CSE373: Data Structures & Algorithms

Lecture 9: Priority Queues

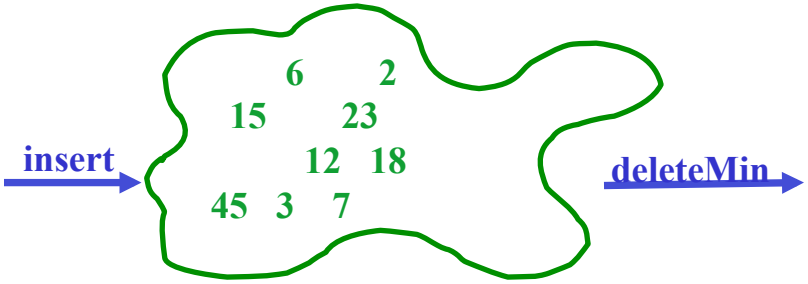
Aaron Bauer

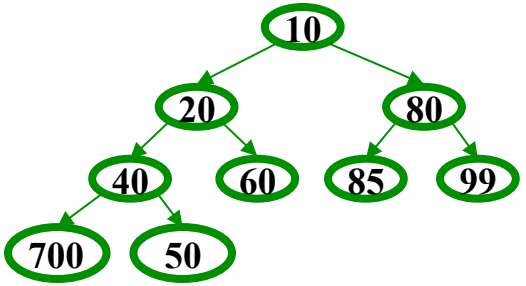
Winter 2014

Midterm

- On Wednesday, in class
- Closed book
- Closed note
- Closed electronic devices
- Closed classmate
- Covers everything up through priority queues and binary heaps
  - does not include AVL tree delete
  - does not include proof AVL tree has logarithmic height

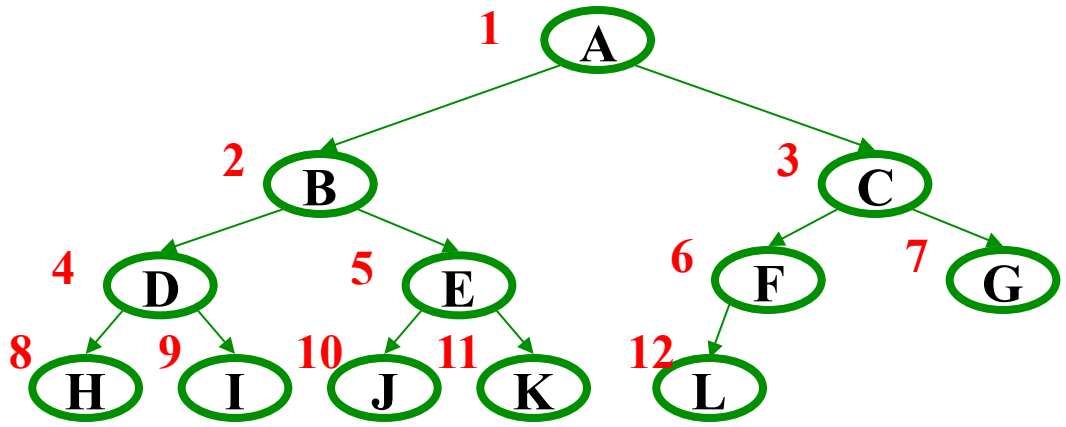
Review





- 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:

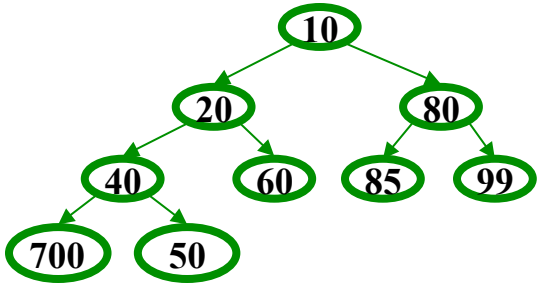
	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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;
}
```

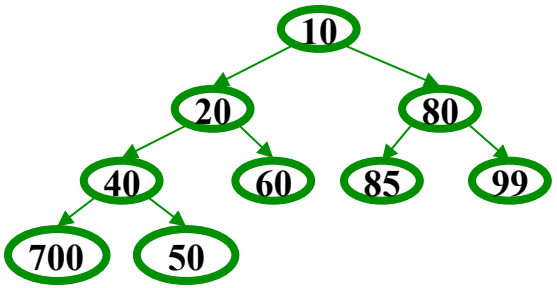


	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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;
}
```

