National Institute of Technology Calicut
Department of Computer Science and Engineering
Third Semester B. Tech.(CSE)-Monsoon 2024
CS2091E Data Structures Laboratory
Assignment 3

Submission deadline (on or before): 28/08/2024, 11:59 PM

Policies for Submission and Evaluation:

- You must submit all the solutions of this assignment following the below-mentioned guidelines in the Eduserver course page, on or before the submission deadline.
- Ensure that your programs will compile and execute using GCC compiler without errors. The programs should be compiled and executed in the SSL/NSL.
- During the evaluation, failure to execute programs without compilation errors may lead to zero marks for that evaluation.
- Your submission will also be tested for plagiarism, by automated tools. In case your code fails to pass the test, you will be straightaway awarded zero marks for this assignment and considered by the examiner for awarding F grade in the course. Detection of ANY malpractice related to the lab course can lead to awarding an F grade in the course.

Naming Conventions for Submission

- Submit a single ZIP (.zip) file (do not submit in any other archived formats like .rar, .tar, .gz). The name of this file must be

  `ASSG<NUMBER>_<ROLLNO>_<FIRST-NAME>.zip`

  (Example: *ASSG1_BxxyyyyCS_LAXMAN.zip*). DO NOT add any other files (like temporary files, input files, etc.) except your source code, into the zip archive.

- The source codes must be named as

  `ASSG<NUMBER>_<ROLLNO>_<FIRST-NAME>_<PROGRAM-NUMBER>.c`

  (For example: *ASSG1_BxxyyyyCS_LAXMAN_1.c*). If you do not conform to the above naming conventions, your submission might not be recognized by our automated tools, and hence will lead to a score of 0 marks for the submission. So, make sure that you follow the naming conventions.

Standard of Conduct

- Violation of academic integrity will be severely penalized. Each student is expected to adhere to high standards of ethical conduct, especially those related to cheating and plagiarism. Any submitted work MUST BE an individual effort. Any academic dishonesty will result in zero marks in the corresponding exam or evaluation and will be reported to the department council for record keeping and for permission to assign F grade in the course. The department policy on academic integrity can be found at: https://minerva.nitc.ac.in/?q=node/650.

Questions

1. To conduct an event, our institute decided to split the students into four groups using a hash function, h. The procedure for grouping a student A using h is as follows:

   $h(A)$ = (Sum of ASCII value of the characters in the first name of 'A' + age of 'A') % 4, where 'A' represents a student *structure* to hold the student name, roll number as in our institute format and age. Such a hash value obtained is called a *group index.*

   Eg:- Let A.first_name='Veena', A.rollno=B220016CS and A.age=19, then h(A) = (86+101+101+110+97+19) % 4 = 2; where, 86 - ASCII(V), 101 - ASCII(e), 110 - ASCII(n), 97 - ASCII(a).

   If h(A) =0, the student is placed in group 0; if h(A) = 1, the student is placed in group 1; if h(A) = 2, the student is placed in group 2, and if h(A) = 3, the student is placed in group 3.

   Implement the following operations as a menu-driven program. Note that the functions described below except *StudentDetails()* take the hash table H as one of the parameters. Once the students' details are entered, the students are split into four groups based on the hash value.

   Operations

   (a) $GroupIndexAndSplit(A.first\_name, H)$: Using the hash function, calculate the group index for a given student A. Print the group index for the given student A.

   (b) $Group\_CountAndList(H, k)$: For group k, count the number of students and list their names in alphabetical order. Print the count followed by the names.

   (c) $Group\_ListByBranch(H, m, Branch)$: For group m, list the students who belong to the given *Branch*. The branch of a student can be identified from the last two characters of the roll number. If no students from that *Branch* are found in the group, print -1.

   (d) $StudentDetails(rollnumber)$: Retrieve and display the details *(first name, age, branch)* of the student with the given roll number. If no student with a given roll number is found, print -1.

   (e) $Group\_TransferAllByBranch(H, source\_m, target\_m, branch)$: Change the index of all those students in the source group index *source_m*, belonging to the specified *branch* to target group index *target_m*. Print the number of students transferred/changed.

   Note: *Assume all the students are from CS, EC, EE, or CE branches.*

   Input format:

   - First line of the input contains an integer 'n' $\in [1, 10^3]$, the total number of students who are participating in the event.
   - Next 'n' consecutive line contains: *first_name*, *roll_number*, and *age* of each student, separated by a single space.
   - Each subsequent line contains a character from {'a', 'b', 'c', 'd','e', 'f' } followed by zero or more positive integers n, followed by zero or two characters.
   - Character 'a' followed by a first name of A ($A.first\_name$) perform $GroupIndexAndSplit(A, H)$ operation.
   - Character 'b' followed by a positive integer 'k' $\in [0, 3]$. Perform $Group\_CountAndList(H, k)$ operation.

