COL 106 : Data Structures and Algorithms
Semester II, 2022-23, IIT Delhi

Assignment - 4 (due on 18th March, 11:00 PM)

Important Guidelines:

- You must ensure that your submitted code runs in the JDK 11.0.17 environment.
- You are not allowed to share your code with anyone. Cheating of any form will lead to strict disciplinary action. Typical penalty would include Fail grade in the course.
- The assignment must be done individually, and no group submissions are allowed.
- All starter code files must be included in the submission, which must be uploaded on the gradescope without compressing the files.
- The names of files and method signatures must not be changed in the starter code.
- You should write the code in the portion of the starter code labelled **"To be filled in by the student"**.
- While submitting the code zip the folders Q1 and Q2 into a single file rather than zipping the folder containing Q1 and Q2. When we unzip, we should be able to see two folders named Q1 and Q2.

1 2-3-4 Trees

A 2–3–4 tree (also called a 2–4 tree) is a self-balancing data structure that is used to implement dictionaries to store <key, value> pairs. They can search, insert and delete in $O(\log n)$ time, if comparison between two keys takes constant time. Work your way through this assignment to learn how to implement a 2-3-4 tree and use it to solve an interesting problem!

1.1 Implementation

In this part, your task is to implement a `2-3-4 tree` data-structure using a class `Tree` defined in `Tree.java`. In this problem the keys will be strings and values will be positive integers. Example of <key, value> pairs: ("table", 5), ("the", 12).

The `Tree` class will comprise of the following components:

- **Instance variables:**
  1. `TreeNode root`: Points to the root of the `2-3-4 tree`, which will be set to null by the constructor, representing an empty `2-3-4 tree`.
- **Member functions:**
  1. `public void insert(String s)`: Takes a string `s` as a parameter and inserts the string at the leaf of the `2-3-4 tree`. Value of the string will be set to 1. We will ensure that the string `s` given as input always contains only lower case alphabets. Also, the insert function will be called only once for a particular string.
  2. `public boolean search(String s)`: Takes a string `s` as a parameter returns `true` if it is present in the `2-3-4 tree`, and `false` otherwise.
  3. `public boolean delete(String s)`: Takes a string `s` as a parameter and returns `false` if `s` is not present in the `2-3-4 tree`, otherwise removes the string `s` from the `2-3-4 tree` and returns `true`.

     (While deleting an internal string, the string chosen to perform swap should be the *predecessor of the string s*).
  4. `public int increment(String s)`: Takes a string `s` as a parameter and increments the value of string `s` by 1. The function returns the updated value. We will ensure that `increment(s)` is called only if `s` is present in the tree.
  5. `public int decrement(String s)`: Takes a string `s` as a parameter and decrements the value of string `s` by 1. The function returns the updated value. We will ensure that `decrement(s)` is called only if `s` is present in the tree and also that `value` of a node does not go below 1.
  6. `public int getHeight()`: Returns the current height of the (balanced) `2-3-4 Tree`.
  7. `public int getVal(String s)`: Returns the `value` of the string `s`.

Each node in the `Tree` will be an object of the class `TreeNode`. The class `TreeNode` comprises of the following:

- **Instance variables:**
    1. `ArrayList<String> s`: The list of strings stored at the node, initialised to empty array list by the constructor.
    2. `ArrayList<Integer> val`: The list of values of strings in the node containing, initialised to empty array list by the constructor.
    3. `ArrayList<TreeNode>` children: Pointers to children of the current node, initialised to empty array list by the constructor.
    4. `int height`: The height of the node, initialized as 1 by the constructor.
    5. `int count`: You will use this in part 1.2 .
    6. `String max_s`: You will use this in part 1.2 .
    7. `int max_value`: You will use this in part 1.2 .