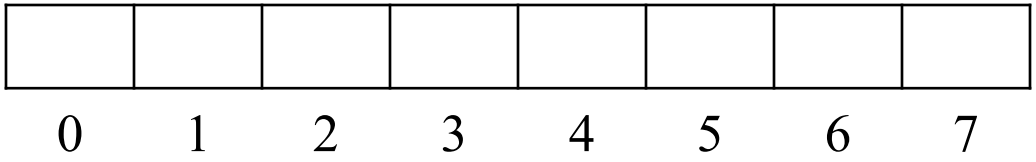


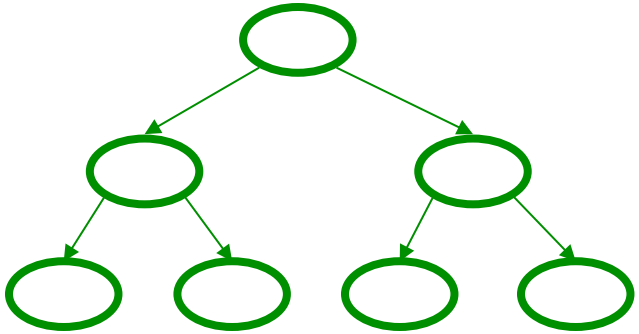

```
int percolateDown(int hole,
                  int val) {
 while(2*hole <= size) {
  left  = 2*hole;
  right = left + 1;
  if(right > size ||
     arr[left] < arr[right])
    target = left;
  else
    target = right;
  if(arr[target] < val) {
    arr[hole] = arr[target];
    hole = target;
  } else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

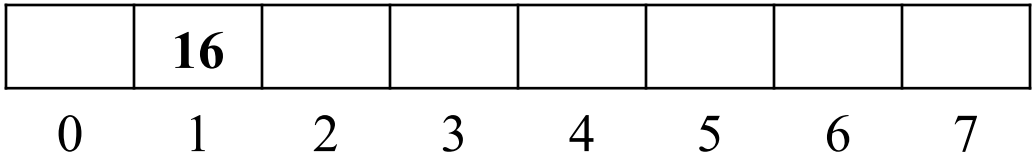
1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

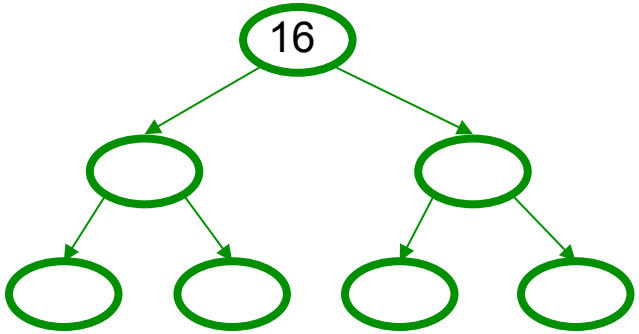




Example

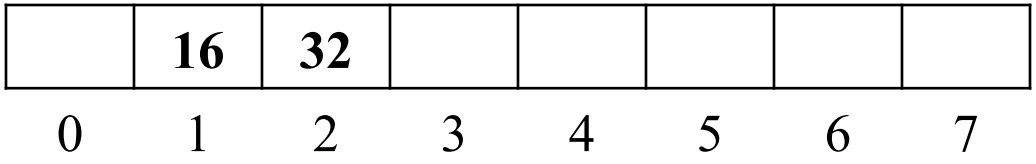
1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

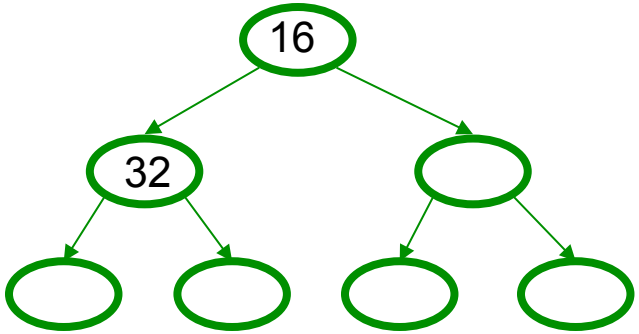




Example

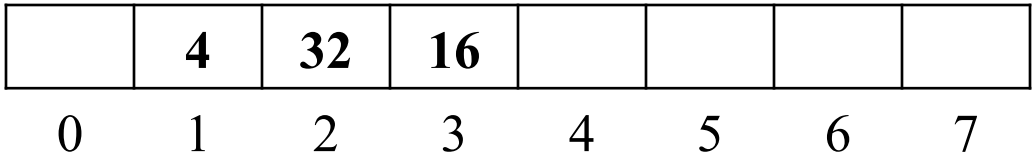
1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

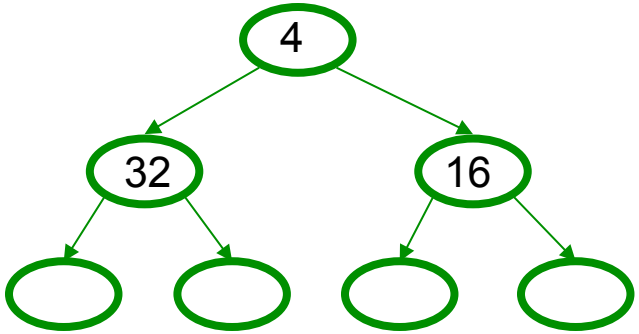




