CSE 332: Data Structures

Priority Queues – Binary Heaps

Richard Anderson, Steve Seitz

Winter 2014

Administrative

- P1A due tonight (Monday) by 11:59pm
  - via catalyst
- HW1 due beginning of class Wednesday
- Reading for this week: Chapter 6.1-6.5

Recall Queues

- FIFO: First-In, First-Out
  - Print jobs
  - File serving
  - Phone calls and operators
  - Lines at the Department of Licensing…

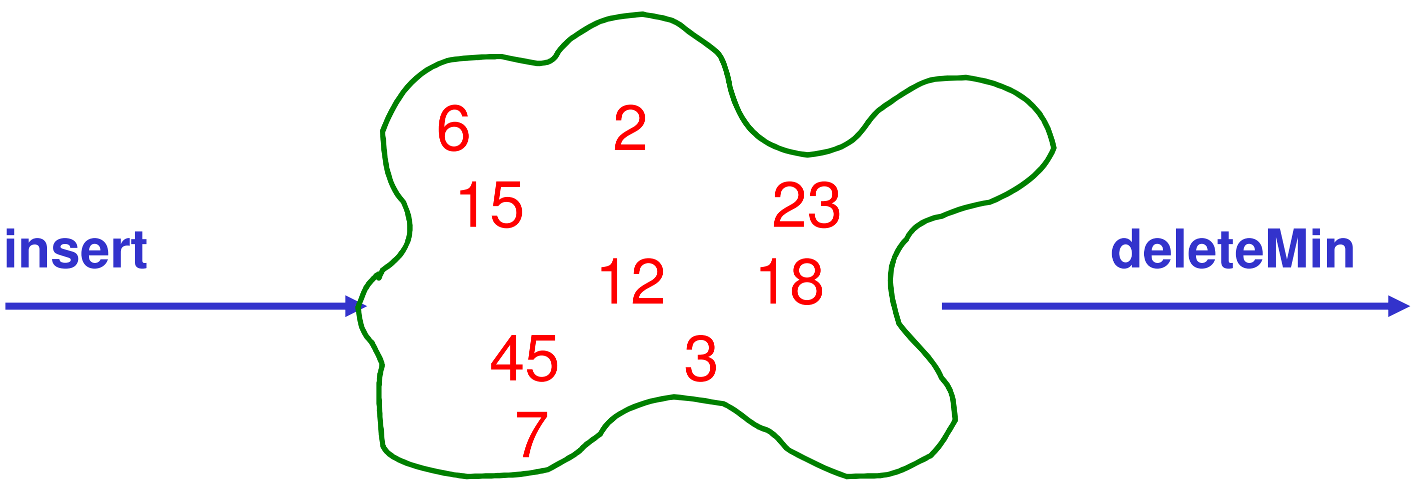
Priority Queues

Prioritize who goes first – a **priority queue**:

- treat ER patients in order of severity
- route network packets in order of urgency
- operating system can favor jobs of shorter duration or those tagged as having higher importance
- Greedy optimization: "best first" problem solving

Priority Queue ADT

- Need a new ADT
- Operations: Insert an Item,
  Remove the “Best” Item



Priority Queue ADT

1. **PQueue <u>data</u>** : collection of data with priority

2. **PQueue <u>operations</u>**
   – insert
   – deleteMin
   (also: create, destroy, is_empty)

3. **PQueue <u>property</u>**: if x has **<u>lower</u>** priority than y, x will be deleted before y

Potential implementations

	insert	deleteMin
Unsorted list (Array)	O(1) or O(n) if full	O(n)
Unsorted list (Linked-List)	O(1)	O(n)
Sorted list (Array)	O(log n) to find O(n) to shift ⇒ O(n)	O(1) to find no shift if reversed
Sorted list (Linked-List)	~~O(log n)~~ O(n)	O(1)
Binary Search Tree (BST)	O(n)	~~O(log n)~~ O(n)

B

Binary Heap data structure

- **binary heap** (a kind of binary tree) for priority queues:
  - O(log n) worst case for both insert and deleteMin
  - O(1) average insert
- It's optimized for priority queues. Lousy for other types of operations (e.g., searching, sorting)

