Administrative Notes

Extra credit homework
• Due Friday
• Counts as 10 point extra credit toward homework grade

Third exam
• Available Friday
• Due April 24 at 5 pm
• Closed notes, closed book
• Material since second exam

Rankings

International Tennis Federation (Tiddlywinks?)
• Rankings of top 100 players
• Stores player’s name, home country, & number matches won
• Frequent queries by rank
  → Who is the 2^{nd} best player? 15^{th} best player?
  → Need a program find-by-rank: ranking number \rightarrow player

Let’s develop a version
• Follow the structural recursion plan
Rankings

Need some data definitions

;;; a player is a structure
;;; (make-player name home wins)
;;; where name and home are symbols and wins is a number
(define-struct player (name home wins))

;;; a ranking is a list of player containing 100 elements
;;; with the players in ascending rank order
;;; We will use Scheme’s built-in list constructor

;;; find-by-rank: ranking number -> player
;;; Purpose: rakes a ranking and a number and returns the player in the
;;; ranking indicated by the number (player in position “number”) (define (find-by-rank a-ranking a-number) … )

We can use the classic list template ...

Rankings

Filling in the template

;;; find-by-rank: ranking number -> player
;;; Purpose: rakes a ranking and a number and returns the player in the
;;; ranking indicated by the number (player in position “number”) (define (find-by-rank a-ranking a-number)
(local [(define (helper alop at)
    (cond [(= at player-number) (first alop)]
          [else (helper (rest alop) (add1 at))])) ]
    (helper a-ranking 1)
)

This is fairly straightforward
Rankings

Could also have written

;; find-by-rank: ranking number -> player
;; Purpose: rakes a ranking and a number and returns the player in the
;; ranking indicated by the number (player in position “number”)
(define (find-by-rank a-ranking player-number)
  (cond [(= player-number 1) (first a-ranking)]
       [else (find-by-rank (rest a-ranking) (sub1 player-number))])
)

This one counts down to the desired position
• Relies implicitly on player-number being a natural number
• Somewhat simpler to read and write

Scheme provides this functionality

list-ref: list-of-alpha number -> alpha

We can write find-by-rank using list-ref

;; find-by-rank: ranking number -> player
;; Purpose: rakes a ranking and a number and returns the player in the
;; ranking indicated by the number (player in position “number”)
(define (find-by-rank a-ranking player-number)
  (list-ref a-ranking (sub1 player-number)))

This is much easier to write!
• Advantage of using pre-written code!
Rankings

What’s wrong with this code?

- Lately, we only put code up to criticize it!

```scheme
;; find-by-rank: ranking number -> player
;; Purpose: takes a ranking and a number and returns the player in the
;; ranking indicated by the number (player in position “number”) (define (find-by-rank a-ranking player-number) (list-ref a-ranking (sub1 player-number)))
```

How long does it take to return an answer?

- Number of recursive calls is proportional to rank
  → Uniform distribution of requests means average of 50
- We should be able to do better than that

   Hint: how many players are in the ranking?

Speeding up find-by-rank

The rankings have fixed length

- Lists work well for unbounded sets of items
- Structures work well for data-sets of known size

What about using a structure for the ranking?

   → Standard COMP 210 reasoning

```scheme
;; a ranking is a structure
;; (make-ranking p1 p2 p3 ... p100)
;; where all the p_i are players
(define-struct ranking (p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 ... p100))

Need to type them all out explicitly
```

Now, how do we write find-by-rank?
### Speeding up find-by-rank

#### Using the ranking structure

`;; find-by-rank: ranking number -> player
(define (find-by-rank a-ranking player-number)
  (cond [(= player-number 1) (ranking-p1 a-ranking)]
        [(= player-number 2) (ranking-p2 a-ranking)]
        [(= player-number 3) (ranking-p3 a-ranking)]
        ...
        [(= player-number 100) (ranking-p100 a-ranking)]
  ))

This has some of the right ideas

- It does not walk the list of rankings
- But, how many `cond` clauses does it evaluate?
  - On average, with normally distributed rankings, 50

### Speeding up find-by-rank

#### What’s the real problem here?

- We pushed the complexity into the data definition
- We pushed the cost into evaluating the `cond` clauses

#### The real issue

- We need a mechanism to compute the name of an element in the ranking
- List-ref simulates this, but we saw how it works
  - The simulation does the computation with structural recursion over the integers, which is expensive
- Need a faster way
Desiderata

Need a data structure with specific properties
- Quick, direct random access of a structure
- Computed names to give a list-ref like interface

Enter the vector
- Fixed number of elements
- Named by their ordinal position in the vector
- Accessed directly by that number
  → Computer scientists start numbers with zero, not one
- Fast, efficient access by element number

Vectors

Interface
- vector is analogous to list

(define KeithFavorites (vector 'COMP412 'CAAM460 'ENGL314))

- vector is supported by several functions
  → vector-length (vector-length KeithFavorites) ⇒ 3
  → vector-ref (vector-ref KeithFavorites 2) ⇒ 'ENGL314
  → vector-set! (vector-set! KeithFavorites 0 'COMP210)

- Initializer: build-vector: num (num->num) → vector
  (build-vector 5 (lambda(x)(* x x))) ⇒ (vector 0 1 4 9 16)
**Vectors**

Why use a vector?
- The cost for vector-ref is independent of position
- Number of components is fixed
- Since index is a number, can compute the index

Rewriting find-by-rank

`;; a ranking is a vector of 100 players
;;; find-by-rank: ranking number -> player
(define (find-by-rank a-ranking player-number)
  (vector-ref a-ranking (sub1 player-number)) )

Cost is constant

Again, index starts at zero

Now, how do we create a vector & modify rankings?

**Finishing up the rankings**

Create an empty ranking

`;; make-ranking: number -> vector
;;; Purpose: create a vector with all components set to false
(define (make-ranking size)
  (build-vector size (lambda(x) false)) )

Cost is proportional to size

Change a ranking

`;; rank-player!: ranking number player -> true
;;; Purpose: fill the rank specified by the number with the player
(define (rank-player! a-ranking a-number a-player)
  (begin
    (vector-set! a-ranking (sub1 a-number) a-player)
    true))

Cost is constant
Next Class

More fun with vectors

• Revisit Hoare’s quicksort algorithm
  → Think about the operation of picking a pivot

• Review for the exam