Priority Queues and Heaps

Lecture 9, Week 10
July 18, 2013
CSC148H1S
Velian Pandeliev

Priority Queue

We are familiar with queues and stacks as ADTs. In a queue, the first element in is the first element out. In a stack, the last element in is the first element out.

A **priority queue** is an ADT in which every element has a **priority** associated with it.

In a priority queue, regardless of the order of insertion, the element with the **highest priority** is retrieved first.

Applications: barbecues, emergency waiting room, network management, pathfinding...

Priority Queue Operations

Priority queues should support at least the following two operations:

insert: add a new element to the PQ

extract_max: return the element with highest priority

What are some ways to implement a PQ?

	insert	**extract_max**
unsorted `list`	O(1)	O(n)
sorted `list`	O(n)	O(1)
BST	O(log n) to O(n)	O(log n) to O(n)

Heaps

We'd like to create a data structure that makes implementing a priority queue more efficient.

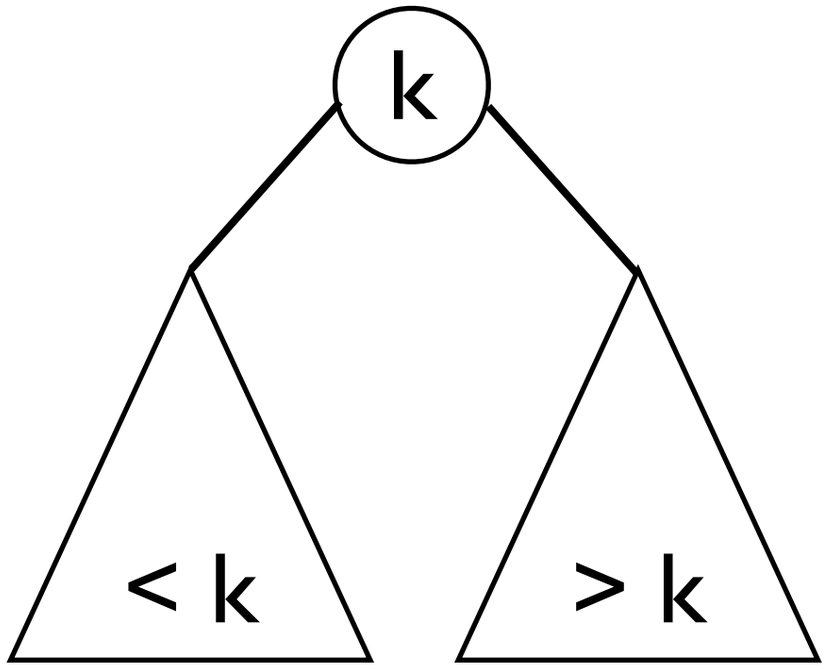
A **heap** is a **binary tree** that adheres to the following two properties:

- the **completeness** property: a heap is a complete binary tree. Every level except the last one is full, and the last one is filled from left to right.

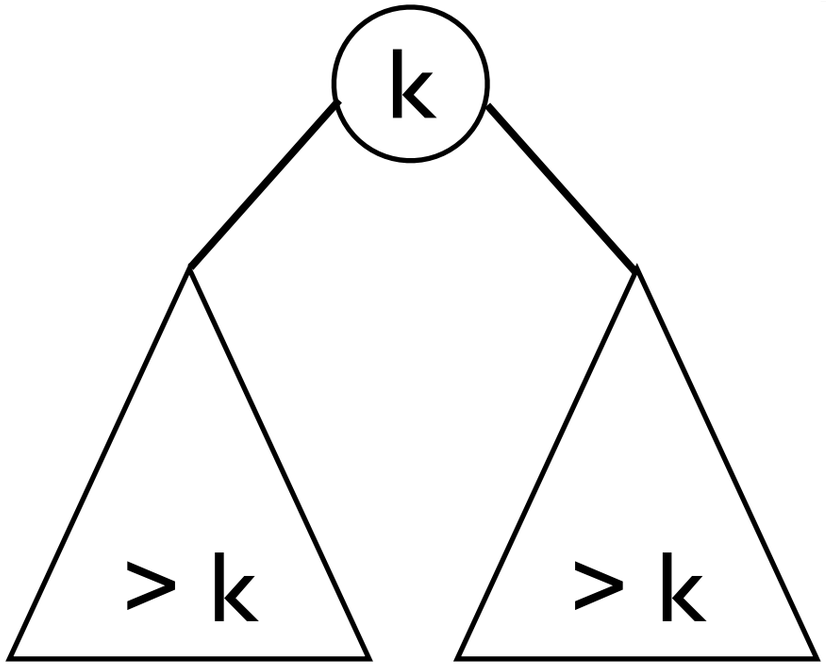
- the **heap** property: In a min-heap, every node is smaller than its two children. In a max-heap, every node is larger than its two children.

It follows that in a min-heap, the smallest node is the root, and in a max-heap, the largest node is the root.

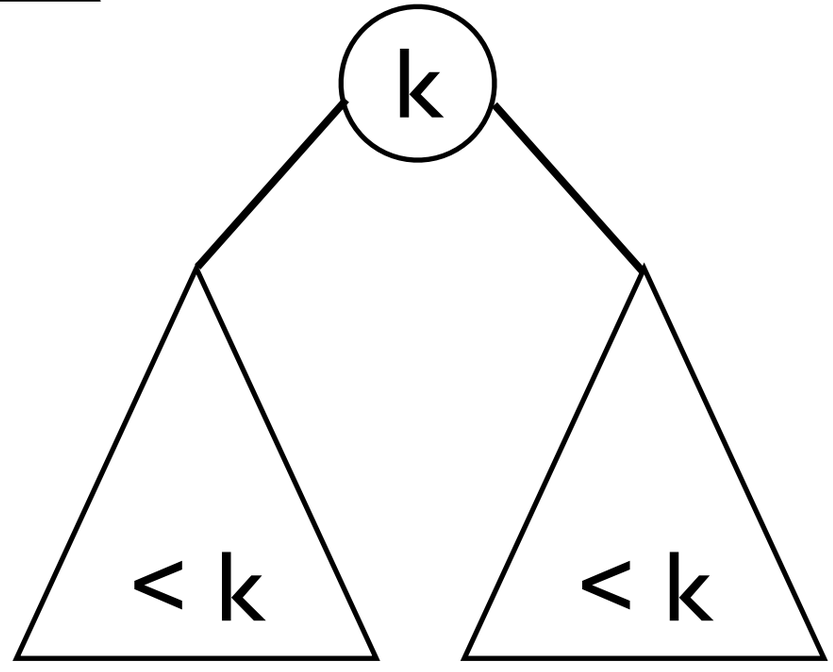
Heaps



BST



Min-heap



Max-heap

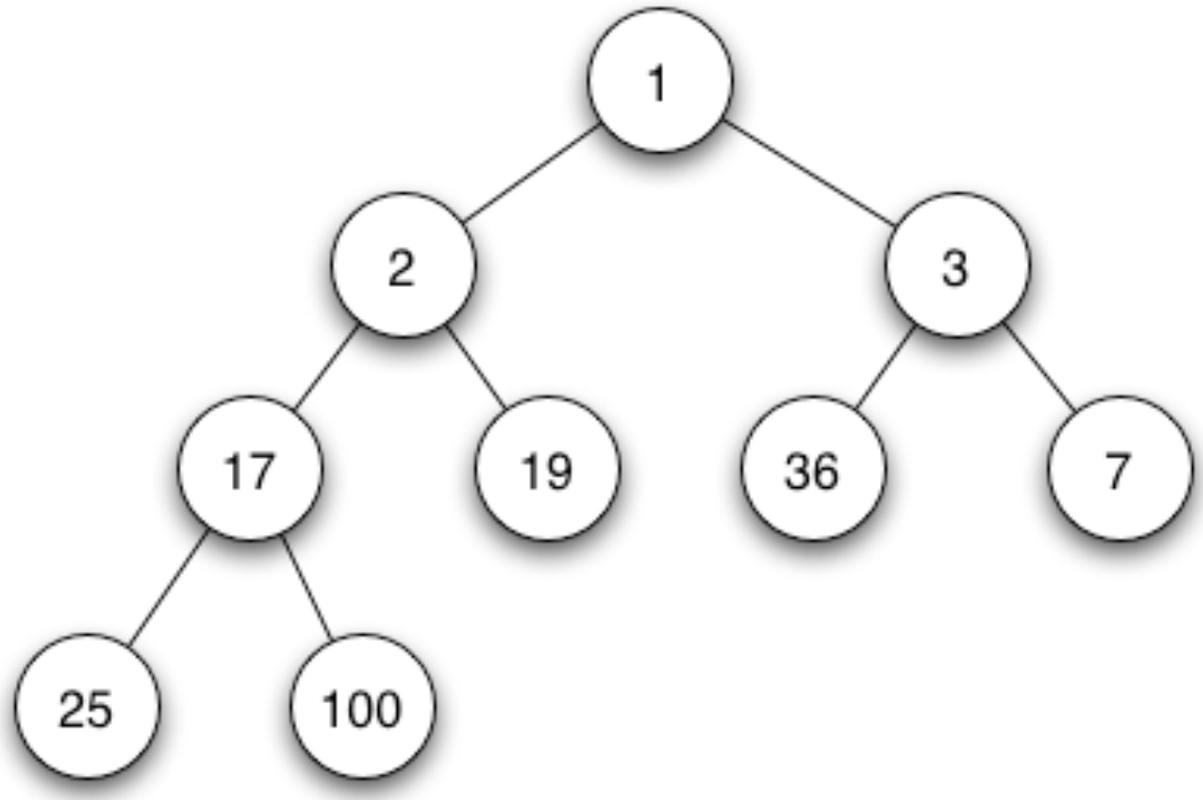
Priority Queues as Heaps

A priority queue can be implemented as either a min-heap or a max-heap, depending on the problem.

For instance, if priority reflects importance, then you'd like the most important element first, so you'd use a **max-heap**.

However, if priority reflects something like time left, you'd want the element with the least time, so you'd use a **min-heap**.

Implementing Heaps



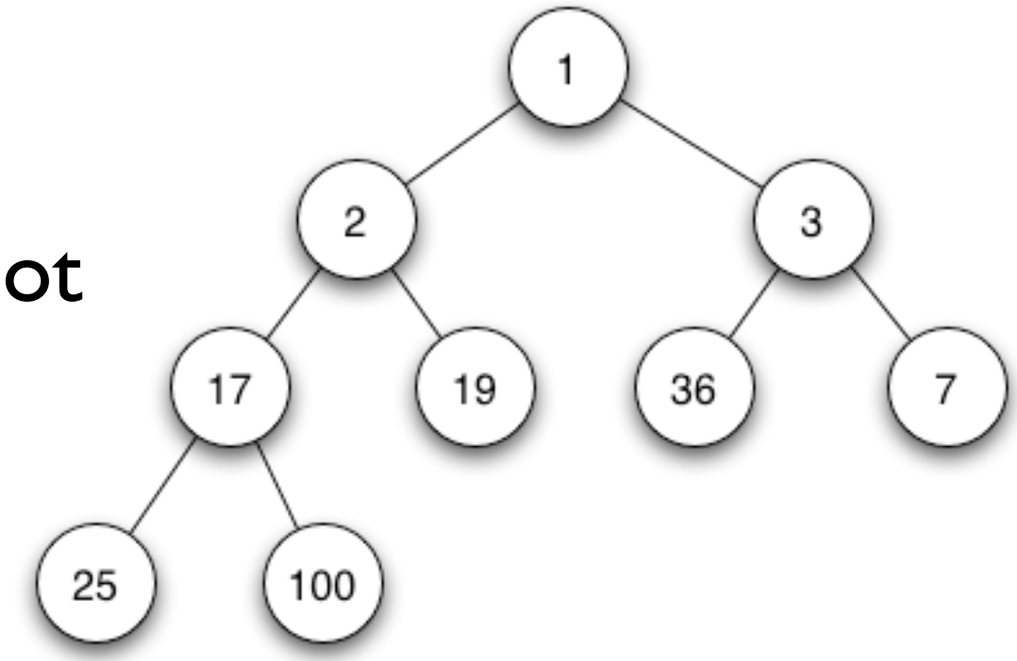
Heaps are binary trees.