Tree Review

root(**T**): A

leaves(**T**): D-F, I-N

children(B): D-F

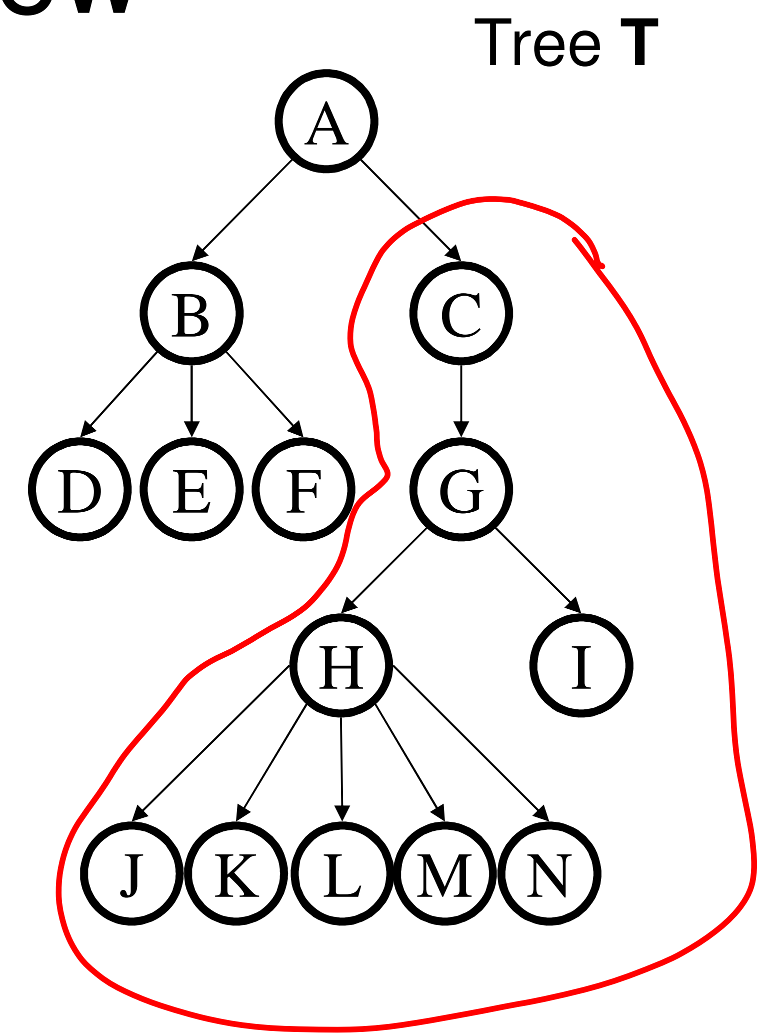
parent(H): G

siblings(E): D,F

ancestors(F): B, A

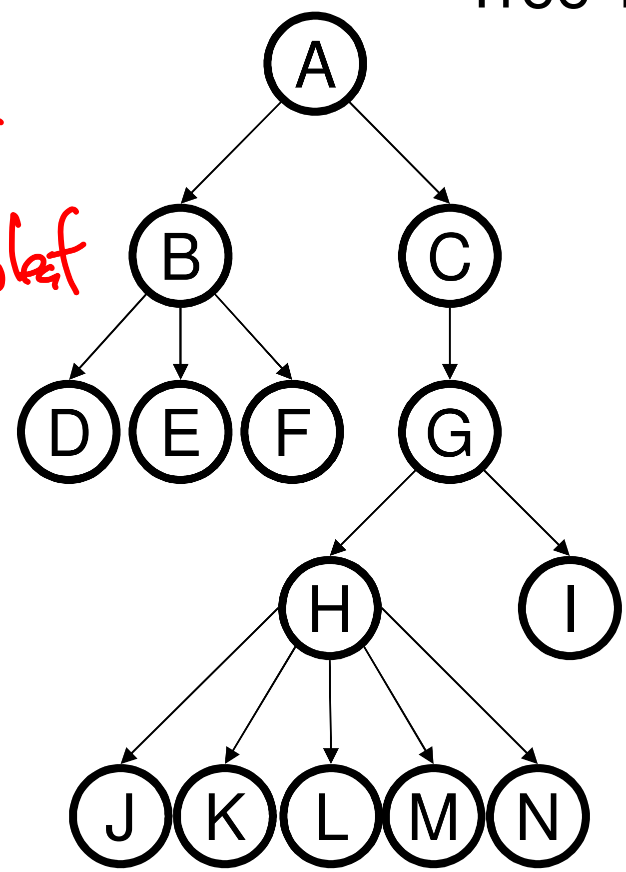
descendents(G): H-N

subtree(C):



More Tree Terminology

Tree **T**



depth(B): 1
#edges on path from root to self

height(G): 2
#edges on the longest path from self to leaf

height(**T**): 4
=height(A)

degree(B): 3
#children

branching factor(**T**): 5
max degree of any node

n-ary tree:
tree with branching factor n

Binary Heap Properties

A binary heap is a binary tree with two important properties that make it a good choice for priority queues:

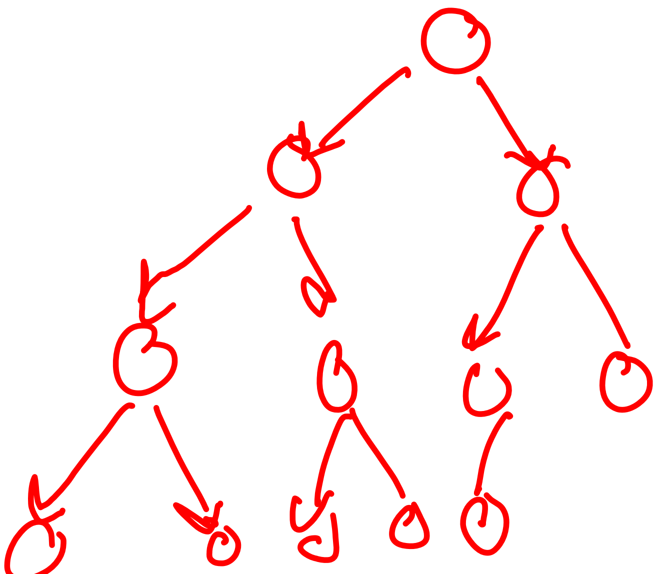
1. **Completeness** (Structure Prop)
2. **Heap Order**

