Ranking

Rank of key $k$ in set $s$: how many keys are smaller than $k$.

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

- $s$.rank(42) returns 0
- $s$.rank(55) returns 1
- $s$.rank(61) returns 2
- $s$.rank($k$) throws exception if $k$ not found

Similarly for dictionaries.

How to extend AVL trees for this? All operations $\Theta(\log(n))$ time.

(Not going to make it: store rank at each node. \texttt{insert} has to update rank fields of the whole tree in the worst case.)
Ranking from Sizes: Introduction

Going to make it: at every tree node, store size of subtree at that node.

```java
class Node<K> {
    public K key;
    public Node<K> left, right, parent;
    public int height;
    public int size;
}
```

$v$.size remembers how many nodes are in the subtree at $v$, including $v$ itself.
$v$.size remembers how many nodes are in the subtree at $v$, including $v$ itself.
Rank of 50 is...
Computing Ranks from Sizes, Part 1/3

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

⇒ Rank is `v.left == null ? 0 : v.left.size`

Except...
Computing Ranks from Sizes, Part 1/3

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

⇒ Rank is `v.left == null ? 0 : v.left.size`

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

If \( v \) has a parent:

![Tree Diagram]

- Rank of 50 is 2.
- Except...
Computing Ranks from Sizes, Part 2/3

If v has a parent:

Rank of 50 is 2.

Except... u also has a parent, etc., ad infinitum. We need to loop or recurse over this.
Computing Ranks from Sizes, Part 3/3

Recursively:

1. Compute “relative” rank, i.e., rank in an arbitrary subtree.

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

private static int size(Node<K> v) {
    return v == null ? 0 : v.size;
}
```
Recursively:

2. Then “absolute” rank is just “relative” rank at root.

```java
public int rank(K k) throws NotFound {
    return relative_rank(k, root);
}
```
Computing Ranks from Sizes, Part 3/3

Iteratively:

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

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

Computing rank takes $\Theta(height)$ time worst case, i.e., $\Theta(\log(n))$. 
Updating Sizes

Assume that $v$’s child nodes already have their size fields set correctly, then $v$’s size field can be set correctly in $\Theta(1)$ time:

$$v.\text{size} = 1 + \text{size}(v.\text{left}) + \text{size}(v.\text{right});$$

When a node is inserted or removed (and before rotation), only $\Theta(\log(n))$ nodes need to change size (path from that node to root).

When rotating once, only 4 nodes need to change size ($x, y, z$, and one parent).

$\Rightarrow$ Old operations still take $\Theta(\log(n))$ time worse case.

Success!
Augmented Data Structures

You have just seen an example of “augmented data structures”.

It means: add some fields to a known data structure to support new lookups, without ruining the old big-Oh or big-Theta worst case times.

In our example: add a \texttt{size} field to support \texttt{rank} lookups, without ruining the old $\Theta(\log(n))$ worst case times.

Usually it requires some cleverness!
Next data structure.

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

A priority queue stores a collection of priority-value pairs.

- `insertItem(p, v)`: insert the pair of priority `p`, value `v`
- `minElement()`: return a value of minimum priority
- `minKey()`: return the minimum priority
- `removeMin()`: remove the priority-value pair of minimum priority
  (some versions also return that priority and/or value)

It's like:

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

A heap is one way to store a priority queue. A heap is:

- a binary tree
- “complete”: every level \( i \) has \( 2^i \) nodes, except the bottom level; the bottom nodes flush to the left
- at each node: its priority \( \leq \) both children’s priorities

(only priorities shown)
A Note on Terminology

Unfortunately! “Complete binary tree” has one meaning in the textbook, but another meaning in other books.

I go with the textbook. But beware when you read other books and articles.
Heap for Priority Queue

$\text{minElement, minKey: } \Theta(1) \text{ time.}$
This is a win over using binary search trees.

$\text{insertItem, removeMin: We will see how, } \Theta(\log(n)) \text{ time.}$
Heap insert: Example

Insert priority 12.

√ The tree is still complete.

! Order of priorities bad. Fix: swap with parent.
Heap insert: Example

The tree is still complete.

Order of priorities bad. Fix: swap with parent.
Heap insert: Example

The tree is still complete.
Order of priorities good.
Heap insert: Summary

1. create new node at bottom level, leftmost unoccupied place (keep the tree complete)
2. put priority and value in new node
3. \( v := \text{new node} \)
4. while \( v \) has parent with bigger priority:
   swap them
   \( v := v\.parent \)

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

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

Someone has to replace the blank!
Heap removeMin: Example

Replace by the bottom level, rightmost item.

✓ The tree is still complete.

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

Replace by the bottom level, rightmost item.

✓ The tree is still complete.

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

Replace by the bottom level, rightmost item.

√ The tree is still complete.
√ Order of priorities good.
Heap removeMin: Summary

1. Replace root by bottom level, rightmost item (keep the tree complete)
2. $v := \text{root}$
3. while $v$ has child with smaller priority:
   - swap with the child with the smallest priority
   - $v := \text{that child node}$

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.

\[ 2^{h-1} - 1 \leq n \leq 2^h - 1 \]

\[ (2^{h-1} - 1) + 1 \leq n \leq 2^h - 1 \]

\[ 2^{h-1} \leq n < 2^h \]

\[ h - 1 \leq \log_2(n) < h \]

\[ h \leq \log_2(n) + 1 < h + 1 \]

\[ h = \lfloor \log_2(n) \rfloor + 1 \]
Heap in Array/Vector
Heap in Array/Vector

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>17</td>
<td>13</td>
<td>23</td>
<td>31</td>
<td>43</td>
<td>27</td>
<td>29</td>
<td>47</td>
<td>37</td>
<td></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 $2 \cdot i$
- right child of index $i$: index $2 \cdot i + 1$
- parent of index $i$: index $\lfloor i/2 \rfloor$
Min vs Max

I have been storing smaller priorities near the top to support minElement, minKey, removeMin.

But you could store bigger priorities near the top to support maxElement, maxKey, removeMax.

In fact, do that for the next few slides on in-place heap-sort.
In-Place Heap-Sort

To sort an array in-place:

1. The heap invades the array from the left. $\Theta(n \cdot \log(n))$ time.
   Each time, the heap takes one more cell and does an insert.

   \[
   76, 21, 38, 15, 40, 32 \quad \rightarrow \quad 76, 21, 38, 15, 40, 32 \\
   \rightarrow \quad 76, 40, 38, 15, 21, 32
   \]

2. The heap retreats from the right. $\Theta(n \cdot \log(n))$ time.
   Each time, the heap does a removeMax and releases one more cell. The removed max goes to the released cell.

   \[
   76, 40, 38, 15, 21, 32 \quad \rightarrow \quad 32, 40, 38, 15, 21, 76 \\
   \rightarrow \quad 40, 32, 38, 15, 21, 76
   \]

This is in-place: uses little temporary space.