Designing OO Data Structures

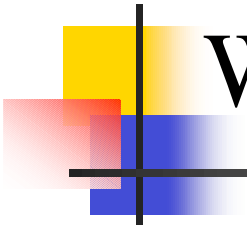
Corky Cartwright
Department of Computer Science
Rice University

Controversial Issue

- 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?
  - Most Java programmers (including recent Comp Sci graduates) don't know enough OO design concepts to write sophisticated OO code.
  - Expressing code in a form where dynamic dispatch is the principal control mechanism often adds some (constant factor) overhead. In fact, the overhead can be substantial if the OO implementation is not written in a *lean* style. In a lean OO style, we try to minimize the creation of extra container (wrapper objects) which are critical to simplified OO control structure.
  - *None* of the major reference books on Data Structures and Algorithms write code in a sophisticated OO style.

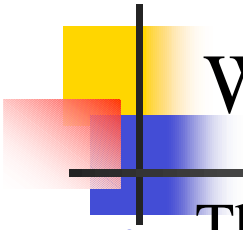
Case Study: Mutable Binary Trees

- Core topic in basic course on data structures & algorithms.
- Code for conventional procedural solution has been repeatedly polished but it is still astonishingly ugly and convoluted (albeit efficient).
- Standard procedural solutions 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 relies on complex looping and conditional (flag passing and testing) control.
- In contrast, the lean OO solution (derived from a heavier, more typical OO solution proposed by Nguyen and Wong) is much lighter weight and easier to understand although there are frequent subtle distinctions between a cell (RefNode) containing (a reference to) a Node and (a reference to) a Node. Fortunately, static type-checking catches nearly all bugs where code confuses the two.



Why Is the OO Code Simpler?

- The OO code uses a very lightweight version of the state pattern. The left and right subtrees in tree nodes are not references to other tree nodes. Instead they are state objects that can be either empty on non-empty. Conventional null references cannot be cleanly used instead emtpy state objects because they are NOT objects; this fact causes ENORMOUS grief. (Note: in languages where fields can be passed by reference, the field containing a null value can be treated as an object. (C++ [in recent incarnations] is such a language. Passing pointer fields by reference may be a good strategy for a heroic C++ programmer.)
- Since the left and right fields of a node are ALWAYS mutable state objects (notably when they are in the empty state), they can be modified as needed.. As a result, the 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 replace by an inserted node.
- **Please compare the code in TreeMap.java (a conventional procedural solution encapsulated as a class and OOTreeMap.java which is the lightest weight OO solution that we know how to construct. We strongly prefer it to the procedural solution unless performance constraints prevent its use.**

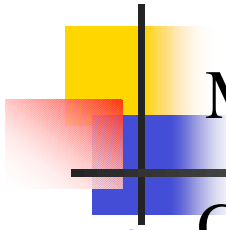


Why Is Well-Written Procedural Code Faster?

- The OO code in effect 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 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 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 referdnce rather than a pointer to an empty object EMPTY.

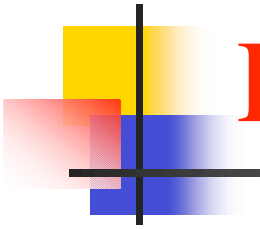
Why Is Well-Written Procedural Code Faster?

- If EMPTY can be represented as a singleton, there is little advantage to using null, but Java's formulation of generic typing forecloses the type safe use of singletons. In this example, we need generic typing to get accurate static type checking so we use null (wrapped inside a state object) to represent empty trees.
- It would be illuminating to compare the performance of the two implementations on some large tests. I expect the OO code to perform well (within a factor of 2) of the procedural code, but only testing can determine whether the OO code is as efficient as my intuition leads me to believe.



Mutable Binary Search Tree Implementations Compared

- Go to Drjava code



For Next Class

- Laundry homework due 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 get up to speed with mutable data structures in Java.
- Two forms for supporting code base:
  - Class per file (prepares you for last two assignments)
  - All classes in one file (easier)
- DrJava makes it easy to practice writing code fragments/exercises. Do it! Don't be afraid to experiment. The interactions pane makes it easy.