RECURSION

Chapter 5

Wholeness of the Lesson

Computation of a function by recursion involves repeated self-calls of the function. Recursion is implicit also at the design level when a reflexive association is present. Recursion mirrors the self-referral dynamics of consciousness, on the basis of which all creation emerges.

Chapter Objectives

- 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

Recursion

- 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

Recursive Thinking

Section 5.1

Recursive Thinking

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means
- Recursion reduces a problem into one or more simpler versions of itself

Recursive Thinking (cont.)

Recursive Algorithm to Process Nested Figures

`if` there is one figure

    do whatever is required to the figure

`else`

    do whatever is required to the outer figure

    process the figures nested inside the outer figure in the same way

Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it/them directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

Recursive Algorithm for Finding the Length of a String

`if` the string is empty (has no characters)

    the length is 0

`else`

    the length is 1 plus the length of the string that excludes the first character

Recursive Algorithm for Finding the Length of a String (cont.)

```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
   if (str == null || str.equals(""))
      return 0;
   else
      return 1 + length(str.substring(1));
}
```

Recursive Algorithm for Printing String Characters

```
/** Recursive method printChars
    post: The argument string is displayed, one
    character per line
    @param str The string
*/
public static void printChars(String str) {
    if (str == null || str.equals(""))
        return;
    else {
        System.out.println(str.charAt(0));
        printChars(str.substring(1));
    }
}
```

Recursive Algorithm for Printing String Characters in Reverse

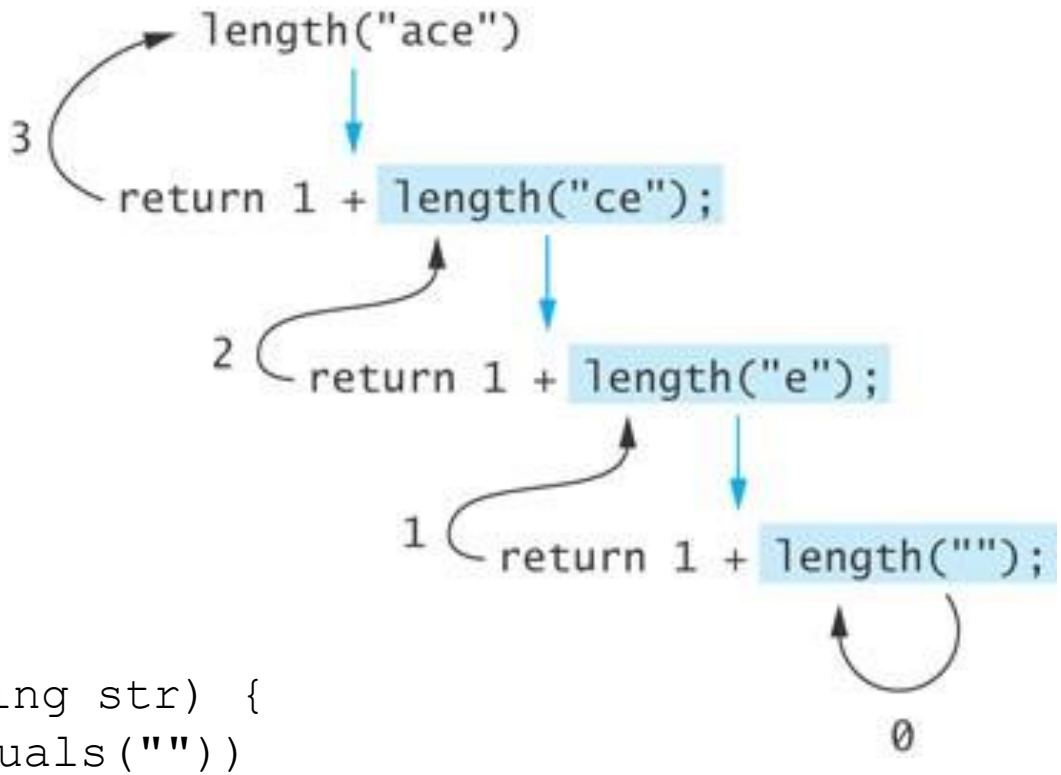
```
/** Recursive method printCharsReverse
    post: The argument string is displayed in reverse,
          one character per line
    @param str The string
*/
public static void printCharsReverse(String str) {
    if (str == null || str.equals(""))
        return;
    else {
        printCharsReverse(str.substring(1));
        System.out.println(str.charAt(0));
    }
}
```

Proving that a Recursive Method is Correct

- Proof by induction
  - Prove the theorem is true for the base case
  - Show that if the theorem is assumed true for n, then it must be true for n+1
- Recursive proof is similar to induction
  - Verify the base case is recognized and solved correctly
  - Verify that each recursive case makes progress towards the base case
  - Verify that if all smaller problems are solved correctly, then the original problem also is solved correctly

Tracing a Recursive Method

- The process of returning from recursive calls and computing the partial results is called *unwinding the recursion*



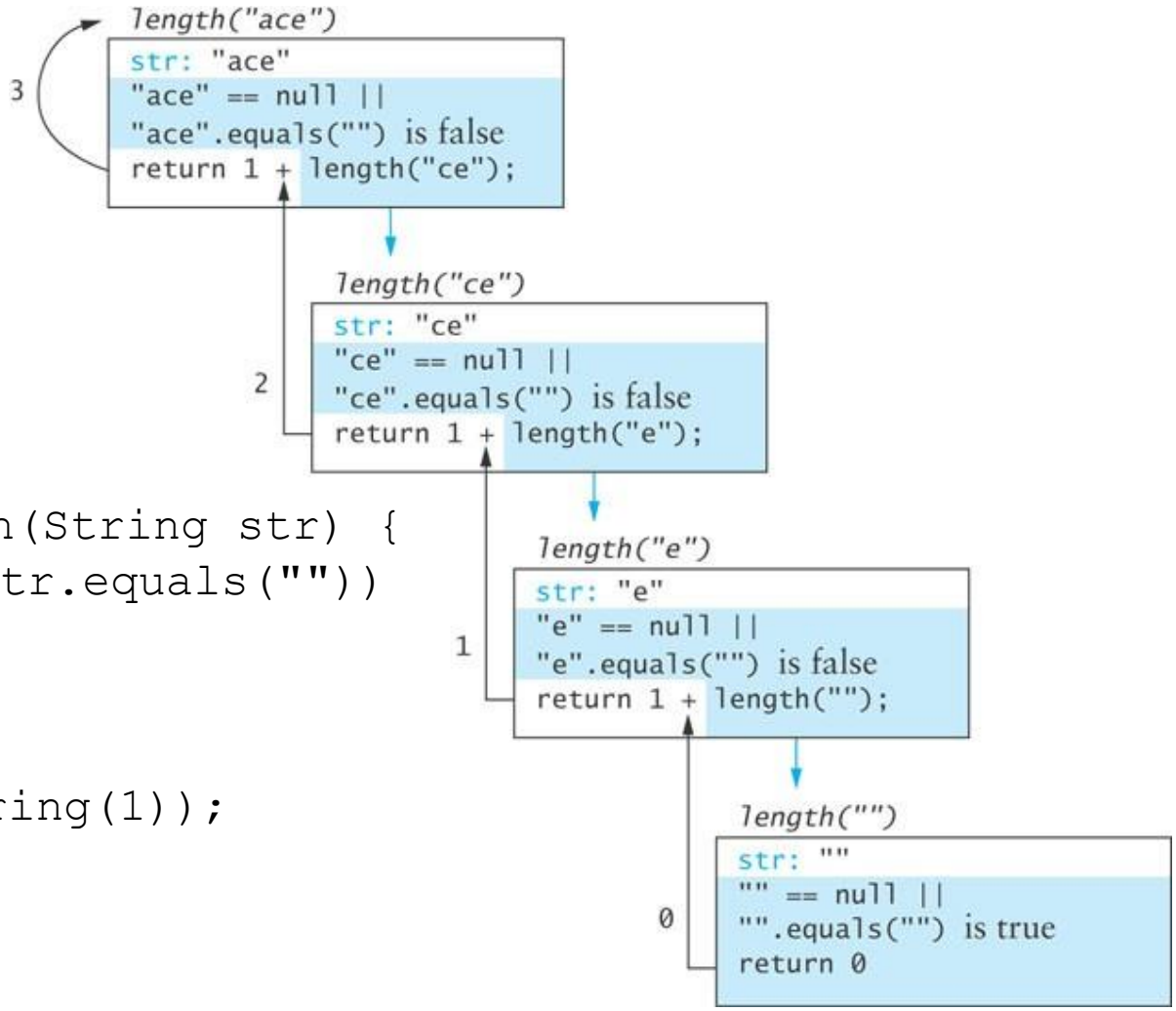

```
public static int length(String str) {
    if (str == null || str.equals(""))
        return 0;
    else
        return 1 + length(str.substring(1));
}
```

Run-Time Stack and Activation Frames

- Java maintains a run-time stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack

Run-Time Stack and Activation Frames

```
public static int length(String str) {
    if (str == null || str.equals(""))
        return 0;
    else
        return 1 +
        length(str.substring(1));
}
```



Run-Time Stack and Activation Frames

Exercise : Trace the runtime stack information and draw the activation frame for printReverse() method.

Let take the input of “JAVA” and print it in reversed order. Refer previous slide to complete this task.

Recursive Definitions of Mathematical Formulas

Section 5.2

Recursive Definitions of Mathematical Formulas

- Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms
- Examples include:
  - factorials
  - powers
  - greatest common divisors (gcd)
  - Fibonacci

Factorial of *n*: *n*!

- The factorial of *n*, or *n*! is defined as follows:

  $0! = 1$

  $n! = n \times (n-1)!$ $(n > 0)$

- The base case: *n* is equal to 0
- The second formula is a recursive definition

Factorial of *n*: *n*! (cont.)

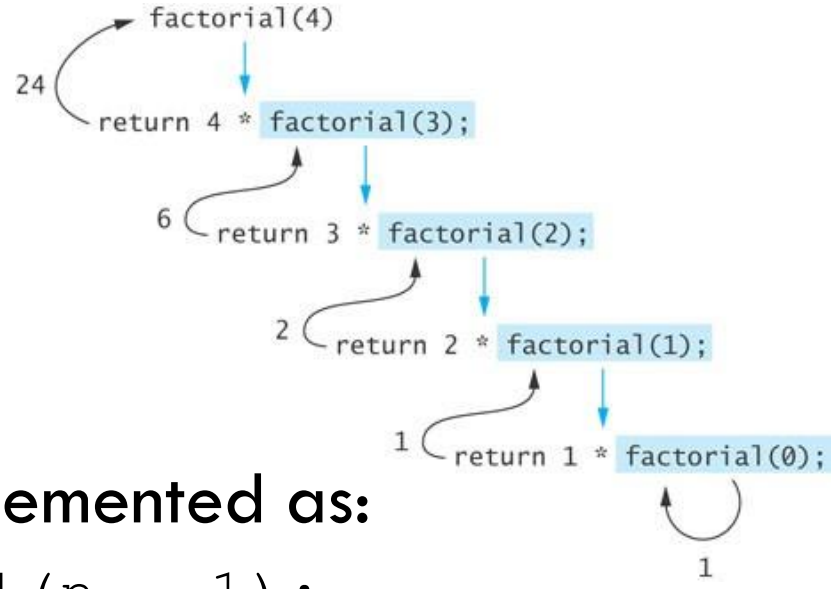
- The recursive definition can be expressed by the following algorithm:

  `if` *n* equals 0

  n! is 1

  `else`

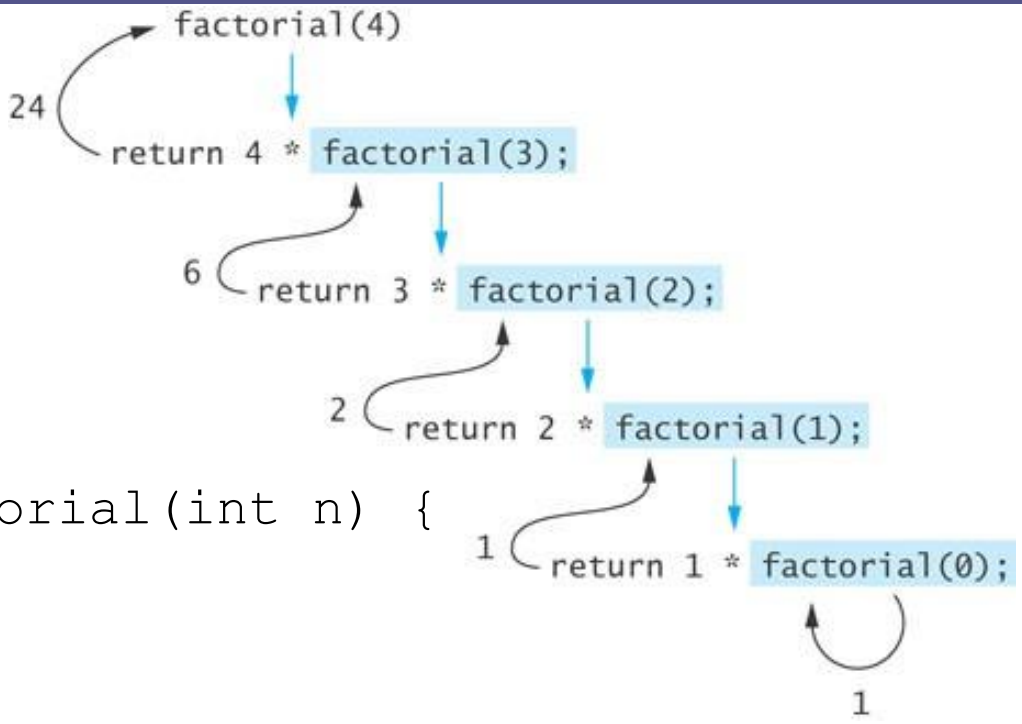
  $n! = n \times (n - 1)!$



- The last step can be implemented as:

```
return n * factorial(n - 1);
```

Factorial of *n*: *n*! (cont.)




```
public static int factorial(int n) {
if (n == 0)
return 1;
else
return n * factorial(n - 1);
}
```

Infinite Recursion and Stack Overflow

- If you call method `factorial` with a negative argument, the recursion will not terminate because `n` will never equal `0`
- If a program does not terminate, it will eventually throw the `StackOverflowError` exception
- Make sure your recursive methods are constructed so that a stopping case is always reached
- In the `factorial` method, you could throw an `IllegalArgumentException` if `n` is negative

Recursive Algorithm for Calculating x^n

Recursive Algorithm for Calculating x^n ($n \geq 0$)

if *n is 0*

The result is 1

else

The result is $x \times x^{n-1}$

```
/** Recursive power method (in RecursiveMethods.java).
    pre: n >= 0
    @param x The number being raised to a power
    @param n The exponent
    @return x raised to the power n
*/
public static double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n - 1);
}
```

Recursive Algorithm for Calculating gcd

- The greatest common divisor (gcd) of two numbers is the largest integer that divides both numbers
- The gcd of 20 and 15 is 5
- The gcd of 36 and 24 is 12
- The gcd of 38 and 18 is 2
- The gcd of 17 and 97 is 1

Recursive Algorithm for Calculating **gcd** (cont.)

- Given 2 positive integers m and n (m > n)

  `if` n is a divisor of m

  gcd(m, n) = n

  `else`

  gcd (m, n) = gcd (n, m % n)

Recursive Algorithm for Calculating gcd (cont.)

```
/** Recursive gcd method (in RecursiveMethods.java).
    pre: m > 0 and n > 0
    @param m The larger number
    @param n The smaller number
    @return Greatest common divisor of m and n
*/
public static double gcd(int m, int n) {
    if (m % n == 0)
        return n;
    else if (m < n)
        return gcd(n, m); // Transpose arguments.
    else
        return gcd(n, m % n);
}
```

Recursion Versus Iteration

- There are similarities between recursion and iteration
- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case
- You can always write an iterative solution to a problem that is solvable by recursion
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Iterative factorial Method

```
/** Iterative factorial method.
    pre: n >= 0
    @param n The integer whose factorial is being computed
    @return n!
*/
public static int factorialIter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
        result = result * k;
    return result;
}
```

```
// Recursive Factorial
public static int factorial(int n) {
if (n == 0)
return 1;
else
return n * factorial(n - 1);
}
```

Efficiency of Recursion

- Recursive methods often have slower execution times relative to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency usually does not outweigh the advantage of readable code that is easy to debug

Fibonacci Numbers

- Fibonacci numbers were used to model the growth of a rabbit colony

$fib_1 = 1$

$fib_2 = 1$

$fib_n = fib_{n-1} + fib_{n-2}$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

```
int fib(int n) {
        if(n == 0 || n == 1) {
                return n;
        }
        return fib(n-1) + fib(n-2);
}
```

Recursive Array Search

Section 5.3

Recursive Array Search

- Searching an array can be accomplished using recursion
- Simplest way to search is a linear search
  - Examine one element at a time starting with the first element and ending with the last
  - On average, $(n + 1)/2$ elements are examined to find the target in a linear search
  - If the target is not in the list, n elements are examined
- A linear search is $O(n)$

Recursive Array Search (cont.)

- Base cases for recursive search:
  - Empty array, target can not be found; result is -1
  - First element of the array being searched = target; result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

Algorithm for Recursive Linear Array Search

Algorithm for Recursive Linear Array Search

`if` the array is empty

    the result is –1

`else if` the first element matches the target

    the result is the subscript of the first element

`else`

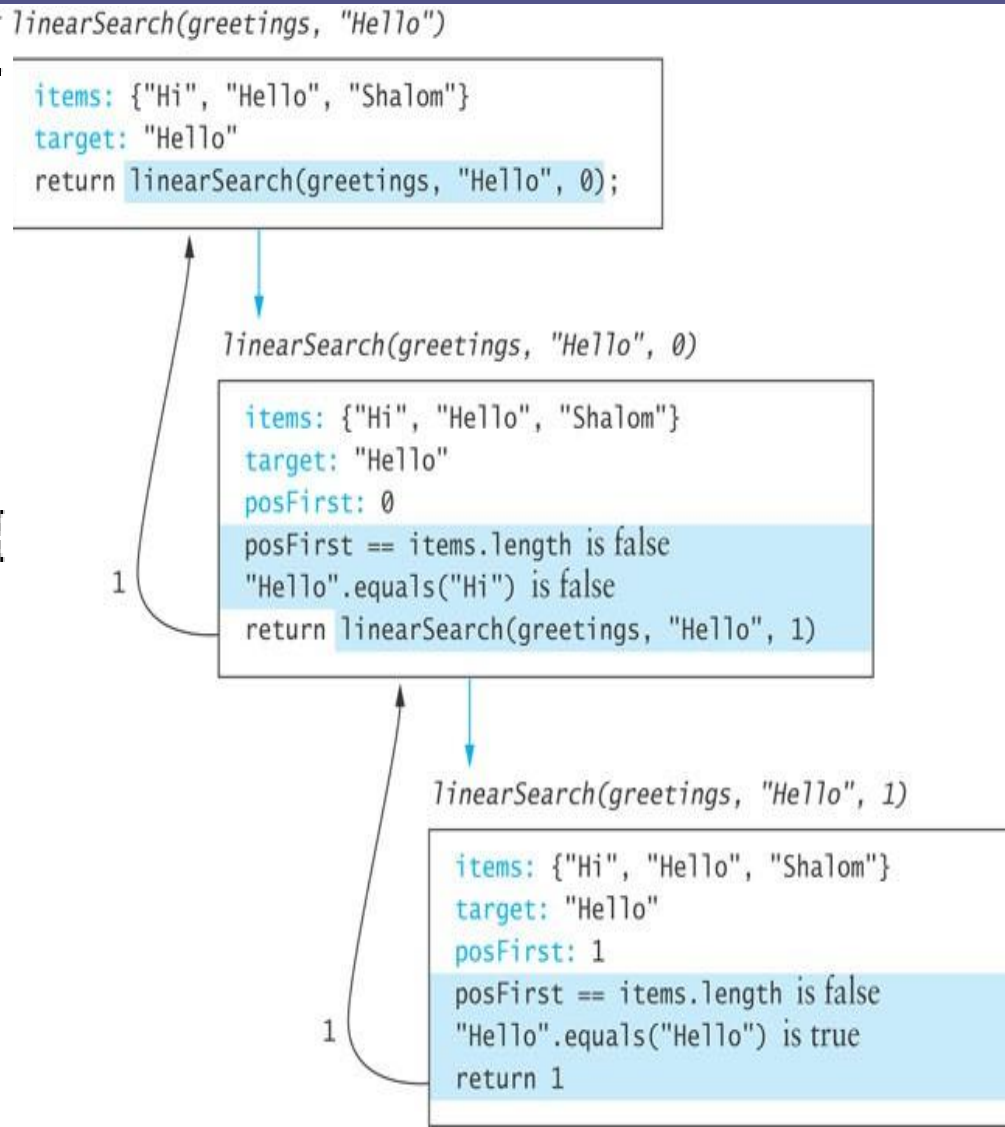
    search the array excluding the first element and return the result

Implementation of Recursive Linear Search

```
/** Recursive linear search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The item being searched for
    @param posFirst The position of the current first element
    @return The subscript of target if found; otherwise -1
*/
private static int linearSearch(Object[] items,
                                Object target, int posFirst) {
    if (posFirst == items.length)
        return -1;
    else if (target.equals(items[posFirst]))
        return posFirst;
    else
        return linearSearch(items, target, posFirst + 1);
}
```

Implementation of Recursive Linear Search (cont.)

```
/** Recursive linear search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The item being searched for
    @param posFirst The position of the current first element
    @return The subscript of target if found; otherwise -1
*/
private static int linearSearch(Object[] items,
                                Object target, int posFirst) {
    if (posFirst == items.length)
        return -1;
    else if (target.equals(items[posFirst]))
        return posFirst;
    else
        return linearSearch(items, target, posFirst + 1);
}
```



Design of a Binary Search Algorithm

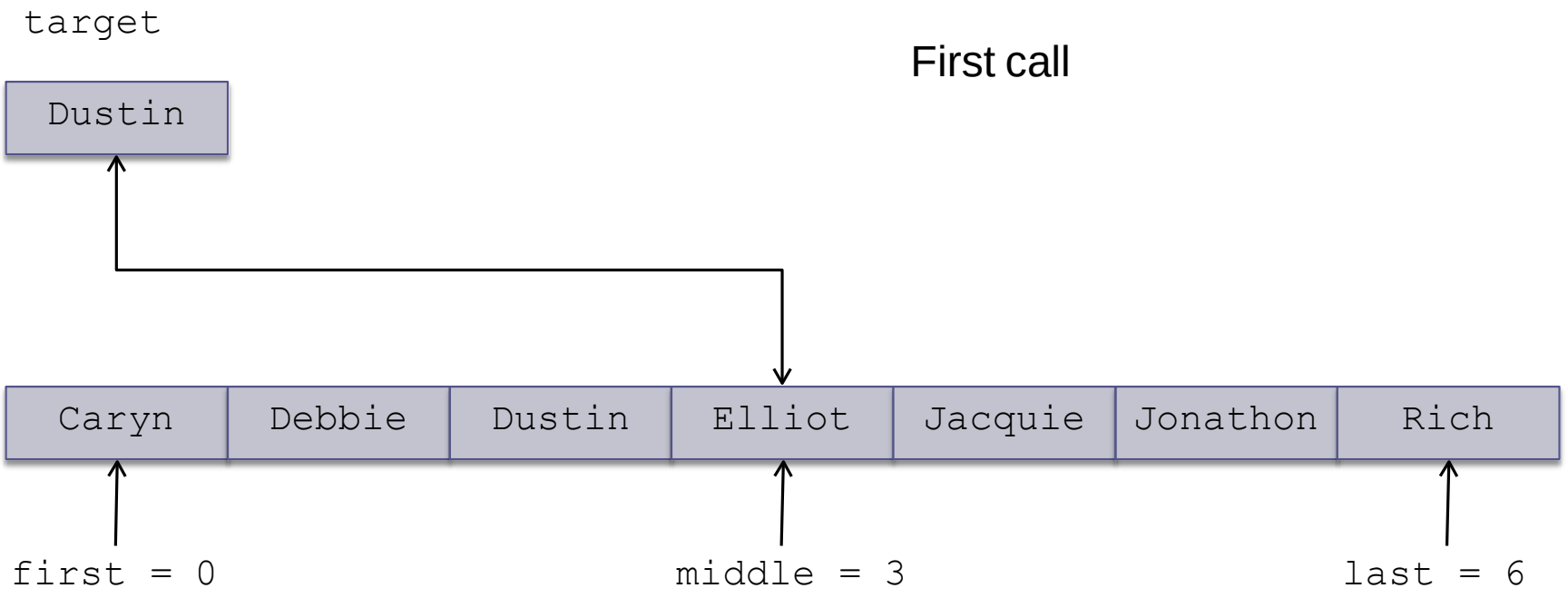
- A binary search can be performed only on an array that has been sorted
- Base cases
  - The array is empty
  - Element being examined matches the target
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- If the middle element does not match the target, a binary search excludes the half of the array within which the target cannot lie

Design of a Binary Search Algorithm (cont.)

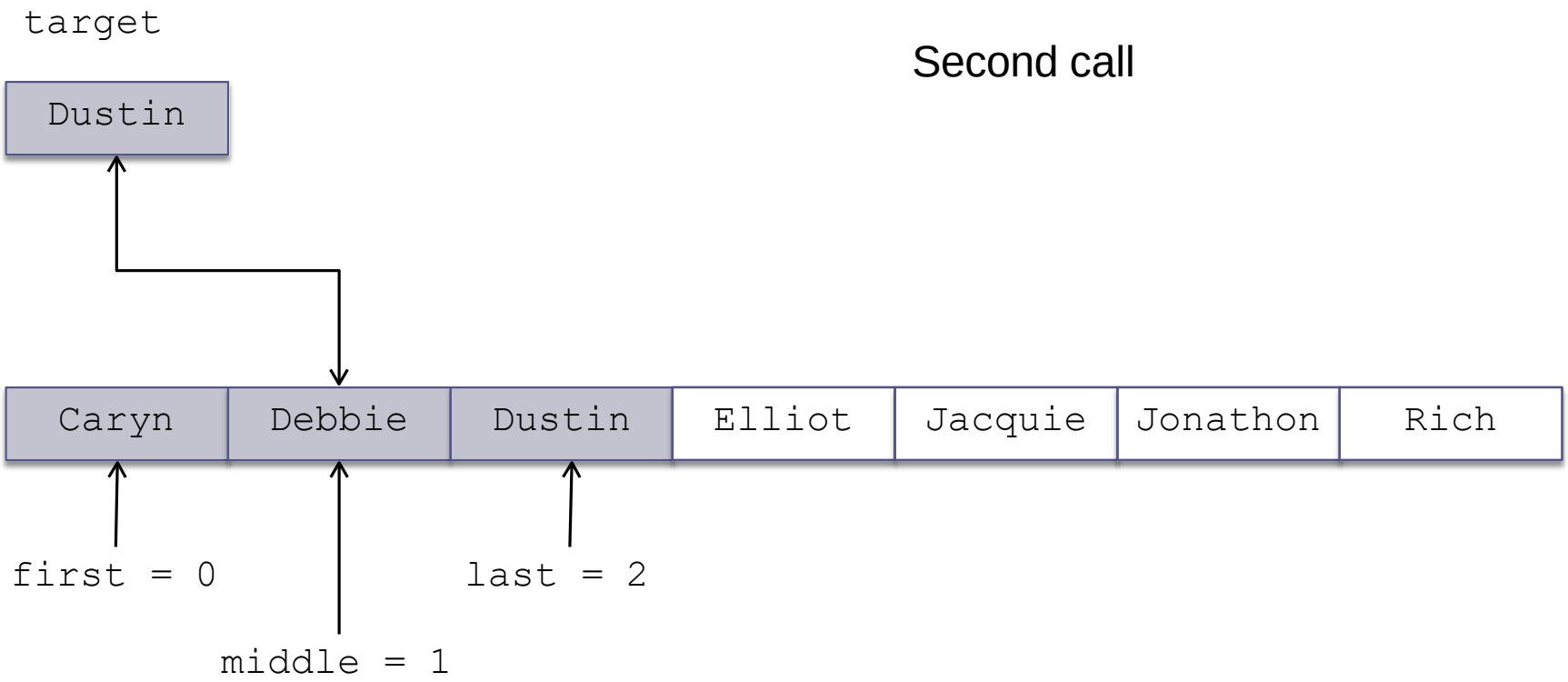
Binary Search Algorithm

```
if the array is empty
      return –1 as the search result
else if the middle element matches the target
      return the subscript of the middle element as the result
else if the target is less than the middle element
      recursively search the array elements before the middle element
      and return the result
else
      recursively search the array elements after the middle element and
      return the result
```

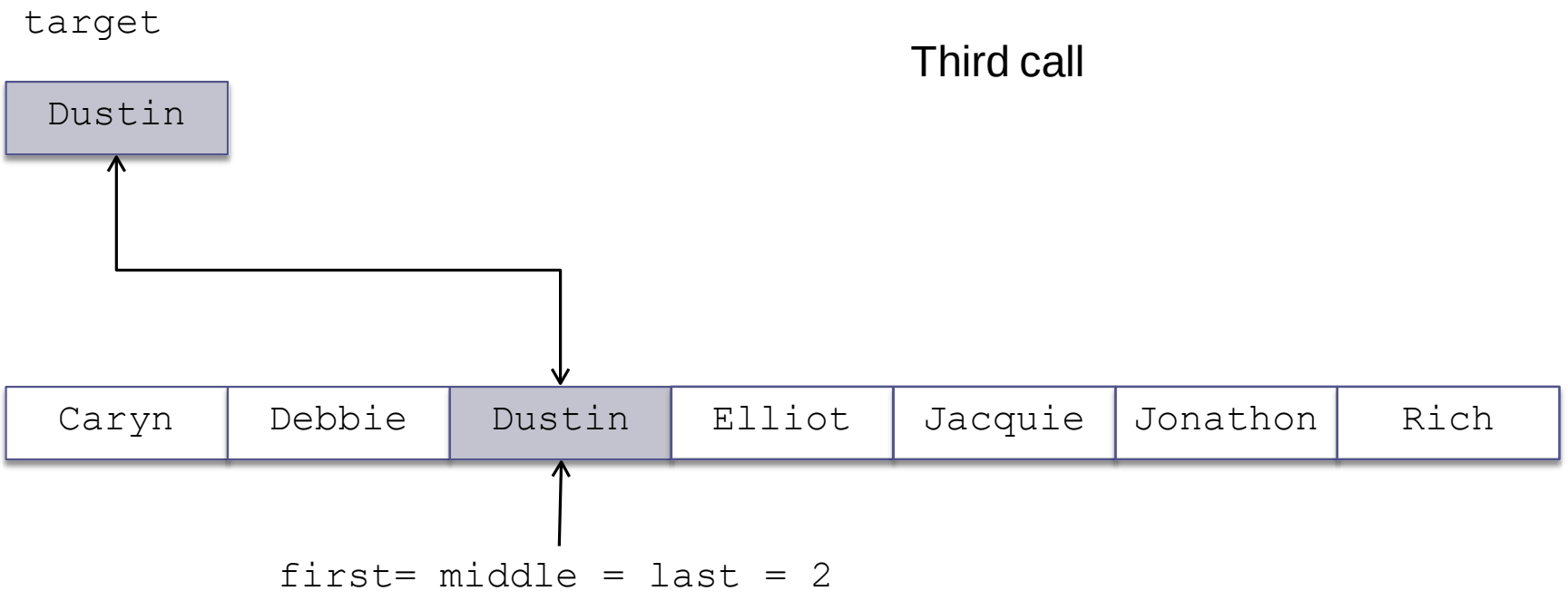
Binary Search Algorithm



Binary Search Algorithm (cont.)



Binary Search Algorithm (cont.)



Efficiency of Binary Search

- At each recursive call we eliminate half the array elements from consideration, making a binary search O(log *n*)
- An array of 16 would search arrays of length 16, 8, 4, 2, and 1: 5 probes in the worst case
  - $16 = 2^4$
  - $5 = \log_2 16 + 1$
- A doubled array size would require only 6 probes in the worst case
  - $32 = 2^5$
  - $6 = \log_2 32 + 1$
- An array with 32,768 elements requires only 16 probes! ($\log_2 32768 = 15$)

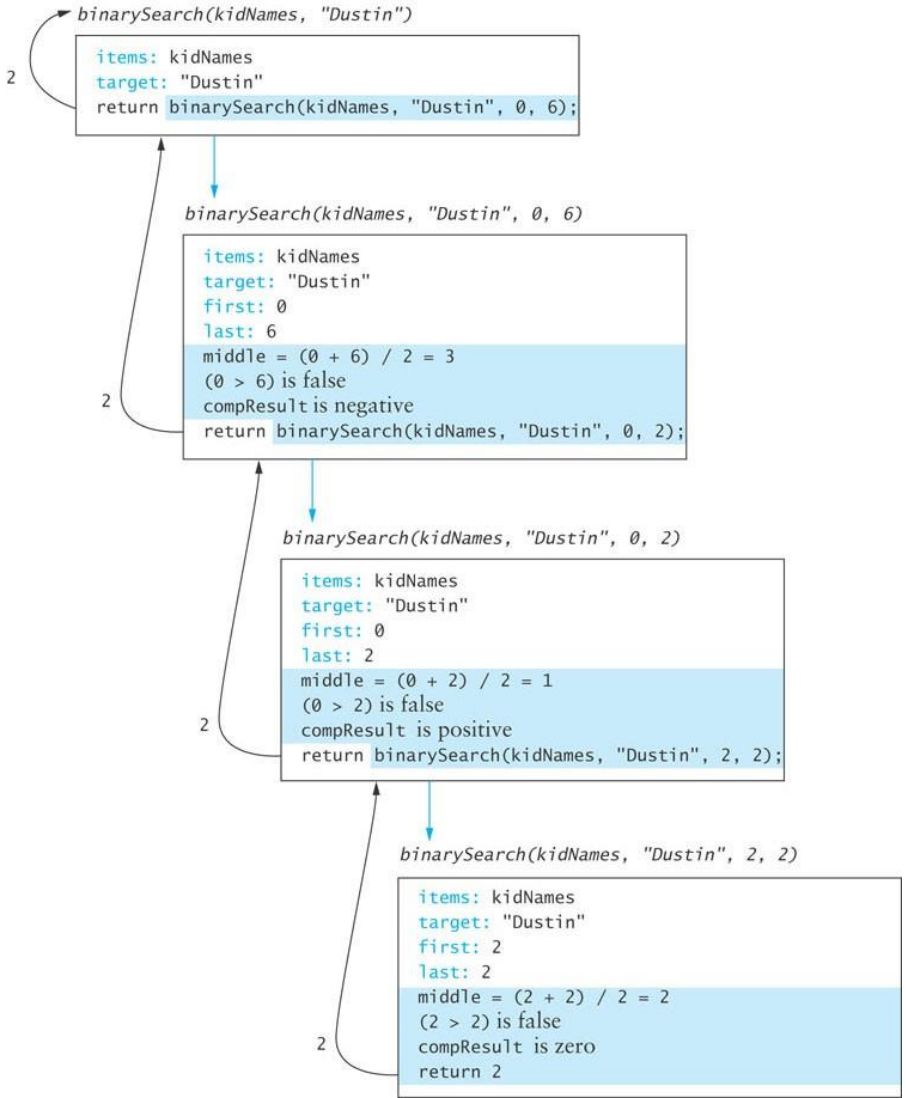
`Comparable` Interface

- Classes that implement the `Comparable` interface must define a `compareTo` method
- Method call `obj1.compareTo(obj2)` returns an integer with the following values
  - negative if `obj1 < obj2`
  - zero if `obj1 == obj2`
  - positive if `obj1 > obj2`
- Implementing the `Comparable` interface is an efficient way to compare objects during a search

Implementation of a Binary Search Algorithm

```
/** Recursive binary search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The object being searched for
    @param first The subscript of the first element
    @param last The subscript of the last element
    @return The subscript of target if found; otherwise -1.
*/
private static int binarySearch(Object[] items, Comparable target,
                                int first, int last) {
    if (first > last)
        return -1;      // Base case for unsuccessful search.
    else {
        int middle = (first + last) / 2;  // Next probe index.
        int compResult = target.compareTo(items[middle]);
        if (compResult == 0)
            return middle;   // Base case for successful search.
        else if (compResult < 0)
            return binarySearch(items, target, first, middle - 1);
        else
            return binarySearch(items, target, middle + 1, last);
    }
}
```

Trace of Binary Search



Testing Binary Search

- You should test arrays with
  - an even number of elements
  - an odd number of elements
  - duplicate elements
- Test each array for the following cases:
  - the target is the element at each position of the array, starting with the first position and ending with the last position
  - the target is less than the smallest array element
  - the target is greater than the largest array element
  - the target is a value between each pair of items in the array

Method `Arrays.binarySearch`

- Java API class `Arrays` contains a `binarySearch` method
  - Called with sorted arrays of primitive types or with sorted arrays of objects
  - If the objects in the array are not mutually comparable or if the array is not sorted, the results are undefined
  - If there are multiple copies of the target value in the array, there is no guarantee which one will be found
  - Throws `ClassCastException` if the target is not comparable to the array elements

Recursive Data Structures

Section 5.4

Recursive Data Structures

- Computer scientists often encounter data structures that are defined recursively – each 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

- 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, that stores data and a reference to a linked list

Class `LinkedListRec`

- 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

```
/** 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

```
/** 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

```
/** 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

```
/** 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);
}
```