Reminders

- Prelim 1 is one week from Thursday (March 12)
  - Fill out conflict survey by this Thursday
  - More exam info (+ practice exam) later today on Ed
  - TA review session Sunday afternoon
- A5 due tomorrow
- Read through grading feedback on assignments

Lecture 12: Collections and Generics

CS 2110

March 3, 2026

Today's Learning Outcomes

52. Describe the differences between *data structures* and *abstract data types*.

53. Implement a generic class or method with one or more generic type parameters. Use generic classes in client code.

54. Describe the semantics of *auto-boxing* and *auto-unboxing* and identify where they happen in a code snippet.

56. Compare and contrast the behaviors of ordinary Java arrays and dynamic array types such as Java's `ArrayList`.

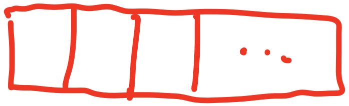
Arrays as Collections

A collection is a type that groups together objects of another type.

First Example: Arrays

String[] is a collection of Strings

state: contiguous "block" of memory cells



behaviors: - query length O(1)

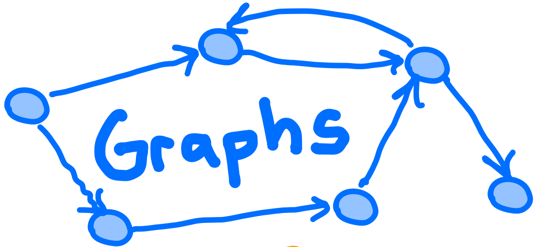
- read and write cell values O(1) ~ does not depend on a.length or i
  using index notation a[i]

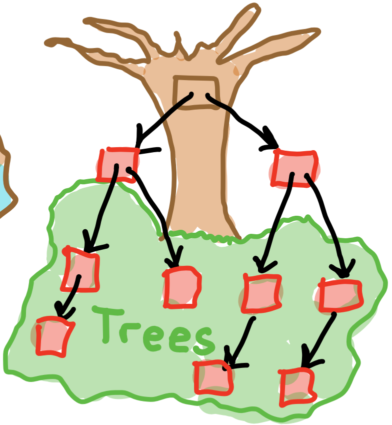
"Random Access Guarantee" of arrays

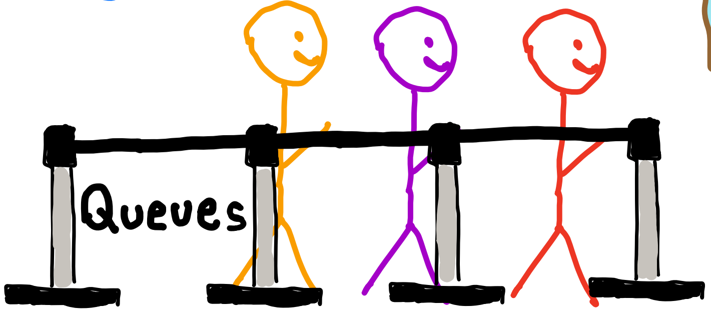
runtime guarantees

Other Collections

Each has its own
- structure/state representation
- behaviors / invariants
- performance guarantees







Abstract Data Types vs. Data Structures

Client and implementer interact with collections in different ways. (abstraction barrier)

Client:

- What operations are supported

How is the collection

- modified (add, remove, etc.)
- queried

- (runtime guarantees for behaviors)

ADT models behaviors

Implementer:

How do we

- represent data in memory
- define promised operations
- meet runtime guarantees

Data Structures answer these "behind the scenes" ?s

Poll Everywhere

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

Which of the following is the most natural way to model an Abstract Data Type in Java?

guarantee behaviors, don't fix state

Abstract Class **(A)**

(Concrete) Class **(B)**

Record Class **(C)**

Interface [✓ **(D)**]

The List ADT

A list is an <u>ordered</u> collection that grows to accommodate an <u>arbitrary</u> number of elements. Its elements are accessible by <u>index</u>.

Behaviors:

<u>Accessing</u>

- size
- get (at index)
- contains (element)
- indexOf (element)

<u>Mutating</u>

- add (at end)
- insert (at index)
- set (particular index)
- remove (at index)
- delete (element)

Coding Demo: `StringList` Interface

Using StringList as a Client

Use the StringList methods to complete the definition of the following method:

```
/** Replaces all instances of the given `word` with
 *  "*****" in these `lyrics`. */
static void censor(StringList lyrics, String word) {





}
```

StringList
add(String elem): void
insert(int index, String elem): void
size(): int
get(int index): String
contains(String elem): boolean
indexOf(String elem): int
set(int index, String elem): void
remove(int index): String
delete(String elem): void

Poll Everywhere

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

Which `StringList` method(s) did you use?

```java
/** Replaces all instances of the given `word` with
 *  "*****" in these `lyrics`. */
static void censor(StringList lyrics, String word) {
    while (lyrics.contains(word)) {
        int i = lyrics.indexOF(word);
        lyrics.set(i, "****");
    }
}
```

StringList

add(String elem): void

insert(int index, String elem): void

size(): int

get(int index): String

contains(String elem): boolean

indexOf(String elem): int

set(int index, String elem): void

remove(int index): String

delete(String elem): void

Generic Classes

What if we want a list of something other than Strings?

- Accounts, Points, other lists?

We'd need to define a new ADT interface

- lots of duplicated code ☹

Instead: Can we write one ADT to handle all data?

A new type of polymorphism (parametric)

Idea: Just like how we can parameterize a method on variables, we can parameterize a class/interface on a type called a generic type.

New angle-bracket < > Syntax

Coding Demo: Generic `CS2110List`

Using Generic Types

Generic Types can take the place of a type name almost everywhere in a class.

- Fields

```
private T elem;
private T[] storage;
```

- Local variables
- Parameters

```
public void add(T elem) { }
```

- Return types

```
public T get(int index) { }
```

cannot:

- invoke methods on objects of type T (for now)

  can't enforce CTRR

- construct new T objects

  don't know constructor args

- contruct new T[] s

  need weird hack...

Brief Aside: Auto-(un)boxing

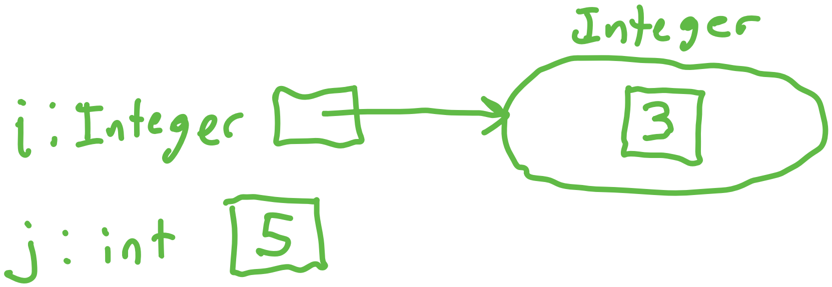
Generic type parameters can only be assigned reference types, not primitive types.

What if we want a list of ints?

Solution: Java has wrapper classes for each primitive type. E.g. Boolean for boolean, Integer for int ...

Conversion between primitives and wrapper class objects happens automatically

```
Integer i = 3; // auto-boxing
  int j = i+2; // auto-unboxing
```



Poll Everywhere

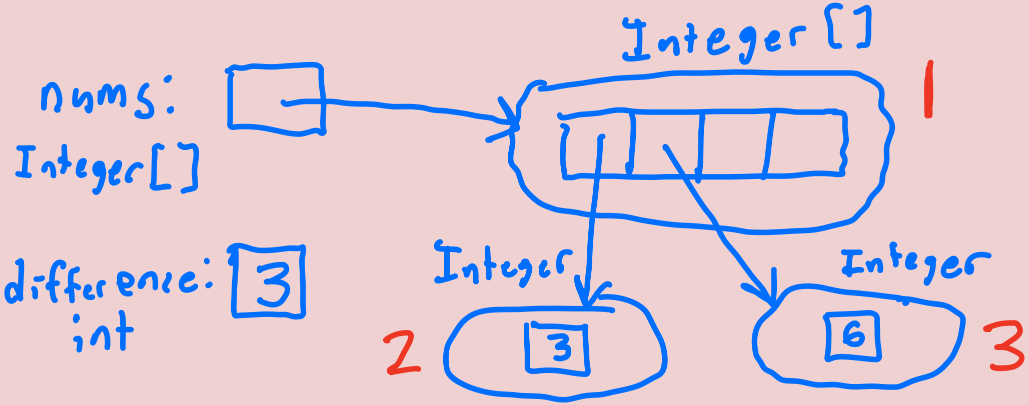
PollEv.com/javabear text **javabear** to 22333

How many objects are allocated on the heap when we execute the following code?

```
Integer[] nums = new Integer[4];
nums[0] = 3;
nums[1] = 6;
int difference = nums[1] – nums[0];
```

autoboxing

auto-unboxing



- 1 (A)
- 3 [✓ (B)]
- 4 (C)
- 5 (D)

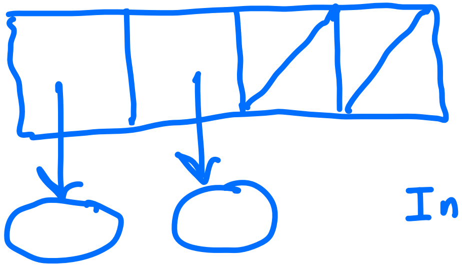
The Dynamic Array Data Structure

How can we use arrays to model a list?

Need to handle unbounded size carefully.

Idea: When array becomes full, replace with new array and copy entries over.

storage: T[]



storage.length = "capacity"

# of elements actually stored = "size"

Inv: First "size" entries hold elements, Last "capacity" - "size" entries are null

Resizing strategy: add()/insert() when "size" = "capacity" ⇒ double the capacity

Coding Demo: `DynamicArrayList` Design

invariantSatisfied() Methods

Data structures often rely on intricate class invariants to achieve good performance and ensure correctness.

Recall: Class invariant must hold at start/end of every public method call.

invariantSatisfied() methods are a good development tool "defensive programming against yourself"

packages up checks for entire class invariant into one boolean method we can assert.

call assertInv() before returning from mutating method.

* and remember to enable assertions!

Coding Demo: `DynamicArrayList` Methods

Space Complexity of DynamicArrayList

- size takes up $O(1)$ space
- storage includes $O(N)$ "full" cells and $O(N)$ empty cells (since it will always* be $\geq$ half full after resizing)
- don't count space of elements, since we didn't construct them

Overall: $\boxed{O(N)}$

Methods:

Most use $O(1)$ space

increaseCapacity() + add()/insert() allocate second $O(N)$ array during resizing copy

Time Complexity of DynamicArrayList

DynamicArrayList	
+ insert(int index, T elem): void	O(N), need to shift all elems when index=0
+ remove(int index): T	
+ size(): int	O(1)
+ get(int index): T	O(1) random access guarantee
+ set(int index, T elem): void	
+ contains(T elem): boolean	O(N) find() does linear search
+ indexOf(T elem): int	
+ delete(T elem): void	
+ add(T elem): void	O(N) because of resize

Poll Everywhere

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

What is the worst-case time complexity of this `censor()` definition, where N = `lyrics.size()`?

```
/** Replaces all instances of the given `word`
 *   with "****" in these `lyrics`. */
static void censor(StringList lyrics, String word) {
  while(lyrics.contains(word)) {
      int i = lyrics.indexOf(word);
      lyrics.set(i,"****");
  }
}
```

O(N) iterations

O(N) search

soon: iterators let us do this faster

$O(1)$ (A)

$O(N)$ (B)

$O(N^2)$ [✓ (C)]

$O(N^3)$ (D)

Amortized Time Complexity

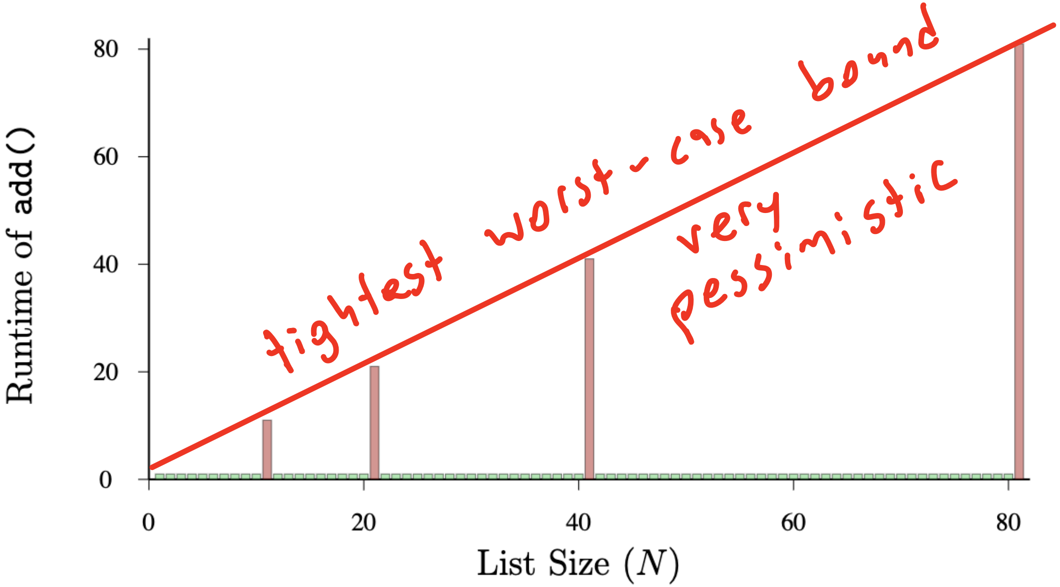
Let's think a bit more about runtime of add()

- Usually, just write to one array cell, update size
  O(1) operation

- Infrequently, resize and copy, O(N) operation

We'd like a notion of the "typical" runtime

$\underbrace{\text{typical}}_{\text{long-run average}}$



tightest worst-case bound

very pessimistic

Amortized Time Complexity

||

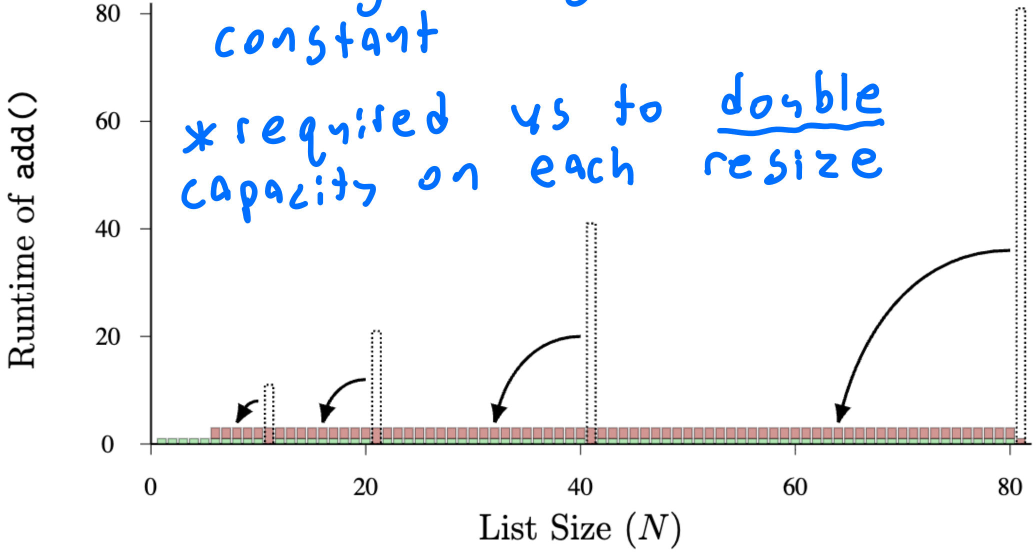
Total complexity of a sequence of method calls, divided by # of calls.

≈ average or expected performance

add()ing N elements to an empty DynamicArrayList requires O(N) total work, so add() has O(1) amortized runtim complexity.

average height of plot is constant

* required us to <u>double</u> capacity on each resize


Runtime of add()
List Size (N)