# GSet

P. Baillehache

March 16, 2019

## Contents

## Introduction

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

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

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

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

// Precision used when sorting a GSet
#define GSET_EPSILON 0.00001

// ================= 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
  long _nbElem;
} GSet;
```

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

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

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

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

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

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

// Return the number of element in the set
#if BUILDMODE != 0
inline
#endif
long _GSetNbElem(const GSet* const 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* const that, FILE* const stream,
  void(*printData)(const void* const data, FILE* const stream),
  const char* const sep);

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

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

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

// Function to get the data at the GSetElem
#if BUILDMODE != 0
inline
#endif
void* GSetElemData(const GSetElem* const that);

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

// Function to get the data at first position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetHead(const GSet* const that);

// Function to get the data at last position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetTail(const GSet* const that);

// Function to get the GSetElem at first position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
const GSetElem* _GSetHeadElem(const GSet* const that);

// Function to get the GSetElem at last position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
const GSetElem* _GSetTailElem(const GSet* const that);

// Function to get the element at the 'iElem'-th position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
const GSetElem* _GSetElement(const GSet* const that, const long 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
long _GSetGetIndexFirst(const GSet* const that, const void* const 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
long _GSetGetIndexLast(const GSet* const that, const void* const 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
const GSetElem* _GSetFirstElem(const GSet* const that,
  const void* const 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
const GSetElem* _GSetLastElem(const GSet* const that,
  const void* const data);

// Function to sort the element of the gset in increasing order of
// _sortVal
void _GSetSort(GSet* const 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* const that, GSet* const 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* const that, GSetElem* const 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* const that, const GSet* const 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* const that, const GSet* const set);

// Switch the 'iElem'-th and 'jElem'-th element of the set
#if BUILDMODE != 0
inline
#endif
void _GSetSwitch(GSet* const that, const long iElem, const long jElem);

// Return the number of (GSetElem._data=='data') in the GSet 'that'
long _GSetCount(const GSet* const that, const void* const data);

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

// Set the data of the GSetElem 'that' to 'd'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetData(GSetElem* const that, void* const d);

// Set the previous element of the GSetElem 'that' to 'e'
// Do not set the link back in 'e'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetPrev(GSetElem* const that, GSetElem* const e);

// Set the next element of the GSetElem 'that' to 'e'
// Do not set the link back in 'e'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetNext(GSetElem* const that, GSetElem* const e);

// Move the 'iElem'-th element to the 'pos' index in the GSet
void _GSetMoveElem(GSet* const that, const long iElem, const long pos);

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

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

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

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

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

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

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

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

// Step back the GSetIterBackward
// Return false if we couldn't step
// Return true else
#if BUILDMODE != 0
inline
#endif
bool GSetIterBackwardStepBack(GSetIterBackward* const 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* const that,
  void(*fun)(void* data, void* param), void* param);

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

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

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

```
inline
#endif
void GSetIterBackwardSetGSet(GSetIterBackward* const that,
  GSet* const set);

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

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

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

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

// Return the sort value of the element currently pointed to by the
// iterator
#if BUILDMODE != 0
inline
#endif
float GSetIterForwardGetSortVal(const GSetIterForward* const that);

// Return the sort value of the element currently pointed to by the
// iterator
#if BUILDMODE != 0
inline
#endif
float GSetIterBackwardGetSortVal(const GSetIterBackward* const that);

// Set the data of the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void GSetIterForwardSetData(const GSetIterForward* const that,
  void* data);

// Set the data of the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardSetData(const GSetIterBackward* const that,
  void* data);

// 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* const 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* const that);

// Return the sort value of GSetElem 'that'
#if BUILDMODE != 0
inline
#endif
float GSetElemGetSortVal(const GSetElem* const that);

// Return the next element of GSetElem 'that'
#if BUILDMODE != 0
inline
#endif
const GSetElem* GSetElemNext(const GSetElem* const that);

// Return the previous element of GSetElem 'that'
#if BUILDMODE != 0
inline
#endif
const GSetElem* GSetElemPrev(const GSetElem* const that);

