CSE373: Data Structures & Algorithms

Lecture 8: Priority Queues

Aaron Bauer
Winter 2014

Announcements

- Midterm next week
  - Midterm review TA session on Tuesday
  - Shuo extra office hours 12:30-1:30 Monday
- Homework 1 feedback out soon

Priority Queue ADT

- Stores elements with data and comparable priorities
  - "priority 1" is more important than "priority 4"
- Operations
  - **`insert`**
  - **`deleteMin`**
  - **`is_empty`**

Applications

Like all good ADTs, the priority queue arises often

- Sometimes blatant, sometimes less obvious

- Run multiple programs in the operating system
  - "critical" before "interactive" before "compute-intensive"
  - Maybe let users set priority level
- Treat hospital patients in order of severity (or triage)
- Select print jobs in order of decreasing length?
- Forward network packets in order of urgency
- Select most frequent symbols for data compression (cf. CSE143)
- Sort (first **`insert`** all, then repeatedly **`deleteMin`**)
  - Much like Homework 1 uses a stack to implement reverse

More applications

- "Greedy" algorithms
  - May see an example when we study graphs in a few weeks
- Discrete event simulation (system simulation, virtual worlds, …)
  - Each event *e* happens at some time *t*, updating system state and generating new events *e1*, …, *en* at times *t*+*t1*, …, *t*+*tn*
  - Naïve approach: advance "clock" by 1 unit at a time and process any events that happen then
  - Better:
    - *Pending events* in a priority queue (priority = event time)
    - Repeatedly: **`deleteMin`** and then **`insert`** new events
    - Effectively "set clock ahead to next event"

Finding a good data structure

- Will show an efficient, non-obvious data structure for this ADT
  - But first let's analyze some "obvious" ideas for *n* data items
  - All times worst-case; assume arrays "have room"

data	*insert algorithm / time*		*deleteMin algorithm / time*	
unsorted array	add at end	$O(1)$	search	$O(n)$
unsorted linked list	add at front	$O(1)$	search	$O(n)$
sorted circular array	search / shift	$O(n)$	move front	$O(1)$
sorted linked list	put in right place	$O(n)$	remove at front	$O(1)$
binary search tree	put in right place	$O(n)$	leftmost	$O(n)$
AVL tree	put in right place	$O(\log n)$	leftmost	$O(\log n)$

More on possibilities

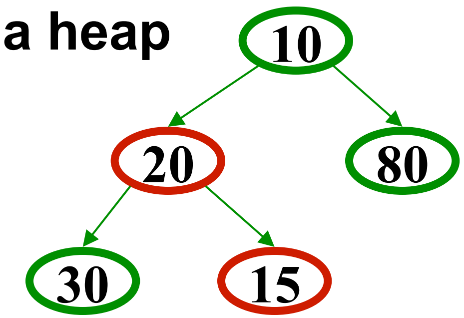
- *If* priorities are random, binary search tree will likely do better
  - *O*(`log` *n*) `insert` and *O*(`log` *n*) `deleteMin` on *average*
- One more idea: if priorities are 0, 1, …, *k* can use array of lists
  - `insert`: add to front of list at `arr[priority]`, *O*(1)
  - `deleteMin`: remove from lowest non-empty list *O(k)*
- We are about to see a data structure called a "binary heap"
  - *O*(`log` *n*) `insert` and *O*(`log` *n*) `deleteMin` *worst*-*case*
    - Possible because we don't support unneeded operations; no need to maintain a full sort
  - *Very* good constant factors
  - *If* items arrive in random order, then `insert` is *O*(1) on *average*

Our data structure

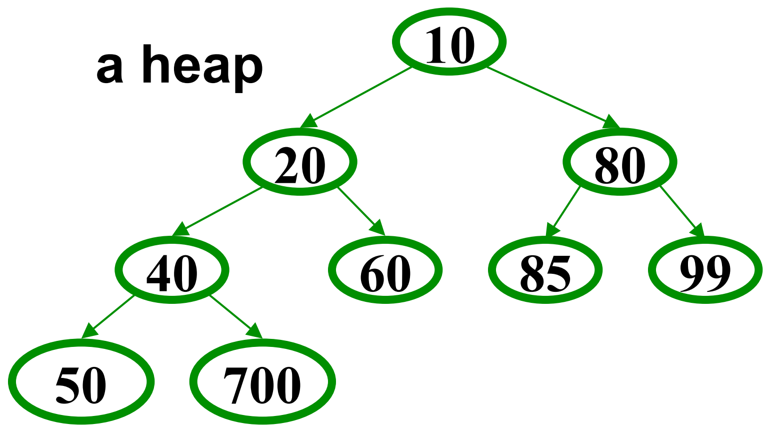
A *binary min-heap* (or just *binary heap* or just *heap*) is:

- Structure property: A *complete* binary tree
- Heap property: The priority of every (non-root) node is greater than the priority of its parent
  - ***Not*** a binary search tree

not a heap



a heap

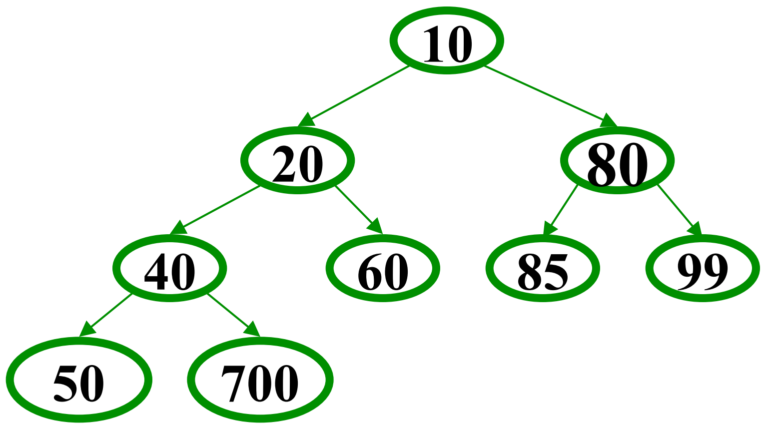


So:

- Where is the highest-priority item?
- What is the height of a heap with *n* items?

