Final Practice

Final exam: Monday, June 5th 8:30-11:30 am
Gates B01(next door to our usual room)

Everyone must attend the regular exam at its scheduled time. We wouldn't want SITN students to miss out on regular sweaty-palm tense-room experience, so they'll join us on campus.

Coverage

The final will cover all of the course material, but will tend to focus more on the later half of the course. 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 106X exams of mine in the past. 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 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

List the order the values would be printed in a pre-order traversal of 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) In lecture, we discussed using the last 2 digits of a phone number as a hashing function for sorting a store's customers. The store manager at The Gap in the Stanford Shopping Center notes that it is much easier for Stanford students to quickly determine the first 2 digits of their phone number than the last 2, so he proposes using the first 2 digits instead. What effect will this have on the memory needs of the customer hash table? What effect will this have on the performance of lookup?

d) You have decided to add a new function **`InsertString`** to the editor buffer interface. One possible implementation of this function would call the standard **`InsertChar`** function in a loop for each character in the string. An alternate implementation would not use **`InsertChar`** but instead insert the string directly using an implementation-specific strategy. For each approach, give one advantage that it has over the other.

e) The **`lexiconADT`** supplied for Boggle included a function **`IsPrefix`** that allowed you to check if a given string was a prefix of any word in the dictionary. Consider the importance of **`IsPrefix`** in implementing the Boggle program. What would happen if this function were not available? Could you still implement Boggle?

f) The version of **`Quicksort`** we used in class to illustrate the algorithm used the first element of the subarray as the pivot element. This is a convenient choice but can result in poor overall performance. Explain.

One suggestion to address this issue was to choose an element at random for the pivot. Couldn't this still result in poor performance? Why is this any improvement?

g) Write a 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: Recursion

Write the **ArithmeticCombinations** function which given an array of integers and a desired result, recursively determines how many different ways you can combine those integers to form that result. Combining the numbers means choosing a sequence of arithmetic operations to apply to the numbers to produce an expression that evaluates to the result. The sum starts as the first number in the array and you process the remaining array entries from left to right. For each number, you can either add it to the current sum, subtract it from the current sum or multiply the current sum by the number (i.e. valid operators are **+**, **-**, and *****). All numbers in the array must be used in the expression and must appear in the same order in the expression as in the array. For example, consider the array **{7, 0, -3, 4, 10}** and desired result **6**. The function should return 3 since there are three possible combinations:

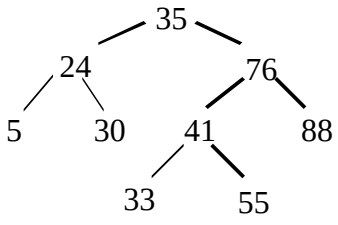
```
7 * 0 * -3 - 4 + 10 = 6
7 + 0 + -3 * 4 - 10 = 6
7 - 0 + -3 * 4 - 10 = 6
```

Note the expression is **not** evaluated using C precedence rules, it is simply evaluated left to right. The first two parameters to the function are the array of numbers and its effective size. You may assume the array always has at least one entry. The third parameter is the desired result. Your function should return the count of the different ways you can combine the numbers to get that result (you do not have to keep track of the combinations, just report the number of possible combinations). The array should not be modified. You will need a wrapper function.

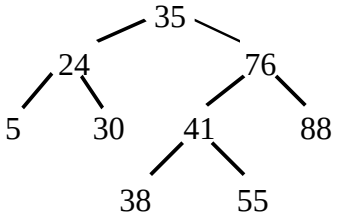
```
int ArithmeticCombinations(int arr[], int n, int result)
```

Problem 4: 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 5: Designing and implementing ADTs

You are going to implement an ADT that can represent a matrix of floating point values. A matrix is a form of a two-dimensional array. Each row and col has a value which, in this case, will be of type `double`. Typical operations on such an ADT would probably include:

```
matrixADT NewMatrix(int numRows, int numCols);
void FreeMatrix(matrixADT m);
double GetElementAt(matrixADT m, int row, int col);
void SetElementAt(matrixADT m, double value, int row, int col);
void PrintMatrix(matrixADT m);
```

The hitch is that you know this ADT will be used most often to represent very large and very sparse matrices. A *sparse* matrix is one in which most of the elements are zero. A matrix of 1000 rows and 10000 columns might only have 100 non-zero values. The overhead of actually using a 2-D array would be prohibitive[1] and so you will need to develop an alternate representation.

