CS 3110

Lecture 13: Hash tables

Prof. Clarkson

Fall 2014

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

Review

Recently:

- Imperative features
  - Refs, arrays, mutable fields
- Imperative data abstractions
  - Functional arrays implemented with refs

Today:

- Hash tables

Question #1

How excited are you about Prelim 1?

A. Excited
B. Super excited
C. Mega excited
D. Ultra excited
E. Super-mega-ultra excited

Prelim 1

- Thursday night
  - Your choice of 5:30-7:00 pm or 7:30-9:00 pm
  - Please arrive 15 minutes early to settle in
  - Three rooms, assigned by NetID (see Piazza)
- Closed book, with one page of notes
  - (8.5x11" two-sided)
- Covers Lecture 1 through Recitation 10, inclusive

Maps*

```
module type MAP = sig
  type ('key, 'value) map
  exception NotFound
  val insert:
    'key -> 'value -> ('key, 'value) map
      -> ('key, 'value) map
  val find: 'key -> ('key, 'value) map
      -> 'value option
  val remove: 'key -> ('key, 'value) map
      -> ('key, 'value) map
  ...
end
```

*aka associative array, dictionary, symbol table

Maps vs. Sets

- **Implement a set** with a map:
  - Abstraction function: a map m represents the set s whose elements are the keys bound by the map
  - e.g., `{k1=v1, k2=v2, ...}` represents the set `{k1,k2,...}`
  - values are just ignored
- **Implement a map** with a set (of pairs):
  - Abstraction function: a set s represents the map m that, for each element (k,v) of the set, contains the binding of key k to value v
  - Representation invariant: no key appears more than once in the set
  - e.g., `{(k1,v1),(k2,v2),...}` represents the map `{k1=v1, k2=v2, ...}`
- For our `MAP` interface, map and set implementations are interchangeable
  - maybe not quite as easy for richer interfaces, e.g., `MAP.all_values`

Map implementations

- Association lists
- Functions
- Balanced search trees
- Arrays
- Hash tables

Association lists

- Representation type:

```
type ('key, 'value) map =
  ('key*'value) 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)

Functions

- Representation type:

```
type ('key, 'value) map =
   'key -> 'value
```

- Abstraction function:
  - A function `fun k -> if k=k1 then v1 else (if k=k2 then v2 else ...)` represents the map `{k1=v1, k2=v2, ...}`
- Efficiency:
  - insert: O(1)
  - find: O(n)
  - remove: not supported.
    - Could introduce *negative entries* in function of the form **`if k=k' then raise NotFound`**
    - But then find is O(N) where N is the number of entries ever added to the map

Balanced search trees

Red-black trees:

- Representation type:

```
type ('key,'value) map = ('key,'value) rbtree
```

- Abstraction function: a node with label **`(k,v)`** and subtrees **`left`** and **`right`** represents the smallest map containing the binding **`{k=v}`** unioned with the bindings of **`left`** and **`right`**
- Representation invariant: *the red-black 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 tree*

Arrays

- Representation type:

```
type ('key, 'value) map = 'value option array
```

- Assume we can convert `'key` 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
  - Easiest realization: restrict keys to be integers!
- Abstraction function: An array `[|v1; v2; ...|]` represents the map `{inverse(1)=v1, inverse(2)=v2, ...}`.
- Aka *direct address table*
- Efficiency:
  - insert: O(1)
  - find: O(1)
  - remove: O(1)
  - wastes space, because some keys are unmapped

Question #2

If you wanted to map office numbers (e.g., 461) to occupant names (e.g., "Clarkson"), which implementation would be most time efficient?

A. Association lists
B. Functions
C. Balanced search trees
D. Arrays

Question #2

If you wanted to map office numbers (e.g., 461) to occupant names (e.g., "Clarkson"), which implementation would be most time efficient?

A. Association lists

B. Functions

C. Balanced search trees

D. Arrays

Map implementations

	insert	find	remove
Association lists	O(1)	O(n)	O(n)
Functions	O(1)	O(n)	N/A
Balanced search trees	O(lg n)	O(lg n)	O(lg n)
Arrays	O(1)	O(1)	O(1)

- Balanced search trees guarantee logarithmic efficiency
- Arrays guarantee constant efficiency, but require injective conversion of keys to integers

...we'd like constant efficiency with arbitrary keys

Hash tables

Main idea: give up on injectivity

- Allow conversion from `'key` to `int` to map multiple keys to the same integer
- Conversion function called a *hash function*
- Locations it maps to called *buckets*
- 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:

```
type ('key, 'value) map =
  ('key*'value) list array
```

- Abstraction function: An array

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

  represents the map `{k11=v11, k12=v12, ...}`.
- Representation invariants:
  - A key **`k`** appears in array index **`b`** iff **`hash(k)=b`**
  - No key appears more than once in its bucket
- Efficiency: ???
  - have to search through list to find key
  - no longer worst-case constant time

Efficiency of hash table

- Terrible hash function: `hash(k) = 42`
  - All keys collide; stored in single bucket
  - (Doesn't violate the RI for rep type on previous slide—it's not a duplication of keys in 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

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 bindings from being added or removed
  - 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, bringing load factor back to around 1.0
  - Rehash elements into new buckets
  - Efficiency:
    - insert: O(1)
    - find & remove: O(2), which is O(1)
    - rehashing: arguably still constant time; will return to this later in course
- If load factor gets too small (hence memory is being wasted), could shrink the array, thus increasing load factor
  - Neither OCaml nor Java do this

Question #3

How would you resize this representation type?

```
type ('key, 'value) map =
  ('key*'value) list array
```

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

Question #3

How would you resize this representation type?

```
type ('key, 'value) map =
  ('key*'value) 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 ('key, 'value) map =
    ('key*'value) list array ref
```

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

Good hash functions

Three steps to transform key to bucket index:

1. **Serialize** key into a stream of bytes
   - should be injective
2. **Diffuse** bytes into a single large integer
   - small change to key should cause large, unpredictable change in integer
   - might lose injectivity here, but good diffusion into an int64 is likely to still be injective
3. **Compress** the integer to be within range of bucket indices
   - dependence on number of buckets: need to map from key to [0..m-1]
   - definitely lose injectivity

Responsibility for each step is typically divided between client and implementer...

Responsibilities

OCaml **`Hashtbl`**:

- function **`Hashtbl.hash : 'a -> int`** does serialization and diffusion in native C code, based on MurmurHash
- function **`Hashtbl.key_index`** does compression
- so implementer is responsible for everything

Responsibilities

OCaml **`Hashtbl.Make`**:

- functor with input signature **`Hashtbl.HashedType`**, with functions
  - **`equal : t -> t -> bool`** and
  - **`hash : t -> int`**
- client provides **`equal`** and **`hash`** to do serialization and diffusion
  - must guarantee that if two keys are equal they have the same hash
- so implementer is responsible only for compression

Responsibilities

`java.util.HashMap:`

- method **`Object.hashCode()`** does serialization and diffusion
  - typical default implementation is to return address of object as an integer; not much diffusion there
  - client may override, must guarantee that if two keys are equal they have the same hash
- method **`HashMap.hash()`** does further diffusion
  - implementer doesn't trust client!
- method **`HashMap.indexFor()`** does compression
- so implementer splits responsibilities with client

Designing your own hash function

- Compression:
  - Both Java and OCaml make the number **`m`** of buckets a power of two, and compress by computing **`mod m`**
- Serialization:
  - Both Java and OCaml provide language support for serialization; in OCaml it's the **`Marshal`** module
- Diffusion:
  - Various techniques, including modular hashing, multiplicative hashing, universal hashing, cryptographic hashing...
  - If you don't achieve good diffusion, you lose constant-time performance!
  - If your hash function isn't constant time, you lose constant-time performance!
  - If you don't obey **`equals`** invariant, you lose correctness!
  - Designing a good hash function is hard

`Hashtbl` representation type

```
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.*

`Hashtbl` hash function

```
(* key_index : ('a, 'b) t -> 'c -> int *)
let key_index h key =
  ...
  (seeded_hash_param 10 100 h.seed key)
    land (Array.length h.data - 1)
  (* first line is serialization and diffusion,
   * second line is compression *)

external seeded_hash_param :
  int -> int -> int -> 'a -> int =
    "caml_hash" "noalloc"
(* caml_hash : 300 lines of C *)
(* hard to write good hash functions! *)
```

Hashtbl insert

```
(* add :  ('a, 'b) t -> 'a -> 'b -> unit *)
let add (h: ('a,'b) t) (key: 'a) info =
  let i = key_index h key in
  let bucket =
    Cons(key, info, h.data.(i)) in
  h.data.(i) <- bucket; (* mutation! *)
  h.size <- h.size + 1;
  if h.size >
    Array.length h.data lsl 1
    (* i.e. #buckets * 2 *)
  then resize key_index h
```

Hashtbl resize

```
let resize indexfun h =
  let odata = h.data in
  let osize = Array.length odata in
  let nsize = osize * 2 in (* double # buckets! *)
  if nsize < Sys.max_array_length then begin
    let ndata = Array.make nsize Empty in
    h.data <- ndata; (* mutation! *)
    let rec insert_bucket = function
        Empty -> ()
      | Cons(key, data, rest) ->
          insert_bucket rest;
          let nidx = indexfun h key in (* rehash! *)
          ndata.(nidx) <- Cons(key, data, ndata.(nidx)) in
    for i = 0 to osize - 1 do
      insert_bucket odata.(i)
    done
  end
```

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Upcoming events

- **PS4 released this week**
- **Prelim 1 on Thursday**

This is #3110.

THIS IS 3110