CS 535 Object-Oriented Programming & Design
Fall Semester, 2001
Doc 13 Assignment 4 Comments

Problem 1...................................................................................................................2
Problem 2...................................................................................................................3
Class, Structs and Intelligence..................................................................................3
Information Hiding - Physical and Logical .................................................................6
Intelligence .................................................................................................................7
Once and Only Once ................................................................................................11
A Solution with Smart Node and Dumb Tree ............................................................12
  Node .........................................................................................................................12
  BinarySearchTree ....................................................................................................16
Polymorphism ............................................................................................................16
Avoid Case (and if) Statements ..................................................................................21
Solution using Polymorphism ....................................................................................22
  NilNode .....................................................................................................................22
  Node using NilNode .................................................................................................24
  BinarySearchTree with NilNode ...............................................................................26
Recursion and Performance .......................................................................................29
Issues .......................................................................................................................31
  Formatting and Structure of Code .........................................................................31
  Flags ........................................................................................................................33
  Use Guards to Simplify code ...................................................................................35
  Parallel Structure ....................................................................................................36

References

Object-Oriented Design Heuristics, Riel, Chapters 1 & 2.

Designing Object-Oriented Software, Wirfs-Brock, Wilkerson, Wiener

Smalltalk Best Practice Patterns, Kent Beck

Copyright ©, All rights reserved.
2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
Problem 1

positives

^((self select: [:each | each > 0]) size)
Problem 2
Class, Structs and Intelligence

A class is an abstraction that contains both:
• Data
• Operations

Some Heuristics

All data should be hidden within its class

Keep related data and behavior in one place

Beware of classes that have many accessor methods defined in their public interfaces. Having many implies that related data and behavior are not being kept in one place
Sample Node Class

Smalltalk.CS535 defineClass: #Node
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'key value left right '
    classInstanceVariableNames: "
    imports: "
    category: 'Course-Assignment'

key
    ^key

key: aMagnitude
    key := aMagnitude

left
    ^left

left: aNode
    left := aNode

right
    ^right

right: aNode
    right := aNode

value
    ^value

value: anObject
    value := anObject
How is the Node class different from a Struct?

A struct

• Is easier to use
• Take less time to create
Information Hiding - Physical and Logical

Physical Information Hiding

Physical information hiding is when a class has a field and there are accessor methods, `getX` and `setX`, setting and getting the value of the field. It is clear to everyone that there is a field named `X` in the class. The goal is just to prevent any direct access to `X` from the outside. The extreme example is a struct converted to a class by adding accessor methods. Physical information hiding provides little or no help in isolating the effects of changes. If the hidden field changes type than one usually ends up changing the accessor methods to reflect the change in type.

Logical Information Hiding

Logical information hiding occurs when the class represents some abstraction. This abstraction can be manipulated independent of its underlying representation. Details are being hidden from the outside world. Examples are integers and stacks. We use integers all the time without knowing any detail on their implementation. Similarly we can use the operations pop and push without knowing how the stack is implemented.
Intelligence

What the system knows
Actions that can be performed
Impact on other parts of the system and users

Evenly Distribute System Intelligence

The above Node class is struct dumb

The BinarySearchTree that uses the Node class has all the intelligence
Some Sample Intelligence for the Node Printing

Node>>printOn: aStream
   aStream
         nextPutAll: '(';
         print: left;
         print: key;
         print: right;
      nextPutAll: ')' 

This makes the node print a representation of tree structure 

This will be printed out in the workspace and in the debugger 

Very useful when writing code 

One of the first things I add to a class
Some Sample Intelligence for the Node
Instance Creation Methods with Arguments

Node class>>key: aMagnitude value: anObject
  ^super new
    setKey: aMagnitude
    setValue: anObject

Some people cluttered methods with:

    newNode := Node new.
    newNode
      key: aKey;
      value: aValue.
    currentNode left: newNode.

This can now be replaced by:

    newNode := Node key: aKey value: aValue.
    currentNode left: newNode.

Or by:

    currentNode left: (Node key: aKey value: aValue).
Some Sample Intelligence for the Node
Parent Pointer set automatically

Smalltalk.CS535 defineClass: #Node
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'key value left right parent
  classInstanceVariableNames: "
  imports: "
  category: 'Course-Assignment'

left: aNode
  left := aNode.
  aNode parent: self

right: aNode
  right := aNode
  aNode parent: self

Now one can forget about setting parent pointers in nodes

One can replace:

currentNode left: aNode.
aNode parent: currentNode

With

currentNode left: aNode.
Once and Only Once

In a program written with good style, everything is aid once and only once

Kent Beck
A Solution with Smart Node and Dumb Tree Node

Smalltalk.CS535 defineClass: #Node
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'key value left right '
  classInstanceVariableNames: "
  imports: "
  category: 'Course-Assignment'

Instance Methods defined

<table>
<thead>
<tr>
<th>Category</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessing</td>
<td>at:</td>
</tr>
<tr>
<td></td>
<td>at:put:</td>
</tr>
<tr>
<td>enumeration</td>
<td>keysAndValuesDo:</td>
</tr>
<tr>
<td>initialize</td>
<td>setKey:setValue:</td>
</tr>
<tr>
<td>printing</td>
<td>printOn:</td>
</tr>
<tr>
<td>private</td>
<td>keyNotFoundError:</td>
</tr>
</tbody>
</table>

CS535.Node class methodsFor: 'instance creation'

key: aKey value: anObject
  ^super new setKey: aKey setValue: anObject

CS535.Node class methodsFor: 'constants'

keyNotFoundErrorException
  ^KeyedCollection keyNotFoundSignal
CS535. Node methodsFor: 'initialize'

setKey: aKey setValue: anObject
  key := aKey.
  value := anObject

CS535. Node methodsFor: 'accessing'

at: aKey
  aKey = key ifTrue: [^value].
  aKey < key
    ifTrue:
      [left isNil ifTrue: [self keyNotFoundError: aKey].
       ^left at: aKey].
  aKey > key
    ifTrue:
      [right isNil ifTrue: [self keyNotFoundError: aKey].
       ^right at: aKey].
at: aKey put: anObject
   aKey = key ifTrue: [^value := anObject].
aKey < key
   ifTrue:
      [left isNil
       ifTrue:
          [left := self class key: aKey value: anObject.
          ^anObject].
          ^left at: aKey put: anObject].
aKey > key
   ifTrue:
      [right isNil
       ifTrue:
          [right := self class key: aKey value: anObject.
          ^anObject].
          ^right at: aKey put: anObject].

CS535.Node methodsFor: 'enumeration'

keysAndValuesDo: aBlock
   "Block has two parameters - key then value"

left notNil ifTrue:[left keysAndValuesDo: aBlock].
aBlock value: key value: value.
right notNil ifTrue: [right keysAndValuesDo: aBlock]
CS535. Node methodsFor: 'printing'

printOn: aStream
    aStream
        nextPut: $;
        print: left;
        print: key;
        print: right;
        nextPut: $)

CS535. Node methodsFor: 'private'

keyNotFoundError: missingKey
    "Raise a signal indicating that the key was not found."

    ^self class keyNotFoundErrorException raiseWith: missingKey
BinarySearchTree

Smalltalk.CS535 defineClass: #BinarySearchTree
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'root '
    classInstanceVariableNames: "
    imports: "
    category: 'Course-Assignment'

CS535.BinarySearchTree class methodsFor: 'constants'

keyNotFoundException
    ^KeyedCollection keyNotFoundSignal

CS535.BinarySearchTree class methodsFor: 'instance creation'

keys: keyCollection values: objectCollection
    | tree |
    tree := super new.
    tree
        addKeys: keyCollection
        withValues: objectCollection.
    ^tree
CS535. BinarySearchTree methodsFor: 'accessing'

addKeys: keyCollection withValues: objectCollection
  keyCollection
    with: objectCollection
    do: [:key :value | self at: key put: value]
  at: aKey
    root isNil ifTrue: [self keyNotFoundError: aKey].
    ^root at: aKey

at: aKey put: anObject
  aKey isNil ifTrue: [^self subscriptBoundsError: aKey].
  root isNil
    ifTrue:
      [root := Node key: aKey value: anObject.
       ^anObject].
    ^root at: aKey put: anObject

size
  | nodeCount |
  nodeCount := 0.
  self do: [:each | nodeCount := nodeCount + 1].
  ^nodeCount.
CS535.BinarySearchTree methodsFor: 'enumeration'

detect: aBlock
  ^self detect: aBlock ifNone: [self notFoundError]

detect: aBlock ifNone: exceptionBlock
  self do: [:each | (aBlock value: each) ifTrue: [^each]].
  ^exceptionBlock value

do: aBlock
  "Block has one parameter - value of each node"
  root isNil ifTrue:[^nil].
  root keysAndValuesDo: [:key :value | aBlock value: value]

CS535.BinarySearchTree methodsFor: 'printing'

printOn: aStream
  aStream
    nextPutAll: 'BST(';
    print: root;
    nextPut: $)

CS535.BinarySearchTree methodsFor: 'private'

keyNotFoundError: missingKey
  "Raise a signal indicating that the key was not found."
  ^self class keyNotFoundException raiseWith: missingKey

notFoundError
  "Raise a signal indicating that an object is not in the collection."
  ^self class notFoundSignal raise
Checking For nil nodes

How many times do you check for a nil node?

