Name:_____ Tufts ID: _____

COMP160: Algorithms, Spring 2020, **Exam 2**

Instructions

- You have 3 hours to answer the 4 questions in the exam.
- The exam is designed to take only 75 minutes so that you will have plenty of time for uploading at the end, plus extra time to work on the problems.
- Please read **each** question carefully, specify any assumptions that you make, and **briefly justify** all your answers unless otherwise indicated.
- Please use a different page to answer each question. Your answers may be handwritten or typewritten or any combination as long as they are easily readable.
- When you are done, upload your answers to Gradescope **before** the end of the 3 hour time period. Late submissions will be penalized 1 pt per minute late.
- This exam is open-book. You may consult any books, notes, webpages, online videos, etc that do not involve interacting with a live person. You may **not** consult a live person in any way.
- If you need any clarifications during the exam make a **private** question on Piazza
- WARNING: This is an exam from previous semesters given as practice. Note that topics were slightly different and learning expectations may have changed. Also, solutions were hastily written just as notes for TAs. Use them at your own discretion

Please write the following affirmation and sign with your name. This will be uploaded as problem 5 on Gradescope and is necessary in order for you to receive credit on the exam:

"I affirm that I have not consulted with any other person (except the instructors) in the course of answering this exam. I understand that doing so may result in a grade of zero on the exam. Signed, "

1. (25 points) We want to have a hash table to store grades. The key of the table will be the student's id, and we will store their grade. All of the questions below assume we have n elements in a table with m buckets and that collisions are handled with chaining:

   (a) (3 points) Look through your notes. What is the expected runtime of a successful search in a hash table? What about an unsuccessful search? Just list runtimes as a function of n and m (no justification necessary).

   Successful is $O(1+\frac{\alpha}{2}) = O(1+\frac{n}{2m})$. Unsuccessful is $O(1+\alpha) = O(1+\frac{n}{m})$

   (b) (5 points) Say we want expected runtime for an unsuccessful search to be $\Theta(\log n)$. Give a value of m that we could choose (as a function of n). What if we wanted expected runtime to be $\Theta(1)$? No justification is necessary.

   If we want $\Theta(\log n)$ we need $1+\frac{n}{m} = c\log n$. Solving for m we get $m = \frac{n}{(c\log n)-1}$. In order for expected runtime to be $\Theta(1)$ we need $\frac{n}{m}$ to be a constant d which means $m = \frac{n}{c}$.

   (c) (5 points) In the classic hash table with chaining, elements which collide (hash to the same bucket) are stored in a linked list. Suppose we still have m buckets, but store elements which hash to the same bucket in an AVL tree instead (each bucket will have its own AVL tree, indexed by tufts id). Draw an example with 5 buckets, 3 of which are empty, 1 only has a single item, and 1 has 7 items

   Simple drawing: note that the tree need not be perfectly balanced (just AVL)

   (d) (7 points) Give the pseudocode of the SEARCH(id) function for the modified hash table (where we use AVL trees instead of linked lists). This function should return the grade of the student whose Tufts id is `id` (or -1 if such a student does not exist). For simplicity, assume that the hash function of a student whose id is `id` can be are computed by function h(`id`).

   First we define a regular search of a BST (which is the same whether it's AVL or not).

```
SEARCHBST(k, x)        search for key k in tree rooted at x
  If x = NIL
    return -1
  elseif k = x
    return x.grade
  elseif k < x
    SEARCHBST(id, x.left)
  elseif k > x
    SEARCHBST(id, x.right)
```

   Then, given a hash function h which maps any id into some array A we define

```
SEARCH(id)
  SEARCHBST(id, A[h(id)])
```

   (e) (5 points) Since we modified the hash structure, the runtime of an unsuccessful search probably changed. What is the new expected runtime? Give your answer in terms of n

and m (for a general table, do not use the values of your answer to question (b)). Justify your answer with 1-2 sentences.

We still have an expected $\frac{n}{m}$ elements at any given hash table position. Since they are now arranged in an AVL tree, searching that tree will take $O(log(\frac{n}{m}))$. So combined with the $O(1)$ for computing the hash function, we now have $O(1+log(\frac{n}{m}))$ expected runtime.
Fun fact (not needed for full credit). In general, we pick m so that $\frac{n}{m} = \Theta(1)$. In this case, runtime is $O(1 + \log \Theta(1)) = O(1)$. In short, although conceptually an AVL tree should help, in practice it is the same as using a linked list.
Fun Fact 2 Even though we want expected, this manages to reduce worst case runtime to $\Theta(\log n)$

2. (20 points, 4 points each) Suppose that we insert **some** integers in the range 1 to 1000 stored in a binary search tree in an unknown order (the only thing we know is that no number is repeated). We then execute a search for number 267 that is in the tree. We are interested in the sequence of nodes that are traversed before reaching that number.

   For each of the sequences below, decide whether it **could** or **could not** be a proper sequence. Justify your answer with 1 sentence (a drawing/description of the tree in which it happens is good enough).

   Example (no need to answer this): 267

   Answer: Yes, this could happen if the root of the BST is number 267

   Example 2 (no need to answer this): $100, 345, 100, 267$

   Answer: No. This could not happen because once we see a node we do not revisit it.

   (a) $911, 369, 269, 268, 267$

   Yes. This could happen if the numbers are inserted in the given order.
   Graders note: for this question it is so hard to justify this so we will give full credit to anyone with the right "yes" answer, and use their justification for partial credit (if the answer happens to be incorrect).

   (b) $911, 169, 860, 238, 269, 267$

   Yes. Again, insert them in this order in BST. Mainly what we have is that all numbers larger than 267 are in a decreasing sequence, and smaller numbers are increasing.

   (c) $264, 265, 266, 267$

   Yes. This would happen if the numbers are inserted in exactly that order in the BST

   (d) $925, 806, 240, 808, 267$

   No, this is not possible: after 806 we go to the left, which means all other numbers we will check should be smaller. However, we afterwards visit node 808 which is larger.

   (e) Would any of your answers change if we are searching in an AVL tree instead? If so, which one(s)? Why?

   Question c becomes a NO. The reason is that the left child of 265 must be empty (because there are no numbers larger than its parent but smaller than itself). However, its right child has height at least 2, which means it is unbalanced and thus not an AVL tree

3. (25 points) You have a bunch of kids forming a line in front of a teacher before their respective parents pick them up. Each kid is represented by the tuple `(id, height, pos)` where `id` is its unique identifier (say, a name), `height` is their height and `pos` listing its location (an strictly positive integer representing the distance to the teacher, in meters).

   Design a data structure that can perform the following operations:

   - ADD(id, height, pos) Record that a new kid has been added to the line
   - PICKUP(id) Record that a kid has been picked up by its parents (thus, it should be removed from the data structure)
   - ISVISIBLE(id) returns true if the teacher can see the kid. Since all kids are on a line, a kid is visible if and only if there is no other kid that is taller and is closer to the teacher.

   Describe your data structure, invariants and an algorithm for performing each operation. Discuss correctness of the algorithm and justify runtime of each operation with 1-2 sentences (for full credit all operations must be executed in $O(\log n)$ worst case time).

The optimal runtime of $O(\log n)$ for each operation can be achieved using two connected data structures: a chaining hash table keyed by id with pointers (both ways) to an AVL tree keyed by pos and augmented with the tallest kid in subtree at each node.

First, initialize a hash table of size n. Initialize an empty AVL tree.

Add: Create a struct that holds the kid's id, height, and pos. Insert this struct into the hash table, keyed by id. Insert a copy of the struct into the AVL tree, keyed by pos. Updates to the tallest kid in subtree are made as the standard AVL insert algorithm follows a path down from the root. At every node on this path, we compare the height of the tallest kid in the subtree rooted at that node with the height of the kid to be inserted. If this new kid is taller, we update the node.

