Advanced Programming Techniques in Java

Running Times & `Comparabl`

Lecture 18

Class Objectives

- Complexity & Running time (section 13.2)

- `Comparable` Interface (section 10.2)
- `List` (first subsection 11.1)

Review: `ArrayList` methods

`add(`**value**`)`	appends value at e
`add(`**index**`, `**value**`)`	inserts given value shifting subsequent
`clear()`	removes all elemen
`indexOf(`**value**`)`	returns first index w (1 if not found)
`get(`**index**`)`	returns the value at
`remove(`**index**`)`	removes/returns va subsequent values
`set(`**index**`, `**value**`)`	replaces value at g

`size()`	returns the number
`toString()`	returns a string repr as "`[3, 42, -7,`

Review: `ArrayList` methods (

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from th (at the end of the list, or
`contains(`**value**`)`	returns true if given value
`containsAll(`**list**`)`	returns true if this list cor
`equals(`**list**`)`	returns true if given othe
`iterator()` `listIterator()`	returns an object used to later)

`lastIndexOf(`**value**`)`	returns last index value is
`remove(`**value**`)`	finds and removes the giv
`removeAll(`**list**`)`	removes any elements fo
`retainAll(`**list**`)`	removes any elements no
`subList(`**from**`, `**to**`)`	returns the sub-portion o indexes **from** (inclusive)
`toArray()`	returns the elements in th

Review: Three main cate

- Syntax Errors

- Run-time errors
- Logic errors

Review: Efficiency

Computing time and memory are bounded resource

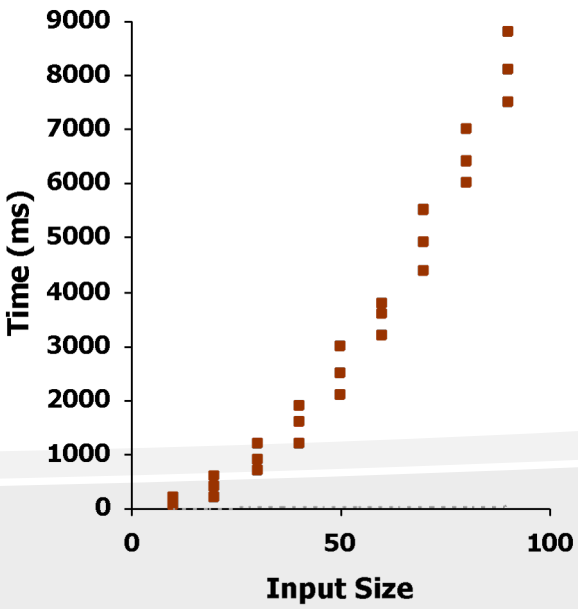
Efficiency:

- Different algorithms that solve the same pro
their efficiency.
- More significant than differences due to har
memory, disks, ...) and software (OS, programm
compiler, ...).

=> Running Time/Computational Complexity

Empirical Analysis

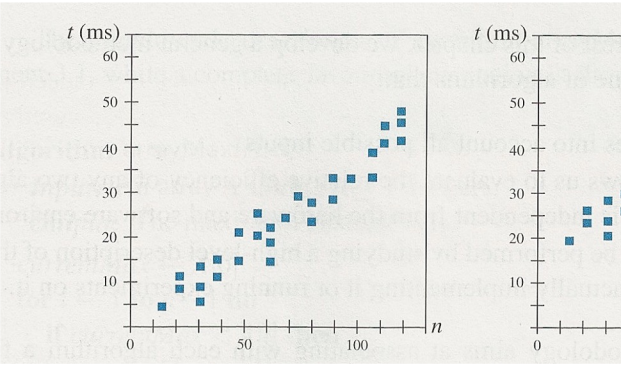
- Run time can be studied experimentally
  - Write a program implementing the algorithm
  - Run the program with inputs of varying size



- 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 i
- Difficult to compare the efficiency of two algorithms u on same environment
  - Hardware environment (processor, clock, rate, memor
  - Software environment (OS, programming language, co



- Necessary to implement and execute an algorithm to

Theoretical Analysis

- General methodology for analyzing run time of algori
  - Consider all possible inputs
  - Can be performed studying high-level description of th

- Evaluate the relative efficiency of any two algorithms i
  hardware and software environment

Asymptotic Performance

- We care most about *asymptotic*
  - How does the algorithm behave as the prob
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/etc.

