Data structue 234218 – wet HW 1 dry part:

General data structure:

AVL:

A balanced AVL tree as learned in the lectures, capable of: (where n is the tree size)

-initialization time of O(1)

-insertion, deletion and access(or notification of non existence) time of O(log(n))

-destruction time of O(n)

- travel pre,post,in order of all nodes O(n)   
- saving walks and conditional walks (walks that saves nodes that fulfill a given condition in 1 array and those that do not in another) O(n)

-restoring itself by walks

- merging itself from 2 AVLs with the same order O(n)

- clearing itself O(n)  
- building an empty version of itself (without data) O(n)

Additional functionalities of AVL we implement(as taught in class):

-access to the highest ordered element of the AVL O(1)

-keeping track of the size of the AVL O(1)

- returning an array of elements according to a given total order O(n)

Smart\_pointers:

-keeps track of number of times a dynamically allocated item is referenced by a pointer.

If last pointer is destroyed, destroys the element.

All actions are done in O(1) except initialization which happens in O(p) where p is the time complexity of initializing the object smart pointer is pointing to.

Assignment specific data structures:

The main DS class will hold the following fields & methods:

* An AVL tree for trainers sorted by trainer ID (t\_AVL)
* An AVL tree for all pokemons sorted by pokemon ID(p\_AVL)
* An AVL tree for pokemon sorted by level and then by pokemon id (pL\_AVL).
* All methods outlined in the assignment description

Each trainer class will hold the following fields & methods (all fields and methods will remain public):

* The trainers ID
* An AVL tree for the trainer’s pokemon sorted by level and then by pokemon. (tp\_AVL)
* An initializer
* destroyer

Each pokemon class will hold the following fields & methods (all fields and methods will remain public):

* Its ID
* Its Level
* Its trainer’s ID
* An initializer
* destroyer

Complexity correctness:

We will explain the idea behind each public DS method and prove it works for the given upper complexity bounds.

***void\* Init():***

We will initialize DS which requires initializations of 3 AVLs, all 3 are initialized O(1), and so the entire initialization is O(1).

***StatusType AddTrainer(void \*DS, int trainerID):***

We check for the trainer’s existence in the trainer AVL (O(log(k)),

If ID is new we create a new trainer.

Trainers creation is O(1) because it contains only 1 int field and an AVL, both have initialization time of O(1).

Then, we add a new trainer reference to the trainers AVL (O(log(k)).

Total time complexity is C1\*(O(log(k)) + C2\*O(1) = (O(log(k))

(C1,C2 are constants)

***StatusType CatchPokemon(void \*DS, int pokemonID, int trainerID, int level):***

We check for existence of pokemon and trainer by the ID ordered AVL of both classes (O(log(n) and O(log(k)) respectively).

If pokemon does not exist and trainer does, we continue (having accessed the trainer while checking for existence)

We create pokemon (O(1) since pokemon contains only 2 basic fields).

We then add a the pokemon reference to the pokemon AVL and MAVL in DS (O(log(n)) and to the correct trainer’s MAVL (O(log(n))

Total time complexity is C3\*(O(log(n)) + C1\*(O(log(k)) + C2\*O(1) = O(log(n)+O(log(k)) <= O log(n) + O(k)

(Ci are constants)

***StatusType FreePokemon(void \*DS, int pokemonID):***

Check for existence of pokemon in p\_AVL (O(log(n)).

If it exist get its level and trainer ID.

Then using the pokemon ID and level we remove it from the p\_AVL and p\_MAVL (O(log(n))

We access its trainer through the t\_AVL (O(log(k)) and remove it from the trainers tp\_MAVL (O(log(n)).

We then free the pokemon O(1)

Total time complexity is C1\*(O(log(n)) + C2\*(O(log(k)) +C3\*O(1)= O(log(n)+O(log(k)) <= O log(n) + O(k)

(Ci are constants)

***StatusType LevelUp(void \*DS, int pokemonID, int levelIncrease):***

Check for existence of pokemon in p\_AVL (O(log(n)).

If it exist get its level and trainer ID.

We remove it from the MAVLs in the same way we did in FreePokemon, but we do not delete it or remove it from p\_AVL (O(log(n)+O(log(k)))

We then change its level O(1) and add it back to the MAVLs just as we did in CatchPokemon, already knowing it’s trainers ID O(log(n)+O(log(k)).

