Find the path to a key

Write a function search-bt-path that searches for an item in the tree. As before, it will return false if the item is not found. However, if it is found search-bt-path will return a list of the symbols 'left and 'right indicating the path from the root to the item.
(check-expect (search-bt-path 0 empty) false)
(check-expect (search-bt-path 0 test-tree) false)
(check-expect (search-bt-path 6 test-tree) empty)
(check-expect (search-bt-path 9 test-tree) '(right right left))
(check-expect (search-bt-path 3 test-tree) '(left))
;; search-bt-path: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path k tree)
  (cond
   ; an empty tree cannot contain a label k.
   [(empty? tree) false]...)
(define (search-bt-path k tree)
  (cond
    [(empty? tree) false]
    ; The current node contains the label k I am looking for. Therefore, this is where I start with my path.
    [(= (node-key tree) k) empty])

(define (search-bt-path k tree)
  (cond
    [(empty? tree) false]
    [(= (node-key tree) k) empty]
    ;; If I have received a list from my left subtree
    ;; (instead of empty), k must be on this subtree,
    ;; and I must be part of the path to k. Therefore,
    ;; I add the direction for me (left) to whatever
    ;; direction I received from my left subtree.
    [(list? (search-bt-path k (node-left tree)))
      (cons 'left (search-bt-path k (node-left tree)))]
   ))
;; search-bt-path: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path k tree)
  (cond
   [(empty? tree) false]
   [(= (node-key tree) k) empty]
   [(list? (search-bt-path k (node-left tree)))
    (cons 'left (search-bt-path k (node-left tree)))]
   ; The same logic applies to my right subtree.
   [(list? (search-bt-path k (node-right tree)))
    (cons 'right (search-bt-path k (node-right tree)))]
   )
(define (search-bt-path k tree)
  (cond
   [(empty? tree) false]
   [(= (node-key tree) k) empty]
   [(list? (search-bt-path k (node-left tree)))]
     (cons 'left (search-bt-path k (node-left tree)))]
   [(list? (search-bt-path k (node-right tree)))]
     (cons 'right (search-bt-path k (node-right tree)))]
   [else false]))
define (search-bt-path k tree)
  (cond
    [(empty? tree) false]
    [(= (node-key tree) k) empty]
    [(list? (search-bt-path k (node-left tree)))]
      (cons 'left (search-bt-path k (node-left tree)))
    [(list? (search-bt-path k (node-right tree)))]
      (cons 'right (search-bt-path k (node-right tree)))
    [else false])}

Double calls to search-bt-path. Uggh!
;; search-bt-path: Nat BT -> (anyof false (listof Sym))
(define (search-bt-path k tree)
  (cond
   [(empty? tree) false]
   [ (= (node-key tree) k) empty]
   [else (choose-path
           (search-bt-path k (node-left tree))
           (search-bt-path k (node-right tree)))]))

;; choose-path: (anyof false (listof Sym))
;;              (anyof false (listof Sym)) ->
;;                             (anyof false (listof Sym))
(define (choose-path path1 path2)
  (cond
   [(list? path1) (cons 'left path1)]
   [(list? path2) (cons 'right path2)]
   [else false]))
Binary search trees
Binary search trees

We will now make one change that can make searching much more efficient. This change will create a tree structure known as a binary search tree (BST).

For any given collection of keys, there is more than one possible tree.

How the keys are placed in a tree can improve the running time of searching the tree when compared to searching the same items in a list.
A Binary Search Tree (BST) is one of:

* empty
* a Node

```
(define-struct node (key left right))
```

A Node is a (make-node Nat BST BST)

- requires: key > every key in left BST
- key < every key in right BST

The BST ordering property:
- key is greater than every key in left.
- key is less than every key in right.
A BST example

```
(make-node 6
  (make-node 3
    (make-node 2 empty empty)
    (make-node 4 empty empty))
  (make-node 7 empty
    (make-node 10
      (make-node 8 empty empty)
      (make-node 14 empty empty))))
```

; root
; root, left subtree
; root, right subtree
A BST example

There can be several BSTs holding a particular set of keys.
A BST example

There can be several BSTs holding a particular set of keys.
A BST example

There can be several BSTs holding a particular set of keys.
Making use of the ordering property

Main advantage: for certain computations, one of the recursive function applications in the template can always be avoided.

This is more efficient (sometimes considerably so).

In the following slides, we will demonstrate this advantage for searching and adding.

We will write the code for searching, and briefly sketch adding, leaving you to write the Racket code.
Searching in a BST

How do we search for a key n in a BST?

