CS 3110

Hash Tables

Nate Foster
Spring 2018

A3

- Index and search books
- Extensive use of data structures and OCaml module system
- Competition for most scalable implementation
- Staff solution: ~500 LoC

Review

Previously in 3110:

- Streams
- Balanced trees
- Refs

Today:

- Hash tables

Maps

- Maps bind keys to values
- Aka associative array, dictionary, symbol table
- Abstract notation:

{k1 : v1, k2 : v2, ..., kn : vn}

e.g.

- {3110 : "Fun", 2110 : "OO"}
- {"Harvard" : 1636, "Princeton" : 1746, "Penn": 1740, “Williams” : 1792, "Cornell" : 1865}

Maps

```
module type Map = sig
  type ('k, 'v) t
  val insert:
    'k -> 'v -> ('k, 'v) t -> ('k, 'v) t
  val find:
    'k -> ('k, 'v) t -> 'v option
  val remove:
    'k -> ('k, 'v) t -> ('k, 'v) t
  ...
end
```

Map implementations

Up next: four implementations

For each implementation:

- What is the representation type?
- What is the abstraction function?
- What are the representation invariants?
- What is the efficiency of each operation?

FUNCTIONAL MAPS

Maps as lists

- Representation type:
  `type ('k, 'v) t = ('k*'v) list`

- AF:
  `[(k1,v1); (k2,v2); ...]` represents `{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. The empty list represents the empty map.

- RI: none

- Efficiency:
  - insert: cons to front of list: O(1)
  - find: traverse entire list: O(n)
  - remove: traverse entire list: O(n)

Maps as red-black trees

- Representation type:
  RB tree where nodes contain (key,value) pair

- AF & RI: as in balanced trees lecture; the keys form a BST

- Efficiency: traverse path from root to node or leaf
  - insert: O(log n)
  - find: O(log n)
  - remove: O(log n)

MAPS AS ARRAYS

Maps as arrays

Array index operation:
efficiently maps integers to values

Index	Value
…	…
459	Fan
460	Gries
461	Clarkson
462	Birrell
463	*does not exist*

Maps as arrays

- Aka *direct address table*
- Keys must be integers

- Representation type:

```
type 'v t = 'v option array
```

Maps as arrays

```
module type ArrayMapSig = sig
  type 'v t
  val create : int -> 'v t
  val insert : int -> 'v -> 'v t -> unit
  val find   : int -> 'v t -> 'v option
  val remove : int -> 'v t -> unit
end
```

Maps as arrays

- AF:
  - `[|Some v0; Some v1; ...|]` represents {0:v0, 1:v1, ...}
  - But if element i is `None`, then i is not bound in the map
- RI: none
- Efficiency: find, insert, remove all O(1)

Map implementations

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

- **Arrays:** fast, but keys must be integers
- **Lists and trees:** allow any keys, but slower

...we'd like the best of all worlds:
constant efficiency with arbitrary keys

HASH TABLES

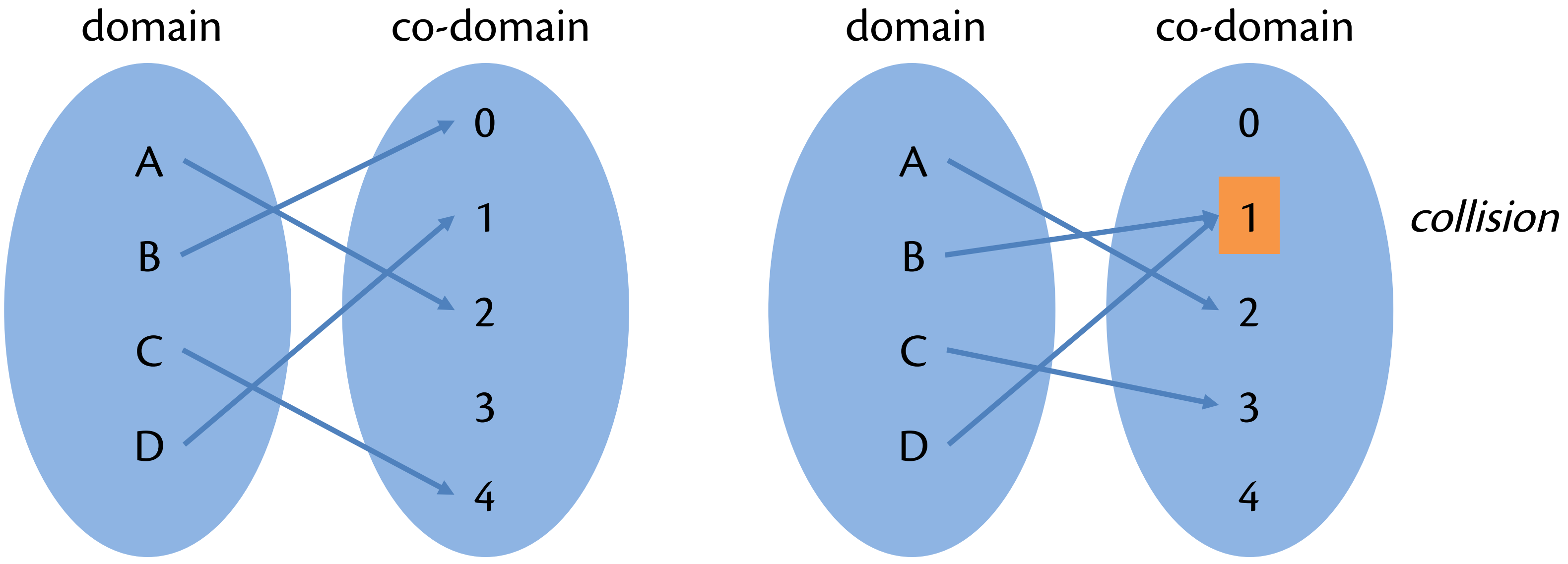
Mutable Maps

```
module type MutableMap = sig
  type ('k, 'v) t
  val insert:
    'k -> 'v -> ('k, 'v) t -> unit
  val find:
    'k -> ('k, 'v) t -> 'v option
  val remove:
    'k -> ('k, 'v) t -> unit
end
```