NOTE: You will need to make sure that the tree is balanced at all times. So, perform the necessary operations while performing insertion and deletion. You will have to follow the standard conventions that at every node there can be 2,3 or 4 strings present. Some convention regarding borrowing while performing delete operation : If node if the leftmost node, then borrow from the just-right sibling. Otherwise every-time borrow from the just-left sibling.

```
Test case:
Tree obj = new Tree();
obj.insert("be");
obj.insert("ce");
obj.insert("de");
obj.insert("ae");

Current Tree :-
            be
          /    \
        ae     ce, de

obj.search("ce");    // returns true
obj.search("fe");    // returns false
obj.delete("ae");    // returns true

Current Tree :-
            ce
          /    \
        be      de

obj.getHeight();     // returns 2
obj.getVal("ce");    // returns 1
obj.increment("ce");
obj.getVal("ce");    // returns 2
```

You can create other helper functions if you like, but don't change the signature of the given functions. **You are not allowed to use any inbuilt/user-defined data structures for this part.**

1.2 Passage Processor using Augmented 2-3-4 Trees

In this part, you will learn a beautiful application of the `Tree` class you created.

Use your dictionary to read an English passage, and store words with corresponding frequencies. When you read a word, say "the", for the first time, you should insert it and set value (frequency) to 1. On each subsequent occurrence of "the", you should increment value by 1. We will use value and frequency interchangeably in this part of the question.

You also need to store some auxiliary information in variables `count` and `max` to answer queries of the following type in $O(\log n)$ time:

1. Given a pair of strings, say (`str1`, `str2`), report the number of strings (counted with multiplicity) that lie between `str1` and `str2` in the lexicographic ordering.

2. Given a pair of strings, say (`str1`, `str2`), report the string `s` lying between `str1` and `str2` in the lexicographic ordering that has maximum frequency.

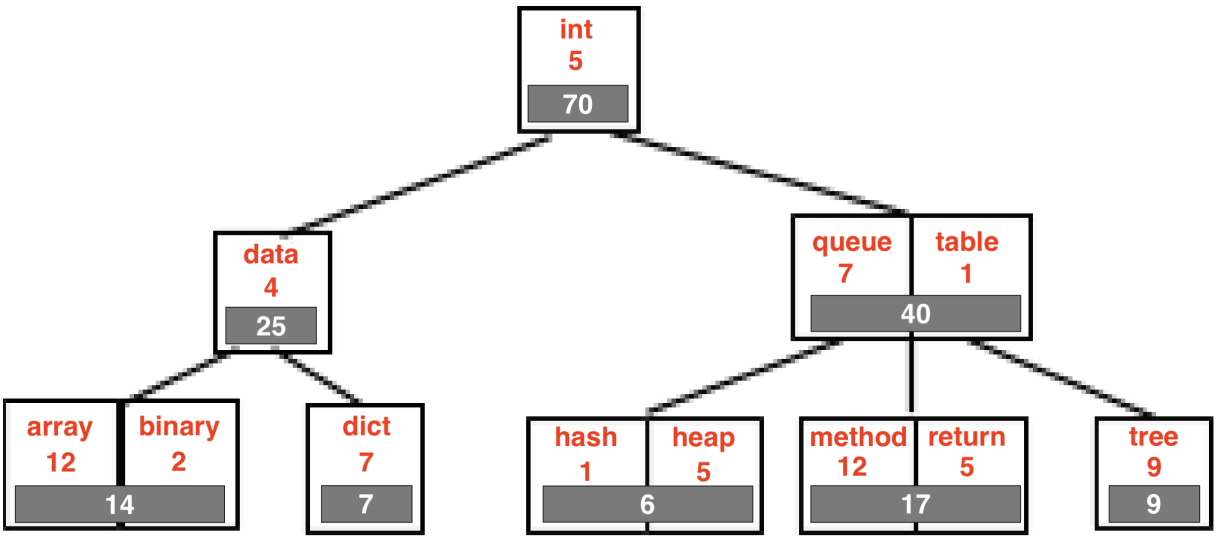

Figure 1: An example depicting a 2-3-4 tree. The fields highlighted in grey color provides a hint for the information that must be stored in the `count` variable.

Your task is to implement the following functions in Application.java file which extends `Tree` class :

1. `public void insert(String s)` : Takes a string `s` as a parameter and inserts string `s` at the leaf of the 2-3-4 tree. Value of the string must be set to 1. You will also need to handle the `count`, `max_s`, `max_value` variables.

2. `public int increment(String s)` : Takes a string `s` as a parameter and increments the `value` by 1. It should return the updated value. We will ensure that `increment(s)` is called only if `s` is present in the tree. You will also need to handle the `count`, `max_s`, `max_value` variables.