- Character 'c' followed by a positive integer 'm' $\in [0, 3]$, representing the group number, followed by two characters representing the branch name (both uppercase and lowercase are considered the same branch) separated by a space. Perform $Group\_ListByBranch(H, m, Branch)$ operation.
- Character 'd' followed by a string *rollnumber*, separated by a space.
  Perform $StudentDetails(rollnumber)$ operation.
- Character 'e' followed by a space separated two positive integers followed by two characters representing the branch name (both uppercase and lowercase are considered the same branch) separated by a space. Perform $Group\_TransferAllByBranch(H, source\_m, target\_m, branch)$ operation.
- Character 'f' is to terminate the sequence of operations.

Output format:

- The output (if any) of each command should be printed on a separate line. However, no output is printed for `'f'`.
- For option 'a': Print the group index (0, 1, 2, or 3) for the given student A.
- For option 'b': Print a positive integer representing the count of students in group k, followed by their names (strings) in the alphabetical order, separated by a space.
- For option 'c': Prints the strings representing the first names of the students in group m from the specified branch in a space-separated manner. Print -1 if no students from that branch are found in the group.
- For option 'd': Print the details $(firstname, age, branch)$ of a given student separated by a space. If no student is found, then print -1.
- For option 'e': Print an integer representing the number of students transferred.

Sample test case 1

Input:

```
5
Abu B210051CS 21
Veena B220016EC 19
Ishan B190016CE 22
Aleena B200036EE 21
Sam B230017CS 21
a Veena
b 1
c 2 EC
c 3 CS
d B190016CE
b 0
d B210151CS
c 2 CS
c 1 CS
e 0 2 CS
e 2 1 ec
f
```

Output:

```
2
2 Abu Ishan
Veena
```

```
-1
Ishan 22 CE
0
-1
Sam
Abu
0
1
```

2. You are tasked with implementing a hash table with unique elements, using chaining as the collision resolution method. The hash table should support the following functions:

   (a) *insert*(*hashTable*, *key*) : Inserts the positive integer *key* at a certain index obtained from the hash function below. If there are any collisions, use chaining. Maintain the sorted order while chaining. If the *key* is already present in the hash table, then print -1.

   $$\text{index} = (\text{key}) \mod (TableSize) \text{ or key\%TableSize.}$$

   (b) *search*(*hashTable*, *key*) : Searches for the key in the hash table. If the *key* is found, print the *index* at which it is stored in the hash table and print its *position* in the chain. If the *key* is not found, print -1.

   Example: *search*(15)

   Hash Table: `< index, chain >`

   ```
   .
   .
   4 :  8 -> NULL
   5 :  5 -> 15 -> 25 -> NULL
   .
   .
   ```

   Output: `5 (index) 2 (position in chain)`

   (c) *delete*(*hashTable*, *key*) : Deletes the key from the *hashTable* and prints the *index* at which it was stored in the hash table and its *position* in the chain. If the key is not found, print -1.

   (d) *update*(*hashTable*, *oldKey*, *newKey*) : Updates the *oldKey* in the hash table with *newKey* by deleting the *oldKey* and inserting the *newKey*. Print the index at which the *oldKey* was stored and its position in the chain. If the *oldKey* is not found or the *newKey* is already present in the hash table, print -1.

   (e) *printElementsInChain*(*hashTable*, *index*) : Print all the elements in sorted order with a space separating each that are present in the chain at the given *index* in the hash table. If the slot is empty print `-1`.

   Input format:

   - The first line contains an integer specifying the hash table's size.
   - Each subsequent line contains a character from {'`a`', '`b`', '`c`', '`d`', '`e`', '`f`'} followed by zero or more positive integers.
   - Input 'a' followed by a positive integer (key) calls the function *insert*(*hashTable*, *key*).
   - Input 'b' followed by a positive integer (key) calls the function *search*(*hashTable*, *key*).
   - Input 'c' followed by a positive integer (key) calls the function *delete*(*hashTable*, *key*).

- Input ‘d’ followed by two positive integers *oldKey* and *newKey* calls the function $update(hashTable, oldKey, newKey)$.
- Input `'e'` followed by a positive integer index $\in [0, \text{TableSize})$ calls the function $printElementsInChain(hashTable, index)$.
- Character *’f’* is to terminate the sequence of operations.

