CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 10
Date: 12th Nov, 2020

Important Instructions

1. **List of Input Files:** input.txt
2. **Output Files to be submitted:** ROLLNO_A10_P1.c/.cpp, ROLLNO_A10_P2.c/.cpp
3. Files must be read using file handling APIs of C/C++. Reading through input redirection isn't allowed.
4. You are to **stick to the file input output formats strictly** as per the instructions.
5. Submission through **.zip files are not allowed**.
6. Write your name and roll number at the beginning of your program.
7. Do not use any global variable unless you are explicitly instructed so.
8. Use proper indentation in your code.
9. Please follow all the guidelines. Failing to do so will cause you to lose marks.
10. **There will be part marking.**

Hashing

Problem Statement

X is a certain company which operates in stockmarket. They enter huge number of financial transaction - call trades - every day. On the other side of each trade, there is some company Y, call it a counterparty of X. To each trade of X is assigned a portfolio number, which is not a unique identifier of the trade, and is not even unique per counterparty. In other words, trades with different counterparties may be assigned the same portfolio, and different trades with the same counterparty may be assigned distinct portfolios as well, according to some internal policy of the company. Whenever X enters into a trade, a new line such as "+ 101 26" is appended to a log file, where "+" indicates that a new trade was initiated by X, "101" is the counterparty id, and "26" is the portfolio of that trade. At any time during a typical day, however, a counterparty Y may withdraw all transactions with X. From X's standpoint, that means all trades it had entered into with Y that day are now considered void, and a line such as "- 101" is logged. Here, the "-" sign indicates that a cancelation took place, and "101" is the id of the counterparty the cancelation refers to. Table 1 illustrates the basic structure of the log file of X.

By the end of each day, the company X needs to determine the set of distinct portfolios associated to trades that are still active by then, that is, the portfolios of trades that have not been canceled during the day. You are required to find an efficient solution to this using Hashing and print the hashtables.

NOTE:

1. The hashing function is a simple mod function (K%*size*), where K is the input key and *size* is the *size* of hash table (aka hash map).

+	101	26
+	2	25
+	101	25
+	3005	4550
-	101	
+	3005	26
+	4	184
+	101	4550
-	2	

Table 1: Sample log file.

2. Collisions will be handled by separate chaining.
3. Use the following definition to define the hash:

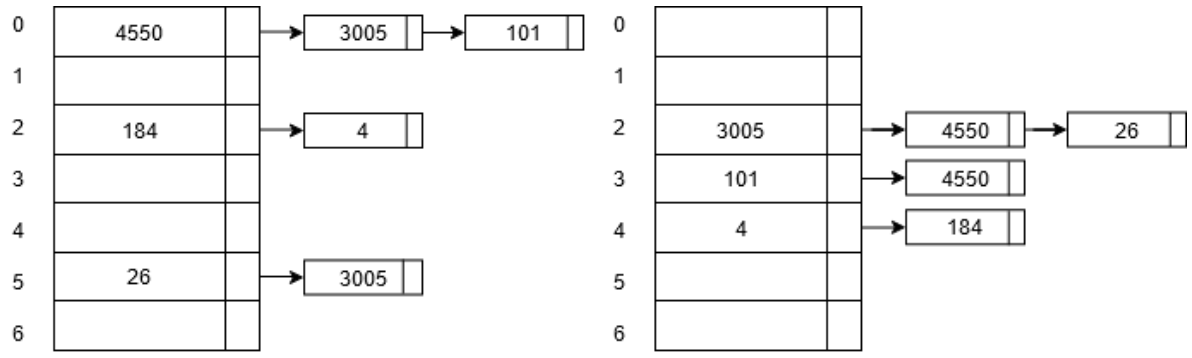
```
typedef struct _hashing {
   int key;
   struct _hashing *next;
} hash;
```

Part 1: Two mirroring hash maps

In the first part, you will have to implement two hash maps: (i) The first one will hold a counterparties-by-portfolio map: the keys are portfolio numbers, and the value stored with each key P is a list containing the counterparties Y of X for which there are active negotiations under that portfolio P. (ii) The second is a portfolios-by-counterparty hash map: the keys are the counterparties of X, and the values associated with each counterparty Y is a list containing the portfolios assigned to active negotiations with that partner.

