**Lecture 30 – Final Exam Review – Wed Dec 6 or Thurs Dec 7**

**Announcements**

Assignments:

* Assignment #11 should be graded by next Monday/Tuesday – see blackboard for grades

**Today’s Goals**

1. Trees
2. Secure C Programming
3. Final Exam Review

**Today’s Terminology**

**Terminology**

* Binary Trees
  + Nonlinear, two-dimensional data structure

**Trees**

**Trees**

* Linked lists, stacks, queues are linear data structures
* Tree - non-linear data structures

**Binary Trees**

* Each node contains two pointers (links)
  + Each pointer refers to a child
* A binary tree contains
  + root node – 1st node
  + left children – nodes reached through left pointer link
  + right children – nodes reached through right pointer link

**rootPtr**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | P |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | C |  |  |  |  |  | C |  |

Null Null Null Null

* A binary ***search*** tree
  + Values in ***any left subtree*** are less than the value in its parent node
  + Values in ***any right subtree*** are less than the value in its parent node

**Binary Tree Example #1**

* Create a simple tree with numbers {2, 4, 6}
* Result:

**rootPtr**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | 4 |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 2 |  |  |  |  |  | 6 |  |

Null Null Null Null

**struct** treeNode {

**struct** treeNode \*leftPtr; Note that there are two pointers!

**int** data;

**struct** treeNode \*rightPtr;

};

**int** **main**(**void**) {

// Create two tree node pointer variables

**struct** treeNode \*rootPtr = NULL;

**struct** treeNode \*treeNodePtr = NULL;

// Allocate memory in the heap for the root node

rootPtr = **malloc**(**sizeof**(**struct** treeNode));

// Setup the 1st tree node

**if** (rootPtr != NULL) {

rootPtr->data = 4;

rootPtr->leftPtr = NULL;

rootPtr->rightPtr = NULL;

}

// Allocate memory in the heap for the 2nd tree node

treeNodePtr = **malloc**(**sizeof**(**struct** treeNode));

// Setup the 2nd tree node

**if** (treeNodePtr != NULL) {

treeNodePtr->data = 2;

treeNodePtr->leftPtr = NULL;

treeNodePtr->rightPtr = NULL;

}

// Insert 2nd node as the left child!

rootPtr->leftPtr = treeNodePtr;

// Allocate memory in the heap for the 3rd tree node

treeNodePtr = **malloc**(**sizeof**(**struct** treeNode));

// Setup the 3rd tree node

**if** (treeNodePtr != NULL) {

treeNodePtr->data = 6;

treeNodePtr->leftPtr = NULL;

treeNodePtr->rightPtr = NULL;

}

// Insert 3rd node as the left child!

rootPtr->rightPtr = treeNodePtr;

// Display values in tree using only rootPtr

**printf** ("rootPtr->data = %d\n", rootPtr->data);

**printf** ("rootPtr->leftPtr->data = %d\n", rootPtr->leftPtr->data);

**printf** ("rootPtr->rightPtr->data = %d\n", rootPtr->rightPtr->data);

// Now we can traverse the tree in the different orders:

// inorder, preorder, postorder

**puts** ("The inOrder Traversal is: ");

inOrder (rootPtr);

} // main

// InOrder traversal means:

// Traverse the left subtree inOrder

// Process the value in the node

// Traverse the right subtree inOrder

// For binary search tree prints the values in ascending order

**void** **inOrder** (**struct** treeNode \*treePtr) {

// If tree is not empty, then traverse

**if** (treePtr != NULL) {

inOrder(treePtr->leftPtr);

**printf**("%3d", treePtr->data);

inOrder(treePtr->rightPtr);

}

}

**Displays**

rootPtr->data = 4

rootPtr->leftPtr->data = 2

rootPtr->rightPtr->data = 6

The inOrder Traversal is: 2 4 6

**Trace**

**I** **added print statements to create this trace:**

**printf** ("treeNodePtr = %p\n", treeNodePtr);

**printf** ("treeNodePtr->data = %d\n", treeNodePtr->data);

rootPtr = 00992AD8 Value stored in rootPtr => 1st node (4)

