Priority Queues

No nulls are allowed, but duplicates are ok.

Elements: value, priority
↳ usually integer.

Convention: smaller the priority, more important it is.

Operations: std operations: clear(), isEmpty(), size(), ...

1. add(x) ≡ offer(x): (Traditional name: insert(x))
   adding a new element x to the priority queue.
2. remove() or poll(): (Traditional name: DeleteMin() or ExtractMin())
   remove and return element with smallest priority (i.e. most important element)
   Tie breaks - arbitrary.

PriorityQueue abstract data type does not support
remove(x) or contains(x) operations.
or iterator()

↳ But Java's implementation supports these operations, badly — Don't use these ops.

3. peek(): return (without removing) element with smallest priority (i.e. most important)

Common usage:

```
PriorityQueue<Integer> q = new PriorityQueue<>();
```

↑ int = value = priority

↑ using natural ordering — compareTo method.

```
PriorityQueue<Vertex> q =
        new PriorityQueue<>(42, comp);
```

Initial size of q = 42.

comp = object that extends (or implements) Comparator<Vertex> interface.

Comparator interface: one method

public int compare(Vertex one, Vertex two)

↳ return negative no if one < two

0 if one = two

positive no if one > two.

Comparator — user-defined ordering.

Example:

```
public class Vertex implements Comparator<Vertex> {...
    public int compare(Vertex a, Vertex b) {
        if (a.dist < b.dist) return -1;
        else if (a.dist == b.dist) return 0;
        else return 1;
    }
}
PriorityQueue<Vertex> q = new PriorityQueue<>(g.size(), new Vertex());
```

```
: q.add(x) ;
:
while (! q.isEmpty()) {
    x = q.remove(); // or x = q.poll();
    :
    q.add(y);
    :
}
```

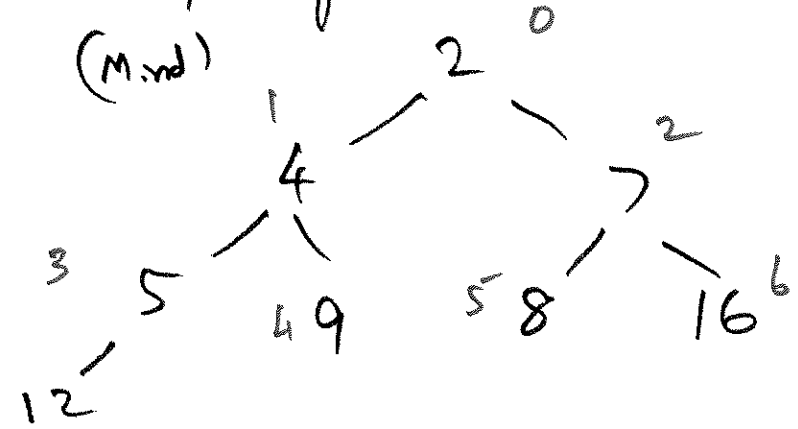
Implementation of Priority Queues: Binary Heaps.

Binary Heap: a binary tree with one element per node, satisfying the following constraints:

1. Ordering constraint: Priority of node ≤ Priority of its children.
2. structure constraint: tree = complete binary tree, close to a full binary tree as possible. Leaves have to packed to the left.

Array implementation of Binary Heaps:

(Mind)



(Reality)

2	4	7	5	9	8	16	12
0	1	2	3	4	5	6	7

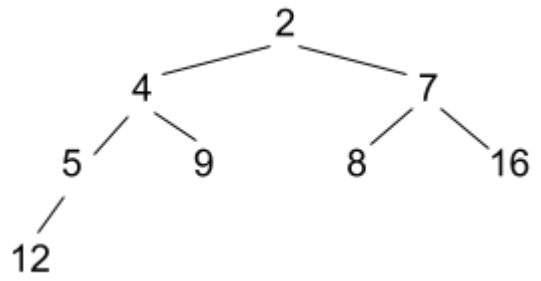
parent(i) = (i-1)/2

children(i) = {2i+1, 2i+2}

Priority Queues: Each element has a priority. Elements are removed in the order of their priorities. Traditionally, smaller the value of priority, higher its priority. Duplicates are allowed, but not null values.

Operation	Traditional name	Java method
Insert a new element	Insert (x)	add (x), offer (x)
Remove element with maximum priority	DeleteMin ( ), ExtractMin ( )	remove ( ), poll ( )
Element with maximum priority	Min ( )	peek ( )

Java's PriorityQueue is iterable, but the iteration order is not in order of priority. The operations contains(x) and remove(x) are inefficient, $O(n)$. If they are needed, use a TreeMap/TreeSet instead.
Implementation of priority queues: Binary heaps (usually stored in arrays).