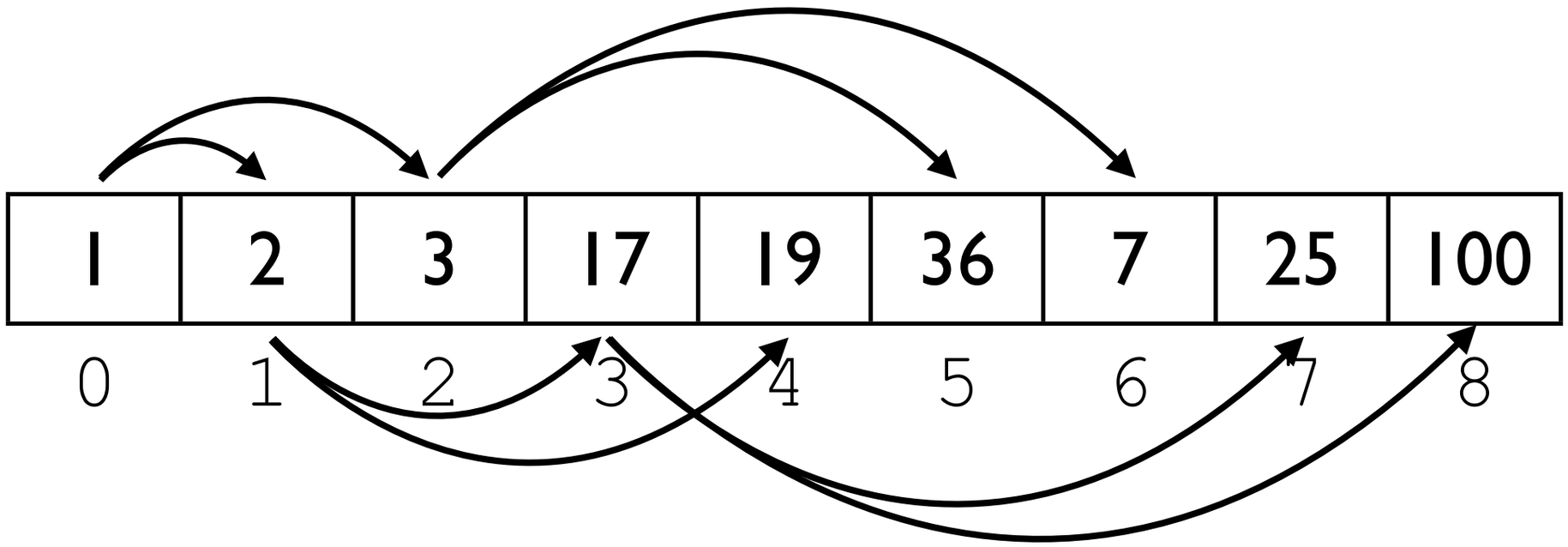
They could be implemented as linked nodes, like BSTs.

They could also be implemented as a single list.

Heap as a List

To implement a binary tree as a list:
- the first element (index 0) is the root
- for every element at index i:
  - its left child is at index 2*i + 1
  - its right child is at index 2*i + 2



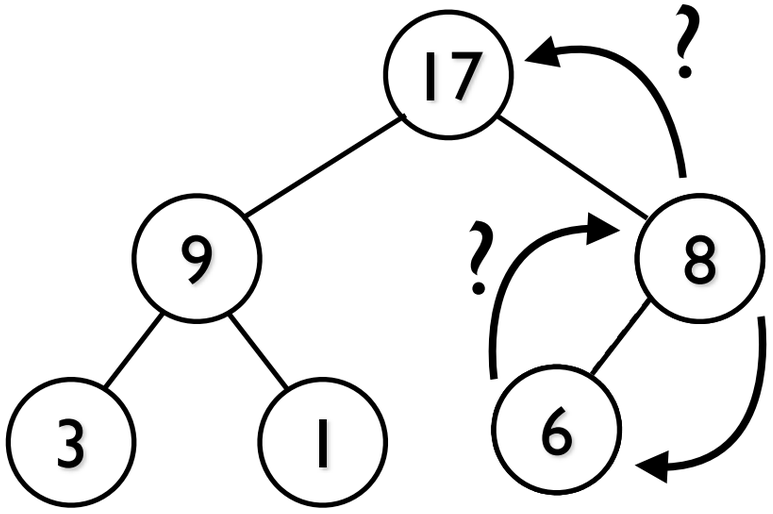


Parents and children

Q: How do we refer to nodes in the heap if it's implemented as a list?
A: We use a specific index in the list.
Q: Where is the root of the heap?
A: At index 0.
Q: How do we find a node's left child if the list only stores integers and not `Node` objects with links?
A: We find it by calculating the index where it should be stored. If our element is at index i, its left child is at 2*i + 1
Q: How will we know the node has no left child?
A: If the index is greater than or equal to the length of the list.

Heap Insertion

Since heaps are complete, there is only one valid place to insert into a heap: the next empty leaf spot.



So, by necessity we insert our new element (e.g. 8) there.