Runtime Efficiency

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

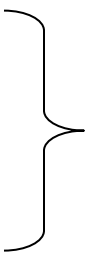
Efficiency examples

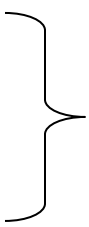
```
statement1;
statement2;
statement3;

for (int i = 1; i <= n; i++) {
statement4; } for (int i = 1; i
<= n; i++) { statement5; statement6;
statement7;
}
```

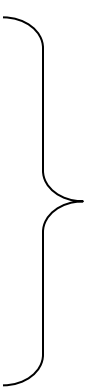
Efficiency examples

```
statement1;
```

statement2;3







statement3;

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

}4n + 3

```
for (int i = 1; i <= n; i++) {
    statement5;
    statement6;
}
```
3n statement7;

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++) {n2 for (
    1; j <= n; j++) {
                statement1;        statement4;
    }                          }
}
                               <= n;
for (int i = 1; i              i++) {
    statement2;
    statement3;
```

- How many statements will execute if n =

Algorithm growth rates

- We measure runtime in proportion to the input data s
  - **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 nex
  - The highest-order term (n^3) dominates the overall runt
  - We say that this algorithm runs ”in the order of" n^3
  - or **$O(n^3)$** for short ("**Big-Oh** of n^3")

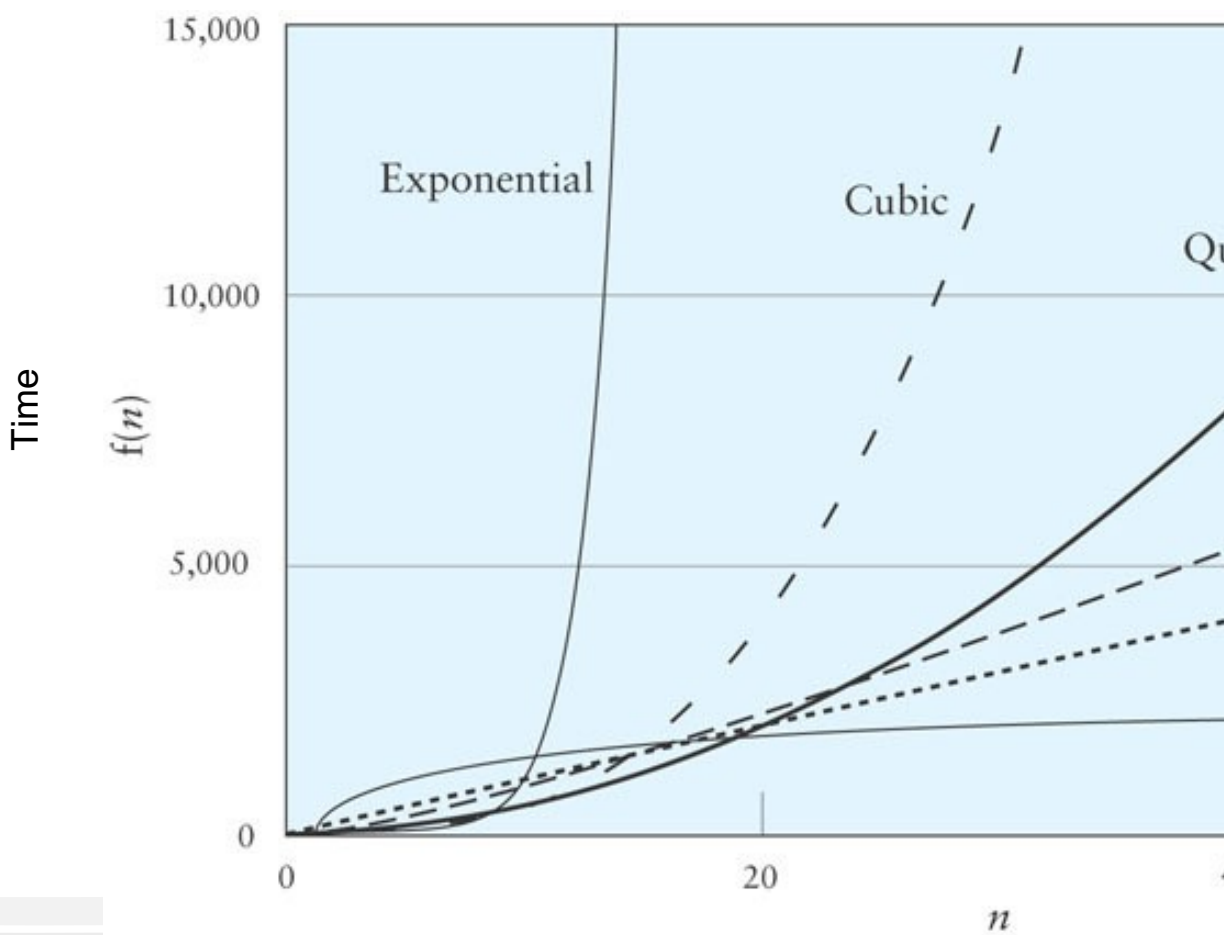
- **Big-Oh It's a measure of the longest amount of ti algorithm to complete (upper bound)**

Complexity classes

- **Complexity class**: A category of algorithm efficienc
  the input size n

Big-O	Name
$O(1)$	Consta
$O(\log n)$	Logarit
$O(n)$	Linear
$O(n \log n)$	Log-lin
$O(n^2)$	Quadra
$O(n^3)$	Cubic
$O(2^n)$	Expone
$O(n!)$	Factori

Practical Complexity


Time
15,000
10,000
5,000
0
f(n)
Exponential
Cubic
0
20
n

Input Size

Effect of Different Growth Rates

O(f(*n*))	f(50)	f(100)
O(1)	1	1
O(log *n*)	5.64	6.64
O(*n*)	50	100
O(*n* log *n*)	282	664
$O(n^2)$	2500	10,000
$O(n^3)$	12,500	100,000
$O(2^n)$	1.126×10^{15}	1.27×10^{30}
O(*n*!)	3.0×10^{64}	9.3×10^{157}

Collection efficiency

- Efficiency of various operations on `ArrayList`:

Method
`add`
`add(`**`index, value`**`)`
`indexOf`
`get`
`remove`
`set`
`size`

The Comparabl

Natural ordering

- Many types have a notion of a **natural ordering** that c
  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`?
  - `GroceryList`: What makes one list "less than" anothe

Uses of natural ordering

- An `ArrayList` of orderable values can be sorted using `C`

```
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 = " + w
Collections.sort(words);
System.out.println("after sort, words = " + wo
```

- Output:

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

Comparable interface

- The natural ordering of a class is specified through t

`Comparable` interface:

Relationship	Primitive comparison
less than	`if (x < y) {`
less than or equal	`if (x <= y) {`
equal	`if (x == y) {`
not equal	`if (x != y) {`

greater than	`if (x > y) {`
greater or equal	`if (x >= y) {`

```
public interface Co
    public int comp
}
```

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

How to compare objects?

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

```
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 illega

```
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 interface

- `Comparable` is a generic interface, `Comparable<T>`
- When implementing it, you must write your class's nam
- Example: `public class Point` **`implements Co`**
- You must also write a method `compareTo` that compa to a given other object
- Example:

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

Comparable example

```
public class Point implements Comparable<Poi
    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
  want `compareTo` to return:

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

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

Comparable implementation

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

public class CalendarDate implements Comparable<Caler
    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;
    }
}
```

Example Client Program

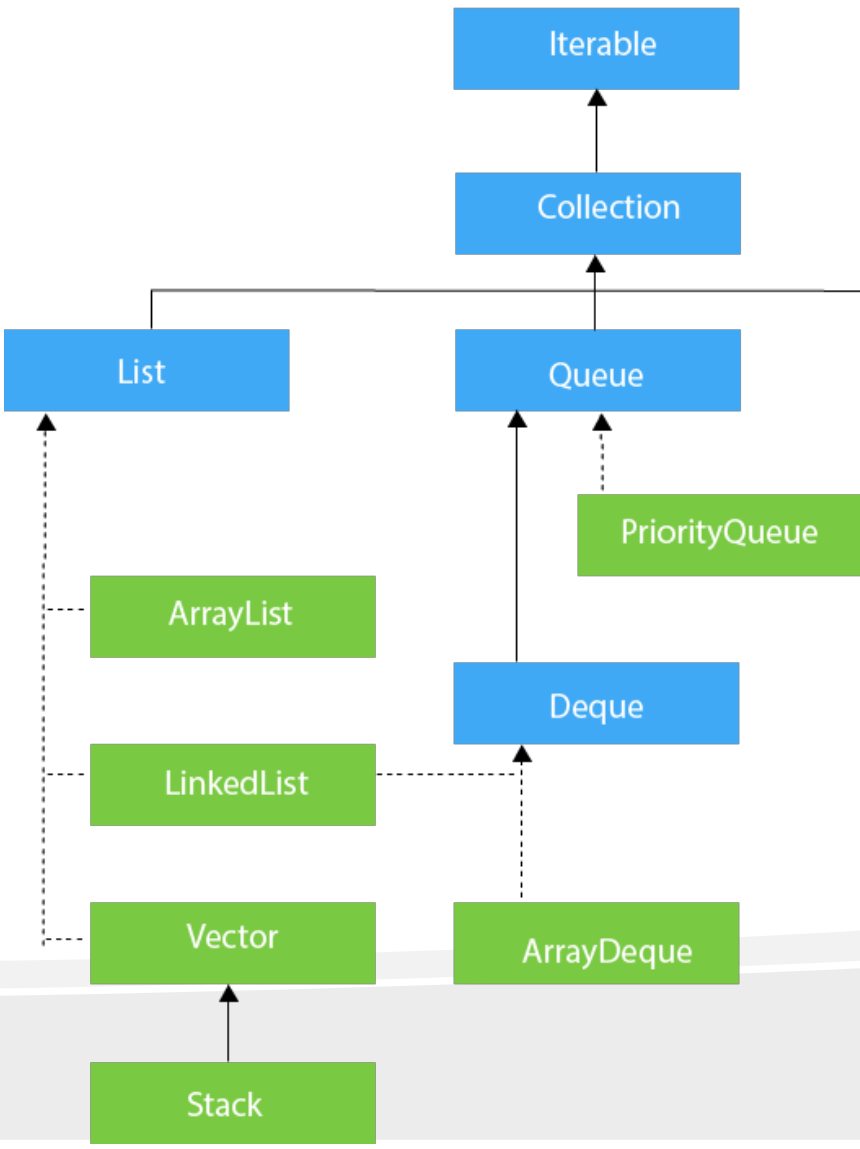
```
// Short program that creates a list of the birthdays of the
// US Presidents and that puts them into sorted order.

import java.util.*;

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

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

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]

Collectio

Collections Framework Diagram


Iterable
Collection
List
Queue
PriorityQueue
ArrayList
Deque
LinkedList
Vector
ArrayDeque
Stack

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, co
- Most collections are built with particular operations or

Collections

- A collection is an object that groups multiple ele

- Very useful
- Store, retrieve and manipulate data
- Transmit data from one method to another
- Data structures and methods written already for

Collection examples

- **List**: Collection of elements
- **Set**: A collection of elements that is guaranteed to can be searched efficiently
- **Map**: A collection of (key, value) pairs in which eac corresponding value
- **Stack**: A collection where the last element added is

- **Queue**: A collection where elements are remov

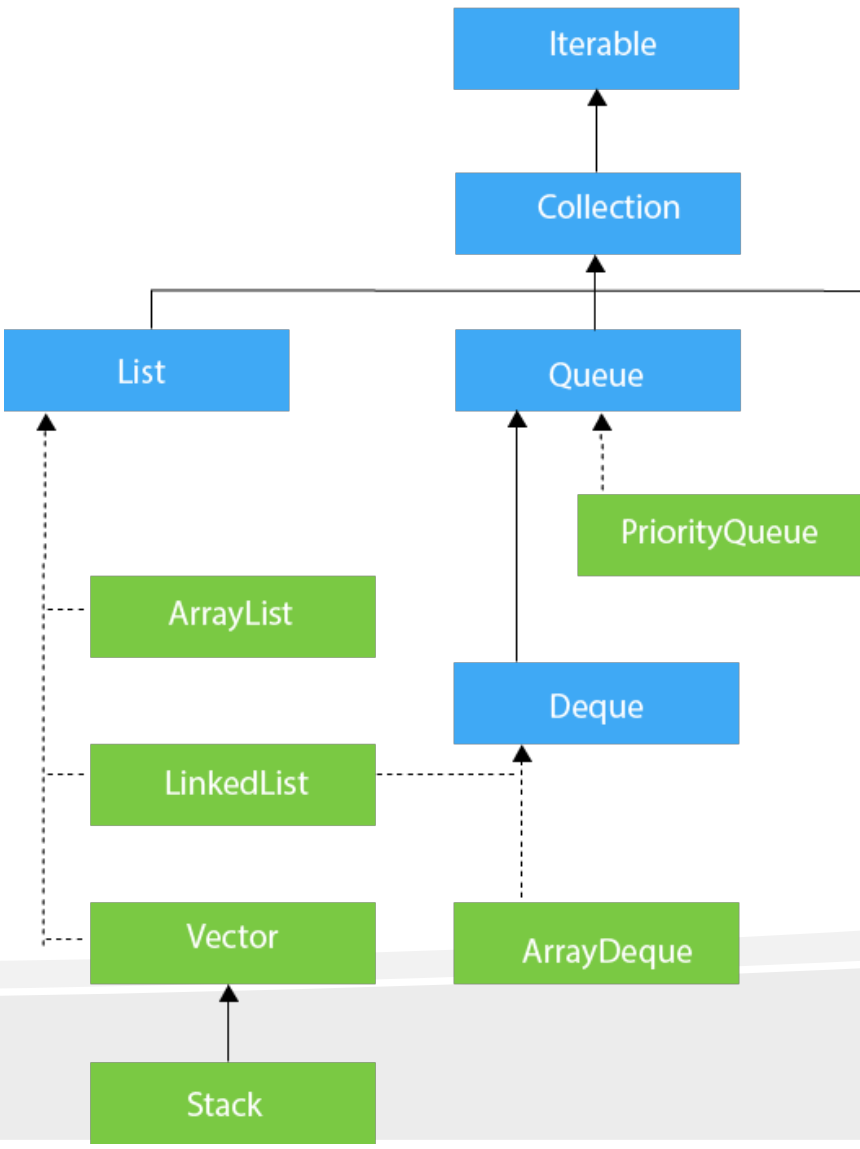
added

Collections Framework

- Unified architecture for representing/storing and ma

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

Collections Framework Diagram


Iterable
Collection
List
Queue
PriorityQueue
ArrayList
Deque
LinkedList
Vector
ArrayDeque
Stack

Java's `Collection` interface

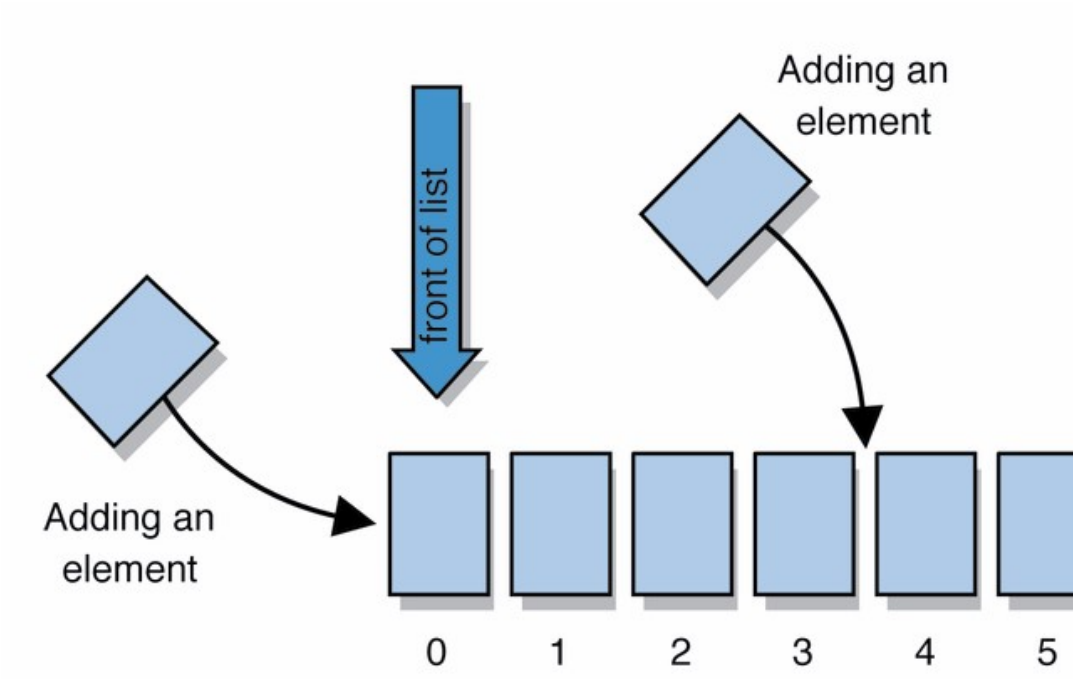
- `Collection<E>` represents many kinds of collec
following methods:

Method name	
`add(`*value*`)`	adds the g changed th
`addAll(`*collection*`)`	adds all ele
`clear()`	removes a
`contains(`*value*`)`	returns `tr`
`containsAll(`*collection*`)`	`true` if thi
`isEmpty()`	`true` if the
`removeAll(`*collection*`)`	removes a

`retainAll(`*collection*`)`	retains the collection
`remove(E element)`	removes the
`iterator()`	returns a sp
`size()`	returns the
`toArray()`	returns an a

List

- **List** is an ordered sequence of elements. One of th


Adding an element
front of list
Adding an element
0
1
2
3
4
5

List features

- Maintains elements in the order they were add
  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 l
  it

Java's `List` interface

- Java has an interface `List<E>` to represent a list

  those in `Collection<E>`:

- **`public void add(int index, E element)`**
  Inserts the specified element at the specified positio
- **`public E get(int index)`**

`List` interface, cont'd.

Returns the element at the specified position in this

- **`public int indexOf(E element)`**
  Returns the index in this list of the first occurrence o
  does not contain it
- ...

`List` interface, cont'd.

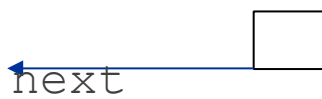
- **`public int lastIndexOf(Object o)`**
  Returns the index in this list of the last occurrence o
  does not contain it

- **`public E remove(int index)`**
  Removes the object at the specified position in this
- **`public E set(int index, E element)`**
  Replaces the element at the specified position in th
- Notice that the methods added to `Collection<E>`
- **A list has indexes while a general collection ma**

`ArrayList`

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

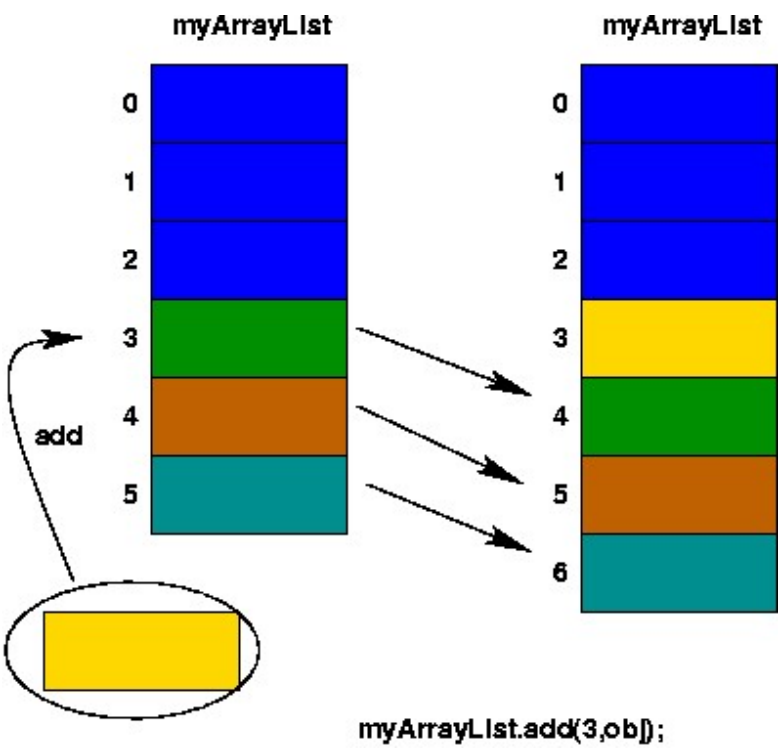
```
input: ArrayList<String> list
    int i=0;
    while (i < list.size()){ String element =
      (element.length()%2 == 0) { list.remov
        for large files
      else { i++; // skip to next
          element
      }
    }
```



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

`ArrayList` limitations

- An `add` or `remove` operation on an `ArrayLi` require elements to be shifted
- This can be slow for a large list
- What is the worst possible case?


myArrayList
myArrayList
0
1
2
3
4
5
0
1
2
3
4
5
6
add
myArrayList.add(3,obj);

Internals of `ArrayList`

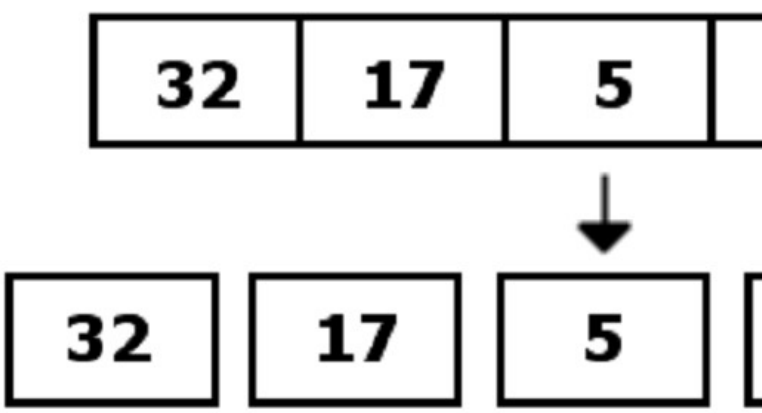
- It is internally stored in an array. So, it has a certa
- Capacity: size of the array used to store the elements i
- It is always at least as large as the list size
- As elements are added to an `ArrayList`, its cap
- Random access to elements
- `set/get` an element to/at specific index position has c
- To `add` at the end of the `ArrayList` it takes **O(1)**
- Remove, or add at a specified index operations ha

The underlying issue

- The elements of an `ArrayList` are too tightly att

them - Can we break the element storage apart into

structure?


32
17
5
32
17
5