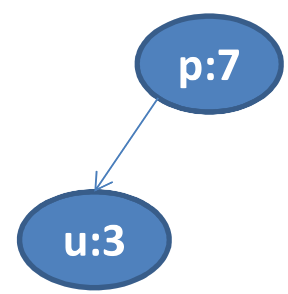
What is a heap?

- Always keep the thing we are most interested in close to the top (and fast to access).
- Like a binary search tree, but less structured.
- No relationship between keys at the same level (unlike BST).

Types of heaps

- Min heap: priority 1 more important than 100
- Max heap: 100 more important than 1
  – We are going to talk about max heaps.

- **Max heap-order property**
  – Look at any node u, and its parent p.
  – p.priority ≥ u.priority



Abstract data type (ADT)

- We are going to use max-heaps to implement the (max) **priority queue** ADT
- A priority queue Q offers (at least) 2 operations:
  – Extract-max(Q): returns the highest priority element
  – Insert(Q, e): inserts e into Q

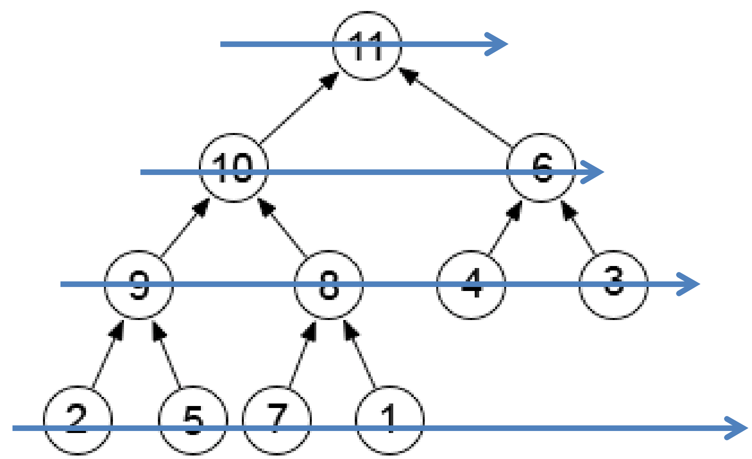
- Every time an Insert or Extract-max *changes* the heap, it must **restore the max-heap order property**. (← Prove by induction on the sequence of inserts and extract-maxes that occur.)

What can we do with a heap

- Can do same stuff with a BST… why use a heap?
  - BST extract-max is O(depth); heap is O(log n)!
- When would we use a BST?
  - When we need to search for a particular key.

Storing a heap in memory

- Heaps are typically implemented with arrays.



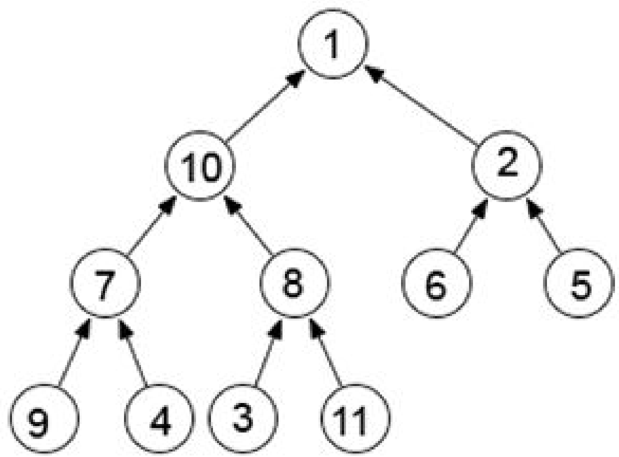
- The array is just a *level-order traversal* of the heap.
- The children of the node at index i are at 2i and 2i+1.

Example time

- Interactive heap visualization
- Insert places a key in a new node that is the *last* node in a level-order-traversal of the heap.
  - The inserted key is then “bubbled” **upwards** until the heap property is satisfied.
- Extract-max removes the last node in a level-order-traversal and moves its key into the root.
  - The new key at the root is then bubbled **down** until the heap property is satisfied.
  - Bubbling down is also called **heapifying**.

Building a max-heap in O(n) time

- Suppose we want to build a heap from an unsorted array: 10, 2, 7, 8, 6, 5, 9, 4, 3, 11.
- We start by interpreting the array as a tree.

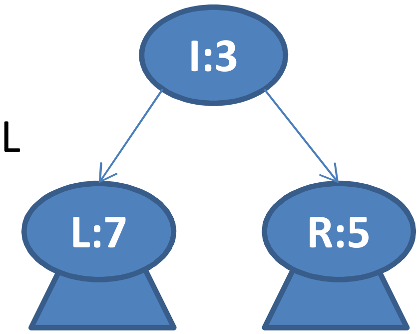


1	10	2	7	8	6	5	9	4	3	11
1	2	3	4	5	6	7	8	9	10	11

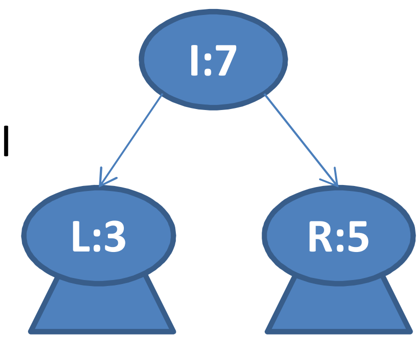
Building a heap: a helper function

Precondition: trees rooted at L and R are heaps
Postcondition: tree rooted at I is a heap

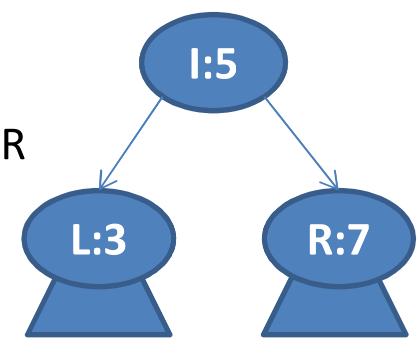
```
MaxHeapify(A,I):
   L = LEFT(I)
   R = RIGHT(I)

   If L ≤ heap_size(A) and A[L] > A[I]
      then max = L
      else max = I
   If R ≤ heap_size(A) and A[R] > A[max]
      then max = R

   If max is L or R then
      swap(A[I],A[max])
      MaxHeapify(A,max)
```



Case 1: max = L
Need to fix…



Case 2: max = I
Heap OK!



Case 3: max = R
Need to fix…

Proving MaxHeapify is correct

- **How would you formally prove that MaxHeapify is correct?**
- **Goal: Prove MaxHeapify is <u>correct</u> for all inputs.**
  "Correct" means: "if the precondition is satisfied when MaxHeapify is called, then the postcondition will be satisfied when it finishes."
- How do we prove a recursive function correct?
  - Define a problem size, and prove correctness by induction on problem size.
  - Base case: show function is correct for any input of the smallest problem size.
  - Inductive step: assume function is correct for problem size j; show it is correct for problem size j+1.

Proving MaxHeapify is correct - 2

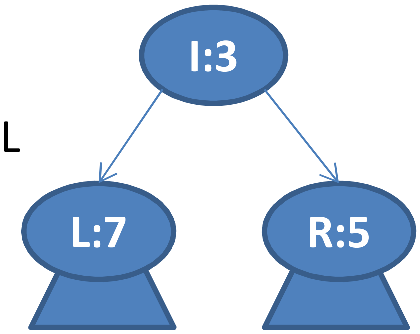
- **Let's apply this to MaxHeapify.**
- **Problem size:** height of node I.
- **Base case:** Prove MaxHeapify is correct for every input with height(I) = 0.
- **Inductive step:** Let A and I be any input parameters that satisfy the precondition.
  Assume MaxHeapify is correct when the problem size is j.
  Prove MaxHeapify is correct when the problem size is j+1.

Proving MaxHeapify is correct - 3

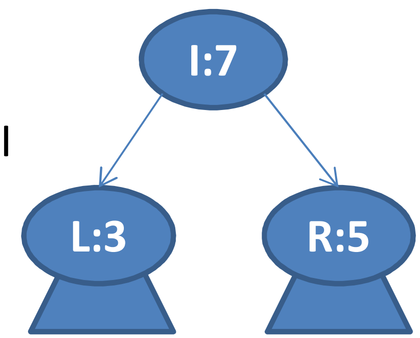
Precondition: trees rooted at L and R are heaps
Postcondition: tree rooted at I is a heap

```
MaxHeapify(A,I):
  L = LEFT(I)
  R = RIGHT(I)

  If L ≤ heap_size(A) and A[L] > A[I]
    then max = L
    else max = I
  If R ≤ heap_size(A) and A[R] > A[max]
    then max = R

  If max is L or R then
    swap(A[I],A[max])
    MaxHeapify(A,max)
```

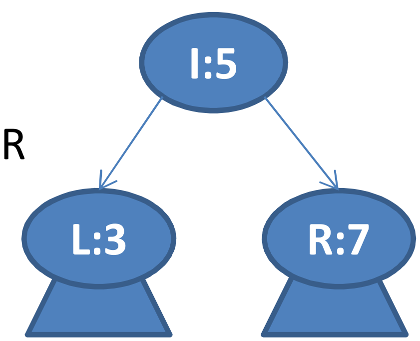
Case 1: max = L
Need to fix…



Case 2: max = I
Heap OK!



Case 3: max = R
Need to fix…



The main function

```
BUILD-MAX-HEAP(A):
  for i = heap_size(A)/2 down to 1
    MaxHeapify(A,i)
```

Analyzing worst-case complexity

- Recall: (for c_1, c_2 constants)
  - $O(n)$ means **worst input** takes **at most** c_1*n steps
  - $\Omega(n)$ means **worst input** takes **at least** c_2*n steps

- How can we show $\boldsymbol{\Omega(n)}$?
  - Recall the code of BUILD-MAX-HEAP(A):

```
BUILD-MAX-HEAP(A):
    for i = heap_size(A)/2 down to 1
        MaxHeapify(A,i)
```

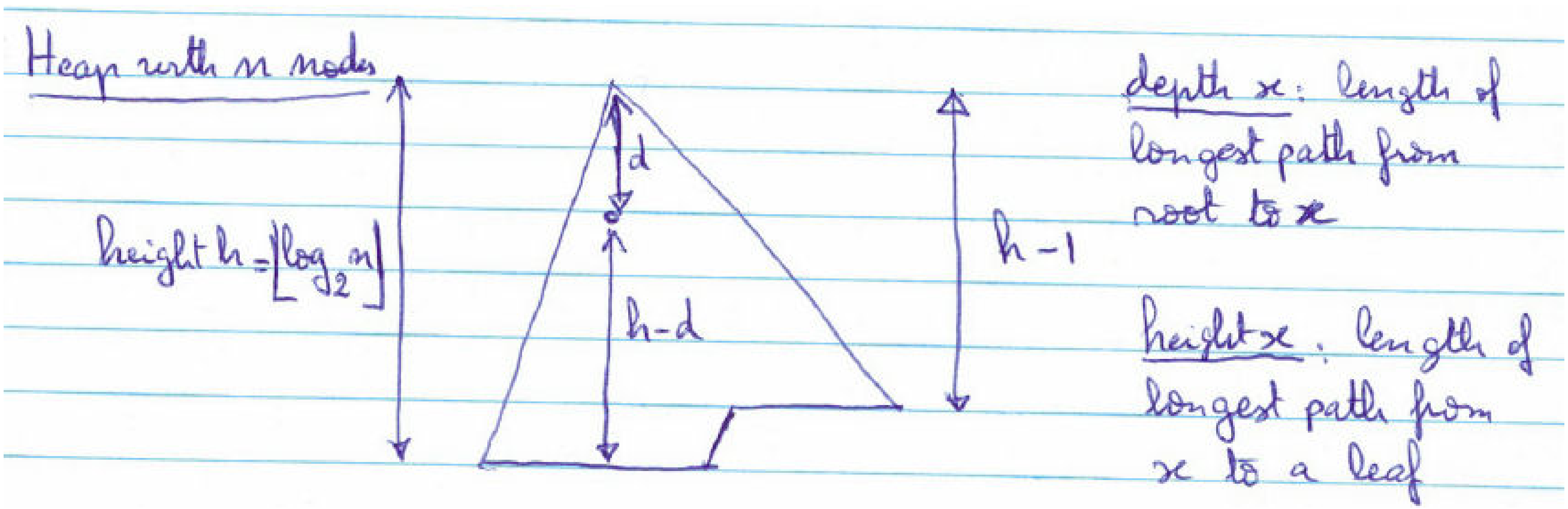
Analyzing worst-case complexity

- Recall: (for c_1, c_2 constants)
  - $O(n)$ means **worst input** takes **at most** c_1*n steps
  - $\Omega(n)$ means **worst input** takes **at least** c_2*n steps

- Harder question: how can we show $\boldsymbol{O(n)}$?
  - Recall the code of BUILD-MAX-HEAP(A):

```
BUILD-MAX-HEAP(A):
    for i = heap_size(A)/2 down to 1
        MaxHeapify(A,i)
```

Showing $O(n)$



- **≤ 2^d** nodes at depth **d**
- Node at depth **d** has height **≤ h-d**
- Cost to "heapify" **one** node at depth **d** is **≤ c(h-d)**
  - Don't care about constants... Ignoring them below...
- Cost to heapify **all** nodes at depth **d** is **≤ 2^d (h-d)**

Showing $O(n)$

- So, cost to heapify **all** nodes over **all** depths is:

$$C \leq \sum_{d=0}^{h-1} 2^d \times (h-d)$$

. Let $i = h-d$; so $d = h-i$

$$\leq \sum_{i=1}^{h} 2^{h-i} \times i$$

$$\leq 2^h \sum_{i=1}^{h} \frac{i}{2^i}$$

$$\leq 2^h \sum_{i=0}^{\infty} \frac{i}{2^i}$$

note $2^h = 2^{\lfloor \log_2 n \rfloor} \leq n$

$$\leq n \sum_{i=0}^{\infty} \frac{i}{2^i}$$

(See next page)

$$< 2n$$

Showing $O(n)$

Recall that: $\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$

So for $|x|<1$ and $n \to \infty$: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

By taking the derivative and multiplying by x we get

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

For $x = \frac{1}{2}$, $\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{1/2}{(1-1/2)^2} = 2$.