

Heaps cont.; Binary Search Trees

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-07-06A

Announcements

- ❖ Quiz 1 has been graded - A lot of people still worked solo
 - People who work together learn and perform better
- ❖ P2 out soon! Partner survey out Tuesday, due Wednesday 6pm
 - We can't pair anyone until EVERYONE has filled it out 😞 😞 😞
- ❖ Ex 4, 5 are out!
- ❖ Ex 2, 3 due tonight, P1 due Tuesday night
- ❖ We are always available for 1:1 meetings! Let us know how we can help!

pollev.com/332summer :: tinyurl.com/332-07-06A

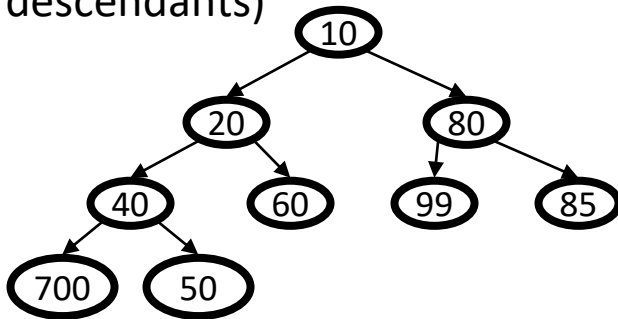
Lecture Outline

- ❖ **Review of Heap Add()/DeleteMin()**
 - buildHeap()
- ❖ Dictionaries
- ❖ Binary Search Trees
 - Binary Trees != Binary Search Trees
 - Binary Tree traversals
 - Binary Search Trees as Dictionary/Set
 - BST Find/Contains

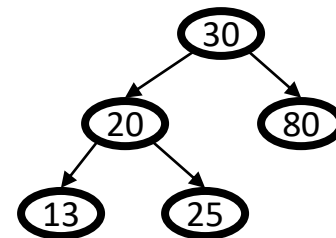
pollev.com/332summer :: tinyurl.com/332-07-06A

Binary (Min-)Heap

- ❖ More commonly known as a *binary heap* or simply a *heap*
 - The “min” refers to the fact that the special priority value is the smallest; a “max heap” tracks the largest priority
- ❖ **Structure Property:** A complete binary tree
- ❖ **Order Property:** Every parent node has a priority value smaller than (or possible equal to) the priority of all its children (and descendants)



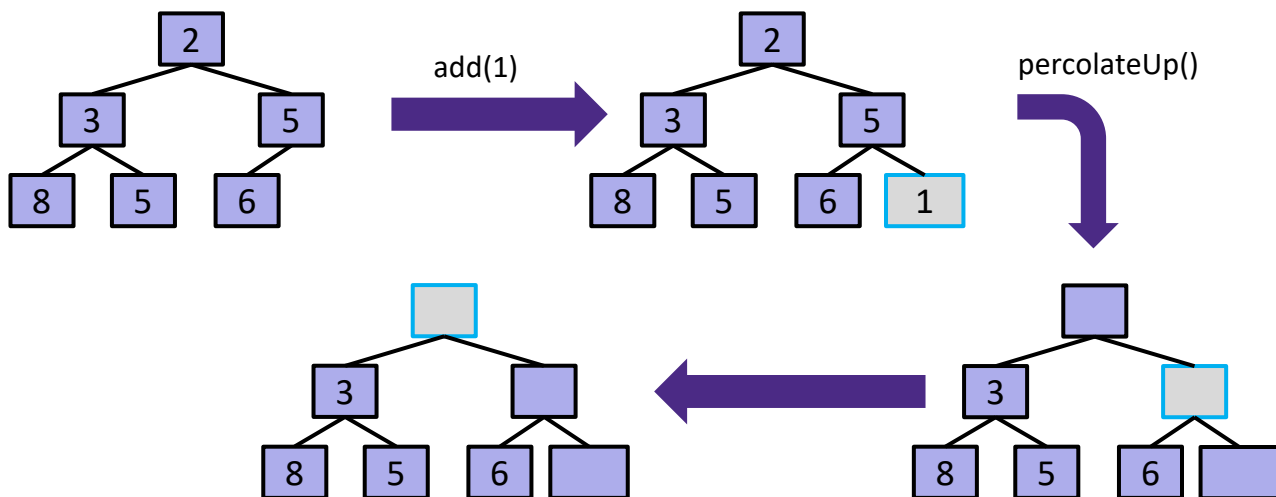
A Heap



Not a Heap

Binary Heap: add()

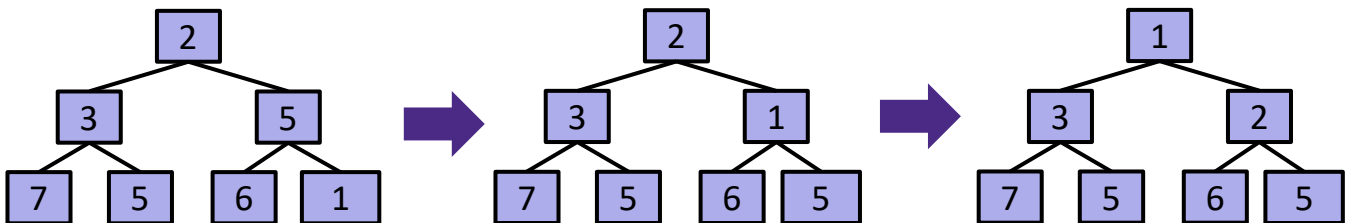
- ❖ Put new node in rightmost position of the last row
- ❖ “Percolate up” to correct layer



percolateUp()