Example

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

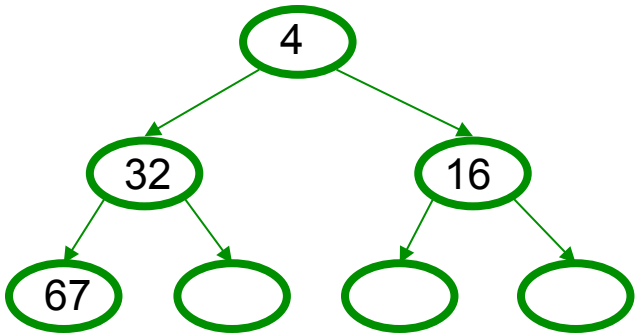




Example

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

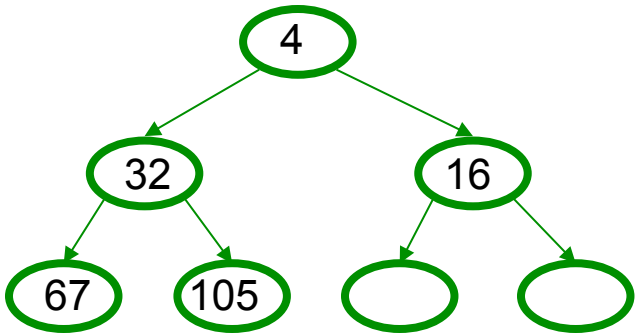
	4	32	16	67			
0	1	2	3	4	5	6	7



Example

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

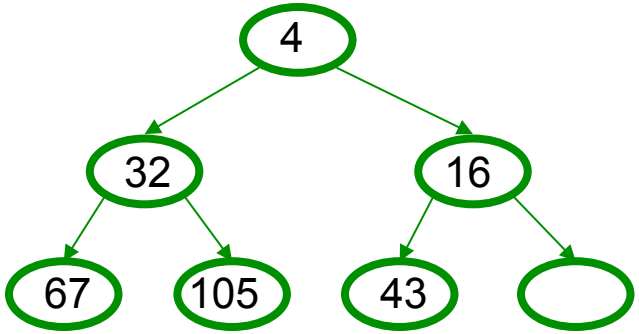
	4	32	16	67	105		
0	1	2	3	4	5	6	7



Example

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

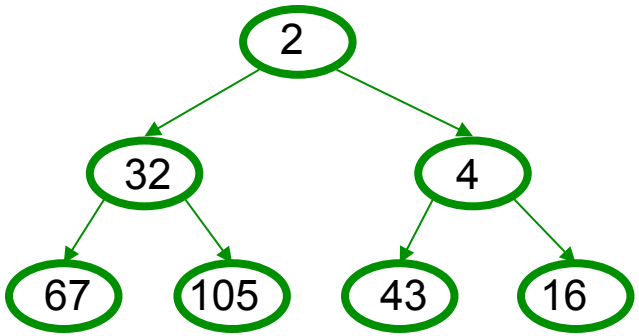
	4	32	16	67	105	43	
0	1	2	3	4	5	6	7



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

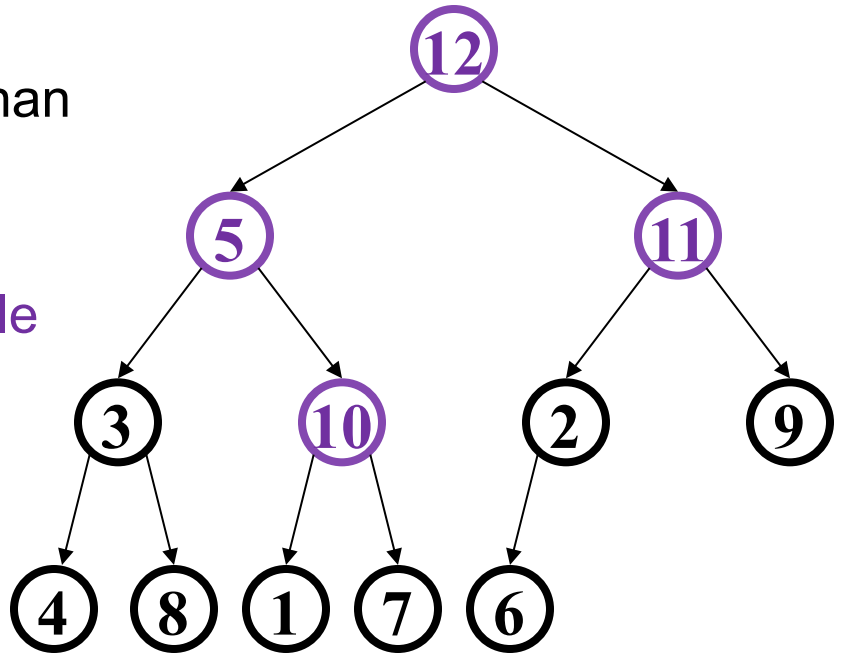
1. Use *n* items to make any complete tree you want
   - That is, put them in array indices 1,…,*n*

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

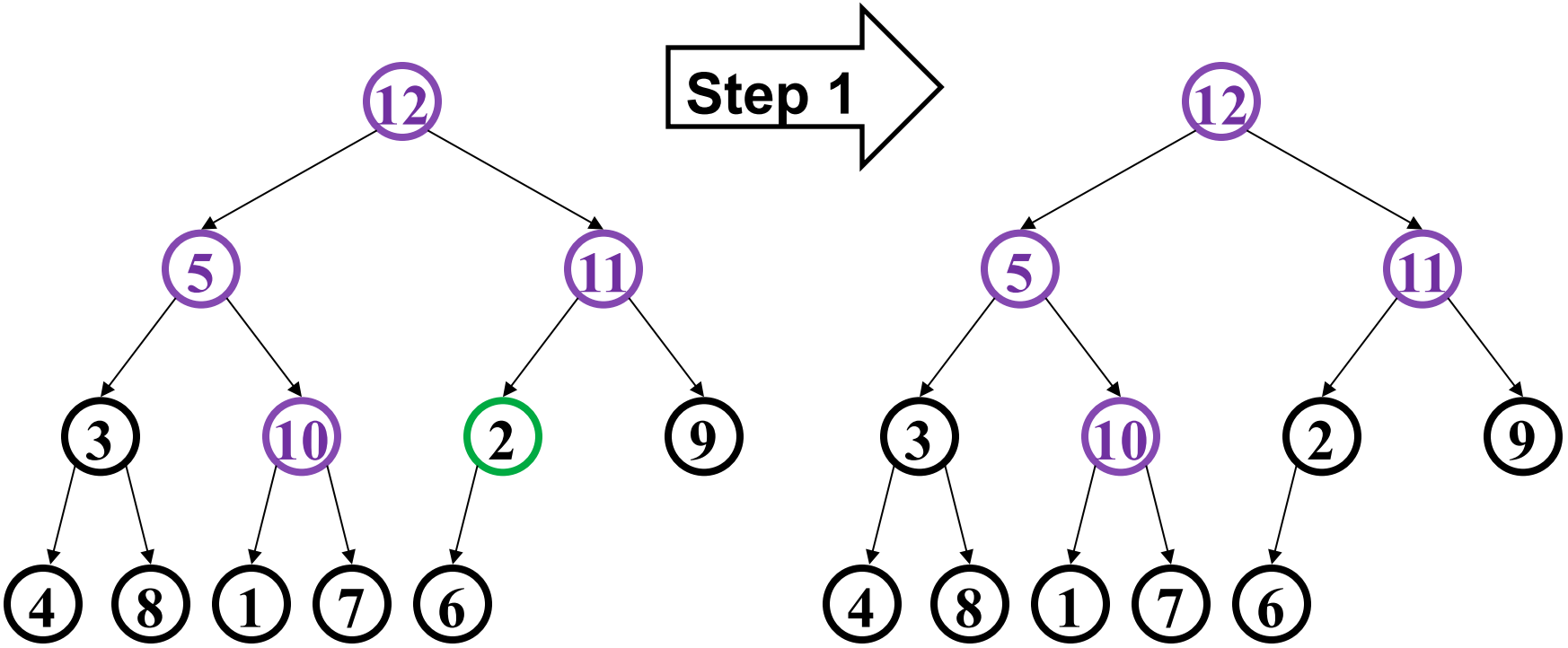
```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

