Fall 2015

1. B. O(n log(n))
   You need at least *n* steps to iterate through all the elements and for each element that you check, assuming you do a binary search to find its 100-complement to form the desired pair, you will need *log(n)* steps.
2. B. O(n)
   At the first iteration of the outer loop, the inner loop will run *n* times. At the second iteration, it will run $\frac{n}{2}$ times, at the third it will be $\frac{n}{4}$ times, and so on.
   $$n+\frac{n}{2}+\frac{n}{4}+\cdots=n\left(\sum_{i=0}^{k}\frac{1}{2^i}\right)=n\left(\frac{\left(\frac{1}{2}\right)^k-1}{\frac{1}{2}-1}\right)=-2n\left(\left(\frac{1}{2}\right)^{\log_2 n}-1\right)=-2n\left(\frac{1}{n}-1\right)$$
   $$=\ 2n-2=O(n)$$
3. A. *(Not tested on this)*
4. D. *(Not tested on this)*
5. D. 1,3,4,5,2,10,12,9,7
   Note that post order traversal is *left child, right child, root*
6. A. 7 12 10 9 5 4 3 1 2
   The order of calls is weirdPreOrder(7), weirdPostOrder(9), weirdPreOrder(12), weirdPostOrder(10), weirdPostOrder(2), weirdPreOrder(5), weirdPostOrder(4), weirdPostOrder(3), weirdPreOrder(1)
7. D. 1 2 3
   The method will remove and return the element at the bottom of the stack.
8. C.
9. C. O(n)
   $$t(n)=\ t(n-1)+c=t(0)+c*n=O(n)$$
   The recursive call is done on the same stack but with one less element.
10. D.
    Consider the case when the added element is less than the old root's left node, then the produced tree is not a valid binary tree.
11. E.
    Recall that for a BST, a node's left subtree contains only nodes that are less while its right subtree contains only nodes that are greater.
12. A.
    Recall that for a heap, each node is smaller than its children.
13. E.
    It actually never prints anything else than "Zero".
14. A. *(Not tested on this)*

15. C.
    You assume the kth step $f(k) = g(k)$ such that you can prove the k+1 th step
    $f(k+1) = g(k+1)$
16. C. *(Not tested on this)*
17. B.
18. C. *(Not tested on this)*
19. D. *(Not tested on this)*
20. E. None of these are true in general (3. is always false)
21. A. *(Not tested on this)*
22. C.
    Left child index is $3i - 1$, middle child index is $3i$ and right child index is $3i + 1$
23. B. (Note that array size doesn't change since there is same number of elements)
24. D.
25. E. *(Not tested on this)*
26. B. *(Not tested on this)*
27. D. *(Not tested on this)*
28. D.

$$\begin{gathered} T(m,n) = 2T(m-1,n-1) \\ = 2^m T(0,n-m) \\ = 2^m(T(0,n-m-1)+1) \\ = 2^m\big(T(0,0)+(n-m)\big) \\ = 2^m(n-m+1) \end{gathered}$$

29. E.
    The actual order is A, B, E, C, G, D, F
30. A.
    The order is A, B, D, E, F, C, G
31. D.
32. A.
33. C.
34. E.
    The algorithm will multiply the two numbers by adding *a* to itself *b* times $(a + a + \cdots)$
35. E.
    The actual answer should be s[1 : length(s) - 2]
    Recall that length(s)-1 is the last element and not the second to last element
36. C.

$$\begin{gathered} T_6 = T_3 + T_4 + T_5 = (T_0 + T_1 + T_2) + (T_1 + T_2 + T_3) + (T_2 + T_3 + T_4) \\ = (0+1+2) + \big(1+2+(0+1+2)\big) + \Big(2+(0+1+2)+\big(1+2+(0+1+2)\big)\Big) \\ = 3 + 6 + (2 + 3 + 6) = 20 \end{gathered}$$

37. A.
38. C.

39. C.
    Since the merge is not done after the arrays are sorted, the answer might not be correct.
40. B.
    Since there is a third recursive call, the algorithm is not O(n log(n)) anymore.
41. B.
    The array is not partitioned into two equal subsets since mid is set to be equal to start
42. A.
    Note that $\log(n!)$ is smaller than $n\log(n^2)$ since in general $\log(\ )$ is smaller than $n\log(\ )$.
43. C.
    $f(n)$ is $O(n\log(n))$ and $g(n)$ is $O(n)$.
44. E.
45. E.

