CSC148 *Intro. to Computer Science*

Lecture 7: *Recursive Functions/Structures Trees*

Amir H. Chinaei, Summer 2016

Office Hours: R 10-12 BA4222

ahchinaei@cs.toronto.edu

http://www.cs.toronto.edu/~ahchinaei/

Course page:

http://www.cs.toronto.edu/~ahchinaei/teaching/20165/csc148/

Last week

- Reading recursive functions utilized list comprehension
- Tracing recursive functions
  - **dig down, come up**
  - Trace max_list([4, 2, [[4, 7],5], 8])

```
def max_list(L):
    if isinstance(L, list):
        return max([max_list(x) for x in L])
    else: # L is an int
        return L
```

- Today
  - More recursive functions
  - Tracing recursive functions using stacks
  - Recursive structures

More recursive examples

- Factorial <u>function</u>

  Factorial(n) = n * Factorial(n-1) **recursive case**

  Factorial(0) = 1 **base case**

- Fibonacci <u>function</u>

  Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2) **recursive case**

  Fibonacci(1) = 1

  Fibonacci(0) = 1 **base cases**

A recursive function has
at least one **base case** and at least one **recursive case**

Another example

A recursive definition: Balanced Strings

- Base case:
  - A string containing no parentheses is balanced

- Recursive cases:
  - (x) is balanced if x is a balanced string
  - xy is balanced if x and y are balanced strings

How about these functions?

- $f(n) = n^2 + n - 1$
- $f(n) = g(n-1) + 1$, $g(n) = n/2$
- $f(n) = 5$, $f(n-1) = 4$
- $f(n) = n * (n-1) * (n-2) * \ldots * 2 * 1$
- $f(n) = f(n/2) + 1$, $f(1) = 1$

Recursive programs

- Solution defined in terms of solutions for smaller problems

```
def solve (n):
    . . .
    value = solve(n-1) + solve(n/2)
    . . .
```

- One or more base cases

```
    if (n < 10):
        value = 1
```

- Some base case is always reached eventually; otherwise it's an infinite recursion

General form of recursion

if (condition to detect a base case):

{do something without recursion}

else: (general case)

{do something that involves recursive call(s)}

Recursive programs cont'd

0! = 1 and n! = n.(n-1)!

```
def factorial(n)
   # pre: n ≥ 0
   # post: returns n!
   if (n==0): return 1
   else: return n * factorial (n-1)
```

- structure of code typically parallels structure of definition

Recursive programs cont'd

Fib(0) = 1, Fib(1) = 1, Fib(n) = Fib(n-1) + Fib(n-2)

```
def fib(n):
  # pre: n ≥ 0
  # post: returns the nth Fibonacci number
  if (n < 2): return 1
  else: return fib(n-1) + fib(n-2)
```

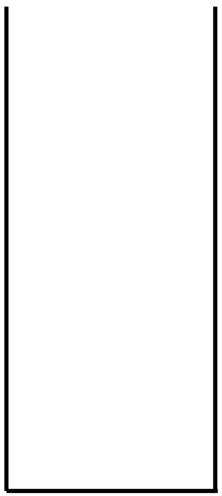
- structure of code typically parallels structure of definition

Stacks and tracing calls

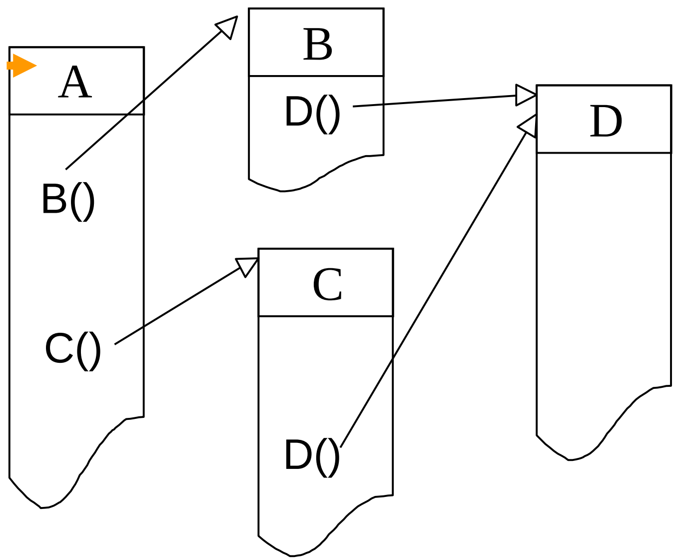
- Recall:
  - stack applications in compilers/interpreters
  - tracing method calls
- Activation record
  - all information necessary for tracing a method call
  - such as parameters, local variables, return address, etc.
- When method called:
  - activation record is created, initialized,
    and pushed onto the stack
- When a method finishes:
  - its activation record (that is on top of the stack)
    is popped from the stack

