Quiz 1 Solutions

- Don't Panic.
- Write your name on every page.
- The quiz contains six problems. You have 100 minutes to earn 100 points.
- The quiz contains 18 pages, including this one and 3 pages of scratch paper.
- The quiz is closed book. You may bring one double-sided sheet of A4 paper to the quiz. (You may not bring any magnification equipment!) You may **not** use a calculator, your mobile phone, or any other electronic device.
- Write your solutions in the space provided. If you need more space, please use the scratch paper at the end of the quiz. Do not put part of the answer to one problem on a page for another problem.
- Read through the problems before starting. Do not spend too much time on any one problem.
- Show your work. Partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	Sorting Jumble	12	
2	Algorithm Analysis	20	
3	Buggy Code: Exploring Planet Nine	16	
4	Exploring Planet Nine, Part 2	18	
5	Mergesort with Linklist	18	
6	Zig-zag Sorting	16	
Total:		100	

Name: _____ Student id.: _____

Please circle your discussion group:

Herbert Mon. 12-2	Vu Long Tues. 2-4	Yuchuan Tues. 4-6	Jia Yee Wed. 10-12	Shi Yuan Wed. 2-4	Govind Thu. 10-12	Advay Thu. 12-2	Ray Thu. 4-6

Problem 1. Sorting Jumble. [12 points]

The first column in the table below contains an unsorted list of words. The last column contains a sorted list of words. Each intermediate column contains a partially sorted list.

Each intermediate column was constructed by beginning with the unsorted list at the left and running one of the sorting algorithms that we learned about in class, stopping at some point before it finishes. Each algorithm is executed exactly as described in the lecture notes. (One column has been sorted using a sorting algorithm that you have not seen in class.)

Identify, below, which column was (partially) sorted with which algorithm. *Hint: Do not just execute each sorting algorithm, step-by-step, until it matches one of the columns. Instead, think about the invariants that are true at every step of the sorting algorithm.*

Unsorted	**InsertionS**	**MergeS**	**BubbleS**	**QuickSort**	**HeapSort**	**SelectionS**	**Sorted**
Juliett	Alfa	Bravo	Bravo	Delta	Mike	Alfa	Alfa
Bravo	Bravo	Juliett	Alfa	Bravo	Lima	Bravo	Bravo
Kilo	Juliett	Kilo	Foxtrot	Echo	Kilo	Charlie	Charlie
Lima	Kilo	Lima	Charlie	Hotel	Juliett	Delta	Delta
Papa	Lima	Alfa	Juliett	Golf	Delta	Echo	Echo
Alfa	Papa	Charlie	Delta	Alfa	India	Juliett	Foxtrot
Foxtrot	Foxtrot	Foxtrot	Kilo	Foxtrot	Hotel	Foxtrot	Golf
Charlie	Charlie	Papa	India	Charlie	Charlie	Kilo	Hotel
Oscar	Oscar	Oscar	Golf	India	Foxtrot	Oscar	India
Delta	Delta	Delta	Hotel	Juliett	Echo	Lima	Juliett
November	November	November	Lima	November	Bravo	November	Kilo
India	India	India	Echo	Oscar	Alfa	India	Lima
Golf	Golf	Golf	Mike	Papa	Golf	Golf	Mike
Hotel	Hotel	Hotel	November	Lima	November	Hotel	November
Mike	Mike	Mike	Oscar	Mike	Oscar	Mike	Oscar
Echo	Echo	Echo	Papa	Kilo	Papa	Papa	Papa
Unsorted	**A**	**B**	**C**	**D**	**E**	**F**	**Sorted**

Problem 2. Algorithm Analysis [20 points]

For each of the following, choose the best (tightest) asymptotic function T from among the given options. Some of the following may appear more than once, and some may appear not at all. **Please write the letter in the blank space beside the question.**

A. $\Theta(1)$	B. $\Theta(\log n)$	C. $\Theta(n)$	D. $\Theta(n \log n)$
E. $\Theta(n^2)$	F. $\Theta(n^3)$	G. $\Theta(2^n)$	H. None of the above.

