C. Warren Hammock

CS520

Homework 1

[COMPLETE]

1. (10 pts) The following relation represents a sequence that can be computed algorithmically. Write out each summation, and then, prove by induction, prove:

Show 1 as base case is true:

Assume that *i* == m and n == x therefore prove that m + 1 is true.

End objective:

Replace all x with x + 1

Which is simplified to:

Proof:

2

Add (x + 1)2 to both sides of the equation.

+ (*x + 1)2*

+ (*x + 1)2*

Get common denominators.

+ 6/6(x + 1)2

Add the two.

Attempt to factor out (x + 1).

Add like terms.

Multiplication can be done in any order.

[COMPLETE]

2. (20 pts) Develop two programs, one using an iterative and another using a recursive algorithm that raise x to the n-th power, where both. Execute the two programs with a variety of input sizes from small to , x n ∈ N\* large but make sure you use the same set on both so you can compare them. Record the timing of execution 1 on your computer and plot the results of your experiments. You must provide: (a) the code listings side-by-side, (b) plots and tables of the raw timing data. Explain your findings and describe the growth function that best characterizes the asymptotic complexity (Big-O) of the programs.

Program and testing is located in \Homework1\Question2

Tested Time in milliseconds. I updated the code for the graphs after the output.

The recursive function was much easier to graph than the iterative. I seem like the timing would fluctuate inconsistently at the beginning. I ran several tests and the fault was never consistent. Through the class, I have learned that asymptotic complexity is not worried about the small data, so I am not concerned with evaluating n <= 10. Once it got past the n <= 10 calculations, the growth remained a consistent O(n). Both graphs were consistent with their growth. The asymptotic complexity for both are O(n) due to the nature of n’s growth being constant. As n grows by a constant z calculation, so to does the timing of the function. The timing growth of the function is relatively constant in relation to the value of n’s growth.

[COMPLETE]

3. (10 pts) Insert 10, 5, 15, 10, 9, 7, 2, 1, 4, 3, 8 (as read) into an originally empty binary search tree (BST). Once the BST is populated, perform a​ delete​ (5) operation. To receive credit, you must show both how the tree looks after each insertion and after the delete(5) operation.

Program and testing is located in \Homework1\Question3

I discarded the second of the number 1, so there were no repeat numbers.

[COMPLETE]

4. (20 pts) Insert 18, 10, 14, 9, 2, 7, 11, 12 (as read) into an empty min-heap. Show how the heap looks after each insertion. Once the min-heap is constructed, perform a ​delete-top operation.

Program and testing is located in \Homework1\Question4

[COMPLETE]

5. Consider a binary tree T.

a.(15 pts). Using pseudocode, write a recursive algorithm that accepts as input a binary tree ​T. One function/procedure of the algorithm, computes the number of leaves in ​T; the height of every node in ​T , and the depth of every node in ​T. Another procedure/function indicates whether ​T​ is a full binary tree (or not).

Struct node:

Int data;

Int height, depth;

Node right, left;

//function simply recursively evaluates each node and children. If the node does not have children,

//the evaluation returns 1. All ones are added up.

def CaculateLeaves(T, Node)

if node = null,

return 0

if node right == null and left == null, return 1

return CacluateLeaves(T, Node.left) + CalculateLeaves(T, Node.right)

//puts the max depth of the tree into an interger variable

int maxDepth = def CalcMaxDepth(T, Node){

if node == null

return -1

leftSide = function(T, node.left)

rightSide = function(T, node.right)

if leftSide > rightSide

return leftSide + 1

else return rightSide + 1

<!—

I tried to implement the height check and the depth check together

Instead of an evaluation for each node in the tree, I attempted to speed up

The process by having the evaluation done for each recursive evaluation.

--!>

def CaculateDepthHeightValue(T, Node node, initialDepth, maxDepth)

if node == null

return 0

node.depth = initialDepth;

node.height = maxDepth – node.depth;

CalculateDepthValue(T, node, initialDepth + 1)

CalculateDepthValue(T, node, initialDepth + 1)

//function recursively goes thru each node and prints data for nodes. The

//printing is processed from the max depth to root.

def PrintNodeValues(T, Node node)

if node == null

return;

PrintNodeValues(T, node.left);

PrintNodeValues(T, node.right);

print “node data: “ + node.data;

print “node height: “ + node.height;

print “node depth: “ + node.depth;

//function evaluates each parent node for two children. If the parent has one child with

//value and the other child null, it returns false. Otherwise it returns true.

def bool IsFullBinaryTree(T, Node node)

if node == null

return true;

if (node.left != null && node.right == null) || (node.right != null && node.left == null)

return false;

bool leftNode = isFullBinaryTree(T, node.left);

bool rightNode = isFullBinaryTree(T, node.right);

if (leftNode == true && rightNode == true) return true;

else return false;

def DriverFunction(T, Node root)

int numOfLeaves = CalculateLeaves(T, root); O(n)

bool isTComplete = isFullBinaryTree(T, root); O(n)

CalculateDepthValue(T, Node root, 0, globalHeight); O(2n)

PrintNodeValues(T, Node root); O(n)

bool answer = IsFullBinaryTree(T, Node root); O(n)

print “Is the tree complete: “ + answer; O(1)

print “Number of leaves: “ + numOfLeaves; O(1)

[COMPLETE]

b.(5 pts) What is the time complexity of your algorithm for Problem 5(a)? To receive credit, you must justify your answer.

I produced a general complexity for each function. I did not show a line by line analysis of the complexity but provided an overall complexity for each function. The function that finds the As shown in the complexity beside my function calls, the overall complexity would be O(6n), which would still be O(n) due to O(1n) and O(1n + 1n) having the same growth rate. O(6n) has the same asymptotic complexity as O(n).

[COMPLETE]

6. (10 pts) Given an n x n, 2-dimensional array, what is the worst time complexity to compute the sum of all elements? Assume you have a “good” algorithm. To receive credit, you must justify your answer.

Assuming we have a “good” algorithm, it will take n \* n to complete a total addition of a 2-dimensional array. Thus, the direct answer is O(n2). It easier to view it as n\*m array. For every calculation of n, you would have to calculate all m’s belonging to that n. The complexity would be O(nm), which would be the same for this problem as O(n2) or O(n\*n). If the multidimensional array of 81 elements became a single array of 81 elements, the complexity would change to O(n). Interesting how the same blocks of data can represent two, very different, asymptotic complexity.

[COMPLETE]

7. (10 pts)Given the following function pseudocode:

float unknown (int n) {

if (n <=1) return (1.0);

else return(unknown(n-1) + unknown(n-2));

}

a. What does the above function compute? This is the dressed down, Fibonacci sequence function. A must for lesson teaching recursive functions. Sum of the two highest number equals the next highest number. I believe the actual creator of the algorithm created it to solve a problem to address livestock repopulation and dates. I think.

b. Define a generalized recurrence relation for the function.

c. How many additions are performed to compute unknown(6)?

Base condition: n <= 1

Return (unknown(6-1) 5 + unknown(6-2) 4)

(5-1)4 + (5-2)3 (4-1)3 + (4-2)2

(4-1)3 + (4-2)2 | (3-1)2 + (3-2)1 (3-1)2 + (3-2)1 | (2-1)1 + (2-2)0

(3-1)2 + (3-2)1 | (2-1)1 + (2-2)0 | (2-1)1+(2-2)0 (2-1)1 + (2-2)0

(2-1)1 + (2-2)0

Answer: 20 additions (count all 0/1 a return of + 1 and count all recursive return additions)