```
// ========================================================================
//
// IMPORTANT NOTE: You should edit this file to uncomment test cases
//    and add your own additional test cases
//
// ========================================================================

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <cassert>
#include <cmath>
#include <ctime>
#include <cstdlib>

// pseudo-random number generator
#include "mtrand.h"

#include "traincar.h"


// Testing Function Prototypes
void SimpleTrainTest();
void ShipFreightTests();
void SeparateTests();
void StudentTests();


// ========================================================================
// ========================================================================

int main() {

  SimpleTrainTest();
  //ShipFreightTests();
  //SeparateTests();

  StudentTests();

  return 0;
}


// ========================================================================
// ========================================================================

// This helper function checks that the forward and backward pointers
// in a doubly-linked structure are correctly and consistently assigned.
void SanityCheck(TrainCar* train) {
  // an empty train is valid
  if (train == NULL) return;
  // the input must be the head car of a train
  assert (train->prev == NULL);
  TrainCar *tmp = train;
  while (tmp->next != NULL) {
```

```
    // the next train better point back to me
    assert (tmp->next->prev == tmp);
    tmp = tmp->next;
  }
}


// ========================================================================
// ========================================================================

// This helper function prints one of the 5 rows of the TrainCar ASCII art
void PrintHelper(TrainCar* t, int which_row) {
  if (t == NULL) {
    // end of the line
    std::cout << std::endl;
    return;
  }

  if (which_row == 0) {
    // the top row only contains "smoke" for engine traincars
    if (t->isEngine()) {
      std::cout << "     ~~~~";
    } else {
      std::cout << "         ";
    }
  } else if (which_row == 1) {
    // the 2nd row only contains the smoke stack for engine traincars
    if (t->isEngine()) {
      std::cout << "    ||   ";
    } else {
      std::cout << "         ";
    }
  } else if (which_row == 2) {
    // the 3rd row contains the ID for each traincar
    // (and engine traincars are shaped a little differently)
    if (t->isEngine()) {
      std::cout << "   " << std::setw(6) << std::setfill('-') << t->getID();
    } else {
      std::cout << std::setw(9) << std::setfill('-') << t->getID();
    }
    std::cout << std::setfill(' ');
  } else if (which_row == 3) {
    // the 4th row is different for each TrainCar type
    if (t->isEngine()) {
      std::cout << " / ENGINE";
    } else if (t->isFreightCar()) {
      // freight cars display their weight
      std::cout << "|" << std::setw(5) << t->getWeight() << "  |";
    } else if (t->isPassengerCar()) {
      // passenger cars are simple empty boxes
      std::cout << "|       |";
    } else if (t->isDiningCar()) {
      std::cout << "|  dine |";
    } else {
      assert (t->isSleepingCar());
      std::cout << "| sleep |";
```

```
    }
  } else if (which_row == 4) {
    // final row is the same for all cars, just draw the wheels
    std::cout << "-oo---oo-";
  }

  // between cars display the '+' link symbol on the 5th row
  // (only if there is a next car)
  if (t->next != NULL) {
    if (which_row == 4) {
      std::cout << " + ";
    } else {
      std::cout << "   ";
    }
  }

  // recurse to print the rest of the row
  PrintHelper(t->next, which_row);
}


void PrintTrain(TrainCar* train) {

  if (train == NULL) {
    std::cout << "PrintTrain: empty train!" << std::endl;
    return;
  }

  // Print each of the 5 rows of the TrainCar ASCII art
  PrintHelper(train, 0);
  PrintHelper(train, 1);
  PrintHelper(train, 2);
  PrintHelper(train, 3);
  PrintHelper(train, 4);

  /*
  // UNCOMMENT THESE ADDITIONAL STATISTICS AS YOU WORK

  int
      total_weight,num_engines,num_freight_cars,num_passenger_cars,num_dining_car
      s,num_sleeping_cars;

      TotalWeightAndCountCars(train,total_weight,num_engines,num_freight_cars,num
      _passenger_cars,num_dining_cars,num_sleeping_cars);
  int total_cars = num_engines+num_freight_cars+num_passenger_cars
      +num_dining_cars+num_sleeping_cars;
  float speed = CalculateSpeed(train);
  std::cout << "#cars = " << total_cars;
  std::cout << ", total weight = " << total_weight;
  std::cout << ", speed on 2% incline = " << std::setprecision(1) << std::fixed
      << speed;

  // If there is at least one passenger car, print the average
  // distance to dining car statistic
  if (num_passenger_cars > 0) {
    float dist_to_dining = AverageDistanceToDiningCar(train);
```

```
    if (dist_to_dining < 0) {
      // If one or more passenger cars are blocked from accessing the
      // dining car (by an engine car) then the distance is infinity!
