Advanced Programming Techniques in Java

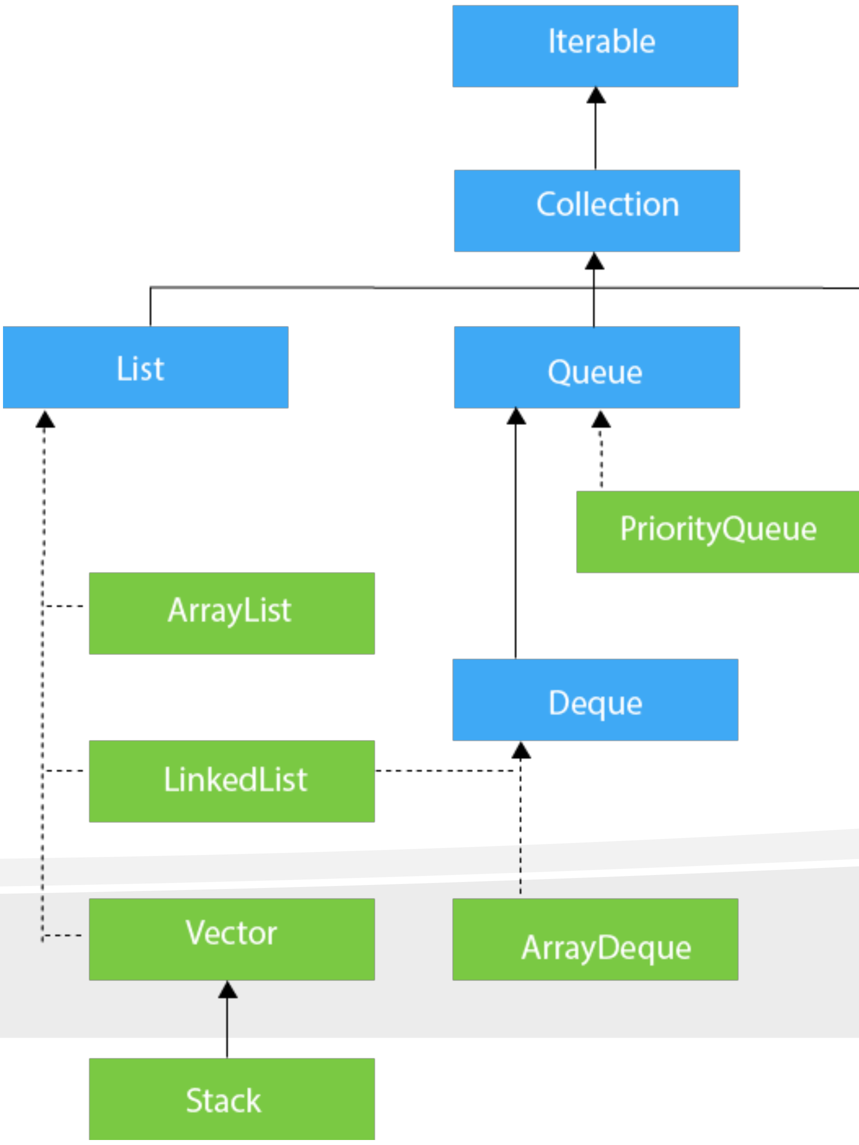
Recursion

Lecture 20

Class Objectives

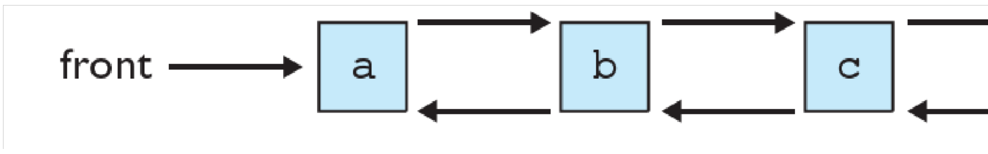
- Recursion (Sections 12.1-12.3)

Review: Collections Framework D


Iterable
Collection
List
Queue
PriorityQueue
ArrayList
Deque
LinkedList
Vector
ArrayDeque
Stack

Review: Linked list

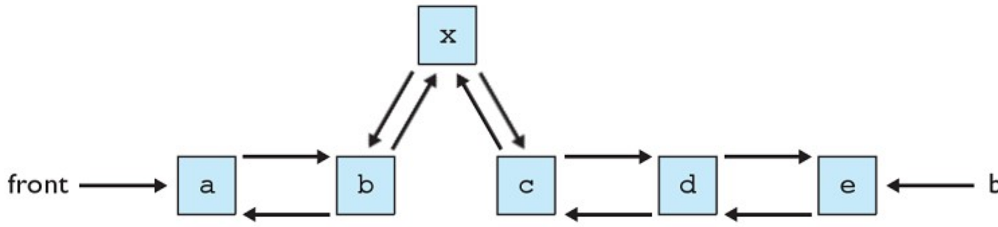
- **Linked list** is a list implemented using a linked s
- Each value is stored in a small object called a **node**, v
neighbor nodes
- The list keeps a reference to the first and/or last node

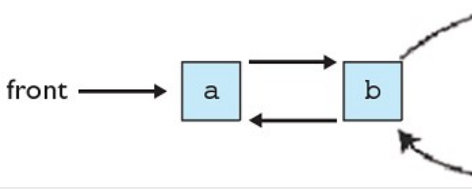


- In Java, represented by the class LinkedList

Review: Linked list performance

- To add, remove, get a value at a given index:
- The list must advance through the list to the node ju
- For example to add a new value to the list, the list cr existing node links to the proper index, and attaches and follow it





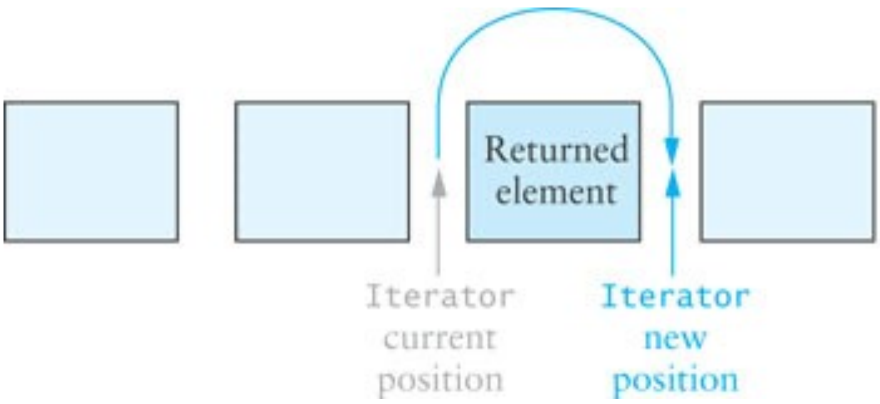
- This is very fast when adding to the front or back of
references to these places), but slow elsewhere

Review: Linked List Implementatio

```
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;
            }
  }
}
```

Review: Iterator Position

- An `Iterator` is conceptually *be*
  it does notrefer to a parti
  given time


Returned element
Iterator current position
Iterator new position

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 g

- A unified way to examine all elements of a collecti
- Every collection in Java has an `iterator` method
- In fact, that's the *only* guaranteed way to examine the e

- Don't have to use indexes

Review: The `ListIterator<E`

- Extends the `Iterator` interface
- The `LinkedList` class implements the `List<E>` inte
- Methods in `LinkedList` that return a `ListIterator`
- `public ListIterator<E> listIterator()` ▪ `public` `index)`
- Methods in the `ListIterator` interface:

- `add, hasNext, hasPrevious, next, previous`
  `remove, set`

Abstract Data Types (ADTs)

- **Abstract data type (ADT)** is a general specific
- Specifies what data the data structure can hold
- Specifies what operations can be performed on the
- Does NOT know how the data structure hold the dat
  operation
- Example ADT: **List**

- Specifies that a list collection will store elements in c
and null values)
- Specifies that a list collection supports `add`, `remove`
`isEmpty`, …
- …

Abstract Data Types (ADTs)

- `ArrayList` and `LinkedList` both implement th
ADT

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

More on ADTs

- Good practice is to use the appropriate interface ty
- `List<Integer> list = new LinkedList<Integ`
- Gives flexibility to change implementations of the list

- You can use the interface type `List` when declar

Strengths

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

- `LinkedList`
- Sequential access, `get/remove/add` fast on
- Adding and removing at either end of the list is
- 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 ope
- Make sure you are not adding an element that is already

ArrayList vs. LinkedList

- Both implements List interface and maintains ins

ArrayList	LinkedList
Uses a **dynamic array** to store the elements	Uses a **doubly link** 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 L ArrayList because it so no shifting is req
An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can both because it imp interfaces.

ArrayList is **better for storing and accessing** data.	LinkedList is **better**

RecursiveThinkin

Recursion

- **recursion**: The definition of an operati terms of itself.
  - Solving a problem using recursion dep smaller occurrences of the same problem

- **recursive programming**: Writing methods themselves to solve problems recursively.
  - An equally powerful substitute for *iteration*
  - Particularly well-suited to solving certain problems

Why learn recursion?

- "cultural experience" - A different way of thi
  problems

- Can solve some kinds of problems bet
  iteration
- Leads to elegant, simplistic, short code
  used well)
- Many programming languages ("functiona
  languages such as Scheme, ML, OCaml
  Haskell) use recursion exclusively (no

Recursive Thinking

- Consider searching for a target va
  - Assume the array elements ar
    increasing order
  - We compare the target to the m
    if the middle element does no
    search either the elements be
    element or the elements after
  - Instead of searching n elements, w
    elements

Recursive Thinking (co

Recursive Algorithm to Search an Array **`if`** the array is empty

return -1 as the search result

`else if` the middle element matches th

return the subscript of the middl

result **`else if`** the target is less than th

recursively search the array eleme

middle element and return the result

```
else
```

recursively search the array eleme
middle element and return the result

Steps to Design a Recu

- There must be at least one case for a small value of n, that can be
- A problem of a given size n can or more smaller versions of the same (recursive case(s))

- Identify the base case and provide a
- Devise a strategy to reduce the proble versions of itself while making progre case
- Combine the solutions to the smaller pr the larger problem

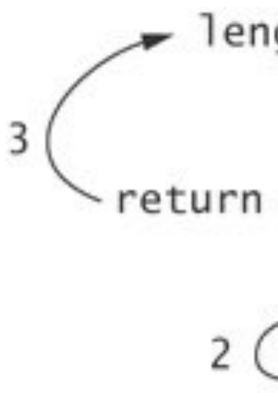
Proving that a Recursive M Correct

- Proof by induction
  - Prove the theorem is true for the base case
  - Show that if the theorem is assumed true fo true for n+1
- Recursive proof is similar to induction
  - Verify the base case is recognized and so
  - Verify that each recursive case makes pr case

- Verify that if all smaller problems ar
  thenthe original problem also is solved co

Tracing a Recursive Me

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

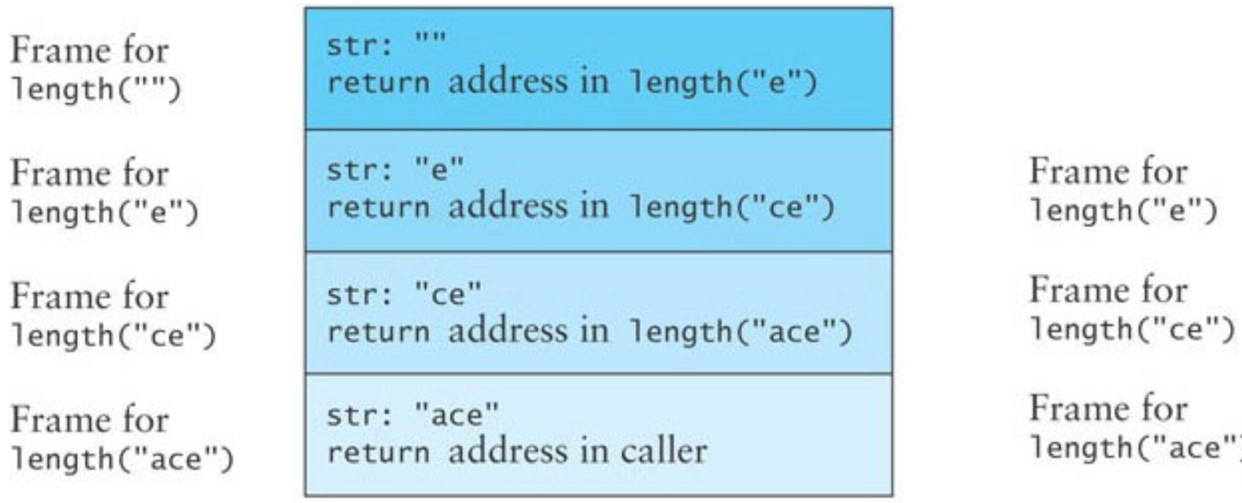


Run-Time Stack and Activa

- Java maintains a run-time stack on
  newinformation in the form of an *ac*
- The activation frame contains storag
  - method arguments
  - local variables (if any)
  - the return address of the instruction

- Whenever a newmethod is called (r
  Javapushes a newactivation frame or
  stack

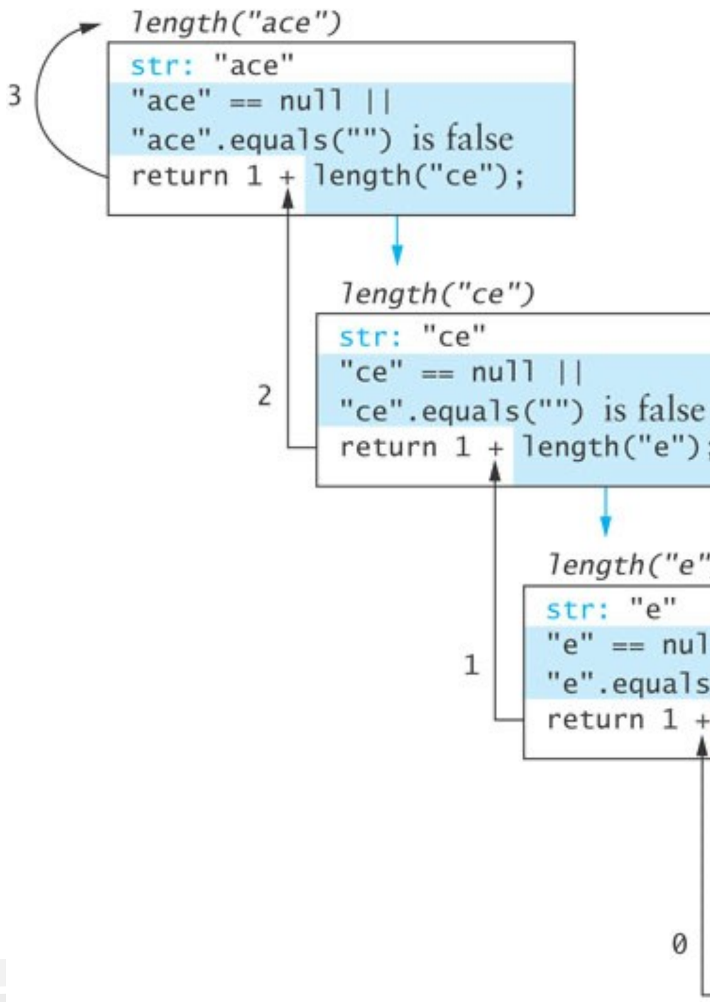
Run-Time Stack and Act
Frames (cont.)



Run-time stack after all calls

Run-ti

Run-Time Stack and Acti
Frames


length("ace")
str: "ace"
"ace" == null ||
"ace".equals("") is false
return 1 + length("ce");
3
length("ce")
str: "ce"
"ce" == null ||
"ce".equals("") is false
return 1 + length("e");
2
str: "e"
1
0

Recursive Definitions of Mat
Formulas

Recursive Definitions of M Formulas

- Mathematicians often userecursive formulas that lead naturally to algorithms
- Examples include:
  - factorials
  - powers
  - greatest common divisors (gcd)

Factorial of *n*: *n*!

- The factorial of n, or $n!$ is de

$$0! = 1$$

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

- The base case: n equal to 0
- The second formula is a rec

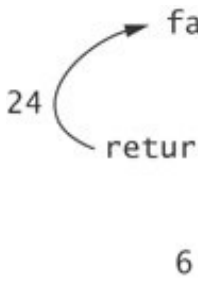
Factorial of n: $n!$ (cont.)

- The recursive definition can be expressed algorithm:

  if n equals 0

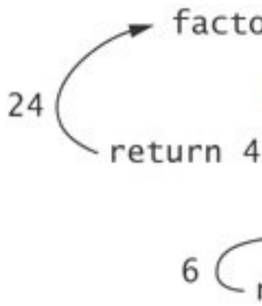
$n!$ is 1 **else**

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



- The last stepcan 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 St

- If you call method `factorial` with a
  the recursion will not terminate bec
  equal `0`
- If a program does not terminate,
  throw the
  `StackOverflowError` exception
- Make sure your recursive method
  thata stopping case is always
- In the `factorial` method, you could

`IllegalArgumentException` if `n` is

Recursive Algorithm for

- The greatest common divisor (gcd) of the largest integer that divides both nu
- 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 (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 (cont.)

```
/** Recursive gcd method (in RecursiveM
```

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

RecursionVersus Iterat

- Thereare similarities between recursion
- In iteration, a loop repetition cor whetherto repeat the loop body or
- In recursion, the condition usually tes case
- **You canalways write an iterative sol problem that is solvable by rec**

- A recursive algorithm may be simpler algorithm and thus easier to write, read

Iterative `factorial` **M**

```
/** Iterative factorial method.
    pre: n >= 0
    @param n The integer whose factorial i
    @return n!
*/
```

```
public static int factorialIter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
        result = result * k;
    return result;
}
```

Efficiency ofRecursion

- Recursive methods often have slower relative to their iterativecounterparts

- The overhead for loop repetition is overhead for a method call and return
- If it is easier to conceptualize an alg recursion, then you should code it method
- The reduction in efficiency does not advantage of readable code thatis

Fibonacci Numbers

- The Fibonacci numbers are a seque
  follows

  $fib_1 = 1$ $fib_2 = 1$

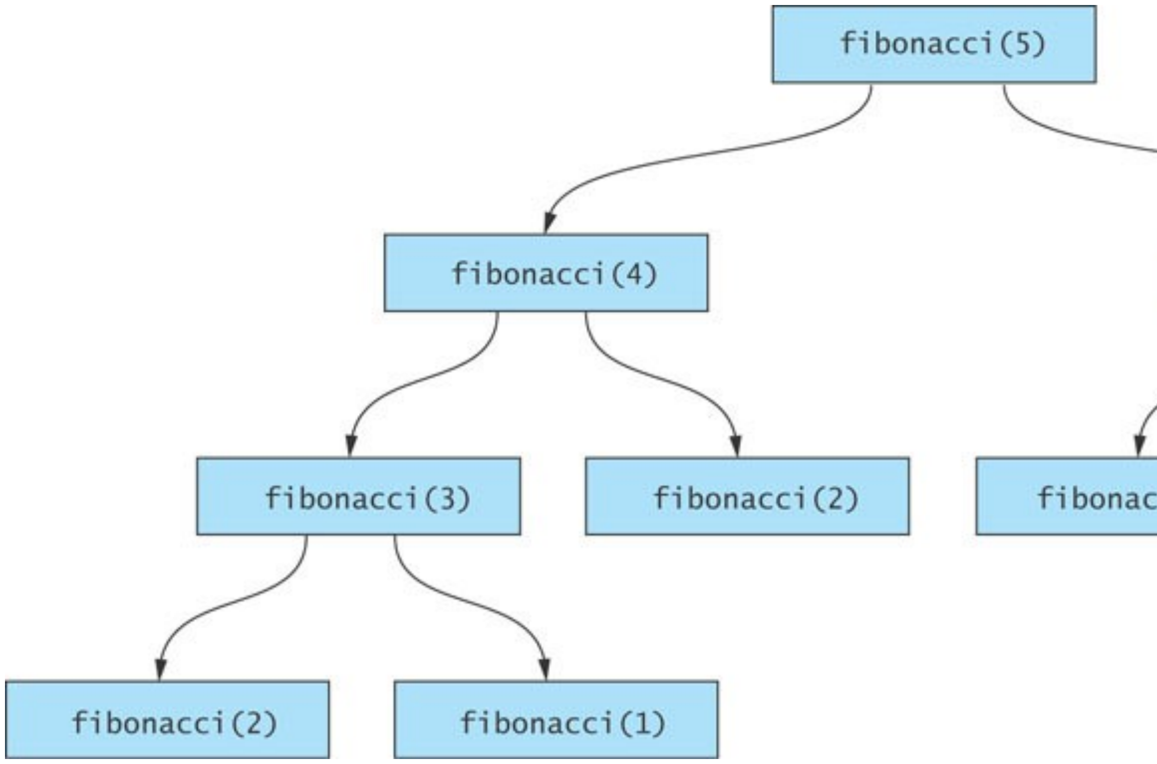
  $fib_n = fib_{n-1} + fib_{n-2}$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 14

AnExponential Recursive Method

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

Efficiency of Recursion: Ex
`fibonacci`



fibonacci(5)
fibonacci(4)
fibonacci(3)
fibonacci(2)
fibonac
fibonacci(2)
fibonacci(1)

An O(n)Recursive `fibon`

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

An O(n)Recursive fibon

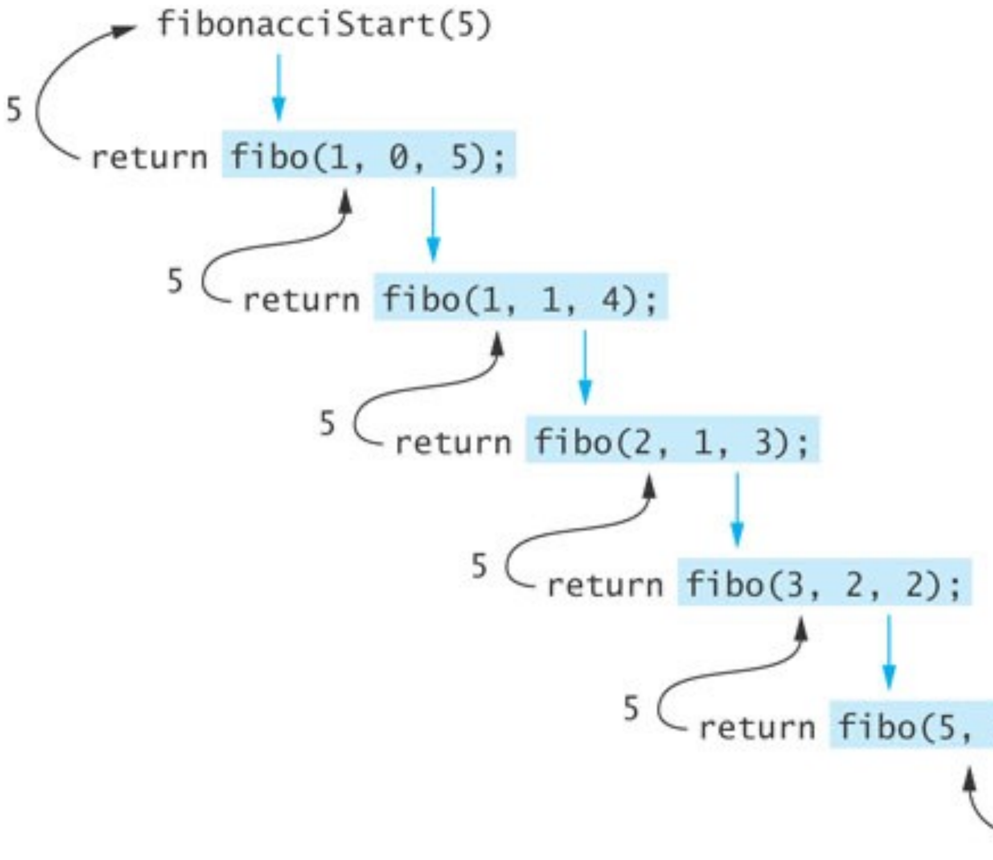
(cont.)

- In order to start the method executi non-recursive wrappermethod:

```
/** Wrapper method for calculating Fibonacci numb
RecursiveMethods.java).

    pre: n >= 1

    @param n The position of the desired Fibonacc

    @return The value of the nth Fibonacci number

*/
```

```
public static int fibonacciStart(int n) {
     return fibo(1, 0, n);
}
```

Efficiency of Recursion: O


fibonacciStart(5)
5
return fibo(1, 0, 5);
5
return fibo(1, 1, 4);
5
return fibo(2, 1, 3);
5
return fibo(3, 2, 2);
5
return fibo(5,

Efficiency of Recursion: O

- Method `fibo` is an example of *tail rec*
- When recursive call is the last line of
  local variables do not need to be
  frame

RecursiveArray

Recursive Array Sea

- Searching an array can be accomplish
- Simplest way to search is a linear
  - Examine one element at a timestar and ending with the last
  - On average, n /2 elementsare exa target in a linear search
  - If the target is not in the list, n el

- A linear search is O(n)

Recursive Array Searc

- Base cases for recursive search:
  - Empty array, target can not be fou
  - First element of the array being
    result is the subscript of first ele

- The recursive stepsearches the res
  excluding the first element

Algorithm for Recursi
Search

Algorithm for Recursive Linear Array
Search `if` the array is empty th
result is −1

```
else if the first element matches the
    target the result is the subscript
    the first element
else
    search the array excluding the firs
  the result
```

Design ofa Binary Sea

- A binary search can be performed
  thathas been sorted
- Basecases
  - The array is empty
  - Element being examined matches
- Rather than looking at the firstele
  search compares the middle elemen
  the target
- A binary search excludes the hal
  which the target cannot lie

Design ofa Binary Sea
(cont.)

Binary Search Algorithm

`if` the array is empty
return –1 as the
search result
`else if` the middle element matches
return the subscript of the mi
the result

`else if` the target is less than the

recursively search the array ele

middle element and return the res

`else`

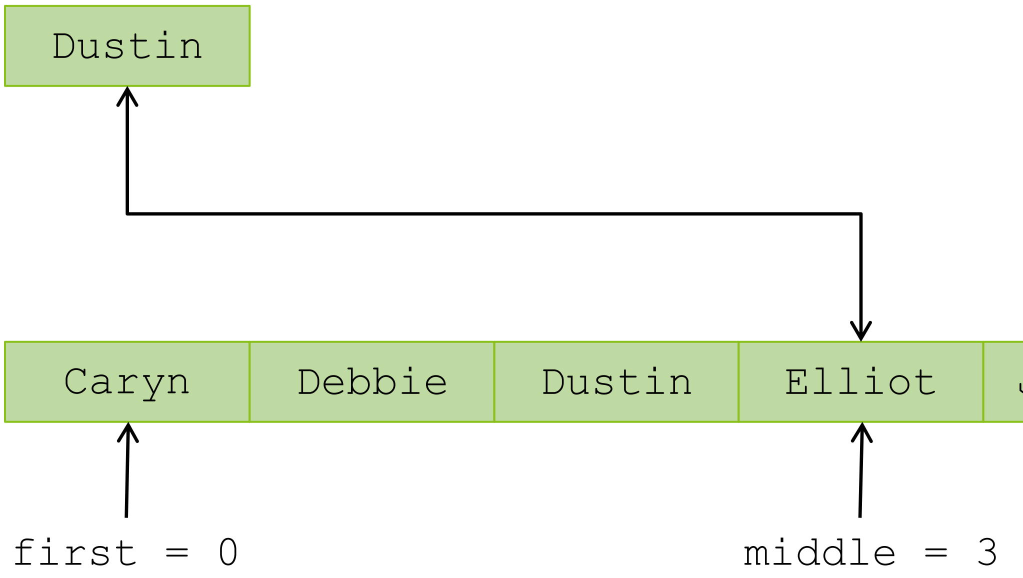
recursively search the array ele

element and return the result

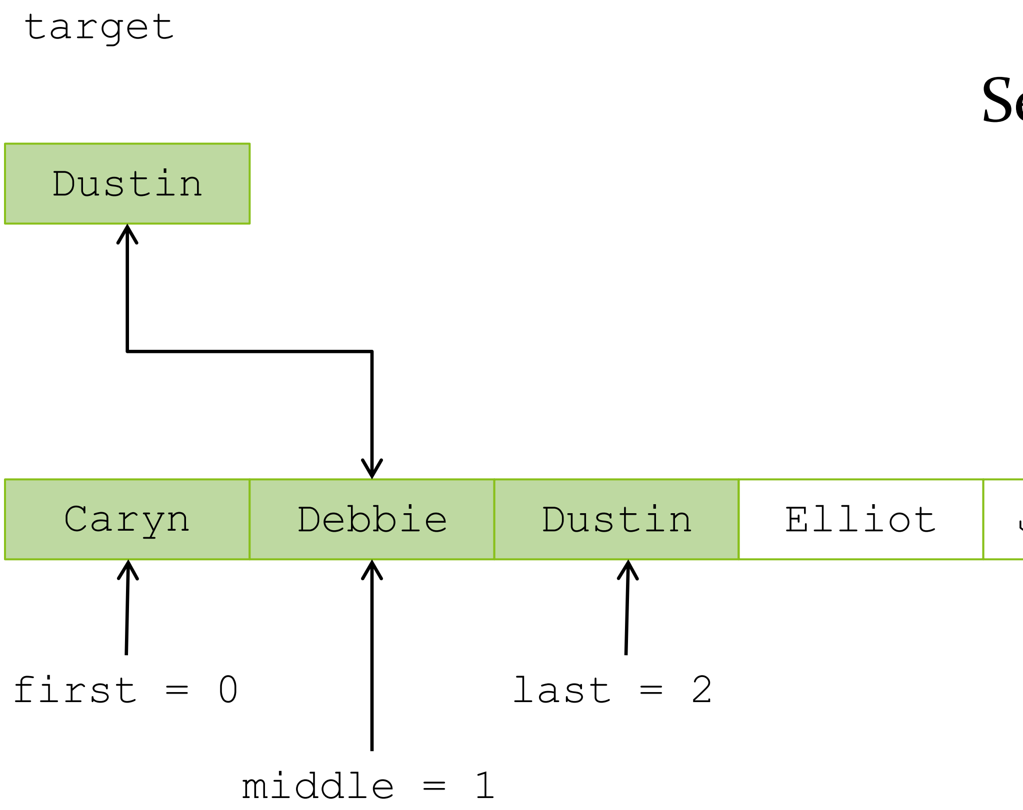
Binary Search Algori

`target`

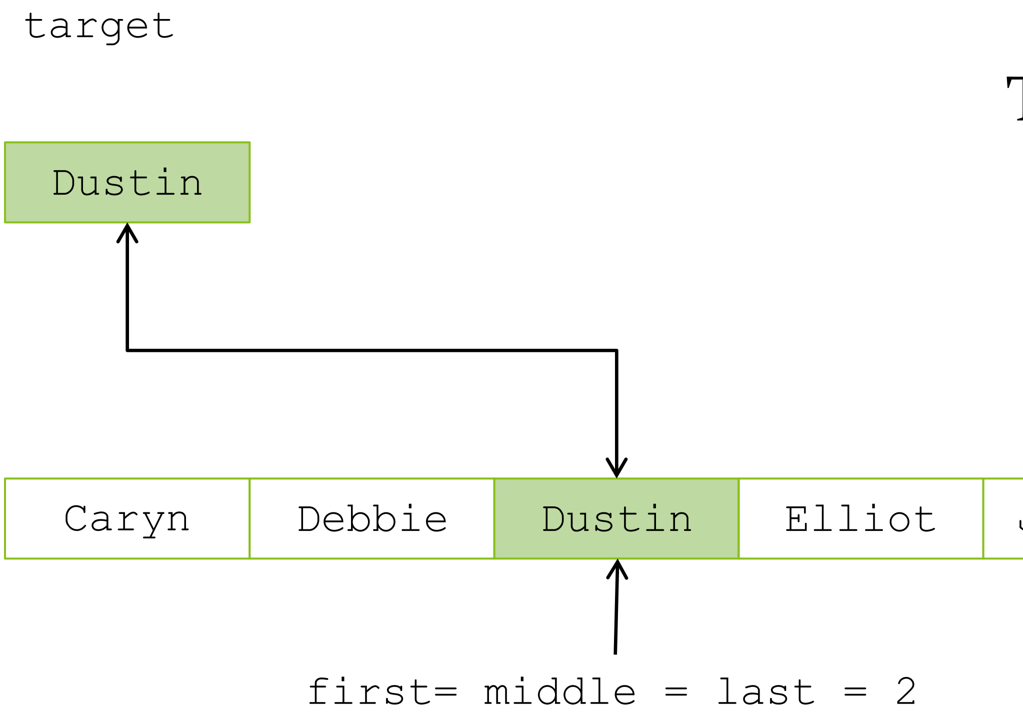
F



Binary Search Algorith



Binary Search Algorith



Efficiency ofBinary

- At each recursive call we elimina
  elements from consideration,
  search O(log n)
- An array of 16 would search arr
  2, and 1; 5 probes in the wo
  - 16 = 2^4
  - 5= $\log_2 16$ + 1
- A doubledarray size would only
  worst case
  - 32 = 2^5

- $6 = \log_2 32 + 1$
- An array with 32,768 elements ($\log_2 32768 = 15$)

Implementation of a Bina
Algorithm

```
/** Recursive binary search method (in Recur
    @param items The array being searched
    @param target The object being searched
    @param first The subscript of the first
    @param last The subscript of the last el
    @return The subscript of target if found
*/
private static int binarySearch(Object[] ite
                                int first, i
    if (first > last)
        return -1;     // Base case for unsu
    else {
        int middle = (first + last) / 2;  //
        int compResult = target.compareTo(it
        if (compResult == 0)
            return middle;   // Base case fo
        else if (compResult < 0)
            return binarySearch(items, targe
        else
            return binarySearch(items, targe
    }
}
```

Implementation of a Bina
Algorithm (cont.)

```
/** Wrapper for recursive binary search method
    @param items The array being searched
    @param target The object being searched fo
    @return The subscript of target if found;
*/
public static int binarySearch(Object[] items,
    return binarySearch(items, target, 0, item
}
```

Testing Binary Sear

- You should test arrays with
  - an even number of elements
  - an odd number of elements
  - duplicate elements
- Test each array for the followi
  - the target is the element at eac
    array, starting with the first position
    position
  - the target is less than the smallest
  - the target is greater than the larg

- the target is a value between
  in the array