# **GSet**

### P. Baillehache

### March 22, 2018

## Contents

1	Interface	2
2	Code         2.1 gset.c	
3	Makefile	49
4	Unit tests	<b>5</b> 0
5	Unit tests output	66

# 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, count elements and sort the set.

The library provides also GSetVecFloat, GSetVecShort, GSetBCurve, GSetSCurve structure with same interface as a GSet but whose contents is restrained to, respectively, VecFloat, VecShort, BCurve, SCurve structures.

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"
// ======== 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
  // Sorting in increasing value of _sortVal
 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 {
  \ensuremath{//} 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
```

```
// If the data is inserted inside the set, the current elements from
// the iElem-th elem are pushed
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 data at first position of the {\tt GSet}
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetGetFirst(GSet* that);
// Function to get the data at last position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetGetLast(GSet* that);
// 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 'set' at the end of the GSet 'that'
// 'that' and 'set' can be empty
// After calling this function 'set' is empty
#if BUILDMODE != 0
inline
#endif
void _GSetMerge(GSet* that, GSet* set);
// 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);
// Append the element of the GSet 'set' at the end of the GSet 'that'
// 'that' and 'set' can be empty
#if BUILDMODE != 0
inline
#endif
void _GSetAppendSet(GSet* that, GSet* set);
// Append the element of the GSet 'that' at the end of the GSet 'set'
// Elements are kept sorted
// 'that' and 'set' can be empty
#if BUILDMODE != 0
inline
#endif
void _GSetAppendSortedSet(GSet* that, GSet* set);
// Switch the 'iElem'-th and 'jElem'-th element of the set
#if BUILDMODE != 0
inline
#endif
void _GSetSwitch(GSet* that, int iElem, int jElem);
// Return the number of (GSetElem._data=='data') in the GSet 'that'
int _GSetCount(GSet* that, void* data);
// Set the sort value of the GSetElem 'that' to 'v'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetSortVal(GSetElem* that, float v);
// Move the 'iElem'-th element to the 'pos' index in the GSet
void _GSetMoveElem(GSet* that, int iElem, int pos);
// 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
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);
```

```
// 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);
// ======= Typed GSet =========
#ifndef VecFloat
  typedef struct VecFloat VecFloat;
#endif
#ifndef VecFloat2D
 typedef struct VecFloat2D VecFloat2D;
#endif
#ifndef VecFloat3D
  typedef struct VecFloat3D VecFloat3D;
#endif
typedef struct GSetVecFloat {GSet _set;} GSetVecFloat;
#define GSetVecFloatCreate() ((GSetVecFloat*)GSetCreate())
inline GSetVecFloat GSetVecFloatCreateStatic(void)
  {GSetVecFloat ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetVecFloat* GSetVecFloatClone(GSetVecFloat* that)
  {return (GSetVecFloat*)GSetClone((GSet*)that);}
inline VecFloat* _GSetVecFloatGet(GSetVecFloat* that, int iElem)
  {return (VecFloat*)_GSetGet((GSet*)that, iElem);}
inline VecFloat* _GSetVecFloatGetFirst(GSetVecFloat* that)
  {return (VecFloat*)_GSetGetFirst((GSet*)that);}
```

```
inline VecFloat* _GSetVecFloatGetLast(GSetVecFloat* that)
 {return (VecFloat*)_GSetGetLast((GSet*)that);}
inline VecFloat* _GSetVecFloatPop(GSetVecFloat* that)
 {return (VecFloat*)_GSetPop((GSet*)that);}
inline VecFloat* _GSetVecFloatDrop(GSetVecFloat* that)
 {return (VecFloat*)_GSetDrop((GSet*)that);}
inline VecFloat* _GSetVecFloatRemove(GSetVecFloat* that, int iElem)
 {return (VecFloat*)_GSetRemove((GSet*)that, iElem);}
inline VecFloat* _GSetVecFloatRemoveElem(GSetVecFloat* that,
 GSetElem** elem)
 {return (VecFloat*)_GSetRemoveElem((GSet*)that, elem);}
#ifndef VecShort
 typedef struct VecShort VecShort;
#endif
#ifndef VecShort2D
 typedef struct VecShort2D VecShort2D;
#endif
#ifndef VecShort3D
 typedef struct VecShort3D VecShort3D;
#endif
#ifndef VecShort4D
 typedef struct VecShort4D VecShort4D;
#endif
typedef struct GSetVecShort {GSet _set;} GSetVecShort;
#define GSetVecShortCreate() ((GSetVecShort*)GSetCreate())
inline GSetVecShort GSetVecShortCreateStatic(void)
 {GSetVecShort ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetVecShort* GSetVecShortClone(GSetVecShort* that)
 {return (GSetVecShort*)GSetClone((GSet*)that);}
inline VecShort* _GSetVecShortGet(GSetVecShort* that, int iElem)
  {return (VecShort*)_GSetGet((GSet*)that, iElem);}
inline VecShort* _GSetVecShortGetFirst(GSetVecShort* that)
 {return (VecShort*)_GSetGetFirst((GSet*)that);}
inline VecShort* _GSetVecShortGetLast(GSetVecShort* that)
 {return (VecShort*) GSetGetLast((GSet*)that):}
inline VecShort* _GSetVecShortPop(GSetVecShort* that)
 {return (VecShort*)_GSetPop((GSet*)that);}
inline VecShort* _GSetVecShortDrop(GSetVecShort* that)
 {return (VecShort*)_GSetDrop((GSet*)that);}
inline VecShort* _GSetVecShortRemove(GSetVecShort* that, int iElem)
 {return (VecShort*)_GSetRemove((GSet*)that, iElem);}
inline VecShort* _GSetVecShortRemoveElem(GSetVecShort* that,
 GSetElem** elem)
 {return (VecShort*)_GSetRemoveElem((GSet*)that, elem);}
#ifndef BCurve
 typedef struct BCurve BCurve;
#endif
typedef struct GSetBCurve {GSet _set;} GSetBCurve;
#define GSetBCurveCreate() ((GSetBCurve*)GSetCreate())
inline GSetBCurve GSetBCurveCreateStatic(void)
 {GSetBCurve ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetBCurve* GSetBCurveClone(GSetBCurve* that)
 {return (GSetBCurve*)GSetClone((GSet*)that):}
inline BCurve* _GSetBCurveGet(GSetBCurve* that, int iElem)
 {return (BCurve*)_GSetGet((GSet*)that, iElem);}
inline BCurve* _GSetBCurveGetFirst(GSetBCurve* that)
 {return (BCurve*)_GSetGetFirst((GSet*)that);}
inline BCurve* _GSetBCurveGetLast(GSetBCurve* that)
 {return (BCurve*)_GSetGetLast((GSet*)that);}
inline BCurve* _GSetBCurvePop(GSetBCurve* that)
```

