Unit #2: Priority Queues

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

Unit Outline

- Rooted Trees, Briefly
- Priority Queue ADT
- Heaps
  - Implementing Priority Queue ADT
  - Focus on Create: Heapify
  - Brief introduction to *d*-Heaps

Learning Goals

- 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

- Family Trees
- Organization Charts
- Classification trees (a.k.a. keys)
  - What kind of flower is this?
  - Is this mushroom poisonous?
- File directory structure
  - folders, subfolders in Windows
  - directories, subdirectories in UNIX
- Non-recursive call graphs

Fib(n)

Fib(n-1) F(n-2)

Fib(n-2) Fib(n-3) ⋮

. . .

Tree Terminology

nodes = vertices
edges = branches = arcs

root: A

leaf: D, E, F, I, J, ..., N

child: of B: D, E, F

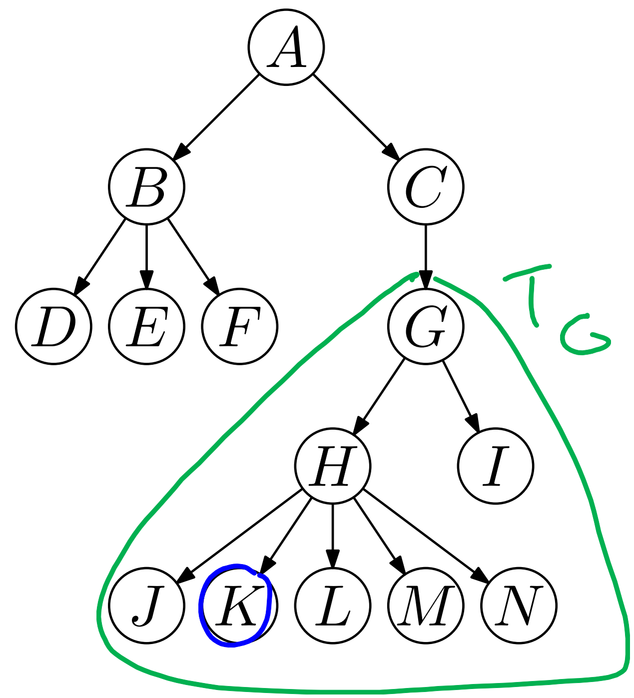
parent: of E: B

sibling: of D: E, F

ancestor: of K: A, C, G, H
(proper)

descendent: of G: H, I, J, ..., N

subtree: T_G
of G:



Tree Terminology Reference

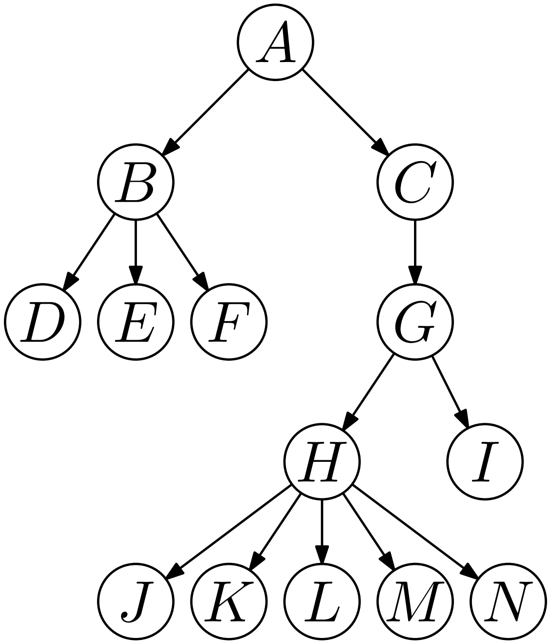
root: the single node with no parent

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

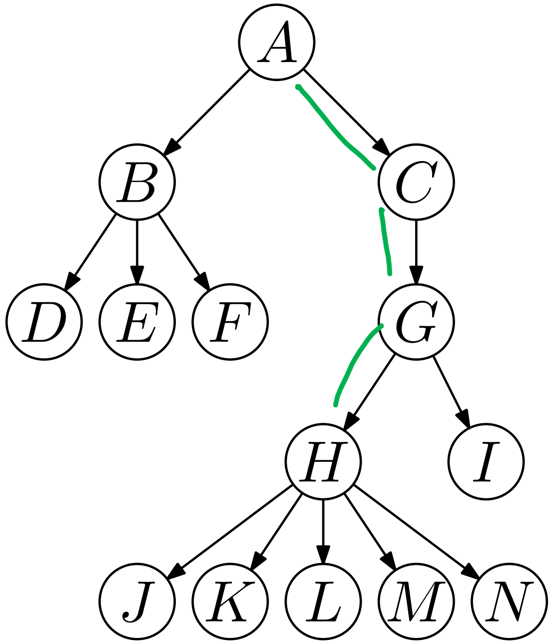
descendent: my child or my child's descendent

subtree: a node and its descendents

More Tree Terminology

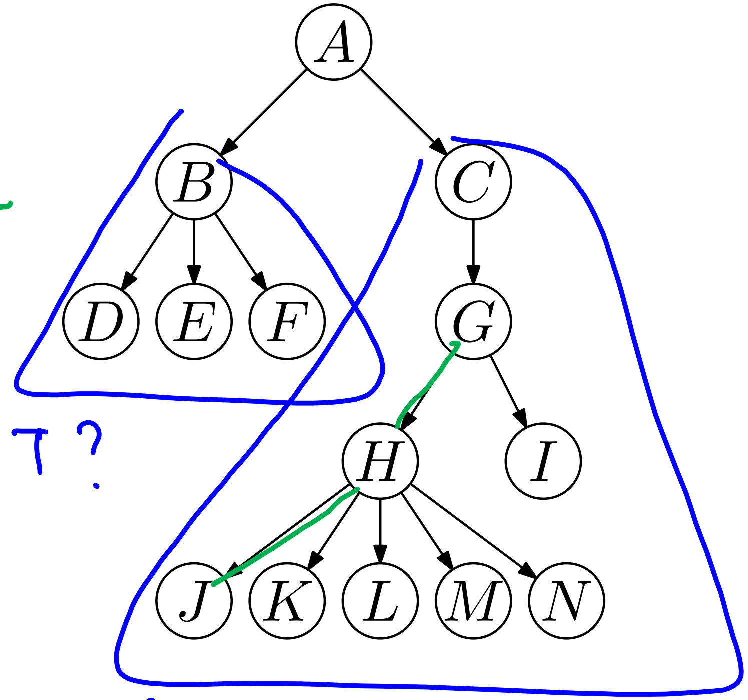
depth: Number of edges on path from root to node

depth of H?   3



More Tree Terminology

height: Number of edges on longest path from node to descendent or, for whole tree, from root to leaf



height of tree? = height of the root = 4

height of G? 2

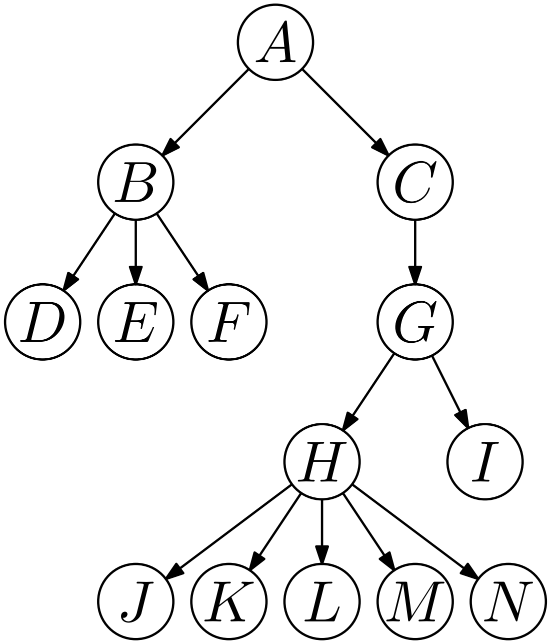
Q. How to calculate height of the T?

A.
$$\text{height}(T_r) = \begin{cases} 1 + \max\limits_{\text{child } c \text{ of } r} \text{height}(T_c) & \text{otherwise} \\ 0 & \text{if } |T_r| = 1 \end{cases}$$

More Tree Terminology

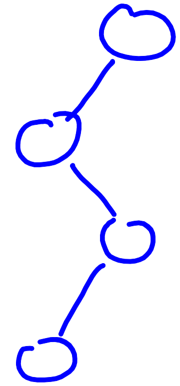
(downward) degree: Number of children of a node

degree of B? 3



One More Tree Terminology Slide

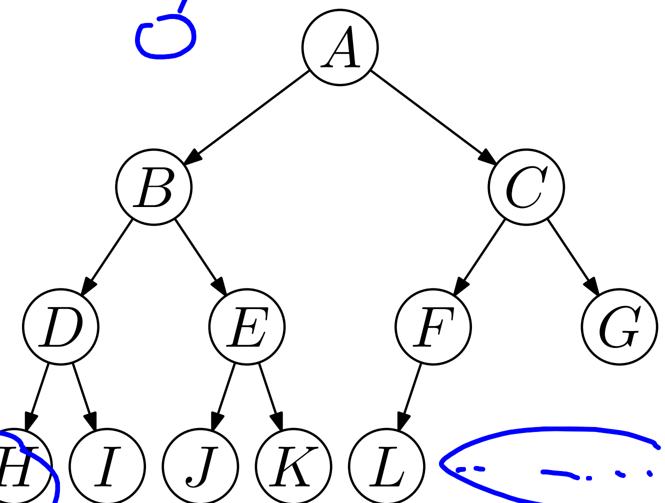
children ordered : left | right



binary tree

binary: each node has degree at most 2

d-**ary:** degree at most *d*



Q # nodes in a binary tree of height h?

$$h+1 \le n \le 2^{h+1}-1$$

complete: as many nodes as possible for its height (each row filled in)

nearly complete: each row except the last one is filled in, all nodes in the last row are as far left as possible

Q. What is the height of a nearly complete tree with n nodes?

$$2^h \le n \le 2^{h+1}-1 < 2^{h+1} \implies h \le \lg n < h+1$$

$\Updownarrow$ ← $\Theta(\log n)$

A: $h = \lfloor \lg n \rfloor$ ← integer part

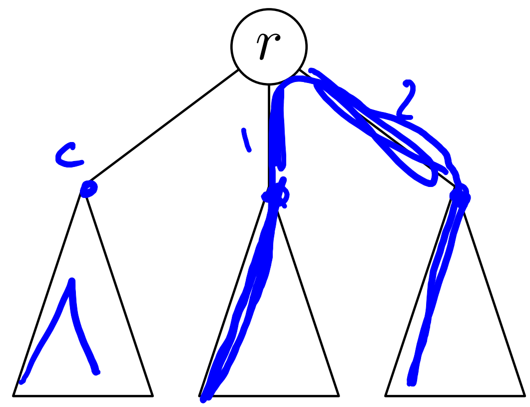
Longest Path

Find the longest *undirected* path in a tree

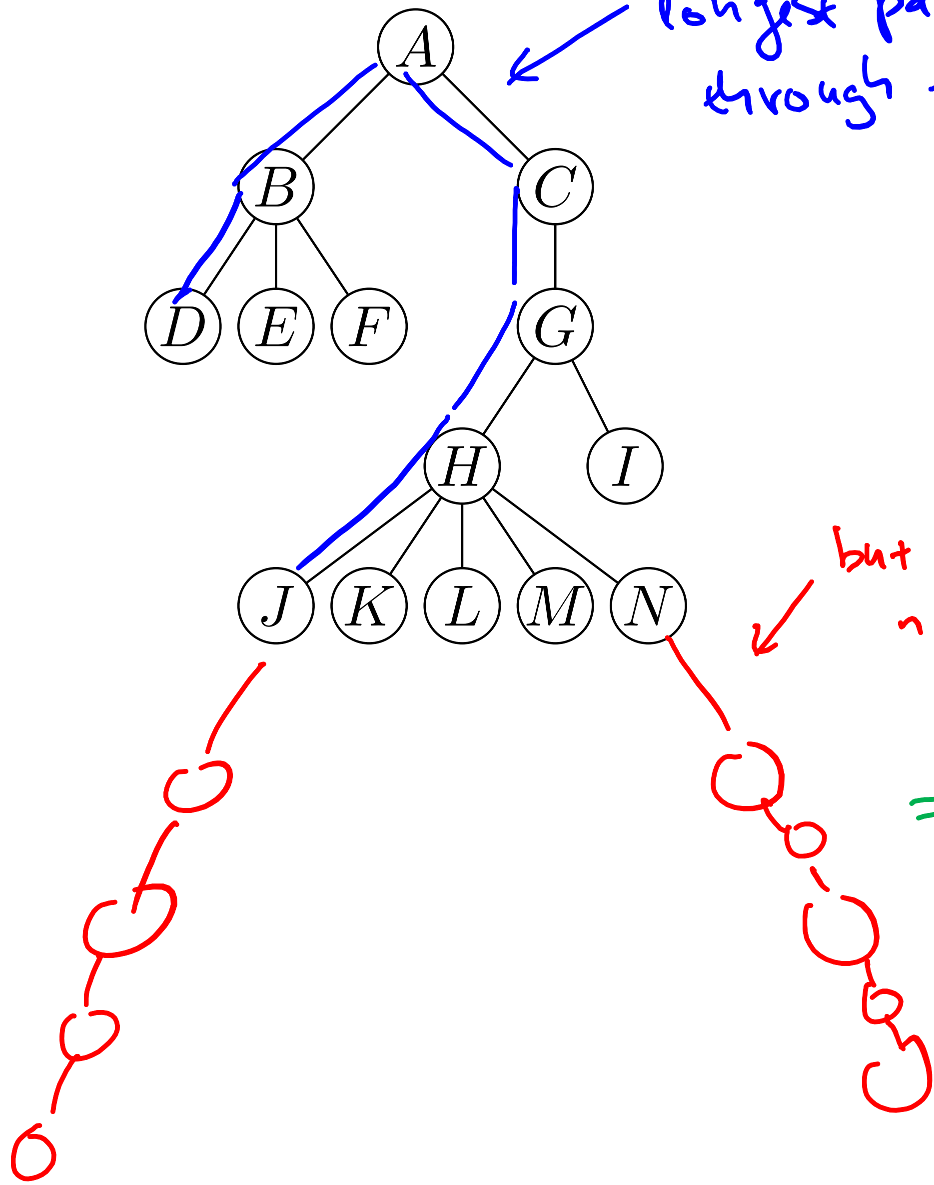
See the example on the next slide first

longestPath(r) {

$$
\begin{cases}
\max \begin{cases} \max\limits_{c \text{ child of } r} \text{longestPath}(c) \\ \max\limits_{c \neq d \text{ children of } r} \text{height}(c) + \text{height}(d) + 2 \end{cases} \\
0 \quad \text{if } r \text{ has no children}
\end{cases}
$$



Longest Path Example



longest path goes through the root

but if we add red nodes it doesn't

$\Rightarrow$ we need to consider both cases

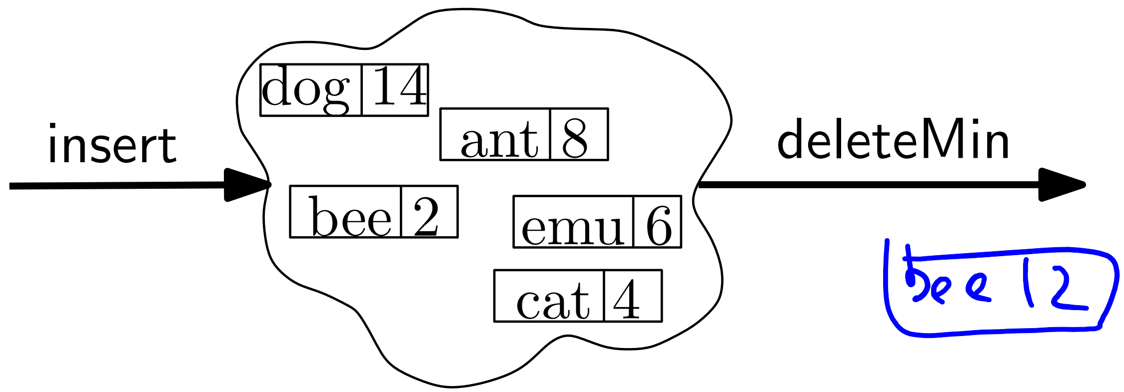
Back to Queues

- Applications
  - ordering CPU jobs
  - simulating events
  - picking the next search site
- But we don't want FIFO ...
  - *short* jobs should go first
  - *earliest* (simulated time) events should go first
  - *most promising* sites should be searched first

priority associated with objects

Priority Queue ADT

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



- Priority Queue property: For two elements in the queue, x and y, if x has a lower priority value than y, x will be deleted before y.

Applications of the Priority Q

- 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*: an algorithm that makes the "locally best choice" at each step

Priority Q Data Structures

- Unsorted list
  - insert time: $\Theta(1)$
  - deleteMin time: $\Theta(n)$

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

    find position / insert at position

    array: $\Theta(\log n) + \Theta(n)$

    linked list: $\Theta(n) + \Theta(1)$
  - deleteMin time: $\Theta(1)$

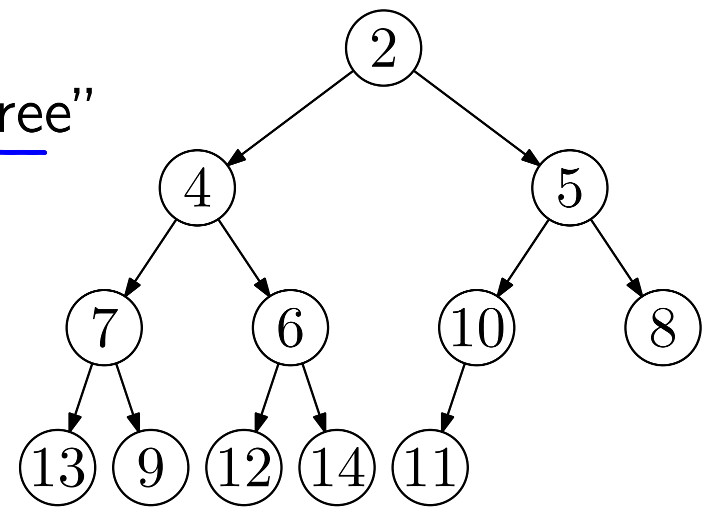
Binary Heap Priority Q Data Structure

Heap-order property: parent's key $\leq$ children's keys.

- minimum is always at the top

Structure property: "nearly complete tree"

- depth is always O(log n)
- next open location always known



```
struct Node {
    int data;
    Node* left;
    Node* rigth;
    Node* parent; };
```

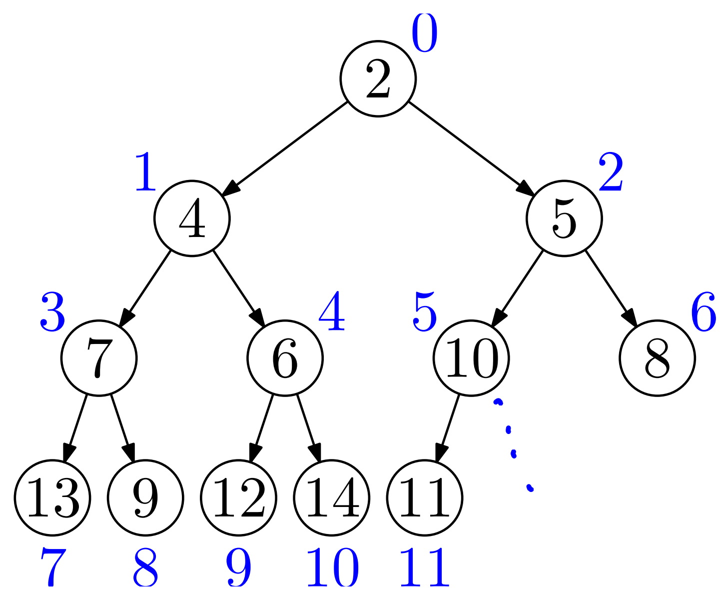
WARNING: This has NO SIMILARITY to the "heap" you hear about when people say "things you create with `new` go on the heap".

Nifty Storage Trick (use array instead)

size = n

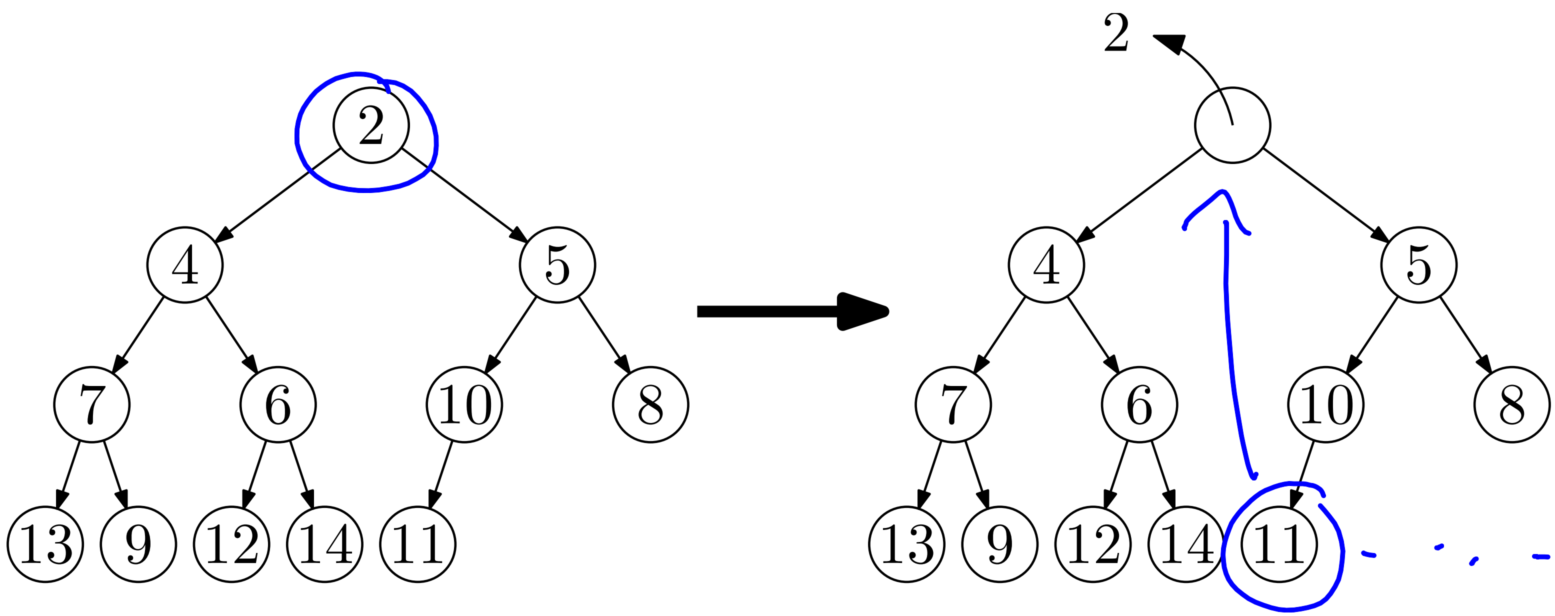
Navigation using indices:

- left_child(i) = $2*i+1$
- right_child(i) = $2*i+2$
- parent(i) = $\lfloor \frac{i-1}{2} \rfloor = \lceil \frac{i}{2} \rceil - 1$
- 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	

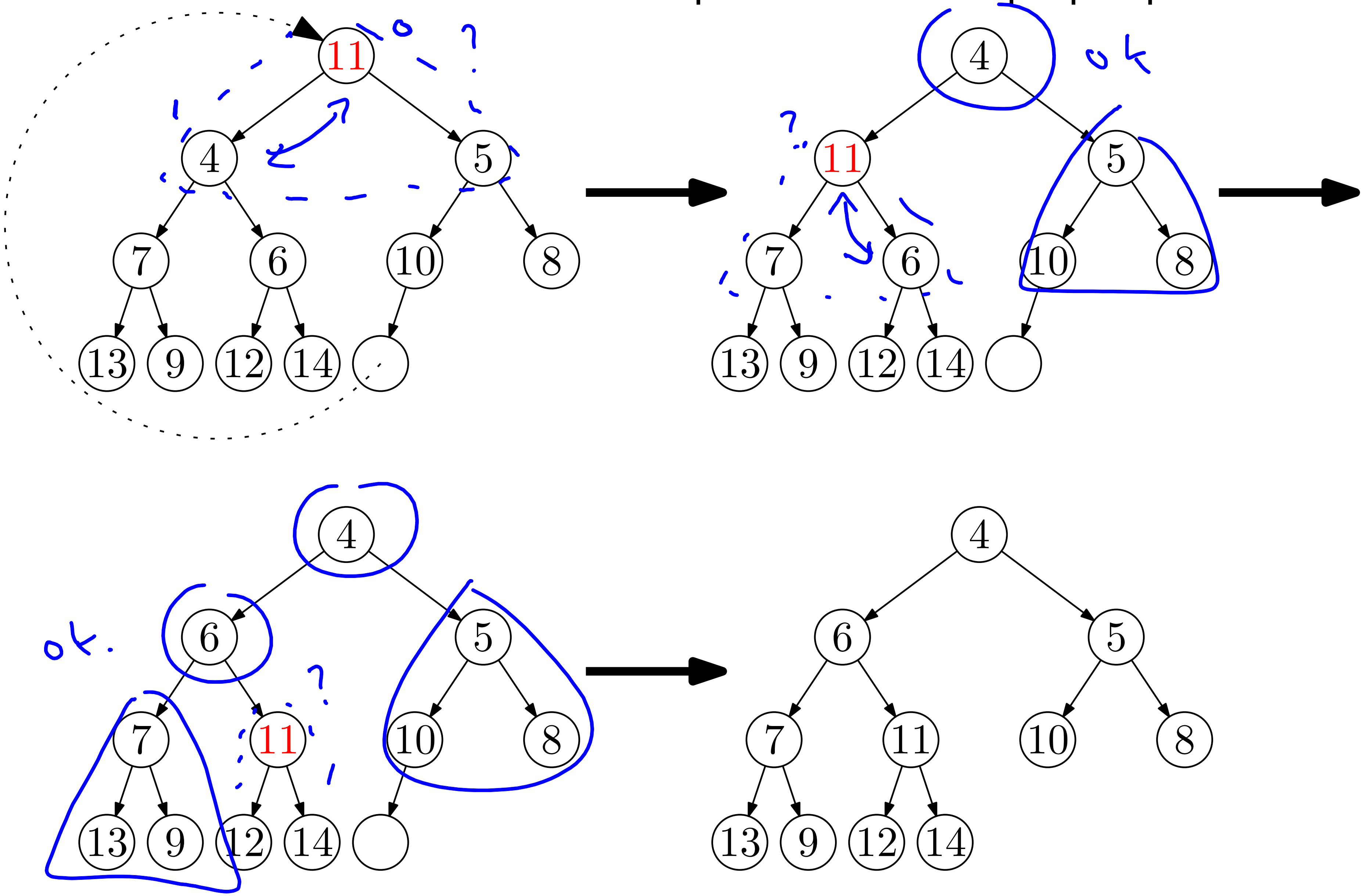
DeleteMin



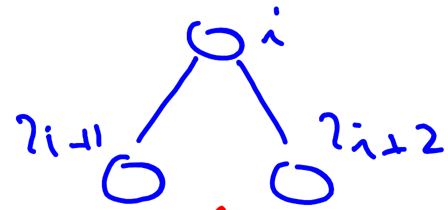
Invariants violated! No longer "nearly complete"

Swap (Heapify) Down

Move last element to root 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;
}
```

```
void swapDown(int i) {
  int s = i;
  int left = i * 2 + 1;
  int right = left + 1;   2i+2
  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);   ← s > 2*i doubling until n
  }
}
```

step

s will be hold index of the smallest key of these

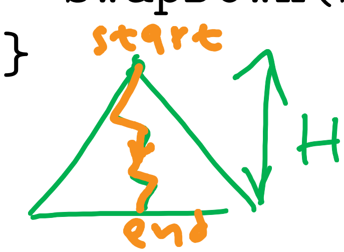
false at a leaf

worst-case Runtime:

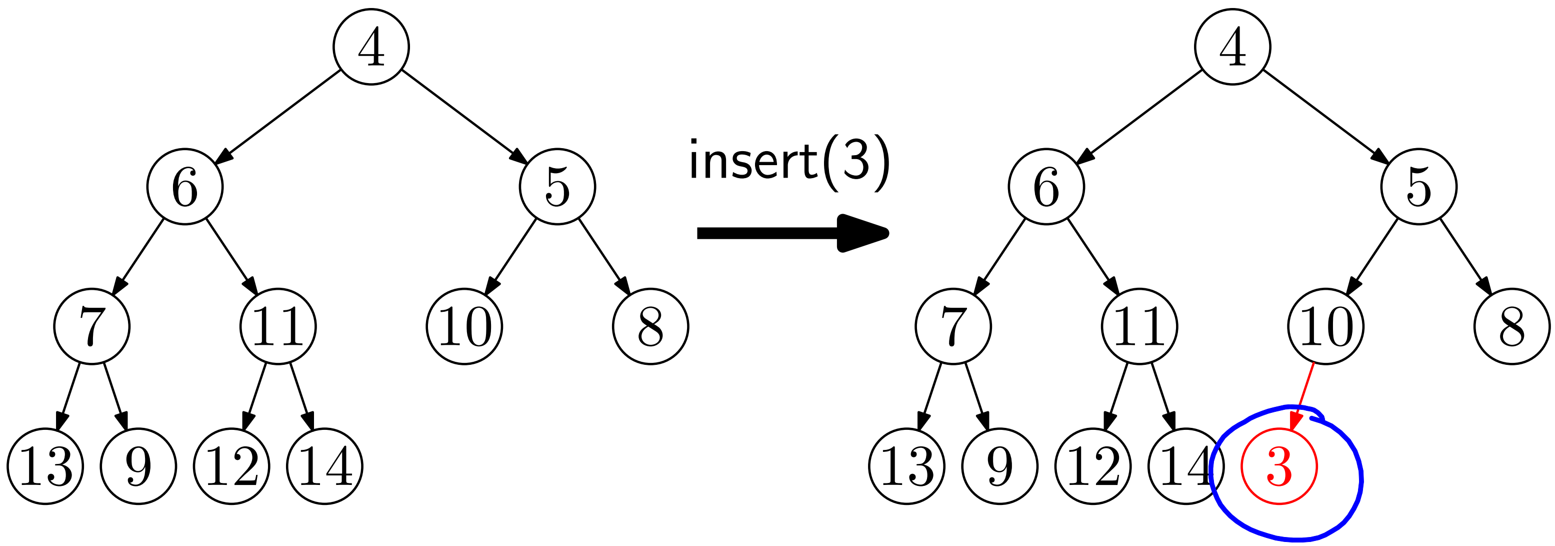
$\Theta(H) = \Theta(\log n)$

height

general time: $O(\log n)$



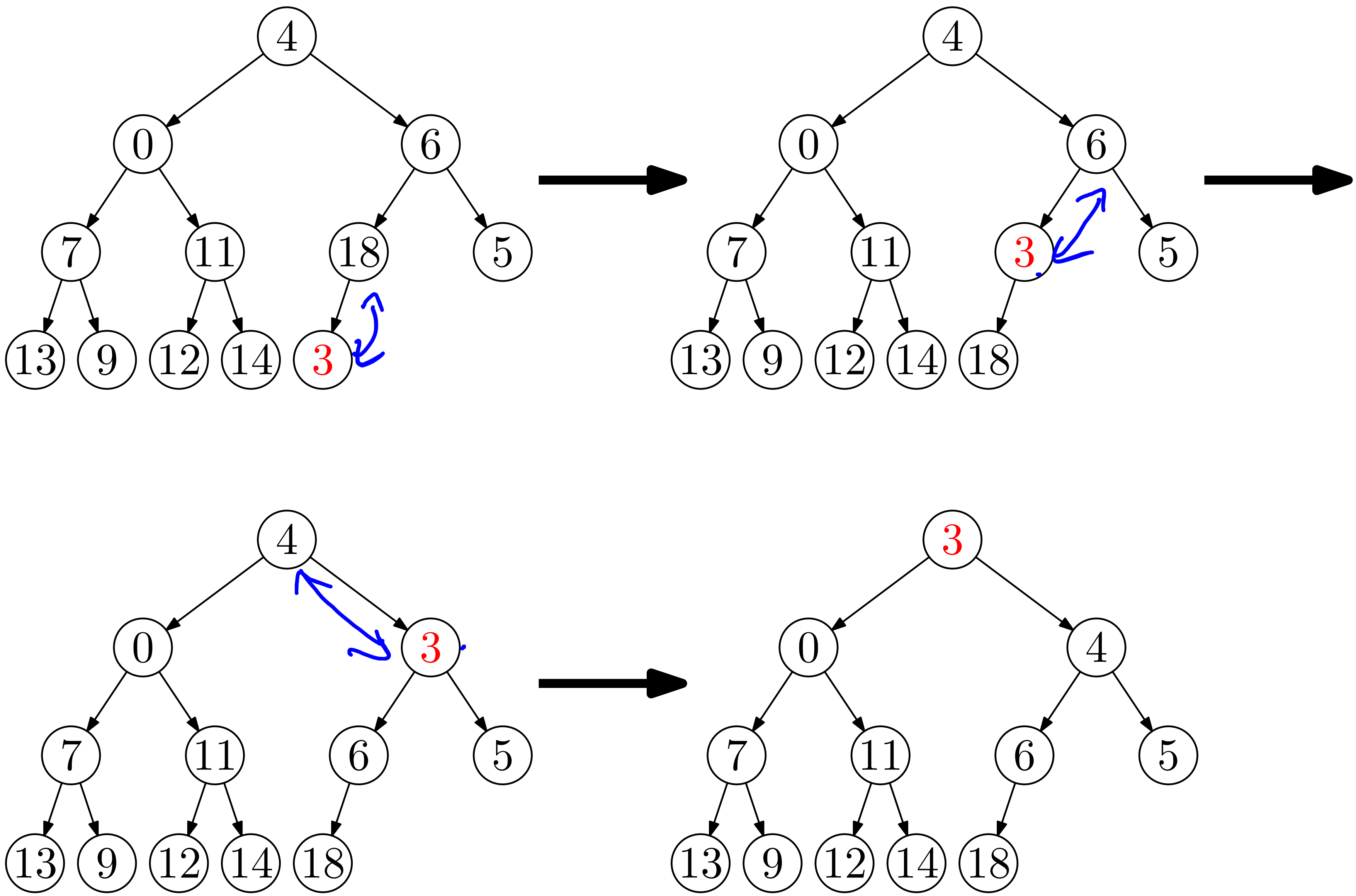
Insert



Invariant violated! Child has smaller key than parent.

Swap (Heapify) Up

Put 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);
}
```

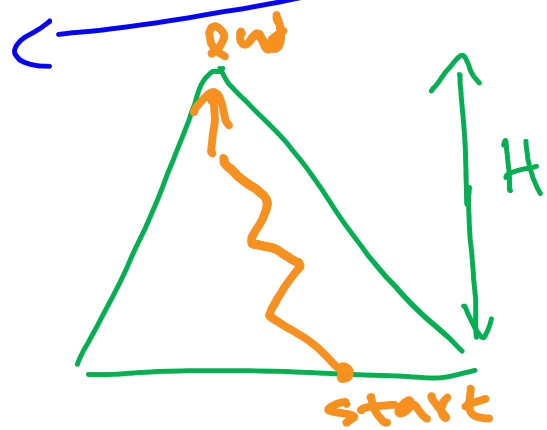
```
void swapUp(int i) {
  if( i == 0 ) return;
  int p = (i - 1)/2;
  if( Heap[i] < Heap[p] ) {
    int tmp = Heap[i];
    Heap[i] = Heap[p];
    Heap[p] = tmp;
    swapUp(p);
  }
}
```

$p < i/2$

halve starting from n until 0

Runtime:

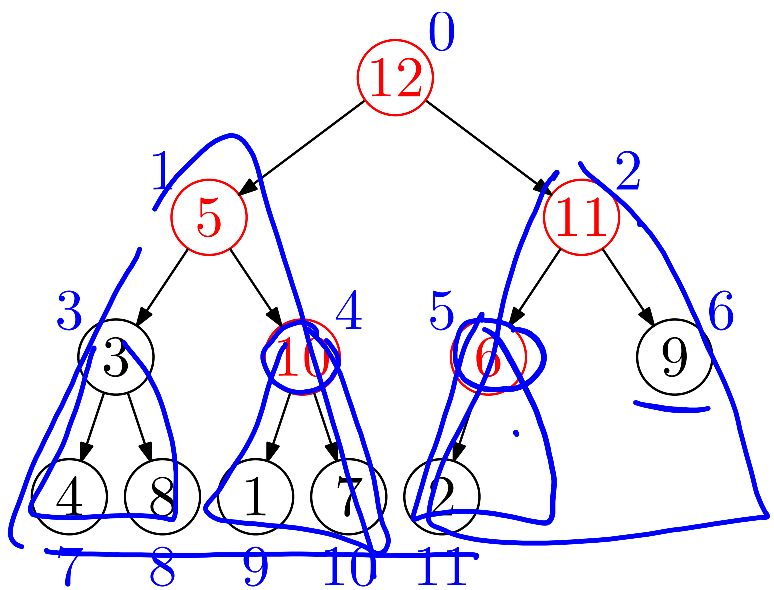
$\Theta(H) = \Theta(\log n)$



Heapify: Create a Heap from a non-Heap Array

1. Start with the input array.

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



Invariant violated!

$\frac{n}{2}$ times

$\times\ O(\log n)$

2. Fix the heap-order property bottom up. Use swapDown.

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

$O(n \log n)$

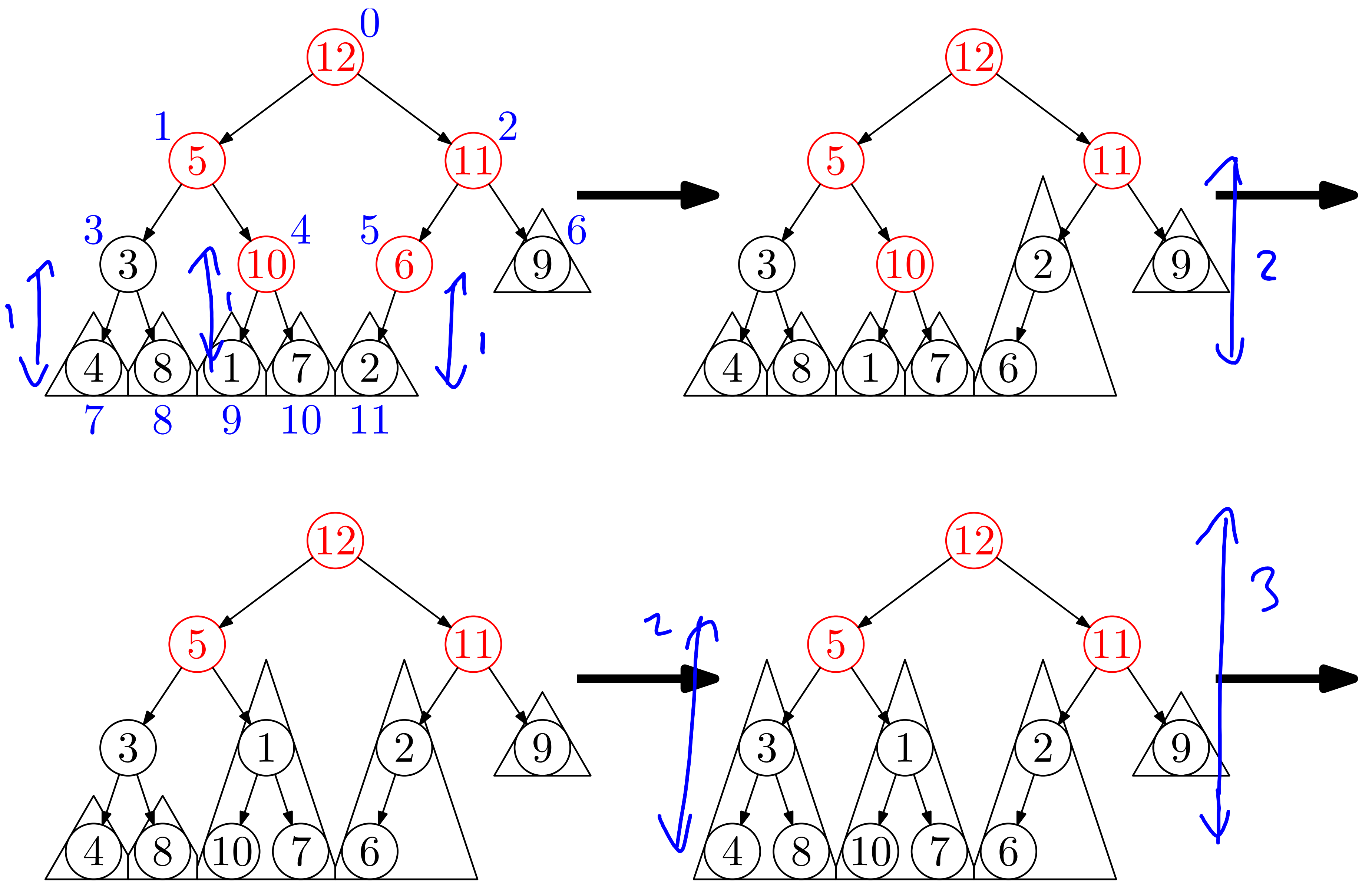
Floyd

parent of last element

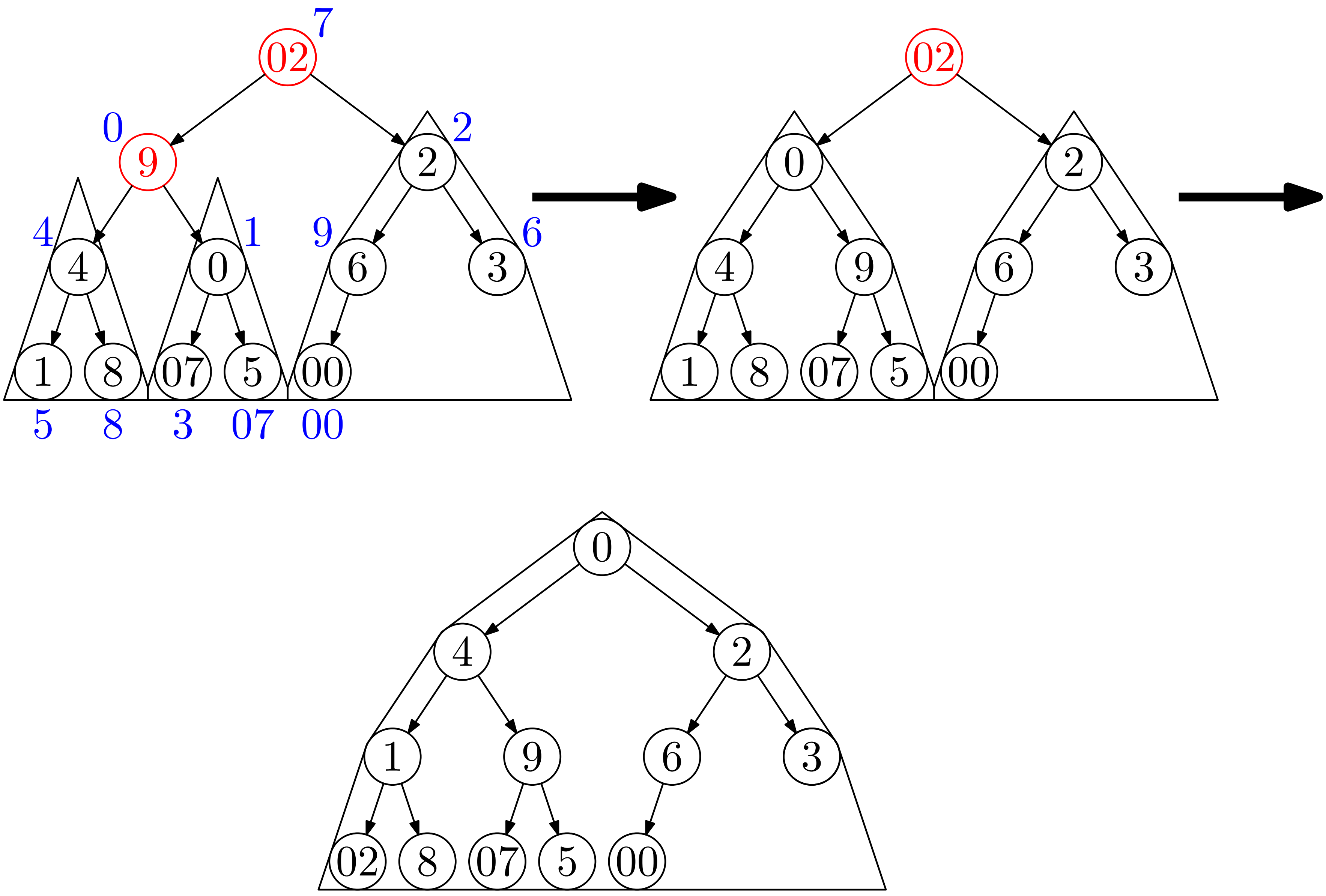
upper bound on time

Heapify Example...

△ .. subtree is already a heap

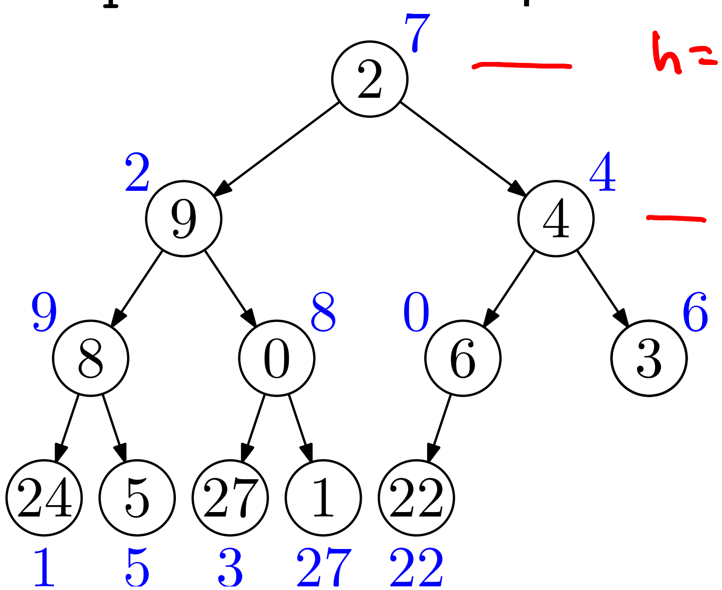


Heapify Example



Heapify Runtime

`swapDown` on a heap of height h takes at most $\underline{\quad h \quad}$ steps.



$h = H$

$h = H-1$

Let H be the height of the heap.

$h = H-2$

$\vdots$

`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	. . . . .	$\vdots$ h
	$\leq 2^{H-1}$ times	on heap of height	1

\# of times swapDown is executed on node with height h

better upper bound on time

$$\text{Total \# steps} \leq \sum_{h=1}^{H} \underbrace{h}_{\text{runtime of swapDown}} 2^{H-h} = 2^H \underbrace{\sum_{h=1}^{H} h/2^h}_{<2} \leq 2^{H+1} = O(n)$$

$$\sum_{h=1}^{H} h/2^h < \underbrace{\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots}_{S}$$

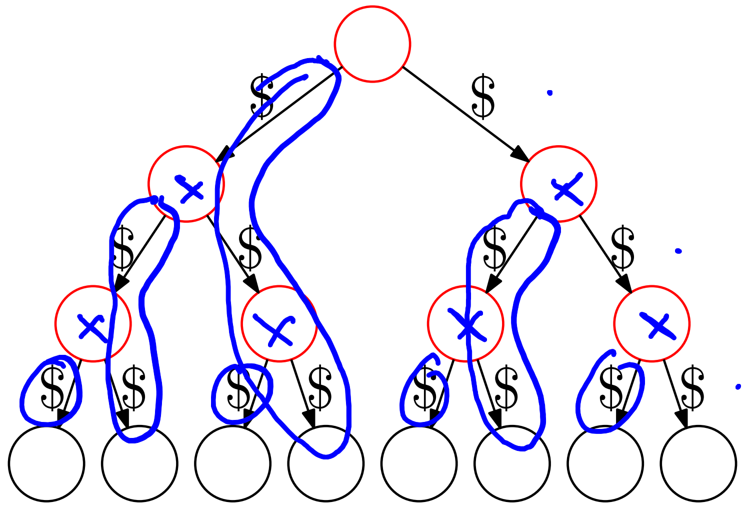
$$2S = 1 + \frac{2}{2} + \frac{3}{4} + \frac{4}{8} + \cdots$$

$$- \quad S = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots$$

$$S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$$

$$= 2$$

Heapify Runtime: Charging Scheme



\$ = n-1

#steps:

$n-1-\log n$

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
- deleteMin and insert access far-apart array locations
- deleteMin accesses all children of visited nodes
- insert accesses only 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 ruin cache performance
- with huge data sets, disk I/O dominates

Solution: *d*-Heaps

Nearly complete *d*-ary trees (representable by array) with Heap-order property.

assume that $d = 2^a$ for some a

swap-down, -up:

$\#\text{steps} = \log_d n$

$= \log_{2^a}(n)$

$= \frac{\lg n}{\lg 2^a} = \frac{\lg n}{a}$

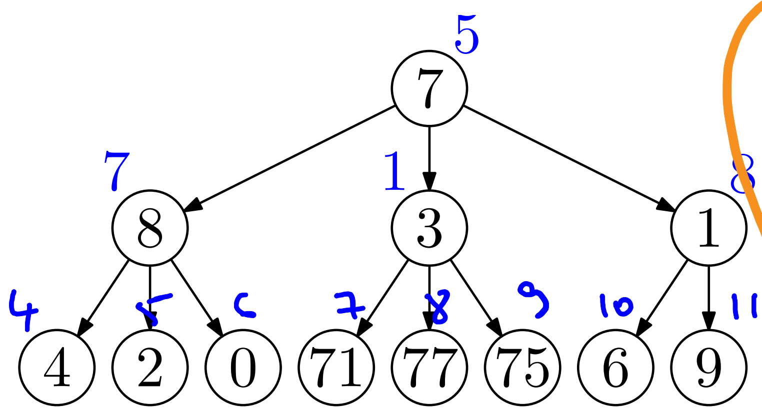
"a" times less steps

(Remark: $a = \lg d$)

$\approx \lg(d)$ times less pages need to be fetched from memory

time per step

$H = \log_d(n)$



7	8	3	1	4	2	0	71	77	75	6	9
5	7	1	8	4	0	6	3	2	9	75	77

time for swap-down: $\Theta(d \cdot \log_d n)$

swap-up: $\Theta(\log_d n)$

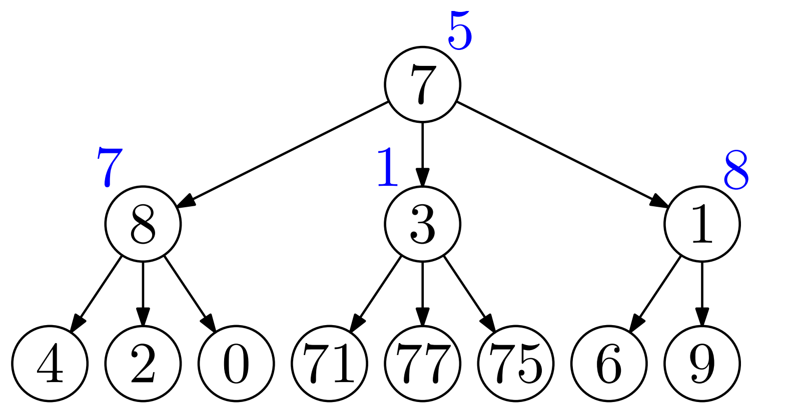
Good choices for *d*:

- 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

but runtime goes up $\approx d$ times . . . .

d-Heap Navigation

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



7	8	3	1	4	2	0	71	77	75	6	9
5	7	1	8	4	0	6	3	2	9	75	77