TrailManager

Design Proposal

Ben Morris

CSC316: Data Structures & Algorithms

bcmorri4@ncsu.edu

North Carolina State University

Department of Computer Science

06.16.22

System Test Plan

Test Data:
For my system test cases, I will use two test files: landmarkinformationtest.txt and trailstest.txt.

landmarkinformation_test.txt

```
LANDMARK_ID,DESCRIPTION,TYPE
L01,Park Entrance,Location
L02,Entrance Fountain,Fountain
L03,Waste Station 1,Pet Waste Station
L04,Entrance Restrooms,Restroom
L05,Overlook 1,Overlook
L06,Rock Formation 1,Rock Formation
L07,Overlook 2,Overlook
L08,Overlook Restrooms,Restroom
L09,Waste Station 2,Pet Waste Station
L10,Hidden Gardens,Gardens
L11,Campsite 1,Campsite
L12,Campsite Restrooms,Restroom
```

(test data continued)

trails_test.txt

```
LANDMARK_ID,LANDMARK_ID,DISTANCE
L01,L02,3013
L01,L03,1046
L01,L04,1179
L02,L10,3613
L03,L05,4204
L04,L09,2311
L05,L06,1039
L06,L07,2912
L07,L08,1891
L11,L12,1066
```

To start the program, run ProgramManager.java

Test ID	Description	Expected Results	Actual Results
Test #1 **testID: testLoadLandmarks** **Strategy: Equivalence class - loading landmarks from file**	**Preconditions:** 1. ProgramManager.java has been loaded successfully. 2. landmarkinformation_test.txt exists. **Steps:** 1. The user is prompted for the path to the file of landmarks. 2. The user inputs a path to landmarkinformation_test.txt and presses enter.	1. The user is prompted for the path to the file of trails.	
Test #2 **testID: testLoadLandmarksFails** **Strategy: Exception/unexpected input – trying to add an invalid file path**	**Preconditions:** 1. ProgramManager.java has been loaded successfully. 2. landmarkinformation_test.txt exists. **Steps:** 1. The user is prompted for the path to the file of landmarks. 2. The user inputs a file to a path that doesn't lead to a text file.	1. The program re-prompts the user for the path to the file of landmarks.	

Test #3 **testID: testLoadTrails** **Strategy: Equivalence class - loading trails from file**	**Preconditions:** 1. ProgramManager.java has been loaded successfully. 2. trails_test.txt exists **Steps:** 1. The user is prompted for the path to the file of landmarks. 2. The user inputs a path to landmarkinformation_test.txt and presses enter.	The user is presented with a menu of three options: 1. Show List of Potential First Aid Stations 2. View Distances to All Reachable Landmarks 3. Close Trail Manager	
Test #4 **testID: testLoadTrailsFails** **Strategy: Exception/unexpected input – trying to add an invalid file path**	**Preconditions:** 1. ProgramManager.java has been loaded successfully. 2. trails_test.txt exists. **Steps:** 1. The user is prompted for the path to the file of landmarks. 2. The user inputs a file to a path that doesn't lead to a text file.	1. The program re-prompts the user for the path to the file of trails.	

Test #5 **testID: testRunFirstAidList_Zero** **Strategy: Boundary value - asking for intersection of zero paths**	**Preconditions:** 1. ProgramManager.java has been loaded successfully. 2. Landmarkinformation_test.txt and trails_test.txt have been successfully loaded. **Steps:** 1. The user selects option #1: Show List of Potential First Aid Stations. 2. The user is prompted for the minimum number of trails that should intersect at a landmark such that the landmark is a suitable location for a first aid station. 3. The user enters “0”.	1. The program outputs “Number of intersecting trails must be greater than 0.” and re-prompts the user for the number of intersecting paths.	

Test #6 **testID: testRunFirstAidList_TooMany** **Strategy: Exception/unexpected input – trying to add an invalid minimum number of intersections**	**Preconditions:** 1. ProgramManager.java has been loaded successfully. 2. Landmarkinformation_test.txt and trails_test.txt have been successfully loaded. **Steps:** 1. The user selects option #1: Show List of Potential First Aid Stations. 2. The user is prompted for the minimum number of trails that should intersect at a landmark such that the landmark is a suitable location for a first aid station. 3. The user enters “12”.	1. The program outputs “No landmarks have at least X intersecting trails.” and re-prompts the user for the number of intersecting paths.	

