# buildHeap

CSE 332 Spring 2021

**Instructor:** Hannah C. Tang

**Teaching Assistants:**

| | | |
|---|---|---|
| Aayushi Modi | Khushi Chaudhari | Patrick Murphy |
| Aashna Sheth | Kris Wong | Richard Jiang |
| Frederick Huyan | Logan Milandin | Winston Jodjana |
| Hamsa Shankar | Nachiket Karmarkar | |

# gradescope

**gradescope.com/courses/256241**

- Given the following list of elements, what ordering results in the *worst case* aggregate runtime for add() : 16, 32, 4, 57, 80, 43, 2

# Announcements

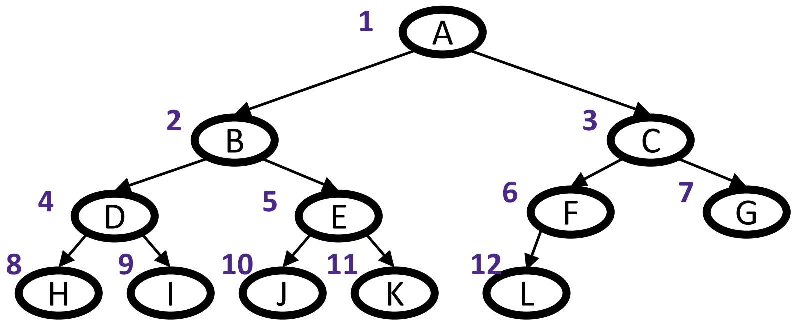
- Quiz #1 released tomorrow morning, due 11am (PDT)
  - Nothing we cover today is on quiz 1
  - Recommended group size: 2-4 students
- Project #1 due Thursday @ 8pm (PDT)
  - If you’re struggling with your partnership, please reach out!
  - If you’re struggling with the project, schedule 1:1 time!
    - Don’t forget to check your pipelines for failures (we do!)
- Sorry about the due dates; this is the only one the entire quarter!

# Lecture Outline

- Heaps, cont.
  - Heaps, cont.
  - **Floyd's `buildHeap` Algorithm**
  - Farewell to Heaps …

# Array Representation of a Binary Heap

- In lecture and in Weiss, skip index 0 to make the math simpler
  - Though, it's a good place to store the current size of the heap
  - P1 doesn't skip; starts counting from 0




- From node i:
  - left child: $2i$
  - right child: $2i+1$
  - parent: $\lfloor \frac{i}{2} \rfloor$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 |

# Evaluating the Array Implementation

- Advantages:
  - Minimal amount of wasted space:
    - Only index 0 and any unused space on right in the array
    - No "holes" due to complete tree property
    - No wasted space representing tree edges
  - Fast lookups:
    - Benefit of array lookup speed
    - Multiplying / dividing by 2 is extremely fast (see CSE 351 and bit-shifting)
    - Last used position is easily found by using the PQueue's size for the index
- Disadvantages:
  - If the array gets too full, needs to be resized
  - If the array is too empty, wastes space and needs to be resized
- ***Advantages outweigh disadvantages: this is how it is done!***

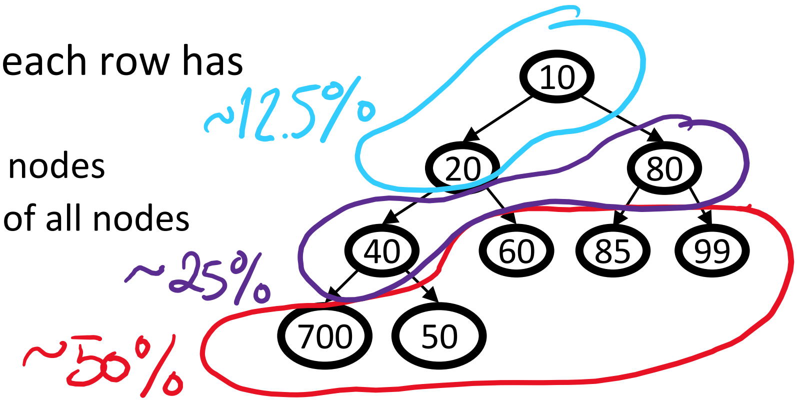
# O(1) average-case add()?! (1 of 2)

- 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 a 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 item 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 priority; 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.
  - Heaps, cont.
  - **Floyd's buildHeap Algorithm**
  - Farewell to Heaps …

# 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 data structure 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

# Robert Floyd

- Turing Award winner
  - Floyd-Warshall algorithm (all-pairs shortest path)
  - Programming parsing and semantics
- Invented in-place Heapsort

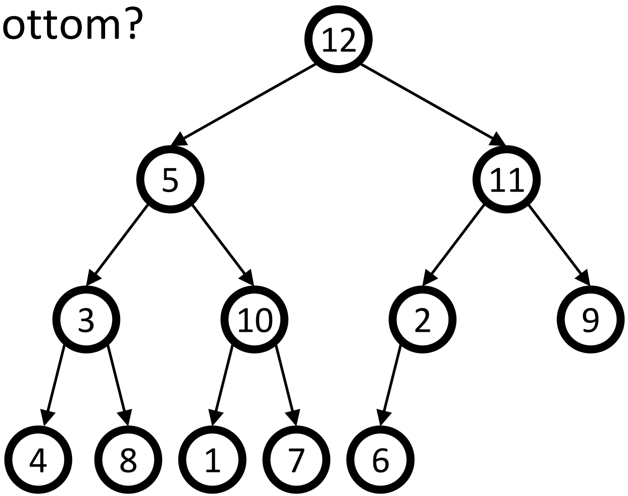
*By Source, Fair use, https://en.wikipedia.org/w/index.php?curid=59539154*

# 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

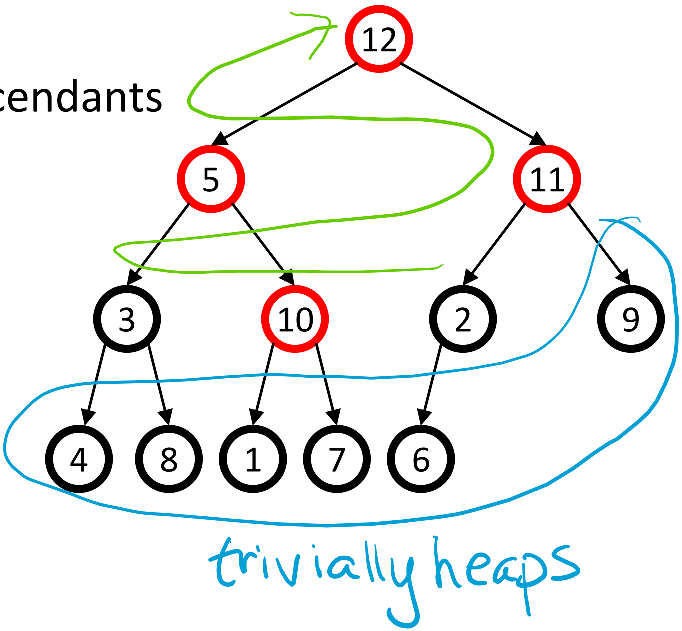
- Bottom-up:
  - Leaves are already in heap order
  - Work *up toward the root* one level at a time, percolating *downwards*

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

Note: P1 doesn't skip; starts counting from 0

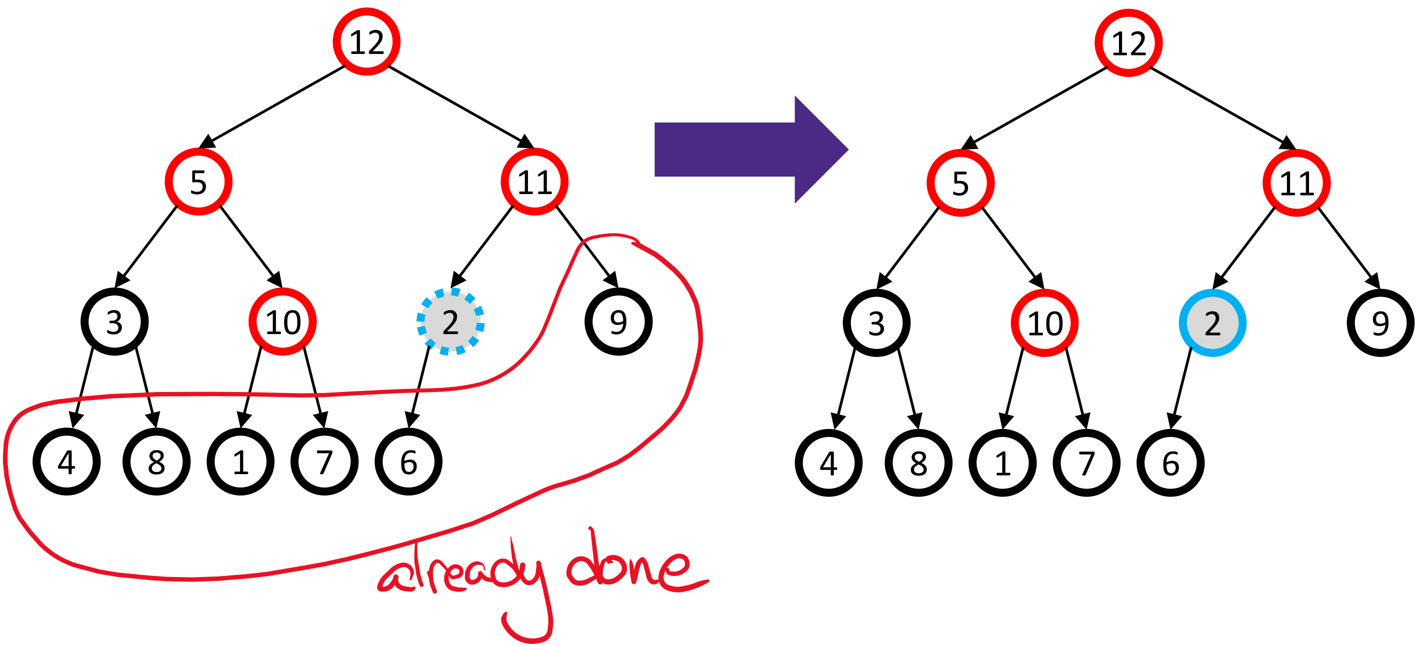
# `buildHeap` Example

- 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!



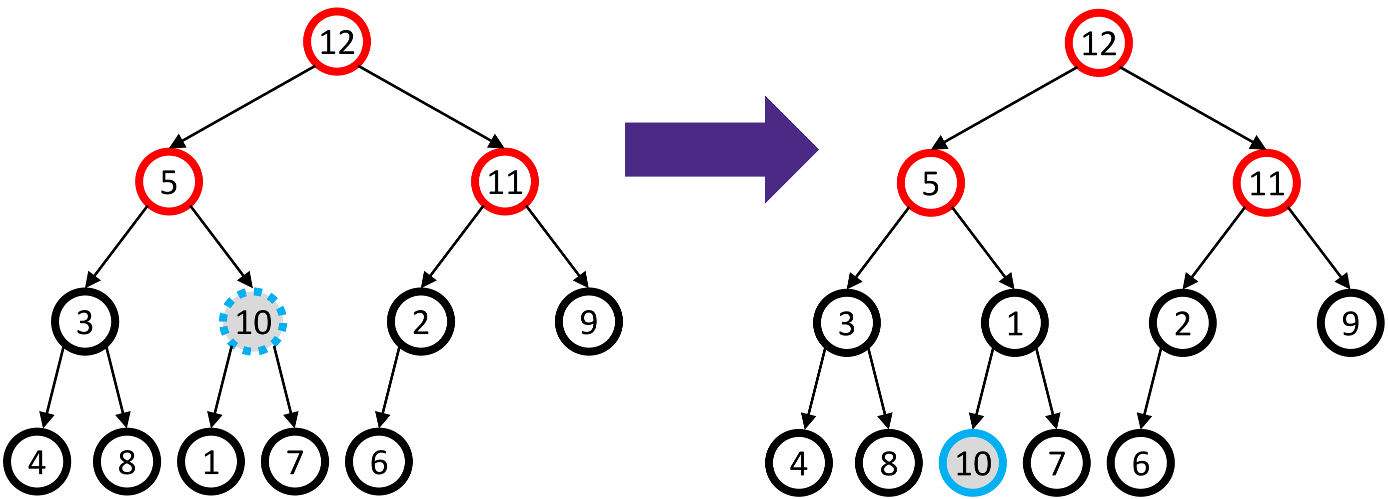
# buildHeap Example: Step 1

- Happens to already be less than child



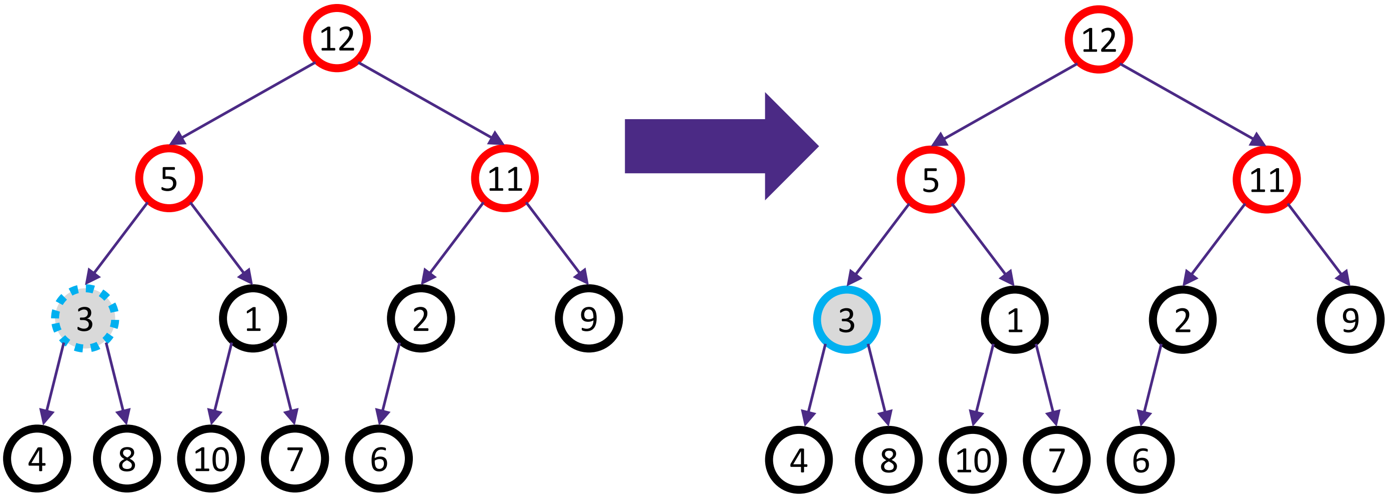
# `buildHeap` Example: Step 2

- Percolate down (notice that this moves up '1')



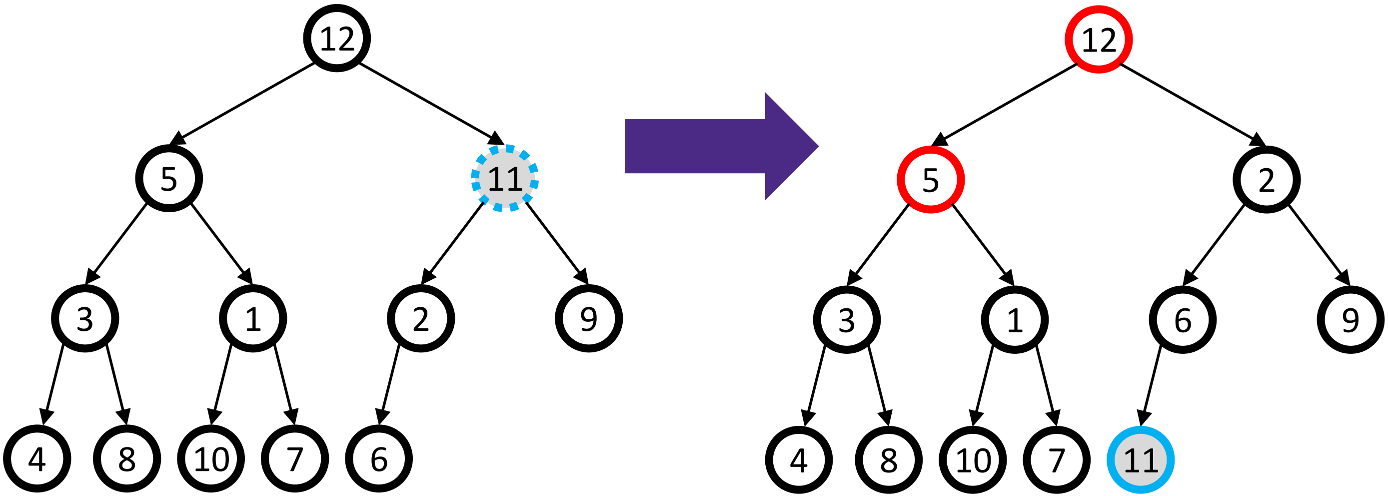
# `buildHeap` Example: Step 3

- Another nothing-to-do step



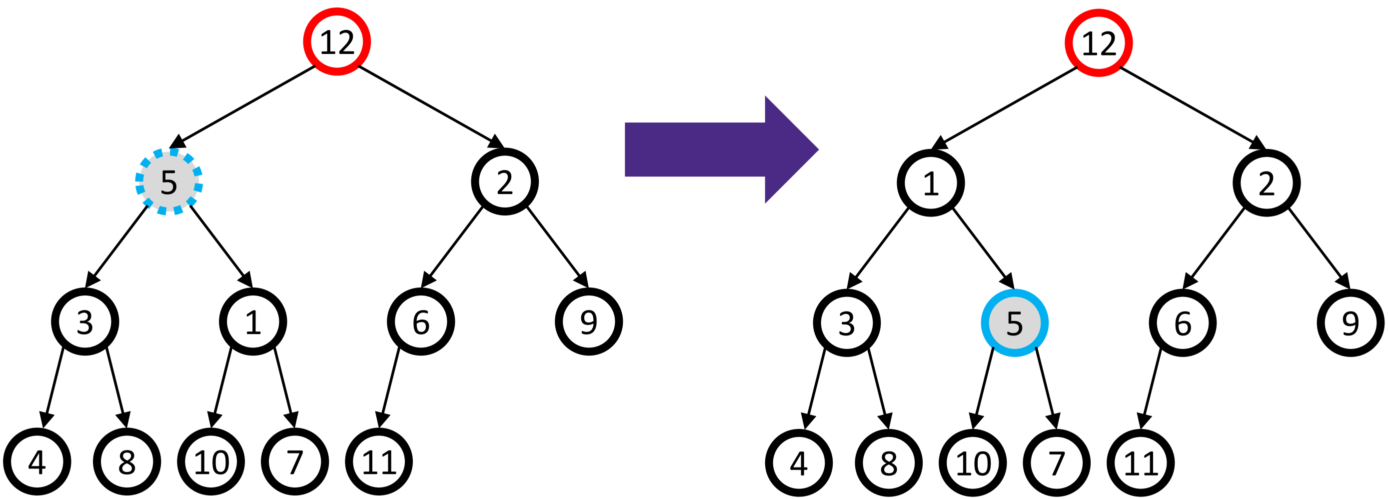
# `buildHeap` Example: Step 4

- Percolate down. Which nodes got moved?



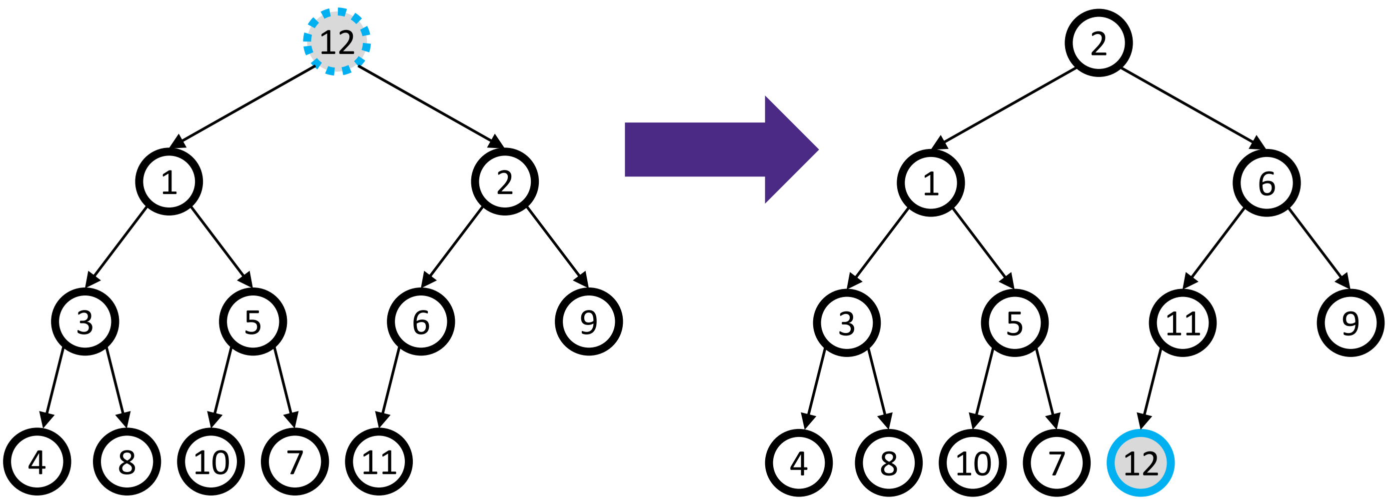
# buildHeap Example: Step 5

- Again, percolate down



# 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;
   }
}
```

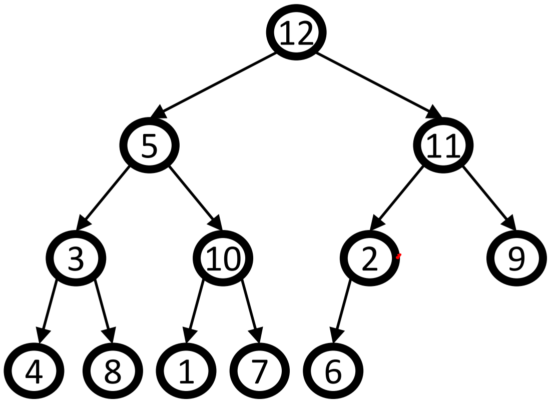
# Floyd's `buildHeap`: Correctness

- ***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

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

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

# 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 |

# 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;
  }
}
```

# 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)

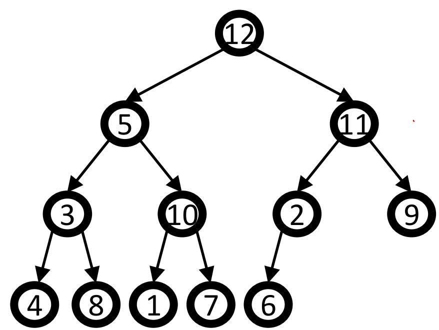
Actual runtime: $\frac{n}{2}\sum_{i=0}^{\lfloor \log n \rfloor} \frac{1}{2^i} i$

We know: $\sum_{i=0}^{\infty} \frac{1}{2^i} i$

We know $\frac{n}{2}\left(\sum_{i=0}^{\lfloor \log n \rfloor} \frac{1}{2^i} i\right) < \frac{n}{2}\left(\sum_{i=0}^{\infty} \frac{1}{2^i} i\right)$ (less than)

$< \frac{n}{2} \cdot 2$

∴ Runtime is $\Theta(n)$



# 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.
  - Heaps, cont.
  - Floyd's buildHeap Algorithm
  - **Farewell to Heaps …**

# Evaluating Heaps

| | add | deleteMin |
|---|---|---|
| Unsorted Array | add at end: O(1) | search: O(N) |
| Sorted Circular Array | search + shift: O(N) | move front pointer: O(1) |

- Unsorted Array: not sorted "enough" to provide fast deletion
- Sorted Array: "too" sorted to provide fast insertion
- Binary Heap: "just enough" sorting to provide "fast enough" insertion and deletion

| | | |
|---|---|---|
| Binary Heap | O(log N), but O(1) expected | O(log N) |

# 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?