**Name:** Solutions

**Student #:** ____________

# CSE373 Summer 2016 Midterm
# July 22, 2016

Rules:

- The exam is closed-book, closed-note, closed-calculator, closed-electronic
- You have 60 minutes to complete the exam. **Please stop promptly at 11:50**
- If you write any answers 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 rather and not on style, unless otherwise noted.
- Unless otherwise stated, all **logs are base 2.**

| **Problem** | **Description** | **Earned** | **Max** |
|---|---|---|---|
| 1 | Asymptotic Analysis | | 7 |
| 2 | Runtime Analysis | | 12 |
| 3 | AVL Trees | | 9 |
| 4 | Miscellaneous Questions | | 12 |
| 5 | Heaps | | 15 |
| 6 | Design Decisions | | 15 |
| 7 | Binary Search Trees | | 10 |
| 8 | Extra Credit | | 1 |
| Total | Total Points | | 80 |

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down 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 projector
- Take a deep breath, relax!

**1) Asymptotic Analysis (7 points)**
For each function $f(n)$ below, give an asymptotic upper bound using "Big-Oh" notation.
**You should give the tightest and simplest bound possible.**

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

a) $f(n) = 12n^4$ $O(n^4)$

b) $f(n) = 17\log(n) + 32n^2 + 2$ $O(n^2)$

c) $f(n) = \log(\log(n))$ $O(\log(\log(n)))$

d) $f(n) = 0.0002n + n\log(n) + 0.5n^{(1/2)}$ $O(n\log(n))$

e) $f(n) = 13log^2(n) + \log(n^{17}) + \log(\log(n))$ $O(\log^2(n))$

f) $f(n) = n(3n-2)2n$ $O(n^3)$

g) $f(n) = \log(n^2) + \log(2n)$ $O(\log n)$

## 2) Runtime Analysis (12 points)

Describe the worst case running time of the following code in "Big-Oh" notation in terms of the variable *n*. You should give the tightest bound possible.

Runtime:

a)
```
void bar(int n) {
        if (n <= 0) {
                print "done";
        else {
                print n;
                bar(n - 3);
        }
}
```

$\boxed{O(n)}$

b)
```
int taz(int n) {
        int z = 0;
        for (int x = 0; x < n; x++) {
                for (int y = 0; y < n; y++) {
                        z = z + 2;
                }
        }
        for (int w = n; w >= 0; w--) {
                print "okay";
        }
        return 0;
}
```

$\boxed{O(n^2)}$

c)
```
void baz(int n, int sum) {
        int i = n;
        while (i > 1) {
                for (int j = 0; j < n * n * n; j++) {
                        sum++;
                }
                i = i/2;
        }
}
```

