

CSCI 553: Networking III

Unix Network Programming

Spring 2007



Python Sets, Dictionaries and
Complexity



Introduction

- The world is *not* made of lists
 - Just because it's the first data structure you meet, doesn't make it the right one for every task
- This lecture introduces two other fundamental data structures
 - Allow you to create programs that are simpler and more efficient
- Also look at what “efficient” really means



Sets

- A set is an unordered collection of distinct items
 - Unordered: items are looked up by value, rather than location
 - Distinct: any value appears at most once
- Fundamental in mathematics, but an afterthought in most programming languages
- The set type is built in to Python 2.4 and higher
 - Create a new set by calling `set()`
 - Then insert and remove values, test for membership, etc.

```
vowels = set()
for char in 'aieoeiaooaaeieou':
    vowels.add(char)
print vowels
```

```
Set(['a', 'i', 'e', 'u', 'o'])
```



Set Operations

- Like other objects, sets have methods

- Many of which can be expressed using operators as well

Method	Purpose	Example	Result	Alternative Form
Example values:	<code>ten = set(range(10))</code>	<code>lows = set([0, 1, 2, 3, 4])</code>	<code>odds = set([1, 3, 5, 7, 9])</code>	
<code>add</code>	Add an element to a set	<code>lows.add(9)</code>	None	<code>lows</code> is now <code>set([0, 1, 2, 3, 4, 9])</code>
<code>clear</code>	Remove all elements from the set	<code>lows.clear()</code>	None	<code>lows</code> is now <code>set()</code>
<code>difference</code>	Create a set with elements that are in one set, but not the other	<code>lows.difference(odds)</code>	<code>set([0, 2, 4])</code>	<code>lows - odds</code>
<code>intersection</code>	Create a set with elements that are in both arguments	<code>lows.intersection(odds)</code>	<code>set([1, 3])</code>	<code>lows & odds</code>
<code>issubset</code>	Are all of one set's elements contained in another?	<code>lows.issubset(ten)</code>	True	<code>lows <= ten</code>
<code>issuperset</code>	Does one set contain all of another's elements?	<code>lows.issuperset(odds)</code>	False	<code>lows >= odds</code>
<code>remove</code>	Remove an element from a set	<code>lows.remove(0)</code>	None	<code>lows</code> is now <code>set([1, 2, 3, 4])</code>
<code>symmetric_difference</code>	Create a set with elements that are in exactly one set	<code>lows.symmetric_difference(odds)</code>	<code>set([0, 2, 4, 5, 7, 9])</code>	<code>lows ^ odds</code>
<code>union</code>	Create a set with elements that are in either argument	<code>lows.union(odds)</code>	<code>set([0, 1, 2, 3, 4, 5, 7, 9])</code>	<code>lows odds</code>



Set Example

- Have several files with observations of birds
- Want to find out which species have been seen
- Program:

```
lines = [  
    'canada goose', 'canada goose', 'long-tailed jaeger', 'canada goose',  
    'snow goose', 'canada goose', 'canada goose', 'northern fulmar'  
]
```

```
seen = set()  
for line in lines:  
    seen.add(line.strip())
```

```
for bird in seen:  
    print bird
```

```
northern fulmar  
snow goose  
long-tailed jaeger  
canada goose
```

- Note: for loops over the values in the set

How Set Values Are Stored

- Implementation goal is to make lookup as quick as possible
 - Without making insertion and removal expensive
- Use a *hash table*
 - Calculate a *hash code* for the object being inserted
 - Store the value at that location in an array
 - If the *hash function* is good, *collisions* will be rare
 - When they occur, chain values together in a sub-list
- Result: looking up a value takes constant time, regardless of how many values are being stored

