Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 120 minutes to earn a maximum of 120 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed one double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write "Continued on S1" (or S2, S3) and continue your solution on the referenced scratch page at the end of the exam.
- Do not spend time and paper rederiving facts that we have presented in lecture or recitation. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
1: Information	2	2
2: Frequentest	2	8
3: Haphazard Heaps	2	10
4: Transforming Trees	2	10
5: Sorting Sock	4	20
6: Triple Sum	1	15
7: Where am i?	1	15
8: Methane Menace	1	20
9: Vapor Invite	1	20
Total		120

Name: Thomas Ren

MIT Kerberos Username: thomasren681@gmail.com (Google email)

Problem 1. [2 points] **Information** (2 parts)

(a) [1 point] Write your name and email address on the cover page.

✓

(b) [1 point] Write your name at the top of each page.

✓

Note: I answer this quiz using orange color and grade it with red color.

Problem 2. [8 points] **Frequentest** (2 parts)

The following two Python functions correctly solve the problem: given an array `X` of n positive integers, where the maximum integer in `X` is k, return the integer that appears the most times in `X`. Assume: a Python `list` is implemented using a dynamic array; a Python `dict` is implemented using a hash table which randomly chooses hash functions from a universal hash family; and `max(X)` returns the maximum integer in array `X` in worst-case $O(|X|)$ time. For each function, state its **worst-case** and **expected** running times **in terms of n and k**.

(a) [4 points]

```
def frequentest_a(X):
    k = max(X)
    H = {}
    for x in X:
        H[x] = 0
    best = X[0]
    for x in X:
        H[x] += 1
        if H[x] > H[best]:
            best = x
    return best
```

O(|X|) initialize X

O(|X|). O(1) O(n).

i. Worst-case:

O(n²)

Worst-case: takes O(n)-time to get access to H[x].

ii. Expected:

O(n).

Expected.: takes O(1)-time to get access to H[x].

(b) [4 points]

```
def frequentest_b(X):
    k = max(X)
    A = []
    for i in range(k + 1):
        A.append(0)
    best = X[0]
    for x in X:
        A[x] += 1
        if A[x] > A[best]:
            best = x
    return best
```

O(n). O(k). O(n). O(1).

i. Worst-case:

O(n+k).

A is a DAA (Direct Access Array), so can be reached any given key in worst-case O(1)-time. But takes O(k)-time to build.

ii. Expected:

O(n+k)

Doesn't contain hash tables, so no expectation here.

Problem 3. [10 points] **Haphazard Heap** (3 parts)

Array `[A,B,C,D,E,F,G,H,I,J]` represents a **binary min-heap** containing 10 items, where the key of each item is a **distinct** integer. State which item(s) in the array could have the key with:

(a) the smallest integer

A.

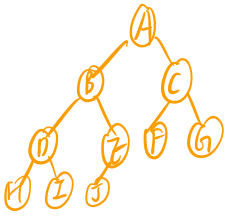
(b) the third smallest integer

~~B or C.~~

B/D/E or C/F/G.

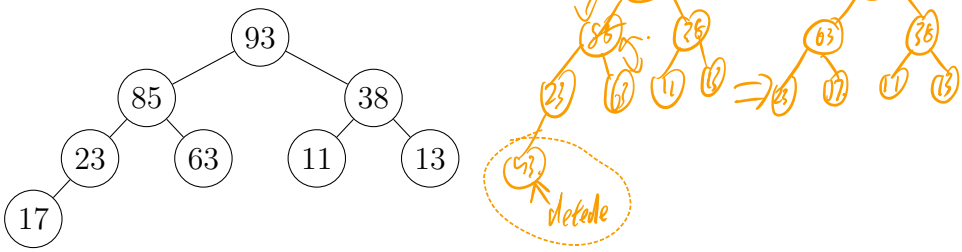
(c) the largest integer

F/G/H/I/J.



Problem 4. [10 points] **Transforming Trees** (2 parts)

The tree below contains 8 items, where each stored item is an integer which is its own key.



(a) [6 points] Suppose the tree drawn above is the implicit tree of a binary max-heap `H`. State the array representation of `H`, **first before** and **then after** performing the operation `H.delete_max()`.

Before: H = [93, 85, 38, 23, 63, 11, 13, 17].

H.delete_max(): swap (93) with last item, i.e. (17)
then heapify_down to maintain max-heap property.

