# CSE 373 Midterm Exam 17wi

Name: Solution Quiz Section: _____

TA: _____ Student ID #: _____

- This test is closed book, closed notes, closed calculators, closed electronics.
- You have 50 minutes to complete this exam. You will receive a deduction if you keep working after the instructor calls for papers.
- If you write your answer on scratch paper, please clearly write your name on every sheet and write a note on the original sheet directing the grader to the scratch paper. **We are not responsible for lost scratch paper or for answers on scratch paper that are not seen by the grader due to poor marking.**
- Code/Pseudocode will be graded on proper behavior/output and not on style, unless otherwise noted.
- Unless otherwise stated, all **logs are base 2.**
- If you enter the room, you must turn in an exam and will not be permitted to leave without doing so.

**Good luck and have fun!**

| # | Problem | Points | Score |
|---|---|---|---|
| 1 | Asymptotic Analysis | 6 | 6 |
| 2 | Pseudocode and Runtime Analysis | 14 | 14 |
| 3 | Heaps | 12 | 12 |
| 4 | AVL Trees | 12 | 12 |
| 5 | Hashing | 10 | 10 |
| 6 | Short Answer | 12 | 12 |
| 7 | Design Decisions | 8 | 8 |
| 8 | Abstraction | 6 | 6 |
| | **Total** | 80 | 80 |

**Advice:**

- Read questions carefully. Understand a question before you start writing.
- Write down assumptions, thoughts and intermediate steps so you can get partial credit. Clearly circle your final answer.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all of the questions.
- If you have questions, ask them. Any clarifications will be noted on the screen.

**1) Asymptotic Analysis (6 points)**

For each function $f(n)$ below, give an asymptotic upper bound using "Big-O" notation. You do not have to show your work or find a c and $n_0$ to prove your answer. **You should give the tightest and simplest bound possible.**

*Example:* f(n) = 5n would be O(n), not O(5n) or O($n^2$)

a) $f(n) = n(3n-2)\,2n$

$= (3n^2 - 2n) * 2n$

$= 6n^3 - 4n^2$

$O(n^3)$

b) $f(n) = \log(n) + n\log(n)$

$O(n \log(n))$

c) $f(n) = 5\log(n) + 12n^2 + 712$

$O(n^2)$

d) $f(n) = 24n^5 - 7n$

$O(n^5)$

e) $f(n) = 0.5\, n^{(1/2)}$

$O(n^{\frac{1}{2}})$

f) $f(n) = 4\, n! + 2^n$

$O(n!)$

**2) Pseudocode and Runtime Analysis (14 points)**

Describe the worst case running time of the following code in "Big-O" notation in terms of the variable *n*. You do not have to show your work or find a c and $n_0$ to prove your answer. **You should give the tightest bound possible.**

a)

```
void mysteryOne(int n) :
   for (i = 0; i < n; i++) : n
      for (j = n; j > 0; j--) : n
         for (k = 0; k < n * (5n + 1); k++) : n^2
           print "alright"
```

$O(n^4)$

b)

```
// data is an array of length n
void mysteryThree(int[] data, BinaryMinHeap heap) :
   for (i = 0; i < data.length; i++) : n
     heap.insert(data[i]); log(n)
    while (!heap.isEmpty()) : n
     System.out.println(heap.deleteMin()); log(n)
```

$O(n \log(n))$

c) Given a Node that is the root of an AVL tree, write **pseudocode** to find the minimum value. Your algorithm should be time-efficient. You can assume there is at least one value in the tree.

```
class Node {
   int data;
   Node left;
   Node right;
}
```

two solutions below:

(1)

```
int findMin(Node n):
    while(n.left != null):
        n = n.left
    return n.data
```

(2)

```
int findMin(Node n):
    if (n.left == null):
        return n.data
    else:
        return findMin(n.left)
```

Evaluate the runtime of this algorithm in terms of *n* where *n* is the number of Nodes in the tree:

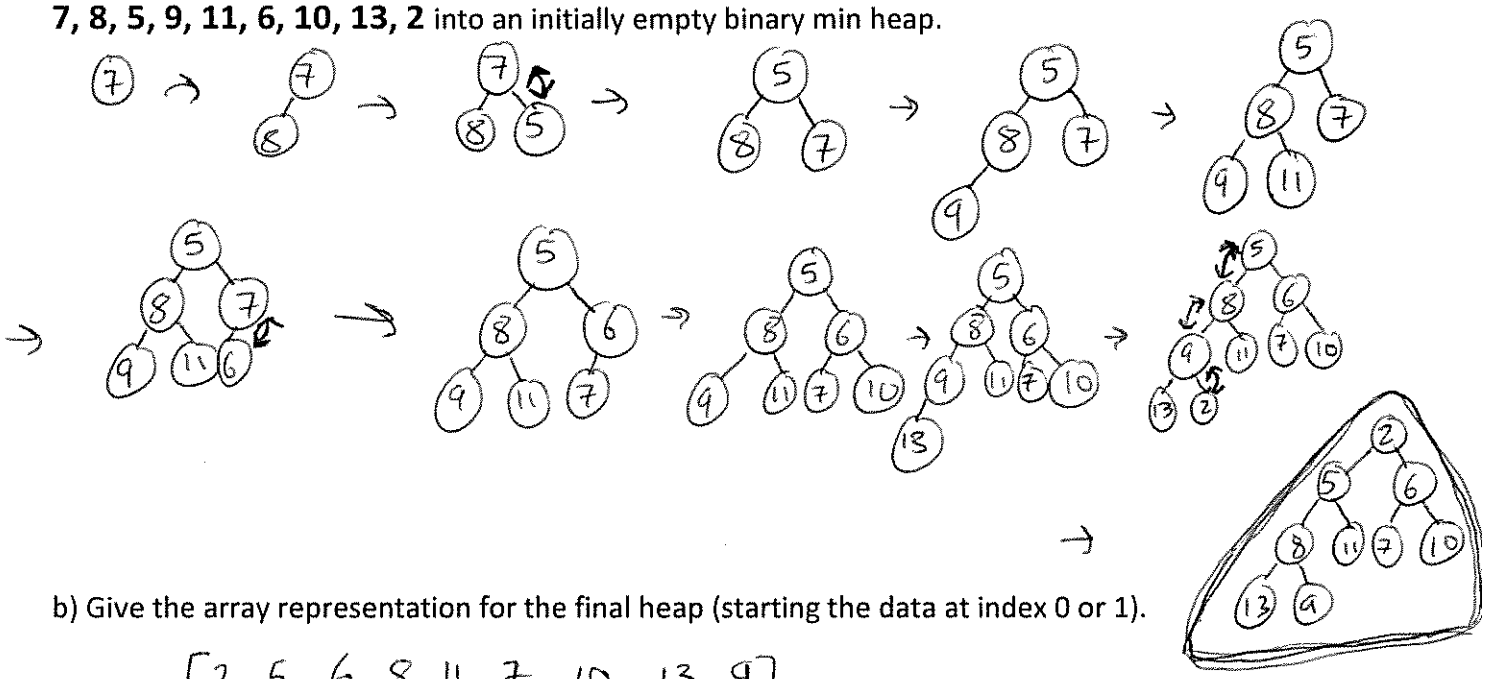
height of AVL tree is log(N) worst case.

$O(\log(n))$

**3) Heaps (12 points)**

You are only required to show the final heap, although if you draw the intermediate trees it may help us award partial credit. **Please circle your final answer for credit.**

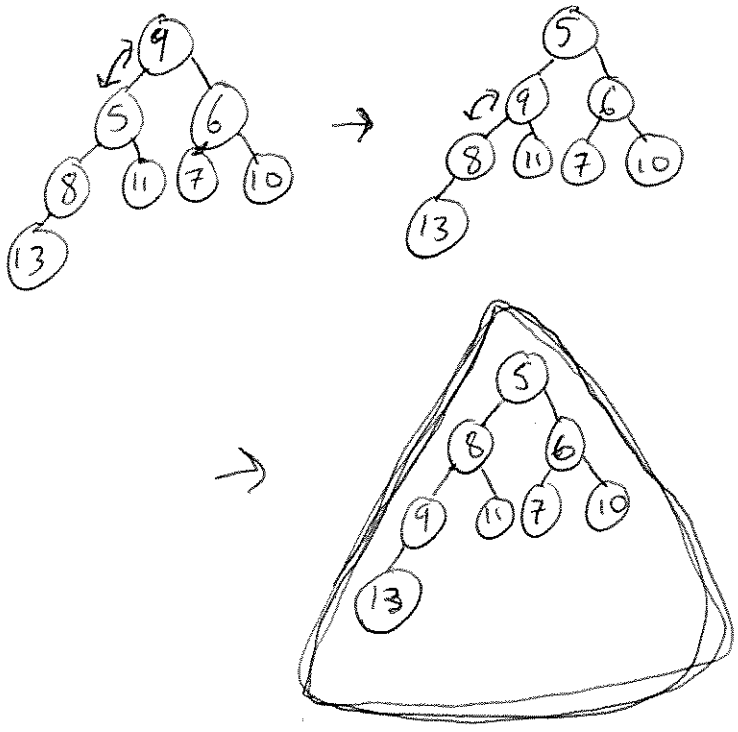
a) Draw the resulting binary tree representation from inserting the following data in this order: **7, 8, 5, 9, 11, 6, 10, 13, 2** into an initially empty binary min heap.



