

# ELORank

P. Baillehache

October 9, 2018

## Contents

<b>1</b>	<b>Definitions</b>	<b>2</b>
<b>2</b>	<b>Interface</b>	<b>2</b>
<b>3</b>	<b>Code</b>	<b>4</b>
3.1	<code>elorank.c</code> . . . . .	4
3.2	<code>elorank-inline.c</code> . . . . .	9
<b>4</b>	<b>Makefile</b>	<b>10</b>
<b>5</b>	<b>Unit tests</b>	<b>11</b>
<b>6</b>	<b>Unit tests output</b>	<b>14</b>
<b>7</b>	<b>ELORank.txt</b>	<b>14</b>

## Introduction

ELORank is a C library providing structures and functions implementing the ELO ranking system in a version supporting several players per run with eventual ties.

It uses the `PBErr`, `PBMath` and `GSet` library.

# 1 Definitions

The ELO rank is calculated incrementally by updating the current ELO rank of each entity according to their result in an evaluation process independent from the ELO ranking system. Given a result of this evaluation process, each pair of winner/looser in this result is updated as follow:

$$\begin{cases} E'_w = E_w + K * \left( 1.0 - \frac{1.0}{1.0 + 10.0^{\frac{E_l - E_w}{400.0}}} \right) \\ E'_l = E_l - K * \left( \frac{1.0}{1.0 + 10.0^{\frac{E_w - E_l}{400.0}}} \right) \end{cases} \quad (1)$$

where  $K = 8.0$  and,  $E_w$  and  $E_l$  are respectively the current ELO of the winner and the current ELO of the loser and,  $E'_w$  and  $E'_l$  are respectively the new ELO of the winner and the new ELO of the loser.

Tie between two entities results in no changes in their respective ELO rank.

# 2 Interface

```
// ===== ELORANK.H =====

#ifndef ELORANK_H
#define ELORANK_H

// ===== Include =====

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "gset.h"
#include "pbmath.h"

// ===== Define =====

#define ELORANK_K 8.0
#define ELORANK_STARTELO 0.0

// ===== Data structure =====

typedef struct ELOEntity {
    // Pointer toward user struct
    void* _data;
    // Number of evaluation
    int _nbrRun;
} ELOEntity;
```

```

typedef struct ELORank {
    // ELO coefficient
    float _k;
    // Set of ELO entities
    GSet _set;
} ELORank;

// ===== Functions declaration =====

// Create a new ELORank
ELORank* ELORankCreate(void);
/**if BUILDMODE == 0
ELORank ELORankCreateStatic(void);
#endif*/

// Free memory used by an ELORank
void ELORankFree(ELORank** that);

// Free memory used by an ELOEntity
void ELOEntityFree(ELOEntity** that);

// Set the K coefficient of 'that' to 'k'
#if BUILDMODE != 0
inline
#endif
void ELORankSetK(ELORank* const that, const float k);

// Get the K coefficient of 'that'
#if BUILDMODE != 0
inline
#endif
float ELORankGetK(const ELORank* const that);

// Add the entity 'data' to 'that'
void ELORankAdd(ELORank* const that, void* const data);

// Remove the entity 'data' from 'that'
void ELORankRemove(ELORank* const that, void* data);

// Get the number of entity in 'that'
#if BUILDMODE != 0
inline
#endif
int ELORankGetNb(const ELORank* const that);

// Update the ranks in 'that' with results 'res' given as a GSet of
// pointers toward entities (_data in GSetElem equals _data in
// ELOEntity) in winning order
// The _sortVal of the GSet represents the score (and so position)
// of the entities for this update (thus, equal _sortVal means tie)
// The set of results must contain at least 2 elements
// Elements in the result set must be in the ELORank
void ELORankUpdate(ELORank* const that, const GSet* const res);

// Get the current rank of the entity 'data' (starts at 0)
int ELORankGetRank(const ELORank* const that, const void* const data);

// Get the current ELO of the entity 'data'
float ELORankGetELO(const ELORank* const that, const void* const data);

```

```

// Get the 'rank'-th entity according to current ELO of 'that'
// (starts at 0)
const ELOEntity* ELORankGetRanked(const ELORank* const that, const int rank);

// ===== Inliner =====

#if BUILDMODE != 0
#include "elorank-inline.c"
#endif

#endif

