Wet HW 1 (wet part)

● Graded

Group

גיא פרידמן
טל כרמל
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

ProjectFiles.h Download

```
1  #pragma once
2  #include "wet1util.h"
3  //#include "AbstractAVL.h"
4  //#include "IndexAble.h"
5  //#include "NonOwningHorseAVL.h"
6  //#include "NonOwningHerdAVL.h"
7  //#include "OwningHorseAVL.h"
8  //#include "OwningHerdAVL.h"
9  #include "Horse.h"
10 #include "Herd.h"
11 //#include "HorseMap.h"
12 //#include "HorseTree.h"
13 //#include "TraceAble.h"
14 #include "AVLTree.h"
15 //#include "LinkedList.h"
16 #include <cassert>
17 #include <new>
18 #include <iostream>
19 //#include "Pair.h"
20 //#include "HorseList.h"
```

plains25a1.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 "wet1util.h"


//#pragma once
#include "ProjectFiles.h"
#include <cassert>


class Plains {
private:
    //
    // Here you may add anything you want
    //
    AVLTree<Horse> allHorses;
    AVLTree<Herd> nonEmptyHerds;
    AVLTree<Herd> emptyHerds;


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

    ~Plains();

    StatusType add_herd(int herdId);

    StatusType remove_herd(int herdId);

    StatusType add_horse(int horseId, int speed);

    StatusType join_herd(int horseId, int herdId);

    StatusType follow(int horseId, int horseToFollowId);

    StatusType leave_herd(int horseId);
```

```
50
51      output_t<int> get_speed(int horseId);
52
53      output_t<bool> leads(int horseId, int otherHorseId);
54
55      output_t<bool> can_run_together(int herdId);
56      // } </DO-NOT-MODIFY>
57  };
58
59  #endif // PLAINS25WINTER_WET1_H_
60
```

plains25a1.cpp

Download

```cpp
// 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 "plains25a1.h"
#include  "ProjectFiles.h"
#define NOT_NULL(a) (a>0)
#include <new>
using std::bad_alloc;

Plains::Plains() : allHorses(), nonEmptyHerds(), emptyHerds() {}
//Plains::Plains() {
//    try {
//        AVL<Herd> allHorses;
//        AVL<Herd> nonEmptyHerds;
//        AVL<Herd> emptyHerds;
//    } catch (...) {
//        delete allHorses;
//        delete nonEmptyHerds;
//        delete emptyHerds;
//        throw;
//    }
//}

Plains::~Plains() = default;
//Plains::~Plains()
//{
//    // Assumes the AVLTree destructor handles its nodes and their values.
//    delete allHorses;
//    delete nonEmptyHerds;
//    delete emptyHerds;
//}

StatusType Plains::add_herd(int herdId)
{
    try {
    if (herdId <= 0){
        return StatusType::INVALID_INPUT;
    }
    if ((nonEmptyHerds.get(herdId) != nullptr)||(emptyHerds.get(herdId) != nullptr)){
        return StatusType::FAILURE;
    }
    Herd* newHerd = nullptr;
    try {
        newHerd = new Herd(herdId);
    } catch (...) {
        return StatusType::ALLOCATION_ERROR;
    }
    bool insertionWorked = emptyHerds.insert(newHerd, herdId);
    return (insertionWorked)?(StatusType::SUCCESS):(StatusType::FAILURE);
```

```
50      } catch (StatusType e ){
51          return StatusType::ALLOCATION_ERROR;
52      }
53  }
54
55  StatusType Plains::remove_herd(int herdId)
56  {
57      bool operationWorked = emptyHerds.remove(herdId);
58      return (operationWorked)?(StatusType::SUCCESS):(StatusType::FAILURE);
59  }
60
61  StatusType Plains::add_horse(int horseId, int speed)
62  {
63      try {
64      if (horseId <= 0){
65          return StatusType::INVALID_INPUT;
66      }
67      if (speed <= 0){
68          return StatusType::INVALID_INPUT;
69      }
70      Horse* newHorse = allHorses.get(horseId);
71      if (newHorse != nullptr){
72          return StatusType::FAILURE;
73      }
74      try {
75          newHorse = new Horse(horseId,speed);
76      } catch (...) {
77          return StatusType::ALLOCATION_ERROR;
78      }
79      bool operationWorked = allHorses.insert(newHorse,horseId);
80      return (operationWorked)?(StatusType::SUCCESS):(StatusType::FAILURE);
81      } catch (StatusType e){
82          return StatusType::ALLOCATION_ERROR;
83      }
84  }
85
86  StatusType Plains::join_herd(int horseId, int herdId) //FIXME ADD PAIR
87  {
88      try {
89      if ((horseId <= 0)||(herdId <= 0)){return StatusType::INVALID_INPUT;}
90
91      Horse* sooson = allHorses.get(horseId);
92
93      if (sooson == nullptr){return StatusType::FAILURE;}
94      if (sooson->getHerdID() > 0){return StatusType::FAILURE;}
95
96      Herd* eder = emptyHerds.get(herdId);
97      if (eder != nullptr){
98          emptyHerds.remove(herdId);
99          try {
100             eder = new Herd(herdId);
101         } catch (...){
```

