

Linear Collections I

Stack

Queue

List

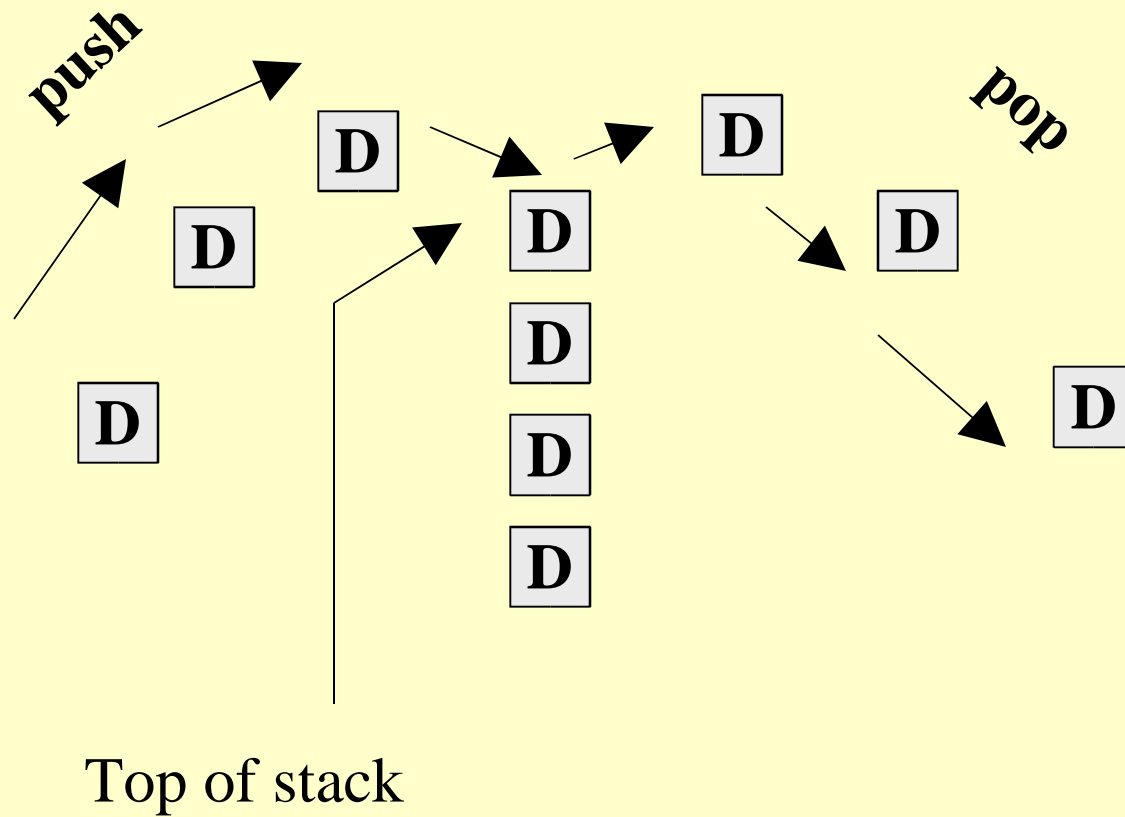
The Stack ADT: Overview

- Formal properties
- Applications
- Implementations

Stack ADT: Formal Properties

- *Stacks* are like arrays and vectors, in that elements are ordered by position (linear)
- However, access is only at one end, called the *top*
- A stack is an ADT that supports last-in, first-out (LIFO) access

LIFO Access



Minimal Set of Stack Operations

```
boolean isEmpty() // Returns true if empty, false otw
```

Minimal Set of Stack Operations

```
boolean isEmpty()    // Returns true if empty, false otw  
int size()           // Returns the number of items
```

Minimal Set of Stack Operations

```
boolean isEmpty()      // Returns true if empty, false otw  
int size()             // Returns the number of items  
void push(Object obj) // Adds obj to top of stack
```

Minimal Set of Stack Operations

```
boolean isEmpty()      // Returns true if empty, false otherwise
int size()             // Returns the number of items
void push(Object obj) // Adds obj to top of stack
Object pop()           // Removes obj from top of stack
                       // and returns it
```

Minimal Set of Stack Operations

```
boolean isEmpty()      // Returns true if empty, false otherwise
```

```
int size()             // Returns the number of items
```

```
void push(Object obj) // Adds obj to top of stack
```

```
Object pop()           // Removes obj from top of stack  
                       // and returns it
```

```
Object peek()          // Returns obj at top of stack
```

NOTE: The precondition of pop and peek is that the stack is not empty.

Example Use of a Stack

```
Stack stack = new LinkedStack();

stack.push ("A string");
stack.push (new Student("Ken", 100, 90, 70));
stack.push (new Integer(45));
stack.push (new ArrayTiny());

while (! stack.isEmpty()){
    Object obj = stack.pop();
    System.out.println (obj);
}
```

Stack ADT: Applications

Stacks are useful for algorithms that must backtrack to the most recent data element in a series of elements

- Run-time support of recursion
- Language processing
- Game playing, puzzle solving

How Recursion Works

- Each call of a method generates an *instance* of that method
- An instance of a method contains
 - memory for each parameter
 - memory for each local variable
 - memory for the return value
- This chunk of memory is also called an *activation record*

Example: `factorial(4)`

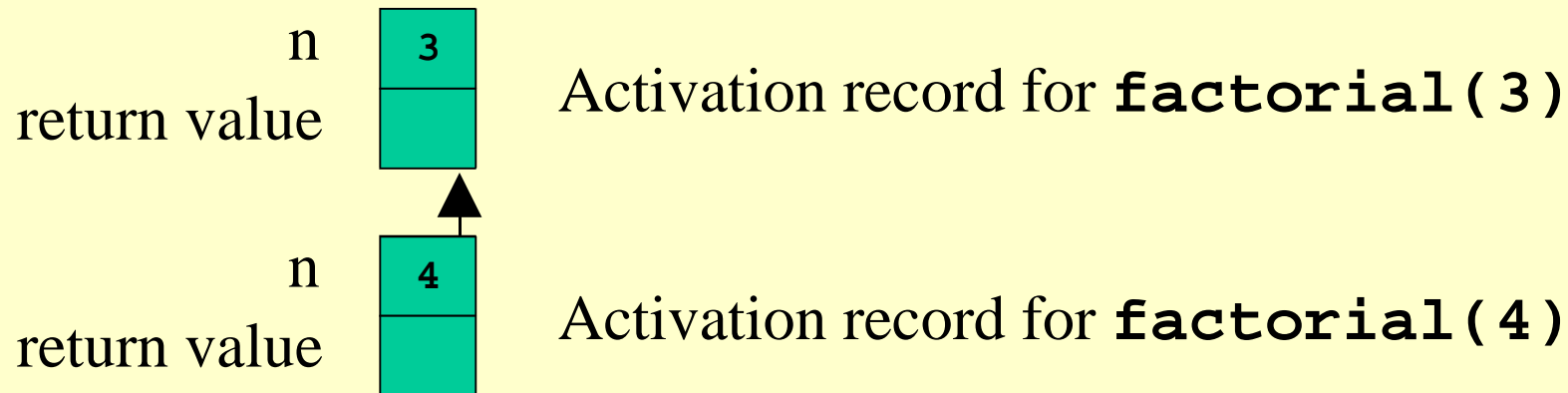
```
int factorial (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n * factorial (n - 1);  
}
```

return value n

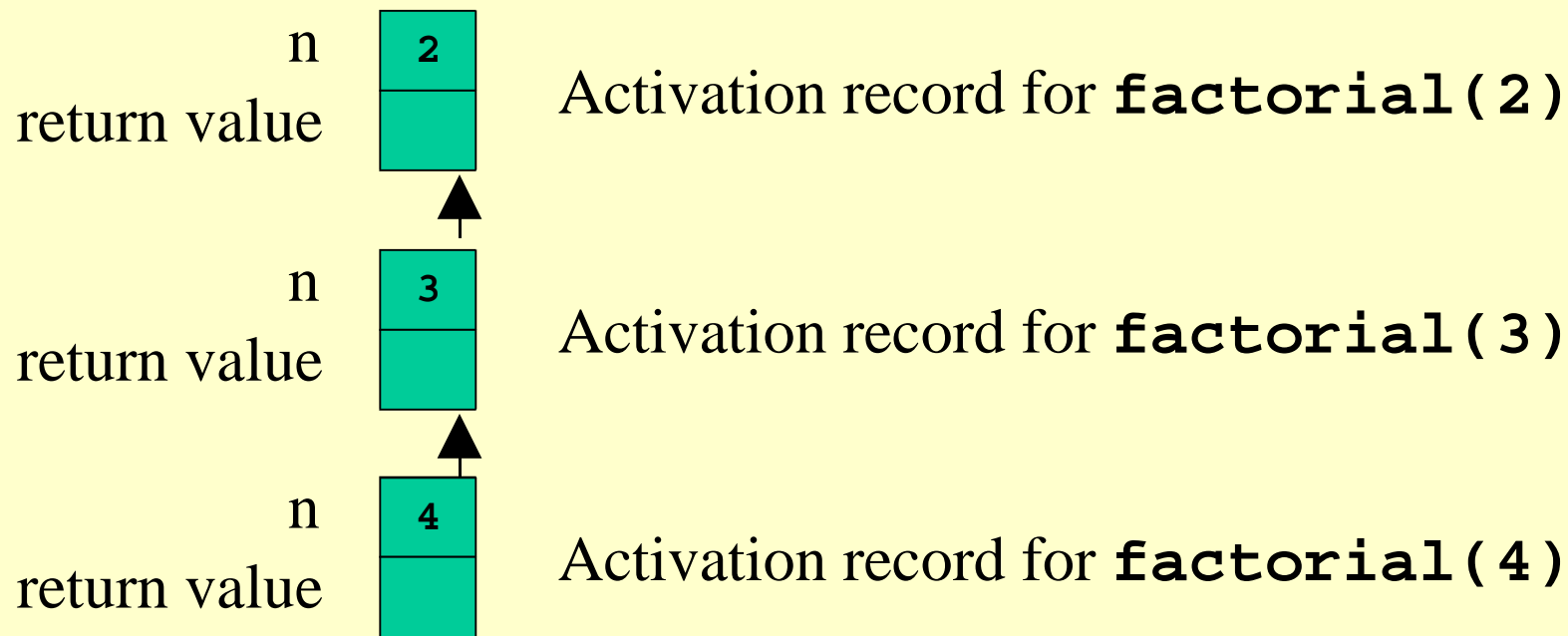
4

 Activation record for `factorial(4)`

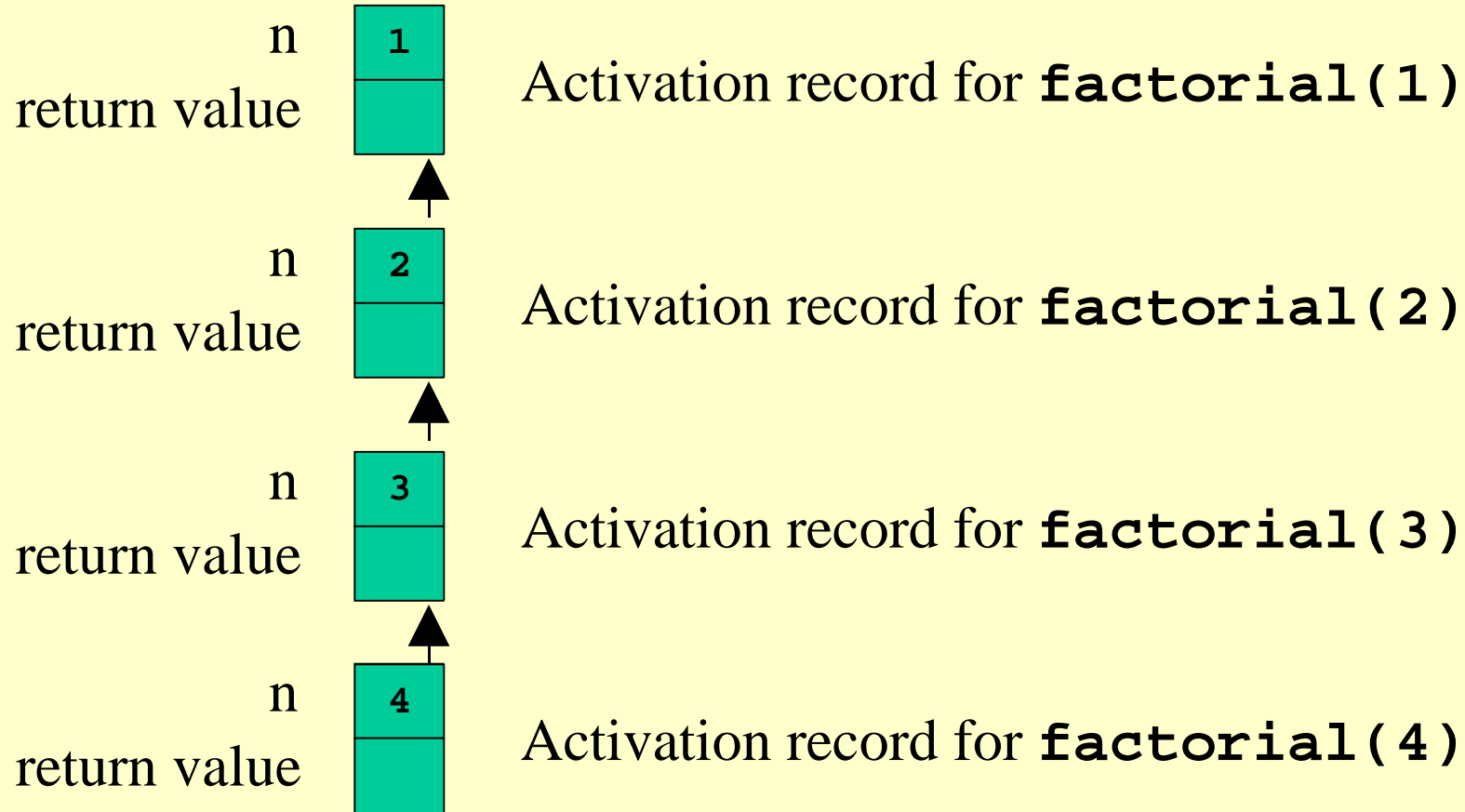
Activations Are Added Dynamically



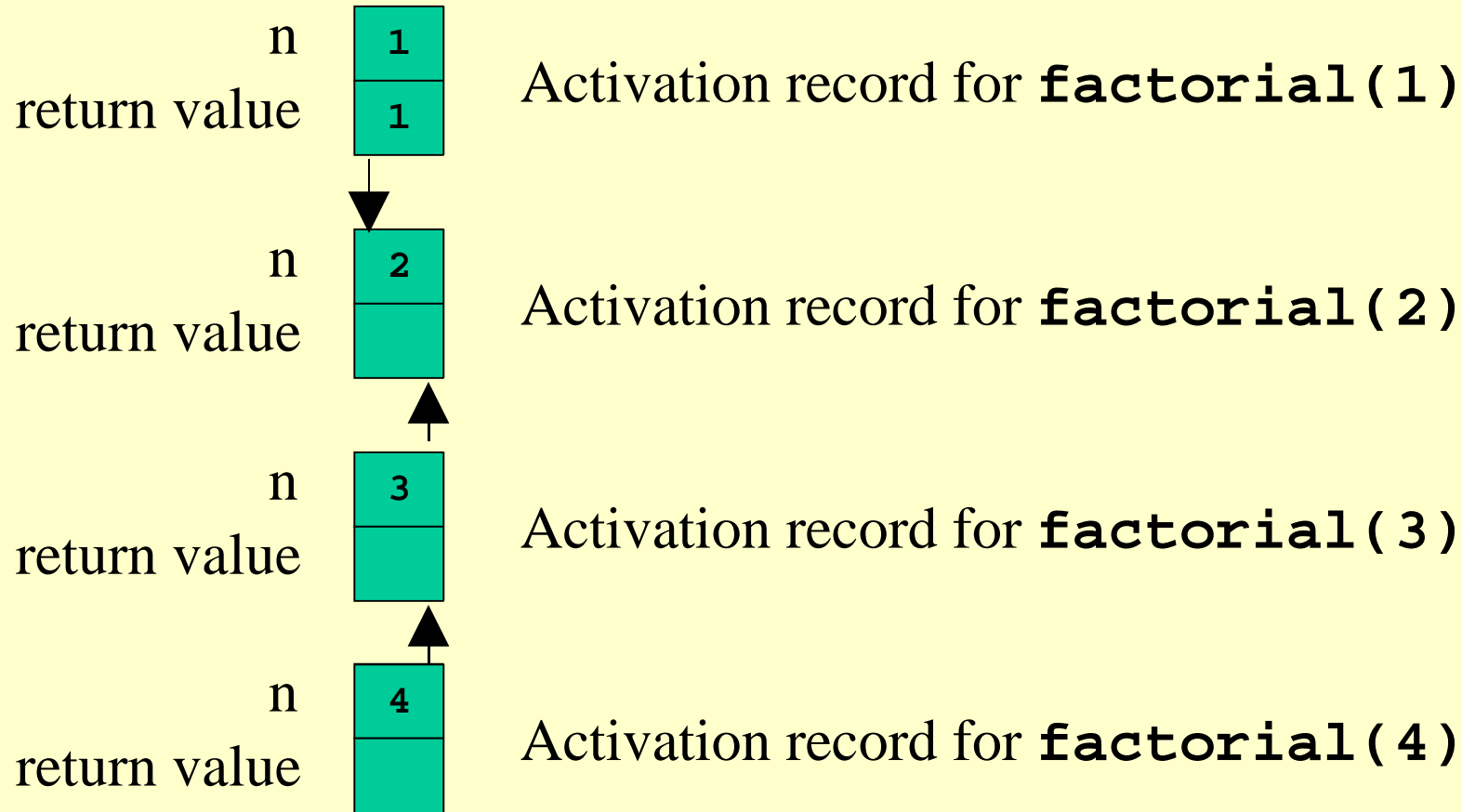
Number of Activations = # Calls



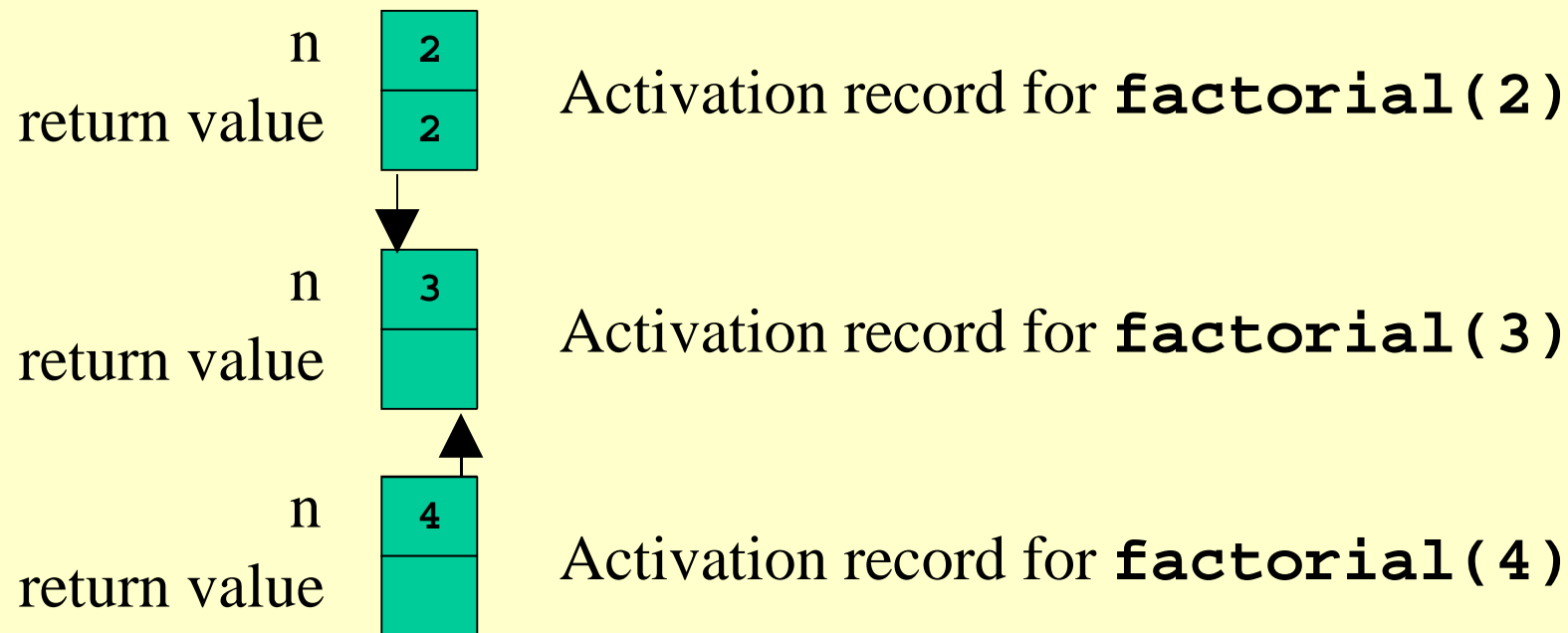
Recursive Process Bottoms out



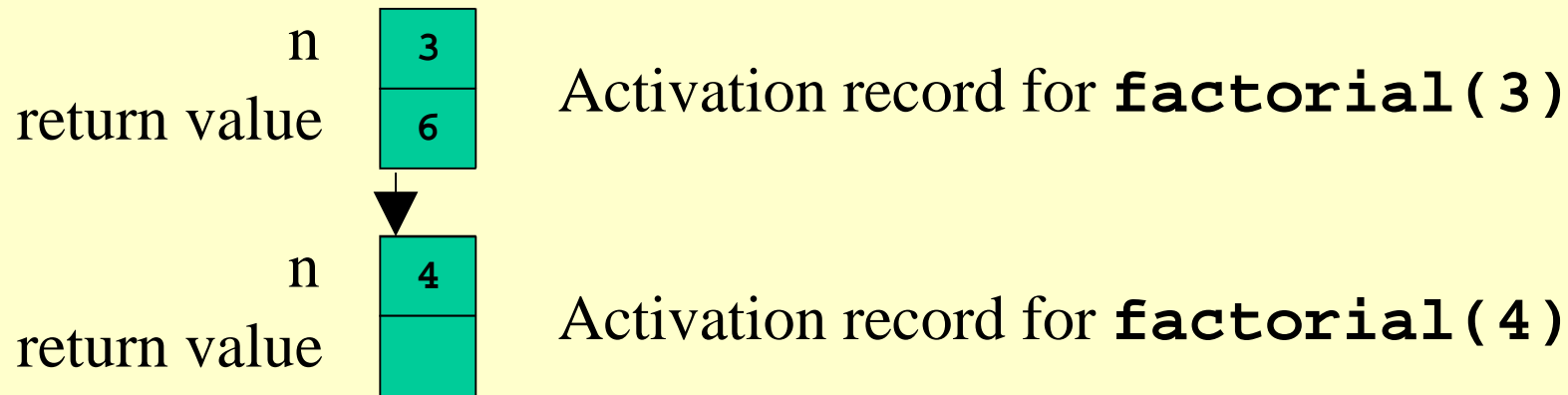
Recursive Process Unwinds



Activations Are Deallocated



Activations Are Deallocated



Value Returned Is in the First Activation

return value n

4
24

 Activation record for **factorial(4)**

Loop with Explicit Stack

```
int factorialWithLoopAndStack (int n){
    Stack stk = new LinkedStack();

    // Load up the stack with the sequence of numbers from n down to 1.
    while (n >= 1){
        stk.push (new Integer(n));
        n --;
    }

    // Pop the top two numbers and push their product onto the stack,
    // until there is just one number left on the stack.
    while (stk.size() > 1){
        int operand1 = ((Integer)stk.pop()).intValue();
        int operand2 = ((Integer)stk.pop()).intValue();
        int product = operand1 * operand2;
        stk.push (new Integer(product));
    }

    // Return the number at the top of the stack.
    return ((Integer)stk.peek()).getValue();
}
```