Problem 2408: Design SQL

Problem Information

Difficulty: Medium
Acceptance Rate: 66.01%
Paid Only: Yes
Tags: Array, Hash Table, String, Design

Problem Description

You are given two string arrays, `names` and `columns`, both of size `n`. The `ith` table is represented by the name `names[i]` and contains `columns[i]` number of columns.

You need to implement a class that supports the following **operations** :

* **Insert** a row in a specific table with an id assigned using an _auto-increment_ method, where the id of the first inserted row is 1, and the id of each _new_ row inserted into the same table is **one greater** than the id of the **last inserted** row, even if the last row was _removed_. * **Remove** a row from a specific table. Removing a row **does not** affect the id of the next inserted row. * **Select** a specific cell from any table and return its value. * **Export** all rows from any table in csv format.

Implement the `SQL` class:

* `SQL(String[] names, int[] columns)` * Creates the `n` tables. * `bool ins(String name, String[] row)` * Inserts `row` into the table `name` and returns `true`. * If `row.length` **does not** match the expected number of columns, or `name` is **not** a valid table, returns `false` without any insertion. * `void rmv(String name, int rowId)` * Removes the row `rowId` from the table `name`. * If `name` is **not** a valid table or there is no row with id `rowId`, no removal is performed. * `String sel(String name, int rowId, int columnId)` * Returns the value of the cell at the specified `rowId` and `columnId` in the table `name`. * If `name` is **not** a valid table, or the cell `(rowId, columnId)` is **invalid** , returns `"<null>"`. * `String[] exp(String name)` * Returns the rows present in the table `name`. * If name is **not** a valid table, returns an empty array. Each row is represented as a string, with each cell value (**including** the row's id) separated by a `","`.

Example 1:

Input:

["SQL","ins","sel","ins","exp","rmv","sel","exp"] [[["one","two","three"],[2,3,1]],["two",["first","second","third"]],["two",1,3],["two",["fourth","fifth","sixth"]],["two"],["two",1],["two",2,2],["two"]]

Output:

[null,true,"third",true,["1,first,second,third","2,fourth,fifth,sixth"],null,"fifth",["2,fourth,fifth,sixth"]]

Explanation:

// Creates three tables. SQL sql = new SQL(["one", "two", "three"], [2, 3, 1]); // Adds a row to the table "two" with id 1. Returns True. sql.ins("two", ["first", "second", "third"]); // Returns the value "third" from the third column // in the row with id 1 of the table "two". sql.sel("two", 1, 3); // Adds another row to the table "two" with id 2. Returns True. sql.ins("two", ["fourth", "fifth", "sixth"]); // Exports the rows of the table "two". // Currently, the table has 2 rows with ids 1 and 2. sql.exp("two"); // Removes the first row of the table "two". Note that the second row // will still have the id 2. sql.rmv("two", 1); // Returns the value "fifth" from the second column // in the row with id 2 of the table "two". sql.sel("two", 2, 2); // Exports the rows of the table "two". // Currently, the table has 1 row with id 2. sql.exp("two");

Example 2:

Input:

["SQL","ins","sel","rmv","sel","ins","ins"] [[["one","two","three"],[2,3,1]],["two",["first","second","third"]],["two",1,3],["two",1],["two",1,2],["two",["fourth","fifth"]],["two",["fourth","fifth","sixth"]]]

Output:

[null,true,"third",null,"<null>",false,true]

Explanation:

// Creates three tables. SQL sQL = new SQL(["one", "two", "three"], [2, 3, 1]); // Adds a row to the table "two" with id 1. Returns True. sQL.ins("two", ["first", "second", "third"]); // Returns the value "third" from the third column // in the row with id 1 of the table "two". sQL.sel("two", 1, 3); // Removes the first row of the table "two". sQL.rmv("two", 1); // Returns "<null>" as the cell with id 1 // has been removed from table "two". sQL.sel("two", 1, 2); // Returns False as number of columns are not correct. sQL.ins("two", ["fourth", "fifth"]); // Adds a row to the table "two" with id 2. Returns True. sQL.ins("two", ["fourth", "fifth", "sixth"]);

Constraints:

* `n == names.length == columns.length` * `1 <= n <= 104` * `1 <= names[i].length, row[i].length, name.length <= 10` * `names[i]`, `row[i]`, and `name` consist only of lowercase English letters. * `1 <= columns[i] <= 10` * `1 <= row.length <= 10` * All `names[i]` are **distinct**. * At most `2000` calls will be made to `ins` and `rmv`. * At most `104` calls will be made to `sel`. * At most `500` calls will be made to `exp`.

Follow-up: Which approach would you choose if the table might become sparse due to many deletions, and why? Consider the impact on memory usage and performance.

Code Snippets

C++:

```
class SQL {
public:
SQL(vector<string>& names, vector<int>& columns) {

}

bool ins(string name, vector<string> row) {

}

void rmv(string name, int rowId) {

}

string sel(string name, int rowId, int columnId) {

}

vector<string> exp(string name) {

}
};

/**
```

```

* Your SQL object will be instantiated and called as such:
* SQL* obj = new SQL(names, columns);
* bool param_1 = obj->ins(name,row);
* obj->rmv(name,rowId);
* string param_3 = obj->sel(name,rowId,columnId);
* vector<string> param_4 = obj->exp(name);
*/
```

Java:

```
class SQL {

public SQL(List<String> names, List<Integer> columns) {

}

public boolean ins(String name, List<String> row) {

}

public void rmv(String name, int rowId) {

}

public String sel(String name, int rowId, int columnId) {

}

public List<String> exp(String name) {

}
}

/**
* Your SQL object will be instantiated and called as such:
* SQL obj = new SQL(names, columns);
* boolean param_1 = obj.ins(name,row);
* obj.rmv(name,rowId);
* String param_3 = obj.sel(name,rowId,columnId);
* List<String> param_4 = obj.exp(name);
*/
```

Python3:

```
class SQL:

def __init__(self, names: List[str], columns: List[int]):


def ins(self, name: str, row: List[str]) -> bool:


def rmv(self, name: str, rowId: int) -> None:


def sel(self, name: str, rowId: int, columnId: int) -> str:


def exp(self, name: str) -> List[str]:



# Your SQL object will be instantiated and called as such:
# obj = SQL(names, columns)
# param_1 = obj.ins(name,row)
# obj.rmv(name,rowId)
# param_3 = obj.sel(name,rowId,columnId)
# param_4 = obj.exp(name)
```