```
{return (BCurve*)_GSetPop((GSet*)that);}
inline BCurve* _GSetBCurveDrop(GSetBCurve* that)
 {return (BCurve*)_GSetDrop((GSet*)that);}
inline BCurve* _GSetBCurveRemove(GSetBCurve* that, int iElem)
 {return (BCurve*)_GSetRemove((GSet*)that, iElem);}
inline BCurve* _GSetBCurveRemoveElem(GSetBCurve* that,
 GSetElem** elem)
 {return (BCurve*)_GSetRemoveElem((GSet*)that, elem);}
#ifndef SCurve
 typedef struct SCurve SCurve;
#endif
typedef struct GSetSCurve {GSet _set;} GSetSCurve;
#define GSetSCurveCreate() ((GSetSCurve*)GSetCreate())
inline GSetSCurve GSetSCurveCreateStatic(void)
 {GSetSCurve ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetSCurve* GSetSCurveClone(GSetSCurve* that)
 {return (GSetSCurve*)GSetClone((GSet*)that);}
inline SCurve* _GSetSCurveGet(GSetSCurve* that, int iElem)
 {return (SCurve*)_GSetGet((GSet*)that, iElem);}
inline SCurve* _GSetSCurveGetFirst(GSetSCurve* that)
 {return (SCurve*)_GSetGetFirst((GSet*)that);}
inline SCurve* _GSetSCurveGetLast(GSetSCurve* that)
 {return (SCurve*)_GSetGetLast((GSet*)that);}
inline SCurve* _GSetSCurvePop(GSetSCurve* that)
 {return (SCurve*)_GSetPop((GSet*)that);}
inline SCurve* _GSetSCurveDrop(GSetSCurve* that)
 {return (SCurve*)_GSetDrop((GSet*)that);}
inline SCurve* _GSetSCurveRemove(GSetSCurve* that, int iElem)
 {return (SCurve*)_GSetRemove((GSet*)that, iElem);}
inline SCurve* _GSetSCurveRemoveElem(GSetSCurve* that,
 GSetElem** elem)
 {return (SCurve*)_GSetRemoveElem((GSet*)that, elem);}
#ifndef Shapoid
 typedef struct Shapoid Shapoid;
#endif
#ifndef Facoid
 typedef struct Facoid Facoid;
#endif
#ifndef Spheroid
 typedef struct Spheroid Spheroid;
#endif
#ifndef Pyramidoid
 typedef struct Pyramidoid Pyramidoid;
#endif
typedef struct GSetShapoid {GSet _set;} GSetShapoid;
#define GSetShapoidCreate() ((GSetShapoid*)GSetCreate())
inline GSetShapoid GSetShapoidCreateStatic(void)
 {GSetShapoid ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetShapoid* GSetShapoidClone(GSetShapoid* that)
 {return (GSetShapoid*)GSetClone((GSet*)that);}
inline Shapoid* _GSetShapoidGet(GSetShapoid* that, int iElem)
 {return (Shapoid*)_GSetGet((GSet*)that, iElem);}
inline Shapoid* _GSetShapoidGetFirst(GSetShapoid* that)
 {return (Shapoid*)_GSetGetFirst((GSet*)that);}
inline Shapoid* _GSetShapoidGetLast(GSetShapoid* that)
 {return (Shapoid*)_GSetGetLast((GSet*)that);}
inline Shapoid* _GSetShapoidPop(GSetShapoid* that)
 {return (Shapoid*)_GSetPop((GSet*)that);}
inline Shapoid* _GSetShapoidDrop(GSetShapoid* that)
 {return (Shapoid*)_GSetDrop((GSet*)that);}
```

```
inline Shapoid* _GSetShapoidRemove(GSetShapoid* that, int iElem)
  {return (Shapoid*)_GSetRemove((GSet*)that, iElem);}
inline Shapoid* _GSetShapoidRemoveElem(GSetShapoid* that,
 GSetElem** elem)
  {return (Shapoid*)_GSetRemoveElem((GSet*)that, elem);}
// ====== Generic functions =========
#define GSetFree(Set) _Generic(Set, \
 GSet**: _GSetFree, \
  GSetVecFloat**: _GSetFree, \
  GSetVecShort**: _GSetFree, \
 GSetBCurve**: _GSetFree, \
  GSetSCurve**: _GSetFree, \
  GSetShapoid**: _GSetFree, \
  default: PBErrInvalidPolymorphism)((GSet**)(Set))
#define GSetPush(Set, Data) _Generic(Set, \
  GSet*: _Generic(Data, \
    default: _GSetPush), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetPush, \
    VecFloat2D*: _GSetPush, \
    VecFloat3D*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetPush, \
   VecShort2D*: _GSetPush, \
VecShort3D*: _GSetPush, \
    VecShort4D*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetPush, \
    Facoid*: _GSetPush, \
    Pyramidoid*: _GSetPush, \
    Spheroid*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetAddSort(Set, Data, Value) _Generic(Set, \
 GSet*: _Generic(Data, \
    default: _GSetAddSort), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetAddSort, \
   VecFloat2D*: _GSetAddSort, \
VecFloat3D*: _GSetAddSort, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetAddSort, \
    VecShort2D*: _GSetAddSort, \
   VecShort3D*: _GSetAddSort, \
VecShort4D*: _GSetAddSort, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetAddSort, \
    default: PBErrInvalidPolymorphism), \
```

```
GSetSCurve*: _Generic(Data, \
   SCurve*: _GSetAddSort, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(Data, \
   Shapoid*: _GSetAddSort, \
   Facoid*: _GSetAddSort, \
   Pyramidoid*: _GSetAddSort, \
   Spheroid*: _GSetAddSort, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data), Value)
#define GSetInsert(Set, Data, Pos) _Generic(Set, \
 GSet*: _Generic(Data, \
   default: _GSetInsert), \
 GSetVecFloat*: _Generic(Data, \
   VecFloat*: _GSetInsert, \
   VecFloat2D*: _GSetInsert, \
   VecFloat3D*: _GSetInsert, \
   default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(Data, \
   VecShort*: _GSetInsert, \
   VecShort2D*: _GSetInsert, \
   VecShort3D*: _GSetInsert, \
   VecShort4D*: _GSetInsert, \
   default: PBErrInvalidPolymorphism), \
 GSetBCurve*: _Generic(Data, \
   BCurve*: _GSetInsert, \
   default: PBErrInvalidPolymorphism), \
 GSetSCurve*: _Generic(Data, \
   SCurve*: _GSetInsert, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(Data, \
   Shapoid*: _GSetInsert, \
   Facoid*: _GSetInsert, \
   Pyramidoid*: _GSetInsert, \
   Spheroid*: _GSetInsert, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data), Pos)
#define GSetAppend(Set, Data) _Generic(Set, \
 GSet*: _Generic(Data, \
   default: _GSetAppend), \
 GSetVecFloat*: _Generic(Data, \
   VecFloat*: _GSetAppend, \
   VecFloat2D*: _GSetAppend, \
   VecFloat3D*: _GSetAppend, \
   default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(Data, \
   VecShort*: _GSetAppend, \
   VecShort2D*: _GSetAppend, \
   VecShort3D*: _GSetAppend, \
   VecShort4D*: _GSetAppend, \
   default: PBErrInvalidPolymorphism), \
 GSetBCurve*: _Generic(Data, \
   BCurve*: _GSetAppend, \
   default: PBErrInvalidPolymorphism), \
 GSetSCurve*: _Generic(Data, \
   SCurve*: _GSetAppend, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(Data, \
   Shapoid*: _GSetAppend, \
   Facoid*: _GSetAppend, \
```

```
Pyramidoid*: _GSetAppend, \
    Spheroid*: _GSetAppend, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetRemoveFirst(Set, Data) _Generic(Set, \
  GSet*: _Generic(Data, \
    default: _GSetRemoveFirst), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetRemoveFirst, \
    \label{eq:VecFloat2D*: _GSetRemoveFirst, } VecFloat2D*: _GSetRemoveFirst, \\ \setminus
    VecFloat3D*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetRemoveFirst, \
    VecShort2D*: _GSetRemoveFirst, \
    VecShort3D*: _GSetRemoveFirst, \
    VecShort4D*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetRemoveFirst, \
    Facoid*: _GSetRemoveFirst, \
    Pyramidoid*: _GSetRemoveFirst, \
    Spheroid*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetRemoveLast(Set, Data) _Generic(Set, \
  GSet*: _Generic(Data, \
   default: _GSetRemoveLast), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetRemoveLast, \
    VecFloat2D*: _GSetRemoveLast, \
    VecFloat3D*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetRemoveLast, \
    VecShort2D*: _GSetRemoveLast, \
    VecShort3D*: _GSetRemoveLast, \
    VecShort4D*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetRemoveLast, \
    Facoid*: _GSetRemoveLast, \
    Pyramidoid*: _GSetRemoveLast, \
    Spheroid*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetRemoveAll(Set, Data) _Generic(Set, \
```