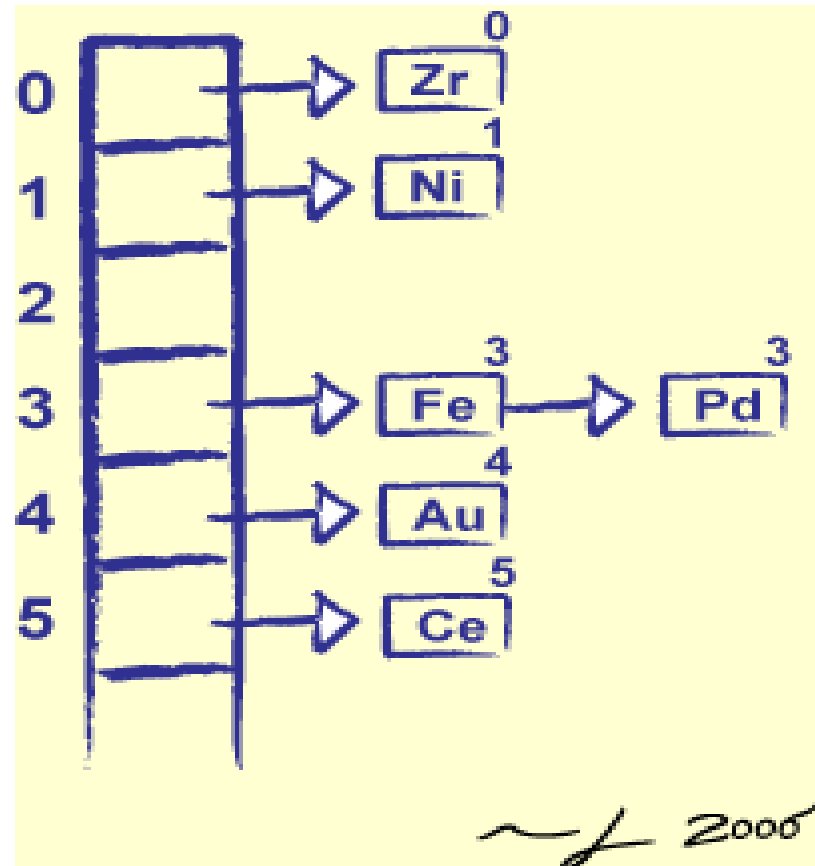


Figure 9.1: Hashing

Immutability

- This only works if a value's hash code never changes after it is inserted
 - If it does, the value will be in the wrong place

