Trees

Introductory examples and terminology
Binary trees
Binary search trees
Augmenting trees
Binary expression trees
General arithmetic expression trees
Nested lists
Readings

- Readings: HtDP, sections 14, 15, 16
Introductory examples and terminology
Example: binary expression trees

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

Parent:

Child:
Tree terminology: siblings

Siblings: nodes that have the same parent
Tree terminology: root

Root: node with no parent
Tree terminology: internal node

Internal node: node with children
Tree terminology: leaf

Leaf: node with no children
Tree terminology: subtree

Subtree: all descendants of a node
Tree terminology: label

Label: value attached to node
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
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 will start with the simplest possible binary tree. It could be used to store a set of natural numbers.
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.
Drawing binary trees

Note: We will consistently use Nat in our binary trees, but it could be a Sym, Str, make-struct, ...
Example functions on a binary tree

Let us fill in the template to make a simple function: count how many nodes in tree have a key equal to key:

\[
\text{;; count-nodes: BT Nat -> Nat}
\]

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

\[
\text{(cond}
\]

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

\[
\text{[else (+}
\]

\[
\text{(cond}
\]

\[
\text{[(= (node-key tree) key) 1]}
\]

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

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

\[
\text{(count-nodes (node-right tree) key))])}
\]
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)))]))
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 are 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**.
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]
   [(= (node-key tree) k) true]
   [else
    (or (search-bt k (node-left tree))
        (search-bt k (node-right tree)))]))

Is this more efficient than searching a list?
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
   [(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)
)

; 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 \((\text{make-node} \ \text{key} \ \text{left} \ \text{right})\), and \( \text{key} \) 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
(augmenting-tree)

We can augment the node with additional data:
(augmenting-tree)

The name value is arbitrary, choose any name you like.
The type of 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)