```

## 3 Code

### 3.1 elorank.c

```

// ===== ELORANK.C =====

// ===== Include =====

#include "elorank.h"
#if BUILDMODE == 0
#include "elorank-inline.c"
#endif

// ===== Functions declaration =====

// Create a new ELOEntity
static ELOEntity* ELOEntityCreate(void* const data);

// Return the GSetElem in 'that'->_set for the entity 'data'
static GSetElem* ELORankGetElem(const ELORank* const that, const void* const data);

// ===== Functions implementation =====

// Create a new ELORank
ELORank* ELORankCreate(void) {
    // Allocate memory
    ELORank* that = PBErrMalloc(ELORankErr, sizeof(ELORank));
    // Set the default coefficient
    that->_k = ELORANK_K;
    // Create the set of entities
    that->_set = GSetCreateStatic();
    // Return the new ELORank
    return that;
}

// Free memory used by an ELORank
void ELORankFree(ELORank** that) {
    // Check the argument
    if (that == NULL || *that == NULL) return;
    // Empty the set of entities
    GSet* set = &((*that)->_set);
    while (GSetNbElem(set) > 0) {
        ELOEntity *ent = GSetPop(set);
        ELOEntityFree(&ent);
    }
}

```

```

    // Free memory
    free(*that);
    // Set the pointer to null
    *that = NULL;
}

// Free memory used by an ELOEntity
void ELOEntityFree(ELOEntity** that) {
    // Check the argument
    if (that == NULL || *that == NULL) return;
    // Free memory
    free(*that);
    // Set the pointer to null
    *that = NULL;
}

// Add the entity 'data' to 'that'
void ELORankAdd(ELORank* const that, void* const data) {
#ifdef BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PErrCatch(ELORankErr);
    }
    if (data == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'data' is null");
        PErrCatch(ELORankErr);
    }
}
#endif
    // Create a new ELOEntity
    ELOEntity *ent = ELOEntityCreate(data);
    // Add the new entity to the set with a default score
    GSetAddSort(&(that->_set), ent, ELORANK_STARTELO);
}

// Create a new ELOEntity
static ELOEntity* ELOEntityCreate(void* const data) {
#ifdef BUILDMODE == 0
    // Check argument
    if (data == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'data' is null");
        PErrCatch(ELORankErr);
    }
}
#endif
    // Allocate memory
    ELOEntity *that = PErrMalloc(ELORankErr, sizeof(ELOEntity));
    // Set properties
    that->_data = data;
    that->_nbRun = 0;
    // Return the new ELOEntity
    return that;
}

// Remove the entity 'data' from 'that'
void ELORankRemove(ELORank* const that, void* data) {
#ifdef BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;

```

```

        sprintf(ELORankErr->_msg, "'that' is null");
        PBErCatch(ELORankErr);
    }
    if (data == NULL) {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg, "'data' is null");
        PBErCatch(ELORankErr);
    }
#endif
    // Search the entity
    GSetElem* elem = ELORankGetElem(that, data);
    // If we have found the entity
    if (elem != NULL) {
        // Free the memory
        ELOEntityFree((ELOEntity**>(&(elem->_data)));
        // Remove the element
        GSetRemoveElem(&(that->_set), &elem);
    }
}

// Return the GSetElem in 'that'->_set for the entity 'data'
static GSetElem* ELORankGetElem(const ELORank* const that, const void* const data) {
#if BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PBErCatch(ELORankErr);
    }
    if (data == NULL) {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg, "'data' is null");
        PBErCatch(ELORankErr);
    }
#endif
    // Search the element
    GSetElem* elem = that->_set._head;
    while (elem != NULL && ((ELOEntity*)(elem->_data))->_data != data)
        elem = elem->_next;
    // Return the element
    return elem;
}

// Update the ranks in 'that' with results 'res' given as a GSet of
// pointers toward entities (_data in GSetElem equals _data in
// ELOEntity) in winning order
// The _sortVal of the GSet represents the score (and so position)
// of the entities for this update (thus, equal _sortVal means tie)
// The set of results must contain at least 2 elements
// Elements in the result set must be in the ELORank
void ELORankUpdate(ELORank* const that, const GSet* const res) {
#if BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PBErCatch(ELORankErr);
    }
    if (res == NULL) {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg, "'res' is null");
        PBErCatch(ELORankErr);
    }

