Lecture 2: Intro to ADTs

CSE 373: Data Structures and Algorithms

Basic Definitions

Data Structure

- A way of organizing and storing data
- Examples from CSE 14X/12X: arrays, linked lists, stacks, queues, trees

Algorithm

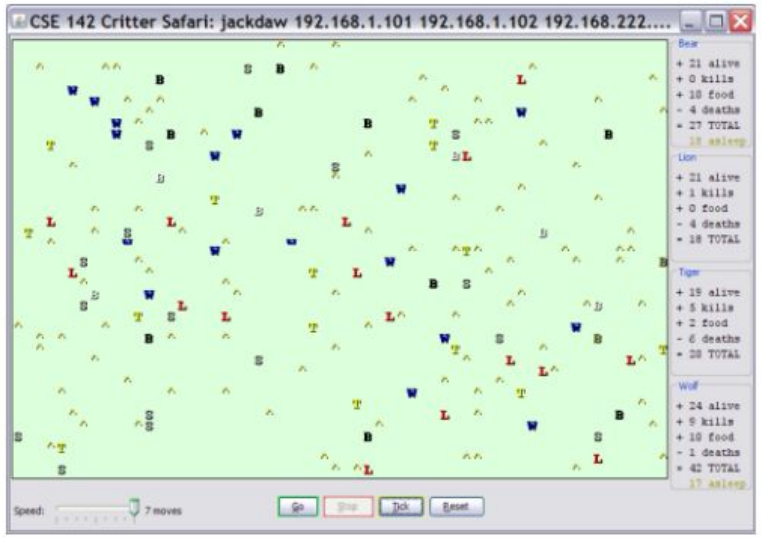
- A series of precise instructions to produce to a specific outcome
- Examples from CSE 14X/12X: binary search, merge sort, recursive backtracking

Review: Clients vs Objects

CLIENT CLASSES

A class that is executable, in Java this means it contains a Main method

```
public static void main(String[] args)
```



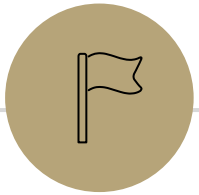
OBJECT CLASSES

A coded structure that contains data and behavior

Start with the data you want to hold, organize the things you want to enable users to do with that data

1. Ant

constructor	`public Ant(boolean walkSouth)`
color	red
eating behavior	always returns `true`
fighting behavior	always scratch
movement	if the Ant was constructed with a `walkSouth` value of `true`, then alternates between south and east in a zigzag (S, E, S, E, ...); otherwise, if the Ant was constructed with a `walkSouth` value of `false`, then alternates between north and east in a zigzag (N, E, N, E, ...)
toString	`"%"` (percent)



ADTs

List Case Study

Generics

Questions

Abstract Data Types (ADT)

Abstract Data Type

- A type of organization of data that we define through its behavior and operations
  - Defines the input and outputs, not the implementations

Invented in 1974 by Barbara Liskov

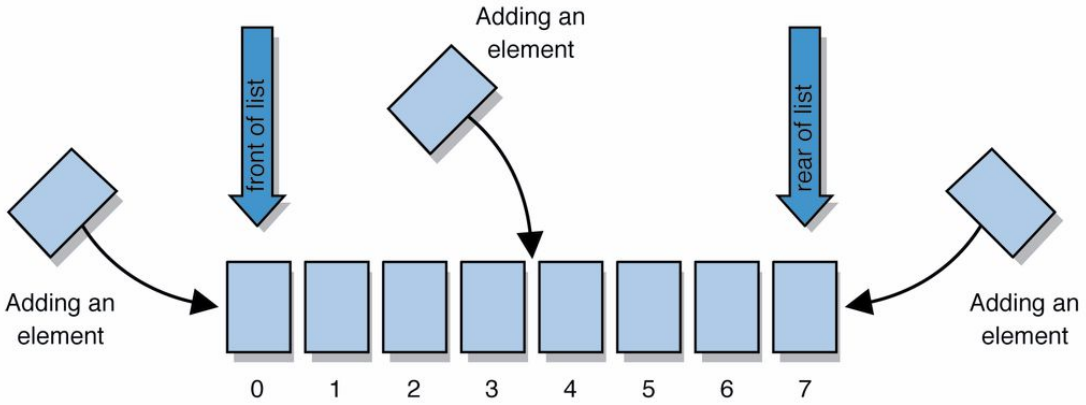
> *What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details. In the case of programming, the use which may be made of an abstraction is relevant; the way in which the abstraction is implemented is irrelevant. — Barbara Liskov*

