i206: Lecture 11:
Hash Tables (Dictionaries); Recursion

Tapan Parikh

Spring 2013

Some slides courtesy Marti Hearst, John Chuang and others

Outline

- What is a data structure
- Basic building blocks: arrays and linked lists
- Data structures (uses, methods, performance):
  - List, stack, queue
  - Dictionary
  - Tree
  - Graph

Dictionary

- Also known as hash table, lookup table, associative array, or map
- A searchable collection of key-value pairs
  - each key is associated with one value
- Many possible applications, e.g.,
  - Address book
  - Student record database
  - Routing tables
  - …

Hash Tables

- We want a data structure that, given a collection of *n* keys, implements the *dictionary operations* **`Insert()`**, **`Delete()`** and **`Search()`** efficiently.
- **Binary search trees:** can do that in O(log n) time and are space efficient.
- **Arrays:** can do this in O(1) time but they are not space efficient.
- **Hash Tables:** A generalization of an array that under some reasonable assumptions is O(1) for Insert/Delete/Search of a key

Slide adapted from lecture by Andreas Veneris

Why Not Arrays?

- How can you store all Social security numbers in an array and have O(1) access?
  - Use an array with range 0 - 999,999,999
  - This will give you O(1) access time but …
  - …considering there are approx. 300,000,000 people in the USA you waste 1,000,000,000-300,000,000 array entries!

- **Problem:** The range of key values we are mapping is too large (0-999,999,999) when compared to the # of keys (American citizens)
- So an array is an inefficent use of space in this example.

Slide adapted from lecture by Andreas Veneris

Dictionary Methods

- get(k): if the dictionary has an entry with key k, return its associated value; else return null
- put(k,v): insert entry (k,v) into the dictionary; if key is not already in dictionary, return null; else return old value associated with k
- remove(k): if dictionary has an entry with key k, remove it from dictionary and return its associated value; else return null
- size()
- isEmpty()

Dictionary in Python

```
>>>
>>> dict ={"cat": 3, "dog": 5, "fish": 7}
>>> dict["cat"]
3
>>> dict.keys()
dict_keys(['fish', 'dog', 'cat'])
>>> dict.values()
dict_values([7, 5, 3])
>>>
```

Python Activity

- Use a loop to create a dictionary mapping characters to their corresponding ASCII integer values
  - Hint #1, your loop should go from 32 to 126
  - Hint #2, chr(n) returns the character represented by ASCII code n
- Use this dictionary to convert any string to a set of comma-delimited ASCII codes

Desirable Properties

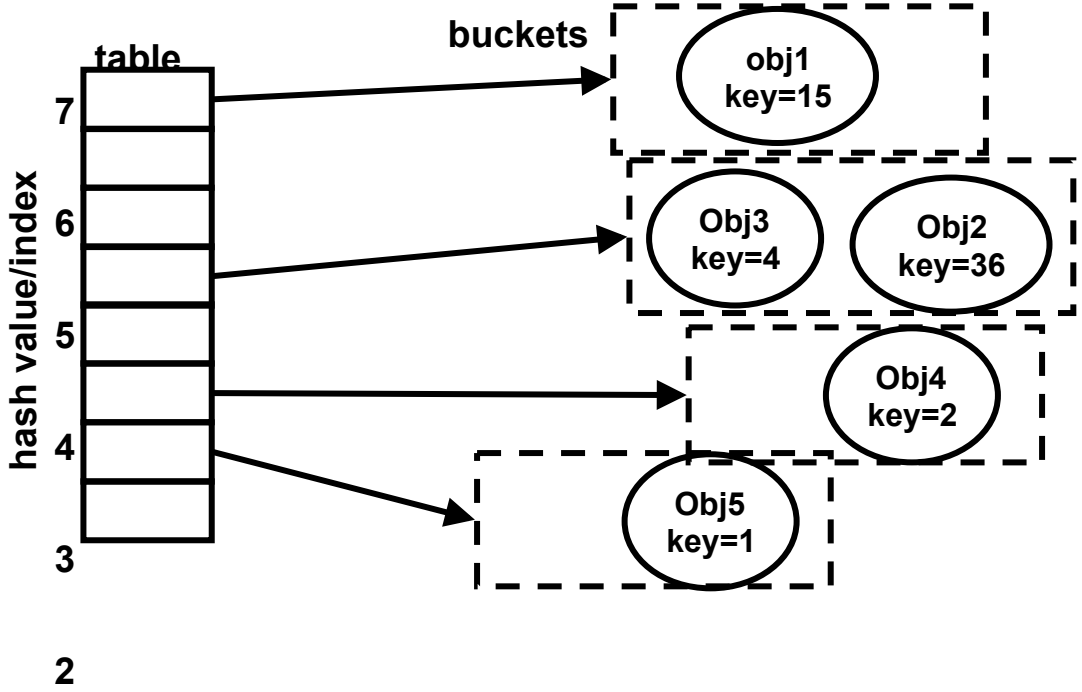
- Fast search, insert and delete (get, put, and remove)
- Efficient space usage

Hash Table

- A generalization of an array that, if properly designed, can realize fast insert/delete/search operations with good space efficiency
  - average run-time of O(1)
  - worst case run-time of O(n)
- A hash table is:
  - Good for storing and retrieving key-value pairs
  - Not good for iterating through a list of items

Hash Table

- A hash table consists of two major components:
  - Bucket array (for storing entries)
  - Hash function (for mapping keys to buckets)



Hash Table Design

- Bucket array is an array A of size N, where each cell of A is considered a "bucket"
- Hash function h maps each key k to an integer in the range [0, N -1]
- Store entry (k,v) in the bucket $A[h(k)]$
- Search for entry (k,v) in the bucket $A[h(k)]$
- Choose hash function such that
  - Can take arbitrary objects (keys) as input
  - Hash computation is fast
  - Key mappings evenly distributed across [0, N -1]

Example Hash Function

- h(k) = k mod N

- mod stands for **modulo,** the remainder of the division of two numbers. For example:
  - 8 mod 5 = 3
  - 9 mod 5 = 4
  - 10 mod 5 = 0
  - 15 mod 5 = 0

- Observe that collisions are possible:
  - two different keys hash to the same value

Example

- h(k) = k mod 5

Insert (2,x)

	key	value
0		
1		
2	2	x
3		
4		

Insert (21,y)

	key	value
0		
1	21	y
2	2	x
3		
4		

Insert (34,z)

	key	value
0		
1	21	y
2	2	x
3		
4	34	z

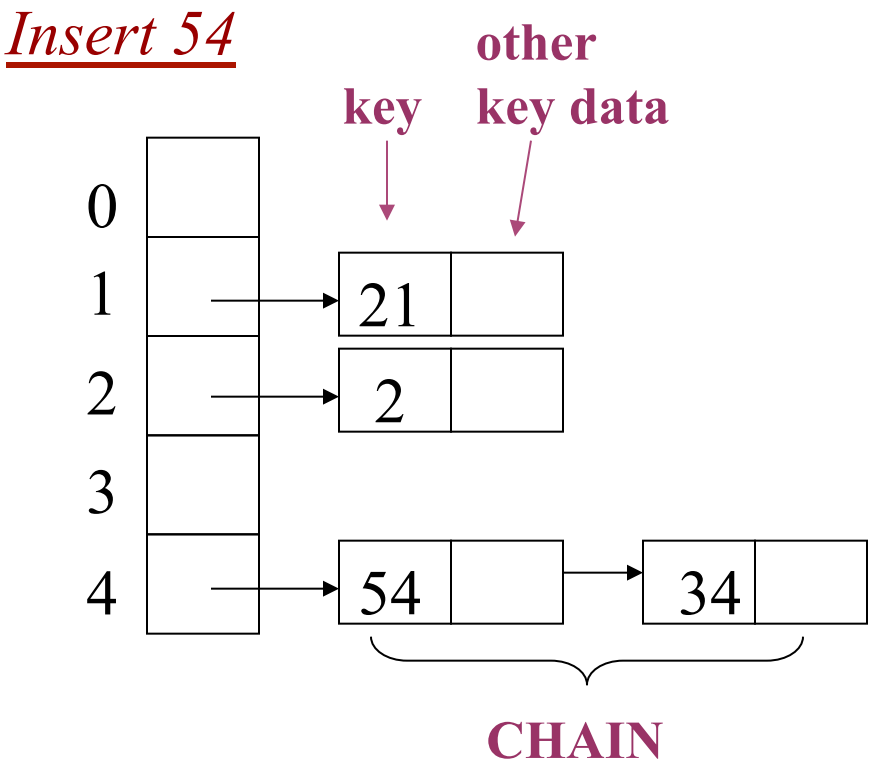
Insert (54,w)

