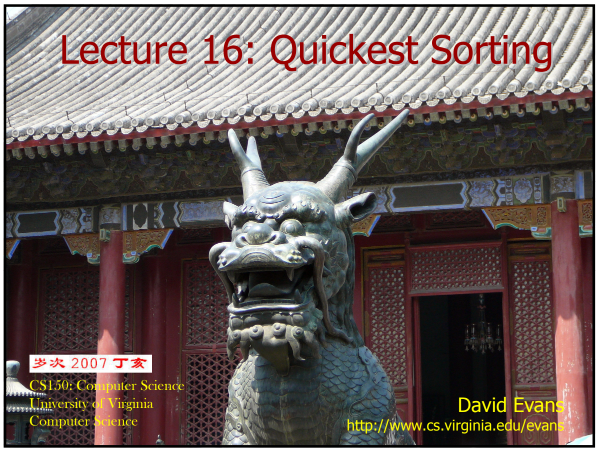


Exam 1 Reminders

- Review Session is tonight at 6:30 in Olsson 228E
- I have office hours after class today and Thursday at 3:30
- Kinga will be in Small Hall Friday morning, 10-11:30am
- If you have topics you want me to review in Friday's class, email me

Last class: Sorting Cost

```
(define (best-first-sort lst cf)
  (if (null? lst) lst
      (let ((best (find-best lst cf)))
        (cons best (best-first-sort (delete lst best) cf)))))
(define (find-best lst cf)
   (if (= 1 (length lst)) (car lst)
       (pick-better cf (car lst) (find-best (cdr lst) cf))))
```

The running time of best-first-sort is in $\boxed{\Theta(n^2)}$ where n is the number of elements in the input list.

This is wrong!

Length

```
(define (length lst)
   (if (null? lst) 0
       (+ 1 (length (cdr lst)))))
```

The running time of length is in $\Theta(n)$ where n is the number of elements in the input list.

find-best Cost

```
(define (find-best lst cf)
   (if (= 1 (length lst)) (car lst)
       (pick-better cf (car lst) (find-best (cdr lst) cf))))
```

assumption: cf is constant time procedure
n = number of elements in input list
there are n recursive applications of find-best
each one involves an application of (length lst) which is in $\Theta(n)$

The running time of find-best (using length) is in $\Theta(n^2)$ where n is the number of elements in the input list.

Sorting Cost

```
(define (best-first-sort lst cf)
  (if (null? lst) lst
      (let ((best (find-best lst cf)))
        (cons best (best-first-sort (delete lst best) cf)))))
(define (find-best lst cf)
   (if (= 1 (length lst)) (car lst)
       (pick-better cf (car lst) (find-best (cdr lst) cf))))
```

The running time of best-first-sort is in $\Theta(n^3)$ where n is the number of elements in the input list.

This is right (but very inefficient)!

best-first-sort

```
(define (best-first-sort lst cf)
  (if (null? lst) lst
      (let ((best (find-best lst cf)))
        (cons best (best-first-sort (delete lst best) cf)))))
(define (find-best lst cf)
   (if (null? (cdr lst)) (car lst)
       (pick-better cf (car lst) (find-best (cdr lst) cf))))
```

The running time of best-first-sort is in $\Theta(n^2)$ where n is the number of elements in the input list.

This is right!

Last class: insert-sort

```
(define (insert-sort lst cf)
  (if (null? lst) null
      (insert-one (car lst) (insert-sort (cdr lst) cf) cf)))

(define (insert-one el lst cf)
  (if (null? lst) (list el)
      (if (cf el (car lst)) (cons el lst)
          (cons (car lst) (insert-one el (cdr lst) cf)))))
```

Assuming cf is a constant time procedure, insert-sort has running time in $\Theta(n^2)$ where n is the number of elements in the input list

Can we do better?

```
(insert-one < 88
            (list 1 2 3 5 6 23 63 77 89 90))
```

Suppose we had procedures
(first-half lst)
(second-half lst)
that quickly divided the list in two halves?

insert-one using halves

```
(define (insert-one el lst cf)
  (if (null? lst) (list el)
      (if (null? (cdr lst))
          (if (cf el (car lst)) (cons el lst) (list (car lst) el))
          (let ((front (first-half lst))
                (back (second-half lst)))
            (if (cf el (car back))
                (append (insert-one el front cf) back)
                (append front
                        (insert-one el back cf)))))))
```


