UMass Boston Computer Science

CS450 High Level Languages (section 2)

Tree Data Definitions

Wednesday, October 23, 2024

Logistics

- HW 7 out
  - <u>due</u>: Monday 10/28 12pm (noon) EDT

Create an image of a tree data structure with a halloween theme

???

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
```

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
```

```
(define (rev lst0)







                 )
```

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator ??? : ???
  ;; invariant: ???

  (define (rev/a lst acc ???)
    ???
  )

                   )
```

1. *Specify* **accumulator:** name, signature, invariant

2. *Define* internal "helper" fn with **accumulator** arg

3. *Call* "helper" fn, with initial **accumulator**

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator rev-so-far : List<X>
  ;; invariant: items of `lst0` “so far” in reverse order

  (define (rev/a lst acc ???)
    ???
  )

  (rev/a lst0 ???))
```

1. *Specify* **accumulator:** name, signature, invariant

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator rev-so-far : List<X>
  ;; invariant: items of `lst0` minus `rst` in reverse order

  (define (rev/a rst rev-so-far)
    ???
  )

  (rev/a lst0 ???))
```

2. *Define* internal "helper" fn with **accumulator** arg

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator rev-so-far : List<X>
  ;; invariant: items of `lst0` minus `rst` in reverse order

  (define (rev/a rst rev-so-far)
    (cond
     [(empty? rst) ...]
     [else (rev/a (rest rst) ... )
                  ... (first rst) ... rev-so-far ... ]))



  (rev/a lst0 ???))
```

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator rev-so-far : List<X>
  ;; invariant: items of `lst0` minus `rst` in reverse order

  (define (rev/a rst rev-so-far)
    (cond
     [(empty? rst) rev-so-far]
     [else (rev/a (rest rst) ... )
                ... (first rst) ... rev-so-far ... ]))

  (rev/a lst0 ???))
```

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator rev-so-far : List<X>
  ;; invariant: items of `lst0` minus `rst` in reverse order

  (define (rev/a rst rev-so-far)
    (cond
     [(empty? rst) rev-so-far]
     [else (rev/a (rest rst)
                  (cons (first rst) rev-so-far))]))

  (rev/a lst0 ???))
```

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator rev-so-far : List<X>
  ;; invariant: items of `lst0` minus `rst` in reverse order

  (define (rev/a rst rev-so-far)
    (cond
     [(empty? rst) rev-so-far]
     [else (rev/a (rest rst)
                  (cons (first rst) rev-so-far))]))

  (rev/a lst0 ???))
```

3. *Call* "helper" fn, with initial **accumulator**

Previously

In-class Coding 10/21: Accumulators

```
;; rev : List<X> -> List<X>
;; Computes a list with elements of the given list reversed
(define (rev lst0)

  ;; accumulator rev-so-far : List<X>
  ;; invariant: items of `lst0` minus `rst` in reverse order

  (define (rev/a rst rev-so-far)
    (cond
     [(empty? rst) rev-so-far]
     [else (rev/a (rest rst)
                  (cons (first rst) rev-so-far))]))

  (rev/a lst0 empty))
```

3.Call "helper" fn, with initial **accumulator**

Previously

Recursive Data Definitions

Template:
Recursive call matches recursion in data definition

