UMass Boston Computer Science
CS450 High Level Languages (section 2)
Tree Data Definitions

Wednesday, October 25, 2023
Logistics

• HW 4 in
  • due: Sun 10/22 11:59 pm EST

• 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
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)) ....]]))
Another Data Structure: Trees

```
#include <stdio.h>

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

A Tree is a recursive data structure!
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

(define (tree? x) (or (empty? x) (node? x)))

(struct node {
  int data;
  struct node* left;
  struct node* right;
});
A Tree\textless{}X\textgreater{} is one of:
- empty
- (node Tree\textless{}X\textgreater{} X Tree\textless{}X\textgreater{})
(struct node [left data right])
- a binary tree data structure
In-class Coding

- `git clone git@github.com:cs450f23/lecture14-inclass`

- `git add tree-template-<your last name>.rkt`
  - E.g., `tree-template-chang.rkt`

- `git commit tree-template-chang.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

(Will get quiz / participation extra credit)
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

• git clone git@github.com:cs450f23/lecture14-inclass
• git add tree-template-<your last name>.rkt
  • E.g., tree-template-chang.rkt
• git commit tree-template-chang.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
- Recursive call(s) match recursion in data definition
- Extract pieces of compound data
Tree Algorithms

Tree Traversal Techniques

Main difference: when to process root node

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 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
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-<your last name>.rkt
  • E.g., tree-traversal-chang.rkt
• git commit tree-traversal-chang.rkt
  –m ‘add chang tree traversal’
• git push origin main
• Might need: git pull --rebase
  • If your local clone is not at HEAD

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

(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

(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)
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)))]))
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))))])

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?
A Tree\<X\> is one of:
- empty
- (node Tree\<X\> X Tree\<X\>)
  
  (struct node [left data right])

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, \(y \geq \) root val
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

```racket
;; 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

(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)))
```

```racket
;; 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)))
```

```racket
;; 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)) ...]])
```

- `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

- `hint: use tree-all?`
Valid BSTs

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

(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))))])
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

(define (tree? x) (or (empty? x) (node? x)))

 Predicate?

(For contracts, BST should use “shallow” tree? predicate, not “deep” valid-bst?)
BST Insert

;;; 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

;; 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)))

;; 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)) ...]]))

• 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
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))))])
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)))))}
BST Insert

(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)))]))
Check-In Quiz 10/25
on gradescope

(due 1 minute before midnight)