

# ELORank

P. Baillehache

October 6, 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 100.0

// ===== 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=../PMake/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

```

1 108.000000 92.000000 100.000000
2 115.723839 84.276161 100.000000
3 115.181870 92.818130 92.000000
4 122.658257 93.065826 84.275917
5 129.878281 85.304527 84.817200
6 136.852005 85.809296 77.338707
7 143.589996 78.295250 78.114754
8 150.101974 79.036179 70.861847
9 148.398239 87.749039 63.852730
10 146.752396 96.165596 57.082012
11 153.164032 96.295837 50.540131
12 159.366806 88.421013 52.212189
13 165.365204 88.811272 45.823536
14 171.173416 89.186234 39.640358
15 176.800735 89.546425 33.652843
16 182.256058 89.892372 27.851570
17 179.547867 98.224609 22.227524
18 184.931519 90.282890 24.785585
19 190.144623 90.601135 19.254234
20 195.206177 82.906754 21.887064
21 200.111649 75.461052 24.427301
22 204.868896 76.254227 18.876888
23 209.494522 77.016190 13.489302
24 205.994965 69.748207 24.256834
25 210.581802 70.720291 18.697908
26 215.044342 63.654324 21.301329
27 219.379044 64.809532 15.811423
28 223.600494 65.919876 10.479629
29 219.714188 74.987167 5.298641
30 223.941559 67.771545 8.286898
31 228.049164 68.779243 3.171598
32 232.054733 69.747925 -1.802656
33 227.963089 78.679214 -6.642300
34 231.989212 79.336388 -11.325594
35 235.918457 79.968155 -15.886599
36 223.755234 80.575607 -4.330823
37 227.890121 81.178123 -9.068232
38 231.923431 73.757248 -5.680658
39 227.842499 82.563080 -10.405563
40 223.880081 91.099045 -14.979122
41 228.036224 91.372734 -19.408962

```

42 232.091019 83.635826 -15.726847  
 43 236.027893 84.134285 -20.162182  
 44 231.872437 92.613571 -24.486013  
 45 235.838440 92.836845 -28.675285  
 46 239.711700 85.051559 -24.763264  
 47 235.472916 93.499359 -28.972279  
 48 239.356689 93.694237 -33.050930  
 49 235.151382 101.881683 -37.033077  
 50 231.069489 93.825729 -24.895229  
 51 227.058121 102.012451 -29.070580  
 52 231.167786 101.951591 -33.119385  
 53 235.179214 101.893066 -37.072296  
 54 231.096710 109.836777 -40.933510  
 55 235.137085 109.544304 -44.681412  
 56 239.082855 109.262909 -48.345795  
 57 242.938065 108.992111 -51.930202  
 58 238.706512 116.731438 -55.437984  
 59 242.600021 108.246323 -50.846375  
 60 230.373001 116.006874 -46.379906  
 61 234.453461 107.533638 -41.987129  
 62 238.407532 99.310677 -37.718243  
 63 242.242157 91.331085 -33.573277  
 64 245.963974 91.587799 -37.551807  
 65 249.603470 91.834824 -41.438316  
 66 253.164108 92.072571 -45.236702  
 67 256.649170 92.301453 -48.950645  
 68 260.061737 92.521851 -52.583618  
 69 263.404724 92.734138 -56.138901  
 70 266.680908 92.938667 -59.619602  
 71 269.892853 93.135773 -63.028656  
 72 273.043030 93.325768 -66.368843  
 73 276.133789 93.508965 -69.642792  
 74 279.167297 93.685654 -72.852997  
 75 282.145660 93.856110 -76.001823  
 76 285.070862 94.020592 -79.091515  
 77 287.944794 94.179352 -82.124199  
 78 282.769226 86.332634 -69.101898  
 79 277.653748 94.701538 -72.355324  
 80 280.663940 94.844955 -75.508949  
 81 283.619873 94.983337 -78.603256  
 82 286.523438 95.116898 -81.640381  
 83 289.376495 95.245834 -84.622368  
 84 292.180756 95.370338 -87.551147  
 85 294.937897 95.490601 -90.428551  
 86 289.649475 103.606789 -93.256332  
 87 284.486267 111.513519 -95.999855  
 88 287.449585 111.214813 -98.664467  
 89 290.360291 110.926208 -101.286552  
 90 293.220154 110.647285 -103.867493  
 91 296.030884 102.377655 -98.408600  
 92 298.755493 102.317200 -101.072739  
 93 293.435547 110.258751 -103.694336  
 94 296.240784 109.998970 -106.239799  
 95 290.998718 117.747780 -108.746536  
 96 293.882721 117.300217 -111.182983  
 97 296.716644 116.867393 -113.584076  
 98 299.502136 108.448723 -107.950882  
 99 302.199768 108.238060 -110.437851  
 100 304.853577 108.034264 -112.887878

