Wet HW 2 (wet part)

● Graded

Group

גיא פרידמן
דניאלה כהן
View or edit group

Total Points

100 / 100 pts

Autograder Score

100.0 / 100.0

Passed Tests

STL

Compilation

Memory Leaks

0) Test 0 (2/2)

1) Test 1 (2/2)

2) Test 2 (2/2)

3) Test 3 (2/2)

4) Test 4 (2/2)

5) Test 5 (2/2)

6) Test 6 (2/2)

7) Test 7 (2/2)

8) Test 8 (2/2)

9) Test 9 (2/2)

10) Test 10 (2/2)

11) Test 11 (2/2)

12) Test 12 (2/2)

13) Test 13 (2/2)

14) Test 14 (2/2)

15) Test 15 (2/2)

16) Test 16 (2/2)

17) Test 17 (2/2)

18) Test 18 (2/2)

19) Test 19 (2/2)

20) Test 20 (2/2)

21) Test 21 (2/2)

22) Test 22 (2/2)

23) Test 23 (2/2)

24) Test 24 (2/2)

25) Test 25 (2/2)

26) Test 26 (2/2)

27) Test 27 (2/2)

28) Test 28 (2/2)

29) Test 29 (2/2)

30) Test 30 (2/2)

31) Test 31 (2/2)

32) Test 32 (2/2)

33) Test 33 (2/2)

34) Test 34 (2/2)

35) Test 35 (2/2)

36) Test 36 (2/2)

37) Test 37 (2/2)

38) Test 38 (2/2)

39) Test 39 (2/2)

40) Test 40 (2/2)

41) Test 41 (2/2)

42) Test 42 (2/2)

43) Test 43 (2/2)
44) Test 44 (2/2)
45) Test 45 (2/2)
46) Test 46 (2/2)
47) Test 47 (2/2)
48) Test 48 (2/2)
49) Test 49 (2/2)

Autograder Results

Autograder Output

```
Please ensure that you add your other group member to this submission.
A tutorial can be found here: https://shorturl.at/ttSty
Valgrind NO LEAKS
Test #0  Passed
Test #10  Passed
Test #20  Passed
Test #30  Passed
Test #40  Passed
```

STL

Compilation

Memory Leaks

0) Test 0 (2/2)

1) Test 1 (2/2)

2) Test 2 (2/2)

3) Test 3 (2/2)

4) Test 4 (2/2)

5) Test 5 (2/2)

6) Test 6 (2/2)

7) Test 7 (2/2)

8) Test 8 (2/2)

9) Test 9 (2/2)

10) Test 10 (2/2)

11) Test 11 (2/2)

12) Test 12 (2/2)

13) Test 13 (2/2)

14) Test 14 (2/2)

15) Test 15 (2/2)

16) Test 16 (2/2)

17) Test 17 (2/2)

18) Test 18 (2/2)

19) Test 19 (2/2)

20) Test 20 (2/2)

21) Test 21 (2/2)

22) Test 22 (2/2)

23) Test 23 (2/2)

24) Test 24 (2/2)

25) Test 25 (2/2)

26) Test 26 (2/2)

27) Test 27 (2/2)

28) Test 28 (2/2)

29) Test 29 (2/2)

30) Test 30 (2/2)

31) Test 31 (2/2)

32) Test 32 (2/2)

33) Test 33 (2/2)

34) Test 34 (2/2)

35) Test 35 (2/2)

36) Test 36 (2/2)

37) Test 37 (2/2)

38) Test 38 (2/2)

39) Test 39 (2/2)

40) Test 40 (2/2)

41) Test 41 (2/2)

42) Test 42 (2/2)

43) Test 43 (2/2)

44) Test 44 (2/2)

45) Test 45 (2/2)

46) Test 46 (2/2)

47) Test 47 (2/2)

48) Test 48 (2/2)

49) Test 49 (2/2)

