

Example solutions.

Please note that these are just examples. Some people wrote variations on this, and they were still marked correct.

In week 13 in lectures I will discuss in detail these answers, as well as covering some revision topics in readiness for the mid year examination.

AKM

(a)

```
int height(treeNode* t) {
// pre : t is a pointer to a binary tree
// post : returns the height of t
    if (t==NULL) return 0;
    else return height(t->left) MAX height(t->right);
}
```

O(n) complexity.

(b)

```
int product(treeNode* t) {
// pre : t is a pointer to a binary tree
// post : returns the product of the items in t

    if (t==NULL) return 1; // Base case for multiplication is always 1
    else return product(t->left) * product(t->right);
}
```

O(n) complexity.

(c)

```
int maxVal(treeNode* t, int k) {
// pre : t is a pointer to a binary tree, k is an integer;
// post : returns the smallest item in t which is at least the value of k;
// returns -1 if there is no such item in t.

    if (t==NULL) return -1;
    else { int a= maxVal(t->left, k); // get the results for the subtrees
           int b= maxVal(t->right, k);
           if (a== -1 || b== -1) {a = a MAX b;} else { a= a MIN b;} // Combine
them for the result
           if (a != -1) { if (t->item < k) return t-> item; else return -1;
           else (if t->item < k) return a; else return (t->item MIN a);
}
}
```

$O(n)$ complexity.

(d)

```
bool full(treeNode* t) {
// pre : t is a pointer to a binary tree
// post : returns true if t is full, and false otherwise.

if (t==NULL) return true;
else { int a= numNodes(t->left);
      int b= numNodes(t->right);
      return full(t->left) && full(t->right) && (a == b);
}
}
```

Complexity is $O(n*n)$ --- this is because numNodes is $O(n)$, and is called for each node.

(e) Only (c) because it could make use of the ordered structure.

Q2:

(a)

```
void takePut(node* &p, node* &q) {
  if (p!=null) { // Nothing to do if p is empty
    node* temp= p; // save p
    p= p->next; // move p
    temp->next= q; // put the old top of p onto q
    q= p; // move q
  }
}
```

(b)

```
node* antiMerge(node* &p, node* &q) {
  node* merged= NULL; // add nodes to an empty list

  while (p!=NULL && q!=NULL) { // choose which one to add, smallest goes
first
    if (p-> item <= q-> item) {
      takePut(p, merged); // use takePut to remove a node to merged
    }
    else takePut(q, merged);
  }

  while (p!=NULL) { // now check for the remaining cases...
    takePut(p, merged);
  }
  while (q!=NULL) {
    takePut(q, merged);
  }
}
```

```
    return merged; // return the result.  
  }  
}
```