CSE 332: Data Structures and Parallelism

Spring 2022

Richard Anderson

Lecture 4: Priority Queues

Announcements

- Reading: Weiss, for Monday and Wednesday
  - Priority Queues, 6.1-6.5
- Ex 2 due Friday
- Checkpoint for P1 on Apr 6

Today

- Leftovers: Algorithm run time and big Oh
- Priority Queues
- Binary Tree Implementation
- Array implementation

Summary of Friday

- We need a rigorous way of talking about the performance of an algorithm
  - Real world to math world
- Ideas
  - Measure run time on an input by counting “steps”
  - Run time function for an algorithm, R(n): Worst case run time on input of size n
  - Only consider the rate of growth of run time functions by ignoring constants with big-Oh

Asymptotic Analysis

Eliminate low order terms

- $4n + 5 \Rightarrow$
- $0.5\ n \log n + 2n + 7 \Rightarrow$
- $n^3 + 3\ 2^n + 8n \Rightarrow$

Eliminate coefficients

- $4n \Rightarrow$
- $0.5\ n \log n \Rightarrow$
- $3\ 2^n \Rightarrow$

Formal definition of Big-Oh

h(n) is O(f(n)) if there exist positive constants c and n_0 such that $h(n) \leq c\ f(n)$ for all $n \geq n_0$

Big-Oh Example

h(n) is O(f(n)) if there exist positive constants c and n_0 such that $h(n) \leq c\ f(n)$ for all $n \geq n_0$

Example:

$100n^2 + 1000 \leq 1\ (n^3 + 2n^2)$ for all $n \geq 100$

So $100n^2 + 1000$ is $O(n^3 + 2n^2)$

Asymptotic Lower Bounds

$\Omega(\ g(n)\ )$ is used for functions asymptotically greater than or equal to $g(n)$

$h(n)$ is $\Omega(\ g(n)\ )$ if there exist $c>0$ and $n_0>0$ such that $h(n) \geq c\ g(n)$ for all $n \geq n_0$

$h(n)$ is $\theta(\ f(n)\ )$ is $h(n)$ is $O(\ f(n)\ )$ and $h(n)$ is $\Omega(f(n))$

Logarithms

- $\log_A B$ is the solution of $A^x = B$

- Default base of log can be 2, e, or 10
- Other bases can be important in CS
  - Height of a d-ary tree

Properties of Logs

Basic:

- $A^{\log_A B} = B$
- $\log_A A =$

Independent of base:

- $\log(AB) =$
- $\log(A/B) =$
- $\log(A^B) =$
- $\log((A^B)^C) =$

Change of base

$$\log_a N = \log_a(b^{\log_b N}) = \log_b N \log_a b$$

$$\log_b N = \frac{\log_a N}{\log_a b}$$

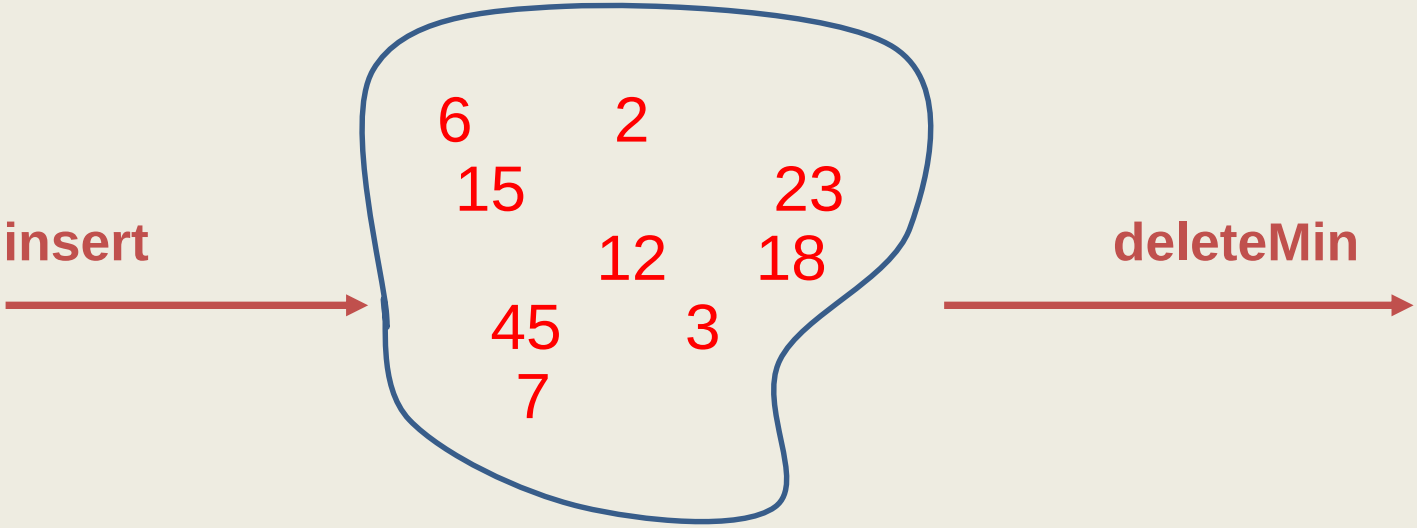
Base of the log can be ignored in big-Oh (provided the base is a constant)