Problem 2.a.

$$T(n) = \left(\frac{n^2}{17}\right)\left(\frac{\sqrt{n}}{4}\right) + \frac{n^3}{n-7} + n^2 \log n$$

$T(n) =$ H: None of the above

Problem 2.b. The running time of the following code, as a function of n:

```
public static int loopy(int n){
    int j = 1;
    int n2 = n;
    for (int i = 0; i < n; i++) {
        n2 *= 5.0/7.0;
        for (int k = 0; k < n2; k++) {
            System.out.println("Hello.");
        }
    }
    return j;
}
```

$T(n) =$ **Sol: C** : $O(n)$

Problem 2.c. $T(n)$ is the running time of a divide-and-conquer algorithm that divides the input of size n into $n/10$ equal-sized parts and recurses on all of them. It uses $O(n)$ work in dividing/recombining all the parts (and there is no other cost, i.e., no other work done). The base case for the recursion is when the input is less than size 20, which costs $O(1)$.

$T(n) =$ **Sol:** C : $O(n)$

Problem 2.d. The running time of the following code, as a function of n:

```
public static int recursiveloopy(int n){
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            System.out.println("Hello.");
        }
    }

    if (n <= 2) {
        return 1;
    } else if (n%2 == 0) {
        return (recursiveloopy(n+1));
    }
    else {
        return (recursiveloopy(n-2));
    }
}
```

$T(n) =$ **Sol:** F : $O(n^3)$

Problem 2.e. Let $T(n)$ be the maximum stack depth of the following function, in terms of n.

```
double foo (int n)
{
    int i;
    double sum;
    if (n == 0) return 1.0;
    else {
        sum = 0.0;
        for (i = 0; i < n; i++)
            sum += foo (i);
        return sum;
    }
}
```

$T(n) =$ **Sol:** E : $O(n)$

Problem 3. Buggy Code: Exploring Planet Nine [16 points]

Last year, some scientists had an exciting idea: there might exist a ninth planet in our solar system! Planet Nine! Below is some Java code that will certainly not help us to explore Planet Nine. Notice all four excerpts are from the same codebase, and the later parts may refer to code in the earlier parts.

For each excerpt of code, there is a single bug that will either prevent compilation or will cause the program to crash.[1] **Please identify only one bug per part.**

Continued on the next page.

[1] Please do not identify warnings, such as the requirement that each class is in its own file, or that some variables may be unused. Do not identify stylistic problems, such as missing comments or bad variable names. Do not identify problems that cause the computation to produce a different answer than you think it should. Only identify problems that either prevent the program from compiling or cause it to crash.

Problem 3.a.

```
1. public class LandingVehicle {
2.
3.            protected String name;
4.            private int fuel;
5.
6.            LandingVehicle(){
7.            }
8.
9.            public void setName(String n){
10.                   name = n;
11.           }
12.
13.           public void dispatch(){
14.           }
15.
16.           protected boolean testVehicle(int i){
17.                   for (int wheel = 0; wheel < 4; wheel++){
18.                           testWheel(wheel);
19.                   }
20.                   return true;
21.           }
22.
23.           private void testWheel(int w) throws Exception {
24.                   boolean failed = true;
25.                   for (int i=0; i<10; i++){
26.                                   // Do test.
27.                   }
28.                   if (failed) throw new Exception("Bad wheels.");
29.           }
30.
31. }
```

Describe the bug here and fix it:

Solution: The exception in `testWheel` is never caught. To fix this, either add a try/catch block around testVehicle, or indicate that testVehicle throws an exception itself. (That might cause further problems elsewhere, but is ok here.)

Problem 3.b.

```
1.      public class Rover extends LandingVehicle {
2.
3.            static int roverCount = 0;
4.            public String pilot;
5.
6.            Rover(String p){
7.                    pilot = p;
8.            }
9.
10.           Rover() {
11.                   roverCount++;
12.           }
13.
14.           static int analyzeRovers(){
15.                   for (int i=0; i<roverCount; i++){
16.                           if (testVehicle(i)){
17.                                   return -1;
18.                           }
19.                   }
20.                   return 1;
21.           }
22. }
```

Describe the bug here and fix it:

Solution: The static method `analyzeRovers` cannot access the non-static method `testVehicle`. In this case, analyzeRovers should not be static. (You could also make testVehiclde static, but that would require also changing testWheel, and seems to defeat the purpose of trying to test a specific vehicle.)

