University of Crete, Department of Computer Science November 19, 2017

HY240: Data Structures

Winter Semester – Academic Year 2017-2018

Teacher: Panagiota Faturou

Programming Work - Part 2

Delivery Date: Friday, December 22, 2017, 11:59 p.m.

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

General description

In the second part of this assignment, you are again asked to implement a program that will simulate the last days of the Trojan war, between the Achaeans and the Trojans, as it is described in the Iliad, using however different data structures from those of the first phase.

Detailed Description of Requested Implementation

The list of belligerents (Rhapsody B, 22nd day). "During the preparations, the poet invokes the help of the muses to list the long list of belligerents" (Homeric Epics: Iliad II Gymnasium). One of the tasks of this work is the implementation of the census. Information about the warriors of the Achaean camp is stored in a *hash table.* This table is called **the registration** *hash table.* To resolve the conflicts you should use the technique of *separate chains (separate chaining)* and as a hash function, *h(k)=k mod N,* where k is the key and N is the size of the hash table. To implement the hash table, the global variable *max_soldiers_g is provided,* which gives the maximum number of soldiers that will participate in the war. Each node of a chain corresponds to a soldier and constitutes a record of type soldier with the following fields:

- **sid:** Identifier (of type int) that uniquely characterizes the warring man.
- **gid:** Identifier (of type int) that uniquely identifies the king/general to whom the man obeys.
- **next:** Pointer (soldier type) to the next node of the chain to which it belongs.

Figure 1 shows the soldiers hash table.

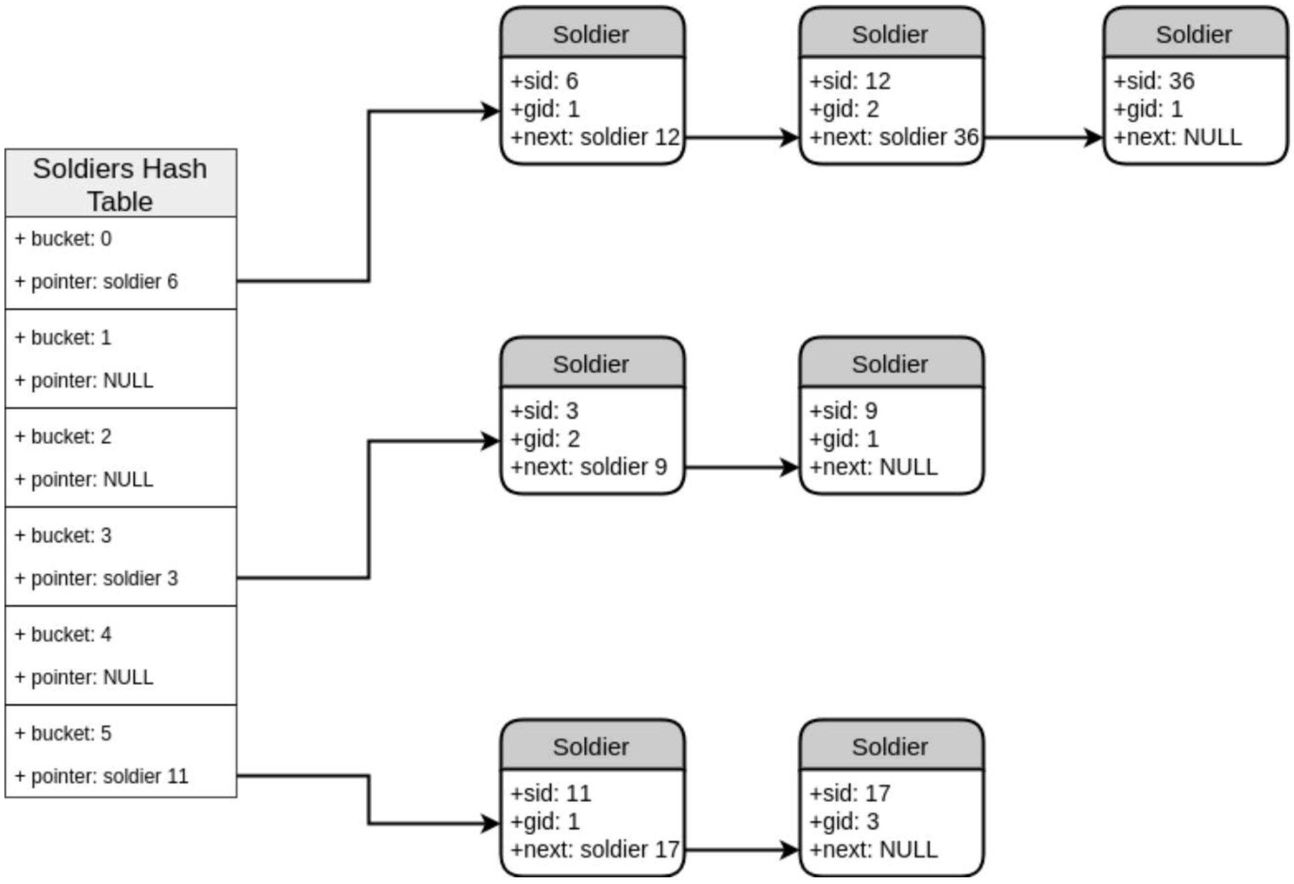

Figure 1: The Soldier hash table and the lists (chains) to solve conflicts.

The Achaean troops are organized into ranks led by brave kings and generals. Each king/general has his own army (set of soldiers) encamped in front of the ships that carried the men to Troy. Information about kings/generals is stored in a simply linked, unordered list. This list is called the generals list. Each node in this list is a general record with the following fields:

- **gid:** Identifier (of type int) that uniquely characterizes the general.
- **combats_no:** Number (of type int) representing the number of combats he has join the general.
- **soldiers_S:** Pointer of type TREE_soldier pointing to the sentry node of its soldier tree said general.
- **soldiers_R:** Pointer of type TREE_soldier pointing to the root of the tree of men obeying the general. This tree is *binary, has pointers to parent nodes, is ordered by the ID of the soldier,* and is called **a soldier tree.** Each element of this tree is a record of type TREE_soldier with the following fields:

- o **sid:** Identifier (of type int) that uniquely characterizes the warring man.
- o **rc:** Pointer (of type TREE_soldier) that points to the right child of the node.
- o **lc:** Pointer (of type TREE_soldier) that points to the left child of the node.
- o **p:** Pointer (of type TREE_soldier) that points to the parent of the node.

• **next:** Pointer (of type general) that points to the next node in the list of generals.

Figure 2 shows the list of generals and the tree of soldiers indexed by each element of the list of generals.

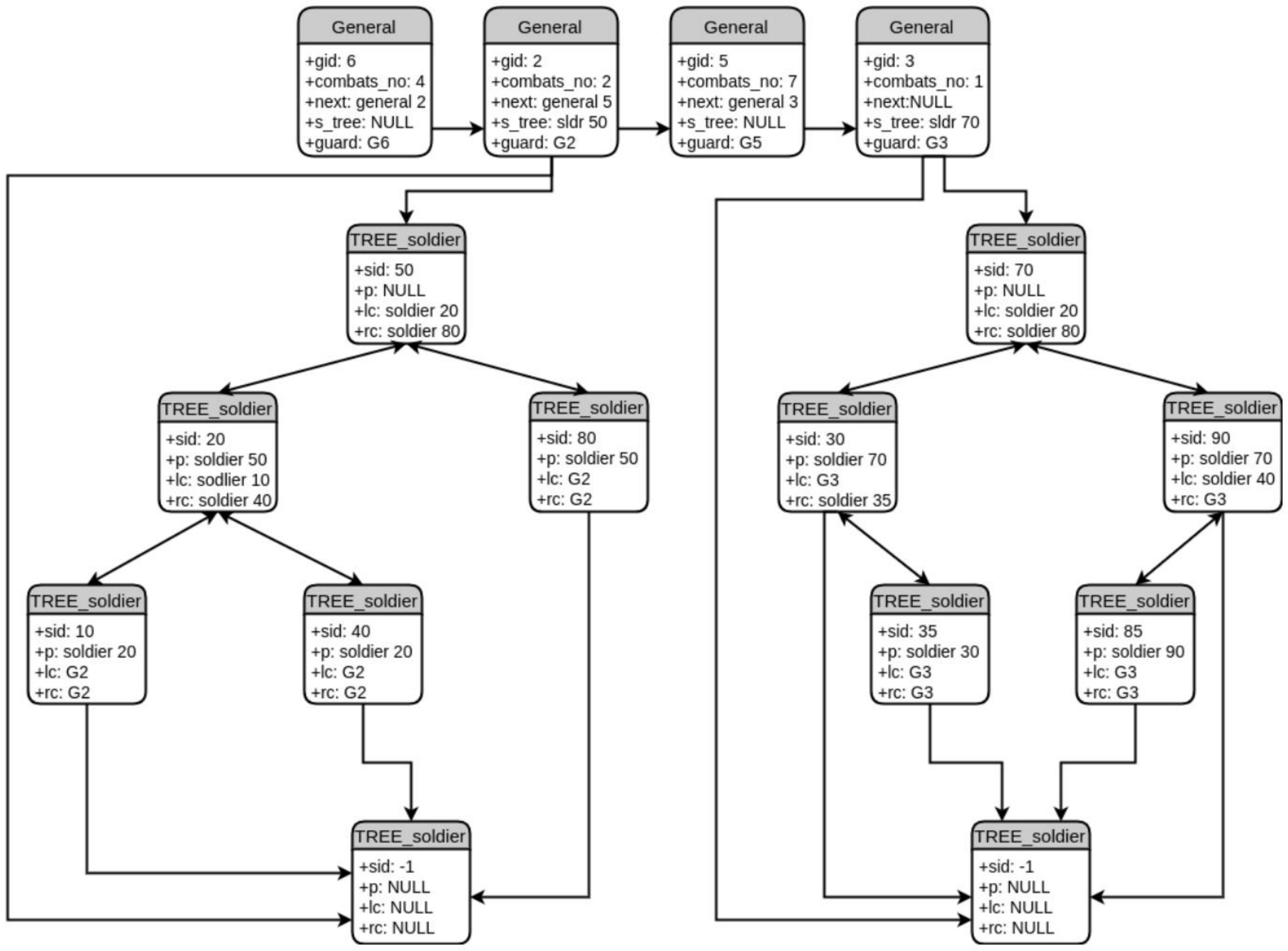

Figure 2: The unordered, simply linked list of generals. Each node (general) corresponds to a king/general and contains a pointer to the root of the soldier tree. Each general's soldier tree is ordered by soldier ID and has a guard node.

Battles in the Rhapsodies D and E (22nd day), T (25th day), L, M and N (26th day, the battles around the Achaean wall), P and R (27th day, Patroclus leads the Myrmidons into battle), F (Achilles returns to battle).

One more of the tasks of this job is to implement the battles. Each battle involves some of the generals and their soldiers. Combat soldiers are stored in a combat type structure containing the following fields:

• **soldiers_cnt:** Integer representing the total number of soldiers participating in battle

• **combat_s:** Pointer (of type c_soldier) pointing to the root of a *binary, sorted by soldier ID,* tree, called **combat tree.** Each element of the tree is a record of type c_soldier with the following fields:

  o **sid:** Identifier (of type int) that uniquely characterizes the fighting man.

  o **alive:** Binary variable representing the state of the fighting man. The value 1 indicates that the man is alive, while a value of 0 indicates the opposite.

  o **gid:** Identifier (of type int) that uniquely identifies the general to whom the fighting man obeys.

  o **rc:** Pointer (of type c_soldier) that points to the right child of the node.

  o **lc:** Pointer (of type c_soldier) that points to the left child of the node.

  o **left_no:** The number of nodes in the left subtree of the node.

Figure 3 shows the combat structure containing the combat tree and the counter of soldiers participating in the combat.

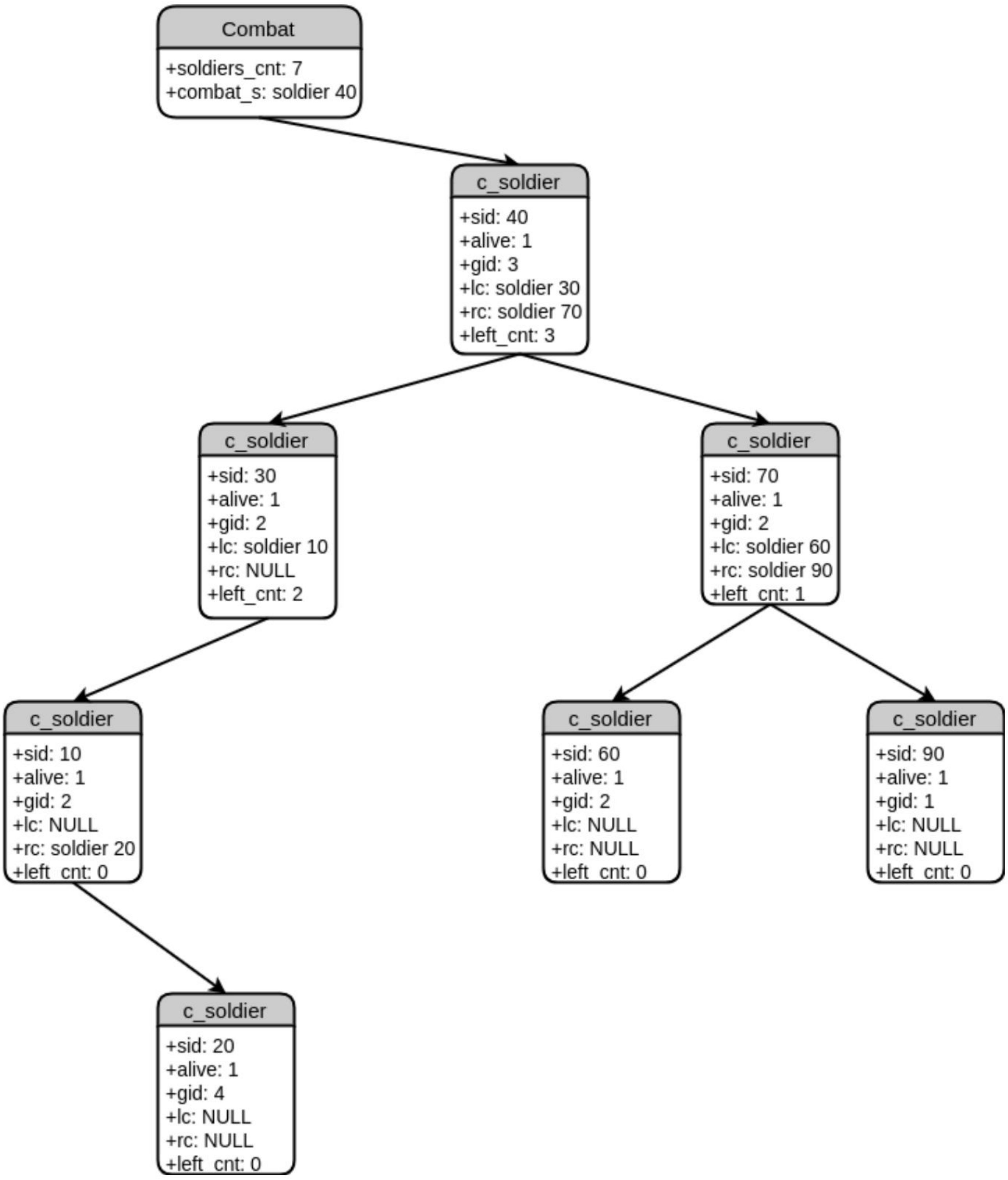

Figure 3: The combat structure containing the combat tree and the counter of soldiers participating in the battle.

How the Program Works

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

<executable> <input-file>

where <executable> is the name of the program's executable file (eg a.out) and <input-file> is the name of an input file (eg testfile) which contains events of the following formats:

- R <sid> <gid>

Event in which a soldier is listed **(Rhapsody B[48-877], 22nd day).** Upon doing so, a new soldier is inserted into the inventory hash table. The soldier will have an identifier <sid> and will act under the orders of the general with an identifier <gid>. To enter the hash table, you should use the hash function described above, as well as the separate chains technique to resolve conflicts. After the execution of such an event the program should print the following information:

```
R <sid> <gid>
        Soldier Hash Table:
        <sid1,1:gid1,1>, <sid1,2:gid1,2>, ..., <sid1,n1:gid1,n1>,
        <sid2,1:gid2,1>, <sid2,2:gid2,2>, ..., <sid2,n2:gid2,n2>,
        ...
        <sidk,1:gidk,1>, <sidk,2:gidk,2>, ..., <sidk,nk:gidk,nk>
DONE
```

where for each i, 1 ÿ i ÿ k, ni is the number of nodes of the i-th chain of the hash table and for each j, 1 ÿ j ÿ ni, sidi,j is the identifier of the j-th node of the i- chain link and gidi,j the ID of the general that this node (soldier) obeys.

- G <gid>

Event indicating that the king/general with id <gid> is participating in the campaign. During this event, a node of type general is inserted in the list of generals. Initially the general's soldier list will be empty (there is only the guard node). The input should be done with the least time complexity, given that the <gid> that follows each type G event in the test files is unique. After the execution of such an event the program should print the following information:

```
G <gid>
        Generals = <gid1>, <gid2>, ..., <gidn>
DONE
```

where n is the number of nodes in the general list and for each i ÿ {1, …, n}, <gidi> is the

ID of the general corresponding to the i-th node of this list.

- D

A *distribute soldiers* type event which signals a night's rest. Soldiers rest in the camps where they belong (depending on the general they are under). In this event, traversing the inventory hash table should create the trees of soldiers subordinate to each general. This can be done by examining the gid field of each hash table entry and then searching for the strategist by id in the strategist list. Then a struct of type TREE_soldier will be imported which will correspond to the soldier in the general's soldier tree. After each import, each general's soldier tree should be sorted by the soldiers ID. At the end of the event, for all soldiers in the soldier hash table, there must be corresponding entries in the appropriate generals soldier trees.

After finishing executing such an event, the program should print the following information:

```
D

        General:
        <gid1>: <sid1,1> .        ·   . <sid1,n1>
        <gid2>: <sid2,1> .        ·   . <sid2,n2>
        ...
        <gidk>: <sidk,1> .        ·   . <sidk,nk>

DONE
```

where for each i, 1 ÿ i ÿ k, ni is the number of soldier tree nodes of the i-th node of the list of generals, and for each j, 1 ÿ j ÿ ni, and <sidi,j> is the identifier of j-th node in the soldier tree of the i-th general (based on interordered traversal).

- M <gid1> <gid2>

An event that marks Achilles' withdrawal from the battle (after his quarrel with Agamemnon), while Patroclus convinces Achilles to lead the Myrmidons into battle.

Consider that Achilles has ID <gid1> and Patroclus has ID <gid2>. In doing so you must delete the node with ID <gid1> from the list of generals. In addition, Achilles' men will be moved to Patroclus' soldier tree (which has ID <gid2>)[1] . The union of the two trees must be achieved with a time complexity of O(n), where n is the sum of the men of Achilles and Patroclus. This will be done with two auxiliary tables, where the first will store Achilles' men and the second their

[1] *In the Iliad, Patroclus leads the Myrmidons into battle wearing Achilles' armor. To serve the educational objectives of the lesson, however, consider that Patroclus also had his own men who participated in the battle alongside the Myrmidons.*

men of Patroclus. Tables should be sorted by soldier ID. You should then create a third table containing the men of both generals, again sorted by their ID. The creation of the third table (which is sorted) should be achieved in O(n) time. Then, based on the third table, you should recreate Patroclus' men tree which will contain all the men of both generals. Attention this tree must have height O(logn). After the process is finished, Patroclus' soldier tree should be sorted by soldier ID.

After the execution of such an event the program should print the following information:

```
M <gid1> <gid2>

        General:
        <gid1>: <sid1,1> .        ·   . <sid1,n1>
        <gid2>: <sid2,1> .        ·   . <sid2,n2>
          ...
        <gidk>: <sidk,1> .        ·   . <sidk,nk>
DONE
```

where for each i, 1 ÿ i ÿ k, ni is the number of soldier tree nodes of the i-th node of the list of generals, and for each j, 1 ÿ j ÿ ni, and <sidi,j> is the identifier of j-th node in the soldier tree of the i-th general (based on interordered traversal).

- P <gid1> <gid2>

A *prepare for battle* type event which marks the selection of the men who will participate in the next battle. Generals with IDs <gid1> and <gid2> will take part in the battle with their men. You will also increment the combats_no counter, which indicates the number of combats each general has participated in.

Soldiers to fight should be stored in the combat tree (combat_s) of the combat record. Each element of this tree contains a record of type c_soldier. Each time a node is inserted into this tree, you must update the combat record's soldiers_cnt counter accordingly. The way in which the soldiers of each general will be inserted into the battle tree is as follows:

Starting with the soldier with the smallest ID in the general's soldier tree with ID <gid1> and the soldier with the largest ID in the general's soldier tree with ID <gid2>, nodes from the two trees will be placed alternately in the battle tree . In particular, you should cross the two soldier trees at the same time, placing in
each step in the battle tree is either the next node in the nested traversal of general <gid1>'s trooper tree, or the previous node in the nested traverse of general <gid2>'s troop tree. To implement this, you'll need to create the *inorder_successor() and inorder_predeccessor() functions,* which will take you to the next or

previous node in the inorder traversal in each tree respectively (ie moving from one node to another is done using either inorder_successor() or inorder_predessecor(), depending on the tree from which you visit the next male to be inserted into the battle tree). For each node you visit, you will insert a corresponding struct of type c_soldier into the battle tree. After each soldier import from one general's soldier tree, the next import will be from the other general's soldier tree. In other words, introductions will take place alternately.

For each soldier you insert into the combat tree of the combat record, you should set the value alive = 1 after the soldier is alive and refresh the value of the *left_cnt* counter on the appropriate nodes.

Figure 4 shows an example of the soldier trees of the generals with IDs 10 and 20, who will participate in the battle. Figure 5 shows the result of inserting the soldiers of the above generals into the battle tree, according to the algorithm described above.

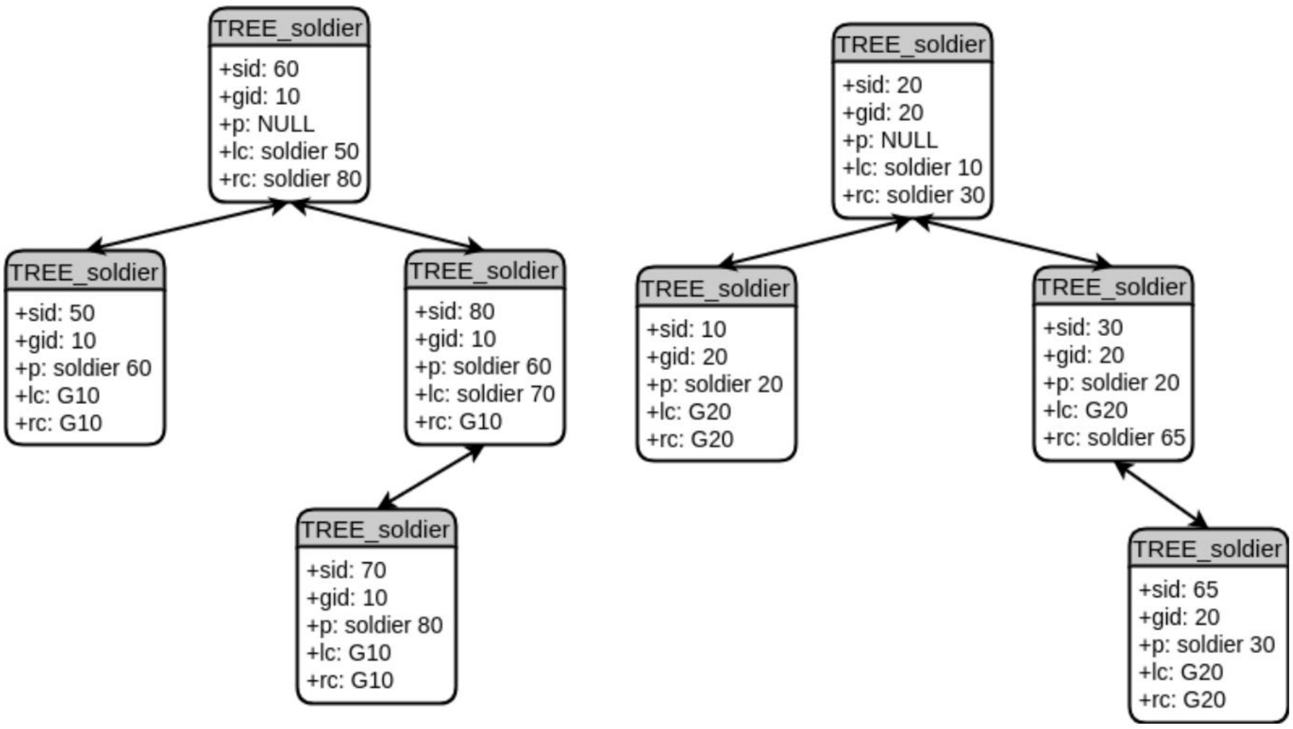

Figure 4 The soldier trees of generals with IDs 10 and 20.

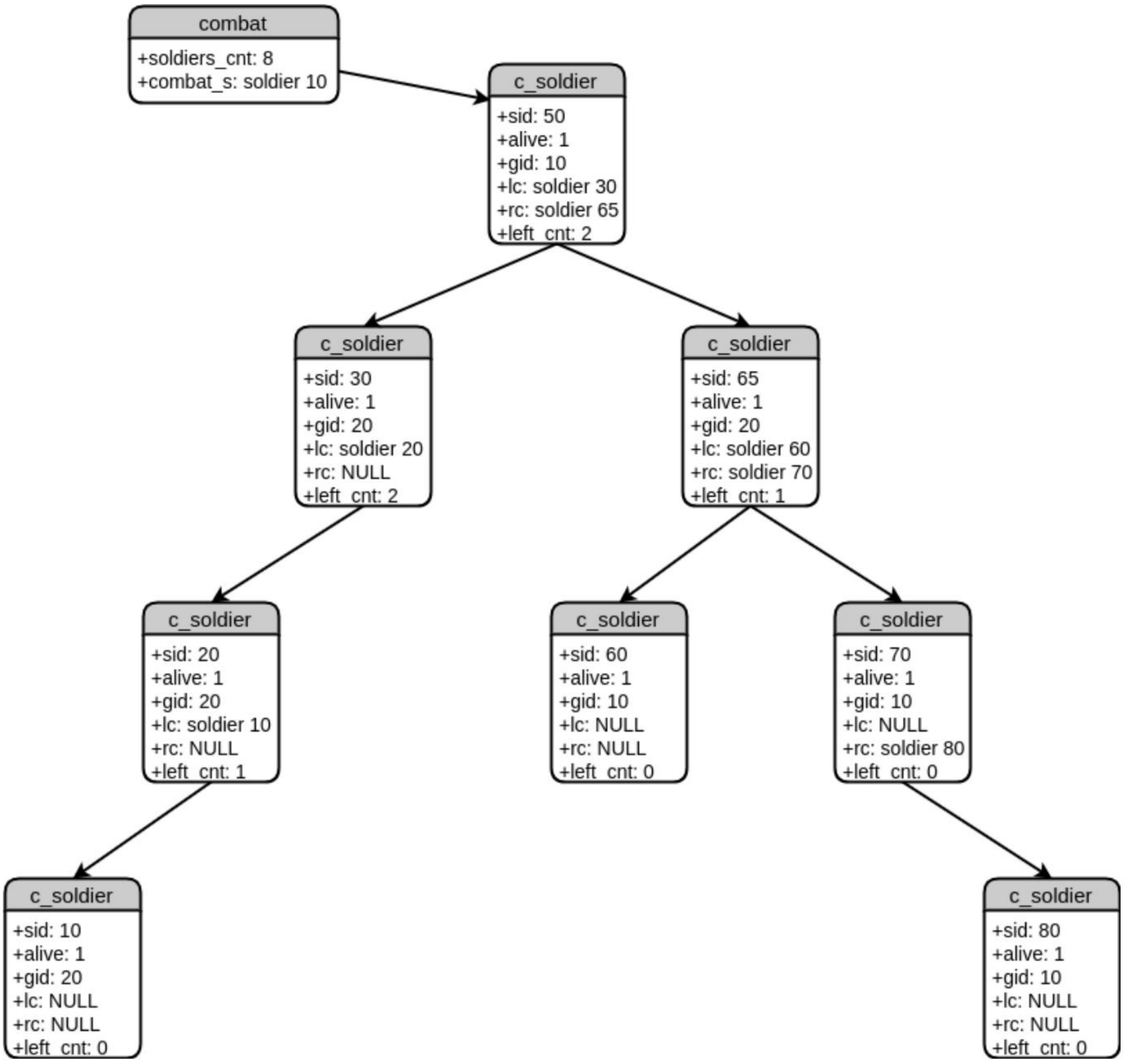

Figure 5 The battle tree as it results from the concatenation of the soldier trees shown in Figure 4. The entries in this tree were made in the following order: 50, 65, 60, 30, 70, 20, 80, 10.

Additionally, you should increment the appropriate combats_no counters, which indicate the number of combats in which each of the two generals has participated. After the execution of such an event the program should print the following information:

```
P <gid1> <gid2>

    Combat soldiers: <sid1>, <sid2> .            .   . <sidn>
DONE
```

where n is the number of battle tree nodes and for each i, 1 ÿ i ÿ n, <sidi> is the ID of the i-th soldier in the battle tree (basis of interordered traversal).

- B <god_favor> <bit_stream>

Event that marks the simulation of battle (see e.g. Rhapsodies D, E, L, M and N).

In this event (which is of the *battle type)* the simulation of the battle is signaled. During the battle some of the combatants die. The amount of losses is affected by the favor of the gods towards the Greeks as defined by the <god_favor> parameter.

In the event that the Gods do not favor the troops of the Greeks (e.g. in the battles of the 25th and 26th day including the battles around the walls of Achaia, Rhapsodies L, M and N), then the losses amount to 40% of the soldiers participating in the battle. Using the combat record's soldiers_cnt counter, you will calculate the number of soldiers lost in combat. You will then traverse the combat tree (combat_s) of the combat record and set 40% of the soldiers in the tree as dead (set the alive field of c_soldier to 0). This should be accomplished in O(m) time, where m is the number of men who will die, given the value of the *left_no* field at each node.

In the event that the Gods favor the troops of the Greeks (e.g. the battles in Rhapsodies D, E, Y and F), then the losses are less. In this case the way you will choose which soldiers will lose their lives is the soldiers corresponding to a path of the combat tree which you will choose each time as follows:

You will use the <bit_stream> parameter, which is in the form of a string of bits. These bits represent the path you will follow in the battle tree, with the aim of killing each soldier you visit. You will read these bits in order and whenever the current bit is 0, you will follow the lc pointer of the current node in the battle tree, while if the bit is 1, you will follow the rc pointer. For each node you visit, you will change the value of the alive variable to 0, thus indicating the death of the corresponding soldier.

In case you reach a leaf and there are more bits left in the bit_stream, the algorithm terminates.

In the event that you have used all the bits of the bit_stream and still not reached a leaf, the algorithm should use the same bit_stream from the beginning and continue the traversal (i.e. reusing the bit stream), as described above.

After the execution of such an event the program should print the following information:

```
B <god_favor> <bit_stream>
        Combat soldiers: <sid1:alive1>, <sid2:alive2>, .          .  . <sidn:aliven>
DONE
```

where n is the number of battle tree nodes and for each i, 1 ÿ i ÿ n, <sidi> and <alivei> are the ID and state of the soldier corresponding to the i-th battle tree node (based on the interorder crossing).

- U

Armistice and collection of dead from the battlefield (23rd day, Rhapsody H381-432). An event in which the dead are collected from the battlefield. In this event, the soldiers who lost their lives in battle should be deleted from the trees of the soldiers of the generals. In doing so, you will traverse the combat tree (combat_s) and delete all dead soldiers from it (all deletions should be done with a traverse). To do this you should choose the appropriate crossing that allows deletions to be made correctly in one pass. Additionally, for each soldier that is dead (field alive=0), you will find out which general it belongs to (by looking at c_soldier's gid field) and then delete that soldier from its general tree. Finally, the soldier will also be deleted from the inventory hash table.

After the execution of such an event the program should print the following information:

```
U

        GENERAL LIST:
        <gid1>: <sid1,1> .     ·  . <sid1,n1>
        <gid2>: <sid2,1> .     ·  . <sid2,n2>
         ...
        <gidk>: <sidk,1> .     ·  . <sidk,nk>

        SOLDIERS HASH TABLE:
        <sid1,1>, <sid1,2>, ..., <sid1,m1>,
        <sid2,1>, <sid2,2>, ..., <sid2,m2>,
         ...
        <sidw,1>, <sidw,2>, ..., <sidw,mw>
DONE
```

where k is the number of nodes in the list of generals, for each i, 1 ÿ i ÿ k, ni is the number of nodes of the soldier tree of the ith node of the list of generals, and for each j, 1 ÿ j ÿ ni, and <sidi,j> is the ID of the j-th node in the i-th general's soldier tree (based on nested traversal) and where w is the number of positions of the soldier hash table, for each i, 1 ÿ i ÿ w , and for each j, 1 ÿ j ÿ mi, sidi,j is the identifier of the j-th node of the i-th chain of the hash table.

- W <gid>

Event of type print general's soldiers which signals the printing of all soldiers of a general. For the general with id <gid> his soldier tree should be printed. After the execution of such an event the program should print the following information:

```
W <gid>
        Soldier tree = <sid1>, <sid2>, ..., DONE                    <sidn>
```

where n is the number of nodes of the general's soldier tree with id <gid> and for each i ÿ {1, …, n}, <sidi> is the soldier id corresponding to the i-th node of the general's soldier tree (based on interorder crossing).

- X

A print generals type event that signals the printing of all generals. For each general all their details should be printed, including their troop tree. After the execution of such an event the program should print the following information:

```
X
        General:
        <gid1>: <sid1,1> .      ·   . <sid1,n1>
        <gid2>: <sid2,1> .      ·   . <sid2,n2>
         ...
        <gidk>: <sidk,1> .      ·   . <sidk,nk>
DONE
```

where k is the number of nodes in the list of generals, for each i, 1 ÿ i ÿ k, ni is the number of nodes of the soldier tree of the ith node of the list of generals, and for each j, 1 ÿ j ÿ ni, and <sidi,j> is the ID of the j-th node in the soldier tree of the i-th general (based on interordered traversal).

- Y

Event which signals the printing of all soldiers of the soldier hash table. After the execution of such an event the program should print it following information:

```
Y

        Registration list = <sid1:gid1>, <sid2:gid2>, ..., <sidn:gidn>
DONE
```

where n is the number of entries of the hash table and for each i ÿ {1, …, n}, <sidi> is the ID of the soldier corresponding to the i-th entry of this table and <gidi> is the ID of the general in which the soldier corresponding to the i-th entry of the table obeys.

Data structures

Your implementation may not use ready-made data structures (eg, ArrayList) whether the implementation is in C or Java. Then the structures in C that must be used for the implementation of this work are presented.

```
struct soldier {
        int sid;
        int gid;
        struct soldier *next;
};
struct TREE_soldier {
        int sid;
        struct TREE_soldier *rc; struct
        TREE_soldier *lc;
        struct TREE_soldier *p;
};

struct general {
        int gid;
        int combats_no;
        struct TREE_soldier *soldiers_R;
        struct TREE_soldier *soldiers_S;
        struct general *next;
};

struct c_soldier {
        int sid;
        int alive;
        int gid;
        int left_cnt;
        struct c_soldier *lc;
        struct c_soldier *rc;
};
```

```
struct combat {
        int soldier_cnt;
        struct c_soldier *combat_s;
};

/* global, the maximum number of soldiers */ extern
unsigned int max_soldiers_g;

|/* global, the registration hashtable */
struct soldier *registration_hashtable;

/* global, the generals list*/
struct general *generals_list;

/* global, variable holding the combat info */
```