```

```

    }
    if (GSetNbElem(res) < 2) {
        ELORankErr->_type = PErrTypeInvalidArg;
        sprintf(ELORankErr->_msg,
            "Number of elements in result set invalid (%ld>=2)",
            GSetNbElem(res));
        PErrCatch(ELORankErr);
    }
#endif
    // Declare a variable to memorise the delta of elo of each entity
    VecFloat* deltaElo = VecFloatCreate(GSetNbElem(res));
    // Calculate the delta of elo for each pair of entity
    GSetElem* elemA = res->_head;
    int iElem = 0;
    while (elemA != NULL) {
        GSetElem* elemAElo = ELORankGetElem(that, elemA->_data);
#ifdef BUILDMODE == 0
        if (elemAElo == NULL) {
            ELORankErr->_type = PErrTypeNullPointer;
            sprintf(ELORankErr->_msg,
                "Entity in the result set can't be found in the ELORank.");
            PErrCatch(ELORankErr);
        }
#endif
        GSetElem* elemB = res->_head;
        while (elemB != NULL) {
            // Ignore tie and match with itself
            if (ISEQUALF(elemA->_sortVal, elemB->_sortVal) == false) {
                GSetElem* elemBElo = ELORankGetElem(that, elemB->_data);
#ifdef BUILDMODE == 0
                if (elemBElo == NULL) {
                    ELORankErr->_type = PErrTypeNullPointer;
                    sprintf(ELORankErr->_msg,
                        "Entity in the result set can't be found in the ELORank.");
                    PErrCatch(ELORankErr);
                }
#endif
                // If elemA has won
                if (elemA->_sortVal > elemB->_sortVal) {
                    float winnerELO = elemAElo->_sortVal;
                    float loserELO = elemBElo->_sortVal;
                    float a =
                        1.0 / (1.0 + pow(10.0, (loserELO - winnerELO) / 400.0));
                    VecSetAdd(deltaElo, iElem, that->_k * (1.0 - a));
                } // Else, if elemA has lost
                else {
                    float winnerELO = elemBElo->_sortVal;
                    float loserELO = elemAElo->_sortVal;
                    float b =
                        1.0 / (1.0 + pow(10.0, (winnerELO - loserELO) / 400.0));
                    VecSetAdd(deltaElo, iElem, -1.0 * that->_k * b);
                }
            }
            elemB = elemB->_next;
        }
        elemA = elemA->_next;
        ++iElem;
    }
    // Apply the delta of elo and update the number of run
    GSetElem* elem = res->_head;
    iElem = 0;
    while (elem != NULL) {

```

```

    GSetElem* elemElo = ELORankGetElem(that, elem->_data);
#if BUILDMODE == 0
    if (elemElo == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg,
            "Entity in the result set can't be found in the ELORank.");
        PErrCatch(ELORankErr);
    }
#endif
    elemElo->_sortVal += VecGet(deltaElo, iElem);
    ++(((ELOEntity*)(elemElo->_data))->_nbRun);
    ++iElem;
    elem = elem->_next;
}
// Free memory
VecFree(&deltaElo);
// Sort the ELORank
GSetSort(&(that->_set));
}

// Get the current rank of the entity 'data' (starts at 0)
int ELORankGetRank(const ELORank* const that, const void* const data) {
#if BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PErrCatch(ELORankErr);
    }
    if (data == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'data' is null");
        PErrCatch(ELORankErr);
    }
#endif
    // Declare a variable to memorize the rank
    int rank = 0;
    // Search the element
    GSetElem* elem = that->_set._tail;
    while (elem != NULL && ((ELOEntity*)(elem->_data))->_data != data) {
        elem = elem->_prev;
        ++rank;
    }
#if BUILDMODE == 0
    if (elem == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg,
            "Entity requested can't be found in the ELORank.");
        PErrCatch(ELORankErr);
    }
#endif
    // Return the element
    return rank;
}

// Get the current ELO of the entity 'data'
float ELORankGetELO(const ELORank* const that, const void* const data) {
#if BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
    }
#endif
}

```



```

        PBErCatch(ELORankErr);
    }
    if (data == NULL) {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg, "'data' is null");
        PBErCatch(ELORankErr);
    }
#endif
    // Declare a variable to memorize the ELO
    float elo = ELORANK_STARTELO;
    // Search the element
    GSetElem* elem = that->_set._head;
    while (elem != NULL && ((ELOEntity*)(elem->_data))->_data != data) {
        elem = elem->_next;
    }
    if (elem != NULL) {
        elo = elem->_sortVal;
    }
    #if BUILDMODE == 0
    } else {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg,
            "Entity requested can't be found in the ELORank.");
        PBErCatch(ELORankErr);
    }
#endif
    // Return the element
    return elo;
}

// Get the 'rank'-th entity according to current ELO of 'that'
// (starts at 0)
const ELOEntity* ELORankGetRanked(const ELORank* const that, const int rank) {
    #if BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PBErTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PBErCatch(ELORankErr);
    }
    if (rank < 0 || rank >= GSetNbElem(&(that->_set))) {
        ELORankErr->_type = PBErTypeInvalidArg;
        sprintf(ELORankErr->_msg, "'rank' is invalid (0<=%d<=%ld)", rank,
            GSetNbElem(&(that->_set)));
        PBErCatch(ELORankErr);
    }
    #endif
    GSetElem* elem = that->_set._tail;
    for (int i = rank; i--;)
        elem = elem->_prev;
    return (ELOEntity*)(elem->_data);
}

```

## 3.2 elorank-inline.c

```

// ===== ELORANK-INLINE.C =====

// ===== Functions implementation =====

// Set the K coefficient of 'that' to 'k'
#if BUILDMODE != 0

```

```

inline
#endif
void ELORankSetK(ELORank* const that, const float k) {
#if BUILDMODE == 0
    // Check arguments
    if (that == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PErrCatch(ELORankErr);
    }
#endif
    that->_k = k;
}

// Get the K coefficient of 'that'
#if BUILDMODE != 0
inline
#endif
float ELORankGetK(const ELORank* const that) {
#if BUILDMODE == 0
    // Check argument
    if (that == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PErrCatch(ELORankErr);
    }
#endif
    return that->_k;
}

// Get the number of entity in 'that'
#if BUILDMODE != 0
inline
#endif
int ELORankGetNb(const ELORank* const that) {
#if BUILDMODE == 0
    // Check argument
    if (that == NULL) {
        ELORankErr->_type = PErrTypeNullPointer;
        sprintf(ELORankErr->_msg, "'that' is null");
        PErrCatch(ELORankErr);
    }
#endif
    return GSetNbElem(&(that->_set));
}

```

## 4 Makefile

```

# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1

all: main

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc

```

```

include $(MAKEFILE_INC)

# Rules to make the executable
repo=elorank
$$(repo)_EXENAME): \
$$(repo)_EXENAME).o \
$$(repo)_EXE_DEP) \
$$(repo)_DEP)
$(COMPILER) 'echo "$$(repo)_EXE_DEP) $$(repo)_EXENAME).o" | tr ' ' '\n' | sort -u' $(LINK_ARG) $$(repo)_LINK_ARG)

$$(repo)_EXENAME).o: \
$$(repo)_DIR)/$$(repo)_EXENAME).c \
$$(repo)_INC_H_EXE) \
$$(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $$(repo)_BUILD_ARG 'echo "$$(repo)_INC_DIR)" | tr ' ' '\n' | sort -u' -c $$(repo)_DIR)/

```

## 5 Unit tests

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "elorank.h"
#include "pberr.h"
#include "pbmath.h"

#define RANDOMSEED 2

typedef struct Player {
    int _id;
} Player;

void UnitTestCreateFree() {
    ELORank* elo = ELORankCreate();
    if (elo == NULL || elo->k != ELORANK_K) {
        ELORankErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankCreate failed");
        PBErrCatch(ELORankErr);
    }
    ELORankFree(&elo);
    if (elo != NULL) {
        ELORankErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankFree failed");
        PBErrCatch(ELORankErr);
    }
    printf("UnitTestCreateFree OK\n");
}

void UnitTestSetGetK() {
    ELORank* elo = ELORankCreate();
    float k = 1.0;
    ELORankSetK(elo, k);
    if (ISEQUALF(elo->k, k) == false) {
        ELORankErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankSetK failed");
    }
}

```

```

    PBErCatch(ELORankErr);
}
if (ISEQUALF(ELORankGetK(elo), k) == false) {
    ELORankErr->_type = PBErTypeUnitTestFailed;
    sprintf(ELORankErr->_msg, "ELORankGetK failed");
    PBErCatch(ELORankErr);
}
ELORankFree(&elo);
printf("UnitTestSetGetK OK\n");
}

void UnitTestAddRemoveGetNb() {
    ELORank* elo = ELORankCreate();
    Player *playerA = PBErMalloc(ELORankErr, sizeof(Player));
    Player *playerB = PBErMalloc(ELORankErr, sizeof(Player));
    ELORankAdd(elo, playerA);
    if (ELORankGetNb(elo) != 1) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankAdd failed");
        PBErCatch(ELORankErr);
    }
    if (((ELOEntity*)(elo->_set._head->_data))->_data != playerA) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankAdd failed, _data invalid");
        PBErCatch(ELORankErr);
    }
    if (((ELOEntity*)(elo->_set._head->_data))->_nbRun != 0) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankAdd failed, _nbRun invalid");
        PBErCatch(ELORankErr);
    }
    if (ISEQUALF(elo->_set._head->_sortVal, ELORANK_STARTELO) == false) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankAdd failed, _sortVal invalid");
        PBErCatch(ELORankErr);
    }
    ELORankAdd(elo, playerB);
    if (ELORankGetNb(elo) != 2) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankAdd failed");
        PBErCatch(ELORankErr);
    }
    if (((ELOEntity*)(elo->_set._head->_next->_data))->_data !=
        playerB) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankAdd failed, _data invalid");
        PBErCatch(ELORankErr);
    }
    ELORankRemove(elo, playerA);
    if (ELORankGetNb(elo) != 1) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankRemove failed");
        PBErCatch(ELORankErr);
    }
    if (((ELOEntity*)(elo->_set._head->_data))->_data != playerB) {
        ELORankErr->_type = PBErTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankRemove failed, _data invalid");
        PBErCatch(ELORankErr);
    }
    ELORankFree(&elo);
    free(playerA);
    free(playerB);
}

```

```

    printf("UnitTestAddRemoveGetNb OK\n");
}

void UnitTestUpdateGetRankGetElo() {
    srandom(RANDOMSEED);
    ELORank* elo = ELORankCreate();
    Player *players[3] = {NULL};
    GSet res = GSetCreateStatic();
    Gauss gaussses[3];
    for (int i = 3; i--;) {
        players[i] = PBErrMalloc(ELORankErr, sizeof(Player));
        players[i]->_id = i;
        ELORankAdd(elo, players[i]);
        gaussses[i] = GaussCreateStatic(3 - i, 1.0);
    }
    int nbRun = 100;
    FILE* f = fopen("./elorank.txt", "w");
    for (int iRun = nbRun; iRun--;) {
        GSetFlush(&res);
        for (int i = 3; i--;)
            GSetAddSort(&res, players[i], GaussRnd(gaussses + i));
        ELORankUpdate(elo, &res);
        fprintf(f, "%d %f %f %f\n", (nbRun - iRun),
            ELORankGetELO(elo, players[0]),
            ELORankGetELO(elo, players[1]),
            ELORankGetELO(elo, players[2]));
    }
    fclose(f);
    for (int i = 3; i--;) {
        if (ELORankGetRank(elo, players[i]) != i) {
            ELORankErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ELORankErr->_msg, "ELORankUpdate failed");
            PBErrCatch(ELORankErr);
        }
    }
    const ELOEntity *winner = ELORankGetRanked(elo, 0);
    if (winner->_data != players[0]) {
        ELORankErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "ELORankGeRanked failed");
        PBErrCatch(ELORankErr);
    }
    if (winner->_nbRun != nbRun) {
        ELORankErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ELORankErr->_msg, "nbRun invalid");
        PBErrCatch(ELORankErr);
    }
    ELORankFree(&elo);
    GSetFlush(&res);
    for (int i = 3; i--;)
        free(players[i]);
    printf("UnitTestUpdateGetRankGetElo OK\n");
}

void UnitTestAll() {
    UnitTestCreateFree();
    UnitTestSetGetK();
    UnitTestAddRemoveGetNb();
    UnitTestUpdateGetRankGetElo();
    printf("UnitTestAll OK\n");
}

int main() {

```

```

    UnitTestAll();
    // Return success code
    return 0;
}

```

## 6 Unit tests output

```

UnitTestCreateFree OK
UnitTestSetGetK OK
UnitTestAddRemoveGetNb OK
UnitTestUpdateGetRankGetElo OK
UnitTestAll OK

```

## 7 ELORank.txt

```

1 8.000000 -8.000000 0.000000
2 15.723836 -15.723836 -0.000000
3 15.181863 -7.181863 -8.000000
4 22.658251 -6.934165 -15.724085
5 29.878273 -14.695467 -15.182804
6 36.852001 -14.190701 -22.661297
7 43.589993 -21.704742 -21.885246
8 50.101974 -20.963814 -29.138155
9 48.398232 -12.250955 -36.147270
10 46.752392 -3.834395 -42.917988
11 53.164032 -3.704153 -49.459869
12 59.366798 -11.578976 -47.787811
13 65.365189 -11.188716 -54.176464
14 71.173401 -10.813753 -60.359642
15 76.800728 -10.453563 -66.347153
16 82.256058 -10.107615 -72.148430
17 79.547867 -1.775375 -77.772476
18 84.931526 -9.717095 -75.214417
19 90.144630 -9.398850 -80.745766
20 95.206184 -17.093233 -78.112938
21 100.111649 -24.538937 -75.572701
22 104.868889 -23.745762 -81.123116
23 109.494507 -22.983797 -86.510704
24 105.994957 -30.251780 -75.743172
25 110.581802 -29.279697 -81.302094
26 115.044350 -36.345665 -78.698669
27 119.379044 -35.190456 -84.188576
28 123.600494 -34.080109 -89.520370
29 119.714195 -25.012819 -94.701355
30 123.941559 -32.228443 -91.713097
31 128.049164 -31.220741 -96.828400
32 132.054733 -30.252058 -101.802650
33 127.963089 -21.320766 -106.642296
34 131.989212 -20.663591 -111.325592
35 135.918457 -20.031824 -115.886597
36 123.755226 -19.424370 -104.330818
37 127.890121 -18.821854 -109.068230
38 131.923431 -26.242733 -105.680656
39 127.842506 -17.436901 -110.405563
40 123.880096 -8.900932 -114.979118
41 128.036240 -8.627240 -119.408958

```

42 132.091034 -16.364151 -115.726845  
 43 136.027908 -15.865696 -120.162178  
 44 131.872452 -7.386406 -124.486008  
 45 135.838455 -7.163136 -128.675278  
 46 139.711716 -14.948422 -124.763252  
 47 135.472931 -6.500619 -128.972260  
 48 139.356705 -6.305744 -133.050919  
 49 135.151398 1.881705 -137.033066  
 50 131.069504 -6.174252 -124.895218  
 51 127.058136 2.012465 -129.070572  
 52 131.167801 1.951608 -133.119385  
 53 135.179230 1.893085 -137.072296  
 54 131.096741 9.836794 -140.933502  
 55 135.137115 9.544323 -144.681412  
 56 139.082886 9.262930 -148.345795  
 57 142.938095 8.992129 -151.930206  
 58 138.706543 16.731459 -155.437988  
 59 142.600052 8.246344 -150.846375  
 60 130.373032 16.006891 -146.379913  
 61 134.453491 7.533657 -141.987137  
 62 138.407562 -0.689304 -137.718246  
 63 142.242188 -8.668893 -133.573273  
 64 145.964005 -8.412176 -137.551804  
 65 149.603500 -8.165155 -141.438309  
 66 153.164139 -7.927407 -145.236694  
 67 156.649200 -7.698526 -148.950638  
 68 160.061768 -7.478126 -152.583603  
 69 163.404770 -7.265838 -156.138885  
 70 166.680939 -7.061309 -159.619583  
 71 169.892883 -6.864205 -163.028641  
 72 173.043076 -6.674206 -166.368820  
 73 176.133820 -6.491009 -169.642776  
 74 179.167343 -6.314322 -172.852982  
 75 182.145721 -6.143868 -176.001816  
 76 185.070938 -5.979387 -179.091507  
 77 187.944855 -5.820625 -182.124191  
 78 182.769272 -13.667344 -169.101898  
 79 177.653793 -5.298442 -172.355316  
 80 180.664001 -5.155023 -175.508942  
 81 183.619919 -5.016639 -178.603241  
 82 186.523483 -4.883080 -181.640366  
 83 189.376526 -4.754143 -184.622345  
 84 192.180801 -4.629635 -187.551117  
 85 194.937943 -4.509374 -190.428528  
 86 189.649536 3.606816 -193.256317  
 87 184.486328 11.513550 -195.999832  
 88 187.449646 11.214846 -198.664444  
 89 190.360336 10.926239 -201.286530  
 90 193.220200 10.647317 -203.867477  
 91 196.030945 2.377687 -198.408585  
 92 198.755539 2.317230 -201.072723  
 93 193.435593 10.258780 -203.694321  
 94 196.240829 9.999002 -206.239792  
 95 190.998749 17.747816 -208.746536  
 96 193.882751 17.300257 -211.182983  
 97 196.716660 16.867434 -213.584076  
 98 199.502136 8.448761 -207.950882  
 99 202.199768 8.238097 -210.437851  
 100 204.853592 8.034303 -212.887878

