```
import graphviz


class Element:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority


i = 0


class PriorityQueue:
    def __init__(self, size) -> None:
        self.size = size
        self.heap = []

    def shift_up(self):
        global i
        if len(self.heap) == 1:
            return
        last_idx = len(self.heap) - 1
        self.display_as_graph(f"pq_enq_{i}")
        i  += 1
        while last_idx > 0:
            parent_idx = (last_idx - 1) >> 1 #
            if (self.heap[parent_idx].priority < self.heap[last_idx].priority) or (
                self.heap[parent_idx].priority == self.heap[last_idx].priority
                and self.heap[parent_idx].data < self.heap[last_idx].data
            ):
                self.heap[last_idx], self.heap[parent_idx] = (
                    self.heap[parent_idx],
                    self.heap[last_idx],
                )
```

```
            self.display_as_graph(f"pq_enq_{i}")
            i += 1
        last_idx = parent_idx


def enqueue(self, item: Element):
    if len(self.heap) == self.size:
        print("Cannot Enqueue. Queue is full")
        return
    self.heap.append(item)
    self.shift_up()

def shift_down(self, index: int):
    global i
    left = (index << 1) + 1
    right = (index << 1) + 2
    largest = index

    if left < len(self.heap) and self.heap[left].priority > self.heap[largest].priority:
        largest = left
    elif left < len(self.heap) and self.heap[largest].priority == self.heap[largest].priority:
        if self.heap[left].data > self.heap[largest].data:
            largest = left

    if right < len(self.heap) and self.heap[right].priority > self.heap[largest].priority:
        largest = right
    elif right < len(self.heap) and self.heap[right].priority == self.heap[largest].priority:
        if self.heap[right].data > self.heap[largest].data:
            largest = right

    if largest != index:
```

```
            self.heap[largest], self.heap[index] = self.heap[index], self.heap[largest]
            self.display_as_graph(f"pq_d_{i}")
            i += 1
            self.shift_down(largest)

    def dequeue(self) -> Element | None:
        global i
        if len(self.heap) == 0 or self.size == 0:
            print("Cannot dequeue. Queue is empty")
            return None
        self.heap[0], self.heap[len(self.heap) - 1] = self.heap[len(self.heap) - 1], self.heap[0]

        element  = self.heap.pop()
        self.display_as_graph(f"pq_d_{i}")
        i += 1
        self.shift_down(0)
        self.size -= 1
        return element

    def peek(self) -> Element:
        return self.heap[0]

    def display(self):
        if len(self.heap) == 0:
            print("Empty queue")
            return
        for i in self.heap:
            print(f"({i.data}, {i.priority})", end=" ")
        print()

    def display_as_graph(self, filename):
```

```
        dot = graphviz.Digraph(format='png')

        for i, element in enumerate(self.heap):
            dot.node(f'node_{i}', f'{element.data},{element.priority}')

        for i, element in enumerate(self.heap):
            left_child_idx = 2 * i + 1
            right_child_idx = 2 * i + 2

            if left_child_idx < len(self.heap):
                dot.edge(f'node_{i}', f'node_{left_child_idx}')

            if right_child_idx < len(self.heap):
                dot.edge(f'node_{i}', f'node_{right_child_idx}')

        dot.render(filename, cleanup=True)

if __name__ == '__main__':
    pq = PriorityQueue(10)
    pq.display()
    pq.enqueue(Element(10, 1))
    pq.display()
    pq.enqueue(Element(14, 0))
    pq.display()
    pq.enqueue(Element(16, 4))
    pq.display()
    pq.enqueue(Element(12, 5))
    pq.display()
    pq.enqueue(Element(34, 20))
    pq.display()
    pq.enqueue(Element(100, 3))
```

```
pq.display()
pq.enqueue(Element(8, 20))
pq.enqueue(Element(85, 20))
pq.display_as_graph("pq_before_dequeue")
pq.dequeue()
pq.display_as_graph("pq")
```