CS2110 Assignment 3 — Inheritance and Trees, Summer 2008
Due Sunday July 13, 2008, 6:00PM

0 Introduction

0.1 Goals

This assignment will help you get comfortable with basic tree operations and algorithms. It is also meant to give you a chance to work with inheritance.

0.2 Submission

Follow the Submission Requirements on the course website. The last section of this assignment summarizes the files that you need to submit on CMS.

0.3 Grading

All code that you submit must run without warnings or errors. Otherwise, you will receive a zero for that portion of the assignment. The assignment will be graded out of 100 points and is worth 10% of your course grade.

1 Short Answers

Purpose: Object-oriented programming concepts, binary trees, asymptotic analysis
Points: 18

1. What is the maximum possible number of nodes in a binary tree of depth k?
   What is the minimum possible number of nodes in a tree of depth k?
   Prove your answers by induction.
2. Insert in the Binary Search Tree elements in the following order: `S E A R C H` .
   What is the sequence of characters generated by an in-order, a pre-order and a post-order traversal?
3. What is the minimum number of nodes in AVL Tree of depth 11? See Section 4 for an explanation of AVL Trees.
4. Suppose we have a class `LivingThing` that is extended by the class `Vertebrate`. Now we instantiate an object from each of the two classes; let them be `l` and `v` respectively. Typically, which of the two objects will have the larger memory footprint? Which class is a subtype of the other?
5. What is the problem with the following code segment?

```
abstract class LivingThing  {
    abstract private double area();
    abstract private double draw();
    ...
}
```

6. Place the following functions in order of increasing growth rate: n, $\sqrt{n}$, n^2, $n \log n$, $n \log^2 n$,$n \log n^2$,$2/n$, 2^n, 42, $n^2 \log n$, n^{30}, n^n, $n!$. Indicate any ties (i.e. functions that grow at the same rate).

Submit your answers in a plain text file called `problem1.txt`. Follow the mathematical notation defined in Assignment 2 for the common operators. If we cannot open the file with a text editor, you will receive no credit. If you do not know what plain text is, refer to the submission format requirements on the website.

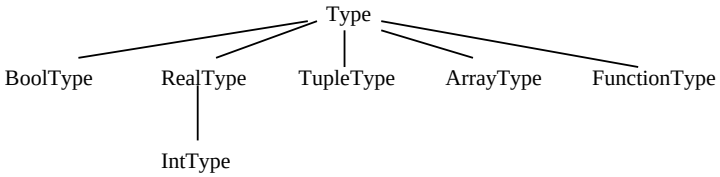
2 Type Equivalence

Purpose: inheritance, dynamic dispatch
Points: 20

We often think of programs as manipulating data. Sometimes, the data manipulated by a program is another program.[1] For example, compilers translate programs written in source code into a binary version that the computer understands. One useful task in manipulating programs is the ability to check for *type equivalence*. In this problem you will implement a method for checking if two types are *exactly* the same for a small set of types. This is mostly a thinking question; only a small amount of code needs to be written.

2.1 Implementing Type Equivalence Checking

We will represent each type using a Java class. For example, there will be `IntType` and `BoolType` classes to represent the types `int` and `boolean`, respectively. To facilitate declaring parameters and variables that can be references to any "flavor" of type object, there will also be a class `Type` that represents the abstract notion of a type. The figure below shows the whole type/class hierarchy for this problem:



The implementation for the base class `Type` can be downloaded from CMS. You should not need to change anything in the `Type` class. Your job is to implement classes to represent all of the other types in the hierarchy. Don't panic: each one is very short.

The `Type` class contains this method declaration:

```
//pre:  other non-null
//post: return true iff other has exactly the same type as this
public abstract boolean equalType(Type other);
```

This method is responsible for testing if the current type (that is, the type represented by the object the method is invoked on) is the same type as the type represented by the parameter named `other`. We can call this to check if two types are equivalent; your job is to define this method appropriately in the classes that extend `Type`.

Most of the types you are implementing are types you are familiar with using in Java. Here is a description of what each type is, and the definition of equivalence for the type.

`BoolType` The type for boolean values. A `BoolType` is type equivalent to any other `BoolType` object.