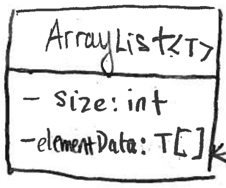
F2012 Final

① a): Notice that there can be two cases when you have to remove element i from an ArrayList.

taken care about by same piece of code!

- $i == (size - 1)$: you don't have to shift anything. Just remove it!
- $i < (Size - 1)$: remove element i and shift everything from $i+1$ to size-1 to the left by one index.

Recall our array list has the following implementation in Java:



Assumption: this private field exists within our ArrayList <T> class!

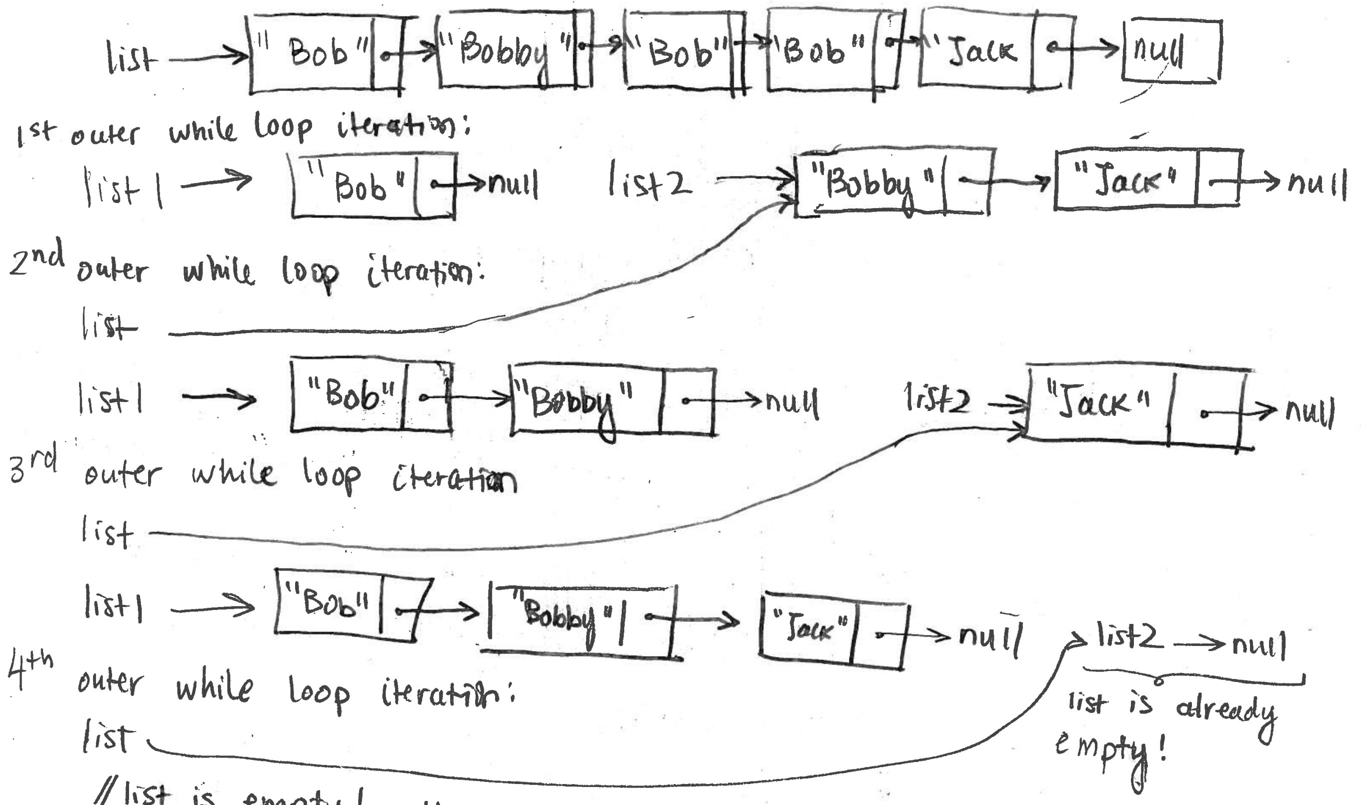
As such (inside the if-statement):

```
tmp = elementData[i]; //obtain element we wanna remove

from (int i=index+1; i< size; i++) {
        elementData[i-1] = elementData[i];
}

size --; //important! tells ArrayList to not consider
            the last element anymore.

return tmp;
```

b): Assume n = original size of our ArrayList. Method remove() i

- $O(n)$ since index could be zero
  Note: it is still $O(n)$ if index is between $[1, size-2]$!
- $\Omega(1)$ since in the best case index could be size-1. Removing the last element takes constant time!

