Algorithms & Data Structures

- Lists
  - Append vs. append!, reverse vs. reverse!, folding, …
  - List accessors: list-ref, list-tail, list-head, …
  - Sort & merge
- Trees
  - ADT for trees
  - Tree-fold, subst
- Compression via Huffman coding

Lists: Constructors, Selectors, Operations

- Basics of construction, selection
  - cons, list, list-ref, list-head, list-tail
- Operations
  - Combining: reverse, append
  - Process elements: map, filter, fold-right, fold-left, sort
- Abstraction: … just use Scheme's

Selectors: Beyond car, cdr

```
> (define ex '(a b c d e f))
> (list-ref ex 3)
d
> (list-tail ex 3)
(d e f)
> (list-tail ex 0)
(a b c d e f)
> (list-head ex 3)
(a b c)
```


Selectors: Beyond car, cdr

```
> (define ex '(a b c d e f))
> (list-ref ex 3)
d

(define (list-ref lst n)
  (cond ((zero? n) (car lst))
        ((null? (cdr lst))
         (error "Cannot take list-ref"
                (list n lst)))
        (else (list-ref (cdr lst) (- n 1)))))
```


Selectors: Beyond car, cdr

```
> (list-tail ex 3)
(d e f)
> (list-tail ex 0)
(a b c d e f)

(define (list-tail lst n)
  (cond ((zero? n) lst)
        ((null? lst)
         (error "Cannot take list-tail"
                (list n lst)))
        (else (list-tail (cdr lst) (- n 1)))))
```


Selectors: Beyond car, cdr

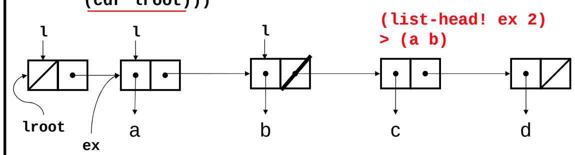
```
> (list-head ex 3)
(a b c)

(define (list-head lst n)
  (if (or (null? lst) (zero? n))
      '()
      (cons (car lst)
            (list-head
             (cdr lst) (- n 1)))))
```

List-head!

```
> (define ex '(a b c d e f))
> (list-head! ex 0)
()
> ex
(a b c d e f)
> (list-head! ex 2)
(a b)
> ex
(a b)
```


Destructive list-head!

```
(define (list-head! lst n)
  (let ((lroot (cons '() lst)))
    (define (iter l i)
      (if (zero? i)
          (set-cdr! l '())
          (iter (cdr l) (- i 1))))
    (iter lroot n)
    (cdr lroot)))
```



Append

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a)
            (append (cdr a) b))))
```

$T(n) = \Theta(n)$

$S(n) = \Theta(n)$

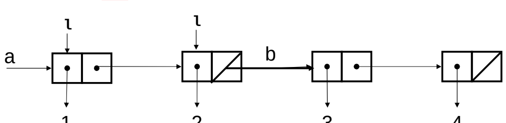
- Append copies first list
- Note on resources:
  - *S* measures space used by *deferred operations*, but not by list structure!

Append!

```
> (define a '(1 2))
> (define b '(3 4))
> (append! a b)
(1 2 3 4)
> a
(1 2 3 4)
> b
(3 4)
```


Append!

```
(define (append! a b)
  (define (iter l)
    (if (null? (cdr l))
        (set-cdr! l b)
        (iter (cdr l))))
  (cond ((null? a) b)
        (else (iter a)
         a)))
```



$T(n) = \Theta(n)$

$S(n) = \Theta(1)$

Reverse

```
(define (reverse0 lst)
  (if (null? lst)
      '()
      (append (reverse0 (cdr lst))
              (list (car lst)))))
```

$T(n) = \Theta(n^2)$

$S(n) = \Theta(n)$

Substitution model:

```
(reverse0 '(1 2 3))
(append (reverse0 '(2 3)) (list 1))
(append (append (reverse0 '(3)) (list 2)) (list 1))
(append (append (append (reverse0 '()) (list 3))
                (list 2))
        (list 1))
```

Reverse (better)

```
(define (reverse lst)
  (define (iter l ans)
    (if (null? l)
        ans
        (iter (cdr l) (cons (car l) ans))))
  (iter lst '()))

(define ex '(1 2 3))
(reverse ex)
(iter (1 2 3) '())
(iter (2 3) (1))
(iter (3) (2 1))
(iter () (3 2 1))
```

$T(n) = \Theta(n)$

$S(n) = \Theta(1)$

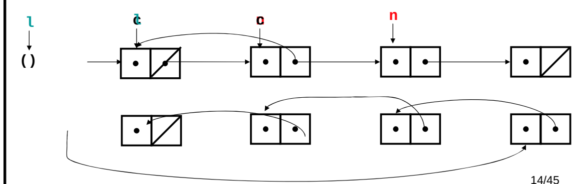
- Lists “come apart” from the front, but “build up” from the back: use this.

Reverse!