I do it 9 times!

Why do we have always check?
Polymorphism

Ability of two or more classes of object to respond to the same message

Objects of different classes can respond to the same message in a way that is appropriate to them
Modular Design Rule

Avoid Case (and if) Statements

The same if (and case) statements tend to appear in many places

This make it hard to change the if (case) statement

Duplicating the if (case) statement takes time

One way to avoid case & if statements is to use polymorphism
Solution using Polymorphism
NilNode

Replaces nil to represent an empty subtree

Smalltalk.CS535 defineClass: #NilNode
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'parent'
   classInstanceVariableNames: "
   imports: "
   category: 'Course-Assignment'

CS535.NilNode class methodsFor: 'constants'

keyNotFoundException
   ^KeyedCollection keyNotFoundSignal

CS535.NilNode class methodsFor: 'instance creation'

new
   self error: 'Use parent: to create an instance of ', self name

parent: aNodeOrTree
   ^super new setParent: aNodeOrTree
NilNode Instance Methods

at: aKey
    self keyNotFoundError: aKey

at: aKey put: anObject
    parent
        replaceNode: self
        with: (Node key: aKey value: anObject).
        ^anObject

keysAndValuesDo: aBlock
    "Block has two parameters - key then value"

setParent: aNodeOrTree
    parent := aNodeOrTree

printOn: aStream

keyNotFoundError: missingKey
    "Raise a signal indicating that the key was not found."

    ^self class keyNotFoundErrorException raiseWith: missingKey
Node using NilNode

Smalltalk.CS535 defineClass: #Node
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'key value left right '
   classInstanceVariableNames: ''
   imports: ''
   category: 'Course-Assignment'

CS535.Node class methodsFor: 'instance creation'

key: aKey value: anObject
   ^super new setKey: aKey setValue: anObject

CS535.Node methodsFor: 'initialize'

setKey: aKey setValue: anObject
   key := aKey.
   value := anObject.
   left := NilNode parent: self.
   right := NilNode parent: self.
Node's Other Instance Methods

at: aKey
  aKey = key ifTrue: [^value].
  aKey < key ifTrue: [^left at: aKey].
  aKey > key ifTrue: [^right at: aKey].

at: aKey put: anObject
  aKey = key ifTrue: [^value := anObject].
  aKey < key ifTrue: [^left at: aKey put: anObject].
  aKey > key ifTrue: [^right at: aKey put: anObject].

keysAndValuesDo: aBlock
  "Block has two parameters - key then value"

    left keysAndValuesDo: aBlock.
    aBlock value: key value: value.
    right keysAndValuesDo: aBlock

printOn: aStream
  aStream
    nextPut: $(
    print: left;
    print: key;
    print: right;
    nextPut: $)

replaceNode: existingNode with: newNode
  existingNode == left ifTrue:[left := newNode].
  existingNode == right ifTrue:[right := newNode].
BinarySearchTree with NilNode

Smalltalk.CS535 defineClass: #BinarySearchTree
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'root '
  classInstanceVariableNames: "
  imports: "
  category: 'Course-Assignment'

CS535.BinarySearchTree class methodsFor: 'instance creation'

keys: keyCollection values: objectCollection
  | tree |
  tree := self new.
  tree
    addKeys: keyCollection
    withValues: objectCollection.
  ^tree

new
  ^super new initialize
BinarySearchTree Instance Methods

addKeys: keyCollection withValues: objectCollection
  keyCollection
  with: objectCollection
  do: [:key :value | self at: key put: value]

at: aKey
  ^root at: aKey

at: aKey put: anObject
  aKey isNil ifTrue:[^self subscriptBoundsError: aKey].
  ^root at: aKey put: anObject

size
  | nodeCount |
  nodeCount := 0.
  self do: [:each | nodeCount := nodeCount + 1].
  ^nodeCount.

notFoundError
  "Raise a signal indicating that an object is not in the collection."

