Unit #2: Priority Queues

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2: January-April 2017

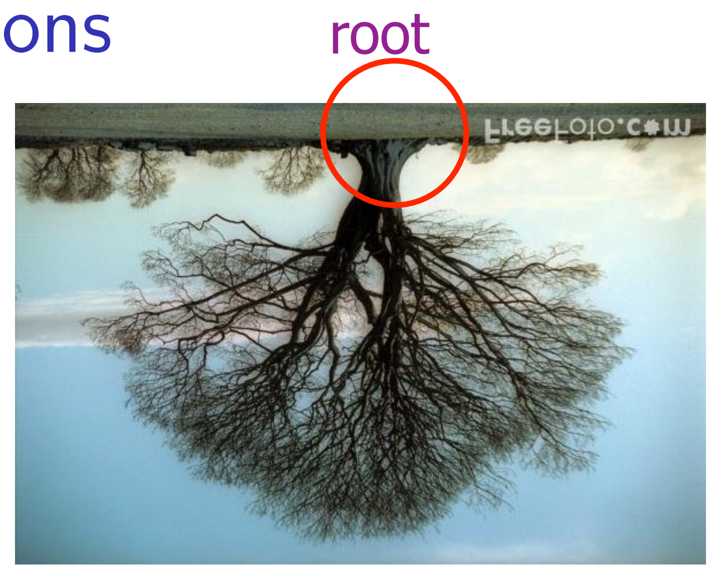
Unit Outline

- Rooted Trees (Briefly)
- Priority Queue ADT
- Heaps
  - Implementing a Priority Queue ADT
  - Operations on a Heap
  - Building a Heap via Heapify
  - Analysis of Operations
  - Brief Introduction to *d*-Heaps

Learning Goals

- Define terminology about trees.
- Provide examples of appropriate applications for priority queues and heaps.
- Manipulate data in heaps.
- Describe and apply the Heapify algorithm, and analyze its complexity.

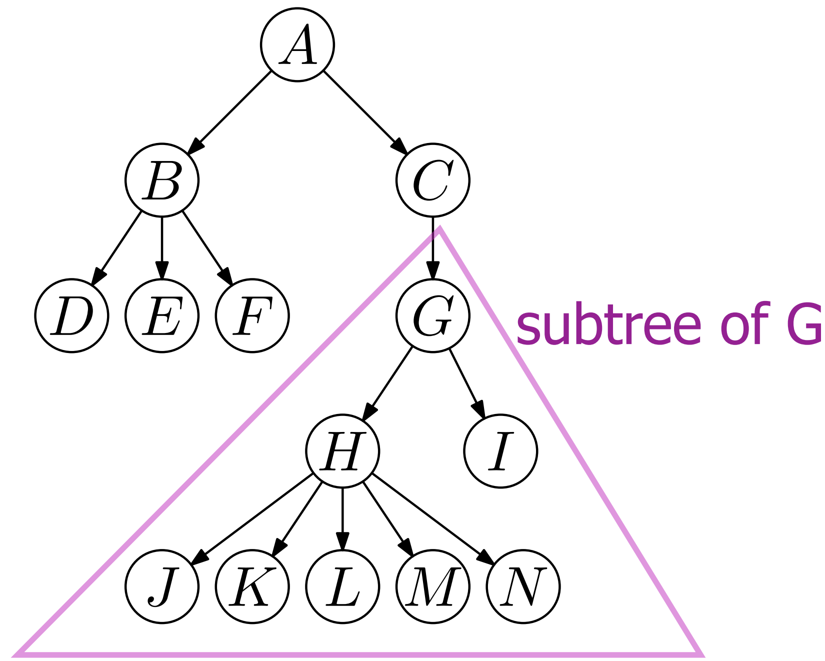
Rooted Trees and Some Applications



- Family Trees
- Organization Charts
- Classification Trees
  - What kind of flower is this?
  - Is this mushroom poisonous?
- File Directory Structure
  - Folders and Subfolders in Windows
  - Directories and Subdirectories in UNIX
- Non-Recursive Call Graphs
- Indexes in Database Systems

Tree Terminology: Examples

```
struct Node {
  string data;
  Node *left, *right;
}
```



root: A

leaf: D E F I J ... N

child of __A__ : B C

parent of __H__ : G

sibling: J K

ancestor of __N__ : H G C A

descendent of __C__ : G H I J K ... N

subtree of __G__ : G and all descendent

Tree Terminology Reference

"nodes" or "vertices"

"edges" or "arcs"

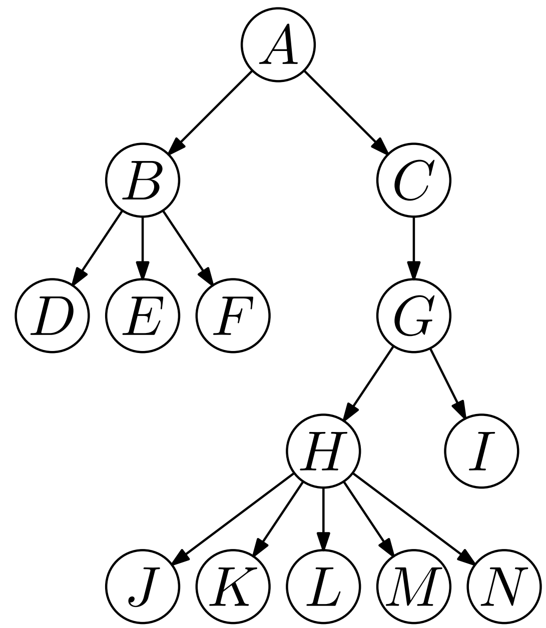
root: the single node with no parent

leaf: a node with no children

child: a node pointed to by me

parent: the node that points to me

sibling: another child of my parent



ancestor: my parent or my parent's ancestor

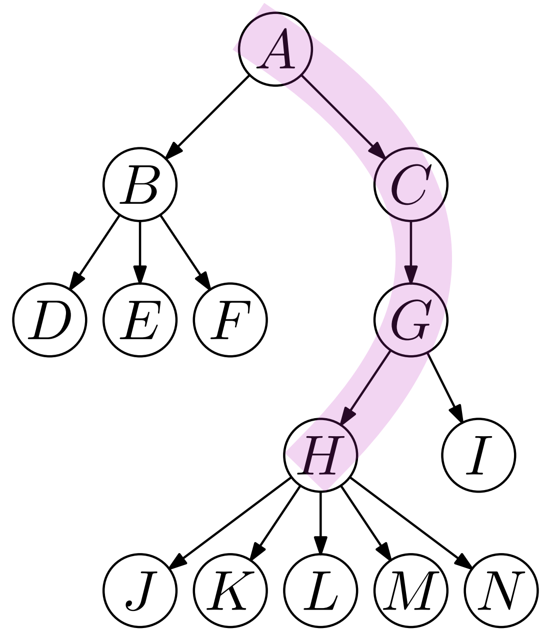
descendent: my child or my child's descendent

subtree: a node and its descendents

More Tree Terminology

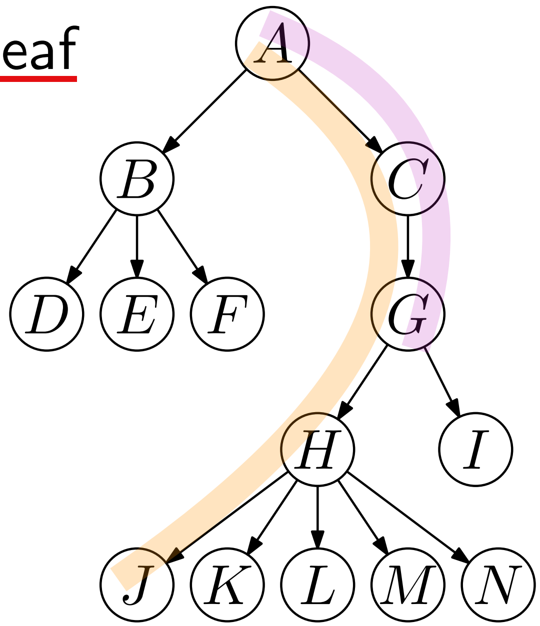
depth: number of edges on path from root to node

depth of H? 3



More Tree Terminology

height: number of edges on longest path from a given node to its furthest descendent; or, when speaking of the whole tree: number of edges on longest path from root to leaf