      std::cout << ", avg distance to dining = inf";
    } else {
      std::cout << ", avg distance to dining = " << std::setprecision(1) <<
          std::fixed << dist_to_dining;
    }
  }

  // If there is at least one sleeping car, print the closest engine
  // to sleeper car statistic
  if (num_sleeping_cars > 0) {
    int closest_engine_to_sleeper = ClosestEngineToSleeperCar(train);
    std::cout << ", closest engine to sleeper = " << closest_engine_to_sleeper;
  }

  std::cout << std::endl;
  */
}

// ========================================================================
// ========================================================================

void SimpleTrainTest() {
  std::cout <<
      "========================================================================
      ==" << std::endl;
  std::cout << "SIMPLE TRAIN TEST" << std::endl;

  // create a train with 6 dynamically-allocated cars in a doubly-linked list
      structure
  TrainCar* simple = NULL;
  PushBack(simple, TrainCar::MakeEngine());
  PushBack(simple, TrainCar::MakePassengerCar());
  PushBack(simple, TrainCar::MakePassengerCar());
  PushBack(simple, TrainCar::MakeDiningCar());
  PushBack(simple, TrainCar::MakePassengerCar());
  PushBack(simple, TrainCar::MakeSleepingCar());

  // inspect the cars, the links, the links, and sequential IDs...
  assert (simple->isEngine());
  assert (simple->prev == NULL);
  assert (simple->next->isPassengerCar());
  assert (simple->next->prev->isEngine());
  assert (simple->next->next->isPassengerCar());
  assert (simple->next->next->next->isDiningCar());
  assert (simple->next->next->next->next->isPassengerCar());
  assert (simple->next->next->next->next->next->isSleepingCar());
  assert (simple->next->next->next->next->next->next == NULL);
  assert (simple->next->getID() == simple->getID()+1);
  assert (simple->next->next->getID() == simple->next->getID()+1);
  assert (simple->next->next->next->getID() == simple->next->next->getID()+1);
  assert (simple->next->next->next->next->getID() == simple->next->next->next->
      getID()+1);
  assert (simple->next->next->next->next->next->getID() == simple->next->next->
```

```
      next->next->getID()+1);

  // helper routine sanity check & print the results
  SanityCheck(simple);
  PrintTrain(simple);

  // fully delete all TrainCar nodes to prevent a memory leak
  DeleteAllCars(simple);
}

// ========================================================================
// ========================================================================

// This function takes a random number generator to create variety in
// the freight car weights
void ShipFreightHelper(MTRand_int32 &mtrand, int num_engines, int num_cars, int
    min_speed, int max_cars_per_train) {

  /*

  // UNCOMMENT THIS FUNCTION WHEN YOU'RE READY TO TEST SHIP FREIGHT

  // create a chain with specified # of engines engines
  TrainCar* all_engines = NULL;
  for (int i = 0; i < num_engines; i++) {
    PushBack(all_engines, TrainCar::MakeEngine());
  }
  // create a chain with specified # of freight cars
  TrainCar* all_freight = NULL;
  for (int i = 0; i < num_cars; i++) {
    // the weight for each car is randomly generated in the range of 30->100 tons
    int weight = 30 + (mtrand()%15)*5;
    PushBack(all_freight, TrainCar::MakeFreightCar(weight));
  }

  // rearrange the two structures into a collection of trains
  // with the specified minimum speed & specified maximum length 12 cars
  std::vector<TrainCar*> trains = ShipFreight(all_engines, all_freight,
      min_speed, max_cars_per_train);

  // when finished, we have either used up all of the engines, or
  // shipped all the freight (or both!)
  assert (all_engines == NULL || all_freight == NULL);

  // print the remaining engines or freight cars
  if (all_engines != NULL) {
    std::cout << "Remaining Unused Engines:" << std::endl;
    SanityCheck(all_engines);
    PrintTrain(all_engines);
  }
  if (all_freight != NULL) {
    std::cout << "Remaining UnShipped Freight:" << std::endl;
    SanityCheck(all_freight);
    PrintTrain(all_freight);
  }
```

```
  // print the trains
  std::cout << "Prepared Trains for Shipment:" << std::endl;
  for (unsigned int i = 0; i < trains.size(); i++) {
    SanityCheck(trains[i]);
    PrintTrain(trains[i]);

    // check that the speed and length rules are followed
    int
        total_weight,num_engines,num_freight_cars,num_passenger_cars,num_dining_c
        ars,num_sleeping_cars;

        TotalWeightAndCountCars(trains[i],total_weight,num_engines,num_freight_ca
        rs,num_passenger_cars,num_dining_cars,num_sleeping_cars);
    int total_cars = num_engines+num_freight_cars+num_passenger_cars
        +num_dining_cars+num_sleeping_cars;
    float speed = CalculateSpeed(trains[i]);
    assert (total_cars <= max_cars_per_train);
    assert (speed >= min_speed);
  }

  // fully delete all TrainCar nodes to prevent memory leaks
  DeleteAllCars(all_engines);
  DeleteAllCars(all_freight);
  for (unsigned int i = 0; i < trains.size(); i++) {
    DeleteAllCars(trains[i]);
  }
  */
}


void ShipFreightTests() {

  // We make two different pseudo-random number generators.