Operations: basic idea

- **`findMin`**: return **`root.data`**
- **`deleteMin`**:
  1. **`answer = root.data`**
  2. Move right-most node in last row to root to restore structure property
  3. "Percolate down" to restore heap property
- **`insert:`**
  1. Put new node in next position on bottom row to restore structure property
  2. "Percolate up" to restore heap property

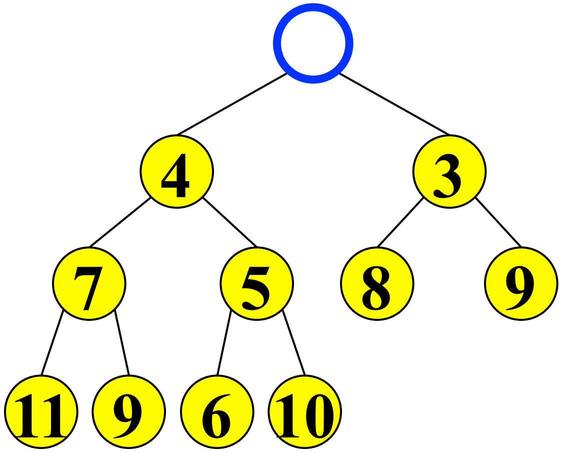


Overall strategy:

- *Preserve structure property*
- *Break and restore heap property*

DeleteMin

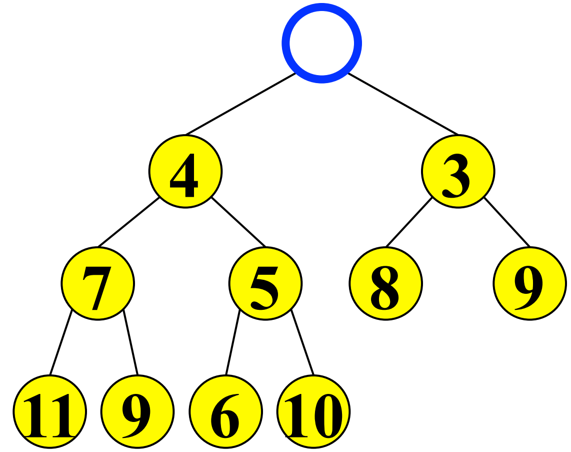
1. Delete (and later return) value at root node

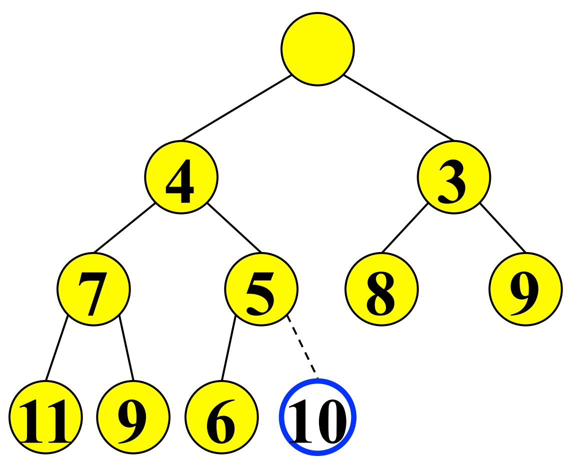


2. Restore the Structure Property

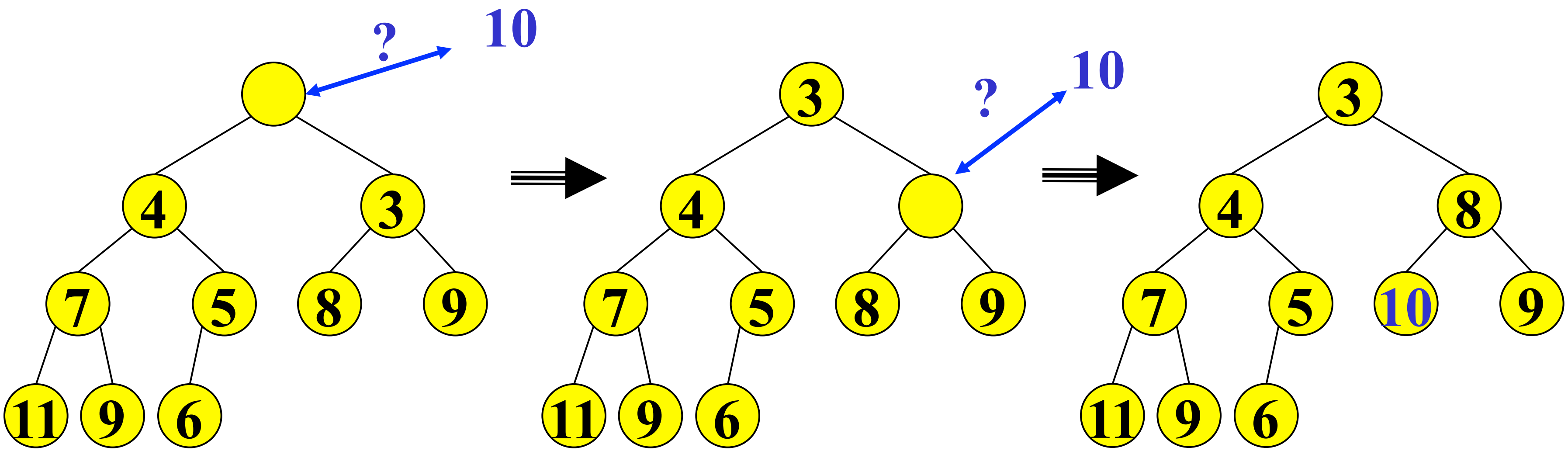
- We now have a "hole" at the root
  - Need to fill the hole with another value

- When we are done, the tree will have one less node and must still be complete





3. Restore the Heap Property



Percolate down:
- Keep comparing with both children
- Swap with lesser child and go down one level
- Done if both children are ≥ item or reached a leaf node

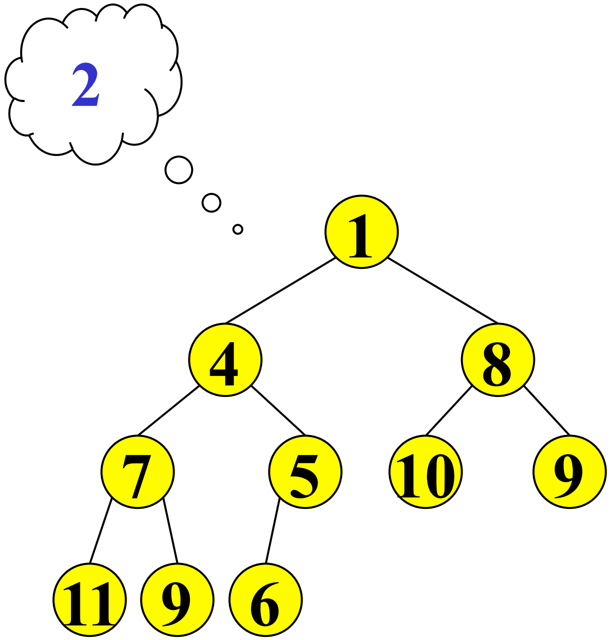
Why is this correct? What is the run time?

