COMP2521 19T0
Week 8, Tuesday: Hash Tables

Jashank Jeremy

jashank.jeremy@unsw.edu.au

hashing
performance

Hash Tables

Searching

The State Of Play

So far we've seen...
linked list: insert $O(1)$, search $O(n)$
ordered linked list: insert $O(n)$, search $O(n)$
array: insert $O(1)$, search $O(n)$
ordered array: insert $O(n)$, search $O(\log n)$
search tree: insert $O(\log n)$, search $O(\log n)$

... but these are still all pretty slow,
and perform less-than-ideally on modern architectures
(due to cache locality effects)

Searching

Even Faster?

In an ideal world, we can index on arbitrary keys,
and get constant-time $O(1)$ access.

Key-indexed arrays get some of the way there,
but have downsides:
... requires dense range of index values
... uses fixed-size array; sizing it is hard
... can't use arbitrary keys!

Hashing

Hashing lets us approximate this:
arbitrary keys! (so long as we can hash them)
map keys into a compact range of index values!
store items in array, accessed by index value!
$O(1)$!

Hashing

Hashing lets us approximate this:
arbitrary keys! (so long as we can hash them)
map keys into a compact range of index values!
store items in array, accessed by index value!
$O(1)$!

We need three things:
an array of Items, of size N

Hashing

Hashing lets us approximate this:
arbitrary keys! (so long as we can hash them)
map keys into a compact range of index values!
store items in array, accessed by index value!
$O(1)$!

We need three things:
an array of Items, of size N
a hash function,
$\textsc{hash} :: \texttt{Key} \rightarrow \texttt{size} \rightarrow [0 \cdots N)$,

Hashing

Hashing lets us approximate this:
arbitrary keys! (so long as we can hash them)
map keys into a compact range of index values!
store items in array, accessed by index value!
$O(1)$!

We need three things:
an array of Items, of size N
a hash function,
$\textsc{hash} :: \texttt{Key} \to \texttt{size} \to [0 \cdots N)$,
a collision resolution method,
for when $k_1 \neq k_2 \wedge \textsc{hash}\,(k_1, N) = h\,(k_2, N)$;
collisions are inevitable when $\textsc{dom}\,(k) \gg N$

Hash Functions

Properties we want h to have:

- for a table of size N, output range is 0 to $N - 1$;
- pure, deterministic: $h(k, N)$ gives the same result;
- spreads key values uniformly over index range
  (assuming keys are unformly distributed)
- cheap (enough) to compute … otherwise, what's the point?

Hash Functions

Aside: Cryptographic Hash Functions

Ideally all of the above, *and*
pre-image resistant:
for $h = \text{HASH}(m)$, given h, hard to pick m;
second pre-image resistant:
for $\text{HASH}(m_1) = \text{HASH}(m_2)$,
given m_1, hard to find $m_2 \neq m_1$;
collision resistant:
for $\text{HASH}(m_1) = \text{HASH}(m_2)$,
hard to find m_1 and m_2.

For our purposes, we don't need cryptographic hash functions.
(COMP6[48]41, MATH3411 go into detail.)

Hash Functions

Example (I)

A simple hash function for single characters, if $N = 128$:

```
size_t hash (char key, size_t N)
{
    return key; // N redundant
}
```

Not really useful:
key range is usually much larger than N.

Hash Functions

Example (II)

Another simple hash function, for integers:

```
size_t hash (int key, size_t N)
{
    return key % N;
}
```

How big is N?
small $N \Rightarrow$ too many collisions!

Hash Functions

Example (III)

A simple hash function, for strings:

```
size_t hash (char *key, size_t N)
{
    return strlen (key) % N;
}
```

(You should never *actually* do this.)

Hash Functions

Example (IV)

A better string hash function:

```
size_t hash (char *key, size_t N)
{
    size_t h = 0;
    for (size_t i = 0; key[i] != '\0'; i++)
        h += key[i];
    return h % N;
}
```

Hash Functions

Example (V)

A more sophisticated hash function:

```
size_t hash (char *key, size_t N)
{
    size_t h = 0;
    unsigned a = 127; // prime
    for (size_t i = 0; key[i] != '\0'; i++)
        h = ((a * h) + key[i]) % N;
    return h;
}
```

Hash Functions

Example (VI)

Using *universal hashing*,
which introduces randomness while using the entire key:

```c
size_t hash (char *key, size_t N)
{
    size_t h = 0;
    unsigned a = 31415, b = 21783;
    for (size_t i = 0; key[i] != '\0'; i++) {
        a = (a * b) % (N - 1);
        h = ((a * h) + key[i]) % N;
    }
    return h;
}
```

Collisions

What happens if two keys hash the same?
We go to the same array index ... then what?