b) Give the array representation for the final heap (starting the data at index 0 or 1).

[2, 5, 6, 8, 11, 7, 10, 13, 9]

c) Perform one `deleteMin` operation on the final heap. **Show the heap after the operation and the value returned. Circle your final answer**

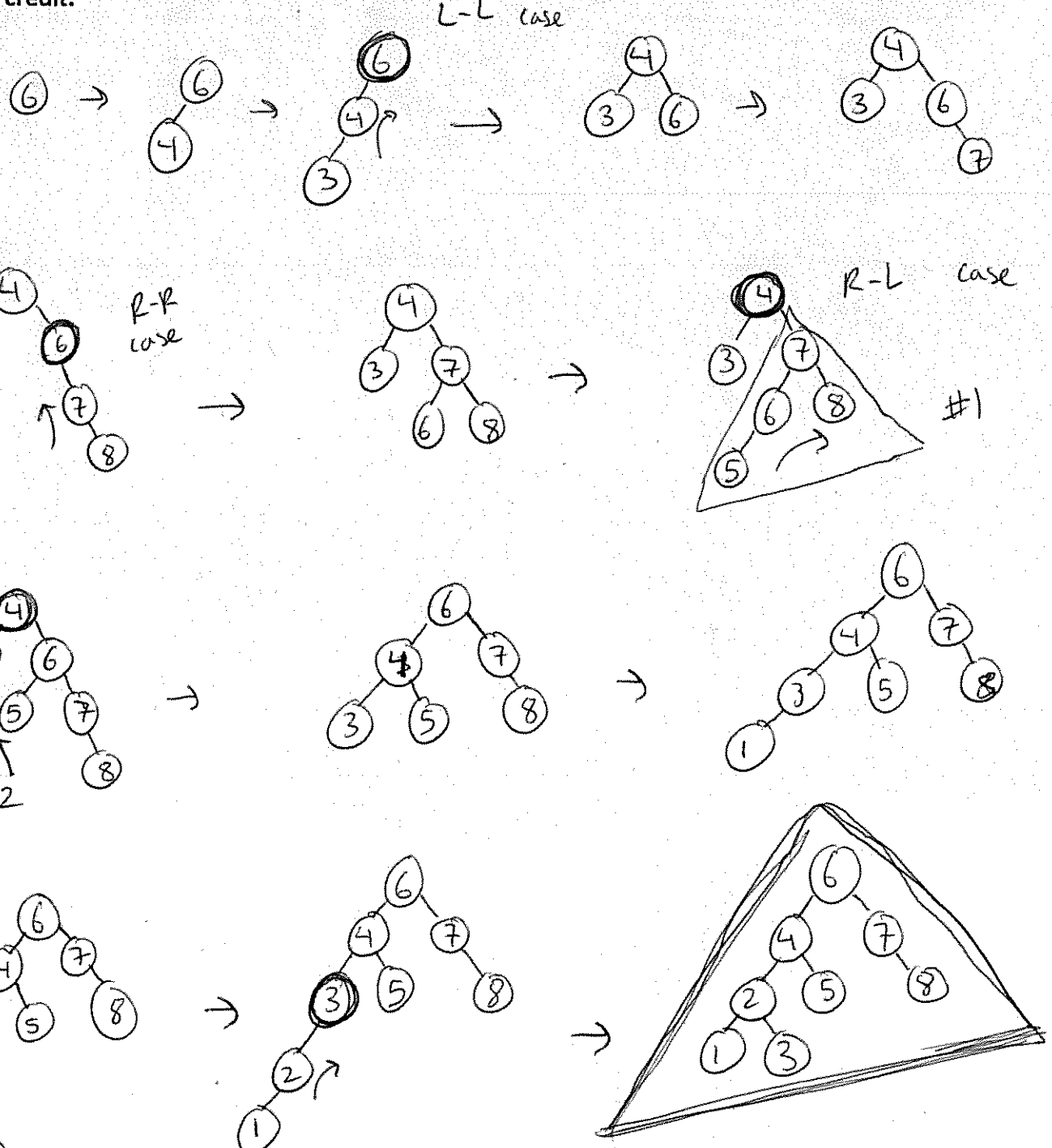


Value Returned:

| 2 |
|---|

**4) AVL Trees (12 points)**

Draw the AVL tree that results from inserting the **keys 6, 4, 3, 7, 8, 5, 1, 2** in that order into an initially empty AVL tree. You are only required to show the final tree, although if you draw intermediate trees it may help us award partial credit. **Please circle your final result for ANY credit.**



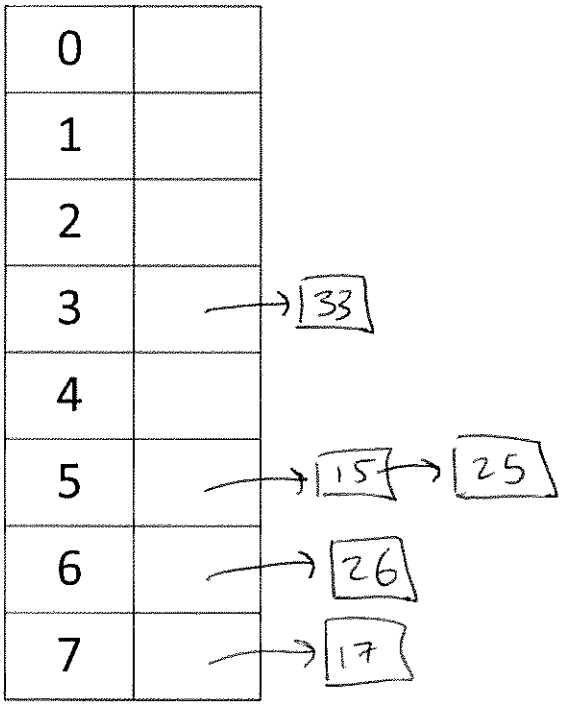
**5) Hashing (10 points)**

For each of the following versions of hash tables, insert the following elements in this order:

**33, 15, 17, 25, 26**

You should use the primary hash function h(k) = k % 10. You do not need to consider load factor or perform any rehashing. You do not have to consider efficiency of the operation when inserting the elements.

**a) Separate Chaining:**




**b) Linear Probing:**

(h(k) + i) % tableSize

| | |
|---|---|
| 0 | 26 |
| 1 | |
| 2 | |
| 3 | 33 |
| 4 | |
| 5 | 15 |
| 6 | 25 |
| 7 | 17 |

c) For linear probing, in class we discussed that "clustering" can happen. Explain in one sentence what is clustering and how does it affect the worst-case runtime of the `find` operation as the load factor approaches 1? You do not need to be specific with Big-O analysis.

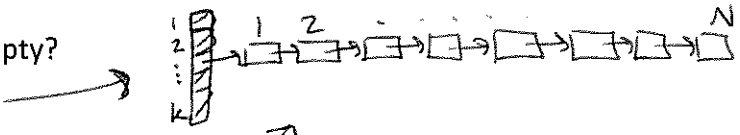
Clustering is when values can all hash to the same neighborhood, causing clusters of values to be probed when searching for a value or open space; this makes the runtime of find get worse: as the load factor approaches 1, the runtime becomes O(N).

**6) Short Answer (12 points)**

a) Consider a hash table with separate chaining with k buckets and N items currently in the table.

1) If N > k, is it possible that any buckets are empty?

Yes.

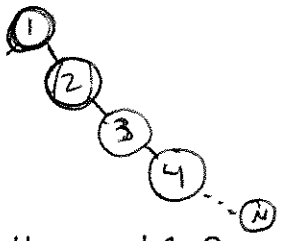



2) In the worst-case how many items could be in one bucket?

N

b) Every BST has a height of at worst log(N) where N is the number of nodes. (TRUE or (FALSE).