```
;; A List<X> is one of:
;; - empty
;; - (cons X List<X>)
```

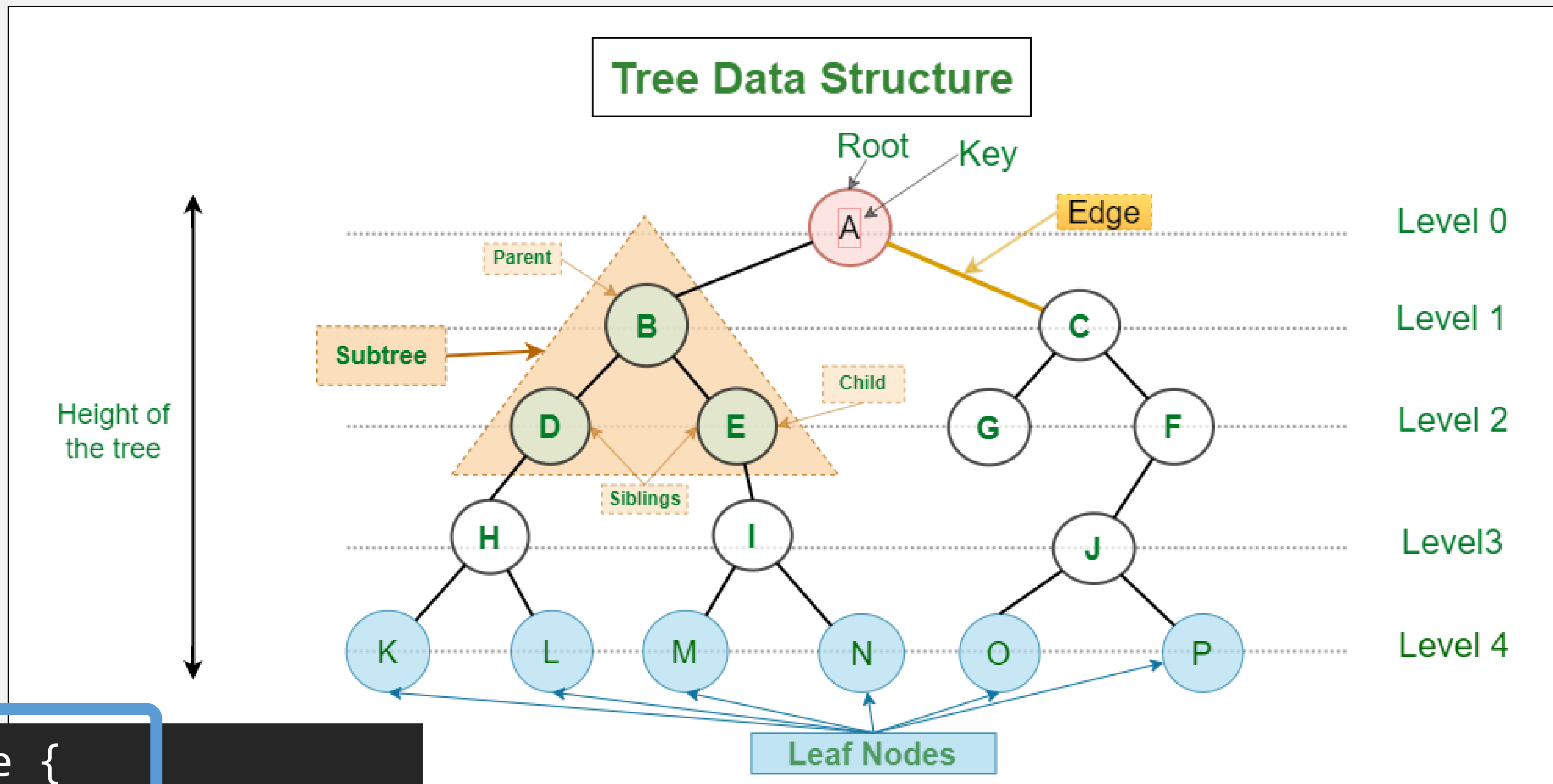
```
;; TEMPLATE for list-fn
;; list-fn : List<X> -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst)    ... (first lst)   ...