(2) a): Let's run the method with an example:



// list is empty! thus we will simply return list1.

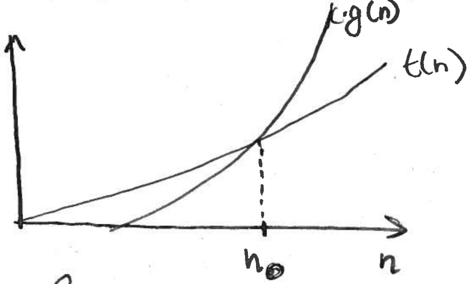
∴ This method returns a new linkedlist of the input linked list with no duplicates!

b): Assuming n: size of input linked list;
We have two nested while loops. In the worst case, all strings are distinct. Thus, we will get a list2 which is of size one less than the linkedlist at the previous outer loop iteration.

Thus, we have
$$O\left(n + (n-1) + \cdots 1\right)$$
$$= O\left(\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2}\right)$$
largest term!
$$= \boxed{O(n^2)}$$

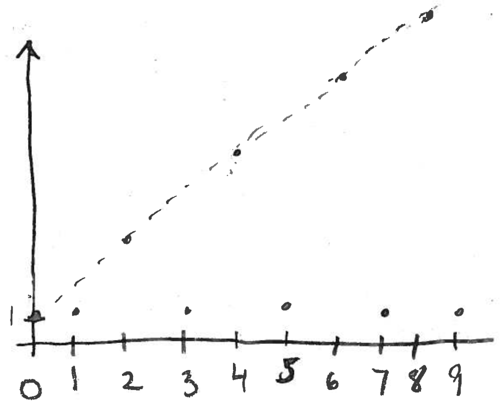
c): In the best case, all strings are equal! In the 1st iteration, list2 will already null. So we will return an empty list1.
We have iterated a total of n times, so $\boxed{\Omega(n)}$

③ a): $t(n)$ is $O(g(n))$ if $\exists c$ s.t. $\forall n \geq n_0$ $f(n) \leq cg(n)$



b): $t(n) = \begin{cases} 1 & \text{when } n \% 2 \; != 0 \\ 1+2n & \text{when } n \% 2 == 0 \end{cases}$

$t(n)$ can be visualized as follows:



As you can see, $t(n)$ still has a linear behaviour as $n \to \infty$, and is thus an $O(n)$ function. However, since $O(n)$ is an upper-bound, $O(n^2)$ also works. The lower bound is described by $\Omega(1)$.

$\therefore \boxed{O(n), \; O(n^2), \; \Omega(1)}$

④ Notice that in the non-recursive version, you are simulating the call stack of the recursive version

1): Push into Stack:
(building the call stack)

```
while (n > 0) {
        s.push (n % 2);
        n /= 2;
}
```

Notice how the code is separated in 2 parts: $\begin{cases} \text{recursive call} \\ \& \\ \text{return logic} \end{cases}$

2): Pop from stack
(returning from call stack)

```
While (!s.isEmpty()) {
    oddEven = s.pop(); // 1 if odd
                       // 0 if even

    if (oddEven == 1) {
        tmp = tmp * tmp * x;
    } else {
        tmp = tmp * tmp;
    }
```

b): ① Recurrence relation:

@k=1 $$t(n) = t\left(\tfrac{n}{2}\right) + 2$$

note that we are solving for an explicit recurrence here and do not want a big O expression i.e. no need for step ④.

2 here means a max. of two multiplications per recursive call.

② Back Substitution:

@k=2 $$t(n) = \left(t\left(\tfrac{n}{4}\right) + 2\right) + 2$$

2^2

@k=3 $$= \left(t\left(\tfrac{n}{8}\right) + 2\right) + 2 + 2$$

2^3

[...]

