HY240: Data Structures

Winter Semester - Academic Year 2020-2021 Instructor:

Panagiota Fatourou

Programming Work - Part 1

Delivery date: Monday, November 16, 2020, time 09:59 am

Delivery method: Using the turnin program. Information on how the turnin program works is provided on the course website.

Photo: https://upload.wikimedia.org/wikipedia/en/7/72/AmongUs_CoverArt.jpg

General Job Description

In this task you are asked to implement a simulation of a batch of the game Among Us.

«Among Us is an online multiplayer social deduction game, developed and published by American game studio InnerSloth and released on June 15, 2018. The game takes place in a space-themed setting where players each take on one of two roles, most being Crewmates , and a predetermined number being Impostors. The goal for the Crewmates is to identify the Impostors, remove them from the game, and complete tasks around the map; the Impostors' goal is to covertly sabotage the Crewmates and remove them from the game before they complete all their tasks. Through a plurality vote, players believed to be Impostors may be removed from the game. If all Impostors are removed from the game or all tasks are completed, the Crewmates win; if there are an equal number of Impostors and Crewmates, or if a critical sabotage goes unresolved, the Impostors win. » (Wikipedia)

Αξίζει να σημειωθεί, ότι για τους εκπαιδευτικούς σκοπούς του μαθήματος ορισμένα σημεία της εργασίας ενδέχεται να αποκλίνουν από το πραγματικό παιχνίδι.

Αναλυτική Περιγραφή Ζητούμενης Υλοποίησης

Κάθε παίκτης (player), εξωγήινος (impostor) ή άνθρωπος (crewmate), περιγράφεται από ένα μοναδικό αναγνωριστικό. Πληροφορίες για τους παίκτες αποθηκεύονται σε μία **διπλά συνδεδεμένη κυκλική λίστα με κόμβο φρουρό (Σχήμα 1), που λέγεται λίστα παικτών.** Κάθε στοιχείο της λίστας είναι ένα

struct Player type and corresponds to one player. For each player, there is an int variable (called is_alien), which indicates whether the player is an alien or a human, and an int variable (called evidence), which indicates whether the player will be considered a suspect in subsequent polls. Please note that, although each player's type is listed, we believe that the other players do not have access to each player's is_alien field and therefore cannot find out if the player is an alien or a human.

More specifically, a struct Player entry has the following fields:

- **pid:** Identifier (int type) that uniquely characterizes the player.
- **is_alien:** Flag (type int) that identifies whether the player is an alien (impostor) or a human (crewmate).
- **evidence:** Integer that describes the degree to which a player is considered a suspect (type int).
- **prev:** Index (type *struct Player)* pointing to the previous node of the player list.
- **next:** Index (type *struct Player)* pointing to the next node in the player list.
- **tasks_head:** Index (type *struct Task)* pointing to the first element of a node list

  guard (called the player's to-do list and described below), which contains information about the tasks to be assigned to the player.
- **tasks_sentinel:** Index (type *struct Task)* pointing to the guard node of the player task list.

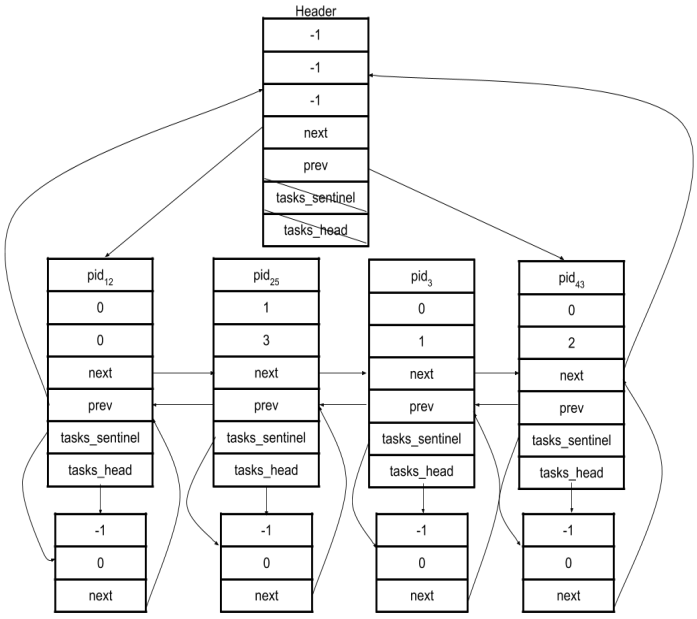

Figure 1. The list of players that is double-linked, circular, with a guard node and not sorted. The second field indicates whether a player is an alien or not, and the third field indicates whether a player is a suspect. To simplify the schema, we assume that the tasks field has the value NULL in all

items in the list above.

