**Name: Patrick O’Brien**

For each of the below questions, write a short sentence or two to express (in your own words) your answer. Keep the answers short, but use complete, correct, English sentences.

If it helps to clarify the questions, feel free to mentally prefix all the questions with the phrase "According to the video…"

1. After you’ve watched all the videos, please answer this question:  
   Of all the videos that you watched, if you could pick one video to be re-recorded by the instructor outside of class which would you choose? Why?  
   (Keep in mind the recording outside of class will omit any pauses from the instructor answering student questions, have less hemming and hawing, etc, and generally be more concise)

|  |
| --- |
| The videos all have an extremely annoying click sound that distracts from the content. |

**VIDEO: BST: Basics, Find, Add Overviews**

1. Intuitively, how is a linked list set up?

|  |
| --- |
| Intuitively, a linked list is set up: A chain of objects that end with null. There is a linked list object that sits in front. Each object in the chain is represented by a node. |

1. What's the running time (and space) of finding something in a linked list (in Big Oh notation) **using a loop**?

|  |
| --- |
| The running time of finding something in a linked list using a loop is O(N). As you are iterating through the linked list in a linear fashion – moving between the different nodes. |

1. What's the running time (and space) of finding something in a linked list (in Big Oh notation) **using a recursive approach**?

|  |
| --- |
| Recursively running through a linked list will result in a running time of O(N) |

1. For pretty much anything you want to do to an entire list (printing, finding a value, etc) what will the running be (in Big Oh notation)?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linked list, it will always be O(N)   |  |  | | --- | --- | | Loop | O(n) – O(1) on space | | Recursion | O(n) – O(N) on space | |

1. How is a Binary Search Tree (aka BST) set up? At each node, what will all the nodes to the left of have in common? What will the nodes to the right all have in common?

|  |
| --- |
| **Binary search tree setup:**  Node based binary tree data structure:  Left subtree of a node contains only nodes with keys lesser than the node’s key  The right subtree of a node contains only nodes with keys greater than the node’s key  The left and right subtree each must also be a binary search tree  There must be no duplicate nodes. |

1. In a linked list we had a **front** reference. What will the BST call the reference to the first node?

|  |
| --- |
| The top node reference |

1. Explain why the value 15 was added where it was to the tree that contained {20, 10, 50}

|  |
| --- |
| In a tree that contains {20, 10, and 50} 15 will be added in the left node with 10. |

1. Is there any restriction on the things that you can put into a BST?

|  |
| --- |
| Anything that you can use comparative logic on can be stored in a BST. |

1. Which nodes were examined while searching for the value 12 in the tree that contained (20, 10, 50, 15)? List them in the order that they were examined.

|  |
| --- |
| The nodes that were examined when searching for 12 in {20,10,50,15} are:  Is there a top nodes – clearly does not exist  Look to the left because 12 < 20 Continue down the tree to 15 because there is a node there and it is greater than 10  Return since other two addresses are null, no more numbers |

1. How did you know that 12 was NOT present in the tree?

|  |
| --- |
| Other two addresses in 15 are null, and 12 can only be to the left address. |

1. Describe, in your own words, how to find a value in a binary search tree.  
   (This is demonstrated in the video multiple times, but isn't really summarized)

|  |
| --- |
| You input the value that you are looking for that can be analyzed with comparative logic.  You travel to the first node if it is present, then if the value is smaller than that node, you continue down the tree to the left, if it is larger, you continue down the tree to the right.  Nodes added to the tree aren’t sorted, so it will continue to search the tree, going larger, then smaller, larger, or smaller (right or left) until it finds the correct value. If it does not find the correct value it will return. |

1. In the C# source code, what stops us from looking to the left of the node containing the value 12?  
   (This is described verbally before it's defined in on-screen C# code)

|  |
| --- |
| What stops us from going to the left of the node once we reach 12.  Mechanically: Because the value is null, and there is a check for that in the code. |

1. List out the C# source code that defines a BST node here:

|  |
| --- |
| Code that defines a BST node:  Class BST  {  BSTNode top;  Private class BSTNode  {  Public int key;  Public int data;  Public BSTNode left;  Public BSTNode right;  } |

1. When searching the BST, how many nodes will be ignored each time the algorithm takes a step down the tree? (Assume that the BST is a 'best case' BST)

|  |
| --- |
| Each step down the tree, one node will be ignored. It can either go left or it can go right. |

1. What will the tree look like in the **best case** for a BST?

|  |
| --- |
| All nodes are in the root’s right subtree, with the one to be inserted belonging to the left. |

