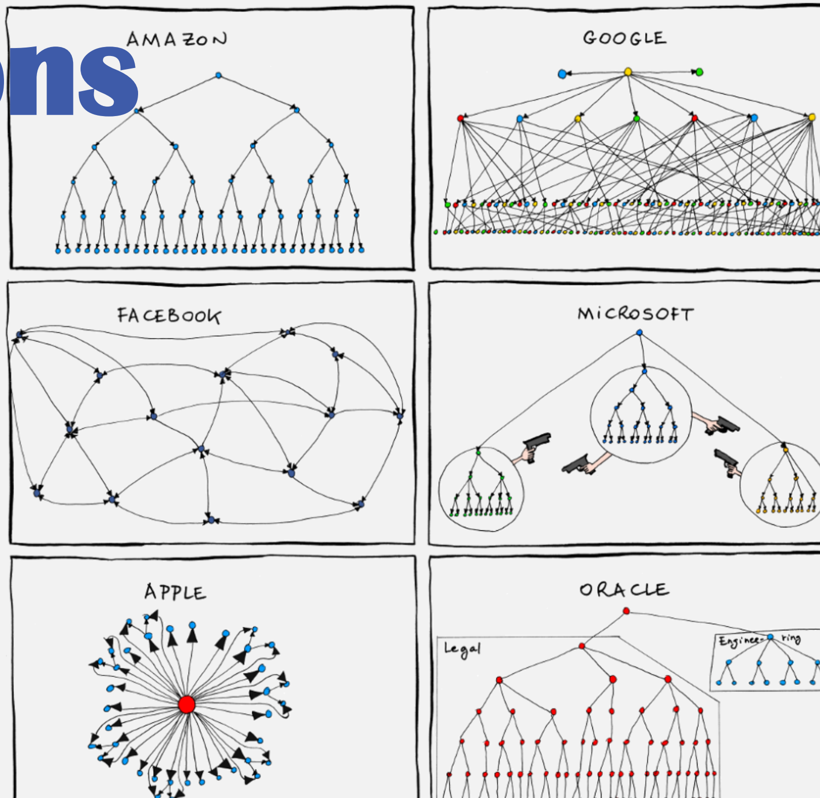
UMass Boston Computer Science

CS450 High Level Languages

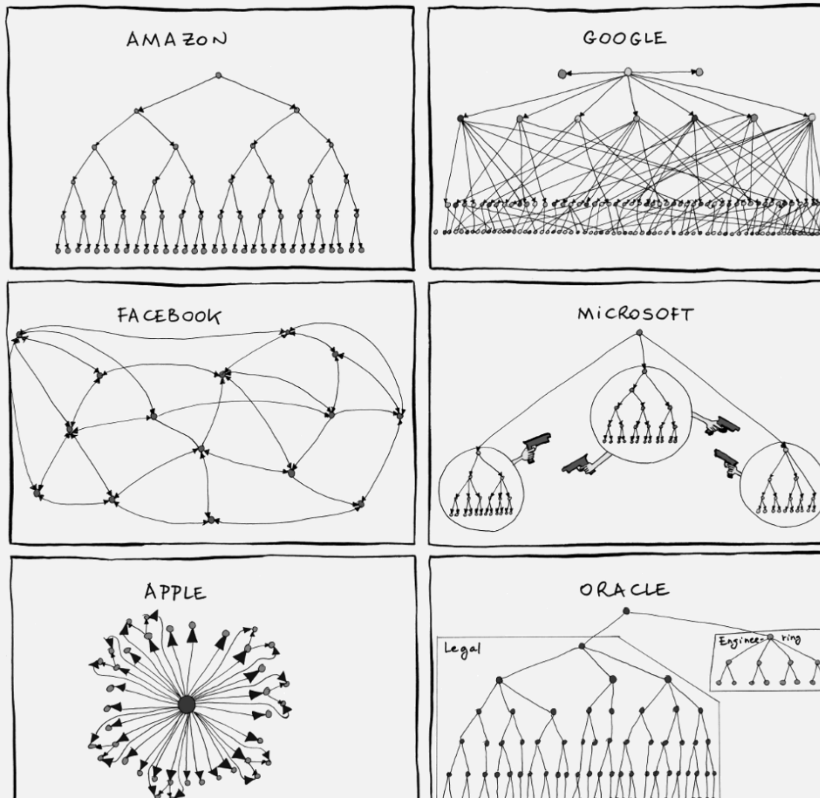
Tree Data Definitions

Thursday, March 27, 2025



Logistics

- HW 7 out
  - due: Tues 4/1, 11am EST



