COSC 121
Computer Programming II

Data Structures

Lists, Stacks, Queues, and Priority Queues

Part 1/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

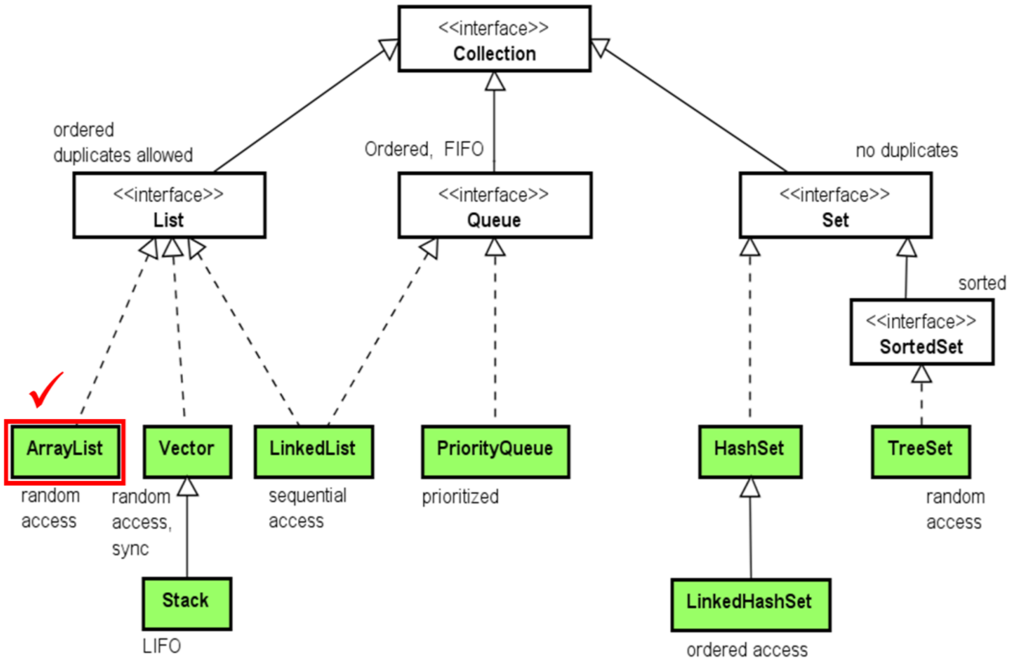
Previous Lecture

- Intro to the implementation of `ArrayLists`
- Iterating Over an `ArrayList`
- Useful Methods for Lists
  - `Arrays` and `Collections` classes
- Intro to Generics

Outline

Today:

- Collection Interface
- Lists:
  - ArrayList (again)
  - LinkedList
  - ArrayList vs LinkedList
    - Big-Oh Notation



Next lecture:

- More Lists: Vector / Stack
- Queues: Queue / PriorityQueue
- Interfaces: Comparable / Comparator

Previously…

Remember: Java Collections Framework

A ***data structure*** is a collection of data organized in some fashion.

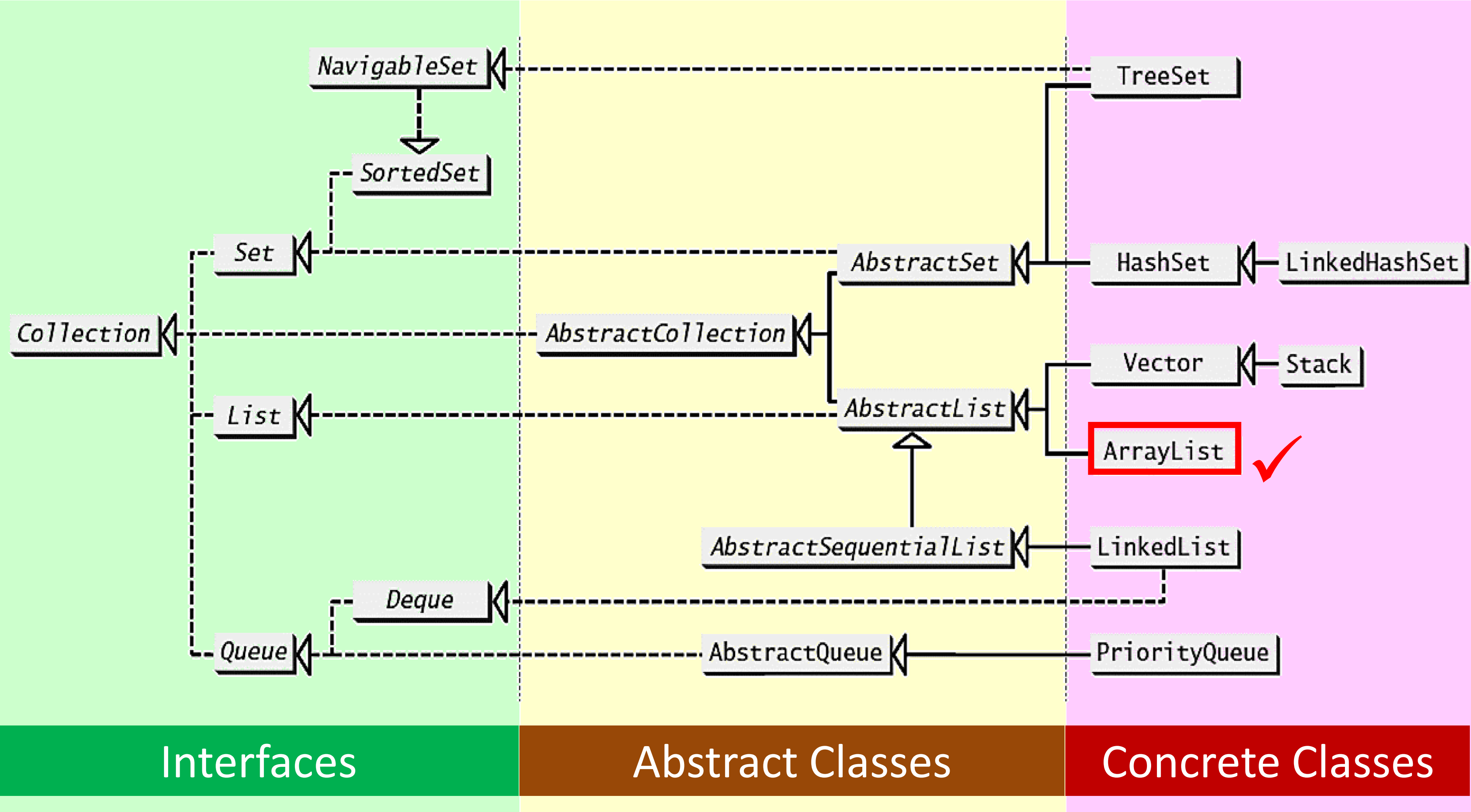
- It is a ***container object*** that stores other objects (data).
- To define a data structure is essentially to define a class.
  - uses data fields to store data and methods to support operations for accessing and manipulating the data.
- **COSC 222**

Java Collections Framework includes several data structures for storing and managing data (objects of any type).

