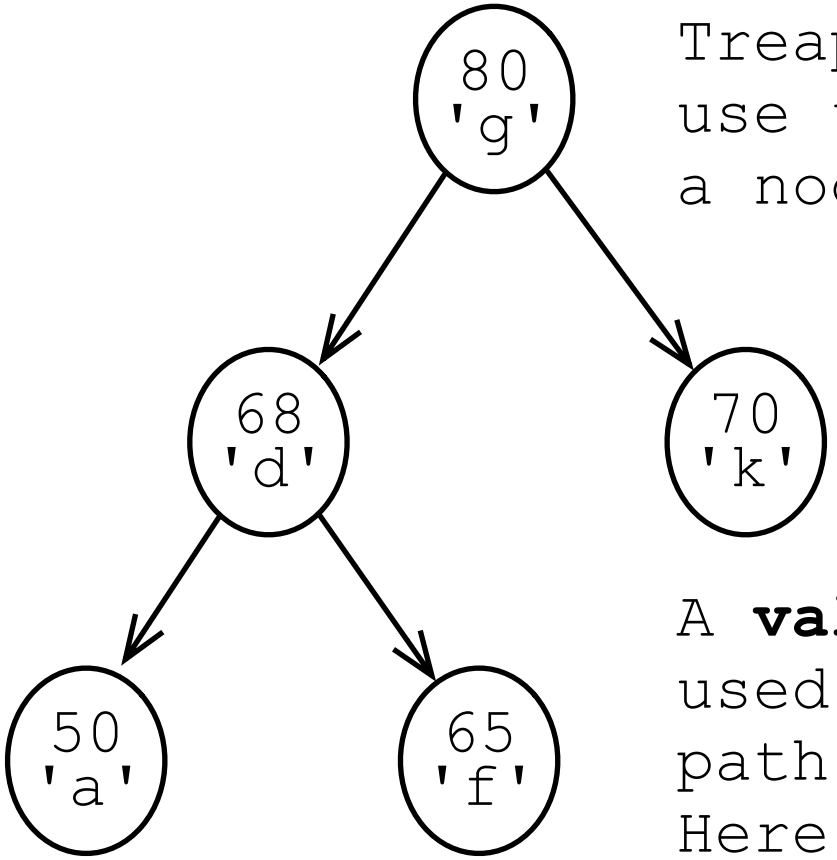
Treaps

A nifty binary search tree
with (some) heap semantics.

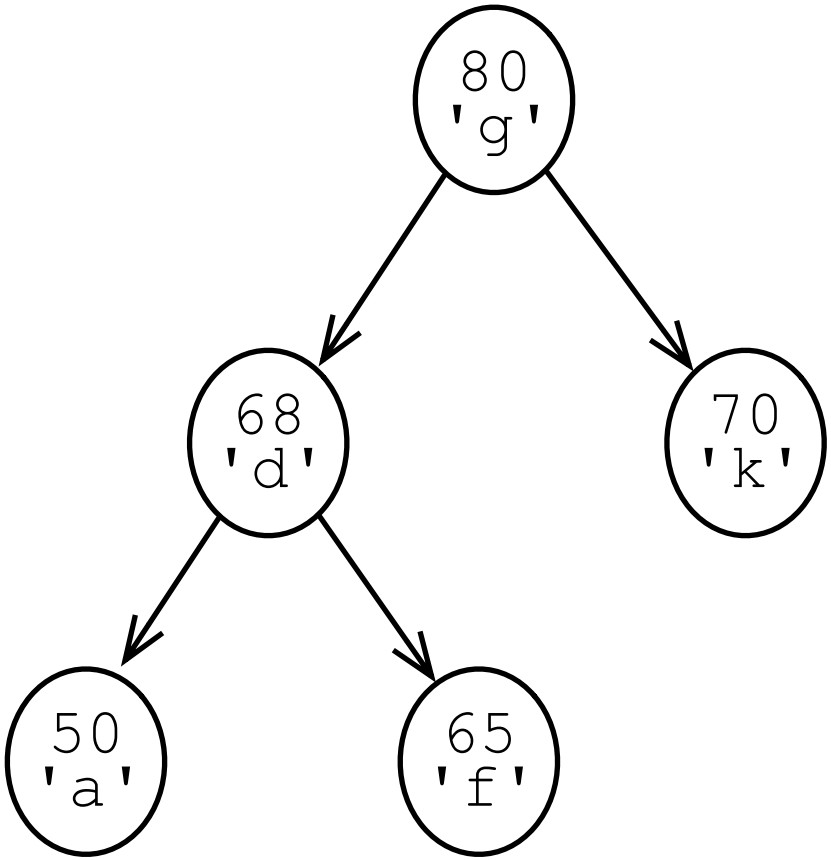
*Note: they should have
called it a beap.*



Treaps are binary trees. They use two data to determine where a node should be.

A **priority** is a randomly chosen number that is used to keep the tree balanced.

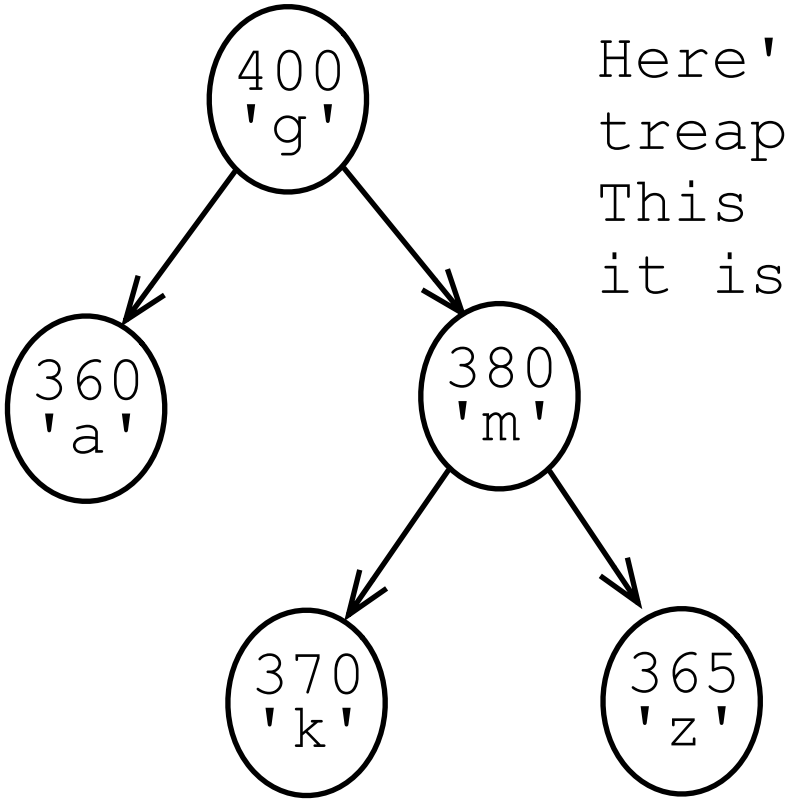
A **value**, also called a **key**, is used to determine which child path the node should be placed. Here the keys are characters.



The **priority** is why we say this has *heap* semantics. If we are using a *max heap*, the large priorities are on the top, and all child nodes must have smaller priorities.

The priority is chosen *randomly*. This is important.

Note that a proper heap is usually represented as a full tree with the leaf layer filling from the left. A *treap* does not have to obey this constraint. We can have a somewhat unbalanced treap, with some leafs all the way to the right without peers to the left. That's OK.