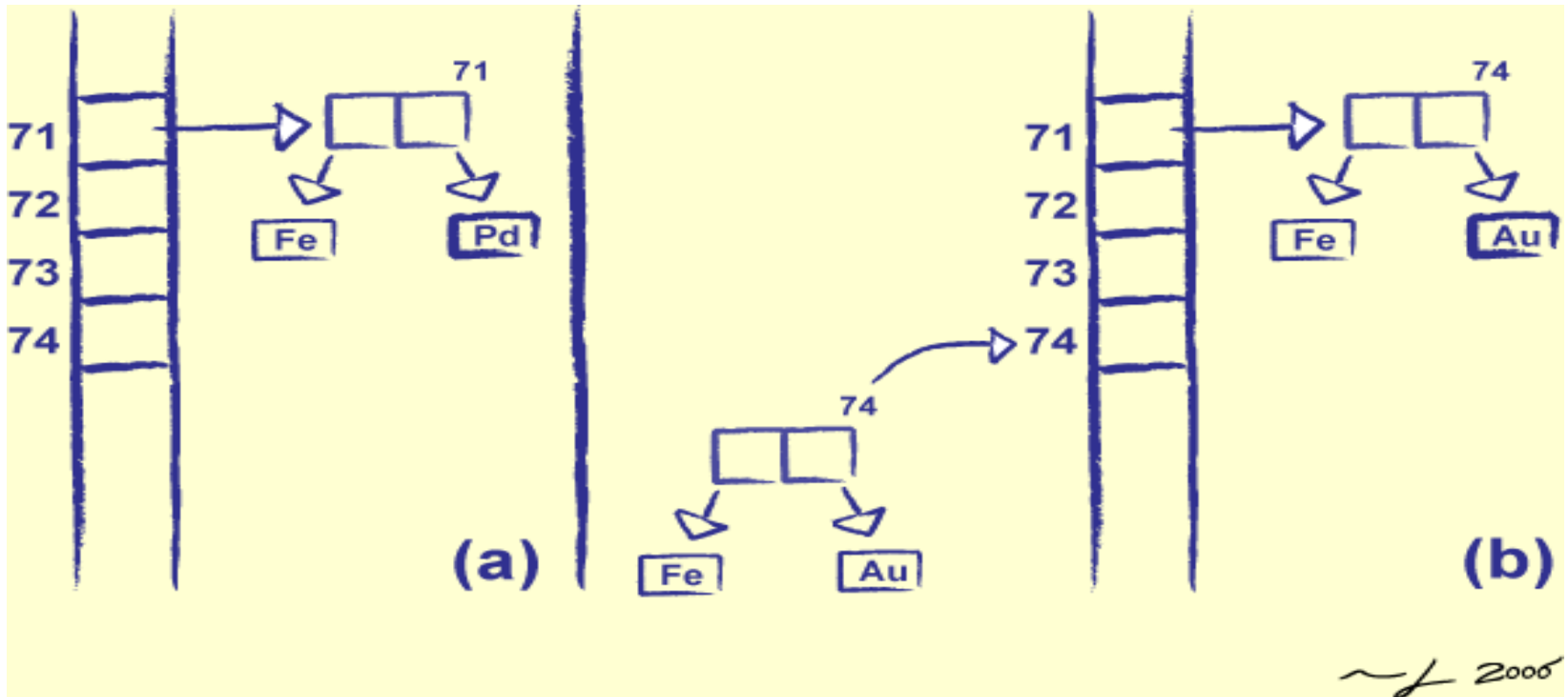


Figure 9.2: Misplaced Values



Immutability

- Python therefore only allows sets to contain immutable values
 - Booleans, numbers, strings, tuples...
 - ...but not lists

```
values = set()
values.add('birds')
print values
values.add(('Canada', 'goose'))
print values
values.add(['snow', 'goose'])
print values
```

```
set(['birds'])
set([('Canada', 'goose'), 'birds'])
Traceback (most recent call last):
  File "mutable_in_set.py", line 8, in ?
    values.add(['snow', 'goose'])
  File "/usr/lib/python2.3/sets.py", line 521, in add
    self._data[element] = True
TypeError: list objects are unhashable
```

- This is one of the reasons tuples were invented
 - Allow you to store multi-part values like ("snow", "goose")



Frozen Sets

- What about sets of sets?
 - Sets themselves have to be mutable, so that values can be inserted and removed
- Python also provides "frozen" sets
 - No changes allowed after creation
 - So they're almost always initialized from a collection of some kind

```
$ python
```

```
>>> birds = set()
```

```
>>> arctic = frozenset(['goose', 'tern'])
```

```
>>> birds.add(arctic)
```

```
>>> print birds
```

```
set([frozenset(['goose', 'tern'])])
```

```
>>> arctic.add('eider')
```

```
AttributeError: 'frozenset' object has no attribute  
'add'
```



A Note on Language Design

- Many languages allow mutable elements in sets, and trust users not to modify them after insertion
 - Which is a rich source of hard-to-find bugs
- Could also have values keep track of which sets they're in
 - “Move” them when their values change
 - Would make all programs slow for the benefit of a few
- Every software system contains tradeoffs like this

Efficiency

- So is using sets worthwhile?
- Imagine storing species names in a list instead of a set
 - Each if name in seen check requires $N/2$ comparisons on average
 - So building up those N values requires $N(N+1)/4$ comparisons
- Only requires N for a set
 - Difference is dramatic