Tracing program calls

- Recall: stack of activation records
  - When method called:
    - activation record created, initialized, and pushed onto the stack
  - When a method finishes,
    - its activation record is popped

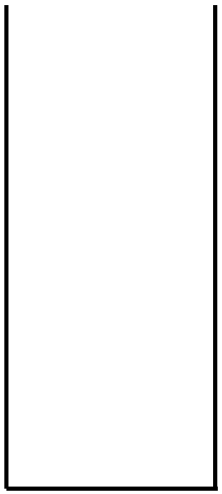


Stack of activation records

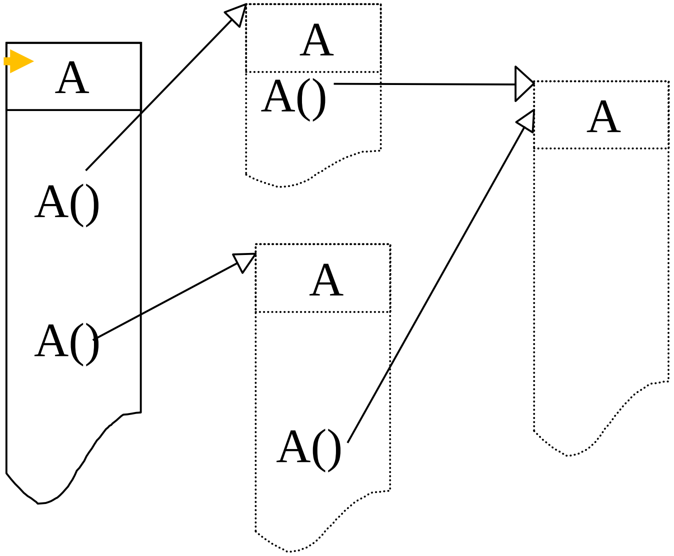


Tracing recursive programs

- same mechanism for recursive programs

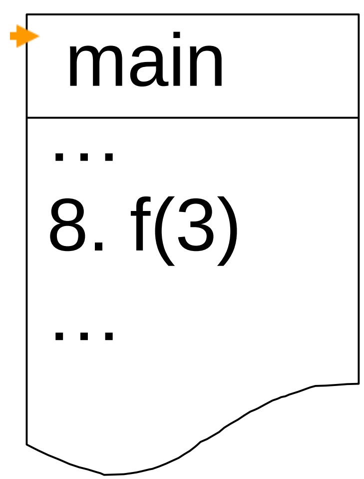


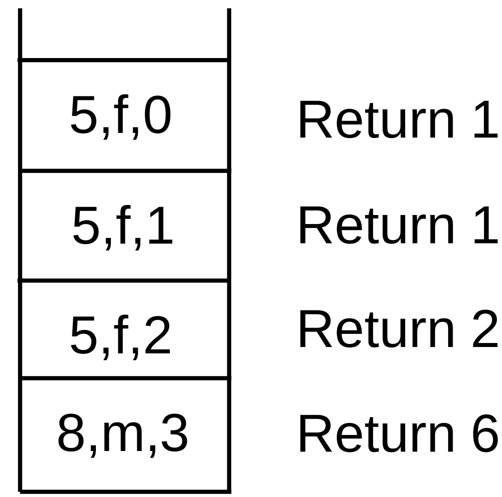
Stack of activation records



Tracing Factorial

```
1. def f(n):
2.    # pre: n≥0
3.    # post: returns n!
4.    if (n==0): return 1
5.    else: return n * f(n-1)
```