Source Article

Abstract Data Types (ADT): List Example

Review: List – a collection storing an ordered sequence of elements

- each element is accessible by a 0-based index
- a list has a size (number of elements that have been added)
- elements can be added to the front, back, or elsewhere
- the ADT of a list can be implemented many ways through different Data Structures
  - in Java, a list can be represented as an ArrayList object



Review: Interfaces

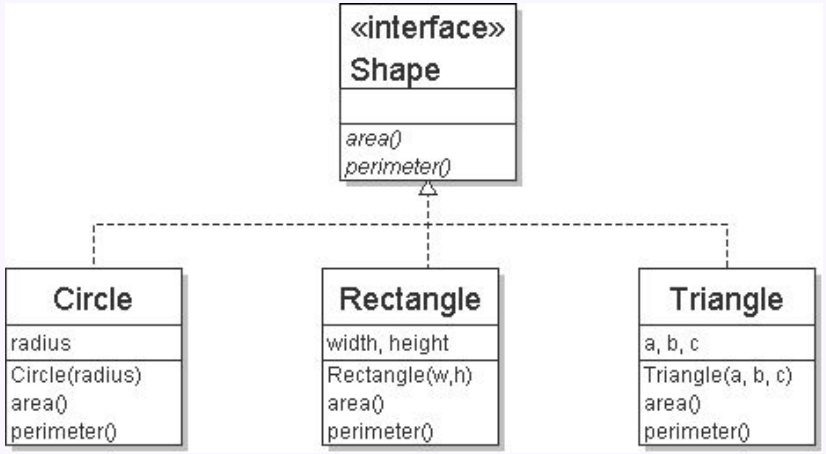
interface: a construct in Java that defines a set of methods that a class promises to implement

```
public interface name {
    public type name(type name, …, type name );
    public type name(type name, …, type name );
    …
}
```

- Interfaces give you an is-a relationship *without* code sharing.
  - A `Rectangle` object can be treated as a `Shape` but inherits no code.
- Analogous to non-programming idea of roles/certifications:
  - "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know **how** to compute my area and perimeter."
  - "I'm 'certified' as a CPA accountant. This assures you I know **how** to do taxes, audits, and consulting."

Example

```
// Describes features common to all
// shapes.

public interface Shape {
      public double area();
      public double perimeter();
}
```



Review: Interfaces: List Example

interface: a construct in Java that defines a set of methods that a class promises to implement

In terms of ADTs, interfaces help us make sure that our implementations of that ADT are doing what they need to

For example, we could define an interface for the ADT `List<E>` and any class that implements it must have implementations for all of the defined methods

```
// Describes features common to all lists.

public interface List<E> {
    public E get(int index);
    public void set(E element, int index);
    public void append(E element);
    public E remove(int index);
    …

    // many more methods
}
```

note: this is not how the `List<E>` interface actually looks, as in reality it extends another interface

Review: Java Collections

Java provides some implementations of ADTs for you!

ADTs	Data Structures
Lists	`List<Integer> a = new ArrayList<Integer>();`
Stacks	`Stack<Character> c = new Stack<Character>();`
Queues	`Queue<String> b = new LinkedList<String>();`
Maps	`Map<String, String> d = new TreeMap<String, String>();`

But some data structures you made from scratch… why?