Array implementation:

2	4	7	5	9	8	16	12
0	1	2	3	4	5	6	7

Root is stored at index 0.
For a node at index i:
Parent's index: $(i - 1) / 2$.
Index of Children: $\{2i + 1,\ 2i + 2\}$.

Heaps are binary trees in which each node stores one element, satisfying the following properties:

1. Order property: priority of node $\leq$ priority of its children, i.e., it has higher priority than its children,
2. Structure property: complete binary trees (all but last levels are full; last level is packed from left).

Operations will restore structure property first, and then ensure order property, using percolate Up/Down.

Let $n = size$. Heap occupies $pq[0..n-1]$. Height of tree is $ceil(\log(n))$. RT of each operation: $O(\log(n))$.

parent (i): return $(i-1)/2$

firstChild (i): return $2 * i + 1$

add (x):
  if $size = pq.length$ then // can resize pq here
    throw Exception "Priority queue is full"
  $pq[size] \leftarrow x$ // rewrite using move ()
  percolateUp ($size$)
  $size$ ++

percolateUp (i):
  $x \leftarrow pq[i]$
  while $i > 0$ *and* $x < pq[parent(i)]$ do
    $pq[i] \leftarrow pq[parent(i)]$
    $i \leftarrow parent(i)$
  $pq[i] \leftarrow x$

remove ( ):
  $min \leftarrow pq[0]$
  $pq[0] \leftarrow pq[--size]$
  percolateDown (0)
  return min

percolateDown (i):
  $x \leftarrow pq[i]$
  $c \leftarrow firstChild(i)$
  while $c \leq size - 1$ do
    if $c + 1 < size$ *and* $pq[c] > pq[c+1]$ then
      $c \leftarrow c + 1$ // right child has higher priority
    if $x \leq pq[c]$ then break
    $pq[i] \leftarrow pq[c]$
    $i \leftarrow c$
    $c \leftarrow firstChild(i)$
  $pq[i] \leftarrow x$

peek ( ): return $pq[0]$
move (i, x): $pq[i] \leftarrow x$ //override in indexed heap

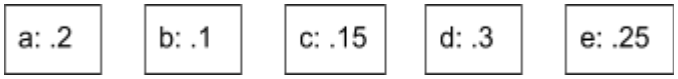
Applications of priority queues:
Huffman coding
Prim's algorithm for MST
Dijkstra's algorithm for shortest paths
Select algorithm (k largest elements of a large array or stream)
Process scheduling, interrupt handling in operating systems
Heapsort
Heuristics for memory management, bin packing
Discrete event simulations, computer games: (randomly generated) events are processed in temporal order
A* search in AI
Reduce round-off errors in floating point computations: best way to find sum of $A[i],\ i = 1..n$? Is $1 + \varepsilon = 1$?
Merge sort using k-way merge
Find perfect powers (numbers of the form $a^b,\ b > 1$) in increasing order, up to some n (say, 10^{18}).
Enumerate numbers whose prime factors are only from a given set (e.g., $\{3, 5, 7\}$), in sorted order.
Pairing buy/sell orders in the stock market by market makers

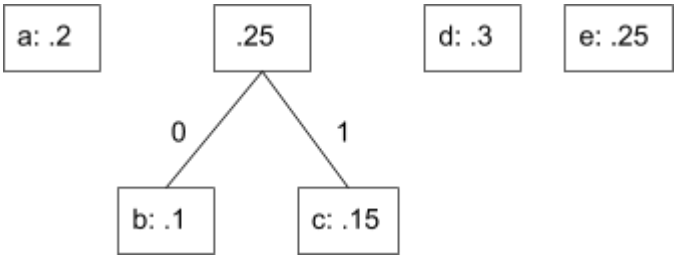
Huffman coding (application of priority queues):
Input: Alphabet Σ, and a frequency function $f : \Sigma \rightarrow \mathcal{R}^+$. Example: $\Sigma = \{A, C, G, T\}, f = \{.5, .25, .1, .15\}$.
In the given input file with n characters, $c \in \Sigma$ occurs $f(c) \cdot n$ times.
Output: A binary code for Σ that minimizes the total length of the file. It is required that no character's code is a proper prefix of another character's code. Such codes are called *prefix* codes, and no boundary markers are needed when a file is encoded with prefix codes.