```
102             return StatusType::ALLOCATION_ERROR;
103         }
104         eder->add_horse(sooson); //FIXME add pair changes
105         nonEmptyHerds.insert(eder,herdId);
106         return StatusType::SUCCESS;
107     }
108     if (eder == nullptr){
109         eder = nonEmptyHerds.get(herdId);
110     }
111     if (eder == nullptr){return StatusType::FAILURE;}
112     eder->add_horse(sooson); //TODO modify for Pair changes
113     return StatusType::SUCCESS;
114     } catch (StatusType e){
115         return StatusType::ALLOCATION_ERROR;
116     }
117 }
118
119 StatusType Plains::follow(int horseId, int horseToFollowId)
120 {
121     try {
122     if ((horseId == horseToFollowId)||(horseId <= 0)||(horseToFollowId <= 0)){
123         return StatusType::INVALID_INPUT;
124     }
125     Horse* firstHorse = allHorses.get(horseId);
126     if (firstHorse == nullptr){
127         return StatusType::FAILURE;
128     }
129     Horse* secondHorse = allHorses.get(horseToFollowId);
130     if (secondHorse == nullptr){
131         return StatusType::FAILURE;
132     }
133     bool operationWorked = firstHorse->follow(secondHorse);
134     return (operationWorked)?(StatusType::SUCCESS):(StatusType::FAILURE);
135     } catch (StatusType e){
136         return StatusType::ALLOCATION_ERROR;
137     }
138 }
139
140 StatusType Plains::leave_herd(int horseId)
141 {
142     try {
143     if (horseId <= 0){
144         return StatusType::INVALID_INPUT;
145     }
146     Horse* horseToLeave = allHorses.get(horseId);
147     if (horseToLeave == nullptr || horseToLeave->getHerdID() <= 0){
148         return StatusType::FAILURE;
149     }
150         Herd* herd = horseToLeave->getHerd();
151         if (herd == nullptr) {
152             return StatusType::FAILURE;
153         }
```

```
154     bool succes = horseToLeave->leaveHerd();
155
156         if (herd != nullptr && herd->isEmpty()) {
157             int oldid = herd->getID();
158             nonEmptyHerds.remove(oldid);
159             Herd* newHerd = nullptr;
160             try {
161                 newHerd = new Herd(oldid);
162                 if (newHerd == nullptr) {
163                     return StatusType::ALLOCATION_ERROR;
164                 }
165             } catch (...) {
166                 return StatusType::ALLOCATION_ERROR;
167             }
168             emptyHerds.insert(newHerd,oldid);
169         }
170     return (succes)?(StatusType::SUCCESS):(StatusType::FAILURE);
171     } catch (StatusType e){
172         return StatusType::ALLOCATION_ERROR;
173     }
174 }
175
176 output_t<int> Plains::get_speed(int horseId)
177 {
178     try {
179     if (horseId <= 0){
180         return StatusType::INVALID_INPUT;
181     }
182     Horse* horseToSpeed = allHorses.get(horseId);
183     if (horseToSpeed == nullptr){
184         return StatusType::FAILURE;
185     }
186     int speed = horseToSpeed->getSpeed();
187     return speed;
188     } catch (StatusType e){
189         return StatusType::ALLOCATION_ERROR;
190     }
191 }
192
193 output_t<bool> Plains::leads(int horseId, int otherHorseId)
194 {
195     try {
196     if ((horseId <= 0)||(otherHorseId <= 0)||(horseId==otherHorseId)){
197         return StatusType::INVALID_INPUT;
198     }
199     Horse* firstHorse = allHorses.get(horseId);
200     if (firstHorse == nullptr){
201         return StatusType::FAILURE;
202     }
203     Horse* secondHorse = allHorses.get(otherHorseId);
204     if (secondHorse == nullptr){
205         return StatusType::FAILURE;
```

```
206     }
207     //if ((firstHorse->getHerd() == nullptr)||(secondHorse->getHerd()==nullptr)) {return
    StatusType::FAILURE;}
208     int HerdIDfirst = firstHorse->getHerdID();
209     int HerdIDsecond = secondHorse->getHerdID();
210     if (HerdIDfirst != HerdIDsecond){
211         return false;
212     }
213     Herd* herd = firstHorse->getHerd();
214     if (herd == nullptr){return false;}
215     return herd->leads(horseId, otherHorseId);
216     } catch (StatusType e){
217         return StatusType::ALLOCATION_ERROR;
218     }
219 }
220
221 output_t<bool> Plains::can_run_together(int herdId)
222 {
223     try {
224     if (herdId <= 0){
225         return StatusType::INVALID_INPUT;
226     }
227     Herd* herd = nonEmptyHerds.get(herdId);
228     if (herd == nullptr){
229         return StatusType::FAILURE;
230     }
231     return herd->can_run_together();
232     } catch (StatusType e){
233         return StatusType::ALLOCATION_ERROR;
234     }
235 }
236
```

AVLTree.h

Download

