UMass Boston Computer Science

CS450 High Level Languages (section 2)

Tree Data Definitions, part 2

Monday, October 30, 2023

Logistics

- HW 5 out
  - **UPDATE**: split into two parts
  - ~~Part 1 due: **Sun 10/29 11:59 pm EST**~~
  - Part 2 due: **Sun 11/5 11:59 pm EST**

- HW 3 graded

HW3 Recap

```
(define-struct editor [pre post])
```

```
;; An Editor¹ is a structure:
;;   (make-editor¹ String String)
;; interp (make-editor¹ s t) describes an editor
;; whose visible text is (string-append s t) with
;; the cursor displayed between s and t
```

VS

```
;; An Editor² is a structure:
;;   (make-editor² Lo1S Lo1S)
;; interp (make-editor² l1 l2) describes an editor
;; whose visible text is (lst->str (append (rev l1) l2))
;; with the cursor displayed in between
```

```
;; An Lo1S is one of:
;; - '()
;; - (cons 1String Lo1S)
```

HW3 Recap: Create Instances

```
;; An Editor¹ is a structure:
;;   (make-editor¹ String String)
;; interp (make-editor¹ s t) describes an editor
;; whose visible text is (string-append s t) with
;; the cursor displayed between s and t
```

```
(make-editor¹ "Hello" "World!")
```

vs

```
(make-editor² (rev (str->lst "Hello"))
              (str->lst "World!"))
```

```
;; An Editor² is a structure:
;;   (make-editor² Lo1S Lo1S)
;; interp (make-editor² l1 l2) describes an editor
;; whose visible text is (lst->str (append (rev l1) l2))
;; with the cursor displayed in between
```

```
(create-editor² "Hello" "World!")
```

HW3 Recap: Pros / Cons

2-string representation

- Construct directly with strings
- Easier to build full string, and render

List of chars (1str) representation

- More complicated to construct
- Need extra string constructor
- More complicated to build full string, and render

HW3 Recap: Editing

```
(define (editor-left¹ ed)
  (make-editor
    (string-drop-last (editor-pre ed))
    (string-append (string-last (editor-pre ed))
                   (editor-post ed))))
```

```
;; An Editor¹ is a structure:
;;   (make-editor¹ String String)
;; interp (make-editor¹ s t) desc
;; whose visible text is (string-append s t) with
;; the cursor displayed between s and t
```

```
(define (editor-left² ed)
  (make-editor
    (rest (editor-pre ed))
    (cons (first (editor-pre ed))
          (editor-post ed))))
```

```
;; An Editor² is a structure:
;;   (make-editor² Lo1S Lo1S)
;; interp (make-editor² l1 l2) desc
;; whose visible text is (lst->str (append (rev l1) l2))
;; with the cursor displayed in between
```

HW3 Recap: Pros / Cons

2-string representation

- Construct directly with strings
- Easier to build full string, and render
- Editor manipulation via string arithmetic
- Strings not as easy to manipulate
  - E.g., "first", "rest", "drop last"
- Theoretically slower and uses more memory

List of chars (1str) representation

- More complicated to construct
- Need extra string constructor
- More complicated to build full string, and render
- Editor manipulation via list functions
- Lists easier to manipulate
  - E.g., `first` and `rest` (reversed list)
- Theoretically more performant and uses less memory

Important: In practice, not allowed to say that something is slow or fast unless you've profiled it!

Racket **for** expressions

Generic "sequence"
(number, most data structures ...)

```
(for/list ([x lst]) (add1 x))
```
⇩
```
(map add1 lst)
```

```
(for/list ([x n]) (add1 x))
```
⇩
```
(build-list n add1)
```

```
(for/list ([x lst] #:when (odd? x)) (add1 x))
```
⇩
```
(filter odd? (map add1 lst))
```

```
(for/sum ([x lst] #:when (odd? x)) (add1 x))
```
⇩
```
(foldl + 0 (filter odd? (map add1 lst)))
```

Note:
These are still expressions!

