CMPE-013/L

Introduction to "C" Programming

Max Lichtenstein

Piazza Poll: What does it print?

```
char recursivePrint(char * strToPrint){
    //test for base case:
    if (*strToPrint == NULL){
        return *(strToPrint);
    } else {
        printf("%c", recursivePrint(strToPrint+1));
        return *(strToPrint);
    }
}

int main(){
    recursivePrint("Hello World!");
}
```

- A: "Hello World!"
- B: "ello World!0"
- C: "!dlroW olleH"
- D: Something else
- E: Runtime error

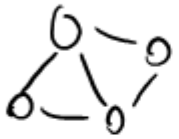
Roadmap

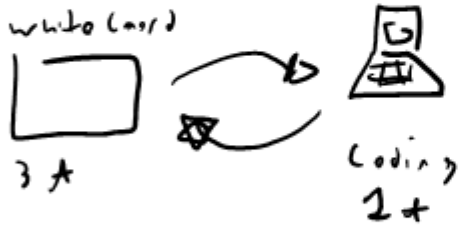
- Quiz
- Announcements
- Lab 07 debrief
- Binary Trees
- Recursion
- BREAK
- More recursion:
  - Examples
  - Binary tree example
- Lab 08 overview

Max Lichtenstein

Announcements

Lab 8 is up





Tough!

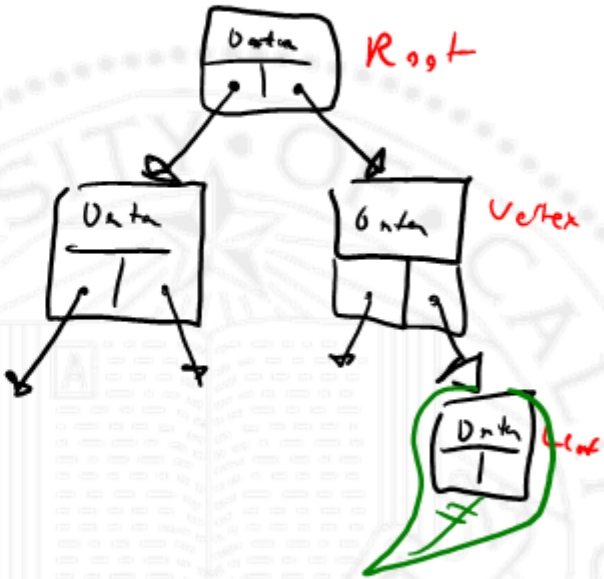
Paper work
drawing, pseudocode,
white board time

Partners for Lab 9
- form later today - pizza
- solo

Binary Trees

Binary Trees

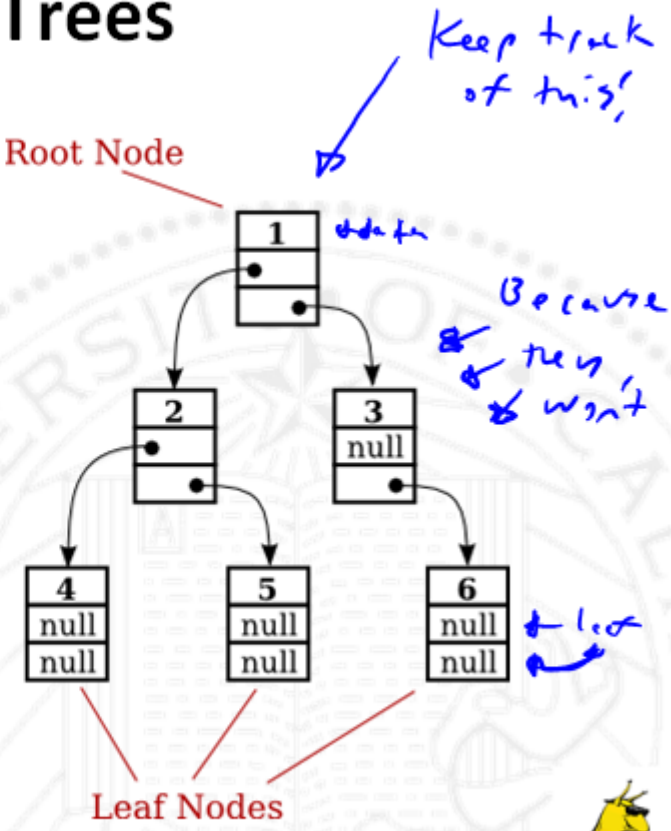
- Abstract data type
- A collection of "nodes"
  - Each node points to (up to) two children
    - Leaf nodes have no children
    - Root node has no parent
  - Each node has some data associated with it
  - Kind of like a linked list, but more structure



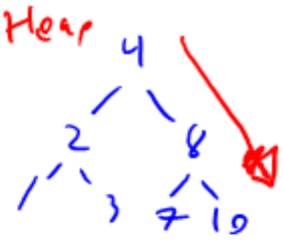
Binary Trees

```
typedef struct Node {
    struct Node *leftChild;
    struct Node *rightChild;
    char data;
} Node;
```

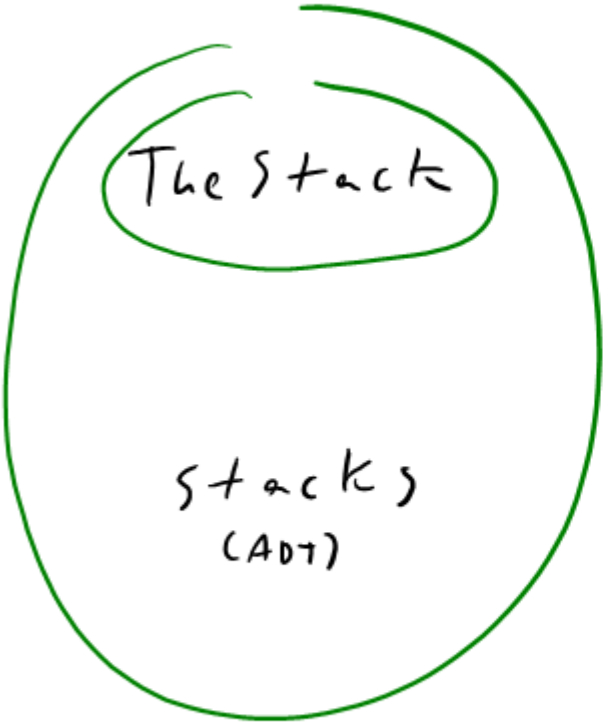
Not a pointer



What are they good for?