// Shuffle the GSet 'that'
// The random generator must have been initialized before calling
// this function
// This function modifies the _sortVal of each elements in 'that'
// Use different algorithm according to the number of elements for
// speed performance
void GSetShuffle(GSet* const that);
void GSetShuffleA(GSet* const that);
void GSetShuffleB(GSet* const that);
void GSetShuffleC(GSet* const 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* const that)
  {return (GSetVecFloat*)GSetClone((GSet* const)that);}
inline VecFloat* _GSetVecFloatGet(const GSetVecFloat* const that,
  const long iElem)
  {return (VecFloat*)_GSetGet((GSet* const)that, iElem);}
inline VecFloat* _GSetVecFloatGetHead(const GSetVecFloat* const that)
  {return (VecFloat*)_GSetHead((const GSet* const)that);}
inline VecFloat* _GSetVecFloatGetTail(const GSetVecFloat* const that)
  {return (VecFloat*)_GSetTail((const GSet* const)that);}
inline VecFloat* _GSetVecFloatPop(GSetVecFloat* const that)
  {return (VecFloat*)_GSetPop((GSet* const)that);}
inline VecFloat* _GSetVecFloatDrop(GSetVecFloat* const that)
  {return (VecFloat*)_GSetDrop((GSet* const)that);}
inline VecFloat* _GSetVecFloatRemove(GSetVecFloat* const that,
  const long iElem)
  {return (VecFloat*)_GSetRemove((GSet* const)that, iElem);}
inline VecFloat* _GSetVecFloatRemoveElem(GSetVecFloat* const that,
  GSetElem** elem)
  {return (VecFloat*)_GSetRemoveElem((GSet* const)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(const GSetVecShort* const that)
  {return (GSetVecShort*)GSetClone((const GSet* const)that);}
inline VecShort* _GSetVecShortGet(const GSetVecShort* const that,
  const long iElem)
  {return (VecShort*)_GSetGet((const GSet* const)that, iElem);}
inline VecShort* _GSetVecShortGetHead(const GSetVecShort* const that)
  {return (VecShort*)_GSetHead((const GSet* const)that);}
inline VecShort* _GSetVecShortGetTail(const GSetVecShort* const that)
  {return (VecShort*)_GSetTail((const GSet* const)that);}
inline VecShort* _GSetVecShortPop(GSetVecShort* const that)
  {return (VecShort*)_GSetPop((GSet* const)that);}
inline VecShort* _GSetVecShortDrop(GSetVecShort* const that)
  {return (VecShort*)_GSetDrop((GSet* const)that);}
inline VecShort* _GSetVecShortRemove(GSetVecShort* const that,
  const long iElem)
  {return (VecShort*)_GSetRemove((GSet* const)that, iElem);}
inline VecShort* _GSetVecShortRemoveElem(GSetVecShort* const that,
  GSetElem** elem)
  {return (VecShort*)_GSetRemoveElem((GSet* const)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(const GSetBCurve* const that)
  {return (GSetBCurve*)GSetClone((const GSet* const)that);}
inline BCurve* _GSetBCurveGet(const GSetBCurve* const that,
  const long iElem)
  {return (BCurve*)_GSetGet((const GSet* const)that, iElem);}
inline BCurve* _GSetBCurveGetHead(const GSetBCurve* const that)
  {return (BCurve*)_GSetHead((const GSet* const)that);}
inline BCurve* _GSetBCurveGetTail(const GSetBCurve* const that)
  {return (BCurve*)_GSetTail((const GSet* const)that);}
inline BCurve* _GSetBCurvePop(GSetBCurve* const that)
  {return (BCurve*)_GSetPop((GSet* const)that);}
inline BCurve* _GSetBCurveDrop(GSetBCurve* const that)
  {return (BCurve*)_GSetDrop((GSet* const)that);}
inline BCurve* _GSetBCurveRemove(GSetBCurve* const that, const long iElem)
  {return (BCurve*)_GSetRemove((GSet* const)that, iElem);}
inline BCurve* _GSetBCurveRemoveElem(GSetBCurve* const that,
  GSetElem** elem)
  {return (BCurve*)_GSetRemoveElem((GSet* const)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(const GSetSCurve* const that)
  {return (GSetSCurve*)GSetClone((const GSet* const)that);}
inline SCurve* _GSetSCurveGet(const GSetSCurve* const that,
  const long iElem)
  {return (SCurve*)_GSetGet((const GSet* const)that, iElem);}
inline SCurve* _GSetSCurveGetHead(const GSetSCurve* const that)
  {return (SCurve*)_GSetHead((const GSet* const)that);}
inline SCurve* _GSetSCurveGetTail(const GSetSCurve* const that)
  {return (SCurve*)_GSetTail((const GSet* const)that);}
inline SCurve* _GSetSCurvePop(GSetSCurve* const that)
  {return (SCurve*)_GSetPop((GSet* const)that);}
inline SCurve* _GSetSCurveDrop(GSetSCurve* const that)
  {return (SCurve*)_GSetDrop((GSet* const)that);}
inline SCurve* _GSetSCurveRemove(GSetSCurve* const that,
  const long iElem)
  {return (SCurve*)_GSetRemove((GSet* const)that, iElem);}
inline SCurve* _GSetSCurveRemoveElem(GSetSCurve* const that,
  GSetElem** elem)
  {return (SCurve*)_GSetRemoveElem((GSet* const)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(const GSetShapoid* const that)
  {return (GSetShapoid*)GSetClone((const GSet* const)that);}
inline Shapoid* _GSetShapoidGet(const GSetShapoid* const that,
  const long iElem)
  {return (Shapoid*)_GSetGet((const GSet* const)that, iElem);}
inline Shapoid* _GSetShapoidGetHead(const GSetShapoid* const that)
  {return (Shapoid*)_GSetHead((const GSet* const)that);}
inline Shapoid* _GSetShapoidGetTail(const GSetShapoid* const that)
  {return (Shapoid*)_GSetTail((const GSet* const)that);}
inline Shapoid* _GSetShapoidPop(GSetShapoid* const that)
  {return (Shapoid*)_GSetPop((GSet* const)that);}
inline Shapoid* _GSetShapoidDrop(GSetShapoid* const that)
  {return (Shapoid*)_GSetDrop((GSet* const)that);}
inline Shapoid* _GSetShapoidRemove(GSetShapoid* const that,
  const long iElem)
  {return (Shapoid*)_GSetRemove((GSet* const)that, iElem);}
inline Shapoid* _GSetShapoidRemoveElem(GSetShapoid* const that,
  GSetElem** elem)
  {return (Shapoid*)_GSetRemoveElem((GSet* const)that, elem);}

