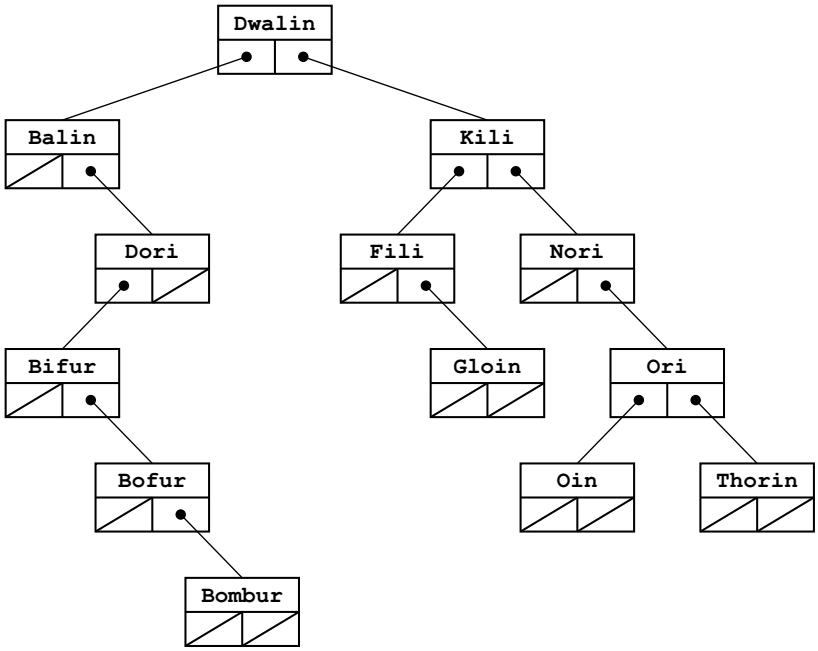
Eric Roberts
CS106B

Handout #49A
February 27, 2012

Solutions to Section Handout #6

1. Tracing binary tree insertion



1a. What is the height of the resulting tree? **6**

1b. Which nodes are leaves? **`Bombur`**, **`Gloin`**, **`Oin`**, and **`Thorin`**

1c. Which nodes are out of balance? **`Balin`**, **`Bifur`**, **`Dori`**, **`Dwalin`**, **`Kili`**, and **`Nori`**

1d. Which key comparisons are required to find the string **`"Gloin"`** in the tree?

`"Gloin" > "Dwalin"`, **`"Gloin" < "Kili"`** , **`"Gloin" > "Fili"`**, and **`"Gloin" == "Gloin"`**

2. Calculating the height of a binary tree

```
/*
 * Function: height
 * Usage: int h = height(tree);
 * ----------------------------
 * Returns the height of the tree, which is defined to be the length
 * of the longest path from the root to a leaf.
 */

int height(BSTNode *tree) {
   if (tree == NULL) {
      return 0;
   } else {
      return max(height(tree->left), height(tree->right)) + 1;
   }
}
```

3. Checking whether a tree is balanced

The simple coding looks like this:

```
/*
 * Function: isBalanced
 * Usage: if (isBalanced(tree)) . . .
 * ----------------------------------
 * Determines whether the tree is balanced, which requires both
 * of the following conditions:
 * 1. The heights of the left and right subtree differ by at most 1.
 * 2. The left and right subtrees are themselves balanced.
 */

bool isBalanced(BSTNode *tree) {
   if (tree == NULL) {
      return true;
   } else {
      return abs(height(tree->left) - height(tree->right)) <= 1
          && isBalanced(tree->left) && isBalanced(tree->right);
   }
}
```