1. What will the tree look like in the **worst case** for a BST?

|  |
| --- |
| An unbalanced binary search tree with sorted data, it forms a linked list-type structure and is equivalent to a singly linked list.  Unbalanced_BST |

1. What is the running time (in Big Oh notation) of the **best case** of the BST?

|  |
| --- |
| O(1)  Or  O(log\_2N) depending on the situation |

1. What is the running time (in Big Oh notation) of the **worst case** of the BST?

|  |
| --- |
| O(N) |

1. Are BSTs ever used in production code? If not, what sort of trees are used instead (and why are these other trees used)?

|  |
| --- |
| Yes they are used in production code: |

**VIDEO: BST Basic Class Definitions**

1. What will the BST (Binary Search Tree) object manage?

|  |
| --- |
| The BST object manages: a list of individual nodes, each one has the left and right and the storage for the object. |

1. What is the name of the one reference that a BST will have?  
   (This is for a BST, NOT a BSTNode)

|  |
| --- |
| A reference to the top node in the tree. (top) |

1. What are the three fields that are present within a Binary Search Tree Node (a BSTNode) object?

|  |
| --- |
| The data is stored at the top, then there is a left reference and a right reference. |

**VIDEO: BigOh Applied To Space & Time**

1. Summarize how the Print() method is implemented in 2-3 concise yet clear sentences

|  |
| --- |
| First refers to a linked list, say there are a total of N items in the list. While first has not advanced to the end of the list, call the print method then move down to the next node. |

1. What is the running time (in Big Oh notation) of the Print() method?

|  |
| --- |
| The running time for the Print() method is O(N) |

1. List some possible functions that might describe how much space an algorithm might use:

|  |
| --- |
| Space is evaluated in the same way that time is.  O(1) | O(N), etc 🡪 running space |

1. When figuring out how much space the Print() method requires, do we count the space occupied by the nodes in the list?

|  |
| --- |
| We do not count the variables in the list, just the variables that are directly allocated, or cause direct allocation to the list. |

1. How much space does the Print() method require (using the Big Oh notation)? Make sure that you're comfortable both with how you'd write this, and how you'd talk about this in English.

|  |
| --- |
| A constant, unchanging amount of space. O(1) |

1. How can we know that the private version of PrintR() will never be given a null reference as a parameter?

|  |
| --- |
| PrintR using recursion. Takes a non-null reference to a node. We know that PrintR will never be given a null reference because it is only accessible by internal code, so it can be assumed that you would only ever give it non-null values. |

1. What is the running time (in Big Oh notation) of the Print**R**() method?

|  |
| --- |
| O(n) |

1. The number of times that the private PrintR is going to call itself is directly proportional to what?

|  |
| --- |
| Directly proportional to the number of nodes in the list |

1. How much space does the PrintR() method require (using the Big Oh notation)?

|  |
| --- |
| O(N) – constant amount of space per stack frame. |

**VIDEO: BST: Implementing Add**

1. When a new BST object is created, what is the value of it's **top** field?

|  |
| --- |
| The value of the top field is: null |

1. If top is null, then how many nodes are in the tree?

|  |
| --- |
| If top is null, it means that everything else in the tree does not exist, 0 nodes. |

1. When a new BSTNode object is created, what is the value of it's **left** and **right** fields?

|  |
| --- |
| The value of BSTNode’s left and right fields when it is first created are: null |

1. While walking down the tree, what comparison are you going to do at each node in the tree?

|  |
| --- |
| When walking down the tree, the comparison you make at each node is: You check to see if the value is null, then 🡪 decision to go left or right if (key < cur.key)  {  If (cur.left == null)  {    }  } |

1. Under what circumstance will you add the new node to the left of the current node?

|  |
| --- |
| If the key is less than the current key, and the value of the left object is equal to null |

1. Under what circumstance will you **move the cur reference** to the left of the current node?

|  |
| --- |
| You will move the cur reference to the left of the current node if: the left value is not null and a value is already stored there. |

1. The code that was written in the video, what is the third possible case that the function needs to handle?

|  |
| --- |
| The third possible for a BST – the key that you are trying to add is already in the tree. |

1. For the normal/basic strategy for dealing with duplicates, what should be done?

|  |
| --- |
| If all the other comparisons are not met, then the value is likely a duplicate, which are not allowed in the tree. |

1. If you did want to add duplicate values to the tree, what off-the-cuff, not-at-all-tested solution does the teacher speculate might work?  
   (And what other solution does the teacher demonstrate does NOT work?)

|  |
| --- |
| If you were to splice it inside the tree (creating a mini-linked list within the tree). |

