Name: Sample Soln

Email address: ______________________

CSE 373 Winter 2012: 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: 65 points. Time: 50 minutes.

Question	Max Points	Score
1	12	
2	12	
3	6	
4	7	
5	11	
6	8	
7	9	
Total	65	

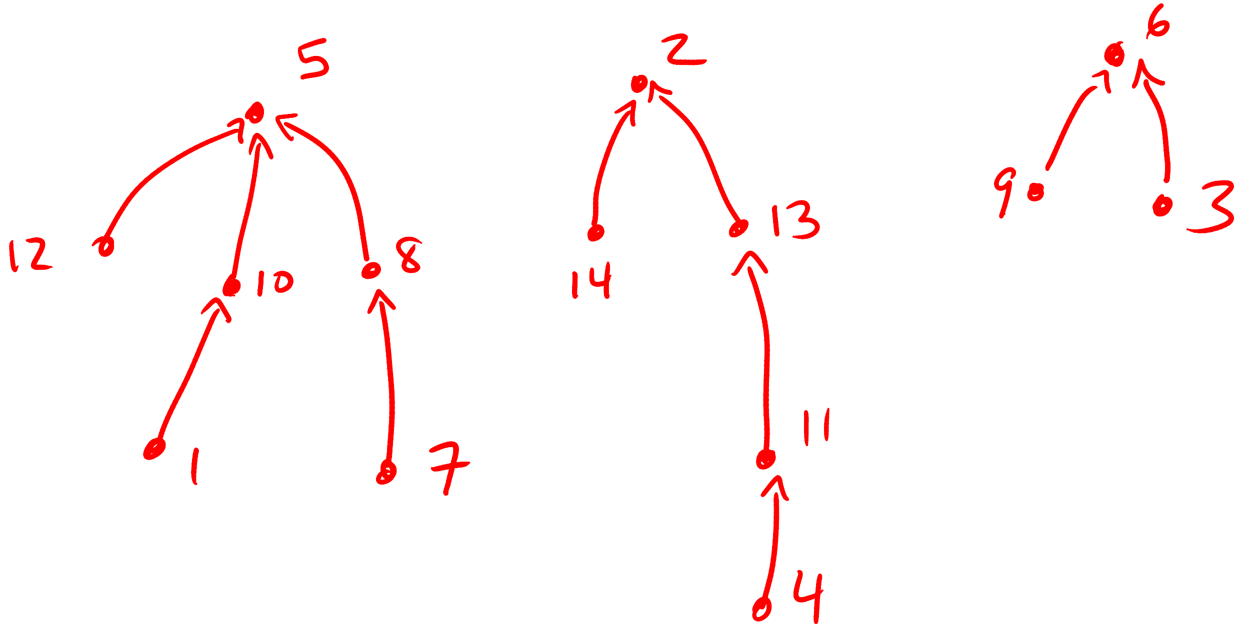
1) [12 points total] Disjoint Sets

The uptrees used to represent sets in the union-find algorithm can be stored in two *n*-element arrays. The **`up`** array stores the parent of each node (or -1 if the node has no parent). The **`weight`** array stores the number of items in a set (its weight) if the node is the root (representative node) of a set. (If a node is not a root the contents of its location in the **`weight`** array are undefined – we don't care what value it holds, it can be zero or any other number.)

The following shows a collection of sets containing the numbers 1 through 14, without the **weight** array filled in:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`up`	~~**10**~~ 5	~~**-1**~~ 5	**6**	**11**	**-1**	**-1**	**8**	**5**	**6**	**5**	~~**13**~~ 2	**5**	**2**	**2**
`weight`					11	3								

a) **[3 points] Draw** a picture of the uptrees represented by the data in the **`up`** array shown above.