... allow multiple Items in a single location,
via *e.g.*, array of item arrays
array of linked lists

... systematically compute new indices
by various *probing* strategies

... resize the array
by adjusting the hash function,
and moving everything (!)

Collisions

Separate Chaining

Given N slots and M items:
best case, all lists have length M/N
worst case, one list with length M, all others 0

with a good hash and $M \leq N$, cost $O(1)$;
with a good hash and $M > N$, cost $O(M/N)$

(The M/N ratio is called *load*.)

Collisions

Linear Probing

If the table is not close to being full,
there are still many empty slots;
we could just use the next available slot along;
open-address hashing.

to reach the first item is $O(1)$;
search for subsequent items depends on load;
successful search cost: $\frac{1}{2}\left(1 + 1/\left(1-\alpha\right)\right)$
unsuccessful search cost: $\frac{1}{2}\left(1 + 1/\left(1-\alpha\right)^2\right)$
(assuming reasonably uniform data, good hash function)

... but tends towards $O(N)$ when α is high.

Collisions

Double-Hash Probing

We switch from $\textsc{hash}$ to $\textsc{hash2}$
(which should not return 0!),
and use it as the step to the 'next' item.
$\textsc{hash}$ and $\textsc{hash2}$ should be
relatively prime to each other, and to N.
(Easy, if we pick a prime N.)

Significantly faster than linear probing for high α

Hash Table Performance

Choosing a good HASH is critical.
Choosing a good N for M is critical.
Choosing a good resolution approach is critical.

linear probing: fastest, given big N!
double hashing: fastest for higher α, more efficient
chaining: possible for $\alpha \geq 1$, but degenerates

Performance

Why do we care, anyway?

good performance $\Rightarrow$ less hardware, happy users.
bad performance $\Rightarrow$ more hardware, unhappy users.

generally, performance is proportional to execution time;
we may be interested in other things (memory, i/o, ...)

Premature
optimisation

is the root

of all evil.

Developing Efficient Programs

1. Design the program well[1]
2. Implement the program well[2]
3. Test the program well
4. Only after you're sure it's working, measure performance
5. If (and only if) performance is inadequate, find the 'hot spots'
6. Tune the code to fix these
7. Repeat measure-analyse-tune cycle until performance ok

[1] See, e.g., *Algorithms* by Sedgewick, *Algorithms* by Cormen/Leierson/Rivest/Stein.
[2] See, e.g., *Programming Pearls*, *the Practice of Programming*.

Performance Estimates

Complexity analysis give info on most appropriate algorithm.
We can also consider an experimental approach to performance:

- determine the *critical operations* in the program
- determine *classes of input* data and *likelihood* of each
- *estimate* the cost (#crit.ops) for each class of data
- produce a weighted sum estimate for *overall cost*

Performance Estimates

Complexity analysis give info on most appropriate algorithm.
We can also consider an experimental approach to performance:

- determine the *critical operations* in the program
- determine *classes of input* data and *likelihood* of each
- *estimate* the cost (#crit.ops) for each class of data
- produce a weighted sum estimate for *overall cost*

Often, however...

- assumptions made in estimating performance are invalid
- we overlook some frequent and/or expensive operation

Performance Measurements

Basis of performance evaluation:
measure program execution.

empirical study suggests the ‘80/20’ rule:
most programs spend
most of their execution time
in a small part of their code.

most code has little impact on overall performance
small parts account for most execution time

To improve performance: focus on *bottlenecks* first.

Performance Measurement

Profiling Execution

We need a way to measure how much
each block of code costs:
a *profiler*.

gprof(1) displays execution profiles
for programs compiled with `-pg`;
profiling info is left behind in `gmon.out`,
which *gprof(1)* can read.

gprof(1) gives a table (a *flat profile*) containing:
number of times each function was called,
% of total execution time spent in the function,
average execution time per call to that function,
execution time for this function and its children

Performance Improvement

Once you have a profile, you can identify hot points.
To improve the performance:

- change the algorithm and/or data-structures
  - may give orders-of-magnitude better performance
  - but it is extremely costly to rebuild the system
- use simple efficiency tricks to reduce costs
  - may improve performance by one order-of-magnitude
- use the compiler's optimization switches (e.g., `-O`, `-O2`, `-O3`)
  - may improve performance by one order-of-magnitude

Conclusions

Time and profile your code
only when you are done.

Don’t optimise code unless you have to.
(You almost never will.)

Fixing your algorithm is
almost always the solution

Using compiler optimisations
is usually good enough.