Name: Solution Key

CS 112—Midterm

Spring, 2013

There are 8 problems on the exam. The first and last are mandatory, and you may eliminate any one of problems 2 – 7 by drawing an X through it. Problem 1 is worth 10 points, and all other problems are worth 15. Please write in pen if possible. If you need more room, use the back of the sheet and tell me this on the front sheet.

Problem One (10 pts -- Mandatory). (True/False) Write True or False to the LEFT of the statement.

1. If a class A implements an interface B, but A provides a public method not mentioned in B, then this will cause a error or warning in Java. **FALSE: interface defines minimum set of public methods**
2. If you declare two classes in one file, one must be declared as `public` and the other declared as `private`. **FALSE: You can not declare a file as private at the file level**.
3. The scope of a field in a class (e.g., a class member which is an integer variable) is from the point of the definition to the end of the closest enclosing braces {...}. **FALSE: this is the rule for local variables; fields are visible throughout the class.**
4. The lifetime of a static member of a class is from the beginning of program execution until the end of program execution. **TRUE**
5. If you declare a local variable inside a method with the `private` keyword, it will be accessible only in the same method. **FALSE: You can't use "public/private" on local variables.**
6. If class Car contains a method `getMileage()` and you see a reference `Car.getMileage(),` then you know that `getMileage()` must be declared as `static` inside Car. **TRUE**
7. If you do not write `public` or `private`, the member or class will have default access, which means it can only be accessed from a class defined in the same file. **FALSE: default is access in the same folder.**
8. A queue is a FIFO data structure. **TRUE**
9. A recursive definition or method can have at most one base case, but can have any number of recursive cases. **FALSE: Can have any number (other than zero) of either.**
10. I have put my name on the top of this exam.

Problem Two. Show the output from the following recursive method when called on the linked list

Head → | 5 | → | 2 | → | 9 | → | 7 | ∘ |

```
public void displayNumbers(Node p) {
    if( p == null )
        System.out.println("---");  // just draw a line
    else
        System.out.println(p.item); // print out the number
        displayNumbers(p.next);
        displayNumbers(p.next);
    }}
```

Here is a printout with each recursive level indented:

```
5
 2
  9
   7
    ---
    ---
   7
    ---
    ---
  9
   7
    ---
    ---
   7
    ---
    ---
 2
  9
   7
    ---
    ---
   7
    ---
    ---
  9
   7
    ---
    ---
   7
    ---
    ---
```

Problem Three. There are two parts to this problem. (1) List all of the elements that could be found by Binary Search in the following list after **exactly three comparisons** (e.g., had I said "one comparison" you would simply list "8"):

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
# comps:	4	3	4	2	4	3	4	1	4	3	4	2	4	3	4

Answer: 2 6 10 14

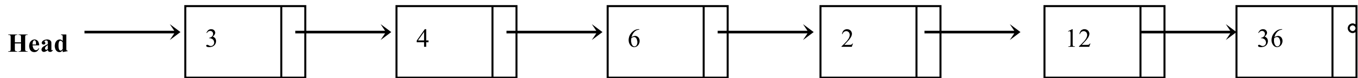
(2) What is the worst-case (in terms of the number of comparisons) for this list? Give an exact number for this particular list.

Maximum over all is 4 comparisons; searching for element not in list would also be 4.

Problem Four. Write a recursive function

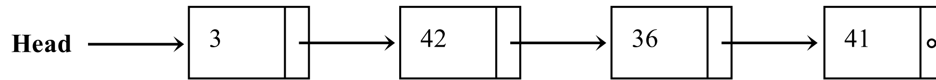
Node everyOther(Node p) { ...... }

which takes a linked list of integers and returns a new list consisting of every other element in the list, starting with the first, i.e., if we apply it to the following list we would get back a list -> 3 -> 6 -> 12 -> . It should work no matter how many nodes are in the list, including if the list is empty. Your algorithm should not destroy the original list.




```
static Node everyOther( Node p ) {
    if( p == null )
      return null;
    else if( p.next == null )
      return new Node( p.item, null );
    else {
      Node q = new Node( p.item, null );
      q.next = everyOther( p.next.next );
      return q;
    }
  }
```

Problem Five. Consider the following example of a linked list:

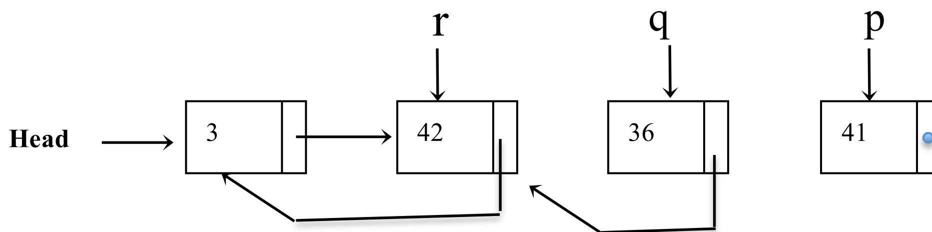


Apply the following algorithm to this list, and stop at the end of the loop when p is pointing to the node containing 36 and stop and draw the data structure and all pointers.

```
public static Node mystery(Node h) {
      if(h == null)
         return null;
      Node p = h.next;
      Node q = h;
      Node r = null;
      while ( p != null ) {
         r = q;
         q = p;
         p = p.next;
         q.next = r;
         if(p.item == 36)
            Stop and draw the data structure and the pointers p, p, and r
      }
      h.next = null;
      return q;
}

// called like this:

head = mystery( head );
```



Problem Six. This question is about Merge Sort and is in three parts, weighted equally.

Part One. Give a "best case" input for Merge Sort of size 8 (using the numbers 1 – 8), sort it using MS, and count the number of comparisons used.

Answer: We will underline the comparisons

$$| \; 1 \; | \; 2 \; | \; 3 \; | \; 4 \; | \; 5 \; | \; 6 \; | \; 7 \; | \; 8 \; | \qquad \text{Comparisons}$$

$$| \; \underline{1} \quad 2 \; | \; \underline{3} \quad 4 \; | \; \underline{5} \quad 6 \; | \; \underline{7} \quad 8 \; | \qquad 4$$

$$| \; \underline{1} \quad \underline{2} \quad 3 \quad 4 \; | \; \underline{5} \quad \underline{6} \quad 7 \quad 8 \; | \qquad 4$$

$$| \; \underline{1} \quad \underline{2} \quad \underline{3} \quad \underline{4} \quad 5 \quad 6 \quad 7 \quad 8 \; | \qquad 4$$

Total: 12

Part Two. Consider the following input list for Merge Sort:

8 1 5 2 6 3 7 4

Sort this list using MS and count the number of comparisons.

Answer: We will underline the comparisons

$$| \; 8 \; | \; 1 \; | \; 5 \; | \; 2 \; | \; 6 \; | \; 3 \; | \; 7 \; | \; 4 \; | \qquad \textbf{Comparisons}$$

$$| \; \underline{1} \quad 8 \; | \; \underline{2} \quad 5 \; | \; \underline{3} \quad 6 \; | \; \underline{4} \quad 7 \; | \qquad 4$$

$$| \; \underline{1} \quad \underline{2} \quad \underline{5} \quad 8 \; | \; \underline{3} \quad \underline{4} \quad \underline{6} \quad 7 \; | \qquad 6$$

$$| \; \underline{1} \quad \underline{2} \quad \underline{3} \quad \underline{4} \quad \underline{5} \quad \underline{6} \quad \underline{7} \quad 8 \; | \qquad 7$$

Total: 17

Part Three. Is the previous example a worst-case for Merge Sort in terms of the number of comparisons? Why or why not?

Answer: Yes, this is the worst-case, since every merge compares all but the last number, which is always just copied down without comparison.

Problem Seven. This problem is about the stability of Selection Sort and is in two parts. Consider the following implementation of Selection Sort, which is NOT stable.

```
public static void selectionSort(int[] a) {
   int N = a.length;
   for (int i = 0; i < N; i++) {
      int min = i;
      for (int j = i+1; j < N; j++) {
         if (a[j] <= a[min])
            min = j;
         }
      exch(a, i, min);       // swap the values in a[i] and a[min]
   }
}
```

Part One. There are two problems with stability in this algorithm. Describe both of them and give an example of each (using short lists, say of size 4).

Answer: The first problem is that the condition (a[j] <= a[min]) will replace min by j if it finds another element equal to the current minimum, which means that when there are duplicates, the minimum will be the last occurrence. Thus, in 1 2 2' 4, in the second iteration of the outer for loop, 2' will be chosen as the minimum, and swapped with 2 to get 1 2' 2 4. This is obviously instable.

The second problem is that the exch in the last line is a "long-range swap" and will move the top of the unsorted part of the list down, potentially past duplicates. Thus, in 4 2 4' 1, the 1 will be found to be the minimum, and swapped with the 4 to get 1 2 4' 4, again producting instability.

Part Two. Describe how to fix both of the problems in the algorithm. (You need not write code for this, but need to precisely describe the way to fix the problem, with a diagram or with words. Feel free to write code, of course!)

Answer: To fix the first problem , simply replace "<=" by "<" (as in the original algorithm. To fix the second problem, you must "bubble up" the minimum using swaps of adjacent elements. You can do this by replacing the last line by

```
for(int k = min; k > i; --k)
    exch(a, k, k-1);
```

Problem Eight. How many times does the following code print out "X"? Give your answer as a function of N in terms of Θ(….). [Hint: Do each loop separately, and give your answer to each; I'll grade for partial credit the best I can!]

```
for( int i = 0; i < N; ++i)                  //   N = Θ( N )
 for( int j = N; j > 0; j = j / 2 )          //   Log(N) + 1 = Θ( Log(N) )
  for( int k = 1`; k < (2 * N);  ++k )       //   2*N - 1 = Θ( N )
   for( int m = 0; m < 10000000 ++m)         //  1000000 = Θ( 1 )
     System.out.println("X");
```

Multiplying these out gives you $\Theta(N^2 Log(N))$

Problem Nine (15 pts -- Mandatory). Consider two implementations of a queue: (1) using a linked list, and (2) using a ring buffer with array resizing (both expansion and contraction, as we did in HW 02). You want to create three methods: enqueue(....), dequeue(), and member(....), the last of which simply scans through the queue (which is essentially a kind of unordered list) to look for a particular key and returns true or false. You implement these using the **most efficient techniques** we have learned this term.

Supposing you are measuring the number of times you "touch" an element of the array (i.e., compare it or access it), give (i) a **worst-case scenario** for each method in terms of what the queue looks like and what the operation does, and (ii) the **Θ(…….) worst-case time complexity** of the method as a function of **N**, where **N** is the number of elements in the queue.

Note: "Most efficient techniques" means that you would implement the linked-list queue with a "last" pointer, as described in class, so that you can add an element to the end of the list without scanning through. For dequeue, there is no issue, as the deletes occur at the beginning. The best implementation of the queue with resizing arrays would be what you implemented in HW 03, where a collapse is triggered when the array is ¼ full.

	enqueue(...)	**dequeue()**	**member(...)**
1. Queue implemented as a linked list	Worst-case Scenario: Any insert will just insert at end, so all cases are the same (see note below).	Worst-case Scenario: Any removal will just occur at beginning, so all cases are the same (see note below).	Worst-case Scenario: Element is in last place you look, so have to scan all N elements
	$\Theta(1)$	$\Theta(1)$	$\Theta(N)$
2. Queue with array resizing	Worst-case Scenario: Array is full and insert triggers an expansion, so all N elements must be copied over.	Worst-case Scenario: Array has N/4+1 elements, so removal triggers a contraction, and all N/4 elements must be copied over.	Worst-case Scenario: Same as for above.
	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$