&rootPtr = 0028FD08 Address of rootPtr

rootPtr->data = 4

treeNodePtr = 00992AF0 Value stored in treeNodePtr => 2nd node (2)

treeNodePtr->data = 2 00992AF0 was assigned to rootPtr->leftPtr

treeNodePtr = 00992B08 Value stored in treeNodePtr => 3rd node (3)

treeNodePtr->data = 6

**Variable Value Stack Address**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rootPtr** | 00992AD8 | **0028FD08** | **Heap address** of 1st node (node = 4) | |
| rootPtr->leftPtr | 00992AF0 |  | **Heap address** of 2nd node (node = 2) | |
| rootPtr->rightPtr | 00992B08 |  | **Heap Address** of 3rd node (node = 6) |

**rootPtr**

**00992AD8**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00992AF0 | 4 | 00992B08 |

Address of node

with data = 2

**00992AF0 00992B08**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 00000000 | 2 | 00000000 |  |  |  | 00000000 | 6 | 00000000 |

**Tree InsertNode Example:**

* Assume tree from above

**struct** treeNode {

**struct** treeNode \*leftPtr;

**int** data;

**struct** treeNode \*rightPtr;

};

**int** **main**(**void**) {

// Assume we have the tree we just created!

**printf** ("Enter a value for node to add to tree ");

**scanf** ("%d", &number);

// rootPtr address is necessary

// Providing the address enables the value stored in rootPtr to be modified

insertTreeNode(&rootPtr, number);

**printf** ("The inOrder Traversal is: ");

inOrder (rootPtr);

}

**void** **insertTreeNode** (**struct** treeNode \*\*treePtr, **int** number) {

// If the tree is empty

**if** (\*treePtr == NULL) {

// Base case to stop recursion!

// Allocate memory in the heap for the node

\*treePtr = **malloc**(**sizeof**(**struct** treeNode));

// Setup the tree node

**if** (\*treePtr != NULL) {

(\*treePtr)->data = number;

(\*treePtr)->leftPtr = NULL;

(\*treePtr)->rightPtr = NULL;

}

**else** {

**printf** ("No memory available to insert %d\n", number);

}

}

**else** {

// Tree is not empty

// Determine if inserting on left or right side of tree

**if** (number < (\*treePtr)->data) {

insertTreeNode( &((\*treePtr)->leftPtr), number );

}

**else** **if** (number > (\*treePtr)->data) {

insertTreeNode( &((\*treePtr)->rightPtr), number );

}

**else** {

**printf** ("%d is a duplicate value so don't insert\n", number);

}

} // insert into tree with nodes

} // treeInsertNode

**Trace**

**I** **added print statements to create this trace:**

rootPtr = 00992AD8 Value stored in rootPtr => address of node 4

&rootPtr = 0028FD08

rootPtr->data = 4

rootPtr->leftPtr = 00992AF0 Value stored in rootPtr->leftPtr => node 2

rootPtr->rightPtr = 00992B08 Value stored in rootPtr->rightPtr => node 6

**\*\*\* 1st CALL TO InsertTreeNode \*\*\***

(\*treePtr)->data = 4

\*treePtr = **00992AD8**

&(\*treePtr) = 0028FD08

**00992AD8**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00992AF0 | 4 | 00992B08 |

**Calling insert for LEFT tree**

(\*treePtr)->leftPtr = 00992AF0 Address of node 2

&((\*treePtr)->leftPtr) = **00992AD8** ***Address of leftPtr*** is inside node 4!

insertTreeNode( &((\*treePtr)->leftPtr), number ); **Calling insertTreeNode and sending**

**address of the leftPtr which is the address of node 4!**

**\*\*\* 2nd CALL TO InsertTreeNode \*\*\***

(\*treePtr)->data = 2

\*treePtr = 00992AF0

&(\*treePtr) = **00992AD8**

**00992AF0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00000000 | 2 | 00000000 |

**Calling insert for LEFT tree**

(\*treePtr)->leftPtr = 00000000 Bottom – leaf node!

&((\*treePtr)->leftPtr) = 00992AF0 ***Address of leftPtr*** is inside node 2!

insertTreeNode( &((\*treePtr)->leftPtr), number ); **Calling insertTreeNode and sending**

**address of the leftPtr which is the address of node 2!**

**\*\*\* 3rd CALL TO InsertTreeNode \*\*\***

BASE CASE - allocate memory for node

(\*treePtr)->data = 0 **HOW NEW NODE IS CONNECTED!**

\*treePtr = 00992B20 1) Node 2’s leftPtr is sent

&(\*treePtr) = 00992AF0 2) leftPtr is NULL! Base case

3) Memory allocated for new node

4) New node is 00992B20

**leftPtr is changed to 00992B20**

5) Because sent address of leftPtr

new node is connected into tree

**00992AF0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00992B20 | 2 | 00000000 |

**00992B20**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00000000 | 0 | 00000000 |

**Returned from recursive insert on LEFT tree**

**Returned from recursive insert on LEFT tree**

**00992AD8**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00992AF0 | 4 | 00992B08 |

**00992AF0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00992B20 | 2 | 00000000 |

**00992B20**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 00000000 | 0 | 00000000 |

**Secure C Programming 5 min**

**Pointers should not be left uninitialized**

* Always assign a pointer either the value NULL or the address of a valid it in memory.