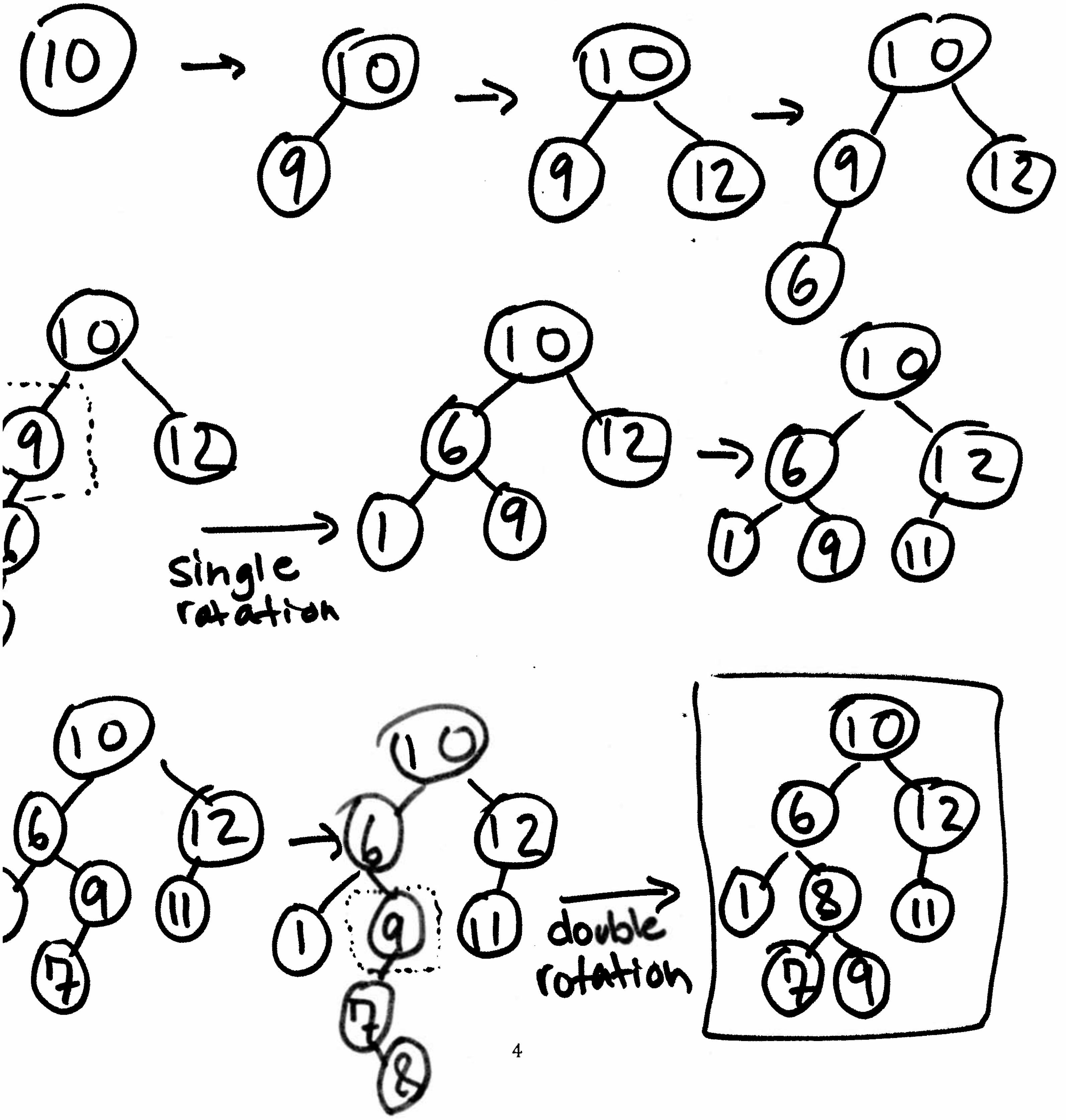
$\boxed{O(n^3 \log(n))}$

d)
```
int foo(int n) {
        if (n < 5) {
                return n - 10;
        else {
                int x = 0;
                while (x < n) {
                        print "foo";
                        x = x + 1;
                }

                return foo(n - 2);
        }
}
```

$\boxed{O(n^2)}$

$T(n) = n + T(n-2)$

**3) AVL Trees (9 points)**
Draw the AVL Tree that results from inserting the **keys 10, 9, 12, 6, 1, 11, 7, 8** in that order into an **initially empty AVL Tree**. You are only required to show the final tree, although if you draw the intermediate trees it may help us award partial credit.
**Please circle your final answer for any credit.**



**4) Miscellaneous Questions (12 points)**

**True/False: Circle one**

a) For the union-find implementation discussed in class: using path compression and union by size, *find*(x) runs in worst case O(1) time and amortized O(1) time. **(TRUE or [✓ FALSE])**

b) Performing *union*(x, y), assuming x and y are representative members of their respective disjoint sets, can be implemented to run in worst case O(1) time. **([✓ TRUE] or FALSE)**

c) Every AVL Tree is a complete binary tree **(TRUE or [✓ FALSE])**

d) The ArrayStack implementation discussed in class has an amortized O(1) *push*(x). **([✓ TRUE] or FALSE)**

e) Every binary heap has its height bounded by log(n), where n is the number of nodes **([✓ TRUE] or FALSE)**

f) Assume a *find*(x) operation in union-find does path compression. The operation that does the compression gets asymptotically slower. **(TRUE or [✓ FALSE])**

worded poorly, everyone received credit.

**Short Answer, be concise:**

g) **(2 points)** What is the balance condition for AVLTrees?

The height of a node's left and right subtrees differs by at most 1.

