CSCI 136
Data Structures & Advanced Programming

Priority Queues
Introduction & Implementations

Priority Queues

- Priority Queues
  - Supports Add & Remove (Min) operations
- Heaps
  - A “somewhat-ordered” data structure
    - Conceptual structure
    - Efficient implementations
      - Array Representations of (Binary) Trees

A New Data Structure

Goal: Design a structure S to hold items with *priorities*

- S should support operations
  - add(E item); // add an item
  - E remove(); // remove highest priority item
- S should be designed to make these two operations fast

Such structures are called *Priority Queues*

Priority Queues

- Priority queues are used for:
  - Scheduling processes in an operating system
    - Priority is function of time waiting + process priority
  - Order services on server
    - Backup is low priority, so don’t do when high priority tasks need to happen
  - Scheduling future events in a simulation
  - Medical waiting room
  - Huffman codes - order by tree size/weight
  - A variety of graph/network algorithms

Priority Queues

- Name is misleading: They are *not* queues
- Always remove object with *highest priority* regardless of when it was enqueued
- Data can be received/inserted in any order, but it is always returned/removed according to priority
- Like ordered structures (i.e., OrderedVectors and OrderedLists), PQs require comparisons of values

On Terminology

- In colloquial English, the phrases "highest priority" and "number 1 priority" are used interchangably
- So keep in mind that, often

  Higher Priority = Smaller Value
- A PQ removes the *smallest* value in an ordering: that is, the *highest priority* value!

PQ Interface

```
public interface PriorityQueue<E extends Comparable<E>> {
  public E getFirst(); // peeks at minimum element
  public E remove();   // removes minimum element
  public void add(E value); // adds an element
  public boolean isEmpty();
  public int size();
  public void clear();
}
```

Notes on PQ Interface

- Unlike previous structures, we do not extend any other interfaces for many reasons
  - Random access is prohibited
  - Removal of arbitrary values is prohibited
- PriorityQueue uses Comparables
  - methods use Comparable parameters and
  - methods return Comparable values
- Could be made to use Comparators instead…

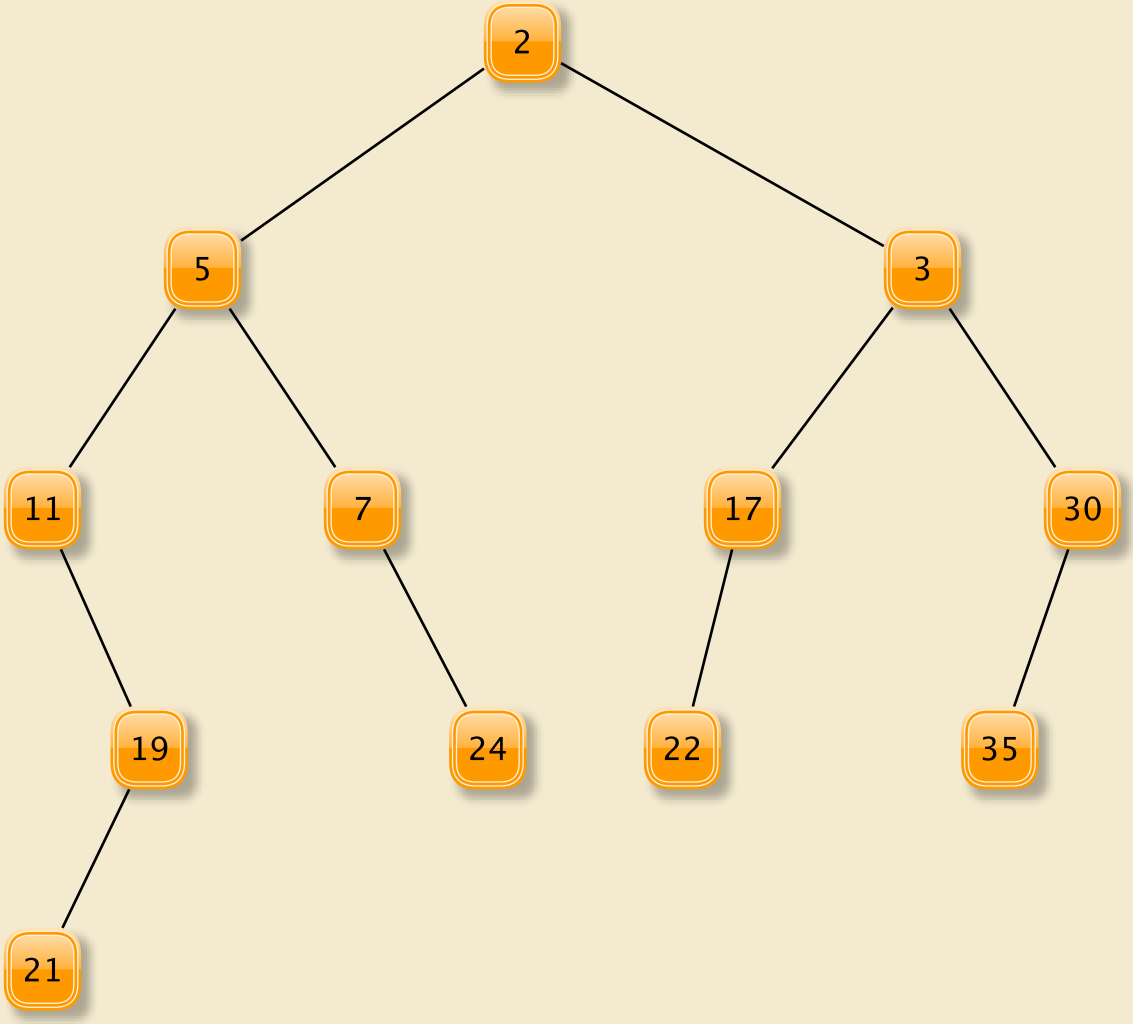
Implementing PQs

- OrderedVector?
  - Keep ordered vector of objects
  - O(n) to add/remove from vector
  - Can we do better than O(n)?
- Binary Search Tree
  - Would need to be balanced for good performance
  - Would get O(log n) which is very good
- Could relaxing requirements of total ordering help
  - Overhead of balancing might be avoided
- Heap!
  - Partially ordered binary tree

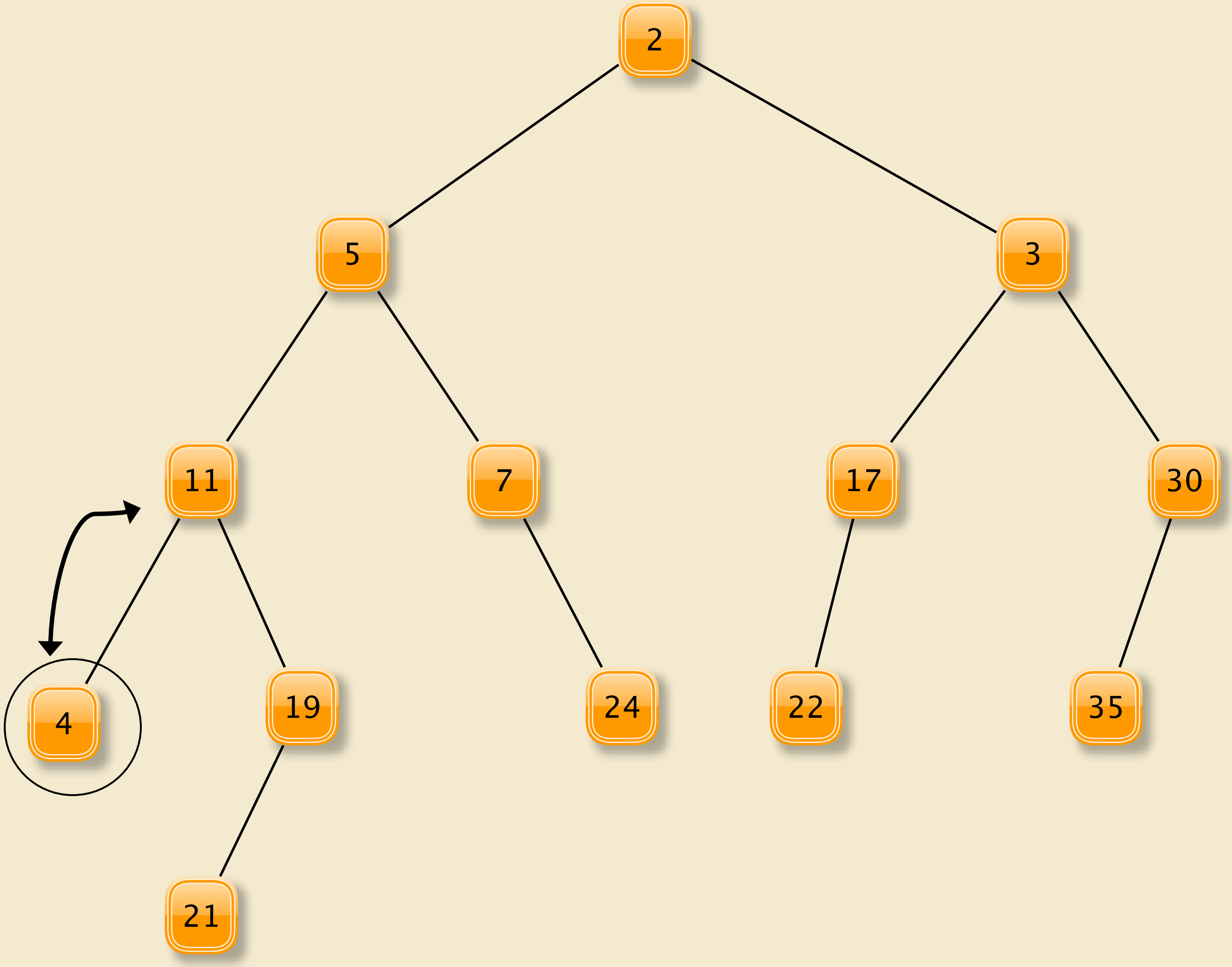
Heap (aka Min-Heap)

