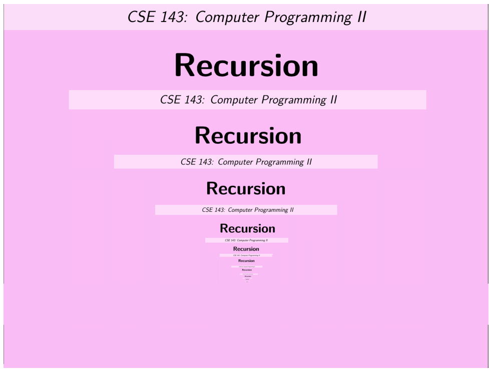
Adam Blank | Lecture 9 | Winter 2015

CSE 143

Computer Programming II

CSE 143: Computer Programming II

Recursion


CSE 143: Computer Programming II
Recursion
CSE 143: Computer Programming II
Recursion
CSE 143: Computer Programming II
Recursion
CSE 143: Computer Programming II
Recursion
Recursion

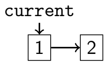
Outline

1. What is Recursion? Why Bother Learning it?
2. Connecting recursion with things we already know
3. A Recursive Demonstration
4. How To Think About Recursion

Questions From Last Time

- Why does the order matter for null checking tests?

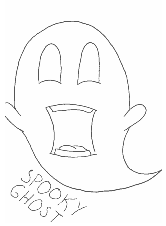
```java
while (current.next.data < value && current.next != null) {
   ...
}
```

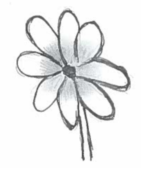


Suppose our linked list is . Then, current.next.data = 2. When we move current over, we get current.next = null. When we try to say current.next.data, we get a NullPointerException. If we had done the current.next != null check first, we would have broken out of the loop instead of getting a NullPointerException.

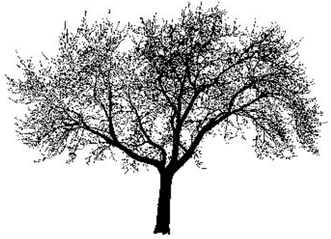
- Do you have to reset this.front if you assign it to something else? Unless you're inserting at the front of the list, you should **never** edit this.front

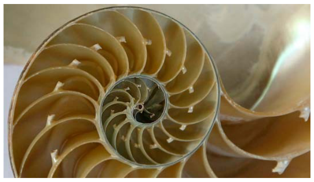
Drawings





Recursion in Nature





Recursive Structures

`LinkedList`s are **recursive structures**.

A `LinkedList` is. . .

a piece of data and a `LinkedList`, which is
    a piece of data and a `LinkedList`, which is
        a piece of data and a `LinkedList`, which is
            a piece of data and a `LinkedList`, which is
                a piece of data and a `LinkedList`, which is
                    a piece of data and a `LinkedList`, which is. . .

A **recursive data structure** is one made up of smaller versions of the same data structure.

Recursion is. . . Recursion is. . .

Definition (Recursion)

Recursion is the definition of an operation **in terms of itself**.

To solve a problem with recursion, you break it down into smaller instances of the problem and solve those.

Definition (Recursive Programming)

Writing methods that call themselves to solve problems recursively

Some problems are **naturally recursive** which means they're easy to solve using recursion and much harder using loops.

Why Learn Recursion?

- It's a different way of thinking about problems
- Recursion leads to **much** shorter code to solve difficult problems.
- Some programming languages **do not have loops**.
- Many data structures are defined recursively, and recursion is the easiest way of dealing with those structures.

Evaluating Arithmetic Expressions

How do we evaluate the mathematical expression $((1*17)+(2*(3+(4*9))))$?

```
((1 * 17) + (2 * (3 + (4 * 9))))
 ------------------------------

((1 * 17) + (2 * (3 + (4 * 9))))
  ------     -----------------

((1 * 17) + (2 * (3 + (4 * 9))))
  -   --     -----------------

(   17    + (2 * (3 + (4 * 9))))
    --       -----------------

(   17    + (2 * (3 + (4 * 9))))
    --       -    -----------

(   17    + (2 * (3 + (4 * 9))))
    --       -    -    -----

(   17    + (2 * (3 + (4 * 9))))
    --       -    -    -   -

(   17    + (2 * (3 +   36   )))
    --       -    -     --

(   17    + (2 *      39      ))
    --       -        --

(   17    +         78         )
    --              --

                95
                --
```

Evaluating Expressions: It's Recursive!

Evaluation of a simple expression $(1*3)+(4+2)$ looks like:

To evaluate $(1*3)+(4+2)$, first evaluate $(1*3)$ and $(4+2)$:
    $(1*3)=3$
    $(4+2)=6$
So, $3+6=9$.

The big instance of the problem is

$$(1*3)+(4+2)$$

and the smaller instances are

$$(1*3) \text{ and } (4+2)$$

Psuedocode for `eval`

`eval` Algorithm

```
1 Find the outermost operation.
2 Figure out the left and right operands.
3 If left is not a number, eval it. Call the result $a$.
4 If right is not a number, eval it. Call the result $b$.
5 Return $a$ op $b$.
```

Running `eval`$((1*3)+(4+2))$

```
1 The outermost operation is $+$.
2 The left is $(1*3)$ and the right is $(4+2)$.
3 $(1*3)$ is not a number. So, evaluate it:
      1 The outermost operation is $*$.
      2 The left is $1$ and the right is $3$.
      3 $a=1$
      4 $b=3$
      5 $a*b=3$
  So, $a=3$.
4 $(4+2)$ is not a number. So, evaluate it:
      1 The outermost operation is $+$.
      2 The left is $4$ and the right is $2$.
      3 $a=4$
      4 $b=2$
      5 $a+b=6$
  So, $b=6$.
5 $a+b=9$
```

Making Change

Someone will ask you "Can you make change for N spirals?"

Instructions

1. If you were asked "Can you make change for 0 spirals?", answer "yes".
2. Otherwise, you should attempt to use one of your remaining bills (the 2, then the 5). Call the value of this bill B.
3. Ask someone with both bills, "Can you make change for $N-B$ spirals?" and wait until you get an answer:
   - If the answer is "yes":
     1. Take the bills the person gives you
     2. Add the bill you used to the pile
     3. Tell the person who asked you, "yes", and hand them the pile of bills.
   - If the answer is "no":
     1. If you have any bills left, go back to step #2 and follow the same procedure attempting to use one.
     2. If you are out of bills to try, tell the person who asked you, "no".

`eval` and `makeChange`

To `eval`(e)

- If e is a number, return it.
- Otherwise, `eval` the left and the right; put them together with op

To `makeChange`(n):

- If $n = 0$, return `true`
- Otherwise:
  1. Check if we can make change for n by using a 2 bill; if so, return `true`
  2. Check if we can make change for n by using a 5 bill; if so, return `true`
  3. Give up and return `false`

Insight: The Structure of Recursive Problems

- Every recursive problem has a "trivial case" (the simplest expression is a number; the simplest number is 0).
  This case is called the **base case**.
- Every recursive problem breaks the problem up into smaller pieces (the expression pieces are left and right; the change pieces are use each type of bill). This case is called the **recursive case**.

The Many Ways To Think About Recursion

The Code Already Works!

This is the most important strategy for recursion!

When you are writing a recursive function, **pretend that it already works** and use it whenever possible.

Let Someone Else Do The Rest

Recursion is an army of people who can answer instances of your question. You solve a tiny piece and pass it on to someone else.

This is like the change example!

Where Can I Use My Function?

Before writing your recursive function, write down what it is supposed to do. Then, when writing it, try to find places that you can apply that idea to.

Writing the Evaluator

Now, let's go ahead and write the `eval` function we talked about. The goals of writing this function are to see the following about recursive code:

- The code is short
- The version with loops is horrid
- You can do really cool things with recursion

`printStars`

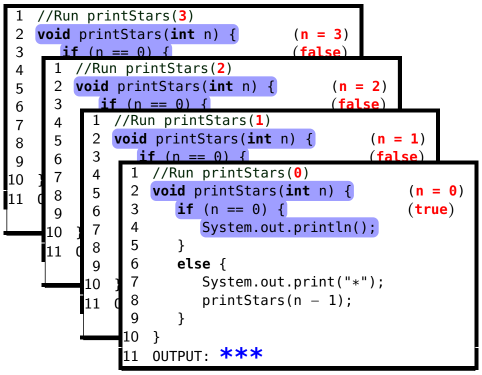
Consider the function `printStars`:

```
public static void printStars(int n) {
   for (int i = 0; i < n; i++) {
      System.out.print("*");
   }
   System.out.println();
}
```

Let's write it recursively.

```
public static void printStars(int n) {
   if (n == 0) {
      System.out.println();
   }
   else {
      System.out.print("*");
      printStars(n - 1);
   }
}
```

Tracing `printStars`



```
//Run printStars(3)
void printStars(int n) {          (n = 3)
   if (n == 0) {                  (false)
```

```
//Run printStars(2)
void printStars(int n) {          (n = 2)
   if (n == 0) {                  (false)
```

```
//Run printStars(1)
void printStars(int n) {          (n = 1)
   if (n == 0) {                  (false)
```

```
//Run printStars(0)
void printStars(int n) {          (n = 0)
   if (n == 0) {                  (true)
      System.out.println();
   }
   else {
      System.out.print("*");
      printStars(n - 1);
   }
}
OUTPUT: ***
```

Some Recursion Tips!

- Once you have a solution, it might feel obvious. This is a tricky feeling. Solving recursion problems is much harder than understanding a solution to a recursion problem.
- Understand the metaphors/ideas/ways to think about recursion. Choose one that makes the most sense to you, and run with it.
- Recursion will always have at least one base case and at least one recursive call.
- Be able to write down the steps in a recursive trace when given a recursive function.