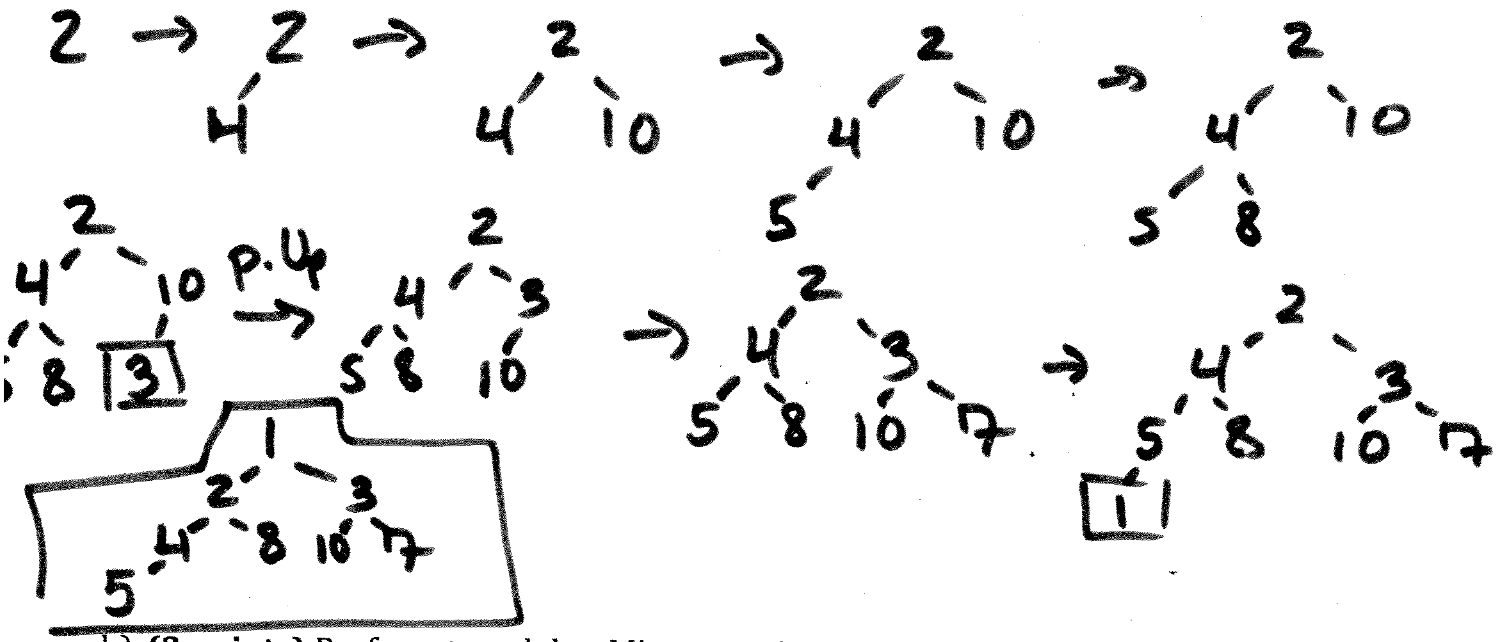
h) **(4 points)** List **two** collision resolution schemes for hash tables. For each scheme, list one disadvantage. Be specific. Note: disadvantage needed to be unique to scheme.

Separate chaining: uses much more space than other schemes.

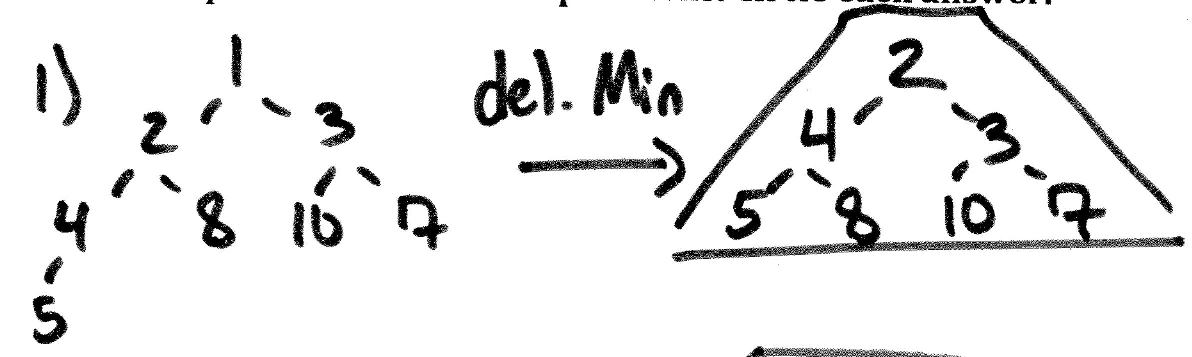
Linear Probing: suffers from primary clustering