**Explain** in one sentence, why or why not:




This is a BST, and it has N nodes where the height is N.

c) Consider the `ArrayStack` for Homework 1. On `push`, when the array is full, you make a new array that is twice the size of the old array to make space for new elements and copy over the old elements into your new array.

1) What is the worst-case operation cost for `push`?

O(N)

2) In one sentence, explain the case where this occurs:

Worst case you have to copy over N elements to a new array to make space for the new elements.

3) What is the amortized runtime for `push`?

O(1)

4) In one sentence, explain why amortized analysis gives the runtime you put for part (3):

The number of cheap pushes (N) you get at O(1) runtime per cheap operation is proportional to how expensive your single worst case operation is, so it averages out.

**7) Design Decisions (8 points)**
For each task, choose the most efficient structure in terms of time complexity. You may choose from the following:

- **unsorted array**
- **binary search tree**
- **AVL tree**
- **min heap with Floyd's method**
- **stack implemented with a linked list**
- **queue implemented with a circular array**
- **hash table using separate chaining**

You do NOT need to explain your decision.

a) You are managing a bus system. You need to keep track of all the drivers and which bus they are driving. You will frequently be modifying whether a driver is currently driving or not and determining which bus they are driving if they are currently driving.

Hash table with separate chaining used as a hash map
need: fast find and insert
don't need: sorting
→ O(1) find & insert.

b) You are managing a browser history of pages a user has visited. You will keep track of the previous 100 pages and want to quickly be able to insert new pages as the user visits them and go back to previous pages the user just left.

stack implemented with a linked list.
need: fast get the last item
don't need: access to $k^{th}$ page visited
→ LIFO
O(1) pop
O(1) push

c) Keeping track of the next customer at a sandwich shop in order of arrival.

Queue implemented with a circular array.
need: fast access to front of line
don't need: access to $k^{th}$ customer
→ FIFO
O(1) add
O(1) remove

d) You have a variety of files. Each file is for one building and the file represents other buildings to visit. Each line of the file has a real number representing the distance to another building and the data in the file is not sorted. Your goal is to read in a provided file and find the distances to the 5 closest buildings.

min heap with Floyd's method.
need: fast build, fast access to top k elements where k is 5
don't need: to keep the structure for long, $k^{th}$ access directly (like a sorted array).
→ O(N) build Heap
O(5 log N) access for 5 closest elements

**8) Abstraction (6 points)**
Suppose you had the following classes:

```
// Stores a flexible list of Person objects
public class PersonList {
   private Person[] data;
   public int size;
   public PersonList() {
      this.data = new Person[10];
      this.size = 0;
   }

   public Person[] getList() {
      Person[] clientData = new Person[this.size];
      for (int i = 0; i < this.size; i++) {
         clientData[i] = this.data[i];
      }
      return clientData;
   }

   // other methods
}
```

#1 public → private

#2 this.data[i]; → new Person(this.data[i].getName());

```
// Represents the data for a Person
public class Person {
   private String name;

   public Person(String name) {
      setName(name);
   }

   public String getName() {
      return this.name;
   }

   public void setName(String name) {
      if (name == null) {
         throw new IllegalArgumentException();
      }
      this.name = name;
   }
}
```

for #2, Also acceptable:
- adding final
- deleting setName() X

**Abstraction Continued**
These two classes make up a flexible list of Person objects. As a client, you want to create Person objects, add/remove Person objects to/from the list, and get the full list of Persons.

There are at least two places where poor design decisions in these classes allow for an adversarial client to break the abstraction and cause incorrect behavior of `PersonList`.

a) Identify two places that an adversarial client might be able to cause incorrect behavior from the `PersonList`. **You should circle the two places in the code above where there are issues.**

b) For each of the cases you identified, write adversarial lines of pseudocode, as the client, to cause the `PersonList` to have incorrect behavior. You do not have to explain how your client code exploits the design flaws.

1)
```
PersonList pl = new PersonList();
pl.size = 100000; // haha!
```

2)
```
PersonList pl = new PersonList();
pl.add(new Person("Pascale"));
pl.add(new Person("Matthew"));
Person[] list = pl.getList();
list[1].setName("haha!");
```

c) Fix both of the design flaws, as the implementer, so that an adversarial client can't cause incorrect behavior of the `PersonList`. You can modify either or both of the classes. **Modify the code by crossing out code you don't want, and write your code next to where you'd like to insert it.**

**You're done! Yay!**