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

root: A

leaf: D E F I J K ··· N

child of B : D E F

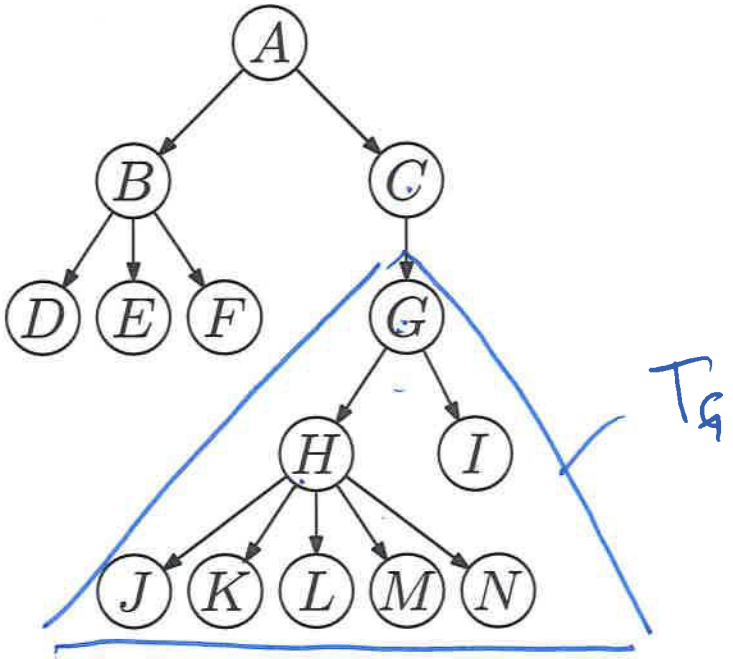
parent of I : G

sibling(s) of D : E F

ancestor of K : A C G H

descendant of G : H I J K ··· N

subtree of G : T_G



Tree Terminology Reference

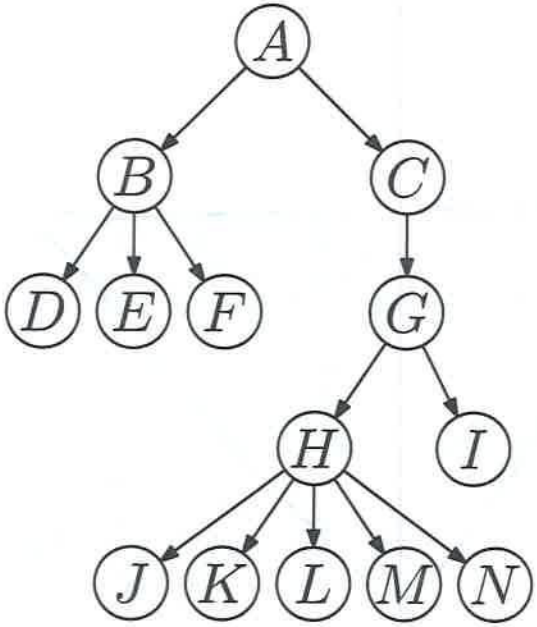
root: the single node with no parent

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

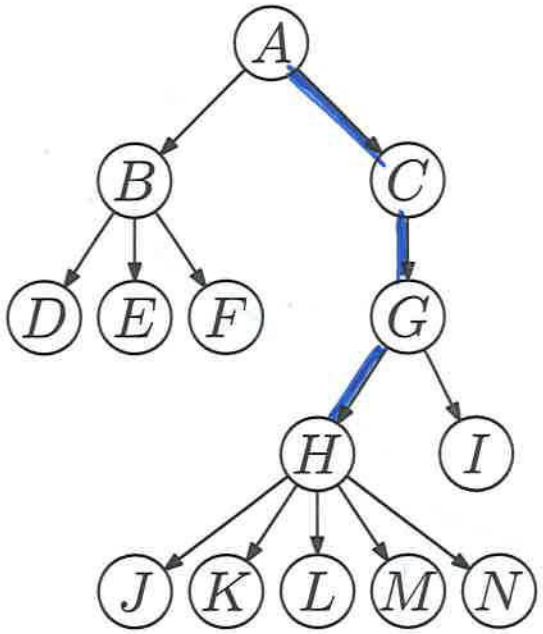
descendent: my child or my child's descendent

subtree: a node and its descendents

More Tree Terminology

depth: number of edges on path from root to node

depth of H? 3

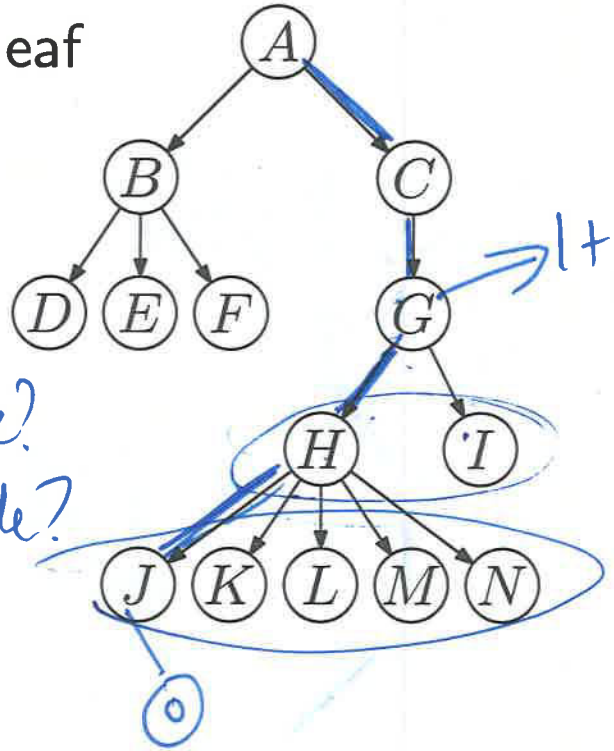


More Tree Terminology

height: number of edges on longest path from 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 the root. 4

height of G? 2



Q: How do we calculate the height of a tree?

↳ How do we calculate the height of a node?

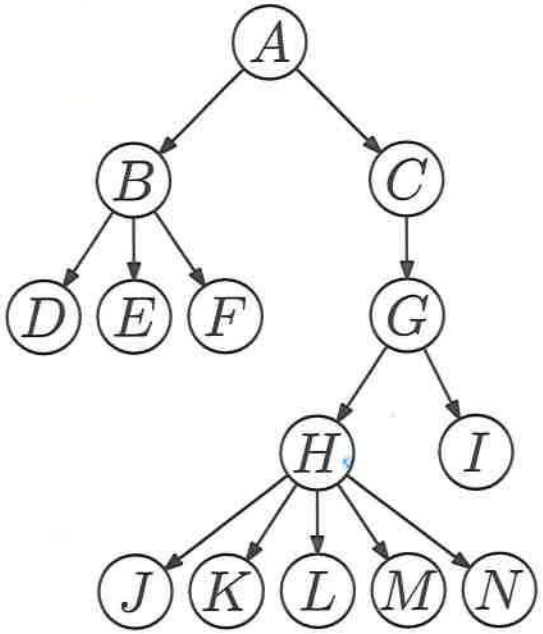
height (node)

- if node is a leaf → 0
- 1 + max (height (node → left), height (node → right))

More Tree Terminology

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

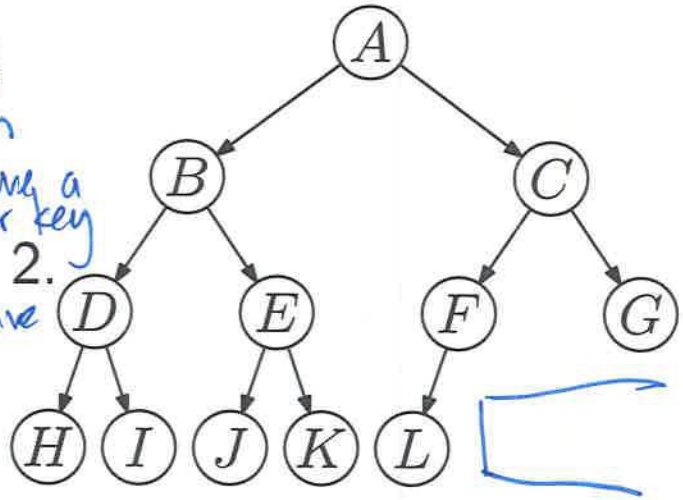
degree of B? 3



How many → 's

One More Tree-Terminology Slide

Binary Search Tree: invariant: given Node n
- All node in left-subtree have a smaller key
- All node in right subtree have a greater value.



binary: Each node has degree at most 2.

