Hash Tables

Nate Foster

Spring 2020

Today's music: Re-hash by Gorillaz

CLICKER QUESTION 1

Review

Previously in 3110:

- Efficiency: Big Oh
- Mutable data types

Today:

- Hash tables: an efficient map implementation

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, "Cornell" : 1865}

Demo

Map implementations

Up next: three 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?

IMPL 1: ASSOCIATION LISTS

Demo

Association lists as rep type

- 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)
  - bindings: nested list traversal: O(n^2)

Discussion: how would efficiency change given "RI: no duplicate keys"?

IMPL 2: ARRAYS

Array operations

- Indexing maps integers to values in O(1) time:
`e1.(e2)`

- Update destructively mutates array in O(1) time:
`e1.(e2) <- e3`

Arrays as rep type

- Aka *direct address table*
- Keys must be integers
- Representation type:

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

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

Interface changes

- Functional (aka persistent) data structures:
  - Take as input old rep
  - Return new rep
- Imperative data structures:
  - Take as input rep
  - Mutate rep, return unit

Demo

Arrays as rep type

- 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: O(1)
  - bindings: O(n)

Map implementations

	insert	find	remove
Arrays	O(1)	O(1)	O(1)
Association lists	O(1)	O(n)	O(n)

- **Arrays:** fast, but keys must be integers
- **Association lists:** allow any keys, but slower

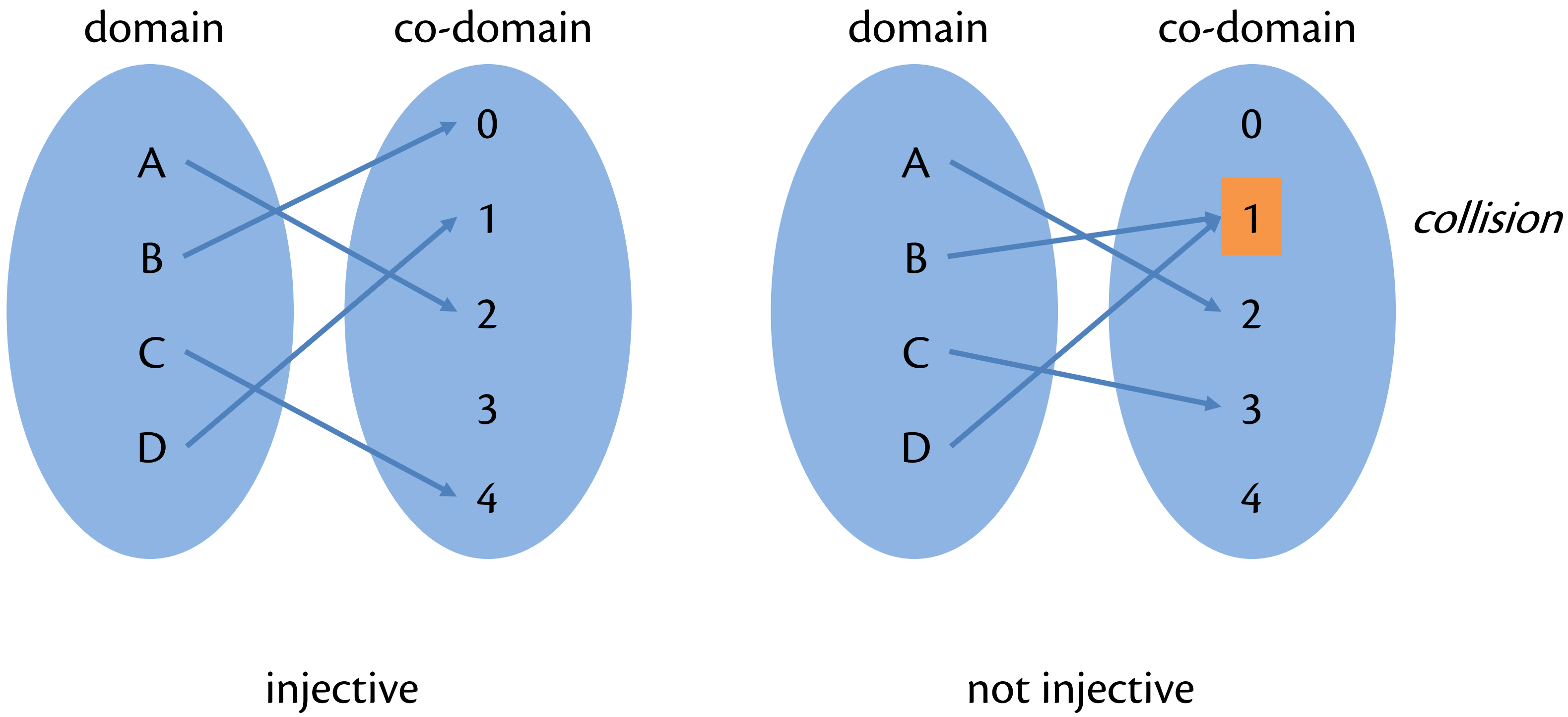
...we'd like the best of all worlds:
constant efficiency with arbitrary keys