         ...  (list-fn (rest lst)) ....]))
```

Template:
Extract pieces of compound data

Template:
cond clause for each itemization item

Another Data Structure: Trees

Recursive!

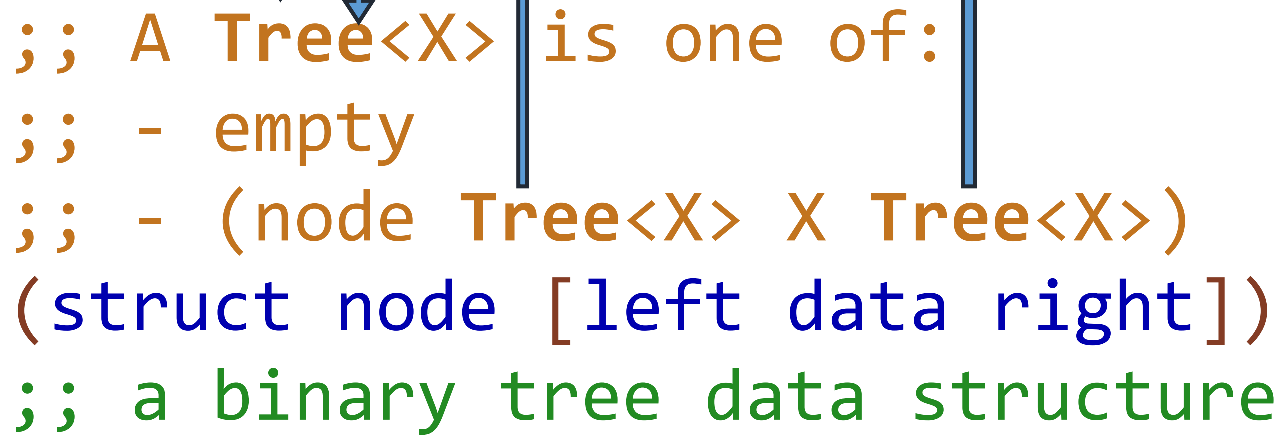


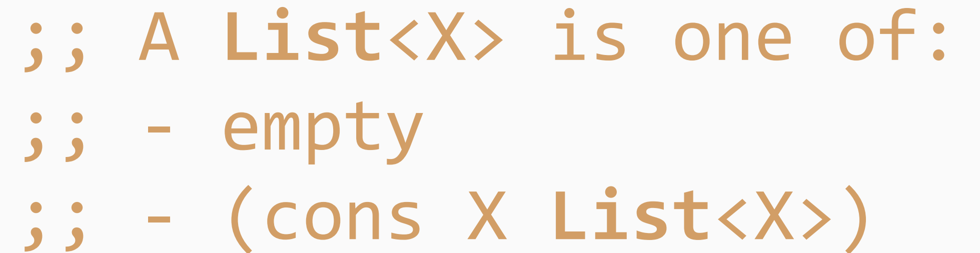

