CS106B Handout #36
Spring 99-00 May 26, 2000

Final Practice

Final Exam: Tuesday, June 6th 12:15 –3:15pm **Kresge Auditorium**

Everyone must attend the regular exam at its scheduled time.

Review Session: Sunday, June 4th, 7-9pm, Location TBA

Your regular section meeting next week will include time for discussing the problems on this sample exam as well as general questions you might have about the course material. Our marvelous head TA Yves will also hold a general review session Sunday night. We'll let you know the location as soon we find it out ourselves!

Coverage

The final is comprehensive and covers material from the entire quarter, but will tend to focus on the topics covered after the midterm. It will be a 3 hour exam. It will be open book /open note, but no computers!

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the exam. All of these problems have appeared on various past exams. We definitely recommend working through the problems in test-like conditions to prepare yourself for the actual exam. In order to help encourage this, we won't give out the solutions until next class. Some of our section problems have been taken from previous exams and chapter exercises from the text often make appearances in same or similar forms on exams, so both of those resources are a valuable source of study material as well.

Be sure to bring your text with you to the exam. We won't repeat the standard ADT definitions on the exam paper, so you'll want your text to refer to.

(FYI: All the space usually left for answers was removed in order to conserve trees).

General instructions to be given for the exam

Answer each of the questions included in the exam. Write all of your answers directly on the exam, including any work that you wish to be considered for partial credit.

When writing programs for the exam, you do not need to be concerned with `#include` lines, just assume any of the libraries that you need (both standard C or 106-specific) are already available to you. If you would like to use a function from a handout or textbook, you do not need to repeat the definition on the exam, just give the name of the function and the handout or chapter number in which its definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Problem 1: Short answer

a) Starting with an empty tree, draw the resulting from inserting these numbers into a binary search tree in this order:

25 19 41 24 20 43 58 26 35

Show the order the values would be printed in a pre-order traversal of your tree.

Show the tree that would result after deleting 41 from your tree.

b) For the stack and queue, we used the generic pointer **`void *`** to implement polymorphism. What does it mean for an ADT to be polymorphic and why is it desirable?

However, in C, using **`void *`** has some drawbacks. Explain two of them.

c) Given this code fragment:

```
typedef union {
     string s;
     int num;
     double *dp;
} binky;

main() {
     binky b;
     ...
```

How much space does the variable **`b`** require? Add code to main to set the string field to "winky".

d) The grammar below for conditional expressions is ambiguous:

```
E -> E conj E
E -> T
Conj -> AND
Conj -> OR
T -> boolean constant
T -> identifier
```

Give an example expression that shows the ambiguity in the grammar. How could you resolve the ambiguity?

e) The "shape" of a Huffman coding tree gives information about how well the algorithm will be able to compress the input. What tree shape leads to significant compression in the output? What shape that leads to little or no compression?

f) Write a short function that given an integer will return the remainder of that number when divided by 4. The function should use only bit operations (ie not / or %).

Problem 2: Function Pointers

Microsoft has just been granted a trademark on the letter M and insists that everyone not licensed remove all Ms from their words. You are to write the function **`RemoveAllMs`**. Given a string, this function will create and return a new string consisting of the original string with all the Ms removed. The function should not alter the original string, it should return a new string. You cannot use any strlib.h functions whatsoever (i.e. no CopyString, Concat, CharToString, Substring, and so on). If passed the string "Mickey Mouse", your function should return the new string "ickey ouse".

a) Write **`RemoveAllMs`**:

```
string RemoveAllMs(string word)
```

b) Generalize the function from part (a) to remove all letters that pass a test that is supplied as a parameter. For example, you could call the function to create a new string with all uppercase letters removed or all punctuation removed. The **`RemoveIf`** function is implemented similarly to the **`RemoveAllMs`** function, but takes one more parameter: the predicate function. **`RemoveIf`** applies the predicate function to each letter in the string and doesn't include the letter in the resulting string if it passes the test.

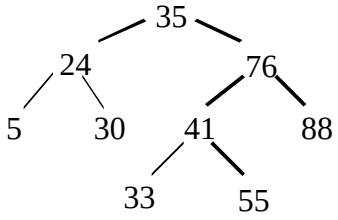
First, create a typedef for the class of function you will use a parameter:

Now implement the **`RemoveIf`** function. (Instead of repeating the code from part a, you can indicate which lines stay the same and re-write the lines that are different here, as long as it is clear what you are doing.