Each player has different tasks to complete. So for every player there is one **simply linked sorted list with a guard node, called the player task list.** For each task of the player there is a unique identifier (tid) and a degree of difficulty (difficulty). Each player's to-do list is sorted by the degree of difficulty of each task it contains.

All the work that needs to be done to defeat people is first introduced in one **just linked list,** which is also classified according to the degree of difficulty. This list is called **general to-do list (Figure 2).** Once all tasks are entered in the general task list, they will eventually be shared in the players task lists (**Figure 3).**

Each node in the general to-do list and to-do list of a specific player is a struct Task entry with the following elements:

- **time:** Identifier (int type) that uniquely characterizes each task.
- **difficulty:** The degree of difficulty (type int) of each task. There are 3 difficulty levels (denoted by the numbers 1, 2 and 3).
- **next:** Index (type *struct Task)* pointing to the next node.

There is a struct, type General_List that contains the following fields:

- **GL:** Index in Task struct that points to the first item in the general to-do list.
- **tasks_count [3]:** Table of three integers. Count [i] counts how many grade jobs (i + 1) are in the general to-do list.

There is also a global variable Total_Tasks that stores the total number of jobs stored in the to-do list.

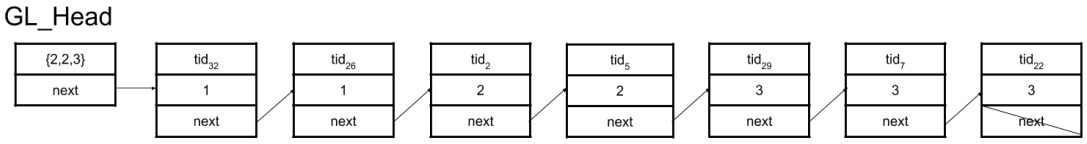

Figure 2. The general to-do list. This is a simple linked list that is sorted by the difficulty of each task.

