Data structure 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 pokemon smart_pointer to the pokemon p_AVL and pL_AVL in DS (O(log(n)) and to the correct trainer's tp_AVL (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 pL_AVL (O(log(n))
We access its trainer through the t_AVL (O(log(k)) and remove it from the trainers tp_AVL (O(log(n)).

We then free the pokemon O(1) (Done automatically by smart_pointer)

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 perform freePokemon on it (O(log(n)+O(log(k)))
We then catch a pokemon with identical pokemon id and trainer id but with level=oldlevel+levelincrease.
(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 or positive O(1) (else return invalid O(1)).
If negative, use the pL_AVL O(1) access to the top ordered element to get top pokemon.
If positive, get trainer if valid from t_AVL O(log(k).
If it exists use tP_AVL O(1) access to top pokemon. If trainer does not exist return failure

Total time complexity is C1*(O(log(k)) +C2*O(1) <= O(k) in case of positive trainer ID
And O(1) in case of negative id.
(Ci are constants)

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

If trainer id<0 we use pL_AVL'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_AVL's ordered walk to get pokemon array ($O(n_{trainerID})$)
We then allocate an int array for the client and fill it with the pokemon ID accoriding to the pokemon array we received (O(n) or $O(n_{trainerID})$ respectively).

Total time complexity is O(n) if trainerID <0 and $O(n_{trainerID})$ + O(log(k)) <= $O(n_{trainerID})$ + 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)))
And get the original pokemon's trainer id and level.

If pokeomn ID exists and evolved ID does not, free pokemon (O(log(n)+O(log(k))).
Then catch pokemon with the same level and trainer ID as original pokemon, but with evolved 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 UpdateLevels(void *DS, int stoneCode, int stoneFactor):*

On any given pokemon level ordered tree (of variable size q):
Make 2 copies of the tree (istree & isnotree) (O(q)).
Preform a conditional removing walk on each copy so that isTree has all the pokemon whos level needs updating and is not tree has all the rest of the pokemon (O(q)).
We then preform an update on each key and data(optional) of istree (O(q)).
Finally we copy construct the original tree with the merging of the changed isTree and the unchanged isNotTree (O(q)).

This operation has a total of (O(q)) time complexity.

We perform this operation on the pL_AVl of DS and on the tp_AVL of each trainer. Notice that since we save pokemon by smart pointer, we will only update the data once across all trees (meaning only changing it in pL_AVL).

Since $\sum_{i=1}^{k} n_{trainer\_i\_ID} = n$, this actions will take O(n) for the pL_AVL and O(n) for all trainer's tp_AVL. Factoring the access to each trainer, which will be performed by a walk on the trainer tree, we get a total time complexity of O(n) + O(k)

*void Quit(void **DS):*

We walk across all trainers (O(k)) and for each trainer we delete it's tp_AVL (O($\sum_{i=1}^{k} n_{trainer\_i\_ID} = n$)
Then, we destroy all of DS's AVLs (O(n) for pokemon and O(k) for trainers).
All pokemon are freed in O(1) per pokemon once the smart pointer is removed from the last tree (total O(n))
Then we free the empty DS.

We have a total of C4*O(k)+C5*O(n)+ C6*O(1) = 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). Note that the int arrays allocated to the client by getAllPokemonByLevel, are under his responsibility, spaceManagment wise and so not count towards (variable*n) space complexity of our code