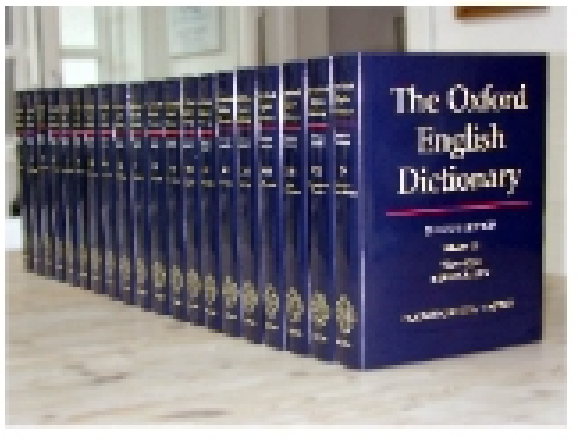
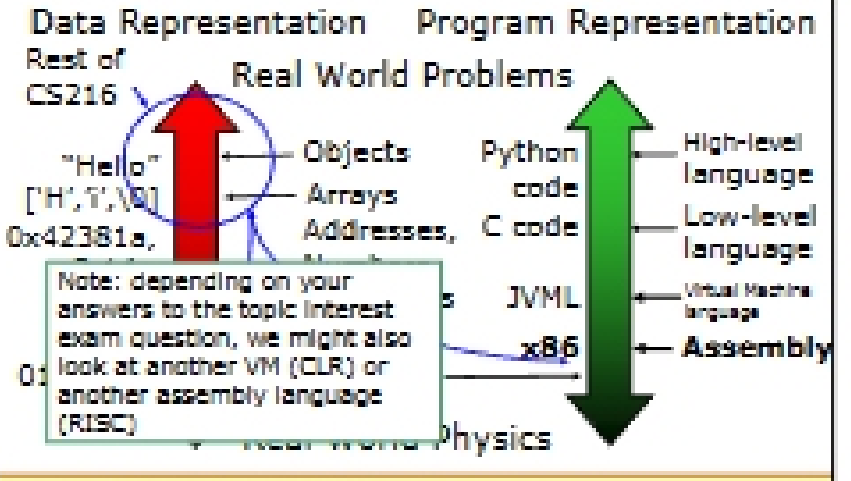
CS216: Program and Data Representation
University of Virginia Computer Science
Spring 2006 David Evans

Lecture 24: Fast Dictionaries



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

CS216 Roadmap



UVa CS216 Spring 2006 - Lecture 23: Fast Dictionaries 2

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?

UVa CS216 Spring 2006 - Lecture 23: Fast Dictionaries 3

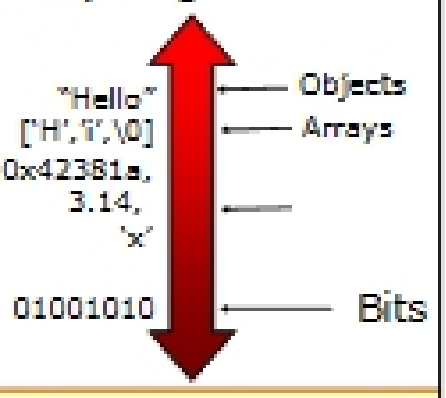
Fast Dictionaries

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!



UVa CS216 Spring 2006 - Lecture 23: Fast Dictionaries 4

Lookup Table

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

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

UVa CS216 Spring 2006 - Lecture 23: Fast Dictionaries 5

Sparse Lookup Table

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

$40 * 7 = 280$

$2^{280} = \sim 1.9 * 10^{84}$ entries

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

UVa CS216 Spring 2006 - Lecture 23: Fast Dictionaries 6

Hash Table

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

Here:
$h = \text{firstLetter}(Key)$

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

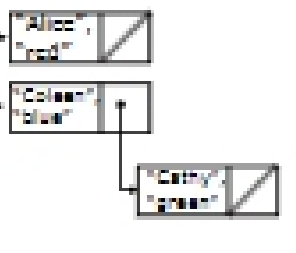
Collisions

- 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

Location	Entry
0	
1	
2	
3	
…	



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

Location	Entry
0	
1	
2	
3	
…	

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

$$\text{hash} = \Sigma\, k_i \bmod m$$

i in indices(k)

- PS6 Mystery code (SHA-1)