height of tree? = height of root = 4

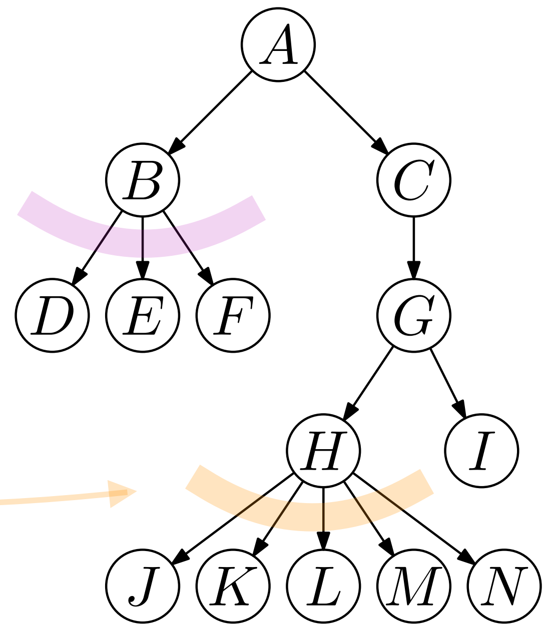
height of G? 2

More Tree Terminology

(downward) degree: number of children of a given node

degree of B? 3

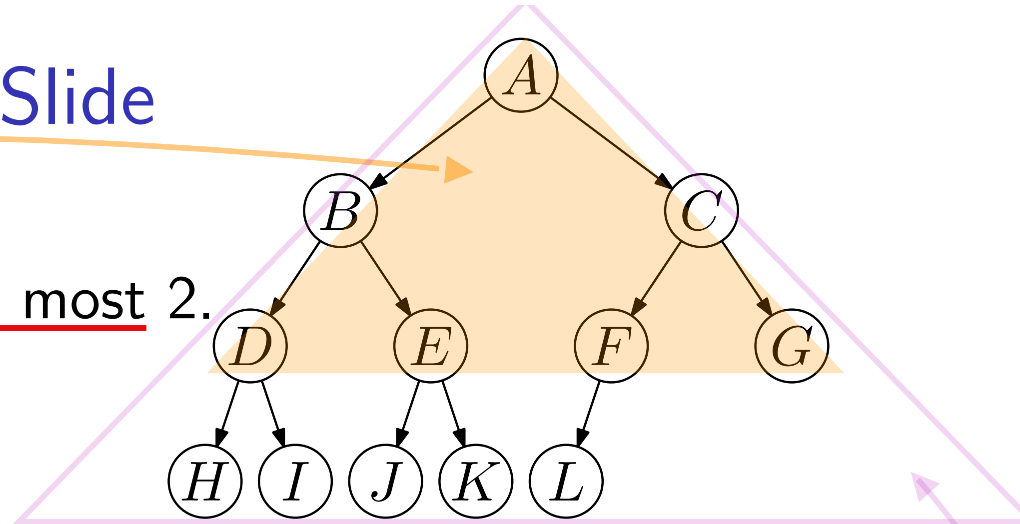
Highest degree here? 5



Questions for next page (slide 10):
Is the tree above …

Binary?	no
d-ary?	yes, d = 5
Full?	no
Complete?	no
Nearly complete?	no

One More Tree-Terminology Slide



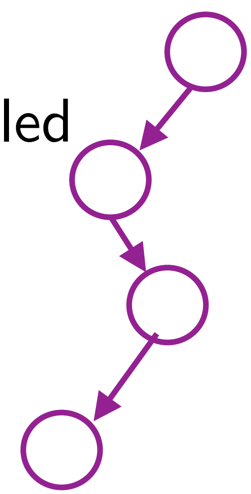
binary: Each node has degree at most 2.

d-**ary:** The degree is at most *d*.

full: Each internal (non-leaf) node has the maximum number of children (2 in the case of a binary tree).

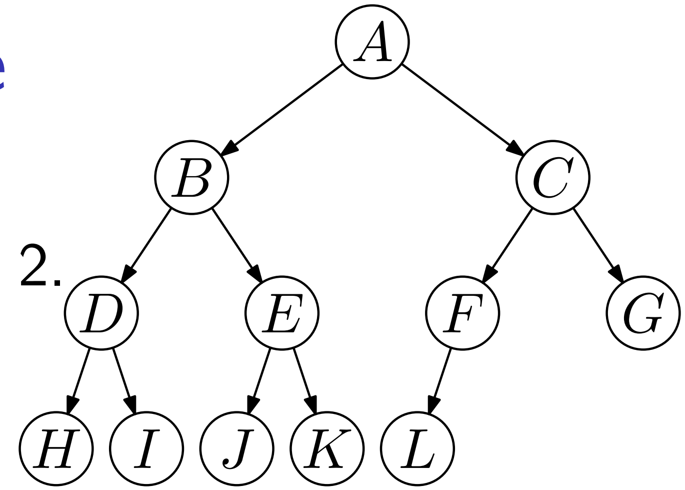
complete: It has as many nodes as possible for its height (i.e., each row is filled in).

nearly complete: Each row, except possibly the last one, is filled in, and all nodes in the last row are as far left as possible. (Warning: Some authors like Koffman/Wolfgang call this a *complete* tree. We'll stick with *nearly complete*.)



Also a tree

One More Tree-Terminology Slide



binary: Each node has degree at most 2.

n: # of nodes in a binary tree of height h

h + 1 ≤ n ≤ 2^(h+1) - 1

e.g. with h=3:
n ≤ 2^(3+1) - 1, so n ≤ 15
also 3+1 ≤ n, so 4 ≤ n. Thus, 4 ≤ n ≤ 15

complete: It has as many nodes as possible for its height (i.e., each row is filled in).

n = 2^(h+1) - 1
e.g. with h=3: n = 15

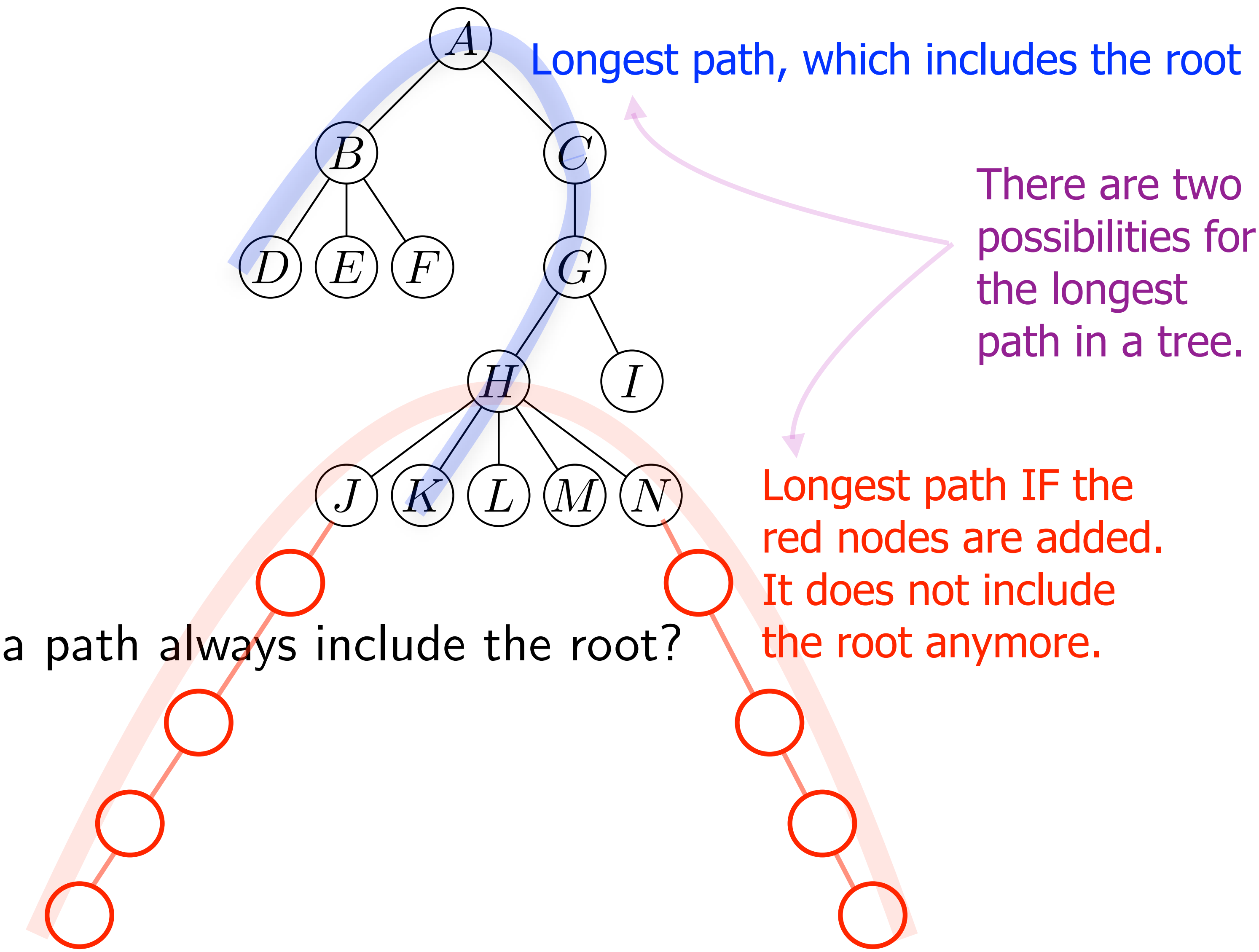
nearly complete: Each row, except possibly the last one, is filled in, and all nodes in the last row are as far left as possible. (Warning: Some authors like Koffman/Wolfgang call this a *complete* tree. We'll stick with *nearly complete*.)

