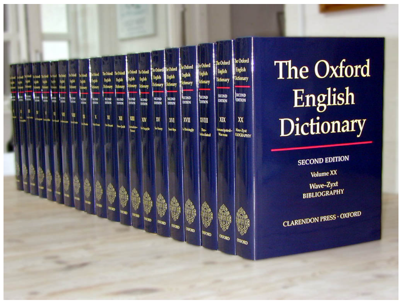
**CS216: Program and Data Representation**
University of Virginia Computer Science
Spring 2006 David Evans

# Lecture 24: Fast Dictionaries




http://www.cs.virginia.edu/cs216

---

## CS216 Roadmap

Data Representation
Program Representation

Rest of CS216

Real World Problems

"Hello"
['H','i',\0]
0x42381a,

Objects
Arrays
Addresses, Numbers

01

Python code
C code
JVML
**x86**

High-level language
Low-level language
Virtual Machine language
**Assembly**

Note: depending on your answers to the topic interest exam question, we might also look at another VM (CLR) or another assembly language (RISC)

Real World Physics



---

## Fast Dictionaries

- Problem set 2, question 5...

  "You may assume Python's dictionary type provides lookup and insert operations that have running times in $O(1)$."

- Class 6: fastest possible search using binary comparator is $O(\log n)$

  Can Python really have an $O(1)$ lookup?



---

## Fast Dictionaries

Data Representation

- If the keys can be anything?

  No – best one comparison can do is eliminate ½ the elements

  The keys must be bits, so we can do better!

"Hello" – Objects
['H','i',\0] – Arrays
0x42381a, 3.14, 'x'
01001010 – Bits



---

## Lookup Table

| Key | Value |
|---|---|
| 000000 | "red" |
| 000001 | "orange" |
| 000010 | "blue" |
| 000011 | null |
| 000100 | "green" |
| 000101 | "white" |
| ... | ... |

Works great...unless the key space is sparse.



---

## Sparse Lookup Table

- Keys: names (words of up to 40 7-bit ASCII characters)
- How big a table do we need?

  40 * 7 = 280
  $2^{280}$ = ~$1.9*10^{84}$ entries

We need lookup tables where many keys can map to the same entry

## Hash Table

- Hash Function:

  $h$: $Key \rightarrow [0, m\text{-}1]$

Here:
$h$ = firstLetter($Key$)

| Location | Key | Value |
|---|---|---|
| 0 | "Alice" | "red" |
| 1 | "Bob" | "orange" |
| 2 | "Coleen" | "blue" |
| 3 | null | null |
| 4 | "Eve" | "green" |
| 5 | "Fred" | "white" |
| … | … | … |
| $m$-1 | "Zeus" | "purple" |



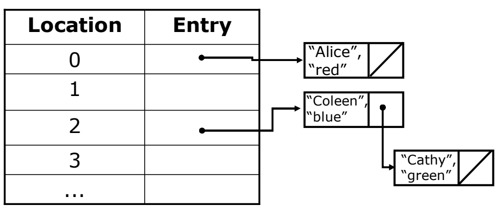
## Collisions

- What if we need both "Colleen" and "Cathy" keys?



## Separate Chaining

- Each element in hash table is not a <key, value> pair, but a list of pairs






## Hash Table Analysis

- Lookup Running Time?

Worst Case: $\Theta(N)$
$N$ entries, all in same bucket
Hopeful Case: $O(1)$
Most buckets with < $c$ entries



## Requirements for "Hopeful" Case

- Function $h$ is well distributed for key space

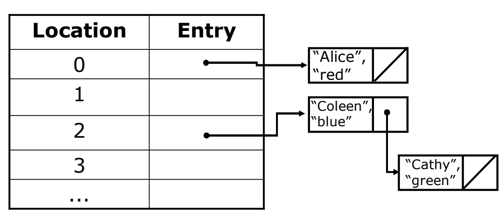
  for a randomly selected $k \in K$,
  probability $(h(k) = i) = 1/m$

- Size of table ($m$) scales linearly with $N$
  - Expected bucket size is $\Theta(N / m)$

Finding a good $h$ can be tough (more later…)



## Saving Memory




Can we avoid the overhead of all those linked lists?

## Linear Open Addressing

| Location | Key | Value |
|---|---|---|
| 0 | "Alice" | "red" |
| 1 | "Bob" | "orange" |
| 2 | "Coleen" | "blue" |
| 3 | "Cathy" | "yellow" |
| 4 | "Eve" | "green" |
| 5 | "Fred" | "white" |
| 6 | "Dave" | "red" |
| … | | |



## Sequential Open Addressing

```
def lookup (T, k):
  i = hash (k)
  while (not looped all the way around):
    if T[i] == null:
      return null
    else if T[i].key == k:
      return T[i].value
    else
      i = i + 1 mod T.length
```



## Problems with Sequential

- "Primary Clustering"
  - Once there is a full chunk of the table, anything hash in that chunk makes it grow
  - Note that this happens even if $h$ is well distributed
- Improved strategy?