This method, however, requires quadratic time because each subtree will be scanned over and over again as you go up the chain. To fix this problem, you can use any of several strategies. The code at the top of the next page uses a helper function that calculates the height and balance status at the same time. The balance status is returned as the value of the function; the height is returned through a reference parameter.

```
bool isBalanced(BSTNode *tree) {
   int height;
   return checkBalance(tree, height);
}

/*
 * Function: checkBalance
 * Usage: (not called by the client)
 * ---------------------------------
 * This function computes two properties of the tree simultaneously.
 * The result of the function itself is a bool indicating whether
 * the tree is balanced.  If it is balanced, the height of the tree
 * is returned in the reference parameter height, so that it can be
 * used in subsequent calculation.  Note that the height value is
 * not guaranteed to be correct if the tree is unbalanced.
 */

bool checkBalance(BSTNode *tree, int & height) {
   if (tree == NULL) {
      height = 0;
      return true;
   }
   int leftHeight, rightHeight;
   if (!checkBalance(tree->left, leftHeight)) return false;
   if (!checkBalance(tree->right, rightHeight)) return false;
   if (abs(leftHeight - rightHeight) > 1) return false;
   height = max(leftHeight, rightHeight) + 1;
   return true;
}
```

Problem 4a.

```
/*
 * File: class.h
 * -------------
 * This file is an interface to a simple class that represents partial
 * information about a class and its inheritance hierarchy.
 */

#ifndef _class_h
#define _class_h

#include <string>
#include "vector.h"

/*
 * Class: Class
 * ------------
 * This class represents one node in the inheritance tree and
 * corresponds to a class.  Classes are assumed to use single
 * inheritance, which is true of all classes in Java and all
 * classes we have seen in C++.  Thus, each class has a unique
 * superclass.
 */

class Class {

public:

/*
 * Constructor: Class
 * Usage: Class *c = new Class(name, superclass);
 * ----------------------------------------------
 * This function constructs a new Class object with the specified
 * name.  The superclass argument is optional; if supplied, it
 * makes this class a subclass of the specified class.
 */

   Class(std::string name, Class *superclass = NULL);

/*
 * Method: getName
 * Usage: std::string name = c->getName();
 * ---------------------------------------
 * Returns the name of the class.
 */

   std::string getName();

/*
 * Method: getSuperclass
 * Usage: Class *superclass = c->getSuperclass();
 * ----------------------------------------------
 * Returns the superclass of this class.  If the class was
 * defined without a superclass, this method returns NULL.
 */

   Class *getSuperclass();
```

☞

```
/*
 * Method: exportMethod
 * Usage: c->exportMethod(name);
 * -----------------------------
 * Indicates that the class exports a method with the specified name.
 */

   void exportMethod(std::string name);

/*
 * Method: getMethods
 * Usage: Vector<std::string> methods = c->getMethods();
 * -----------------------------------------------------
 * Returns a vector of the methods exported directly by this class.
 */

   Vector<std::string> getMethods();

private:
   std::string name;
   Class *superclass;
   Vector<std::string> methods;

};

#endif
```

Problem 4b.

```
/*
 * File: class.cpp
 * ---------------
 * This program implements the class.h interface.  The method bodies
 * are all so simple that no additional documentation is necessary.
 */

#include <string>
#include "class.h"
#include "vector.h"
using namespace std;

Class::Class(string name, Class *superclass) {
   this->name = name;
   this->superclass = superclass;
}

string Class::getName() {
   return name;
}

Class *Class::getSuperclass() {
   return superclass;
}

void Class::exportMethod(string name) {
   methods.add(name);
}

Vector<string> Class::getMethods() {
   return methods;
}
```

Problem 4c.

```
/*
 * Function: listAllMethods
 * Usage: listAllMethods(c);
 * -------------------------
 * Lists all the methods that can be called from the class c, including
 * all methods defined anywhere on the superclass chain.  Each method
 * name in the output is preceded by the name of the class in which it
 * appears.
 */

void listAllMethods(Class *c) {
   if (c != NULL) {
      listAllMethods(c->getSuperclass());
      Vector<string> methods = c->getMethods();
      foreach (string method in methods) {
         cout << c->getName() << "::" << method << endl;
      }
   }
}
```