Runtime: Creating the struct takes constant time. Inserting into the hash table takes constant time due to the load factor of 1. Inserting into the AVL tree (height of $O(\log n)$)and making updates on the path down from the root takes $O(\log n)$.

Pickup: Search the hash table for the struct associated with the id. Delete the struct from the tree. The updates to the tallest kid in each subtree would have do be updated as well. The tallest kid in a subtree is the min of 3 values, i.e. this.tallest = max(this.height, this.left.tallest, this.right.tallest). Therefore, all updates can be made first locally, then propagating upwards. Also, delete the struct from the hash table.

Runtime: Searching into the hash table takes constant time due to the load factor of 1. Deleting from the AVL tree and updating up the path to the root takes $O(\log n)$.

isVisible: Search the hash table for the struct associated with the id. Then call the following algorithm with the node associated with that id in the AVL:

Algorithm 1 isVisibleHelper(targetNode)

```
if (targetNode.height < targetNode.left.tallest) then
  return false
for curNode on the path from targetNode back to root do
  if (coming from the left child) then
    continue
  if (coming from the right child) then
    if (targetNode.height < curNode.height) then
      return false
    if (targetNode.height < curNode.left.tallest) then
      return false
return true
```

The above algorithm is similar to using left subtrees to compute rank. Instead of adding size of left subtree and adding 1 whenever you turn left, we compare the target node's height with tallest of left subtree and with the current node's height. This algorithm is effectively determining whether or not the target node's height is greater than the heights of all the kids with positions closer to the teacher.

Runtime: Searching the hash table takes constant time. The above algorithm traverses the path from a node up to the root so it has a runtime of $O(\log n)$.

Details about rotations are not required.

4. (25 points)

ONGRATULATIONS! After a long fight you defeated the dragon from homework 7! Now it is time to escape from the collapsing dragon's lair while collecting as much treasure as you can. The dragon's lair has a rectangular grid-like structure: As soon as you enter a cell you automatically pick up any coins in that cell. You look at the scriptures for advice and you read:

When exiting the dragon's lair don't be greedy. Go to the escape by only walking down or to the right of your current position.

The scriptures magically contain an array A with the amount of gold in each position (that is $A[i, j]$ is equal to the amount of gold in position (i, j) of the dragon's lair). Also, you can assume that you start in position $(1, 1)$ and must reach the exit in position (n, m). Moving right means you increase the x coordinate and by moving down you increase the y coordinate. Your goal is to design an algorithm that returns the maximum number of coins that you can collect while preserving the above constraints. Answer the following questions:

(a) What technique will you use to solve the problem? Just say the name and explain why you think it is better than other possible options (2-3 sentences is enough)

DP is the right answer. Any justification should be given partial credit.

(b) Describe in detail your solution and justify correctness

Try to focus on the 5 part structure that we define
memoization: DP(i,j)=maximum profit I can make if I start at $(1,1)$ and walk to (i,j)
base cases: if $i = 1$ we have $DP(1,j) = \sum_{k=1}^{j} A[1,k]$. Similarly, if $j = 1$ we have $DP(i,1) = \sum_{k=1}^{i} A[k,1]$
recurrence for other cases we have $DP(i,j) = A[i,j] + \max\{DP(i-1,j), DP(i,j-1)\}$
justification similar to the example done in class. In order to enter position (i,j) you have two options: you either enter from above or from the left. You pick the best of the two paths and gain whatever is in cell (i,j)
runtime We have $\Theta(nm)$ entries in the table and each one needs constant amount of time/space $\Rightarrow \Theta(nm)$ runtime

(c) What is the runtime of your algorithm?

see above. Of course, ok if they answered somewhere else.

Note: as usual, points will be given for clear structure, efficiency, and arguing correctness.

5. (5 points) Write the following affirmation and sign with your name. This is necessary in order for you to receive credit on the exam:

   "I affirm that I have not consulted with any other person (except the instructors) in the course of answering this exam. I understand that doing so may result in a grade of zero on the exam. Signed, "