DeleteMin: Run Time Analysis

- Run time is *O*(height of heap)
- A heap is a complete binary tree
- Height of a complete binary tree of *n* nodes?
  - height = $\lfloor \log_2(n) \rfloor$
- Run time of **`deleteMin`** is $O(\log n)$

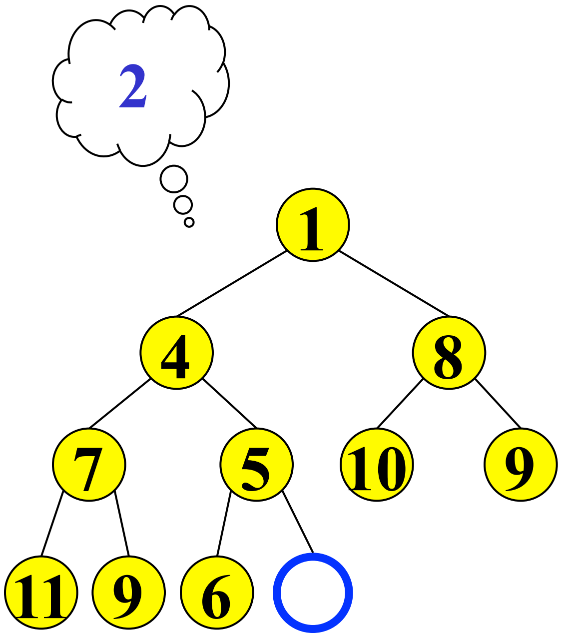
Insert

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct

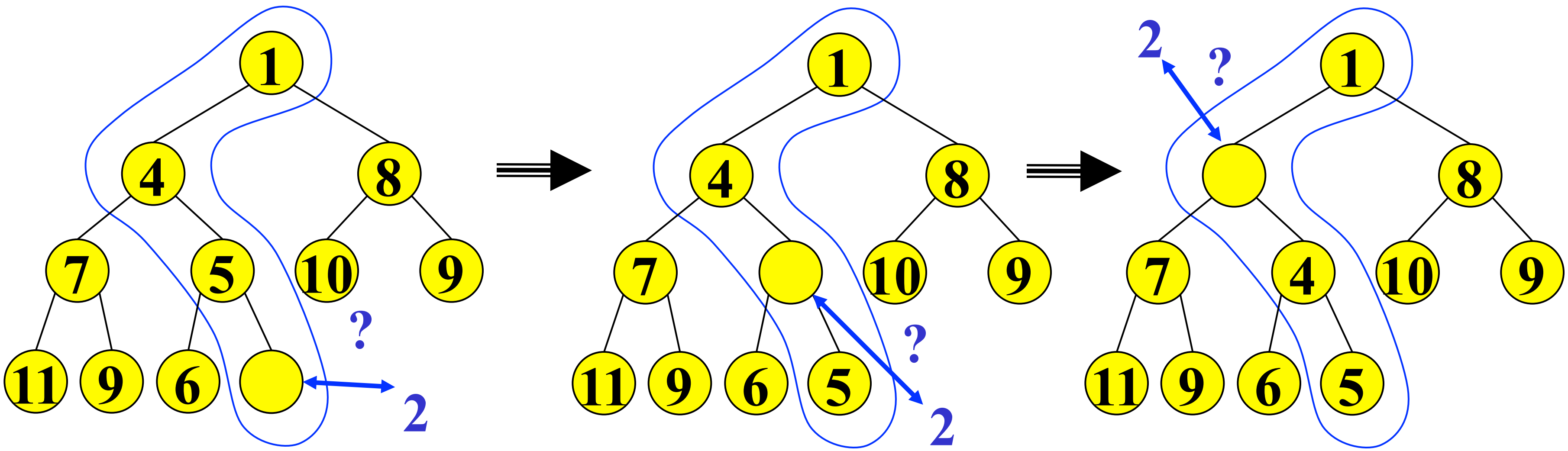


Insert: Maintain the Structure Property

- There is only one valid tree shape after we add one more node
- So put our new data there and then focus on restoring the heap property



Maintain the heap property



Percolate up:

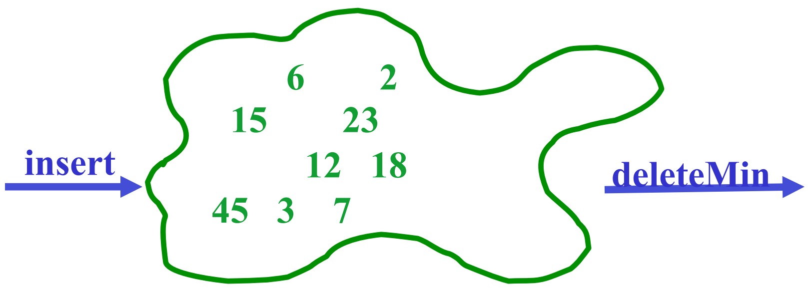
- Put new data in new location
- If parent larger, swap with parent, and continue
- Done if parent ≤ item or reached root

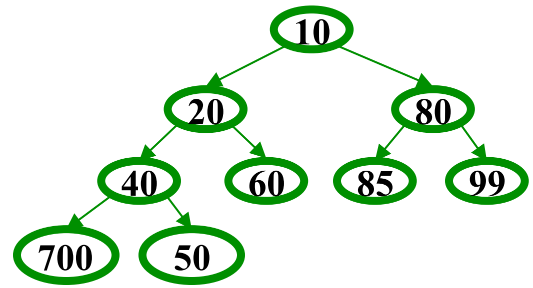
Why is this correct? What is the run time?

Insert: Run Time Analysis

- Like **`deleteMin`**, worst-case time proportional to tree height
  - *O*(**`log`** *n*)
- But… **`deleteMin`** needs the “last used” complete-tree position and **`insert`** needs the “next to use” complete-tree position
  - If “keep a reference to there” then **`insert`** and **`deleteMin`** have to adjust that reference: *O*(**`log`** *n*) in worst case
  - Could calculate how to find it in *O*(**`log`** *n*) from the root given the size of the heap
    - But it’s not easy
    - And then **`insert`** is always *O*(**`log`** *n*), promised *O*(1) on average (assuming random arrival of items)
- There’s a “trick”: don’t represent complete trees with explicit edges!

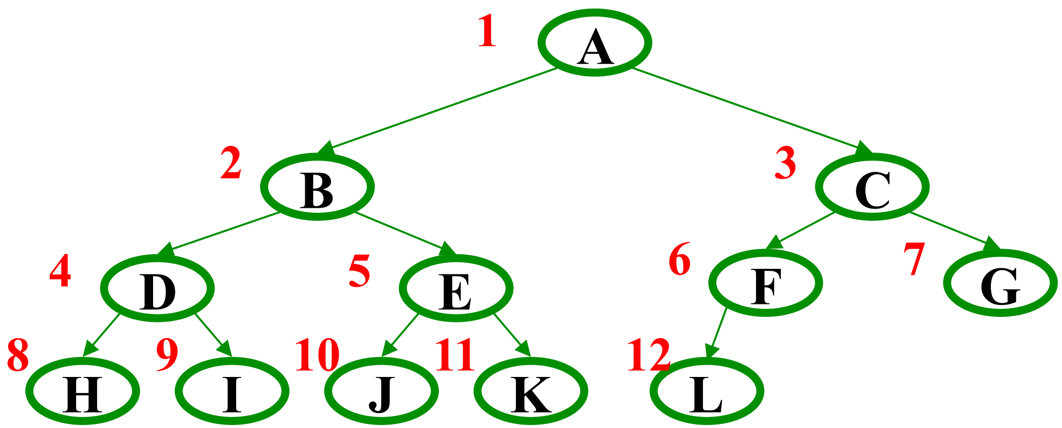
Review





- Priority Queue ADT: **`insert`** comparable object, **`deleteMin`**
- Binary heap data structure: Complete binary tree where each node has priority value greater than its parent
- *O*(height-of-tree)=*O*(**`log`** *n*) **`insert`** and **`deleteMin`** operations
  - **`insert`**: put at new last position in tree and percolate-up
  - **`deleteMin`**: remove root, put last element at root and percolate-down
- But: tracking the “last position” is painful and we can do better