- Taking advantage of structure in data to search rapidly
  - Decoding / lookups (dict, decoding tree)
  - Finding maximum/minimum values
- Maintaining structure in data (to search rapidly)
  - Heaps (not The Heap!) ✓
  - Buddy allocation ✓
  - MANY more examples


The Stack
stacks
(ADT)


The Heap
malloc/free


Heaps
(ADT)

Binary tree use case:

X	bestIt(x)
100	99
7	2

- Objective: Given X, find the LARGEST value in this list that is LESS THAN X:
  - 103, 55, 200, 1, 90, 111, 300, 0, 2, 77, 99, 100, 114, 250, 999
- Slow way: Simple linear search
  - Requires N comparisons

Binary tree use case (2):

- Objective: Given X, find the LARGEST value in this list that is LESS THAN X:

X = 88

- Slightly less slow way: Sorted list:
  - 0, 1, 2, 55, 77, 90, 99, 103, 100, 111, 114, 200, 250, 300, 999
  - Requires ~N/2 comparisons

Binary tree use case:

- Objective: Given X, find the LARGEST value in this list that is LESS THAN X:
  - 103, 55, 200, 1, 90, 111, 300, 0, 2, 77, 99, 100, 114, 250, 999

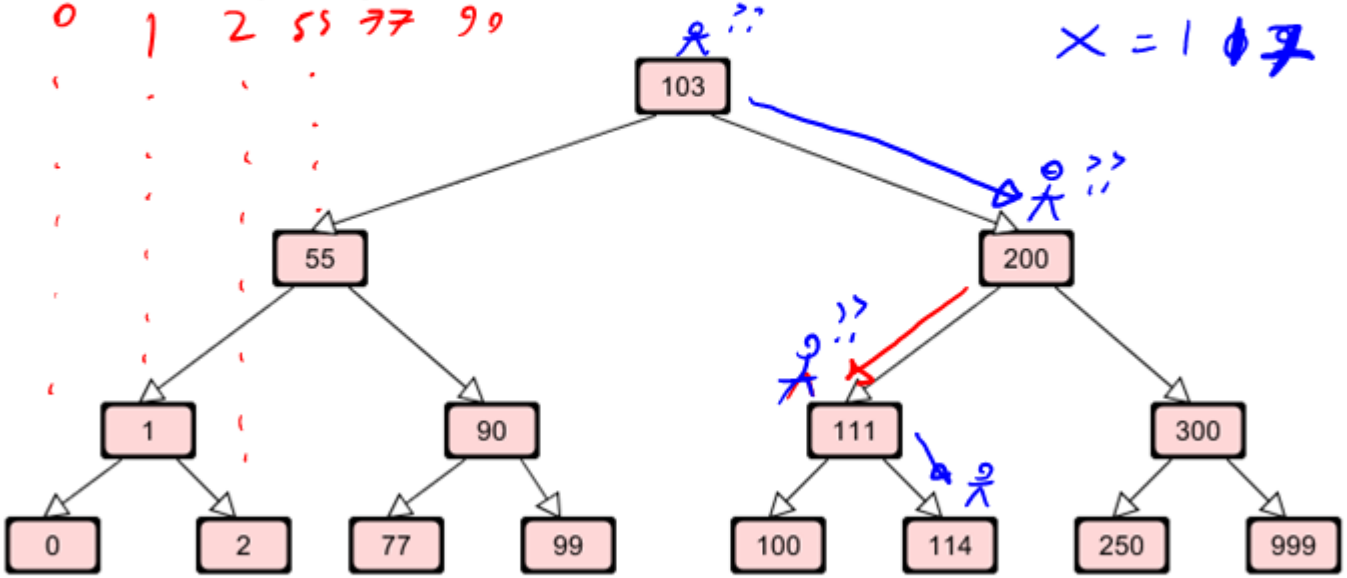
- Fast way: Binary tree (~$\log_2(N)$ comparisons)

4 for us

Binary tree use case:

Binary tree rule:
No node can have a value greater than any node to its right anywhere in the tree