It supports two types of containers:

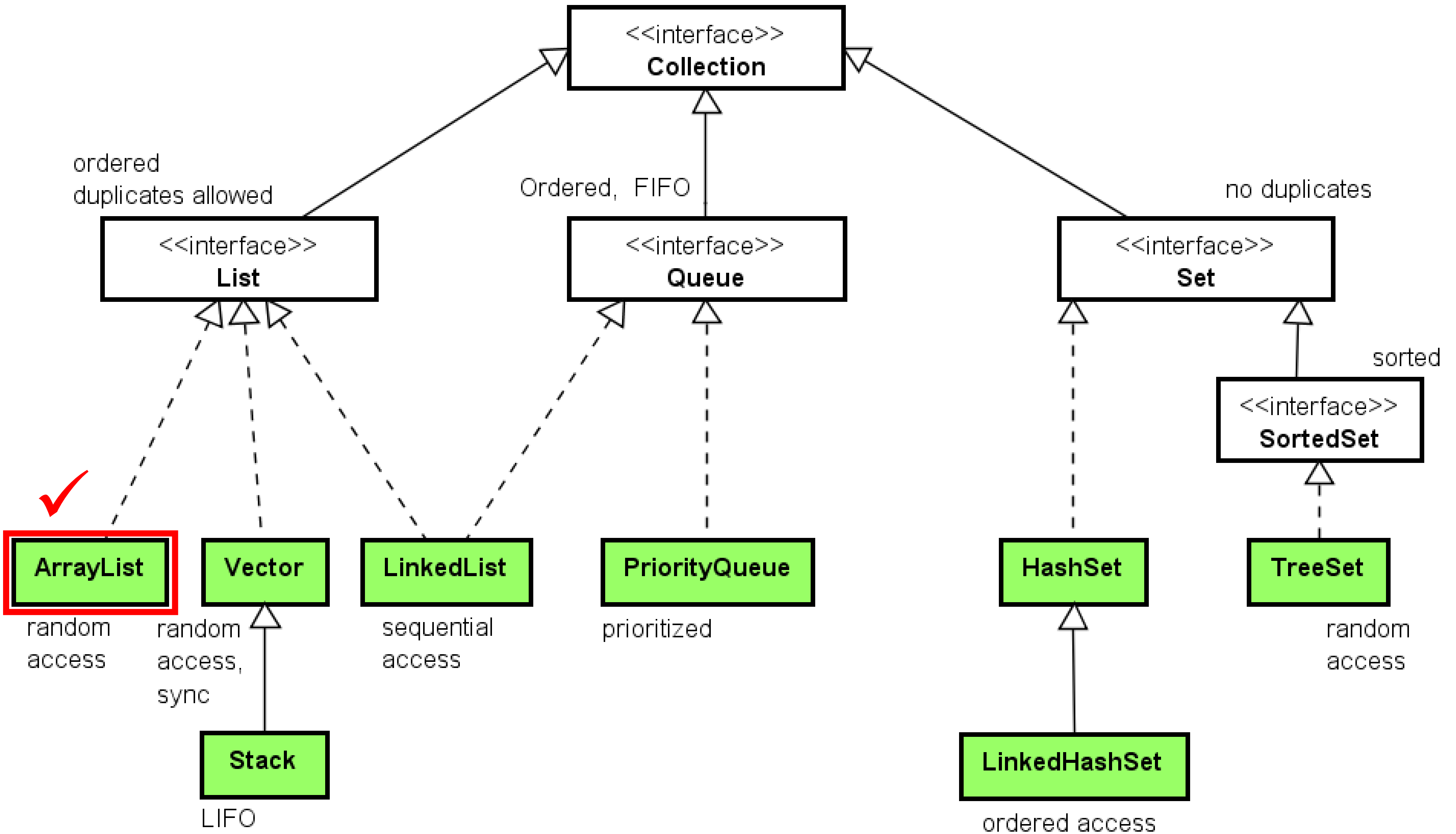
- ***collection****:* for storing a collection of elements.
  - e.g. Sets (nonduplicates), **Lists** (ordered), Stacks (LIFO), Queues (FIFO), PriorityQueues
- ***map****:* for storing key/value pairs.
  - e.g., HashMap

Remember: Collection Class Hierarchy (accurate)



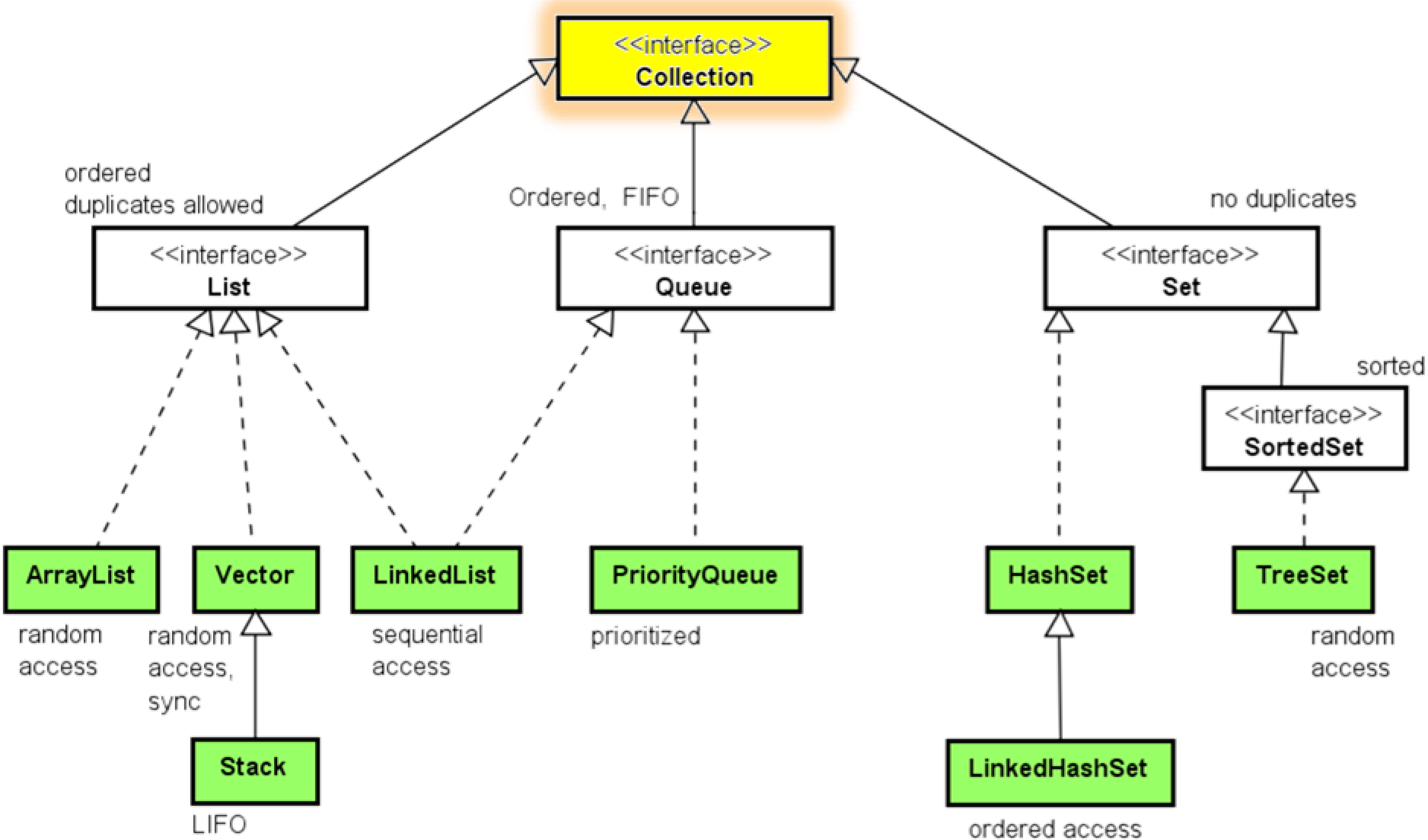
Remember: Collections Class Hierarchy

This diagram is **not accurate**. There are several abstract classes and interfaces missing, but it should help in visualization



Collection

Collections Class Hierarchy (simplified)



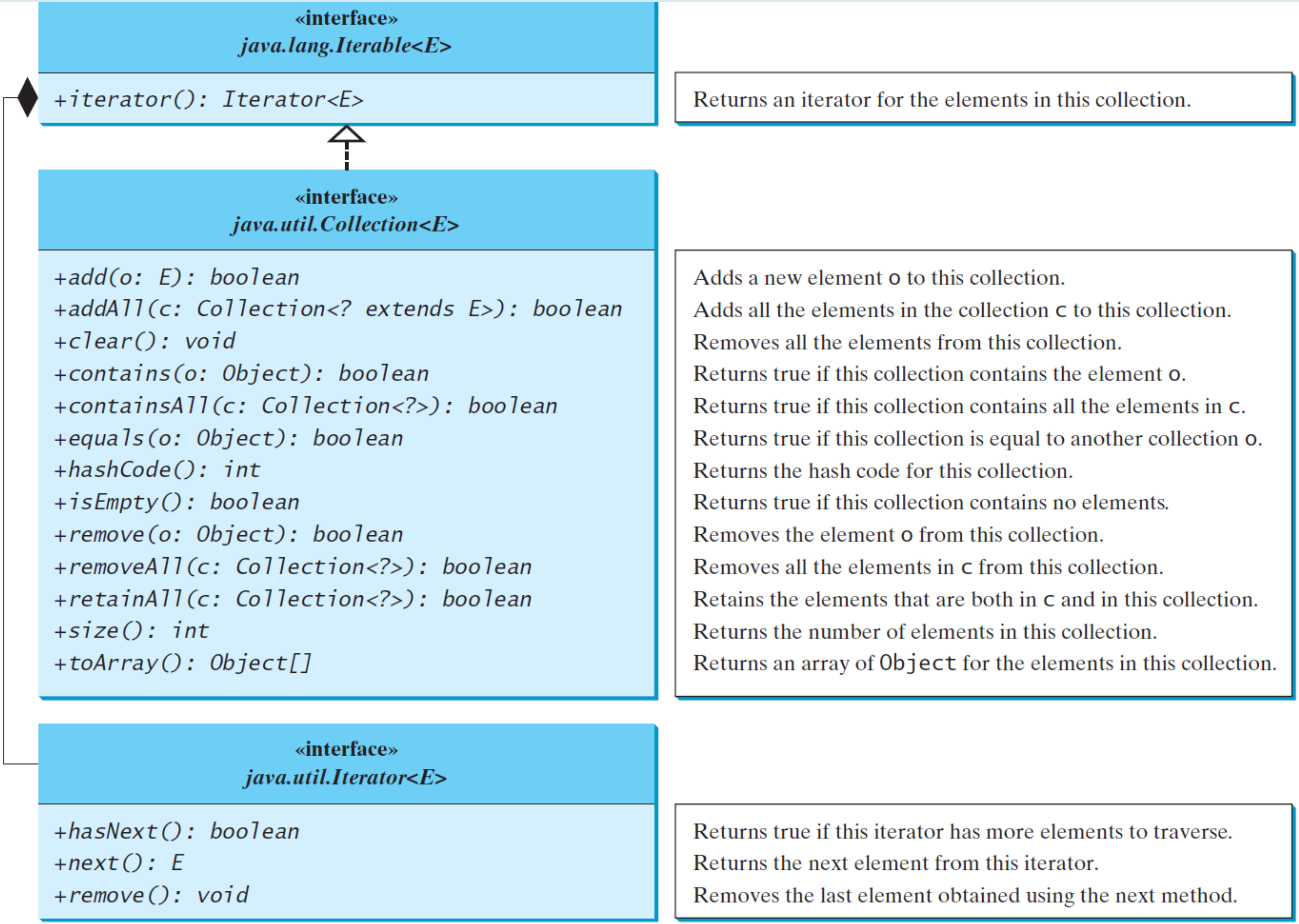
The `Collection` Interface

The `Collection` interface is the root interface for manipulating a collection of objects. It declares the methods for:

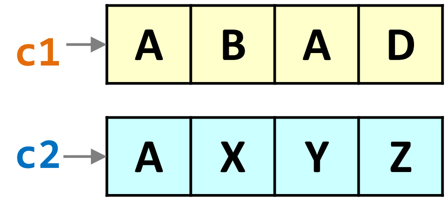
adding elements	**`add`** , **`addAll`**
removing elements	**`clear`** , **`remove`** , **`removeAll`** , **`retainAll`**
checking for elements	**`contains`** , **`containsAll`**
checking for equality	**`equals`**
checking the size	**`size`**, **`isEmpty`**
returning as array	**`toArray`**
returning an iterator*	**`iterator`**

* *by inheritance from* *`Iterable`*

The `Collection` Interface



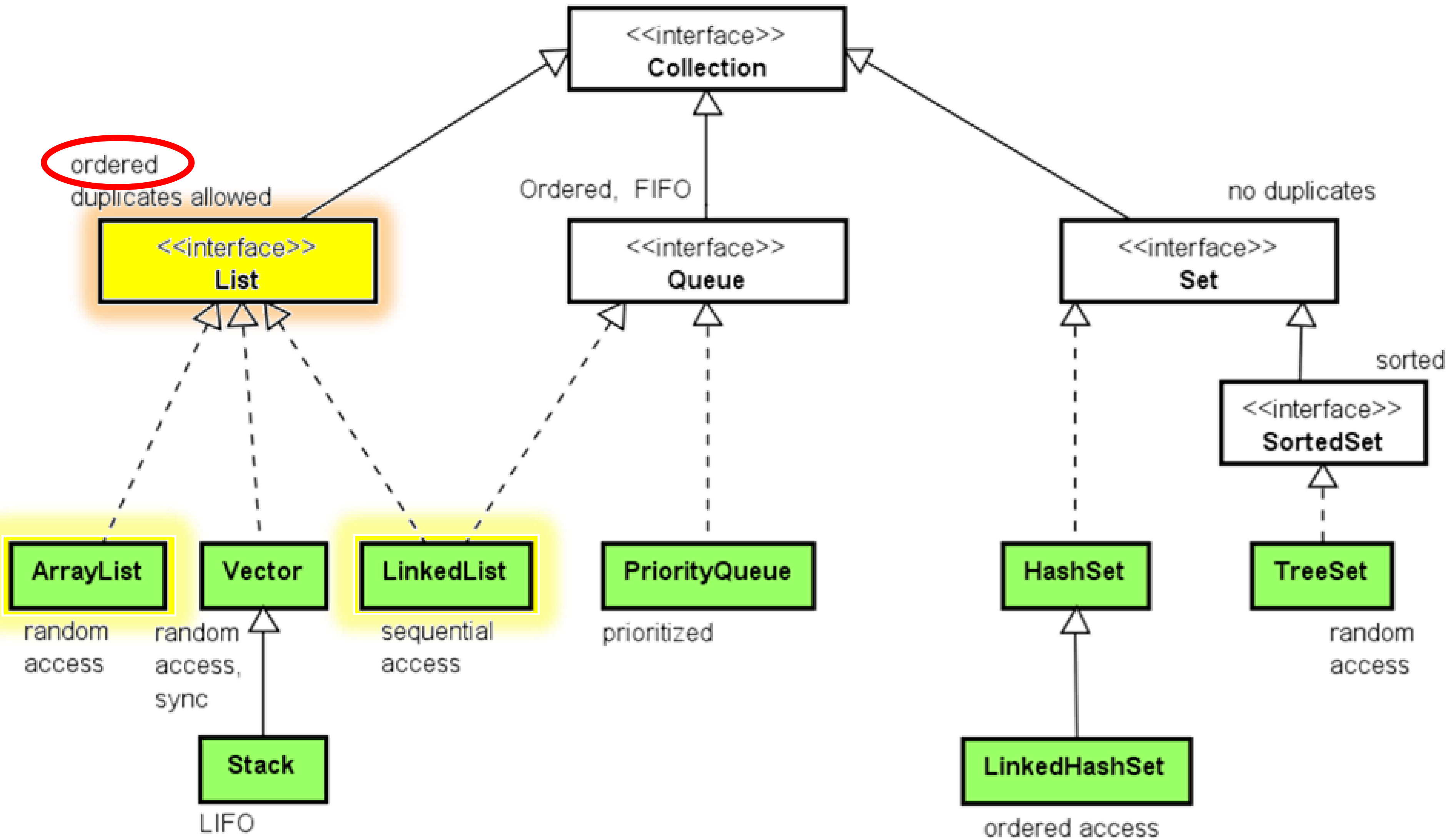
Example on Collection




```
ArrayList<String> c1 = new ArrayList<>(Arrays.asList("A","B","A","D"));
ArrayList<String> c2 = new ArrayList<>(Arrays.asList("A","X","Y","Z"));
System.out.println("c1: " + c1);
System.out.println("c2: " + c2);

//cloning, checking and removing
ArrayList<String> c3 = (ArrayList<String>) c1.clone();
System.out.println("Is B in c1? " + c3.contains("B"));      //true
c3.remove("B");
System.out.println("Is B in c1? " + c3.contains("B"));      //false

//addAll
c3 = (ArrayList<String>) c1.clone(); //restore c1 items
c3.addAll(c2); //add all c2 items to c3. Duplicates are allowed
System.out.println("Items in c1 and c2 combined: " + c3);   //[A,B,A,D,A,X,Y,Z]

//removeAll
c3 = (ArrayList<String>) c1.clone(); //restore c1 items
c3.removeAll(c2); //remove all c2 items from c3
System.out.println("Items in c1 but not in c2: " + c3);     //[B,D]

//retainAll
c3 = (ArrayList<String>) c1.clone(); //restore c1 items
c3.retainAll(c2); //keep items in c3 that are also in c1
System.out.println("Items in c1 (intersection) c2: " + c3); //[A,A]
```

Lists

Collections Class Hierarchy (simplified)



`List` Interface

The `List` interface

- Ordered (each element has an index)
- Duplicates allowed.

Methods:

- All methods from **`Collection`**
- & **iterator** methods: **`listIterator()`**
- & **index-based** methods
  - **Adding** `add(index,element)`
    `addAll(index, collection)`
  - **Removing** `remove(index)`
  - **Setting** `set(index, element)`
  - **Getting** `get(index)`
  - **Checking** `indexOf(element)`
    `lastIndexOf(element)`
  - **Sublist** `subList(fromIndex, toIndex)`

Concrete classes

- **`ArrayList`**, **`LinkedList`**

`ArrayList<E>`

Methods:

- All methods from **`List`**
- **& extra method**
  - **`trimToSize()`** Trims the capacity of this ArrayList instance to be the list's current size.

Constructors

- ArrayList()
  - Creates an empty list with default capacity.
- ArrayList(int initialCapacity)
- ArrayList(c: Collection<? extends E>)
  - Create ArrayList from an existing collection of any subtype of E

`LinkedList<E>`

Methods:

- All methods from **`List`**
- **& extra methods** specifically for *first* and *last* elements
  - **Adding** `addFirst(), addLast()`
  - **Removing** `removeFirst(), removeLast()`
  - **Getting** `getFirst(), getLast()`

Constructors

- LinkedList()
  - Creates a default empty list
- LinkedList(c: Collection<? extends E>)

Initializing a List as We Create it

In Java 9+, all of the following is valid:

A) To create and initialize a specific type of a list (e.g. ArrayList):

```
ArrayList<Integer> y=new ArrayList<>(Arrays.asList(1,2,3));
ArrayList<Integer> y=new ArrayList<>(List.of(1,2,3));
```

B) To create a general list (i.e. can only use methods from List):

```
List<Integer> y = Arrays.asList(1,2,3);//y is modifiable
List<Integer> x = List.of(1,2,3);      //x is unmodifiable
```

Aside: difference between `Arrays.asList` and `List.of`:

`List.of` returns an **unmodifiable** (immutable) list (Java 9+ only)

`Arrays.asList` returns a **modifiable** list

Example: **y**`.set(1,10)` is valid (y is modifiable).

x`.set(1,10)` causes a runtime error (x is immutable)

Note: `List.of()` is a static (implemented) method in `List` interface.

Example

The code on the right will produces the **same outcome** when list is declared using any of these two statements:

```
List<String> list = new ArrayList<>();
```

OR

```
List<String> list = new LinkedList<>();
```

```
list.add("A");
list.add("B");
list.add("C");
list.set(0, "X");
list.add("D");
System.out.println(list);
System.out.println(list.indexOf("C"));
System.out.println(list.indexOf("Z"));
System.out.println(list.remove("Z"));
System.out.println(list.remove("C"));
System.out.println(list);
```

OUTPUT

```
[X, B, C, D]
2
-1
false
true
[X, B, D]
```

Practice

Assume `list1 = ["red", "yellow", "green"]`
`list2 = ["red", "yellow", "blue"].`

What is **list1** after executing each statement in the following code fragment (as if they were in **one program** in this order):

a) **list1.addAll(list2)**?

b) **list1.remove(list2)**?

c) **list1.removeAll(list2)**?

d) **list1.retainAll(list2)**?

e) **list1.clear()**?

f) **list1.add(list2)**?

Q: if we write the statements (a) to (f) once where list1 & list2 are ArrayLists and once when they are LinkedLists, **will there be any difference in the way we write the statements a-f?**

When to use which?

In order to answer this question, we need to learn a few things first about ArrayLists vs LinkedLists, namely:

1) The **internal structure** of the two types of lists (ArrayList vs LinkedList) and they are stored in the memory.

2) How their **methods are implemented**.

- e.g. while add(E data) is supported by the two lists, this method is **implemented differently** in each list.

and the **time complexity** of the different methods

- In terms of the Big-Oh notation

Next slides, we will discuss each of the above items

Remember: Structure of an ArrayList

An array list is implemented using an **array** (a fixed-size data structure). Whenever the current array cannot hold new elements in the list, a larger new array is created to replace the current array. Arrays use an **INDEX** to reference its elements

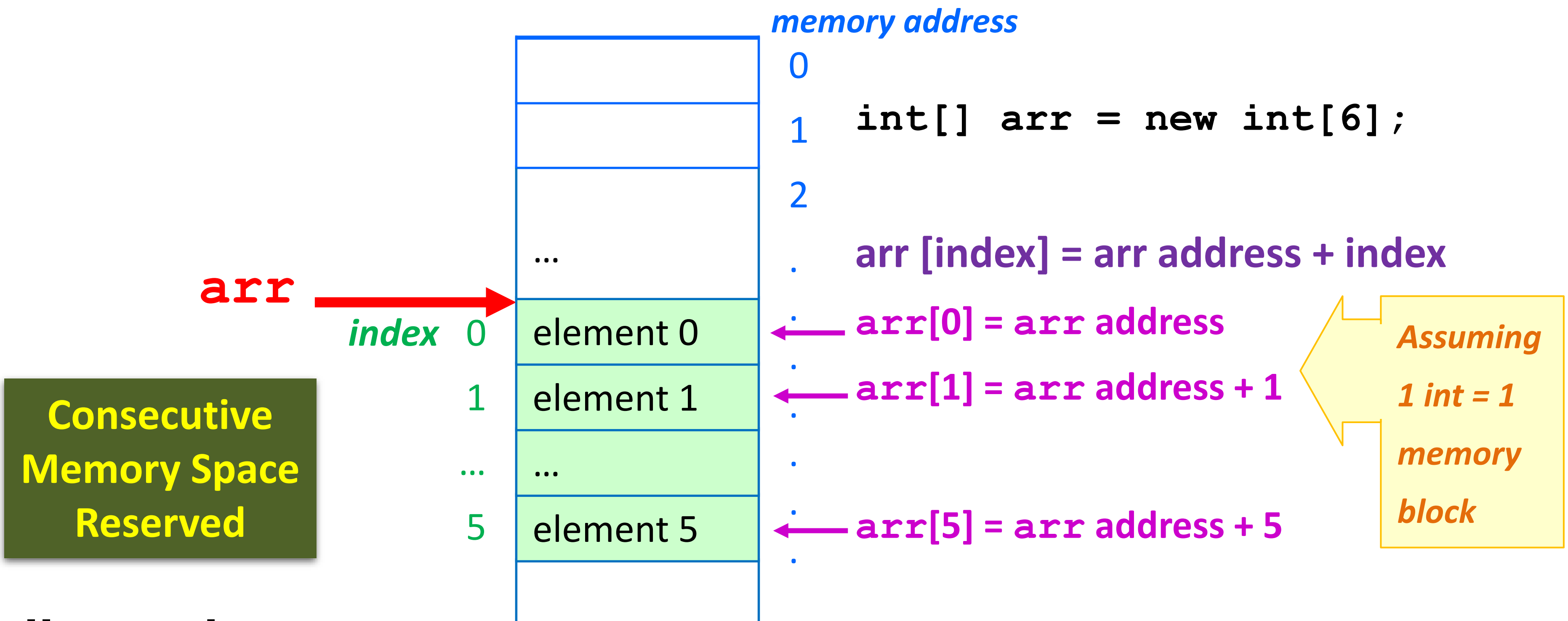


Illustration:

- http://cs.armstrong.edu/liang/animation/web/ArrayList.html

Basic Structure of a LinkedList, cont.

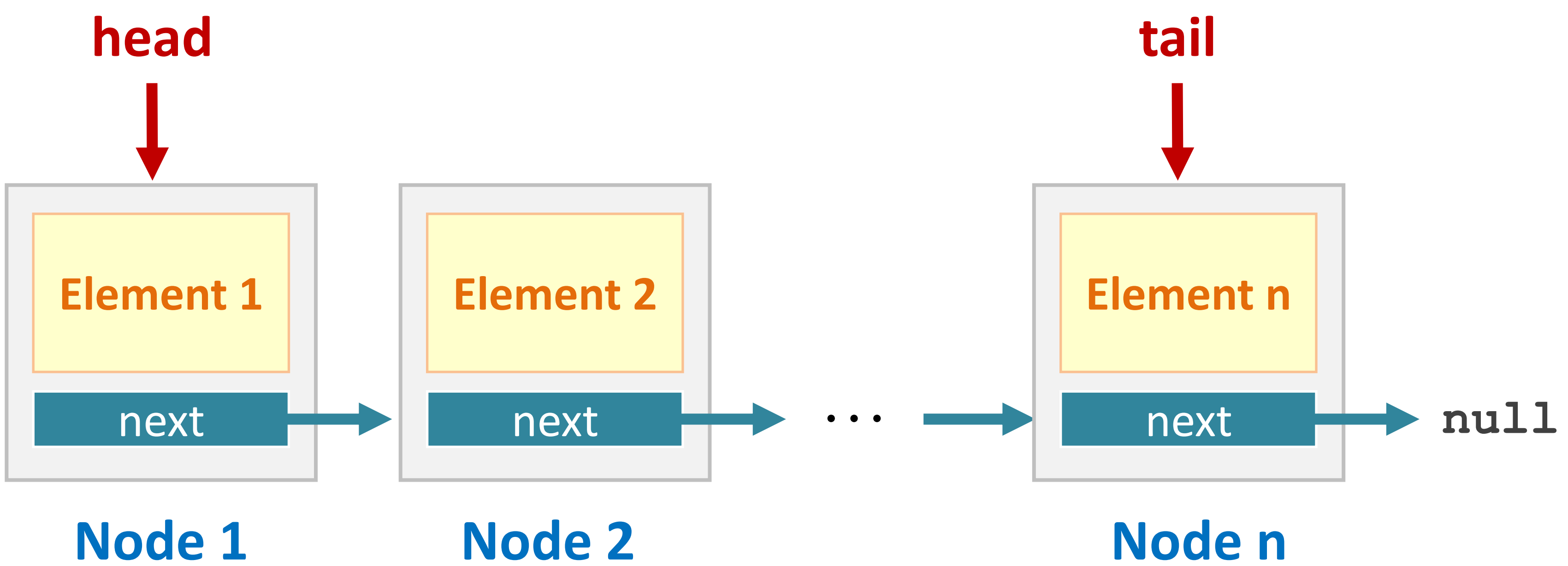
Each data element is contained in an object, called the ***node***.

When a new element is added to the list, a node is created to contain it.

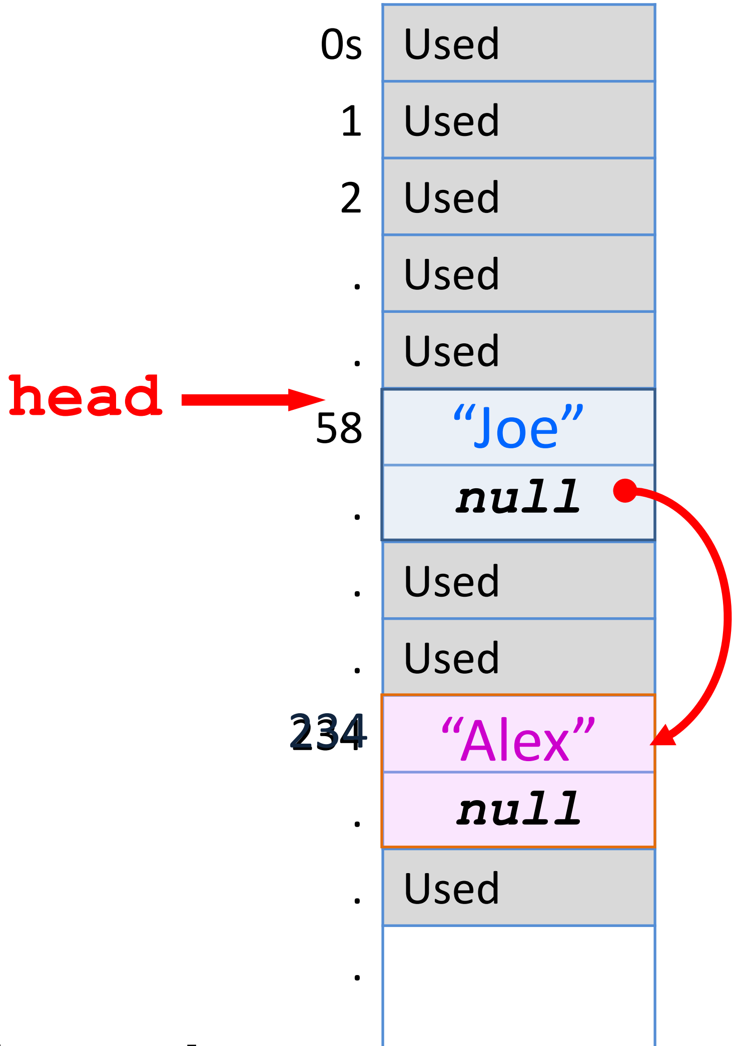
Each node is linked to its next neighbor.

A variable **head** refers to the first node, and a variable **tail** to the last node.

- If the list is empty, both **head** and **tail** are **null**.



How LinkedLists are stored in Memory?




```
LinkedList list= new LinkedList();
```

// head → null , no space reserved

```
list.add("Joe");
list.add("Alex");
```

Assumption:

1 string =

1 memory block

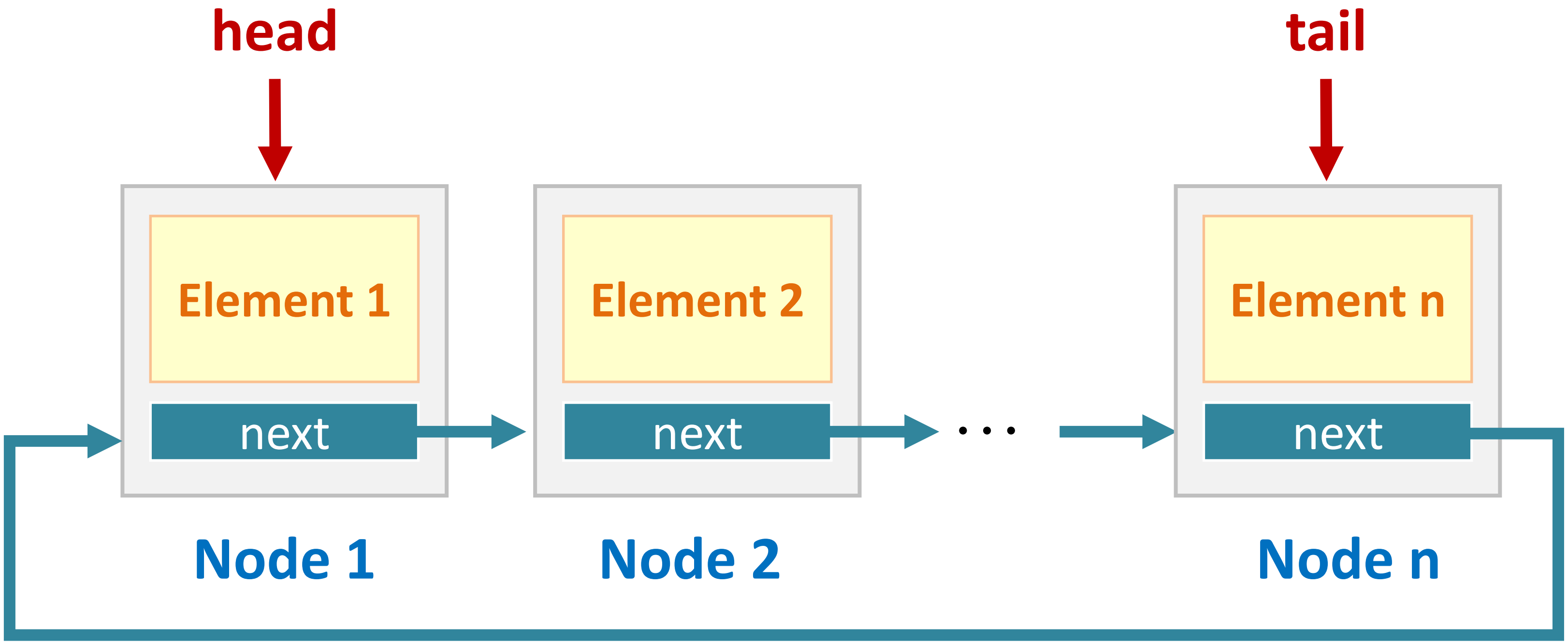
No easy indexing!!

Illustrations:

- http://cs.armstrong.edu/liang/animation/web/LinkedList.html
- https://visualgo.net/en/list

Others types of Linked Lists

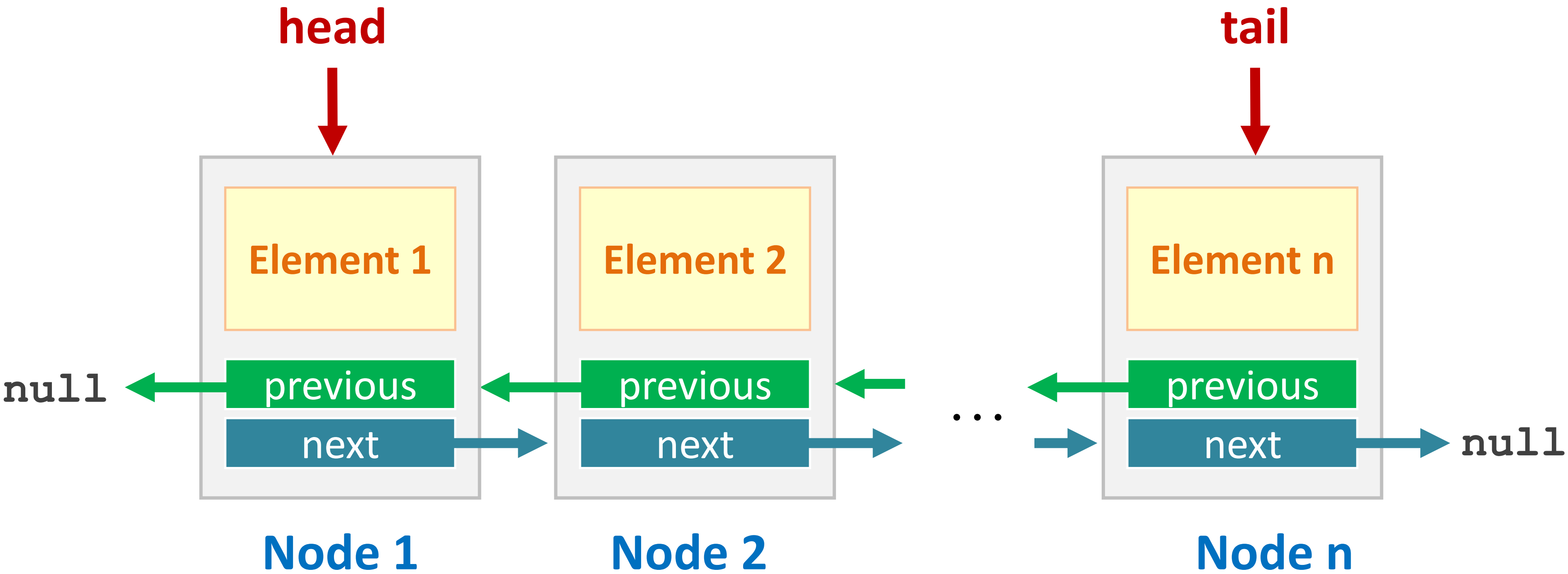
Circular Linked List



Others types of Linked Lists, cont.

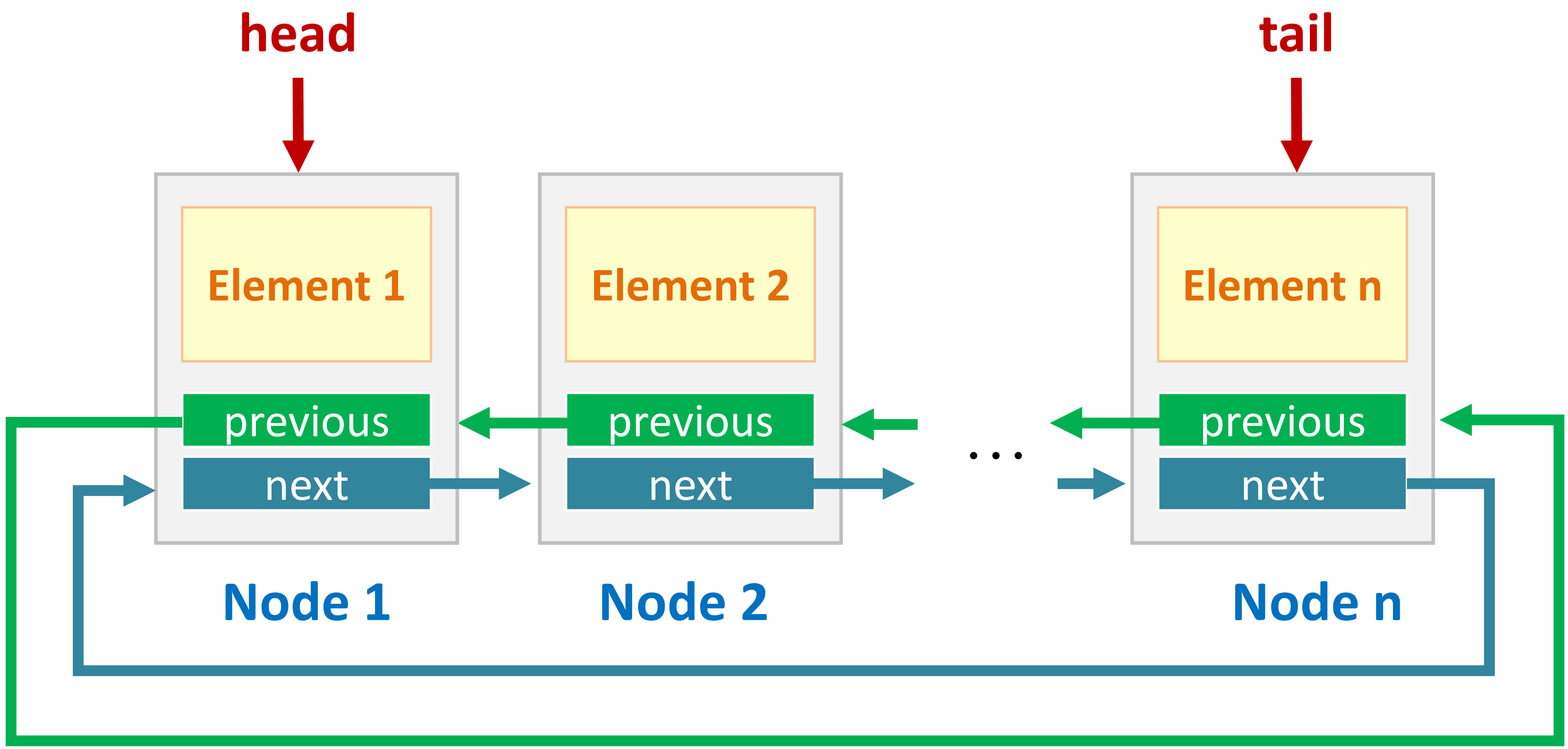
Doubly Linked List

- This is the type that Java 1.7 uses for `LinkedList` class
- Can be traversed in both directions



Others types of Linked Lists, cont.

Circular Doubly Linked List



When to use which (again)?

In general, use either one if you need **access through an index.**

However, note the following (based on the table on the next slide):

- `ArrayList`:
  - Efficient for **getting/setting elements by an index**
  - Good for Inserting/removing elements near **end** (but not beginning)
  - Access elements using "**Random Access**" (good)
- `LinkedList`:
  - Efficient for inserting/removing elements near **beginning** and **end.**
  - Efficient when using **iterators**.
  - Access elements using "**Sequential Access**" (bad)
    - hence it is not good for getting/setting elements away from beginning/end

When should I use standard arrays?

A list can grow or shrink dynamically. An array is fixed once it is created. **IF** your application does not require insertion or deletion of elements, the **most efficient data structure is the array**.

When to use which?

Methods	ArrayList	LinkedList
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`remove(index: int)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`set(index: int, e: E)`	$O(1)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`clear()`	$O(1)$	$O(1)$
`size()`	$O(1)$	$O(1)$

get(i) in ArrayList vs LinkedList

Run the code below and compare the time taken by get(i) in ArrayLists vs LinkedLists

```
final int N = 100000;   long startTime, endTime, totalTime;
//create an ArrayList and a LinkedList of 100,000 elements
ArrayList<Integer> arraylist = new ArrayList<>(N);
for (int i = 0; i < N; i++) arraylist.add(i);
LinkedList<Integer> linkedlist = new LinkedList<Integer>();
for (int i = 0; i < N; i++) linkedlist.add(i);

//get(i) all elements in ArrayList and compute time
startTime = System.currentTimeMillis();
for (int i = 0; i < N; i++)
   arraylist.get(i);
endTime = System.currentTimeMillis();
totalTime = endTime - startTime;
System.out.printf("ArrayList: get(i) %d elements took %d ms\n",N,totalTime);

//get(i) all elements in LinkedList and compute time
startTime = System.currentTimeMillis();
for (int i = 0; i <N; i++)
    linkedlist.get(i);
endTime = System.currentTimeMillis();
totalTime = endTime - startTime;
System.out.printf("LinkedList: get(i) %d elements took %d ms\n",N,totalTime);
```

Exercise: Replace get(i) in this code with other methods from the table in previous slide and compare the time.

Big-Oh Notation

Big-Oh notation is a mechanism for quickly **communicating the efficiency of an algorithm.**

- Big-Oh notation indicates the **growth rate of a function** (efficiency) when the **size of data (n)** changes.
  - The letter O is used because the growth rate of a function is also referred to as the "**O**rder of a function".

In big-Oh notation:

- The performance is specified as **a function of *n* which is the size of the problem**.
  - e.g. ***n*** may be the size of an array, or the number of values to compute
- Only the **most significant expression** of ***n*** is chosen:
  - e.g. If the method performs $n^3 + n^2 + n$ **steps**, it is O(n^3).
  - **Significance ordering**: ***2^n , n^5 , n^4 , n^3 , n^2 , $n*log(n)$, n , $log(n)$***
- **Constants are ignored for big-Oh:**
  - e.g. If the method performs $5*n^3 + 4*n^2$ steps, it is O(n^3).
  - This is because we measure the growth rate, not the number of executions, and hence both n and 10n will grow at the same rate.

Common Big-Oh Notation Values

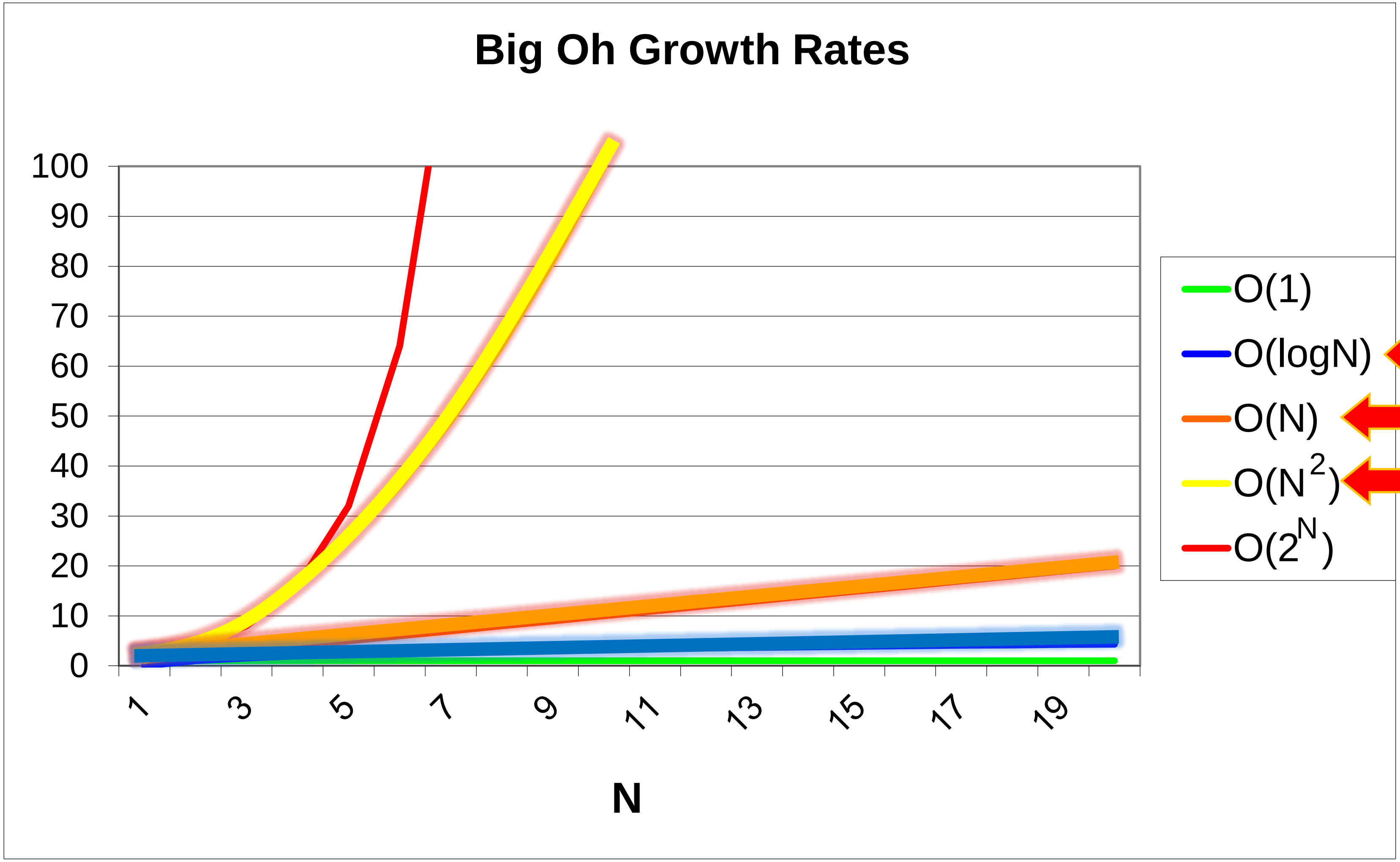
There are certain classes of functions with common names:

- constant = $O(1)$
- logarithmic = $O(\log n)$
- linear = $O(n)$
- quadratic = $O(n^2)$
- exponential = $O(2^n)$

These functions are listed in order of fastest to slowest.

- For example, for large values of *n*, an algorithm that is considered $O(n)$ is much faster than an algorithm that is $O(2^n)$.
- Big-Oh notation is useful for specifying the growth rate of the algorithm execution time.
  - How much longer does it take the algorithm to run if the input size is doubled?

Big Oh Growth Rates



Practice: Big-O Notation

What is the Big-O of the following segments of codes?

```
System.out.print(a[16]);
System.out.print(a[n-1]);
```

$O(1)$

```
for(i = 0; i< n; i++)
  System.out.print(a[i]);
```

$O(n)$

```
for(i = 0; i< n; i++)
   for(j = 0; j<n; j++)
      System.out.print(a[i][j]);
```

$O(n^2)$

Demo: `ArrayList` and `LinkedList`

The program below creates an integer ArrayList with chosen values. **Then it** creates an Object LinkedList **from the above ArrayList** and insert/remove elements from the list. **Finally**, it prints all elements from both lists as in this sample output:.

```
A list of integers in the array list: [10, 1, 2, 30]
Linked list elelemts (->): [green, 10, red, 1, 2]
Linked list elements (<-): 2 1 red 10 green
```

```java
ArrayList<Integer> arrayList = new ArrayList<>();
arrayList.add(1); // 1 is autoboxed to new Integer(1)
arrayList.add(2);
arrayList.add(0, 10);
arrayList.add(3, 30);
System.out.print("A list of integers in the array list: ");
System.out.println(arrayList);

LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
linkedList.add(1, "red");
linkedList.removeLast();
linkedList.addFirst("green");
System.out.print("Linked list elelemts (->): ");
System.out.println(linkedList);
System.out.print("Linked list elements (<-): ");
for (int i = linkedList.size() - 1; i >= 0; i--)
    System.out.print(linkedList.get(i) + " ");
```

Q: Is this the best way?

Experiment: Printing all elements in reverse order

Experiment (based on the previous slide):

- For 100,000 elements, the reported time
  - Loop1: 4.212 sec
  - Loop2: 0.016 sec

```
long startTime, endTime;
LinkedList<Integer> list = new LinkedList<Integer>();
for (int i = 0; i < 100000; i++)  list.add(i);

//Loop 1: using an index
startTime = System.currentTimeMillis();
for (int i = list.size() - 1; i >= 0; i--) list.get(i);
endTime = System.currentTimeMillis();
System.out.printf("Time using index: %d ms\n", (endTime-startTime));

// Loop 2: using an iterator
ListIterator<Integer> itr = list.listIterator();
startTime = System.currentTimeMillis();
while (itr.hasNext()) itr.next();
while (itr.hasPrevious()) itr.previous();
endTime = System.currentTimeMillis();
System.out.printf("Time using iterator: %d ms\n",(endTime-startTime));
```

Practice: `ArrayList` and `LinkedList`

Repeat the previous practice question (3 slides ago) but use an **iterator** to display the elements of the linked list.

```
ArrayList<Integer> arrayList = new ArrayList<>();
arrayList.add(1); // 1 is autoboxed to new Integer(1)
arrayList.add(2);
arrayList.add(0, 10);
arrayList.add(3, 30);
System.out.print("A list of integers in the array list: ");
System.out.println(arrayList);

LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
linkedList.add(1, "red");
linkedList.removeLast();
linkedList.addFirst("green");

ListIterator<Object> itr = linkedList.listIterator();
System.out.print("Linked list elelemts (->): ");
while(itr.hasNext()) System.out.print(itr.next() + " ");
System.out.println();
System.out.print("Linked list elements (<-): ");
while(itr.hasPrevious()) System.out.print(itr.previous() + " ");
```