

CS3500: Object-Oriented Design

Spring 2014

Class 15
2.28.2014

Today...

- Assignments
- Red-Black Trees
- Refactoring
- hashCode Improvement
- Visitor pattern

Assignment 7

- Red-Black Tree implementation for $\text{MyMap}\langle K, V \rangle$
- Due: Friday, February 28, 2014 at 11:59pm

ICE

MyMap<String, Integer> with StringByLex

- <Jack, 10>
- <Ellie, 5>
- <Dan, 4>
- <Ben, 2>
- <Carol, 3>
- <Oliver, 15>
- <Tim, 20>
- <Yvette, 25>
- <Victor, 22>

Red-Black Trees






[Okasaki]

Red-black trees are binary search trees in which each node has a color (either red or black) and the following balancing invariants are preserved by all operations:

1. No red node has a red child.
2. Every path from the root to an empty tree/node contains the same number of black nodes.

Binary Search Tree (BST)

data Tree elt

- **t is empty**  = E | T Color (Tree elt) elt (Tree elt)
- **t is a node** 
 - a label 
 - the left subtree of t is a BST, 
 - the right subtree of t is a BST, 
 - every label within the left subtree of t is less than the label of t,
 - every label within the right subtree of t is greater than the label of t

Insertions

[Okasaki]

```
insert :: Ord elt => elt -> Set elt ->
    Set elt
insert x s = makeBlack (ins s)

where ins E = T R E x E

ins (T color a y b) | x < y  = balance color (ins a) y b
                    | x == y = T color a y b
                    | x > y  = balance color a y (ins b)

makeBlack (T _ a y b) = T B a y b
```

Balance

[Okasaki]

balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)

balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)

balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)

balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)