Evaluating insert-one

```
> (insert-one < 3 (list 1 2 4 5 7))
|(insert-one #<procedure:traced-<> 3 (1 2 4 5 7))
| (< 3 1)
| #f
| (< 3 5)
| #t
| (insert-one #<procedure:traced-<> 3 (1 2 4))
| |(< 3 1)
| |#f
| |(< 3 4)
| |#t
| |(insert-one #<procedure:traced-<> 3 (1 2))
| | (< 3 1)
| | #f
| | (< 3 2)
| | #f
| | (insert-one #<procedure:traced-<> 3 (2))
| | |(< 3 2)
| | |#f
| | (2 3)
| |(1 2 3)
| (1 2 3 4)
|(1 2 3 4 5 7)
(1 2 3 4 5 7)
```

```
(define (insert-one el lst cf)
  (if (null? lst) (list el)
      (if (null? (cdr lst))
          (if (cf el (car lst))
              (cons el lst)
              (list (car lst) el))
          (let ((front (first-half lst))
                (back (second-half lst)))
            (if (cf el (car back))
                (append (insert-one el front cf) back)
                (append front
                        (insert-one el back cf)))))))
```

Every time we call insert-one, the length of the list is approximately halved!

How much work is insert-one?

Each time we call insert-one, the size of lst halves. So, doubling the size of the list only increases the number of calls by 1.

```
(define (insert-one el lst cf)
  (if (null? lst) (list el)
      (if (null? (cdr lst))
          (if (cf el (car lst))
              (cons el lst)
              (list (car lst) el))
          (let ((front (first-half lst))
                (back (second-half lst)))
            (if (cf el (car back))
                (append (insert-one el front cf) back)
                (append front
                        (insert-one el back cf)))))))
```

List Size	Number of insert-one applications
1	1
2	2
4	3
8	4
16	5

Remembering Logarithms

$\log_b n = x$ means $b^x = n$

What is $\log_2 1024$?

What is $\log_{10} 1024$?

Is $\log_{10} n$ in $\Theta(\log_2 n)$?

Changing Bases

$$\log_b n = \underbrace{(1/\log_k b)}_{\text{If } k \text{ and } b \text{ are constants, this is constant}} \log_k n$$

$$\Theta(\log_2 n) \equiv \Theta(\log_{10} n) \equiv \Theta(\log n)$$

No need to include a constant base within asymptotic operators.

Number of Applications

Assuming the list is well-balanced, the number of applications of insert-one is in $\Theta(\log n)$ where n is the number of elements in the input list.

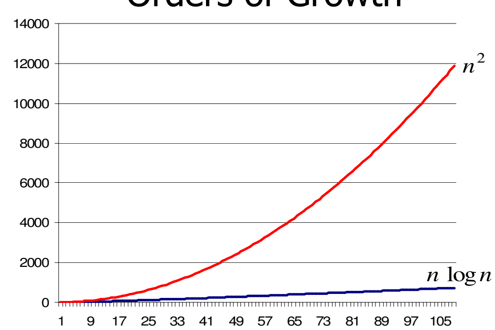
insert-sort

```
(define (insert-sort lst cf)
  (if (null? lst) null
      (insert-one
        (car lst)
        (insert-sort (cdr lst) cf)
       cf)))
```

```
(define (insert-one el lst cf)
  (if (null? lst) (list el)
      (if (null? (cdr lst))
          (if (cf el (car lst))
              (cons el lst)
              (list (car lst) el))
          (let ((front (first-half lst))
                (back (second-half lst)))
            (if (cf el (car back))
                (append (insert-one el front cf) back)
                (append front
                        (insert-one el back cf)))))))
```

insert-sort using halves would have running time in $\Theta(n \log n)$ **if** we had first-half, second-half, and append procedures that run in constant time

Orders of Growth



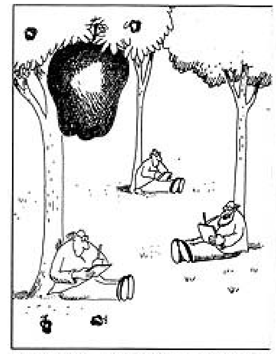
Is there a fast first-half procedure?

