COSC2430 GA2: Hashing, Trees

Due Date: 11/11/20 11:59PM

Introduction

You work the night shift at Marshalls and your job is to accept the nightly inventory truck and put the inventory out on the store floor for sale. This particular night you came into the back warehouse and were greeted with a total mess. You received a large inventory of shirts. All of the shirts are strung out in piles and the pages of the shipping manifest have been scattered all over the floor and you don't know which page corresponds to the piles of unfolded shirts laying before you. Some of the pages seem to contain items that weren't even delivered to this store! ***Your first task is to figure out which shipping manifest page corresponds to the T-shirt delivery sitting in front of you.*** You see a list of binary codes on each of the papers that you know are UPC codes for the items. You also happen to recall that this particular brand uses a technique called "Cuckoo Hashing" to catalogue their inventory…

Background on "Cuckoo Hashing"

Your manager recently explained to you this supplier's difficulties implementing a workable hashing scheme. In the beginning, the company employed regular hashing. Here's how it worked. First, one created a hashing function $h:\Sigma* \rightarrow [0\ldots n-1]$ from all strings to the integer range $0,1,..,n-1$. This company uses a program that takes the letters printed on the shirt (a string) as an input and outputs an integer between 0 and $n-1$. Then an empty hash table O of size n is allocated and for every string s in O(the Order of T-shirts), they set $O[h(s)]=s$. Thus, given a certain t-shirt string t, you only need to calculate h(t) and see if $O[h(t)] = t$, to determine if the shirt was in the Order. They liked this simple process, but they ran into problems when two shirts would occupy the same UPC in the Order. This phenomenon, called collision, happened fairly often.

Then they switched to a process called "Cuckoo Hashing". With this approach they use two hash functions h1 and h2. So each T-shirt maps to two UPC codes in the order. A T-shirt string t is now handled as follows: they compute both h1(t) and h2(t), and if O[h1(t)]=t, or O[h2(t)]=t, they know that t is in O. Here is how O is populated. First, they make an empty table O. Then they take all the shirts to be contained in the order and insert them one by one. If O[h1(t)] is free, they set O[h1(t)]=t. Otherwise if O[h2(t)] is free, they set O[h2(t)]=t. If both are occupied, they evict the shirt r in O[h1(t)] and set O[h1(t)]=t. Then they put r back into the table in its alternative (O[h2(r)]) place (and if that entry was already occupied, they evict that word and move it to its alternative place, and so on). Of course, they may end up in an infinite loop here, in which case they must rebuild the table with other choices of hash functions to determine the UPC.

Your Mission (Part A)

You are going to design a program that works with your scanner to scan in different pieces of paper that you found on the ground. You need to determine which piece of shipping manifest paper belongs to this order. Given your knowledge of their UPC system and their use of Cuckoo hashing, the program should pick which paper belongs to the order. You will know which is correct by assessing if the codes on the paper are "hashable" – that is, they won't result in an infinite loop with the Cuckoo Hashing scheme.

Input (Part A)

The first line of the input will have a number 1≤***p***≤50 representing the number of pages (hashing number sets) that will be fed into your scanner. Each paper(hashing number set) begins with two positive integers **m** and **n** 1≤m≤n≤1000 each on their own line, ***m*** telling the number of t-shirt styles in the order and **n** the size of the hash table O in the test case. Next follow ***m*** lines of binary which the i:th describes the i:th t-shirt style t_i in the order by two non-negative binary integers h1(t_i) and h2(t_i) less than **n** giving the two hash function values of the style t_i. The two values may be identical.

Output (Part A)

For each test case there should be exactly one line of output either containing the string "valid inventory log" if it is possible to insert all t-shirt styles in the given order into the table(that is, the hashing number set ***IS HASHABLE***), or the string "INVALID INVENTORY LOG" if it is impossible.

