Data Structures 1

Wet 2 – dry part

Names of students: Matam Miller, Shahar Fox

Overview of the Data Structures we used:

We use four principal data structures (plus a small “banned ID” hash):

1) **Union–Find** to group teams into “components":

Each Team node has:

- parent: pointer to another Team or to itself if it is a “root.”
- active: boolean indicating if it is the currently alive root (true) or a loser in a merge (false).
- record: integer sum of all jockey wins minus losses in that root’s entire set.
- teamId: the official ID for that team.

2) **Hash Table** for Teams by teamId.

- Key = integer teamId, Value = pointer to the Team node.
- Used in add_team, findTeamById, merge_teams checks and more.

3) **Hash Table** for Jockeys by jockeyId.

Key = integer jockeyId, Value = pointer to a Jockey struct containing:

- jockeyId
- pointer to the Team node it belongs to
- record: personal jockey win–loss balance

4) **Hash Table** for Teams by record.

- Key = integer (the “record”)
- Value = chain of pointers to Team nodes with that record.
- We primarily store root teams in their record’s chain, enabling quick scans for unite_by_record(record).

5) **Hash Table** for Banned IDs.

- Once a team ID is used and then "loses" in a merge, that ID is "banned" from being added again.
- Checking or inserting in this structure is O(1) expected.

Each hash table uses separate chaining with singly linked lists. We pick a prime size (e.g., 40 001) and a simple mod-based hash function.

Union-Find implementation sketch:

When we merge team A and team B:

- We pick a winner root and a loser root.
- We set loser->active = false and loser->parent = winner.
- The winner's record is set to be the sum of both records.
- The loser's ID is inserted into the "banned" structure so that future add_team(loserId) fails.

When we want to find which team a jockey belongs to, we do a find up the parent pointers until we reach an active root. The standard union–find approach ensures near-constant time for repeated merges/finds.

Overview of the functions and their algorithms:

Add_team(teamId):

Algorithm:

1. If teamId <= 0, return INVALID_INPUT.
2. Check the banned hash. If teamId is present, return FAILURE.
3. Look up teamId in the Team hash. If found, return FAILURE.
4. Allocate a new Team object with:
    - parent = self
    - active = true
    - record = 0
    - teamId = teamId
5. Insert it into the Team hash and into the "record = 0" chain in the record hash.
6. Return SUCCESS.

Time Complexity:

- Step 2 (banned hash check) is an expected O(1) hash lookup.
- Step 3 (team hash check) is O(1) expected.

- Steps 5 (insert) is O(1) expected.

Hence the entire function is O(1) expected amortized.

Mathematical Justification:

- A hash table with capacity H and a randomizing hash function has average chain length $\alpha = N/H$, where N is the current number of elements. The expected cost of insertion or lookup in separate chaining is $O(1+\alpha)$. We keep α effectively constant by picking a suitable table size (like 40,001). Therefore, each hash operation is O(1) on average.
- No other step introduces more than a constant overhead, so total is O(1).

<u>Add_jockey(jockeyId, teamId):</u>

Algorithm:

1. If jockeyId <= 0 or teamId <= 0, return INVALID_INPUT.
2. Look up jockeyId in the Jockey hash. If found, return FAILURE.
3. Find teamId in the Team hash. If not found, return FAILURE.
4. Allocate a new Jockey object with record = 0 and pointer to the found team.
5. Insert the jockey into the Jockey hash.
6. Return SUCCESS.

Time Complexity: All lookups/insertions are O(1) average in a hash, so total is O(1) expected.

Math: same separate chaining hash argument as before.

<u>Update_match(victoriousJockeyId, losingJockeyId):</u>

Algorithm:

1. If the IDs are invalid or the same, return INVALID_INPUT.
2. Look up each jockey in the Jockey hash. If either is missing, return FAILURE.
3. For each jockey, do Union–Find find to get the root Team. If they have the same root or if either root is inactive, return FAILURE.
4. Increase winner->record by 1, decrease loser->record by 1.
5. Remove each root from the record hash: O(1).
6. rootW->record++; rootL->record--.
7. Re-insert them in the record hash with updated record.
8. Return SUCCESS.

Time Complexity:

- Two hash lookups (O(1) each).
- Two union-find “find” calls. Typically union-find with path compression is near O(1) amortized.
- Remove + Insert in the record hash: O(1) each.

Therefore, update_match is O(1) expected amortized.

Math:

- Union–Find with path compression has well-known upper bound $O(\alpha(n))$ per operation, where α is the inverse Ackermann function. This is < 5 for all practical n. So it is effectively constant time.
- The hash operations are again O(1) expected. Summing constant or near-constant terms yields O(1) total.

<u>Merge_teams(teamId1, teamId2):</u>

Algorithm:

1. If teamId1 <= 0 or teamId2 <= 0 or the IDs are the same, return INVALID_INPUT.
2. Look up both teams in the Team hash. If either missing, return FAILURE.
3. Union–Find find their roots. If the roots are the same or inactive, return FAILURE.
4. Compare root records. The higher record is the winner. On tie, pick root1 as winner.
5. Remove both from the record hash.
6. Sum the records into the winner’s root.
7. Mark loser as active = false, ban loser->teamId, link loser->parent = winner.
8. Re-insert winner with updated record in the record hash.

Time Complexity:

- 2 team hash lookups => O(1) each.
- 2 union-find finds => O(1) amortized.
- removal + insertion in record hash => O(1) each.
- insertion into banned hash => O(1).

Hence total is O(1) expected.

Math:

- Again, each operation on a hash is expected O(1), union-find find is $O(\alpha(n) \sim O(1)$.
- Summation => O(1).

Unite_by_record(record):

Algorithm:

1. If record <= 0, return INVALID_INPUT.
2. In the record hash for record, find exactly one active root with that record. In the record hash for -record, find exactly one active root with -record.
3. If not exactly one in each, return FAILURE.
4. Merge them (like merge_teams): the positive-record root is the winner, negative is the loser.

Time Complexity:

- Checking each chain is O(k) where k is the number of teams that currently have that record. Usually we expect a small chain or at worst O(m) if many teams share the same record. But typically, the problem statement says “if exactly 2 teams with ±record, then unite,” so it is effectively O(1).
- The final merge is O(1).

So unite_by_record is O(1) average (amortized).

Math:

- The chain length for a hash bucket is expected α, typically a small constant if we spread out the record values well.
- Merging is O(1). Summation => O(1).

Get_jockey_record(jockeyId):

Algorithm:

1. If jockeyId <= 0, return INVALID_INPUT.
2. Look up jockeyId in Jockey hash. If absent, return FAILURE.
3. Return that jockey’s record field.

Time Complexity: Single hash lookup => O(1).

Math: E(chain length)≈α. So O(1) average.

Get_team_record(teamId):

Algorithm:

1. If teamId <= 0, return INVALID_INPUT.
2. Team hash lookup. If absent, return FAILURE.
3. Union-Find “find” to get the root. If that root is not active, return FAILURE.
4. Return root->record.

Time Complexity:

- O(1) hash, plus O(1) union-find.
  So total O(1).

Math: same hashing + union-find argument.

Proving the Time Complexity requirements:

Hash-Table O(1) average:

A separate-chaining hash with table size H and a uniform hash function typically yields an expected chain length $\alpha = N/H$. So an insertion or lookup is $O(1+\alpha)$ expected. If we pick H around the maximum N, α stays near 1, giving us O(1) average time.

Formally, for each operation, the expected insertion or search cost is:

E[Time] = $O(1+\alpha)$, where α is constant or near-constant. Summing constants yields O(1).

Union-Find O(α(n)):

In typical union-find with path compression, the amortized cost of each union or find is α(n), where α is the inverse Ackermann function. For all practical n up to huge sizes, $\alpha(n) \leq 4$. This is effectively **constant** time. Even in a simplified approach, merges remain short if we always point the loser to the winner. Summation => near **O(1)**.

Space Complexity O(n+m):

We must store:

1. One Team node for each team that is ever added, including those that lost merges. That is up to m.
2. One Jockey node for each jockey => n.
3. Each node is in exactly one chain for its ID or record, plus an array of size ~40 001 for each hash. The sum of pointers and overhead remains linear in n+m.
4. The "banned ID" hash can have at most m entries (once per losing ID).
5. We do not keep any additional structures that grow faster than n or m.

Hence total memory is at most some constant factor times (n+m).

Formally:

MemoryUsed = O(n) + O(m) = O(n+m).

Conclusion:

- The data structure is built around Union–Find for merging teams plus hash tables for quick lookups (by team ID, jockey ID, record, banned ID).
- All major functions achieve O(1) expected time (amortized). The union–find "find" is near-constant due to path compression, and each hash operation is O(1) on average.
- The entire system uses O(n + m) memory, meeting the problem's requirements.

Hence we satisfy the assignment's time and space complexities with a combination of hashing and union–find.