3. `public void buildTree(String fileName)` : Takes a string *fileName* as parameter which stores the path of the file which you need to read. You will need to build a `2-3-4 Tree` using the strings (space-separted) in the file. You need to use the insert and increment functions for this. This function will be called only once in the very beginning.

4. `public int cumulativeFreq(String s1, String s2)` : The function takes two strings `s1`, `s2`. It returns the total frequency of strings between `s1` and `s2` (inclusive). You need to use the count variable and perform the task in $O(\log n)$ time. We will make sure that `s1` is lexicographically smaller or same as `s2`.

5. `public String maxFreq(String s1, String s2)` : The function takes two strings `s1`, `s2`. It returns the string `s` lying between `s1` and `s2` (inclusive) in the lexicographic ordering that has maximum frequency. You need to use the `max_s` and `max_value` variables and perform the task in $O(\log n)$ time. If there are two or more strings with the maximum frequency, return the lexicographically smaller one. We will make sure that `s1` is lexicographically smaller or same as `s2`.

This question can be solved with various data structures. However, for an optimal solution, you require $O(\log n)$ time for search, insert and delete. You must use the 2-3-4 tree data structure from part 1 here. **You are not allowed to use any other inbuilt/user defined data structure for this question.**

NOTE: We will ensure that the file only contains spaces and strings which are lower case letters.

Example:

```
Test case:
Assume(hypothetically) that in your tree is such as the one given in the
   figure 1. Now, the following function calls are performed :

Inputs :
cumulativeFreq("dict","queue");          // returns 25
cumulativeFreq("array", "binary");       // returns 14
maxFreq("hash","return");                // returns "method"
maxFreq("array","binary");               // returns "array"


Explanation for cumulativeFreq:
Strings between dict and queue : dict, int, queue, hash, heap.
Adding the frequency of the strings gives answer as 7+5+6+7 = 25.
Note : You need to make use of "count" auxiliary variables so that you can
   do the computation in O(log n) time.

Explanation for maxFreq:
Strings between hash and return : hash, heap, queue, method, return.
String with the maximum frequency is : method.
Note : You need to make use of "max_s" and "max_value" variables so that
   you can do the computation in O(log n) time.
```

2 Heap Data Structure

A max-heap is a tree-based data structure that satisfies the property that for any given node x, if y is a parent of x, then the key of y is greater than or equal to the key of x. A heap is a useful data structure when it is necessary to repeatedly remove/find the object having maximum key in $O(1)$ time. Furthermore, the data-structure can be augmented to support insertion/search/deletion of keys in an efficient manner.

Work your way through this assignment to learn how to implement a max heap and to leverage them to solve an interesting problem of Poker game simulation!

2.1 Heap Implementation

In this part of the assignment, you will implement a max heap using binary tree. Such a binary tree is almost complete in the sense that each level of tree except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. You need to make sure that the heap property and structure is satisfied after all operations.
The heap data structure will be implemented through a class named `Heap` containing the following components.

- **Instance variables:**
  1. `protected Node root`: Points to the root of the binary tree used to implement max heap. Initially, it is null, representing an empty heap.
  2. `private int max_size`: Maximum number of nodes that could be inserted in the heap. We will make sure that number of nodes inserted in priority queue is bounded by this number.
  3. `protected Node[] nodes_array`: Array of size `max_size` where you have to store `Node` address corresponding to the keys. (The keys will be integer in the range [0, `max_size`−1]). So, `nodes_array[k]` should contain the address of `Node` corresponding to key `k`.
  4. `private static final String NullKeyException`: Exception to be thrown when the key is null.
  5. `private static final String NullRootException`: Exception to be thrown when the heap is empty.
  6. `private static final String KeyAlreadyExistsException`: Exception to be thrown when key already exists in the heap.

- **Constructor:**
  1. `public Heap(int max_size, int[] keys_array, int[] values_array) throws Exception`: Computes a max heap storing pairs <`keys_array`[i], `values_array`[i]>. The size of `keys_array` and `values_array` will be equal and bounded by `max_size`. Furthermore, the keys will be in range [0, `max_size`−1]. Note that the root element of the heap must correspond to the `value` which is largest in `values_array`. You must appropriately modify the `nodes_array`, and in case of duplicate keys in `keys_array` throw `KeyAlreadyExistsException`.

