Unit #2: Priority Queues

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2: January-April 2017

Ed's class:
annotated slides

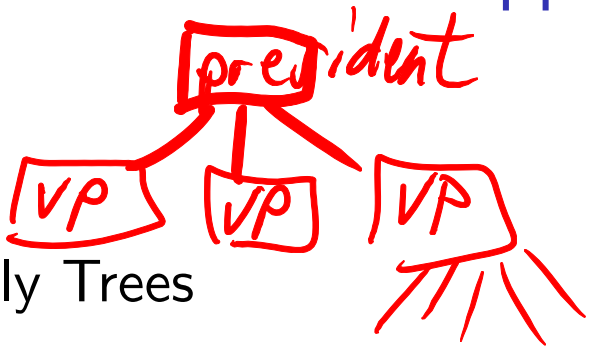
Unit Outline

- 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

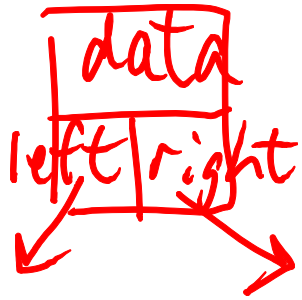


- Family Trees
- Organization Charts
- Classification Trees
  - What kind of flower is this?
  - Is this mushroom poisonous?
- File Directory Structure
  - Folders and Subfolders in Windows
  - Directories and Subdirectories in UNIX
- Non-Recursive Call Graphs
- Indexes in Database Systems

- cancer, non-cancer
- types of hockey players
- animals
- objects in general

Tree Terminology: Examples

Binary Node



↳ need not be binary

root: A (no parent)

leaf: D, E, F, J–N, I

child of B : D, E, F

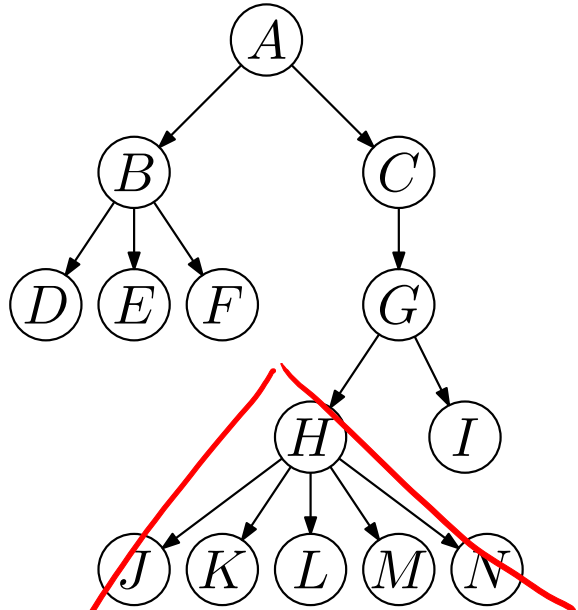
parent of G : C

sibling: of F : D, E

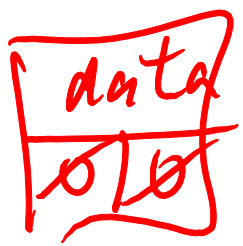
ancestor of L : H, G, C, A

descendent of G : H, I, J–N

subtree of H : H, J–N



leaf node (binary)



Tree Terminology Reference

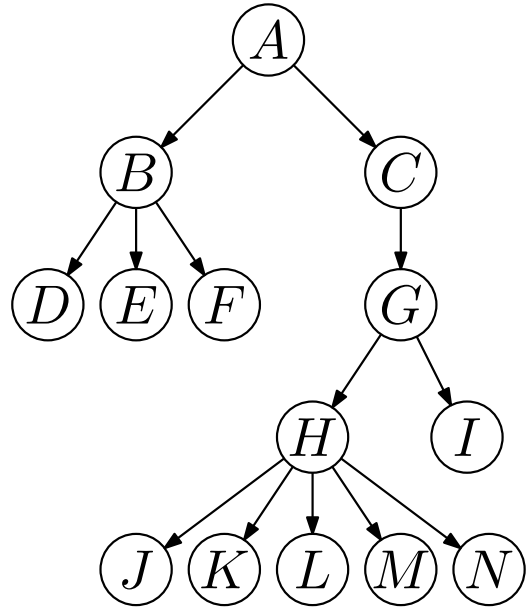
root: the single node with no parent

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

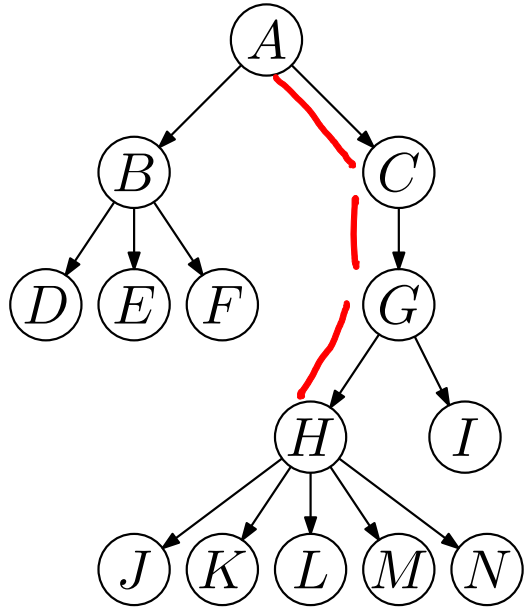
subtree: a node and its descendents

More Tree Terminology

depth: number of edges on path from root to node

depth of H? 3

depth of A? 0

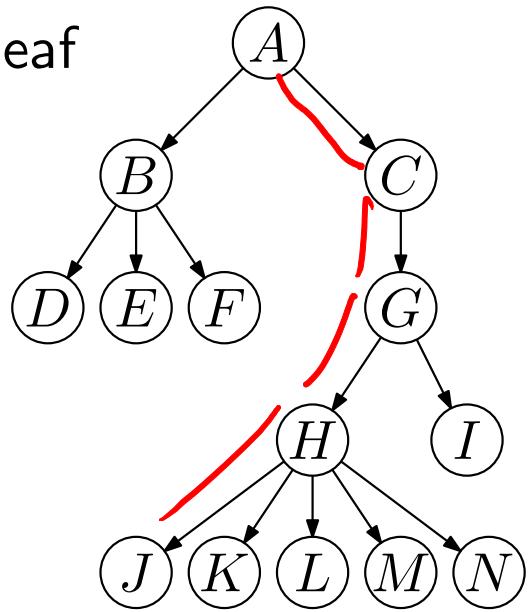


More Tree Terminology

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

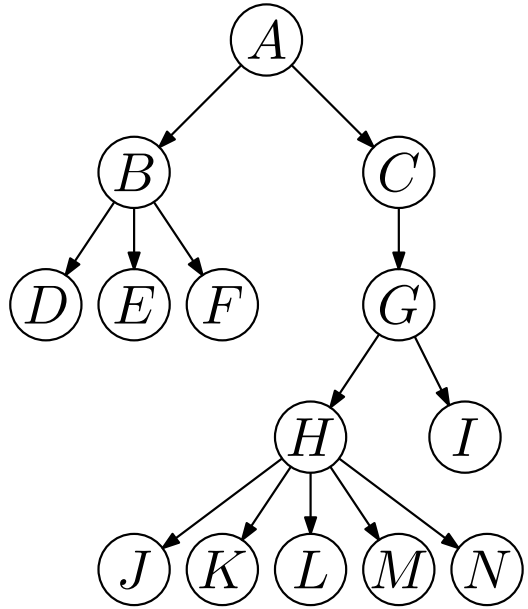
fan out

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

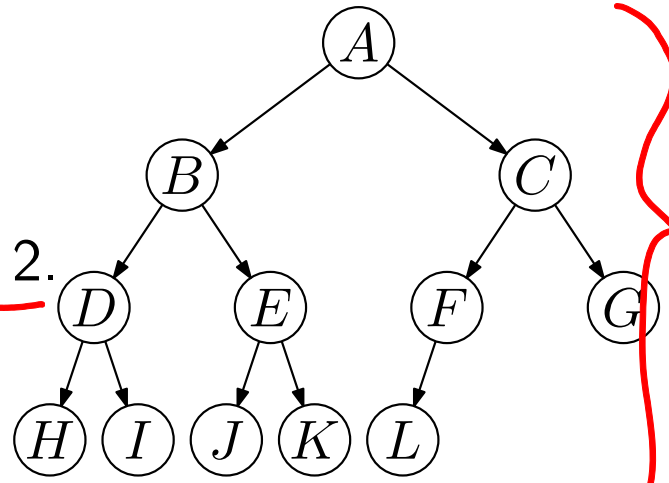
degree of B? 3

degree of I? 0

degree of G? 2



One More Tree-Terminology Slide



binary: Each node has degree at most 2.

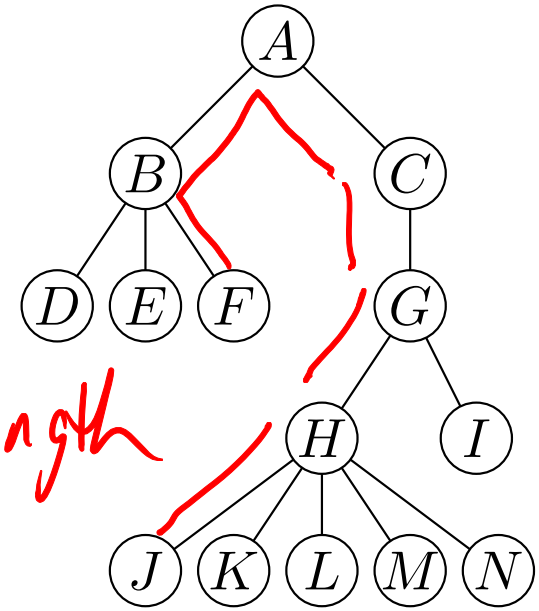
d-**ary:** The degree is at most *d*.

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

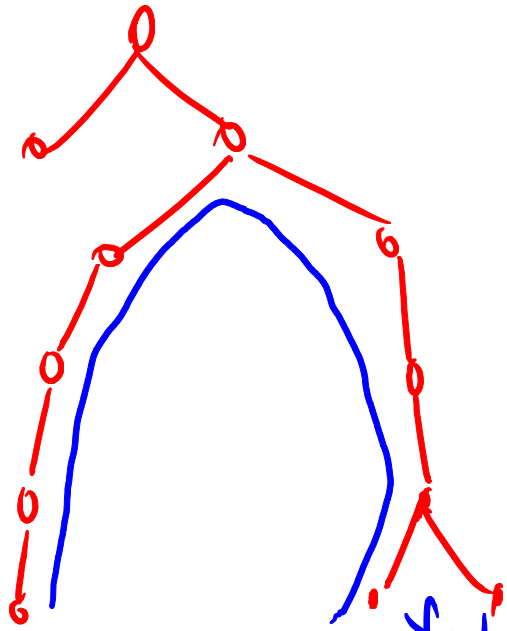
complete: It has as many nodes as possible for its height (i.e., each row is filled in).

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*.)

Example: Finding the Longest Undirected Path in a Tree



longest path length = 6



one of two examples of a longest path

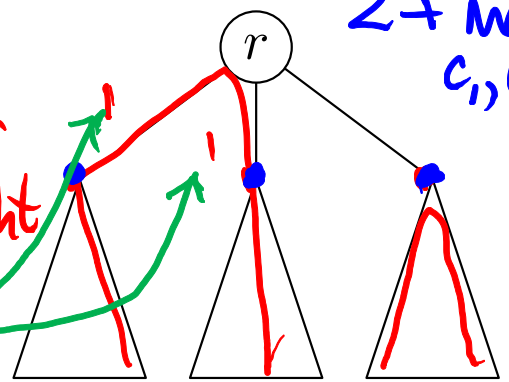
Does such a path always include the root?

Longest Path (LP)

② LP(tree T rooted at node r) =
{ 0 if |T| = 1
else if |T| > 1
max { a) [if ≥ 2 children]
2 + max(height(c_1) + height(c_2))
c_1, c_2
where $c_1, c_2 \in$ {children or r}
$c_1 \neq c_2$
b) [if only 1 child]
height(r)
c) $\max_c$ {LP(c)} where $c \in$ {children of r}.
}

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

① choose the maximum of
- if root r is included
  - compute r's 2 tallest children (2 subtrees with greatest height)
  - add the heights of these two
  - add 2
- if root r is not included
  - compute the longest path in a child's subtree



Back to Queues

height(node t) =
{ 0 if t has no child
else
max_c { height(c) + 1 }
where c ∈ {children of t}
}

check cases:

1) r LP(r) = 0

2) r, c LP(r) = 1

- 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.

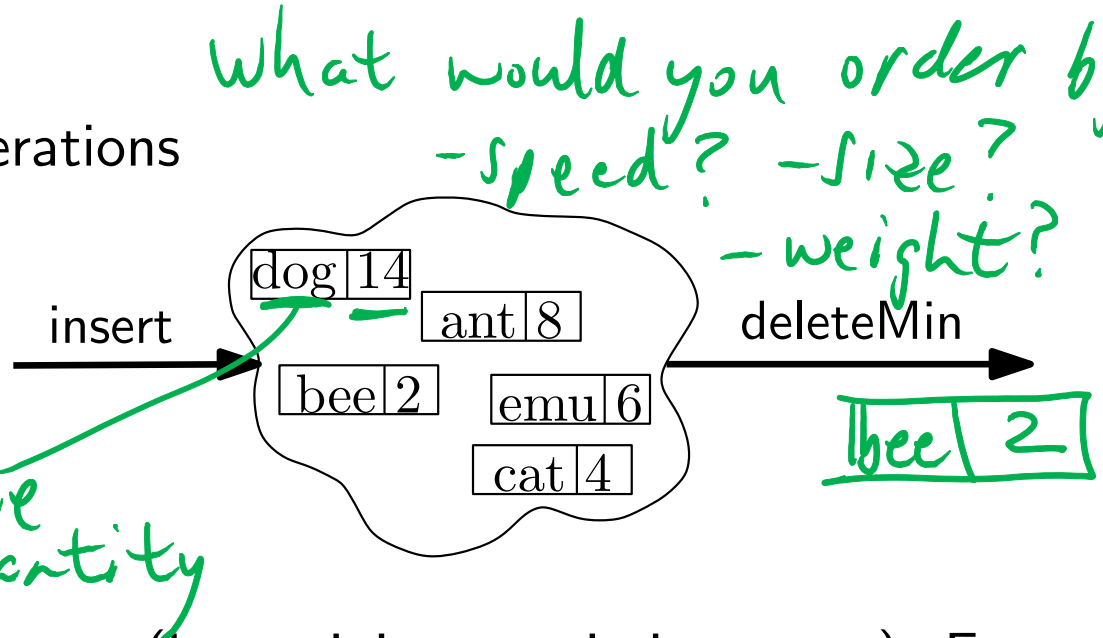
3) r, c, x LP(r) = 2

4) r, c_1, c_2 LP(r) = 2

5) r, c, x_1, x_2, x_3, x_4 LP(r) = 4

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

e.g., array or linked list (LL) {pretend

- Unsorted List
  - insert time: O(1) for either
  - deleteMin time: find it first: O(n) } O(n) + O(1)
    delete it: O(1) } ⇒ O(n)
    ↑ array or LL
- Sorted List
  - insert time: find { array O(lg n) / LL O(n) } & insert { array O(n) / LL O(1) }
  - deleteMin time: find { array O(1) / LL O(1) } delete { array O(1) / LL O(1) } O(1)

best choice: min (O(lg n) + O(n), O(n) + O(1)) ⇒ O(n)

Can we do better? θ for all

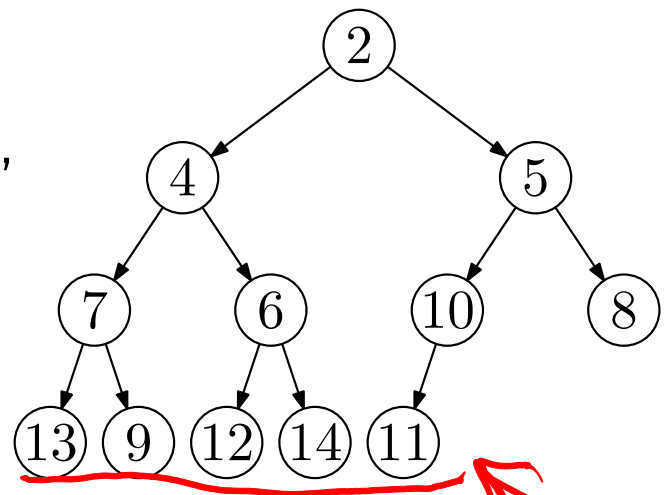
Binary Heap Priority Queue Data Structure

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)
- next open location is always known

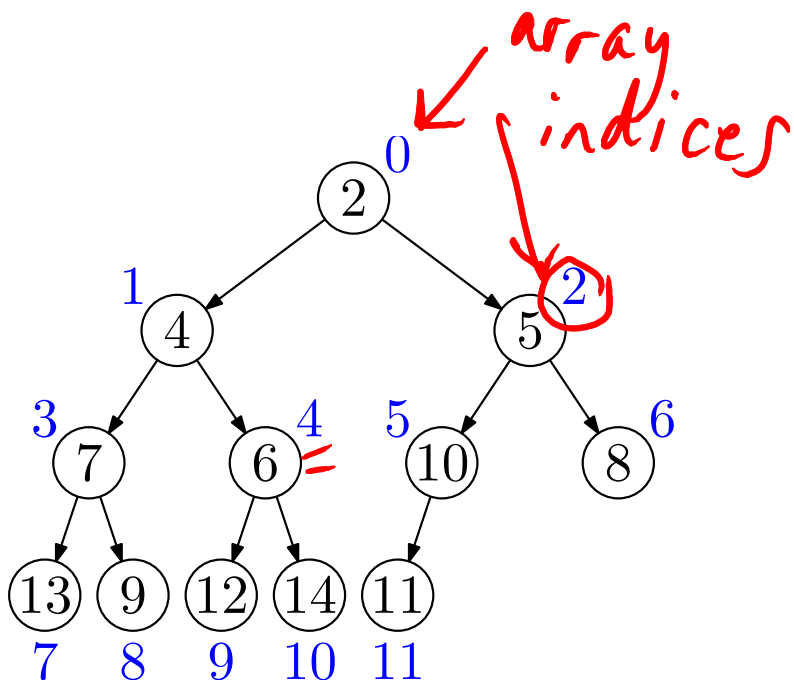


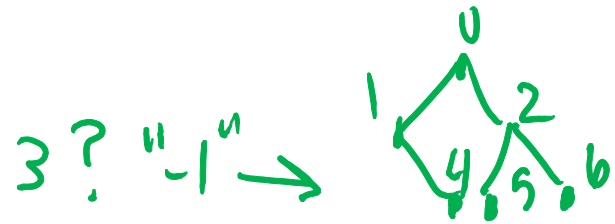
WARNING: This has no similarity to the memory "heap" we talk about when using C++'s `new` operator.

Nifty Storage Trick

Navigation using indices:

- left_child(i) = $2i+1$
- right_child(i) = $2i+2$
- parent(i) = $\lfloor \frac{i-1}{2} \rfloor$
- root = 0
- next free position = n

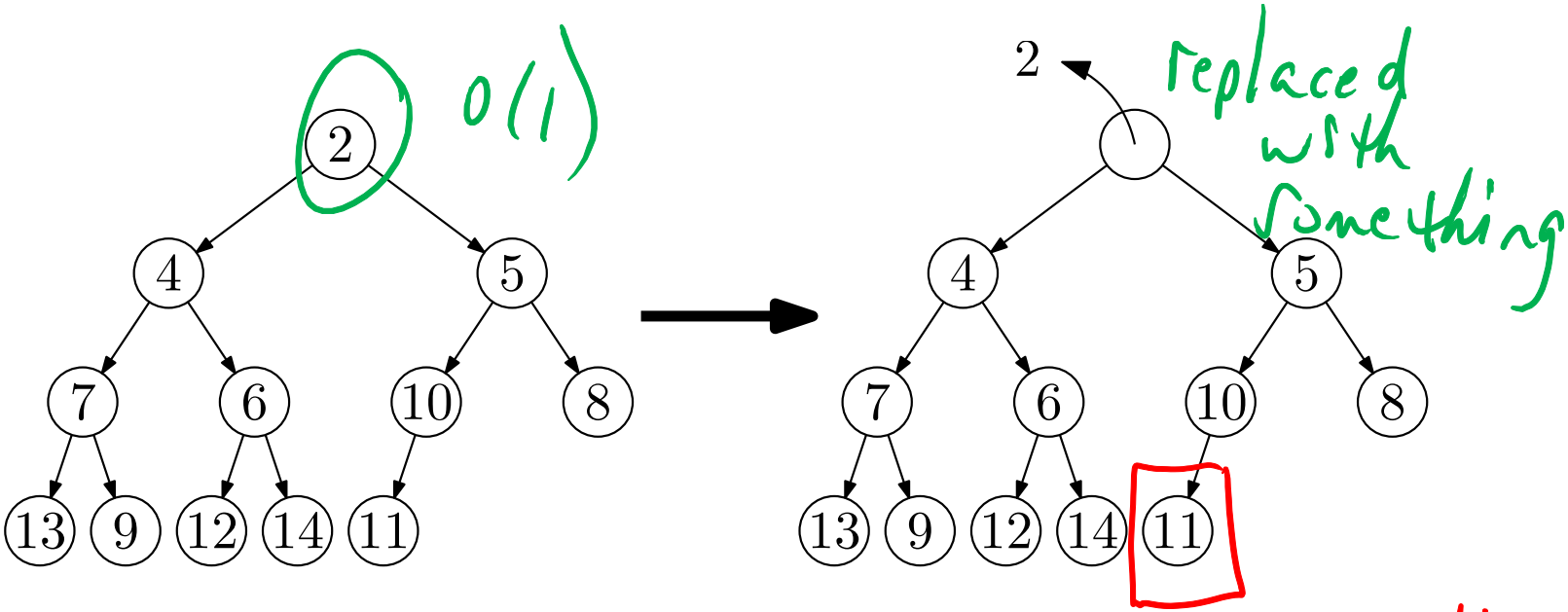




0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	5	7	6	10	8	13	9	12	14	11	

no holes if nearly complete

deleteMin



O(1)

replaced with something

not without a root

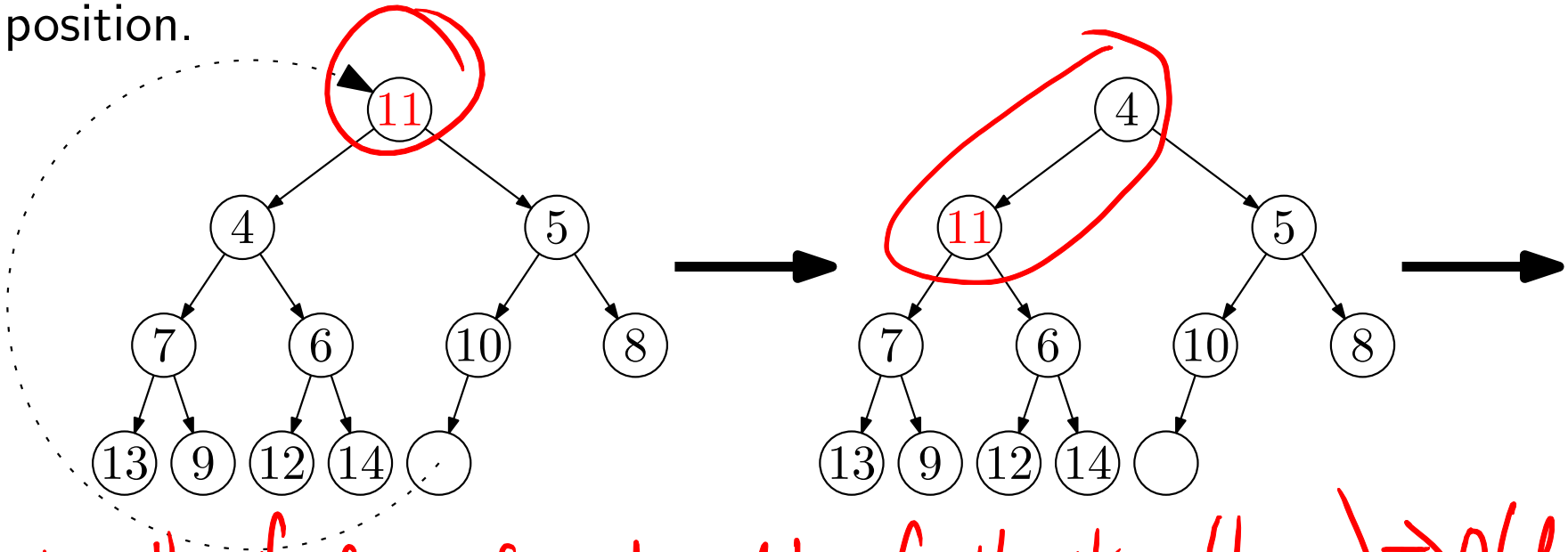
Invariants violated! It's no longer a "nearly complete" binary tree.

- property/assertion that is always true at a given point in the program
  e.g., for loop: - if you're inside, then $i <= n$ (for example)
  - running product is $i!$ etc.

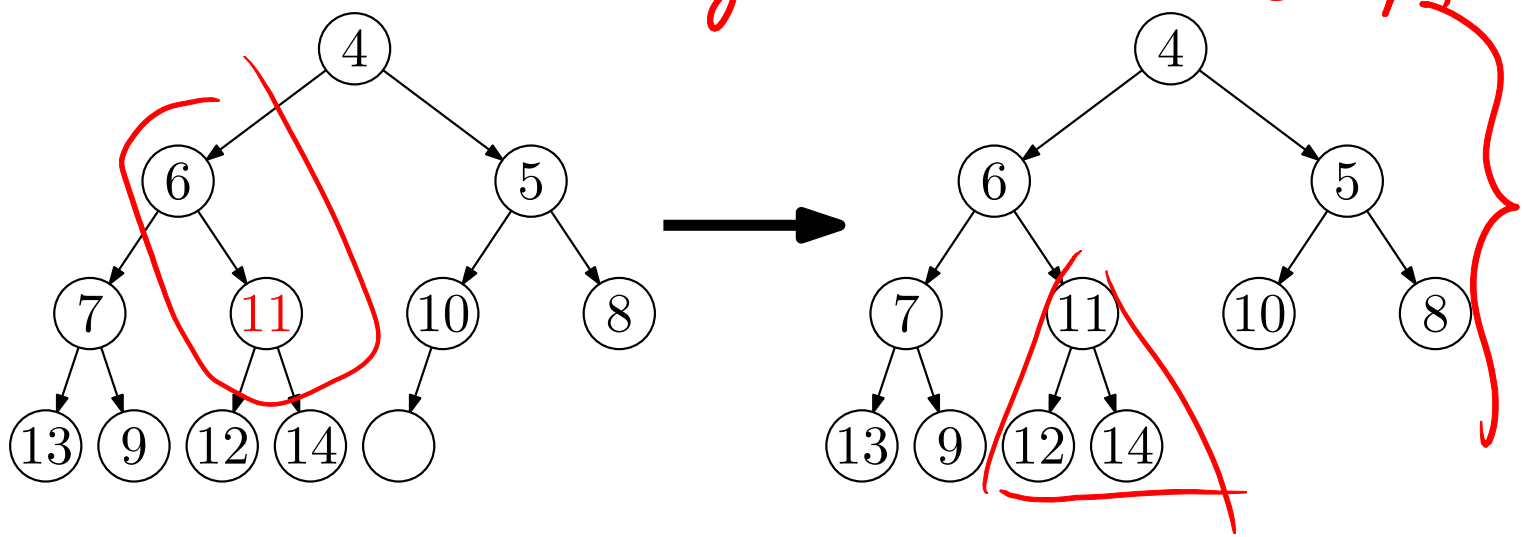
Swap (Heapify) Down

SwapDown is used by Heapify

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



max. # of swaps = height of the tree (heap) $\Rightarrow O(\lg n)$



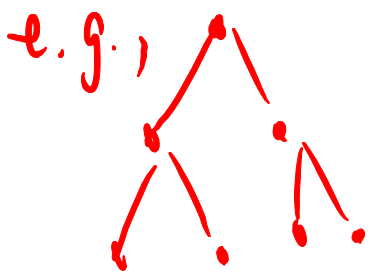
heap

deleteMin Code

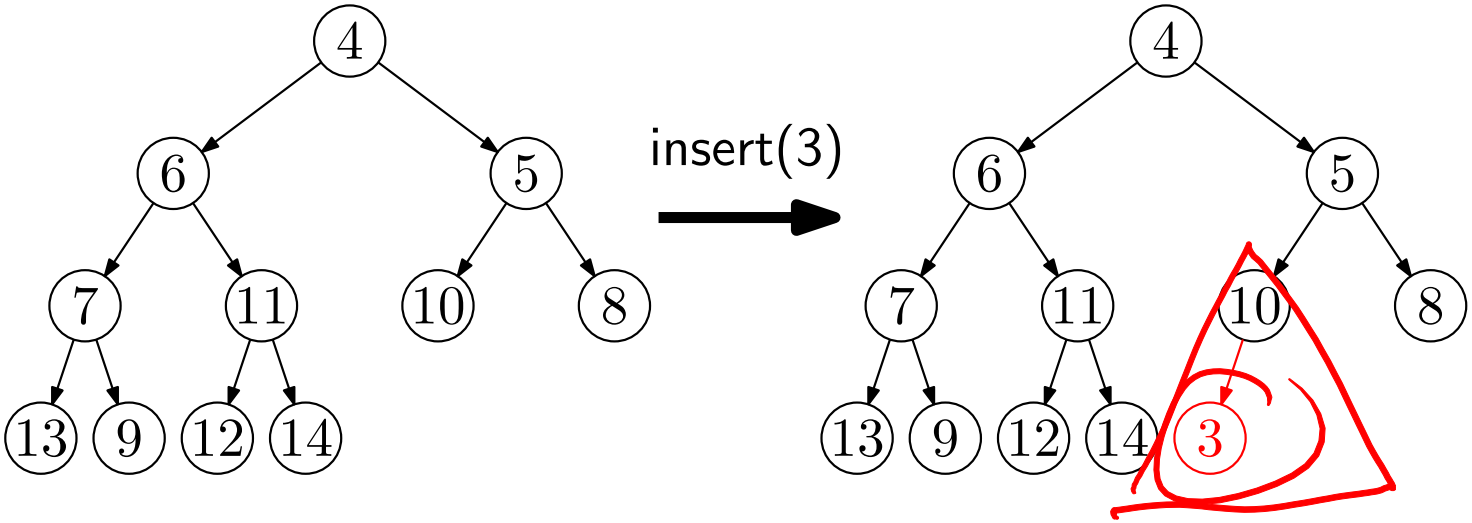
```
int deleteMin() {
  assert(!isEmpty());
  int returnVal = Heap[0];
  Heap[0] = Heap[n-1];
  n--;
  swapDown(0);
  return returnVal;
}
```

Runtime:

```
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);
  }
}
```



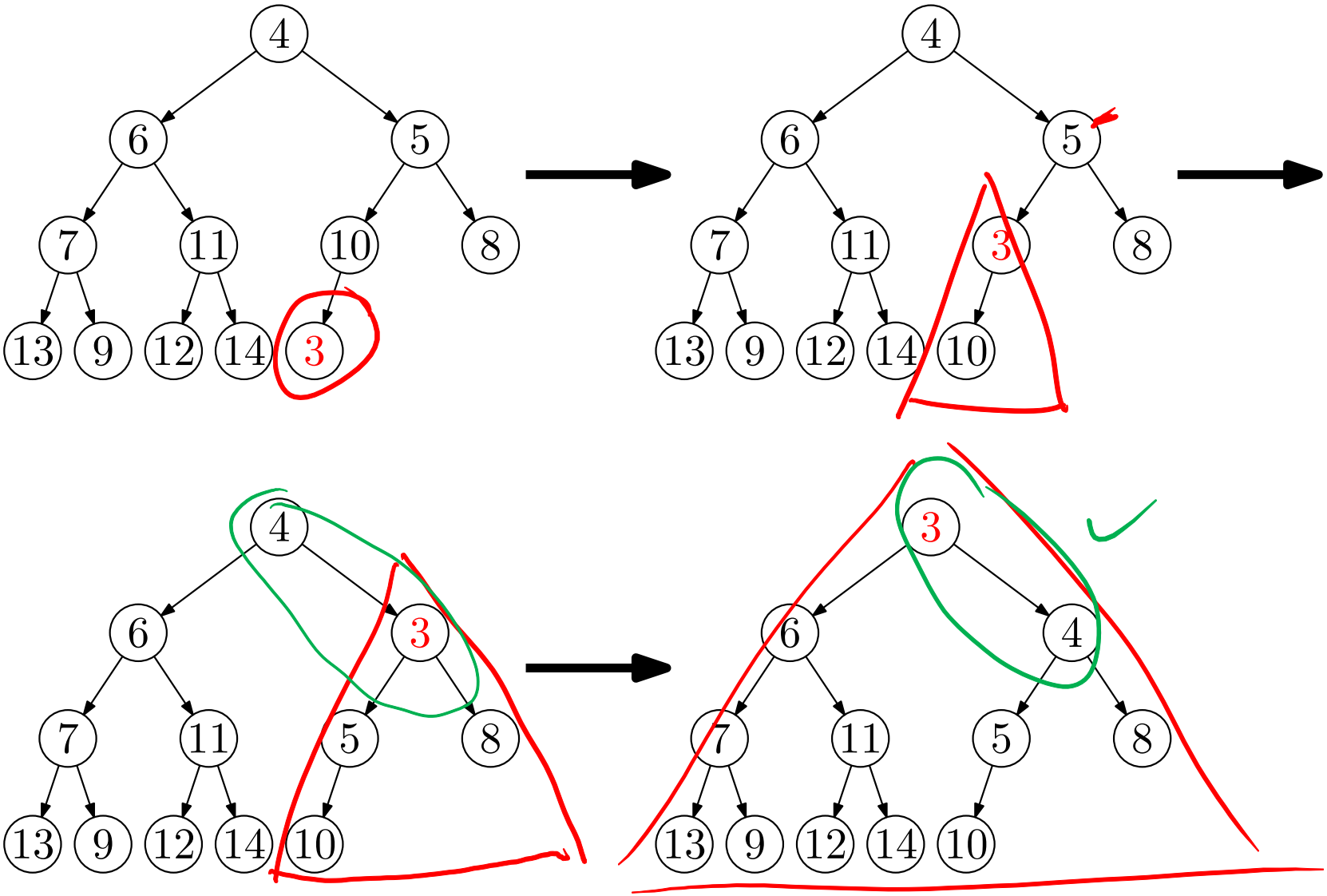
Inserting a New Node



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);
}
```

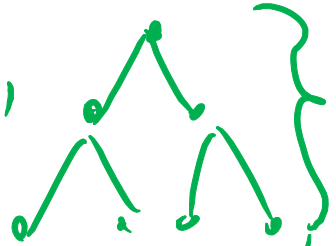
```
void swapUp(int i) {
  if( i == 0 ) return;
  int p = (i - 1)/2;
  if( Heap[i] < Heap[p] ) {
    int tmp = Heap[i];
    Heap[i] = Heap[p];
    Heap[p] = tmp;
    swapUp(p);
  }
}
```

max $\lfloor \lg n \rfloor$ swaps

Runtime: $O(\lg n)$

2 calls
max. 2 swaps

e.g.,



nearly complete

$T(n) = \lfloor \lg n \rfloor$

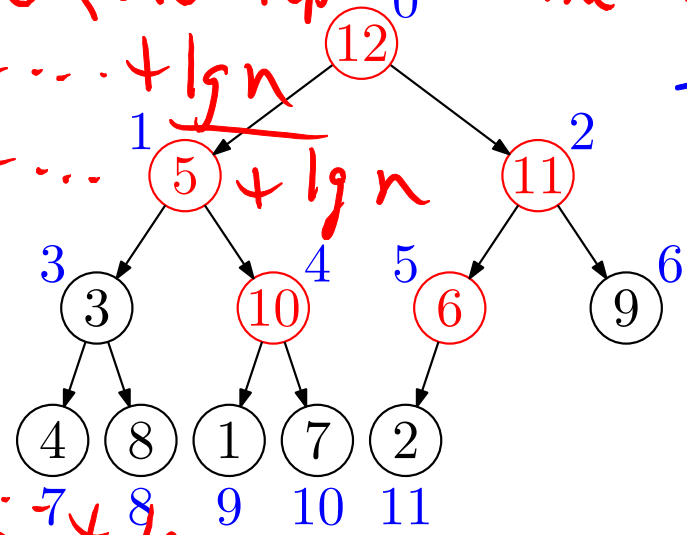
$T(n) \in O(\lg n)$

$h = 2$ { complete $\lfloor \lg 7 \rfloor = 2$
nearly " $\lfloor \lg 4 \rfloor = 2$

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

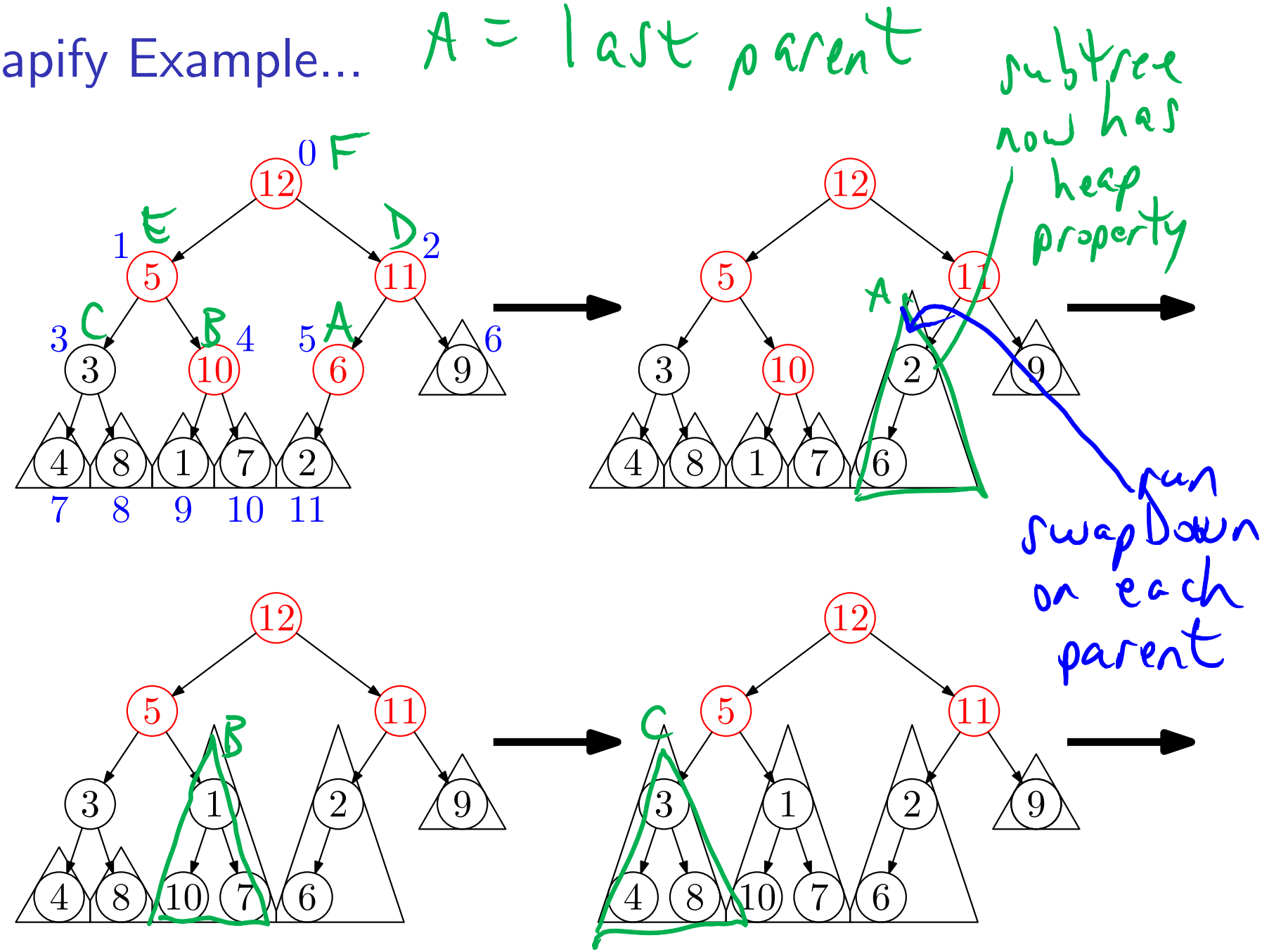


Invariant violated!

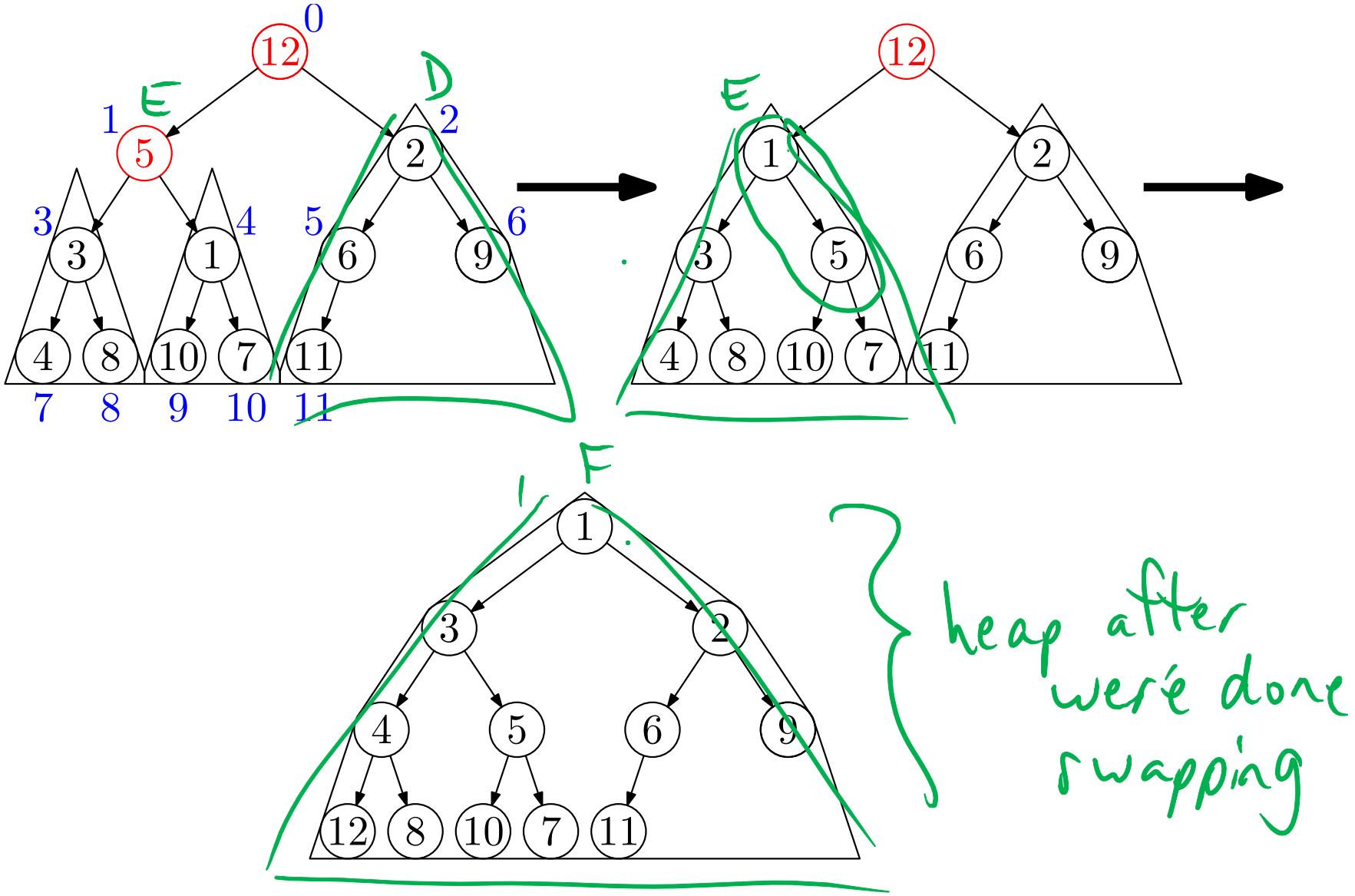
2. Fix the heap-order property, starting from the bottom, and going up. Use `swapDown`.

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

Heapify Example...



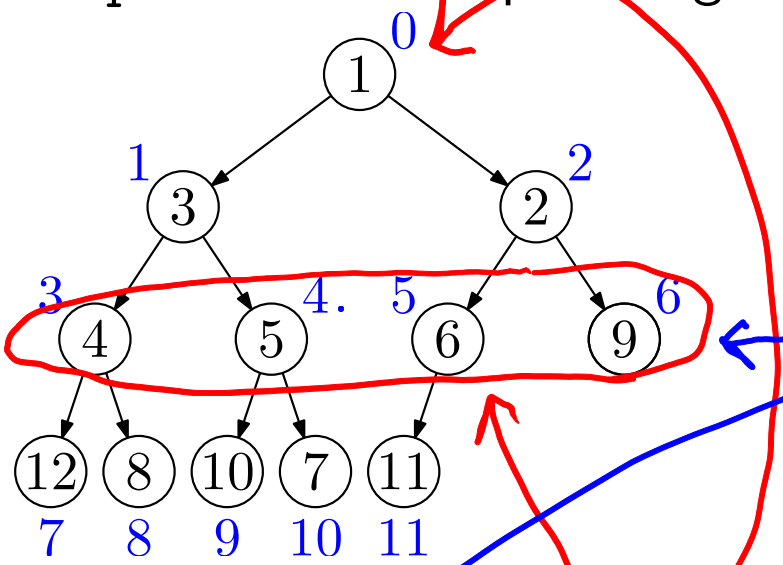
Heapify Example



Heapify Runtime

Proof #1 *

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



$h \le 1$

$h = \lfloor \lg n \rfloor$
$h = O(\lg n)$
$\therefore 2^h = O(n)$

Let H be the height of the heap. here $H = 3$

③ means:
- on height = 1 (parent level) we will call swapDown $\le 2^2 = 4$ times
- on height = 2, " " $\le 2^1$ times
- on height = 3, " " $\le 2^0$ times

`swapDown` is called once on heap of height $H = 3$
≤ 2 times on heap of height $H-1 = 2$
≤ 4 times on heap of height $H-2 = 1$
⋮
$\le 2^{H-1}$ times on heap of height 1 stops at 1

② $1(2^{H-1}) + 2(2^{H-2}) + 3(2^{H-3}) + \dots$
$1(2^2) + 2(2^1) + 3(2^0)$

⑦ $2(2^H) = 2(2^{\lfloor \lg n \rfloor}) \le 2(2^{\lg n}) = 2n$

⑧

Total # steps $\le \sum_{h=1}^{H} h 2^{H-h}$ ① $= 2^H \sum_{h=1}^{H} h/2^h$ ⑥ $\le 2^{H+1} = O(n)$ *

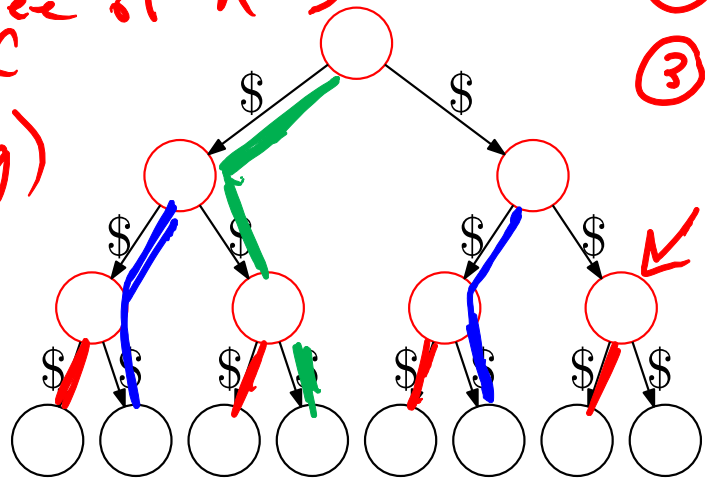
④ $\frac{h \cdot 2^H}{2^h} = 2^H \frac{h}{2^h}$

⑤ $\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots \le 2$

Heapify Runtime: Charging Scheme

Proof #2

① - worst case tree of h=3
- without loss of generality (wlog) always choose the leftmost uncolored path to swap in the worst case



② #edges = n-1

③ only the rightmost path (root to rightmost child is uncolored)

$\Rightarrow$ (n-1) - H swaps

$\leq$ (n-1) - lg n

$\leq$ n

$\Rightarrow$ O(n) swaps!

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.

Thinking about Binary Heaps

Observations

- Finding a child/parent index is a multiply/divide by two operation.
- Both deleteMin and the subsequent insert might access far-apart array locations.
- deleteMin accesses all children of visited nodes.
- insert accesses only the parent of visited nodes.
- insert is at least as common as deleteMin.

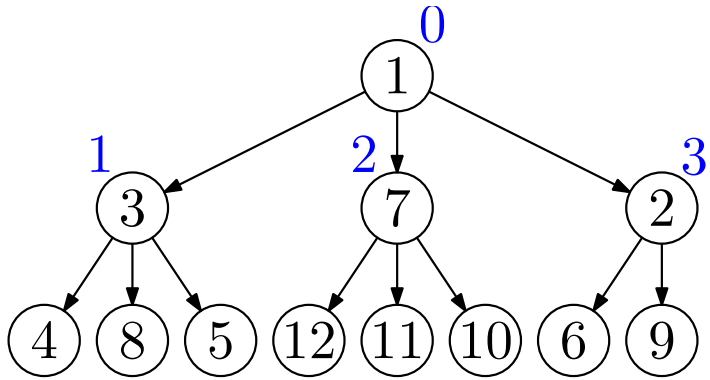
Realities

- 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.

Solution: d-Heaps

arbitrary arity (d)

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



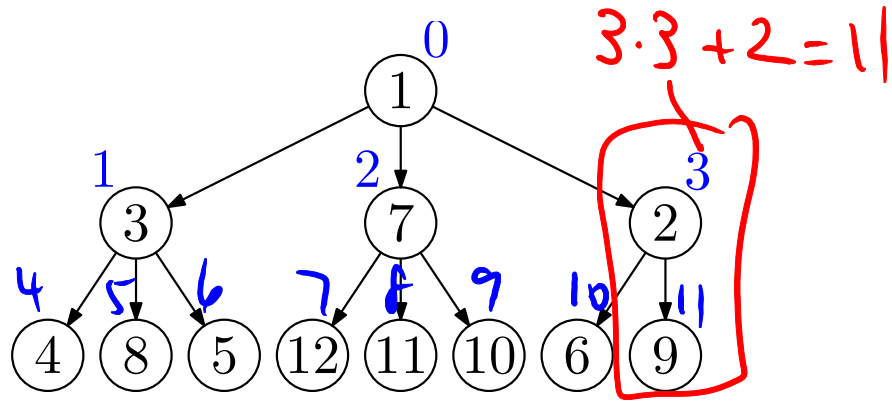
1	3	7	2	4	8	5	12	11	10	6	9
0	1	2	3	4	5	6	7	8	9	10	11

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

- jth-child(i) = $d \cdot i + j$
- parent(i) = $\left\lfloor \frac{i-1}{d} \right\rfloor$
- root = 0
- next free position = n



1	3	7	2	4	8	5	12	11	10	6	9
0	1	2	3	4	5	6	7	8	9	10	11

$H = \log_d(n)$

swapUp $O(\log_d n)$

swapDown $O(d \log_d n)$

↑ up to d children } better to measure # of comparisons in this case since d can be large; but $O(\log_d n)$ swaps