Name: Sample Solution

Email address: ______________________

CSE 373 Autumn 2010: Midterm #2

(closed book, closed notes, NO calculators allowed)

Instructions: Read the directions for each question carefully before answering. We may give partial credit based on the work you **write down**, so if time permits, show your work! Use only the data structures and algorithms we have discussed in class or that were mentioned in the book so far.

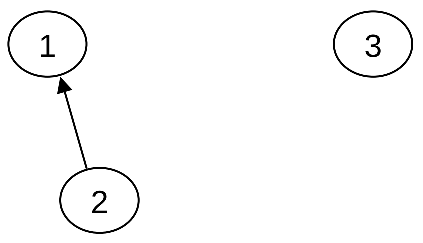
Note: For questions where you are drawing pictures, please circle your final answer for any credit.

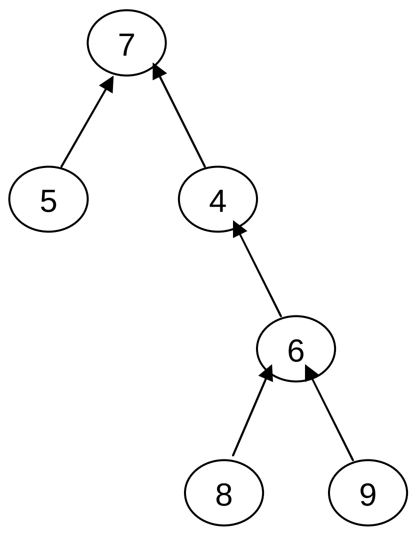
Good Luck!

Total: 61 points. Time: 50 minutes.

Question	Max Points	Score
1	11	
2	10	
3	10	
4	6	
5	8	
6	16	
Total	61	

1) [11 points total] Disjoint Sets: Given the array representation of up-trees discussed in class (where `size[x]` = total number of values in set x, and `up[x]` == -1 indicates that node x is a root node), for example (`size` is unspecified for non-root nodes):





	1	2	3	4	5	6	7	8	9
up	-1	1	-1	7	7	4	-1	6	6
size	2		1				6		

a) [6 points] Fill in the code below for **Union by Size** (weighted Union) (does not have to be perfect Java code). *(On a tie in size, the set referred to by the second parameter should be added to the set referred to by the first.)*

```
int up[N];
int size[N];

void Union(int x, int y) {
     // Assuming x and y are roots of two different sets,
     // fill in code to implement union by size (weight)
     // Break ties in size by adding the set y
     // to the set x.
     int wx = size[x];
     int wy = size[y];

     if (wx < wy) { // x should be added to y
          up[x] = y;
          size[y] = wx + wy;
     } else { // y should be added to x
          up[y] = x;
          size[x] = wx + wy;
     }
}
```

1) (cont)
b) (1 pts) Now, **draw** the **set of uptrees** that results from executing a **`find(9)` with path compression** on the sets shown in on the previous page. You may update the set of uptrees on the previous page or redraw the sets here. If no changes occur, state that.

$$\begin{tikzcd}
1 & 3 & & 7 & & \\
2 \arrow[u] & & 5 \arrow[ur, bend left] & 4 \arrow[u] & 6 \arrow[ul, bend right] & 9 \arrow[ull, bend right] \\
& & & & 8 \arrow[u] &
\end{tikzcd}$$

c) (2 pts) Update the **array** to reflect the changes (if any) caused by that find operation. You may update the **array** on the previous page or redraw the array here. If no changes occur, state that.

	1	2	3	4	5	6	7	8	9
up	**-1**	**1**	**-1**	**7**	**7**	**7**	**-1**	**6**	**7**
	2		**1**				**6**		

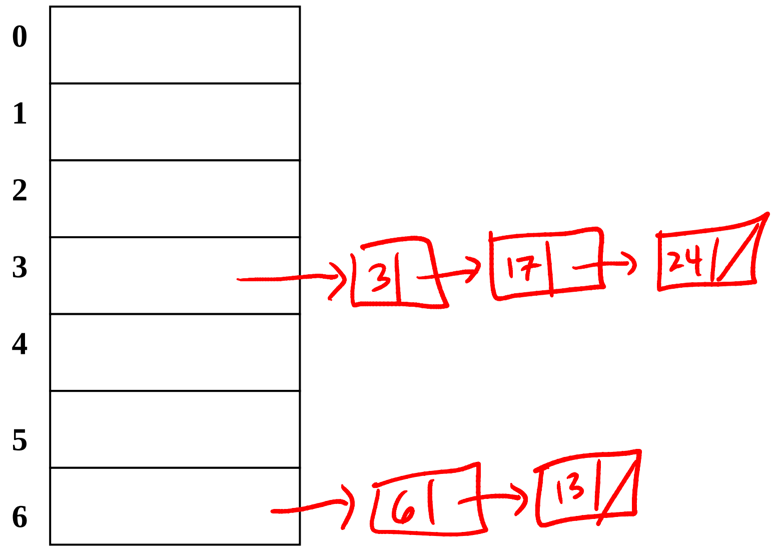
d) (2 pts) What is the worst case running time of a single find operation if union by size and path compression are used? N = total # of elements in all sets. (no explanation required)