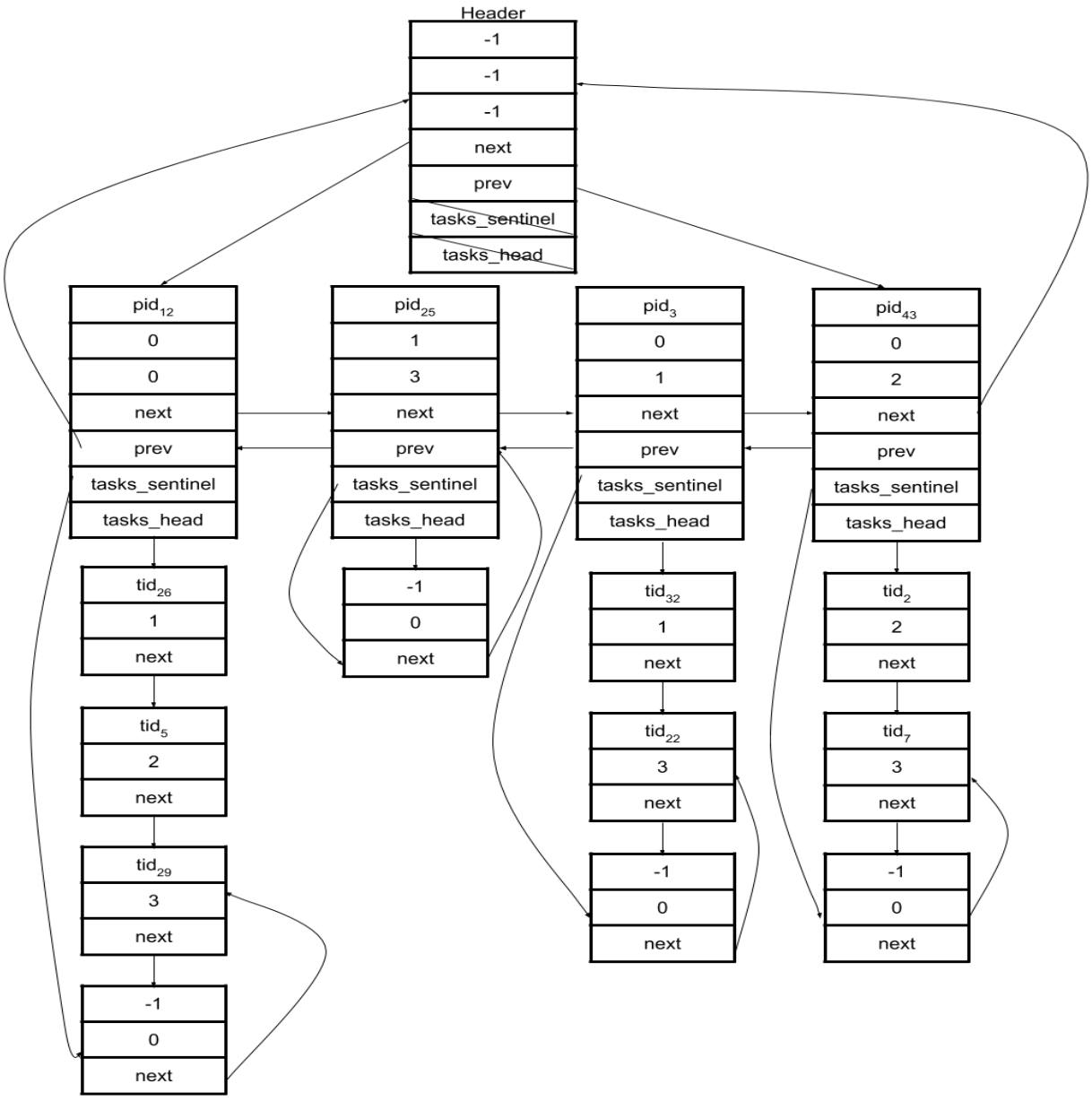

Figure 3. The double-linked circular list with a player guard node, which is unclassified after assigning tasks to players. Notice that each player's to-do list is sorted as well as the general to-do list based on the difficulty of each task.

Once the jobs have been registered to the people (crewmates), then they start implementing them so that they can win the game. Each time a person completes a task then it is removed from their to-do list and inserted into a **completion stack (Figure 4).** Each node of the stack is one

struct Task type entry (ie it has the same type of data as the general to-do list and player to-do lists). There is a struct, type Head_Completed_Task_Stack that contains the following fields:

- **Head:** Pointer to a Task struct that points to the top element of the stack.
- **count:** Integer that stores the number of items in the stack.

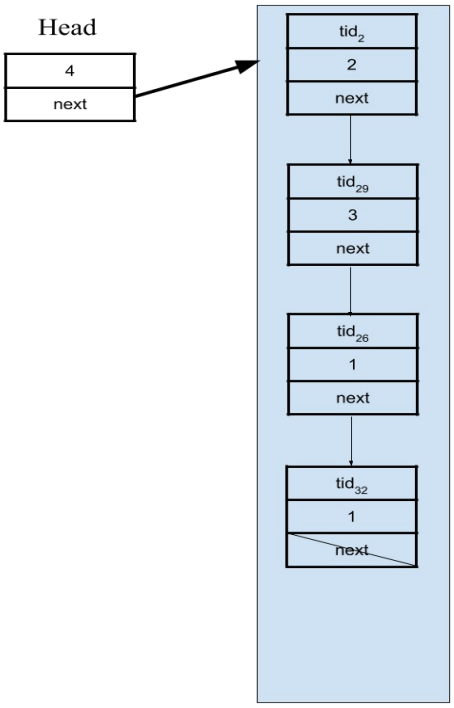

Figure 4. The stack with the completed tasks. If it fills up then people won the game.