Array Representation of Binary Trees



From node **`i`**:

left child: **`i*2`**
right child: **`i*2+1`**
parent: **`i/2`**

(wasting index 0 is convenient for the index arithmetic)

implicit (array) implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Judging the array implementation

Plusses:

- Non-data space: just index 0 and unused space on right
  - In conventional tree representation, one edge per node (except for root), so *n*-1 wasted space (like linked lists)
  - Array would waste more space if tree were not complete
- Multiplying and dividing by 2 is very fast (shift operations in hardware)
- Last used position is just index **`size`**

Minuses:

- Same might-be-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

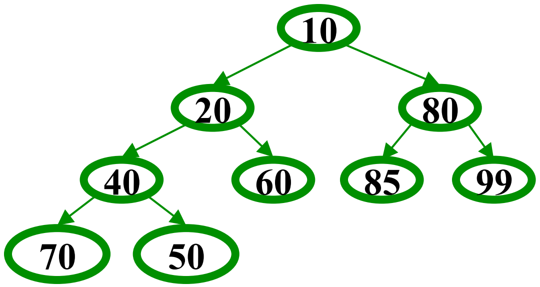
Plusses outweigh minuses: “this is how people do it”

Pseudocode: insert

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
void insert(int val) {
  if(size==arr.length-1)
    resize();
  size++;
  i=percolateUp(size,val);
  arr[i] = val;
}
```

```
int percolateUp(int hole,
                int val) {
  while(hole > 1 &&
        val < arr[hole/2])
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```



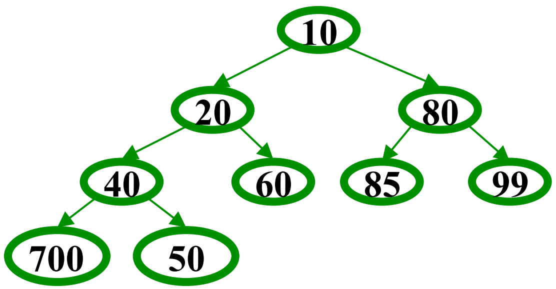
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	10	20	80	40	60	85	99	700	50				