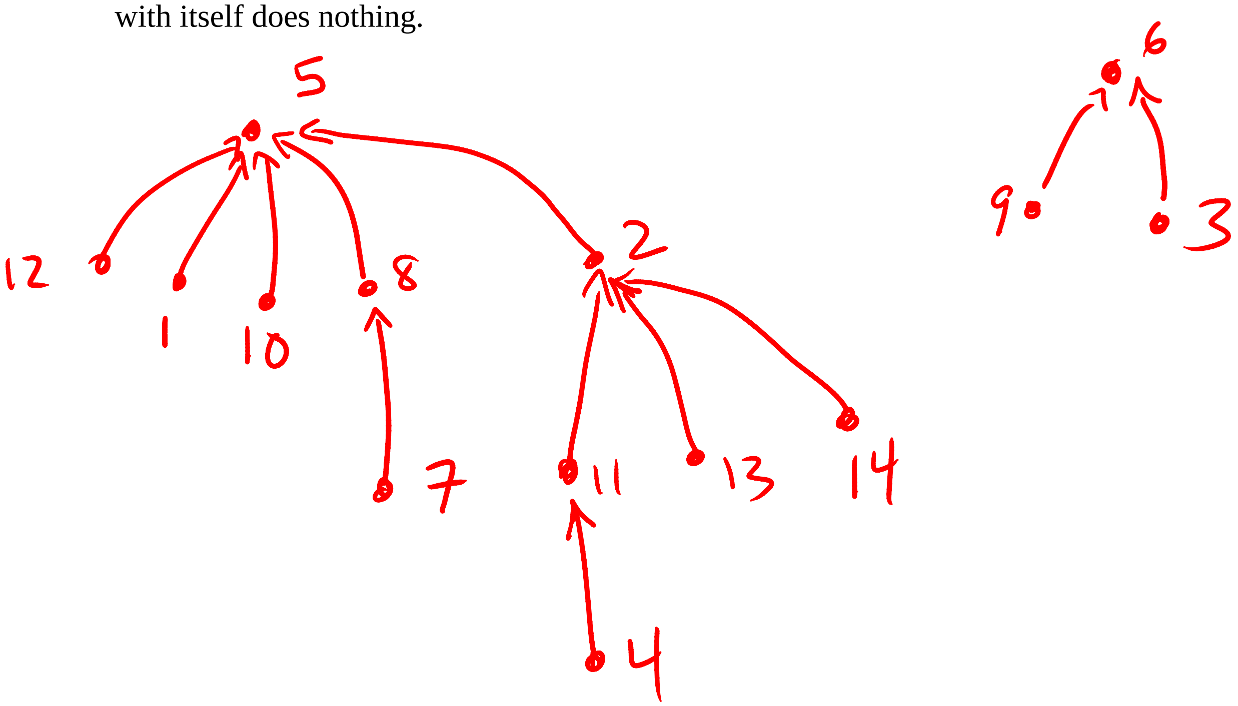


1) (cont)

b) **[3 points]** Now, draw a new set of uptrees to show the results of executing:

```
union(find(1), find(11));
find(9);
```

Regardless of how the trees from part a) were constructed, here assume that find uses **path compression** and that union uses **union-by-size (aka union by weight)**. In case of ties in size, always make the higher numbered root point to the lower numbered one. Unioning a set with itself does nothing.



c) **[2 points]** Update the **`up`** and **`weight`** arrays at the top of the previous page to reflect the picture after part b). That is, fill in the contents of the **`weight`** array and update the contents of the **`up`** array.

d) **[2 points]** What is the worst case big-O running time of a single **find** operation if union by size (aka union by weight) and path compression are used (~~assuming you are always passed roots as parameters~~)? N = total # of elements in all sets. **(no explanation required)**

$O(\log N)$

e) **[2 points]** Assuming that you are using union by size and path compression, how long would we expect a sequence of N-1 union operations and J find operations to take? (N = total # of elements in all sets) Express your answer in terms of big-O. **(no explanation required)**

$O(N - 1 + J)$

2) [12 points total] Hashing:

a) [8 points] Draw the contents of the two hash tables below after inserting the values shown. Show your work for partial credit. *If an insertion fails, please indicate which values fail and attempt to insert any remaining values*. The hash function used is H(k) = k mod tablesize.

Table 1: **Separate chaining**, (where each bucket points to a linked list *sorted from smallest to largest*)