Program Mode

The program to be created should be executed by calling the following command:

<executable> <input-file>

where <executable> is the name of the executable program file (eg a.out) executable> is the name of the executable file of the program (eg a.out) and <executable> is the name of the executable file of program (eg a.out) input-fle> is the name of an input file (eg testile) that contains events of the following formats:

P <pid> <is_alien>

An insert Player that indicates the insertion of a player with an <executable> ID is the name of the program executable file (eg a.out) pid> in the game. In this case an item (corresponding to the new player) will be added to the player list. The evidence and is_alien fields of this item must be initialized to 0 and <executable> is the name of the executable file of the program (eg a.out) is_allien>, respectively (if <executable> is its name executable file of the program (eg a.out) is_allien> is 1 then the player is an alien, otherwise he is a human, crew member). The player's to-do list is initialized to contain only him

her guard node. Assume that <executable> is the name of the program executable file (eg a.out) pid> is unique (ie all <executable> is the name of the program executable file (eg a.out) related to type P events in test_file are different). Implement the introduction in the most efficient way. At the end of such an event, the program should print the following information:

P <pid> <is_alien>

Players = <pid $_1$: is_alien $_{1>}$ < pid $_2$: is_alien $_{2>}$ … <Pid $_n$: is_alien $_{n>}$

DONE

where n is the number of nodes in the player list and for each i ∈ {1, ..., n}, <pid {1, ..., n}, <executable> is the name of the executable file of the program (π .x. a.out) pid $_{i>}$ is the player ID that corresponds to the i-th node of this list.

T <tid> <difficulty>

An Insert Task that indicates the creation of a task with an <executable> ID is the name of the program executable file (eg a.out) tid> and a degree of difficulty <executable> is the name of the program executable file (e.g. .x. a.out) difficulty>. In this case, a new item corresponding to this task will be added to the general task list. The <executable> field is the name of the program executable file (eg a.out) pid> of the new item is initially undefined (will be specified by another event later). Also, the appropriate fields of the struct Genenal_List should be updated correctly. Upon completion of such an event, the program should print the following information:

T <tid> <difficulty>

Tasks = <tid $_1$, difficulty $_{1>}$ < time $_2$, difficulty $_{2>}$ … <Tid $_n$, difficulty $_{n>}$

DONE

where n is the number of nodes in the general task list and for each i ∈ {1, ..., n}, <pid {1, ..., n}, <executable> is the name of the executable file of the program (e.g. a.out) tid $_{i>}$ is the job ID that corresponds to the i-th node of this list.

D

Distribute Tasks event that indicates the assignment of tasks to players. In this case, the tasks in the general to-do list should be distributed to the players following a round robin algorithm. Specifically, process i in the to-do list should be assigned to the player (i% n) where n is the number of items in the to-do list. This fact should have a time complexity O (m), where m is the number of items in the general to-do list. To achieve this time complexity, use the next pointer of the guard node to remember the last item in each player's to-do list. At the end of such an event, the program should print the following information:

D

Player $_{1}$ = < time $_{1.1}$, difficulty $_{1.1>}$, < time $_{1,2}$, difficulty $_{1,2>}$ … <Tid $_{1, mn}$, difficulty $_{1, m1>}$

Player $_{2}$ = < time $_{2.1}$, difficulty $_{2.1>}$, < time $_{2.2}$, difficulty $_{2.2>}$ … <Tid $_{2, mn}$, difficulty $_{2, m2>}$

…

Player $_{n}$ **=** **<** **time** $_{n,1}$, **difficulty** $_{n,1}$**>,** **<** **time** $_{n,2}$, **difficulty** $_{n,2}$**>** **…** **<Tid** $_{n,mn}$, **difficulty** $_{n,mn}$**>**

DONE

where for each i, 1≤i≤ n, m_i is the number of nodes in the i-th player's task list, and
for each j∈ {1, ..., n}, <pid {1, ..., m_i}, $time_{i,j}$ and $difficulty_{i,j}$ is the ID of each task and its degree of difficulty respectively, which corresponds to the j-th node of the i-bone task list
player.

