CSC148-Section:L0301
Week#9-Monday

Instructed by
AbdulAziz Al-Helali
a.alhelali@mail.utoronto.ca
Office hours: Wednesday 11-1, BA2230.

Slides adapted from Professor Danny Heap course material
winter17
Exam 2 Topics

• Trace recursion
• Recursion on nested Python list
• Recursion on class Tree
• Recursion on class BinaryTree
• Definitions for trees and binary trees, traversals (inorder, postorder, preorder, levelorder)
• Binary search trees
Outline

• Binary *Search* Tree
  • Delete

• Recursion efficiency
  • Maximum recursion depth
  • Redundancy
Recall – BinaryTree node

class BinaryTree:
    """
    A Binary Tree, i.e. arity 2.
    """

def __init__(self, value: object, left: Union['BinaryTree', None]=None, right: Union['BinaryTree', None]=None) -> None:
    """
    Create BinaryTree self with value and children left and right.
    """

    self.value, self.left, self.right = value, left, right
deletion of value from BST rooted at node?

• what return value?
• what to do if node is None?
• what if value to delete is less than that at node?
• what if it's more?
• what if the value equals this node's value and...
  • this node has no left child
  • ... no right child?
  • both children?
• what return value?
  • `return node` (for every call to delete)

A. what to do if node is None?
  A. if `node is None`:
     pass

B. what if value to delete is less than that at node?
  • #Branch to the left
  • `elif value < node.value`:
    node.left = delete(node.left, value)

C. what if it's more?
  • #Branch to the right
  • `elif value > node.value`:
    node.right = delete(node.right, value)
D. what if the value equals this node's value and... (neither greater nor smaller)

• We have 3 cases:

1. this node has no left child
   • **elif** node.left is None:
     node = node.right

2. ... no right child?
   • **elif** node.right is None:
     node = node.left

3. both children?
   • # One way to **not break BST definition**
   • # find the max node in left tree and put it in place of
   • # deleted node
     • node.value = **find max**(node.left).value
     node.left = **delete**(node.left, node.value)
   • # Alternatively
   • # find the min node in right tree and put it in place of
   • # deleted node
     • node.value = **find min**(node.right).value
     node.right = **delete**(node.right, node.value)
Recursion efficiency: Maximum recursion depth

```python
from linked_list_Wed import LinkedListNode

def recursive_append(b: LinkedListNode, data: object) -> None:
    """
    recursively append a node with data to linked list headed by b
    """
```
Recursion efficiency: Maximum recursion depth

```python
from linked_list_Wed import LinkedListNode

def recursive_append(b: LinkedListNode, data: object) -> None:
    """recursively append a node with data to linked list headed by b""
    if b.next is None:
        b.next = LinkedListNode(data)
    else:
        recursive_append(b.next, data)
```

Recursion efficiency: Maximum recursion depth

```python
b = LinkedListNode(1)
print(b)
recursive_append(b, 2)
print(b)
for i in range(3, 950):
    recursive_append(b, i)
print(b)
for i in range(950, 998):
    recursive_append(b, i)
```

File "D:/csci48/lectures/week9/limits.py", line 8, in recursive_append
b.next_ = LinkedListNode(data)
RecursionError: maximum recursion depth exceeded
Recursion efficiency: Redundancy

• Fibonacci numbers
  • “By definition, the first two numbers in the Fibonacci sequence are either 1 and 1, or 0 and 1, depending on the chosen starting point of the sequence, and each subsequent number is the sum of the previous two.”[wikipedia.org]

• The sequence $F_n$ of Fibonacci numbers is defined as:

\[ \text{If } n < 2: \quad F_n = 1 \]
\[ \text{Else: } F_n = F_{n-1} + F_{n-2} \]
Recursion efficiency: Redundancy

• Implement the following function recursively

```python
def fibonacci(n: int) -> int:
    """
    Return the nth fibonacci number, that is n if n < 2, or fibonacci(n-2) + fibonacci(n-1) otherwise.
    """

>>> fibonacci(0)
0
>>> fibonacci(1)
1
>>> fibonacci(3)
2
"""
```

src: wikipedia.org
Recursion efficiency: Redundancy

```python
def fibonacci(n: int) -> int:
    """
    Return the nth fibonacci number, that is n if n < 2, or fibonacci(n-2) + fibonacci(n-1) otherwise.
    """
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

```python
>>> fibonacci(0)
0
>>> fibonacci(1)
1
>>> fibonacci(3)
2
"""
```
Recursion efficiency: Redundancy

- Although the implementation is very easy using recursion
- It is not efficient. Why?
Recursion efficiency: Redundancy

- Although the implementation is very easy using recursion
- It is not efficient. Why?
  - Repeated calculations
Recursion efficiency: Redundancy

• One Solution is to use **Memoization**

• “In computing, memoization or memoisation is an optimization technique used primarily to **speed up** computer programs by **storing the results of expensive function calls** and returning the cached result when the same inputs occur again.”[wikipedia.org]
Recursion efficiency: Redundancy-One Solution  

Memoization

```python
def fib_memo(n: int, seen: dict) -> int:
    """
    Return the nth fibonacci number (n) reasonably quickly.
    uses seen to store already-seen results
    """

    if n not in seen:
        if n < 2:
            seen[n] = n
        else:
            seen[n] = fib_memo(n - 2, seen) + fib_memo(n - 1, seen)
    return seen[n]
```