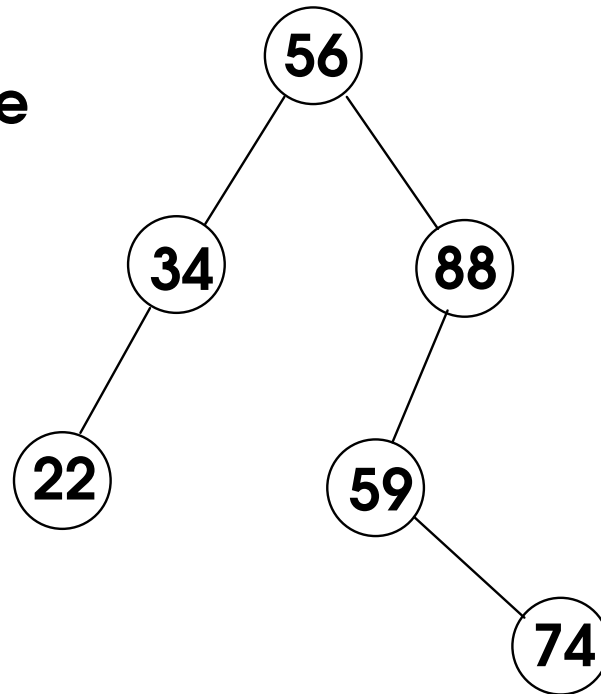


Binary trees in SCHEME (with internal nodes labeled)

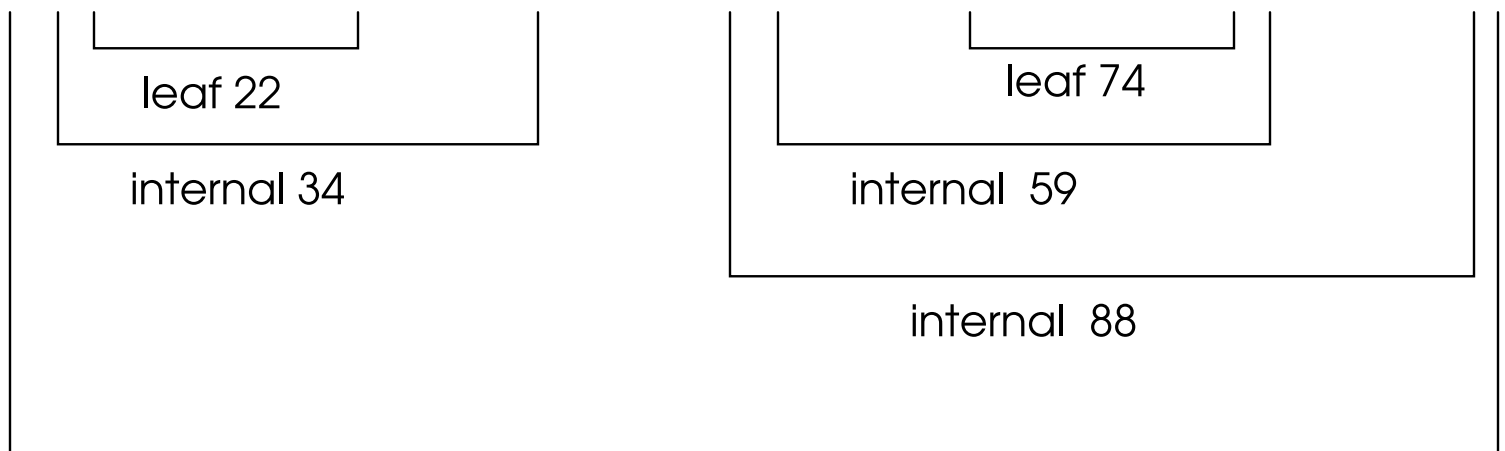
We can build our own type of binary tree using the default data structures (pairs, atoms) of SCHEME.

We represent each node as a 3-element list.

So the tree shown on the right is represented by the list below.



`(((() 22 ()) 34 ()) 56 ((() 59 (() 74 ())) 88 ()))`



root 56

Binary Trees in SCHEME (with internal nodes labeled)

Each node is a 3-element list:

(<left subtree> root_contents <right subtree>)

The empty tree is simply the empty list ().

We may build such trees in **inorder**.

For any subtree:

All nodes in the left subtree \leq value at root

All nodes in the right subtree $>$ value at root

If we perform an *inorder traversal* of the tree, the nodes are visited in increasing order.

An inorder traversal is defined recursively.

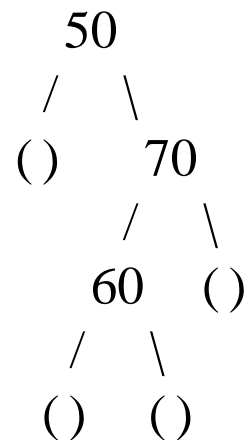
Base case: Tree is empty -- do nothing & return

otherwise: inorder traverse the left subtree,
visit (e.g., print) the root,
inorder traverse the right subtree.

So an inorder traversal of

(() 50 ((() 60 ()) 70 ()))

visits 50, 60, 70 in that order.



```

; this function uses the random number generator to create
; a list of integers of length in the range 1 - 100000.
;
(define makelist (lambda (n)
  (set! *random-state* (make-random-state #t))
  (mkl n) ))

(define mkl (lambda (n)
  (cond ((< n 2) (list (random 100000)))
        (#t (cons (random 100000)
                   (mkl (- n 1)) )) ) ))

(define compares 0) ; global counter

; invokes the binary tree sort function and prints
; #compares. then resets the global counter to zero.
; returns the ordered list whose elements are from L.
;
(define btsort (lambda (L)
  (let* ((iobt (btsrt L))
         (newL (ttl iobt)))
    (write compares) (write-string " comparisons used")
    (set! compares 0) (newline)
; could produce both the tree and list (write newL) iobt )
    newL ) ))

```

; this is an ordered binary tree-based sort.
; takes a list and returns an inorder binary tree.

```
(define btsrt (lambda (L)
  (cond ((null? L) L)
        ((null? (cdr L)) (list () (car L) ()))
        (#t (ins (car L)
                  (btsrt (cdr L)) ) ) ) )
```

; produces the tree resulting from adding an element
; to an inorder binary tree, preserving inorder.

```
(define ins (lambda (e L)
  (cond ((null? L) (list () e ()))
        (#t (set! compares (+ 1 compares))
              (cond ((< e (cadr L))
                     (cons (ins e (car L))
                             (cdr L) ) )
                    (#t (list (car L)
                               (cadr L)
                               (ins e (caddr L))
                               )
                          )
                  )
        )
  )) ) ) )
```

; take an inorder binary tree and produces
; the corresponding ordered list.

```
(define ttl (lambda (bt)
  (cond
    ((null? bt) ())
    (#t (append
          (ttl (car bt))
          (cons (cadr bt) (ttl (caddr bt)))) ) ) ) )
```