Submitted Files

▾ NewTeam.h

Download

```cpp
//
// Created by Guy Friedman on 28/01/2025.
//

#ifndef NEWTEAM_H
#define NEWTEAM_H

#define SELECT_ID_BY_MAX_RECORD(record1,record2,id1,id2) (record1>=record2?id1:id2)
#include <cassert>


class NewTeam {
protected:
    int id;
    int size;
    int record;
    NewTeam* root_by_mass;
    NewTeam* root_by_record; //for now, it is not used
public:

    NewTeam() = delete;

    NewTeam(int id) : id(id), size(1), record(0), root_by_mass(this), root_by_record(this) {}

    /*
    NewTeam(int id,int newRec, int newSize) : NewTeam(id) { //todo maybe can delete
        this->record = newRec;
        this->size = newSize;
    }
    */

    ~NewTeam() {
        this->root_by_mass = nullptr;
        this->root_by_record = nullptr;
    }

    NewTeam* get_root_by_mass() {
        if (this->root_by_mass == this) {
            return this;
        }
        NewTeam* actual_root_by_mass = this->root_by_mass->get_root_by_mass();
        this->root_by_mass = actual_root_by_mass;
        return actual_root_by_mass;
    }

    int get_id() {
        return this->id;
    }

    int get_record() {
        //make sure that it's only called on the leader
        assert(this ==  this->root_by_mass);

        return this->record;
    }

    bool isTeam(int id) {
        //make sure that it's only called on the leader
        assert(this ==  this->root_by_mass);
        return this->id == id;
    }

    void unite_team(NewTeam* other) { //done - make sure in main that 'this' is team 1 and 'other' is team 2
        //make sure main has taken care of its responsibilities for union
        this->verify_main_for_union(other);

        //get new parameters after union
        int new_id = SELECT_ID_BY_MAX_RECORD(this->record, other->record, this->id, other->id); //SELECT_ID_BY_MAX_RECORD(record1,record2,id1,id2) (record1>=record2?id1:id2)
        int new_size = this->size + other->size;
        int new_record = this->get_record() + other->get_record(); //        int new_record = this->record + other->record;

        //regular unite
        this->unite_helper(other,0);

        //update_merge_details(int new_id, int new_size, int new_record)
        //I don't care who is actually the root, I'll just update both to save me the hassle of checking
        this->update_merge_details(new_id,new_size,new_record);
        other->update_merge_details(new_id,new_size,new_record);
    }

    /**
     *just unite by size, don't care about id, record and other stuff,
     *unite team takes care of this
     */
    void unite_helper(NewTeam* other, int make_signature_different_to_prevent_confusion) {
        //make sure main has taken care of its responsibilities for union
        this->verify_main_for_union(other);

        //retrieve original team sizes
        int this_size = this->size;
        int other_size = other->size;

        //unify by moving small (not small 'mamash') tree root to point at big tree root
        if (this_size <= other_size) {
            this->root_by_mass = other; //other->root_by_mass;
        } else {
            other->root_by_mass = this; //this->root_by_mass;
        }
```

```cpp

        //update sizes (even though it's updated in 'unite_team', make sure)
        int sum_size = this_size + other_size;
        this->size = sum_size;
        other->size = sum_size;
    }


    void verify_main_for_union(NewTeam* other) {
        //should only call to unite by roots - responsibility of the main
        assert(other == other->root_by_mass);
        assert(this == this->root_by_mass);
    }

    void update_merge_details(int new_id, int new_size, int new_record) {
        this->id = new_id;
        this->record = new_record;
        this->size = new_size;
    }

    bool check_active_immediate(int idVerify) {
        return ((this->id == idVerify) && (this->check_active_immediate()));
    }
    bool check_active_immediate() {
        return this->root_by_mass == this || this->root_by_mass->id == this->id;
    }

    void winMatch() {
        assert(this->root_by_mass == this);
        ++this->record;
    }

    void loseMatch() {
        assert(this->root_by_mass == this);
        --this->record;
    }






};



#endif //NEWTEAM_H

```

▾ NewTeamArr.h

Download

```cpp
//
// Created by Guy Friedman on 28/01/2025.
//

#ifndef NEWTEAMARR_H
#define NEWTEAMARR_H
#include "NewTeam.h"
#include "ChainHashArray.h"


class NewTeamArr : public ChainHashArray<NewTeam> {
public:

    NewTeamArr() : ChainHashArray<NewTeam>() {}

    ~NewTeamArr() = default;

    bool team_outdated(int team_id) {
        //@brief - get team root, compare root id to 'team_id'
        //legacy function, instead of rewriting the code to remove it,
        //I just utilised existing code and logic.
        bool team_is_active = this->team_active(team_id);
        return !team_is_active;
    }

    bool team_active(int team_id) {
        //done - cri8 dis foonktzion

        //find root of team with 'team_id' key
        NewTeam* root_team_of_team_id = this->get_root_of(team_id);

        //if team doesn't exist return false
        if (root_team_of_team_id == nullptr) {
            return false;
        }

        //get the id of the root
        int id_of_root = root_team_of_team_id->get_id();

        return id_of_root == team_id;
    }

    void unite_teams(int team1, int team2) {
        //@brief - team1->unite_team(other);

        //make sure that we got existing teams
        assert(this->team_active(team1));
        assert(this->team_active(team2));

        //get the roots of team1 and team2
        NewTeam* team_one = this->get_root_of(team1);
        NewTeam* team_two = this->get_root_of(team2);

        //make sure that we received valid teams
        assert(team_one->get_id() == team1);
        assert(team_two->get_id() == team2);

        //execute union by utilising existing code and logic
        team_one->unite_team(team_two);
    }

    NewTeam* get_root_of(int team_id) {
        //@brief - get_root_by_mass() of team held by key 'team_id'

        //get team with key 'team_id'
        NewTeam* temp_placeholder = this->find(team_id);

        //maybe there isnt any team in initial 'team_id'
        if (temp_placeholder == nullptr) {
            return nullptr;
        }

        //get root of team 'hana"l'
        NewTeam* root_team = temp_placeholder->get_root_by_mass();

        return root_team;
    }

    NewTeam* get_jockeys_actual_team(int id) {
        //@brief - get team with id, get root of team with id, return root

        //get root of team corresponding with key of 'id'
        NewTeam* temp_placeholder = this->get_root_of(id);

        //make sure no funny business is going on
        assert(temp_placeholder != nullptr);
        assert(this->team_active(temp_placeholder->get_id()));

        return temp_placeholder;
    }

    bool check_active_immediate(int team_id) {
        NewTeam* temp_placeholder = this->find(team_id);
        if (temp_placeholder == nullptr) {
            return false;
        }
        return temp_placeholder->check_active_immediate(team_id);
    }

};