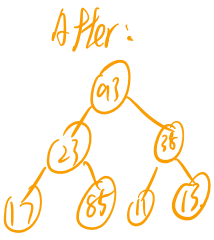
After: H = [85, 63, 38, 23, 17, 11, 13].

(b) [4 points] Suppose instead that the original tree drawn above is a Sequence AVL Tree `S` (note Sequence data structures are zero-indexed). The items in **the leaves** of `S` in traversal order are $(17, 63, 11, 13)$. Perform operation `S.delete_at(3)` on `S` including any rotations, and then **list the items** stored in **the leaves** of `S` in traversal order, after the operation has completed. (You do not need to draw the tree.)

Before: The traversal order of S is: [17, 23, 85, 63, 93, 11, 38, 13].

S.delete_at(3): deletes (63), since it's a leaf, we delete that directly
the right_rotate((85)) to maintain AVL property.

After:



After: The traversal order of S is: [17, 23, 85, 93, 11, 38, 13] (stable)
– – – – – leaves is: [17, 85, 11, 13].

Problem 5. [20 points] **Sorting Sock** (4 parts)

At Wog Hearts School of Wizcraft and Witcherdry, n incoming students are sorted into four houses by an ancient magical artifact called the Sorting Sock. The Sorting Sock first sorts the n students by each of the four houses' attributes and then uses the results to make its determinations. For each of the following parts, state and justify what type of sort would be most efficient. (By "efficient", we mean that faster correct algorithms will receive more points than slower ones.)

(a) [5 points] For House Puffle Huff, students must be sorted by ***friend number***, i.e., how many of the other $n-1$ incoming students they are friends with, which can be determined in $O(1)$ time.

friend $\leq n-1$.

Since for every student, his/her friend number is at most $n-1$, so counting sort or radix sort keyed on friend number can be used and finished in $O(n)$-time.

Counting Sort: Running time:
takes $O(k)$-time, where k is the maximum number of keys, here is upperbounded by $n-1$.
Thus taking $O(n)$-time.

Correctness:
Since friend number is a non-negative integer ranging from 0 to $n-1$, and different students may have the same friend number, the condition is satisfied to use either algorithm.

Stability:
Both algorithms are stable.

(b) [5 points] For House Craven Law, students must be sorted by the weight of their books. Book weights cannot be measured precisely, but the Sorting Sock has a ***scale*** that can determine in $O(1)$ time whether one set of books has total weight greater than, less than, or equal to another set of books.

The subject tells that we can only differentiate one book weight with another by comparing those two, so a comparison model, which takes at least $O(n \log n)$-time, is destinied.
Algorithm like Merge Sort hit this lower bound.

Correctness: By directly comparing two students' book weights and merge together recursively, we get a sorted array.
Running time: by using the scale to compare book weights in $O(1)$-time, the algorithm runs in $O(n \log n)$-time, which reaches the optimal according to the property of comparison model.
In-place: the MergeSort isn't in-place.

(c) [5 points] For House Driven Gore, students must be sorted by ***bravery***, which can't be directly measured or quantified, but for any set of students, the Sorting Sock can determine the bravest among them in $O(1)$ time, e.g., by presenting the students with a scary situation.

This sorting strategy is similar to priority queue sort without the cost of maintaining a max-heap data structure.
We can achieve a sorted array by repeated call the bravest, finished in O(n)-time.

(d) [5 points] For House Leather Skin, students must be sorted by their ***magical lineage***: how many of a student's ancestors within the previous $3\lceil \log n \rceil + 4$ generations were magical. Recall that humans, magical or not, always have two parents in the previous generation, unlike binary tree nodes which have at most one. Assume the Sorting Sock can compute the magical lineage of a student in $O(1)$ time.

The magical lineage is a non-negative integer number, so we can use linear sorting to achieve O(n)-time.
The key part is to decide the upper bound of this magical lineage.
Since every human has parents, so the ancestors of one student is $2^{3\lceil \lg n \rceil + 4} = O(n^3)$, which is polynomially bounded.
Thus using Radix Sort can still reach linear-time sorting.

Problem 6. [15 points] **Triple Sum**

Given three arrays A, B, C, each containing n integers, give an $O(n^2)$-time algorithm to find whether some $a \in A$, some $b \in B$, and some $c \in C$ have zero sum, i.e., $a + b + c = 0$. State whether your running time is worst-case, expected, and/or amortized.

The brute-force strategy takes $O(n^3)$-time to traverse all A, B, C's items, but we're only allowed for $O(n^2)$-time. $O(n^2)$-time allows us traversing two of the sets and build a hash table of all sums $a+b$, $a \in A$, $b \in B$. Then for every $c \in C$, check whether $-c$ is in the hash table or not.

The pseudocode is as follows:

```
M = { }
for a in A:                O(n^2)
  for b in B:
    if M[a+b] is None:
      M[a+b] = 1
    else:
      M[a+b] += 1.
for c in C:                O(n)
  if -c in M:
    return True.
return False
```

Thereby, our algorithm takes $O(n^2)$ expected time.

Problem 7. [15 points] **Where Am *i*?**

Given a Sequence AVL Tree T containing n nodes, and a pointer to a node v from T , describe an $O(\log n)$-time algorithm to return the (zero-indexed) index i of node v in the traversal order of T. (Recall that every node `u` in a Sequence AVL Tree T stores an item `u.item`, parent `u.parent`, left child `u.left`, right child `u.right`, subtree height `u.height`, and subtree size `u.size`.)

The root's index can be easily known by calling the root.left.size, then its children's index can be easily computed by calling its subtree's size and computed relative to the root's index.

Thus, wherever node v is, we can recursively compute the index in the path from node v to the root.

1. Traverse and record the node from node v to the root in O(logn)-time by repeatedly calling u.parent. and recording whether u is in its parent's left subtree or right subtree.

2. Recursively compute the nodes' recorded indices. to node v in O(logn)-time, each operation in O(1)-time.

Correctness: Prove by induction

Base Case: the root's index = root.left.size.

Induction: the node y is node x's parent, wlog, assume x is in y's right subtree.

we already know y's index.

x's index = y + x.left.size + 1.

Thus getting x's index.

Q.E.D.

Problem 8. [20 points] **Methane Menace**

FearBird is a supervillian who has been making small holes in the methane gas pipe network of **mahtoG City**. The network consists of n pipes, each labeled with a distinct positive integer. A hole i is designated by a pair of positive integers (p_i, d_i), where p_i denotes the label of the pipe containing the hole, and d_i is a positive integer representing the ***distance*** of the hole from the ***front*** of pipe p_i. Assume any two holes in the same pipe p_i will be at different distances from the front of p_i. When a new hole (p_i, d_i) is spotted, the city receives a ***report*** of the hole to keep track of. The city will periodically patch holes using the following priority scheme:

- if each pipe contains at most one hole, patch any hole (if one exists);
- otherwise, among pairs of holes (p_i, d_i) and (p_j, d_j) appearing on the **same pipe**, i.e., $p_i = p_j$, identify any pair with smallest distance $|d_i - d_j|$ between them, and patch one of them.

Describe a database supporting the following operations, where k is the number of recorded but unpatched holes in the network at the time of the operation. State whether your running times are worst-case, expected, and/or amortized.

`initialize(`H`)`	Initialize the database with n holes $H = \{(p_0, d_0), \ldots, (p_{n-1}, d_{n-1})\}$, with one hole on each pipe, in $O(n)$ time
`report(`p_i`,` d_i`)`	Record existence of a hole in pipe p_i at distance d_i in $O(\log k)$ time
`patch()`	Patch any hole that follows the priority scheme above in $O(\log k)$ time

Note: I finish the database except for operation initialize(H) that takes O(n log n) expected time.

Database:

1. A Hash table H mapping the set $P=\{p_0, p_1, \cdots, p_{n-1}\}$ to pointers pointing to a bunch of Set_AVL Trees.
2. For each p_i, construct a Set_AVL Tree T_i keyed on distances d_j.
3. Augment each Set AVL Tree with the priority scheme, i.e.:

```
if T_i.size == 1:
    return float('inf')
else:
    augment the smallest |d_i - d_j|
    augmentation can be returned in O(1)-time and maintained in O(log k)-time, k is the size of the AVL Tree T_i.
```

4. A min-heap P, each node contains (p_i, δ_i), where δ_i is the smallest distance in T_i. According to our AVL Tree augmentation, δ_i is a positive number. Min-heap property is maintained: that $u.\delta \geq v.\delta$ if u is v's ancestor.

Then I will demonstrate how to use our data structures to implement given operations while analyzing their running time.

Implementation:

1. initialize(H): First, build a Hash table H to map p_i to T_i. Build this takes $O(n)$-time so that we get access to any AVL Tree T_i in expected $O(1)$-time.

Continued on S1.

Problem 9. [20 points] **Vapor Invite**

Vapor is an online gaming platform with n users. Each user has a unique positive integer ***ID*** d_i and an updatable ***status***, which can be either active or inactive. Every day, Vapor will post online an ***active range***: a pair of positive integers (a, b) with the property that **every user** having an ID d_i contained in the range (i.e., with $a \leq d_i \leq b$) **must be active**. Vapor wants to post an active range containing as many active users as possible, and invite them to play in a special tournament. Describe a database supporting the following **worst-case** operations:

`build(`D`)`	Initialize the database with user IDs $D = \{d_0, \ldots, d_{n-1}\}$, setting all user statuses initially to active, in $O(n \log n)$ time
`toggle_status(`d_i`)`	Toggle the status of the user with ID d_i, e.g., from active to inactive or vice versa, in $O(\log n)$ time
`big_active_range()`	Return an active range (a, b) containing the largest number of active users possible in $O(1)$ time

We can build a database as a Set AVL Tree designated by T with augmentation to compute the max (a,b) pair in O(1)-time.

Augmentation: for each node v, suppose its subtree's keys ranging from d_x to d_y

1. v.status : True if it's active, False otherwise.
2. v.left_prefix: an active range starting from d_x, if d_x is active. Otherwise set to (d_x-1, d_x-1).
3. v.right_prefix: an active range terminating at d_y, if d_y is active. Otherwise set to (d_y+1, d_y+1)
4. v.max_prefix: the max active range in v's subtree.

Then we will show how to maintain such augmentation.

1. v.max_prefix: for a node v and its left and right subtree

   v.max_prefix = max (left, middle, right).

   where left, right and middle are computed as follows.

   left = v.left.max_prefix.

   middle = v.left.right_prefix ∪ v ∪ v.right.left_prefix if v.status else 0 (zero).

   right = v.right.max_prefix.

2. v.left_prefix: if v.left.left_prefix = v.left.right_prefix and v.status:

   v.left_prefix = v.left.left_prefix ∪ v ∪ v.right.left_prefix.

   else:

   v.left_prefix = v.left.left_prefix.

3. v.right_prefix: if v.right.right_prefix = v.right.left_prefix and v.status:

   v.right_prefix = v.left.right_prefix ∪ v ∪ v.right.

   else:

   v.right_prefix = v.right.right_prefix.

Continued on S2.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S1" on the problem statement's page.

Continued from Problem 8.

we then insert each tuple. from holes H to our self-BVL tree, each insertion takes $O(\log s)$-time, where s is the tree's size, i.e. $T_i.size$, and $O(1)$-time to find the tree T_i.
Thereby, initializing the whole set of n items takes $O_e(n \log n)$-time.

2. report (p_i, d_i): Find T_i according to pipe i using hash table H in $O_e(1)$-time.
Insert d_i to T_i takes $O(\log s)$-time, where s is the tree's size and is upper-bounded by k, i.e. $s \leq O(k)$. Thus taking $O(\log k)$ time.
Besides, maintaining our augmentation and the min-heap P also takes $O(\log k)$-time.
Therefore the whole procedure takes $O_e(\log k)$-time.

3. patch(): Pop the root of the min-heap in $O(\log k)$-time.
Find the tree T_i using hash table. H in $O(1)$-time.
Delete the node, say, d_i while maintaining BVL property and augmentation in $O(\log k)$-time.
Insert the new smallest distance from T_i into min-heap P in $O(\log k)$-time.

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S2" on the problem statement's page.

Continued from Problem 9.

Since we finished describing our data structure and how to maintain such augmented data structure, we will. then show more how to use our data structure to implement the operations.

1. Build(D): Set AVL Tree T keyed on the integer ID.d_i by repeatedly inserting a new ID d_i while maintaining the tree balanced.
   According to the AVL Tree property, inserting a node takes worst-case $O(\log n)$-time, so, for the whole procedure, build(D) takes $O(n \log n)$-time.

2. toggle_status(d_i): we have two tasks to do here: ① binary search the status with ID d_i ② change the status while maintaining our augmentation.
   ①: by comparing the integer IDs, we can achieve the node with key d_i in $O(\log n)$-time, since AVL Tree is balanced, $O(h) = O(\log n)$.
   ②: After changed d_i's status, we need to recompute our augmentation for all d_i's ancestors including d_i. There are at most $O(h)$ of them into tal, thus taking $O(\log n)$-time.

3. big-active-range(l): return the root's max_prefix in $O(1)$-time, since our augmentation is designed for this operation.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S3" on the problem statement's page.

6.006 Introduction to Algorithms

Spring 2020