```
#pragma once
#include <cassert>
#include "AVLNode.h"
#include <new>
#define NULL_ID (-1)


template <typename Value>
class AVLTree {
protected:
    AVLNode<Value>* head;

public:
    AVLTree(Value* value, int id):head(AVLNode<Value>(value,id)){}
    AVLTree():head(nullptr){}
    virtual ~AVLTree(){
        delete head;
        head = nullptr;
    }

    bool insert(Value* value, int id){
        if (this->head == nullptr){
            this->head = new (std::nothrow) AVLNode<Value>(value,id);
            if (!this->head){throw StatusType::ALLOCATION_ERROR;}
            return true;
        }
        Value* exists = this->head->find(id);
        if (exists != nullptr){
            return false;
        }
        this->head = this->head->insert(value,id);
        assert(this->verifyTree());
        return true;
    }

    bool remove(int index){
        if (this->head == nullptr){
            return false;
        }
        Value* exists = this->head->find(index);
        if (exists == nullptr){
            return false;
        }
        this->head = this->head->deleteNode(index);
        assert(this->verifyTree());
        return true;
    }

    Value* get(int index){
```

```
50          //assert(this->verifyTree());
51          if (this->head == nullptr){
52              return nullptr;
53          }
54          return this->head->find(index);
55      }
56
57      /**
58       * this function will be used for debugging
59       */
60      bool verifyTree(){
61          if (this->head == nullptr){return true;}
62          bool heightVerified = this->head->heightVerified();
63          bool balanceVerified = this->head->isBalanced();
64          return heightVerified && balanceVerified;
65      }
66
67  };
68
69
```

AVLNode.h Download

```
#pragma once
#include <cassert>
#define NULL_ID (-1)
#include  <iostream>
//#include "AVL.h"
#include "wet1util.h"
#define EMPTY_TREE_HEIGHT -1
#include <new>
using std::cout;

template <typename Value>
class AVLNode {
protected:
    int index;
    Value* value;
    AVLNode<Value>* left;
    AVLNode<Value>* right;
    int height;

public:

    enum class Roll {
        noRoll,
        LL,
        LR,
        RL,
        RR
    };

    AVLNode(Value* value, int id)
        : index(id), value(value), left(nullptr), right(nullptr), height(EMPTY_TREE_HEIGHT+1){}

    virtual ~AVLNode(){
        delete this->left;
        delete this->right;
        delete this->value;
        this->left = nullptr;
        this->right = nullptr;
        this->value = nullptr;

    }

    /**
     * this function will be used for debugging in AVL class
     * for example - assert(heightVerified()) after inserting or deleting
     * values
     */
    bool heightVerified(){
        //if (node == nullptr){return true;}
```

```
50          bool leftTree = true;
51          if (this->left != nullptr) {
52              leftTree = this->left->heightVerified();
53          }
54          bool rightTree = true;
55          if (this->right != nullptr) {
56              rightTree = this->right->heightVerified();
57          }
58          int oldHeight = this->height;
59          int newHeight = this->heightUpdate();
60          bool thisNode = (oldHeight == newHeight);
61          return leftTree&&rightTree&&thisNode;
62      }
63
64      /**
65       * same as above
66       */
67      bool isBalanced(){
68          //no need for bull check, taken care of by AVLTree
69          bool leftTree = true;
70          if (this->left != nullptr) {
71              leftTree = this->left->isBalanced();
72          }
73          bool rightTree = true;
74          if (this->right != nullptr) {
75              rightTree = this->right->isBalanced();
76          }
77          int nodesBalance = this->balanceFactor();
78          bool thisNode = ((-1<=nodesBalance) && (nodesBalance<=1));
79          return leftTree&&rightTree&&thisNode;
80      }
81
82
83  protected:
84      template<typename T>
85      friend class AVL;
86      void insertRight(AVLNode<Value>* node) {
87          if (!this->right) {
88              this->right = node;
89          } else {
90              this->right = this->right->insert(node);
91          }
92      }
93
94      void insertLeft(AVLNode<Value>* node) {
95          if (!this->left) {
96              this->left = node;
97          } else {
98              this->left = this->left->insert(node);
99          }
100     }
101
```

