1. The time (number of comparisons) required to insert into a binary search tree that contains N nodes is:
   a. always BigOmega(N)
   b. BigOmega(N) in the worst case.
   c. always bigO(N)
   d. bigTheta(N) in the worst case.
   e. always bigOmega(lgN)
   f. b and c
   g. b,c,d,e
   h. c and e

2. If an array containing N items starts out in ascending order and is then modified by swapping 10 pairs of items, then in the worst case, what can you say about the asymptotic times required by the algorithms discussed in class and in the book?
   a. insertion-sorting the result will require bigOmega(N^2) time
   b. heap-sorting the result will require bigO(N^2) time.
   c. merge-sorting the result will be faster than insertion-sorting it.
   d. heap-sorting the result will be faster than insertion-sorting it.
   e. a, c, d
   f. a and c
   g. a and d

3. If program A requires bigTheta(N^2) time to perform its calculation on inputs of size N in the worst case and algorithm B requires bigTheta(100N) time in the worst case, then
   a. B takes less time in the worst case, and is therefore the algorithm to use for any application
   b. B takes less time as long as N > 100
   c. if there is a max size input your program has to deal with, then either A or B might be preferable
   d. b and c

4. Suppose that we have a linked list of records with strings of letters 'a' through 'z' as keys, where there are N records, and the longest key has length L.
   a. LSD radix sort will be substantially faster on this data set if the records are already sorted.
   b. the worst case for MSD radix sort will occur if all keys are the same.
   c. if all the keys are distinct, then L >= log(base26)N.
   d. in the worst case, insertion sort will require bigO(N^2) machine instructions.
   e. in the worst case, insertion sort will require bigOmega((N^2)L) machine instructions.
   f. a, b, c
   g. b, c, d
   h. b, c, e
   i. a, c, e

5. Consider a hash table containing N keys. Assume that the hash function requires bigO(1) time, and that it maps equal numbers of these keys (+1,-1) to each bucket number, regardless of the number of buckets. Assume also that the table's load factor is bounded by a constant.
   a. if the hash table uses external chaining, then the total cost of adding all N keys to the table and then performing K additional look-up operations is bigO(N+K).
   b. (a) is still true if the hash table uses open-address hashing instead.
   c. (a) is only true if the number of buckets remains constant.
   d. none of the above
   e. a and b
f. a and c

6. Consider tries for strings that consist only of lower-case letters a-z and an end-of-string characters. Assume also that the pointers to each node's children are stored in 27-element arrays indexed by this alphabet of characters. For a trie containing N distinct keys, the longest of which has L characters,
a. looking up a string requires bigO(L) operations in the worst case.
b. looking up strings of length K requires bigTheta(K) operations in all cases, if they turn out to be present in the trie.
c. if there are K keys in the trie between artichoke and basilisk in the dictionary order, then finding and printing them all requires bigTheta(N+K) operations in the worst case.
d. if there are K keys in the trie between artichoke and basilisk in the dictionary order, then finding and printing them all requires bigTheta(KL) operations in the worst case.
e. a and c
f. b and c
g. a, b, c
h. a, b, d

7. Consider an AVL tree whose height is h.
a. this tree contains bigOmega(alpha^h) keys, where alpha = (1/2)(1+5^(1/2)), approximates to 1.62.
b. this tree contains bigTheta(2^h) keys.
c. this tree contains bigO(h) keys in the worst case.
d. none of the above.

8. suppose that Ak is the int value whose least significant k bits are 1s and whose other bits are 0s. Which of the following expressions computers Ak, for all values 0<=k<=32.
a. (1< b. (-1)>>>(32-k)
c. ~((-1)< d. all the above
e. b and c
f. none of the above.

9. Which of the following are true of an order 7 B-tree containing N > 0 keys?
a. if we say the root is at level 0, its children at level 1, their children at level 2, etc., then the number of keys stored at level k+1 is at least 4 times the number at level k, for k > 1.
b. The maximum level number of any key (see item (a)) is <=log(base4)N.
c. the height of the tree is bigOmega(lg N).
d. all of the above
e. a and b
f. a and c
g. none of the above

