tree, which takes O(1) time. Then, it performs the "bubbling down" operation, which is also O(log n) in the average case and O(n) in the worst case. The worst case occurs when the tree is not a complete binary tree and the new root needs to be moved all the way down to a leaf node. In this case, the overall time complexity of the delMin method is O(1 + log n) = O(log n).

It is important to note that these time complexities are based on the assumption that the tree is a complete binary tree, which ensures that the height of the tree is log n. If the tree is not complete, the time complexities will be worse.

In conclusion, using a singly linked list to implement a minimum priority queue can have a time complexity of O(n) for insert operation and O(log n) for delMin operation, assuming that the tree is a complete binary tree.

4.
```
if __name__ == "__main__":

    # Initialize an empty heap

    heap = MinPriorityQueue()

    # Insert a set number of elements (e.g. 100, 1000, 10000, etc.) into the heap using the insert() method,

    # and measure the time it takes to complete the operation. , 1000, 10000, 100000

    insert_times = []

    elements_count = [10, 100, 1000, 2000]

    for count in elements_count:

        elements = [random.randint(0, 100) for _ in range(count)]

        start_time = time.time()

        for element in elements:
```

```
            heap.insert(element)
        end_time = time.time()
        insert_time = end_time - start_time
        insert_times.append(insert_time)
# Repeat steps 2 and 3 for the delMin() method.
delMin_times = []
for count in elements_count:
        start_time = time.time()
        for _ in range(count):
            heap.delMin()
        end_time = time.time()
        delMin_time = end_time - start_time
        delMin_times.append(delMin_time)


# Plot the results
plt.plot(elements_count, insert_times, label='Insert')
plt.plot(elements_count, delMin_times, label='delMin')
plt.xlabel('Number of elements')
plt.ylabel('Time (s)')
plt.legend()
plt.show()
```



5.