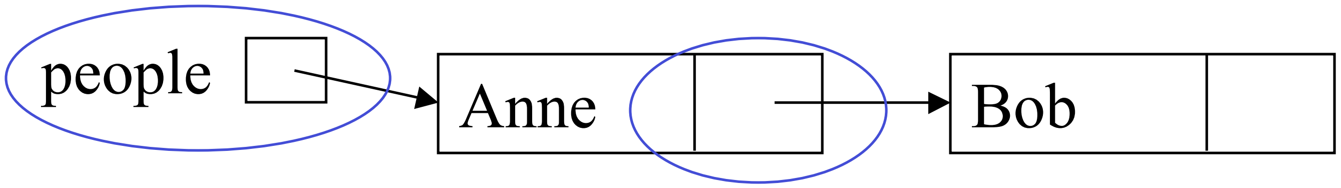
CS112: Data Structures

Lecture 04

Recursion on lists

Review: Dummy Headers

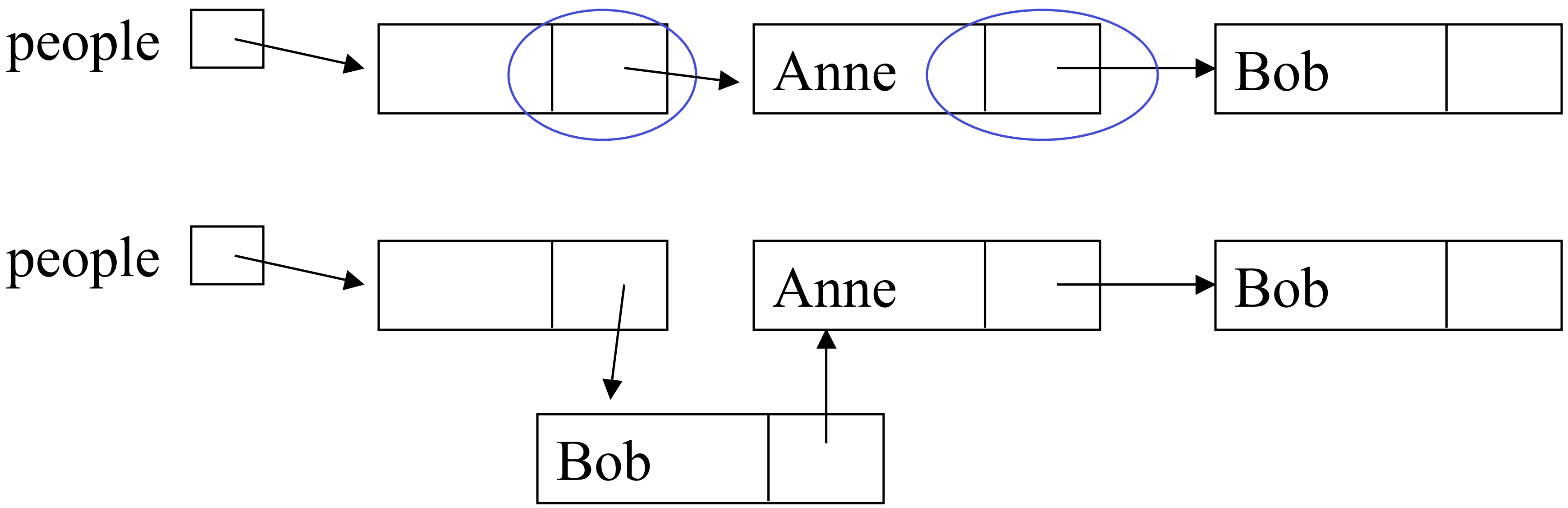
- **Problem: In a simple linked list there are two different kinds of place we can have a pointer to a Node**



- **Insert-at-head and insert-after-node require different code**

Review: Dummy Headers

- **Solution: add an extra "dummy" Node to point to the first real Node in the list**



Review: Iterators

- Abstract data type: a container
  - E.g. array or linked list
  - Can do mostly the same things with them, e.g. insert. delete
  - One of the things I want to do is go through the data items one by one

Solution

- Methods you can use to build the loop
  - hasNext
  - getNext
- State: an object
  - Represents a particular instance of iteration
  - Initialized by new

Abstract List Traversal

- while (list.hasNext()) {
  ```
  while (list.hasNext()) {
        print(list.getNext().data);
  }
  ```

- list could be an Array:
  ```
        hasNext()  { return (i != list.length) }
        getNext()  { i++;  return list[i]; }
  ```

Abstract List Traversal

- while (list.hasNext()) {

```
      print(list.getNext().data);
}
```

- list could be a LinkedList:

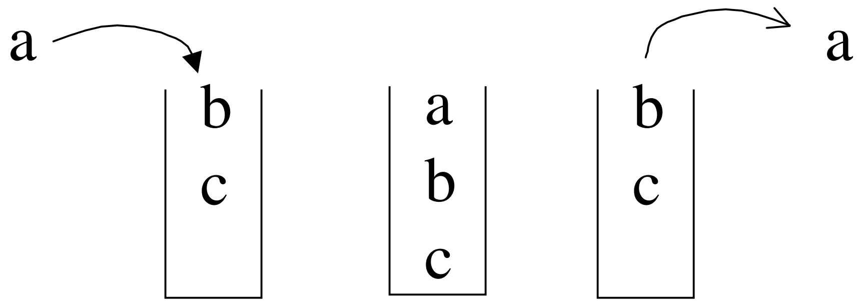
```
      hasNext()  { return (curr != null) }
      getNext()  { curr = curr.next;
                   return curr; }
```

Iterators

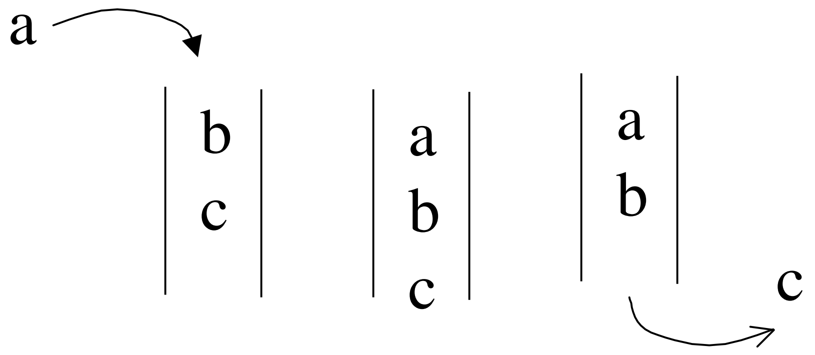
- See StringList.java and StringListIterator.java

Review: Stacks & Queues

- **Last in first out: Stack**



- **First in first out: Queue**



Operations

- **Queue**
  - enqueue, dequeue
  - isEmpty, size
  - clear, remove, removeAll
  - first, next (Enumerator would be better.)
- **Stack**
  - same but enqueue, dequeue called push, pop

Uses of Queues

- **Printer queue**
- **Simulation of real world queues**
  - **Queue in simulator models line of students.**
- **More generally, waiting lists when processing one item creates two more items to process**
  - **E.g. simulator**
  - **E.g. family tree**

Invocation Record

- **Each procedure / method call needs to record values of**
  - **Parameters**
  - **Local variables**
  - **Other things**
- **When a procedure starts, space allocated for “invocation record” to store these things.**
- **Invocation record is kept until invocation exits**
- **Behavior is LIFO**

Stack of Invocation Records

Proc foo(int a)

… int b, c;

… fie(b);

… fie(c);

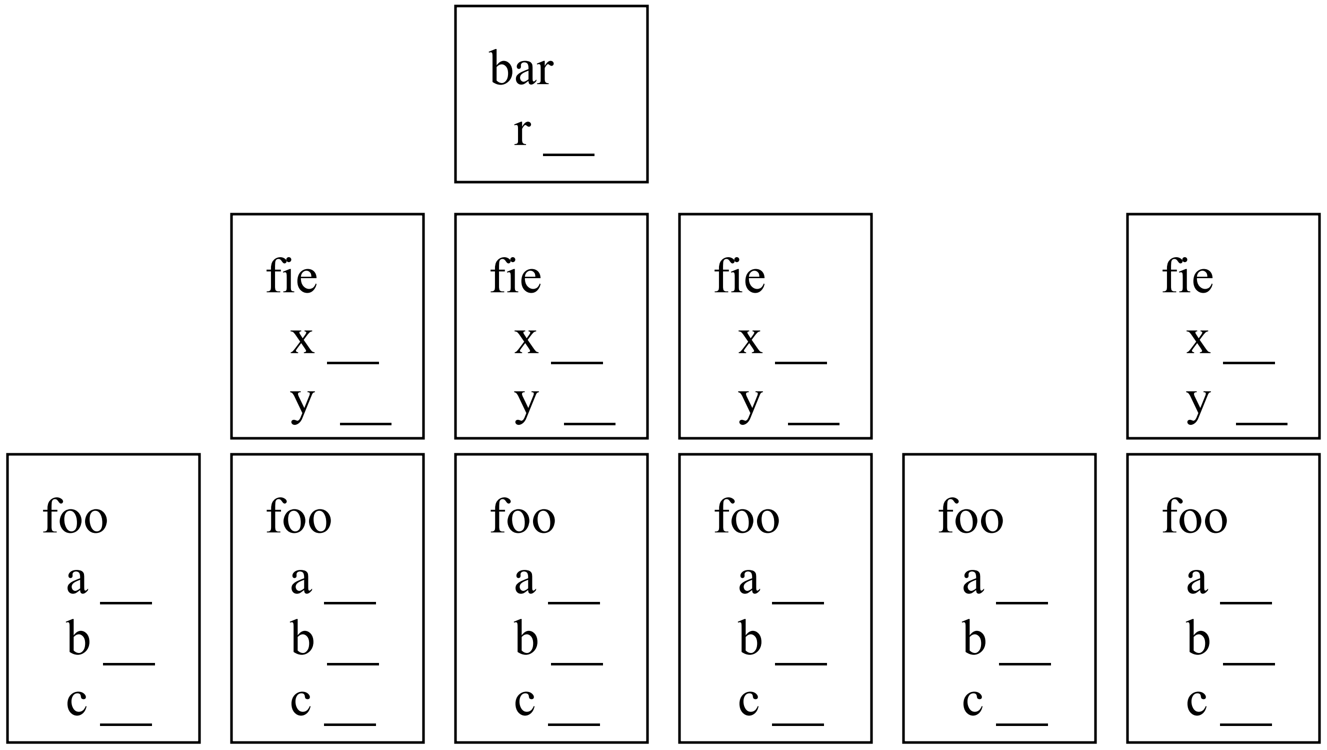
Proc fie(int x)