- **Member functions:**

  1. `public ArrayList<Integer> getMax() throws Exception`: Returns `ArrayList` storing the keys with maximum value in the heap. There could be multiple keys having same maximum value. If heap is empty, throw `NullRootException`. The time complexity of this function should be $O(\ell)$, where ℓ is the size of the output.
  2. `public void insert(int key, int value) throws Exception`: Insert a `Node` whose key is `key` and value is `value` in the heap and store the address of new node in `nodes_array[key]`. If `key` is already present in the heap then throw `KeyAlreadyExistsException`. The time complexity of this function should be $O(\log n)$, where n denotes the size of current heap.
  3. `public ArrayList<Integer> deleteMax() throws Exception`: Remove all nodes with the maximum value in the heap and returns `ArrayList` storing their keys. There could be multiple nodes having the same maximum value. If heap is empty, throw `NullRootException`. You must appropriately modify the `nodes_array`. The time complexity of this function should be $O(\log n)$, where n denotes the size of current heap.
  4. `public void update(int key, int diffvalue) throws Exception`: Update the heap by adding `diffvalue` to the `value` of the `Node` whose key is `key`. Note that `diffvalue` can be both positive as well as negative. If `key` doesn't exist in the heap, throw `NullKeyException`. The time complexity of this function should be $O(\log n)$, where n denotes the size of current heap.
  5. `public int getMaxValue() throws Exception`: Returns maximum value in the heap. If heap is empty, throw `NullRootException`. The time complexity of this function should be $O(1)$.
  6. `public ArrayList<Integer> getKeys() throws Exception`: Returns `ArrayList` storing keys of the nodes in the heap. If heap is empty, throw `NullRootException`. The elements in returned ArrayList can be in any arbitrary ordering. The time complexity of this function should be $O(n)$, where n is the size of current heap.

Each node in `Heap` will be an object of the class `Node`. The class `Node` will comprise of:

- **Instance variables:**

  1. `public int key`: The key stored in the node.
  2. `public int value`: The value stored in the node.
  3. `public Node left`: Points to the left child of node. For leaf node it is `null`.
  4. `public Node right`: Points to the right child of node. For leaf node it is `null`.
  5. `public Node parent`: Points to the parent of node. For root, it is `null`.
  6. `public int height`: Height of the sub-tree rooted at this node. For leaf node, it is 1.
  7. `public boolean is_complete`: Whether the binary sub-tree rooted at this node is complete or not.

```
Sample Test Case 1 :-

int[] keys = new int[]{ 3, 5, 10, 15, 24 };
int[] values = new int[]{ 10, -4, 15, -100, 175};
Heap heap = new Heap(100, keys, values);  // Initialization

Current heap (One possibility of heap):

                    (k:24,v:175,addr:0x0010)
                    /                      \
        (k:10,v:15,addr:0x0111)     (k:3,v:10,addr:0x1011)
        /                      \
(k:15,v:-100,addr:0x0101)  (k:5,v:-4,addr:0x0001)

heap.nodes_array[24] = 0x0010; heap.nodes_array[10] = 0x0111;
heap.nodes_array[15] = 0x0101; heap.nodes_array[3] = 0x1011;
heap.nodes_array[5] = 0x0001;

heap.getMax();    // returns ArrayList<Integer> arr = {24}

heap.insert(1, 5);     // insert()

Current heap (One possibility of heap):

                    (24,175,0x0010)
                    /             \
        (10,15,0x0111)             (3,10,0x1011)
       /             \             /
(15,-100,0x0101)  (5,-4,0x0001)   (1,5,0x1001)

heap.nodes_array[24] = 0x0010; heap.nodes_array[10] = 0x0111;
heap.nodes_array[15] = 0x0101; heap.nodes_array[5] = 0x0001;
heap.nodes_array[3] = 0x1011; heap.nodes_array[1] = 0x1001;

heap.update(1, 170);   // update()

Current heap (One possibility of heap):

                   (24,175,0x0010)
                    /             \
        (10,15,0x0111)             (1,175,0x1011)
       /             \             /
(15,-100,0x0101)  (5,-4,0x0001)   (3,10,0x1001)

heap.nodes_array[24] = 0x0010; heap.nodes_array[10] = 0x0111;
heap.nodes_array[15] = 0x0101; heap.nodes_array[5] = 0x0001;
heap.nodes_array[3] = 0x1001; heap.nodes_array[1] = 0x1011;
```

```
heap.deleteMax();  // returns ArrayList<Integer> arr = {24,1}

Current heap (One possibility of heap):

                    (k:10,v:15,addr:0x0010)
                    /                      \
        (k:5,v:-4,addr:0x0111)     (k:3,v:10,addr:0x1011)
        /
(k:15,v:-100,addr:0x0101)

heap.nodes_array[24] = null; heap.nodes_array[5] = 0x0111;
heap.nodes_array[15] = 0x0101; heap.nodes_array[10] = 0x0010;
heap.nodes_array[3] = 0x1011; heap.nodes_array[1] = null;

heap.getMaxValue();   // returns integer 15
heap.getKeys();     // returns ArrayList<Integer> arr = {10,5,15,3}
heap.insert(3,7800); // throw KeyAlreadyExistsException as key=3 is in heap
heap.update(1,1587); // throw NullKeyException as key=1 is not in heap
```

```
Sample Test Case 2 :-

int[] keys = new int[]{ 2, 10, 2, 5};
int[] values = new int[]{ -4000, 1000, 105, 0};
Heap heap = new Heap(50, keys, values);
// throw KeyAlreadyExistsException as key=2 is duplicate in array keys
```

```
Sample Test Case 3 :-

int[] keys = new int[]{ 150 };
int[] values = new int[]{ 10 };
Heap heap = new Heap(200, keys, values); // Initialization
Current heap : (k:10,v:15,addr:0x0000)
               heap.nodes_array[10] = 0x0000;
heap.deleteMax();        // returns ArrayList<Integer> arr = {10}
Current heap : null
              heap.nodes_array[10] = null;
heap.getMax();    // throw NullRootException as heap is empty
heap.deleteMax();  // throw NullRootException as heap is empty
heap.getKeys();    // throw NullRootException as heap is empty
heap.getMaxValue();    // throw NullRootException as heap is empty
```

