



Project #04: Divvy Trip Analysis

Complete By: Tuesday, March 14th @ 11:59pm

Assignment: C program to perform Divvy Trip analysis

Policy: Individual work only, late work **is** accepted (see “Policy” section on last page for more details)

Submission: online via zyLabs (“Project04 Divvy Analysis”, section 4.26)

Assignment

Divvy is a well-known bike sharing system in Chicago. The assignment is to input Divvy station and trip data, and then interact with the user to support the following six commands:

1. Statistics about the AVL trees
2. Station info
3. Trip info
4. Bike info
5. Find stations nearby
6. Route analysis

The data will come from 2 input files, both in CSV format (Comma-Separated Values). Your job is to write a C program to input this data, organize it into 3 AVL trees, and perform the requested analyses / output. You’re also required to free all memory allocated by the program --- strings, tree nodes, tree handles, etc.

Here’s a screenshot for the input files “stations.csv” and “trips.csv” --- the user input, with corresponding output, is highlighted in red. The user inputs the filenames (stations first), and then supplies one or more commands to the program. Your program must match this output exactly, since the first level of grading will use the zyLabs system.

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window shows the output of a C program. The output is color-coded: green for program output and red for user input. The output includes a welcome message, statistics for stations, trips, and bikes, and detailed information for a specific station (Station 90) and trip (Trip 10422254). The user input, which includes the filenames "stations.csv" and "trips.csv", and several commands, is highlighted with red boxes. The output matches the user input and provides detailed information for each command.

```
C:\Windows\system32\cmd.exe
** Welcome to Divvy Route Analysis **
stations.csv
trips.csv
** Ready **
stats
** Trees:
  Stations: count = 581, height = 10
  Trips:    count = 1520, height = 11
  Bikes:    count = 1186, height = 12
station 90
**Station 90:
  Name: 'Millennium Park'
  Location: (41.881032,-87.624084)
  Capacity: 35
  Trip count: 35
trip 10422254
**Trip 10422254:
  Bike: 2179
  From: 90
  To: 7
  Duration: 6 min, 57 secs
bike 2179
**Bike 2179:
  Trip count: 2
find 41.8810 -87.6240 0.25
Station 90: distance 0.004865 miles
Station 197: distance 0.097514 miles
Station 39: distance 0.136850 miles
Station 43: distance 0.202864 miles
Station 284: distance 0.217922 miles
route 10422254 0.25
** Route: from station #90 to station #7
** Trip count: 2
** Percentage: 0.131579%
exit
** Freeing memory **
** Done **
```

Input Files

The input files are text files in **CSV** format — i.e. comma-separated values. The first file defines the set of **stations**, in no particular order. An example of the format is as follows:

```
id,name,latitude,longitude,dpcapacity,online_date
456,2112 W Peterson Ave,41.991178,-87.683593,15,5/12/2015
101,63rd St Beach,41.78101637,-87.57611976,23,4/20/2015
109,900 W Harrison St,41.874675,-87.650019,19,8/6/2013
.
.
.
```

The first line contains column headers, and should be ignored. The data starts on line 2, and each data line represents one Divvy station for docking/undocking bicycles. Each station contains 6 values: a unique station ID (integer), the stations name (string), the position of the station in (longitude, latitude) format, the capacity of the station (integer, # of bikes you can dock there), and the date the station came online (string). Do not assume the station IDs are contiguous 1..N, they are not.

The second file defines a set of **trips**, in no particular order. With Divvy, a “bike trip” means a rider checks out a bike from one station, rides it around the city, and then checks it back into the same station, or a different station. The main purpose of Divvy is commuting, so most bike rides are short — less than 30 minutes. Here’s an example of the file format (you may need to open one of the data files to see this better):

```
trip_id,starttime,stoptime,bikeid,tripduration,from_station_id,from_station_name,to_station_id,to_station_name,usertype,gender,birthyear
10426648,6/30/2016 23:57,7/1/2016 0:22,4050,1466,259,California Ave & ...,123,California Ave & ...,Subscriber,Female,1986
10426638,6/30/2016 23:55,7/1/2016 0:40,4579,2713,177,Theater on the Lake,340,Clark St & Wrightwood Ave,Customer,,
.
.
.
```