10. Consider a priority queue implemented with a heap stored in an array, where the first item in the queue is the largest. If the heap contains N keys, then
a. the time required to find the kth largest key is bigO(k lnN), if we are allowed to remove items from the heap to find it.
b. the elements of the array used to represent the heap could be in descending order.
c. the keys stored in the last floor(N/2) items of the heap have no children.
d. the smallest floor(N/2) keys are stored in the last floor(N/2) elements of the array, but are in arbitrary order.
e. all of the above
f. a, b, c
g. b, c, d
h. none of the above
11. We can represent nodes in arbitrary ordered trees (that is, where each node can have any number (>=0) of non-empty children) using the following representations:

```java
import java.util.*;

class TreeNode {
    String label; /* The value stored at the node. */
    TreeNode child0; /* The first child of THIS node (or null if none) */
    TreeNode sib; /* The next sibling of THIS node (that is, the next node having the same parent). */
    ...
}
```

We wish to write a program to determine if two trees represented in this fashing have the same (ie, .equal) labels in the same order (that is, a preorder walk of each tree gives the same sequence of labels). Here is a skeleton for a solution that is to replace the '...' in the class above.

```java
static boolean equalTrees(TreeNode t0, TreeNode t1) {
    Iterator i0,i1;
    for (i0 = t0.iterator(), i1 = t1.iterator(); ______(A)______;) {
        ______(B)______;
    }
    return ______(C)______;
}
```

```java
Iterator iterator() { return new TreeIter(this); }
```

```java
static class TreeIter implements Iterator {
    Stack nodes = new Stack() ;
    TreeIter(TreeNode root) { nodes.push(root); }

    public boolean hasNext() {
        return _________(D)_____;
    }

    public Object next() {
        TreeNode r = ________(E)______;
        ________(F)______;
        ________(G)______;
        return r.label;
    }

    public void remove() { throw new UnsupportedOperationException(); }
}
```

A. Which of the following work as replacements for (A), (B) and (C)?

a. (A): i0.hasNext() && i1.hasNext();
(B): if (!i0.next().equals(i1.next())) return false;
(C): !i0.hasNext() && !i1.hasNext();
b. (A): i0.hasNext() || i1.hasNext();
   (B): if (! i0.next().equals(i1.next())) return false;
   (C): i0.hasNext() || i1.hasNext();
c.
   (A): i0.hasNext() && i1.hasNext() && !i0.next().equals(i1.next())
   (B): /* Empty statement*/
   (C): i0.hasNext() ? false : !i1.hasNext()
d. (A): i0.hasNext() && i1.hasNext();
   (B): if (! ((String) i0.next()).equals((String)i1.next())) return false;
   (C): !i0.hasNext() && !i1.hasNext();
e. all of the above
f. a and c
g. a, c, d
h. a and d
i. none of the above

B. Which of the following works as a replacement for (D)?
a. !nodes.isEmpty()
b. nodes.size()>0
c. nodes.size()>1
d. nodes.peek() = =null
e. a and b
f. none of the above

C. Which of the following works as a replacement for (E)?
a. (TreeNode) nodes.peek();
b. nodes.peek();
c. nodes.pop();
d. all of the above
e. a and b
f. none of the above

D. Which of the following work as replacements for (F) and (G)?
a. (F): if (r.child0 != null) nodes.push (r.child0);
   (G): if (r.sib != null) nodes.push(r.sib);
b. (F): if (r.sib != null) nodes.push(r.sib);
   (G): if (r.child0 != null) nodes.push(r.child0);
c. (F): if (r.child0 != null ) nodes.push(r.child0);
   (G):
   for (TreeNode p =r.sib; p!=null; p=p.sib)
      nodes.push(p);
d. (F): for (TreeNode p=r.sib; p!=null; p=p.sib)
       nodes.push(p);
   (G): if (r.child0!=null) nodes.push (r.child0);
e. all of the above
12. After declaring:

```java
String S1 = "Hello, world",
S2 = "Hello, ",
S3 = "world",
S4 = S2+S3,
S5 = S4;
```

Which of the following are true?

a. S4==S1
b. S4.equals(S1)
c. S5==S4
d. S4==S2 + S3
e. all of the above
f. b and c
g. b, c, d
h. none of the above

13. A certain graphic library defines a class called Component, which is supposed to represent an arbitrary piece of a screen image. A Component may contain subcomponents (each of which is a Component). The graphics system renders components, drawing them on the screen. Rendering a Component renders its subcomponents and then adds any other drawing that is specific to that particular kind of Component (renderMe). Here are some of the Components in the library:

```java
/** A section of a drawing. A Component contains >=0 subcomponents. */
public ____ (A) ____ class Component {
  /** A Component at location (X, Y) on the screen */
  public Component (int x, int y) {this.x=x; this.y=y; }
  /** X and Y coordinates of THIS. */
  public int x () {return x; }
  public int y () {return y; }

  /** Draw the picture represented by THIS, not including its subcomponents. */
  protected ____ (A) ____ void renderMe () ____ (B) ____

  /** Draw the picture represented by THIS, including its subcomponents. */
  public void render() {
    ____ (C) ____
  }

  /** An iterator returning all subcomponents of THIS. */
  public Iterator iterator () {____ (D) ____; }
  private int x,y;
}

/** A Container serves simply to hold other Components. */
public class Container extends Component {
    public Container (int x, int y) {___(E)___}
    ___(F)___
    public Iterator iterator () { return ___(G)___; }
    /** Add C as a subcomponent of This. */
    public void add (Component c) {subComponent.add(c); }
    List subComponents=new LinkedList();
}

public class Text extends Component {
    /** Displays the text S at location (X,Y) when rendered. */
    public Text (String s, int x, int y) { ___(E)___ contents =s; }
    protected void renderMe () { /*draw contents on screen at x(), y(); */}
    private String contents;
}

public class Rectangle extends Component {
    /** Displays a WxH rectangle location (X,Y) when rendered. */
    public Rectangle (int x, int y, int w, int h) { ___(E)___ this.w=w; this.h=h; }
    protected void renderMe() { /*draw wxh rectangle screen at x(), y(); */}
    private int w, h;
}

A. Which of the following are valid replacements for (A), (B) and (F)? (Note: "valid" means that they have to work with the rest of the classes listed here).

a.  
(A): abstract
(B): ; // ie. just a semicolon
(F): public void renderMe() {}

b.  
(A): /* nothing */
(B): {}
(F): /*nothing */

c. both of the above
d. none of the above

B. Which of the following are valid replacements for (C)?

a.  
(C): for (Iterator i=iterator(); i.hasNext(); )
    ((Component) i.next()).render();
    renderMe();

b.  
(C): if (this instanceof Container)
    for (Iterator i=iterator(); i.hasNext(); )
    ((Component)i.next()).render();
    else
    renderMe();

c.  
(C): for (Iterator i=iterator(); i.hasNext(); )
    ((Component)i.next()).renderMe();
    renderMe();

d. all of the above
e. a and b
f. a and c
  g. none of the above

C. Which of the following are valid replacements for (E), in all the places it appears?
  a. Nothing (leave blank)
  b. this(x,y);
  c. super(x,y);
  d. this.x=x; this.y=y;
  e. all of the above
  f. b and d
  g. c and d
  h. none of the above

D. Which of the following are valid replacements for (D) and (G)?
  a. (D): if (this instanceof Container)
    return subComponents.iterator();
    else
    return Collections.EMPTY_LIST.iterator()
  (G): subComponents.iterator();
  b. (D): if (this instanceof Container)
    return subComponents.listIterator();
    else
    return Collections.EMPTY_LIST.iterator()
  (G): subComponents.listIterator();
  c. (D): if (this instanceof Container)
    return ((Container) this).subComponents.iterator();
    else
    return Collections.EMPTY_LIST.iterator()
  (G): subComponents.listIterator();
  d. (D): Collections.EMPTY_LIST.iterator()
  (G): subComponents.iterator();
  e. (D): Collections.EMPTY_LIST.iterator()
  (G): subComponents.listIterator();
  f. all of the above
  g. a and b
  h. d and e
  i. c, d, e
  j. none of the above

15. Consider the following methods:

    static List uniq1 (List L0) {
      List L = L0;
      return uniq1a (L, L.size ()-1);
    }

    static List uniq1a (List L, int k) {
      if (k<=0)
return L;
else if (L.get(k).equals(L.get(k-1)))
L.remove(k);
return uniq1a (L,k-1);
}

static List uniq2 (List L0) {
if (L0.size()<= 1)
return new LinkedList (L0);
else if (L0.get(0).equals(L0.get(1)))
return uniq2(L0.sublist(1, L0.size()));
else {
List r=uniq2(L0.sublist(1,L0.size()));
r.add(L0.get(0),0);
return r;
}
}

These methods remove adjacent duplicate elements from a list. Which of the following is true?
a. uniq1 is non-destructive and uniq2 is destructive.
b. uniq1 is destructive and uniq2 is non-destructive.
c. both uniq1 and uniq2 are destructive.
d. both uniq1 and uniq2 are non-destructive.