https://bonkersworld.net/organizational-charts

Previously

Recursive Data Definitions

Template:
Recursive call matches recursion in data definition

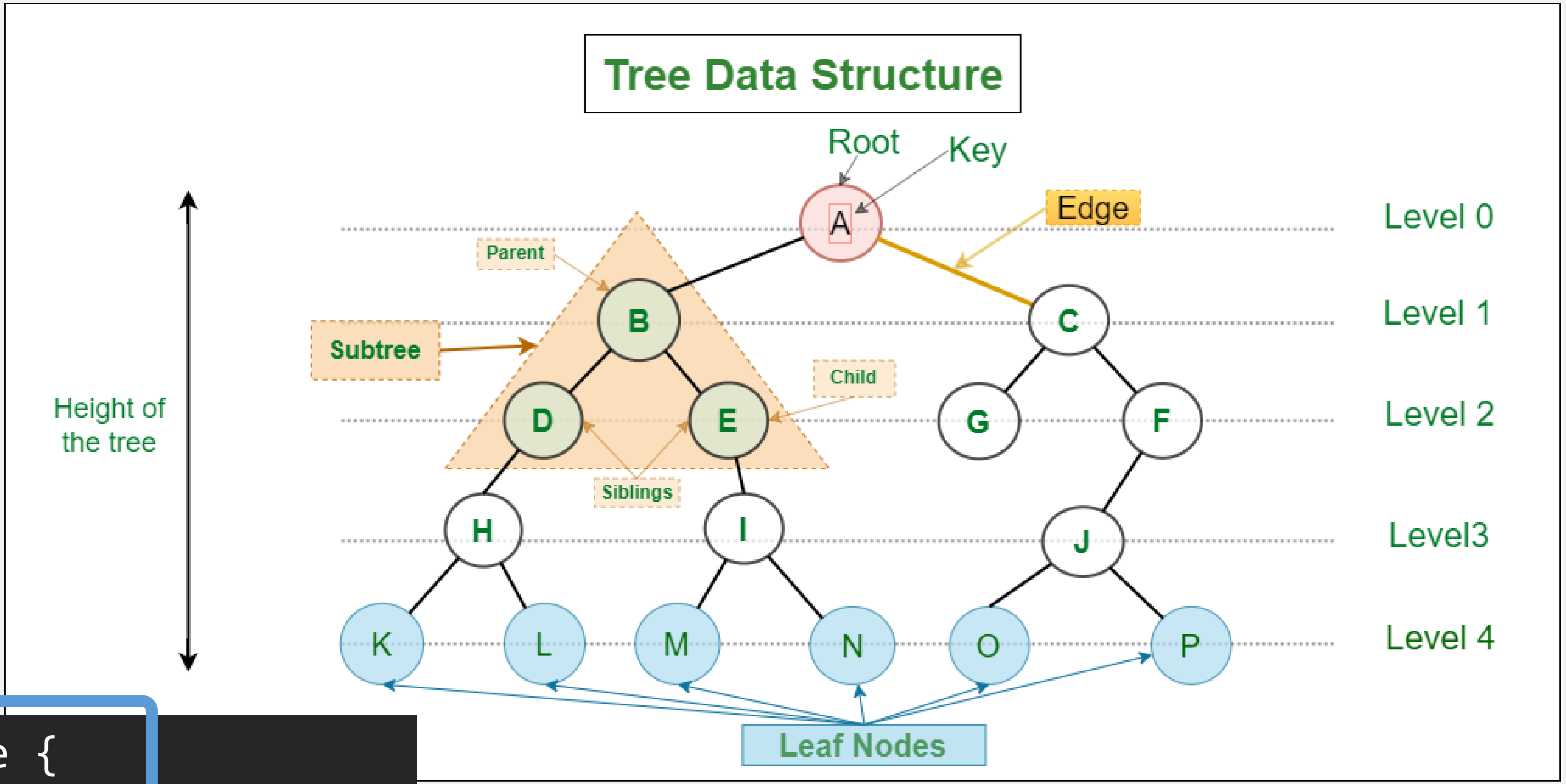
```
;; A List<X> is one of:
;; - empty
;; - (cons X List<X>)
```