Pseudocode: deleteMin

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
int deleteMin() {
  if(isEmpty()) throw…
  ans = arr[1];
  hole = percolateDown
           (1,arr[size]);
  arr[hole] = arr[size];
  size--;
  return ans;
}
```

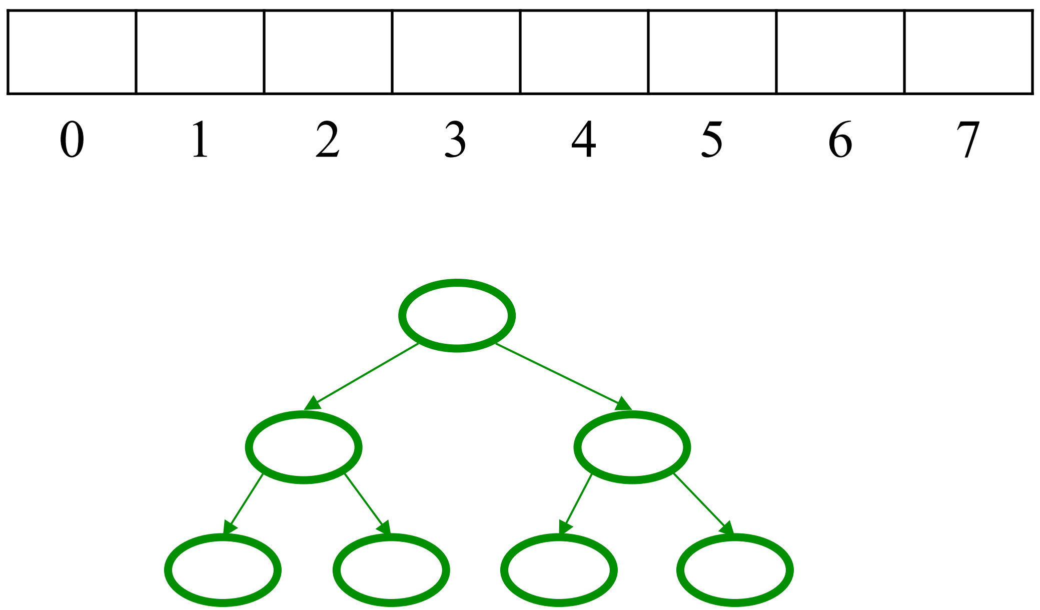
```
int percolateDown(int hole,
                  int val) {
 while(2*hole <= size) {
  left  = 2*hole;
  right = left + 1;
  if(right > size ||
     arr[left] < arr[right])
    target = left;
  else
    target = right;
  if(arr[target] < val) {
    arr[hole] = arr[target];
    hole = target;
  } else
      break;
 }
 return hole;
}
```



	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

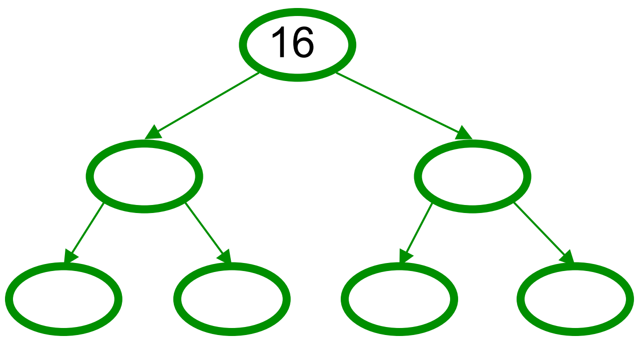
1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin



Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

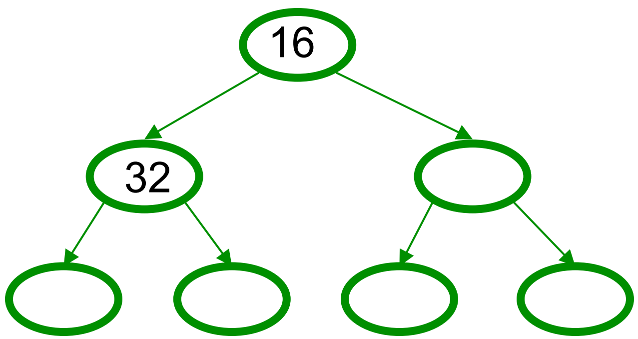
	16						
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

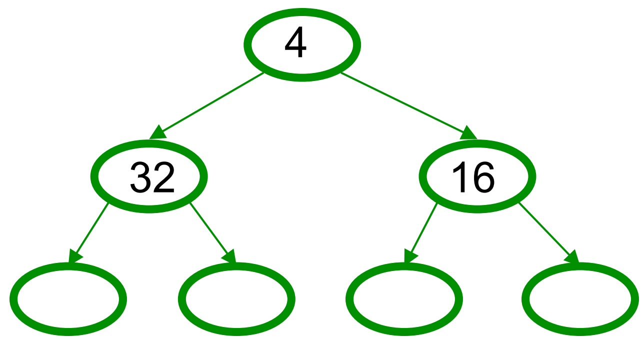
	16	**32**					
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

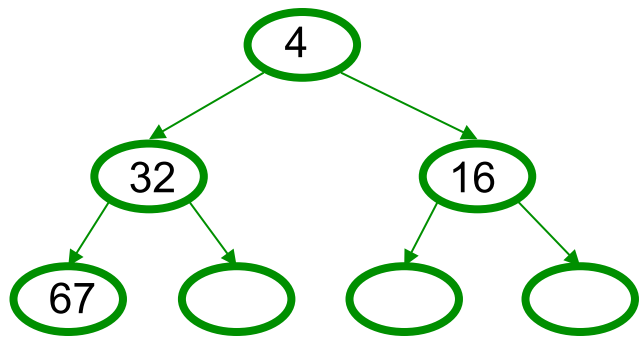
	4	**32**	**16**				
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

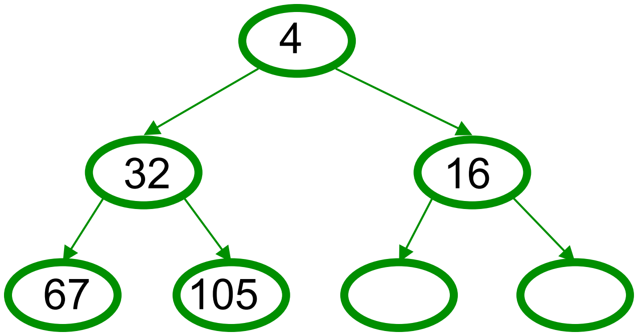
	4	**32**	**16**	**67**			
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

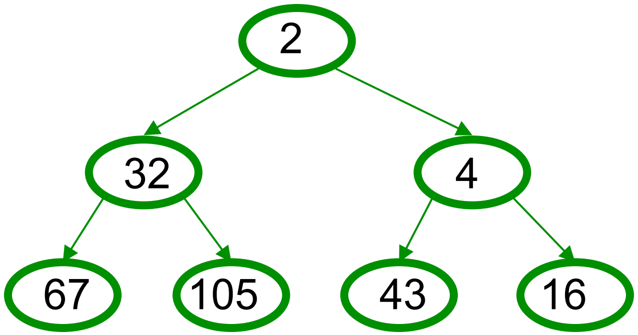
	4	**32**	**16**	**67**	**105**		
0	1	2	3	4	5	6	7



Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

	2	32	4	67	105	43	16
0	1	2	3	4	5	6	7



Other operations

- **`decreaseKey`**: given pointer to object in priority queue (e.g., its array index), lower its priority value by *p*
