CSCI 136
Data Structures & Advanced Programming

“Heapifying” an Array

Video Outline

- Heaps
    - Quick review of implementation strategies
    - Creating heaps from unsorted arrays
        - A top-down approach
        - A bottom-up approach
        - Some analysis + proofs

VectorHeap Design: Recap

- A heap is a *semi*-sorted tree
  - Rather than a “global” sort ordering, “partial” ordering is maintained for all root-to-leaf paths
- Data stored directly in an implicit binary tree
  - Children of `i` are at `2i+1` and `2i+2`
  - Parent is at `(i-1)/2`
- Tree is always complete
  - A prefix of the Vector is always occupied–no gaps

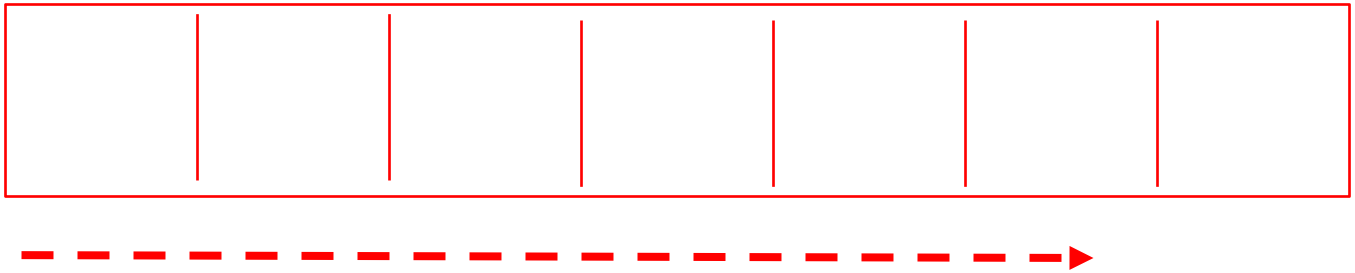
VectorHeap Operations: Recap

- Strategy: perform tree modifications that always preserve tree *completeness*, but may violate heap property. Then fix.
  - Add/remove never create gaps in between array elements
    - We always add in next available array slot (left-most available spot in binary tree)
    - We always remove using “final” leaf (rightmost element in array)
  - When elements are added and removed, do small amount of work to “re-heapify”
    - pushDownRoot(): recursively swaps large element down the tree
    - percolateUp(): recursively swaps small element up the tree

Heapifying A Vector (or array)

Problem: You are given a Vector `V` that is not a valid heap, and you want to “heapify” `V`

- Method I: Top-Down
  - Given `V[0...k]` satisfies the heap property
  - Call `percolateUp` on item in location `k+1`
  - Now, `V[0..k+1]` satisfies the heap property!