The first line contains column headers, and should be ignored. The data starts on line 2, and each data line represents one Divvy trip. Each data line consists of 12 values:

- Trip id: integer
- Start time: date and time of trip start
- Stop time: date and time of trip end
- Bike id: integer
- Trip duration: integer, in seconds
- From station id: integer, where bike was checked out
- From station name: string, where bike was checked out
- To station id: integer, where bike was returned
- To station name: string, where bike was returned
- User type: a registered “subscriber” or a new “customer”
- Gender: optional string, may be missing
- Birth year: optional integer, may be missing

You only need to store a subset of these values. In case you’re curious, the Divvy data is available for browsing on Chicago’s data [portal](#); the raw data was downloaded from Divvy @ <https://www.divvybikes.com/data>.

User commands

The user can input any number of commands; for simplicity assume the input is valid in the sense that if the user needs to input an integer or a real number, he/she will do so. The six commands are “stats”, “station”, “trip”, “bike”, “find”, and “route”. The user enters “exit” when he/she wishes to stop.

1. stats

Your program is required to input the station and trip data, and store this data into 3 AVL trees. The “stats” command outputs the # of nodes, and the height, of each tree. Example:

Where does the “bikes” tree come from? As you input the data about each trip, you’ll build a “bikes” tree to store info about each bike.

```
C:\Windows\system32\cmd.exe
** Welcome to Divvy Route Analysis **
stations.csv
trips.csv
** Ready **
stats
** Trees:
   Stations: count = 581, height = 10
   Trips:    count = 1520, height = 11
   Bikes:    count = 1186, height = 12
```

2. station id

3. trip id

4. bike id

These commands use the trees to lookup a station, trip, or bike, based on the id. The use of AVL trees is required. If not found, output “**not found”. Examples:

For each station, output the id, name, location in (latitude, longitude), capacity (the # of bikes that can be docked at this location), and the trip count. The trip count is the # of trips that originated, or ended, at this station. If a bike trip starts and ends at the same station, this counts as 2 trips.

For each trip, output the trip id, the bike id for the bike that was used, the “from” station id (where the trip originated), and the “to” station id (where the trip ended). Also output the duration of the trip in minutes and seconds.

For each bike, output the bike id, and the trip count --- the # of times the bike was used in a trip.

```
C:\Windows\system32\cmd.exe
** Welcome to Divvy Route Analysis **
stations.csv
trips.csv
** Ready **
station 424
**Station 424:
   Name: 'Museum of Science and Industry'
   Location: (41.791728,-87.583945)
   Capacity: 27
   Trip count: 1
trip 10406143
**Trip 10406143:
   Bike: 5116
   From: 117
   To: 257
   Duration: 10 min, 30 secs
bike 4199
**Bike 4199:
   Trip count: 1
station 22451
**not found
trip 123
**not found
bike 8
**not found
```

5. find latitude longitude distance

The “find” command finds Divvy stations that are nearby to the position input by the user. The user inputs their latitude, longitude, and distance they are willing to ride (in miles), and the program outputs the Divvy stations whose position is \leq that distance away.

Notice the stations are sorted and output by their distance from the entered position; if 2 stations are the same distance away, then output those stations in order by station id. [A function is provided in the given code to compute the distance between 2 points in (lat, long) format.]

```
C:\Windows\system32\cmd.exe
** Welcome to Divvy Route Analysis **
stations.csv
trips.csv
** Ready **
find 41.86 -87.62 0.5
Station 338: distance 0.168045 miles
Station 273: distance 0.278975 miles
Station 72: distance 0.300635 miles
Station 168: distance 0.340126 miles
Station 97: distance 0.383504 miles
Station 370: distance 0.404626 miles
Station 4: distance 0.429038 miles
Station 178: distance 0.454386 miles
```

6. route tripID distance

The “route” command performs an analysis to see how many trips are taken along a given route. The user enters a trip id, which defines a starting station **S** and a destination station **D**. The user also enters a distance, in miles.