@k=k $$= t\left(\tfrac{n}{2^k}\right) + 2\cdot k$$

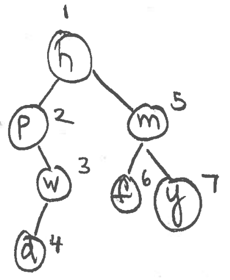
assume $n\%2==0$, $n = 2^k \Leftrightarrow k = \log(n)$

③ Closed form:

$$t(n) = t\left(\tfrac{n}{n}\right) + 2\cdot\log(n)$$
$$= t(1) + 2\log(n)$$
$$\boxed{t(n) = 2\cdot\log(n).}$$

Base case is not given here. However, how many multiplications do we have when computing x^1? We can assume zero

⑤ a):



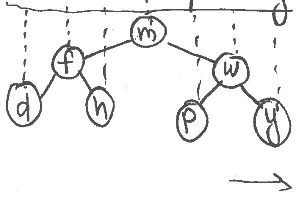
$\Rightarrow$ $\boxed{h\ p\ w\ d\ m\ f\ y}$

i.e. visit the root first, then ALL the subtree of each subchild.

b): $\Rightarrow$ $\boxed{p\ d\ w\ h\ f\ m\ y}$

c): $\Rightarrow$ $\boxed{d\ w\ p\ f\ y\ m\ h}$

d): We want an alphabetic ordering of the letters as such: d f h m p w y
As such, just use projection to construct your tree!



$\longrightarrow$

7 a): a.other = b; a.share (2); { a.sum = 8, b.sum = 9 } ⇒ b.other=c; b.share(4); { $\boxed{\text{b.sum = 13}}$, c.sum = 11 }

c.other=a; c.share(7); { $\boxed{\text{c.sum = 18}}$, $\boxed{\text{a.sum = 1}}$ }

⇒ $\boxed{\text{"Geoff 1 Tina 13 Ted 18"}}$

b): Since the fields are encapsulated with 'private', getters & setters are now needed, in the Sharer superclass:

```
Sharer getOther() { return this.other }
int    getSum () { return this.sum }
void   setOther(Sharer other){ this.other = other }
void   setSum (int sum){ this.sum = sum }
```

Now, whenever you access the fields, you will use these methods:

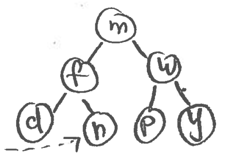
Giver class
```
void share (int n) {
    getOther(). setSum (getOther(). getSum() +n);
    setSum()(getSum()-n);
}
```

Taker class.
```
void share (int n) {
    getOther(). setSum (getOther() getSum() -n);
    setSum (getSum() +n);
}
```

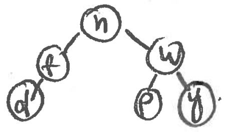
8: NOT IN YOUR EXAM

e): You can either use projection again to construct the tree without 'm', or use the remove(m) method's logic:

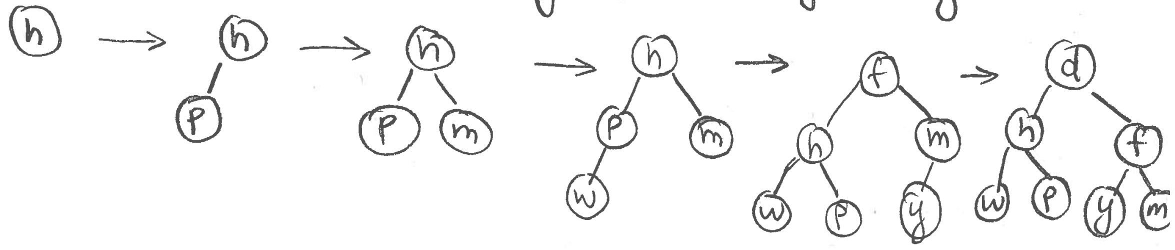
1): Find predecessor of m:



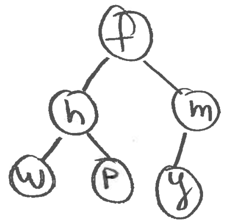
2): n is a leaf! So just replace m with it:



⑥ a): You have not seen this faster O(n) algorithm this semester (F2018). Instead, let's build a heap using the O(n.log(n)) algorithm:



b): Replace root w/ last node & heapify-down



c): • removeMin(): O(log(n)) in worst case you have to bubble down the whole height of heap.

