CS 6301 Implementation of data structures and algorithms
Long Project 4: Hashing & Multi-dimensional Search

Ver 1.0: Initial description (Sunday, Apr 9).
Due: 11:59 PM, Tue, Apr 25.
Deadline for EC problem is 11:59 PM, Sun Apr 30th,

Max excellence credits: 1.0.

- The submission procedure is same as the same as that of prior projects.
- For each group, only its last submission is kept, and earlier submissions are discarded.
- Your code must be of good quality, well commented, and pass all test cases within time limits to earn excellence credits.

1. **Multi-dimensional search**

Implement the operations described below that are required to perform multi-dimensional search. Starter code (MDS.java) is provided. Do not change the signatures of methods declared to be public. Driver code is also provided along with several test cases.

Consider the web site of a seller like Amazon. They carry tens of thousands of products, and each product has many attributes (Name, Size, Description, Keywords, Manufacturer, Price, etc.). The search engine allows users to specify attributes of products that they are seeking, and shows products that have most of those attributes. To make search efficient, the data is organized using appropriate data structures, such as balanced trees. But, if products are organized by Name, how can search by price implemented efficiently? The solution, called indexing in databases, is to create a new set of references to the objects for each search field, and organize them to implement search operations on that field efficiently. As the objects change, these access structures have to be kept consistent.

In this project, each object has 3 attributes: id (long int), description (one or more long ints), and price (dollars and cents). The following operations are supported:

```
Insert(id,price,list): insert a new item whose description is given in
the list.  If an entry with the same id already exists, then its
description and price are replaced by the new values, unless list   is
null or empty, in which case, just the price is updated. Returns 1 if
the item is new, and 0 otherwise.

Find(id): return price of item with given id (or 0, if not found).
```

Delete(id): delete item from storage. Returns the sum of the long ints that are in the description of the item deleted(or 0, if such an id did not exist).

FindMinPrice(n): given a long int, find items whose description contains that number (exact match with one of the long ints in the item's description), and return lowest price of those items. Return 0 if there is no such item.

FindMaxPrice(n): given a long int, find items whose description contains that number, and return highest price of those items.
Return 0 if there is no such item.

FindPriceRange(n,low,high): given a long int n, find the number of items whose description contains n, and in addition, their prices fall within the given range, [low, high].

PriceHike(l,h,r): increase the price of every product, whose id is in the range [l,h] by r%. Discard any fractional pennies in the new prices of items. Note that you are truncating, not rounding.
Returns the sum of the net increases of the prices.

RemoveNames(id, list): Remove elements of list from the description of id. It is possible that some of the items in the list are not in the id's description. Return the sum of the numbers that are deleted from the description of id. Return 0 if there is no such id.

Implement the operations using data structures that are best suited for the problem.

Input specification:

Initially, the store is empty, and there are no items. The input contains a sequence of lines (use test sets with millions of lines). Lines starting with "#" are comments. Other lines have one operation per line: name of the operation, followed by parameters needed for that operation (separated by spaces). Lines with Insert operation will have a "0" at the end, that is not part of the name. The output is a single number, which is the sum of the following values obtained by the algorithm as it processes the input.

```
Sample input:
Insert 22 19.97 475 1238 9742 0
# New item with id=22, price="$19.97", name="475 1238 9742"
# Return: 1
#
Insert 12 96.92 44 109 0
# Second item with id=12, price="96.92", name="44 109"
# Return: 1
#
Insert 37 47.44 109 475 694 88 0
# Another item with id=37, price="47.44", name="109 475 694 88"
# Return: 1
#
```

```
PriceHike 10 22 10
# 10% price increase for id=12 and id=22
# New price of 12: 106.61, Old price = 96.92.  Net increase = 9.69
# New price of 22: 21.96.  Old price = 19.97.  Net increase = 1.99
# Return: 11.68 (sum of 9.69 and 1.99).  Added to total: 11
#
FindMaxPrice 475
# Return: 47.44 (id of items considered: 22, 37).  Added to total: 47
#
Delete 37
# Return: 1366 (=109+475+694+88)
#
FindMaxPrice 475
# Return: 21.96 (id of items considered: 22).  Added to total: 21
#


Output:
1448
```

2. **Cuckoo Hashing**

Implement Cuckoo hashing algorithm. Use a secondary hash table when the threshold is reached during insertion. You can even use hash table from java library for the secondary hash table.

3. **Comparison Study (optional for EC)**

Compare your Cuckoo hashing with Java's HashTable/HashMap/HashSet on millions of operations: add, contains, and remove. Compare for load factors of 0.5, and 0.75. Also, investigate the performance of Cuckoo algorithm with the load factor of 0.9. Write a short report on your studies. Do not restrict the key space from 0 to size. If you are using an instance of Random class from Java, use the method nextInt(). Do not use nextInt(size).