Computer Programming

Hashing. Randomization. Timing

Marius Minea

marius@cs.upt.ro

9 January 2016

Looking up values

Lookup or *search* is a basic, widely encountered problem

1) find whether we have seen a value before
= *set*: store all values seen so far

2) retrieve information associated to some identifier k (called *key*)
= *function* $f(k)$ from *key* k to *value* (info)
implemented as *map* (dictionary, association): (key, value) pair

in some languages, maps are primitive data types
others have libraries

Hashing. Hash Tables

used for *maps* from arbitrary *keys* to *values*
arrays only work when keys are integers (in a given range)

Idea: find a function `h` with an integer value in a restricted range (usable as index in an array)
Every object (key) `x` is stored in array at index `h(x)`
(usually, `h(x)` modulo table size)

Objects with different hash value are surely different
Different objects may have same hash value (collision)

Hash functions

need to be fast (easily computed), mixing all bytes of the object

have few *collisions* (esp. for objects with close/related values)

clearly, collisions cannot be avoided if domain larger than range

Examples for strings:

```
for (h=len; len--;) h = ((h<<7) ^ (h<<27)) ^ *s++; // Knuth
for (h=5381; c=*s++; ) h += (h << 5) + c;     // Bernstein
for (h=0; c=*s++; ) h = (h<<6) + (h<<16) - h + c; // SDBM
```

Hash functions usually return (32-bit) integers

Cryptographic functions need more stringent randomness properties and have larger bit width ($\geq$ 224 bits currently)

Open and closed hashing

Closed hashing

if a different object is found at index `idx=h(x)`, continue search using a sequence of indices:

sequential: `idx++`, linear: `idx+=i`

with another hash function: `idx+=h2(x)` until element found

when table fills up, objects must be re-hashed

deleted objects must be marked ($\neq$ *empty*) to stop useless search

Open hashing

entry in hash table is (linked) *list* of objects with same hash value

$\Rightarrow$ hashing followed by linear search in (hopefully short) list

need dynamic allocation for list elements

hash table size comparable to element count (avoid long lists)

Cuckoo hashing (Pagh & Rodler 2001)

constant-time lookup
amortized constant-time insert

Each key may be found in one of *two* locations (use two different hash functions)

On collision, displace existing key to 2$^{\text{nd}}$ location; if that location is full, successively displaced

If a cycle is reached, rebuild (larger) table

Works well up to $\sim$ 50% fill factor

Arrows in figure show alternate location for a key

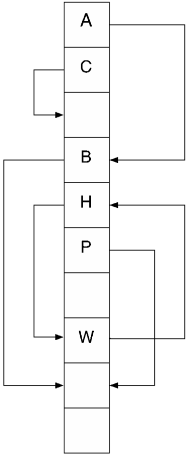

Image: `http://en.wikipedia.org/wiki/File:Cuckoo.svg`

Date and time (time.h)

time.h contains structures and functions to measure time

clock_t and time_t are real types representing times

struct tm holds a broken-down calendar time (sec, min, ... year)
struct timespec holds time in seconds and nanoseconds

```
clock_t clock(void);
```

returns (approximation) of processor time used
divide by CLOCKS_PER_SEC (usually 10^6) to get time in seconds

```
int timespec_get(struct timespec *ts, int base);
```

gives time in s and ns since a reference point base (use TIME_UTC)

```
struct timespec {
  time_t tv_sec;
  long tv_nsec;
};
```

Measuring time

Place the code to be benchmarked in a loop running many times
total time: order of seconds (account for limited clock precision)

Ensure compiler doesn't optimize away repetition (check assembly)
e.g. computing/assigning the same value many times
may need to use `volatile` specifier for variables
(forces writing/reading to memory every time, like in source)

Repeat measurements and make an average.

Time may be affected by other running processes, caching, etc.

Pseudo-random numbers (`stdlib.h`)

Only natural phenomena can be truly random.
Computer uses algorithm to generate numbers ⇒ *pseudo-random*
  period of number generator should be high
  all bits should appear to be random

Quality of `stdlib` random number generator may not be high
  (esp. for lower bits)
Need to use special RNG in cryptography applications.

```
int rand(void);
```

returns an integer in range 0 to `RAND_MAX` (at least $2^{15} - 1$)
Re-running program will produce the same sequence of numbers!
  ⇒ need to initialize state of RNG with a *seed*

```
void srand(unsigned int seed);
```

could use calendar time (seconds) as seed – different in each run
e.g. `srand((unsigned)time(NULL));`