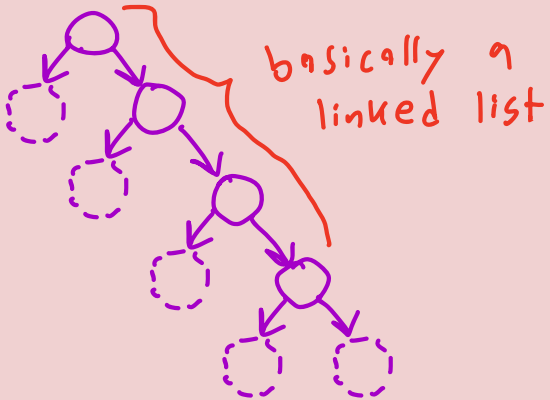
Poll Everywhere

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

All BST operations have a worst-case $O(H)$ runtime, where H is the tree's height. What's the tightest runtime bound in terms of N = the tree's size?

H can be as large as N



upper bound

- $O(\log N)$ **(A)**
- $O(\sqrt{N})$ **(B)**
- $O(N)$ **(C)**
- $O(N \log N)$ **(D)**

BST Complexity Analysis

Every BST operation involves one find() call, up to one successorDescendant() call, and O(1) other work

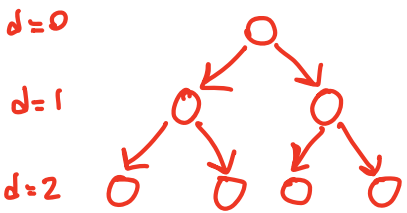
Helper methods both do O(1) work per level so run in O(H) time
↖ height

⇒ All BST operations have worst case O(H) runtimes

Our recursive definitions require O(H) space for call frames, but iterative variants use O(1) space

Balanced BSTs

For a fixed height H, at most how many nodes can a binary tree have?



$$N \leq \sum_{d=0}^{H} 2^d = 2^{H+1} - 1 \quad \text{nodes.}$$

lower-bounded by

$$N = O(2^H) \Rightarrow H = \Omega(\log N)$$

Balanced binary trees have $H = O(\log N)$

Families of self-balancing BSTs maintain a balance invariant in $O(H) = O(\log N)$ time (take 3110 for more info)



Lecture 18: Heaps and Priority Queues

CS 2110

October 27, 2026

Today's Learning Outcomes

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

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

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

72. 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 of different urgencies.

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

Triage = prioritize the 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", `PriorityQueue`s support four operations:

1. **Add another element**

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

2. **Remove/return "next" element**
   (ADT determines which is "next")

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

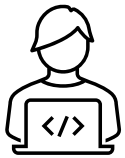
3. **Return "next" element**
   (without removing it)

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

4. **Check if there are more elements**

```
boolean isEmpty();
```

Approach 1: Compose with `List`



- Auxiliary `Entry` type to associate priorities with elements
- `add()` at the most efficient index
- Perform linear scan to find highest priority element

Runtime Complexities:

`add()`: O(1) (amortized)

space: O(N)

`remove()`: O(N)

`peek()`: O(N)

How can we avoid scanning all elements each time?

`isEmpty()`: O(1)

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

memory shift is O(N)

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

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

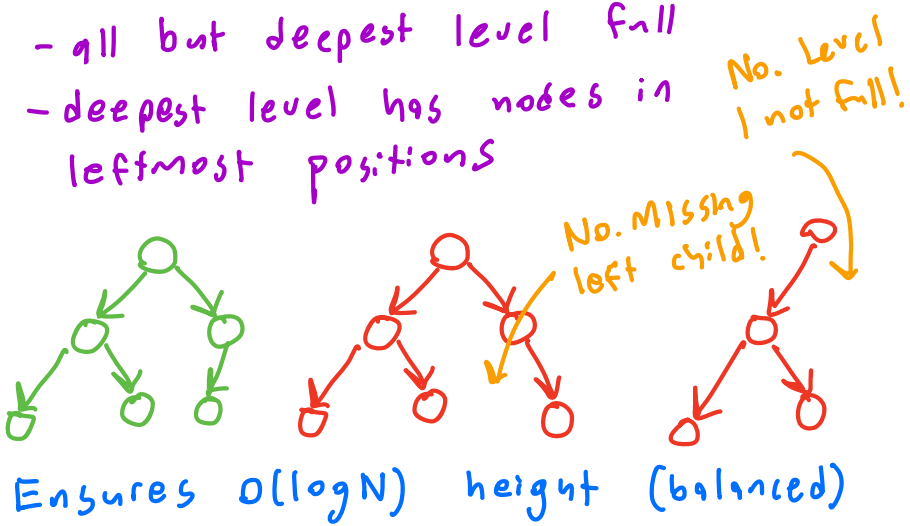
Yes. Heaps are a data structure that does this!

Max Heaps

= Binary trees with `Comparable` elements and two additional invariants:

1. Shape Invariant

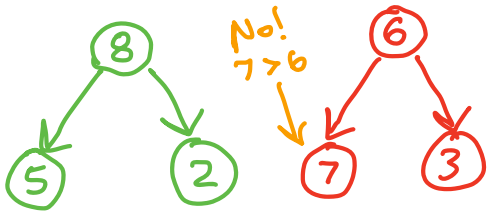
- all but deepest level full
- deepest level has nodes in leftmost positions



Ensures O(log N) height (balanced)

2. Heap Order Invariant

- every (non-root) node is "≤" its parent



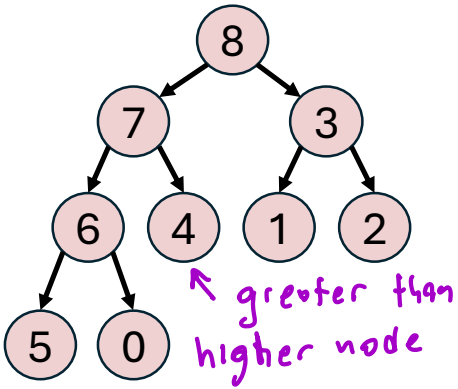
Ensures largest element at tree root.

Poll Everywhere

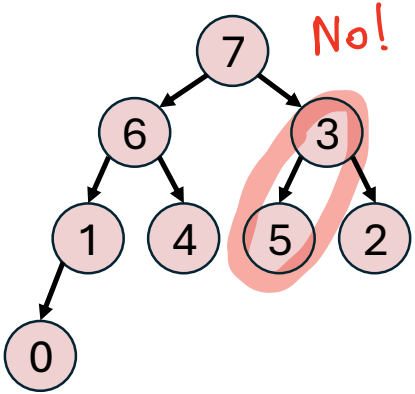
PollEv.com/javabear text **javabear** to 22333

Which of the following is a valid Max Heap?

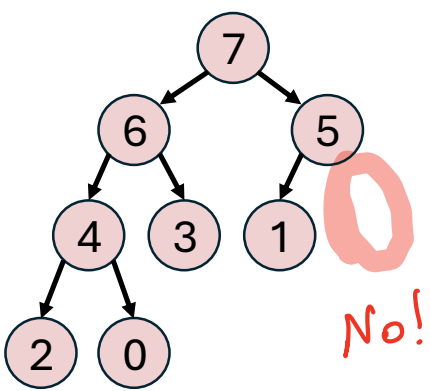
(A)



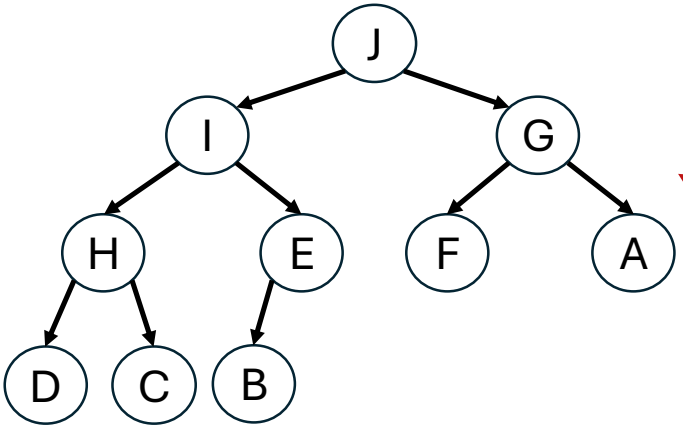
(B)



(C)



Array Representation



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

Unambiguously reconstruct tree level by level

J	I	G	H	E	F	A	D	C	B
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

For a node at array index `i`:

Left child at $2i+1$ Right child at $2i+2$ Parent at $\left\lfloor \frac{i-1}{2} \right\rfloor$



Coding Demo: `MaxHeap` State



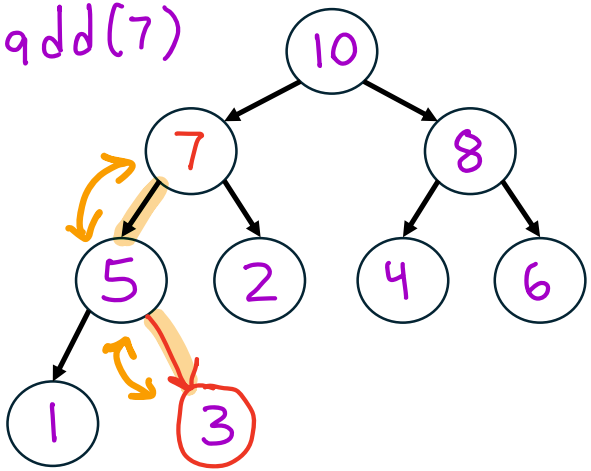
Key Ideas:

- Generic Type Bound
- `ArrayList` backing storage
- `parent()`, `left()`, and `right()` `static` helper methods
- `invariantSatisfied()`
- `size()` and `peek()`

add()ing to a Max Heap

Simplest approach: `add()` to the backing `ArrayList`

Issue: This may violate the heap order invariant



- New element appears in leftmost empty slot in bottom level
- If it's larger than its parent, swap them to "bubble up" new element to a valid position.

The bubbleUp() Method

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

MaxHeap.java

```
/** Swaps the `i`th and `j`th entries 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?

MaxHeap.java

```java
/** Swaps the `i`th and `j`th entries 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) {







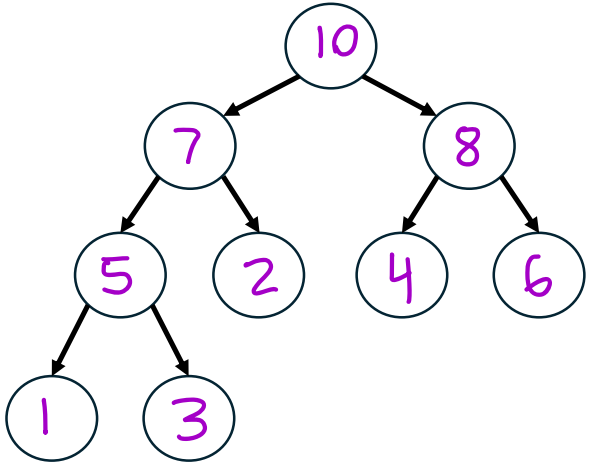


}
```

Max Heap `remove()`

We need to remove/return the first element of the backing `ArrayList`

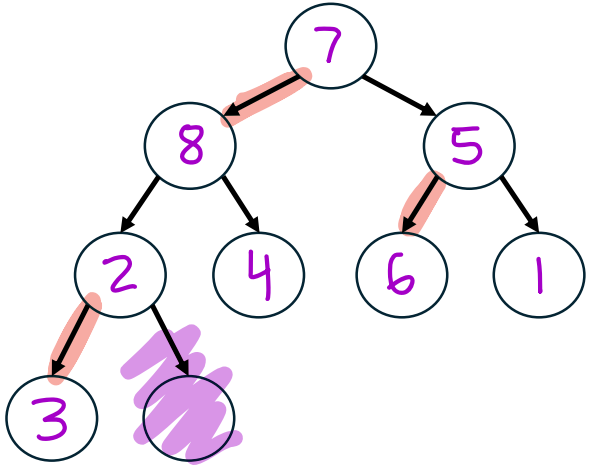
Issue: Doing this with a `remove(0)` call can "catastrophically break the heap order invariant



Max Heap `remove()`

We need to remove/return the first element of the backing `ArrayList`

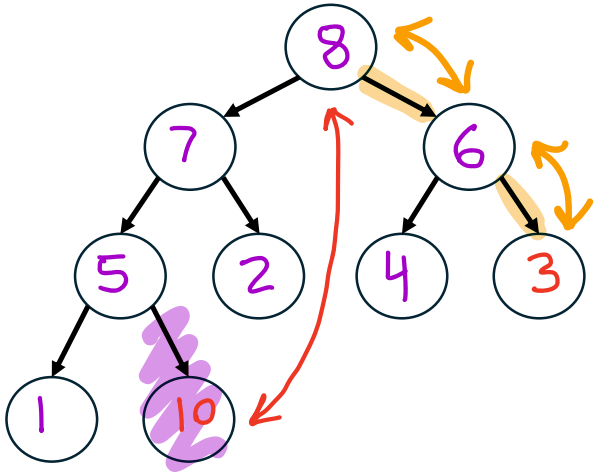
Issue: Doing this with a `remove(0)` call can "catastrophically break the heap order invariant



Max Heap `remove()`

We need to remove/return the first element of the backing `ArrayList`

Issue: Doing this with a `remove(0)` call can "catastrophically break the heap order invariant



Idea: We want remove() to get rid of 3's node and 10's value in tree, so swap them

New root element may be smaller than its children, so "bubble it down"

* swap() with larger child



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



Max Heap Complexity Analysis

Space Complexity: O(N), or O(1) space per element

Worst-case Runtime Complexity:

size(): O(1)

peek(): O(1)

delegate to ArrayList methods

add(): O(logN)*

remove(): O(logN)

work dominated by bubbleUp() and bubbleDown(), which each do O(1) work per level of tree

shape invariant ensures balanced tree

so O(H) = O(logN) runtime

* This is an amortized guarantee since add() in an ArrayList has an O(N) worst-case runtime.



Coding Demo: MaxHeapPriorityQueue



Key Ideas:

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