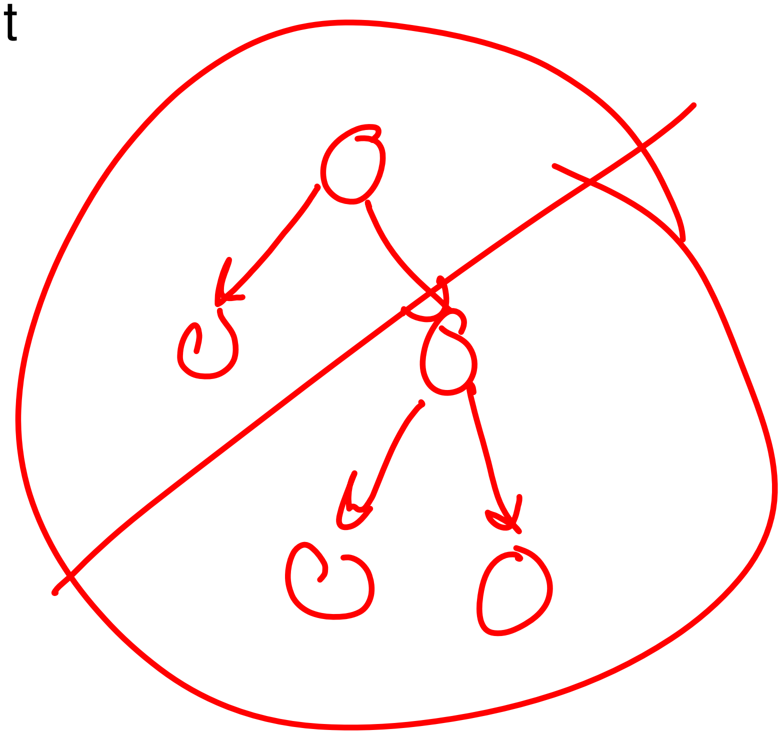
Note: we will sometimes refer to a binary heap as simply a “heap”.

Completeness Property

- A binary heap is a ***complete*** binary tree:
  - a binary tree with all levels full, except possibly the bottom level, which is filled left to right

Examples:



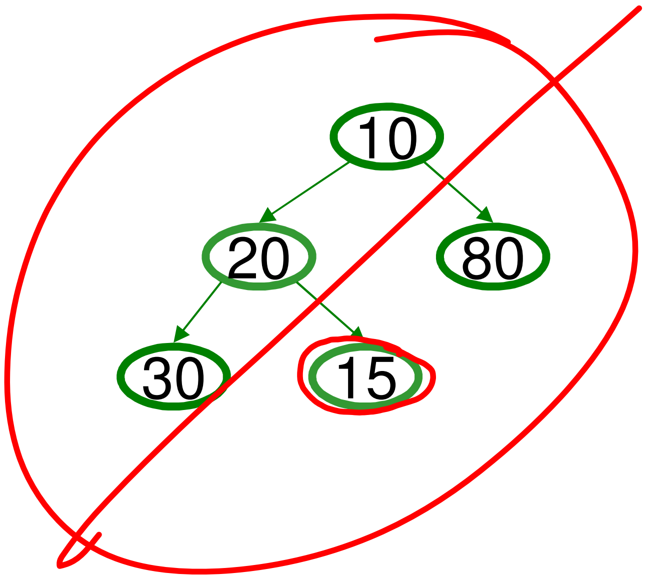


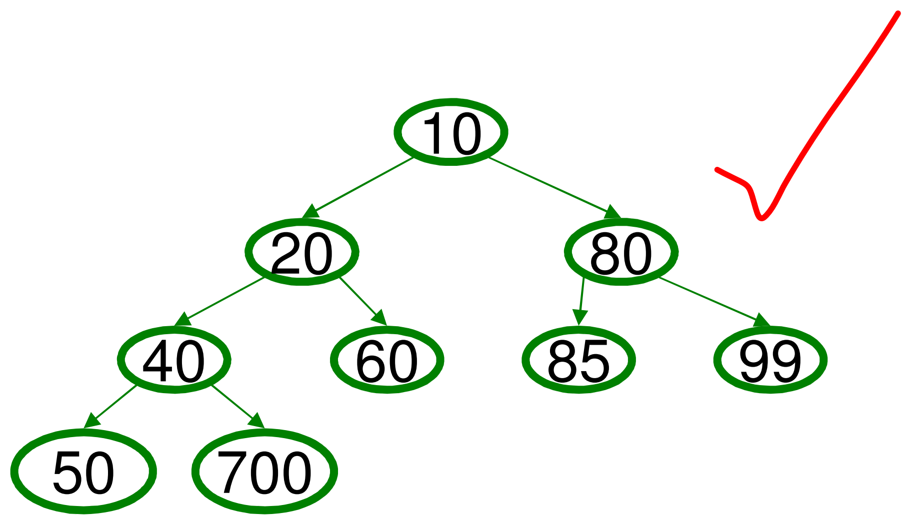
Height of a **complete** binary tree with n nodes?