Test #7
 	
testID: testRunFirstAidList_Two
 	
Strategy: Equivalence class - requesting an acceptable number of intersections for Show List of Potential First Aid Stations
	Preconditions:
	1. ProgramManager.java has been loaded successfully.
	2. Landmarkinformation_test.txt and trails_test.txt have been successfully loaded.
	
	Steps:
	
	1. The user selects option #1: Show List of Potential First Aid Stations.
	2. The user is prompted for the minimum number of trails that should intersect at a landmark such that the landmark is a suitable location for a first aid station.
	3. The user enters “2”.
		The program outputs:
		`Proposed Locations for First Aid Stations {`
		`  Park Entrance (L01) - 3 intersecting trails`
		`  Entrance Fountain (L02) - 2 intersecting trails`
		`  Entrance Restrooms (L04) - 2 intersecting trails`
		`  Overlook 1 (L05) - 2 intersecting trails`
		`  Overlook 2 (L07) - 2 intersecting trails`
		`  Rock Formation 1 (L06) - 2 intersecting trails`
		`  Waste Station 1 (L03) - 2 intersecting trails`
		`}`
		
		`Press [Enter] to continue`	

See the appendix for additional system tests.

Algorithm Design

```
Algorithm: getDistances(L, s)
Inputs: L, a map of Landmark -> list of trails that intersect the landmark
        start, the starting landmark to begin calculating distances
Output: a map of landmarks in the park -> distances (in feet) to those Landmarks from
        start

// Create a new map to store Landmarks and distances to start
M <- new empty map

// Create previous to check if next node has already been visited
previous <- new empty Landmark
// Initialize previous to null
previous <- null
// Initialize totalDistance to zero
totalDistance <- 0

// Add the starting point and zero distance to the distance map
M.put(start, totalDistance)

// Call helper method getNeighborDistances() to iterate through start's neighbors
getNeighborDistances(L, start, previous, M, totalDistance)

return M
```

```
Algorithm: getNeighborDistances(L, current, previous, M, totalDistance)
Inputs: L, a map of Landmark -> list of trails that intersect the landmark
        current, the starting landmark to begin calculating distances
        previous, the previous landmark (before start)
        M, a map of Landmarks -> distances (in feet) to those landmarks from the
          original start
        totalDistance, the total distance (in feet) to the original start
Output: none


// Get the list of Trails intersecting current
T <- new empty list of Trails
T <- L.get(current)
for each Trail t in T
    next <- new empty Landmark
    // Use helper method getOtherEndpoint to get the other endpoint of the Trail, t
    next <- t.getOtherEndpoint(current)

    // (Since there are no loops, the only way for next to have been already visited is
    // if it were the previous node to current.) If the next Landmark is not the same
    // as the previous Landmark to current, add to the total distance and put the new
    // Landmark in the distance Map, M.
    if next != previous
        totalDistance <- totalDistance + t.getDistance()
        M.put(next, totalDistance)
        //Call getNeighborDistances() for next with current becoming the previous node
        getNeighborDistances(L, next, current, M, totalDistance)
        //Decrement totalDistance after all of start's neighbors' distances are
        //calculated before moving to the next neighbor of previous
        totalDistance <- totalDistance - t.getDistance()
```

```
Algorithm: getOtherEndpoint(start)
    Input: start, one end of the Trail, t
    Output: the Landmark at the other end of the trail

    //Return the endpoint that start is not equal to
    if t.getEndpointOne() = start
        return t.getEndpointTwo()
    else
        return t.getEndpointOne()
```

Data Structures

I will use the following abstract data types:

- A map, mapping each Landmark to a list of intersecting Trails.
- Another map, mapping each Landmark to its distance from the chosen starting point
- A list for sorting the distance map by distance.

I chose maps for the first two because each key is unique and will be able to be looked up using a skip list in expected O(log n) time, which is as efficient as we can do at this point.

I chose a list so that I can use counting sort to sort the list by distance.

I will use the following data structures:

- Because the program constraints require time efficiency over memory efficiency, I will use an ordered **skip list** linked-list data structure to implement the main map. This will allow for the efficiency of an approximate binary search while also providing expected O(log n) search and insert operations. For the DistanceMap, I will also use a skip-list to create a map of alphabetically-ordered landmarks to their distance from the chosen Landmark. Later, I will sort this map by distance.
- I will use a fixed array for the list to sort by distance. I have three reasons for this choice. The first reason is that I have chosen counting sort as the most efficient method to sort and I need an array to do so. Second, I can create the unsorted array in O(n) time by copying the map to the array (skip lists are O(log n) to get and arrays are O(1) to addLast() for each of the n entries). Last, I can choose a fixed array because I know how many entries there will be.

I will use the following search algorithms:
Once completed, the entries in the DistanceMap will need to be sorted by value. Because it's a numerical sort and because time is a constraint, I will use **counting sort** to optimize the performance. Counting sort has a performance of O(n + k), where k could be as large as 16,000,000 feet for the longest trail in the US (the Continental Divide Trail), but is likely no larger than 1,000,000 feet, which would give the same or better efficiency than radix sort for the constrained data set of 262,144 Landmarks and much better than bubble, insertion, or selection sorts. For a smaller data set, radix sort, mergesort, or quicksort might be more efficient, depending on the data.

- Bubble sort, insertion sort, and selection sort: $O(n^2)$
- Radix sort: O(wn), where in this case w is likely to be between 4 and 7 or O(7n) = O(n)
- Mergesort and Quicksort: O(n * log n), where $\log_2$ 262,144 is 18 or O(18 n) = O(n)

I will be using Mergesort to sort alphabetical lists..

Algorithm Analysis

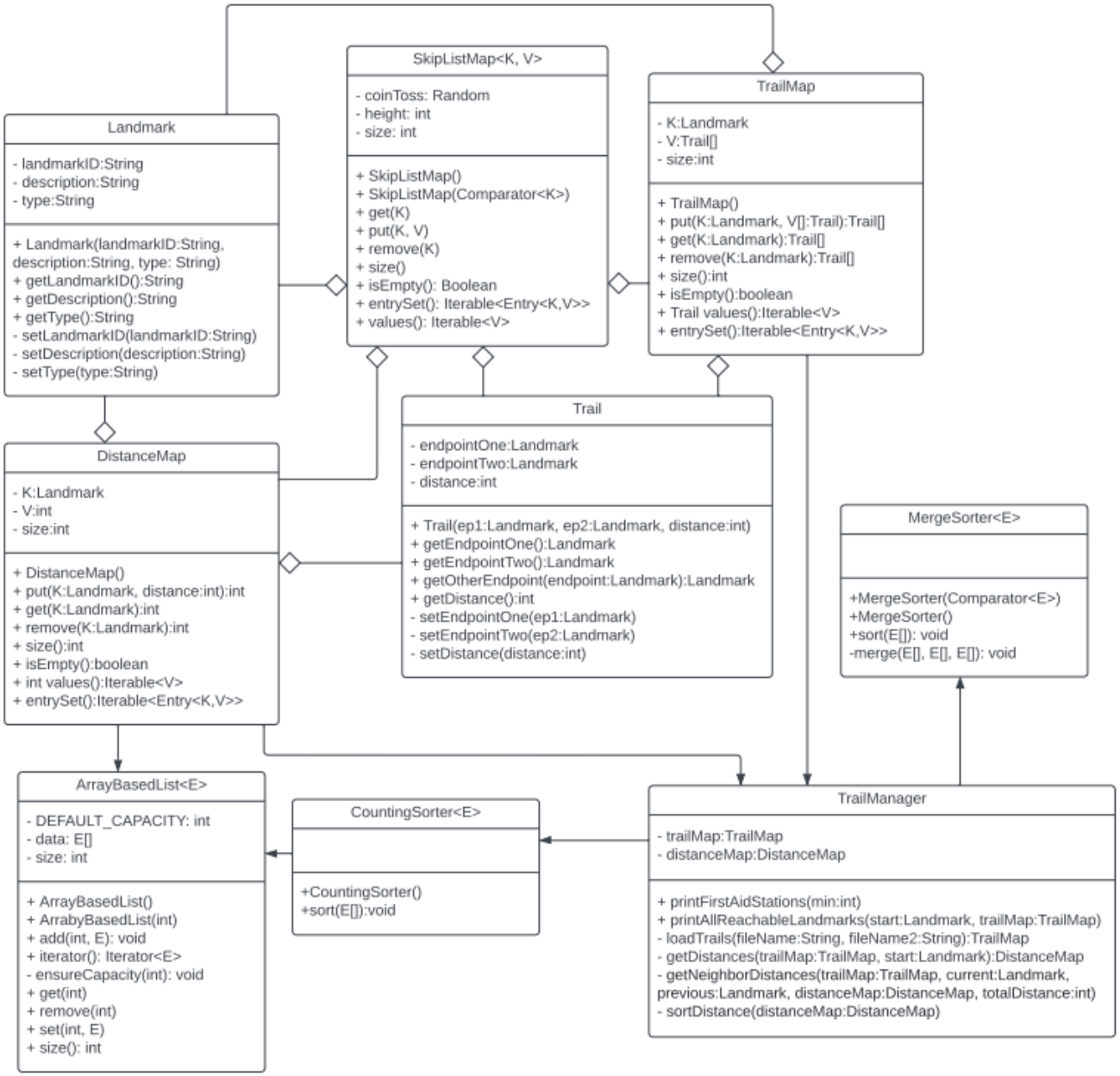
Algorithm:	Analysis/Runtime Rationale
See algorithm on pages 8 and 9 for comments. **Algorithm:** getNeighborDistances(L, current, previous, M, totalDistance) **Inputs:** L, a map of Landmark -> list of trails that intersect the landmark current, the starting landmark to begin calculating distances previous, the previous landmark (before start) M, a map of Landmarks -> distances (in feet) to those landmarks from the original start totalDistance, the total distance (in feet) to the original start **Output:** none 1 T <- new empty list of Trails 2 T <- L.get(current) 3 for each Trail t in T 4 next <- new empty Landmark 5 next <- t.getOtherEndpoint(current) 6 if next != previous 7 totalDistance <- totalDistance + t.getDistance() 8 M.put(next, totalDistance) 9 getNeighborDistances(L, next, current, M, totalDistance) 10 totalDistance <- totalDistance - t.getDistance()	getDistances is dependent on getNeighborDistances(), but is O(log n) because of its put(). No matter how the trails are distributed, the most number of trails for n Landmarks without creating a loop is n - 1, which means the loop at lines 3-10 will run at most n - 1 times. Lines 4, 5, 6, 7, and 10 are all O(1). Line 8: put() for a skip list is dependent on search and is O(log n) Line 9: The method call is performed at most n-1 times whether that is all at once or spread out over many trails because there are at most n - 1 trails (see above). It has an internal O(log n) for a total O(n * log n). T(n) = log n + O(n*log n) or an overall O(n log n).

Intentionally left blank

Software Design

Description:
For this software, I will use the Factory, Facade, and Model-View-Controller design patterns to allow the client to access distance reports and suggested First Aid Station reports from trail map data that they provide. The model is the trail system as modeled through the skip list maps of Landmarks, trails, and distances. The view is the UI. And the controller is the software handling the choices from the user menu. I decided to use MVC because it allows the user to create reports from a standardized input file.They only need to provide the standardized data for the model and choose the report they want via the UI.

UML Class Diagram:



Appendix

Test #8 **testID: testRunDistances_NotValidLandmark** **Strategy: Exception/unexpected input – trying to add an invalid landmark**	**Preconditions:** 1. ProgramManager.java has been loaded successfully. 2. Landmarkinformation_test.txt and trails_test.txt have been successfully loaded. **Steps:** 1. The user selects option #2: View Distances to All Reachable Landmarks. 2. The user is prompted for the minimum number of trails that should intersect at a landmark such that the landmark is a suitable location for a first aid station. 3. The user enters “L222”.	1. The program outputs “The provided landmark ID (X) is invalid for the park.” and re-prompts the user for a valid landmark.	

Test #9
 	
testID: testRunDistances
 	
Strategy: Equivalence class - requesting an acceptable landmark for View Distances to All Reachable Landmarks
	Preconditions:
	1. ProgramManager.java has been loaded successfully.
	2. Landmarkinformation_test.txt and trails_test.txt have been successfully loaded.
	
	Steps:
	
	1. The user selects option #2: View Distances to All Reachable Landmarks.
	2. The user is prompted for the minimum number of trails that should intersect at a landmark such that the landmark is a suitable location for a first aid station.
	3. The user enters “L02”.
		1. The program outputs:
		
		`Landmarks Reachable from Entrance Fountain (L02) {`
		
		`3013 feet to Park Entrance (L01)`
		
		`3613 feet to Hidden Gardens (L10)`
		
		`4059 feet to Waste Station 1 (L03)`
		
		`4192 feet to Entrance Restrooms (L04)`
		
		`6503 feet (1.23 miles) to Waste Station 2 (L09)`
		
		`8263 feet (1.56 miles) to Overlook 1 (L05)`
		
		`9302 feet (1.76 miles) to Rock Formation 1 (L06)`
		
		`12214 feet (2.31 miles) to Overlook 2 (L07)`
		
		`14105 feet (2.67 miles) to Overlook Restrooms (L08)`
		
		`}`
		
		`Press [Enter] to continue`	