We reason using the data definition of BST.

• If the BST is empty, then n is not in the BST.
• If the BST is of the form \texttt{(make-node key left right)}, and key equals n, than we have found it.
• Otherwise it might be in either of the tree’s left or right.
If $n < key$, then $n$ must be in left if it is present at all, and we only need to recursively search in left.

If $n > key$, then $n$ must be in right if it is present at all, and we only need to recursively search in left.

Either way, we save one recursive function application.
;; (search-bst key tree) produces true if key is in
;;   tree, and false otherwise.

;; search-bst: Nat BST -> Bool
(define (search-bst key tree)
  (cond
[(empty? tree) false]
[(= (node-key tree) key) true]
[(> (node-key tree) key)
  (search-bst key (node-left tree))]
[(< (node-key tree) key)
  (search-bst key (node-right tree))])))
Adding to a BST

How do we add a new key key to a BST tree?

If tree is empty, then the result is a BST with only one node. Otherwise tree is of the form (make-node node left right).

If $k = n$, the key is already in the tree and we can simply return tree.

If $k < n$, then the new key must be added to left, and if $k > n$, then the key must be added to right.

Again, we need only make one recursive function application.
Creating a BST from a list

How do we create a BST from a list of keys?

We reason using the data definition of a list. If the list is empty, the is empty.

If the list is of the form (cons key lst), we add the key key to the BST created from the list lst. The first key is inserted last.

It is also possible to write a function that inserts keys in the opposite order.
Binary search trees in practice

If the BST has all left subtrees empty, it looks and behaves like a sorted list, and the advantage is lost.

In later courses, you will see ways to keep a BST “balanced” so that “most” nodes have nonempty left and right children. We will also cover better ways to analyze the efficiency of algorithms and operations on data structures.
Augmenting trees
Augmenting trees

So far nodes have been
(\texttt{\textbf{define-struct} node (key left right))}.

We can augment the node with additional data:
(\texttt{\textbf{define-struct} node (key value left right))}.

The name \texttt{value} is arbitrary, choose any name you like.
The type of \texttt{value} is also arbitrary: could be a number, string, structure, etc.

You could augment with multiple values.

The set of keys remains unique; could have duplicate values.
**BST dictionaries**

An augmented BST can serve as a dictionary that can perform significantly better than an association list.

Recall from Module 08 that a dictionary stores a set of (key, value)-pairs, with at most one occurrence of any key. A dictionary supports lookup, add, and remove operations.

We implemented dictionaries using an association list, a list of two-element lists. Search could be inefficient for large lists.

We need to modify node to include the value associated with the key. Search needs to produce the associated value, if found.
(define-struct node (key val left right))

;; A binary search tree dictionary (BSTD) is either
;; * empty
;; * (make-node Nat Str BSTD BSTD)

;; (search-bst-dict key tree) produces the val associated
;; with key if key is in tree, and false otherwise.
;; search-bst: Nat BSTD -> (anyof Str false)

(define (search-bst-dict key tree)
  (cond
    [(empty? tree) false]
    [(= (node-key tree) key) (node-val tree)]
    [ (> (node-key tree) key)
      (search-bst-dict key (node-left tree))]
    [ (< (node-key tree) key)
      (search-bst-dict key (node-right tree))])))
(define test-tree
  (make-node 5 "Susan"
    (make-node 1 "Juan"
      empty
      empty)
  (make-node 14 "David"
    (make-node 6 "Lucy"
      empty
      empty)))

(check-expect (search-bst-dict 5 empty) false)
(check-expect (search-bst-dict 5 test-tree) "Susan")
(check-expect (search-bst-dict 6 test-tree) "Lucy")
(check-expect (search-bst-dict 2 test-tree) false)
Example: Evolutionary trees

Evolutionary trees are another kind of augmented tree.

![Evolutionary Trees Diagram]
Evolutionary trees are binary trees that show the evolutionary relationships between species. Biologists believe that all life on Earth is part of a single evolutionary tree, indicating common ancestry.

Internal nodes represent an evolutionary event when a common ancestor species split into two new species. Internal nodes are augmented with the common ancestor species name and an estimate of how long ago the evolutionary event took place (in millions of years).

Leaves represent a most recent species. They are augmented with a name and whether the species is endangered.
The fine print

We have simplified a lot:

• The correct terms are “phylogenetic tree” and “speciation event”. Nodes are often called “taxonomic units”. This is an active area of research; see Wikipedia on “phylogenetic tree”.

• Evolutionary trees are built with incomplete data and theories, so there could be many different evolution trees.