Objective: given X, find the LARGEST value in the list that is LESS THAN X



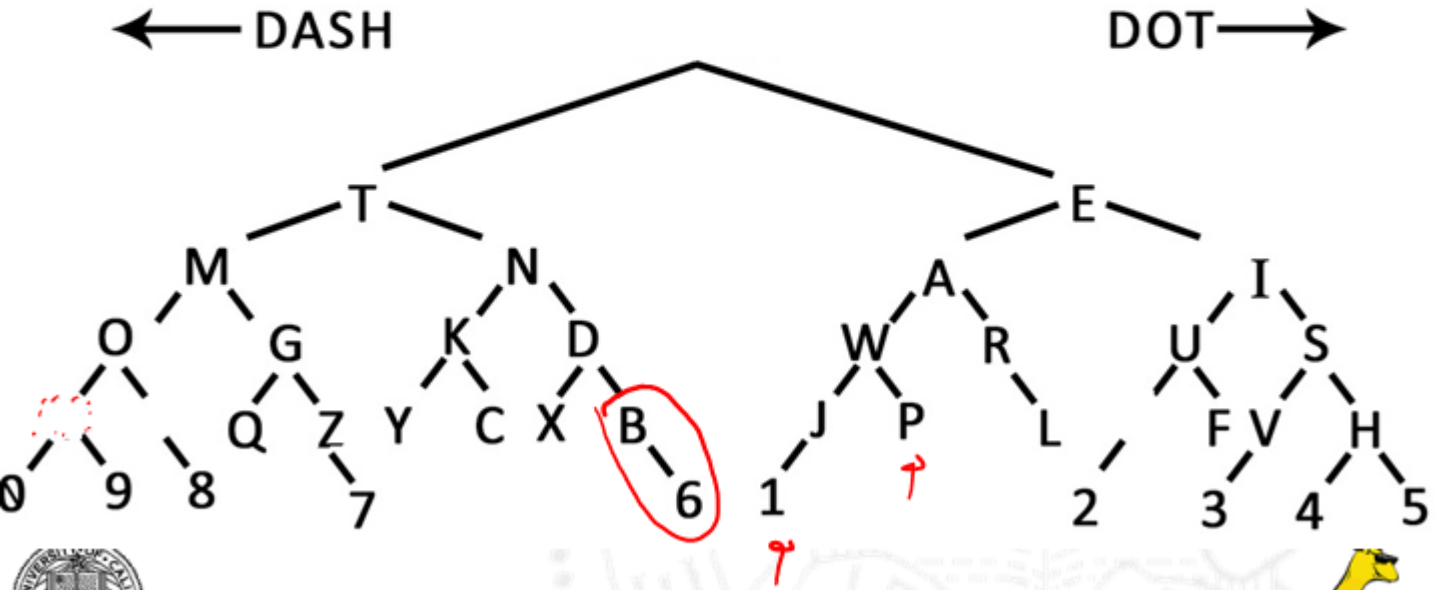
Max Lichtenstein

A few more notes on binary trees

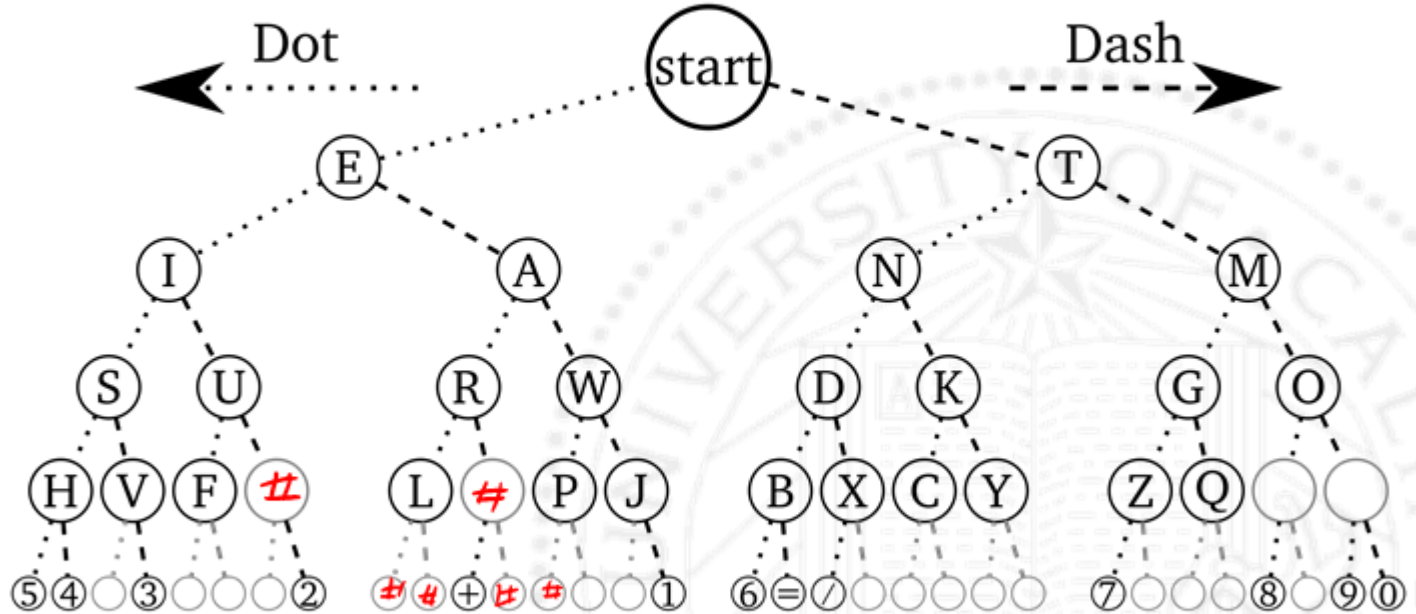
- Binary trees can be:
  - Balanced
    - Same distance to each leaf
  - Full (proper)
    - No nodes with 0 children
  - Complete
    - Each node has data

- Much of the work with binary trees goes into *maintaining* them
  - But we don't do that in this class!

The Morse Tree
(incomplete, unbalanced, improper)



The Morse Tree
(balanced, proper, still incomplete)



functions that call themselves

Linear Homogenous Recurrence Relations

Recursion

Recurrence + proof by Induction

1) a base case
   - simple, easily resolved
2) Mechanism to reduce other cases to base case

Recursion

- Solving problems by breaking them into smaller parts
- "divide and conquer"
- Relies on the problem having self-similarity

Example

```
int Factorial(int n)
{
  if (n <= 1) {
    return 1;
  }
  return n * Factorial(n - 1);
}
```

factorial:

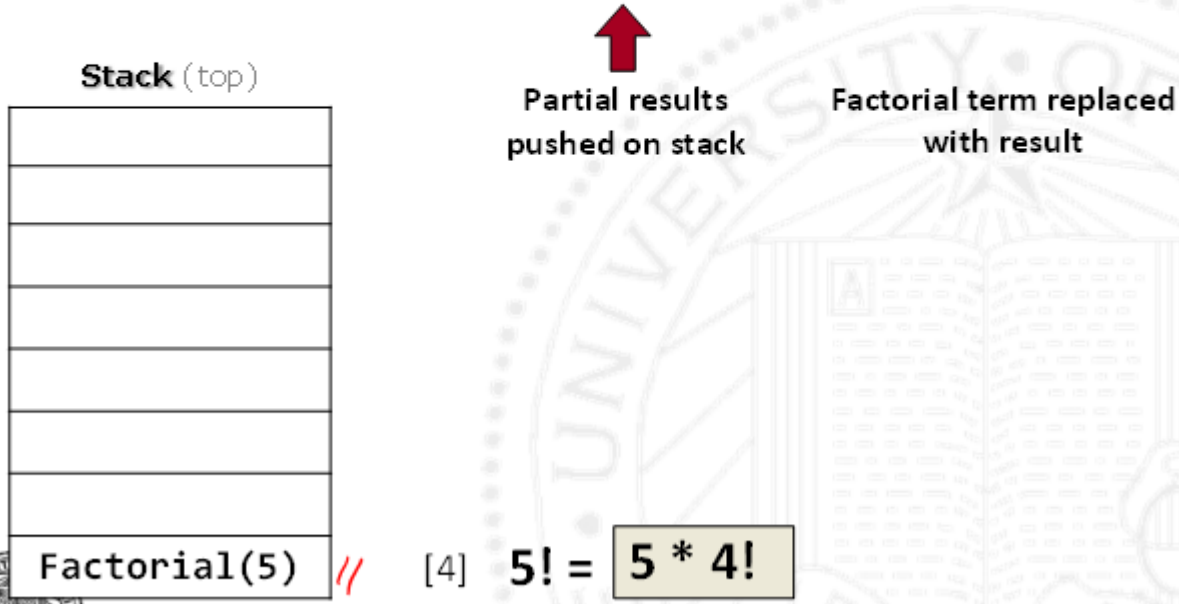
x	$x!$
1	1
2	$2 \cdot 1 = 2$
3	$3 \cdot 2 \cdot 1$
4	$4 \cdot 3 \cdot 2 \cdot 1 = 24$
4	$4 \cdot (3 \cdot 2 \cdot 1)$
	$4 \cdot (3!)$
	$4 \cdot (3 \cdot (2 \cdot 1))$
	$4 \cdot 3 (2!)$

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

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Factorial term replaced with result

[3] 4! = 4 * 3!

[4] 5! = 5 * 4!

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Factorial term replaced with result

[2] 3! = 3 * 2!

[3] 4! = 4 * 3!

[4] 5! = 5 * 4!