```
GSet*: _Generic(Data, \
   default: _GSetRemoveAll), \
 GSetVecFloat*: _Generic(Data, \
   VecFloat*: _GSetRemoveAll, \
   VecFloat2D*: _GSetRemoveAll, \
   VecFloat3D*: _GSetRemoveAll, \
   default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(Data, \
   VecShort*: _GSetRemoveAll, \
   VecShort2D*: _GSetRemoveAll, \
   VecShort3D*: _GSetRemoveAll, \
   VecShort4D*: _GSetRemoveAll, \
   default: PBErrInvalidPolymorphism), \
 GSetBCurve*: _Generic(Data, \
   BCurve*: _GSetRemoveAll, \
   default: PBErrInvalidPolymorphism), \
 GSetSCurve*: _Generic(Data, \
   SCurve*: _GSetRemoveAll, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(Data, \
   Shapoid*: _GSetRemoveAll, \
   Facoid*: _GSetRemoveAll, \
   Pyramidoid*: _GSetRemoveAll, \
   Spheroid*: _GSetRemoveAll, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetGetIndexFirst(Set, Data) _Generic(Set, \
 GSet*: _Generic(Data, \
   default: _GSetGetIndexFirst), \
 GSetVecFloat*: _Generic(Data, \
   VecFloat*: _GSetGetIndexFirst, \
   VecFloat2D*: _GSetGetIndexFirst, \
   VecFloat3D*: _GSetGetIndexFirst, \
   default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(Data, \
   VecShort*: _GSetGetIndexFirst, \
   VecShort2D*: _GSetGetIndexFirst, \
   VecShort3D*: _GSetGetIndexFirst, \
   VecShort4D*: _GSetGetIndexFirst, \
   default: PBErrInvalidPolymorphism), \
 GSetBCurve*: _Generic(Data, \
   BCurve*: _GSetGetIndexFirst, \
   default: PBErrInvalidPolymorphism), \
 GSetSCurve*: _Generic(Data, \
   SCurve*: _GSetGetIndexFirst, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(Data, \
   Shapoid*: _GSetGetIndexFirst, \
   Facoid*: _GSetGetIndexFirst, \
   Pyramidoid*: _GSetGetIndexFirst, \
   Spheroid*: _GSetGetIndexFirst, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetGetIndexLast(Set, Data) _Generic(Set, \
 GSet*: _Generic(Data, \
   default: _GSetGetIndexLast), \
 GSetVecFloat*: _Generic(Data, \
   VecFloat*: _GSetGetIndexLast, \
   VecFloat2D*: _GSetGetIndexLast, \
   VecFloat3D*: _GSetGetIndexLast, \
```

```
default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(Data, \
   VecShort*: _GSetGetIndexLast, \
   VecShort2D*: _GSetGetIndexLast, \
   VecShort3D*: _GSetGetIndexLast, \
   VecShort4D*: _GSetGetIndexLast, \
   default: PBErrInvalidPolymorphism), \
 GSetBCurve*: _Generic(Data, \
   BCurve*: _GSetGetIndexLast, \
   default: PBErrInvalidPolymorphism), \
 GSetSCurve*: _Generic(Data, \
   SCurve*: _GSetGetIndexLast, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(Data, \
   Shapoid*: _GSetGetIndexLast, \
   Facoid*: _GSetGetIndexLast, \
   Pyramidoid*: _GSetGetIndexLast, \
   Spheroid*: _GSetGetIndexLast, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetGetFirstElem(Set, Data) _Generic(Set, \
 GSet*: _Generic(Data, \
   default: _GSetGetFirstElem), \
 GSetVecFloat*: _Generic(Data, \
   VecFloat*: _GSetGetFirstElem, \
   VecFloat2D*: _GSetGetFirstElem, \
   VecFloat3D*: _GSetGetFirstElem, \
   default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(Data, \
   VecShort*: _GSetGetFirstElem, \
   VecShort2D*: _GSetGetFirstElem, \
   VecShort3D*: _GSetGetFirstElem, \
   VecShort4D*: _GSetGetFirstElem, \
   default: PBErrInvalidPolymorphism), \
 GSetBCurve*: _Generic(Data, \
   BCurve*: _GSetGetFirstElem, \
   default: PBErrInvalidPolymorphism), \
 GSetSCurve*: _Generic(Data, \
   SCurve*: _GSetGetFirstElem, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(Data, \
   Shapoid*: _GSetGetFirstElem, \
   Facoid*: _GSetGetFirstElem, \
   Pyramidoid*: _GSetGetFirstElem, \
   Spheroid*: _GSetGetFirstElem, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetGetLastElem(Set, Data) _Generic(Set, \
 GSet*: _Generic(Data, \
   default: _GSetGetLastElem), \
 GSetVecFloat*: _Generic(Data, \
   VecFloat*: _GSetGetLastElem, \
   VecFloat2D*: _GSetGetLastElem, \
   VecFloat3D*: _GSetGetLastElem, \
   default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(Data, \
   VecShort*: _GSetGetLastElem, \
   VecShort2D*: _GSetGetLastElem, \
   VecShort3D*: _GSetGetLastElem, \
   VecShort4D*: _GSetGetLastElem, \
```

```
default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetGetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetGetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetGetLastElem, \
    Facoid*: _GSetGetLastElem, \
    Pyramidoid*: _GSetGetLastElem, \
    Spheroid*: _GSetGetLastElem, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
#define GSetPrint(Set, Stream, Fun, Sep) _Generic(Set, \
 GSet*: _GSetPrint, \
  GSetVecFloat*: _GSetPrint, \
  GSetVecShort*: _GSetPrint, \
 GSetBCurve*: _GSetPrint, \
  GSetSCurve*: _GSetPrint, \
  GSetShapoid*: _GSetPrint, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), Stream, Fun, Sep)
#define GSetFlush(Set) _Generic(Set, \
  GSet*: _GSetFlush, \
  GSetVecFloat*: _GSetFlush, \
  GSetVecShort*: _GSetFlush, \
 GSetBCurve*: _GSetFlush, \
 GSetSCurve*: _GSetFlush, \
  GSetShapoid*: _GSetFlush, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))
#define GSetNbElem(Set) _Generic(Set, \
  GSet*: _GSetNbElem, \
 GSetVecFloat*: _GSetNbElem, \
 GSetVecShort*: _GSetNbElem, \
 GSetBCurve*: _GSetNbElem, \
GSetSCurve*: _GSetNbElem, \
 GSetShapoid*: _GSetNbElem, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set))
#define GSetPop(Set) _Generic(Set, \
 GSet*: _GSetPop, \
 GSetVecFloat*: _GSetVecFloatPop, \
  GSetVecShort*: _GSetVecShortPop, \
  GSetBCurve*: _GSetBCurvePop, \
 GSetSCurve*: _GSetSCurvePop, \
  GSetShapoid*: _GSetShapoidPop, \
 default: PBErrInvalidPolymorphism)(Set)
#define GSetDrop(Set) _Generic(Set, \
 GSet*: _GSetDrop, \
 GSetVecFloat*: _GSetVecFloatDrop, \
GSetVecShort*: _GSetVecShortDrop, \
  GSetBCurve*: _GSetBCurveDrop, \
  GSetSCurve*: _GSetSCurveDrop, \
 GSetShapoid*: _GSetSCurveDrop, \
  default: PBErrInvalidPolymorphism)(Set)
#define GSetRemove(Set, Pos) _Generic(Set, \
  GSet*: _GSetRemove, \
```

```
GSetVecFloat*: _GSetVecFloatRemove, \
  GSetVecShort*: _GSetVecShortRemove, \
  GSetBCurve*: _GSetBCurveRemove, \
  GSetSCurve*: _GSetSCurveRemove, \
  GSetShapoid*: _GSetSCurveRemove, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), Pos)
#define GSetRemoveElem(Set, Elem) _Generic(Set, \
  GSet*: _GSetRemoveElem, \
 {\tt GSetVecFloat*: \_GSetVecFloatRemoveElem, \ } \\
  GSetVecShort*: _GSetVecShortRemoveElem, \
  GSetBCurve*: _GSetBCurveRemoveElem, \
 GSetSCurve*: _GSetSCurveRemoveElem, \
  GSetShapoid*: _GSetSCurveRemoveElem, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), Elem)
#define GSetGet(Set, Pos) _Generic(Set, \
 GSet*: _GSetGet, \
  GSetVecFloat*: _GSetVecFloatGet, \
  GSetVecShort*: _GSetVecShortGet, \
  GSetBCurve*: _GSetBCurveGet, \
 GSetSCurve*: _GSetSCurveGet, \
GSetShapoid*: _GSetSCurveGet, \
  default: PBErrInvalidPolymorphism)(Set, Pos)
#define GSetGetFirst(Set) _Generic(Set, \
  GSet*: _GSetGetFirst, \
  GSetVecFloat*: _GSetVecFloatGetFirst, \
 GSetVecShort*: _GSetVecShortGetFirst, \
  GSetBCurve*: _GSetBCurveGetFirst, \
  GSetSCurve*: _GSetSCurveGetFirst, \
 GSetShapoid*: _GSetSCurveGetFirst, \
 default: PBErrInvalidPolymorphism)(Set)
#define GSetGetLast(Set) _Generic(Set, \
 GSet*: _GSetGetLast, \
  GSetVecFloat*: _GSetVecFloatGetLast, \
  GSetVecShort*: _GSetVecShortGetLast, \
  GSetBCurve*: _GSetBCurveGetLast, \
  GSetSCurve*: _GSetSCurveGetLast, \
  GSetShapoid*: _GSetSCurveGetLast, \
  default: PBErrInvalidPolymorphism)(Set)
#define GSetGetElem(Set, Pos) _Generic(Set, \
  GSet*: _GSetGetElem, \
  GSetVecFloat*: _GSetGetElem, \
  GSetVecShort*: _GSetGetElem, \
 GSetBCurve*: _GSetGetElem, \
 GSetSCurve*: _GSetGetElem, \
  GSetShapoid*: _GSetGetElem, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), Pos)
#define GSetSort(Set) _Generic(Set, \
  GSet*: _GSetSort, \
 GSetVecFloat*: _GSetSort, \
  GSetVecShort*: _GSetSort, \
  GSetBCurve*: _GSetSort, \
 GSetSCurve*: _GSetSort, \
  GSetShapoid*: _GSetSort, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))
#define GSetMerge(IntoSet, MergedSet) _Generic(IntoSet, \
```

```
GSet*: _Generic(MergedSet, \
    GSet*: _GSetMerge, \
    GSetVecFloat*: _GSetMerge, \
    GSetVecShort*: _GSetMerge, \
    GSetBCurve*: _GSetMerge, \
   GSetSCurve*: _GSetMerge, \
GSetShapoid*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetVecFloat*: _Generic(MergedSet, \
    GSetVecFloat*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(MergedSet, \
    GSetVecFloat*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(MergedSet, \
    GSetBCurve*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(MergedSet, \
    GSetSCurve*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(MergedSet, \
    GSetShapoid*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(IntoSet), \
    (GSet*)(MergedSet))
#define GSetSplit(Set, Elem) _Generic(Set, \
  GSet*: _GSetSplit, \
  GSetVecFloat*: _GSetSplit, \
  GSetVecShort*: _GSetSplit, \
  GSetBCurve*: _GSetSplit, \
  GSetSCurve*: _GSetSplit, \
  GSetShapoid*: _GSetSplit, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), Elem)
#define GSetAppendSet(IntoSet, AppendSet) _Generic(IntoSet, \
  GSet*: _Generic(AppendSet, \
    GSet*: _GSetAppendSet, \
    GSetVecFloat*: _GSetAppendSet, \
    GSetVecShort*: _GSetAppendSet, \
    GSetBCurve*: _GSetAppendSet, \
    GSetSCurve*: _GSetAppendSet, \
    GSetShapoid*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetVecFloat*: _Generic(AppendSet, \
    GSetVecFloat*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(AppendSet, \
    GSetVecShort*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(AppendSet, \
    GSetBCurve*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(AppendSet, \
    GSetSCurve*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(AppendSet, \
    GSetShapoid*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(IntoSet), \
    (GSet*)(AppendSet))
```

```
#define GSetAppendSortedSet(IntoSet, AppendSet) _Generic(IntoSet, \
 GSet*: _Generic(AppendSet, \
   GSet*: _GSetAppendSortedSet, \
   GSetVecFloat*: _GSetAppendSortedSet, \
   GSetVecShort*: _GSetAppendSortedSet, \
   GSetBCurve*: _GSetAppendSortedSet, \
   GSetSCurve*: _GSetAppendSortedSet, \
   GSetShapoid*: _GSetAppendSortedSet, \
   default: PBErrInvalidPolymorphism), \
 GSetVecFloat*: _Generic(AppendSet, \
   GSetVecFloat*: _GSetAppendSortedSet, \
   default: PBErrInvalidPolymorphism), \
 GSetVecShort*: _Generic(AppendSet, \
   GSetVecShort*: _GSetAppendSortedSet, \
   default: PBErrInvalidPolymorphism), \
 GSetBCurve*: _Generic(AppendSet, \
   GSetBCurve*: _GSetAppendSortedSet, \
   default: PBErrInvalidPolymorphism), \
 GSetSCurve*: _Generic(AppendSet, \
   GSetSCurve*: _GSetAppendSortedSet, \
   default: PBErrInvalidPolymorphism), \
 GSetShapoid*: _Generic(AppendSet, \
   GSetShapoid*: _GSetAppendSortedSet, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)((GSet*)(IntoSet), \
    (GSet*)(AppendSet))
#define GSetSwitch(Set, PosA, PosB) _Generic(Set, \
 GSet*: _GSetSwitch, \
 GSetVecFloat*: _GSetSwitch, \
 GSetVecShort*: _GSetSwitch, \
 GSetBCurve*: _GSetSwitch, \
 GSetSCurve*: _GSetSwitch, \
 GSetShapoid*: _GSetSwitch, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), PosA, PosB)
#define GSetMoveElem(Set, From, To) _Generic(Set, \
 GSet*: _GSetMoveElem, \
 GSetVecFloat*: _GSetMoveElem, \
 GSetVecShort*: _GSetMoveElem, \
 GSetBCurve*: _GSetMoveElem, \
 GSetSCurve*: _GSetMoveElem, \
 GSetShapoid*: _GSetMoveElem, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), From, To)
#define GSetCount(Set, Data) _Generic(Set, \
 GSet*: _GSetCount, \
 GSetVecFloat*: _GSetCount, \
 GSetVecShort*: _GSetCount, \
 GSetBCurve*: _GSetCount, \
 GSetSCurve*: _GSetCount, \
 GSetShapoid*: _GSetCount, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set), Data)
#define GSetIterForwardCreate(Set) _Generic(Set, \
 GSet*: _GSetIterForwardCreate, \
 GSetVecFloat*: _GSetIterForwardCreate, \
 GSetVecShort*: _GSetIterForwardCreate, \
 GSetBCurve*: _GSetIterForwardCreate, \
 GSetSCurve*: _GSetIterForwardCreate, \
 GSetShapoid*: _GSetIterForwardCreate, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set))
```

```
#define GSetIterForwardCreateStatic(Set) _Generic(Set, \
 GSet*: _GSetIterForwardCreateStatic, \
 GSetVecFloat*: _GSetIterForwardCreateStatic, \
 GSetVecShort*: _GSetIterForwardCreateStatic, \
 GSetBCurve*: _GSetIterForwardCreateStatic, \
 GSetSCurve*: _GSetIterForwardCreateStatic, \
 GSetShapoid*: _GSetIterForwardCreateStatic, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set))
#define GSetIterBackwardCreate(Set) _Generic(Set, \
 GSet*: _GSetIterBackwardCreate, \
 GSetVecFloat*: _GSetIterBackwardCreate, \
 GSetVecShort*: _GSetIterBackwardCreate, \
 GSetBCurve*: _GSetIterBackwardCreate, \
 GSetSCurve*: _GSetIterBackwardCreate, \
 GSetShapoid*: _GSetIterBackwardCreate, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set))
#define GSetIterBackwardCreateStatic(Set) _Generic(Set, \
 GSet*: _GSetIterBackwardCreateStatic, \
 GSetVecFloat*: _GSetIterBackwardCreateStatic, \
 GSetVecShort*: _GSetIterBackwardCreateStatic, \
 GSetBCurve*: _GSetIterBackwardCreateStatic, \
 GSetSCurve*: _GSetIterBackwardCreateStatic, \
 GSetShapoid*: _GSetIterBackwardCreateStatic, \
 default: PBErrInvalidPolymorphism)((GSet*)(Set))
#define GSetIterSetGSet(Iter, Set) _Generic(Iter, \
 GSetIterForward*: _Generic(Set, \
   GSet*: GSetIterForwardSetGSet, \
   GSetVecFloat*: GSetIterForwardSetGSet, \
   GSetVecShort*: GSetIterForwardSetGSet, \
   GSetBCurve*: GSetIterForwardSetGSet, \
   GSetSCurve*: GSetIterForwardSetGSet, \
   GSetShapoid*: GSetIterForwardSetGSet, \
   default: PBErrInvalidPolymorphism), \
 GSetIterBackward*: _Generic(Set, \
   GSet*: GSetIterBackwardSetGSet, \
   GSetVecFloat*: GSetIterBackwardSetGSet, \
   GSetVecShort*: GSetIterBackwardSetGSet, \
   GSetBCurve*: GSetIterBackwardSetGSet, \
   GSetSCurve*: GSetIterBackwardSetGSet, \
   GSetShapoid*: GSetIterBackwardSetGSet, \
   default: PBErrInvalidPolymorphism), \
 default: PBErrInvalidPolymorphism)(Iter, (GSet*)(Set))
#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 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)
// ======== Inliner ========
#if BUILDMODE != 0
#include "gset-inline.c"
#endif
#endif
```

### 2 Code

### 2.1 gset.c

```
// Return a pointer toward the new GSet
GSet* GSetCreate(void) {
  // Allocate memory for the GSet
  GSet* s = PBErrMalloc(GSetErr, sizeof(GSet));
  \ensuremath{//} Set the pointer to head and tail, and the number of element
  s->_head = NULL;
  s->_tail = NULL;
  s \rightarrow 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);
  \ensuremath{//} 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;
 \ensuremath{//} 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 \rightarrow prev \rightarrow 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
// If the data is inserted inside the set, the current elements from
// the iElem-th elem are pushed
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 \rightarrow 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
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);
  \ensuremath{//} 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;
  \ensuremath{//} 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);
      \ensuremath{//} If the poped element has a sort value lower than the pivot
      if (val < pivot) {</pre>
        // 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;
// Move the 'iElem'-th element to the 'pos' index in the GSet
void _GSetMoveElem(GSet* that, int iElem, int pos) {
#if BUILDMODE == 0
 if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
   PBErrCatch(GenBrushErr);
  if (iElem < 0 || iElem >= GSetNbElem(that)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'iElem' is invalid (0<=%d<%d)",</pre>
     iElem, GSetNbElem(that));
    PBErrCatch(GenBrushErr);
```

```
if (pos < 0 || pos >= GSetNbElem(that)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is invalid (0<=%d<%d)",
      pos, GSetNbElem(that));
    PBErrCatch(GenBrushErr);
  }
#endif
  // If the origin and destination position are the same
  // there is nothing to do
  if (iElem == pos)
    return;
  // Get a pointer to the mmoved element
  GSetElem* elem = GSetGetElem(that, iElem);
  //Declare two variables to memorize the sort value and data
  // of the moved element
  float sortVal = elem->_sortVal;
  void* data = elem->_data;
  // Remove the moved element
  GSetRemove(that, iElem);
  // Insert new element
  GSetInsert(that, data, pos);
  // Get a pointer to the newly inserted element
  elem = GSetGetElem(that, pos);
  // Correct the sorted value with the original value
  elem->_sortVal = sortVal;
}
// Return the number of (GSetElem._data=='data') in the GSet 'that'
int _GSetCount(GSet* that, void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
#endif
  // Declare a variable to memorize the result
  int nb = 0;
  // If the set is not empty
  if (GSetNbElem(that) > 0) {
    // Loop on the set's elements
    GSetIterForward iter = GSetIterForwardCreateStatic(that);
    do {
      // If the current element's data is the searched data
      if (GSetIterGet(&iter) == data)
        // Increment the result
        ++nb;
    } while (GSetIterStep(&iter));
  }
  // return the result
 return nb;
}
// 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
{\tt GSetIterBackward*} \ {\tt GSetIterBackwardClone(GSetIterBackward*} \ {\tt 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) {
  \ensuremath{//} 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 \rightarrow 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
  \ensuremath{//} 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;
    {\tt 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
  \ensuremath{//} 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) {
    \ensuremath{//} 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 data at first position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetGetFirst(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
   PBErrCatch(GSetErr);
#endif
  \ensuremath{//} Set a pointer for the return value
  void* ret = that->_head->_data;
  // Return the data
 return ret;
// Function to get the data at last position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetGetLast(GSet* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
#endif
  // Set a pointer for the return value
  void* ret = that->_tail->_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;
  \ensuremath{//} 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 {\tt 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 'set' at the end of the GSet 'that'
// 'that' and 'set' can be empty
```

```
// After calling this function 'set' is empty
#if BUILDMODE != 0
inline
#endif
void _GSetMerge(GSet* 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
  // If 'set' is not empty
  if (set->_nbElem != 0) {
    // If 'that' is empty
    if (that->_nbElem == 0) {
      // Copy 'set' into 'that'
      memcpy(that, set, sizeof(GSet));
      // Empty 'set'
      set->_head = NULL;
      set->_tail = NULL;
      set->_nbElem = 0;
    // Else, if 'that' is not empty
    } else {
      // Add 'set' to the tail of 'that'
      that->_tail->_next = set->_head;
      // Add 'that' to the head of 'set'
      set->_head->_prev = that->_tail;
      // Update the tail of 'that'
      that->_tail = set->_tail;
      // Update the number of element of 'that'
      that->_nbElem += set->_nbElem;
      // Empty 'set'
      set->_head = NULL;
      set->_tail = NULL;
      set->_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;
    {\tt 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
```

```
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
\verb|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 BUTLDMODE != 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;
    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
\verb"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);
  7
#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
#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 % \left( 1\right) =\left( 1\right) \left( 1\right) 
#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 % \left( 1\right) =\left( 1\right) \left( 1\right) 
#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 % \left( 1\right) =\left( 1\right) \left( 1\right) 
// 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;
}
// Append the element of the GSet 'set' at the end of the GSet 'that'
// 'that' and 'set' can be empty
#if BUILDMODE != 0
inline
#endif
void _GSetAppendSet(GSet* 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
  // If there are elements in the set to append
  if (GSetNbElem(set) > 0) {
    // Declare an iterator on the set to append
    GSetIterForward iter = GSetIterForwardCreateStatic(set);
    // Loop on element to append
    do {
     // Get the data to append
      void* data = GSetIterGet(&iter);
      // Append the data to the end of the set
      GSetAppend(that, data);
    } while (GSetIterStep(&iter));
 }
// Append the element of the GSet 'that' at the end of the GSet 'set'
// Elements are kept sorted
// 'that' and 'set' can be empty
#if BUILDMODE != 0
inline
#endif
void _GSetAppendSortedSet(GSet* 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
  // If there are elements in the set to append
  if (GSetNbElem(set) > 0) {
    // Declare an iterator on the set to append
    GSetIterForward iter = GSetIterForwardCreateStatic(set);
    // Loop on element to append
    do {
      // Get the element to append
      GSetElem* elem = GSetIterGetElem(&iter);
      // Append the data of the element according to the sorting value
      GSetAddSort(that, elem->_data, elem->_sortVal);
    } while (GSetIterStep(&iter));
```

## 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=0
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
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);
    } 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);
    7
    ++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);
   }
  if (*(int*)GSetGetFirst(&set) != 0) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetGetFirst NOK");
    PBErrCatch(GSetErr);
  if (*(int*)GSetGetLast(&set) != 4) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetGetLast 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 (v < 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 UnitTestGSetMoveElem() {
```

```
int a[5] = \{1, 2, 3, 4, 5\};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
   GSetPush(&set, a + i);
  GSetMoveElem(&set, 3, 1);
  int checka[5] = {1, 4, 2, 3, 5};
  for (int i = 5; i--;) {
    if (checka[i] != *((int*)GSetGet(&set, i))) {
     GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetMoveElem NOK");
     PBErrCatch(GSetErr);
 GSetMoveElem(&set, 1, 3);
 int checkb[5] = \{1, 2, 3, 4, 5\};
  for (int i = 5; i--;) {
   if (checkb[i] != *((int*)GSetGet(&set, i))) {
     GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetMoveElem NOK");
     PBErrCatch(GSetErr);
   }
 }
 GSetMoveElem(&set, 0, 3);
  int checkc[5] = \{2, 3, 4, 1, 5\};
 for (int i = 5; i--;) {
    if (checkc[i] != *((int*)GSetGet(&set, i))) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetMoveElem NOK");
     PBErrCatch(GSetErr);
   }
 }
 GSetMoveElem(&set, 4, 1);
 int checkd[5] = \{2, 5, 3, 4, 1\};
  for (int i = 5; i--;) {
    if (checkd[i] != *((int*)GSetGet(&set, i))) {
     GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetMoveElem NOK");
     PBErrCatch(GSetErr);
   }
 }
 GSetFlush(&set);
 printf("UnitTestGSetMoveElem OK\n");
void UnitTestGSetMergeSet() {
 int a[5] = \{1, 2, 3, 4, 5\};
 GSet setA = GSetCreateStatic();
 for (int i = 3; i--;)
   GSetPush(&setA, a + i);
  GSet setB = GSetCreateStatic();
  for (int i = 2; i--;)
   GSetPush(&setB, a + i + 3);
  GSetAppendSet(&setA, &setB);
  for (int i = 5; i--;) {
    if (a[i] != *((int*)GSetGet(&setA, i))) {
     GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetAppendSet NOK");
     PBErrCatch(GSetErr);
   }
 GSetFlush(&setA);
 GSetFlush(&setB);
```

```
for (int i = 3; i--;)
    GSetAddSort(&setB, a + i, i);
  for (int i = 2; i--;)
    GSetAddSort(&setA, a + i + 3, i + 3);
  GSetAppendSortedSet(&setA, &setB);
  for (int i = 5; i--;) {
    if (a[i] != *((int*)GSetGet(&setA, i))) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetAppendSortedSet NOK");
      PBErrCatch(GSetErr);
   }
  GSetFlush(&setA);
  GSetFlush(&setB);
  printf("UnitTestGSetMergeSet OK\n");
void UnitTestGSetCount() {
  int a[5] = \{1, 2, 3, 4, 5\};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    for (int j = i; j--;)
     GSetPush(&set, a + i);
  for (int i = 5; i--;)
    if (GSetCount(&set, a + i) != i) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetCount NOK");
      PBErrCatch(GSetErr);
  GSetFlush(&set);
 printf("UnitTestGSetMergeSet OK\n");
void UnitTestGSet() {
  UnitTestGSetCreateFree();
  UnitTestGSetClone();
  UnitTestGSetFlush();
  UnitTestGSetPrint();
  UnitTestGSetPushPopAppendDrop();
  UnitTestGSetAddSort();
  UnitTestGSetInsertRemove();
  UnitTestGSetNbElemGet();
  UnitTestGSetGetIndex();
  UnitTestGSetSort();
  UnitTestGSetSplitMerge();
  UnitTestGSetSwitch();
  UnitTestGSetMoveElem();
  UnitTestGSetMergeSet();
  UnitTestGSetCount();
 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
UnitTestGSetPushPopAppendDrop OK
UnitTestGSetAddSort OK
UnitTestGSetInsertRemove OK
UnitTestGSetNbElemGet OK
UnitTestGSetGetIndex OK
UnitTestGSetSort OK
UnitTestGSetSplitMerge OK
{\tt UnitTestGSetSwitch\ OK}
UnitTestGSetMoveElem OK
UnitTestGSetMergeSet OK
{\tt UnitTestGSetMergeSet\ OK}
UnitTestGSet OK
UnitTestGSetIteratorForwardCreateFree OK
{\tt UnitTestGSetIteratorForwardClone} \ \ {\tt OK}
UnitTestGSetIteratorForwardReset OK
{\tt UnitTestGSetIteratorForwardStepGetGetElem\ OK}
UnitTestGSetIteratorForwardApply OK
UnitTestGSetIteratorForwardIsFirstIsLast OK
UnitTestGSetIteratorForwardSetGSet OK
UnitTestGSetIteratorForwardRemoveElem OK
UnitTestGSetIteratorForward OK
{\tt UnitTestGSetIteratorBackwardCreateFree\ OK}
UnitTestGSetIteratorBackwardClone OK
{\tt UnitTestGSetIteratorBackwardReset\ OK}
UnitTestGSetIteratorBackwardStepGetGetElem OK
UnitTestGSetIteratorBackwardApply OK
UnitTestGSetIteratorBackwardIsFirstIsLast OK
UnitTestGSetIteratorBackwardSetGSet OK
```

UnitTestGSetIteratorBackward OK

UnitTestAll OK