```
102 //    /**
103 //     * this function will be used for debugging in AVL class
104 //     * for example - assert(heightVerified()) after inserting or deleting
105 //     * values
106 //     */
107 //    bool heightVerified(AVLNode<Value>* node){
108 //        if (node == nullptr){
109 //            return true;
110 //        }
111 //        bool leftTree = heightVerified(node->left);
112 //        bool rightTree = heightVerified(node->right);
113 //        int oldHeight = node->height;
114 //        int newHeight = node->heightUpdate();
115 //        bool thisNode = (oldHeight == newHeight);
116 //        return leftTree&&rightTree&&thisNode;
117 //    }
118 //
119 //    /**
120 //     * same as above
121 //     */
122 //    bool isBalanced(AVLNode<Value>* node){
123 //        if (node == nullptr){
124 //            return true;
125 //        }
126 //        bool leftTree = isBalanced(node->left);
127 //        bool rightTree = isBalanced(node->right);
128 //        int nodesBalance = node->balanceFactor();
129 //        bool thisNode = ((-1<=nodesBalance) && (nodesBalance<=1));
130 //        return leftTree&&rightTree&&thisNode;
131 //    }
132
133
134
135     AVLNode<Value>* Balance() {
136         this->heightUpdate(); //make sure height is updated
137         Roll roll = this->getRoll();
138         switch (roll) {
139             case Roll::noRoll:
140                 return this;
141             case Roll::LL:
142                 return this->LL();
143             case Roll::LR:
144                 return this->LR();
145             case Roll::RL:
146                 return this->RL();
147             case Roll::RR:
148                 return this->RR();
149             default:
150                 assert(false);
151                 return this;
152         }
153     }
```

```
154
155     /**
156      * update the height value,
157      * this function is going to be called many times, sometimes seemingly unnecesairly,
158      * but it is done in order to account for inhereting classes overriding some methods and forgetting to
    call this function
159      */
160     int heightUpdate() {
161         int leftHeight = this->left ? this->left->height : EMPTY_TREE_HEIGHT;
162         int rightHeight = this->right ? this->right->height : EMPTY_TREE_HEIGHT;
163         this->height = 1 + ((leftHeight > rightHeight) ? leftHeight : rightHeight);
164         return this->height;
165     }
166
167     int balanceFactor() {
168         this->heightUpdate(); //make sure height is updated
169         int leftHeight = this->left ? this->left->height : EMPTY_TREE_HEIGHT;
170         int rightHeight = this->right ? this->right->height : EMPTY_TREE_HEIGHT;
171         return leftHeight - rightHeight;
172     }
173
174     Roll getRoll() {
175         this->heightUpdate(); //make sure height is updated
176         int balance = this->balanceFactor();
177         if (-1 <= balance && balance <= 1) return Roll::noRoll;
178         if (balance > 1) {
179             return this->left->balanceFactor() >= 0 ? Roll::LL : Roll::LR;
180         }
181         return this->right->balanceFactor() <= 0 ? Roll::RR : Roll::RL;
182     }
183
184     AVLNode<Value>* LL() {
185         AVLNode<Value>* temp = this->left;
186         this->left = temp->right;
187         temp->right = this;
188         this->heightUpdate();
189         temp->heightUpdate();
190         return temp;
191     }
192
193     AVLNode<Value>* LR() {
194         this->left = this->left->RR();
195         return this->LL();
196     }
197
198     AVLNode<Value>* RL() {
199         this->right = this->right->LL();
200         return this->RR();
201     }
202
203     AVLNode<Value>* RR() {
204         AVLNode<Value>* temp = this->right;
```

```
205         this->right = temp->left;
206         temp->left = this;
207         this->heightUpdate();
208         temp->heightUpdate();
209         return temp;
210     }
211 
212     inline bool isLeaf(){
213         return (this->left == nullptr) && (this->right == nullptr);
214     }
215 
216     inline bool oneChild(){
217         return (this->left == nullptr) ^ (this->right == nullptr);
218     }
219 
220     inline bool twoChildern(){
221         return (this->left != nullptr) && (this->right != nullptr);
222     }
223 
224     /**
225      * absorb a given node into 'this', effectively 'deleting' 'this'.
226      */
227     void absorbNode(AVLNode<Value>* nodeToAbsorb){
228         //assert(!(this->isLeaf()));
229         this->index = nodeToAbsorb->index;
230         Value* temp = this->value;
231         this->value = nodeToAbsorb->value;
232         nodeToAbsorb->value = temp;
233         delete nodeToAbsorb;
234         this->heightUpdate(); //extra call
235     }
236 
237     /**
238      * absorb the child of the node.
239      */
240     void absorbChild(){
241         assert(this->oneChild());
242         assert(this->left == nullptr || this->left->isLeaf());
243         assert(this->right == nullptr || this->right->isLeaf());
244         AVLNode<Value>* child = nullptr;
245         if (this->left != nullptr){
246             child = this->left;
247         } else {
248             child = this->right;
249         }
250         this->left = this->right = nullptr;
251         this->absorbNode(child);
252     }
253 
254 
255 
256     /**
```