Insert: 14, 23, 2, 19, 20, 5

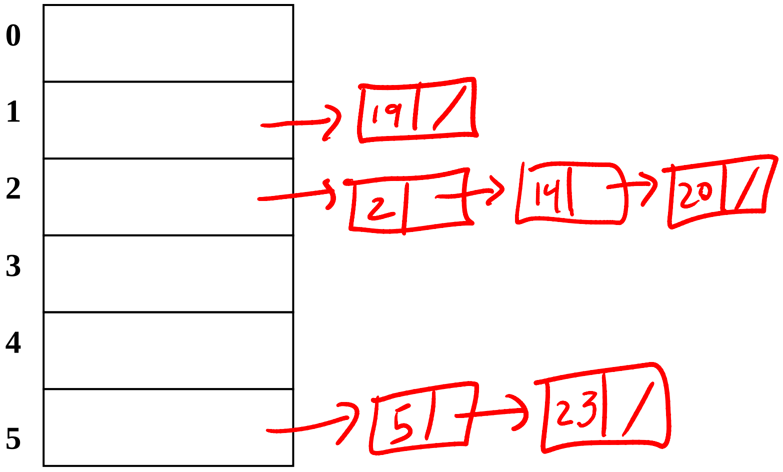
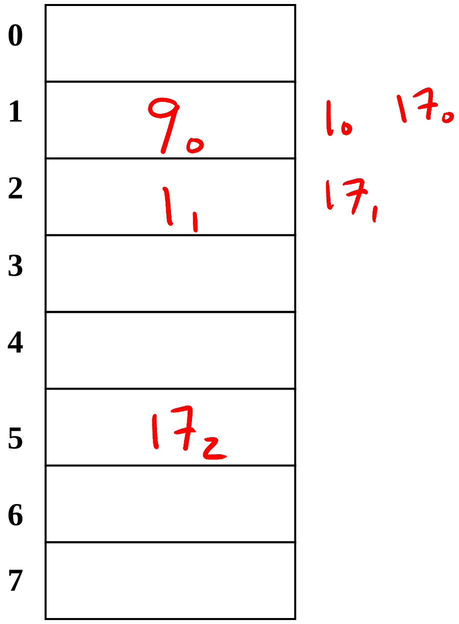


Table 2: **Quadratic Probing**

Insert: 9, 1, 17



b) [2 points] Give the load factor for each table:

Load factor for Table 1:

6/6 = 1

Load factor for Table 2:

3/8

2) (cont)

c) **[1 point]** Table 1 will (circle one):

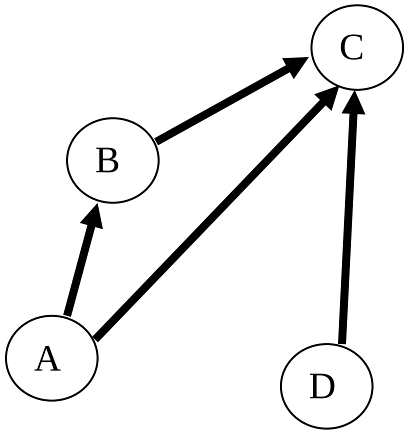
[✓ i. gradually degrade in performance as more values are inserted]
ii. possibly fail to find a location on the next insertion
iii. be fine on the next insertion, but may fail to find a location on any insertions after that
iv. none of the above

d) **[1 point]** Table 2 will (circle one):

i. gradually degrade in performance as more values are inserted
[✓ ii. possibly fail to find a location on the next insertion]
iii. be fine on the next insertion, but may fail to find a location on any insertions after that
iv. none of the above

3) [6 points total] Graphs

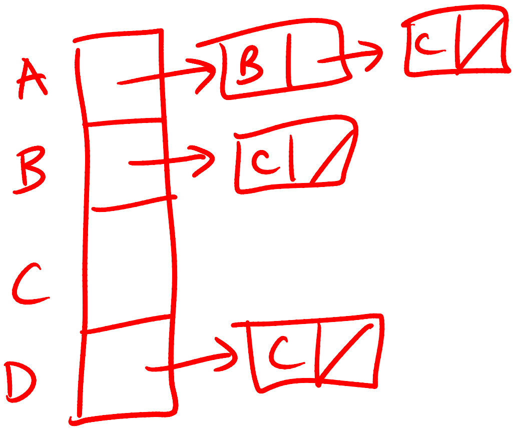
a) **[2 points]** Draw both the adjacency matrix and adjacency list representations of this graph. *For this problem, assume there are no implicit self loops* (e.g. an edge from A to A). That is, unless there is a self loop explicitly drawn in the graph, there should not be one in the representation.



Adjacency Matrix:

	A	B	C	D
A	F	T	T	F
B	F	F	T	F
C	F	F	F	F
D	F	F	T	F

Adjacency List:



What is the worst case big-O running time of the following operations (use V and E rather than N in your answers). **No explanation is required**.

b) **[2 points]** Find the *out-degree* of a single vertex whose graph is stored in an adjacency matrix.