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)

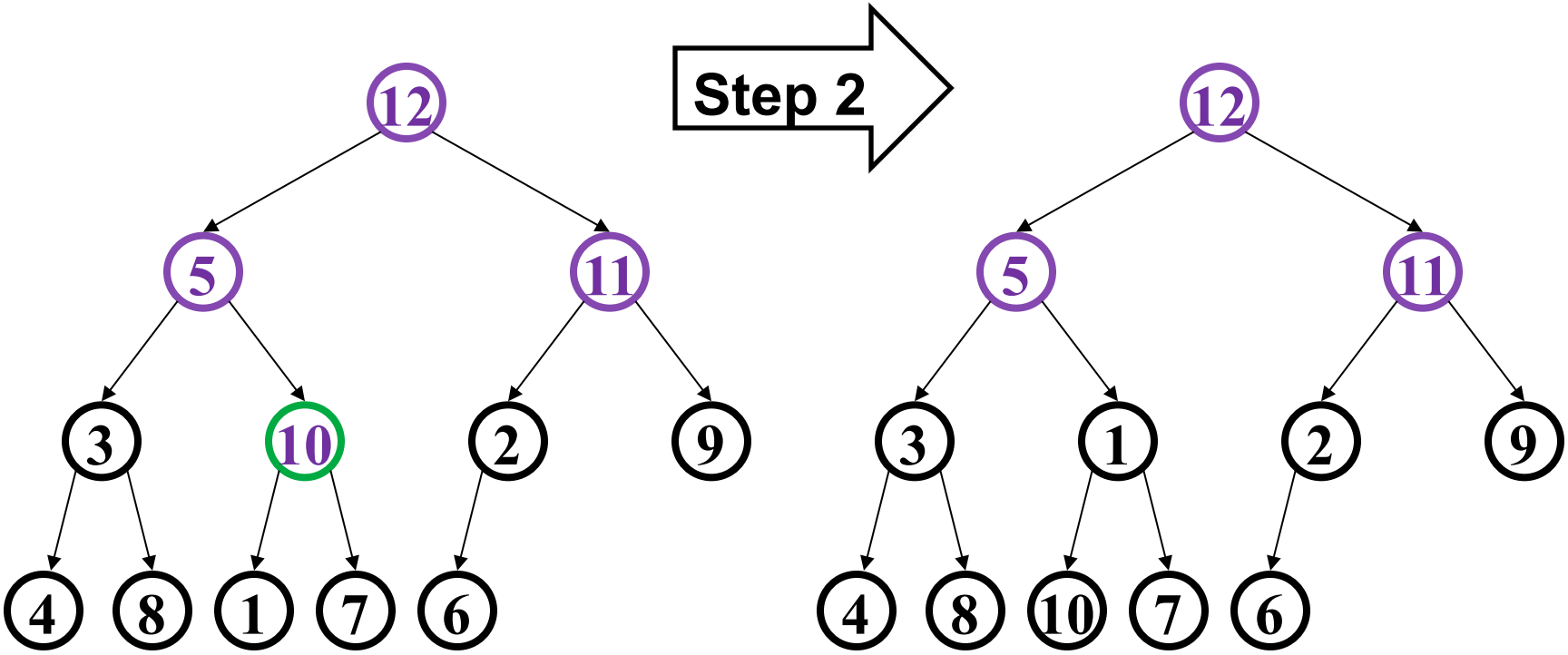


Example



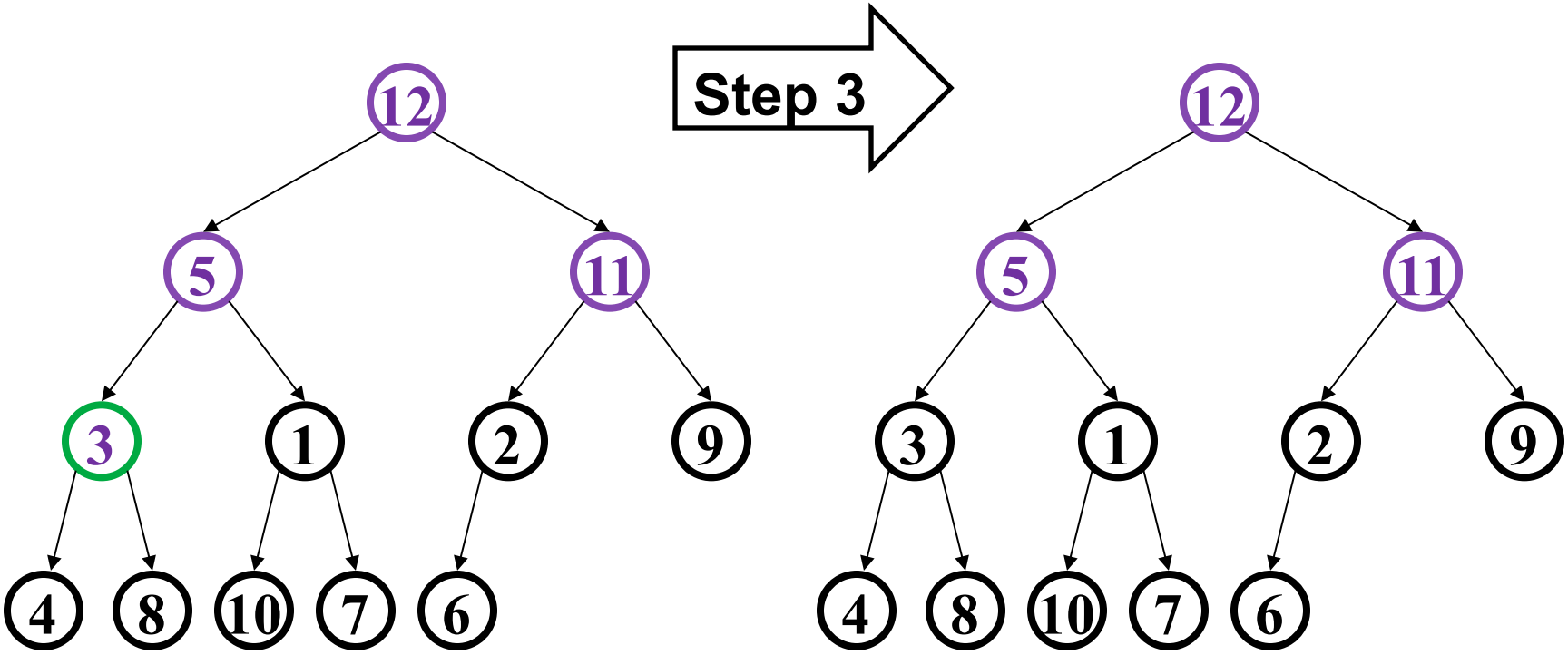
- Happens to already be less than children (er, child)

Example



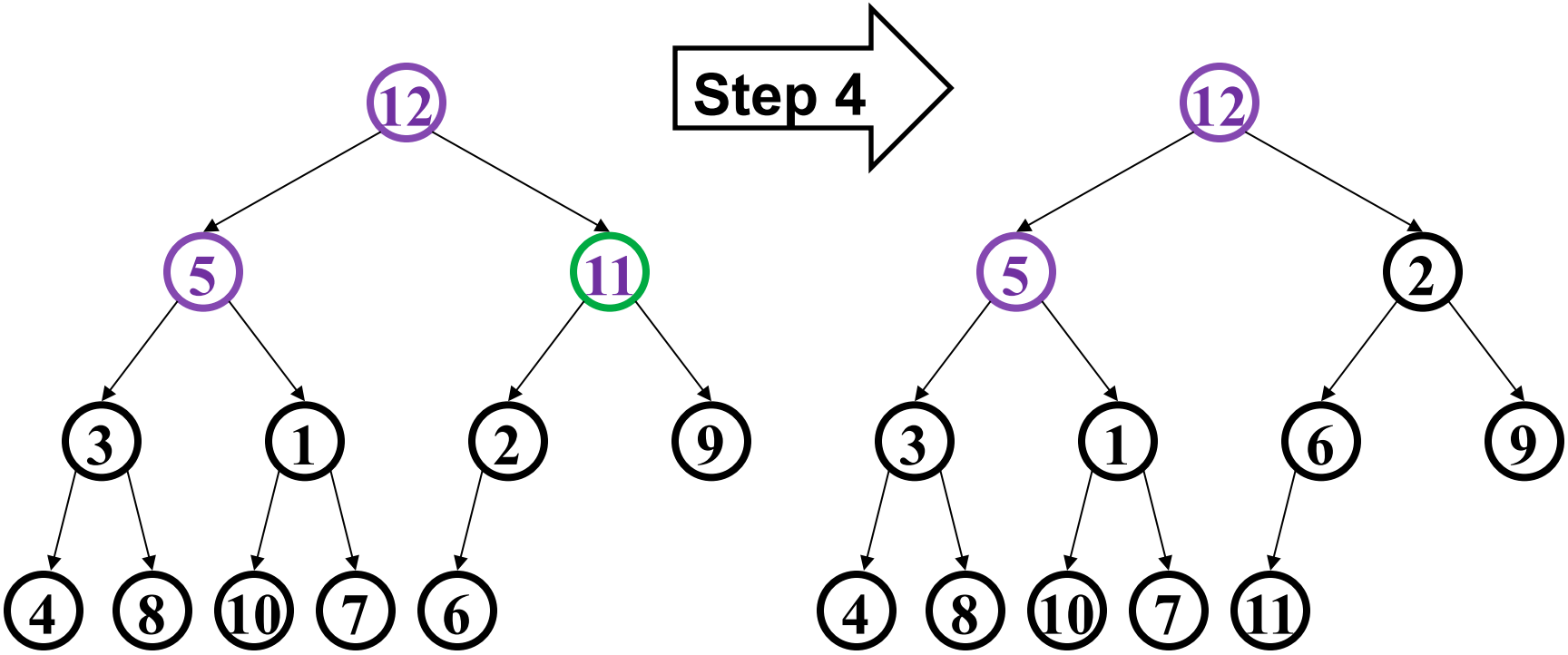
- Percolate down (notice that moves 1 up)

