Recursion

Objectives

- To understand how to think recursively
- To learn how to write recursive methods
- To learn how to trace a recursive method
- To learn how to write recursive algorithms and methods for searching arrays

RECURSIVE THINKING

Recursive Thinking

Maya and the dragon

Boy and Dragon ©2009-2015 doodlediddy

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 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));
    }
}
```

Recursive Algorithm for Finding the Length of a String

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

    the length is 0

`else`

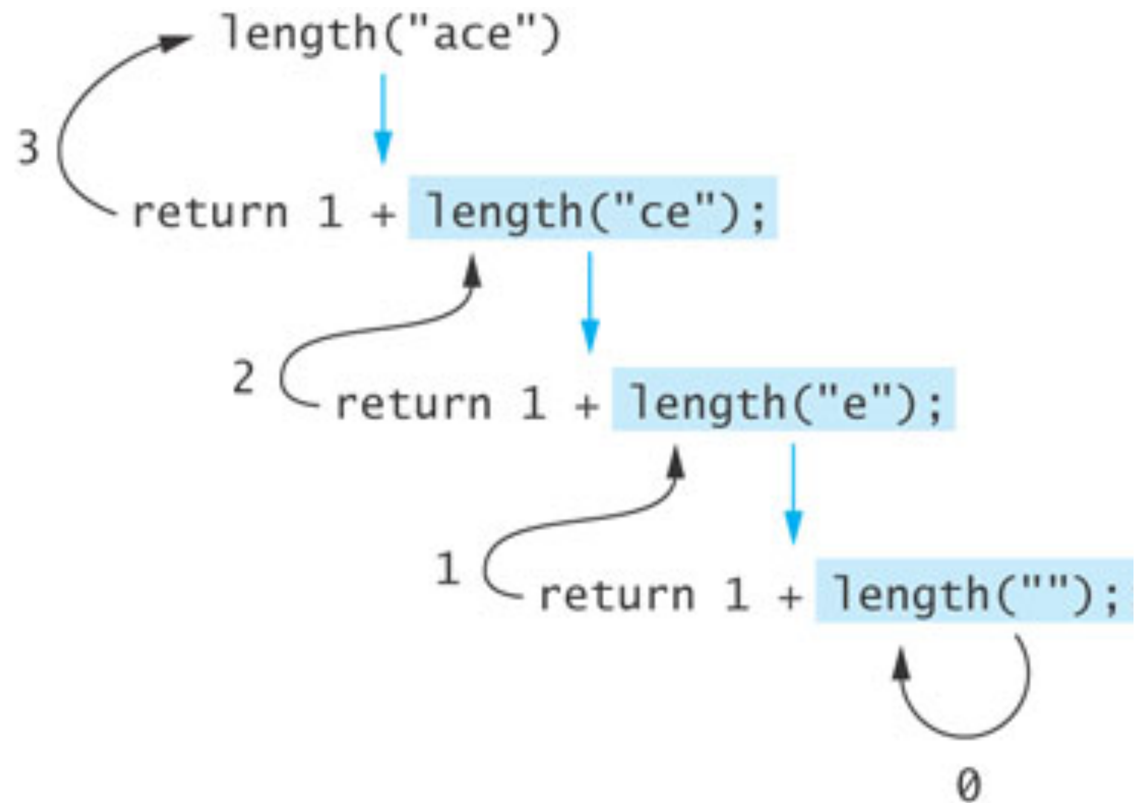
    the length is 1 plus the length of the string that excludes the first character

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));
}
```

Tracing a Recursive Method

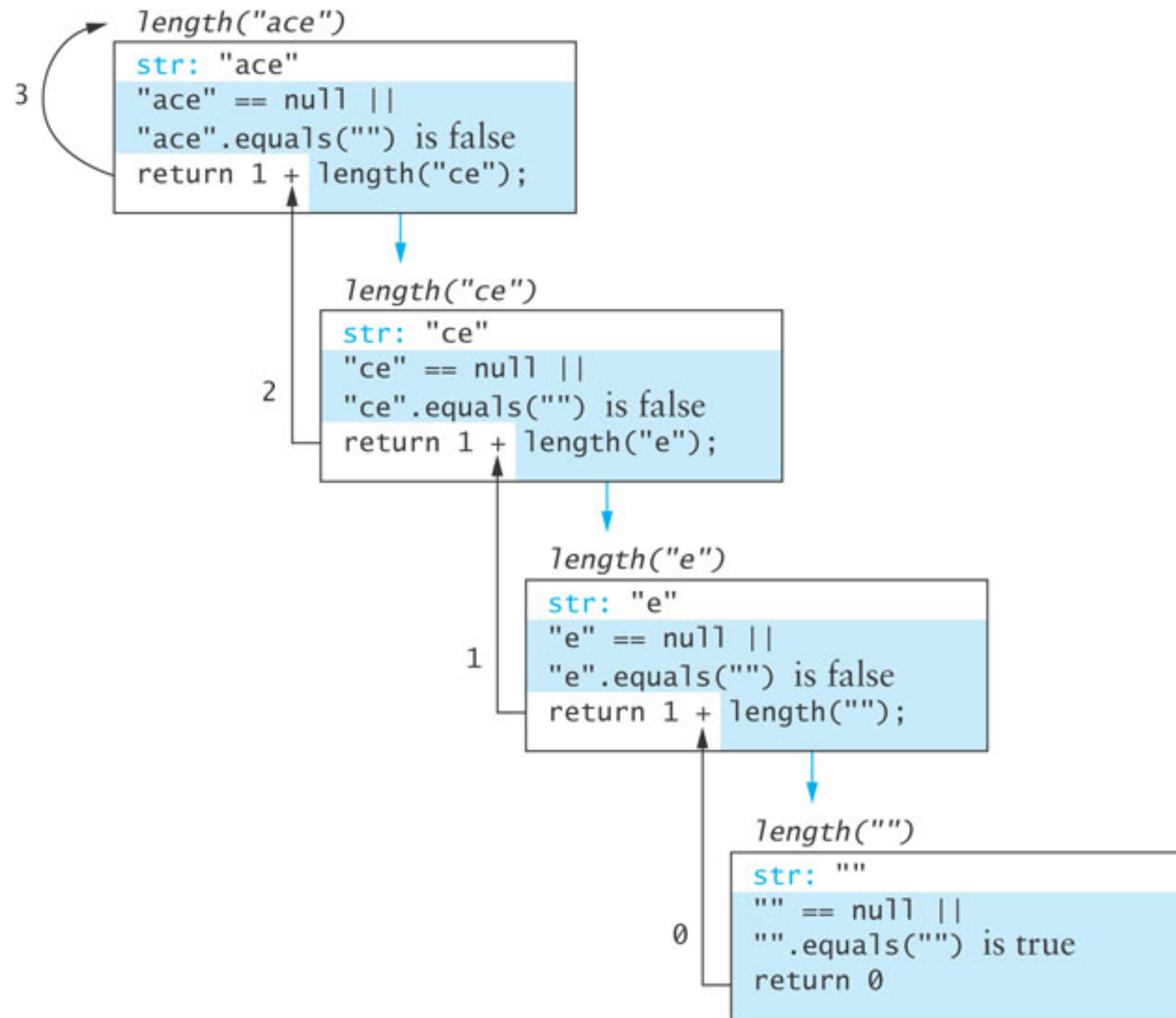
- The process of returning from recursive calls and computing the partial results is called *unwinding the recursion*



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
  - function 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



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, $(n + 1)/2$ 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.)

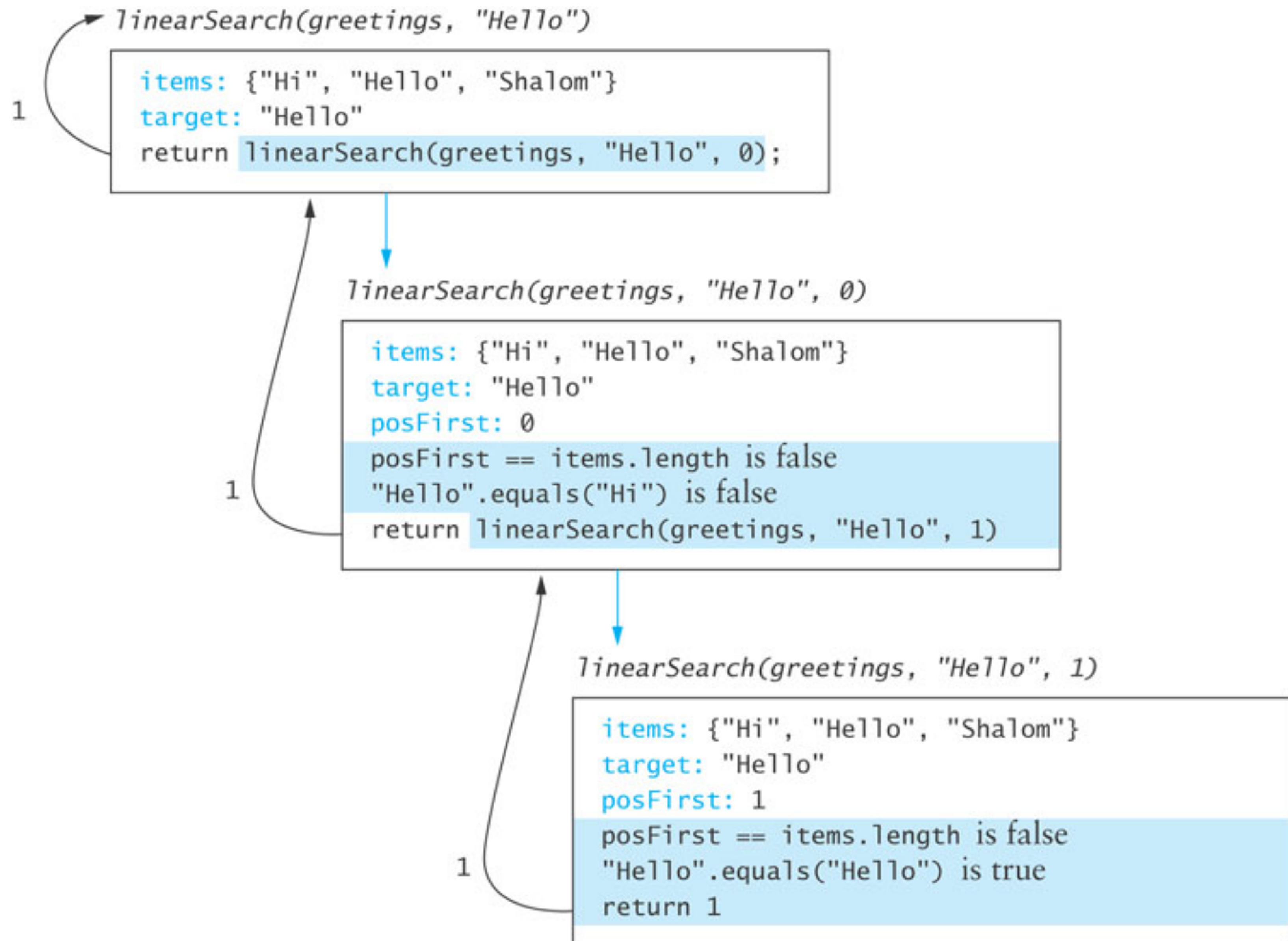
- Base cases for recursive search:
  - Empty array, target can not be found; result is -1
  - First element of the array being searched = target; result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

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 position of the first element
else
    search the array excluding the first element and return the result
```

Implementation of Recursive Linear Search (cont.)



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

target

Dustin

First call

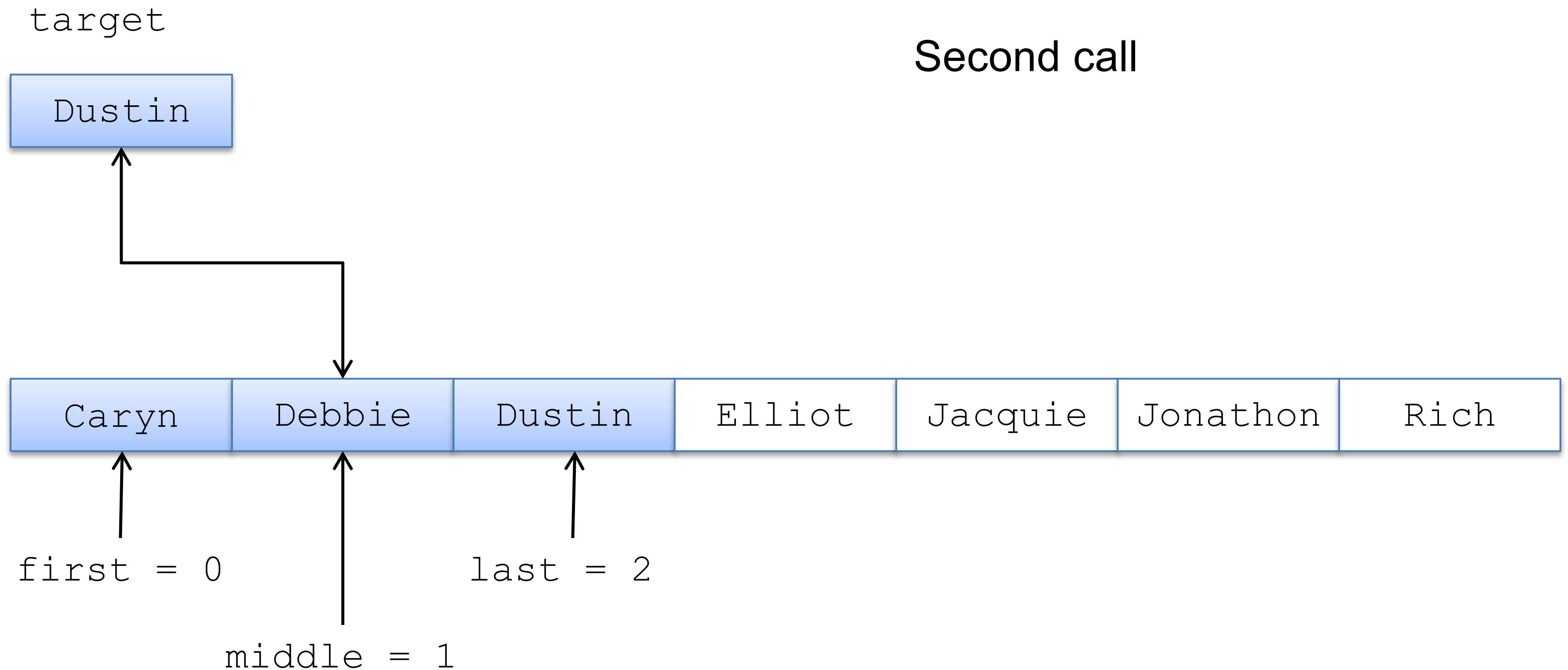
Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathon	Rich

first = 0

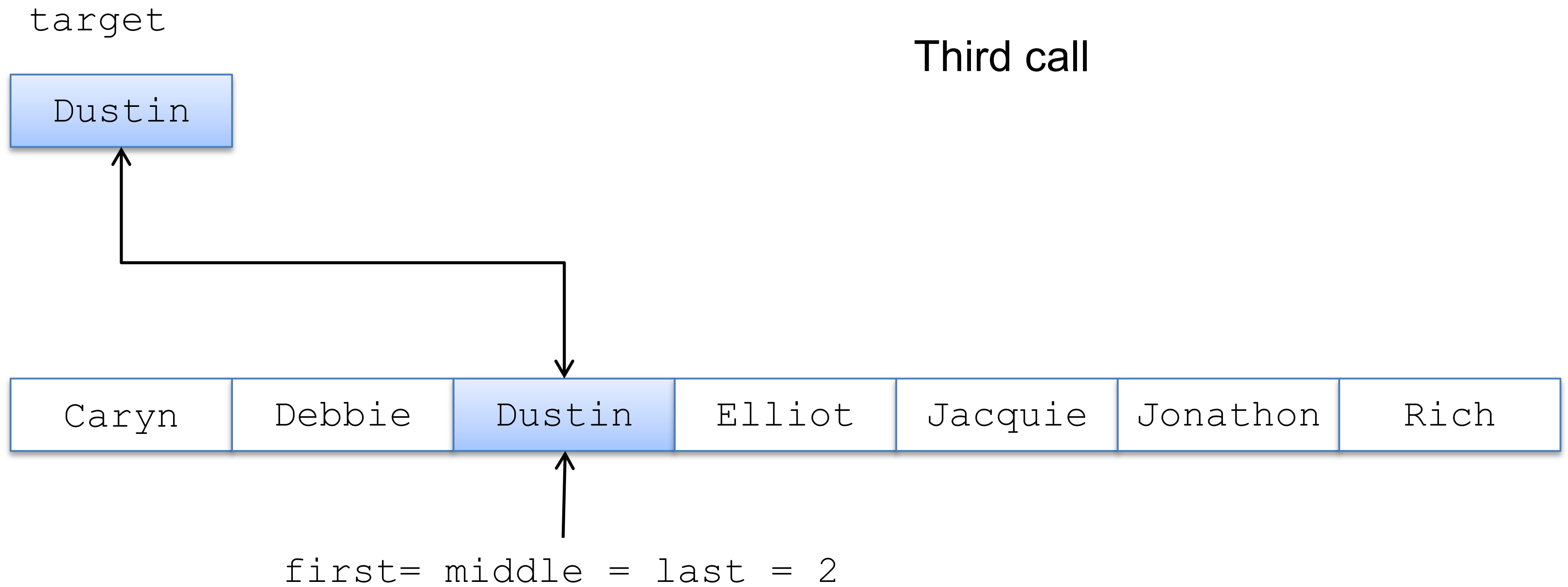
middle = 3

last = 6

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! ($\log_2 32768 = 15$)

Trace of Binary Search