Exercises: Iterators And Comparators

This document defines the exercises for "Java Advanced" course @ Software University. Please submit your solutions (source code) of all below described problems in Judge.

Problem 1. Listylterator

Create a class "Listylterator", it should receive the collection of **Strings** which it will iterate, through its constructor. You should store the elements in a List. The class should have three main functions:

- Move should move an internal index position to the next index in the list, the method should return true if it successfully moved and false if there is no next index.
- HasNext should return true if there is a next index and false if the index is already at the last element of the list.
- **Print** should **print** the element at the current internal index, calling Print on a collection without elements should throw an appropriate exception with the message "Invalid Operation!".

By default, the internal index should be pointing to the **0**th **index** of the List. Your program should support the following commands:

Command	Return Type	Description
Create {e1 e2}	void	Creates a ListyIterator from the specified collection. In case of a Create command without any elements, you should create a ListyIterator with an empty collection.
Move	boolean	This command should move the internal index to the next index.
Print	void	This command should print the element at the current internal index.
HasNext	boolean	Returns whether the collection has a next element.
END	void	Stops the input.

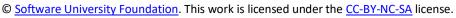
Input

Input will come from the console as lines of commands. The first line will always be Create command in the input. The last command received will always be **END** command.

Output

For every command from the input (with the exception of the END and Create commands) print the result of that command on the console, each on a new line. In case of Move or HasNext commands print the returned value of the method, in case of a Print command you don't have to do anything additional as the method itself should already print on the console. Your program should catch any exceptions thrown because of validations (calling Print on an empty collection) and print their messages instead.

















- There will always be only one Create command and it will always be the first command passed.
- The number of commands received will be **between [1...100]**.
- The last command will always be the **END** command.

Examples

Input	Output
Create	Invalid Operation!
Print	
END	
Create Stefcho Goshky	true
HasNext	Stefcho
Print	true
Move	Goshky
Print	
END	
Create 1 2 3	true
HasNext	true
Move	true
HasNext	true
HasNext	true
Move	false
HasNext	
END	

Problem 2. Collection

Using the ListyIterator from the last problem, extend it by implementing the "Iterable" interface, implement all methods desired by the interface manually. Add a new method to the class PrintAll(), the method should foreach the collection and print all elements on a single line separated by a space.

Input

Input will come from the console as lines of commands. The first line will always be the Create command in the input. The **last** command received will always be the **END** command.

Output

For every command from the input (with the exception of the END and Create commands) print the result of that command on the console, each on a new line. In case of Move or HasNext commands print the returned value of the method, in case of a **Print** command you don't have to do anything additional as the method itself should already print on the console. In case of a PrintAll command you should print all elements on a single line separated by spaces. Your program should catch any exceptions thrown because of validations and print their messages instead.

















- Do not use the built in iterators!
- There will always be only one Create command and it will always be the first command passed.
- The number of commands received will be **between [1...100]**.
- The last command will always be the END command.

Examples

Input	Output
Create 1 2 3 4 5	true
Move	1 2 3 4 5
PrintAll	
END	
Create Stefcho Goshky Peshu	Stefcho Goshky Peshu
PrintAll	true
Move	true
Move	Peshu
Print	false
HasNext	
END	

Problem 3. Stack Iterator

Since you have passed the basics algorithms course, now you have a task to create your custom stack. You are aware of the Stack structure. There is a collection to store the elements and two functions (not from the functional programming) - to push an element and to pop it. Keep in mind that the first element which is popped is the last in the collection. The push method adds an element to the top of the collection and the pop method returns the top element and removes it.

Write your custom implementation of Stack<Integer> and implement your custom iterator. There is a way that IntelliJ could help you, your Stack class should implement the "Iterable<Integer>" interface and your Iterator Class should implement "Iterator<Integer>" interface. Your Custom Iterator should follow the rules of the Abstract Data Type - Stack. Like we said, the first element is the element at the top and so on. Iterators are used only for iterating through the collection, they **should not** remove or mutate the elements.

Input

The input will come from the console as lines of commands. Commands will only be **push** and **pop**, followed by integers for the **push** command and **no another** input for the **pop** command.

Format:

- "Push {element, secondElement...}"
- "Pop"

Output

When you receive "END", the input is over. Foreach the stack twice and print all elements. Each element should be on a new line.





















- The elements in the push command will be valid integers between [2⁻³²...2³²-1].
- The commands will always be valid (always be either Push, Pop or END).
- There will be no more than **16 elements** in **Push** command.
- If Pop command **could not** be executed as expected (e.g. no elements in the stack), print on the console: "No elements".

Examples

Input	Output
Push 1, 2, 3, 4	2
Рор	1
Рор	2
END	1
Push 1, 2, 3, 4	1
Рор	3
Push 1	2
END	1
	1
	3
	2
	1
Push 1, 2, 3, 4	No elements
Рор	
Рор	
Рор	
Рор	
Pop	
END	

Problem 4. Froggy

Let's play a game. You have a tiny little Frog, and a Lake with numbers. The Lake and its numbers, you will get by an input from the console. Imagine, your Frog belongs to the Lake. The Frog jumps only when "END" command is received. When the Frog starts jumping, print on the console each number the Frog has stepped over. To calculate the jumps, use the guide lines:

The jumps start from the Oth index. And follows the pattern - first all even indexes in ascending order(0->2->4->6 and so on) and then all odd indexes in **ascending** order (1->3->5->7 and so on). Consider the **0**th index as **even**.

Long story short: Create a Class "Lake", it should implement the interface - "Iterable". Inside the Lake, create a Class - "Frog" and implement the interface "Iterator". Keep in mind, you will be given integers only.















Input

The input will consist of two lines. First line - the initial numbers of the lake, separated by comma and a single space. Second line command is "END".

Output

When you receive "END", the input is over. Foreach the collection of numbers the Frog has jumped over and print each element.

The output should be print on a single line.

Format:

{number}, {second number}, {third number} ...

Constraints

- Lake's numbers will be valid integers in the range [2⁻³²...2³²-1].
- The command will always be valid "END".

Examples

Input	Output
1, 2, 3, 4, 5, 6, 7, 8 END	1, 3, 5, 7, 2, 4, 6, 8
1, 2, 3 END	1, 3, 2

Problem 5. Comparing Objects

There is a Comparable interface but probably you already know. Your task is simple. Create a Class Person. Each person should have name, age and town. You should implement the interface - Comparable and try to override the compareTo method. When you compare two people, first you should compare their names, after that - their age and last but not at least - compare their town.

Input

On single lines, you will be given people in format:

{name} {age} {town}

Collect them till you receive "END"

After that, you will receive an integer **N** - the **N**th person in your collection.

Output

On the single output line, you should bring statistics, how many people are equal to him, how many people are not equal to him and the total people in your collection.

Format:

{number of equal people} {number of not equal people} {total number of people}

















- Names, ages and addresses will be valid.
- Input number will be always a valid integer in range [2...100].
- If there are no equal people print: "No matches"

Examples

Input	Output
Pesho 22 Vraca	No matches
Gogo 14 Sofeto	
END	
2	
Pesho 22 Vraca	2 1 3
Gogo 22 Vraca	
Gogo 22 Vraca	
END	
2	

Problem 6. Strategy Pattern

An interesting pattern you may have heard of is the Strategy Pattern, if we have multiple ways to do a task (let's say sort a collection) it allows the client to choose the way that most fits his needs. A famous implementation of the pattern in Java is the Collections.sort() method that takes a Comparator.

Create a class Person that holds name and age. Create 2 Comparators for Person (classes which implement the Comparator<Person> interface). The first comparator should compare people based on the length of their name as a first parameter, if 2 people have a name with the same length, perform a case-insensitive compare based on the first letter of their name instead. The second comparator should compare them based on their age. Create 2 TreeSets of type Person, the first should implement the name comparator and the second should implement the age comparator.

Input

On the first line of input you will receive a number N. On each of the next N lines you will receive information about people in the format "<name> <age>". Add the people from the input into both sets (both sets should hold all the people passed in from the input).

Output

Foreach the sets and print each person from the set on a new line in the same format that you received them. Start with the set that implements the name comparator.

Constraints

- A person's name will be a string that contains **only** alphanumerical characters with a length **between [1...50]** symbols.
- A person's age will be a **positive** integer **between [1...100]**.
- The number of people **N** will be a **positive** integer **between [0...100]**.

















Examples

Input	Output
3	Joro 100
Pesho 20	Pesho 20
Joro 100	Pencho 1
Pencho 1	Pencho 1
	Pesho 20
	Joro 100
5	asen 33
Ivan 17	Ivan 17
asen 33	Joro 3
Stoqn 25	Nasko 99
Nasko 99	Stoqn 25
Joro 3	Joro 3
	Ivan 17
	Stoqn 25
	asen 33
	Nasko 99

Problem 7. *Equality Logic

Create a class Person holding name and age. A person with the same name and age should be considered the same, override any methods needed to enforce this logic. Your class should work with both standard and hashed collections. Create a TreeSet and a HashSet of type Person.

Input

On the first line of input you will receive a number N. On each of the next N lines you will receive information about people in the format "<name> <age>". Add the people from the input into both sets (both sets should hold all the people passed in from the input).

Output

The output should consist of exactly 2 lines. On the first you should print the size of the TreeSet and on the second the size of the HashSet.

Constraints

- A person's name will be a string that contains only alphanumerical characters with a length between [1...50] symbols.
- A person's age will be a **positive** integer **between [1...100]**.
- The number of people **N** will be a positive integer **between [0...100]**.

















Examples

Input	Output
4	4
Pesho 20	4
Peshp 20	
Joro 15	
Pesho 21	
7	5
Ivan 17	5
ivan 17	
Stoqn 25	
Ivan 18	
Ivan 17	
Stopn 25	
Stoqn 25	

Hint

You should override both the equals and hashCode methods. You can check online for an implementation of hashCode - it doesn't have to be perfect, but it should be good enough to produce the same hash code for objects with the same name and age, and different enough hash codes for objects with different name and/or age.

Problem 8. *Pet Clinics

You are a young and ambitious owner of a Pet Clinics Holding. You ask your employees to create a program which will store all information about the pets in the database. Each pet should have name, age and kind.

Your application should support a few basic operations such as creating a pet, creating a clinic, adding a pet to a clinic, releasing a pet from a clinic, printing information about a specific room in a clinic or printing information about all rooms in a clinic.

Clinics should have an odd number of rooms, attempting to create a clinic with an even number of should fail and throw an appropriate exception.

Accommodation Order

For example let us take a look at a clinic with 5 rooms. The first room where a pet will be treated is the central one (room 3). So the order of which an animal is entering is: first animal is going to the centre (3) room and after that the next pets are entering first to the **left** (2) and then to the **right** (4) room. The last rooms in which pets can enter are room 1 and room 5. In case a room is already occupied, we skip it and go to the next room in the above order. Your task is to model the application and make it support some commands.

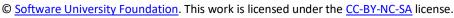
12345 The first pet enters room 3.

1 **2** 3 4 5 After that, the next pet enters room 2.

123**4**5 The third pet would enter room 4.

1234**5** And the last two pets would be going to rooms - 1 and 5.

















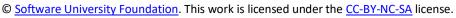
Now when we have covered adding the pets, it is time to find a way to release them. The process of releasing them is not so simple, when the release method is called, we start from the centre room (3) and continue right (4, 5... and so on) until we find a pet or reach the last room. If we reach the last room, we start from the first (1) and again move right until we reach the centre room (3). If a pet is found, we remove it from the collection, stop further search and return true, if a pet is NOT found, the operation returns false.

When a print command for a room is called, if the room contains a pet we print the pet on a single line in the format "<pet name> <pet age> <pet kind>". Alternatively if the room is empty print "Room empty" instead. When a print command for a clinic is called it should print all rooms in the clinic in order of their number.

Commands

Command	Return Type	Description
Create Pet {name} {age} {kind}	void	Creates a pet with the specified name and age.
		(true if the operation is successful and false if it isn't)
Create Clinic {name} {rooms}	void	Creates a Clinic with the specified name and number of rooms.
		(if the rooms are not odd, throws an exception)
Add {pet's name} {clinic's name}	boolean	This command should add the given pet in the specified clinic.
		<pre>(true if the operation is successful and false if it isn't).</pre>
Release {clinic's name}	boolean	This command should release an animal from the specified clinic.
		<pre>(true if the operation is successful and false if it isn't).</pre>
HasEmptyRooms {clinic's name}	boolean	Returns whether the clinic has any empty rooms.
		<pre>(true if it has and false if it doesn't).</pre>
Print {clinic's name}	void	This command should print each room in the specified clinic, ordered by room number.
Print {clinic's name} {room}	void	Prints on a single line the content of the specified room.















Input

On the first line, you will be given an integer N - the number of commands you will receive. On each of the next N lines you will receive a command. Commands and parameters will always be correct (Add, Release, HasEmptyRooms and Print commands will always be passed to existing clinics/pets), except for the number of rooms in the Create Clinic command which can be any valid integer between 1 and 101.

Output

For each command with a boolean return type received through the input, you should print its return value on a separate line. In case of a method throwing an exception (such as trying to create a clinic with even number of rooms or trying to add a pet that doesn't exist) you should catch the exceptions and instead print "Invalid Operation!". The Print command with a clinic and a room should print information for that room in the format specified above. The Print command with only a clinic should print information for each room in the clinic in order of their numbers.

Constraints

- The number of commands N will be a valid integer between [1...1000], no need to check it explicitly.
- Pet names, Clinic names, and kind will be strings consisting only of alphabetical characters with length between [1...50] characters.
- **Pet age** will be a positive integer **between [1...100]**.
- Clinic rooms will be a positive integer between [1...101].
- Room number in a Print command will always be between 1 and the number of rooms in that Clinic.
- Input will consist only of correct commands and they will always have the correct type of parameters.

Example

Input	Output
9	Invalid Operation!
Create Pet Gosho 7 Cat	true
Create Clinic Rezovo 4	false
Create Clinic Rizovo 1	true
HasEmptyRooms Rizovo	false
Release Rizovo	false
Add Gosho Rizovo	
HasEmptyRooms Rizovo	
Create Pet Sharo 2 Dog	
Add Sharo Rizovo	
8	true
Create Pet Gosho 7 Cat	true
Create Pet Sosho 1 Cata	Gosho 7 Cat
Create Clinic Rezovo 5	true
Add Gosho Rezovo	Room empty
Add Sosho Rezovo	Sosho 1 Cata
Print Rezovo 3	Room empty
Release Rezovo	Room empty



















Print Rezovo	Room empty

***Linked List Traversal Problem 9.

You need to write your own simplified implementation of a generic Linked List which has an Iterator. The list should support the Add and Remove operations, should reveal the amount of elements it has with a getSize function and should have an implemented iterator (should be foreachable). The add method should add the new element at the end of the collection. The remove method should remove the first occurrence of the item starting at the beginning of the collection, if the element is successfully removed the method returns true, alternatively if the element passed is not in the collection the method should return false. The getSize method should return the number of elements currently in the list. The iterator should iterate over the collection starting from the first entered element.

Input

On the first line of input you will receive a number N. On each of the next N lines you will receive a command in one of the following formats:

- "Add <number>" adds a number to your linked list.
- "Remove <number>" removes the first occurrence of the number from the linked list. If there is no such element this command leaves the collection unchanged.

Output

The output should consist of exactly 2 lines. On the first you should print the result of calling the getSize function on the Linked list. On the next lines you should print all elements of the collection by calling foreach on the collection.

Constraints

- All numbers in the input will be valid integers between [2-32...232-1].
- All commands received through the input will be valid (will be only "Add" or "Remove").
- The number **N** will be a positive integer **between [1...500]**.

Examples

Input	Output
5	2
Add 7	-50 20
Add -50	
Remove 3	
Remove 7	
Add 20	
6	4
Add 13	13 20 4 4
Add 4	
Add 20	
Add 4	
Remove 4	

















Add 4	

Hint

You can use the Linked List from your **Workshop**.















