Untitled

HTML Content

1. **Dynamic median.** Design a data type that supports *insert* in logarithmic time, *find-the-median* in constant time, and *remove-the-median* in logarithmic time. If the number of keys in the data type is even, find/remove the *lower median*. 1 1

Note: these interview questions are ungraded and purely for your own enrichment. To get a hint, submit a solution.

```
class MediaHeap {
private MaxPQ<Integer> left;
private MinPQ<Integer> right;
private int L;
private int R;

MediaHeap() {
left = new MaxPQ<Integer>();
right = new MinPQ<Integer>();
}

public double findMedian() {
int L = left.size();
int R = right.size();
if (L == R)
return ((double)left.max() + (double)right.min()) / 2;
else if (L > R)
return left.max();
else
return right.min();
}

public void insert(int key) {
double median = findMedian();
int L = left.size();
int R = right.size();
if (key <= median) {
left.insert(key);
if (L - R > 1)
right.insert(left.delMax());
}
else {
right.insert(key);
if (R - L > 1)
left.insert(right.delMin());
}
}
```

```
public void removeMedian() {
int L = left.size();
int R = right.size();
if (L > R) {
left.delMax();
}
else {
right.delMin();
}
}

}
```

Correct

Hint: maintain *two* binary heaps, one that is max-oriented and one that is min-oriented.

2. Randomized priority queue. Describe how to add the methods sample() and delRandom() to our binary heap implementation. The two methods return a key that is chosen uniformly at random among the remaining keys, with the latter method also removing that key. The sample() method should take constant time; the delRandom() method should take logarithmic time. Do not worry about resizing the underlying array. 1 1

Generate random number from 0 - N, sample() just return that number
When delete random, exchange with last, delete last, then compare with parent and children to decide whether swim or sink

Correct

Hint: use sink() and swim().

3. Taxicab numbers. A *taxicab* number is an integer that can be expressed as the sum of two cubes of positive integers in two different ways:

$a^3 + b^3 = c^3 + d^3$. For example, 1729 is the smallest taxicab number: $9^3 + 10^3 = 1^3 + 12^3$. Design an algorithm to find all taxicab numbers with a, b, c, and d less than n. 1 / 1pc

- Version 1: Use time proportional to $n^2 \log n$ and space proportional to n^2.
- Version 2: Use time proportional to $n^2 \log n$ and space proportional to n.

```
class Taxicab implements Comparable<Taxicab>{
int n1;
int n2;
int cube;

Taxicab(int n1, int n2) {
this.n1 = n1;
this.n2 = n2;
this.cube = n1 * n1 * n1 + n2 * n2 * n2;
}

@Override
public int compareTo(Taxicab that) {
if (that.cube > this.cube) return -1;
if (that.cube < this.cube) return 1;
return 0;
}

@Override
public boolean equals(Object o) {
if (o instanceof Taxicab) {
if (((Taxicab)o).compareTo(this) == 0)
return true;
}
return false;
}

@Override
public String toString() {
return "number: " + cube + " (" + n1 + ", " + n2 + ")";
}
}

public void findTaxinumber(int N) {
MinPQ<Taxicab> candidates = new MinPQ<Taxicab>();

for (int i = 1; i <= N; i++) {
for (int j = i + 1; j <= N; j++) {
Taxicab t = new Taxicab(i, j);
if (candidates.size() < N) {
candidates.insert(t);
}
```

```
else {
Queue<Taxicab> temp = new Queue<Taxicab>();
Taxicab min = candidates.delMin();
while (candidates.min().equals(min)) {
temp.enqueue(candidates.delMin());
}
if (!t.equals(min)) {
candidates.insert(t);
}
else {
temp.enqueue(t);
}
if (!temp.isEmpty()) {
for (Taxicab taxi: temp) {
System.out.println(taxi);
}
System.out.println(min);
}
}
}
}
}

public static void main(String[] args) {
PriorityQueue p = new PriorityQueue();
p.findTaxinumber(12);
}
```

Correct

Hints:

- Version 1: Form the sums $a^3 + b^3$ and sort.
- Version 2: Use a min-oriented priority queue with n items.