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

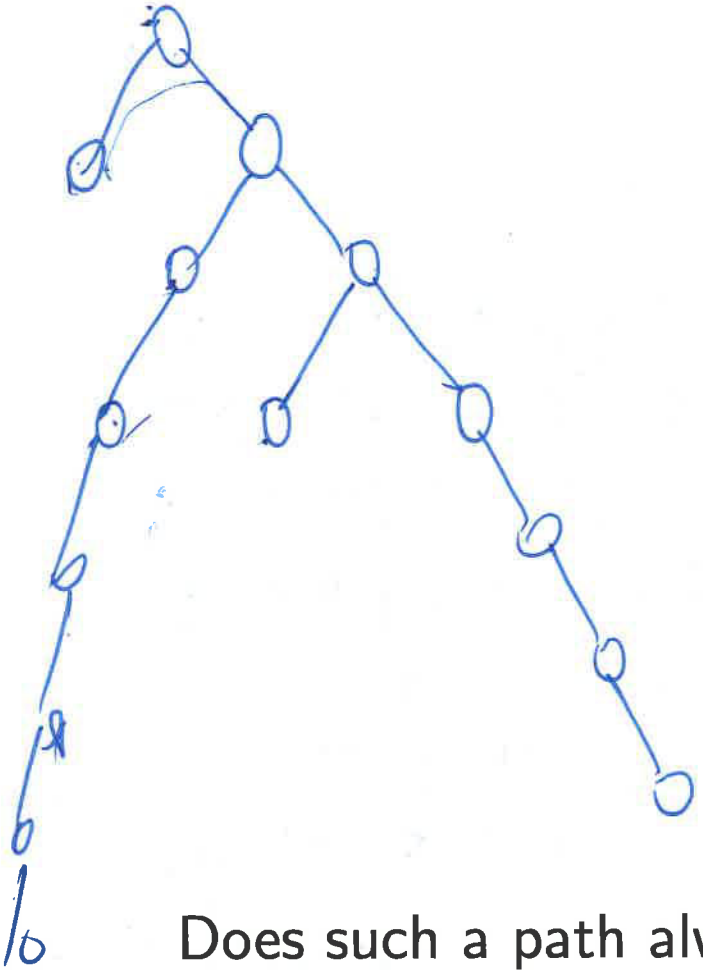
full: Each internal (non-leaf) node has the maximum number of children (2 in the case of a binary tree).

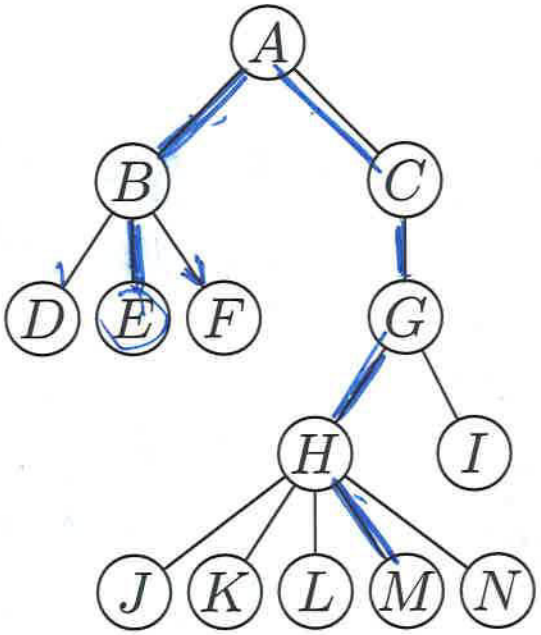
complete: It has as many nodes as possible for its height (i.e., each row is filled in).

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*.)

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

Example: Finding the Longest Undirected Path in a Tree



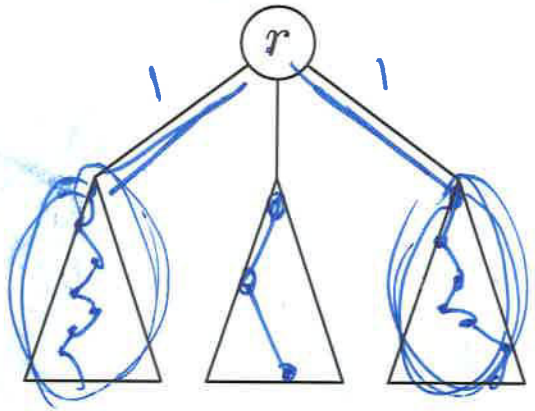


Does such a path always include the root? — No.

Longest Path

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

- Compute the maximum of:
  - If root is included in the solution
    - compute root's 2 tallest children (children with largest heights)
    - add those 2 heights together
    - add 2 (why?)
  - otherwise if root is not included
    - compute the longest path in a children's subtree



Back to Queues

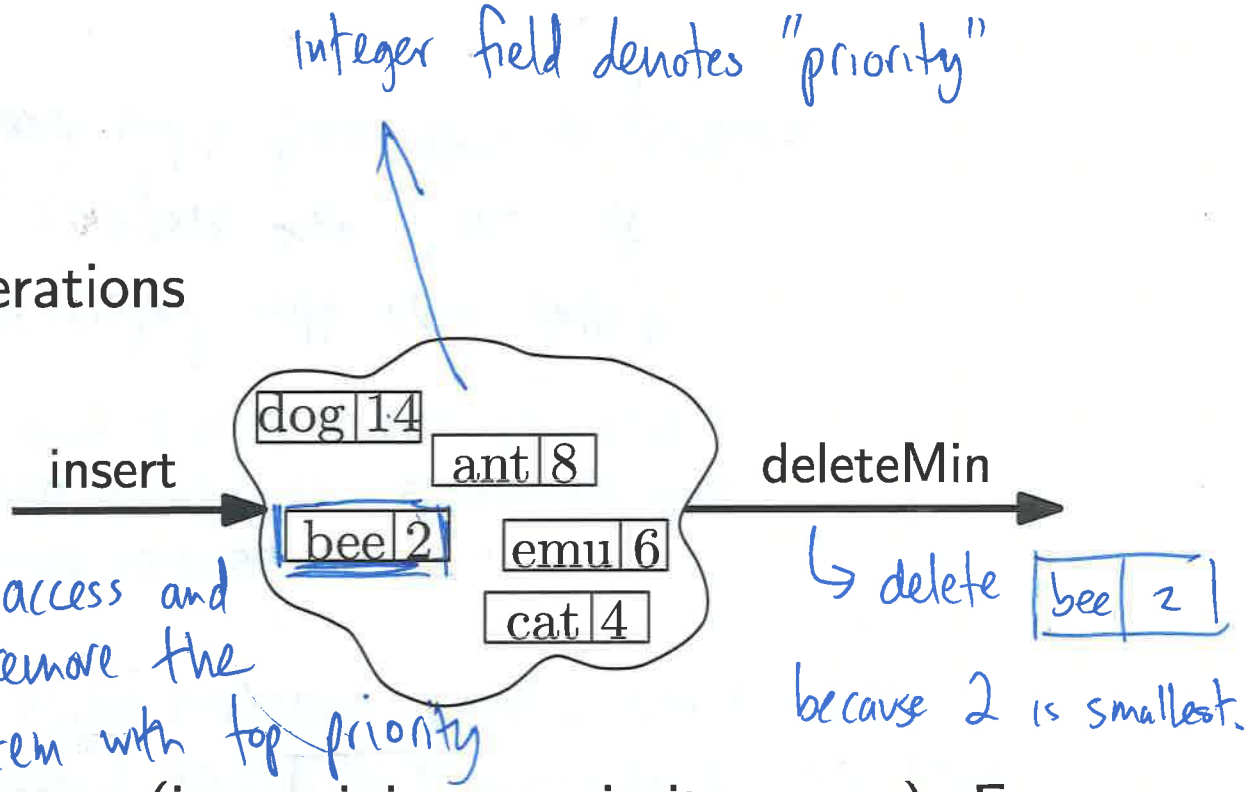
- Applications
  - Ordering jobs/processes on a CPU
  - Simulating events
  - Picking the next search site
- But we don't necessarily want FIFO. You can choose your order, according to some carefully thought-out priority. Maybe:
  - *Shorter* jobs should go first.
  - *Earliest* (simulated time) events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

different priority

- have some priority associated with each object.
  ↳ as long as we can compare two items of the same type and order them this is possible.

