CS18000: Problem Solving and Object-Oriented Programming

Recursion

Video 1
What is Recursion?

Recursion and Recursive Data Structures

Recursion and Stacks

What is Recursion?

- A self reference
- Methods:
  - A method can call itself
  - Example: Fibonacci method
- Data:
  - A data structure can reference itself
  - Example: LinkedList Node class

```
private class Node {
      String value;
      Node link;
 }
```

Recursive Problem Solving

- Sometimes…
  - Easier to partially solve a problem
  - Delegate the rest to someone else
- "Want me to compute Fibonacci(n)?"
- "OK..."
  - If n == 0, then "The answer is: 0" (easy!)
  - If n == 1, then "The answer is: 1" (easy!)
  - else (get help!)
    - "Alice: What is Fibonacci(n-1)?"
    - "Bob: What is Fibonacci(n-2)?"
    - "The answer is: " Alice's answer + Bob's answer

Why Recursion Works

- The method does not *always* call itself
- The data structure does not *always* link to another copy of itself
- There's always a "basis case" (or base case)

- Recursion works well for problems that can be split in this way: a basis case and a recursive case

Recursive Definitions

- Fibonacci(n)
  - If n == 0, then 0
  - If n == 1, then 1
  - else Fibonacci(n-1) + Fibonacci(n-2)
- Factorial(n)
  - If n == 0, then 1
  - else n * Factorial(n-1)
- 2^n
  - If n == 0, then 1
  - If n == 1, then 2
  - If n is even, then $2^{n/2} * 2^{n/2}$
  - If n is odd, then $2 * 2^{(n-1)/2} * 2^{(n-1)/2}$

Key Task When Programming Recursion

- Break the problem down into two pieces:
  - Basis case: what can be done without a recursive call
  - Recursive case: the same problem but "smaller"
- The parameter(s) to the recursive case must be "smaller" in some sense: closer to the basis case

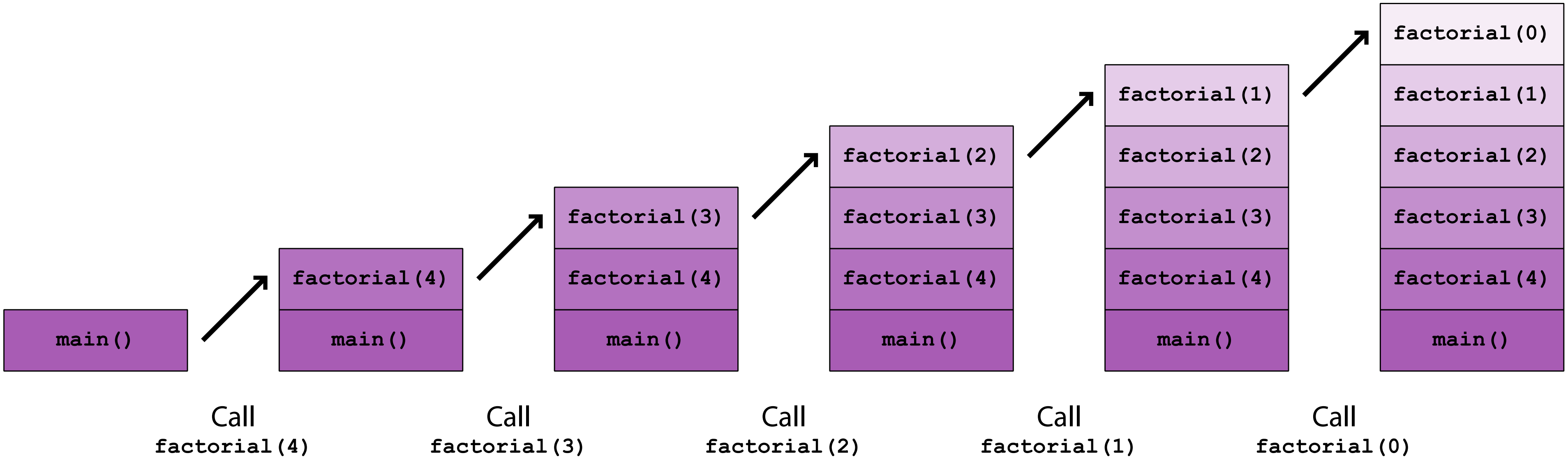
Video 2
Recursion Examples

How Recursion is Implemented

- Recall that…
  - A stack is used to handle method calls
  - When a method is called, parameters and local variables are “pushed” onto the “call stack”
- Each recursive method call has its own copy of parameters and local variables
- When a method returns, the previously executing method (“below it” on the stack) picks up where it left off