    ^self class notFoundSignal raise

replaceNode: existingNode with: newNode
  root := newNode

detect: aBlock
  ^self detect: aBlock ifNone: [self notFoundError]
detect: aBlock ifNone: exceptionBlock
    self do: [:each | (aBlock value: each) ifTrue: [^each]].
    ^exceptionBlock value

do: aBlock
    "Block has one parameter - value of each node"

    root keysAndValuesDo: [:key :value | aBlock value: value]

printOn: aStream
    aStream
        nextPutAll: 'BST(';
        print: root;
        nextPut: $)

initialize
    root := NilNode parent: self
Recursion and Performance

Searching an unbalanced binary search tree recursively is dangerous

- You can run out of stack space

Use iteration to search an unbalanced BST

Recursion is OK on a balance tree

Exercise:

  Modify the Node and NilNode to work in an iterative search
NilNode and Performance

A BST with N keys requires N+1 NilNodes in the above implementation.

It is possible to use only one NilNode per tree.

In timing tests in Java using the NilNode comparable to the first solution.
Issues
Formatting and Structure of Code

Indenting your code to make it readable is important

Indentation should show the structure of your code

Some bad Examples

(currentNode isNil)
 ifTrue: [^nil]
 ifFalse: [^currentNode value]

(currentNode isNil) ifTrue: [^nil]
 ifFalse: [^currentNode value]

root isNil
 ifTrue:[ root := blah]
 ifFalse: [more blah]

In the future you will lose even more points for bad indentation
Smalltalk Standard

The template you see when creating a method is to be followed

message selector and argument names
"comment stating purpose of message"

| temporary variable names |
statements

Smalltalk can format the Code for you

In the bottom pane of the browser click the right mouse button

In the pop up menu select the format item

It will format the code currently in the code pane
Flags

Avoid flags - they make code hard to read

If need flags use a good name

found what?

insert: aKey data: aValue
| treeNode found saveNode |
found := false.
treeNode := root.
[treeNode ~= nil and: [found = false]] whileTrue:
[saveNode := treeNode.
(aKey = treeNode key)
ifTrue: [found := true]
ifFalse:
[(aKey < treeNode key)
ifTrue: [treeNode := treeNode leftChild]
ifFalse: [treeNode := treeNode rightChild].
].
].
(found = false)
ifTrue:
[treeNode := Node key: aKey data: aValue.
size := size + 1.
(aKey < saveNode key)
ifTrue: [saveNode leftChild: treeNode]
ifFalse: [saveNode rightChild: treeNode].
].
**Code without Flag**

```plaintext
insert: aKey data: aValue
    | treeNode saveNode |
    treeNode := root.
    [treeNode notNil] whileTrue:
        [saveNode := treeNode. 
         (aKey = treeNode key) ifTrue: [^nil]. 
         (aKey < treeNode key) 
             ifTrue: [treeNode := treeNode leftChild] 
             ifFalse: [treeNode := treeNode rightChild]. 
        ].
    treeNode := Node key: aKey data: aValue.
    size := size + 1.
    (aKey < saveNode key)
        ifTrue: [saveNode leftChild: treeNode]
        ifFalse: [saveNode rightChild: treeNode].
```

**Code With Some Methods**

```plaintext
insert: aKey data: aValue
    | child parent |
    child := root.
    parent := child.
    [child notNil] whileTrue:
        [(aKey = child key) ifTrue: [^nil]. 
         parent := child. 
         child := parent nextNodeFor: aKey].
    ].
    parent addToSelf: (Node key: aKey data: aValue).
```
Use Guards to Simplify code

checkFrom: aNode toFind: aKey
    aNode isNil
        ifFalse:
            [aNode key = aKey
                ifTrue: [^aNode value]
                ifFalse:
                    aKey < aNode key
                        ifTrue: [^self checkFrom: aNode leftChild toFind: aKey]
                        ifTrue: [^self checkFrom: aNode rightChild toFind: aKey]]
            ifTrue: [^nil]

With Guards

checkFrom: aNode toFind: aKey
    aNode isNil ifTrue: [^nil].
    aNode key = aKey ifTrue: [^aNode value]
    aKey < aNode key
        ifTrue: [^self checkFrom: aNode leftChild toFind: aKey]
        ifTrue: [^self checkFrom: aNode rightChild toFind: aKey]]
Parallel Structure

Smalltalk.CS535 defineClass: #BinarySearchTree
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'root nodesInOrderedCollection'
   classInstanceVariableNames: "
   imports: "
   category: 'Course-Assignment'

Parallel structures are hard to maintain in sync

Just use one