Project #06: Part 2 of 2: hashing DIVVY data

Complete By: **On-time:** Sunday 4/12 @ 11:59pm
Late: Monday 4/13 @ 11:59pm (-10%)

Assignment: DIVVY hashing program

Policy: Individual work only, late work *is* accepted

Submission: all .cpp and .h files via Gradescope; the first 12 submissions are free, each additional submission 1 pt

Overview

In part 1 you built a **hashmap** class for storing (key, value) pairs in a hash table. Here in part 2 we're going to use this class to store and analyze data from the **DIVVY** bike sharing company.

Your program is going to input 2 types of data: stations in the DIVVY system, and bike trips that riders have taken on a DIVVY bike. Your program needs to lookup stations by their station id (integer) or their station abbreviation (string). This implies you'll need two hashmaps, one to lookup by id and another to lookup by string. Your program will also need another hashmap to lookup trips, and another hashmap to lookup bikes. In total, your program will have at least 4 hashmap variables:

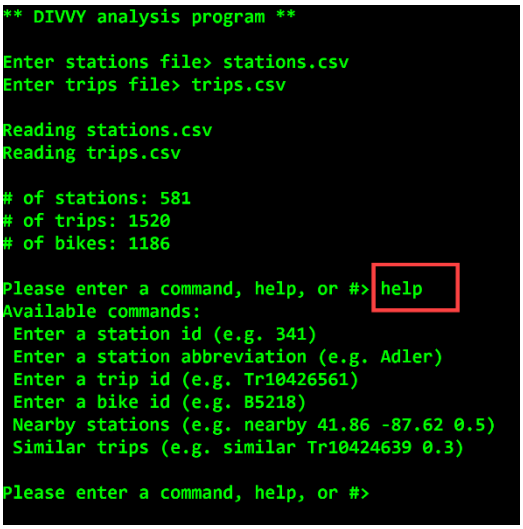
```
int main()
{
   hashmap<?, ?>  stationsById(?);
   hashmap<?, ?>  stationsByAbbrev(?);
   hashmap<?, ?>  tripsById(?);
   hashmap<?, ?>  bikesById(?);
```

```
** DIVVY analysis program **

Enter stations file> stations.csv
Enter trips file> trips.csv

Reading stations.csv
Reading trips.csv

# of stations: 581
# of trips: 1520
# of bikes: 1186

Please enter a command, help, or #>
```

It's very important to separate the abstraction --- hashmap --- from the # of instances that you need to store all the data. The hashmap class implements a hash table. This one class is used to create 4 different hash tables to store the data needed by the program. Note that you'll also need a separate hash function to work with each hashmap instance: HashByStationID, HashByStationAbbrev, etc. You cannot use the built-in hash functions.

Assignment

The program inputs data from two files, whose filenames are entered by the user. The 1st file contains stations data, and the 2nd file contains trip data. The program then offers a choice of 7 commands, and repeats until the user enters #. The screenshot to the right shows the output from the "help" command ----------> The 6 commands are summarized as follows:



1. Lookup station by id
2. Lookup station by abbreviation (e.g. "Adler")
3. Lookup a bike trip by trip id (e.g. "Tr10426561")
4. Lookup bike usage by bike id (e.g. "B5218")
5. Find nearby stations based on latitude / longitude
6. Find similar trips based on latitude / longitude

A set of input files are provided for testing purposes: stations.csv, trips.csv, and trips-2016-q1.csv. To help you with finding nearby stations based on latitude and longitude, a function **distBetween2Points(lat1, long1, lat2, long2)** is provided in the file "util.cpp". The corresponding header file "util.h" is also provided. You can find these files on the course dropbox under Projects, project06-part02-files. If you are working on Codio, use the File menu, Upload command to move these files to your Codio project. Note that in order to call the provided function from main.cpp, you'll need to #include "util.h". Then you'll need to modify your makefile to compile "util.cpp" as part of the program:

```
build:
   rm -f program.exe
   g++ -O2 -std=c++11 -Wall main.cpp hash.cpp util.cpp -o program.exe
```