O(log N)

2) [10 points total] Hashing: Draw the contents of the two hash tables below. Show your work for partial credit. *If an insertion fails, please indicate which values fail and attempt to insert any remaining values*. The size of the hash table is 7. The hash function used is H(k) = k mod 7

13, 17, 6, 24, 3

a) Separate chaining, where each bucket points to a sorted linked list.



b) Double hashing, where H2(k) = 7 – (k mod 5)

0	31
1	
2	24
3	17
4	
5	6
6	13

H2(6) = 7 – (6 mod 5) = 7 – 1 = 6

H2(24) = 7 – (24 mod 5) = 7 – 4 = 3

H2(3) = 7 – (3 mod 5) = 7 – 3 = 4

2) (cont)
c) (1 pt) What is the **load factor** for the table a)?

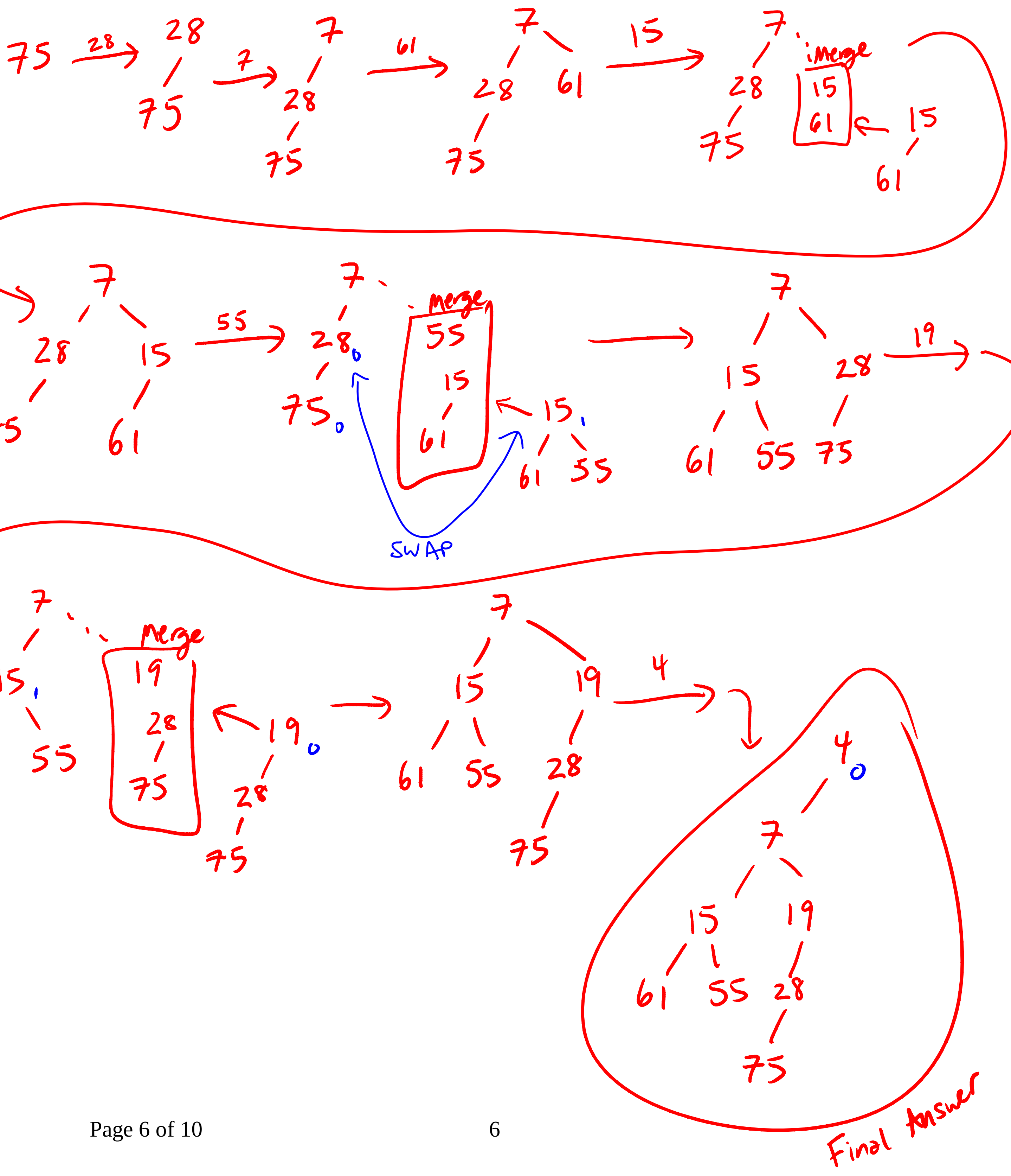
5/7

d) (1 pt) What is the **load factor** for the table b)?