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

↳ arrays and linked lists

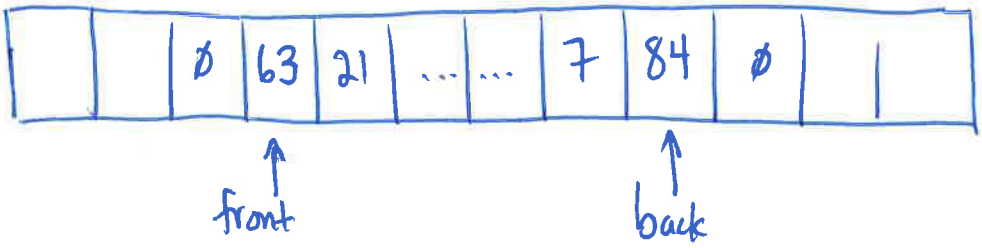
- Unsorted List
  - insert time: Array: $O(1)$, LL: $O(1)$
  - deleteMin time: array: $O(n)$, LL: $O(n)$

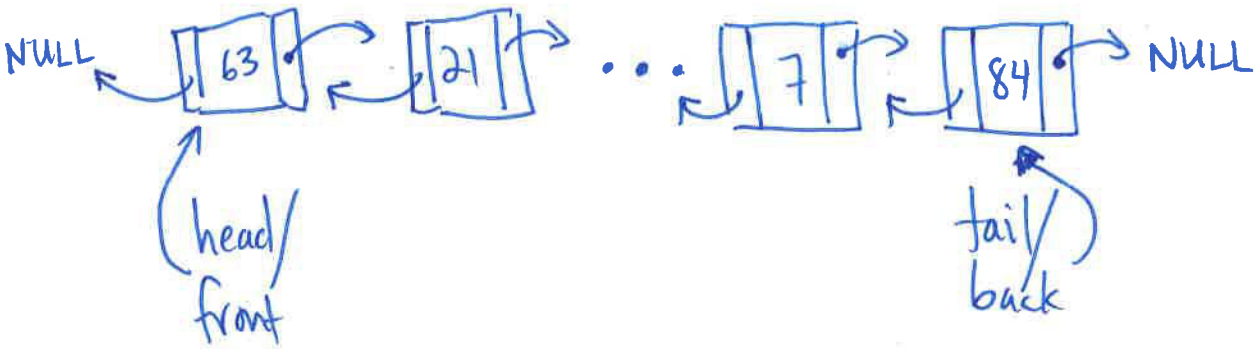
findMin	delete
$O(1) \Rightarrow O(n)$	$O(1) \Rightarrow O(n)$
$O(1) \Rightarrow O(n)$	$O(1) \Rightarrow O(n)$

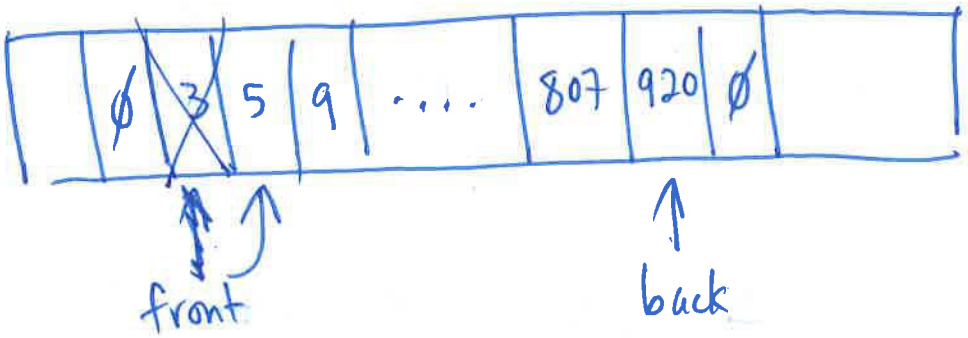
- Sorted List
  - insert time: array: $O(\log n)$, LL: $O(n)$
  - deleteMin time: fa findPos: Array: $O(1)$, LL: $O(1)$

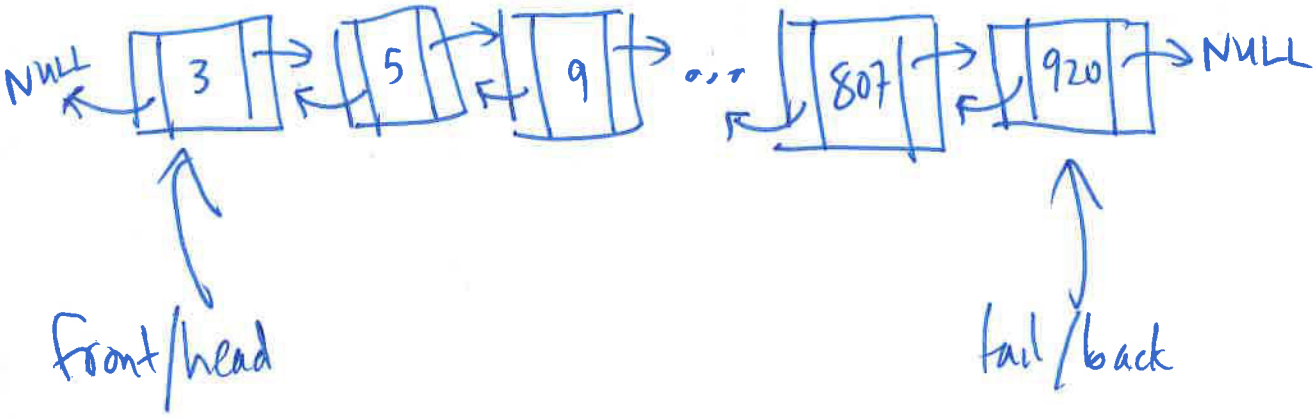
findPos	insert
$O(\log n)$	$O(n) \Rightarrow O(n)$
$O(n)$	$O(1) \Rightarrow O(n)$

findPos	delete
$O(1)$	$O(1) \Rightarrow O(1)$ (depends on implementation)
$O(1)$	$O(1) \Rightarrow O(1)$


Ø
63
21
...
...
7
84
Ø
front
back


NULL
63
21
7
84
NULL
head/
front
tail/
back


ø
3
5
9
....
807
920
ø
front
back


NULL
3
5
9
...
807
920
NULL
front/head
tail/back

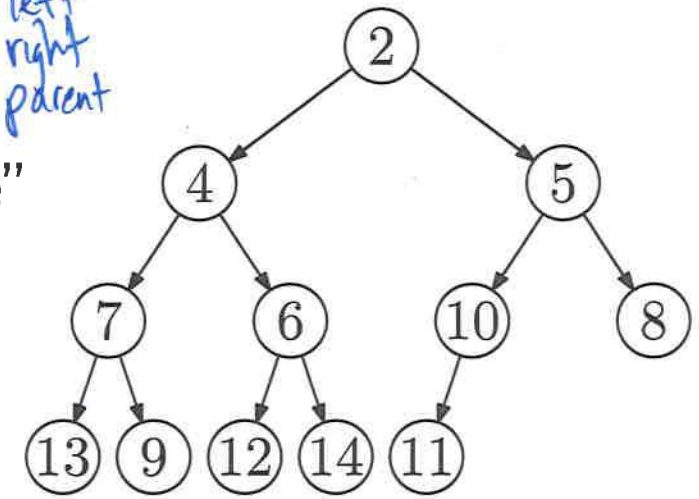
Binary Heap Priority Queue Data Structure

invariant important for adding and removing

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

Node
int data
Node* left
Node* right
Node* parent

- minimum is always at the top



Structure Property: "nearly complete tree"