Priority Queues

Manage a set, with insert item and get highest priority item

Priority Queue ADT

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



Priority Queue ADT

1. PQueue data : collection of data with priority

2. PQueue operations
   - insert
   - deleteMin

   (also: create, destroy, is_empty)

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

Potential Implementations

	Insert	DeleteMin
Unsorted list (Array)		
Unsorted list (Linked list)		
Sorted list (Array)		
Sorted list (Linked list)		
Binary Search Tree		

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 some other types of operations (e.g., searching, sorting)

Tree Review

root(T): A

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

children(B): D-F

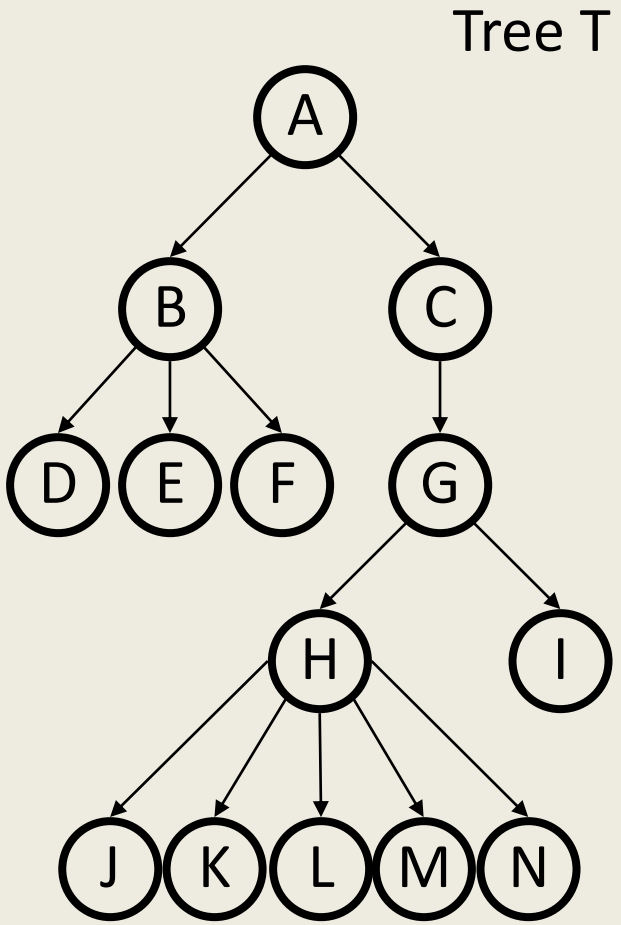
parent(H): G

siblings(E): D,F

ancestors(F):

descendants(G):

subtree(C):



More Tree Terminology

depth(B):