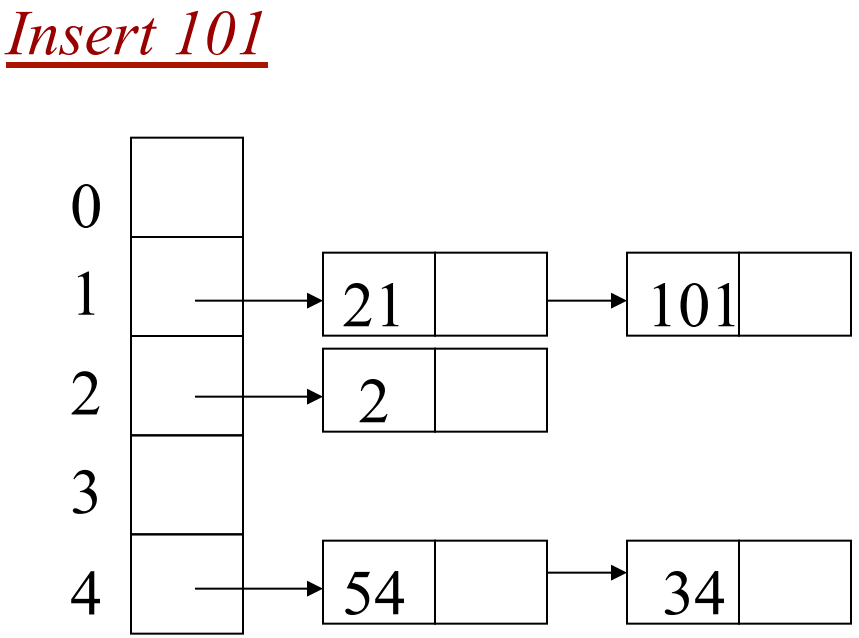
There is a ***collision*** at array entry #4

???

Dealing with Collisions

•Hashing with Chaining: every hash table entry contains a pointer to a linked list of keys that hash in the same entry.