Example: Factorial

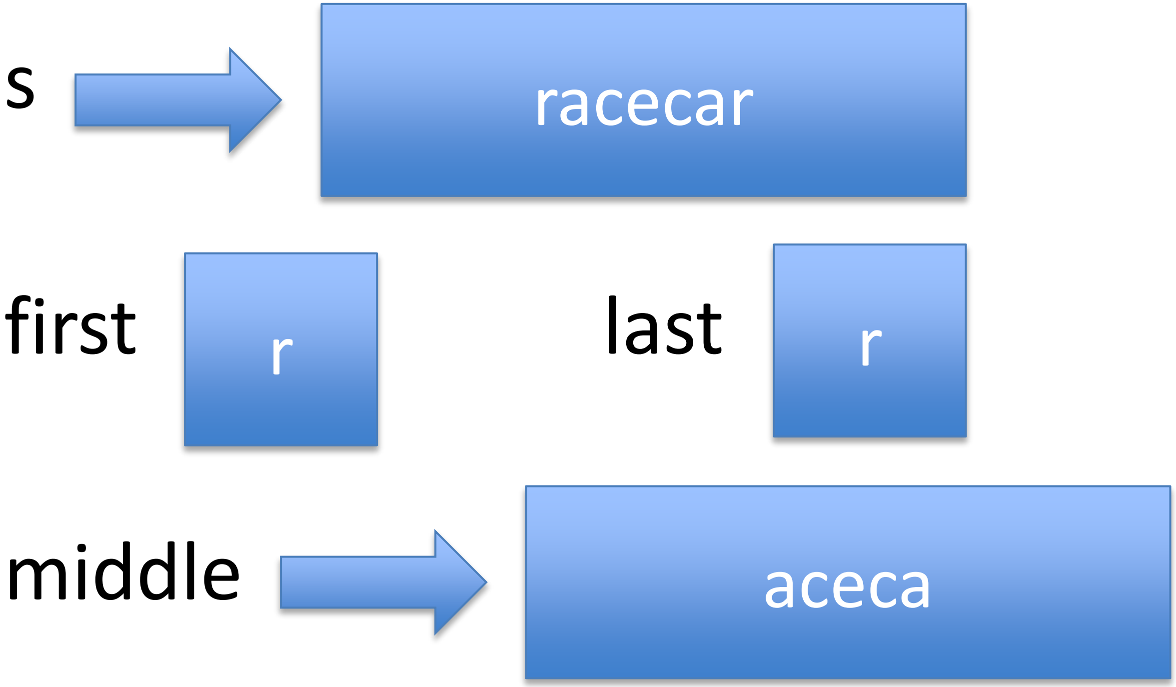
```
public class Factorial {
    public static long factorial(long n) {
        if (n == 0)
            return 1;
        else
            return n * factorial(n-1);
    }

    public static void main(String[] args) {
        for (int n = 0; n <= 20; n++)
            System.out.printf("%3d! = %d\n", n, factorial(n));
    }
}
```



main()
factorial(4)
main()
factorial(3)
factorial(4)
main()
factorial(2)
factorial(3)
factorial(4)
main()
factorial(1)
factorial(2)
factorial(3)
factorial(4)
main()
factorial(0)
factorial(1)
factorial(2)
factorial(3)
factorial(4)
main()
Call
factorial(4)
Call
factorial(3)
Call
factorial(2)
Call
factorial(1)
Call
factorial(0)

Example: isPalindrome

```
public static boolean isPalindrome(String s) {
    if (s == null || s.length() <= 1)
        return true;

    char first = s.charAt(0);
    char last = s.charAt(s.length() - 1);
    if (first != last)
        return false;

    String middle = s.substring(1, s.length() - 1);
    return isPalindrome(middle);
}
```


s
racecar
first
r
last
r
middle
aceca

Example: pow2n

```
public static long pow2n(long n) {
    if (n == 0)
        return 1;
    else if (n == 1)
        return 2;
    else {
        long t = pow2n(n / 2);
        if (n % 2 == 0)
            return t * t;
        else
            return 2 * t * t;
    }
}
```

