COMP 241Review

Topics:

- For this exam, previous concepts like Java programming, abstract data types, interface vs implementation, and especially big O analysis remain important. You should not forget about arrays, linkedLists, stacks, and queues, but that will not be a focus of Exam 2.
- Trees
    - Binary Search Trees (BSTs)
        - Functions: insertion, deletion, search, etc.
        - Implementation
    - Traversals (preorder, postorder, inorder)
    - Big O running time
- Hashtables
    - The hashing function and how it works
    - Implementation
    - Collision strategies (chaining, linear probing, quadratic probing)
    - Big O running time
- Sets
    - Functions
    - Implementation
    - Big O running time
- Maps
    - Functions
    - Implementation
    - Big O running time
- Sorting algorithms (mergesort and quicksort)
- Big O analysis

1. In a binary tree (not a binary search tree), ints have been stored. Write a function to search the list for the number 14 and return true if it is present and false if it is not present. In the binary tree class, you have access to a Node named root; in the Node class you have access to int data, Node left, and Node right.

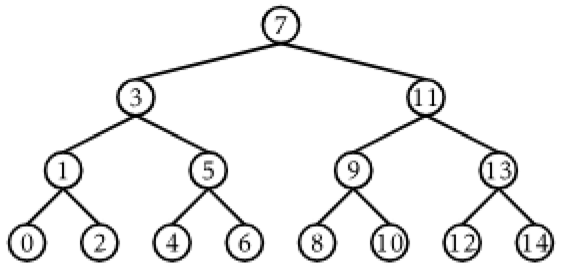
```
//This assumes that the value initially passed in is the root of the tree.

public static bool searchFor14(Node root)
{
        if(root != null)
        {
                If(root.data == 14)
                        return true;
                else if(searchFor14(root.left))
                        return true;
                else
                        return searchFor14(root.right);
        }
        else
        {
                return false;
        }
}
```

What is the big O time for this function?

O(n)

2. Given this tree, what would the following code print out if this function was called? (The top node is passed in as root.)



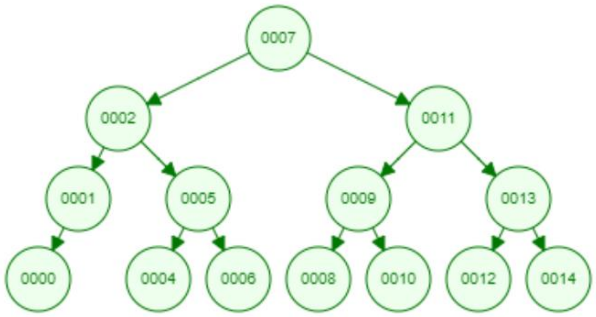
```
public static void printValues(Node root)
{
        if(root != null)
        {
                System.out.println(root.value);
                printValues(root.left);
                printValues(root.right);
                System.out.println(root.value);
        }
}
```

With each value on a new line:

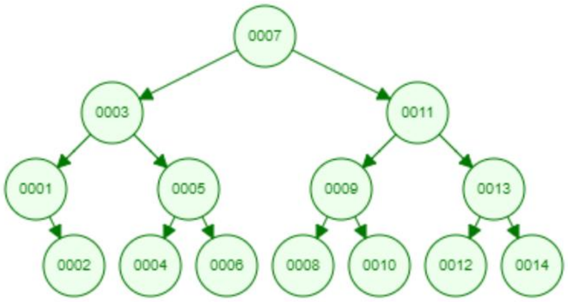
7 3 1 0 0 2 2 1 5 4 4 6 6 5 3 11 9 8 8 10 10 9 13 12 12 14 14 13 11 7

Note that the highlighted section would be needed to actually make the above code work!

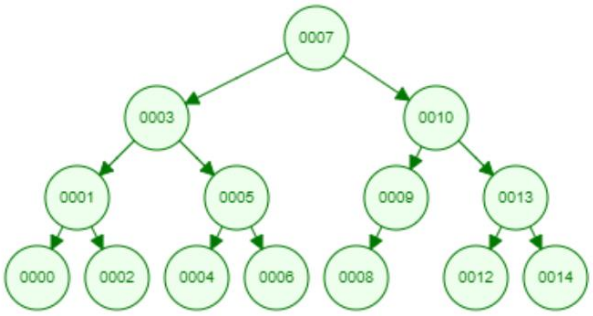
3. Resetting after each line, demonstrate how the above tree would look after:
   - -4 was added, 16 was added, -10 was added, -12 was added, and -6 was added.
     - **16 is added as the right child of 14, and the other numbers form a subtree to the left of 0. -4 at the root, with -10 to its right, and -12 as the next one on the right.**
   - 3 was deleted.