• Leaves could represent extinct species that died off before splitting. Hence the term “most recent species”.
Representing evolutionary trees

Internal nodes each have exactly two children. Each internal node has the name of the common ancestor species and the estimated date of the evolutionary event.

Leaves have names and endangerment status of the most recent species.

The order of children does not matter.

The structure of the tree is dictated by a hypothesis about evolution.
Data definitions for evolutionary trees

;; An EvoTree (Evolution Tree) is one of:
;; * a RSpecies (recent species)
;; * a EvoEvent (evolutionary event)

(define-struct rspecies (name endangered))
;; A RSpecies is a (make-rspecies Str Bool)

(define-struct evoevent (name age lef tright))
;; A EvoEvent is a
;; (make-evoevent Str Num EvoTree EvoTree)

Note that the EvoEvent data definition uses a pair of EvoTrees.
Constructing the example evolutionary tree

(define-struct rspecies (name endangered))
(define-struct evoevent (name age left right))

(define human (make-rspecies "human" false))
(define chimp (make-rspecies "chimp" true))
(define rat (make-rspecies "rat" false))
(define crane (make-rspecies "crane" true))
(define chicken (make-rspecies "chicken" false))
(define worm (make-rspecies "worm" false))
(define fruit-fly (make-rspecies "fruit fly" false))
(define primate
  (make-evoevent "Primate" 5 human chimp))
(define mammal
  (make-evoevent "Mammal" 65 primate rat))
(define bird
  (make-evoevent "Bird" 100 crane chicken))
(define vertebrate
  (make-evoevent "Vertebrate" 320 mammal bird))
(define invertebrate
  (make-evoevent "Invertebrate" 530 worm fruit-fly))
(define animal
  (make-evoevent "Animal" 535 vertebrate invertebrate))
Derive the EvoTree template from the data definition.

`; evotree-template: EvoTree -> Any`

\[
\text{(define (evotree-template-template tree)}
  \text{(cond}
    \text{[(rspecies? tree (rspecies-template-template tree)]}
    \text{[(evoevent? tree (evoevent-template-template tree)])])}
\]

This is a straightforward implementation based on the data definition. It is also a good strategy to take a complicated problem (dealing with an EvoTree) and decompose it into simpler problems (dealing with a RSpecies or an EvoEvent).

Functions for these two data definitions are on the next slide.
;; rspecies-template: RSpecies -> Any
(define (rspecies-template rs)
  (... 
      ...(rspecies-name rs) 
      ...(rspecies-endangered rs))))

;; evoevent-template: EvoEvent -> Any
(define (evoevent-template ee)
  (... 
      ...(evoevent-name ee) 
      ...(evoevent-age ee) 
      ...(evoevent-left ee) 
      ...(evoevent-right ee)))
We know that (evoevent-left ee) and (evoevent-right ee) are EvoTree, so apply the EvoTree-processing function to them.

```scheme
;; evoevent-template: EvoEvent -> Any
(define (evoevent-template ee)
  (...  
    ...(evoevent-name ee)  
    ...(evoevent-age ee)  
    ...(evotree-template (evoevent-left ee))  
    ...(evotree-template (evoevent-right ee)))
```

evoevent-template uses evotree-template and evotree-template uses evoevent-template. This is called **mutual recursion**.
A function on EvoTrees

This function counts the number of recent species within an EvoTree.

;; (count-species tree): Counts the number of recent species (leaves) in the EvoTree tree.

;; count-species: EvoTree -> Nat

(define (count-species tree)
  (cond
   [(rspecies? tree) (count-recent tree)]
   [(evoevent? tree) (count-evoevent tree)]))

(check-expect (count-species animal) 7)
(check-expect (count-species human) 1)
;;; count-recent RSpecies -> Nat
(define (count-recent tree)
  1)

;;; count-evoevent EvoEvent -> Nat
(define (count-evoevent tree)
  (+ (count-species (evoevent-left tree))
      (count-species (evoevent-right tree))))
Traversing a tree

A tree traversal refers to the process of visiting each node in a tree exactly once. The increment example from binary trees is one example of a traversal.

We will now traverse an EvoTree to produce a list of all the names it contains.

