HY240: Data Structures

Winter Semester - Academic Year 2020-2021 Instructor:

Panagiota Fatourou

Programming Work - Part 2

Delivery date: Monday, December 21, 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)

It is worth noting that for the educational purposes of the course some points of the work may deviate from the actual game.

Detailed Description of Required Implementation

Each player, alien (impostor) or human (crewmate), is described by a unique int (pid) type identifier. Information about the players is stored in one **a double-linked binary search tree with a guard node, called a player tree (Figure 1).**

Each node in the tree is a Player type struct 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. Note that, although each player's type is recorded in the tree, we consider that the other players do not have access to each player's is_allien field and therefore cannot find out from there if the player is an alien or a human. The player tree is sorted (according to the intersection) according to the pid field of the players it contains.

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

- **pid:** Identifier (int type) that uniquely characterizes the player.
- **is_alien:** Variable (int type) that indicates 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).
- **parent:** Index (struct Player type) that indicates **parent node** of the node with pid ID.
- **lc:** Index (struct Player type) that indicates **left affiliate node** of the node with pid ID.
- **rc:** Index (struct Player type) that indicates **right affiliate node** of the node with pid ID.
- **tasks:** Index (type *struct Task)* at the root of a simple-connected binary search tree (without guard node), which contains information about the tasks to be assigned to the player.

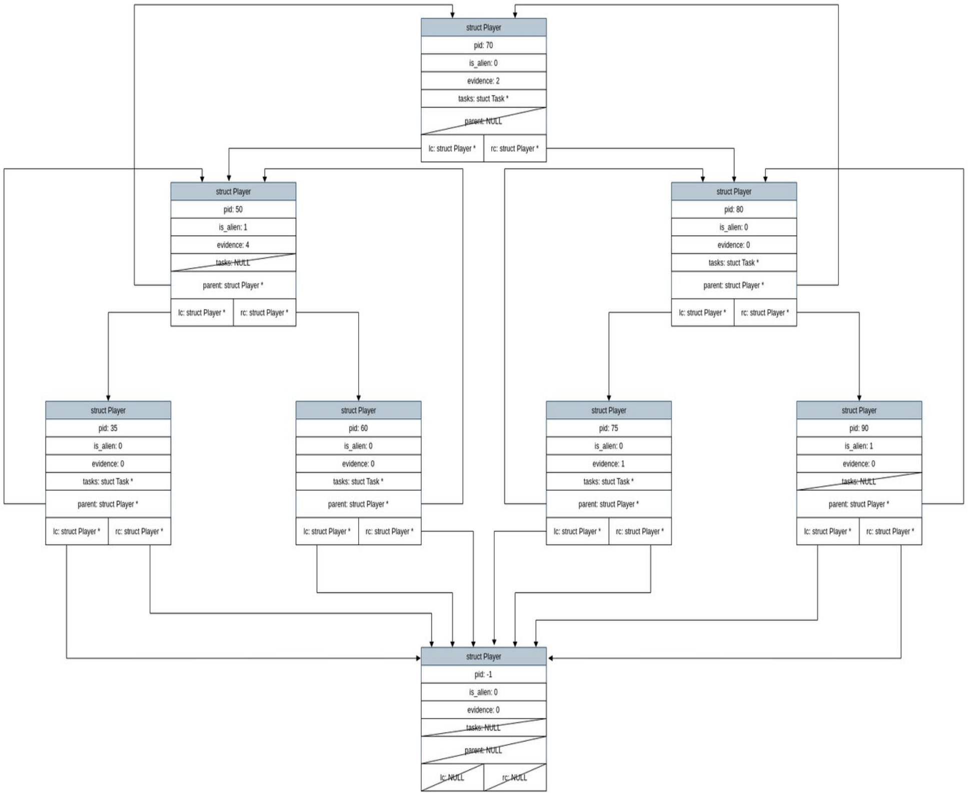


Figure 1. The player tree that is double connected, with a guard node and sorted. The second field indicates whether a player is an alien or not, and the third field indicates whether a player is a suspect.

Each player has different tasks to complete. So, for every player there is one **a binary search tree, called the player task tree.** Each task assigned to the player is characterized by a unique identifier of type int (tid) and has a degree of difficulty (indicated in the field difficulty, also of type int). Additionally, each node in a player task tree has one more lcnt field that stores an integer. The tree provides the following property that we call **cash status of the tree:** *"The integer lcnt of each node v of the tree stores the number of nodes contained in the left subtree of v."*

Each player's task tree is sorted according to the task tid it contains.

