Binary trees

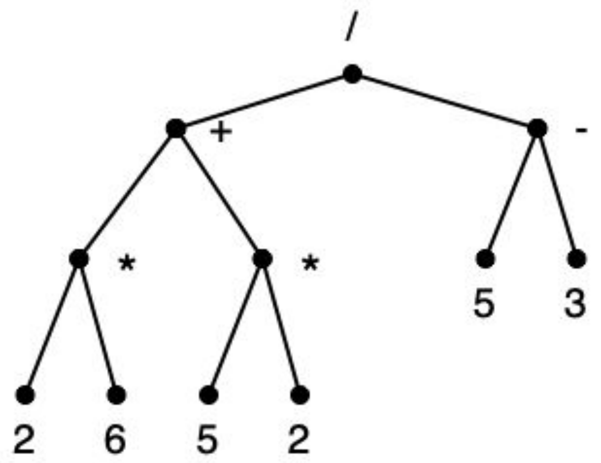
CS135 Lecture 13

L13.0 Trees as data abstractions

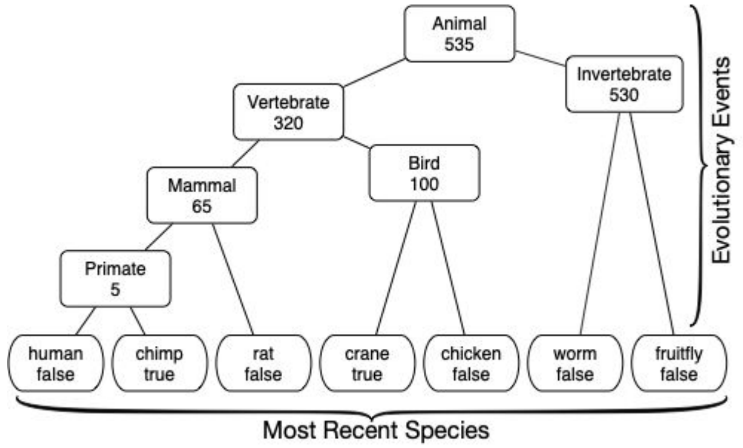
Trees

Like dictionaries, trees provide an **abstraction** for organizing data.

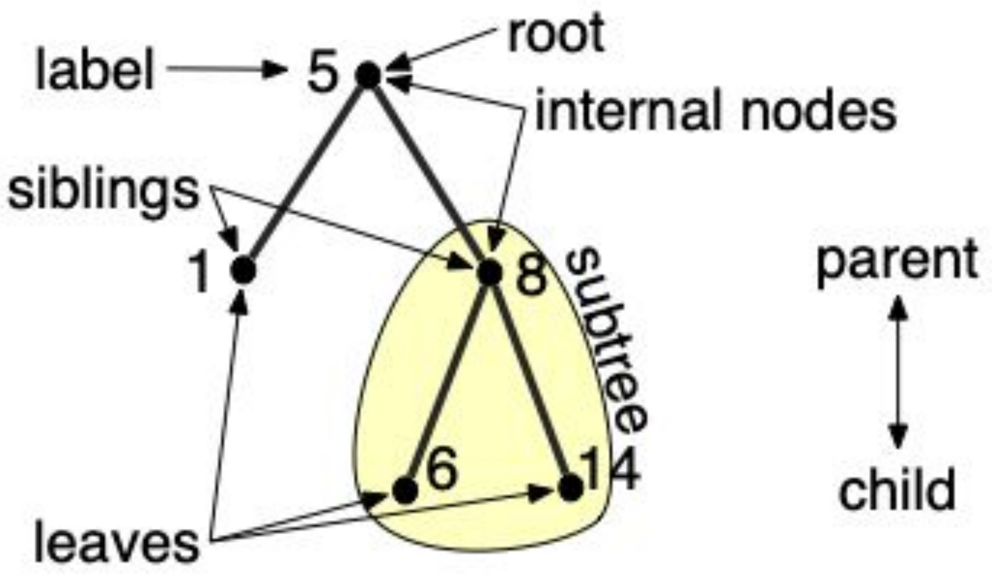
The expression $((2 * 6) + (5 * 2))/(5 - 3)$ can be represented by the tree:



Information related to the evolution of species can also be represented as a tree:



Tree terminology



A labeled point is called a “node”.