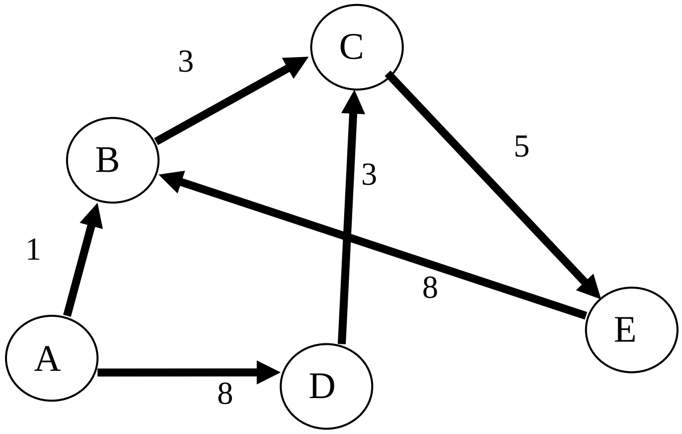
$O(V)$

c) **[2 points]** Find the *in-degree* of a single vertex whose graph is stored in an adjacency list.

$O(V+E)$

4) [7 points total] Graphs

Use the following graph for the questions _on this page_:



a) **[2 points]** If possible, list ***two*** valid topological orderings of the nodes in the graph above. If there is only one valid topological ordering, list that one ordering. If there is no valid topological ordering, state why one does not exist.

None possible : cycle BCE

b) **[2 points]** What is the worst case big-O running time of topological sort for a graph represented as an adjacency list? (note this refers to the *un-optimized* version first presented in lecture, a queue is NOT used) (use V and E rather than N in your answer) **No explanation is needed**.

$O(V^2 + E)$

c) **[2 points]** This graph is: (Circle all that are true):

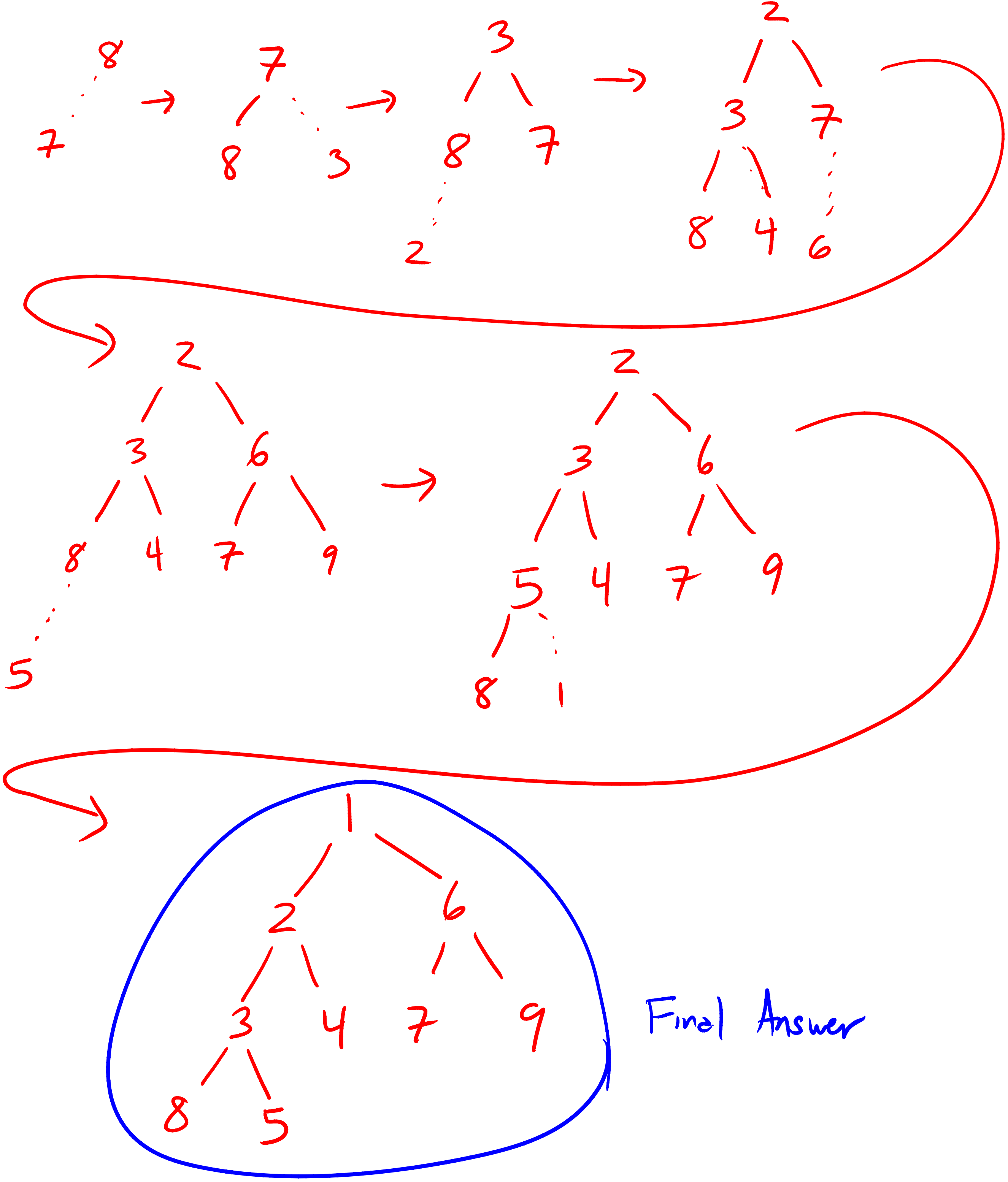
[✓ directed] [✓ weakly connected] undirected