```
struct node {
        int data;
        struct node* left;
        struct node* right;
};
```

A **Tree** is a recursive data structure!

geeksforgeeks.org

More Recursive Data Definitions: Trees






```
(define (Tree? x) (or (empty? x) (node? x)))
```

(predicate only does top-level check)

```
struct node {
        int data;
        struct node* left;
        struct node* right;
};
```

More Recursive Data Definitions: Trees

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```

Template:
cond clause for each itemization item

Template:
Extract pieces of compound data

Template:
Recursive call matches recursion in data definition

Template?

In-class Coding #1: Write the Tree Template

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```

Template:
cond clause for each itemization item

Template:
Extract pieces of compound data

Template:
Recursive call matches recursion in data definition

- `git clone git@github.com:`**`cs450f24/in-class-10-23`**
- `git add` **`tree-template-`**<*Last*>**`-`**<*First*>`.rkt`
  - E.g., **`tree-template-`**`Chang-Stephen.rkt`
- `git commit` **`tree-template-`**`Chang-Stephen.rkt –m 'add chang tree template'`
- `git push origin main`
- Might need: **`git pull --rebase`**
  - If someone pushed before you, and your local clone is not at HEAD

In-class Coding #1: Tree Template

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```

```
;; tree-fn : Tree<X> -> ???
(define (tree-fn t)
  (cond
    [(empty? t) ...]
    [(node? t) ... (tree-fn (node-left t)) ...
                ... (node-data t) ...
       ... (tree-fn (node-right t)) ...]))
```

Template:
cond clause for each itemization item

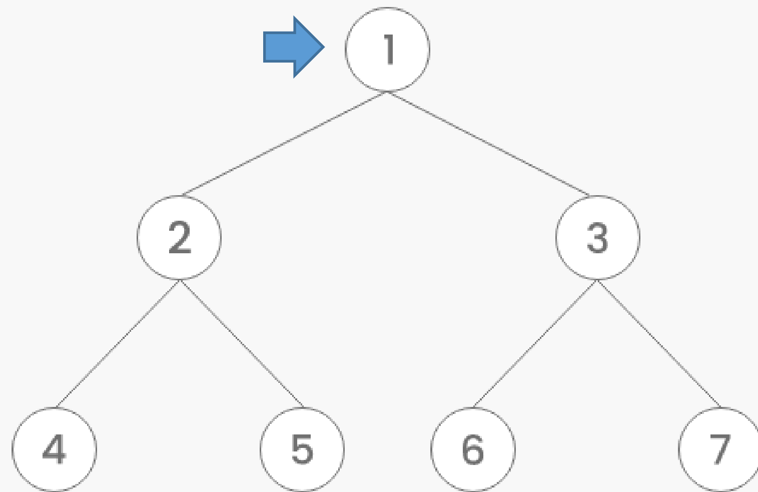
Template:
Recursive call(s) match recursion in data definition

Template:
Extract pieces of compound data

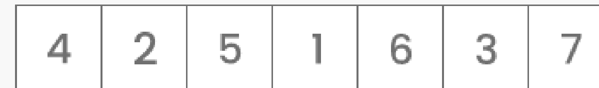
Tree Algorithms

Main difference: when to process root node

Tree Traversal Techniques

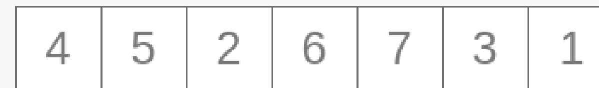


Inorder Traversal



Preorder Traversal

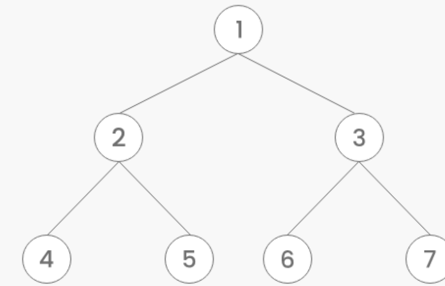
Postorder Traversal



geeksforgeeks.org

Tree Algorithms

Tree Traversal Techniques



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

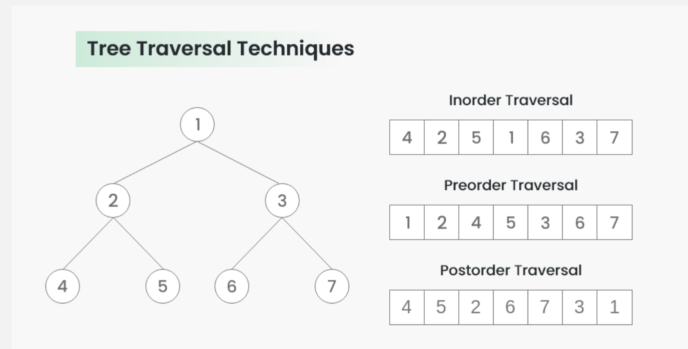
```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
;; tree->lst/post : Tree<X> -> List<X>
;; converts given tree to a list of values, by postorder
```

In-class Coding #2: Use the Template

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```




```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of
values, by inorder
```

```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of
values, by preorder
```

```
;; tree->lst/post : Tree<X> -> List<X>
;; converts given tree to a list of
values, by postorder
```

- git add tree-**traversal**-<*Last*>-<*First*>.rkt
  - E.g., tree-**traversal**-Chang-Stephen.rkt
- git commit tree-traversal-Chang-Stephen.rkt -m ‘add chang tree traversal’
- git push origin main
- Might need: git pull --rebase
  - If your local clone is not at HEAD

```
;; tree-fn : Tree<X> -> ???
(define (tree-fn t)
  (cond
    [(empty? t) ...]
    [(node? t) ... (tree-fn (node-left t)) ...
                    ... (node-data t) ...
               ... (tree-fn (node-right t)) ...]))
```

In-order Traversal

Tree Traversal Techniques

Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

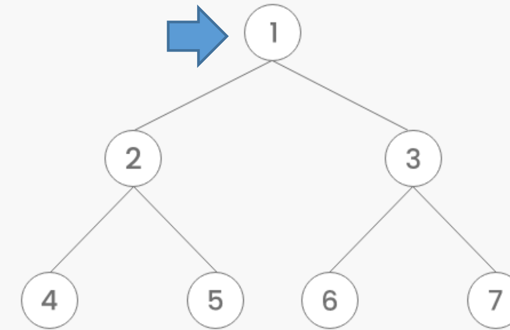
4	5	2	6	7	3	1

