Write a recursive function that searches for an element in a list

(define (recursive-find value L)
  ...
)

• You can assume L is a list of integers
• Returns true or false
• **Bonus:** write a non-recursive version that uses higher-order functions
(define (recursive-find value L)
  (cond [(empty? L) false]
        [(equal? value (first L)) true]
        [else (recursive-find value (rest L))])))

(define (higher-find value L)
  (let* ([matches? (lambda (v) (equal? v value))]
          [matches (filter matches? L)]
          (not (empty? matches)))))
Given a problem:
   Is there a solution?  
   What is it?  
   How good is it?

Is this element among these values?

Yep!

Racket lists
Racket lists

inductive data structure, manipulated via constructors, accessors, and operations

<table>
<thead>
<tr>
<th>constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>'()</td>
</tr>
<tr>
<td>(cons &lt;value&gt; &lt;list&gt;)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>accessors</th>
</tr>
</thead>
<tbody>
<tr>
<td>(first &lt;list&gt;)</td>
</tr>
<tr>
<td>(rest &lt;list&gt;)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(length &lt;list&gt;)</td>
</tr>
<tr>
<td>(empty? &lt;list&gt;)</td>
</tr>
<tr>
<td>(list? &lt;value&gt;)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Given a problem:
Is there a solution?  
What is it?  
How good is it?

Is this element among these values?

Yep!

Racket lists

...
Trees

- A unique path from root to every element

- **Node**
- **Edge**
- **Root** (no parent)
- **Leaves** (no children)
- **Height** (length of longest path from root to leaf)
Binary trees

structure constraint: every node has at most two children
Binary search trees (BSTs)

order constraint: every parent is greater than all the nodes in its left subtree and less than all the nodes in the right subtree.
Balanced binary search trees

structure constraint: every subtree is about the same size as its sibling
Designing and implementing a new data structure

Interface and implementation

- **Interface**
  
  *Answers: what can this data structure do*

- **Implementation: encoding**
  
  *Answers: how the structure is stored, using existing data structures*

- **Implementation: operations**
  
  *Answers: how the structure provides its interface via algorithms over the encoding*

It should be possible to replace the implementation without modifying the interface.
**Our Racket trees: Interface**

Inductive data structure, manipulated via constructors, accessors, and operations

<table>
<thead>
<tr>
<th>Constructors</th>
<th>Accessors</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>put together</td>
<td>take apart</td>
<td>often recursive</td>
</tr>
<tr>
<td>(make-empty-BST)</td>
<td>(emptyTree? &lt;tree&gt;)</td>
<td>(size &lt;tree&gt;)</td>
</tr>
<tr>
<td>(make-BST-leaf &lt;key&gt;)</td>
<td>(key &lt;tree&gt;)</td>
<td></td>
</tr>
<tr>
<td>(make-BST &lt;key&gt; &lt;left&gt; &lt;right&gt;)</td>
<td>(leftTree &lt;tree&gt;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(rightTree &lt;tree&gt;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our BSTs won't be balanced
Our Racket trees: Implementation

How would you encode a BST as a list?

<table>
<thead>
<tr>
<th>Constructors put together</th>
</tr>
</thead>
<tbody>
<tr>
<td>(make-empty-BST)</td>
</tr>
<tr>
<td>(make-BST-leaf &lt;key&gt;)</td>
</tr>
<tr>
<td>(make-BST &lt;key&gt; &lt;left&gt; &lt;right&gt;)</td>
</tr>
</tbody>
</table>

(define (make-empty-BST) '())

(define (make-BST key left right)
  (list key left right))
Interface vs implementation

(define (size tree)
  (if (emptyTree? tree)
    0
    (+ 1
      (size (leftTree tree))
      (size (rightTree tree))))

(define (size tree)
  (if (null? tree)
    0
    (+ 1
      (size (second tree))
      (size (third tree))))

What are some good test cases for trees?