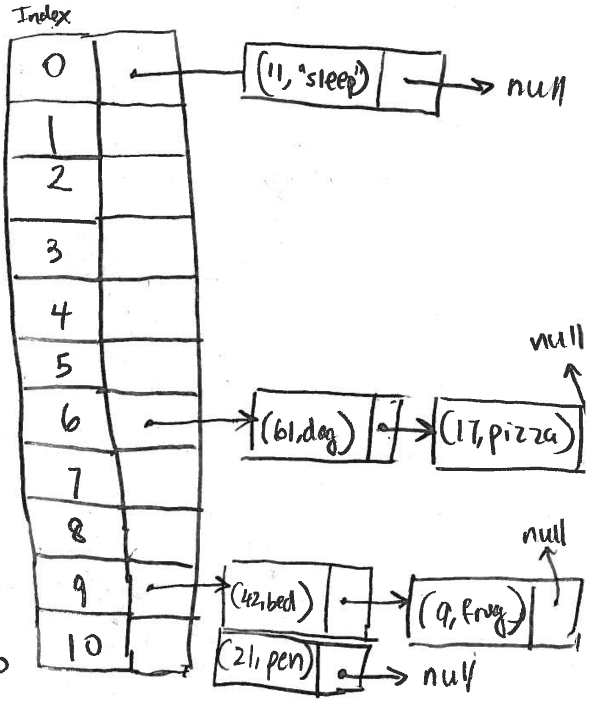
$\Omega(1)$ in best case since you might not bubble anything!

• remove(key): O(n) in worst case. Recall that to delete a key, you must first search for it which may require iterating through all nodes of the tree.

$\Omega(1)$: you are removing the root of the tree.

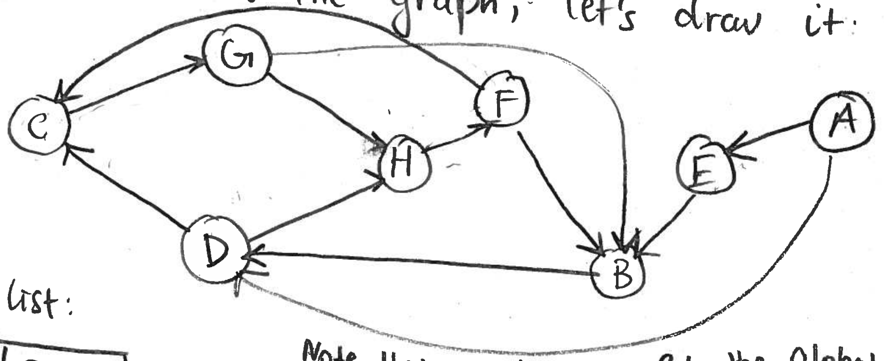
(9) a): By applying hash function h(k) to all keys, we obtain hash values {0, 9, 9, 6, 6, 10}. Note that since we mod the original key with 11, the resulting value must be between [0, 10].

b): • contains(key): notice that computing the hash function is independent on n, so it must be O(1). However, if that function is not properly selected and most (k,v) pairs end up in the same bucket, it might take up to O(b), where b is the max size of a bucket. But, since the purpose of a hashmap is really to provide $\boxed{O(1)}$ access, this is the correct answer.



• contains(value): you would have to iterate through all possible (k,v) pairs to find a value, over all buckets. As such, it takes $\boxed{O(m,n)}$

(10) · To better visualize the graph, let's draw it:



Adjacency list:

A	D, E
B	D
C	G
D	C, H
E	B
F	B, C
G	B, H
[illegible]	[illegible]

Note that you have respect the alphabetic ordering here.

• Breadth first: A D E C H B G F

• Depth first: A D C G B H F E

Winter 2010

1. (a) $(231)_{10} = (11100111)_2$

   (b) h(231) = 231 mod 16 = (224 + 7) mod 16 = 7 OR $(11100111)_2$ mod $(10000)_2 = (0111)_2$

   (c) If we consider the number of unique multiplications, the answer is 12.

$$13^{231} = 13^{115} \cdot 13^{115} \cdot 13$$
$$13^{115} = 13^{57} \cdot 13^{57} \cdot 13$$
$$13^{57} = 13^{28} \cdot 13^{28} \cdot 13$$
$$13^{28} = 13^{14} \cdot 13^{14}$$
$$13^{14} = 13^{7} \cdot 13^{7}$$
$$13^{7} = 13^{3} \cdot 13^{3} \cdot 13$$
$$13^{3} = 13 \cdot 13 \cdot 13$$

   If we instead consider the total number of multiplications by counting multiplications in duplicate recursive calls, the answer is 230.