Lots of variations!
(see docs)

Racket **for*** expressions

“nested” for loops

```
> (for* ([i '(1 2)]
         [j "ab"])
    (display (list i j)))
(1 a)(1 b)(2 a)(2 b)
```

```
> (for*/list ([i '(1 2)]
              [j "ab"])
    (list i j))
'((1 #\a) (1 #\b) (2 #\a) (2 #\b))
```

```
(for*/list (for
(for*/lists (id
  body-or-break
(for*/vector ma
(for*/hash (for
(for*/hasheq (f
(for*/hasheqv (
(for*/hashalw (
(for*/and (for-
(for*/or (for-c
(for*/sum (for-
(for*/product (
(for*/first (fo
(for*/last (for
(for*/fold ([ac
  body-or-break
(for*/foldr ([a
             (for
```

Lots of variations! (see docs)

Last Time

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
```

Last Time

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)) ...]))
```

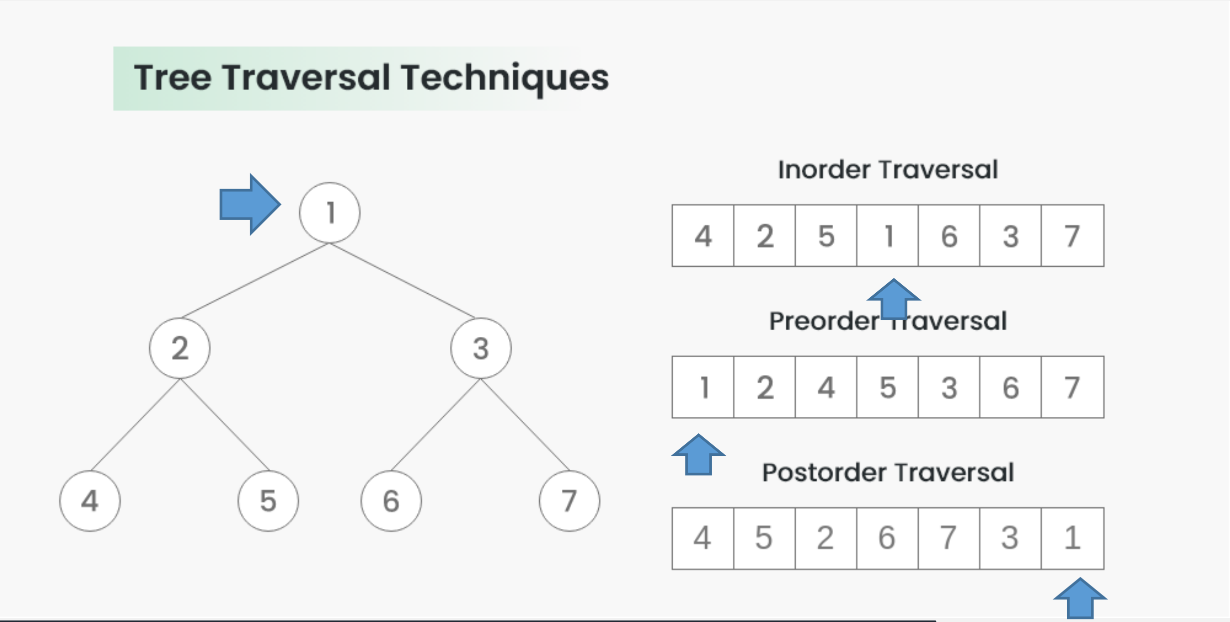
Template:
cond clause for each itemization item

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

Template:
Extract pieces of compound data

Last Time

Tree Algorithms




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

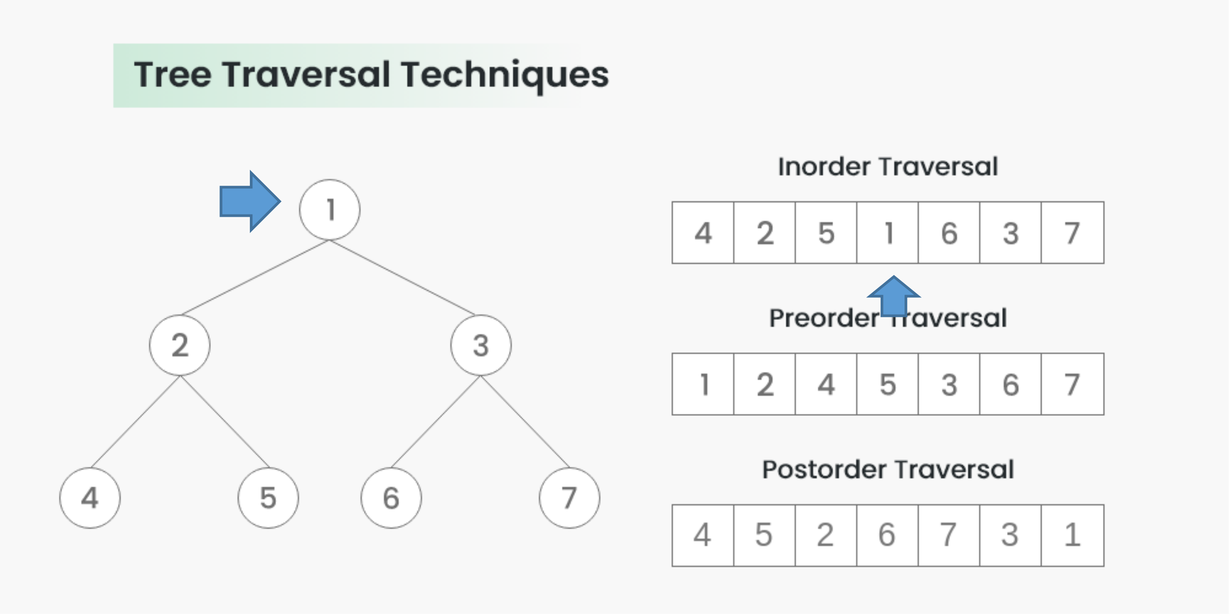
```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

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

Main difference: when to process root node

Last Time

In-order Traversal

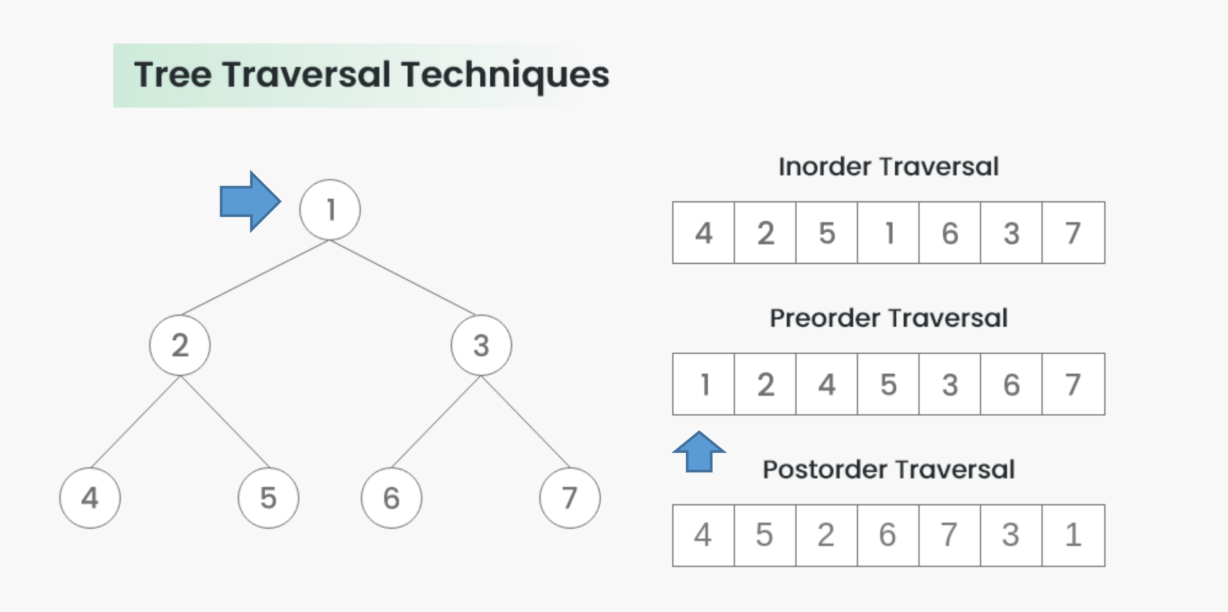



```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

```
(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))))]))
```

Last Time

Pre-order Traversal

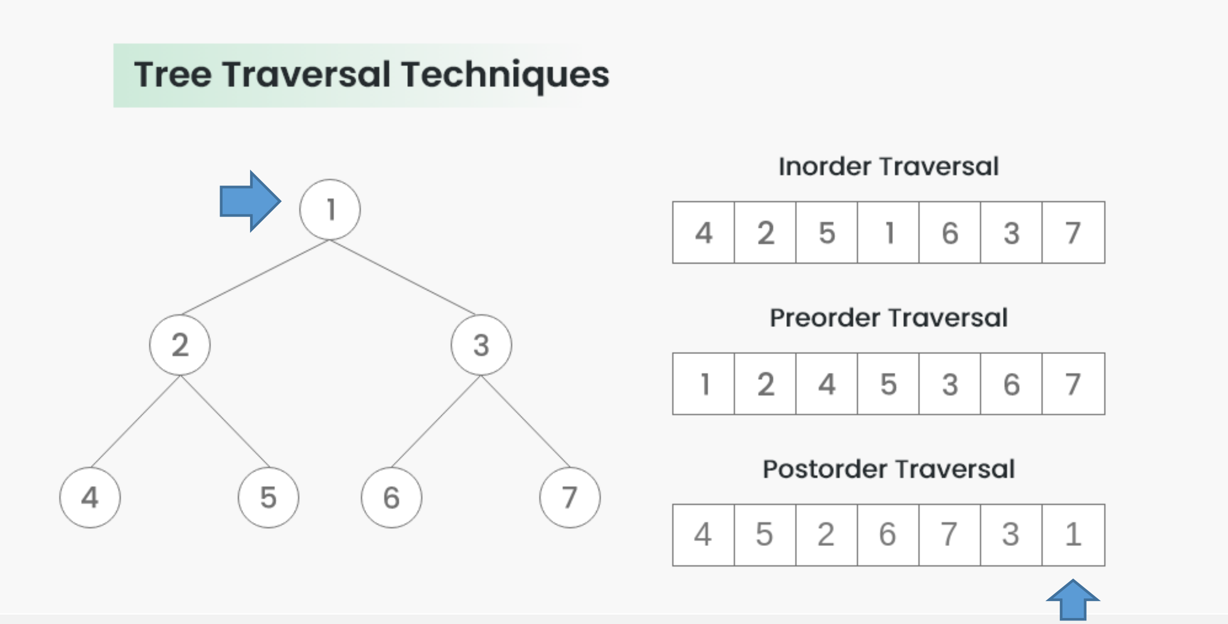



```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
(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))))]))
```

Last Time

Post-order Traversal




```
;; 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)))]))
```

Last Time

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()

```
const isBelowThreshold = (currentValue) => currentValue < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold));
// Expected output: true

```

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)))]))
```

Combine the pieces with arithmetic to complete the function!

cond that evaluates to a boolean is just boolean arithmetic!

```
(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?

- Do we have to search the entire tree?

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
```

```
;; 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

- `git clone git@github.com:`**`cs450f23/lecture15-inclass`**
- `git add bst-valid-`<*your last name*>`.rkt`
  - E.g., `bst-valid-chang.rkt`
- `git commit bst-valid-chang.rkt –m ‘add chang bst-valid? fn’`
- `git push origin main`
- Might need: `git pull --rebase`
  - If someone pushed before you, and your local clone is not at HEAD

(Will get quiz / participation extra credit)

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)))
```

Remember:
boolean arithmetic doesn't use cond

- `git add bst-valid-`*<your last name>*`.rkt`
  - E.g., `bst-valid-chang.rkt`
- `git commit bst-valid-chang.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 <u>use “shallow”</u> `tree?` predicate, <u>not “deep”</u> `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 TREE2 (node empty 2 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 TREE2 (node empty 2 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)))
```

- git add **bst-insert**-<*your last name*>.rkt
  - E.g., bst-insert-chang.rkt
- git commit bst-insert-chang.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 (< x (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!**

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 (< x (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

Tree Find?

- Do we have to search the entire tree?

BST Find

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

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

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

```
(check-true (bst-has? TREE123 1))
(check-false (bst-has? TREE123 4))
```

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

In-class Coding #5: BST-has?

```
;; 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-has?: BST<X> X -> Bool
;; Returns true if the given BST
;; has the given value
```

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

```
(check-true (bst-has? TREE123 1))
(check-false (bst-has? TREE123 4))
```

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

- git add **bst-has-**<*your last name*>.rkt
  - E.g., bst-has-chang.rkt
- git commit bst-has-chang.rkt
  –m ‘add chang bst-has?’
- 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)) ...]))
```

Check-In Quiz 10/30
on gradescope

(due 1 minute before midnight)