There are special names for some nodes:

- The “root” has no parent
- “Leaves” have no children
- “Internal node” ⇒ not a leaf

A line connecting a parent and child is called an “edge”

Characteristics of trees

A tree is a very general abstraction for organizing data. When trees are used in practice they often have more specific characteristics, for example:

- Number of children of internal nodes:
    - exactly two
    - at most two
    - any number
- Labels:
    - on all nodes
    - just on leaves
- Order of children (does it matter?)

L13.1 Binary trees

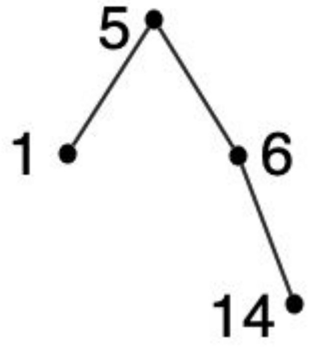
Binary trees

Binary trees are a fundamental concept in computer science.

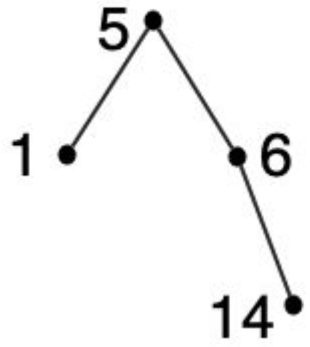
A binary tree is a tree with at most two children for each node.

In this module, we will label nodes with natural numbers.

The labels could also be symbols, strings, etc. and our basic data definition will work with any labels.



Binary tree data definition



```
;; A binary tree (BT) is one of
;; * empty
;; * (list Any BT BT)
```

The empty list is also the empty binary tree.

While our examples use natural numbers as labels, this definition works for any labels.

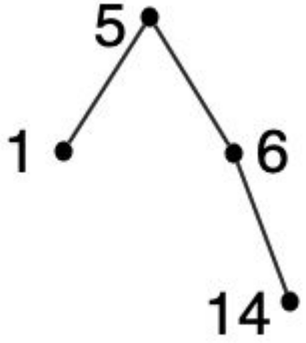
This is a recursive data definition with two recursive components. Is this going to work under the “Rules for Recursion”???

Binary tree helper functions

```
;; mk-node consumes a label, a left child, and a right child
;; mk-node produces a BT
;; mk-node: Any BT BT -> BT
(define (mk-node label left right) (list label left right))
;; get the label from a binary tree
;; get-label: BT -> Any
(define (get-label bt) (first bt))
;; get the left child from a binary tree
;; get-left: BT -> BT
(define (get-left bt) (second bt))
;; get the right child from a binary tree
;; get-right: BT -> BT
(define (get-right bt) (third bt))
```

Creating a binary tree

```
(define example-bt
   (mk-node 5 (mk-node 1 empty empty)
              (mk-node 6 empty (mk-node 14 empty empty))))

(check-expect
  example-bt
   (list 5 (list 1 empty empty)
           (list 6 empty (list 14 empty empty))))
```



Example: Count the nodes in a binary tree

```
;; Count the nodes in a binary tree
;; count-nodes: BT -> Nat
(define (count-nodes bt)
  (cond [(empty? bt) 0]
        [else (+ 1
                 (count-nodes (get-left bt))
                 (count-nodes (get-right bt)))]))
(check-expect (count-nodes example-bt) 4)
```

This doesn't follow the Rules for Recursion! What to do?