Linked Lists – LinkedIntList was a collection of ListNode
Binary Search Trees – SearchTree was a collection of SearchTreeNodes

Full Definitions

Abstract Data Type (ADT)

- *A definition for expected operations and behavior*
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Describes what a collection does, not how it does it
- Can be expressed as an interface
- Examples: List, Map, Set

Data Structure

- *A way of organizing and storing related data points*
- An object that implements the functionality of a specified ADT
- Describes exactly how the collection will perform the required operations
- Examples: LinkedIntList, ArrayIntList

ADTs and Data Structures: (Loose) Analogy

Abstract Data Type (ADT)

Mode of Transportation
Must be able to move
Must be able to be steered

Data Structures

Car	**Airplane**	**Bike**
Tires	Engines/wings	Wheels
Steering wheel	Control column	Handlebars

ADTs and Data Structures: List Example

List – Abstract Data Type (ADT)

```
// Describes features common to all lists.

public interface List<E> {
    public E get(int index);
    public void set(E element, int index);
    public void append(E element);
    public E remove(int index);
    …
}
```

ArrayIntList – Data Structure

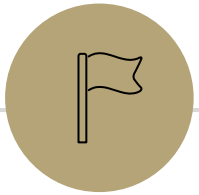
```
public class ArrayIntList extends List<E>{
    private int[] list;
    private int size;

    public ArrayIntList(){
        //initialize fields
    }

    public int get(int index){
        return list[index];
    }

    public void set(E element, int index){
        list[index] = element;
    }
    …
}
```

Full Definitions

- Abstract Data Type (ADT)
  - *A definition for expected operations and behavior*
  - A mathematical description of a collection with a set of supported operations and how they should behave when called upon
  - Describes what a collection does, not how it does it
  - Can be expressed as an interface
  - Examples: List, Map, Set
- Data Structure
  - *A way of organizing and storing related data points*
  - An object that implements the functionality of a specified ADT
  - Describes exactly how the collection will perform the required operations
  - Examples: LinkedIntList, ArrayIntList

ADTs we'll discuss this quarter

- List: an ordered sequence of elements
- Set: an unordered collection of elements
- Map: a collection of "keys" and associated "values"
- Stack: a sequence of elements that can only go in or out from one end
- Queue: a sequence of elements that go in one end and exit the other
- Priority Queue: a sequence of elements that is ordered by "priority"
- Graph: a collection of points/vertices and edges between points
- Disjoint Set: a collection of sets of elements with no overlap



Quick ADT Review

List Case Study

Generics

Questions

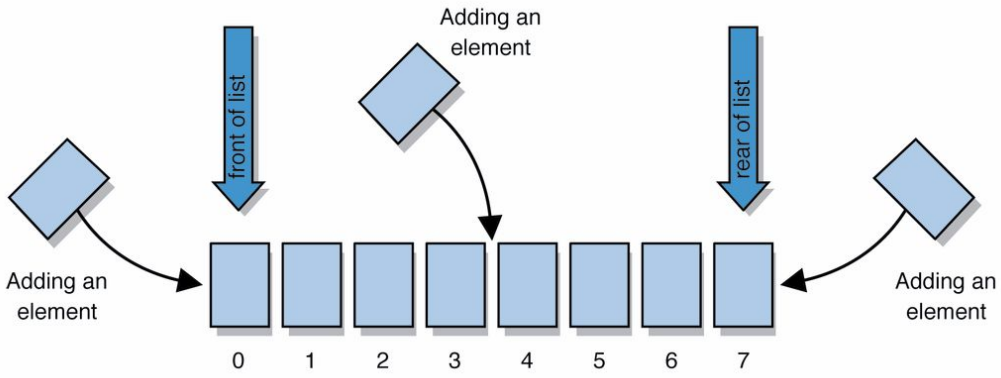
Case Study: The List ADT

list: a collection storing an ordered sequence of elements

- Each item is accessible by an index
- A list has a size defined as the number of elements in the list