```

```


#endif //NEWTEAMARR_H

```

▾ **RecordArr.h** Download

```
//
// Created by Guy Friedman on 28/01/2025.
//

#ifndef RECORDARR_H
#define RECORDARR_H
#include "Record.h"
#include "ChainHashArray.h"
#include "NewTeam.h" //todo maybe change this to a forward declaration instead of inclusion


class RecordArr : public ChainHashArray<Record> {
    public:

    RecordArr() : ChainHashArray<Record>() {}

    ~RecordArr() = default;

    void add_team_to_record(NewTeam* team, int team_id, int record_id) {
        //@brief - check if record exists, if so then add, if not create, store and then add

        //make sure input is valid
        assert(team_id > 0);
        assert(team != nullptr);

        //get relevant record
        Record* record = this->find(record_id);

        //if the record doesn't already exist, create it and store it.
        if (record == nullptr) {
            record = new Record(record_id);
            this->insert(record_id,record);
        }

        //add team to relevant record
        record->insert(team_id, team);
    }

    void remove_team_from_record( int team_id, int record_id) {
        //@brief - use this->remove(), then check if the record is empty, if it is - then delete it

        //retrieve relevant record
        Record* record = this->find(record_id);

        //verify input - make sure nothing funny is happening
        assert(record != nullptr);

        //remove relevant team from relevant record
        record->remove(team_id);

        //if the record is empty now, delete it.
        if(record->isEmpty()) {
            this->deleteItem(record_id);
        }
    }

    bool can_unite_by_record(int record_id) {
        //@brief - find record of (record_id) and record of (-record_id),
        //@brief - check if they are singletons,
        //@brief - if either of them isn't singleton or doesn't exist - return false
        //@brief - if both are singletons - return true

        //make sure input is valid
        assert(record_id > 0);

        //retrieve relevant records
        Record* positive_record = this->find(record_id);
        Record* negative_record = this->find(-record_id);

        //if either of them doesn't exist, can not unite by record
        if (positive_record == nullptr || negative_record == nullptr) {
            return false;
        }

        //if at least one of them is not a singleton, can not unite by record
        if (!((positive_record->isSingleton())&&(negative_record->isSingleton()))) {
            return false;
        }

        //if none of the previous checks failed
        return true;
    }

    int return_team_id_of_singleton_record(int record) {
        //@brief - get record, assert it is singleton, pop team, store team id, re-insert team, return saved id.

        //get proper record
        Record* record_singleton = this->find(record);

        //make sure record exists and is actually singleton
        assert(record_singleton != nullptr);
        assert(record_singleton->isSingleton());

        //get team from record
        NewTeam* team = record_singleton->pop();

        //save team id
        int team_id = team->get_id();

        //return team to its place in record_singleton
        record_singleton->insert(team_id, team);

        //return the id of the team that we got
        return team_id;

    }

};



#endif //RECORDARR_H

```

Jockey.h

Download

```
//
// Created by Guy Friedman on 27/01/2025.
//

#ifndef JOCKEY_H
#define JOCKEY_H



class Jockey {
protected:
    int id;
    int record;
    int team_id;
public:
    Jockey() = delete;

    /*
    Jockey(int id) {
        this->id = id;
        this->record = 0;
        this->team_id = 0;
    }
    */

    Jockey(int id, int team_id) {
        this->id = id;
        this->record = 0;
        this->team_id = team_id;
    }

    virtual ~Jockey() = default;

    int getId() const {
        return this->id;
    }

    int getRecord() const {
        return this->record;
    }

    int getTeamId() const {
        return this->team_id;
    }

    void setId(int id) {
        this->id = id;
    }

    void setRecord(int record) {
        this->record = record;
    }

    void setTeamId(int team_id) {
        this->team_id = team_id;
    }

    void winMatch() {
        ++this->record;
    }

    void loseMatch() {
        --this->record;
    }
};



#endif //JOCKEY_H