This has preserved the completeness property of our heap, but not the heap property.

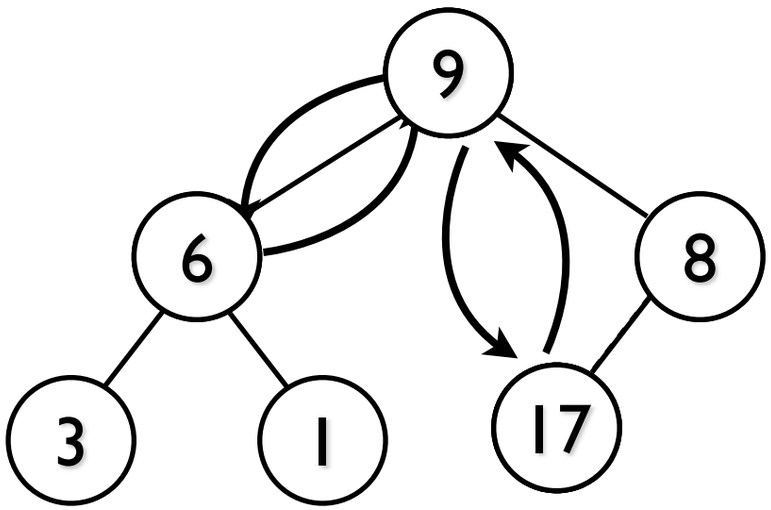
To fix this, we will swap the new value with that of its parent until the heap property is no longer violated.

This is called percolating up the heap.

Question: what's the time complexity of insertion?

Heap Deletion

The only thing we ever delete from a heap is the root.



However, the heap needs to "shrink" from the last leaf node.

So, we will swap the root with the last leaf and delete the last leaf.

Now we have a complete tree, but it's not a heap anymore.

The root is violating the heap property, so we should percolate it down by swapping it with the **greater child** at every step.

Why the greater child? And how fast is this?

Priority Queue Efficiency

Priority queues should support at least the following two operations:

insert: add a new element to the PQ

extract_max: return the element with highest priority

	insert	**extract_max**
unsorted `list`	O(1)	O(n)
sorted `list`	O(n)	O(1)
BST	O(log n) to O(n)	O(log n) to O(n)
heap	**O(log n)**	**O(log n)**