$\lceil \log_2 n \rceil$

Heap **Order** Property

Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.





which of these is a heap?

Heap Operations

- Main ops: insert, deleteMin
- Key is to maintain
  - Completeness
  - Heap Order

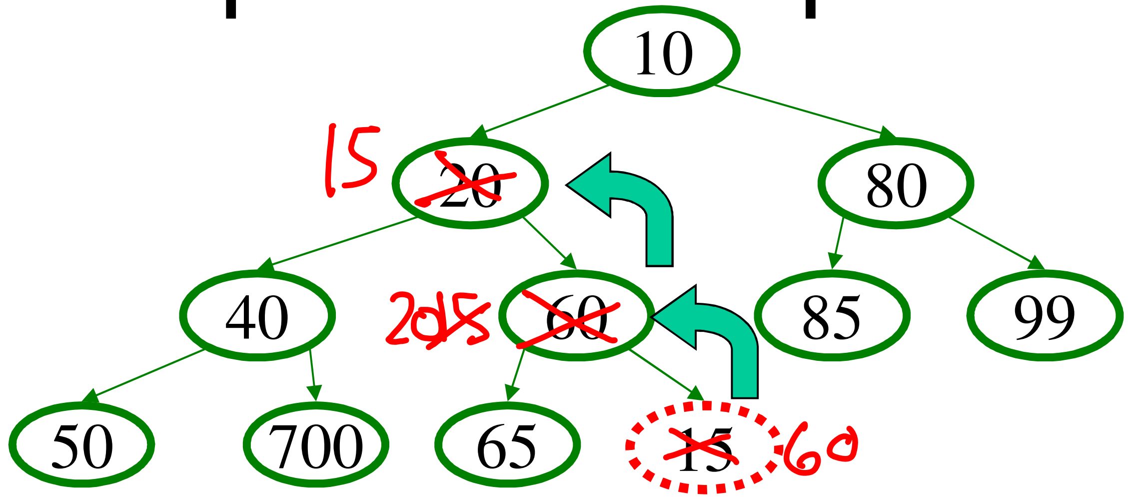
- Basic idea is to propagate changes up/down the tree, fixing order as we go

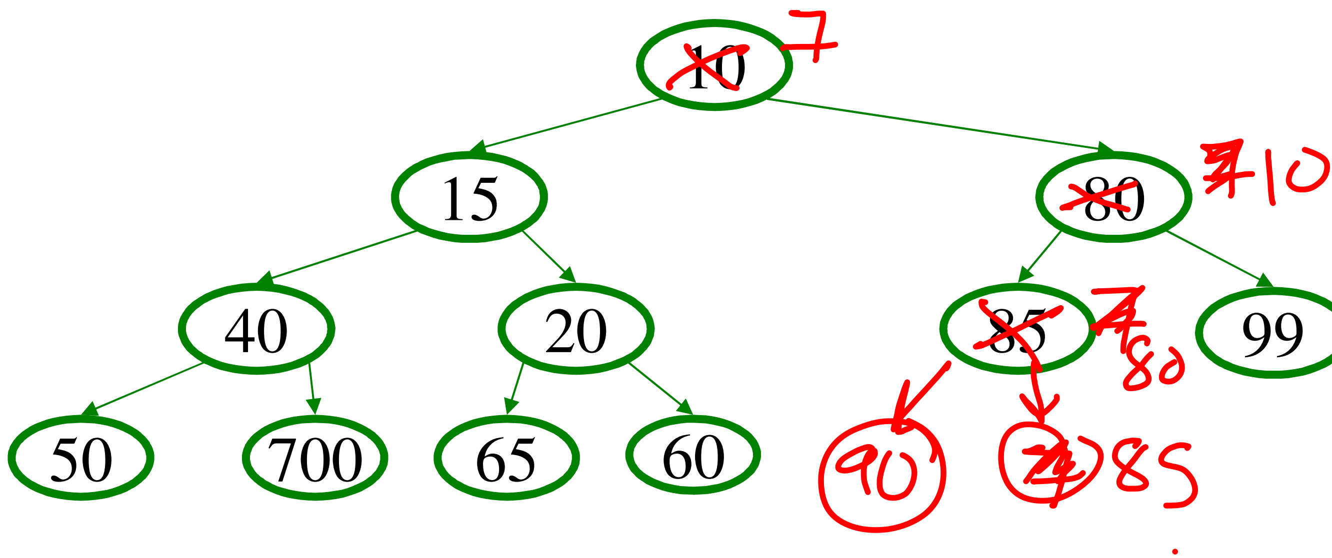
Heap – insert(val)

Basic Idea:

1. Put val at last leaf position
2. Percolate up by repeatedly exchanging node with parent as long as needed

Insert: percolate up





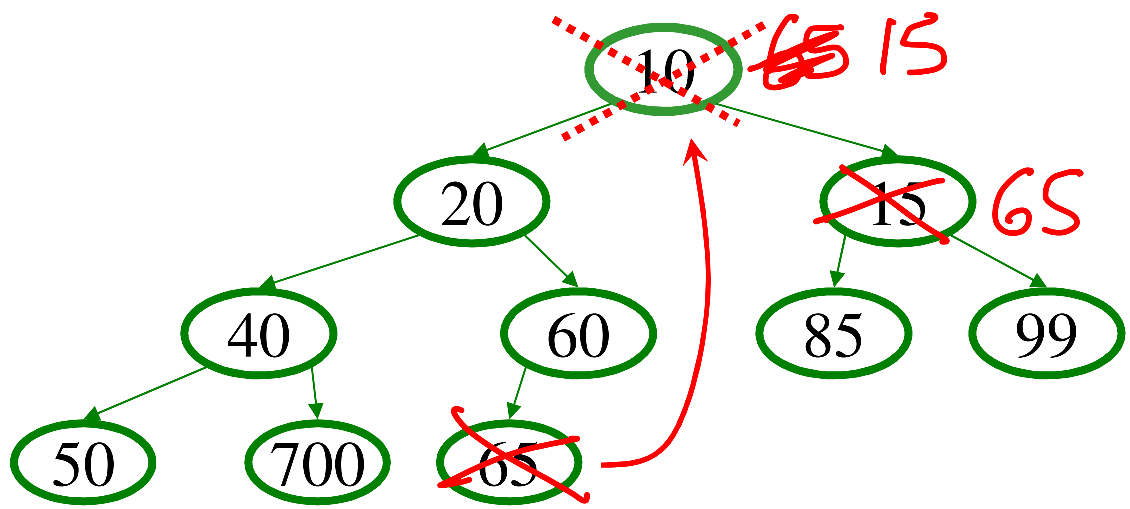
Heap – deleteMin

Basic Idea:

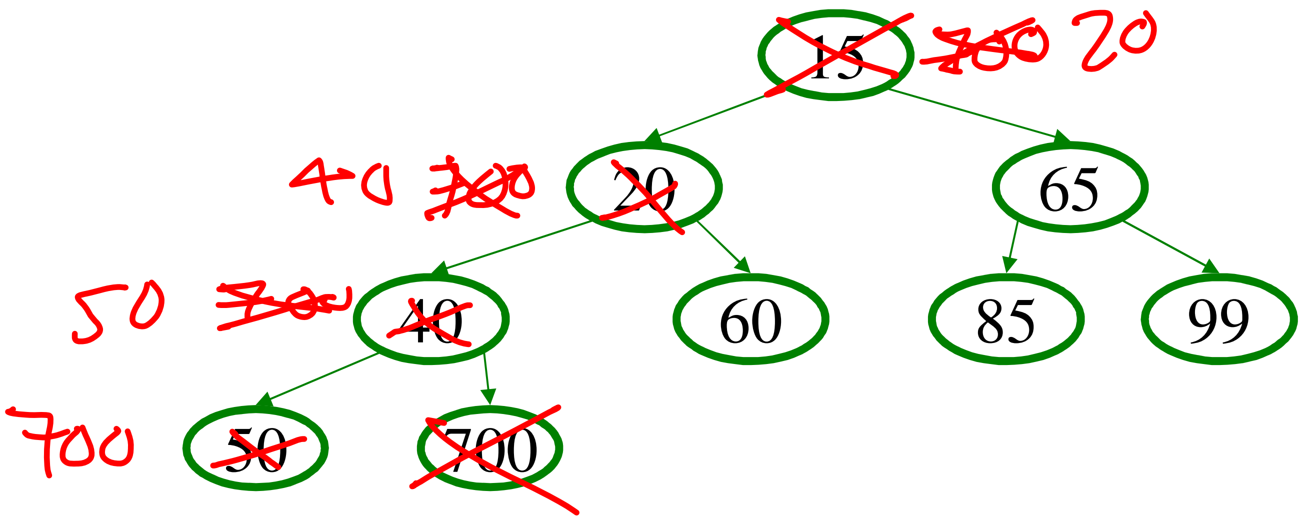
1. Remove min element
2. Put "last" leaf node value at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

$O(\log n)$

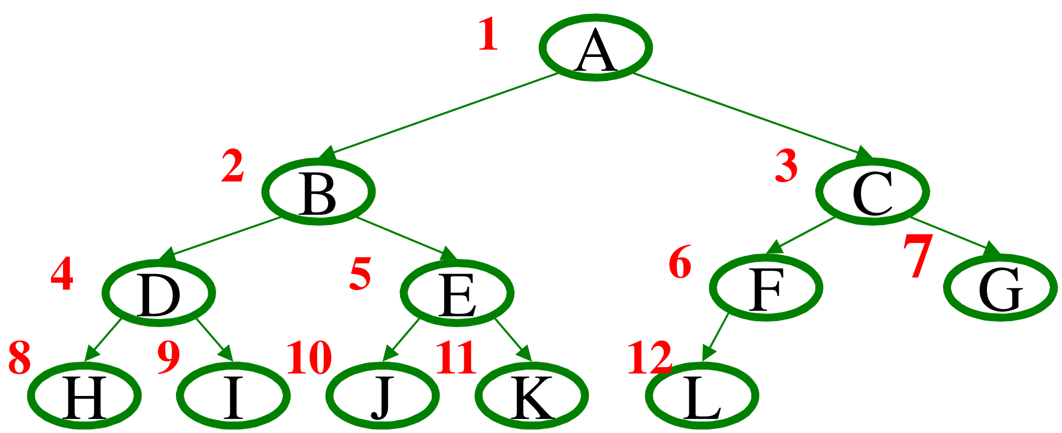
DeleteMin: percolate down



DeleteMin: percolate down



Representing Complete Binary Trees in an Array



From node **i**:

left child: $2i$
right child: $2i+1$
parent: $\lfloor i/2 \rfloor$

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Why use an array?

locality

easy to access any element (leaves, end)

more memory efficient?