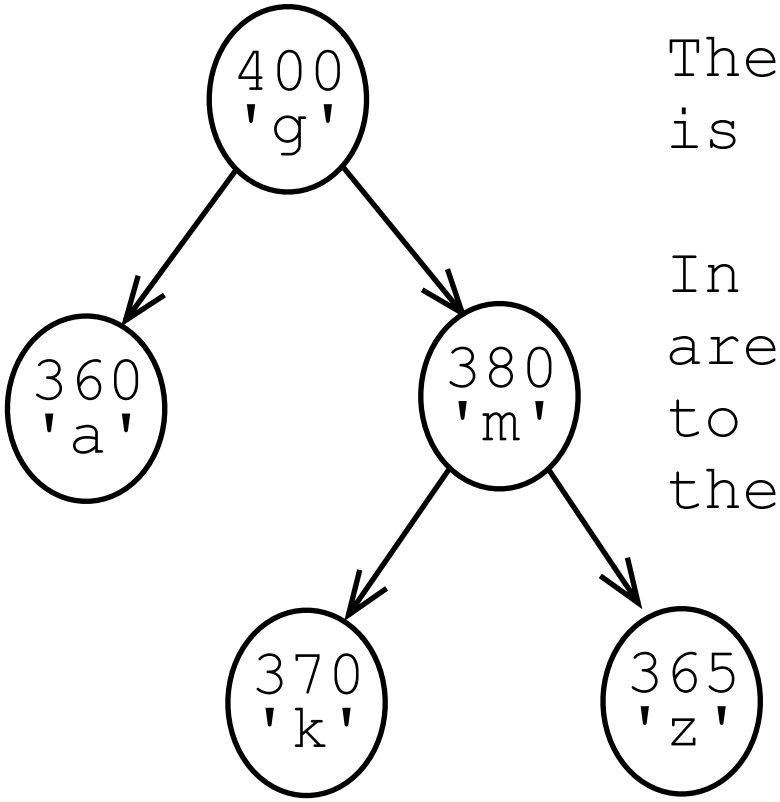


Here's what I mean by having a treap that isn't full to the left. This would be an invalid *heap* but it is a perfectly correct *treap*.

max heap = top larger

min heap = top smaller

As a reminder, the *max heap* property is that a parent's priority is larger than its children. That's it. It doesn't say anything about the relative values of (say) children or uncles.



The binary search tree property is in effect using the *keys*.

In all of these diagrams the keys are represented with characters, to easily tell them apart from the priority.

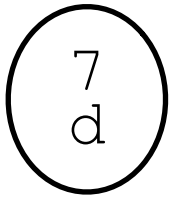
Keys are ordered in the same way a standard binary search tree are: lower values are found in the left child, other values are in the right child.

Treap Example

Lets say we want to insert some letters into a treap. When we do this we have to assign each one a random priority. In a real-world implementation we would probably choose numbers from the whole range of integers, but for this example I'll use single digit priorities.

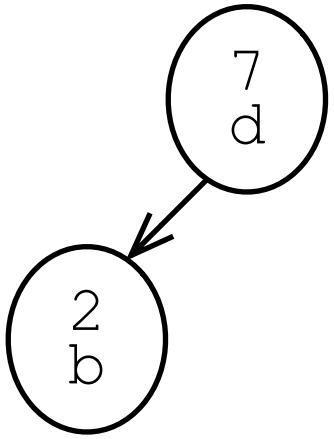
We'll insert from top to bottom with these randomly chosen priorities.

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



Inserting into an empty treap is easy. Just make an initial node.

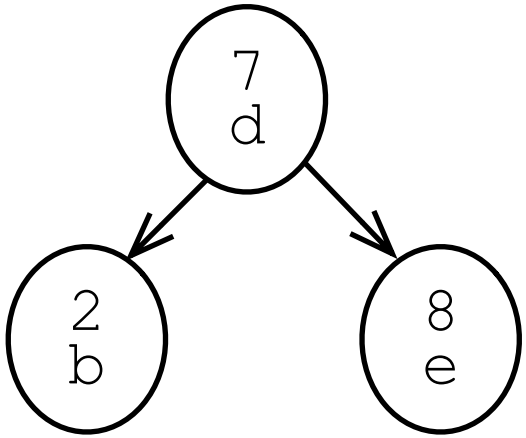
d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



Insert **b** with a priority of **2**. b is lexically less than d, so we know to make a left child. That's always the first step. We only fix the priority situation later, using rotations.

In this case the priorities are already in the correct order, so no action is needed.

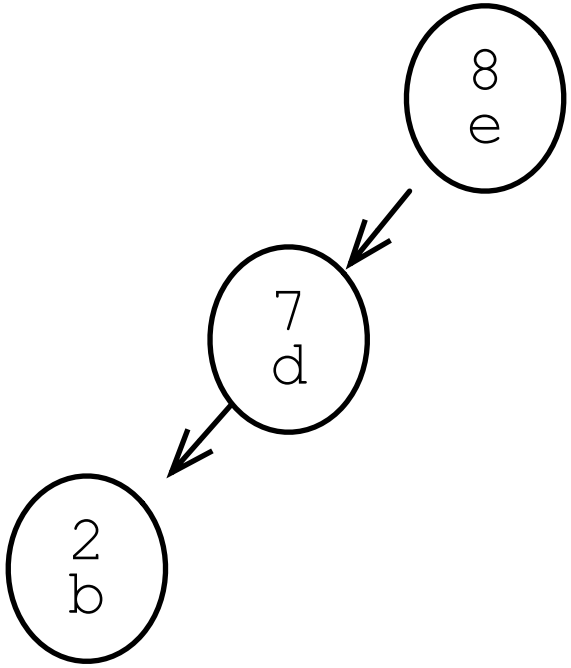
d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



After inserting the 'e' node in the correct location, we see that the *max heap* property is broken.

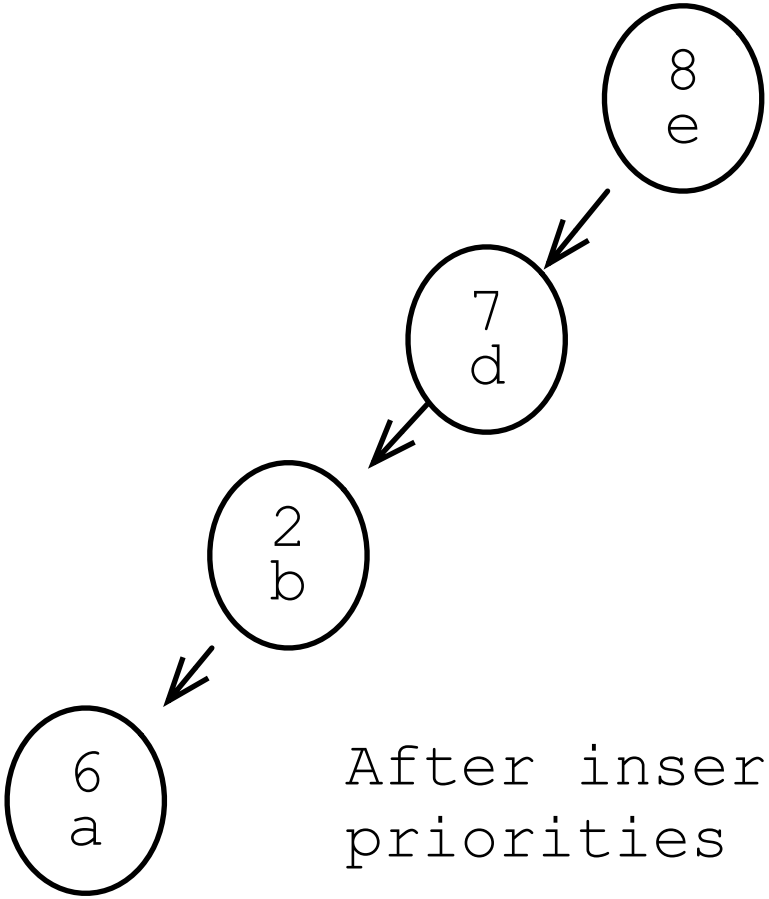
We fix this with the cunning use of *rotations*. Keep the keys sorted in the correct order, but arrange the nodes so higher priorities are on top.