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. Here are the first few lines from the provided "stations.csv" file:

```
id,abbrev,fullname,latitude,longitude,capacity,online_date
456,2112 W Peterson,2112 W Peterson Ave,41.991178,-87.683593,15,5/12/2015
101,63rd Beach,63rd St Beach,41.78101637,-87.57611976,23,4/20/2015
109,900 W Harrison,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 7 values: a unique integer ID, a unique abbreviated name, the stations's fullname, latitude, longitude, the capacity of the station (i.e. the # of bikes you can dock there), and the date the station came online in the DIVVY system. Assume the data is valid and properly formatted, however do not assume the station IDs are contiguous 1..N, they are not. For

hashing purposes, use the station's ID and abbreviated name, these are guaranteed to uniquely identify each station.

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 are the first few lines from the provided "trips.csv" file:

```
tripid,starttime,stoptime,bikeid,duration,from,to,identifies,birthyear
Tr10426561,6/30/2016 23:35,7/1/2016 0:02,B5229,1620,329,307,Male,1968
Tr10426434,6/30/2016 23:09,6/30/2016 23:35,B5218,1547,228,253,Female,1988
Tr10426287,6/30/2016 22:48,6/30/2016 23:13,B4199,1521,145,35,,
Tr10426168,6/30/2016 22:31,6/30/2016 22:42,B2659,668,113,69,Male,1986
.
.
.
```

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 9 values:

- Trip id: string consisting of "Tr" following by 1 or more digits
- Start time: date and time of trip start
- Stop time: data and time of trip end
- Bike id: string consisting of "B" followed by 1 or more digits
- Trip duration: integer, in seconds
- From station id: integer, station where bike was checked out
- To station id: integer, station where bike was returned
- Identifies: gender the rider identifies as (string, optional)
- Birth year: the rider's year of birth (integer, optional)

Assume the data is valid and properly formatted. This implies you may assume the station ids --- the **from** and **to** station ids --- exist in the provided stations file. In case you're curious, the DIVVY data is available for browsing on Chicago's data portal; the raw data is from DIVVY @ https://www.divvybikes.com/data.

With hashing, it's important to know how much data to expect, so that appropriate-sized hash tables can be allocated. We're going to assume a load factor of 5, meaning the hash tables will be 5x larger than necessary to reduce collisions (i.e. 20% full, 80% empty). Expect at most 2,000 stations, 10,000 bikes, and 500,000 trips. This implies your hashmaps should have the following sizes:

```
int main()
{
   hashmap<?, ?>  stationsById(10000);      // 10K
   hashmap<?, ?>  stationsByAbbrev(10000);  // 10K
   hashmap<?, ?>  tripsById(2500000);       // 2.5M
   hashmap<?, ?>  bikesById(50000);         // 50K
```

Command #1: lookup station by id

If the user inputs a numeric string, assume it's a station id and search for the station by this id. If found, output the station information, otherwise output "Station not found".

Command #2: lookup station by abbreviation

If the user inputs a string that doesn't match any of the other commands, assume it's an abbreviated station name and search for the station by this abbreviation. If found, output the station information, otherwise output "Station not found".

```
Please enter a command, help, or #> 451
Station:
 ID: 451
 Abbrev: Sheridan & Loyola
 Fullname: Sheridan Rd & Loyola Ave
 Location: (42.001, -87.6612)
 Capacity: 15
 Online date: 5/15/2015

Please enter a command, help, or #> 9125
Station not found

Please enter a command, help, or #> Sheridan & Loyola
Station:
 ID: 451
 Abbrev: Sheridan & Loyola
 Fullname: Sheridan Rd & Loyola Ave
 Location: (42.001, -87.6612)
 Capacity: 15
 Online date: 5/15/2015

Please enter a command, help, or #> fred
Station not found

Please enter a command, help, or #>
```

An interesting design decision is how do you store the data. Since you'll have 2 hashmaps, one by id and one by abbreviation, do you store the station data twice? That seems wasteful. What's are some alternative? What's the best approach that saves memory yet doesn't overly complicate the programming?

Command #3: lookup trip by tripid

If the user inputs a string starting with "Tr" and followed by 1 or more digits, assume they have entered a tripid. Search for the trip by this id, and output the trip information if found. If not, output "Trip not found".

Notice the duration is converted to hours, minutes and seconds. Only output the hours if the hours are > 0; likewise, only output the minutes if the minutes are > 0. If the case of 1 hour, or 1 minute, or 1 second, the program outputs (incorrectly) 1 hours, or 1 minutes, or 1 seconds. You don't have to worry about the if statement to correct for this special case.

Also notice that not all riders choose to supply their birthyear, or their identifying gender. In this case output nothing.

