GSet

P. Baillehache

February 4, 2018

Contents

Introduction

GSet library is a C library to manipulate sets of data.

Elements of the GSet are void pointers toward any kind of data. These data must be allocated and freed separately. The GSet only provides a mean to manipulate sets of pointers toward these data.

The GSet offers functions to add elements (at first position, last position, given position, or sorting based on a float value), to access elements (at first position, last position, given position), to get index of first/last element pointing to a given data, to remove elements (at first position, last position, given position, or first/last/all pointing toward a given data), to search for data in elements (first one or last one), to print the set on a stream, to split, merge and sort the set.

The library also provides two iterator structures to run through a GSet forward or backward, and apply a user defined function on each element.

It uses the `PBErr` library.

1 Interface

```
// *************** GSET.H ***************
#ifndef GSET_H
#define GSET_H

// ================= Include ==================
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include "pberr.h"

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

// ================= Generic functions ==================

#define GSetIterFree(IterRef) _Generic(IterRef, \
  GSetIterForward**: GSetIterForwardFree, \
  GSetIterBackward**: GSetIterBackwardFree, \
  default: PBErrInvalidPolymorphism)(IterRef)

#define GSetIterClone(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardClone, \
  GSetIterBackward*: GSetIterBackwardClone, \
  default: PBErrInvalidPolymorphism)(Iter)

#define GSetIterReset(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardReset, \
  GSetIterBackward*: GSetIterBackwardReset, \
  default: PBErrInvalidPolymorphism)(Iter)

#define GSetIterStep(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardStep, \
  GSetIterBackward*: GSetIterBackwardStep, \
  default: PBErrInvalidPolymorphism)(Iter)

#define GSetIterApply(Iter, Fun, Param) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardApply, \
  GSetIterBackward*: GSetIterBackwardApply, \
  default: PBErrInvalidPolymorphism)(Iter, Fun, Param)

#define GSetIterIsFirst(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardIsFirst, \
  GSetIterBackward*: GSetIterBackwardIsFirst, \
  default: PBErrInvalidPolymorphism)(Iter)

#define GSetIterIsLast(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardIsLast, \
  GSetIterBackward*: GSetIterBackwardIsLast, \
  default: PBErrInvalidPolymorphism)(Iter)
```

```
#define GSetIterSetGSet(Iter, Set) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardSetGSet, \
  GSetIterBackward*: GSetIterBackwardSetGSet, \
  default: PBErrInvalidPolymorphism)(Iter, Set)

#define GSetIterGet(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardGet, \
  GSetIterBackward*: GSetIterBackwardGet, \
  default: PBErrInvalidPolymorphism)(Iter)

#define GSetIterGetElem(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardGetElem, \
  GSetIterBackward*: GSetIterBackwardGetElem, \
  default: PBErrInvalidPolymorphism)(Iter)

#define GSetIterRemoveElem(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardRemoveElem, \
  GSetIterBackward*: GSetIterBackwardRemoveElem, \
  default: PBErrInvalidPolymorphism)(Iter)

// ================= Data structures ===================

// Structure of one element of the GSet
struct GSetElem;
typedef struct GSetElem {
  // Pointer toward the data
  void* _data;
  // Pointer toward the next element in the GSet
  struct GSetElem* _next;
  // Pointer toward the previous element in the GSet
  struct GSetElem* _prev;
  // Value to sort element in the GSet, 0.0 by default
  float _sortVal;
} GSetElem;

// Structure of the GSet
typedef struct GSet {
  // Pointer toward the element at the head of the GSet
  GSetElem* _head;
  // Pointer toward the last element of the GSet
  GSetElem* _tail;
  // Number of element in the GSet
  int _nbElem;
} GSet;

// Structures of the GSet iterators
typedef struct GSetIterForward {
  // GSet attached to the iterator
  GSet* _set;
  // Current element
  GSetElem* _curElem;
} GSetIterForward;

typedef struct GSetIterBackward {
  // GSet attached to the iterator
  GSet* _set;
  // Current element
  GSetElem* _curElem;
} GSetIterBackward;

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

```
// Function to create a new GSet,
// Return a pointer toward the new GSet
GSet* GSetCreate(void);

// Static constructors for GSet
#if BUILDMODE != 0
inline
#endif
GSet GSetCreateStatic(void);

// Function to clone a GSet,
// Return a pointer toward the new GSet
GSet* GSetClone(GSet* that);

// Function to free the memory used by the GSet
void GSetFree(GSet** s);

// Function to empty the GSet
#if BUILDMODE != 0
inline
#endif
void GSetFlush(GSet* that);

// Return the number of element in the set
#if BUILDMODE != 0
inline
#endif
int GSetNbElem(GSet* that);

// Function to print a GSet
// Use the function 'printData' to print the data pointed to by
// the elements, and print 'sep' between each element
// If printData is null, print the pointer value instead
void GSetPrint(GSet* that, FILE* stream,
  void(*printData)(void* data, FILE* stream), char* sep);

// Function to insert an element pointing toward 'data' at the
// head of the GSet
#if BUILDMODE != 0
inline
#endif
void GSetPush(GSet* that, void* data);

// Function to insert an element pointing toward 'data' at the
// position defined by 'v' sorting the set in increasing order
void GSetAddSort(GSet* that, void* data, double v);

// Function to insert an element pointing toward 'data' at the
// 'iElem'-th position
// If 'iElem' is greater than or equal to the number of element
// in the GSet, elements pointing toward null data are added
void GSetInsert(GSet* that, void* data, int iElem);

// Function to insert an element pointing toward 'data' at the
// tail of the GSet
#if BUILDMODE != 0
inline
#endif
void GSetAppend(GSet* that, void* data);

// Function to remove the element at the head of the GSet
```

```
// Return the data pointed to by the removed element, or null if the
// GSet is empty
#if BUILDMODE != 0
inline
#endif
void* GSetPop(GSet* that);

// Function to remove the element at the tail of the GSet
// Return the data pointed to by the removed element, or null if the
// GSet is empty
#if BUILDMODE != 0
inline
#endif
void* GSetDrop(GSet* that);

// Function to remove the element at the 'iElem'-th position of the GSet
// Return the data pointed to by the removed element
#if BUILDMODE != 0
inline
#endif
void* GSetRemove(GSet* that, int iElem);

// Function to remove the element 'elem' of the GSet
// Return the data pointed to by the removed element
// The GSetElem is freed and *elem == NULL after calling this function
#if BUILDMODE != 0
inline
#endif
void* GSetRemoveElem(GSet* that, GSetElem** elem);

// Function to remove the first element of the GSet pointing to 'data'
// If there is no element pointing to 'data' do nothing
#if BUILDMODE != 0
inline
#endif
void GSetRemoveFirst(GSet* that, void* data);

// Function to remove the last element of the GSet pointing to 'data'
// If there is no element pointing to 'data' do nothing
#if BUILDMODE != 0
inline
#endif
void GSetRemoveLast(GSet* that, void* data);

// Function to remove all the selement of the GSet pointing to 'data'
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void GSetRemoveAll(GSet* that, void* data);

// Function to get the data at the 'iElem'-th position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* GSetGet(GSet* that, int iElem);

// Function to get the element at the 'iElem'-th position of the GSet
// without removing it
#if BUILDMODE != 0
inline
```

```
#endif
GSetElem* GSetGetElem(GSet* that, int iElem);

// Function to get the index of the first element of the GSet
// which point to 'data'
// Return -1 if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
int GSetGetIndexFirst(GSet* that, void* data);

// Function to get the index of the last element of the GSet
// which point to 'data'
// Return -1 if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
int GSetGetIndexLast(GSet* that, void* data);

// Function to get the first element of the GSet
// which point to 'data'
// Return NULL if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetGetFirstElem(GSet* that, void* data);

// Function to get the last element of the GSet
// which point to 'data'
// Return NULL if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetGetLastElem(GSet* that, void* data);

// Function to sort the element of the gset in increasing order of
// _sortVal
void GSetSort(GSet* that);

// Merge the GSet 'r' at the end of the GSet 's'
// 'r' and 's' can be empty
// After calling this function r is empty
#if BUILDMODE != 0
inline
#endif
void GSetMerge(GSet* s, GSet* r);