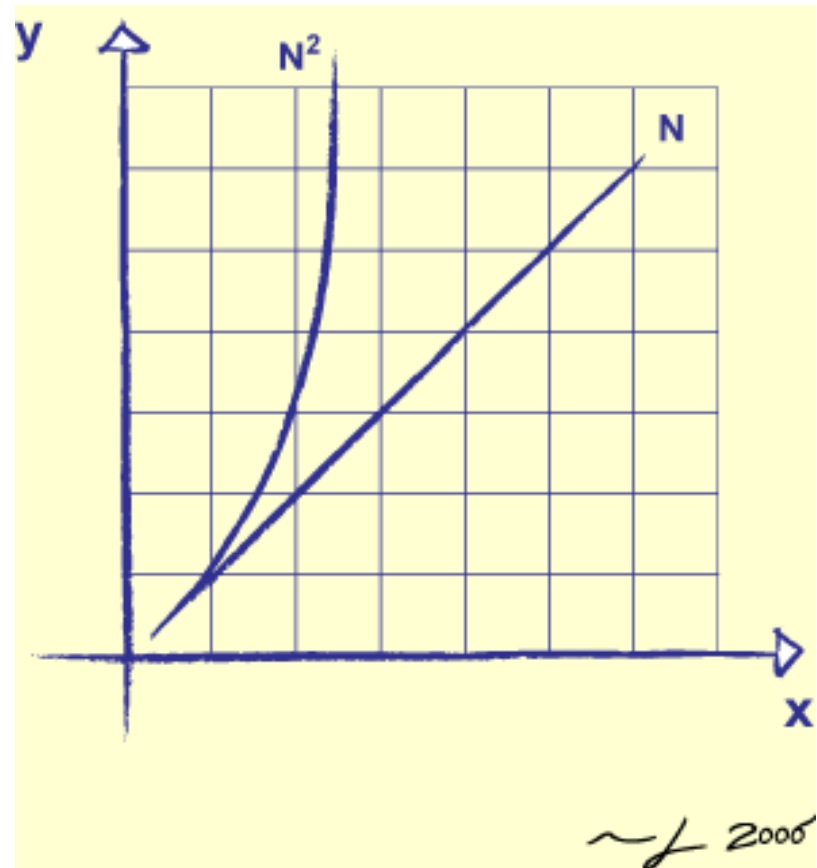


Figure 9.3: List vs. Set Performance

Complexity Curves

- Can get better performance out of the list if we keep it sorted
 - Use *binary search* to look for values

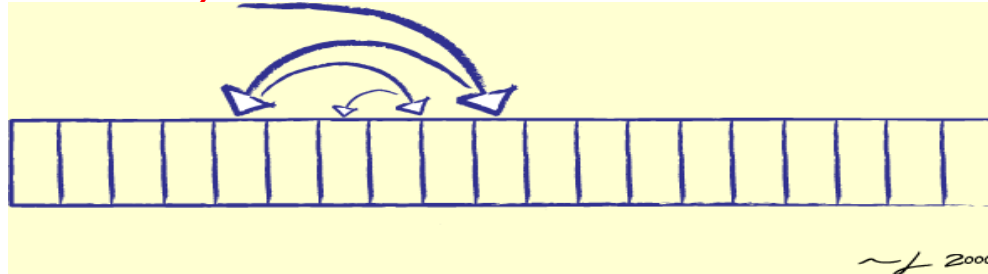


Figure 9.4: Binary Search

- K checks is enough to find a value in a list of length $2K$
 - So a list containing N values can be searched in $\log_2 N$ computational steps
 - Building a list of N values therefore requires roughly $N \log_2 N$ steps

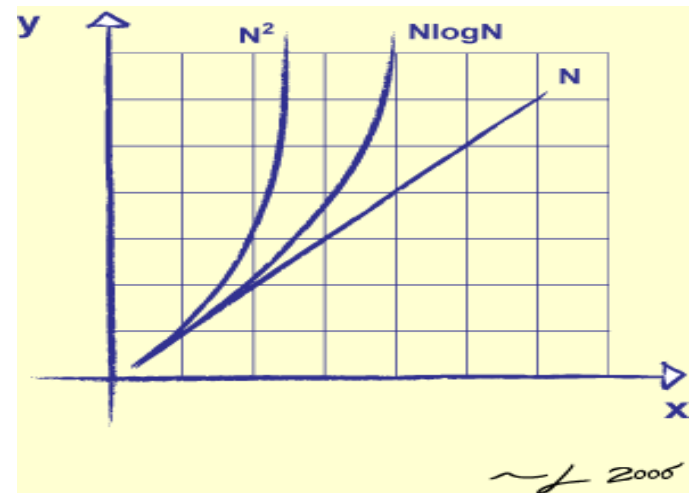


Figure 9.5: List vs. Set Performance Revisited



Algorithmic Complexity

- The relationship between problem size and running time is called *algorithmic complexity*
 - Usually described in terms of upper bounds
 - If $f(x) < kg(x)$ for large x and some constant k , then $f(x)$ is $O(g(x))$
- For example:
 - Something that takes the same time, regardless of data size, is $O(1)$
 - If the time grows as the logarithm of the data size, it is $O(\log N)$
 - If the time is proportional to the number of values, it is $O(N)$
 - Storing species names in a list is $O(N^2)$
 - Because if you throw away the constant 4, the difference between $N(N+1)/4 = (N^2 + N)/4$ and N^2 becomes insignificant as N grows large