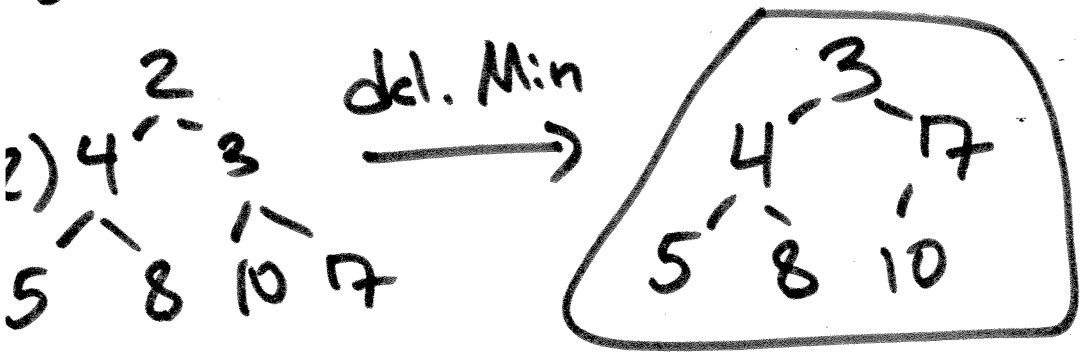
**5) Heaps (15 points)**
For this problem, you are only required to show the final heaps, although if you draw the intermediate trees it may help us award partial credit.
**Please circle your final answer for any credit.**

a) **(4 points)** Draw the resulting binary tree representation from inserting the following **keys 2, 4, 10, 5, 8, 3, 7, 1** in that order into an <u>initially empty binary heap</u>.