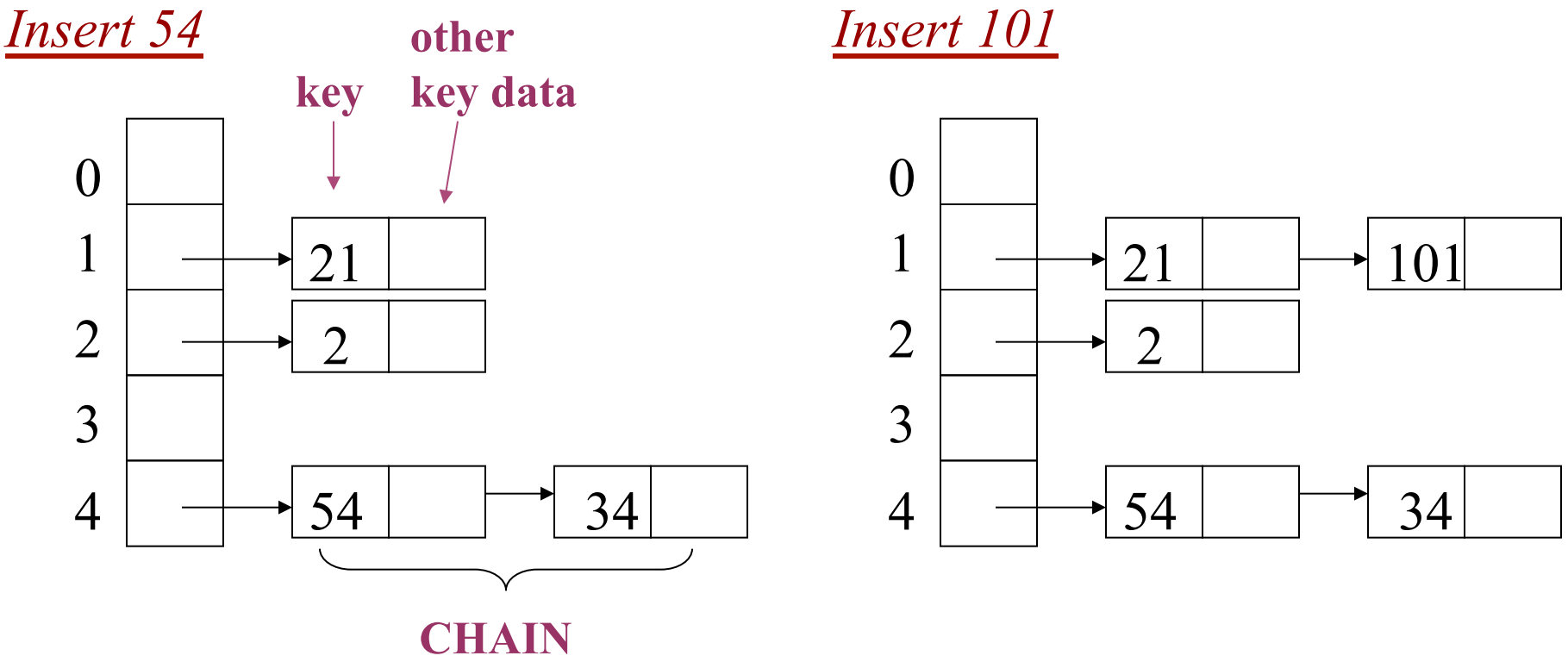




Slide adapted from lecture by Andreas Veneris

Hashing with Chaining

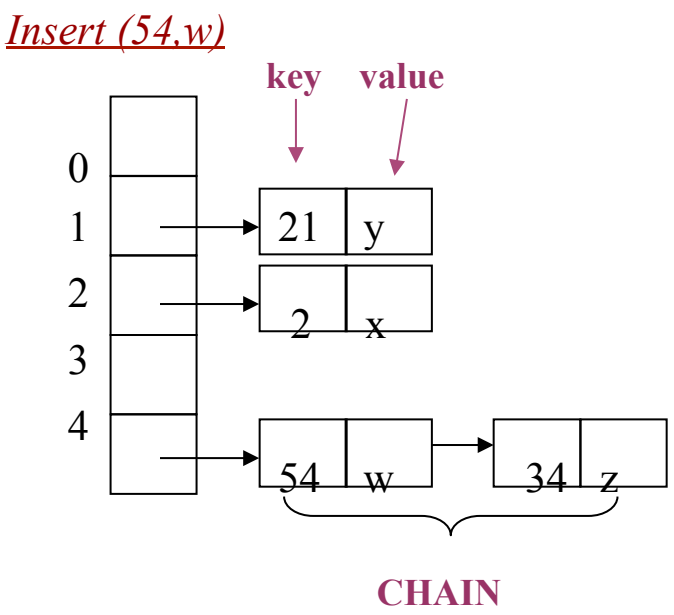
The problem is that keys 34 and 54 hash in the same entry (4). We solve this *collision* by placing all keys that hash in the same hash table entry in a LIFO list (**chain or bucket**) pointed by this entry:

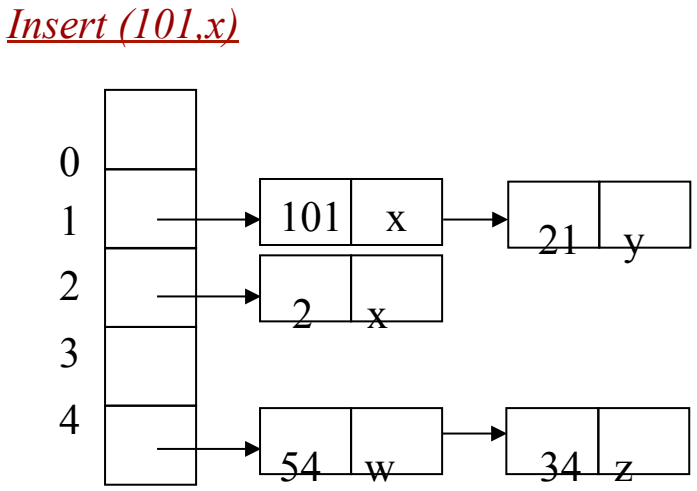


Slide adapted from lecture by Andreas Veneris

Dictionary Performance

- What is the run time for insert/search/delete?
  - Insert: It takes O(1) time to compute the hash function and insert at head of linked list
  - Search: It is proportional to the length of the linked list
  - Delete: Same as search





Load Factor

- Average length of the linked lists is a function of the load factor
  - Load factor =

    number of items in hash table / array size

- In Python, the implementation details are transparent to the programmer (load factor kept under 2/3)
- In Java, the programmer can set the initial table size (default=16) and/or the load factor (default = 0.75)

Rule of thumb:

- Try to keep space utilization (load factor) between 50% and 80%

$$\textbf{Load factor} = \frac{\text{\# keys used}}{\text{total \# slots in table}}$$

- If < 50%, wasting space
- If > 80%, overflows significant
  depends on how good hash function is & on # keys/bucket

Slide adapted from lecture by Hector Garcia-Molina

Choosing a Hash Function

- The performance of the hash table depends on a having a hash function which evenly distributes the keys.
- Choosing a good hash function requires taking into account the kind of data that will be used.
  - The statistics of the key distribution needs to be accounted for.
  - E.g., Choosing the first letter of a last name will cause problems depending on the nationality of the population
- Most programming languages (including java) have hash functions built in.