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



This is what it looks like after rotating nodes to satisfy both the *max heap* and binary search tree invariants.

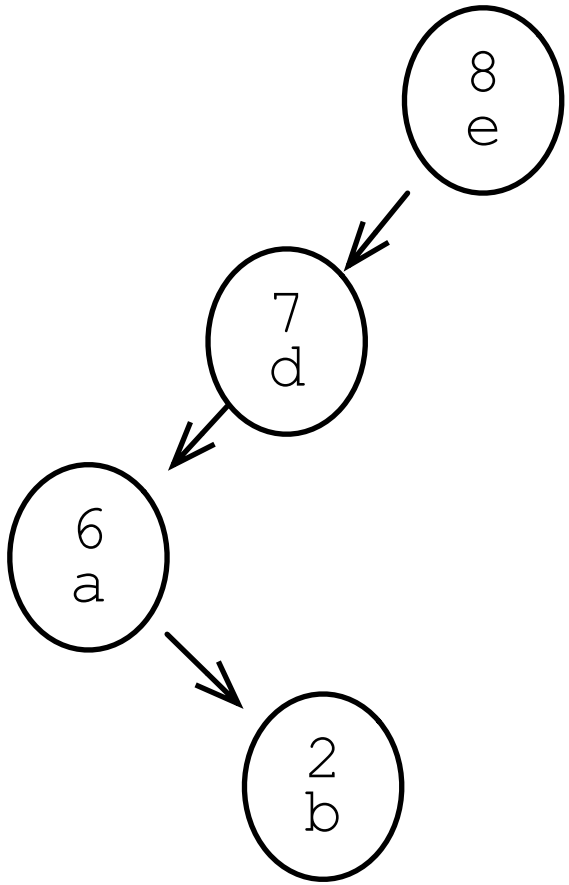
d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



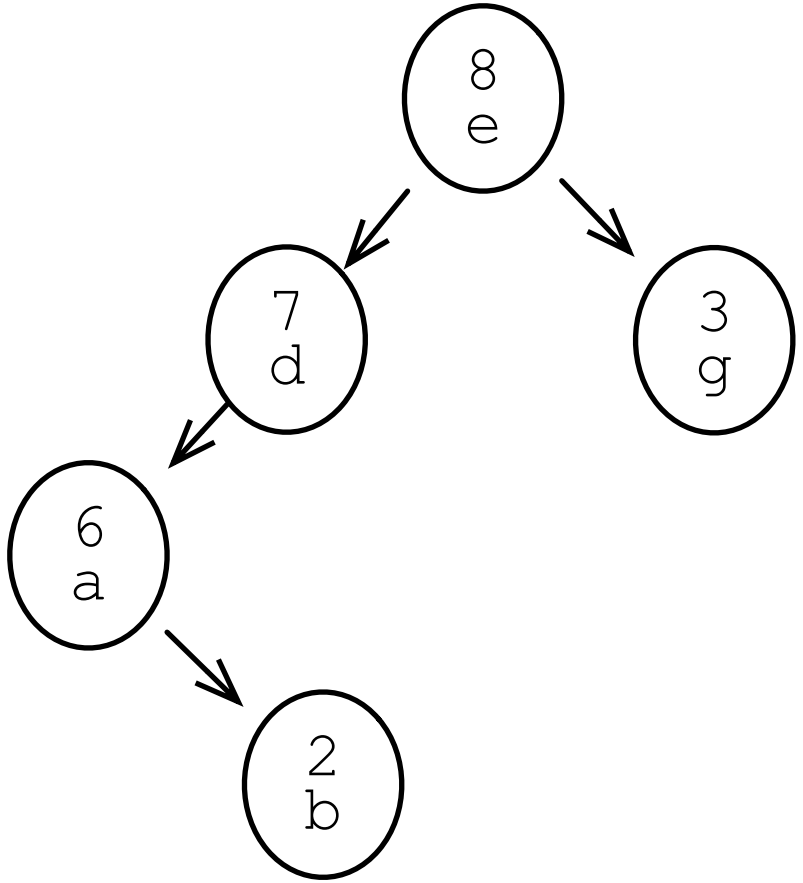
After inserting 'a', the
priorities are messed up again.

Rotate.

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**

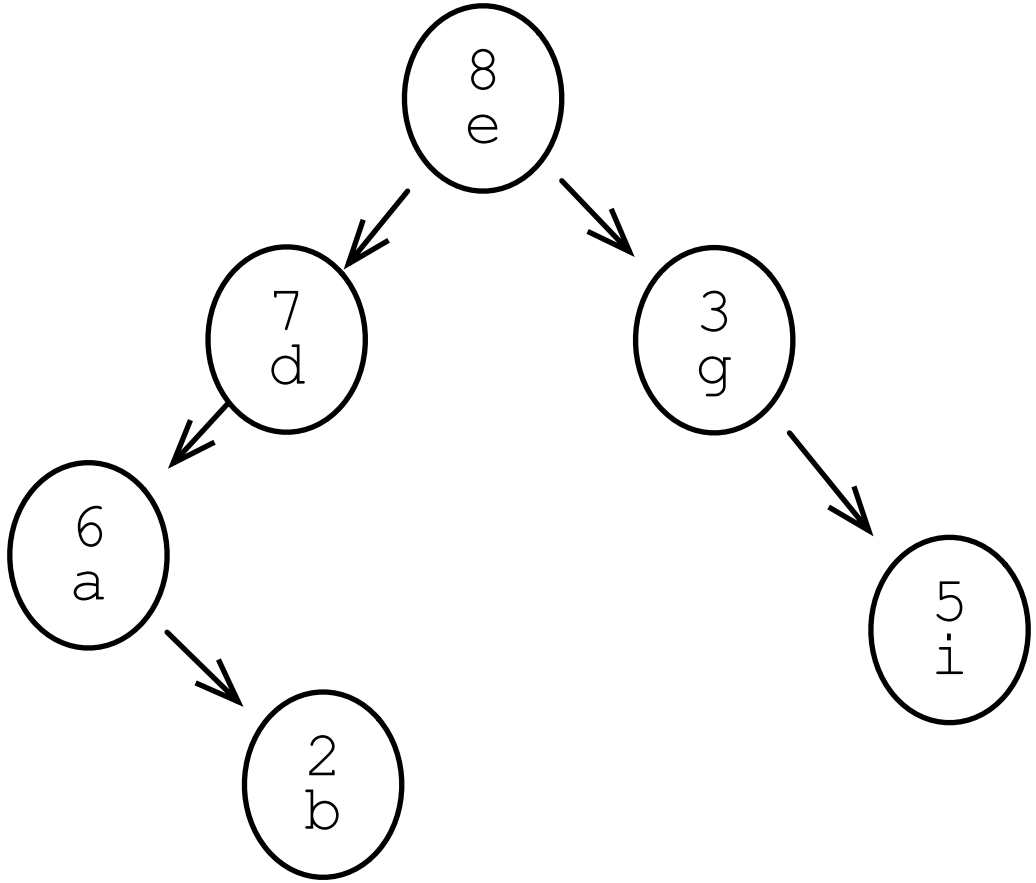


d	7
b	2
e	8
f	1
a	6
g	3
i	5
c	4
h	9



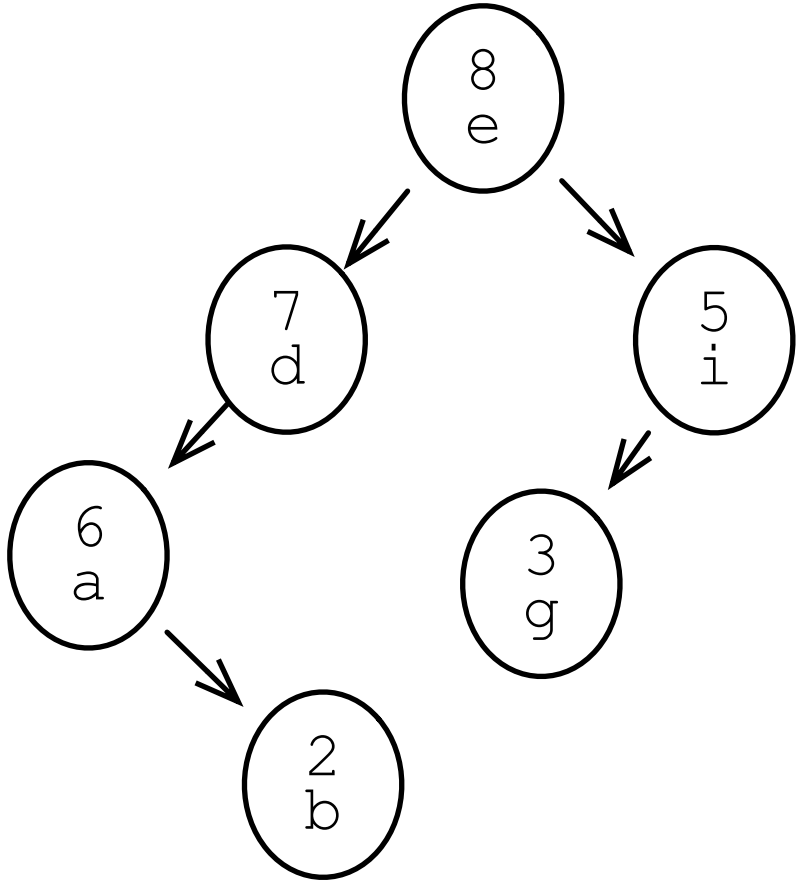
Inserting 'g' causes no problems.

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**

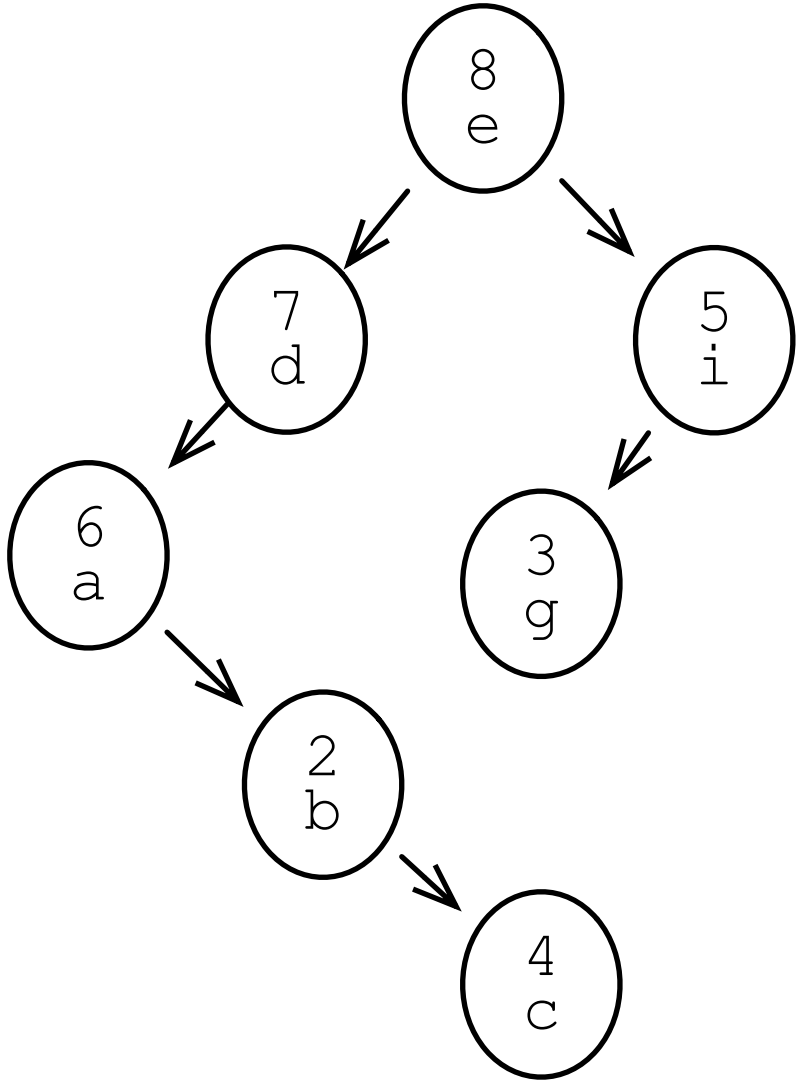


Inserting 'i' causes breaks the heap, so rotate.

