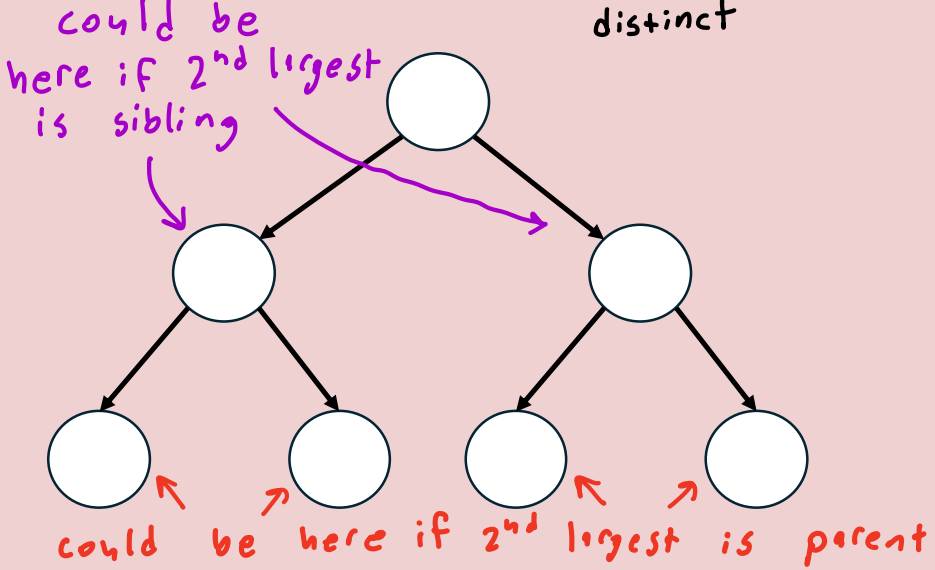
Poll Everywhere

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

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



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



Lecture 19: Sets and Maps

CS 2110

October 29, 2026

Today's Learning Outcomes

73. Describe the operations of the Set and Map ADTs, and utilize them in programs.

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

57. Identify and define functional interfaces and implement them using lambda expressions.

The Set<T> ADT

A **set** is an *unordered* collection of *distinct* elements.

The Set<T> ADT is used to model a set of elements with type T.

```
boolean   add(T elem)
boolean   remove(T elem)

boolean   contains(T elem)
int       size()

Iterator<T>   iterator()
```

Attempt to update collection and return whether this was successful

These methods have both return value and side-effect

Set<T> <: Iterable<T>

There's no guarantee of the iteration order

Approach 1: Compose with `List`

This approach allows any type T to be stored. Since the elements may not be `Comparable`, we can't impose invariants on the element order.

`add()`: Place element at "convenient" end of list

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

find() helper method does O(N) scan and returns index of match

`remove()`:
`contains()`: } O(N), dominated by find() call

How can we avoid O(N) scan?

`size()`:
`iterator()`: } O(1), delegate to list

* sort elements

Approach 2: Compose with sorted `ArrayList`

For "sorted" to make sense, we must impose the generic type bound `T extends Comparable<T>`.

`contains()`: O(log N) because find() becomes a binary search

`add()`:
`remove()`:
} still O(N), maintaining sorted order requires memory shifting

`size()`:
`iterator()`:
} O(1), delegate to list

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

Again, we need the generic type bound `T extends Comparable<T>`.

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(log N), 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 that contains 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

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

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

Let's add a `union()` instance method to our `Set` implementations:

```
public Set<T> union(Set<T> other) { ... }
```

Coding Demo: Three Union Definitions

N = this.size()
M = other.size()

1: Naive iterative add()

2: Smarter iterative add()

3: Incorporating sorting

In ListSet:

$$\sum_{i=0}^{N+M} O(i) = O((N+M)^2)$$

$$O(N) + M \cdot O(N) + O(M+N) = O(NM)$$

$O(N)$: copy this

$M \cdot O(N)$: this.contains()

$O(M+N)$: amortized add()s

Poll Everywhere

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

What is the tightest worst-case runtime we can achieve for computing the `union()` of two `SortedListSets` in terms of N = `this.size()` and M = `other.size()`?

This is almost identical to a deduplicating merge.

$O(\min(N, M))$	**(A)**	$O(NM)$	**(C)**
$O(N + M)$	**(B)**	$O(N^2 + M^2)$	**(D)**

Additional Set Operations: Restriction

A **predicate** maps each element of a type T to a `boolean` value.

Recall: Java's `Predicate<T>` functional interface with method

```
public 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;
```



Coding Demo: Set Restriction



* This is a good review of functional interface syntax.*

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(elem -> other.contains(elem));
}
```

or method reference syntax

other::contains

Maps

A **Map** is an unordered collection of (**key**, **value**) pairs called its **entries**.

Each key in a map is associated with one value (i.e., belongs to one entry).

(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 such as other collections (Dis 10)

To figure out which is which:

"Every Key is paired with exactly one value."

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

The Map<K, V> ADT

Generic on both key type K and value type V

Modifying Entries:

```
void put(K key, V value)
V remove(K key)
```

- adds new entry if Key not in Map
- overwrites old entry if key in Map

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 all the scientists with a particular birthday.

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 of the scientists in `map` who were
 *  born on the same date (month and day) as `bday`. */
void haveBDay(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 the collection
- Use entire `Entry` to meet the method post-condition once we get to the right place

Dynamic Priority Queues

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

> E.g., In Dijkstra's algorithm (Lecture 23), we'll update the best-known distance to a node when we discover a new incoming path to it.

Heaps (alone) don't give us a good way to do this, since searching in a 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 (A5)