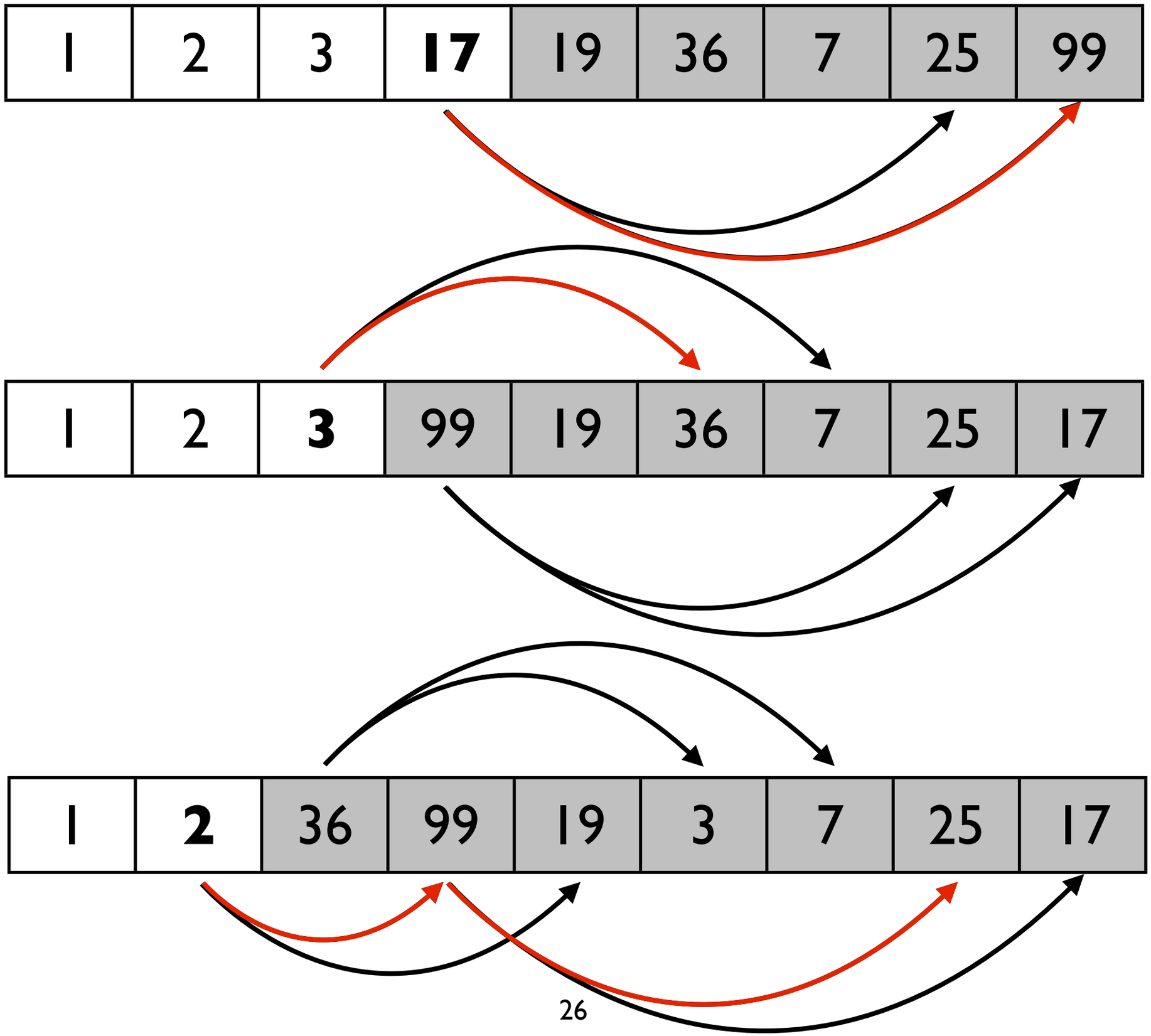
Heapify

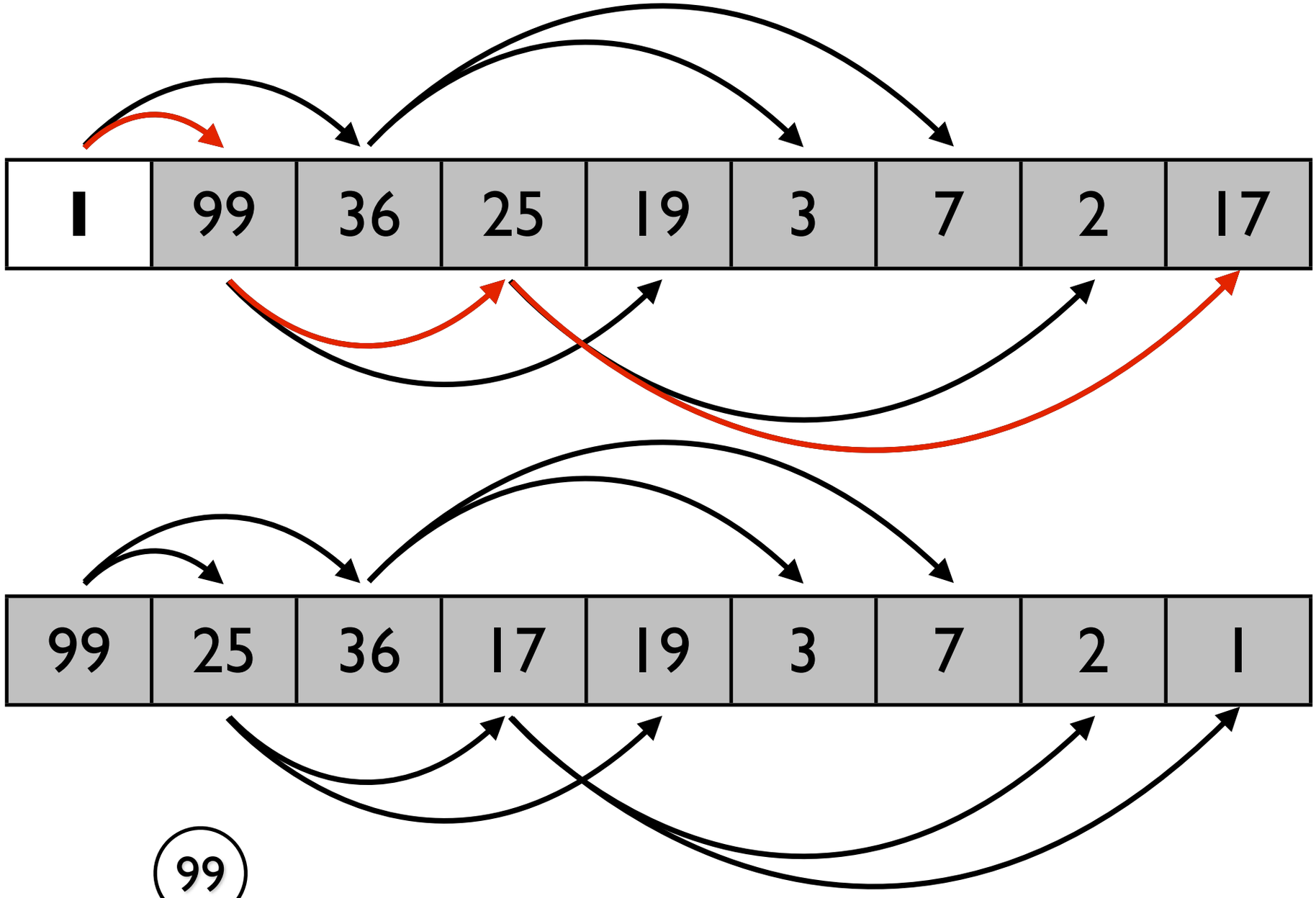
If we have a 1-D list of integers and we want to turn it into a heap, we have to make it conform to the heap property: every node should be greater than its two children (for a max heap).