…int y;

…bar(y);

Proc bar(int r)

…

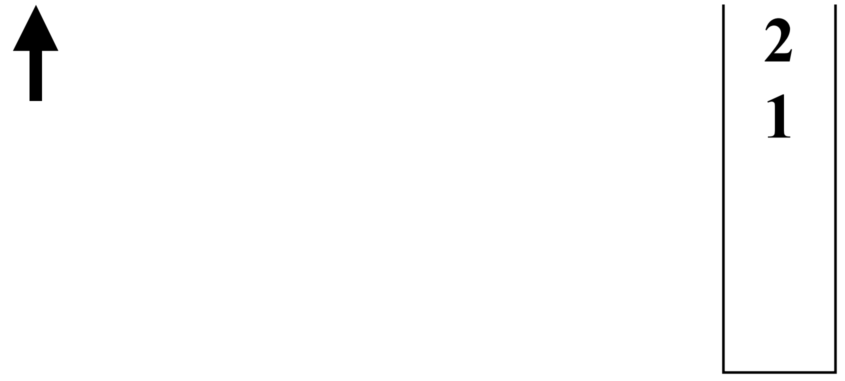


Uses of stacks

- Postfix (RPN) calculator
  - Permits any expression to be evaluated
  - Does not require parentheses

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /

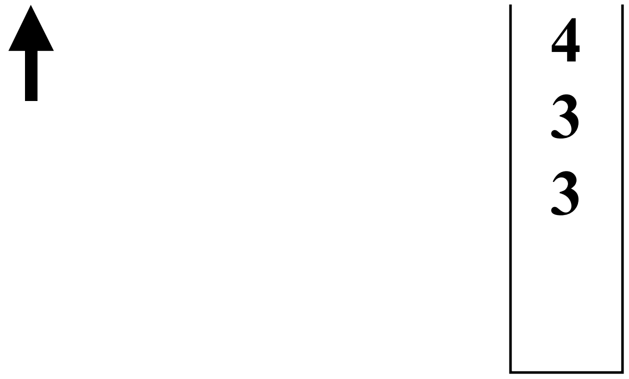


Uses of stacks

- Postfix (RPN) calculator
  - Permits any expression to be evaluated
  - Does not require parentheses

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /

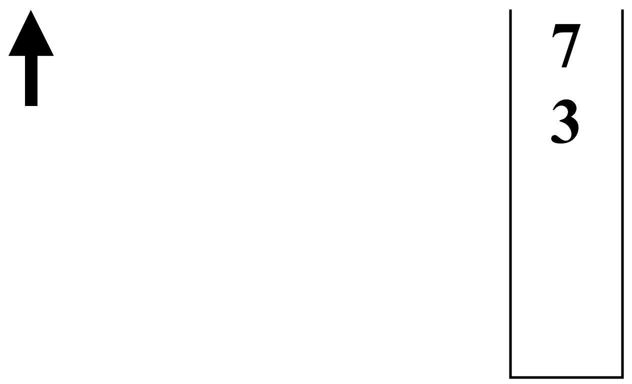


Uses of stacks

- **Postfix (RPN) calculator**
  - **Permits any expression to be evaluated**
  - **Does not require parentheses**

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /

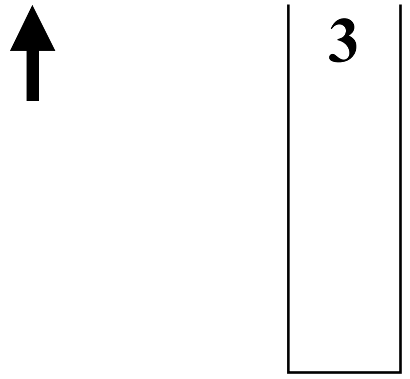


Uses of stacks

- Postfix (RPN) calculator
  - Permits any expression to be evaluated
  - Does not require parentheses

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /

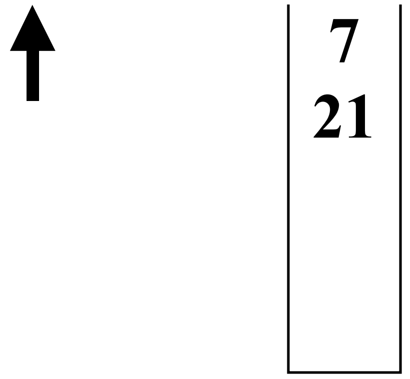


Uses of stacks

- Postfix (RPN) calculator
  - Permits any expression to be evaluated
  - Does not require parentheses

((1 + 2) * (3 + 4)) / 7

1 2 + 3 4 + * 7 /



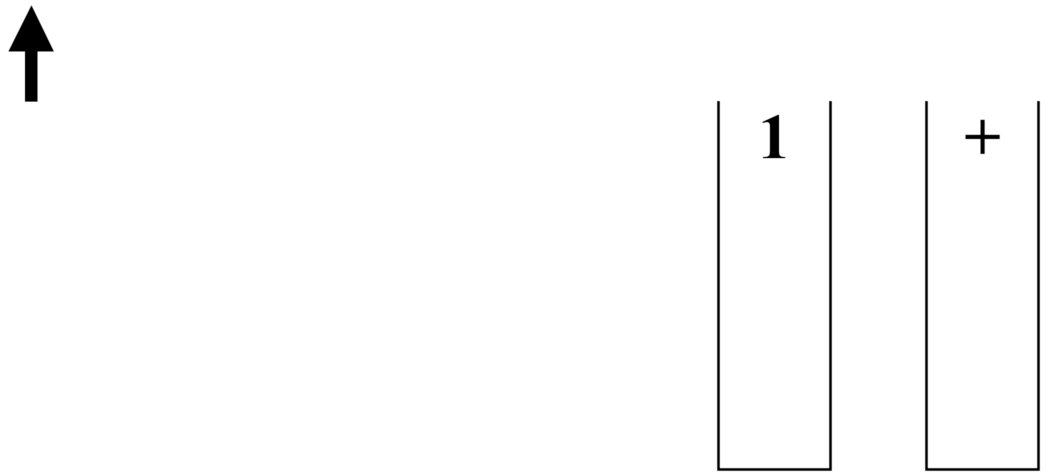
Uses of stacks

- Interpret infix-with-precedence
  - Each operator has a numeric precedence
    - + 10, * 20, > 5, < 5
  - Two stacks: operators, operands
  - scan expression:
    - operand: push
    - operator:
      - if operator stack empty or
        precedence > top of stack, push
      - else: pop operator & do
- e.g. 1 + 2 * 3 > 4 * 5

Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

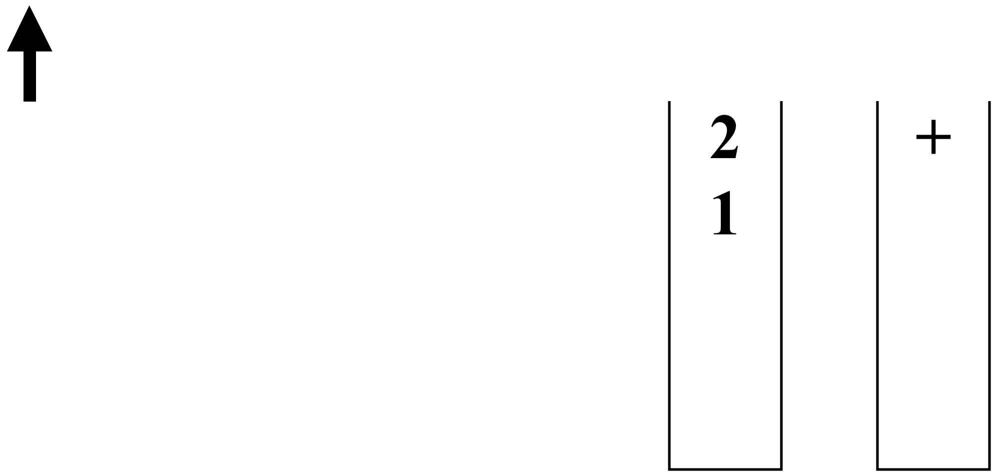
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    $\boxed{\text{precedence > top of stack}}$, push
  - else: pop operator & do

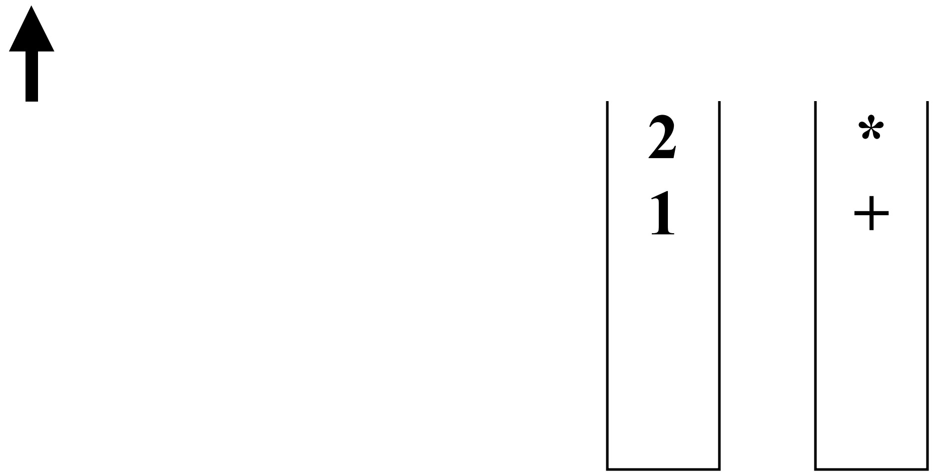
- **e.g. 1 + 2 * 3 > 4 * 5**



Uses of stacks

  - operand: push
  - operator:
    - if operator stack empty or
      precedence > top of stack, push
    - else: pop operator & do

