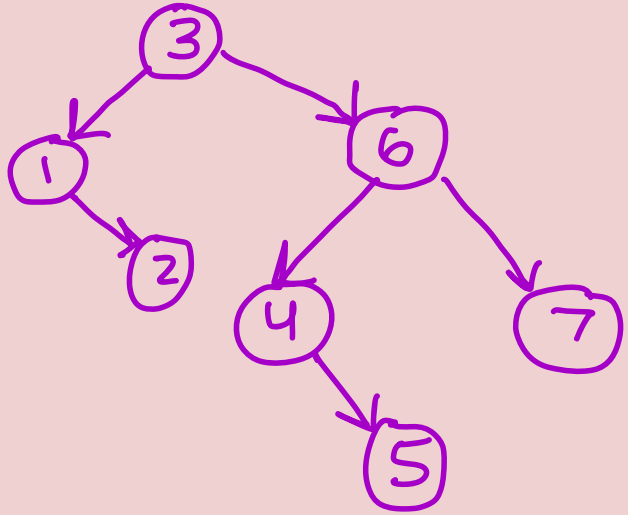
Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

A BST has pre-order traversal: **3, 1, 2, 6, 4, 5, 7**

Which is *not* a leaf in this tree?

root left right



2	(A)
4	(B)
5	(C)
7	(D)

Lecture 18: Heaps and Priority Queues

CS 2110

March 24, 2026

Today's Learning Outcomes

75. Describe the invariants of a heap and determine whether they are satisfied by a given binary tree.

76. Translate between the tree and array representations of a heap.

77. Implement operations on a heap and determine their time/space complexities.

78. Use a heap to implement a priority queue and analyze its performance.

Motivation: Emergency Room Triage

Patients arrive arbitrarily with a variety of medical issues with different urgencies.

Hospital has limited bed spaces, and patients must wait for one to become available to be admitted.

Triage = prioritize most critical patients

We want a collection that lets us arbitrarily add patients with assigned priorities and systematically remove the highest-priority patient.

New ordered collection: Priority Queue

The PriorityQueue<T> ADT

Like all "Ordered collections", supports 4 operations:

1. Add another element

```
/** Adds with associated priority */
void add(T elem, double priority);
```

2. Remove and return a specific element (ADT determines which)

```
/** Removes + returns highest-priority
                                element */
T remove();
```

3. Access next element that will be removed (without removing)

```
/** Returns highest-priority element */
T peek();
```

4. Check if there are more elements to process

```
boolean isEmpty();
```

Approach 1: Compose with `List`

- Introduce auxiliary Entry type to connect element with its priority
- Always add() at end
- Linear scan to find highest priority element

add(): O(1) (amortized)

remove(): O(N)

peek(): O(N)

isEmpty(): O(1)

How can we avoid scanning all elements each time?

space: O(N)

Poll Everywhere

PollEv.com/javabear text **javabear** to **22333**

As an alternate approach, we can compose with a `List` that stores its elements **sorted by priority**.

DAL = `DynamicArrayList`
SLL = `SinglyLinkedList`

What will be the runtime of add( ) if we do this?

DAL: Ascending Sort SLL: Descending Sort

isEmpty(), peek(), remove() are all O(1)

memory shift is O(N)

search is O(N)

DAL: $O(\log N)$ **SLL:** $O(\log N)$	**(A)**	**DAL:** $O(N)$ **SLL:** $O(\log N)$	**(C)**
DAL: $O(\log N)$ **SLL:** $O(N)$	**(B)**	**DAL:** $O(N)$ **SLL:** $O(N)$	**(D)**

Toward a Better Approach

Unsorted lists require O(N) scan to remove element

Sorted lists require O(N) add() to maintain their invariant

Can we do better, so all operations better than O(N)?

- Middle ground: a weaker invariant than fully sorted that we can:
  - Maintain efficiently
  - Gives fast way to locate highest priority element

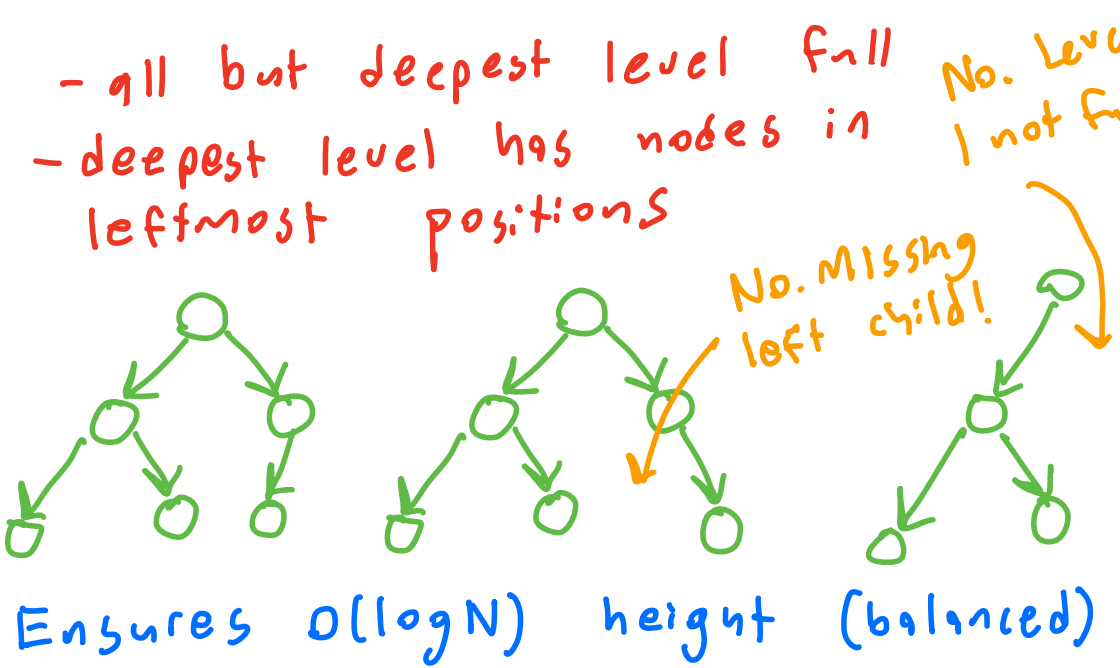
Yes. Heaps are a data structure that does this!

Max Heaps

A max heap is a binary tree with comparable elements that maintains two additional invariants:

1. Shape Invariant

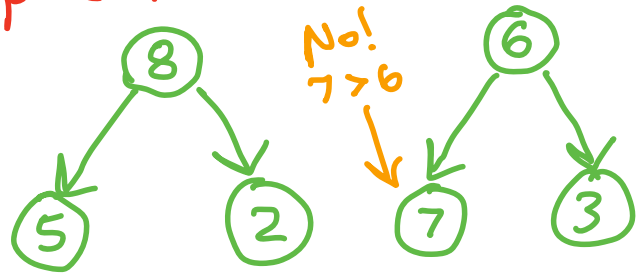
- all but deepest level full
- deepest level has nodes in leftmost positions



Ensures O(log N) height (balanced)

2: Heap order invariant

- every (non-root) node is "less than" its parent



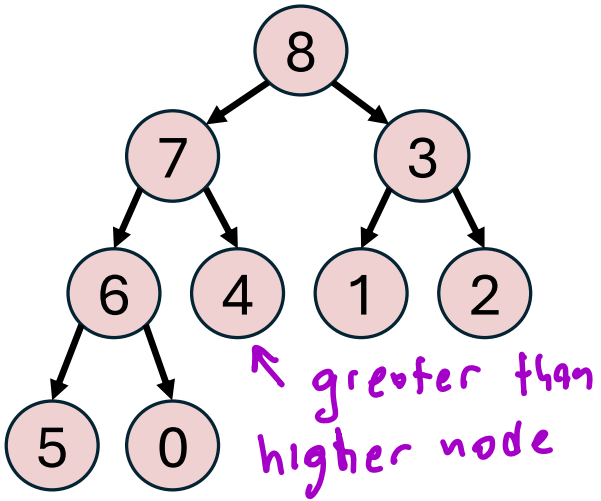
Ensures largest element at tree root.

Poll Everywhere

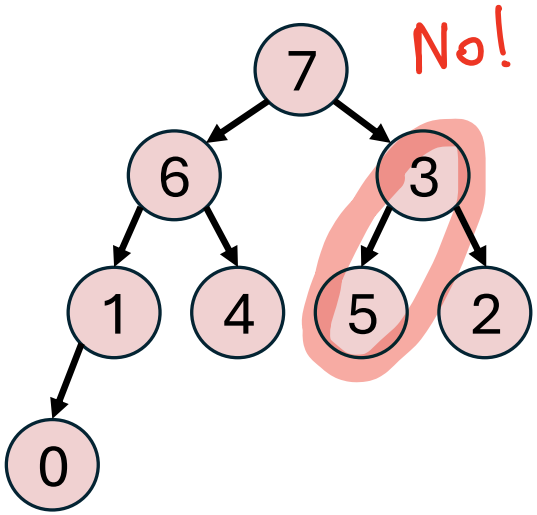
PollEv.com/javabear text **javabear** to **22333**

Which of the following is a valid Max Heap?

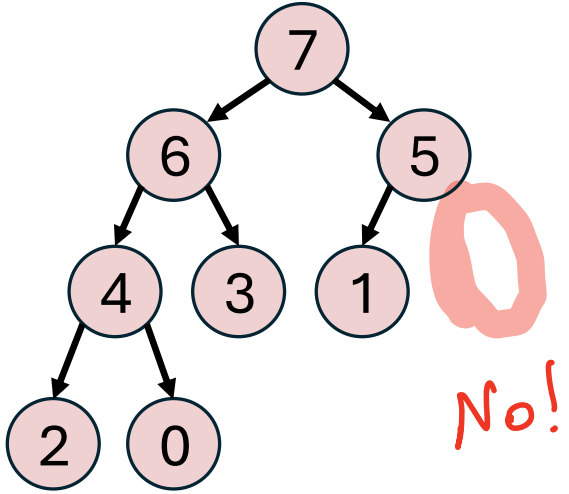
(A)



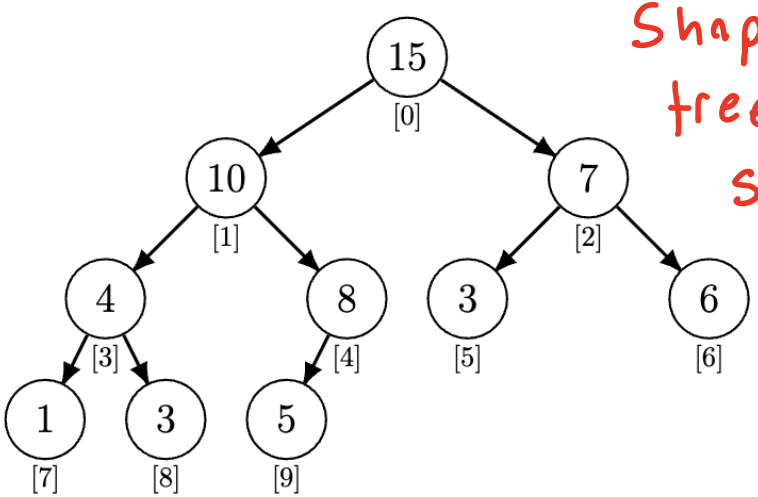
(B)



(C)



Array Representation



Shape invariant ensures a unique tree structure for each tree size.

15	10	7	4	8	3	6	1	3	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Unambiguously reconstruct tree level by level

For node at index i:

left child at index $2i+1$

right child at index $2i+2$

parent at index $\left\lfloor \frac{i-1}{2} \right\rfloor$

Coding Demo: MaxHeap State

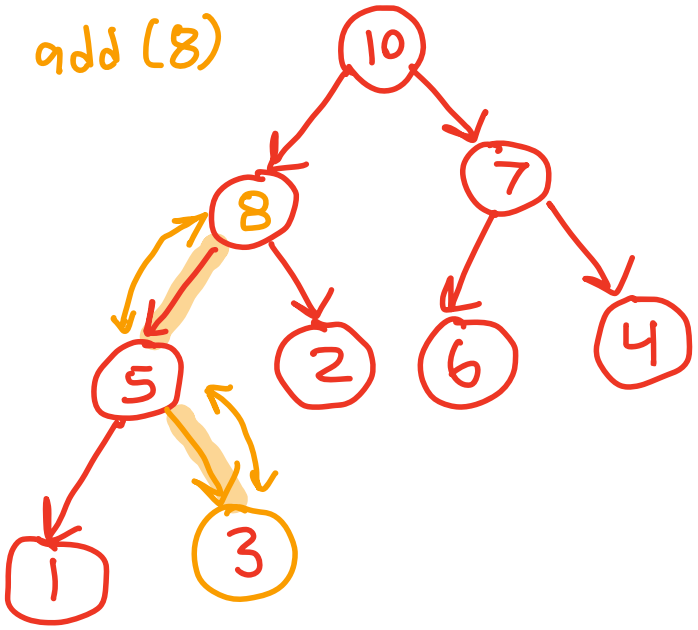
Key Ideas:

- Generic type bound
- ArrayList backing storage
- parent(), left(), right() static helpers
- invariantSatisfied()
- size() and peek()

add()ing to a Max Heap

Need to restore shape + order invariants

- array representation makes shape free, just append new element at end



- New element added to bottom level
- May violate order invariant if it's too big
- Repeatedly swap with (smaller) parent until order invariant restored

"bubble up"

The bubbleUp() Method

Exercise: Work with a partner to write the bubbleUp() method *iteratively*.

```
/** Swaps the entries at indices `i` and `j` in this heap. Requires `0 <= i,j < heap.size()`. */
private void swap(int i, int j) { ... }
/** Returns the index of the parent of the heap entry at index `i`. Requires `i > 0`, */
private static int parent(int i) { ... }

/** Relocates the element initially at index `i` to restore the heap invariant. */
private void bubbleUp(int i) {
    while (i > 0 && heap.get(i).compareTo(heap.get(parent(i))) > 0) {
        swap(i, parent(i));
        i = parent(i);
    }
}
```

Poll Everywhere

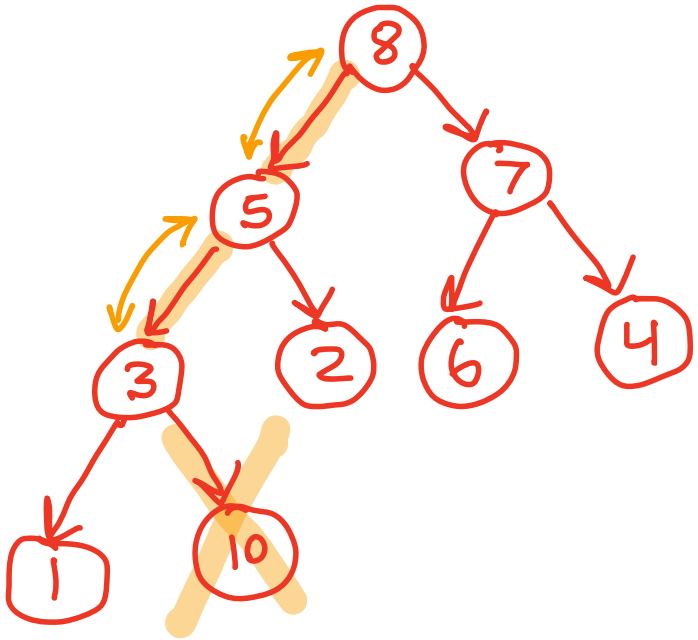
PollEv.com/javabear text **javabear** to **22333**

What is the loop guard of your `bubbleUp()` method?

```
/** Swaps the entries at indices `i` and `j` in this heap. Requires `0 <= i,j < heap.size()`. */
private void swap(int i, int j) { ... }
/** Returns the index of the parent of the heap entry at index `i`. Requires `i > 0`, */
private static int parent(int i) { ... }

/** Relocates the element initially at index `i` to restore the heap invariant. */
private void bubbleUp(int i) {



}
```

Max Heap `remove()`

Want to remove tree root without disrupting entire tree (shift can catastrophically break order invariant



- Shape invariant tells us which node must be removed
- Swap that element into root (breaking order invariant)
- Restore order invariant by "bubbling down" new root
  - ↕ repeatedly swap with larger child

Coding Demo: `remove()` and `bubbleDown()`

Max Heap Complexity Analysis

space: O(1)

size(): O(1)

peek(): O(1)

add(): O(logN)

remove(): O(logN)

Work dominated by bubbleUp() and bubbleDown() which limit their attention to one "branch" of tree and perform O(1) work per level

shape invariant ensures balanced tree, giving O(height) = O(logN) runtime

Coding Demo: MaxHeapPriorityQueue

Key Ideas:

- Comparable Entry class
  - Define compareTo() using priorities
- Composition relationship with MaxHeap<Entry>