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Factorial term replaced with result

[1] 2! = 2 * 1!

[2] 3! = 3 * 2!

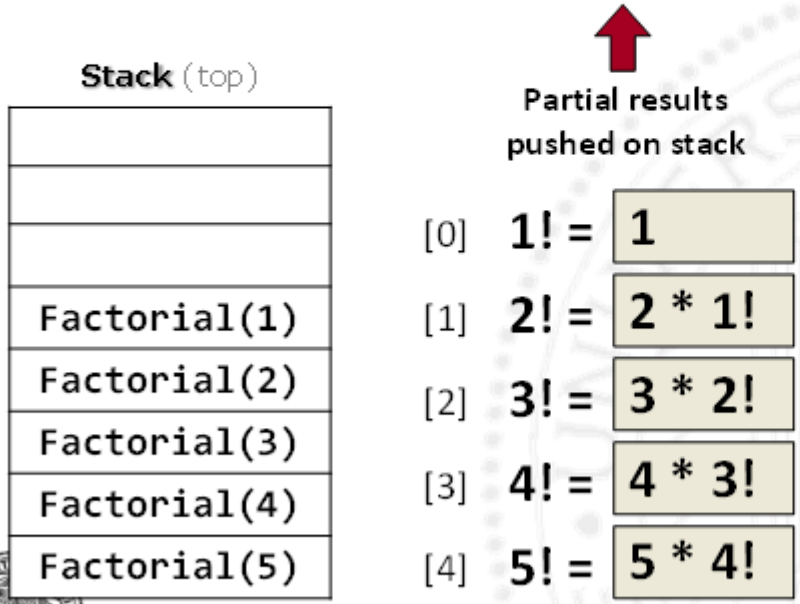
[3] 4! = 4 * 3!

[4] 5! = 5 * 4!

Recursion

Evaluation of Recursive Functions

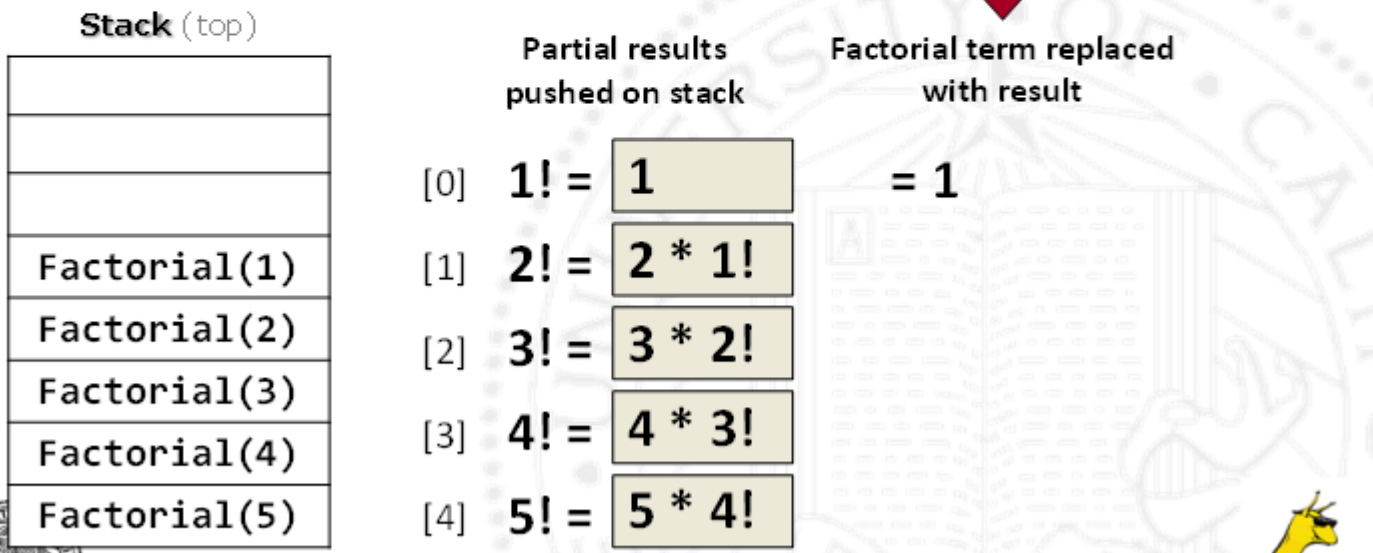
- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

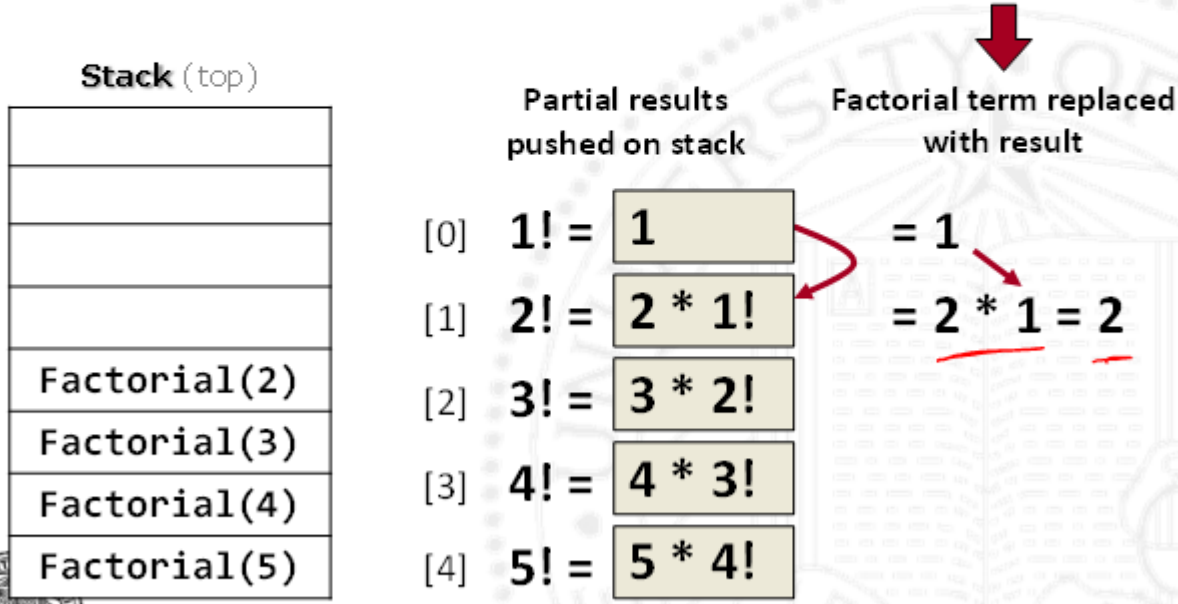
- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