I <pid> <difficulty>

Implement Task event, which indicates that the player with the <executable> ID is the name of the program executable file (eg a.out) pid>, executes a task with a degree of difficulty <executable> is the name of the executable program file (eg a.out) difficulty>. If there is no task with this degree of difficulty, then the player completes a task from those that have the most difficulty in his to-do list. In this case the following actions should be performed. Initially, you will be looking for the specific player in the player list. Once you have found the specific player, you will be looking for a task w <eecutable> is the name of the program executable file (eg a.out) difficulty> in the player task list. Subsequently, you will delete w from the <executable> player task list is the name of the program executable file (eg a.out) pid> (so that the list remains sorted after the event) and will be inserted a node for w in the stack with the implemented tasks (correctly updating the count field of the struct Stack). Upon completion of such an event, the program should print the following information:

I <pid> <difficulty>

Player $_{1}$ **=** **<** **time** $_{1.1}$, **difficulty** $_{1.1}$**>,** **<** **time** $_{1,2}$, **difficulty** $_{1,2}$**>** **…** **<Tid** $_{1,mn}$, **difficulty** $_{1,mn}$**>**

Player $_{2}$ **=** **<** **time** $_{2.1}$, **difficulty** $_{2.1}$**>,** **<** **time** $_{2.2}$, **difficulty** $_{2.2}$**>** **…** **<Tid** $_{2,mn}$, **difficulty** $_{2,mn}$**>**

…

Player $_{n}$ **=** **<** **time** $_{n,1}$, **difficulty** $_{n,1}$**>,** **<** **time** $_{n,2}$, **difficulty** $_{n,2}$**>** **…** **<Tid** $_{mn,mn}$, **difficulty** $_{mn,mn}$**>**

DONE

where for each i, 1≤i≤ n, mi is the number of nodes in the i-th player's to-do list, and
for each j∈ {1, ..., n}, <pid {1, ..., mi}, $tid_{i,j}$ and $difficulty_{i,j}$ is the ID of each task and its degree of difficulty respectively, which corresponds to the j-th node of the i-bone task list
player.

E <pid>

Eject Player type that indicates the touch of the player with <executable> ID is the name of the program executable file (eg a.out) pid> from an alien and its removal from the spaceship (without anyone noticing other player). In this case, the player with the <executable> ID is the name of the executable file of the program (eg a.out) pid> is removed from the list of players and all his tasks are transferred to the player who has the least work at that time. The player <executable> is the name of the executable file of the program (eg a.out) pid_min> with the least tasks will be as follows: You will have to cross the list of players and for each player count how many items it has his to-do list, holding an auxiliary index at the top of that to-do list that has been found to have the fewest items at any given time. Finding <executable> is the name of the program executable file (eg a.out) pid_min> should therefore be done by scrolling through the player list (and each player to-do list). THE

<executable> player to-do list is the name of the program executable file (eg a.out) pid_min> must be sorted after merging with <executable> player's to-do list is the name of the executable file program (eg a.out) pid>. Once <executable> is found is the name of the executable file of the program (eg a.out) pid_min>, the merging of the two lists should be performed at time O ($n_{1+} n_{2)}$, where n1 is the number of items in the player's to-do list <executable> is the name of the program executable file (eg a.out) pid1> and n2 is the number of items in the player's to-do list <executable> is the name of the executable file of the program (eg a.out) pid_min>. At the end of such an event, the program should print the following information:

E <pid>

Player $_{1}$ **= < time** $_{1.1,}$ **difficulty** $_{1.1>, <}$ **time** $_{1,2,}$ **difficulty** $_{1,2>}$ **… <Tid** $_{1, mn,}$ **difficulty** $_{1, mn>}$

Player $_{2}$ **= < time** $_{2.1,}$ **difficulty** $_{2.1>, <}$ **time** $_{2.2,}$ **difficulty** $_{2.2>}$ **… <Tid** $_{2, mn,}$ **difficulty** $_{2, mn>}$

…

Player $_{n}$ **= < time** $_{n, 1,}$ **difficulty** $_{n, 1>, <}$ **time** $_{n, 2,}$ **difficulty** $_{n, 2>}$ **… <Tid** $_{mn, mn,}$ **difficulty** $_{mn, mn>}$

DONE

where for each i, 1≤i≤ n, mi is the number of nodes in the i-th player's to-do list, and
for each j∈ {1, ..., n}, <pid {1, ..., mi}, tid $_{i, j}$ and difficulty $_{i, j}$ is the ID of each task and its degree of difficulty respectively, which corresponds to the j-th node of the i-bone task list
player.