Problem 3.c.

```
1. public class Probe extends LandingVehicle {
2.
3.            public String name;
4.
5.            Probe(String n){
6.                    name = n;
7.            }
8.
9.            boolean checkFuel(){
10.                   if (fuel > 10) return true;
11.                   else return false;
12.           }
13. }
```

Describe the bug here and fix it:

Solution: The variable `fuel` is private in the parent class and cannot be accessed here. You would probably change the variable to be `protected` in the parent class. You could also instead add a method to the parent class `getFuel` which returns the fuel.

Problem 3.d.

```
1. public class NinthPlanet {
2.
3.           private LandingVehicle[] rovers;
4.
5.           NinthPlanet(String[] names){
6.                   int numRovers = names.length;
7.                   rovers = new Rover[numRovers];
8.
9.                   for (int i=0; i<numRovers; i++){
10.                          rovers[i].setName(names[i]);
11.                  }
12.           }
13.
14.           public int dispatchRovers() {
15.                  for (int i=0; i<rovers.length; i++){
16.                         rovers[i].dispatch();
17.                  }
18.                  return 17;
19.           }
20. }
```

Describe the bug here and fix it:

Solution: There is a null-pointer exception, since the rovers[i] are never initialized. To fix this, you need to add a line like: `rovers[i] = new Rover()` for each rover.

And now you are read to launch for Planet Nine!

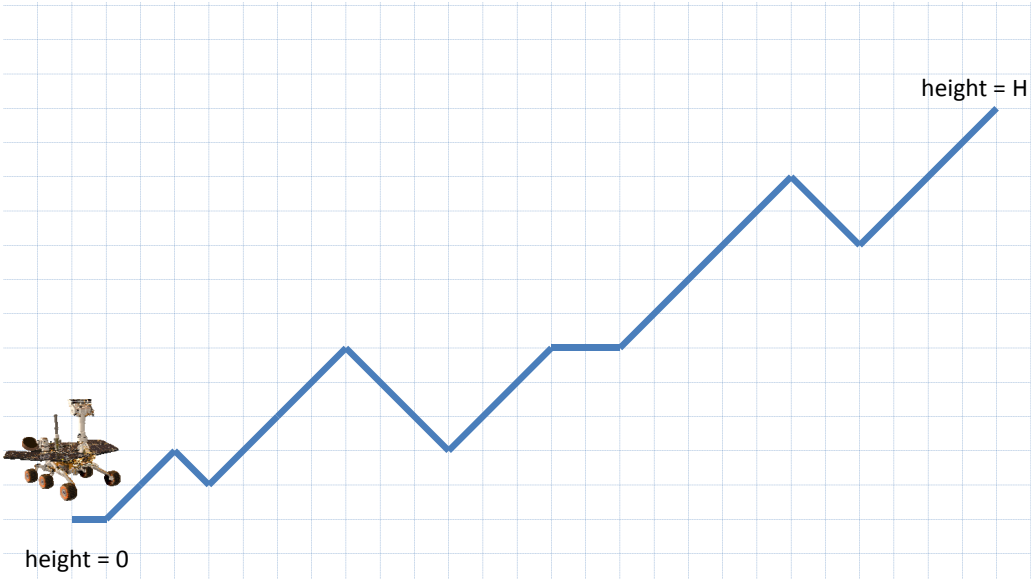
Problem 4. Exploring Planet Nine, Part 2 [18 points]

Our trusty rover has now arrived on Planet 9, and is beginning its exploration. The rover is currently at location "kilometer 0"at the base of a big mountain range. There is a route that continues for $n-1$ kilometers up through the extraterrestrial mountains up to a peak at height H.

From our reconaissance mission, we have created a map of the elevations along this path. That is, we have an array `A[0..n-1]` where `A[i]` is the altitude at kilometer i. All altitudes are integers. `A[0] = 0` is the base of the mountain and `A[n-1]` $= H$ is the top of the mountain. In between, it goes up and down and up and down, the way mountains do. (That is, the mountain is *not* strictly uphill.) Luckily, there are no cliffs. The changes in altitude are pretty gradual. Each

$$|A[i] - A[i+1]| \leq 1.$$

In this example, you see that $A[0] = A[1] = 0$, $A[2] = 1$, $A[3] = 2$, $A[4] = 1$, etc.