DeleteMin Code

```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    percolateDown(1,
        Heap[size + 1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

runtime:

(Java code in book)

```
int percolateDown(int hole,
                  Object val) {
while (2*hole <= size) {
    left = 2*hole;
    right = left + 1;
    if (right ≤ size &&
        Heap[right] < Heap[left])
      target = right;
    else
      target = left;

    if (Heap[target] < val) {
      Heap[hole] = Heap[target];
      hole = target;
    }
    else
      break;
  }
  return hole;
}
```

Insert Code

```
void insert(Object o) {
  assert(!isFull());
  size++;
  newPos =
    percolateUp(size,o);
  Heap[newPos] = o;
}
```

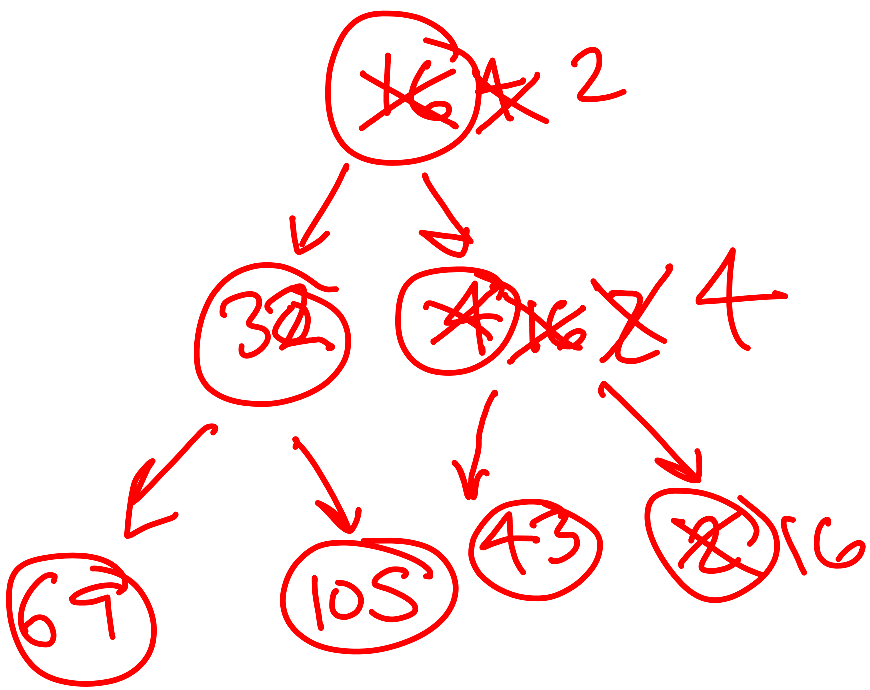
```
int percolateUp(int hole,
                Object val) {
  while (hole > 1 &&
         val < Heap[hole/2])
    Heap[hole] = Heap[hole/2];
    hole /= 2;
  }
  return hole;
}
```

runtime: O(log n)

(Java code in book)

Insert: 16, 32, 4, 69, 105, 43, 2

	2	32	4	69	105	43	16	
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**



More Priority Queue Operations

decreaseKey(nodePtr, amount):
given a pointer to a node in the queue, reduce its priority

Binary heap: change priority of node and percolate up

increaseKey(nodePtr, amount):