```

ChainHashArray.h Download

```cpp
//
// Created by Guy Friedman on 24/01/2025.
//

#ifndef CHAINHASHARRAY_H
#define CHAINHASHARRAY_H
#include "DeQue.h"
#include "Pair.h"
#include <cassert>
constexpr int MAX_FILL_RATIO_CHAIN_HASH_ARRAY = 10;
constexpr int INITIAL_SIZE_CHAIN_HASH_ARRAY = 16;
#define MAX(a,b) (a>b)?a:b
#define MIN(a,b) (a<b)?a:b
#define MAKE_DOUBLE(a) (a*2)
#define HALF_OF(a) (a*0.5)
#define QUARTER_OF(a) (a*0.25)
#define EMPTY (0)




template<class T>
class ChainHashArray {
private:
    DeQue<Pair<T>>* data_arr;
    int arr_size;
    int amount_of_items;
    int capacity;

public:

    ChainHashArray() : capacity(EMPTY) {
        this->data_arr = nullptr;
        this->arr_size = INITIAL_SIZE_CHAIN_HASH_ARRAY;
        this->amount_of_items = EMPTY;
        this->updateCapacity();
        this->initializeArray();
    }

    ~ChainHashArray() {
        delete[] data_arr;
    }

    void insert(int key, T* value) {
        this->insert(key, value, true);
    }

    T* find(int key) {
        int index = this->calcIndex(key);
        Pair<T> temp = Pair<T>(key);//new Pair<T>(key);
        Pair<T>* toFind = this->data_arr[index].find(temp);
        //delete temp;
        return toFind == nullptr ? nullptr : toFind->value;
    }

    T* remove(int key) {
        return this->remove(key,true);
    }

    void deleteItem(int key) {
        delete this->remove(key);
    }

    int size() {
        return this->amount_of_items;
    }

    T* popRandom() {
        if (this->amount_of_items == EMPTY) {
            return nullptr;
        }
        for (int i = 0; i < this->arr_size; i++) {
            DeQue<Pair<T>>& desiredSlot = this->data_arr[i];
            if (desiredSlot.getSize() != EMPTY) {
                Pair<T>* temp = desiredSlot.pop();
                --this->amount_of_items;
                T* tempVal = temp->extract();
                delete temp;
                return tempVal;
            }
        }
        //NOTICE - no check for this->checkUpdateArr();
        return nullptr; //if we got here then there is a problem
    }

    void clear() {
        int items_amount = this->amount_of_items;
        for (int i = 0; i < items_amount; i++) {
            this->popRandom();
        }
        assert(this->amount_of_items == EMPTY);
        this->checkUpdateArr();
    }

protected:

    void insert(int key, T* value, bool checkForUpdateSize) {
        int insertionIndex = this->calcIndex(key);
        DeQue<Pair<T>>& desiredSlot = this->data_arr[insertionIndex];
        Pair<T>* newItem = new Pair<T>(key, value);
        desiredSlot.append(newItem);
        ++this->amount_of_items;
        if (checkForUpdateSize) {
            this->checkUpdateArr();
        }
    }

    /**
     *the same functionality as regular insert, but without
     *checking for size updates, for inside methods and usages
     *
     * @param key
     * @return
     */
    void insertImmediate(int key, T* value) {
        this->insert(key, value, false);
    }

    void insertImmediate(Pair<T>* newItem) {
        this->insertImmediate(newItem->key);
    }

    T* remove(int key, bool checkForUpdateSize) {
        int index = this->calcIndex(key);
        DeQue<Pair<T>>* toRemove = &(this->data_arr[index]);
        Pair<T> toRemovePair = Pair<T>(key);

        Pair<T>* toFind = (toRemove->remove(toRemovePair));
        //delete toRemovePair;
        if (toFind == nullptr) {
            return nullptr;
        }
        T* value = toFind->extract();
        delete toFind;
        --this->amount_of_items;
        if (checkForUpdateSize) {
            this->checkUpdateArr();
        }
        return value;
    }

    /**
     *the same functionality as regular remove, but without
     *checking for size updates, for inside methods and usages
     *
     * @param key
     * @return
     */
    T* removeImmediate(int key) {
        return this->remove(key, false);
    }


    void initializeArray() {
        //delete[] this->data_arr;
        this->data_arr = new DeQue<Pair<T>>[this->arr_size];
        for (int i = 0; i < this->arr_size; ++i) {
            //this->data_arr[i] = *(new DeQue<Pair<T>>());
            DeQue<Pair<T>>& desiredSlot = this->data_arr[i];
            desiredSlot.verifyInitialisation();
        }
    }

    ChainHashArray(int size)  : capacity(EMPTY) {
        this->data_arr = nullptr;
        this->arr_size = MAX(INITIAL_SIZE_CHAIN_HASH_ARRAY,size);
        this->amount_of_items = EMPTY;
        this->updateCapacity();
        this->initializeArray();
    }

    void updateCapacity() {
        capacity = this->arr_size * MAX_FILL_RATIO_CHAIN_HASH_ARRAY;
    }

    int calcIndex(int key) {
        int sheerit = key % this->arr_size;
        sheerit += this->arr_size;
        int positive_sheerit = sheerit % this->arr_size;
        return positive_sheerit;
    }

    void checkUpdateArr() {
        if (this->amount_of_items == this->capacity) {
            this->makeBigger();
            return;
        }
        if (this->amount_of_items <= QUARTER_OF(this->capacity)) {
            this->makeSmaller();
        }
    }

    void resize(int new_capacity) {
        ChainHashArray<T>* other = new ChainHashArray(new_capacity);
        for (int i = 0; i < this->arr_size; i++) {
            DeQue<Pair<T>>& temp = (this->data_arr)[i];
            int tempSize = temp.getSize();
            for (int j = 0; j < tempSize; j++) {
                Pair<T>* tempItem = temp.pop();
                other->insertImmediate(tempItem->key, tempItem->extract());
                delete tempItem;
            }
        }
        this->swapData(other);
        delete other;
```

```cpp
    }

    void makeBigger() {
        this->resize(MAKE_DOUBLE(this->arr_size));
    }

    void makeSmaller() {
        if(this->arr_size <= INITIAL_SIZE_CHAIN_HASH_ARRAY) {
            return;
        }
        this->resize(HALF_OF(this->arr_size));
    }

    template <typename K>
    void swap(K& item_1, K& item_2) {
        K temp = item_1;
        item_1 = item_2;
        item_2 = temp;
    }

    void swapData(ChainHashArray* other) {
        assert(other != nullptr);
        swap<int>(other->arr_size, this->arr_size);
        swap<int>(other->amount_of_items, this->amount_of_items);
        swap<int>(other->capacity, this->capacity);
        swap<DeQue<Pair<T>>*>(other->data_arr, this->data_arr);
    }





};



#endif //CHAINHASHARRAY_H