**VIDEO: BST: Implementing Find, Using A Loop**

1. When looking for a value in a BST, what's the first thing that should be checked for?

|  |
| --- |
| The first thing that should be checked for is:  Is the target value is in the tree, is the tree null? |

1. At the start of the loop, what node does **cur** refer to?

|  |
| --- |
| Cur refers to the top (or root) of the BST. |

1. Within the loop, what's the first thing we check for?

|  |
| --- |
| The first thing that you check for within the loop is:  Whether or not the target value is the current key value. |

1. If the function leaves the loop, what does the function do (and why)?

|  |
| --- |
| If the function leaves the loop, it: it returns false, because we checked to see if the value was null, and it was, so therefore for the value is not in the tree. |

1. Why do we NOT need to check for cur.left (or cur.right) being null inside the loop?

|  |
| --- |
| We do not need to check for cur.left or cur.right being null inside the loop because: we only need to check if the target value is equal to the current key – if it is, return true because you have found the target. Otherwise if it is greater, then we go to the right, otherwise go to the left. Then the loop restarts, and we check to see if the current value is null. Because we check at the start of the loop we do not need to individually check for left and right. |

**VIDEO: BST: Implementing Add, Recursively**

1. Why did the teacher decide to first assign the "find a value in a BST iteratively" exercise before this one? What general strategy is sometimes helpful in writing recursive logic?

|  |
| --- |
| Write out a loop based version first to see how the logic flows, then write the recursion after using the earlier logic sketch with the loop. |

1. What's the first thing that the public FindR method should do?

|  |
| --- |
| The first thing that the public FindR method should do is: Check to see if top/root is equal to null. |

1. In the BSTNode.FindR method, what are the three basic possibilities that might occur?

|  |
| --- |
| 1. tg < this.key  2.tg > this.key  3. tg == this.key |

(There's a fair amount time spent responding to a question, which implements the recursive FindR on the BST class, instead of the BSTNode class)

1. Within the FindRINternal, if the value we're looking for is less than the current node's key, then what are the two possible actions we might take (and under what circumstances do we do each one)?

|  |
| --- |
| The other two possible actions we might take   1. Cur.left == null   Circumstance: The value is not there   1. Pass the value back through with recursion FindRInternal(tg, cur.left);   Circumstance: The left value is present |

(There's an in-depth demonstration of how FindRINternal works, then we go back to FindR\_Node)

1. In FindR\_Node, where does the "this" field/parameter come from?

|  |
| --- |
| In the FindR\_Node, the “this” param comes from: the top node |

1. How is the C# source code of FindR\_Node different from that of FindRINternal?

|  |
| --- |
| The FindR\_Node does not take t he BSTNode cur, while FindRInternal does |

**VIDEO: BST: Print**

1. Prior to this point, what technique have we used for all the BST methods?

|  |
| --- |
| All prior BST methods have used the iterative (loop with iteration instead of recursion) |

1. Can you (easily) traverse every node in the tree using a loop?  
   What abstract data type will you need to use?

|  |
| --- |
| Can you easily traverse every single node in the entire tree – cannot use the loop by itself, have to use a stack somehow. Or a stack class  Abstract data type = stack implementation using recursion |

1. What does the externally available, public version of the print method do?  
   (I.e., what responsibilities does it have?)

|  |
| --- |
| The externally available print method – handles external responsibilities and then calls a private method to perform the recursion |

1. Why can't we make the "internal", private version of Print public?

|  |
| --- |
| We can’t make the internal private version of print public because: because the private recursive reference does not have a null check for top, which is handled by the public Print method. If the private version of print was public, a null top could be used and it would crash the program. ALSO: the parameter type is actually private so the method has to be private to take a private param. |

1. In terms of the "onion model of recursion", what is the work that we're going to do at each level of recursion?

|  |
| --- |
| Onion model recursion: work done at each level of recursion is:  Console.WriteLine is the work that we’re going, recursively print left and right subtree. |

1. In terms of the "onion model of recursion", what are the two recursive aspects of the Print algorithm?

|  |
| --- |
| 1. printing the data in the node  2. Moving through the tree |

1. In the PrintR method, what is the significance of cur.Left not being null?  
   (Note that the video has the PrintR method mistakenly calling Print – PrintR should be recursively calling Print**R** instead)

|  |
| --- |
| The significance of cur.Left not being null means that the value is less than 10, and could be anywhere in the left tree. It could be the very next node, or any of a million. The tree is not empty. |