Let **S'** be all stations that are \leq distance away from **S**, and let **D'** be all stations that are \leq distance away from **D**. Search the trip data and count all trips that start from a station in **S'**, and end at a station in **D'**. Then compute the overall percentage this count represents, i.e. (trip count / total # of trips) * 100.

```
C:\Windows\system32\cmd.exe
** Welcome to Divvy Route Analysis **
stations.csv
trips.csv
** Ready **
route 10422254 0.5
** Route: from station #90 to station #7
** Trip count: 37
** Percentage: 2.434211%
```

Requirements

You are required to use AVL trees to store the data about stations, trips, and bikes. You also need to be efficient in the face of large data files --- this implies the need to use the trees as much as possible for fast lookup, and don't try to pre-compute all possible output combinations. And unlike previous projects, you also need to free all memory: strings you malloc, tree nodes, tree handles, files, etc. You should run the **valgrind** tool to ensure you have returned all memory; we'll run valgrind when we grade your program.

Since we will be using trees to store 3 different types of data --- stations, trips, and bikes --- this raises the question of how to define **AVLKey** and **AVLValue**:

```
typedef ??? AVLKey;
typedef ??? AVLValue;
```

The key is easy: use an integer since all three trees will be ordered by an ID, either station, trip, or bike:

```
typedef int AVLKey;
```

The value is harder, since different data needs to be stored in each case. For example, stations have a **name**

and a **location**, while trips have **from** and **to** station ids. And bikes have only an **id** and a trip **count**. The solution to this problem in C is to use a **union**¹. A union allows you to define a single structure that can store *either* data about one station, one trip, or one bike. Here are the definitions from the provided “avl.h”. First we define the data about each STATION, TRIP, and BIKE. Then we define **AVLValue** to be the union of these:

```
typedef struct STATION
{
    int  StationID;
    int  Data;
} STATION;

typedef struct TRIP
{
    int  TripID;
    int  Data;
} TRIP;

typedef struct BIKE
{
    int  BikeID;
    int  Data;
} BIKE;

enum UNIONTYPE
{
    STATIONTYPE,
    TRIPTYPE,
    BIKETYPE
};

typedef struct AVLValue
{
    enum UNIONTYPE Type;  // Station, Trip, or Bike:

    union
    {
        STATION  Station;  // union => only ONE of these is stored:
        TRIP      Trip;
        BIKE      Bike;
    };
} AVLValue;
```

The idea is that you'll modify the STATION, TRIP, and BIKE structs to store the data needed by the program, and then use the AVLValue struct to store these values into the corresponding trees. For example, suppose we create a tree to store stations:

```
AVL *stations = AVLCreate();
```

¹ https://www.tutorialspoint.com/cprogramming/c_unions.htm

Now let's insert the (key, value) pair (123, 456), where 123 is the station id and 456 is a data value. We declare an **AVLValue**, set the **Type** to denote that this is a STATION, and then we fill the **Station** component of the union with the id and data:

```
AVLValue value;

value.Type = STATIONTYPE; // union holds a station:
value.Station.StationID = 123;
value.Station.Data       = 456;

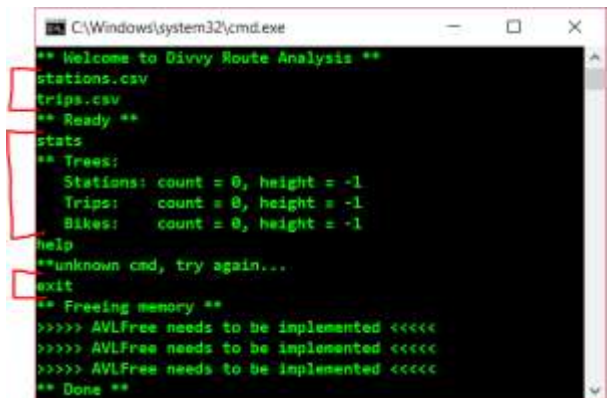
int success = AVLInsert(stations, value.Station.StationID, value);
```

You'll use similar techniques to store TRIP and BIKE data into the corresponding "trips" and "bikes" trees. The provided "main.c" file contains an example of using AVLValue in each case.