$13^{3} = 13 \cdot 13 \cdot 13$	0 + 0 + 2 = 2
$13^{7} = 13^{3} \cdot 13^{3} \cdot 13$	2 + 2 + 2 = 6
$13^{14} = 13^{7} \cdot 13^{7}$	6 + 6 + 1 = 13
$13^{28} = 13^{14} \cdot 13^{14}$	13 + 13 + 1 = 27
$13^{57} = 13^{28} \cdot 13^{28} \cdot 13$	27 + 27 + 2 = 56
$13^{115} = 13^{57} \cdot 13^{57} \cdot 13$	56 + 56 + 2 = 114
$13^{231} = 13^{115} \cdot 13^{115} \cdot 13$	114 + 114 + 2 = 230

2. $f(n)$ is $O(g(n))$ if there exists positive numbers n_o and c such that $f(n) \leq c\ g(n)$ for all $n \geq n_o$

   Given $f(n) = (n + 237)^3 + 4n \log n + 3$ and $g(n) = n^3$, want to show that there exists an n_o and c that will satisfy $(n + 237)^3 + 4n \log n + 3 < c\, n^3$

   Knowing that n^3 grows faster than $n \log n$ and 1, can write the following

$$(n + 237)^3 + 4n \log n + 3 < (n + 237n)^3 + 4n^3 + 3n^3 = (238^3 + 4 + 3)n^3$$

   Thus, it is satisfied for $c = 238^3 + 4 + 3,\ n_o = 1$

3. (a) d, e, f, g, h where front is leftmost element

add(a)	a
add(b)	a, b
add(c)	a, b, c
add(d)	a, b, c, d
remove()	b, c, d
add(e)	b, c, d, e
add(f)	b, c, d, e, f
remove()	c, d, e, f
remove()	d, e, f
add(g)	d, e, f, g
add(h)	**d, e, f, g, h**

(b) a, b, c, g, h where top is rightmost

add(a)	a
add(b)	a, b
add(c)	a, b, c
add(d)	a, b, c, d
remove()	a, b, c
add(e)	a, b, c, e
add(f)	a, b, c, e, f
remove()	a, b, c, e
remove()	a, b, c
add(g)	a, b, c, g
add(h)	**a, b, c, g, h**

4. (a) $t(n) = n + 1$

$$t(n) = 1 + t(n-1) = 1 + 1 + t(n-2) = \cdots = 1 + 1 + \cdots + 1 + t(0) = n + t(0) = n + 1$$

(b) $t(n) = nc + 1$

$$t(n) = c + t(n-1) = c + c + t(n-2) = \cdots = c + c + \cdots + c + t(0) = nc + 1$$

(c) $t(n) = \frac{n(n+1)}{2} + 1$

$$t(n) = n + t(n-1) = n + n - 1 + t(n-2) = \cdots = n + n - 1 + \cdots + 2 + 1 + t(0)$$
$$= \sum_{k=1}^{n} k + t(0) = \frac{n(n+1)}{2} + 1$$

(d) $t(n) = c^n$

$$t(n) = c\, t(n-1) = c \cdot c\, t(n-2) = c \cdot c \cdot \ldots \cdot c\, t(0) = c^n \cdot 1 = c^n$$

(e) $t(n) = c\log n + 1$ (let $n = 2\hat{}k$ and then $k = \log n$)

$$t(n) = c + t\left(\frac{n}{2}\right) = c + c + t\left(\frac{n}{4}\right) = c + c + \cdots + c + t\left(\frac{n}{2^k}\right) = c \cdot k + t\left(\frac{n}{2^k}\right) = c\log n + t\left(\frac{n}{n}\right)$$
$$= c\log n + t(1) = c\log n + 1$$

(f) $t(n) = n\log n + n$ (let $n = 2\hat{}k$ and then $k = \log n$)

$$t(n) = n + 2\,t\left(\frac{n}{2}\right) = n + 2\left(\frac{n}{2} + 2\,t\left(\frac{n}{4}\right)\right) = n + n + 2 \cdot 2\,t\left(\frac{n}{2^2}\right) = n + n + \cdots + n + 2^k t\left(\frac{n}{2^k}\right)$$
$$= n\,k + 2^k t\left(\frac{n}{2^k}\right) = n\log n + n\,t(1) = n\log n + n$$