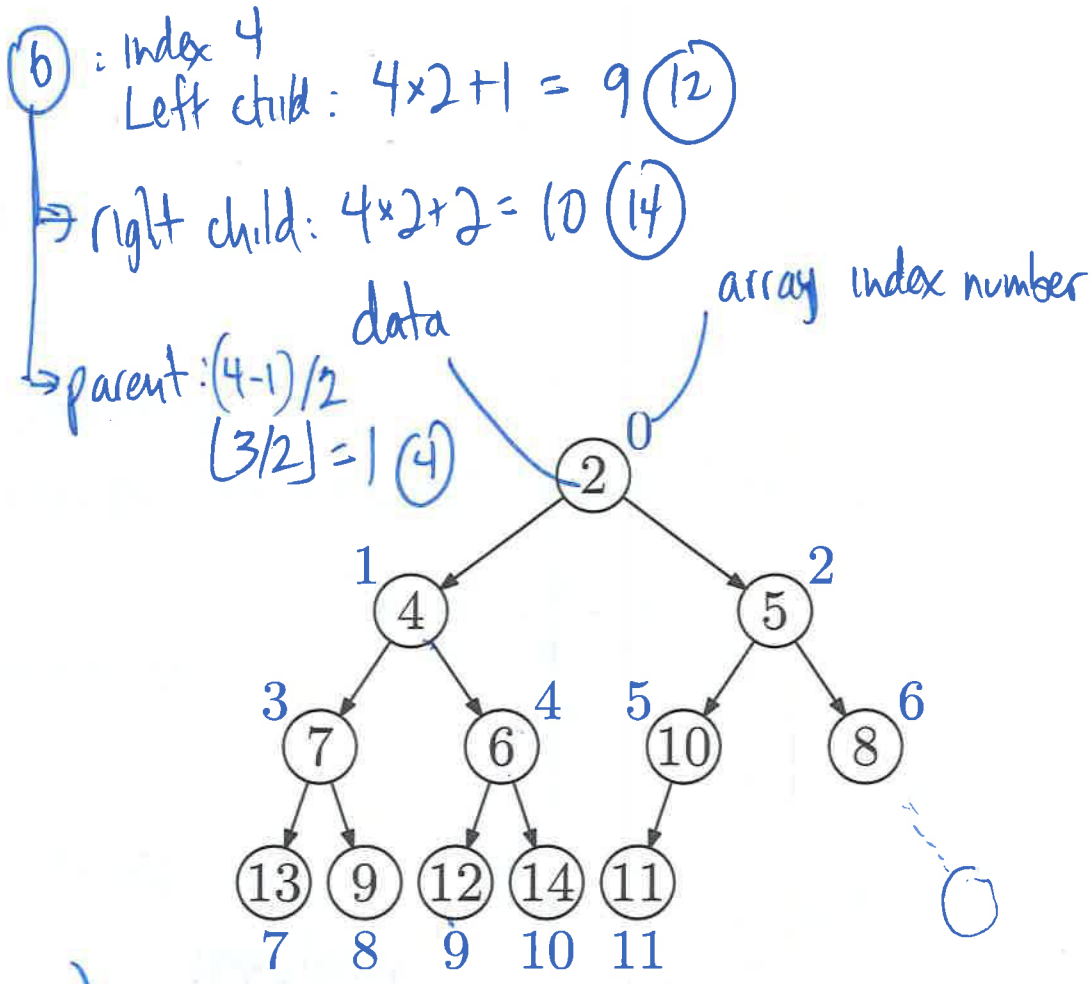
- depth is always O(lg n)
- next open location is always known

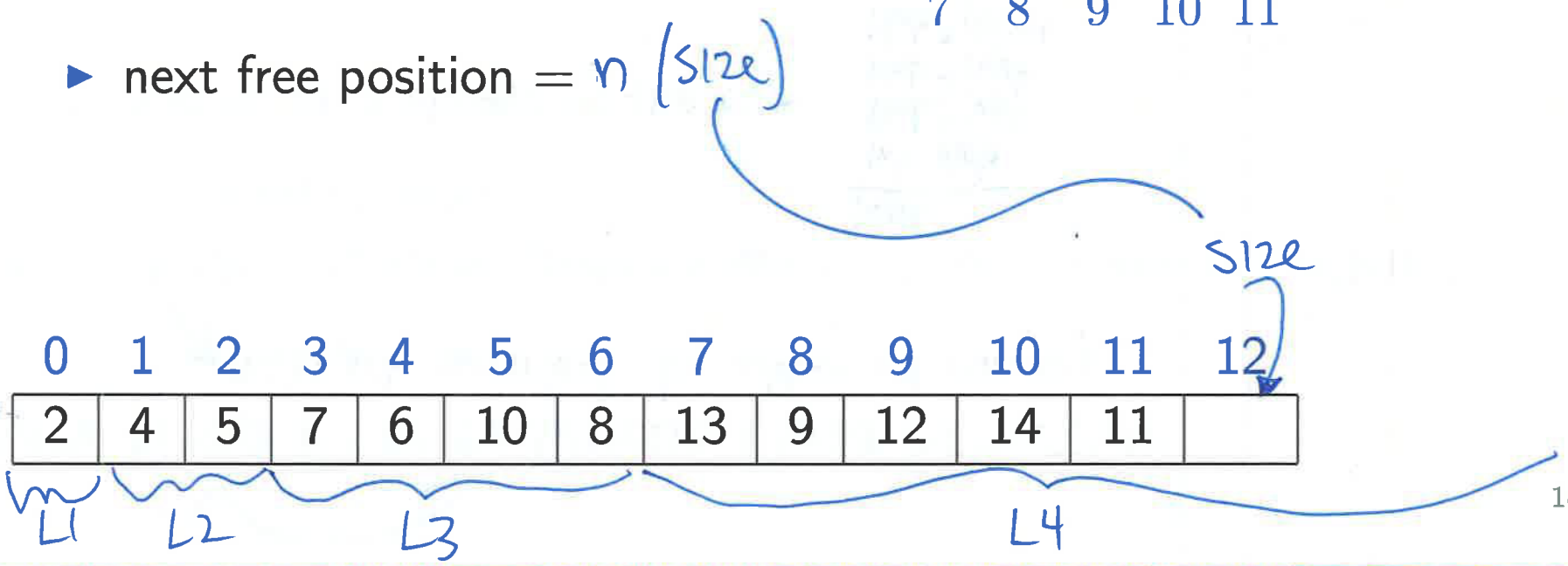
WARNING: This has no similarity to the memory "heap" we talk about when using C++'s `new` operator.

Nifty Storage Trick

Navigation using indices:

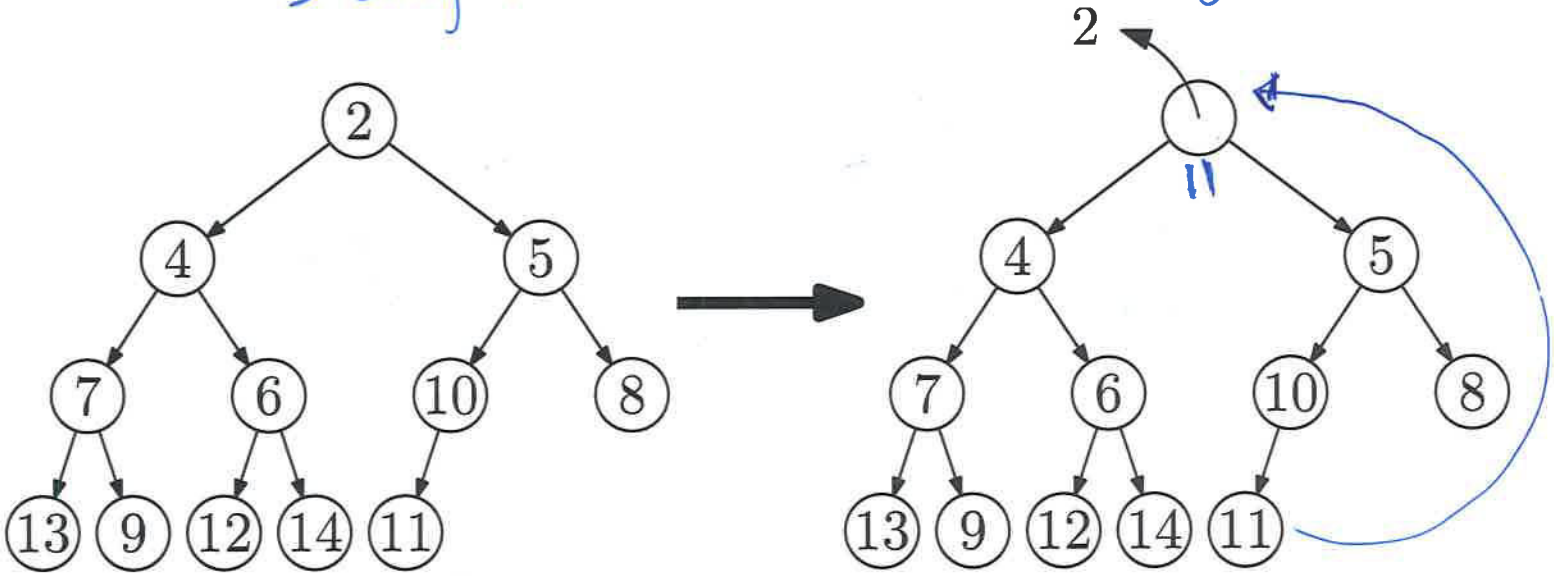
- left_child(i) = $2n+1$
- right_child(i) = $2n+2$
- parent(i) = $\lfloor (n-1)/2 \rfloor$
- root = 0
- next free position = n (size)