`RealType` The type for real numbers. (Java has 2 versions: `float` and `double`.) A `RealType` is type equivalent to any other `RealType` object.

`IntType` A specialization of real numbers that can only store integer values. Equivalent to any other `IntType` object.

`TupleType` A `TupleType` is a pair of types, which we'll call `first` and `second`. One example might be the pair (`BoolType`, `IntType`). This is *not* the same type as (`BoolType`, `RealType`). Two `TupleType` objects `a` and `b` are type equivalent if `a.first` is type equivalent to `b.first`, and `a.second` is type equivalent to `b.second`.

`ArrayType` The type for an array of values. Two arrays have the same type if they a) hold the same type of values, and b) have the same capacity.

[1] Software cannibalism!

`FunctionType` Some languages (not Java) fully support functions as another data type. In other words, they can be saved to variables, passed as parameters, etc. A `FunctionType` would be the type of a function in such a language. Two functions are type equivalent if a) they have the same name, b) their parameters are type equivalent, and c) their return values are type equivalent.

We will test your code using a test program. As a result, your classes must define the following constructors:

```
public BoolType();
public RealType();
public IntType();
public TupleType(Type first, Type second);
public ArrayType(int capacity, Type type);
public FunctionType(String funcName, Type parameters, Type retType);
```

Caveat

As you know, Java has an operator `instanceof` that can be used to check if an object IS-A given type. It is often used to make sure that a down-cast will work before doing the cast. **For this question, assume that `instanceof` is broken or does not exist.** Your solution must not use `instanceof`. Further, you should not need to perform any explicit casts (a.k.a. down-casts). *Read the comments and methods in class `Type` for clues about how to solve the problem with these restrictions.*

2.2 Followup Questions

Once your code is working, save a backup copy before answering these questions. That way you can exploratively alter the code and see what happens.

1. What happens if you add the following method to the `Type` class:

   ```
   protected boolean typeMatches(Type other)
   {
       return true;
   }
   ```

   Why does this happen?

2. Try to remove all duplicated code from your implementation. After all, we already have a base class (i.e. `Type`) where we can implement common functionality. Feel free to change class `Type` any way you like in your pursuit of duplication-free code. Is it possible to remove all duplicate code? If not, explain why.

3. Imagine a world where your instructor is a nice guy. He tells you that you can use `instanceof` and down-casts for this problem. Do you still need the `typeMatches()` methods defined in `Type`? Why or why not?

Turn in all the class definitions (including `Type`) in a single file called `Type.java`. Include the answers to the questions in Section 2.2 in the comments at the top of the file, below your name.

3 Binary Search Tree

Purpose: Binary search tree operations, recursion
Points: 30

As discussed in class, a binary search tree (BST) is a useful data structure for organizing information in a way that makes searching efficient. Your goal in this problem is to implement a binary search tree class with the following basic operations:

1. `public void insert(int key)`: add a data item to the tree. Remember that the item must be added to the tree carefully so as to preserve the sorted property.
2. `public boolean search(int key)`: search for the specified data item in the tree. If found, return true, otherwise return false.
3. `public int delete(int key)`: remove the specified data item from the tree and return it.
4. `public int depth(void)`: return the depth of the tree (i.e. the length of the longest path from the root to a child).
5. `public int nodeDistance(int key1, int key2)`: find the distance between the nodes containing the specified keys in the tree. For example, for the tree in Figure 1, the distance between nodes **d** and **f** is 2, while the distance between nodes **e** and **i** is 6. If either key is not in the tree, return -1.
6. `public void print(void)`: prints the tree in a breadth-first fashion. A breadth-first traversal visits the root, then all children of the root (from left to right), then all grandchildren of the root (from left to right), then all great-grandchildren, etc. You may want to use a data structure like `ArrayList` or `LinkedList`. Note that this is a different way to traverse the tree than the ones discussed in class (post-order, pre-order etc.) For example, the result of printing the nodes of the tree in Figure 1 using a breadth-first traversal is **a b j c g i d f e**.

