



# CS 570: Data Structures

## Intro to Java

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# About me

- Prof. Iraklis Tsekourakis
  - Grew up in Kavala, Greece
  - Started programming in BASIC/MS-DOS in my last year of elementary school (1995)
  - Moved to the U.S. 7 years ago for my graduate studies

# About me

- Education
  - BS in ECE – Aristotle university of Thessaloniki (2010)
  - MS in CS – Stevens Institute of Technology (2014)
  - PhD in CS – Stevens Institute of Technology (2016)
- Professional Experience
  - Assistant Teaching Professor (2016-present)
  - Research Associate in CERTH/ITI (Information Technologies Institute) (2011-2012)
    - Software development, Project management, Research

# Teaching

- I've been an instructor on
  - Intro to Web Programming
  - Operating Systems
  - Algorithms
  - Data Structures

# Research

- Autonomous Software Agents
- **3-D Computer Vision**
- Machine Learning

# Objectives

- Review of Java language.
- Understand the notion of Abstract Data Types, and their use in object-oriented designs.
- Calculate the Big-O of diverse non-recursive algorithms and use it to compare efficiency.
- Use and understand the Collection class in Java, with major emphasis on Lists, Stacks and Queues.
- Implement Binary Search Trees, Max/Min Heaps, Priority Queues in Java, and understand the basic concepts of self-balancing Binary Search Trees.
- Understand what are Sets and Maps, and more specifically implement hash tables in Java.
- Understand and implement recursive algorithms, and data structures.
- Combine different classes together to implement big programming assignments in Java, including a final project that combines some of the data structures studied in class

# Important Points

- At any point, ask me WHY?
- USE the virtual office hours
  - If you think a homework is taking too long or is wrong
  - Etc. etc.
- You can ask me anything about the course in my virtual office hours, or by email
- This is a graduate course.
- Expect to work, and to be challenged

# Logistics

- Class webpage: CANVAS
  - <https://www.stevens.edu/canvas>
- Virtual Office hours: Thursday 6-7pm
  - [https://sit.instructure.com/courses/34684/external\\_tools/86566](https://sit.instructure.com/courses/34684/external_tools/86566)

# Logistics II

- Course Evaluation
  - Quizzes 10%
  - Assignments 60%
  - Final 30%

# Assignments

- Individual
- Will be done in Java (more details on that later)
- This is a Java Course
  - Programming is arguably the number one skill of a CS graduate
- But I care equally (or slightly more) for algorithms, and data structures than syntax

# Resources

- Textbook
  - Data Structures: Abstraction and Design Using Java, 3<sup>rd</sup> Edition
- Slides and Videos
  - On Canvas

# Syllabus

	<b>Topic(s)</b>	<b>Reading(s)</b>	<b>HW</b>
<b>Week 1</b>	Java Syntax	Appendix A	
<b>Week 2</b>	Java Syntax (II), ADT	Appendix A, Chapter 1	<b>Assignment on Java</b>
<b>Week 3</b>	Complexity	Chapter 2	
<b>Week 4</b>	Array based Lists, Linked Lists	Chapter 2	<b>Assignment on Complexity</b>
<b>Week 5</b>	Double Linked Lists, Iterators, Collections interface	Chapter 2	
<b>Week 6</b>	Stacks	Chapter 3	<b>Assignment on Double Linked Lists</b>
<b>Week 7</b>	Queues	Chapter 4	
<b>Week 8</b>	Recursion	Chapter 5	
<b>Week 9</b>	More Recursion	Chapter 5	<b>Assignment on Recursion</b>
<b>Week 10</b>	Introduction to Trees	Chapter 6	
<b>Week 11</b>	Binary Search Trees, max/min heaps, priority queues	Chapter 6	<b>Assignment on Priority Queues</b>
<b>Week 12</b>	Sets & Maps, Hashing	Chapter 7	
<b>Week 13</b>	Sets & Maps, Hashing	Chapter 7	<b>Final Programming Project</b>
<b>Week 14</b>	Sorting	Chapter 8	

# Grading Policies

- Late submission?
- Quizzes
- Practice coding exercises
- Individual Assignments: NO collaboration
  - Any sign of collaboration -> Honor Board
- Midterm surveys and not only: Provide feedback!

# Week 1 Objective

- Intro to Java
- Reading Assignment: Koffman and Wolfgang Appendix A
- **TODO: Install Eclipse**
- Also requires the Java Development Kit

# Java

- WORA: Write Once, Run Anywhere
- Java Virtual Machine
- Other advantages include security, extensibility and low software production cost

# Java is Object Oriented

- Classes and Objects
  - Class definitions in .java files
  - Def: a *class* is a named description for a group of entities that have the same characteristics
  - *Objects* or *instances* of the class is the group of entities
  - The characteristics are the attributes (**data fields**) for each object and the operations (**methods**) that can be performed on these objects

# Data Fields and Types

- Data fields are variables
- Java is **strongly typed**
  - Every variable must be **declared** with a data type before it can be used
  - There are built-in types and user-defined types

# Primitive Data Types

- byte -128 to 127
- short -32,768 to 32,767
- int -2,147,483,648 to 2,147,483,647
- long -9,223,372,036,854,775,808 to ...
- float  $\pm 10^{38}$  incl. 0 with 6 digits of precision
- double  $\pm 10^{308}$  incl. 0 with 15 digits of precision
- char Unicode character set
- boolean true, false

# Methods

- **Method:** a group of statements to perform a particular operation (called function in many other languages)
- **Instance Methods:** Applied to an object using dot notation
- `object.method(arguments)`
- E.g. the `println` method that can be applied to `PrintStream` **object** `System.out`
  - `System.out.println("The value of x is "+x);`

# Static Methods

- static char minChar(char ch1, char ch2) {
- **static** indicates that it is a *static or class method*
  - There is one per class, not one per object like instance methods
- **Called using dot notation**
  - char ch=ClassName.minChar('a','A');
- **Static methods cannot call instance methods**

# Static vs. Instance Methods

```
public class Car{  
...  
?? float km2Miles(float km)  
?? float getOdometerMiles()  
}
```

# The main Method

- Point where execution begins

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

public:

static:

void:

# Defining your own classes

- A Java program is a collection of classes
- Let's see some examples

# Rectangle Example

```
public class Rectangle{  
    public double width;  
    public double height;  
  
    // constructor  
    public Rectangle(double x, double y) {  
        width = x;  
        height = y;  
    }  
  
    public double area() {  
        return width*height;  
    }  
}
```

# Rectangle Example

```
// int main() method  
// create a rectangle with width 3.5 and height 2.6  
Rectangle rect = new Rectangle(3.5, 2.6);  
  
// get its area  
double ar;  
ar = rect.area();
```

# Person Example

- For example:
  - A class Person may store:
    - Given name
    - Family name
    - ID number
    - Year of birth
  - It can perform operations such as:
    - Calculate person's age
    - Test whether two Person objects refer to same person
    - Determine if the person is old enough to vote
    - Get one or more of the data fields from the Person object
    - Set one or more of the data fields of the Person object

# UML Diagram

Person
String givenName
String familyName
String IDNumber
int birthYear
int age()
boolean canVote()
boolean isSenior()

Data fields (instance variables)

Methods

```
/** Person is a class that represents a human being.  
 *  @author Koffman and Wolfgang  
 * */  
  
public class Person {  
    // Data Fields  
    /** The given name */  
    private String givenName;  
  
    /** The family name */  
    private String familyName;  
  
    /** The ID number */  
    private String IDNumber;  
  
    /** The birth year */  
    private int birthYear = 1900;  
  
    // Constants  
    /** The age at which a person can vote */  
    private static final int VOTE_AGE = 18;  
  
    /** The age at which a person is considered a senior citizen */  
    private static final int SENIOR_AGE = 65;
```

```
// Constructors

/** Construct a person with given values
 * @param first The given name
 * @param family The family name
 * @param ID The ID number
 * @param birth The birth year
 */
public Person(String first, String family, String ID, int birth) {
    givenName = first;
    familyName = family;
    IDNumber = ID;
    birthYear = birth;
}

/** Construct a person with only an IDNumber specified.
 * @param ID The ID number
 */
public Person(String ID) {
    IDNumber = ID;
}
```

```
// Modifier Methods

/** Sets the givenName field.
 * @param given The given name
 */
public void setGivenName(String given) {
    givenName = given;
}

/** Sets the familyName field.
 * @param family The family name
 */
public void setFamilyName(String family) {
    familyName = family;
}

/** Sets the birthYear field.
 * @param birthYear The year of birth
 */
public void setBirthYear(int birthYear) {
    this.birthYear = birthYear;
}
```

```
// Accessor Methods

/** Gets the person's given name.
 * @return the given name as a String
 */
public String getGivenName() {
    return givenName;
}

/** Gets the person's family name.
 * @return the family name as a
String
 */
public String getFamilyName() {
    return familyName;
}

/** Gets the person's ID number.
 * @return the ID number as a String
 */
public String getIDNumber() {
    return IDNumber;
}

/** Gets the person's year of birth.
```

```
     * @return the year of birth as an
int value
    */
public int getBirthYear() {
    return birthYear;
}
```

```
// Other Methods
/** Calculates a person's age at this year's birthday.
 * @param year The current year
 * @return the year minus the birth year
 */
public int age(int year) {
    return year - birthYear;
}

/** Determines whether a person can vote.
 * @param year The current year
 * @return true if the person's age is greater than or
 *         equal to the voting age
 */
public boolean canVote(int year) {
    int theAge = age(year);
    return theAge >= VOTE_AGE;
}
```

```
/** Determines whether a person is a senior citizen.  
 * @param year the current year  
 * @return true if person's age is greater than or  
 *         equal to the age at which a person is  
 *         considered to be a senior citizen  
 */  
public boolean isSenior(int year) {  
    return age(year) >= SENIOR_AGE;  
}  
  
/** Retrieves the information in a Person object.  
 * @return the object state as a string  
 */  
public String toString() {  
    return "Given name: " + givenName + "\n"  
        + "Family name: " + familyName + "\n"  
        + "ID number: " + IDNumber + "\n"  
        + "Year of birth: " + birthYear + "\n";  
}
```

```
/** Compares two Person objects for equality.  
 * @param per The second Person object  
 * @return true if the Person objects have same  
 *         ID number; false if they don't  
 */  
public boolean equals(Person per) {  
    if (per == null)  
        return false;  
    else  
        return IDNumber.equals(per.IDNumber);  
}  
}
```

# Private Data Fields, Public Methods

- Better control of how data is accessed
- Details of how data are stored and represented can be changed without affecting class's clients

# Constructors

- Four-parameter
- One-parameter
- No-parameter constructor is not defined
- Person p = new Person() is invalid
- No-parameter constructor has to be explicitly defined if other constructors are defined

# Use of this.

```
public void setBirthYear(int birthYear) {  
    this.birthYear = birthYear;  
}
```

- `birthYear` is interpreted by the Java compiler as the local variable (parameter here) and not the data field with the same name

# The Method `toString`

- To display the state of `author1` (an instance of `Person`), we could use:
  - `System.out.println(author1.toString());`
  - `System.out.println(author1);`
- `System.out.println` and `System.out.print` automatically apply method `toString()` to an object that appears in their argument list

# The Method equals

```
public boolean equals(Person per) {  
    if (per == null)  
        return false;  
    else  
        return IDNumber.equals(per.IDNumber);  
}
```

- We can look at `per`'s private ID number because `per` references an object of this class (`Person`)

# testPerson

```
public class TestPerson {  
    public static void main(String[] args) {  
        Person p1 = new Person("Sam", "Jones", "1234", 1930);  
        Person p2 = new Person("Jane", "Jones", "5678", 1990);  
        System.out.println("Age of " + p1.getGivenName() +  
                           " is " + p1.age(2012));  
        if (p1.isSenior(2004))  
            System.out.println(p1.getGivenName() +  
                                " can ride the subway for free");  
        else  
            System.out.println(p1.getGivenName() +  
                                " must pay to ride the subway");  
  
        System.out.println("Age of " + p2.getGivenName() +  
                           " is " + p2.age(2012));  
        if (p2.canVote(2004))  
            System.out.println(p2.getGivenName() + " can vote");  
        else  
            System.out.println(p2.getGivenName() + " can't vote");  
    }  
}
```

# Arrays

```
int[] scores = new int[5];
String[] names = {"Sally", "Jill", "Hal",
    "Rick"};
```

```
Person[] people;
// define n in some way
int n = ...
people = new Person[n];
people[0] = new Person("Elliot",
    "Koffman", "123", 1942);
```

length is a data field not a method

# Arrays of Arrays

```
double[][] matrix = new double[5][10];
```

- In Java, you can have two-dimensional arrays with rows of different sizes

```
char[][] letters = new char[5][];
```

```
letters[0] = new char[4];
```

```
letters[1] = new char[10];
```

# Style

- Camel notation
  - myVariable, thisLongIdentifier
- Primitive type constants
  - all caps: static final int MAX\_SCORE=999
- Postfix/prefix increment
  - z=i++;
  - z=++i;
  - Don't use x\*++i

# Pre-increment VS post-increment

- `++i` will increment the value of `i`, and then return the incremented value.

```
i = 1;  
j = ++i;  
(i is 2, j is 2)
```

- `i++` will increment the value of `i`, but return the original value that `i` held before being incremented.

```
i = 1;  
j = i++;  
(i is 2, j is 1)
```

# Type Compatibility and Conversion

- When mixed type operands are used, the type with the smaller range is converted to the type of the larger range
  - E.g. int+double is converted to double
  - *Widening* conversion
  - ```
int item = ...;
double realItem = item; // valid ?
```
  - ```
double y = ...;
int x=y; // valid ?
```

# Referencing Objects

- String greeting;
- greeting = "hello";
  - String object “hello” is now referenced by greeting
  - greeting stores the **address** of where a particular String is stored.
- **Primitive types store values not addresses**
  - x=3;

# References

- Two reference variables can reference the same object
  - `String welcome=greeting;`
  - copies the address in `greeting` to `welcome`
- Creating new objects
  - `String keyboard = new String("qwerty");`

# Self-Check

- String y=new String("abc");
- String z="def";
- String w=z;

# Control Statements

- if ... else
- switch
- while
- do ... while
- for

# Examples

- Compute the sum of all even numbers from 2 to 200 unless they are multiples of 7
- Come up with a natural do ... while example

# Calling by Value

- In Java all arguments are **call-by-value**
  - If the argument is a primitive type, its value, not its address, are passed to the method
  - The method cannot modify the argument value and have this modification remain after returning
  - If the argument is of class type, it can be modified using its own methods and the changes are permanent
- Other languages also support call-by-reference

# The Math Class

- Collection of useful methods
- All static

```
public class SquareRoots {  
    public static void main(String[] args) {  
        System.out.println("n \tsquare root");  
        for (int n = 1; n <= 10; n++) {  
            System.out.println(n + "\t" +  
                Math.sqrt(n));  
        }  
    }  
}
```

# The String Class

- Assume keyboard is a String that contains “qwerty”

keyboard.charAt(0)

keyboard.length()

keyboard.indexOf('o')

keyboard.indexOf('y')

String upper=keyboard.toUpperCase();

Creates a new string object without changing  
keyboard

# Strings are Immutable

- Strings are different from other objects in that they are immutable
  - A String object cannot be modified
  - New Strings are generated when changes are made

```
String myName = "Elliot Koffman";
myName = myName.substring(7) + ", " +
    myName.substring(0, 6);
```

```
myName[0] = 'X'; // invalid
myName.charAt(0) = 'X'; // invalid
```

# Comparing Objects

```
String anyName = new String(myName);  
anyName == myName ?
```

- `==` operator compares the addresses and not the contents of the objects
- Use `equals`, `equalsIgnoreCase`, `compareTo`, `compareToIgnoreCase`
- Comparison methods need to be implemented for user-defined classes

# Wrapper Classes for Primitive Types

- Primitive numeric types are not objects, but sometimes they need to be processed like objects
  - When?
- Java provides *wrapper classes* whose objects contain primitive-type values
  - `Float`, `Double`, `Integer`, `Boolean`, `Character`
  - They provide constructor methods to create new objects that “wrap” a specified value
  - Also provide methods to “unwrap”

# Autoboxing/Unboxing

- Before Java 5.0, int values and Integer objects could not be mixed

```
int n = nInt.intValue();
```

```
nInt = new Integer(n++);
```

- Java 5.0 introduced autoboxing/unboxing

```
int n = nInt;
```

```
nInt = n++;
```

or

```
nInt++;
```

# Examples

```
Integer i1=35;  
Integer i2=1234;  
Integer i3=i1+i2;  
int i2Val=i2++;  
int i3Val=Integer.parseInt("-357");  
Integer i4= Integer.valueOf(753);  
System.out.println(i1);
```

# More Examples

- `System.out.println(i1+i2);`
- `System.out.println(i1.toString()  
+i2.toString());`
- Operations are the same as primitive types  
in current version of java
  - autoboxes before the operator is applied.
- What happens for the == operator?

# Practice: Tic Tac Toe

- The 9 cells on the board are numbered
- Read integers from 1 to 9 from text file
  - Each move is given by the cell number the player wishes to occupy
  - Players alternate starting with X, so the ID of the current player is not given
- -1 denotes the end of the game

0	1	2
3	4	5
6	7	8

# Requirements

1. If the number is -1, reset the board and start a new game.
2. Check that the move is legal, i.e. the cell is not occupied.
3. Check if there is a winner. There are multiple ways to do this.
4. Check if there is a tie.



# CS 570: Data Structures Java & Object-Oriented Design (Chapter 1)

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



## Six Stages of Debugging

- 1.** That can't happen.
- 2.** That doesn't happen on my machine.
- 3.** That shouldn't happen.
- 4.** Why does that happen?
- 5.** Oh, I see.
- 6.** How did that ever work?

# Textbook Companion Site

3

- <http://bcs.wiley.com/he-bcs/Books?action=index&itemId=0471692646&itemTypeld=BKS&bcsId=2200>
- Google: Koffman and Wolfgang student companion site
  - Source code
  - Solutions to self check problems

# Chapter Objectives

4

- Interfaces
- Inheritance and code reuse
- How Java determines which method to execute when there are multiple methods
- Abstract classes
- Abstract data types and interfaces
- Object class and overriding Object class methods
- Exception hierarchy
- Packages and visibility
- Class hierarchy for shapes

# Week 2

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 1.1-1.5

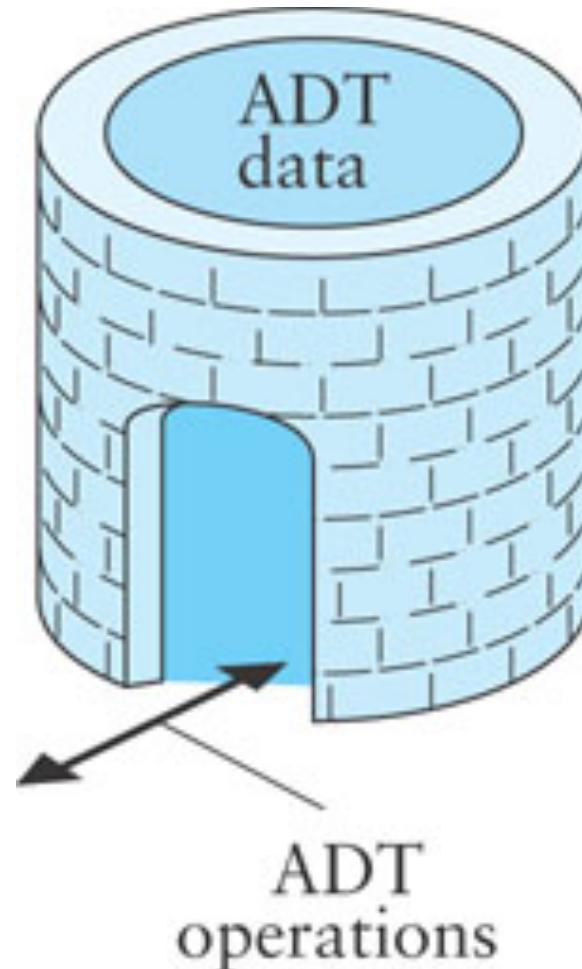
# ADTs, Interfaces, and the Java API

## Section 1.1

# ADTs

7

- Abstract Data Type (ADT)
- An encapsulation of data and methods
- Allows for reusable code
- The user need not know about the implementation of the ADT
- A user interacts with the ADT using only public methods



# ADTs (cont.)

8

- ADTs facilitate storage, organization, and processing of information
- Such ADTs often are called *data structures*
- The Java Collections Framework provides implementations of common data structures

# Interfaces

9

- An interface specifies or describes an ADT to the applications programmer:
  - the methods and the actions that they must perform
  - what arguments, if any, must be passed to each method
  - what result the method will return
- The interface can be viewed as a *contract* which guarantees how the ADT will function

# Interfaces (cont.)

10

- A class that *implements the interface* provides code for the ADT
- As long as the implementation satisfies the ADT contract, the programmer may implement it as he or she chooses
- In addition to implementing all data fields and methods in the interface, the programmer may add:
  - data fields not in the interface
  - methods not in the interface
  - constructors (an interface cannot contain constructors because it cannot be instantiated)

# Example: ATM Interface

11

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must provide operations to:
  - verify a user's Personal Identification Number (PIN)
  - allow the user to choose a particular account
  - withdraw a specified amount of money
  - display the result of an operation
  - display an account balance
- A class that implements an ATM must provide a method for each operation

# Example: ATM Interface (cont.)

12

## Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - ▣ verify a user's Personal Identification Number (PIN)
  - ▣ allow the user to choose a particular account
  - ▣ withdraw a specified amount of money
  - ▣ display the result of an operation
  - ▣ display an account balance

## Code

```
public interface ATM {  
}
```

The keyword `interface` in the header indicates that an interface is being declared

# Example: ATM Interface (cont.)

13

## Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - ▣ verify a user's Personal Identification Number (PIN)
  - ▣ allow the user to choose a particular account
  - ▣ withdraw a specified amount of money
  - ▣ display the result of an operation
  - ▣ display an account balance

## Code

```
public interface ATM {  
  
    /** Verifies a user's PIN.  
     * @param pin The user's PIN  
     */  
    boolean verifyPIN(String pin);  
}
```

# Example: ATM Interface (cont.)

14

## Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - ▣ verify a user's Personal Identification Number (PIN)
  - ▣ allow the user to choose a particular account
  - ▣ withdraw a specified amount of money
  - ▣ display the result of an operation.
  - ▣ display an account balance

## Code

```
public interface ATM {  
  
    /** Verifies a user's PIN.  
     * @param pin The user's PIN  
     */  
    boolean verifyPIN(String pin);  
  
    /** Allows the user to select an  
     * account.  
     * @return a String representing  
     * the account selected  
     */  
    String selectAccount();  
  
}
```

# Example: ATM Interface (cont.)

15

## Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - ▣ verify a user's Personal Identification Number (PIN)
  - ▣ allow the user to choose a particular account
  - ▣ withdraw a specified amount of money
  - ▣ display the result of an operation.
  - ▣ display an account balance

## Code

```
/** Withdraws a specified amount  
 * of money  
  
 * @param account The account  
 * from which the money  
 * comes  
  
 * @param amount The amount of  
 * money withdrawn  
  
 * @return whether or not the  
 * operation is  
 * successful  
  
 */  
  
boolean withdraw(String account,  
                 double amount);  
  
}
```

# Example: ATM Interface (cont.)

16

## Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - ▣ verify a user's Personal Identification Number (PIN)
  - ▣ allow the user to choose a particular account
  - ▣ withdraw a specified amount of money
  - ▣ display the result of an operation
  - ▣ display an account balance

## Code

```
/** Displays the result of an
 *  operation
 *
 * @param account The account
 *                 from which money was
 *                 withdrawn
 *
 * @param amount The amount of
 *               money withdrawn
 *
 * @param success Whether or not
 *                the withdrawal took
 *                place
 */
void display(String account,
            double amount,
            boolean success);
```

# Example: ATM Interface (cont.)

17

## Interface

- An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations:
  - ▣ verify a user's Personal Identification Number (PIN)
  - ▣ allow the user to choose a particular account
  - ▣ withdraw a specified amount of money
  - ▣ display the result of an operation.
  - ▣ display an account balance

## Code

```
/** Displays an account balance
 * @param account The account
 * selected
 */
void showBalance(String
account);
```

# Interfaces (cont.)

18

- The interface definition shows only headings for its methods
- Because only headings are shown, they are considered *abstract methods*
- Each abstract method must be defined in a class that implements the interface

# Interface Definition

19

FORM:

```
public interface interfaceName {  
    abstract method headings  
    constant declarations  
}
```

- Constants are defined in the interface
- DEDUCTIONS is accessible in classes that implement the interface

EXAMPLE:

```
public interface Payable {  
    public abstract double calcSalary();  
    public abstract boolean salaried();  
    public static final  
        double DEDUCTIONS = 25.5;  
}
```

# Interface Definition (cont.)

20

FORM:

```
public interface interfaceName {  
    abstract method headings  
    constant declarations  
}
```

EXAMPLE:

```
public interface Payable {  
    public abstract double calcSalary();  
    public abstract boolean salaried();  
    public static final  
        double DEDUCTIONS = 25.5;  
}
```

- The keywords **public** and **abstract** are implicit in each **abstract method definition**
- And keywords **public** **static** **final** are implicit in each **constant declaration**
- As such, they may be omitted

# The implements Clause

21

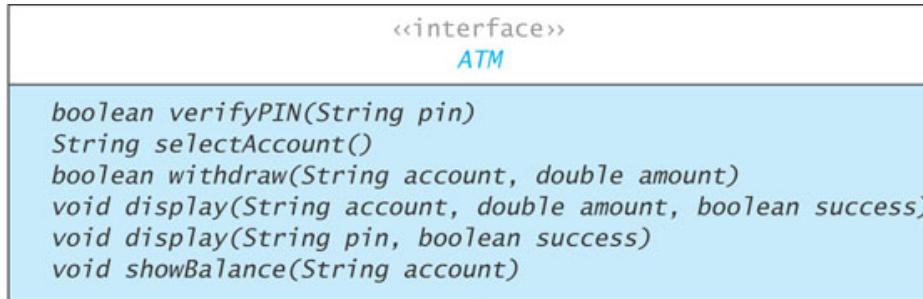
- For a class to implement an interface, it must end with the implements clause

```
public class ATMbankAmerica implements ATM  
public class ATMforAllBanks implements ATM
```

- A class may implement more than one interface—  
their names are separated by commas

# UML Diagram of Interface & Implementers

22



ATMbankAmerica

```
boolean verifyPIN(String pin)
String selectAccount()
boolean withdraw(String account, double amount)
void display(String account, double amount, boolean success)
void display(String pin, boolean success)
void showBalance(String account)
```

ATMforAllBanks

```
boolean verifyPIN(String pin)
String selectAccount()
boolean withdraw(String account, double amount)
void display(String account, double amount, boolean success)
void display(String pin, boolean success)
void showBalance(String account)
```

# The implements Clause: Pitfalls

23

- The Java compiler verifies that a class defines all the abstract methods in its interface(s)
- A syntax error will occur if a method is not defined or is not defined correctly:

Class ATMforAllBanks should be declared abstract; it does not define method verifyPIN(String) in interface ATM

- If a class contains an undefined abstract method, the compiler will require that the class be declared an abstract class

# The implements Clause: Pitfalls (cont.)

24

- You cannot instantiate an interface:

```
ATM anATM = new ATM();      // invalid statement
```

- Doing so will cause a syntax error:

```
interface ATM is abstract; cannot be instantiated
```

# Declaring a Variable of an Interface Type

25

- While you cannot instantiate an interface, you can declare a variable that has an interface type

```
/* expected type */
```

```
ATMbankAmerica ATM0 = new ATMBankAmerica();
```

```
/* interface type */
```

```
ATM ATM1 = new ATMBankAmerica();
```

```
ATM ATM2 = new ATMforAllBanks();
```

- The reason for wanting to do this will become clear when we discuss *polymorphism*

# Introduction to Object-Oriented Programming

## Section 1.2

# Object-Oriented Programming

27

- Object-oriented programming (OOP) is popular because:
  - it enables *reuse* of previous code saved as *classes*
  - saves time because previously written code has been tested and debugged already
- If a new class is similar to an existing class, the existing class can be extended
- This extension of an existing class is called *inheritance*

# Inheritance

28

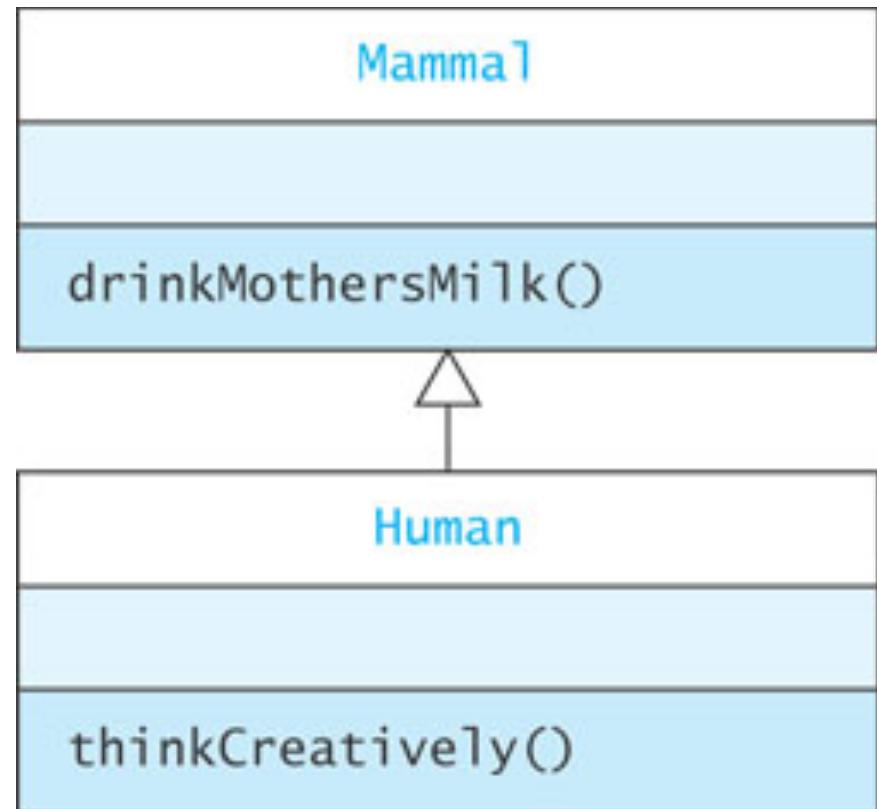
- A Human *is* a Mammal
- Human has all the data fields and methods defined by Mammal
- Mammal is the *superclass* of Human
- Human is a *subclass* of Mammal
- Human may define other variables and methods that are not contained in Mammal



# Inheritance (cont.)

29

- **Mammal has only method**  
`drinkMothersMilk()`
- **Human has method**  
`drinkMothersMilk()`  
**and** `thinkCreatively()`
- **Objects lower in the hierarchy are generally more powerful than their superclasses because of additional attributes**



# A Superclass and Subclass Example

30

- Computer
- A computer has
  - ▣ manufacturer
  - ▣ processor
  - ▣ RAM
  - ▣ disk

```
Computer
String manufacturer
String processor
int ramSize
int diskSize
double processorSpeed
```



# A Superclass and Subclass Example (cont.)

31

```
Computer

String manufacturer
String processor
int ramSize
int diskSize
double processorSpeed

int getRamSize()
int getDiskSize()
double getProcessorSpeed()
Double computePower()
String toString()
```



# A Superclass and Subclass Example

## (cont.)

32

- Notebook
- A Notebook has all the properties of Computer,
  - manufacturer
  - processor
  - RAM
  - Disk
- plus,
  - screen size
  - weight



# A Superclass and Subclass Example (cont.)

33

## Computer

```
String manufacturer  
String processor  
int ramSize  
int diskSize  
double processorSpeed
```

```
int getRamSize()  
int getDiskSize()  
double getProcessorSpeed()  
Double computePower()  
String toString()
```



## Notebook

```
double screenSize  
double weight
```

# A Superclass and Subclass Example

## (cont.)

34

- The constructor of a subclass begins by initializing the data fields inherited from the superclass(es)

```
super(man, proc, ram, disk, procSpeed);
```

which invokes the superclass constructor with the signature

```
Computer(String man, String processor, double ram,  
        int disk, double procSpeed)
```

# A Superclass and Subclass Example

## (cont.)

35

```
/** Class that represents a computer */
public class Computer {
    // Data fields
    private String manufacturer;
    private String processor;
    private double ramSize;
    private int diskSize;
    private double processorSpeed;
```

# A Superclass and Subclass Example

## (cont.)

36

```
// Methods

/** Initializes a Computer object with all properties specified.

@param man The computer manufacturer
@param processor The processor type
@param ram The RAM size
@param disk The disk size
@param procSpeed The processor speed

*/
public Computer(String man, String processor, double ram, int disk,
                double procSpeed) {
    manufacturer = man;
    this.processor = processor;
    ramSize = ram;
    diskSize = disk;
    processorSpeed = procSpeed;
}
```

# A Superclass and Subclass Example (cont.)

37

```
// Methods
/** Initializes a Computer object with a
 * @param man The computer manufacturer
 * @param processor The processor type
 * @param ram The RAM size
 * @param disk The disk size
 * @param procSpeed The processor speed
 */
public Computer(String man, String processor,
                 double procSpeed) {
    manufacturer = man;
    this.processor = processor;
    ramSize = ram;
    diskSize = disk;
    processorSpeed = procSpeed;
}
```

## Use of this

If you wrote this line as

```
processor = processor;
```

you would simply copy the variable processor to itself. To access the field, you need to prefix this:

```
this.processor = processor;
```

# A Superclass and Subclass Example

## (cont.)

38

```
public double computePower() { return ramSize * processorSpeed; }
public double getRamSize() { return ramSize; }
public double getProcessorSpeed() { return processorSpeed; }
public int getDiskSize() { return diskSize; }
// insert other accessor and modifier methods here

public String toString() {
    String result = "Manufacturer: " + manufacturer +
        "\nCPU: " + processor +
        "\nRAM: " + ramSize + " megabytes" +
        "\nDisk: " + diskSize + " gigabytes" +
        "\nProcessor speed: " + processorSpeed +
        " gigahertz";
    return result;
}
```

# A Superclass and Subclass Example

## (cont.)

39

```
/** Class that represents a notebook computer */
public class Notebook extends Computer {
    // Data fields
    private double screenSize;
    private double weight;

    . . .

}
```

# A Superclass and Subclass Example

## (cont.)

40

```
// methods  
/* Initializes a Notebook object with all properties specified.  
 @param man The computer manufacturer  
 @param processor The processor type  
 @param ram The RAM size  
 @param disk The disk size  
 @param procSpeed The processor speed  
 @param screen The screen size  
 @param wei The weight  
 */  
public Notebook(String man, String processor, double ram, int disk,  
                double procSpeed, double screen, double wei) {  
    super(man, proc, ram, disk, procSpeed);  
    screenSize = screen;  
    weight = wei;  
}
```

# The No-Parameter Constructor

41

- If the execution of any constructor in a subclass does not invoke a superclass constructor—an explicit call to `super()`—Java automatically invokes the no-parameter constructor for the superclass
- If no constructors are defined for a class, the no-parameter constructor for that class is provided by default
- However, if any constructors are defined, you must explicitly define a no-parameter constructor

# Protected Visibility for Superclass Data Fields

42

- Variables with *private visibility* cannot be accessed by a subclass
- Variables with *protected visibility* (defined by the keyword `protected`) are accessible by any subclass or any class in the same package
- In general, it is better to use *private visibility* and to restrict access to variables to accessor methods

# **Is-a versus Has-a Relationships**

43

- In an *is-a* or *inheritance* relationship, one class is a subclass of the other class
- In a *has-a* or *aggregation* relationship, one class has the other class as an attribute

# Is-a versus Has-a Relationships (cont.)

44

```
public class Computer {  
    private Memory mem;  
    ...  
}  
  
public class Memory {  
    private int size;  
    private int speed;  
    private String kind;  
    ...  
}
```

A **Computer** has only one  
**Memory**

But a **Computer** is not a  
**Memory** (i.e. not an *is-a*  
relationship)

If a **Notebook** extends  
**Computer**, then the  
**Notebook** *is-a Computer*

# Method Overriding, Method Overloading, and Polymorphism

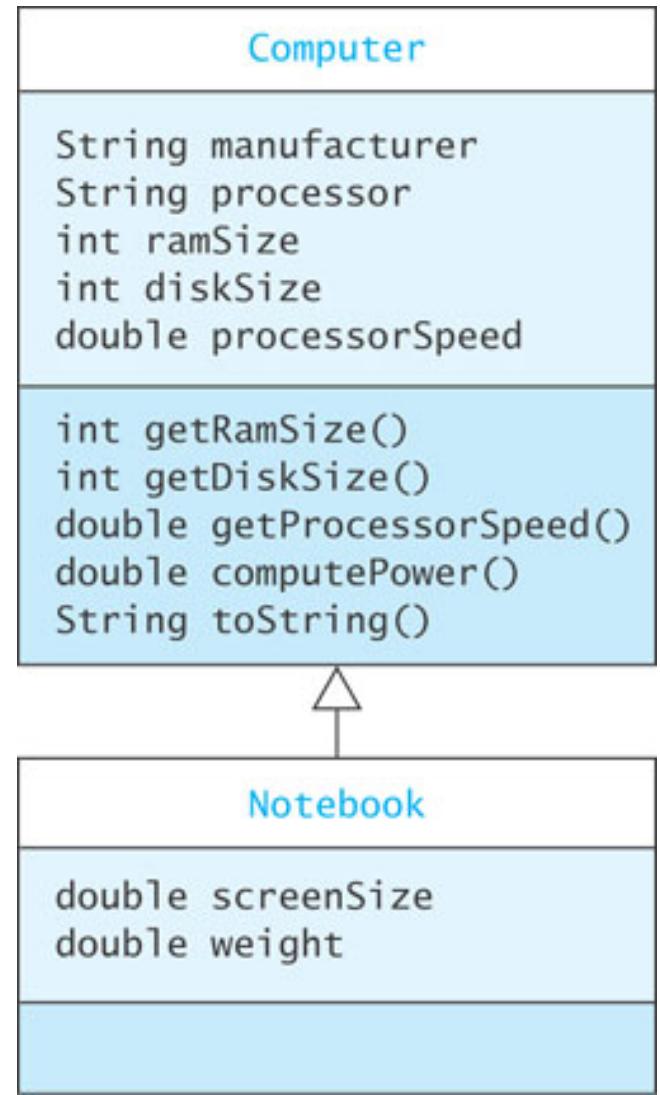
## Section 1.3

# Method Overriding

46

- Continuing the previous example, if we run:

```
Computer myComputer = new  
    Computer("Acme", "Intel", 2, 160,  
    2.4);  
  
Notebook yourComputer = new  
    Notebook("DellGate", "AMD", 4, 240,  
    1.8, 15.0, 7.5);  
  
System.out.println("My computer is:\n" +  
    myComputer.toString());  
  
System.out.println("Your computer is:\n" +  
    yourComputer.toString());
```



# Method Overriding (cont.)

47

- the output would be:

My Computer is:

Manufacturer: Acme

CPU: Intel

RAM: 2.0 gigabytes

Disk: 160 gigabytes

Speed: 2.4 gigahertz

Your Computer is:

Manufacturer: DellGate

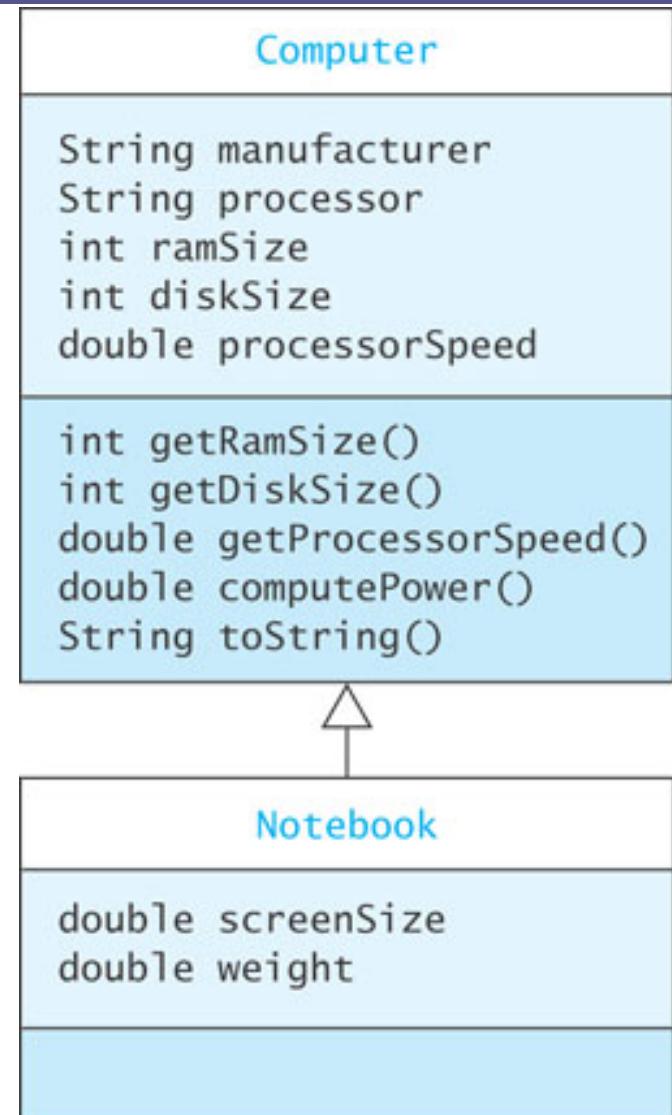
CPU: AMD

RAM: 4.0 gigabytes

Disk: 240 gigabytes

Speed: 1.8 gigahertz

- The screensize and weight variables are not printed because Notebook has not defined a `toString()` method



# Method Overriding (cont.)

48

- To define a `toString()` for `Notebook`:

```
public String toString() {  
    String result = super.toString() +  
        "\nScreen size: " +  
        screenSize + " inches" +  
        "\nWeight: " + weight +  
        " pounds";  
  
    return result;  
  
}
```

- Now `Notebook`'s `toString()` method will **override** `Computer`'s **inherited** `toString()` method and will be called for all `Notebook` objects

# Method Overriding (cont.)

49

- To define a `toString()` for `Notebook`:

```
public String toString() {  
    String result = super.toString() +  
        "\nScreen size: " +  
        screenSize + " inches" +  
        "\nWeight: " + weight +  
        " pounds";  
    return result +  
        super.methodName()  
}
```

- Note: Using the prefix `super` in a call to a method `methodName` calls the method with that name in the superclass of the current class  
be called for all `NOTEBOOK` objects

# Method Overloading (cont.)

50

- Methods in the class hierarchy which have the same name, return type, and parameters **override** corresponding inherited methods
- Methods with the same name but different parameters are **overloaded**

# Method Overloading (cont.)

51

- Take, for example, our Notebook constructor:

```
public Notebook(String man, String processor, double ram, int  
disk, double procSpeed, double screen, double wei) {  
    . . .  
}
```

- If we want to have a default manufacturer for a Notebook, we can create a constructor with six parameters instead of seven

```
public Notebook(String processor, double ram, int disk,  
                double procSpeed, double screen, double wei) {  
    this(DEFAULT_NB_MAN, double ram, int disk, double procSpeed,  
          double screen, double wei);
```

# Method Overloading: Pitfall

52

- When overriding a method, the method must have the same name and the same number and types of parameters in the same order
- If not, the method will overload
- This error is common; the annotation `@Override` preceding an overridden method will signal the compiler to issue an error if it does not find a corresponding method to override

```
@Override  
public String toString() {  
    . . .  
}
```

- It is good programming practice to use the `@Override` annotation in your code

# Polymorphism

53

- Polymorphism means *having many shapes*
- Polymorphism is a central feature of OOP
- It enables the JVM to determine at run time which of the classes in a hierarchy is referenced by a superclass variable or parameter

# Polymorphism (cont.)

54

- For example, if you write a program to reference computers, you may want a variable to reference a Computer or a Notebook
- If you declare the reference variable as  
Computer theComputer; it can reference either a Computer or a Notebook—because a Notebook *is-a* Computer

# Polymorphism (cont.)

55

- Suppose the following statements are executed:

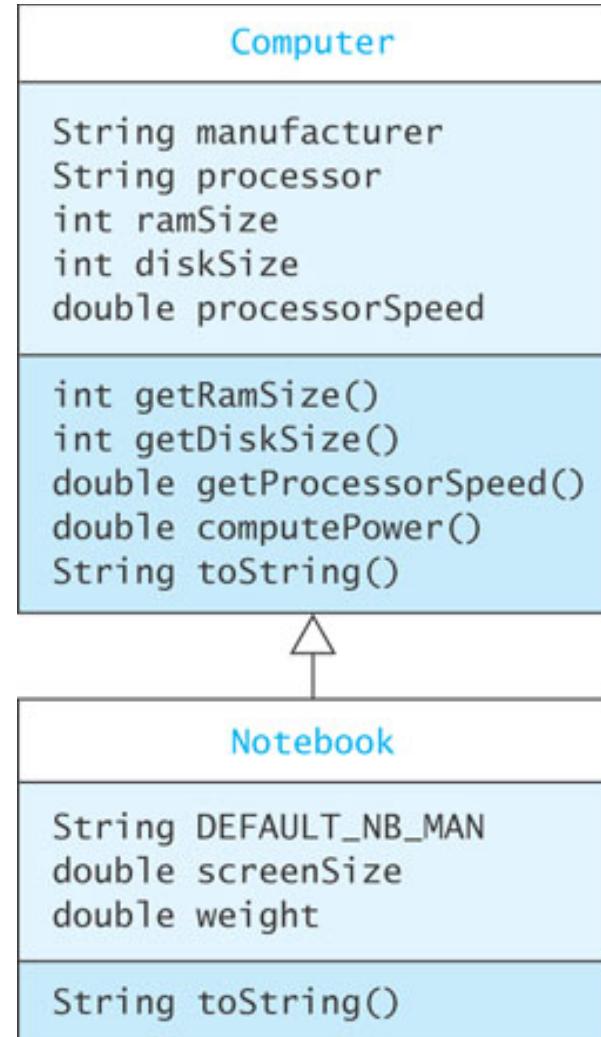
```
theComputer = new Notebook("Bravo", "Intel",
    4, 240, 2.4, 15, 7.5);
System.out.println(theComputer.toString());
```

- The variable **theComputer** is of type Computer,
- Which **toString()** method will be called, Computer's or Notebook's?

# Polymorphism (cont.)

56

- The JVM correctly identifies the **type of theComputer as Notebook and calls the `toString()` method associated with Notebook**
- This is an example of **polymorphism**
- The type cannot be determined at *compile time*, but it can be determined at *run time*



# Polymorphism (cont.)

57

- Computer [] labComputers = new Computer[10];
- labComputers[i] **can reference either a Computer or a Notebook because Notebook is a subclass of Computer**
- **For** labComputers[i].toString() **polymorphism ensures that the correct toString method will be executed**

# Methods with Class Parameters

58

- Polymorphism simplifies programming when writing methods with class parameters
- If we want to compare the power of two computers (either Computers or Notebooks) we do not need to overload methods with parameters for two Computers, or two Notebooks, or a Computer and a Notebook
- We simply write one method with two parameters of type Computer and allow the JVM, using polymorphism, to call the correct method

# Methods with Class Parameters (cont.)

59

```
/** Compares power of this computer and its argument computer
@param aComputer The computer being compared to this computer
@return -1 if this computer has less power,
        0 if the same, and
        +1 if this computer has more power.

*/
public int comparePower(Computer aComputer) {
    if (this.computePower() < aComputer.computePower())
        return -1;
    else if (this.computePower() == aComputer.computePower())
        return 0;
    else return 1; }
```

# Abstract Classes

## Section 1.4

# Abstract Classes

61

- An abstract class is denoted by using the word **abstract** in its heading:

*visibility abstract class className*

- An abstract class differs from an actual class (sometimes called a concrete class) in two respects:
  - An abstract class cannot be instantiated
  - An abstract class may declare abstract methods
- Just as in an interface, an abstract method is declared through a method heading:

*visibility abstract resultType methodName (parameterList);*

- A concrete class that is a subclass of an abstract class must provide an implementation for each abstract method

# Abstract Classes (cont.)

62

- Use an abstract class in a class hierarchy when you need a **base class** for two or more subclasses that share some attributes
- You can declare some or all of the attributes and define some or all of the methods that are common to these subclasses
- You can also require that the actual subclasses implement certain methods by declaring these methods **abstract**

# Example of an Abstract Class

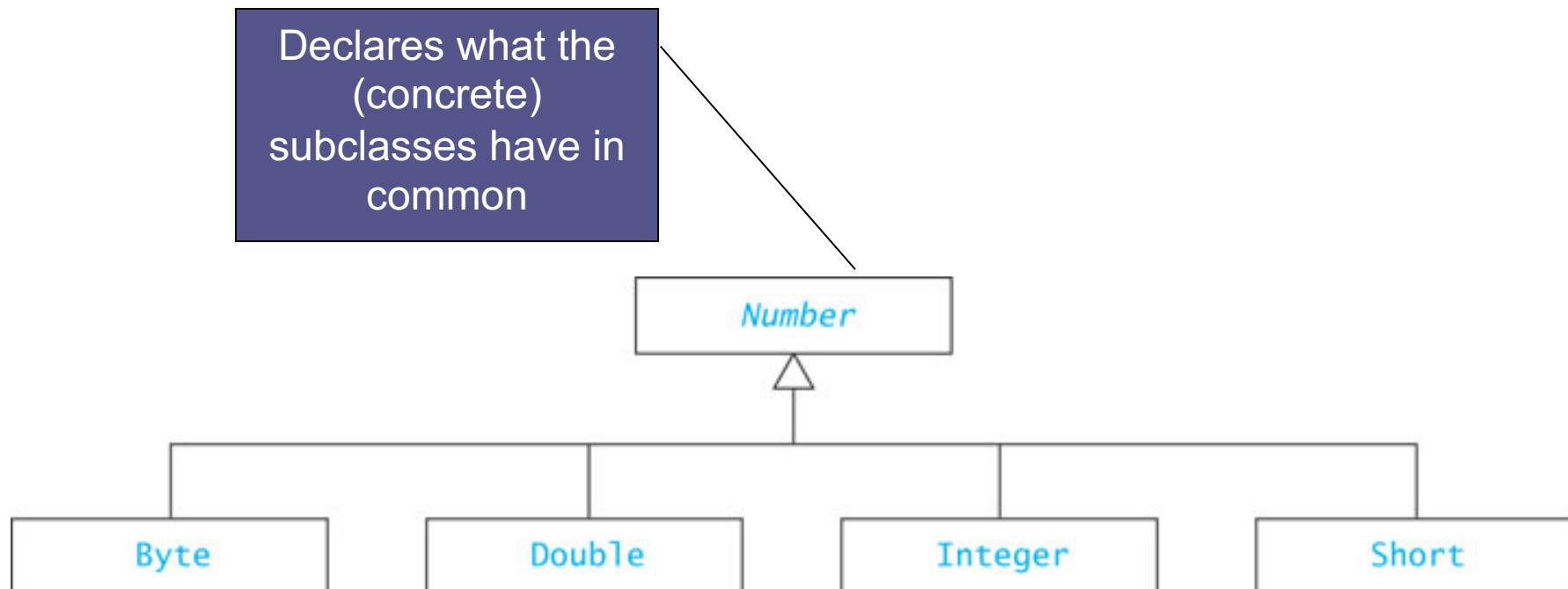
63

```
public abstract class Food {  
    public final String name;  
    private double calories;  
    // Actual methods  
    public double getCalories () {  
        return calories;  
    }  
    protected Food (String name, double calories) {  
        this.name      = name;  
        this.calories = calories;  
    }  
    // Abstract methods  
    public abstract double percentProtein();  
    public abstract double percentFat();  
    public abstract double percentCarbs();  
}
```

# Java Wrapper Classes

64

- A *wrapper class* is used to store a primitive-type value in an object type



# Interfaces, Abstract Classes, and Concrete Classes

65

- A Java *interface* can declare methods, but cannot implement them
- Methods of an interface are called abstract methods.
- An *abstract class* can have:
  - abstract methods (no body)
  - concrete methods (with a body)
  - data fields
- Unlike a concrete class, an *abstract class*
  - cannot be instantiated
  - can declare abstract methods which *must* be implemented in all concrete subclasses

# Abstract Classes and Interfaces

66

- Abstract classes and interfaces cannot be instantiated
- An abstract class *can have constructors!*
  - Purpose: initialize data fields when a subclass object is created
  - The subclass uses **super (...)** to call the constructor
- An abstract class may *implement* an interface, but need not define all methods of the interface
  - Implementation is left to subclasses

# Inheriting from Interfaces vs. Classes

67

- A class can *extend* 0 or 1 superclass
- An interface cannot extend a class
- A class or interface can *implement* 0 or more interfaces

# Summary of Features of Actual Classes, Abstract Classes, and Interfaces

68

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created.	Yes	No	No
This can define instance variables and methods.	Yes	Yes	No
This can define constants.	Yes	Yes	Yes
The number of these a class can extend.	0 or 1	0 or 1	0
The number of these a class can implement.	0	0	Any number
This can extend another class.	Yes	Yes	No
This can declare abstract methods.	No	Yes	Yes
Variables of this type can be declared.	Yes	Yes	Yes

# Class Object and Casting

## Section 1.5

# Class Object

70

- Object is the root of the class hierarchy
- Every class has Object as a superclass
- All classes inherit the methods of Object but may override them

Method	Behavior
boolean equals(Object obj)	Compares this object to its argument.
int hashCode()	Returns an integer hash code value for this object.
String toString()	Returns a string that textually represents the object.
Class<?> getClass()	Returns a unique object that identifies the class of this object.

# Method `toString`

71

- You should always override `toString` method if you want to print object state
- If you do not override it:
  - ▣ `Object.toString` will return a String
  - ▣ Just not the String you want!

**Example:** `ArrayBasedPD@ef08879`

The name of the class, @, instance's hash code

# Operations Determined by Type of Reference Variable

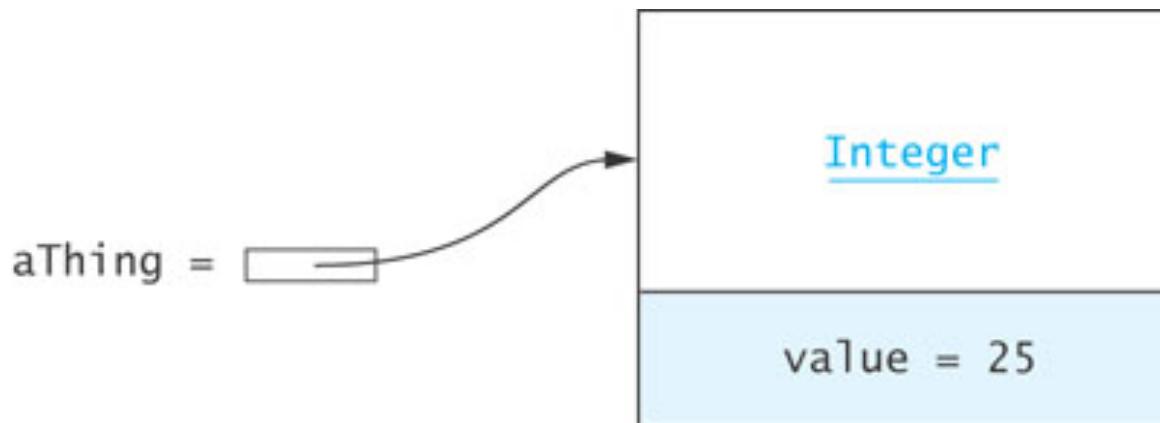
72

- As shown previously with `Computer` and `Notebook`, a variable can refer to object whose type is a *subclass* of the variable's declared type

- Java is *strongly typed*

```
Object aThing = Integer.valueOf(25);
```

- The compiler always verifies that a variable's type includes the class of every expression assigned to the variable (e.g., *class Object must include class Integer*)



# Operations Determined by Type of Reference Variable (cont.)

73

- The type of the variable determines what operations are legal

```
Object athing = Integer.valueOf(25);
```

- The following is legal:

```
athing.toString();
```

- But this is not legal:

```
athing.intValue();
```

- Object has a `toString()` method, but it does not have an `intValue()` method (even though `Integer` does, the reference is considered of type `Object`)

# Operations Determined by Type of Reference Variable (cont.)

74

- The following method will compile,

```
aThing.equals(Integer.valueOf("25")) ;
```

- Object has an `equals` method, and so does Integer
- Which one is called? Why?
- Why does the following generate a syntax error?

```
Integer aNum = aThing;
```

- Incompatible types!

# Casting in a Class Hierarchy

75

- Casting obtains a reference of a different, but *matching*, type
- Casting does *not change* the object!
  - It creates an anonymous reference to the object

```
Integer aNum = (Integer) aThing;
```

- Does this work?

```
((Integer) aThing).intValue()
```

# Casting in a Class Hierarchy (cont.)

76

- *Downcast:*
  - Cast *superclass type* to *subclass type*
  - Java checks at *run time* to make sure it's legal
  - If it's not legal, it throws **ClassCastException**
- *Upcast:*
  - Always valid but unnecessary

# Using instanceof to Guard a Casting Operation

77

- instanceof can guard against a ClassCastException

```
Object obj = ...;
if (obj instanceof Integer) {
    Integer i = (Integer) obj;
    int val = i;
    ...
} else {
    ...
}
```

# Method Object.equals

78

- `Object.equals` method has a parameter of type `Object`

```
public boolean equals (Object other) {  
    ... }
```

- Compares two objects to determine if they are equal
- A class must override `equals` in order to support comparison

# Employee.equals()

79

```
/** Determines whether the current object matches its argument.  
 * @param obj The object to be compared to the current object  
 * @return true if the objects have the same name and address;  
 *         otherwise, return false  
 */  
@Override  
public boolean equals(Object obj) {  
    if (obj == this) return true;  
    if (obj == null) return false;  
    if (this.getClass() == obj.getClass()) {  
        Employee other = (Employee) obj;  
        return name.equals(other.name) &&  
               address.equals(other.address);  
    } else {  
        return false;  
    }  
}
```

# Class Class

80

- Every class has a Class object that is created automatically when the class is loaded into an application
- Each Class object is unique for the class
- Method getClass() is a member of Object that returns a reference to this unique object
- In the previous example, if this.getClass() == obj.getClass() is true, then we know that obj and this are both of class Employee



# CS 570: Data Structures

## Computational Complexity

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# Week 3

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 1.6, 1.7, 2.1 & Chapter 3

# A Java Inheritance Example—The Exception Class Hierarchy

## Section 1.6

# Run-time Errors or Exceptions

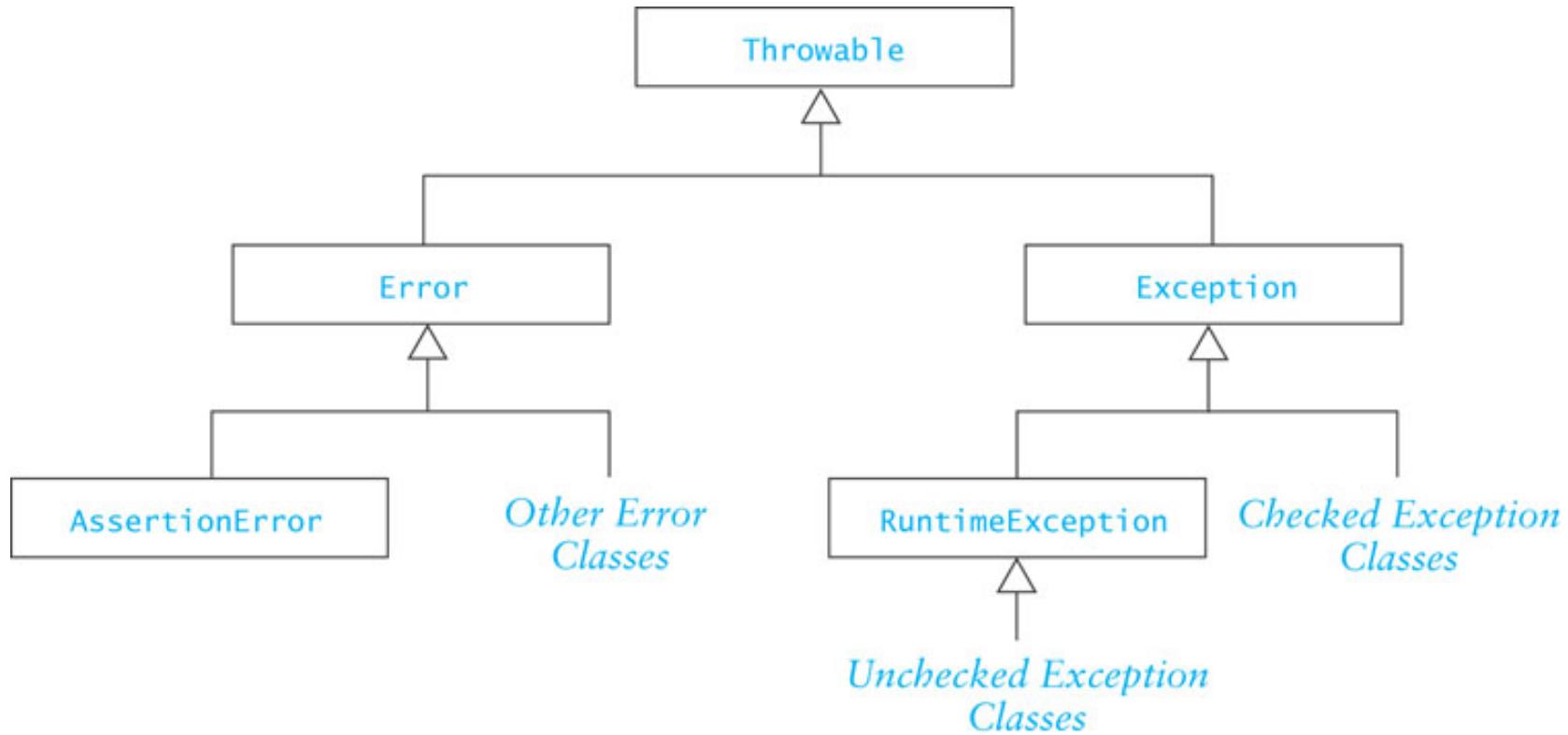
4

- Run-time errors
  - occur during program execution (i.e. at run-time)
  - occur when the JVM detects an operation that it knows to be incorrect
  - cause the JVM to throw an exception
- Examples of run-time errors include
  - division by zero
  - array index out of bounds
  - number format error
  - null pointer exception

# Class Throwable

5

- ❑ Throwable is the superclass of all exceptions
- ❑ All exception classes inherit its methods



# Checked and Unchecked Exceptions

6

- **Checked exceptions**
  - normally not due to programmer error
  - generally beyond the control of the programmer
  - all input/output errors are checked exceptions
  - **Examples:** IOException, FileNotFoundException
- **Unchecked exceptions result from**
  - programmer error (try to prevent them with defensive programming)
  - a serious external condition that is unrecoverable
  - **Examples:** NullPointerException, ArrayIndexOutOfBoundsException

# Checked and Unchecked Exceptions (cont.)

7

- The class `Error` and its subclasses represent errors due to serious external conditions; they are unchecked
  - ▣ Example: `OutOfMemoryError`
  - ▣ You cannot foresee or guard against them
  - ▣ While you can attempt to handle them, it is generally not a good idea as you will probably be unsuccessful
- The class `Exception` and its subclasses can be handled by a program
  - ▣ `RuntimeException` and its subclasses are unchecked
  - ▣ All others must be either:
    - explicitly caught or
    - explicitly mentioned as *thrown* by the method

# Some Common Unchecked Exceptions

8

- `ArithmaticException`: **division by zero, etc.**
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`: **converting a “bad” string to a number**
- `NullPointerException`

# Handling Exceptions

9

- When an exception is thrown, the normal sequence of execution is interrupted
- Default behavior (no handler)
  - ▣ Program stops
  - ▣ JVM displays an error message
- The programmer may provide a *handle*
  - ▣ Enclose statements in a `try` block
  - ▣ Process the exception in a `catch` block

# The try-catch Sequence

10

- The **try-catch** sequence resembles an **if-then-else** statement

```
try {  
    // Execute the following statements until an  
    // exception is thrown  
  
    ...  
    // Skip the catch blocks if no exceptions were thrown  
  
} catch (ExceptionTypeA ex) {  
    // Execute this catch block if an exception of type  
    // ExceptionTypeA was thrown in the try block  
    ...  
  
} catch (ExceptionTypeB ex) {  
    // Execute this catch block if an exception of type  
    // ExceptionTypeB was thrown in the try block  
    ...  
}
```

# The try-catch Sequence

11

- The **try-catch** sequence resembles an **if**

```
try {  
    // Execute the following statements until  
    // exception is thrown  
  
    ...  
    // Skip the catch blocks if no exception  
  
} catch (ExceptionTypeA ex) {  
    // Execute this catch block if an exception  
    // ExceptionTypeA was thrown in the try  
    ...  
  
} catch (ExceptionTypeB ex) {  
    // Execute this catch block if an exception  
    // ExceptionTypeB was thrown in the try block  
    ...  
}
```

## PITFALL!

### Unreachable catch block

**ExceptionTypeB** cannot be a subclass of **ExceptionTypeA**. If it is, its exceptions would be caught by the first **catch** clause and its **catch** clause would be unreachable.

# Using try-catch

12

- User input is a common source of exceptions

```
public static int getIntValue(Scanner scan) {  
    int nextInt = 0;           // next int value  
    boolean validInt = false; // flag for valid input  
    while(!validInt) {  
        try {  
            System.out.println("Enter number of kids: ");  
            nextInt = scan.nextInt();  
            validInt = true;  
        } catch (InputMismatchException ex) {  
            scan.nextLine(); // clear buffer  
            System.out.println("Bad data-enter an integer");  
        }  
    }  
    return nextInt;  
}
```

# Throwing an Exception When Recovery is Not Obvious

13

- In some cases, you may be able to write code that detects certain types of errors, but there may not be an obvious way to recover from them
- In these cases an exception can be *thrown*
- The calling method receives the thrown exception and must handle it

# Throwing an Exception When Recovery is Not Obvious (cont.)

14

```
public static void processPositiveInteger(int n) {  
    if (n < 0) {  
        throw new IllegalArgumentException("Invalid negative  
            argument");  
    } else {  
        // Process n as required  
        ...  
    }  
}
```

# Throwing an Exception When Recovery is Not Obvious (cont.)

15

```
public static void main(String[] args) {  
    Scanner scan = new Scanner(System.in);  
    try {  
        int num = getIntValue(scan);  
        processPositiveInteger(num);  
    } catch (IllegalArgumentException ex) {  
        System.err.println(ex.getMessage());  
        System.exit(1); // error indication  
    }  
    System.exit(0); // normal exit  
}
```

# Packages and Visibility

## Section 1.7

# Packages

17

- A Java package is a group of *cooperating classes*
- The Java API is organized as packages
- Indicate the package of a class at the top of the file:  
`package classPackage;`
- Classes in the *same package* should be in the *same directory* (folder)
- The folder must have the *same name* as the package
- Classes in the *same folder* must be  
in the *same package*

# Packages and Visibility

18

- Classes *not* part of a package can only access public members of classes in the package
- If a class is not part of the package, it must access the public classes by their complete name, which would be packagename.className
- For example,  
`x = Java.awt.Color.GREEN;`
- If the package is imported, the packageName prefix is not required.

```
import java.awt.Color;  
...  
x = Color.GREEN;
```

# The Default Package

19

- Files which do not specify a package are part of the default package
- If you do not declare packages, all of your classes belong to the default package
- The default package is intended for use during the early stages of implementation or for small prototypes
- When you develop an application, declare its classes to be in the same package

# Visibility

20

- We have seen three visibility layers, public, protected, private
- A fourth layer, package visibility, lies between private and protected
- Classes, data fields, and methods with package visibility are accessible to all other methods of the same package, but are not accessible to methods outside the package
- Classes, data fields, and methods that are declared protected are visible within subclasses that are declared outside the package (in addition to being visible to all members *inside* the package)
- There is no keyword to indicate package visibility
- Package visibility is the default in a package if public, protected, private are not used

# Visibility Supports Encapsulation

21

- **Visibility rules enforce encapsulation in Java**
- **private:** for members that should be invisible even in subclasses
- **package:** shields classes and members from classes outside the package
- **protected:** provides visibility to extenders or classes in the package
- **public:** provides visibility to all

# Visibility Supports Encapsulation (cont.)

22

Visibility	Applied to Classes	Applied to Class Members
<b>private</b>	Applicable to inner classes. Accessible only to members of the class in which it is declared.	Visible only within this class.
Default or package	Visible to classes in this package.	Visible to classes in this package.
<b>protected</b>	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared.	Visible to classes in this package and to classes outside the package that extend this class.
<b>public</b>	Visible to all classes.	Visible to all classes. The class defining the member must also be public.

# Visibility Supports Encapsulation (cont.)

23

- Encapsulation insulates against change
- Greater visibility means less encapsulation
  
- So... use the most restrictive visibility possible to get the job done!

# Algorithm Efficiency and Big-O

## Section 2.1

# Data structure

## Data structure

- a way to store and organize data in order to facilitate access and modification.
- no single data structure optimal for all purposes.
- usually optimized for a specific problem setting.
- important to know the strength and limitations of several of them.

## Examples:

- Trees (binary search trees, red-black trees, b-trees, ...).
- Stacks (last in, first out), queues (first in, first out), priority queues.

# Algorithms

Algorithms:

- well-defined computational procedure.
- takes value or set of values as input.
- produces value or set of values as output.
- tool for solving a well-specified computational problem.
- instance of a problem consists of the input needed to compute a solution to the problem.
- correct algorithm solves the given computational problem.

Sorting problem:

**Input:** A sequence of  $n$  numbers  $a_1, \dots, a_n$ .

**Output:** A permutation  $a_1, \dots, a_n$  of the input sequence such that  $a_1 \leq \dots \leq a_n$ .

# 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, ...).

Example:

- Insertion sort ( $c_1n^2$ ) vs. merge sort ( $c_2nlgn$ ).

# Asymptotic Performance

- In this course, 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/logic gates/etc.

# Asymptotic Notation

- By now you should have an intuitive feel for asymptotic (big-O) notation:
  - *What does  $O(n)$  running time mean?  $O(n^2)$ ?  $O(n \lg n)$ ?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our first task is to define this notation more formally and completely

# Input Size

- Time and space complexity
  - This is generally a function of the input size
    - E.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - Etc

# Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - $y = m * x + b$
    - $c = 5 / 9 * (t - 32)$
    - $z = f(x) + g(y)$
- We can be more exact if need be
- Worst case vs. average case

# Algorithm Efficiency and Big-O

32

- Getting a precise measure of the performance of an algorithm is difficult
- Big-O notation expresses the performance of an algorithm as a function of the number of items to be processed
- This permits algorithms to be compared for efficiency
- For more than a certain number of data items, some problems cannot be solved by any computer

# Linear Growth Rate

33

- If processing time increases in proportion to the number of inputs  $n$ , the algorithm grows at a linear rate

```
public static int search(int[] x, int target)
{
    for(int i=0; i<x.length; i++) {
        if (x[i]==target)
            return i;
    }
    return -1; // target not found
}
```

# Linear Growth

34

- If processing time increases linearly with the number of inputs  $n$ , the growth rate

- If the target is not present, the for loop will execute  $x.length$  times
- If the target is present the for loop will execute (on average)  $(x.length + 1)/2$  times
- Therefore, the total execution time is directly proportional to  $x.length$
- This is described as a growth rate of order  $n$  OR  $O(n)$

```
public static int search(int[] x, int target) {  
    for(int i=0; i<x.length; i++) {  
        if (x[i]==target)  
            return i;  
    }  
    return -1; // target not found  
}
```

# **n × m Growth Rate**

35

- Processing time can be dependent on two different inputs

```
public static boolean areDifferent(int[] x, int[] y) {  
    for(int i=0; i<x.length; i++) {  
        if (search(y, x[i]) != -1)  
            return false;  
    }  
    return true;  
}
```

# **n x m Growth Rate (cont.)**

36

## Processing time of inputs.

- The **for loop** will **execute** `x.length` **times**
- But it will call `search`, which will **execute** `y.length` **times**
- The **total execution time** is proportional to `(x.length * y.length)`
- The **growth rate has an order of n x m or O(n x m)**

```
public static boolean areDifferent(int[] x, int[] y) {  
    for(int i=0; i<x.length; i++) {  
        if (search(y, x[i]) != -1)  
            return false;  
    }  
    return true;  
}
```

# Quadratic Growth Rate

37

- If processing time is proportional to the square of the number of inputs  $n$ , the algorithm grows at a quadratic rate

```
public static boolean areUnique(int[] x) {  
    for(int i=0; i<x.length; i++) {  
        for(int j=0; j<x.length; j++) {  
            if (i != j && x[i] == x[j])  
                return false;  
        }  
    }  
    return true;  
}
```

# Quadratic Growth Rate (cont.)

38

- If processing time is proportional to the number of inputs  $n$ ,

- The for loop with  $i$  as index will execute  $x.length$  times
- The for loop with  $j$  as index will execute  $x.length$  times
- The total number of times the inner loop will execute is  $(x.length)^2$
- The growth rate has an order of  $n^2$  or  $O(n^2)$

```
public static boolean isIdentical(int[] x) {  
    for(int i=0; i<x.length; i++) {  
        for(int j=0; j<x.length; j++) {  
            if (i != j && x[i] == x[j])  
                return false;  
        }  
    }  
    return true;  
}
```

# Big-O Notation

39

- The  $O()$  in the previous examples can be thought of as an abbreviation of "order of magnitude"
- A simple way to determine the big-O notation of an algorithm is to look at the loops and to see whether the loops are nested
- Assuming a loop body consists only of simple statements,
  - a single loop is  $O(n)$
  - a pair of nested loops is  $O(n^2)$
  - a nested pair of loops inside another is  $O(n^3)$
  - and so on . . .

# Big-O Notation (cont.)

40

- You must also examine the *number of times* a loop is executed

```
for(i=1; i < x.length; i *= 2) {  
    // Do something with x[i]  
}
```

- The loop body will execute  $k-1$  times, with  $i$  having the following values:

$1, 2, 4, 8, 16, \dots, 2^k$   
until  $2^k$  is greater than  $x.length$

- Since  $2^{k-1} = x.length < 2^k$  and  $\log_2 2^k$  is  $k$ , we know that  $k-1 = \log_2(x.length) < k$
- Thus we say the loop is  $O(\log n)$  (in analyzing algorithms, we use logarithms to the base 2)
- Logarithmic functions grow slowly as the number of data items  $n$  increases

# Formal Definition of Big-O

41

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        Simple Statement  
    }  
}  
for (int i = 0; i < n; i++) {  
    Simple Statement 1  
    Simple Statement 2  
    Simple Statement 3  
    Simple Statement 4  
    Simple Statement 5  
}  
Simple Statement 6  
Simple Statement 7  
...  
Simple Statement 30
```

# Formal Definition of Big-O (cont.)

42

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        Simple Statement  
    }  
}  
for (int i = 0; i < n; i++) {  
    Simple Statement 1  
    Simple Statement 2  
    Simple Statement 3  
    Simple Statement 4  
    Simple Statement 5  
}  
Simple Statement 6  
Simple Statement 7  
...  
Simple Statement 30
```

This nested loop executes  
a *Simple Statement*  $n^2$   
times

# Formal Definition of Big-O (cont.)

43

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        Simple Statement  
    }  
}  
for (int i = 0; i < n; i++) {  
    Simple Statement 1  
    Simple Statement 2  
    Simple Statement 3  
    Simple Statement 4  
    Simple Statement 5  
}  
Simple Statement 6  
Simple Statement 7  
...  
Simple Statement 30
```

This loop executes 5  
*Simple Statements*  $n$  times  
( $5n$ )

# Formal Definition of Big-O (cont.)

44

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        Simple Statement  
    }  
}  
for (int i = 0; i < n; i++) {  
    Simple Statement 1  
    Simple Statement 2  
    Simple Statement 3  
    Simple Statement 4  
    Simple Statement 5  
}  
Simple Statement 6  
Simple Statement 7  
...  
Simple Statement 30
```

Finally, 25 Simple  
Statements are executed

# Formal Definition of Big-O (cont.)

45

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        Simple Statement  
    }  
}  
for (int i = 0; i < n; i++) {  
    Simple Statement 1  
    Simple Statement 2  
    Simple Statement 3  
    Simple Statement 4  
    Simple Statement 5  
}  
Simple Statement 6  
Simple Statement 7  
...  
Simple Statement 30
```

We can conclude that the relationship between processing time and  $n$  (the number of date items processed) is:

$$T(n) = n^2 + 5n + 25$$

# Formal Definition of Big-O (cont.)

46

- In terms of  $T(n)$ ,

$$T(n) = O(f(n))$$

- means that there exist
  - two constants,  $n_0$  and  $c$ , greater than zero, and
  - a function,  $f(n)$ ,
- such that for all  $n > n_0$ ,  $cf(n) \geq T(n)$
- In other words, as  $n$  gets sufficiently large (larger than  $n_0$ ), there is some constant  $c$  for which the processing time will always be less than or equal to  $cf(n)$
- $cf(n)$  is an upper bound on performance

# Formal Definition of Big-O (cont.)

47

- The growth rate of  $f(n)$  will be determined by the fastest growing term, which is the one with the largest exponent
- In the example, an algorithm of

$$O(n^2 + 5n + 25)$$

is more simply expressed as

$$O(n^2)$$

- In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude

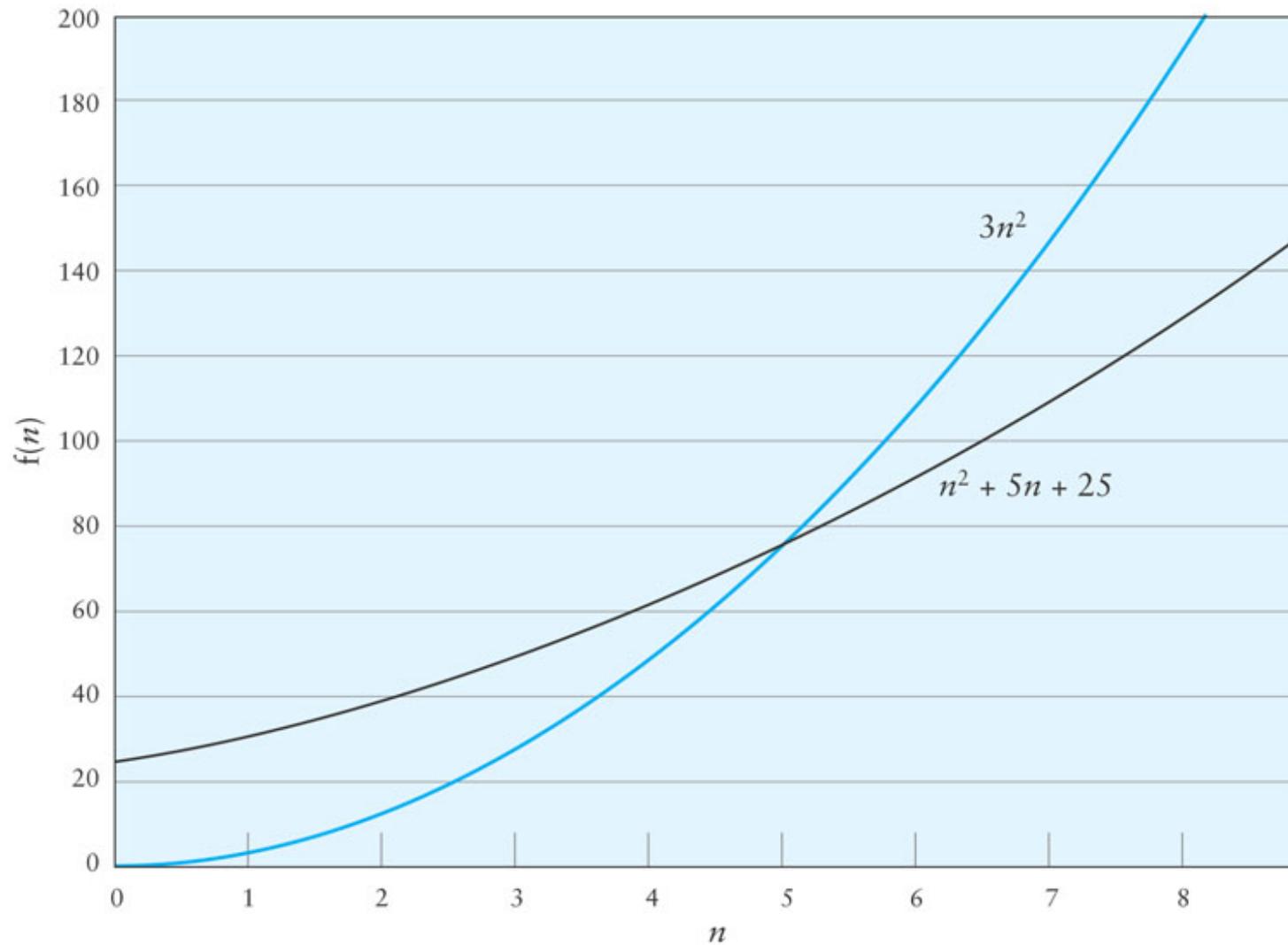
# Big-O Example 1

48

- Given  $T(n) = n^2 + 5n + 25$ , show that this is  $\mathcal{O}(n^2)$
- Find constants  $n_0$  and  $c$  so that, for all  $n > n_0$ ,  $cn^2 > n^2 + 5n + 25$ 
  - ▣ Find the point where  $cn^2 = n^2 + 5n + 25$
  - ▣ Let  $n = n_0$ , and solve for  $c$ 
$$c = 1 + 5/n_0 + 25/n_0^2$$
- When  $n_0$  is 5, the RHS is  $(1 + 5/5 + 25/25)$ ,  $c$  is 3
- So,  $3n^2 > n^2 + 5n + 25$  for all  $n > 5$
- Other values of  $n_0$  and  $c$  also work

# Big-O Example 1 (cont.)

49



# Big-O Example 2

50

- Consider the following loop

```
for (int i = 0; i < n; i++) {  
    for (int j = i + 1; j < n; j++) {  
        3 simple statements  
    }  
}
```

- $T(n) = 3(n - 1) + 3(n - 2) + \dots + 3$
- Factoring out the 3,  
 $3(n - 1 + n - 2 + \dots + 1)$
- $1 + 2 + \dots + n - 1 = (n \times (n-1))/2$

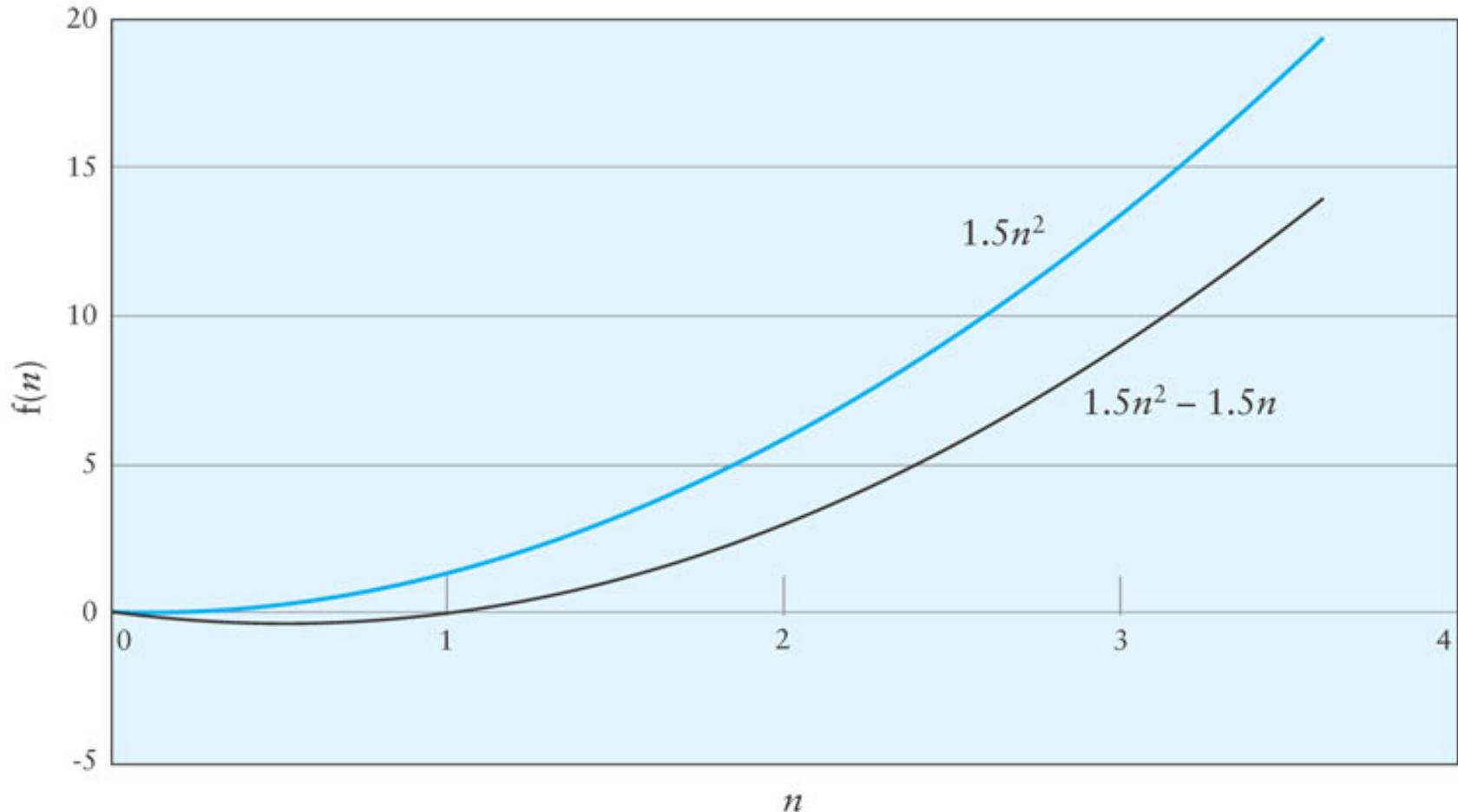
# Big-O Example 2 (cont.)

51

- Therefore  $T(n) = 1.5n^2 - 1.5n$
- When  $n = 1$ , the polynomial has value 0
- For values of  $n > 1$ ,  $1.5n^2 > 1.5n^2 - 1.5n$
- Therefore  $T(n)$  is  $O(n^2)$  when  $n_0$  is 1 and c is 1.5

# Big-O Example 2 (cont.)

52



# Big-O of polynomials

- Suppose runtime is  $an^2 + bn + c$ 
  - If any of  $a$ ,  $b$ , and  $c$  are less than 0 replace the constant with its absolute value
- $$\begin{aligned} an^2 + bn + c &\leq (a + b + c)n^2 + (a + b + c)n + (a + b + c) \\ &\leq 3(a + b + c)n^2 \text{ for } n \geq 1 \end{aligned}$$
- Let  $c' = 3(a + b + c)$  and let  $n_0 = 1$

# Big-O of polynomials

- A **polynomial of degree k** is  $O(n^k)$
- Proof:
  - Suppose  $f(n) = b_k n^k + b_{k-1} n^{k-1} + \dots + b_1 n + b_0$ 
    - Let  $a_i = |b_i|$
  - $f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$

$$\leq n^k \sum a_i \frac{n^i}{n^k} \leq n^k \sum a_i \leq cn^k$$

# Symbols Used in Quantifying Performance

55

Symbol	Meaning
$T(n)$	The time that a method or program takes as a function of the number of inputs, $n$ . We may not be able to measure or determine this exactly.
$f(n)$	Any function of $n$ . Generally, $f(n)$ will represent a simpler function than $T(n)$ , for example, $n^2$ rather than $1.5n^2 - 1.5n$ .
$O(f(n))$	Order of magnitude. $O(f(n))$ is the set of functions that grow no faster than $f(n)$ . We say that $T(n) = O(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$ .

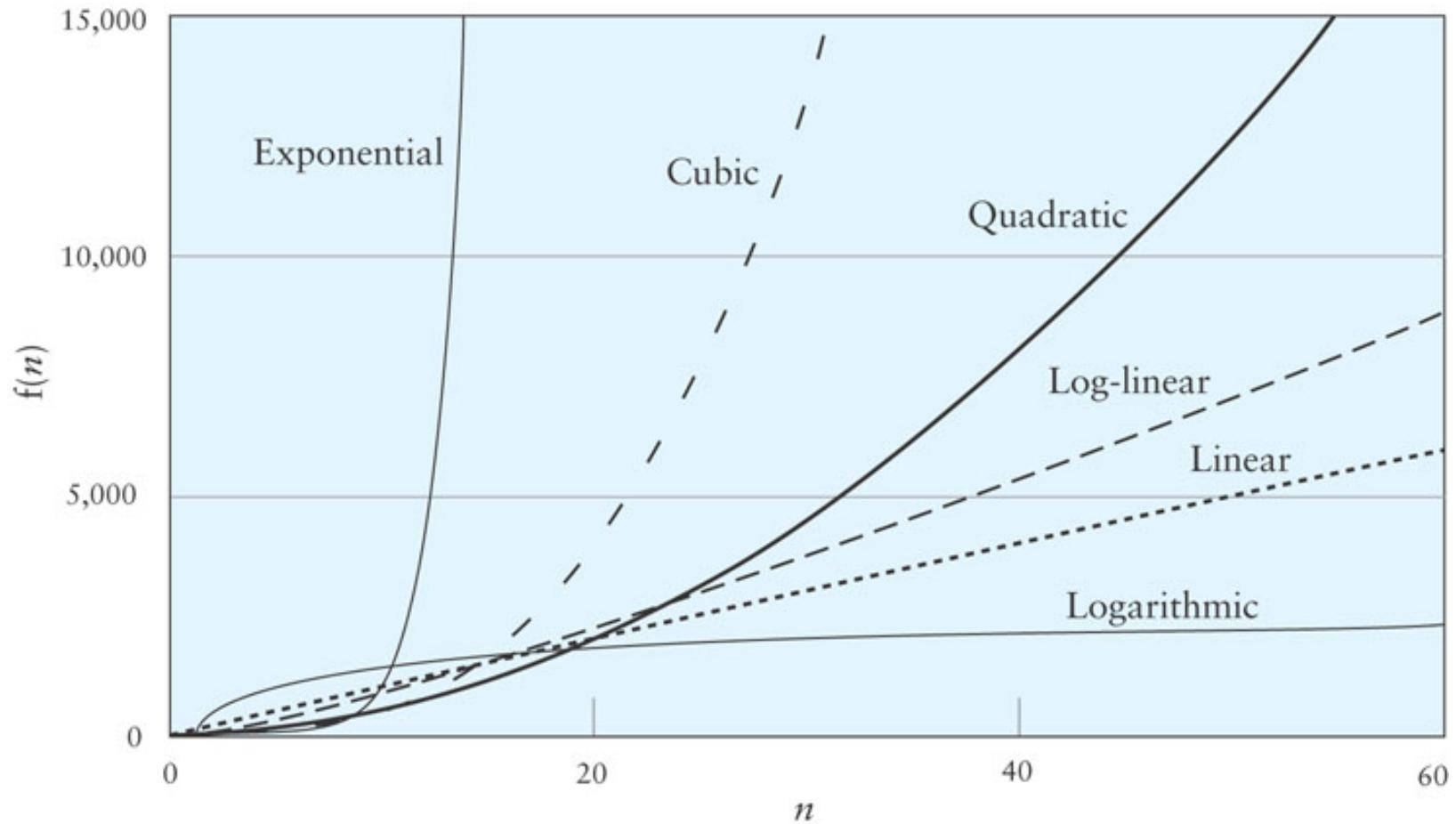
# Common Growth Rates

56

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

# Different Growth Rates

57



# Effects of Different Growth Rates

58

$O(f(n))$	$f(50)$	$f(100)$	$f(100)/f(50)$
$O(1)$	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
$O(2^n)$	$1.126 \times 10^{15}$	$1.27 \times 10^{30}$	$1.126 \times 10^{15}$
$O(n!)$	$3.0 \times 10^{64}$	$9.3 \times 10^{157}$	$3.1 \times 10^{93}$

# Algorithms with Exponential and Factorial Growth Rates

59

- Algorithms with exponential and factorial growth rates have an effective practical limit on the size of the problem they can be used to solve
- With an  $O(2^n)$  algorithm, if 100 inputs takes an hour then,
  - ▣ 101 inputs will take 2 hours
  - ▣ 105 inputs will take 32 hours
  - ▣ 114 inputs will take 16,384 hours (almost 2 years!)

# Algorithms with Exponential and Factorial Growth Rates (cont.)

60

- Encryption algorithms take advantage of this characteristic
- Some cryptographic algorithms can be broken in  $O(2^n)$  time, where  $n$  is the number of bits in the key
- A key length of 40 is considered breakable by a modern computer,
- but a key length of 100 bits will take a billion-billion ( $10^{18}$ ) times longer than a key length of 40

# Testing

## Section 2.11

# Testing

62

- Testing runs a program or part of a program under controlled conditions to verify that results are as expected
- Testing detects program defects after the program compiles (all syntax error have been removed)
- While extremely useful, testing cannot detect the absence of all defects in complex programs

# Why is Testing Important?

China Airlines Flight 140, April 26th, 1994



# Why is Testing Important?



# Why is Testing Important?

April, 1999

Failed Satellite Launch



\$ 1.2 billion lost

# Why is Testing Important?

May, 1996

## U.S. Bank Accounts



823 Customers paid \$920 million

Paul Ehrlich:  
*“To err is human, but to  
really foul things up  
you need a computer.”*

# Testing Levels

67

- **Unit testing:** tests the smallest testable piece of the software, often a class or a sufficiently complex method
- **Integration testing:** tests integration among units
- **System testing:** tests the whole program in the context in which it will be used
- **Acceptance testing:** system testing designed to show that a program meets its functional requirements

# Types of Testing

68

- Black-box testing:
  - tests the item (method, class, or program) based on its interfaces and functional requirements
  - is also called *closed-box* or *functional* testing
  - is accomplished by varying input parameters across the allowed range and outside the allowed range, and comparing with independently calculated results

# Types of Testing (cont.)

69

- White-box testing:
  - ▣ tests the item (method, class, or program) with knowledge of its internal structure
  - ▣ is also called *glass-box*, *open-box*, or *coverage* testing
  - ▣ exercises as many paths through the element as possible
  - ▣ provides appropriate coverage
    - statement – ensures each statement is executed at least once
    - branch – ensures each choice of branch (if , switch, loops) is taken
    - path – tests each path through a method

# Preparations for Testing

70

- A test plan should be developed early in the design stage—the earlier an error is detected, the easier and less expensive it is to correct it
- Aspects of test plans include deciding:
  - how the software will be tested
  - when the tests will occur
  - who will do the testing
  - what test data will be used

# Testing Tips

71

1. Carefully document method operation, parameter, and class attributes using comments; follow Javadoc conventions
2. Leave a trace of execution by displaying the method name as you enter it
3. Display values of all input parameters upon entering a method and values of any class attributes accessed by the method
4. Display values of all method outputs after returning from a method, together with any class attributes that are modified by a method

# Testing Tips (cont.)

72

- An efficient way to display values of parameters, return values, and class attributes:

```
private static final boolean TESTING = true; // or false to
                                            // disable

if (TESTING) {
    // Code for output statements
}
```

- Remove these features when you are satisfied with the testing results
- You can define different boolean flags for different tests

# Developing the Test Data

73

- In black-box testing, test data should check for all expected inputs as well as unanticipated data
- In white-box testing, test data should be designed to ensure all combinations of paths through the code are executed

# Testing Boundary Conditions

74

## □ Example

```
for (int i = 0; i < x.length; i++) {  
    if (x[i] == target)  
        return i;  
}
```

- Test the boundary conditions (for white-box and black-box testing) when target is:
  - ▣ first element ( $x[0] == \text{target}$  is true)
  - ▣ last element ( $x[\text{length}-1] == \text{target}$  is true)
  - ▣ not in array ( $x[i] == \text{target}$  is always false)
  - ▣ present multiple times ( $x[i] == \text{target}$  for more than one value of  $i$ )

# Testing Boundary Conditions (cont.)

75

```
for (int i = 0; i < x.length; i++) {  
    if (x[i] == target)  
        return i;  
}
```

- Test for the typical situation when target is:
  - somewhere in the middle
- and for the boundary conditions when the array has
  - only one element
  - no elements

# Testing Boundary Conditions (cont.)

76

```
private static void verify(int[] x, int target, int
expected) {
    int actual = search(x, target);
    System.out.print("search(x, " + target + ") is "
                    + actual + ", expected " + expected);
    if (actual == expected)
        System.out.println(": Pass");
    else
        System.out.println(": ****Fail");
}
```

# Testing Boundary Conditions (cont.)

77

```
public static void main(String[] args) {  
    // Array to search.  
    int[] x = {5, 12, 15, 4, 8, 12, 7};  
    // Test for target as first element.  
    verify(x, 5, 0);  
    // Test for target as last element.  
    verify(x, 7, 6);  
    // Test for target not in array.  
    verify(x, -5, -1);  
    // Test for multiple occurrences of target.  
    verify(x, 12, 1);  
    ...
```

# Testing Boundary Conditions (cont.)

78

```
public static void main(String[] args) {  
    int[] x = {5, 12, 15, 4, 8, 12, 7}; // Array to search.  
  
