Jacob Alspaw
jaa134
HW2
2/19/2015

1) This is illogical. Because f(n) = O(g(n)), then the function f(n)'s upper bound is g(n). And because t(n) = O(g(n)), the function t(n)'s upper bound is g(n). The totality of information we have tells us the upper bounds of functions f(n) and g(n). This little of information cannot be used to denote f(n) = Ω(t(n)), where f(n)'s lower bound is t(n). There is no logical connection between functions f(n) and t(n) to examine if one function remains larger, at all times, than the other.

2)

```
1  /**
2   * A method to reverse a singly linked list.
3   */
4  public void reverse() {
5    Node current = this.getFront(); //the first node in a list
6    Node back = null; //the previous node in a list
7    Node forward = head.getNext(); //the next node in the list
8
9    //if the list is empty or the list is of size null
10   if (current == null || current.getNext() == null) {
11   }
12   else {
13     //loop to switch pointers of nodes in the opposite direction
14     while(forward != null) {
15       current.setNext(back);
16       back = current;
17       current = forward;
18       forward = forward.getNext();
19     }
20     //finishing switching the last node in the list
21     current.setNext(back);
22     //setting the head of the list
23     this.setFront(current);
24   }
25 }
```

3)

```
1  public interface intStack {
2    /** A method that determines if the stack is empty. */
3    public boolean empty();
4
5    /** A method that pops the top item on the stack. */
6    public int pop();
7
8    /** A method that peeks at the top item on a stack. */
9    public int peek();
10
11   /** A method that push an item into the stack. */
12   public int push(int item);
13
14   /** A method that will search for an item in the stack. */
15   public int search(int value);
16
17   /** A method that will evaluate a postfix expression.
18     * @param expr The postfix expression.
19     * @return int The evaluation.
20     */
21   public static int postfixExpr(String expr) {
22     myIntStack stack = new myIntStack();
23     int first = 0;
24     int second = 0;
25     int result = 0;
26     char chr;
27
28     for (int j = 0; j < expr.length(); j++) {
29       chr = expr.charAt(j);
30       if (chr != '+' || chr != '-' || chr != '*' || chr != '/') {
31         stack.push(chr);
32       }
33       else {
34         first = stack.pop() - 48;
35         second = stack.pop() - 48;
36         if (chr == '+') {
37           result = first + second;
38         }
39         else if (chr == '-') {
40           result = first - second;
41         }
42         else if (chr == '*') {
43           result = first * second;
44         }
45         else if(chr == '/') {
46           result = first / second;
```

```
47          }
48          stack.push(result);
49        }
50      }
51      return stack.pop();
52    }
53
```

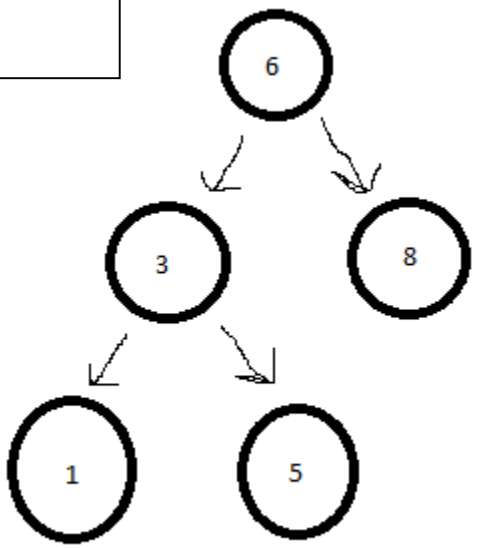
4) A) $2^{h+1} - 1$
   B) 2^h
   C) $2^{h+1} - 1 - 2^h = 2^h - 1$
   D) $n + 1 = 2^{h+1}$ ➔ $\log_2(n+1) - 1 = h$

5)

