# VG101 — Intoduction to Computers & Programming

*Lab 5*

Instructor: Manuel Charlemagne

TA: Yifei – UM-JI (Summer 2019)

**Goals of the lab**

- Generate random number
- Revisit structure
- Revisit recursion
- Practice array and pointer

## 1 Introduction

Haruka, Kana, and Chiaki are still busy with many many exams this week, but Haruka recalls the excitement the World Cup brought to her last year, so despite for the heavy workload, she decides to review all the matches in last year's World Cup.

Haruka also recalls that it was the unpredictable result of each match that seized her heart, so she goes to browse the prediction of the match outcome published before the starting of the World Cup (Fig 1).

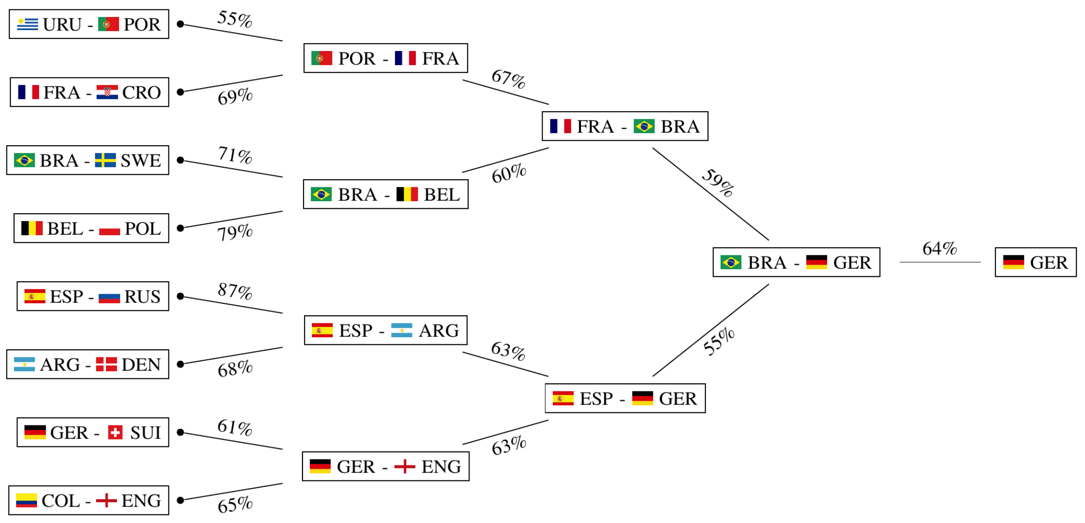

Figure 1: Predicted match map of 2018 World Cup

She soon finds out that the actual result of 2018 World Cup is far different from the predicted situation, and she wants to simulate the matching process by herself to figure out what can be other potential outcomes of the World Cup results.

## 2 Match Simulation

### 2.1 Simulation Model

If two teams $A$ and $B$ are going to have a match, and their world rank is $r_A$ and $r_B$ respectively, we can define the winning rate of the match as

$$\text{rate} = \frac{r_A + k}{r_A + r_B + 2k}$$

in which $k$ is an arbitrary coefficient specified by the input.

Haruka still feels confused when she is asked to simulate the result of a match based on the winning rate, so she asks Jane for help. Jane tells her that she can do this by generating a random number $w \in [0, 1)$, so that if $w \geq$ rate, team $A$ will be regarded as the winner of the match, while if $w <$ rate, team $B$ will be the winner.

For generating a random number, we can use the function `rand` provided by standard library. However, we also need to preset a random seed for the `rand` function. Moreover, if the random seed is a fixed value, the outcome will also be fixed (you can check it by yourself), so now, Haruka decides to search keyword *rand* in the official documentation to get to know how to generate random numbers by using this function.

### 2.2 Implementation

Based on our simulation model, we used the following structure to represent a country

```c
typedef struct country {
    int rank;
    char name[4];
} country_t;
```

Implement the following function to simulate a single match

```c
// get_winner would simulate the game between first and second
// it would print the game information to stdout and return the winner
// printing format:
// FIRSTNAME WIN SECONDNAME
// FIRSTNAME LOSE SECONDNAME
country_t get_winner(const country_t first, const country_t second, int k);
```

## 3 The World Cup

After observing and analyzing Fig 1, Haruka encounters another difficulty. How can she make sure that all the simulations of matches will happen in correct order? Through some search, clever Haruka observes that Fig 1 is similar to a tree structure, and there is a tree traversal algorithm called post-order traversal, which can exactly meet with the requirements.

## 3.1 Binary Tree

A binary tree is a tree data structure. It contains either

- nothing
- a piece of data that we care about, together with the information about 2 other binary trees, usually referred as children

In this scenario, we can use a binary tree to store the record of the world cup. Each binary tree contains the information about a country $\alpha$ and 2 other binary trees, *left* and *right*. $\alpha$ is defined as the winner of the 2 countries stored in *left* and *right*.

Take Figure 1 as an example, one node contains Brazil as a runner-up and 2 children. One child contains Brazil itself, and the other child contains France, which was defeated by Brazil. The runner-up node is a child of another tree, which contains the Champion Germany.

To visit all the elements in the tree and do operations on them, there are mainly 3 types of depth first searches. These types are represented by the following pseudo code.

```
function pre_order(tree) {
    if (tree is not empty) {
        do_operation_on(root data of tree);
        pre_order(left child of tree);
        pre_order(right child of tree);
    }
}
```

```
function in_order(tree) {
    if (tree is not empty) {
        in_order(left child of tree);
        do_operation_on(root data of tree);
        in_order(right child of tree);
    }
}
```

```
function post_order(tree) {
    if (tree is not empty) {
        post_order(left child of tree);
        post_order(right child of tree);
        do_operation_on(root data of tree);
    }
}
```

## 3.2 Array Based Implementation

Without the knowledge of pointer, a binary tree can be implemented as an array. If we want the champion Germany be node 1, the 2nd and 3rd node would contain Brazil and Germany. The 4th node to 7th node would contains France, Brazil, Spain, and Germany. So on and so forth. In this case, for a node with index $n$, the index of its left node is $2n$, the index of its right node is $2n+1$, and the index of its parent node is $n/2$.

Kana points out that, in C language, the array index starts from 0 instead of 1. Haruka says that the storing principle is the same if a 0 based index is used. In that case, for a node with index $n$, the index of its left node is $2n+1$, the index of its right node is $2n+2$, and the index of its parent node is $(n+1)/2-1$. She says that they can also put a dummy element at index 0 to keep the simpler index relationship of 1 based index.

To simulate the world cup. We would need to fill in all the nodes based on its 2 children. Haruka points out that, only in using post order traversal, the operation on the root data applied after both children are settled. Take Fig 1 as an example, the post order transverse should be URU-POR, FRA-CRO, POR-FRA, BRA-SWE, BEL-POL, BRA-BEL, FRA-BRA, ...

Implement this simulation together with Haruka and Kana. You would need to

1. Declare an array to simulate the tree.
2. Load the initial data from stdin to the leaves of the tree (i.e. the back of the array). The format is specified in 3.3.
3. Perform a post order traversal to fill in all the non-leaf nodes using the function `get_winner()`

## 3.3 Input and output format

The first line of the input only contains an integer $k$, each of the next 16 lines contains a country name of length 3 and its world ranking. The order of the countries is same as their order from top to bottom on the match map.

Take Figure 1 as an example, the sample input is (note that the win rate shown in the figure is not calculated by our method, so your result may be different):

```
0
URU 14
POR 4
FRA 7
CRO 20
BRA 2
SWE 24
BEL 3
POL 8
ESP 10
RUS 70
ARG 5
DEN 12
GER 1
SUI 6
COL 16
ENG 12
```

The output should be the result of the 15 matches between the 16 countries, ordered by the post-order traversal. The sample output can be

```
URU LOSE POR
FRA WIN CRO
POR LOSE FRA
BRA WIN SWE
BEL LOSE POL
BRA WIN POL
FRA LOSE BRA
ESP WIN RUS
ARG WIN DEN
ESP LOSE ARG
GER WIN SUI
COL LOSE ENG
GER WIN ENG
ARG LOSE GER
BRA LOSE GER
```

To avoid typing the long input again and again, you can make use of a terminal functionality called input redirection. By writing the input into `input.txt` and append `< input.txt` to your command, you can redirect the content of the file to stdin.

```
./predictor < input.txt
```

For windows users, omit the "./" at the start to make it work.

## 4 Week 2

### 4.1 Node based implementation

After Kana learns pointer and dynamic memory in the lecture, she is curious about whether they can be used to implement a binary tree. She proposes the following structure.

```c
typedef struct node {
    struct country *country;
    struct node *left, *right;
} node_t;
```

Help Kana to implement the world cup simulator using node based binary tree. Again:

1. Declare a root pointer.
2. Propagate the tree using `malloc(sizeof(node_t))`. Load the initial data from stdin to the leaf nodes of the tree, following the format specified in 3.3.
3. Perform a post order traversal to fill in all the non-leaf nodes using the function `get_winner()`
4. Free the all the dynamic memory used in the tree.

### 4.2 File Input & Output

Chiaki is taking a statistic course VE401 this semester. The topic of the course project is how random the pseudo random functions are in programming languages. She would like the some sample output of the world cup simulator to study the randomness of `rand()` in C.

To help Chiaki, adjust the simulator above. Instead of reading from stdin, the new simulator would read from a file `ranks.txt` in the current folder. Instead of writing to stdout, the new simulator would simulate `n` times and dump the output to `sampleX.txt`, where `X` ranges from 1 to n.

## 5 Optional Post Lab Exercises

- Rewrite the simulator structure without using a binary tree structure.
- Adjust `country_t`, so that it also includes randomly generated scores of both teams. You may want to rename the structure after the adjustment.
- Based on the tree, find the match where the most goals are made.

## 6 Acknowledgement

Thanks Yihao Liu, TA from previous semesters, for providing the idea and structures of this lab.