2^8

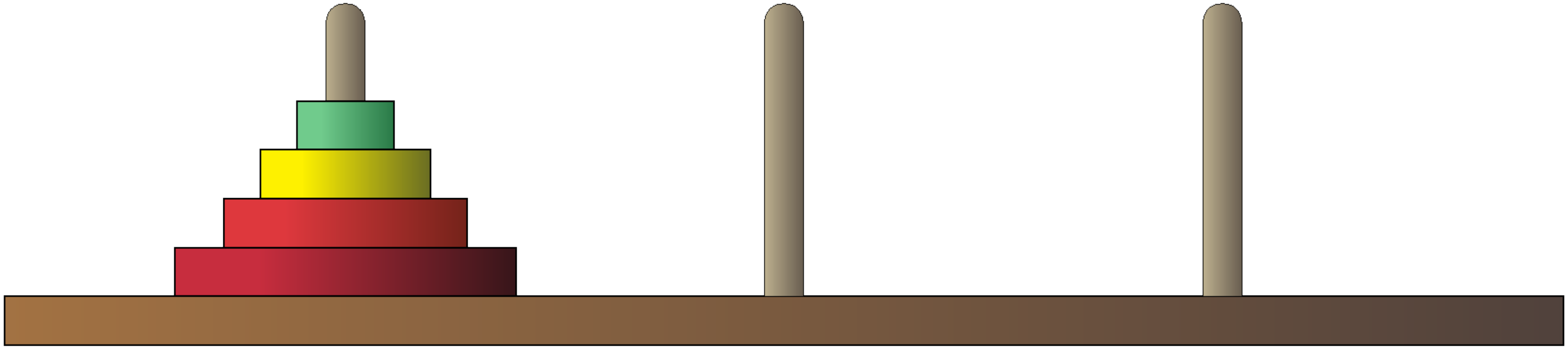
```
t = pow2n(4);
return t * t;
```

2^9

```
t = pow2n(4);
return 2 * t * t;
```

Video 3
Tower of Hanoi

Tower of Hanoi



Tower of Hanoi

- Three pegs and a tower of n disks
- Stacked in order of decreasing size
- Goal: Move all disks on one peg to another
- Rules:
  - Only move one disk at a time
  - No disk can be put on top of a smaller disk
- Demos at

  https://toh-visualizer.netlify.app/

  https://www.mathsisfun.com/games/towerofhanoi.html

Think Recursively

- Suppose I'm faced with moving a stack of 4 disks from A to C
- Pretend I can move 3 disks where ever I want by magic
  - Magic: move block of 3 disks from A to B (using C)
  - Move 4^{th} disk from A to C
  - Magic: move block of 3 disks from B to C (using A)
- "Magic" == "Recursion"

Example: Tower of Hanoi

```
public class TowerOfHanoi {
    public static void moveDisks(int n, char from, char using, char to) {
        if (n == 1) {
            System.out.printf("move disk from peg %s to peg %s\n", from,
to);
        } else {
            moveDisks(n-1, from, to, using);
            moveDisks(1, from, using, to);
            moveDisks(n-1, using, from, to);
        }
    }

    public static void main(String[] args) {
        moveDisks(4, 'A', 'B', 'C');
    }
}
```

```
moveDisks(4, 'A', 'B', 'C');

moveDisks(3, 'A', 'C', 'B');
moveDisks(1, 'A', 'B', 'C');
moveDisks(3, 'B', 'A', 'C');
```

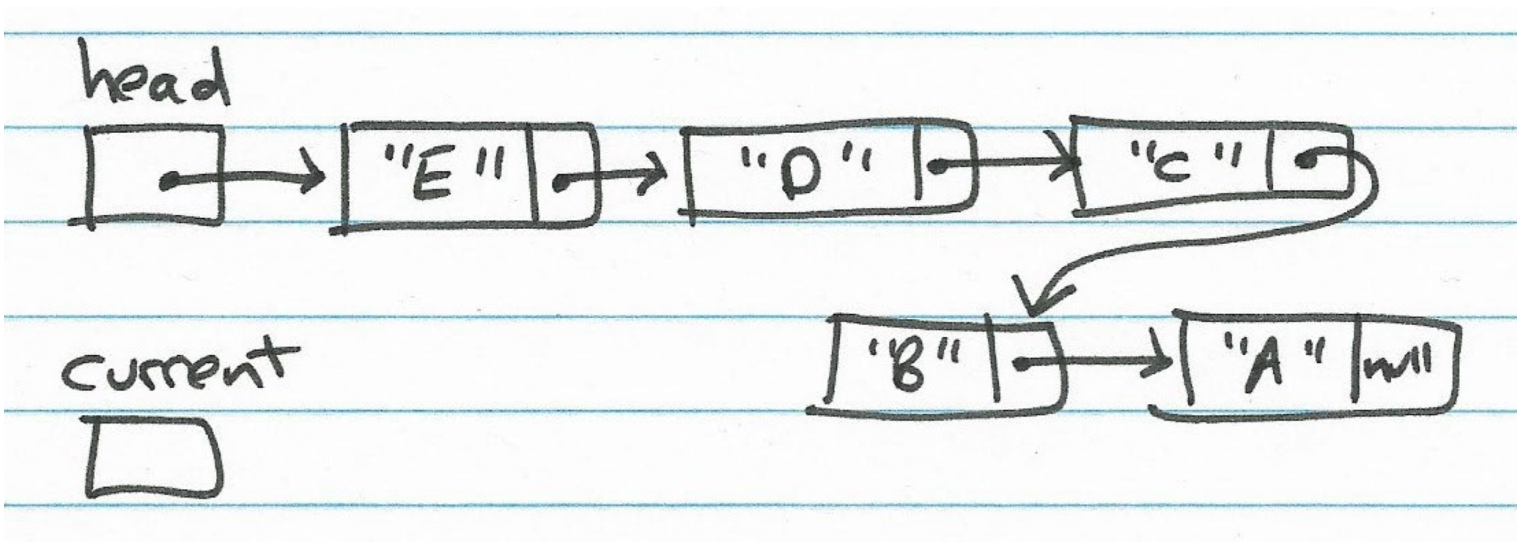
Video 4
Recursion and Linked Lists

Linked List Reminder

- Outer class contains head and tail Nodes
- Private nested class Node:
  - String value
  - Node link
- When head == tail == null, list is empty
- Method add appends to end (tail) of list
- See next slide to “walk” the list in order

Linked List

```
 public class LinkedList {
    private Node head;
    private Node tail;
    private int size;

    private class Node {
        String value;
        Node link;
    }
//...
    public String[] toArray() {  // convert list to array
        String[] array = new String[size];
        Node current = head;
        int i = 0;
        while (current != null) {  // iterate through the list
            array[i++] = current.value;
            current = current.link;
        }
        return array;
    }
```


head
"E"
"D"
"C"
current
"B"
"A"
null

Think Recursively

- A linked list is either
  - empty (head is null), or
  - a node with a link to a linked list
- Process the list recursively
  - If head is null, done
  - Else process head, then call recursively with head.link

Basis Case

Recursive Case

toArray Using Recursive fillArray

```
public String[] toArray() {
        String[] array = new String[size];
        fillArray(array, head, 0);
        return array;
}

public void fillArray(String[] array, Node current, int i)
{
        if (current == null)
            return;
        array[i++] = current.value;
        fillArray(array, current.link, i);
}
```

Counting Nodes in a Linked List

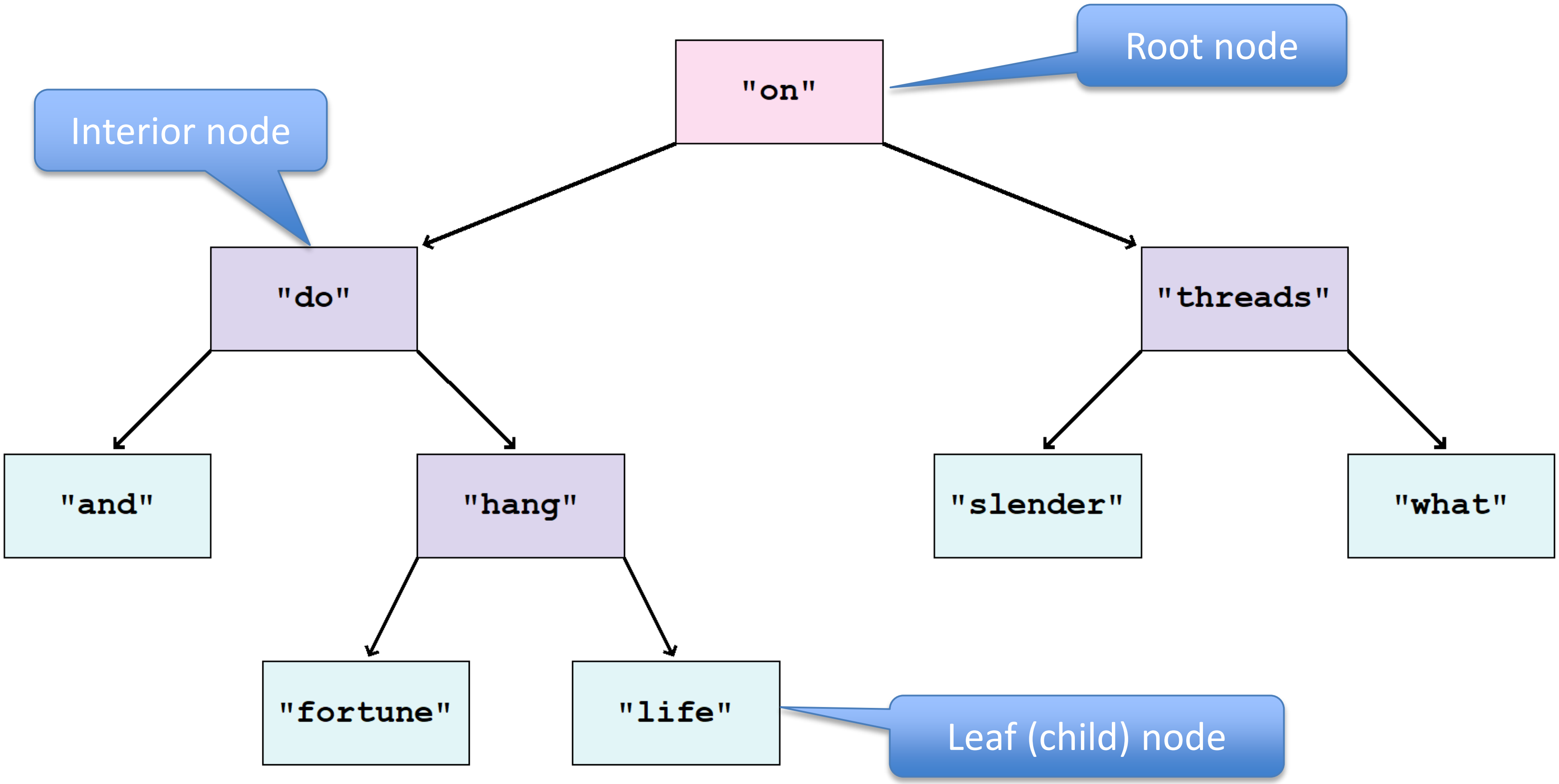
```
public int count() {                           // public method
    return count(head);
}

private int count(Node current) {              // internal helper routine
    if (current == null)                       // is this a “real” node?
       return 0;                               // no, then length is 0
   else                                        // yes, +1 for current node
        return 1 + count(current.link);        // recurse on link
}
```

Video 1
Binary Search Trees

Trees

- Linked list Node is linear with one-to-one links
- Tree Node is hierarchical with one-to-many links…
  - Parent to children
  - Boss to employees
  - Directory to files
- Can be used to model hierarchically structured data
- Allows efficient searching and sorting

Tree Example

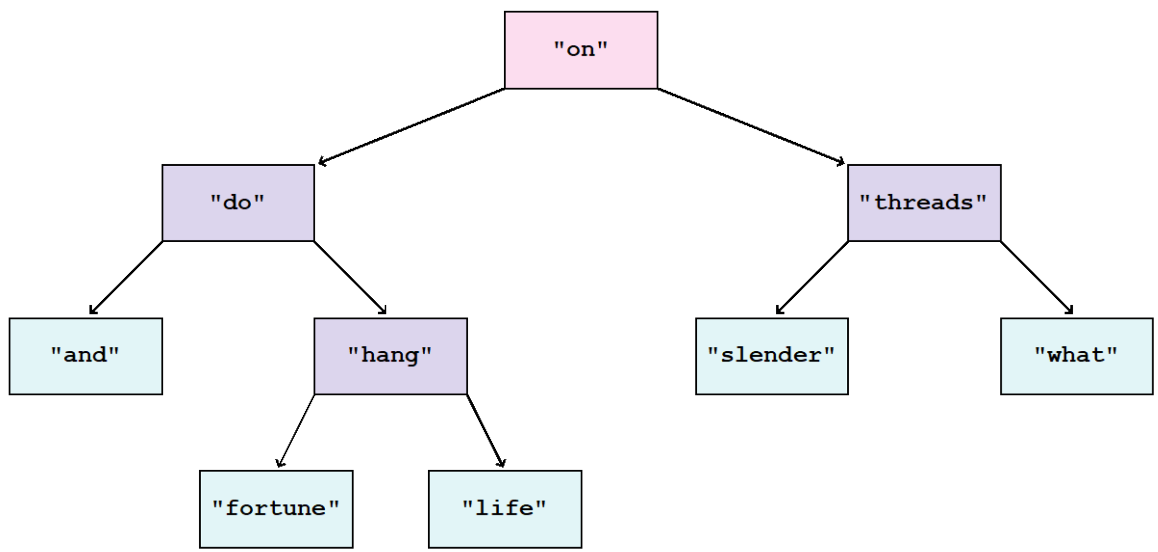


Tree Terminology

- Root node: A node with no parents
- Leaf node: A node with no children
- Interior node: Neither of the above

Think Recursively

- A tree is either
  - Empty (root is null), or
  - A node with links to 0 or more trees



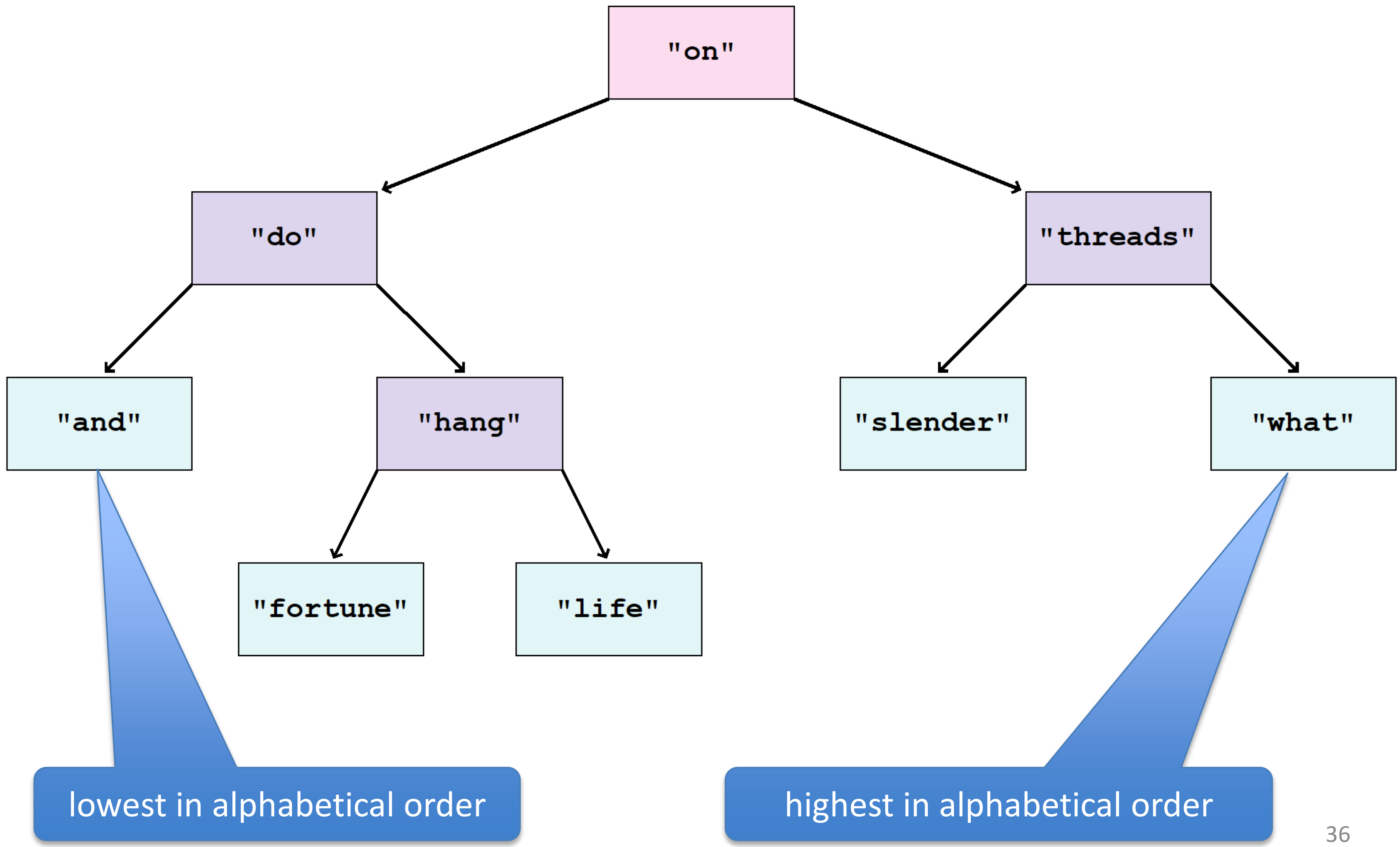
- Special case:
  - Binary tree
  - Each node references at most two other trees

Binary Search Tree

- A binary tree with a "key" at each node
- A binary search tree has three properties:
  - Key in left child of root is smaller than root
  - Key in right child of root is larger than root
  - Each child is also a binary search tree

"On what slender threads do life and fortune hang." Alexandre Dumas, *The Count of Monte Cristo*

Binary Search Tree Example



Searching a Binary Search Tree

- Problem: Is a value in the tree?
- Check root (basis case):
  - if null, return false
  - if equal, return true
- If value less than root
  - Return check of left subtree
- If value greater than root
  - Return check of right subtree
- Performance:
  - "Divide and conquer" finds the value in $\log_2 n$ comparisons
  - Compare to linked list: linear search takes n comparisons

Adding to a Binary Search Tree

- Problem: Add a new value to a binary search tree
- If tree is empty (basis case): add new Node
- If value in left subtree
  - Recursively add value to left subtree
- If value in right subtree
  - Recursively add value to right subtree
- Tricky bit: Use “proxy method” to handle initially empty tree

Example: Tree (1)

```
public class Tree {
    private static class Node {
        String value;
        Node left = null;
        Node right = null;
    }

    private Node root = null;

    // proxy add
    public void add(String value) {
        root = add(value, root);
    }
```

Example: Tree (2)

```
// ... continued

    private static Node add(String value, Node tree) {
        if (tree == null) { // basis case
            tree = new Node();
            tree.value = value;
        }
        // left recursive case
        else if (value.compareTo(tree.value) < 0)
            tree.left = add(value, tree.left);
        // right recursive case
        else if (value.compareTo(tree.value) > 0)
            tree.right = add(value, tree.right);
        return tree;
    }
```

Example: Tree (3)

```
// ... continued

    // proxy print
    public void print() {
        print(root);
    }

    private static void print(Node tree) {
        if (tree != null) {
            print(tree.left);
            System.out.println(tree.value);
            print(tree.right);
        }
    }
}
```

Traversing a Tree

- Print method on previous slide:
  - Visit left subtree
  - Visit root
  - Visit right subtree
- Called an “inorder traversal”
- Three orders:
  - inorder: visit left, visit root, visit right
  - preorder: visit root, visit left, visit right
  - postorder: visit left, visit right, visit root

Video 2
Backtracking and Recursion

Recursion and Recursive Data Structures

Recursion Examples

Another Use of Recursion

- Backtracking: Problem solving by trial and error
- Problem must be decomposable into a series of steps
  - Try step
  - So far so good? Move on (recursively)
  - Failure? Backtrack, undo step
- Each recursive instance “remembers” what step was taken and how to undo it if things don’t work out

Example: MazeSolver

- Finds a path through a maze by exhaustively trying all possible routes

Maze Representation

- Use a plain-text file of rows and columns
- In initial maze, each character is…
  - Space: an empty space (path) in the maze
  - Non-space: a wall
- Starting and ending points are pre-defined

- Goal: Place * at locations in maze to form a path

Example Maze File


```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |         |     |   |   |       |   |
+-+ +-+-+-+-+ + +-+ + + +-+ + +-+-+ +-+ +
| |     | |     |   | | |   |     |   | |
+ + +-+ + + +-+-+ + + + + +-+ +-+ +-+ + +
| | | | | | | |   | | | |       | | | | |
+ + + + + + + + +-+ + + +-+-+-+ + + + + +
|   | | |   | | |   | |     |   | |     |
+-+-+ + + + + + +-+-+ +-+-+ +-+-+ +-+ +-+
|   |     | |   |     |     |   |   |   |
+ +-+-+ +-+-+-+-+ +-+-+ +-+-+ + +-+ +-+ +
|       |   |   | |   |       |     |   |
+-+-+ +-+-+ + + + + +-+-+-+ + +-+-+-+ + +
|   | |       |   |   |   | | |     | | |
+-+ + + +-+-+-+-+-+-+ + + + + + +-+ + + +
|   | |       |   |     | | | |   |   | |
+ +-+ +-+ +-+ + +-+ + +-+ + + +-+ +-+-+ +
|     |   |   |     | |   | | |   |   | |
+ +-+-+ +-+ + +-+-+-+ +-+-+-+ + +-+-+ + +
|       |   |       |         |       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Solved Maze

start location
(1, 0)

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
****|         |     |***|   |*******|   |
+-+*+-+-+-+-+ + +-+ +*+*+-+ +*+-+-+*+-+ +
| |*****| |     |   |*|*|   |*****|***| |
+ + +-+*+ + +-+-+ + +*+*+ +-+ +-+*+-+*+ +
| | | |*| | | |   | |*|*|       |*| |*| |
+ + + +*+ + + +-+ +*+*+-+-+-+ +*+ +*+ +
|   | |*|   | | |   |*|*****|   |*|  *  |
+-+-+ +*+ + + + +-+-+*+-+-+*+-+-+*+-+*+-+
|   |  *  | |   |*****|*****|***|***|***|
+ +-+-+*+-+-+-+-+*+-+-+*+-+-+*+*+-+*+-+*+
|    ***|   |***|*|   |*******|*****|  *|
+-+-+*+-+-+ +*+*+*+ +-+-+-+ + +-+-+-+ +*+
|   |*|*******|***|   |   | | |     | |*|
+-+ +*+*+-+-+-+-+-+-+ + + + + + +-+ + +*+
|   |*|***    |   |     | | | |   |   |*|
+ +-+*+-+*+-+ + +-+ + +-+ + + +-+ +-+-+*+
|*****|***|   |     | |   | | |   |   |*|
+*+-+-+*+-+ + +-+-+-+ +-+-+-+ + +-+-+ +*+
|*******|   |       |         |     |**
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

end location
(rows-2, cols-1)

Solution Approach

- Read in the maze, store as a char[][] matrix
- Identify start and end locations (row, col)
- Call solve() method

The solve() Method

- Proxy method to get started
  - Returns true if a solution exists
  - Returns false if no solution exists
- If a solution exists, it is marked in the maze array as a series of '*' characters
- To do the work, it calls the recursive method with the starting row and column:

  solve(startRow, startCol)

Video 3
Solving a Maze Recursively

The solve(row, col) Method

- Starts at location row, col in the maze
- Assumes…
  - A series of ‘*’ are in the maze leading up to this location
- Needs to check (the special cases)…
  - Are we standing on a wall? Return false
  - Are we standing on an existing path? Return false
  - Are we at the end location? Return true

The solve(row, col) Method

- Once the special cases are done…
- Leave mark (‘*’) behind as we move
  - Like “bread crumbs”
  - Ensures that final path is identified
  - Prevents us from looping back onto path
- If we reach a dead end…
  - Remove mark (reset to ‘ ’)
  - Return false

A Trick

- Since we are not in a physical maze…
  - It is OK to move first and ask questions later
  - If outside maze, on a wall, or on an existing path, then return false

Solve: Failure Cases

- Moved outside the maze
- Standing on a wall
- Standing on an existing path location (looping)

- In all three cases: return failure to initiate backtracking at the previous level

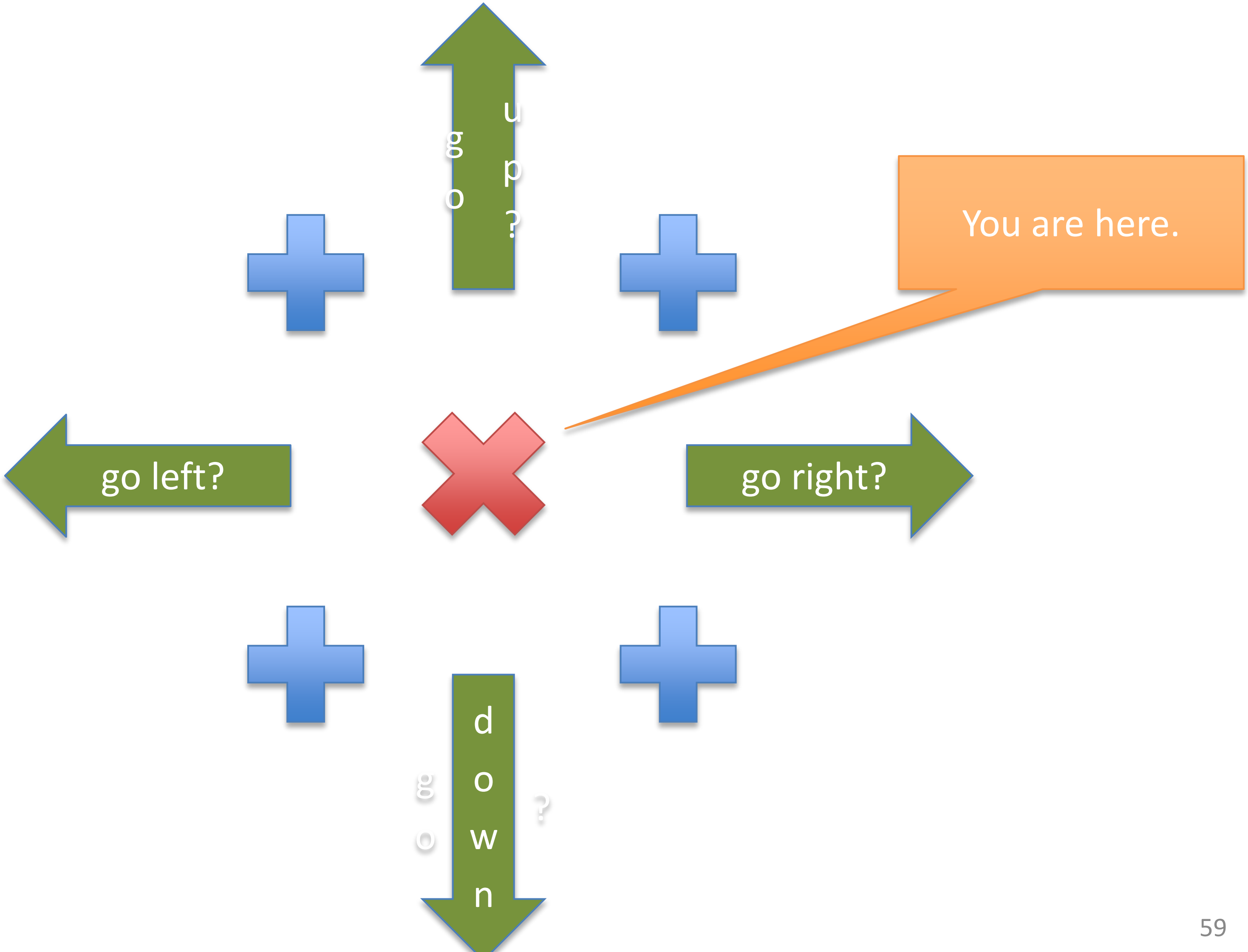
Solve: Basis Case

- Current location == end location: we’re done!
- Return true

Solve: Recursive Case

- Mark the current square as on the path
- Make calls to solve(…) on all adjacent locations to see if we can get to the end
- If any of them returns true, return true to our caller (success!)
- Else unmark the current square and return false (failure!)

In the Maze



Example: MazeSolver

- Create simple maze
  - Entrance
  - Forked path, one dead-end, other working
  - Exit
- Start at start, follow algorithm to dead-end
- Backtrack
- Continue recursion to exit

MazeSolver: solve Method

```
private boolean solve(int row, int col) {
    // handle special cases (out of bounds and walls)
    if (row < 0 || col < 0 || row >= rows || col >= cols || maze[row][col] != ' ')
        return false;

    // mark this location as on the path...
    maze[row][col] = '*';

    // basis case: see if we're done...
    if (row == endRow && col == endCol)
        return true;

    // recursive case: try surrounding spaces...
    if (solve(row-1, col) || solve(row+1, col) || solve(row, col-1) || solve(row, col+1))
        return true;

    // no solution found from this location; backtrack and return failure...
    maze[row][col] = ' ';
    return false;
}
```