Trees

Readings: HtDP, sections 14, 15, 16.

Topics:

• Introductory examples and terminology [2–5]
• Binary trees [6–16]
• Binary search trees [17–27]
• Augmenting trees [28–48]
• Binary expression trees [49–57]
• General arithmetic expression trees [58–80]
• Nested lists [81–90]
Example: binary expression trees

The expression \(((2 \times 6) + (5 \times 2))/(5 - 3)\) can be represented as a tree:

```
          /
         /  
       +    -
      /     
     2  6   5  2
    /    /  /  
  2    6  5  2
```
Example: evolution trees

Information related to the evolution of species can also be represented as a tree. This tree shows how species evolved to become new species.
Tree terminology

- **root**
- **leaves**
- **parent**
- **child**
- **siblings**
- **internal nodes**
- **subtree**
- **label**
Tree-terminology

- **root**: top node in the tree (only node without a parent)
- **parent**: next node on the direct path up to the root
- **child**: next node on the path down to a leaf
- **sibling**: another child of your parent
- **leaf**: a node without any children
- **internal node**: a node with at least one child
Characteristics of trees

• Number of children of internal nodes:
  ★ exactly two
  ★ at most two
  ★ any number

• Labels:
  ★ on all nodes
  ★ just on leaves

• Order of children (matters or not)

• Tree structure (from data or for convenience)
Binary trees

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

Binary trees are a fundamental part of computer science, independent of what language you use.

Binary arithmetic expression trees and evolution trees are both examples of binary trees.

We’ll start with the simplest possible binary tree. It could be used to store a set of natural numbers.
Note: We will consistently use Nat\text{s} in our binary trees, but it could be a symbol, string, struct, ...

Drawing binary trees

```
      5
     / \  \
 1   6
     / \  \
 14  14
```

```
  5
 / \ \
1  6
    /  \
   14
```
Binary tree data definition

(define-struct node (key left right))

;; A Node is a (make-node Nat BT BT)

;; A binary tree (BT) is one of:

;; ★ empty

;; ★ Node

The node’s label is called “key” in anticipation of using binary trees to implement dictionaries.

What is the template?
Example functions on a binary tree

Let us fill in the template to make a simple function: count how many nodes in the BT have a key equal to \( k \):

\[
\text{;; count-nodes: BT Nat \(\rightarrow\) Nat}
\]

\[
\text{(define (count-nodes tree k)}
\]

\[
\text{(cond [(empty? tree) 0]}
\]

\[
\text{[else (+ (cond [(= k (node-key tree)) 1]}
\]

\[
\text{[else 0])}
\]

\[
\text{(count-nodes (node-left tree) k)}
\]

\[
\text{(count-nodes (node-right tree) k))])}
\]
Example functions on a binary tree

Add 1 to every key in a given tree:

;; increment: BT → BT
(define (increment tree)
  (cond
    [(empty? tree) empty]
    [else (make-node (add1 (node-key tree))
                    (increment (node-left tree))
                    (increment (node-right tree)))]))
Searching binary trees

We are now ready to try to search our binary tree for a given key. It will produce **true** if it’s in the tree and **false** otherwise.

Our strategy:

- See if the *root node contains the key* 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 binary trees

Now we can fill in our BT template to write our search function:

;; search-bt: Nat BT → Bool
(define (search-bt k tree)
  (cond [(empty? tree) false]
        [(= k (node-key tree)) true]
        [else (or (search-bt k (node-left tree))
                  (search-bt k (node-right tree))))])

Is this more efficient than searching a list?

CS 135 Fall 2019 11: Trees
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.
Find the path to a key

(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
   [(empty? tree) false]
   [(= k (node-key tree)) 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 which is inefficient.
Finding the path to a key

- For `search-bt-path` the base cases produces two different types:
  1. the `Bool`, `false`, if the key has not been found or
  2. a `(listof Sym)` if the key has been found.

- If one of the recursive applications (`node-left` vs. `node-right`) yielded a list, the appropriate symbol (`'left` vs. `'right`) is `cons'ed` to the answer.

- There may be two recursive applications of `search-bt-path` if `list?` is true, which is expensive.

- This function can be written to avoid two recursive calls...
;; search-bt-path: Nat BT → (anyof false (listof Sym))
(define (search-bt-path k tree)
  (cond
   [(empty? tree) false]
   [(= k (node-key tree)) empty]
   [else (choose-path (search-bt-path k (node-left tree))
                         (search-bt-path k (node-right tree))))])

(define (choose-path path1 path2)
  (cond 
    [(list? path1) (cons 'left path1)]
    [(list? path2) (cons 'right path2)]
    [else false])))
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

\[
\text{(define-struct node (key left right))}
\]

A Node is a \( (\text{make-node} \ \text{Nat} \ \text{BST} \ \text{BST}) \)

\text{requires: key > every key in left BST}
\text{key < every key in right BST}

The BST \textbf{ordering property}:

- \textbf{key} is \textit{greater than} every key in \textbf{left}.
- \textbf{key} is \textit{less than} every key in \textbf{right}.
A BST example

(make-node 5
  (make-node 1 empty empty)
  (make-node 6
    empty
    (make-node 14
      empty
      empty)))
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 \((\text{make-node } k \ l \ r)\), and \( k \) equals \( n \), then we have found it.

Otherwise it might be in either of the trees \( l, r \).

\textbf{Key Point:} we can use the ordering property of the BST to avoid one recursive call...
Searching in a BST

If $n < k$, then $n$ must be in $l$ if it is present at all, and we only need to recursively search in $l$.

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

Either way, we save one recursive function application.
Searching in a BST

;; (search-bst k t) produces true if k is in t; false otherwise.
;; search-bst: Nat BST → Bool

(define (search-bst k t)
  (cond [(empty? t) false]
        [(= k (node-key t)) true]
        [(< k (node-key t)) (search-bst k (node-left t))]
        [(> k (node-key t)) (search-bst k (node-right t))])))
Adding to a BST

How do we add a new key, \( k \), to a BST \( t \)?

If \( t \) is empty, then the result is a BST with only one node.

Otherwise \( t \) is of the form \( \text{make-node} \ n \ l \ r \).

If \( k = n \), the key is already in the tree and we can simply return \( t \).

If \( k < n \), then the new key must be added to \( l \), and if \( k > n \), then the pair must be added to \( r \). 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 BST is empty.

If the list is of the form (cons k lst), we add the key k 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

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

*Solution:* 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

So far nodes have been `(define-struct node (key left right))`. We can **augment** the node with additional data: `(define-struct node (key val left right))`.

- The name `val` is arbitrary – choose any name you like.
- The type of `val` 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 return the associated value, if found.
(define-struct node (key val left right))
;; A binary search tree dictionary (BSTD) is either
;; empty or (make-node Nat Str BSTD BSTD)

;; (search-bst-dict k t) produces the value associated with k
;; if k is in t; false otherwise.
;; search-bst: Nat BSTD → anyof(Str false)
(define (search-bst-dict k t)
  (cond[(empty? t) false]
       [(= k (node-key t)) (node-val t)]
       [(< k (node-key t)) (search-bst-dict k (node-left t))]
       [(> k (node-key t)) (search-bst-dict k (node-right t))]))
(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)
Evolutionary trees

*Evolutionary trees* are another kind of augmented tree.

![Evolutionary tree diagram]

- **Animal**: 535
- **Vertebrate**: 320
- **Mammal**: 65
- **Primate**: 5
- **Bird**: 100
- **Invertebrate**: 530
- **Human**: false
- **Chimp**: true
- **Rat**: false
- **Crane**: true
- **Chicken**: false
- **Worm**: false
- **Fruitfly**: false

**Most Recent Species**

**Evolutionary Events**
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’ve 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 left right))

;; 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))
Constructing the example evolutionary tree (cont)

(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))
EvoTree template
Derive the EvoTree template from the data definition.

;; evotree-template: EvoTree → Any
(define (evotree-template t)
  (cond [(rspecies? t) (rspecies-template t)]
        [(evoevent? t) (evoevent-template t)]))

This is a straightforward implementation based on the data definition. It’s 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.
EvoTree template

;; 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 EvoTrees, 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)) ....))
```

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

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

;; (count-species t): Counts the number of recent species
;; (leaves) in the EvoTree t.
;; count-species: EvoTree → Nat
(define (count-species t)
  (cond [(rspecies? t) (count-recent t)]
        [(evoevent? t) (count-evoevent t)]))

(check-expect (count-species animal) 7)
(check-expect (count-species human) 1)
A function on EvoTrees (cont)

;; count-recent RSpecies → Nat
(define (count-recent t)
  1)

;; count-evoevent EvoEvent → Nat
(define (count-evoevent t)
  (+ (count-species (evoevent-left t))
      (count-species (evoevent-right t))))
A function on EvoTrees

- **count-species** checks whether the node is a leaf (rspecies?) or an internal node (evoevent?) and applies the appropriate function.

- **count-recent** returns 1 for each leaf (recent species)

- **count-evoevent** adds the values obtained from applying count-species on its two children.

- **count-species** applies **count-evoevent** and **count-evoevent** applies **count-species**. This pairing is an example of mutual recursion.