Priority Queues

A priority queue is an ADT to store a colection of elements supporting three main operations:

- insert(x): adds an element x to the colection;
- peek(): returns (without removing) the element with the most *priority*;
- remove(): returns and removes the element with most *priority*.

Priority queues are useful in a wide range of scenarios. Here are some exemples of where they are used:

- A priority queue on supermakets;
- A router with a certain priority traffic;
- Simulation of discreet events;

There are also a lot of algorithms which use priority queues as their building bloks:

- Dijkstra's algorithm: used to find the shortest path between two nodes;
- Prim's algorithm: used to know which closest node on a tree wasn't added;
- A* algorithm: to know what is the next node to go to which has the best heuristics to use.

Let's assume that we are working with comparable elements and that the one which has the most priority is the lowest value.

This way, the method *remove()* can also be understood as a *removeMin()* which removes the lowest value on the set.

Heaps

A *heap* is a tree which obeys the following restriction: The parent of each node has always the most priority compared to its child node, e.g in a minHeap, the parent node is always less then its children.

Using arrays

The easiest and compact way to implement a heap is using an array which implicitly represents a tree.

- The elements are displayed on their level order (from top to bottom, left to right);
- If the root is on position 1, then: – The child of a node at position x are at positions x*2 and x*2+1; – The parent of a node at position x is at position x/2 (integer division).

This way, the minimum element is simply the first element on the array and the min() method is O(1)!

removeMin()

When the root is deleted, it's necessary to go back to the heap's original structure. To accomplish that:

- Take the last element of the array and put it as the root of the heap;
- This element will go down (downHeap) switching places with smaller elements until the structure of the heap is assured;
- The worse case scenario would traverse the depth of the tree, which is O(log n).

Therefore, *removeMin()* is O(log n)!

Insert(x)

The process of inserting an element into the heap consists in:

- Add it at the end of the array (last occupied position);
- The element "goes up" (upHeap), switching places with the node until the heap structure is restored;
- The worse case scenario would traverse the depth of the tree, e.g O(log n).

Therefore, *insert(x)* is O(log n)!

HeapSort

You can sort an array using heaps: simply add all the elements of the array into a heap and then take them out one by one. Since the elements are removed by their priority, in the end they will be sorted.

This process means n insertions followed by n deletions. Since every operation is logarithmic, heapsort will be O(nlog n). This algorithm is known as HeapSort!

Implementation

You can check out the code for the methods min(), insert(x), removeMin() in the code present in the parent folder of this directory.

Unatural Comparator

What if you want to use a comparison different from the natural? It's possible to pass in a comparator and the Heap will use it!

We could create a new class called LengthComparator which implements the class Comparator and has a function compare() which compares two strings:

```
class LengthComparator implements Comparator<String >{
    public int compare (String a, String b) {
        return a.length() - b.length();
    }
}
```

Then, if you use

```
MinHeap<String> h = new MinHeap<>(100,new LengthComparator());
String [] v = {"aaaaa", "bbb", "cccc", "d", "ee"};
for(int i=0; i<v.length;i++)
h.insert (v[i]);
for(int i=0;i<v.length;i++)
    System.out.print(h.removeMin() + " ");
System.out.println ();
```

the result will be:

```
d ee bbb cccc aaaaa
```