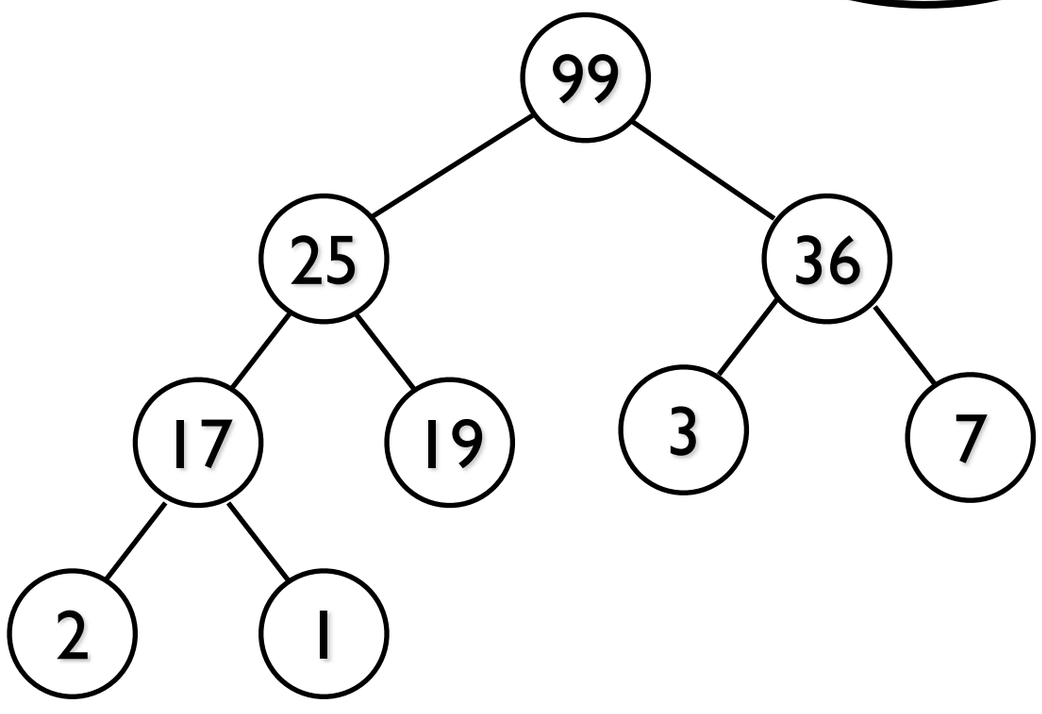
The second half of the list representation of a heap contains leaves, which already conform to the property (by not having children).

Starting at the last internal node in the list, we will make sure each element conforms as well by percolating down. Finally, when we reach the node at index 0, our entire list will be a valid max-heap.

We call this operation **heapifying** a list.


1 2 3 17 19 36 7 25 99
1 2 3 99 19 36 7 25 17
1 2 36 99 19 3 7 25 17





The grey part of the list are max-heaps at any point of the execution.