Getting Started

The course web page contains a set of files to help you get started. Look under "Projects", and then "[project04-files](#)". You'll find the following files:

1. main.c
2. avl.h
3. avl.c
4. stations.csv
5. stations-big.csv
6. trips.csv
7. trips-2016-q1.csv



```
C:\Windows\system32\cmd.exe
** Welcome to Divvy Route Analysis **
stations.csv
trips.csv
** Ready **
stats
** Trees:
  Stations: count = 0, height = -1
  Trips:    count = 0, height = -1
  Bikes:    count = 0, height = -1
help
**unknown cmd, try again...
exit
** Freeing memory **
>>>> AVLFree needs to be implemented <<<<
>>>> AVLFree needs to be implemented <<<<
>>>> AVLFree needs to be implemented <<<<
** Done **
```

As given the program compiles and accepts commands from the user. However, the source files are minimal; you'll need to add your own AVLSearch and AVLInsert functions, as well as functions to parse the input files and build the trees. Note that solutions to earlier projects are available on the course web page, which you are free to use in this assignment; a solution to AVLInsert is also provided under "Homeworks", "[hw13-solution](#)" --- you are free to build upon this as well.

Note that you need to use the provided "avl.h" and "avl.c" files, since they defined the necessary typedefs for the project. Then copy-paste your code for AVLSearch, AVLInsert, etc. Also, if you are using gcc, you'll need to link with the math library to gain access to the trig functions used in "main.c":

```
gcc -std=c99 -Wall main.c avl.c -o project04 -lm
```

This is not necessary in Visual Studio, not sure about other environments such as CodeBlocks or DevC++.

Efficiency

Once your program is working, you should run against the larger input files to make sure your solution does not contain any hidden inefficiencies. Example:

```
C:\Windows\system32\cmd.exe
** Welcome to Divvy Route Analysis **
stations-big.csv
trips-2016-q1.csv
** Ready **
stats
** Trees:
    Stations: count = 15581, height = 16
    Trips:     count = 396912, height = 18
    Bikes:     count = 4318, height = 14
station 424
**Station 424:
    Name: 'Museum of Science and Industry'
    Location: (41.791728,-87.583945)
    Capacity: 27
    Trip count: 624
trip 8918180
**Trip 8918180:
    Bike: 1544
    From: 283
    To: 81
    Duration: 28 min, 44 secs
bike 1544
**Bike 1544:
    Trip count: 129
find 41.86 -87.62 0.25
Station 87135: distance 0.026547 miles
Station 94825: distance 0.076816 miles
Station 14698: distance 0.090135 miles
Station 84797: distance 0.091292 miles
Station 43107: distance 0.105060 miles
Station 27077: distance 0.140796 miles
Station 97741: distance 0.140828 miles
Station 82762: distance 0.159596 miles
Station 338: distance 0.168045 miles
Station 71581: distance 0.171961 miles
Station 39186: distance 0.182673 miles
Station 75120: distance 0.193254 miles
Station 99874: distance 0.200576 miles
Station 38298: distance 0.207673 miles
Station 84429: distance 0.213441 miles
Station 51670: distance 0.215526 miles
Station 73669: distance 0.220165 miles
Station 7457: distance 0.243802 miles
Station 1784: distance 0.247200 miles
route 8918180 0.25
** Route: from station #283 to station #81
** Trip count: 1544
** Percentage: 0.389003%
exit
** Freeing memory **
** Done **
```


Submission

The program is to be submitted via zyLabs: see “**Project04 Divvy Analysis**”, section 4.26. Submit your three files in the tabs provided: `main.c`, `avl.h`, and `avl.c`. When you’re ready to submit, copy-paste your program into the zyLabs editor pane, switch to “Submit” mode, and click “SUBMIT FOR GRADING”.

Keep in mind this is only a first level grading mechanism to ensure you have (a) submitted the correct files, and (b) are following the basic requirements and passing some simple tests. The grade you receive from your zyLabs submission is only a preliminary grade. After the deadline, the TAs will then manually review and test each program for the following:

- adherence to requirements
- overall design
- correctness against additional tests (involving other input files)
- commenting, indentation, whitespace and readability

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 25%. After 24 hours, no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is described here:

<http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf> .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at <http://www.uic.edu/depts/dos/studentconductprocess.shtml> .