Insert 10 values into a min-heap from a file. Display the data, level by level. Then delete 5 items. After each delete, display the heap, level by level.

**Heaps**

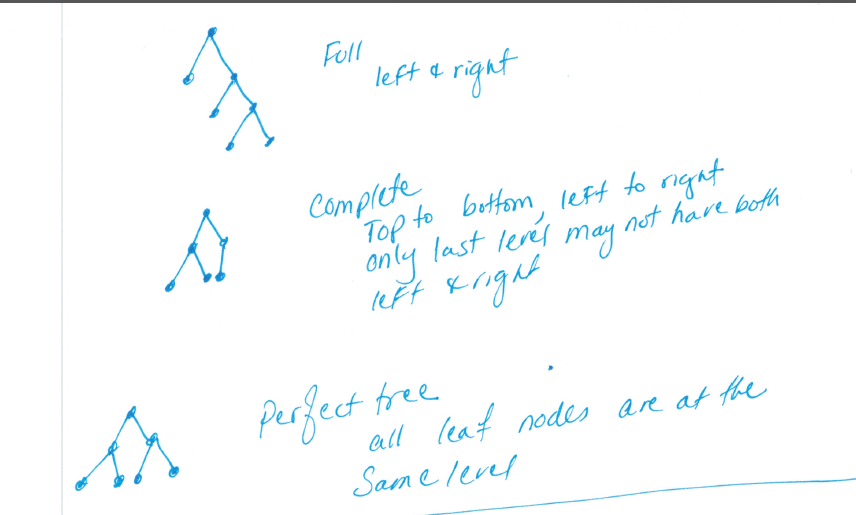
There are two types of heap: min-heap and max-heap. Elements in the heap are not perfectly ordered as in a binary search tree. The heap has two basic properties:

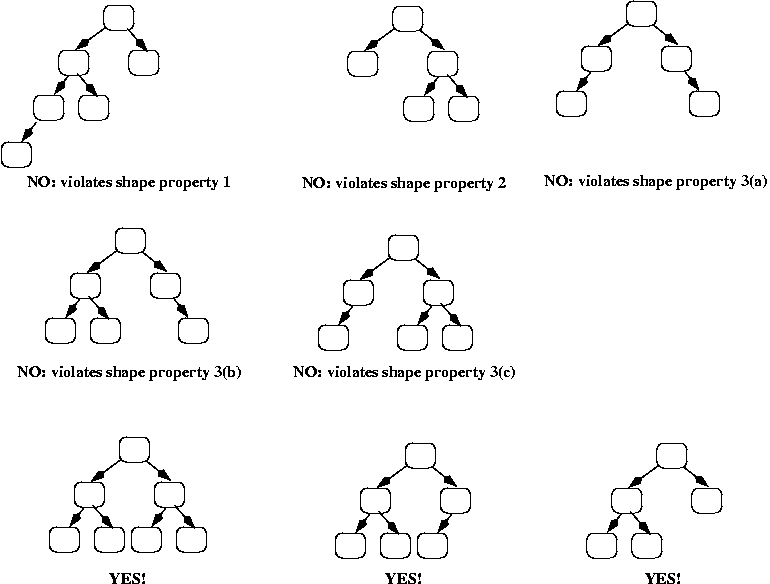
**Order property**

|  |  |
| --- | --- |
| Min-heap:  The value of each node is smaller than or equal to the values stored in each of its children. | Max-heap:  The value of each node is bigger  than or equal to the values stored in  each of its children |
|  |  |

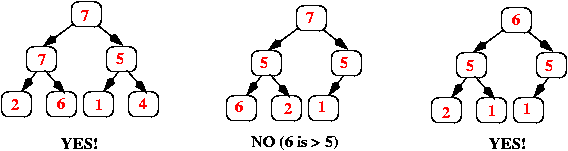
**Shape property**

It has to be a complete tree.

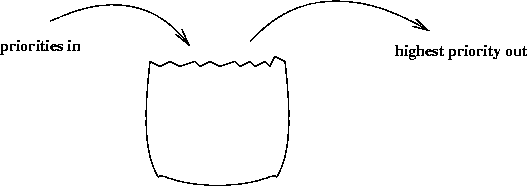




Given a max-heap, they all have the **shape** property, but some violate the order property.



**What is a heap used for: Priority Queues**

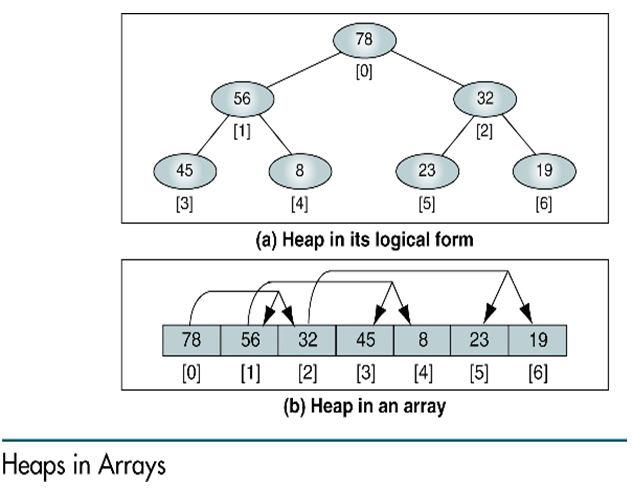


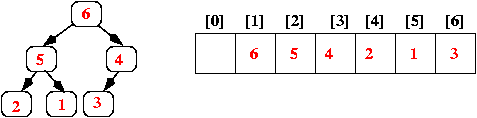
Think of a priority queue as a kind of bag that holds priorities. You can put one in, and you can take out the current **highest** priority.

A priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order by priority. A priority queue might be used, for example, to handle the jobs sent to the Computer Science Department's printer: Jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by graduate students, and finally those sent by undergraduates. The values put into the priority queue would be the priority of the sender (e.g., using 4 for the chair, 3 for professors, 2 for grad students, and 1 for undergrads), and the associated information would be the document to print. Each time the printer is free, the job with the highest priority would be removed from the print queue, and printed. (Note that it is OK to have multiple jobs with the same priority; if there is more than one job with the same **highest** priority when the printer is free, then any one of them can be selected.)

A priority queue can be implemented using many of the data structures that we've already studied (an array, a linked list, or a binary search tree). However, those data structures do not provide the most efficient operations. To make all of the operations very efficient, we'll use a new data structure called a **heap**.

**How to implement a heap? Use arrays**





Two options:

Option 1: Start with index 0

Children of a[i]

Left=a[2i+1]

Right=a[2i+2]

Parent of a[i]=(i-1)/2

Option 2: Start with index 1

Children of a[i]

Left=a[2i]

Right=a[2i+1]

Parent of a[i]=i/2

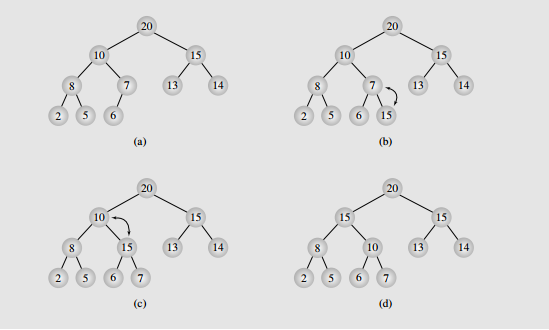
**Insert (Sift-up/Reheap-up)**

The following algorithm is for **max-heap**. For max-heap, ensure that the parent is bigger than children

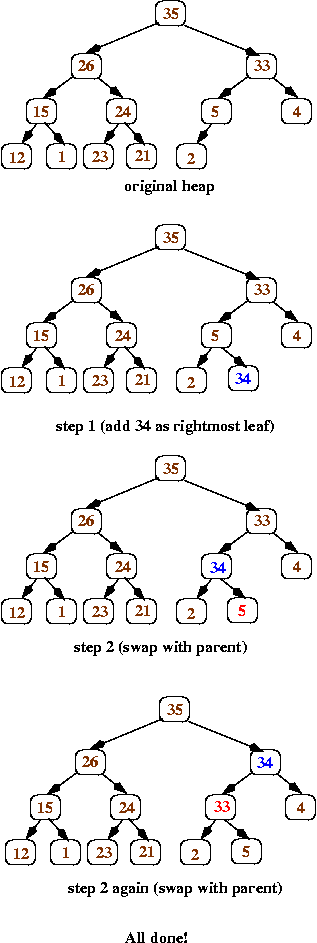
Add the value so that the heap still has the order and shape properties:

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. This ensures that the heap still has the shape property
2. To deal with 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.



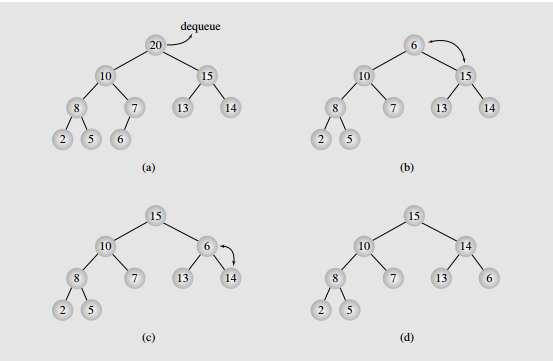
|  |  |
| --- | --- |
|  |  |



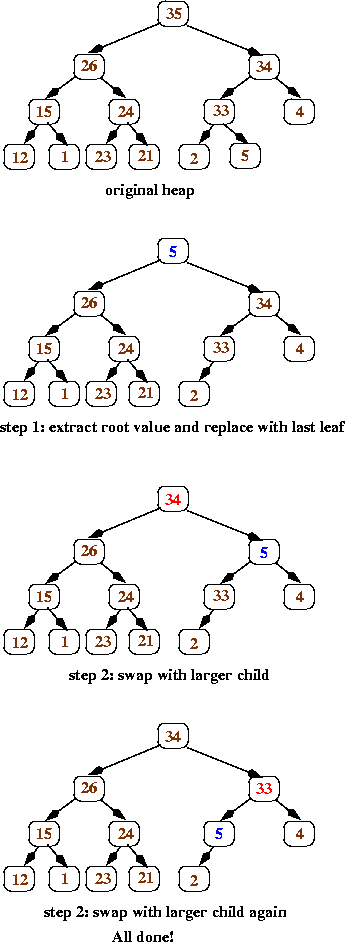
**Delete (Sift down, reheap-down)**

The following algorithm is for max-heap. For max-heap, ensure that the parent is bigger than children

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 smaller than one of its children, then swap its value with the larger child (that ensures that the new root value is bigger than both of its children).



|  |  |
| --- | --- |
|  |  |



//After the item has been inserted at the end, call this function

void siftUp(int nodeIndex) {

//figure out parent index

//if parent is smaller, then swap and siftUp parentindex

}

//First remove item from beginning, then move last item to the root

//position and finally call this function

void siftDown(int parent) {

// figure out which child is bigger

// if the parent is < biggest child, then swap and siftDown on the

// index of the biggest child

}