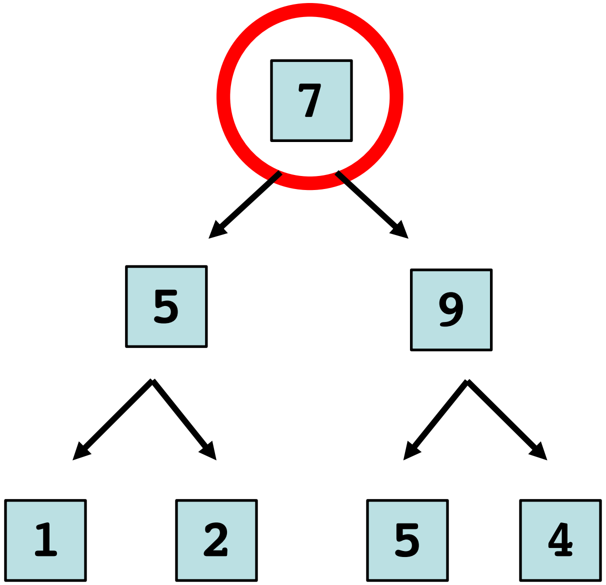


Grow valid heap region one element at a time

Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

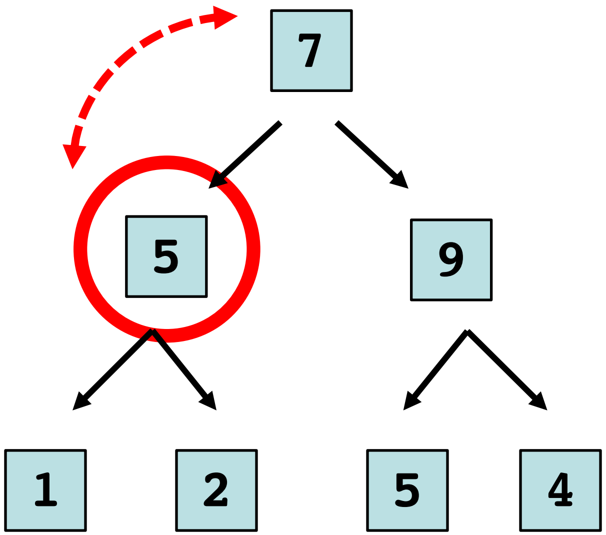
```
         0  1  2  3  4  5  6
    a = [7  5  9  1  2  5  4]
```



Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

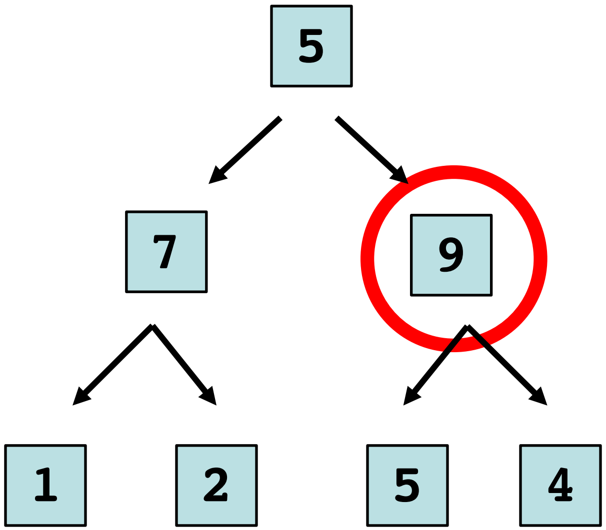
```
         0 1 2 3 4 5 6
    a = [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
```



Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

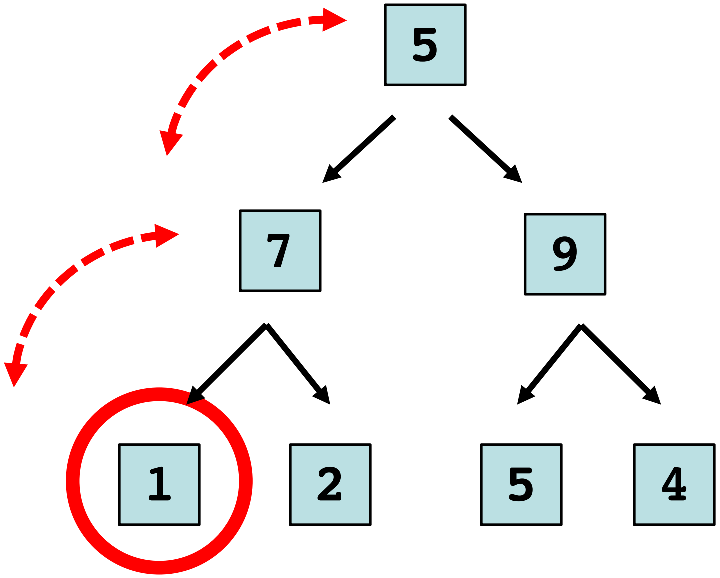
```
       0 1 2 3 4 5 6
a = [7 5 9 1 2 5 4]
    [7 5 9 1 2 5 4]
    [5 7 9 1 2 5 4]
```



Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

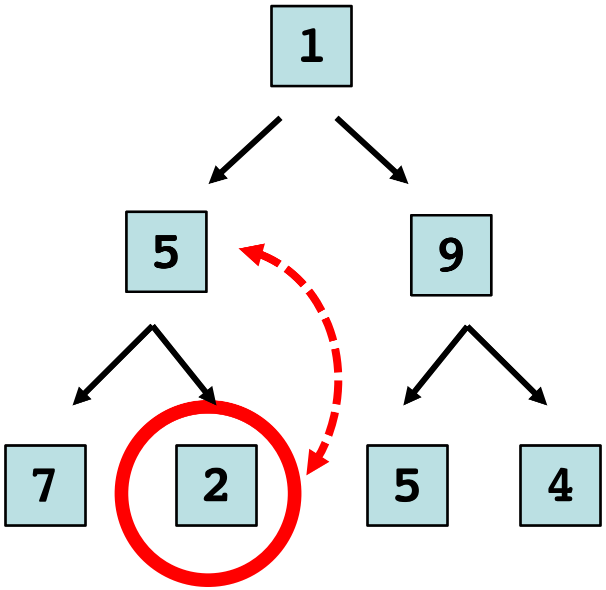
```
         0 1 2 3 4 5 6
    a = [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
        [5 7 9 1 2 5 4]
        [5 7 9 1 2 5 4]
```



Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

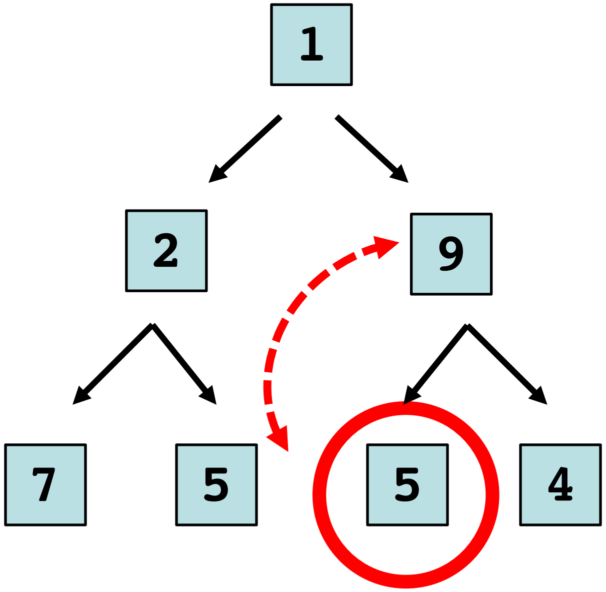
```
       0 1 2 3 4 5 6
  a = [7 5 9 1 2 5 4]
      [7 5 9 1 2 5 4]
      [5 7 9 1 2 5 4]
      [5 7 9 1 2 5 4]
      [1 5 9 7 2 5 4]
```



Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

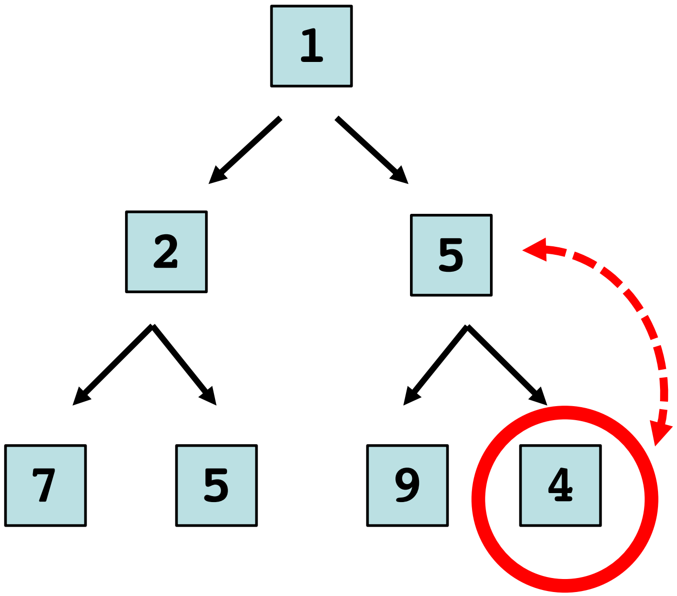
```
         0 1 2 3 4 5 6
    a = [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
        [5 7 9 1 2 5 4]
        [5 7 9 1 2 5 4]
        [1 5 9 7 2 5 4]
        [1 2 9 7 5 5 4]
```



Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

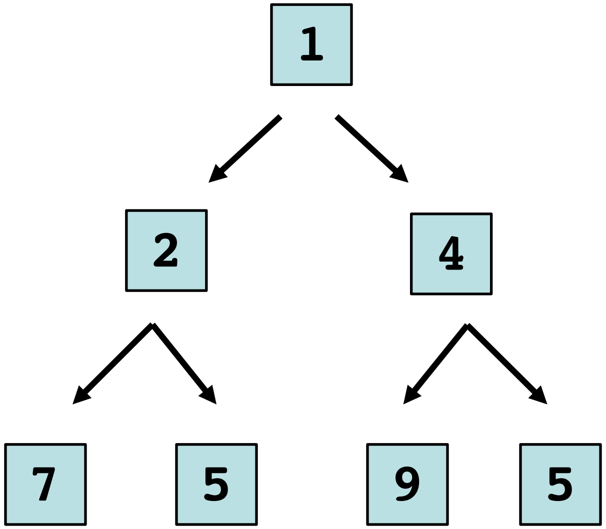
```
          0 1 2 3 4 5 6
    a = [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
        [5 7 9 1 2 5 4]
        [5 7 9 1 2 5 4]
        [1 5 9 7 2 5 4]
        [1 2 9 7 5 5 4]
        [1 2 5 7 5 9 4]
```