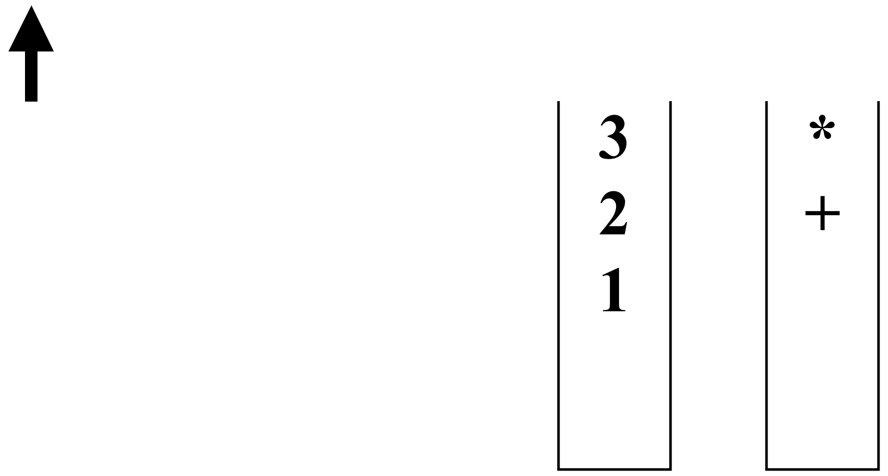
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - $\boxed{\text{else}}$: pop operator & do

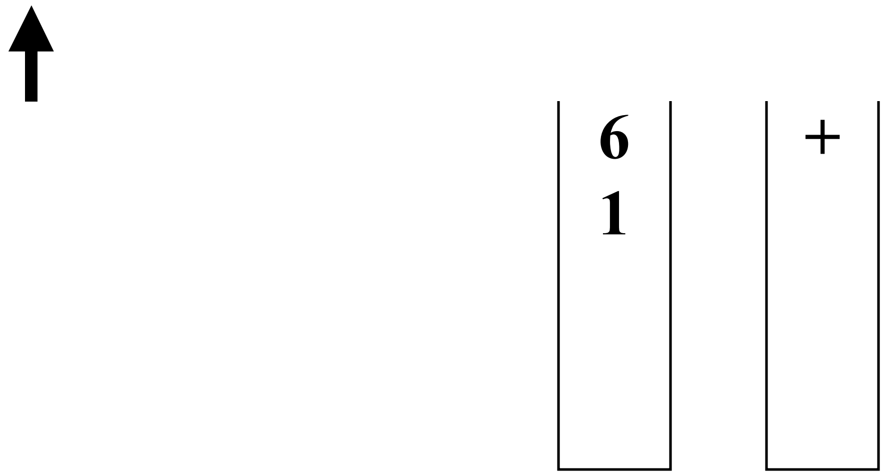
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - $\boxed{\text{else}}$: pop operator & do

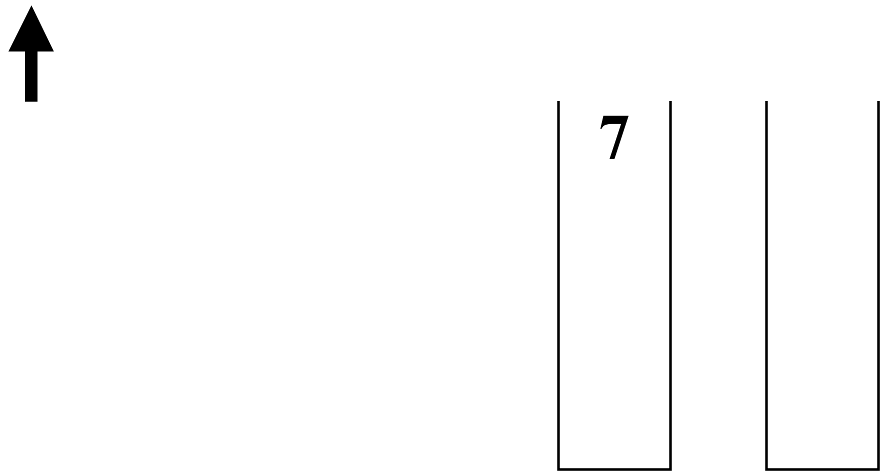
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or precedence > top of stack, push
  - else: pop operator & do

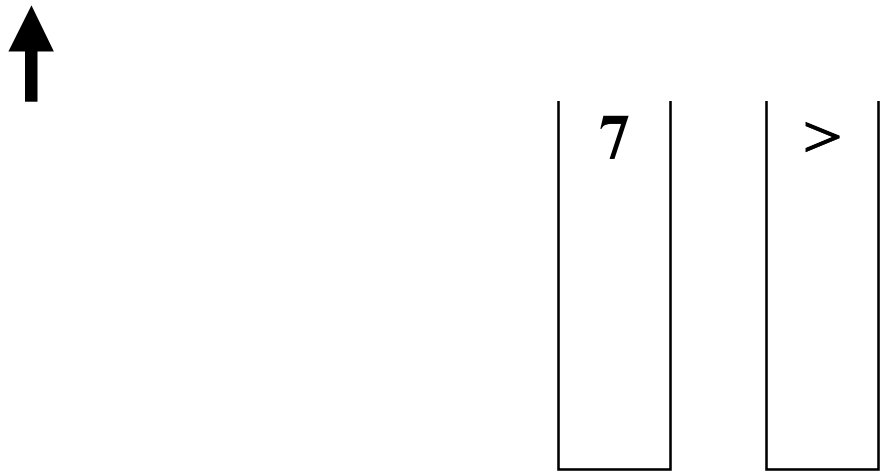
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or precedence > top of stack, push
  - else: pop operator & do

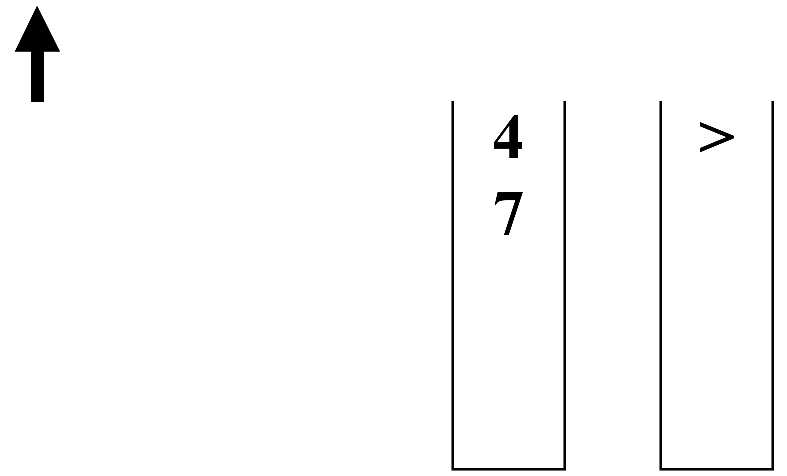
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

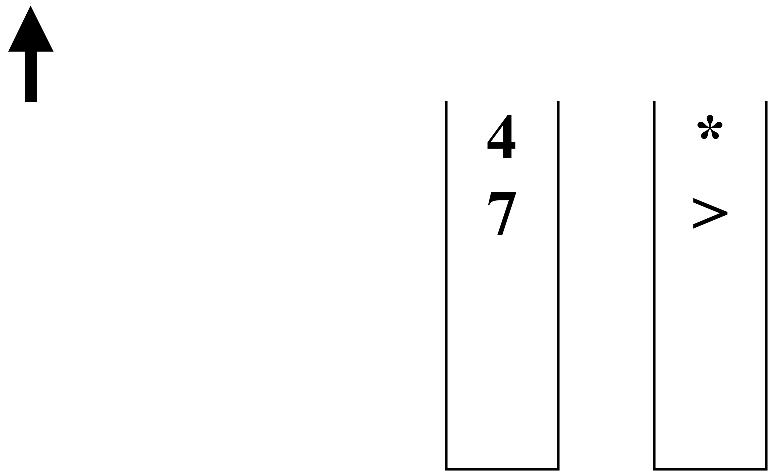
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

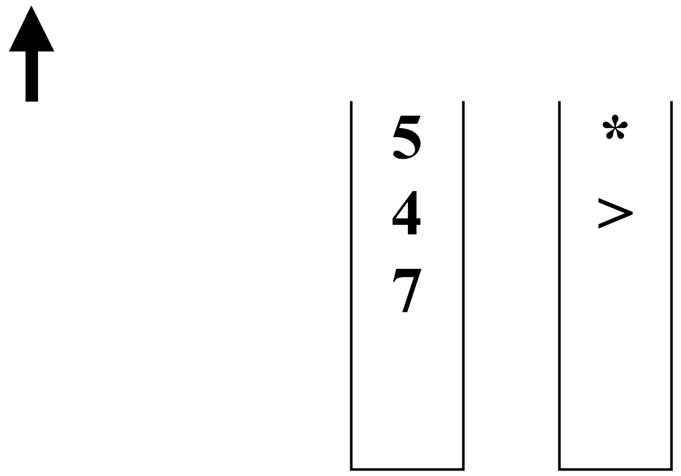
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or precedence > top of stack, push
  - else: pop operator & do

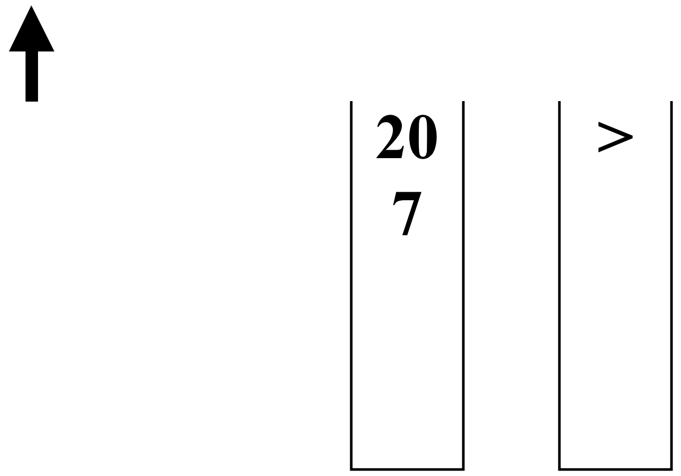
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

- operand: push
- operator:
  - if operator stack empty or precedence > top of stack, push
  - else: pop operator & do

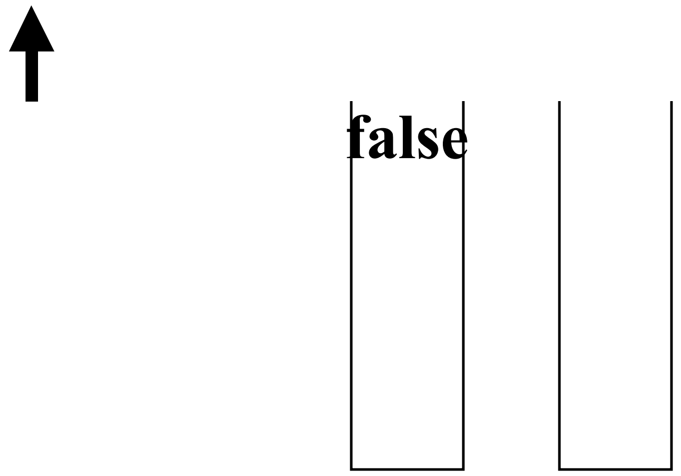
- e.g. 1 + 2 * 3 > 4 * 5



Uses of stacks

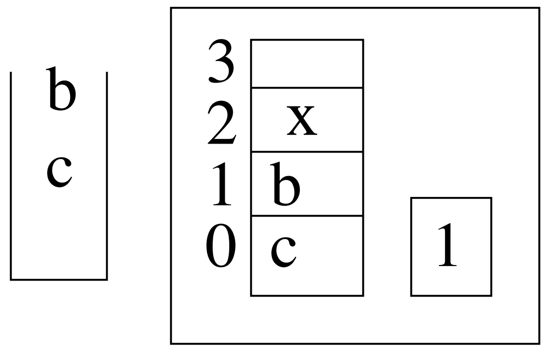
- operand: push
- operator:
  - if operator stack empty or
    precedence > top of stack, push
  - else: pop operator & do

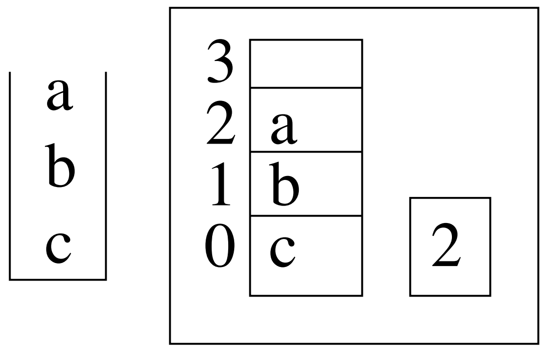
- e.g. 1 + 2 * 3 > 4 * 5



Implementing Stacks

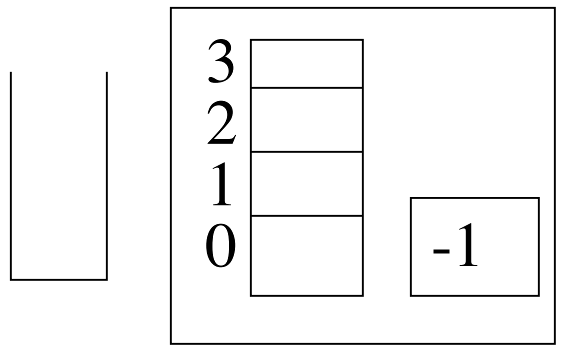
- Stacks can be implemented using
  - Arrays
  - Linked lists
- Arrays:
  - Array holds data, also need int "top of stack"



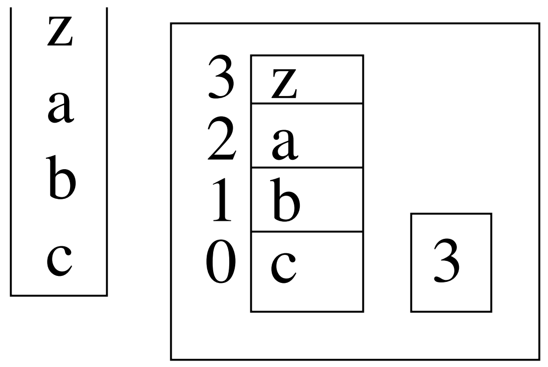


Implementing Stacks

- **Empty stack: top == -1**

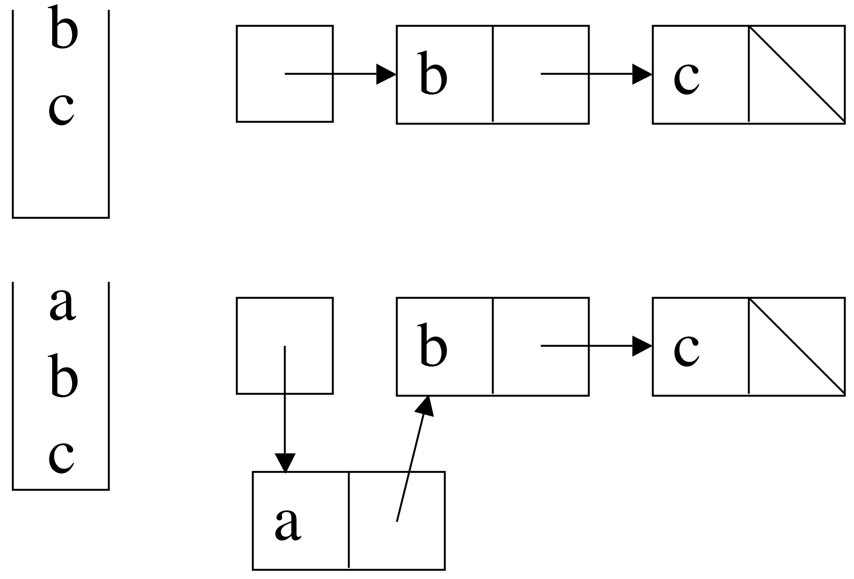


- **Full stack: top == array size -1**



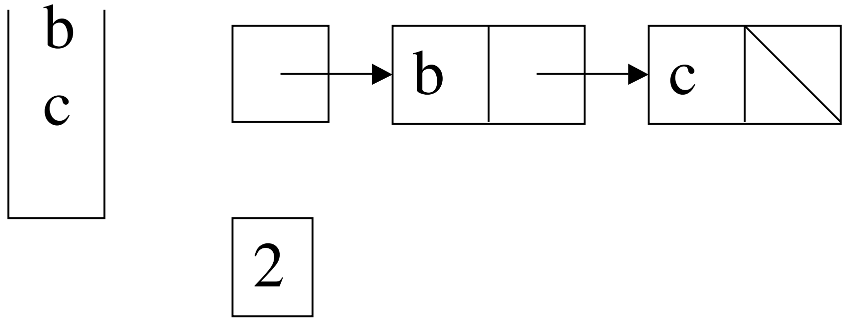
Stacks as linked lists

- **Linked lists are easy to manipulate at head -> natural representation for stacks**



Stacks as linked lists

- Only operation that is not fast is size
- So also have an int

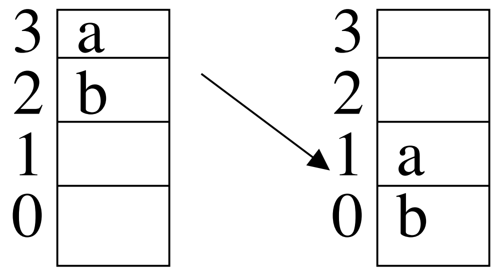


Implementing Queues

- **Queues need to be accessed at both ends, so implementations are a bit messier**
  - **Arrays: need two ints to keep track of both front and back**
  - **linked lists: use circular lists or have two pointers**

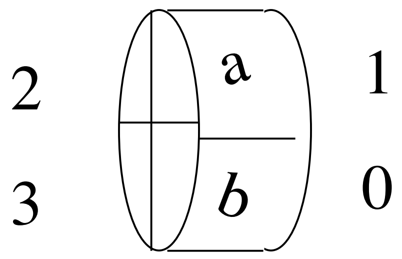
Queues as arrays

- **Problem: how to reuse space emptied by dequeue?**
  - **Could move data down**



  - **Treat array as circular**

    front = (front + 1) %size



Queues as linked lists

- **Problem: Need to access both ends**
- **Solution: Linked list with head/tail pointer or circular linked list**
- **Which end of the list should be the front of the queue?**
  - **enqueue is O(1) time whether at head or tail**
  - **dequeue is O(1) at head but O(n) at tail (Why? Need pointer to second to last.)**
  - **so more efficient when front is head**

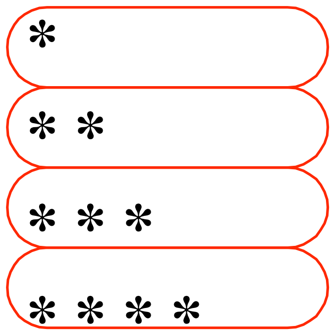
Recursion

- Recursion is a way of looking at a problem
- EG problem: print pattern like

```
*
* *
* * *
* * * *
```

Recursion

- **Non-recursive view**

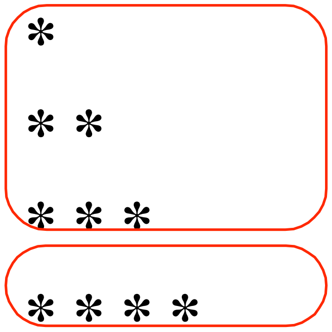


```
*
* *
* * *
* * * *
```

- **A size 4 triangle is four lines, lengths 1, 2, 3, 4**

Recursion

- **Recursive view**



- **A size 4 triangle is**
  - **A size 3 triangle, followed by**
  - **A line of length 4**

Recursive Definitions in Math

- **Factorial**

  n! = n * (n-1)!

  1! = 1

  e.g., 3! = 3*2! = 3 * (2 * 1!) = 3 * (2 * 1) = 6

- **Definition looks circular, but is not**
- **Two parts:**
  - **recursive case**
  - **base case**

Recursive Methods

- **Example: palindrome**
  - **same letters backwards as forwards (assume no space or punctuation)**
  - **e.g, radar**
    madam im adam

    a man a plan a canal panama
- **How can we write a method to test if a string is a palindrome?**

Recursive Definition

- A string is a palindrome if
  - first and last characters are the same, and

    radar

  - rest of string without first and last is a palindrome

    ada

- A string of length 0 or 1 is a palindrome

  d

Integer Power

How many multiplies does it take to calculate 3^8 ?

$3 * 3 = 9$	3^2
$9 * 3 = 27$	3^3
$27 * 3 = 81$	3^4
$81 * 3 = 243$	3^5
$243 * 3 = 729$	3^6
$729 * 3 = 2187$	3^7
$2187 * 3 = 6561$	3^8

7 *'s

$3 * 3 = 9$	3^2
$9 * 9 = 81$	3^4
$81 * 81 = 6561$	3^8

3 *'s

$$3^{y/2} * 3^{y/2} = 3^{(y/2+y/2)}$$
$$= 3^y$$

Integer Power

What happens when y is odd?

3 * 3 = 9 3^2

9 * 9 = 81 3^4

81 * 3 = 243 3^5

3 *'s

Recursive Definition

- y even: $x^y = x^{y/2} * x^{y/2} = (x^{y/2})^2$
- y odd: $x^y = x * x^{\lfloor y/2 \rfloor} * x^{\lfloor y/2 \rfloor}$

  $= x * (x^{\lfloor y/2 \rfloor})^2$
- y = 1: $x^y = x$
- y = 0: $x^y = 1$

- **Seeing recursion**
  - **look at a problem as if it were "pregnant":**
    - **inside it is a small version of the same problem**

Is "radar" a palindrome?
is "ada" a palindrone?

How Does Recursion Work?

Non Recursive:

```
static void triangle1( ){
        printNStars(1);}

static void triangle2( ){
        triangle1( );
        printNStars(2);}

static void triangle3( ){
        triangle2( );
        printNStars(3);}
```

How Does Recursion Work?

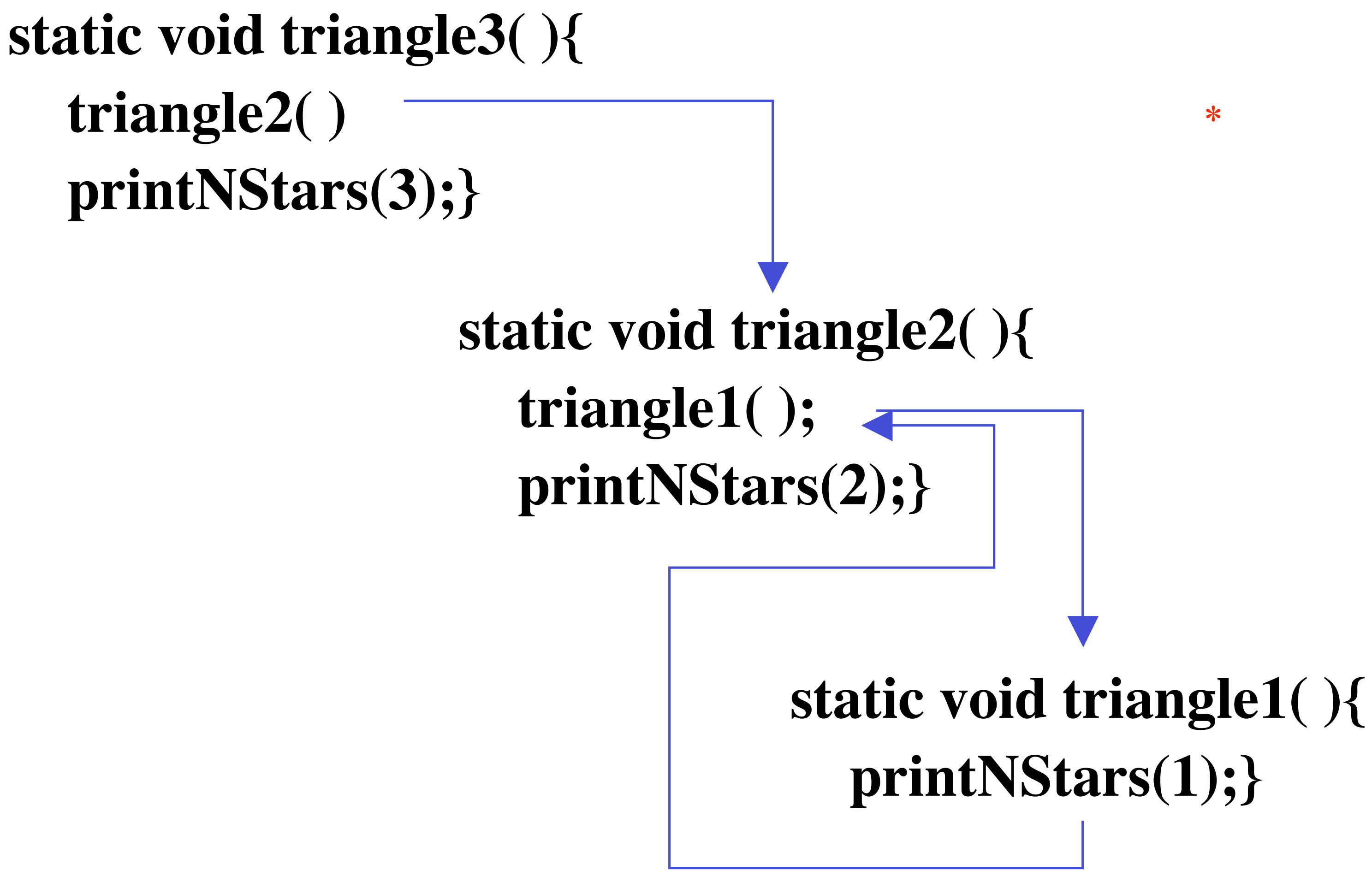
```
static void triangle3( ){
    triangle2( )
    printNStars(3);}

                              static void triangle2( ){
                                  triangle1( );
                                  printNStars(2);}

                                                      static void triangle1( ){
                                                          printNStars(1);}
```

*

How Does Recursion Work?



```
static void triangle3( ){
    triangle2( )
    printNStars(3);}
```

*

```
static void triangle2( ){
    triangle1( );
    printNStars(2);}
```

```
static void triangle1( ){
    printNStars(1);}
```

How Does Recursion Work?

```
static void triangle3( ){
    triangle2( )
    printNStars(3);}
```

```
*
**
```

```
static void triangle2( ){
    triangle1( );
    printNStars(2);}
```

How Does Recursion Work?

```
static void triangle3( ){
    triangle2( )
    printNStars(3);}
```

```
*
**
***
```

How Does Recursion Work?

- **Recursive:**

```
static void triangle(int n){
  if (n==1){
     printNStars(1);
  } else {
     triangle(n-1);
     printNStars(n);}}
```

How Does Recursion Work?

n = 3

```
static void triangle(n ){
   if …
    triangle(n-1)
    printNStars(n);}
```

*

n = 2

```
static void triangle(n ){
   if …
    triangle(n-1 );
    printNStars(n);}
```

n = 1

```
static void triangle1(n ){
    if …
    printNStars(1);}
```

How Does Recursion Work?

n = 3

```
static void triangle(n ){
   if …
    triangle(n-1)
    printNStars(n);}
```

*

n = 2

```
static void triangle(n ){
   if …
    triangle(n-1 );
    printNStars(n);}
```

n = 1

```
static void triangle1(n ){
    if …
    printNStars(1);}
```

How Does Recursion Work?

n = 3

```
static void triangle(n ){
  if …
   triangle(n-1)
   printNStars(n);}
```

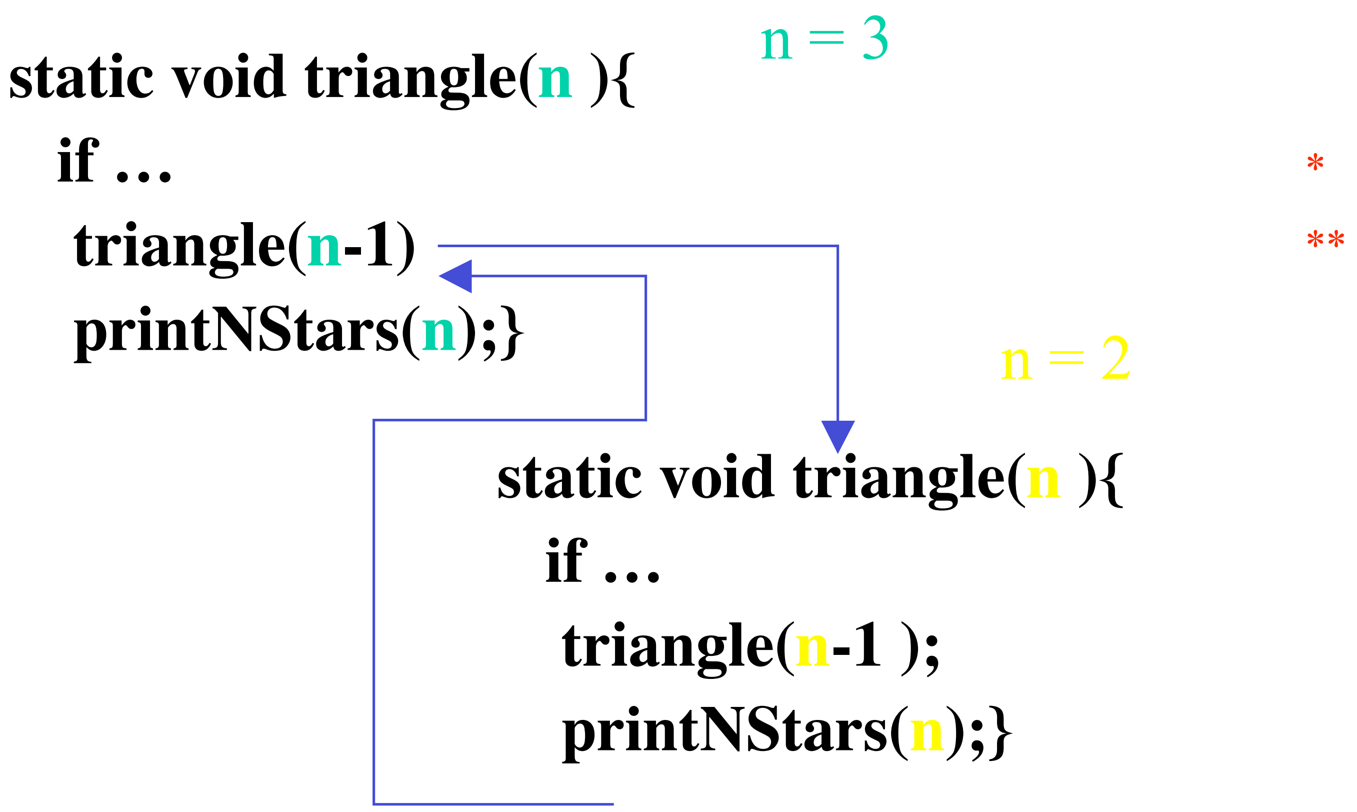
*

**

n = 2

```
static void triangle(n ){
  if …
   triangle(n-1 );
   printNStars(n);}
```

How Does Recursion Work?



```
static void triangle(n ){
   if …
    triangle(n-1)
    printNStars(n);}
```

```
*
**
```

```
static void triangle(n ){
   if …
    triangle(n-1 );
    printNStars(n);}
```

How Does Recursion Work?

n = 3

```
static void triangle(n ){
  if …
   triangle(n-1)
   printNStars(n);}
```

```
*
**
***
```

Designing Recursive Methods

- Print triangle of *s

```
*
**
***
****
```

- Print a triangle size 4,
  - Can you see how solving a similar but smaller problem would help solve this one?

Designing Recursive Methods

- **Print triangle of *s**

```
*
**
***
****
```

- **To print triangle size 4,**
  - **print a triangle of size 3**
  - **print 4 stars**

Ruler Pattern

```
*
**
*
***
*
**
*
****
*
**
*
***
*
**
*
```

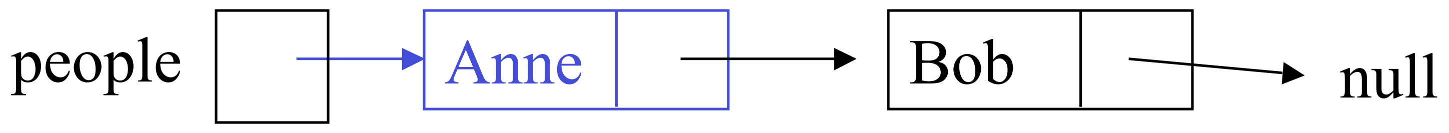
Ruler Pattern

```
*
**
*
***
*
**
*
****
*
**
*
***
*
**
*
```

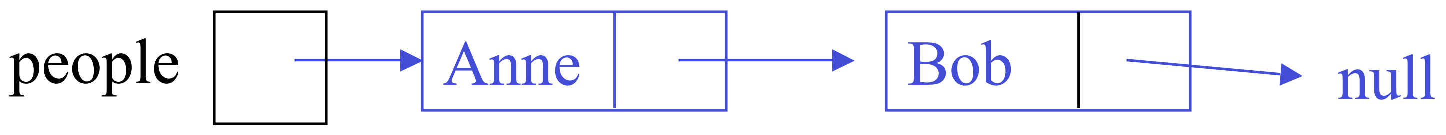
- Smaller problem appears twice!
- To do ruler n:
  - do ruler n-1
  - print` n *s
  - do ruler n-1

“Recursive” Data Types

- We can look at a reference to a node in two ways
  - It refers to a specific node

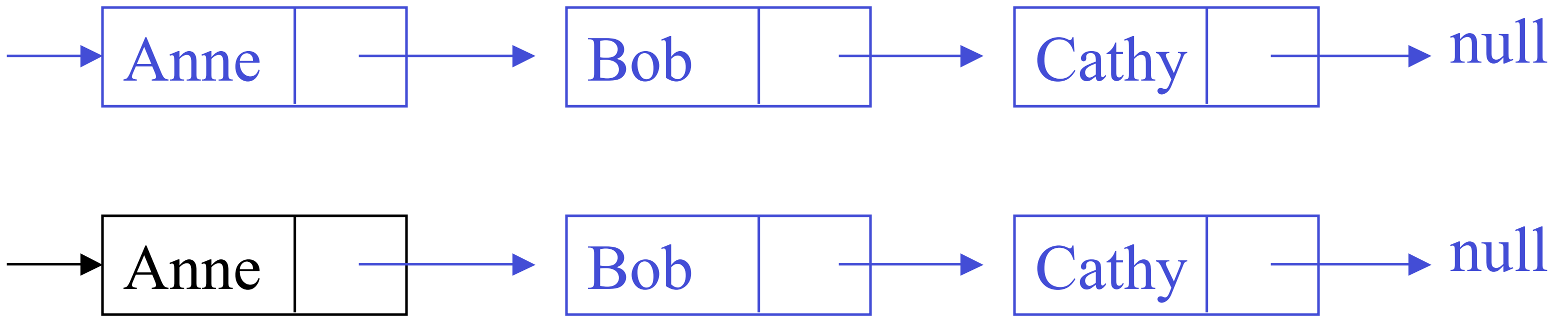


  - It refers to the entire list that the node starts



“Recursive” Data Types

- If a reference to a node means the whole list, then the next field of that node is “the rest” of the list.

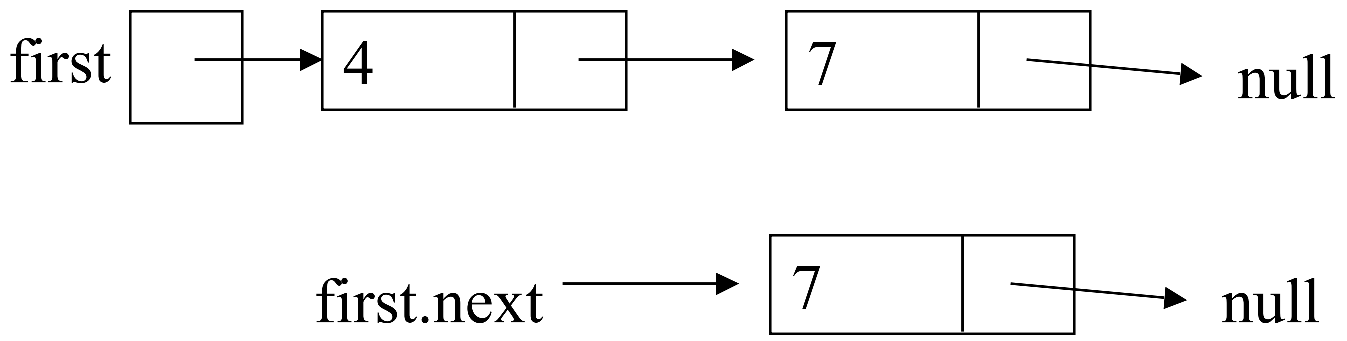


- “Next is the first node of the rest of your list.”

“Recursive” Data Types

- **See RecNode2.java**

NodeToString

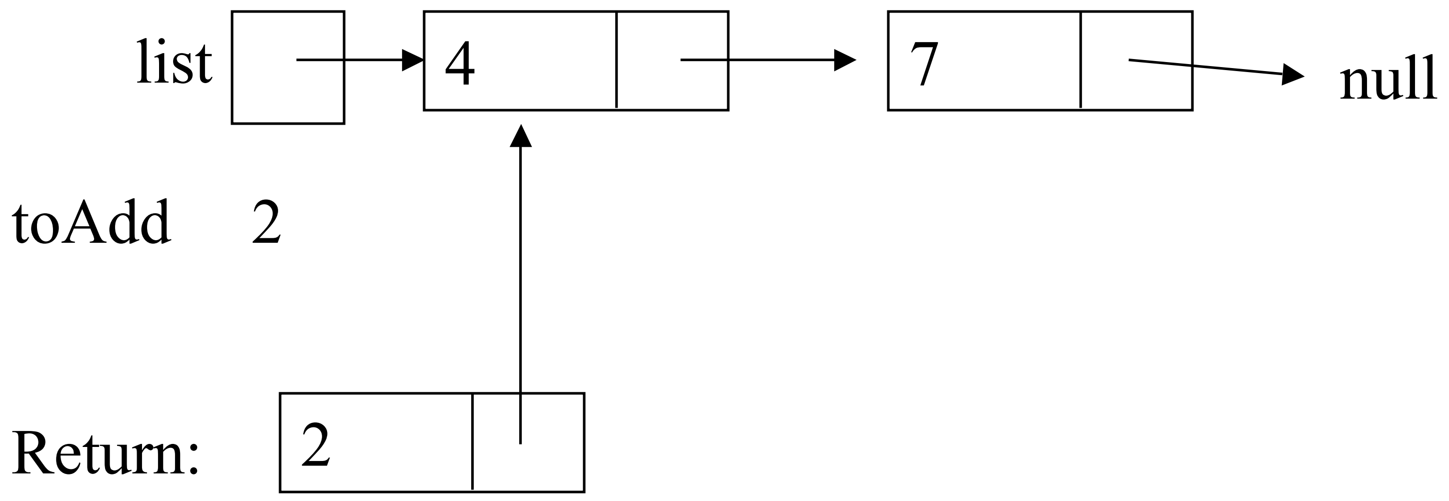


nodeToString(first.next) is “7 -> [end]”

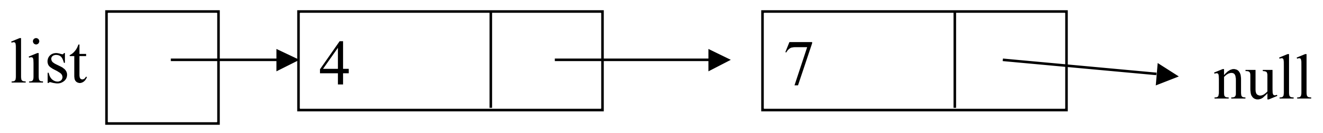
first.data is 4

nodeToString(first) returns “4 -> 7 ->[end]”

InsertInOrder

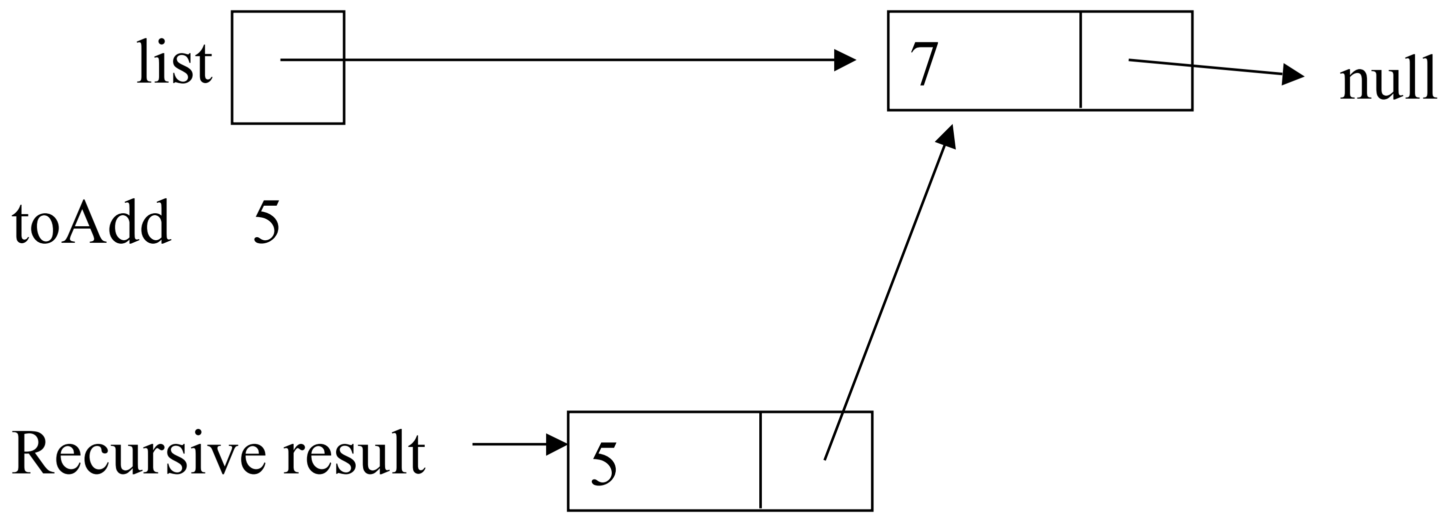


InsertInOrder

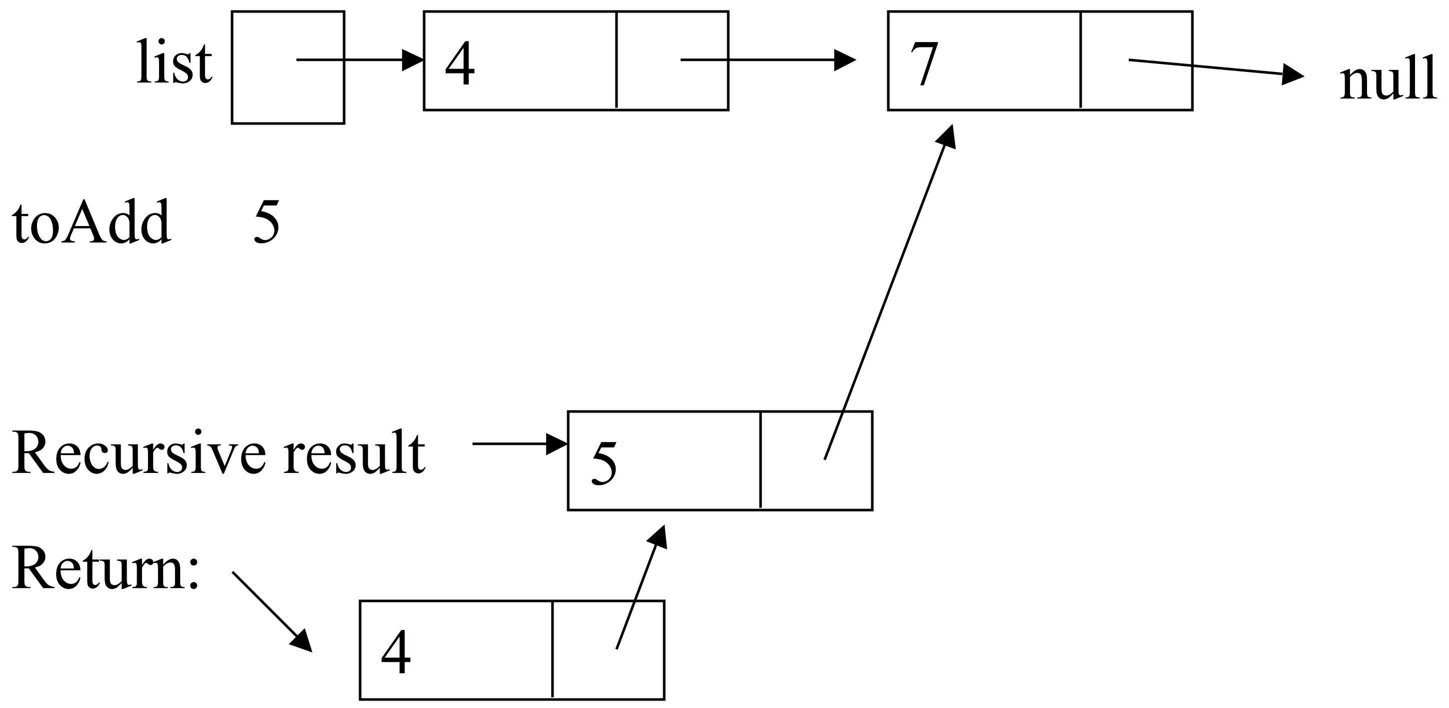


toAdd 5

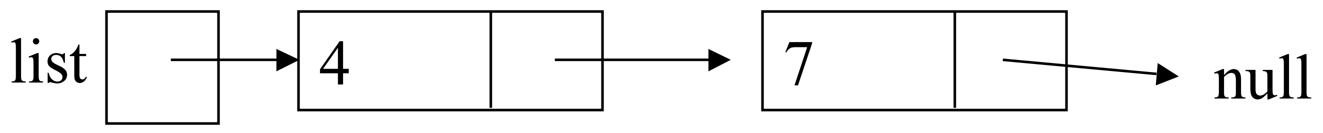
Recursive call



InsertInOrder

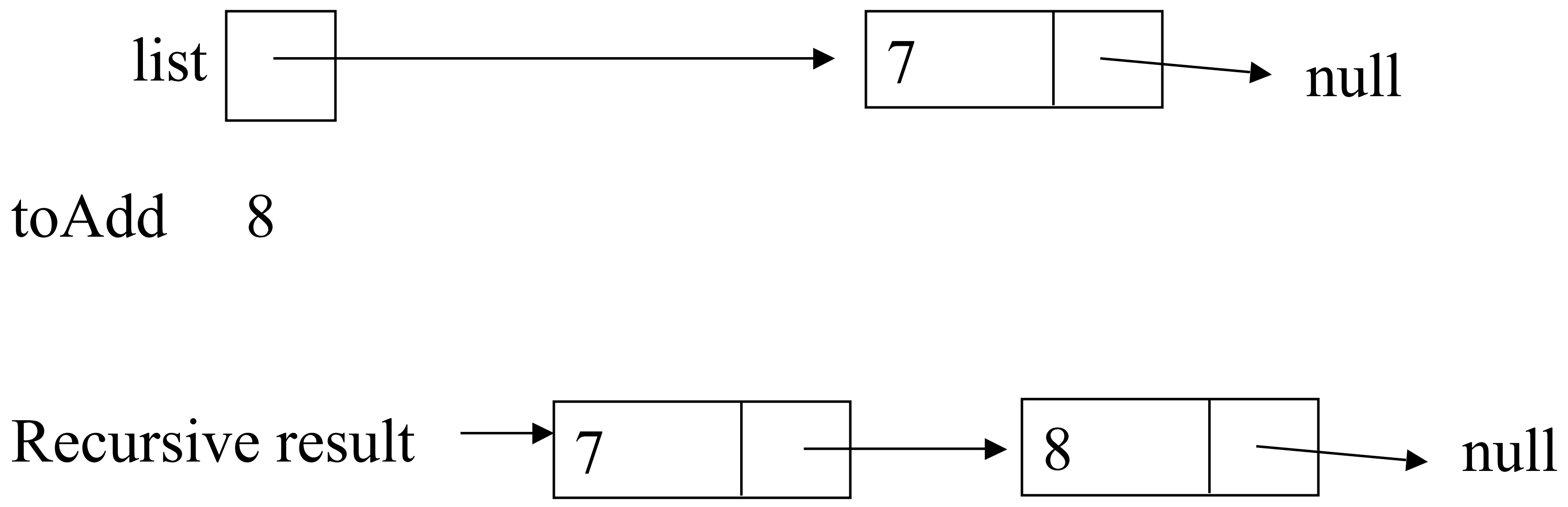


InsertInOrder

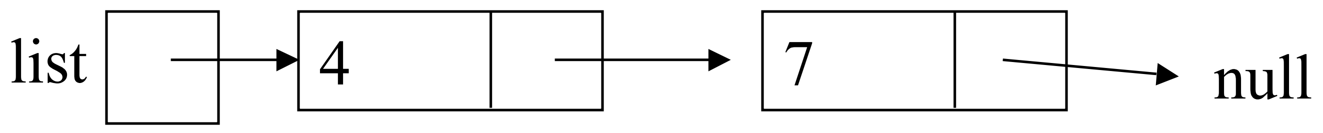


toAdd 8

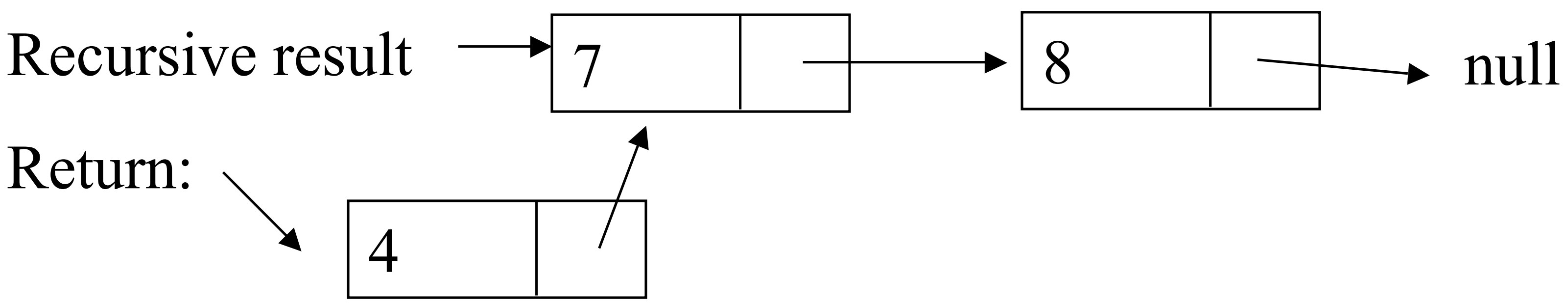
Recursive call



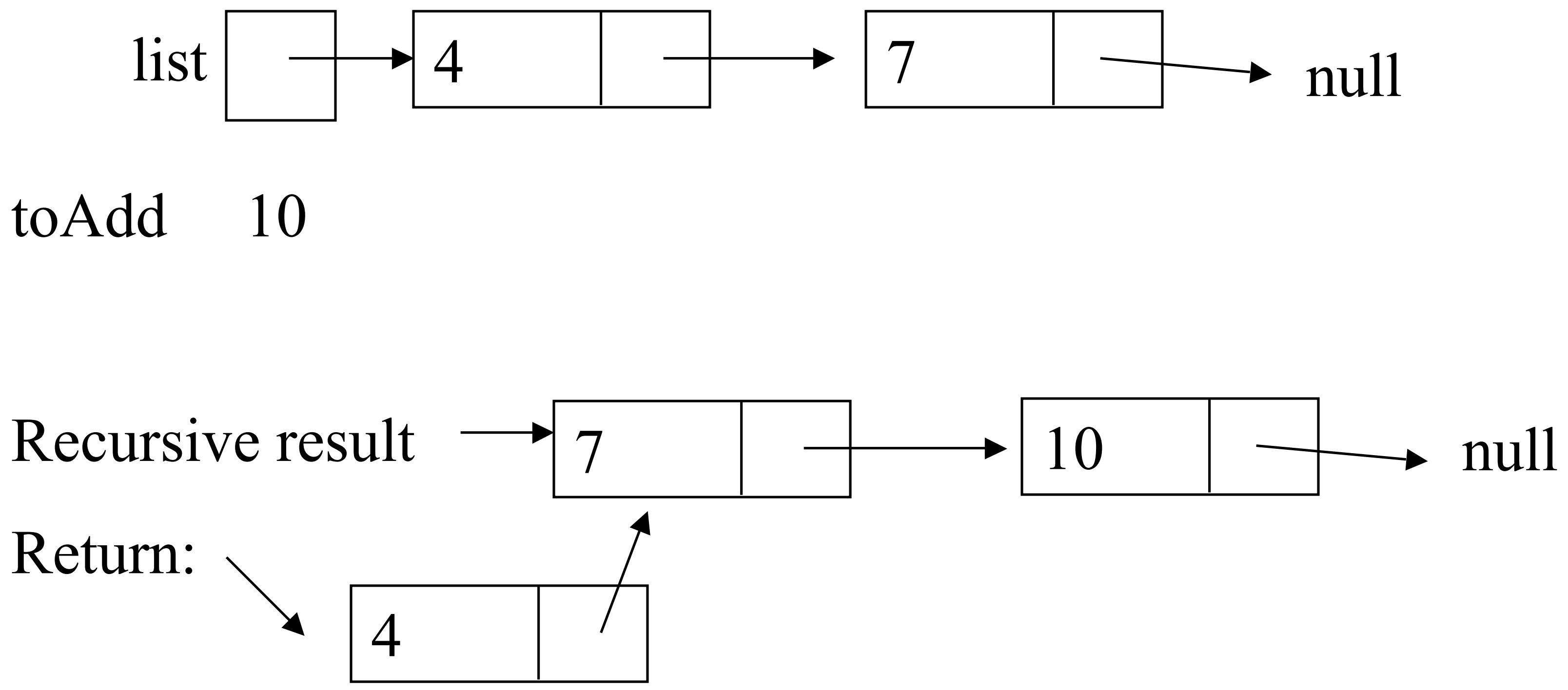
InsertInOrder



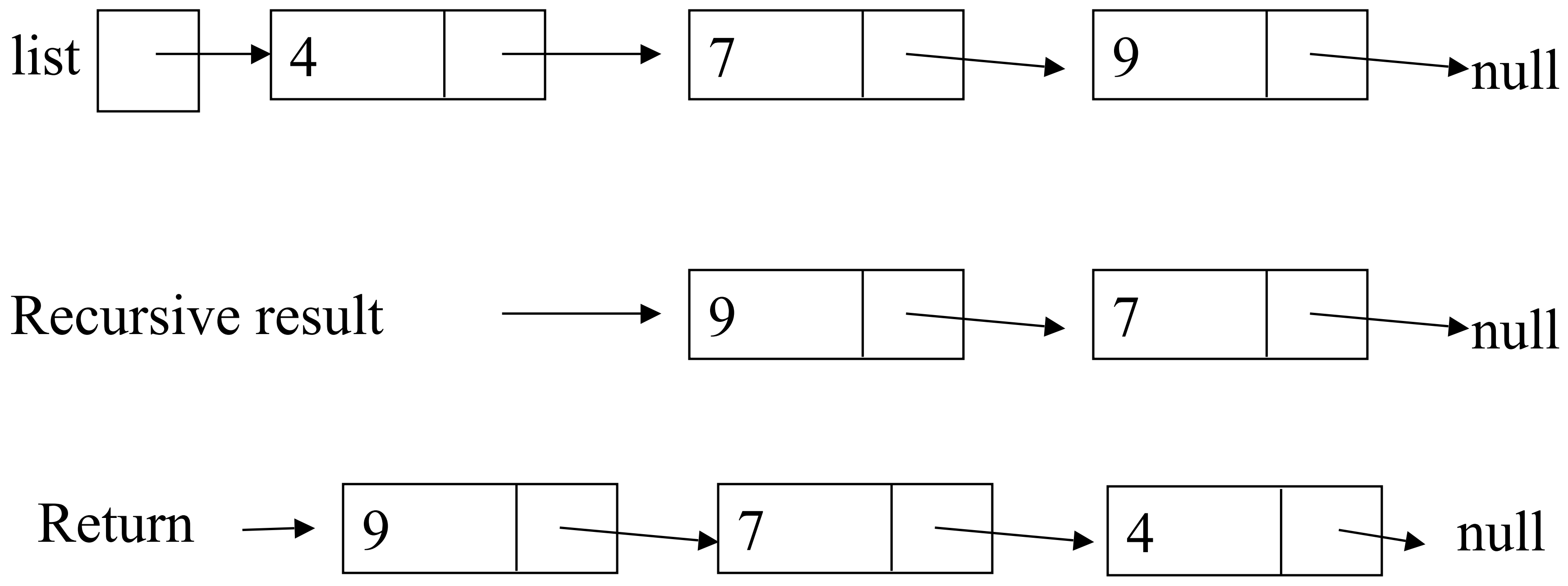
toAdd 8



AddAtTail



Reverse



WithoutAll

- You draw the pictures