Advanced Programming Techniques in Java

COSI 12B

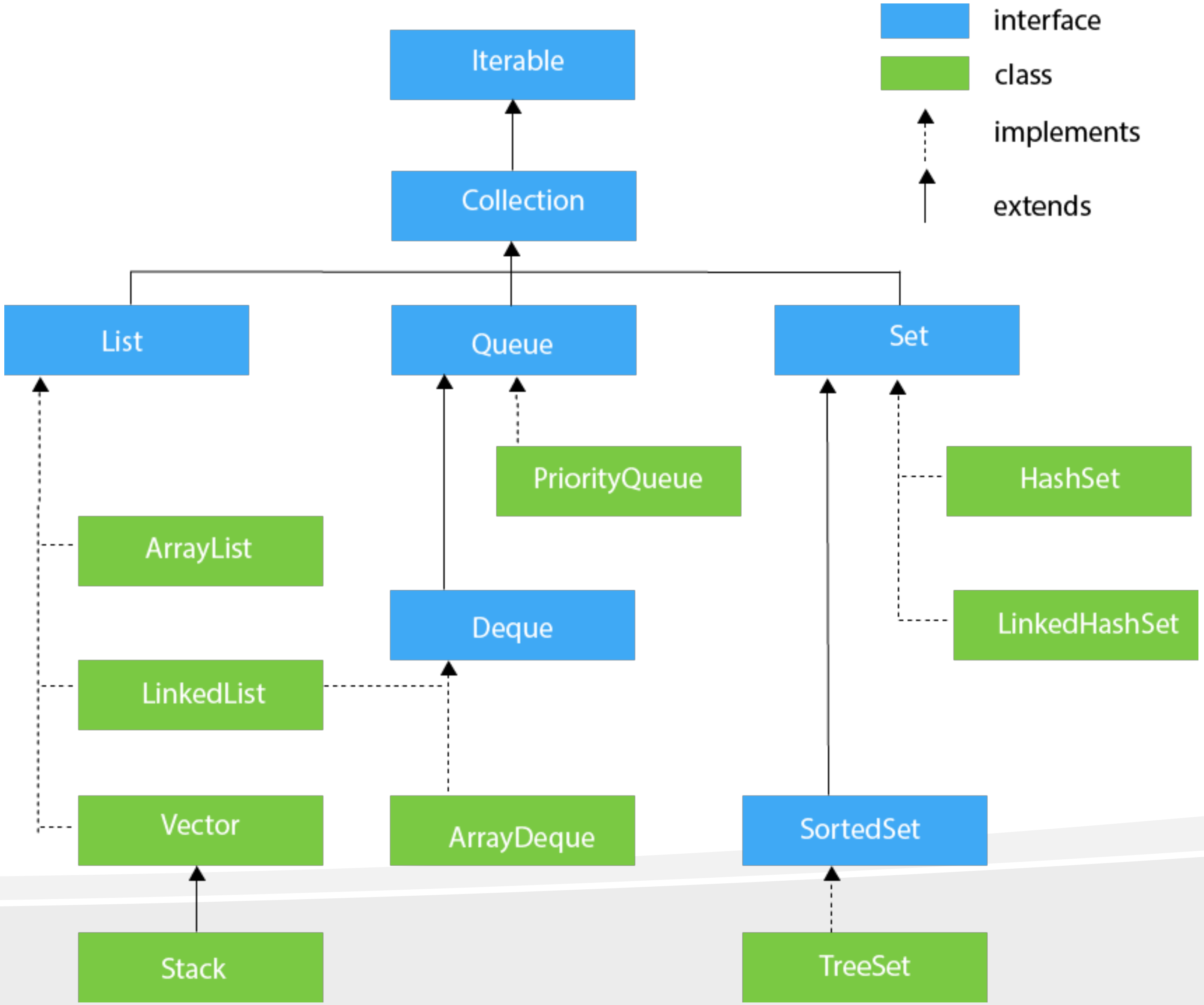
Recursion

Lecture 20

Class Objectives

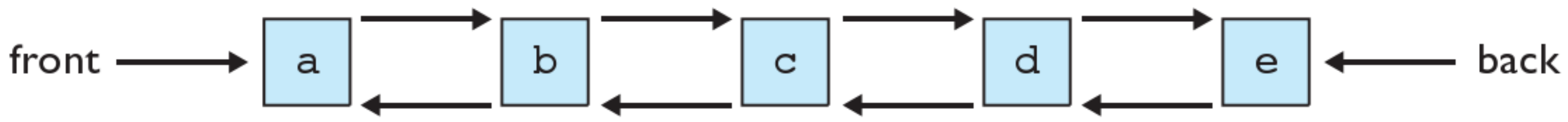
- Recursion (Sections 12.1-12.3)