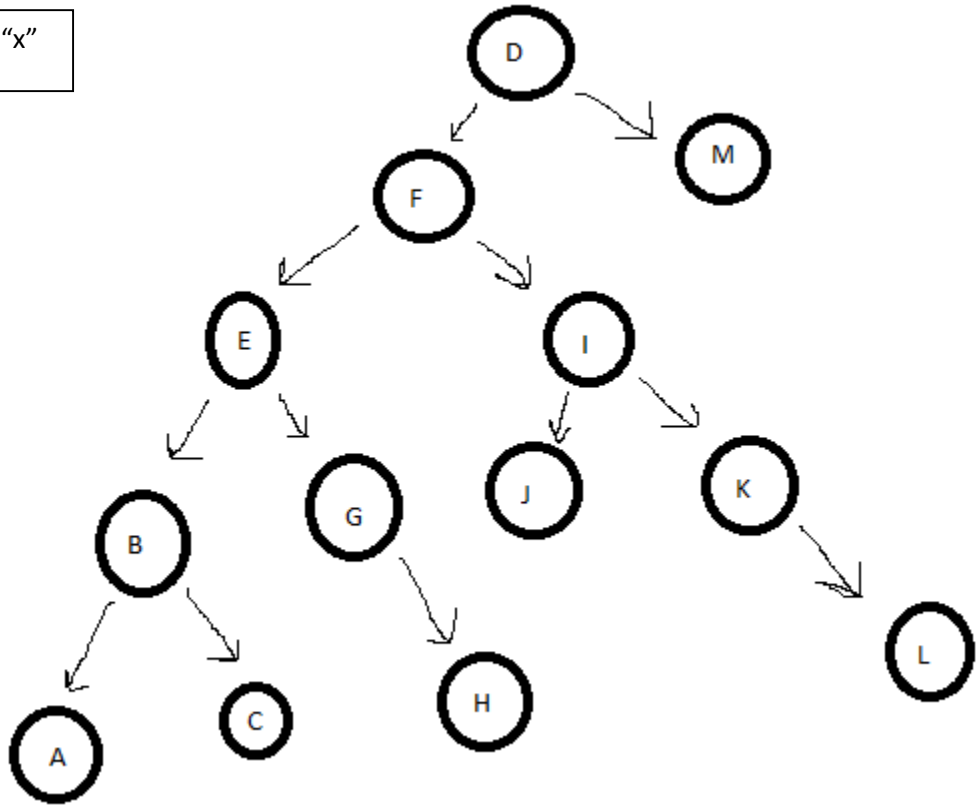
```
1   /**
2    * A method to insert a node into a binary tree.
3    * @param key The key of a node.
4    * @param data The data stored on a node.
5    */
6   public void insert(int key, String data) {
7     //if the parent has no children
8     if(this.left == null && this.right == null) {
9       //if the node is to the left direction
10      if(key < this.key) {
11        //make a new node
12        this.left = new Node(key, data);
13      }
14      //if the node is to the right direction
15      if(key > this.key) {
16        //make a new node
17        this.right = new Node(key, data);
18      }
19    }
20    //if the parent has only a left child
21    if(this.left != null && this.right == null) {
22      //if the node is to the left direction
23      if(key < this.key) {
24        //restart from a new node
25        this.left.insert(key, data);
26      }
27      //if the node is to the right direction
28      if(key > this.key) {
29        //make a new node
30        this.right = new Node(key, data);
31      }
32    }
33    //if the parent has only a right child
34    if(this.left == null && this.right != null) {
35      //if the node is to the left direction
36      if(key < this.key) {
37        //make a new node
38        this.left = new Node(key, data);
39      }
40      //if the node is to the right direction
41      if(key > this.key) {
42        //restart from a new node
43        this.right.insert(key, data);
44      }
45    }
46    //if the parent has two children
```

```
47    if(this.left != null && this.right != null) {
48      //if the node is to the left direction
49      if(key < this.key) {
50        //restart from a new node
51        this.left.insert(key,data);
52      }
53      //if the node is to the left direction
54      if(key > this.key) {
55        //restart from a new node
56        this.right.insert(key, data);
57      }
58    }
59  }
```

6) After inserting the sequence in order.



7a) After deleting “x”



7b) After deleting “E”