Output format:

- The output (if any) of each command should be printed on a separate line. However, no output is printed for `‘f’`.
- For option ’*a*’: If the *key* already exists, print -1, otherwise insert the element and no output is printed.
- For option ’*b*’: If the *key* is found, print the *index* and the *position* in the chain, separated by a space. If the *key* is not found, print -1.
- For option ’*c*’: If the *key* is successfully deleted, print the *index* and the *position* in the chain from where it was deleted, separated by a space. If the *key* is not found, print -1.
- For option ’*d*’: If the *oldKey* is deleted and the *newKey* is successfully inserted, print the *index* and the *position* in the chain separated by a space, where the *oldKey* was stored before updating. If the *oldKey* is not found or the *newKey* already exists, no operations are performed and print -1.
- For option ’*e*’: Print all the elements at the given *index* in sorted order, separated by a space. If the chain at the given *index* is empty, print -1.

Sample test case 1

Input:

```
5
a 1
a 5
a 10
a 15
a 20
e 0
b 15
c 15
a 10
d 15 25
d 10 25
e 1
f
```

Output:

```
5 10 15 20
0 3
0 3
-1
-1
0 2
1
```

3. Open addressing is a method for handling collisions in hashing. The three different methods for open addressing are *linear probing*, *quadratic probing*, and *double hashing*. A brief description of the three methods are given below:

- In **linear probing**, the function $h'(k)$ used to calculate the next location during a collision is:
$$h'(k) = (h(k) + i) \mod N, \quad i = 1, 2, \ldots$$
- In **quadratic probing**, the function $h'(k)$ used to calculate the next location during a collision is:
$$h'(k) = \left(h(k) + i^2\right) \mod N, \quad i = 1, 2, \ldots$$
- In the **double hashing** scheme:
  - The primary hash function is:
$$h_1(k) = k \mod N$$
    where N is the table size.
  - The secondary hash function is:
$$h_2(k) = R - (k \mod R)$$
    where R is the largest prime number less than the table size, N.
  - Double hashing can be done using:
$$h'(k) = (h_1(k) + i \times h_2(k)) \mod N, \quad i = 0, 1, 2, \ldots$$

Write a menu-driven program to implement the operations outlined below in a hash table. Note that the operations described below take the hash table as the parameter.
Operations

(a) *LinearProbing*(*hashTable*): The input keys should be inserted into the hash table in the prescribed order using *linear probing.* If a collision occurs, use the linear probing function to find the next available slot. Print the indices where the keys are stored (in the insertion order) and the number of collisions that occurred during the insertion.

(b) *QuadraticProbing*(*hashTable*): The input keys should be inserted into the hash table in the prescribed order using *quadratic probing.* If a collision occurs, use the quadratic probing function to find the next available slot. Print the indices where the keys are stored (in the insertion order) and the number of collisions that occurred during the insertion.

(c) *DoubleHashing*(*hashTable*): The input keys should be inserted into the hash table in the prescribed order using *double hashing.* Use the primary and secondary hash functions as described, and resolve collisions using the double hashing function. Print the indices where the keys are stored (in the insertion order) and the number of collisions that occurred during the insertion.

Input format:

- The first line contains two space-separated positive integers N, m where N specifies the size of the hash table and m specifies the number of positive integers intended to be inserted, respectively.
- The second line contains a sequence of m positive integers, separated by a space. representing the keys to be inserted.
- Each subsequent line contains a character from {‘`a`’, ‘`b`’, ‘`c`’, ‘`d`’ }.
- Character ’a’ perform *LinearProbing*(*hashTable*) operation.
- Character ’b’ perform *QuadraticProbing*(*hashTable*) operation.
- Character ’c’ perform *DoubleHashing*(*hashTable*) operation.
- Character ’d’ is to terminate the sequence of operations.

Output format:

- The output (if any) of each command should be printed on a separate line. However, no output is printed for `‘d’`.
- For option ’*a*’: Print the indices where the *keys* are stored in a single line separated by a space, and print the total number of collisions that occurred during insertion in the next line.
- For option ’*b*’: Print the indices where the *keys* are stored in a single line separated by a space, and print the total number of collisions that occurred during insertion in the next line.
- For option ’*c*’: Print the indices where the *keys* are stored in a single line separated a by space, and print the total number of collisions that occurred during insertion in the next line.

Sample test case 1

Input:

```
7 6
76 93 40 47 10 55
a
b
c
d
```

Output:

```
6 2 5 0 3 1
4
6 2 5 0 3 1
6
6 2 5 1 3 4
2
```