Review: Collections Framework Diagram



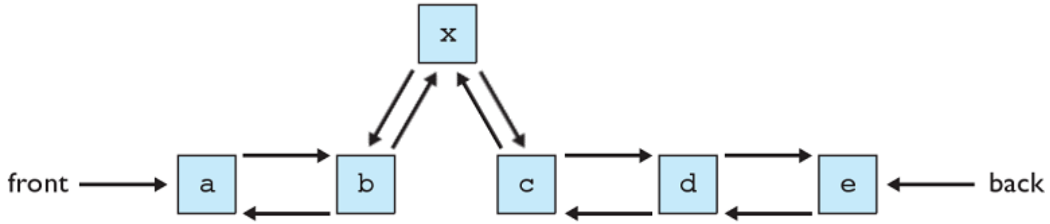
Review: Linked list

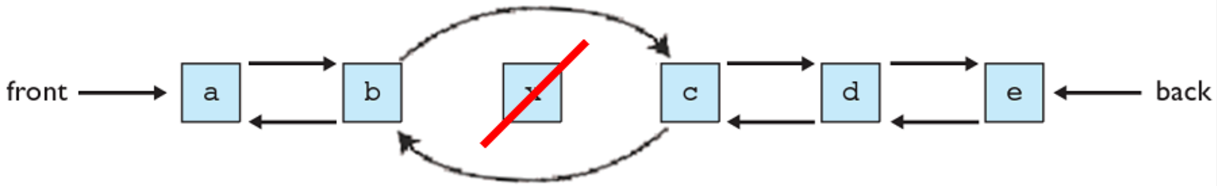
- **Linked list** is a list implemented using a linked sequence of values
  - Each value is stored in a small object called a **node**, which also contains references to its neighbor nodes
  - The list keeps a reference to the first and/or last node
  - In Java, represented by the class LinkedList



Review: Linked list performance

- To add, remove, get a value at a given index:
  - The list must advance through the list to the node just before the one with the proper index
  - For example to add a new value to the list, the list creates a new node, walks along its existing node links to the proper index, and attaches it to the nodes that should precede and follow it
  - This is very fast when adding to the front or back of the list (because the list contains references to these places), but slow elsewhere





Review: Linked List Implementation – Inner Classes

```
public class myLinkedList<E> {
  private Node<E> head;
  private Node<E> tail;
  private int size;

  private static class Node<E>{
          private E data;
          private Node<E> next;
          private Node<E> previous;

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

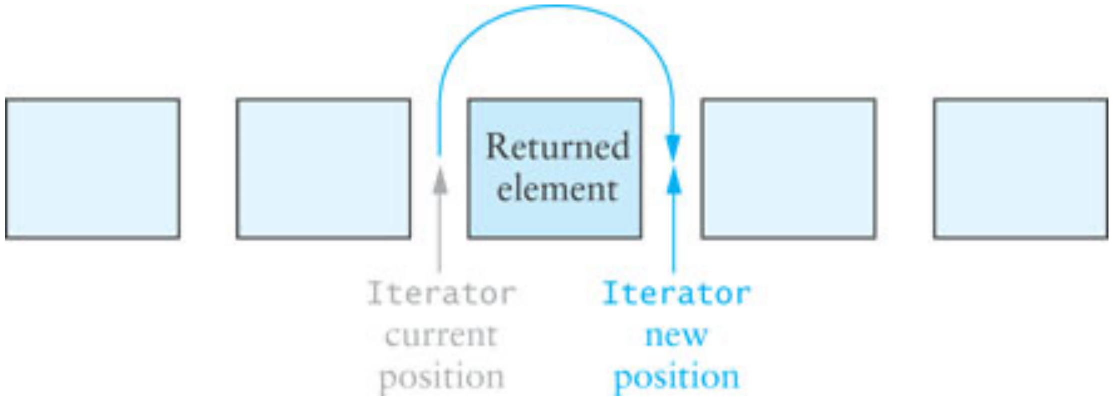
Generally, all details of the `Node` class should be private. This applies also to the data fields and constructors.

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

Static inner classes are also called *nested classes*

Review: Iterator Position

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



Review: Benefits of iterators

- Speed up loops over lists' elements
  - Implemented for both `ArrayLists` and `LinkedLists`
  - Makes more sense to use it for `LinkedLists` since `get` operations is cheap in `ArrayList`
- A unified way to examine all elements of a collection
  - Every collection in Java has an `iterator` method
  - In fact, that's the *only* guaranteed way to examine the elements of any `Collection`
- Don't have to use indexes

Review: The `ListIterator<E>` interface

- Extends the `Iterator` interface
- The `LinkedList` class implements the `List<E>` interface using a doubly-linked list
- Methods in `LinkedList` that return a `ListIterator`:
  - `public ListIterator<E> listIterator()`
  - `public ListIterator<E> listIterator(int index)`
- Methods in the `ListIterator` interface:
  - `add, hasNext, hasPrevious, next, previous, nextIndex, previousIndex, remove, set`

Abstract Data Types (ADTs)

- **Abstract data type (ADT)** is a general specification of a data structure
  - Specifies what data the data structure can hold
  - Specifies what operations can be performed on the data
  - Does NOT know how the data structure hold the data internally, nor how it implements each operation
- Example ADT: **List**
  - Specifies that a list collection will store elements in order with integer indexes (allowing duplicates and null values)
  - Specifies that a list collection supports `add`, `remove`, `get(index)`, `set(index)`, `size`, `isEmpty`, …
- …

Abstract Data Types (ADTs)

- `ArrayList` and `LinkedList` both implement the data/operations specified by the **list** ADT

- ADTs in Java are specified by interfaces
  - `ArrayList` and `LinkedList` both implement `List` interface

More on ADTs

- Good practice is to use the appropriate interface type rather than the class type
  - `List<Integer> list = new LinkedList<Integer>();`
  - Gives flexibility to change implementations of the list

- You can use the interface type `List` when declaring parameters, return types or fields

Strengths

- `ArrayList`
  - Random access; any element can be accessed quickly
  - Adding or removing at the end of the list is fast

- `LinkedList`
  - Sequential access, `get/remove/add` fast only with an iterator(or ListIterator)
  - Adding and removing at either end of the list is fast
  - No need to expand an array when full

List limitation

- Slow to search
  - You have to look for elements sequentially

- It is not easy to prevent a list from storing duplicates
  - You have to sequentially search the list on every add operation
  - Make sure you are not adding an element that is already there

`ArrayList` vs. `LinkedList`

- Both implements `List` interface and maintains insertion order

ArrayList	LinkedList
Uses a **dynamic array** to store the elements	Uses a **doubly linked list** to store the elements
Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, shifting is required.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no shifting is required.
An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Recursive Thinking

Recursion

- **recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.

- **recursive programming**: Writing methods that call themselves to solve problems recursively.
  - An equally powerful substitute for *iteration* (loops)
  - Particularly well-suited to solving certain types of problems

Why learn recursion?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, OCaml and Haskell) use recursion exclusively (no loops)

Recursive Thinking

- Consider searching for a target value in an array
  - Assume the array elements are sorted in increasing order
  - We compare the target to the middle element and, if the middle element does not match the target, search either the elements before the middle element or the elements after the middle element
  - Instead of searching *n* elements, we search *n*/2 elements

Recursive Thinking (cont.)

Recursive Algorithm to Search an Array

`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

Steps to Design a Recursive Algorithm

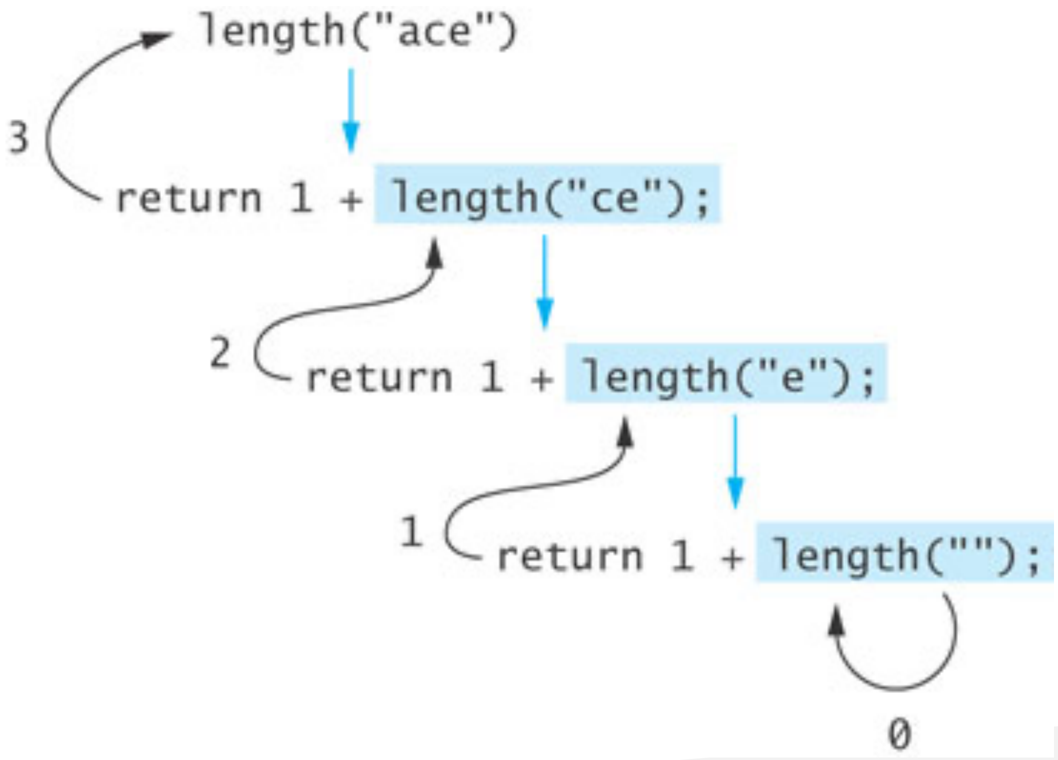
- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case and provide a solution to it
- Devise a strategy to 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

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

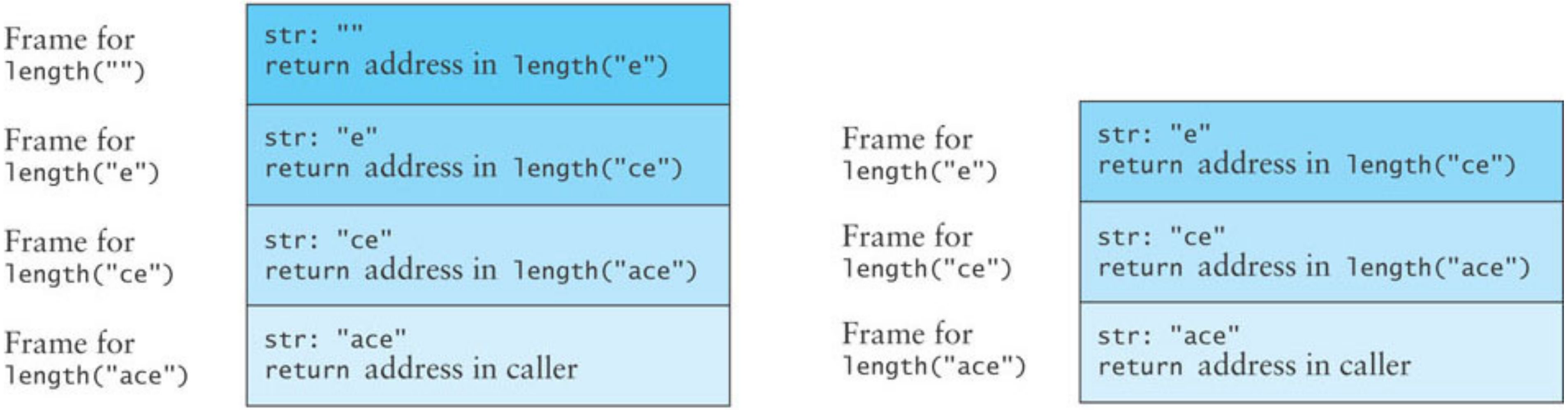
- The process of returning from recursive calls and computing the partial results is called *unwinding the recursion*



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

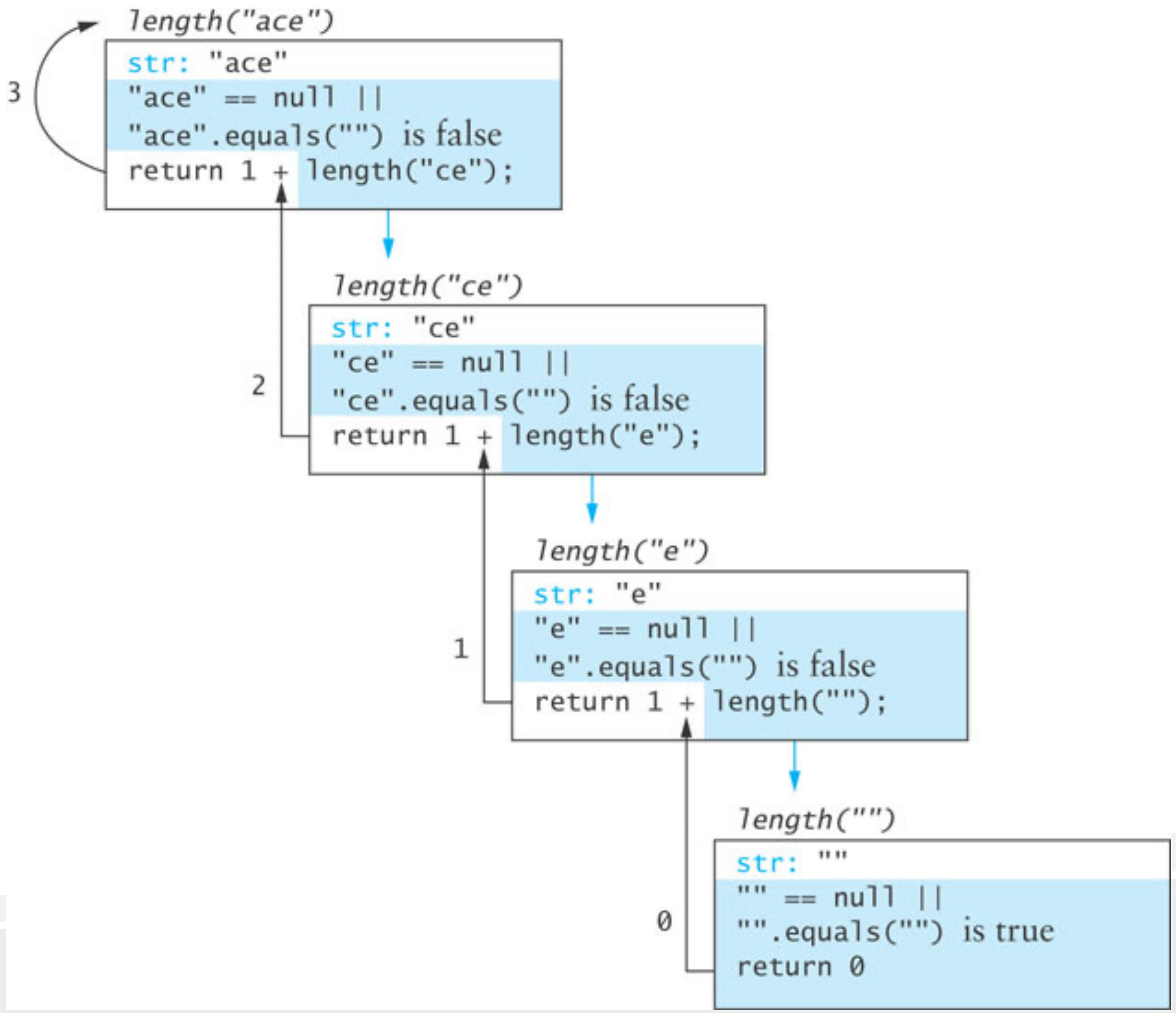
Run-Time Stack and Activation Frames (cont.)



Run-time stack after all calls

Run-time stack after return from last call

Run-Time Stack and Activation Frames



Recursive Definitions of Mathematical Formulas

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)

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* 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

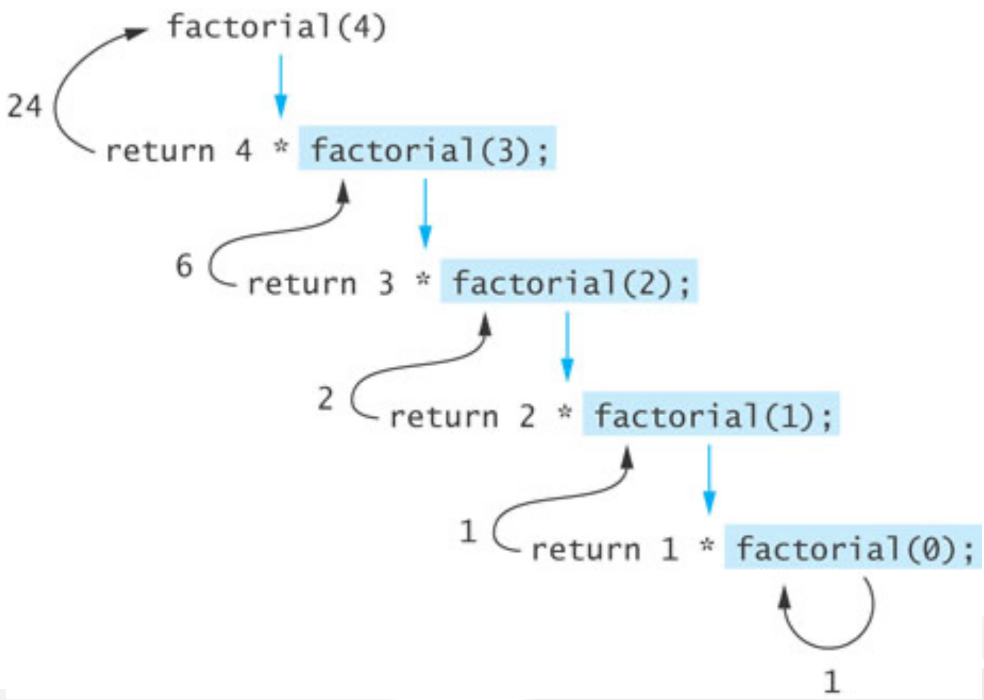
  $n!$ is 1

  else

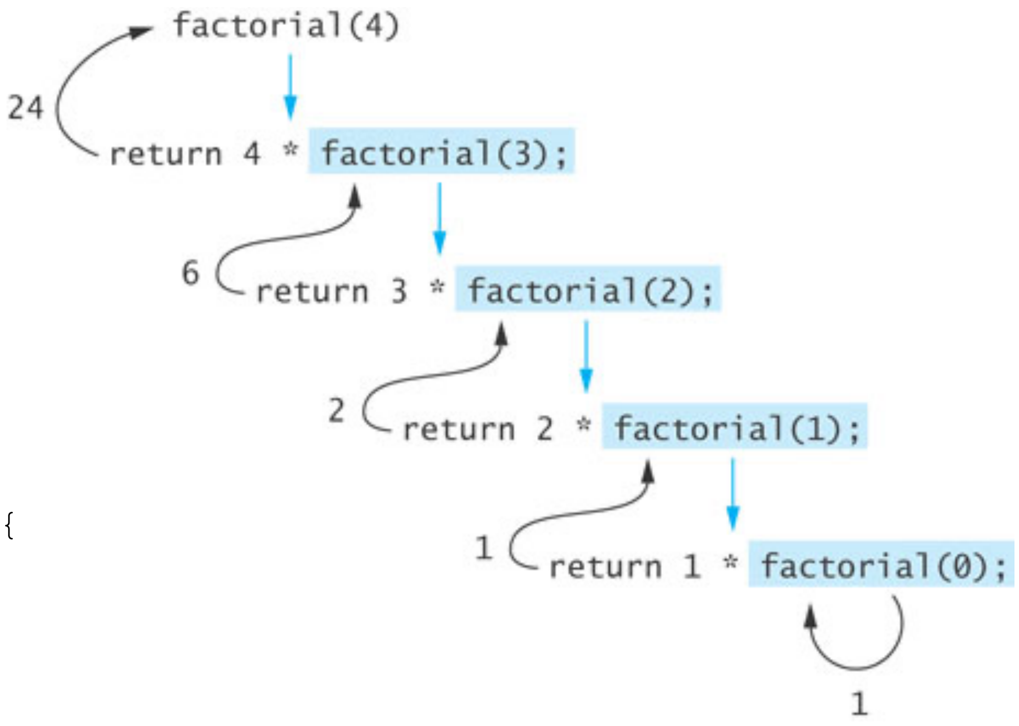
  $n! = n \times (n - 1)!$

- 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 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

Recursive Algorithm for Calculating gcd (cont.)

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

  if n is a divisor of m

  $\quad\quad$ gcd(m, n) = n

  else

  $\quad\quad$ 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;
}
```

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 does not outweigh the advantage of readable code that is easy to debug

Fibonacci Numbers

- The Fibonacci numbers are a sequence defined as follows

  $fib_1 = 1$

  $fib_2 = 1$

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

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

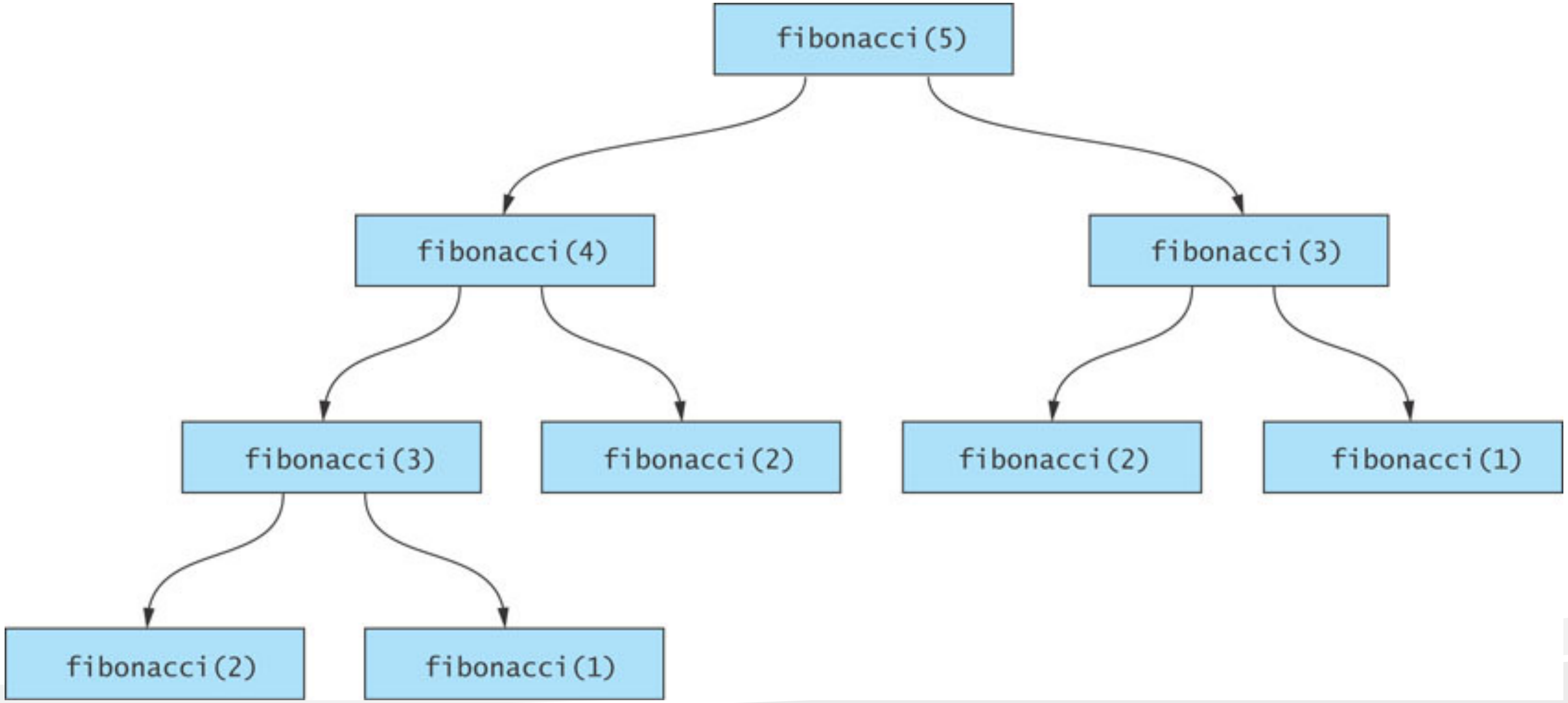
An Exponential Recursive `fibonacci` Method

```
/** Recursive method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
*/
public static int fibonacci(int n) {
     if (n <= 2)
          return 1;
     else
          return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Efficiency of Recursion: Exponential

`fibonacci`

Inefficient:
exponential complexity



An O(n) Recursive `fibonacci` Method

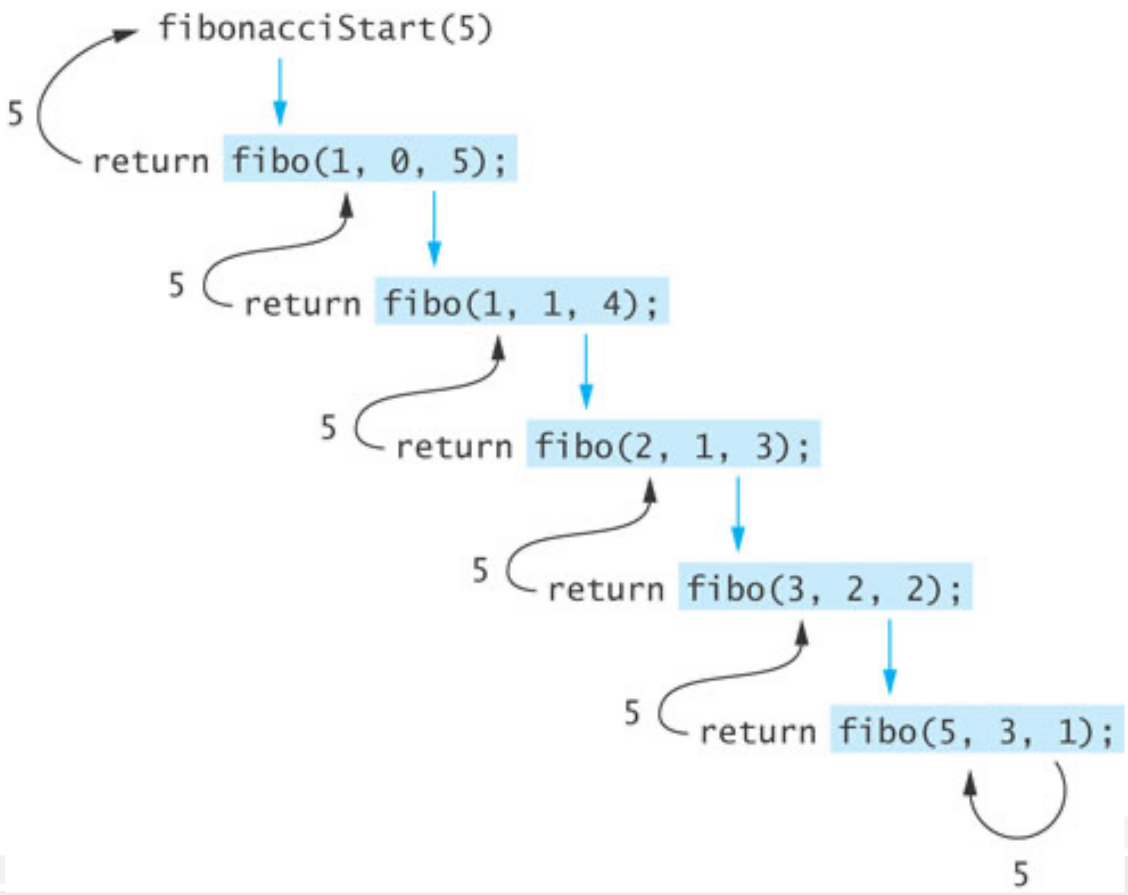
```java
/** Recursive O(n) method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param fibCurrent The current Fibonacci number
    @param fibPrevious The previous Fibonacci number
    @param n The count of Fibonacci numbers left to calculate
    @return The value of the Fibonacci number calculated so far
*/
private static int fibo(int fibCurrent, int fibPrevious, int n) {
    if (n == 1)
        return fibCurrent;
    else
        return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
}
```

An O(n) Recursive `fibonacci` Method (cont.)

- In order to start the method execution, we provide a non-recursive wrapper method:

```
/** Wrapper method for calculating Fibonacci numbers (in
RecursiveMethods.java).

    pre: n >= 1

    @param n The position of the desired Fibonacci       number

    @return  The value of the nth Fibonacci number

*/

public static int fibonacciStart(int n) {

     return fibo(1, 0, n);

}
```

Efficiency of Recursion: O(n) `fibonacci`



Efficient

Efficiency of Recursion: O(n) `fibonacci`

- Method `fibo` is an example of *tail recursion* or *last-line recursion*
- When recursive call is the last line of the method, arguments and local variables do not need to be saved in the activation frame

Recursive Array Search

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 /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
```

Design of a Binary Search Algorithm

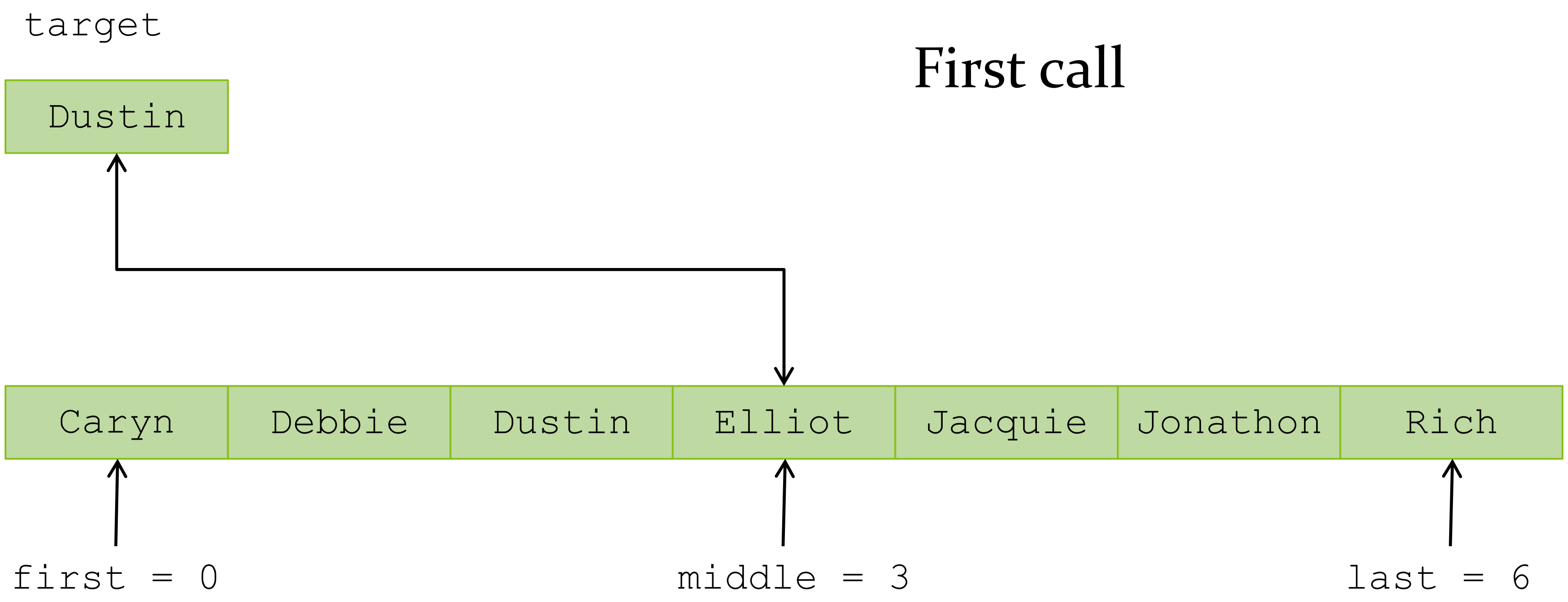
- A binary search can be performed only on an array that has been sorted
- Base cases
  - The array is empty
  - Element being examined matches the target
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- A binary search excludes the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

Design of a Binary Search Algorithm (cont.)

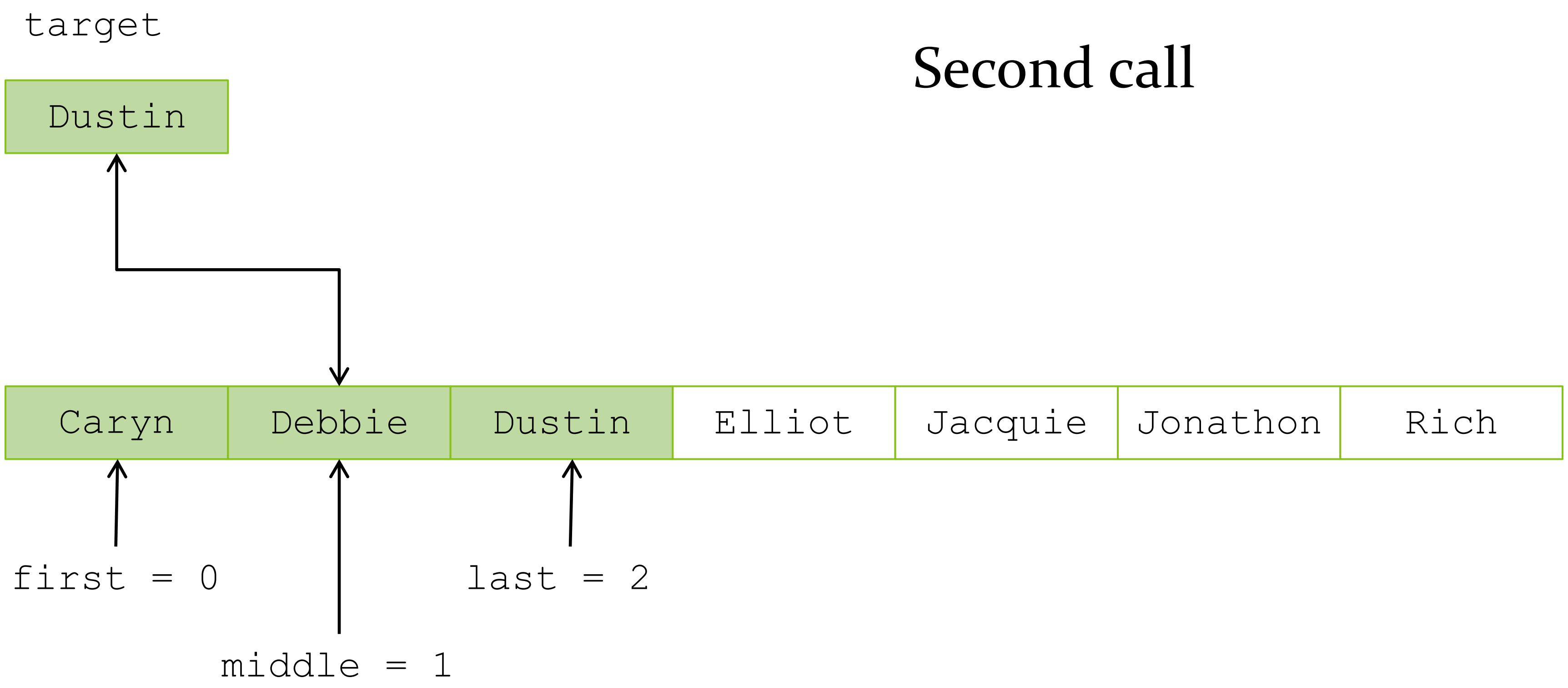
Binary Search Algorithm

`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

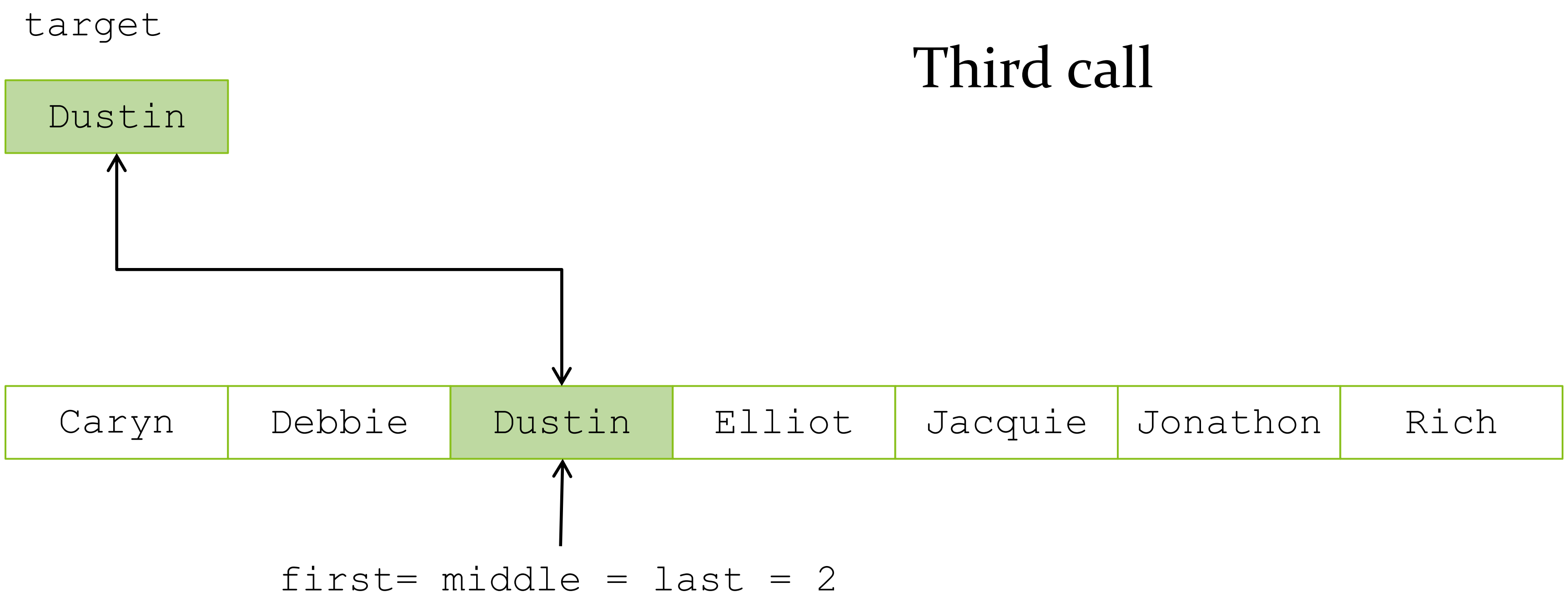


Binary Search Algorithm (cont.)



Binary Search Algorithm (cont.)

Third call



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 only require 6 probes in the worst case
  - $32 = 2^5$
  - $6 = \log_2 32 + 1$
- An array with 32,768 elements requires only 16 probes! ($\log_2 32768 = 15$)

Implementation of a Binary Search Algorithm

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

Implementation of a Binary Search Algorithm (cont.)

```
/** Wrapper for recursive binary search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The object being searched for
    @return The subscript of target if found; otherwise -1.
*/
public static int binarySearch(Object[] items, Comparable target) {
    return binarySearch(items, target, 0, items.length - 1);
}
```

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