2^h ≤ n ≤ 2^(h+1) - 1

e.g. with h=3: 8 ≤ n ≤ 15

If a nearly complete tree has n nodes, what is the h?
2^h ≤ n < 2^(h+1)
h ≤ lg n < (h+1)
h = floor(lg n) (ie. the integer part of lg n)

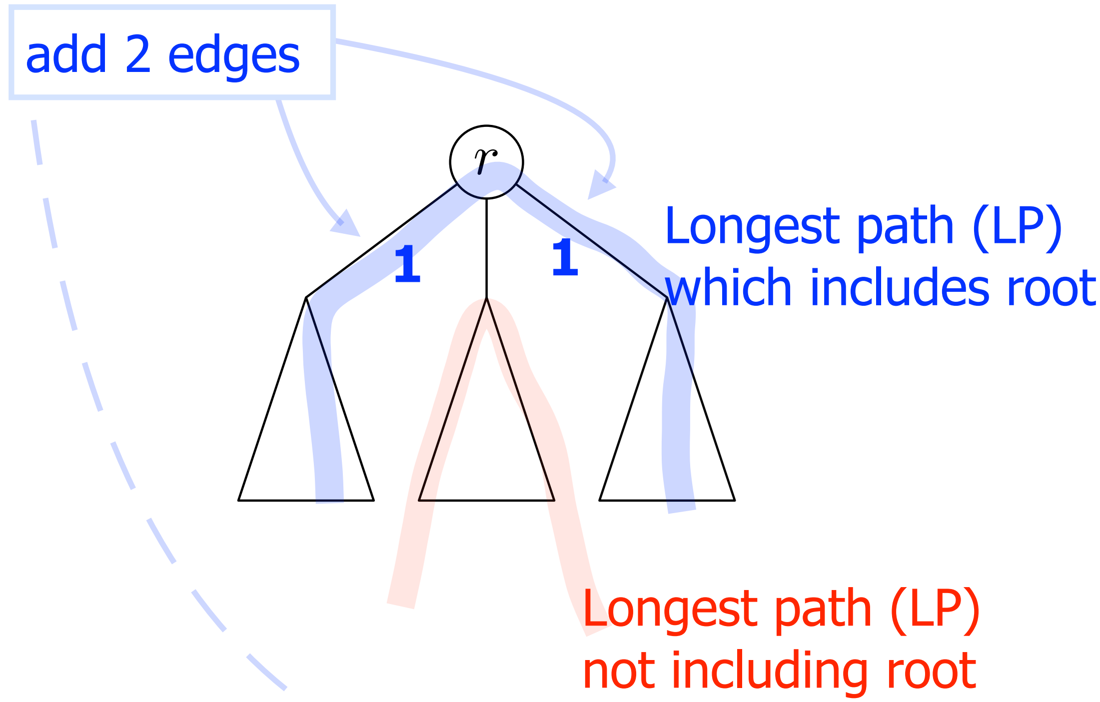
Example: Finding the Longest Undirected Path in a Tree



Does such a path always include the root?

Longest Path

An algorithm to find the longest *undirected* path in a tree:



LongestPath(r) = 0 if r has no children

LongestPath(r) = 1 if r has one child

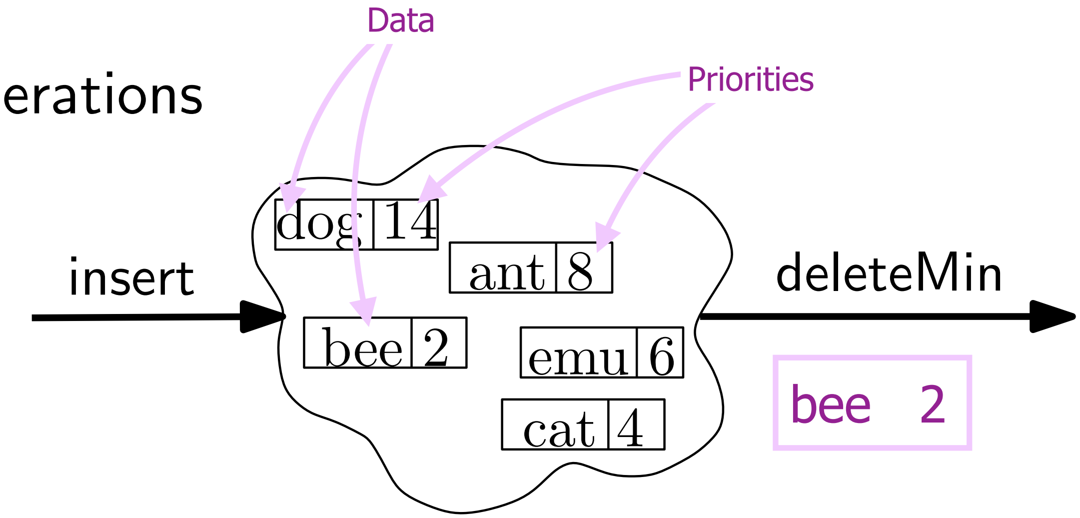
LongestPath(r) = MAX [MAX [Height(c) + Height(d)] + 2 where c ≠ d are children of r ; MAX [LongestPath(c)] where c is a child of r]

Back to Queues

- Applications
  - Ordering jobs/processes on a CPU
  - Simulating events
  - Picking the next search site
- But we don't necessarily want FIFO. You can choose your order, according to some carefully thought-out priority. Maybe:
  - *Shorter* jobs should go first.
  - *Earliest* (simulated time) events should go first.
  - *Most promising* sites should be searched first.

Priority Queue ADT

- Priority Queue Operations
  - create
  - destroy
  - insert
  - deleteMin
  - is_empty



- Priority Queue Property (in a minimum priority queue): For two elements in the queue, x and y, if x has a lower priority value than y, x will be deleted before y when performing a deleteMin operation.

Applications of a Priority Queue

- Hold jobs for a printer in order of length.
- Store packets on network routers in order of urgency.
- Simulate events.
- Select symbols for compression.
- Sort numbers.
- Anything *greedy*: In this case, an algorithm makes the "locally best choice" (not necessarily the overall best choice) at each step.

Priority Queue Data Structures

Consider two data structures: Array and Linked List

- Unsorted List
  - insert time: $\Theta(1)$ — Add new item to Array or Linked List
  - deleteMin time: $\Theta(n)$ — Find item in the unsorted Array or Linked List

- Sorted List
  - insert time: $\Theta(n)$

	Find position:		Insert at position:		
Array:	$\Theta(\log n)$	+	$\Theta(n)$	=	$\Theta(n)$
Linked List:	$\Theta(n)$	+	$\Theta(1)$	=	$\Theta(n)$

  - deleteMin time: $\Theta(1)$ — Remove 1st item in the sorted Array or Linked List

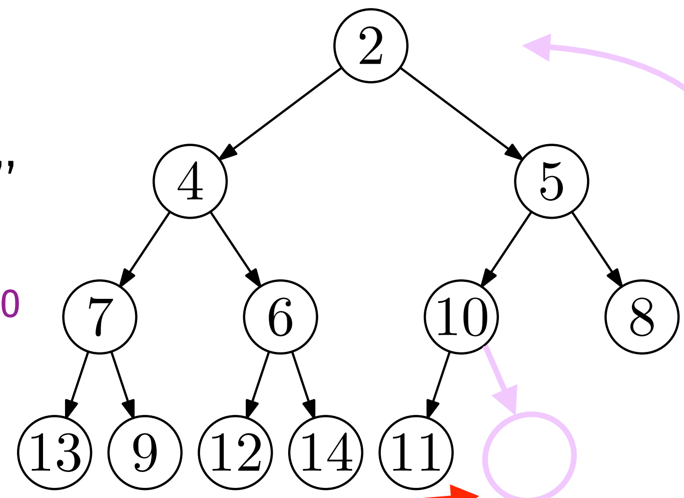
Binary Heap Priority Queue Data Structure