```
(define (reverse! lst)
  (define (iter last current)
    (if (null? current)
        last
        (let ((next (cdr current)))
          (set-cdr! current last)
          (iter current next))))
  (iter '() lst))
```

$T(n) = \Theta(n)$

$S(n) = \Theta(1)$



Two map’s & filter

```
(define (map0 f lst)
  (if (null? lst)
      '()
      (cons (f (car lst))
            (map0 f (cdr lst)))))
```

$T(n) = \Theta(n)$

$S(n) = \Theta(n)$

```
(define (map f lst)
  (define (iter l ans)
    (if (null? l)
        (reverse! ans)
        (iter (cdr l) (cons (f (car l)) ans))))
  (iter lst '()))
```

$T(n) = \Theta(n)$

$S(n) = \Theta(1)$

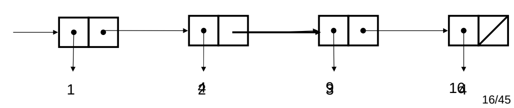
```
(define (filter f lst)
  (cond ((null? lst) '())
        ((f (car lst))
         (cons (car lst) (filter f (cdr lst))))
        (else (filter f (cdr lst)))))
```

$T(n) = \Theta(n)$

$S(n) = \Theta(n)$

map!

```
(define (map! f lst)
  (define (iter l)
    (cond ((null? l) lst)
          (else (set-car! l (f (car l)))
                (iter (cdr l)))))
  (iter lst))

  > (define ex '(1 2 3 4))
  > (map! square ex)
    (1 4 9 16)
  > ex
    (1 4 9 16)
```



Fold Operations

```
(define (fold-right0 fn init lst)
  (if (null? lst)
      init
      (fn (car lst)
          (fold-right0 fn init (cdr lst)))))
                                 ;; T(n)=O(n), S(n)=O(n)

(define (fold-left fn init lst)
  (define (iter l ans)
    (if (null? l)
        ans
        (iter (cdr l) (fn ans (car l)))))
  (iter lst init))
                                 ;; T(n)=O(n), S(n)=O(1)

(define (fold-right fn init lst)
  (fold-left (lambda (x y) (fn y x)) init (reverse lst)))
                                 ;; T(n)=O(n), S(n)=O(1)
```


Sorting a list

1. Split in half
2. Sort each half
3. Merge the halves
   - Merge two sorted lists into one
   - Take advantage of the fact they are sorted

```
(4 1 7 9  4 2 11 5)
(4 1 7 9)(4 2 11 5)
(1 4 7 9)(2 4 5 11)
(1 2 4 4 5 7 9 11)
```

Merge

```
(define (merge x y less?)
  (cond ((and (null? x) (null? y)) '())
        ((null? x) y)
        ((null? y) x)
        ((less? (car x) (car y))
         (cons (car x) (merge (cdr x) y less?)))
        (else (cons (car y) (merge x (cdr y) less?)))))
```

```
> (merge '(1 4 7 9) '(2 4 5 11) <)
(1 2 4 4 5 7 9 11)
> (merge '(4 1 7 9) '(5 2 11 4) <)
(4 1 5 2 7 9 11 4)
```

X GIGO

Sorting a list

1. Split in half
2. Sort each half
3. Merge the halves

```
(4 1 7 9 4 2 11 5)
(4 1 7 9)              (4 2 11 5)
(4 1)        (7 9)
(4) (1)
(1 4)        (7)(9)
(1 4 7 9)
(1 4 7 9)              (4 2)(11 5)
```


Sort

```
(define (sort lst less?)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (else (let ((halves (halve lst)))
                (merge (sort (car halves) less?)
                       (sort (cdr halves) less?)
                       less?)))))

(define (halve lst)
  (let* ((halflength (quotient (length lst) 2))
         (mid (list-tail lst halflength)))
    (cons (list-head lst halflength)
          mid)))

> (sort '(4 1 5 2 7 9 11 4) <)
(1 2 4 4 5 7 9 11)
```


Of course, there is merge!

```
(define (merge! x y less?)
  (let ((xroot (cons '() x))
        (yroot (cons '() y)))
    (define (iter ans)
      (cond ((and (null? (cdr xroot)) (null? (cdr yroot)))
             ans)
            ((null? (cdr xroot))
             (append! (reverse! (cdr yroot)) ans))
            ((null? (cdr yroot))
             (append! (reverse! (cdr xroot)) ans))
            ((less? (cadr xroot) (cadr yroot))
             (let ((current (cdr xroot)))
               (set-cdr! xroot (cdr current))
               (set-cdr! current ans)
               (iter current)))
            (else
             (let ((current (cdr yroot)))
               (set-cdr! yroot (cdr current))
               (set-cdr! current ans)
               (iter current)))))
    (cond ((null? x) y)
          ((null? y) x)
          (else (reverse! (iter '()))))))
```


… and sort! and halve!

```
(define (sort! lst less?)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (else (let ((halves (halve! lst)))
                (merge! (sort! (car halves) less?)
                        (sort! (cdr halves) less?)
                        less?)))))
(define (halve! lst)
  (cond ((null? lst) (error "Can't halve" lst))
        ((null? (cdr lst)) (cons lst '()))
        (else
         (let* ((mid (list-tail lst (- (quotient (length lst) 2)
                                       1)))
                (2nd-half (cdr mid)))
           (set-cdr! mid '())
           (cons lst 2nd-half)))))
```

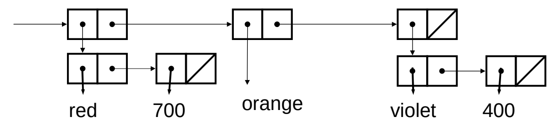

Sort of final word on sort

- Finding midpoint of list is expensive, and we keep having to do it
- Instead, nibble away from left
  - Pick off first two sublists of length 1 each
  - Merge them to get a sorted list of length 2
  - Pick off another sublist of length 2, sort it, then merge with previous ==> length 4
  - …
  - Pick off another sublist of length 2^n, sort, then merge with prev ==> length 2^{n+1}

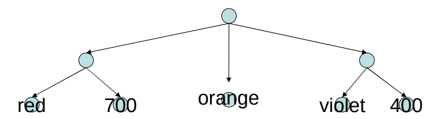
Trees

- Abstract Data Type for trees
  - Tree<C> = Leaf<C> | List<Tree<C>>
  - Leaf<C> = C
  - Note: C had best *not* be a list

```
(define (leaf? obj) (not (pair? obj))) ;; () can be a leaf

(define (leaf? obj) (not (list? obj))) ;; () is the empty tree
```

represents the tree



Counting leaves

```
(define (count-leaves tree)
  (cond ((leaf? tree) 1)
        (else (fold-left
               + 0
               (map count-leaves tree)))))

(define tr (list 4 (list 5 7) 2))
(define tr2 (list 4 (list '() 7) 2))
> (count-leaves tr)
4
> (count-leaves tr2)
4
```


General operations on trees

```
(define (tree-map f tree)
  (if (leaf? tree)
      (f tree)
      (map (lambda (e) (tree-map f e))
           tree)))

> tr
(4 (5 7) 2)
> (tree-map (lambda (x) (* x x)) tr)
(16 (25 49) 4)
```


Using tree-map and tree-fold

```
(define (tree-fold leaf-op combiner init tree)
  (if (leaf? tree)
      (leaf-op tree)
      (fold-right
       combiner
       init
       (map (lambda (e) (tree-fold leaf-op combiner init e))
            tree))))

> (tree-fold (lambda (x) 1) + 0 tr)
4
```


subst in terms of tree-fold

```
(define (subst replacement original tree)
  (tree-fold (lambda (x)
               (if (eqv? x original)
                   replacement
                   x))
             cons
             '()
             tree))

> (subst 3 'x '(+ (* x y) (- x x)))
(+ (* 3 y) (- 3 3))
```

Huffman Coding

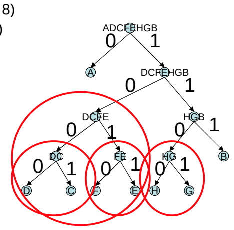
- If some symbols in an alphabet are more frequently used than others, we can compress messages
- ASCII uses 7 or 8 bits/char (128 or 256)
- In English, “e” is far more common than “z”, which in turn is far more common than Ctl-K (vertical tab(?))
- Huffman: use shorter bit-strings to encode most common characters
  - *Prefix codes:* no two codes share same prefix

Making a Huffman Code

- Start with a list of symbol/frequency nodes, sorted in order of increasing freq
- Merge the first two into a new node. It will represent the union of the symbols and sum of frequencies; sort it back into the list
- Repeat until there is only one node

Example of building a Huffman Tree

(H 1) (G 1) (F 1) (E 1) (D 1) (C 1) (B 3) (A 8)
(F 1) (E 1) (D 1) (C 1) ({H G} 2) (B 3) (A 8)
(D 1) (C 1) ({F E} 2) ({H G} 2) (B 3) (A 8)
({D C} 2) ({F E} 2) ({H G} 2) (B 3) (A 8)
({H G} 2) (B 3) ({D C F E} 4) (A 8)
({D C F E} 4) ({H G B} 5) (A 8)
(A 8) ({D C F E H G B} 9)
({A D C F E H G B} 17)



AHA ==> 0 1100 0

Leaf holds symbol & weight

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))

(define (leaf? obj)
  (and (pair? obj)
       (eq? (car obj) 'leaf)))

(define symbol-leaf cadr)
(define weight-leaf caddr)
```


Code tree

```
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))

(define left-branch car)
(define right-branch cadr)
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```


Building the Huffman Tree

```
(define (generate-huffman-tree pairs)
  (successive-merge (make-leaf-set pairs)))

(define (successive-merge leaf-set)
  (cond ((null? leaf-set) (error "bug in Huffman construction"))
        ((null? (cdr leaf-set)) (car leaf-set))
        (else
         (successive-merge
          (adjoin-set (make-code-tree
                        (car leaf-set)
                        (cadr leaf-set))
                       (cddr leaf-set))))))

(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))
```

Our training sample

(define text1 "The algorithm for generating a Huffman tree is very simple. The idea is to arrange the tree so that the symbols with the lowest frequency appear farthest away from the root. Begin with the set of leaf nodes, containing symbols and their frequencies, as determined by the initial data from which the code is to be constructed. Now find two leaves with the lowest weights and merge them to produce a node that has these two nodes as its left and right branches. The weight of the new node is the sum of the two weights. Remove the two leaves from the original set and replace them by this new node. Now continue this process. At each step, merge two nodes with the smallest weights, removing them from the set and replacing them with a node that has these two as its left and right branches. The process stops when there is only one node left, which is the root of the entire tree.")

Statistics

((leaf |H| 1) (leaf |B| 1) (leaf |R| 1) (leaf |A| 1) (leaf q 2)
(leaf |N| 2) (leaf |T| 4) (leaf v 5) (leaf |,| 5) (leaf u 7) (leaf b 7)
(leaf y 8) (leaf |.| 9) (leaf p 10) (leaf g 17) (leaf c 17) (leaf l 19)
(leaf f 19) (leaf m 20) (leaf d 22) (leaf w 25) (leaf r 37) (leaf n 41)
(leaf a 42) (leaf i 43) (leaf o 51) (leaf s 51) (leaf h 57)
(leaf t 84) (leaf e 109) (leaf | | 170))

The tree

```
(((leaf | | 170)
  ((((leaf m 20) (leaf d 22) (m d) 42) (leaf i 43) (m d i) 85)
   ((leaf o 51) (leaf s 51) (o s) 102)
   (m d i o s)
   187)
  (| | m d i o s)
  357)
 (((leaf e 109)
   (((leaf w 25)
     (((leaf |,| 5) (leaf u 7) (|,| u) 12)
      ((leaf b 7) (leaf y 8) (b y) 15)
      (|,| u b y)
      27)
     (w |,| u b y)
     52)
    (leaf h 57)
    (w |,| u b y h)
    109)
   (e w |,| u b y h)
   218) …
```

Space=>00
e=>100
t=>1111
h=>1011
s=>0111
o=>0110
…

How efficient?

- Our sample text has 887 characters, or 7096 bits in ASCII.
- Our generated Huffman code encodes it in 3648 bits, ≈51% (4.1 bits/char)
- Because code is built from this very text, it's as good as it gets!
- LZW (Lempel-Ziv-Welch) is most common, gets ≈50% on English.

Summary

- Lists: standard and mutating operators…
- Sort & merge
- Trees
- Compression via Huffman coding
- The organization of the code reflects the organization of the data it operates on.

Happy Spring Break!