```

DeQue.h

Download

```
//
// Created by Guy Friedman on 24/01/2025.
//

#ifndef DEQUE_H
#define DEQUE_H

#include "DeQueNode.h"
#define EMPTY (0)



template <typename T>
class DeQue {
protected:
    DeQueNode<T>* head;
    DeQueNode<T>* tail;
    int size;
public:

    DeQue() : head(new DeQueNode<T>()), tail(new DeQueNode<T>()), size(0) {
        //head->addInitial(this->tail);
        this->head->next = this->tail;
        this->tail->prev = this->head;
    }

    ~DeQue() {
        delete head;
    }

    int getSize() const {
        return size;
    }

    void append(T* item) {
        auto newNode = new DeQueNode<T>(item);
        this->tail->queueAdd(newNode);
        ++this->size;
    }

    void insert(T* item) {
        auto newNode = new DeQueNode<T>(item);
        this->head->stackAdd(newNode);
        ++this->size;
    }

    T* pop() {
        assert(this->head->hasNext());
        auto newNode = this->head->popNext();
        T* tempVal = newNode->getData();
        newNode->nullify();
        delete newNode;
        --this->size;
        return tempVal;
    }

    T* find(T& value) {
        auto node = this->head->find(value);
        return (node == nullptr)?nullptr:node->getData();
    }

    T* remove(T& item) {
        DeQueNode<T>* temp = this->head->remove(item);
        if (temp == nullptr) {
            return nullptr;
        }
        T* newTemp = temp->extract();
        temp->verifyDeCouple();
        temp->nullify();
        delete temp;
        --this->size;
        return newTemp;
    }

    void verifyInitialisation() {
        if(this->size != EMPTY) {
            return;
        }
        this->head->next = this->tail;
        this->tail->prev = this->head;
    }





};



#endif //DEQUE_H

