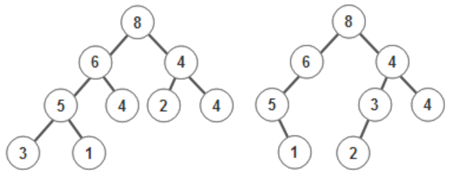
Heaps

We introduce the *heap*: a tree that maintains a bag of integers with O(log n) insertion and deletion times for a heap of size n. We'll see how to store a heap in an array. There are two kinds of heaps: min-heaps and max-heaps.

A *heap* is a binary tree that satisfies two properties.

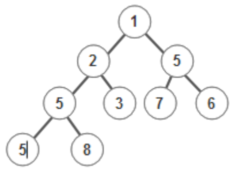
1. **Completeness**: Every level of the tree (except possibly the lowest) is completely filled, and on the lowest level the nodes are as far left as possible.

   The first tree on the right is complete. The second tree is not, for at least two reasons: (1) The node with value 6 is missing a right subtree; this is a "hole:". (2) On the lowest level, the node with value 1 is not as far left as possible.

   Since a heap is complete, a heap of size n has height O(log n).

2. **Heap-order invariant**: For a max-heap, the value in every node is at most the value in its parent. For a min-heap, the value in every node is at least the value in its parent.

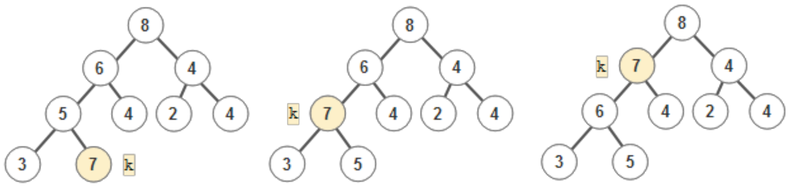
   

   The complete tree above to the right (with root 8) is a max-heap, for the value in each node is no bigger than the value in its parent. The complete tree to the right is a min-heap because the value in each node is no smaller than the value in its parent.

It should be clear from the examples that a heap can have duplicate values. Later, we will see uses of heaps where duplicates are not allowed.

Adding an element to a heap

To add a value to a heap, add it at the next possible position, keeping the tree complete. In the leftmost tree below, assume that 7, in the node named k, was just added to a max-heap. The heap-order invariant is no longer true because 5 < 7. To truthify the heap-order invariant, the 7 has to be bubbled up. First, swap k's value with its parent's value (second tree below). The parent of k is still smaller, so swap again (third tree below). Now, the value in node k is not larger than its parent and the heap-invariant is true again.



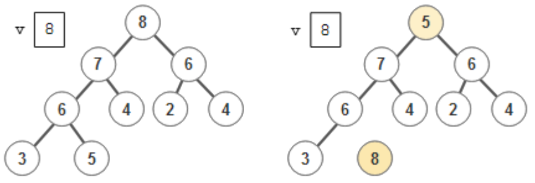
This bubbling-up process is described to the right. As long as k's value is greater than its parent's value, k's value is bubbled up.

Since the height of a heap of size n is O(log n), bubbling up takes time O(log n).

```
// invariant: The tree is a max-heap except that node
// k's value may be greater than its parent's value.
while (k has a parent and k's value > parent's value) {
    Swap values in k and its parent;
    Change k to the parent;
}
```

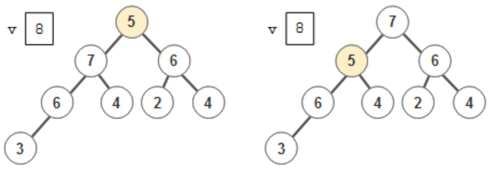
Polling the heap

By *polling* a max-heap, we mean removing (and returning) its largest value, which is at the root of the heap. (Similarly, polling a min-heap removes its smallest value.) Polling is generally done as a three-step process. (1) save the root value in a variable. We have done this in the first heap to the right. (2) Swap the root with the last node and remove the last node from the tree. The second tree to the right shows this. The node containing 8 is no longer part of the tree. Step (3) is given on the next page.



In some uses of a heap, swapping can be replaced by just assigning the last node's value to the root. In others, it's better to swap.

(3) **Bubbling down**. To the right, we show the tree after the values in the root and last node were swapped. We don't show the removed node, which contained 8.



This tree no longer satisfies the max-heap invariant because the root value 5 is smaller than the 7 and 6 at the roots of its subtrees. In order to truthify the max-heap invariant, the 5 must be bubbled down.

But we have to be careful. Bubbling down to the right would cause the 6 to be at the root, and the 6 is smaller than the left subtree's root, 7. That's not good. Instead, always bubble down to the largest of the left- and right-subtree values. In this case, bubbling down once changes the tree to the second tree on the right above. Bubbling down once more will swap the 5 and the 6, and the max-heap invariant will be true again.

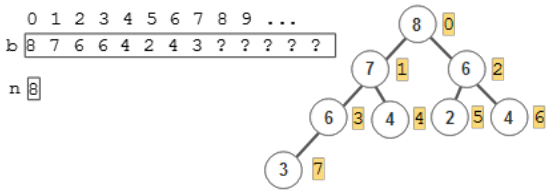
Obviously, bubbling down is more complicated than bubbling up.

```
// invariant: The tree is a max-heap except that
// node k's value may be less than a child's value.
while (k has a child AND
        k's value < a child's value) {
   Let c be the child with largest value;
   Swap values in k and c;
   Change k to c;
}
```

Storing the heap in an array

Above, we wrote pseudo code for bubbling values up and down a tree. You are justified in having questions about this. For example, how can one reference the parent of a node? Rarely have we seen implementations of trees where parents are easily accessible.

Because the tree is complete, we can store the values of the tree in the beginning of an array `b`. To the right is a heap of size 8, showing how its values are stored in array `b`. We have numbered the nodes in what one could call breadth-first order: the root is numbered 0, so its value goes in `b`[0]. The nodes on level 1 are numbered 1 and 2, so their values go in `b`[1] and `b`[2]. And so on. Variable `n` contains the size of the heap, so the values of the heap are in `b`[0..`n`-1]. We don't care what is in `b`[`n`..].



From now on, we will shorten the terminology *node whose number is* `k` to simply *node* `k`.

Here's a simple relationship between a node `k` and its parent and children:

- The parent of node `k` is node (`k` -1)/2 (using Java's **int** division).
- The children of node `k` are nodes 2`k`+1 and 2`k`+2.

Therefore, we can write the algorithm to bubble `b[k]` up (shown in pseudo code on the previous page) as shown to the right.

```
// Bubble b[k] up
int p= (k-1)/2;
// invariant: The tree is a max-heap except that node
// k's value may be greater than its parent's value AND
// p is k's parent (if k != 0)
while (k > 0 and b[k] > b[p]) {
   Swap b[k] and b[p];
   k= p;
   p= (k-1)/2;
}
```

In the JavaHyperText entries for heaps and for trees, you can find Java code for maintaining a heap in an array**.**