CS106X
Spring 99-00

Handout #33
May 14, 2000

Section Exercises 7: Polymorphism and Trees

Problem 1 : Polymorphic ADTs

Suppose you need to write a program that will have stacks of several types of data, and thus will use a polymorphic stack ADT. As you recall, a polymorphic `void*` stack implementation stores pointers to the data rather than the data themselves. The implication is that all elements in the stack interface are passed and returned using `void*` pointers.

Making an ADT polymorphic vastly increases its usefulness since you can re-use the same ADT for many varying purposes without changes. But having to work with all elements by pointer comes with some pitfalls, such as the one Julie mentioned in class of entering the same pointer over and over again into a stack. That isn't the only trap that can befall you when working with a polymorphic ADT and this exercise tries to point out a few others.

a) Take a look at this code:

```
stackADT BuildStackFromArray(void)
{
     int arr[5] = {56, 34, 23, 10, 14};
     int i;
     stackADT s;

     s = NewStack();
     for (i = 0; i < 5; i++) {
        Push(s, &arr[i]):
     }
     return s;
}
```

It doesn't have the problem of repeatedly pushing the same element pointer, but it has an error of equally serious magnitude. Trace through and figure out where this code goes astray.

b) Here's another example of some trouble you could run into. You are using a stack to track user-entered numbers which are stored as strings returned from `GetLine()`. You use the following code segment to populate the stack:

```
stackADT s;
string numFromUser;

s = NewStack();
while (TRUE) {
  numFromUser = GetLine();
  if (StringEqual(numFromUser, ""))
     break;
  Push(s, numFromUser)
}
```

Later in the program, you want to add up the numbers entered by the user. You use the following loop:

```
int total, *numberPtr;

total = 0;
while (!IsStackEmpty(s)) {
  numberPtr = (int *)Pop(s);
  total += *numberPtr;
}
```

Everything compiles and seems to work okay until you check the value of `total` which has a completely unreasonable value. What went wrong?

Problem 2 : Binary Trees

a) Write a function which counts the number of nodes in a binary tree that have one or no direct children.

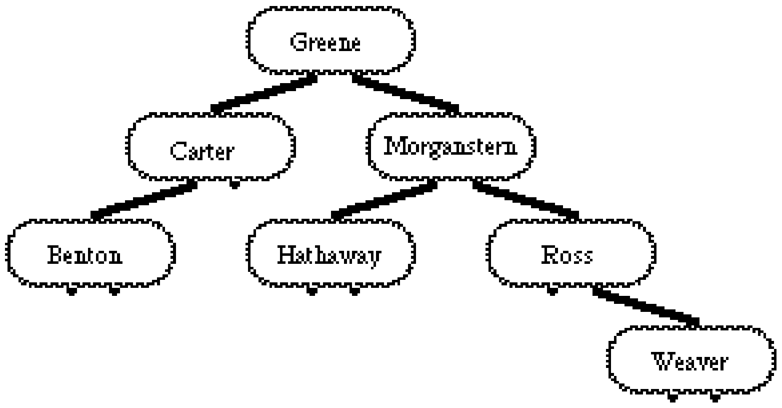
b) Write a function which takes two binary trees and returns a boolean value which indicates whether the two trees are equal (ie. have the same structure and values).

c) Write a function that will traverse a binary tree level by level—i.e. the root is visited first, then the immediate children of the root, next the grandchildren of the root, and so on. This is a *breadth-first* traversal. You will need to use the standard queueADT from Chapter 10 to help you manage the traversal. Print each node as you encounter it.

d) An interesting (and somewhat tricky) task for binary trees is deleting a node from a tree. As a challenging exercise, write a function which given a tree and a key, deletes the node with the indicated key, assuming that it exists in the tree at all. If the key does not exists, the function has no effect.

The difficulty that arises in trying to delete a node from a tree is that you are left with two orphaned trees, corresponding to the `left` and `right` subtrees of the deleted node. Figuring out an effective strategy for reconnecting the tree can be tricky. Spend some time drawing pictures and thinking about this problem before you try to write any code. In particular, think about all the different cases that could occur. How do you go about finding the node to delete? What if it isn't there? What if it has two children? Only one child? What if the children (if any) are leafs? What if they are trees?

Problem 3: Binary Search Tree Analysis



Given the above binary search tree:

- **a)** What is the height of this tree? What is the depth of the node 'Ross'? Who are the descendants of the node 'Morganstern'?
- **b)** What is the sequence of nodes visited on a pre-order tree traversal? in-order? post-order?
- **c)** If we entered the node 'Del Amico', where would it end up in this tree?
- **d)** What was the first node inserted into this tree?

Problem 4: Symbol Tables

a) Suppose you are using a symbol table to keep track of employee data. The keys in you table are strings that are the names of the employees. The values in the table are void * pointers that point to client-side records of the following type:

```
typedef struct {
  int idNum;
  string dept;
  int yrEmployed;
  double salary;
} employeeDataT;
```

As items are added to the table, where and how do you allocate storage for the employee data?