  Don't look for slots sequentially

  i = i + s mod T.length

Doesn't help – just makes clusters appear scattered



## Double Hashing

- Use a second hash function to look for slots

  i = i + hash2 (K) mod T.length
- Desirable properties of hash2:
  - Should eventually try all slots

    result of hash2(K) should be relatively prime to m
    (Easiest to make m prime)
  - Should be independent from hash



## Good Hash Functions

- Deterministic
- Arbitrary fixed-size output
- Easy to compute
- Well-distributed

for a randomly selected $k \in K$,
probability $(h(k) = i) = 1/m$



## Reasonable Hash Functions?

- Just take the first $\log m$ bits
- Just take the lowest $\log m$ bits
- Sum all key characters

$$\text{hash} = \sum_{i \text{ in indexes}(k)} k_i \bmod m$$

- PS6 Mystery code (SHA-1)

# What does Python do?

```
long
PyObject_Hash(PyObject *v)
{
        PyTypeObject *tp = v->ob_type;
        if (tp->tp_hash != NULL)
                return (*tp->tp_hash)(v);
        if (tp->tp_compare == NULL && RICHCOMPARE(tp) == NULL) {
                return _Py_HashPointer(v); /* Use address as hash value */
        }
        /* If there's a cmp but no hash defined, the object can't be hashed */
        PyErr_SetString(PyExc_TypeError, "unhashable type");
        return -1;
}
```

Types can have their own hash functions

Python-2.4/Objects/object.c



# _Py_HashPointer

```
long
_Py_HashPointer(void *p)
{
#if SIZEOF_LONG >= SIZEOF_VOID_P
        return (long)p;
#else
        /* convert to a Python long and hash that */
        PyObject* longobj;
        long x;
        if ((longobj = PyLong_FromVoidPtr(p)) == NULL) {
                x = -1;
                goto finally;
        }
        x = PyObject_Hash(longobj);
finally:
        Py_XDECREF(longobj);
        return x;
#endif
}
```

What does this mean for Python's garbage collector?



# Dictionary

```
/*
Major subtleties ahead:  Most hash schemes depend on having
a "good" hash function, in the sense of simulating randomness.
Python doesn't:  its most important hash functions (for strings
and ints) are very regular in common cases:
>>> map(hash, (0, 1, 2, 3))
[0, 1, 2, 3]
>>> map(hash, ("namea", "nameb", "namec", "named"))
[-1658398457, -1658398460, -1658398459, -1658398462]
>>>

This isn't necessarily bad!  ...
```

Python-2.4/Objects/dictobject.c



To the contrary, in a table of size 2**i, taking the low-order i bits as the initial table index is extremely fast, and there are no collisions at all for dicts indexed by a contiguous range of ints. The same is approximately true when keys are "consecutive" strings. So this gives better-than-random behavior in common cases, and that's very desirable.

OTOH, when collisions occur, the tendency to fill contiguous slices of the hash table makes a good collision resolution strategy crucial.



# Collision Avoidance

Taking only the last i bits of the hash code is also vulnerable: for example, consider
[i << 16 for i in range(20000)]
as a set of keys. Since ints are their own hash codes, and this fits in a dict of size 2**15, the last 15 bits of every hash code are all 0: they *all* map to the same table index.



# Impact

Scott Crosby, Dan Wallach. *Denial of Service via Algorithmic Complexity Attacks*. USENIX Security 2003.

Python Example:
10,000 inputs in dictionary:
Expected case: 0.2 seconds
Collision-constructed case: 20 seconds

http://www.cs.rice.edu/~scrosby/hash/

From: Guido van Rossum guido@python.org
Subject: **[Python-Dev] Algoritmic Complexity Attack on Python**
Date: *Fri, 30 May 2003 07:39:18 -0400*
[Tim Peters]
*> I'm uninterested in trying to "do something" about these. If*
*> resource-hogging is a serious potential problem in some context, then*
*> resource limitation is an operating system's job, ...*
[Scott Crosby]
*> I disagree. Changing the hash function eliminates these attacks...*

At what cost for Python? 99.99% of all Python programs are not vulnerable to this kind of attack, because they don't take huge amounts of arbitrary input from an untrusted source. If the hash function you propose is even a *teensy* bit slower than the one we've got now (and from your description I'm sure it has to be), everybody would be paying for the solution to a problem they don't have. You keep insisting that you don't know Python. Hashing is used an awful lot in Python -- as an interpreted language, most variable lookups and all method and instance variable lookups use hashing. So this would affect every Python program.

Scott, we thank you for pointing out the issue, but I think you'll be wearing out your welcome here quickly if you keep insisting that we do things your way based on the evidence you've produced so far.
--Guido van Rossum

http://mail.python.org/pipermail/python-dev/2003-May/035874.html



# Charge

- Start thinking about what you want to do for PS8 now
- You will receive an email by tomorrow night