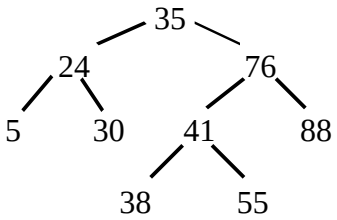
c) You decide to add a **`RemoveIfNot`** function that operates as in part b, except that it removes any element that does not pass the predicate test. The client can use it to remove all set members that are not divisible by 3 or not prime. At first glance, you might consider duplicating the above code and making a few minor changes to implement this. Instead you will create one private helper function which is called by both **`RemoveIf`** and **`RemoveIfNot`**. **`RemoveIf`** and **`RemoveIfNot`** now become "wrapper" functions that call this helper function passing some additional state. Show the changes you need to make to turn the above function into the helper function and then show the code for the wrapper functions **`RemoveIf`** and **`RemoveIfNot`** that you will export to the client.)

Problem 3: Binary Trees

Write a predicate function **`IsSearchTree`** that takes a binary tree of integers and returns whether or not this tree is ordered so that it fulfills the binary search tree property. In a properly ordered search tree, every node in the left subtree is less than the current node and every value in the right subtree is greater and this holds throughout the entire tree. You can assume all the values in the tree are distinct and the values are numbers in the range -10000 to 10000 inclusive. A NULL tree is considered to be a valid search tree. Carefully consider the following two trees and make sure your function will correctly report that the left one is NOT a search tree:



Not Search Tree



Is Search Tree

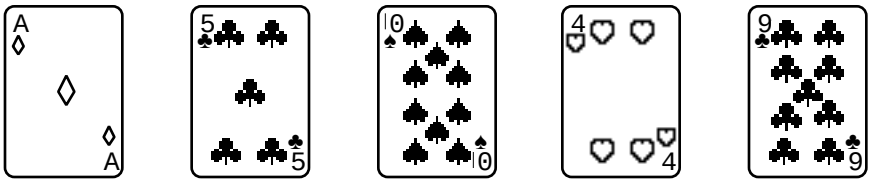
Here is the type definition for a tree. Your job is to write the **`IsSearchTree`** function below. You may want to write a helper function.

```
typedef struct _node {
      int value;
      struct _node *left, *right;
} node, *tree;

bool IsSearchTree(tree t)
```

Problem 4: Recursion

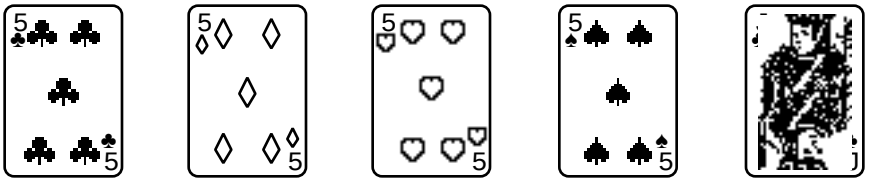
In the card game called Cribbage, part of the game consists of adding up the score from a set of five playing cards. One of the components of the score is the number of distinct card combinations whose values add up to 15, with aces counting as 1 and all face cards (jacks, queens, and kings) counting as 10. Consider, for example, the following cards:



There are three different combinations that sum to 15, as follows:

AD + 10S + 4H **AD + 5C + 9C** **5C + 10S**

As a second example, the cards



contains the following eight different combinations that add up to 15:

5C + JC	**5D + JC**	**5H + JC**	**5S + JC**
5C + 5D + 5H	**5C + 5D + 5S**	**5C + 5H + 5S**	**5D + 5H + 5S**

Assume you have access to a **card.h** interface that exports these types and operations:

```
typedef enum {Ace = 1, Two, Three, ...,Ten, Jack, Queen, King} rankT;
typedef enum {Clubs, Hearts, Spades, Diamonds} suitT;

cardADT NewCard(rankT rank, suitT suit);
rankT Rank(cardADT card);
```

Write a function

```
int CountFifteens(cardADT cards[], int n);
```

that takes an array of **cardADT** values and the effective size of the array and returns the number of different combinations of cards in the array that sum to fifteen. For example, the following code replicates the first example and should display the value 3:

```
main()
{
    cardADT hand[5];

    hand[0] = NewCard(Ace, Diamonds);
    hand[1] = NewCard(5, Clubs);
    hand[2] = NewCard(10, Spades);
    hand[3] = NewCard(4, Hearts);
    hand[4] = NewCard(9, Clubs);
    printf("%d\n", CountFifteens(hand, 5));
}
```