Practice Top-Down

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = 0; i < a.length; i++)
    percolateUp(a, i);
```

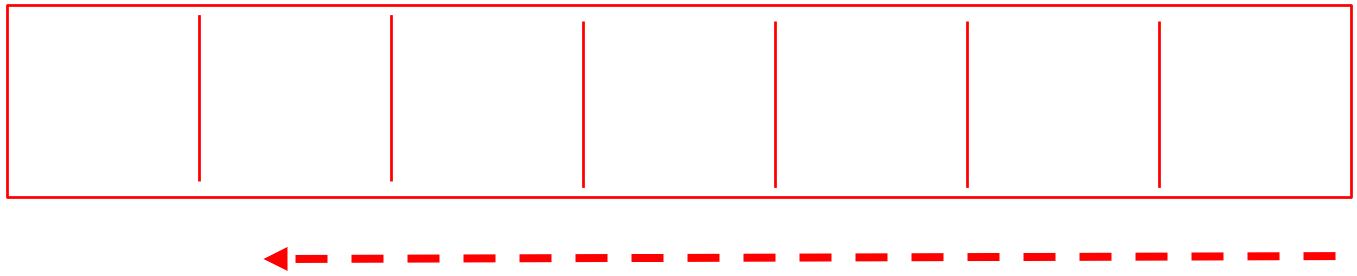
```
        0 1 2 3 4 5 6
    a = [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
        [5 7 9 1 2 5 4]
        [5 7 9 1 2 5 4]
        [1 5 9 7 2 5 4]
        [1 2 9 7 5 5 4]
        [1 2 5 7 5 9 4]
        [1 2 4 7 5 9 5]
```



Heapifying A Vector (or array)

Problem: You are given a Vector `V` that is not a valid heap, and you want to “heapify” `V`

- Method II: **Bottom-up**
  - Given `V[k..n]` satisfies the heap property
  - Call pushDown on item in location `k-1`
  - Now, `V[k-1..n]` satisfies heap property!

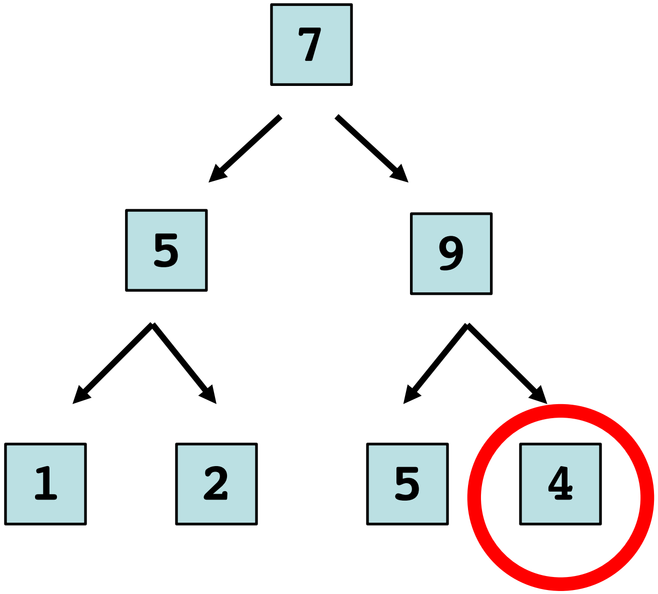


Grow valid heap region one element at a time

Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

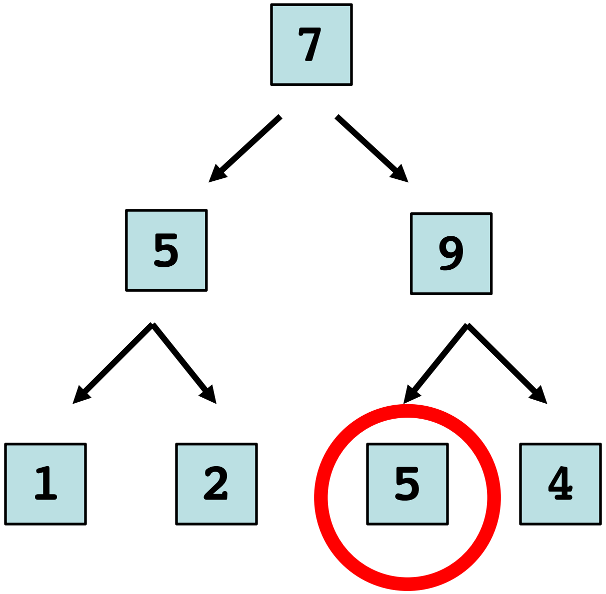
```
      0 1 2 3 4 5 6
a = [7 5 9 1 2 5 4]
```



Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

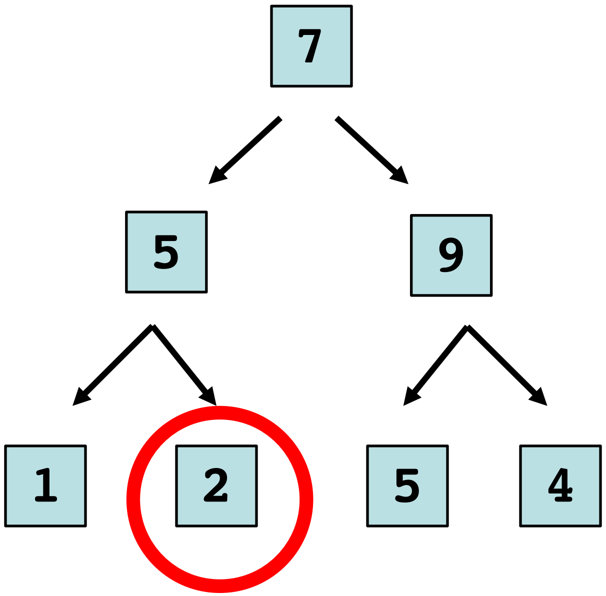
```
         0 1 2 3 4 5 6
    a = [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
```



Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

