CS240: Data Structures

Fall Semester – Academic Year 2017-2018

Professor: Panagiota Faturu

Project – part B

Due Date: Monday, 22 December 2017, time 23:59.

Turnin Method: Using the turnin program. Information about how turnin works are supplied by the course website.

General Description

In this project you are asked to create a program that simulates the Trojan War, between Greeks (Achaeans by Homer) and Trojans, under the walls of Troy. The events of this project concern the concentration of the Greeks for campaigning and redeeming the city thanks to the trick of the Trojan Horse. This war is one of the main events of Greek Mythology and was the source of inexhaustible inspiration for ancient Greek literature, including the works of Homer: the Iliad and the Odyssey. In the second part of this project you will focus on data structures such as trees and hash tables.

Detailed Description of Requested Implementation

The registration (Rhapsody B, Day 22). "*During the preparations, the poet invokes the help of the Muses to quote the long list of the warriors*" (Homeric Epics: Iliad II Secondary High School). One of the tasks of this project is the soldiers registration on the battlefield. The **registration hash table** will hold information about the soldiers during the registration process. To handle the collitions, you have to use the seperate chaining method and the hash function *h(k) = k mod N*, where k is the key and N is the size of the hash table. During the parsing of the test files, the global variable max_soldiers_g will be initialized. The value of this variable indicates the total number of soldiers that will take part to the campaign. Based on this value you will choose the size of the hash table. Each node in the hash table lists is *soldier* record with the following fields:

- **sid:** Unique identifier of soldier (type: int).
- **gid:** Unique identified of the general who command the soldier (type: int)

- **next:** Pointer to the next node of the chain list (type: soldier pointer).

Figure 1 depicts an instance of the registration hash table.

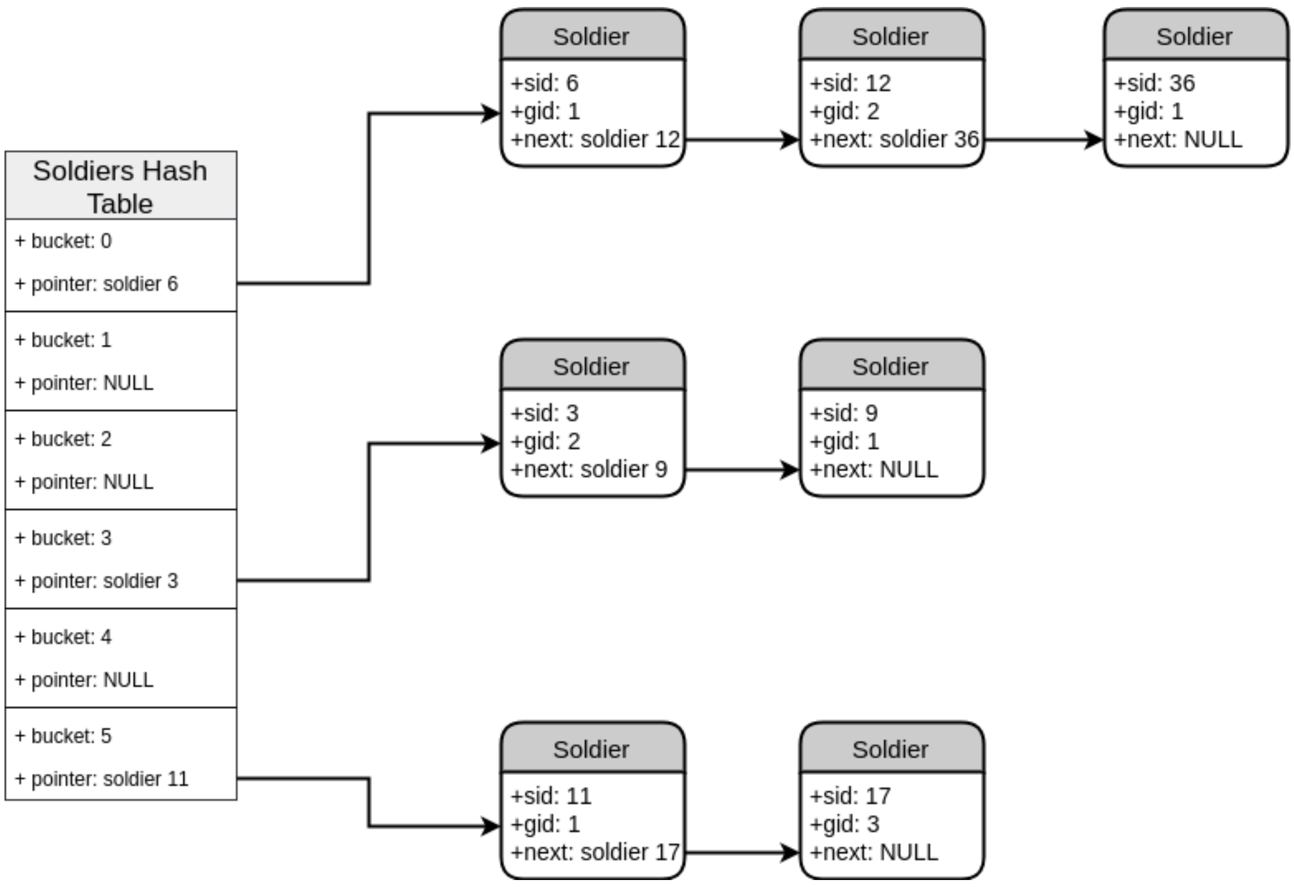

Figure 1The registration hash table

The greek army organized in groups. A brave kings or generals (such as Agamemnon, Menelaus, Nestor, Odysseus, Achilles, Diomedes and others) is in charge for each group. Each king or general has his own army (a group of soldiers) camped by the ships that carried the soldiers to Troy. Information about kings and generals is stored in an unsorted, singly linked list. The list is called **generals list**. Each node of the list is a *general* record consisting of the following fields:

- **gid:** Unique identifier of the general (type: int).
- **combats_no:** The number of battles in which the general has participated in (type: int).
- **soldiers_R:** Pointer to the root of a binary search tree, sorted by the soldiers identifiers (type: TREE_soldier). The tree is double linked and is called **soldiers tree** and represent the soldiers that general commands. Each node of the tree is a *TREE_soldier* record consisting fo the following fields:
    - **sid**: Uniqur identifier of the soldier (type: int).
    - **rc**: Pointer the right child of the node (type: TREE_soldier pointer).
    - **lc**: Pointer to the left child of the node (type: TREE_soldier pointer).
    - **p**: Pointer to the parent of the node (type: TREE_soldier pointer).
- **soldiers_S:** Pointer to the sentinel node of the soldiers tree of the general (type: TREE_soldier pointer).
- **next:** Pointer to the next node of generals list (type: general pointer).

The sentinel node is a TREE_soldier record with a value -1 in the gid and combats_no fields, while the soldiers_R and soldiers_S fields are NULL.

Figure 2 depicts an instance of the generals list. Each node of the list has a pointer to the root of a soldiers tree.

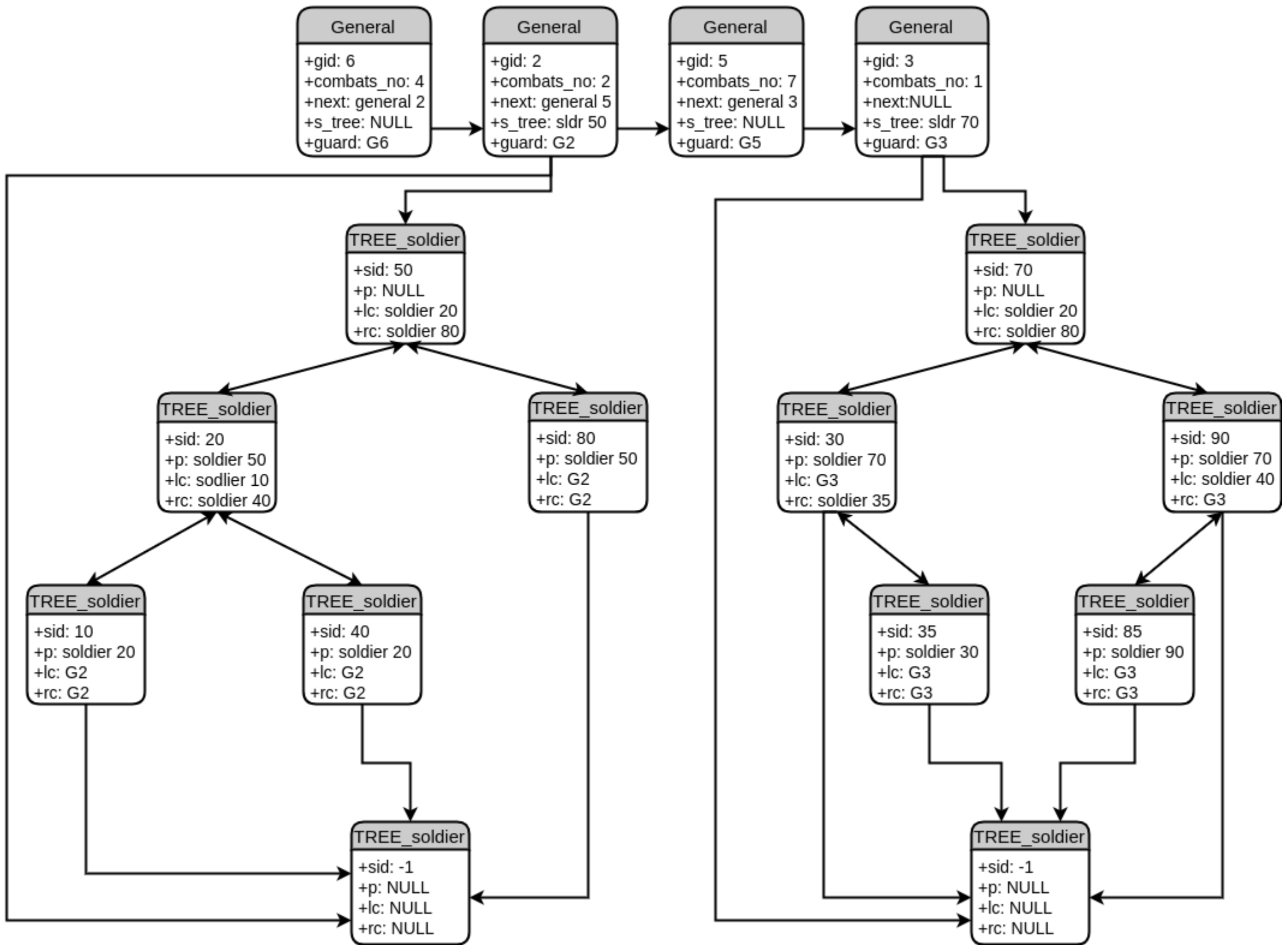

Figure 2 The unsorted, single linked list, called generals list. Each node of the list has a pointer to the root of a binary search tree sorted by the soldiers' identifiers. Each soldiers tree has a sentinel node.

Battles in Rhapsodies D and E (22nd day), Theta (25th day), L, M and N (26th day, battles around the Achaean wall), P and R (27th day, Patroklos leads the Myrmedons to battle), F (Achilles returns to the battle and faces, apart from the Trojans, the river Skamandros that rushes against him).

Next you will simulate the battles. A battle involves two generals and their corresponding soldiers. Soldiers that are involved in a battle are stored in the **combat** data structure consisting of the following fields:

- **soldiers_cnt**: The total number of soldiers involved in the battle (type: int).
- **combat_s**: Pointer (of type c_soldier) at the root of a binary search tree, sorted by the soldiers identifie, called **combat tree.** Each node of the tree is a c_soldier record consisting of the following fields:
    - **sid**: Unique identifier of the soldier (type: int).
    - **alive**: The status of the soldier (type: int). The acceptable values of this field is 0 or 1. Value 1 indicates that the soldier is alive, otherwise he is dead.

- **gid**: Unique identifier of the general who commands the soldier (type: int).
- **rc**: Pointer (of type c_soldier) to the right child of the node.
- **lc**: Pointer (of type c_soldier) to the left child of the node.
- **left_no:** The number of nodes contained in the left subtree of the node (type: int).

Figure 3 presents and instance of the combat data structure consisting of the fields described above.

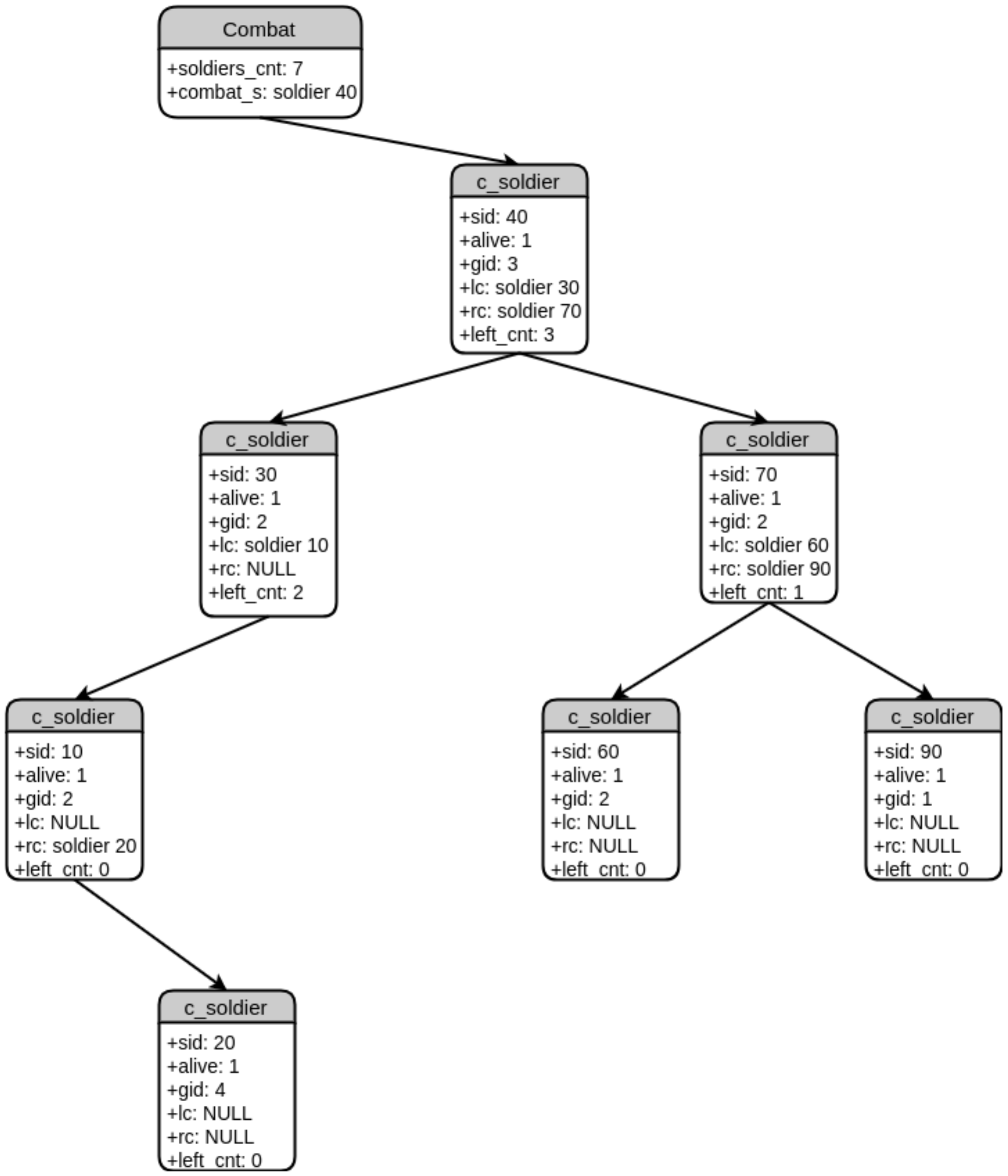

Figure 3 The combat data structure consising of a counter and a binary search tree, the combat tree.

How to run

The executable gets a parameter with the input test file. To execute type the following command:

<executable> <input-file>

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

– **R <sid> <gid>**

Event indicating the registration of the soldiers on the battlefield **(Rhapsody B [48-877], Day 22)**. In this event you will insert a new soldier to the registration hash table. The soldier's unique identifier is <sid> and the general that commands him is <gid>. During the insertion you have to use the hash function described above. Upon completion of the event, you should print the following:

```
R <sid> <gid>
    Soldiers 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
```

for i in {1, … k}, n_i is the size of the list contained in the i-th bucket of the hashtable and for j in {1, …, n_i} $sid_{i,j}$ is the soldiers identifier of the j-th node of the list in the i-th bucket of the hash table and $gid_{i,j}$ is the id of the general that commands the soldier $sid_{i,j}$.

– **G <gid>**

Event indicating that the king/general with identifier <gid> participates in the campaign. In this event you have to insert a new general with <gid> in the generals list. You should insert the node with the optimal complexity. Upon completion of the event, you should print the following:

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

where n is the number of nodes in the list of generals and for each i ∈ {1, ..., n}, <gid_i> is the general id corresponding to the i-th node in that list.

– **D**

Distribute soldiers type event that marks evening rest. The soldiers reside in the camps they belong to (according to the general who commands them). You have to iterate through the registration hash table and fill the soldiers tree of each general. For each soldier in the registration hash table, you have to search the generals list for the corresponding general based on the value of the gid fields of each soldier.

Then insert the soldier to the soldiers tree of the general. At the end of the event a copy of each soldier of the registration hash table should be placed in the corresponding soldiers tree in the generals list.

Upon completion of the event, you should print the following:

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

where for each i, $1 \leq i \leq k$, n_i is the size of the tree of soldiers in the i-th node of the generals list, and for each j, $1 \leq j \leq n_i$, και $<sid_{i,j}>$ is the identifier of the j-th node in the tree of soldiers of the i-th general.

– **M $<gid_1>$ $<gid_2>$**

This event simulates that Achilles left the battle (after his argument with Agamemnon), while Patroclos convinces Achilles to lead the Myrmoneses into the battle.

Consider that Achilles' identifier is $<gid_1>$ and Patroklos identifier is $<gid_2>$. You must delete the node with identifier $<gid_1>$ from the list of generals. In addition, Patroclos is now in charge for the Achilles soldiers as well[1]. In other words, the soldiers of the TREE_soldier corresponding to the general with id $<gid_1>$ should be moved into the TREE_soldier corresponding to the general with id $<gid_2>$. This merge operation should have O(n) complexity, with n the total number of soldiers of both trees.

To merge the trees, you can use two helper arrays. You can copy the soldiers of the first tree (Achilles TREE_soldier) into the first array and the soldiers of the second tree (Patroclos TREE_soldier) to the second array. Both tables should be sorted by the soldiers' id. Next you can merge the tables and the results will be stored in a third helper array. The resulting array contains the soldiers of both Achilles and Patroclos and is sorted by the soldiers' id as well. The merge operation of the tables should be achieved with O(n) complexity. Next you have to reconstruct the Patroclos' TREE_soldier tree with the contents of the third array. The height of the tree should be O(logn) and be sorted by the soldiers' id.

Upon completion of the event, you should print the following:

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

where for each i, $1 \leq i \leq k$, n_i is the size of the soldiers tree in the i-th node of the list of generals, and for each j, $1 \leq j \leq n_i$, και $<sid_{i,j}>$ is the identifier of the j-th node of the soldiers tree of the i-th general.

- **P <gid$_1$> <gid$_2$>**

Prepare for battle, indicates that generals <gid1> and <gid2> prepare their soldiers for the next battle. For each of the generals <gid1> and <gid2>, you will increase the combat counter combats_no, indicating the number of battles that each general has participated in.

The soldiers who will participate in the battle will be inserted in the battle tree of the combat record. Each node of the tree is a c_soldier record. When you insert a new node into the tree, you should also update the soldiers_cnt field of the compbat record. You should insert a new node in the tree according to the following steps:

First you have to insert the soldier with the smallest sid from the TREE_soldier tree corresponding to the general with id <gid1>. Then you will insert the soldier with the highest sid in the TREE_soldier tree of the <gid2> general. Then you will insert the soldier with the second smallest sid of the soldiers tree of <gid1> general. After that you will insert the soldier with the second highest sid of the soldiers tree of <gid2> general. In every step you will choose a soldier of one of the generals and in the next step you will choose a soldier from the other general. You will continue this method until tou insert all soldiers of both generals.

To do so, you can implement two helper methods: *inorder_successor()* and *inorder_predeccesor(),* that will be used to get the next soldier with the smallest or highest sid according to the in-order tree traversal in a TREE_soldier tree. Every time you want to select a node from soldier tree of generals <gid1> or <gid2>, you can use these methods. For each soldier you insert in the battle tree of the combat structure, you will set the value of alive = 1 to the soldier being alive.

Figure 4 presents an example of soldier trees corresponding to generals with ids 10 and 20. These generals will participate in the battle. Figure 5 shows the result of merging the soldiers of the generals to the battle tree based on the algorithm described earlier.

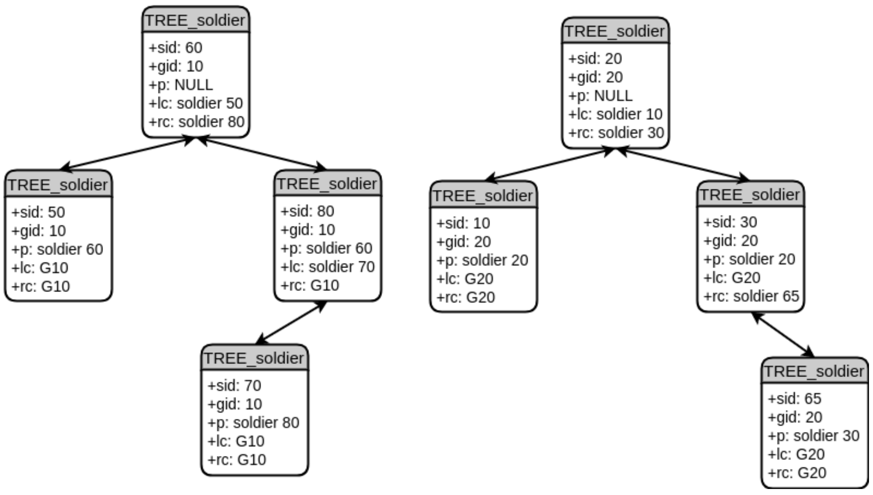

Figure 4 soldiers trees of generals with ids 10 and 20

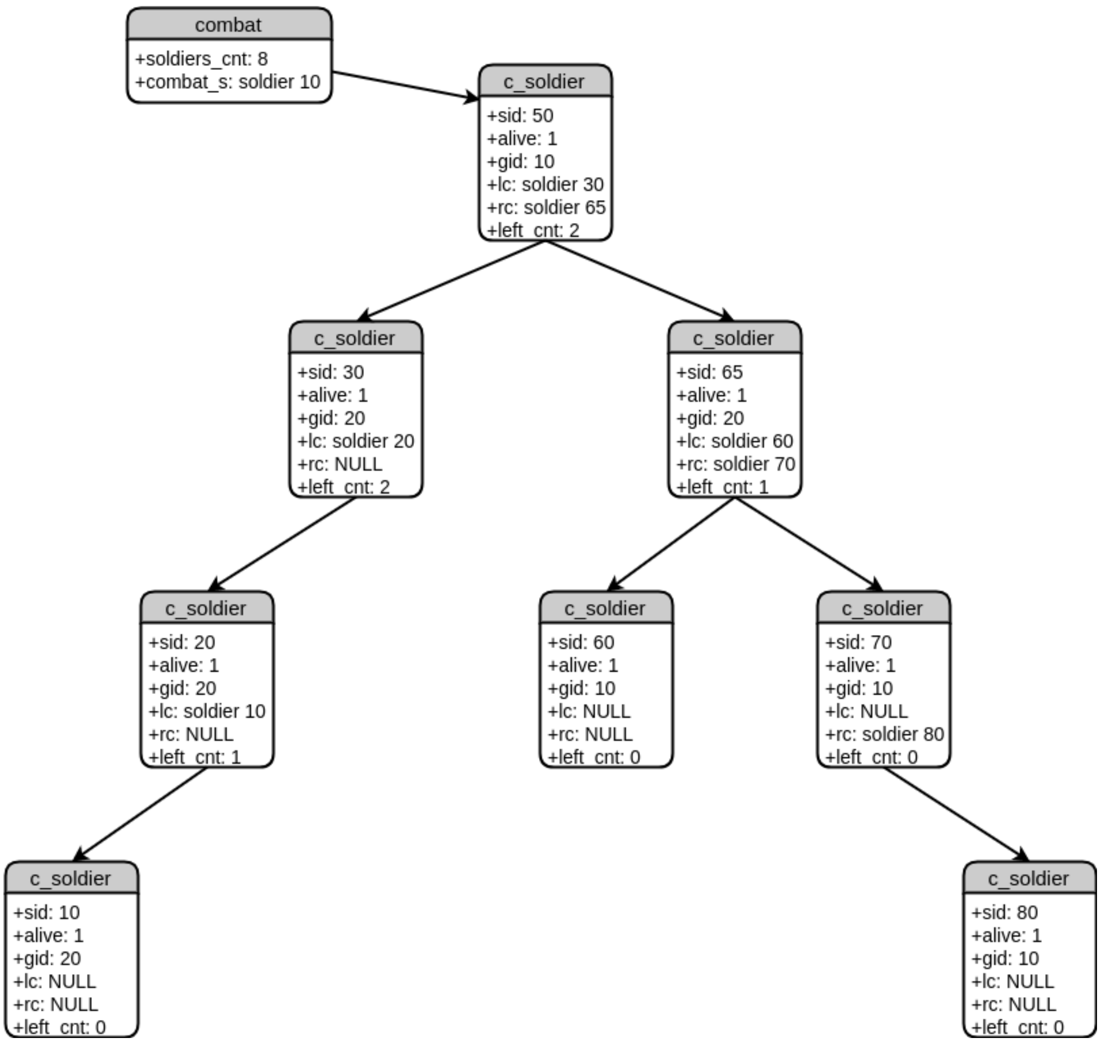

Figure 5 The resulting combat tree. The order of the insetions: 50, 65, 60, 30,70, 20, 80, 10

Upon completion of the event, you should print the following:

```
P <gid1> <gid2> <gid3>

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

DONE
```

where n is the size of the battle tree and for each i, $1 \leq i \leq n$, $<sid_i>$ is the i-th soldier's identifier on the battle tree according to the in-order tree traversal.

– **B <god_favor> <bit_stream>**

Event that simulates the battle (rapsodies D, E, L, M, and N). During the battle some of the soldiers die. The amount of loss is affected by the god’s favor against the Greeks as defined by the parameter <god_favor> of this event.

If the gods do not favor the Greeks (e.g. in the battles of the 25th and 26th day, including the battles around the Achaean walls, Rhapsodies R, L, M, and N), then the losses amount to 40% of the soldiers that participate in the battle. Using the soldiers_cnt counter of the combat record, you can identify the number of soldiers that die in the battle. Then you will traverse the combat tree of the combat record and mark 40% of the soldiers as dead (set field alive = 0). This process should be achieved with complexity O(m), with m the number of dead soldiers. Keep in mind that you can use the filed *left_no* of each node of the tree.

On the other hand, if Gods favor the Greeks (e.g. battles described in Rhapsodies D, E, Y and F Rays), then the losses are 10% of the participating soldiers. In this case you mark the dead soldiers as follows:

The <bit_stream> parameter of the event will indicate the path that you will follow while traversing the combat tree. The parameter is a string representation of a bit stream (a string consisting of '0' and '1' characters only). You have to iterate through the charactes of the <bit_stream> parameters and starting with the root of the combat tree, you will follow down the path based on the value of each character. If the the character is '0', you will choose the left child (lc) of the tree node, otherwise you will choose the right chils (rc). Next you will use the next character of the <bit_stream> to choose the next tree node that you will visit. You repeat this process until you hit a leaf node of the tree. In case you have already consumed all the characters of the <bit_stream> but you haven't hit a leaf node yet, your algorithm should continue the tree traversal by using the same <bit_stream> from the beginning. Each soldier (node of the tree) you visit with this method, you will mark him as dead (set field alive = 0).

Upon completion of the event, you should print the following:

```
B <god_favor>

    Combat soldiers: <sid1:alive1>, <sid2:alive2>, . . . <sidn:aliven>

DONE
```

where n is the size of the battle tree and for each i, $1 \leq i \leq n$, <sid$_i$> και <alive$_i$> is the identifier and status of the soldier corresponding to the i-th node in the battle tree according to the in-order tree traversal.

– U

Truce and burial of the dead (day 23, Rhapsody H381-432). Event of burial of the dead. In this event, the soldiers who have lost their lives in the battle should be deleted from the corresponding soldier trees in generals list. You have to traverse through the battle tree of the combat record and delete all dead soldiers (records with field alive = 0). You should be able to delete all nodes with dead soldiers with a single tree traversal. To do so, you have to identify the appropriate tree traversal method. In addition, for every dead soldier, you have to identify the general that is in charge (use the gid field of the each c_soldier record) and delete the soldier from the corresponding TREE_soldier tree. Moreover you should alse delete the record of every dead soldier in the registration hash table.

After the completion of the event, you should print the following:

```
U
    GENERALS LIST:
    <gid_1>: <sid_1,1> . . . <sid_1,n1>
    <gid_2>: <sid_2,1> . . . <sid_2,n2>
    ...
    <gid_k>: <sid_k,1> . . . <sid_k,nk>
DONE
```

where for each i, $1 \le i \le k$, n_i is the size of the soldiers tree in the i-th node of the generals list, and for each j, $1 \le j \le n_i$, και $<sid_{i,j}>$ is the identifier of the j-th node of the soldiers tree of the i-th general (according to the in-order tree traversal).

- W <gid>

Print the soldiers of general with id <gid>. You should print the nodes of TREE_soldier corresponding to general <gid>. After the completion of the event, you should print the following:

```
W <gid>
    Soldier tree = <sid_1>, <sid_2>, ..., <sid_n>
DONE
```

where n is the number of nodes of soldiers tree corresponding to general with id <gid> and for i $\in$ {1, …, n}, $<sid_i>$ is the identifier of soldier corresponding to the i-th node of the tree (according to the in-order traversal).

– X

A *print generals* event that marks the printing of all generals. For each general, you must print all the details, including soldiers tree. After the completion of the event, you should print the following:

```
X
    GENERALS:
    <gid_1>: <sid_1,1> . . . <sid_1,n1>
    <gid_2>: <sid_2,1> . . . <sid_2,n2>
    ...
    <gid_k>: <sid_k,1> . . . <sid_k,nk>
DONE
```

where for each i, $1 \le i \le k$, n_i is the size of soldiers tree in the i-th node of the list of generals, and for each j, $1 \le j \le n_i$, και $<sid_{i,j}>$ is the identifier of the j-th node of the soldiers tree of the i-th general (according to the in-order traversal).

– **Y**

Print all soldiers contained in the registration hash table. Upon completion of the event, you should print the following:

```
Y
    Registration hash = <sid_1:gid_1>, <sid_2:gid_2>, ..., <sid_n:gid_n>
DONE
```

where n is the number of soldiers in the registration hash table and for each $i \in \{1, \dots, n\}$, $<sid_i>$ is the soldier's identifier corresponding to the i-th node node in the hash table and $<gid_i>$ the identifier of the general who the i-th soldier obeys to (according to the in-order traversal)

Data structures

In your implementation you are not allowed to use libraries providing data structures implementations provided by the language. You can use the following data stractures to implement the functionality described above.

```
struct soldier {
      int sid;
      int gid;
      struct soldier *next;
};
struct TREE_soldier {
      int sid;
      struct TPEE_soldier *rc;
      struct TPEE_soldier *lc;
      struct TPEE_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 */
```