Caller-ID

Our goal today is to write a caller-id system that combines multiple strategies for efficient search in a phonebook. We will use association lists and hash tables.

Our "slow" strategy will rely on association lists.

We will then use a fast yet not comprehensive hash table that does not resolve collisions. Instead of storing association lists in the vector, we will only store one number. This strategy will be super-fast when it succeeds (truly $\Theta(1)$) and it uses a fixed storage, but it will sometimes be unable to find the answer. In this case, we will fall back to a more complete hash table, which might be stored remotely or on disk.

Our comprehensive strategy will use hash tables to accelerate the search, and will use association lists to resolve conflicts (when two keys map to the same index). This is the strategy used in lecture.

Association list (alist) :
List of 2 element lists (key value)

```
(define (find-assoc key alist)
   (cond
     ((null? alist) #f)
     ((equal? key (caar alist)) (cadar alist))
     (else (find-assoc key (cdr alist)))))

(define (add-assoc key val alist)
  (cons (list key val) alist))

(define (make-empty-asso-phone-book)
   YOUR JOB
(define (is-asso-phone-book? apb)
   YOUR JOB
(define (get-list-entry apb)   ;; returns a list of lists (1 name and 1 number each)
   YOUR JOB
(define (add-entry! name nr apb)
   YOUR JOB
(define (find-number name apb) ;; usual request
   YOUR JOB
(define (find-name nr apb)     ;;caller ID. Return 'unknown when you fail.
   YOUR JOB
```

Association list (alist) : List of 2 element lists (key value)

```
(define (find-assoc key alist)
   (cond
     ((null? alist) #f)
     ((equal? key (caar alist)) (cadar alist))
     (else (find-assoc key (cdr alist)))))

(define (add-assoc key val alist)
  (cons (list key val) alist))
```

```
(define (get-name asso) (car asso))
(define (get-nr asso) (cadr asso))

(define (make-empty-asso-phone-book)
   (cons 'asso-phone-book '()))
(define (is-asso-phone-book? apb)
   (and (pair? apb) (eq? (car apb) 'asso-phone-book)))
(define (get-list-entry apb)        ;; returns a list of lists (1 name and 1 number each)
   (cdr apb))
(define (add-entry! name nr apb)
   (set-cdr! apb (add-assoc name nr (get-list-entry apb))))
(define (find-number name apb) ;; usual request
   (find-assoc name (get-list-entry apb)))
(define (find-name nr apb)          ;;caller ID. Return 'unknown when you fail.
   (define (helper alist)
     (cond
       ((null? alist) 'unknown
       ((equal? nr (cadar alist)) (caar alist))
       (else (helper (cdr alist)))))
   (helper (get-list-entry apb)))
```

vectors: (a.k.a. arrays, "efficient lists")

- **vectors are *fixed* length (price to pay for efficiency)**
- **retrieval in Θ(1) (it's O(n) for lists)**

  `(make-vector k)`
  ; k is length of vector, initialize entries to #f

  `(vector-ref vector k)`
  ;returns the kth element of vector. Θ(1)

  `(vector-set! vector k obj)`
  ; sets the kth element of the vector to obj. Θ(1)
- **The first index is 0**
- **can you re-implement make-vector & vector-ref using cons?(ignoring efficiency)**
- **what about vector-set! ?**
- **What is the perfect caller-id system using a huge vector?**

vectors: (a.k.a. arrays, "efficient lists")

- **can you re-implement make-vector & vector-ref using cons?(ignoring efficiency)**

```
(define (make-vector N)
  (if (= N 0) '()
      (cons #f (make-vector (- N 1)))))
(define (vector-ref vector k)
  (if (= k 0) (car vector)
    (vector-ref (cdr vector) (- k 1))))
```

- **what about vector-set! ?**

```
(define (vector-set! vector k obj)
  (if (= k 0) (set-car! vector obj)
    (vector-set! (cdr vector) (- k 1)
                 obj))))
```

- **What is the perfect caller-id system using a huge vector?**
  - **Use a huge vector where the index is the phone number! Access is Θ(1).**
  - **Hash tables are a practical way to get almost this.**

Hash tables

table: list of bindings/pairings of a key and a value

hash tables: vector of bindings/pairings of a key and value. the key is mapped to an *index* by a *hash function*

hash functions: "uniformly" distributes the keys throughout the range of legal index values (0 -> k-1). "chooses a bucket".

collisions: multiple keys often map to same index. *One* solution is to store an association list at each index.

hash tables:

`(make-empty-table)`	: Θ(n) need to create & initialize the vector
`(put! key value)`	: Θ(1) if the hash function is easy to compute
`(get key)`	: ~Θ(1) for a "good" hash function, and big enough vector

Discussion

Who has implemented fast association search?

What data structure have you used?

Which ones have you heard about

When are hash tables used?

- **Internet caching**
- **Google**
- **Databases**
- **Dictionary**

fast & bounded storage yet not comprehensive hash table

Our first hash table only stores a single association of name-number per index.

```
(define (make-empty-hash1-phone-book size hash-func)
        ;;hash-func hashes based on number.
  YOUR JOB
(define (h1-get-size h1-table) ;; returns number of buckets
 YOUR JOB
(define (h1-get-func h1-table)         ;; returns hash function
 YOUR JOB
(define (h1-get-buckets h1-table) ;; returns the vector
 YOUR JOB
(define (h1-get-index nr h1-table)
 YOUR JOB
(define (h1-get-bucket nr h1-table) ;; returns an list with one single
   association
 YOUR JOB
(define (h-func nr size) ;;implement something simple
 YOUR JOB
(define (h1-add-entry name nr h1-table);; You should replace the existing
   entry if any.
 YOUR JOB
(define (asso-pb->h1-pb assopb size hash-func) ;;convert from previous
   representation.
 YOUR JOB
(define (h1-find-name nr h1-table)
        ;; return 'unknown when not present
 YOUR JOB
```

fast & bounded storage yet not comprehensive hash table

```
(define (make-empty-hash1-phone-book size hash-func);;hash-func hashes based on number.
  (let ((buckets (make-vector size)))
    (define (fill k)
      (if (< k size)
        (begin (vector-set! buckets k '())
               (fill (+ 1 k)))))
    (fill 0)
    (list 'hash1-phone-book buckets size hash-func))
(define (h1-get-size h1-table) ;; returns number of buckets
  (caddr h1-table))
(define (h1-get-func h1-table)          ;; returns hash function
  (cadddr h1-table))
(define (h1-get-buckets h1-table) ;; returns the vector
  (cadr h1-table))
(define (h1-get-index nr h1-table)
  ((h1-get-func h1-table) nr (h1-get-size h1-table)))
(define (h1-get-bucket nr h1-table) ;; returns an list with one single association
  (vector-ref (h1-get-buckets h1-table) (h1-get-index nr h1-table)))
(define (h-func nr size) ;;implement something simple
   (remainder nr size))
(define (h1-add-entry name nr h1-table);; You should replace the existing entry if any.
  (vector-set! (h1-get-buckets h1-table) (h1-get-index nr h1-table) (list name nr)))
(define (asso-pb->h1-pb assopb size hash-func) ;;convert from previous representation.
  (let ((h1-table (make-empty-hash1-phone-book size hash-func)))
       (map (lambda(asso)
              (h1-add-entry (get-name asso) (get-nr asso) h1-table))
            (get-list-entry assopb))
       h1-table))
(define (h1-find-name nr h1-table) ;; return 'unknown when not present
  (let ((L (h1-get-bucket nr h1-table)))
    (cond ((null? L) 'unknown)
          ((= (cadr L) nr) (car L)
          else 'unknown)))
```

fast & bounded storage yet not comprehensive hash table

```
(define (make-empty-hash1-phone-book size hash-func);;hash-
   func hashes based on number.
  (let ((buckets (make-vector size)))
    (define (fill k)
      (if (< k size)
        (begin (vector-set! buckets k '())
               (fill (+ 1 k)))))
    (fill 0)
    (list 'hash1-phone-book buckets size hash-func))
(define (h1-get-size h1-table) ;; returns number of buckets
  (caddr h1-table))
(define (h1-get-func h1-table)     ;; returns hash function
  (cadddr h1-table))
(define (h1-get-buckets h1-table) ;; returns the vector
  (cadr h1-table))
(define (h1-get-index nr h1-table)
  ((h1-get-func h1-table) nr (h1-get-size h1-table)))
(define (h1-get-bucket nr h1-table) ;; returns an list with one single
   association
  (vector-ref (h1-get-buckets h1-table) (h1-get-index nr
  h1-table)))
(define (h-func nr size) ;;implement something simple
   (remainder nr size))
```

fast & bounded storage yet not comprehensive hash table

```
(define (h1-add-entry name nr h1-table);; You should
  replace the existing entry if any.
  (vector-set! (h1-get-buckets h1-table)
    (h1-get-index nr h1-table) (list name nr)))
(define (asso-pb->h1-pb assopb size hash-func)
  ;;convert from previous representation.
  (let ((h1-table (make-empty-hash1-phone-book
                                size hash-func)))
        (map (lambda(asso)
               (h1-add-entry (get-name asso)
                             (get-nr asso) h1-table))
             (get-list-entry assopb))
        h1-table))
(define (h1-find-name nr h1-table) ;; return 'unknown when
  not present
  (let ((L (h1-get-bucket nr h1-table)))
    (cond ((null? L) 'unknown)
          ((= (cadr L) nr) (car L)
          else 'unknown)))
```

Discussion

What would be a not-so-good hash function?

If we use the first digits: many numbers might start with 617 which will result in many collisions

Probabilities to have collision?

The probability not to have a collision is quite low. This is related to the birthday paradox: for a group of 23 people, the probability to have twice the same birthday is more than 50%

Comprehensive hash table

Resolve conflicts using association lists

```
(define (make-empty-hash2-phone-book size hash-func) ;;hash-func hashes based on number .
   YOUR JOB
(define (h2-get-buckets h2-table) ;; returns buckets
 YOUR JOB
(define (h2-get-size h2-table) ;; returns number of buckets
 YOUR JOB
(define (h2-get-func h2-table) ;; returns hash function
 YOUR JOB
(define (h1-get-index nr h1-table)
 YOUR JOB
(define (h2-get-bucket nr h2-table) ;; returns an asso-phone-book
 YOUR JOB
(define (h2-add-entry! name nr h2-table)
    YOUR JOB
(define (asso-pb->h2-pb assopb size hash-func) ;;convert from association list representation.
   YOUR JOB
(define (h2-find-name nr h2-table)
   YOUR JOB
```

Comprehensive hash table

```
(define (make-empty-hash2-phone-book size hash-func) ;;hash-func hashes based on number .
  (let ((buckets (make-vector size)))
    (define (fill k)
      (if (< k size)
        (begin (vector-set! buckets k (make-empty-asso-phone-book))
               (fill (+ 1 k)))))
    (fill 0)
    (list 'hash1-phone-book buckets size hash-func))
(define (h2-get-buckets h2-table) ;; returns buckets
    (cadr h2-table))
(define (h2-get-size h2-table) ;; returns number of buckets
    (caddr h2-table))
(define (h2-get-func h2-table) ;; returns hash function
    (cadddr h2-table))
(define (h2-get-index nr h2-table)
  ((h2-get-func h2-table) nr (h2-get-size h2-table)))
(define (h2-get-bucket nr h2-table) ;; returns an asso-phone-book
    (vector-ref (h2-get-buckets h2-table) (h2-get-index nr h2-table)))
(define (h2-add-entry! name nr h2-table)
   (let ((index (h2-get-index nr h2-table)))
      (vector-set! (h2-get-buckets h2-table) index
        (add-entry name nr (h2-get-bucket nr h2-table)))
(define (asso-pb->h2-pb assopb size hash-func) ;;convert from association list representation.
  (let ((h2-table (make-empty-hash1-phone-book size hash-func)))
       (map (lambda(asso)
              (h2-add-entry! (get-name asso) (get-nr asso) h2-table))
            (get-list-entry assopb))
       h2-table))
(define (h2-find-name nr h2-table)
  (let ((bucket (h2-get-bucket nr h2-table)))
     (get-name nr bucket)))
```

Comprehensive hash table

```
(define (make-empty-hash2-phone-book size hash-func) ;;hash-
  func hashes based on number.
  (let ((buckets (make-vector size)))
    (define (fill k)
      (if (< k size)
        (begin (vector-set! buckets k (make-empty-asso-
  phone-book))
               (fill (+ 1 k)))))
    (fill 0)
    (list 'hash1-phone-book buckets size hash-func))
(define (h2-get-buckets h2-table) ;; returns buckets
  (cadr h2-table))
(define (h2-get-size h2-table) ;; returns number of buckets
  (caddr h2-table))
(define (h2-get-func h2-table) ;; returns hash function
  (cadddr h2-table))
(define (h2-get-index nr h2-table)
  ((h2-get-func h2-table) nr (h2-get-size h2-table)))
(define (h2-get-bucket nr h2-table) ;; returns an asso-phone-book
  (vector-ref (h2-get-buckets h2-table) (h2-get-index nr
  h2-table)))
```

Comprehensive hash table

```
(define (h2-add-entry! name nr h2-table)
   (let ((index (h2-get-index nr h2-table)))
    (vector-set! (h2-get-buckets h2-table) index
        (add-entry name nr
            (h2-get-bucket nr h2-table)))
(define (asso-pb->h2-pb assopb size hash-func)
  ;;convert from association list representation.
  (let ((h2-table (make-empty-hash1-phone-book
                                    size hash-func)))
        (map (lambda(asso)
               (h2-add-entry (get-name asso)
                     (get-nr asso) h2-table))
           (get-list-entry assopb))
        h2-table))
(define (h2-find-name nr h2-table)
  (let ((bucket (h2-get-bucket nr h2-table)))
    (get-name nr bucket)))
```

Putting it all together

Now write a data structure caller-id and the various functions to construct and query it. It will use both a small h1-table for fast queries, and will fall back to an h2-table when h1 fails. In a real system, the fast hash table might be local while the comprehensive one might be an internet query resolved on a different computer, on disk or via the internet.

```
(define (make-caller-id                ;;takes an asso-phone-book
```
and the parameters necessary to set up the two hash tables.

```
;;write selectors when necessary
(define (get-caller-name nr caller-id)
```
;;first try the fast hash table abd fall back to the comprehensive one when you fail

How do we need to change our overall strategy to be able to look for numbers given a name?

Putting it all together

```
(define (make-caller-id apb size1 size2)
  (let* ((f (lambda(x size)(remainder x size)))
         (h1 (asso-pb->h1-pb apb size1 f))
         (h2 (asso-pb->h2-pb apb size2 f))
    (list 'caller-id h1 h2))
;;write selectors when necessary
(define (get-h1 cid) (cadr cid))
(define (get-h2 cid) (caddr cid))
(define (get-caller-name nr caller-id)
  ;;first try the fast hash table abd fall back to the comprehensive one when you fail
  (let ((name1 (h1-get-name nr (get-h1 cid))))
    (if (not (eq? name1 'unknown)) name1
      (let ((name2 (h2-get-name nr (get-h2 cid))))
        (if (not (eq? name2 'unknown))
          (h1-add-entry name2 nr (get-h1 cid)))
                    ;;if found, update the fast hash table
        name2))))
```

How do we need to change our overall strategy to be able to look for numbers given a name?

We need additional hash tables indexed by the names.

Choices

Discuss the choice for the sizes

Can we replace the association list by something faster?

We can use binary search tree (which we'll study soon)

The cost is log instead of linear

Philosophy

- **When deciding for a data structure**
  - **Consider the frequency of various operations (e.g. read vs. write)**
    - **Example: Michael Stonebreaker (Adjunct Professor in CSAIL) criticize databases because their optimization for write makes them slow for reads**
  - **Consider the complexity of the code and the pain of debugging**
    - **For an infrequent operation, is it worth going through the pain of debugging mutation?**
    - **For example, many researchers prototype ideas using matlab although it is known to be slow**

For advanced students

- **2-left hashing: Use two hash tables and two hash functions and for each key, use the table with the fewest collisions.**
  - **Look up cost is the same, but collisions are far less common. The table is more balanced.**
- **Dynamic hashing: change the size of the table and the hashing function when the size of the data set grows.**
- **…**
- **See also**
- **http://www.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm**

- **Hashing is also a critical component of cryptography**
  - **weird mappings that cannot be discovered or inverted**