5. (a) Each of the three subsets needs to be sorted so there is a recursive call on $\frac{n}{3}$ of the elements. In addition there is the merging operation which is done in n steps. Thus, the following recurrence relation is obtained: $t(n) = 3\,t\left(\frac{n}{3}\right) + n$
(b) $t(n) = n\log_3 n + n$ (let $n = 3\hat{}k$ and then $k = \log_3 n$)

$$t(n) = n + 3t\left(\frac{n}{3}\right) = n + 3\left(\frac{n}{3} + 3t\left(\frac{n}{9}\right)\right) = n + n + 3 \cdot 3t\left(\frac{n}{9}\right) = n + n + \cdots + n + 3^k t\left(\frac{n}{3^k}\right)$$
$$= n\,k + 3^k t\left(\frac{n}{3^k}\right) = n\log_3 n + n\,t(1) = n\log_3 n + n$$

6. (a) [c,d,m,h,w,p]

[b,c,m,h,d,p,w] → [_,c,m,h,d,p,w] → [w,c,m,h,d,p] → [c,w,m,h,d,p] → [c,d,m,h,w,p]

(b)

```
      h
    /   \
   c     p
  / \   / \
 b   d m   w
```

(c) 2 possible answers

```
      d                  m
    /   \              /   \
   c     p            c     p
  /     / \          / \     \
 b     m   w        b   d     w
```

7. (a) The base case is when the root is found and that happens when the parent node is null. Otherwise, if there is a valid parent, can call the recursive function on the parent.

```
findRoot(node){
      if(node.parent == null)
            return node
      else
            return findRoot(node.parent)
}
```

(b) The input node is i since it has height of 1 there and was moved along with g.
(c) The base case is when the input node is the root. Otherwise, the root can be set as parent of the input node and can call the recursive function on the old parent.

```
flatten(node, root){
      if(node != root){
            flatten(node.parent, root)
            node.parent = root
      }
}
```

(d) The advantage is that after the tree is flattened, it will be easy to find the root of the input node using findRoot() again because it will have a depth of 1.

8. (a) Note that if A overlaps B, then B cannot overlap A because it is under it.

A	B
B	
C	E, F, G
D	E
E	H
F	D
G	D
H	B, F

(b) C, E, F, G, H, D, B
Note that you visit a node first and then its children and then each of their children (one level at a time)
(c) C, E, H, B, F, D, G
Note that you visit a node first and then its first child and then the first child's child and so on. You must clear all children of a node before moving on to another node of the same level.

9. (a) Overriding is when a method in a child class has the same signature (same method name and same arguments) as a method in the parent class and its implementation is rewritten.
   Overloading is when multiple methods (in a same class or across a class and its parents) have the same method name but different signatures (different arguments).
   (b) If a.equals(b) is true then they must have the same hash code (ie a.hashCode() == b.hashCode() is true)
   If a.equals(b) is false then nothing can be said about their hash codes

   Compilation error is at (e) since the class A has no method foo() that takes an argument. When foo() is added in the child class, it is not also overloaded in the parent class, only in the child class.

Winter 2009

1) You can ignore this question ☺
For those interested, the answers are

F1	.0011
F2	.1000
F3	.1001
F4	.1110
F5	1.0000

Labels are 1112222234444455

2) (a) The method will print the nodes in the list, starting from the front of the list until the input node. Note that if the recursive call was done after the print statement, it would instead print the nodes from the start to the input node in reverse order.
(b) The printing operation takes constant time c and given that n is the number nodes between the start of the list and the input node, then the recursive call is done on $n-1$ nodes. Accordingly, the recurrence relation is $t(n) = c + t(n-1)$

$$t(n) = c + t(n-1) = c + c + t(n-2) = c + c + \cdots + c + t(1) = c(n-1) + t(1)$$

3) $f(n)$ is $O(g(n))$ if there exists positive numbers n_o and c such that $f(n) \leq c\ g(n)$ for all $n \geq n_o$. Also, $f(n)$ is not $O(g(n))$ if there exists positive numbers n_o and c such that $f(n) > c\ g(n)$ for all $n \geq n_o$.

(a) $3n + 8 \leq 3n + 8n = 11n$ so let $c = 11$ and $n_o = 1$
(b) $3n + 8 > c \leftrightarrow n > \frac{c-8}{3}$ so let $n_o > \frac{c-8}{3}$
(c) Want to show $6n \leq c\ 2^n$ for any $n > n_o$.
Let $c = 6, n_o = 1$ then only need to show $n \leq 2^n$ for $n > 1$.

