buildHeap;
Algorithm Analysis II: Amortization

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- Tell us how Quiz #1 went!
  - **rn: 👉 👉 pollev.com/cse332 👈 👈**
  - Later/more details: cse332-staff@cs or anonymous feedback
- Exercises 2-4 due on Monday @ midnight
- Project 1 due on Tuesday @ midnight
  - If you’re struggling with your partnership, please reach out!
  - If you’re struggling with the project, schedule 1:1 time!
  - Don’t forget to check your pipelines for failures (we do!)

Learning Objectives

- Implement Floyd’s algorithm for building heaps and reason about its runtime
- Be able to formally prove a big-O runtime bound
- Intuit, but not necessarily prove, amortized analysis
- Explain the different axes of case complexity, and how it relates to complexity bounds

Lecture Outline

- Heaps, cont.
  - **O(1) average case insert**
  - Floyd's `buildHeap` Algorithm
  - Farewell to Heaps …

- Algorithm Analysis II
  - Proof example
  - Amortized bounds
  - Where We've Come / Where We're Going

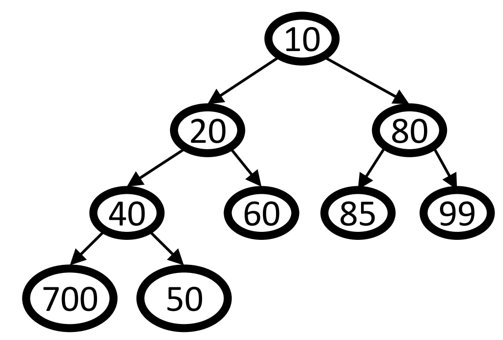
O(1) average-case add()?! (1 of 2)

- Yes, `add`'s worst case is O(log n)
  - It all depends on the order the items are inserted
  - What is the worst case order?

- Empirical studies of randomly ordered inputs shows:
  - Average 2.607 comparisons per insert (# of percolation passes)
  - An element usually moves up 1.607 levels

- If we define "average" as a *single operation with random input, occurring after a sequence of similarly randomized operations* :
  - `add`'s *average case* is O(1)
  - `deleteMin`'s average case is still O(log n)
    - Moving a leaf to the root usually requires re-percolating that priority back to the bottom

O(1) average-case add()?! (2 of 2)



- In a complete binary tree, each row has 2x nodes of its parent row
  - Bottom level has ~1/2 of all nodes
  - Second to bottom has ~1/4 of all nodes
  - …
- Intuition:
  - When inserting a *random* priority, likely not to have highest nor lowest value; somewhere in middle
  - Given a random distribution of priorities in the heap:
    - Bottom level should have the upper ½ of priorities
    - Second to bottom, next ¼
    - …
  - Expect to only percolate up 1-2 levels

Lecture Outline

- Heaps, cont.
  - O(1) average case insert
  - **Floyd's `buildHeap` Algorithm**
  - Farewell to Heaps …

- Algorithm Analysis II
  - Proof example
  - Amortized bounds
  - Where We've Come / Where We're Going

One Final Operation: buildHeap

- `buildHeap`() takes an array of size N and applies the heap-ordering principle to it

- Naïve implementation:
  - Start with an empty array (representing an empty binary heap)
  - Call `add`() N times
  - Runtime: ??

- Can we do better?
  - If we only have **`add`** and **`deleteMin`** operations, **NO**
  - There is a faster way -- O(n) -- but requires the ADT to have a specialized **`buildHeap`** operation
  - **Is it convenient? Efficient? Simple?**

Floyd's `buildHeap` Method

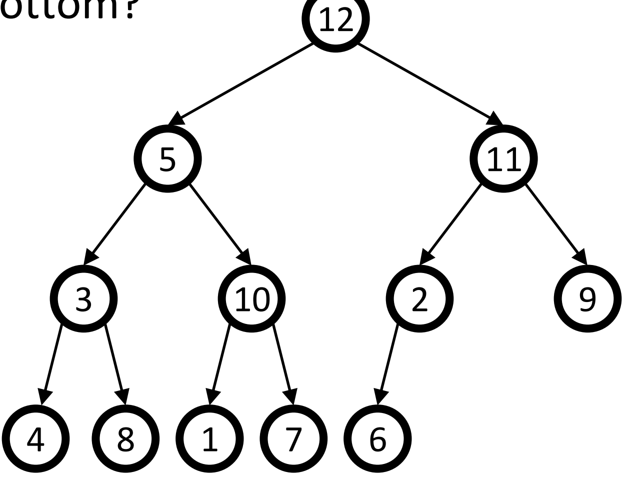
- Recall our general strategy for working with the heap:
  - Preserve structure property
  - (Break and) Restore heap ordering property

- Floyd's buildHeap:
  - Create a complete tree by putting the n items in an array
    - *Structure property!*
  - Treat the array as a binary heap and fix the heap-order property
    - *Order property!*
  - Exactly how we do this is where we gain efficiency

Reminder: a priority queue contains *priorities* and *values*; an *item* or *data* refers to the (priority, value) pair

Thinking about `buildHeap`

- Say we start with this array: [12,5,11,3,10,2,9,4,8,1,7,6]
- Where should we start? Top vs bottom?
- To "fix" the ordering can we use:
  - percolateUp?
  - percolateDown?