`count-nodes` works with any binary tree

```
(define sym-bt
  (mk-node 'apple
           (mk-node 'banana
                    (mk-node 'peach empty empty)
                    empty)
           (mk-node 'pear
                    empty
                    (mk-node 'apple empty empty))))

(check-expect (count-nodes sym-bt) 5)
```

Structural recursion

Structural recursion is a style of recursion in which the recursive structure of a function directly mirrors a data definition. For example, when applied to a list, the rules for recursion directly mirror our data definition for a list from back in L06.

```
;; A (listof X) is one of:
;; ★ empty
;; ★ (cons X (listof X))
```

Now that we have more complex data structures and data definitions, we are ready to generalize our rules to mirror this complexity. From the data definition of binary trees we see that we should recur separately on the left and right subtrees.

Rules for recursion on binary trees

The Rules for Recursion are there to ensure termination. For any binary tree, both the left and right “subtrees” will have fewer nodes. Therefore, we can ensure termination by:

1. Change at least one argument closer to termination while recurring.
2. When recurring on a binary tree use `(get-left tree)` and `(get-right tree)` and test termination with `empty?`
3. Arguments that aren’t involved in recursion can change.

Whenever you work with binary trees, you should follow these rules.

We will try to remind you on assignments and exams.

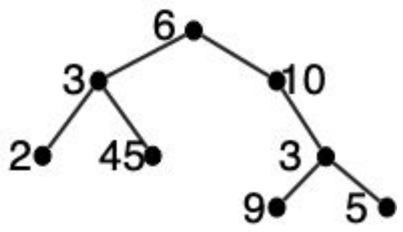
Example: increment nodes in a binary tree of numbers

```
;; Add one to every label in a binary tree of numbers
;; increment: BT -> BT
;; Requires: labels are numbers
(define (increment bt)
  (cond [(empty? bt) empty]
        [else (mk-node (add1 (get-label bt))
                       (increment (get-left bt))
                       (increment (get-right bt)))]))
(check-expect
 (increment example-bt)
 (list 6 (list 2 empty empty)
       (list 7 empty (list 15 empty empty))))
```

Searching binary trees

Does this tree contain 2? Does it contain 3? Does it contain 7?

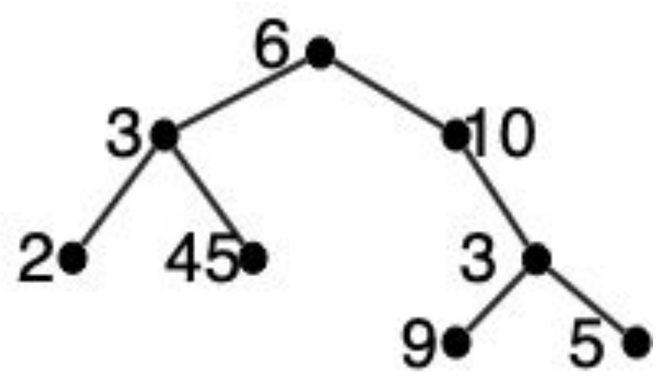
```
(define example-bt2
  (mk-node 6
          (mk-node 3
                  (mk-node 2 empty empty)
                  (mk-node 45 empty empty))
          (mk-node 10
                  empty
                  (mk-node 3
                          (mk-node 9 empty empty)
                          (mk-node 5 empty empty)))))
```



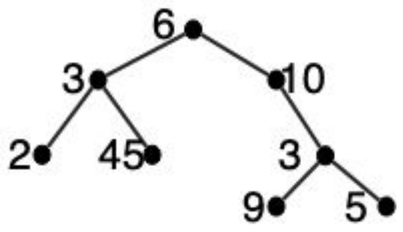
Searching binary trees

Our strategy:

- If the tree is empty, produce **`false`**.
- See if the root node contains the label we're looking for. If so, produce **`true`**.
- Otherwise, recursively search in the left subtree and in the right subtree. If either recursive search finds the key, produce **`true`**. Otherwise, produce **`false`**.