balance color a x b = T color a x b

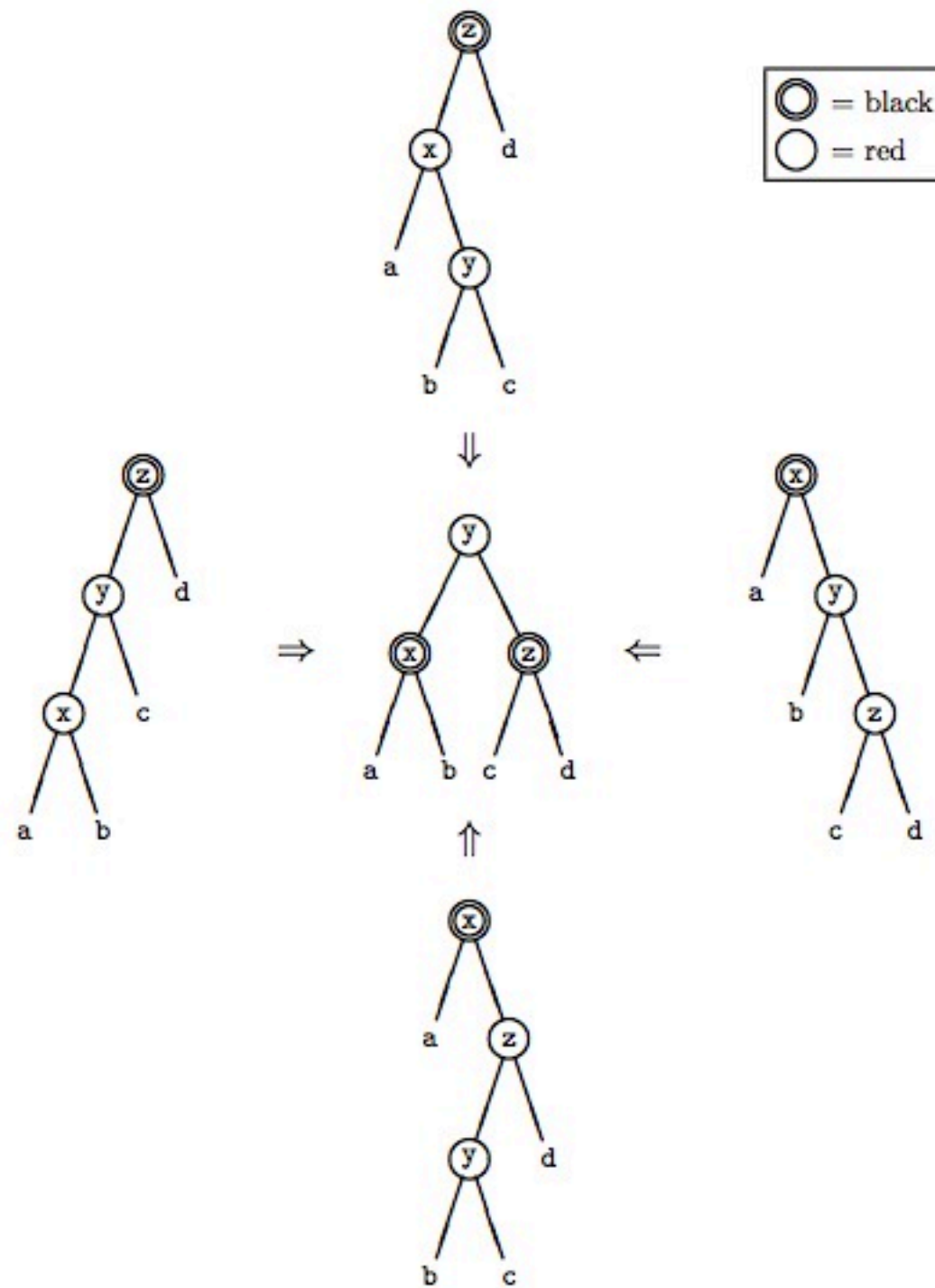
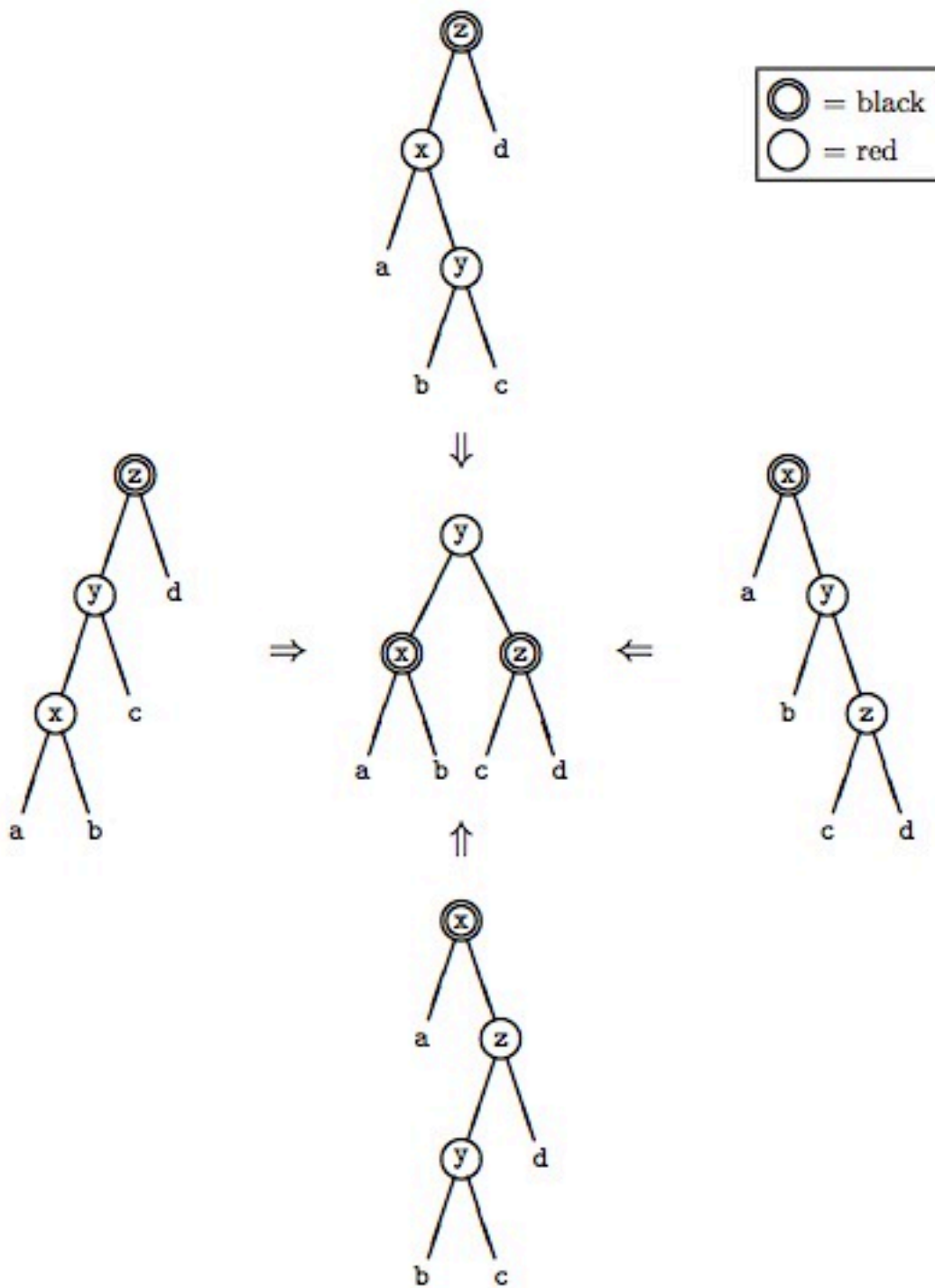