#ifndef KnapSackPod
  typedef struct KnapSackPod KnapSackPod;
#endif
typedef struct GSetKnapSackPod {GSet _set;} GSetKnapSackPod;
#define GSetKnapSackPodCreate() ((GSetKnapSackPod*)GSetCreate())
inline GSetKnapSackPod GSetKnapSackPodCreateStatic(void)
  {GSetKnapSackPod ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetKnapSackPod* GSetKnapSackPodClone(
  const GSetKnapSackPod* const that)
  {return (GSetKnapSackPod*)GSetClone((const GSet* const)that);}
inline KnapSackPod* _GSetKnapSackPodGet(
  const GSetKnapSackPod* const that, const long iElem)
  {return (KnapSackPod*)_GSetGet((const GSet* const)that, iElem);}
inline KnapSackPod* _GSetKnapSackPodGetHead(
  const GSetKnapSackPod* const that)
  {return (KnapSackPod*)_GSetHead((const GSet* const)that);}
inline KnapSackPod* _GSetKnapSackPodGetTail(
  const GSetKnapSackPod* const that)
  {return (KnapSackPod*)_GSetTail((const GSet* const)that);}
inline KnapSackPod* _GSetKnapSackPodPop(GSetKnapSackPod* const that)
  {return (KnapSackPod*)_GSetPop((GSet* const)that);}
inline KnapSackPod* _GSetKnapSackPodDrop(GSetKnapSackPod* const that)
  {return (KnapSackPod*)_GSetDrop((GSet* const)that);}
inline KnapSackPod* _GSetKnapSackPodRemove(
  GSetKnapSackPod* that, const long iElem)
  {return (KnapSackPod*)_GSetRemove((GSet* const)that, iElem);}
inline KnapSackPod* _GSetKnapSackPodRemoveElem(
  GSetKnapSackPod* const that, GSetElem** elem)
  {return (KnapSackPod*)_GSetRemoveElem((GSet* const)that, elem);}