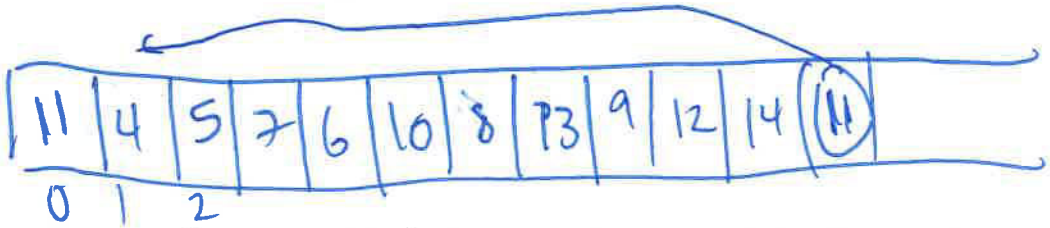




deleteMin

→ delete item in index 0.
→ bring last element from the back array to index 0.





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

↳ parent value <= children.

```
swap(int A[], int pos1, int pos2) {
    int temp = A[pos1];
    A[pos1] = A[pos2];
    A[pos2] = temp;
}
```

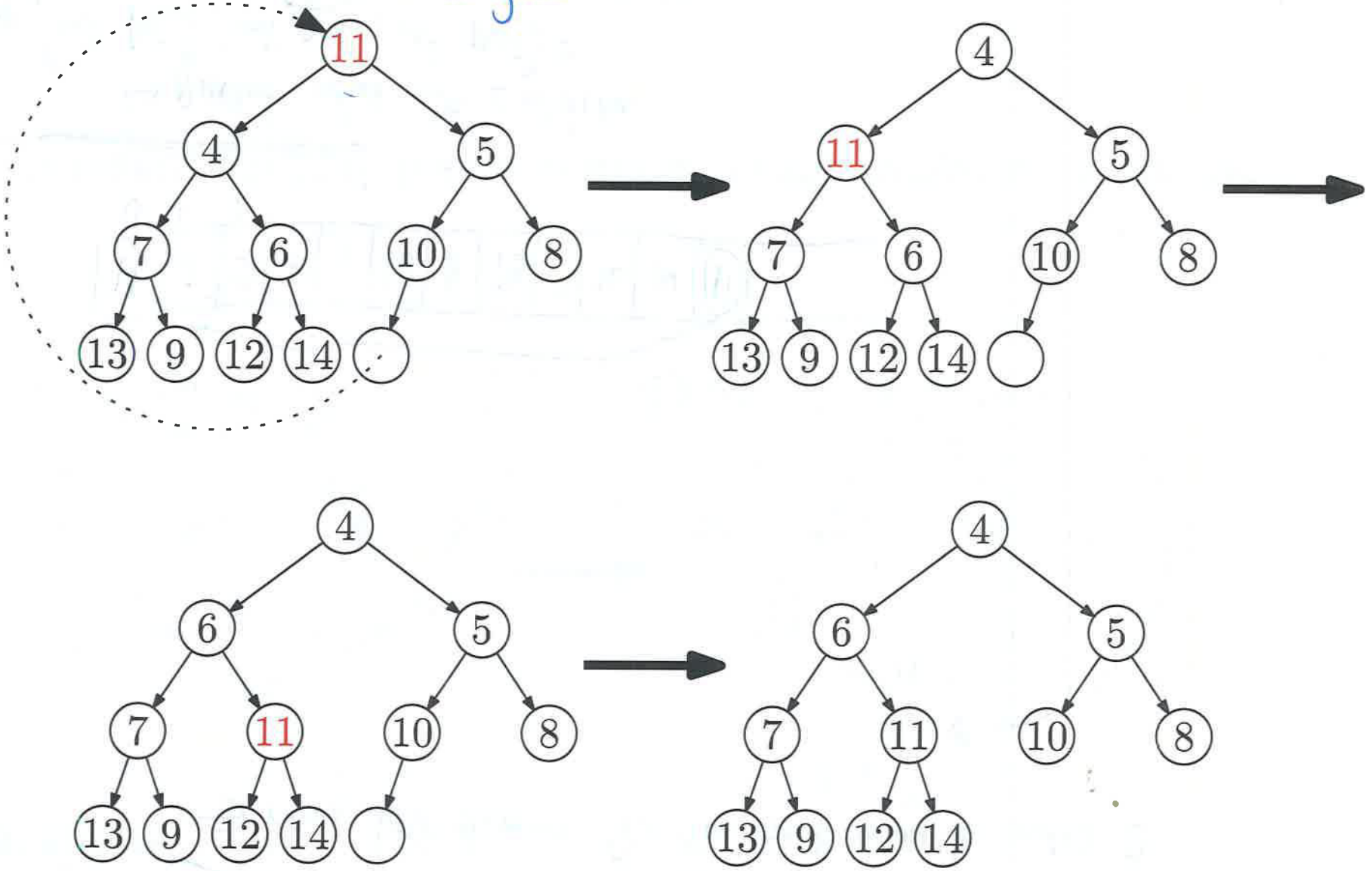
bubble.

Swap (Heapify) Down

While parent has greater value than a child swap with min child.

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

2 is gone.



deleteMin Code

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

n: size

Runtime:

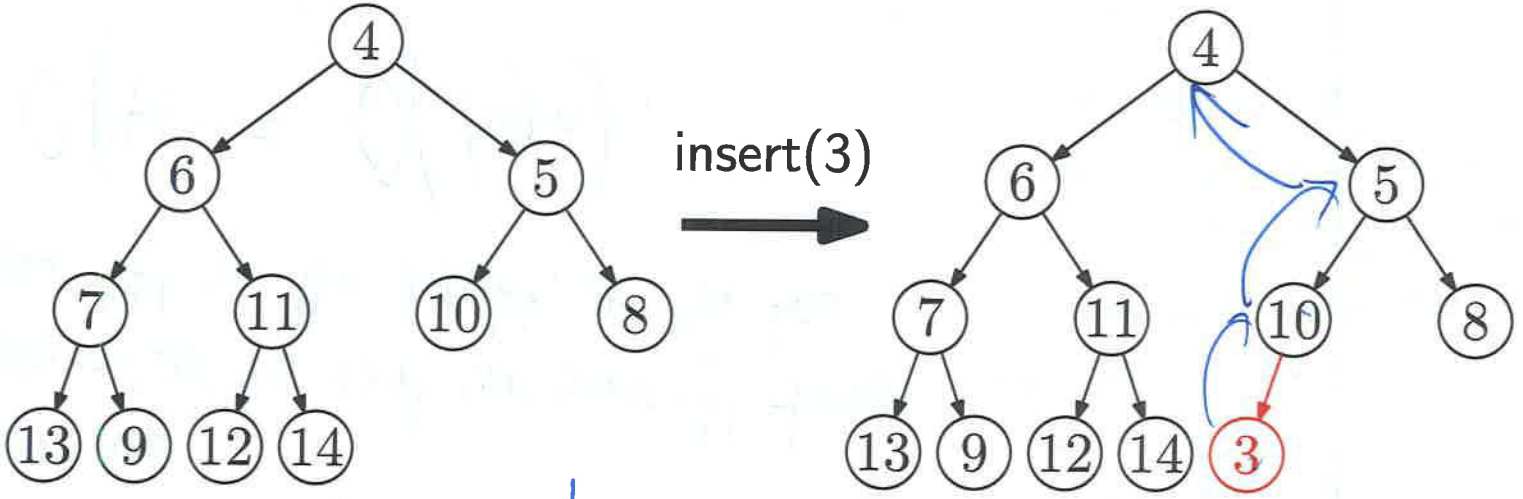
Worst-case is that we swap H times
Where H is the height of the tree