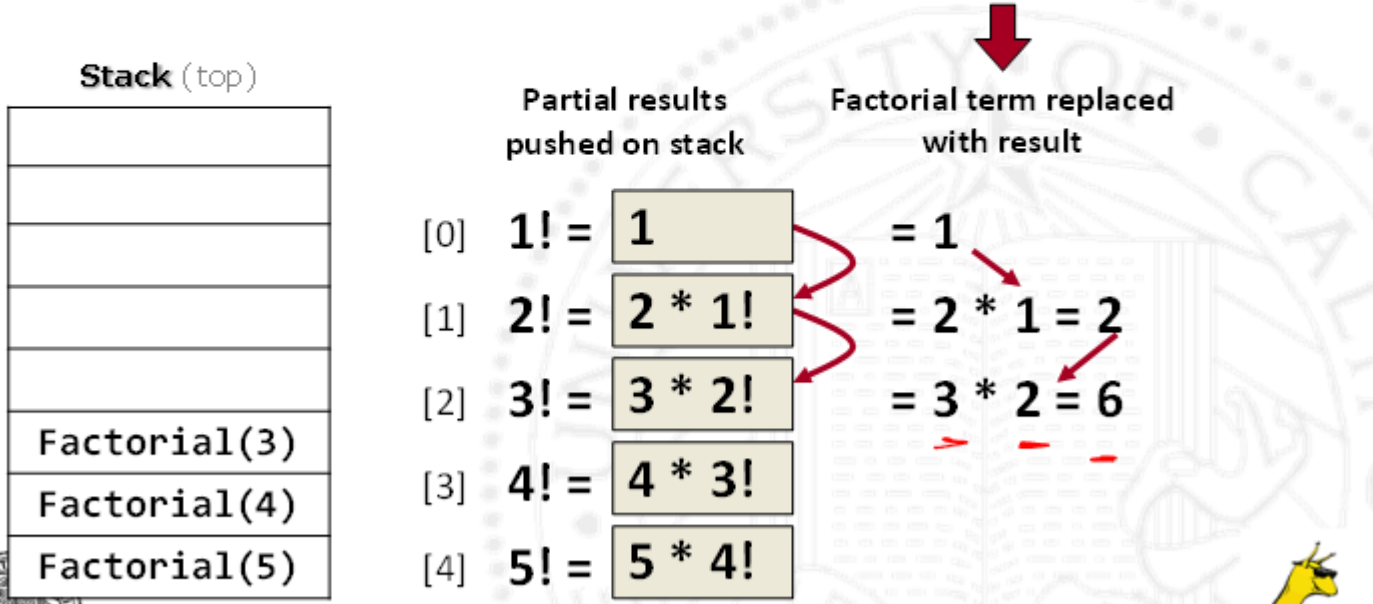
- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

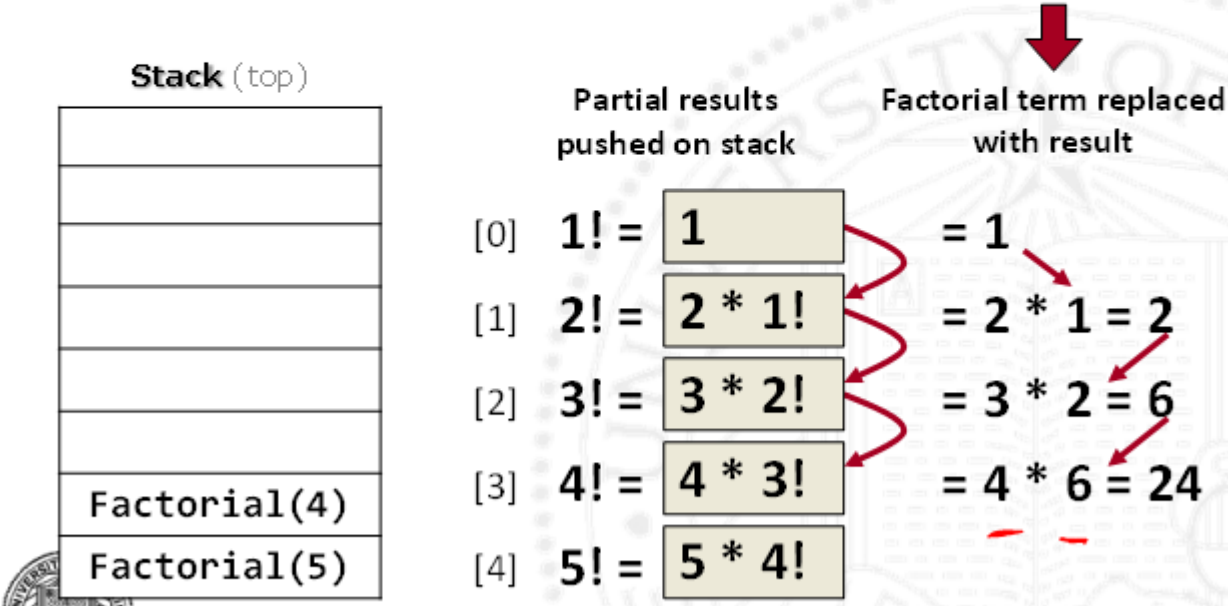
- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

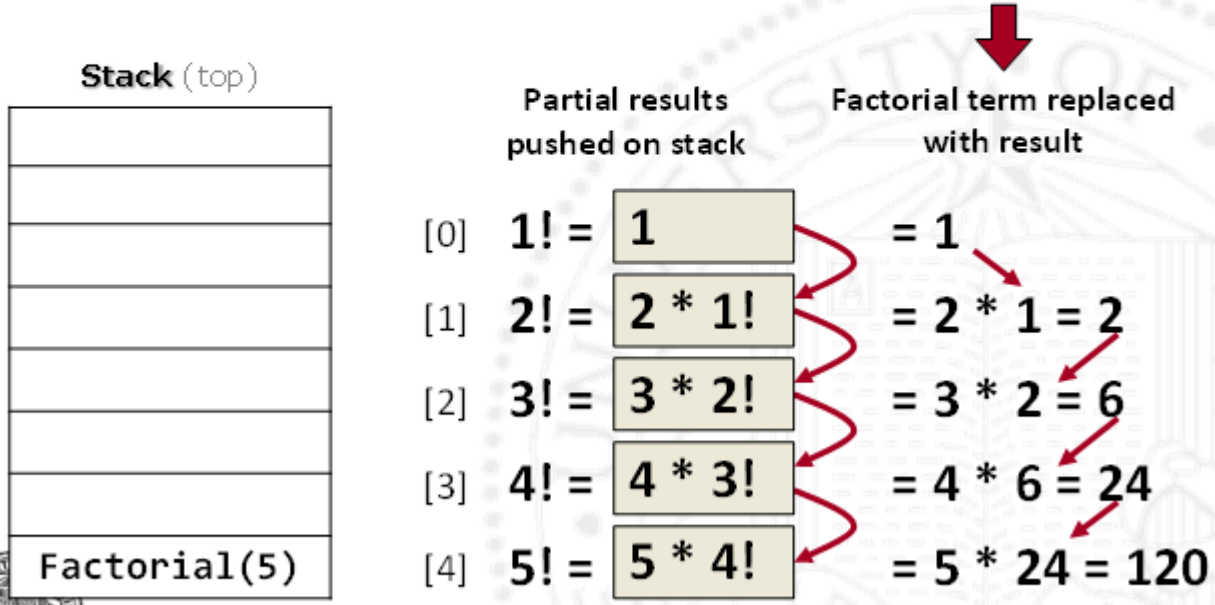
- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Recursion