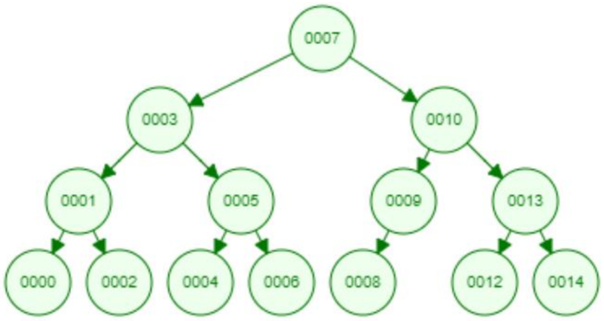
- 0 was deleted.



- 11 was deleted.



- 7 was deleted.



4. For binary search trees, is there any way to improve or reduce the performance (for example, in searching for a specific element)?
   The structure of the tree can affect run time.

   What would the best case be for searching a binary tree, and what would the big O run time be for that?
   A completely balanced tree – O(log(n))

   What would the worst case be, and what would the big O time be for that?
   A completely unbalanced tree (ex, where each node only has a single child) – O(n)

5. Suppose you are tasked with designing a hash function for a Pet object created by a small company. The Pet object has variables containing pet name, pet species, breed, age, weight, and owner. If you were tasked with creating a hash function to give a unique identifier to each animal, how would you approach that task? Describe what you would do, and then write a function that does that.

One approach might be to use the owner's name and the pet's name to create a hash. These variables are helpful because they don't generally change and don't generally overlap for different pets. Especially there may be an overlap in one of the variables (ex, one owner has multiple pets or there are several pets with the same name) but very rarely both. One way you could do this is by using the first six values in the owner's name as ints, followed by the first six values in the pet's name (as ints) and multiply each by a value for its location.

```
public static int hash(Pet input)
{
        int result = 0;
        int locationValue = 12;
        int i =0;
        int j = input.petName.length;


        while(i  < input.ownerName.length && i < 6)
        {
                result += ((int) input.ownerName.getCharAt(i)) *locationValue;
                locationValue--;
                i++;
        }
        locationValue = 6;
        while(j  < input.petName.length && j < 6)
        {
                result += ((int) input.petName.getCharAt(j)) *locationValue;
                locationValue--;
                j++;
        }
        return result;

}
```

What is the big O time for this?

Constant!

6. Given an array of elements, write a function that would apply the mergesort algorithm to those elements (and, thus, sort the list).

See the code posted on the class website!

Show the mergesort algorithm sort the array [11, 4, 7, 2, 88, 5, 6]. (This should work the same way as your code does from the first part of the problem.)

[11, 4, 7] [2, 88, 5, 6]

[11][4, 7][2, 88][5, 6]

[11][4][7][2][88][5][6] Now each element has been sorted and we start combining arrays!

We do this by comparing up to n times to figure out where in the sorted list we should insert

[4,11] [7] [2,88] [5,6]

[4, 7, 11] [2, 5, 6, 88]

[2,4,5,6,7,11,88]

What is the big O running time for your code?

O(log(n))

7. You are developing code for a Map implemented with a hashtable and need to write a function to delete an element that's in a map. If the element is deleted, return true and, if it isn't, return false. (Assume the hash function has been provided and you can access it simply by calling hash() on the object you're interested in hashing. You are not required to take into account collision strategies when writing this function.)

```
Public class RHashMap<E>
{
        private E[] data;//Where the hash table is stored.
        //Other code necessary for the program to run is included but is not relevant here.
```

```
    public Boolean delete(E input)
    {
        int index = input.hash();


        int finalIndex = index % data.length;


        if(data[finalIndex] != null)
        {
            data[finalIndex] = null;
            return true;
        }
        return false;

    }
}
```

What is the big O running time of the deletion?

Constant!