```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

```
(define (tree->lst/in t)
  (cond
    [(empty? t) empty]
    [(node? t) (append (tree->lst/in (node-left t))
                       (cons (node-data t)
                             (tree->lst/in (node-right t))))]))
```

Pre-order Traversal

Tree Traversal Techniques



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

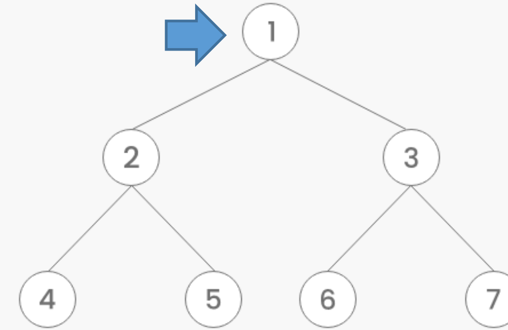
4	5	2	6	7	3	1

```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
(define (tree->lst/pre t)
  (cond
    [(empty? t) empty]
    [(node? t) (cons (node-data t)
                     (append (tree->lst/pre (node-left t))
                             (tree->lst/pre (node-right t))))]))
```

Post-order Traversal

Tree Traversal Techniques



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

```
;; tree->lst/post : Tree<X> -> List<X>
;; converts given tree to a list of values, by postorder
```

```
(define (tree->lst/post t)
  (cond
    [(empty? t) empty]
    [(node? t) (append (tree->lst/post (node-left t))
                       (tree->lst/post (node-right t))
                       (list (node-data t)))]))
```

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given pred returns true
;; for all values in given tree
```

```
(define TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

```
(check-true (tree-all? (curry < 4) TREE123))
```

Sometimes called **andmap** (for **Racket lists**) or **every** (for **JS Arrays**)

```
> (andmap positive? '(1 2 3))
#t
```

JavaScript Demo: Array.every()

```
1  const isBelowThreshold = (currentValue) => currentValue < 40;
2
3  const array1 = [1, 30, 39, 29, 10, 13];
4
5  console.log(array1.every(isBelowThreshold));
6  // Expected output: true
7
```

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given pred returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

Template:
cond clause for each itemization item

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given pred returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given pred returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

Template:
Recursive call(s) match recursion in data definition

Template:
Extract pieces of compound data

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given pred returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

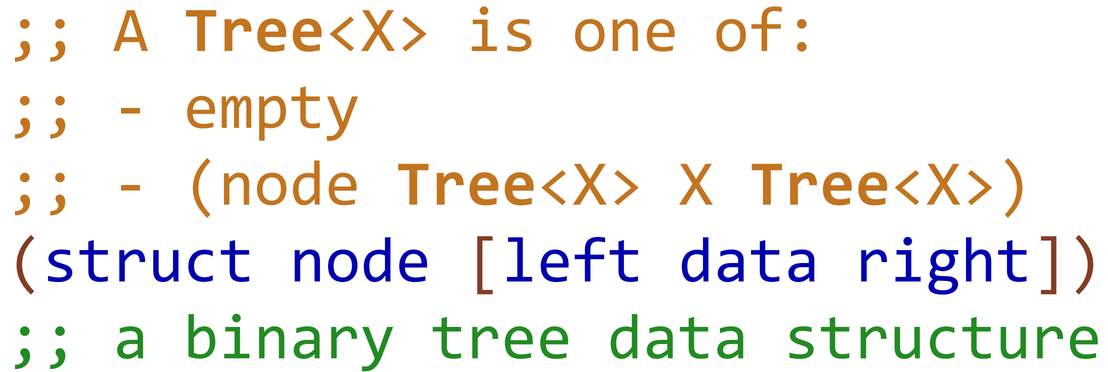
cond that evaluates to a boolean is just boolean arithmetic!

Combine the pieces with arithmetic to complete the function!

```
(define (tree-all? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (tree-all? p? (node-left t))
           (tree-all? p? (node-right t)))))
```

Tree Find?

Data Definitions With Invariants




```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where:
```

```
;; Invariant 1: for all values x in left tree, x < root val
```

```
;; Invariant 2: for all values y in right tree, y >= root val
```

Predicate?

Valid BSTs

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

```
(check-true (valid-bst? TREE123))
```

```
(check-false (valid-bst? (node TREE3 1 TREE2)))
```

In-class Coding #3: Valid BST

Hint: use `tree-all?`

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where:
;; Invariant 1:
;; for all values x in left tree, x < root
;; Invariant 2:
;; for all values y in right tree, y >= root
```

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

```
(check-true (valid-bst? TREE123))
```

```
(check-false (valid-bst? (node TREE3 1 TREE2)))
```

- git add **bst-valid**-<*Last*>-<*First*>.rkt
  - E.g., bst-valid-Chang-Stephen.rkt
- git commit bst-valid-Chang-Stephen.rkt -m ‘add chang valid-bst?’
- git push origin main
- Might need: git pull --rebase
  - If your local clone is not at HEAD

```
;; tree-fn : Tree<X> -> ???
(define (tree-fn t)
  (cond
    [(empty? t) ...]
    [(node? t) ... (tree-fn (node-left t)) ...
                   ... (node-data t) ...
               ... (tree-fn (node-right t)) ...]))
```

Valid BSTs

Hint: use `tree-all?`

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

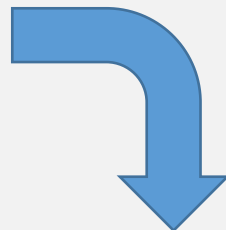
cond that evaluates to a boolean is just boolean arithmetic!

```
(define (valid-bst? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (tree-all? (curry > (node-data t)) (node-left t))
          (tree-all? (curry <= (node-data t)) (node-right t)))]))
```

```
(define (valid-bst? t)
  (or (empty? t)
      (and (tree-all? (curry > (node-data t)) (node-left t))
           (tree-all? (curry <= (node-data t)) (node-right t)))))
```

Data Definitions With Invariants

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```



"Deep" Invariants are enforced by individual functions

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where:
;; Invariant 1: for all values x in left tree, x < root val
;; Invariant 2: for all values y in right tree, y >= root val
```

```
(define (Tree? x) (or (empty? x) (node? x)))
```

Predicate?

(For contracts, BST should use "shallow" `Tree?` predicate, <u>not</u> "deep" `valid-bst?`)

BST Insert

Hint: use valid-bst? For tests

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define TREE1 (node empty 1 empty))
(define TREE2 (node empty 2 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

```
(check-equal? (bst-insert (bst-insert TREE2 1) 3)
              TREE123))
```

```
(check-true (valid-bst? (bst-insert TREE123 4)))
```

In-class Coding #4: BST Insert

Hint: use valid-bst? For tests

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where:
;; Invariant 1:
;; for all values x in left tree, x < root
;; Invariant 2:
;; for all values y in right tree, y >= root
```

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst,
;; result is still a bst
```

```
(define TREE1 (node empty 1 empty))
(define TREE2 (node empty 2 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

```
(check-equal? (bst-insert (bst-insert TREE2 1) 3) TREE123))
```

```
(check-true (valid-bst? (bst-insert TREE123 1)))
```

- git add **bst-insert**-<*Last*>-<*First*>.rkt
  - E.g., bst-insert-Chang-Stephen.rkt
- git commit bst-insert-Chang-Stephen.rkt -m ‘add chang bst-insert’
- git push origin main
- Might need: git pull --rebase
  - If your local clone is not at HEAD

```
;; tree-fn : Tree<X> -> ???
(define (tree-fn t)
  (cond
    [(empty? t) ...]
    [(node? t) ... (tree-fn (node-left t)) ...
                           ... (node-data t) ...
                ... (tree-fn (node-right t)) ...]))
```

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Template:
cond clause for each itemization item

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Template:
Recursive call matches recursion in data definition

Template:
Extract pieces of compound data

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
      (node (bst-insert (node-left t) x)
            (node-data t)
            (node-right t))
      (node (node-left t)
            (node-data t)
            (bst-insert (node-right t) x)))]))
```

Allowed because of data definition (invariant)

Result must maintain **BST invariant!**

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Result must maintain **BST invariant!**

Smaller values on left

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Result must maintain **BST invariant!**

Larger values on right