Base case: for n=1 we get $1 < 2$ true
Induction hypothesis: Assume $k \leq 2^k$ true
Then add 1 to both sides and get $k + 1 \leq 2^k + 1 \leq 2^k * 2 = 2^{k+1}$

4) (a) O(1) since arrays are constant time access
(b) O(n) since need to make space for new element by shifting all
(c) O(1) since you can just change the head and the next pointer of the added element
(d) O(n) since you need to iterate from the start of the list until the desired node
(e) O(n) since you need to iterate from the start of the list until the desired node
(f) O(logn) since you need to only shift half as many elements every iteration
(g) O(1) since the minimum is the root
(h) O(n) since you need to iterate through all elements to find the max

5) (a) d, e, f, g, h where front is leftmost element

enqueue(a)	a
enqueue(b)	a, b
enqueue(c)	a, b, c
dequeue()	b, c
enqueue(d)	b, c, d
dequeue()	c, d
enqueue(e)	c, d, e
enqueue(f)	c, d, e, f
dequeue()	d, e, f
enqueue(g)	d, e, f, g
enqueue(h)	**d, e, f, g, h**

(b) The front of the queue is highlighted in yellow. Note that at the last enqueue, it is required to expand the array as it reached maximum capacity and a new element needs to be added.

enqueue(a)	**a**	*	*	*	
enqueue(b)	**a**	b	*	*	
enqueue(c)	**a**	b	c	*	
dequeue()	*	**b**	c	*	
enqueue(d)	*	**b**	c	d	N/A
dequeue()	*	*	**c**	d	
enqueue(e)	e	*	**c**	d	
enqueue(f)	e	f	**c**	d	
dequeue()	e	f	*	**d**	
enqueue(g)	e	f	g	**d**	
enqueue(h)	**d**	e	f	g	h

6) (a) Maximum number of nodes = $2^{h+1} - 1$
(b) Given depth i, there are 2^i nodes at that level. Thus the sum is $\sum_{i=0}^{h} i\, 2^i$
(c) If we make recursive calls of the function on all of the children, the base case is when the child (next input node) is null.

```
numNodes(node){
        if(node == null)
                return 0
        else{
                sum = 1
                for each child c of node
                        sum += numNodes(c)
                return sum
        }
}
```

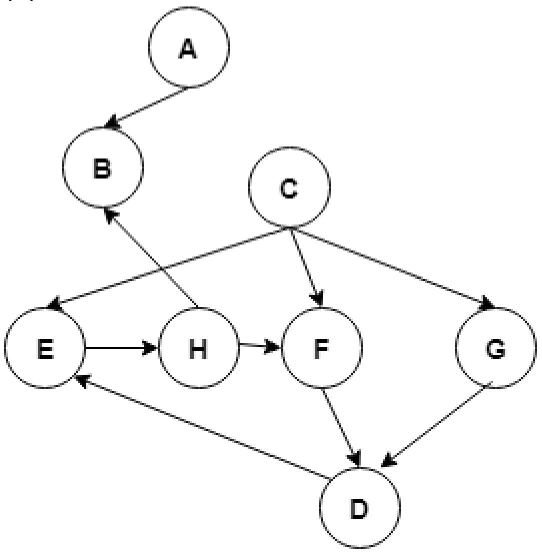
(d) Again, if we make recursive calls of the function on all of the children, the base case is when the child (next input node) is null.

```
sumDepths(node,depth){
      if(node == null)
            return 0
      else{
            sum = depth
            for each child c of node
                  sum += sumDepths(c,depth+1)
            return sum
      }
}
```

7) (a) Note that if A overlaps B, then B cannot overlap A because it is under it.

A	B
B	
C	E, F, G
D	E
E	H
F	D
G	D
H	B, F

(b)



(c) Ordering of visited nodes is CEFGDHB

```
      C
    / | \
   E  F  G
  / \
 D   H
     |
     B
```

(d) Ordering of visited nodes is CEDHBFG

```
      C
    /   \
   E     G
  / \
 D   H
    / \
   B   F
```

(e) There are many possible answers. An example is A C GE DH F B

(f) Can arrange rectangles such that each rectangle overlaps another one and is overlapped by another one.

A → B → C → D → A

```
     |    |
   --|-------
     |    |
     |    |
   -------|-
     |    |
```