  // With a fixed seed, we will see the same sequence of numbers
  // everytime we run the program (very helpful for debugging).
  MTRand_int32 mtrand_fixed_seed(42);
  std::cout <<
      "=============================================================================
      ==" << std::endl;
  std::cout << "SHIP FREIGHT TEST, FIXED SEED" << std::endl;
  ShipFreightHelper(mtrand_fixed_seed,10,25,60,12);

  /*

  // UNCOMMENT THIS FUNCTION WHEN THE FIXED SEED SHIP FREIGHT TEST LOOKS GOOD

  // Alternatively, we can let the seed be set from the computer
  // clock, so the number sequence will be different each time the
  // program is run.
  MTRand_int32 mtrand_autoseed(time(NULL));
  for (int i = 1; i <= 5; i++) {
    std::cout <<
        "=============================================================================
        ====" << std::endl;
    std::cout << "SHIP FREIGHT TEST, RANDOM SEED #" << i << std::endl;
```

```
    ShipFreightHelper(mtrand_autoseed,6,25,65,10);
  }
  */
}


// ========================================================================
// ========================================================================

// Helper function that writes down the train car IDs into a vector,
// to allow faster calculation of link/unlink/shift costs
std::vector<int> RecordIDs(TrainCar* t1) {
  std::vector<int> answer;
  TrainCar* tmp = t1;
  // loop over all the train cars and store the car IDs in a vector
  while (tmp != NULL) {
    answer.push_back(tmp->getID());
    tmp = tmp->next;
  }
  return answer;
}


// Given the vectors of IDs before & after separation, verify that no
// cars are missing or duplicated, and determine the number of unlink,
// link, and train shifts that are necessary to create these two trains
void SeparateStatistics(const std::vector<int> &original,
                        const std::vector<int> &left,
                        const std::vector<int> &right,
                        int &num_unlinks, int &num_links, int &num_shifts) {

  // Simple checks on the number ot total cars in the trains
  if (original.size() < (left.size() + right.size())) {
    std::cerr << "ERROR: One or more extra cars after Separate" << std::endl;
    return;
  }
  if (original.size() > (left.size() + right.size())) {
    std::cerr << "ERROR: One or more missing cars after Separate" << std::endl;
    return;
  }

  // initialize the counter variables
  num_links = 0;
  num_unlinks = 0;
  num_shifts = 0;

  // loop over all of the cars in the original train
  for (int i = 0; i < int(original.size()); i++) {
    int found = false;
    // before and after will store the car's new neighbors
    int before = -1;
    int after = -1;
    // find this car in the left or right separated trains
    for (int j = 0; j < int(left.size()); j++) {
      if (original[i] == left[j]) {
        assert (found == false);
```

```
        found = true;
        num_shifts += abs(i-j);
        if (j > 0) before = left[j-1];
        if (j < int(left.size())-1) after = left[j+1];
      }
    }
    for (int k = 0; k < int(right.size()); k++) {
      if (original[i] == right[k]) {
        assert (found == false);
        found = true;
        num_shifts += abs(i-(int(left.size())+k));
        if (k > 0) before = right[k-1];
        if (k < int(right.size())-1) after = right[k+1];
      }
    }

    if (found == false) {
      std::cerr << "ERROR: Missing ID=" << original[i] << std::endl;
      return;
    }

    // special cases for first & last car links
    if (i == 0 && before != -1) { num_links++; }
    if (i == int(original.size())-1 && after != -1) { num_links++; }
    // middle links
    if (i > 0) {
      if (original[i-1] != before) {
        num_unlinks++;
        if (before != -1) num_links++;
      }
    }
  }
  // Note: swapping neighboring cars counts as a two units of shift
  //   the total number of shifts must be even!