Given the array of the altitudes `A`, and a target height h, your job is to devise an algorithm to find one outpost i such that `A[i]`$= h$. In the example above, if the target height were 3, then it might return either kilometers 6, 10, or 12.

Describe your algorithm in at most two sentences:

Solution: Use binary search, maintaining the invariant that the target is within the current range of heights.

Give pseudocode specifying your algorithm precisely, in detail:

Solution:

```
searchTarget(A, begin, end, target)
    // Base case
    if (begin==end) then return begin

    // Invariant:
    // A[begin] <= target <= A[end]

    int mid = begin + ((end-begin)/2)
    if (A[mid] == target) return mid;
    else if (target < A[mid])
         return searchTarget(A, begin, mid, target)
    else // if (target > A[mid])
         return searchTarget(A, mid+1, end, target)

 searchTarget(A, target)
     searchTarget(A, 0, A.size()-1, target)
```

Explain why your algorithm works:

Solution: Throughout the execution, the algorithm maintains the invariant that at all times, $A[begin] \leq target \leq A[end]$. Since the altitudes change only by unit steps of size 1, this ensures (by continuity) that there is some point t between begin and end where $A[t] == target$. This invariant is maintained at each step of the binary search by comparing $A[mid]$ to the target and recursing on one of the two sides. The running time is $O(\log n)$.

Problem 5. Merge Sort with Linklist [18 points]

You are given the `LinkList` class with the following implementation:

```
public class LinkList {
    int num(); // returning the number of elements in the list
    int peekHead(); // return the first element of the list
    int peekTail(); // return the first element of the list
    void prepend(int i); // add the integer i to the head of the list
    void append(int i); // add the integer i to the tail of the list
    void deleteHead(); // delete the first element of the list
    void deleteTail(); // delete the last element of the list
}
```

You can assume all the functions are `public` and well implemented in $O(1)$ time.

Problem 5.a. Your job is to write a function `splitIntoTwo(a,b,c)` such that `a,b,c` are linklists. The function will move the first half of `a` into `b` and the second half of `a` into `c`. Thus, after the function, the list `a` will be empty. And the order of the integers in `b` and `c` are the same as when they were in `a`. If `a` has odd number of elements, `b` will have one more element than `c`.

```
void splitIntoTwo(LinkList a, LinkList b, LinkList c)
{
     count = a.num();
     half = count/2;
     if ((count % 2) == 1) half++;
     // Copy first half of list to b
     for (int i=0; i<count; i++){
          int e = a.peekHead();
          a.deleteHead();
          if(i<half)
            b.append(e);
                    else
            c.append(e);
     }
}
```

Problem 5.b. Given a linklist with n unsorted numbers, use the above class and implement merge sort by Java. Of course, you cannot use any ready-made Java code of sorting.

```
void mergeSortLinkList(LinkList ll)
{
       // Base case
       if (ll.num() == 1) return;

       // Divide in two
       LinkList left = new LinkList();
       LinkList right = new LinkList();
       splitIntoTwo(ll, left, right);

       // Recursive merge sort
       mergeSortLinkList(left);
       mergeSortLinkList(right);

       // Merging
       while ((left.num() > 0) && (right.num() > 0)) {
           int l = left.peekHead();
           int r = right.peekHead();
           if (l <= r) {  // stable merge sort?
               left.deleteHead();
               ll.append(l);
           }
           else {
               right.deleteHead();
               ll.append(r);
           }
       }

       while (left.num() > 0) {
           assert(right.num() == 0);
           int l = left.peekHead();
           left.deleteHead();
           ll.append(l);
       }

       while (right.num() > 0) {
           assert(left.num() == 0);
           int r = right.peekHead();
           right.deleteHead();
           ll.append(r);
       }
}
```

Problem 5.c. State the running time and extra space needed of your algorithm above with big O notation.

Solution: The running time is $O(n \log n)$.

Problem 6. Zig-zag Sorting [16 points]
Given an array `A` of n distinct elements, write the fastest algorithm to rearrange the elements of the array in a zig-zag fashion and state the time complexity in big O notation. The converted array should be in the form `A[0]` $<$ `A[1]` $>$ `A[2]` $<$ `A[3]` $>$ ...

```
Example:
Input:  A[] = {4, 3, 7, 8, 6, 2, 1}
Output: A[] = {3, 7, 4, 8, 2, 6, 1}

Input:  A[] =  {1, 4, 3, 2}
Output: A[] =  {1, 4, 2, 3}
```

You can express your algorithm in pseudo code.