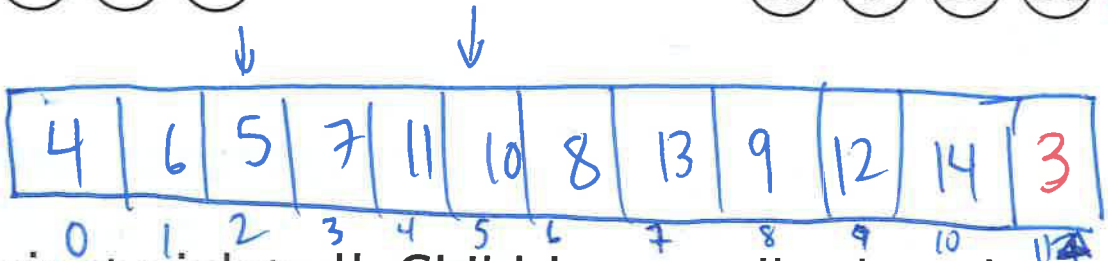
$O(H) = O(\log n)$.

```
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

parent index = (i-1)/2





Invariant violated! Child has smaller key than parent.

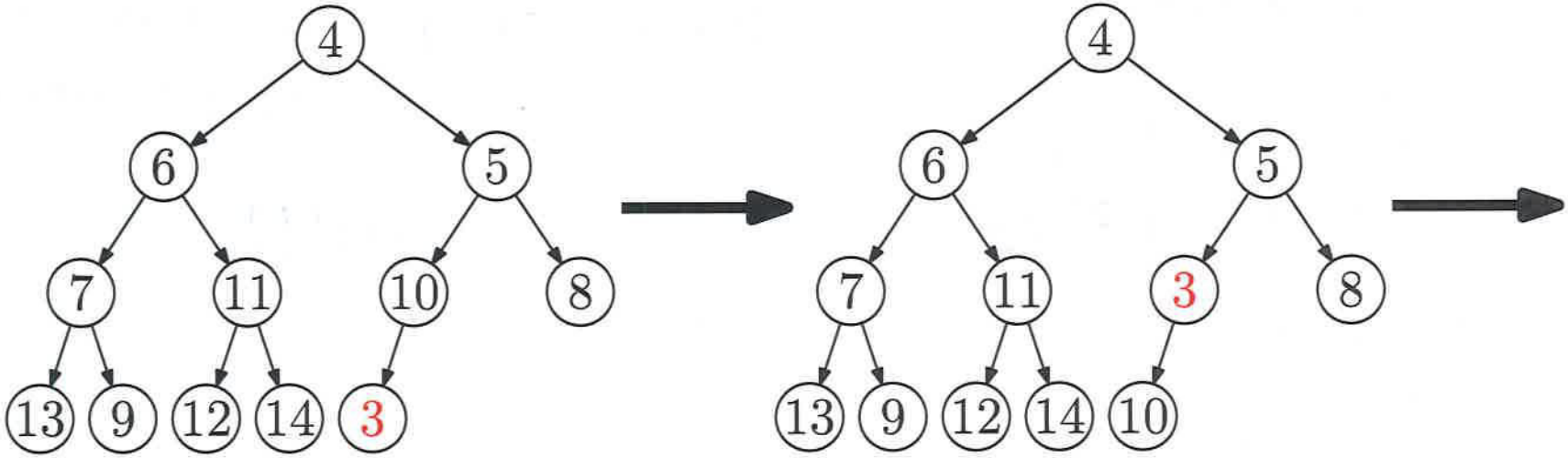
n
size

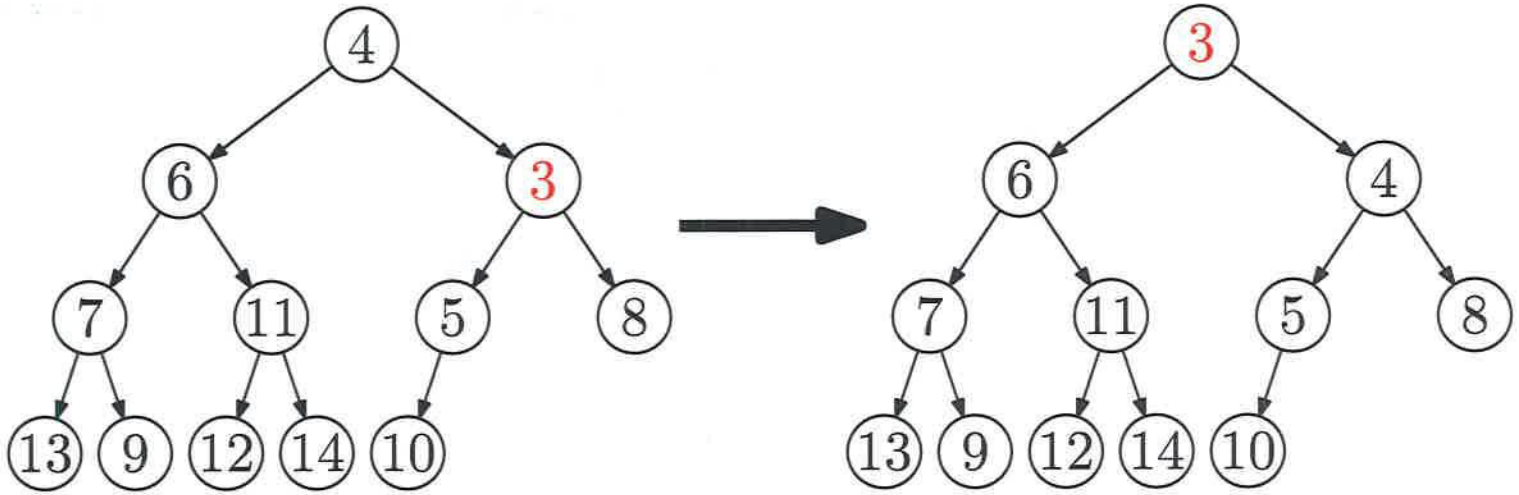
put item into array at index n (size).
(first empty spot in array)

Then, swap with parent until invariant is not violated.

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);
}
```

insert into first empty spot.

```
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);
  }
}
```

$p < i/2$

Runtime: $O(\log n)$

$\Rightarrow$ total number of swapUps is $\log n$

$\downarrow$

Can only swap the height of a tree

* Remember slide 16:

Using arrays and linked lists it was $O(n)$

and now we can insert and delete in $O(\log n)$

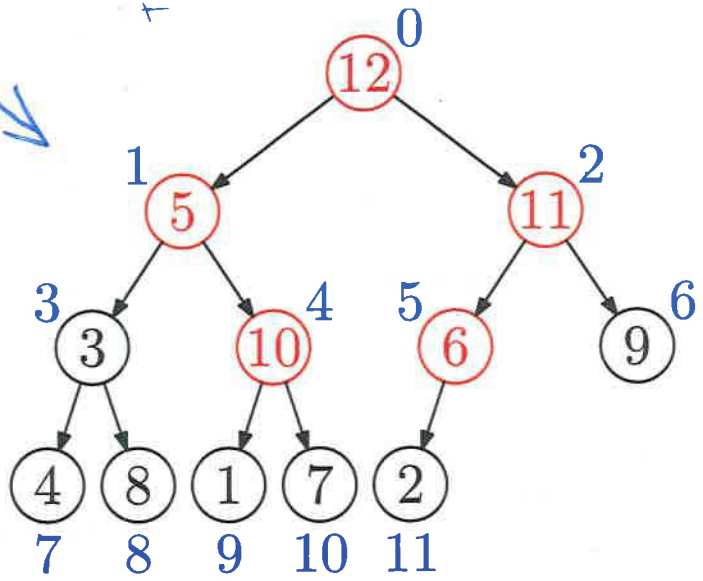
Heapify: Build a Heap from an Array

log1 + log2 + log3 + ... + logn
Ex: 9 (slide 37) from Unit #1
↓
$\Theta(n \lg n)$

1. Start with the input array. (unsorted)

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

tree/heap representation of array



Invariant violated!