  - Change priority and percolate up

- **`increaseKey`**: given pointer to object in priority queue (e.g., its array index), raise its priority value by *p*
  - Change priority and percolate down

- **`remove`**: given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - **`decreaseKey`** with $p = \infty$, then **`deleteMin`**

Running time for all these operations?

Build Heap

- Suppose you have *n* items to put in a new (empty) priority queue
  - Call this operation **`buildHeap`**

- *n* **`insert`**s works
  - Only choice if ADT doesn't provide **`buildHeap`** explicitly
  - $O(n \log n)$

- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an $O(n)$ algorithm called Floyd's Method
  - Common issue in ADT design: how many specialized operations

Floyd's Method

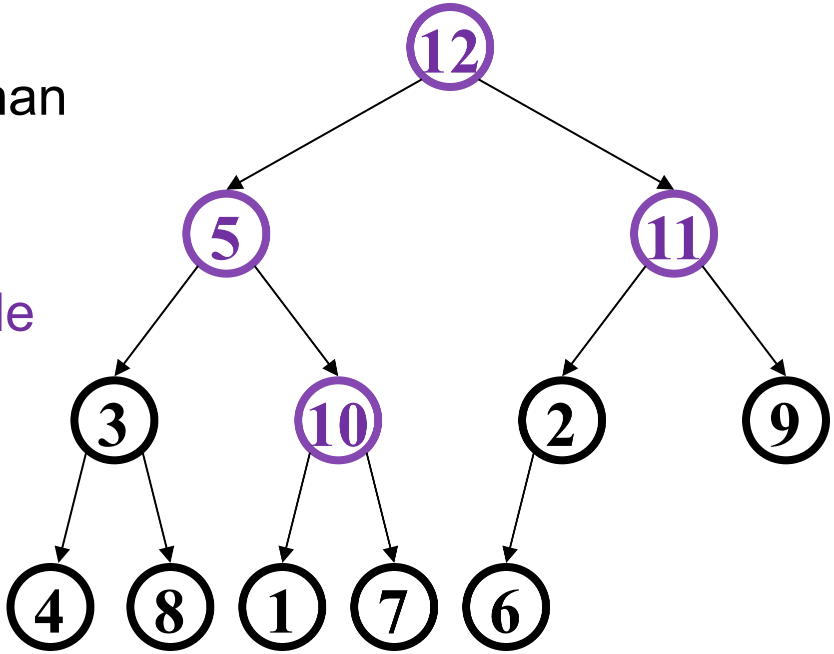
1. Use *n* items to make any complete tree you want
   – That is, put them in array indices 1,…,*n*

2. Treat it as a heap and fix the heap-order property
   – Bottom-up: leaves are already in heap order, work up toward the root one level at a time

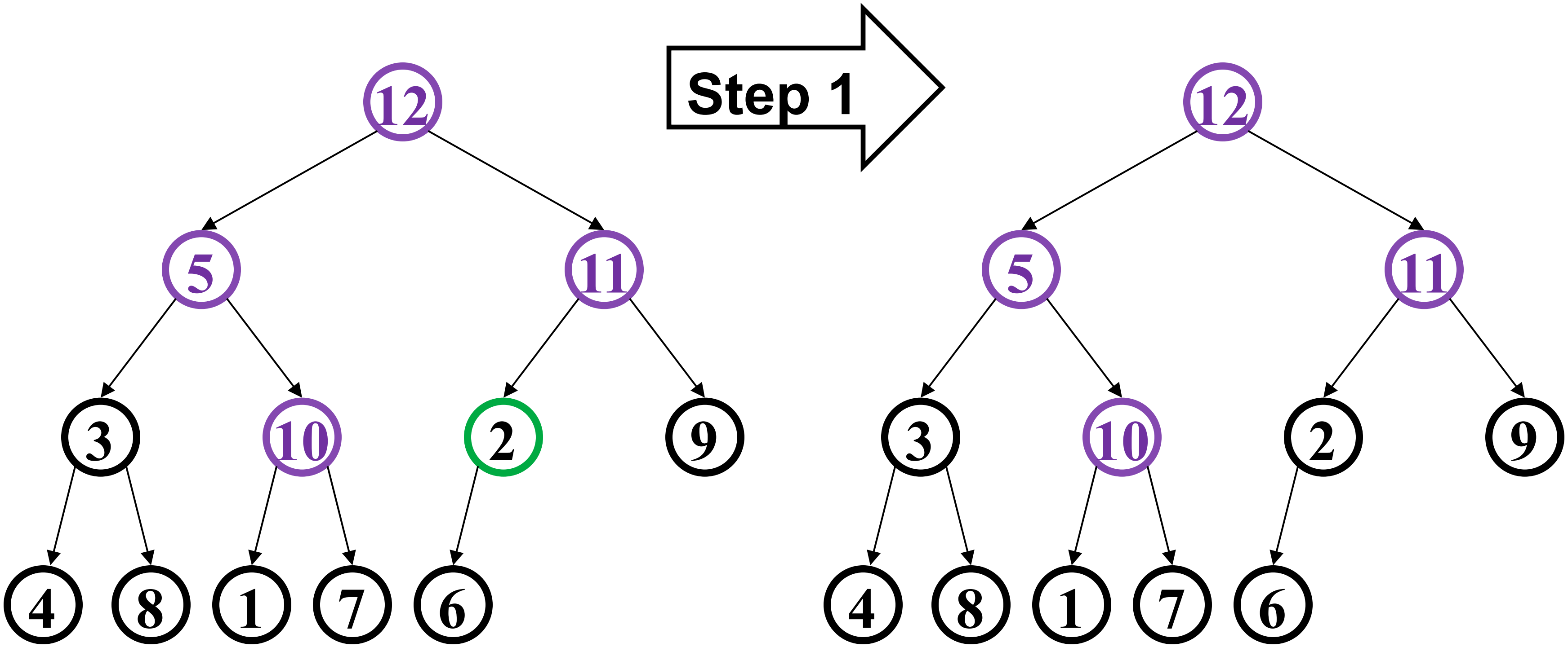
```
void buildHeap() {
   for(i = size/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i,val);
     arr[hole] = val;
   }
}
```

Example

- In tree form for readability
  - Purple for node not less than descendants
    - heap-order problem
  - Notice no leaves are purple
  - Check/fix each non-leaf bottom-up (6 steps here)

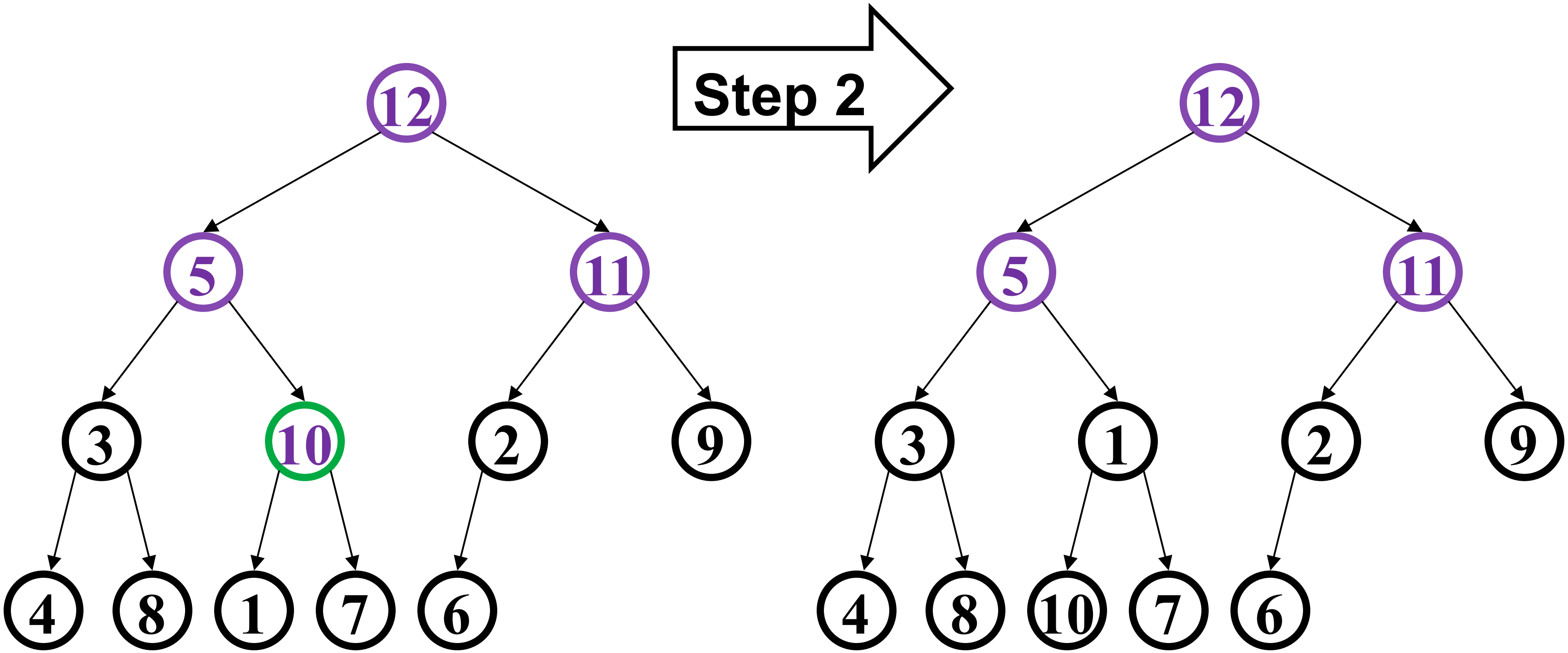


Example



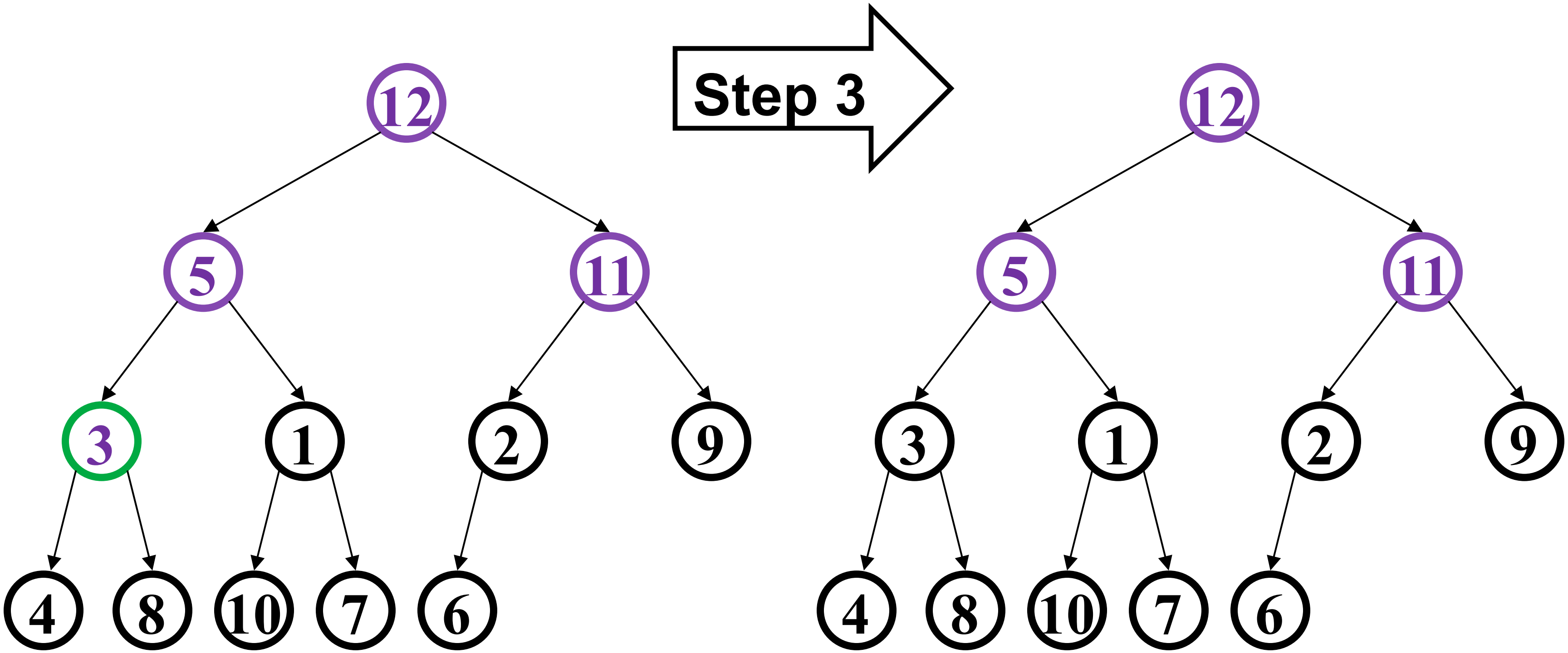
