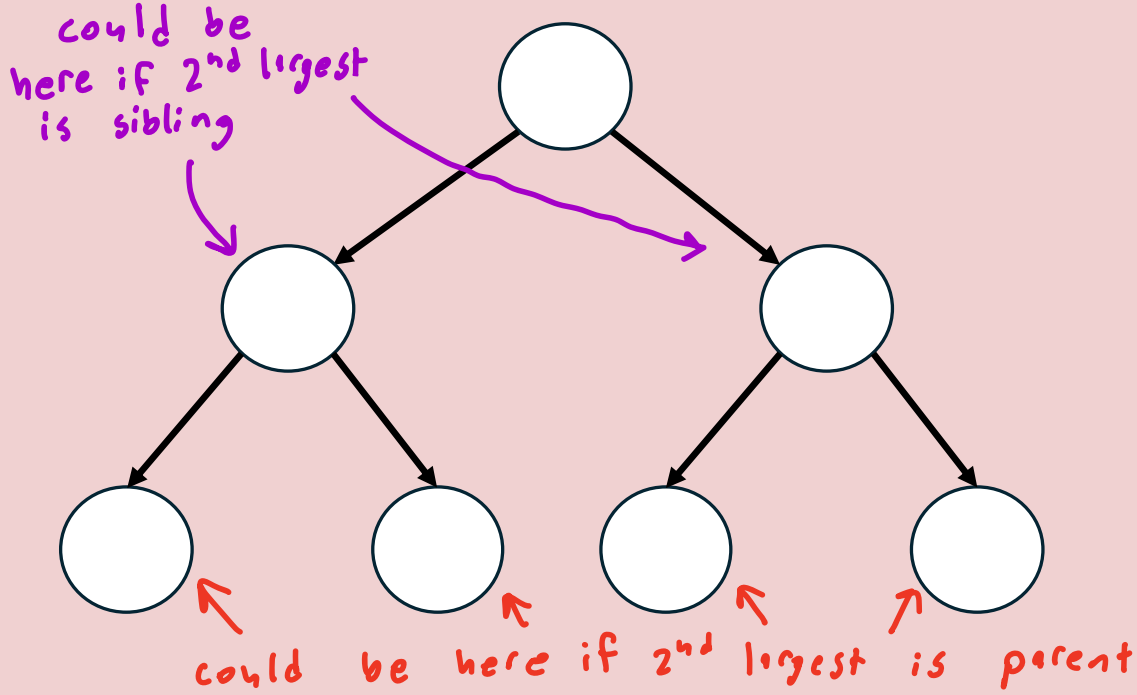
Poll Everywhere

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

At how many different indices can the *third largest* element in a Max Heap with 7 elements be located?



2	(A)
3	(B)
4	(C)
6	(D)

Lecture 19: Sets and Maps

CS 2110

March 26, 2026

Today's Learning Outcomes

79. Describe the operations of the `Set` and `Map` ADTs.

80. Write programs utilizing `Sets` and `Maps`.

81. Compare the efficiency of `Set` implementations backed by arrays, sorted arrays, and (balanced) binary search trees.

82. Use functional interfaces to implement operations on data structures.

The Set<T> ADT

A set is an unordered collection of distinct elements

- If we try to add an element to a set that is already there it should fail.

```
boolean   add(T elem)
boolean   remove(T elem)
boolean   contains(T elem)
int       size()
Iterator<T>   iterator()
```

returns whether add was successful
* has return value and side effect

returns false if elem wasn't there

Set<T> <: Iterable<T>

no guarantee on the order of this iteration

Approach 1: Compose with `List`

add(): Place elem at "convenient" end of list

O(1)? No! Need to check if elem already there

find() helper method performs this O(N) scan

remove() and contains() also use find() and run in O(N) time

size() and iterator() delegate to list: O(1)

How can we avoid O(N) scan? Order the elements.

Approach 2: Compose with sorted `ArrayList`

* Requires elements to be Comparable (ooh... generic type bound)

find() becomes O(logN) using a binary search

contains(): O(logN)

add(): still O(N) because of memory shift

remove(): still O(N) because of memory shift

size() and iterator() delegate to list: O(1)

Poll Everywhere

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

A colleague claims that they can reduce the runtime of `add()` and `remove()` by using a sorted (doubly) linked chain, since splicing nodes in and out is an O(1) operation.

Is this colleague correct?

Binary search doesn't work in a linked list because we lose random access guarantee.
O(N) linear scan is optimal.

Yes **(A)**	No **(B)**

Approach 3: Compose with a BST

The BST order invariant makes find() (as well as add(), remove(), and contains()) an O(H) operation.

H: height

Unbalanced BST: O(H) = O(N), so no benefit over list

Self-balancing BST: O(H) = O(logN), this becomes best Set implementation (so far...)

Strategy of Java's TreeSet class.

Additional Set Operations: Union

The union of two sets S and T is the set containing all elements that belong to either S or T (or both).

$S = \{1, 2, 3, 4, 5\}$ ← math notation for sets, not an array literal

$T = \{2, 4, 6, 8\}$

$S \cup T = \{1, 2, 3, 4, 5, 6, 8\}$

Coding Demo: Two Union Definitions

Additional Set Operations: Restriction

A predicate maps each element of a type to a boolean value (recall discussion 7).

Java has Predicate<T> functional interface with method

```
boolean test(T elem)
```

When we restrict a set on a predicate, it produces the subset of its elements that the predicate maps to true.

$S = \{1, 2, 3, 4, 5\}$ S.restrict(isEven) = $\{2, 4\}$

```
Predicate<Integer> isEven = x -> x % 2 == 0;
```

Poll Everywhere

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

The **intersection** of two sets is the set containing all elements common to both sets.

What lambda expression can we pass into `restrict()` to complete this intersection() definition?

```
/** Returns the subset of this set whose elements satisfy the given predicate. */
public Set<T> restrict(Predicate<T> pred) { ... }

/** Returns a new set containing the intersection of this set and `other`. */
public Set<T> intersection(Set<T> other) {
  return restrict(x -> other.contains(x) );
}
```

or special method reference syntax

other::contains

Maps

Unordered collections of (key, value) pairs called entries.

Each key is associated with one value.

(Python's dictionary type is a map)

When we use a map, we typically search for the entry with a given key so we can obtain the associated value

- look up definition of word in dictionary
- look up grades using student ID
- look up nutrition info given food name

Choosing Keys and Values

Keys and values play different roles in a Map

Keys

- used for lookup (searching the structure)
- small, immutable

Values

- store lots of data about key
- can be larger, richer, mutable types

Figuring out which is which:

"Every key is paired with exactly one value."

"I want to use key to look up value."

* Sometimes, we'll want to have Maps in both directions. (Discussion 10)

The Map<K, V> ADT

Generic on both key type K and value type V

Modifying Entries:

- void put(K key, V value) * overwrites old value associated with key
- V remove(K key)

Querying:

- V get(K key)
- boolean containsKey(K key)
- int size()

Iterating:

- Set<K> keySet()

Practice with Map Operations

Given a `Map` associating the names of famous scientists (`String`) with their birthdays (`LocalDate`), write a method to print the scientists with the same birthday as you.

Map
put(K key, V value): void
containsKey(K key): boolean
get(K key): V
remove(K key): V
keySet(): Set<K>
size(): int

```
/** Prints the names in `map` of scientists who were born on
 *  the same date (month and day) as `bday`. */
void sharesBDay (Map<String, LocalDate> map, LocalDate bday) {
    for ( String  sci : map.keySet()) {
      LocalDate d = map.get(sci);
      if(*) { System.out.println(sci); }
    }
}
```

```
d.getMonth() == bday.getMonth && d.getDayOfMonth() == bday.getDayOfMonth()
```

Coding Demo: Implementing Maps

Like Set, but with one level of indirection

- Auxiliary Entry type (stored in underlying collection)
- Use Key to search collection
- Use entire entry to do what's required when we get to the right place

Dynamic Priority Queues

- Sometimes, we want to update the priority of an element while it is in a priority queue

  Ex. In Dijkstra's algorithm (Lec 23) to update best known node distance upon path discovery

- Heaps (alone) don't give us a good way to do this since searching for an entry in heap is inefficient

Idea: Use a Map to associate each priority queue entry with its heap index.

For a good Map implementation (Lec 20) this gives us an efficient way to update priorities (AB)

Have a Nice Spring Break!!!