```
Please enter a command, help, or #> Tr10426561
Trip:
 ID: Tr10426561
 Starttime: 6/30/2016 23:35
 Bikeid: B5229
 Duration: 27 minutes, 0 seconds
 From station: Lake Shore & Diversey (329)
 To station: Southport & Clybourn (307)
 Rider identifies as: Male
 Birthyear: 1968

Please enter a command, help, or #> Tr11
Trip not found

Please enter a command, help, or #> Tr10426287
Trip:
 ID: Tr10426287
 Starttime: 6/30/2016 22:48
 Bikeid: B4199
 Duration: 25 minutes, 21 seconds
 From station: Mies van der Rohe & Chestnut (145)
 To station: Streeter & Grand (35)
 Rider identifies as:
 Birthyear:

Please enter a command, help, or #>
```

Command #4: lookup bike usage by bikeid

If the user inputs a string starting with "B" and followed by 1 or more digits, assume they have entered a bikeid. Search for a bike by this id, and output the # of trips that involved this bike. If the user enters a bikeid that doesn't exist (i.e. was never used in any trip), output "Bike not found".

You could compute this on demand by searching through all the trips, but this linear search is expensive on real trip files (that contain 100,000+ trips). Instead, as you input the trip data, build a hashmap of each bike and accumulate the # of trips in which it was used. That way you can answer the query in O(1) time instead of O(N) time.

```
Please enter a command, help, or #> B55
Bike:
 ID: B55
 Usage: 1

Please enter a command, help, or #> B1658
Bike:
 ID: B1658
 Usage: 4

Please enter a command, help, or #> B123
Bike not found

Please enter a command, help, or #>
```

Command #5: find nearby stations based on lat / long

You are no doubt familiar with finding things that are nearby. This is easily done with a phone based on its **latitude** (Y coordinate) and **longitude** (X coordinate). Since we aren't programming on a phone, we'll have to manually input the coordinates. The command searches *all* stations for those that are <= the specified distance, which is in given in miles. If no stations are found, then "none found" is output. Otherwise, the stations are output in sorted order, closest first. If 2 stations are the same distance away, then output in order by station ID. Feel free to use the built-in std::sort function.

```
Please enter a command, help, or #> nearby 41.86 -87.62 0.5
Stations within 0.5 miles of (41.86, -87.62)
 station 338: 0.168045 miles
 station 273: 0.278975 miles
 station 72: 0.300635 miles
 station 168: 0.340126 miles
 station 97: 0.383504 miles
 station 370: 0.404626 miles
 station 4: 0.429038 miles
 station 178: 0.454386 miles

Please enter a command, help, or #> nearby 41.86 -87.62 0.1
Stations within 0.1 miles of (41.86, -87.62)
 none found

Please enter a command, help, or #> nearby 41.8810 -87.6240 0.25
Stations within 0.25 miles of (41.8810, -87.6240)
 station 90: 0.00486477 miles
 station 197: 0.0975142 miles
 station 39: 0.13685 miles
 station 43: 0.202864 miles
 station 284: 0.217922 miles