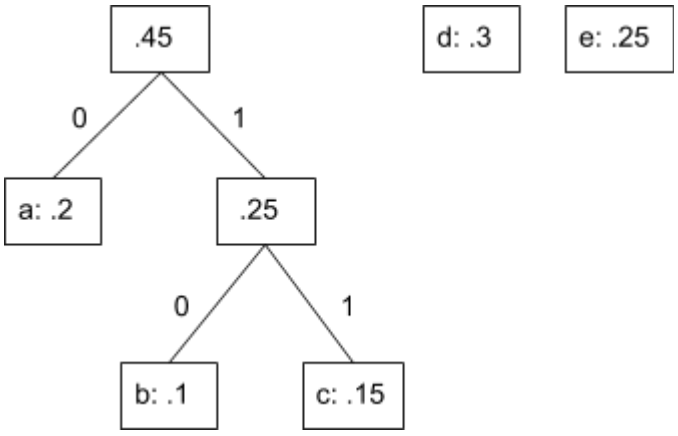
	$A: 0.5$	$C: 0.25$	$G: 0.1$	$T: 0.15$	Weighted average of bits per character
Fixed length code	00	01	10	11	$2 * .5 + 2 * .25 + 2 * .1 + 2 * .15 = 2$
Variable length code	0	10	110	111	$1 * .5 + 2 * .25 + 3 * .1 + 3 * .15 = 1.75$

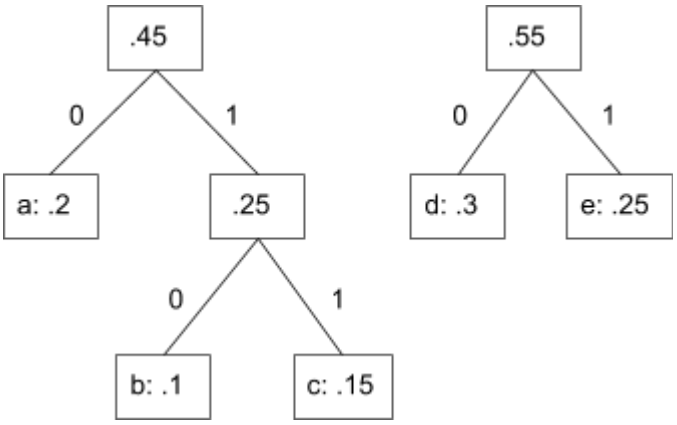
Huffman's algorithm to compute an optimal prefix code:
Create a priority queue q with the characters of Σ, where $c \in \Sigma$ has priority $f(c)$.
while q has more than one node do
    $x \leftarrow q.remove()$
    $y \leftarrow q.remove()$
    Create a new node z with frequency $f(z) = f(x) + f(y)$.
    Attach nodes x and y as the 2 children of z, along edges labeled 0 and 1.
    $q.add(z)$
The single node in q contains a tree that is an optimal coding tree for the given problem.
Code for $c \in \Sigma$ is obtained by concatenating the labels along the edges from the root of the tree to the leaf node corresponding to c.

Huffman coding example: $\Sigma = \{a, b, c, d, e\}$, $f = \{.2,\ .1,\ .15,\ .3,\ .25\}$

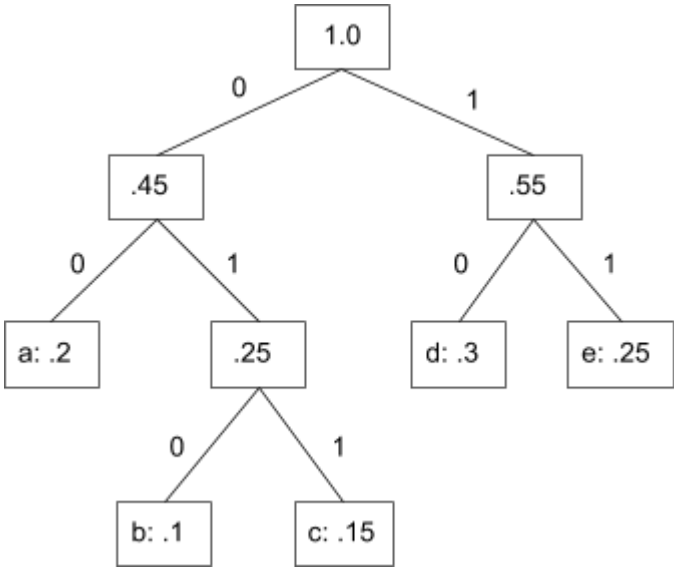








Final encoding tree:



Codes:

a	00
b	010
c	011
d	10
e	11

Weighted average number of bits per character:
$2 * .2 + 3 * .1 + 3 * .15 + 2 * .3 + 2 * .25 = 2.25.$