Advanced Programming Techniques in Java

COSI 12B

Running Times & `Comparable`

Lecture 18

Class Objectives

- Complexity & Running time (section 13.2)
- `Comparable` Interface (section 10.2)
- `List` (first subsection 11.1)

Review: `ArrayList` methods c

`add(`**`value`**`)`	appends value at end of list
`add(`**`index, value`**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**`value`**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**`index`**`)`	returns the value at given index
`remove(`**`index`**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**`index, value`**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

Review: `ArrayList` methods (cont.)

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list (seen later)
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Review: Three main categories of Errors

- Syntax Errors
- Run-time errors
- Logic errors

Review: Efficiency

Computing time and memory are bounded resources.

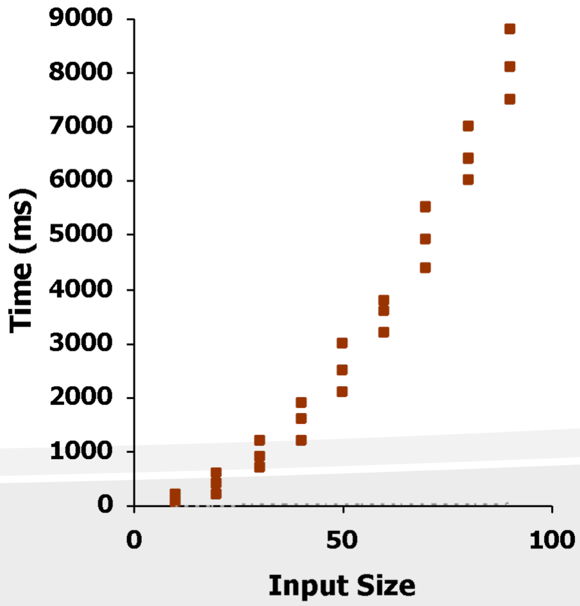
Efficiency:

- Different algorithms that solve the same problem often differ in their efficiency.
- More significant than differences due to hardware (CPU, memory, disks, ...) and software (OS, programming language, compiler, ...).

=> Running Time/Computational Complexity

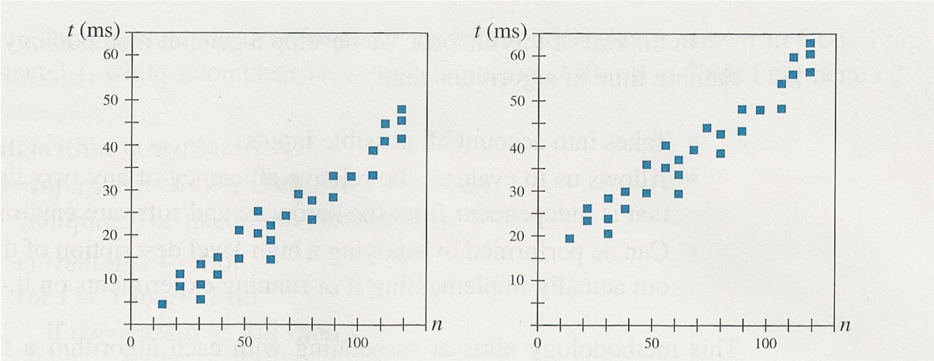
Empirical Analysis

- Run time can be studied experimentally
  - Write a program implementing the algorithm
  - Run the program with inputs of varying size
  - Get an accurate measure of the actual running time
  - Plot the results



Limitations of Empirical Analysis

- Experiment can be done only on a limited set of test inputs
- Difficult to compare the efficiency of two algorithms unless experiments have been performed on same environment
  - Hardware environment (processor, clock, rate, memory, etc.)
  - Software environment (OS, programming language, compiler, interpreter, etc.)



- Necessary to implement and execute an algorithm to study its run time

Theoretical Analysis

- General methodology for analyzing run time of algorithms that:
  - Consider all possible inputs
  - Can be performed studying high-level description of the algorithm
  - Evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment

Asymptotic Performance

- We care most about *asymptotic performance*
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/etc.

Runtime Efficiency

- Assume the following:
  - Any single Java statement takes the same amount of time to run
  - A method call's runtime is measured by the total number of statements inside the method's body
  - A loop's runtime, if the loop repeats n times, is n times the runtime of the statements in its body
  - We want to count the number of statements that are needed to complete the execution

Efficiency examples

```
statement1;
statement2;
statement3;

for (int i = 1; i <= n; i++) {
    statement4;
}

for (int i = 1; i <= n; i++) {
    statement5;
    statement6;
    statement7;
}
```

Efficiency examples

```
statement1;
statement2;
statement3;
```

3

```
for (int i = 1; i <= n; i++) {
   statement4;
}
```

n

```
for (int i = 1; i <= n; i++) {
   statement5;
   statement6;
   statement7;
}
```

3n

4n + 3

Efficiency examples (cont.)

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        statement1;
    }
}

for (int i = 1; i <= n; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

Efficiency examples (cont.)

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        statement1;
    }
}

for (int i = 1; i <= n; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

n^2

$4n$

$n^2 + 4n$

- How many statements will execute if n = 10? If n = 1000?

Algorithm growth rates

- We measure runtime in proportion to the input data size, n
  - **Growth rate**: Change in runtime as n changes
- Say an algorithm runs **$0.4n^3 + 25n^2 + 8n + 17$**
  - Consider the runtime when n is extremely large
  - We ignore constants like 25 because they are tiny next to n
  - The highest-order term (n^3) dominates the overall runtime
  - We say that this algorithm runs ”in the order of" n^3
  - or **O(n^3)** for short ("**Big-Oh** of n^3")

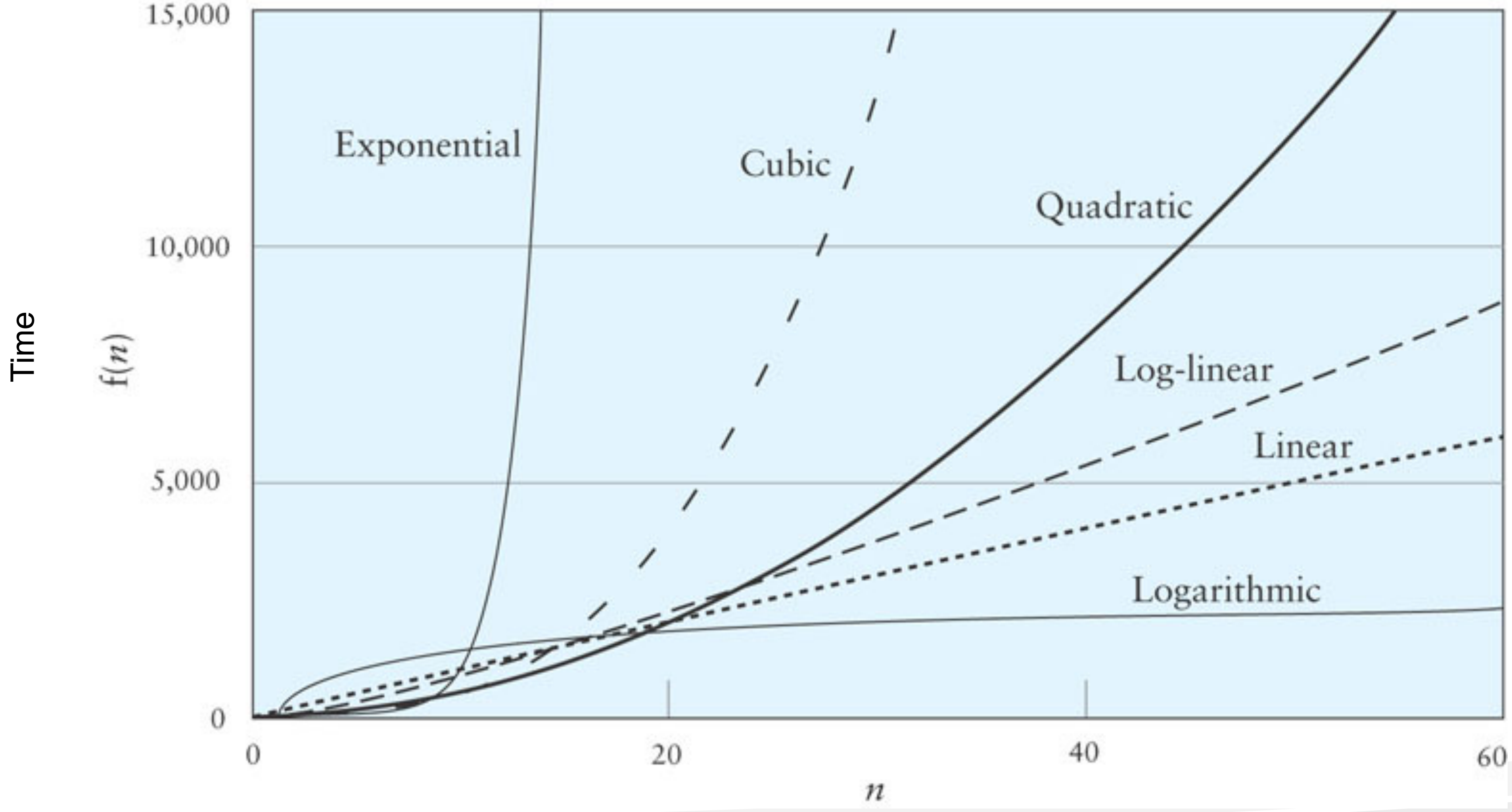
- **Big-Oh It's a measure of the longest amount of time it could possibly take for the algorithm to complete (upper bound)**

Complexity classes

- **Complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size n

Big-O	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential
O($n!$)	Factorial

Practical Complexity



Input Size

Effect of Different Growth Rates

$O(f(n))$	f(50)	f(100)	f(100)/f(50)
$O(1)$	1	1	1
$O(\log n)$	5.64	6.64	1.18
$O(n)$	50	100	2
$O(n \log n)$	282	664	2.35
$O(n^2)$	2500	10,000	4
$O(n^3)$	12,500	100,000	8
$O(2^n)$	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
$O(n!)$	3.0×10^{64}	9.3×10^{157}	3.1×10^{93}

Collection efficiency

- Efficiency of various operations on `ArrayList`:

Method	ArrayList
`add`	O(1)
`add(`**index**`,` **value**`)`	O(n)
`indexOf`	O(n)
`get`	O(1)
`remove`	O(n)
`set`	O(1)
`size`	O(1)

The `Comparable` interface

Natural ordering

- Many types have a notion of a **natural ordering** that describes whether one value of that type is "less than" or "greater than" another:
  - `int`, `double`: numeric value
  - `String`: lexical (alphabetical) order

- Not all types have a natural ordering:
  - `Point`: How would they be ordered? By `y`? By `x`? Distance from origin?
  - `GroceryList`: What makes one list "less than" another?

Uses of natural ordering

- An `ArrayList` of orderable values can be sorted using `Collections.sort()`

```
ArrayList<String> words = new ArrayList<String>();
words.add("four");
words.add("score");
words.add("and");
words.add("seven");
words.add("years");
words.add("ago");

// show list before and after sorting
System.out.println("before sort, words = " + words);
Collections.sort(words);
System.out.println("after sort, words = " + words);
```

- Output:

```
before sort, words = [four, score, and, seven, years, ago]
after sort, words = [ago, and, four, score, seven, years]
```

Comparable interface

- The natural ordering of a class is specified through the `compareTo` method of the `Comparable` interface:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- Classes such as `String`, `Integer`, … implement `Comparable`
- `compareTo` returns an integer that is < 0, > 0, or 0:

Relationship	Primitive comparison	Object comparison
less than	`if (x < y) {`	`if (x.compareTo(y) < 0)`
less than or equal	`if (x <= y) {`	`if (x.compareTo(y) <= 0)`
equal	`if (x == y) {`	`if (x.compareTo(y) == 0)`
not equal	`if (x != y) {`	`if (x.compareTo(y) != 0)`
greater than	`if (x > y) {`	`if (x.compareTo(y) > 0)`
greater or equal	`if (x >= y) {`	`if (x.compareTo(y) >= 0)`

How to compare objects?

- For any class that implements the `Comparable` interface
  - You can compare to objects with the `compareTo` method

```
String s1 = "hello";
String s2 = "world";
if (s1.compareTo(s2)<0) {
      System.out.println("S1 less than S2");
}
```

  - You cannot use relational operators. The next is illegal:

```
String s1 = "hello";
String s2 = "world";
if (s1 < s2) {
      System.out.println("S1 less than S2");
}
```

Implementing `Comparable`

- You can define a natural ordering for your own class by making it implement the `Comparable` interface
  - `Comparable` is a generic interface, `Comparable<T>`
  - When implementing it, you must write your class's name in `<>` after the word `Comparable`
  - Example: `public class Point` **`implements Comparable<Point>`**
  - You must also write a method `compareTo` that compares the current object (the implicit parameter) to a given other object
  - Example:

```
public int compareTo(Point p) {
    ...
}
```

Comparable example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    …

]
    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;     // same x, larger y
        } else {
            return 0;     // same x and same y
        }
    }
}
```

`compareTo` tricks

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;   // different x
    } else {
        return y - other.y;   // same x; compare y
    }
}
```

- The idea:
  - if `x > other.x`, then `x - other.x > 0`
  - if `x < other.x`, then `x - other.x < 0`
  - if `x == other.x`, then `x - other.x == 0`

Comparable implementation

```
// The CalendarDate class stores information about a single calendar
   date (month and day but no year).

public class CalendarDate implements Comparable<CalendarDate> {
    private int month;
    private int day;

    public CalendarDate(int month, int day) {
        this.month = month;
        this.day = day;
    }
  public int compareTo(CalendarDate other) {
        if (this.month != other.month) {
            return this.month - other.month;
        } else {
            return this.day - other.day;
        }
    }
    public String toString() {
        return this.month + "/" + this.day;
    }
}
```

Compares this calendar date to another date.Dates are compared by month and then by day.

Example Client Program

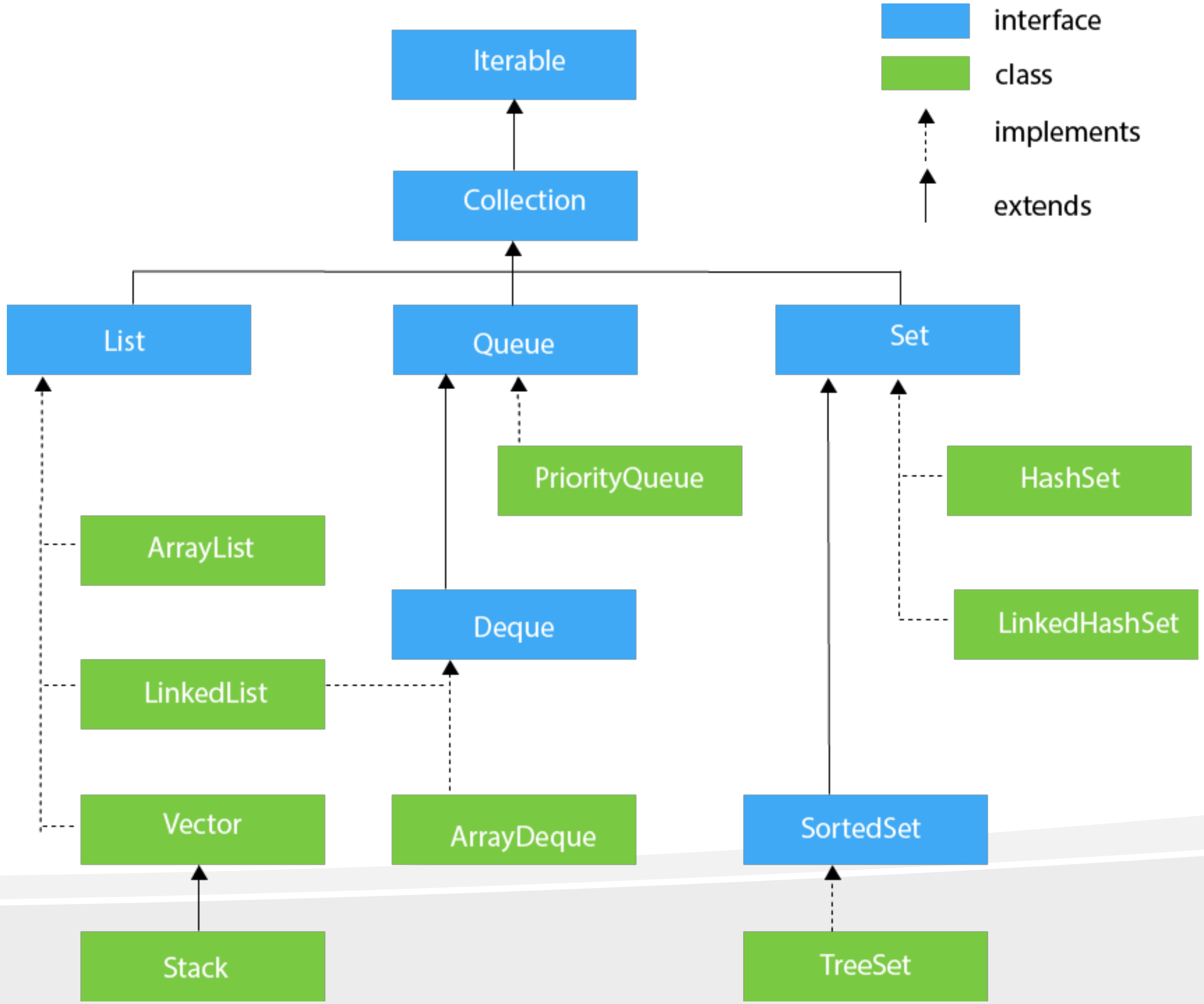
```
// Short program that creates a list of the birthdays of the first 5
// US Presidents and that puts them into sorted order.

import java.util.*;

public class CalendarDateTest {
    public static void main(String[] args) {
        ArrayList<CalendarDate> dates = new ArrayList<CalendarDate>();
        dates.add(new CalendarDate(2, 22));
        dates.add(new CalendarDate(10, 30));
        dates.add(new CalendarDate(4, 13));
        dates.add(new CalendarDate(3, 16));
        dates.add(new CalendarDate(4, 28));

        System.out.println("birthdays before sorting = " + dates);
        Collections.sort(dates);
        System.out.println("birthdays after sorting = " + dates);
    }
}
```

since CalendarDate implements the Comparable we can use the Collections.sort method

OUTPUT:
birthdays before sorting = [2/22, 10/30, 4/13, 3/16, 4/28]
birthdays after sorting = [2/22, 3/16, 4/13, 4/28, 10/30]

Collections

Collections Framework Diagram



Collections

- **Collection** is an object that stores data inside it
  - The data stored are called **elements**
  - Some collections maintain an ordering, some don't
  - Some collections allow duplicates, some don't

- Typical operations:
  - Add element, remove element, clear all elements, contains or find element, get size
  - Most collections are built with particular operations on that data, in mind

Collections

- A collection is an object that groups multiple elements into a single unit
- Very useful
  - Store, retrieve and manipulate data
  - Transmit data from one method to another
  - Data structures and methods written already for you

Collection examples

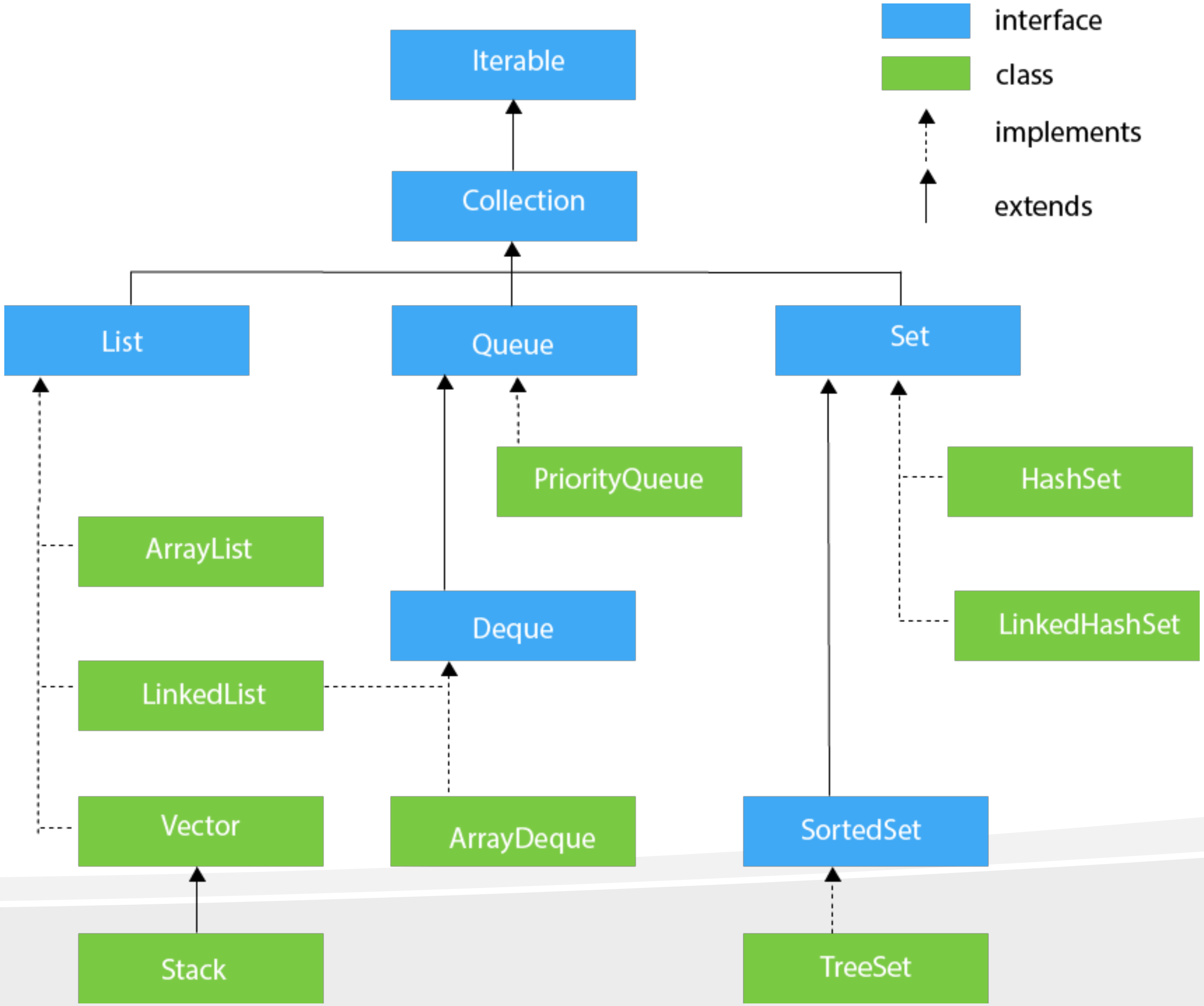
- **List**: Collection of elements
- **Set**: A collection of elements that is guaranteed to contain no duplicates and generally can be searched efficiently
- **Map**: A collection of (key, value) pairs in which each key is associated with a corresponding value
- **Stack**: A collection where the last element added is the first one to be removed
- **Queue**: A collection where elements are removed in the order in which they were added

Collections Framework

- Unified architecture for representing/storing and manipulating collections

- It has:
  - **interfaces** (`Set`, `List`, `Queue`, `Deque`)
  - **classes** (`ArrayList`, `Vector`, `LinkedList`, `PriorityQueue`, `HashSet`, `LinkedHashSet`, `TreeSet`)
  - algorithms

Collections Framework Diagram



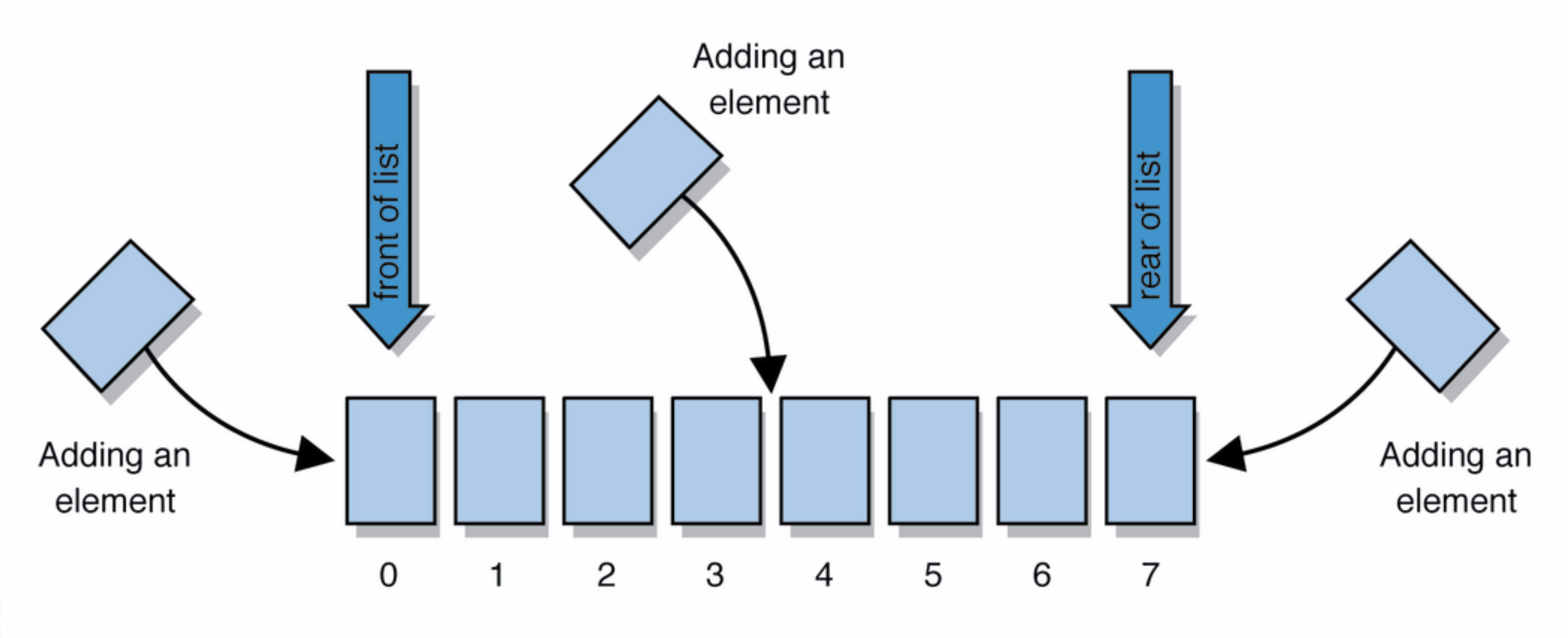
Java's `Collection` interface

- `Collection<E>` represents many kinds of collections. Every collection has the following methods:

Method name	Description
`add(`*value*`)`	adds the given value to the collection, returns true if the operation changed the collection
`addAll(`*collection*`)`	adds all elements from given collection to this one
`clear()`	removes all elements
`contains(`*value*`)`	returns `true` if the element is in the collection
`containsAll(`*collection*`)`	`true` if this collection contains all elements from the other
`isEmpty()`	`true` if the collection does not contain any elements
`removeAll(`*collection*`)`	removes all values contained in the given collection from this collection
`retainAll(`*collection*`)`	retains the elements of this collection that are also contained in the given collection
`remove(E element)`	removes the first occurrence in this list of the specified element
`iterator()`	returns a special object for examining the elements of the list in order (seen later)
`size()`	returns the number of elements in this list
`toArray()`	returns an array containing all the elements from this list

List

- **List** is an ordered sequence of elements. One of the most basic collections



List features

- Maintains elements in the order they were added (new elements are added to the end by default)
- Duplicates are allowed
- Operations:
  - add element to the list
  - insert element at given index
  - clear all elements
  - search for element
  - get element at given index
  - remove element at given index
  - get size
  - some of these operations are inefficient
- The list manages its own size; the user of the list does not need to worry about overfilling it

Java's `List` interface

- Java has an interface `List<E>` to represent a list of objects
  - It adds the following methods to those in `Collection<E>`:
- **`public void add(int index, E element)`**
  Inserts the specified element at the specified position in this list
- **`public E get(int index)`**
  Returns the element at the specified position in this list
- **`public int indexOf(E element)`**
  Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain it
- …

`List` interface, cont'd.

- **`public int lastIndexOf(Object o)`**
  Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain it
- **`public E remove(int index)`**
  Removes the object at the specified position in this list
- **`public E set(int index, E element)`**
  Replaces the element at the specified position in this list with the specified element
- Notice that the methods added to `Collection<E>` by `List<E>` all deal with indexes
- **A list has indexes while a general collection may not**

ArrayList

- `ArrayList` implements the `List` interface
  - It implements all the methods of that interface

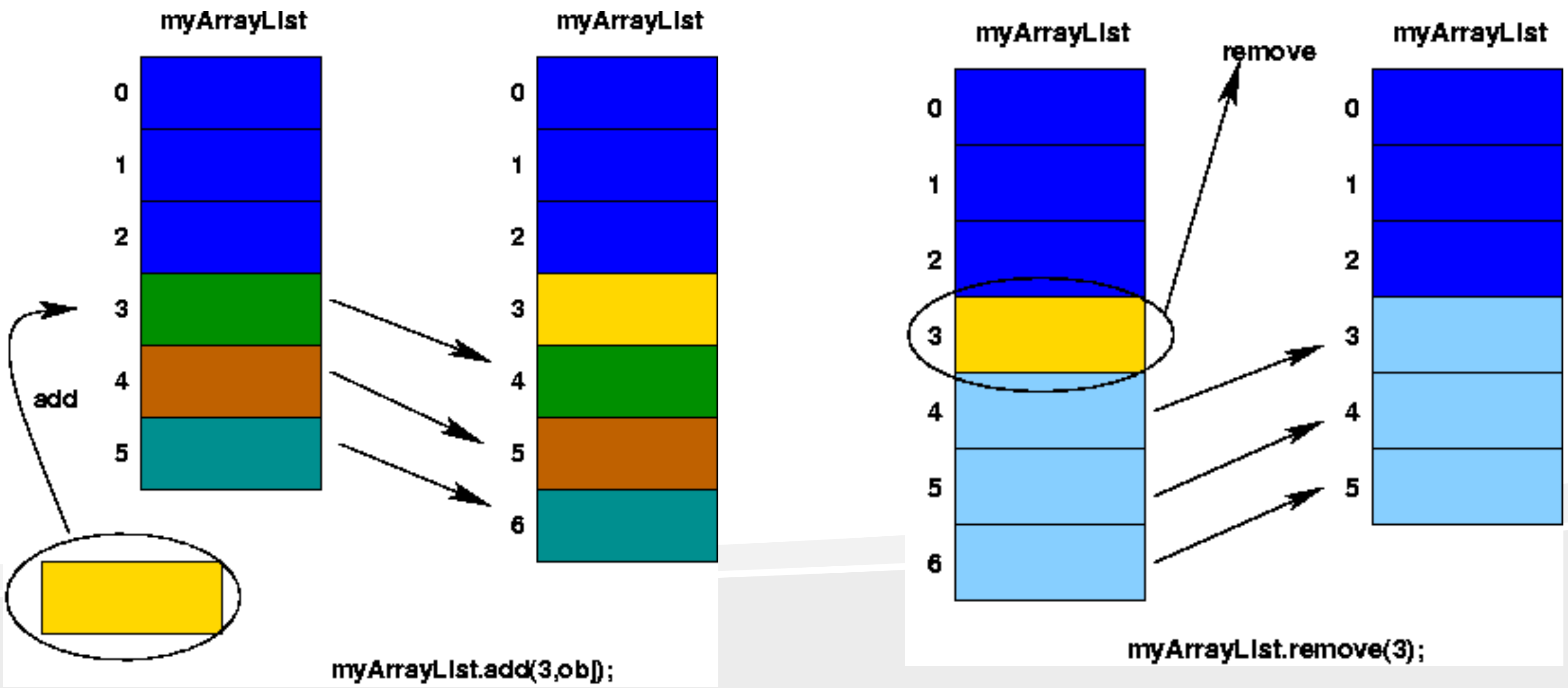
```
input: ArrayList<String> list
    int i=0;
    while (i < list.size()){
       String element = list.get(i);
       if (element.length()%2 == 0) {
           list.remove(i);
       else {
           i++; // skip to next element
       }
    }
```

slow performance for large files

- Read a list of strings and remove elements with even length

`ArrayList` limitations

- An `add` or `remove` operation on an `ArrayList` that is not at the end of the list will require elements to be shifted
  - This can be slow for a large list
  - What is the worst possible case?



Internals of `ArrayList`

- It is internally stored in an array. So, it has a certain capacity
  - Capacity: size of the array used to store the elements in the list
- It is always at least as large as the list size
- As elements are added to an `ArrayList`, its capacity grows “automatically”
- Random access to elements
  - `set/get` an element to/at specific index position has constant time, **O(1)**
  - To `add` at the end of the `ArrayList` it takes **O(1)**
- Remove, or add at a specified index operations have linear time **O(n)**

The underlying issue

- The elements of an `ArrayList` are too tightly attached; can't easily rearrange them
- Can we break the element storage apart into a more dynamic and flexible structure?