5/7

3) [10 points total] Leftist Heaps:

a) [8 pts] Draw the ***leftist*** heap that results from inserting: 75, 28, 7, 61, 15, 55, 19, 4 in that order into an initially empty heap. You are only required to show the final heap, although if you draw intermediate heaps, ***please circle your final result for ANY credit.***



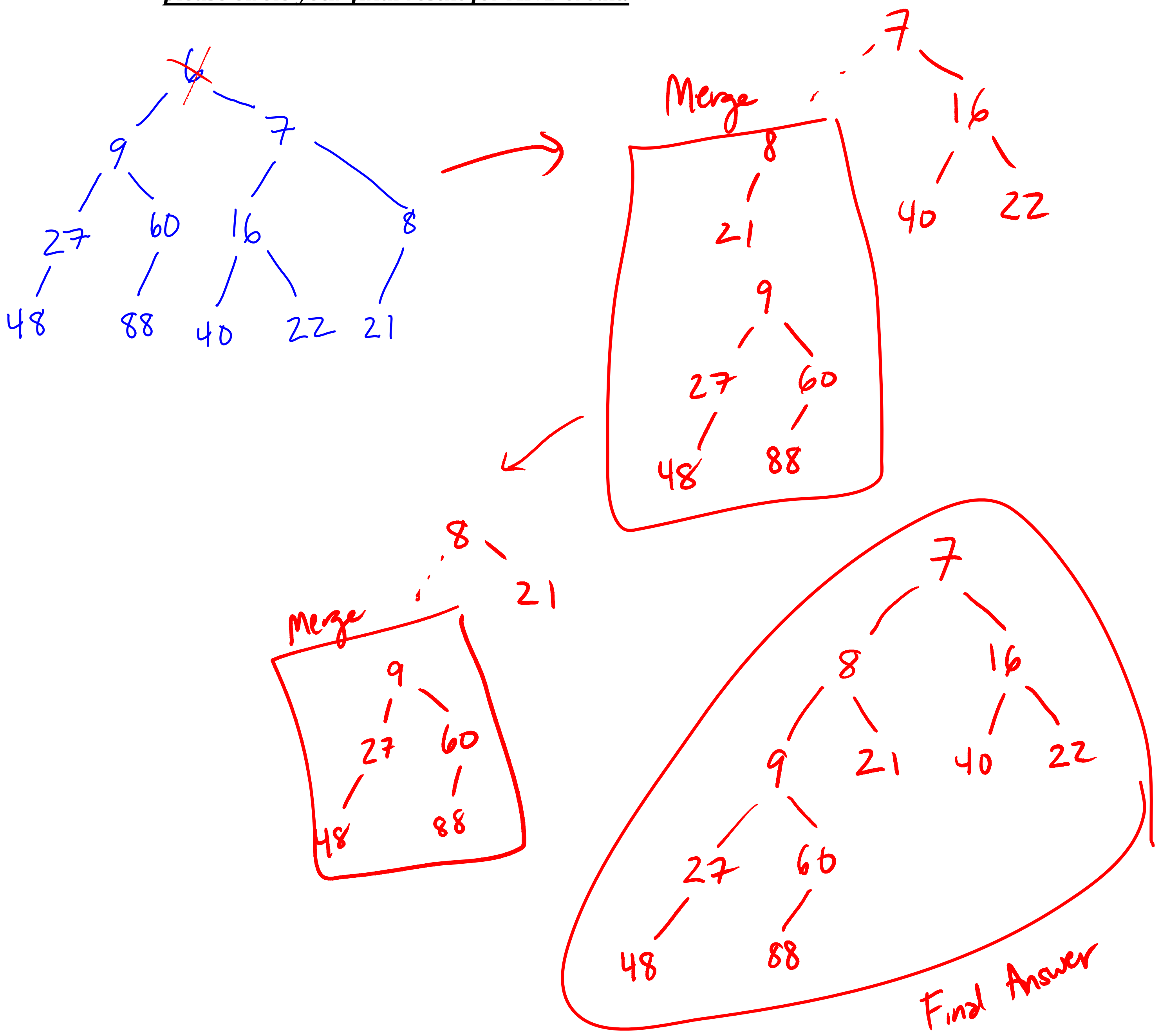
3) (cont)

b) [2 pts] What is the null path length of the **root** in your final heap from part a)?