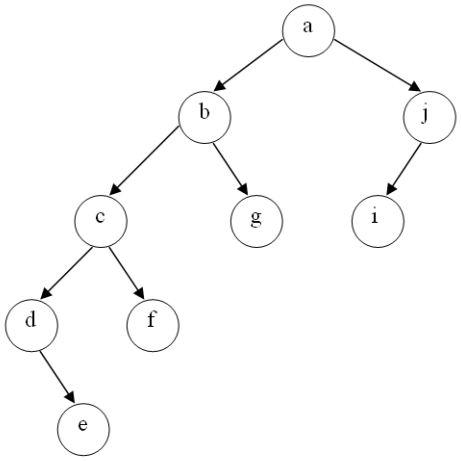


Figure 1. Binary Search Tree.

For the sake of simplicity, you may assume that the tree will store integers and that all the integers will be unique. (That is, you may assume that for any integer k, *insert(k)* will be called at most once.) To get you started, we have provided a skeleton file called `BinarySearchTree.java`. You may not rename or change the formal parameters of any of the methods above, though you may add private methods and/or private member variables to the `BinarySearchTree` class if you wish.

4 AVL Tree

Purpose: Inheritance, practice with trees
Points: 32

Binary search trees (BSTs) are incredibly useful. However, the Achilles Heel of traditional BSTs is that their efficiency at retrieving information is dependent on the order in which data was inserted into the tree. In the worst case scenario, if data is inserted into a BST in sorted order, the tree becomes unbalanced and degenerates into a linked list. Thus, maintaining a balanced BST is important for efficiency. This question

explores a BST-variant that incrementally reorganizes the tree during insertions and deletions to prevent the tree from becoming too unbalanced.

An **AVL Tree** is a Binary Search Tree with the extra property that at every node the difference between the heights of the right and left subtrees is at most one.

Let's define **balance factor** as difference the between the heights of the right and left subtrees of a given node. So, balance factor is equal to -1, 0 or +1 for an AVL tree.

Insertion. Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were a Binary Search Tree, and then retracing one's steps toward the root updating the balance factor of the nodes. If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary. (Rotations are described below.) If the balance factor becomes 2 or -2 then the tree rooted at this node is unbalanced, and a tree rotation is needed. At most a single or double rotation will be needed to balance the tree.

Let us call the node that must be rebalanced α. There are four possible cases when the balance factor becomes more than 1.

1. The insertion was into the left subtree of the left child of α.
2. The insertion was into the right subtree of the left child of α.
3. The insertion was into the left subtree of the right child of α.
4. The insertion was into the right subtree of the right child of α.

The first and the fourth cases can be fixed with *Single Rotation*. The second and the third cases can be fixed with *Double Rotation*.

Single Rotation. The Single Rotation for cases 1 and 4 are shown in Figure 2 and Figure 3. Note that the trees on the right are valid BSTs since $X < k1 < Y < k2 < Z$. Also, for every node and for nodes $k1$ and $k2$ the difference between the height of the right and left subtrees is at most one; therefore they AVL Trees.

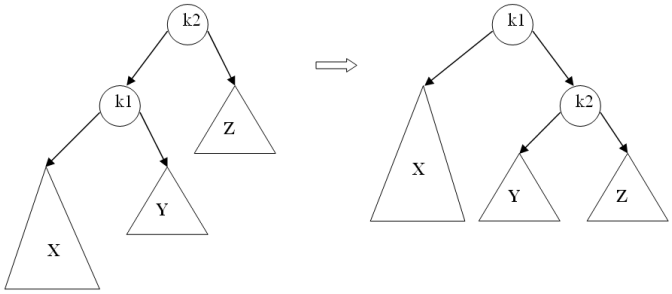


Figure 2. Single Rotation for case 1.

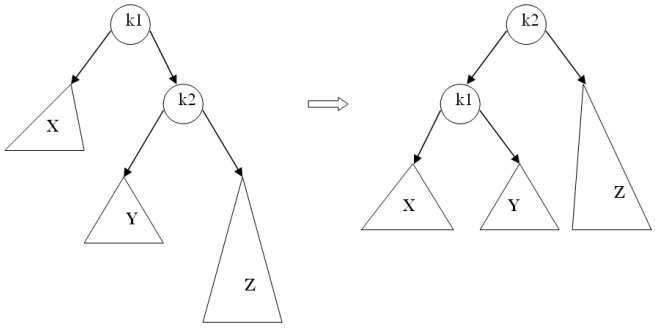


Figure 3. Single Rotation for case 4.