Slide adapted from lecture by Andreas Veneris

Python Activity

- Create hash function mapping any string to a value between 0 and 9 by using the ASCII codes of the individual characters
  - Hint: ord(n) returns the ASCII integer code of character n

Recursive Programs / Algorithms

An algorithmic technique in which a function, in order to accomplish a task, calls itself with some part of the task.

- Need
  - A base case (so the program will end)
  - The recursive step(s)
    - The recursive step must “narrow” or “reduce” the problem in some manner.

Recursion

- A method that invokes itself
- Examples:
  - GNU stands for: GNU's Not Unix
    (part of its own definition!)
  - A dog is a Collie if its name is Lassie or its mother was a Collie.
    - Start with a given dog.
    - Is its name Lassie? If so, done.
    - Else is its mother a Collie?
      - Is the mother named Lassie? If so, done.
      - Else is *its* mother a Collie?
      - … and so on.

Recursion Example

Let’s illustrate with a mother-child relationship.

Definition: A dog is a collie if its name is Lassie or if its mother is Lassie, or if its mother’s mother is Lassie, or … if any of its female ancestors is Lassie.

```
class Collie ():
    def __init__(self, name="", mother=None):
        self.name = name
        self.mother = mother
    def getName(self):
        return self.name
    def getMother(self):
        return self.mother
```

Recursion Example

Definition: A dog is a collie if its name is Lassie or if its mother is Lassie, or if its mother's mother is Lassie, or … if any of its female ancestors is Lassie.

```
class Collie ():
    def __init__(self, name="",
    mother=None):
        self.name = name
        self.mother = mother
    def getName(self):
        return self.name
    def getMother(self):
        return self.mother

def isCollie(dog):
    if not dog: return False
    print("Dog name is " + dog.getName())
    if dog.getName() == "lassie": return True
    return(isCollie(dog.getMother()))
```

Recursion Methods

- Recursive methods need:
  - A base case / terminating condition (or else it doesn't halt)
  - A body (containing the recursive call)

```
class Collie ():
    def __init__(self, name="",
    mother=None):
        self.name = name
        self.mother = mother
    def getName(self):
        return self.name
    def getMother(self):
        return self.mother

def isCollie(dog):
    if not dog: return False
    print("Dog name is " + dog.getName())
    if dog.getName() == "lassie": return True
    return(isCollie(dog.getMother()))
```

Base cases

Recursive step

```
def makeDogAncenstry1():
    lassie = Collie("lassie", None)  #last ancestor
    fido = Collie("fido", lassie)
    fifi = Collie("fifi", fido)
    rover = Collie("rover", fifi)
    return rover

def makeDogAncenstry2():
    lassie = Collie("missie", None)
    fido = Collie("fido", lassie)
    fifi = Collie("fifi", fido)
    rover = Collie("rover", fifi)
    return rover


def isCollie(dog):
    if not dog: return False
    print("Dog name is " + dog.getName())
    if dog.getName() == "lassie": return True
    return(isCollie(dog.getMother()))

firstDog = makeDogAncenstry1()
print("Is Collie? " + str(isCollie(firstDog)))

firstDog = makeDogAncenstry2()
print("Is Collie? " + str(isCollie(firstDog)))
```

Recursion vs. Loops

- We could use recursion as an alternative to a loop for our various summation problems.
- This does not reduce (or increase) the O(n) of the problem.

```
def recursiveSum1(n):
    if n == 0: return n
    return 1 + recursiveSum1(n-1)

print(recursiveSum1(5))

def recursiveSum2(n):
    if n == 0: return n
    return n + recursiveSum2(n-1)

print(recursiveSum2(5))
```

Recursive Summing

- Try it out here:

http://www.pythontutor.com/visualize.html

```
def recursiveSum2(n):
    if n == 0: return 0
    return n + recursiveSum2(n-1)
recursiveSum2(5)
```

Use **left** and **right** arrow keys to step through this code:

```
1 def recursiveSum2(n):
2     if n == 0: return 0
3     return n + recursiveSum2(n-1)
4 recursiveSum2(5)
```

