Hash Tables

Prof. Clarkson
Fall 2015

Today's music: *Re-hash* by Gorillaz

Review

Current topic: Reasoning about performance

- Efficiency
- Big Oh
- Amortized analysis

Today:

- Implementation and efficiency analysis of hash tables

Question

How often do you dictionaries/maps/hash tables/associative arrays/etc. in your own programming?

A. Never
B. Infrequently
C. Frequently
D. Nearly every program I write
E. Compilers

Maps*

```
module type MAP = sig
  type ('k, 'v) map
  val insert:
    'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
  val find:
    'k -> ('k, 'v) map -> 'v option
  val remove:
    'k -> ('k, 'v) map -> ('k, 'v) map
  ...
end
```

*aka associative array, dictionary, symbol table

Mutable Maps*

```
module type MAP = sig
  type ('k, 'v) map
  val insert:
    'k -> 'v -> ('k, 'v) map -> unit
  val find:
    'k -> ('k, 'v) map -> 'v option
  val remove:
    'k -> ('k, 'v) map -> unit
  ...
end
```

*aka associative array, dictionary, symbol table

Map implementations

- Immutable:
  - Association lists
  - Balanced search trees
- Mutable
  - Arrays
  - Hash tables

Map implementations

For each implementation:

- What is the representation type?
- What is the abstraction function?
- What are the representation invariants (if any)?
- What is the efficiency of each operation?

Association lists

- Representation type:

```
type ('k, 'v) map = ('k*'v) list
```

- Abstraction function:
  - A list `[(k1,v1); (k2,v2); ...]` represents the map `{k1=v1, k2=v2, ...}`.
  - If k occurs more than once in the list, then in the map it is bound to the left-most value in the list.
- Efficiency:
  - insert: O(1)
  - find: O(n)
  - remove: O(n)

Balanced search trees

2-3 trees:

- Representation type: (omitted; see A3)
- Abstraction function: a node with label `(k,v)` and subtrees `left` (and `middle`) and `right` represents the smallest map containing the binding `{k=v}` unioned with the bindings of `left` (and `middle`) and `right`
- Representation invariant:
  - none for the map itself, but note that the tree has its own *2-3 tree invariants*
- Efficiency:
  - insert: O(lg n)
  - find: O(lg n)
  - remove: O(lg n)
- OCaml's `Map` module uses a closely-related balanced search tree called *AVL trees*

Arrays

- Representation type:

```
type ('k, 'v) map = 'v option array
```

- Assume we can convert **`'k`** to **`int`** in constant time
  - Conversion must be *injective*: never maps two keys to the same integer
  - Then there is a unique *inverse* mapping integers to keys: inverse(i) = k
  - Easiest realization: restrict keys to be integers!

Arrays

- Abstraction function: An array `[|v1; v2; ...|]` represents the map `{k1=v1, k2=v2, ...}`, where `k1=inverse(1), k2=inverse(2), ...` If `vi = None`, then `ki` is not bound in the map.
- Aka *direct address table*
- Efficiency:
  - insert: O(1)
  - find: O(1)
  - remove: O(1)
  - wastes space, because some keys are unmapped

Map implementations

	insert	find	remove
Arrays	O(1)	O(1)	O(1)
Association lists	O(1)	O(n)	O(n)
Balanced search trees	O(lg n)	O(lg n)	O(lg n)

- Arrays guarantee constant efficiency, but require injective conversion of keys to integers (and waste space)
- Balanced search trees guarantee logarithmic efficiency

…we'd like the best of both worlds:

constant efficiency with arbitrary keys

Hash tables

Main idea: give up on injectivity

- Allow conversion from `'k` to `int` to map multiple keys to the same integer
- Conversion function called a *hash* function
- Location it maps to called a *bucket*
- When two keys map to the same bucket, called a *collision*

...how to handle collisions?

Collision resolution strategies

1. Store multiple key-value pairs in a collection at a bucket; usually the collection is a list
   - called *open hashing, closed addressing, separate chaining*
   - this is what OCaml's **`Hashtbl`** does
2. Store only one key-value pair at a bucket; if bucket is already full, find another bucket to use
   - called *closed hashing, open addressing*

Hash table implementation

- Representation type combines association list with array:

```
type ('k, 'v) map = ('k*'v) list array
```

- Abstraction function: An array

```
[|[(k11,v11); (k12,v12);...];
  [(k21,v21); (k22,v22);...]; ...|]
```

  represents the map `{k11=v11, k12=v12, ...}`.
  - If k occurs more than once in a bucket, then in the map it is bound to the left-most value in the bucket.
- Representation invariant:
  - A key `k` appears in array index `b` iff `hash(k)=b`
- Efficiency: ???
  - have to search through list to find key
  - no longer constant time

Question

Why does the representation type need to contain the `'k`?

```
type ('k, 'v) map =
     ('k*'v) list array
```

A. The type system requires it
B. A given bucket might contain many keys
C. To support an inverse operation
D. The hash table representation invariant requires it
E. None of the above

Question

Why does the representation type need to contain the `'k`?

```
type ('k, 'v) map =
     ('k*'v) list array
```

A. The type system requires it
B. A given bucket might contain many keys
C. To support an inverse operation
D. The hash table representation invariant requires it
E. None of the above

Efficiency of hash table

- Terrible hash function: `hash(k) = 42`
  - All keys collide; stored in single bucket
  - Degenerates to an association list in that bucket
    - insert: O(1)
    - find & remove: O(n)
- Perfect hash function: injective
  - Each key in its own bucket
  - Degenerates to array implementation
    - insert, find & remove: O(1)
  - Surprisingly, possible to design
    - if you know the set of all keys that will ever be bound in advance
    - size of array is the size of that set
    - so you want the size of the set to be much smaller than the size of the universe of possible keys
- Middleground? Compromise?