Key idea: convert keys to integers

- Assume we have a conversion function `hash : 'k -> int`
- Want to implement `insert` by
  - hashing key to **`int`** within array bounds
  - storing binding at that index
- Conversion should be fast: ideally, constant time
- Problem: what if conversion function is not *injective*?

Injective: one-to-one



injective

not injective

Hash tables

- If hash function not injective, multiple keys will collide at same array index
- We're okay with that
- Strategy:
  - Integer output of hash called a *bucket*
  - Store multiple key-value pairs in a list at a bucket
    - Called *open hashing, closed addressing, separate chaining*
    - OCaml's **`Hashtbl`** does this
  - Alternative: try to find an empty bucket somewhere else
    - Called *closed hashing, open addressing*

Maps as hash tables

- Representation type combines association list with array:

```
type ('k, 'v) t = ('k*'v) list array
```

- Abstraction function: An array

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

represents the map

```
   {k11:v11,   k12:v12, ...,
    k21:v21,   k22:v22, ...,   ...}
```

Maps as hash tables

Representation invariants:

- No key appears more than once in array (so, no duplicate keys in association lists)
- All keys are in the right buckets: **`k`** appears in array index **`b`** iff **`hash(k)=b`**

Maps as hash tables

- Efficiency:
  - have to search through association list to find key
  - so efficiency depends on how long the lists are
  - which in turn depend on hash function...
- Terrible hash function: **`hash(k) = 42`**
  - All keys collide; stored in single bucket
  - Degenerates to an association list (with no dups) in that bucket
  - insert, find, remove: O(n)

Efficiency of hash table

- New goal: constant-time efficiency on average
  - Desired property of hash function: distribute keys randomly among buckets
    - Keep average bucket length small
    - Hard; similar to designing good PRNG
  - If length is on average L, then insert, find, remove will have expected running time that is O(L)
- New problem: how to make L a **constant** that doesn't depend on number of bindings in table?

Independence from # bindings

Load factor:

= (# bindings in hash table) / (# 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

= average bucket length

Both OCaml **`Hashtbl`** *and* **`java.util.HashMap`** *provide functionality to find out current load factor*

Independence from # bindings

- Load factor = # bindings / # buckets
  - # bindings not under implementer's control
  - # buckets is
- *Resize* array to be bigger or smaller
  - control average bucket length L
  - if L is bounded by a constant, then insert, find, remove will on average be O(L): constant time

Resizing the array

Requires a new representation type:

```
type ('k, 'v) t =
  ('k*'v) list array ref
```

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

Rehashing

- If load factor ≥ (or perhaps just >) 2.0 then:
  - double array size
  - rehash elements into new buckets
  - thus bringing load factor back to around 1.0

- Both OCaml **`Hashtbl`** and **`java.util.HashMap`** do this

- Efficiency:
  - find, and remove: expected O(2), which is still constant time
  - But insert: O(n), because it can require rehashing all elements
  - So why is the common wisdom that hash tables offer constant-time performance?

Improved analysis of insert

- Key insight: rehashing occurs very rarely
  - more and more rarely as size of table doubles!
  - e.g., at 16th insert, at 32nd, at 64th, …

- Consider: what is average cost of each insert operation in a long sequence?

Improved analysis of insert

Example: 8 buckets with 8 bindings, do 8 more inserts:

- first 7 inserts are (on average) constant-time
- 8th insert is linear time: rehashing 15 bindings plus final insert causes 16 inserts
- total cost: 7+16 inserts; round up to 16+16 = 32 inserts
- So average cost per insert in sequence is equivalent to 4 non-rehashing inserts
- i.e., could just pretend every insert costs quadruple!

This "creative accounting" technique: simple example of **amortized analysis**

Amortized analysis of efficiency

- *Amortize:* put aside money at intervals for gradual payment of debt [Webster's 1964]
  - *L. "mort-" as in "death"*
- Pay extra money for some operations as a *credit*
- Use that credit to pay higher cost of some later operations
- Invented by Sleator and Tarjan (1985)

Robert Tarjan

b. 1948

Turing Award Winner (1986)
with Prof. John Hopcroft

For fundamental achievements in the design and analysis of algorithms and data structures.

Cornell CS faculty 1972-1973

Hash tables

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

Upcoming events

- [Last Night] Prelim grades out
- [Today 3-4pm] Foster Office Hours