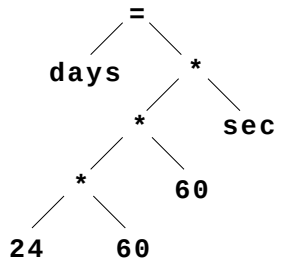
Problem 5: Parsing and expressions

After a commercial compiler parses an expression, it typically looks for ways to simplify that expression so that it can be computed more efficiently. This process is called **optimization.** A part of the optimization process consists of identifying subexpressions that are composed entirely of constants and replacing them with their value. For example, if a compiler encountered the expression

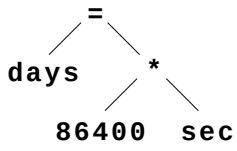
```
days = 24 * 60 * 60 * sec
```

there would be no point in generating code to perform the first two multiplications when the program was executed. The value of the subexpression **`24 * 60 * 60`** is constant and might as well be replaced by its value (86400) before the compiler actually starts to generate code.

Write a function **`SimplifyConstants`** that takes an **`expressionADT`** and returns a new **`expressionADT`** in which any subexpressions that are entirely composed of constants are replaced by the computed value. Thus, if **`exp`** contained the expression tree



calling **`SimplifyConstants(exp)`** should return a new expression tree that looks like this:



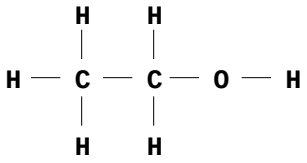
Problem 6: Using ADTs

Write a function **`PathExists(n1, n2)`** that returns **`TRUE`** if there is a path in the graph from node **`n1`** to node **`n2`**. You do not need to find the shortest path, simply use a depth-first search to traverse the graph from **`n1`**; if you encounter **`n2`** along the way, then a path exists. You should not use recursion, instead your function should maintain an explicit stack to store unexplored nodes. You have available to you the standard **`stackADT`** from Chapter 8, where **`stackElementT`** has been typdef-d to hold **`nodeADT`** elements. You also have the standard **`graphADT`**, **`nodeADT`**, and **`arcADT`** from Chapter 16 at your disposal.

```
bool PathExists(nodeADT start, nodeADT finish)
```

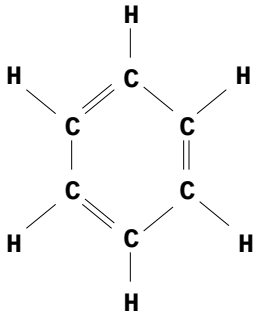
Problem 7: Designing ADTs

The structure of molecules, particularly in organic chemistry, is often represented geometrically using diagrams like these:



ethyl alcohol

acetylene



benzene

Each molecule is composed of individual atoms, which are represented by their chemical symbols (`C` for carbon, `H` for hydrogen, `O` for oxygen, `Cl` for chlorine, and so forth). The individual atoms in a molecule are linked to other atoms by chemical bonds, which are represented by the lines in the diagram. In some cases, the connection may be a double bond, as indicated by the doubled lines in the benzene molecule, or even as a triple bond as shown in the acetylene example.

Suppose you have been asked by a group of chemists to design an abstract data type that can represent molecular structures of this sort. The requirements of the interface are as follows:

- The interface must export at least two abstract types: one for an individual atom and one for a complete molecule.
- Given an atom, it must be possible to determine what kind of atom it is.
- For a particular atom in a molecule, it must be possible to determine what other atoms it is connected to and by what kind of bond (single, double, or triple). Note that the connection type is a property of the bond and not of the atom; in acetylene, for example, the two carbon atoms are connected to each other by a triple bond and to their respective hydrogen atoms by a single bond.
- For a given molecule, it must be possible to identify all of the atoms within it. Thus, given the ADT for the acetylene molecule, it must be possible somehow to get access to each of the four atoms (`H`, `C`, `C`, and `H`) that make it up.

Your interface is a low-level interface and need not provide any capabilities other than the ones listed here. For example, you do not have to store any information about atoms other than their chemical symbol, nor do you have to represent anything about the connection structure beyond representing the existence and type of the individual bonds. You may also count on the following restrictions:

- All chemical symbols are strings of one or two characters.
- No atom in a molecule is ever connected to more than four other atoms.
- There are no other types of bonds besides single, double, and triple.

a) Design the interface (i.e., the header file **`"molecule.h"`**) that can represent atoms and molecules as described above. Your interface specification should include the declarations of any types and functions exported by the interface, along with very short comments that show potential clients how to use those functions.

b) Using C code only if you think it is clearer to do so, sketch in English an implementation strategy for the data types and functions you defined in part a. Remember that your answer will be evaluated as an essay question, and the important criterion is whether it is clear that you have a successful strategy for implementing the package. Correct answers can be quite short, particularly if you describe your implementation strategy in terms of existing models from other abstractions you have seen in the course.