Ranking

In a set or a dictionary:
Rank of key $k$ is $i$ iff $k$ is the $i$th smallest key.

Example: if $s$ stores these keys 42, 55, 61, then:

- $s.rank(42)$ returns 1
- $s.rank(55)$ returns 2
- $s.rank(61)$ returns 3
- $s.rank(k)$ throws exception if $k$ not found
Ranks from Sizes: Introduction

Recall: each node stores size of subtree.

```java
class Node<K> {
    public K key;
    public Node<K> left, right, parent;
    public int num;
}
static int size(Node<?> u) {
    return u == null ? 0 : u.num;
}
```

This will be useful for finding ranks.
Rank of 50 is...
Rank of 50 is...3 because there are 2 keys smaller than 50: those in the left subtree.

⇒ Rank is \( \text{size}(v.\text{left}) + 1 \)

Except...
Rank of 50 is... 3 because there are 2 keys smaller than 50: those in the left subtree.

⇒ Rank is size(v.left) + 1

Except... this may be just part of a larger tree.
Computing Ranks from Sizes, Part 2/3

If v has a parent:

\[
\begin{align*}
\text{Rank of 50 is } 3. \\
\text{Except...}
\end{align*}
\]
If \( v \) has a parent:

- Rank of 50 is 3.
- Rank of 50 is 7+1+3.

Except... \( u \) also has a parent, etc., ad infinitum. We need to loop or recurse over this.
Recursively:

```java
private static int relativeRank(K k, Node<K> u) throws NotFound {
    if (u == null) { throw new NotFound(); }
    int c = k.compareTo(u.key);
    if (c < 0) { return relativeRank(k, u.left); }
    if (c > 0) { return size(u.left) + 1 +
                  relativeRank(k, u.right); }
    return size(u.left) + 1;
}
```

```java
public int rank(K k) throws NotFound {
    return relativeRank(k, root);
}
```
Iteratively:

Keep a count on the number of smaller keys seen so far.

```java
int numSmaller = 0;
Node<K> v = root;
while (v != null) {
    int c = k.compareTo(v.key);
    if (c == 0) {
        return numSmaller + size(v.left) + 1;
    } else if (c > 0) {
        numSmaller += size(v.left) + 1;
        v = v.right;
    } else {
        // no change to numSmaller
        v = v.left;
    }
}
throw new NotFound();
```
Either way:

Computing rank takes $\Theta(\text{height})$ time worst case, i.e., $\Theta(\log(n))$. 
Next data structure.

(Just to confuse you: It’s a binary tree again!)
Priority Queue (Simplified Story)

A priority queue stores a collection of priorities.

- `insert(p)`: insert a priority `p`
- `maxKey()`: return the maximum priority
- `extractMax()`: remove and return the maximum priority

It's like:

- a hospital’s emergency room
- an OS’s ordering of things to do
- your ordering of things to study

(The real story: a collection of priority-job pairs.)
A heap is one way to store a priority queue. A heap is:

- a binary tree
- “nearly complete”: every level $i$ has $2^i$ nodes, except the bottom level; the bottom nodes flush to the left
- at each node: its priority $\geq$ both children’s priorities
Heap for Priority Queue

maxKey: $\Theta(1)$ time.
This is a win over using binary search trees.

insert, extractMax: We will see how, $\Theta(\log(n))$ time.
Heap insert: Example

Insert priority 15. At the bottom level, leftmost free space.

√ The tree is still “nearly-complete”.

! Order of priorities bad. Fix: swap with parent.
Aprueba

El árbol sigue siendo “nearly-complete”.

! Orden de prioridades malo. Corrige: intercambia con el padre.
Heap insert: Example

The tree is still “nearly-complete”.

Order of priorities good.
Heap insert: Summary

1. create new node at bottom level, leftmost free place (keep the tree “nearly-complete”)
2. put priority in new node
3. $v := $ that new node
4. while $v$ has parent with smaller priority:
   swap them
   $v := v.parent$

Worst case time $\Theta(height)$.

Later we will see why $height = \lfloor \log_2(n) \rfloor + 1$. Therefore worse case time $\Theta(\log(n))$. 
Heap extractMax: Example

Someone has to replace the blank!
Heap extractMax: Example

Replace by the bottom level, rightmost item.

✓ The tree is still “nearly-complete”.

! Order of priorities bad. Fix: swap with the larger child.
(Why not the smaller child?)
Replace by the bottom level, rightmost item.

✓ The tree is still “nearly-complete”.

! Order of priorities bad. Fix: swap with the larger child.
   (Why not the smaller child?)
Heap extractMax: Example

Replace by the bottom level, rightmost item.

✓ The tree is still “nearly-complete”.
✓ Order of priorities good.
Heap extractMax: Summary

1. Replace root by bottom level, rightmost item (keep the tree “nearly-complete”.)
2. $\nu := \text{root}$
3. while $\nu$ has larger child:
   swap with the largest child
   $\nu := \text{that child node}$

Worst case $\Theta(\text{height})$ time.

Next we will see why $\text{height} = \lfloor \log_2(n) \rfloor + 1$. Therefore worse case time $\Theta(\log(n))$. 
Heap: Height

Let $n$ be the number of nodes, $h$ be the height.

$$2^{h-1} - 1 \leq n \leq 2^h - 1$$

$$2^{h-1} \leq n < 2^h$$

$$h - 1 \leq \log_2(n) < h$$

$$h \leq \log_2(n) + 1 < h + 1$$

$$h = \lceil \log_2(n) \rceil + 1$$
Heap in Array/Vector

![Heap Diagram]

```
  16
  /  
14   10
 /  \
8    9
/  \
4   7
\  \
2   1
```

```
<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>14</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>9</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Heap in Array/Vector

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>16</th>
<th>14</th>
<th>10</th>
<th>8</th>
<th>7</th>
<th>9</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Convenience:

- Where to insert/remove: simply at the end.
- Saves space. (No pointers to store.)

Formulas for:

- left child of index $i$: index $2 \times i$
- right child of index $i$: index $2 \times i + 1$
- parent of index $i$: index $\lfloor i/2 \rfloor$
Increase A Priority

HeapNode heap[];

class HeapNode {
  int priority;
  Job job;
}

class Job {
  ... job data ...
  int heapPos; // index in the heap’s array
}

Init heapPos when inserting a new job.
Update when swapping two jobs during insert, extractMax.
(Can you see how?)

Then we can support: increaseKey(job, newPriority)
Increase A Priority

increaseKey(Job j, int newPriority):

1. \( v := j.heapPos \)
2. if \( newPriority < heap[v].priority \): error
3. \( heap[v].priority := newPriority \)
4. while \( v \) has parent with larger priority:
   swap them
   \( v := \lfloor v/2 \rfloor \)
Max vs Min

I have been storing larger priorities near the top to support maxKey, extractMax, increaseKey.
⇒ These are max priority queues and max-heaps.

But you could store smaller priorities near the top to support minKey, extractMin, decreaseKey.
⇒ These are min priority queues and min-heaps.

Example of min priority queue: A todo-list with start times.

Another application: coming soon!
Min-Heap Example

```
     11
    /  \
   17 /   \
  /     \
23  13
 /     /
29  47 43
 /     /
27  31 27
```

```
<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>17</th>
<th>13</th>
<th>23</th>
<th>31</th>
<th>43</th>
<th>27</th>
<th>29</th>
<th>47</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
```