#ifndef PBPhysParticle
  typedef struct PBPhysParticle PBPhysParticle;
#endif
typedef struct GSetPBPhysParticle {GSet _set;} GSetPBPhysParticle;
#define GSetPBPhysParticleCreate() ((GSetPBPhysParticle*)GSetCreate())
inline GSetPBPhysParticle GSetPBPhysParticleCreateStatic(void)
  {GSetPBPhysParticle ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetPBPhysParticle* GSetPBPhysParticleClone(
  const GSetPBPhysParticle* const that)
  {return (GSetPBPhysParticle*)GSetClone((const GSet* const)that);}
```

```
inline PBPhysParticle* _GSetPBPhysParticleGet(
  const GSetPBPhysParticle* const that, const long iElem)
  {return (PBPhysParticle*)_GSetGet((const GSet* const)that, iElem);}
inline PBPhysParticle* _GSetPBPhysParticleGetHead(
  const GSetPBPhysParticle* const that)
  {return (PBPhysParticle*)_GSetHead((const GSet* const)that);}
inline PBPhysParticle* _GSetPBPhysParticleGetTail(
  const GSetPBPhysParticle* const that)
  {return (PBPhysParticle*)_GSetTail((const GSet* const)that);}
inline PBPhysParticle* _GSetPBPhysParticlePop(
  GSetPBPhysParticle* const that)
  {return (PBPhysParticle*)_GSetPop((GSet* const)that);}
inline PBPhysParticle* _GSetPBPhysParticleDrop(
  GSetPBPhysParticle* const that)
  {return (PBPhysParticle*)_GSetDrop((GSet* const)that);}
inline PBPhysParticle* _GSetPBPhysParticleRemove(
  GSetPBPhysParticle* const that, const long iElem)
  {return (PBPhysParticle*)_GSetRemove((GSet* const)that, iElem);}
inline PBPhysParticle* _GSetPBPhysParticleRemoveElem(
  GSetPBPhysParticle* const that, GSetElem** elem)
  {return (PBPhysParticle*)_GSetRemoveElem((GSet* const)that, elem);}

#ifndef GenTree
  typedef struct GenTree GenTree;
#endif
typedef struct GSetGenTree {GSet _set;} GSetGenTree;
#define GSetGenTreeCreate() ((GSetGenTree*)GSetCreate())
inline GSetGenTree GSetGenTreeCreateStatic(void)
  {GSetGenTree ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetGenTree* GSetGenTreeClone(const GSetGenTree* const that)
  {return (GSetGenTree*)GSetClone((const GSet* const)that);}
inline GenTree* _GSetGenTreeGet(const GSetGenTree* const that, const long iElem)
  {return (GenTree*)_GSetGet((const GSet* const)that, iElem);}
inline GenTree* _GSetGenTreeGetHead(const GSetGenTree* const that)
  {return (GenTree*)_GSetHead((const GSet* const)that);}
inline GenTree* _GSetGenTreeGetTail(const GSetGenTree* const that)
  {return (GenTree*)_GSetTail((const GSet* const)that);}
inline GenTree* _GSetGenTreePop(GSetGenTree* const that)
  {return (GenTree*)_GSetPop((GSet* const)that);}
inline GenTree* _GSetGenTreeDrop(GSetGenTree* const that)
  {return (GenTree*)_GSetDrop((GSet* const)that);}
inline GenTree* _GSetGenTreeRemove(GSetGenTree* const that, const long iElem)
  {return (GenTree*)_GSetRemove((GSet* const)that, iElem);}
inline GenTree* _GSetGenTreeRemoveElem(GSetGenTree* const that,
  GSetElem** elem)
  {return (GenTree*)_GSetRemoveElem((GSet* const)that, elem);}