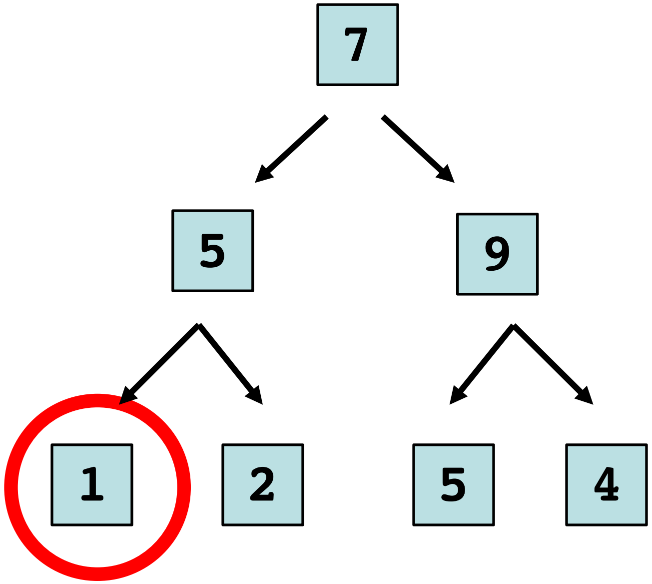
```
         0 1 2 3 4 5 6
     a = [7 5 9 1 2 5 4]
         [7 5 9 1 2 5 4]
         [7 5 9 1 2 5 4]
```



Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

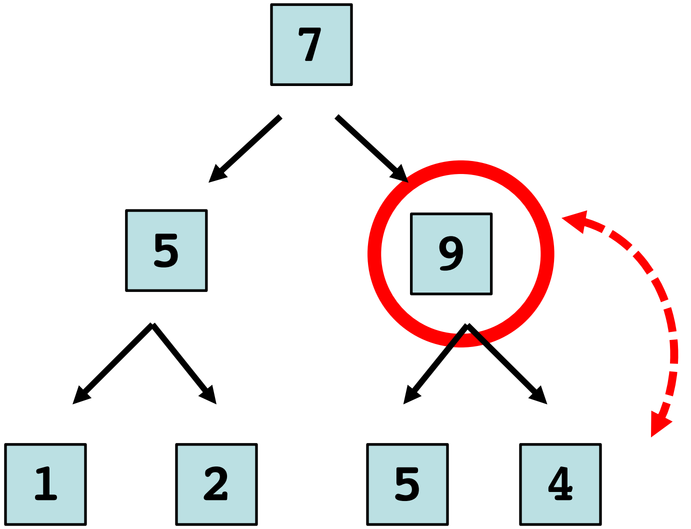
```
        0 1 2 3 4 5 6
a =    [7 5 9 1 2 5 4]
       [7 5 9 1 2 5 4]
       [7 5 9 1 2 5 4]
       [7 5 9 1 2 5 4]
```



Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

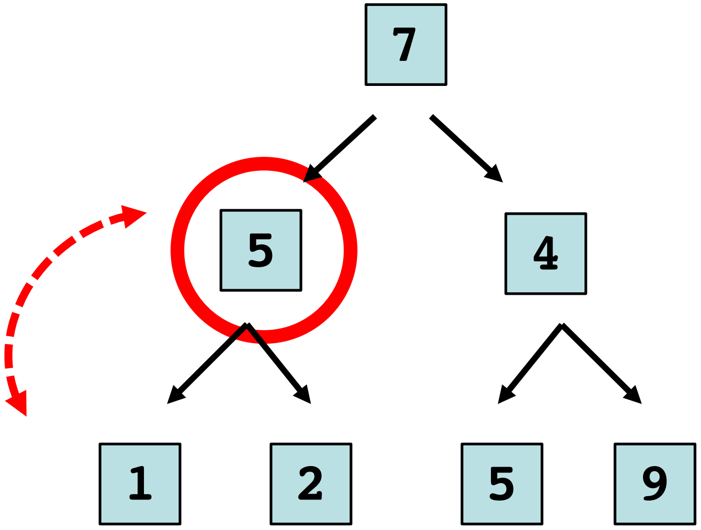
```
         0  1  2  3  4  5  6
    a = [7  5  9  1  2  5  4]
        [7  5  9  1  2  5  4]
        [7  5  9  1  2  5  4]
        [7  5  9  1  2  5  4]
        [7  5  9  1  2  5  4]
```



Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

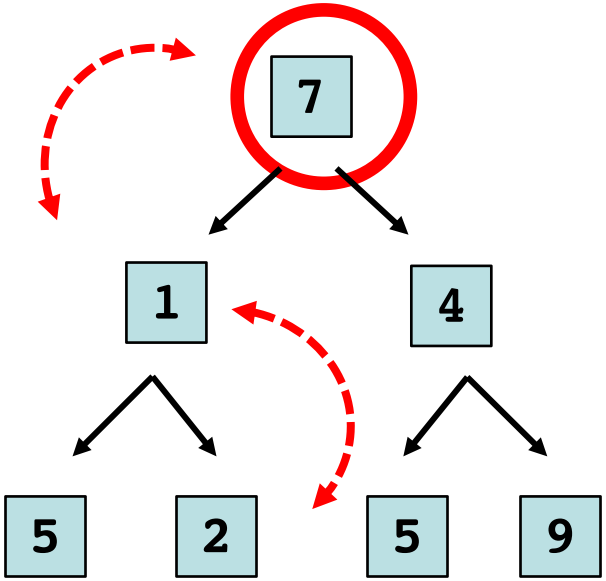
```
     0 1 2 3 4 5 6
a = [7 5 9 1 2 5 4]
    [7 5 9 1 2 5 4]
    [7 5 9 1 2 5 4]
    [7 5 9 1 2 5 4]
    [7 5 4 1 2 5 9]
```



Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

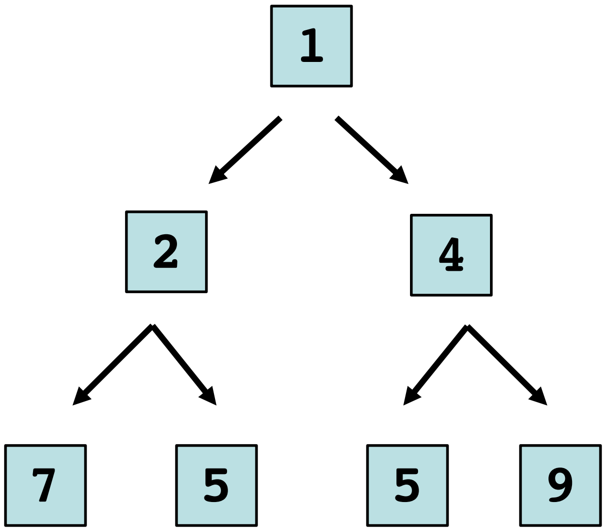
```
        0 1 2 3 4 5 6
    a = [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
        [7 5 9 1 2 5 4]
        [7 5 4 1 2 5 9]
        [7 1 4 5 2 5 9]
```



Practice Bottom-up