Sample Input 1 (Part A)

2 //one decimal number 🡐 signaling there will be two test cases

3 3 //two *decimal* numbers 🡐 beginning of test case 1

0 1 //two binary numbers

1 10 //two binary numbers

10 0 //two binary numbers

5 6 //two *decimal* numbers 🡐 beginning of test case 2

10 11 //two binary numbers

11 1 //two binary numbers

1 10 //two binary numbers

101 1 //two binary numbers

10 101 //two binary numbers

Sample Output 1 (Part B)

valid inventory log

INVALID INVENTORY LOG!!!

Your Mission (Part B – Sorting the T-shirts)

Now that you have found which piece of paper corresponds to the Brand's order, you will use this to select the proper page listing the T-shirt descriptions. The number of pages with T-shirt descriptions will be the same as the number of pages with UPC codes. If there are two test cases (pages with UPC codes) then there will be two pages with descriptions of the T-shirts in the order. If the number of pages with UPC codes is 10, then there will be 10 pages that have T-shirt descriptions on them. The T-shirt descriptions will correspond to the text on the front of the T-shirt. ***Convert the characters in the text on the front of the t-shirt into their ASCII code values. Then sum the ASCII code values of all the characters on the text of the shirt. Use this number to organize the shirts into folded stacks of t-shirts on the floor in the shape of a BST tree.*** This way if your boss wants to find the pile of shirts with a certain text, she will have to check less than or equal to O(n), but a minimum of O(log n)[where n is the number of t-shirt styles], piles of different types of t-shirt styles to find the right one. You will be given the text for a t-shirt style and asked to report how many stacks of shirts your boss will have to look through before she finds the right style.

Sample Input 1 (Part B)

aacc // shirt to query for and report how many piles of shirts your boss needs to look through.

aabcc //text on front of a t-Shirt → because there are 3 UPCs in the valid order that come first, pick this style and the next two to add to the bst tree

aaccbb //text on front of a t-Shirt → pick, belongs to valid log

aacc //text on front of a t-Shirt → pick, belongs to valid log

ac //text on front of a t-Shirt → don't add to tree, part of invalid log

acdc //text on front of a t-Shirt → don't add to tree, part of invalid log

cac //text on front of a t-Shirt → don't add to tree, part of invalid log

ccf //text on front of a t-Shirt → don't add to tree, part of invalid log

cadc //text on front of a t-Shirt → don't add to tree, part of invalid log

Sample Output 1 (Part B)

2 // number of piles checked (nodes in tree) to find t-shirt style

NOTE ABOUT INPUT/OUTPUT

In these examples, I separated the input and output samples to make the instructions easier to digest. When the assignment is graded, the script will input one large input and expect one large output per test case. So the final sample inputs and outputs will be similar to the example shown below:

Sample Input 1 (Combined)

2 //one decimal number ← signaling there will be two hashing number sets

3 3 //two decimal numbers ← beginning of hashing number set 1

0 1 //two binary numbers

1 10 //two binary numbers

10 0 //two binary numbers

5 6 //two decimal numbers ← beginning of hashing number set 2

10 11 //two binary numbers

11 1 //two binary numbers

1 10 //two binary numbers

101 1 //two binary numbers

10 101 //two binary numbers

aacc // shirt to query for and report how many piles of shirts your boss needs to look through.

aabcc //text on front of a t-Shirt → because there are 3 UPCs in the valid order that come first, pick this style and the next two to add to the tree

aaccbb //text on front of a t-Shirt → pick, belongs to valid log

aacc //text on front of a t-Shirt → pick, belongs to valid log

ac //text on front of a t-Shirt → don't add to tree, part of invalid log

acdc //text on front of a t-Shirt → don't add to tree, part of invalid log

cac //text on front of a t-Shirt → don't add to tree, part of invalid log

ccf //text on front of a t-Shirt → don't add to tree, part of invalid log

cadc //text on front of a t-Shirt → don't add to tree, part of invalid log

Sample Output 1 (Combined)

valid inventory log

INVALID INVENTORY LOG!!!

2 // number of piles checked (nodes in tree) to find t-shirt style