Solution: There are several ways you might solve this problem. For example, you might sort the list (in $O(n \log n)$ time), and then interleave the first half and the second half of the resulting list. However, there is a simpler and faster solution!

Imagine trying to solve the problem greedily (or inductively) from left to right on the array, always maintaining a prefix that satisfies the zig-zag invariant. The base case (of one element) is trivially true, and it only necessary to determine the inductive step: how to increase the zig-zag prefix by one element. This results in the following pseudocode:

```
ZigZagSort(A[1,n])
   up = true
   for (i = 1 to n-1) do
      if (up == true)
          if (A[i] > A[i+1]) then swap(A, i, i+1)
      else if (up == false)
          if (A[i] < A[i+1]) then swap(A, i, i+1)
```

Consider the case where the sequence is supposed to increase. The key insight is that if the step $(A[i], A[i+1])$ is supposed to go up and does not, then by swapping them they are oriented in

the correct order, i.e., going down. There is, however, one more issue to check: by swapping i and $i+1$, are we disrupting the step $(A[i-1], A[i])$? In fact, no: if $A[i]$ to $A[i+1]$ is supposed to be up, then $A[i-1]$ to $A[i]$ is supposed to be down, and hence it is safe to decrease the value of $A[i]$, which is the only result of the swap.

The running time of the algorithm is $O(n)$.

Problem 2. Hash it! [14 points]

Problem 2.a. Suppose the following keys are inserted into a hash table in the following order:

A B C D E F G

The keys are inserted using the following hash function:

key	hash(key)
A	3
B	6
C	6
D	4
E	3
F	4
G	5

The keys are inserted using open addressing with linear probing. Indicate where each key is placed in the resulting array (drawn below with seven slots). Assume that the array size is fixed and does not double or shrink.

0	
1	
2	
3	
4	
5	
6	

Solution: 0.C 1.F 2.G 3.A 4.D 5.E 6.B

Problem 2.b. Suppose the following keys are inserted into a cuckoo hash table in the following order:

A B C D E F G H

The keys are inserted using the following two hash functions:

key	f(key)	g(key)
A	5	4
B	3	4
C	3	0
D	2	5
E	3	1
F	1	2
G	1	0
H	5	6

The keys are inserted using cuckoo hashing where hash function f is used for array X and hash function g is used for array Y. Assume that the array size is fixed and does not double or shrink, and that the hash functions are fixed and do not change.

Indicate where each key is placed in the resulting arrays after all the insertions complete, if all the insertions succeed. Otherwise, if all the insertions do not succeed, show which insertion fails.

X	
0	
1	
2	
3	
4	
5	
6	

Y	
0	
1	
2	
3	
4	
5	
6	

Solution:
X (0.__) (1.G) (2.D) (3.B) (4.__) (5.H) (6.__)
Y (0.C) (1.E) (2.F) (3.__) (4.A) (5.__) (6.__)

Problem 3. Who has the most bonus points? [22 points]

Professor Chelbert uses an AVL tree to keep track of the students in class. The key for the tree is the students' name, a string.[1] (The value attached to each name is a record containing student information, but you can ignore that for this problem.) The tree is sorted by name, and supports three AVL tree operations: insert, delete, and search (by name).

About halfway through the semester, Professor Chelbert decides to give out bonus points for particularly innovative ideas and provocative questions. And he wants to reward the student with the most bonus points with a special reward each week. In order to implement this new system, Professor Chelbert decides to modify the existing AVL tree to support two new operations:

- `void addBonus(String name, int points)`: adds the specified number of bonus points to the student with the specified name. Notice that the number of points added may be negative (if we want to subtract points from the student). However, a student may never have less than zero bonus points in total once the operation is complete.
- `String getMax()`: returns the name of the student with the most bonus points. If there is a tie, it returns the student whose name comes later in alphabetic order (i.e., if Alice, Bob, and Collin are tied, then it returns Collin).[2]

In this problem, your job is to help Professor Chelbert to implement this improved data structure. Notice that your goal is to *modify* the existing data in as minimal a manner as possible in order to implement these two operations efficiently. You should *not* design a whole new or different data structure to solve this problem. Your solution should be both time and space efficient.