Summary

- Usable for solving problems that are divided into subproblems
  - Divide and conquer
- Initial conditions must be similar to conditions for any of the subproblems
  - No difference between solving the smaller computation stand-alone versus as part of a larger computation
- Requires well-defined termination condition

Recusion

Caveats

- Problem must have a well-defined termination condition/base case
- Must have enough memory
  - Memory use high from filling the function stack

Stack overflow

Is recursion fast?

Is it small?

NO. NO.

not faster than an imperative solution (loops, etc)

```
fact(x):
  int i, result = 1;
  for (i = 1; i <= x; i++) {
      result = i * result;
  }
  ret result
```

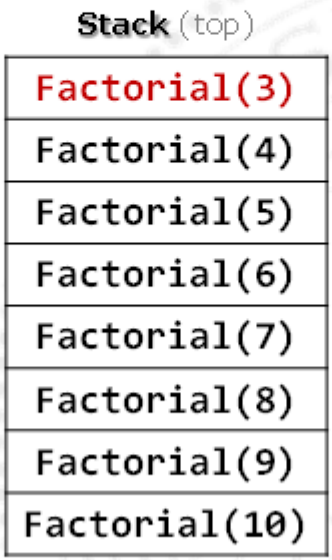
Why Recurse?

- Easy to read
- Easy to think about
- Elegant

Recursion

Limitations

- Limited stack space



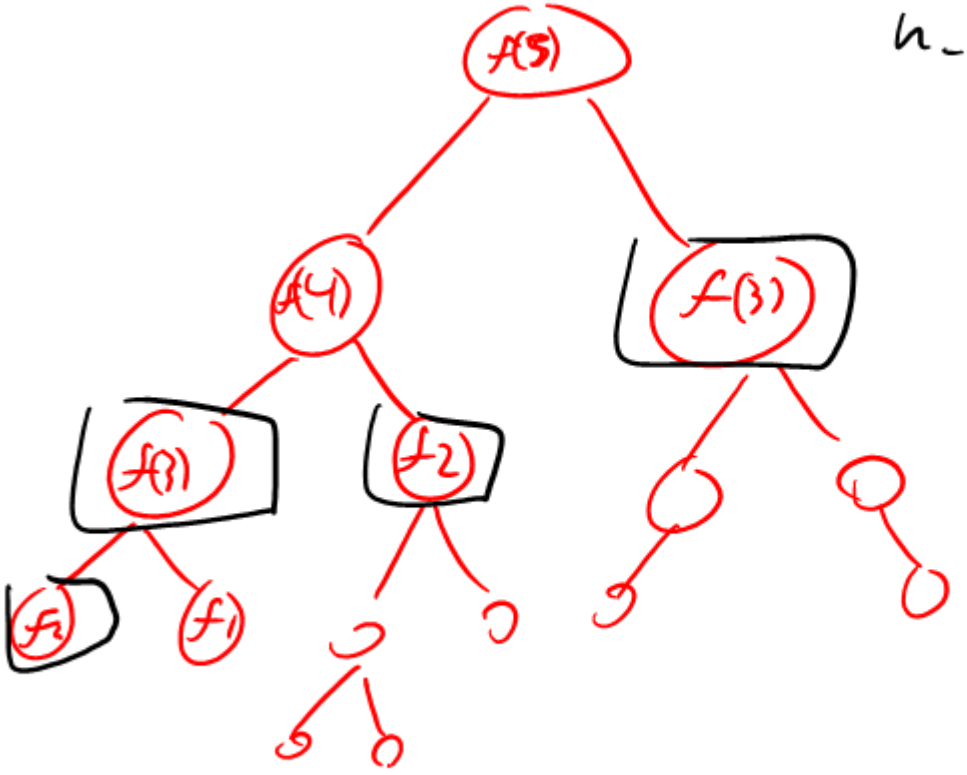
Recursion

Multiple recursion

- Recursion is not limited to a single function call

Example

```
int Fibonacci(int n)
{
  if (n <= 1) {
    return 1;
  }
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```



h_calls $\propto 2^x$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

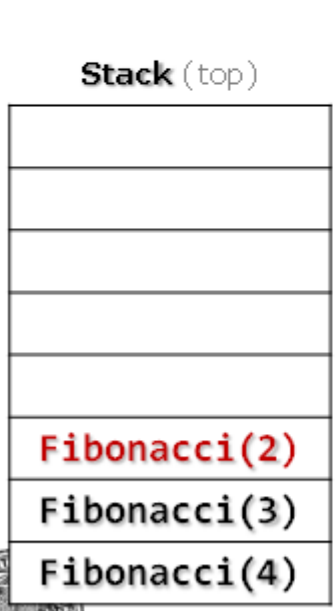
$F_3 = F_2 + F_1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_2 = F_1 + F_0$

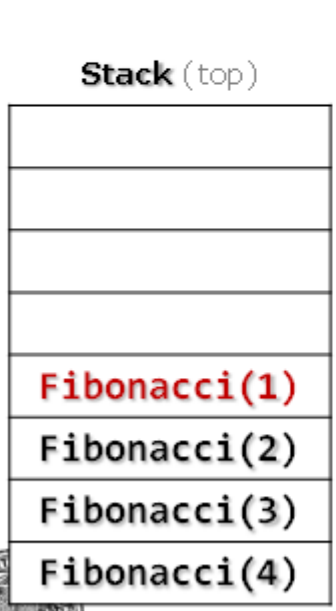
$F_3 = F_2 + F_1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_1 = 1$

$F_2 = F_1 + F_0$

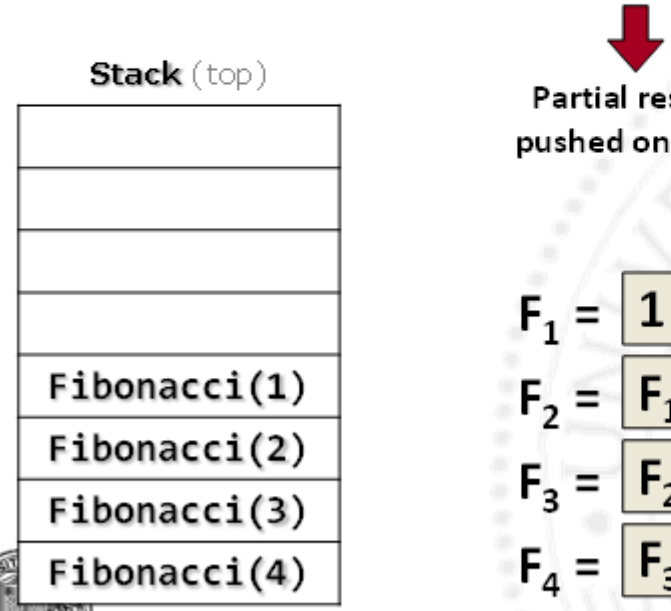
$F_3 = F_2 + F_1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

$F_2 = 1 + F_0$

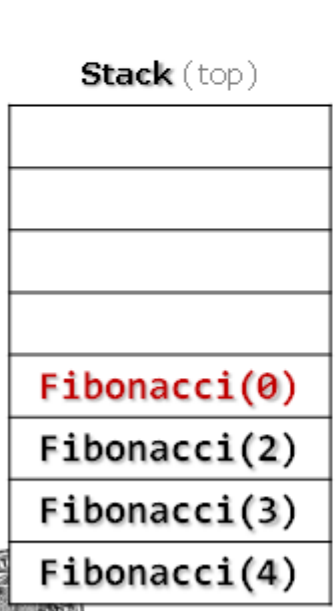
$F_3 = F_2 + F_1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_0 = 1$

$F_2 = 1 + F_0$

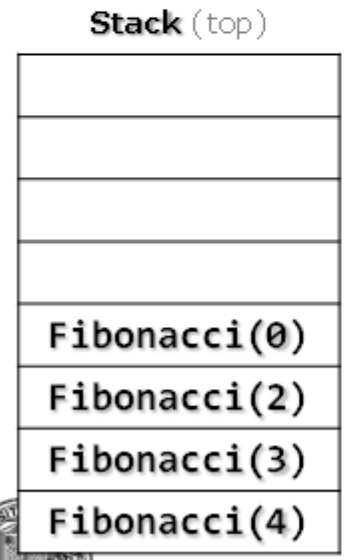
$F_3 = F_2 + F_1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_0 = 1$

$F_2 = 1 + F_0$

$F_3 = F_2 + F_1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

$F_2 = 1 + 1$

$F_3 = F_2 + F_1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

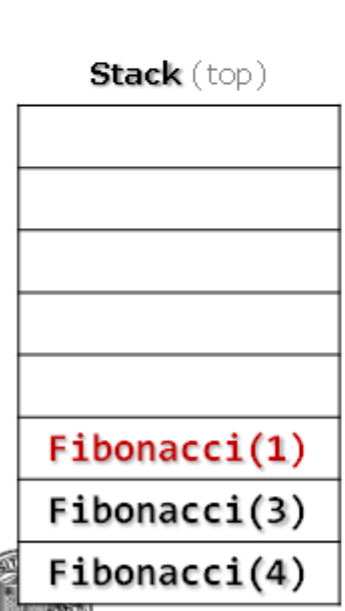
$F_3 = \boxed{2 + F_1}$

$F_4 = \boxed{F_3 + F_2}$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_1 =$ 1

$F_3 =$ 2 + F_1

$F_4 =$ $F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

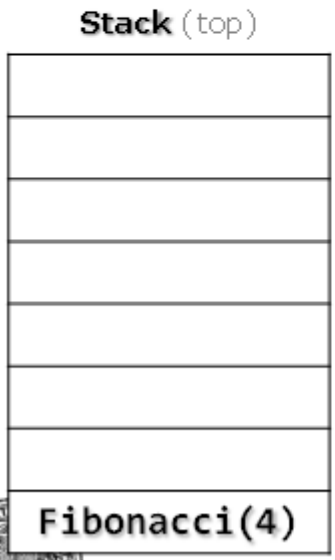
$F_3 = 2 + 1$

$F_4 = F_3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

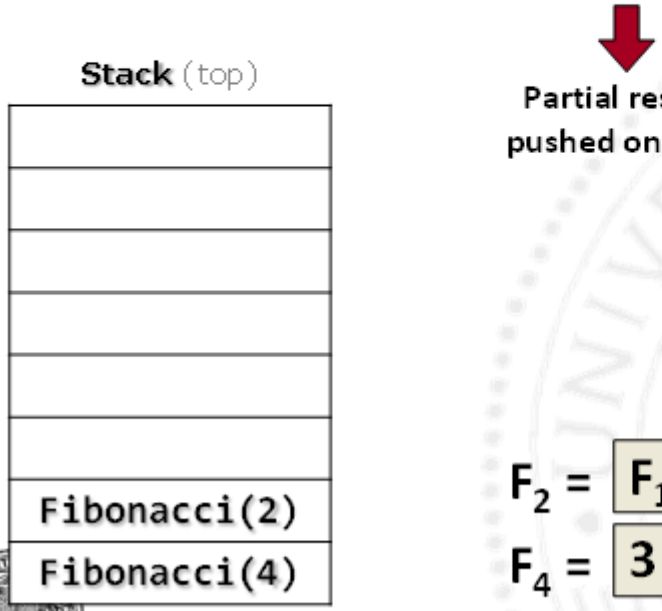
Function call replaced with result

$F_4 = 3 + F_2$

Recursion

Evaluation of Recursive Functions

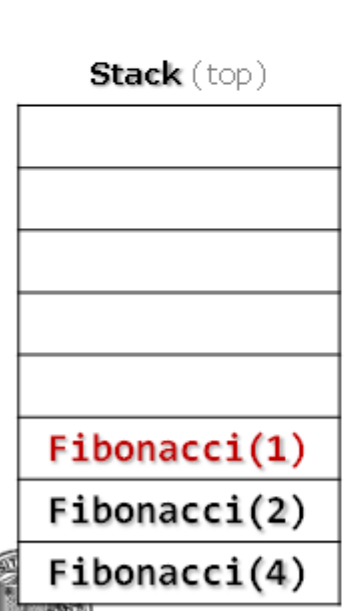
- Evaluation of 5!
  (based on code from previous slide)



Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_1 =$ 1

$F_2 =$ $F_1 + F_0$

$F_4 =$ $3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

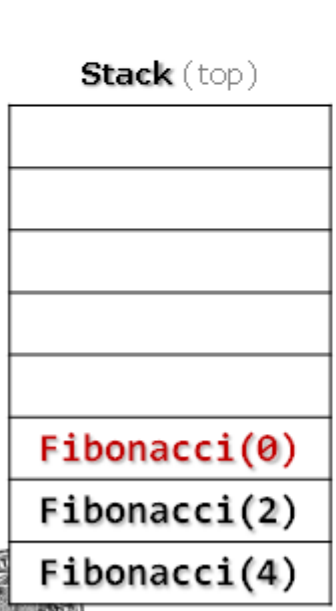
$F_2 = 1 + F_0$

$F_4 = 3 + F_2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_0 =$ 1

$F_2 =$ 1 + F_0

$F_4 =$ 3 + F_2

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)

Partial results pushed on stack

Function call replaced with result