Heap-Order Property: parent's key $\leq$ children's key (we often call this a *minimum* heap)

- minimum is always at the top

Structure Property: "nearly complete tree"

- depth is always $O(\lg n)$: See proof on slide 10
- next open location is always known



WARNING: This has no similarity to the memory "heap" we talk about when using C++'s `new` operator.

```
struct Node {
  string data;
  int priority;
  Node *left, *right, *parent;
}
```

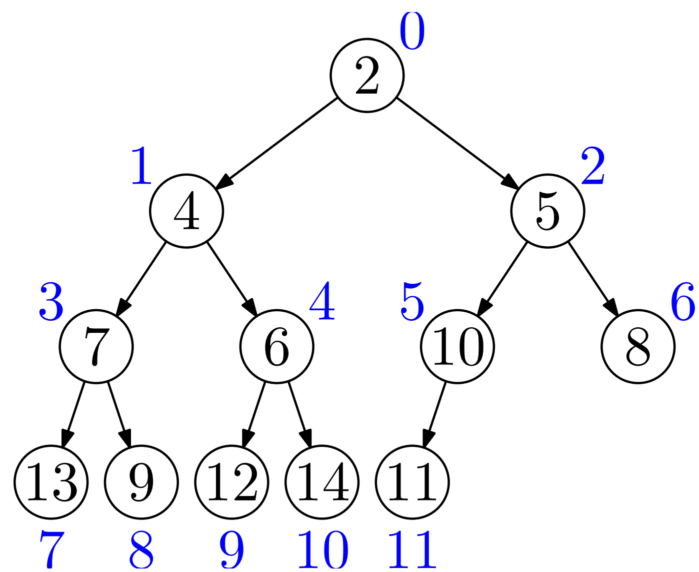
In illustrations usually:
◇ Only priorities are shown
◇ The "data" for each node is omitted to avoid clutter

Nifty Storage Trick: use an array to represent a heap

Navigation using indices:

- left_child(i) = 2i + 1
- right_child(i) = 2i + 2
- parent(i) = ⌊(i-1)/2⌋ = ⌈i/2⌉ - 1
- root = 0
- next free position = n

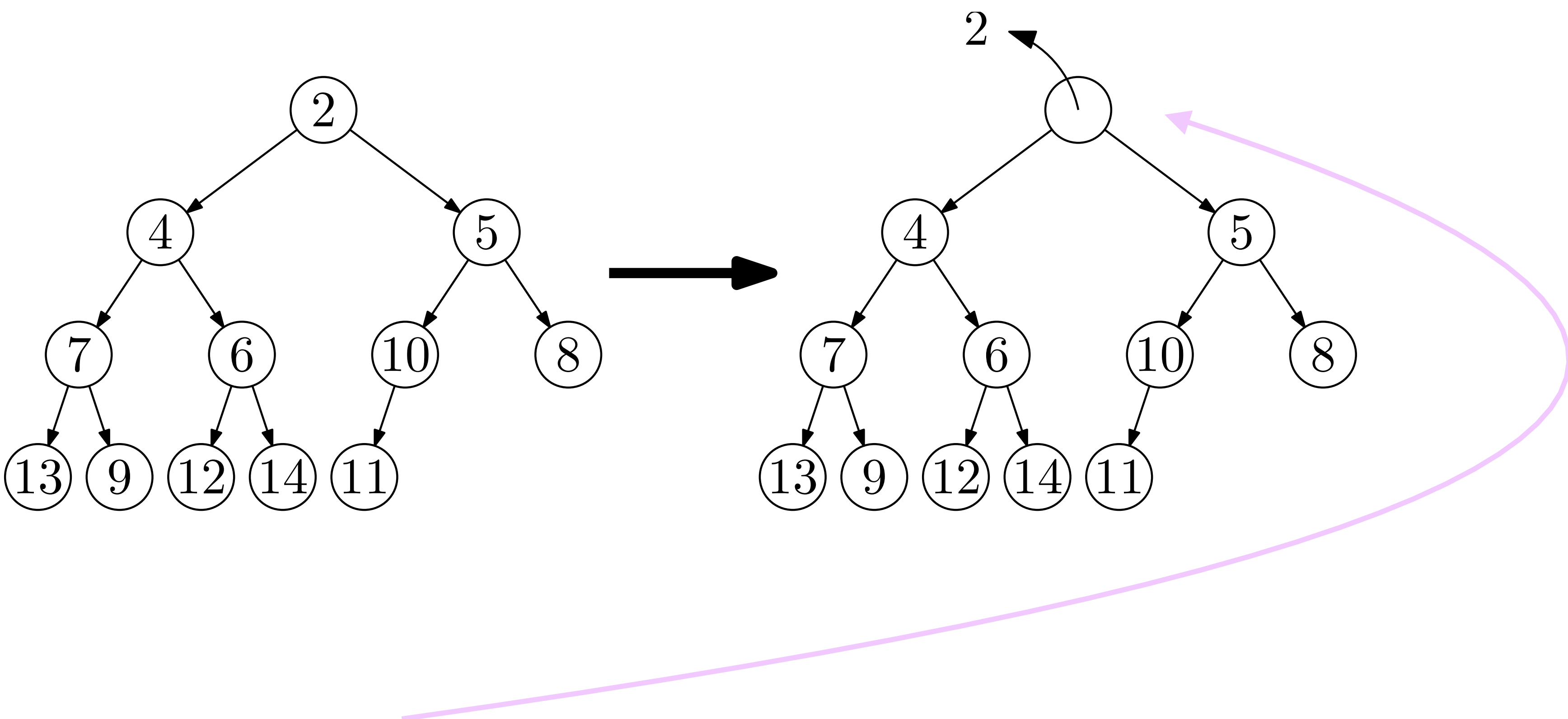
size = n



No gaps if "nearly complete"