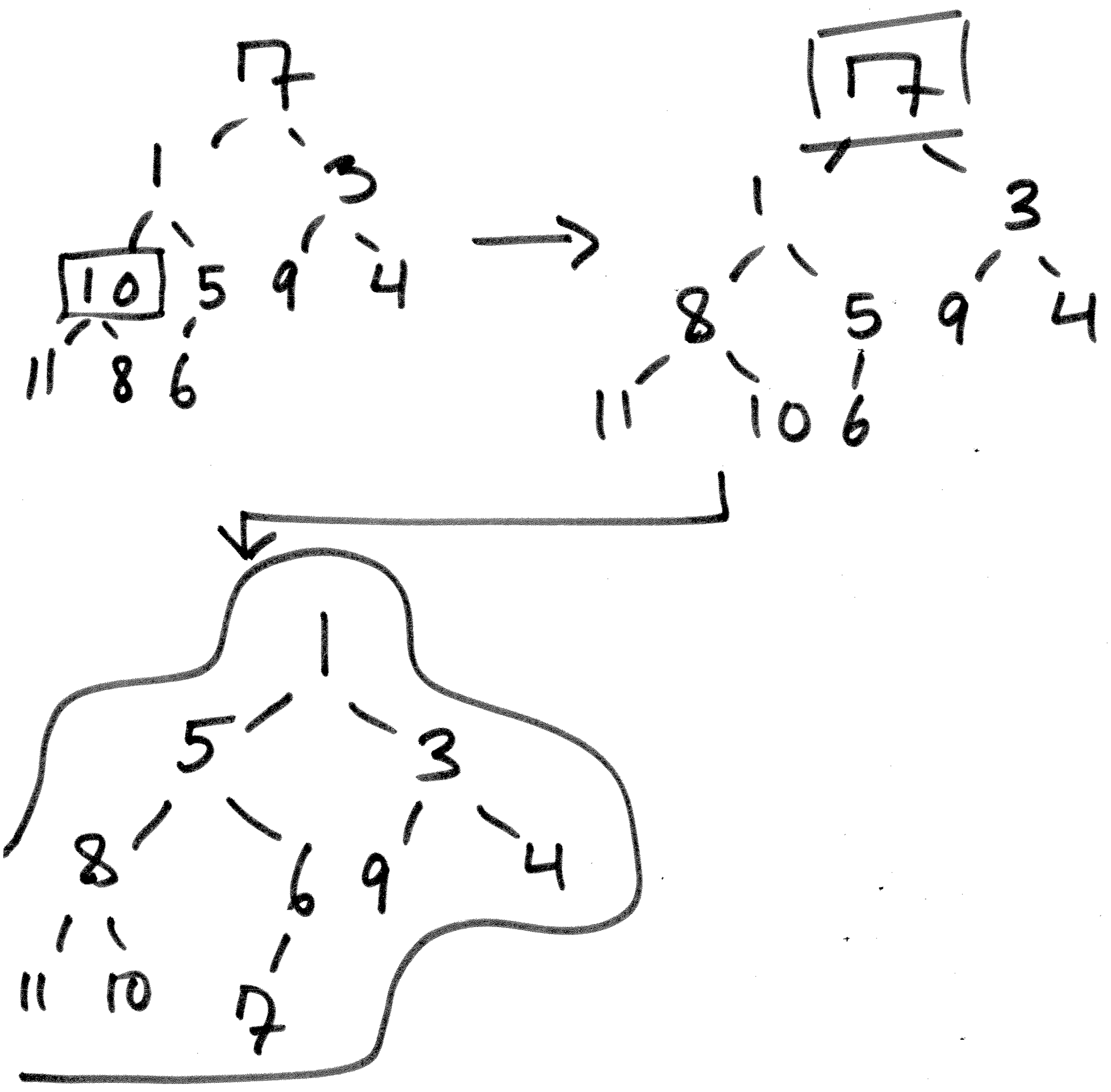


b) **(2 points)** Perform two deleteMin operations on your heap from part **a)**. **Show the heap after each of these operations. Circle each answer.**





c) **(4 points)** Now create a heap from **keys [7, 1, 3, 10, 5, 9, 4, 11, 8, 6]** using **Floyd's buildHeap** algorithm



Here is pseudocode for a client implementation of buildHeap, using the **insert method described in class.**

```
// Builds a heap from an array of n elements
public buildHeap(array) {

   MinHeap h = new MinHeap
   for each element in array:
       h.insert(element)
}
```

d) **(2 points)** Give a tight bound for the worst case runtime of this implementation in terms of *n*. Will this produce a valid heap?

n inserts, log(n) runtime each.

O(n log(n)).

Yes, it will produce a valid heap

e) **(3 points)** Is this implementation as good as Floyd's buildHeap algorithm? Explain. What is the runtime of Floyd's buildheap algorithm?

No. Floyd's runtime is O(n) whereas the above runtime is O(n log n)

Note: Needed some sort of comparison to another structure for (b) and (c)

**6) Design Decisions (15 points)**

For each design decision, choose the most efficient structure in terms of time complexity. You may chose from the following:

**Array, sorted array, sorted linked list, binary search tree, AVL Tree, min heap, up tree, queue implemented with an array, stack implemented with a linked list.**

Explain your decision. If you use a sorted structure, explain what it is sorted by.

a) You are in charge of managing an airport terminal's gates, with ID's 1 through 50. Important operations include: marking a gate as in use or unused.

Sorted array 1 to 1 correspondence of ID → index. gives us O(1) lookup. Better than all other options.

b) Your task is to store a directory of employees who work at a supermarket. Important operations include the ability to add an employee to the directory, to determine whether someone works at the store (based on name), and the ability to print all of the employees in sorted order. You may assume that no two employees have the same name.

AVLTree. Allows for fast sorted insert O(log n), as well as fast lookup O(log n), and print sorted O(n). A sorted array or LL would have O(n) insert.

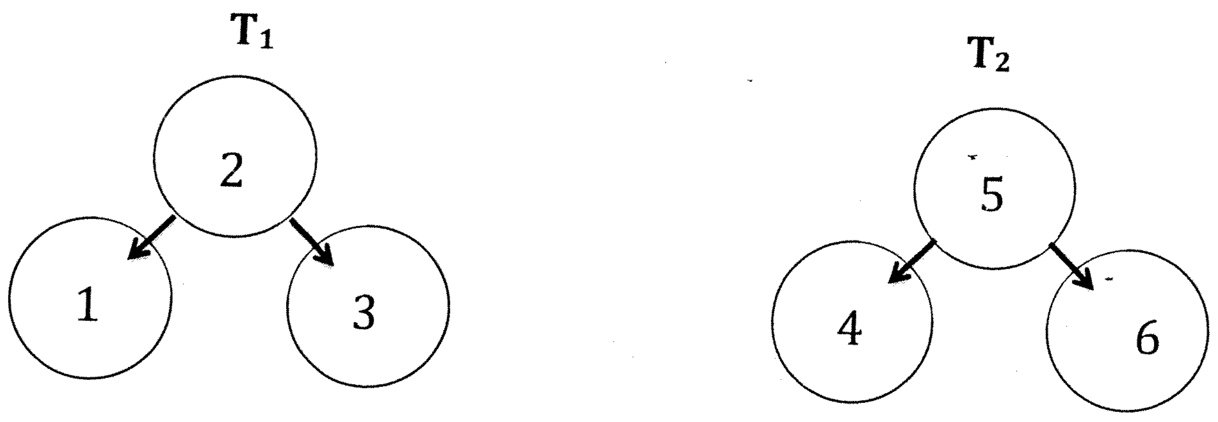
c) You want to keep track of the 100 most expensive houses in a neighborhood. The only operations that must run fast are adding a new price, and the retrieval of the *n*'th highest price. For instance, if the following prices are stored: 20, 30, 50, 10, 90, 100 and I ask for the 2nd highest price, 90 would be returned.