Total time complexity is C1\*(O(log(n)) + C2\*(O(log(k)) +C3\*O(1)= O(log(n)+O(log(k)) <= O log(n) + O(k)

(Ci are constants)

***StatusType GetTopPokemon(void \*DS, int trainerID, int \*pokemonID):***

Check if trainerID is negative and or (O(log(k)).

If so use the MAVL O(1) access to the top ordered element to get top pokemon.

If trainer is valid we use the tp\_MAVL and if trainerID < 0 then we use p\_MAVL.

Total time complexity is C2\*(O(log(k)) <= O(k)

(Ci are constants)

***StatusType GetAllPokemonsByLevel(void \*DS, int trainerID, int \*\*pokemons, int numOfPokemon):***

If trainer id<0 we use p\_MAVL’s ordered walk to return the pokemons array (O(n))

If trainer id >0 we check if its valid O(log(k)). If so, we use tp\_MAVL’s ordered walk to get pokemon array

(O(ntrainerID))

Total time complexity is O(n) if trainerID <0 and O(ntrainerID) + O(log(k)) <= O(ntrainerID) + O(k) if trainer ID >0

***StatusType EvolvePokemon(void \*DS, int pokemonID, int evolvedID):***

Check if pokemon ID or evolved ID exists in p\_AVL (O(log(n))).

If pokeomn ID exists and evolved ID does not, remove pokemon from p\_AVL,p\_MAVL and tp\_MAVL as is done in FreePokemon (except we don’t free the pokemon) (O(log(n)+O(log(k))).

Then we change the ID of the pokemon O(1) and return in back to p\_AVL,p\_MAVL and tp\_MAVL as is done in catch pokemon (O(log(n)+O(log(k))).

Note that we have the correct trainer ID from the initial check of pokemonID’s existence.

Total time complexity is C1\*(O(log(n)) + C2\*(O(log(k)) +C3\*O(1)= O(log(n)+O(log(k)) <= O log(n) + O(k)

(Ci are constants)

***StatusType UpdateLevels(void \*DS, int stoneCode, int stoneFactor):***

We save 2 different conditional walks of each pokemon MAVL according to the

(pokemonID % stoneCode == 0 ) condition O(n) where n is the size of the tree.  
for the DS we have p\_AVL and p\_MAVL (both of size n thus time of O(n)) and for the trainer’s tp\_MAVL we have a total of k MAVLs with a total size of n so 2 conditional walks looped across all trainers will take = O(n+k).

The total time complexity up to this point is O(n+k)

We will multiply the level of all pokemon in the arrays that fulfilled the condition by stone factor O(1) per pokemon and at most O(n) for all pokemon.

Since the multiplication is by a constant factor across all multiplied pokemon, the positive conditioned walks will maintain correct order among themselves.

Then, we will restore all the AVL trees from the walks. Since we have C1\*n nodes across k+C2 trees, this will take (according to the same calculation of the walks) O(n+k) in both time and space complexity.

We will then merge each positive condition tree with it’s counterpart negative condition tree and, once again according to the previous calculation, this will take O(n+k) time and space.

We have a total of C4\*O(n+k)+C5\*O(n) = C6\*O(n+k) time and space complexity.

(Ci are constants)

***void Quit(void \*\*DS):***

we destroy all of the AVLs and MAVLs, taking O(n) time for each global AVL/MAVL and O(k+n) across all trainer MAVLs. After destroying each trainers MAVL and before destroying t\_AVL we destroy each trainer (for a total of O(k) for all k trainers together). Before destroying p\_AVL, we free all pokemon (O(n) across all pokemon).

Then we free the empty DS.

We have a total of C4\*O(n+k)+C5\*O(n) = C6\*O(n+k) time complexity.

(Ci are constants)

***Space Complexity:***

Overall, the total size of all AVLs/MAVLs is a constant multiple of n+k.  
when updating levels we add at most another constant multiple of n+k space (explained in the UpdateLevels function analysis).  
  
these tree structures are the only structures of variable size and all other data is constant in size and is bound by n and k in number (referring to the constant fields of pokemon/trainers who’s total space is bound by a multiple of n+k and constant size DS space which occurs only once).   
  
overall we have at most a constant multiple of n+k space used, thus our space complexity is   
O(n+k)