HASH TABLES

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



Hash tables

- Integer output of hash called a *bucket*
  - If hash function not injective, multiple keys will collide at same bucket
  - We're okay with collisions
- Dealing with collisions:
  - Probing: find an empty bucket somewhere else
    - aka *closed hashing, open addressing*
  - Chaining: store multiple key-value pairs in a list at a bucket
    - aka *open hashing, closed addressing*
    - OCaml's **`Hashtbl`** does this
    - Let's use it ourselves…

Hash table rep type, v1

- 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, ...,   ...}
```

Hash table rep type, v1

Representation invariants:

- No key appears more than once in array
- All keys are in the right buckets:
  if **`k`** is in bucket **`b`** then **`hash(k)=b`**

Implementation of operations

- Insert (k, v):
  - Hash k to find bucket b
  - Search through b to delete any previous binding of k (to maintain RI)
  - Mutate bucket to add new binding of k
- Find k:
  - Hash k to find bucket b
  - Search through b to find binding of k
- Remove k:
  - Hash k to find bucket b
  - Search through b to delete any binding of k

…every operation requires search through bucket

…efficiency depends on bucket length

Bucket length

- Bucket length depends on hash function
- Terrible hash function: `hash(k) = 42`
  - All keys collide; stored in single bucket
  - Degenerates to an association list in that bucket
  - insert, find, remove: O(n)

Bucket length

- Assume new property of hash function: distribute keys randomly among buckets
- Random distribution implies all buckets have about the same length
- If expected bucket length is L, then insert, find, remove will have expected running time that is O(L)
- If L is bounded by a constant, then goal achieved: O(1) operations with arbitrary key types

Expected bucket length

Assuming hash function distributes uniformly...

Expected bucket length

= (# bindings in hash table) / (# buckets in array)

- *e.g.*, 10 bindings, 10 buckets, expected length = 1.0
- *e.g.*, 20 bindings, 10 buckets, expected length = 2.0
- *e.g.*, 5 bindings, 10 buckets, expected length = 0.5

Load factor

Regardless of hash function distribution…

Load factor =

(# bindings in hash table) / (# buckets in array)

Both OCaml `Hashtbl` *and* `java.util.HashMap` *provide functionality to query load factor*

Bounding the load factor

- # bindings not under implementer's control
- # buckets is
- When load factor gets above some constant, make array bigger
  - Which makes load factor smaller
  - Then redistribute keys across bigger array

Hash table rep type, v2

Resizing requires a new representation type:

```
type ('k, 'v) t = {
  mutable buckets
    : ('k * 'v) list array
}
```

- Mutate an array element to `insert` or `remove`
- Mutate buckets field to resize

Demo

Rehashing

- If load factor $\geq$ 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(1)
  - But insert: O(n), because it can require rehashing all elements
  - Next lecture: how to make insert O(1)

Rehashing

- If load factor < 0.5 then:
  - half array size
  - rehash elements into new buckets
  - thus bringing load factor back to around 1.0

- Neither OCaml **`Hashtbl`** nor **`java.util.HashMap`** do this

Upcoming events

- [last night] A3 due
- [tonight] Level Up!
- [Friday] deadline to schedule A3 demo
- [Saturday/Sunday] review sessions
- [Tuesday] prelim exam

This is #3110.

THIS IS 3110