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**

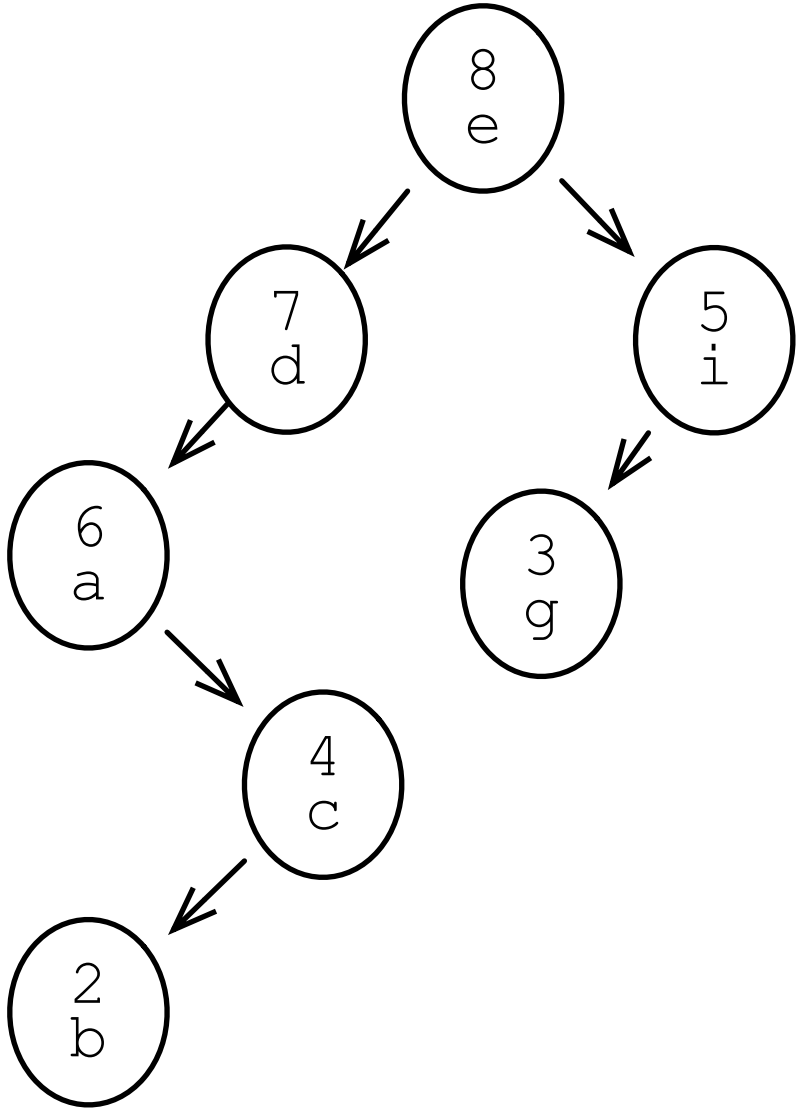


d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**

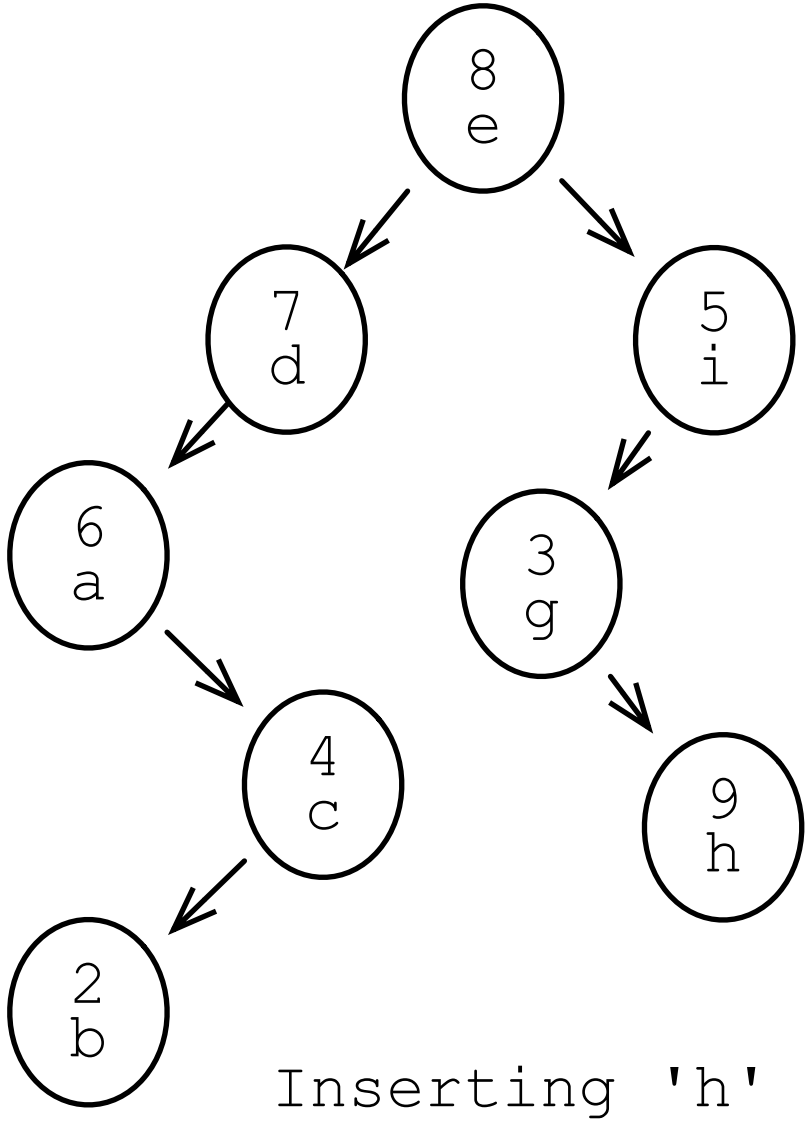


Inserting 'c' breaks the heap. Rotate...