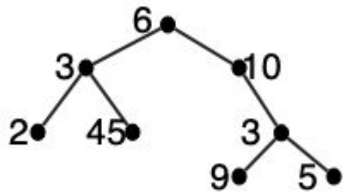


Searching a binary tree



```
;; Does the tree contain the label
;; bt-contains?: Num BT -> Bool
;; Requires: labels are numbers
(define (bt-contains? label bt)
  (cond [(empty? bt) false]
        [(= (get-label bt) label) true]
        [else (or (bt-contains? label (get-left bt))
                  (bt-contains? label (get-right bt)))]))
(check-expect (bt-contains? 2 example-bt2) true)
(check-expect (bt-contains? 3 example-bt2) true)
(check-expect (bt-contains? 7 example-bt2) false)
```

Find a path to a label



Write a function, **`bt-path`**, that searches for an label in the tree:

- If the label is not found produce **`empty`**
- If the label is found produce a list of the symbols **`'left`**, **`'right`**, and **`'found`** indicating the path from the root to the item.
- If there are duplicates in the tree, we prefer the leftmost

```
(check-expect (bt-path 2 example-bt2)
              (list 'left 'left 'found))
(check-expect (bt-path 3 example-bt2) (list 'left 'found))
(check-expect (bt-path 7 example-bt2) empty)
(check-expect (bt-path 9 example-bt2)
              (list 'right 'right 'left 'found))
(check-expect (bt-path 999 empty) empty)
```

Find a path to a label

```
;; Produce a path to a label
;; bt-path: Num BT -> (listof (anyof 'left 'right 'found))
;; Requires: labels are numbers
(define (bt-path label bt)
  (cond [(empty? bt) empty]
        [(= (get-label bt) label) (list 'found)]
        [(bt-contains? label (get-left bt))
         (cons 'left (bt-path label (get-left bt)))]
        [(bt-contains? label (get-right bt))
         (cons 'right (bt-path label (get-right bt)))]
        [else empty]))
```

L13.2 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 simplicity in the rest of the lecture, we assume labels are unique.

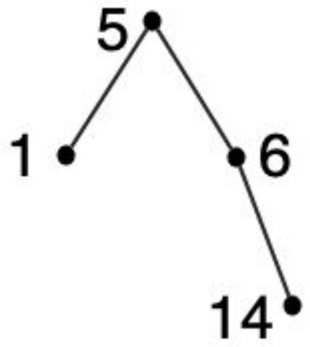
All the examples in this module use natural numbers as labels, but binary search trees can store any data type with an ordering operation (<, or equivalently >).

For any given collection of labels, there is more than one possible tree. How the keys are placed in a tree can improve the efficiency of searching the tree when compared to searching the same items in a list.

The BST ordering property: the label of a node is > than every label in the left subtree and < every label in the right subtree.