Motivating Dictionaries

- Suppose you want to count how often each species of bird is seen
 - Can't store (name, count) in set...
 - ...because then you couldn't look up a species' count unless you already knew what it was
- Could fall back on lists of pairs...
- Better solution: store extra data with each element of a set
 - A *dictionary* associates one value with each of its *keys*
 - An unordered mutable collection
 - Also called maps, hashes, and associative arrays
 - Often visualized as two-column table

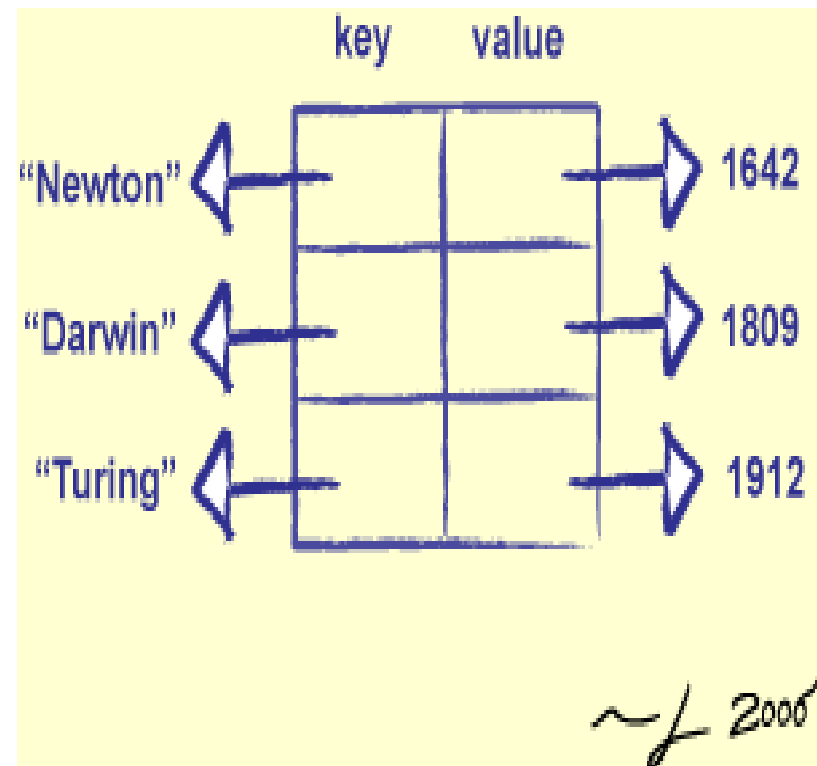


Figure 9.6: Dictionaries as Tables



Creating and Indexing

- Create a dictionary by putting key/value pairs inside {}
 - {'Newton':1642, 'Darwin':1809}
 - Empty dictionaries are written {}
- Look up the value associated with a key using []

```
birthday = {  
    'Newton' : 1642,  
    'Darwin' : 1809  
}  
print "Darwin's birthday:", birthday['Darwin']  
print "Newton's birthday:", birthday['Newton']
```

```
Darwin's birthday: 1809  
Newton's birthday: 1642
```

- Can only access keys that are present
 - Just as you can't index elements of a list that aren't there

```
birthday = {  
    'Newton' : 1642,  
    'Darwin' : 1809  
}  
print birthday['Turing']
```

```
Traceback (most recent call last):  
  File "key_error.py", line 5, in ?  
    print birthday['Turing']  
KeyError: 'Turing'
```



Updating Dictionaries

- **Assigning to a dictionary key:**
 - Creates a new entry if the key is not already in dictionary
 - Overwrites the previous value if the key is already present

```
birthday = {}  
birthday['Darwin'] = 1809  
birthday['Newton'] = 1942 # oops  
birthday['Newton'] = 1642  
print birthday
```

```
{'Darwin': 1809, 'Newton': 1642}
```



Updating Dictionaries

- Remove an entry using `del d[k]`
 - Can only remove entries that are actually present

```
birthday = {
    'Newton' : 1642,
    'Darwin' : 1809,
    'Turing' : 1912
}

print 'Before deleting Turing:', birthday
del birthday['Turing']
print 'After deleting Turing:', birthday
del birthday['Faraday']
print 'After deleting Faraday:', birthday
```

```
Before deleting Turing: {'Turing': 1912, 'Newton': 1642, 'Darwin': 1809}
After deleting Turing: {'Newton': 1642, 'Darwin': 1809}
```

```
Traceback (most recent call last):
  File "dict_del.py", line 10, in ?
    del birthday['Faraday']
KeyError: 'Faraday'
```



Membership and Loops

- Test whether a key `k` is in a dictionary `d` using `k in d`
 - Once again, inconsistent with behavior of lists, but useful

```
birthday = {  
    'Newton' : 1642,  
    'Darwin' : 1809  
}
```

```
for name in ['Newton', 'Turing']:  
    if name in birthday:  
        print name, birthday[name]  
    else:  
        print 'Who is', name, '?'
```

Newton 1642

Who is Turing ?