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**

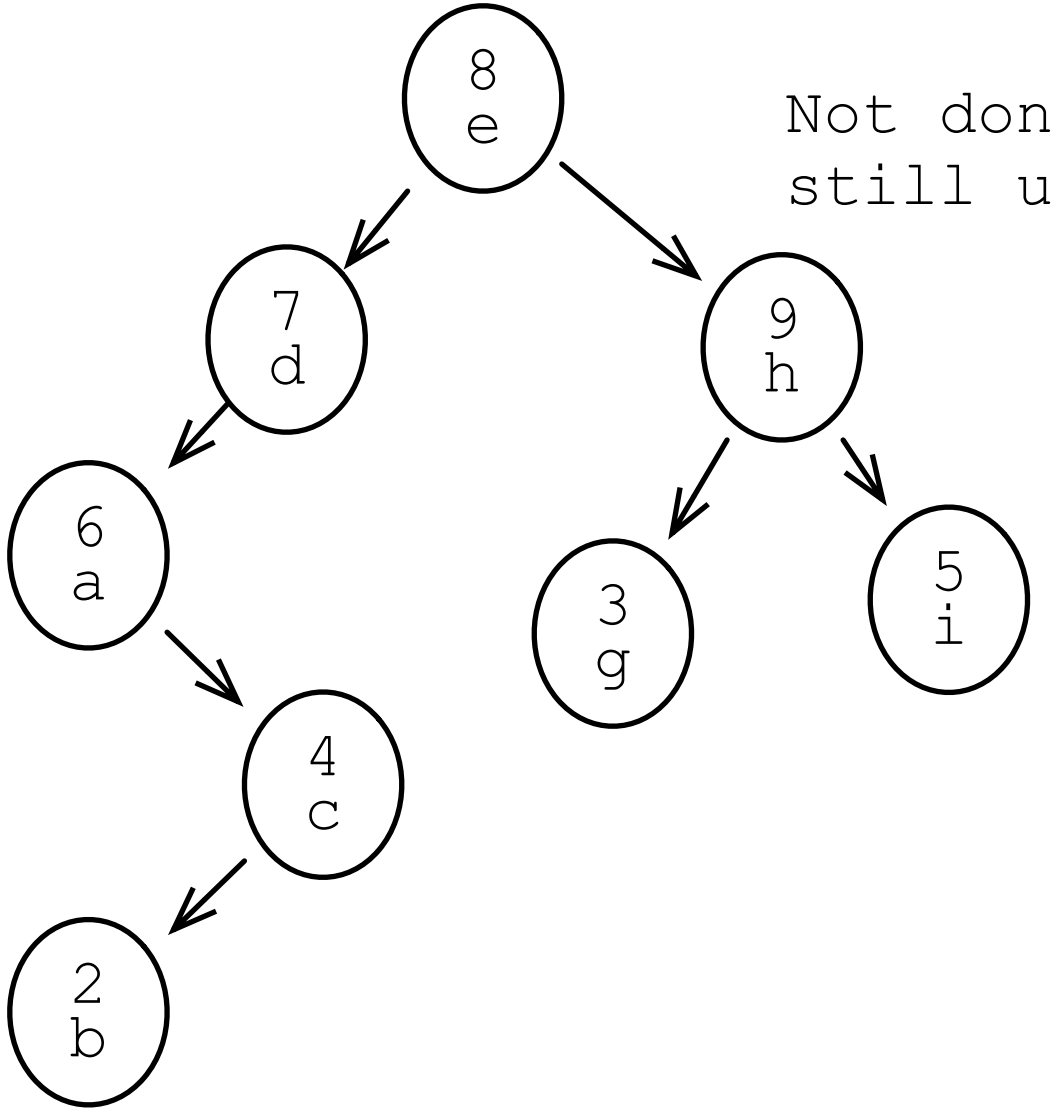


d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



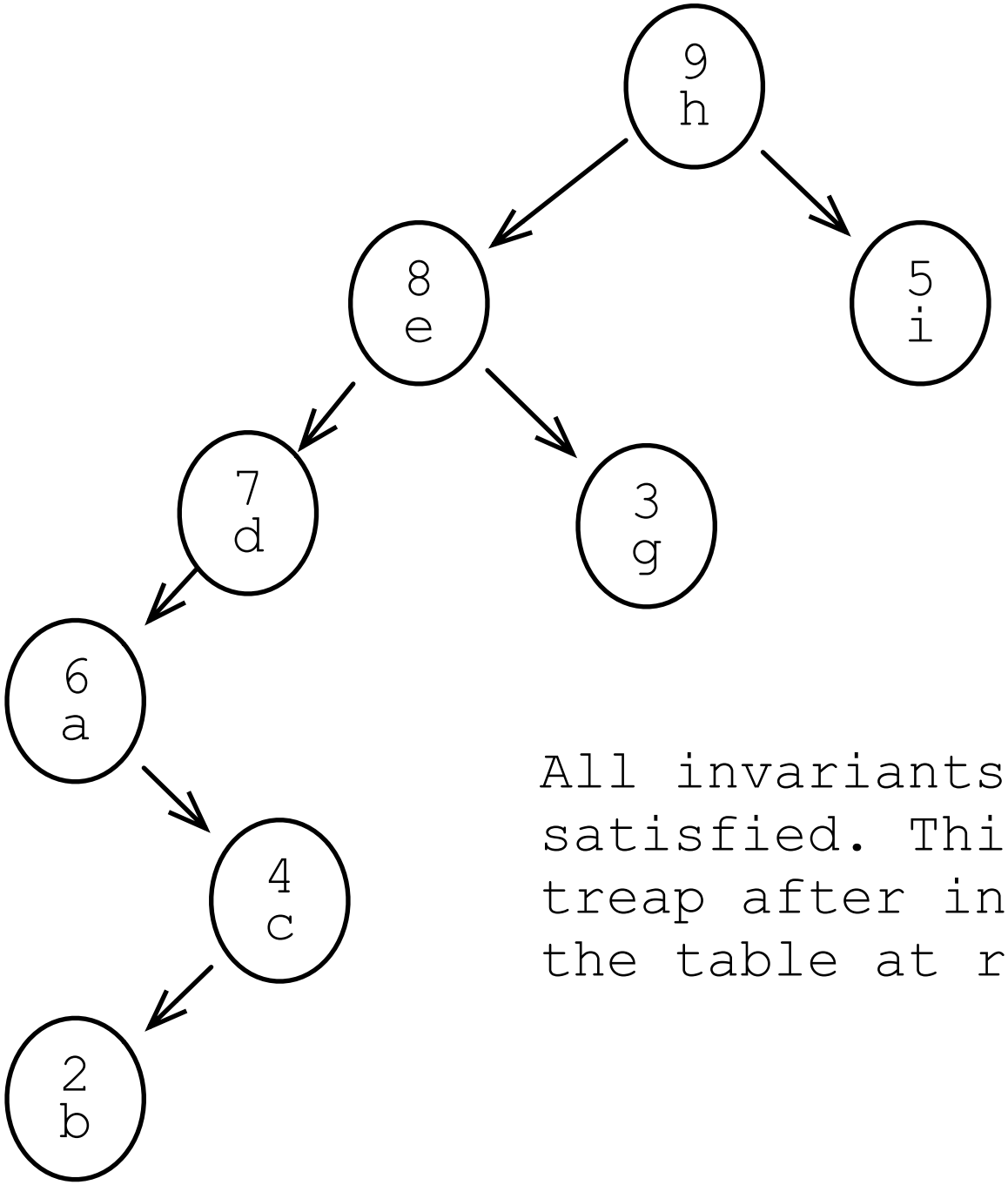
Inserting 'h' breaks the heap. We can see that its priority will bubble to the very top.

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



Not done yet. Heap is still unhappy.

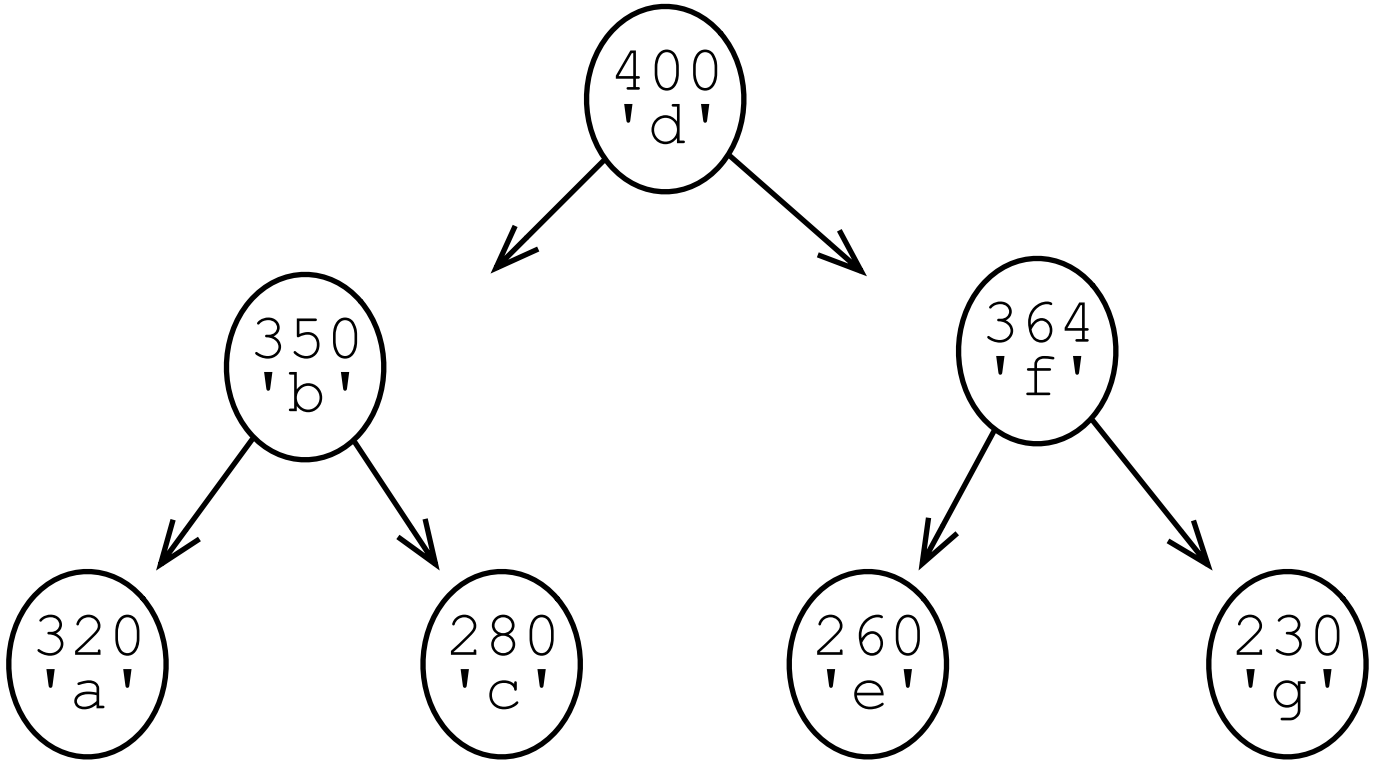
d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**



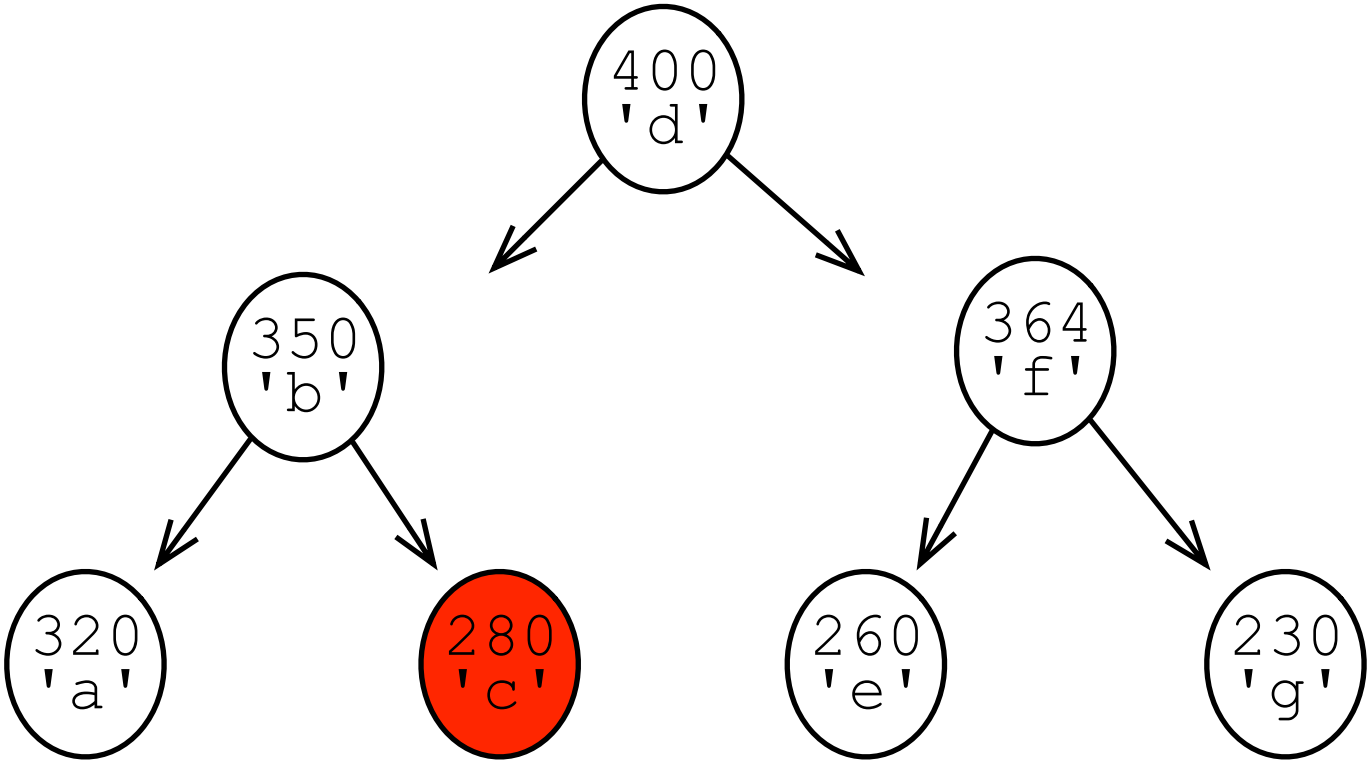
All invariants are now satisfied. This is the treap after inserting the table at right.

d	**7**
b	**2**
e	**8**
f	**1**
a	**6**
g	**3**
i	**5**
c	**4**
h	**9**

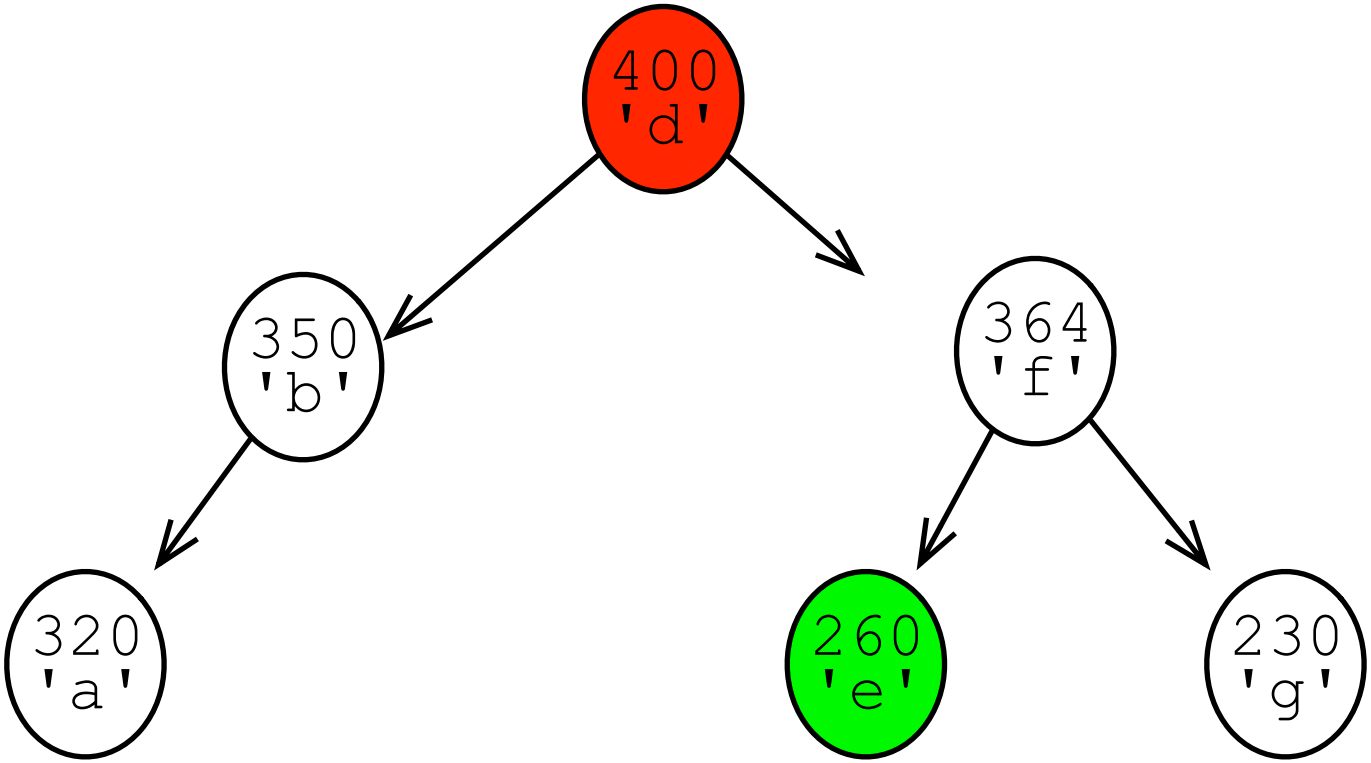
How does deleting work?
Similar to deleting from a
BST but with an additional
step of rotations to fix
broken heap invariants.



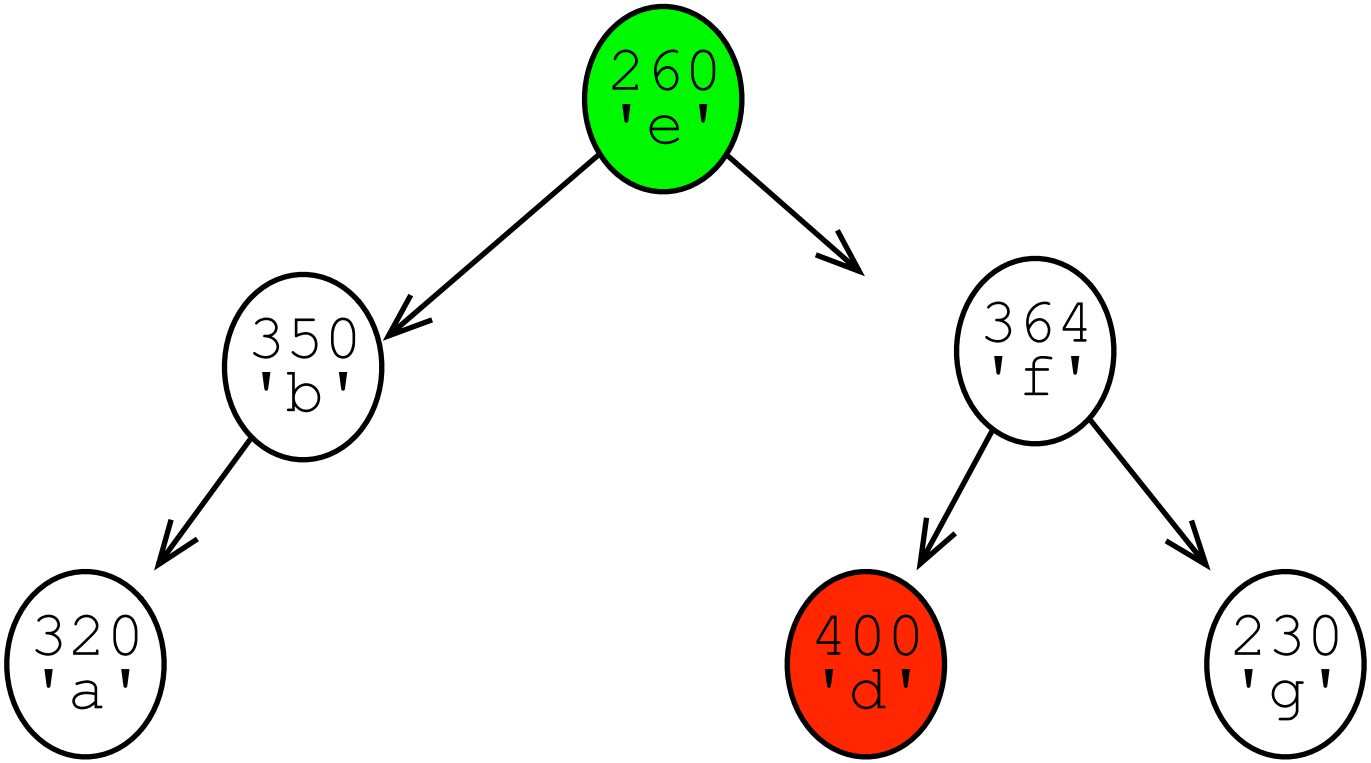
Deleting a leaf is obvious. Just remove the reference to it.



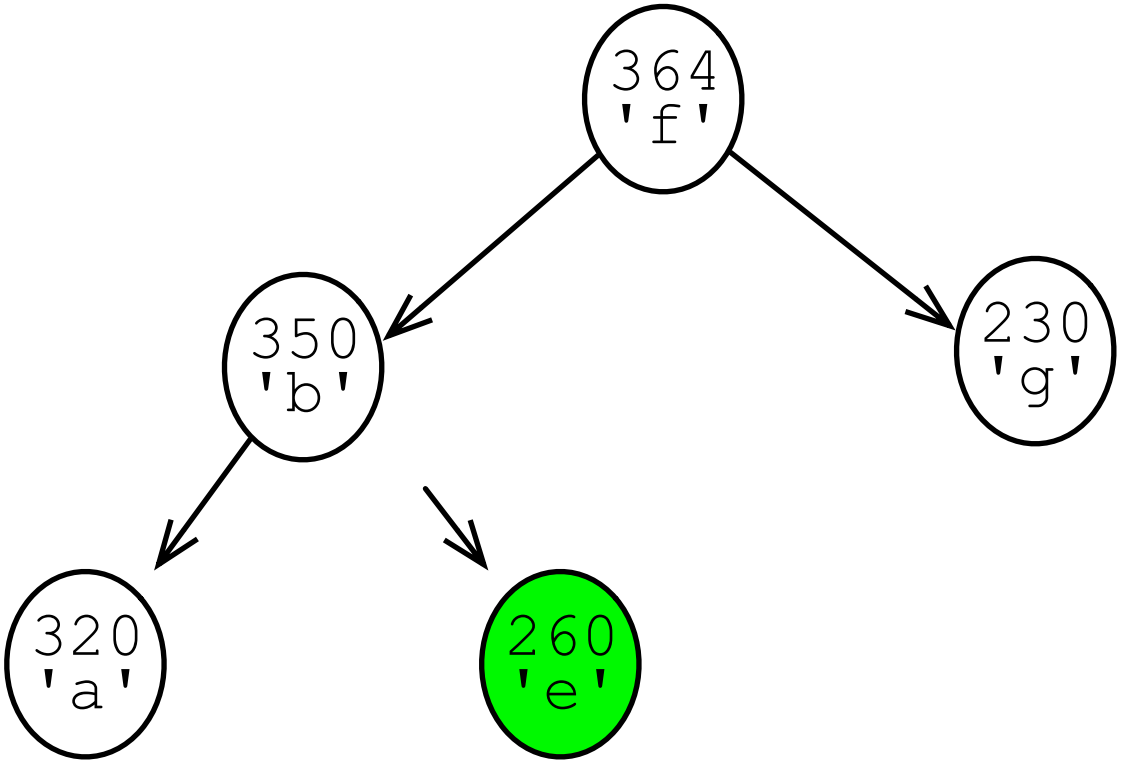
Deleting an internal node is just like with a BST remove operation. Say we remove the root node. It isn't a leaf, so find its sort-order successor, swap them, and remove it as you normally would.



Swapped doomed node with sort-order successor node (priority and all).



Post-rotation to preserve heap invariant *and* BST invariant.



Bonus Round!

You can re-assign the priority of a node over time. As the treap is used to find nodes, you may notice some nodes are more popular than others. Other nodes may never be accessed at all.

You may promote frequently used nodes by increasing their priority; demote unused nodes by reducing their priority.

This way, the most commonly used items are at the top of the tree. This doesn't change the big-oh notation, but it *does* reduce the constant number of operations to reach a node.