Stack of activation records

line#	func.	n

Tracing Factorial: intuitively

- f(3)

Tracing max_list(), using stack?

```
def max_list(L):
    if isinstance(L, list):
        return max([max_list(x) for x in L])
    else: # L is an int
        return L
```

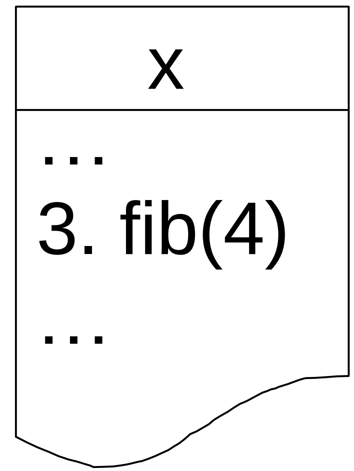
Trace max_list([4, 2, [[4, 7],5], 8])

Tracing max_list(), using stack?

Trace max_list([4, 2, [[4, 7],5], 8])

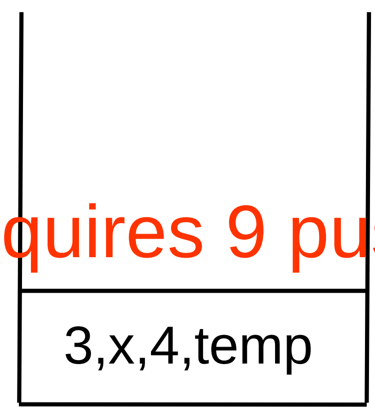
Tracing Fibonacci

```
1. def fib(n):
2.     # pre: n ≥ 0
3.     # post: returns the
4.     # nth Fibonacci number
4.     if (n < 2): return 1
5.     else: return fib(n-1) +
6.                  fib(n-2)
```



line#	func.	n	temp

Hint: requires 9 pushes



Stack of
activation records

Why 9?

- Using rewriting

fib(4)

fib(3)+ fib(2)

fib(2)+fib(1)

fib(1)+fib(0)

Recursive vs iterative

- Recursive functions impose a loop
- The loop is implicit and the compiler/interpreter (here, Python) takes care of it
- This comes at a price: time & memory
- The price may be negligible in many cases

- After all, no recursive function is more efficient than its iterative equivalent

Recursive vs iterative cont'ed

- Every recursive function can be written iteratively (by explicit loops)
  - may require stacks too
- yet, when the nature of a problem is recursive, writing it iteratively can be
  - time consuming, and
  - less readable

- So, recursion is a very powerful technique for <u>problems that are naturally recursive</u>

More examples

- Merge Sort
- Quick Sort
- Tower of Hanoi

- Balanced Strings
- Traversing Trees
- In general, properties of Recursive Definitions/Structures

- ….

Looking for exercises? Implement the above examples without seeing the sample solutions/algorithms.

Merge sort

```
Msort (A, i, j)
if (i < j)
     S1 := Msort(A, i , (i+j)/2)
     S2 := Msort(A, (i+j)/2, j)
     Merge(S1,S2, i, j)
end
```

Implement it in Python

Quick sort

```
Qsort (A, i, j)
if (i < j)
     p := partition(A)
     Qsort (A, i, p-1)
     Qsort (A, p+1, j)
end
```

Implement it in Python

Tower of Hanoi

```
Hanoi (n, s, d, aux)
if (n=1)
    "move from " +s+ " to " +d
else
    Hanoi (n-1, s, aux, d)
    "move from " +s+ " to " +d
    Hanoi (n-1, aux, d, s)
end
```

Implement it in Python

Let’s move on to a new topic

Tree terminology

- Set of **nodes** (possibly with values or labels), with directed **edges** between some pairs of nodes
- One node is distinguished as **root**
- Each non-root node has exactly one **parent**
- A **path** is a sequence of nodes $n_1; n_2; \ldots; n_k$, where there is an edge from n_i to n_{i+1} , $i<k$
- The **length** of a path is the number of edges in it
- There is a unique path from the root to each node. In the case of the root itself this is just n_1, if the root is node n_1
- There are no **cycles**; no paths that form loops.

Tree terminology cont'd

- **leaf**: node with no children
- **internal node**: node with one or more children
- **subtree**: tree formed by any tree node together with its descendants and the edges leading to them.
- **height**: 1+ the maximum path length in a tree. A node also has a height, which is 1+ the maximum path length of the tree rooted at that node
- **depth**: length of the path from the root to a node, so the root itself has depth 0
- **arity**, branching factor: maximum number of children for any node

General tree implementation

```python
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.
    """

    def __init__(self, value=None, children=None):
        """
        Create Tree self with content value and 0 or more children

        :param value: value contained in this tree
        :type value: object
        :param children: possibly-empty list of children
        :type children: list[Tree]
        """

        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []
```

How many leaves?

```python
def leaf_count(t):
    """
    Return the number of leaves in Tree t.

    :param t: tree to count the leaves of
    :type t: Tree
    :rtype: int

    >>> t = Tree(7)
    >>> leaf_count(t)
    1
    >>> t = descendants_from_list(Tree(7), [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> leaf_count(t)
    6
    """
    pass
```

How many leaves?

```
def leaf_count(t):
    """
    Return the number of leaves in Tree t.

    :param t: tree to count the leaves of
    :type t: Tree
    :rtype: int

    >>> t = Tree(7)
    >>> leaf_count(t)
    1
    >>> t = descendants_from_list(Tree(7), [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> leaf_count(t)
    6
    """
    if len(t.children) == 0:
            # t is a leaf
            return 1
    else:
             # t is an internal node
            return sum([leaf_count(c) for c in t.children])
```

Height of this Tree

```
def height(t):
    """
    Return 1 + length of longest path of t.
    :param t: tree to find height of
    :type t: Tree
    :rtype: int
    >>> t = Tree(13)
    >>> height(t)
    1
    >>> t = descendants_from_list(Tree(13),
    [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> height(t)
    3
    """
    # 1 more edge than the maximum height of a child, except
    # what do we do if there are no children?
    pass
```

Height of this Tree

```python
def height(t):
    """
    Return 1 + length of longest path of t.
    :param t: tree to find height of
    :type t: Tree
    :rtype: int
    >>> t = Tree(13)
    >>> height(t)
    1
    >>> t = descendants_from_list(Tree(13),
    [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> height(t)
    3
    """
    # 1 more edge than the maximum height of a child, except
    # what do we do if there are no children?
    if len(t.children) == 0:
        # t is a leaf
        return 1
    else:
        # t is an internal node
        return 1+max([height(c) for c in t.children])
```

arity, branch factor

```python
def arity(t):
    """
    Return the maximum branching factor (arity) of Tree t.

    :param t: tree to find the arity of
    :type t: Tree
    :rtype: int

    >>> t = Tree(23)
    >>> arity(t)
    0
    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(7)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> arity(tn1)
    4
    """
    pass
```

arity, branch factor

```python
def arity(t):
    """
    Return the maximum branching factor (arity) of Tree t.

    :param t: tree to find the arity of
    :type t: Tree
    :rtype: int

    >>> t = Tree(23)
    >>> arity(t)
    0
    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(7)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> arity(tn1)
    4
    """
    if len(t.children) == 0:
        # t is a leaf
        return 0
    else:
        # t is an internal node
        return max([len(t.children)]+[arity(n) for n in t.children])
```

count

```python
def count(t):
    """
    Return the number of nodes in Tree t.

    :param t: tree to find number of nodes in
    :type t: Tree
    :rtype: int

    >>> t = Tree(17)
    >>> count(t)
    1
    >>> t4 = descendants_from_list(Tree(17), [0, 2, 4, 6, 8, 10, 11], 4)
    >>> count(t4)
    8
    """
    pass
```

count

```python
def count(t):
    """
    Return the number of nodes in Tree t.

    :param t: tree to find number of nodes in
    :type t: Tree
    :rtype: int

    >>> t = Tree(17)
    >>> count(t)
    1
    >>> t4 = descendants_from_list(Tree(17), [0, 2, 4, 6, 8, 10, 11], 4)
    >>> count(t4)
    8
    """
  if len(t.children) == 0:
      # t is a leaf
      return 1
  else:
      # t is an internal node
      return 1+ sum([count(n) for n in t.children])
```