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

Template:
cond clause for **each** itemization item

Template:
Extract pieces of compound data

Recursive!
Another Data Structure: Trees




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

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

geeksforgeeks.org

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
```

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

```
(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

on gradescope

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)) ...]))
```

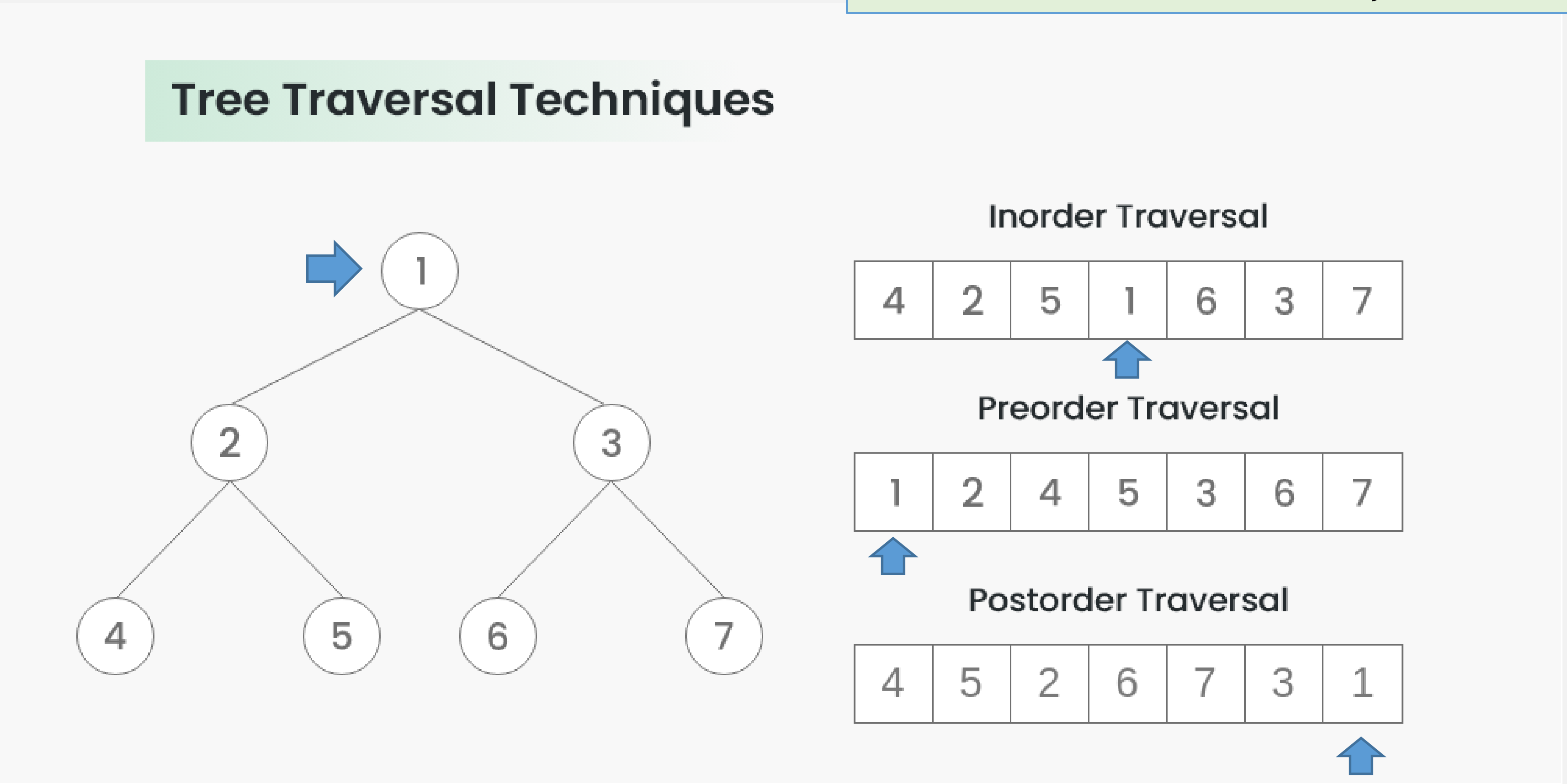
Template:
cond clause for each itemization item

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

Template:
Extract pieces of compound data

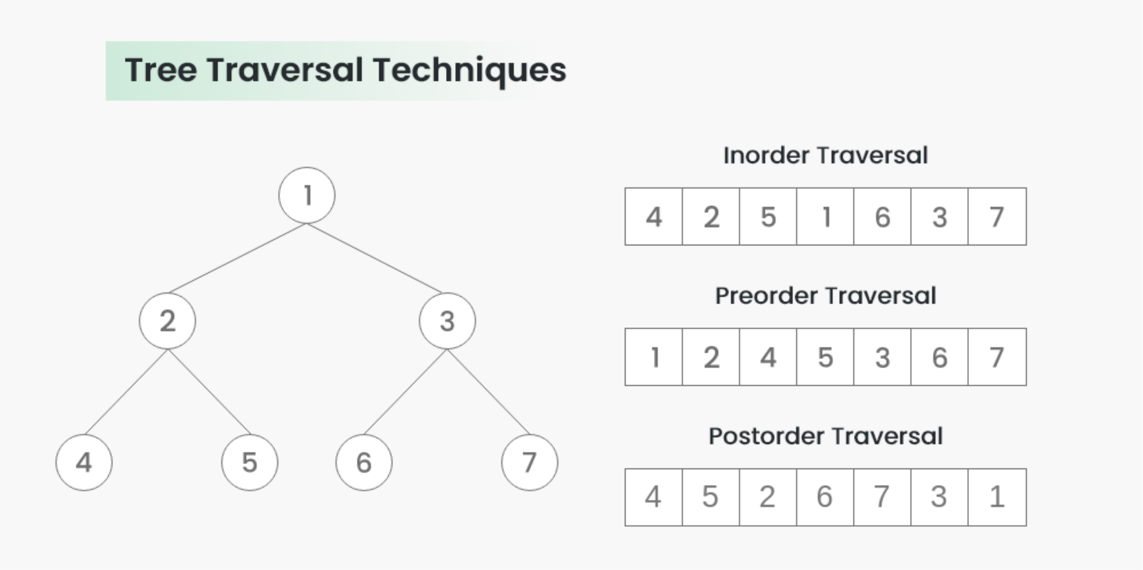
Tree Algorithms

Main difference: when to process root node



geeksforgeeks.org

Tree Algorithms



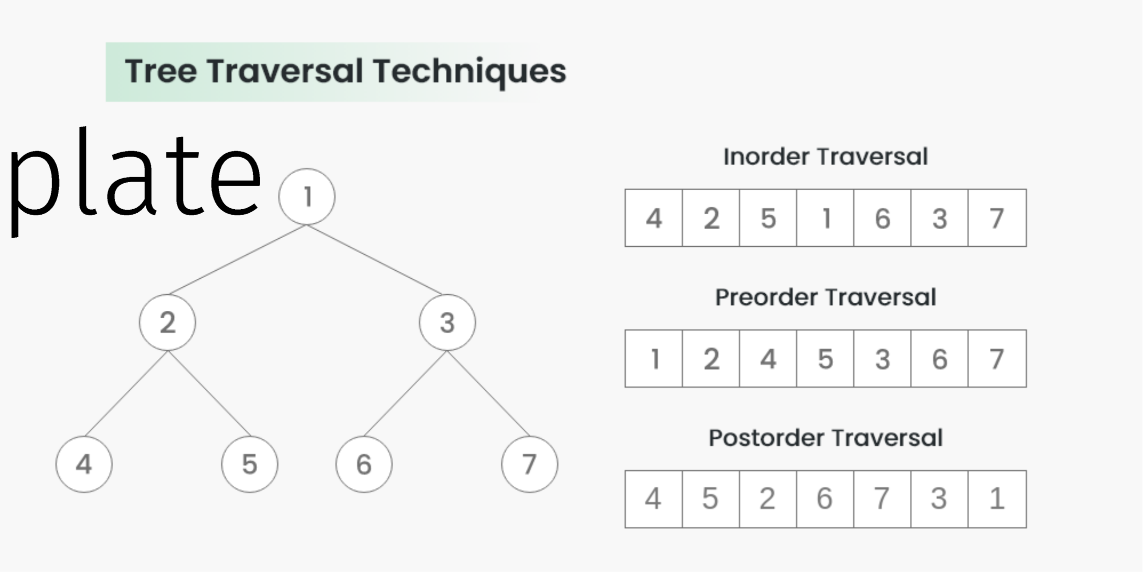