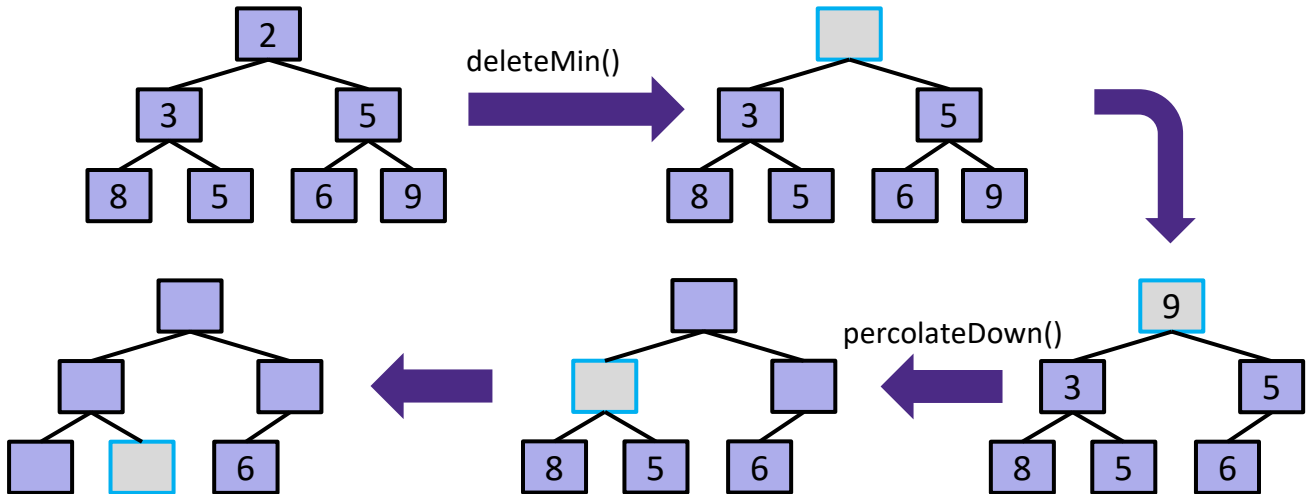
- ❖ `percolateUp()`:
 - Put new item in new location
 - If parent larger, swap with parent, and continue
 - Done if parent \leq item or reached root

- ❖ Why does this work? What is the run time?



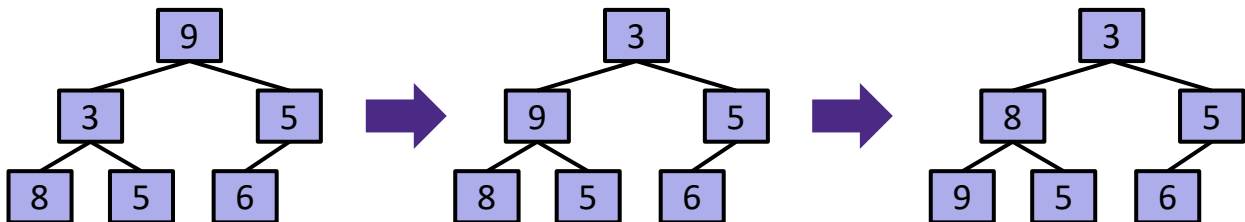
Binary Heaps: deleteMin()

- ❖ Move rightmost node in last row to root
- ❖ “Percolate down” to correct layer



percolateDown()

- ❖ percolateDown:
 - Keep comparing with both children
 - Move *smaller* child up and go down one level
 - Done if both children are \geq item or reached a leaf node
- ❖ Why does this work? What is the run time?

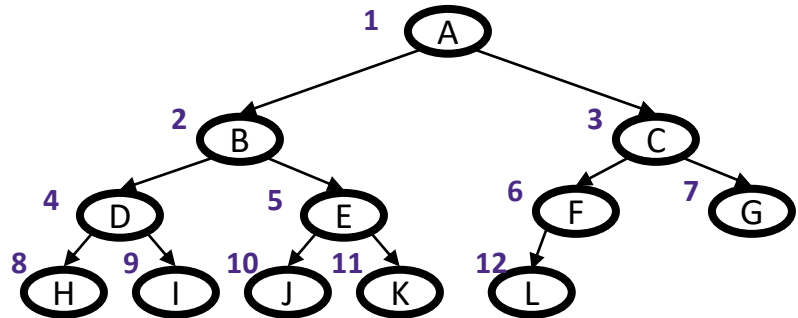


Array Representation of a Binary Heap

- ❖ We skip index 0 to make the math simpler, though it's a good place to store the current size of the heap
 - Note: Exercises and P1 start counting from 0

❖ From node i :

- left child:
- right child:
- parent:

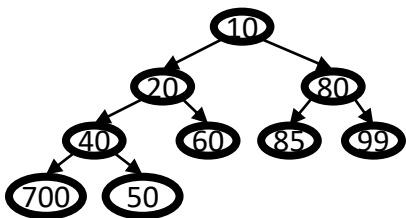


	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pseudocode: add()

```
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;
}
```



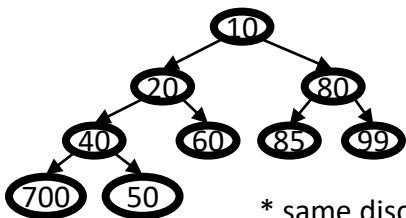
Disclaimers:

- This pseudocode uses ints. In real use, you will have nodes with priorities and values
- Exercises and P1 start counting from 0

	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()

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



* same disclaimers apply

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

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

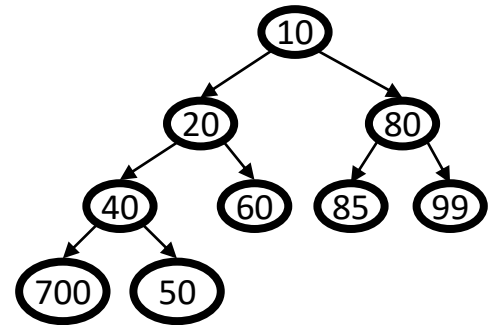
$O(1)$ average-case $\text{add}()$?! (2 of 2)

- ❖ In a complete binary tree, each row has 2x nodes of its parent row

- Bottom level has $\sim 1/2$ of all nodes
- Second to bottom has $\sim 1/4$ 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 $\frac{1}{2}$ of priorities
 - Second to bottom, next $\frac{1}{4}$
 - ...
- Expect to only percolate up 1-2 levels



Lecture Outline

- ❖ Review of Heap Add()/DeleteMin()
 - **buildHeap()**
- ❖ Dictionaries
- ❖ Binary Search Trees
 - Binary Trees != Binary Search Trees
 - Binary Tree traversals
 - Binary Search Trees as Dictionary/Set
 - BST Find/Contains

pollev.com/332summer :: tinyurl.com/332-07-06A

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?**

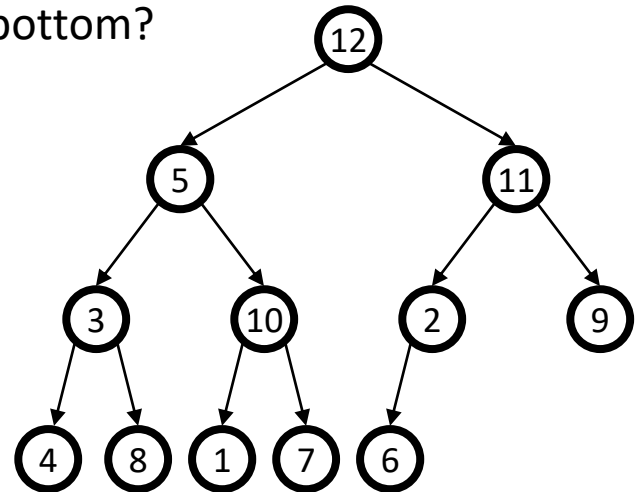
Floyd's buildHeap Method

- ❖ 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

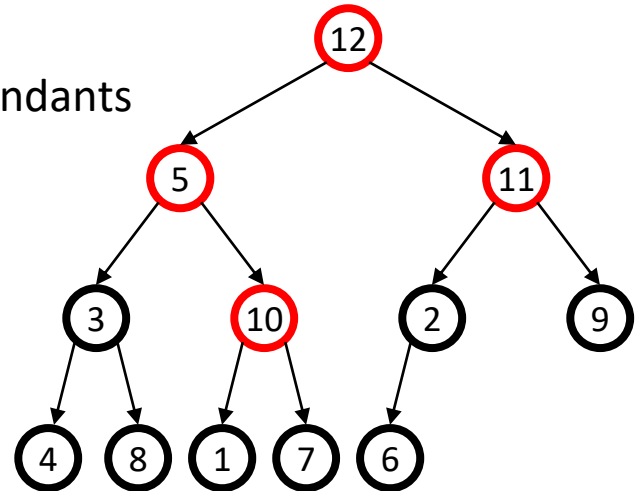
❖ Bottom-up:

- 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;
    }
}
```

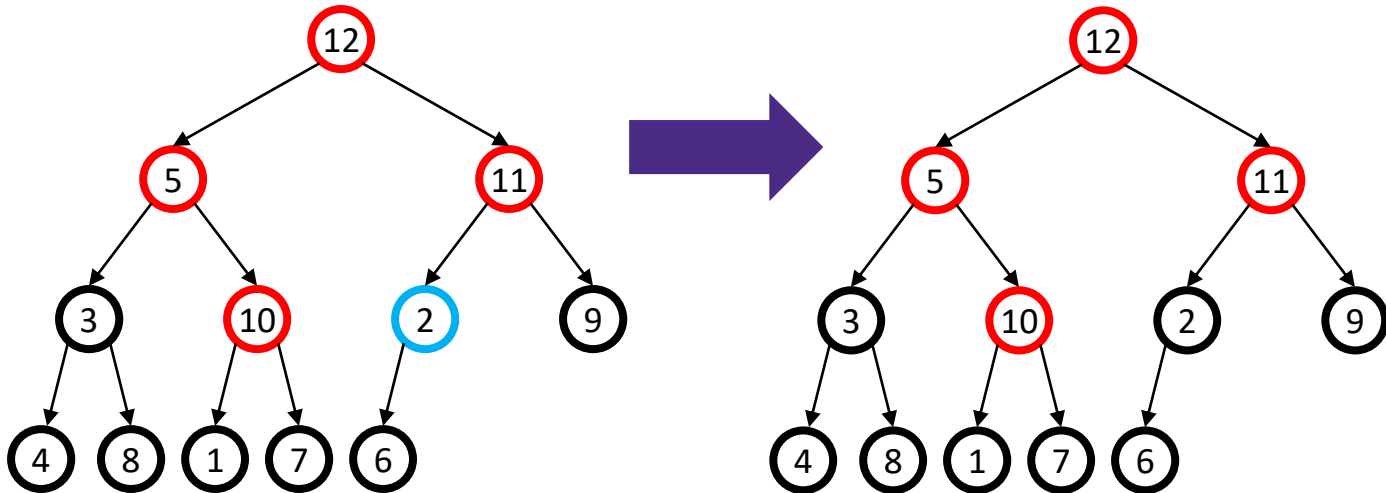
buildHeap Example

- ❖ 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**!



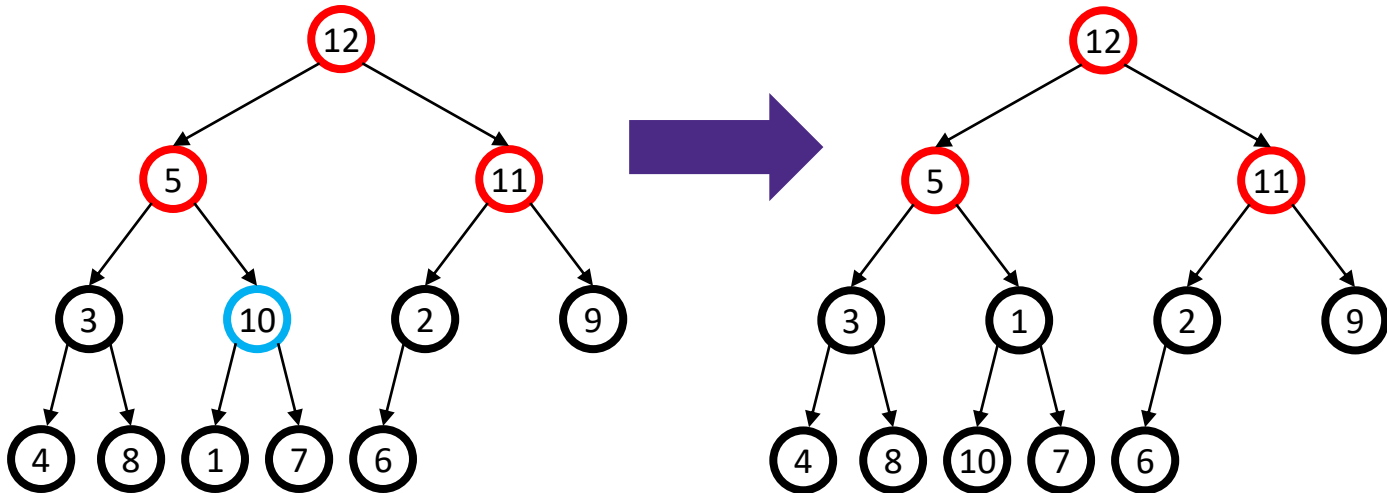
buildHeap Example: Step 1

- ❖ Happens to already be less than child



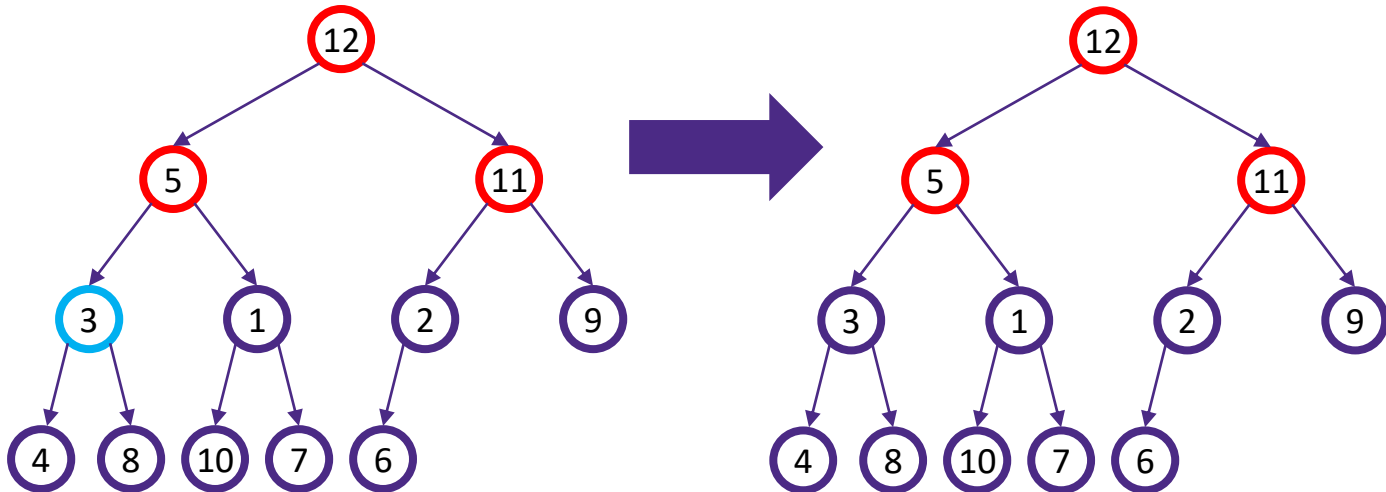
buildHeap Example: Step 2

- ❖ Percolate down (notice that moves 1 up)



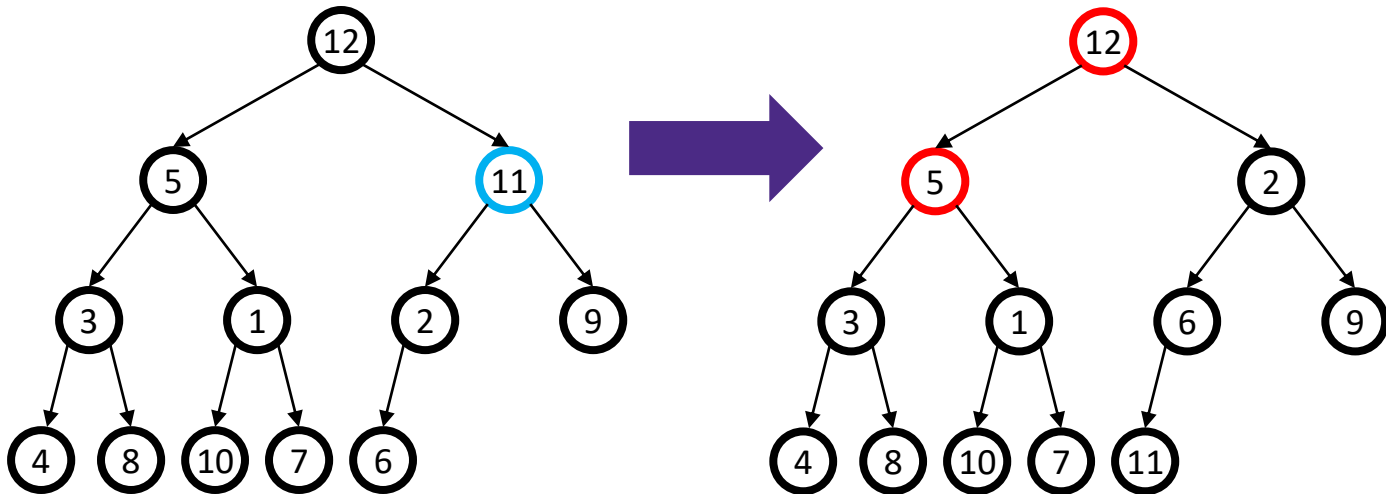
buildHeap Example: Step 3

- ❖ Another nothing-to-do step



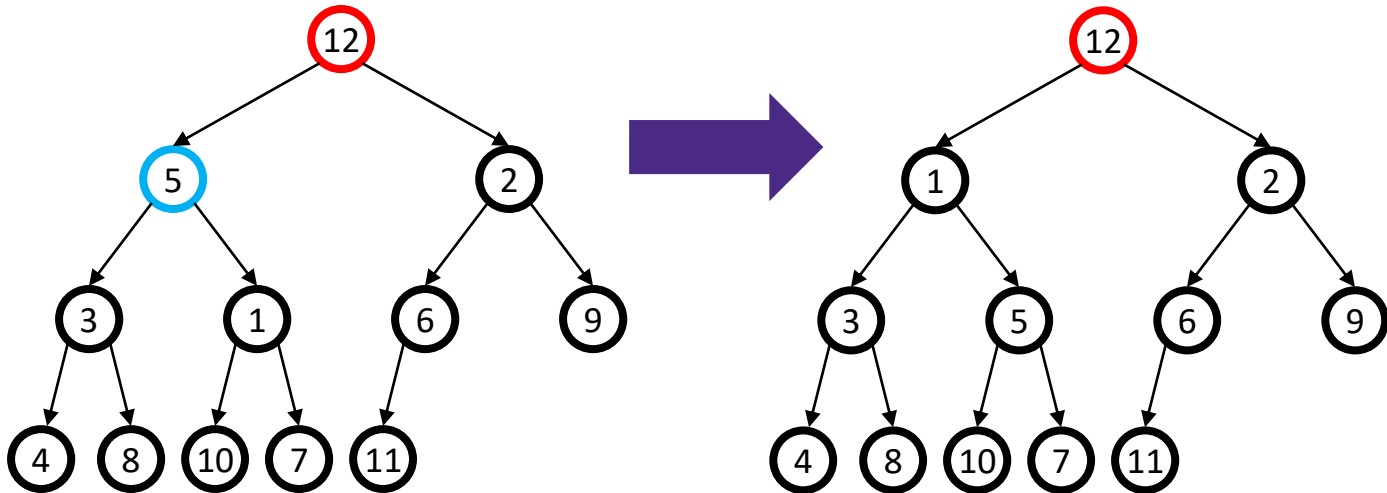
buildHeap Example: Step 4

- ❖ Percolate down as necessary



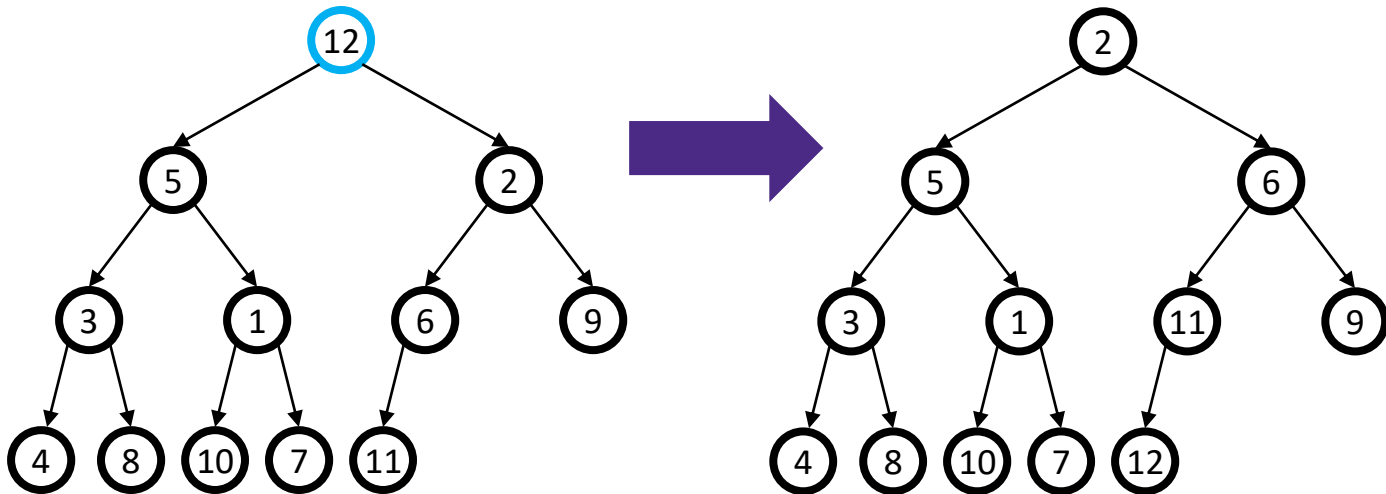
buildHeap Example: Step 5

- ❖ 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)

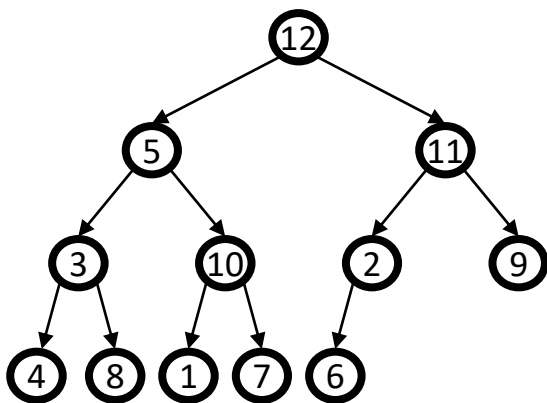
```
void buildHeap(arr) {
    n = arr.length
    for(i = n/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i, val);
        arr[hole] = val;
    }
}
```

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
- ❖ 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;
    }
}
```

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;  
    }  
}
```

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 $((1/2) + (2/4) + (3/8) + \dots) = 2$

- See page 4 of Weiss
- Also see Weiss 6.3.4, sum of heights of nodes in a perfect tree

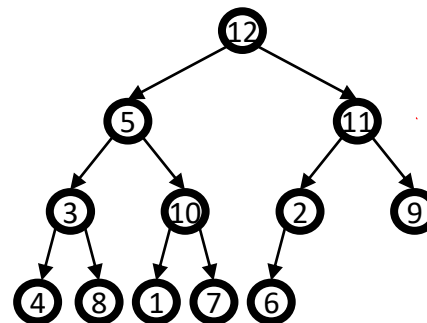
■ So Floyd's `buildHeap` is $n/2 * 2 = O(n)$

Actual runtime:

$$\frac{n}{2} \sum_{i=0}^{\lfloor \lg n \rfloor} \frac{1}{2^i} i$$

We know:

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



Lessons from `buildHeap`

- ❖ Without `buildHeap`, our ADT let clients implement their own in $\Theta(n \log n)$ 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 $O(n)$

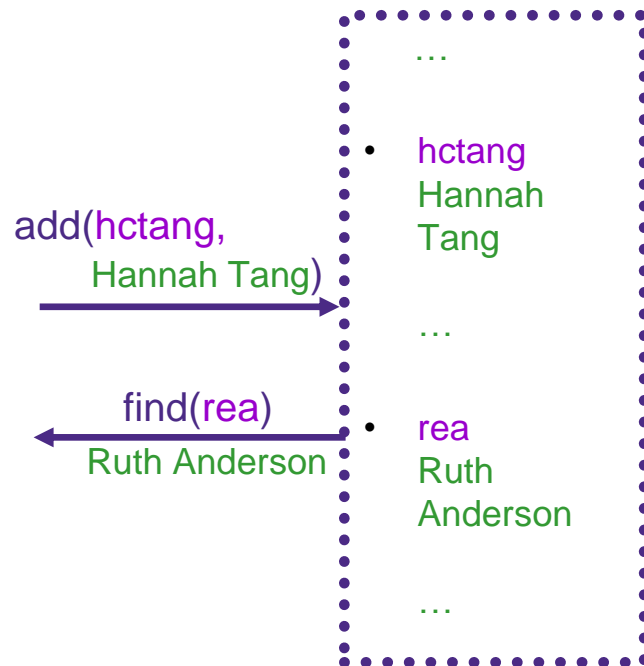
Lecture Outline

- ❖ Review of Heap Add()/DeleteMin()
 - buildHeap()
- ❖ **Dictionaries**
- ❖ Binary Search Trees
 - Binary Trees != Binary Search Trees
 - Binary Tree traversals
 - Binary Search Trees as Dictionary/Set
 - BST Find/Contains

pollev.com/332summer :: tinyurl.com/332-07-06A

Dictionary ADT

- ❖ Also known as “**Map ADT**”
- ❖ Operations:
 - **add(k, v)** :
 - places (k,v) in dictionary
 - if key already present, typically overwrites existing item
 - **find(k)** :
 - Returns v associated with k
 - **contains(k)** :
 - Returns true if k is in the dictionary
 - **remove(k)** :
 - ...



We will tend to emphasize the keys, but don't forget about the stored values!

Dictionary ADT: Data Structures

- ❖ For a dictionary with n key/value pairs, what is the runtime for:

	<code>insert</code>	<code>find</code>	<code>delete</code>
Unsorted linked list			
Unsorted array			
Sorted linked list			
Sorted array			

Reminder: a dictionary maps *keys* to *values*;
an *item* or *data* refers to the (key, value) pair

Dictionary ADT: Data Structures

- ❖ For a dictionary with n key/value pairs, what is the runtime for:

	<code>insert</code>	<code>find</code>	<code>delete</code>
Unsorted linked list	$O(n)^*$	$O(n)$	$O(n)$
Unsorted array	$O(n)^*$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$	$O(n)$

- ❖ * *Note:* If we allow duplicates keys to be inserted, you could do these in $O(1)$ because you do not need to check for a key's existence before insertion

Dictionary ADT: Better Data Structures

- ❖ We will spend the next several lectures looking at dictionaries:
 - Binary Search Trees
 - AVL trees
 - Binary search trees with guaranteed balancing
 - B-Trees
 - Also always balanced, but different and shallower
 - “B” != “Binary”; B-Trees generally have large branching factor
 - Hash Tables
 - Not tree-like at all

- ❖ Skipping: Other balanced binary search trees
 - Eg, red-black tree (and LLRBs), splay tree

Lecture Outline

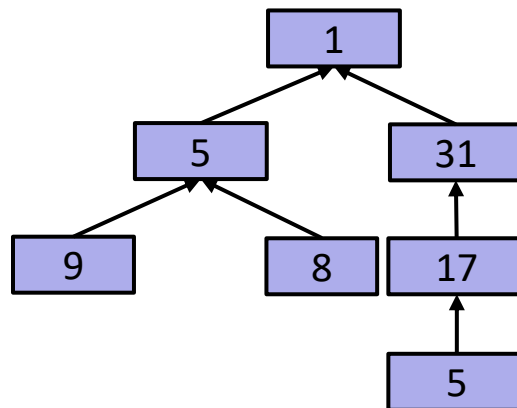
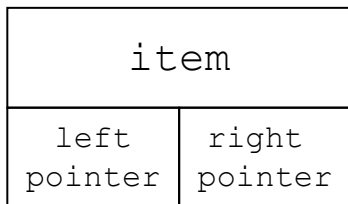
- ❖ Review of Heap Add()/DeleteMin()
 - buildHeap()
- ❖ Dictionaries
- ❖ Binary Search Trees
 - **Binary Trees != Binary Search Trees**
 - Binary Tree traversals
 - Binary Search Trees as Dictionary/Set
 - BST Find/Contains

pollev.com/332summer :: tinyurl.com/332-07-06A

Binary Tree

- ❖ A **Binary Tree** is empty or
 - a root (*with item*)
 - a left subtree (*maybe empty*)
 - a right subtree (*maybe empty*)

- ❖ Representation:



- ❖ For a dictionary, `item` will include a key and a value

Binary Tree: Some Numbers

- ❖ Recall: height of a tree = longest path from root to leaf
 - Count # of edges!
- ❖ For a binary tree of height h :
 - max # of leaves:
 - max # of nodes:
 - min # of leaves:
 - min # of nodes:

Binary Tree: Some Numbers

- ❖ Recall: height of a tree = longest path from root to leaf
 - Count # of edges!
- ❖ For a binary tree of height h :
 - max # of leaves: 2^h
 - max # of nodes: $2^{h+1} - 1$
 - min # of leaves: 1
 - min # of nodes: $h+1$

Lecture Outline

- ❖ Review of Heap Add()/DeleteMin()
 - buildHeap()
- ❖ Dictionaries
- ❖ Binary Search Trees
 - Binary Trees != Binary Search Trees
 - **Binary Tree traversals**
 - Binary Search Trees as Dictionary/Set
 - BST Find/Contains

pollev.com/332summer :: tinyurl.com/332-07-06A

Tree Traversals

❖ A *traversal* is an order for visiting all the nodes of a tree

■ *Pre-order*: root, left subtree, right subtree

■ *In-order*: left subtree, root, right subtree

■ *Post-order*: left subtree, right subtree, root

❖ Sometimes order doesn't matter

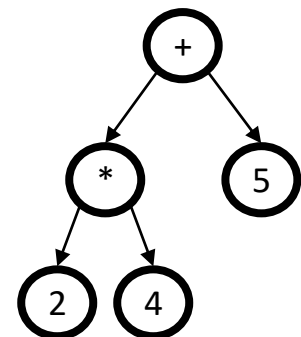
■ Eg: sum all elements

■ Eg: find an element

❖ Sometimes order matters

■ Eg: print tree with indented children (pre-order)

■ Eg: evaluate an expression tree (post-order)



(an expression tree)

PrintIndented:

+

 *

 2

 4

 5

Tree Traversals

❖ A *traversal* is an order for visiting all the nodes of a tree

- *Pre-order*: root, left subtree, right subtree

+ * 2 4 5

- *In-order*: left subtree, root, right subtree

2 * 4 + 5

- *Post-order*: left subtree, right subtree, root

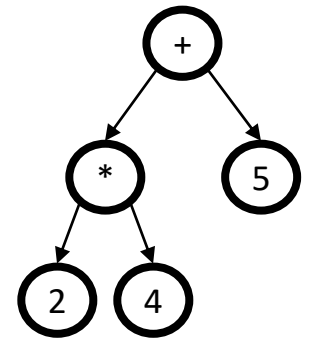
2 4 * 5 +

❖ Sometimes order doesn't matter

- Eg: sum all elements
- Eg: find an element

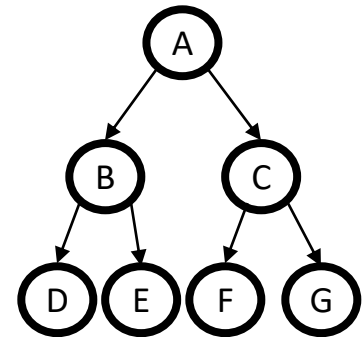
❖ Sometimes order matters

- Eg: print tree with indented children (pre-order)
- Eg: evaluate an expression tree (post-order)



Traversals: Recursive Implementation

```
void inOrdertraversal(Node t) {  
    if (t != null) {  
        traverse(t.left);  
        process(t.element);  
        traverse(t.right);  
    }  
}
```



- ❖ The difference between the 3 traversals (in their recursive implementations) is *when process() gets called*
- ❖ Non-recursive implementation is harder to implement

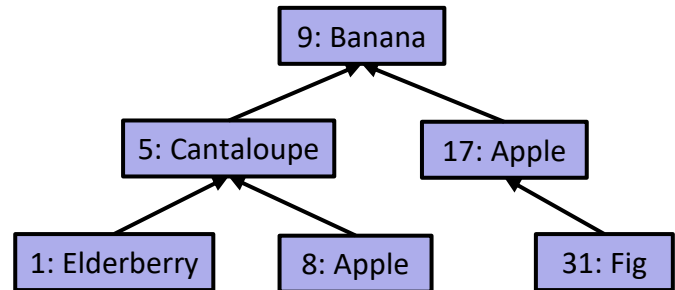
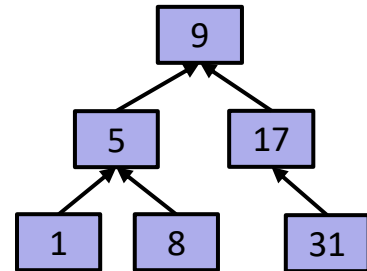
Lecture Outline

- ❖ Review of Heap Add()/DeleteMin()
 - buildHeap()
- ❖ Dictionaries
- ❖ Binary Search Trees
 - Binary Trees != Binary Search Trees
 - Binary Tree traversals
 - **Binary Search Trees as Dictionary/Set**
 - BST Find/Contains

pollev.com/332summer :: tinyurl.com/332-07-06A

Binary Search Trees

- ❖ A **Binary Search Tree** is a binary tree with the following invariant: for every node with key k in the BST:
 - The left subtree only contains keys $<k$
 - The right subtree only contains keys $>k$
- ❖ Reminder: BSTs can also contain (key, value) pairs



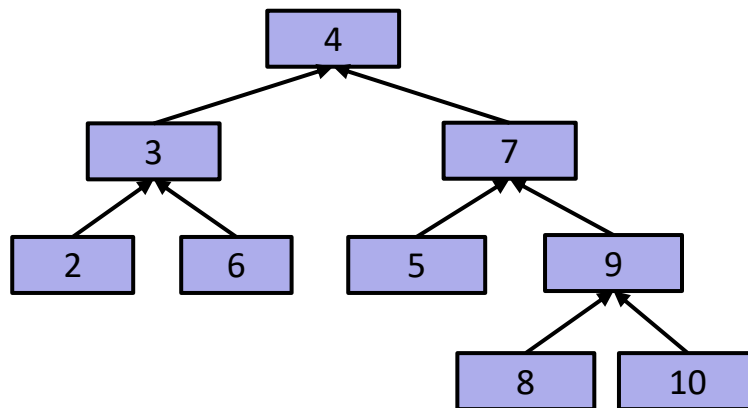
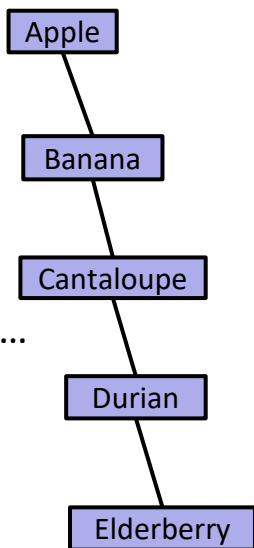
The BST ordering applies recursively to the entire subtree

Poll Everywhere

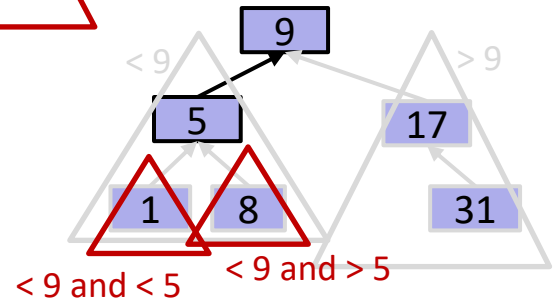
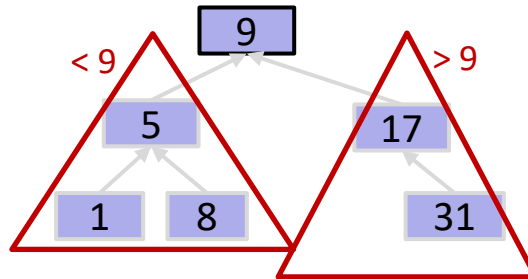
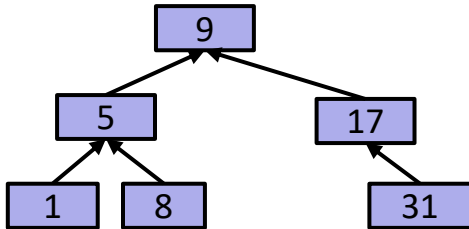
pollev.com/cse332summer

❖ Are these Binary Search Trees?

- A. Yes / Yes
- B. Yes / No
- C. No / Yes
- D. No / No
- E. I'm not sure ...



BST Ordering Applies *Recursively*



Lecture Outline

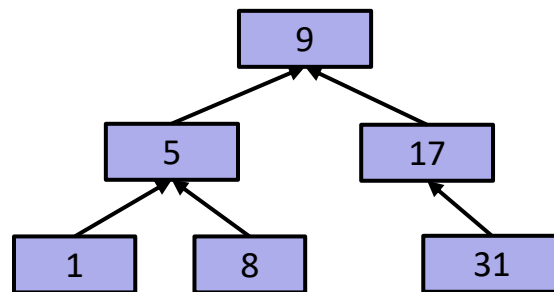
- ❖ Review of Heap Add()/DeleteMin()
 - buildHeap()
- ❖ Dictionaries
- ❖ Binary Search Trees
 - Binary Trees != Binary Search Trees
 - Binary Tree traversals
 - Binary Search Trees as Dictionary/Set
 - **BST Find/Contains**

pollev.com/332summer :: tinyurl.com/332-07-06A

Binary Search Trees: Find/Contains

- ❖ Unsurprisingly, this looks a lot like binary search
- ❖ Can you implement contains by putting the following statements in the correct order?
 - Hint: remember BST's invariants
- ❖ What is find's worst-case runtime?

```
boolean contains(BSTNode n,  
                Key k) {  
  
}
```



A

```
if (n == null)  
    return false;
```

B

```
if (k.equals(n.key))  
    return true;
```

C

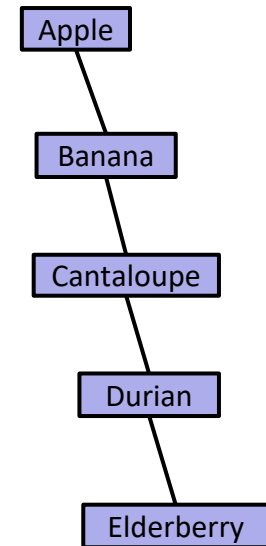
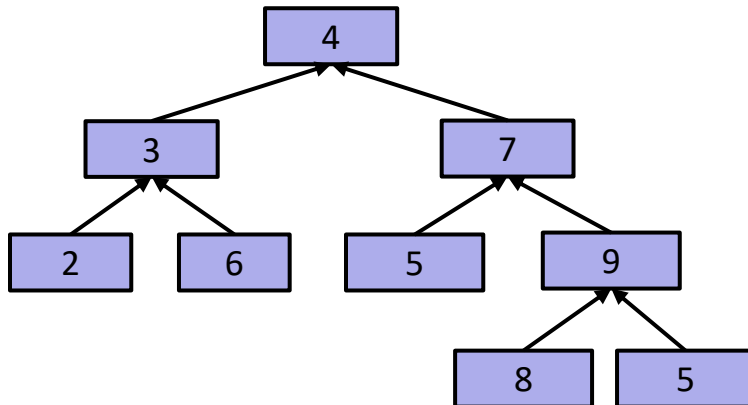
```
if (k < n.k) {  
    return contains(  
        n.left, k);  
}
```

D

```
if (k >= n.k) {  
    return contains(  
        n.right, k);  
}
```

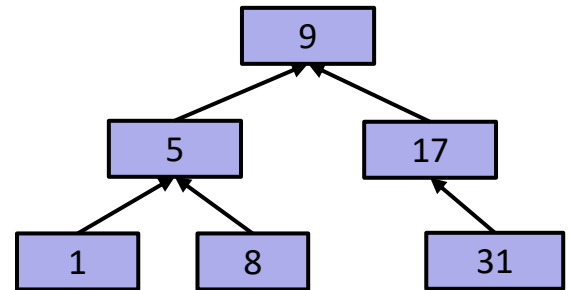
BST Find/Contains's runtime

- ❖ What is find's worst-case runtime, as a function of n ?
- ❖ What is find's worst-case runtime, as a function of *height*?



BST Find/Contains: Iterative

```
boolean contains(BSTNode n,  
                Key k) {  
    while (n != null  
           && n.key != k) {  
        if (k < n.key)  
            n = n.left;  
        else( k > n.key)  
            n = n.right;  
    }  
    if (n == null)  
        return false;  
    return true;  
}
```



Other “finding operations”

- ❖ Find *minimum* node
- ❖ Find *maximum* node
- ❖ Find *predecessor* node
- ❖ Find *successor* node

