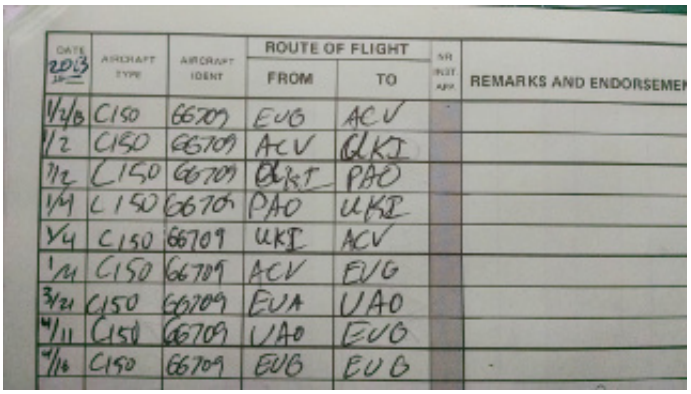
CSCI 145

Project 3

100pts



DATE 2013	AIRCRAFT TYPE	AIRCRAFT IDENT	ROUTE OF FLIGHT FROM	TO	SR INST APP	REMARKS AND ENDORSEME
1/1	C150	66709	EUG	ACV		
1/2	C150	66709	ACV	UKI		
1/2	C150	66709	UKI	PAO		
1/4	C150	66709	PAO	UKI		
1/4	C150	66709	UKI	ACV		
1/21	C150	66709	ACV	EUG		
3/21	C150	66709	EUA	UAO		
4/11	C150	66709	UAO	EUG		
7/16	C150	66709	EUG	EUG		

Figure 1: Sample logbook entries.

Flight Reassembly

Like all pilots, I keep a logbook of the flights I have flown. There are times when I need to show how many hours I have flown in different conditions. I can also use it to reminisce.

I once got to fly my Cessna 150 from Eugene, OR to Palo Alto, CA. The first three lines of Figure 1 shows my logbook entries for that flight. Airports are identified by a three or four character identifier.[1] On the second of January, I flew from Eugene (EUG) to Arcata, CA (ACV). I then flew on to Palo Alto (PAO). The plane probably could have made it from EUG to PAO without taking on fuel at ACV, but I was not 100% sure it could make it.

As a computer scientist, this looks like a linked list of Airports to me:

$$\text{EUG} \rightarrow \text{ACV} \rightarrow \text{PAO}.$$

Since my handwriting is poor and difficult to read, I tried to digitize my logbook. That way it would be easier to read and maintain. Unfortunately, after entering in all of the longer cross-country flights, I sorted the list. So the flight from Eugene to PAO is listed backwards:

from	to
ACV	PAO
EUG	ACV

[1] They are really four character identifiers, but the last three are unique in the United States, so we generally leave off the leading K or P.

For short flights like this, I can see that the two legs of the flight are reversed. So this is not really a problem. However, there are some longer flights in my logbook. I am sure that the FAA will not accept this as a paper logbook replacement.

Task Description

Write a program called `Flight` that will reconstruct the correct order of airports for a few of my flights.

Flight File

The flights are in separate text files and they all list the airport I left from and the airport I flew to. Each line is one leg or "hop." For example, the Palo Alto flight file would looks like this:

```
ACV PAO
EUG ACV
```

The program should read from a file that is specified on the command line. For example, if the Palo Alto flight were in a file called `pao.txt`, then I should see the following program behavior:

```
$ java Flight pao.txt
EUG
ACV
PAO
```

Locations

So we can verify that the order is correct, your program must be able to print the latitude and longitude of the airports when the given the `-l` command line option. The file `ap_loc.txt` contains the identifier, latitude, and longitude for each of 5104 public airports in the United States.

Printing coordinates is selected with the `-l` command line option followed by the `ap_loc.txt` file. The output should be the latitude, longitude, and airport identifier separated by commas. For example, the following is the output for the Palo Alto flight with coordinates:

```
$ java Flight -l ap_loc.txt pao.txt
44.1245833333333, -123.211972222222, EUG
40.9778333333333, -124.108472222222, ACV
37.4611111111111, -122.115055555556, PAO
```

To visualize this flight, copy-and-paste the program's output to the Coordinates box in the webstie `www.darrinward.com/lat-long`. This will generate the map shown in Figure 2. Technically speaking, the airport identifier is unused by the website.

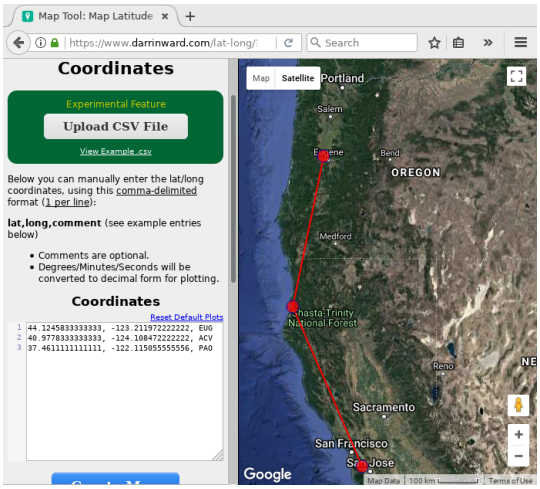


Figure 2: Coordinates mapped by `www.darrinward.com/lat-long`.

Solution Method

In this project, you will use three different kinds of linked lists. The first is a linked list of strings (`LinkedList<String>`) that contains airport identifiers. The second is a linked list of these linked lists of strings (`LinkedList<LinkedList<String>>`). Initially each hop is a linked list of two airports. These hops are collected into a list of lists. Finally, you will need to define an `Airport` class that has id, latitude, and longitude fields so your program can print locations. These airports will be contained within a linked list of airports (`LinkedList<Airport>`).

Outline of solution

Open the flight file (e.g., `pao.txt`) and construct one linked list of strings for each line in the file. Since each line is exactly two words, the pseudo-code looks something like this:

```
read_hop(Scanner sc):
  LinkedList<String> hop = new LinkedList<String>()
  hop.add(sc.next())
  hop.add(sc.next())
  return hop
```

Keep reading hops until the entire file is read:

```
read_file(Scanner sc)
  LinkedList<LinkedList<String>> flight = new LinkedList<LinkedList<String>>();
  while(sc.hasNext())
    flight.add(read_hop(sc))
```

Now you have a linked list of linked lists of strings. If we use square brackets as list delimiters, the initial state of the Palo Alto flight looks the this:

```
[[ACV, PAO], [EUG, ACV]]
```

We are going to collect these hops into longer and longer sequences of airports. Since the word hop does not really describe a sequence of more than two airports, let us call lists of airports routes. The sample list above contains two routes: `[ACV, PAO]` and `[EUG, ACV]`. The plan is to incrementally combine routes together until there is just one route in the list.

Select a route and remove it from the list of all routes. Unless this is the last route in the entire flight, the last airport the selected route must be the first airport in some other route in the list. Find this other route, combine the two routes together, and add the combined route to the list of routes.

If you cannot find a second route that starts where the selected route ends, then try the reverse. Find a second route that ends where the selected route begins. Combine these two routes and add the combination to the list of routes.

For example, select `[ACV, PAO]` from the list. I would expect `PAO` to be the start of some other...the other route. Since `PAO` is the destination, we have to do the reverse. We look for a route that ends at `ACV` and find `[EUG, ACV]`. We combine these two route to make `[EUG, ACV, PAO]` and put it in the list.

Continue this process until there is only one route in the list. This is the entire flight. In the Palo Alto case, we end with `[EUG, ACV, PAO]`.

Location

If the user requests locations, then your program needs to know the location of each airport. This information is contained in a file named `ap_loc.txt`.

Each line contains the airport identifier, the latitude, and the longitude. For example, the following are the first few lines of this file:

```
0E0     40.7562277777778        -93.2403361111111
0E8     35.7176588888889        -108.201596111111
0E9     34.9781666666667        -106.000027777778
3E0     35.7139322222222        -100.603192222222
```

Define a new class called `Airport` with these three properties. Read each line of the file into a list of `Airports`.[2]

When you need to print out the location of an airport, search through the list until you find the airport you need. You could print the coordinates and name. A better approach is to define a `toString` method and just print the `Airport` object.

If the user does not request location information, then you do not need the `ap_loc.txt` file. Simply print the airport identifier held in the route list.

TestData

There are three test files on Canvas: N66709.txt, N808BS.txt, and RNDFLT.txt.

The first is a flight from St George, UT to Eugene, OR with just 7 hops. A couple of coworkers and I purchased the plane at auction and I got to fly it home. There are pictures on Google if you look up the tail number N66709.

The second flight is from Chesapeake Bay, VA. This was another airplane delivery with 21 hops. N808BS is a small, home-built airplane. There are also pictures of this plane online.

The last file is a random flight that consists of almost all of the airports in the United States. This is a stress-test for your program. Do not try to plot this flight on the mapping software. It takes a really long time and looks like scribbles across the map.

[2] Airport information should be stored in an associative container, but we have not developed those kinds of containers. This is what you will develop in CSCI-241.

Coding Standards

Your program should follow the same standards described in Lab 1. For this assignment, the standard: "Use comments at the start of each method to describe the purpose of the method and the purpose of each parameter to the method" is particularly important. Your program will be graded on conformance to the coding standards as well as correct functionality.

Deliverables

Create a `zip` file with the `java` files that make up your program including a linked list test program. At the very least, you should have `Flight.java`, `LinkedList.java`, and `LLTest.java`. Include any other files as necessary.

Grading

- Implement your own linked list class.
  - (20 pts) Complete linked list implementation.
  - (15 pts) Generic linked list (e.g., `LinkedList<T>`).
  - (10 pts) An `append` method that safely combines two lists.
  - (5 pts) Using both `front` and `back` reference. That is, $O(1)$ append.
- (10 pts) Printing airport location.
- (10 pts) Separate program logic correctly. Do not include non-list functionality in your linked list code. Similarly, do not include list logic in the main program.
- (10 pts) A comprehensive test suite for the linked list class. You are free to start with the test program from last week's lab, but you will probably need to add additional test cases for the new list functionality.
- (20 pts) Coding style like organization, function and variable names, etc.

Outcomes

- Basic understanding of the use of recursion in the definition and implementation of linear data structures, including stacks, lists and queues.
- Basic understanding of the concepts of abstract data types and generics and their implementation in a modern programming language.
- Basic understanding of access or pointer types and dynamic memory allocation, as implemented in a modern programming language.
- The use of a modern programming language for the implementation of abstract data types for linear data structures in the solution of problems.
- The ability to create test cases for problems involving linear data structures.