Each node in a specific player taskbar is a struct Task entry with the following fields:

- **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).
- **lcnt:** The number of nodes in the left subtree (int type) of the node with a tid identifier.
- **lc:** Index (struct Task type) that indicates **left affiliate node** of the node with tid ID.
- **rc:** Index (struct Task type) that indicates **right affiliate node** of the node with tid ID.

All the work that needs to be done to defeat the spaceship crew (humans) is first introduced in a **hash table, called the general task hash table (Figure 2). Conflict resolution is based on the method of individual chains. The hash function is selected based on the universal hash technique.** To implement global hash, a table of primes_g [650] with prime numbers in ascending order, the maximum number of tasks through the variable max_tasks_g and the maximum ID between tasks through the variable max_tid_g is provided. Each node of any general task table hash chain is a struct HT_Task entry containing the following fields:

- **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:** Pointer (struct type HT_Task) to the next node of the chain. There is a struct, type

General_Tasks_HT that contains the following fields:

- **count:** which stores the total number of jobs stored in the taskbar.
- **tasks:** Pointer to HT_Task struct that implements the general task table (which is dynamically bound).

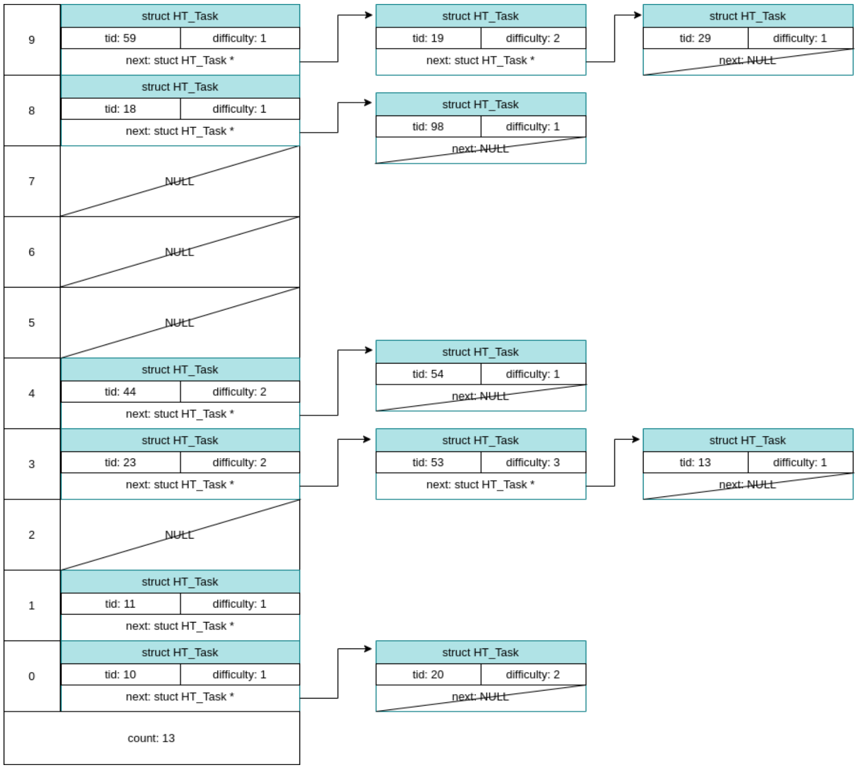


Figure 2. The general task hash table. Jobs are entered in the corresponding chain based on their ID. In the example, the hash function is h (tid) = tid mod 10.