You can create other helper functions (keep them private) but don't change the signature of the given functions. Note that you are not allowed to use any inbuilt/user-defined data structures for this part.

2.2 Poker Game Simulator

Poker is a family of comparing card games in which players bet on the set of cards they received. Player who has got best collection of cards wins the game and gets all the bets and other players lose their bets. Your task is to use `Heap` data structure, you need to solve the design a Poker Game Simulator.
In a city named `'COL106'`, citizens decides to earn extra income by using their knowledge of poker. The population of the city is `city_size` and citizens have their unique id numbered as `0,1,...,city_size-1`. There is a `Poker arena` where citizens come and play poker. Each citizen has `$100,000` initially.
When a citizen enter the `Poker arena`, they come up with their maximum loss that they can bear and maximum profit they wish to get. When a citizen has suffered loss more than it's maximum loss or has gained more profit than it's maximum profit, he/she leaves the `Poker arena`.
In each game, a set of m players play and each player bids his/her amount (m can vary for each game and $1 < m \leq n$, where n is the number of citizens present in `Poker arena` currently). The players playing a poker game will be present in the `Poker arena`. Winner of a poker game is decided using a random generator and the winner gets all the money raised by other players and other players lose their bids.
Number of citizens in `Poker arena` will change when an existing citizen leave the `Poker arena` as he/she has reached his/her maximum profit/maximum loss or when a citizen enters the `Poker arena`. Any citizen could enter the `Poker arena` irrespective of whether he/she has entered the `Poker arena` earlier or not.

You need to implement methods for the class `Poker`. The only global data structure you are allowed to use is `Heap` data structure computed in 2.1. Furthermore, the number of instances allowed of `Heap` data structure is at max **four**. You may use `ArrayList` locally inside member functions.

The class `Poker` will contain the following components.

- **Instance variables:**
    1. `private int city_size`: Population of the city `COL106`.
    2. `public int[] money`: An array of size `city_size` denoting the current money each citizen has. Initially, this is `100,000` for all citizens.
- **Constructor:**
    1. `public Poker(int city_size, int[] players, int[] max_loss, int[] max_profit)`: The variable `city_size` denotes the population of the city `COL106`. The array `players` contains the id of the citizens who are first to come to the `Poker arena` to play Poker, and the size of the array is bounded by `city_size`. The value `max_loss[i]` denotes the maximum loss player `players[i]` can bear. Similarly, `max_profit[i]` denotes the maximum profit player `players[i]` wishes to get,

before he/she decides to exit the arena. The array `players` will contains distinct values. You can initialize the `Heap` data structure (if needed).