  assert (num_shifts %2 == 0);
}


// Each different input train configuration to Separate is handled similarly
void SeparateTestHelper(TrainCar* &train1, const std::string &which_test) {
  /*

  // UNCOMMENT THIS FUNCTION WHEN YOU'RE READY TO TEST SEPARATE

  std::cout <<
      "================================================================================
      ==" << std::endl;
  std::cout << "SEPARATE TRAINS " << which_test << std::endl;

  SanityCheck(train1);
  // record the original IDs for later comparison and statistics calculation
  std::vector<int> original = RecordIDs(train1);
  PrintTrain(train1);
  float speed_original = CalculateSpeed(train1);

  TrainCar* train2;
```

```
  TrainCar* train3;
  Separate(train1, train2, train3);
  assert (train1 == NULL);
  SanityCheck(train2);
  SanityCheck(train3);
  // record the IDs after separation
  std::vector<int> left = RecordIDs(train2);
  std::vector<int> right = RecordIDs(train3);

  // calculate the number of links, unlinks, and train shifts
  // (all of these counts should be kept small to minimize train yard costs
  int num_unlinks, num_links, num_shifts;
  SeparateStatistics(original, left, right, num_unlinks, num_links, num_shifts);
  std::cout << "Separate Statistics: num unlinks = " << num_unlinks;
  std::cout << ", num links = " << num_links;
  std::cout << ", num shifts = " << num_shifts;
  std::cout << "   Total Cost: " << num_unlinks+num_links+num_shifts <<
      std::endl;

  float speed_left = CalculateSpeed(train2);
  float speed_right = CalculateSpeed(train3);
  float left_percent = 100.0 * (speed_original-speed_left) / speed_original;
  float right_percent = 100.0 * (speed_original-speed_right) / speed_original;
  if (speed_left < 0.99*speed_original) {
    assert (speed_right > speed_original);
    std::cout << "left train is " << std::setprecision(1) << std::fixed <<
        left_percent
              << "% slower than the original and the right train is " <<
                  std::setprecision(1) << std::fixed
              << -right_percent << "% faster than the original." << std::endl;
  } else if (speed_right < 0.99*speed_original) {
    assert (speed_left > speed_original);
    std::cout << "right train is " << std::setprecision(1) << std::fixed <<
        right_percent
              << "% slower than the original and the left train is " <<
                  std::setprecision(1) << std::fixed
              << -left_percent << "% faster than the original." << std::endl;
  } else {
    std::cout << "  left and right train speeds are equal to the original." <<
        std::endl;
  }


  PrintTrain(train2);
  PrintTrain(train3);
  // cleanup memory usage
  DeleteAllCars(train2);
  DeleteAllCars(train3);
  */
}


// Several specific test cases for the Separate Trains function
void SeparateTests() {

  TrainCar* t;
```

```
/*

// UNCOMMENT THESE TESTS ONE AT A TIME AS YOU WORK ON SEPARATE

t = NULL;
PushBack(t, TrainCar::MakeEngine());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
PushBack(t, TrainCar::MakeSleepingCar());
PushBack(t, TrainCar::MakeSleepingCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeEngine());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
PushBack(t, TrainCar::MakePassengerCar());
SeparateTestHelper(t, "#1");

t = NULL;
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
PushBack(t, TrainCar::MakeSleepingCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeEngine());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeSleepingCar());
PushBack(t, TrainCar::MakeEngine());
SeparateTestHelper(t, "#2");

t = NULL;
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeEngine());
PushBack(t, TrainCar::MakeEngine());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
SeparateTestHelper(t, "#3");

t = NULL;
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
PushBack(t, TrainCar::MakeSleepingCar());
PushBack(t, TrainCar::MakeEngine());
PushBack(t, TrainCar::MakeEngine());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeDiningCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakePassengerCar());
PushBack(t, TrainCar::MakeSleepingCar());
```

```
  SeparateTestHelper(t, "#4");

  t = NULL;
  PushBack(t, TrainCar::MakeEngine());
  PushBack(t, TrainCar::MakeEngine());
  PushBack(t, TrainCar::MakePassengerCar());
  PushBack(t, TrainCar::MakePassengerCar());
  PushBack(t, TrainCar::MakeDiningCar());
  PushBack(t, TrainCar::MakeSleepingCar());
  PushBack(t, TrainCar::MakePassengerCar());
  PushBack(t, TrainCar::MakeDiningCar());
  PushBack(t, TrainCar::MakePassengerCar());
  PushBack(t, TrainCar::MakePassengerCar());
  PushBack(t, TrainCar::MakeSleepingCar());
  SeparateTestHelper(t, "#5");
  */

  // Note: SeparateTestHelper takes care of deleting all memory
  // associated with these tests
}


// ============================================================================
// ============================================================================

void StudentTests() {

  std::cout <<
      "================================================================================
      ==" << std::endl;
  std::cout << "STUDENT TESTS" << std::endl;

  //
  //
  // Write your own test cases here
  //
  //

  std::cout << "StudentTests complete" << std::endl;
}


// ============================================================================
// ============================================================================
```