CS340 Project 2 Priority Queue with index

A ***priority queue*** is a data structure for maintaining a set of elements, which we will call nodes. Each node has an ***ID*** and an associated value called a ***key***. A priority queue can either be a max-priority-queue, in which a call to extract( ) returns the node with the highest value key, or a min-priority queue in which a call to extract( ) returns the node with the smallest key.

A ***priority queue*** supports the following operations:

insert(x) inserts the node x into the priority queue.
peek() returns the node with the largest or smallest key, depending on the type of queue.
extract() removes and returns the element of S with the largest or smallest key, depending on the type of queue.
changeKey(i, k) changes the value of the key of the element located at index i in the heap array to the new value k, then swaps the key up the heap until it is in the correct place. (Notice that Java's PriorityQueue and most library priority queues do not contain this method.)

Using the heapsort implementation you have already written as the starting point, implement a node class and priority queue to contain items of the node class. The node class should have instance variables that define an ID and a key. It should also have a compare method that can be adjusted to make the queue either min or max priority. The priority queue implementation should extend the heap class by adding the additional methods listed above. You should also add a location table to the heap. The location table keeps track of the location of each node in the heap by ID.

The book is confusing in its implementation of changeKey(i, k). The problem is in the parameter *i*. This is the index of an item in the priority queue array. In the context in which changeKey is used, you may not know the index of the item in the priority queue array, but you know its ID (for example, you want to changeKey of node 7). You have no way of knowing where node 7 is located in the priority queue array. One solution to this problem is to do a linear search through the priority queue array. This is a terrible solution, as it increases the time complexity of changeKey from O(lg n) to O(n).

A clever way to implement the priority queue is to use a separate array to keep track of where each node ID is located in the queue. For example, imagine the priority queue contents look like this, where id is the node number:

Id=1, key = 6	Id = 3, key=9	Id = 2, key=11	Id = 0, key=10	Id=4, key =12

Our nodes are numbered starting at zero, and we know how many there are, so we can create an index array that simply tells the location of each node in the queue. Here is the table, which we will call locationTable:

3	0	2	1	4

For example, if we are looking for node ID 3, then locationTable[3] tells us that the node is located in slot 1 in the priority queue. We can then call changeKey(1,v) to changeKey on node 1 to the value v.

This location information must be updated every time values are changed in the priority queue. For example, in heapify(), part of the method is swapping the smallest node with node i. One way to write this code would be:

```
if (significant != i) {
      // exchange heap[i] with heap[significant]
      Node temp = heap[i];
      heap[i] = heap[significant];
      heap[significant] = temp;
     // swap the location table as well
     location[heap[significant].ID] = significant;
     location[heap[i].ID] = i;
      heapify(significant);
}
```

Here is the idea of the code. The code below is in Java, but your can do the assignment in C++, Java or C#.

```
public class Node implements Comparable<Node> {
    public int ID;
    public int key;

    public Node(int ID, int key) {
      // code
    }

    @Override
    public int compareTo (Node node2) {
      // code
    }
}

public class Heap {
    protected Node[] heap;
    protected int heapsize;
    protected int[] location;

    public Heap() {
       // create a heap with a default size, maybe 20
    }

    public Heap(int n) {
      // create a heap of size n
    }

    public Heap(Node[] a) {
      // create a heap from an array of nodes
    }

    public int parent(int i) {
      // code
    }

    public int left(int i) {
      // code
    }

    public int right(int i) {
      // code
    }

    public void heapify(int i) {
      // code
    }

    public void buildheap() {
      // code
    }
}

public class PriorityQueue extends Heap{
    public PriorityQueue(int maxsize) {
      // create a priority queue of size maxsize
    }

    public void insert(Node newNode) {
      // code
    }

    public Node peek() {
      // code
    }

    public Node extract() {
      // code
    }

    public void changeKey(int i, int k) {
      // code
    }
}

public class Main {
    public static void main(String[] args) {
        PriorityQueue pq = new PriorityQueue(7);
        pq.insert(new Node(0,3));
        pq.insert(new Node(1,14));
        pq.insert(new Node(2,7));
        pq.insert(new Node(3,9));
        pq.insert(new Node(4,99));
        pq.insert(new Node(5,2));
        pq.insert(new Node(6,46));

        for (int i = 0; i < 7; i++) {
            System.out.print(pq.extract().key + " ");
        }
    }
}
```

```
The above code should print out 99 46 14 9 7 3 2

For your use in troubleshooting:
The heap should contain 99 14 46 3 9 2 7
The location table should contain 3 1 6 4 0 5 2
```

Make sure your code works correctly for **any** input and size.

This project should be submitted on Moodle. The project may be done in pairs of 2. Late projects will be accepted according to rules given in the syllabus. The project can be completed in Java, C++, or C#. No report is due as part of this project. Grading will be according to the rubric below.

Node Class	0 pts No node class	5 pts Node class implemented with minor errors	10 pts Node class implemented correctly	
Heap	0 pts No heap	5 pts Heap has some errors	10pts Heap has no errors	
Insert()	0 pts No insert method	5 pts Method implemented with errors	10 pts Method implemented correctly	
Peek()	0 pts No peek method or incorrect implementation	5 pts Method implemented correctly		
Extract()	0 pts No extract method	5 pts Method implemented with major errors	10 pts Method implemented with minor errors	15 pts Method implemented correctly
ChangeKey()	0 pts No changekey method	5 pts Method implemented with major errors.	10 pts Method implemented with minor errors	15 pts Method implemented correctly
Output	0 pts No output	5pts Incorrect output	10 pts Correct output	