Please enter a command, help, or #>
```

For simplicity, you may assume valid user input with a single space between each value. You can use the C++ stringstream object (much like parsing a CSV file) to obtain the individual values in the input.

As noted earlier, a function **distBetween2Points(lat1, long1, lat2, long2)** is provided to compute the distance between two points (lat1, long1) and (lat2, long2). See the top of page 2 for a discussion of how to use this function in your program. It turns out that you may get slightly different results based on the order in which you pass the coordinates to this function. To match the screenshot above, the coordinates entered by the user should be the first coordinate (lat1, long1) passed to the function.

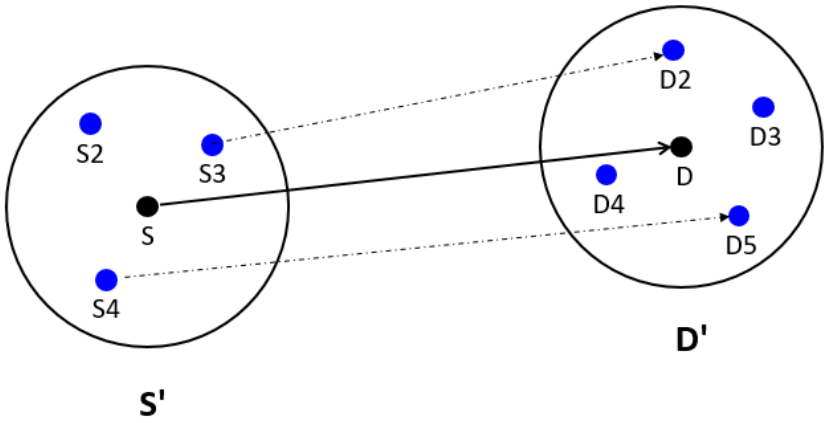
This also presents an interesting design question, because to implement this command you need to search *all* the stations in the hashmap. But the given hashmap class does not provide a mechanism for doing this;

you need a station's ID or abbreviation to find its data. Think about how to extend the hashmap class, in the easiest way possible, to enable the main program to access all the stations. What you cannot do is simply make the underlying hash table public, that's too easy and goes against the notions of abstraction that we have been trying to build upon. And you cannot move the "search for nearby stations" into the hashmap class, because the hashmap is meant to be general-purpose --- the class is being used for a variety of different purposes, so a station-specific search function makes no sense in a general-purpose templated class. You also cannot copy-paste the hashmap class to make one that is station-specific.

Instead, think about ways to provide general access to the elements of a hashmap. With trees, we added **begin()** and **next()** functions to walk through the tree and provide access; maybe something similar could be done here? Or perhaps you could implement an iterator class as we discussed in class, and allow foreach-like iteration through the hashmap? Or in the simplest form, how about a function that returns a vector of all the keys in the hashmap? One could then iterate through the vector and call search() to obtain the values.

Command #6: find similar trips based on lat / long

Given a trip from station S to station D, this command searches and counts how many trips are similar to this one --- i.e. start at S or a station nearby S *and* finish at destination D or a station nearby D. Visually, we are counting the # of trips that fall within the circles below:



One possible algorithm is to find the set of stations S' that include D and those nearby S. Likewise, find the set of stations D' that include D and those nearby D. Now search all the trips and find those that start from a station in S' and end with a station in D'. Based on your choice of data structures, it should be possible to implement this algorithm in time O(S + T), where S is the # of stations and T is the # of trips. Feel free to use the built-in set or map data structures, or perhaps your hashmap. [NOTE: if you end up with doubly or triply-nested loops to solve this, you are on the wrong track.]

```
Please enter a command, help, or #> similar Tr10424639 0.3
Trips that follow a similar path (+/-0.3 miles) as Tr10424639
 nearby starting points: 36 40 50 75 89 192 283 286 287
 nearby ending points: 39 43 44 49 52 81 90 197
 trip count: 10

Please enter a command, help, or #> similar Tr10426561 0.5
Trips that follow a similar path (+/-0.5 miles) as Tr10426561
 nearby starting points: 34 157 220 313 324 329 340
 nearby ending points: 58 87 188 223 307
 trip count: 3

Please enter a command, help, or #>
```

For simplicity, you may assume valid user input with a single space between each value. You can use the C++ stringstream object (much like parsing a CSV file) to obtain the individual values in the input. It is possible for the user to input a trip that doesn't exist, in this case output "no such trip".

```
Please enter a command, help, or #> similar Tr123 0.5
Trips that follow a similar path (+/-0.5 miles) as Tr123
 no such trip

Please enter a command, help, or #> similar Tr10424021 0.4
Trips that follow a similar path (+/-0.4 miles) as Tr10424021
 nearby starting points: 126 180 268 301
 nearby ending points: 33 37 38 39 43 44 49 51 52 81 90 98 194 195 197 264 284
 trip count: 5

Please enter a command, help, or #>
```

To match the output shown in the screenshot, the coordinates from the stations involved in the specified trip --- the coordinates of S and D --- should be the first coordinates (lat1, long1) passed to the **distBetween2Points()** function.

Efficiency

To test the efficiency of some of your code, a bigger trips file is provided in "trips-2016-q1.csv".

Given the amount of data, you may want to compile your program with optimization enabled --- this can dramatically reduce execution time (from minutes to seconds in some cases). To optimize with g++, add the option -O2 to your makefile:

g++ -O2 -std=c++11 main.cpp …

If you are working in Visual Studio, switch from "Debug" to "Release" mode using the toolbar (and then run without debugging).