complete acyclic strongly connected

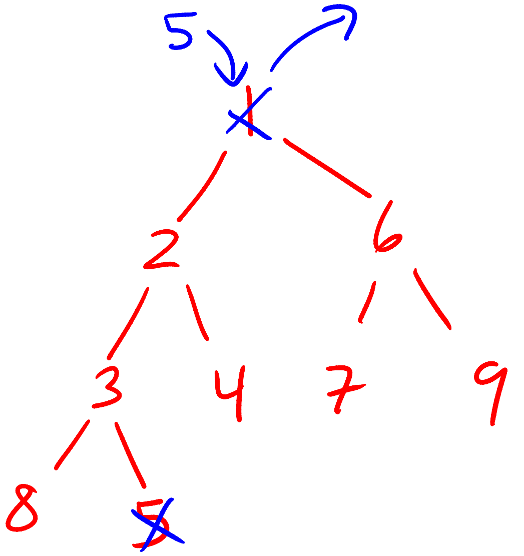
d) **[1 point]** What is the in-degree of node B? 2

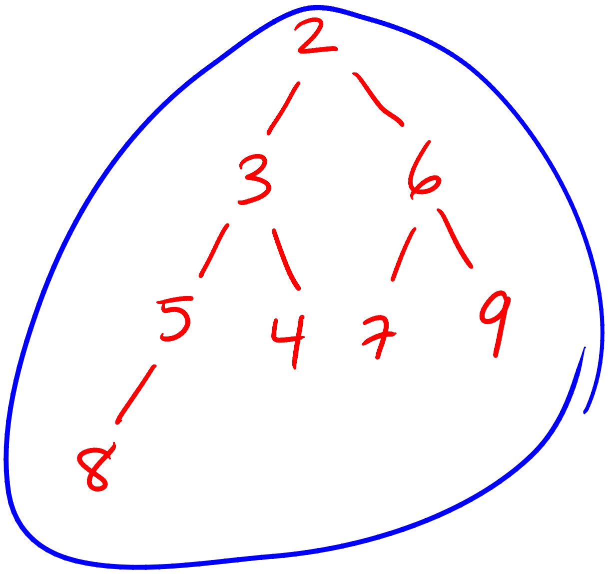
5) [11 points total] Heaps

a) **[6 points]** Draw the binary min heap that results from inserting **~~8, 7, 3, 2, 4, 6~~, 9, 5, 1** <u>in that order</u> into an initially empty binary min heap. *You do not need to show the array representation of the heap*. You are only required to show the final tree, although drawing intermediate trees may result in partial credit. If you draw intermediate trees, please **<u>circle your final result</u>** for any credit.



b) **[2 points]** Draw the result of one deletemin call on your heap drawn at the end of part (a).





Final Answer

c) **[3 points]** For large values of N, would you expect an 8-heap to have better or worse locality than a binary heap? (you may assume that they are both min heaps)

Circle the best answer: (circle <u>one</u> answer only)

i. An 8-heap would tend to have better spatial locality than a binary heap on an insert operation.

ii. A binary heap would tend to have better spatial locality than an 8-heap on an insert operation.