- A heap is a special type of binary tree
- A heap is a binary tree where:
  - Root holds smallest (highest priority) value
  - Subtrees are also heaps (this is crucial!)
- So values increase in priority (decrease in value) from leaves to root (from descendant to ancestor)
- *Alternate definition:* A tree is a heap if and only if
  - For all nodes: node.value() >= node.parent.value()
    - This is called the *heap property* or the *heap invariant*
- Several valid heaps for same data set (no unique representation)
  - Note: variants allow more than 2 children per node

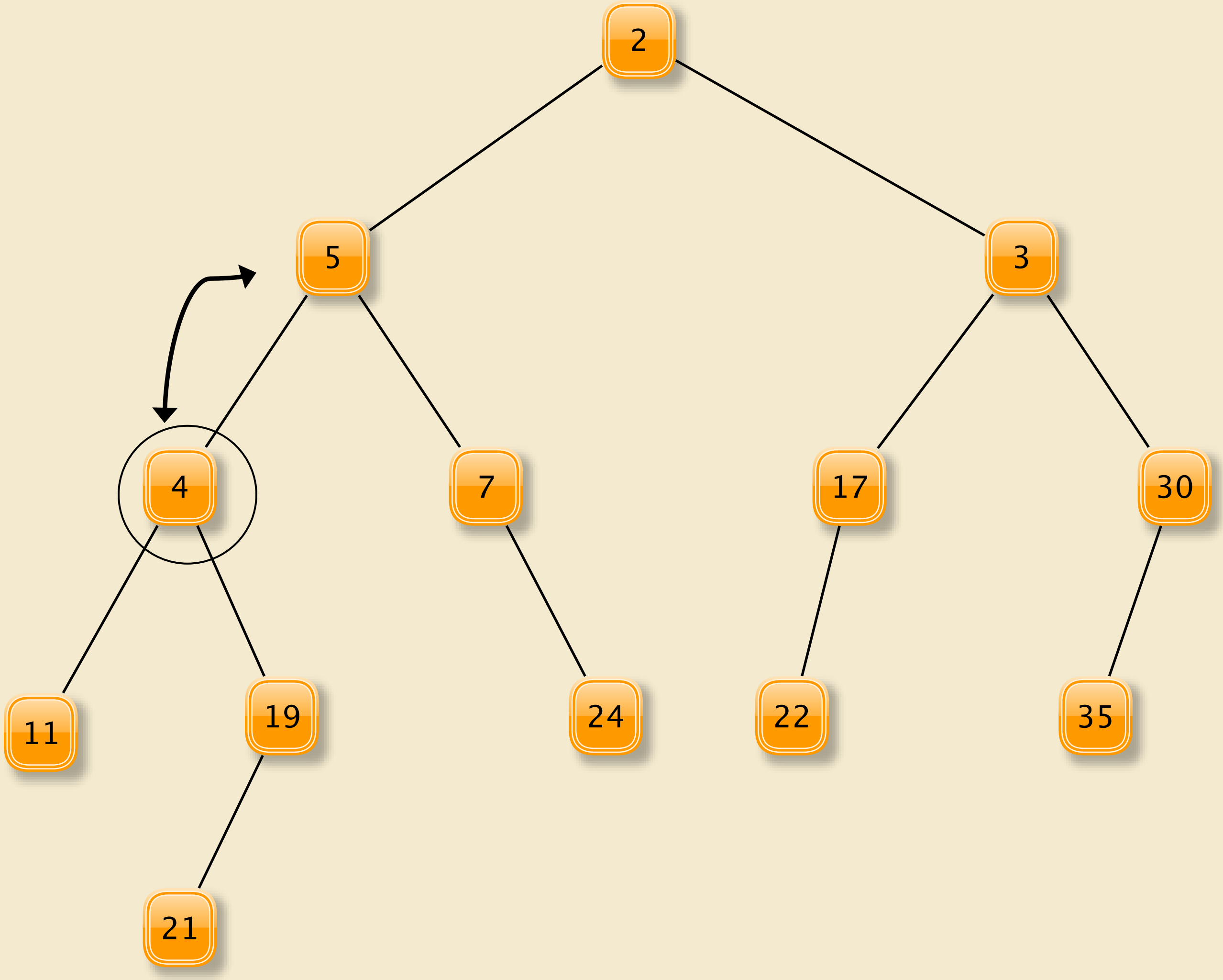
Inserting into a Heap



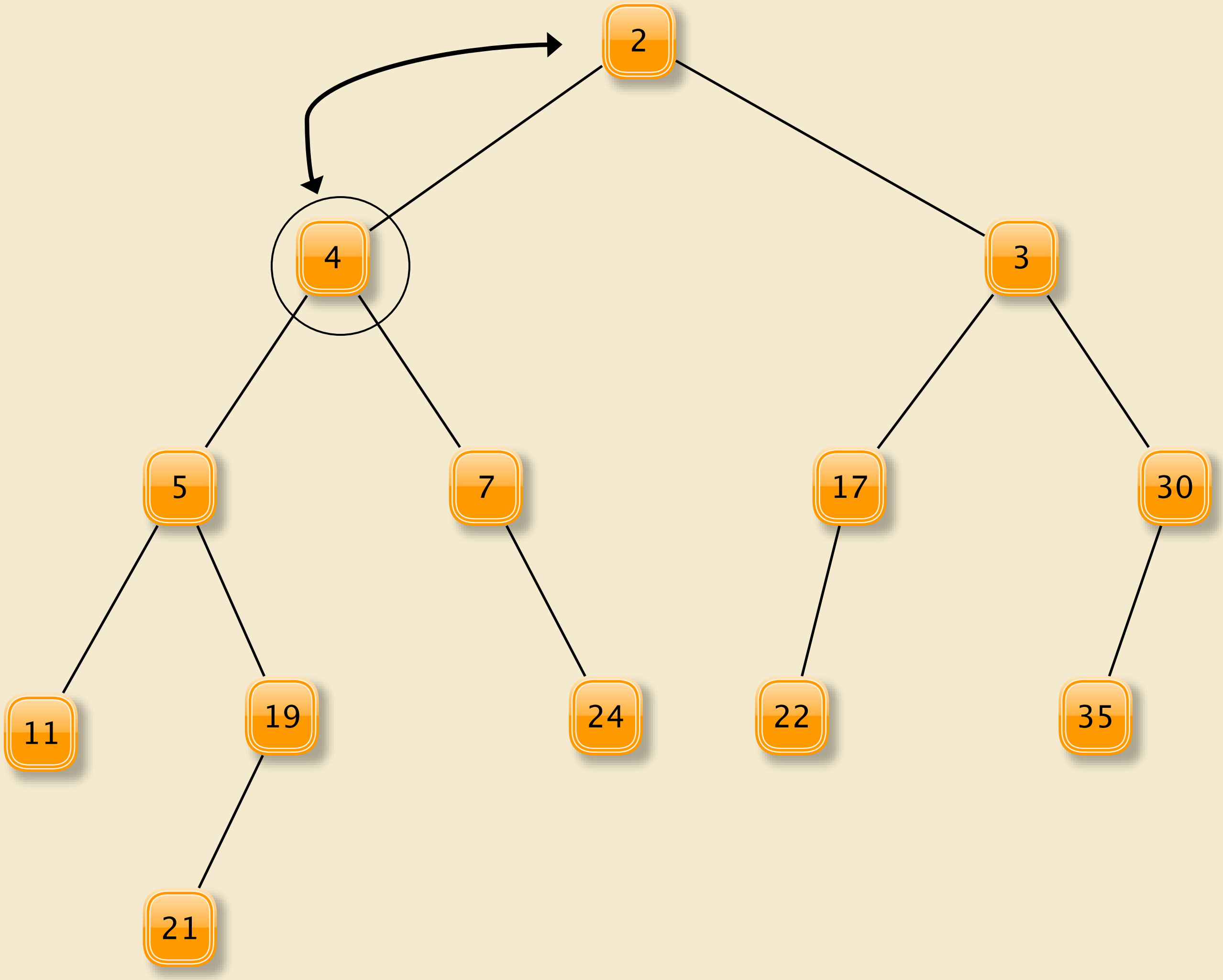
Inserting into a Heap



Inserting into a Heap



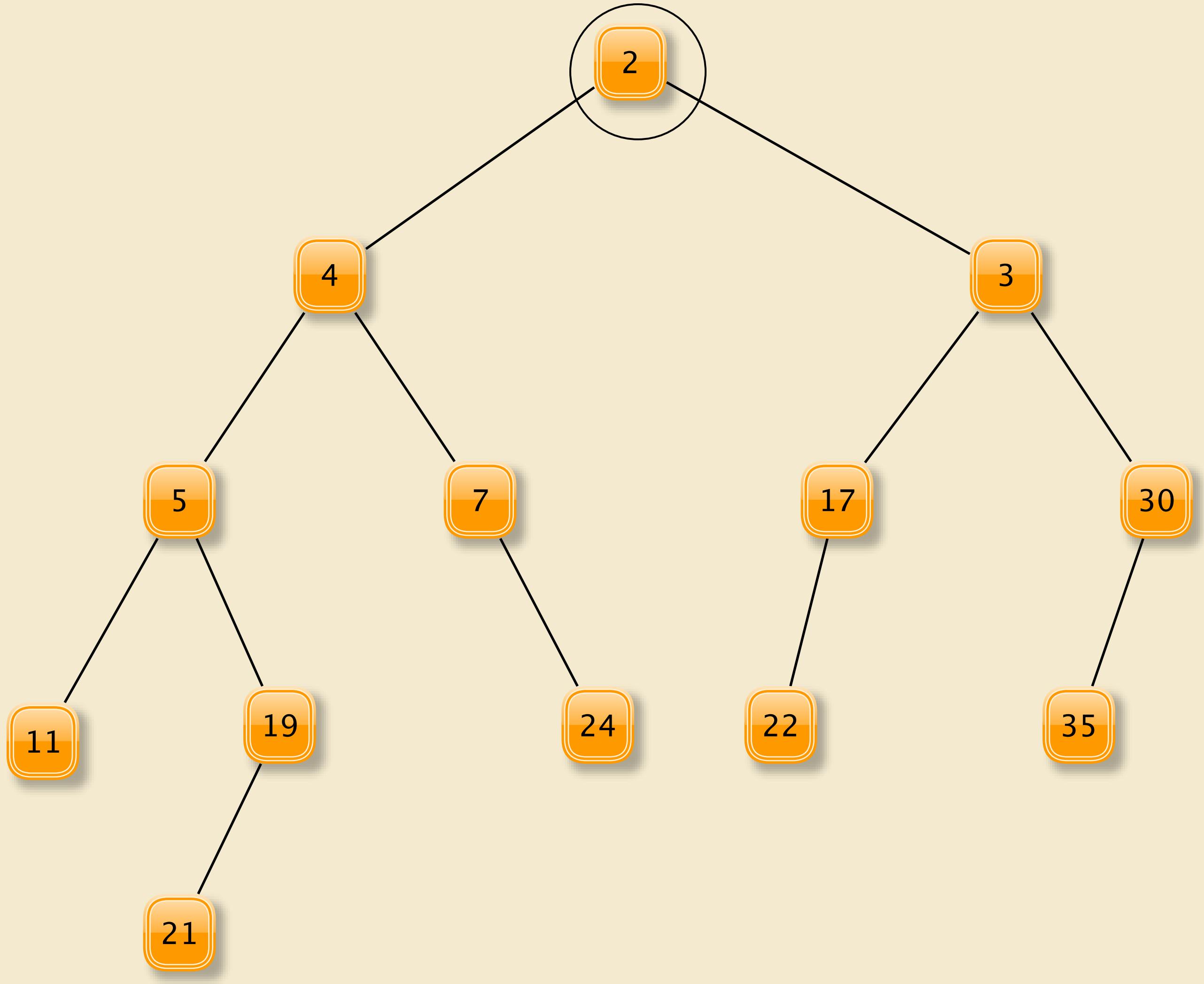
Inserting into a Heap



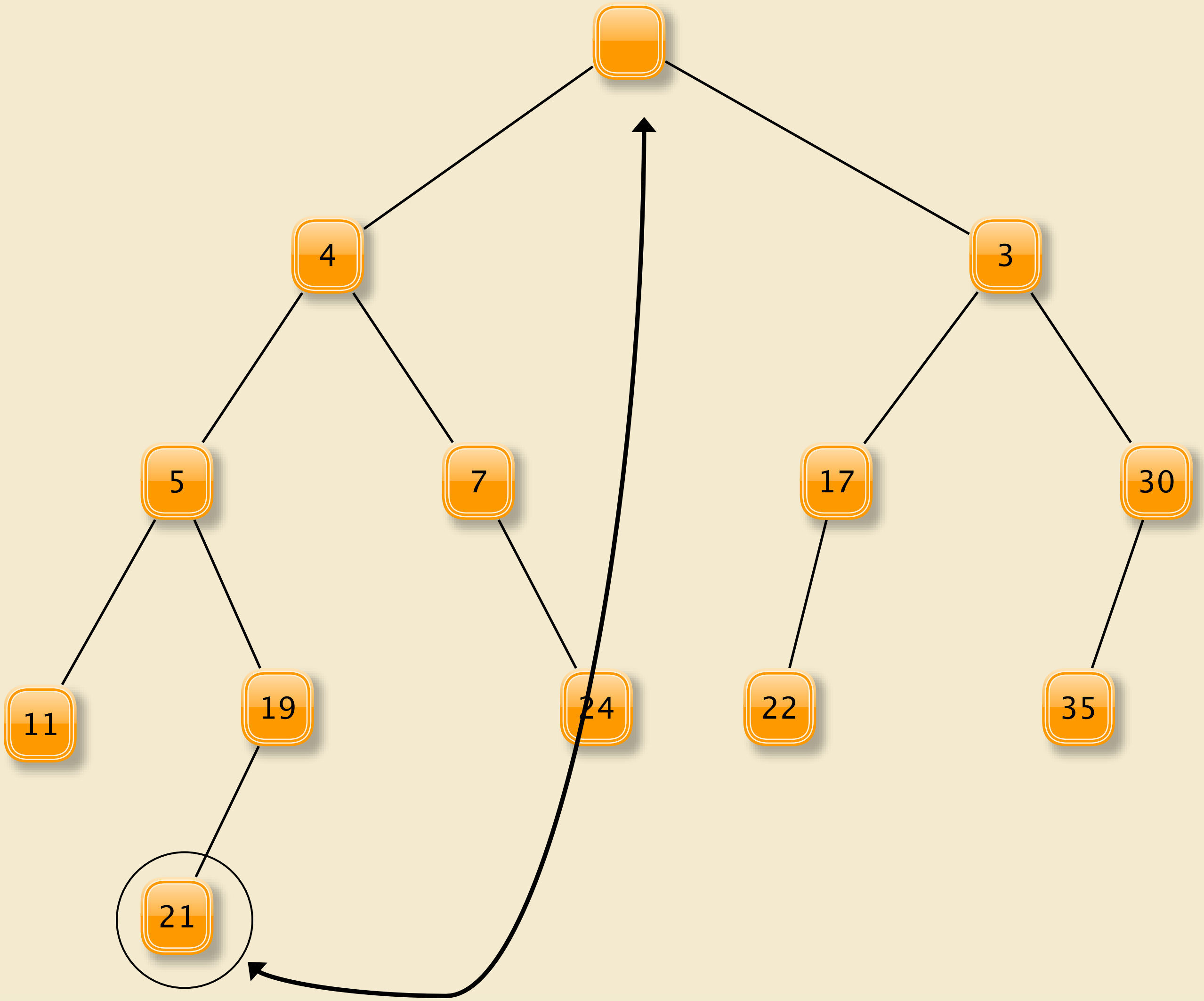
Inserting into a Heap

- Add new value as a leaf
- “Percolate” it up the tree
  - while (value < parent’s value) swap with parent
- This operation preserves the heap property since new value was the only one violating heap property
- Efficiency depends upon speed of
  - Finding a node at which to add new child
  - Finding location of parent
  - Tree height

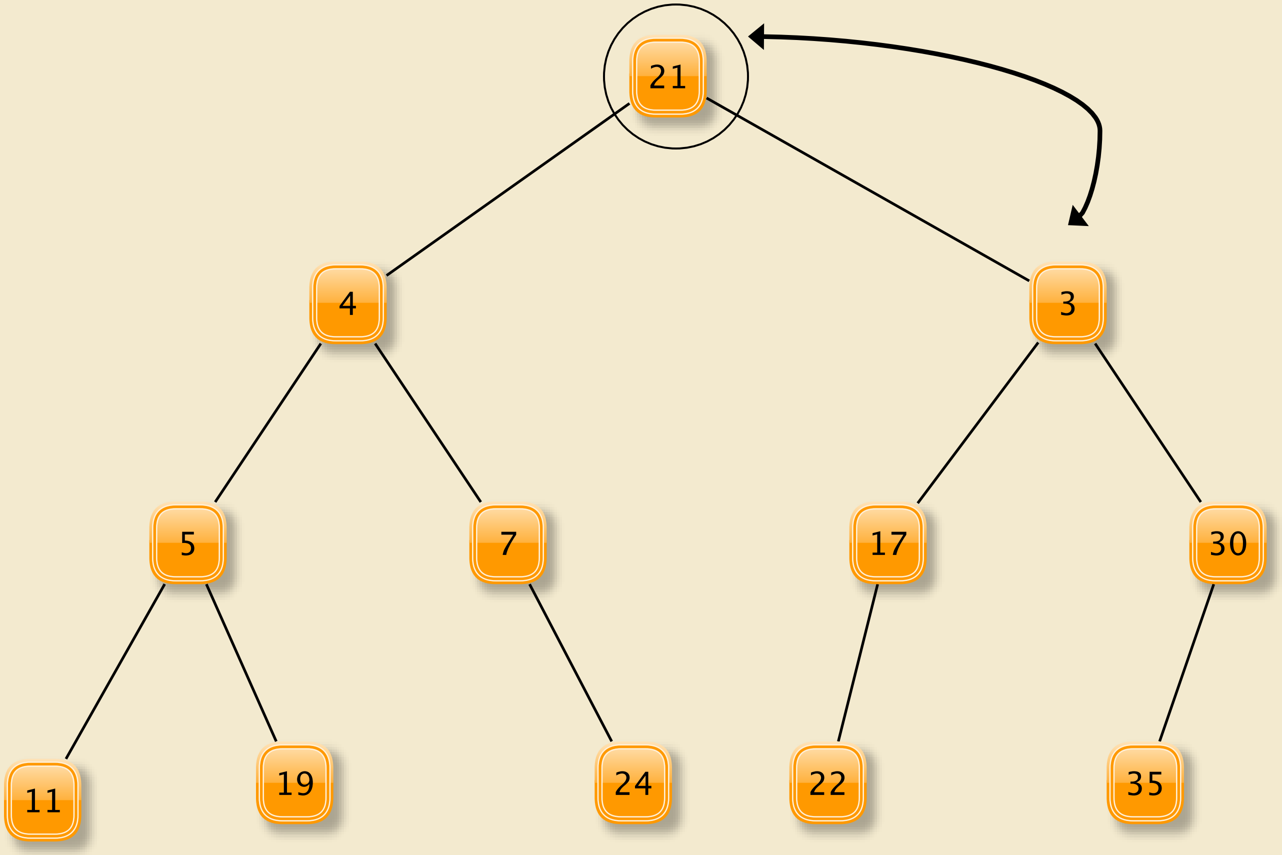
Removing Min From a PQ



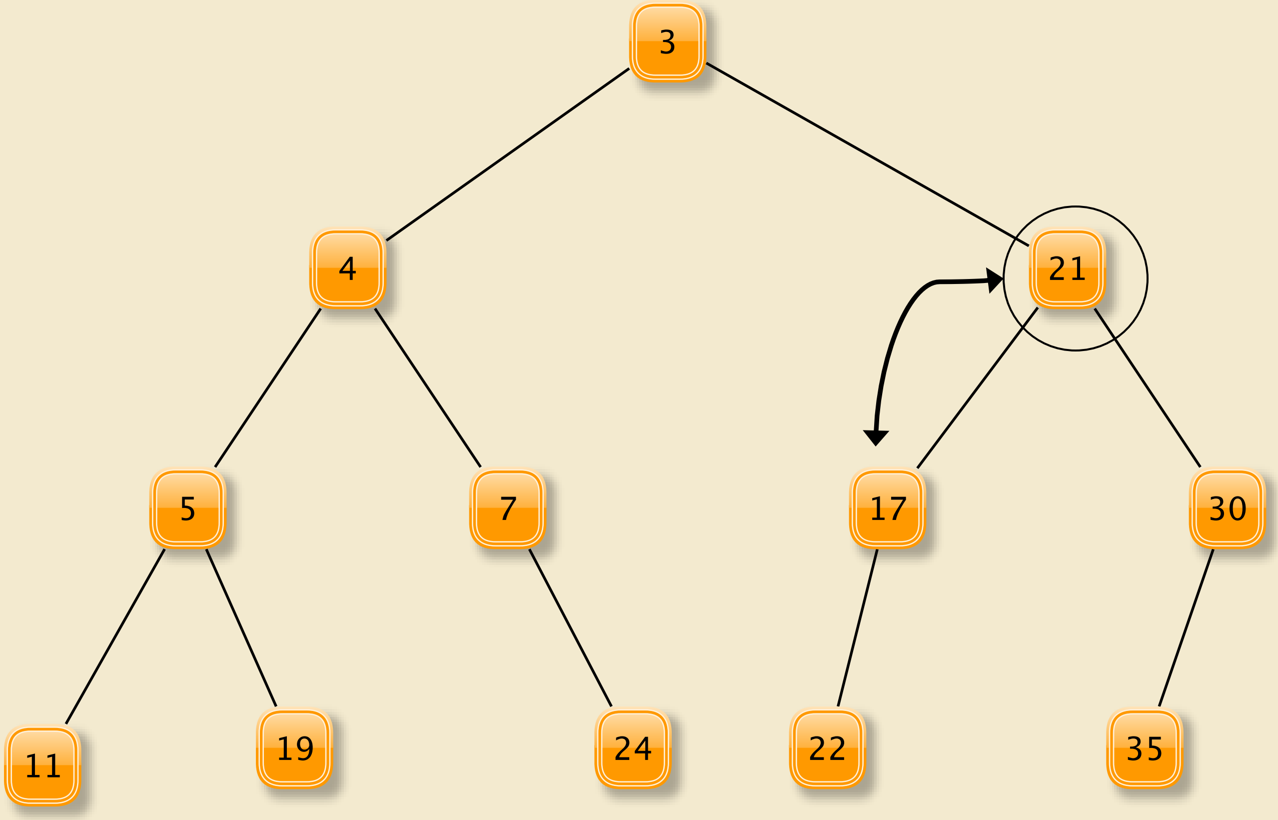
Removing Min From a PQ



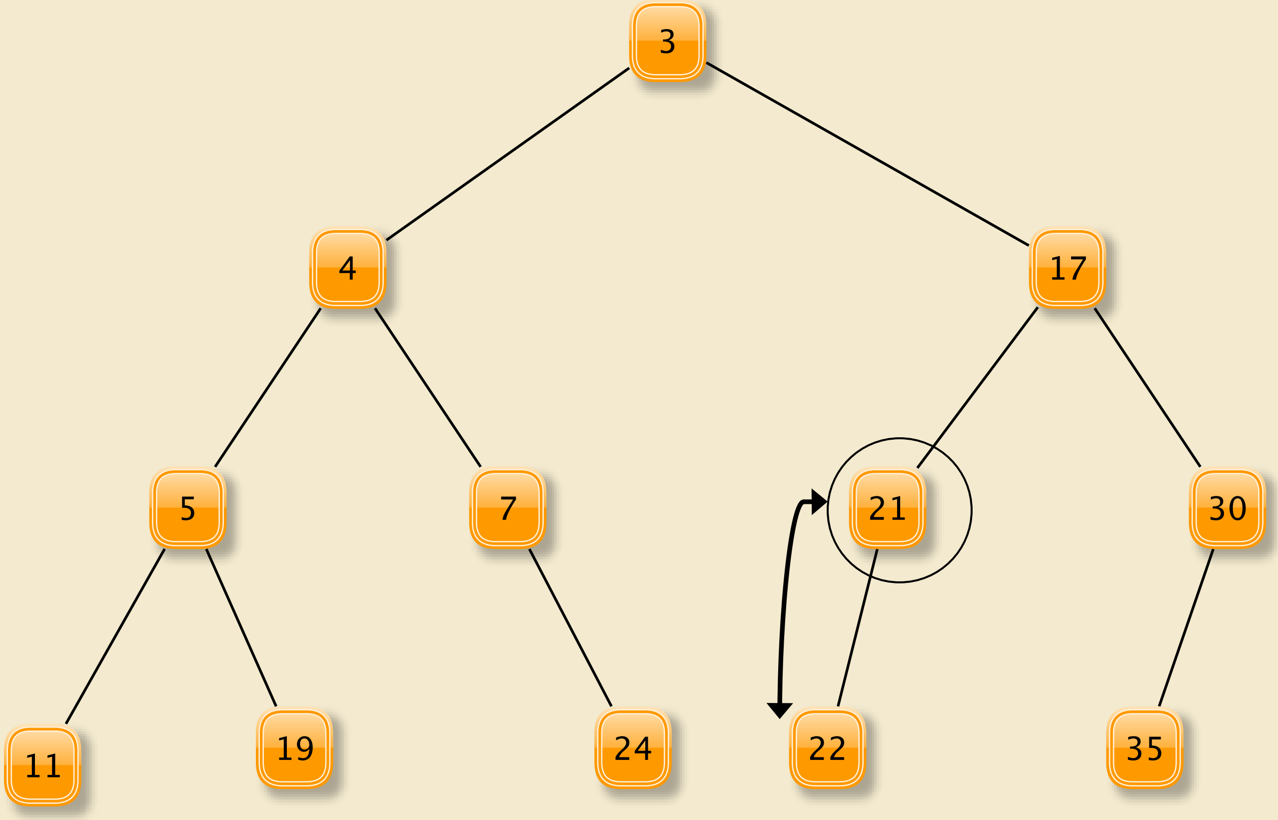
Removing Min From a PQ



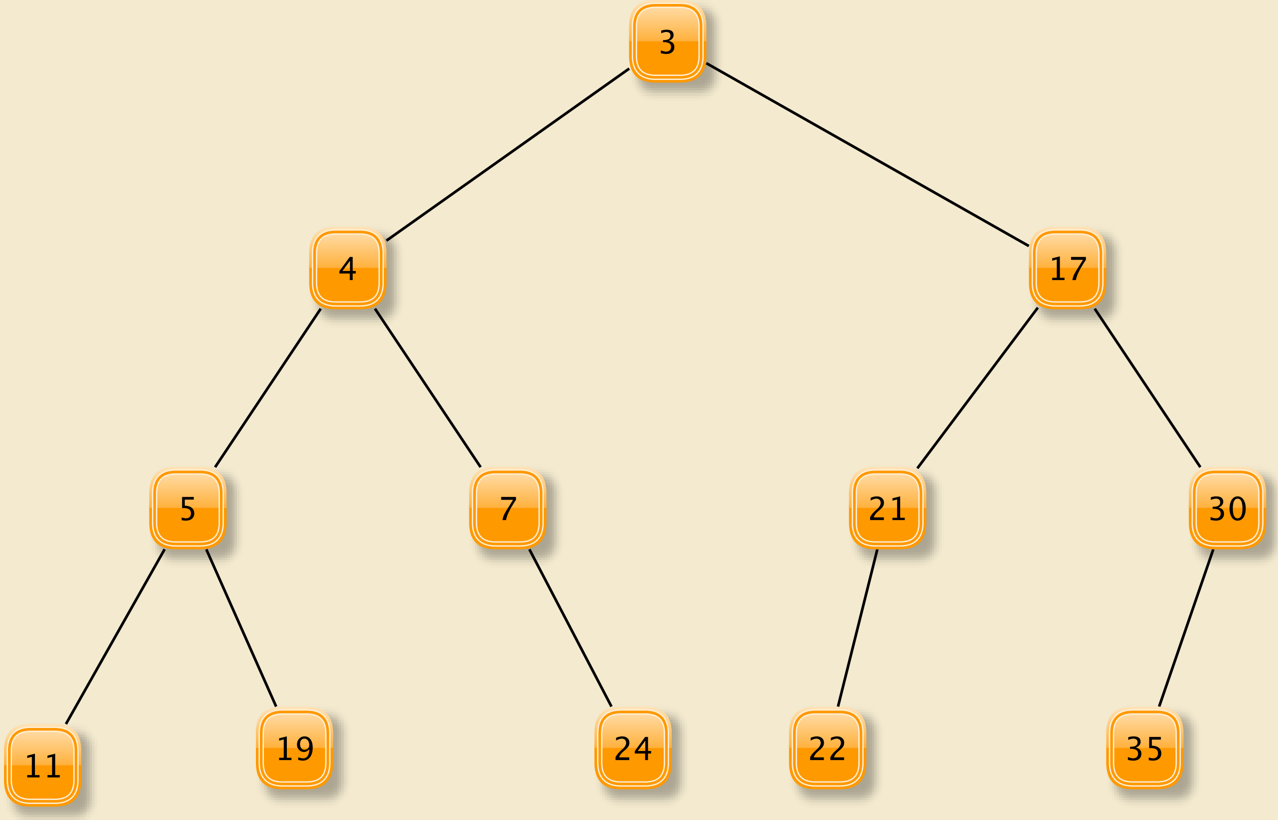
Removing Min From a PQ



Removing Min From a PQ



Removing Min From a PQ



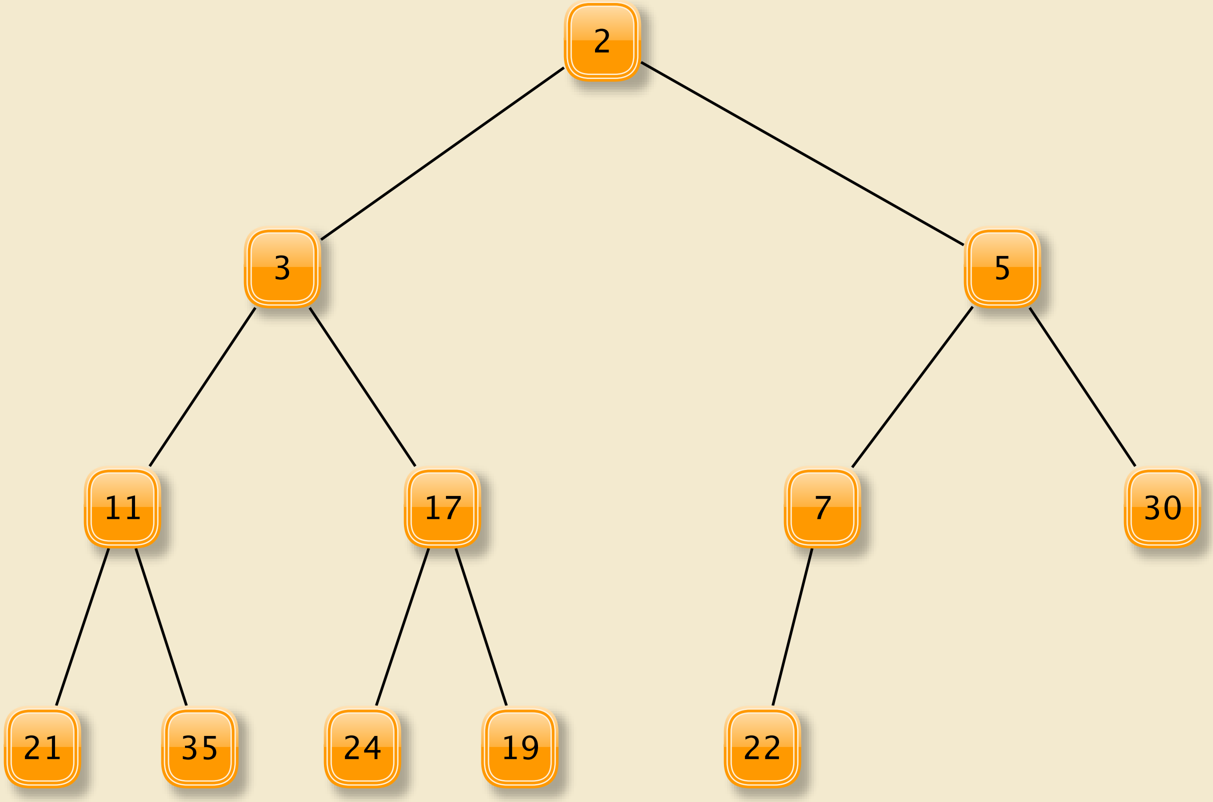
Removing Root From a PQ

- Copy root value, save it to return
- Find a leaf, delete it, put its *data* in the root
- "Push" *data* down through the tree
    - while (*data.value* > value of (at least) one child)
        - Swap *data* with data of *smaller* child
- This operation preserves the heap property
- Efficiency depends upon speed of
    - Finding a leaf
    - Finding locations of children
    - Height of tree

Key Operations/Properties

- Insert efficiency depends upon speed of
  - Finding a node at which to add new child
  - Finding location of parent
  - Tree height
- RemoveMin efficiency depends upon speed of
  - Finding a leaf
  - Finding locations of children
  - Tree Height
- Goal: Find tree structure to optimize these

Array-Based Binary Trees



2	3	5	11	17	7	30	21	35	24	19	22
0	1	2	3	4	5	6	7	8	9	10	11

Array-Based Binary Trees

- Encode structure of tree in array indexes
  - Put root at index 0
  - Leave empty slots if no child
- Where are children of node i?
  - Children of node i are at 2i+1 and 2i+2
  - Look at example
- Where is parent of node j?
  - Parent of node j is at (j-1)/2

Recall : ArrayTrees

- Why are ArrayTrees good?
  - Save space for links
  - No need for additional memory allocated/garbage collected
  - Works well for *full* or *complete* trees
    - Complete: All levels except last are full and all gaps are at right
    - "A *complete* binary tree of height h is a full binary tree with 0 or more of the rightmost leaves of level h removed"
    - No empty slots!
- Why is this an OK assumption for us? Most trees are not complete…
- Insight: We can guarantee that our heap is always a complete tree by smart add/remove choices!

Implementing Heaps

- VectorHeap
    - Use conceptual array representation of BT (ArrayTree)
    - But use extensible Vector instead of array (makes adding elements easier)
    - Note:
        - Root of tree is location 0 of Vector
        - Children of node in location i are in locations 2i+1 (left) and 2i+2 (right)
        - Parent of node i is in location (i-1)/2

Implementing Heaps

- Features
  - Guarantee no gaps in array (array is *complete*)
    - Always add in next available array slot (left-most available spot in binary tree;
    - Always remove using "right-most" leaf
  - *Heap Invariant* becomes
    - data[i] <= data[2i+1]; data[i]<=data[2i+2] (or kids might be null)
  - When elements are added and removed, do small amount of work to "re-heapify"
    - How small? Note: finding a node's child or parent takes constant time, as does finding "final" leaf or next slot for adding
    - Since this heap corresponds to a full binary tree, the depth of the tree is O(log n), so add/remove take O(log n) time!

Implementing Heaps

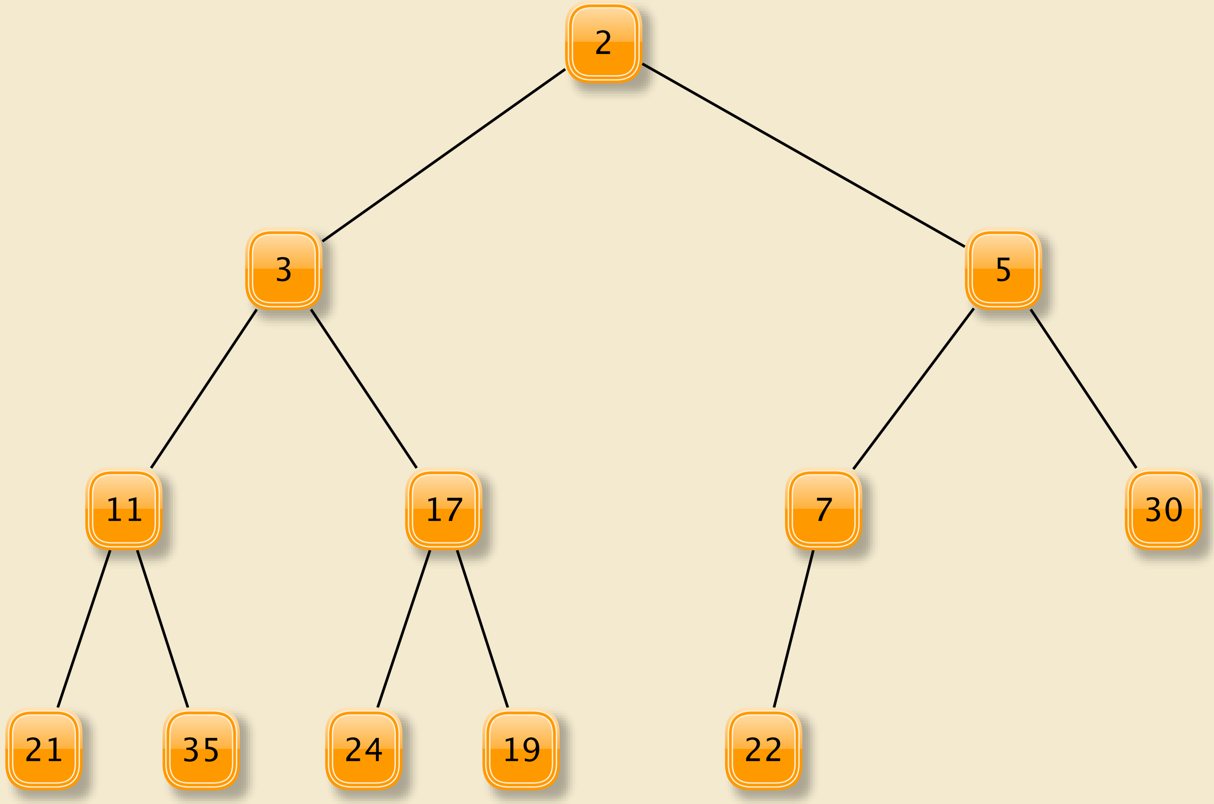
Details

- Add method uses helper percolateUp(int location)
  - percolateUp moves newly inserted value up the tree until heap property is restored

- Remove method uses helper pushDownRoot(int root)
  - Moves value that remove moved from deleted leaf to root down the tree until heap property is restored

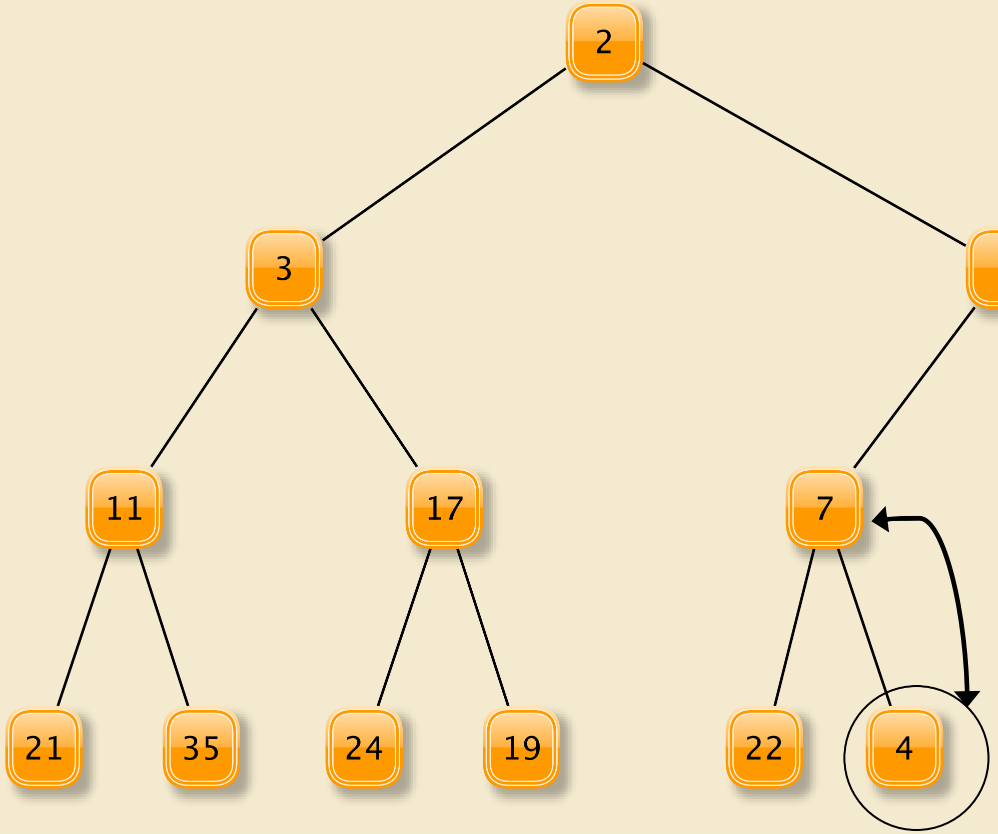
- Let's look some examples

Example : Add(4)



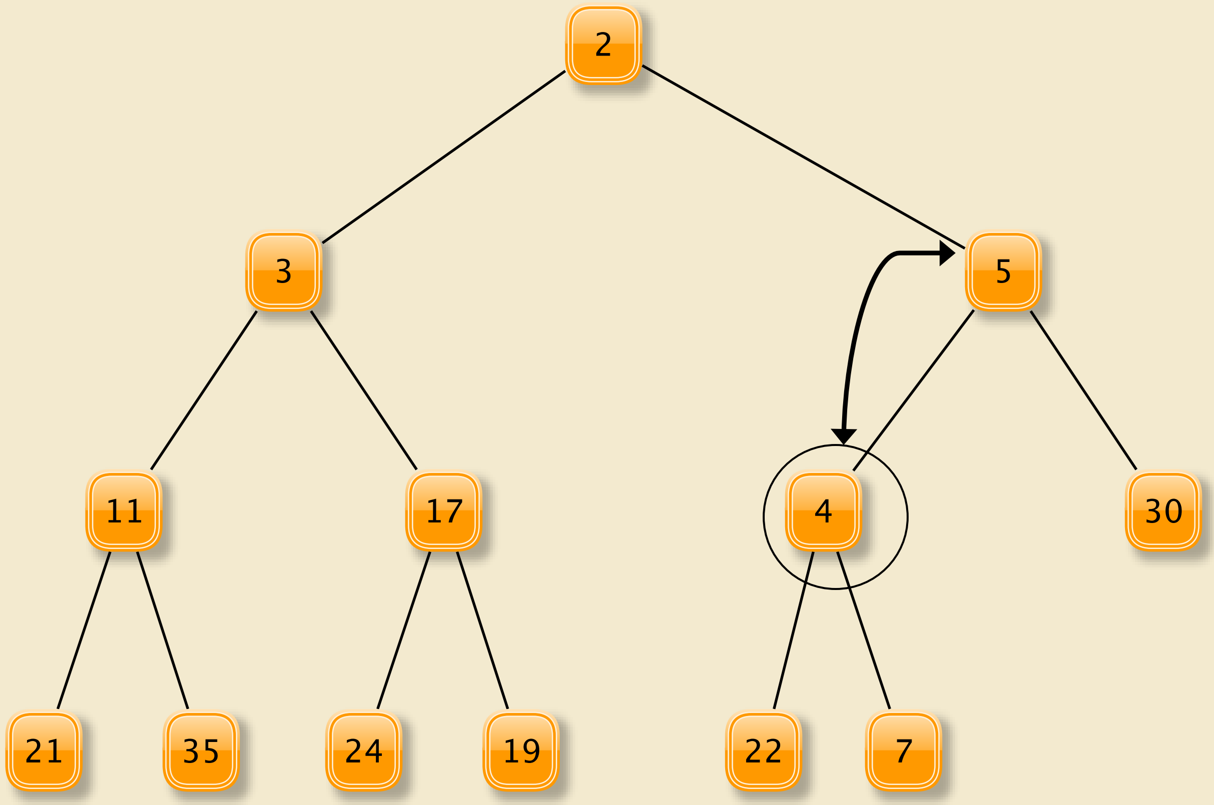
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	11	17	7	30	21	35	24	19	22	-	-	-

Example : Add(4)



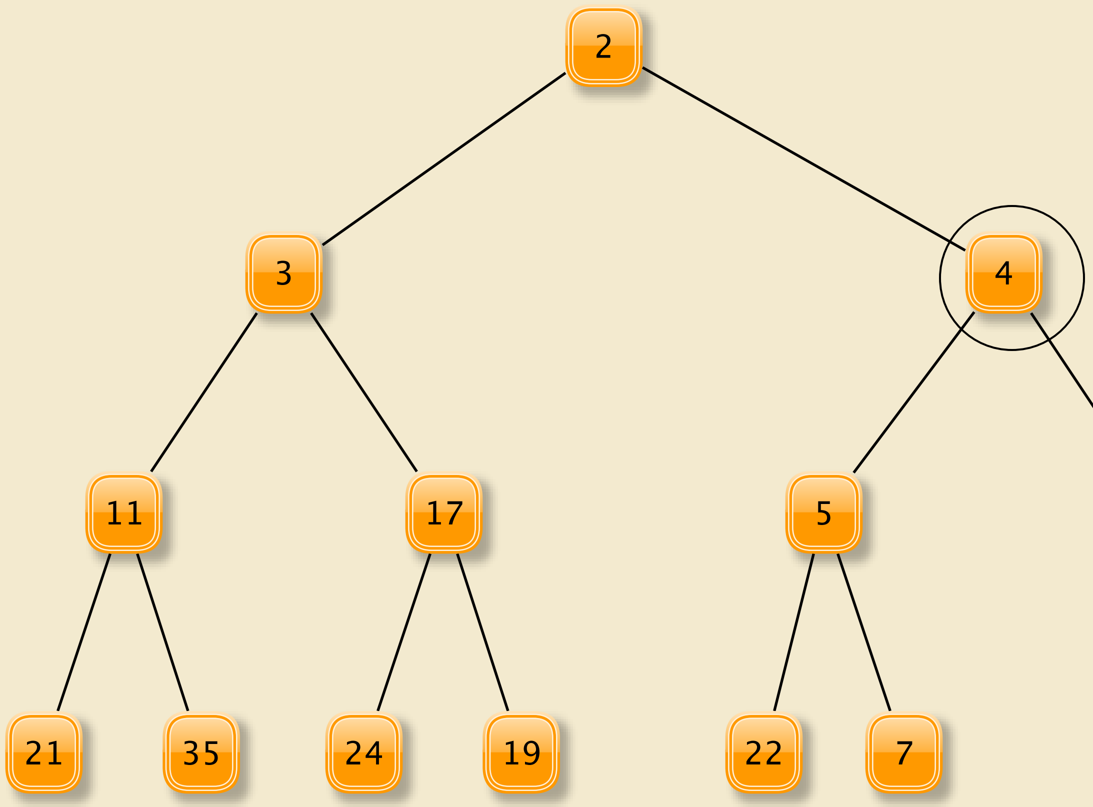
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	11	17	7	30	21	35	24	19	22	4	-	-

Example : Add(4)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	11	17	4	30	21	35	24	19	22	7	-	-

Example : Add(4)

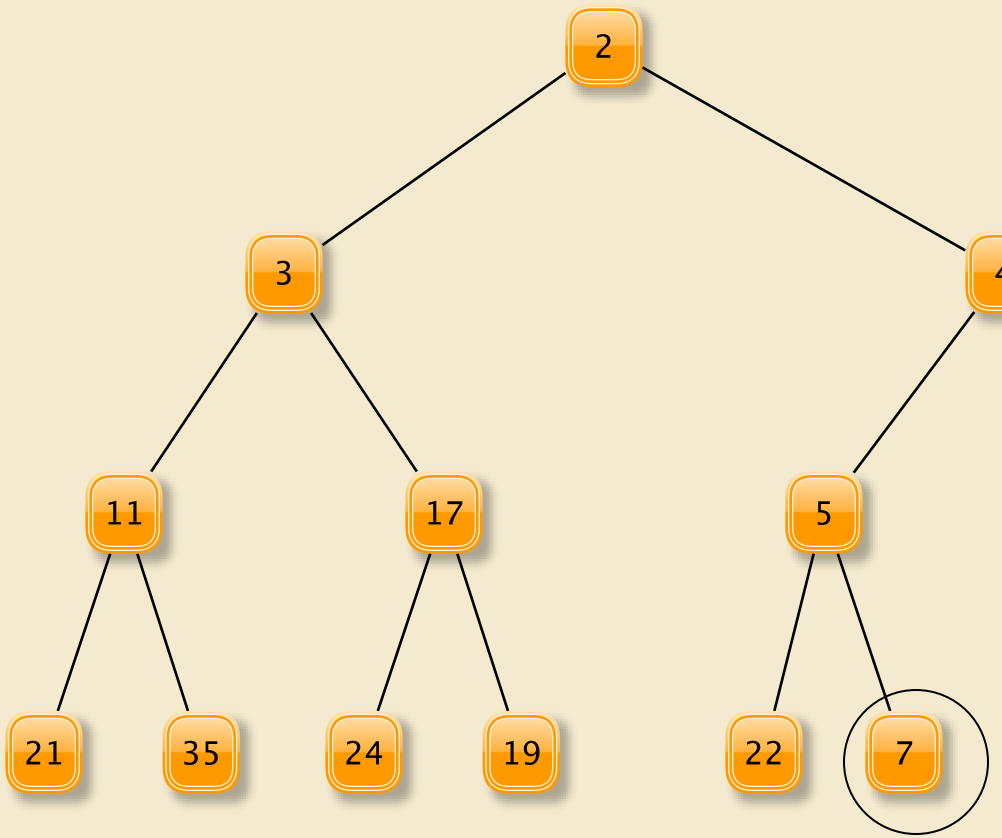


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	4	11	17	5	30	21	35	24	19	22	7	-	-

Add : Uses PercolateUp

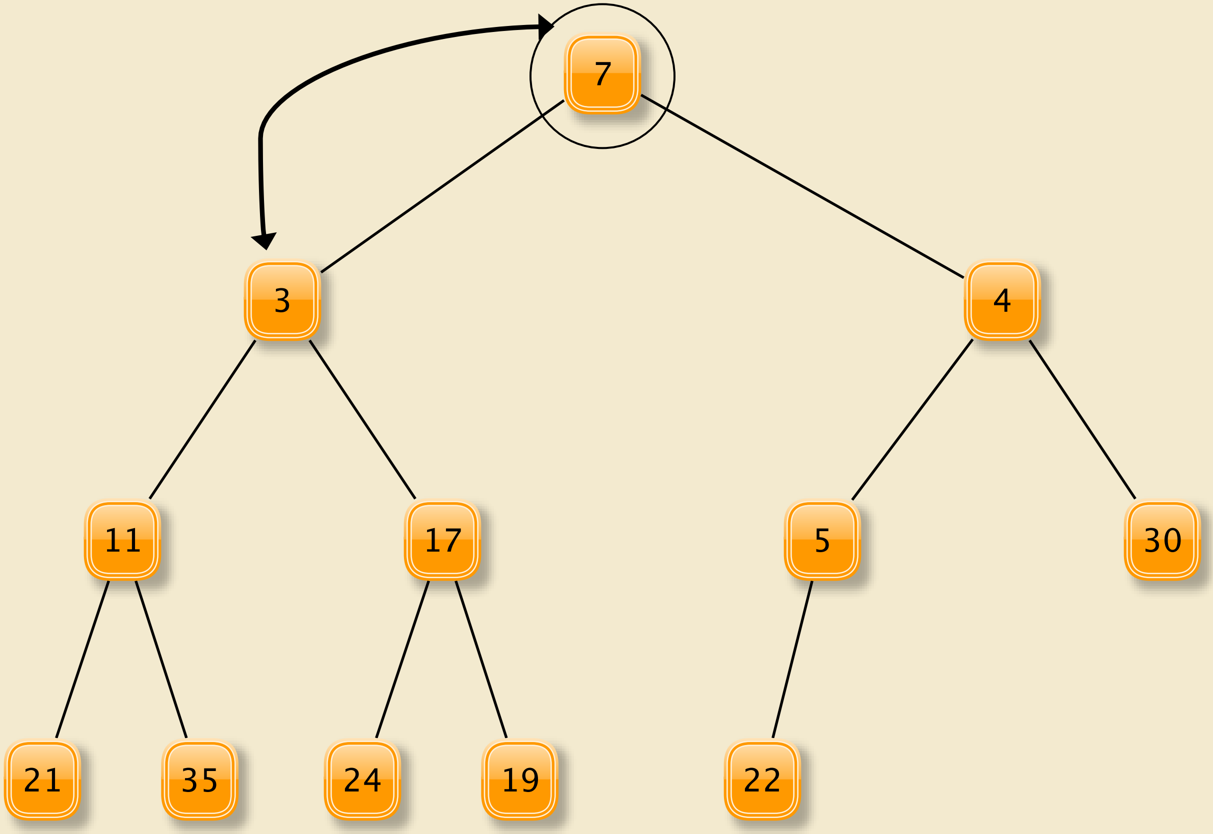
```
protected void percolateUp(int leaf) {
  int parent = parent(leaf);
  E value = data.get(leaf);
  while (leaf > 0 &&
     (value.compareTo(data.get(parent)) < 0)){

    data.set(leaf,data.get(parent));
    leaf = parent;
    parent = parent(leaf);
  }
  data.set(leaf,value);
}
```

Example : Remove()



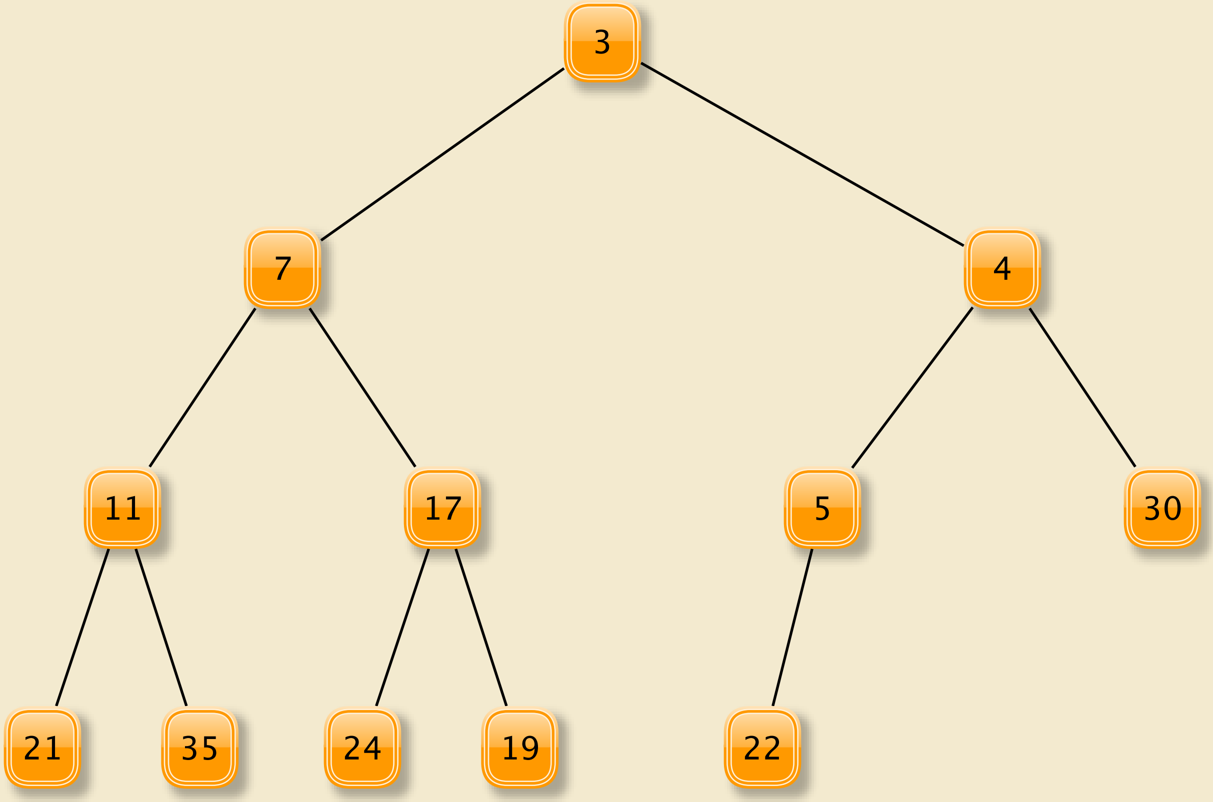
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	4	11	17	5	30	21	35	24	19	22	7	-	-

Example : Remove()



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	3	4	11	17	5	30	21	35	24	19	22	-	-	-

Example : Remove()



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	7	4	11	17	5	30	21	35	24	19	22	-	-	-

Remove : Uses PushDownRoot

```
protected void pushDownRoot(int root) {
  int heapSize = data.size();
  E value = data.get(root);
  while (root < heapSize) {
    int childpos = left(root);
    // If node has left child
    if (childpos < heapSize) {
      // If right child has smaller value
      if ((right(root) < heapSize) &&
               ((data.get(childpos+1)).compareTo
                 (data.get(childpos)) < 0)) {
        childpos++;
      }
```

Remove : Uses PushDownRoot

```
  // Assert: childpos indexes smaller child
  // Compare child to value being pushed down
  if((data.get(childpos)).compareTo(value)<0){
    data.set(root,data.get(childpos));
    root = childpos; // keep moving down
  } else { // found right location
      data.set(root,value);
      return;
  }
} else { // at a leaf! insert and halt
    data.set(root,value);
    return;
} } }
```

VectorHeap Summary

- Add/Remove are both O(log n)
- Data is not completely sorted
  - “Partial” order is maintained
- Note: VectorHeap(Vector<E> v)
  - Takes an unordered Vector and uses it to construct a heap
  - How
    - Uses VectorHeap add method to insert elements of v
    - This builds the VectorHeap in O(n log n) time
    - As always, we ask: Can we do better?

Heapifying A Vector (or array)

- Method I: Top-Down
  - Assume V[0...k] satisfies the heap property
  - Now call percolate on item in location k+1
  - Then V[0..k+1] satisfies the heap property
- Method II: Bottom-up
  - Assume V[k..n] satisfies the heap property
  - Now call pushDown on item in location k-1
  - Then V[k-1..n] satisfies heap property

Top-Down vs Bottom-Up

- Top-down heapify: elements at depth d may be swapped d times: Total # of swaps is at most

$$\sum_{d=0}^{h} d2^d = (h-1)2^{h+1} + 2 = (\log n - 1)2n + 2$$

- This is O(n log n)
- Some intuition: most of the elements are in the lowest levels of the tree, so each of them might have to move to root: O(log n) swaps per element

Top-Down vs Bottom-Up

- Bottom-up heapify: elements at depth d may be swapped h-d times: Total # of swaps is at most

$$\sum_{d=0}^{h} (h-d)2^d = 2^{h+1} - h - 2 = 2n - \log n + 2$$

  - This is O(n) --- beats top-down!
  - Some intuition: most of the elements are in the lowest levels of the tree, so each of them will only be pushed down (swapped) a small number of times SO COOL!!!

HeapSort

- Heaps yield another O(n log n) sort method
- To HeapSort a Vector "in place"
  - Perform bottom-up heapify on the reverse ordering: that is: highest rank/lowest priority elements are near the root (low end of Vector)
  - Now repeatedly remove elements to fill in Vector from tail to head
    - For(int i = v.size() – 1; i > 0; i--)
      - RemoveMin from v[0..i]   // v[i] is now not in heap
      - Put removed value in location v[i]