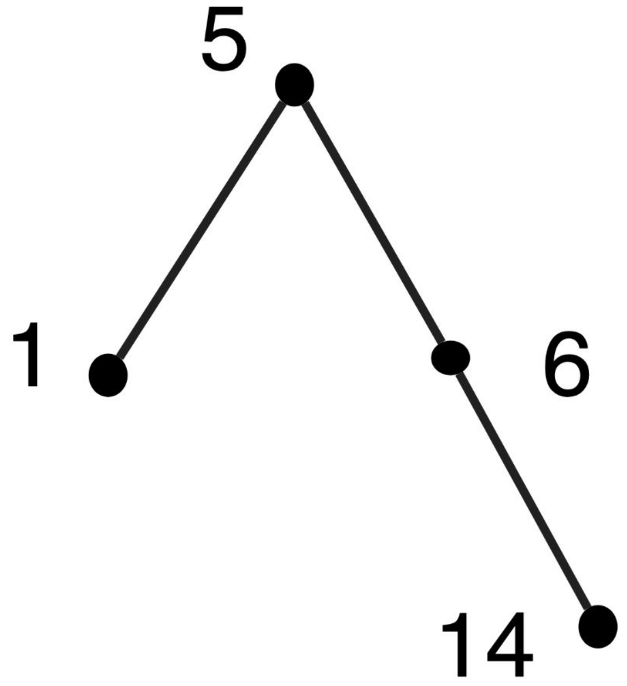
Binary search tree data definition

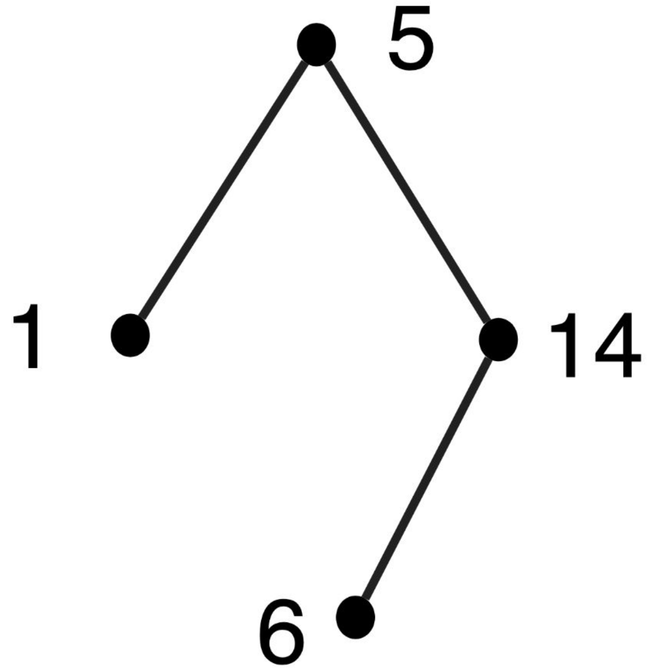
```
;; A binary search tree (BST) is one of
;; * empty
;; * (list Nat BST BST)
;; Requires: label is > than labels in left subtree
;;           and label is < labels in right subtree

(define example-bst
  (mk-node 5 (mk-node 1 empty empty)
             (mk-node 6 empty (mk-node 14 empty empty))))
```



Helper functions are the same as for binary trees (BT).

There can be multiple BSTs storing the same labels



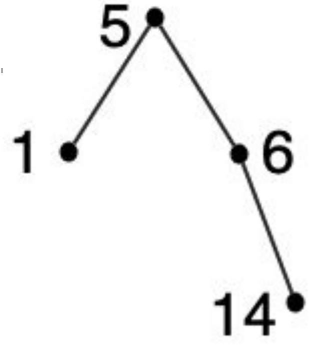


Making use of the ordering property

When searching binary tree one of the recursive applications to the subtrees can be avoided, which can be more efficient (sometimes considerably so).

To search a binary tree for the label *n*:

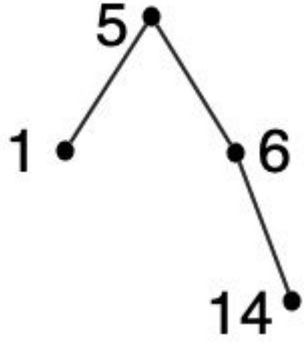
1. If the tree is empty, the label is not in the tree, produce **`false`**.
2. If the label at the root = n, produce **`true`**.
3. If the label at the root > n, recursively search the left subtree.
4. Otherwise, recursively search the right subtree.



```
(check-expect (bst-contains? 1 example-bst) true)
(check-expect (bst-contains? 6 example-bst) true)
(check-expect (bst-contains? 8 example-bst) false)
```