```
257     * replace the value of 'this' with its succesor in the in-order order.
258     */
259    void replaceWithSuccssessor(){
260        assert(this->twoChildern());
261        AVLNode<Value>* succssesor = nullptr;
262        if (this->right->left == nullptr){
263            succssesor = this->right;
264            this->right = succssesor->right;
265            succssesor->right = nullptr;
266        } else {
267        succssesor = this->right->getSmallest();
268        }
269        //int succssesorIndex = succssesor->index;
270        this->absorbNode(succssesor);
271        if (this->right != nullptr) {
272            this->right = this->right->updateLeftPath();
273        }
274        this->heightUpdate(); //extra call
275    }
276
277    /**
278     * get the value with the smallest index of a given tree,
279     * notice this function leaves the tree unorganized, calling functions must organize afterwards
280     * using the updatePath function
281     *
282     * @return - the value with the smallest index of a given tree
283     */
284    AVLNode<Value>* getSmallest() {
285        assert(this->left != nullptr);
286        if(this->left->left != nullptr){
287            return this->left->getSmallest();
288        }
289        AVLNode<Value>* temp = this->left;
290        this->left = temp->right;
291        temp->right = nullptr;
292        assert(temp->left == nullptr);
293        return temp;
294    }
295
296    /**
297     * update path along an index.
298     *
299     * @return - the head of the balanced sub tree
300     */
301    AVLNode<Value>* updatePath(int index){ //function takes O(n) time!
302        assert(false); //this function should not be called
303        int thisIndex = this->index;
304        int fixIndex = index;
305        assert(thisIndex != fixIndex);
306        if (fixIndex < thisIndex){
307            this->left = (this->left == nullptr)?  nullptr : this->left->updatePath(fixIndex);
308        } else {
```

```
309             this->right = (this->right == nullptr)?  nullptr : this->right->updatePath(fixIndex);
310         }
311         this->heightUpdate();
312         return this->Balance();
313     }
314 
315         /**
316      * update path along an index.
317      *
318      * @return - the head of the balanced sub tree
319      */
320     AVLNode<Value>* updateLeftPath(){
321         if (this->left == nullptr){
322             return this->Balance();
323         }
324         this->left = this->left->updateLeftPath();
325         return this->Balance();
326     }
327 
328     AVLNode<Value>* deleteThis() { //return the sub tree of 'this' without the node of 'this'.
329         delete this->value;
330         this->value = nullptr;
331         if (this->isLeaf()) {
332             delete this;
333             return nullptr;
334         }
335         if (this->oneChild()){
336             this->absorbChild();
337         }
338         if (this->twoChildern()){
339             this->replaceWithSuccssessor();
340         }
341         this->heightUpdate();
342         return this->Balance();
343     }
344 
345 
346 public:
347 
348     /**
349      * wrapping function to overload insert, same functionality
350      * as inserting a an existing node / sub tree.
351      * provides extra functionality.
352      *
353      * @param value - value of the node to be inserted.
354      * @return - the root of the balanced tree after the addition of the new value.
355      */
356     AVLNode<Value>* insert(Value* value, int id) { //removed const, we are deleting the value after
    runtime
357         AVLNode<Value>* insertThis = new (std::nothrow) AVLNode<Value>(value,id);
358         if (!insertThis){throw StatusType::ALLOCATION_ERROR;}
359         return this->insert(insertThis);
```

```
360     }
361
362     /**
363      * insert a new node into the tree of 'this', return the balanced tree.
364      *
365      * @param node - the node to be inserted into the tree of 'this'
366      * @return - the balanced tree of 'this' after insertion
367      */
368     AVLNode<Value>* insert(AVLNode<Value>* node) {
369         int nodeIndex = node->index;
370         int thisIndex = this->index;
371
372         if (nodeIndex == thisIndex) {
373             cout << "inserted node with same index as existing node";
374             delete node;
375             return this;
376         }
377
378         if (nodeIndex < thisIndex) {
379             this->insertLeft(node);
380         } else {
381             this->insertRight(node);
382         }
383         this->heightUpdate();
384         return this->Balance();
385     }
386
387     /**
388      * wrapping function to overload deleteNode, provide extra functionality in case
389      * a reference for the value to be deleted already exists.
390      *
391      * @param value - value to be deleted/ removed from the tree.
392      * @return - the balanced tree without the removed value.
393      */
394     /**
395      *
396     AVLNode<Value>* deleteNode(Value* value) {
397         int toDelete = value.getId();
398         return this->deleteNode(toDelete);
399     }
400      *
401      */
402
403
404     /**
405      * remove a node from the tree of 'this', return the balanced tree, with returnVal updated at
406      * the head of the tree to contain the appropriate value.
407      *
408      * value may or may not be deleted as well, subject to the definition of 'deleteValue()'.
409      *
410      * @param index - the index of the node to be removed.
411      * @return - the balanced tree after the removal of said node.
```

```
412      */
413     AVLNode<Value>* deleteNode(int index) {
414         if (index == this->index) {
415             return this->deleteThis();
416         }
417 
418         if (index < this->index) {
419             if (!this->left) {
420                 return this;
421             }
422             this->left = this->left->deleteNode(index);
423         } else {
424             if (!this->right) {
425                 return this;
426             }
427             this->right = this->right->deleteNode(index);
428         }
429         this->heightUpdate();
430         return this->Balance();
431     }
432 
433 
434     /**
435      * return a pointer to a value by a given index,
436      * if value is not found (index not in the tree) returned pointer will be null,
437      * however for the case that the value held at that index is supposed to be null,
438      * returnValue handling is encased in a specified function - findNotFound().
439      *
440      * @param index - the index of the value to retrieve its pointer.
441      * @return - a pointer to the value of the corresponding index.
442      */
443     Value* find(int index){
444             if (index == this->index) {
445             return this->value;
446         }
447         Value* searchedValue = nullptr;
448         if (index < this->index) {
449             if (!this->left) {
450                 return nullptr;
451             }
452             searchedValue = this->left->find(index);
453         } else {
454             if (!this->right) {
455                 return nullptr;
456             }
457             searchedValue = this->right->find(index);
458         }
459         return searchedValue;
460     }
461 };
462 
463 
```

Horse.h Download

```
#pragma once
#include "ProjectFiles.h"
//#include "Herd.h"
#include <cassert>
#define NULL_ID (-1)
//#include "HorseList.h"
#include "LinkedList.h"

class Herd; //added as forward declaration due to circular inclusion

class Horse { //: public IndexAble<Horse>, public TraceAble<Horse>{
private:
    friend class HorseList;
    int horseId;
    int speed;
    int herdID;
    int herdInsertions;
    Horse* follows;
    int followsInsertion;
    bool special_bool;
    Herd* herd;
    Node<Horse>* thisLink;
public:
    void setHerd(Herd* herd) {
        this->herd = herd;
    }
    void setLink(Node<Horse>* link) {
        this->thisLink = link;
    }
    int getID() const{
        return this->horseId;
    }
    int getHerdID(){
        return this->herdID;
    }


    Horse(int id, int speed) : horseId(id), speed(speed), herdID(NULL_ID), herdInsertions(0),
                               follows(nullptr), followsInsertion(0), special_bool(false), herd(nullptr),
                                thisLink(nullptr){}
    ~Horse(){
        this->follows = nullptr;
        this->herd = nullptr;
        this->thisLink = nullptr;
    }
    int getSpeed() const{ return this->speed;}
    void setFollow(int horseToFollow);
    void setHerd(int herdId);
// comparison operators overload for use in tree methods.
```

```
50      bool operator==(const Horse& otherHorse) const;
51      bool operator>(const Horse& otherHorse) const;
52      bool operator<(const Horse& otherHorse) const;
53      bool independant(){
54          Horse* followedHorse = this->follows;
55          if (followedHorse == nullptr){return true;}
56          if (!(this->sameHerd(followedHorse))){return true; }
57          int otherHerdInsertions = followedHorse->herdInsertions;
58          //if the horse that 'this' follows has moved a herd since
59          //'this' started following it.
60          if (this->followsInsertion != otherHerdInsertions){return true;}
61          assert((this->followsInsertion == otherHerdInsertions)&&(this->herdID == followedHorse->herdID));
    //FIXME might be a source of a problem here or with leave / join herd
62          return false;
63      }
64
65      inline bool sameHerd(Horse* otherHorse){return this->herdID == otherHorse->herdID;}
66
67      bool follow(Horse* leader){
68          if((this->herd == nullptr)||(!(this->sameHerd(leader)))){ //fixme same non herd horses somehow
    pass this condition
69              return false;
70          }
71          this->follows = leader;
72          this->followsInsertion = leader->herdInsertions;
73          return true;
74      }
75
76      Horse* getFollows();
77      inline bool alreadyChecked(){return this->special_bool;}
78      inline void markChecked(){this->special_bool = true;}
79      inline void unCheck(){this->special_bool = false;}
80
81      /**
82       * NOTICE - THIS FUNCTION DOES NOT PRESERVE special_bool == false,
83       * RESET BOOL **MUST** BE INVOKED ON HERD AFTER THIS FUNCTION IS CALLED
84       */
85      bool inCircularReferance(int jumps){
86          if (jumps < 0){return true;}
87          bool maybeLeader = this->independant();
88          bool checkedNext = (maybeLeader)?true:this->follows->alreadyChecked();
89          if (maybeLeader||checkedNext){
90              this->markChecked();
91              return false;
92          }
93          bool checkNext = this->follows->inCircularReferance(--jumps);
94          this->markChecked();
95          return checkNext;
96      }
97
98      /**
99       * return true if left herd,
```

```
100      * false if was not in a herd
101      */
102     bool leaveHerd();
103 
104     Herd* getHerd() {
105         return this->herd;
106     }
107 
108     void join_herd(Herd* herd);
109 //    void join_herd(Herd* herd){
110 //        this->herdID = herd->getID();
111 //        ++(this->herdInsertions);
112 //    }
113 };
```

Horse.cpp Download

```cpp
#include "ProjectFiles.h"
#include <cassert>

#define NULL_ID (-1)

//Horse::Horse(int id, double speed): horseId(id), speed(speed), following(NULL_ID), herd(NULL_ID), herdInsertions(0), follows(nullptr), special_bool(false) {}

//int Horse::getID() const{return this->horseId;}

//double Horse::getSpeed() const{return this->speed;}
//void Horse::setFollow(int horseToFollow){this->following = horseToFollow;}
//void Horse::setFollow(Horse* horseToFollow){assert(this->herd == horseToFollow->herd);}

//void Horse::setHerd(int herdId){this->herd = herdId;}

bool Horse::operator==(const Horse& otherHorse) const{
    return this->horseId == otherHorse.horseId;
}
bool Horse::operator>(const Horse& otherHorse) const{
    return this->horseId > otherHorse.horseId;
}
bool Horse::operator<(const Horse& otherHorse) const{
    return this->horseId < otherHorse.horseId;
}

void Horse::join_herd(Herd* herd){
    this->herdID = herd->getID();
    this->herd = herd;
    int prev = this->herdInsertions;
    this->herdInsertions = prev + 1; //TODO double increment, because one may not be working properly
    this->herdInsertions++;
}

bool Horse::leaveHerd(){
    if (this->herd == nullptr){
        return false;
    }
    this->herdID = NULL_ID;
    this->herd->leave();
    this->herd = nullptr;
    this->follows = nullptr;
    thisLink->allPointersNull(); //FIXME
    delete thisLink;
    this->thisLink = nullptr;
    ++(this->herdInsertions);
    return true;
}

```

```
49
50  /**
51   *
52  bool Horse::independant(){
53      if (this->follows == nullptr){ return true;}
54      if (this->follows->herd != this->herd){return true;} //FIXME can a horse follow itself?
55      if (true){return true;} //TODO add check that insertion id is correct
56      return false;
57  }
58   */
59
60
```

Herd.h

Download

```
#pragma once
#include "ProjectFiles.h"
#include "LinkedList.h"
#define NULL_ID (-1)

class Herd {
private:
    int herdId;
    LinkedList<Horse> herdMembers;
    int totalMembers;
public:
    explicit Herd(int id):herdId(id), herdMembers(), totalMembers(0){}
    ~Herd() = default;
    int getID() const{
        return this->herdId;
    }
    // comparison operators overload for use in tree methods.
    bool operator==(const Herd& otherHerd) const;
    bool operator>(const Herd& otherHerd) const;
    bool operator<(const Herd& otherHerd) const;

    int get_total_members() {
        return this->totalMembers;
    }
    bool add_horse(Horse* horse){
        assert(addHorseAssertHelper(horse));
        herdMembers.insert(horse);
        ++(this->totalMembers);
        horse->join_herd(this);
        horse->setLink(this->herdMembers.getHead());
        return true;
    }

    bool isEmpty() {
        return this->herdMembers.getHead()->getData() == nullptr;
    }

    bool addHorseAssertHelper(Horse* horse) {
        Node<Horse>* curr = this->herdMembers.getHead();
        Node<Horse>* last = this->herdMembers.getLast();
        while (curr != last) {
            Horse* horsy = curr->getData();
            if (horsy == horse) {
                return false;
            }
            curr = curr->getNext();
        }
        return true;
    }
```

```
50
51      void leave() {
52          --(this->totalMembers);
53      }
54
55      bool leads(int followerID, int leaderID){
56          assert((followerID != leaderID)&&(followerID>0)&&(leaderID>0));
57          Horse* follower = nullptr;
58          Horse* leader = nullptr;
59          Node<Horse>* curr = this->herdMembers.getHead();
60          Node<Horse>* last = this->herdMembers.getLast();
61          while (curr != last) {
62              Horse* horse = curr->getData();
63              horse->unCheck();
64              if (horse->getID() == followerID) {
65                  follower = horse;
66              }
67              if (horse->getID() == leaderID) {
68                  leader = horse;
69              }
70              curr = curr->getNext();
71          }
72
73          if ((follower == nullptr)||(leader == nullptr)){
74              return false;
75          }
76          follower->inCircularReferance(this->totalMembers);
77          return leader->alreadyChecked();
78      }
79
80      bool can_run_together(){
81          int totalIndependant = 0;
82          Node<Horse>* curr = this->herdMembers.getHead();
83          Node<Horse>* last = this->herdMembers.getLast();
84          while (curr != last) {
85              Horse* horse = curr->getData();
86              horse->unCheck();
87              if (horse->independant()) {
88                  totalIndependant++;
89              }
90              curr = curr->getNext();
91          }
92          if (totalIndependant != 1){
93              return false;
94          }
95          curr = this->herdMembers.getHead();
96          last = this->herdMembers.getLast();
97          while (curr != last) {
98              Horse* horse = curr->getData();
99              if (horse->inCircularReferance(this->totalMembers)){
100                 return false;
101             }
```

```
102
103
104             curr = curr->getNext();
105         }
106
107
108         return true;
109     }
110 };
111
112 /**
113     Node<Horse>* curr = this->herdMembers.getHead();
114         Node<Horse>* last = this->herdMembers.getLast();
115         while (curr != last) {
116             Horse* horse = curr->getData();
117
118
119
120             curr = curr->getNext();
121     }
122  */
123
```

LinkedList.h Download

```
#pragma once
#define NULL_ID (-1)
#include  <new>
#include "ProjectFiles.h"

template <typename T>
class Node {
   //friend class LinkedList;
   public:
   T* data;
   Node<T>* next;
   Node<T>* previous;

public:
   explicit Node(T* data = nullptr, Node<T>* next = nullptr, Node<T>* previous = nullptr)
         : data(data), next(next), previous(previous) {}
   //~Node(){delete this->next;this->next = nullptr;previous = nullptr;data = nullptr;};
   ~Node() {
      // Do not recursively delete next node; let LinkedList manage memory.
      next = nullptr;
      previous = nullptr;
      data = nullptr;
   }

   Node<T>* getNext() {
      return this->next;
   }
   Node<T>* getPrevious() {
      return this->previous;
   }
   T* getData() {
      return this->data;
   }
   void setNext(Node<T>* newNext) {
      this->next = newNext;
   }
   void setPrevious(Node<T>* newPrevious) {
      this->previous = newPrevious;
   }
   void allPointersNull() {
      this->next->previous = this->previous;
      this->previous->next = this->next;
      this->next = nullptr;
      this->previous = nullptr;
      this->data = nullptr;
   }
};

template <typename T>
```

```
50  class LinkedList{
51  private:
52      Node<T>* head;
53      Node<T>* tail;
54
55  public:
56      LinkedList() {
57          this->head = new (std::nothrow) Node<T>();
58          this->tail = new (std::nothrow) Node<T>();
59          if (this->head == nullptr || this->tail == nullptr) {
60              throw StatusType::ALLOCATION_ERROR;
61          }
62          this->head->setNext(this->tail);
63          this->tail->setPrevious(this->head);
64      }
65
66      /**
67       *
68      ~LinkedList() {
69          Node<T>* current = head;
70          while (current) {
71              Node<T> *next = current->getNext();
72              delete current;
73              current = next;
74          }
75      }
76       */
77
78      //~LinkedList() = default;
79      Node<T>* getHead() {
80          return this->head->getNext();
81      }
82      ~LinkedList() {
83          Node<T>* current = head;
84          while (current) {
85              Node<T>* next = current->getNext();
86              delete current;  // Safely deletes each node
87              current = next;
88          }
89      }
90
91      bool insert(T* type){
92          Node<T>* newnode = new (std::nothrow) Node<T>(type);
93          if (!newnode) {
94              throw StatusType::ALLOCATION_ERROR;
95          }
96          this->head->next->previous = newnode;
97          newnode->setNext(this->head->getNext());
98          this->head->setNext(newnode);
99          newnode->setPrevious(this->head);
100
101         return true;
```

```
102     }
103
104     Node<T>* getFirst() {
105         return this->head->getNext();
106     }
107
108     Node<T>* getLast() {
109         return this->tail;
110     }
111
112     bool remove(T* type){
113         //for (Node<T> node : this->head){  ====== attempts to iterate over this->head directly, which is
    not iterable ======
114         for (Iterator it = begin(); it != end(); it++) {
115             Node<T>* node = it.current;
116
117             if (node->data != type){continue;}
118             if (node->previous != nullptr){
119                 node->previous->next = node->next;
120             }
121             if (node->next != nullptr){
122                 node->next->previous = node->previous;
123             }
124             node->data = nullptr; //linked list should not delete held data
125             delete node; //====== changed from delete &node ======
126             return true;
127         }
128         return false;
129     }
130
131     class Iterator {
132         Node<T>* current;
133     public:
134         explicit Iterator(Node<T>* node = nullptr) : current(node) {}
135         T& operator*() {
136             return *(current->getData());
137         }
138         Iterator& operator++() {
139             if (current) current = current->getNext();
140             return *this;
141         }
142 //        Iterator& operator--() {
143 //            if (current) current = current->previous;
144 //            return *this;
145 //        }
146         bool operator==(const Iterator& other) const {
147             return current == other.current;
148         }
149         bool operator!=(const Iterator& other) const {
150             return current != other.current;
151         }
152     };
```

```
153     Iterator begin() {
154         return Iterator(this->head->getNext());
155     }
156     Iterator end() {
157         return Iterator(this->tail->getPrevious());
158     }
159 };
160 
```

Herd.cpp Download

```
1  #include "Herd.h"
2  
3  //Herd::Herd(int id) : herdId(id), herdMembers(), totalMembers(0) {}
4  
5  //int Herd::getID() const{return this->herdId;}
6  //void Herd::setMembers(Horse* rep) {this->herdMembers = rep;}
7  
8  bool Herd::operator==(const Herd& otherHerd) const{
9      return this->herdId == otherHerd.herdId;
10 }
11 bool Herd::operator>(const Herd& otherHerd) const{
12     return this->herdId > otherHerd.herdId;
13 }
14 bool Herd::operator<(const Herd& otherHerd) const{
15     return this->herdId < otherHerd.herdId;
16 }
17 
```