[✓ iii. An 8-heap would tend to have better spatial locality than a binary heap on a deletemin operation.]

iv. A binary heap would tend to have better spatial locality than an 8-heap on a deletemin operation.

v. I would expect the 8-heap and the binary heap to have very similar spatial locality.

6) [8 points] Memory Hierarchy & Locality: Examine the code example below:

```
a = 30;
w[2] = 36;
b = 14;
c = 98;
for (i = 1; i < 1000; i++) {
     a = y[i] + y[4];
     j = z[3] + b;
     c = c + x[i+1] + a;
}
```

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**
i	**spatial locality**	[✓ **temporal locality**]	**no locality**
w	**spatial locality**	**temporal locality**	[✓ **no locality**]
x	[✓ **spatial locality**]	**temporal locality**	**no locality**
y	[✓ **spatial locality**]	[✓ **temporal locality**]	**no locality**
z	**spatial locality**	[✓ **temporal locality**]	**no locality**

7) [9 points total] Running Time Analysis:

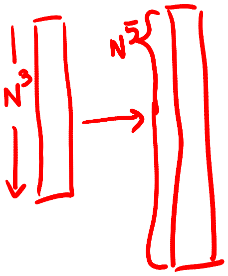
- **Describe the most time-efficient way to implement the operations listed below**. Assume no duplicate values and that you can implement the operation as a member function of the class – with access to the underlying data structure.
- Then, give the tightest possible upper bound for the ***worst case*** 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) -d)).

  $O(N^2)$, $O(N^{1/2})$, $O(N \log N)$, $O(N)$, $O(N^2 \log N)$, $O(N^5)$, $O(2^N)$, $O(N^3)$,
  $O(\log N)$, $O(1)$, $O(N^4)$, $O(N^6)$, $O(N^{15})$, $O(N (\log N)^2)$, $O(N^2 (\log N)^2)$

a) Given an open addressing **hash table** where linear probing is used to resolve collisions, what is the worst case run time of a rehash operation. Assume that original tablesize = N^3 (before re-hashing), new tablesize = N^5 and there are currently N items in the hash table.

a) $O(N^3)$

Explanation:



① scan all locations in orig hash table = $O(N^3)$

② For each of the N elements, rehash into new table

Assume hash function takes $O(1)$ time, but worst case is all values has to same bucket, take $1+2+3+\dots+N-1$ probes to resolve

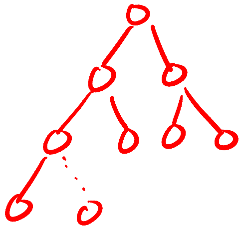
Total is still $O(N^2)$ (size of new hash table doesn't matter.

$\therefore$ Worst case is $O(N^3+N^2) \rightarrow O(N^3)$

b) Given a **binary min heap,** what is the worst case runtime of a single insert operation.

b) $O(\log N)$

Explanation:



① Insert value in next empty location in array $O(1)$

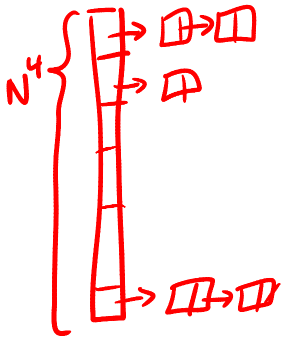
② Percolate Up: compare w. parent $O(1)$, do this a maximum of $O(\log N)$ times, because the height of a complete binary tree w. N elements is $O(\log N)$.

This only occurs if you have just inserted a value that is smaller than all other values in the heap (this is the worst case)

$\therefore$ Worst case is $O(\log N)$

c) Given a **hash table** that uses separate chaining where each bucket points to a linked list that is sorted from low to high, what is the worst case run time to find what the minimum value in the hash table is (you do not know what this value is ahead of time). Assume: tablesize N^4 and there are currently N items in the hash table. **Explanation:**

c) $O(N^4)$



① Scan the array to find the non-empty buckets (worst case: last bucket is non-empty + must visit all N^4 buckets)

② For each non-empty bucket, only need to examine the first value in its linked list - $O(1)$ and compare it to the current min value - $O(1)$. When have visited all non-empty buckets, you have found the min value. $\therefore O(N^4)$

(It will take longest if each value is in its own bucket → $O(N)$ comparisons)

Note: If you had an extra pointer that linked together all of the nodes in the linked list, in theory you could do this in $O(N)$