Figure from [Okasaki]



balance B (T R (T R a x b) y c) z d
 = T R (T B a x b) y (T B c z d)

balance B (T R a x (T R b y c)) z d
 = T R (T B a x b) y (T B c z d)

balance B a x (T R (T R b y c) z d)
 = T R (T B a x b) y (T B c z d)

balance B a x (T R b y (T R c z d))
 = T R (T B a x b) y (T B c z d)

balance color a x b = T color a x b

from [Okasaki]

Refactoring

- Privatization of members
- Nested classes
- Merge subclass with base class
- Using the Singleton pattern
- Using null as a legitimate value
- Inlining

Privatization of members

Privatization of members

- Hide members from clients so that they can't make mistakes when using them.
- We can change implementation without changing client code.

Nested classes

Nested classes

- Logical grouping of classes
- Increased encapsulation
- More readable, maintainable code

Merge subclass with base class

Merge subclass with base class

- A superclass and subclass are not very different.
- A class isn't doing very much.
- Not always a good approach

Using the Singleton pattern

Using the Singleton pattern

- When all of a class's instances are behaviorally equivalent, then we might as well use the Singleton pattern to create only one instance of that class.

Using `null` as a legitimate value

Using `null` as a legitimate value

- Usually a **bad** idea
- Many checks for null
- Lose the idea of the class having multiple types
- Makes programs harder to read
- May be harder to interpret
- More difficult if add subclasses later
- Lose dynamic dispatch

Inlining

Inlining

- Usually a **bad** idea
- You end up with a lot more in your basic methods
- You lose the dynamic methods
- Difficult for change
- Can be hard to interpret—duplicate calls—may miss one if making changes

Refactoring

Software Qualities

Aspects of software quality

[Lewis & Chase]

- **Correctness:** The degree to which software adheres to its specific requirements.
- **Reliability:** The frequency and criticality of software failure
- **Robustness:** The degree to which erroneous situations are handled gracefully.
- **Usability:** The ease with which users can learn and execute tasks within the software.
- **Maintainability:** The ease with which changes can be made to the software.
- **Reusability:** The ease with which software components can be reused in the development of other software system.
- **Portability:** The ease with which software components can be used in multiple computer environments.
- **Efficiency:** The degree to which the software fulfills its purpose without wasting resources.

Code Quality

Good

- shorter
- easier to read
- elegant
- clean
- modular

Bad

- longer
- harder to read
- less elegant
- tangled
- less modular (different levels of abstraction mixed up in code)

Reliability Testing

Computing better hash codes

hashCode from Object

<http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html>

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by `java.util.Hashtable`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns: a hash code value for this object.

Definitions

- **If** `f1.hashCode() == f2.hashCode()` **but** `f1` and `f2` are not equal, then the hash codes for `f1` and `f2` are said to *collide*.
- The *collision probability* is the conditional probability that `f1.hashCode() == f2.hashCode()` given that `f1` and `f2` are not equal.

Assignment 8

- Due: Friday, March 14, 2014 at 11:59pm
- Visitor pattern

Visitor Pattern

Visitor Pattern

(From <http://www.ccs.neu.edu/course/cs2510h/dpc.pdf>. Version: April 8, 2013)

- “The visitor pattern is a general design pattern that allows you to separate data from functionality in an object-oriented style.”
- “[I]t requires you to implement one method which accepts what we call a ‘visitor’ that is then exposed to a view of the data.”

Visitor Pattern

[Gamma et al.]

“Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”

When to Use Visitor Pattern

[Gamma et al.]

- Many classes of objects
- Distinct and unrelated operations to perform on object
- Object structures rarely change but add new operations often

Benefits of Using Visitor Pattern

[Gamma et al.]

- Adding new operations is easy
- Gathers related operations and separate unrelated ones
- Accumulating state