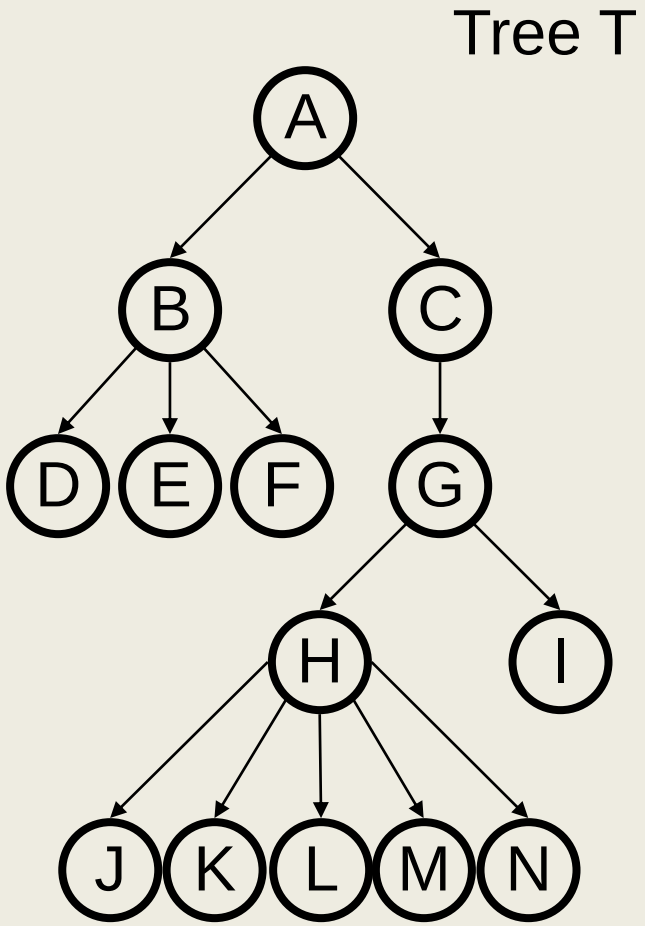
height(G):

height(T):

degree(B):

branching factor(T):

n-ary tree:



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**
2. **Heap Order**

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

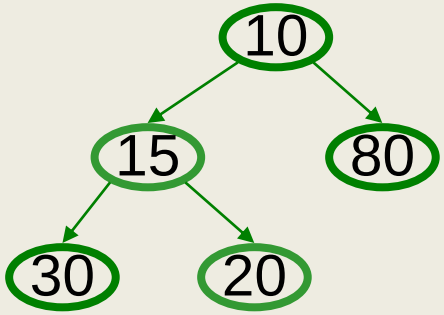
Completeness

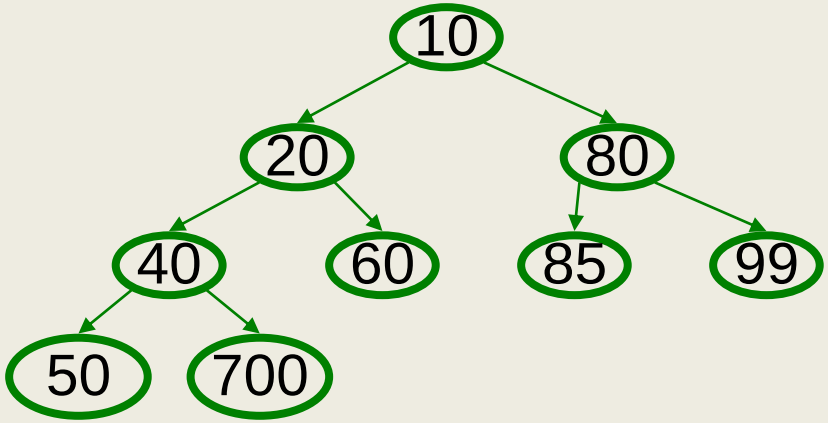
A binary heap is a complete binary tree

All levels are full, except the bottom, which is filled to the right

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.