**Undefined behavior occurs when attempt to use free on memory already deallocated**

* This is called “double free vulnerability”

**Final Review**

Covers

* Chapter 1-12

Time

* Mon/Wed class - **Wed Dec 13th 8:00 am to 10:00 am**
* Tues/Thurs class – **Thurs Dec 14th 8:00 am to 10:00 am**

This is an overview of the material you should study for the final exam.

**Chapter 1 - Introduction to Computers, the Internet and the Web**

* Difference between compiler, logical and runtime errors
  + What causes each type of error?
  + When or how they occur

**Chapter 2 – Introduction to C**

* Know how to print output to console and obtain input from keyboard
* Know the different data types and what a "data type" tells the compiler
  + int, float, double, char
* Know when and how to declare variables, assign values to, use in statements
* Know when and how to declare constants, assign values to, use in statements
* Be able to use numeric operations (+, - , \*, /, %)
  + Integer division and the issues you can have with integer division
* Know how to evaluate numeric expressions
* Understand operator precedence when it comes to evaluating numeric expressions
* Know what the **relational operators** and **equality operators** are
* Know how to use relational operators to create boolean expressions
* Know how to evaluate boolean expressions

**Chapter 3 – Structured Program Development in C**

* Understand how **if statements** work
* Understand how a **while loop** works
* Simple type conversions - explicit casting – we did this with integer division example
* Be able to use the increment and decrement operators
* Be able to understand code and/or write code snippets with **if statements**
* Be able to understand code and/or write code snippets with **while loops**

**Chapter 4 – C Program Control**

* Understand how a ***for loop*** works
* Understand how a ***do-while loop*** works
* Know when to use one loop structure over the other (while, for, do-while)
* Know how loops work when they are nested
* Understand what causes infinite loops
* Understand how a **switch statement** works
  + Understand how fall-through behavior works
  + Understand how the default case works
* Understand char data types
  + What ASCII code is and how it is used at a high level
* Know what the **logical operators** are
* Know how to use **logical operators** with the **relational/equality operators** to create boolean expressions
* Be able to understand code and/or write code snippets with **for loops** and **do-while loops**
* Be able to understand code and/or write code snippets with **switch statements**

**Chapter 5 – C Functions**

* Know how to use these 2 basic math functions - sqrt, pow
* Know how to define and create a function, so understand:
  + Function prototypes
  + Function invocation
  + Function definition
* Understand how to pass values to a function and how to return values from functions
* Understand automatic conversions:
  + Know that conversions occur when argument types do not correspond to parameter types
  + Know what kinds of issues automatic conversions can cause
* Understand the scope of variables and how to determine the scope of variables
* Understand what ***pass by value*** means and the impact on variables when a function is called
* Know what recursion is on a high-level, you will NOT be asked to write a recursive function
* Be able to explain code, trace code, and write code snippets with functions

**Chapter 6 – C Arrays**

* Know how to declare, create, and initialize an array (one or two dimensional)
* Know how to access elements within an array (one or two dimensional)
* Understand initialization of arrays
  + What is in array when you don’t initialize the array
  + Understand initializer lists (i.e. how they work, what if fewer items than array elements, etc.)
* How to manipulate elements within an array (i.e. compute sum, find largest, display elements, etc.)
* Understand what happens with arrays
  + When code accesses elements in an array outside the array bounds
  + When there is an off-by-one situation (mostly issue when forgetting arrays are zero-based)
* Understand character arrays
  + Know how to create and initialize strings using character arrays
  + Understand the importance of the null character in strings
  + How to read and display strings
* Understand and know how to ***pass arrays to functions***
  + Understand what ***pass by reference*** means and the impact when passing an array to a function
  + Know how to pass an entire array or a single element in the array to functions
  + Understand when to use ***const*** qualifier on an array in a parameter list
* Know how to perform a linear search for a key value
* Understand the **concept** of how the binary search works
* Know when one search is better over the other
* Be able to explain code, trace code, and write code snippets with arrays