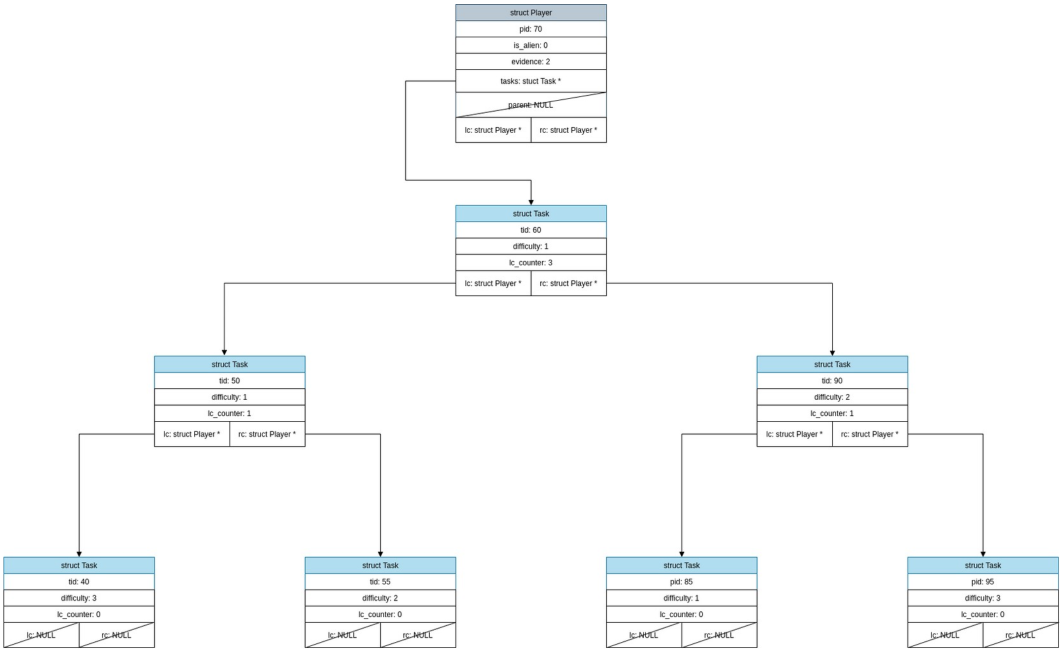
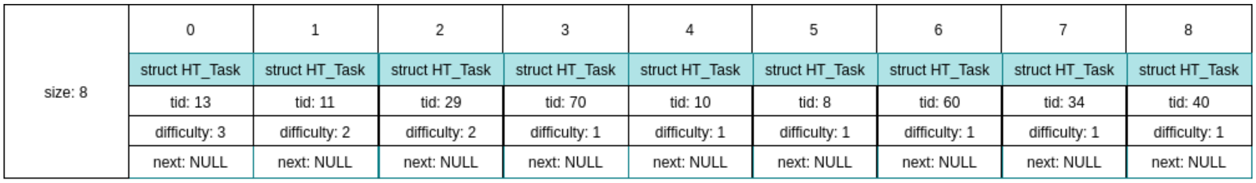


Figure 3. A player sorted work tree.

Once all tasks are entered in the general task fragmentation table, they will eventually be shared in the crew members' task trees (**Figure 3).** Then they start implementing them. Each time a person completes a task, it is removed from the task tree and inserted into a **Maximum priority queue, called completed task queue (Figure 4).** Each element of the completed task queue is a HT_Task struct type entry (ie it is of the same type as the nodes in the chains of the players general task fragmentation table, except that the next field is not used). The priority of each queue element is defined by the value of the difficulty field (higher values indicate higher priority).

The completed task queue is implemented with a Completed_Tasks_PQ struct that contains the following fields:

- s **ize:** Integer that stores the number of priority queue items at any given time.
- **tasks []:** A table (with struct HT_Task elements) that will be dynamically bound and will contain the priority queue elements,



size: 8	0	1	2	3	4	5	6	7	8
	struct HT_Task	struct HT_Task	struct HT_Task	struct HT_Task	struct HT_Task	struct HT_Task	struct HT_Task	struct HT_Task	struct HT_Task
	tid: 13	tid: 11	tid: 29	tid: 70	tid: 10	tid: 8	tid: 60	tid: 34	tid: 40
	difficulty: 3	difficulty: 2	difficulty: 2	difficulty: 1	difficulty: 1	difficulty: 1	difficulty: 1	difficulty: 1	difficulty: 1
	next: NULL	next: NULL	next: NULL	next: NULL	next: NULL	next: NULL	next: NULL	next: NULL	next: NULL

Figure 4. The highest priority queue with completed tasks. If the number of items in it is equal to the number of items in the general task fragmentation table then people have won the lot.

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

P <pid> <is_alien>

Insert Player type event that indicates the introduction of a player with <pid> ID in the game. In this case an item (corresponding to the new player) will be inserted into the player tree. The evidence and is_alien fields of this item must be initialized with the values 0 and <is_allien>, respectively (if <is_allien> is 1 then the player is an alien, otherwise he is a human, crew member). 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 tree and for each i { 1, ..., n}, <pid$_{i>}$ is the player ID that corresponds to the i-th node of the tree according to
intersecting crossing.