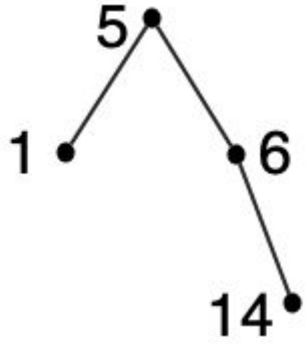
Searching a binary search tree

```
;; Does a binary search tree contain a label?
;; bst-contains?: Nat BST -> Bool
(define (bst-contains? n bst)
  (cond [(empty? bst) false]
        [(= (get-label bst) n) true]
        [(> (get-label bst) n)
         (bst-contains? n (get-left bst))]
        [else (bst-contains? n (get-right bst))]))

(check-expect (bst-contains? 1 example-bst) true)
(check-expect (bst-contains? 6 example-bst) true)
(check-expect (bst-contains? 8 example-bst) false)
```



Adding a label to a binary search tree

Write a function **`bst-add`** that consumes a label and a binary search tree and produces a new tree with the label added if it's not already in the tree.



```
(check-expect
  (bst-add 10 example-bst)
  (list 5 (list 1 empty empty)
          (list 6 empty
                  (list 14 (list 10 empty empty) empty))))
(check-expect (bst-add 6 example-bst) example-bst)
```

Adding a label to a binary search tree

```
;; Add a label to a binary search tree
;; bst-add: Nat BST -> BST
(define (bst-add n bst)
  (cond [(empty? bst) (mk-node n empty empty)]
        [(= (get-label bst) n) bst]
        [(> (get-label bst) n)
         (mk-node (get-label bst)
                  (bst-add n (get-left bst))
                  (get-right bst))]
        [else (mk-node (get-label bst)
                       (get-left bst)
                       (bst-add n (get-right bst)))]))
```

Binary search trees in practice

A “complete binary tree” is a binary tree in which all levels except possibly the last are completely filled, and all nodes in the last level are as far left as possible.

Searching a complete binary tree is $O(\log n)$ where n is the number of nodes.

If the BST has all left subtrees empty, it looks and behaves like a sorted list, and searching is $O(n)$.

In later courses, you will see ways to keep a BST “balanced” so that “most” nodes have non-empty left and right children. Those courses will also cover additional ways to analyze the efficiency of algorithms and operations on data structures.

Implementing a dictionary with a binary search tree?

A binary search tree can be use to store key-value pairs:

```
;; A binary search tree dictionary (BSTD) is one of
;; * empty
;; * (list (list Str Nat) BSTD BSTD)
;; Requires: binary search tree ordering properties
;;
;; mk-node produces a BSTD
;; mk-node: Str Nat BSTD BSTD -> BSTD
(define (mk-node key left right value) (...)
```

As an exercise, implement `lookup`, and `add.` Implementing `delete` is harder.

L13 Summary

L13: You should know

- Fundamental concepts and terminology for trees as a data abstraction
- Data definition and helper functions for binary trees
- How structural recursion mirrors a data definition
- How the data definition for binary trees induce their rules for recursion
- How to write functions that produce and consume binary trees
- Finding paths in binary trees
- Searching binary search trees by taking advantage of the ordering property

Lists in Racket are secretly binary trees (without values) where **`first`** *is the left tree and* **`rest`** *is the right tree. In this course, we are more explicit in our construction of binary trees as lists of lists, but under the hood the fundamental data structure used in Racket is a binary tree.*

L13: Allowed constructs

Newly allowed constructs:
none

Previously allowed constructs:
`( ) [ ] + - * / = < > <= >= ;`
`abs acos add1 and append asin atan boolean? char? char=? char<? check-expect check-within cond cons cons? cos define e else empty empty? even? exp expt false first inexact? integer? length list list? log max min not number? odd? or pi quotient rational? remainder rest reverse second sin sqr sqrt string? string=? string<? string->list list->string sub1 symbol? symbol=? tan third true zero?`
`listof Any anyof Bool Char Int Nat Num Rat Str Sym`