0

4) [6 points] Skew Heaps

Draw the skew heap that results from doing a **deletemin** on the skew heap shown below. You are only required to show the final tree, although if you draw intermediate trees, ***please circle your final result for ANY credit.***



5) [8 points] Memory Hierarchy & Locality: Examine the code example below:

```
c[1] = 5;
z = 68;
j = 2;
while (j < 2000)  {
     a[j] = a[j] + a[j-1];
     z = b[10] + z;
     System.out.println("d[j] = " + d[j]);
     j = j + 1;
}
f[25] = 60;
y = c[2000] + c[2001] + c[2002] + c[2003]
```

Considering only their use in the code segment above, for each of the following variables, indicate below what type of locality (if any) is demonstrated. Please circle ***all that apply*** (you may circle more than one item for each variable):

a	[✓ spatial locality]	[✓ temporal locality]	no locality
b	spatial locality	[✓ temporal locality]	no locality
c	[✓ spatial locality]	temporal locality	no locality
d	[✓ spatial locality]	temporal locality	no locality
f	spatial locality	temporal locality	[✓ no locality]
j	spatial locality	[✓ temporal locality]	no locality
y	spatial locality	temporal locality	[✓ no locality]
z	spatial locality	[✓ temporal locality]	no locality

6) [16 points total] Running Time Analysis:

- **Describe the most time-efficient way to implement the operations listed below.** Assume no duplicate values and that you can implement the operation as a member function of the class – with access to the underlying data structure.
- Then, give the tightest possible upper bound for the ***worst case*** running time for each operation in terms of *N*. ****For any credit, you must explain *why* it gets this worst case running time.** You must choose your answer from the following (not listed in any particular order), each of which could be re-used (could be the answer for more than one of a) -f)).

$O(N^2)$, $O(N^{½})$, $O(N \log N)$, $O(N)$, $O(N^2 \log N)$, $O(N^5)$, $O(2^N)$, $O(N^3)$,
$O(\log N)$, $O(1)$, $O(N^4)$, $O(N^N)$, $O(N^6)$, $O(N (\log N)^2)$, $O(N^2 (\log N)^2)$

a) DecreaseKey(k, v) on a **binary min heap** containing N elements. Assume you have a reference to the key k. v is the amount that k should be decreased. **Explanation:**

a) $\boxed{O(\log N)}$

Using the reference to k, decrement its value by v (takes O(1)). Then percolate k up by comparing to its parent and swapping if necessary. In the worst case k is at the bottom row of the heap and has been decremented to become the smallest value in the heap and thus must move up O(log N) levels. Thus total running time is O(log N).

b) Inserting an element in a **hash table** containing N elements where separate chaining is used and each bucket points to a sorted linked list. The table size = N. **Explanation**:

b) $\boxed{O(N)}$

To insert, do the hash function to find the bucket, band then search through the sorted linked list until you find a value greater than the value to be inserted or reach the end of the list. Insert the value at that point in the list. In the worst case all elements have been hashed to the same bucket, and the value you are inserting is the largest value in the hash table, causing you to search through all N values before adding the value at the end of the list. Finding the bucket takes O(1) but searching to the end of the sorted linked list takes O(N), so total running time is O(N).

c) Converting a **4-heap** (a d-heap where d=4) containing N elements into a **binary min heap**. **Explanation:**

c) $\boxed{O(N)}$

Unfortunately a 4-heap is not guaranteed to be a 2-heap. To convert a 4-heap to a binary min heap just run Floyds buildheap method on the 4-heap array. Buildheap takes O(N) since the sum of the heights that must be percolated down is O(N).

d) Rehashing values from a **hash table** containing 2N elements (where separate chaining is used and each bucket points to a sorted linked list). The original hash table size = N, the new hash table is size 2N. **Explanation:**

d) $\boxed{O(N^2)}$

To rehash, go through all N buckets of the original hash table. For each bucket, go through each element in the bucket. For each element, recalculate the hash function to find the bucket in the new table, and insert the value into the new table. Examining all buckets and all linked list nodes in the original hash table takes time N to examine each bucket and time 2N to examine each element = O(N). You then have to reinsert each value. In the worst case all 2N values hash to the same bucket in both the original and the new hash table. If you read the values in original hash table lists in ascending order, and thus insert in ascending order (always at end of the list), then this means the inserts will take O(1) to calculate the hash function, but O(N) to insert into the linked list for each value, resulting in $O(N^2)$ for the inserts.