W <pid> <pid $_{a>}$ **< number_witnesses>**

A Witness Ejection event that indicates the player's touch with an <executable> ID is the name of the program executable file (eg a.out) pid> by extraterrestrial with <executable> ID is the name of the program executable file (eg a.out) pid $_{a>}$ and removing the player <executable> is the name of the executable program file (eg a.out) pid> from the spaceship. However, in this case, removing the player <executable> is the name of the program executable file (eg a.out) pid> see it <ex (crewmates). As in the E event, in this case, the player with the <executable> ID is the name of the program executable file (eg a.out) pid> is removed from the player list and all its tasks are transferred to the player who has the fewest tasks at the moment. However, in addition the following should be done: scroll through the list of players and find the alien with the <executable> ID is the name of the executable file of the program (eg a.out) pid $_{a>}$ and update the evidence field for it by adding the number <executable> is the name of the executable file of the program (eg a.out) number_witnesses>. At the end of such an event, the program should print the following information:

W <pid> <pid $_{a>}$ **< number_witnesses>**

<Player $_{1,}$ **evidence** $_{1>}$ **= < time** $_{1.1,}$ **difficulty** $_{1.1>, <}$ **time** $_{1,2,}$ **difficulty** $_{1,2>}$ **… <Tid** $_{1, mn,}$ **difficulty** $_{1, mn>}$

<Player $_{2,}$ **evidence** $_{2>}$ **= < time** $_{2.1,}$ **difficulty** $_{2.1>, <}$ **time** $_{2.2,}$ **difficulty** $_{2.2>}$ **… <Tid** $_{2, mn,}$ **difficulty** $_{2, mn>}$

…

<Player $_{n,}$ **evidence** $_{n>}$ **= < time** $_{n, 1,}$ **difficulty** $_{n, 1>, <}$ **time** $_{n, 2,}$ **difficulty** $_{n, 2>}$ **… <Tid** $_{mn, mn,}$ **difficulty** $_{mn, mn>}$

DONE

where for each i, 1 ≤ i ≤ n, mi is the number of nodes in the i-th player's to-do list
and evidence $_{i}$ is, and for every j∈ {1, ..., n}, <pid {1, ..., mi}, tid $_{i, j}$ and difficulty $_{i, j}$ is the identifier of each task and its degree of difficulty respectively, corresponding to the j-th node of the list of
i-bone player tasks.

S <number_of_tasks> <pid>

A Sabbotage event that suggests an alien removes <executable> is the name of the program executable file (eg a.out) number_of_tasks> tasks from the completion stack and redistributes them to players. When this is done, <executable> is the name of the executable file of the program (eg a.out) number_of_tasks> pop from the task stack (by properly updating the count field of the struct Stack). Each task removed from the stack is assigned to a player using the following algorithm: First, we look for the player with the <executable> ID is the name of the program executable file (eg a.out) pid> in the player list . The first task is assigned to player p located ⌊ number_of_tasks / 2 ⌋ positions to the left of the player with the <executable> ID is the name of the program executable file (eg a.out) pid>. If player p is alien then the task is assigned to the first node to the right of non-alien p. Each subsequent task is assigned to the next nodes of p (following next pointers) that are not extraterrestrial. In this way, the ⌊ number_of_tasks / 2 ⌋ previous nodes of p and at least the ⌊ number_of_tasks / 2 ⌋ next nodes of p will be examined to assign them a task to each. At the end of such an event, the program should print the following information:

S <number_of_tasks> <pid>

Player $_1$ **= < time** $_{1.1}$**, difficulty** $_{1.1}$**>, < time** $_{1,2}$**, difficulty** $_{1,2}$**> … <Tid** $_{1, mn}$**, difficulty** $_{1, mn}$**>**

Player $_2$ **= < time** $_{2.1}$**, difficulty** $_{2.1}$**>, < time** $_{2.2}$**, difficulty** $_{2.2}$**> … <Tid** $_{2, mn}$**, difficulty** $_{2, mn}$**>**

…

Player $_n$ **= < time** $_{n, 1}$**, difficulty** $_{n, 1}$**>, < time** $_{n, 2}$**, difficulty** $_{n, 2}$**> … <Tid** $_{mn, mn}$**, difficulty** $_{mn, mn}$**>**

DONE

where for each i, 1≤i≤ n, mi is the number of nodes in the i-th player's to-do list, and
for each j∈ {1, ..., n}, <pid {1, ..., mi}, tid $_{i, j}$ and difficulty $_{i, j}$ is the ID of each task and its degree of difficulty respectively, which corresponds to the j-th node of the i-bone task list
player.

V <pid> <vote_evidence>

Voting event, which indicates the voting and removal of a player (impostor or crewmate) from the spaceship. In this case, the following happens. Initially, the player with the <executable> ID is the name of the executable file of the program (eg a.out) pid> is considered suspicious (because we consider that he is the one who makes the most noise during the voting) and so the evidence field of the struct corresponding to the player list increases by <executable> is the name of the executable file of the program (eg a.out) vote_evidence>. Next, you need to scroll through the list of players, find the player with the most evidence, and remove him. The list of player tasks to be dropped from the spaceship will be assigned to the player with the fewest tasks (as discussed in events E and W). Upon completion of such an event, the program should print the following information:

V <pid> <vote_evidence>

<Player $_1$**, evidence** $_{1>}$ **= < time** $_{1.1}$**, difficulty** $_{1.1}$**>, < time** $_{1,2}$**, difficulty** $_{1,2}$**> … <Tid** $_{1, mn}$**, difficulty** $_{1, mn}$**>**

<Player $_2$**, evidence** $_{2>}$ **= < time** $_{2.1}$**, difficulty** $_{2.1}$**>, < time** $_{2.2}$**, difficulty** $_{2.2}$**> … <Tid** $_{2, mn}$**, difficulty** $_{2, mn}$**>**

…

<Player $_n$**, evidence** $_{n>}$ **= < time** $_{n, 1}$**, difficulty** $_{n, 1}$**>, < time** $_{n, 2}$**, difficulty** $_{n, 2}$**> … <Tid** $_{mn, mn}$**, difficulty** $_{mn, mn}$**>**

DONE

where for each i, 1 ≤ i ≤ n, mi is the number of nodes in the i-th player's to-do list
and evidence $_i$ is, and for every j∈ {1, ..., n}, <pid {1, ..., mi}, tid $_{i, j}$ and difficulty $_{i, j}$ is the identifier of each task and its degree of difficulty respectively, corresponding to the j-th node of the list of
i-bone player tasks.

G

Give Away Work event, which indicates the transfer of tasks from one of the players with the most tasks to one of the players with the fewest tasks (or one of those who has completed all of their tasks and is not an alien). In this case you will go through the list of players, you will find the player <executable> is the name of the executable file of the program (eg a.out) pmin> with the fewest tasks in its to-do list and the player <executable > is the name of the executable file of the program (eg a.out) pmax> with the most tasks in its task list (following an algorithm corresponding to the one discussed in event E). Once this is done, you will transfer half the tasks, the ones with the least degree of difficulty, belonging to player <executable> is the name of the program executable file (eg a.out) pmax>, to player <executable> is the name of the program executable file (eg a.out) pmin >. Your algorithm should run in the most efficient way you can think of. Upon completion of such an event, the program should print the following information:

G

Player $_1$ **= < time** $_{1.1,}$ **difficulty** $_{1.1>,}$ **< time** $_{1,2,}$ **difficulty** $_{1,2>}$ **… <Tid** $_{1, mn,}$ **difficulty** $_{1, mn>}$

Player $_2$ **= < time** $_{2.1,}$ **difficulty** $_{2.1>,}$ **< time** $_{2.2,}$ **difficulty** $_{2.2>}$ **… <Tid** $_{2, mn,}$ **difficulty** $_{2, mn>}$

…

Player $_n$ **= < time** $_{n, 1,}$ **difficulty** $_{n, 1>,}$ **< time** $_{n, 2,}$ **difficulty** $_{n, 2>}$ **… <Tid** $_{mn, mn,}$ **difficulty** $_{mn, mn>}$

DONE

where for each i, 1≤i≤ n, mi is the number of nodes in the i-th player's to-do list, and
for each j∈ {1, ..., n}, <pid {1, ..., mi}, tid $_{i, j}$ and difficulty $_{i, j}$ is the ID of each task and its degree of difficulty respectively, which corresponds to the j-th node of the i-bone task list
player.

F

A fact that indicates the end of the game. In this case the following actions should be performed. First, you will check the list of players and if the aliens are more than humans then the game ends in favor of the aliens. If there are no aliens or the task stack contains as many tasks as the total number of tasks assigned to players, then the game ends in favor of humans. After the execution of such an event, in case the aliens win, the program should print the following information:

F

Aliens win.

DONE

In case people win, the program should print the following information:

F

Crewmates win.

DONE

X

Print Players type event that indicates the printing of all players. During this event, all players are printed with their IDs and the variable that determines whether they are aliens or not. At the end of such an event, the program should print the following information:

X

Players = <pid $_1$: is_alien $_1$> < pid $_2$: is_alien $_2$> … <Pid $_n$: is_alien $_n$>

DONE

where n is the number of nodes in the player list and each i ∈ {1, ..., n}, <pid {1, ..., n}, <executable> is the name of the executable file of the program (e.g. x. a.out) pid $_i$: is_alien $_i$> is the player ID and whether it is an alien respectively, corresponding to its i-th node
her list.

Y

Print Tasks event that indicates the printing of all jobs from the general task list. During this event, all tasks are printed with their IDs and their degree of difficulty. At the end of such an event, the program should print the following information:

Y

List_Tasks = <tid $_1$, difficulty $_1$> < time $_2$, difficulty $_2$> … <Tid $_n$, difficulty $_n$>

DONE

where n is the number of nodes in the general task list and for each i ∈ {1, ..., n}, <pid {1, ..., n}, <executable> is the name of the executable file of the program (e.g. a.out) tid $_i$> is the job ID that corresponds to the i-th node of this list.

Z

Print Stack event that indicates the printing of all jobs in the job stack. During this event, all tasks are printed with their IDs and the degree of difficulty that are in the stack. After the execution of this event, the stack should remain unchanged (with all the data it contained before the printing of its data). At the end of such an event, the program should print the following information:

Z

Stack_Tasks = <tid $_1$, difficulty $_1$> < time $_2$, difficulty $_2$> … <Tid $_n$, difficulty $_n$>

DONE

where n is the number of nodes in the task list and for each i i {1, ..., n n, <pid {1, ..., n n, <executable> is the name of the executable file of the program (π .x. a.out) tid i> is the job ID that corresponds to the i-th node of this list.

U

Print Tasks event that indicates the printing of all players and their to-do list. At the end of such an event, the program should print the following information:

U

Player 1 = < **time** 1.1, **difficulty** 1.1>, < **time** 1,2, **difficulty** 1,2> … <**Tid** 1, mn, **difficulty** 1, mn>

Player 2 = < **time** 2.1, **difficulty** 2.1>, < **time** 2.2, **difficulty** 2.2> … <**Tid** 2, mn, **difficulty** 2, mn>

…

Player n = < **time** n, 1, **difficulty** n, 1>, < **time** n, 2, **difficulty** n, 2> … <**Tid** mn, mn, **difficulty** mn, mn>

DONE

where for each i, 1≤i≤ n, mi is the number of nodes in the i-th player's to-do list, and
for each j∈ {1, ..., n}, <pid {1, ..., mi}, tid i, j and difficulty i, j is the ID of each task and its degree of difficulty respectively, which corresponds to the j-th node of the i-bone task list
player.

Small Bonus [5%]

Free up memory (data from all structures you have created) before ending your program.

Data structures

You are not allowed to use ready-made data structures (eg, ArrayList in Java, etc.) in your implementation. The following are the structures in C that should be used to perform the task.

```
struct Players {
   int pid;
   int is_alien;
   int evidence;
   struct Players * prev;
   struct Players * next;
   struct Tasks * tasks_head;
   struct Tasks * tasks_sentinel;
};

struct Tasks {
   int tid;
   int difficulty;
   struct Tasks * next;
};

struct Head_GL {
      int tasks_count [3];
      struct Tasks * head;
};

struct Head_Completed_Task_Stack {
   int count;
   struct Tasks * head;
};
```

Grading

P	7

T	8
D	15
I	10
E	15
W	5
S	13
V	7
G	10
F	2
X	2
Y	2
Z	2
U	2