Membership and Loops

- for k in d loops over the dictionary's keys (rather than its values)
 - Different from lists, where for loops over the values, rather than indices

```
birthday = {  
    'Newton' : 1642,  
    'Darwin' : 1809,  
    'Turing' : 1912  
}  
for name in birthday:  
    print name, birthday[name]
```

```
Turing 1912  
Newton 1642  
Darwin 1809
```



Dictionary Methods

- Yes, dictionaries are objects too...

Method	Purpose	Example	Result
clear	Empty the dictionary.	<code>d.clear()</code>	Returns None, but d is now empty.
get	Return the value associated with a key, or a default value if the key is not present.	<code>d.get('x', 99)</code>	Returns <code>d['x']</code> if "x" is in d, or 99 if it is not.
keys	Return the dictionary's keys as a list. Entries are guaranteed to be unique.	<code>birthday.keys()</code>	<code>['Turing', 'Newton', 'Darwin']</code>
items	Return a list of (key, value) pairs.	<code>birthday.items()</code>	<code>[('Turing', 1912), ('Newton', 1642), ('Darwin', 1809)]</code>
values	Return the dictionary's values as a list. Entries may or may not be unique.	<code>birthday.values()</code>	<code>[1912, 1642, 1809]</code>
update	Copy keys and values from one dictionary into another.	See the example below.	



Dictionary Methods

```
birthday = {
    'Newton' : 1642,
    'Darwin' : 1809,
    'Turing' : 1912
}

print 'keys:', birthday.keys()
print 'values:', birthday.values()
print 'items:', birthday.items()
print 'get:', birthday.get('Curie', 1867)

temp = {
    'Curie' : 1867,
    'Hopper' : 1906,
    'Franklin' : 1920
}
birthday.update(temp)
print 'after update:', birthday

birthday.clear()
print 'after clear:', birthday

keys: ['Turing', 'Newton', 'Darwin']
values: [1912, 1642, 1809]
items: [('Turing', 1912), ('Newton', 1642), ('Darwin', 1809)]
get: 1867
after update: {'Curie': 1867, 'Darwin': 1809, 'Franklin': 1920, 'Turing': 1912, 'Newton': 1642, 'Hopper': 1906}
after clear: {}
```



Counting Frequency

- So, back to our birds...
 - Use species names as keys in a dictionary
 - The value associated with each key is the number of times it has been seen so far

```
# Data to count.
names = ['tern','goose','goose','hawk','tern','goose', 'tern']

# Build a dictionary of frequencies.
freq = {}
for name in names:

    # Already seen, so increment count by one.
    if name in freq:
        freq[name] = freq[name] + 1

    # Never seen before, so add to dictionary.
    else:
        freq[name] = 1

# Display.
print freq

{'goose': 3, 'tern': 3, 'hawk': 1}
```



A Slight Simplification

- Can simplify this code using dict.get
 - Get either the count associated with the key, or 0, then add one to it

```
freq = {}  
for name in names:  
    freq[name] = freq.get(name, 0) + 1  
print freq
```

```
{'goose': 3, 'tern': 3, 'hawk': 1}
```



Imposing Order

- A dictionary's keys are unordered (just like the elements in a set)
 - Remember, we deliberately randomize (hash) in order to make lookup fast
- So, to print counts in alphabetic order:
 - Get the list of keys
 - Sort that list
 - Loop over it

```
keys = freq.keys()
keys.sort()
for k in keys:
    print k, freq[k]
```

```
goose 3
hawk 1
tern 3
```



Inverting a Dictionary

- But how to print in order of frequency?
 - Need to *invert* the dictionary
 - I.e., swap the keys and values
- But there might be collisions, since values aren't guaranteed to be unique
 - What is the inverse of {'a':1, 'b':1, 'c':1}?
- Solution: store a list of values instead of just a single value
 - Use `dict.get(key, [])` instead of `dict.get(key, 0)`

