NAME: **circle precept:**
login ID: **P01 P01A P02 P02A P03**

COS 226 Midterm Exam, Spring 2010

This test is 10 questions, weighted as indicated. The exam is closed book, except that you are allowed to use a one page cheatsheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. *Put your name, login ID, and precept number on this page* (*now*), and write out and sign the Honor Code pledge before turning in the test. You have 80 minutes to complete the test.

"I pledge my honor that I have not violated the Honor Code during this examination."

1	/5
2	/5
3	/10
4	/5
5	/5
6	/10
7	/10
8	/10
9	/10
10	/20
11	/10
TOTAL	/100

March 8, 2010

1. **Partitioning** (5 points). Give the result of partitioning the array with standard Quicksort partitioning (taking the N at the left as the partitioning element).

```
N   O   P   A   R   T   I   T   I   O   N   I   N   G   B   U   G   S
    G                                                           O
        B                                               P
                G                                   R
                    N                           T
                            I               T
                                    N   O
N                                   N
```

2. **Estimating running time** (5 points). Suppose that you run the code fragment below (generate and then Mergesort an array of random double values) for N = 10,000,000 and observe that it takes 5.3 seconds.

```
Double[] a = new Double[N];
for (int i = 0; i < N; i++)
  a[i] = Math.random();
Merge.sort(a);
```

Assuming you have enough memory which of the following is a reasonable prediction of its running time (in seconds) for N = 1,000,000,000?

A. 53 seconds.

B. 340 seconds

C. 530 seconds.

D. 680 seconds

E. 1060 seconds

F. 5300 seconds

3. **Social networking** (10 points). Suppose that a social networking website FRIENDS needs to support two operations: (*i*) declare A and B to be friends (thus making all of As friends and all of Bs friends friends of each other); and (*ii*) determine whether A and B are friends.

Which APIs should FRIENDS use to support these operations (*circle two*)?

A. Queue.

B. **Union-find.**

C. Stack.

D. Priority queue.

E. **Symbol table.**

F. Randomized queue.

Give the worst case order of growth of the running time that FRIENDS can guarantee for *M* operations, where *N* is the number of people listed on the website (circle one).

G. N log M.

H. **M log N.**

I. N log N.

J. M.

K. N log* M.

L. M log* N.

In *one or two sentences*, justify your answer (describe how FRIENDS should implement the two operations).

Use a symbol table to map names to ids. For (i), union(A, B). For (ii), find(A, B).

4. **Sorting algorithms** (5 points). Match each of the types of input files described at right below with the *most appropriate sorting algorithm* (as presented in lecture and in the book) by writing the letter corresponding to an algorithm in the blank to the left of the corresponding file type. You should use each letter only once (and leave two letters unused).

A. Mergesort	__F___ Huge file, not many different key values
B. Quicksort	__E___ Huge records
C. Heapsort	__D___ Several new records appended to huge sorted file
D. Insertion sort	__B___ Huge file of `double` values, not much extra space available, speed matters
E. Selection sort	__A___ Huge file, speed and stability matter
F. 3-way quicksort	
G. Shellsort	

5. **Random sort** (5 points). Operating under court order, a certain computer company recently decided to randomly assign the order of browsers for customers to choose by using a system sort with the following broken `compareTo()` implementation.

```
public int compareTo(Browser b)
{  if (Math.random() < 0.5) return -1; else return +1;  }
```

Assume that (since the list of browsers is short) the system uses our version of insertion sort for the task. Where would you prefer that your company's browser be in the list given as input to the sort?

A. At the beginning.

B. Second from the beginning.

C. Doesn't matter, since the sort randomizes the array.

D. Next to last.

E. At the end.

F. **Either at the beginning or second from the beginning.**

6. **Mergesort** (10 points). Consider the following implementation of recursive mergesort:

```
public class Merge
{
  public static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
  {
     if (hi <= lo) return;
     int mid = lo + (hi - lo) / 2;
     sort(a, aux, lo, mid);
     sort(a, aux, mid+1, hi);
     merge(a, aux, lo, mid, hi); // merges 2 sorted subarrays into a[lo..hi].

     System.out.print(lo + " " + hi + " ");
     for (int i = lo; i <= hi; i++)
        System.out.print(a[i] + " ");
     System.out.println();
  }

  public static void sort(Comparable[] a)
  {
     int N = a.length;
     Comparable[] aux = new Comparable[N];
     sort(a, aux, 0, N-1);
  }
}
```

