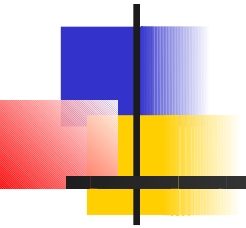
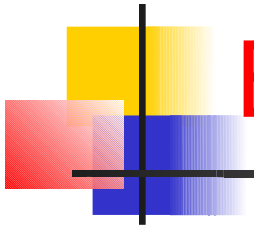


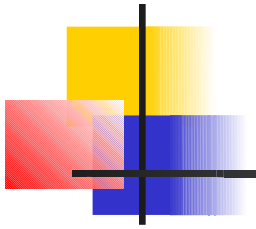
Mutable Trees

Corky Cartwright
Stephen Wong
Department of Computer Science
Rice University



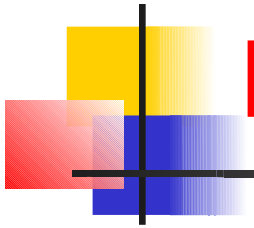
Background

- Functional binary trees
  - Scheme: ancestor trees, binary trees, binary search trees, ...
  - Java equivalents: obvious image of Scheme notion using the composite pattern to represent binary trees. Perhaps we should have assigned some exercises. No surprises.
- Other functional trees
  - Scheme: descendant trees with arbitrary number of children, abstract syntax trees (ASTs) where the children signature depends on the node type, …
  - Java equivalents: … notably ASTs for boolean formulas
- Focus on binary trees rather than more general forms of trees (with varying number of children at internal nodes) because they have regular type signatures and they are ubiquitous. Binary are used far more often than any other specific form of tree.



Tree Processing is Traversal

- Tree-walking: in processing trees we visit the nodes in some order. There are three established orders that are named based on when the root of a tree is visited.
  - Pre-order: the root is visited before the children
  - Post-order: the root is visited after the children
  - In-order (applicable only to binary trees) : the root is visited after the left tree but before the right tree.
  - `tree-map` from Exam 1 performed a post-order traversal.
- Conventional tree-walking is relatively uncommon in code written in an OO style because it fixes the calling structure (like `tree-map`). Visitors, which let each specific visitor specify the traversal (recursive calling) strategy.

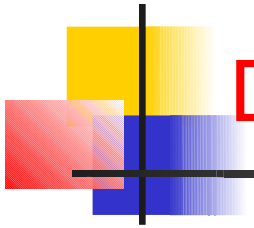


Binary (Search) Trees

- In Java, functional binary (search) trees are internally represented using objects (nodes) equipped with **`key`**, **`value`**, **`left`** and **`right`** fields (in some cases additional data fields) and degenerate leaves that are typically **`null`**. (Other degenerate leaf values such as an explicit **`EMPTY`** object are possible, but **`null`** (regrettably) is dominant. Recall that **`null`** is not an OO data representation.
- Mutable trees have exactly the same fields in internal (non-leaf) nodes except that they are mutable.
- In building a mutable binary (search) tree, new nodes can be added simply by replacing a degenerate leaf by an internal tree node. The asymptotic cost does not change (assuming you are starting at the root) but the constant factor is reduced.
- Example: **`TreeMap`** in DrJava

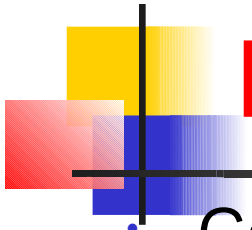
Coding Mutable Tree Data Types

- General Observation: treating boundary conditions (`null` references) correctly is critical; they are a frequent source of errors.
- In an OO language, all ugliness should be encapsulated inside the class representing a data structure. The internal representation of a type (*e.g.*, the `Node` class for a binary tree) should not be visible to clients of the data type.
- Deleting nodes from a binary search tree, no matter what the formulation, is rather ugly, particularly if we are seeking a minimum cost solution. See Cormen, Leiserson, Rivest (referenced in the course wiki) for code snippets (that are often inscrutable) along these lines.
- Trees naturally represent both ordered sets and maps on an ordered set of keys. But these two abstract data structures have incompatible interfaces. Hence, a collections library needs both a `TreeSet` and a `TreeMap`.



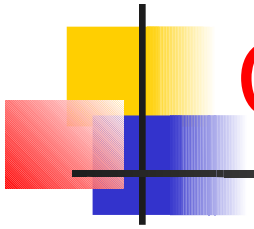
Deleting Nodes from a Binary Search Tree

- This operation is messy in all formulations of binary search trees, particularly if some notion of optimal code is sought.
- There is an intelligible strategy with good performance that relies on a cute trick. Deleting the minimum element of a tree is straightforward because the node containing the minimum element has no left subtree. Deletion of a node with a non-empty right subtree (the empty right subtree case is easy) can be reduced to finding the minimum node in the right subtree, hoisting its data into the node being "deleted", and deleting the hoisted node. Easy in OO approach. We will examine this code in **`OOTreeMap`** in DrJava shortly.



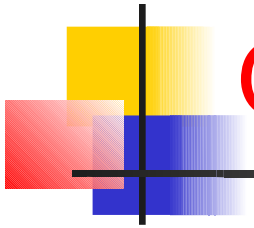
Mutable Binary Search Trees

- Go to Drjava code and study **`TreeMap.java`**, which is a traditional procedural solution encapsulated in OO classes. The code is inherently messy with many cases and complex control.
- Can we produce a more intelligible solution using OO ideas?



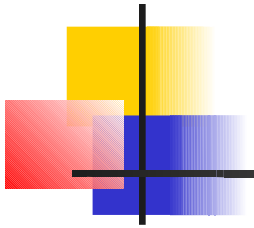
Controversial Issue in OO Design

- How much should we compromise good object-oriented style to achieve high performance or to incorporate conventional procedural data structure implementations?
- Fact: most Java code, even Java libraries, is not written in a sophisticated OO style. Why? OO Design is only understood by a (growing) minority of software developers.



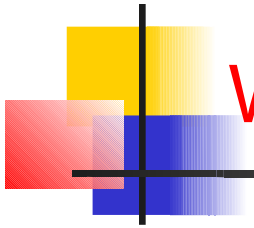
OO Mutable Binary Trees

- Code for conventional procedural solution has been repeatedly polished but it is still astonishingly convoluted (albeit efficient).
- A standard procedural solution can be hidden inside an OO class, *e.g*. the **`TreeMap`** class posted with this lecture. Please read it. It is reasonably clean in comparison to most procedural solutions but it is ugly because it relies on complex looping and conditional control and raw mutable references that make mutation difficult.
- In contrast, the lean OO solution (derived from a more overt OO solution developed by Nguyen and Wong) is much lighter weight and easier to understand although there are occasionally subtle distinctions between a cell (**`RefNode`**) containing a **`Node`** and a **`Node`**. Fortunately, static type-checking catches nearly all bugs where code might confuse the two.



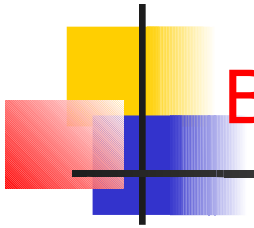
Why Is the OO Code Simpler?

- The OO code uses a lightweight version of the *state* pattern. The left and right subtrees in tree nodes are not other tree nodes. Instead they are state objects that can be either empty on non-empty. Null references cannot be cleanly used instead of empty state objects because they are **not** objects. (Note: in languages where fields can be passed by reference like C++, the field containing a null value can be treated as an object.)
- Since the left and right fields of a node are **always** mutable objects, they can be modified as needed. As a result, OO code never has to focus on the parents of nodes to be deleted or inserted. The deletion or insertion *can be performed directly* on the non-empty node to be deleted or the empty node to be replaced.
- Please compare the code in **`TreeMap.java`** (a procedural solution encapsulated as a class) and **`OOTreeMap.java`** which is the lightest weight OO solution that we know how to construct.
- Let's look at the code in **`OOTreeMap.java`**



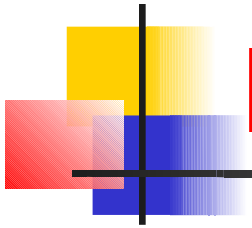
Why Is Well-Written Procedural Code Faster?

- The OO code uses an extra level of indirection to eliminate the need for ugly special cases. A state object is a container that can hold different variants in a union or composite pattern. (Think of state as a mutable field bound to the possible variants of a union or composite type.)
- In this case, we have a simple composite that is either empty or non-empty (much like functional Lists) where the non-empty object contains a key value and two state objects representing left and right subtrees. To minimize the cost of the state pattern, we represent the the empty state by a null reference rather than a pointer to an `EMPTY` object.



Binary Search Tree Implementations Compared

- Go to the respective code bases using DrJava.



For Next Class

- Laundry homework due on Wednesday. Assignment specs are much longer than the code you must write. Straightforward but not conducive to last-minute solution. Play with it. Have fun. There is nothing conceptually hard about the data or algorithms in this assignment. It is an exercise to help you learn about programming with mutable data structures in Java.
- The supporting code base is formulated as a DrJava project with named packages.
- DrJava makes it easy to practice writing code fragments/exercises. Do it! Don't be afraid to experiment. The interactions pane makes it easy.