```

DeQueNode.h Download

```
//
// Created by Guy Friedman on 24/01/2025.
//

#ifndef DEQUENODE_H
#define DEQUENODE_H


#include <cassert>




template <typename T>
class DeQue;

template <typename T>
class DeQueNode {
private:

    // Declare the templated DeQue as a friend
    template <typename U>
    friend class DeQue;

    inline bool initialNode() const {
        return (this->next == nullptr) && (this->prev == nullptr);
    }

    inline bool noData() const {
        return this->data == nullptr;
    }

    inline bool canBecomeHead() {
        return (this->initialNode()) && (this->noData());
    }

    inline bool canBecomeTail() {
        return (this->initialNode()) && (this->noData());
    }

protected:

    T* data;
    DeQueNode<T>* next;
    DeQueNode<T>* prev;

public:

    DeQueNode() : data(nullptr), next(nullptr), prev(nullptr) {}

    DeQueNode(T* data) : data(data), next(nullptr), prev(nullptr) {}

    ~DeQueNode() {
        delete next;
        delete data;
    }

    void nullify() {
        this->next = nullptr;
        this->prev = nullptr;
        this->data = nullptr;
    }

    void deCouple() {
        this->next->prev = this->prev;
        this->prev->next = this->next;
        this->next = nullptr;
        this->prev = nullptr;
    }

    void verifyDeCouple() {
        if (this->next == nullptr && this->prev == nullptr) {
            return;
        }
        this->deCouple();
    }

    T* extractAndDelete() {
        this->deCouple();
        T* temp = this->data;
        this->data = nullptr;
        this->nullify();
        //delete this;
        return temp;
    }

    T* extract() {
        T* temp = this->data;
        this->data = nullptr;
        return temp;
    }


    inline bool hasNext() const { //fixme problem
        return !((this->next != nullptr) && (this->next->isTail()));
    }

    inline bool isTail() const {
        return (this->prev != nullptr) && (this->data == nullptr) && (this->next == nullptr);
    }

    inline bool isHead() const {
        return (this->next != nullptr) && (this->data == nullptr) && (this->prev == nullptr);
    }

    DeQueNode<T>* popNext() {
        assert(this->hasNext());
        assert(this->next != nullptr && this->next->next != nullptr);
        DeQueNode<T>* temp = this->next;
        temp->deCouple();
        return temp;
    }

    void queueAdd(DeQueNode<T>* node) {
        assert(node != nullptr);
        assert(!this->isHead());
        this->prev->next = node;
        node->prev = this->prev;
        node->next = this;
        this->prev = node;
    }

    void stackAdd(DeQueNode<T>* node) {
        assert(node != nullptr);
        assert(!this->isTail());
        this->next->prev = node;
        node->next = this->next;
        node->prev = this;
        this->next = node;
    }

    void addInitial(DeQueNode<T>* node) {
        assert(node != nullptr);
        //make sure that 'this' node is in a state required to become head
        assert(this->canBecomeHead());
        //make sure that 'node' node is in a state required to become tail
        assert(node->canBecomeTail());
        this->next = node;
        node->prev = this;
    }

    T* getData() const {
        return this->data;
    }

    DeQueNode<T>* find(T& toFind) {
        if(this->isTail()) {
            return nullptr;
        }
        if (this->isHead()) {
            return this->next->find(toFind);
        }
        if (*(this->data) == toFind) {
            return this;
        }
        return this->next->find(toFind);
    }

    DeQueNode<T>* remove(T& toFind) {
        if(this->isTail()) {
            return nullptr;
        }
        if (this->isHead()) {
            return this->next->remove(toFind);
        }
        if (*(this->data) == toFind) {
            this->deCouple();
            return this;
        }
        return this->next->find(toFind);
    }

    //int getKey() {return this->key;}




};



#endif //DEQUENODE_H

```

▾ Pair.h Download

```cpp
//
// Created by Guy Friedman on 24/01/2025.
//

#ifndef PAIR_H
#define PAIR_H


constexpr int DEFAULT_KEY = 0;

template <typename T>
class Pair {
public:
    int key;
    T* value;

    Pair(int key, T* value) : Pair(key) {
        this->value = value;
    }

    Pair(int key) : Pair() {
        this->key = key;
    }

    Pair() : key(DEFAULT_KEY), value(nullptr) {}

    ~Pair() {
        delete this->value;
    }

    T* extract() {
        auto value = this->value;
        this->value = nullptr;
        return value;
    }

    void nullify() {
        this->key = DEFAULT_KEY;
        this->value = nullptr;
    }


    bool operator ==(const Pair& other) const {
        return this->key == other.key;
    }

    // Overloaded operator== as a member function
    //bool operator==(int otherKey) const {return this->key == otherKey;}

    // Friend operator== (int == Pair<int>)
    friend bool operator==(int lhs, const Pair& rhs) {
        return rhs == lhs;
    }




};



#endif //PAIR_H

```

▾ plains25a2.cpp Download

```
// You can edit anything you want in this file.
// However you need to implement all public Plains function, as provided below as a template

#include "plains25a2.h"
#include "wet2util.h"
#include "ChainHashArray.h"
#include <new>
#include "Jockey.h"
#include "NewTeamArr.h"
#include "Record.h"
#include "RecordArr.h"
#include "Jockey.h"

#define MIN(a,b) (a<b)?a:b

#define ABS(a) ((a)<0?(-(a)):(a))

Plains::Plains()
{
    /*
    RecordArr* records;
    NewTeamArr* teams;
    ChainHashArray<Jockey>* jockeys;
     */
    this->records = new RecordArr();
    this->teams = new NewTeamArr();
    this->jockeys = new ChainHashArray<Jockey>();

}

Plains::~Plains()
{
    delete records;
    delete teams;
    delete jockeys;
}

StatusType Plains::add_team(int teamId)
{
    try{
        if(teamId<=0) {
            return StatusType::INVALID_INPUT;
        }

        //check if team already exists (don't care if active or not)
        NewTeam* newTeam = this->teams->find(teamId);
        if (newTeam != nullptr) {
            return StatusType::FAILURE;
        }
```

```cpp

        //after we checked that there is no and was no team for 'teamId', create it.
        newTeam = new NewTeam(teamId);

        //put team in teams
        this->teams->insert(teamId, newTeam);

        //the record for every beginner team
        int initial_record = 0;

        //add team to corresponding record
        this->records->add_team_to_record(newTeam,teamId,initial_record);

        return StatusType::SUCCESS;
    } catch (std::bad_alloc& e) {
        return StatusType::ALLOCATION_ERROR;
    }
}


StatusType Plains::add_jockey(int jockeyId, int teamId)
{
    try{
        if(jockeyId<=0 || teamId <= 0) {
            return StatusType::INVALID_INPUT;
        }

        //make sure team is valid, without path modification
        if(!this->teams->check_active_immediate(teamId)) {
            return StatusType::FAILURE;
        }

        //make sure jockey doesn't already exist
        Jockey* jockey = this->jockeys->find(jockeyId);
        if(jockey != nullptr) {
            return StatusType::FAILURE;
        }

        //create jockey
        jockey = new Jockey(jockeyId,teamId);

        //store jockey in our system
        this->jockeys->insert(jockeyId,jockey);

        //todo increment size of team by 1

        return StatusType::SUCCESS;
    } catch (std::bad_alloc& e) {
        return StatusType::ALLOCATION_ERROR;
    }
}

```

```

StatusType Plains::update_match(int victoriousJockeyId, int losingJockeyId)
{
    try{
        //check input validity
        if(victoriousJockeyId<=0 || losingJockeyId <= 0||victoriousJockeyId == losingJockeyId) {
            return StatusType::INVALID_INPUT;
        }

        //@brief - check jockeys exist, check jockeys are not in the same team, remove teams from their records
        //@brief - update score for jockeys and teams, put teams in their new records

        //check that winning jockey exists
        Jockey* winningJockey = this->jockeys->find(victoriousJockeyId);
        if (winningJockey == nullptr) {
            return StatusType::FAILURE;
        }

        //check that losing jockey exists
        Jockey* losingJockey = this->jockeys->find(losingJockeyId);
        if (losingJockey == nullptr) {
            return StatusType::FAILURE;
        }

        //get team of winning jockey
        NewTeam* winningTeam = this->teams->get_root_of(winningJockey->getTeamId());

        //get team of losing jockey
        NewTeam* losingTeam = this->teams->get_root_of(losingJockey->getTeamId());

        //make sure it is not the same group
        if (winningTeam==losingTeam) {
            return StatusType::FAILURE;
        }

        //make sure no funny business is going on
        assert(winningTeam != nullptr);
        assert(losingTeam != nullptr);

        //remove each team from their record
        this->records->remove_team_from_record(winningTeam->get_id(),winningTeam->get_record());
        this->records->remove_team_from_record(losingTeam->get_id(),losingTeam->get_record());

        //update scores
        winningJockey->winMatch();
        losingJockey->loseMatch();
        winningTeam->winMatch();
        losingTeam->loseMatch();

        //store teams in their new records
        this->records->add_team_to_record(winningTeam,winningTeam->get_id(),winningTeam->get_record());
        this->records->add_team_to_record(losingTeam,losingTeam->get_id(),losingTeam->get_record());

        return StatusType::SUCCESS;
    } catch (std::bad_alloc& e) {
        return StatusType::ALLOCATION_ERROR;
    }
}


StatusType Plains::merge_teams(int teamId1, int teamId2)
{
    try{
        if(teamId1<=0||teamId2<=0||teamId1==teamId2) {
            return StatusType::INVALID_INPUT;
        }

        //make sure team1 is active
        if (!this->teams->team_active(teamId1)) {
            return StatusType::FAILURE;
        }

        //make sure team2 is active
        if (!this->teams->team_active(teamId2)) {
            return StatusType::FAILURE;
        }

        //retrieve team1 and team2
        NewTeam* teamOne = this->teams->get_root_of(teamId1);
        NewTeam* teamTwo = this->teams->get_root_of(teamId2);

        //make sure no funny business is happening
        assert(teamOne->get_id() == teamId1);
        assert(teamTwo->get_id() == teamId2);

        //if they are the same team - cant merge
        if (teamOne == teamTwo) {
            return StatusType::FAILURE;
        }

        //remove each team from their record
        this->records->remove_team_from_record(teamId1,teamOne->get_record());
        this->records->remove_team_from_record(teamId2,teamTwo->get_record());

        //use logic in TeamArr to handle merging of teams
        this->teams->unite_teams(teamId1,teamId2);

        //make sure all is going good and no funny business, should be ok if merge went ok in TeamArr.
        assert(teamOne->get_id() == teamTwo->get_id());

        //now ID will be updated in both teams, get the new root (would automatically select either teamOne or teamTwo, according to the merge)
        NewTeam* newRoot = this->teams->get_root_of(teamOne->get_id());

        //store merged team in the appropriate record
        this->records->add_team_to_record(newRoot, newRoot->get_id(), newRoot->get_record());

        return StatusType::SUCCESS;
    } catch(std::bad_alloc& e){
        return StatusType::ALLOCATION_ERROR;
    }
}


StatusType Plains::unite_by_record(int record)
{
    try{
        if (record<=0) {
            return StatusType::INVALID_INPUT;
        }

        //make sure that we can unite by record
        if (!this->records->can_unite_by_record(record)) {
            return StatusType::FAILURE;
        }

        //get id of teams in singleton records
        int positive_team_id = this->records->return_team_id_of_singleton_record(record);
        int negative_team_id = this->records->return_team_id_of_singleton_record(-record);

        //utilise existing logic and code to handle merging execution
        return this->merge_teams(positive_team_id,negative_team_id);

    } catch(std::bad_alloc& e){
        return StatusType::ALLOCATION_ERROR;
    }
}


output_t<int> Plains::get_jockey_record(int jockeyId)
{
    try{
        if (jockeyId<=0) {
            return StatusType::INVALID_INPUT;
        }
        Jockey* jock = this->jockeys->find(jockeyId);
        if (jock==nullptr) {
            return StatusType::FAILURE;
        }
        return jock->getRecord();
    } catch(std::bad_alloc& e) {
        return StatusType::ALLOCATION_ERROR;
```

```cpp
    }
}


output_t<int> Plains::get_team_record(int teamId)
{
    try{
        if (teamId<=0) {
            return StatusType::INVALID_INPUT;
        }

        //make sure team is active without path modifications
        if (!this->teams->team_active(teamId)) { //check_active_immediate
            return StatusType::FAILURE;
        }

        //get root of team
        NewTeam* newTeam = this->teams->get_root_of(teamId);

        //make sure no funny business is going on
        assert(newTeam->get_id() == teamId);

        //the record of our team
        int record = newTeam->get_record();

        //return it
        return record;

    } catch(std::bad_alloc& e) {
        return StatusType::ALLOCATION_ERROR;
    }
}

```

plains25a2.h

Download

```
//
// 234218 Data Structures 1.
// Semester: 2025A (Winter).
// Wet Exercise #1.
//
// The following header file contains all methods we expect you to implement.
// You MAY add private methods and fields of your own.
// DO NOT erase or modify the signatures of the public methods.
// DO NOT modify the preprocessors in this file.
// DO NOT use the preprocessors in your other code files.
//

#ifndef PLAINS25WINTER_WET1_H_
#define PLAINS25WINTER_WET1_H_

#include "wet2util.h"
#include "ChainHashArray.h"
#include <new>

#include "Jockey.h"
#include "NewTeamArr.h"
#include "Record.h"
#include "RecordArr.h"

class Plains {
private:
    //
    // Here you may add anything you want
    //
    RecordArr* records;
    NewTeamArr* teams;
    ChainHashArray<Jockey>* jockeys;

public:
    // <DO-NOT-MODIFY> {
    Plains();

    ~Plains();

    StatusType add_team(int teamId);

    StatusType add_jockey(int jockeyId, int teamId);

    StatusType update_match(int victoriousJockeyId, int losingJockeyId);

    StatusType merge_teams(int teamId1, int teamId2);

    StatusType unite_by_record(int record);

    output_t<int> get_jockey_record(int jockeyId);

    output_t<int> get_team_record(int teamId);
    // } </DO-NOT-MODIFY>
};

#endif // PLAINS25WINTER_WET1_H_

```

▾ **Record.h** Download

```cpp
//
// Created by Guy Friedman on 26/01/2025.
//


#ifndef RECORD_H
#define RECORD_H
#include "ChainHashArray.h"
#include "NewTeam.h"

typedef NewTeam what_to_hold;

class Record {
protected:
    ChainHashArray<what_to_hold>* held_items;
    int record_value;
public:
    Record() = delete;

    Record(int record_value) : record_value(record_value) {
        this->held_items = new ChainHashArray<what_to_hold>();
    }

    ~Record() {
        this->held_items->clear();
        delete this->held_items;
    }

    bool isEmpty() {
        return this->held_items->size() == 0;
    }

    bool isSingleton() {
        return this->held_items->size() == 1;
    }

    what_to_hold* pop() {
        return this->held_items->popRandom();
    }

    what_to_hold* remove(int key) {
        return this->held_items->remove(key);
    }

    void insert(int key, what_to_hold* value) {
        this->held_items->insert(key, value);
    }

    int get_records_val() {
        return this->record_value;
    }

    int get_singleton_team_id() { //use is depreceated, already implemented in RecordArr
        //todo - pop only team (assert this is singleton), save team id, re insert team with team id, return team id

        assert(this->isSingleton());

        //fixme
        return 0;
    }



};



#endif //RECORD_H

```