OPTION 1:
start with empty array. Add each item and reorder.

OPTION 2:
Put all items into array, and then reorder once.

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

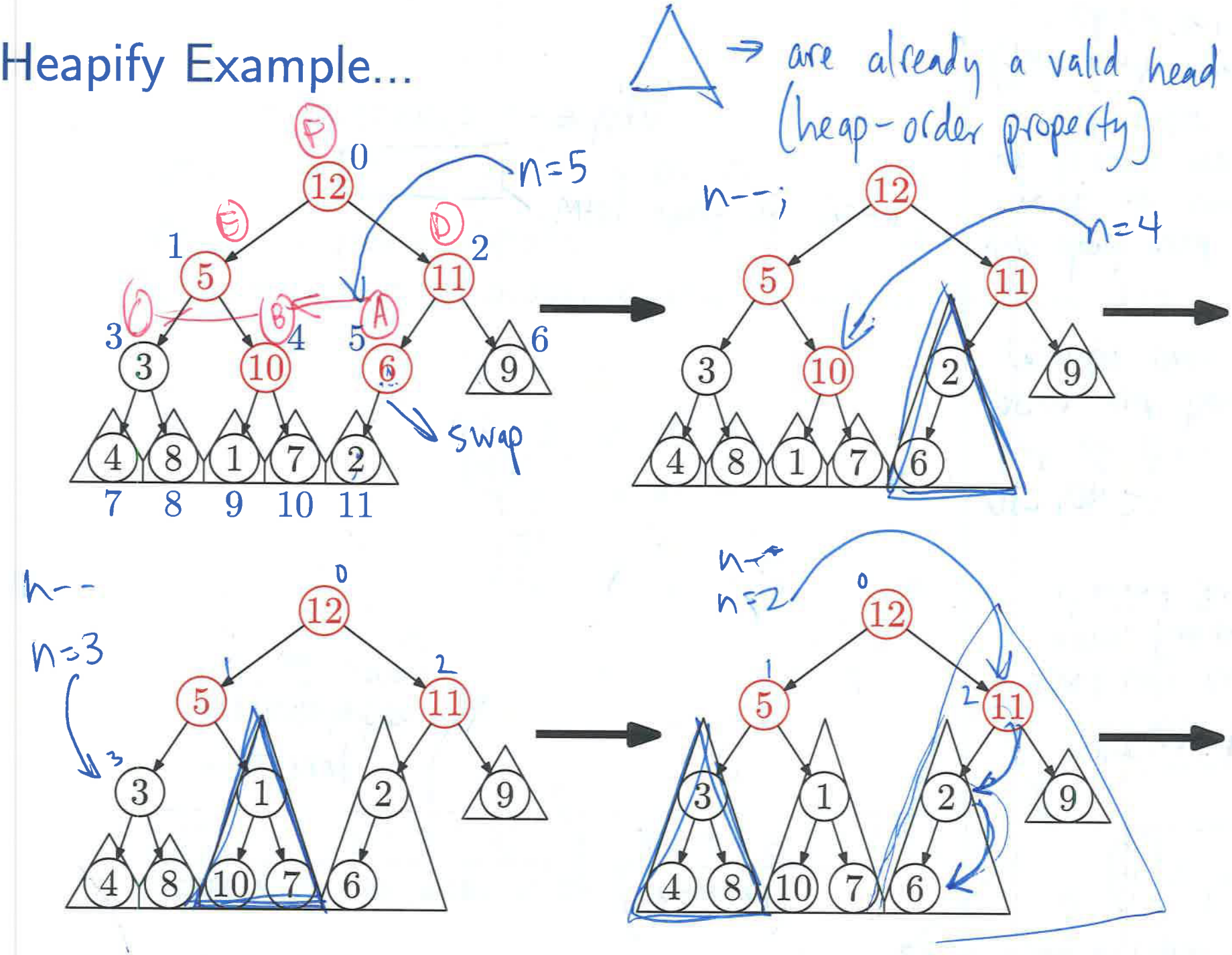
```
for( i = n/2 - 1; i >= 0; i-- )
    swapDown(i);
```

Why start at n/2 - 1?

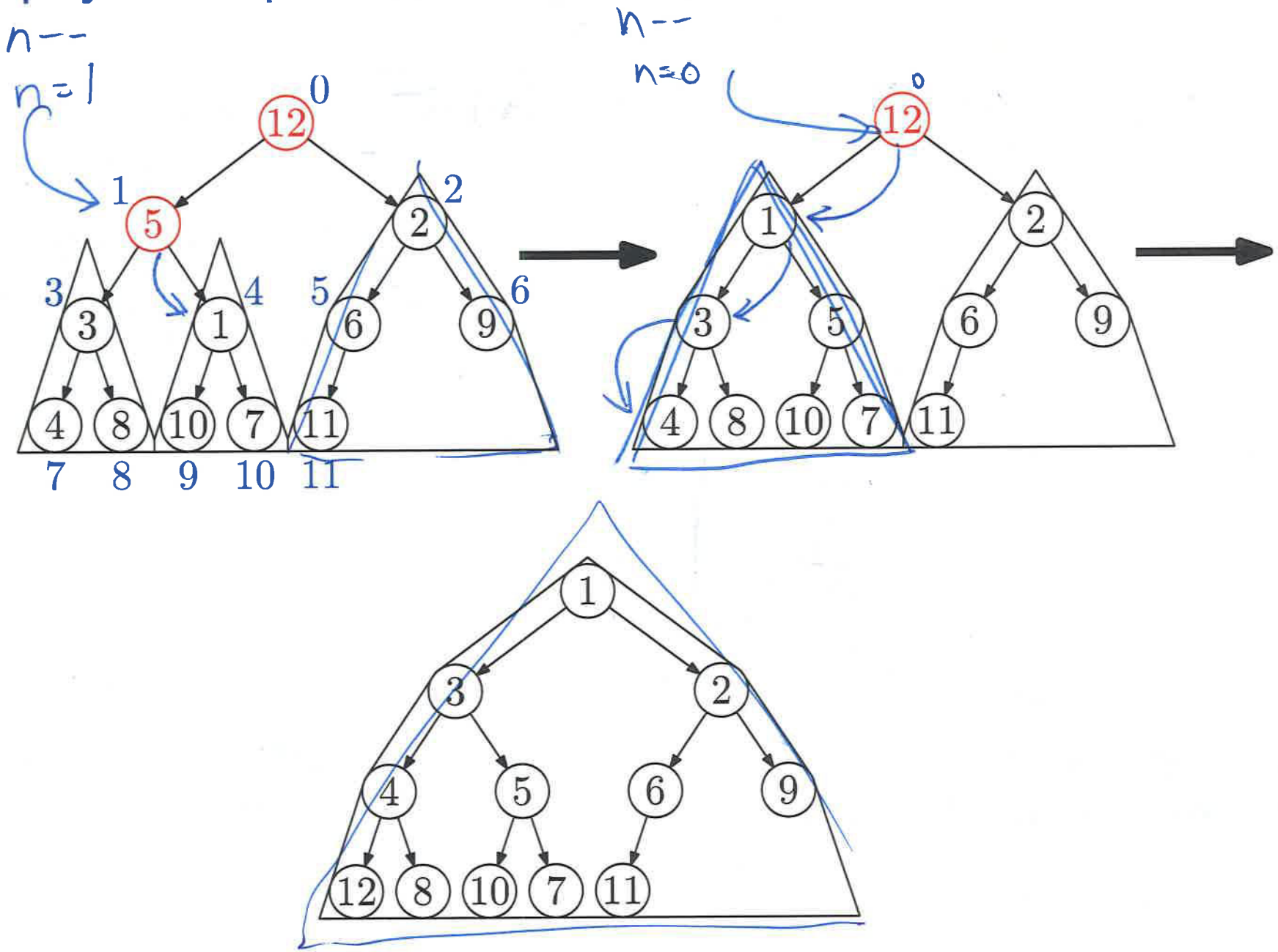
→ logn

We don't need to check if children of leaves are out of order (they don't have children)

Heapify Example...