Inverting a Dictionary

```
inverse = {}  
for (key, value) in freq.items():  
    seen = inverse.get(value, [])  
    seen.append(key)  
    inverse[value] = seen
```

```
keys = inverse.keys()  
keys.sort()  
for k in keys:  
    print k, inverse[k]
```

```
1 ['hawk']  
3 ['goose', 'tern']
```

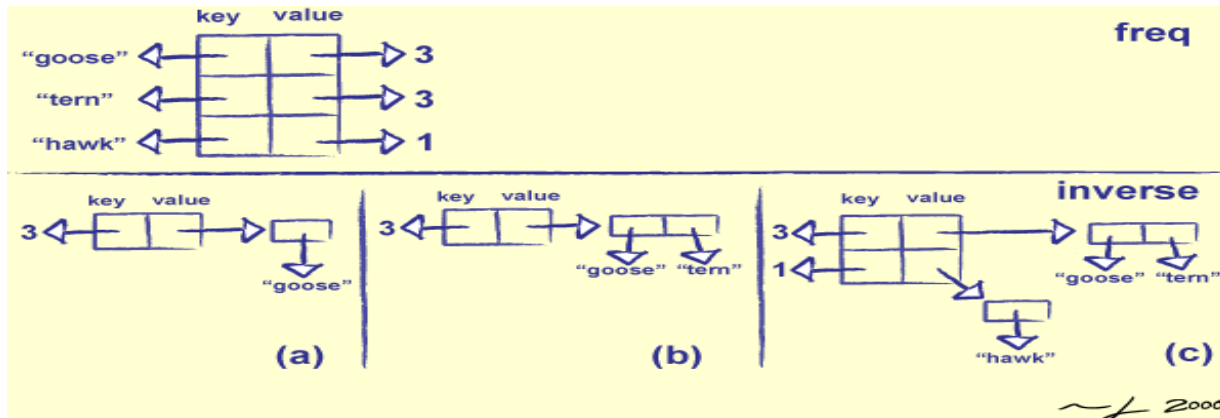


Figure 9.7: Inverting a Dictionary



Another Way to Do It

- The previous example can also be written as:

```
inverse = {}
```

```
for (key, value) in freq.items():
```

```
    if value not in inverse:
```

```
        inverse[value] = []
```

```
        inverse[value].append(key)
```

- I.e., store an empty list when needed, and always append

Formatting Strings with Dictionaries

- Complex string formatting can be hard to understand
 - Especially if one value needs to be used several times
- Instead of a tuple, "%" can take a dictionary as its right argument
 - Use "%(varname)s" inside the format string to identify what's to be substituted

```
birthday = {
    'Newton' : 1642,
    'Darwin' : 1809,
    'Turing' : 1912
}
entry = '%(name)s: %(year)s'
for (name, year) in birthday.items():
    temp = {'name' : name, 'year' : year}
    print entry % temp
```

```
Turing: 1912
Newton: 1642
Darwin: 1809
```



Extra Keyword Arguments

- Consider this example:

```
def settings(title, **kwargs):
    print 'title:', title
    for key in kwargs:
        print '    %s: %s' % (key, kwargs[key])

settings('nothing extra')
settings('colors', red=0.0, green=0.5, blue=1.0)

title: nothing extra
title: colors
    blue: 1.0
    green: 0.5
    red: 0.0
```

- The ** in front of kwargs means "Put any extra keyword arguments in a dictionary, and assign it to kwargs"
- Allows you to create functions that can handle arbitrary arguments



Extra Positional Arguments

- Can do something similar with extra positional (unnamed) arguments

```
def sum(*values):  
    result = 0.0  
    for v in values:  
        result += v  
    return result
```

```
print "no values:", sum()  
print "single value:", sum(3)  
print "five values:", sum(3, 4, 5, 6, 7)
```

```
no values: 0.0  
single value: 3.0  
five values: 25.0
```

- The single * in front of values means "Put any extra unnamed arguments in a tuple, and assign it to values"
- Can have at most one * argumen per function
- Question: what does ** mean? How and why would you use it?



Summary

- The world isn't made of lists
- Other basic data structures can make your programs much simpler, and much more efficient
- And learning a few advanced features of whatever language you're using can do the same