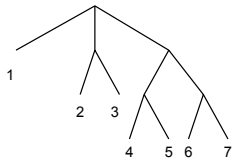
Trees

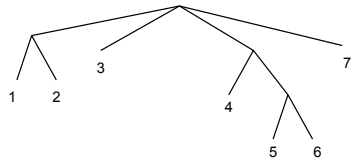
One standard way to represent tree structures: lists of lists

```
(1 (2 3) ((4 5) (6 7)))
```



Write a scheme expression to create the following tree

```
(list (list 1 2) 3 (list 4
 (list 5 6)) 7)
```



Tree structures in the world

Championship
Genealogy
Binary subdivision
Zoology
Any classification
Hierarchical data
  PowerPoint sub- bullets

Trees

Again, the list of list structure is only one of the possibilities to represent trees.

The leaves are the tree nodes that are not a pair

This means e.g. that we can't store pairs in such trees.

Write a function depth that takes a tree and returns the maximum depth of the tree.

```
(depth (list 1 (list (list 2) 3)
 (list 4))) ==> 3
(define (depth tree)
 (cond ((pair? tree)
   (max (+ 1 (depth (car tree)))
        (depth (cdr tree))))
 (else 0)) )
```

Now write depth using map and foldr...

```
(define (depth tree)
 (if (not (pair? tree)) 0
   (+ 1
     (foldr max 0
       (map depth tree)))) )
```

So far, we've been working on lists, while we've ignored the elements of the list. What does the following return?

```
(reverse (list 1 (list 2 3) (list 4 5 6)))
```

```
((4 5 6) (2 3) 1)
```

Write a function deep-reverse that when called on the above tree will reverse all the elements.

```
(deep-reverse (list 1 (list 2 3) (list 4 5
  6))) ==> ((6 5 4) (3 2) 1)
```

```
(define (deep-reverse x)
  (define (aux x ans)
     (cond ((null? x) ans)
           ((not (pair? x)) x)
           (else (aux (cdr x)
                 (cons (deep-reverse (car x))
                       ans)))))
  (aux x nil) )
```

Now write deep-reverse using map...

```
(define (deep-reverse x)
  (if (not (pair? x)) x
    (map deep-reverse
         (reverse x))) )
```

Write the function flatten that takes a tree structure and returns a flat list of the leaves of the tree.

For example

```
(flatten (list 1 (list 2) 3))
  ==> (1 2 3)
```

```
(define (flatten x)
  (cond ((null? x) nil)
    ((not (pair? x)) (list x))
    (else
      (append (flatten (car x))
              (flatten (cdr x))))) )
```

Now try writing flatten using map and foldr.

```
(define (flatten x)
  (if (not (pair? x))
      (list x)
  (foldr append '()
      (map flatten x))) )
```

Write the function all-pairs that takes a list and returns a list of all (unordered) pairs of the elements in the list. For example,

```
(all-pairs (list 1 2 3 4)) ==> ((1
  2) (1 3) (1 4) (2 3) (2 4) (3 4))
```

Again, first think about how you'd do this, and then translate into Scheme. Maybe start with trying to just get the pairs ((1 2) (1 3) (1 4)), and then build up from there.

```
(define (all-pairs s)
  (if (null? s) ()
    (append
      (map (lambda(x)(list (car s) x))
        (cdr s))
      (all-pairs (cdr s)))) )
```

Quiz

```
Mean 86
Median 90
Deviation 13
A >=90  (59)
B >=80  (21)
C>=65   (17)
D>=55   (4)
F       (3)
```

Quiz difficult points

Part 3

beware of abstraction violations

terms and transcripts are abstract data types, not lists. You MUST convert them using convert-to-list

Part 4

find-best is linear in time and constant in space (iterative process)

remove is linear in time and in space (pending cons)

sort is quadratic in time (both find-best and remove are called a linear number of times)

sort is linear in space (the linear pending expression of remove is reduced a the end of each call. Therefore it does not accumulate)