Heap Operations

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

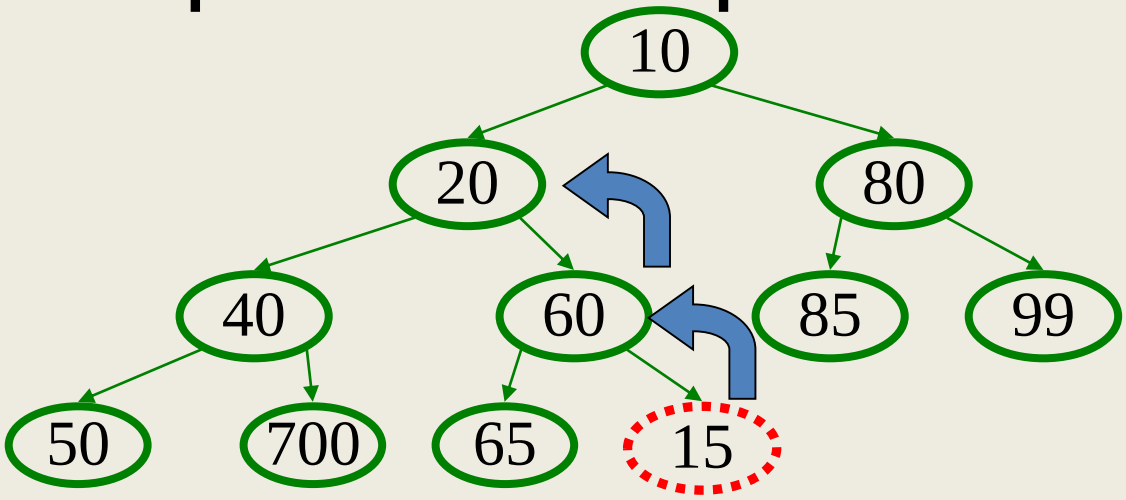
- Basic idea is to propagate changes up/down the tree, fixing order as we go

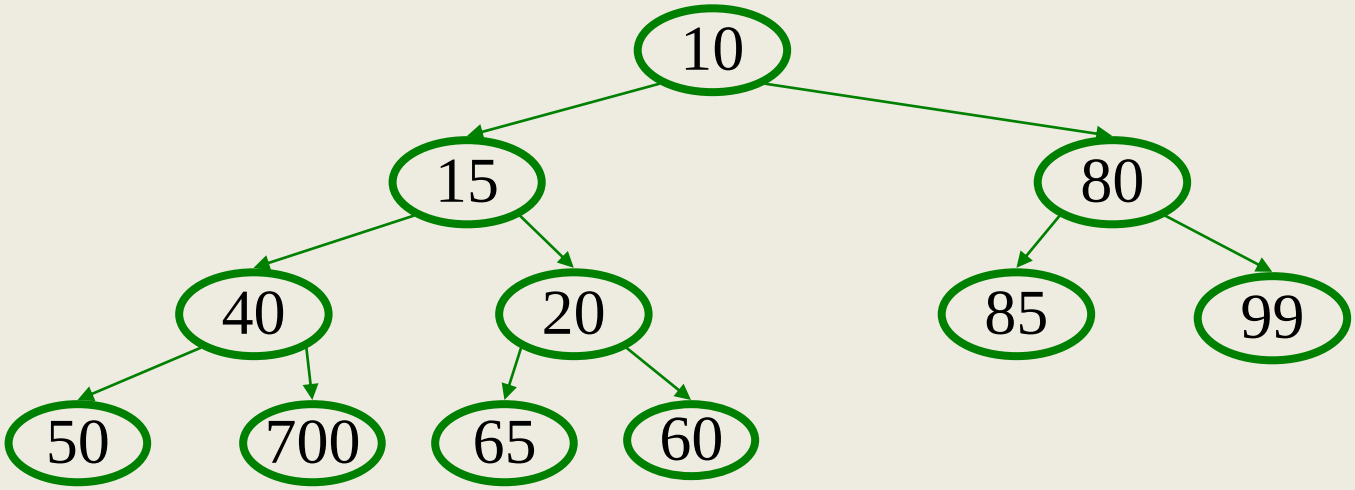
Heap – insert(val)