// Split the GSet at the GSetElem 'e'
// 'e' must be and element of the set
// the set new end is the element before 'e', the set becomes empty if
// 'e' was the first element
// Return a new GSet starting with 'e', or NULL if 'e' is not
// an element of the set
#if BUILDMODE != 0
inline
#endif
GSet* GSetSplit(GSet* that, GSetElem* e);

// Switch the 'iElem'-th and 'jElem'-th element of the set
#if BUILDMODE != 0
inline
#endif
```

```
void GSetSwitch(GSet* that, int iElem, int jElem);

// Set the sort value of the GSetElem 'that' to 'v'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetSortVal(GSetElem* that, float v);

// Create a new GSetIterForward for the GSet 'set'
// The iterator is reset upon creation
GSetIterForward* GSetIterForwardCreate(GSet* set);
#if BUILDMODE != 0
inline
#endif
GSetIterForward GSetIterForwardCreateStatic(GSet* set);

// Create a new GSetIterBackward for the GSet 'set'
// The iterator is reset upon creation
GSetIterBackward* GSetIterBackwardCreate(GSet* set);
#if BUILDMODE != 0
inline
#endif
GSetIterBackward GSetIterBackwardCreateStatic(GSet* set);

// Free the memory used by a GSetIterForward (not by its attached GSet)
// Do nothing if arguments are invalid
void GSetIterForwardFree(GSetIterForward** that);

// Free the memory used by a GSetIterBackward (not by its attached GSet)
// Do nothing if arguments are invalid
void GSetIterBackwardFree(GSetIterBackward** that);

// Clone a GSetIterForward
GSetIterForward* GSetIterForwardClone(GSetIterForward* that);

// Clone a GSetIterBackward
GSetIterBackward* GSetIterBackwardClone(GSetIterBackward* that);

// Reset the GSetIterForward to its starting position
#if BUILDMODE != 0
inline
#endif
void GSetIterForwardReset(GSetIterForward* that);

// Reset the GSetIterBackward to its starting position
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardReset(GSetIterBackward* that);

// Step the GSetIterForward
// Return false if we couldn't step
// Return true else
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardStep(GSetIterForward* that);

// Step the GSetIterBackward
// Return false if we couldn't step
// Return true else
#if BUILDMODE != 0
```

```
inline
#endif
bool GSetIterBackwardStep(GSetIterBackward* that);

// Apply a function to all elements of the GSet of the GSetIterForward
// The iterator is first reset, then the function is apply sequencially
// using the Step function of the iterator
// The applied function takes to void* arguments: 'data' is the _data
// property of the nodes, 'param' is a hook to allow the user to pass
// parameters to the function through a user-defined structure
#if BUILDMODE != 0
inline
#endif
void GSetIterForwardApply(GSetIterForward* that,
  void(*fun)(void* data, void* param), void* param);

// Apply a function to all elements of the GSet of the GSetIterBackward
// The iterator is first reset, then the function is apply sequencially
// using the Step function of the iterator
// The applied function takes to void* arguments: 'data' is the _data
// property of the nodes, 'param' is a hook to allow the user to pass
// parameters to the function through a user-defined structure
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardApply(GSetIterBackward* that,
  void(*fun)(void* data, void* param), void* param);

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardIsFirst(GSetIterForward* that);

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardIsFirst(GSetIterBackward* that);

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardIsLast(GSetIterForward* that);

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardIsLast(GSetIterBackward* that);

// Change the attached set of the iterator, and reset it
#if BUILDMODE != 0
```

```
inline
#endif
void GSetIterForwardSetGSet(GSetIterForward* that, GSet* set);

// Change the attached set of the iterator, and reset it
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardSetGSet(GSetIterBackward* that, GSet* set);

// Return the data currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void* GSetIterForwardGet(GSetIterForward* that);

// Return the data currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void* GSetIterBackwardGet(GSetIterBackward* that);

// Return the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetIterForwardGetElem(GSetIterForward* that);

// Return the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetIterBackwardGetElem(GSetIterBackward* that);

// Remove the element currently pointed to by the iterator
// The iterator is moved forward to the next element
// Return false if we couldn't move
// Return true else
// It's the responsibility of the user to delete the content of the
// element prior to calling this function
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardRemoveElem(GSetIterForward* that);

// Remove the element currently pointed to by the iterator
// The iterator is moved backward to the next element
// Return false if we couldn't move
// Return true else
// It's the responsibility of the user to delete the content of the
// element prior to calling this function
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardRemoveElem(GSetIterBackward* that);

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

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

```
#endif
```

2 Code

2.1 gset.c

```
// *************** GSET.C ***************

// ================= Include ==================
#include "gset.h"
#if BUILDMODE == 0
#include "gset-inline.c"
#endif

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

// Function to create a new GSet,
// Return a pointer toward the new GSet
GSet* GSetCreate(void) {
  // Allocate memory for the GSet
  GSet* s = PBErrMalloc(GSetErr, sizeof(GSet));
  // Set the pointer to head and tail, and the number of element
  s->_head = NULL;
  s->_tail = NULL;
  s->_nbElem = 0;
  // Return the new GSet
  return s;
}

// Function to clone a GSet,
// Return a pointer toward the new GSet
GSet* GSetClone(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create the clone
  GSet* c = GSetCreate();
  // Set a pointer to the head of the set
  GSetElem* ptr = that->_head;
  // While the pointer is not at the end of the set
  while (ptr != NULL) {
    // Append the data of the current pointer to the clone
    GSetAppend(c, ptr->_data);
    // Copy the sort value
    c->_tail->_sortVal = ptr->_sortVal;
    // Move the pointer to the next element
    ptr = ptr->_next;
  }
  // Return the clone
  return c;
}

// Function to free the memory used by the GSet
void GSetFree(GSet** that) {
```

```
  if (that == NULL || *that == NULL) return;
  // Empty the GSet
  GSetFlush(*that);
  // Free the memory
  free(*that);
  // Set the pointer to null
  *that = NULL;
}

// Function to print a GSet
// Use the function 'printData' to print the data pointed to by
// the elements, and print 'sep' between each element
// If printData is null, print the pointer value instead
// Do nothing if arguments are invalid
void GSetPrint(GSet* that, FILE* stream,
  void(*printData)(void* data, FILE* stream), char* sep) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (stream == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'stream' is null");
    PBErrCatch(GSetErr);
  }
  if (sep == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'sep' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set a pointer to the head element
  GSetElem* p = that->_head;
  // While the pointer hasn't reach the end
  while (p != NULL) {
    // If there is a print function for the data
    if (printData != NULL) {
      // Use the argument function to print the data of the
      // current element
      (*printData)(p->_data, stream);
    // Else, there is no print function for the data
    } else {
      // Print the pointer value instead
      fprintf(stream, "%p", p->_data);
    }
    // Move to the next element
    p = p->_next;
    // If there is a next element
    if (p != NULL)
      // Print the separator
      fprintf(stream, "%s", sep);
  }
  // Flush the stream
  fflush(stream);
}

// Function to insert an element pointing toward 'data' at the
// position defined by 'v' sorting the set in increasing order
void GSetAddSort(GSet* that, void* data, double v) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Allocate memory for the new element
  GSetElem* e = PBErrMalloc(GSetErr, sizeof(GSetElem));
  // Memorize the pointer toward data
  e->_data = data;
  // Memorize the sorting value
  e->_sortVal = v;
  // If the GSet is empty
  if (that->_nbElem == 0) {
    // Add the element at the head of the GSet
    that->_head = e;
    that->_tail = e;
    e->_next = NULL;
    e->_prev = NULL;
  } else {
    // Set a pointer to the head of the GSet
    GSetElem* p = that->_head;
    // While the pointed element has a lower value than the
    // new element, move the pointer to the next element
    while (p != NULL && p->_sortVal <= v)
      p = p->_next;
    // Set the next element of the new element to the current element
    e->_next = p;
    // If the current element is not null
    if (p != NULL) {
      // Insert the new element inside the list of elements before p
      e->_prev = p->_prev;
      if (p->_prev != NULL)
        p->_prev->_next = e;
      else
        that->_head = e;
      p->_prev = e;
    // Else, if the current element is null
    } else {
      // Insert the new element at the tail of the GSet
      e->_prev = that->_tail;
      if (that->_tail != NULL)
        that->_tail->_next = e;
      that->_tail = e;
      if (that->_head == NULL)
        that->_head = e;
    }
  }
  // Increment the number of elements
  ++(that->_nbElem);
}

// Function to insert an element pointing toward 'data' at the
// 'iElem'-th position
// If 'iElem' is greater than or equal to the number of element
// in the GSet, elements pointing toward null data are added
void GSetInsert(GSet* that, void* data, int iElem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
```

```
  }
  if (iElem < 0) {
    GSetErr->_type = PBErrTypeInvalidArg;
    sprintf(GSetErr->_msg, "'iElem' is invalid (%d>=0)", iElem);
    PBErrCatch(GSetErr);
  }
#endif
  // If iElem is greater than the number of elements, append
  // elements pointing toward null data to fill in the gap
  while (iElem > that->_nbElem)
    GSetAppend(that, NULL);
  // If iElem is in the list of element or at the tail
  if (iElem <= that->_nbElem + 1) {
    // If the insert position is the head
    if (iElem == 0) {
      // Push the data
      GSetPush(that, data);
    // Else, if the insert position is the tail
    } else if (iElem == that->_nbElem) {
      // Append data
      GSetAppend(that, data);
    // Else, the insert position is inside the list
    } else {
      // Allocate memory for the new element
      GSetElem* e = PBErrMalloc(GSetErr, sizeof(GSetElem));
      // Memorize the pointer toward data
      e->_data = data;
      // By default set the sorting value to 0.0
      e->_sortVal = 0.0;
      // Set a pointer toward the head of the GSet
      GSetElem* p = that->_head;
      // Move the pointer to the iElem-th element
      for (int i = iElem; i > 0 && p != NULL; --i, p = p->_next);
      // Insert the element before the pointer
      e->_next = p;
      e->_prev = p->_prev;
      p->_prev = e;
      e->_prev->_next = e;
      // Increment the number of elements
      ++(that->_nbElem);
    }
  }
}

// Function to sort the element of the gset in increasing order of
// _sortVal
// Do nothing if arguments are invalid or the sort failed
static GSet* GSetSortRec(GSet** s);
void GSetSort(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create a clone of the original set
  GSet* clone = GSetClone(that);
  // Create recursively the sorted set
  GSet* res = GSetSortRec(&clone);
  // If we could sort the set
  if (res != NULL) {
```

```
    // Update the original set with the result one
    GSetFlush(that);
    memcpy(that, res, sizeof(GSet));
    // Free the memory used by the result set
    free(res);
    res = NULL;
  }
}
GSet* GSetSortRec(GSet** s) {
  // Declare a variable for the result
  GSet* res = NULL;
  // If the set contains no element or one element
  if ((*s)->_nbElem == 0 || (*s)->_nbElem == 1) {
    // Return the set
    res = *s;
  // Else, the set contains several elements
  } else {
    // Create two sets, one for elements lower than the pivot
    // one for elements greater or equal than the pivot
    GSet* lower = GSetCreate();
    GSet* greater = GSetCreate();
    res = GSetCreate();
    // Declare a variable to memorize the pivot, which is equal
    // to the sort value of the first element of the set
    float pivot = (*s)->_head->_sortVal;
    // Pop the pivot and put it in the result
    void* data = GSetPop(*s);
    GSetAppend(res, data);
    res->_head->_sortVal = pivot;
    // Pop all the elements one by one from the set
    while ((*s)->_nbElem != 0) {
      // Declare a variable to memorize the sort value of the head
      // element
      float val = (*s)->_head->_sortVal;
      // Pop the head element
      data = GSetPop(*s);
      // If the poped element has a sort value lower than the pivot
      if (val < pivot) {
        // Insert it in the lower set
        GSetAppend(lower, data);
        // Copy the sort value
        lower->_tail->_sortVal = val;
      // Else, the poped element has a sort value greater than or
      // equal to the pivot
      } else {
        // Insert it in the greater set
        GSetAppend(greater, data);
        // Copy the sort value
        greater->_tail->_sortVal = val;
      }
    }
    // At the end of the loop the original set is empty and we
    // don't need it anymore
    GSetFree(s);
    // Sort the two half
    GSet* sortedLower = GSetSortRec(&lower);
    GSet* sortedGreater = GSetSortRec(&greater);
    // Merge back the sorted two halves and the pivot
    GSetMerge(sortedLower, res);
    GSetMerge(sortedLower, sortedGreater);
    GSetFree(&res);
    res = sortedLower;
```

```
    GSetFree(&sortedGreater);
  }
  // Return the result
  return res;
}

// Create a new GSetIterForward for the GSet 'set'
// The iterator is reset upon creation
GSetIterForward* GSetIterForwardCreate(GSet* set) {
#if BUILDMODE == 0
  if (set == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'set' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Allocate memory
  GSetIterForward* ret =
    PBErrMalloc(GSetErr, sizeof(GSetIterForward));
  // Set properties
  ret->_set = set;
  ret->_curElem = set->_head;
  // Return the new iterator
  return ret;
}

// Create a new GSetIterBackward for the GSet 'set'
// The iterator is reset upon creation
GSetIterBackward* GSetIterBackwardCreate(GSet* set) {
#if BUILDMODE == 0
  if (set == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'set' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Allocate memory
  GSetIterBackward* ret =
    PBErrMalloc(GSetErr, sizeof(GSetIterBackward));
  // Set properties
  ret->_set = set;
  ret->_curElem = set->_tail;
  // Return the new iterator
  return ret;
}

// Free the memory used by a GSetIterForward (not by its attached GSet)
// Do nothing if arguments are invalid
void GSetIterForwardFree(GSetIterForward** that) {
  // Check arguments
  if (that == NULL || *that == NULL)
    return;
  (*that)->_set = NULL;
  (*that)->_curElem = NULL;
  free(*that);
  *that = NULL;
}

// Free the memory used by a GSetIterBackward (not by its attached GSet)
// Do nothing if arguments are invalid
void GSetIterBackwardFree(GSetIterBackward** that) {
  // Check arguments
```

```
  if (that == NULL || *that == NULL)
    return;
  (*that)->_set = NULL;
  (*that)->_curElem = NULL;
  free(*that);
  *that = NULL;
}

// Clone a GSetIterForward
GSetIterForward* GSetIterForwardClone(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create the clone
  GSetIterForward* ret = GSetIterForwardCreate(that->_set);
  ret->_curElem = that->_curElem;
  // return the clone
  return ret;
}

// Clone a GSetIterBackward
GSetIterBackward* GSetIterBackwardClone(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create the clone
  GSetIterBackward* ret = GSetIterBackwardCreate(that->_set);
  ret->_curElem = that->_curElem;
  // return the clone
  return ret;
}
```

2.2 gset-inline.c

```
 // ============ GSET-INLINE.C ================

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

// Static constructors for GSet
#if BUILDMODE != 0
inline
#endif
GSet GSetCreateStatic(void) {
  // Declare a GSet and set the properties
  GSet s = {._head = NULL, ._tail = NULL, ._nbElem = 0};
  // Return the GSet
  return s;
}

// Function to empty the GSet
#if BUILDMODE != 0
```

```
inline
#endif
void GSetFlush(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Pop element until the GSet is null
  while (GSetPop(that) || that->_nbElem > 0);
}

// Function to insert an element pointing toward 'data' at the
// head of the GSet
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void GSetPush(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Allocate memory for the new element
  GSetElem* e = PBErrMalloc(GSetErr, sizeof(GSetElem));
  // Memorize the pointer toward data
  e->_data = data;
  // By default set the sorting value to 0.0
  e->_sortVal = 0.0;
  // Add the element at the head of the GSet
  e->_prev = NULL;
  if (that->_head != NULL)
    that->_head->_prev = e;
  e->_next = that->_head;
  that->_head = e;
  if (that->_tail == NULL)
    that->_tail = e;
  // Increment the number of elements in the GSet
  ++(that->_nbElem);
}

// Function to insert an element pointing toward 'data' at the
// tail of the GSet
#if BUILDMODE != 0
inline
#endif
void GSetAppend(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  GSetElem* e = PBErrMalloc(GSetErr, sizeof(GSetElem));
  if (e != NULL) {
    e->_data = data;
```

```
    e->_sortVal = 0.0;
    e->_prev = that->_tail;
    e->_next = NULL;
    if (that->_tail != NULL)
      that->_tail->_next = e;
    that->_tail = e;
    if (that->_head == NULL)
      that->_head = e;
    ++(that->_nbElem);
  }
}

// Function to remove the element at the head of the GSet
// Return the data pointed to by the removed element, or null if the
// GSet is empty
#if BUILDMODE != 0
inline
#endif
void* GSetPop(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  void* ret = NULL;
  GSetElem* p = that->_head;
  if (p != NULL) {
    ret = p->_data;
    that->_head = p->_next;
    if (p->_next != NULL)
      p->_next->_prev = NULL;
    p->_next = NULL;
    p->_data = NULL;
    if (that->_tail == p)
      that->_tail = NULL;
    free(p);
    --(that->_nbElem);
  }
  return ret;
}

// Function to remove the element at the tail of the GSet
// Return the data pointed to by the removed element, or null if the
// GSet is empty
#if BUILDMODE != 0
inline
#endif
void* GSetDrop(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  void* ret = NULL;
  GSetElem* p = that->_tail;
  if (p != NULL) {
    ret = p->_data;
    that->_tail = p->_prev;
```

```
    if (p->_prev != NULL)
      p->_prev->_next = NULL;
    p->_prev = NULL;
    p->_data = NULL;
    if (that->_head == p)
      that->_head = NULL;
    free(p);
    --(that->_nbElem);
  }
  return ret;
}

// Function to remove the element 'elem' of the GSet
// Return the data pointed to by the removed element
// The GSetElem is freed and *elem == NULL after calling this function
#if BUILDMODE != 0
inline
#endif
void* GSetRemoveElem(GSet* that, GSetElem** elem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (elem == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'elem' is null");
    PBErrCatch(GSetErr);
  }
  if (*elem == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'*elem' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Variable to memorize the return value
  void* ret = NULL;
  // Memorize the data at iElem-th position
  ret = (*elem)->_data;
  // Remove the element
  if ((*elem)->_next != NULL)
    (*elem)->_next->_prev = (*elem)->_prev;
  if ((*elem)->_prev != NULL)
    (*elem)->_prev->_next = (*elem)->_next;
  if (that->_head == (*elem))
    that->_head = (*elem)->_next;
  if (that->_tail == (*elem))
    that->_tail = (*elem)->_prev;
  (*elem)->_next = NULL;
  (*elem)->_prev = NULL;
  (*elem)->_data = NULL;
  free((*elem));
  *elem = NULL;
  // Decrement the number of elements
  --(that->_nbElem);
  // Return the data
  return ret;
}

// Function to remove the first element of the GSet pointing to 'data'
// If there is no element pointing to 'data' do nothing
```

```
#if BUILDMODE != 0
inline
#endif
void GSetRemoveFirst(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Get the first element pointing to 'data'
  GSetElem* elem = GSetGetFirstElem(that, data);
  // If we could find an element
  if (elem != NULL)
    // Remove this element
    while (GSetRemoveElem(that, &elem) && false);
}

// Function to remove the last element of the GSet pointing to 'data'
// If there is no element pointing to 'data' do nothing
#if BUILDMODE != 0
inline
#endif
void GSetRemoveLast(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Get the last element pointing to 'data'
  GSetElem* elem = GSetGetLastElem(that, data);
  // If we could find an element
  if (elem != NULL)
    // Remove this element
    while (GSetRemoveElem(that, &elem) && false);
}

// Function to remove the element at the 'iElem'-th position of the GSet
// Return the data pointed to by the removed element
#if BUILDMODE != 0
inline
#endif
void* GSetRemove(GSet* that, int iElem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (iElem < 0 || iElem >= that->_nbElem) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'iElem' is invalid (0<=%d<%d)",
      iElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
#endif
  // Variable to memorize the return value
  void* ret = NULL;
  // Set a pointer to the head of the Gset
```

```
  GSetElem* p = that->_head;
  // Move the pointer to the iElem-th element
  for (int i = iElem; i > 0 && p != NULL; --i, p = p->_next);
  // Memorize the data at iElem-th position
  ret = p->_data;
  // Remove the element
  if (p->_next != NULL)
    p->_next->_prev = p->_prev;
  if (p->_prev != NULL)
    p->_prev->_next = p->_next;
  if (that->_head == p)
    that->_head = p->_next;
  if (that->_tail == p)
    that->_tail = p->_prev;
  p->_next = NULL;
  p->_prev = NULL;
  p->_data = NULL;
  free(p);
  // Decrement the number of elements
  --(that->_nbElem);
  // Return the data
  return ret;
}

// Function to remove all the selement of the GSet pointing to 'data'
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void GSetRemoveAll(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set a pointer toward the tail of the GSet
  GSetElem* p = that->_tail;
  // Loop on elements until we reached the head of the list
  while (p != NULL) {
    // If the element points toward data
    if (p->_data == data) {
      // Memorize the previous element before deleting
      GSetElem* prev = p->_prev;
      // Remove the element
      GSetRemoveElem(that, &p);
      // Continue with previous element
      p = prev;
    // Else, the element doesn't point toward data
    } else {
      // Continue with previous element
      p = p->_prev;
    }
  }
}

// Function to get the data at the 'iElem'-th position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
```

```
void* GSetGet(GSet* that, int iElem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (iElem < 0 || iElem >= that->_nbElem) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'iElem' is invalid (0<=%d<%d)",
      iElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
#endif
  // Set a pointer for the return value
  void* ret = NULL;
  // Get the iElem-th element
  GSetElem* e = GSetGetElem(that, iElem);
  // Get the data of the element
  ret = e->_data;
  // Return the data
  return ret;
}

// Function to get the element at the 'iElem'-th position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetGetElem(GSet* that, int iElem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (iElem < 0 || iElem >= that->_nbElem) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'iElem' is invalid (0<=%d<%d)",
      iElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
#endif
  // Set a pointer for the return value
  GSetElem* ret = NULL;
  // Set the pointer to the head of the GSet
  ret = that->_head;
  // Move to the next element iElem times
  for (int i = iElem; i > 0 && ret != NULL; --i, ret = ret->_next);
  // Return the element
  return ret;
}

// Function to get the index of the first element of the GSet
// which point to 'data'
// Return -1 if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
int GSetGetIndexFirst(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set a pointer toward the head of the GSet
  GSetElem* p = that->_head;
  // Set a variable to memorize index
  int index = 0;
  // Loop on elements until we have found the
  // requested data or reached the end of the list
  while (p != NULL && p->_data != data) {
    ++index;
    p = p->_next;
  }
  // If the pointer is null it means the data wasn't in the GSet
  if (p == NULL)
    index = -1;
  // Return the index
  return index;
}

// Function to get the index of the last element of the GSet
// which point to 'data'
// Return -1 if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
int GSetGetIndexLast(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set a pointer toward the tail of the GSet
  GSetElem* p = that->_tail;
  // Set a variable to memorize index
  int index = that->_nbElem - 1;
  // Loop on elements until we have found the
  // requested data or reached the head of the list
  while (p != NULL && p->_data != data) {
    --index;
    p = p->_prev;
  }
  // Return the index
  return index;
}

// Function to get the first element of the GSet
// which point to 'data'
// Return NULL if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetGetFirstElem(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
```

```
  }
#endif
  // Set a pointer toward the head of the GSet
  GSetElem* p = that->_head;
  // Loop on elements until we have found the
  // requested data or reached the end of the list
  while (p != NULL && p->_data != data)
    p = p->_next;
  // Return the pointer
  return p;
}

// Function to get the last element of the GSet
// which point to 'data'
// Return NULL if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetGetLastElem(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set a pointer toward the head of the GSet
  GSetElem* p = that->_tail;
  // Loop on elements until we have found the
  // requested data or reached the end of the list
  while (p != NULL && p->_data != data)
    p = p->_prev;
  // Return the pointer
  return p;
}

// Merge the GSet 'r' at the end of the GSet 's'
// 'r' and 's' can be empty
// After calling this function r is empty
#if BUILDMODE != 0
inline
#endif
void GSetMerge(GSet* s, GSet* r) {
#if BUILDMODE == 0
  if (s == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'*s' is null");
    PBErrCatch(GSetErr);
  }
  if (r == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'*r' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // If the set r is not empty
  if (r->_nbElem != 0) {
    // If the set s is empty
    if (s->_nbElem == 0) {
      // Copy r into s
      memcpy(s, r, sizeof(GSet));
      // Empty r
```

```
      r->_head = NULL;
      r->_tail = NULL;
      r->_nbElem = 0;
    // Else, if the set s is not empty
    } else {
      // Add r to the tail of s
      s->_tail->_next = r->_head;
      // Add s to the head of r
      r->_head->_prev = s->_tail;
      // Update the tail of s
      s->_tail = r->_tail;
      // Update the number of element of s
      s->_nbElem += r->_nbElem;
      // Empty r
      r->_head = NULL;
      r->_tail = NULL;
      r->_nbElem = 0;
    }
  }
}

// Split the GSet at the GSetElem 'e'
// 'e' must be and element of the set
// the set new end is the element before 'e', the set becomes empty if
// 'e' was the first element
// Return a new GSet starting with 'e', or NULL if 'e' is not
// an element of the set
#if BUILDMODE != 0
inline
#endif
GSet* GSetSplit(GSet* that, GSetElem* e) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (e == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'e' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Check that e is an element of that
  // Declare a variable to count element before e in that
  int nb = 0;
  // If e is not the head of that
  if (that->_head != e) {
    GSetElem* ptr = e;
    // While there is an element before e
    do {
      // Increment the number of element
      ++nb;
      // Move to the previous element
      ptr = ptr->_prev;
    } while (ptr != NULL && ptr != that->_head);
    // If we have reached an element without previous element, this
    // element is not the head of that, meaning e is not in the set
    if (ptr == NULL)
      // Stop here
      return NULL;
  }
```

```
  // Allocate memory for the result
  GSet* res = GSetCreate();
  // Set the head of res
  res->_head = e;
  // Set the tail of res
  res->_tail = that->_tail;
  // Set the number of element of res
  res->_nbElem = that->_nbElem - nb;
  // Set the tail of s
  that->_tail = e->_prev;
  // Set the number of element of that
  that->_nbElem = nb;
  // If that is empty
  if (nb == 0)
    // Update head
    that->_head = NULL;
  // Else, that is not empty
  else
    // Disconnect the tail of that
    that->_tail->_next = NULL;
  // Disconnect the head of res
  res->_head->_prev = NULL;
  // Return the result
  return res;
}

// Switch the 'iElem'-th and 'jElem'-th element of the set
#if BUILDMODE != 0
inline
#endif
void GSetSwitch(GSet* that, int iElem, int jElem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (iElem < 0 || iElem >= that->_nbElem) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'iElem' is invalid (0<=%d<%d)",
      iElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
  if (jElem < 0 || jElem >= that->_nbElem) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'jElem' is invalid (0<=%d<%d)",
      jElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
#endif
  // Get the two elements
  GSetElem* iPtr = GSetGetElem(that, iElem);
  GSetElem* jPtr = GSetGetElem(that, jElem);
  // Switch the elements
  float v = iPtr->_sortVal;
  iPtr->_sortVal = jPtr->_sortVal;
  jPtr->_sortVal = v;
  void* dat = iPtr->_data;
  iPtr->_data = jPtr->_data;
  jPtr->_data = dat;
}
```

```
// Set the sort value of the GSetElem 'that' to 'v'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetSortVal(GSetElem* that, float v) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  that->_sortVal = v;
}

// Create a new GSetIterForward for the GSet 'set'
// The iterator is reset upon creation
#if BUILDMODE != 0
inline
#endif
GSetIterForward GSetIterForwardCreateStatic(GSet* set) {
#if BUILDMODE == 0
  if (set == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'set' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create the iterator
  GSetIterForward ret = {._set = set, ._curElem = set->_head};
  // Return the new iterator
  return ret;
}

// Create a new GSetIterBackward for the GSet 'set'
// The iterator is reset upon creation
#if BUILDMODE != 0
inline
#endif
GSetIterBackward GSetIterBackwardCreateStatic(GSet* set) {
#if BUILDMODE == 0
  if (set == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'set' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create the iterator
  GSetIterBackward ret = {._set = set, ._curElem = set->_tail};
  // Return the new iterator
  return ret;
}

// Reset the GSetIterForward to its starting position
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void GSetIterForwardReset(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Reset
  that->_curElem = that->_set->_head;
}

// Reset the GSetIterBackward to its starting position
// Do nothing if arguments are invalid
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardReset(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Reset
  that->_curElem = that->_set->_tail;
}

// Step the GSetIterForward
// Return false if arguments are invalid or we couldn't step
// Return true else
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardStep(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Step
  if (that->_curElem != NULL && that->_curElem->_next != NULL)
    that->_curElem = that->_curElem->_next;
  else
    return false;
  return true;
}

// Step the GSetIterBackward
// Return false if arguments are invalid or we couldn't step
// Return true else
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardStep(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Step
```

```
  if (that->_curElem != NULL && that->_curElem->_prev != NULL)
    that->_curElem = that->_curElem->_prev;
  else
    return false;
  return true;
}

// Apply a function to all elements of the GSet of the GSetIterForward
// The iterator is first reset, then the function is apply sequencially
// using the Step function of the iterator
// The applied function takes to void* arguments: 'data' is the _data
// property of the nodes, 'param' is a hook to allow the user to pass
// parameters to the function through a user-defined structure
#if BUILDMODE != 0
inline
#endif
void GSetIterForwardApply(GSetIterForward* that,
  void(*fun)(void* data, void* param), void* param) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (fun == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'fun' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Reset the iterator
  GSetIterReset(that);
  // If the set is not empty
  if (that->_curElem != NULL)
    // Loop on element
    do {
      // Apply the user function
      fun(that->_curElem->_data, param);
    } while (GSetIterStep(that) == true);
}

// Apply a function to all elements of the GSet of the GSetIterBackward
// The iterator is first reset, then the function is apply sequencially
// using the Step function of the iterator
// The applied function takes to void* arguments: 'data' is the _data
// property of the nodes, 'param' is a hook to allow the user to pass
// parameters to the function through a user-defined structure
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardApply(GSetIterBackward* that,
  void(*fun)(void* data, void* param), void* param) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (fun == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'fun' is null");
    PBErrCatch(GSetErr);
```

```
  }
#endif
  // Reset the iterator
  GSetIterReset(that);
  // If the set is not empty
  if (that->_curElem != NULL)
    // Loop on element
    do {
      // Apply the user function
      fun(that->_curElem->_data, param);
    } while (GSetIterStep(that) == true);
}

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardIsFirst(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  if (that->_curElem == that->_set->_head)
    return true;
  else
    return false;
}

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardIsFirst(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  if (that->_curElem == that->_set->_tail)
    return true;
  else
    return false;
}

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardIsLast(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  if (that->_curElem == that->_set->_tail)
    return true;
  else
    return false;
}

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardIsLast(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  if (that->_curElem == that->_set->_head)
    return true;
  else
    return false;
}

// Change the attached set of the iterator, and reset it
#if BUILDMODE != 0
inline
#endif
void GSetIterForwardSetGSet(GSetIterForward* that, GSet* set) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (set == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'set' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set the GSet
  that->_set = set;
  // Reset the iterator
  GSetIterReset(that);
}

// Change the attached set of the iterator, and reset it
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardSetGSet(GSetIterBackward* that, GSet* set) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (set == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'set' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set the GSet
  that->_set = set;
  // Reset the iterator
  GSetIterReset(that);
}

// Return the data currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void* GSetIterForwardGet(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data
  return that->_curElem->_data;
}

// Return the data currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void* GSetIterBackwardGet(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data
  return that->_curElem->_data;
}

// Return the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetIterForwardGetElem(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data
  return that->_curElem;
}
```

```
// Return the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GSetElem* GSetIterBackwardGetElem(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data
  return that->_curElem;
}

// Return the number of element in the set
#if BUILDMODE != 0
inline
#endif
int GSetNbElem(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data
  return that->_nbElem;
}

// Remove the element currently pointed to by the iterator
// The iterator is moved forward to the next element
// Return false if we couldn't move
// Return true else
// It's the responsibility of the user to delete the content of the
// element prior to calling this function
#if BUILDMODE != 0
inline
#endif
bool GSetIterForwardRemoveElem(GSetIterForward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  GSetElem *next = that->_curElem->_next;
  GSetRemoveElem(that->_set, &(that->_curElem));
  that->_curElem = next;
  if (next != NULL)
    return true;
  else
    return false;
}

// Remove the element currently pointed to by the iterator
// The iterator is moved backward to the next element
// Return false if we couldn't move
```

```
// Return true else
// It's the responsibility of the user to delete the content of the
// element prior to calling this function
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardRemoveElem(GSetIterBackward* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  GSetElem *prev = that->_curElem->_prev;
  GSetRemoveElem(that->_set, &(that->_curElem));
  that->_curElem = prev;
  if (prev != NULL)
    return true;
  else
    return false;
}
```

3 Makefile

```
#directory
PBERRDIR=../PBErr

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

include $(PBERRDIR)/Makefile.inc

INCPATH=-I./ -I$(PBERRDIR)/
BUILDOPTIONS=$(BUILDPARAM) $(INCPATH)

# compiler
COMPILER=gcc

#rules
all : main

main: main.o pberr.o gset.o Makefile
$(COMPILER) main.o pberr.o gset.o $(LINKOPTIONS) -o main

main.o : main.c $(PBERRDIR)/pberr.h gset.h gset-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c main.c

gset.o : gset.c gset.h gset-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c gset.c

pberr.o : $(PBERRDIR)/pberr.c $(PBERRDIR)/pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBERRDIR)/pberr.c

clean :
```

```
rm -rf *.o main

valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main

unitTest :
main > unitTest.txt; diff unitTest.txt unitTestRef.txt
```

4 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 <math.h>
#include "pberr.h"
#include "gset.h"

#define RANDOMSEED 0
#define rnd() (float)(rand())/(float)(RAND_MAX)

void UnitTestGSetCreateFree() {
  GSet* set = GSetCreate();
  if (set == NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "set is null");
    PBErrCatch(GSetErr);
  }
  if (set->_nbElem != 0) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_nbElem is invalid (%d==0)", set->_nbElem);
    PBErrCatch(GSetErr);
  }
  if (set->_head != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_head is not null");
    PBErrCatch(GSetErr);
  }
  if (set->_tail != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_tail is not null");
    PBErrCatch(GSetErr);
  }
  GSetFree(&set);
  if (set != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "set is not null after free");
    PBErrCatch(GSetErr);
  }
  set = GSetCreate();
  GSetPush(set, NULL);
  GSetFree(&set);
  if (set != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "set is not null after free");
```

```
    PBErrCatch(GSetErr);
  }
  GSet setstatic = GSetCreateStatic();
  if (setstatic._nbElem != 0) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_nbElem is invalid (%d==0)",
      setstatic._nbElem);
    PBErrCatch(GSetErr);
  }
  if (setstatic._head != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_head is not null");
    PBErrCatch(GSetErr);
  }
  if (setstatic._tail != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_tail is not null");
    PBErrCatch(GSetErr);
  }
  printf("UnitTestGSetCreateFree OK\n");
}

void UnitTestGSetClone() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSet* clone = GSetClone(&set);
  if (clone->_nbElem != 5) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetClone NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterForward iter = GSetIterForwardCreateStatic(clone);
  int i = 0;
  do {
    if (a + i != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetClone NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetFree(&clone);
  GSetFlush(&set);
  printf("UnitTestGSetClone OK\n");
}

void UnitTestGSetFlush() {
  GSet* set = GSetCreate();
  for (int i = 5; i--;)
    GSetPush(set, NULL);
  GSetFlush(set);
  if (set->_head != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_head is not null after flush");
    PBErrCatch(GSetErr);
  }
  if (set->_tail != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_tail is not null after flush");
    PBErrCatch(GSetErr);
```

```
  }
  if (set->_nbElem != 0) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "_nbElem is not 0 after flush");
    PBErrCatch(GSetErr);
  }
  GSetFree(&set);
  printf("UnitTestGSetFlush OK\n");
}

void printData(void* data, FILE* stream) {
  fprintf(stream, "%d", *(int*)data);
}

void UnitTestGSetPrint() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetPrint(&set, stdout, printData, ", ");
  printf("\n");
  GSetFlush(&set);
  printf("UnitTestGSetPrint OK\n");
}

void UnitTestGSetPushPopAppendDrop() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;) {
    GSetPush(&set, a + i);
    GSetPrint(&set, stdout, printData, ", ");
    printf("\n");
  }
  if (set._nbElem != 5) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetPushPopAppendDrop NOK");
    PBErrCatch(GSetErr);
  }
  for (int i = 5; i--;) {
    while (GSetPop(&set) == NULL);
    GSetPrint(&set, stdout, printData, ", ");
    printf("\n");
  }
  if (set._nbElem != 0) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetPushPopAppendDrop NOK");
    PBErrCatch(GSetErr);
  }
  for (int i = 5; i--;) {
    GSetAppend(&set, a + i);
    GSetPrint(&set, stdout, printData, ", ");
    printf("\n");
  }
  if (set._nbElem != 5) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetPushPopAppendDrop NOK");
    PBErrCatch(GSetErr);
  }
  for (int i = 5; i--;) {
    while (GSetDrop(&set) == NULL);
    GSetPrint(&set, stdout, printData, ", ");
    printf("\n");
```

```
  }
  if (set._nbElem != 0) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetPushPopAppendDrop NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetPushPopAppendDrop OK\n");
}

void UnitTestGSetAddSort() {
  srandom(RANDOMSEED);
  int a[5] = {-2, -1, 0, 1, 2};
  int nbTest = 1000;
  GSet set = GSetCreateStatic();
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  for (int iTest = nbTest; iTest--;) {
    for (int i = 10; i--;) {
      int j = (int)floor(rnd() * 5);
      GSetAddSort(&set, a + j, a[j]);
    }
    GSetIterReset(&iter);
    int v = *(int*)GSetIterGet(&iter);
    GSetIterStep(&iter);
    do {
      int w = *(int*)GSetIterGet(&iter);
      if (w < v) {
        GSetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GSetErr->_msg, "GSetAddSort NOK");
        PBErrCatch(GSetErr);
      }
      v = w;
    } while (GSetIterStep(&iter));
    GSetFlush(&set);
  }
  printf("UnitTestGSetAddSort OK\n");
}

void UnitTestGSetInsertRemove() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetInsert(&set, a, 2);
  int *checka[3] = {NULL, NULL, a};
  int i = 0;
  GSetIterReset(&iter);
  do {
    if (checka[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetInsert NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetFlush(&set);
  GSetInsert(&set, a, 0);
  GSetInsert(&set, a + 1, 1);
  GSetInsert(&set, a + 2, 1);
  GSetInsert(&set, a + 3, 1);
  GSetInsert(&set, a + 4, 3);
  int *checkb[5] = {a, a + 3, a + 2, a + 4, a + 1};
  i = 0;
```

```
GSetIterReset(&iter);
do {
  if (checkb[i] != GSetIterGet(&iter)) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetInsert NOK");
    PBErrCatch(GSetErr);
  }
  ++i;
} while (GSetIterStep(&iter));
GSetRemove(&set, 0);
int *checkc[4] = {a + 3, a + 2, a + 4, a + 1};
i = 0;
GSetIterReset(&iter);
do {
  if (checkc[i] != GSetIterGet(&iter)) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetRemove NOK");
    PBErrCatch(GSetErr);
  }
  ++i;
} while (GSetIterStep(&iter));
GSetRemove(&set, 3);
int *checkd[3] = {a + 3, a + 2, a + 4};
i = 0;
GSetIterReset(&iter);
do {
  if (checkd[i] != GSetIterGet(&iter)) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetRemove NOK");
    PBErrCatch(GSetErr);
  }
  ++i;
} while (GSetIterStep(&iter));
GSetRemove(&set, 1);
int *checke[2] = {a + 3, a + 4};
i = 0;
GSetIterReset(&iter);
do {
  if (checke[i] != GSetIterGet(&iter)) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetRemove NOK");
    PBErrCatch(GSetErr);
  }
  ++i;
} while (GSetIterStep(&iter));
GSetRemove(&set, 1);
int *checkf[1] = {a + 3};
i = 0;
GSetIterReset(&iter);
do {
  if (checkf[i] != GSetIterGet(&iter)) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetRemove NOK");
    PBErrCatch(GSetErr);
  }
  ++i;
} while (GSetIterStep(&iter));
GSetRemove(&set, 0);
if (set._nbElem != 0 || set._head != NULL || set._tail != NULL) {
  GSetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GSetErr->_msg, "GSetRemove NOK");
  PBErrCatch(GSetErr);
```

```
  }
  printf("UnitTestGSetInsertRemove OK\n");
}

void UnitTestGSetNbElemGet() {
  int a[5] = {0, 1, 2, 3, 4};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;) {
    GSetPush(&set, a + i);
    if (5 - i != GSetNbElem(&set)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetNbElem NOK");
      PBErrCatch(GSetErr);
    }
  }
  for (int i = 5; i--;)
    if (i != *(int*)GSetGet(&set, i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetGet NOK");
      PBErrCatch(GSetErr);
    }
  GSetFlush(&set);
  printf("UnitTestGSetNbElemGet OK\n");
}

void UnitTestGSetGetIndex() {
  int a[5] = {0, 1, 2, 3, 4};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  for (int i = 5; i--;)
    GSetAppend(&set, a + i);
  for (int i = 5; i--;) {
    if (i != GSetGetIndexFirst(&set, a + i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetGetIndexFirst NOK");
      PBErrCatch(GSetErr);
    }
    if (9 - i != GSetGetIndexLast(&set, a + i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetGetIndexLast NOK");
      PBErrCatch(GSetErr);
    }
  }
  GSetFlush(&set);
  printf("UnitTestGSetGetIndex OK\n");
}

void UnitTestGSetSort() {
  srandom(RANDOMSEED);
  int a[5] = {-2, -1, 0, 1, 2};
  int nbTest = 1000;
  GSet set = GSetCreateStatic();
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  for (int iTest = nbTest; iTest--;) {
    for (int i = 10; i--;) {
      int j = (int)floor(rnd() * 5);
      GSetPush(&set, a + j);
      GSetElemSetSortVal(GSetGetElem(&set, 0), a[j]);
    }
    GSetSort(&set);
    GSetIterReset(&iter);
```

```
    int v = *(int*)GSetIterGet(&iter);
    GSetIterStep(&iter);
    do {
      int w = *(int*)GSetIterGet(&iter);
      if (w < v) {
        GSetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GSetErr->_msg, "GSetSort NOK");
        PBErrCatch(GSetErr);
      }
      v = w;
    } while (GSetIterStep(&iter));
    GSetFlush(&set);
  }
  printf("UnitTestGSetSort OK\n");
}

void UnitTestGSetSplitMerge() {
  int a[5] = {0, 1, 2, 3, 4};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  for (int i = 5; i--;)
    GSetAppend(&set, a + i);
  GSet* split = GSetSplit(&set, GSetGetElem(&set, 5));
  if (split->_nbElem != 5 || set._nbElem != 5) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetSplit NOK");
    PBErrCatch(GSetErr);
  }
  for (int i = 5; i--;) {
    if (a[i] != *(int*)GSetGet(&set, i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetSplit NOK");
      PBErrCatch(GSetErr);
    }
    if (a[i] != *(int*)GSetGet(split, 4 - i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetSplit NOK");
      PBErrCatch(GSetErr);
    }
  }
  GSetMerge(&set, split);
  if (split->_nbElem != 0 || set._nbElem != 10) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetMerge NOK");
    PBErrCatch(GSetErr);
  }
  for (int i = 5; i--;) {
    if (i != GSetGetIndexFirst(&set, a + i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetMerge NOK");
      PBErrCatch(GSetErr);
    }
    if (9 - i != GSetGetIndexLast(&set, a + i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetMerge NOK");
      PBErrCatch(GSetErr);
    }
  }
  GSetFlush(&set);
  GSetFree(&split);
  printf("UnitTestGSetSplitMerge OK\n");
```

```
}

void UnitTestGSetSwitch() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetSwitch(&set, 0, 4);
  GSetSwitch(&set, 1, 3);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  int *checka[5] = {a + 4, a + 3, a + 2, a + 1, a};
  int i = 0;
  GSetIterReset(&iter);
  do {
    if (checka[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetSwitch NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetFlush(&set);
  printf("UnitTestGSetSwitch OK\n");
}

void UnitTestGSet() {
  UnitTestGSetCreateFree();
  UnitTestGSetClone();
  UnitTestGSetFlush();
  UnitTestGSetPrint();
  UnitTestGSetPushPopAppendDrop();
  UnitTestGSetAddSort();
  UnitTestGSetInsertRemove();
  UnitTestGSetNbElemGet();
  UnitTestGSetGetIndex();
  UnitTestGSetSort();
  UnitTestGSetSplitMerge();
  UnitTestGSetSwitch();
  printf("UnitTestGSet OK\n");
}

void UnitTestGSetIteratorForwardCreateFree() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterForward* iter = GSetIterForwardCreate(&set);
  if (iter->_set != &set || iter->_curElem != set._head) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardCreateFree NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterFree(&iter);
  if (iter != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "iter is not NULL after free");
    PBErrCatch(GSetErr);
  }
  GSetIterForward iterb = GSetIterForwardCreateStatic(&set);
  if (iterb._set != &set || iterb._curElem != set._head) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardCreateFree NOK");
```

```
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardCreateFree OK\n");
}

void UnitTestGSetIteratorForwardClone() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetIterForward* iterb = GSetIterClone(&iter);
  if (iter._set != iterb->_set || iter._curElem != iterb->_curElem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardClone NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterFree(&iterb);
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardClone OK\n");
}

void UnitTestGSetIteratorForwardReset() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetIterStep(&iter);
  GSetIterReset(&iter);
  if (iter._curElem != set._head) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardReset NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardReset OK\n");
}

void UnitTestGSetIteratorForwardStepGetGetElem() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetElem* elem = set._head->_next;
  GSetIterStep(&iter);
  if (iter._curElem != elem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorForwardStepGetGetElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGetElem(&iter) != elem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorForwardStepGetGetElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGet(&iter) != a + 1) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorForwardStepGetGetElem NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardStepGetGetElem OK\n");
}

void FunInc(void* data, void* param) {
  while (param != param);
  ++(*(int*)data);
}

void UnitTestGSetIteratorForwardApply() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetIterApply(&iter, &FunInc, NULL);
  for (int i = 5; i--;)
    if (a[i] != i + 2) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardApply NOK");
      PBErrCatch(GSetErr);
    }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardApply OK\n");
}

void UnitTestGSetIteratorForwardIsFirstIsLast() {
  int a[3] = {1, 2, 3};
  GSet set = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&set, a + i);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  if (GSetIterIsFirst(&iter) == false || GSetIterIsLast(&iter) == true) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorForwardIsFirstIsLast NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterStep(&iter);
  if (GSetIterIsFirst(&iter) == true || GSetIterIsLast(&iter) == true) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorForwardIsFirstIsLast NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterStep(&iter);
  if (GSetIterIsFirst(&iter) == true || GSetIterIsLast(&iter) == false) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorForwardIsFirstIsLast NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardIsFirstIsLast OK\n");
}

void UnitTestGSetIteratorForwardSetGSet() {
  int a[3] = {1, 2, 3};
```

```
  GSet set = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&set, a + i);
  int b[3] = {1, 2, 3};
  GSet setb = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&setb, b + i);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetIterSetGSet(&iter, &setb);
  if (iter._set != &setb || iter._curElem != setb._head) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardSetGSet NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  GSetFlush(&setb);
  printf("UnitTestGSetIteratorForwardSetGSet OK\n");
}

void UnitTestGSetIteratorForwardRemoveElem() {
  int a[3] = {1, 2, 3};
  GSet set = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&set, a + i);
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetIterStep(&iter);
  if (GSetIterRemoveElem(&iter) == false) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetNbElem(&set) != 2) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  if (iter._curElem != set._head->_next) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterRemoveElem(&iter) == true) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetNbElem(&set) != 1) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorForwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardRemoveElem OK\n");
}

void UnitTestGSetIteratorForward() {
  UnitTestGSetIteratorForwardCreateFree();
  UnitTestGSetIteratorForwardClone();
  UnitTestGSetIteratorForwardReset();
  UnitTestGSetIteratorForwardStepGetGetElem();
  UnitTestGSetIteratorForwardApply();
  UnitTestGSetIteratorForwardIsFirstIsLast();
```

```
  UnitTestGSetIteratorForwardSetGSet();
  UnitTestGSetIteratorForwardRemoveElem();
  printf("UnitTestGSetIteratorForward OK\n");
}

void UnitTestGSetIteratorBackwardCreateFree() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterBackward* iter = GSetIterBackwardCreate(&set);
  if (iter->_set != &set || iter->_curElem != set._tail) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardCreateFree NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterFree(&iter);
  if (iter != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "iter is not NULL after free");
    PBErrCatch(GSetErr);
  }
  GSetIterBackward iterb = GSetIterBackwardCreateStatic(&set);
  if (iterb._set != &set || iterb._curElem != set._tail) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardCreateFree NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorBackwardCreateFree OK\n");
}

void UnitTestGSetIteratorBackwardClone() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterBackward iter = GSetIterBackwardCreateStatic(&set);
  GSetIterBackward* iterb = GSetIterClone(&iter);
  if (iter._set != iterb->_set || iter._curElem != iterb->_curElem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardClone NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterFree(&iterb);
  GSetFlush(&set);
  printf("UnitTestGSetIteratorBackwardClone OK\n");
}

void UnitTestGSetIteratorBackwardReset() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterBackward iter = GSetIterBackwardCreateStatic(&set);
  GSetIterStep(&iter);
  GSetIterReset(&iter);
  if (iter._curElem != set._tail) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardReset NOK");
    PBErrCatch(GSetErr);
  }
```

```
  GSetFlush(&set);
  printf("UnitTestGSetIteratorBackwardReset OK\n");
}

void UnitTestGSetIteratorBackwardStepGetGetElem() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterBackward iter = GSetIterBackwardCreateStatic(&set);
  GSetElem* elem = set._tail->_prev;
  GSetIterStep(&iter);
  if (iter._curElem != elem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorBackwardStepGetGetElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGetElem(&iter) != elem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorBackwardStepGetGetElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGet(&iter) != a + 3) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorBackwardStepGetGetElem NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorBackwardStepGetGetElem OK\n");
}

void UnitTestGSetIteratorBackwardApply() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetIterBackward iter = GSetIterBackwardCreateStatic(&set);
  GSetIterApply(&iter, &FunInc, NULL);
  for (int i = 5; i--;)
    if (a[i] != i + 2) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardApply NOK");
      PBErrCatch(GSetErr);
    }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorBackwardApply OK\n");
}

void UnitTestGSetIteratorBackwardIsFirstIsLast() {
  int a[3] = {1, 2, 3};
  GSet set = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&set, a + i);
  GSetIterBackward iter = GSetIterBackwardCreateStatic(&set);
  if (GSetIterIsFirst(&iter) == false || GSetIterIsLast(&iter) == true) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorBackwardIsFirstIsLast NOK");
    PBErrCatch(GSetErr);
```

```
  }
  GSetIterStep(&iter);
  if (GSetIterIsFirst(&iter) == true || GSetIterIsLast(&iter) == true) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorBackwardIsFirstIsLast NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterStep(&iter);
  if (GSetIterIsFirst(&iter) == true || GSetIterIsLast(&iter) == false) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg,
      "UnitTestGSetIteratorBackwardIsFirstIsLast NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorBackwardIsFirstIsLast OK\n");
}

void UnitTestGSetIteratorBackwardSetGSet() {
  int a[3] = {1, 2, 3};
  GSet set = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&set, a + i);
  int b[3] = {1, 2, 3};
  GSet setb = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&setb, b + i);
  GSetIterBackward iter = GSetIterBackwardCreateStatic(&set);
  GSetIterSetGSet(&iter, &setb);
  if (iter._set != &setb || iter._curElem != setb._tail) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardSetGSet NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  GSetFlush(&setb);
  printf("UnitTestGSetIteratorBackwardSetGSet OK\n");
}

void UnitTestGSetIteratorBackwardRemoveElem() {
  int a[3] = {1, 2, 3};
  GSet set = GSetCreateStatic();
  for (int i = 3; i--;)
    GSetPush(&set, a + i);
  GSetIterBackward iter = GSetIterBackwardCreateStatic(&set);
  GSetIterStep(&iter);
  if (GSetIterRemoveElem(&iter) == false) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetNbElem(&set) != 2) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  if (iter._curElem != set._head) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
```

```
  if (GSetIterRemoveElem(&iter) == true) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetNbElem(&set) != 1) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "UnitTestGSetIteratorBackwardRemoveElem NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorBackwardRemoveElem OK\n");
}

void UnitTestGSetIteratorBackward() {
  UnitTestGSetIteratorBackwardCreateFree();
  UnitTestGSetIteratorBackwardClone();
  UnitTestGSetIteratorBackwardReset();
  UnitTestGSetIteratorBackwardStepGetGetElem();
  UnitTestGSetIteratorBackwardApply();
  UnitTestGSetIteratorBackwardIsFirstIsLast();
  UnitTestGSetIteratorBackwardSetGSet();
  printf("UnitTestGSetIteratorBackward OK\n");
}

void UnitTestAll() {
  UnitTestGSet();
  UnitTestGSetIteratorForward();
  UnitTestGSetIteratorBackward();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();
  // Return success code
  return 0;
}
```

5 Unit tests output

```
UnitTestGSetCreateFree OK
UnitTestGSetClone OK
UnitTestGSetFlush OK
1, 2, 3, 4, 5
UnitTestGSetPrint OK
5
4, 5
3, 4, 5
2, 3, 4, 5
1, 2, 3, 4, 5
2, 3, 4, 5
3, 4, 5
4, 5
5

5
5, 4
5, 4, 3
```

```
5, 4, 3, 2
5, 4, 3, 2, 1
5, 4, 3, 2
5, 4, 3
5, 4
5

UnitTestGSetPushPopAppendDrop OK
UnitTestGSetAddSort OK
UnitTestGSetInsertRemove OK
UnitTestGSetNbElemGet OK
UnitTestGSetGetIndex OK
UnitTestGSetSort OK
UnitTestGSetSplitMerge OK
UnitTestGSetSwitch OK
UnitTestGSet OK
UnitTestGSetIteratorForwardCreateFree OK
UnitTestGSetIteratorForwardClone OK
UnitTestGSetIteratorForwardReset OK
UnitTestGSetIteratorForwardStepGetGetElem OK
UnitTestGSetIteratorForwardApply OK
UnitTestGSetIteratorForwardIsFirstIsLast OK
UnitTestGSetIteratorForwardSetGSet OK
UnitTestGSetIteratorForward OK
UnitTestGSetIteratorBackwardCreateFree OK
UnitTestGSetIteratorBackwardClone OK
UnitTestGSetIteratorBackwardReset OK
UnitTestGSetIteratorBackwardStepGetGetElem OK
UnitTestGSetIteratorBackwardApply OK
UnitTestGSetIteratorBackwardIsFirstIsLast OK
UnitTestGSetIteratorBackwardSetGSet OK
UnitTestGSetIteratorBackward OK
UnitTestAll OK
```