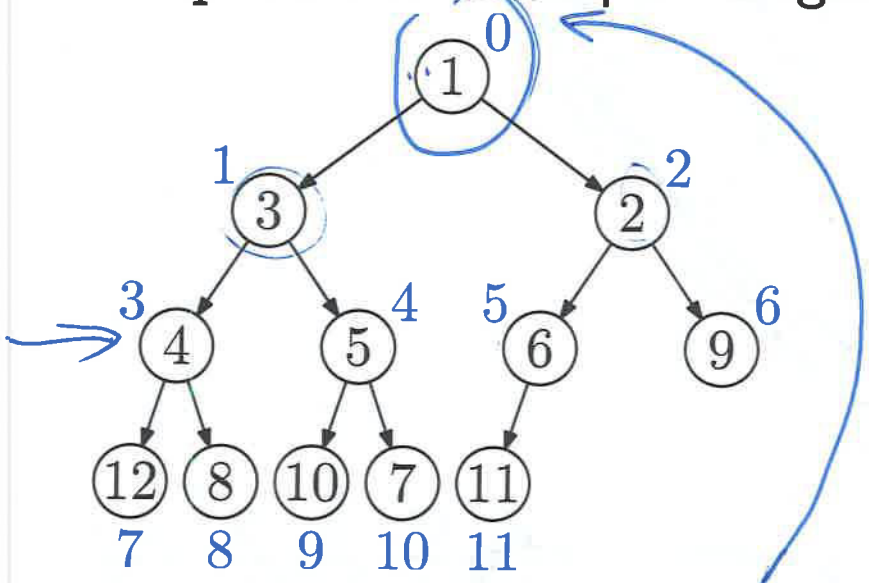


Heapify Example



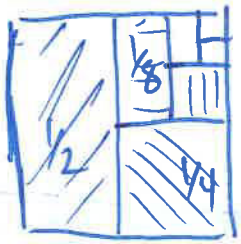
Heapify Runtime

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



will only ever need to swap the root or item at index 0 H times

Let H be the height of the heap.



`swapDown` is called

once	on heap of height	H
≤ 2 times	on heap of height	$H-1$
≤ 4 times	on heap of height	$H-2$
$\vdots$		
$\leq 2^{H-1}$ times	on heap of height	1

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

S

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

$-S \quad -\frac{1}{2} - \frac{2}{4} - \frac{3}{8} - \ldots$

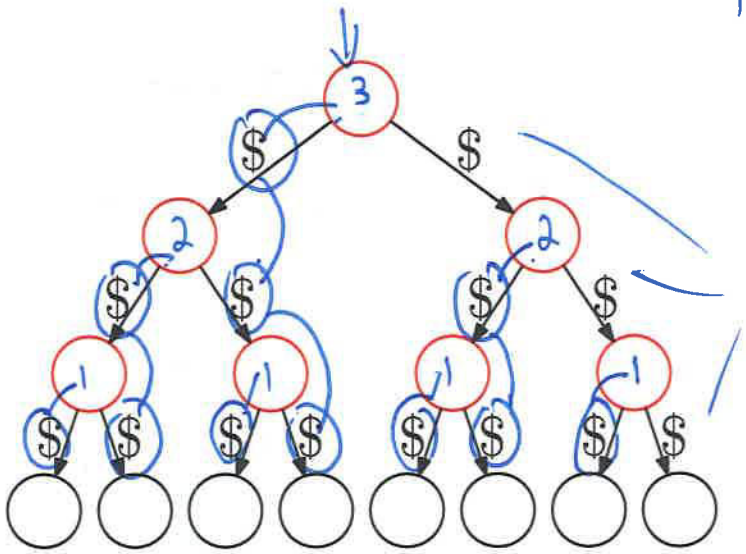
$S = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots \Rightarrow 1$

$S = 2$

Total # steps $\leq \sum_{h=1}^{H} h2^{H-h} = 2^H \sum_{h=1}^{H} h/2^h \leq 2^{H+1} = O(n)$

$2^{\log n + 1} = 2^{\log n} \times 2 = n \times 2 \therefore O(n)$

Heapify Runtime: Charging Scheme



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

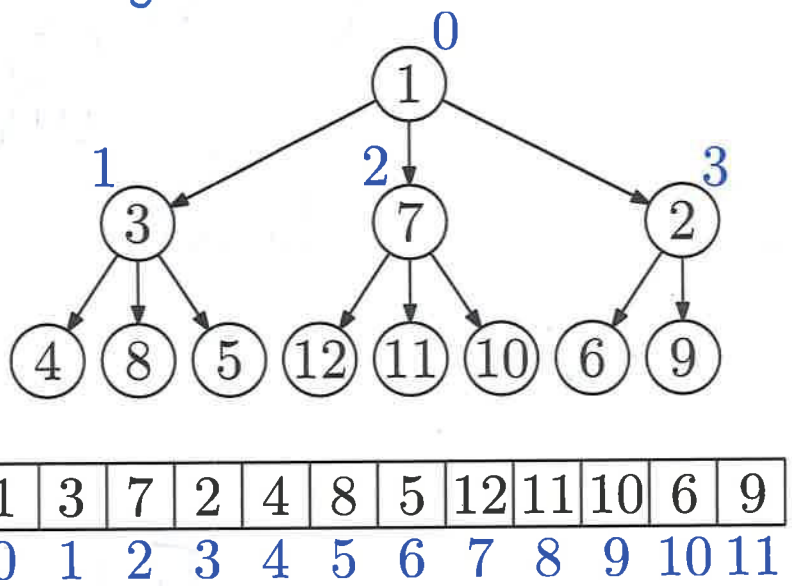
- Finding a child/parent index is a multiply/divide by two operation.
- Both deleteMin and the subsequent insert might access far-apart array locations.
- deleteMin accesses all children of visited nodes. → swapDown
- insert accesses only the parent of visited nodes. → swapUp
- insert is at least as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

Realities

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

Solution: d-Heaps

These are 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

When $d=3$ → nodes have 3 children

at index i

left child(i): $3i+1$

mid child(i): $3i+2$

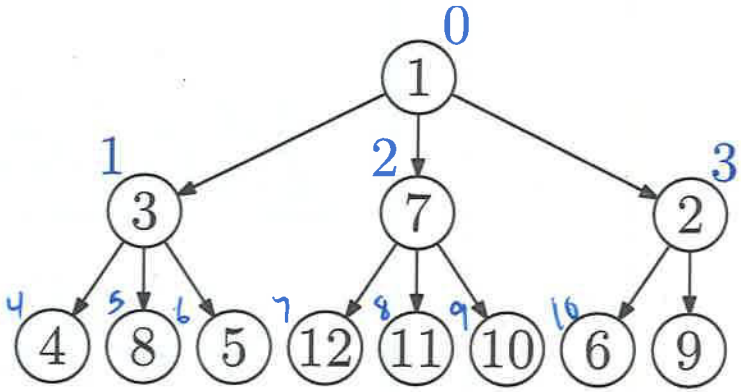
right child(i): $3i+3$

root: 0

next free pos: n (size)

Ex: $i=2$ (7)

- jth-child(i) $= di + j$
- parent(i) $= \lfloor (i-1)/d \rfloor$
- root $= 0$
- next free position $= n$ (size)



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

n
size

height: $\log_d n$

swap-down: $O(d \log_d n)$

be careful if d gets very large (close to n)