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



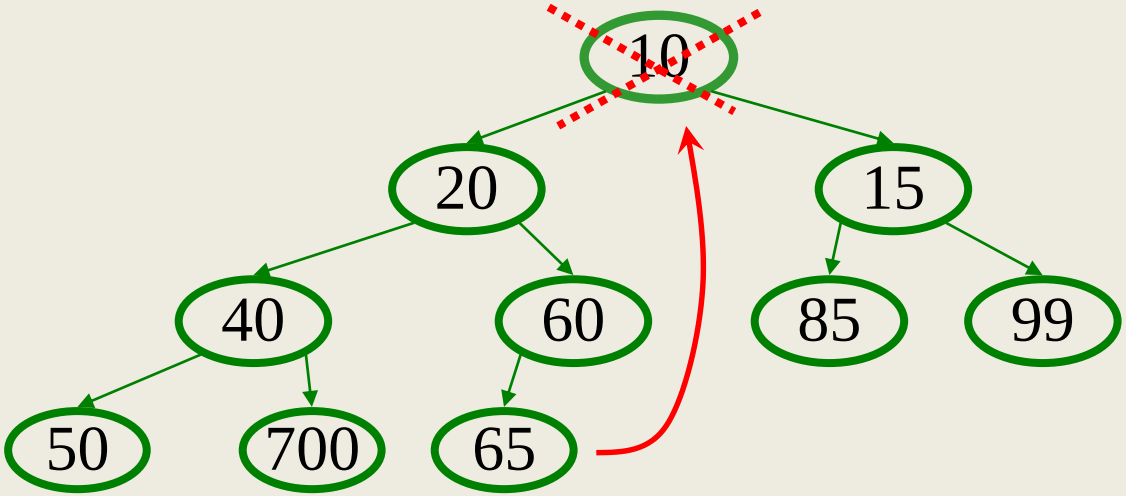


Heap – deleteMin

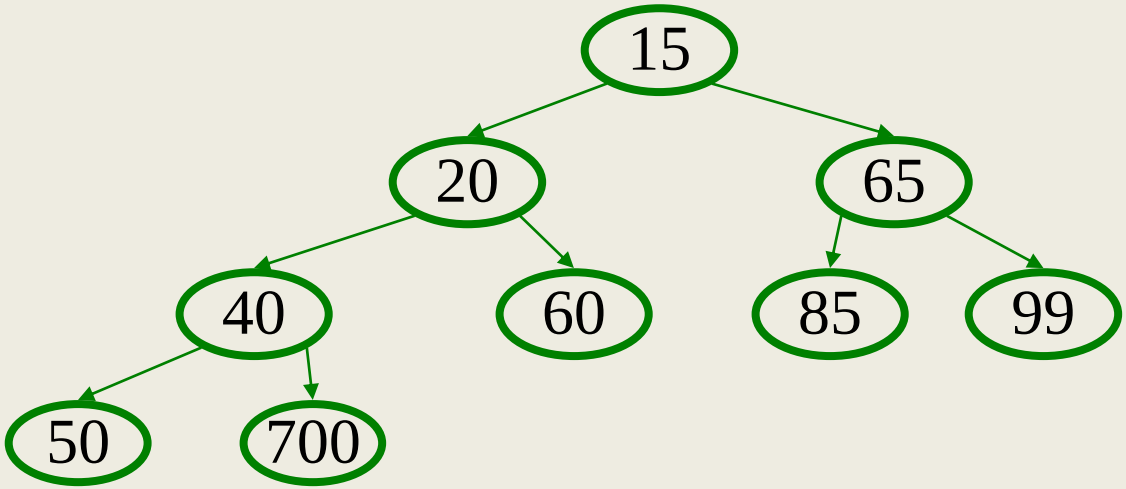
Basic Idea:

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.

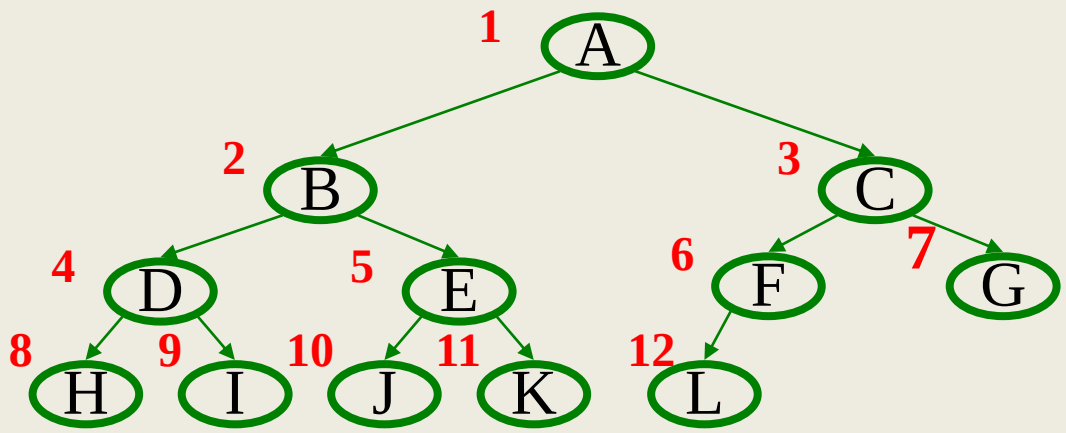
DeleteMin: percolate down



DeleteMin: percolate down



Representing Complete Binary Trees in an Array



From node **i**:

left child:
right child:
parent:

	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?

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

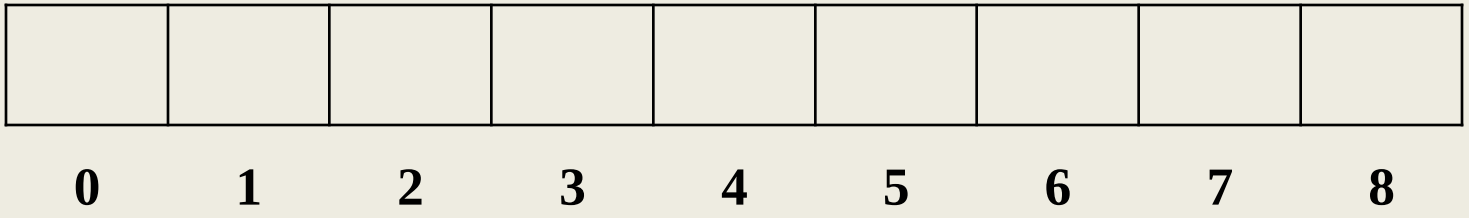
```
void insert(Object o) {
  assert(!isFull());
  size++;
  newPos =
    percolateUp(size,o);
  Heap[newPos] = o;
}
```

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

runtime:

(Java code in book)

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



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 _____

increaseKey(nodePtr, amount):

given a pointer to a node in the queue, increase its priority

Binary heap: change priority of node and _____

Why do we need a *pointer*? Why not simply data value?

Worst case running times?

More Priority Queue Operations

remove(objPtr):

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

Binary heap: _____

findMax( ):

Find the object with the highest value in the queue

Binary heap: _____

Worst case running times?

More Binary Heap Operations

expandHeap( ):

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

- Running time:

buildHeap(objList):

Given list of objects with priorities, fill the heap.

- Running time:

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

Building a Heap: Take 1

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

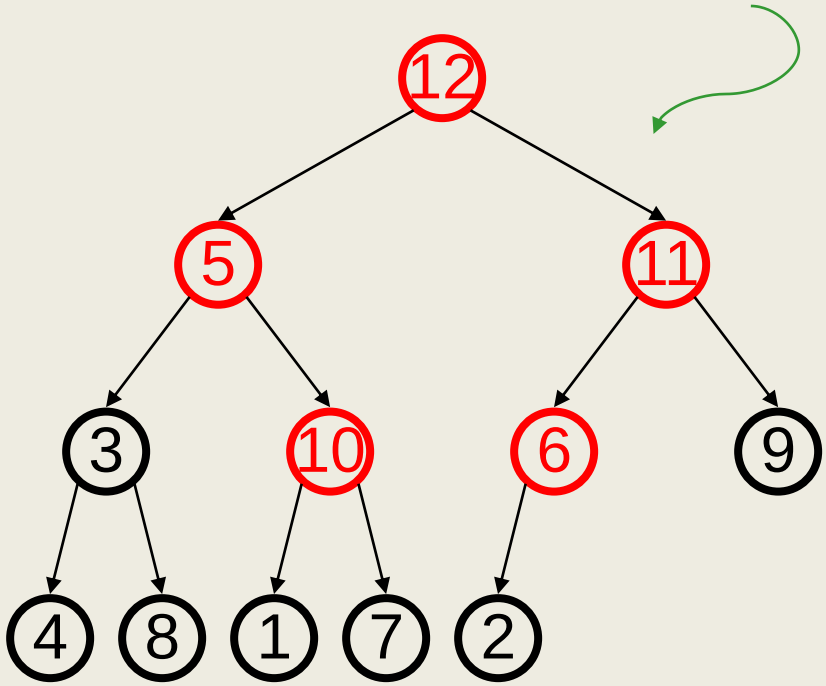
BuildHeap: Floyd's Method

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

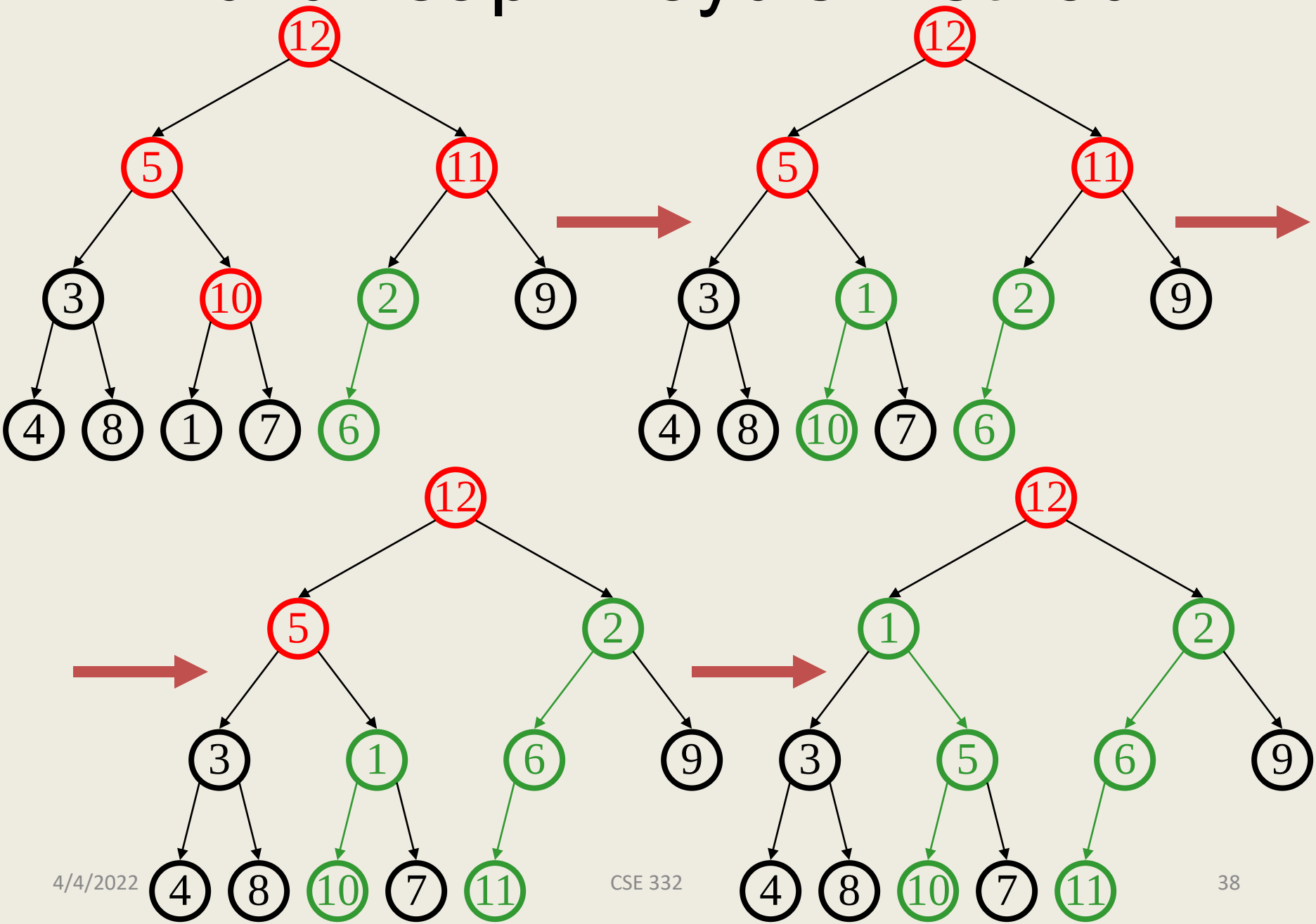
Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

Red nodes need to percolate down

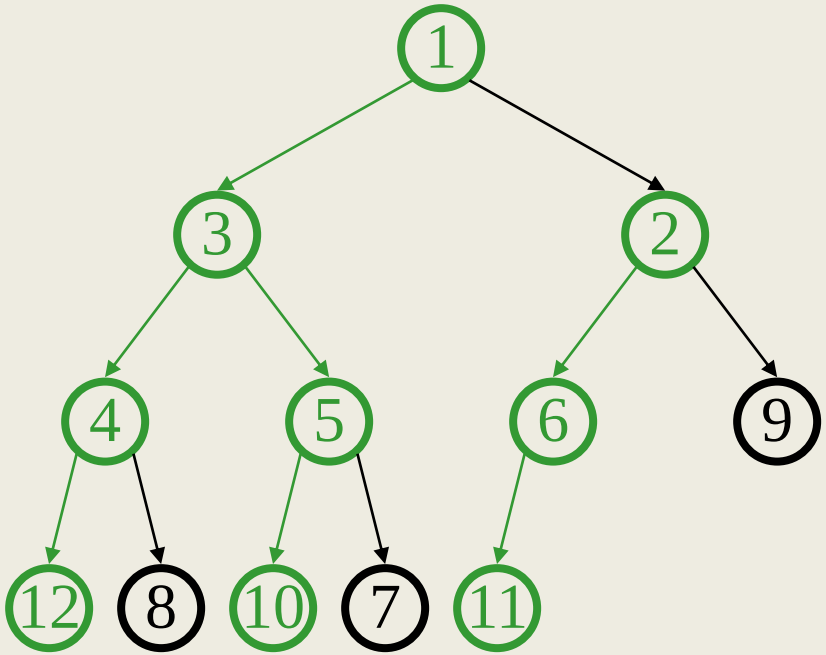
Key idea: fix red nodes from **bottom-up**



BuildHeap: Floyd's Method



Finally…



Buildheap pseudocode

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

runtime:

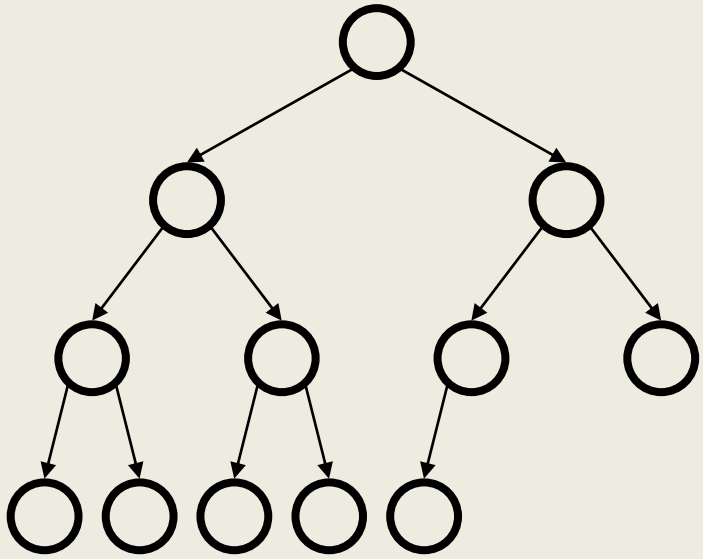
Buildheap Analysis

n/4 nodes percolate at most 1 level

n/8 percolate at most 2 levels

n/16 percolate at most 3 levels

...



runtime: