Consider implementing a priority queue using an array, a linked list, or a BST. For each, describe how each of the priority queue operations would be implemented, and what the worst-case time would be.

Using an Array

If we use an array to store the values in the priority queue, we can either store the values in sorted order (which will make the insert operation slow, and the removeMax operation fast), or in arbitrary order (which will make the insert operation fast, and the removeMax operation slow). Note also that we'll need a field to keep track of the number of values currently in the queue, we'll need to decide how big to make the array initially, and we'll need to expand the array if it gets full. Here's a partial class definition:

```
public class PriorityQueue {
    // *** fields ***
    private Comparable[] queue;
    private int numItems;
    private static final int INIT_SIZE = 10;

    // *** constructor ***
    public PriorityQueue() {
        queue = new Comparable[INIT_SIZE];
  numItems = 0;
    }

    ...
}
```

As mentioned above, if we keep the array sorted, then the insert operation is worst-case O(N) when there are N items in the queue: we can find the place for the new value efficiently using binary search, but then we'll need to move all the larger values over to the right to make room for the new value. As long as we keep the array sorted from low to high, the removeMax is O(1) -- just decrement numItems and return the last value.

If we keep the array unsorted, then insert is O(1) -- ignoring the time to expand the array, but removeMax is O(N) since we must search the whole array for the largest value.

The empty operation is trivial -- just return true if numItems == 0, so it (and the constructor) are both O(1).

Using a Linked List

Using a linked list is similar to using an array; again, we can keep the list sorted (which makes insert O(N)) or unsorted (which makes removeMax O(N)). Note that if the list is sorted we must use linear search to find the place to insert the new value, but there is no need to move old values over. Also, unless we maintain a "tail" pointer, we should keep the list in order from high to low, since removing from the front of the list can be done in O(1) time. Note that for a linked list we don't need the numItems field; the empty operation returns true if the list is empty (which can be determined in O(1) time).

Using a BST

If we use a BST, then the largest value can be found in time proportional to the height of the tree by going right until we reach a node with no right child. Removing that node is also O(tree height), as is inserting a new value. If the tree stays balanced, then these operations are no worse than O(log N); however, if the tree is not balanced, insert and removeMax can be O(N). As for the linked-list implementation, there is no need for a numItems field.

Heaps

A **heap** is a binary tree (in which each node contains a Comparable key value), with two special properties:

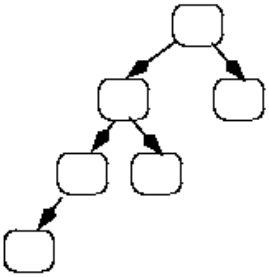
The **ORDER** property:

> For every node n, the value in n is **greater than or equal to** the values in its children (and thus is also greater than or equal to all of the values in its subtrees).

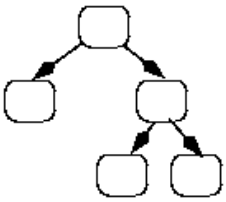
The **SHAPE** property:

1. All leaves are either at depth d or d-1 (for some value d).
2. All of the leaves at depth d-1 are to the **right** of the leaves at depth d.
3. (a) There is at most 1 node with just 1 child. (b) That child is the **left** child of its parent, and (c) it is the **rightmost** leaf at depth d.

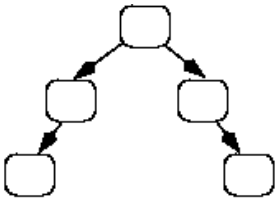
Here are some binary trees, some of which violate the shape properties, and some of which respect those properties:



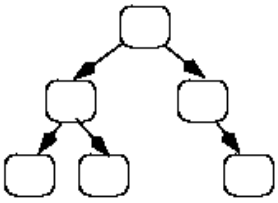
NO: violates shape property 1



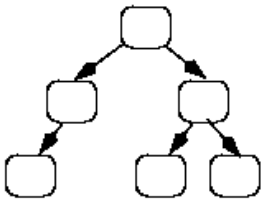
NO: violates shape property 2



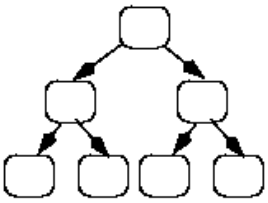
NO: violates shape property 3(a)



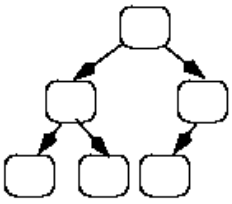
NO: violates shape property 3(b)



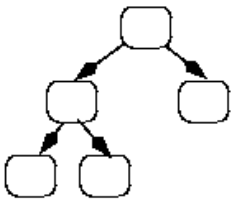
NO: violates shape property 3(c)



YES!

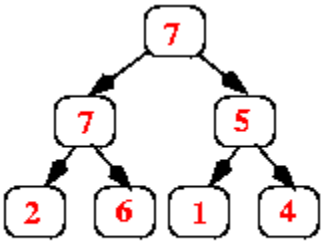


YES!

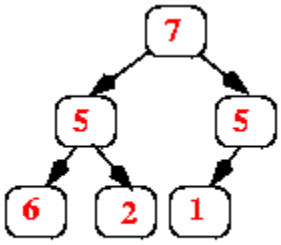


YES!

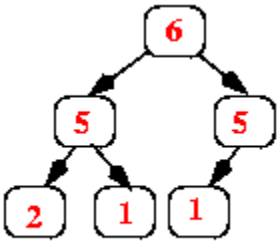
And here are some more trees; they all have the **shape** property, but some violate the order property:



YES!



NO (6 is > 5)



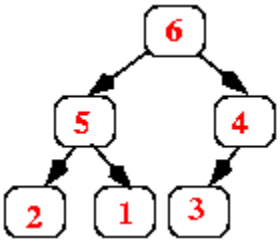
YES!

Implementing priority queues using heaps

Now let's consider how to implement priority queues using a heap. The standard approach is to use an **array** (or an ArrayList), starting at position 1 (instead of 0), where each item in the array corresponds to one node in the heap:

- The root of the heap is always in array[1].
- Its **left** child is in array[2].
- Its **right** child is in array[3].
- In general, if a node is in array[k], then its left child is in array[k*2], and its right child is in array[k*2 + 1].
- If a node is in array[k], then its **parent** is in array[k/2] (using integer division, so that if k is odd, then the result is truncated; e.g., 3/2 = 1).

Here's an example, showing both the conceptual heap (the binary tree), and its array representation:



[0]	[1]	[2]	[3]	[4]	[5]	[6]
	6	5	4	2	1	3

Note that the heap's "shape" property guarantees that there are never any "holes" in the array.

The operations that create an empty heap and return the size of the heap are quite straightforward; below we discuss the **insert** and removeMax operations.

Implementing insert

When a new value is inserted into a priority queue, we need to:

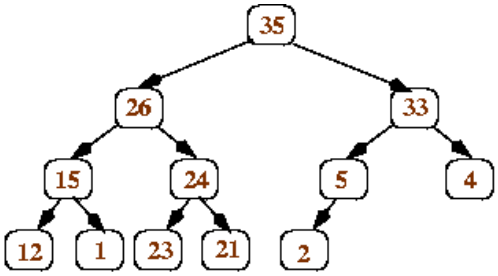
- Add the value so that the heap still has the order and shape properties, and
- Do it efficiently!

The way to achieve these goals is as follows:

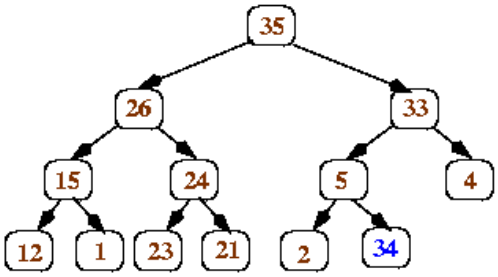
1. Add the new value at the **end** of the array; that corresponds to adding it as a new rightmost leaf in the tree (or, if the tree was a **complete** binary tree, i.e., all leaves were at the **same** depth d, then that corresponds to adding a new leaf at depth d+1).

2. Step 1 above ensures that the heap still has the **shape** property; however, it may not have the **order** property. We can check that by comparing the new value to the value in its parent. If the parent is smaller, we swap the values, and we continue this check-and-swap procedure up the tree until we find that the order property holds, or we get to the root.

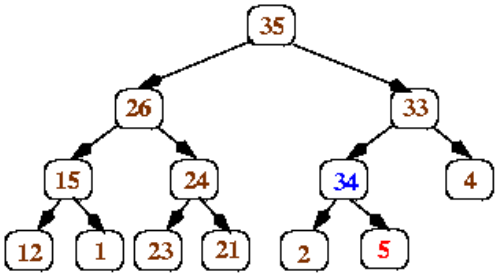
Here's a series of pictures to illustrate inserting the value 34 into a heap:



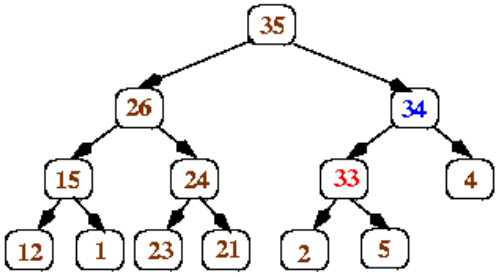
original heap



step 1 (add 34 as rightmost leaf)



step 2 (swap with parent)

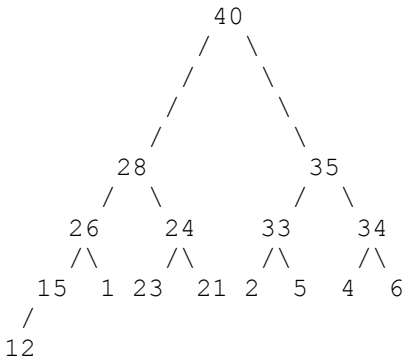


step 2 again (swap with parent)

All done!

TEST YOURSELF #2

Insert the values 6, 40, and 28 into the tree shown above (after 34 has been inserted).



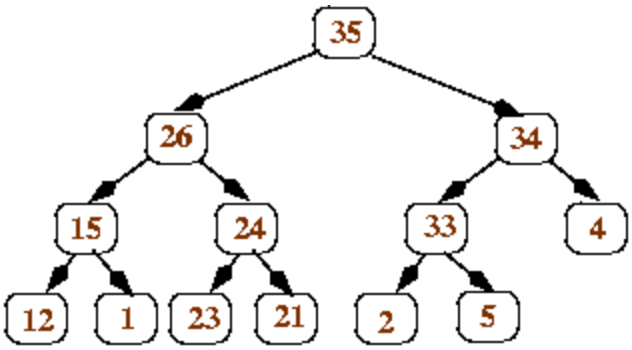
Implementing removeMax

Because heaps have the **order** property, the largest value is always at the root. Therefore, the **removeMax** operation will always remove and return the root value; the question then is how to replace the root node so that the heap still has the order and shape properties.

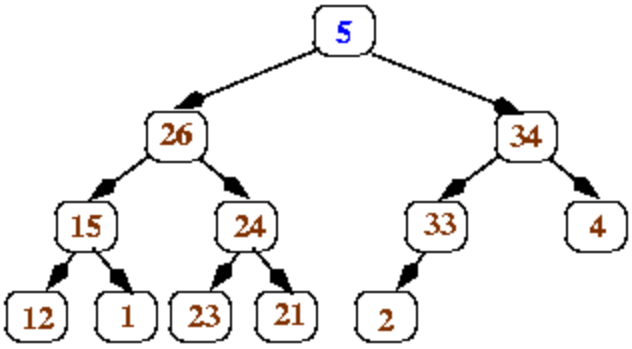
The answer is to use the following algorithm:

1. Replace the value in the root with the value at the end of the array (which corresponds to the heap's rightmost leaf at depth d). Remove that leaf from the tree.
2. Now work your way down the tree, swapping values to restore the order property: each time, if the value in the current node is less than one of its children, then swap its value with the **larger** child (that ensures that the new root value is larger than both of its children).

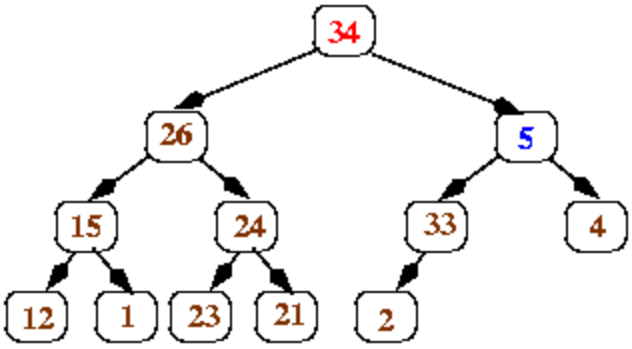
Here's a series of pictures to illustrate the removeMax operation applied to the heap shown above.



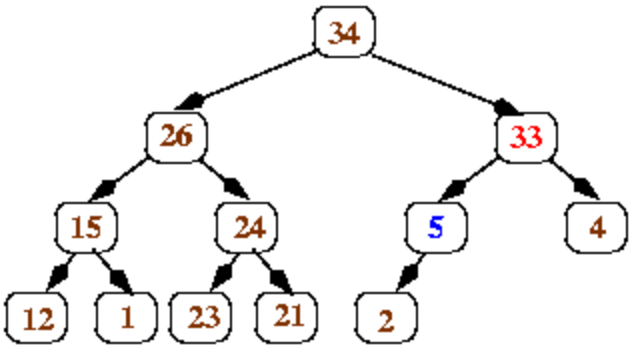
original heap



step 1: extract root value and replace with last leaf



step 2: swap with larger child

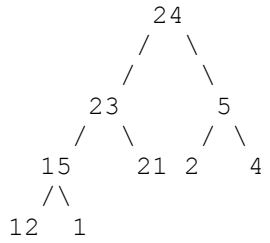


step 2: swap with larger child again

All done!

TEST YOURSELF #3

Perform 3 more removeMax operations using the example tree.



Complexity

For the **insert** operation, we start by adding a value to the end of the array (constant time, assuming the array doesn't have to be expanded); then we swap values up the tree until the order property has been restored. In the worst case, we follow a path all the way from a leaf to the root (i.e., the work we do is proportional to the height of the tree). Because a heap is a **balanced** binary tree, the height of the tree is O(log N), where N is the number of values stored in the tree.

The removeMax operation is similar: in the worst case, we follow a path down the tree from the root to a leaf. Again, the worst-case time is O(log N).

Summary

A priority queue is a data structure that stores priorities (comparable values) and perhaps associated information. A priority queue supports inserting new priorities, and removing/returning the highest priority. When a priority queue is implemented using a **heap**, the worst-case times for both insert and removeMax are logarithmic in the number of values in the priority queue.