- Happens to already be less than children (er, child)

Example



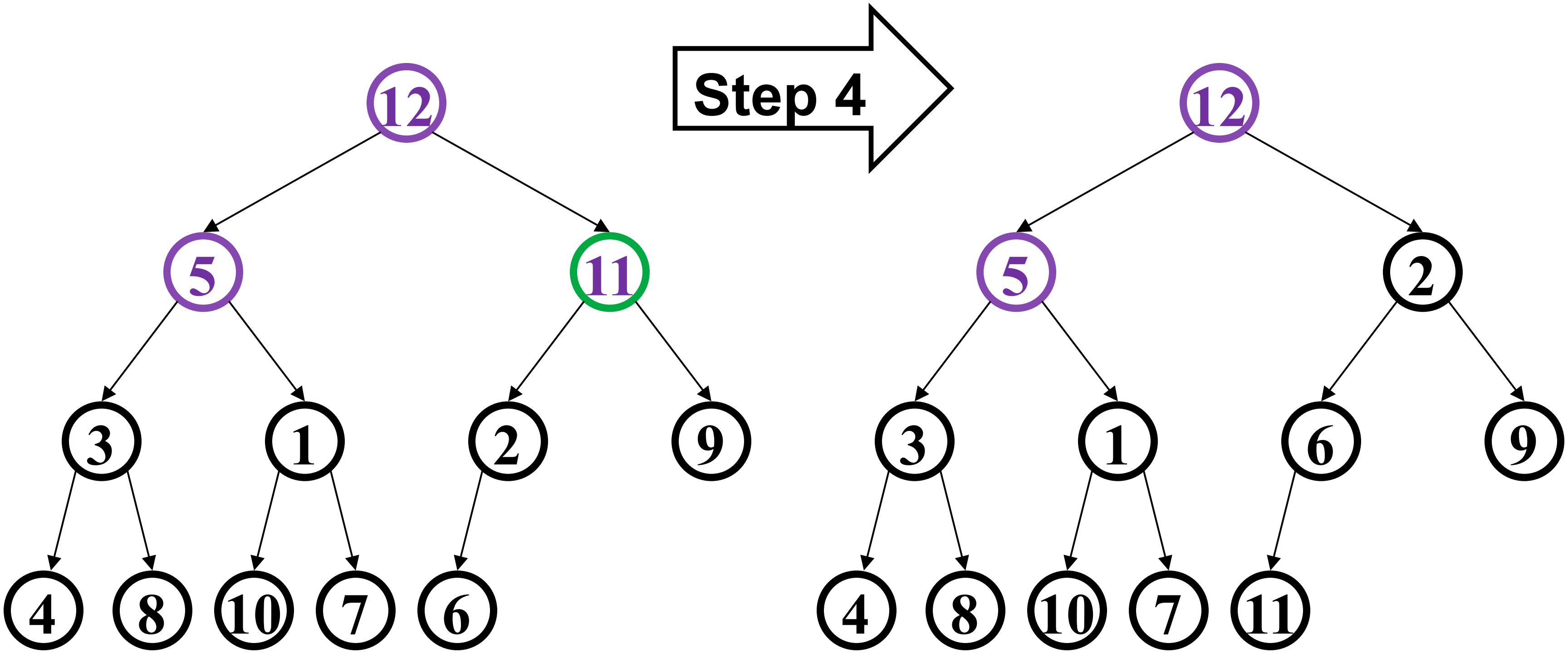
- Percolate down (notice that moves 1 up)

Example



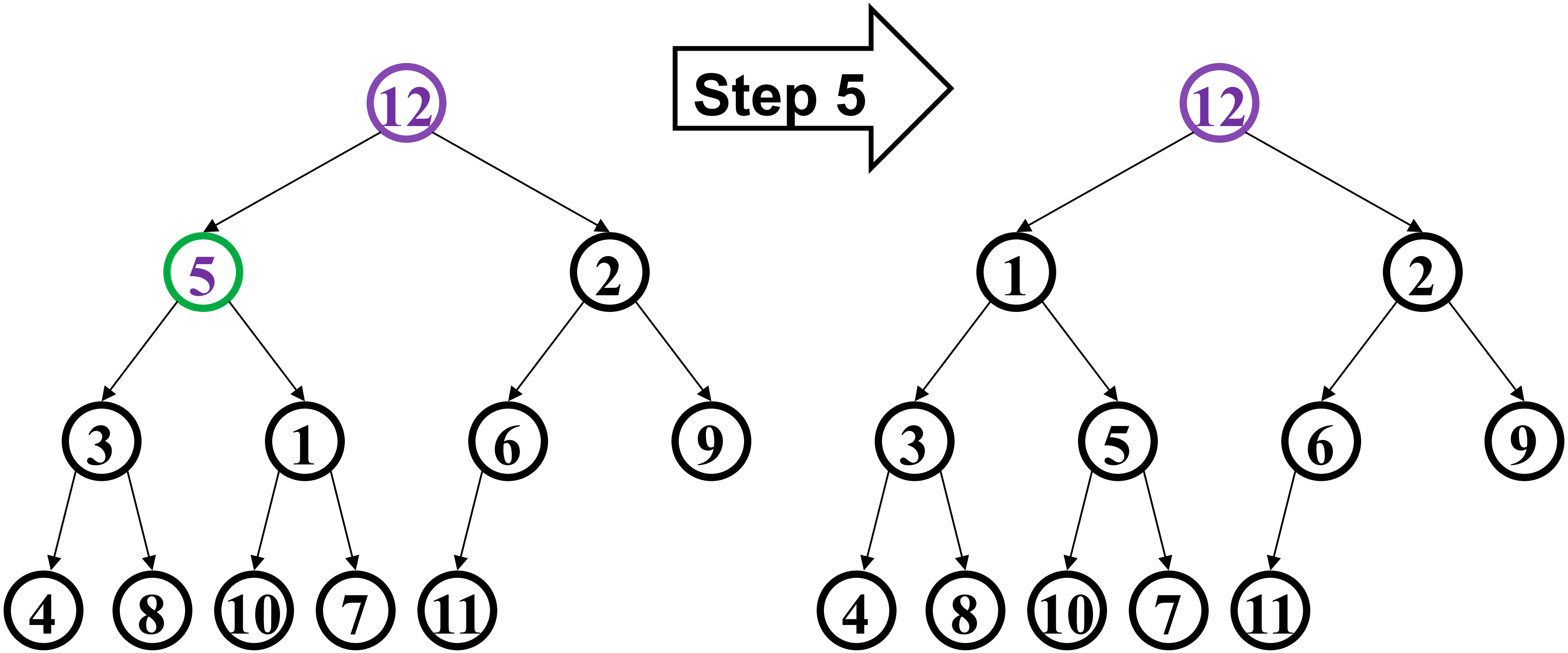
- Another nothing-to-do step

Example

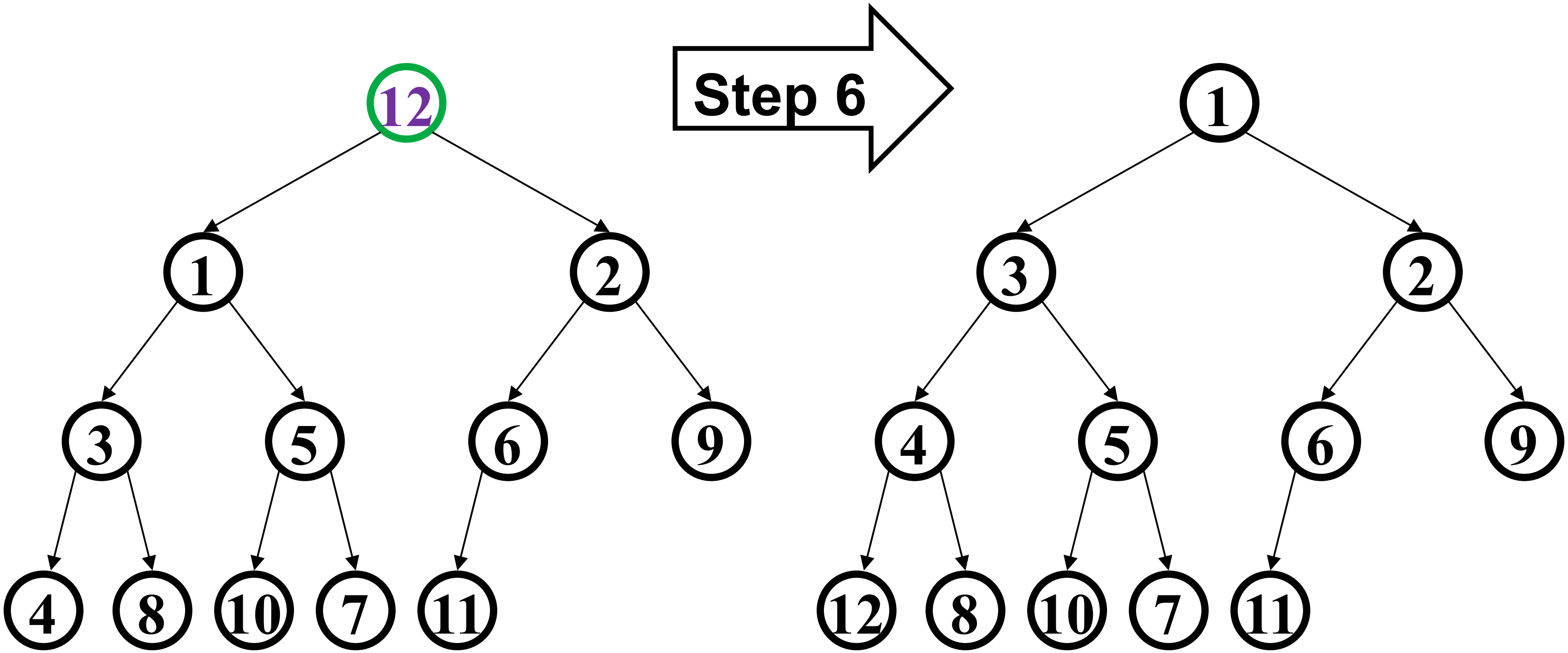


- Percolate down as necessary (steps 4a and 4b)

Example



Example



But is it right?

- "Seems to work"
  - Let's *prove* it restores the heap property (correctness)
  - Then let's *prove* its running time (efficiency)

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Correctness

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Loop Invariant: For all **j>i**, **arr[j]** is less than its children

- True initially: If **j > size/2**, then **j** is a leaf
  - Otherwise its left child would be at position > **size**
- True after one more iteration: loop body and **percolateDown** make **arr[i]** less than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children

Efficiency

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Easy argument: **`buildHeap`** is $O(n$ **`log`** $n)$ where n is **`size`**

- **`size/2`** loop iterations
- Each iteration does one **`percolateDown`**, each is $O($**`log`** $n)$

This is correct, but there is a more precise ("tighter") analysis of the algorithm…

Efficiency

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Better argument: **`buildHeap`** is *O*(*n*) where *n* is **`size`**

- **`size/2`** total loop iterations: *O*(*n*)
- 1/2 the loop iterations percolate at most 1 step
- 1/4 the loop iterations percolate at most 2 steps
- 1/8 the loop iterations percolate at most 3 steps
- …
- ((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + …) < 2 (page 4 of Weiss)
  - So at most **`2(size/2)`** *total* percolate steps: *O*(*n*)

Lessons from `buildHeap`

- Without **`buildHeap`**, our ADT already let clients implement their own in $O(n \log n)$ worst case
  - Worst case is inserting lower priority values later
- By providing a specialized operation internal to the data structure (with access to the internal data), we can do $O(n)$ worst case
  - Intuition: Most data is near a leaf, so better to percolate down
- Can analyze this algorithm for:
  - Correctness:
    - Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First analysis easily proved it was $O(n \log n)$
    - Tighter analysis shows same algorithm is $O(n)$

Other branching factors

- *d*-heaps: have *d* children instead of 2
  - Makes heaps shallower, useful for heaps too big for memory (or cache)
- Homework: Implement a 3-heap
  - Just have three children instead of 2
  - Still use an array with all positions from 1…heap-size used

Index	Children Indices
1	2,3,4
2	5,6,7
3	8,9,10
4	11,12,13
5	14,15,16
…	…

What we are skipping

- **`merge:`** given two priority queues, make one priority queue
  - How might you merge binary heaps:
    - If one heap is much smaller than the other?
    - If both are about the same size?
  - Different pointer-based data structures for priority queues support logarithmic time **`merge`** operation (impossible with binary heaps)
    - Leftist heaps, skew heaps, binomial queues
    - Worse constant factors
    - Trade-offs!