- No! (at least not on lists)
- To produce the first half of a list length n, we need to cdr down the first $n/2$ elements
- So, first-half has running time in $\Theta(n)$

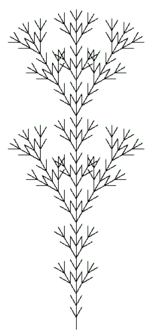
Making it faster

We need to either:

1. Reduce the number of applications of insert-one in insert-sort
   Impossible – need to consider each element
2. Reduce the number of applications of insert-one in insert-one
   Unlikely…each application already halves the list
3. Reduce the time for each application of insert-one
   Need to make first-half, second-half and append faster than $\Theta(n)$

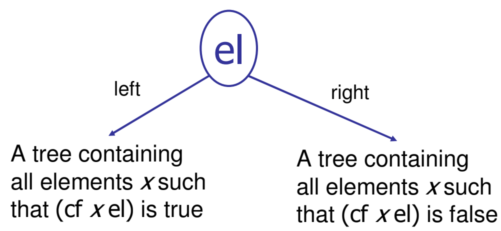


"Nothing yet. ...How about you, Newton?"

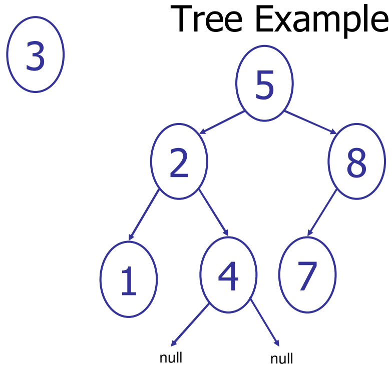


Happy Bird Tree
by Jon Morgan and Nadine Natour

Sorted Binary Trees



Tree Example



cf: <

Representing Trees

```
(define (make-tree left el right)
  (cons el (cons left right)))
```

left and right are trees
(**null** is a tree)

```
(define (tree-element tree)
  (car tree))
```

tree must be a non-null tree

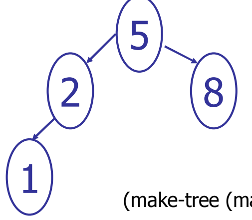
```
(define (tree-left tree)
  (car (cdr tree)))
```

tree must be a non-null tree

```
(define (tree-right tree)
  (cdr (cdr tree)))
```

tree must be a non-null tree

Representing Trees




```
(make-tree (make-tree (make-tree null 1 null)
                      2
                      null)
           5
           (make-tree null 8 null))
```


insert-one-tree

```
(define (insertel-tree cf el tree)
  (if (null? tree)
      (make-tree null el null)
      (if (cf el (get-element tree))
          (make-tree
           (insert-one-tree cf el (get-left tree))
           (get-element tree)
           (get-right tree))
          (make-tree
           (get-left tree)
           (get-element tree)
           (insert-one-tree cf el (get-right tree))))))
```

If the tree is null, make a new tree with el as its element and no left or right trees.

Otherwise, decide if el should be in the left or right subtree. insert it into that subtree, but leave the other subtree unchanged.

How much work is insert-one-tree?

```
(define (insert-one-tree cf el tree)
  (if (null? tree)
      (make-tree null el null)
      (if (cf el (get-element tree))
          (make-tree
           (insertel-tree cf el (get-left tree))
           (get-element tree) (get-right tree))
          (make-tree (get-left tree)
           (get-element tree)
           (insertel-tree cf el (get-right tree))))))
```

Each time we call insert-one-tree, the size of the tree approximately halves (if it is well balanced).

Each application is constant time.

The running time of insertel-tree is in $\Theta(\log n)$ where n is the number of elements in the input tree, which must be well-balanced.

insert-sort-helper

```
(define (insert-sort-helper cf lst)
  (if (null? lst) null
      (insert-one-tree
        cf (car lst)
        (insert-sort-helper cf (cdr lst)))))
```

No change (other than using insert-one-tree)...but evaluates to a tree not a list!

((((() 1 ()) 2 ()) 5 (() 8 ()))

Charge

- Exam 1 is out Friday, due Monday
- Exam Review, Wednesday 6:30 in Olsson 228E