Double Rotation. A Single Rotation is not able to fix case 2 or case 3. Figure 4 illustrates the result of a single rotation for case 2. We can fix this using double rotation. The Double Rotation for cases 2 and 3 are shown at Figure 5 and Figure 6.[2]

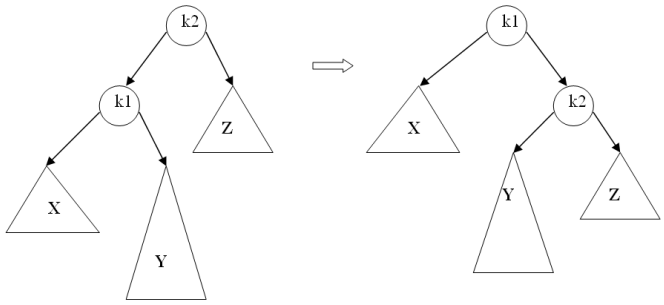


Figure 4. Single Rotation does not help for case 2.

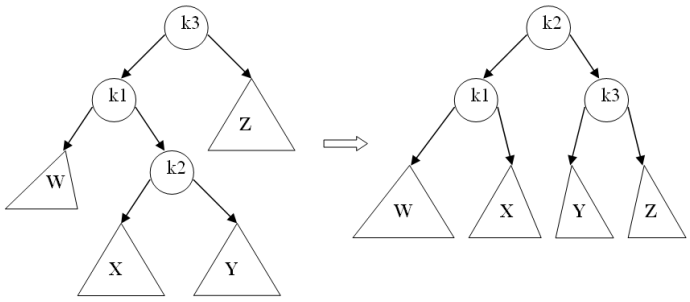


Figure 5. Double Rotation for case 2.

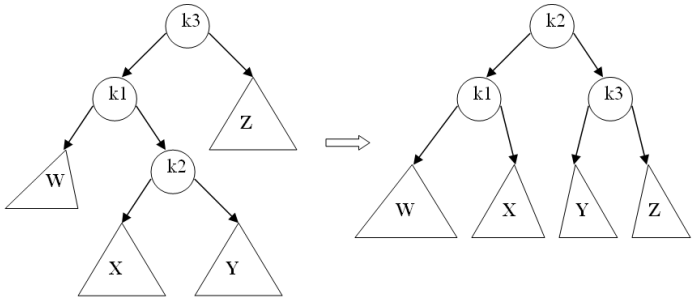


Figure 6. Double Rotation for case 3.

Deletion. If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as replacement has at most one subtree. After deletion retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by

[2]Oops! Fig 5 and 6 are actually the same. I will try to get the correct image file from Ainur. In the mean time, imagine that Fig 6 as it would be shown in a mirror. *–Art*

one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

Search. Search in an AVL Tree is performed exactly as in a Binary Search Tree.

Your task is to write a class called `AVLTree` that implements an AVL Tree data structure. Because an AVL Tree tree is a specialized type of binary search tree, it should be derived from the `BinarySearchTree` class you wrote in the previous problem. The class should inherit all the methods of `BinarySearchTree`. Put your implementation of AVLTree in a separate file called `AVLTree.java`.

Hint: Make sure you understand very well how the single and double rotation operations work before you start trying to write code. Run through single and double rotation operations with some sample trees using pencil and paper until you figure out how it works.

5 What to submit

Submit the following files:

1. `problem1.txt`
2. `Type.java`
3. `BinarySearchTree.java`
4. `AVLTree.java`

Extra Credit, Extra Fun!

If you have finished everything above and are craving some more programming, here are a couple extra credit items you can try your hand at. *We will award extra credit based on how well they are done. However, extra credit is always worth less than the main part of the homework. Therefore, your time is best spent finishing the homework well and only working on extra credit afterwards.*

Make sure you keep a pristine backup copy of the solutions before you start.

5.1 Saving and Loading BSTs

Points: 5

Implement the following methods for the class `BinarySearchTree`:

1. `public void saveTree(String fileName)`: save tree in the file.
2. `public static BinarySearchTree loadTree(String fileName)`: load tree from the file.

Make sure that you are loading the same tree as you saved in the file and there is no ambiguity in the structure of the tree.

Start Early and Good Luck! Have Fun!