CMSC 132:
OBJECT-ORIENTED PROGRAMMING II

Polymorphic Lists & Trees

Department of Computer Science
University of Maryland, College Park
Polymorphic Binary Search Trees

• Second approach to implement BST
• What do we mean by polymorphic?
• Implement two subtypes of Tree
  • EmptyTree
  • NonEmptyTree
• Use EmptyTree to represent the empty tree
  • Rather than null
• Invoke methods on tree nodes
  • Without checking for null (IMPORTANT!)
Polyomorphc Binary Tree Implementation

Interface Tree {
    Tree insert ( Value data1 ) { … } } 

Class EmptyTree implements Tree {
    Tree insert ( Value data1 ) { … } 
}

Class NonEmptyTree implements Tree {
    Value data;
    Tree left, right; // Either Empty or NonEmpty
    Tree insert ( Value data1 ) { … } 
}
Standard vs. Polymorphic Binary Tree

Class Node {
    Node left, right;
}

Class EmptyTree {}

Class NonEmptyTree {
    Tree left, right;
}

Node X {
    left = Y;
    right = Z;
}

Node Y {
    left = null;
    right = null;
}

Node Z {
    left = null;
    right = W;
}

Node W {
    left = null;
    right = null;
}

NonEmptyTree X {
    left = Y;
    right = Z;
}

NonEmptyTree Y {
    left = ET;
    right = ET;
}

NonEmptyTree Z {
    left = ET;
    right = W;
}

NonEmptyTree W {
    left = ET;
    right = ET;
}

EmptyTree {}
Singleton Design Pattern

• Definition
  • One instance of a class or value accessible globally

• Where to use & benefits
  • Ensure unique instance by defining class final
  • Access to the instance only via methods provided

• EmptyTree class will be a singleton class
Singleton Example

public final class MySingleton {
    // declare the unique instance of the class
    private static MySingleton uniq = new MySingleton();
    // private constructor only accessed from this class
    private MySingleton() { … }
    // return reference to unique instance of class
    public static MySingleton getInstance() {
        return uniq;
    }
}
Using Singleton EmptyTree

Class Node {
    Node left, right;
}

Class EmptyTree {}

Class NonEmptyTree {
    Tree left, right;
}

Node X {
    left = Y;
    right = Z;
}

Node Y {
    left = null;
    right = null;
}

Node Z {
    left = null;
    right = W;
}

Node W {
    left = null;
    right = null;
}

NonEmptyTree X {
    left = Y;
    right = Z;
}

NonEmptyTree Y {
    left = ET;
    right = ET;
}

NonEmptyTree Z {
    left = ET;
    right = W;
}

NonEmptyTree W {
    left = ET;
    right = ET;
}

EmptyTree ET {}
BST– Deletion (PseudoCode for project)

• Algorithm
  1. Perform search for value $X$
  2. If $X$ is a leaf, delete $X$
  3. Else // must delete internal node
      a) Replace with largest value $Y$ on left subtree
         OR smallest value $Z$ on right subtree
      b) Delete replacement value (Y or Z) from subtree

• Observation
  • $O(\log(n))$ operation for balanced tree
  • Deletions may unbalance tree
Polymorphic List Implementation

• Let’s see a polymorphic list implementation
• See code distribution: LecturePolymorphicListCode.zip