We will solve this problem two different ways: using append and using accumulative recursion.
;; list-names: EvoTree -> (listof Str)
(define (list-names tree)
  (cond
   [(rspecies? tree) (list-rs-names tree)]
   [(evoevent? tree) (list-ee-names tree)]))

;; list-rs-names: RSpecies -> (listof Str)
(define (list-rs-names rs)
  (...(...(rspecies-name rs)))

;; list-ee-names: EvoEvent -> (listof Str)
(define (list-ee-names ee)
  (...(...(evoevent-name ee)
           (list-names (evoevent-left ee))
           (list-names (evoevent-right ee)))))
list-names with an accumulator

;;; list-names: EvoTree -> (listof Str)
(define (list-names tree)
  (list-names/acc tree empty))

;;; list-names/acc: EvoTree (listof Str) -> (listof Str)
(define (list-names/acc tree names)
  (cond
    [(rspecies? tree) (list-rs-names tree names)]
    [(evoevent? tree) (list-ee-names tree names)]))
;;; list-rs-names: RSpecies (listof Str) -> (listof Str)
(define (list-rs-names rs names)
  (cons (rspecies-name-name rs) names))

;;; list-ee-names: EvoEvent (listof Str) -> (listof Str)
(define (list-ee-names ee names)
  (cons (evoevent-name-name ee)
    (list-names/acc
      (evoevent-left ee)
      (list-names/acc (evoevent-right-right ee) names))))

(check-expect (list-names human) '("human"))
(check-expect (list-names mammal) '("Mammal" "Primate" "human"))
Practice problems with EvoTrees

Count the number of evolutionary events (internal nodes) with and age less than n. For example, the sample tree has 4 events that are less than 400 million years old.

Count the number of evolutionary events that occurred to produce a given recent species.

Find the evolutionary path between the root of a (sub)tree and a recent species. For example, the path from animal to rat is '(animal vertebrate mammal rat).

Modify list-names to produce the names of endangered species.
Binary expression trees
Binary expression trees

The expression
\[
\frac{(2 \times 6) + (5 \times 2)}{5 - 3}
\]
or
\[
\frac{(/ (+ (* 2 6) (* 5 2)) (- 5 3))}{-}
\]
can be represented as a tree:
Representing binary arithmetic expressions

Internal nodes each have exactly two children.
Leaves have number labels. Internal nodes have symbol labels. We care about the order of children.
The structure of the tree is dictated by the expression.
(define-struct binode (op left right))

;; A Binary arithmetic expression Internal Node (BINode)
;; is a (make-binode (anyof '* '+ '/ ')-) BinExp BinExp)
;; A binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode

Some examples of binary arithmetic expressions:

• 5
• (make-binode '* 2 6)
• (make-binode '+ 2 (make-binode '- 5 3))
(make-binode '/
  (make-binode '+
    (make-binode '* 2 6)
    (make-binode '* 5 2))
  (make-binode '-' 5 3))
Templates for binary arithmetic expressions

;; binexp-template: BinExp -> Any
(define (binexp-template ex)
  (cond
   [(number? ex) ...ex]
   [(binode? ex) (binode-template ex)]))

;; binode-template: BINode -> Any
(define (binode-template node)
  (... 
  ...(binode-op node)
  ...(binexp-template (binode-left node))
  ...(binexp-template (binode-right node))))
Evaluating expressions

;; (eval ex) evaluates the expression ex and produces its value.
;; eval: BinExp -> Num
(check-expect (eval 5) 5)
(check-expect (eval (make-binode '+ 2 5)) 7)
(check-expect (eval (make-binode '/'
                        (make-binode '-' 10 2)
                        (make-binode '+' 2 2))) 2)

(define (eval ex)
  (cond
   [(number? ex) ex]
   [(binode? ex) (eval-binode ex)]))
;;; (eval-binode node) evaluates the node.
;;; eval-binode BINode -> Num
(define (eval-binode node)
  (cond
    [(symbol=? '*(binode-op node))
     (* (eval (binode-left node))
        (eval (binode-right node)))]
    [(symbol=? '/(binode-op node))
     (/ (eval (binode-left node))
        (eval (binode-right node)))]
    [(symbol=? '+ (binode-op node))
     (+ (eval (binode-left node))
        (eval (binode-right node)))]
    [(symbol=? '- (binode-op node))
     (- (eval (binode-left node))
        (eval (binode-right node)))]))
Eval, refactored

\[
\text{(define (eval ex)} \\text{(cond [\text{(number? ex) ex]} [\text{(binode? ex)} \text{(eval-binode (binode-op ex) (eval (binode-left ex)) (eval (binode-right ex)))}])])}
\]

\[
\text{(define (eval-binode op left right)} \\text{(cond [\text{(symbol=? op '*) (* left right)] [\text{(symbol=? op '/) (/ left right)] [\text{(symbol=? op '+) (+ left right)] [\text{(symbol=? op '−) (− left right)]})])}}
\]