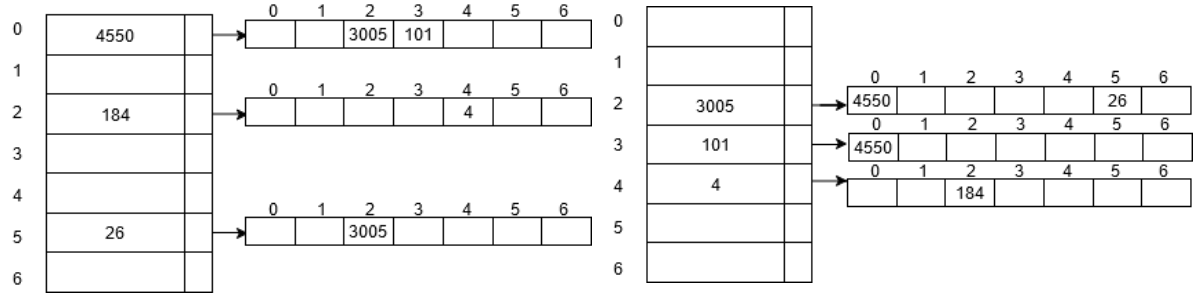
The file is traversed top-down. Each line with a "+" sign, be it "+ Y P", triggers two updates in our data structure: the first is the inclusion of Y in the list of counterparties associated with portfolio P in the counterparties-by-portfolio hash map (the key P will be inserted for the first time if it is not already there); the second change is the inclusion of P in the list of portfolios associated with counterparty Y in the portfolios-by-counterparty hash map (the key Y will be first inserted for the first time if it is not already there).

Whenever a line with a "-" sign is read, be it "- Y", we must remove all occurrences of Y from both tables. In the second, portfolios-by-counterparty hash map, Y is the search key, hence the bucket associated to Y can be determined directly by the hash function, and the list of portfolios associated to Y can be retrieved (and deleted, along with Y) in average time O(1). In counterparties-by-portfolio, however, Y can belong to lists associated to several distinct portfolios. To avoid traversing the lists of all portfolios, we can use the information in the first, portfolios-by-counterparty map (sure enough, before deleting the key Y) so we know beforehand which portfolios will have Y on their counterparty lists. We may thus go directly to those lists and remove Y from them, without the need to go through the entire table. If the list of counterparties associated to a portfolio P becomes empty after the removal of Y, then we delete the portfolio P from that first hash map altogether. You will need to handle collisions using separate chaining.



(a) counterparties-by-portfolio map (b) portfolios-by-counterparty map

Figure 1: Two mirroring hash maps. Collision handling is done using seperate chaining.



(a) counterparties-by-portfolio map (b) portfolios-by-counterparty map

Figure 2: Hash map multi level example

Part 2 : Multi Level Hashing

So our problem is the time-consuming task of traversing whole lists to locate a single element. We therefore want to try and replace those lists with more performatic structures, hash sets! In other words, the value associated with each key P in the counterparties-by-portfolio hash map will be, instead of a list, a hash set whose keys are the ids of the counterparties associated with P (see Figure 2). The same goes for the second, portfolios-by-counter-party hash map, where portfolios will be stored in a hash set associated to each counterparty Y. Thus, counterparties and portfolios can be included and removed without the need to traverse possibly lengthy lists, but in average constant time instead. Since the cost of processing each line is now O(1) (due to a constant number of hash table dictionary operations being performed), the whole algorithm, implemented this way, runs in expected linear time in the size of the file.

To implement the hash set, you can re-use the hash map data structure. Here, you will only need to maintain the key values.

Sample Input File

FILE: *input-part1.txt*

```
7
9
+  55  23
+  45  56
+  78  67
+  55  78
-  45
+  56  23
+  45  67
+  78  679
-  78
```

The first line contains the size of the hash map. The second line contains the number of test cases T. Each of the next T lines contains '+' sign with two integers Y and P, or a '-' sign with single integer Y.

Sample Output File

FILE: *output-part1.txt*

```
p 0 -1 -1
p 1 78 55
p 2 23 56
p 2 23 55
p 3 -1 -1
p 4 67 45
p 5 -1 -1
p 6 -1 -1
c 0 56 23
c 1 -1 -1
c 2 -1 -1
c 3 45 67
c 4 -1 -1
c 5 -1 -1
c 6 55 78
c 6 55 23
```

Each of the lines in output file should contain the four parts. First, 'p/c' to denote if the hash map being printed is counterparties-by-portfolio (p) or portfolios-by-counter-party (c). Second will be the index position in the hash map. Third will be the key. If the key is empty, print '-1'. Fourth will be the values present in that key (list/ hash set). If multiple values are present for a key, print them in the order present. If no value is present, print '-1'.

Make sure to read from a file named *"input.txt"*. Create a sample input file like the one shown above to test your code. At the time of evaluation, the input data might be different from the one given here.

You need not to submit *"output.txt"*, but your code should write the output in the given format in a file named *"output.txt"*