given a pointer to a node in the queue, increase its priority

Binary heap: change priority of node and percolate down

Why do we need a *pointer*? Why not simply data value?
search O(n)

Worst case running times? O(log n)

More Priority Queue Operations

remove(objPtr):

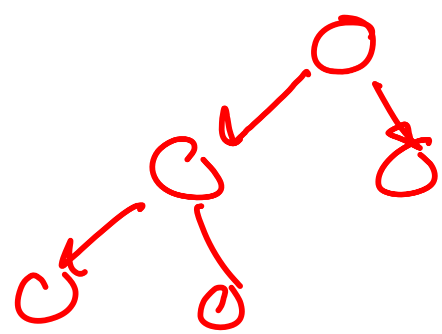
given a pointer to an object in the queue, remove it

Binary heap: decrease(objPtr, ∞), deletemin

findMax( ):

Find the object with the highest value in the queue

Binary heap: largest leaf



Worst case running times?

More Binary Heap Operations

expandHeap( ):

If heap has used up array, copy to new, larger array.

- Running time: O(n)

buildHeap(objList):

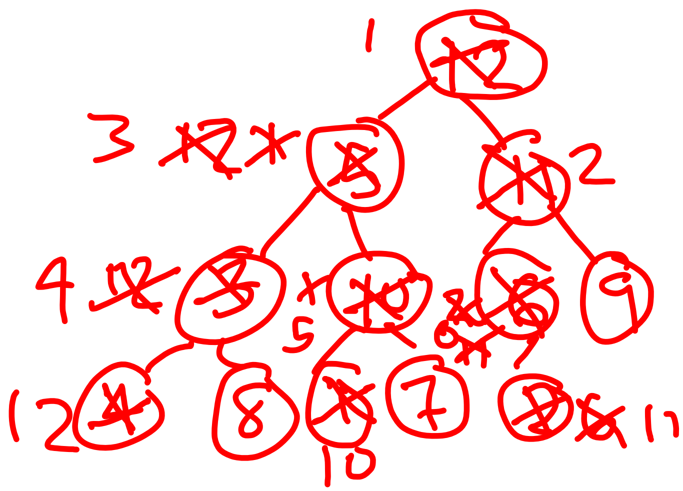
Given list of objects with priorities, fill the heap.

- Running time: O(n log n)

We do better with **buildHeap**...

Building a Heap: ~~Take 1~~

12	5	11	3	10	6	9	4	8	1	7	2



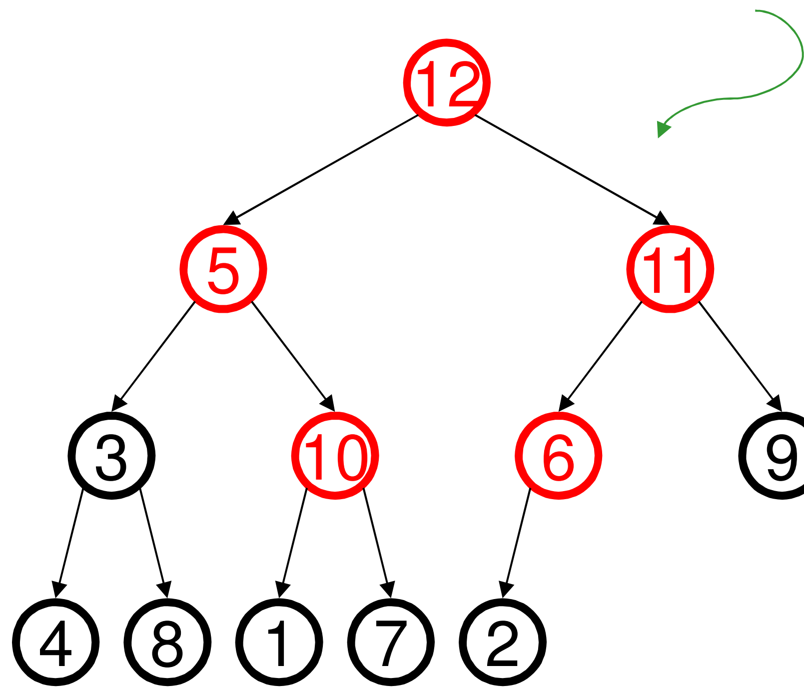
BuildHeap: Floyd's Method

12	5	11	3	10	6	9	4	8	1	7	2

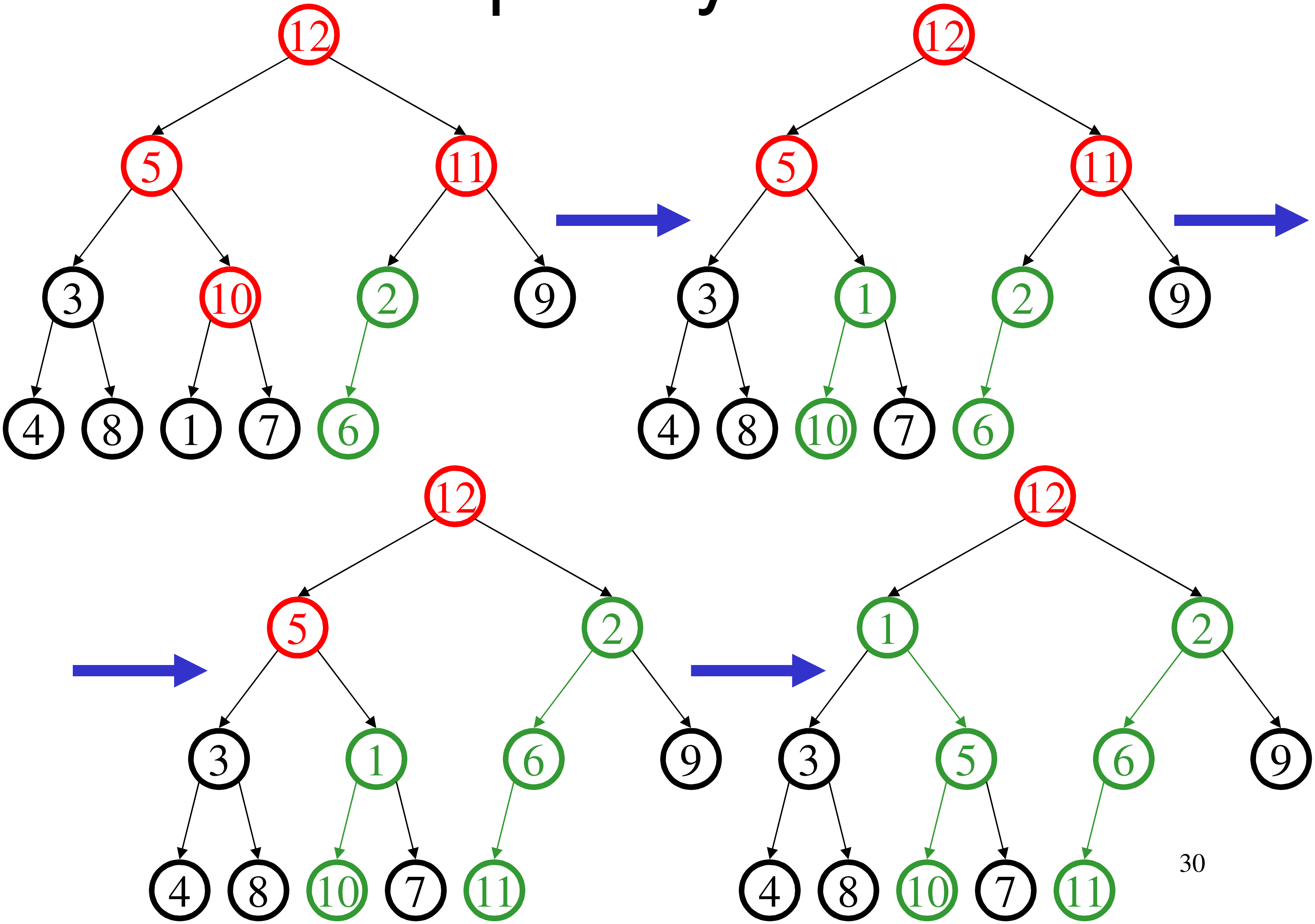
Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

Red nodes need to percolate down

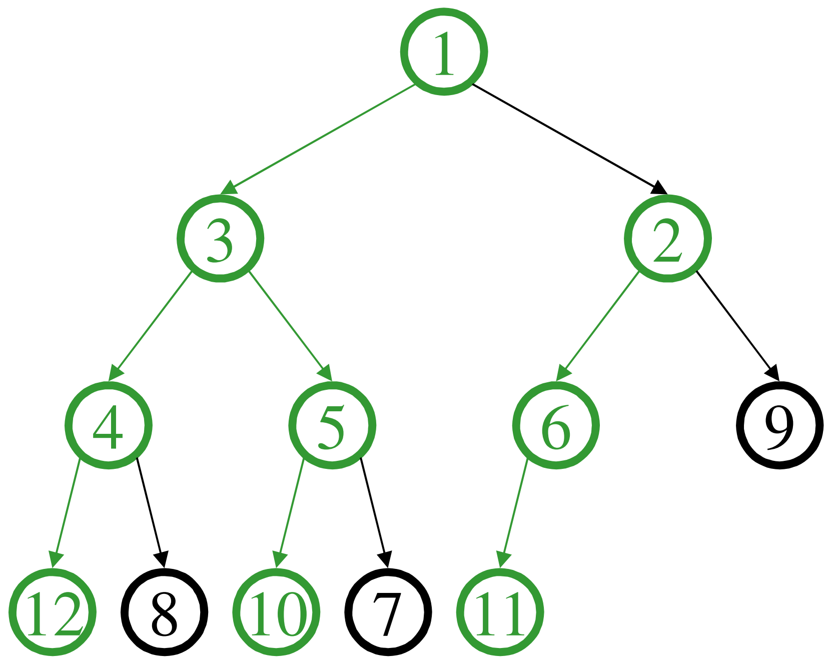
Key idea: fix red nodes from **bottom-up**



BuildHeap: Floyd's Method



Finally…



Buildheap pseudocode

```
private void buildHeap() {
  for ( int i = currentSize/2; i > 0; i-- )
     percolateDown( i );
}
```

runtime:

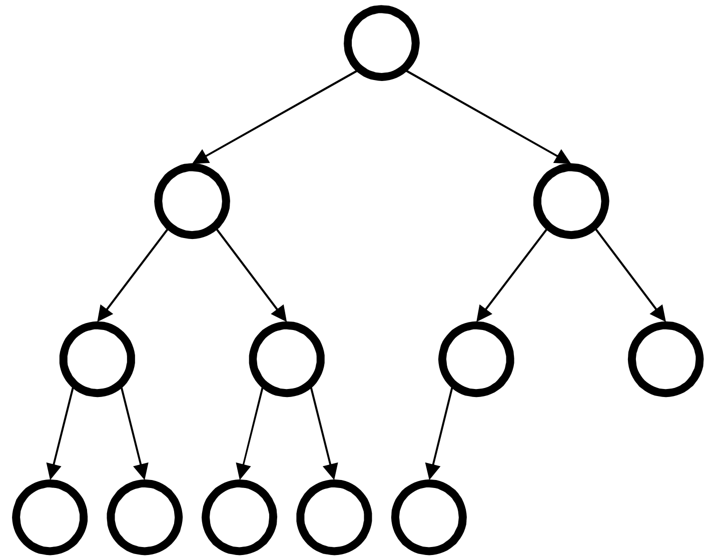
Buildheap Analysis

n/4 nodes percolate at most 1 level

n/8 percolate at most 2 levels

n/16 percolate at most 3 levels

...



runtime: