Spring 2026 COP 3330 Program #7: Use of Java API (StringTokenizer, ArrayList, HashMap, ?)

Part A: Strange Line for Tickets

In a normal line for tickets, whenever people show up, they add to the back of the line. When a salesperson is ready, they take the person from the front of the line. For various reasons, for this concert tour, the line doesn't work in this simple manner. Instead, the following expanded list operations are allowed:

1) Add a person to the back of the line.
2) Insert a person at index idx in the line (so they may cut some people)
3) Letting the person at the front of the line buy tickets, first removing them from the line.
4) Letting the person at index idx in the line buy tickets, first removing them from the line.

Create a simple class Person to store the following information about each person in line:

1) First Name
2) Last Name
3) Number of Tickets

Use an ArrayList<Person> to simulate these operations as they are entered in order. The input format will cover the exact method by which the input will be given for each of the operations. Note that some of these operations (namely #2, #3 and #4) are quite slow (O(n)).

Input Format (from standard input)

The first line of input will contain one positive integer: ***n*** (***n*** $\leq$ 1000), representing the number of operations to simulate. Assume that the line starts empty before the operations start and that no "illegal" operation will be requested. Namely, people will only be asked to buy tickets if there is someone in line to buy them and if a person is added or removed from an index in line, that index will be valid.

The next ***n*** lines will contain the requested operations, one per line.

The first item on each line will be a number (1, 2, 3 or 4), representing one of the operations outlined previously.

If the operation is 1, then this is followed by the first name, last name and number of tickets for the person being added to the back of the line. Both names will have letters only and the number of tickets will be a positive integer.

If the operation is 2, this is followed by the first name, last name, number of tickets and 0-based position in line to which the person is being inserted.

If the operation is 3, no further information is given.

If the operation is 4, this is followed by a single non-negative integer representing the position in line from which a person is pulled to buy tickets.

Output Format

For each request of type 3 or 4, output a single line with the following format:

```
First Last bought X tickets.
```

Sample Input

```
10
1 Samantha Givens 3
1 Rob Prather 4
2 Jim CutInLine 2 0
3
3
2 Maryam Shenk 5 0
1 Anna Cambridge 12
4 2
1 Oscar Moore 8
3
```

Sample Output

```
Jim CutInLine bought 2 tickets.
Samantha Givens bought 3 tickets.
Anna Cambridge bought 12 tickets.
Maryam Shenk bought 5 tickets.
```

Implementation requirements (for part A)

1. You must create a Person class with 3 instance variables.
2. You must use an ArrayList<Person> for your simulation.

Part B: Concert Sponsors

Artists are always changing their sponsorship so it can be very hard to keep track of who their current sponsor is. Due to strict industry rules, artists can only have at most one sponsor at any given time. Thus, when they are approached with a "better deal", they typically just switch their sponsor. In very rare occasions, a sponsor will drop their sponsorship of an artist, due to various reasons.

For this program, you'll use a HashMap to quickly track current sponsors of artists and answer queries about an artists' current sponsor.

The input will contain 3 types of commands:

1) A band changes its sponsor to X.
2) A band loses its sponsor.
3) A query for who, if anyone, is a band's sponsor.

Input Format (from standard input)

The first line of input will contain a single positive integer: n ($n \leq 3 \times 10^5$), representing the number of actions to simulate.

The first line of each action will contain a single integer, 1, 2 or 3, indicating the action.

The second line of each action will contain a string, possibly with spaces, indicating the name of the band for the operation.

Operation #1 will have a third line of input, another string, representing the sponsor for the band. This string may have spaces as well.

Output Format

For each action of type 3, output on a line by itself the band's sponsor, if it exists, or the string:

```
No sponsor
```

If the band is not currently sponsored.

Sample Input

```
10
1
Red Hot Chili Peppers
Pepsi
1
Billy Joel
Coke
3
Billy Joel
1
Red Hot Chili Peppers
Disney
3
Red Hot Chili Peppers
3
Taylor Swift
2
Billy Joel
3
Billy Joel
1
Taylor Swift
Apple
3
Taylor Swift
```

Sample Output

```
Coke
Disney
No sponsor
No sponsor
Apple
```

Implementation requirements (for part B)

1. You must only read input by lines and use Integer.parseInt as needed.
2. You must use a HashMap.

Part C: In Demand Tickets with Priority!