Floyd's `buildHeap` Method

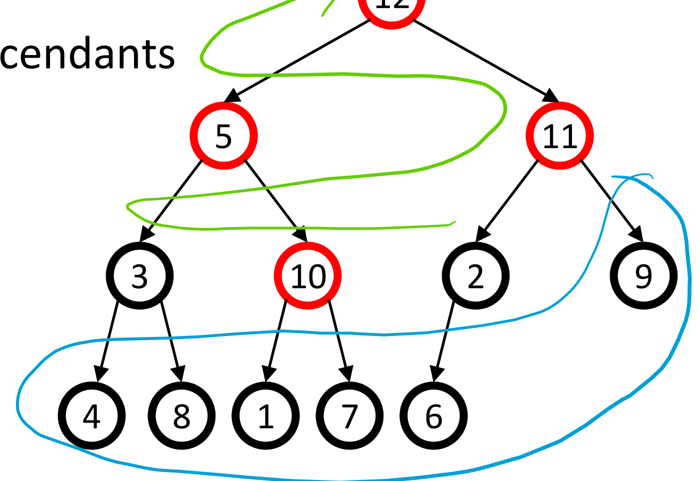
- Bottom-up:
  - Leaves are already in heap order
  - Work up toward the root one level at a time

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

Note: Exercises and P1 start counting from 0

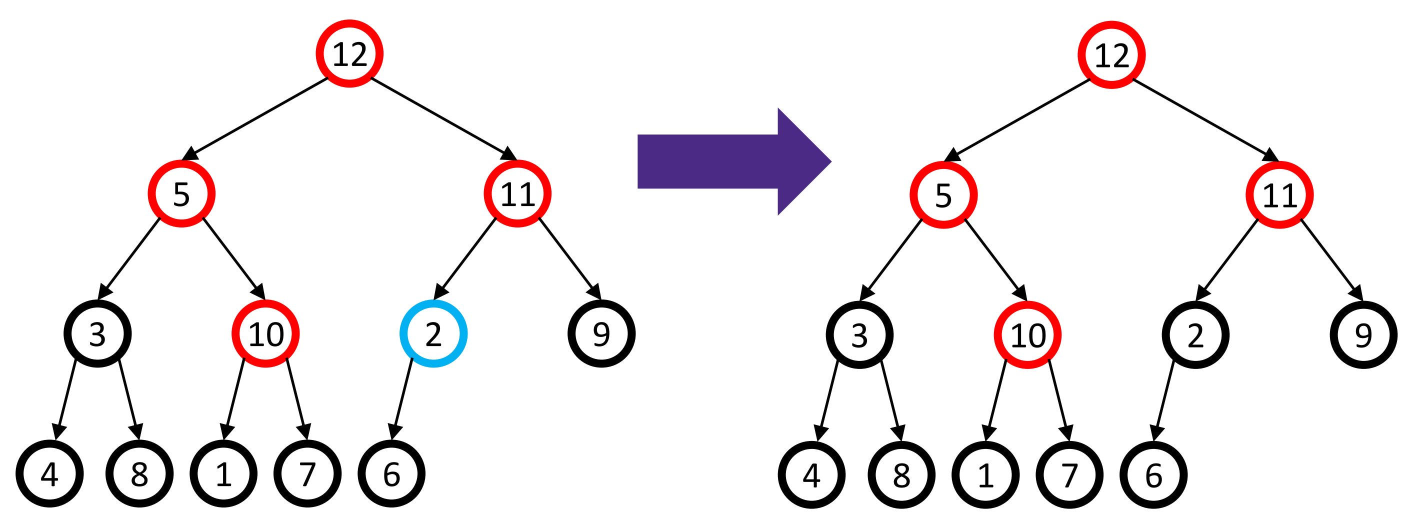
buildHeap Example

- Say we start with this array: [12,5,11,3,10,2,9,4,8,1,7,6]
  - In tree form for readability
- Red for node not less than descendants
  - Ie, heap-order problem
  - Notice no leaves are red!



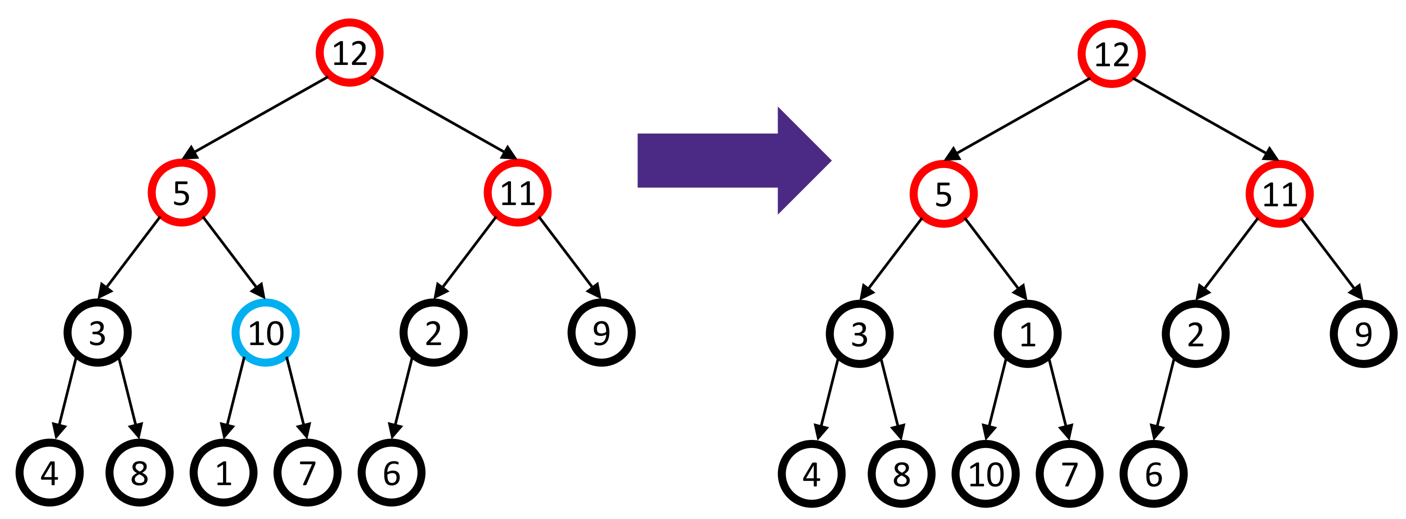
`buildHeap` Example: Step 1

- Happens to already be less than child



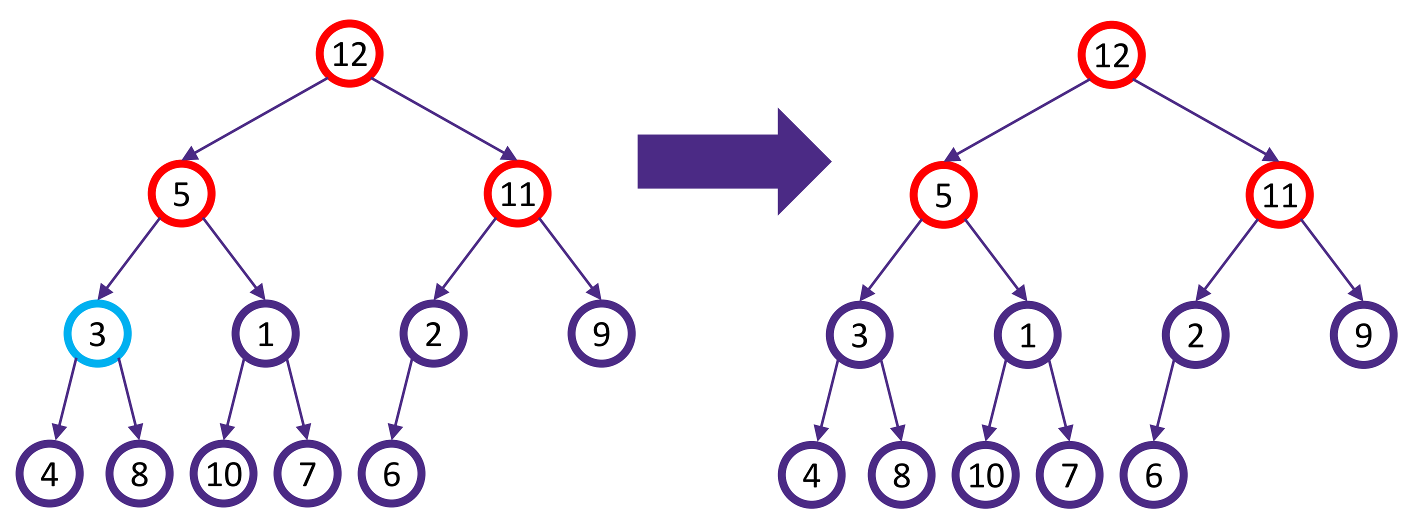
`buildHeap` Example: Step 2

- Percolate down (notice that moves 1 up)



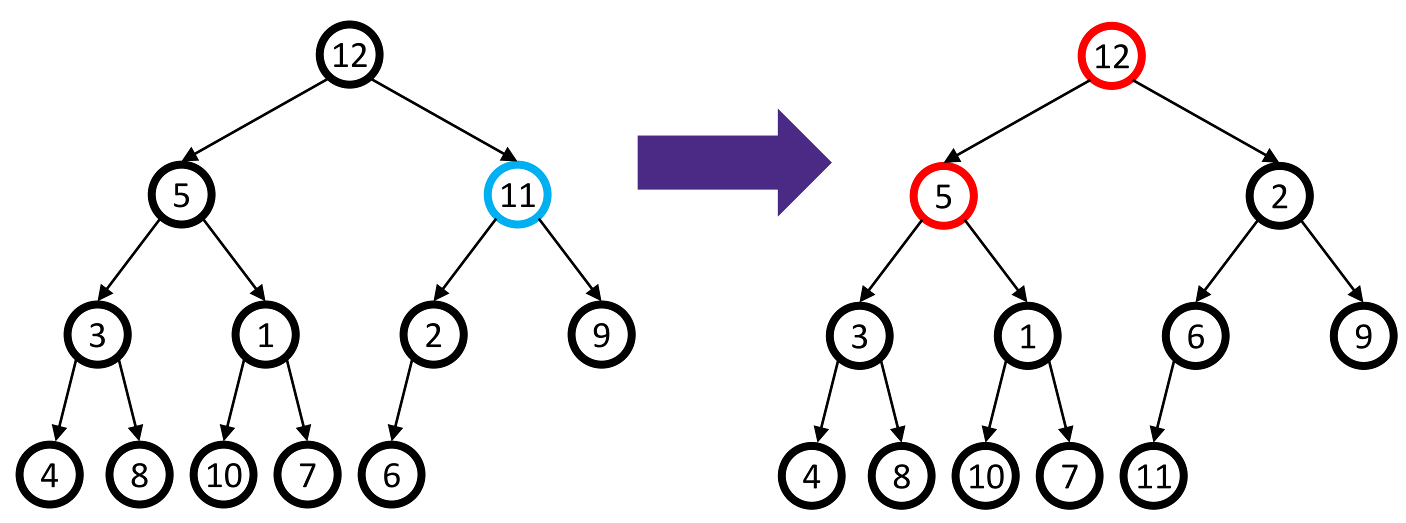
`buildHeap` Example: Step 3

- Another nothing-to-do step



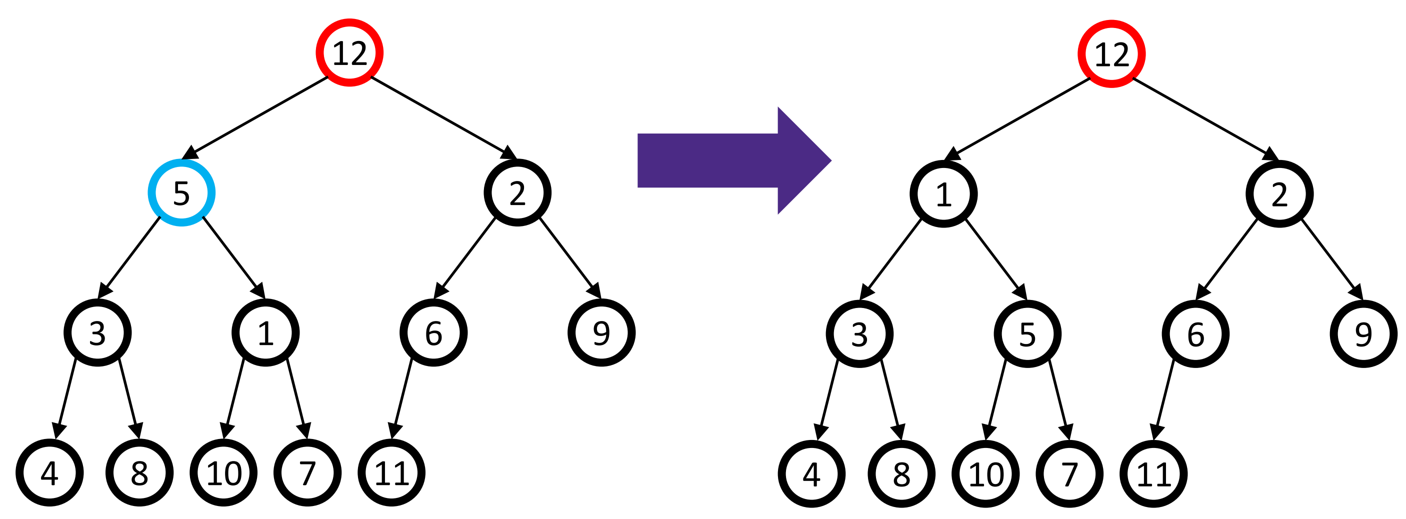
`buildHeap` Example: Step 4

- Percolate down as necessary



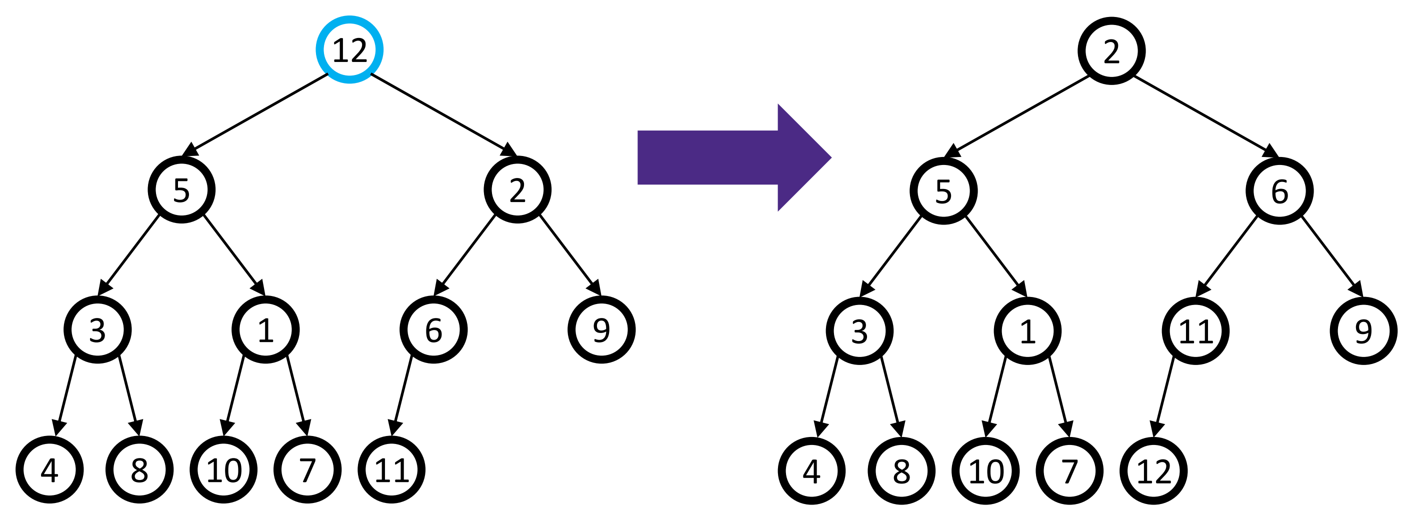
`buildHeap` Example: Step 5

- Again, percolate down as necessary



`buildHeap` Example: Step 6

- Lastly, percolate down as necessary



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(arr) {
   n = arr.length
   for(i = n/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i, val);
     arr[hole] = val;
  }
}
```

Note: Exercises and P1 start counting from 0

Floyd's `buildHeap`: Correctness

- ***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 iteration: loop body and `percolateDown` make arr[i] less than children without breaking the property for any descendants

Base Case

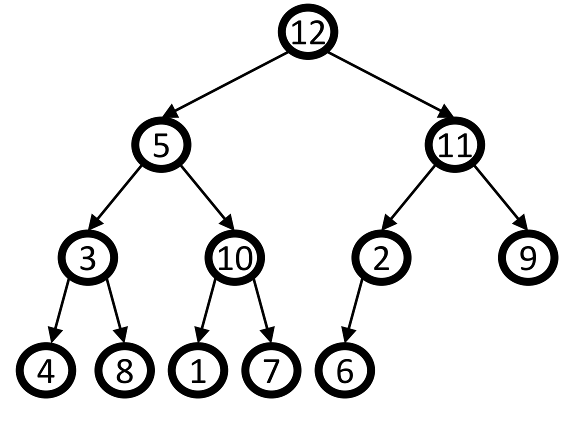
Inductive Step

- Therefore, after loop terminates, ***all nodes are less than their children***

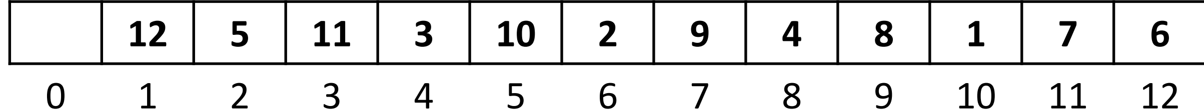
```
void buildHeap(arr) {
    n = arr.length
    for(i = n/2; i>0; i--) {
      val  = arr[i];
      hole = percolateDown(i, val);
      arr[hole] = val;
    }
}
```

Note: Exercises and P1 start counting from 0

Floyd's `buildHeap`: Correctness Example




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



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

Note: Exercises and P1 start counting from 0

Floyd's `buildHeap`: Efficiency (1 of 2)

- Easy argument: `buildHeap` is O(n log n) where n is array size
  - n/2 loop iterations
  - Each iteration does one `percolateDown`, which are O(log n) each
  - So Floyd's `buildHeap` is n/2 * log n = O(n log n)

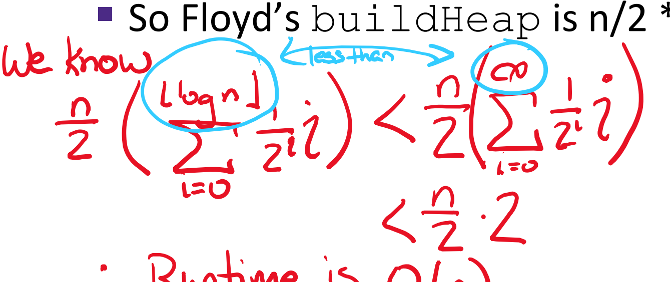
- This is correct, but there is a more precise ("tighter") analysis

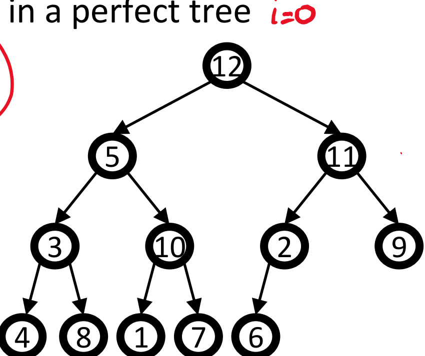
```
void buildHeap() {
   for(i = size/2; i>0; i--) {
    val  = arr[i];
     hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Note: Exercises and P1 start counting from 0

Floyd's `buildHeap`: Efficiency (2 of 2)

- Better argument: `buildHeap` is O(n) where n is array size
  - n/2 total loop iterations: O(n)
    - 1/2 of the loop iterations percolate at most **1 step**
    - 1/4 of the loop iterations percolate at most **2 steps**
    - 1/8 of the loop iterations percolate at most **3 steps**
    - … etc …
  - But we know (1 + (**1**/2) + (**2**/4) + (**3**/8) + …) = 2
    - See page 4 of Weiss
    - Also see Weiss 6.3.4, sum of heights of nodes in a perfect tree
  - So Floyd's `buildHeap` is n/2 * 2 = O(n)





Lessons from `buildHeap`

- Without `buildHeap`, our ADT let clients implement their own in θ(n log n) worst case
  - Worst case is inserting lower priorities later
- By providing a specialized operation (with access to the internal data structure), we can do O(n) worst case
  - Intuition: Most items are 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)
    - A "tighter" analysis shows same algorithm is O(n)

Lecture Outline

- Heaps, cont.
  - O(1) average case insert
  - Floyd's buildHeap Algorithm
  - **Farewell to Heaps …**

- Algorithm Analysis II
  - Proof example
  - Amortized bounds
  - Where We've Come / Where We're Going

What we're skipping (see text if curious)

- ***d-heaps***: have `d` children instead of 2 (Weiss 6.5)
  - Makes heaps shallower, useful for heaps too big for memory
  - How does this affect the asymptotic run-time (for small d's)?

- ***Leftist heaps, skew heaps, binomial queues*** (Weiss 6.6-6.8)
  - Different data structures for priority queues that support a logarithmic time merge operation (impossible with binary heaps)
  - `merge`: given two priority queues, make one priority queue
  - `add` & `deleteMin` defined in terms of merge (!!)

- Aside: How might you merge binary heaps:
  - If one heap is much smaller than the other?
  - If both are about the same size?

Other Operations

- **`decreasePriority`**: given pointer to object in priority queue (e.g., its array index), lower its priority by p
  - Change priority and percolate up
- **`increasePriority`**: given pointer to object in priority queue (e.g., its array index), raise its priority 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 = ∞, then `deleteMin`
- Running time for all these operations?

Priority Queue Summary

- Without **`buildHeap`**, our ADT already let clients implement their own in $\theta(n \log n)$ worst case
  - Worst case is inserting lower priorities later
- By providing a specialized operation internally (with access to the data structure), we can do $O(n)$ worst case
  - Intuition: Most items are 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 $O(n \log n)$
    - A "tighter" analysis shows same algorithm is $O(n)$

Heap-y Birthday, Kenny!

Lots of and for an awesome year ahead!

Lecture Outline

- Heaps, cont.
  - O(1) average case insert
  - Floyd's buildHeap Algorithm
  - Farewell to Heaps …
- Algorithm Analysis II
  - **Proof example**
  - Amortized bounds
  - Where We've Come / Where We're Going

The Proof is in the Practice

$c > 0$
$n_0 \geq 1$ (natural)

- Prove **g(n)** is in **O(f(n))** by picking a **c** large enough to "cover the constant factors" and **n_0** large enough to "cover the lower-order terms"
  - Let **g(n)** = $4n^2 + 3n + 4$, and **f(n)** = n^3

$4n^2 \leq 4n^3$
$3n \leq 3n^3$
$4 \leq 4n^3$

$4n^2 + 3n + 4 \leq 11n^3$ (11 = c)

Let $c = 11$, $n_0 = 1$

Lecture Outline

- Heaps, cont.
  - O(1) average case insert
  - Floyd's buildHeap Algorithm
  - Farewell to Heaps …
- Algorithm Analysis II
  - Proof example
  - **Amortized bounds**
  - Where We've Come / Where We're Going

Linear Search: Best vs Worst Case

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
  for (int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
  return false;
}
```

Best case: **`find(2)`**

Worst case: **`find(126)`**

Complexity Cases

- We started with two cases:
  - **Worst-case complexity**: *maximum* number of steps algorithm takes on "most challenging" input of size N
  - **Best-case complexity**: *minimum* number of steps algorithm takes on "easiest" input of size N

- We punted on one case: **Average-case complexity**
  - Sometimes: relies on distribution of inputs
    - Eg, binary heap's O(1) insert
    - See CSE312 and STAT391
  - Sometimes: uses randomization in the algorithm
    - Will see an example with sorting; also see CSE312

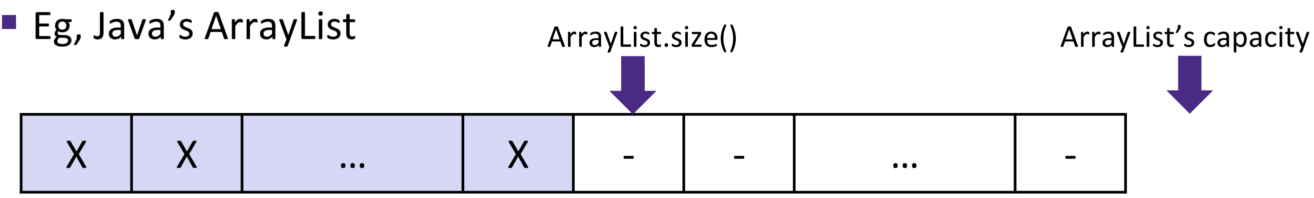
- We've mentioned, but not defined, one *category* of cases:
  - **Amortized-case complexity**

Amortized Analyses = Multiple Executions

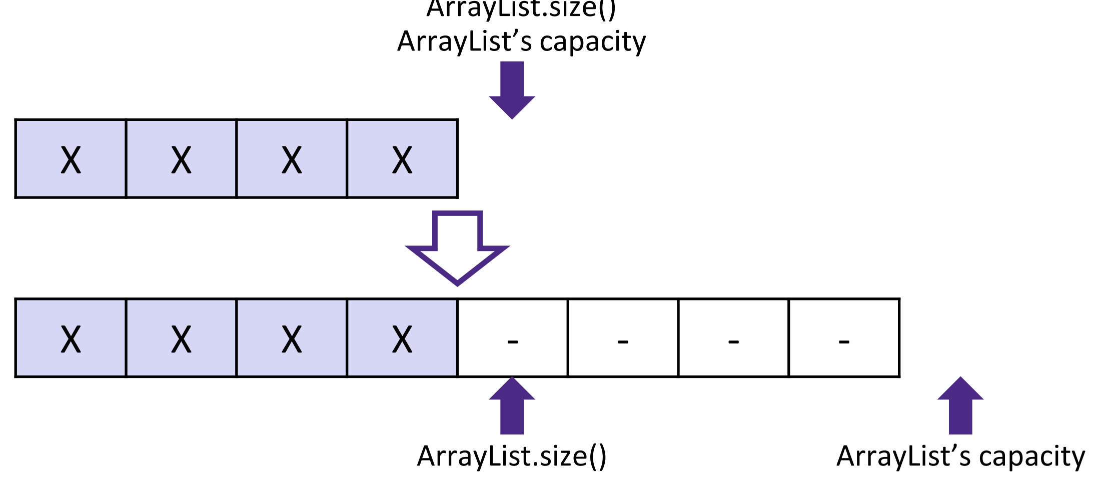
Single Execution	Multiple Executions
Worst Case	Amortized Worst Case
Best Case	Amortized Best Case
Average Case	*Amortized Average Case*

Amortized Analysis: `ArrayList.add()`

- Consider adding an element to an array-backed structure
  - Eg, Java's ArrayList



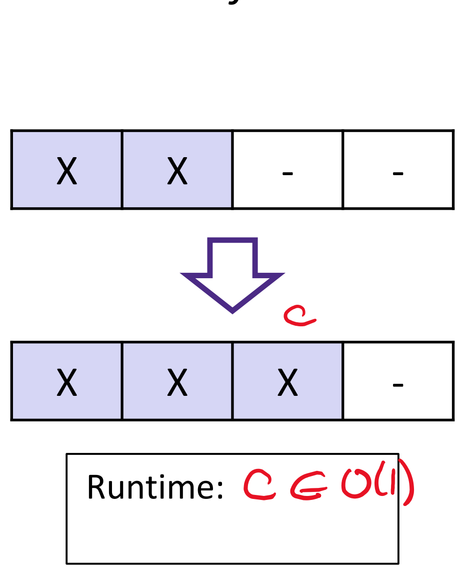
- When the underlying array fills, we allocate and copy contents



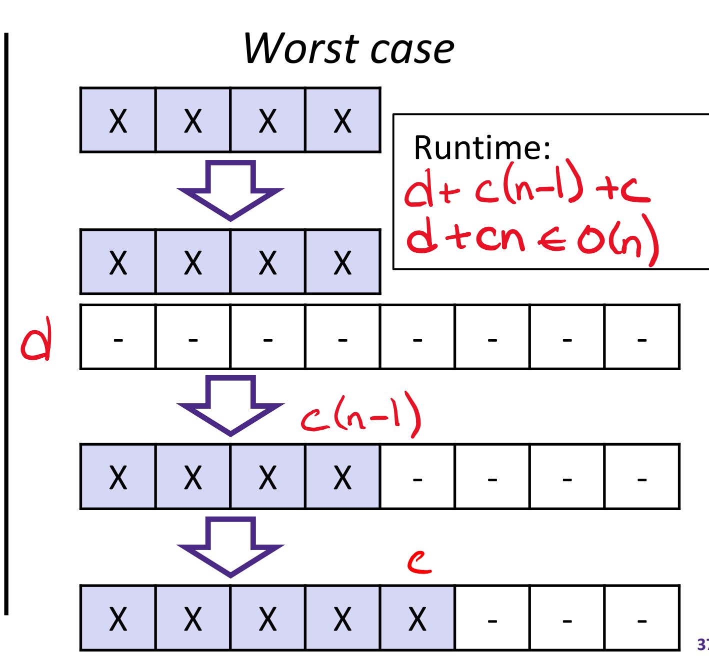
`ArrayList.add()` Runtime (1 of 2)

- We know that copying a single element and allocating arrays are both constant-time operations
  - Let's call their runtimes 'c' and 'd', respectively

Most of the time



Worst case

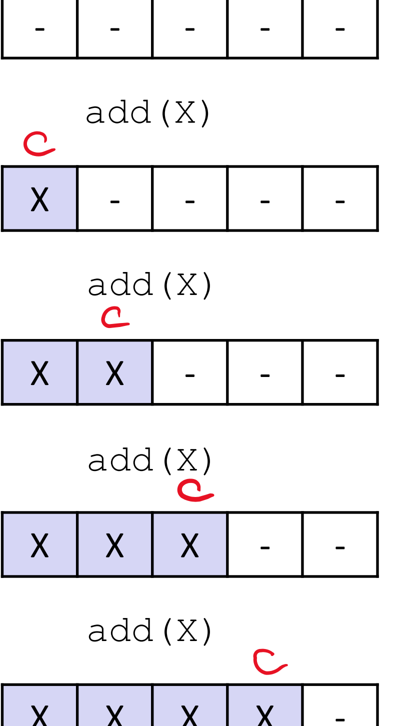


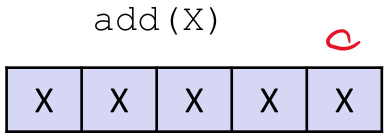
ArrayList.add() Runtime (2 of 2)

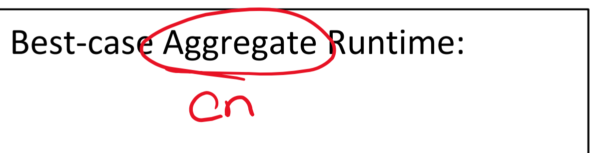
Single Execution	Multiple Executions
Worst Case: O(n)	Amortized Worst Case:
Best Case: O(1)	Amortized Best Case:

- Some applications *cannot tolerate* the “occasional O(n) behavior”
- Other applications *can tolerate* “occasional O(n) behavior” if we can show that it’s “not too bad” / “not too common”

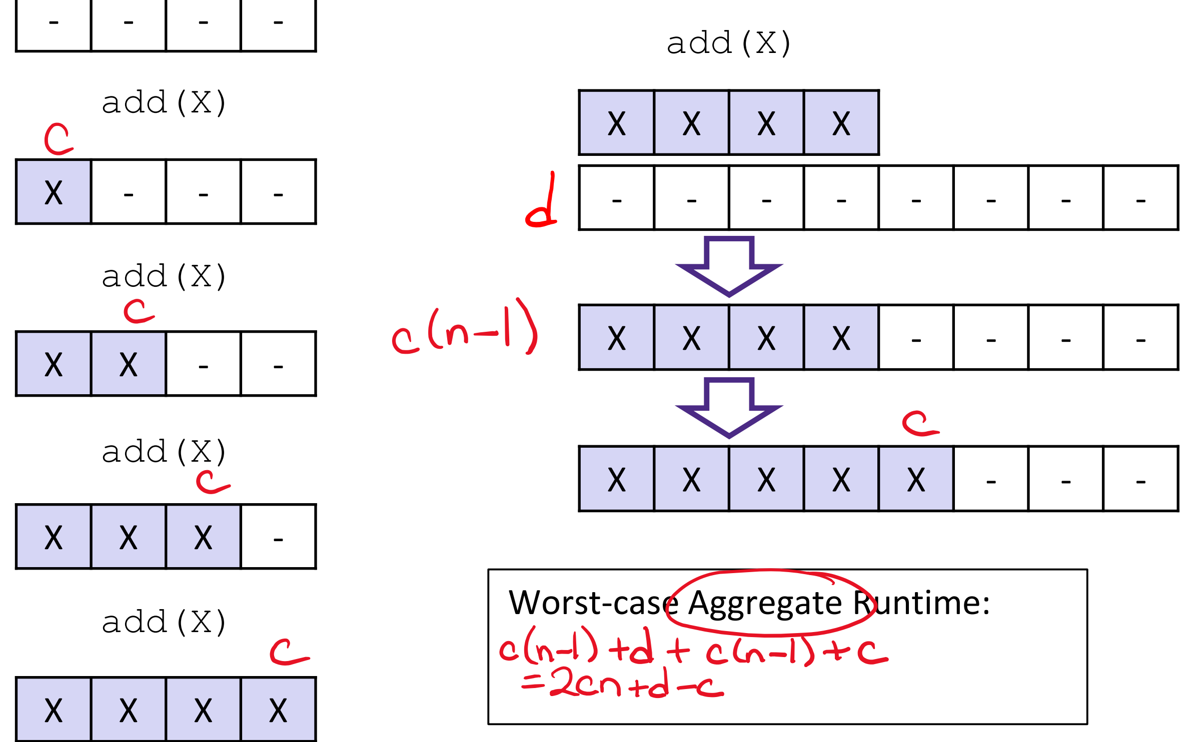
`ArrayList.add()`: Best-Case Aggregate Runtime







`ArrayList.add()`: Worst-Case Aggregate Runtime



Amortized Analysis Intuition

- See Weiss, ch 11, for formal methods
- But the intuition is: if our client is willing to tolerate it, we will "smooth" the *aggregate cost of n operations* over n itself

Single Execution	Multiple Executions
Worst Case: $O(n)$	Amortized Worst Case: $\frac{2cn+d-c}{n} \in O(1)$
Best Case: $O(1)$	Amortized Best Case: $\frac{cn}{n} \in O(1)$

- Note: we increased our array size by a factor of n (eg, 2n, 3n, etc). What if we increased it by a constant factor (eg, 1, 100, 1000) instead?