Example



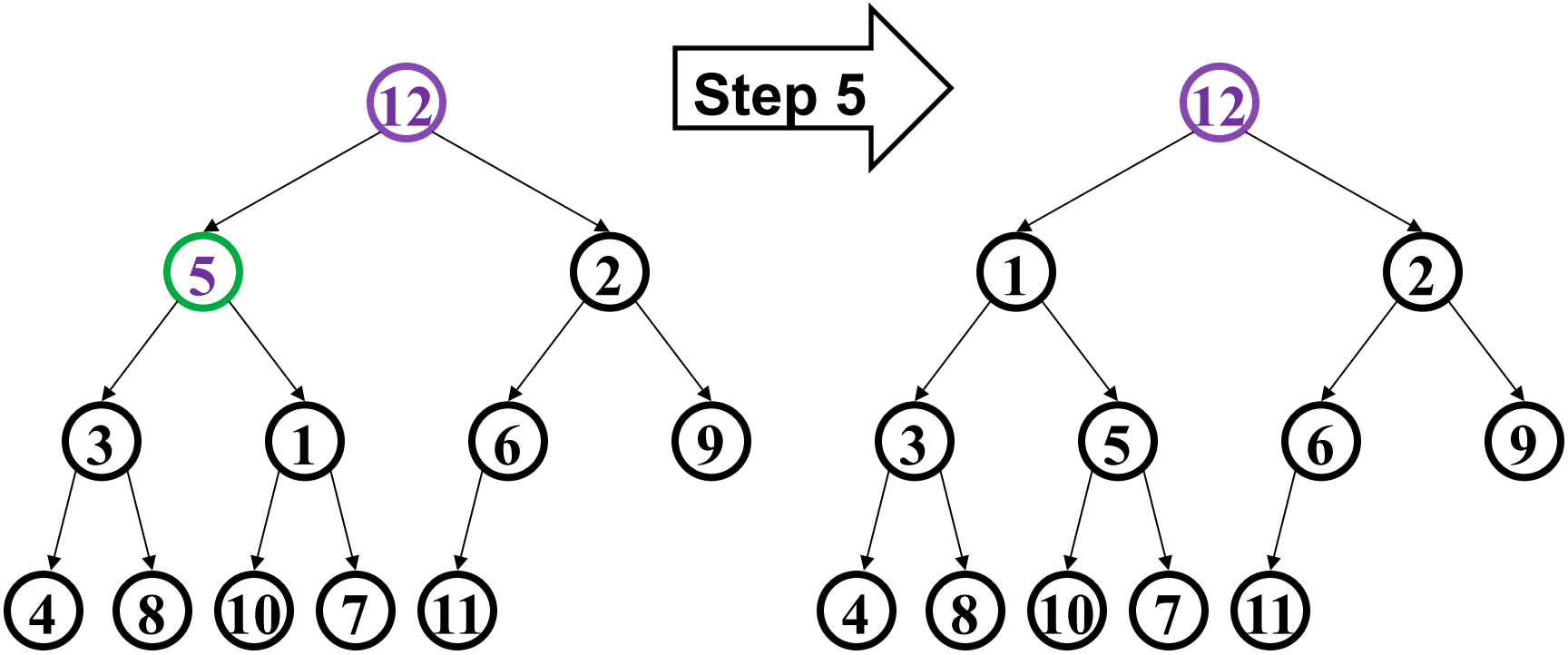
- Another nothing-to-do step

Example

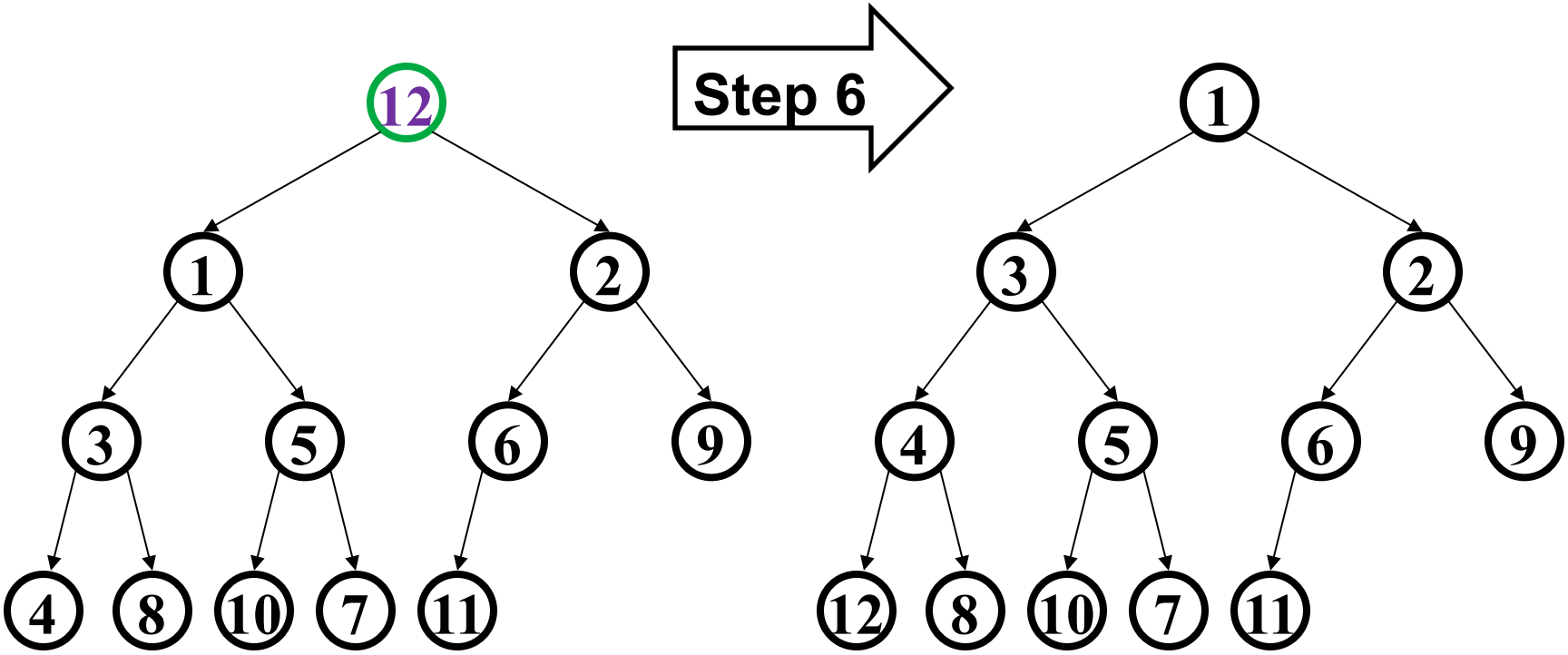


- Percolate down as necessary (steps 4a and 4b)

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!

Amortized

- Recall our plain-old stack implemented as an array that doubles its size if it runs out of room
  - How can we claim **`push`** is $O(1)$ time if resizing is $O(n)$ time?
  - We *can't*, but we *can* claim it's an $O(1)$ amortized operation

- What does amortized mean?
- When are amortized bounds good enough?
- How can we prove an amortized bound?

Will just do two simple examples

- Text has more sophisticated examples and proof techniques
- *Idea* of how amortized describes average cost is essential

Amortized Complexity

If a sequence of **M** operations takes $O($**M f(n)**$)$ time,
we say the amortized runtime is $O($**f(n)**$)$

Amortized bound: worst-case guarantee over sequences of operations

- Example: If any **n** operations take $O($**n**$)$, then amortized $O($**1**$)$
- Example: If any **n** operations take $O($**n**$^3)$, then amortized $O($**n**$^2)$

- The worst case time per operation can be larger than **f(n)**
  - As long as the worst case is *always* "rare enough" in *any* sequence of operations

Amortized guarantee ensures the average time per operation for any sequence is $O($**f(n)**$)$

"Building Up Credit"

- Can think of preceding "cheap" operations as building up "credit" that can be used to "pay for" later "expensive" operations

- Because any sequence of operations must be under the bound, enough "cheap" operations must come *first*
  - Else a prefix of the sequence, which is also a sequence, would violate the bound

Example #1: Resizing stack

A stack implemented with an array where we double the size of the array if it becomes full

Claim: Any sequence of **`push`**/**`pop`**/**`isEmpty`** is amortized $O(1)$

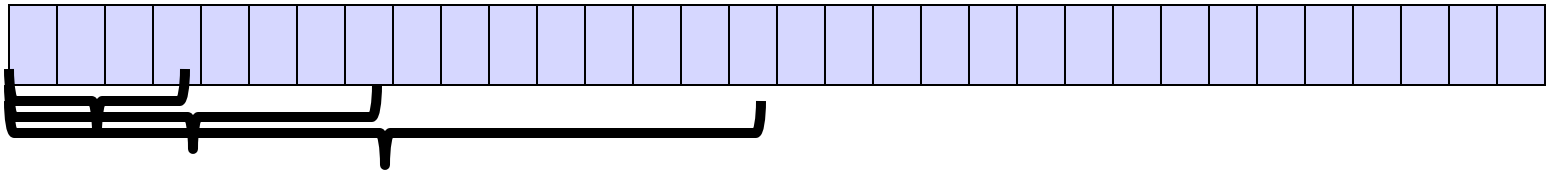
Need to show any sequence of **`M`** operations takes time $O(M)$

- Recall the non-resizing work is $O(M)$ (i.e., $M * O(1)$)
- The resizing work is proportional to the total number of element copies we do for the resizing
- So it suffices to show that:

  After **`M`** operations, we have done **`< 2M`** total element copies

  (So average number of copies per operation is bounded by a constant)

Amount of copying



After **`M`** operations, we have done **`< 2M`** total element copies

Let **`n`** be the size of the array after **`M`** operations

- Then we have done a total of:

  `n/2 + n/4 + n/8 +` … **`INITIAL_SIZE < n`**

  element copies
- Because we must have done at least enough **`push`** operations to cause resizing up to size **`n`**:

  `M ≥ n/2`
- So

  `2M ≥ n >` *number of element copies*

Other approaches

- If array grows by a constant amount (say 1000),
  operations are not amortized $O(\mathtt{1})$
  - After $O(\mathtt{M})$ operations, you may have done $\Theta(\mathtt{M}^2)$ copies

- If array shrinks when 1/2 empty,
  operations are not amortized $O(\mathtt{1})$
  - Terrible case: **`pop`** once and shrink, **`push`** once and grow, **`pop`** once and shrink, …

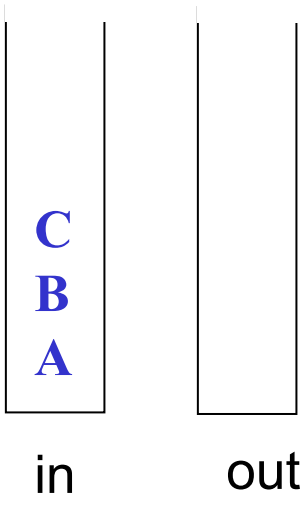
- If array shrinks when 3/4 empty,
  it is amortized $O(\mathtt{1})$
  - Proof is more complicated, but basic idea remains: by the time an expensive operation occurs, many cheap ones occurred

Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

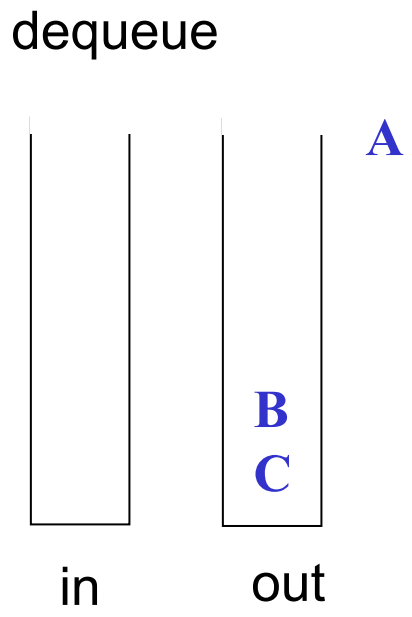
enqueue: A, B, C



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

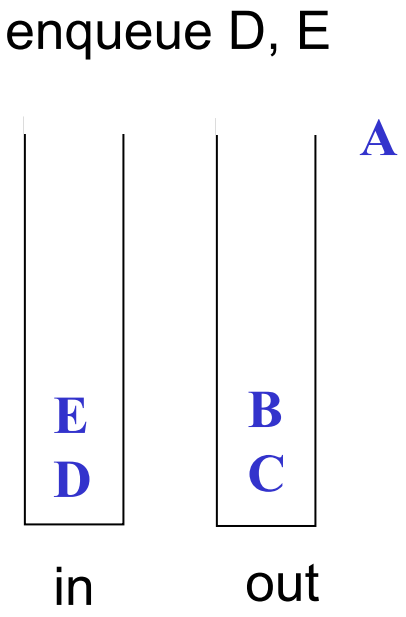
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

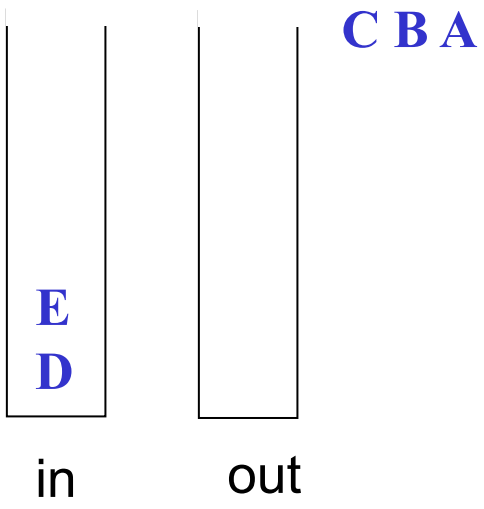


Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

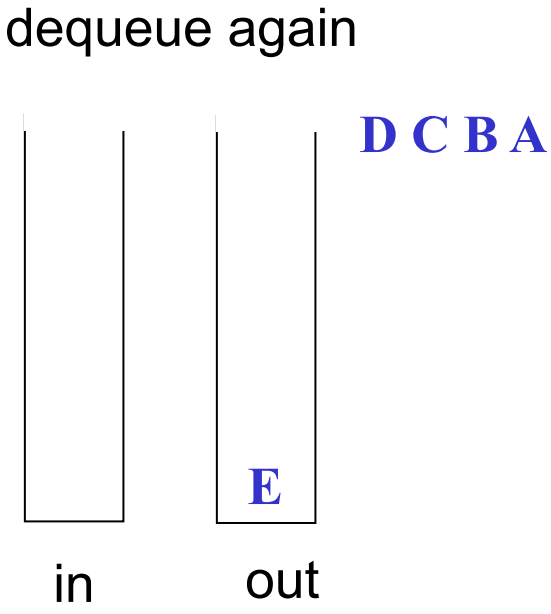
dequeue twice



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Correctness and usefulness

- If **`x`** is enqueued before **`y`**, then **`x`** will be popped from **in** later than **`y`** and therefore popped from **out** sooner than **`y`**
  - So it is a queue

- Example:
  - Wouldn't it be nice to have a queue of t-shirts to wear instead of a stack (like in your dresser)?
  - So have two stacks
    - *in*: stack of t-shirts go after you wash them
    - *out*: stack of t-shirts to wear
    - if *out* is empty, reverse *in* into *out*

Analysis

- **`dequeue`** is not *O*(**`1`**) worst-case because **out** might be empty and **in** may have lots of items

- But if the stack operations are (amortized) *O*(**`1`**), then any sequence of queue operations is amortized *O*(**`1`**)

  - The total amount of work done per element is 1 **`push`** onto **in**, 1 **`pop`** off of **in**, 1 **`push`** onto **out**, 1 **`pop`** off of **out**

  - When you reverse **`n`** elements, there were **`n`** earlier *O*(**`1`**) **`enqueue`** operations to average with

Amortized useful?

- When the average per operation is all we care about (i.e., sum over all operations), amortized is perfectly fine
  - This is the usual situation

- If we need every operation to finish quickly (e.g., in a web server), amortized bounds may be too weak

- While amortized analysis is about averages, we are averaging cost-per-operation on worst-case input
  - Contrast: Average-case analysis is about averages across possible inputs. Example: if all initial permutations of an array are equally likely, then quicksort is $O(\mathtt{n\ log\ n})$ on average even though on some inputs it is $O(\mathtt{n}^2)$)

Not always so simple

- Proofs for amortized bounds can be much more complicated
- Example: Splay trees are dictionaries with amortized $O($`log n`$)$ operations
  - No extra height field like AVL trees
  - See Chapter 4.5 if curious
- For more complicated examples, the proofs need much more sophisticated invariants and "potential functions" to describe how earlier cheap operations build up "energy" or "money" to "pay for" later expensive operations
  - See Chapter 11 if curious
- But complicated *proofs* have nothing to do with the code!