```
** DIVVY analysis program **

Enter stations file> stations.csv
Enter trips file> trips-2016-q1.csv

Reading stations.csv
Reading trips-2016-q1.csv

# of stations: 581
# of trips: 396912
# of bikes: 4318

Please enter a command, help, or #> B1544
Bike:
 ID: B1544
 Usage: 129

Please enter a command, help, or #> Tr9080551
Trip:
 ID: Tr9080551
 Starttime: 3/31/2016 23:53
 Bikeid: B155
 Duration: 14 minutes, 1 seconds
 From station: Ravenswood & Lawrence (344)
 To station: Broadway & Thorndale (458)
 Rider identifies as: Male
 Birthyear: 1986

Please enter a command, help, or #> similar Tr8918180 0.25
Trips that follow a similar path (+/-0.25 miles) as Tr8918180
 nearby starting points: 33 36 37 40 50 89 283 286 287
 nearby ending points: 38 44 49 51 81 98
 trip count: 1544

Please enter a command, help, or #>
```

Programming notes

Since the commands may contain multiple words, use the **getline()** function for all console input. Also use getline() to input the filenames initially. Here's a nice trick to avoid having to type the filenames over and over again when testing (instead you can just press enter when prompted):

```
string stationsFilename = "stations.csv";
string tripsFilename = "trips.csv";
string filename;

cout << "Enter stations file> ";
getline(cin, filename);  // user can type a filename, or just press ENTER

if (filename != "")      // user entered a filename, use it
   stationsFilename = filename;
```

What's the best strategy for success? Build and test in stages.

1. Finish part 1 with a working hashmap that you have tested.
2. Start to build main(), and input the stations data. Do some searches and make sure it's working, both by id and abbreviation. Compare results against the input file.
3. Now input the trips data, do some searches, and compare again the input file.
4. Extend the trip input code to accumulate the bike usage stats.
5. Add each command one by one, in the order shown.

The Gradescope submission scripts will grade the program based on the # of commands you have working, so you don't need to complete the entire program to obtain a passing score.

Requirements

1. The hashmap class must use hashing, and probing. You must use this class to store the data from the stations and trips input files, and compute the bike usages. You are free to extend the hashmap class from part 1. However, the hashmap class must remain general-purpose. Do *not* add problem-specific data or functions to the class --- no DIVVY-related structures or functions may be added to the hashmap class (this is discussed further on pages 5-6). Also, do not simply make everything in the class public so the main() program can gain access to the underlying hash table directly.
2. There can be only one hashmap class. Do not copy-paste and create multiple hashmap classes (one for stations, one for trips, one for bikes). You don't need to do this, that's why the class is templated.
3. You can use the built-in vector, map and set abstractions to help implement the main program, but not the unordered_map or its variants. However, the storing of all DIVVY data --- bikes,

stations, and trips --- must be done using hashmap. You'll need to do some sorting, feel free to use the built-in sort algorithm.

4. You cannot use the built-in hash functions, write your own hash functions to use with your hashmap class. Since you will have 4 hashmaps, you'll need to write 4 hash functions.
5. You can only open the input files, and read the data at most once; additional operations involving the data must involve memory-based data structures.
6. You are required to free all memory allocated by your class. Valgrind will be used to confirm that memory has been properly freed. Note that Codio may have a version of valgrind installed that reports a memory leak; see Piazza post @1707 for more details.
7. No global variables.
8. If any of the requirements are broken, we reserve the right to score the entire program as a 0.

Grading, electronic submission, and Gradescope

Submission: all .cpp and .h files required to compile your program

Your score on this project is based on two factors: (1) correctness as determined by Gradescope, and (2) manual review of program files for commenting, style, and approach (e.g. adherence to requirements). Since part 2 is more complex than part 1, it is worth 100 points for correctness, and 40 points for manual review of commenting, style and overall approach / efficiency. In this class we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness. Example: if you submit to Gradescope and your score is a reported as a 0, then that's your correctness score. The only way to raise your correctness score is to re-submit.

In this project, your program will be allowed **12 free submissions**. After 12 submissions, each additional submission will cost 1 point. Example: suppose you score 100 after 15 submissions, and you activate this submission for grading. Your autograder score will be 97: 100 – 3 extra submissions. Note that you cannot use another student's account to test your work; this is considered academic misconduct because you have given your code to another student for submission on their account.

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), 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 available here:

https://dos.uic.edu/conductforstudents.shtml .

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. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .