Programming Assignment: Priority Queue

Priority queues are a workhorse for other algorithms. See the lecture notes for a description of an implementation using heaps.

We are going to use priority queues for implementing topological sort using in-degrees. First, we need to implement a data structure that combines vertex identifiers with in-degrees. With this data structure, we need to compare by in-degree meaning that we need comparison functions.

```
class Gl:
    INFTY = 2**32-1

class Vertex:
    def __init__(self, x, val = Gl.INFTY):
        self.id = x
        self.degree = val
    def __str__(self):
        return '({},{})'.format(self.id, self.degree)
    def __eq__(self, other):
        return self.degree == other.degree
    def __le__(self, other):
        return self.degree <= other.degree
    def __lt__(self, other):
        return self.degree < other.degree
    def __gt__(self, other):
        return self.degree > other.degree
    def __ne__(self, other):
        return self.degree != other.degree
    def __ge__(self, other):
        return self.degree >= other.degree
    def assign(self, val):
        self.degree = val
```

As you can see, we create vertices with a default value of INFTY, an entirely fictional name since Python integers have no maximum value.

For the priority queue, we use the standard array-based heap. We also implement methods for iterating through all elements as well as navigation. up, left, and right refer to calculating the index of the parent, the left child and the right child of an index. The len function is important so that we can write something like `while pq`.

```
class PQ:
    def __init__(self):
        self.array = []

    def up(index):
        return (index+1)//2-1

    def left(index):
        return 2*index + 1

    def right(index):
        return 2*index + 2
```

```
    def __str__(self):
        return 'Contents\n'+'\n'.join([str(x) for x in self.array])+'\n'

    def __len__(self):
        return len(self.array)

    def __iter__(self):
        self.index = 0
        return self

    def __next__(self):
        try:
            ret_val = self.array[self.index]
        except IndexError:
            raise StopIteration
        self.index += 1
        return ret_val

    def test_heap(self):
        for i in range(1, len(self.array)):
            if self.array[i] < self.array[PQ.up(i)]:
                return False
        return True
```

The insert operation is fairly simple. You insert at the end of the array and then check whether we need to swap with the parent.

```
def insert(self, value):
        n = len(self.array)
        self.array.append(value)
        while n>0:
            parent = PQ.up(n)
            if self.array[parent] > value:
                self.array[n], self.array[parent] = self.array[parent], \
self.array[n]
                n = parent
            else:
                return
```

The pop operation is more difficult. The first array element is the value that we want to return. We swap the last element into the first position and then we repair the array so that heap properties are maintained. This is tedious, because we need to check whether a current node has no, one, or two children.

```
def pop(self):
        ret_val = self.array[0]
        if len(self.array) == 1:
            del self.array[-1]
            return ret_val
        last = self.array[-1]
        del self.array[-1]
        self.array[0] = last
        n=0
        while n < len(self.array):
            left = PQ.left(n)
            right = PQ.right(n)
            if right < len(self.array):   # current node has two children
```

```
                if self.array[n] <= self.array[left] and self.array[n] <= \
self.array[right]:
                    return ret_val
                if self.array[left] >= self.array[right]:
                    m = right
                else:
                    m = left
                self.array[n], self.array[m] = self.array[m], self.array[n]
                n = m
            elif left < len(self.array): # current node has one child
                if self.array[n] > self.array[left]:
                    self.array[n], self.array[left] = self.array[left], \
self.array[n]
                return ret_val
            else:  # current node has no child
                return ret_val
        return ret_val
```

For the topological sort, we need to lower some values in the priority queue and then rearrange the heap so that the heap property is maintained. This would be most appropriately done by providing a glue structure between the graph data structure and the heap structure that allows us to find the position of a vertex in the heap. This structure is a dictionary that needs to be updated whenever we swap values in the heap. In lieu of this structure, here is a simple find value that given a vertex identifier, returns the index in the heap array.

```
    def find(self, vertex_number):
        for i in range(len(self.array)):
            if self.array[i].id == vertex_number:
                return i
```

Your task is the implementation of the lower(self, vertex_number, new_degree). You need to provide the test_heap method and the gen_ran(nr) function for testing.

```
def gen_ran(nr):
    pq = PQ()
    for i in range(nr):
        pq.insert(Vertex(i, random.randint(0,12)))
    return pq
```