```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

```
;; 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
```

Tree Fns - 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
```

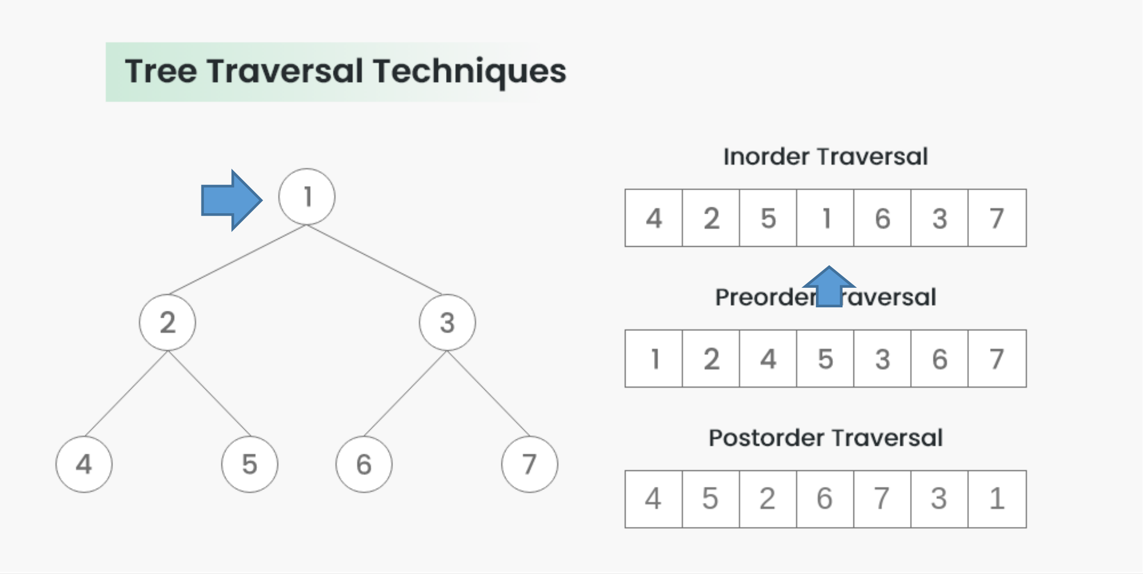