    ...  
  
    // Test for target somewhere in middle.  
    verify(x, 4, 3);  
  
    // Test for 1-element array.  
    x = new int[1];  
    x[0] = 10;  
  
    verify(x, 10, 0);  
    verify(x, -10, -1);  
  
    // Test for an empty array.  
    x = new int[0];  
    verify(x, 10, -1);  
  
}
```

# Stubs

79

- *Stubs* are method placeholders for methods called by other classes, but not yet implemented
- Stubs allowing testing as classes are being developed
- A sample stub:

```
public void save() {  
    System.out.println("Stub for save has  
        been called");  
    modified = false;  
}
```

# Stubs (cont.)

80

- Stubs can
  - print out value of inputs
  - assign predictable values to outputs
  - change the state of variables

# Preconditions and Postconditions

81

- A *precondition* is a statement of any assumptions or constraints on the input parameters before a method begins execution
- A *postcondition* describes the result of executing the method, including any change to the object's state
- A method's preconditions and postconditions serve as a contract between a method caller and the method programmer

# Preconditions and Postconditions

82

```
/** Method Save  
    pre: the initial directory contents are read  
         from a data file  
    post: writes the directory contents back to  
          a data file  
 */  
public void save() {  
    . . .  
}
```

# Drivers

83

- Another testing tool
- A driver program
  - ▣ declares any necessary object instances and variables
  - ▣ assigns values to any of the method's inputs (specified by the preconditions)
  - ▣ calls the method
  - ▣ displays the outputs returned by the method
- Driver code can be added to a class's main method (each class can have a main method; only one main method - the one you designate to execute - will run)

# JUnit

84

- JUnit, a popular program for Java projects, helps you develop testing programs (see Appendix D)
- Many IDEs are shipped with debugger programs you can use for testing
- Use Javadoc!



# CS 570: Data Structures

## Collections Framework:

### Lists

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 2 (PART 1)

Lists and the  
Collections Framework

# Chapter Objectives

3

- The List interface
- Writing an array-based implementation of List
- Linked list data structures:
  - ▣ Singly-linked
  - ▣ Doubly-linked
  - ▣ Circular
- Implementing the List interface as a linked list
- The Iterator interface
  - ▣ Low priority for CS 570
- The Java Collections framework (hierarchy)
  - ▣ Low priority for CS 570

# Week 4

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 2.2 – 2.4

# Introduction

5

- A *list* is a collection of elements, each with a position or index
- *Iterators* facilitate sequential access to lists
- **Classes** `ArrayList`, `Vector`, and `LinkedList` **are subclasses of abstract class `AbstractList`** and *implement the List interface*

# The List Interface and ArrayList Class

## Section 2.2

# List Interface and ArrayList Class

7

- An array is an indexed structure
- In an indexed structure,
  - ▣ elements may be accessed in any order using subscript values
  - ▣ elements can be accessed in sequence using a loop that increments the subscript
- With the Java Array object, you cannot
  - ▣ increase or decrease its length (length is fixed)
  - ▣ add an element at a specified position without shifting elements to make room
  - ▣ remove an element at a specified position and keep the elements contiguous without shifting elements to fill in the gap

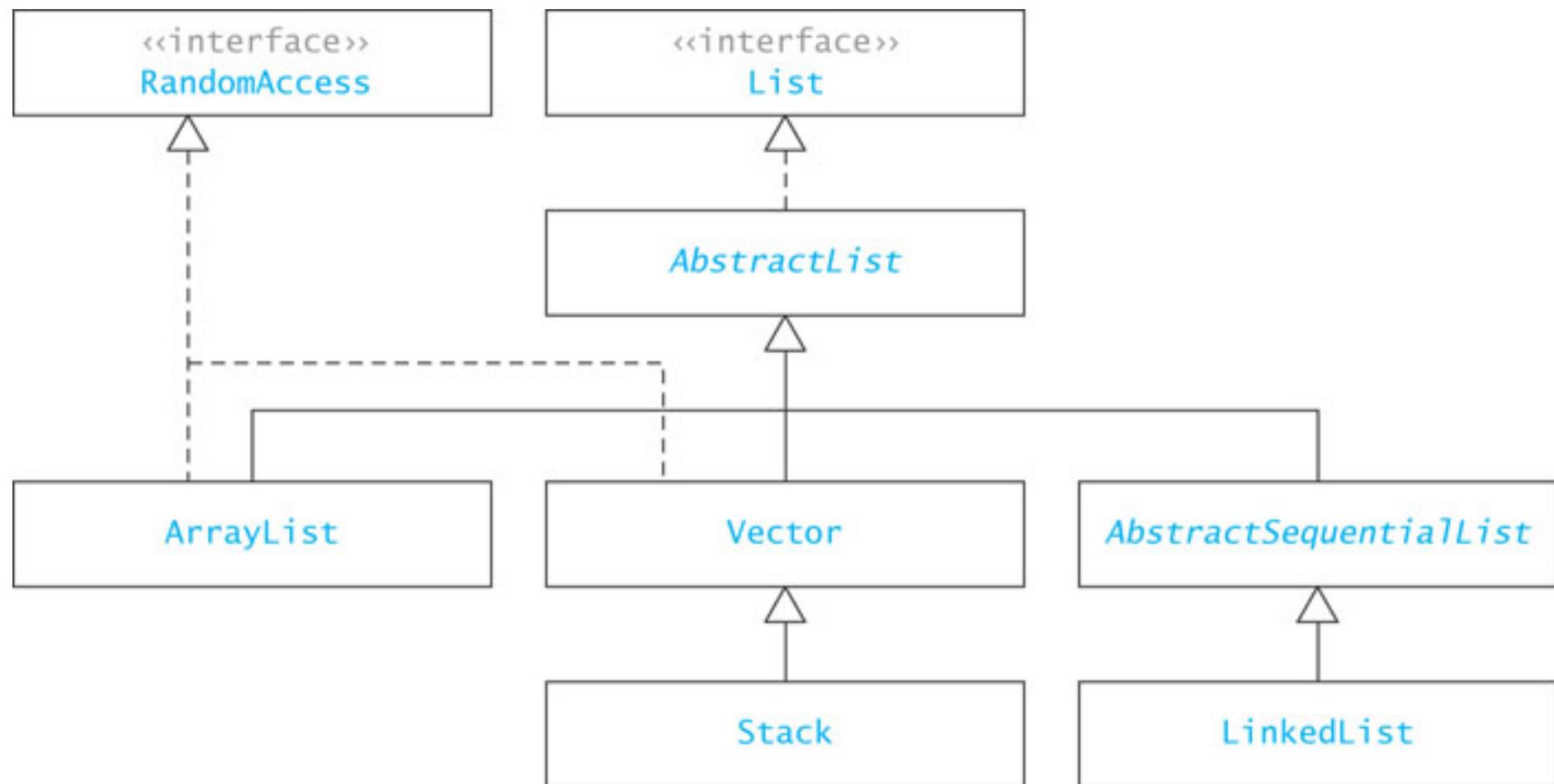
# List Interface and ArrayList Class (cont.)

8

- Java provides a List interface as part of its API `java.util`
- Classes that implement the List interface provide the functionality of an indexed data structure and offer many more operations
- A sample of the operations:
  - Obtain an element at a specified position
  - Replace an element at a specified position
  - Find a specified target value
  - Add an element at either end
  - Remove an element from either end
  - Insert or remove an element at any position
  - Traverse the list structure without managing a subscript
- All classes introduced in this chapter support these operations, but they do not support them with the same degree of efficiency

# java.util.List Interface and its Implementers

9



# ArrayList **Class**

10

- The simplest class that implements the List interface
- An improvement over an **array object**
- Use when:
  - you will be adding new elements to the end of a list
  - you need to access elements quickly in any order

# List Interface and ArrayList Class

11

- Unlike the Array data structure, classes that implement the List interface cannot store primitive types
- Classes must store values as objects
- This requires you to wrap primitive types, such as int and double in object wrappers, such as Integer and Double

# ArrayList Class (cont.)

12

- To declare a List “object” whose elements will reference String objects:

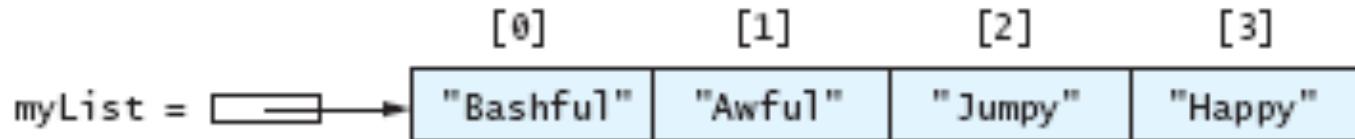
```
List<String> myList = new ArrayList<String>();
```

- The initial List is empty and has a default initial capacity of 10 elements
- To add strings to the list,

```
myList.add("Bashful");  
myList.add("Awful");  
myList.add("Jumpy");  
myList.add("Happy");
```

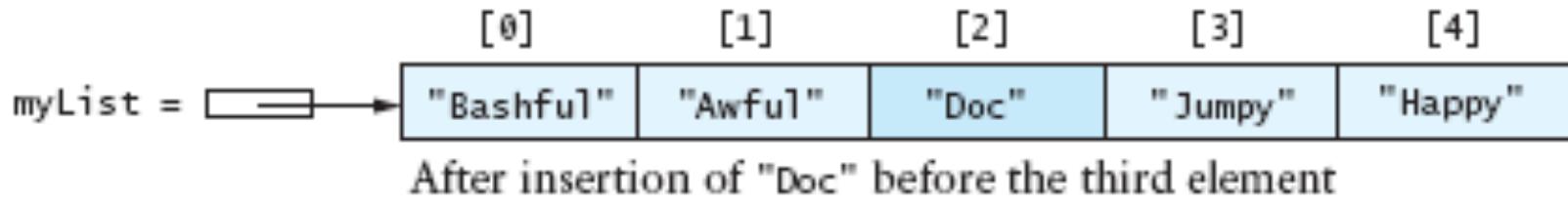
# ArrayList Class (cont.)

13



## □ Adding an element with subscript 2:

```
myList.add(2, "Doc");
```



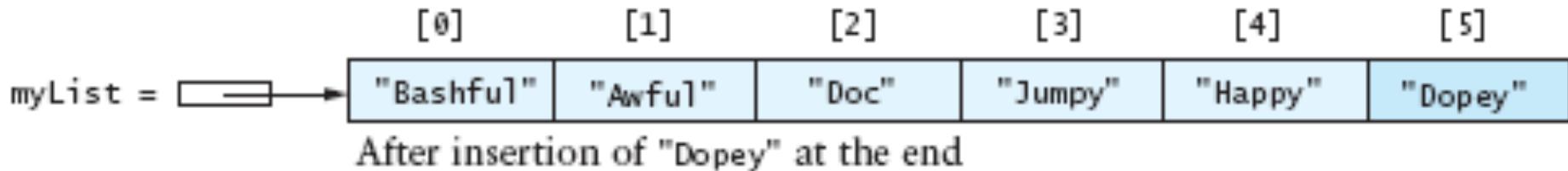
## □ Notice that the subscripts of "Jumpy" and "Happy" have changed from [2],[3] to [3],[4]

# ArrayList Class (cont.)

14

- When no subscript is specified, an element is added at the end of the list:

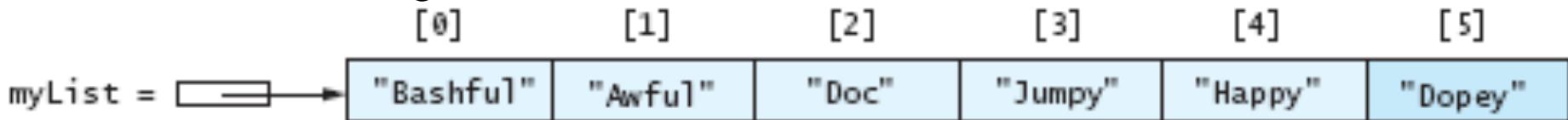
```
myList.add("Dopey");
```



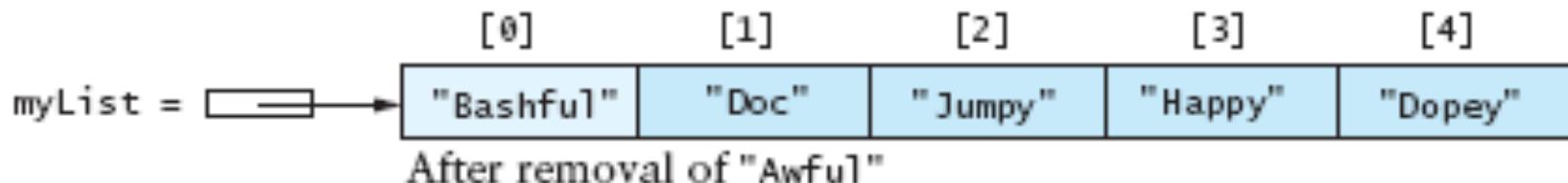
# ArrayList Class (cont.)

15

## □ Removing an element:



```
myList.remove(1);
```

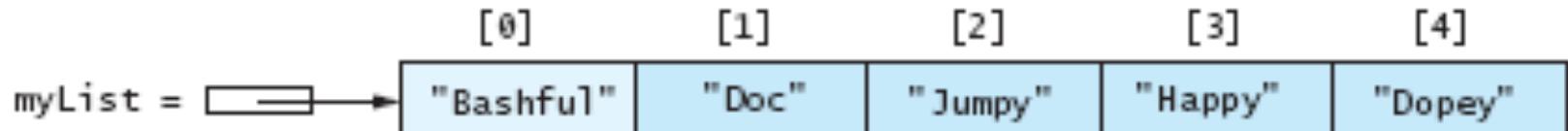


- The strings referenced by [2] to [5] have changed to [1] to [4]

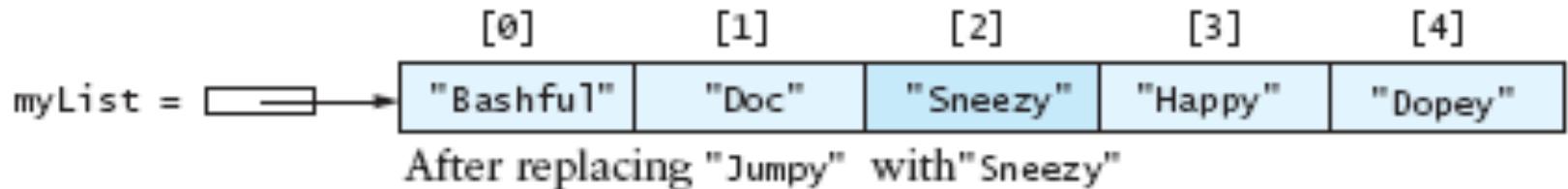
# ArrayList Class (cont.)

16

- You may also replace an element:

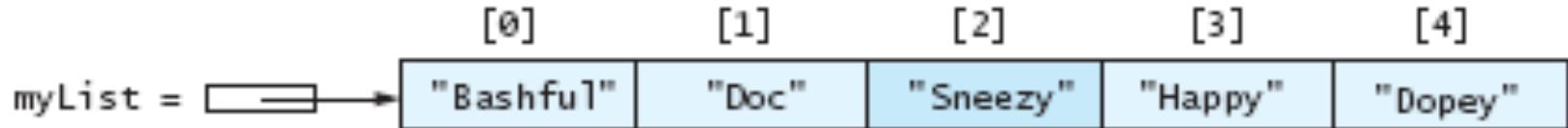


```
myList.set(2, "Sneezy");
```



# ArrayList Class (cont.)

17



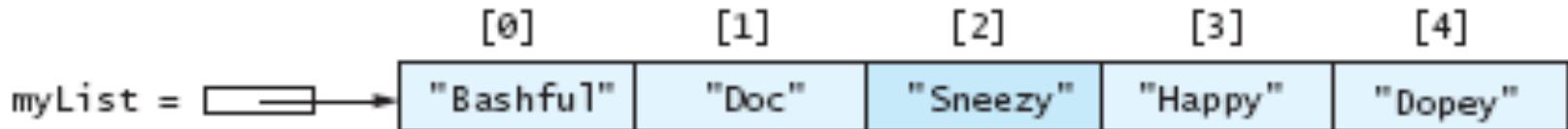
- You cannot access an element using a bracket index as you can with arrays (`array[1]`)
- Instead, you must use the `get()` method:

```
String dwarf = myList.get(2);
```

- The value of `dwarf` becomes "Sneezy"

# ArrayList Class (cont.)

18



- You can also search an ArrayList:

```
myList.indexOf("Sneezy");
```

- This returns 2 while

```
myList.indexOf("Jumpy");
```

- returns -1 which indicates an unsuccessful search

# Generic Collections

19

- The statement

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

uses a language feature called **generic collections** or **generics**

- The statement creates a List **of** String; only references of type String can be stored in the list
- String in this statement is called a **type parameter**
- The type parameter sets the data type of all objects stored in a collection

# Generic Collections (cont.)

20

- The general declaration for generic collection is

```
CollectionClassName<E> variable =  
    new CollectionClassName<E>();
```

- The <E> indicates a type parameter
- Adding a noncompatible type to a generic collection will generate an error during compile time
- However, primitive types will be autoboxed:

```
ArrayList<Integer> myList = new ArrayList<Integer>();  
myList.add(new Integer(3)); // ok  
myList.add(3); // also ok! 3 is automatically wrapped  
                in an Integer object  
myList.add(new String("Hello")); // generates a type  
                                incompatibility error
```

# Why Use Generic Collections?

21

- Better type-checking: catch more errors, catch them earlier

```
// without Generics
List list = new ArrayList();
list.add("hello");

// With Generics
List<Integer> list = new ArrayList<Integer>();
list.add("hello"); // will not compile
```

- Documents intent
- Avoids the need to downcast from Object

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

# Specification of the ArrayList Class

22

Method	Behavior
<code>public E get(int index)</code>	Returns a reference to the element at position <code>index</code> .
<code>public E set(int index, E anEntry)</code>	Sets the element at position <code>index</code> to reference <code>anEntry</code> . Returns the previous value.
<code>public int size()</code>	Gets the current size of the <code>ArrayList</code> .
<code>public boolean add(E anEntry)</code>	Adds a reference to <code>anEntry</code> at the end of the <code>ArrayList</code> . Always returns <code>true</code> .
<code>public void add(int index, E anEntry)</code>	Adds a reference to <code>anEntry</code> , inserting it before the item at position <code>index</code> .
<code>int indexOf(E target)</code>	Searches for <code>target</code> and returns the position of the first occurrence, or <code>-1</code> if it is not in the <code>ArrayList</code> .
<code>public E remove(int index)</code>	Returns and removes the item at position <code>index</code> and shifts the items that follow it to fill the vacated space.

# Applications of ArrayList

## Section 2.3

# Example Application of ArrayList

24

```
ArrayList<Integer> someInts = new ArrayList<Integer>();
int[] nums = {5, 7, 2, 15};
for (int i = 0; i < nums.length; i++) {
    someInts.add(nums[i]);
}

// Display the sum
int sum = 0;
for (int i = 0; i < someInts.size(); i++) {
    sum += someInts.get(i);
}
System.out.println("sum is " + sum);
```

# Phone Directory Application

25

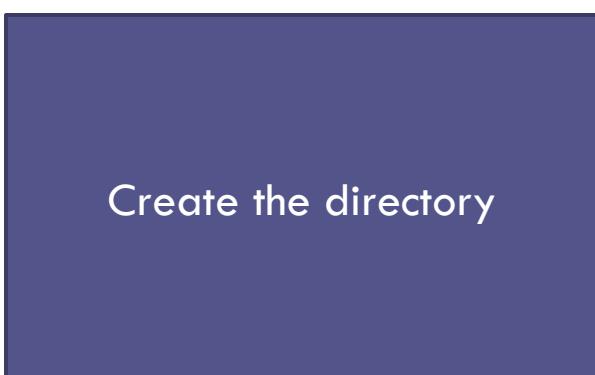
```
public class DirectoryEntry {  
    String name;  
    String number;  
}
```

Create a class for objects  
stored in the directory

# Phone Directory Application (cont.)

26

```
public class DirectoryEntry {  
    String name;  
    String number;  
}  
  
private ArrayList<DirectoryEntry> theDirectory =  
    new ArrayList<DirectoryEntry>();
```



Create the directory

# Phone Directory Application (cont.)

27

```
public class DirectoryEntry {  
    String name;  
    String number;  
}
```

Add a  
DirectoryEntry  
object

```
private ArrayList<DirectoryEntry> theDirectory =  
    new ArrayList<DirectoryEntry>();
```

```
theDirectory.add(new DirectoryEntry("Jane Smith",  
    "555-1212"));
```

# Phone Directory Application (cont.)

28

```
public class DirectoryEntry {  
    String name;  
    String number;  
}  
  
private ArrayList<DirectoryEntry> theDirectory =  
    new ArrayList<DirectoryEntry>();  
  
theDirectory.add(new DirectoryEntry("Jane Smith",  
    "555-1212"));  
  
int index = theDirectory.indexOf(new DirectoryEntry(aName, ""));
```

Method `indexOf` searches theDirectory by applying the `equals` method for class `DirectoryEntry`. Assume `DirectoryEntry`'s `equals` method compares name fields.

# Phone Directory Application (cont.)

29

...

```
int index = theDirectory.indexOf(new  
    DirectoryEntry(aName, ""));  
  
if (index != -1)  
    dE = theDirectory.get(index);  
else  
    dE = null;
```

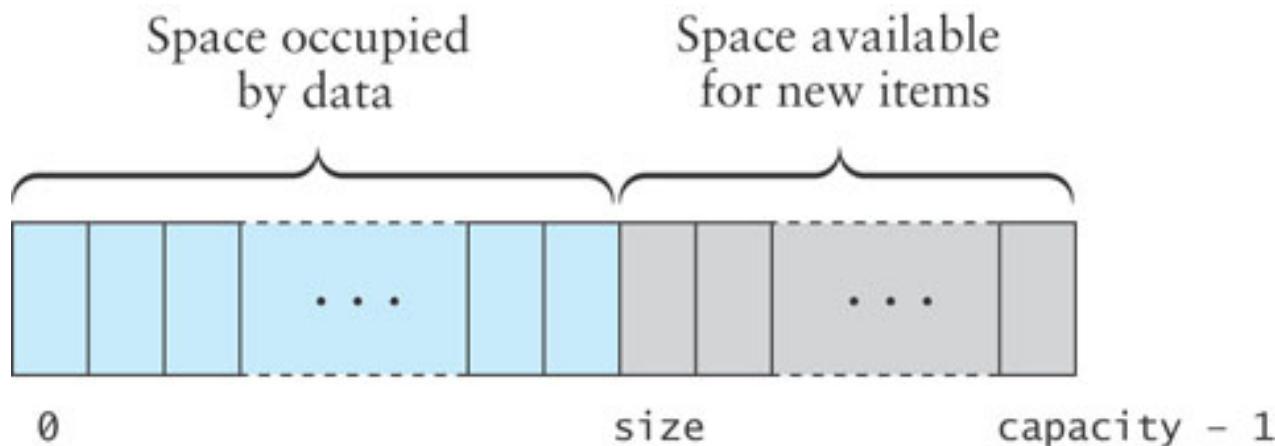
# Implementation of an ArrayList Class

## Section 2.4

# Implementing an ArrayList Class

31

- KWArrayList: a simple implementation of ArrayList
  - ▣ Physical size of array indicated by data field **capacity**
  - ▣ Number of data items indicated by the data field **size**



# KWArrayList Fields

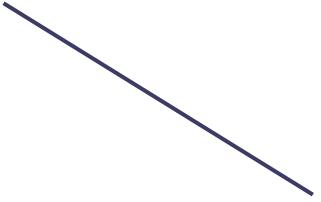
32

```
import java.util.*;  
  
/** This class implements some of the methods of the Java ArrayList  
class */  
public class KWArrayList<E> {  
    // Data fields  
    /** The default initial capacity */  
    private static final int INITIAL_CAPACITY = 10;  
  
    /** The underlying data array */  
    private E[] theData;  
  
    /** The current size */  
    private int size = 0;  
  
    /** The current capacity */  
    private int capacity = 0;  
}
```

# KWArrayList Constructor

33

```
public KWArrayList () {  
    capacity = INITIAL_CAPACITY;  
    theData = (E[]) new Object[capacity];  
}
```



This statement allocates storage for an array of type Object and then casts the array object to type E[]

Although this may cause a compiler warning, it's ok

# Implementing ArrayList.add(E)

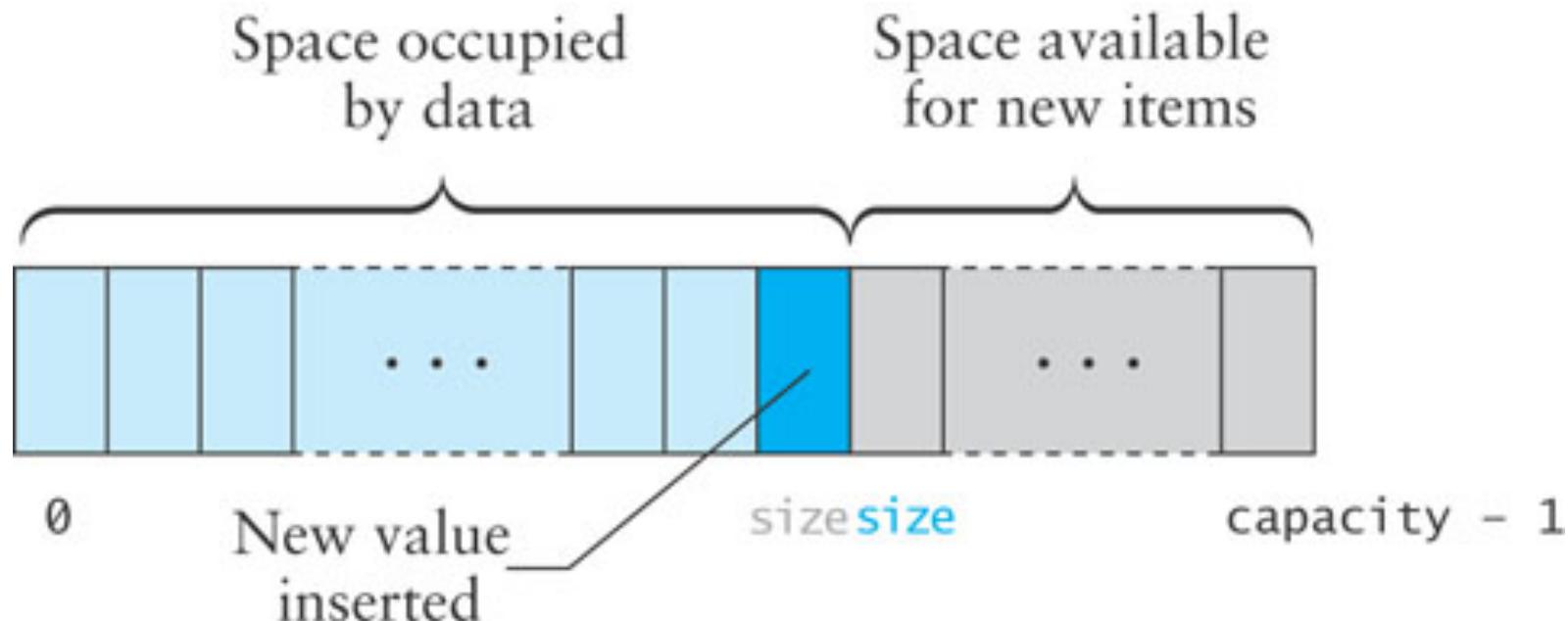
34

- We will implement two add methods
- One will append at the end of the list
- The other will insert an item at a specified position

# Implementing ArrayList.add(E) (cont.)

35

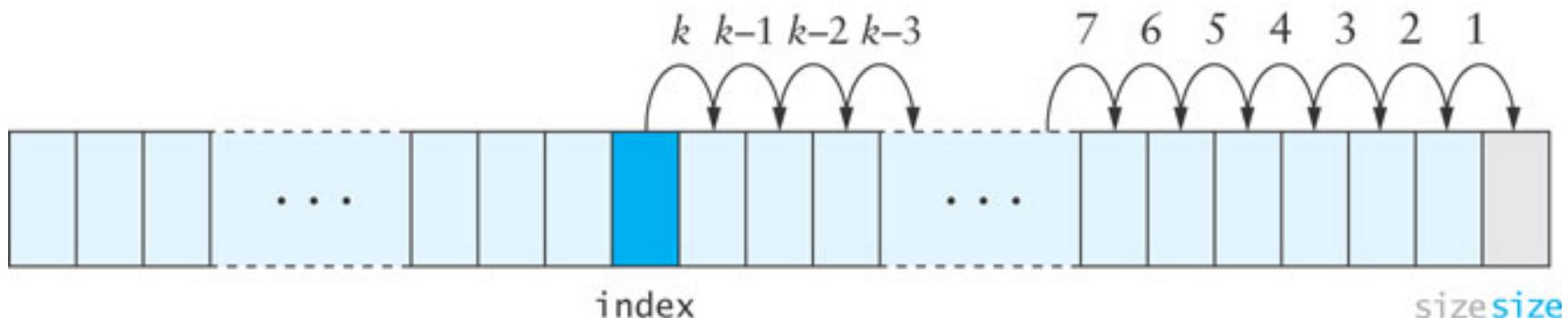
- If size is less than capacity, then to append a new item
  1. insert the new item at the position indicated by the value of size
  2. increment the value of size
  3. return true to indicate successful insertion



# Implementing ArrayList.add(int index, E anEntry)

36

- To insert into the middle of the array, the values at the insertion point are shifted over to make room, beginning at the end of the array and proceeding in the indicated order



# Implementing ArrayList.add(index, E)

37

```
public void add (int index, E anEntry) {  
    // check bounds  
    if (index < 0 || index > size) {  
        throw new ArrayIndexOutOfBoundsException(index);  
    }  
  
    // Make sure there is room  
    if (size >= capacity) {  
        reallocate();  
    }  
  
    // shift data  
    for (int i = size; i > index; i--) {  
        theData[i] = theData[i-1];  
    }  
  
    // insert item  
    theData[index] = anEntry;  
    size++;  
}
```

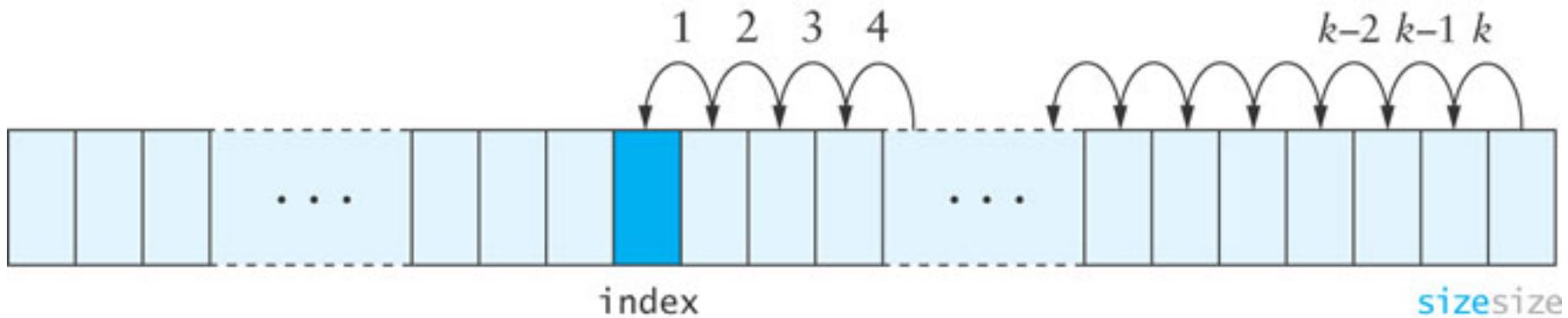
# set and get Methods

38

```
public E get (int index) {  
    if (index < 0 || index >= size) {  
        throw new ArrayIndexOutOfBoundsException(index);  
    }  
    return theData[index];  
}  
  
public E set (int index, E newValue) {  
    if (index < 0 || index >= size) {  
        throw new ArrayIndexOutOfBoundsException(index);  
    }  
    E oldValue = theData[index];  
    theData[index] = newValue;  
    return oldValue;  
}
```

# remove Method

39



- When an item is removed, the items that follow it must be moved forward to close the gap
- Begin with the item closest to the removed element and proceed in the indicated order

# remove Method (cont.)

40

```
public E remove (int index) {  
  
    if (index < 0 || index >= size) {  
        throw new ArrayIndexOutOfBoundsException(index);  
    }  
  
    E returnValue = theData[index];  
  
    ??????    Fill in the blank    ???????????  
  
    size--;  
    return returnValue;  
}
```

# reallocate **Method**

41

- Create a new array that is twice the size of the current array and then copy the contents of the new array

```
private void reallocate () {  
    capacity *= 2;  
    theData = Arrays.copyOf(theData,  
    capacity);  
}
```

# KWArrayList as a Collection of Objects

42

- Earlier versions of Java did not support generics; all collections contained only Object elements
- To implement KWArrayList this way,
  - ▣ remove the parameter type <E> from the class heading,
  - ▣ replace each reference to data type E by Object
  - ▣ The underlying data array becomes

```
private Object[] theData;
```

# Performance of KWArrayList

43

- The set and get methods execute in constant time:  $O(1)$
- Inserting or removing general elements is linear time:  $O(n)$
- Adding at the end is (usually) constant time:  $O(1)$ 
  - With our reallocation technique the average is  $O(1)$
  - The worst case is  $O(n)$  because of reallocation



# CS 570: Data Structures

## Collections Framework:

### Linked Lists

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 2 (PART 2)

Lists and the  
Collections Framework

# Week 5

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 2.5-2.10

# Single-Linked Lists

## Section 2.5

# Single-Linked Lists

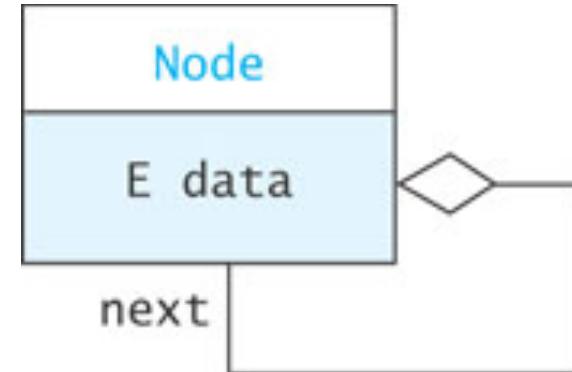
5

- A linked list is useful for inserting and removing at arbitrary locations
- The ArrayList is limited because its add and remove methods operate in linear ( $O(n)$ ) time—requiring a loop to shift elements
- A linked list can add and remove elements at a known location in  $O(1)$  time
- In a linked list, instead of an index, each element is linked to the following element

# A List Node

6

- A node can contain:
  - a data item
  - one or more links
- A link is a reference to a list node
- In our structure, the node contains a data field named data of type E
- and a reference to the next node, named next



# List Nodes for Single-Linked Lists

7

```
private static class Node<E> {  
    private E data;  
    private Node<E> next;  
  
    /** Creates a new node with a null next field  
     * @param dataItem The data stored  
     */  
    private Node(E dataItem) {  
        data = dataItem;  
        next = null;  
    }  
}
```

# List Nodes for Single-Linked Lists

8

```
/** Creates a new node that references  
another node  
  
 @param dataItem The data stored  
 @param nodeRef The node referenced by  
 new node  
  
 */  
  
private Node(E dataItem, Node<E> nodeRef) {  
    data = dataItem;  
    next = nodeRef;  
}  
}
```

# List Nodes for Single-Linked Lists

## (cont.)

9

```
private static class Node<E> {  
    private E data;  
    private Node<E> next;  
  
    /** Creates a new node with a  
     * @param dataItem The data  
     */  
    private Node(E data) {  
        data = dataItem;  
        next = null;  
    }  
}
```

The keyword **static** indicates that the `Node<E>` class will not reference its outer class

Static inner classes are also called *nested classes*

# List Nodes for Single-Linked Lists

## (cont.)

10

```
private static class Node<E> {  
    private E data;  
    private Node<E> next;  
  
    /** Creates a new node with a null  
     * @param dataItem The data stored in the node.  
     */  
    private Node(E dataItem) {  
        data = dataItem;  
        next = null;  
    }  
}
```

Generally, all details of the Node class should be private. This applies also to the data fields and constructors.

# Connecting Nodes (cont.)

11

```
Node<String> tom = new Node<String>("Tom");
Node<String> bill = new Node<String>("Bill");
Node<String> harry = new
    Node<String>("Harry");
Node<String> sam = new Node<String>("Sam");

tom.next = bill;
bill.next = harry;
harry.next = sam;
```

# A Single-Linked List Class

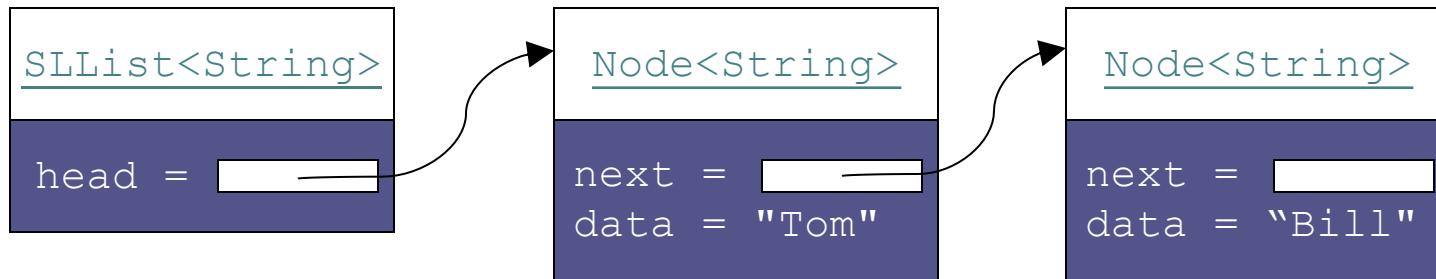
12

- Generally, we do not have individual references to each node
- A `SingleLinkedList` object has a data field `head`, the *list head*, which references the first list node

```
public class SingleLinkedList<E> {  
    private Node<E> head = null;  
    private int size = 0;  
    ...  
}
```

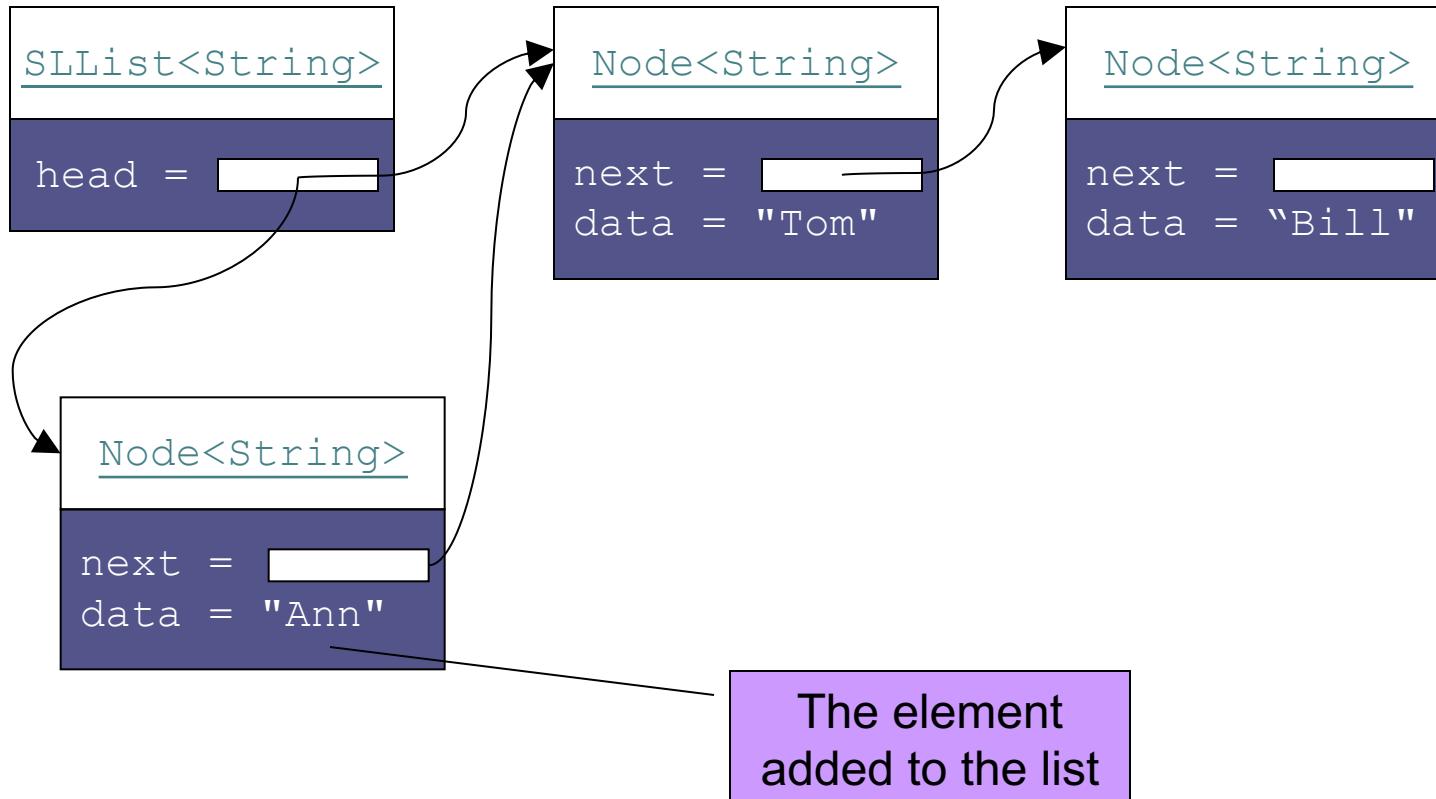
# SLList: An Example List

13



# Implementing SLLList.addFirst(E item)

14



# Implementing SLList.addFirst (E item) (cont.)

15

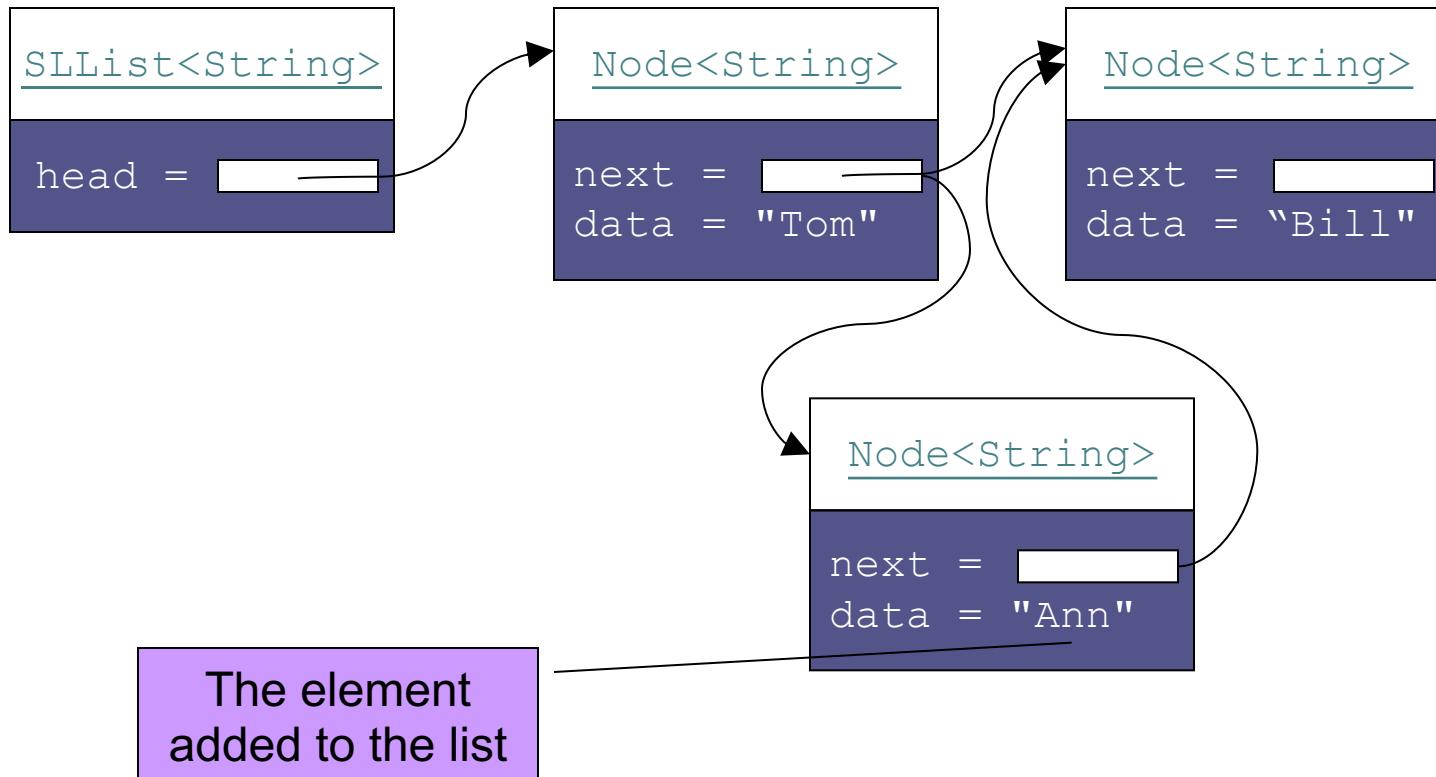
```
private void addFirst (E item) {  
    Node<E> temp = new Node<E>(item, head);  
    head = temp;  
    size++;  
}
```

**or, more simply**

```
private void addFirst (E item) {  
    head = new Node<E>(item, head);  
    size++;  
}
```

# Implementing addAfter (Node<E> node, E item)

16



# Implementing addAfter (Node<E> node, E item) (cont.)

17

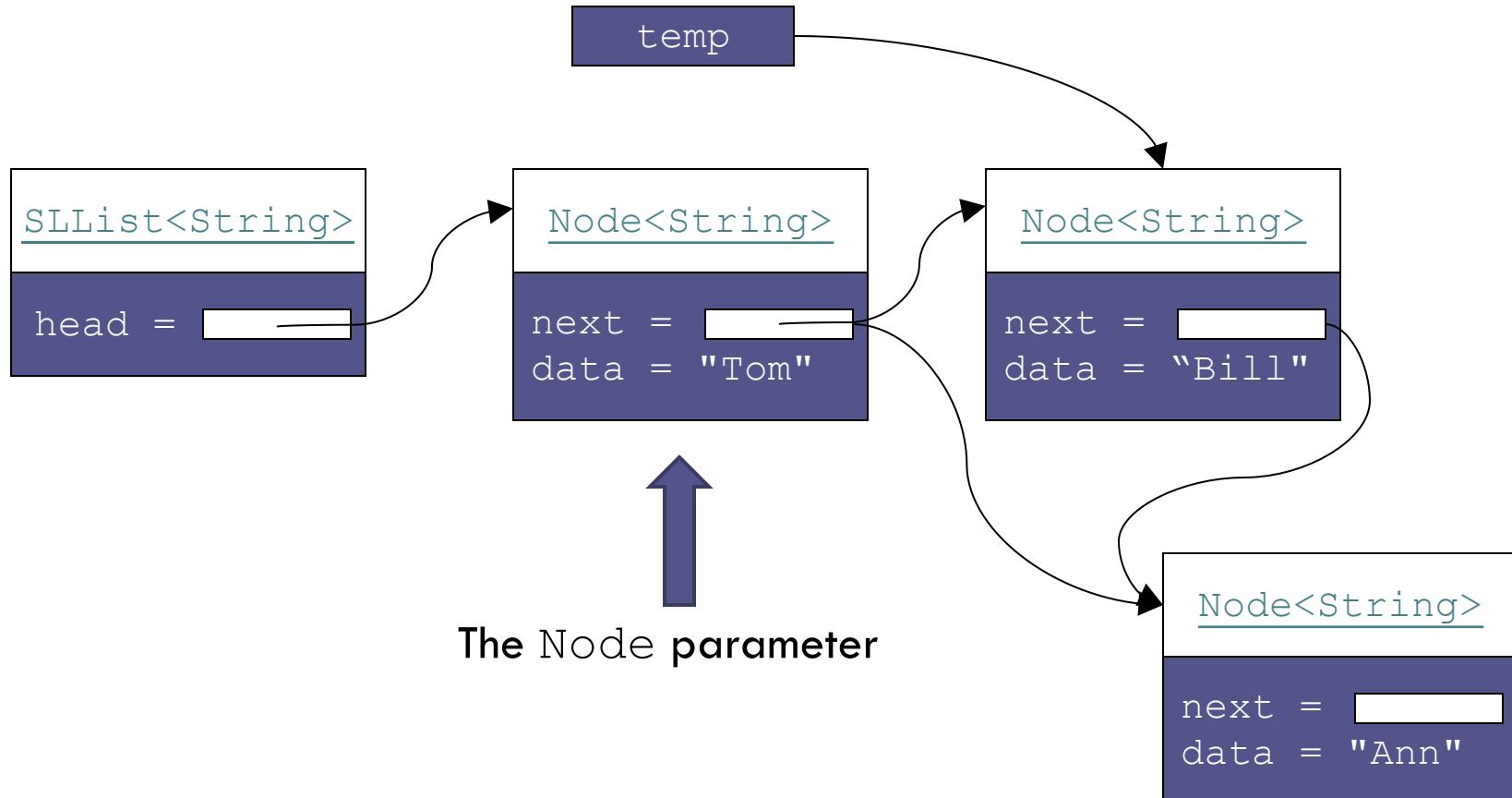
```
private void addAfter (Node<E> node, E item) {  
    Node<E> temp = new Node<E>(item, node.next);  
    node.next = temp;  
    size++;  
}
```

or, more simply ...

We declare this method private since it should not be called from outside the class. Later we will see how this method is used to implement the public add methods.

```
private void addAfter (Node<E> node, E item) {  
    node.next = new Node<E>(item, node.next);  
    size++;  
}
```

# Implementing removeAfter (Node<E> node)

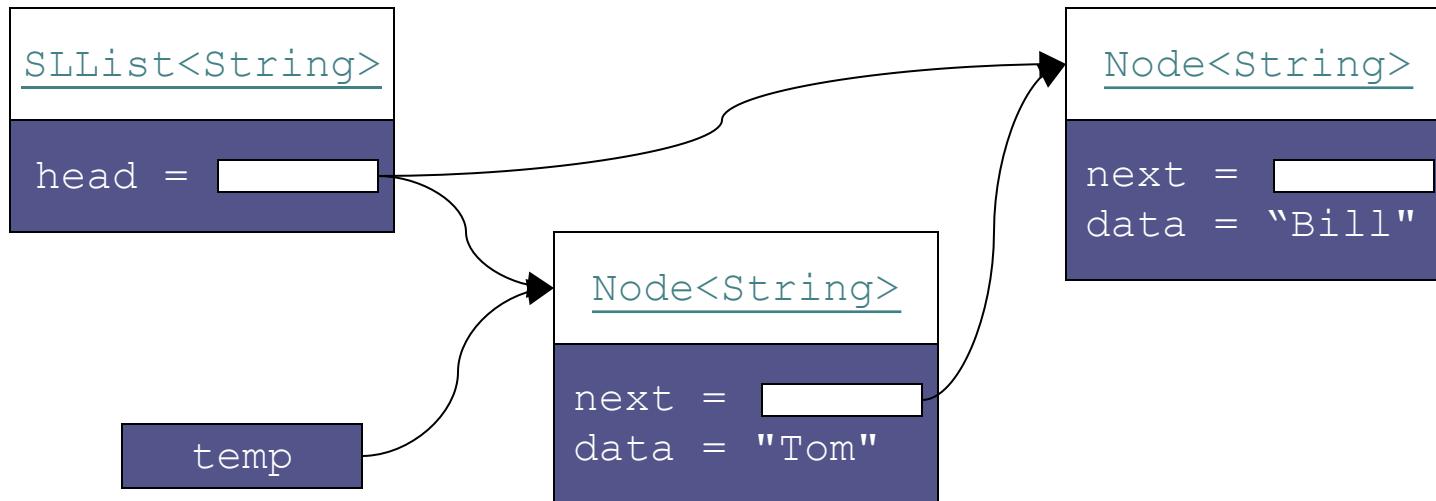


# Implementing removeAfter (Node<E> node) (cont.)

19

```
private E removeAfter (Node<E> node)
{
    Node<E> temp = node.next;
    if (temp != null) {
        node.next = temp.next;
        size--;
        return temp.data;
    } else {
        return null;
    }
}
```

# Implementing SLList.removeFirst()



# Implementing

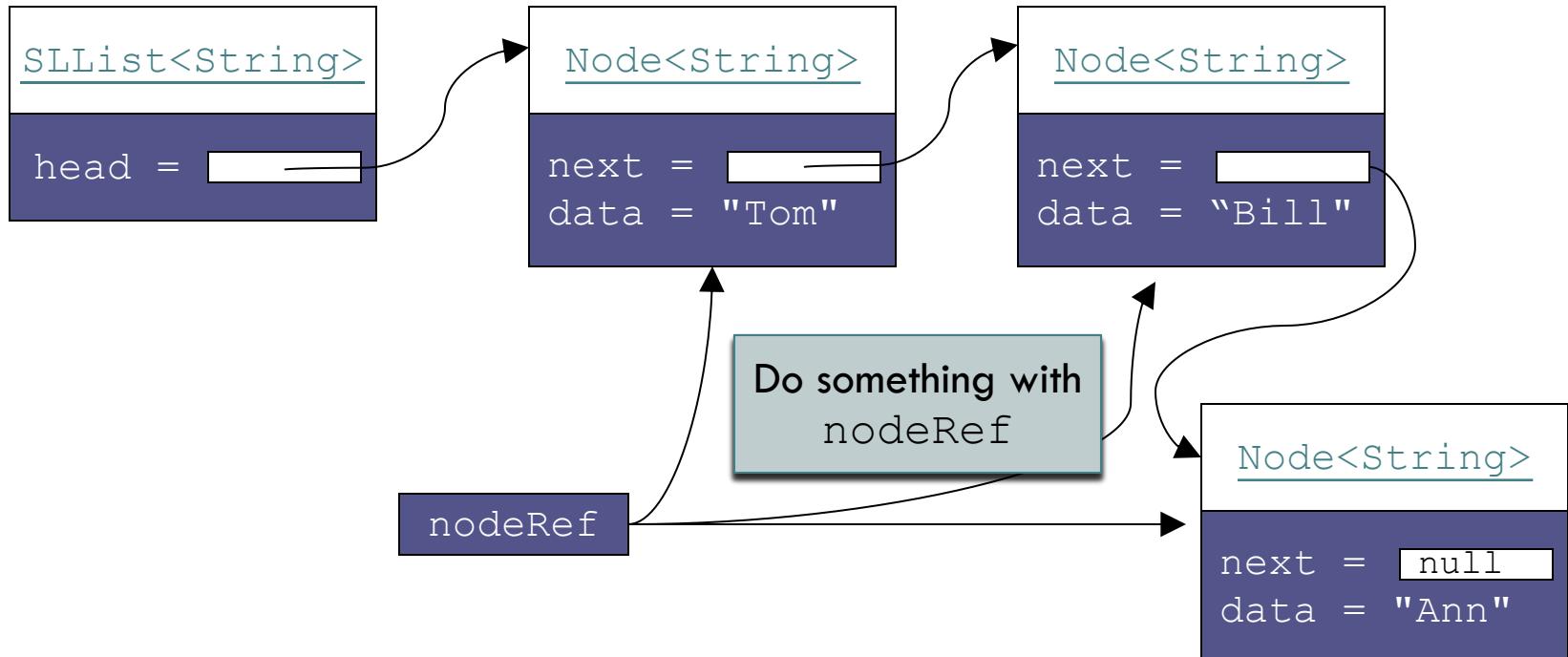
## SLList.removeFirst () (cont.)

21

```
private E removeFirst () {  
    Node<E> temp = head;  
    if (head != null) {  
        head = head.next;  
        size--;  
        return temp.data  
    } else {  
        return null;  
    }  
}
```

# Traversing a Single-Linked List

22



# Traversing a Single-Linked List (cont.)

23

```
public String toString() {  
    Node<String> nodeRef = head;  
    StringBuilder result = new StringBuilder();  
    while (nodeRef != null) {  
        result.append(nodeRef.data);  
        if (nodeRef.next != null) {  
            result.append(" ==> ");  
        }  
        nodeRef = nodeRef.next;  
    }  
    return result.toString();  
}
```

# SLList.getNode(int)

24

- In order to implement methods required by the List interface, we need an additional helper method:

```
private Node<E> getNode(int index) {  
    Node<E> node = head;  
    for (int i=0; i<index && node != null;  
         i++) {  
        node = node.next;  
    }  
    return node;  
}
```

# Completing the SingleLinkedList Class

25

Method	Behavior
public E get(int index)	Returns a reference to the element at position <code>index</code> .
public E set(int index, E anEntry)	Sets the element at position <code>index</code> to reference <code>anEntry</code> . Returns the previous value.
public int size()	Gets the current size of the <code>List</code> .
public boolean add(E anEntry)	Adds a reference to <code>anEntry</code> at the end of the <code>List</code> . Always returns <code>true</code> .
public void add(int index, E anEntry)	Adds a reference to <code>anEntry</code> , inserting it before the item at position <code>index</code> .
int indexOf(E target)	Searches for <code>target</code> and returns the position of the first occurrence, or <code>-1</code> if it is not in the <code>List</code> .

# public E get(int index)

26

```
public E get (int index) {  
    if (index < 0 || index >= size) {  
        throw new  
            IndexOutOfBoundsException(Integer.toString(index));  
    }  
    Node<E> node = getNode(index);  
    return node.data;  
}
```

```
public E set(int index, E newValue)
```

27

```
public E set (int index, E anEntry) {  
    if (index < 0 || index >= size) {  
        throw new  
            IndexOutOfBoundsException(Integer.toString  
                (index));  
    }  
    Node<E> node = getNode(index);  
    E result = node.data;  
    node.data = newValue;  
    return result;  
}
```

```
public void add(int index, E item)
```

28

```
public void add (int index, E item) {  
    if (index < 0 || index > size) {  
        throw new  
            IndexOutOfBoundsException(Integer.toString  
                (index));  
    }  
    if (index == 0) {  
        addFirst(item);  
    } else {  
        Node<E> node = getNode(index-1);  
        addAfter(node, item);  
    }  
}
```

```
public boolean add(E item)
```

29

- To add an item to the end of the list

```
public boolean add (E item) {  
    add(size, item);  
    return true;  
}
```

# Performance of SingleLinkedList

30

- The set **and** get methods:
- Inserting or removing general elements:
- Adding at the beginning:
- Adding at the end:

# Double-Linked Lists and Circular Lists

## Section 2.6

# Double-Linked Lists

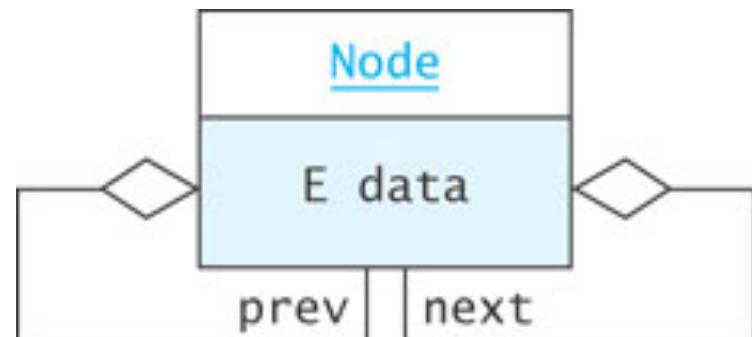
32

- Limitations of a singly-linked list include:
  - ▣ Insertion at the front is  $O(1)$ ; insertion at other positions is  $O(n)$
  - ▣ Insertion is convenient only after a referenced node
  - ▣ Removing a node requires a reference to the previous node
  - ▣ We can traverse the list only in the forward direction
- We can overcome some of these limitations:
  - ▣ Add a reference in each node to the previous node, creating a *double-linked list*

# Node Class

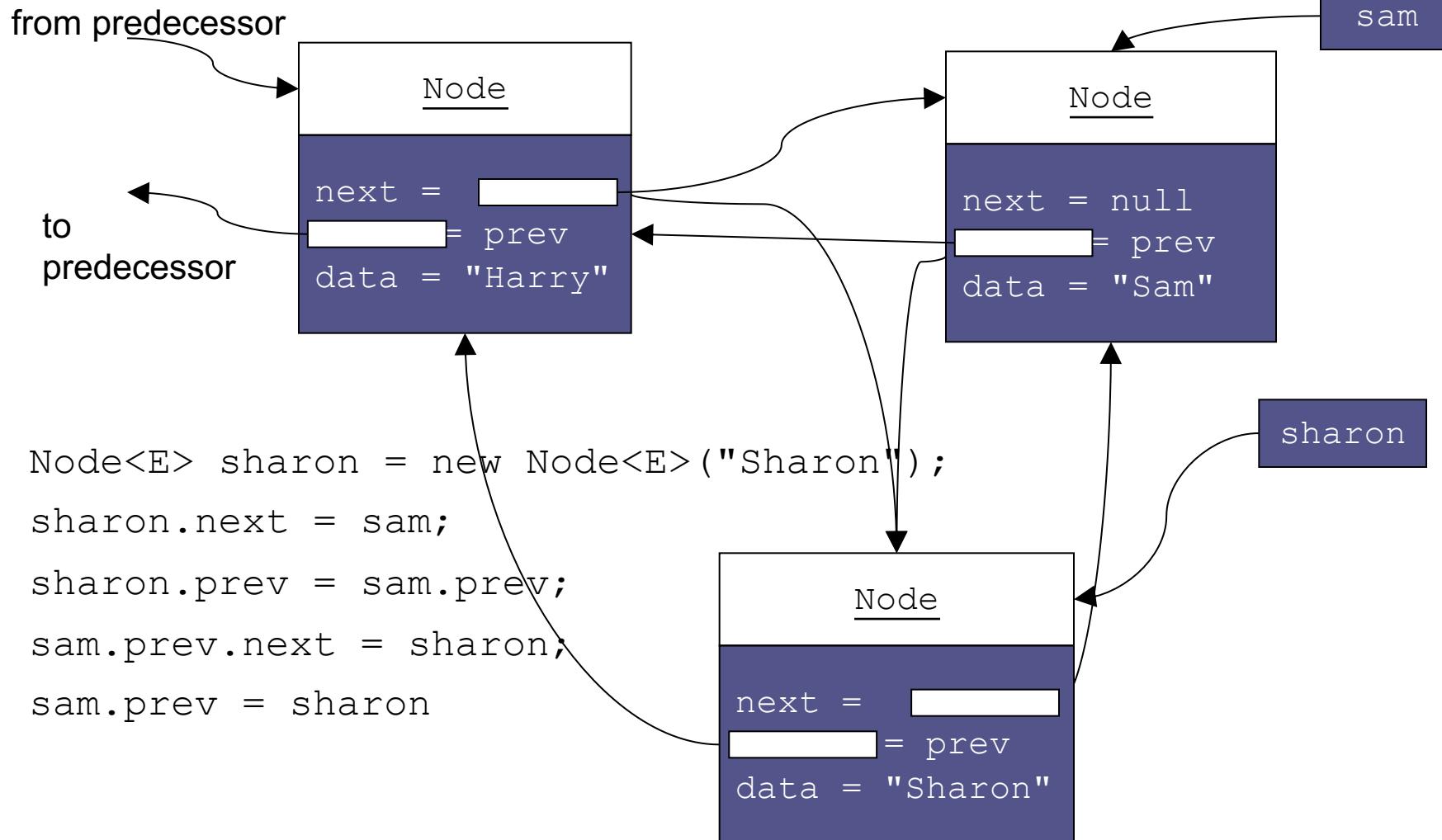
33

```
private static class Node<E> {  
    private E data;  
    private Node<E> next = null;  
    private Node<E> prev = null;  
  
    private Node(E dataItem) {  
        data = dataItem;  
    }  
}
```



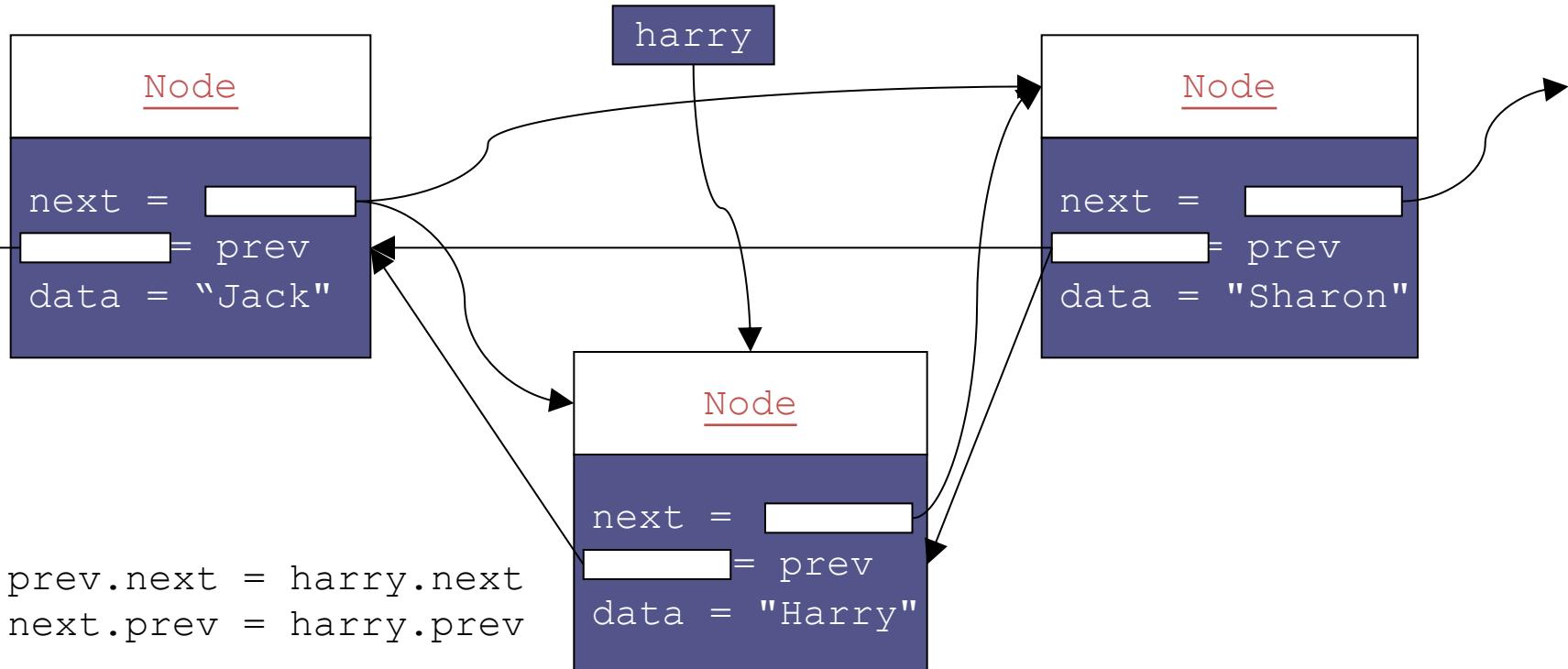
# Inserting into a Double-Linked List

34



# Removing from a Double-Linked List

35



# A Double-Linked List Class

36

- So far we have worked only with internal nodes
- As with the single-linked class, it is best to access the internal nodes with a double-linked list object
- A double-linked list object has data fields:
  - ▣ head (a reference to the first list Node)
  - ▣ tail (a reference to the last list Node)
  - ▣ size
- Insertion at either end is  $O(1)$ ; insertion elsewhere is still  $O(n)$



# Circular Lists

37

- Circular double-linked list:
  - ▣ Link last node to the first node, and
  - ▣ Link first node to the last node
- We can also build singly-linked circular lists:
  - ▣ Traverse in forward direction only
- Advantages:
  - ▣ Continue to traverse even after passing the first or last node
  - ▣ Visit all elements from any starting point
  - ▣ Never fall off the end of a list
- Disadvantage: Code must avoid an infinite loop!

# The LinkedList Class and the Iterator, ListIterator, and Iterable Interfaces

## Section 2.7

# The LinkedList Class

39

Method	Behavior
<code>public void add(int index, E obj)</code>	Inserts object <code>obj</code> into the list at position <code>index</code> .
<code>public void addFirst(E obj)</code>	Inserts object <code>obj</code> as the first element of the list.
<code>public void addLast(E obj)</code>	Adds object <code>obj</code> to the end of the list.
<code>public E get(int index)</code>	Returns the item at position <code>index</code> .
<code>public E getFirst()</code>	Gets the first element in the list. Throws <code>NoSuchElementException</code> if the list is empty.
<code>public E getLast()</code>	Gets the last element in the list. Throws <code>NoSuchElementException</code> if the list is empty.
<code>public boolean remove(E obj)</code>	Removes the first occurrence of object <code>obj</code> from the list. Returns <code>true</code> if the list contained object <code>obj</code> ; otherwise, returns <code>false</code> .
<code>public int size()</code>	Returns the number of objects contained in the list.

# The Iterator

40

- An iterator can be viewed as a moving place marker that keeps track of the current position in a particular linked list
- An Iterator object for a list starts at the list head
- The programmer can move the Iterator by calling its next method.
- The Iterator stays on its current list item until it is needed
- An Iterator traverses in  $O(n)$  while a list traversal using get () calls in a linked list is  $O(n^2)$

# Iterator Interface

41

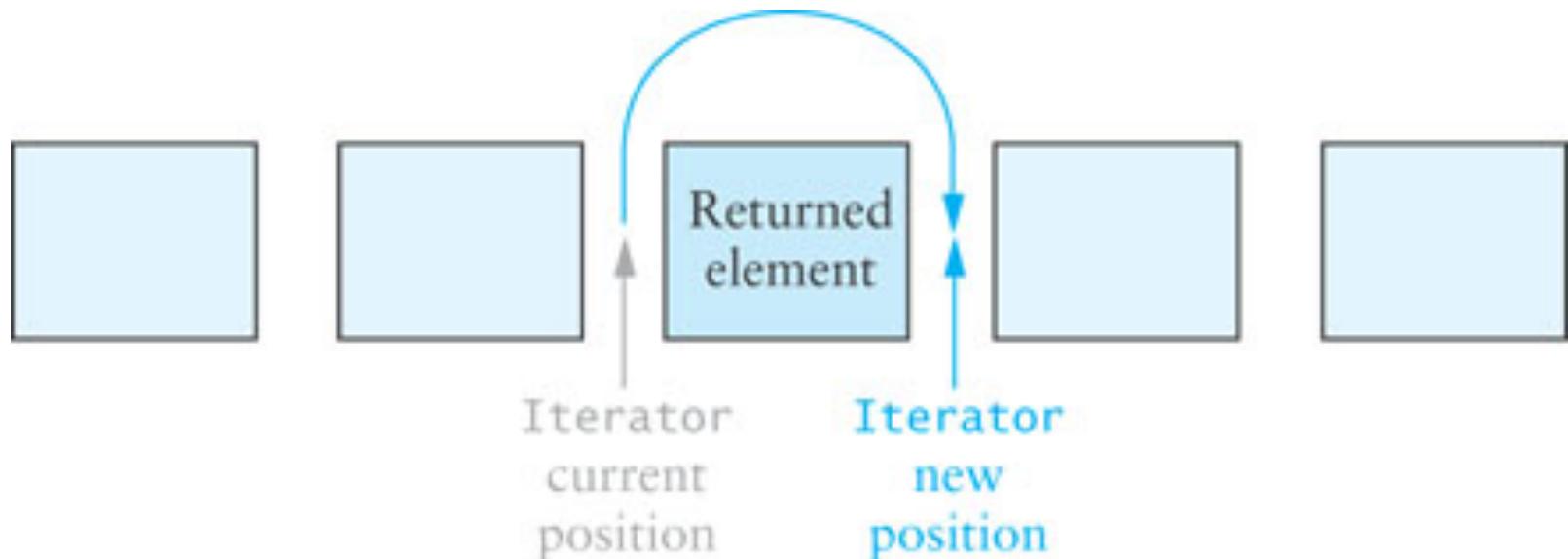
- The Iterator interface is defined in `java.util`
- The List interface declares the method `iterator` which returns an Iterator object that iterates over the elements of that list

Method	Behavior
<code>boolean hasNext()</code>	Returns true if the next method returns a value.
<code>E next()</code>	Returns the next element. If there are no more elements, throws the <code>NoSuchElementException</code> .
<code>void remove()</code>	Removes the last element returned by the <code>next</code> method.

# Iterator Interface (cont.)

42

- An Iterator is conceptually *between elements*; it does not refer to a particular object at any given time



# Iterator Interface (cont.)

43

- In the following loop, we process all items in List<Integer> through an Iterator

```
Iterator<Integer> iter = aList.iterator();
while (iter.hasNext()) {
    int value = iter.next();
    // Do something with value
    ...
}
```

# Enhanced for Statement

44

- Java 5.0 introduced an enhanced for statement
- The enhanced for statement creates an Iterator object and implicitly calls its hasNext and next methods
- Other Iterator methods, such as remove, are not available

# Enhanced for Statement (cont.)

45

- The following code counts the number of times target occurs in myList (type LinkedList<String>)

```
count = 0;  
for (String nextStr : myList) {  
    if (target.equals(nextStr)) {  
        count++;  
    }  
}
```

# Enhanced for Statement (cont.)

46

- The enhanced for statement can also be used with arrays, in this case, chars or type `char [ ]`

```
for (char nextCh : chars) {  
    System.out.println(nextCh);  
}
```

# Implementation of a Double-Linked List Class

## Section 2.8

- Makes heavy use of iterators
- Can also be implemented using previous/next references

# KWLinkedList

48

- We will define a KWLinkedList class which implements some of the methods of the List interface
- The KWLinkedList class is for demonstration purposes only; Java provides a standard LinkedList class in java.util which you should use in your programs (after this course)

Data Field	Attribute
private Node<E> head	A reference to the first item in the list
private Node<E> tail	A reference to the last item in the list
private int size	A count of the number of items in the list

# KWLinkedList (cont.)

49

```
import java.util.*;  
  
/** Class KWLinkedList implements a double linked list  
 * and a ListIterator. */  
  
public class KWLinkedList <E> {  
    // Data Fields  
    private Node <E> head = null;  
  
    private Node <E> tail = null;  
  
    private int size = 0;  
  
    . . .
```

# Add Method

50

1. Obtain a reference, nodeRef, to the node at position index
2. Insert a new Node containing obj before the node referenced by nodeRef

To use a ListIterator object to implement add:

1. Obtain an iterator that is positioned just before the Node at position index
2. Insert a new Node containing obj before the Node currently referenced by this iterator

# Add Method

51

```
/** Add an item at the specified index.  
 * @param index The index at which the object is  
 *               to be inserted  
 * @param obj The object to be  
 *            inserted  
 * @throws IndexOutOfBoundsException  
 *         if the index is out of range  
 *         (i < 0 || i > size())  
 */  
public void add(int index, E obj) {  
    listIterator(index).add(obj);  
}
```

# Other Add and Get Methods

52

```
public void addFirst(E item) {  
    add(0, item);  
}  
  
public void addLast(E item) {  
    add(size, item);  
}  
  
public E getFirst() {  
    return head.data;  
}  
  
public E getLast() {  
    return tail.data;  
}
```

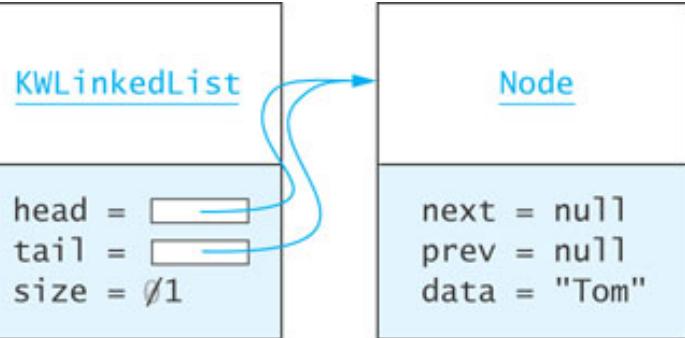
# The Add Method

53

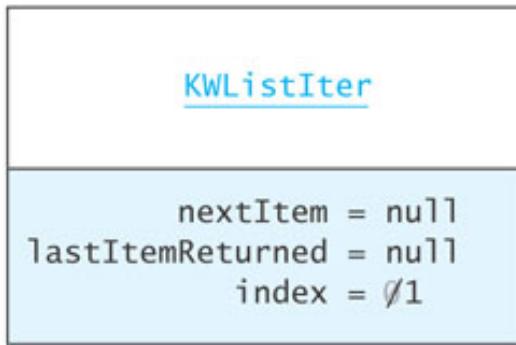
- When adding, there are four cases to address:
  - Add to an empty list
  - Add to the head of the list
  - Add to the tail of the list
  - Add to the middle of the list

# Adding to an Empty List

54



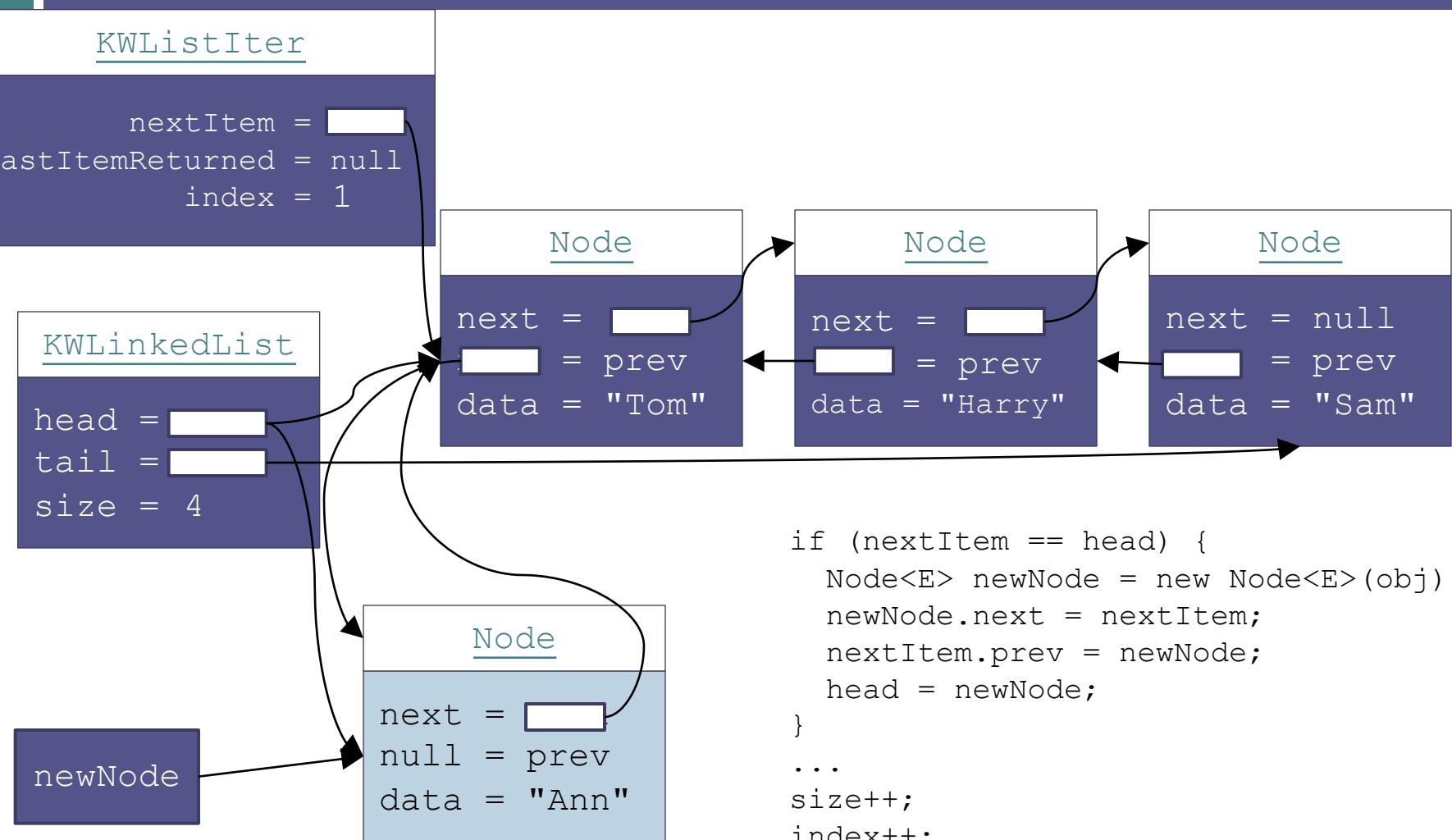
(after insertion)



```
if (head == null) {  
    head = new Node<E>(obj);  
    tail = head;  
}  
...  
size++
```

# Adding to the Head of the List

55



# Adding to the Tail of the List

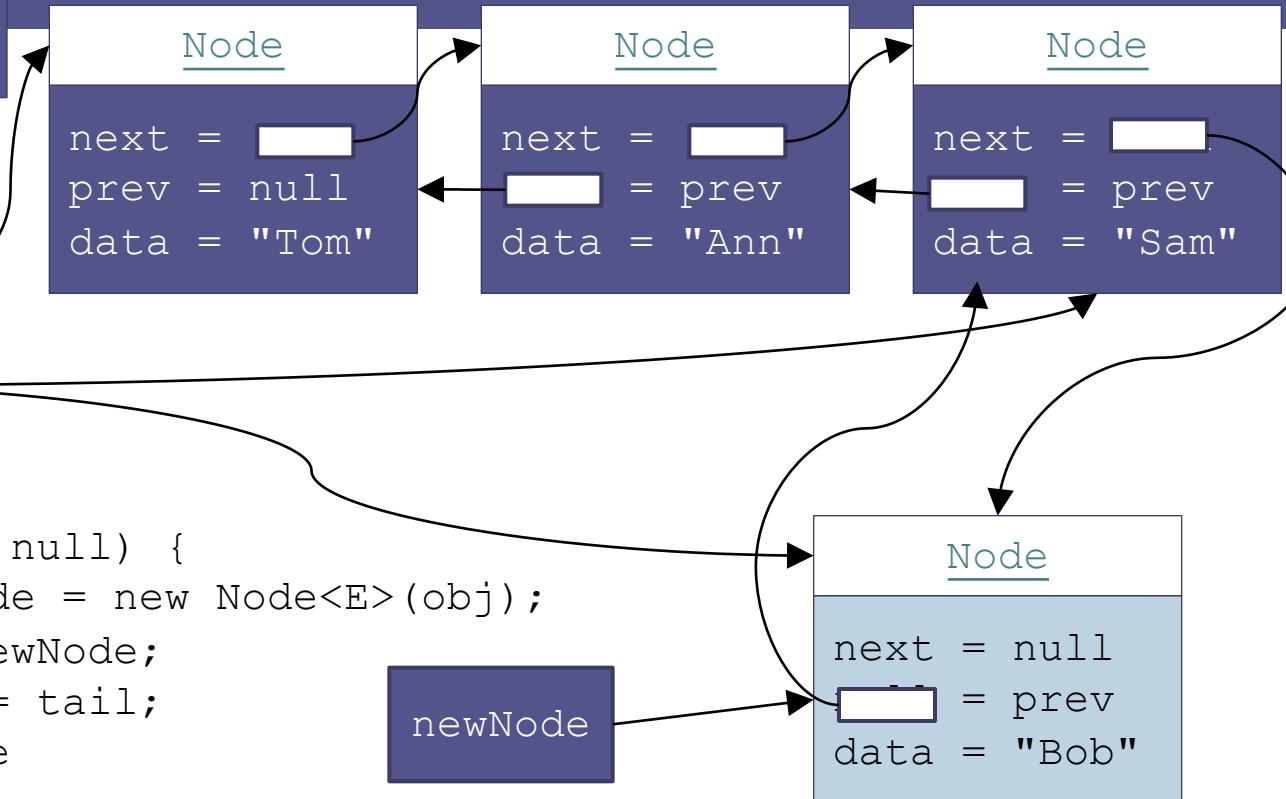
KWListIter

```
nextItem = null  
lastItemReturned = null  
index = 3
```

KWLinkedList

```
head = [ ]  
tail = [ ]  
size = 4
```

```
if (nextItem == null) {  
    Node<E> newNode = new Node<E>(obj);  
    tail.next = newNode;  
    newNode.prev = tail;  
    tail = newNode  
}  
...  
size++;  
index++;
```



# Adding to the Middle of the List

KWListIter

```
nextItem = [ ]  
lastItemReturned = null  
index = 2
```

KWLinkedList

```
head = [ ]  
tail = [ ]  
size = 4
```

```
else {  
    Node<E> newNode = new Node<E>(obj);  
    newNode.prev = nextItem.prev;  
    nextItem.prev.next = newNode;  
    newNode.next = nextItem;  
    nextItem.prev = newNode;  
}  
...  
size++;  
index++;
```

Node

```
next = [ ]  
prev = null  
data = "Tom"
```

Node

```
next = [ ]  
prev = [ ]  
data = "Ann"
```

Node

```
next = null  
prev = [ ]  
data = "Sam"
```

Node

```
next = [ ]  
prev = [ ]  
data = "Bob"
```

newNode

# Inner Classes: Static and Nonstatic

58

- KWLinkedList contains two inner classes:
  - ▣ Node<E> is declared static: there is no need for it to access the data fields of its parent class, KWLinkedList
  - ▣ KWListIter cannot be declared static because its methods access and modify data fields of KWLinkedList's parent object which created it
- An inner class which is not static contains an implicit reference to its parent object and can reference the fields of its parent object

# The Collections Framework Design

## Section 2.9

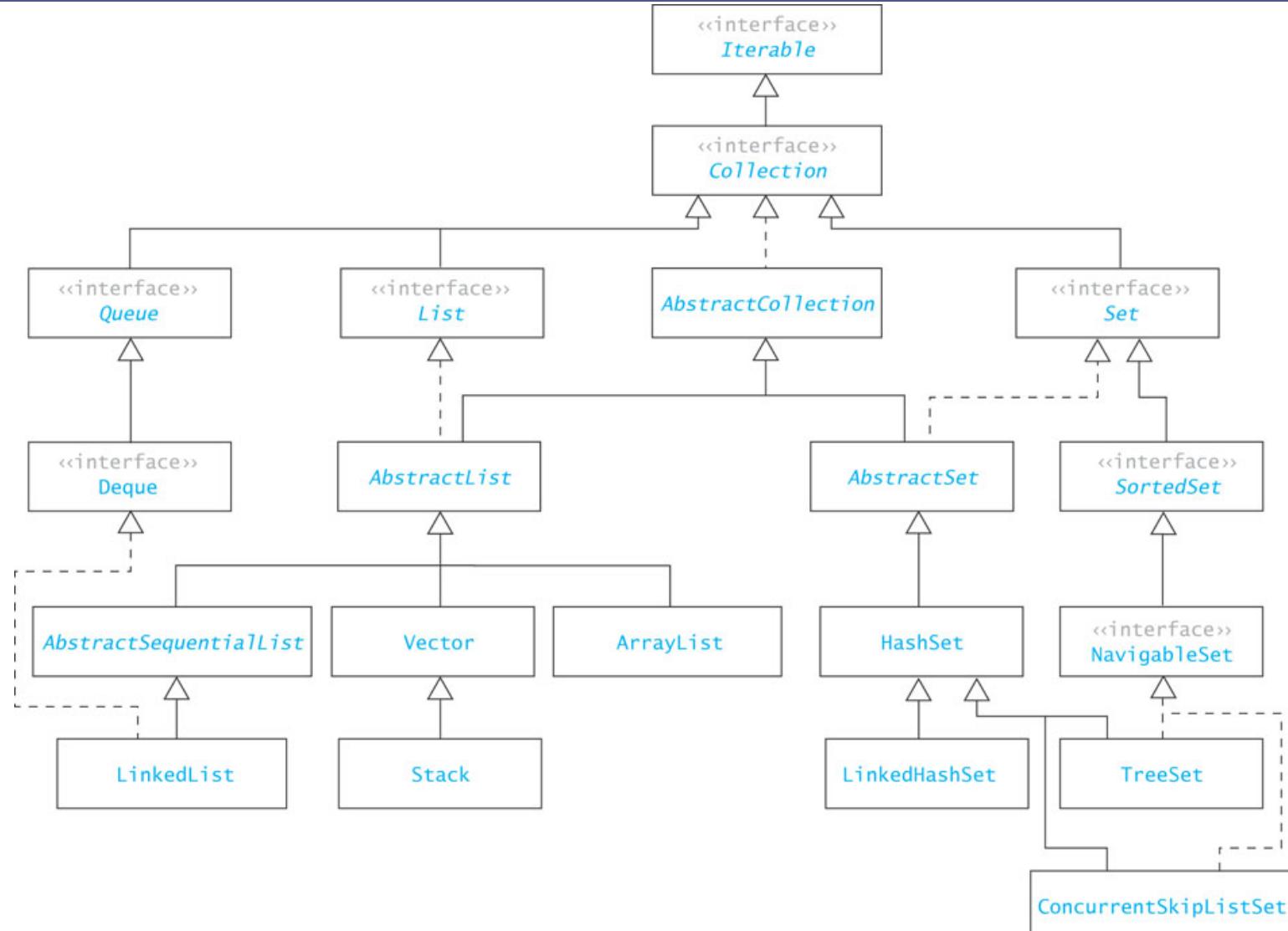
# The Collection Interface

60

- Specifies a subset of methods in the List interface, specifically excluding
  - add(int, E)
  - get(int)
  - remove(int)
  - set(int, E)
- but including**
  - add(E)
  - remove(Object)
  - **the iterator method**

# The Collection Framework

61



# Common Features of Collections

62

- Collections
  - grow as needed
  - hold references to objects
  - have at least two constructors: one to create an empty collection and one to make a copy of another collection

# Application of the LinkedList Class

## Section 2.10

# An Application: Ordered Lists

64

- We want to maintain a list of names in alphabetical order at all times
- **Approach**
  - ▣ Develop an `OrderedList` class (which can be used for other applications)
  - ▣ Implement a Comparable interface by providing a `compareTo(E)` method
  - ▣ Use a `LinkedList` class as a component of the `OrderedList`
    - ▣ if `OrderedList` extended `LinkedList`, the user could use `LinkedList`'s add methods to add an element out of order

# Class Diagram for OrderedList

65



# Design

66

Data Field	Attribute
private LinkedList<E> theList	A linked list to contain the data.
Method	Behavior
public void add(E obj)	Inserts obj into the list preserving the list's order.
public Iterator iterator()	Returns an Iterator to the list.
public E get(int index)	Returns the object at the specified position.
public int size()	Returns the size of the list.
public E remove(E obj)	Removes first occurrence of obj from the list.

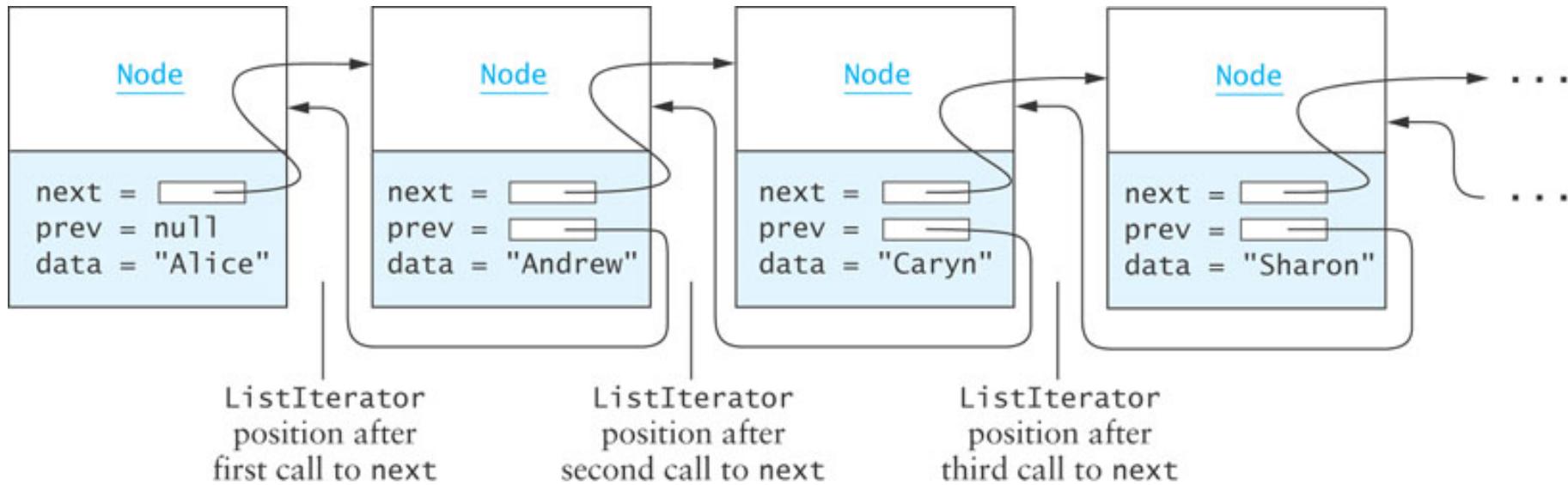
# Inserting into an `OrderedList`

67

- Strategy for inserting new element  $e$ :
  - ▣ Find first item  $> e$
  - ▣ Insert  $e$  before that item
- Refined with an iterator:
  - ▣ Create `ListIterator` that starts at the beginning of the list
  - ▣ While the `ListIterator` is not at the end of the list and  $e \geq$  the next item
    - Advance the `ListIterator`
  - ▣ Insert  $e$  before the current `ListIterator` position

# Inserting Diagrammed

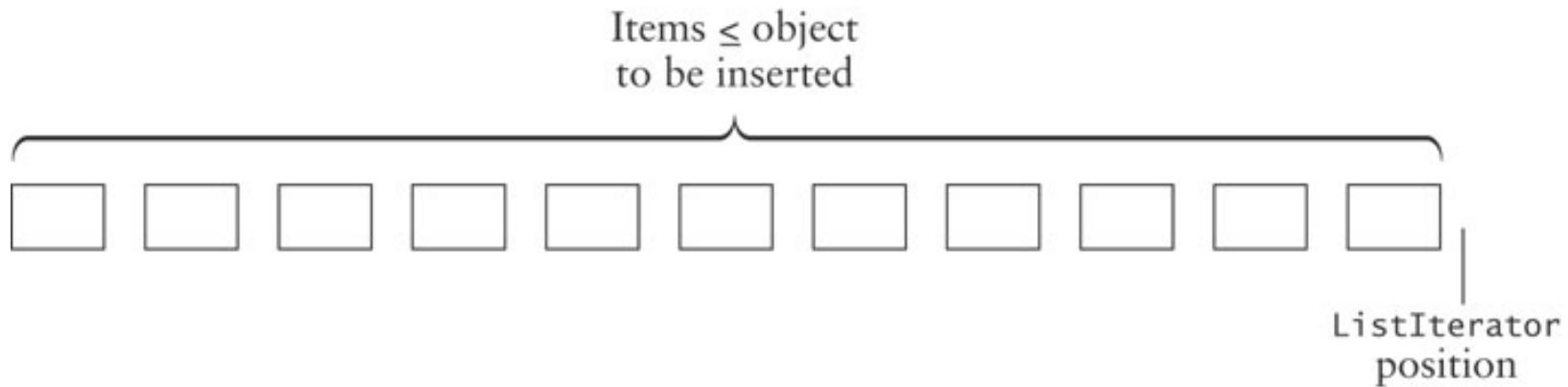
68



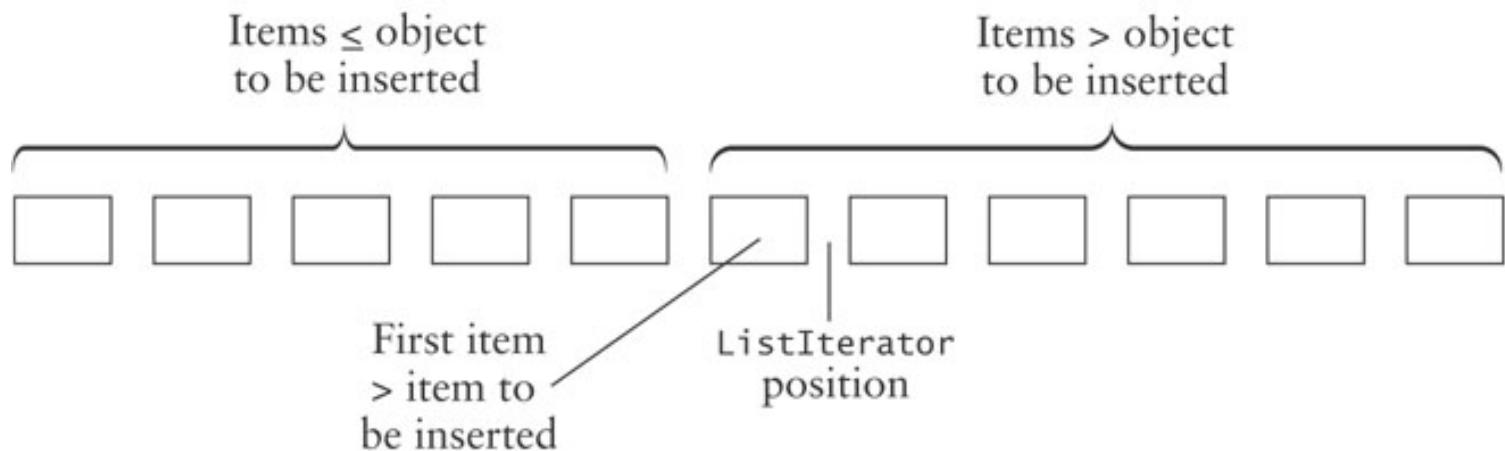
# Inserting Diagrammed (cont.)

69

## Case 1: Inserting at the end of a list



## Case 2: Inserting in the middle of a list



# OrderedList.add

70

```
public void add (E e) {  
    ListIterator<E> iter = theList.listIterator();  
    while (iter.hasNext()) {  
        if (e.compareTo(iter.next()) < 0) {  
            // found element > new one  
            iter.previous(); // back up by one  
            iter.add(e); // add new one  
            return; // done  
        }  
    }  
    iter.add(e); // will add at end  
}
```

# Using Delegation to Implement the Other Methods

71

```
public E get (int index) {  
    return theList.get(index);  
}  
  
public int size () {  
    return theList.size();  
}  
  
public E remove (E e) {  
    return theList.remove(e);  
}
```

# Testing `OrderedList`

72

- To test an `OrderedList`,
  - ▣ store a collection of randomly generated integers in an `OrderedList`
  - ▣ test insertion at beginning of list: insert a negative integer
  - ▣ test insertion at end of list: insert an integer larger than any integer in the list
  - ▣ create an iterator and iterate through the list, displaying an error if any element is smaller than the previous element
  - ▣ remove the first element, the last element, and a middle element, then traverse to show that order is maintained

# Testing `OrderedList` (cont.)

73

```
Class TestOrderedList

import java.util.*;

public class TestOrderedList {
    /** Traverses ordered list and displays each
     * element.
     * Displays an error message if an element is out of
     * order.
     * @param testList An ordered list of integers
    */
```

# Testing `OrderedList` (cont.)

74

```
public static void traverseAndShow(OrderedList<Integer>
        testList) {
    int prevItem = testList.get(0);

    // Traverse ordered list and display any value that
    // is out of order.
    for (int thisItem : testList) {
        System.out.println(thisItem);
        if (prevItem > thisItem)
            System.out.println("*** FAILED, value is "
                               + thisItem);
        prevItem = thisItem;
    }
}
```

Think of how this can be done using references

# Testing `OrderedList` (cont.)

75

```
public static void main(String[] args) {  
    OrderedList<Integer> testList = new  
        OrderedList<Integer>();  
    final int MAX_INT = 500;  
    final int START_SIZE = 100;  
  
    // Create a random number generator.  
    Random random = new Random();  
    for (int i = 0; i < START_SIZE; i++) {  
        int anInteger = random.nextInt(MAX_INT);  
        testList.add(anInteger);  
    }  
}
```

# Testing `OrderedList` (cont.)

76

```
// Add to beginning and end of list.  
testList.add(-1);  
testList.add(MAX_INT + 1);  
traverseAndShow(testList); // Traverse and display.  
  
// Remove first, last, and middle elements.  
Integer first = testList.get(0);  
Integer last = testList.get(testList.size() - 1);
```

# Testing `OrderedList` (cont.)

77

```
Integer middle = testList.get(testList.size() / 2);
testList.remove(first);
testList.remove(last);
testList.remove(middle);
traverseAndShow(testList); // Traverse and display.
}
}
```



# CS 570: Data Structures

## Stacks

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 3

Stacks

# Chapter Objectives

3

- To learn about the stack data type and how to use its four methods:
  - push
  - pop
  - peek
  - empty
- To learn how to implement a stack using an underlying array or linked list
- To see how to use a stack to perform various applications, including finding palindromes, testing for balanced (properly nested) parentheses, and evaluating arithmetic expressions

# Week 6

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 4.1-4.4

# Stack Abstract Data Type

## Section 4.1

# Stack Abstract Data Type

6

- A stack is one of the most commonly used data structures in computer science
- A stack can be compared to a Pez dispenser
  - Only the top item can be accessed
  - You can extract only one item at a time
- The top element in the stack is the one added to the stack most recently
- The stack's storage policy is *Last-In, First-Out, or LIFO*



# Specification of the Stack Abstract Data Type

7

- Only the top element of a stack is visible; therefore the number of operations performed by a stack are few
- We need the ability to
  - test for an empty stack (`empty`)
  - inspect the top element (`peek`)
  - retrieve the top element (`pop`)
  - put a new element on the stack (`push`)

Methods	Behavior
<code>boolean empty()</code>	Returns <code>true</code> if the stack is empty; otherwise, returns <code>false</code> .
<code>E peek()</code>	Returns the object at the top of the stack without removing it.
<code>E pop()</code>	Returns the object at the top of the stack and removes it.
<code>E push(E obj)</code>	Pushes an item onto the top of the stack and returns the item pushed.

# A Stack of Strings

8

Jonathan
Dustin
Robin
Debbie
Rich

(a)

Dustin
Robin
Debbie
Rich

(b)

Philip
Dustin
Robin
Debbie
Rich

(c)

- “Rich” is the oldest element on the stack and “Jonathan” is the youngest (Figure a)
- `String last = names.peek();` stores a reference to “Jonathan” in last
- `String temp = names.pop();` removes “Jonathan” and stores a reference to it in temp (Figure b)
- `names.push(“Philip”);` pushes “Philip” onto the stack (Figure c)

# Stack Applications

## Section 4.2

# Finding Palindromes

10

- **Palindrome:** a string that reads identically in either direction, letter by letter (ignoring case)
  - ▣ kayak
  - ▣ "I saw I was I"
  - ▣ “Able was I ere I saw Elba”
  - ▣ "Level, madam, level"
- **Problem:** Write a program that reads a string and determines whether it is a palindrome

# Finding Palindromes (cont.)

11

Data Fields	Attributes
private String inputString	The input string.
private Stack<Character> charStack	The stack where characters are stored.
Methods	Behavior
public PalindromeFinder(String str)	Initializes a new <code>PalindromeFinder</code> object, storing a reference to the parameter <code>str</code> in <code>inputString</code> and pushing each character onto the stack.
private void fillStack()	Fills the stack with the characters in <code>inputString</code> .
private String buildReverse()	Returns the string formed by popping each character from the stack and joining the characters. Empties the stack.
public boolean isPalindrome()	Returns <code>true</code> if <code>inputString</code> and the string built by <code>buildReverse</code> have the same contents, except for case. Otherwise, returns <code>false</code> .

# Finding Palindromes (cont.)

12

```
import java.util.*;  
  
public class PalindromeFinder {  
    private String inputString;  
    private Stack<Character> charStack = new  
        Stack<Character>();  
  
    public PalindromeFinder(String str) {  
        inputString = str;  
        fillStack();  
    }  
    ...
```

# Finding Palindromes (cont.)

13

## □ Solving using a stack:

- Push each string character, from left to right, onto a stack



k a y a k

```
private void fillStack() {  
    for(int i = 0; i < inputString.length(); i++) {  
        charStack.push(inputString.charAt(i));  
    }  
}
```

# Finding Palindromes (cont.)

14

## □ Solving using a stack:

- Pop each character off the stack, appending each to the StringBuilder result



k a y a k

```
private String buildReverse() {
    StringBuilder result = new StringBuilder();
    while(!charStack.empty()) {
        result.append(charStack.pop());
    }
    return result.toString();
}
```

# Finding Palindromes (cont.)

15

...

```
public boolean isPalindrome() {  
    return inputString.equalsIgnoreCase(buildReverse());  
}  
}
```

# Testing

16

- We can test this class using the following inputs:
  - a single character (always a palindrome)
  - multiple characters in a word
  - multiple words
  - different cases
  - even-length strings
  - odd-length strings
  - the empty string (considered a palindrome)

# Balanced Parentheses

17

- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

$$( a + b * ( c / ( d - e ) ) ) + ( d / e )$$

- The problem is further complicated if braces or brackets are used in conjunction with parentheses
- The solution is to use stacks!

# Balanced Parentheses (cont.)

18

Method	Behavior
public static boolean isBalanced(String expression)	Returns <b>true</b> if expression is balanced with respect to parentheses and <b>false</b> if it is not.
private static boolean isOpen(char ch)	Returns <b>true</b> if ch is an opening parenthesis.
private static boolean isClose(char ch)	Returns <b>true</b> if ch is a closing parenthesis.

# Balanced Parentheses (cont.)

19

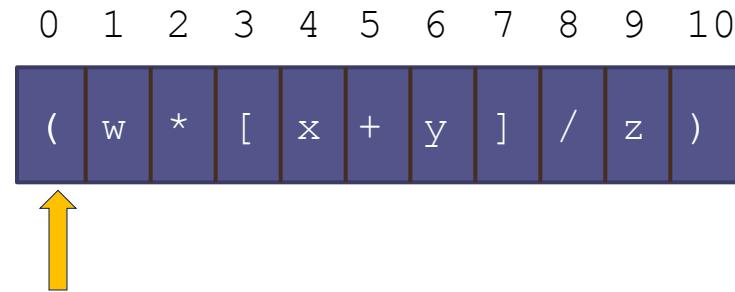
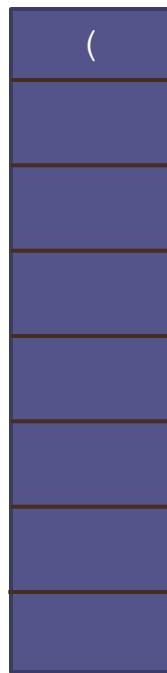
## Algorithm for method `isBalanced`

1. Create an empty stack of characters.
2. Assume that the expression is balanced (`balanced` is `true`).
3. Set index to 0.
4. `while balanced` is `true` and `index < the expression's length`
  - 5. Get the next character in the data string.
  - 6. `if` the next character is an opening parenthesis
    - 7. Push it onto the stack.
  - 8. `else if` the next character is a closing parenthesis
    - 9. Pop the top of the stack.
  - 10. `if` stack was empty or its top does not match the closing parenthesis
    - 11. Set `balanced` to `false`.
  - 12. Increment index.
13. Return `true` if `balanced` is `true` and the stack is empty.

# Balanced Parentheses (cont.)

20

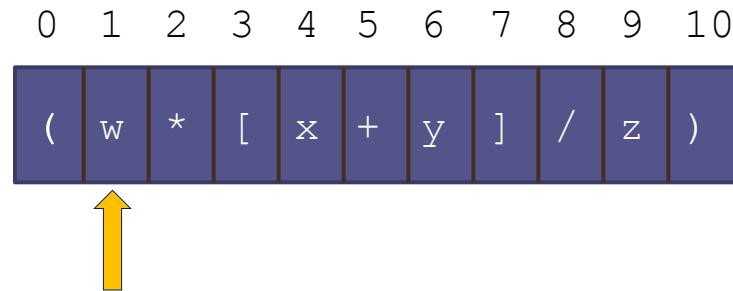
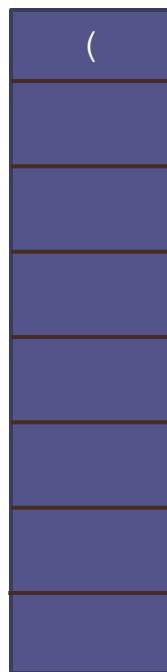
Expression:  $(w * [x + y] / z)$



# Balanced Parentheses (cont.)

21

Expression: (w \* [x + y] / z)

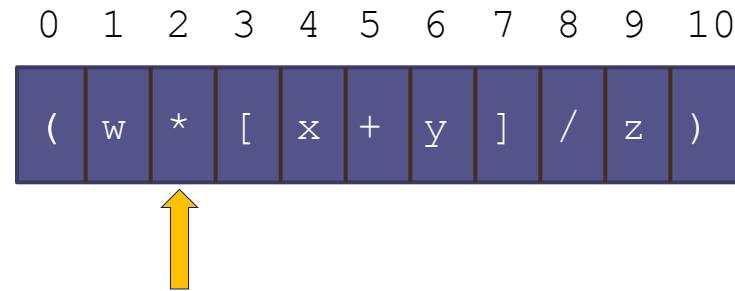
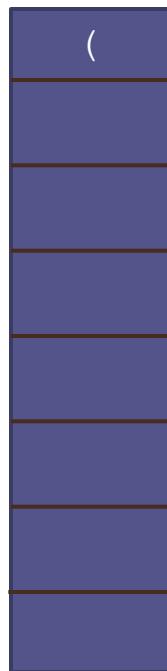


```
balanced : true  
index     : 1
```

# Balanced Parentheses (cont.)

22

Expression: (w \* [x + y] / z)

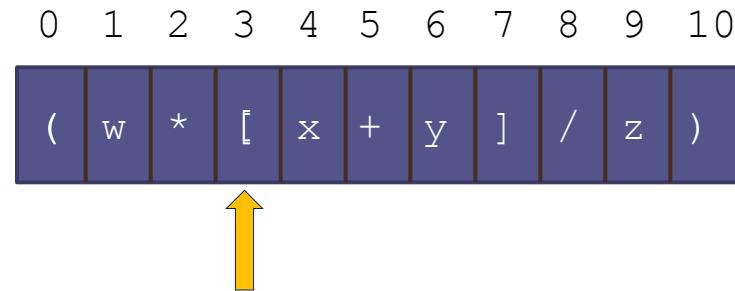
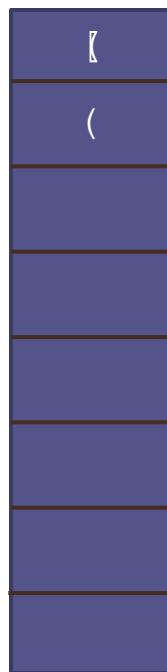


balanced : **true**  
index : 2

# Balanced Parentheses (cont.)

23

Expression: (w \* [x + y] / z)

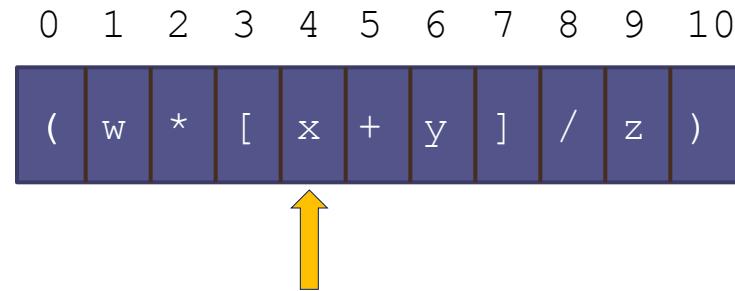
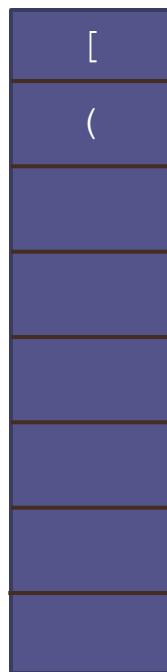


balanced : **true**  
index : 3

# Balanced Parentheses (cont.)

24

Expression: (w \* [x + y] / z)

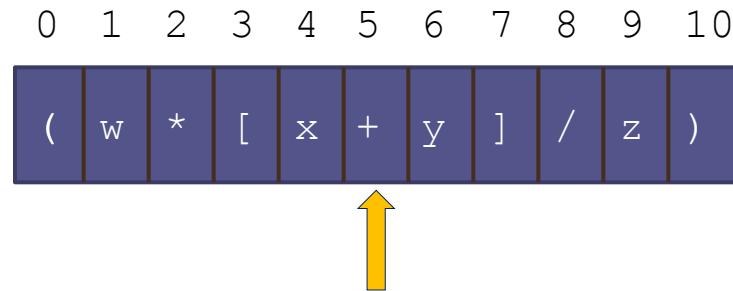
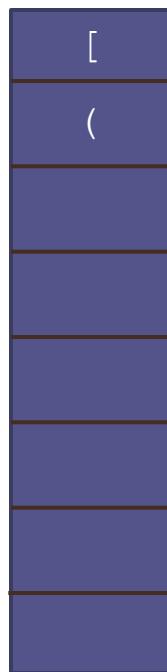


```
balanced : true
index     : 4
```

# Balanced Parentheses (cont.)

25

Expression: (w \* [x + y] / z)

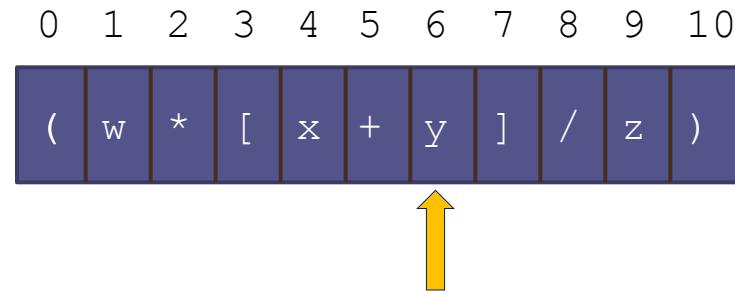
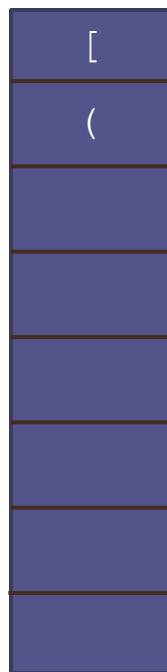


balanced : **true**  
index : 5

# Balanced Parentheses (cont.)

26

Expression: (w \* [x + y] / z)

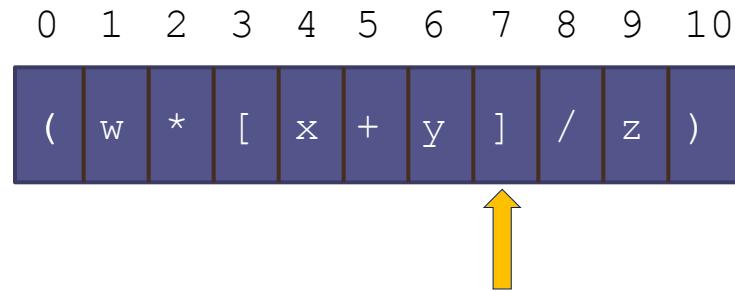
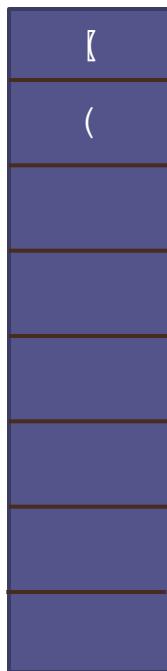


```
balanced : true  
index     : 6
```

# Balanced Parentheses (cont.)

27

Expression: (w \* [x + y] / z)



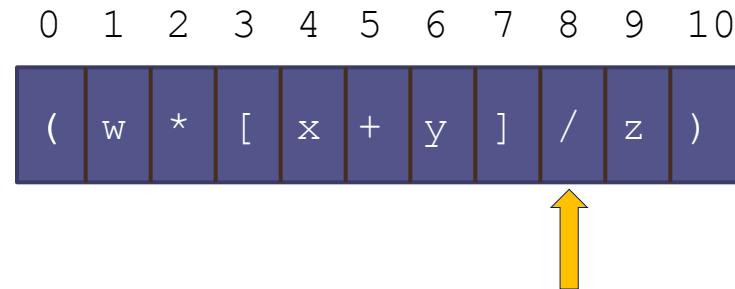
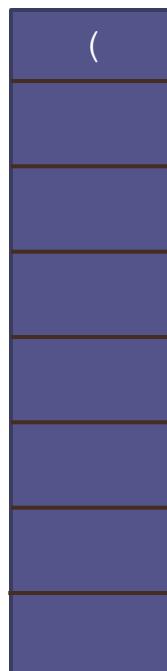
Matches!  
Balanced still true

```
balanced : true
index      : 7
```

# Balanced Parentheses (cont.)

28

Expression: (w \* [x + y] / z)

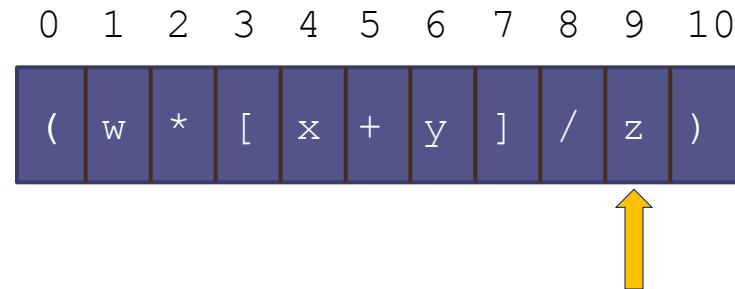
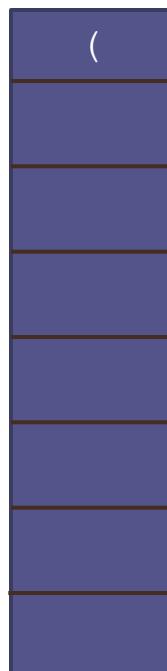


```
balanced : true  
index     : 8
```

# Balanced Parentheses (cont.)

29

Expression: (w \* [x + y] / z)

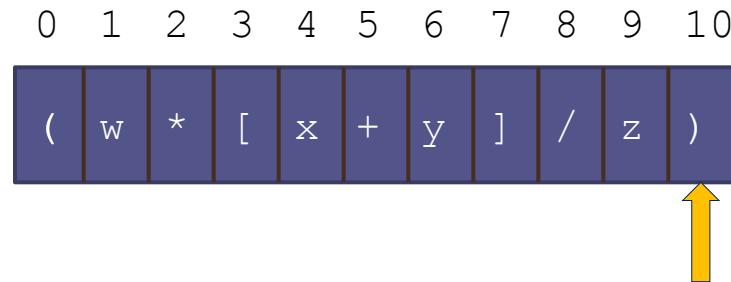
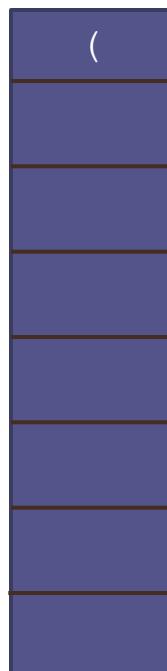


```
balanced : true  
index     : 9
```

# Balanced Parentheses (cont.)

30

Expression: (w \* [x + y] / z)



Matches!  
Balanced still true

balanced : **true**  
index : 10

# Testing

31

- Provide a variety of input expressions displaying the result true or false
- Try several levels of nested parentheses
- Try nested parentheses where corresponding parentheses are not of the same type
- Try unbalanced parentheses
  
- PITFALL: attempting to pop an empty stack will throw an EmptyStackException. You can guard against this by either testing for an empty stack or catching the exception

# Implementing a Stack

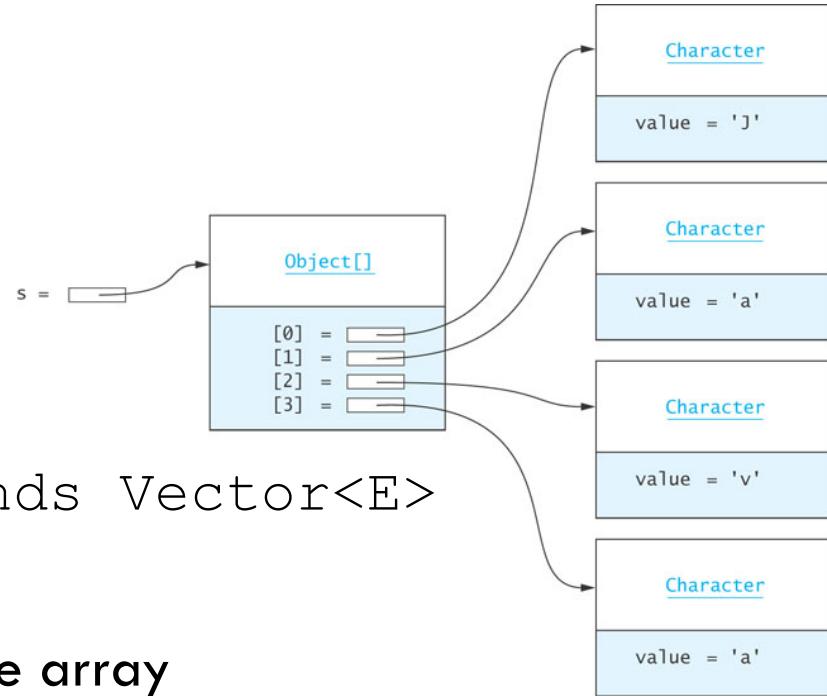
## Section 4.3

# Implementing a Stack as an Extension of Vector

33

- The Java API includes a Stack class as part of the package `java.util`:

```
public class Stack<E> extends Vector<E>
```



- The `Vector` class implements a growable array of objects
- Elements of a `Vector` can be accessed using an integer index and the size can grow or shrink as needed to accommodate the insertion and removal of elements

# Implementing a Stack as an Extension of Vector (cont.)

34

- We can use Vector's add method to implement push:

```
public E push(obj E) {  
    add(obj);  
    return obj;  
}
```

- pop can be coded as

```
public E pop() throws EmptyStackException {  
    try {  
        return remove (size() - 1);  
    } catch (ArrayIndexOutOfBoundsException ex) {  
        throw new EmptyStackException();  
    }  
}
```

# Implementing a Stack as an Extension of Vector (cont.)

35

- Because a Stack *is* a Vector, all of Vector operations can be applied to a Stack (such as searches and access by index)
- But, since only the top element of a stack should be accessible, this violates the principle of information hiding

# Implementing a Stack with a List Component

36

- As an alternative to a stack as an extension of `Vector`, we can write a class, `ListStack`, that has a `List` component (in the example below, `theData`)
- We can use either the `ArrayList`, `Vector`, or the `LinkedList` classes, as all implement the `List` interface. The `push` method, for example, can be coded as

```
public E push(E obj) {  
    theData.add(obj);  
    return obj;  
}
```

- A class which adapts methods of another class by giving different names to essentially the same methods (`push` instead of `add`) is called an *adapter class*
- Writing methods in this way is called *method delegation*

# Implementing a Stack Using an Array

37

- If we implement a stack as an array,  
we would need . . .

```
public class ArrayStack<E> implements StackInt<E> {  
    private E[] theData;  
    int topOfStack = -1;  
    private static final int INITIAL_CAPACITY = 10;  
  
    @SupressWarnings("unchecked")  
    public ArrayStack() {  
        theData = (E[]) new Object[INITIAL_CAPACITY];  
    }
```

# Implementing a Stack Using an Array

38

- If we implement a stack as an array we would need . . .

```
public class ArrayStack<E> implements Stack<E> {  
    private E[] theData;  
    int topOfStack = -1;  
    private static final int INITIAL_CAPACITY = 10;  
  
    @SuppressWarnings("unchecked")  
    public E[] getTheData() {  
        return theData;  
    }  
}
```

We do not need a size variable or  
method

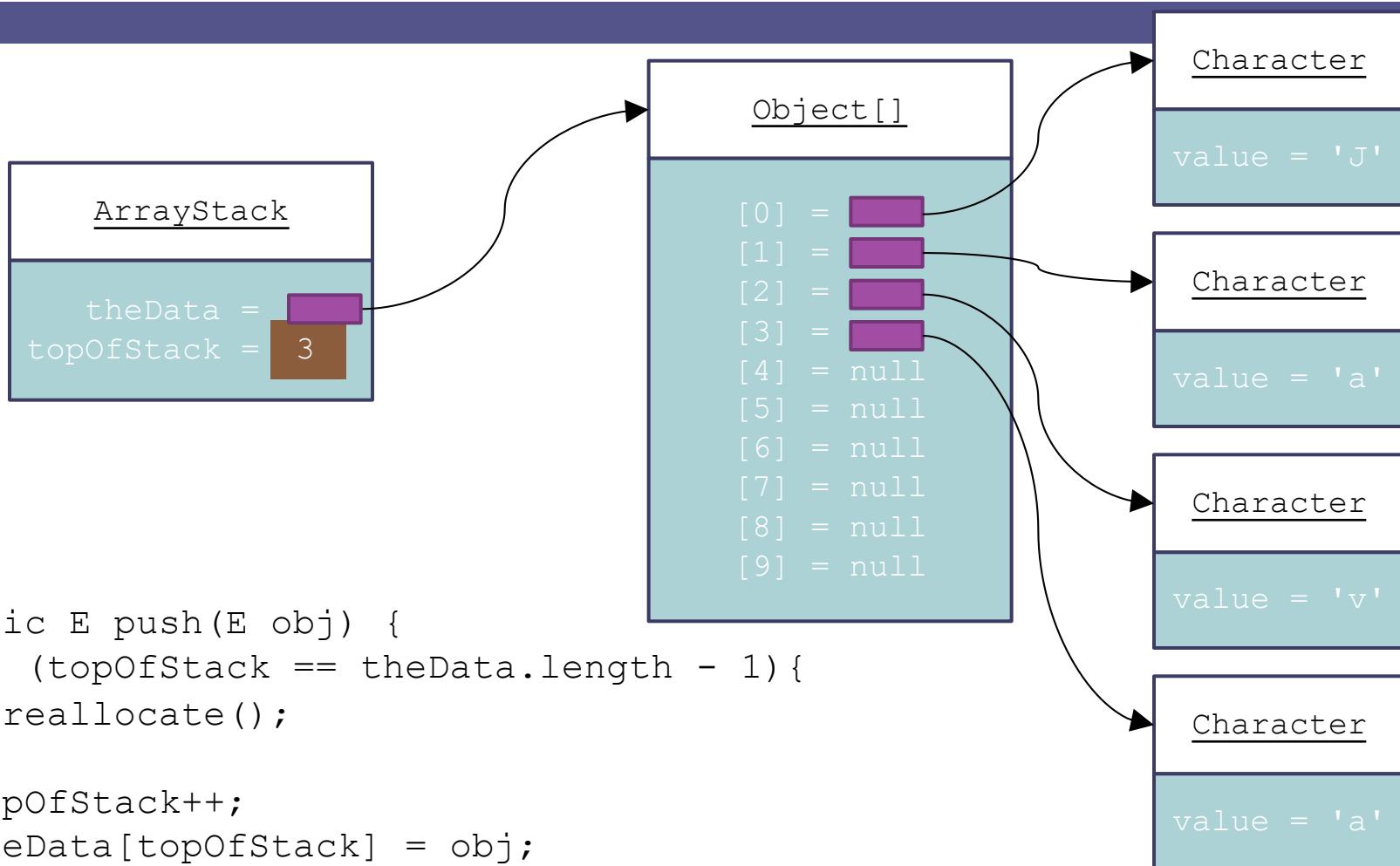
Allocate storage for an array with a default capacity

Keep track of the top of the stack (subscript of the element at the top of the stack; for empty stack = -1)

# Implementing a Stack Using an Array

## (cont.)

39



# Implementing a Stack Using an Array

## (cont.)

40

```
@Override  
public E pop() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    return theData[topOfStack--];  
}
```

# Implementing a Stack Using an Array

## (cont.)

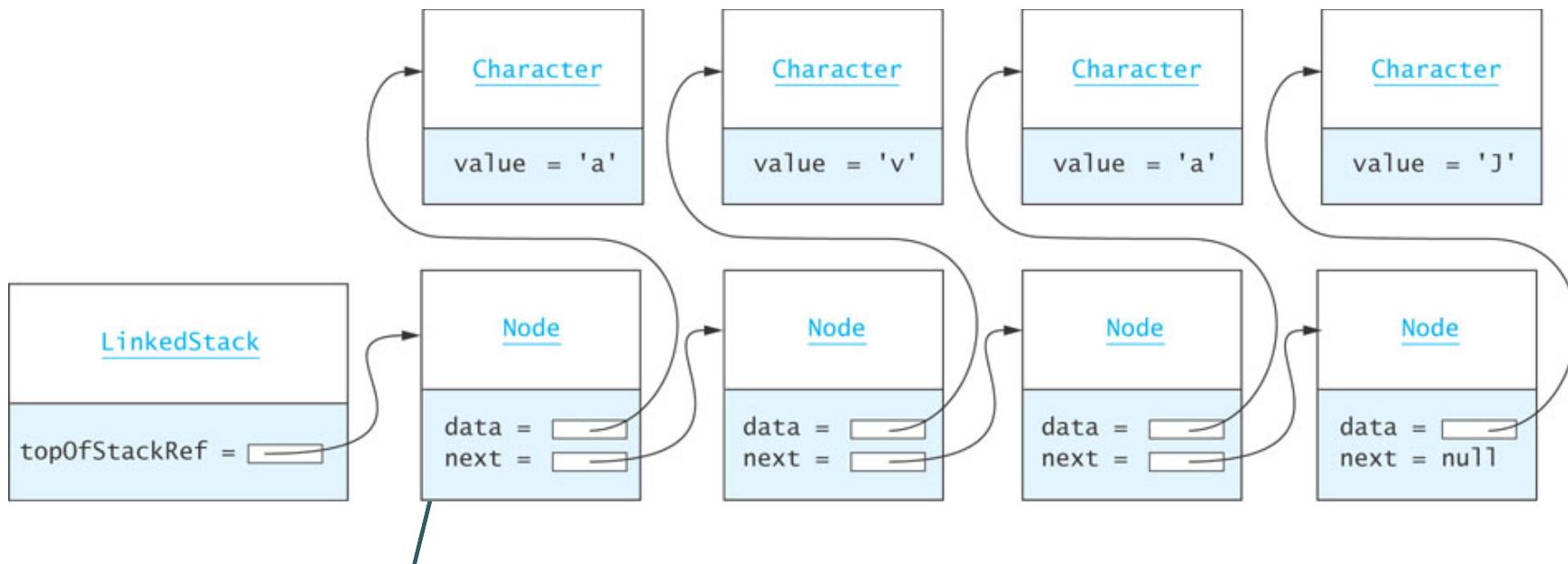
41

- This implementation is  $O(1)$ , in contrast to the Pez analogy and the “kayak” example, which are both  $O(n)$

# Implementing a Stack as a Linked Data Structure

42

- We can also implement a stack using a linked list of nodes



when the list is empty, pop  
returns null

# Implementing a Stack as a Linked Data Structure (cont.)

43

```
import java.util.EmptyStackException;

public class LinkedStack < E > implements StackInt < E > {

    /** A Node is the building block for a single-linked list. */
    private static class Node < E > {
        // Data Fields
        /** The reference to the data. */
        private E data;
        /** The reference to the next node. */
        private Node next;
    }
}
```

# Implementing a Stack as a Linked Data Structure (cont.)

44

```
// Constructors

/** Creates a new node with a null next field.

    @param dataItem The data stored
    */

private Node(E dataItem) {
    data = dataItem;
    next = null;
}

/** Creates a new node that references another node.

    @param dataItem The data stored
    @param nodeRef The node referenced by new node
    */

private Node(E dataItem, Node < E > nodeRef) {
    data = dataItem;
    next = nodeRef;
}

} //end class Node
```

# Implementing a Stack as a Linked Data Structure (cont.)

45

```
// Data Fields  
/** The reference to the first stack node. */  
private Node < E > topOfStackRef = null;  
  
/** Insert a new item on top of the stack.  
 post: The new item is the top item on the stack.  
         All other items are one position lower.  
 @param obj The item to be inserted  
 @return The item that was inserted  
 */  
public E push(E obj) {  
    topOfStackRef = new Node < E > (obj, topOfStackRef);  
    return obj;  
}
```

# Implementing a Stack as a Linked Data Structure (cont.)

46

```
/** Remove and return the top item on the stack.  
 * pre: The stack is not empty.  
 * post: The top item on the stack has been  
 *        removed and the stack is one item smaller.  
 * @return The top item on the stack  
 * @throws EmptyStackException if the stack is empty  
 */  
  
public E pop() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    else {  
        E result = topOfStackRef.data;  
        topOfStackRef = topOfStackRef.next;  
        return result;  
    }  
}
```

# Implementing a Stack as a Linked Data Structure (cont.)

47

```
/** Return the top item on the stack.  
 * pre: The stack is not empty.  
 * post: The stack remains unchanged.  
 * @return The top item on the stack  
 * @throws EmptyStackException if the stack is empty  
 */  
  
public E peek() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    else {  
        return topOfStackRef.data;  
    }  
}
```

# Implementing a Stack as a Linked Data Structure (cont.)

48

```
/** See whether the stack is empty.  
 @return true if the stack is empty  
 */  
  
public boolean empty() {  
    return topOfStackRef == null;  
}  
}
```

# Comparison of Stack Implementations

49

- Extending a Vector (as is done by Java) is a poor choice for stack implementation, since all Vector methods are accessible
- The easiest implementation uses a List component (`ArrayList` is the simplest) for storing data
  - ▣ An underlying array requires reallocation of space when the array becomes full, and
  - ▣ an underlying linked data structure requires allocating storage for links
  - ▣ As all insertions and deletions occur at one end, they are constant time,  $O(1)$ , regardless of the type of implementation used

# Additional Stack Applications

## Section 4.4

# Additional Stack Applications

51

- Postfix and infix notation
  - Expressions normally are written in infix form, but
  - it easier to evaluate an expression in postfix form since there is no need to group sub-expressions in parentheses or worry about operator precedence

Postfix Expression	Infix Expression	Value
<u>4</u> <u>7</u> <u>*</u>	$4 * 7$	28
<u>4</u> <u>7</u> <u>2</u> <u>+</u> <u>*</u>	$4 * (7 + 2)$	36
<u>4</u> <u>7</u> <u>*</u> <u>20</u> <u>-</u>	$(4 * 7) - 20$	8
<u>3</u> <u>4</u> <u>7</u> <u>*</u> <u>2</u> <u>/</u> <u>+</u>	$3 + ((4 * 7) / 2)$	17

# Evaluating Postfix Expressions

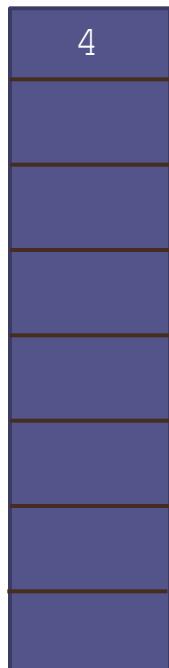
52

- Write a class that evaluates a postfix expression
- Use the space character as a delimiter between tokens

Data Field	Attribute
Stack<Integer> operandStack	The stack of operands (Integer objects).
Method	Behavior
public int eval(String expression)	Returns the value of expression.
private int evalOp(char op)	Pops two operands and applies operator op to its operands, returning the result.
private boolean isOperator(char ch)	Returns <b>true</b> if ch is an operator symbol.

# Evaluating Postfix Expressions (cont.)

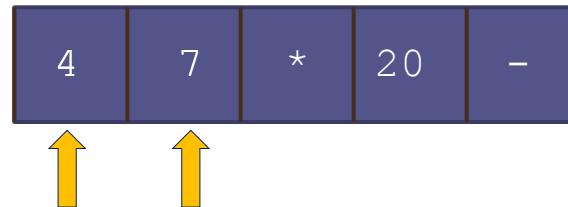
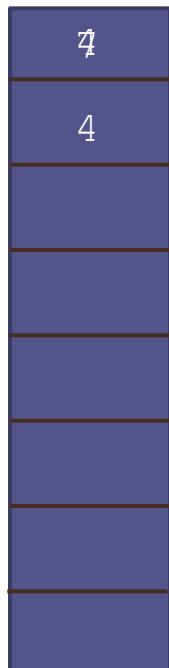
53



- ➡ 1. create an empty stack of integers
- ➡ 2. while there are more tokens
- ➡ 3. get the next token
- ➡ 4. if the first character of the token is a digit
- ➡ 5.     push the character on the stack
- 6. else if the token is an operator
- 7.     pop the right operand off the stack
- 8.     pop the left operand off the stack
- 9.     evaluate the operation
- 10.    push the result onto the stack
- 11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

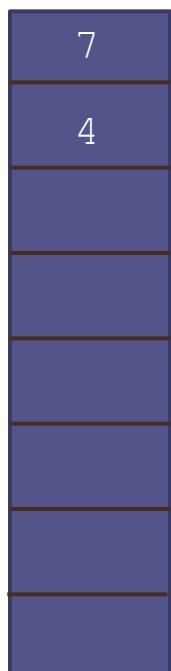
54



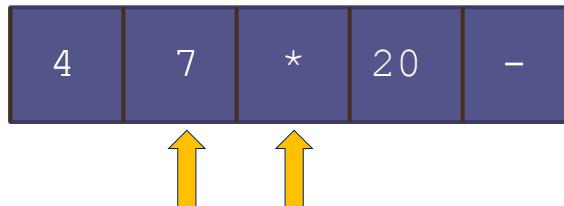
1. create an empty stack of integers
2. while there are more tokens
  3. get the next token
  4. if the first character of the token is a digit
  5. push the character on the stack
  6. else if the token is an operator
  7. pop the right operand off the stack
  8. pop the left operand off the stack
  9. evaluate the operation
  10. push the result onto the stack
  11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

55



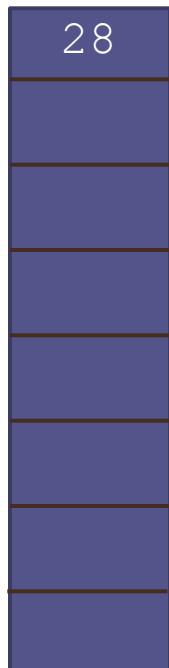
4 \* 7



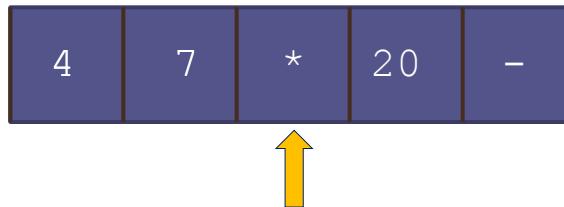
1. create an empty stack of integers
2. while there are more tokens
  - 3. get the next token
  - 4. if the first character of the token is a digit
    - 5. push the character on the stack
  - 6. else if the token is an operator
    - 7. pop the right operand off the stack
    - 8. pop the left operand off the stack
    - 9. evaluate the operation
    - 10. push the result onto the stack
  - 11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

56



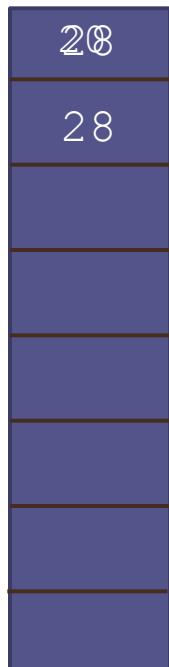
28



1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5.     push the character on the stack
6. else if the token is an operator
7.     pop the right operand off the stack
8.     pop the left operand off the stack
9.     evaluate the operation  
    →
10.    push the result onto the stack
11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

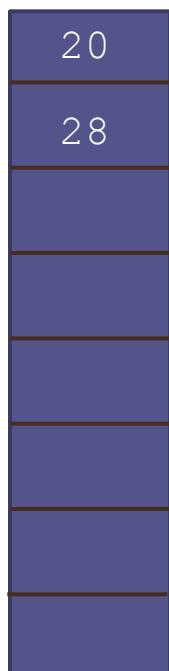
57



1. create an empty stack of integers
2. while there are more tokens
  3. get the next token
  4. if the first character of the token is a digit
  5. push the character on the stack
  6. else if the token is an operator
  7. pop the right operand off the stack
  8. pop the left operand off the stack
  9. evaluate the operation
  10. push the result onto the stack
  11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

58



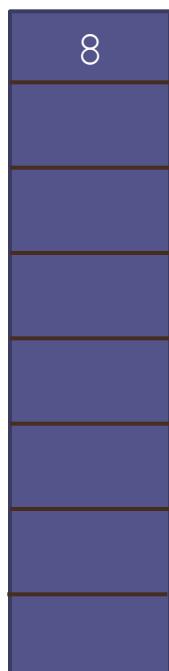
28 - 20



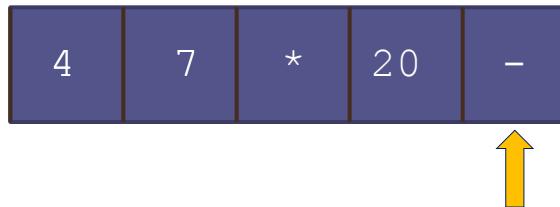
1. create an empty stack of integers
2. while there are more tokens
  3. get the next token
  4. if the first character of the token is a digit
    5. push the character on the stack
  6. else if the token is an operator
    7. pop the right operand off the stack
    8. pop the left operand off the stack
    9. evaluate the operation
    10. push the result onto the stack
  11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

59



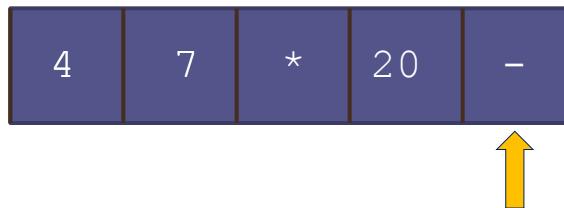
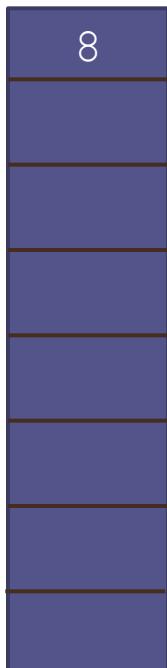
8



1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5.     push the character on the stack
6. else if the token is an operator
7.     pop the right operand off the stack
8.     pop the left operand off the stack
9.     evaluate the operation  
    →
10.    push the result onto the stack
11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

60



1. create an empty stack of integers
2. while there are more tokens
  3. get the next token
  4. if the first character of the token is a digit
  5. push the number on the stack
  6. else if the token is an operator
  7. pop the right operand off the stack
  8. pop the left operand off the stack
  9. evaluate the operation
  10. push the result onto the stack
11. pop the stack and return the result

# Evaluating Postfix Expressions (cont.)

61

- Testing: write a driver which
  - creates a PostfixEvaluator object
  - reads one or more expressions and reports the result
  - catches PostfixEvaluator.SyntaxErrorException
  - exercises each path by using each operator
  - exercises each path through the method by trying different orderings and multiple occurrences of operators
  - tests for syntax errors:
    - an operator without any operands
    - a single operand
    - an extra operand
    - an extra operator
    - a variable name
    - the empty string

# Converting from Infix to Postfix

62

- Convert infix expressions to postfix expressions
- Assume:
  - ▣ expressions consists of only spaces, operands, and operators
  - ▣ space is a delimiter character
  - ▣ all operands that are identifiers begin with a letter or underscore
  - ▣ all operands that are numbers begin with a digit

Data Field	Attribute
private Stack<Character> operatorStack	Stack of operators.
private StringBuilder postfix	The postfix string being formed.
Method	Behavior
public String convert(String infix)	Extracts and processes each token in <code>infix</code> and returns the equivalent postfix string.
private void processOperator(char op)	Processes operator <code>op</code> by updating <code>operatorStack</code> .
private int precedence(char op)	Returns the precedence of operator <code>op</code> .
private boolean isOperator(char ch)	Returns <code>true</code> if <code>ch</code> is an operator symbol.

# Converting from Infix to Postfix

## (cont.)

63

- Example: convert

$w - 5.1 / \text{sum} * 2$

to its postfix form

$w\ 5.1\ \text{sum}\ / \ 2\ * \ -$

# Converting from Infix to Postfix

## (cont.)

64

Next Token	Action	Effect on operatorStack	Effect on postfix
w	Append w to postfix.		w
-	The stack is empty Push - onto the stack	-	w
5.1	Append 5.1 to postfix	-	w 5.1
/	precedence(/) > precedence(-), Push / onto the stack	/ -	w 5.1
sum	Append sum to postfix	/ -	w 5.1 sum
*	precedence(*) equals precedence(/) Pop / off of stack and append to postfix	-	w 5.1 sum /

# Converting from Infix to Postfix

## (cont.)

65

Next Token	Action	Effect on operatorStack	Effect on postfix
*	precedence(*) > precedence(-), Push * onto the stack	* -	w 5.1 sum /
2	Append 2 to postfix	* -	w 5.1 sum / 2
End of input	Stack is not empty, Pop * off the stack and append to postfix	-	w 5.1 sum / 2 *
End of input	Stack is not empty, Pop - off the stack and append to postfix		w 5.1 sum / 2 * -

# Converting from Infix to Postfix

## (cont.)

66

### Algorithm for Method convert

1. Initialize postfix to an empty `StringBuilder`.
2. Initialize the operator stack to an empty stack.
3. **while** there are more tokens in the infix string
4.     Get the next token.
5.     **if** the next token is an operand
6.         Append it to postfix.
7.     **else if** the next token is an operator
8.         Call `processOperator` to process the operator.
9.     **else**
10.         Indicate a syntax error.
11.     Pop remaining operators off the operator stack and append them to postfix.

# Converting from Infix to Postfix

## (cont.)

67

### Algorithm for Method `processOperator`

1. **if** the operator stack is empty
2.     Push the current operator onto the stack.
3. **else**
4.     Peek the operator stack and let `topOp` be the top operator.
5.     **if** the precedence of the current operator is greater than the precedence of `topOp`
6.         Push the current operator onto the stack.
7.     **else**
8.         **while** the stack is not empty and the precedence of the current operator is less than or equal to the precedence of `topOp`
9.             Pop `topOp` off the stack and append it to `postfix`.
10.          **if** the operator stack is not empty
11.             Peek the operator stack and let `topOp` be the top operator.
12.     Push the current operator onto the stack.



# CS 570: Data Structures

## Queues

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 4 (PART 2)

Queues

# Chapter Objectives

3

- To learn how to use the methods in the Queue interface for insertion (`offer` and `add`), removal (`remove` and `poll`), and for accessing the element at the front (`peek` and `element`)
- To understand how to implement the Queue interface using a single-linked list, a circular array, and a double-linked list
- To become familiar with the Deque interface and how to use its methods to insert and remove items from either end of a deque

# Week 7

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 4.5-4.8

# Queue

5

- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way
  - ▣ A stack is LIFO list – *Last-In, First-Out*
  - ▣ while a queue is FIFO list, *First-In, First-Out*

# Example: Print Queue

6

- Operating systems use queues to
  - keep track of tasks waiting for a scarce resource
  - ensure that the tasks are carried out in the order they were generated
- Print queue: printing is much slower than the process of selecting pages to print, so a queue is used

Document Name	Status	Owner	Pages	Size	Submitted	P
Microsoft Word - Queues_Paul_1007.doc	Paul Wolfgang	52	9.75 MB	1:53:18 PM	10/7/2003	
Microsoft Word - Stacks.doc	Paul Wolfgang	46	9.05 MB	1:53:57 PM	10/7/2003	
Microsoft Word - Trees2.doc	Paul Wolfgang	54	38.4 MB	1:54:41 PM	10/7/2003	

3 document(s) in queue

# Unsuitability of a Print Stack

7

- Stacks are Last-In, First-Out (LIFO)
- The most recently selected document would be the next to print
- Unless the printer stack is empty, your print job may never be executed if others are issuing print jobs

# Specification for a Queue Interface

8

Method	Behavior
boolean offer(E item)	Inserts <code>item</code> at the rear of the queue. Returns <code>true</code> if successful; returns <code>false</code> if the item could not be inserted.
E remove()	Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a <code>NoSuchElementException</code> .
E poll()	Removes the entry at the front of the queue and returns it; returns <code>null</code> if the queue is empty.
E peek()	Returns the entry at the front of the queue without removing it; returns <code>null</code> if the queue is empty.
E element()	Returns the entry at the front of the queue without removing it. If the queue is empty, throws a <code>NoSuchElementException</code> .

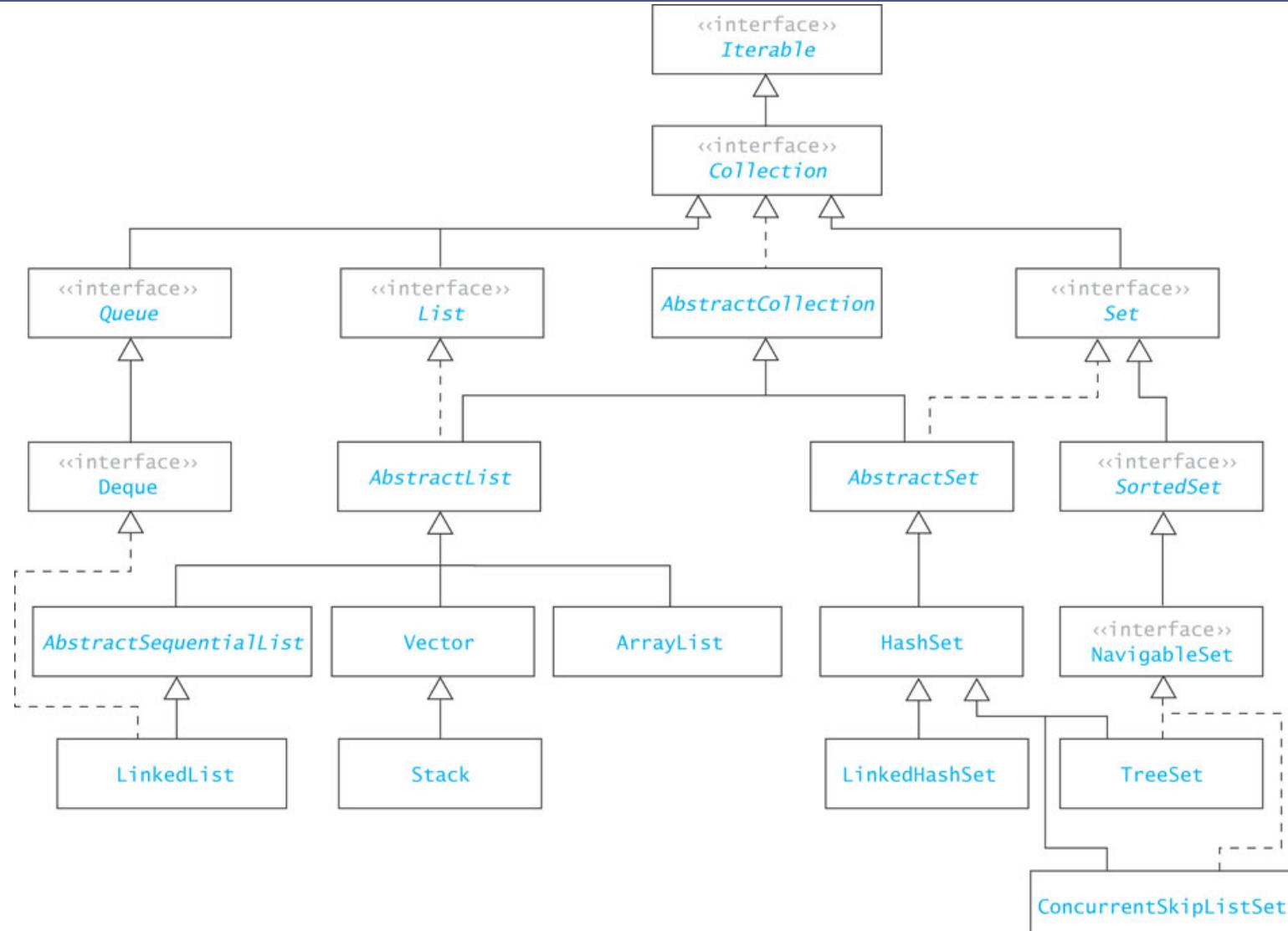
# Class LinkedList Implements the Queue Interface

9

- The **LinkedList** class provides methods for inserting and removing elements at either end of a double-linked list, which means all Queue methods can be implemented easily
- The Java **LinkedList** class implements the Queue interface  
`Queue<String> names = new LinkedList<String>();`
  - ▣ creates a new Queue reference, names, that stores references to String objects

# The Collection Framework

10



# Implementing the Queue Interface

## Section 4.7

# Using a Double-Linked List to Implement the Queue Interface

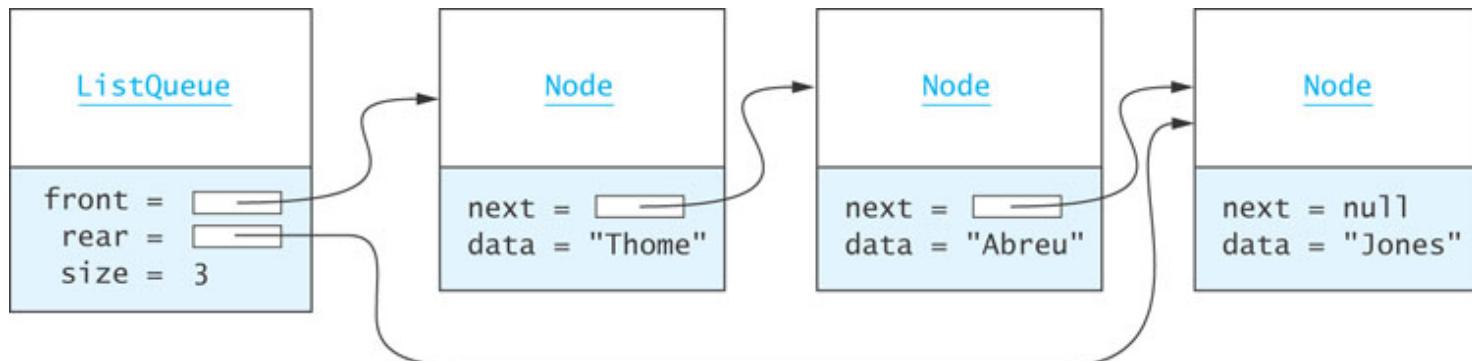
12

- Insertion and removal from either end of a double-linked list is  $O(1)$  so either end can be the front (or rear) of the queue
- Java designers decided to make the head of the linked list the front of the queue and the tail the rear of the queue
- Problem: If a `LinkedList` object is used as a queue, it will be possible to apply other `LinkedList` methods in addition to the ones required and permitted by the `Queue` interface
- Solution: Create a new class with a `LinkedList` component and then code (by delegation to the `LinkedList` class) only the public methods required by the `Queue` interface

# Using a Single-Linked List to Implement a Queue

13

- Insertions are at the rear of a queue and removals are from the front
- We need a reference to the last list node so that insertions can be performed at O(1)
- The number of elements in the queue is changed by methods `insert` and `remove`



# Using a Single-Linked List to Implement a Queue

14

```
import java.util.*;  
/** Implements the Queue interface using a single-linked  
list.  
 * @author Koffman & Wolfgang * */  
public class ListQueue < E >  
    extends AbstractQueue < E >  
    implements Queue < E > {  
  
    // Data Fields  
    /** Reference to front of queue. */  
    private Node < E > front;  
  
    /** Reference to rear of queue. */  
    private Node < E > rear;
```

# Using a Single-Linked List to Implement a Queue

15

```
/** Size of queue. */
private int size;

/** A Node is the building block for a single-linked list.
 */
private static class Node < E > {
    // Data Fields
    /** The reference to the data. */
    private E data;

    /** The reference to the next node. */
    private Node next;
```

# Using a Single-Linked List to Implement a Queue

16

```
// Constructors

/** Creates a new node with a null next field.
 * @param dataItem The data stored
 */
private Node(E dataItem) {
    data = dataItem;
    next = null;
}

/** Creates a new node that references another node.
 * @param dataItem The data stored
 * @param nodeRef The node referenced by new node
 */
private Node(E dataItem, Node < E > nodeRef) {
    data = dataItem;
    next = nodeRef;
}
} //end class Node
```

# Using a Single-Linked List to Implement a Queue

17

```
// Methods

/** Insert an item at the rear of the queue.
 * post: item is added to the rear of the queue.
 * @param item The element to add
 * @return true (always successful) */
public boolean offer(E item) {
    // Check for empty queue.
    if (front == null) {
        rear = new Node < E > (item);
        front = rear;
    }
    else {
        ...
    }
}
```

# Using a Single-Linked List to Implement a Queue

18

```
else {  
    // Allocate a new node at end, store item in  
    // it, and  
    // link it to old end of queue.  
  
    ??????  
}  
size++;  
return true;  
}
```

# Using a Single-Linked List to Implement a Queue

19

```
/** Remove the entry at the front of the queue and return it
 * if the queue is not empty.
 *
 * post: front references item that was second in the queue.
 *
 * @return The item removed if successful, or null if not
 */
public E poll() {
    E item = peek(); // Retrieve item at front.
    if (item == null)
        return null;
    // Remove item at front.
    front = front.next;
    size--;
    return item; // Return data at front of queue.
}
```

# Using a Single-Linked List to Implement a Queue

20

```
/** Return the item at the front of the queue
 * without removing it.
 *
 * @return The item at the front of the queue if
 * successful;
 *
 *         return null if the queue is empty
 */
public E peek() {
    if (size == 0)
        return null;
    else
        return front.data;
}
```

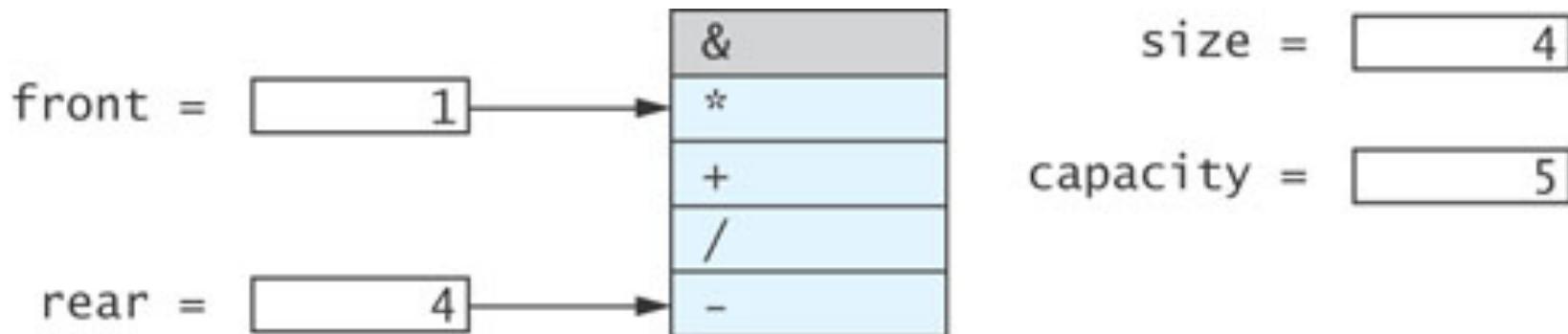
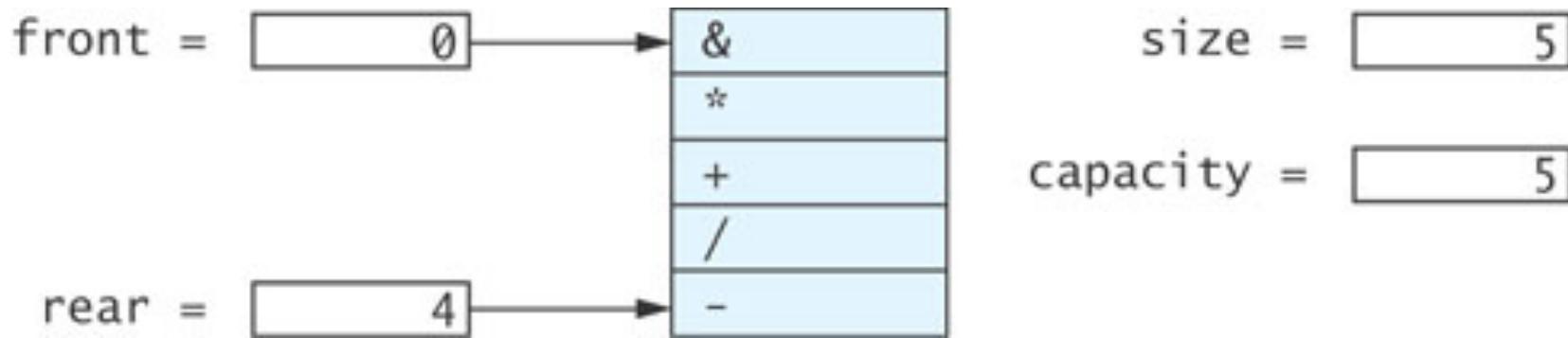
# Implementing a Queue Using a Circular Array

21

- The time efficiency of using a single- or double-linked list to implement a queue is acceptable
- However, there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored in the nodes
- Array Implementation
  - ▣ Insertion at rear of array is constant time  $O(1)$
  - ▣ Removal from the front is linear time  $O(n)$
  - ▣ Removal from rear of array is constant time  $O(1)$
  - ▣ Insertion at the front is linear time  $O(n)$
- We can avoid these inefficiencies in an array

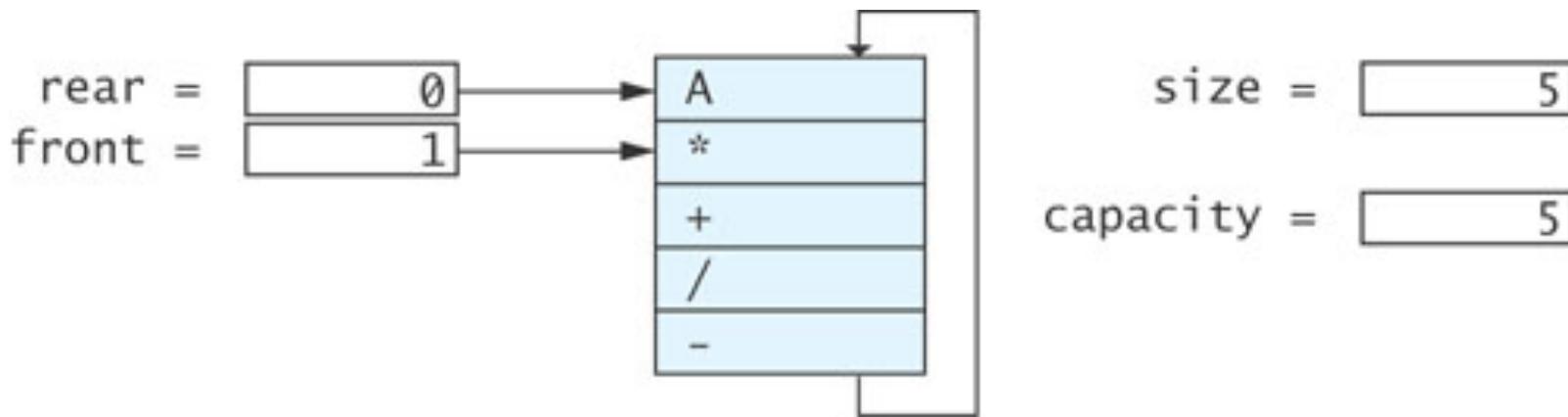
# Implementing a Queue Using a Circular Array (cont.)

22



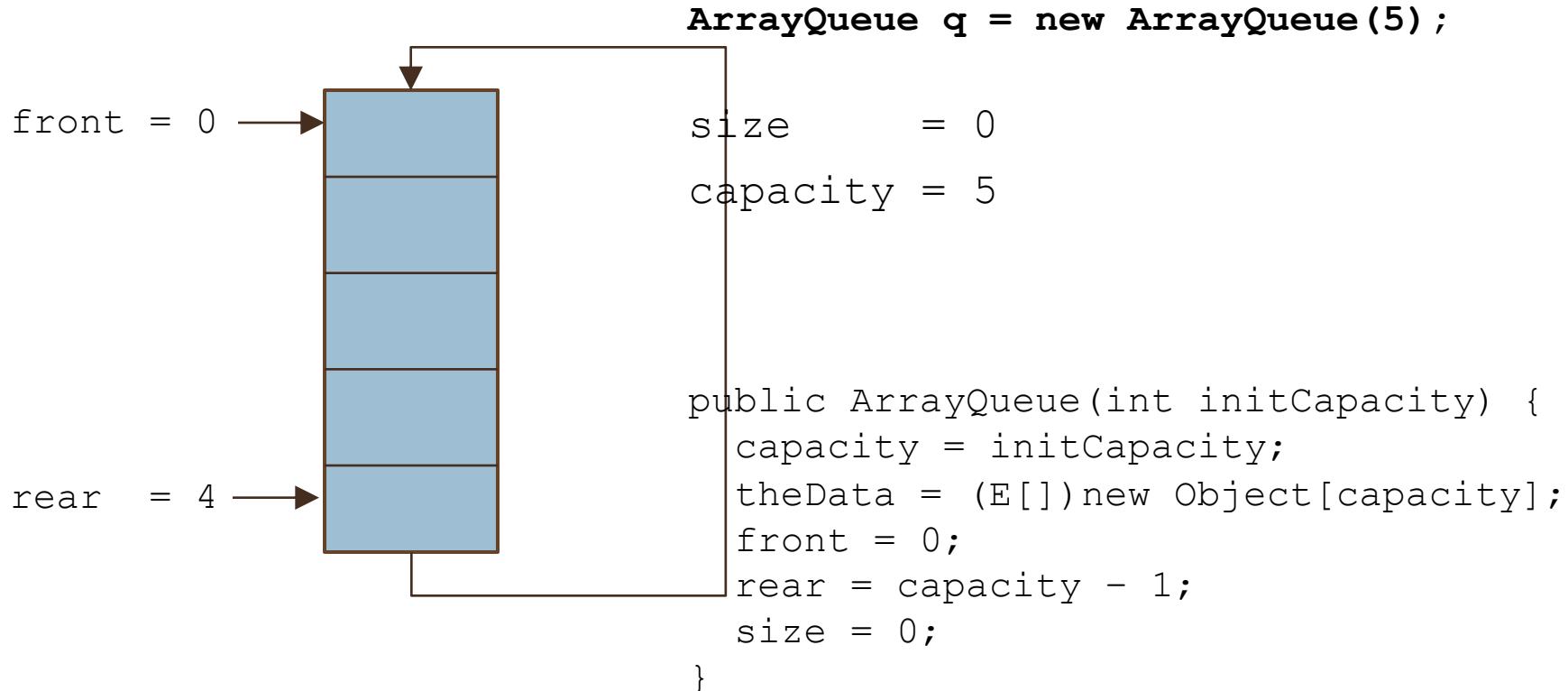
# Implementing a Queue Using a Circular Array (cont.)

23



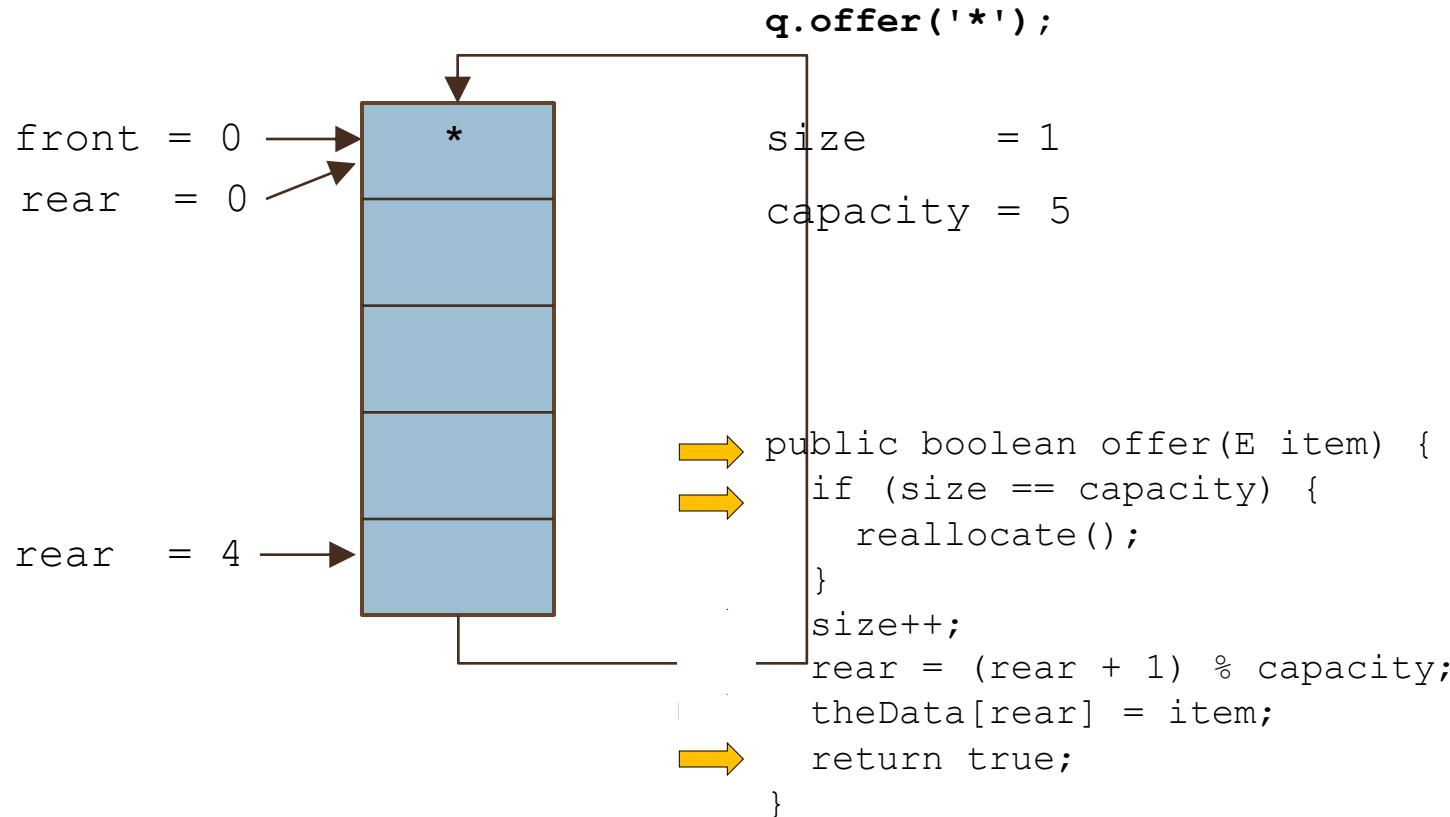
# Implementing a Queue Using a Circular Array (cont.)

24



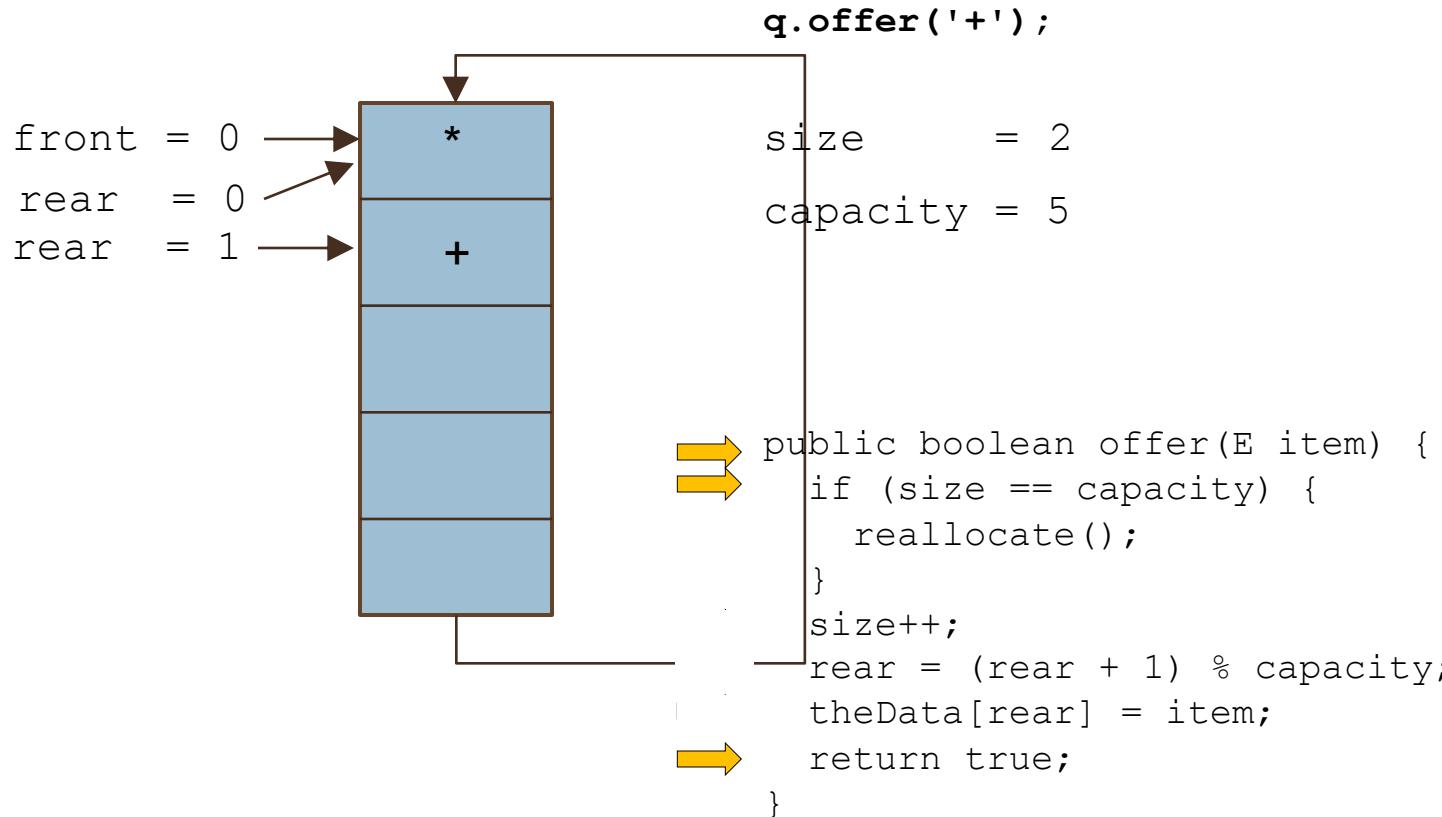
# Implementing a Queue Using a Circular Array (cont.)

25



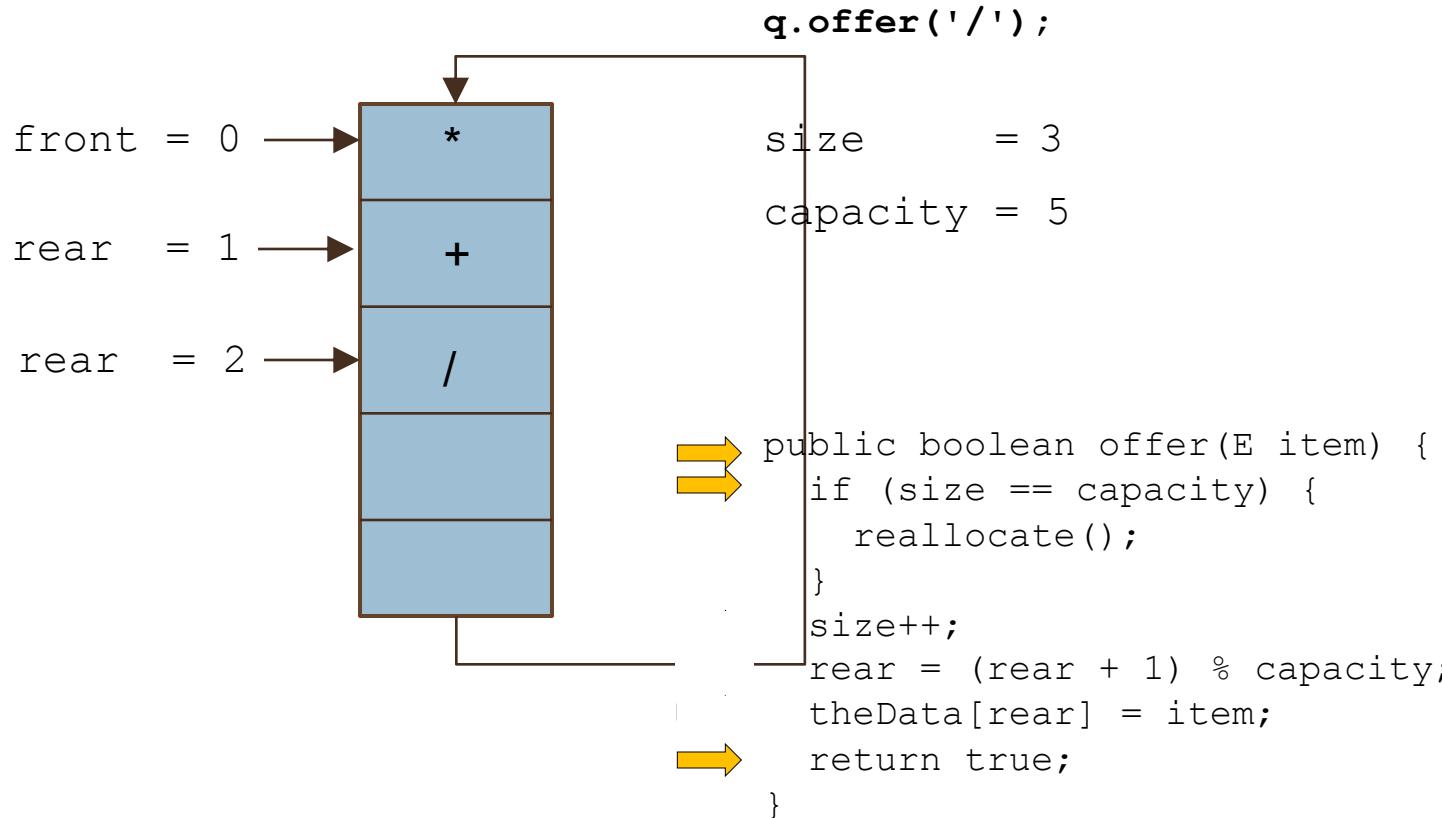
# Implementing a Queue Using a Circular Array (cont.)

26



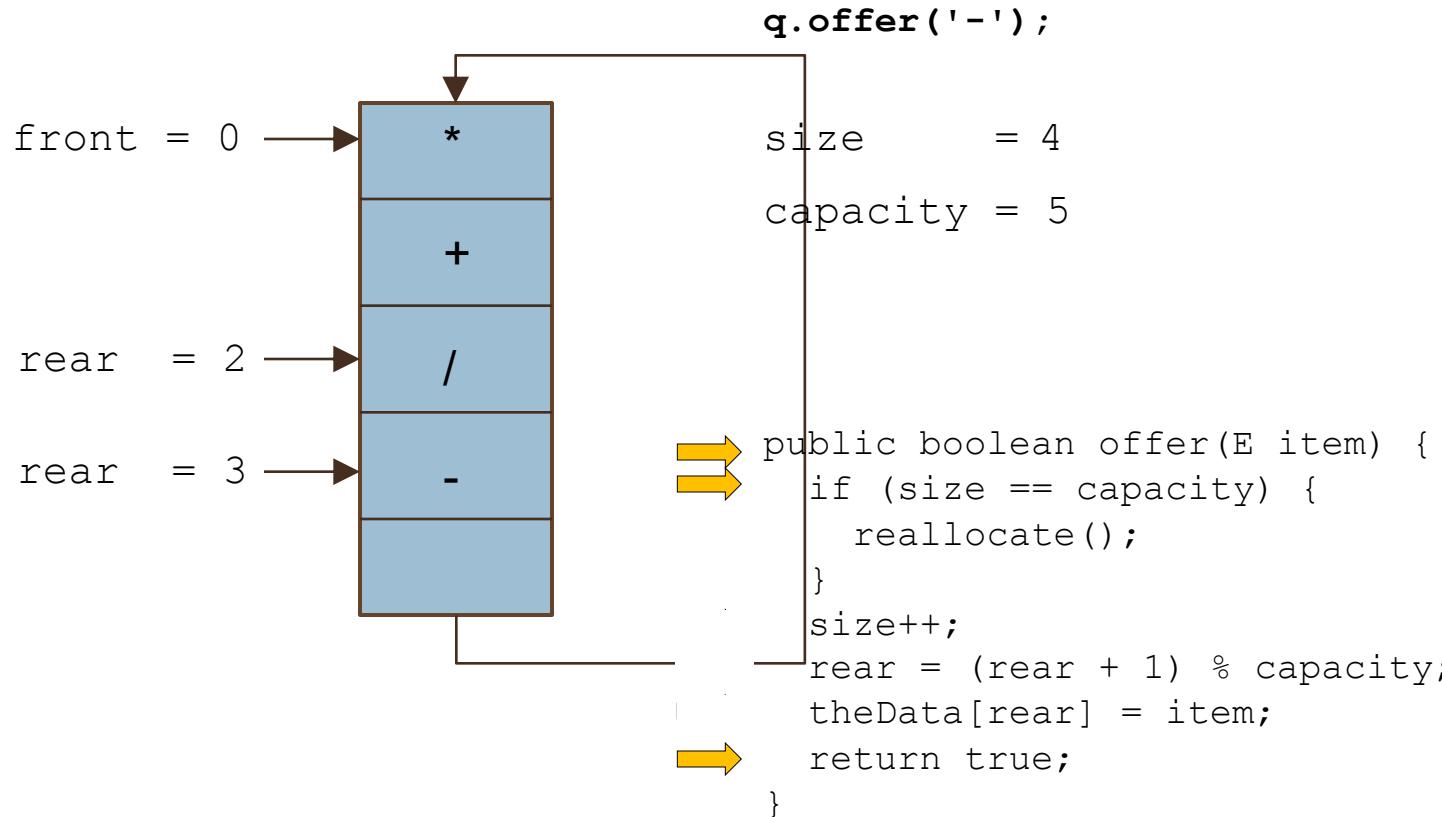
# Implementing a Queue Using a Circular Array (cont.)

27



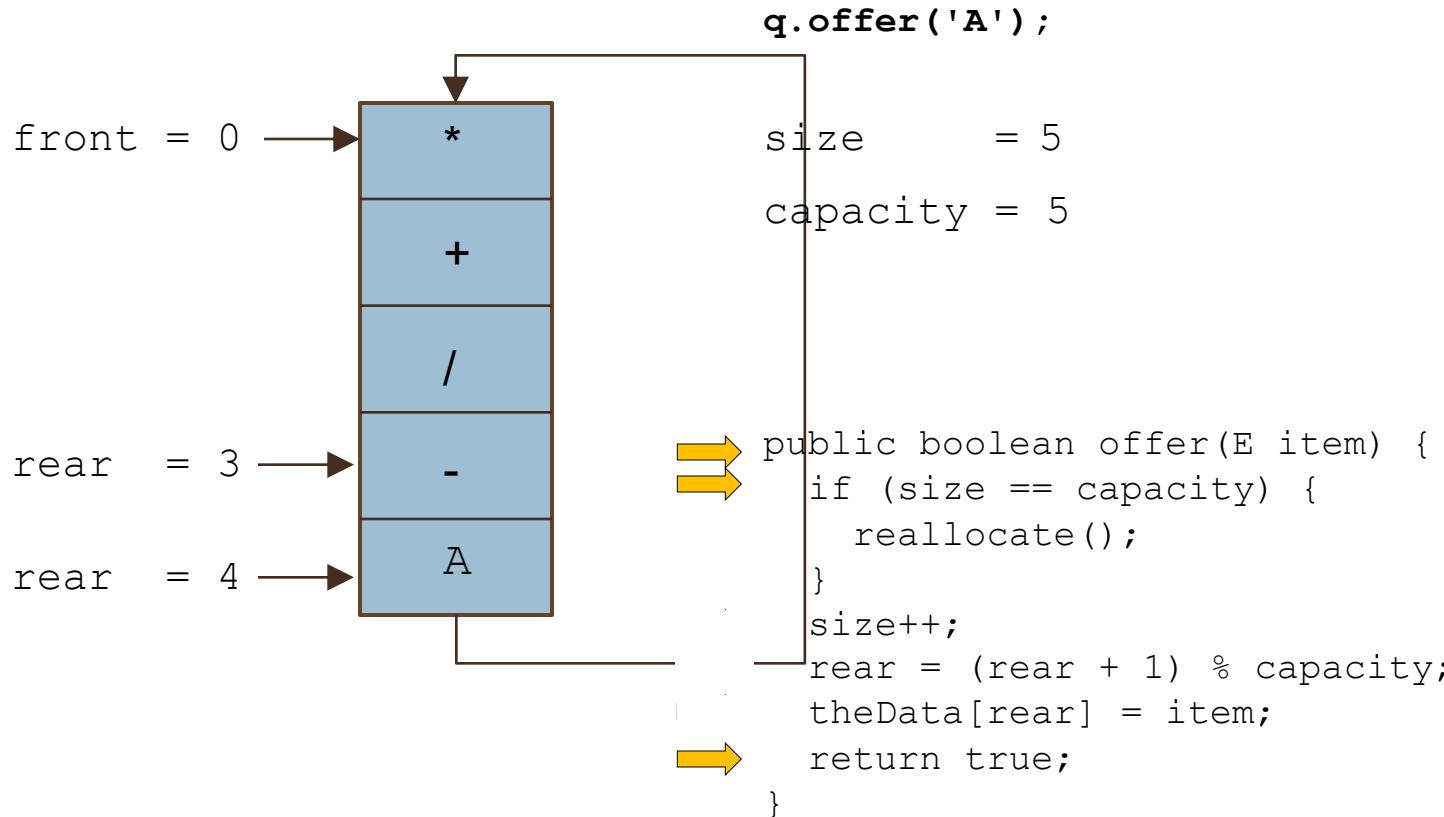
# Implementing a Queue Using a Circular Array (cont.)

28



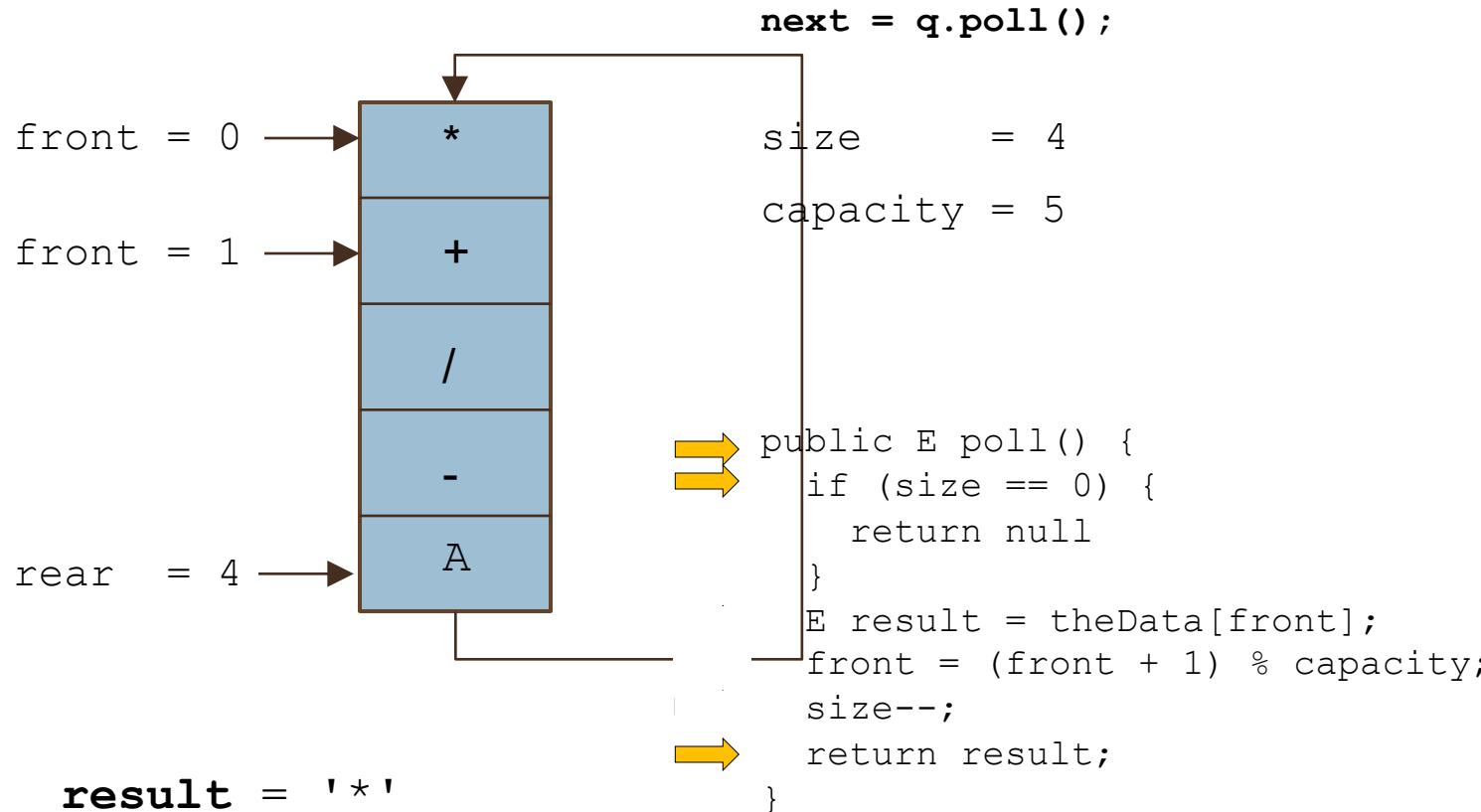
# Implementing a Queue Using a Circular Array (cont.)

29



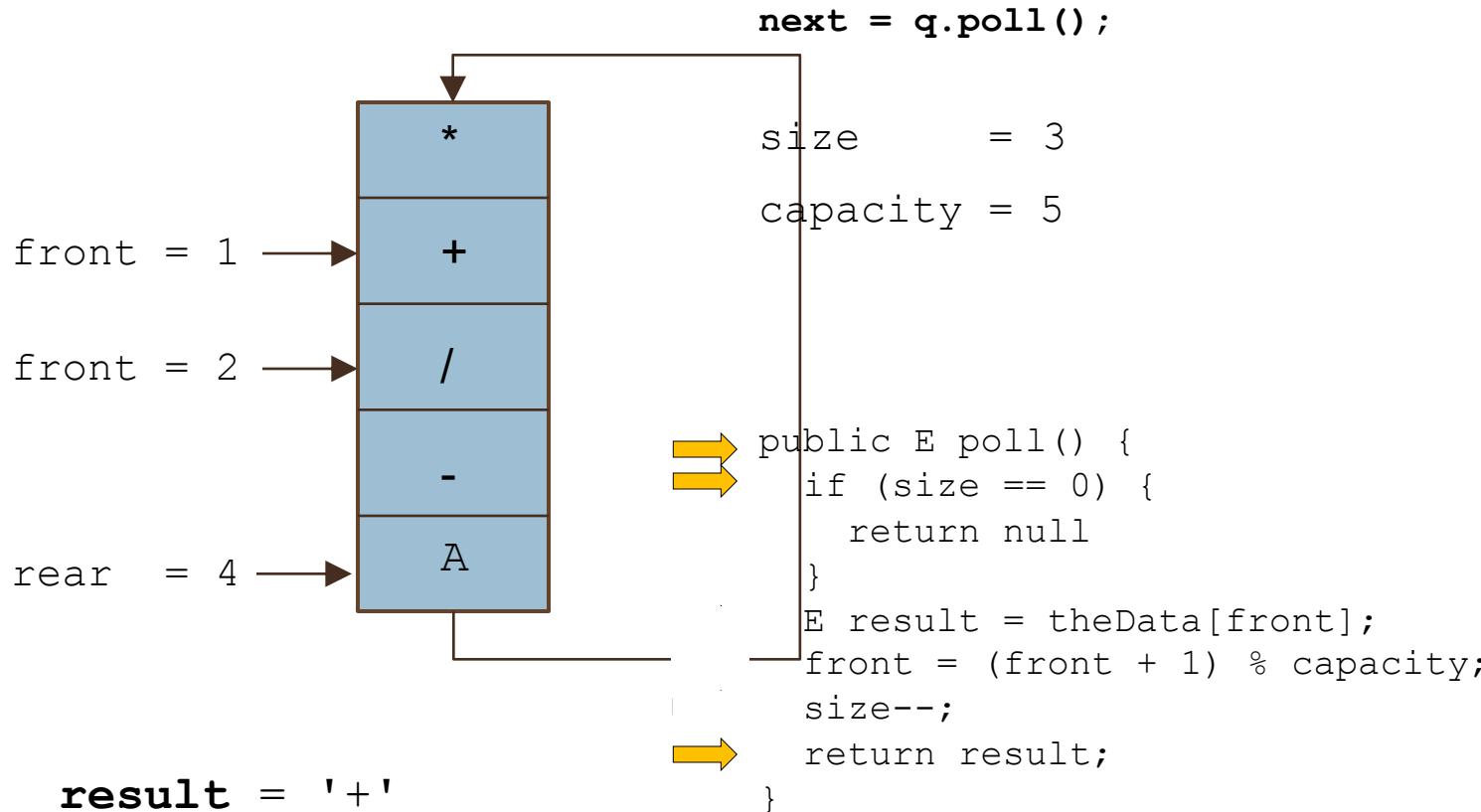
# Implementing a Queue Using a Circular Array (cont.)

30



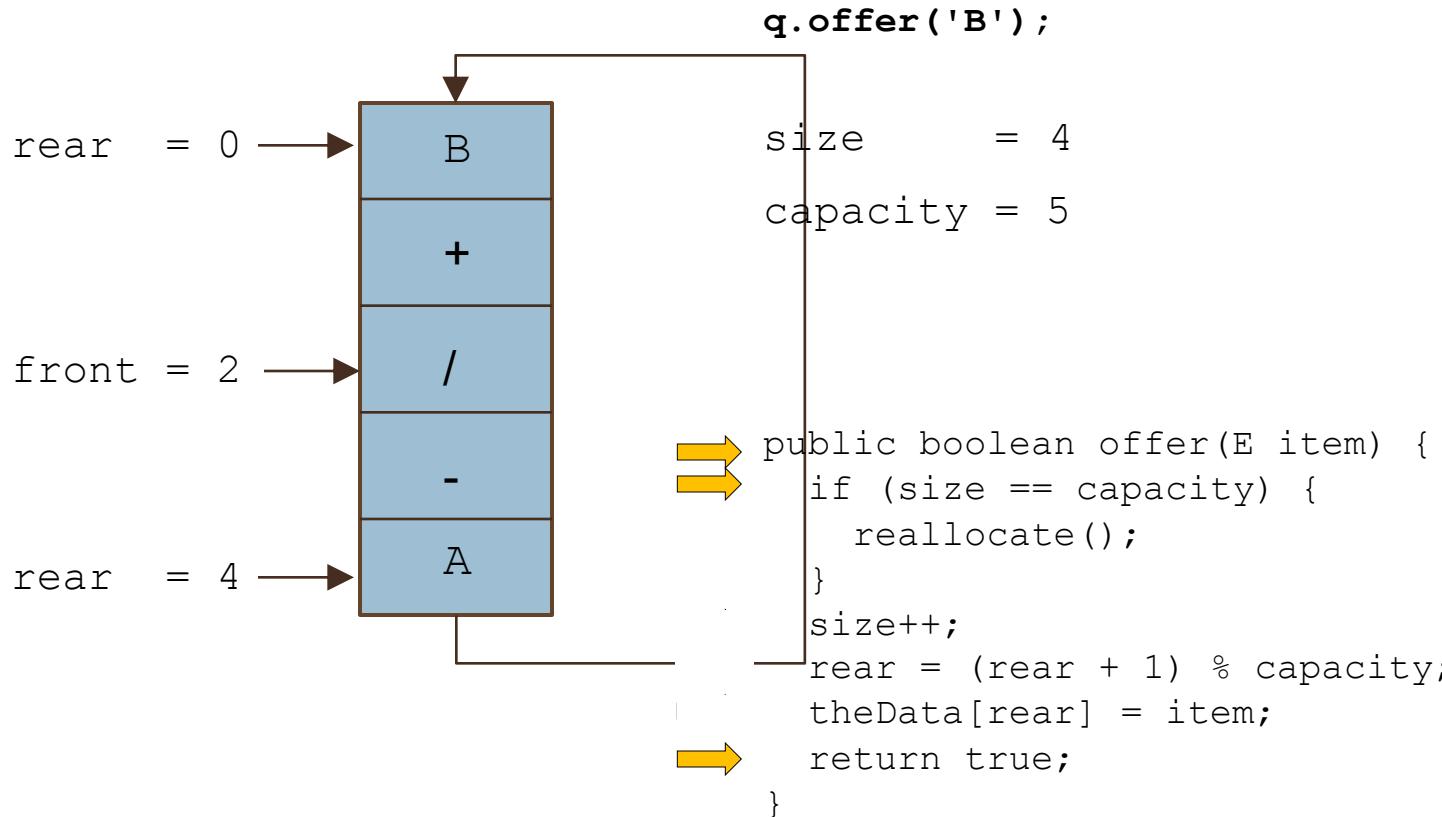
# Implementing a Queue Using a Circular Array (cont.)

31



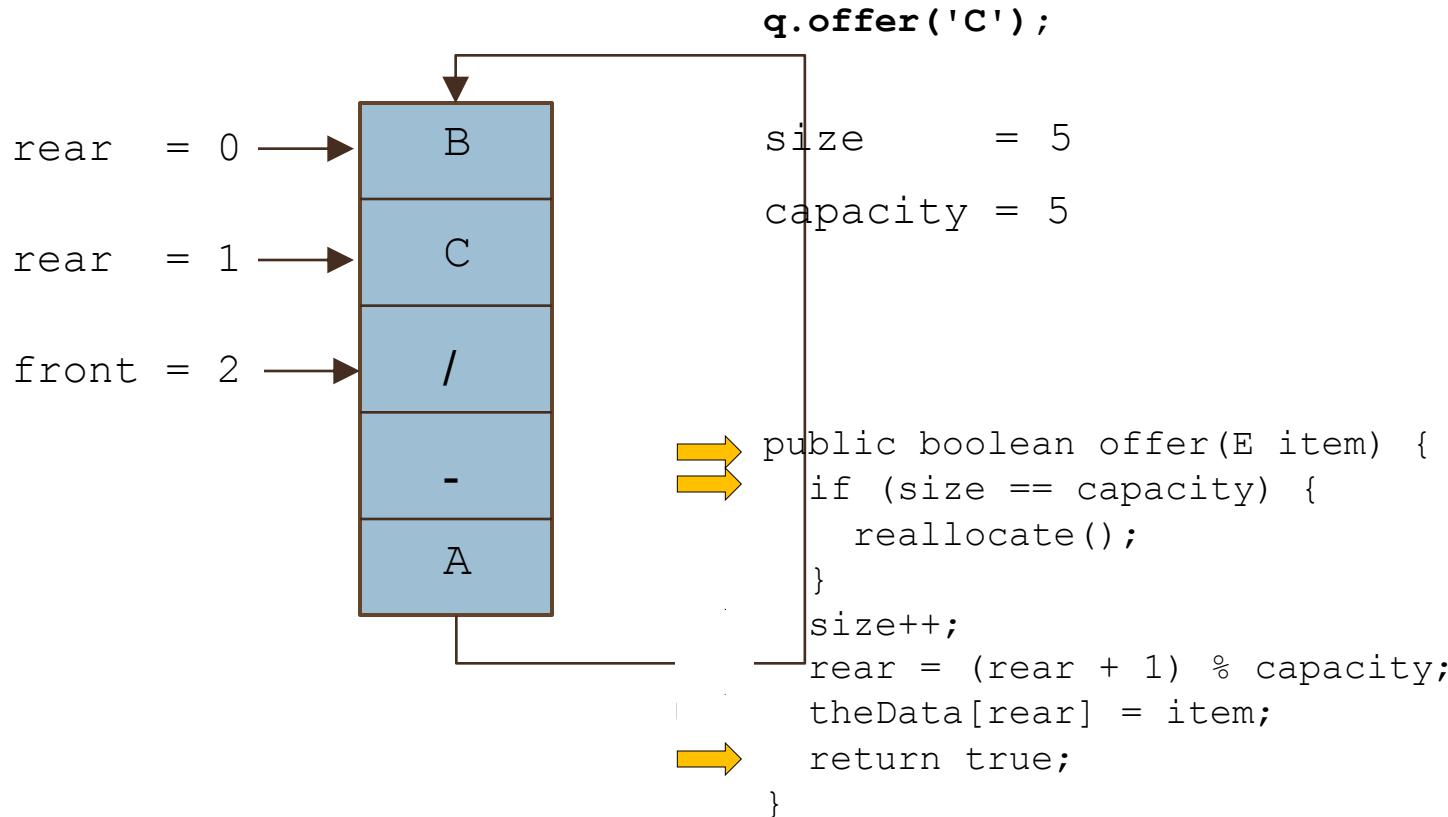
# Implementing a Queue Using a Circular Array (cont.)

32



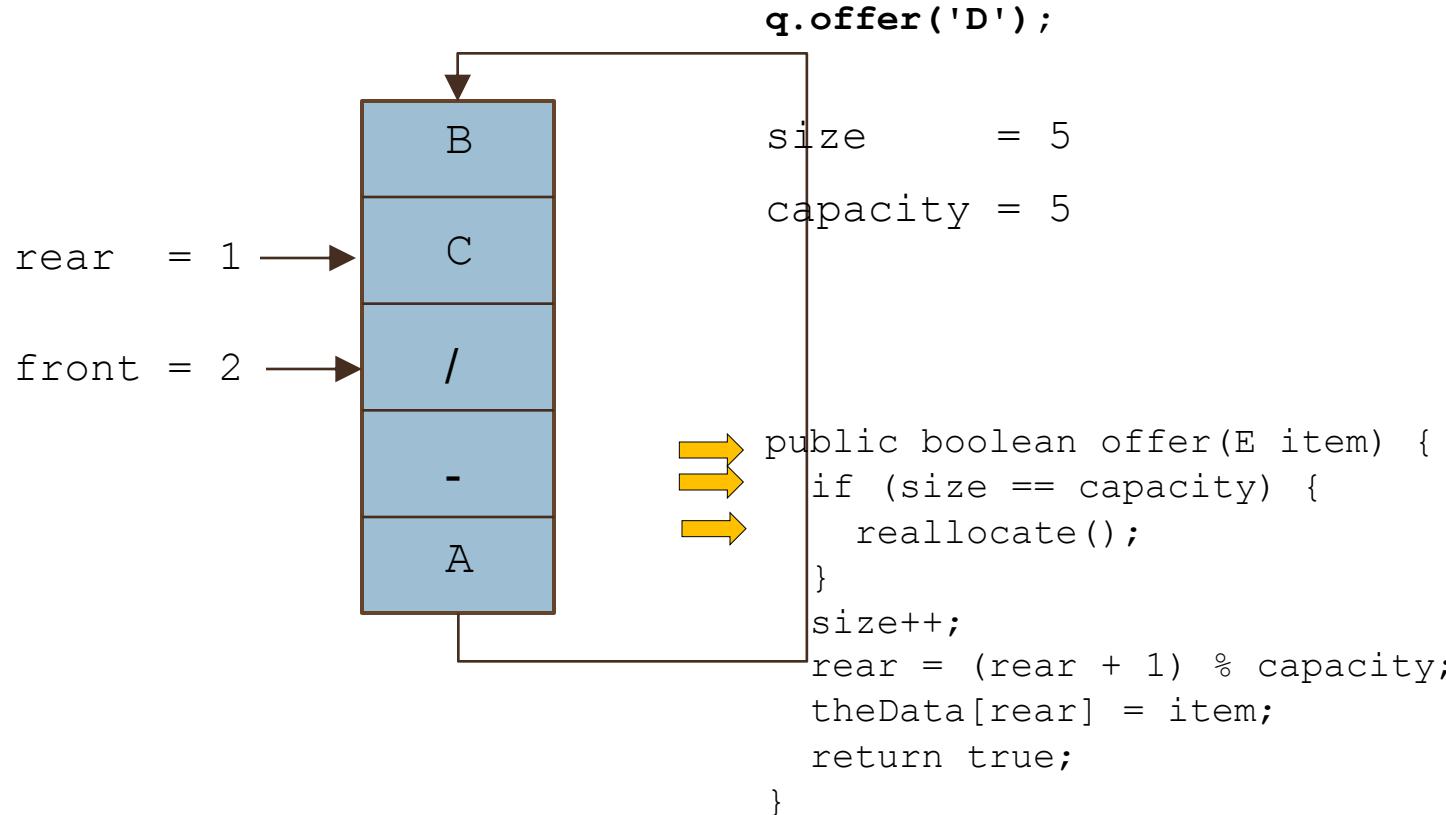
# Implementing a Queue Using a Circular Array (cont.)

33



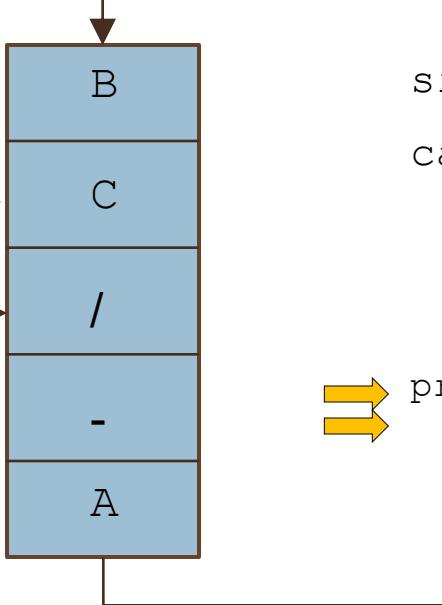
# Implementing a Queue Using a Circular Array (cont.)

34



# Implementing a Queue Using a Circular Array (cont.)

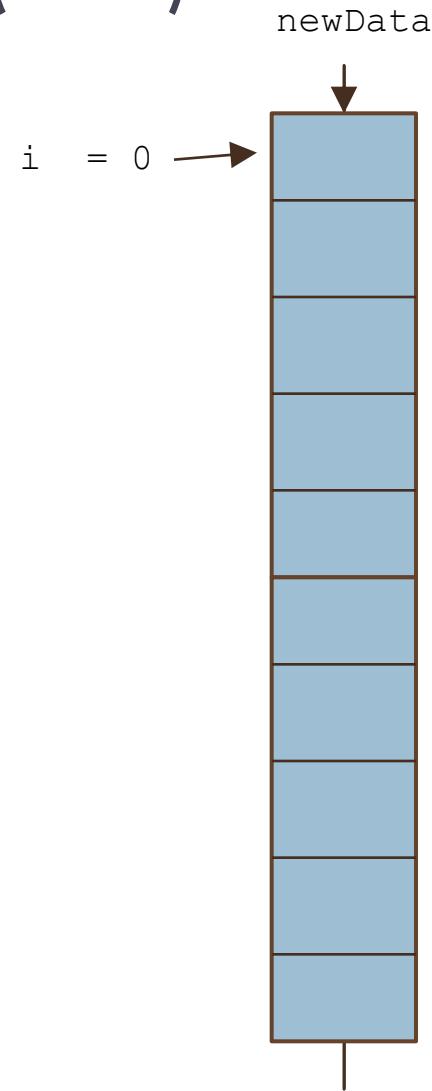
```
q.offer('D');  
size      = 5  
capacity = 5  
  
rear  = 1 →  
front = 2 →  
  
newCapacity = 10
```



```
theData  
B  
C  
/  
-  
A  
  
rear  = 1 →  
front = 2 →  
  
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

# Implementing a Queue Using a Circular Array

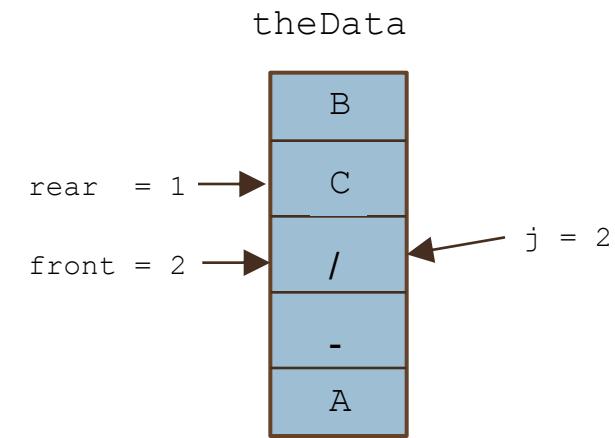
(cont.)



```
q.offer('D');
```

```
size      = 5
```

```
capacity = 5
```



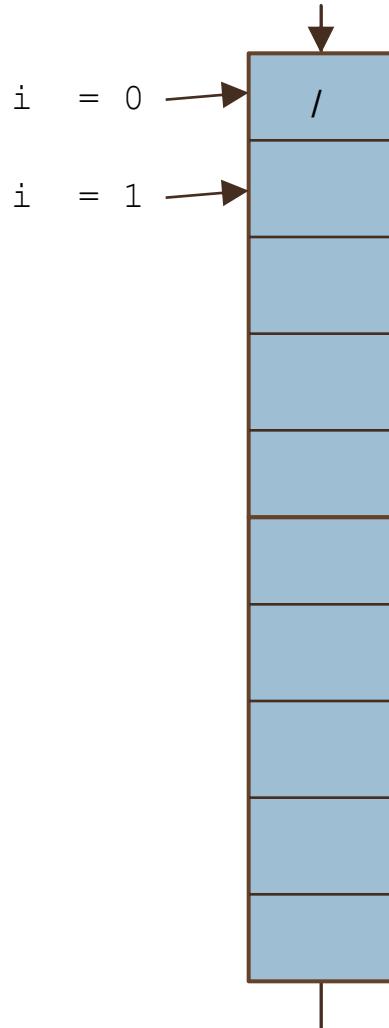
```
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    ➤ for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

newCapacity = 10

# Implementing a Queue Using a Circular Array

(cont.)

newData



newCapacity = 10

q.offer('D');

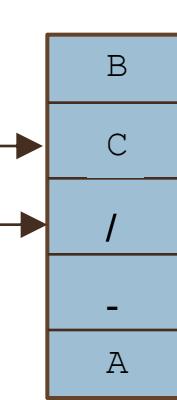
size = 5

capacity = 5

theData

rear = 1

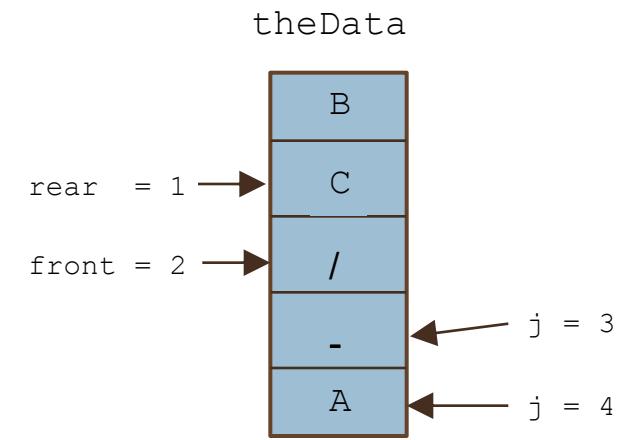
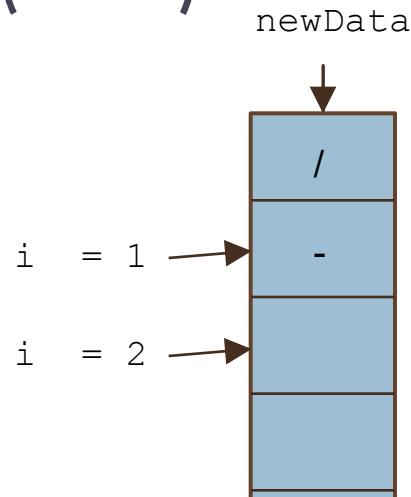
front = 2



```
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

# Implementing a Queue Using a Circular Array

(cont.)



```
q.offer('D');
```

```
size      = 5
```

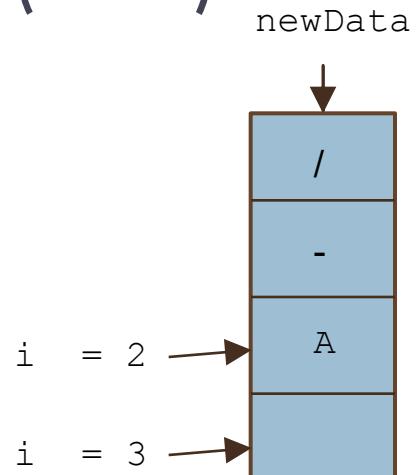
```
capacity = 5
```

```
newCapacity = 10
```

```
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

# Implementing a Queue Using a Circular Array

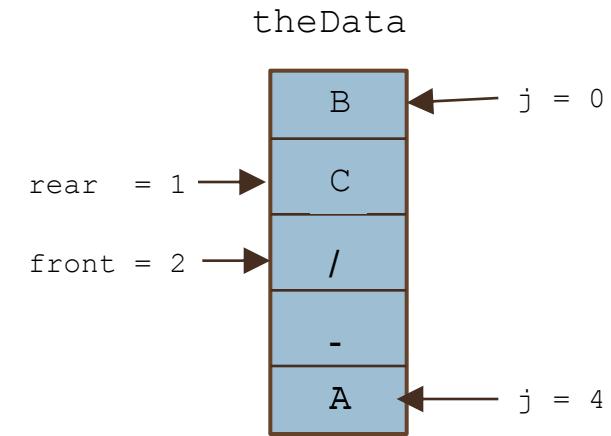
(cont.)



```
q.offer('D');
```

```
size      = 5
```

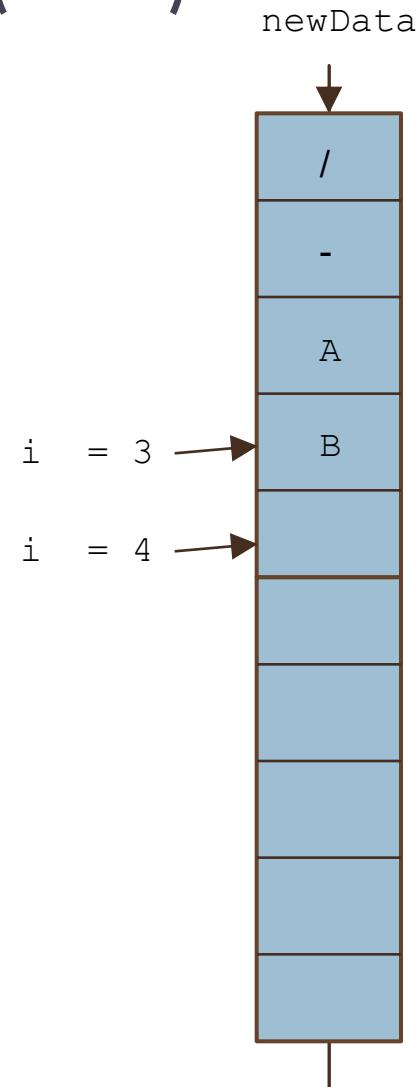
```
capacity = 5
```



```
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

# Implementing a Queue Using a Circular Array

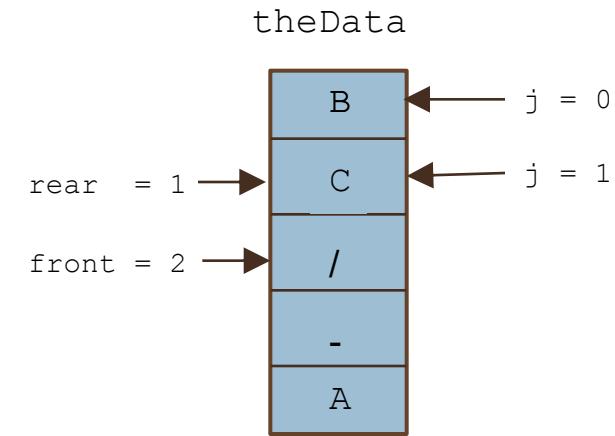
(cont.)



```
q.offer('D');
```

```
size      = 5
```

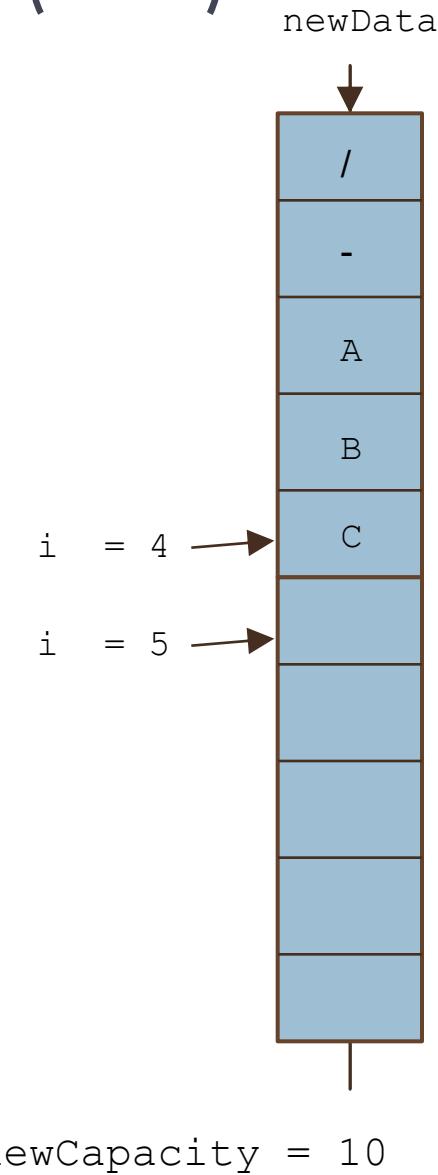
```
capacity = 5
```



```
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

# Implementing a Queue Using a Circular Array

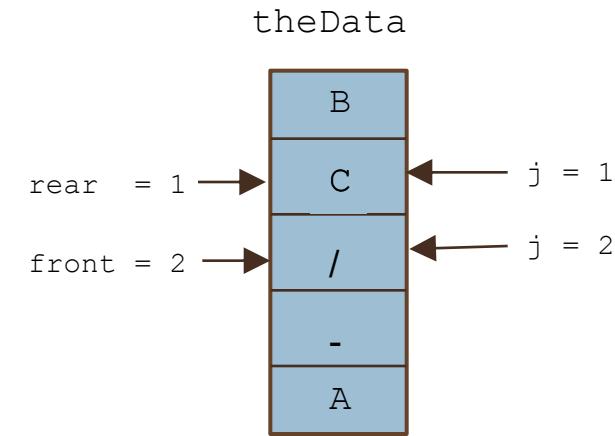
(cont.)



```
q.offer('D');
```

```
size      = 5
```

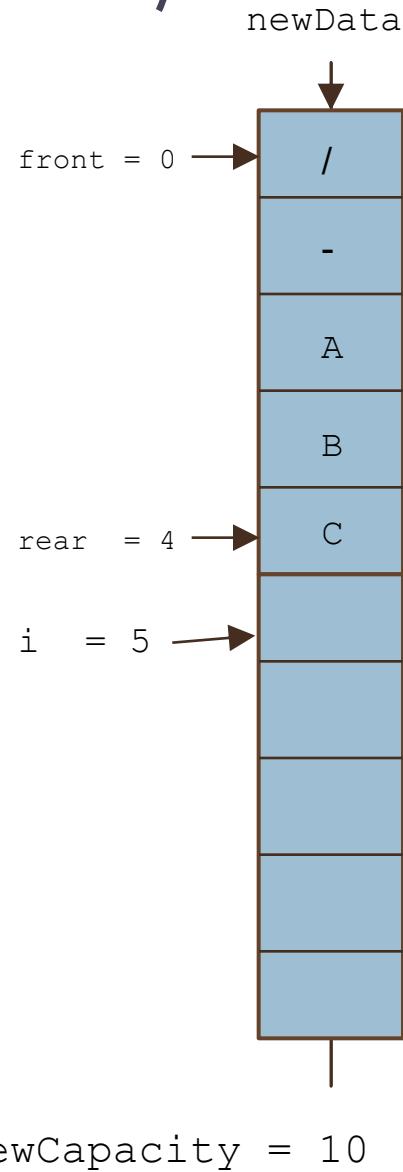
```
capacity = 5
```



```
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

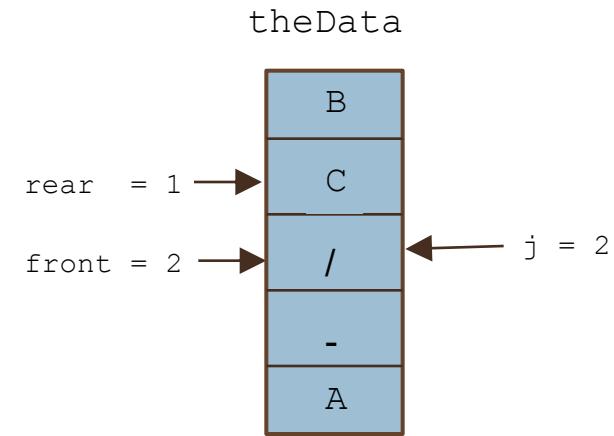
# Implementing a Queue Using a Circular Array

(cont.)



```
q.offer('D');
```

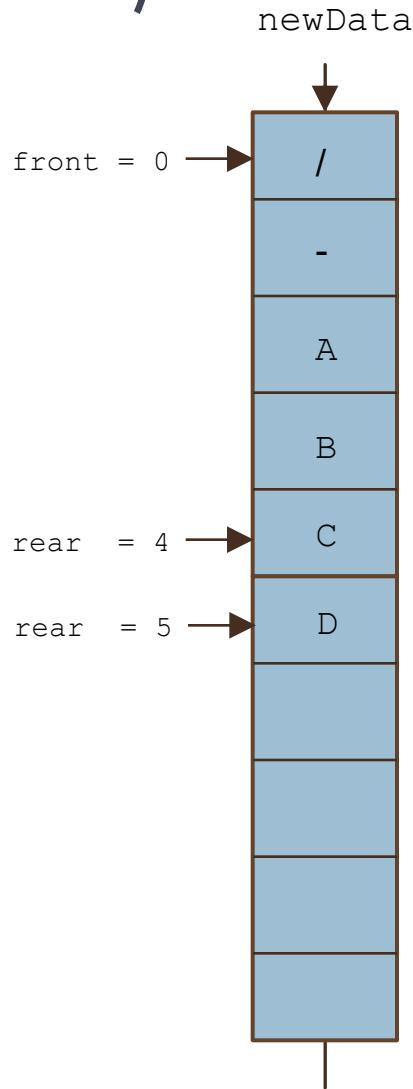
```
size      = 5  
capacity = 10
```



```
private void reallocate() {  
    int newCapacity = 2 * capacity;  
    E[] newData = (E[])new Object[newCapacity];  
    int j = front;  
    for (int i = 0; i < size; i++) {  
        newData[i] = theData[j];  
        j = (j + 1) % capacity;  
    }  
    front = 0;  
    rear = size - 1;  
    capacity = newCapacity;  
    theData = newData;  
}
```

# Implementing a Queue Using a Circular Array

(cont.)



```
q.offer('D');
```

```
size      = 6
```

```
capacity = 10
```

```
public boolean offer(E item) {  
    if (size == capacity) {  
        reallocate();  
    }  
    size++;  
    rear = (rear + 1) % capacity;  
    theData[rear] = item;  
    return true;  
}
```

# Implementing a Queue Using a Circular Array (cont.)

44

```
// Public Methods

/** Inserts an item at the rear of the queue.
 * post: item is added to the rear of the queue.
 * @param item The element to add
 * @return true (always successful) */
public boolean offer(E item) {
    if (size == capacity) {
        reallocate();
    }
    size++;
    rear = (rear + 1) % capacity;
    theData[rear] = item;
    return true;
}
```

# Implementing a Queue Using a Circular Array (cont.)

45

```
/** Removes the entry at the front of the queue and returns it
 * if the queue is not empty.
 * post: front references item that was second in the queue.
 * @return The item removed if successful or null if not
 */
public E poll() {
    if (size == 0) {
        return null;
    }
    E result = theData[front];
    front = (front + 1) % capacity;
    size--;
    return result;
}
```

# Comparing the Three Implementations

46

- Computation time
  - ▣ All three implementations (double-linked list, single-linked list, circular array) are comparable in terms of computation time
  - ▣ All operations are  $O(1)$  regardless of implementation
  - ▣ Although reallocating an array is  $O(n)$ , its is amortized over  $n$  items, so the cost per item is  $O(1)$

# Comparing the Three Implementations

## (cont.)

47

- Storage
  - ▣ Linked-list implementations require more storage due to the extra space required for the links
    - Each node for a single-linked list stores two references (one for the data, one for the link)
    - Each node for a double-linked list stores three references (one for the data, two for the links)
  - ▣ A double-linked list requires 1.5 times the storage of a single-linked list
  - ▣ A circular array that is filled to capacity requires half the storage of a single-linked list to store the same number of elements,
  - ▣ but a recently reallocated circular array is half empty, and requires the same storage as a single-linked list

# The Deque Interface

## Section 4.8

# Deque Interface

49

- A deque (pronounced "deck") is short for double-ended queue
- A double-ended queue allows insertions and removals from both ends
- The Java Collections Framework provides two implementations of the Deque interface
  - ▣ ArrayDeque
  - ▣ LinkedList
- ArrayDeque uses a resizable circular array, but (unlike LinkedList) does not support indexed operations
- ArrayDeque is the recommended implementation

# Deque Interface (cont.)

50

Method	Behavior
<code>boolean offerFirst(E item)</code>	Inserts <code>item</code> at the front of the deque. Returns <code>true</code> if successful; returns <code>false</code> if the item could not be inserted.
<code>boolean offerLast(E item)</code>	Inserts <code>item</code> at the rear of the deque. Returns <code>true</code> if successful; returns <code>false</code> if the item could not be inserted.
<code>void addFirst(E item)</code>	Inserts <code>item</code> at the front of the deque. Throws an exception if the item could not be inserted.
<code>void addLast(E item)</code>	Inserts <code>item</code> at the rear of the deque. Throws an exception if the item could not be inserted.
<code>E pollFirst()</code>	Removes the entry at the front of the deque and returns it; returns <code>null</code> if the deque is empty.
<code>E pollLast()</code>	Removes the entry at the rear of the deque and returns it; returns <code>null</code> if the deque is empty.
<code>E removeFirst()</code>	Removes the entry at the front of the deque and returns it if the deque is not empty. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>E removeLast()</code>	Removes the item at the rear of the deque and returns it. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>E peekFirst()</code>	Returns the entry at the front of the deque without removing it; returns <code>null</code> if the deque is empty.
<code>E peekLast()</code>	Returns the item at the rear of the deque without removing it; returns <code>null</code> if the deque is empty.
<code>E getFirst()</code>	Returns the entry at the front of the deque without removing it. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>E getLast()</code>	Returns the item at the rear of the deque without removing it. If the deque is empty, throws a <code>NoSuchElementException</code> .
<code>boolean removeFirstOccurrence(Object item)</code>	Removes the first occurrence of <code>item</code> in the deque. Returns <code>true</code> if the item was removed.
<code>boolean removeLastOccurrence(Object item)</code>	Removes the last occurrence of <code>item</code> in the deque. Returns <code>true</code> if the item was removed.
<code>Iterator&lt;E&gt; iterator()</code>	Returns an iterator to the elements of this deque in the proper sequence.
<code>Iterator&lt;E&gt; descendingIterator()</code>	Returns an iterator to the elements of this deque in reverse sequential order.

# Deque Example

51

Deque Method	Deque d	Effect
d.offerFirst('b')	b	'b' inserted at front
d.offerLast('y')	by	'y' inserted at rear
d.addLast('z')	byz	'z' inserted at rear
d.addFirst('a')	abyz	'a' inserted at front
d.peekFirst()	abyz	Returns 'a'
d.peekLast()	abyz	Returns 'z'
d.pollLast()	aby	Removes 'z'
d.pollFirst()	by	Removes 'a'

# Deque Interface (cont.)

52

- The Deque interface extends the Queue interface, so it can be used as a queue
- A deque can be used as a stack if elements are pushed and popped from the front of the deque
- Using the Deque interface is preferable to using the legacy Stack class (based on Vector)

Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()
empty()	isEmpty()



# CS 570: Data Structures

## Recursion (Part II)

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 5: RECURSION

Part II

# Chapter Objectives

3

- To understand how to think recursively
- To learn how to trace a recursive method
- To learn how to write recursive algorithms and methods for searching arrays
- To learn about recursive data structures and recursive methods for a `LinkedList` class
- To understand how to use recursion to solve the Towers of Hanoi problem
- To understand how to use recursion to process two-dimensional images
- To learn how to apply backtracking to solve search problems such as finding a path through a maze

# Recursion

4

- Recursion can solve many programming problems that are difficult to conceptualize and solve linearly
- In the field of artificial intelligence, recursion often is used to write programs that exhibit intelligent behavior:
  - ▣ playing games of chess
  - ▣ proving mathematical theorems
  - ▣ recognizing patterns, and so on
- Recursive algorithms can
  - ▣ compute factorials
  - ▣ compute a greatest common divisor
  - ▣ process data structures (strings, arrays, linked lists, etc.)
  - ▣ search efficiently using a binary search
  - ▣ find a path through a maze, and more

# Week 9

- Reading Assignment: Koffman and Wolfgang,  
Sections 5.4-5.6

# Recursive Data Structures

## Section 5.4

# Recursive Data Structures

7

- Computer scientists often encounter data structures that are defined recursively – with another version of itself as a component
- Linked lists and trees (Chapter 6) can be defined as recursive data structures
- Recursive methods provide a natural mechanism for processing recursive data structures
- The first language developed for artificial intelligence research was a recursive language called LISP

# Recursive Definition of a Linked List

8

- A linked list is a collection of nodes such that each node references another linked list consisting of the nodes that follow it in the list
- The last node references an empty list
- A linked list is empty, or it contains a node, called the list head, it stores data and a reference to a linked list

# Class LinkedListRec

9

- We define a class `LinkedListRec<E>` that implements several list operations using recursive methods

```
public class LinkedListRec<E> {  
    private Node<E> head;  
  
    // inner class Node<E> here  
    // (from chapter 2)  
}
```

# Recursive size Method

10

```
/** Finds the size of a list.  
 * @param head The head of the current list  
 * @return The size of the current list  
 */  
private int size(Node<E> head) {  
    if (head == null)  
        return 0;  
    else  
        return 1 + size(head.next);  
}  
  
/** Wrapper method for finding the size of a list.  
 * @return The size of the list  
 */  
public int size() {  
    return size(head);  
}
```

# Recursive `toString` Method

11

```
/** Returns the string representation of a list.
 * @param head The head of the current list
 * @return The state of the current list
 */
private String toString(Node<E> head) {
    if (head == null)
        return "";
    else
        return head.data + "\n" + toString(head.next);
}

/** Wrapper method for returning the string representation of a list.
 * @return The string representation of the list
 */
public String toString() {
    return toString(head);
}
```

# Recursive replace Method

12

```
/** Replaces all occurrences of oldObj with newObj.  
 * post: Each occurrence of oldObj has been replaced by newObj.  
 * @param head The head of the current list  
 * @param oldObj The object being removed  
 * @param newObj The object being inserted  
 */  
private void replace(Node<E> head, E oldObj, E newObj) {  
    if (head != null) {  
        if (oldObj.equals(head.data))  
            head.data = newObj;  
        replace(head.next, oldObj, newObj);  
    }  
}  
  
/* Wrapper method for replacing oldObj with newObj.  
 * post: Each occurrence of oldObj has been replaced by newObj.  
 * @param oldObj The object being removed  
 * @param newObj The object being inserted  
 */  
public void replace(E oldObj, E newObj) {  
    replace(head, oldObj, newObj);  
}
```

# Recursive add Method

13

```
/** Adds a new node to the end of a list.
 * @param head The head of the current list
 * @param data The data for the new node
 */
private void add(Node<E> head, E data) {
    // If the list has just one element, add to it.
    if (head.next == null)
        head.next = new Node<E>(data);
    else
        add(head.next, data);      // Add to rest of list.
}

/** Wrapper method for adding a new node to the end of a list.
 * @param data The data for the new node
 */
public void add(E data) {
    if (head == null)
        head = new Node<E>(data); // List has 1 node.
    else
        add(head, data);
}
```

# Recursive remove Method

14

```
/** Removes a node from a list.  
post: The first occurrence of outData is removed.  
@param head The head of the current list  
@param pred The predecessor of the list head  
@param outData The data to be removed  
@return true if the item is removed  
        and false otherwise  
*/  
private boolean remove(Node<E> head, Node<E> pred, E outData) {  
    if (head == null) // Base case - empty list.  
        return false;  
    else if (head.data.equals(outData)) { // 2nd base case.  
        pred.next = head.next; // Remove head.  
        return true;  
    } else  
        return remove(head.next, head, outData);  
}
```

# Recursive remove Method (cont.)

15

```
/** Wrapper method for removing a node (in LinkedListRec).
 * post: The first occurrence of outData is removed.
 * @param outData The data to be removed
 * @return true if the item is removed,
 *         and false otherwise
 */
public boolean remove(E outData) {
    if (head == null)
        return false;
    else if (head.data.equals(outData)) {
        head = head.next;
        return true;
    } else
        return remove(head.next, head, outData);
}
```

# Problem Solving with Recursion

## Section 5.5

# Simplified Towers of Hanoi

17

- Move the three disks to a different peg, maintaining their order (largest disk on bottom, smallest on top, etc.)
  - ▣ Only the top disk on a peg can be moved to another peg
  - ▣ A larger disk cannot be placed on top of a smaller disk



# Towers of Hanoi

18

## Problem Inputs

Number of disks (an integer)

Letter of starting peg: L (left), M (middle), or R (right)

Letter of destination peg: (L, M, or R), but different from starting peg

Letter of temporary peg: (L, M, or R), but different from starting peg and destination peg

## Problem Outputs

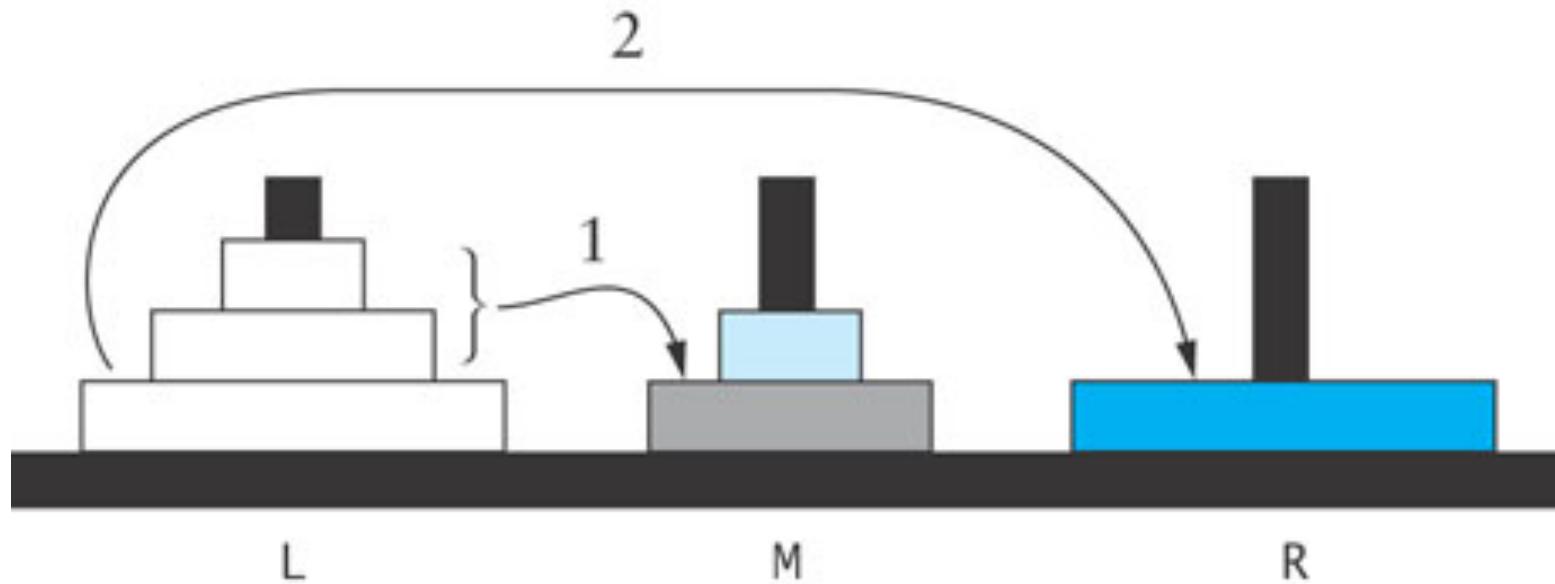
A list of moves

# Algorithm for Towers of Hanoi

19

**Solution to Three-Disk Problem: Move Three Disks from Peg L to Peg R**

1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.

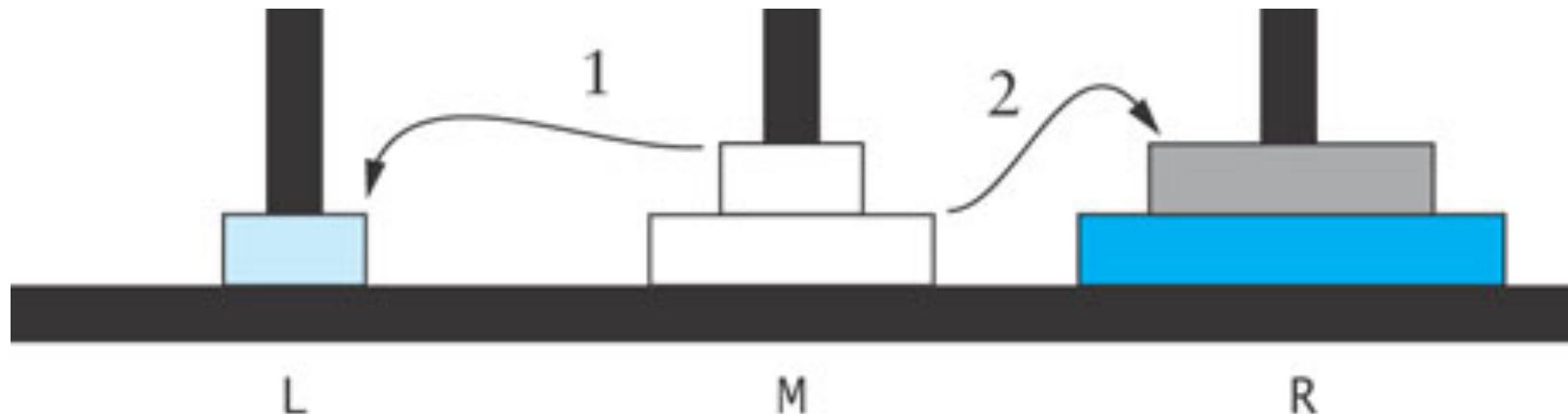


# Algorithm for Towers of Hanoi (cont.)

20

## Solution to Two-Disk Problem: Move Top Two Disks from Peg M to Peg R

1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.

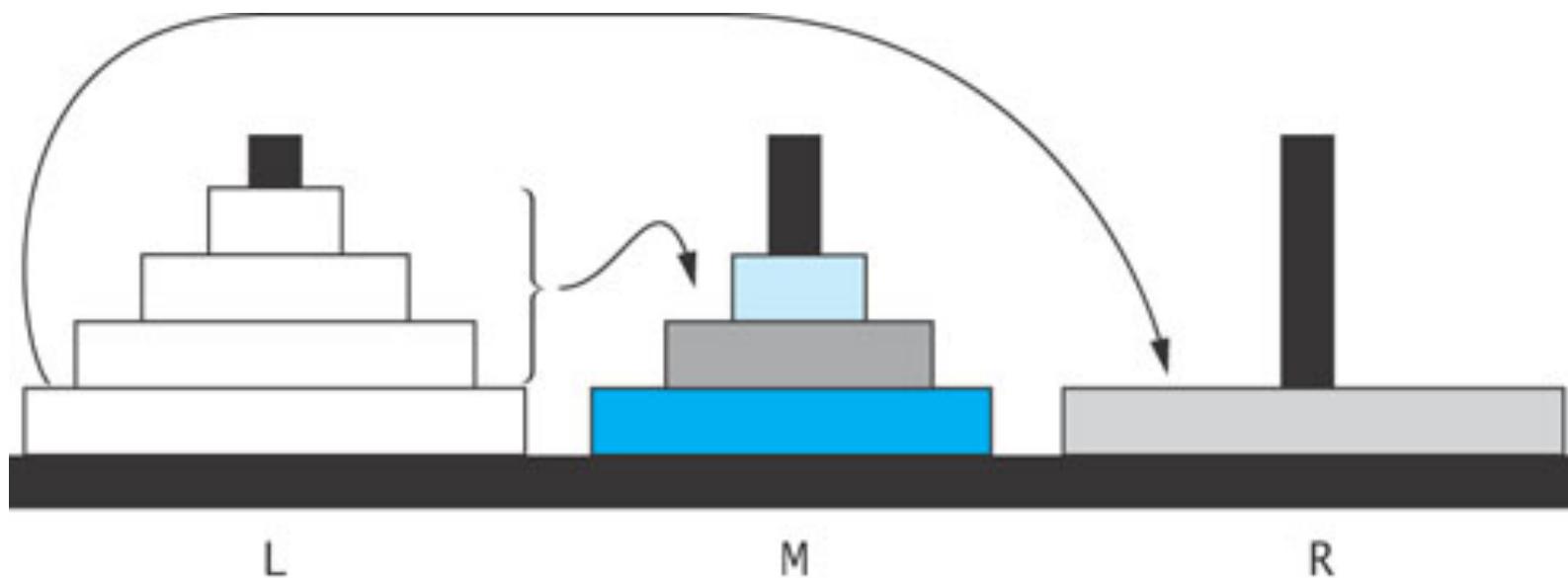


# Algorithm for Towers of Hanoi (cont.)

21

## Solution to Four-Disk Problem: Move Four Disks from Peg L to Peg R

1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.



# Recursive Algorithm for Towers of Hanoi

22

Recursive Algorithm for  $n$ -Disk Problem: Move  $n$  Disks from the Starting Peg to the Destination Peg

```
if  $n$  is 1
    move disk 1 (the smallest disk) from the starting peg to the destination
    peg
else
    move the top  $n - 1$  disks from the starting peg to the temporary peg
        (neither starting nor destination peg)
    move disk  $n$  (the disk at the bottom) from the starting peg to the
        destination peg
    move the top  $n - 1$  disks from the temporary peg to the destination peg
```

# Implementation of Recursive Towers of Hanoi

23

```
/** Class that solves Towers of Hanoi problem. */
public class TowersOfHanoi {
    /** Recursive method for "moving" disks.
        pre: startPeg, destPeg, tempPeg are different.
        @param n is the number of disks
        @param startPeg is the starting peg
        @param destPeg is the destination peg
        @param tempPeg is the temporary peg
        @return A string with all the required disk moves
    */
    public static String showMoves(int n, char startPeg,
                                   char destPeg, char tempPeg) {
        if (n == 1) {
            return "Move disk 1 from peg " + startPeg +
                   " to peg " + destPeg + "\n";
        } else { // Recursive step
            return showMoves(n - 1, startPeg, tempPeg, destPeg)
                + "Move disk " + n + " from peg " + startPeg
                + " to peg " + destPeg + "\n"
                + showMoves(n - 1, tempPeg, destPeg, startPeg);
        }
    }
}
```

# Counting Cells in a Blob

24

- Consider how we might process an image that is presented as a two-dimensional array of color values
- Information in the image may come from
  - ▣ an X-ray
  - ▣ an MRI
  - ▣ satellite imagery
  - ▣ etc.
- The goal is to determine the size of any area in the image that is considered abnormal because of its color values

# Counting Cells in a Blob – the Problem

25

- Given a two-dimensional grid of cells, each cell contains either a normal background color or a second color, which indicates the presence of an abnormality
- A *blob* is a collection of contiguous abnormal cells
- A user will enter the x, y coordinates of a cell in the blob, and the program will determine the count of all cells in that blob

# Counting Cells in a Blob - Analysis

26

- Problem Inputs
  - the two-dimensional grid of cells
  - the coordinates of a cell in a blob
- Problem Outputs
  - the count of cells in the blob

# Counting Cells in a Blob - Design

27

Method	Behavior
void recolor(int x, int y, Color aColor)	Resets the color of the cell at position (x, y) to aColor.
Color getColor(int x, int y)	Retrieves the color of the cell at position (x, y).
int getNRows()	Returns the number of cells in the y-axis.
int getNCols()	Returns the number of cells in the x-axis.

Method	Behavior
int countCells(int x, int y)	Returns the number of cells in the blob at (x, y).

# Counting Cells in a Blob - Design (cont.)

28

## Algorithm for `countCells(x, y)`

```
if the cell at (x, y) is outside the grid  
    the result is 0  
else if the color of the cell at (x, y) is not the abnormal color  
    the result is 0  
else  
    set the color of the cell at (x, y) to a temporary color  
    the result is 1 plus the number of cells in each piece of the blob that  
    includes a nearest neighbor
```

# Counting Cells in a Blob - Implementation

29

```
import java.awt.*;  
/** Class that solves problem of counting abnormal cells. */  
public class Blob implements GridColors {  
  
    /** The grid */  
    private TwoDimGrid grid;  
  
    /** Constructors */  
    public Blob(TwoDimGrid grid) {  
        this.grid = grid;  
    }  
}
```

# Counting Cells in a Blob - Implementation (cont.)

30

```
/** Finds the number of cells in the blob at (x,y).
pre: Abnormal cells are in ABNORMAL color;
     Other cells are in BACKGROUND color.
post: All cells in the blob are in the TEMPORARY color.
@param x The x-coordinate of a blob cell
@param y The y-coordinate of a blob cell
@return The number of cells in the blob that contains (x, y)
*/
public int countCells(int x, int y) {
    int result;

    if (x < 0 || x >= grid.getNCols()
        || y < 0 || y >= grid.getNRows())
        return 0;
    else if (!grid.getColor(x, y).equals(ABNORMAL))
        return 0;
    else {
        grid.recolor(x, y, TEMPORARY);
        return 1
            + countCells(x - 1, y + 1) + countCells(x, y + 1)
            + countCells(x + 1, y + 1) + countCells(x - 1, y)
            + countCells(x + 1, y) + countCells(x - 1, y - 1)
            + countCells(x, y - 1) + countCells(x + 1, y - 1);
    }
}
```

# Counting Cells in a Blob -Testing

31

Toggle a button to change its color --  
When done, press SOLVE.  
Blob count will start at the last button pressed

0,0	1,0	2,0	3,0	4,0	5,0
0,1	1,1	2,1	3,1	4,1	5,1
0,2	1,2	2,2	3,2	4,2	5,2
0,3	1,3	2,3	3,3	4,3	5,3

**SOLVE**

Toggle a button to change its color --  
When done, press SOLVE.  
Blob count will start at the last button pressed

0,0	1,0	2,0			
0,1	1,1	2,1	3,1		5,1
0,2	1,2	2,2			4,2
0,3	1,3	2,3	3,3		5,3

**SOLVE**

# Counting Cells in a Blob -Testing (cont.)

32

- Verify that the code works for the following cases:
  - A starting cell that is on the edge of the grid
  - A starting cell that has no neighboring abnormal cells
  - A starting cell whose only abnormal neighbor cells are diagonally connected to it
  - A "bull's-eye": a starting cell whose neighbors are all normal but their neighbors are abnormal
  - A starting cell that is normal
  - A grid that contains all abnormal cells
  - A grid that contains all normal cells

# Backtracking

## Section 5.6

# Backtracking

34

- Backtracking is an approach to implementing a systematic trial and error search for a solution
- An example is finding a path through a maze
- If you are attempting to walk through a maze, you will probably walk down a path as far as you can go
  - ▣ Eventually, you will reach your destination or you won't be able to go any farther
  - ▣ If you can't go any farther, you will need to consider alternative paths
- Backtracking is a systematic, nonrepetitive approach to trying alternative paths and eliminating them if they don't work

# Backtracking (cont.)

35

- If you never try the same path more than once, you will eventually find a solution path if one exists
- Problems that are solved by backtracking can be described as a set of choices made by some method
- Recursion allows you to implement backtracking in a relatively straightforward manner
  - ▣ Each activation frame is used to remember the choice that was made at that particular decision point
- A program that plays chess may involve some kind of backtracking algorithm

# Finding a Path through a Maze

36

## □ Problem

- Use backtracking to find and display the path through a maze
- From each point in a maze, you can move to the next cell in a horizontal or vertical direction, if the cell is not blocked

# Finding a Path through a Maze (cont.)

37

## □ Analysis

- The maze will consist of a grid of colored cells
- The starting point is at the top left corner (0,0)
- The exit point is at the bottom right corner  
`(getNCols() - 1, getNRows() - 1)`
- All cells on the path will be BACKGROUND color
- All cells that represent barriers will be ABNORMAL color
- Cells that we have visited will be TEMPORARY color
- If we find a path, all cells on the path will be set to PATH color

# Recursive Algorithm for Finding Maze Path

38

## Recursive Algorithm for `findMazePath(x, y)`

```
if the current cell is outside the maze
    return false (you are out of bounds)
else if the current cell is part of the barrier or has already been visited
    return false (you are off the path or in a cycle)
else if the current cell is the maze exit
    recolor it to the path color and return true (you have successfully
    completed the maze)
else // Try to find a path from the current path to the exit:
    mark the current cell as on the path by recoloring it to the path color
    for each neighbor of the current cell
        if a path exists from the neighbor to the maze exit
            return true
    // No neighbor of the current cell is on the path
    recolor the current cell to the temporary color (visited) and return
    false
```

# Implementation

39

```
import java.awt.*;  
/** Class that solves maze problems with  
backtracking.  
 *  @author Koffman and Wolfgang  
 * . */  
  
public class Maze implements GridColors {  
  
    /** The maze */  
    private TwoDimGrid maze;  
  
    public Maze(TwoDimGrid m) {  
        maze = m;  
    }
```

# Implementation

40

```
/** Wrapper method. */
public boolean findMazePath() {
    return findMazePath(0, 0); // (0, 0) is the start point.
}

/** Attempts to find a path through point (x, y).
    pre: Possible path cells are in BACKGROUND color;
         barrier cells are in ABNORMAL color.
    post: If a path is found, all cells on it are set to the
          PATH color; all cells that were visited but are
          not on the path are in the TEMPORARY color.
    @param x The x-coordinate of current point
    @param y The y-coordinate of current point
    @return If a path through (x, y) is found, true;
            otherwise, false
*/
```

# Implementation

41

```
public boolean findMazePath(int x, int y) {  
    if (x < 0 || y < 0  
        || x >= maze.getNCols() || y >= maze.getNRows())  
        return false; // Cell is out of bounds.  
    else if (!maze.getColor(x, y).equals(BACKGROUND))  
        return false; // Cell is on barrier or dead end.  
    else if (x == maze.getNCols() - 1  
            && y == maze.getNRows() - 1) {  
        maze.recolor(x, y, PATH); // Cell is on path  
        return true; // and is maze exit.  
    }  
}
```

# Implementation

42

```
else { // Recursive case.  
    // Attempt to find a path from each neighbor.  
    // Tentatively mark cell as on path.  
    maze.recolor(x, y, PATH);  
    if (findMazePath(x - 1, y)  
        || findMazePath(x + 1, y)  
        || findMazePath(x, y - 1)  
        || findMazePath(x, y + 1)) {  
        return true;  
    }  
    else {  
        maze.recolor(x, y, TEMPORARY); // Dead end.  
        return false;  
    } } }
```

# Testing

43

- Test for a variety of test cases:
  - Mazes that can be solved
  - Mazes that can't be solved
  - A maze with no barrier cells
  - A maze with a single barrier cell at the exit point



# CS 570: Data Structures

## Trees

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 6

# TREES

# Chapter Objectives

3

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the difference between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays

# Week 10

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 6.1-6.3, 6.5

# Trees - Introduction

5

- All previous data organizations we've learned are linear—each element can have only one predecessor or successor
- Accessing all elements in a linear sequence is  $O(n)$
- Trees are nonlinear and hierarchical
- Tree nodes can have multiple successors (but only one predecessor)

# Trees - Introduction (cont.)

6

- Trees can represent hierarchical organizations of information:
  - ▣ class hierarchy
  - ▣ disk directory and subdirectories
  - ▣ family tree
- Trees are recursive data structures because they can be defined recursively
- Many methods to process trees are written recursively

# Trees - Introduction (cont.)

7

- This chapter focuses on the *binary tree*
- In a binary tree each element has two successors
- Binary trees can be represented by arrays and linked data structures
- Searching in a binary search tree, an ordered tree, is generally more efficient than searching in an ordered list— $O(\log n)$  versus  $O(n)$

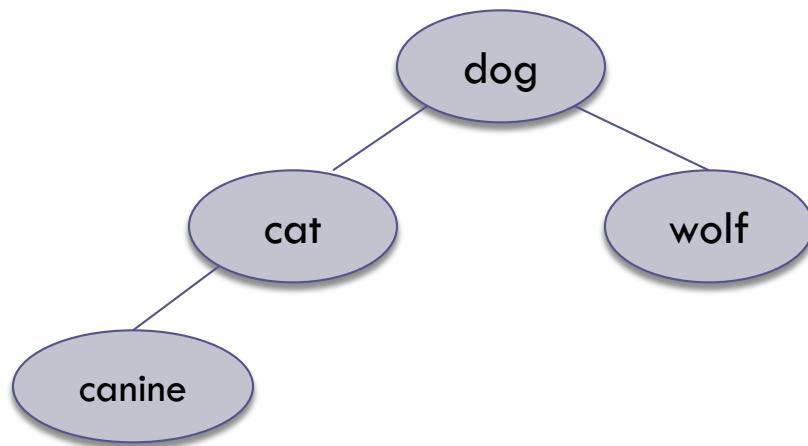
# Tree Terminology and Applications

## Section 6.1

# Tree Terminology

9

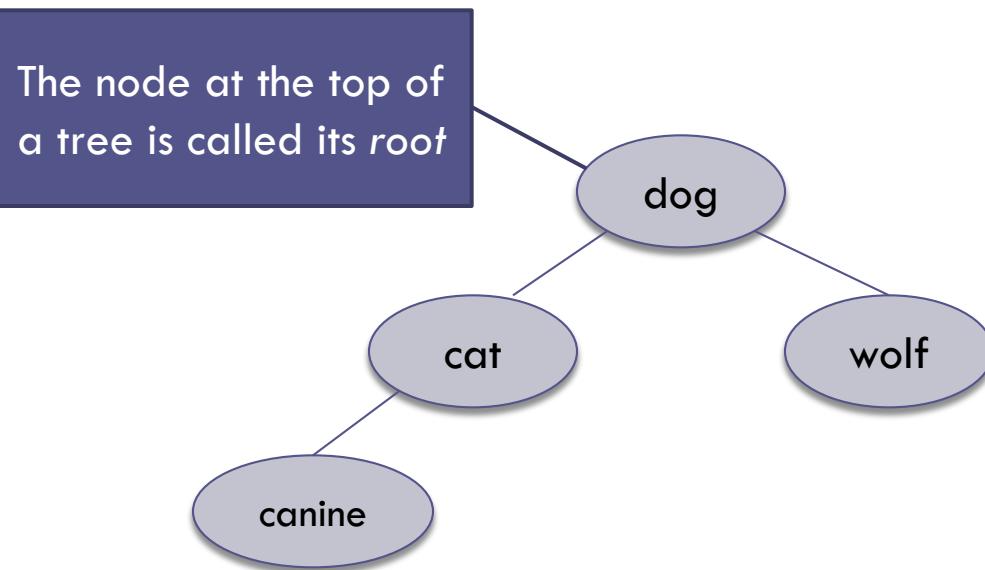
A tree consists of a collection of elements or nodes, with each node linked to its successors



# Tree Terminology (cont.)

10

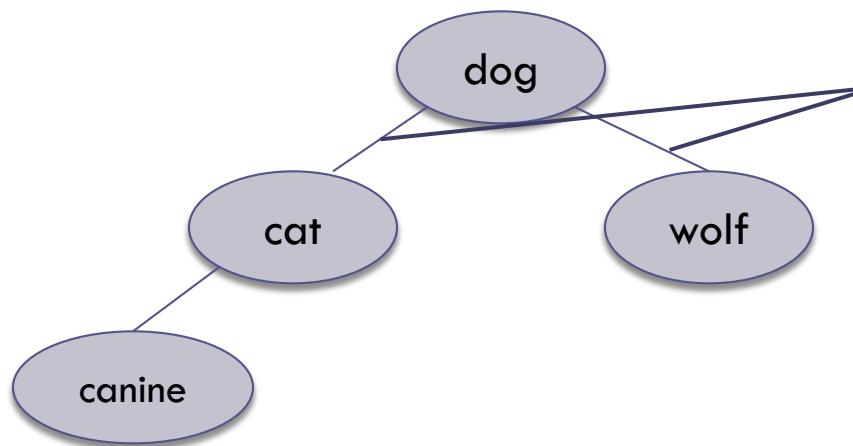
A tree consists of a collection of elements or nodes, with each node linked to its successors



# Tree Terminology (cont.)

11

A tree consists of a collection of elements or nodes, with each node linked to its successors



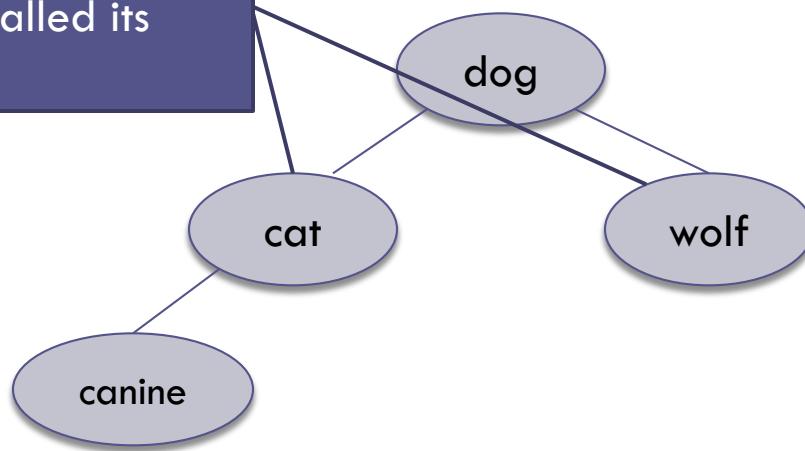
The links from a node to its successors are called *branches*

# Tree Terminology (cont.)

12

A tree consists of a collection of elements or nodes, with each node linked to its successors

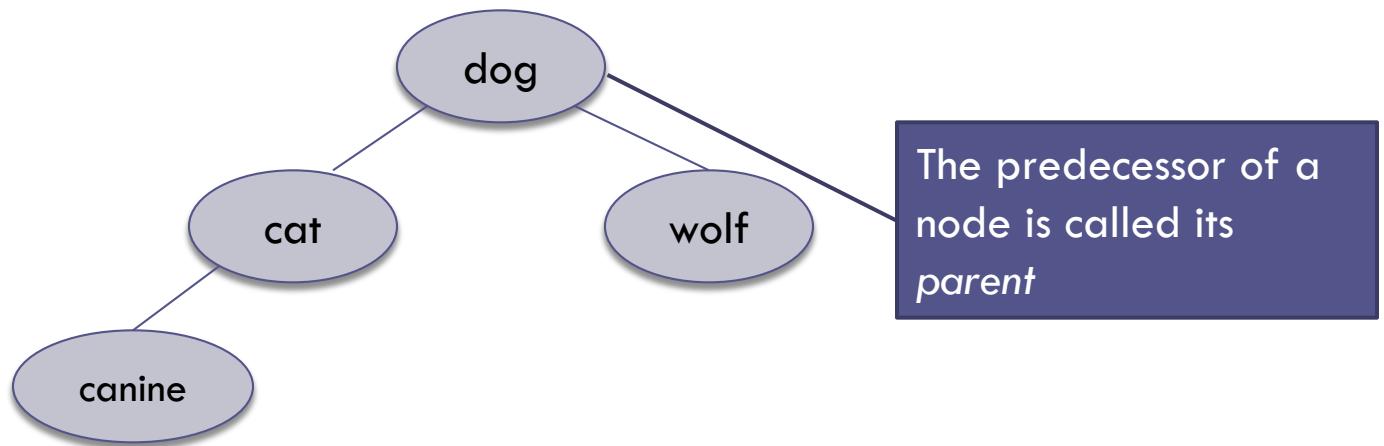
The successors of a node are called its *children*



# Tree Terminology (cont.)

13

A tree consists of a collection of elements or nodes, with each node linked to its successors

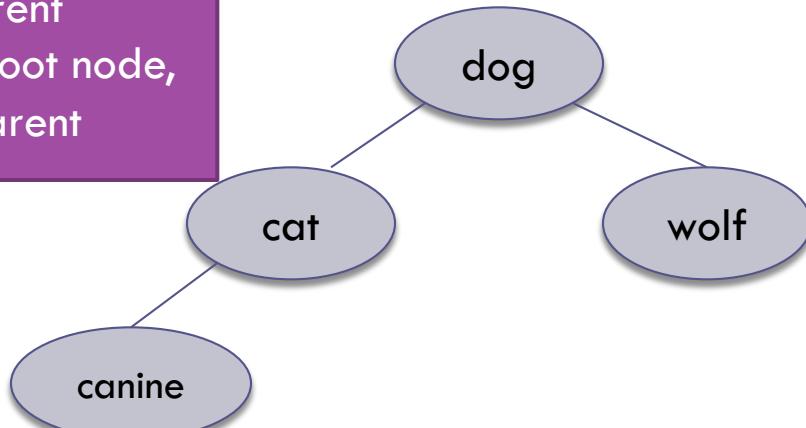


# Tree Terminology (cont.)

14

A tree consists of a collection of elements or nodes, with each node linked to its successors

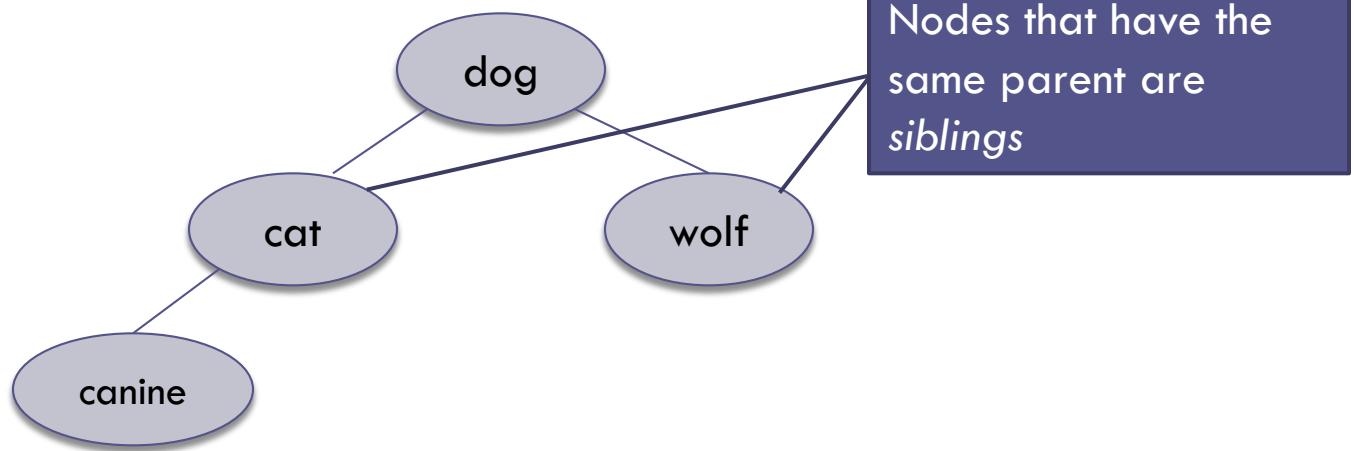
Each node in a tree has exactly one parent except for the root node, which has no parent



# Tree Terminology (cont.)

15

A tree consists of a collection of elements or nodes, with each node linked to its successors

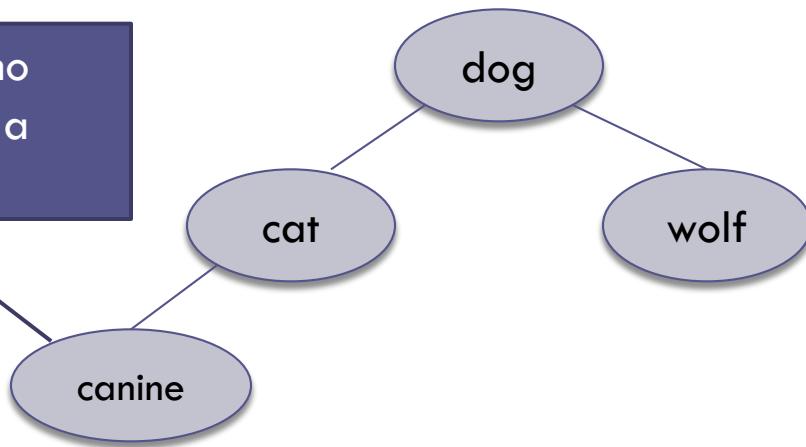


# Tree Terminology (cont.)

16

A tree consists of a collection of elements or nodes, with each node linked to its successors

A node that has no children is called a *leaf node*

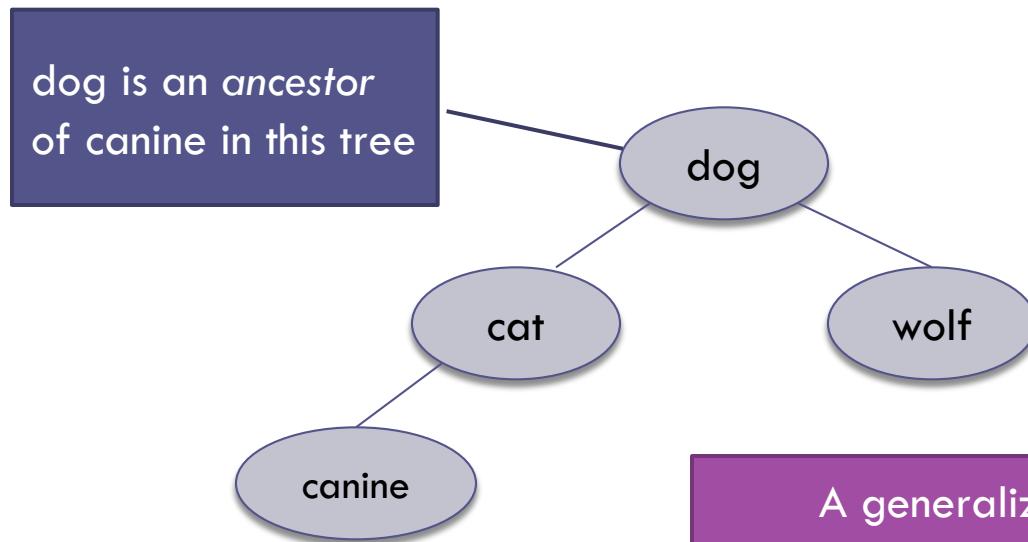


Leaf nodes also are known as *external nodes*, and nonleaf nodes are known as *internal nodes*

# Tree Terminology (cont.)

17

A tree consists of a collection of elements or nodes, with each node linked to its successors

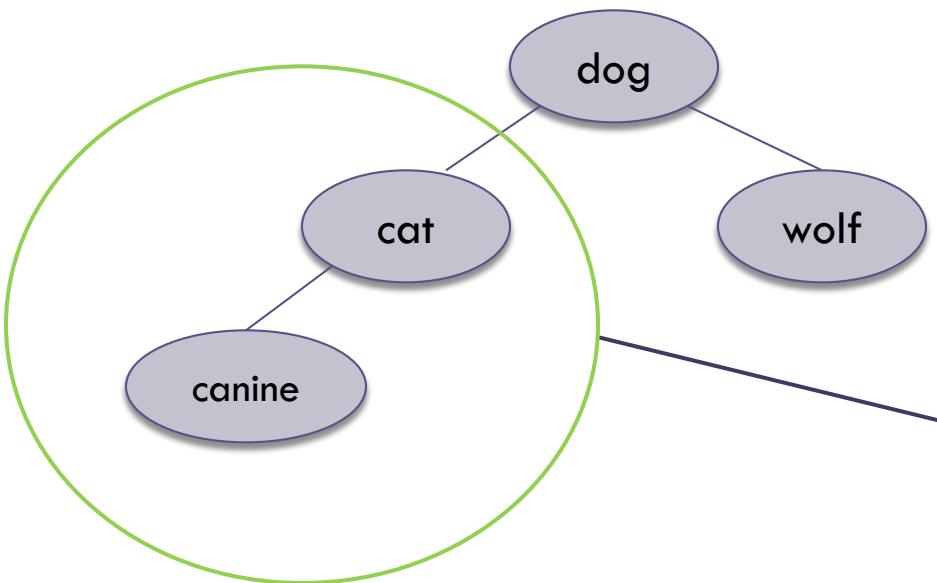


A generalization of the parent-child relationship is the *ancestor-descendant relationship*

# Tree Terminology (cont.)

18

A tree consists of a collection of elements or nodes, with each node linked to its successors

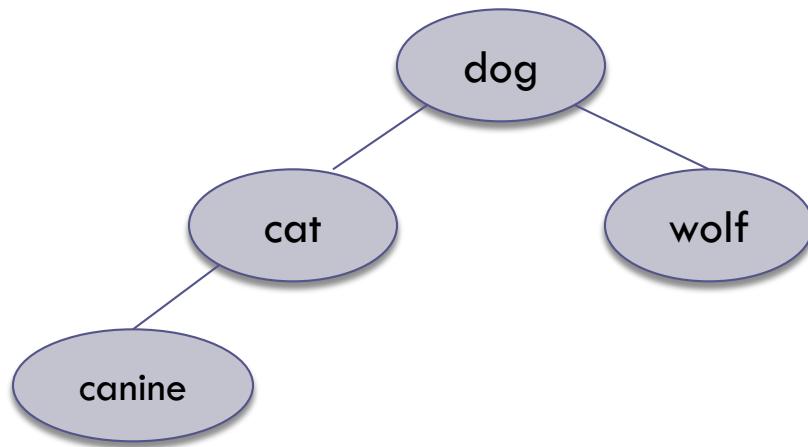


A *subtree* of a node is a tree whose root is a child of that node

# Tree Terminology (cont.)

19

A tree consists of a collection of elements or nodes, with each node linked to its successors

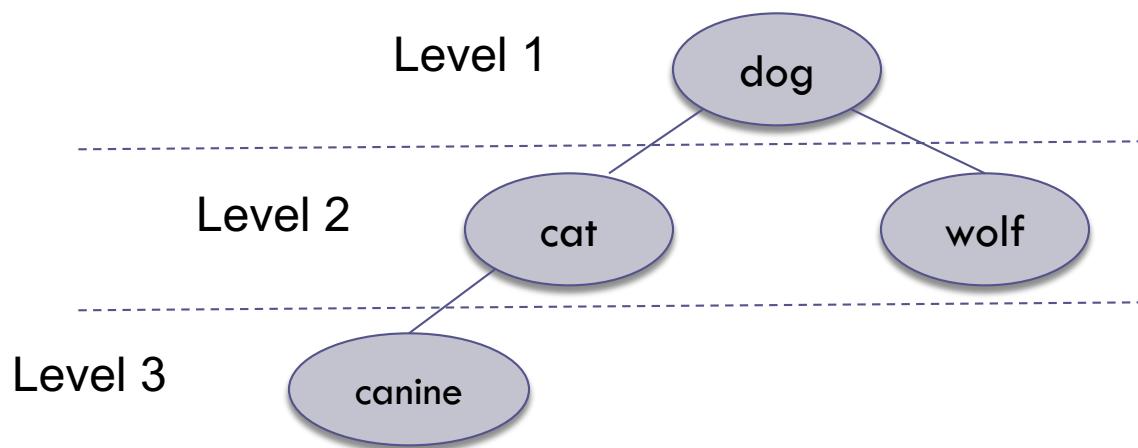


The *level* of a node is a measure of its distance from the root

# Tree Terminology (cont.)

20

A tree consists of a collection of elements or nodes, with each node linked to its successors

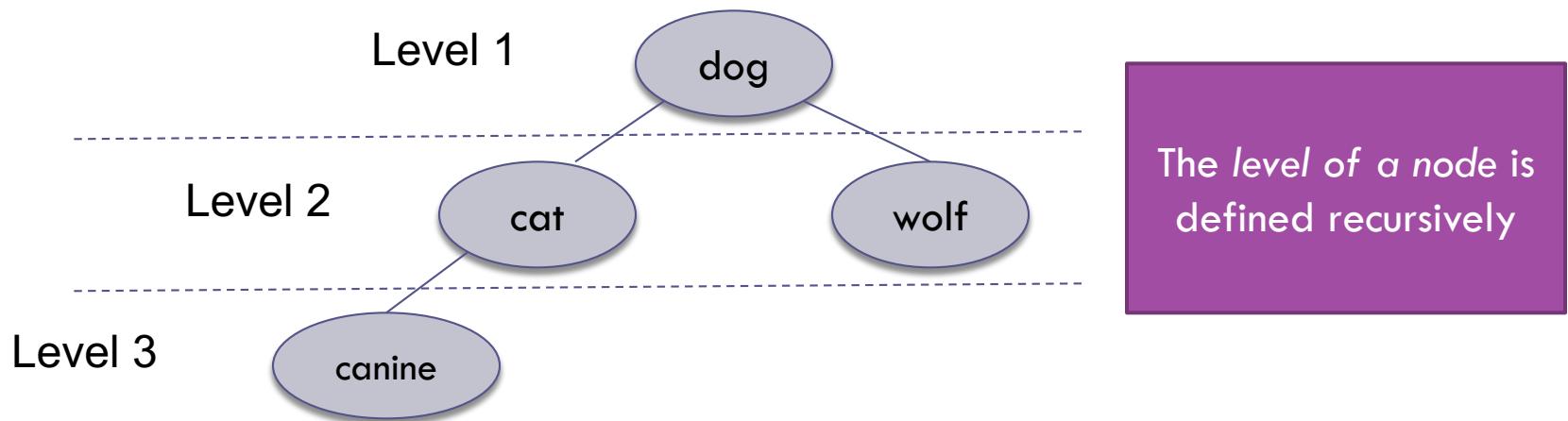


The *level* of a node is a measure of its distance from the root plus 1 and is defined recursively

# Tree Terminology (cont.)

21

A tree consists of a collection of elements or nodes, with each node linked to its successors



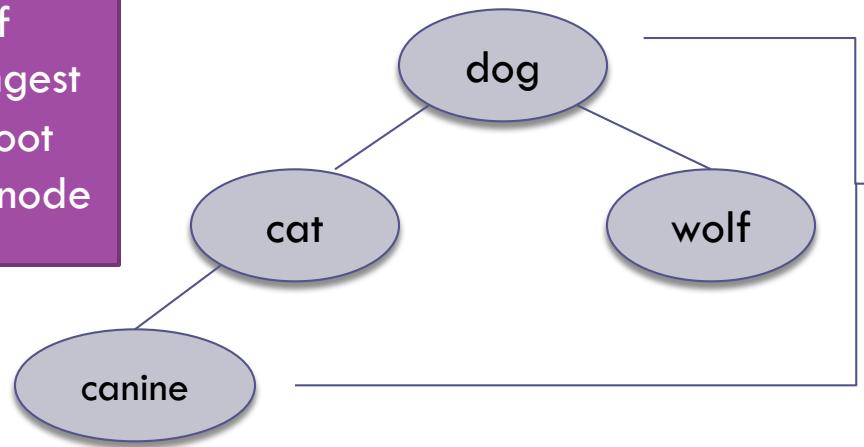
- If node  $n$  is the root of tree  $T$ , its level is 1
- If node  $n$  is not the root of tree  $T$ , its level is  $1 + \text{the level of its parent}$

# Tree Terminology (cont.)

22

A tree consists of a collection of elements or nodes, with each node linked to its successors

The *height of a tree* is the number of nodes in the longest path from the root node to a leaf node



The height of this tree is 3

# Binary Trees

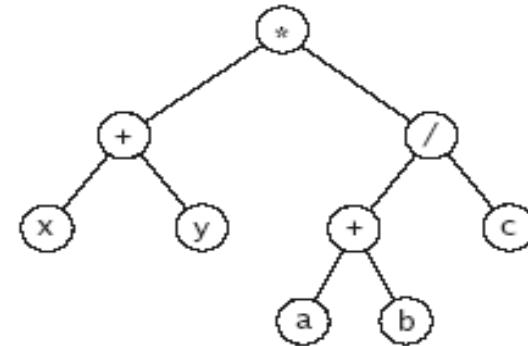
23

- In a binary tree, each node has two subtrees
- A set of nodes  $T$  is a binary tree if either of the following is true
  - $T$  is empty
  - Its root node has two subtrees,  $T_L$  and  $T_R$ , such that  $T_L$  and  $T_R$  are binary trees  
( $T_L$  = left subtree;  $T_R$  = right subtree)

# Expression Tree

24

- Each node contains an operator or an operand
- Operands are stored in leaf nodes
- Parentheses are not stored in the tree because the tree structure dictates the order of operand evaluation
- Operators in nodes at higher levels are evaluated after operators in nodes at lower levels



$(x + y) * ((a + b) / c)$

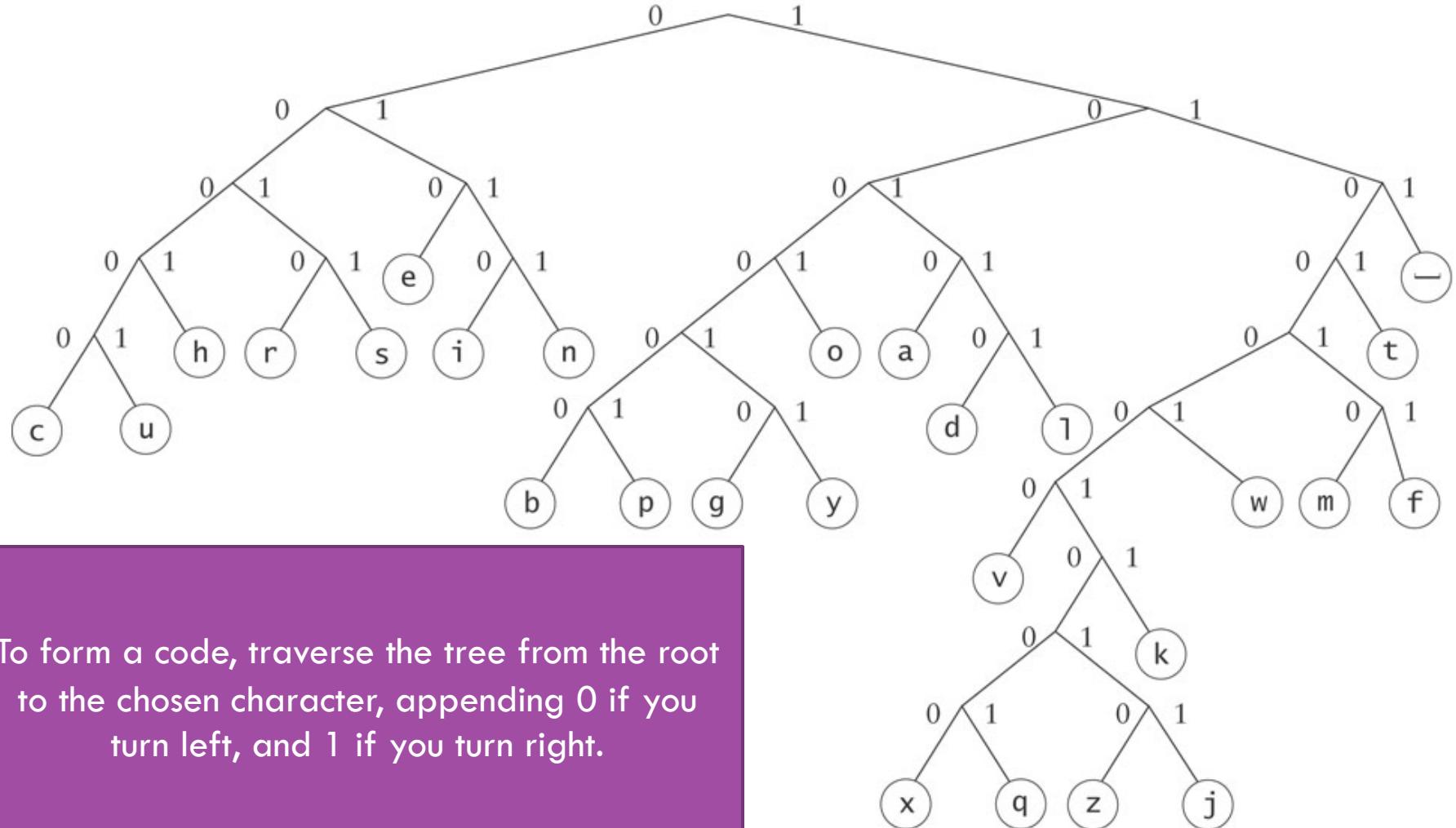
# Huffman Tree

25

- A *Huffman tree* represents *Huffman codes* for characters that might appear in a text file
- As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters; more common characters use fewer bits
- Many programs that compress files use Huffman codes

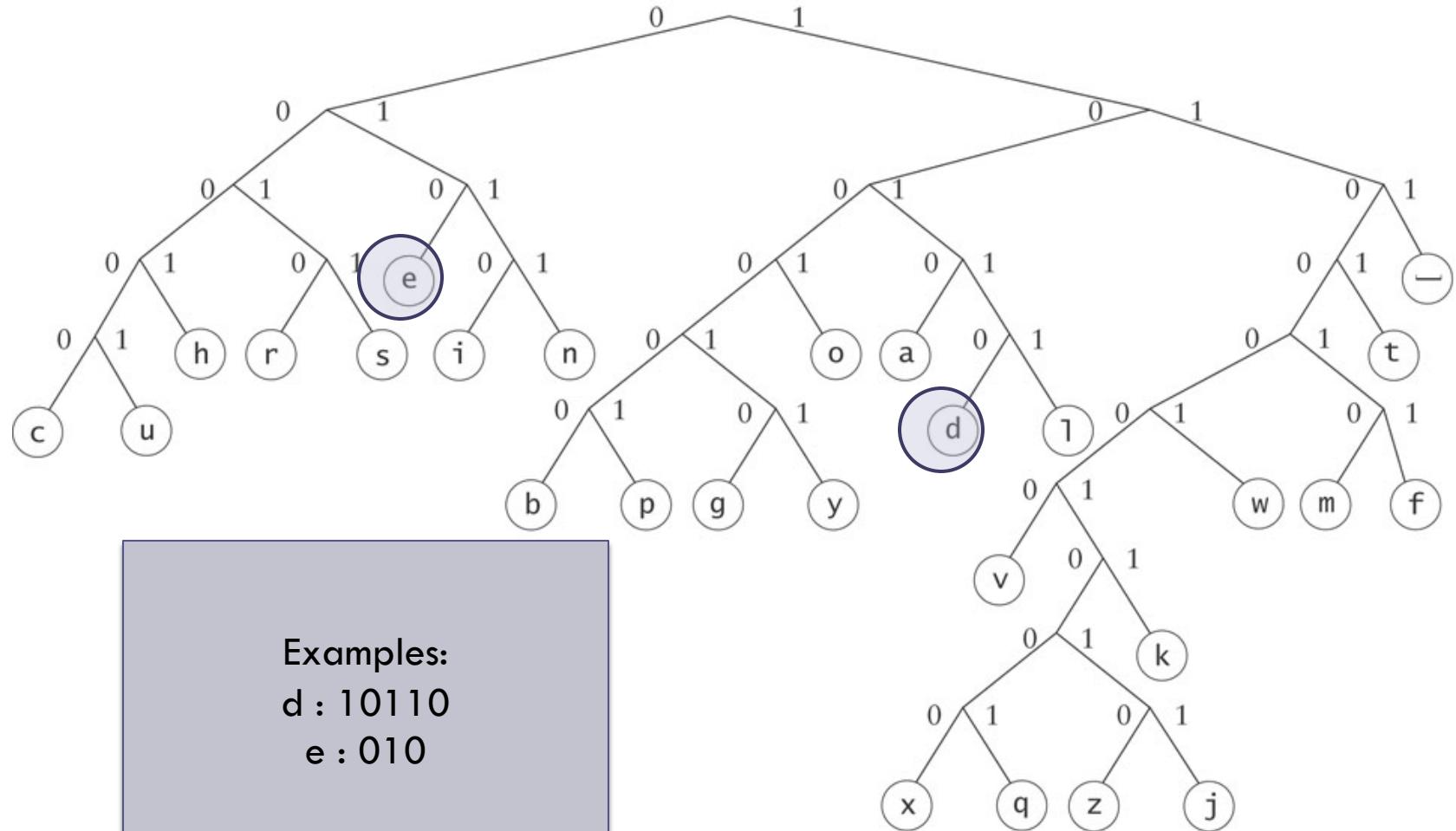
# Huffman Tree (cont.)

26



# Huffman Tree (cont.)

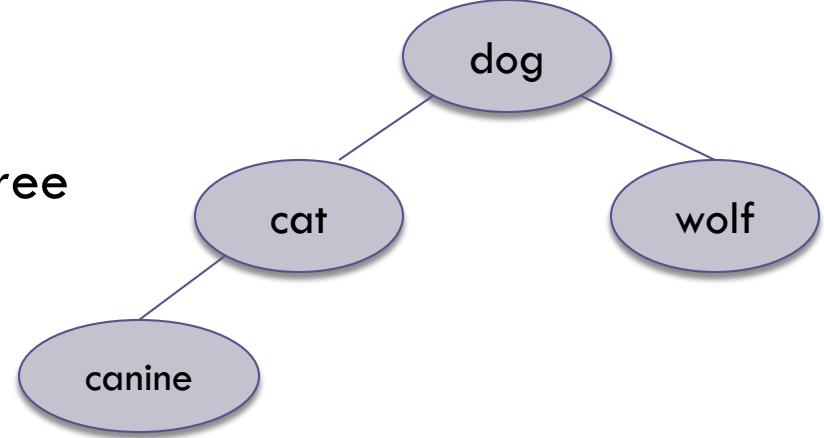
27



# Binary Search Tree

28

- Binary search trees
  - ▣ All elements in the left subtree precede those in the right subtree
- A formal definition:



A set of nodes  $T$  is a binary search tree if either of the following is true:

- $T$  is empty
- If  $T$  is not empty, its root node has two subtrees,  $T_L$  and  $T_R$ , such that  $T_L$  and  $T_R$  are binary search trees and the values in the root node of  $T$  is greater than all values in  $T_L$  and is less than all values in  $T_R$

# Binary Search Tree (cont.)

29

- A binary search tree never has to be sorted because its elements always satisfy the required order relations
- When new elements are inserted (or removed) properly, the binary search tree maintains its order
- In contrast, an array must be expanded whenever new elements are added, and compacted when elements are removed—expanding and contracting are both  $O(n)$

# Binary Search Tree (cont.)

30

- When searching a BST, each probe has the potential to eliminate half the elements in the tree, so searching can be  $O(\log n)$
- In the worst case, searching is  $O(n)$

# Recursive Algorithm for Searching a Binary Tree

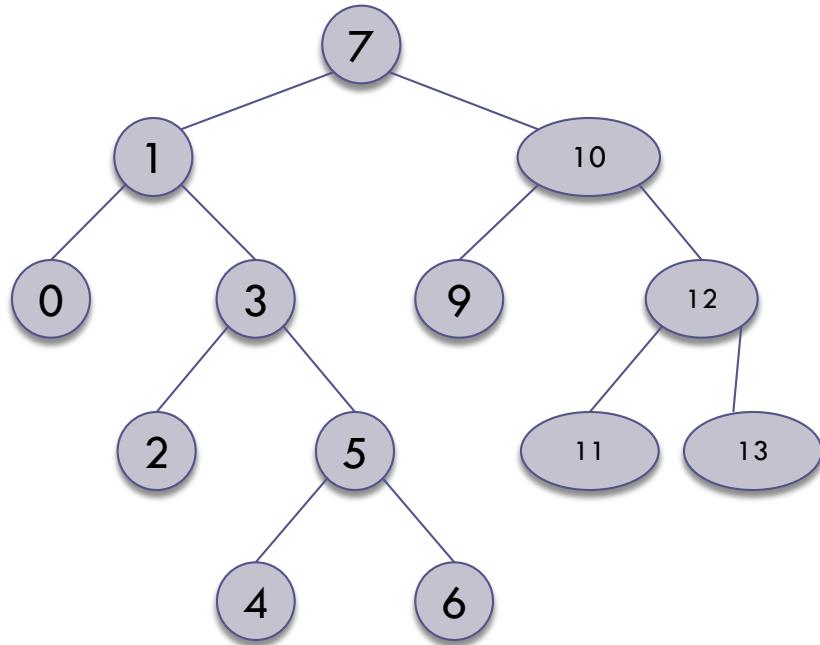
31

1. **if** the tree is empty
2.       return null (*target is not found*)
3.       **else if** the target matches the root node's data
4.       return the data stored at the root node
5.       **else if** the target is less than the root node's data
6.       return the result of searching the left subtree of the root
7.       **else**
8.       return the result of searching the right subtree of the root

# Full, Perfect, and Complete Binary Trees

32

- A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)

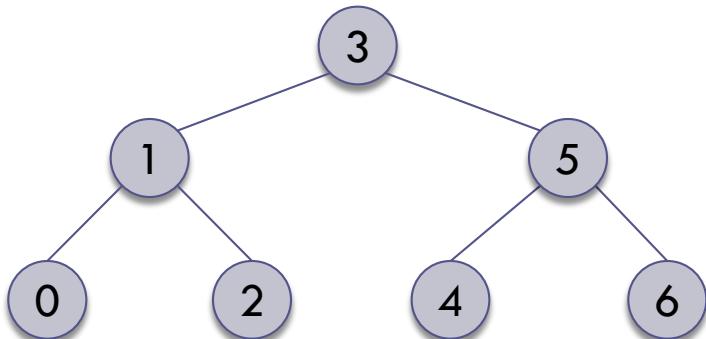


# Full, Perfect, and Complete Binary Trees

(cont.)

33

- A *perfect binary tree* is a full binary tree of height  $n$  with exactly  $2^n - 1$  nodes
- In this case,  $n = 3$  and  $2^n - 1 = 7$

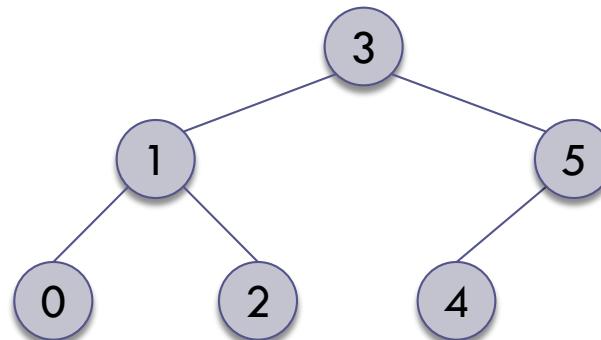


# Full, Perfect, and Complete Binary Trees

(cont.)

34

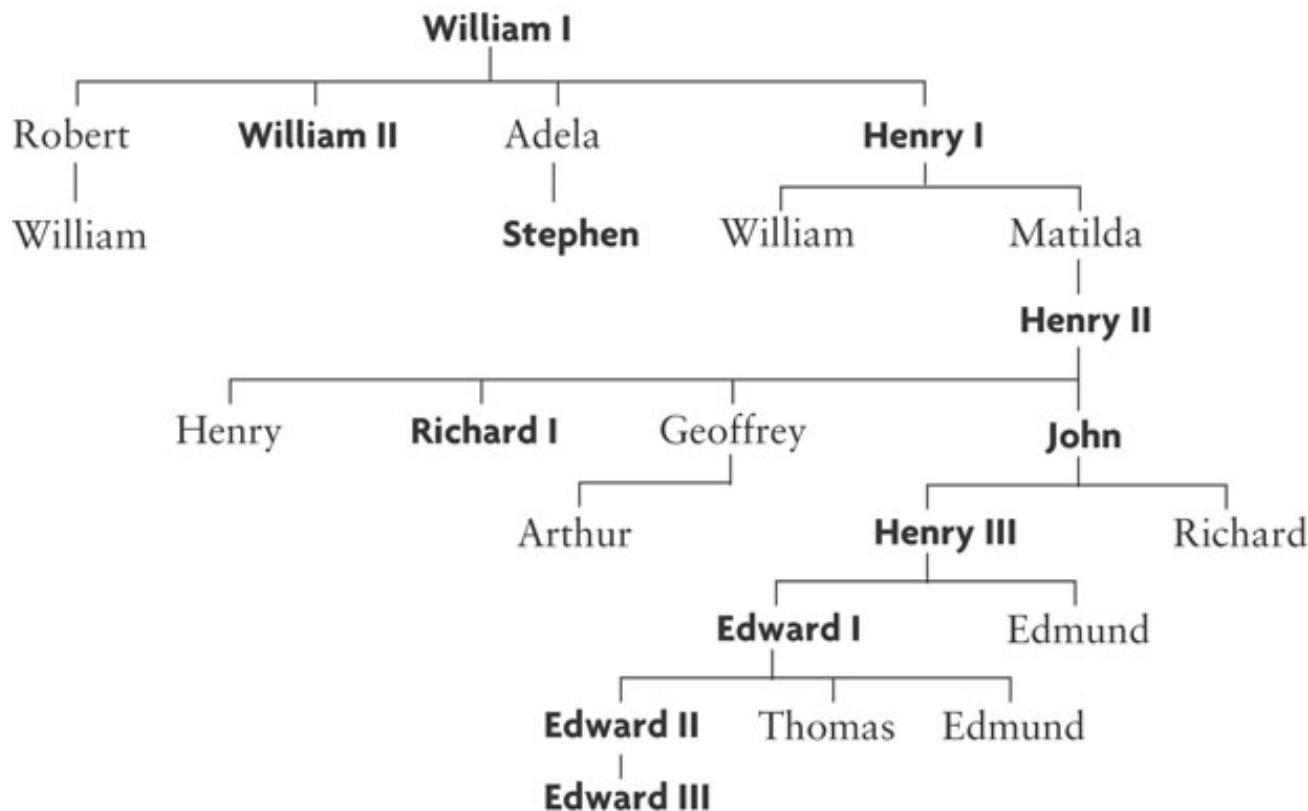
- A *complete binary tree* is a perfect binary tree through level  $n - 1$  with some extra leaf nodes at level  $n$  (the tree height), all toward the left



# General Trees

35

- We do not discuss general trees in this chapter, but nodes of a general tree can have any number of subtrees



# Tree Traversals

## Section 6.2

# Tree Traversals

37

- Often we want to determine the nodes of a tree and their relationship
  - ▣ We can do this by walking through the tree in a prescribed order and visiting the nodes as they are encountered
  - ▣ This process is called *tree traversal*
- Three common kinds of tree traversal
  - ▣ Inorder
  - ▣ Preorder
  - ▣ Postorder

# Tree Traversals (cont.)

38

- Preorder: visit root node, traverse  $T_L$ , traverse  $T_R$
- Inorder: traverse  $T_L$ , visit root node, traverse  $T_R$
- Postorder: traverse  $T_L$ , traverse  $T_R$ , visit root node

## Algorithm for Preorder Traversal

1. if the tree is empty
2.     Return.
- else
3.     Visit the root.
4.     Preorder traverse the left subtree.
5.     Preorder traverse the right subtree.

## Algorithm for Inorder Traversal

1. if the tree is empty
2.     Return.
- else
3.     Inorder traverse the left subtree.
4.     Visit the root.
5.     Inorder traverse the right subtree.

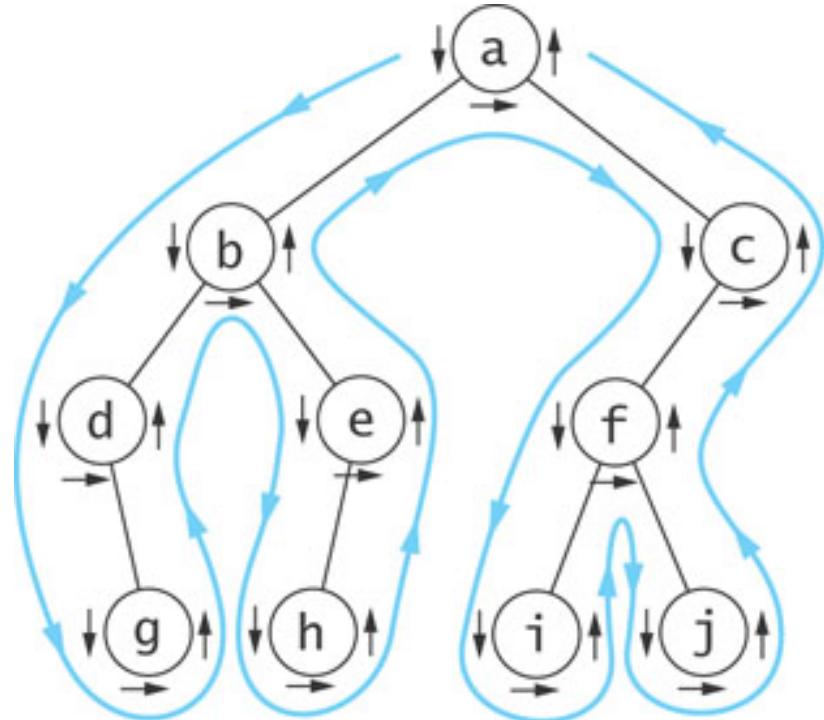
## Algorithm for Postorder Traversal

1. if the tree is empty
2.     Return.
- else
3.     Postorder traverse the left subtree.
4.     Postorder traverse the right subtree.
5.     Visit the root.

# Visualizing Tree Traversals

39

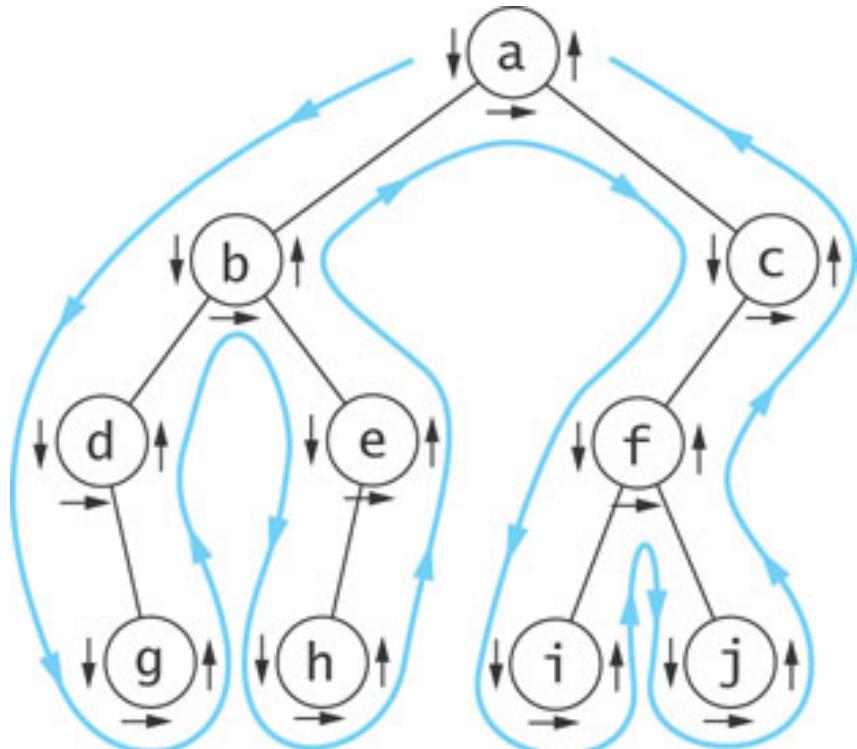
- You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree
- If the mouse always keeps the tree to the left, it will trace a route known as the *Euler tour*
- The Euler tour is the path traced in blue in the figure on the right



# Visualizing Tree Traversals (cont.)

40

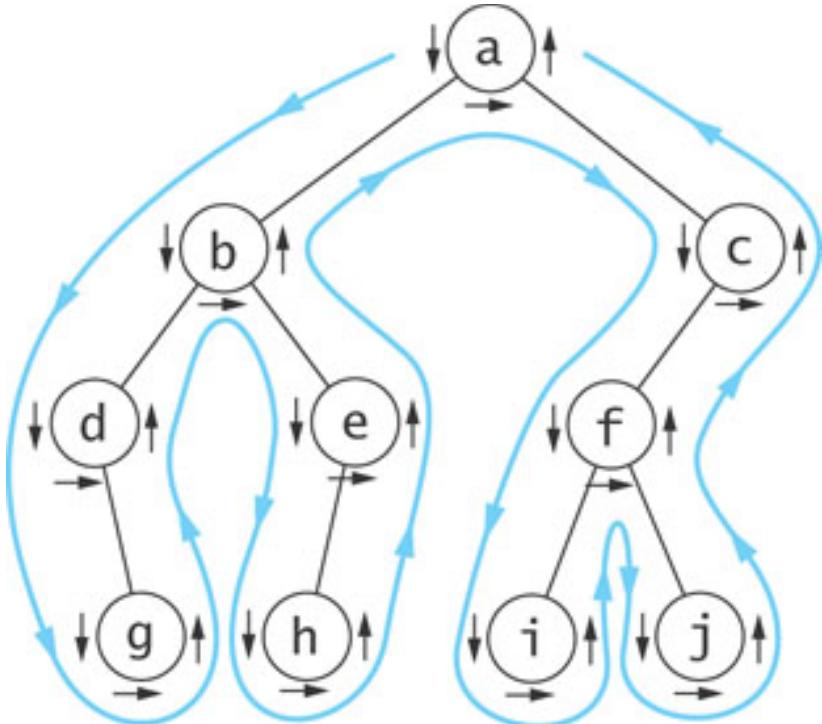
- An Euler tour (blue path) is a preorder traversal
- The sequence in this example is  
a b d g e h c f i j
- The mouse visits each node before traversing its subtrees (shown by the downward pointing arrows)



# Visualizing Tree Traversals (cont.)

41

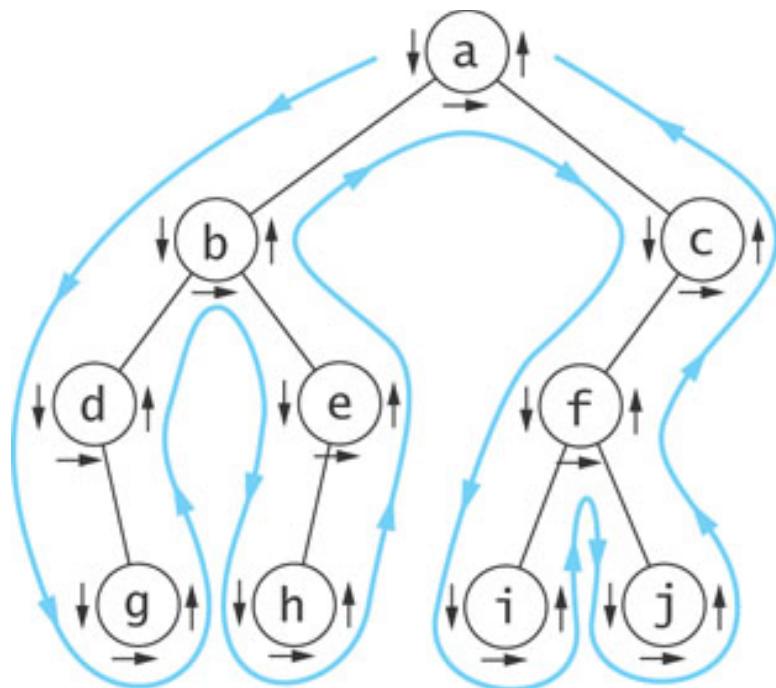
- If we record a node as the mouse returns from traversing its left subtree (horizontal black arrows in the figure) we get an inorder traversal
- The sequence is d g b h e a i f j c



# Visualizing Tree Traversals (cont.)

42

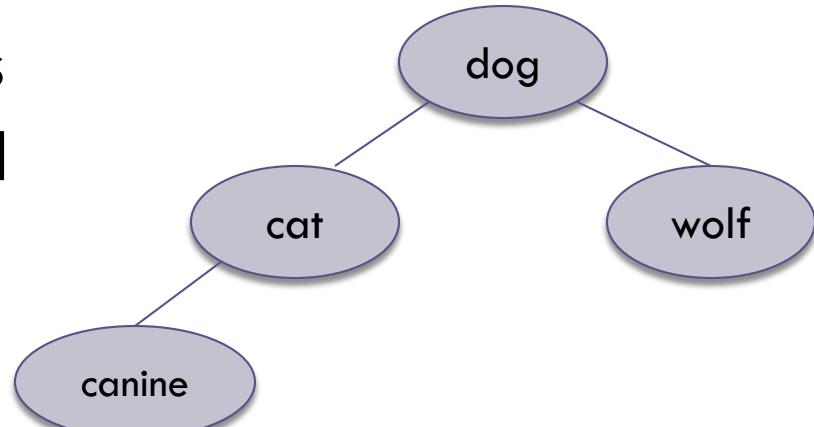
- If we record each node as the mouse last encounters it, we get a postorder traversal (shown by the upward pointing arrows)
- The sequence is g d h e b i j f c a



# Traversals of Binary Search Trees and Expression Trees

43

- An inorder traversal of a binary search tree results in the nodes being visited in sequence by increasing data value



*canine, cat, dog, wolf*

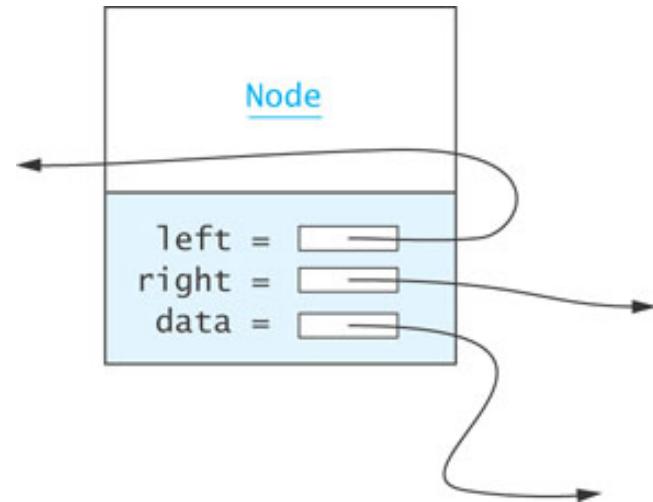
# Implementing a BinaryTree Class

## Section 6.3

# Node<E> Class

45

- Just as for a linked list, a node consists of a data part and links to successor nodes
- The data part is a reference to type E
- A binary tree node must have links to both its left and right subtrees



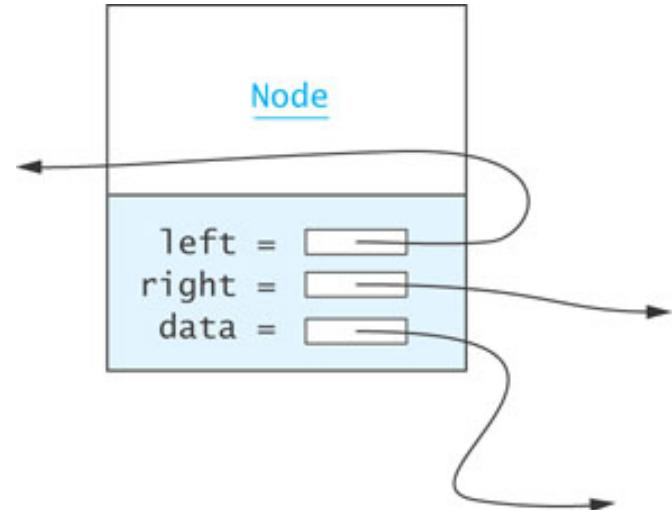
# Node<E> Class (cont.)

46

```
protected static class Node<E>
    implements Serializable {
    protected E data;
    protected Node<E> left;
    protected Node<E> right;

    public Node(E data) {
        this.data = data;
        left = null;
        right = null;
    }

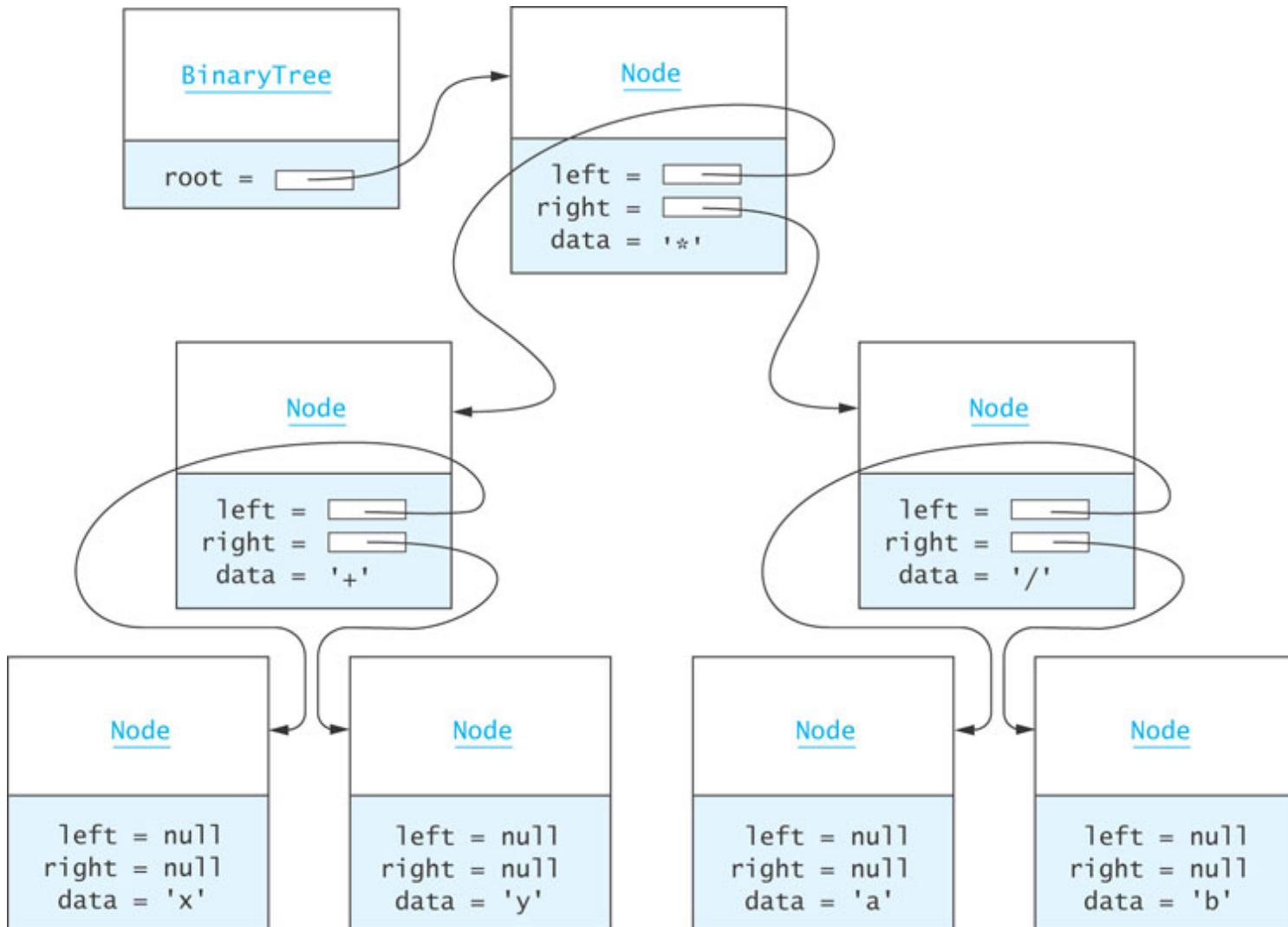
    public String toString() {
        return data.toString();
    }
}
```



Node<E> is declared as an  
inner class within  
BinaryTree<E>

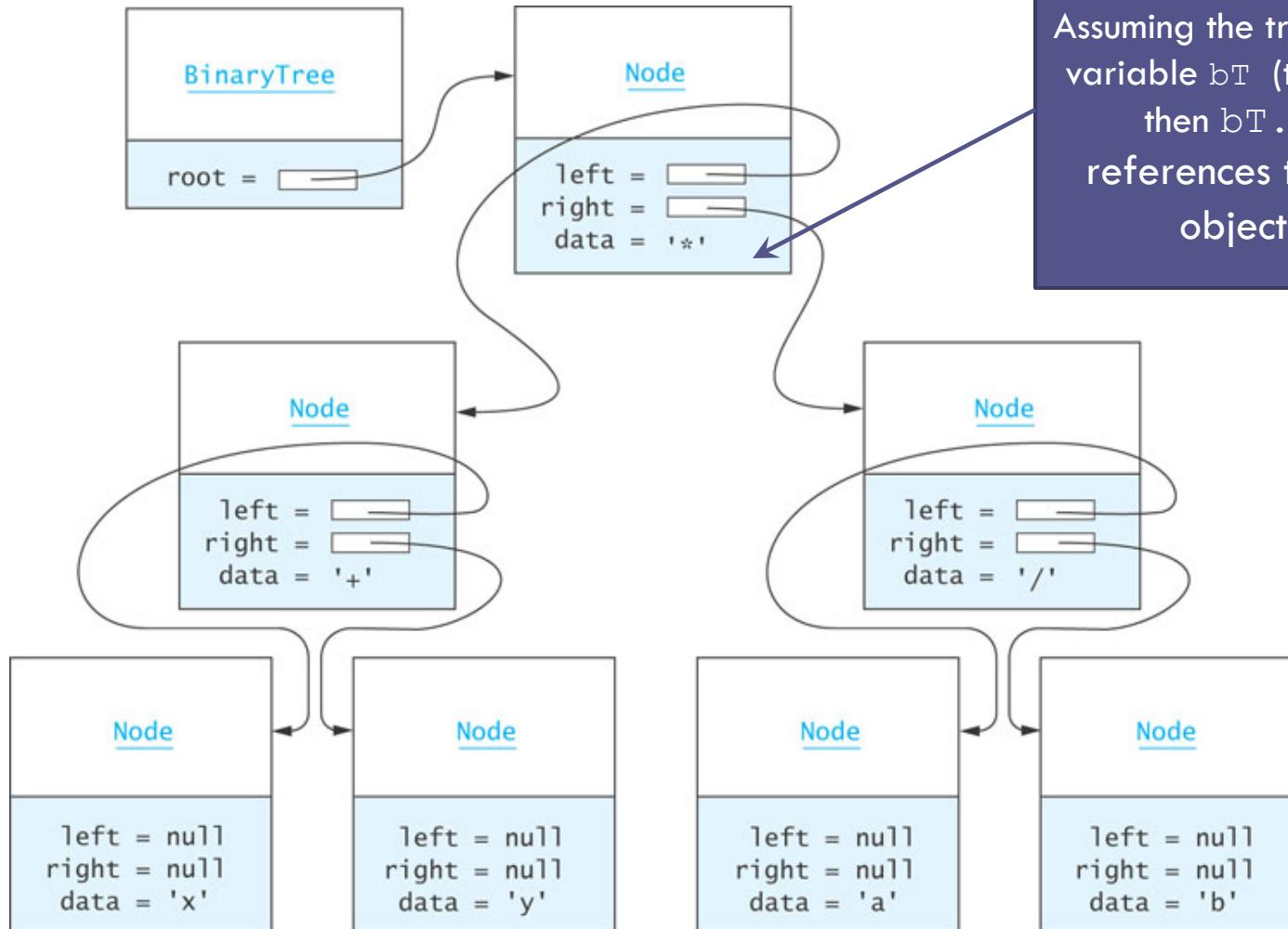
# BinaryTree<E> Class (cont.)

48



# BinaryTree<E> Class (cont.)

49



Assuming the tree is referenced by variable `bT` (type `BinaryTree`)  
then `bT.root.data` references the Character object storing '\*'

# BinaryTree<E> Class (cont.)

50

Data Field	Attribute
<code>protected Node&lt;E&gt; root</code>	Reference to the root of the tree.
Constructor	Behavior
<code>public BinaryTree()</code>	Constructs an empty binary tree.
<code>protected BinaryTree(Node&lt;E&gt; root)</code>	Constructs a binary tree with the given node as the root.
<code>public BinaryTree(E data, BinaryTree&lt;E&gt; leftTree, BinaryTree&lt;E&gt; rightTree)</code>	Constructs a binary tree with the given data at the root and the two given subtrees.
Method	Behavior
<code>public BinaryTree&lt;E&gt; getLeftSubtree()</code>	Returns the left subtree.
<code>public BinaryTree&lt;E&gt; getRightSubtree()</code>	Returns the right subtree.
<code>public E getData()</code>	Returns the data in the root.
<code>public boolean isLeaf()</code>	Returns <b>true</b> if this tree is a leaf, <b>false</b> otherwise.
<code>public String toString()</code>	Returns a <code>String</code> representation of the tree.
<code>private void preOrderTraverse(Node&lt;E&gt; node, int depth, StringBuilder sb)</code>	Performs a preorder traversal of the subtree whose root is <code>node</code> . Appends the representation to the <code>StringBuilder</code> . Increments the value of <code>depth</code> (the current tree level).
<code>public static BinaryTree&lt;E&gt; readBinaryTree(Scanner scan)</code>	Constructs a binary tree by reading its data using <code>Scanner scan</code> .

# BinaryTree<E> **Class** (cont.)

51

## □ Class heading and data field declarations:

```
import java.io.*;  
  
public class BinaryTree<E> implements Serializable {  
    // Insert inner class Node<E> here  
  
    protected Node<E> root;  
  
    . . .  
}
```

# BinaryTree<E> **Class** (cont.)

52

- The Serializable interface defines no methods
- It provides a marker for classes that can be written to a binary file using the ObjectOutputStream and read using the ObjectInputStream

# Constructors

53

- The no-parameter constructor:

```
public BinaryTree() {  
  
    root = null;  
}
```

- The constructor that creates a tree with a given node at the root:

```
protected BinaryTree(Node<E> root) {  
  
    this.root = root;  
}
```

# Constructors (cont.)

54

- The constructor that builds a tree from a data value and two trees:

```
public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E>
    rightTree) {
    root = new Node<E>(data);
    if (leftTree != null) {
        root.left = leftTree.root;
    } else {
        root.left = null;
    }
    if (rightTree != null) {
        root.right = rightTree.root;
    } else {
        root.right = null;
    }
}
```

# getLeftSubtree **and** getRightSubtree **Methods**

55

```
public BinaryTree<E> getLeftSubtree() {  
    if (root != null && root.left != null) {  
        return new BinaryTree<E>(root.left);  
    } else {  
        return null;  
    }  
}
```

- **getRightSubtree method is symmetric**

# isLeaf Method

56

```
public boolean isLeaf() {
```

????

```
}
```

# toString Method

57

- The `toString` method generates a string representing a preorder traversal in which each local root is indented a distance proportional to its depth

```
public String toString() {  
    StringBuilder sb = new StringBuilder();  
    preOrderTraverse(root, 1, sb);  
    return sb.toString();  
}
```

# preOrderTraverse Method

58

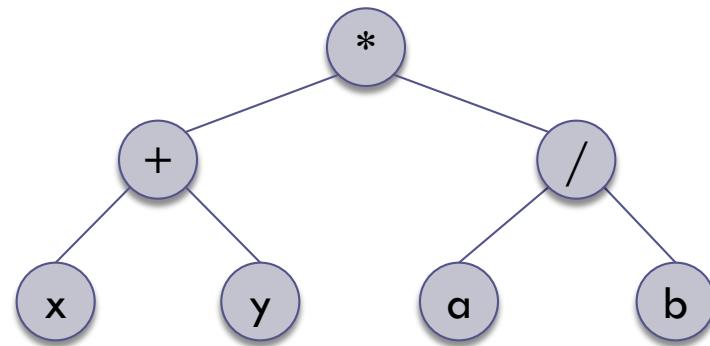
```
private void preOrderTraverse(Node<E> node, int depth,
                           StringBuilder sb) {

    for (int i = 1; i < depth; i++) {
        sb.append("    ");
    }
    if (node == null) {
        sb.append("null\n");
    } else {
        sb.append(node.toString());
        sb.append("\n");
        preOrderTraverse(node.left, depth + 1, sb);
        preOrderTraverse(node.right, depth + 1, sb);
    }
}
```

# preOrderTraverse Method (cont.)

59

```
*  
+  
x  
null  
null  
y  
null  
null  
/  
a  
null  
null  
b  
null  
null
```



$(x + y) * (a / b)$

# Binary Search Trees

## Section 6.5

# Overview of a Binary Search Tree

61

- Recall the definition of a binary search tree:

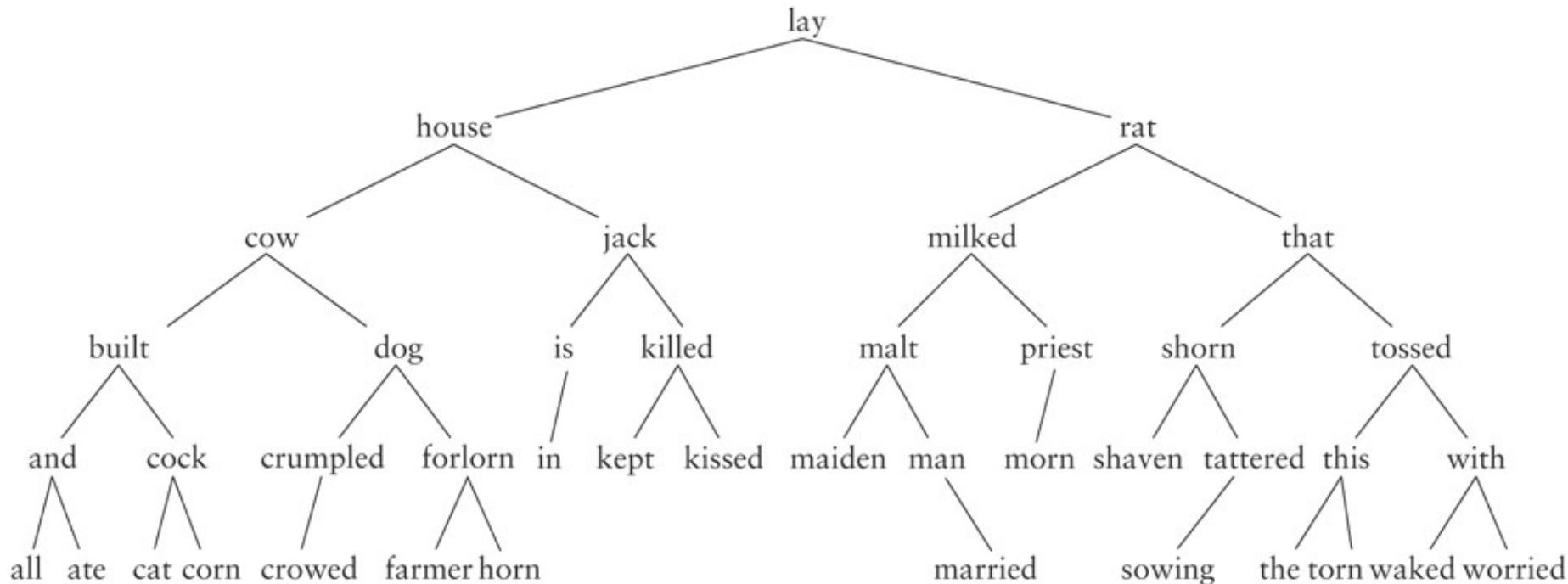
A set of nodes  $T$  is a binary search tree if either of the following is true

  - $T$  is empty
  - If  $T$  is not empty, its root node has two subtrees,  $T_L$  and  $T_R$ , such that  $T_L$  and  $T_R$  are binary search trees and the value in the root node of  $T$  is greater than all values in  $T_L$  and less than all values in  $T_R$

# Overview of a Binary Search Tree

## (cont.)

62



# Recursive Algorithm for Searching a Binary Search Tree

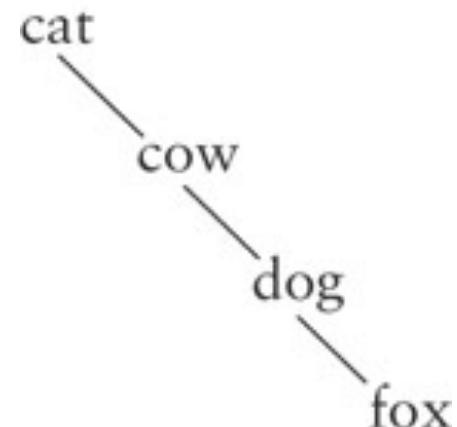
63

1. **if** the root is **null**
  2.     the item is not in the tree; return **null**
  3.     Compare the value of **target** with **root.data**
  4.     **if** they are equal
    5.         the target has been found; return the data at the root
  6.     **else if** the target is less than **root.data**
    6.         return the result of searching the left subtree
  7.     **else**
    7.         return the result of searching the right subtree

# Performance

64

- Search a tree is generally  $O(\log n)$
- If a tree is not very full, performance will be worse
- Searching a tree with only right subtrees, for example, is  $O(n)$



# Interface SearchTree<E>

65

Method	Behavior
boolean add(E item)	Inserts item where it belongs in the tree. Returns <b>true</b> if item is inserted; <b>false</b> if it isn't (already in tree).
boolean contains(E target)	Returns <b>true</b> if target is found in the tree.
E find(E target)	Returns a reference to the data in the node that is equal to target. If no such node is found, returns <b>null</b> .
E delete(E target)	Removes target (if found) from tree and returns it; otherwise, returns <b>null</b> .
boolean remove(E target)	Removes target (if found) from tree and returns <b>true</b> ; otherwise, returns <b>false</b> .

Data Field	Attribute
protected boolean addReturn	Stores a second return value from the recursive add method that indicates whether the item has been inserted.
protected E deleteReturn	Stores a second return value from the recursive delete method that references the item that was stored in the tree.

# Implementing find Methods

---

## LISTING 6.3

BinarySearchTree find Method

```
/** Starter method find.  
 * pre: The target object must implement  
 *       the Comparable interface.  
 * @param target The Comparable object being sought  
 * @return The object, if found, otherwise null  
 */  
public E find(E target) {  
    return find(root, target);  
}  
  
/** Recursive find method.  
 * @param localRoot The local subtree's root  
 * @param target The object being sought  
 * @return The object, if found, otherwise null  
 */  
private E find(Node<E> localRoot, E target) {  
    if (localRoot == null)  
        return null;  
  
    // Compare the target with the data field at the root.  
    int compResult = target.compareTo(localRoot.data);  
    if (compResult == 0)  
        return localRoot.data;  
    else if (compResult < 0)  
        return find(localRoot.left, target);  
    else  
        return find(localRoot.right, target);  
}
```

# Insertion into a Binary Search Tree

67

## Recursive Algorithm for Insertion in a Binary Search Tree

1. **if** the root is `null`
  2. Replace empty tree with a new tree with the item at the root and return `true`.
  3. **else if** the item is equal to `root.data`
    4. The item is already in the tree; return `false`.
  5. **else if** the item is less than `root.data`
    6. Recursively insert the item in the left subtree.
  7. **else**
    8. Recursively insert the item in the right subtree.

# Implementing the add Methods

68

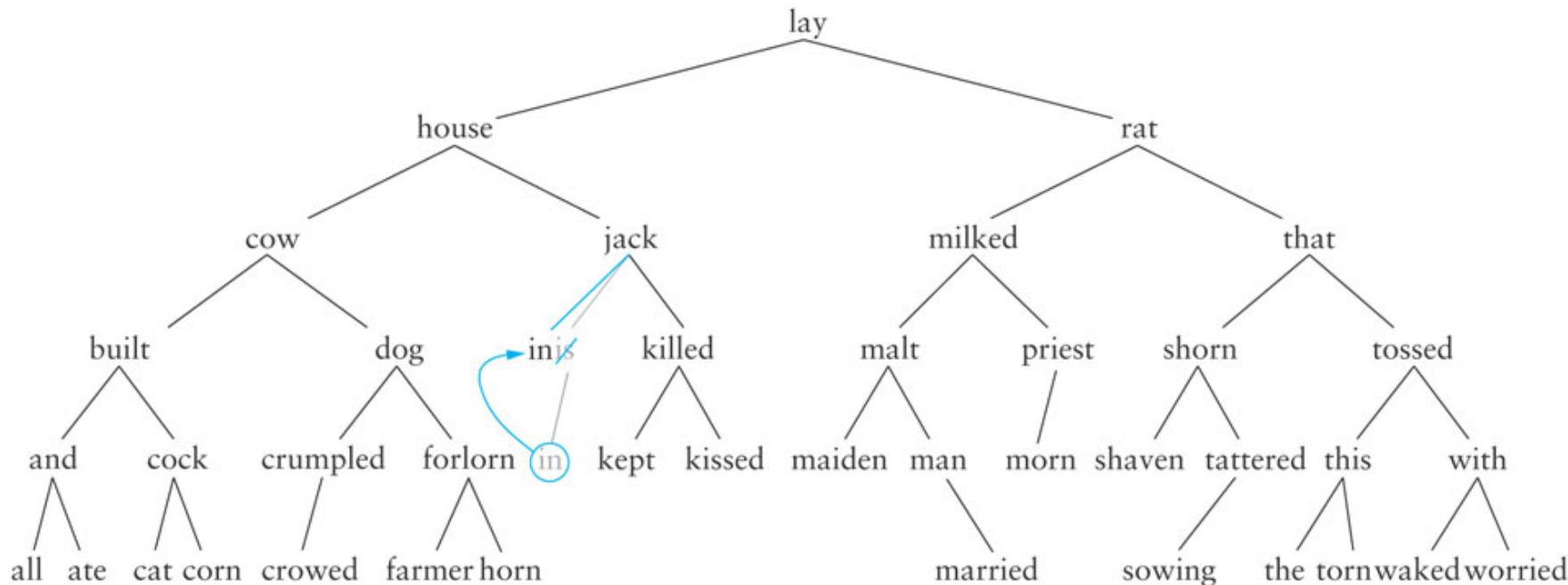
```
/** Starter method add.  
 * pre: The object to insert must implement the  
 * Comparable interface.  
 * @param item The object being inserted  
 * @return true if the object is inserted, false  
 *         if the object already exists in the tree  
 */  
public boolean add(E item) {  
    root = add(root, item);  
    return addReturn;  
}
```

# Implementing the add Methods (cont.)

```
/** Recursive add method.  
post: The data field addReturn is set true if the item is added to  
      the tree, false if the item is already in the tree.  
@param localRoot The local root of the subtree  
@param item The object to be inserted  
@return The new local root that now contains the  
        inserted item  
*/  
private Node<E> add(Node<E> localRoot, E item) {  
    ????????????????????
```

# Removal from a Binary Search Tree

70

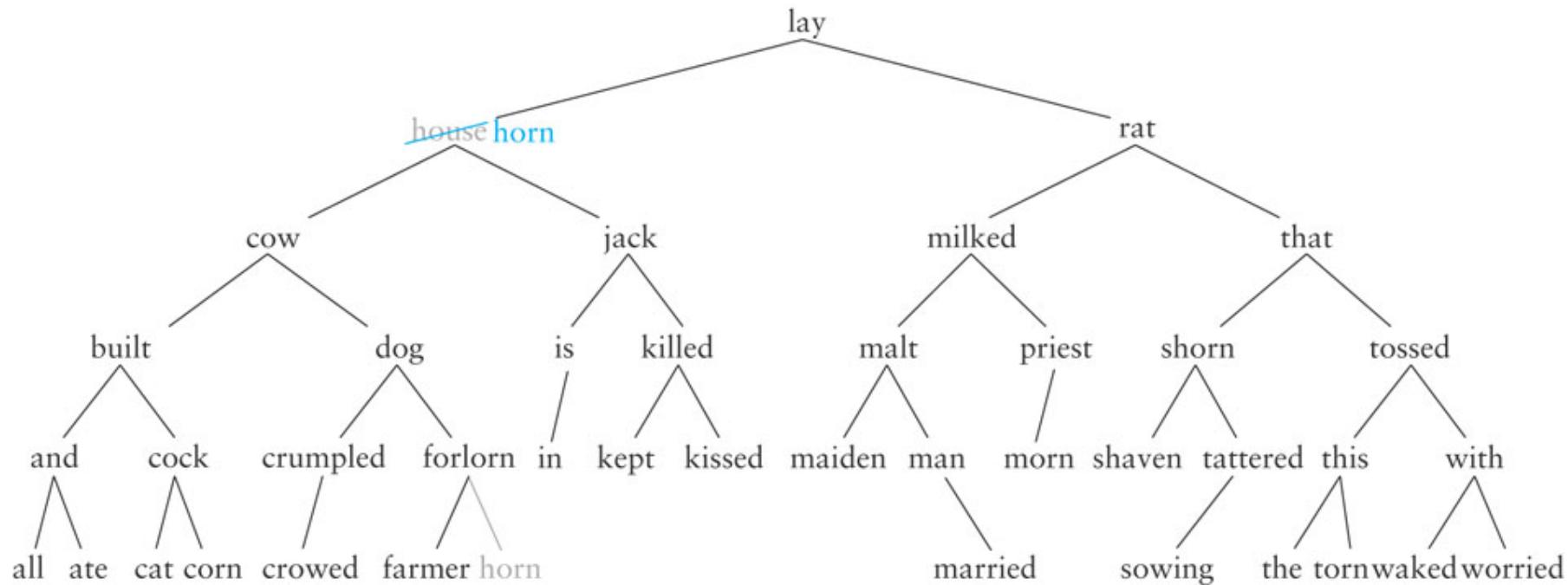


- If the item to be removed has two children, replace it with the largest item in its left subtree – the *inorder predecessor*

# Removing from a Binary Search Tree

(cont.)

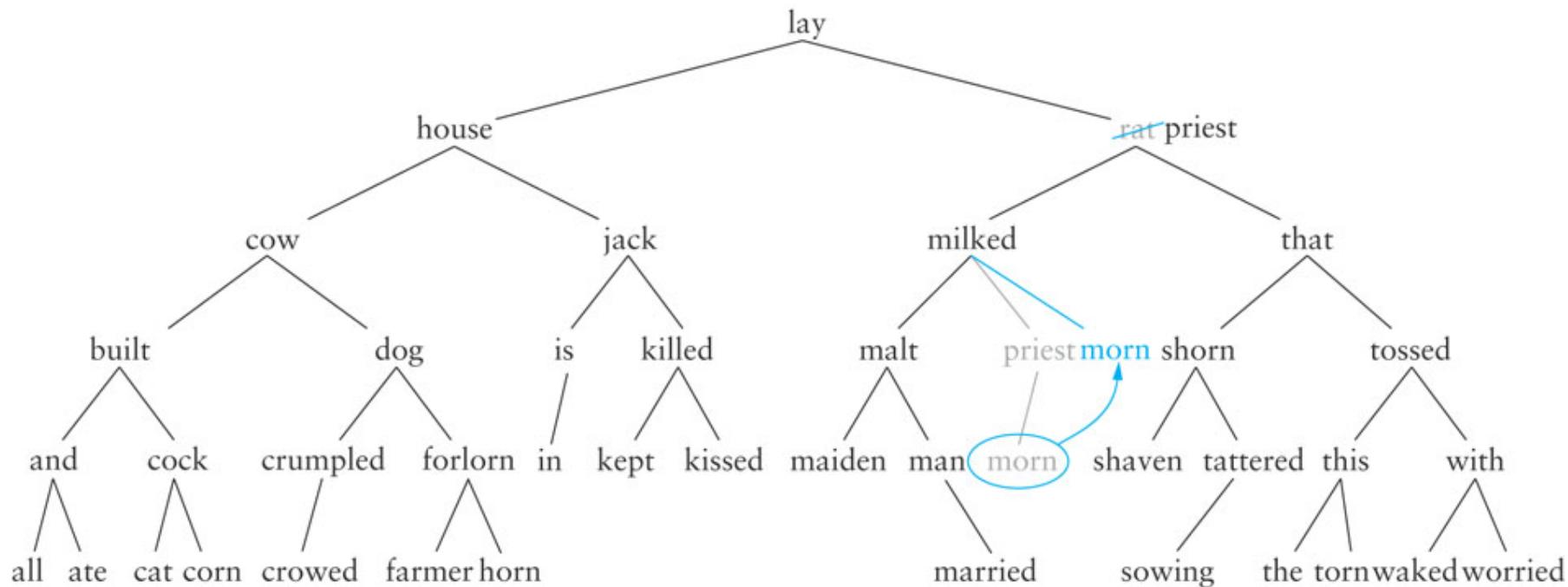
71



# Removing from a Binary Search Tree

(cont.)

72



# Algorithm for Removing from a Binary Search Tree

73

## Recursive Algorithm for Removal from a Binary Search Tree

1. if the root is null  
The item is not in tree – return `null`.
3. Compare the item to the data at the local root.
4. if the item is less than the data at the local root  
Return the result of deleting from the left subtree.
6. else if the item is greater than the local root  
Return the result of deleting from the right subtree.
8. else // *The item is in the local root*  
Store the data in the local root in `deletedReturn`.
10. if the local root has no children  
Set the parent of the local root to reference `null`.
12. else if the local root has one child  
Set the parent of the local root to reference that child.
14. else // *Find the inorder predecessor*  
if the left child has no right child it is the inorder predecessor  
Set the parent of the local root to reference the left child.
18. else  
Find the rightmost node in the right subtree of the left child.  
Copy its data into the local root's data and remove it by setting its parent to reference its left child.

# Implementing the delete Method

74

```
/** Starter method delete.  
 * post: The object is not in the tree.  
 * @param target The object to be deleted  
 * @return The object deleted from the tree  
 *         or null if the object was not in the tree  
 * @throws ClassCastException if target does not implement  
 *         Comparable  
 */  
  
public E delete(E target) {  
    root = delete(root, target);  
    return deleteReturn;  
}
```

# Implementing the delete Method

75

```
/** Recursive delete method.  
 post: The item is not in the tree;  
       deleteReturn is equal to the deleted item  
       as it was stored in the tree or null  
       if the item was not found.  
 @param localRoot The root of the current subtree  
 @param item The item to be deleted  
 @return The modified local root that does not contain  
         the item  
 */  
private Node < E > delete(Node < E > localRoot, E item) {
```

?????????????????



# CS 570: Data Structures

## Trees (Part 2)

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# Week 11

---

- Reading Assignment: Koffman and Wolfgang,  
Section 6.6

# Implementing the add Methods

3

```
/** Starter method add.  
 * pre: The object to insert must implement the  
 * Comparable interface.  
 * @param item The object being inserted  
 * @return true if the object is inserted, false  
 *         if the object already exists in the tree  
 */  
public boolean add(E item) {  
    root = add(root, item);  
    return addReturn;  
}
```

# Implementing the add Methods (cont.)

```
/** Recursive add method.  
post: The data field addReturn is set true if the item is added to  
      the tree, false if the item is already in the tree.  
@param localRoot The local root of the subtree  
@param item The object to be inserted  
@return The new local root that now contains the  
        inserted item  
*/  
private Node<E> add(Node<E> localRoot, E item) {  
    if (localRoot == null) {  
        // item is not in the tree - insert it.  
        addReturn = true;  
        return new Node<E>(item);  
    } else if (item.compareTo(localRoot.data) == 0) {  
        // item is equal to localRoot.data  
        addReturn = false;  
        return localRoot;  
    } else if (item.compareTo(localRoot.data) < 0) {  
        // item is less than localRoot.data  
        localRoot.left = add(localRoot.left, item);  
        return localRoot;  
    } else {  
        // item is greater than localRoot.data  
        localRoot.right = add(localRoot.right, item);  
        return localRoot;  
    }  
}
```

# Implementing the delete Method - cont. from Notes part 9

5

```
/** Recursive delete method.  
 post: The item is not in the tree;  
        deleteReturn is equal to the deleted item  
        as it was stored in the tree or null  
        if the item was not found.  
  
 @param localRoot The root of the current subtree  
 @param item The item to be deleted  
 @return The modified local root that does not contain  
         the item  
 */  
  
private Node < E > delete(Node < E > localRoot, E item) {  
    if (localRoot == null) {  
        // item is not in the tree.  
        deleteReturn = null;  
        return localRoot;  
    }  
}
```

# Implementing the delete Method

6

```
// Search for item to delete.  
int compResult = item.compareTo(localRoot.data);  
if (compResult < 0) {  
    // item is smaller than localRoot.data.  
    localRoot.left = delete(localRoot.left, item);  
    return localRoot;  
}  
else if (compResult > 0) {  
    // item is larger than localRoot.data.  
    localRoot.right = delete(localRoot.right, item);  
    return localRoot;  
}
```

# Implementing the delete Method

7

```
else {  
    // item is at local root.  
    deleteReturn = localRoot.data;  
    if (localRoot.left == null) {  
        // If there is no left child, return right child  
        // which can also be null.  
        return localRoot.right;  
    }  
    else if (localRoot.right == null) {  
        // If there is no right child, return left child.  
        return localRoot.left;  
    }  
}
```

# Implementing the delete Method

8

```
// Node being deleted has 2 children, replace the  
// data with inorder predecessor.  
  
if (localRoot.left.right == null) {  
    // The left child has no right child.  
    // Replace the data with the data in the  
    // left child.  
  
    localRoot.data = localRoot.left.data;  
    // Replace the left child with its left child.  
    localRoot.left = localRoot.left.left;  
  
    return localRoot;  
}
```

# Implementing the delete Method

9

```
else {  
    // Search for the inorder predecessor (ip)  
    //and replace deleted node's data with ip.  
    localRoot.data =  
        findLargestChild(localRoot.left);  
    return localRoot;  
}  
}  
}  
}
```

# Method findLargestChild

10

## LISTING 6.6

BinarySearchTree findLargestChild Method

```
/** Find the node that is the
     inorder predecessor and replace it
     with its left child (if any).
     post: The inorder predecessor is removed from the tree.
     @param parent The parent of possible inorder
               predecessor (ip)
     @return The data in the ip
 */
private E findLargestChild(Node<E> parent) {
    // If the right child has no right child, it is
    // the inorder predecessor.
    if (parent.right.right == null) {
        E returnValue = parent.right.data;
        parent.right = parent.right.left;
        return returnValue;
    } else {
        return findLargestChild(parent.right);
    }
}
```

# Testing a Binary Search Tree

11

- To test a binary search tree, verify that an inorder traversal will display the tree contents in ascending order after a series of insertions and deletions are performed

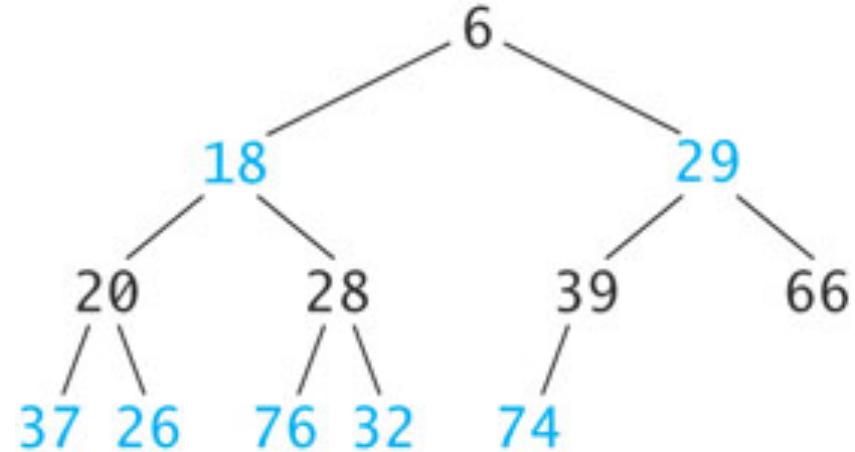
# Heaps and Priority Queues

## Section 6.6

# Heaps and Priority Queues

13

- A heap is a complete binary tree with the following properties
  - The value in the root is the smallest item in the tree
  - Every subtree is a heap

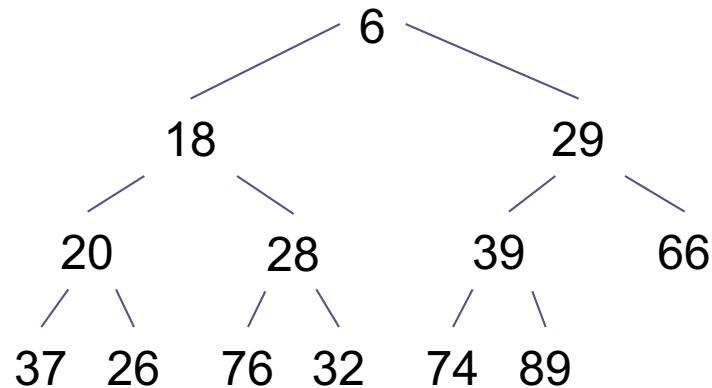


# Inserting an Item into a Heap

14

## Algorithm for Inserting in a Heap

1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3.     Swap the new item with its parent, moving the new item up the heap.

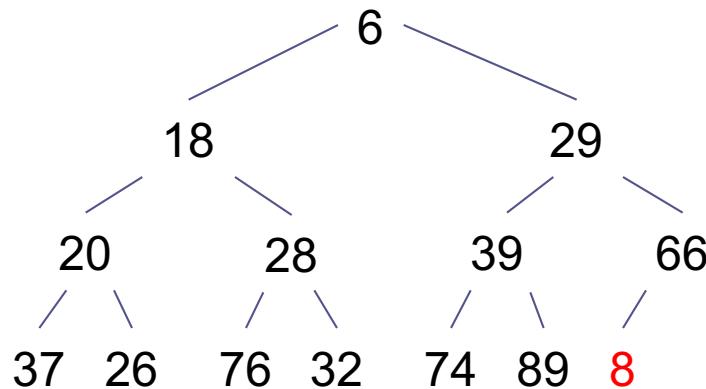


# Inserting an Item into a Heap (cont.)

15

## Algorithm for Inserting in a Heap

1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.

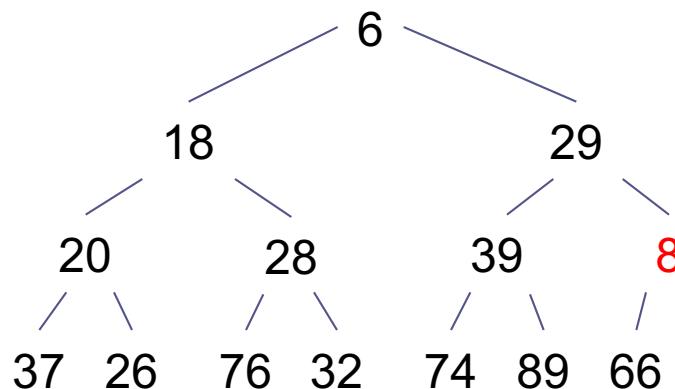


# Inserting an Item into a Heap (cont.)

16

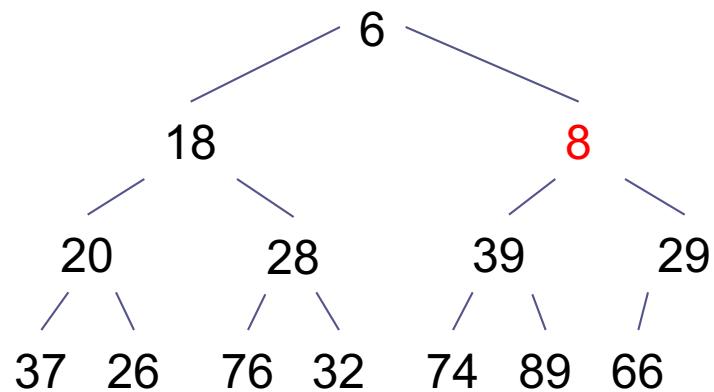
## Algorithm for Inserting in a Heap

1. Insert the new item in the next position at the bottom of the heap.
2. **while** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.



# Inserting an Item into a Heap (cont.)

17

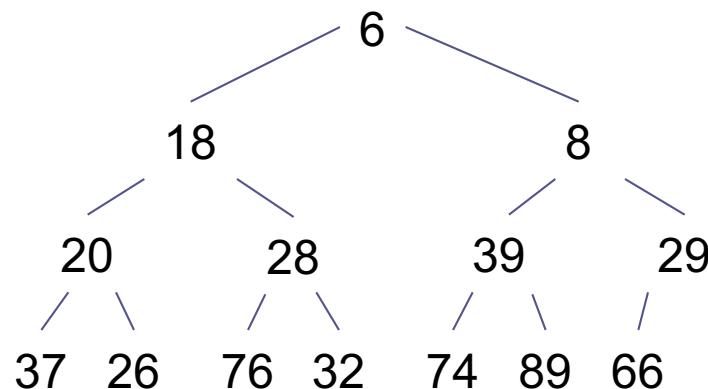


# Removing an Item from a Heap

18

## Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3.       Swap item LIH with its smaller child, moving LIH down the heap.

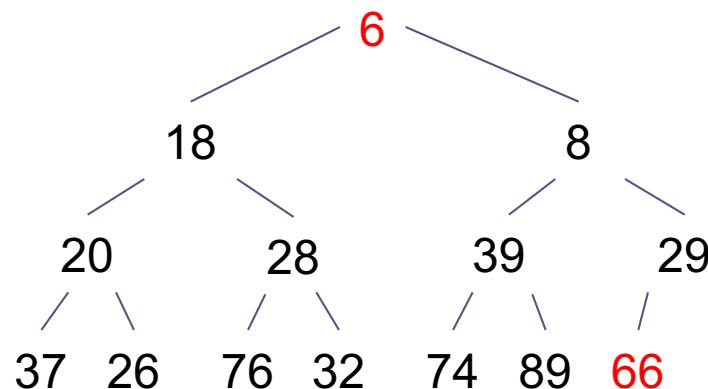


# Removing an Item from a Heap (cont.)

19

## Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3.     Swap item LIH with its smaller child, moving LIH down the heap.

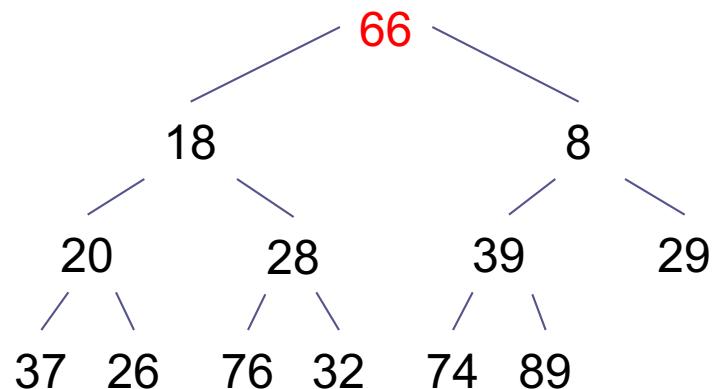


# Removing an Item from a Heap (cont.)

20

## Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3.     Swap item LIH with its smaller child, moving LIH down the heap.

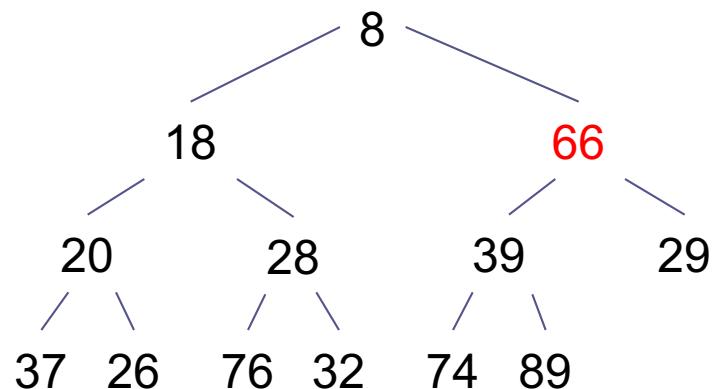


# Removing an Item from a Heap (cont.)

21

## Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3.     Swap item LIH with its smaller child, moving LIH down the heap.

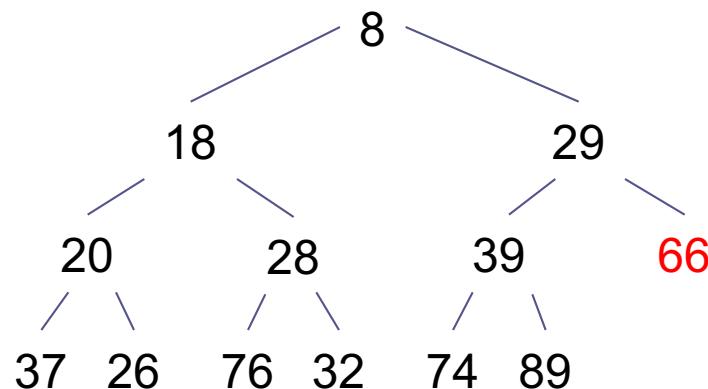


# Removing an Item from a Heap (cont.)

22

## Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. **while** item LIH has children and item LIH is larger than either of its children
3.     Swap item LIH with its smaller child, moving LIH down the heap.



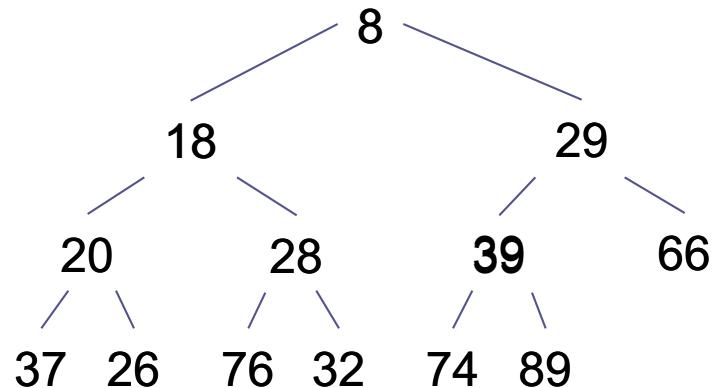
# Implementing a Heap

23

- Because a heap is a complete binary tree, it can be implemented efficiently using an array rather than a linked data structure

# Implementing a Heap (cont.)

24



0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

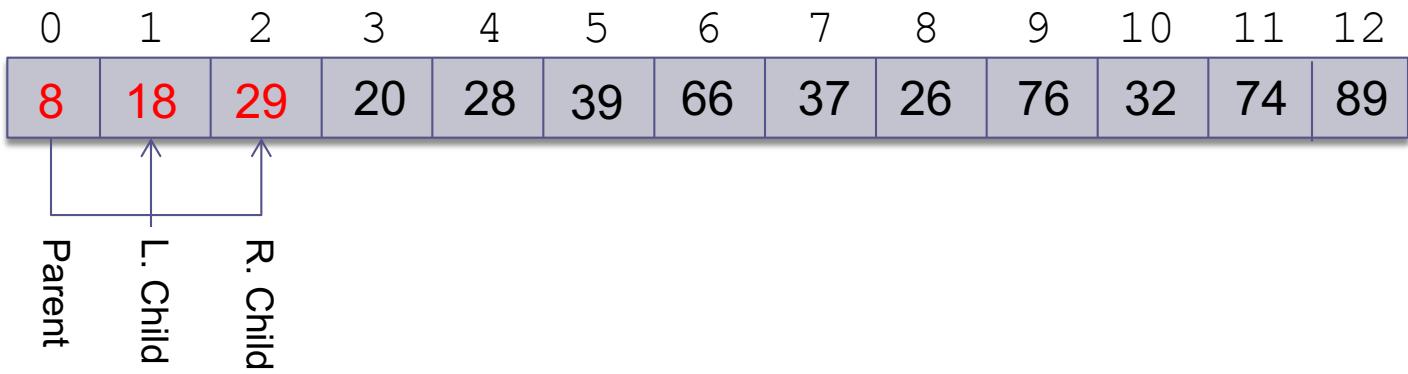
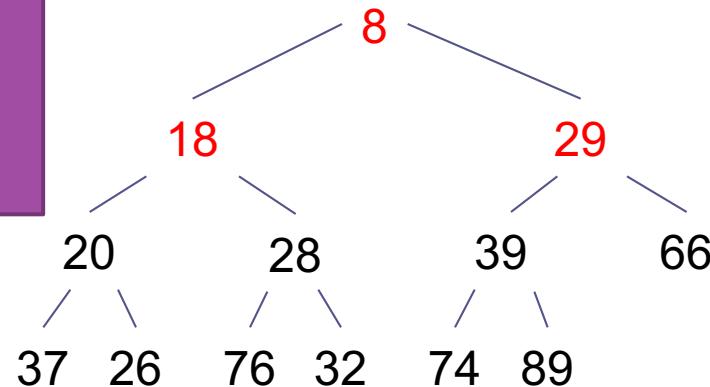
# Implementing a Heap (cont.)

25

For a node at position  $p$ ,

L. child position:  $2p + 1$

R. child position:  $2p + 2$



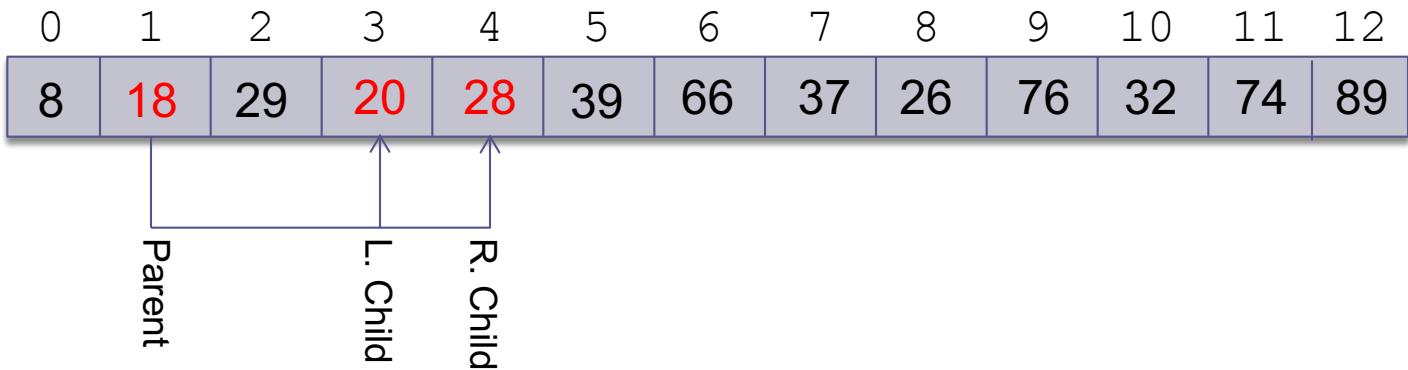
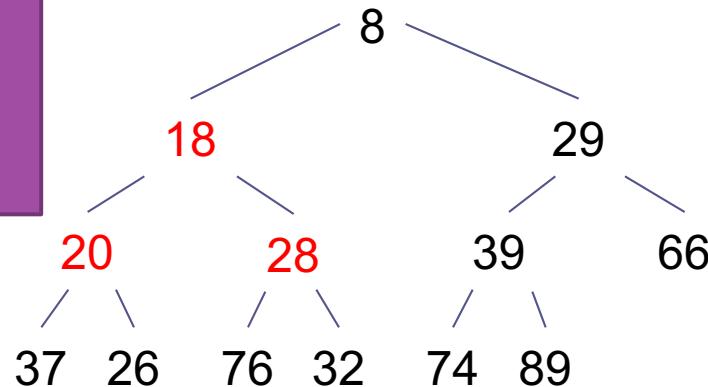
# Implementing a Heap (cont.)

26

For a node at position  $p$ ,

L. child position:  $2p + 1$

R. child position:  $2p + 2$



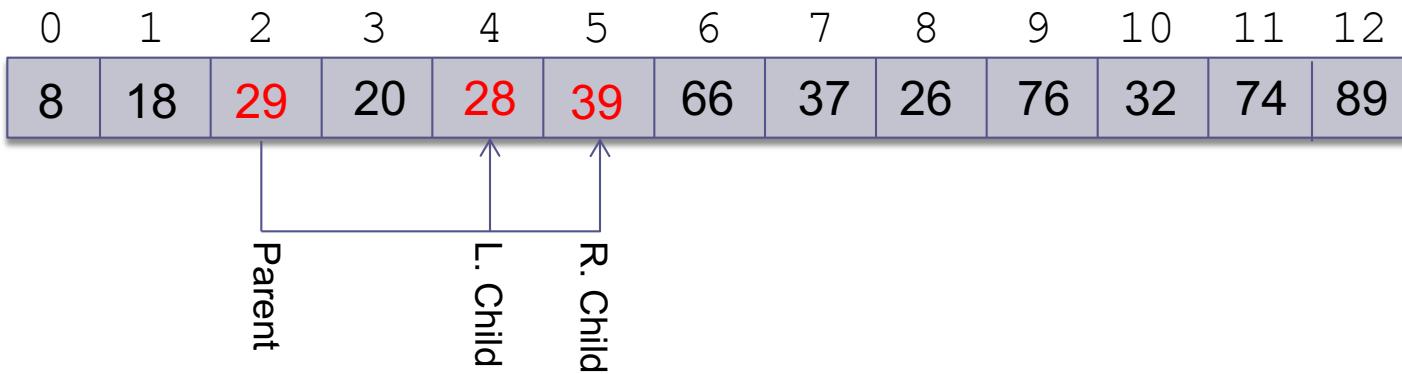
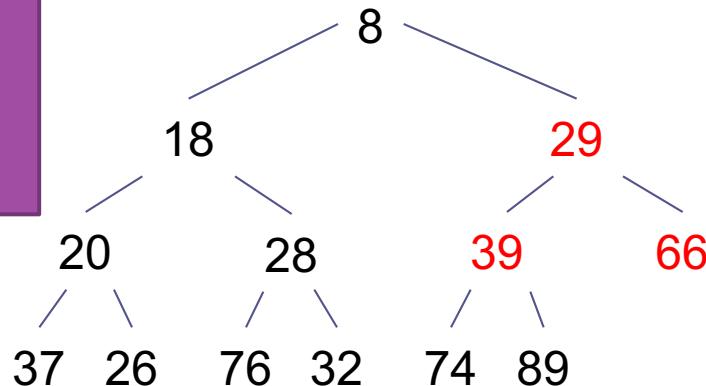
# Implementing a Heap (cont.)

27

For a node at position  $p$ ,

L. child position:  $2p + 1$

R. child position:  $2p + 2$



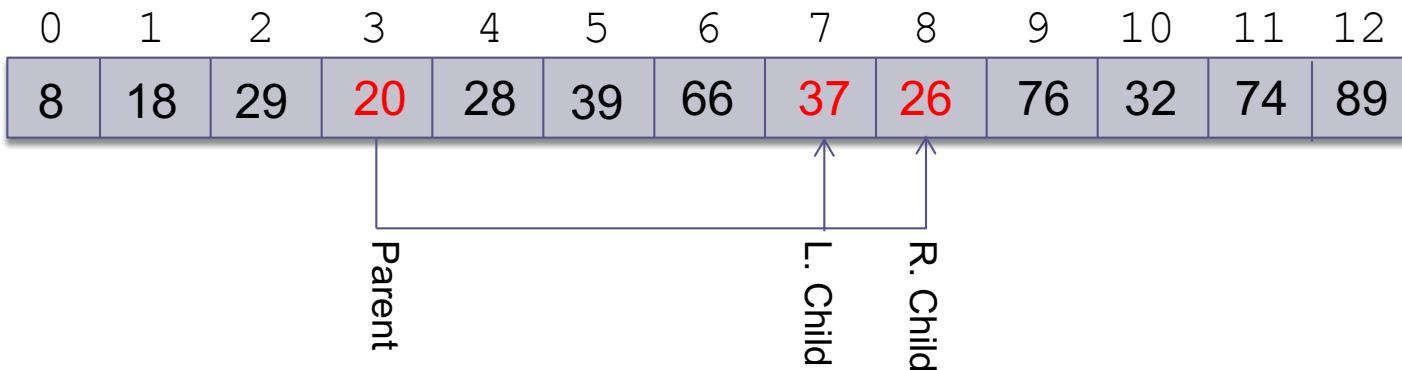
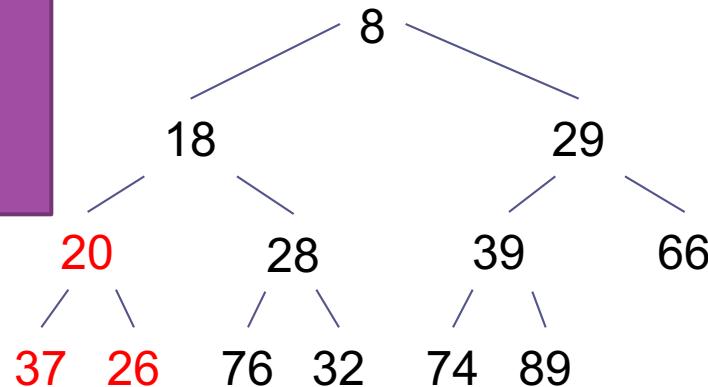
# Implementing a Heap (cont.)

28

For a node at position  $p$ ,

L. child position:  $2p + 1$

R. child position:  $2p + 2$



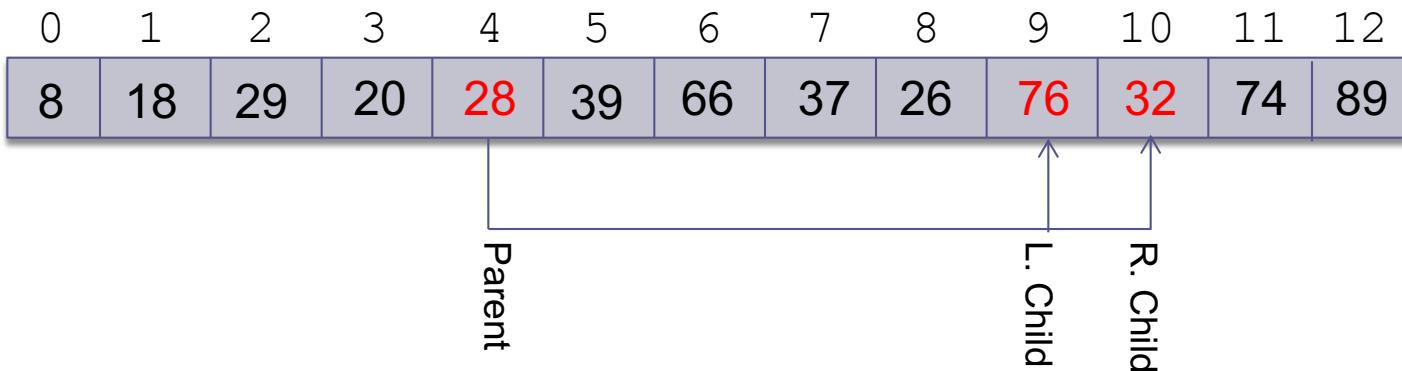
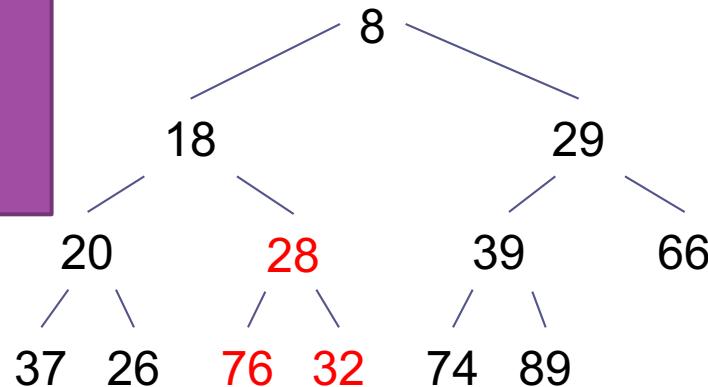
# Implementing a Heap (cont.)

29

For a node at position  $p$ ,

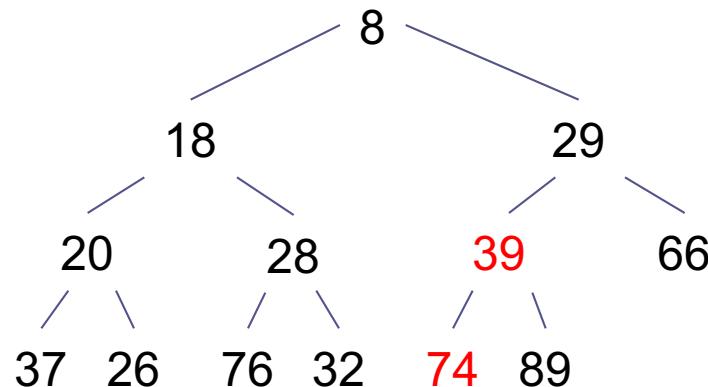
L. child position:  $2p + 1$

R. child position:  $2p + 2$

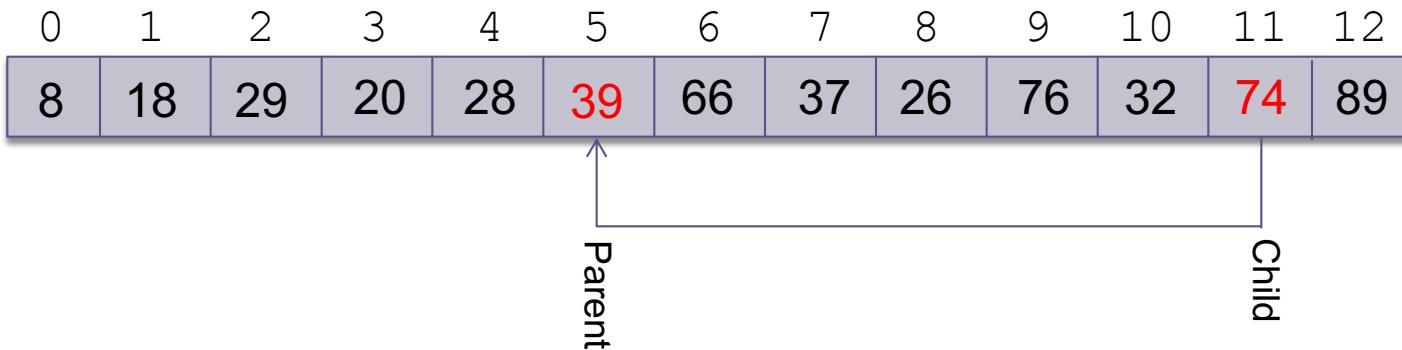


# Implementing a Heap (cont.)

30

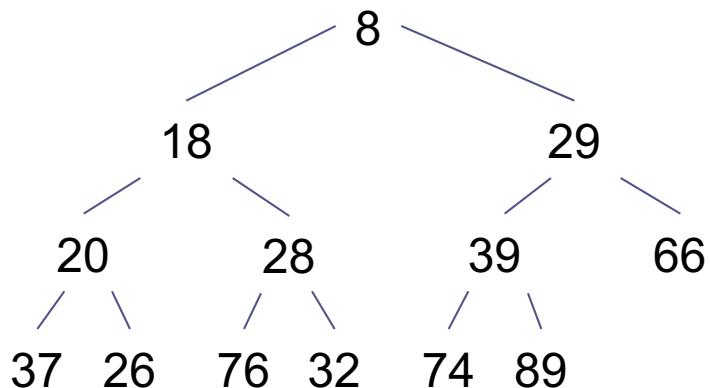


A node at position  $c$  can find its parent at  $(c - 1)/2$



# Inserting into a Heap Implemented as an ArrayList

31

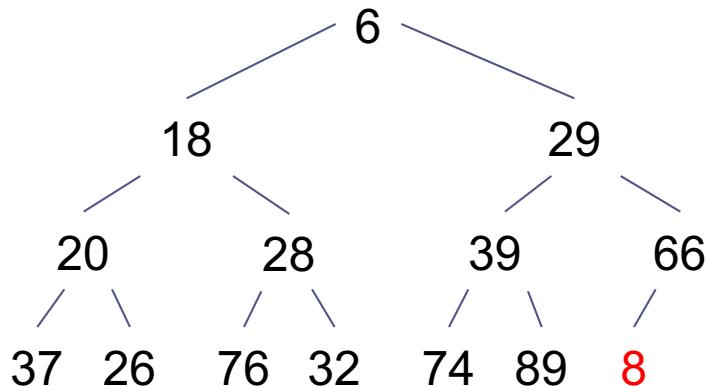


1. Insert the new element at the end of the ArrayList and set child to `table.size() - 1`

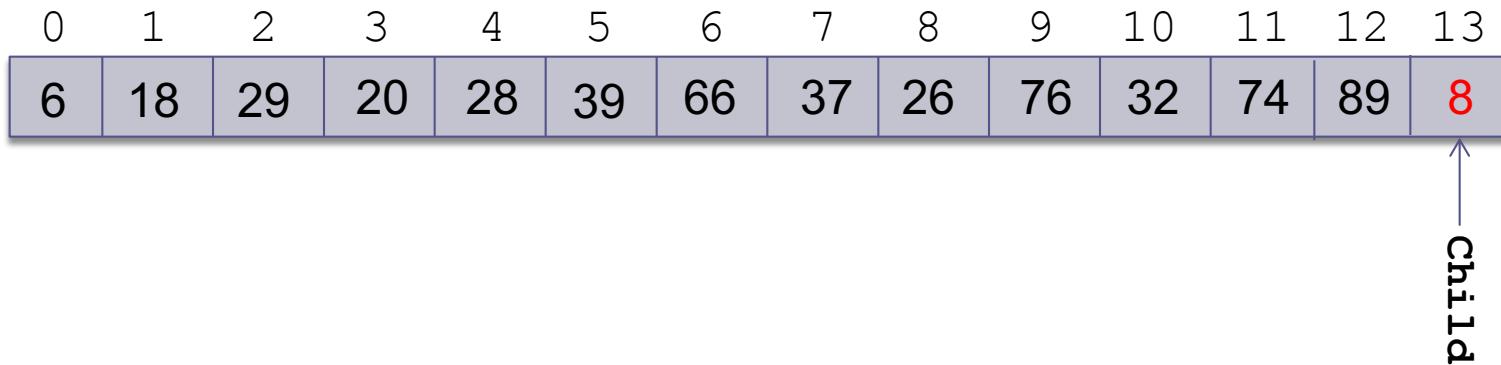
0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	18	29	20	28	39	66	37	26	76	32	74	89	

# Inserting into a Heap Implemented as an ArrayList (cont.)

32



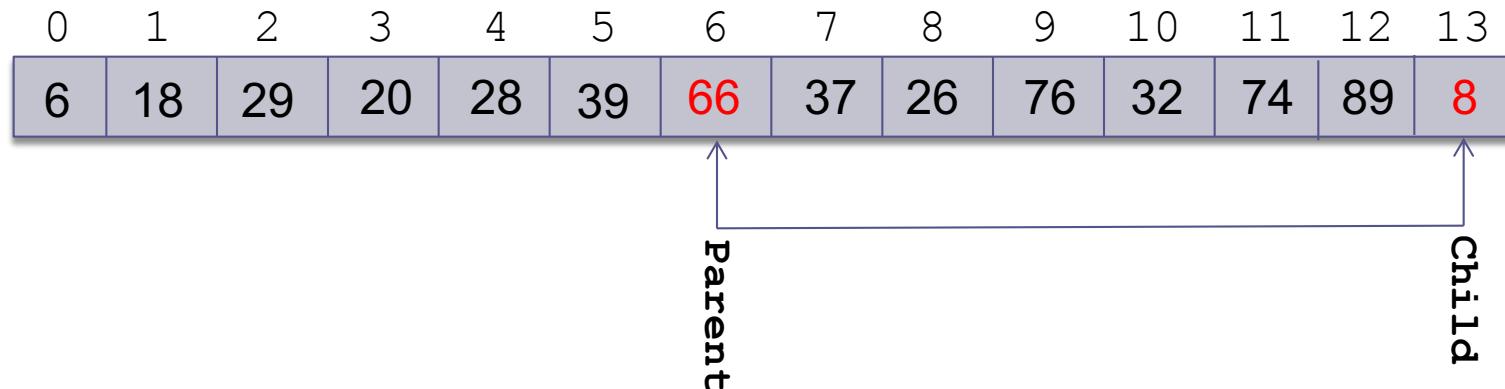
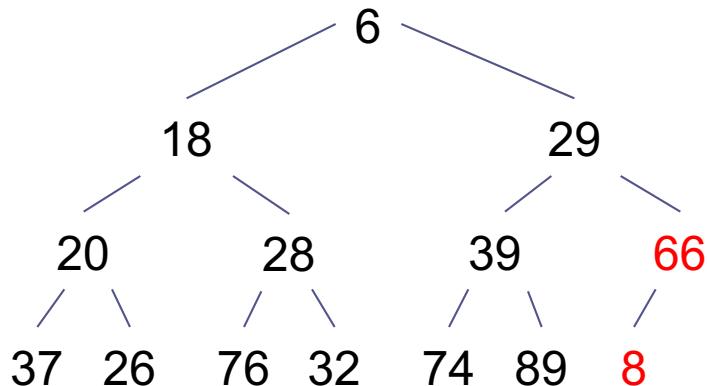
1. Insert the new element at the end of the ArrayList and set child to `table.size() - 1`



# Inserting into a Heap Implemented as an ArrayList (cont.)

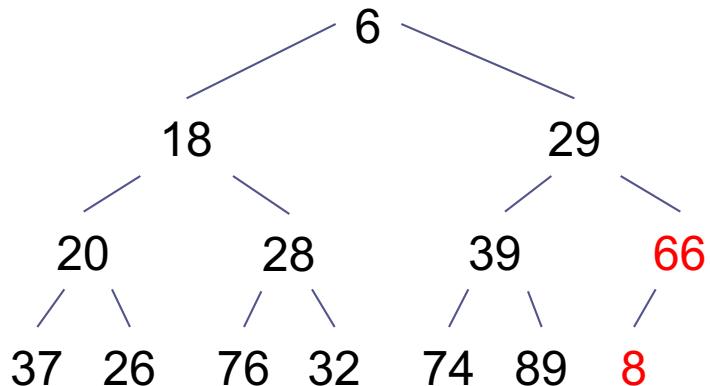
33

2. Set parent to  $(\text{child} - 1) / 2$

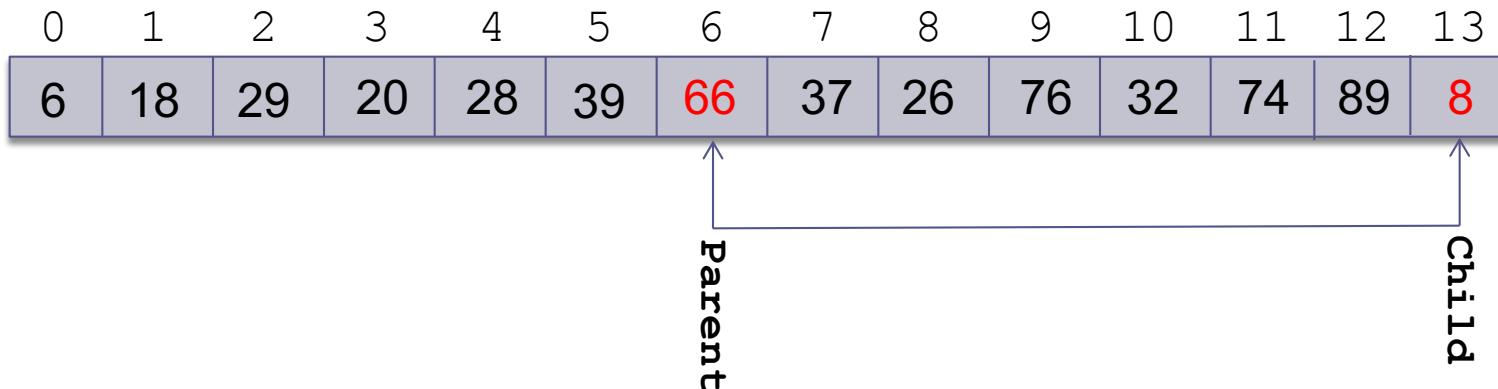


# Inserting into a Heap Implemented as an ArrayList (cont.)

34

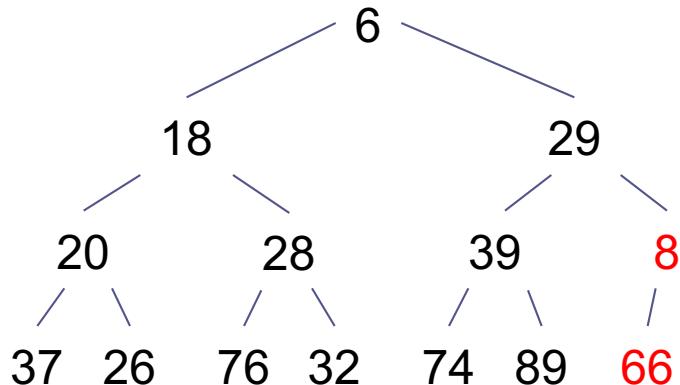


3. while (parent >= 0  
and  
table[parent] > table[child])
4. Swap table[parent]  
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2

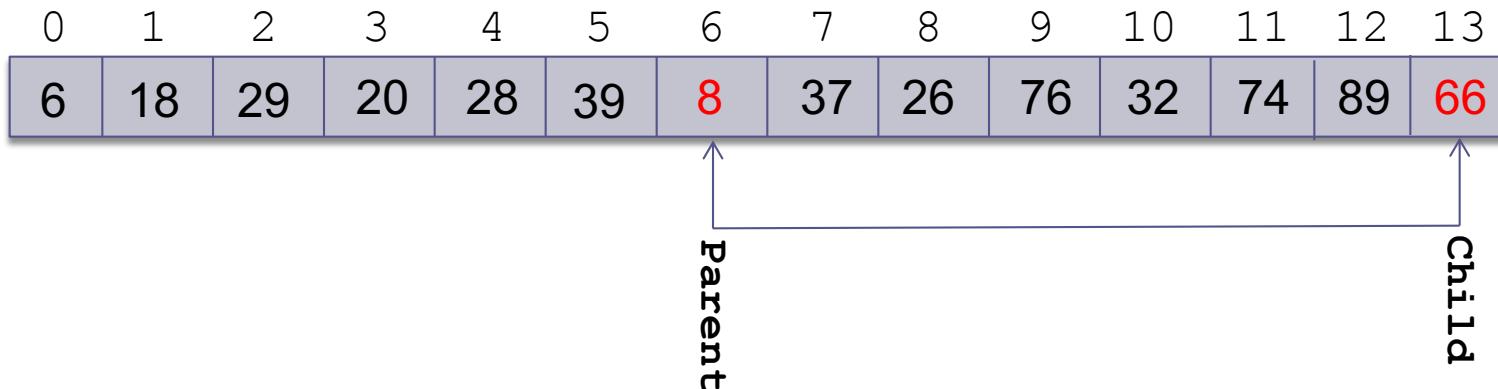


# Inserting into a Heap Implemented as an ArrayList (cont.)

35

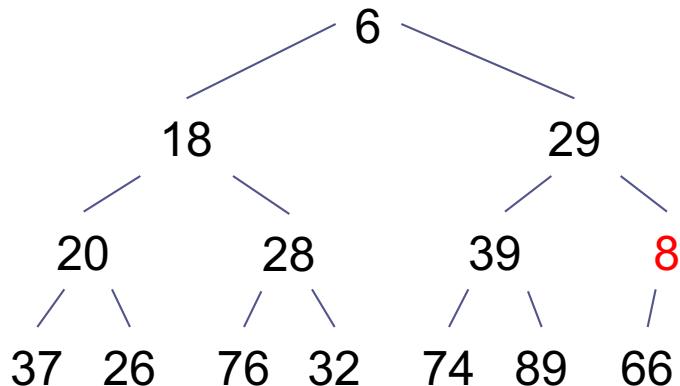


3. while (parent >= 0  
and  
table[parent] > table[child])
4. Swap table[parent]  
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2

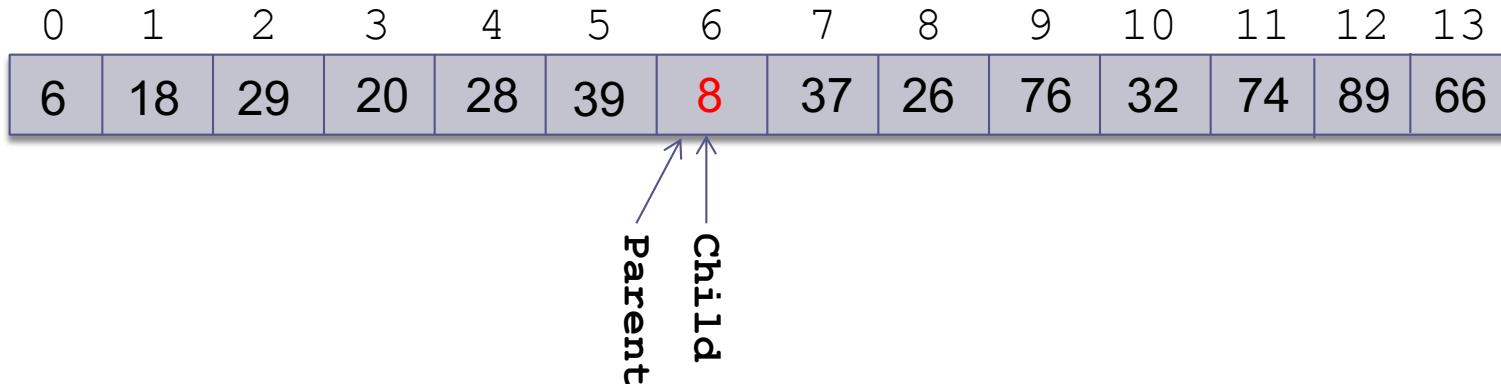


# Inserting into a Heap Implemented as an ArrayList (cont.)

36

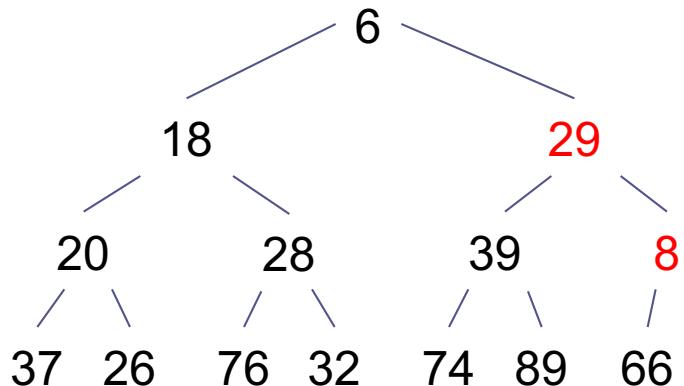


3. while (`parent >= 0`  
and  
`table[parent] > table[child]`)
4. Swap `table[parent]`  
and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to  $(\text{child}-1)/2$

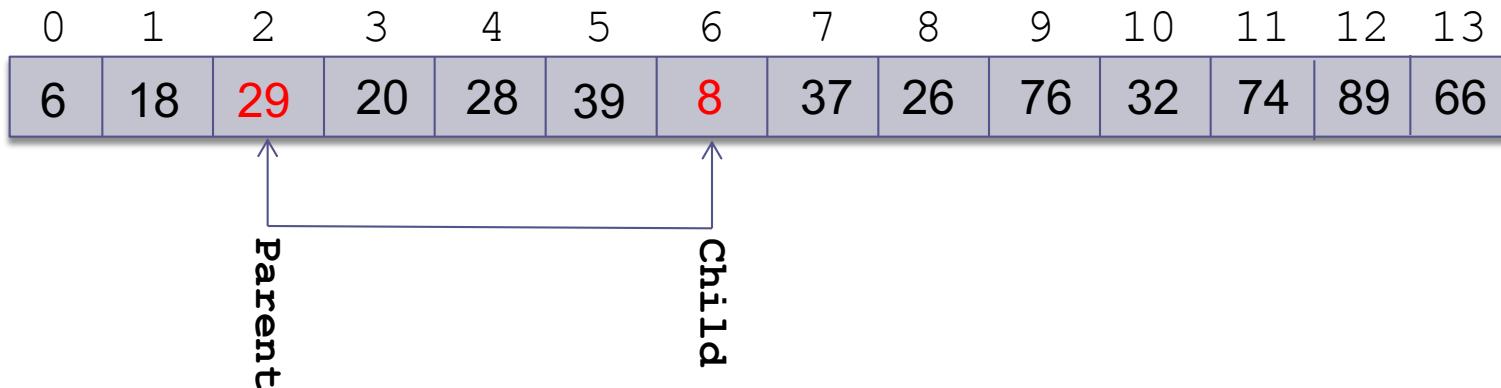


# Inserting into a Heap Implemented as an ArrayList (cont.)

37

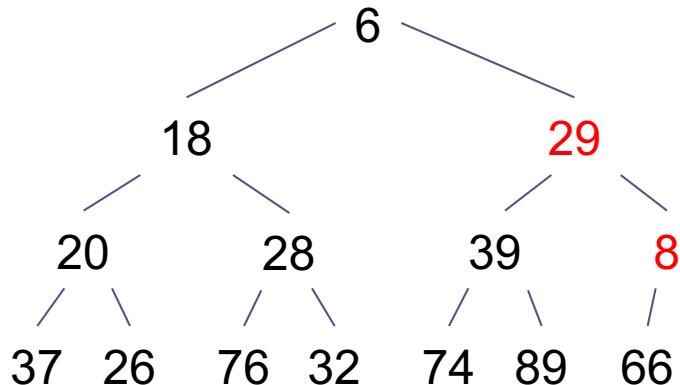


3. while (`parent >= 0`  
and  
`table[parent] > table[child]`)
4. Swap `table[parent]`  
and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to  $(\text{child}-1)/2$

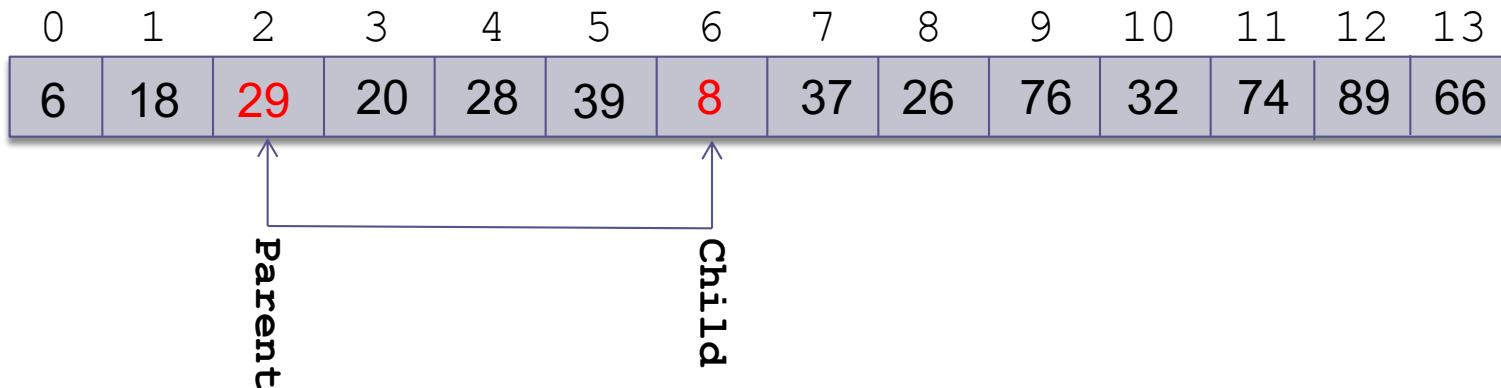


# Inserting into a Heap Implemented as an ArrayList (cont.)

38

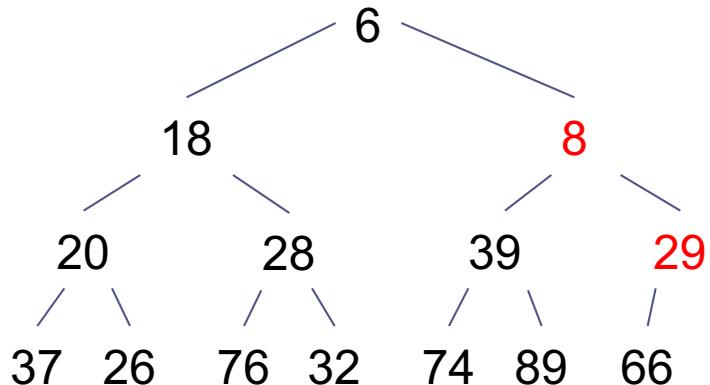


3. while (`parent >= 0`  
and  
`table[parent] > table[child]`)
4. Swap `table[parent]`  
and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to  $(\text{child}-1)/2$

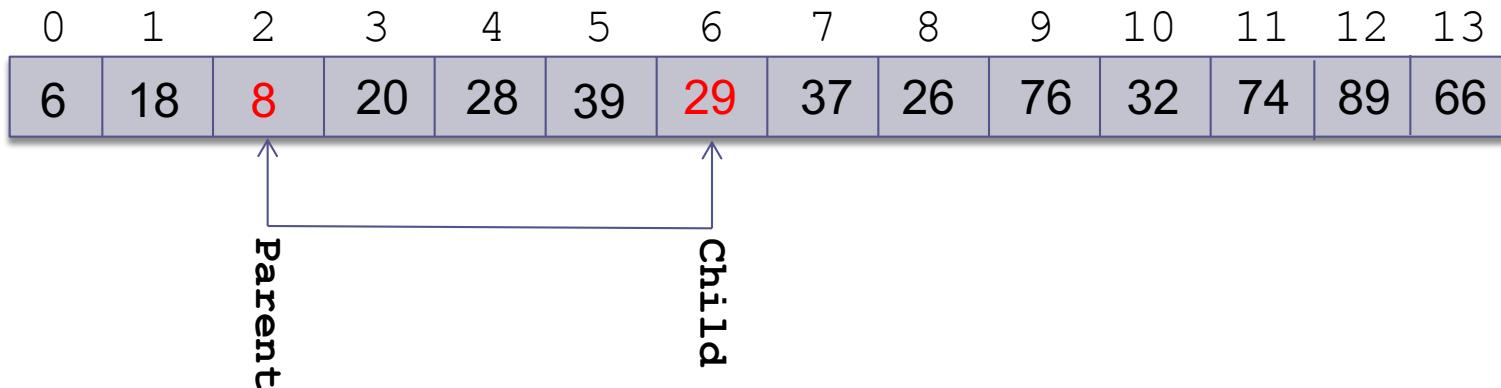


# Inserting into a Heap Implemented as an ArrayList (cont.)

39

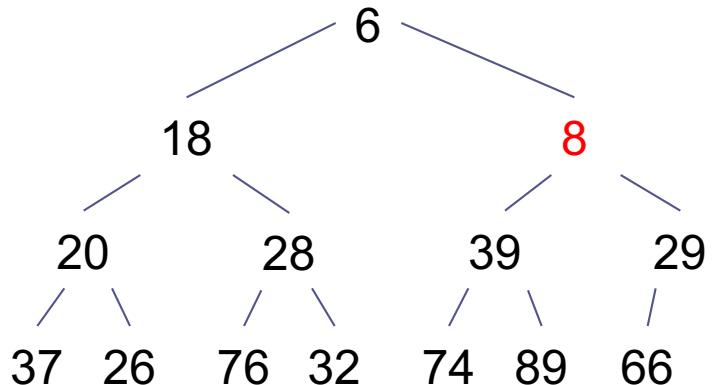


3. while (`parent >= 0`  
and  
`table[parent] > table[child]`)
4. Swap `table[parent]`  
and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to  $(\text{child}-1)/2$

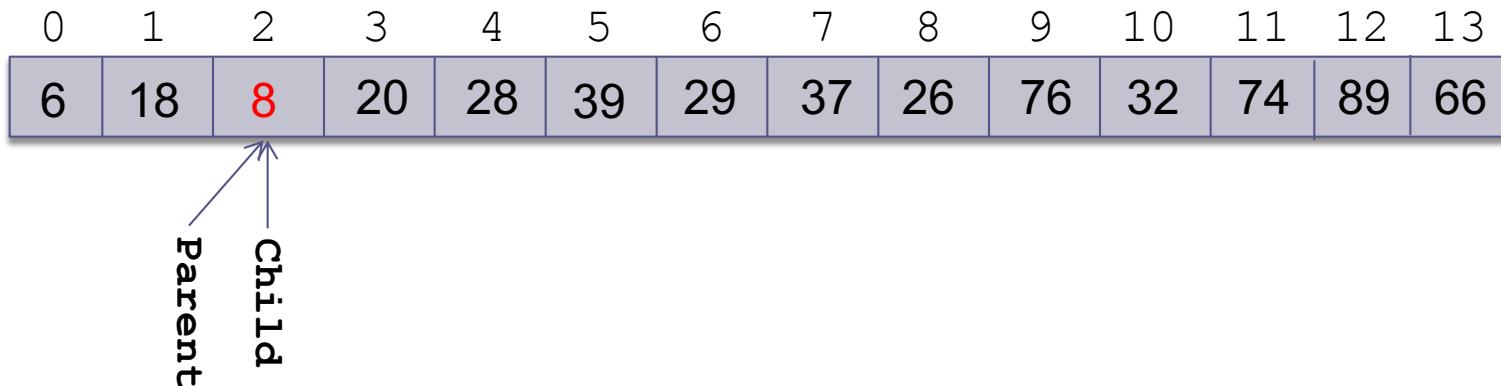


# Inserting into a Heap Implemented as an ArrayList (cont.)

40

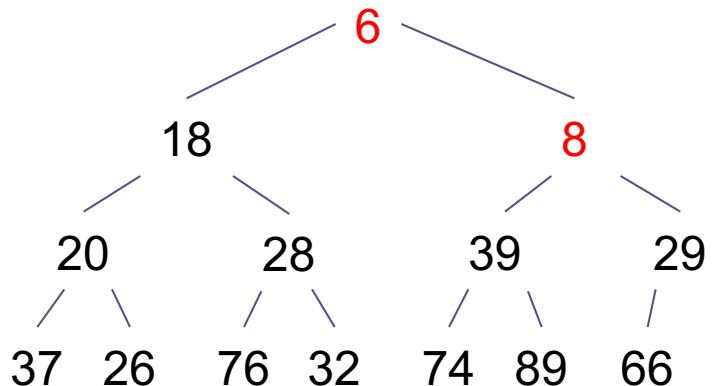


3. **while** (**parent** >= 0  
and  
**table**[**parent**] > **table**[**child**])
4. **Swap** **table**[**parent**]  
and **table**[**child**]
5. Set **child** **equal to** **parent**
6. Set **parent** **equal to** (**child**-1)/2

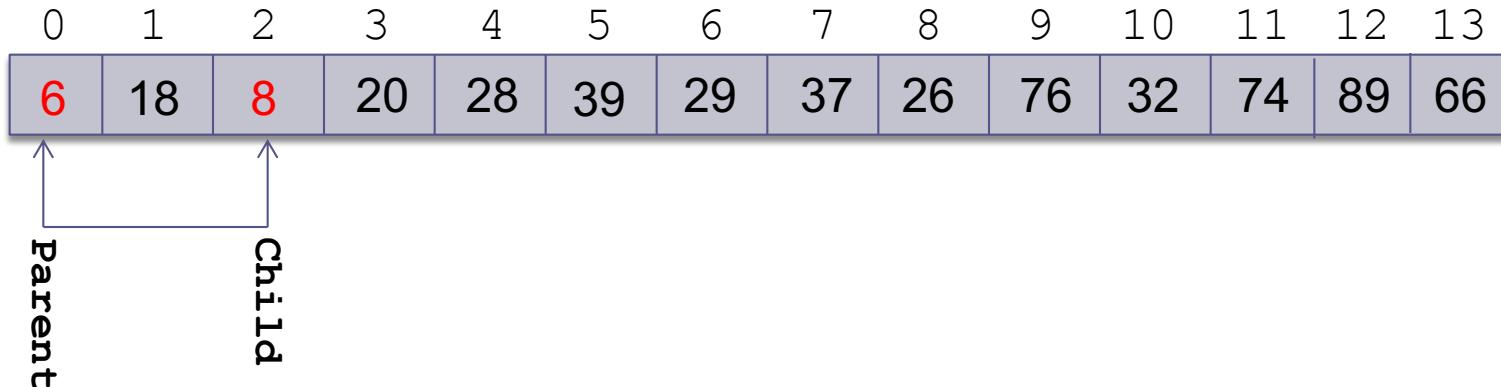


# Inserting into a Heap Implemented as an ArrayList (cont.)

41

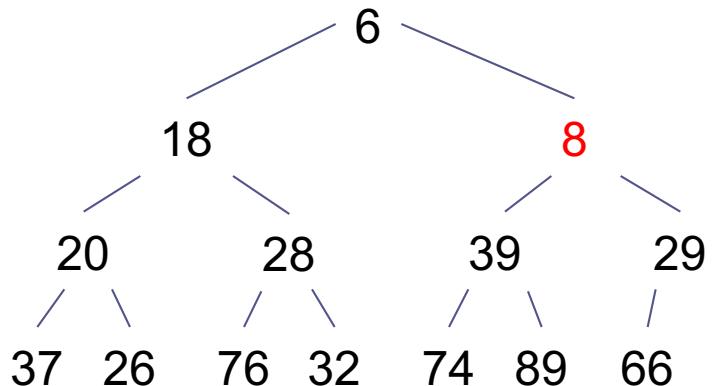


3. while (`parent >= 0`  
and  
`table[parent] > table[child]`)
4. Swap `table[parent]`  
and `table[child]`
5. Set `child` equal to `parent`
6. Set `parent` equal to  $(\text{child}-1)/2$



# Inserting into a Heap Implemented as an ArrayList (cont.)

42



3. while (parent >= 0  
and  
table[parent] > table[child])
4. Swap table[parent]  
and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	8	20	28	39	29	37	26	76	32	74	89	66

# Removal from a Heap Implemented as an ArrayList

43

## Removing an Element from a Heap Implemented as an ArrayList

1. Remove the last element (i.e., the one at `size() - 1`) and set the item at 0 to this value.
2. Set `parent` to 0.
3. **while (true)**
4.     Set `leftChild` to  $(2 * \text{parent}) + 1$  and `rightChild` to `leftChild + 1`.
5.     **if `leftChild >= table.size()`**
6.         Break out of loop.

# Removal from a Heap Implemented as an ArrayList

44

7. Assume `minChild` (the smaller child) is `leftChild`.
8.     **if** `rightChild < table.size()` and  
        `table[rightChild] < table[leftChild]`
9.         Set `minChild` to `rightChild`.
10.      **if** `table[parent] > table[minChild]`
11.         Swap `table[parent]` and `table[minChild]`.
12.         Set `parent` to `minChild`.
- else**
13.         Break out of loop.

# Performance of the Heap

45

- remove traces a path from the root to a leaf
- insert traces a path from a leaf to the root
- This requires at most  $h$  steps where  $h$  is the height of the tree
- The largest *full* tree of height  $h$  has  $2^h - 1$  nodes
- The smallest *complete* tree of height  $h$  has  $2^{(h-1)}$  nodes
- Both insert and remove are  $O(\log n)$

# Priority Queues

46

- The heap is used to implement a special kind of queue called a priority queue
- The heap is not very useful as an ADT on its own
  - ▣ We will not create a Heap interface or code a class that implements it
  - ▣ Instead, we will incorporate its algorithms when we implement a priority queue class and heapsort
- Sometimes a FIFO queue may not be the best way to implement a waiting line
- A priority queue is a data structure in which only the highest-priority item is accessible

# Priority Queues (cont.)

47

- In a print queue, sometimes it is quicker to print a short document that arrived after a very long document
- A *priority queue* is a data structure in which only the highest-priority item is accessible (as opposed to the first item entered)

# Insertion into a Priority Queue

48

```
pages = 1  
title = "web page 1"
```

```
pages = 4  
title = "history paper"
```

After inserting document with 3 pages

```
pages = 1  
title = "web page 1"
```

```
pages = 3  
title = "Lab1"
```

```
pages = 4  
title = "history paper"
```

After inserting document with 1 page

```
pages = 1  
title = "web page 1"
```

```
pages = 1  
title = "receipt"
```

```
pages = 3  
title = "Lab1"
```

```
pages = 4  
title = "history paper"
```

# PriorityQueue Class

49

- Java provides a `PriorityQueue<E>` class that implements the `Queue<E>` interface given in Chapter 4.

Method	Behavior
<code>boolean offer(E item)</code>	Inserts an item into the queue. Returns <code>true</code> if successful; returns <code>false</code> if the item could not be inserted.
<code>E remove()</code>	Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a <code>NoSuchElementException</code> .
<code>E poll()</code>	Removes the smallest entry and returns it. If the queue is empty, returns <code>null</code> .
<code>E peek()</code>	Returns the smallest entry without removing it. If the queue is empty, returns <code>null</code> .
<code>E element()</code>	Returns the smallest entry without removing it. If the queue is empty, throws a <code>NoSuchElementException</code> .

# Using a Heap as the Basis of a Priority Queue

50

- In a priority queue, just like a heap, the smallest item **always** is removed first
- Because heap insertion and removal is  $O(\log n)$ , a heap can be the basis of a very efficient implementation of a priority queue
- While the `java.util.PriorityQueue` uses an `Object[]` array, we will use an `ArrayList` for our custom priority queue, `KWPriorityQueue`

# Design of a KWPriorityQueue Class

51

Data Field	Attribute
ArrayList<E> theData	An ArrayList to hold the data.
Comparator<E> comparator	An optional object that implements the Comparator<E> interface by providing a compare method.
Method	Behavior
KWPriorityQueue()	Constructs a heap-based priority queue that uses the elements' natural ordering.
KWPriorityQueue(Comparator<E> comp)	Constructs a heap-based priority queue that uses the compare method of Comparator comp to determine the ordering of the elements.
private int compare(E left, E right)	Compares two objects and returns a negative number if object left is less than object right, zero if they are equal, and a positive number if object left is greater than object right.
private void swap(int i, int j)	Exchanges the object references in theData at indexes i and j.

# Design of a KWPriorityQueue Class

## (cont.)

52

```
import java.util.*;  
/** The KWPriorityQueue implements the Queue interface  
    by building a heap in an ArrayList. The heap is  
    structured so that the "smallest" item is at the top.  
 */  
public class KWPriorityQueue<E> extends AbstractQueue<E>  
    implements Queue<E> {  
    // Data Fields  
    /** The ArrayList to hold the data. */  
    private ArrayList<E> theData;  
    /** An optional reference to a Comparator object. */  
    Comparator<E> comparator = null;  
  
    // Methods  
    // Constructor  
    public KWPriorityQueue() {  
        theData = new ArrayList<E>();  
    }
```

# offer Method

53

```
/** Insert an item into the priority queue.  
 * pre: The ArrayList theData is in heap order.  
 * post: The item is in the priority queue and  
 *       theData is in heap order.  
 * @param item The item to be inserted  
 * @throws NullPointerException if the item to be  
 *       inserted is null.  
 */  
@Override  
public boolean offer(E item) {  
    // Add the item to the heap.  
    theData.add(item);
```

# offer Method

54

```
// child is newly inserted item.  
int child = theData.size() - 1;  
int parent = (child - 1) / 2; // Find child's parent.  
// Reheap  
while (parent >= 0 && compare(theData.get(parent),  
        theData.get(child)) > 0) {  
    swap(parent, child);  
    child = parent;  
    parent = (child - 1) / 2;  
}  
return true;
```

# poll Method

55

```
/** Remove an item from the priority queue
pre: The ArrayList theData is in heap order.
post: Removed smallest item, theData is in heap
order.
@return The item with the smallest priority value
or null if empty.
*/
@Override
public E poll() {
    if (isEmpty()) {
        return null;
    }
}
```

# poll Method

56

```
// Save the top of the heap.  
E result = theData.get(0);  
// If only one item then remove it.  
if (theData.size() == 1) {  
    theData.remove(0);  
    return result;  
}  
  
/* Remove the last item from the ArrayList and  
   place it into the first position. */  
theData.set(0, theData.remove(theData.size() - 1));  
// The parent starts at the top.  
int parent = 0;
```

# poll Method (cont.)

```
while (true) {  
    int leftChild = 2 * parent + 1;  
    if (leftChild >= theData.size()) {  
        break; // Out of heap.  
    }  
    int rightChild = leftChild + 1;  
    // Assume leftChild is smaller.  
    int minChild = leftChild;  
    // See whether rightChild is smaller.  
    if (rightChild < theData.size()  
        && compare(theData.get(leftChild),  
                  theData.get(rightChild)) > 0) {  
        minChild = rightChild;  
    }  
}
```

# poll Method (cont.)

```
// assert: minChild is the index of the
// smaller child.
// Move smaller child up heap if necessary.
if (compare(theData.get(parent),
            theData.get(minChild)) > 0) {
    swap(parent, minChild);
    parent = minChild;
}
else
{ // Heap property is restored.
    break;
}
return result;
}
```

# Other Methods

59

- The iterator and size methods are implemented via delegation to the corresponding ArrayList methods
- Method isEmpty tests whether the result of calling method size is 0 and is inherited from class AbstractCollection
- The implementations of methods peek and remove are left as exercises

# Using a Comparator

60

- To use an ordering that is different from the natural ordering, provide a constructor that has a `Comparator<E>` parameter

```
/** Creates a heap-based priority queue with the specified
 * initial capacity that orders its elements according to the
 * specified comparator.
 *
 * @param cap The initial capacity for this priority queue
 *
 * @param comp The comparator used to order this priority
 * queue
 *
 * @throws IllegalArgumentException if cap is less than 1
 */
```

# Using a Comparator

61

```
public KWPriorityQueue(int cap, Comparator<E> comp) {  
    if (cap < 1)  
        throw new IllegalArgumentException();  
    theData = new ArrayList<E>();  
    comparator = comp;  
}
```

# compare **Method**

62

- If **data field comparator** references a Comparator<E> object, method compare delegates the task to the object's compare method
- If **comparator** is null, it will delegate to method compareTo

# compare Method (cont.)

63

```
/** Compare two items using either a Comparator object's compare
method or their natural ordering using method compareTo.
pre: If comparator is null, left and right implement
Comparable<E>.
@param left One item
@param right The other item
@return Negative int if left less than right,
         0 if left equals right,
         positive int if left > right
@throws ClassCastException if items are not Comparable
*/
private int compare(E left, E right) {
    if (comparator != null) { // A Comparator is defined.
        return comparator.compare(left, right);
    } else {                  // Use left's compareTo method.
        return ((Comparable<E>) left).compareTo(right);
    }
}
```

# PrintDocuments Example

64

- The class PrintDocument is used to define documents to be printed on a printer
- We want to order documents by a value that is a function of both size and time submitted
- In the client program, use

```
Queue printQueue =  
    new PriorityQuene(new ComparePrintDocuments());
```

# PrintDocuments Example (cont.)

65

---

## LISTING 6.8

ComparePrintDocuments.java

```
import java.util.Comparator;

/** Class to compare PrintDocuments based on both
     their size and time stamp.
*/
public class ComparePrintDocuments implements Comparator<PrintDocument> {
    /** Weight factor for size. */
    private static final double P1 = 0.8;
    /** Weight factor for time. */
    private static final double P2 = 0.2;

    /** Compare two PrintDocuments.
        @param left The left-hand side of the comparison
        @param right The right-hand side of the comparison
    }
```

# PrintDocuments Example (cont.)

66

```
        @return -1 if left < right; 0 if left == right;  
              and +1 if left > right  
*/  
public int compare(PrintDocument left, PrintDocument right) {  
    return Double.compare(orderValue(left), orderValue(right));  
}  
  
/** Compute the order value for a print document.  
 * @param pd The PrintDocument  
 * @return The order value based on the size and time stamp  
*/  
private double orderValue(PrintDocument pd) {  
    return P1 * pd.getSize() + P2 * pd.getTimeStamp();  
}  
}
```



# CS 570: Data Structures

## Sets and Maps

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 7 (PART 1)

# Chapter Objectives

3

- To understand the Java Map and Set interfaces and how to use them
- To learn about hash coding and its use to facilitate efficient search and retrieval
- To study two forms of hash tables—open addressing and chaining—and to understand their relative benefits and performance trade-offs

# Chapter Objectives (cont.)

4

- To learn how to implement both hash table forms
- To be introduced to the implementation of Maps and Sets
- To see how two earlier applications can be implemented more easily using Map objects for data storage

# Week 12

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 7.1-7.3

# Introduction

6

- We learned about part of the Java Collection Framework in Chapter 2 (`ArrayList` and `LinkedList`)
- The classes that implement the `List` interface are all *indexed collections*
  - An index or subscript is associated with each element
  - The element's index often reflects the relative order of its insertion in the list
  - Searching for a particular value in a list is generally  $O(n)$
  - An exception is a binary search of a sorted object, which is  $O(\log n)$

# Introduction (cont.)

7

- In this chapter, we consider another part of the Collection hierarchy: the Set interface and the classes that implement it
- Set objects
  - are not indexed
  - do not reveal the order of insertion of items
  - enable efficient search and retrieval of information
  - allow removal of elements without moving other elements around

# Introduction (cont.)

8

- Relative to a Set, Map objects provide efficient search and retrieval of entries that contain pairs of objects (a unique key and the information)
- Hash tables (implemented by a Map or Set) store objects at arbitrary locations and offer an average constant time for insertion, removal, and searching

# Sets and the Set Interface

## Section 7.1

# The Set Abstraction

10

- A set is a collection that contains no duplicate elements and at most one null element
  - adding "apples" to the set `{"apples", "oranges", "pineapples"}` results in the same set (no change)
- Operations on sets include:
  - testing for membership
  - adding elements
  - removing elements
  - union  $A \cup B$
  - intersection  $A \cap B$
  - difference  $A - B$
  - subset  $A \subset B$

# The Set Abstraction(cont.)

11

- The union of two sets A, B is a set whose elements belong either to A or B or to both A and B.  
Example:  $\{1, 3, 5, 7\} \cup \{2, 3, 4, 5\}$  is  $\{1, 2, 3, 4, 5, 7\}$
- The intersection of sets A, B is the set whose elements belong to both A and B.  
Example:  $\{1, 3, 5, 7\} \cap \{2, 3, 4, 5\}$  is  $\{3, 5\}$
- The difference of sets A, B is the set whose elements belong to A but not to B.  
Examples:  $\{1, 3, 5, 7\} - \{2, 3, 4, 5\}$  is  $\{1, 7\}$ ;  $\{2, 3, 4, 5\} - \{1, 3, 5, 7\}$  is  $\{2, 4\}$
- Set A is a subset of set B if every element of set A is also an element of set B.  
Example:  $\{1, 3, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\}$  is true

# The Set Interface and Methods

12

- Required methods: testing set membership, testing for an empty set, determining set size, and creating an iterator over the set
- Optional methods: adding an element and removing an element
- Constructors to enforce the “no duplicate members” criterion
  - The add method does not allow duplicate items to be inserted

# The Set Interface and Methods(cont.)

13

- Required method: `containsAll` tests the subset relationship
- Optional methods: `addAll`, `retainAll`, and `removeAll` perform union, intersection, and difference, respectively

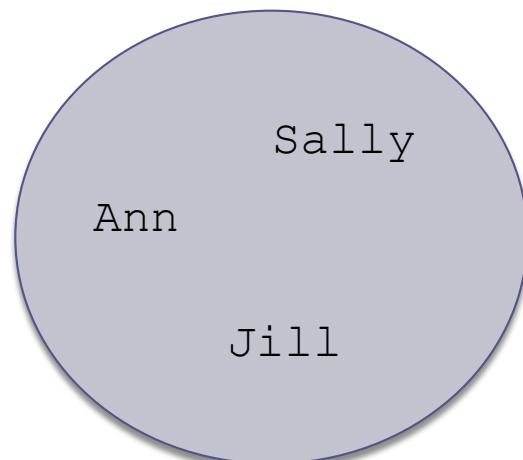
# The Set Interface and Methods(cont.)

14

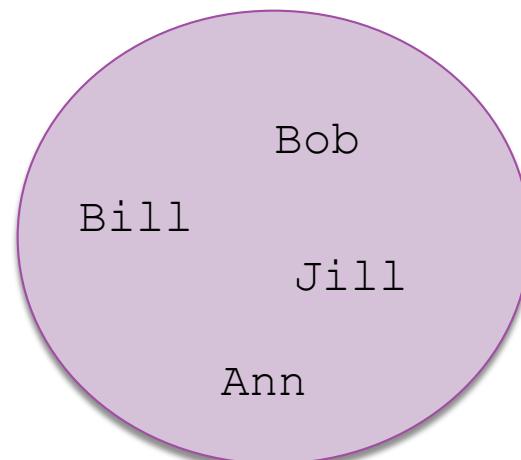
Method	Behavior
<code>boolean add(E obj)</code>	Adds item <code>obj</code> to this set if it is not already present (optional operation) and returns <code>true</code> . Returns false if <code>obj</code> is already in the set.
<code>boolean addAll(Collection&lt;E&gt; coll)</code>	Adds all of the elements in collection <code>coll</code> to this set if they're not already present (optional operation). Returns <code>true</code> if the set is changed. Implements <i>set union</i> if <code>coll</code> is a Set.
<code>boolean contains(Object obj)</code>	Returns <code>true</code> if this set contains an element that is equal to <code>obj</code> . Implements a test for <i>set membership</i> .
<code>boolean containsAll(Collection&lt;E&gt; coll)</code>	Returns <code>true</code> if this set contains all of the elements of collection <code>coll</code> . If <code>coll</code> is a set, returns <code>true</code> if this set is a subset of <code>coll</code> .
<code>boolean isEmpty()</code>	Returns <code>true</code> if this set contains no elements.
<code>Iterator&lt;E&gt; iterator()</code>	Returns an iterator over the elements in this set.
<code>boolean remove(Object obj)</code>	Removes the set element equal to <code>obj</code> if it is present (optional operation). Returns <code>true</code> if the object was removed.
<code>boolean removeAll(Collection&lt;E&gt; coll)</code>	Removes from this set all of its elements that are contained in collection <code>coll</code> (optional operation). Returns <code>true</code> if this set is changed. If <code>coll</code> is a set, performs the <i>set difference</i> operation.
<code>boolean retainAll(Collection&lt;E&gt; coll)</code>	Retains only the elements in this set that are contained in collection <code>coll</code> (optional operation). Returns <code>true</code> if this set is changed. If <code>coll</code> is a set, performs the <i>set intersection</i> operation.
<code>int size()</code>	Returns the number of elements in this set (its cardinality).

# The Set Interface and Methods(cont.)

15



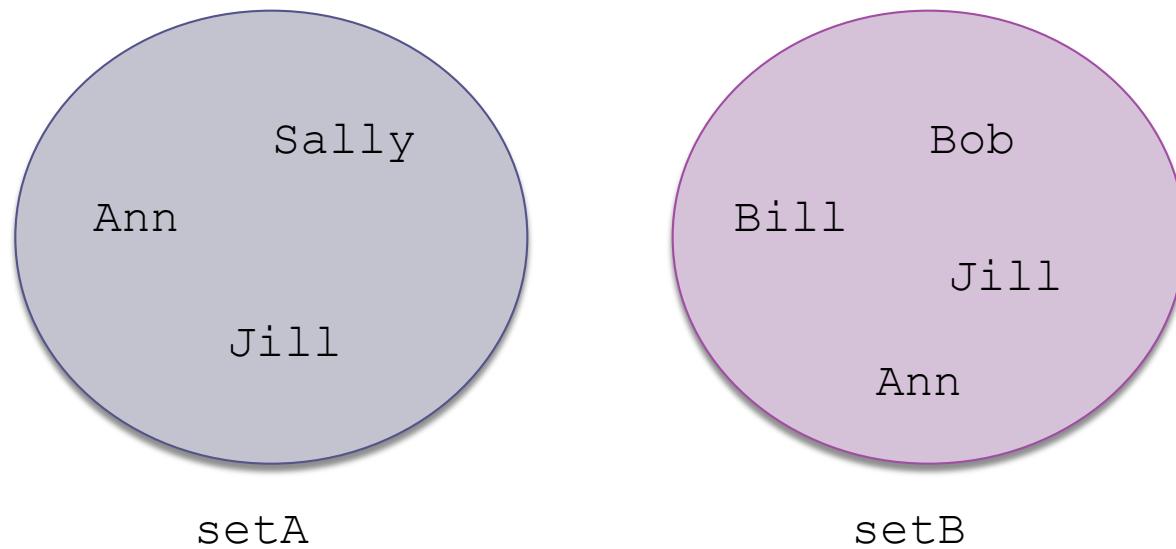
setA



setB

# The Set Interface and Methods(cont.)

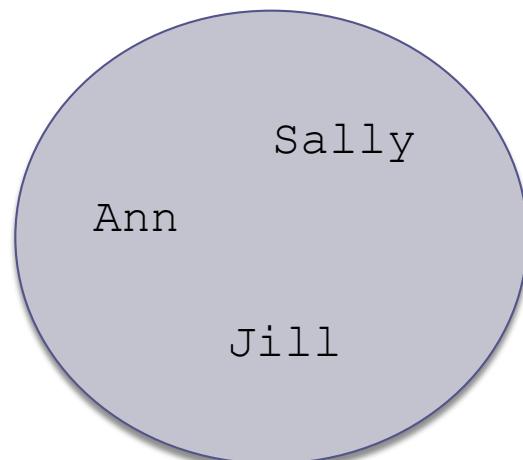
16



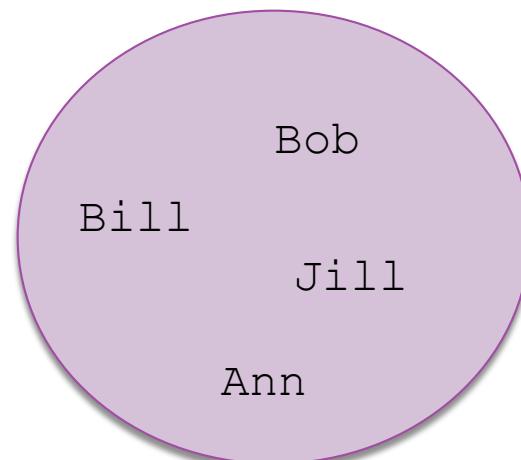
```
setA.addAll(setB);
```

# The Set Interface and Methods(cont.)

17



setA



setB

```
setA.addAll(setB);
```

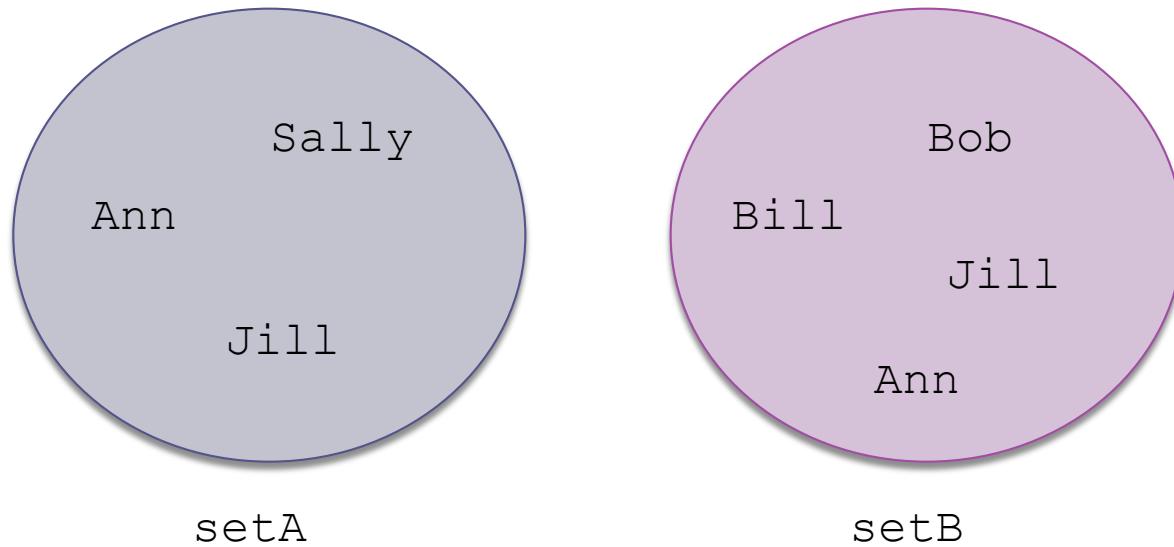
```
System.out.println(setA);
```

Outputs:

```
[Bill, Jill, Ann, Sally, Bob]
```

# The Set Interface and Methods(cont.)

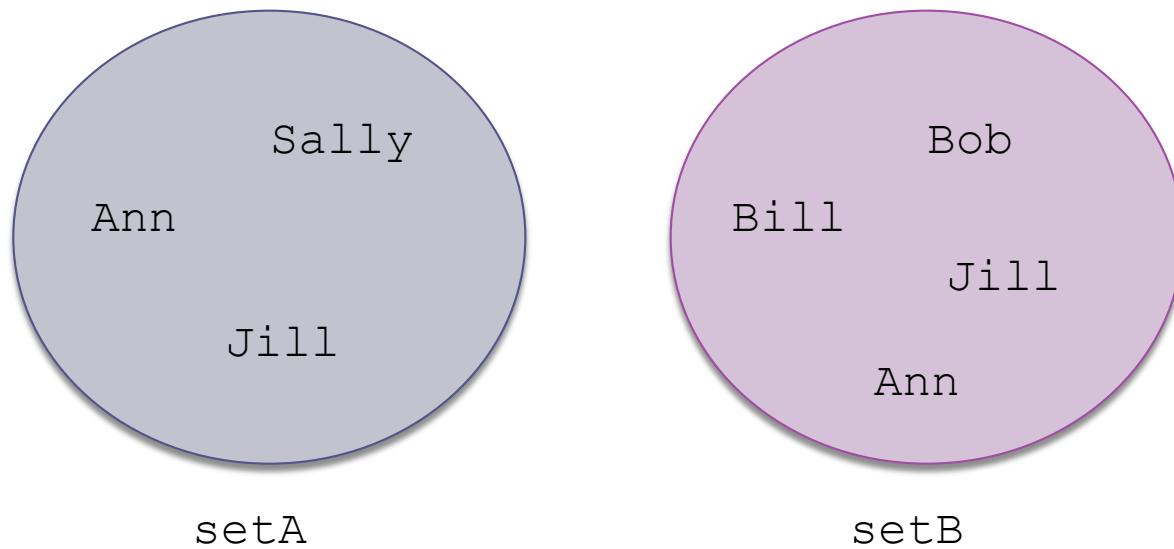
18



If a copy of original `setA` is in `setACopy`, then . . .

# The Set Interface and Methods(cont.)

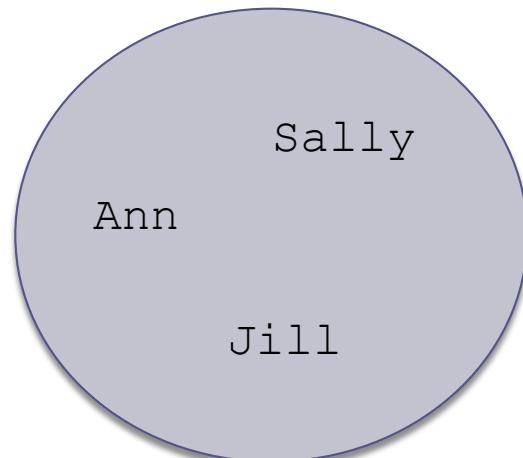
19



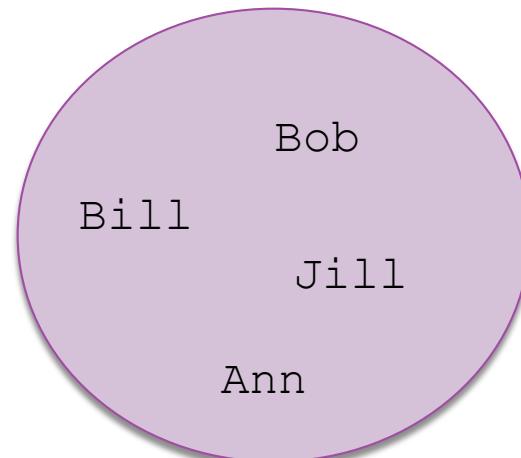
```
setACopy.retainAll(setB);
```

# The Set Interface and Methods(cont.)

20



setA



setB

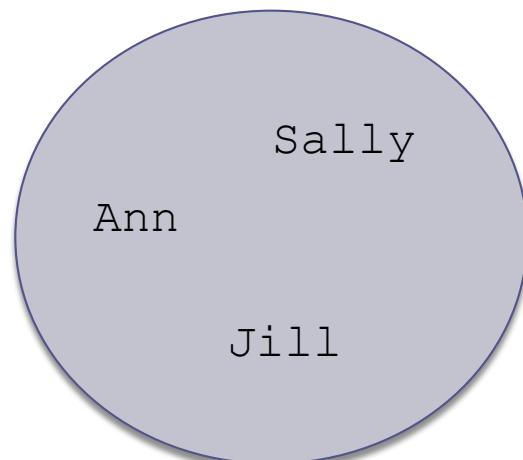
```
setACopy.retainAll(setB);  
  
System.out.println(setACopy);
```

Outputs:

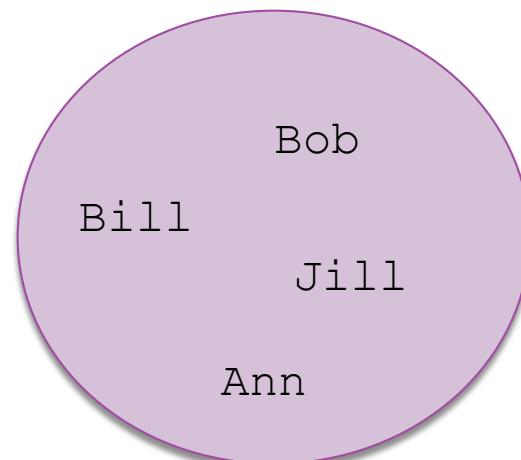
[Jill, Ann]

# The Set Interface and Methods(cont.)

21



setA



setB

```
setACopy.removeAll(setB);  
  
System.out.println(setACopy);
```

Outputs:  
[Sally]

# Comparison of Lists and Sets

22

- Collections implementing the Set interface must contain unique elements
- Unlike the List.add method, the Set.add method returns false if you attempt to insert a duplicate item
- Unlike a List, a Set does not have a get method—elements cannot be accessed by index

# Comparison of Lists and Sets (cont.)

23

- You can iterate through all elements in a Set using an Iterator object, but the elements will be accessed in arbitrary order

```
for (String nextItem : setA) {  
    //Do something with nextItem  
    ...  
}
```

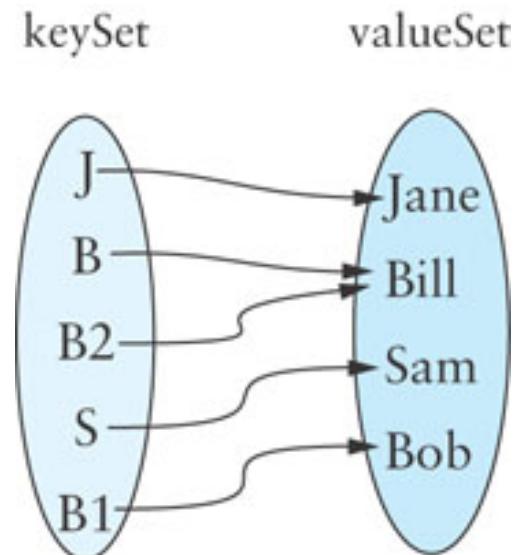
# Maps and the Map Interface

## Section 7.2

# Maps and the Map Interface

25

- The Map is related to the Set
- Mathematically, a Map is a set of ordered pairs whose elements are known as the key and the value
- Keys must be unique, but values need not be unique
- You can think of each key as a “mapping” to a particular value
- A map provides efficient storage and retrieval of information in a table
- A map can have *many-to-one* mapping: (B, Bill), (B2, Bill)



{ (J, Jane), (B, Bill),  
(S, Sam), (B1, Bob),  
(B2, Bill) }

# Maps and the Map Interface(cont.)

26

- In an *onto* mapping, all the elements of `valueSet` have a corresponding member in `keySet`
- The Map interface should have methods of the form
  - V.get (Object key)
  - V.put (K key, V value)

# Maps and the Map Interface(cont.)

27

- When information about an item is stored in a table, the information should have a unique ID
- A unique ID may or may not be a number
- This unique ID is equivalent to a key

Type of item	Key	Value
University student	Student ID number	Student name, address, major, grade point average
Online store customer	E-mail address	Customer name, address, credit card information, shopping cart
Inventory item	Part ID	Description, quantity, manufacturer, cost, price

# Map Interface

28

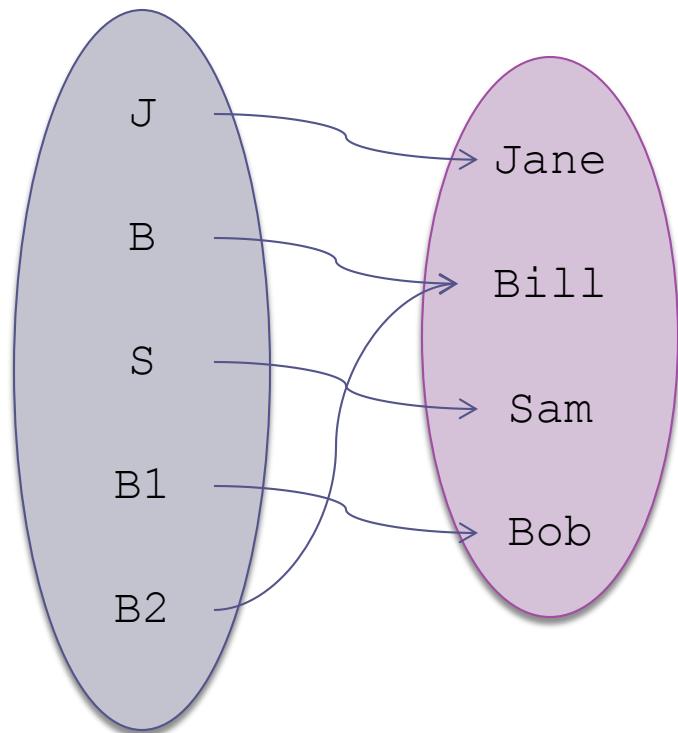
Method	Behavior
V get(Object key)	Returns the value associated with the specified key. Returns <b>null</b> if the key is not present.
boolean isEmpty()	Returns <b>true</b> if this map contains no key-value mappings.
V put(K key, V value)	Associates the specified <b>value</b> with the specified <b>key</b> in this map (optional operation). Returns the previous <b>value</b> associated with the specified <b>key</b> , or <b>null</b> if there was no mapping for the <b>key</b> .
V remove(Object key)	Removes the mapping for this <b>key</b> from this map if it is present (optional operation). Returns the previous <b>value</b> associated with the specified <b>key</b> , or <b>null</b> if there was no mapping for the <b>key</b> .
int size()	Returns the number of key-value mappings in this map.

# Map Interface (cont.)

29

- The following statements build a Map object:

```
Map<String, String> aMap =  
    new HashMap<String,  
    String>();  
  
aMap.put("J", "Jane");  
aMap.put("B", "Bill");  
aMap.put("S", "Sam");  
aMap.put("B1", "Bob");  
aMap.put("B2", "Bill");
```



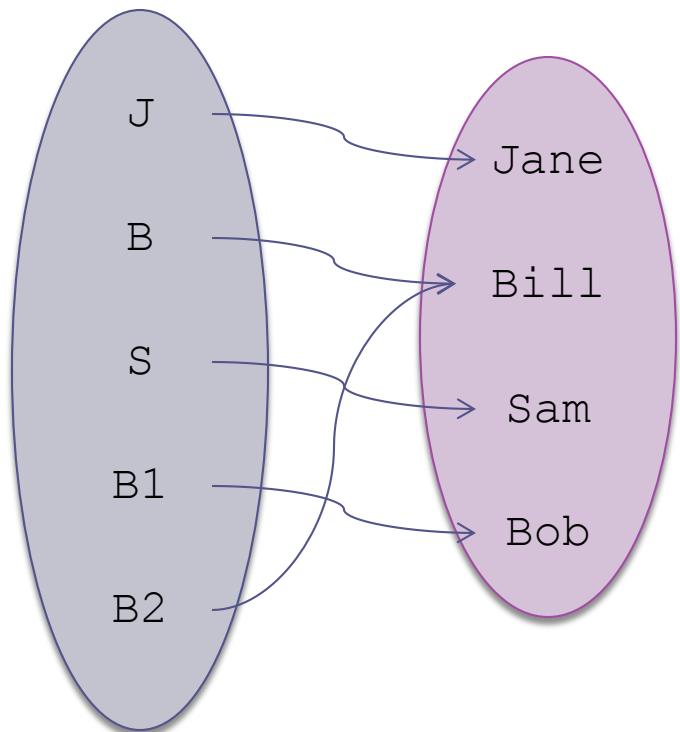
# Map Interface (cont.)

30

```
aMap.get("B1")
```

**returns:**

"Bob"



# Map Interface (cont.)

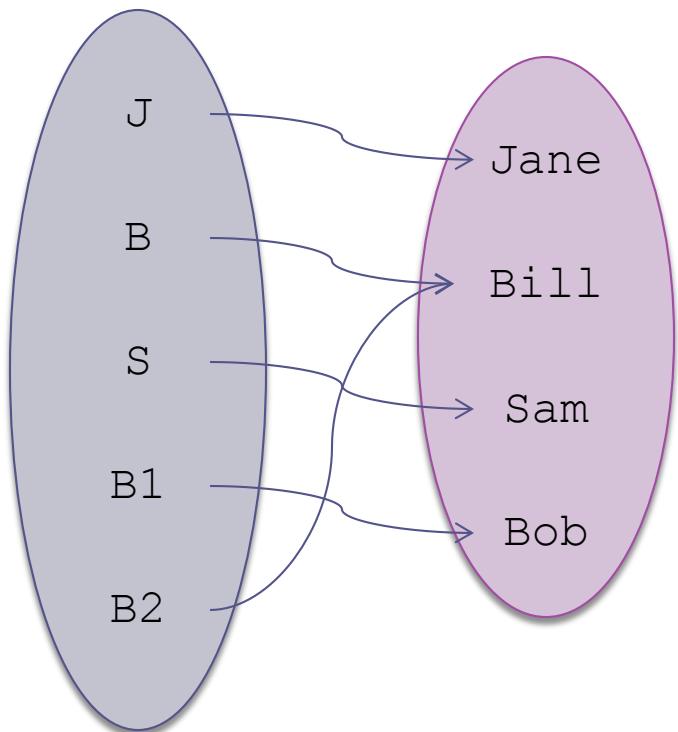
31

```
aMap.get("Bill")
```

**returns:**

null

("Bill" is a value, not a key)



# Hash Tables

## Section 7.3

# Hash Tables

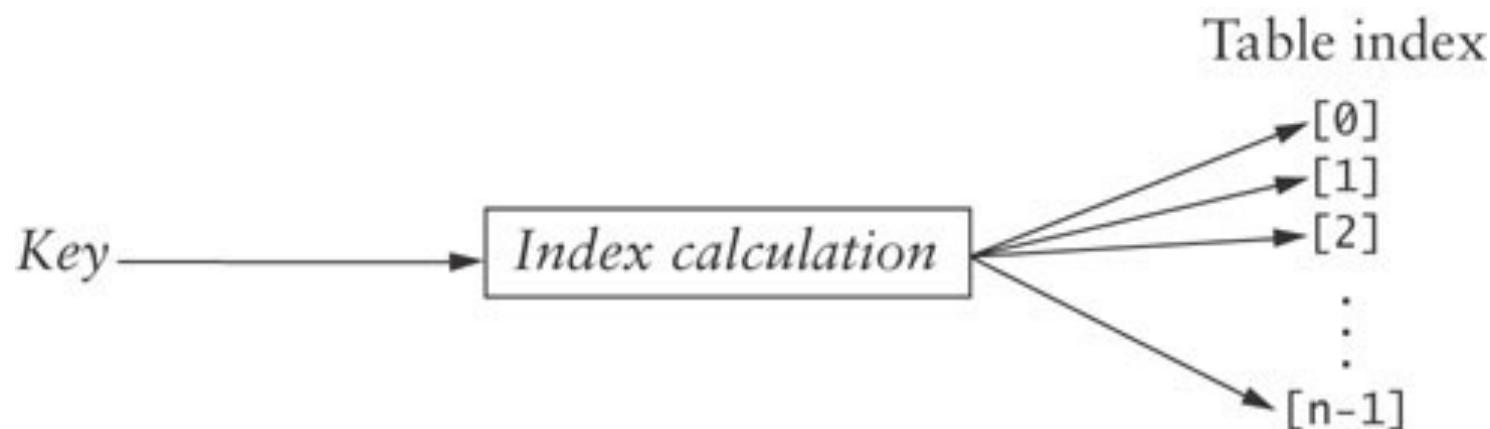
33

- The goal of hash table is to be able to access an entry based on its key value, not its location
- We want to be able to access an entry directly through its key value, rather than by having to determine its location first by searching for the key value in an array
- Using a hash table enables us to retrieve an entry in constant time (on average,  $O(1)$ )

# Hash Codes and Index Calculation

34

- The basis of hashing is to transform the item's key value into an integer value (its *hash code*) which is then transformed into a table index



# Hash Codes and Index Calculation

## (cont.)

35

- Consider the Huffman code problem from chapter 6
- If a text contains only ASCII values, which are the first 128 Unicode values we could use a table of size 128 and let its Unicode value be its location in the table

```
int index = asciiChar;
```

...	...
65	A, 8
66	B, 2
67	C, 3
68	D, 4
69	E, 12
70	F, 2
71	G, 2
72	H, 6
73	I, 7
74	J, 1
75	K, 2
...	...

# Hash Codes and Index Calculation

## (cont.)

36

- However, what if all 65,536 Unicode characters were allowed?
- If you assume that on average 100 characters were used, you could use a table of 200 characters and compute the index by:

```
int index = unicode % 200;
```

# Hash Codes and Index Calculation

## (cont.)

37

- If a text contains this snippet:

. . . mañana (tomorrow), I'll finish my program. . .

- Given the following Unicode values:

Hexadecimal	Decimal	Name	Character
0x0029	41	right parenthesis	)
0x00F1	241	lower case n with tilde	ñ

- The indices for letters 'ñ' and ')' are both 41

$$41 \% 200 = 41 \text{ and } 241 \% 200 = 41$$

- This is called a *collision*; we will discuss how to deal with collisions shortly

# Methods for Generating Hash Codes

38

- In most applications, a key will consist of strings of letters or digits (such as a social security number, an email address, or a partial ID) rather than a single character
- The number of possible key values is much larger than the table size
- Generating good hash codes typically is an experimental process
- The goal is a *random (uniform) distribution of values*
- Simple algorithms sometimes generate lots of collisions

# Hash Function Examples

- Ideal goal: scramble the keys uniformly to produce a table index
  - ▣ Efficiently computable
  - ▣ Each table index equally likely for each key
- Phone numbers
  - ▣ Bad: first three digits
  - ▣ Better: last three digits
- Social Security numbers
  - ▣ Bad: first three digits
  - ▣ Better: last three digits

# Java HashCode Method

40

- For strings, simply summing the int values of all characters returns the same hash code for "sign" and "sing"
- The Java API algorithm accounts for position of the characters as well
- `String.hashCode()` returns the integer calculated by the formula:

$$s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \dots + s_{n-1}$$

where  $s_i$  is the  $i$ th character of the string, and  $n$  is the length of the string

- "Cat" has a hash code of:

$$'C' \times 31^2 + 'a' \times 31 + 't' = 67,510$$

- 31 is a prime number, and prime numbers generate relatively few collisions

# Java hashCode Method (cont.)

41

- Because there are too many possible strings, the integer value returned by `String.hashCode` cannot be unique
- However, because the `String.hashCode` method distributes the hash code values fairly evenly throughout the range, the probability of two strings having the same hash code is low
- The probability of a collision with  
 $s.hashCode() \% \text{table.length}$   
is proportional to how full the table is

# Methods for Generating Hash Codes

## (cont.)

42

- A good hash function should be relatively simple and efficient to compute
- It doesn't make sense to use an  $O(n)$  hash function to avoid doing an  $O(n)$  search

# Open Addressing

43

- We now consider two ways to organize hash tables:
  - open addressing
  - chaining
- In open addressing, *linear probing* can be used to access an item in a hash table
  - If the index calculated for an item's key is occupied by an item with that key, we have found the item
  - If that element contains an item with a different key, increment the index by one
  - Keep incrementing until you find the key or a null entry (assuming the table is not full)

# Open Addressing (cont.)

44

## Algorithm for Accessing an Item in a Hash Table

1. Compute the index by taking the item's hashCode() % table.length.
2. **if** table[index] is **null**
  3. The item is not in the table.
4. **else if** table[index] is equal to the item
  5. The item is in the table.
6. **else**
  6. Continue to search the table by incrementing the index until either the item is found or a **null** entry is found.

# Table Wraparound and Search Termination

45

- As you increment the table index, your table should wrap around as in a circular array
- This enables you to search the part of the table before the hash code value in addition to the part of the table after the hash code value
- But it could lead to an infinite loop
- How do you know when to stop searching if the table is full and you have not found the correct value?
  - ▣ Stop when the index value for the next probe is the same as the hash code value for the object
  - ▣ Ensure that the table is never full by increasing its size after an insertion when its load factor exceeds a specified threshold

# Hash Code Insertion Example

46

Tom Dick Harry Sam Pete

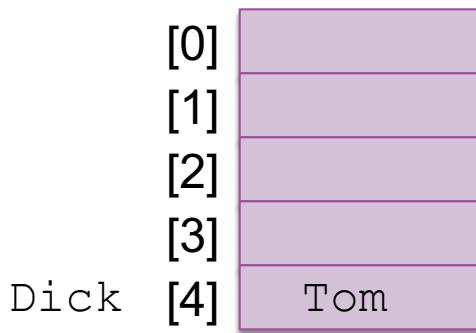


Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

47

Dick Harry Sam Pete

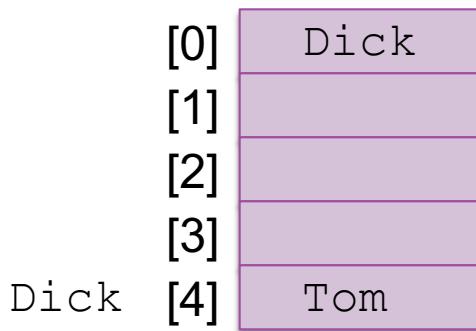


Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

48

Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

49

Harry Sam Pete

[0]	Dick
[1]	
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

50

	Sam Pete	Name	hashCode()	hashCode()%5
[0]	Dick	"Tom"	84274	4
[1]		"Dick"	2129869	4
[2]		"Harry"	69496448	3
[3]	Harry	"Sam"	82879	4
Sam [4]	Tom	"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

51

		Pete	Name	hashCode()	hashCode()%5
Sam	[0]	Dick	"Tom"	84274	4
	[1]		"Dick"	2129869	4
	[2]		"Harry"	69496448	3
Sam	[3]	Harry	"Sam"	82879	4
	[4]	Tom	"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

52

		Pete	Name	hashCode()	hashCode()%5
Sam	[0]	Dick	"Tom"	84274	4
	[1]	Sam	"Dick"	2129869	4
	[2]		"Harry"	69496448	3
	[3]	Harry	"Sam"	82879	4
	[4]	Tom	"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

53

Pete	Name	hashCode()	hashCode()%5
	"Tom"	84274	4
	"Dick"	2129869	4
	"Harry"	69496448	3
[0] Dick	"Sam"	82879	4
[1] Sam	"Pete"	2484038	3
[2]			
Pete [3] Harry			
[4] Tom			

# Hash Code Insertion Example (cont.)

54

Pete	[0]	Dick
	[1]	Sam
	[2]	
	[3]	Harry
	[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

55

Pete

[0]	Dick
[1]	Sam
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

56

Pete	[0]	Dick
	[1]	Sam
	[2]	
	[3]	Harry
	[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

# Hash Code Insertion Example (cont.)

57

Pete	[0]	Dick
	[1]	Sam
	[2]	Pete
	[3]	Harry
	[4]	Tom

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Retrieval of "Tom" or "Harry" takes one step,  
 $O(1)$

Because of collisions, retrieval of the others  
requires a linear search

# Hash Code Insertion Example (cont.)

58

Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7



# Hash Code Insertion Example (cont.)

59

Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7

The best way to reduce the possibility of collision (and reduce linear search retrieval time because of collisions) is to increase the table size



Only one collision occurred

# Traversing a Hash Table

60

- You cannot traverse a hash table in a meaningful way since the sequence of stored values is arbitrary

[0]	Dick
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

Dick, Sam, Pete, Harry, Tom

[0]	
[1]	
[2]	
[3]	Tom
[4]	
[5]	Dick
[6]	Sam
[7]	Pete
[8]	
[9]	
[10]	Harry

Tom, Dick, Sam,  
Pete, Harry

# Deleting an Item Using Open Addressing

61

- When an item is deleted, you cannot simply set its table entry to null
- If we search for an item that may have collided with the deleted item, we may conclude incorrectly that it is not in the table.
- Instead, store a dummy value or mark the location as available, but previously occupied
- Deleted items waste storage space and reduce search efficiency unless they are marked as available

# Reducing Collisions by Expanding the Table Size

62

- Use a prime number for the size of the table to reduce collisions
- A fuller table results in more collisions, so, when a hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted
- You must reinsert (*rehash*) values into the new table; do not copy values as some search chains which were wrapped may break
- Deleted items are not reinserted, which saves space and reduces the length of some search chains

# Reducing Collisions Using Quadratic Probing

63

- Linear probing tends to form clusters of keys in the hash table, causing longer search chains
- Quadratic probing can reduce the effect of clustering
  - ▣ Increments form a quadratic series ( $1 + 2^2 + 3^2 + \dots$ )

```
probeNum++;  
index = (startIndex + probeNum * probeNum) %  
table.length
```

- If an item has a hash code of 5, successive values of index will be 6 (5+1), 9 (5+4), 14 (5+9), . . .

# Problems with Quadratic Probing

64

- The disadvantage of quadratic probing is that the next index calculation is time-consuming, involving multiplication, addition, and modulo division
- A more efficient way to calculate the next index is:

```
k += 2;
```

```
index = (index + k) % table.length;
```

# Problems with Quadratic Probing (cont.)

65

- Examples:
  - If the initial value of k is -1, successive values of k will be 1, 3, 5, ...
  - If the initial value of index is 5, successive value of index will be 6 ( $= 5 + 1$ ), 9 ( $= 5 + 1 + 3$ ), 14 ( $= 5 + 1 + 3 + 5$ ), ...
- The proof of the equality of these two calculation methods is based on the mathematical series:

$$n^2 = 1 + 3 + 5 + \dots + 2n - 1$$

# Problems with Quadratic Probing (cont.)

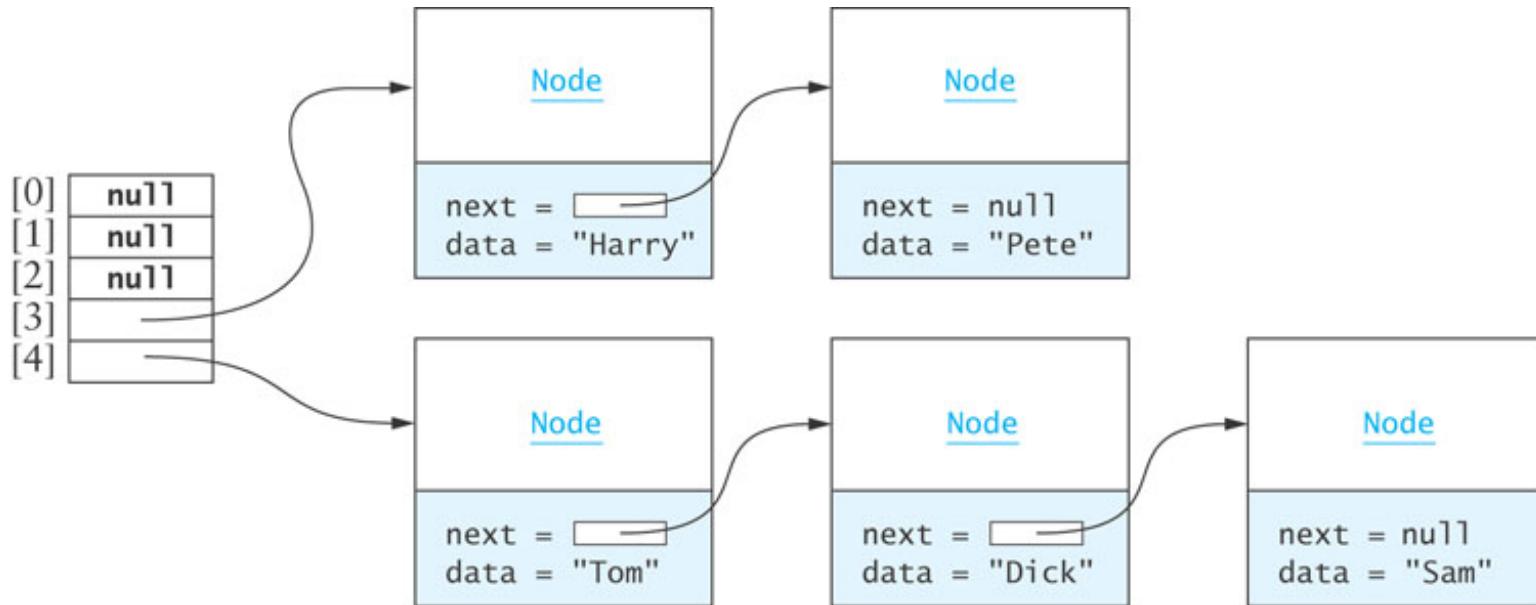
66

- A more serious problem is that not all table elements are examined when looking for an insertion index; this may mean that
  - ▣ an item can't be inserted even when the table is not full
  - ▣ the program will get stuck in an infinite loop searching for an empty slot
- If the table size is a prime number and it is never more than half full, this won't happen
- However, requiring a half empty table wastes a lot of memory

# Chaining

67

- Chaining is an alternative to open addressing
- Each table element references a linked list that contains all of the items that hash to the same table index
  - ▣ The linked list often is called a *bucket*
  - ▣ The approach sometimes is called *bucket hashing*



# Chaining (cont.)

68

- Advantages relative to open addressing:
  - ▣ Only items that have the same value for their hash codes are examined when looking for an object
  - ▣ You can store more elements in the table than the number of table slots (indices)
  - ▣ Once you determine an item is not present, you can insert it at the beginning or end of the list
  - ▣ To remove an item, you simply delete it; you do not need to replace it with a dummy item or mark it as deleted

# Performance of Hash Tables

69

- *Load factor* is the number of filled cells divided by the table size
- Load factor has the greatest effect on hash table performance
- The lower the load factor, the better the performance as there is a smaller chance of collision when a table is sparsely populated
- If there are no collisions, performance for search and retrieval is  $O(1)$  regardless of table size

# Performance of Open Addressing versus Chaining

70

- Donald E. Knuth derived the following formula for the expected number of comparisons,  $c$ , required for finding an item that is in a hash table using open addressing with linear probing and a load factor  $L$

$$c = \frac{1}{2} \left( 1 + \frac{1}{1 - L} \right)$$

# Performance of Open Addressing versus Chaining (cont.)

71

- Using chaining, if an item is in the table, on average we must examine the table element corresponding to the item's hash code and then half of the items in each list
- The average number of items in a list is  $L$ , the number of items divided by the table size

$$c = 1 + \frac{L}{2}$$

# Performance of Open Addressing versus Chaining (cont.)

72

<i>L</i>	Number of Probes with Linear Probing	Number of Probes with Chaining
0.0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

# Performance of Hash Tables versus Sorted Array and Binary Search Tree

73

- The number of comparisons required for a binary search of a sorted array is  $O(\log n)$ 
  - ▣ A sorted array of size 128 requires up to 7 probes ( $2^7$  is 128) which is more than for a hash table of any size that is 90% full
  - ▣ A binary search tree performs similarly
- Insertion or removal

hash table	$O(1)$ expected; worst case $O(n)$
sorted array	$O(n)$
binary search tree	$O(\log n)$ ; worst case $O(n)$

# Storage Requirements for Hash Tables, Sorted Arrays, and Trees

74

- The performance of hashing is superior to that of binary search of an array or a binary search tree, particularly if the load factor is less than 0.75
- However, the lower the load factor, the more empty storage cells
  - there are no empty cells in a sorted array
- A binary search tree requires three references per node (item, left subtree, right subtree), so more storage is required for a binary search tree than for a hash table with load factor 0.75

# Storage Requirements for Open Addressing and Chaining

75

- For open addressing, the number of references to items (key-value pairs) is  $n$  (the size of the table)
- For chaining , the average number of nodes in a list is  $L$  (the load factor) and  $n$  is the number of table elements
  - ▣ Using the Java API `LinkedList`, there will be three references in each node (item, next, previous)
  - ▣ Using our own single linked list, we can reduce the references to two by eliminating the previous-element reference
  - ▣ Therefore, storage for  $n + 2L$  references is needed

# Storage Requirements for Open Addressing and Chaining (cont.)

76

- Example:
  - Assume open addressing, 60,000 items in the hash table, and a load factor of 0.75
  - This requires a table of size 80,000 and results in an expected number of comparisons of 2.5
  - Calculating the table size  $n$  to get similar performance using chaining

$$2.5 = 1 + L/2$$

$$5.0 = 2 + L$$

$$3.0 = 60,000/n$$

$$n = 20,000$$

# Storage Requirements for Open Addressing and Chaining (cont.)

77

- A hash table of size 20,000 provides storage space for 20,000 references to lists
- There are 60,000 nodes in the table (one for each item)
- This requires storage for 140,000 references ( $2 \times 60,000 + 20,000$ ), which is 175% of the storage needed for open addressing



# CS 570: Data Structures

## Sets and Maps (Part 2)

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 7 (PART 2)

# Week 13

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 7.4-7.5

# Implementing the Hash Table

## Section 7.4

# Interface KHashMap

5

Method	Behavior
V get(Object key)	Returns the value associated with the specified key. Returns <b>null</b> if the key is not present.
boolean isEmpty()	Returns <b>true</b> if this table contains no key-value mappings.
V put(K key, V value)	Associates the specified value with the specified key. Returns the previous value associated with the specified key, or <b>null</b> if there was no mapping for the key.
V remove(Object key)	Removes the mapping for this key from this table if it is present (optional operation). Returns the previous value associated with the specified key, or <b>null</b> if there was no mapping.
int size()	Returns the size of the table.

# Class Entry

6

Data Field	Attribute
private K key	The key.
private V value	The value.
Constructor	Behavior
public Entry(K key, V value)	Constructs an Entry with the given values.
Method	Behavior
public K getKey()	Retrieves the key.
public V getValue()	Retrieves the value.
public V setValue(V val)	Sets the value.

# Class Entry (cont.)

7

```
/** Contains key-value pairs for a hash table. */
private static class Entry < K, V > {

    /** The key */
    private K key;

    /** The value */
    private V value;

    /** Creates a new key-value pair.
        @param key The key
        @param value The value
    */
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

# Class Entry (cont.)

8

```
/** Retrieves the key.  
 * @return The key  
 */  
public K getKey() {  
    return key;  
}  
  
/** Retrieves the value.  
 * @return The value  
 */  
public V getValue() {  
    return value;  
}
```

# Class Entry (cont.)

9

```
/** Sets the value.  
 * @param val The new value  
 * @return The old value  
 */  
public V setValue(V val) {  
    V oldVal = value;  
    value = val;  
    return oldVal;  
}  
}
```

# Class HashTableOpen

10

Data Field	Attribute
private Entry<K, V>[] table	The hash table array.
private static final int START_CAPACITY	The initial capacity.
private double LOAD_THRESHOLD	The maximum load factor.
private int numKeys	The number of keys in the table excluding keys that were deleted.
private int numDeletes	The number of deleted keys.
private final Entry<K, V> DELETED	A special object to indicate that an entry has been deleted.

# Class HashTableOpen

11

```
/** Hash table implementation using open addressing. */
public class HashtableOpen<K, V> implements KWHashMap<K, V> {
    // Data Fields
    private Entry<K, V>[] table;
    private static final int START_CAPACITY = 101;
    private double LOAD_THRESHOLD = 0.75;
    private int numKeys;
    private int numDeletes;
    private final Entry<K, V> DELETED =
        new Entry<K, V>(null, null);

    // Constructor
    public HashTableOpen() {
        table = new Entry[START_CAPACITY];
    }

    // Insert inner class Entry<K, V> here.
    . . .
```

# Class HashTableOpen (cont.)

12

Method	Behavior
<code>private int find(Object key)</code>	Returns the index of the specified key if present in the table; otherwise, returns the index of the first available slot.
<code>private void rehash()</code>	Doubles the capacity of the table and permanently removes deleted items.

## Algorithm for `HashtableOpen.find(Object key)`

1. Set `index` to `key.hashCode() % table.length`.
2. if `index` is negative, add `table.length`.
3. while `table[index]` is not empty and the key is not at `table[index]`
4.     increment `index`.
5.     if `index` is greater than or equal to `table.length`
6.         Set `index` to 0.
7. Return the `index`.

# Class HashTableOpen (cont.)

13

```
/** Finds either the target key or the first empty slot in
 * the search chain using linear probing.
 * pre: The table is not full.
 * @param key The key of the target object
 * @return The position of the target or the first empty
 *         slot if the target is not in the table.
 */
private int find(Object key) {
    // Calculate the starting index.
    int index = key.hashCode() % table.length;
    if (index < 0)
        index += table.length; // Make it positive.
```

# Class HashTableOpen (cont.)

14

```
// Increment index until an empty slot is reached
// or the key is found.
while ( (table[index] != null)
        && (!key.equals(table[index].key))) {
    index++;
    // Check for wraparound.
    if (index >= table.length)
        index = 0; // Wrap around.
}
return index;
}
```

# Class HashTableOpen (cont.)

15

## Algorithm for get (Object key)

1. Find the first table element that is empty or the table element that contains the key.
2. if the table element found contains the key  
    return the value at this table element.
3. else
4.     return null.

# Class HashTableOpen (cont.)

16

```
/** Method get for class HashtableOpen.  
 * @param key The key being sought  
 * @return the value associated with this key if found;  
 *         otherwise, null  
 */  
public V get(Object key) {  
    // Find the first table element that is empty  
    // or the table element that contains the key.  
    int index = find(key);  
  
    // If the search is successful, return the value.  
    if (table[index] != null)  
        return table[index].value;  
    else  
        return null; // key not found.  
}
```

# Class HashTableOpen (cont.)

17

## Algorithm for `HashtableOpen.put(K key, V value)`

1. Find the first table element that is empty or the table element that contains the key.
2. if an empty element was found
3.     insert the new item and increment numKeys.
4.     check for need to rehash.
5.     return null.
6. The key was found. Replace the value associated with this table element and return the old value.

# Class HashTableOpen (cont.)

18

```
/** Method put for class HashtableOpen.  
 * post: This key-value pair is inserted in the table and  
 * numKeys is incremented. If the key is already in  
 * the table, its value is changed to the argument  
 * value and numKeys is not changed. If the  
 * LOAD_THRESHOLD is exceeded, the table is expanded.  
 * @param key The key of item being inserted  
 * @param value The value for this key  
 * @return Old value associated with this key if found;  
 * otherwise, null  
 */
```

# Class HashTableOpen (cont.)

19

```
public V put(K key, V value) {  
    // Find the first table element that is empty  
    // or the table element that contains the key.  
    int index = find(key);  
    // If an empty element was found, insert new entry.  
    if (table[index] == null) {  
        table[index] = new Entry < K, V > (key, value);  
        numKeys++;
```

# Class HashTableOpen (cont.)

20

```
// Check whether rehash is needed.  
double loadFactor =  
    (double) (numKeys + numDeletes) / table.length;  
if (loadFactor > LOAD_THRESHOLD)  
    rehash();  
return null;  
}  
  
// assert: table element that contains the key was found.  
// Replace value for this key.  
v oldVal = table[index].value;  
table[index].value = value;  
return oldVal;  
}
```

# Class HashTableOpen (cont.)

21

## Algorithm for `remove (Object key)`

1. Find the first table element that is empty or the table element that contains the key.
2. if an empty element was found
3.     return null.
4. Key was found. Remove this table element by setting it to reference DELETED, increment numDeletes, and decrement numKeys.
5. Return the value associated with this key.

# Class HashTableOpen (cont.)

22

## Algorithm for HashtableOpen. rehash

1. Allocate a new hash table that is at least double the size and has an odd length.
2. Reset the number of keys and number of deletions to 0.
3. Reinsert each table entry that has not been deleted in the new hash table.

# Class HashTableOpen (cont.)

23

```
private void rehash() {  
    // Save a reference to oldTable.  
    Entry < K, V > [] oldTable = table;  
    // Double capacity of this table.  
    table = new Entry[2 * oldTable.length + 1];  
  
    // Reinsert all items in oldTable into expanded table.  
    numKeys = 0;  
    numDeletes = 0;  
    for (int i = 0; i < oldTable.length; i++) {  
        if ( (oldTable[i] != null) && (oldTable[i] != DELETED) ) {  
            // Insert entry in expanded table  
            put(oldTable[i].key, oldTable[i].value);  
        }  
    }  
}
```

# Class HashTableChain

24

Data Field	Attribute
private LinkedList<Entry<K, V>>[] table	A table of references to linked lists of Entry<K, V> objects.
private int numKeys	The number of keys (entries) in the table.
private static final int CAPACITY	The size of the table.
private static final int LOAD_THRESHOLD	The maximum load factor.

# Class HashTableChain (cont.)

25

```
/** Hash table implementation using chaining.  
 *  @author Koffman and Wolfgang  
 * */  
  
public class HashtableChain < K, V >  
    implements KWHashMap < K, V > {  
    /** The table */  
    private LinkedList < Entry < K, V >> [] table;  
  
    /** The number of keys */  
    private int numKeys;  
  
    /** The capacity */  
    private static final int CAPACITY = 101;
```

# Class HashTableChain (cont.)

26

```
/** The maximum load factor */
private static final double LOAD_THRESHOLD = 3.0;

/** Insert class Entry < K, V > here */

// Constructor
public HashtableChain() {
    table = new LinkedList[CAPACITY];
}

/** Returns the number of entries in the map */
public int size() {
    return numKeys;
}

/** Returns true if empty */
public boolean isEmpty() {
    return numKeys == 0;
}
```

# Class HashTableChain (cont.)

27

## Algorithm for HashtableChain.get(Object key)

1. Set index to key.hashCode() % table.length.
2. if index is negative  
3.     add table.length.
4. if table[index] is null  
5.     key is not in the table; return null.
6. For each element in the list at table[index]  
7.     if that element's key matches the search key  
8.         return that element's value.
9. key is not in the table; return null.

# Class HashTableChain (cont.)

28

```
/** Method get for class HashtableChain.  
 * @param key The key being sought  
 * @return The value associated with this key if found;  
 *         otherwise, null  
 */  
  
public V get(Object key) {  
    int index = key.hashCode() % table.length;  
    if (index < 0)  
        index += table.length;  
    if (table[index] == null)  
        return null; // key is not in the table.
```

# Class HashTableChain (cont.)

29

```
// Search the list at table[index] to find the key.  
for (Entry < K, V > nextItem : table[index]) {  
    if (nextItem.key.equals(key))  
        return nextItem.value;  
}  
  
// assert: key is not in the table.  
return null;  
}
```

# Class HashTableChain (cont.)

30

## Algorithm for `HashtableChain.put(K key, V value)`

1. Set index to `key.hashCode() % table.length.`
2. if index is negative, add `table.length.`
3. if `table[index]` is null
4.     create a new linked list at `table[index].`
5. Search the list at `table[index]` to find the key.
6. if the search is successful
7.     replace the value associated with this key.
8.     return the old value.
9. else
10.     insert the new key-value pair in the linked list located at `table[index].`
11.     increment `numKeys.`
12.     if the load factor exceeds the `LOAD_THRESHOLD`
13.         Rehash.
14.     return `null.`

# Class HashTableChain (cont.)

31

```
/** Method put for class HashtableChain.  
 * post: This key-value pair is inserted in the  
 *        table and numKeys is incremented. If the key is  
 *        already in the table, its value is changed to the  
 *        argument value and numKeys is not changed.  
 * @param key The key of item being inserted  
 * @param value The value for this key  
 * @return The old value associated with this key if  
 *         found; otherwise, null  
 */  
  
public V put(K key, V value) {  
    int index = key.hashCode() % table.length;  
    if (index < 0)  
        index += table.length;
```

# Class HashTableChain (cont.)

32

```
if (table[index] == null) {  
    // Create a new linked list at table[index].  
    table[index] = new LinkedList < Entry < K, V >> ();  
}  
  
// Search the list at table[index] to find the key.  
for (Entry < K, V > nextItem : table[index]) {  
    // If the search is successful, replace the old value.  
    if (nextItem.key.equals(key)) {  
        // Replace value for this key.  
        V oldVal = nextItem.value;  
        nextItem.setValue(value);  
        return oldVal;  
    }  
}
```

# Class HashTableChain (cont.)

33

```
// assert: key is not in the table, add new item.  
table[index].addFirst(new Entry < K, V > (key, value));  
numKeys++;  
if (numKeys > (LOAD_THRESHOLD * table.length))  
    rehash();  
return null;  
}
```

# Class HashTableChain (cont.)

34

## Algorithm for `HashtableChain.remove(Object key)`

1. Set `index` to `key.hashCode() % table.length.`
2. if `index` is negative, add `table.length.`
3. if `table[index]` is null
4.     key is not in the table; return null.
5. Search the list at `table[index]` to find the key.
6. if the search is successful
7.     remove the entry with this key and decrement `numKeys.`
8.     if the list at `table[index]` is empty
9.         Set `table[index]` to null.
10.       return the value associated with this key.
11. The key is not in the table; return null.

# Testing the Hash Table Implementation

35

- Write a method to
  - create a file of key-value pairs
  - read each key-value pair and insert it in the hash table
  - observe how the hash table is filled
- Implementation
  - Write a `toString` method that captures the index of each non-null **table** element and the contents of the table element
  - For open addressing, the contents is the string representation of the key-value pair
  - For chaining, a list iterator can traverse at the table element and append each key-value pair to the resulting string

# Testing the Hash Table Implementation (cont.)

36

- Cases to examine:
  - Does the array index wrap around as it should?
  - Are collisions resolved correctly?
  - Are duplicate keys handled appropriately? Is the new value retrieved instead of the original value?
  - Are deleted keys retained in the table but no longer accessible via a get?
  - Does rehashing occur when the load factor reaches 0.75 (3.0 for chaining)?
- Step through the get and put methods to
  - observe how the table is probed
  - examine the search chain followed to access or retrieve a key

# Testing the Hash Table Implementation

## (cont.)

37

- Alternatively, insert randomly generated integers in the hash table to create a large table with  $O(n)$  effort

```
for (int i = 0; i < SIZE; i++) {  
    Integer nextInt = (int) (32000 * Math.random());  
    hashTable.put(nextInt, nextInt);  
}
```

# Testing the Hash Table Implementation

## (cont.)

38

- Insertion of randomly generated integers into a table allows testing of tables of very large sizes, but is less helpful for testing for collisions
- You can add code to count the number of items probed each time an insertion is made—these can be totaled to determine the average search chain length
- After all items are inserted, you can calculate the average length of each linked list and compare that with the number predicted by the formula discussed in section 7.3

# Implementation Considerations for Maps and Sets

## Section 7.5

# Methods hashCode and equals

40

- Class Object implements methods hashCode and equals, so every class can access these methods unless it overrides them
- Object.equals compares two objects based on their addresses, not their contents
- Most predefined classes override method equals and compare objects based on content
- If you want to compare two objects (whose classes you've written) for equality of content, you need to override the equals method

# Methods hashCode and equals (cont.)

41

- `Object.hashCode` calculates an object's hash code based on its address, not its contents
- Most predefined classes also override method `hashcode`
- Java recommends that if you override the `equals` method, then you should also override the `hashCode` method
- Otherwise, you violate the following rule:  
If `obj1.equals(obj2)` is true,  
then `obj1.hashCode == obj2.hashCode`

# Methods hashCode and equals (cont.)

42

- Make sure your hashCode method uses the same data field(s) as your equals method

# Note

43

- The second part of Section 7.5 and Section 7.6, and 7.7 are out of scope for CS 570



# CS 570: Data Structures

## Sorting

*Instructor: Iraklis Tsekourakis*

Email: [itsekour@stevens.edu](mailto:itsekour@stevens.edu)



# CHAPTER 8

# SORTING

# Chapter Objectives

3

- To learn how to implement the following sorting algorithms:
  - ▣ selection sort
  - ▣ insertion sort
  - ▣ merge sort
  - ▣ quicksort
- To understand the differences in performance of these algorithms, and which to use for small, medium arrays, and large arrays

# Week 14

---

- Reading Assignment: Koffman and Wolfgang,  
Sections 8.2, 8.3, 8.6, 8.9

# Introduction

5

- Sorting entails arranging data in order
- Familiarity with sorting algorithms is an important programming skill
- The study of sorting algorithms provides insight
  - ▣ into problem solving techniques such as *divide and conquer*
  - ▣ into the analysis and comparison of algorithms which perform the same task

# Using Java Sorting Methods

6

- The Java API provides a class `Arrays` with several overloaded sort methods for different array types
- The Collections class provides similar sorting methods for Lists
- Sorting methods for arrays of primitive types are based on the quicksort algorithm
- Sorting methods for arrays of objects and Lists are based on the merge sort algorithm
- Both algorithms are  $O(n \log n)$

# Selection Sort

## Section 8.2

# Selection Sort

8

- Selection sort is relatively easy to understand
- It sorts an array by making several passes through the array, selecting a next smallest item in the array each time and placing it where it belongs in the array
  - ▣ While the sort algorithms are not limited to arrays, throughout this chapter we will sort arrays for simplicity
- All items to be sorted must be Comparable objects, so, for example, any int values must be wrapped in Integer objects

# Trace of Selection Sort

9

$n$  = number of elements in the array

1. **for**  $fill = 0$  to  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$

0	1	2	3	4
35	65	30	60	20

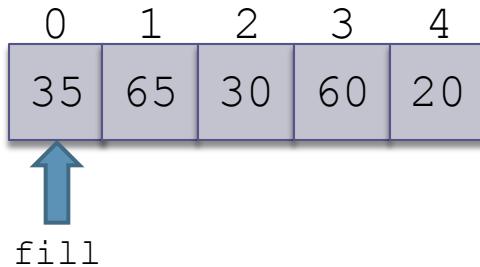
$n$	5
$fill$	
$posMin$	

# Trace of Selection Sort (cont.)

10

$n$  = number of elements in the array

- 1. **for**  $fill = 0$  to  $n - 2$  **do**
- 2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
- 3. Exchange the item at  $posMin$  with the one at  $fill$



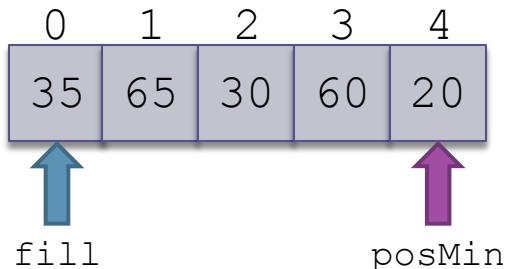
$n$	5
$fill$	0
$posMin$	

# Trace of Selection Sort (cont.)

11

$n$  = number of elements in the array

1. **for**  $fill = 0$  to  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



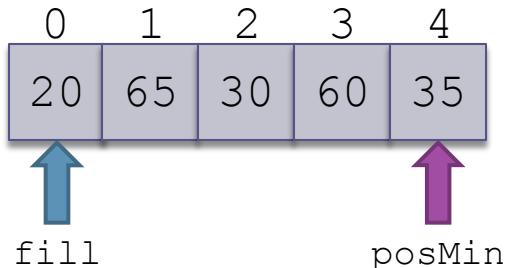
$n$	5
$fill$	0
$posMin$	4

# Trace of Selection Sort (cont.)

12

$n$  = number of elements in the array

1. **for**  $fill = 0$  to  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



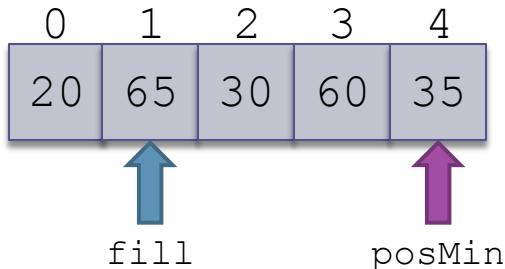
$n$	5
$fill$	0
$posMin$	4

# Trace of Selection Sort (cont.)

13

$n$  = number of elements in the array

- 1. **for**  $fill = 0$  to  $n - 2$  **do**
- 2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
- 3. Exchange the item at  $posMin$  with the one at  $fill$



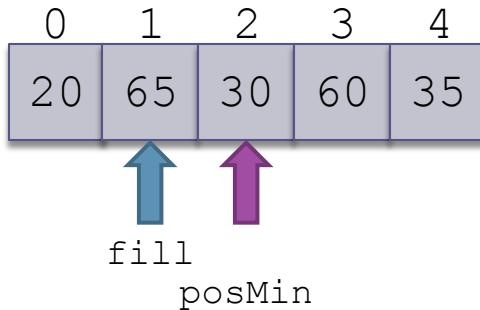
$n$	5
$fill$	1
$posMin$	4

# Trace of Selection Sort (cont.)

14

$n$  = number of elements in the array

1. **for**  $fill = 0$  **to**  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



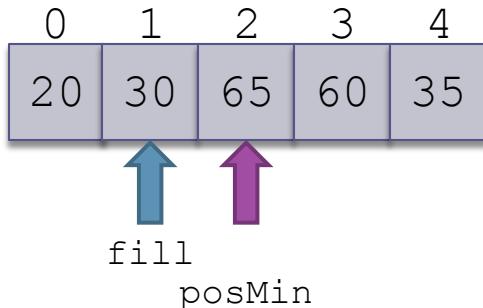
$n$	5
$fill$	1
$posMin$	2

# Trace of Selection Sort (cont.)

15

$n$  = number of elements in the array

1. **for**  $fill = 0$  to  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



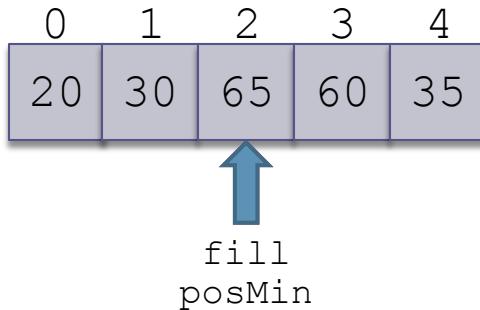
$n$	5
$fill$	1
$posMin$	2

# Trace of Selection Sort (cont.)

16

$n$  = number of elements in the array

- 1. **for**  $fill = 0$  to  $n - 2$  **do**
- 2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
- 3. Exchange the item at  $posMin$  with the one at  $fill$



$n$	5
$fill$	2
$posMin$	2

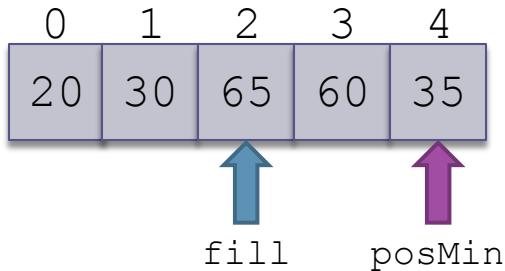
# Trace of Selection Sort (cont.)

17

$n$  = number of elements in the array

1. **for**  $fill = 0$  **to**  $n - 2$  **do**

- 2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



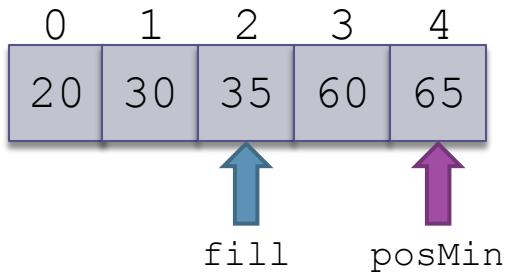
$n$	5
$fill$	2
$posMin$	4

# Trace of Selection Sort (cont.)

18

$n$  = number of elements in the array

1. **for**  $fill = 0$  to  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



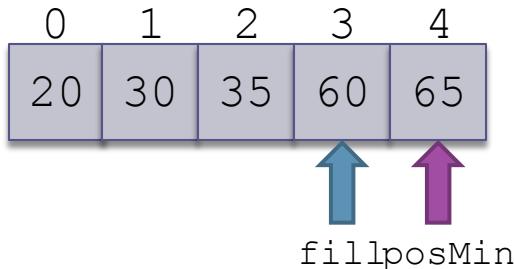
$n$	5
$fill$	2
$posMin$	4

# Trace of Selection Sort (cont.)

19

$n$  = number of elements in the array

- 1. **for**  $fill = 0$  to  $n - 2$  **do**
- 2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
- 3. Exchange the item at  $posMin$  with the one at  $fill$



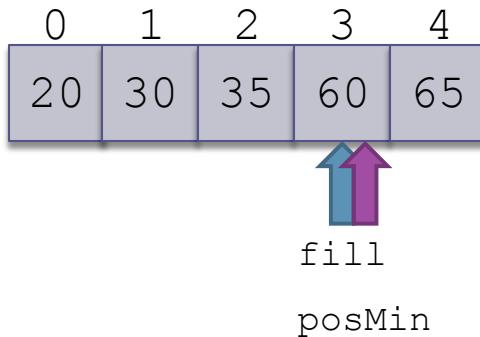
$n$	5
$fill$	3
$posMin$	4

# Trace of Selection Sort (cont.)

20

$n$  = number of elements in the array

1. **for**  $fill = 0$  to  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



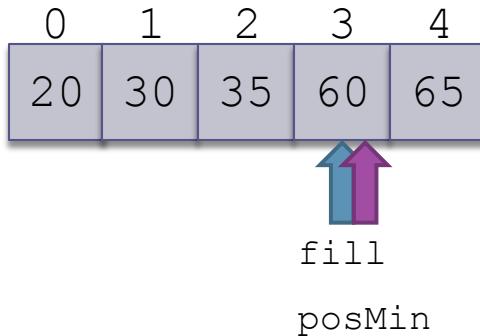
$n$	5
$fill$	3
$posMin$	3

# Trace of Selection Sort (cont.)

21

$n$  = number of elements in the array

1. **for**  $fill = 0$  **to**  $n - 2$  **do**
2. Set  $posMin$  to the subscript of the smallest item in the subarray starting at subscript  $fill$
3. Exchange the item at  $posMin$  with the one at  $fill$



$n$	5
$fill$	3
$posMin$	3

# Trace of Selection Sort (cont.)

22

$n$  = number of elements in the array

1. **for**  $\text{fill} = 0$  to  $n - 2$  **do**
2. Set  $\text{posMin}$  to the subscript of the smallest item in the subarray starting at subscript  $\text{fill}$
3. Exchange the item at  $\text{posMin}$  with the one at  $\text{fill}$

0	1	2	3	4
20	30	35	60	65

$n$	5
$\text{fill}$	3
$\text{posMin}$	3

# Trace of Selection Sort Refinement

23

n	5
fill	
posMin	
next	

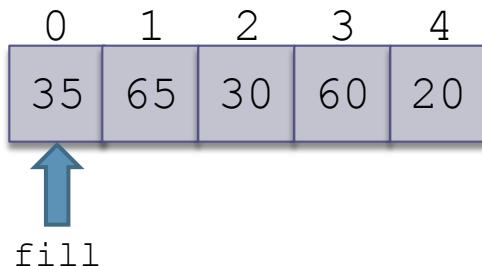
0	1	2	3	4
35	65	30	60	20

1. **for** fill = 0 **to** n - 2 **do**
2.     Initialize posMin to fill
3.     **for** next = fill + 1 **to** n - 1 **do**
4.         **if** the item at next is less than the item at posMin
5.             Reset posMin to next
6.     Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

24

n	5
fill	0
posMin	
next	

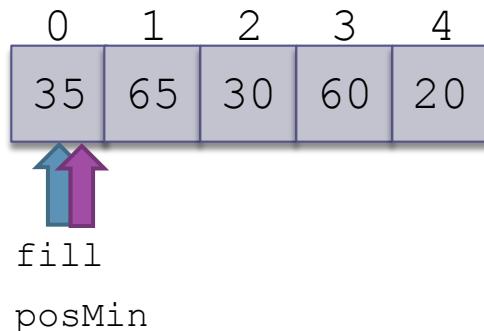


- 1. **for** fill = 0 to n - 2 do
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1 do
- 4.     **if** the item at next is less than the item at posMin
- 5.         Reset posMin to next
- 6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

25

n	5
fill	0
posMin	0
next	

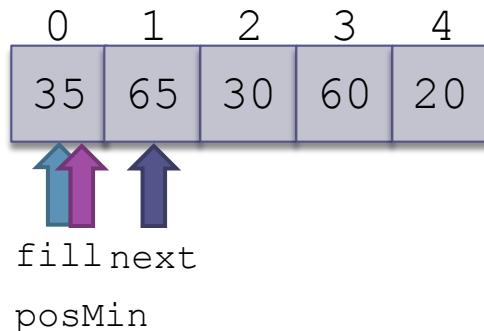


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

26

n	5
fill	0
posMin	0
next	1

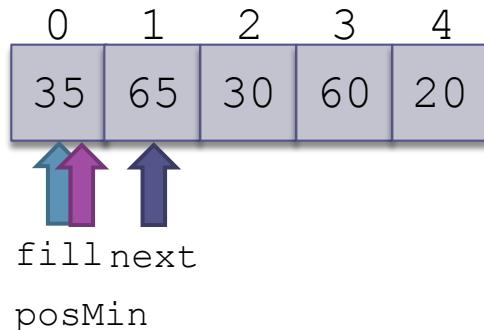


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

27

n	5
fill	0
posMin	0
next	1

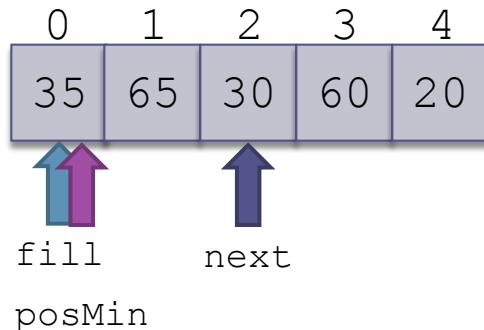


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

28

n	5
fill	0
posMin	0
next	2

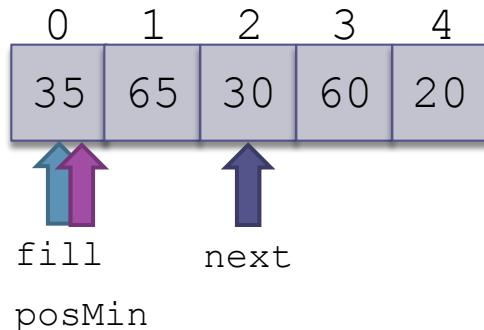


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement

(cont.)

n	5
fill	0
posMin	0
next	2

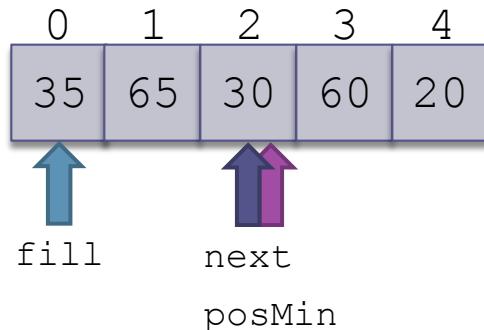


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4. **if** the item at next is less than the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

30

n	5
fill	0
posMin	2
next	2

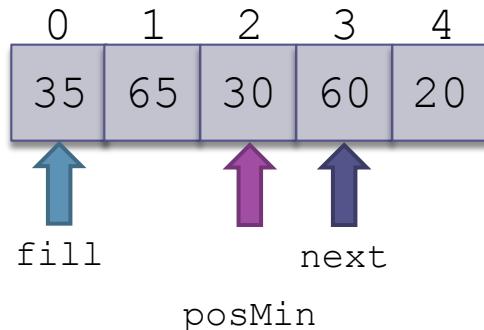


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

31

n	5
fill	0
posMin	2
next	3

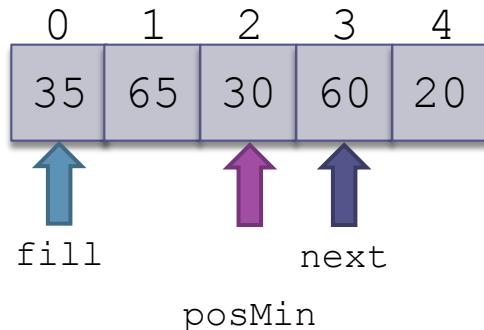


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

32

n	5
fill	0
posMin	2
next	3

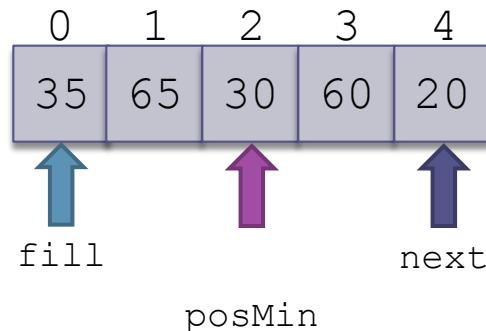


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

33

n	5
fill	0
posMin	2
next	4

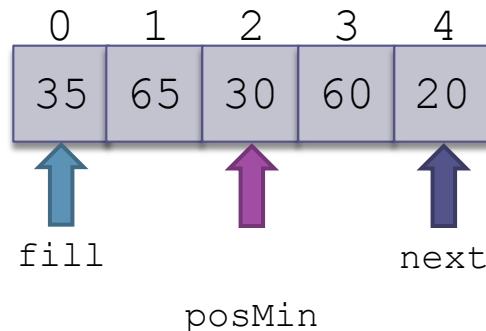


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

34

n	5
fill	0
posMin	2
next	4

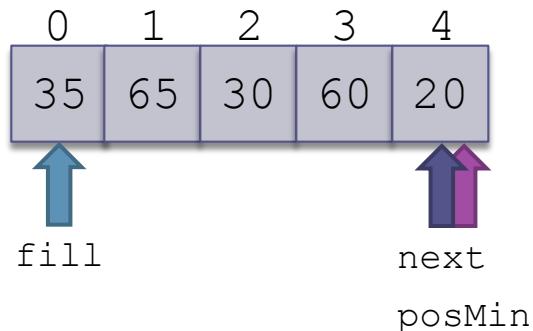


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

35

n	5
fill	0
posMin	4
next	4

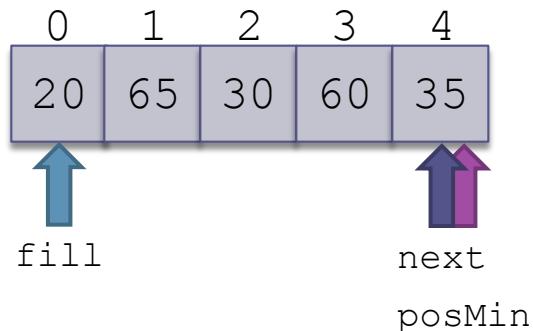


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

36

n	5
fill	0
posMin	4
next	4

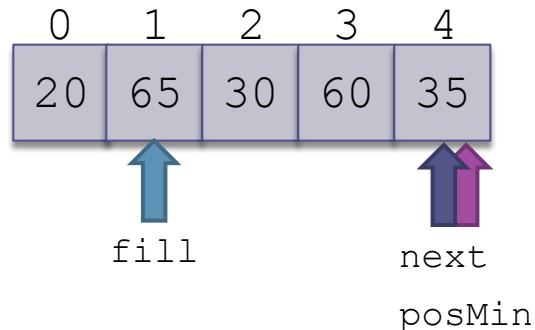


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

37

n	5
fill	1
posMin	4
next	4

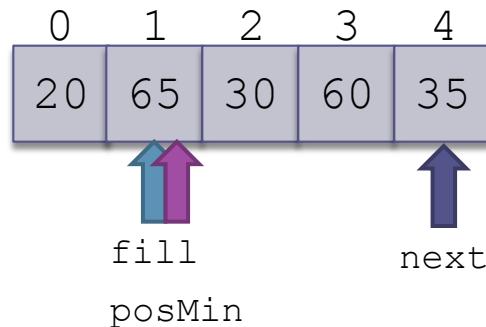


- 1. **for** fill = 0 to n - 2 do
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1 do
- 4.     **if** the item at next is less than the item at posMin
- 5.         Reset posMin to next
- 6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

38

n	5
fill	1
posMin	1
next	4

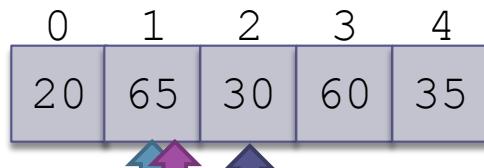


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

39

n	5
fill	1
posMin	1
next	2



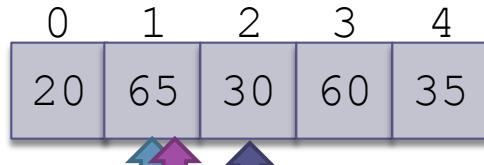
fill next  
posMin

1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

40

n	5
fill	1
posMin	1
next	2



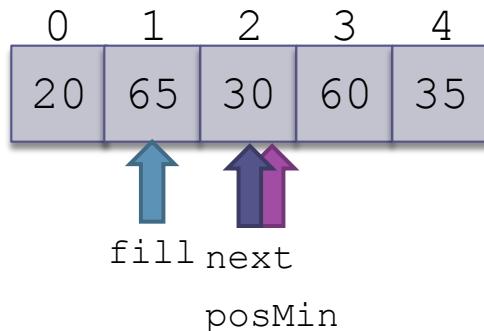
fill next  
posMin

1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

41

n	5
fill	1
posMin	2
next	2

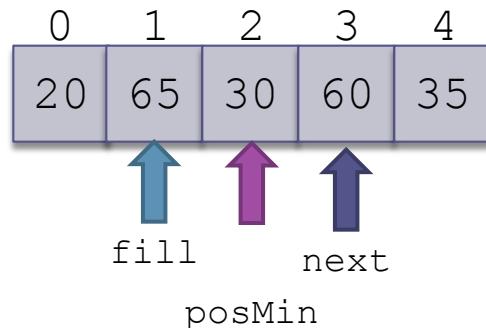


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

42

n	5
fill	1
posMin	2
next	3

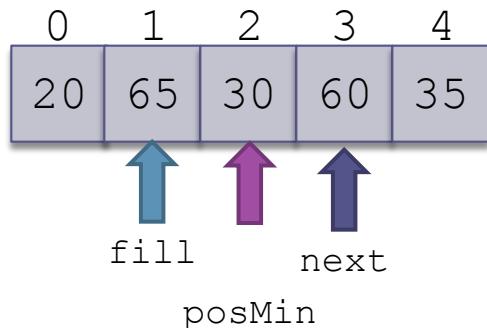


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

43

n	5
fill	1
posMin	2
next	3

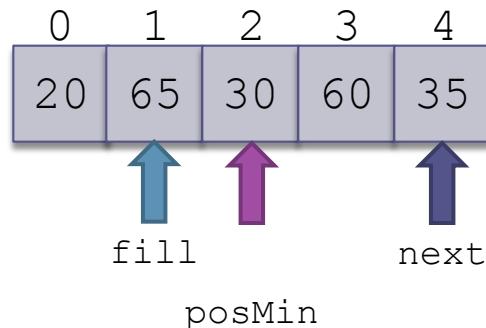


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

44

n	5
fill	1
posMin	2
next	4

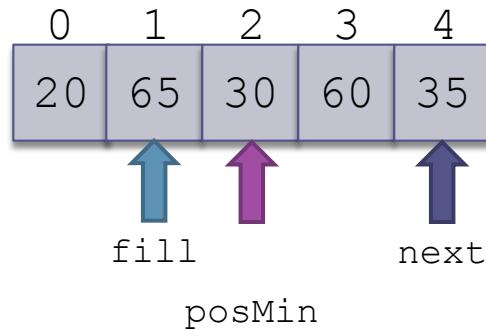


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

45

n	5
fill	1
posMin	2
next	4

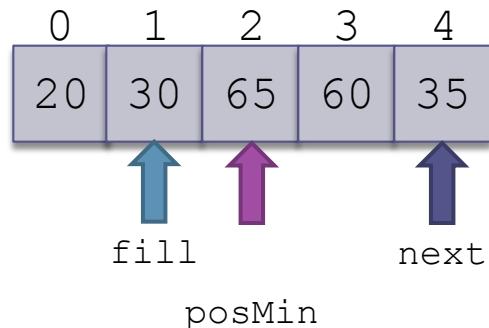


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

46

n	5
fill	1
posMin	2
next	4

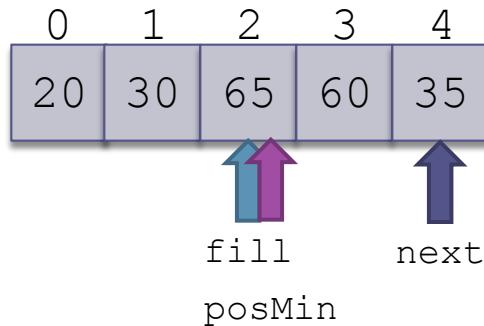


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

47

n	5
fill	2
posMin	2
next	4

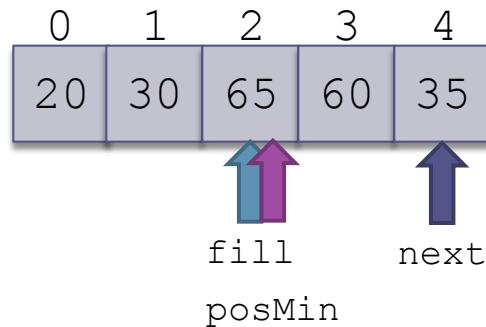


- 1. **for** fill = 0 to n - 2 do
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1 do
- 4.     **if** the item at next is less than the item at posMin
- 5.         Reset posMin to next
- 6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

48

n	5
fill	2
posMin	2
next	4



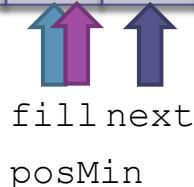
1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement

## (cont.)

49

n	5
fill	2
posMin	2
next	3



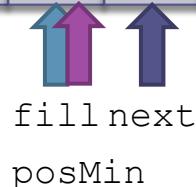
1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement

## (cont.)

50

n	5
fill	2
posMin	2
next	3

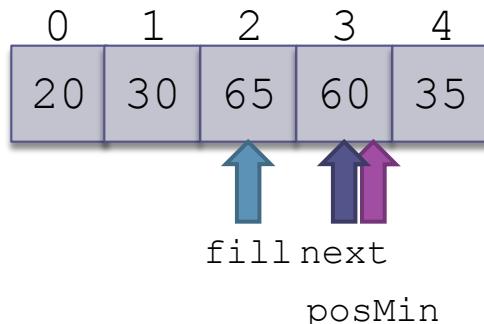


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4. **if** the item at next is less than the item at posMin
5. Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

51

n	5
fill	2
posMin	3
next	3



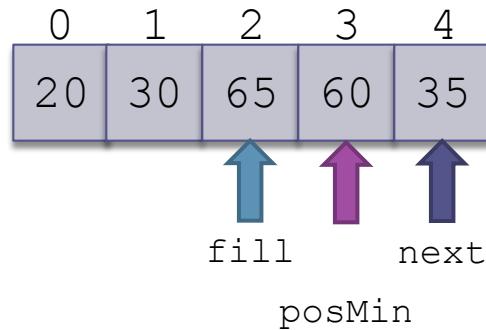
1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6.     Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement

## (cont.)

52

n	5
fill	2
posMin	3
next	4

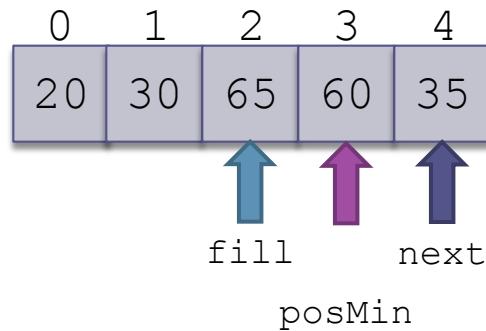


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

53

n	5
fill	2
posMin	3
next	4

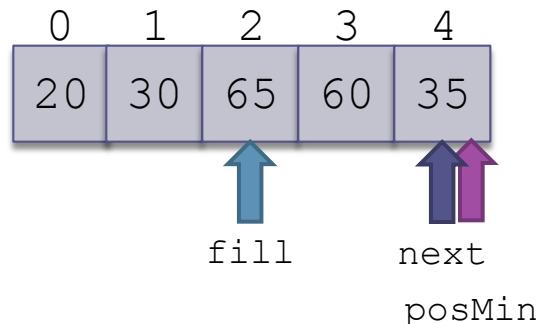


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

54

n	5
fill	2
posMin	4
next	4

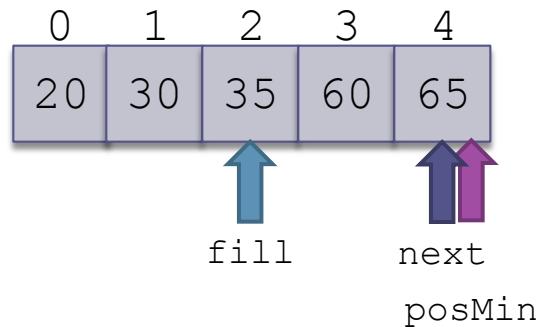


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

55

n	5
fill	2
posMin	4
next	4

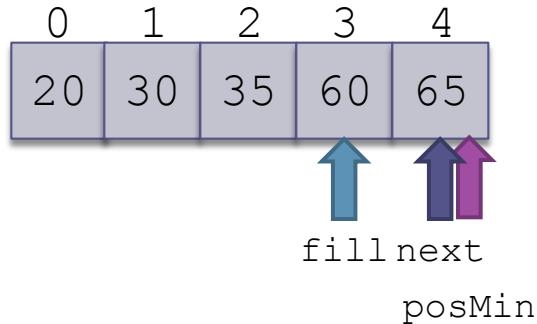


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

56

n	5
fill	3
posMin	4
next	4

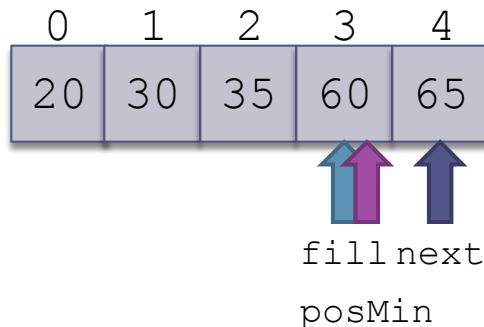


- 1. **for** fill = 0 to n - 2 do
- 2. Initialize posMin to fill
- 3. **for** next = fill + 1 to n - 1 do
- 4.     **if** the item at next is less than the item at posMin
- 5.         Reset posMin to next
- 6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

57

n	5
fill	3
posMin	3
next	4

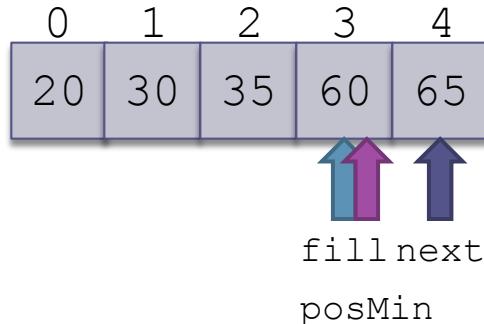


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

58

n	5
fill	3
posMin	3
next	4

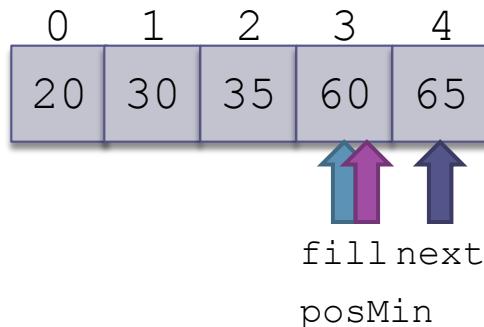


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

59

n	5
fill	3
posMin	3
next	4

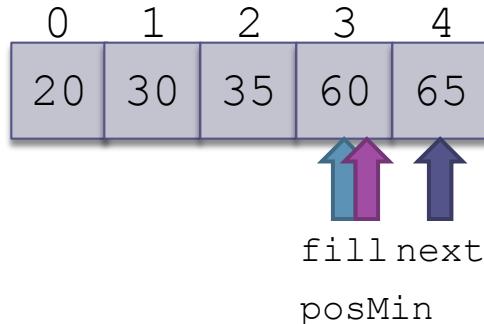


1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

60

n	5
fill	3
posMin	3
next	4



1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Trace of Selection Sort Refinement (cont.)

61

n	5
fill	3
posMin	3
next	4

0	1	2	3	4
20	30	35	60	65

1. **for** fill = 0 to n - 2 do
2. Initialize posMin to fill
3. **for** next = fill + 1 to n - 1 do
4.     **if** the item at next is less than the item at posMin
5.         Reset posMin to next
6. Exchange the item at posMin with the one at fill

# Analysis of Selection Sort

62

This loop is  
performed  $n-1$   
times

1. **for** fill = 0 **to** n - 2 **do**
2.     Initialize posMin to fill
3.     **for** next = fill + 1 **to** n - 1 **do**
4.         **if** the item at next is less than the item at posMin
5.             Reset posMin to next
6.     Exchange the item at posMin with the one at fill

# Analysis of Selection Sort (cont.)

63

There are  $n-1$  exchanges

1. **for** fill = 0 **to** n - 2 **do**
2.     Initialize posMin to fill
3.     **for** next = fill + 1 **to** n - 1 **do**
4.         **if** the item at next is less than the item at posMin
5.             Reset posMin to next
6.             Exchange the item at posMin with the one at fill

# Analysis of Selection Sort (cont.)

64

This comparison is performed  
 $(n - 1 - fill)$   
times for each value of *fill* and  
can be represented by the  
following series:  
 $(n-1) + (n-2) + \dots + 3 + 2 + 1$

1. **for** *fill* = 0 **to** *n* - 2 **do**
2.     Initialize *posMin* to *fill*
3.     **for** *next* = *fill* + 1 **to** *n* - 1 **do**
4.         **if** the item at *next* is less than the  
item at *posMin*
5.             Reset *posMin* to *next*
6.     Exchange the item at *posMin* with the one  
at *fill*

# Analysis of Selection Sort (cont.)

65

For very large  $n$  we can ignore all but the most significant term in the expression, so the number of

- comparisons is  $O(n^2)$
- exchanges is  $O(n)$

An  $O(n^2)$  sort is called a *quadratic sort*

1. **for** fill = 0 **to** n - 2 **do**
2.     Initialize posMin to fill
3.     **for** next = fill + 1 **to** n - 1 **do**
4.         **if** the item at next is less than the item at posMin
5.             Reset posMin to next
6.     Exchange the item at posMin with the one at fill

# Code for Selection Sort

66

```
public class SelectionSort {  
    /** Sort the array using selection sort algorithm.  
     * pre: table contains Comparable objects.  
     * post: table is sorted.  
     * @param table The array to be sorted  
     */  
  
    public static void sort(Comparable[] table) {  
        int n = table.length;  
        for (int fill = 0; fill < n - 1; fill++) {  
            // Invariant: table[0 . . . fill - 1] is sorted.  
            int posMin = fill;
```

# Code for Selection Sort

67

```
for (int next = fill + 1; next < n; next++) {  
    // Invariant: table[posMin] is the smallest  
    // item in  
    // table[fill . . . next - 1].  
    if (table[next].compareTo(table[posMin]) < 0)  
    {  
        posMin = next;  
    }  
    // assert: table[posMin] is the smallest item  
    // in table[fill . . . n - 1].  
    // Exchange table[fill] and table[posMin].
```

# Code for Selection Sort

68

```
Comparable temp = table[fill];
table[fill] = table[posMin];
table[posMin] = temp;
// assert: table[fill] is the smallest item in
// table[fill . . . n - 1].
}
// assert: table[0 . . . n - 1] is sorted.
}
}
```

# Insertion Sort

## Section 8.3

# Insertion Sort

70

- Another quadratic sort, *insertion sort*, is based on the technique used by card players to arrange a hand of cards
  - ▣ The player keeps the cards that have been picked up so far in sorted order
  - ▣ When the player picks up a new card, the player makes room for the new card and then inserts it in its proper place



# Trace of Insertion Sort

71

[0]	30
[1]	25
[2]	15
[3]	20
[4]	28

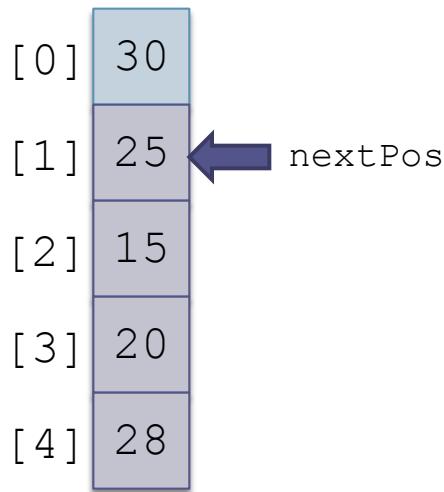
1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

To adapt the insertion algorithm to an array that is filled with data, we start with a sorted subarray consisting of only the first element

# Trace of Insertion Sort (cont.)

72

nextPos	1
---------	---

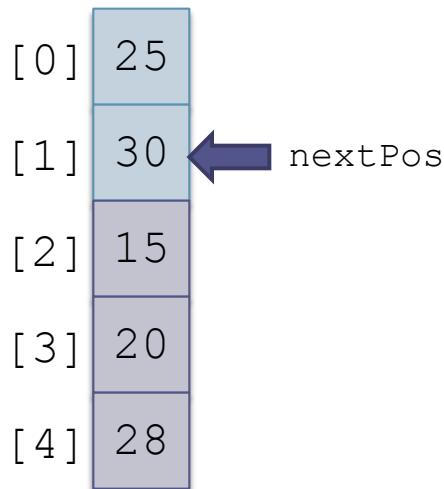


1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort (cont.)

73

nextPos	1
---------	---



1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort (cont.)

74

nextPos

2

[ 0 ]	25
[ 1 ]	30
[ 2 ]	15
[ 3 ]	20
[ 4 ]	28

nextPos

1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort (cont.)

75

nextPos

2

[ 0 ]	15
[ 1 ]	25
[ 2 ]	30
[ 3 ]	20
[ 4 ]	28

nextPos

1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort (cont.)

76

nextPos

3

[ 0 ]	15
[ 1 ]	25
[ 2 ]	30
[ 3 ]	20
[ 4 ]	28

1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort (cont.)

77

nextPos

3

[ 0 ]	15
[ 1 ]	20
[ 2 ]	25
[ 3 ]	30
[ 4 ]	28

nextPos

1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort (cont.)

78

nextPos

4

[ 0 ]	15
[ 1 ]	20
[ 2 ]	25
[ 3 ]	30
[ 4 ]	28

nextPos

1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort (cont.)

79

nextPos

4

[ 0 ]	15
[ 1 ]	20
[ 2 ]	25
[ 3 ]	28
[ 4 ]	30

nextPos

1. **for** each array element from the second (**nextPos** = 1) to the last
2. Insert the element at **nextPos** where it belongs in the array, increasing the length of the sorted subarray by 1 element

# Trace of Insertion Sort Refinement

80

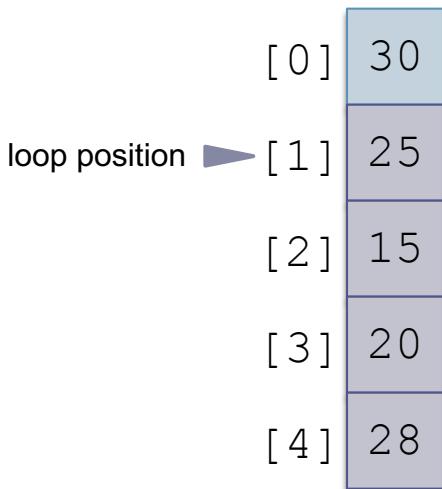
[0]	30
[1]	25
[2]	15
[3]	20
[4]	28

1. **for** each array element from the second (`nextPos = 1`) to the last
2. `nextPos` is the position of the element to insert
3. Save the value of the element to insert in `nextVal`
4. **while** `nextPos > 0` and the element at `nextPos - 1 > nextVal`
5. Shift the element at `nextPos - 1` to position `nextPos`
6. Decrement `nextPos` by 1
7. Insert `nextVal` at `nextPos`

# Trace of Insertion Sort Refinement (cont.)

81

nextPos	1
nextVal	



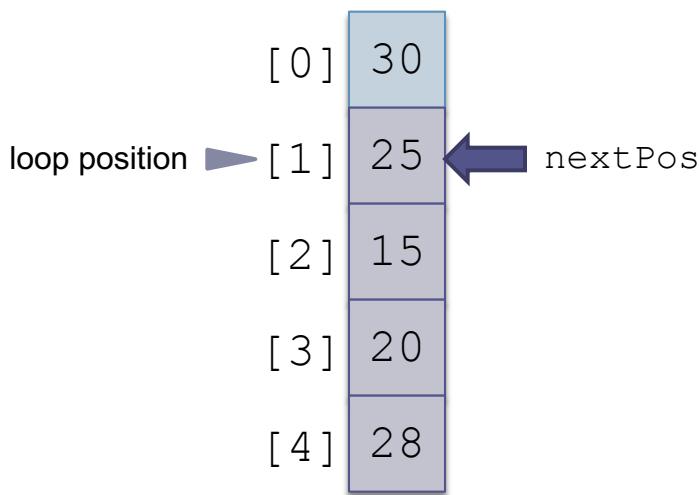
- ▶ 1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
- 2.  $\text{nextPos}$  is the position of the element to insert
- 3. Save the value of the element to insert in  $\text{nextVal}$
- 4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
- 5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
- 6. Decrement  $\text{nextPos}$  by 1
- 7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement

## (cont.)

82

nextPos	1
nextVal	



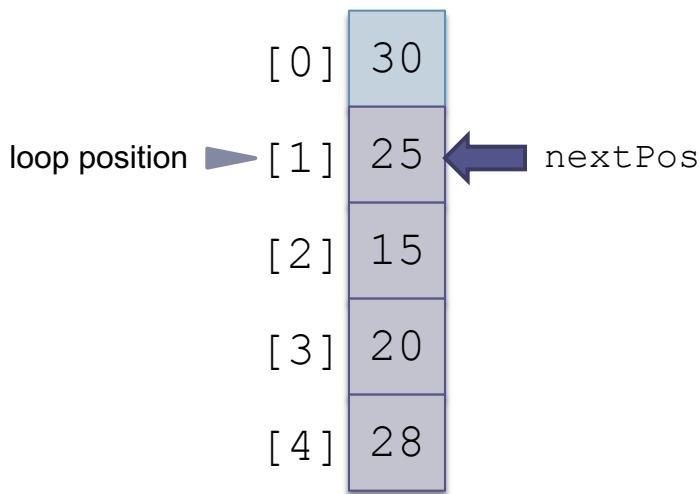
1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement

## (cont.)

83

nextPos	1
nextVal	25

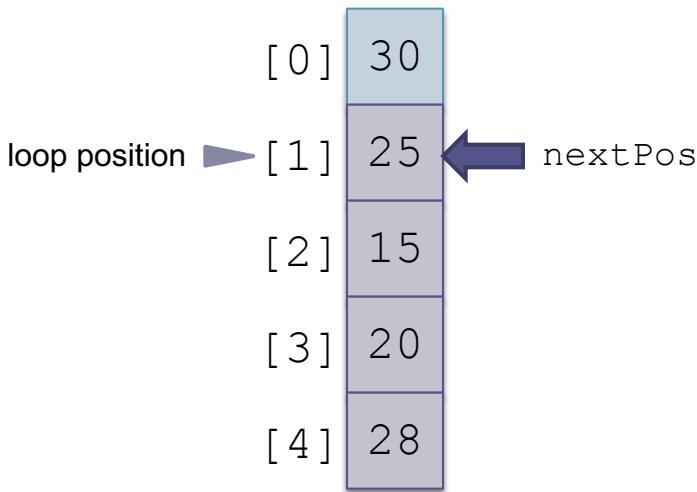


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

84

nextPos	1
nextVal	25

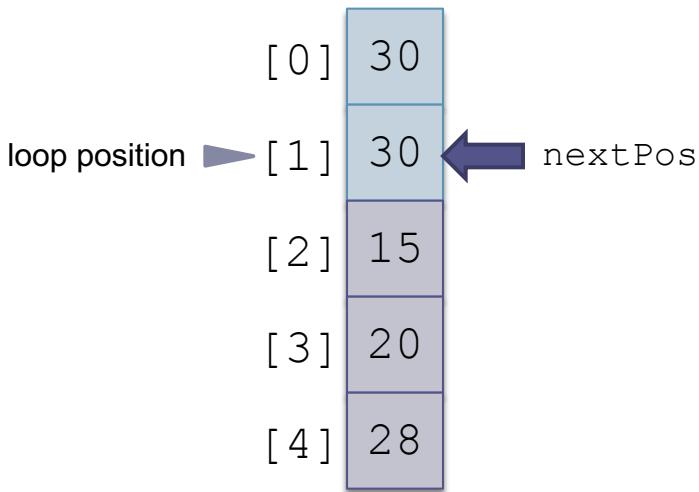


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

85

nextPos	1
nextVal	25

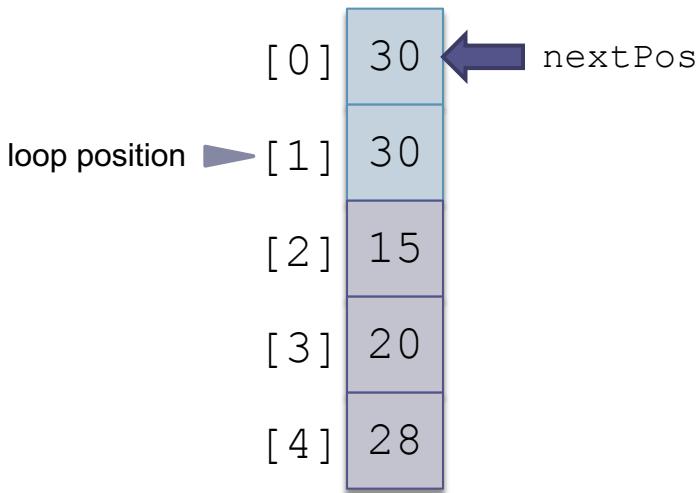


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

86

nextPos	0
nextVal	25

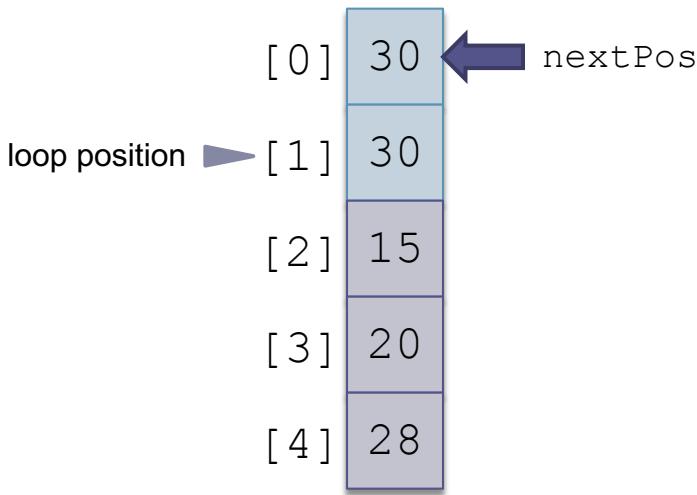


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

87

nextPos	0
nextVal	25



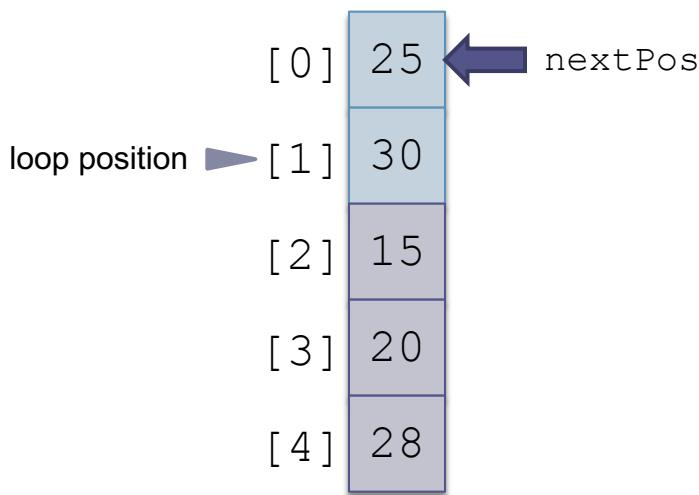
1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement

## (cont.)

88

nextPos	0
nextVal	25

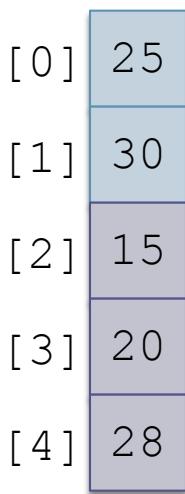


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. **Insert**  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

89

nextPos	0
nextVal	25

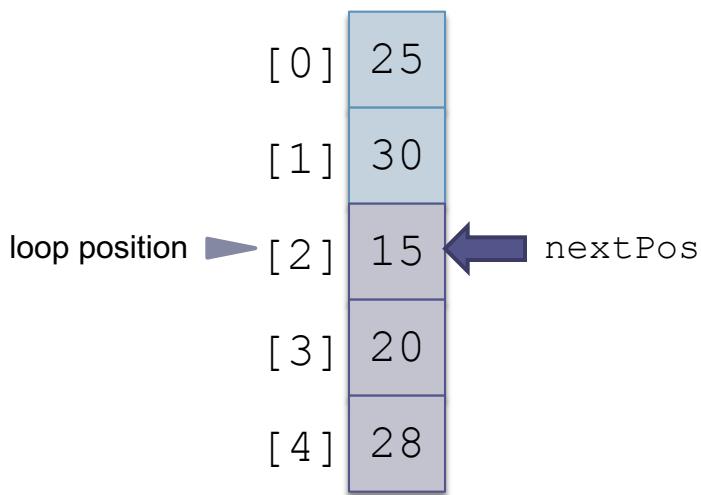


- 1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
- 2.  $\text{nextPos}$  is the position of the element to insert
- 3. Save the value of the element to insert in  $\text{nextVal}$
- 4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
- 5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
- 6. Decrement  $\text{nextPos}$  by 1
- 7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

90

nextPos	2
nextVal	25

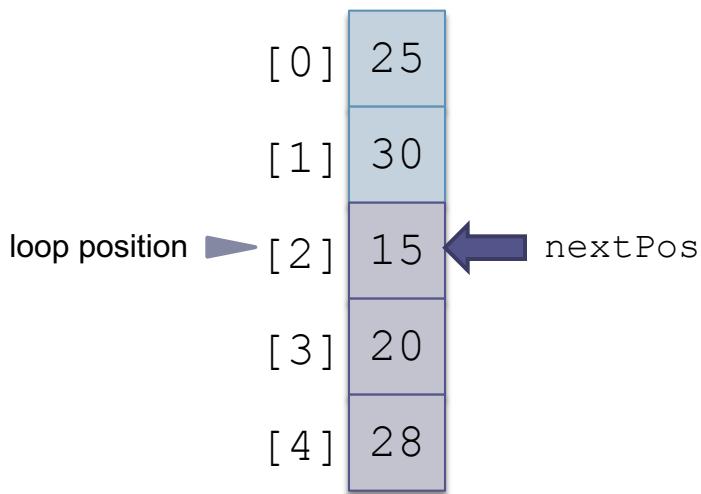


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

91

nextPos	2
nextVal	15

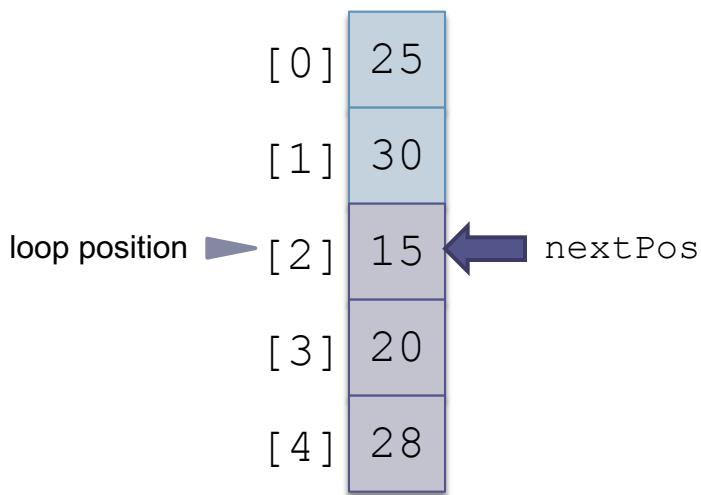


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

92

nextPos	2
nextVal	15

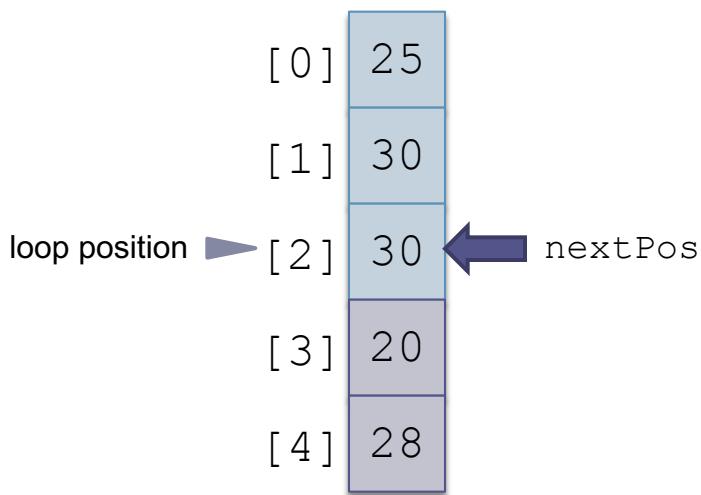


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

93

nextPos	2
nextVal	15

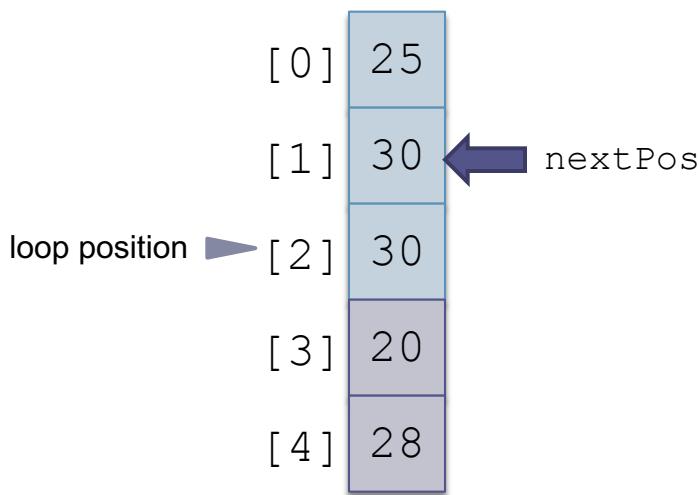


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

94

nextPos	1
nextVal	15

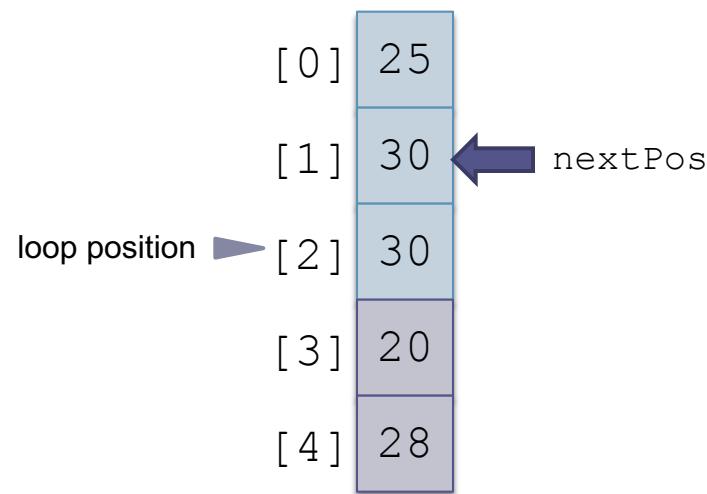


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

95

nextPos	1
nextVal	15

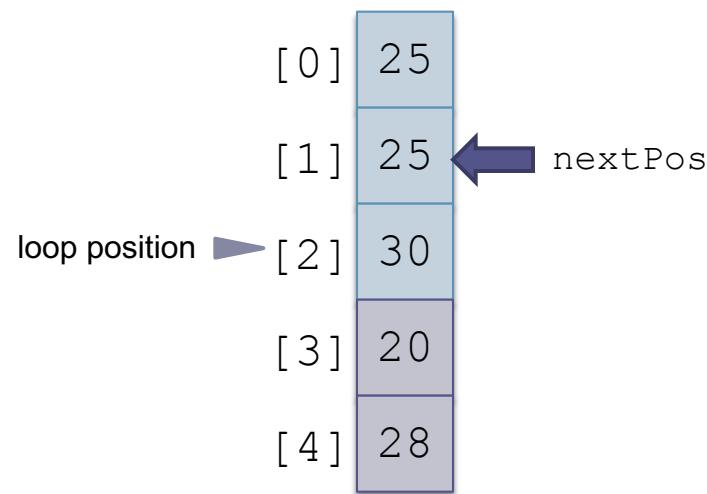


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

96

nextPos	1
nextVal	15

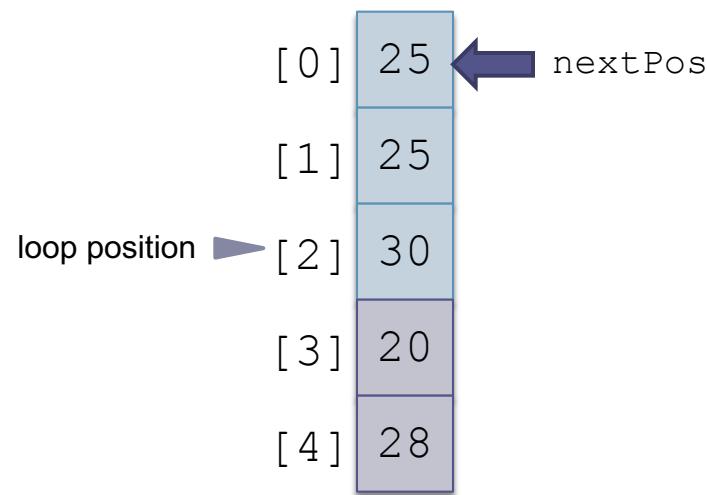


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

97

nextPos	0
nextVal	15

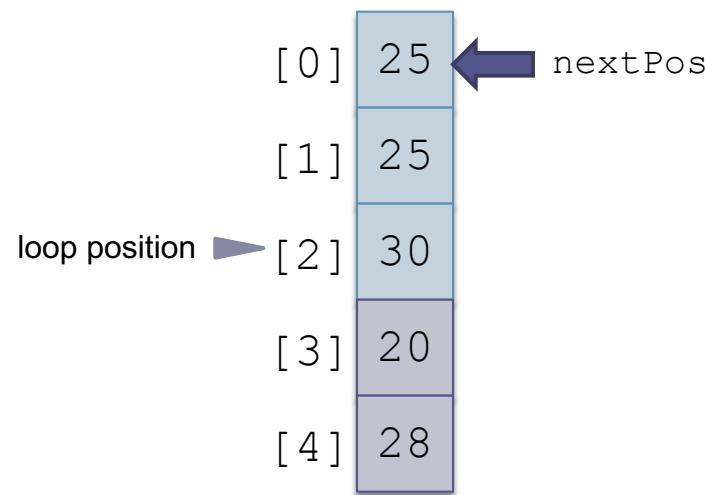


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

98

nextPos	0
nextVal	15

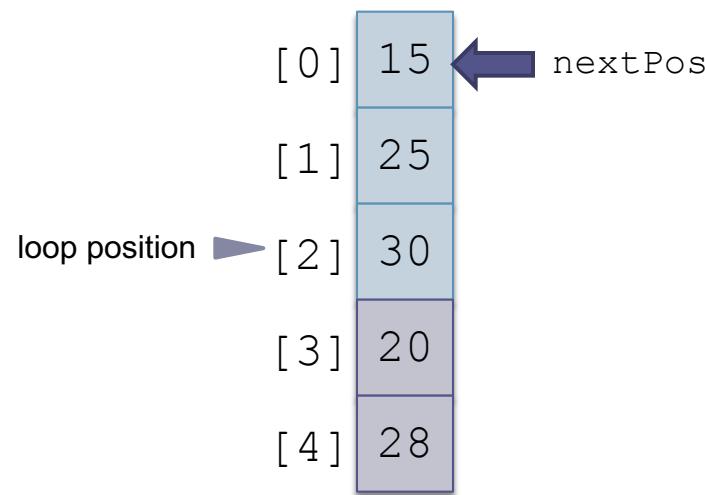


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

99

nextPos	0
nextVal	15

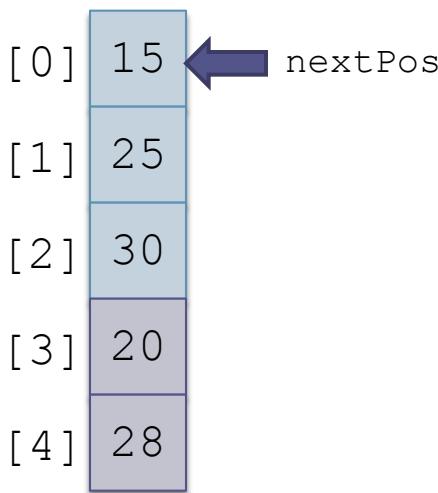


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. **Insert**  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

100

nextPos	0
nextVal	15

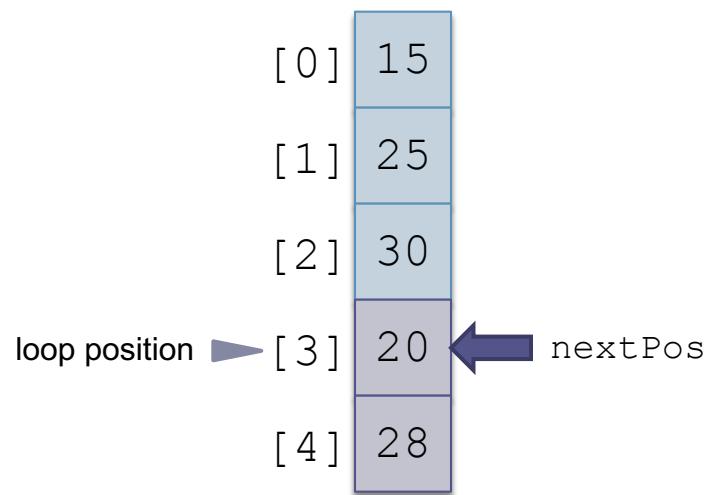


- 1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
- 2.  $\text{nextPos}$  is the position of the element to insert
- 3. Save the value of the element to insert in  $\text{nextVal}$
- 4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
- 5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
- 6. Decrement  $\text{nextPos}$  by 1
- 7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

101

nextPos	3
nextVal	15

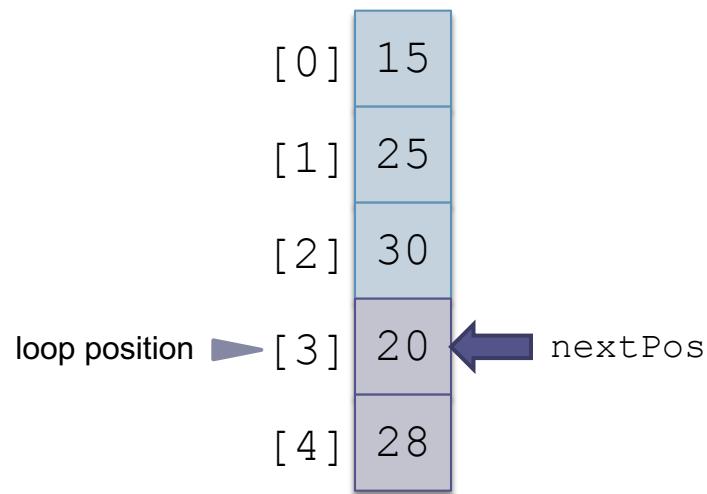


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

102

nextPos	3
nextVal	20

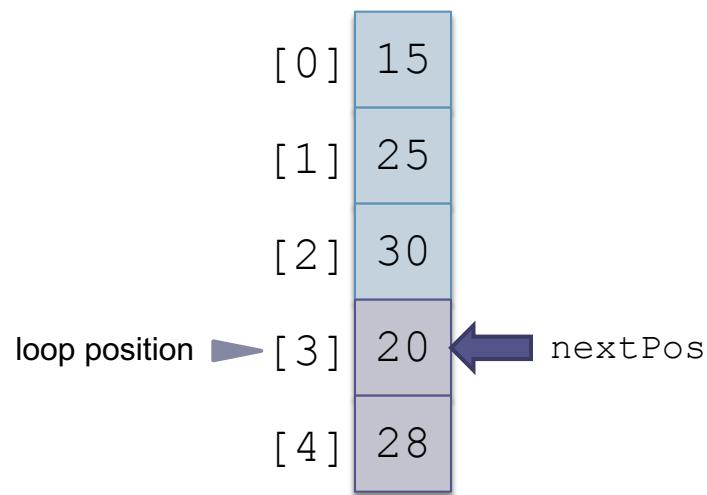


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

103

nextPos	3
nextVal	20

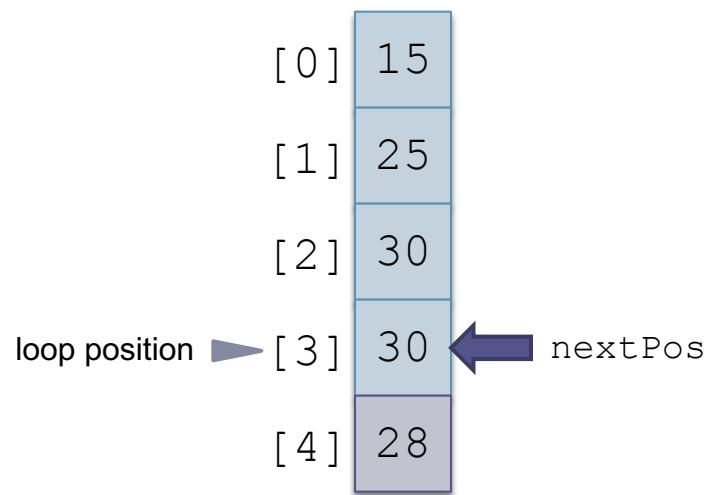


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

104

nextPos	3
nextVal	20

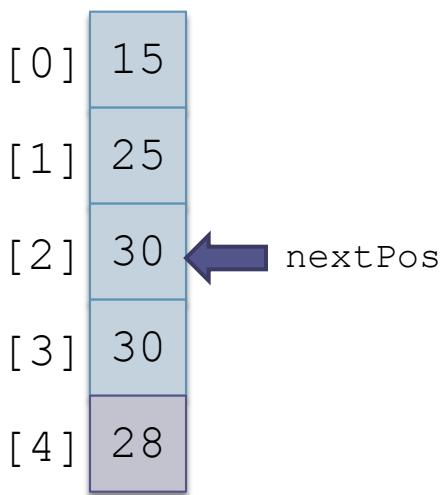


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

105

nextPos	2
nextVal	20

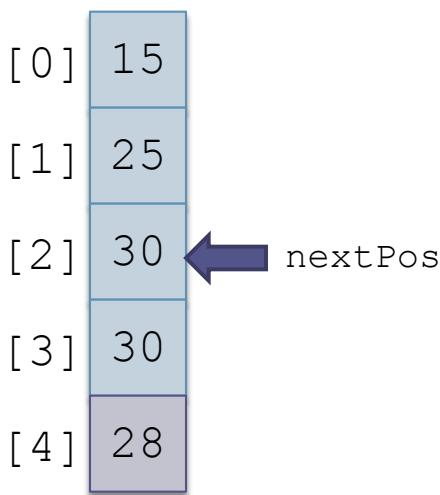


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

106

nextPos	2
nextVal	20

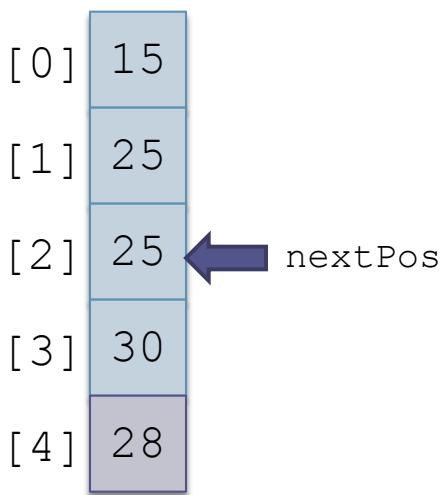


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

107

nextPos	2
nextVal	20

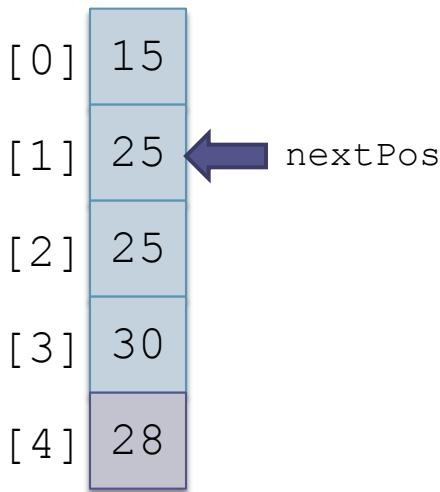


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

108

nextPos	1
nextVal	20

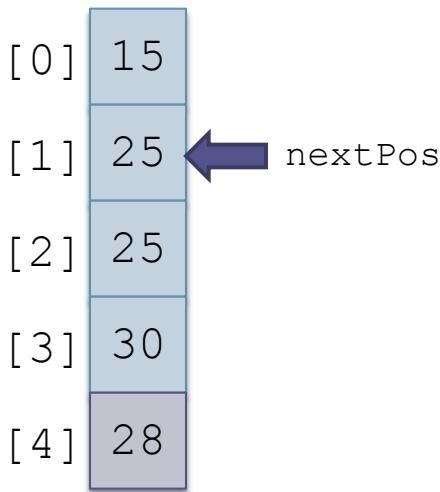


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

109

nextPos	1
nextVal	20

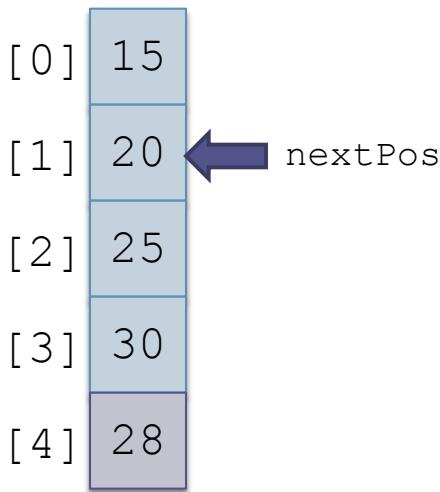


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

110

nextPos	1
nextVal	20

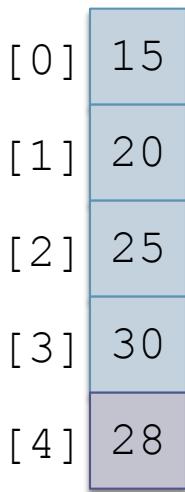


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

111

nextPos	1
nextVal	20

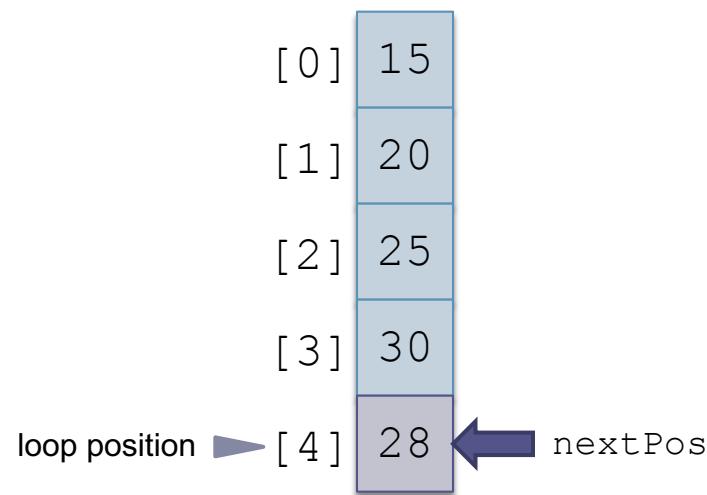


- ▶ 1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
- 2.  $\text{nextPos}$  is the position of the element to insert
- 3. Save the value of the element to insert in  $\text{nextVal}$
- 4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
- 5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
- 6. Decrement  $\text{nextPos}$  by 1
- 7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

112

nextPos	4
nextVal	20

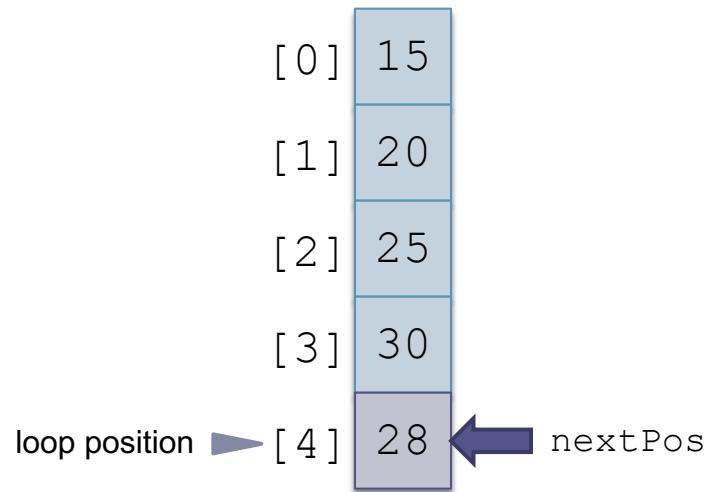


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

113

nextPos	4
nextVal	28

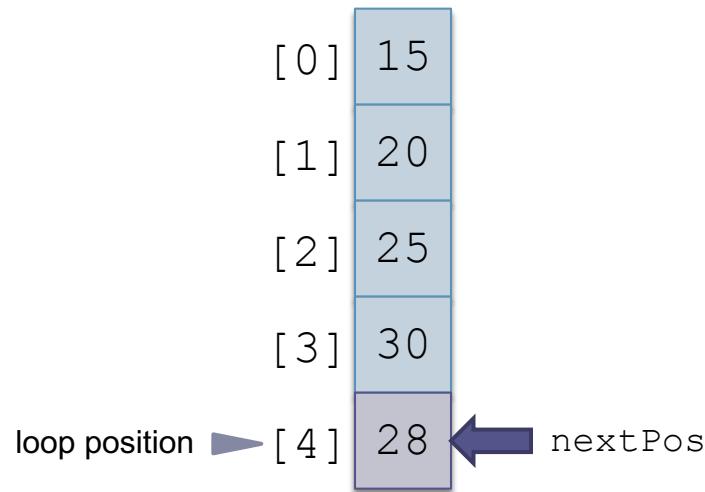


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

114

nextPos	4
nextVal	28

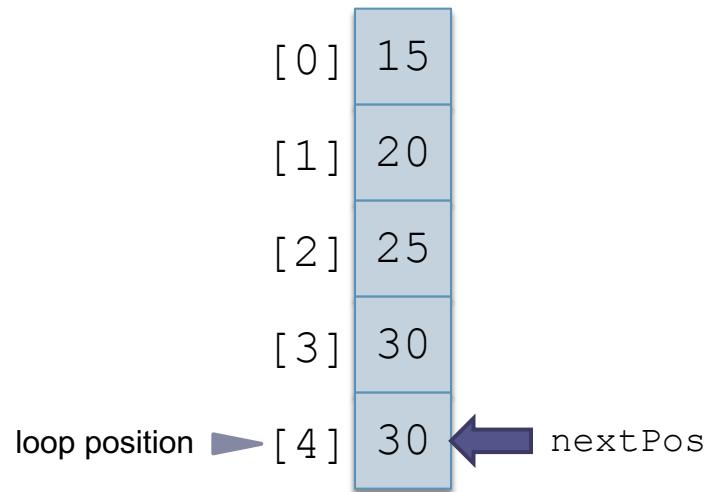


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

115

nextPos	4
nextVal	28

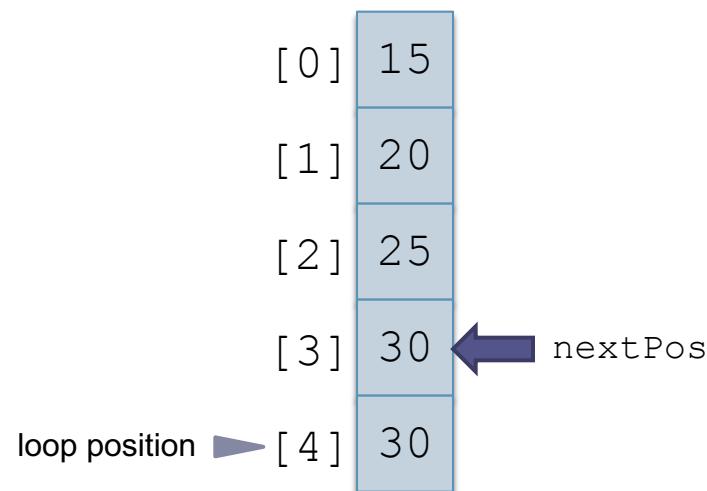


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

116

nextPos	3
nextVal	28

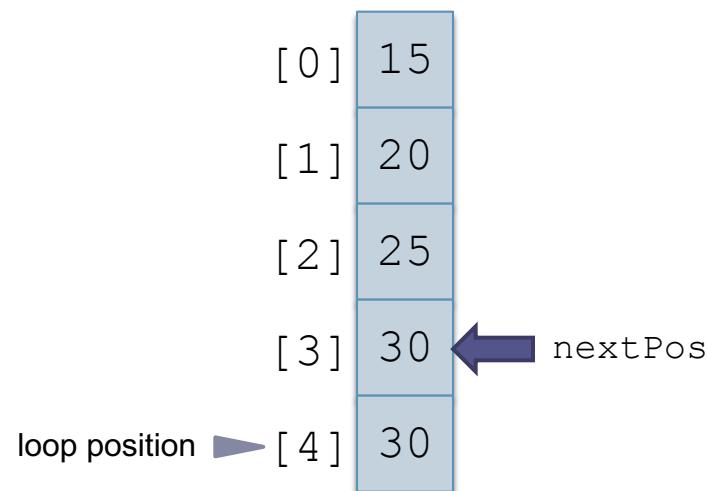


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

117

nextPos	3
nextVal	28

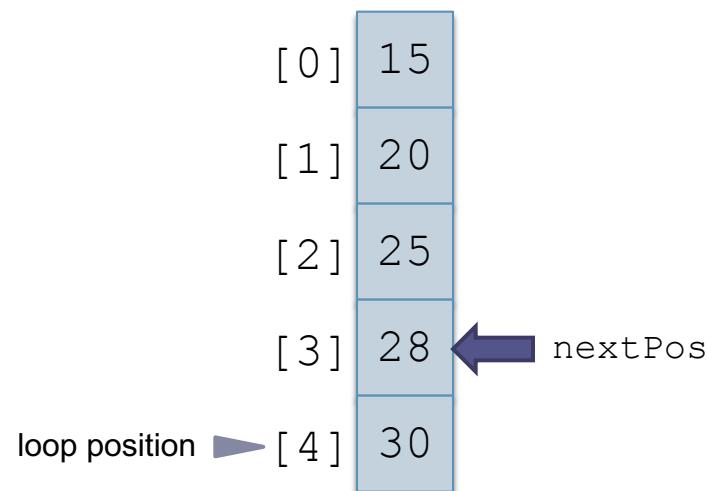


1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. Insert  $\text{nextVal}$  at  $\text{nextPos}$

# Trace of Insertion Sort Refinement (cont.)

118

nextPos	3
nextVal	28



1. **for** each array element from the second ( $\text{nextPos} = 1$ ) to the last
2.  $\text{nextPos}$  is the position of the element to insert
3. Save the value of the element to insert in  $\text{nextVal}$
4. **while**  $\text{nextPos} > 0$  and the element at  $\text{nextPos} - 1 > \text{nextVal}$
5. Shift the element at  $\text{nextPos} - 1$  to position  $\text{nextPos}$
6. Decrement  $\text{nextPos}$  by 1
7. **Insert**  $\text{nextVal}$  at  $\text{nextPos}$

# Analysis of Insertion Sort

119

- The insertion step is performed  $n - 1$  times
- In the worst case, all elements in the sorted subarray are compared to `nextVal` for each insertion
- The maximum number of comparisons will then be:
$$1 + 2 + 3 + \dots + (n - 2) + (n - 1)$$
- which is  $\mathcal{O}(n^2)$

# Analysis of Insertion Sort (cont.)

120

- In the best case (when the array is sorted already), only one comparison is required for each insertion
- In the best case, the number of comparisons is  $O(n)$
- The number of shifts performed during an insertion is one less than the number of comparisons
- Or, when the new value is the smallest so far, it is the same as the number of comparisons

# Analysis of Insertion Sort (cont.)

121

- A shift in an insertion sort requires movement of only 1 item, while an exchange in selection sort involves a temporary item and the movement of three items
  - ▣ The item moved may be a primitive or an object reference
  - ▣ The objects themselves do not change their locations

# Code for Insertion Sort

122

```
public class InsertionSort {  
    /** Sort the table using insertion sort algorithm.  
     * pre: table contains Comparable objects.  
     * post: table is sorted.  
     * @param table The array to be sorted  
    */  
  
    public static < T  
        extends Comparable < T >> void sort(T[] table) {  
        for (int nextPos = 1; nextPos < table.length; nextPos++)  
        {  
            // Invariant: table[0 . . . nextPos - 1] is sorted.  
            // Insert element at position nextPos  
            // in the sorted subarray.  
            insert(table, nextPos);  
        }  
    }  
}
```

# Code for Insertion Sort

123

```
        } // End for.  
    } // End sort.  
  
/** Insert the element at nextPos where it belongs  
 * in the array.  
 *  
 * pre: table[0 . . . nextPos - 1] is sorted.  
 * post: table[0 . . . nextPos] is sorted.  
 * @param table The array being sorted  
 * @param nextPos The position of the element to  
 * insert  
 */
```

# Code for Insertion Sort

124

```
private static < T extends Comparable < T >>
    void insert(T[] table, int nextPos) {
    T nextVal = table[nextPos]; // Element to insert.
    while (nextPos > 0
        && nextVal.compareTo(table[nextPos - 1]) < 0)
    {
        table[nextPos] = table[nextPos - 1]; // Shift down.
        nextPos--; // Check next smaller element.
    }
    // Insert nextVal at nextPos.
    table[nextPos] = nextVal;
}
```

# Comparison of Quadratic Sorts (cont.)

125

## Comparison of growth rates

$n$	$n^2$	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

# Comparison of Quadratic Sorts (cont.)

126

- Insertion sort
  - gives the best performance for most arrays
  - takes advantage of any partial sorting in the array and uses less costly shifts
- None of the quadratic search algorithms are particularly good for large arrays ( $n > 1000$ )
- The best sorting algorithms provide  $n \log n$  average case performance

# Comparison of Quadratic Sorts (cont.)

127

- All quadratic sorts require storage for the array being sorted
- However, the array is sorted in place
- While there are also storage requirements for variables, for large  $n$ , the size of the array dominates and extra space usage is  $O(1)$

# Comparisons versus Exchanges

128

- In Java, an exchange requires a switch of two object references using a third object reference as an intermediary
- A comparison requires an execution of a `compareTo` method
- The cost of a comparison depends on its complexity, but is generally more costly than an exchange
- For some other languages, an exchange may involve physically moving information rather than swapping object references. In these cases, an exchange may be more costly than a comparison
  - In some cases, writing to memory is much slower than reading

# Merge Sort

## Section 8.6

# Merge

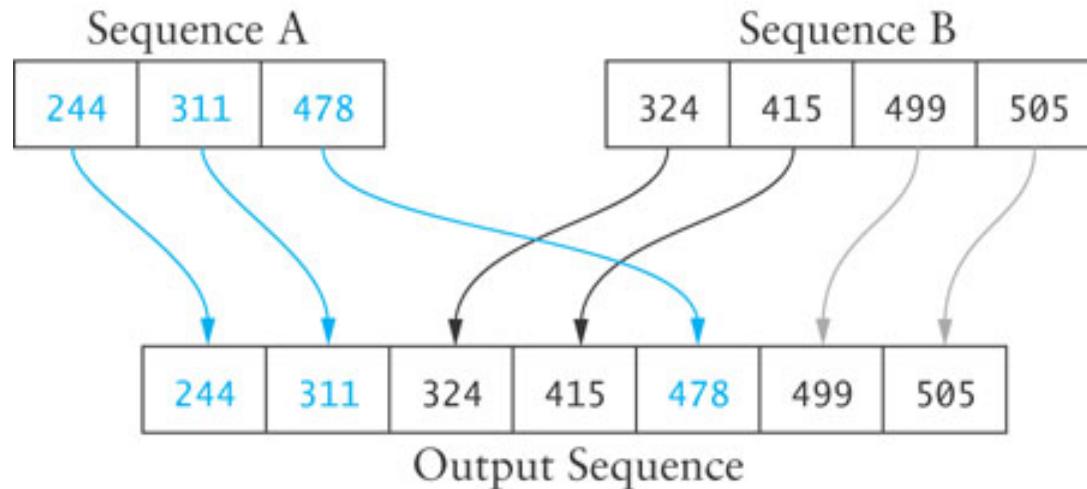
130

- A *merge* is a common data processing operation performed on two sequences of data with the following characteristics
  - ▣ Both sequences contain items with a common `compareTo` method
  - ▣ The objects in both sequences are ordered in accordance with this `compareTo` method
- The result is a third sequence containing all the data from the first two sequences

# Merge Algorithm

131

1. Access the first item from both sequences.
2. while not finished with either sequence
3. Compare the current items from the two sequences, copy the smaller current item to the output sequence, and access the next item from the input sequence whose item was copied.
4. Copy any remaining items from the first sequence to the output sequence.
5. Copy any remaining items from the second sequence to the output sequence.



# Analysis of Merge

132

- For two input sequences each containing  $n$  elements, each element needs to move from its input sequence to the output sequence
- Merge time is  $O(n)$
- Space requirements
  - ▣ The array cannot be merged in place
  - ▣ Additional space usage is  $O(n)$

# Code for Merge

133

# Code for Merge

134

```
int i = 0; // Index into the left input sequence.  
int j = 0; // Index into the right input sequence.  
int k = 0; // Index into the output sequence.  
  
// While there is data in both input sequences  
while (i < leftSequence.length && j <  
        rightSequence.length) {  
    // Find the smaller and  
    // insert it into the output sequence.  
    if (leftSequence[i].compareTo(rightSequence[j]) < 0) {  
        outputSequence[k++] = leftSequence[i++];  
    }  
}
```

# Code for Merge

135

```
else {
    outputSequence[k++] = rightSequence[j++];
}
}
// assert: one of the sequences has more items to copy.
// Copy remaining input from left sequence to the output.
while (i < leftSequence.length) {
    outputSequence[k++] = leftSequence[i++];
}
// Copy remaining input from right sequence into output.
while (j < rightSequence.length) {
    outputSequence[k++] = rightSequence[j++];
}
}
}
```

# Merge Sort

136

- We can modify merging to sort a single, unsorted array
  1. Split the array into two halves
  2. Sort the left half
  3. Sort the right half
  4. Merge the two
- This algorithm can be written with a recursive step

# (recursive) Algorithm for Merge Sort

137

## Algorithm for Merge Sort

1. if the tableSize is > 1
2.     Set halfSize to tableSize divided by 2.
3.     Allocate a table called leftTable of size halfSize.
4.     Allocate a table called rightTable of size tableSize - halfSize.
5.     Copy the elements from table[0 ... halfSize - 1] into leftTable.
6.     Copy the elements from table[halfSize ... tableSize] into rightTable.
7.     Recursively apply the merge sort algorithm to leftTable.
8.     Recursively apply the merge sort algorithm to rightTable.
9.     Apply the merge method using leftTable and rightTable as the input and the original table as the output.

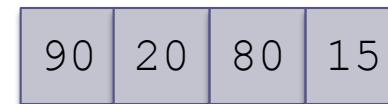
# Trace of Merge Sort

138



# Trace of Merge Sort (cont.)

139



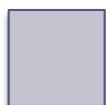
# Trace of Merge Sort (cont.)

140



# Trace of Merge Sort (cont.)

141



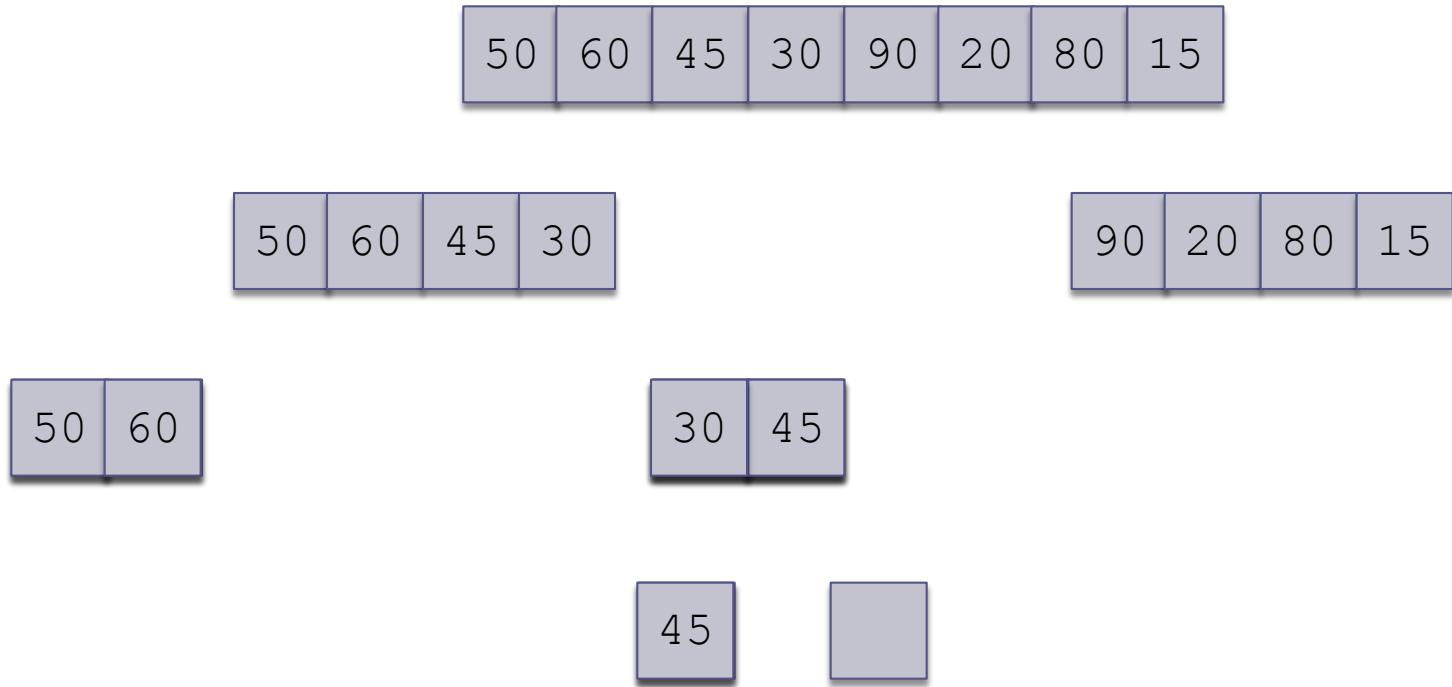
# Trace of Merge Sort (cont.)

142



# Trace of Merge Sort (cont.)

143



# Trace of Merge Sort (cont.)

144



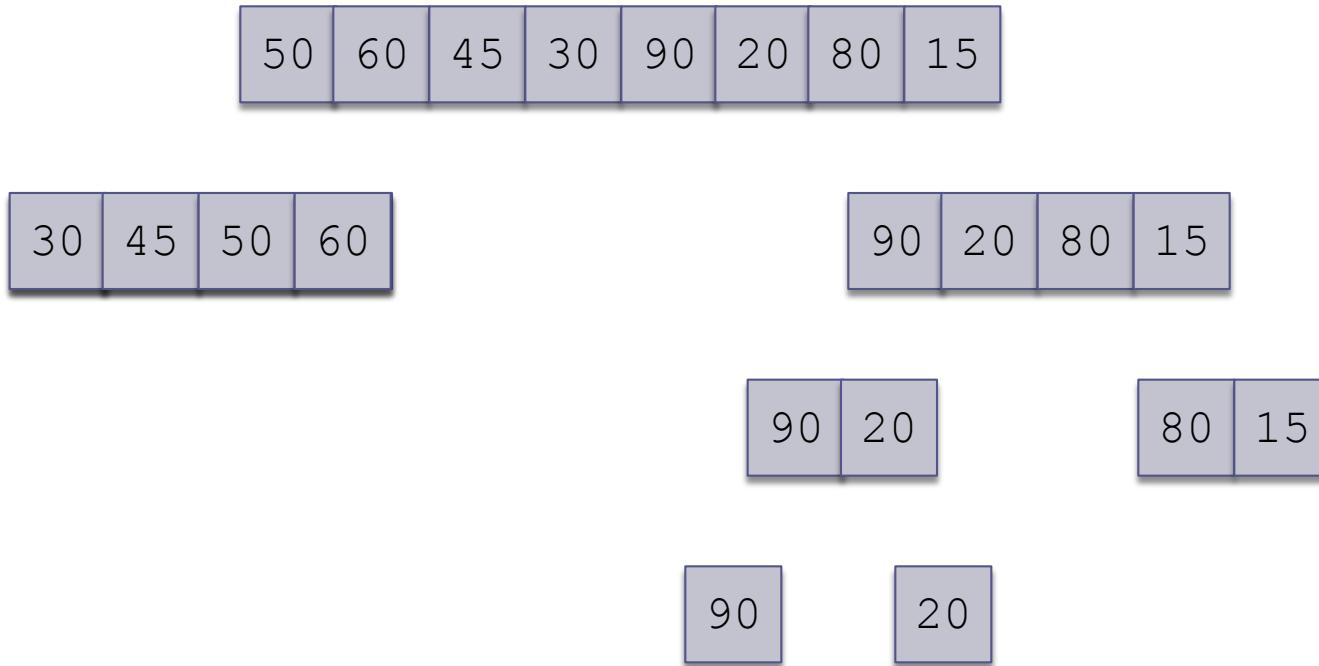
# Trace of Merge Sort (cont.)

145



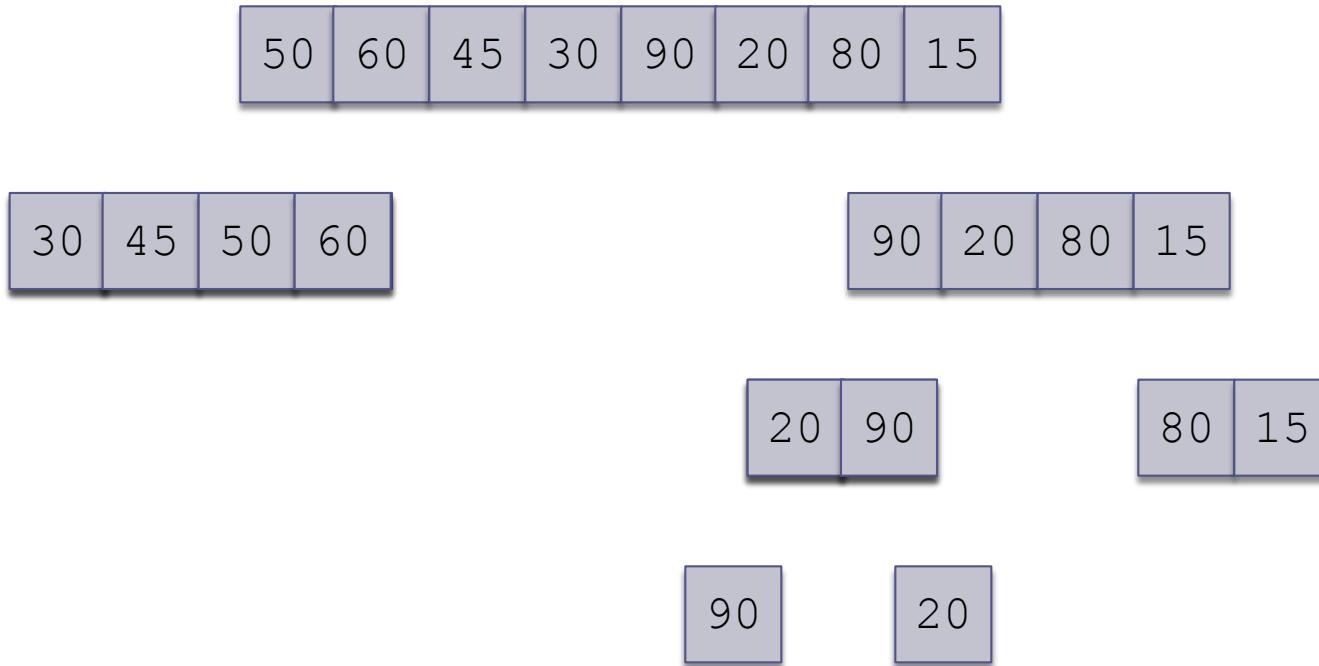
# Trace of Merge Sort (cont.)

146



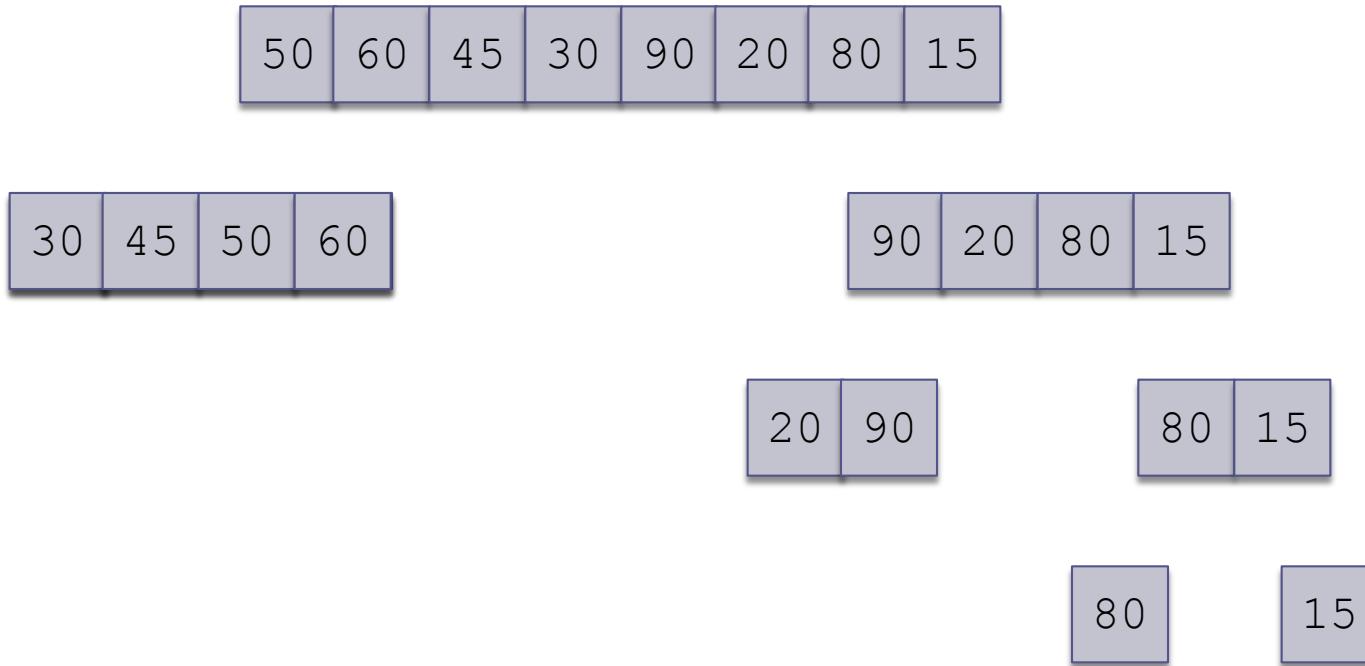
# Trace of Merge Sort (cont.)

147



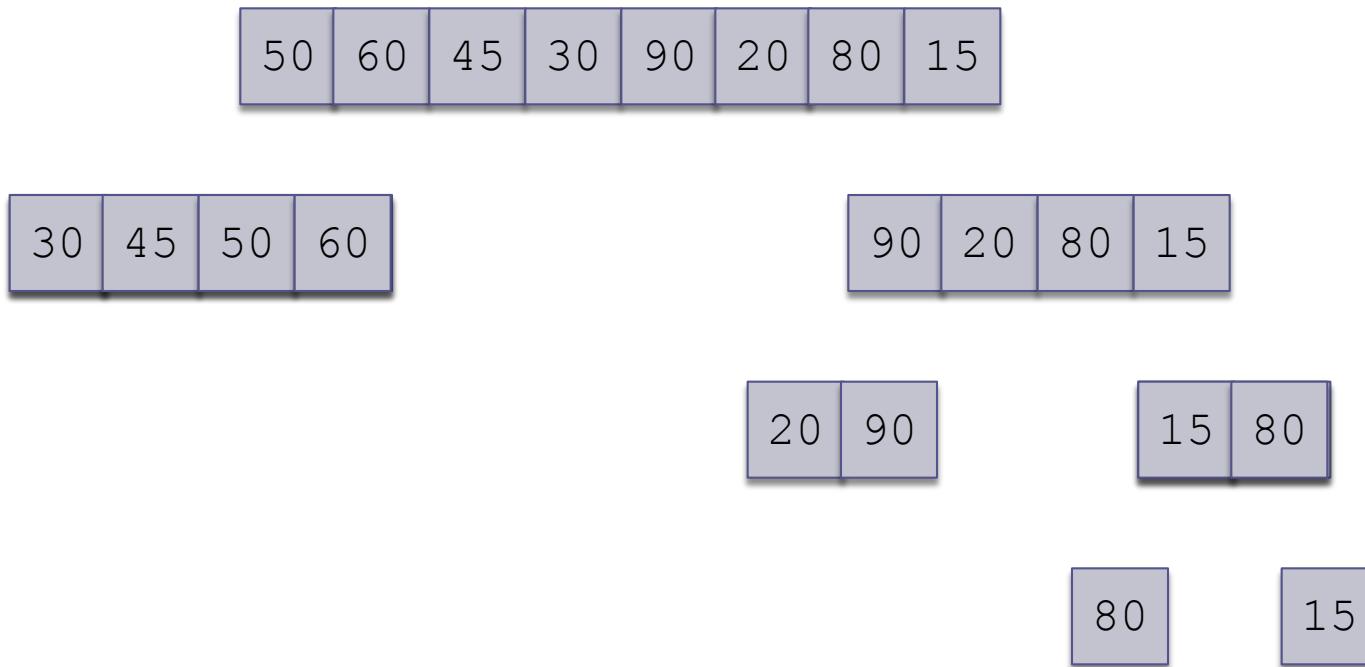
# Trace of Merge Sort (cont.)

148



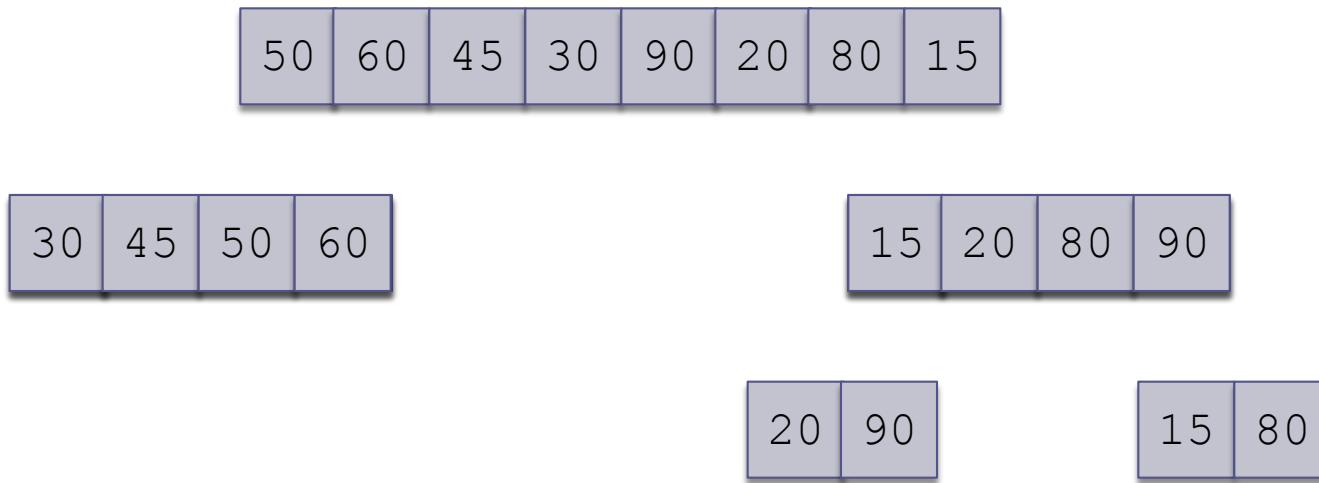
# Trace of Merge Sort (cont.)

149



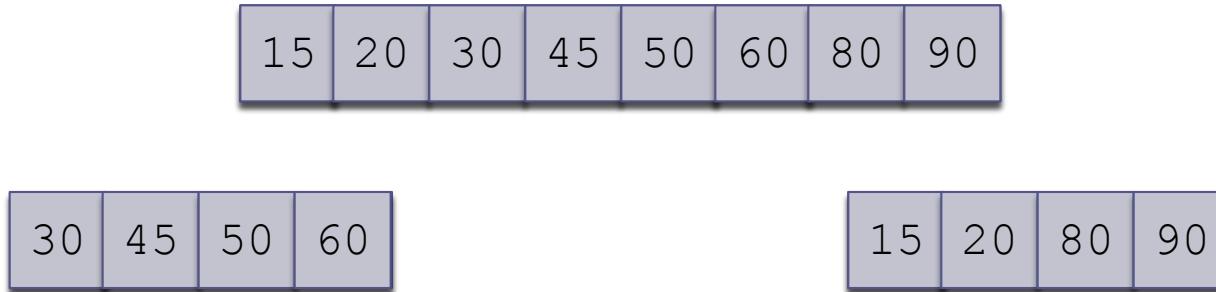
# Trace of Merge Sort (cont.)

150



# Trace of Merge Sort (cont.)

151



# Analysis of Merge Sort

152

- Each backward step requires a movement of  $n$  elements from smaller-size arrays to larger arrays; the effort is  $O(n)$
- The number of lines which require merging at each step is  $\log n$  because each recursive step splits the array in half
  - ▣ A line here refers to a subdivision of all current arrays
- The total effort to reconstruct the sorted array through merging is  $O(n \log n)$

# Analysis of Merge Sort (cont.)

153

- Going down through the recursion chain, sorting the left tables, a sequence of right tables of size

$$\frac{n}{2}, \frac{n}{4}, \dots, \frac{n}{2^k}$$

is allocated

- Since

$$\frac{n}{2} + \frac{n}{4} + \dots + 2 + 1 = n - 1$$

a total of  $n$  additional storage locations are required

# Code for Merge Sort

154

```
/** Implements the recursive merge sort algorithm. In this
version, copies of the subtables are made, sorted, and
then merged.

* @author Koffman and Wolfgang
*/
public class MergeSort {
    /** Sort the array using the merge sort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
    */
}
```

# Code for Merge Sort

155

```
public static < T  
    extends Comparable < T >> void sort(T[] table) {  
    // A table with one element is sorted already.  
    if (table.length > 1) {  
        // Split table into halves.  
        int halfSize = table.length / 2;  
        T[] leftTable = (T[]) new Comparable[halfSize];  
        T[] rightTable =  
            (T[]) new Comparable[table.length - halfSize];
```

# Code for Merge Sort

156

```
System.arraycopy(table, 0, leftTable, 0, halfSize);  
System.arraycopy(table, halfSize, rightTable, 0,  
                 table.length - halfSize);  
  
        //Sort the halves.  
        sort(leftTable);  
        sort(rightTable);  
  
        // Merge the halves.  
        merge(table, leftTable, rightTable);  
    }  
}
```

# Quicksort

## Section 8.9

# Quicksort

158

- Developed in 1962
- Quicksort selects a specific value called a pivot and rearranges the array into two parts (called *partitioning*)
  - ▣ all the elements in the left subarray are less than or equal to the pivot
  - ▣ all the elements in the right subarray are larger than the pivot
  - ▣ The pivot is placed between the two subarrays
- The process is repeated until the array is sorted

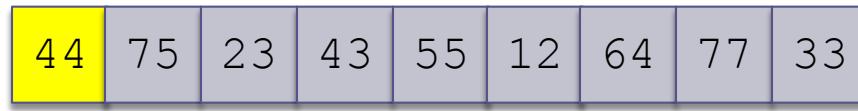
# Trace of Quicksort

159



# Trace of Quicksort (cont.)

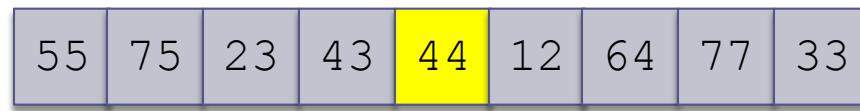
160



Arbitrarily select  
the first element as  
the pivot

# Trace of Quicksort (cont.)

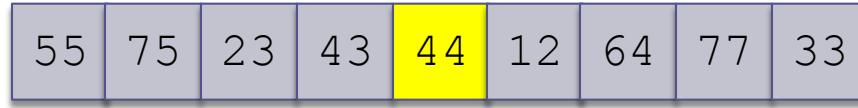
161



Swap the pivot with the  
element in the middle

# Trace of Quicksort (cont.)

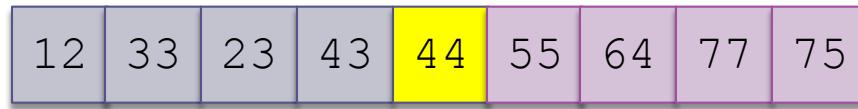
162



Partition the elements so  
that all values less than or  
equal to the pivot are to  
the left, and all values  
greater than the pivot are  
to the right

# Trace of Quicksort (cont.)

163

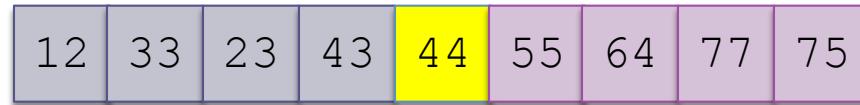


Partition the elements so  
that all values less than or  
equal to the pivot are to  
the left, and all values  
greater than the pivot are  
to the right

# Quicksort Example(cont.)

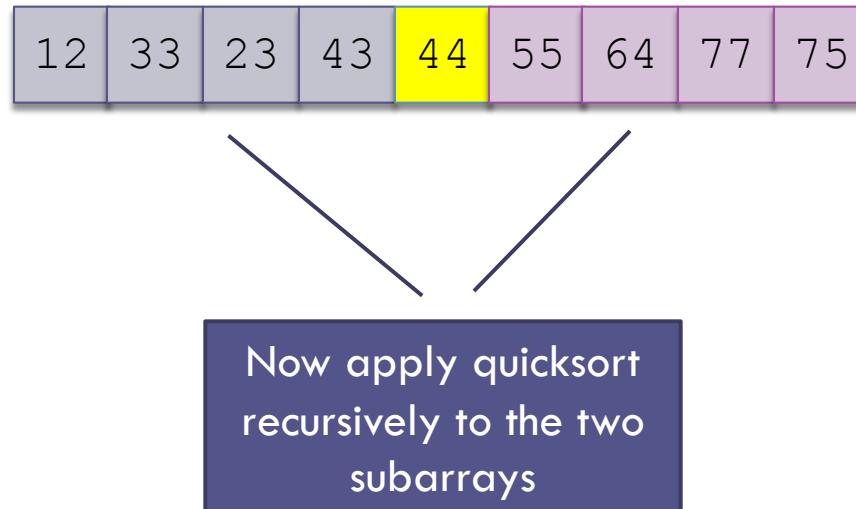
164

44 is now in its correct  
position



# Trace of Quicksort (cont.)

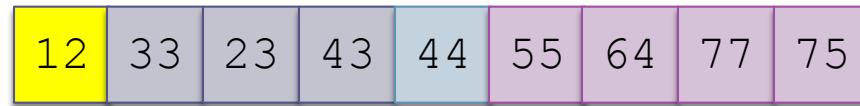
165



# Trace of Quicksort (cont.)

166

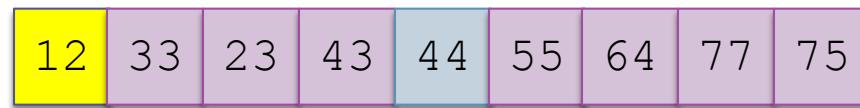
Pivot value = 12



# Trace of Quicksort (cont.)

167

Pivot value = 12



# Trace of Quicksort (cont.)

168

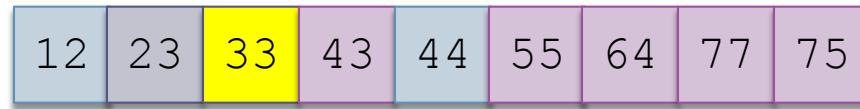
Pivot value = 33



# Trace of Quicksort (cont.)

169

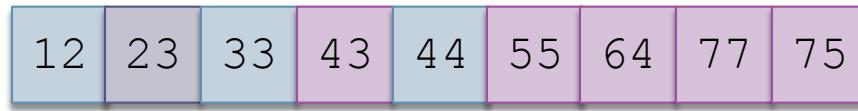
Pivot value = 33



# Trace of Quicksort (cont.)

170

Pivot value = 33



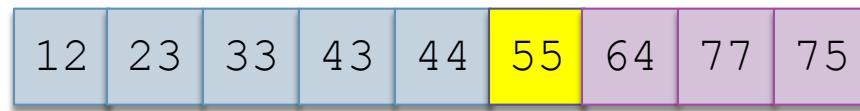
Left and right  
subarrays have single  
values; they are  
sorted

Two dark blue lines point from the text box down to the first element (12) and the last element (75) of the array, indicating that these are the single values in the left and right subarrays respectively.

# Trace of Quicksort (cont.)

171

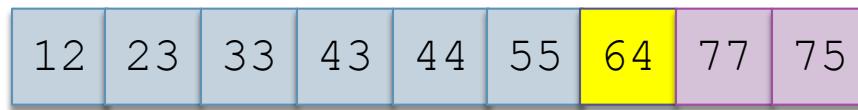
Pivot value = 55



# Trace of Quicksort (cont.)

172

Pivot value = 64



# Trace of Quicksort (cont.)

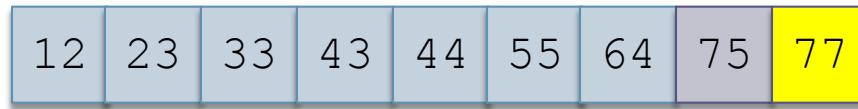
173

Pivot value = 77



# Trace of Quicksort (cont.)

Pivot value = 77



# Trace of Quicksort (cont.)

175



Left subarray has  
single value; it is  
sorted

# Algorithm for Quicksort

176

- We describe how to do the partitioning later
- The indexes `first` and `last` are the end points of the array being sorted
- The index of the pivot after partitioning is `pivIndex`

## Algorithm for Quicksort

1. `if first < last then`
2.     **Partition the elements in the subarray `first . . . last` so that the pivot value is in its correct place (subscript `pivIndex`)**
3.     **Recursively apply quicksort to the subarray `first . . . pivIndex - 1`**
4.     **Recursively apply quicksort to the subarray `pivIndex + 1 . . . last`**

# Analysis of Quicksort

177

- If the pivot value is a random value selected from the current subarray,
  - then statistically half of the items in the subarray will be less than the pivot and half will be greater
- If both subarrays have the same number of elements (best case), there will be  $\log n$  levels of recursion
- At each recursion level, the partitioning process involves moving every element to its correct position— $n$  moves
- Quicksort is  $O(n \log n)$ , just like merge sort

# Analysis of Quicksort (cont.)

178

- The array split may not be the best case, i.e. 50-50
- An exact analysis is difficult (and beyond the scope of this class), but, the running time will be bound by a constant  $\times n \log n$

# Analysis of Quicksort (cont.)

179

- A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty.
- In that case, the sort will be  $O(n^2)$
- Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts
  - We'll discuss a solution later

# Code for Quicksort

180

```
/** Implements the quicksort algorithm.  
 *  @author Koffman and Wolfgang  
 *  */  
public class QuickSort {  
    /** Sort the table using the quicksort algorithm.  
     *  pre: table contains Comparable objects.  
     *  post: table is sorted.  
     *  @param table The array to be sorted  
     */  
    public static < T  
        extends Comparable < T >> void sort(T[] table) {  
        // Sort the whole table.  
        quickSort(table, 0, table.length - 1);  
    }
```

# Code for Quicksort

181

```
/** Sort a part of the table using the quicksort
algorithm.

post: The part of table from first through last is
sorted.

@param table The array to be sorted
@param first The index of the low bound
@param last The index of the high bound

*/
private static < T
    extends Comparable < T >> void quickSort(T[] table,
                                              int first, int last) {
    if (first < last) { // There is data to be sorted.
        // Partition the table.

        int pivIndex = partition(table, first, last);
```

# Code for Quicksort

182

```
// Sort the left half.  
quickSort(table, first, pivIndex - 1);  
// Sort the right half.  
quickSort(table, pivIndex + 1, last);  
}  
}  
  
// Insert partition method  
}
```

# Algorithm for Partitioning

183



If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript first

# Trace of Partitioning (cont.)

184

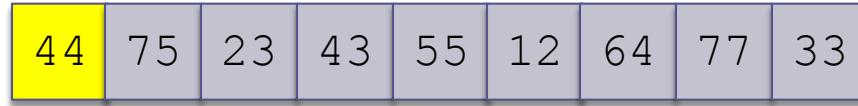


If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript first

# Trace of Partitioning (cont.)

185



For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

# Trace of Partitioning (cont.)

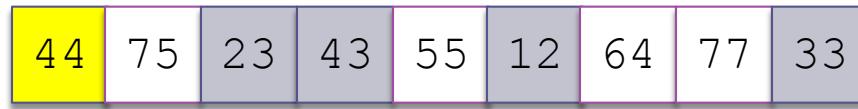
186



For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

# Trace of Partitioning (cont.)

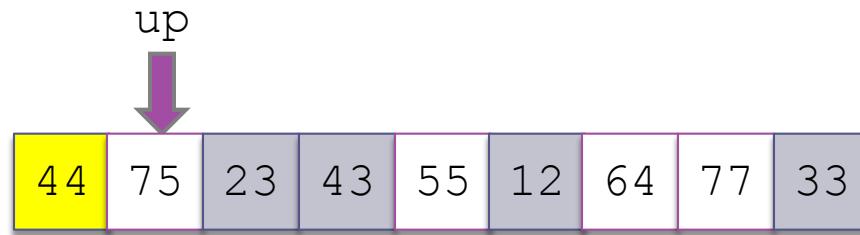
187



Search for the first value at the left end of the array that is greater than the pivot value

# Trace of Partitioning (cont.)

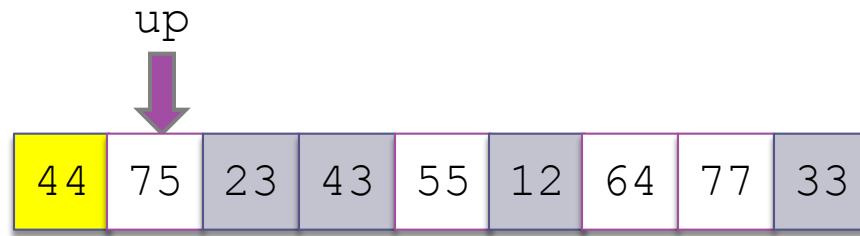
188



Search for the first value at the left end of the array that is greater than the pivot value

# Trace of Partitioning (cont.)

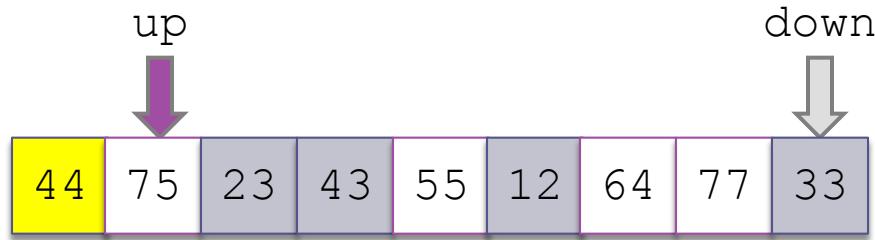
189



Then search for the first value at the right end of the array that is less than or equal to the pivot value

# Trace of Partitioning (cont.)

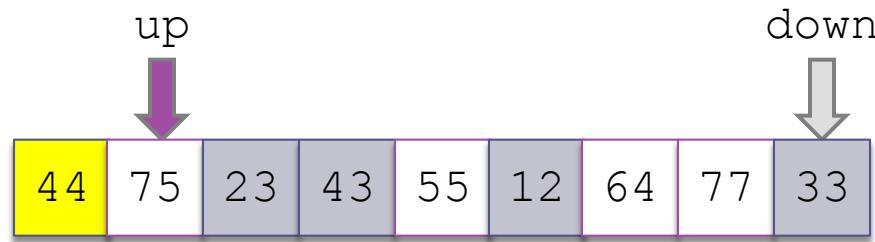
190



Then search for the first value at the right end of the array that is less than or equal to the pivot value

# Trace of Partitioning (cont.)

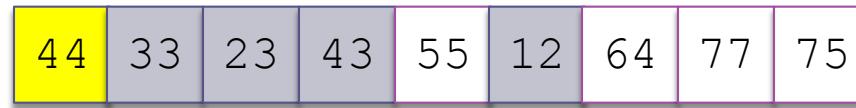
191



Exchange these values

# Trace of Partitioning (cont.)

192



Exchange these values

# Trace of Partitioning (cont.)

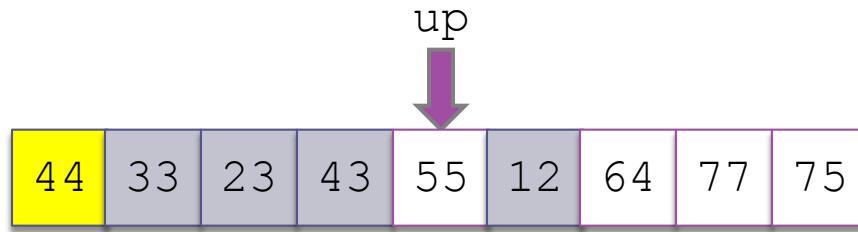
193



Repeat

# Trace of Partitioning (cont.)

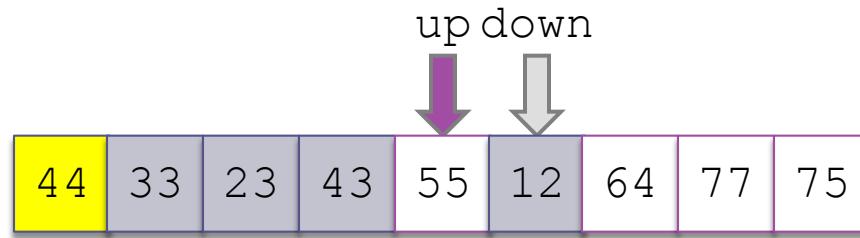
194



Find first value at left end greater  
than pivot

# Trace of Partitioning (cont.)

195



Find first value at right end less than  
or equal to pivot

# Trace of Partitioning (cont.)

196



Exchange

# Trace of Partitioning (cont.)

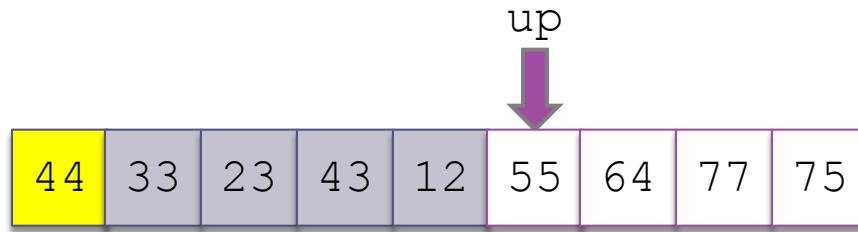
197



Repeat

# Trace of Partitioning (cont.)

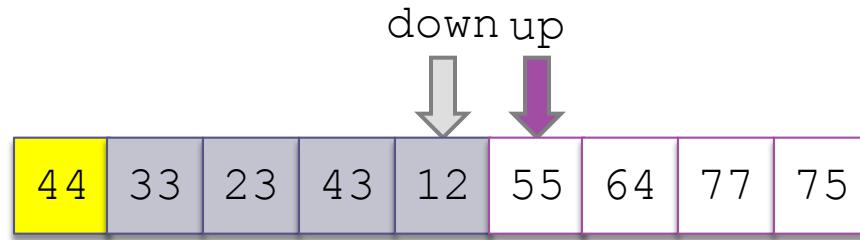
198



Find first element at left end  
greater than pivot

# Trace of Partitioning (cont.)

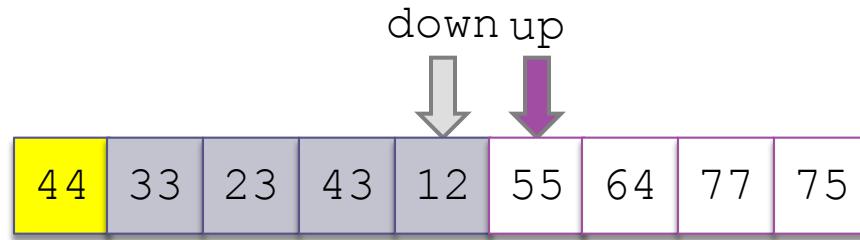
199



Find first element at right end less  
than or equal to pivot

# Trace of Partitioning (cont.)

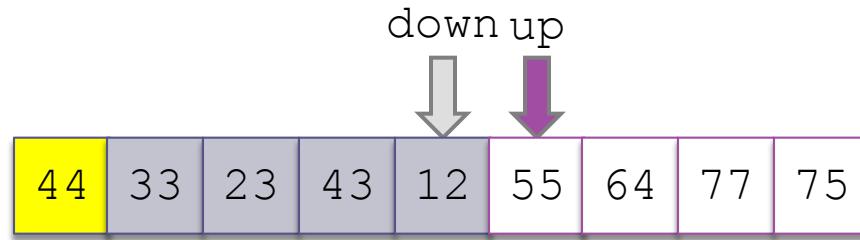
200



Since down has "passed" up, do  
not exchange

# Trace of Partitioning (cont.)

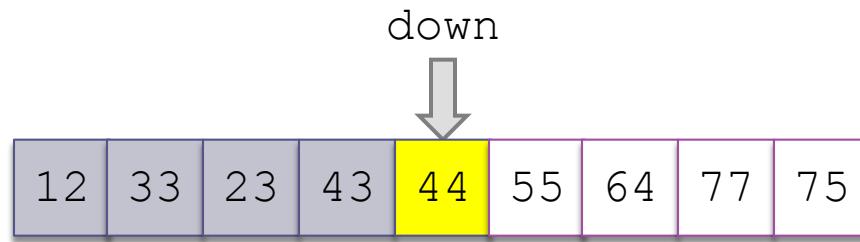
201



Exchange the pivot value with the  
value at down

# Trace of Partitioning (cont.)

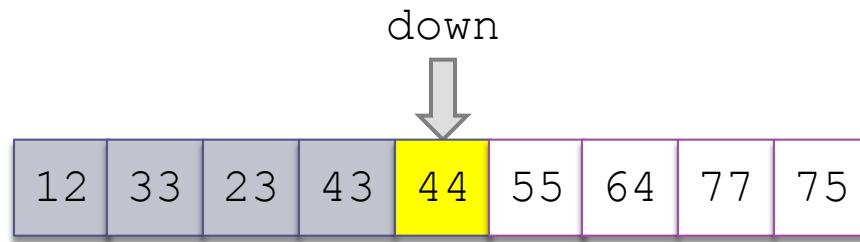
202



Exchange the pivot value with the  
value at down

# Trace of Partitioning (cont.)

203



The pivot value is in the correct position; return the value of down and assign it to the pivot index  
pivIndex

# Algorithm for Partitioning

204

## Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`
2. Initialize up to `first` and down to `last`
3. do
4.     Increment up until it selects the first element greater than the pivot value or up has reached last
5.     Decrement down until it selects the first element less than or equal to the pivot value or up has reached `first`
6.     if `up < down` then
7.         Exchange `table[up]` and `table[down]`
8. while `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`
10. Return the value of `down` to `pivIndex`

# Code for partition (cont.)

205

```
/** Partition the table so that values from first to pivIndex  
     are less than or equal to the pivot value, and values  
     from pivIndex to last are greater than the pivot value.  
     @param table The table to be partitioned  
     @param first The index of the low bound  
     @param last The index of the high bound  
     @return The location of the pivot value  
 */  
private static < T  
    extends Comparable < T >> int partition(T[] table,  
                                              int first, int last) {  
    // Select the first item as the pivot value.  
    T pivot = table[first];
```

# Code for partition (cont.)

206

```
int up = first;
int down = last;
do {
    /* Invariant:
       All items in table[first . . . up - 1] <= pivot
       All items in table[down + 1 . . . last] > pivot
    */
    while ( (up < last) && (pivot.compareTo(table[up]) >= 0) )
    {
        up++;
    }
    // assert: up equals last or table[up] > pivot.
    while (pivot.compareTo(table[down]) < 0) {
        down--;
    }
```

# Code for partition (cont.)

207

```
// assert: down equals first or table[down] <= pivot.  
    if (up < down) { // if up is to the left of down.  
        // Exchange table[up] and table[down].  
        swap(table, up, down);  
    }  
}  
  
while (up < down); // Repeat while up is left of down.  
  
// Exchange table[first] and table[down] thus putting  
// the pivot value where it belongs.  
swap(table, first, down);  
// Return the index of the pivot value.  
return down;  
}
```

# Revised Partition Algorithm

208

- Quicksort is  $O(n^2)$  when each split yields one empty subarray, which is the case when the array is presorted
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split
  - ▣ Use three references: first, middle, last
  - ▣ Select the median of the these items as the pivot

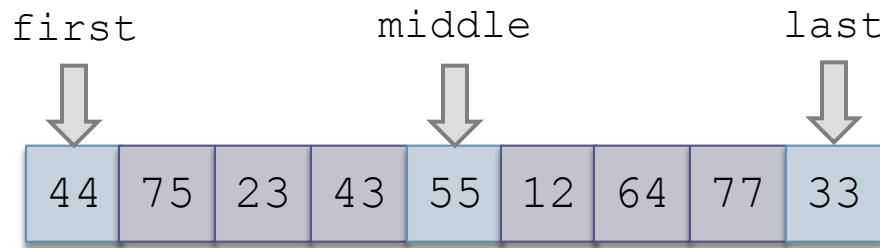
# Trace of Revised Partitioning

209

44	75	23	43	55	12	64	77	33
----	----	----	----	----	----	----	----	----

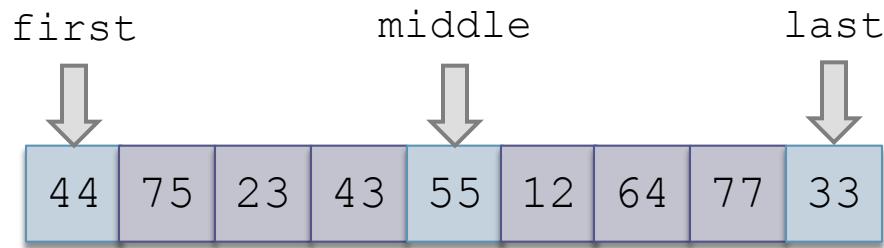
# Trace of Revised Partitioning (cont.)

210



# Trace of Revised Partitioning (cont.)

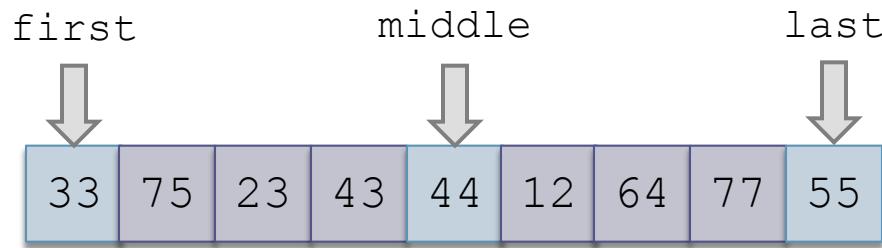
211



## Sort these values

# Trace of Revised Partitioning (cont.)

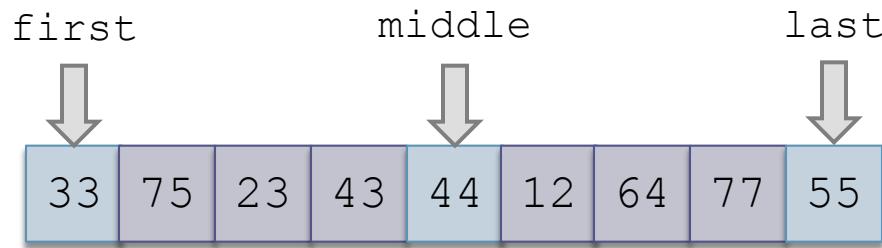
212



## Sort these values

# Trace of Revised Partitioning (cont.)

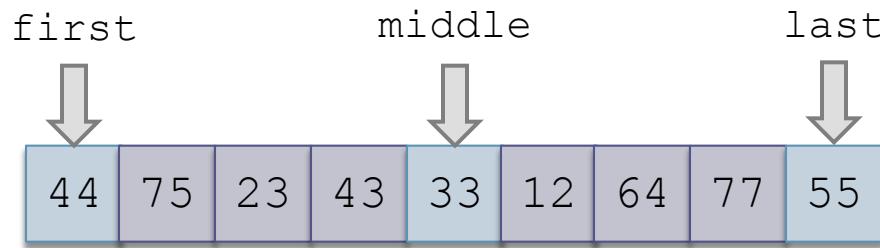
213



Exchange middle  
with first

# Trace of Revised Partitioning (cont.)

214



# Trace of Revised Partitioning (cont.)

215



Run the partition  
algorithm using the  
first element as the  
pivot

# Algorithm for Revised partition Method

216

## Algorithm for Revised partition Method

1. Sort `table[first]`, `table[middle]`, and `table[last]`
2. Move the median value to `table[first]` (the pivot value) by exchanging `table[first]` and `table[middle]`.
3. Initialize up to `first` and down to `last`

# Algorithm for Revised partition Method

217

4. do
5. Increment up until up selects the first element greater than the pivot value or up has reached last
6. Decrement down until down selects the first element less than or equal to the pivot value or down has reached first
7. if up < down then
8.     Exchange table[up] and table[down]
9. while up is to the left of down
10. Exchange table[first] and table[down]
11. Return the value of down to pivIndex

# Sort Review

218

Number of Comparisons			
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$