Sorted array: index directly to nth largest. Insert is still O(n), but others constant time

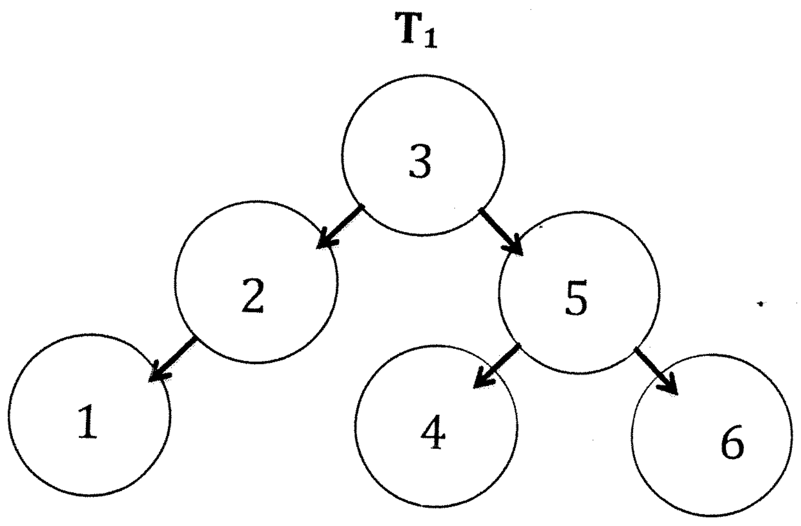
AVL tree would give O(log n) insert, but [illegible] long runtime to find the nth largest, since there is no way to directly find nth largest in a tree. So Sorted array is Better.

**7. Binary Search Tree: Efficient Merge (10 points)**
Given two Binary Search Trees: $T_1$ and $T_2$, where every element in $T_1$ is less than every element in $T_2$ (i.e. $T_1 < T_2$). Write code to merge the two Binary Search Trees. The resulting tree should also be a Binary Search Tree.

For example: Given the following BSTs $T_1$ and $T_2$:



The following would be a valid representation of $T_1$ after calling $T_1.\texttt{merge}(T_2)$:



**Note**: For $T_1.\texttt{merge}(T_2)$, $T_2$ is merged into $T_1$. We do not care what happens to $T_2$ after the merge.

**Your method should be as efficient as possible.**

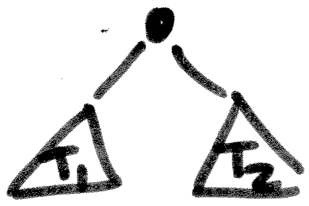
**Remember**: Objects of the same class can access each other's fields.

**Provide your implementation on the next page, some code has already been provided for you.**

```
public class BSTNode {
     public int data;
     public BSTNode left;
     public BSTNode right;

     // Creates a new BSTNode with the given data, and left
     // and right subtrees.
     public BSTNode(int data, BSTNode left, BSTNode right) {
          this.data = data;
          this.left = left;
          this.right = right;
     }
}

public class BST {
     public BSTNode root; // the root of the tree

     // Removes the given node from the tree and returns it.
     // After removal, the tree is still a BST.
     public BSTNode delete(BSTNode nodeToDelete) {
          // Already implemented - you may use this method
     }

     // you may assume T2 != null
     // you may assume T1.root != null
     // you may assume T2.root != null
     // Remember: Objects of the same class can access each
     // other's fields.
     public void merge(BST T2) {
          // YOUR CODE HERE
```

One solution:

```
BSTNode temp = root;
while (temp != null)                //find max(T1)
   temp = temp.right;
BSTNode newRoot = delete(temp);
newRoot.left = root;
newRoot.right = T2.root;
root = newRoot;
```



**8. Bonus (1pt):** Tell me a joke!

Why did the programmer quit his job?

He didn't get arrays!