	0	1	2	3	4	5	6	7	8	9	10	11	12
Heap:	2	4	5	7	6	10	8	13	9	12	14	11	

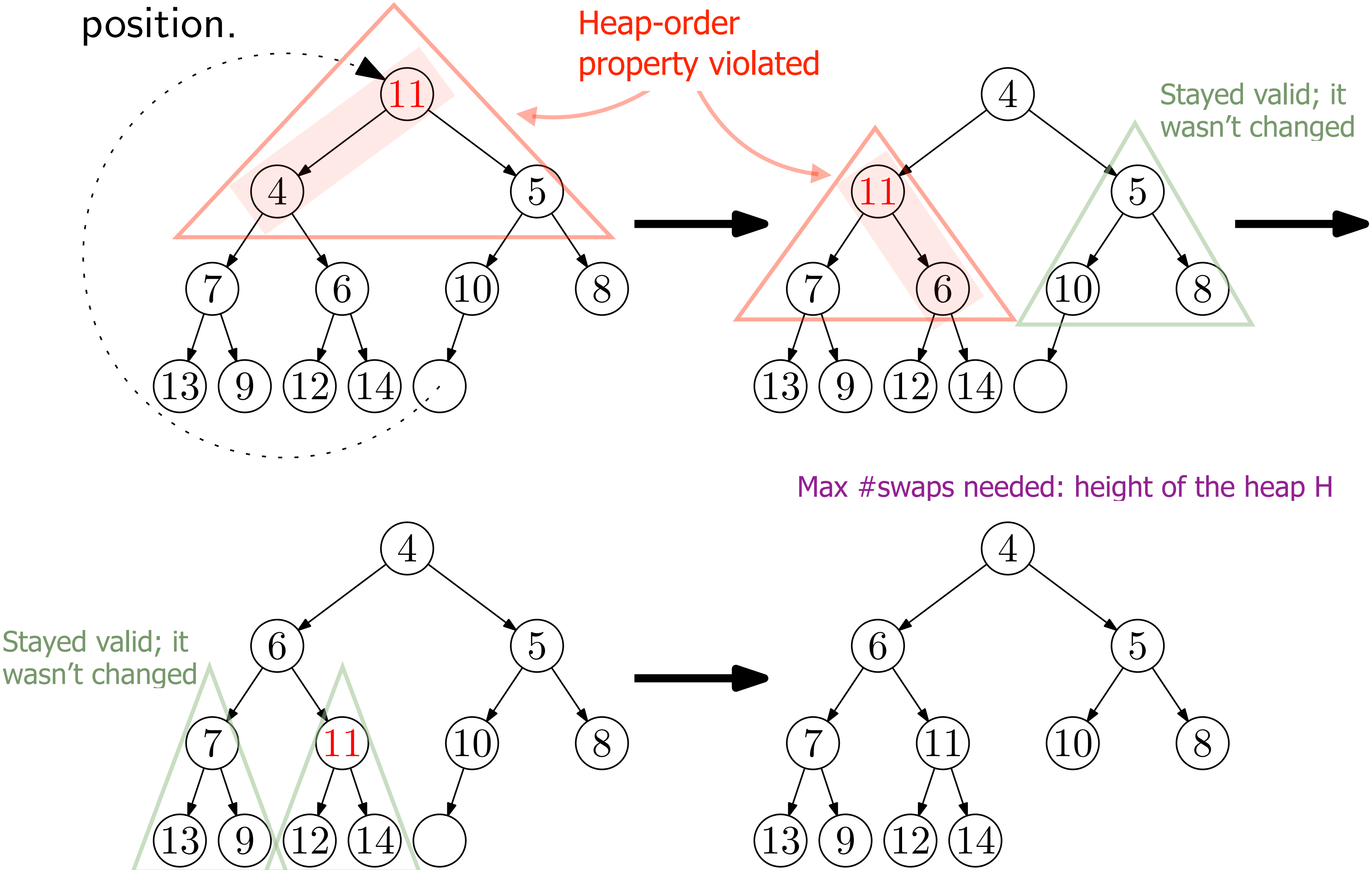
deleteMin



Invariants violated! It's no longer a "nearly complete" binary tree.

Swap (Heapify) Down

Move last element to the root, and then swap it down to its proper position.



deleteMin Code

```
int deleteMin() {
  assert(!isEmpty());
  int returnVal = Heap[0];
  Heap[0] = Heap[n-1];
  n--;
  swapDown(0);
  return returnVal;
}
```

Constant time

```
void swapDown(int i) {
  int s = i;
  int left = i * 2 + 1;
  int right = left + 1;
  if( left < n &&
      Heap[left] < Heap[s] )
    s = left;
  if( right < n &&
      Heap[right] < Heap[s] )
    s = right;
  if( s != i ) {
    int tmp = Heap[i];
    Heap[i] = Heap[s];
    Heap[s] = tmp;
    swapDown(s);
  }
}
```

checks heap boundary

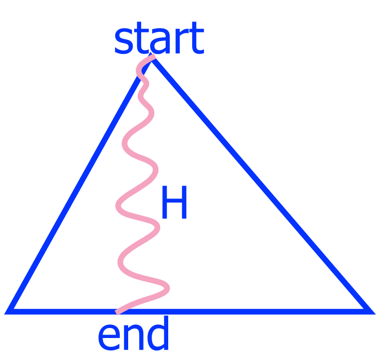
false at leafs

swap nodes i and s

s > 2*i

Runtime:

Another approach:



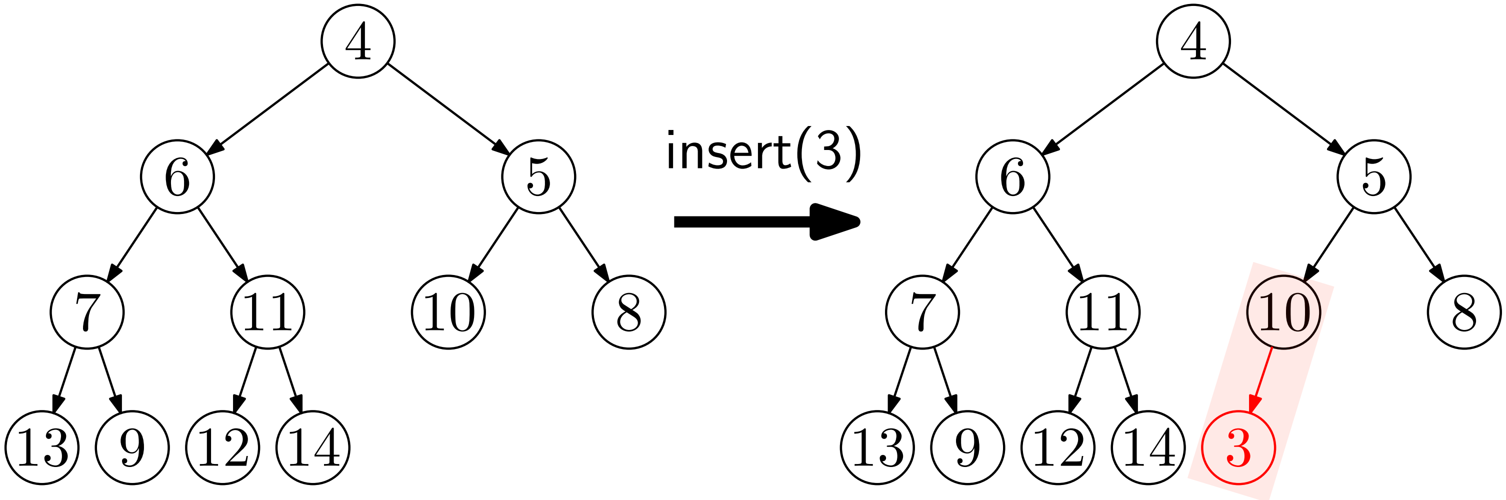
#swapDown ∈ O(H)=O(lg n)

Example recursive calls:

swapDown(0);
swapDown(1);
swapDown(3);
swapDown(7);
swapDown(15);

#swapDown ∈ O(lg n)

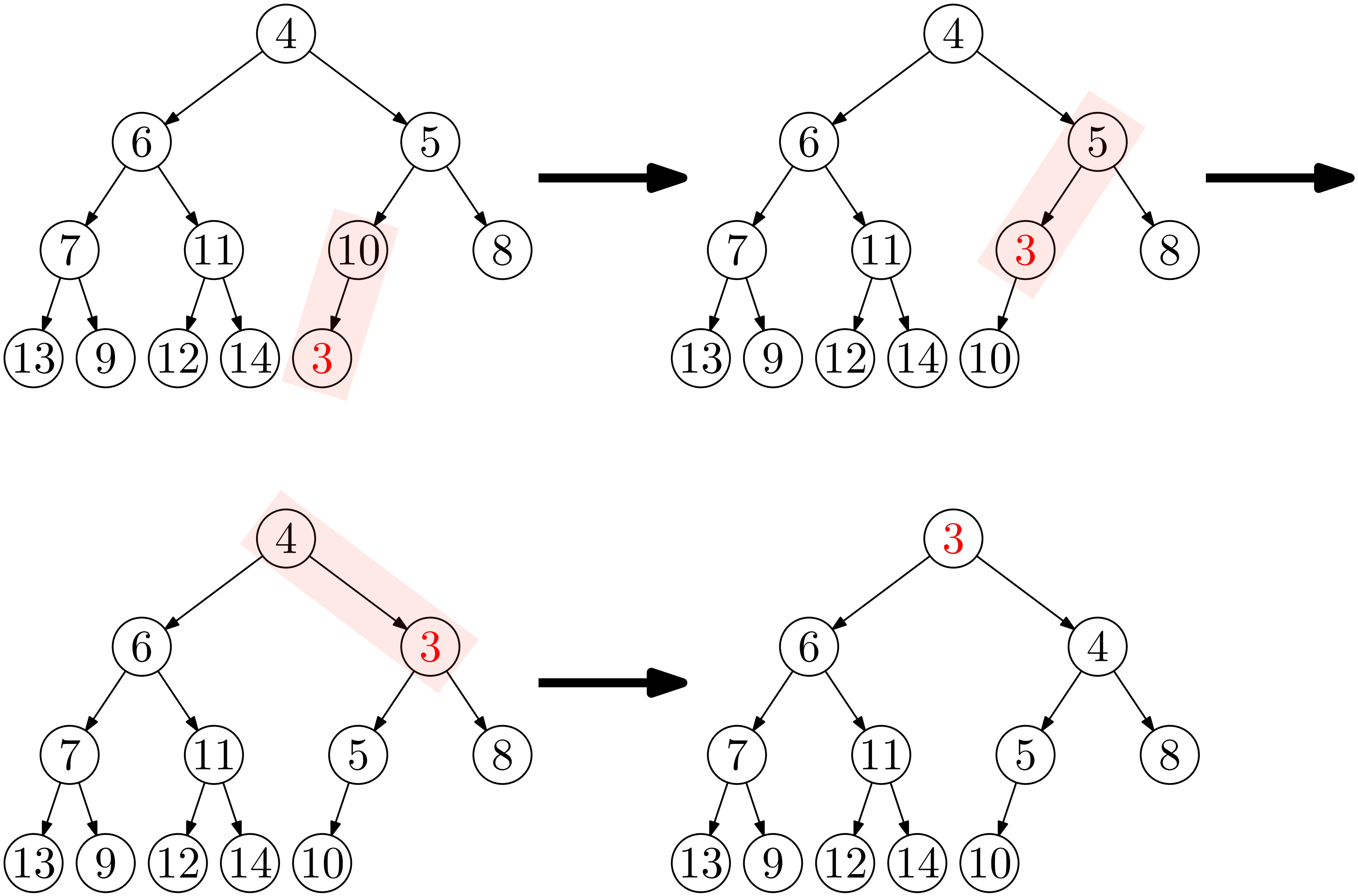
Inserting a New Node



Invariant violated! Child has smaller key than parent.

Swap (Heapify) Up

Begin by putting the new element last, then swap it up to its proper position.



insert Code

```
void insert(int x) {
  assert(!isFull());
  Heap[n] = x;
  n++;
  swapUp(n-1);
}
```

```
void swapUp(int i) {
  if( i == 0 ) return;
  int p = (i - 1)/2;
  if( Heap[i] < Heap[p] ) {
    int tmp = Heap[i];
    Heap[i] = Heap[p];
    Heap[p] = tmp;
    swapUp(p);
  }
}
```

Constant time

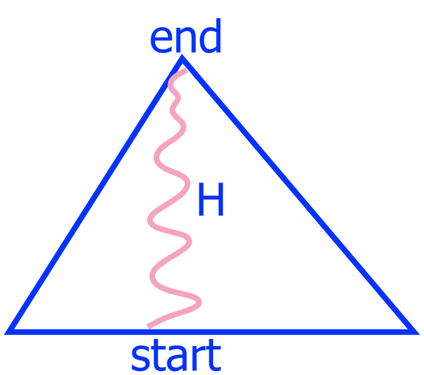
p < i / 2

Example recursive calls:

swapUp(11);
swapUp(5);
swapUp(2);
swapUp(0);

#swapUp ∈ O(lg n)

Runtime:



#swapUp ∈ O(H) = O(lg n)

Heapify: Build a Heap from an Array

1. Start with the input array.

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

First consider a rather naive approach using "insert" (from slide 24):
Starting from an empty heap, insert input array elements into the heap one by one.

```
for( i = 0; i < n; i++ )
      insert(i);
```

log i

$T(n) = \log 1 + \log 2 + \log 3 + \ldots + \log n$
$= \log (1 \times 2 \times 3 \times \ldots \times n)$
$= \log (n!)$
$\in \Theta (n \log n)$ as we saw in lec01 notes

Can we do better? Yes!

Consider the entire input array as an invalid heap which violates the heap-order property

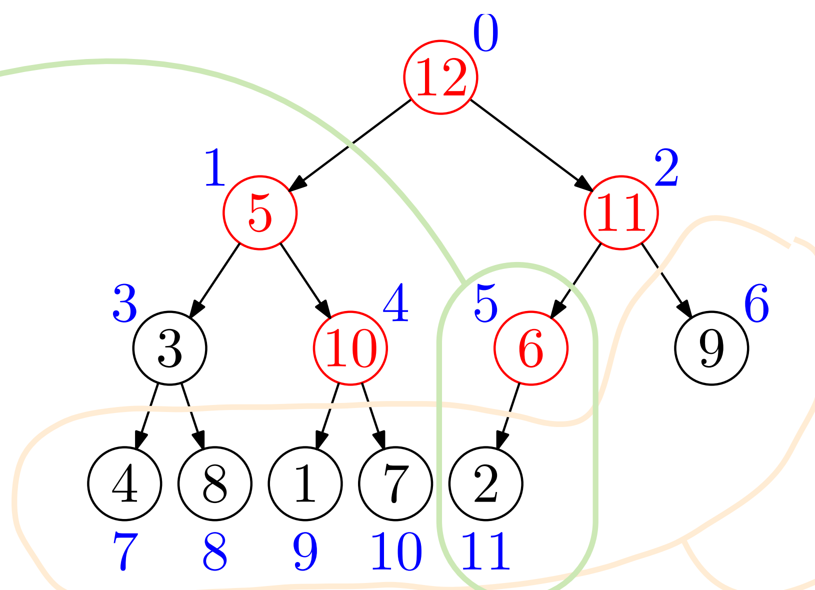
Then, "fix" the heap-order property one by one, but starting from the end and going up (see next slide ...)

Heapify: Build a Heap from an Array

1. Start with the input array.

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

Start from the first node with some children, i.e. the parent of last node: i = 12/2 - 1 = 5



Invariant violated!

Each Leaf is already a proper heap because it has no children

2. Fix the heap-order property, starting from the bottom, and going up. Use `swapDown`.

```
for( i = n/2 - 1; i >= 0; i-- )
      swapDown(i);
```

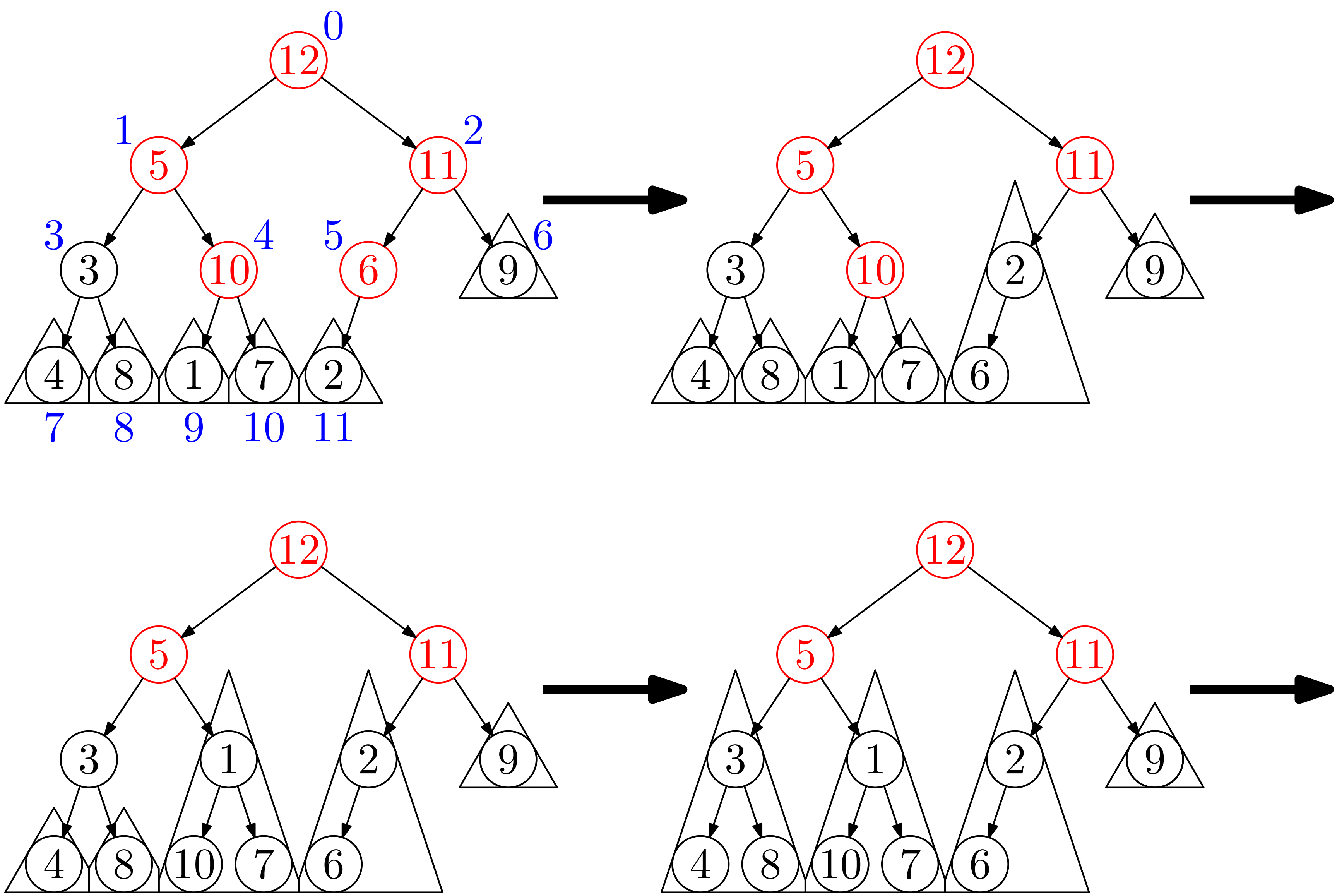
Thus, this would also work:

```
for( i = n; i >= 0; i-- )
      swapDown(i);
```

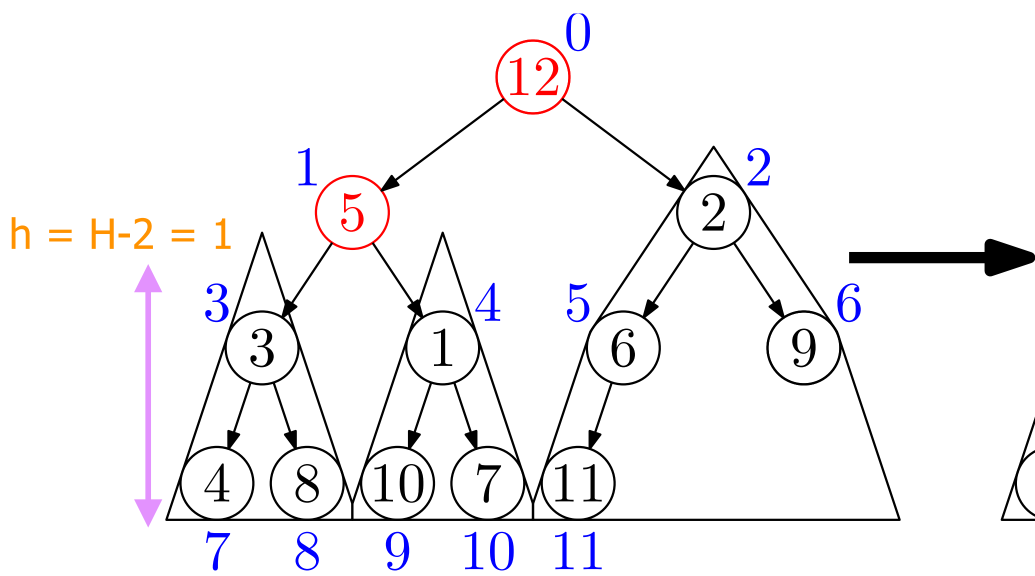
But it makes wasteful calls to swapDown (2 times more calls)

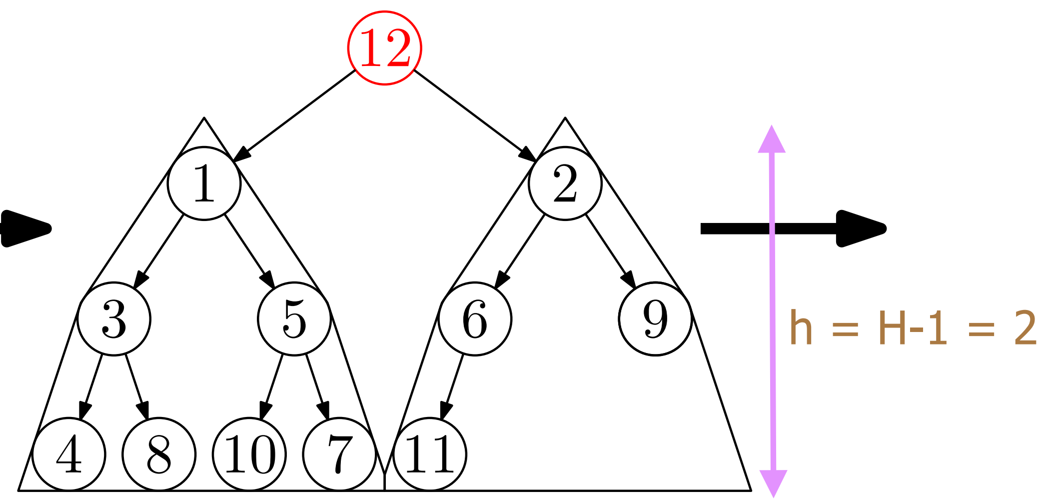
Heapify Example...

△ a triangle denotes a valid heap



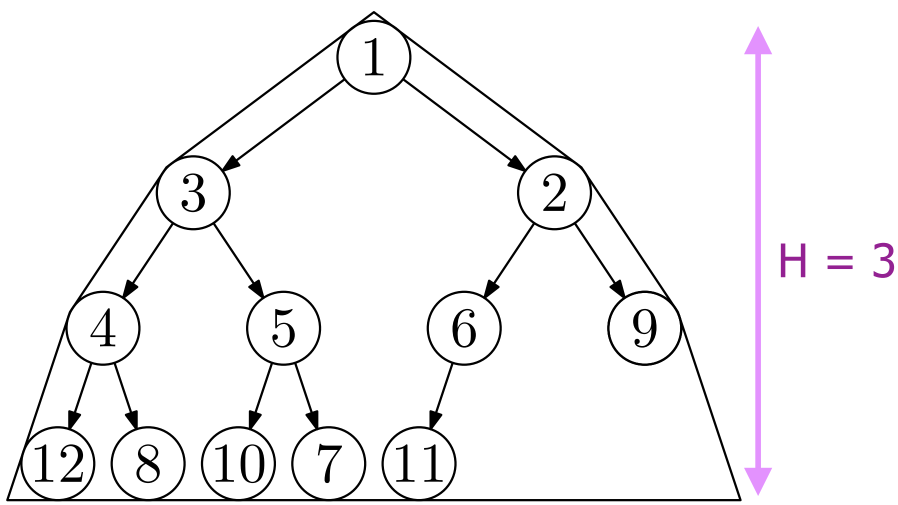
Heapify Example





swapDown is called ≤ 2 times on heaps of height H-1

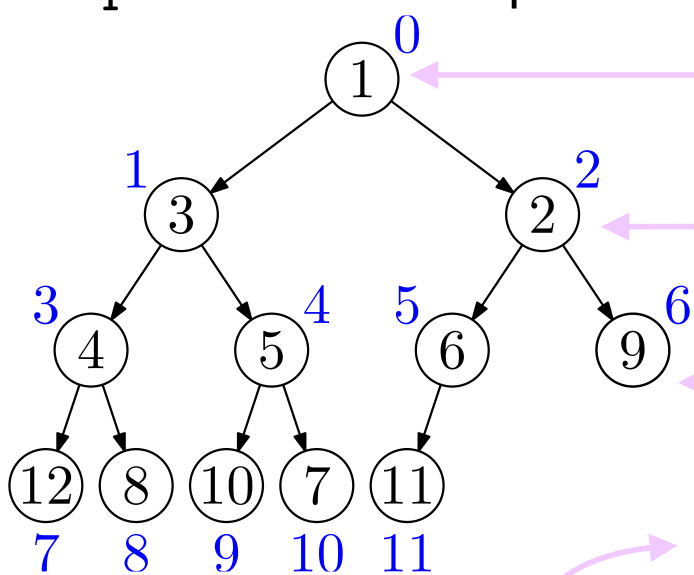
swapDown is called ≤ 4 times on heaps of height H-2



swapDown is called once on a heap of height H

Heapify Runtime

`swapDown` on a heap of height h takes at most ___h___ steps.



h = H = 3

h = H-1 = 2

h = H-2 = 1

Let H be the height of the heap.

H = ⌊lg n⌋

by heapify

`swapDown` is called	once	on heap of height	H
	≤ 2 times	on heap of height	$H - 1$
	≤ 4 times	on heap of height	$H - 2$
	$\vdots \leq 2^{H-h}$ times	on heap of height	h
	$\leq 2^{H-1}$ times	on heap of height	1

Total # steps $\leq \sum_{h=1}^{H} h 2^{H-h} = 2^H \sum_{h=1}^{H} h/2^h \leq 2^{H+1} = O(n)$

< 2 (see next slide)

$$\sum_{h=1}^{H} h / 2^h < 1/2 + 2/4 + 3/8 + 4/16 + \ldots = 2$$

call it S

So

because

$$2S = 1 + 2/2 + 3/4 + 4/8 + \ldots$$
$$S = \quad 1/2 + 2/4 + 3/8 + \ldots$$
$$2S - S = 1 + 1/2 + 1/4 + 1/8 + \ldots$$
$$S = 1 + 1/2 + 1/4 + 1/8 + \ldots$$
$$S = 2$$

because

$$S = 1 + 1/2 + 1/4 + 1/8 + \ldots$$
$$S/2 = \quad 1/2 + 1/4 + 1/8 + 1/16 + \ldots$$
$$S - S/2 = 1$$
$$S/2 = 1$$
$$S = 2$$

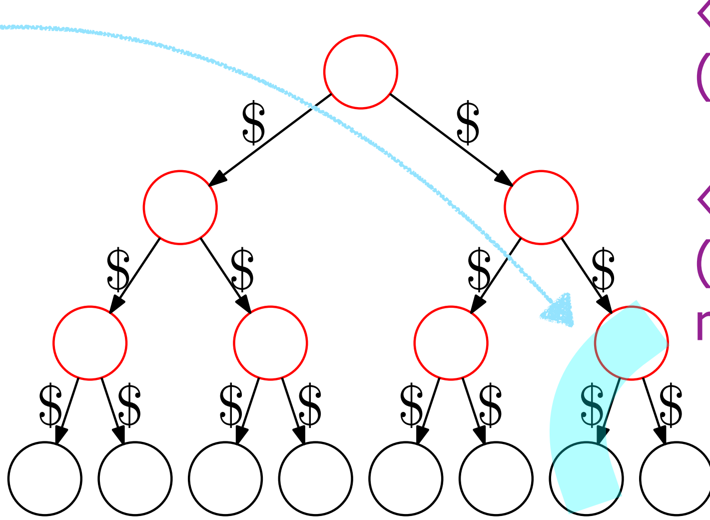
Heapify Runtime: Charging Scheme

◇ When two nodes are swapped, $1 is charged

◇ Each edge only has $1

◇ Thus, there can only be one swap for each edge

◇ But still the worst case tree can be heapified! (see next slide)



Worst case:

◇ heap is a "complete" tree (i.e., all rows are filled in)

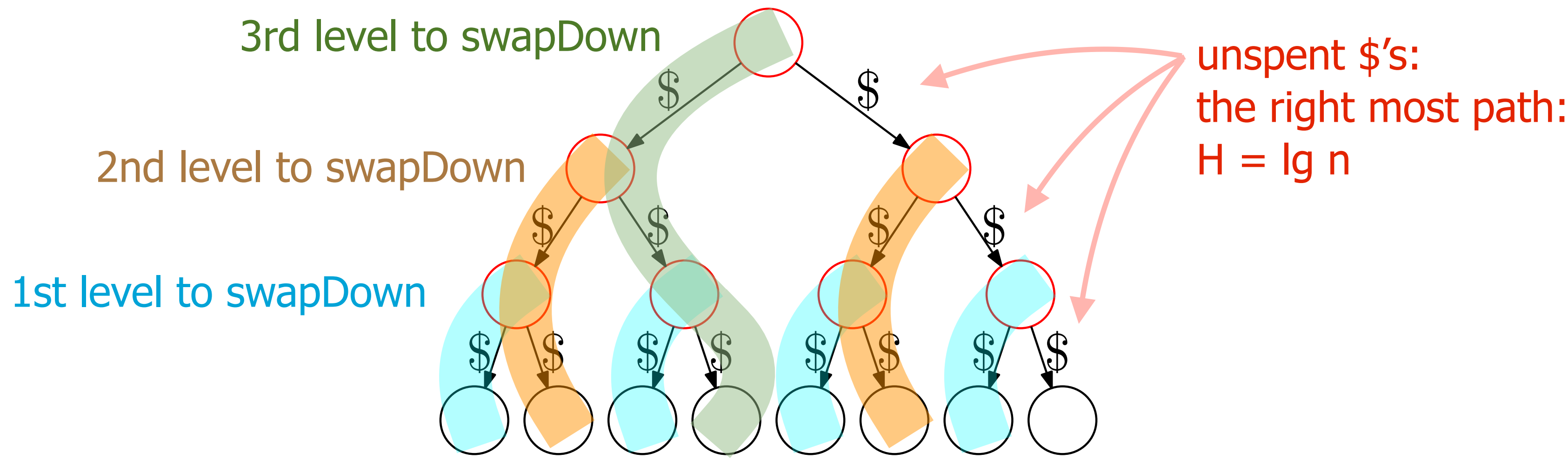
◇ all leafs have high priorities (i.e., have small values in a minimum heap)

Possible violations. How much time to fix them?

Place a dollar on each edge of the heap. One dollar pays for one step of `swapDown`. By induction, we can show that when `swapDown` is called on a node v, both children of v have a path (the rightmost path) to a leaf that is uncharged. The edges on the left child's rightmost path plus the edge to the left child pay for the steps of `swapDown` at v. The edges on the right child's rightmost path plus the edge to the right child form the uncharged path available to the parent of v.

Heapify Runtime: Charging Scheme

total $'s = #edges = n - 1



#swaps = (total $'s) - (unspent $'s)
= (#edges) - (H)
= (n - 1) - (lg n)
= n - 1 - lg n
≤ n
∈ O(n)

Thus this second proof has the same results as the first proof that we saw on slide 28.

Thinking about Binary Heaps

Observations

- Finding a child/parent index is a multiply/divide by two (i.e. 2i or i/2) operation. left = 2i+1, p = ⌊(i-1)/2⌋ recall that
- Both deleteMin and the subsequent insert might access far-apart array locations. seperated by large gaps
- deleteMin accesses all children of visited nodes. swapDown
- insert accesses only the parent of visited nodes. swapUp
- insert is at least as common as deleteMin.
  Generally true: you can delete something that has already been inserted
  But not necessarily: you may start with heapify and never insert before delete

Realities

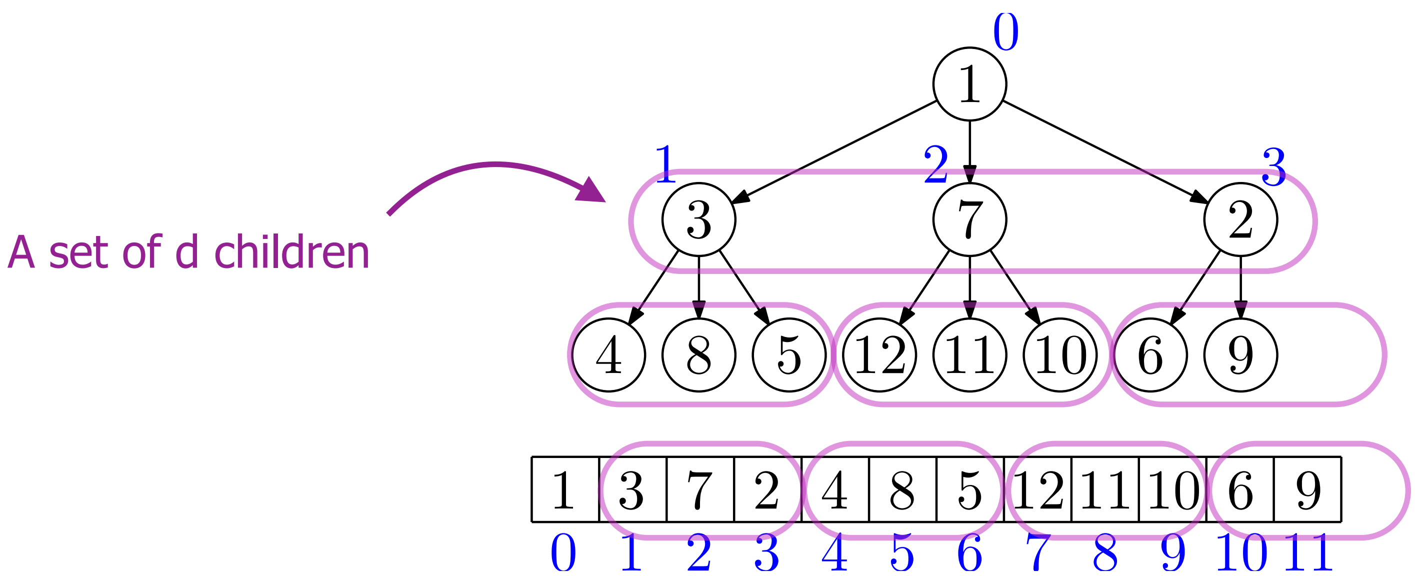
- Division and multiplication by powers of two are fast.
- Far-apart array accesses can ruin cache performance.
- With large datasets, disk I/O dominates CPU time.

Using bit shifts, which are fast; e.g.:

```
i*2  ==  i<<1
i*4  ==  i<<2
i*8  ==  i<<3
i/2  ==  i>>1
i/4  ==  i>>2
i/8  ==  i>>3
...
```

Solution: *d*-Heaps

These are ^nearly complete *d*-ary trees (representable by an array) with a heap-order property.



Good choices for *d*:

- fit one set of children on a memory page/disk block
- fit one set of children in a cache line
- optimize performance based on ratio of inserts/deleteMins
- make *d* a power of two for efficiency

d-Heap Navigation

So all children: d*i + 1 through d*i + d

- *j*th-child(*i*) = d*i + j
- parent(*i*) = ⌊(i-1)/d⌋
- root = 0
- next free position = n