Efficiency of hash table

- New goal: constant-time efficiency on average
  - Desired property of hash function: distribute keys randomly among buckets to keep average bucket length small
  - If expected length is on average L:
    - insert: O(1)
    - find & remove: O(L)
- Two new problems to solve:
  1. How to make L a constant that doesn't depend on number of bindings in table?
  2. How to design hash function that distributes keys randomly?

Independence from # bindings

Let's think about the *load factor*...

= average number of bindings in a bucket = expected bucket length

= n/m, where n=# bindings in hash table, m=# buckets in array

- *e.g.*, 10 bindings, 10 buckets, load factor = 1.0
- *e.g.*, 20 bindings, 10 buckets, load factor = 2.0
- *e.g.*, 5 bindings, 10 buckets, load factor = 0.5

- Both OCaml **`Hashtbl`** and **`java.util.HashMap`** provide functionality to find out current load factor
- Implementor of hash table can't prevent client from adding or removing bindings
  - so n isn't under control
- But can *resize* array to be bigger or smaller
  - so m can be controlled
  - hence load factor can be controlled
  - hence expected bucket length can be controlled

Control the load factor

- If load factor gets too high, make the array bigger, thus reducing load factor
  - OCaml **`Hashtbl`** and **`java.util.HashMap`**: if load factor > 2.0 then:
    - double array size
    - rehash elements into new buckets
    - thus bringing load factor back to around 1.0
  - Efficiency on average for that strategy:
    - insert: O(1)
    - find & remove: O(2), which is still constant time
    - rehashing: let's come back to that...
- If load factor gets too small (hence memory is being wasted), could shrink the array, thus increasing load factor
  - Neither OCaml nor Java does this

Question

How would you resize this representation type?

```
type ('k, 'v) map =
  ('k*'v) list array
```

A. Mutate the array elements
B. Mutate the array itself
C. Neither of the above

Question

How would you resize this representation type?

```
type ('k, 'v) map =
  ('k*'v) list array
```

A. Mutate the array elements

B. Mutate the array itself *(can't—it's immutable)*

C. Neither of the above

Resizing the array

Requires a new representation type:

```
type ('k, 'v) map =
  ('k*'v) list array ref
```

- Mutate an array element to **`insert`** or **`remove`**
- Mutate array ref to resize

`Hashtbl` in OCaml library

```
type ('a, 'b) t =
  { mutable size: int;
    mutable data: ('a, 'b) bucketlist array;
    ... }

and ('a, 'b) bucketlist =
    Empty
  | Cons of 'a * 'b * ('a, 'b) bucketlist
```

Why not use **`list`**? *Probably to save on one indirection.*

Hash tables: physicist's method

- Simplifying assumptions:
  - no `remove` operation
  - ignore cost of all operations until load factor reaches 1 for the first time
- Potential: $U(h) = 4(n - m)$
  - where n is number of elements in h
  - and m is number of buckets in h
  - Causes potential to increase as load factor (=n/m) grows
  - When load factor is 1, it holds that m=n, so U(h) = 0
    - no extra credit stored up immediately after resize
  - When load factor is 2, it holds that m=n/2, so U(h) = 2n
    - enough extra credit stored up to pay to rehash and insert each element just when we need to resize

Hash tables: physicist's method

- Amortized cost of **`insert`** (including resize)
  - Let n be # elements and m be # buckets before insert
  - If no resize is triggered:
    - Actual cost of 1 each to hash and insert element
    - Change in potential = $4(n+1-m) - 4(n - m) = 4n + 4 - 4m - 4n + 4m = 4$
    - Amortized cost = actual + change = $1 + 1 + 4 = 6 = O(1)$

Hash tables: physicist's method

- Amortized cost of `insert` (including resize)
  - Let n be # elements and m be # buckets before insert
  - If resize is triggered:
    - Then $n+1 = 2m$
    - Actual cost of $2(n+1)$ to hash and insert $n+1$ elements
    - Change in potential $= 4(n+1-2m) - 4(n-m) = 4n + 4 - 8m - 4n + 4m = 4 - 4m = 4 - 2(2m) = 4 - 2(n+1) = 4 - 2n - 2$
    - Amortized cost = actual + change $= 2(n+1) + 4 - 2n - 2 = 2n + 2 + 4 - 2n - 2 = 4 = O(1)$
- Whether resize occurs or not, amortized cost of $O(1)$

Hash tables

Conclusion: resizing hash tables have *amortized expected worst-case running time* that is constant!

Upcoming events

- [Wed-Fri] No class: Happy Thanksgiving!
- [next Thursday] A6 (including Project Implementation) due

This is #3110.

THIS IS 3110