1. In the PrintR method, what is the significance of cur.Right not being null?

|  |
| --- |
| If cur.Right is not null, then it means that there is a number that is larger than 10 in that tree, and that the tree is not empty. |

1. In PrintR, why does the Console.WriteLine command (abbreviated at C.WL) go in between the first and second recursive calls?

|  |
| --- |
| The Console.WriteLine command goes between the first and second recursive calls because: It needs to print out the value of Cur.Data for the first recursive check if the value is true. If the value is not true, it will skip the line and print the value from the second previous check where then right may have been true, before checking right once again for the next data. |

**VIDEO: BST: Patterns**

1. What are the three traversal patterns listed on the screen at the start of this video?

|  |
| --- |
| 1. In-order transversal 2. Pre-order transversal 3. Post-order transversal |

1. What order will the "in-order traversal" traverse all the nodes?

|  |
| --- |
| In order of left to right after working on the current node. (smallest to largest) |

1. How would you traverse all the nodes in the reverse order (by "reverse", I mean the reverse of the order listed in the prior question)

|  |
| --- |
| You would swap the order of comparisons so that your process right first, and then left. (larger to smaller) |

1. What is a good mnemonic (memory-aid/trick) to remember where to put the "do work" part of the in-order traversal?

|  |
| --- |
| In order is RDR, Pre-order is before(pre), post-order goes at the end  RDR, DRR, RRD |

1. What would be one good usage of the "pre-order traversal" pattern?  
   Why is the pre-order traversal good for this task?

|  |
| --- |
| A good use case for the pre-order transversal pattern: Because the work is a find() implementation  Good for the task because: It will process the work done first, if you find the thing at the top of the tree, you’ve done the work, and you’re good to go. |

1. Given the usage you listed in prior question (for pre-order traversal), why would the post-order traversal be a particularly bad choice for that same usage?

|  |
| --- |
| Because post order explores both the right and the left nodes before doing any work. So you would process and explore the entire tree before checking each node and doing the work. |

1. How would you find the smallest value in the tree?  
   (This is demonstrated in response to a student's question, and it isn't really mentioned what's going on until near the end of the demonstration)

|  |
| --- |
| You find the smallest value in the tree by: Following the tree that stores the smallest values (left). Then you follow it through the left value until the left value is null, which means you have reached the smallest number in that tree. |

**VIDEO: Recursive Add**

1. What is the first thing to check for when adding a new value to the BST?  
   If this is true, what do you do next?

|  |
| --- |
| The first thing to do when adding a new value to the BST is check:  If there is no root node (null)  Create a new BSTNode with that value and return |

1. For this implementation, which class is the public AddR method located on? Where is the private, recursive AddR method located?

|  |
| --- |
| The public AddR method is located on the BST class, while the private version is located on the BSTNode class. |

1. What are the two basic options (in the recursive AddR method)?

|  |
| --- |
| The two basic options for the recursive AddR method are:  1. If there is no node, create a node there and fill it in with the value  2. Want to add a value (2), you will say that there is a node there, so you recursively call the AddR value and recursively add the value down. |

1. In terms of the "onion model of recursion", what is the small amount of work that can be done immediately? How is the remaining problem a smaller version of the overall problem (of adding a new value to the tree)?

|  |
| --- |
| The small amount of work that can be done immediately: the particular case, add a value to the tree, determine if you should add it to the parent or child node. If not, then you shave off a little bit of work by knowing that the current node is not the place to add the value.  How is the remaining a smaller version of the overall problem?: There are less nodes to recursively call |

1. How do we know if we need to add the value to the left of the current node?  
   Explain this in English AND provide a snippet of C# source code.

|  |
| --- |
| Public void AddR(int val)  {  If (this.left == null && val < this.data)  {  We know to add the value to the left of the current code if the value for is equal to null and the current val is less than the value currently in the node. |

1. If you know that you need to add the node to the left, what are the two cases that might be true?  
   Explain this in English AND provide a snippet of C# source code.

|  |
| --- |
| 1. We know to add the value to the left of the current code if the value for is equal to null and the current val is less than the value currently in the node. 2. Or if the current value is less than the data and the left is not empty  if (val < this.data)   {  If (this.left == null)  {  This.left = new  BSTNode(val);  Return;  }  Else  {  This.left.AddR(val);  Return;  } |

1. How can you check for duplicate values (i.e., that the value you want to add is already in the tree)?

|  |
| --- |
| You check to see if val == data |

1. What is the one thing you should NOT do if your BST finds a duplicate value in the tree?

|  |
| --- |
| One thing that you should not do if your BST finds a duplicate value in the tree:  Printing a message to the console |