It never seems like there are enough concert tickets available for the demand!

Also, the game seems rigged. Instead of all tickets being sold at the same time, tickets are released in batches, and then of course there's resale...ugh!

Finally, not all people waiting for tickets are the same. Radio stations and other entities get priority. So, even if you've been waiting longer, the radio station gets tickets first when they become available. For this problem you'll simulate this process.

Each entity buying tickets will be a Buyer object. A Buyer object will have the following instance variables:

1) Name (string with printable non-whitespace characters)
2) Priority Number (positive integer in between 1 and a million, 1 is higher priority...)
3) Number of Tickets for the Request

Whenever multiple entities are waiting to buy tickets, if some number of tickets become available, they are first sold to the Buyer with the lowest priority number. If there is a tie with priority number, then the Buyer with the fewer number of tickets gets dibs. If two Buyers happen to have the same priority number and the same number of tickets requested, then the tie is broken by a lexicographical string comparison between the names of the Buyers with the Buyer with the first name lexicographically coming first. All Buyers will have unique names and no buyer will make more than one request.

Your program will process a sequence of requests. Each request is of one of two types:

1) Add Buyer to the queue
2) Some number of tickets have become available, all are given out to Buyers in accordance to the priorities previously described.

It's guaranteed that the number of tickets that become available will never exceed the number of tickets being requested at that moment in time.

Also, a few things could happen with type 2 requests. It's possible that the number of tickets is less than the next Buyer waiting for tickets in the queue. In this case, all of those tickets will get disbursed immediately to that Buyer, and then the Buyer will be re-entered in the queue for their remaining number of tickets. (Say the first buyer wants 10 tickets and 4 become available. The four tickets will be given to that Buyer as they are removed from the queue, and then the Buyer is reinserted in the queue with 6 tickets left on the request.)

It's possible that the number of tickets matches exactly, in which case the Buyer is NOT put back in the queue.

Finally, it's possible that many tickets become available and several Buyers can be sold their full allotment of tickets, with the last Buyer maybe getting only some of the tickets they need.

Input Format (from standard input)
The first line of input will contain a single positive integer: n ($n \leq 3x10^5$), representing the number of actions to simulate.

Each action will be described in a single line, with the actions listed in order.

Each action starts with an integer, 1 or 2, indicating either adding a Buyer to the queue or a number of tickets being released for sale.

If the integer is 1, then this is followed by a single string representing the name of the buyer, followed by that buyer's priority number (integer in between 1 and 10^6) and followed by the number of tickets they would like to buy (positive integer in between 1 and 10^5).

If the integer is 2, this is followed by a second positive integer representing the number of tickets being released for sale.

Output Format
For each action of type 2, output one or more lines of the form:

```
Buyer receives X tickets.
```

where Buyer is the name of the Buyer who is receiving the tickets and X is the number of tickets they are receiving. X must either equal the number of tickets requested or be less than it. If the latter is true, then this must be the last print for that action and the Buyer should be readded to the queue. The order of these lines should be in the order the Buyers receive the tickets.

Sample Input
```
6
1 XL106.7 2 50
1 Mary 100 4
1 RichGuy 1 12
2 35
1 PoorGuy 101 2
2 32
```

Sample Output
```
RichGuy receives 12 tickets.
XL106.7 receives 23 tickets.
XL106.7 receives 27 tickets.
Mary receives 4 tickets.
PoorGuy receives 1 tickets.
```

Implementation requirements (for part B)
1. You must create a Buyer class which implements Comparable<Buyer>.
2. You must use a PriorityQueue to simulate the ticket queue for the problem.

Deliverables

Please submit the following source files for each Part:

1. **Person.java and StrangeLine.java** which contains your solution to Part A.
2. **Sponsors.java** which contains your solution to Part B.
3. **Buyer.java and TicketSimulation.java** which contains your solution to Part C.

Make sure to include a header comment for each file you submit and appropriate internal comments.