The time complexity of this function must be $O(n)$ where n denotes the size of array `players`.

- **Member functions:**

1. `public void initMoney()`: Initialize the `money` array. Do not change anything in this function.
2. `public ArrayList<Integer> Play(int[] players, int[] bids, int winnerIdx)`: The array `players` stores the id of the citizens who will play the poker game. `winnerIdx` is index of player who will win the game, so player `players[winnerIdx]` will win this game. It will be ensured that the players who have come to play the poker game will be present in the `Poker arena`. Size of `players` and `bids` will be equal and `players` will contain distinct values. The value `bids[i]` denotes the bid made by player `players[i]` in this game. You must update the money of the players who has played in this game in array `money`, and return `ArrayList` storing all players who will leave the poker arena after this game. (In case no player leaves, return an empty `ArrayList`).

   The time complexity of this function must be $O(m \log n)$ where m denotes the size of `players` and n denotes the number of citizens present in the `Poker arena` currently.
3. `public void Enter(int player, int max_loss, int max_profit)`: A citizen with id `player` enters the `Poker arena`. Here `max_loss` denotes the (new) maximum loss the `player` can bear, and `max_profit` is (new) maximum profit `player` wants to get. It will be ensured that `player` is not in the `Poker arena`. In particular, a citizen enters only if he/she had either left the `Poker arena` earlier, or was not added to the `Poker arena` during the initialization phase. The Money that this citizen has will be same as the money that he/she had before leaving the `Poker arena`. If the citizen is entering for the first time, then the money he/she has will be 100,000.

   The time complexity of this function must be $O(\log n)$ where n is the number of citizens in the `Poker arena` currently.
4. `public ArrayList<Integer> nextPlayersToGetOut()`: Returns `ArrayList` storing the id of citizens who are likely to get out of `Poker arena` in the next game. Citizens in `Poker arena` are ranked based on score and you have to return the citizens who have minimum score. The score for citizen with id `idx` who entered the `Poker arena` with money `M[idx]`, maximum profit `max_profit[idx]` and maximum loss `max_loss[idx]` is calculated as :

   `score[idx]` = `minimum(max_profit[idx]+M[idx]-money[idx], money[idx]+max_loss[idx]-M[idx])`.

The time complexity of this function should be $O(\ell)$, where ℓ denotes the size of the output.

5. `public ArrayList<Integer> playersInArena()`: Returns `ArrayList` storing id of citizens present in the `Poker arena`.

   The time complexity of this function must be $O(n)$ where n is the number of citizens in the `Poker arena` currently.

6. `public ArrayList<Integer> maximumProfitablePlayers()`: Returns `ArrayList` storing id of citizens who has gained maximum profit. You need to include all citizens present in the city and not only those currently present in the `Poker arena`. Here, profit for citizen with id `idx` is calculated as `money[idx]-100000`.

   The time complexity of this function should be $O(\ell)$, where ℓ denotes the size of the output.

7. `public ArrayList<Integer> maximumLossPlayers()`: Returns `ArrayList` which stores id of citizens who has suffered maximum loss. You need to include all citizens present in the city and not only those currently present in the `Poker arena`. Here, loss for citizen with id `idx` is calculated as `100000-money[idx]`.

   The time complexity of this function should be $O(\ell)$, where ℓ denotes the size of the output.

Note:

1. Whenever return type is ArrayList, ordering of elements in ArrayList doesn't matter.
2. `max_loss` for a citizen with id `idx` will be in range [0, `money[idx]`] always.
3. `bids` for a citizen with id `idx` will be in range [0, `money[idx]`] always.
4. `key` will be in range [0, `max_size` $-$ 1] for heap in all the methods.

```
Sample Test Case :-

int[] players = new int[]{ 4, 72, 12, 17, 1 };
int[] max_loss = new int[]{ 4100, 2312, 175, 9007, 63125 };
int[] max_profit = new int[]{ 7500, 6010, 1700, 2100, 31275 };
Poker p = new Poker(100, players, max_loss, max_profit);   // Initialization
Id :          4   |  72  |  12  |  17  |   1
max_loss :   4100  | 2312 |  175 | 9007 | 63125
max_profit : 7500  | 6010 | 1700 | 2100 | 31275

int[] pokerPlayers = new int[]{ 4, 17, 12 };
int[] bids = new int[]{2000, 1000, 500};
int winnerIdx = 1                        // citizen id 17 wins.
p.Play(pokerPlayers, bids, winnerIdx); // returns ArrayList<Integer> arr =
   {12, 17}
Current players in Poker Arena : {72, 1, 4}
money[4] = 98000; money[17] = 102500; money[12] = 99500;
Explanation: citizen 17 has gain profit of 2000+500=2500 and maximum profit
   that citizen 17 wish to get is 2100. Similarly, citizen 12 lose 500 and
   maximum loss citizen 12 wants to bear is 175. So, citizen 17 and 12
   leave the Poker arena.

p.Enter(3, 1500, 1500);  // Citizen with id 3 enters Poker arena

p.nextPlayersToGetOut(); // returns ArrayList<Integer> arr = {3}
Explanation: Let M[i] be the money citizen with id i enters in Poker arena.
   In this case, M[i] = 100000 for all players as initially all citizens
   have 100000.
ID  |   max_loss  | max_profit  | money  |    M    |  score
4   |    4100     | 7500        | 98000  | 100000  |  2100
72  |    2312     | 6010        | 100000 | 100000  |  2312
1   |    63125    | 31275       | 100000 | 100000  |  31275
3   |    1500     | 1500        | 100000 | 100000  |  1500
So, citizen with id 3 has minimum score.

p.Enter(12, 250, 800);   // citizen with id 12 enters Poker arena
money[12] = 99500;  // Money when citizen with id 12 left the Poker arena

Id :          4   |  72    |  12   |  3    |   1
max_loss :   4100  | 2312   |  250  | 1500  | 63125
max_profit : 7500  | 6010   |  800  | 1500  | 31275
money :      98000 | 100000 | 99500 | 100000| 100000
int[] pokerPlayers = new int[]{ 12, 3, 1, 4 };
int[] bids = new int[]{200, 100, 100, 200};
int winnerIdx = 0                        // citizen id 12 wins.
p.Play(pokerPlayers, bids, winnerIdx); // returns ArrayList<Integer> arr={}
money[4] = 97800; money[1] = 99900; money[3] = 99900; money[12] = 99900
```

```
Explanation :
1. Citizen with id 12 wins 100+100+200=400 which is less than his/her
   max_profit=800 and so, he/she stays in the poker arena. In this case as
   he/she has re-entered the Poker arena, so it's previous outing
   loss/profit will not be included.
2. Citizen with id 4 lost 200 and so, it's overall loss is 2200 which is
   less than his/her maximum loss of 4100 and so, he/she stay in Poker
   arena.
3. Citizen with id 1 lost 100 which is less than his/her maximum loss of
   63125 and so, he/she stay in Poker arena. Similar situation with
   citizen id 3.

p.nextPlayersToGetOut(); // returns ArrayList<Integer> arr = {12}
Explanation :
ID  |   max_loss  | max_profit  | money  |    M    |  score
4   |    4100     | 7500        | 97800  | 100000  |  1900
72  |    2312     | 6010        | 100000 | 100000  |  2312
1   |    63125    | 31275       | 99900  | 100000  |  31375
3   |    1500     | 1500        | 99900  | 100000  |  1400
12  |    250      | 800         | 99900  |  99500  |  400
So, citizen with id 3 has minimum score.

p.playersInArena(); // returns ArrayList<Integer> arr = {12, 4, 1, 72, 3}

ID |   money   |  profit  | loss
4  |   97800   | -2200    | 2200
72 |   100000  |  0       | 0
1  |   99900   | -100     | 100
3  |   99900   | -100     | 100
12 |   99900   | -100     | 100
17 |   102500  |  2500    | -2500

p.maximumProfitablePlayer();   // returns ArrayList<Integer> arr = {17}
p.maximumLossPlayer();         // returns ArrayList<Integer> arr = {4}
```