**Chapter 7 – C Pointers**

* Understand what a pointer variable is
  + How pointers are different from other variables
  + What is the purpose of pointers
* Know how to create a pointer and initialize it
* Know how to use the pointer operators **&** and ***\****
  + Know what each operator does
  + Know when do you use one over the other
  + Understand code that contains these operators
* Understand and know how to ***pass pointers to functions***
  + Understand what ***pass by reference*** means and the impact when passing a pointer to a function
  + Understand and know what needs to be in argument and parameter lists when passing pointers
* Understand the 4 different cases of using the **const** qualifier with pointers
  + Non-constant pointer to non-constant data
  + Constant pointer to non-constant data
  + Non-constant pointer to constant data
  + Constant pointer to constant data
* Know how to use the ***sizeof*** operator
* Know how to use pointers in expressions and how to perform pointer arithmetic
  + Comparing pointers
  + Incrementing and decrementing pointers
  + Adding and subtracting values from pointers
  + Subtracting one pointer from anther
* Understand what a void pointer is
* Understand the relationship between arrays and pointers
* Understand how to create, access, manipulate an array of pointers
* Be able to explain code, trace code, and write code snippets with pointers

The following is the material covered after the midterm. The final will be focused on the following material; however, you need to understand the material from chapters 1 - 7 since it is the foundation for chapters 8 - 12.

**Chapter 8 – C Characters and Strings**

* Know the difference between characters and strings
* Know how to declare and initialize strings
  + Know that strings are terminated with null character
  + Know that an array needs to be big enough to store string and null character
  + Know what happens when there is not enough room for null character
* If I ask you to use any of the functions, I will give you the prototypes but look over Lecture #22 notes and understand how to use these and the issues that can occur:
  + fgets
  + strcpy and strncpy
  + strcat and strncat
  + strcmp and strncmp

**Chapter 9 – C Formatted Input/Output**

* Read Lecture #23 notes and book section called “Printing Strings and Characters”
  + Know when to use **%c** or **%s**
* Know what field width and precision specifiers do when included in printf

**Chapter 10 – C Structures, Unions, Bit Manipulation and Enumerations**

* Understand how a structure is different from an array
* Know how to **define** a structure
* Know how to **declare** and initialize variables of the structure type
* Know how to access members within a structure
  + Know difference between the dot and arrow operators
* Understand these two notations
  + employeePtr->firstName
    - (\*employeePtr).firstName
* Understand initialization of structures
  + What is in structure when you don’t initialize the structure
  + Understand initializer lists (i.e. how they work, what if fewer items than structure members, etc.)
* How to manipulate members within a structure (i.e. display members, things like assignment #11, etc.)
* Know how to declare and create more complex data structures
  + Structure with an array as a member
  + Structure containing another structure
  + Array of structures
* Know how to ***pass structures to functions***
  + Understand default is ***pass by value***
  + Know how to ***pass by reference*** with a structure
  + Know how to pass an **entire structure** or **individual members in the structure** to functions
  + Know what happens when pass array of structures to a function or passing structure that contains an array
* Understand the difference between structures and unions
* Understand the difference between the bitwise operators and the logical operators
* Know how to use bitwise AND - bitwise OR operators
* Know that the dot operator has a higher precedence than address operator

**Chapter 11 – C File Processing (11.1 – 11.4 – Sequential Files Only)**

* Know how to create and use a file pointer
* Understand the following functions when dealing with files
  + fopen
  + fgets
  + fscanf
  + fprintf
  + feof
  + fclose
* Be able to explain code, trace code, and write code snippets dealing with files

**Chapter 12 – C Data Structures**

* Understand the different between **static** data structures and **dynamic** data structures
* Understand how to create a node that will be used in a linked list – self-referential structure
* Understand dynamic allocation and deallocation of memory for a node
* Know the strengths and weaknesses of arrays vs linked lists
* Understand how to connect nodes together when creating a linked list
* Know the differences between linked lists, stacks, queues
* Be able to explain and trace code snippets with a linked list.

**Study tips**

* Review lectures notes
* Review programming examples in lecture notes
* Review homework assignments (assignment #7 and #11 are good ones!)
* Go over examples in the chapters - see if you understand the code or can write the code
* Do the easier exercises in book
* Practice solving problems on paper
* Practice writing little code snippets on paper
* Get enough sleep and drink water - your brain needs water to work properly!

Thank You for an Awesome Semester!

You Made It!