Edit code

<< First | < Back | About to do step 20 of 25 | Forward > | Last >>

Program output:

Stack grows down

Global variables

recursiveSum2

recursiveSum2

n 5

recursiveSum2

n 4

recursiveSum2

n 3

recursiveSum2

n 2

recursiveSum2

n 1

recursiveSum2

n 0

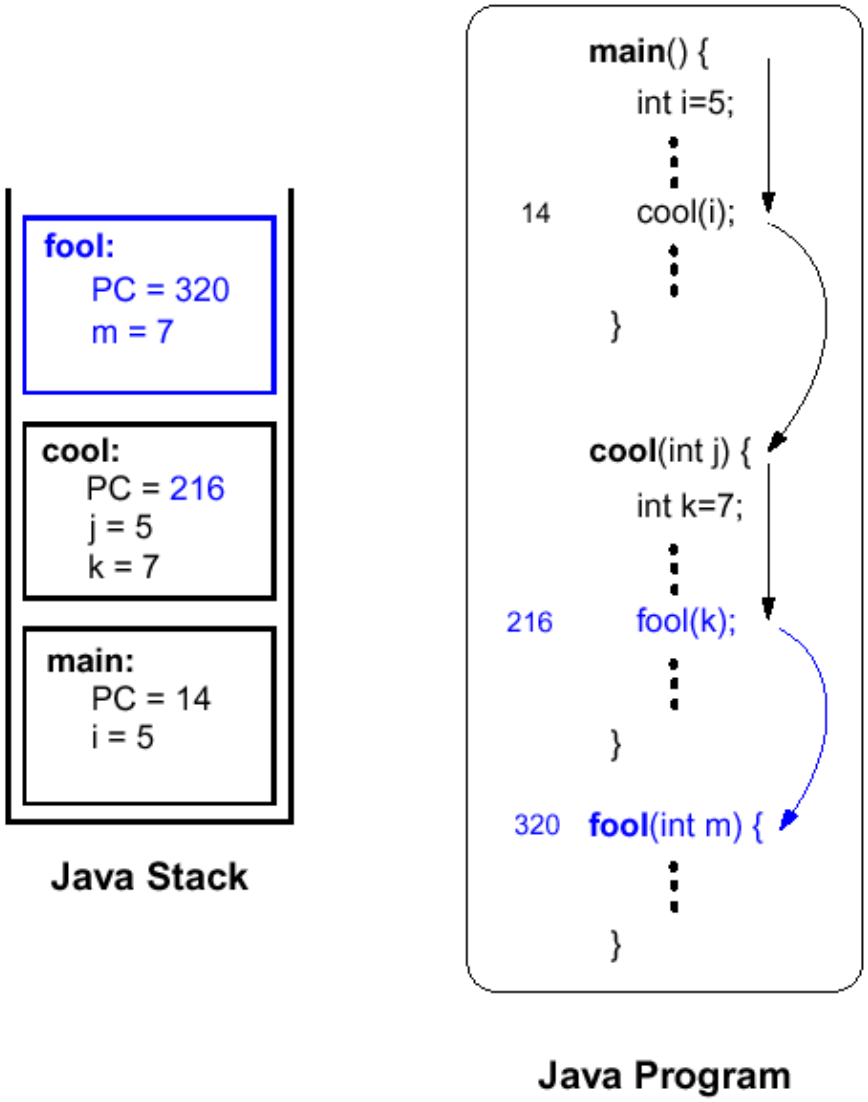
About to return to caller

Return value: 0

Methods and the Operating System's Runtime Stack

- Each time a method is called
  - A new copy of the method's data is pushed on the stack
    - This data includes local variables
  - This applies to all methods, not only recursive ones
- The "return" command does two things:
  - Pops the data for the method off the stack
    - Local variables disappear
  - Passes the return value to the calling method
- If there is no return statement
  - The method is popped
  - Nothing is passed to the calling method

Java Method Stack



From Goodrich & Tamassia

Recursion & the Runtime Stack

- Each call of the recursive method is placed on the stack.
- When we finally reach the base case, all the frames are popped off of the stack.

Summary: Recursion

- Recursion is a useful problem-solving technique
  - We will see it again when we study trees and sorting
- But it takes some getting used to.
- To help:
  - Remember that each method call gets put on the stack until the base case is found; then the calls are popped off the stack.
  - You HAVE TO understand basic programming concepts like
    - Method calls
    - Returning from methods
    - Passing parameters