```
int a[7] = {7,5,9,1,2,5,4};
for (int i = a.length-1; i > 0; i--)
    pushDownRoot(a, i);
```

```
        0 1 2 3 4 5 6
a =    [7 5 9 1 2 5 4]
       [7 5 9 1 2 5 4]
       [7 5 9 1 2 5 4]
       [7 5 9 1 2 5 4]
       [7 5 4 1 2 5 9]
       [7 1 4 5 2 5 9]
       [1 2 4 7 5 5 9]
```



Let's Compare

- Which is faster: Top down or Bottom Up?
  - Q: Think about a complete binary tree. Where do most of the nodes live?
  - A: The leaves!
  - Given that most of the nodes are leaves, should we `percolateUp` or `pushDown`?
    - To answer this, we should think about "how far" we need to move a node in the worst case.

Some Sums (for your toolbox)

$$\sum_{d=0}^{n} 2^d = 2^{n+1} - 1$$

$$\sum_{d=0}^{n} r^d = (r^{n+1})/(r - 1)$$

⇨

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

⇨

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

All of these can be proven by (weak) induction.

Try these proofs to hone your skills!

(The second sum is called a geometric series. It works for any r≠0)

Top-Down vs Bottom-Up

- **Top-down heapify (percolate up)**: elements at depth `d` may be swapped `d` times.
- The total # of swaps is:

(recall: h = log n)

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

  - This is `O(n log`$_2$`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`$_2$`n)` swaps per element

Top-Down vs Bottom-Up

- **Bottom-up heapify (push down):** elements at depth `d` may be swapped `h−d` times.
- The total # of swaps is:

(recall: h = log n)

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

$$= 2n - 2\log n - 2$$

  - This is O(n) — it 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!!!

Summary

- There are multiple valid ways to create a heap from an unsorted array
- The choices we make impact performance, so think carefully about the problem structure when developing your approach
- The same arguments apply to min-heaps and max-heaps: just inverse the swapping condition.