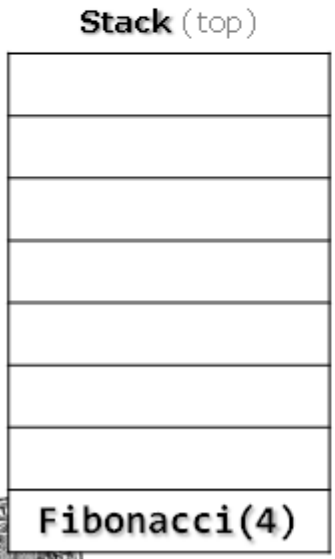
$F_2 =$ 1 + 1

$F_4 =$ 3 + F_2

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

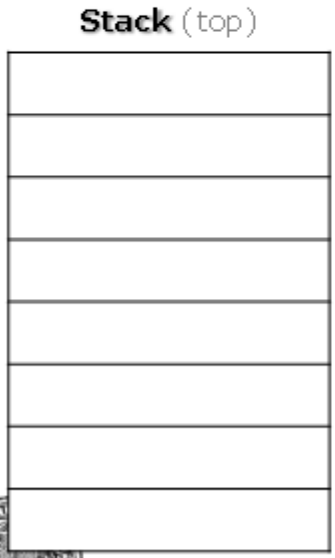
Function call replaced with result

$F_4 = 3 + 2$

Recursion

Evaluation of Recursive Functions

- Evaluation of 5!
  (based on code from previous slide)



Partial results pushed on stack

Function call replaced with result

$F_4 = 5$

Recursion

Self-similarity

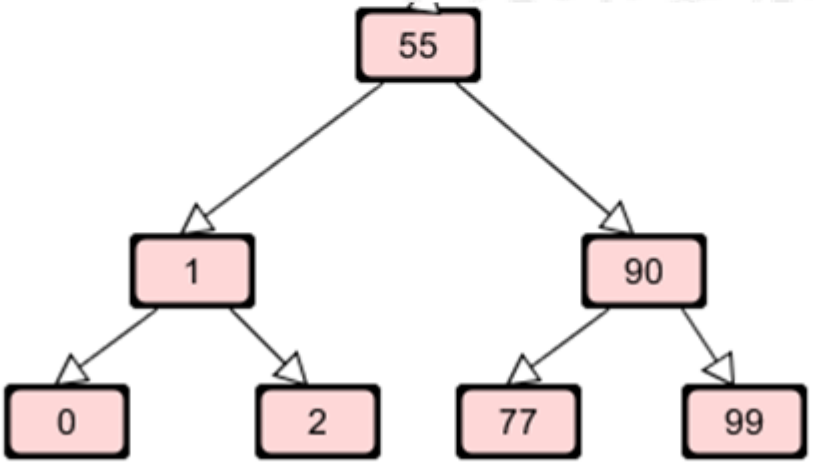
- A structure that is similar to part of itself
  - Example: fractals
- Computation **and** data must be self-similar for recursion
- Previous examples only dealt with single integers
- But what about more complicated data?

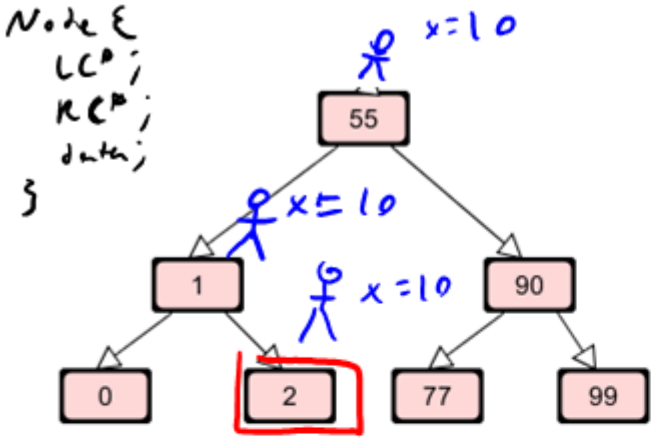
BREAK

Max Lichtenstein

Recursion on a binary tree

Objective:
given X, find largest value in tree that is less than X





Algorithm design:

Top level call

bestIt(root, x);

int bestIt(Node* node, int x)

is x > data?

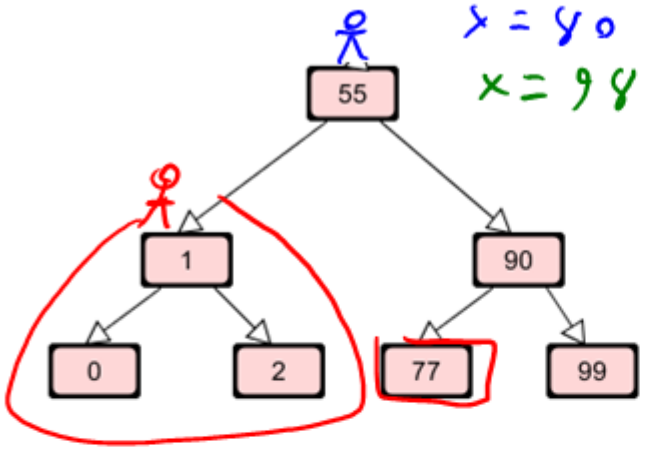
No:

Left

Yes:

Right

Max Lichtenstein



x = 90

x = 99

Pseudocode:

```
best_lt(Node *node, x):
    if x >= data:
        return best_lt(leftchild, x)
    if x < data:
        x = best_lt(rightchild, x)
        return max(x, data)
```

```
base case:
    if LC = NULL:
        return data
```

NOT QUITE RIGHT

Lab 08 overview

Lab08

- BinaryTree.h
- Morse.h:
  - MorseInit/MorseDecode
  - MorseCheckEvents
- Lab08_main.c
  - Run MorseCheckEvents
  - Feed results to MorseDecode and print services
    - Feed *those* results to print services