Note that the last three lines of the recursive method have been instrumented to print the values of the indices and the contents of the array. The output produced by these methods when invoked by the following code appears below in scrambled order:

```
Character[] a = { 'z', 'y', 'x', 'w', 'v', 'u', 't' , 's' , 'r' };
Merge.sort(a);
```

A. `0 2 x y z`

B. `3 4 v w`

C. `0 4 v w x y z`

D. `5 8 r s t u`

E. `0 8 r s t u v w x y z`

F. `0 1 y z`

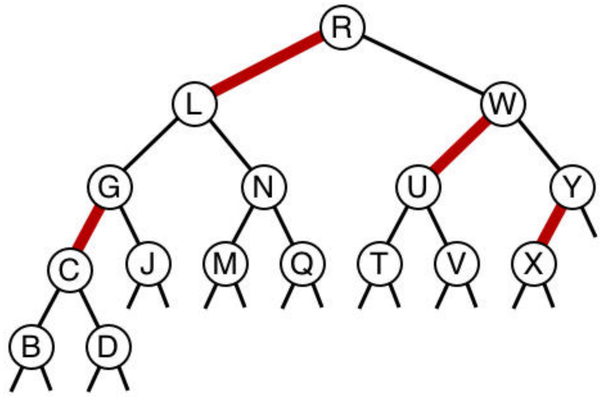
G. `7 8 r s`

H. `5 6 t u`

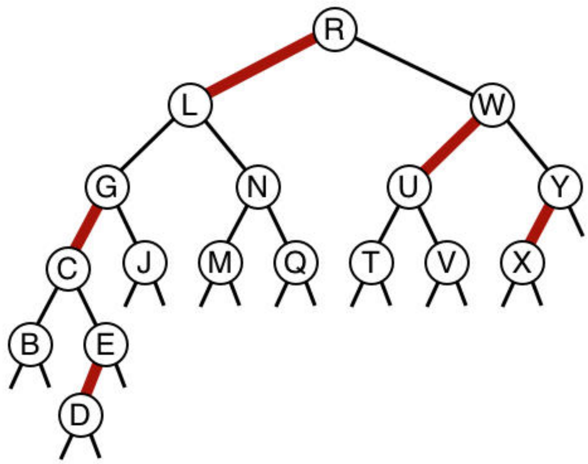
Give the order in which these lines actually appear in the output by writing one letter in each of the blanks below (the last one is filled in for you).

__F__ __A__ __B__ __C__ __H__ __G__ __D__ __E__

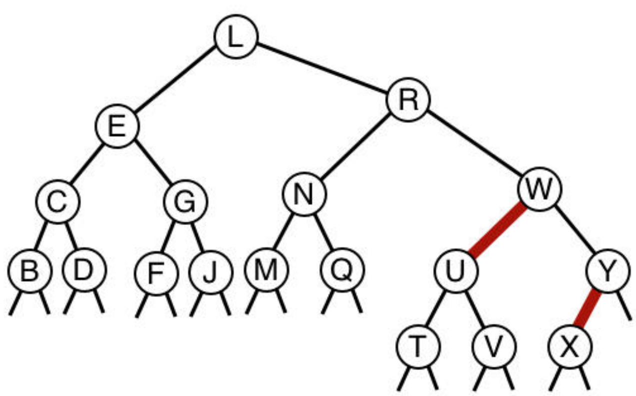
7. **LLRB insertion** (10 points). The following diagram shows a left-leaning red-black tree Thick lines are red links.



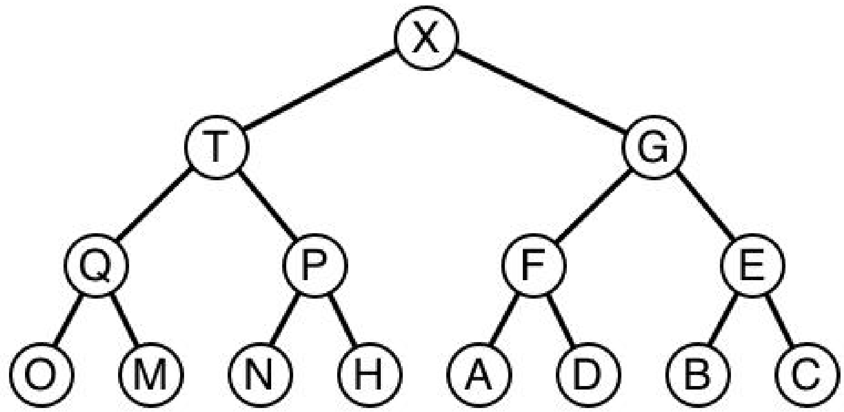
A. (2 points) Draw the tree that results after E is inserted.



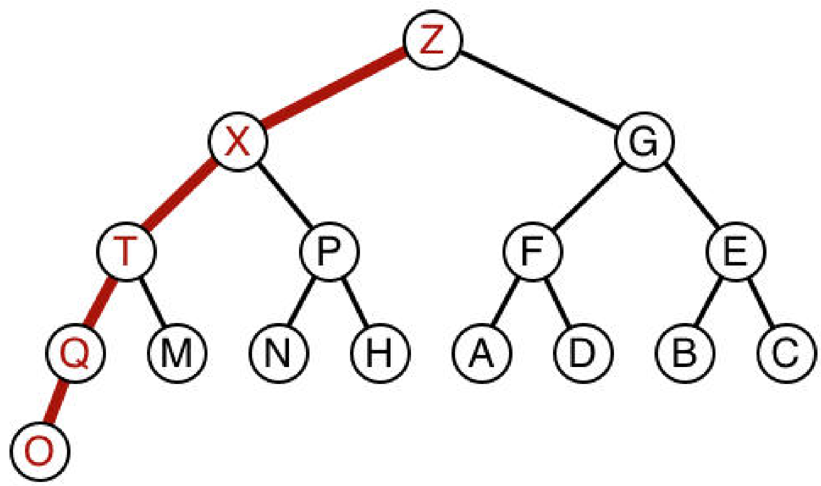
B. (8 points) Draw the tree that results after F is inserted into your tree from Part A. *Hint*: You might find it easiest to convert to the 2-3 tree representation, then do the insertion, then convert back to the red-black tree representation.



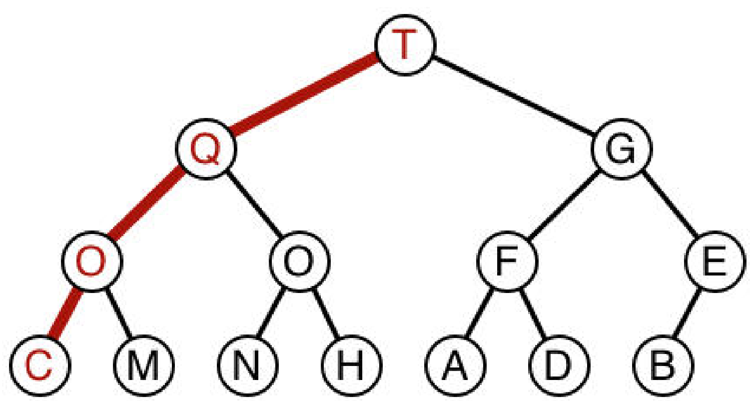
8. **Heap operations** (10 points). Consider the following max-heap:



A. Draw the result of inserting Z.



B. Draw the result of deleting the maximum from the original max-heap shown above (before Z has been inserted).



9. **Linear probing** (10 points). Give the result of inserting the following keys P R O B I N G into an empty linear probing hash table of size M = 7, using the hash function `f(x) = i % 7`, where `x` is the `i`th letter of the alphabet.

x	P	R	O	B	I	N	G
i	16	18	15	2	9	14	7
f(i)	2	4	1	2	2	0	0

0	1	2	3	4	5	6
N	O	P	B	R	I	G

10. **7 sorting algorithms** (20 points). The leftmost column is the original input of strings to be sorted, and the rightmost column is the sorted result. The other columns are the contents at some intermediate step during one of the 7 sorting algorithms listed below. Match up each algorithm by writing its letter under the corresponding column. Use each letter exactly once.

rush	abba	blue	abba	fixx	abba	neyo	zman	abba
korn	acdc	cars	blue	inxs	acdc	korn	yani	acdc
fixx	blue	devo	cars	korn	beck	fixx	yoyo	beck
inxs	beck	enya	devo	rush	blue	inxs	tatu	blue
cars	cars	fixx	dido	cars	cake	cars	styx	cake
enya	cake	fuel	enya	devo	cars	enya	ween	cars
devo	devo	inxs	fixx	enya	cher	devo	seal	cher
fuel	epmd	korn	fuel	fuel	devo	fuel	lons	devo
tatu	cher	moby	inxs	blue	dido	lons	kiss	dido
styx	inxs	rush	korn	moby	doom	mims	nofx	doom
blue	dido	styx	moby	styx	enya	blue	pras	enya
moby	fuel	tatu	muse	tatu	epmd	moby	rush	epmd
abba	doom	abba	rush	abba	rush	abba	neyo	fixx
muse	kiss	muse	seal	dido	muse	muse	muse	fuel
seal	enya	seal	styx	muse	seal	cher	mims	inxs
dido	lons	dido	tatu	seal	tatu	dido	fuel	kiss
beck	fixx	beck	acdc	acdc	fixx	beck	beck	korn
kiss	neyo	kiss	beck	beck	kiss	kiss	inxs	lons
acdc	korn	acdc	doom	kiss	korn	acdc	acdc	mims
yani	moby	yani	kiss	yani	yani	epmd	cars	moby
nofx	muse	nofx	nofx	doom	nofx	nofx	korn	muse
doom	pras	doom	pras	nofx	styx	doom	doom	neyo
pras	mims	pras	yani	pras	pras	pras	blue	nofx
yoyo	seal	yoyo	yoyo	yoyo	yoyo	cake	moby	pras
ween	nofx	ween	ween	cake	ween	rush	fixx	rush
zman	tatu	zman	zman	neyo	zman	zman	abba	seal
neyo	rush	neyo	neyo	ween	neyo	ween	enya	styx
cake	yani	cake	cake	zman	inxs	yoyo	cake	tatu
epmd	ween	epmd	epmd	cher	moby	yani	epmd	ween
cher	zman	cher	cher	epmd	fuel	seal	cher	yani
mims	styx	mims	mims	lons	mims	styx	devo	yoyo
lons	yoyo	lons	lons	mims	lons	tatu	dido	zman
	B	C	F	A	E	D	G	

A. Bottom-up mergesort
B. Shellsort
C. Insertion sort
D. Quicksort (with no random shuffle)
E. Selection sort
F. Top-down mergesort
G. Heapsort

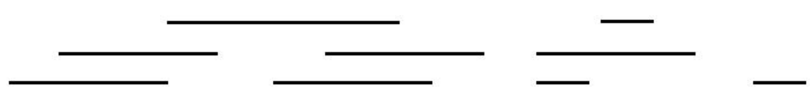
11. **Interval clusters** (10 points). Consider the following data type, for intervals on the line:

```
public class Interval implements Comparable<Interval>
{
   private final int left;
   private final int right;

   Interval(int left, int right)
   {  this.left = left; this.right = right;  }

   public int compareTo(Interval b)
   { return this.left - b.left;  }
}
```

For a particular application, *clusters* of intervals are of importance. To find clusters, replace any pair of intervals that intersects (by even an endpoint) by the union of the two intervals, continuing until all intervals do not intersect. For example, the following set of intervals has 3 clusters:



Note that you are guaranteed to have Intervals with non-negative numbers. Given an array of intervals, how many clusters are there? The brute-force algorithm is quadratic, but an enterprising COS226 student figured out a way to find the number of clusters in an array of intervals in *linearithmic* time, with the following code to be added to `Interval`.

```
public static int count(Interval[] a)
{
   Arrays.sort(a);
   int cnt = 1;
   int max = a[0].right;
   for (int i = 1; i < a.length; i++)
   {
      // "Missing" line of code
      if (a[i].left > max) cnt++;
      if (a[i].right > max) max = a[i].right;
   }
   return cnt;
}
```

In the space below, write the *one line* of code that is missing.