T <tid> <difficulty>

Insert Task event that indicates the creation of a task with an <tid> ID and a degree of difficulty <difficulty>. In this case, a new item corresponding to this task will be entered in the general task hash table. Specifically, the <tid> ID should be given as input to the hash function and then the new node will be inserted into the corresponding table array. At the end of such an event, the program should print the following information:

T <tid> <difficulty>

Chain 0: <tid$_{0.1,}$ **difficulty**$_{0.1>, <}$ **pid**$_{0.2,}$ **difficulty**$_{0.2>, ..., <}$ **aid**$_{0, N0,}$ **difficulty**$_{0, N0>}$

Chain 1: <tid$_{1.1,}$ **difficulty**$_{1.1>, <}$ **aid**$_{1,2,}$ **difficulty**$_{1,2>, ..., <}$ **aid**$_{1, N1,}$ **difficulty**$_{0, N1>}$

. . .

Chain n: <tid$_{n, 1,}$ **difficulty**$_{n, 1>, <}$ **aid**$_{n, 2,}$ **difficulty**$_{n, 2>, ..., <}$ **aid**$_{n, Nn,}$ **difficulty**$_{n, Nn>}$

DONE

where n is the number of positions in the general task fragmentation table and for each i, 0 ≤ i ≤ n-1, N$_i$ is the number of nodes in the chain of position i of the hash table and
for each j, 1 ≤ j ≤ N$_{i,}$ time$_{j, i}$ and difficulty$_{i, j}$ is the job ID corresponding to the j-th node of the chain located in position i of the hash table and the difficulty of executing it, respectively.

D

Distribute Tasks event that indicates the assignment of tasks to players. In this case, the tasks in the general task fragmentation table should be distributed to the players following a round robin algorithm. Specifically, the fragmentation table is crossed from the first to the last chain and the i-th job to be crossed is assigned to the i-th player (MOD n) according to the intersection, where n is the number of elements in the player tree. Crossing the hash table requires the use of cash.

To accomplish this, a retro crossing of the tree must be performed and each non-alien node must be assigned a task. The above procedure is performed repeatedly until all tasks are assigned to the players. Therefore, a series of successive tree crossings must be performed and each such crossing must be assigned a task by each of the crew players. The temporal complexity of this event should be O (n), where n is the total number of tasks in the general task table.

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 n is the number of players in the player tree and for each i, 1≤i≤ n, m_i is the number of nodes in the i-th player task tree in the player tree (according to intra-arranged crossing), and for each j { 1, ..., $m_{i\}}$, time $_{i,j}$ and difficulty $_{i,j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree (according to the intersection).

I <pid> <tid>

An Implement Task event, which indicates that the player with the <pid> ID is performing a task with the <tid> ID. In this case the following actions should be performed. Initially, you will be looking for that player in the player tree. Once you have found that player, you will be looking for job t with an <tid> ID in the player job tree. Then if you find t, you will delete it from the player's task tree (so that the tree remains sorted and the left node counters are correctly updated after the event). Finally, a node for t will be inserted in the completed task queue. At the end of such an event, the program should print the following information:

```
I <pid> <tid>
   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 n is the number of players in the player tree and for each i, 1≤i≤ n, m $_{i}$ is the number of nodes in the i-th player task tree in the player tree (according to
intra-arranged crossing), and for each j { 1, ..., m $_{i\},}$ time $_{i, j}$ and difficulty $_{i, j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree (according to the intersection).

E <pid $_{1> <}$ pid $_{2>}$

Eject Player type event that indicates the player's touch with <pid ID $_{1>}$ from an alien and its removal from the spacecraft (without anyone else noticing
player). In this case, the player with ID <pid $_{1>}$ is removed from the player tree and the player task tree merges with that of the player with ID $_{2>.}$ The player taskbar with <pid ID $_{2>}$ resulting from the merger must be sorted, have an O (logn) height and the left node counters have the correct values after the merger. The merge process must be performed in time O (n1 + n2), where n1 and n2 are the number of nodes in the player's task trees <pid1> and <pid2>, respectively. At the end of such an event, the program should print the following information:

E <pid $_{1> <}$ pid $_{2>}$

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 n is the number of players in the player tree and for each i, 1≤i≤ n, m $_{i}$ is the number of nodes in the i-th player task tree in the player tree (according to
intra-arranged crossing), and for each j { 1, ..., m $_{i\},}$ time $_{i, j}$ and difficulty $_{i, j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree
(according to the intersectional crossing).

W <pid $_{1> <}$ pid $_{2> <}$ pid $_{a> <}$ number_witnesses>

Witness Ejection event that indicates the player's touch with <pid ID $_{1>}$ from the alien with ID <pid $_{a>}$ and player removal <pid $_{1>}$ from the spaceship. However, in this case, the removal of the player <pid $_{1>}$ it is seen by <number_witnesses> people (crewmates). This event is performed in exactly the same way as event E: the player with <pid ID $_{1>}$ is removed from the player tree and its task tree is merged with that of the player with an <pid $_{2>.}$ However, the following should be added: find the alien with the <pid ID $_{a>}$ and update the evidence field for it by adding the number <number_witnesses>. At the end of such an event, the program should print the following information:

```
W <pid 1> < pid 2> < 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 n is the number of players in the player tree and for each i, 1≤i≤ n, m_i is the number of nodes in the i-th player task tree in the player tree (according to intra-arranged crossing), and for each j { 1, ..., $m_{i\}}$, $time_{i,j}$ and $difficulty_{i,j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree (according to the intersectional crossing).

S <number_of_tasks> <pid>

Sabotage event, which suggests that an alien removes <number_of_tasks> tasks from the priority queue of completed tasks and redistributes them to players. In this case, <number_of_tasks> delete those tasks with the highest degree of difficulty (ie with the highest priority) from the queue of completed tasks. Each task removed from the priority queue is assigned to a player using the following algorithm:

- First, we find the player with the <pid> ID in the player tree.
- The first task is assigned to player p preceding player <pid> against ⌊ number_of_tasks / 2 ⌋ positions in the intersection (or, if p is extraterrestrial, at the first node corresponding to a human from the previous p). To do this, you need to implement the FindInorderPredecessor () function, which finds the antecedent of a node in the intersection. This process should be repeated repeatedly until you reach that player.
- Each subsequent task is assigned to each of the following nodes of p according to the intra-arranged intersection, which are not extraterrestrial. To do this, you need to implement the function, FindInorderSuccessor (), which finds the next one node in the intersection.
- In this way, each of them will be assigned a task ⌊ number_of_tasks / 2 ⌋ previous nodes of p corresponding to crew members (according to the intersection) and the ⌊ number_of_tasks / 2 ⌋ subsequent nodes of p (according to the intersection).
- Assume that the previous node in the intersection is the last node (in the intersection). Also, consider that the next node (in the intersection) is the first node (in the intersection).

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 n is the number of players in the player tree and for each i, 1≤i≤ n, m_i is the number of nodes in the i-th player task tree in the player tree (according to
intra-arranged crossing), and for each j { 1, ..., m_i}, time$_{i, j}$ and difficulty$_{i, j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree
(according to the intersectional crossing).

V <pid$_{1}$> < **pid**$_{2}$> < **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 ID <pid$_{1}$> is considered a suspect (because we consider that he is the one who makes the most noise during the voting) and so the evidence field of the struct that corresponds to the player tree increases by <vote_evidence>. Next, you need to go through the player tree, find the player with the most evidence, and remove it. The player tree of work thrown out of the spaceship will be merged with that of the player with ID <pid$_{2}$>

(as discussed in events E and W). At the end of such an event, the program should print the following information:

V <pid$_{1}$> < **pid**$_{2}$> < **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 n is the number of players in the player tree and for each i, 1≤i≤ n, Player$_{i}$ and evidence$_{i}$
is the i-th player and the corresponding evidence (respectively) in the player tree (according to
the intra-arranged crossing), m_i is the number of nodes in the i-th player task tree in the player tree, and for each j { 1, ..., m_i}, time$_{i, j}$ and difficulty$_{i, j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree (according to the intersection).

G <pid$_{1>}$ < pid$_{2>}$

Give Away Work event, which indicates the transfer of tasks by the player with ID <pid$_{1>}$ to a new player with <pid ID$_{2>.}$ This will enter the player with a <pid ID$_{2>}$ on the player tree. You will then look for the player with the <pid ID$_{1>}$ in the player tree and you will transfer half of the tasks belonging to

player <pid$_{1>,}$ to the player with ID <pid$_{2>.}$ To accomplish this, you need to implement an algorithm that measures the total number of nodes in the player's T tree <pid1>. Next, you need to split the T into two other trees, each containing half as much data as the T. Your algorithm should run in

time O (h), where h is the height of the player's task tree <pid$_{1>.}$ At the end of such an event, the program should print the following information:

```
G < pid 1> < pid 2>
    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 n is the number of players in the player tree and for each i, 1≤i≤ n, m_i is the number of nodes in the i-th player task tree in the player tree (according to
intra-arranged crossing), and for each j { 1, ..., m$_{i\},}$ time$_{i,\,j}$ and difficulty$_{i,\,j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree
(according to the intersectional crossing).

F

A fact that indicates the end of the game. In this case the following actions should be performed. Initially, you will control the player tree and if the aliens are more than humans then the game ends in favor of the aliens. If there are no aliens or the priority queue of completed tasks contains as many tasks as the total number of tasks assigned to players, then the game ends in favor of the people. 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 tree and for each i { 1, ..., n}, <pid$_{i:}$ is_alien$_{i>}$ is the player ID and whether it is alien respectively, corresponding to its i-th node
tree according to the intersectional crossing.

Y

Print All Tasks event that indicates the printing of all tasks from the general task hash table. 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

Index 0: <pid$_{0.1,}$ **difficulty**$_{0.1>,}$ < **pid**$_{0.2,}$ **difficulty**$_{0.2>, ...,}$ < **aid**$_{0, N0,}$ **difficulty**$_{0, N0>}$

Index 1: <pid$_{1.1,}$ **difficulty**$_{1.1>,}$ < **aid**$_{1,2,}$ **difficulty**$_{1,2>, ...,}$ < **aid**$_{1, N1,}$ **difficulty**$_{0, N1>}$

. . .

Index n: <pid$_{n, 1,}$ **difficulty**$_{n, 1>,}$ < **aid**$_{n, 2,}$ **difficulty**$_{n, 2>, ...,}$ < **aid**$_{n, Nn,}$ **difficulty**$_{n, Nn>}$

DONE

where n is the number of positions in the general task hash and for every j, 0 ≤ j ≤ n-1, N j is the number of nodes in the chain of position j of the hash, as for every i, 1 ≤ i ≤ N j, <tid j, i> is the job identifier corresponding to the i-th node of the chain located at position j of the hash table.

Z

Print Completed Tasks event that indicates the printing of all jobs that are in the priority queue of most completed jobs. During this event, all tasks are printed with their IDs and the degree of difficulty that exist in the priority queue. At the end of such an event, the program should print the following information:

Z

Completed Tasks = <tid$_{1,}$ **difficulty**$_{1>}$ < **time**$_{2,}$ **difficulty**$_{2>}$ **… <Tid**$_{n,}$ **difficulty**$_{n>}$

DONE

where n is the number of items in the priority queue of completed tasks and for each i { 1, ..., n}, <tid$_{i>}$ is the job ID that corresponds to the i-th element of the array.

U

Print Tasks event that indicates the printing of all players and their task trees. 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 n is the number of players in the player tree and for each i, 1≤i≤ n, m$_{i}$ is the number of nodes in the i-th player task tree in the player tree (according to
intra-arranged crossing), and for each j { 1, ..., m$_{i\}}$, time$_{i, j}$ and difficulty$_{i, j}$ is the identifier of the j-th job and its degree of difficulty respectively, in the i-th player task tree
(according to the intersectional crossing).

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 Player {
    int pid;
    int is_alien;
    int evidence;
    struct Player * parrent;
    struct Player * lc;
    struct Player * rc;
    struct Tasks * tasks;
};

struct Task {
    int tid;
    int difficulty;
    int lcnt;
    struct Task * lc;
    struct Tasks * rc;
};

struct HT_Task {
    int tid;
    int difficulty;
    struct Task * next;
};

int primes_g [650];
unsigned int max_tasks_g;
unsigned int max_tid_g;

struct General_Tasks_HT {
    int count;
    struct HT_Task ** tasks;
}
```

```
struct General_Tasks_HT general_tasks_ht;


struct Completed_Tasks_PQ {
    int size;
    struct HT_Task tasks [];
};


struct Completed_Tasks_PQ completed_tasks_pq;
```

Grading

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