Problem continued on next page.

[1] Luckily, all the students in his class have unique names.

[2] Professor Chelbert has always liked names that start with Z.

Problem 3.a. Explain, succinctly in **one to two sentences**, the main idea of how you plan to modify the existing AVL tree.

Solution: Augment the tree by storing in each node the maximum number of bonus points of any student in the subtree rooted at that node. This value can be used to guide the search to the right student, and it can be updated efficiently when nodes are added or deleted. (There is an alternate solution wherein you store the name as well as the points; this yields faster `getMax` operations but uses more space.)

Problem 3.b. Draw a picture that illustrates your idea.

Problem 3.c. Assume that the tree contains n students. (Assume that an integer takes $O(1)$ space.) What is the time complexity of each operation? How much additional space does your augmentation require? Use asymptotic (big-O) notation.

Time complexity for `addbonus`	:	$O(\log n)$
Time complexity for `getMax`	:	$O(\log n)$
Additional space for augmentation:	:	$O(n)$

Problem 3.d. Give the complete details for how you augment the AVL-tree. (Remember to include all the necessary cases when the tree is modified.)

Solution: First, each node u in the tree is augmented with two additional integers: $points$, that specifies the number of points for that student, and max, that stores the maximum number of points that any student has in the subtree rooted at u.

Second, we consider how to implement the two new operations:

- `addBonus(String n, int p)`: We search the AVL-tree in the usual way for the specified name n. (If the student does not exist in the tree, then we throw an exception.) Assume that we find some node u. Then we add p to the number of points at that student: $u.points = u.points + p$. If $u.points < 0$, then we set $u.points = 0$. Finally, we update the tree: starting with u, and for every node v on the path from u to the root, we update $v.max = \max(v.left.max, v.right.max, v.points)$. (Obviously we should check if the left and right children are null before dereferencing them.)

- `getMax()`: Let $m = root.max$, the largest bonus score of any student. We search the tree, looking for a student with m bonus points as follows. We begin at the root with $v = root$. We then walk down the tree as follows: if $v.right.max = m$, then go right; if $v.points = m$, then return $v.name$; else go left (in which case $v.left.max = m$). Since m is the maximum bonus score in the subtree, it has to be the case that it is either the maximum in the left subtree, the right subtree, or at the node itself. Since we want to prioritize names that are earlier in the alphabet, we check the options in the specified order.

Third, we consider how to maintain the newly augmented values when the tree is modified via insert and delete operations. There are three things to consider:

- When a *rotation* happens: When we perform a rotation on u, only u and u's parent change; all other subtrees remain the same. Let v be the parent of u. To update after a rotation of u, we set $u.max = v.max$, and then we set $v.max = \max(v.points, v.left.points, v.right.poitns)$ (avoiding null dereferences, of course).

- When an *insert* occurs: Insert the new node as per the usual AVL-tree insertion (as a leaf in the proper sorted order). Let u be the new node. Set $u.points = 0$ and $u.max = 0$. Since no student can have fewer than zero points, notice that all the max values in the tree remain correct and no further update is needed. Now perform rotations as needed in the normal way to maintain the balance of the tree.

- When a *delete* occurs: Assume node u is deleted. Recall that there are three cases for deleting a node in a binary tree. First, if node u is a leaf, we can just delete it. In that case, we start at the part of u and update every node v on the path to the root: $v.max = \max(v.left.max, v.right.max, v.points)$. Second, if node u has only one child, then we delete u, connecting the parent of u to the unique child of u. Again, in this case, we start at the part of u and update every node v on the path to the root: $v.max = \max(v.left.max, v.right.max, v.points)$. The third case is the most interesting: node u has two children. Here, we find the maximum (rightmost) node in u's left subtree; let v be this node. We delete v from the tree, and for every node w on the path from v to u, we update w: $w.max = \max(w.left.max, w.right.max, w.points)$. We then replace node u with node v (removing u from the tree and putting v in its place), and continue to update every node w on the path from v to the root: $w.max = \max(w.left.max, w.right.max, w.points)$. We then proceed to do rotations as notations in the normal way to maintain the balance of the tree.

These modifications ensure that for each of the operations performed, the max value is properly maintained as the maximum number of points of any student in the subtree.

Scratch Paper

Scratch Paper

Scratch Paper