In implementing the matrixADT, you must only store those elements that are non-zero. For each row, you must keep a singly-linked list of the non-zero elements, and the elements in each row should be stored in sorted order (sorted by column). The decisions of what data is stored in the CDT itself, what is kept in each list cell, whether to have a dummy header cell, etc. are left up to you. Your choices must result in an implementation that at least meets the following running time specifications (where R is the number of rows, C the number of columns):

NewMatrix	O(R)
FreeMatrix	O(R*C)
GetElementAt	O(C)
SetElementAt	O(C)
PrintMatrix	O(C*R)

The three functions in bold are ones that you have to write. You will not have to implement the others. As always, you are encouraged to write private helper functions to assist you in implementing the public operations. The interface file is given beginning on the next page. Be sure to carefully read the descriptions provided in the header file and make sure that your implementation is true to the published interface.

This space intentionally left blank. Please go on to next page.

[1] If you used the declaration

```
double matrix[1000][10000];
```

The variable `matrix` would take up sizeof(double)*1000*10000 = 80 million bytes or ~80 MB, about the size of your entire zip disk!

```
/* File: matrix.h
 * -------------
 * This interface exports the type and operations for a matrix of
 * floating points values.
 */
#include "genlib.h"

typedef struct matrixCDT *matrixADT;


/* Function: NewMatrix
 * Usage: m = NewMatrix(1000,5000)
 * -------------------------------
 * Creates a new matrixADT object and returns it.  The new matrix has the
 * specified number of rows & cols.  There is no upper limit on the size of
 * a matrix that can be created.  Each element is assumed to be zero at start.
 */
matrixADT NewMatrix(int numRows, int numCols);

/* Function: FreeMatrix
 * Usage:  FreeMatrix(m);
 * ----------------------
 * Frees the storages allocated with matrix m.
 */
void FreeMatrix(matrixADT m);


/* Function: SetElementAt
 * Usage:  SetElementAt(m, 3.14159, 12, 4300);
 * -------------------------------------------
 * Sets (or replaces!) the value at (row,col) position in matrix with new
 * value.  Raises an error when asked to set a position that is not a legal
 * row and col position. Row and col both start with 0 index.
 */
void SetElementAt(matrixADT m, double value, int row, int col);


/* Function: GetElementAt
 * Usage:  GetElementAt(m, 945, 4999);
 * -----------------------------------
 * Returns the value at (row,col) position of the matrix.  Any position
 * which has not been set is assumed to be zero.  Raises an error when asked
 * to retrieve a position that is not legal element in the matrix.
 */
double GetElementAt(matrixADT m, int row, int col);


/* Function: PrintMatrix
 * Usage: PrintMatrix(m);
 * ----------------------
 * Prints matrix in row col format. Do not be concerned about decimal
 * places, lining up decimals points and other printf formatting details.
    3.4 0.0 5.12           /* Row 0 */
    0.0 0.0 0.0            /* Row 1 */
    0.0 0.1 0.0            /* Row 2 */
 */
void PrintMatrix(matrixADT m);
```

Fill in the implementation of matrix.c. Define helper functions as needed. You can assume the FreeMatrix and GetElementAt functions are already written for you.

/* put type definitions, including struct matrixCDT, here */

```
matrixADT NewMatrix(int numRows, int numCols)
```

```
void SetElementAt(matrixADT m, double value, int row, int col)
```

```
void PrintMatrix(matrixADT m)
```

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)
```