Expected Behavior:

- **`get(index)`:** returns the item at the given index
- **`set(value, index)`:** sets the item at the given index to the given value
- **`append(value)`:** adds the given item to the end of the list
- **`insert(value, index)`:** insert the given item at the given index maintaining order
- **`delete(index)`:** removes the item at the given index maintaining order
- **`size()`:** returns the number of elements in the list




```
List<String> names = new ArrayList<>();
names.add("Anish");
names.add("Amanda");
names.add(0, "Brian");
```

Case Study: List Implementations

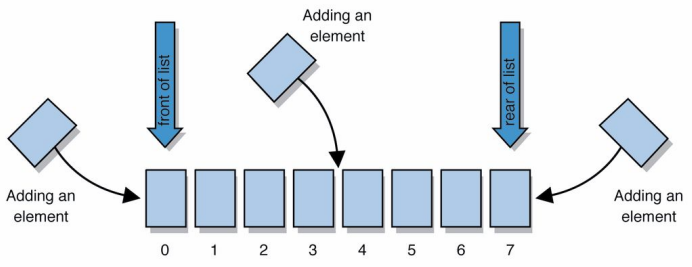
List ADT

state

Set of ordered items
Count of items

behavior

get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items



ArrayList

uses an Array as underlying storage

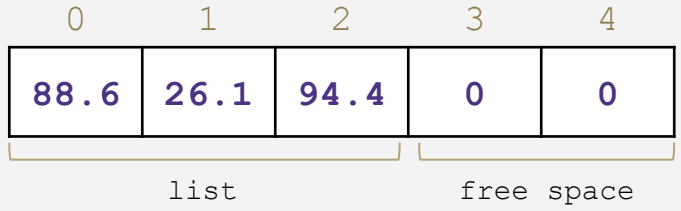
ArrayList<E>

state

data[]
size

behavior

get return data[index]
set data[index] = value
append data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size



LinkedList

uses nodes as underlying storage

LinkedList<E>

state

Node front
size

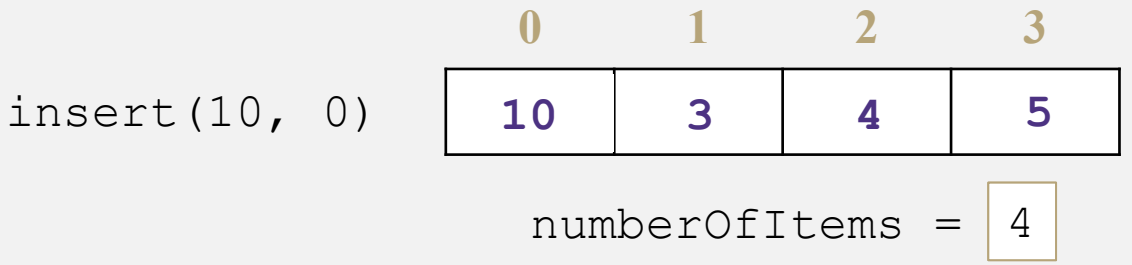
behavior

get loop until index, return node's value
set loop until index, update node's value
append create new node, update next of last node
insert create new node, loop until index, update next fields
delete loop until index, skip node
size return size

Implementing Insert

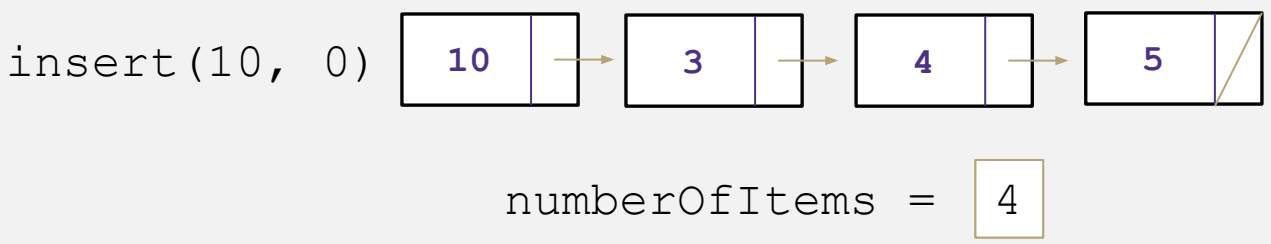
ArrayList<E>

`insert(element, index)` with shifting



LinkedList<E>

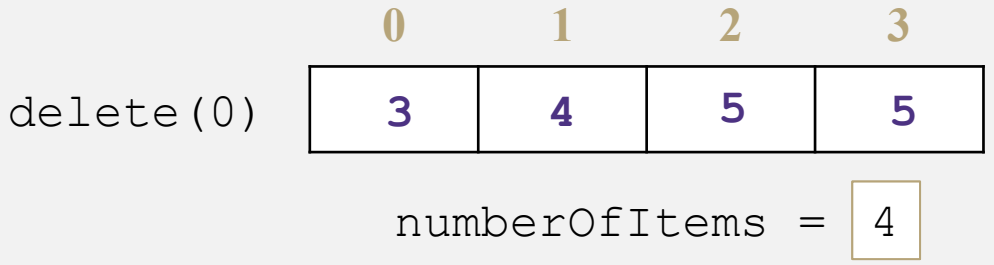
`insert(element, index)` with shifting



Implementing Delete

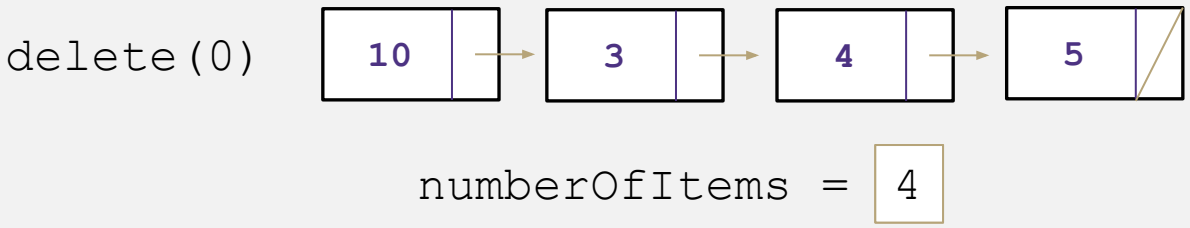
ArrayList<E>

`delete(index)` with shifting



LinkedList<E>

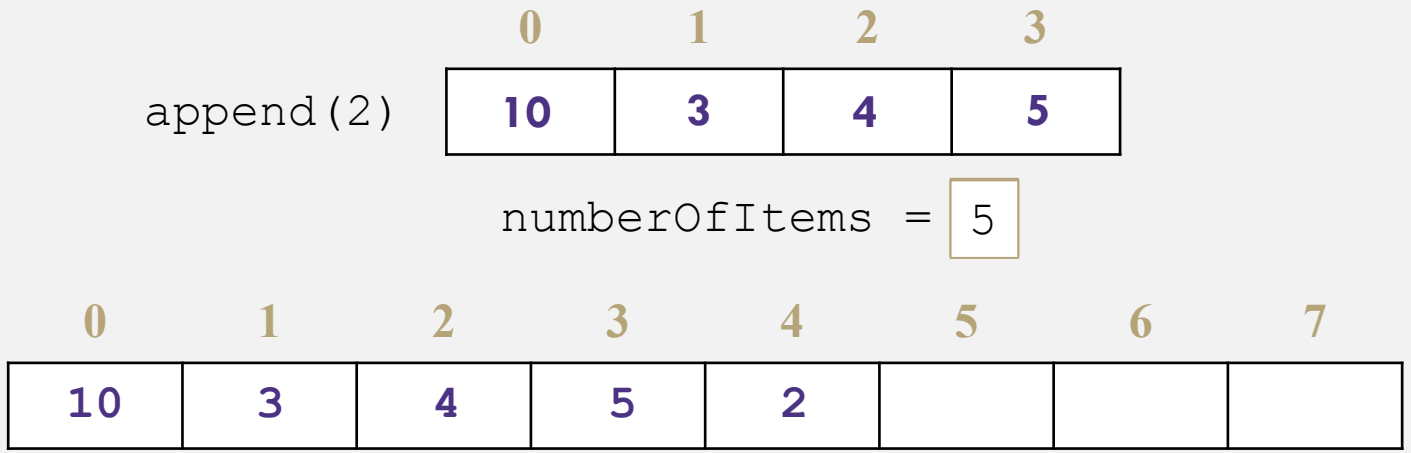
`delete(index)` with shifting



Implementing Append

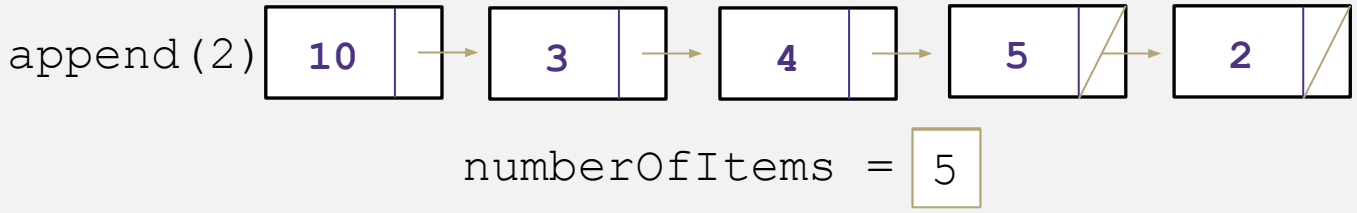
ArrayList<E>

`append(element)` with growth



LinkedList<E>

`append(element)` with growth

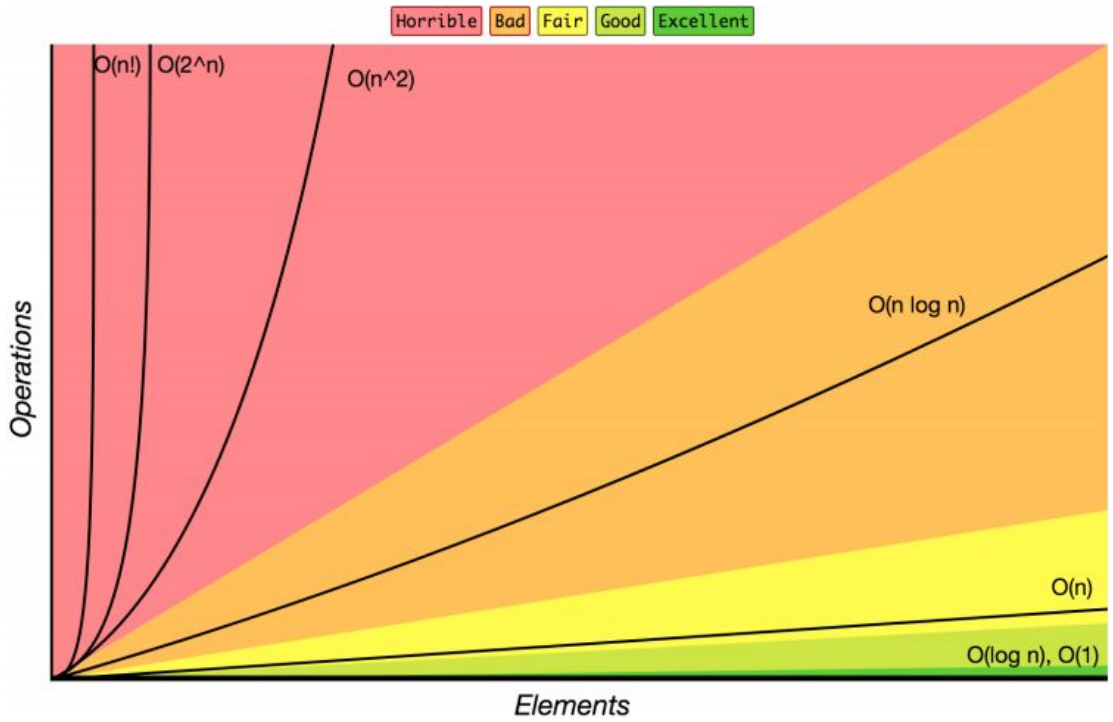


Review: Complexity Class

Note: You don't have to understand all of this right now – we'll dive into it soon.

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
…	…	…	…
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



List ADT tradeoffs

Last time: we used "slow" and "fast" to describe running times.
Let's be a little more precise.

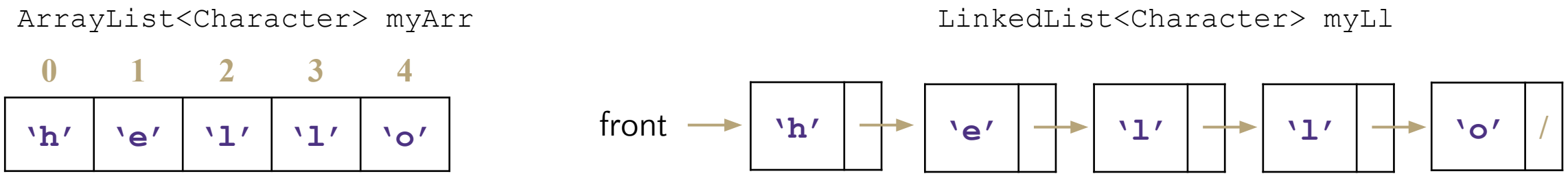
Recall these basic Big-O ideas from 12X: Suppose our list has N elements

- If a method takes a constant number of steps (like 23 or 5) its running time is O(1)
- If a method takes a linear number of steps (like 4N+3) its running time is O(N)

For ArrayLists and LinkedLists, what is the O() for each of these operations?

- Time needed to access Nth element
- Time needed to insert at end (what if the array is full?)

What are the memory tradeoffs for our two implementations?



List ADT tradeoffs

Time needed to access Nth element:

- ArrayList: O(1) constant time
- LinkedList: O(N) linear time

Time needed to insert at Nth element (if the array is full!)

- ArrayList: O(N) linear time
- LinkedList: O(N) linear time

Amount of space used overall/across all elements

- ArrayList: sometimes wasted space at end of array
- LinkedList: compact, one node for each entry

Amount of space used per element

- ArrayList: minimal, one element of array
- LinkedList: tiny bit extra, object with two fields

Design Decisions

For every ADT there are lots of different ways to implement them

Based on your situation you should consider:

- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

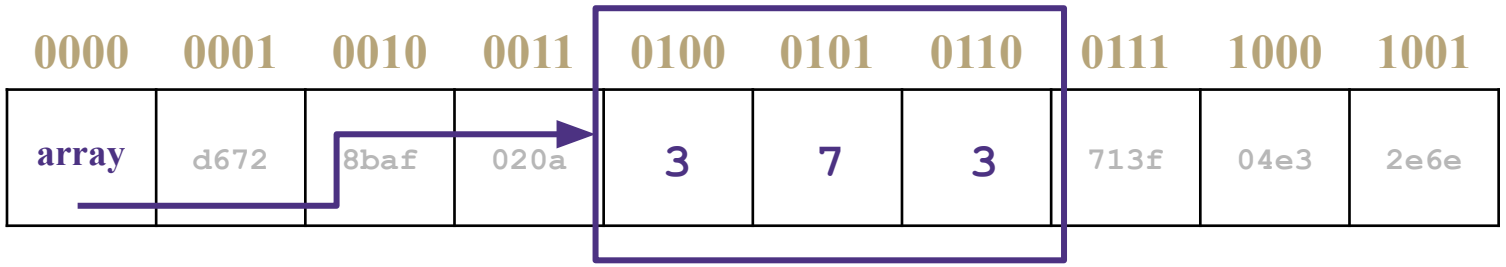
This class is all about implementing ADTs based on making the right design tradeoffs!

A common topic in interview questions!

A quick aside: Types of memory

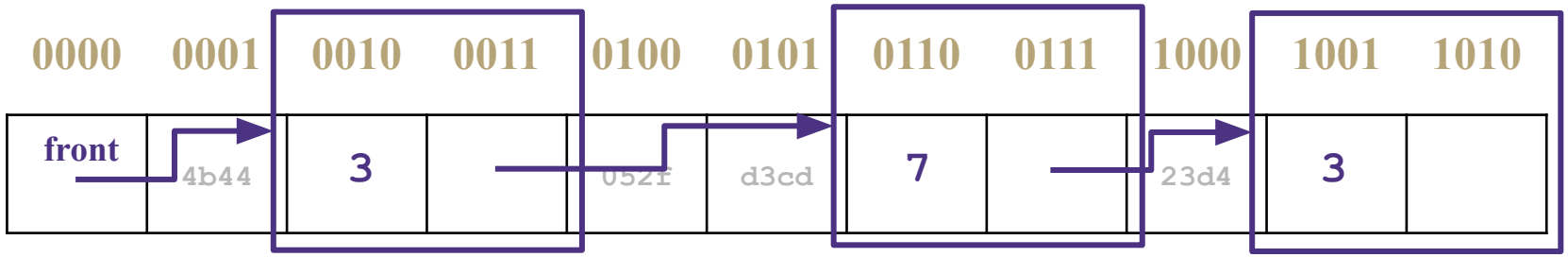
Arrays - **contiguous memory**: when the "new" keyword is used on an array the operating system sets aside a single, right-sized block of computer memory

```
int[] array = new int[3];
array[0] = 3;
array[1] = 7;
array[2] = 3;
```



Nodes- **non-contiguous memory**: when the "new" keyword is used on a single node the operating system sets aside enough space for that object at the next available memory location

```
Node front = new Node(3);
front.next = new Node(7);
front.next.next = new Node(3);
```



More on how memory impacts runtime later in this course...

Design Decisions

Situation #1: Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

Features to consider:

- add or remove songs from list
- change song order
- shuffle play

Why ArrayList?

- optimized element access makes shuffle more efficient
- accessing next element faster in contiguous memory

Why LinkedList?

- easier to reorder songs
- memory right sized for changes in size of playlist, shrinks if songs are removed

Design Decisions

Situation #2: Write a data structure that implements the List ADT that will be used to store the history of a bank customer's transactions.

Features to consider:

- adding a new transaction
- reviewing/retrieving transaction history

Why ArrayList?

- optimized element access makes reviewing based on order easier
- contiguous memory more efficient and less waste than usual array usage because no removals

Why LinkedList?

- if structured with front pointing to most recent transaction, addition of transactions constant time
- memory right sized for large variations in different account history size

Design Decisions

Situation #3: Write a data structure that implements the List ADT that will be used to store the order of students waiting to speak to a TA at a tutoring center

ArrayList – optimize for addition to back
LinkedList – optimize for removal from front

Real-World Scenarios: Lists

LinkedList

- Image viewer
  - Previous and next images are linked, hence can be accessed by next and previous button
- Dynamic memory allocation
  - We use linked list of free blocks
- Implementations of other ADTs such as Stacks, Queues, Graphs, etc.

ArrayList

- Maintaining Database Records
  - List of records you want to add / delete from and maintain your order after
- Implementations of other ADTs such as Stacks, Queues, Graphs, etc.