Recursion

Learning Objectives

- To understand how to think recursively
- To be able to write recursive functions
- To be able to trace a recursive function
- To be able to write recursive algorithms and functions for searching arrays

Recursive Thinking

> The journey of a thousand miles starts with one mile.
>
> And then a journey of 999 miles.

Recursive Thinking

A function is recursive if it invokes itself to do part of its work.

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.

FIGURE 7.1
A Set of Nested Wooden Figures

Recursion

An alternate to using loops for solving problems

- Some problems are easier to solve with recursion
- Some people just prefer using recursion!

The core of recursion is taking a big task and breaking it up into a series of related small tasks.

- Example: handing out papers for an exam
  - Iterative: have a TA walk down a row of students, giving each person an exam
  - Recursive: A student takes one exam, pass the rest down the aisle

Breaking up a large problem

We want to write a program that prints N stars on one line, but without loops.

```
public static void printStars(int N);
```

Here's

```
printStars(N) ---> printStars(1) + printStars(N - 1)
printStars(3) ---> printStars(1) + printStars(2)
printStars(2) ---> printStars(1) + printStars(1)
printStars(1) ---> System.out.print("*");
```

Anatomy of a Recursive Function

Every recursive function needs at least one **base case** and at least one **recursive part**.

The **base case:**

- handles a simple input that can be solved without resorting to a recursive call. Can also be thought of as the case where we "end“ our recursion.

The **recursive part:**

- contains one or more recursive calls to the function.
- In every recursive call, the parameters must be in some sense “closer” to the base case than those of the original call

Anatomy of a Recursive Function

Pretend that you never learned about `*` as an operator!

```
/**
 * The function takes two ints x and y and returns x * y
 * @param x the first operand
 * @param y the second operand
 * @return x * y
 */
public static int multiply(int x, int y) {
    if (x == 1) { // base case (multiplying by 1 is easy)
        return y; // 1 * y == y, so just return y.
    } else {
        return multiply(x - 1, y) + y; // recursive call!
    }

}
```

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending
`multiply(2, 10)`	`2`	`10`	`10 + multiply(1, 10)`	pending

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending
`multiply(2, 10)`	`2`	`10`	`10 + multiply(1, 10)`	pending
`multiply(1, 10)`	`1`	`10`	`10`	complete (base case triggered)

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + multiply(2, 10)`	pending
`multiply(2, 10)`	`2`	`10`	`10 + 10` → `20`	complete
`multiply(1, 10)`	`1`	`10`	`10` (base case triggered!)	complete (base case triggered)

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + multiply(3, 10)`	pending
`multiply(3, 10)`	`3`	`10`	`10 + 20` → `30`	complete
`multiply(2, 10)`	`2`	`10`	`10 + 10` → `20`	complete
`multiply(1, 10)`	`1`	`10`	`10` (base case triggered!)	complete (base case triggered)

Tracing a Returning Recursive Function

Tracing `multiply(4, 10);`

call	x	y	return	call status
`multiply(4, 10)`	`4`	`10`	`10 + 30` → `40`	complete
`multiply(3, 10)`	`3`	`10`	`10 + 20` → `30`	complete
`multiply(2, 10)`	`2`	`10`	`10 + 10` → `20`	complete
`multiply(1, 10)`	`1`	`10`	`10` (base case triggered!)	complete (base case triggered)

So `multiply(4, 10)` evaluates to `40`.

Steps to Design a Recursive Algorithm

Identify the base case(s) and solve it/them directly

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

Design a Recursive Solution to a Problem

```
public static boolean isPalindrome(String s) {...}
```

Given a String, decide if it is a *palindrome*, that is, if it reads the same forwards and backwards.

- **Base case?**
- **Recursive steps?**

Design a Recursive Solution to a Problem

```
public static boolean isPalindrome(String s) {...}
```

Given a String, decide if it is a *palindrome*, that is, if it reads the same forwards and backwards.

- **Base case?** Strings of length 0 or 1 are palindromes always! (`""` and `"a"`, for example))
- **Recursive steps?**

Design a Recursive Solution to a Problem

```
public static boolean isPalindrome(String s) {...}
```

Given a String, decide if it is a *palindrome*, that is, if it reads the same forwards and backwards.

- **Base case?** Strings of length 0 or 1 are palindromes always! (`""` and `"a"`, for example))
- **Recursive steps?** For any String, it can only be a palindrome if its first and last letters are the same.
    - If they are, then we still need to check the rest of the String
    - If they are not the same, then we can stop immediately.

Design a Recursive Solution to a Problem

```
public static boolean isPalindrome(String s) {
    // base case
    if (???) {
        return ???;
    } else { // recursive case
        return ???;
    }
}
```

Design a Recursive Solution to a Problem

What's the condition for the base case?

```
public static boolean isPalindrome(String s) {
    // base case
    if (s.length() <= 1) {
        return ???;
    } else { // recursive case
        return ???;
    }
}
```

Design a Recursive Solution to a Problem

What do we return for a String that's empty or 1 character long?

```
public static boolean isPalindrome(String s) {
    // base case
    if (s.length() <= 1) {
        return true
    } else { // recursive case
        return ???;
    }
}
```

Design a Recursive Solution to a Problem

How do we check if the first and last characters are the same?

```
public static boolean isPalindrome(String s) {
    // base case
    if (s.length() <= 1) {
        return true
    } else { // recursive case
        boolean firstMatchesLast = s.charAt(0) == s.charAt(s.length() - 1);
        return ???;
    }
}
```

Design a Recursive Solution to a Problem

How do we use this boolean value?

```
public static boolean isPalindrome(String s) {
    // base case
    if (s.length() <= 1) {
        return true
    } else { // recursive case
        boolean firstMatchesLast = s.charAt(0) == s.charAt(s.length() - 1);
        if (firstMatchesLast) {
            return isPalindrome(s.substring(1, s.length() - 1));
        } else {
            return false;
        }
    }
}
```

Design a Recursive Solution to a Problem

Slightly more succinctly...

```
public static boolean isPalindrome(String s) {
    // base case
    if (s.length() <= 1) {
        return true
    } else { // recursive case
        boolean firstMatchesLast = s.charAt(0) == s.charAt(s.length() - 1);
        String theRest = s.substring(1, s.length() - 1);
        return firstMatchesLast && isPalindrome(theRest);
    }
}
```

Recursion & Nodes

The Node class is defined in terms of itself! Nodes are considered *recursive* data structures, and so recursion is a natural way to process sequences of Nodes.

```
public class Node {
    public int data;
    public Node next;

    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```

Recursion & Nodes

Problem: Given a Node reference, how long is the chain of Nodes rooted at this reference?

Base Case?

Recursive Steps?

Recursion & Nodes

Problem: Given a Node reference, how long is the chain of Nodes rooted at this reference?

Base Case? If a Node reference is null, the chain is empty, so the length is `0`.

Recursive Steps?

Recursion & Nodes

Problem: Given a Node reference, how long is the chain of Nodes rooted at this reference?

Base Case? If a Node reference is null, the chain is empty, so the length is `0`.

Recursive Steps? If a Node reference is not null, then it points to some Node!

- The length of the chain rooted at this Node will be 1 plus the length of the chain rooted at the Node it points to.

Recursion & Nodes

```
public static int length(Node n) {
    // base case
    if (???) {
        return ???;
    } else { // recursive case
        return ???;
    }
}
```

Recursion & Nodes

```
public static int length(Node n) {
    if (n == null) {
        return 0;
    } else {
        return 1 + length(n.next);
    }
}
```

Recursion & Nodes

(The iterative solution, if you were curious…)

```
public static int length(Node n) {
    int count = 0;
    Node curr = n;
    while (curr != null) {
        count++;
        curr = curr.next;
    }
}
```

Recursion & Nodes

Problem: Given a Node reference, what's the sum of values stored in the chain of Nodes rooted here?

Base Case?

Recursive Steps?

Recursion & Nodes

Problem: Given a Node reference, what's the sum of values stored in the chain of Nodes rooted here?

Base Case? If a Node reference is null, the chain is empty, so the sum of values here is `0`.

Recursive Steps?

Recursion & Nodes

Problem: Given a Node reference, what's the sum of values stored in the chain of Nodes rooted here?

Base Case? If a Node reference is null, the chain is empty, so the sum of values here is `0`.

Recursive Steps? If a Node reference is not null, then it points to some Node!

- The sum of the data in the chain rooted at this Node will be the data in the current node plus the sum of the data in the rest of the chain.

Recursion & Nodes

```
public static int sumOfData(Node n) {
    // base case
    if (???) {
        return ???;
    } else { // recursive case
        return ???;
    }
}
```

Recursion & Nodes

```
public static int sumOfData(Node n) {
    if (n == null) {
        return 0;
    } else {
        return n.data + sumOfData(n.next);
    }
}
```

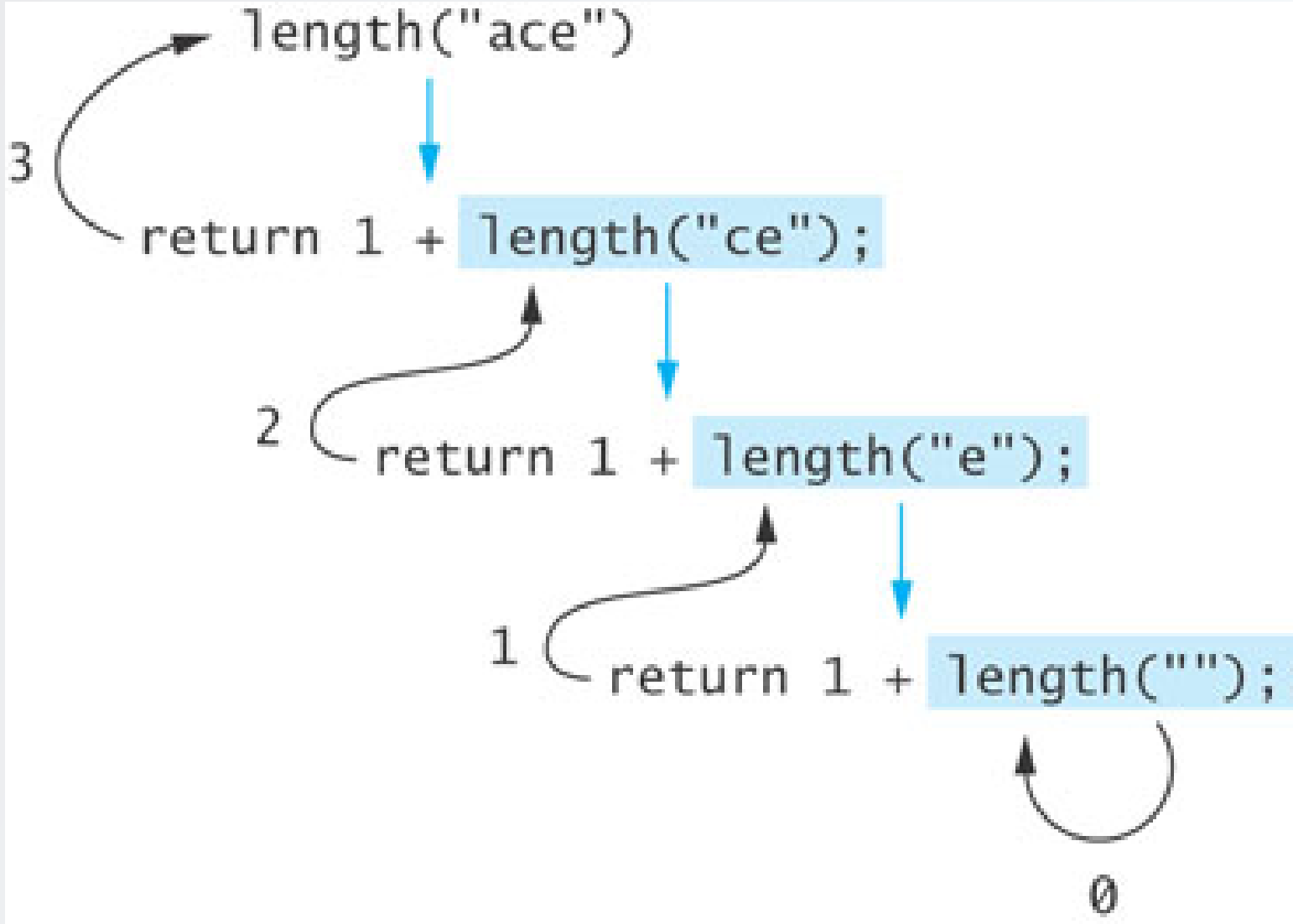
Recursion & Nodes

(The iterative solution, if you were curious…)

```
public static int length(Node n) {
    int sum = 0;
    Node curr = n;
    while (curr != null) {
        sum += curr.data;
        curr = curr.next;
    }
}
```

Tracing a Recursive Function

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

Details not at all important for this course; much more on this in CIS 2400.

Recursive Array Search

Recursion & Arrays

Recall that we want to make the problem *smaller*, but it's not actually possible to change the size of an array. How can we recurse on an array?

```
public int findLargest(int[] arr)
```

Recursion & Arrays

Recall that we want to make the problem *smaller*, but it's not actually possible to change the size of an array. How can we recurse on an array?

- Solution: use an index (or indices) to specify the portion of the array that you're recursing over.

```
public int findLargest(int[] arr)
```

becomes

```
public int findLargest(int[] arr, int currIdx)
```

Recursion & Arrays

```
public int findLargest(int[] arr, int currIdx)
```

Given an array of ints, and a current index for searching at, return the largest int in the array **at or after the current index.**

- **Base case?**
  - Empty arrays have no largest element, so return `Integer.MIN_VALUE`
- **Recursive steps?**
  - Return whichever is greater: the element at the current position, or the greatest element in the rest of the array.

Recursion & Arrays

```
public int findLargest(int[] arr, int currIdx) {
    // base case(s)
    if (???) {
        return ???;
    }
    // recursive case
    return ???;
}
```

Recursion & Arrays

```
public int findLargest(int[] arr, int currIdx) {
    // base case(s)
    if (arr == null || currIdx >= arr.length) {
        return Integer.MIN_VALUE;
    }
    // recursive case
    return ???;
}
```

Recursion & Arrays

```
public int findLargest(int[] arr, int currIdx) {
    // base case 1
    if (arr == null || currIdx >= arr.length) {
        return Integer.MIN_VALUE;
    }
    // recursive case
    return ???;
}
```

Recursion & Arrays

```
public int findLargest(int[] arr, int currIdx) {
    // base case 1
    if (arr == null || currIdx >= arr.length) {
        return Integer.MIN_VALUE;
    }
    int biggestInRest = findLargest(arr, currIdx + 1);
    return Math.max(arr[currIdx], biggestInRest);
}
```

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

How many recursive calls are needed?

- Each recursive call examines one element
- On average, $\frac{(n+1)}{2}$ elements are examined to find the target in a linear search
- If the target is not in the list, n elements are examined

Recursive Array Search (cont.)

- **Base cases?**
  - Empty array, target can not be found; result is -1
  - First element of the portion of the array being searched is target; result is the index of first element
- **Recursive steps?**
  - 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

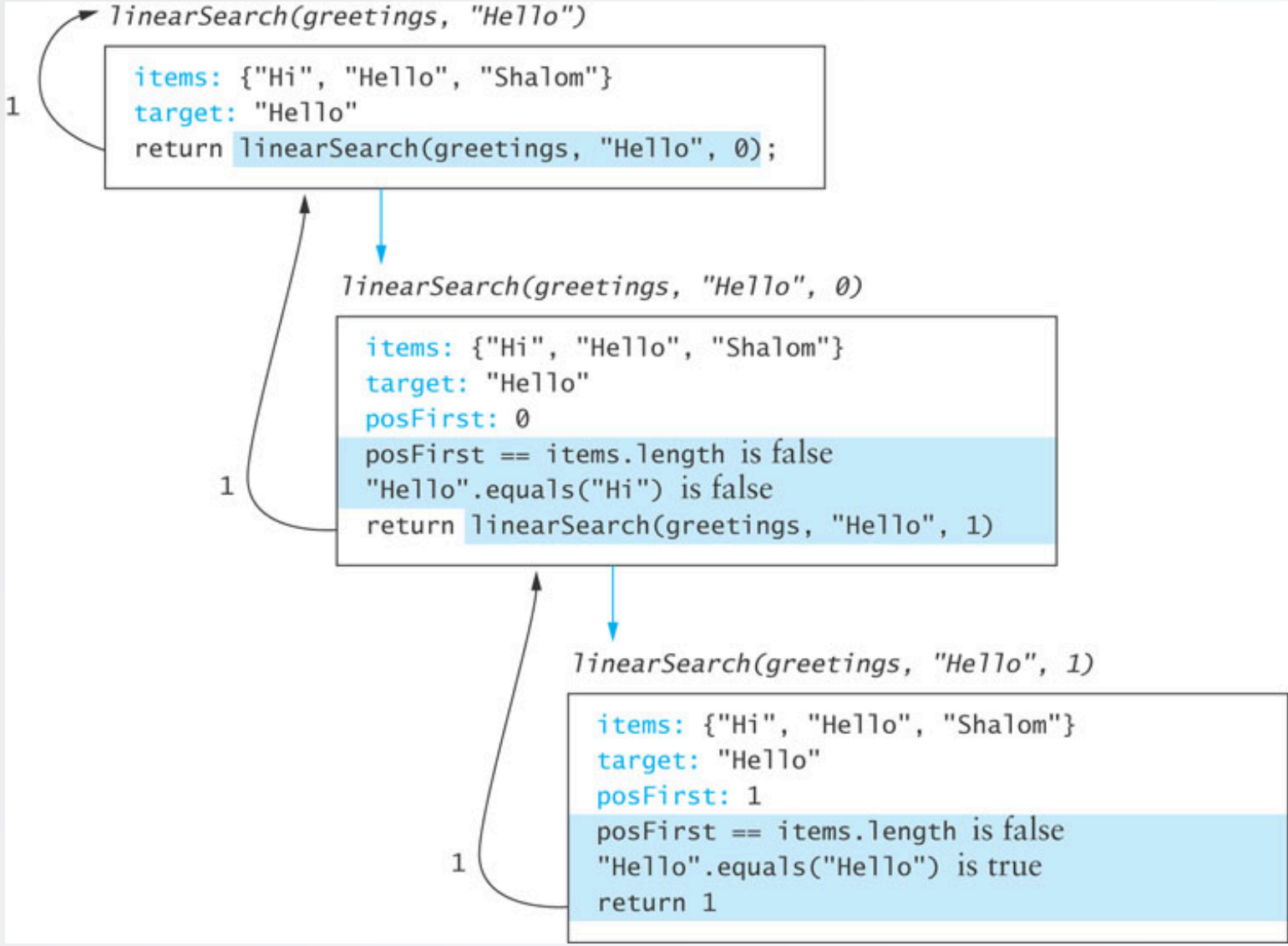
if the array is empty, the result is –1

else if the first element of the feasible area matches the target, the result is the position of the first element

else, search the array excluding the first element and return the result

```
public static void int linearSearch(String[] items, String target,
                                    int idx) {
    if (idx >= items.length) {
        return -1;
    } else if (items[idx].equals(target)) {
        return idx;
    } else {
        return linearSearch(items, target, idx + 1);
    }
}
```

Implementation of Recursive Linear Search



Design of a Binary Search Algorithm

A binary search can be performed only on an array that has been sorted. Remember: rather than looking at the first element, a binary search compares the **middle element** for a match with the target

- **Base cases?**
  - The array is empty
  - Element being examined matches the target
- **Recursive steps?**
  - 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

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

```
public static void int binarySearch(String[] items, String target,
                                    int left, int right) {
    if (left > right || items.length == 0) {
        return -1;
    }
    int middle = (left + right) / 2;
    if (items[middle].equals(target)) {
        return middle;
    } else if (target.compareTo(items[middle]) < 0) {
        return binarySearch(items, target, left, middle - 1);
    } else {
        return binarySearch(items, target, middle + 1, right);
    }

}
```