```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

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

In-order Traversal

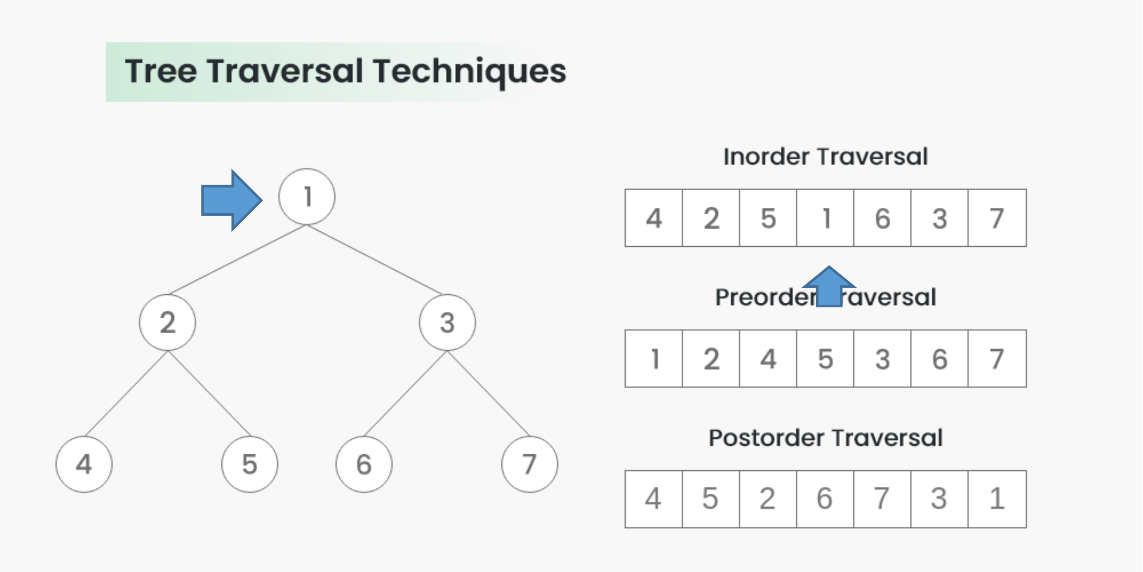
```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```




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

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

In-order Traversal

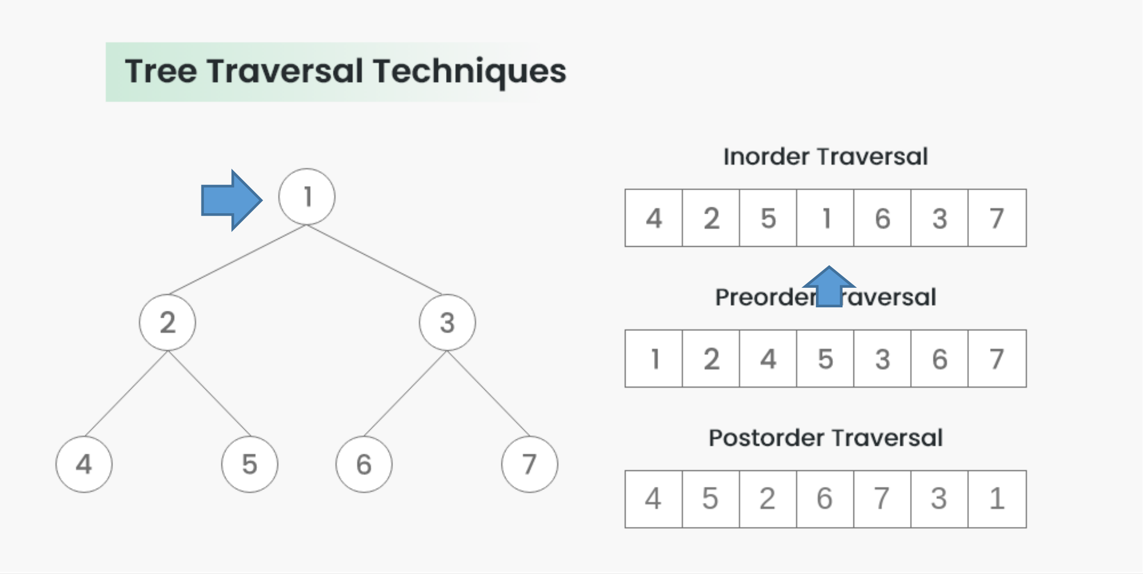



```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

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

Not list!

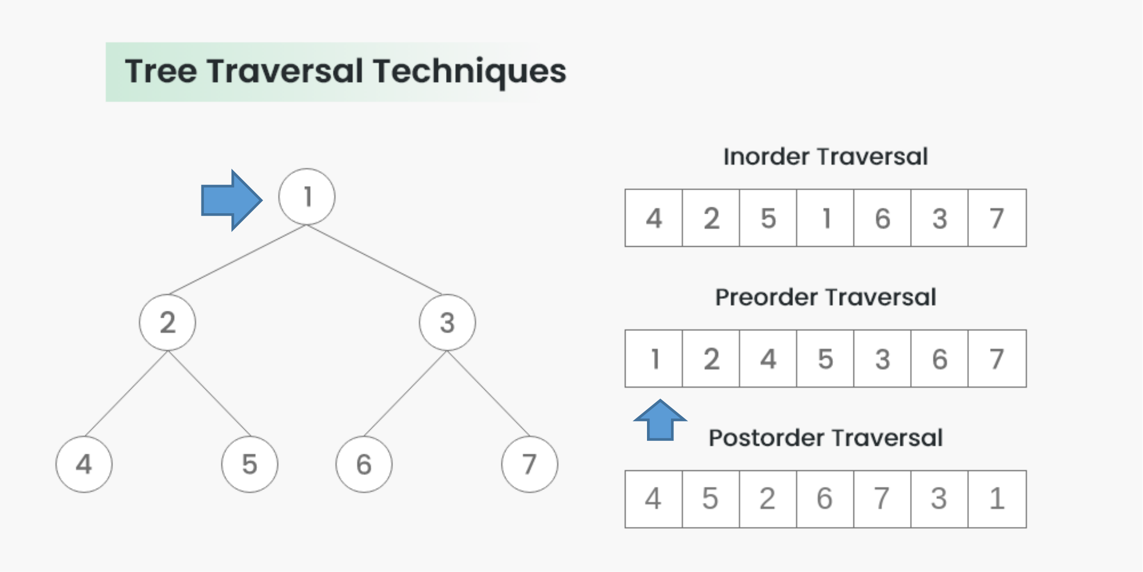
In-order Traversal




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

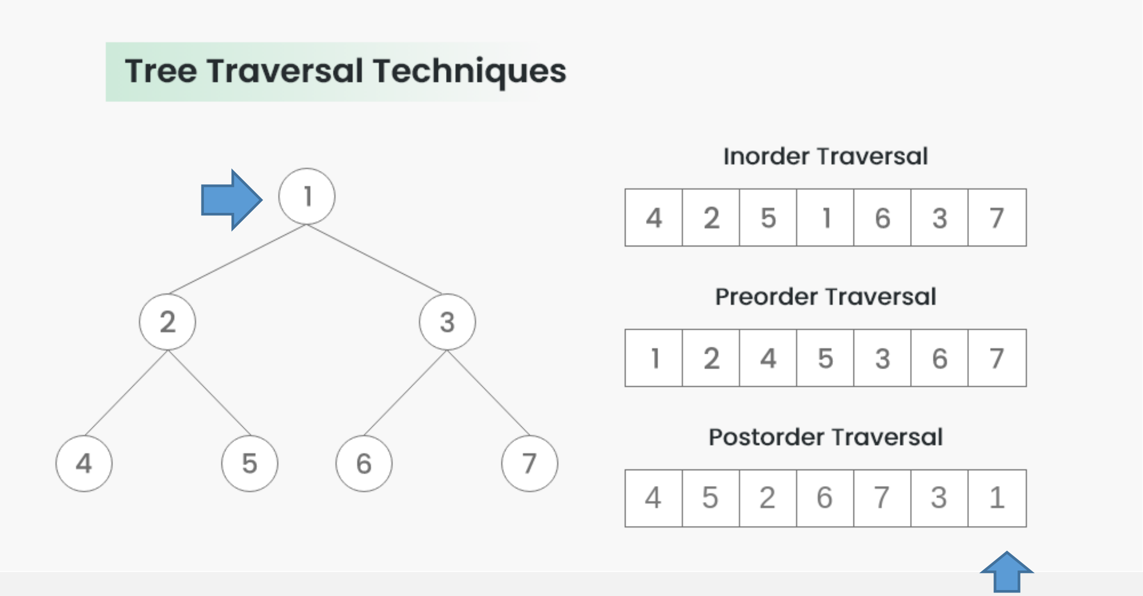
Pre-order Traversal




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

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

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
```

```
;; map : (X -> Y) List<X> -> List<Y>
;; Applies fn to each element of lst
```

```
;; tree-map : (X -> Y) Tree<X> -> Tree<Y>
;; Applies fn to each element of tree
```

```
(define (map fn lst)
  (cond
   [(empty? lst) empty]
   [else (cons (fn (first lst))
               (map (rest lst)))
```

```
;; 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-class Coding #2: tree-map

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
```

```
;; tree-map : (X -> Y) Tree<X> -> Tree<Y>
;; Applies fn to each element of tree
```

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

on gradescope

In-class Coding #2: tree-map

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
```

```
;; tree-map : (X -> Y) Tree<X> -> Tree<Y>
;; Applies fn to each element of tree
```

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

In-class Coding #2: tree-map

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
```

```
;; tree-map : (X -> Y) Tree<X> -> Tree<Y>
;; Applies fn to each element of tree
```

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

In-class Coding #2: tree-map

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
```

```
;; tree-map : (X -> Y) Tree<X> -> Tree<Y>
;; Applies fn to each element of tree
```

```
;; tree-map : Tree<X> -> ???
(define (tree-map fn t)
  (cond
   [(empty? t) empty]
   [(node? t) (node (tree-map fn (node-left t))
                    (fn (node-data t))
                    (tree-map fn (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 TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

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

Called **andmap** (for **Racket lists**) or **every** (for **JS Arrays**)

```
> (andmap positive? '(1 2 3))
#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:
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?

- Search the whole 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
```

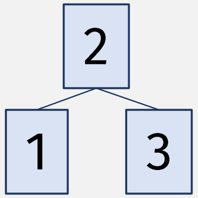
```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀x ∈left tree, x < node-data
;; Invariant 2: ∀y ∈right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

Predicate?

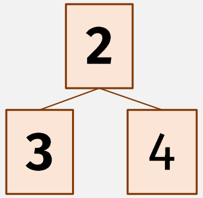
Valid BSTs

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

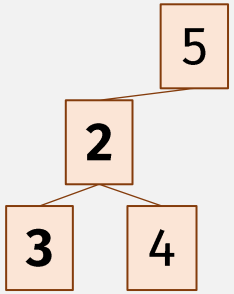
Valid



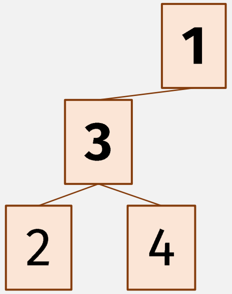
Not Valid



left value > root ☒



left values less than root ☑,
but left subtree not BST ☒



Left subtree is valid BST ☑,
but left values not less than root ☒

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))
          (valid-bst? (node-left t))
          (valid-bst? (node-right t)))]))
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

cond that evaluates to a boolean is just boolean arithmetic!

```
(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))
           (valid-bst? (node-left t))
           (valid-bst? (node-right t)))))
```

Data Definitions With Invariants

Predicate?

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

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

```
;; 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, if tree is a node:
;; Invariant 1: ∀x ∈left tree, x < node-data
;; Invariant 2: ∀y ∈right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

BST Insert

Must preserve BST invariants

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 #3: BST Insert

Hint: use valid-bst? For tests

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀x ∈left tree, x < node-data
;; Invariant 2: ∀y ∈right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

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

on gradescope

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 bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) 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 bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) 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 bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) 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 bst) x)
            (node-data bst)
            (node-right bst))
      (node (node-left bst)
            (node-data bst)
            (bst-insert (node-right bst) 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 (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) 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 (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) x)))]))
```

Result must maintain **BST invariant!**

Larger values on right

Finding a Value in a Tree?

- Do we have to search the entire tree?

Finding a Value in a BST?

```
;; 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))
```

Finding a Value in a BST?

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

```
(define (bst-has? bst x)
       ???  (empty? bst)
       ???           (node-data bst)
       ??? (bst-has? (node-left bst) x)
       ??? (bst-has? (node-right bst) x) )
```

BST (bool result) **Template**

Finding a Value in a BST?

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

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       ???              (node-data bst)
       ??? (bst-has? (node-left bst) x)
       ??? (bst-has? (node-right bst) x) )
```

BST cannot be empty

Finding a Value in a BST?

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

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
       ??? (bst-has? (node-left bst) x)
       ??? (bst-has? (node-right bst) x) )
```

Either:
- (node-data bst) is x

Finding a Value in a BST?

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

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (bst-has? (node-left bst) x)
       ??? (bst-has? (node-right bst) x) )
```

Either:
- (node-data bst) is x
- left subtree has x

Finding a Value in a BST?

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

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (bst-has? (node-left bst) x)
           (bst-has? (node-right bst) x))))
```

Either:
- (node-data bst) is x
- left subtree has x
- right subtree has x

Finding a Value in a BST?

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

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (bst-has? (node-left bst) x)
           (bst-has? (node-right bst) x))))
```

and and **or** are "short circuiting"
(stop search as soon as **x** is found)