typedef struct GSetStr {GSet _set;} GSetStr;
#define GSetStrCreate() ((GSetStr*)GSetCreate())
inline GSetStr GSetStrCreateStatic(void)
  {GSetStr ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetStr* GSetStrClone(const GSetStr* const that)
  {return (GSetStr*)GSetClone((const GSet* const)that);}
inline char* _GSetStrGet(const GSetStr* const that, const long iElem)
  {return (char*)_GSetGet((const GSet* const)that, iElem);}
inline char* _GSetStrGetHead(const GSetStr* const that)
  {return (char*)_GSetHead((const GSet* const)that);}
inline char* _GSetStrGetTail(const GSetStr* const that)
  {return (char*)_GSetTail((const GSet* const)that);}
inline char* _GSetStrPop(GSetStr* const that)
  {return (char*)_GSetPop((GSet* const)that);}
inline char* _GSetStrDrop(GSetStr* const that)
```

```
  {return (char*)_GSetDrop((GSet* const)that);}
inline char* _GSetStrRemove(GSetStr* const that, const long iElem)
  {return (char*)_GSetRemove((GSet* const)that, iElem);}
inline char* _GSetStrRemoveElem(GSetStr* const that, GSetElem** elem)
  {return (char*)_GSetRemoveElem((GSet* const)that, elem);}

#ifndef GenTreeStr
  typedef struct GenTreeStr GenTreeStr;
#endif
typedef struct GSetGenTreeStr {GSet _set;} GSetGenTreeStr;
#define GSetGenTreeStrCreate() ((GSetGenTreeStr*)GSetCreate())
inline GSetGenTreeStr GSetGenTreeStrCreateStatic(void)
  {GSetGenTreeStr ret = {._set=GSetCreateStatic()}; return ret;}
inline GSetGenTreeStr* GSetGenTreeStrClone(const GSetGenTreeStr* const that)
  {return (GSetGenTreeStr*)GSetClone((const GSet* const)that);}
inline GenTreeStr* _GSetGenTreeStrGet(const GSetGenTreeStr* const that,
  const long iElem)
  {return (GenTreeStr*)_GSetGet((const GSet* const)that, iElem);}
inline GenTreeStr* _GSetGenTreeStrGetHead(const GSetGenTreeStr* const that)
  {return (GenTreeStr*)_GSetHead((const GSet* const)that);}
inline GenTreeStr* _GSetGenTreeStrGetTail(const GSetGenTreeStr* const that)
  {return (GenTreeStr*)_GSetTail((const GSet* const)that);}
inline GenTreeStr* _GSetGenTreeStrPop(GSetGenTreeStr* const that)
  {return (GenTreeStr*)_GSetPop((GSet* const)that);}
inline GenTreeStr* _GSetGenTreeStrDrop(GSetGenTreeStr* const that)
  {return (GenTreeStr*)_GSetDrop((GSet* const)that);}
inline GenTreeStr* _GSetGenTreeStrRemove(GSetGenTreeStr* const that,
  const long iElem)
  {return (GenTreeStr*)_GSetRemove((GSet* const)that, iElem);}
inline GenTreeStr* _GSetGenTreeStrRemoveElem(
  GSetGenTreeStr* const that, GSetElem** elem)
  {return (GenTreeStr*)_GSetRemoveElem((GSet* const)that, elem);}

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

#define GSetFree(Set) _Generic(Set, \
  GSet**: _GSetFree, \
  GSetVecFloat**: _GSetFree, \
  GSetVecShort**: _GSetFree, \
  GSetBCurve**: _GSetFree, \
  GSetSCurve**: _GSetFree, \
  GSetShapoid**: _GSetFree, \
  GSetKnapSackPod**: _GSetFree, \
  GSetPBPhysParticle**: _GSetFree, \
  GSetGenTree**: _GSetFree, \
  GSetStr**: _GSetFree, \
  GSetGenTreeStr**: _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), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetPush, \
    const KnapSackPod*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetPush, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetAddSort, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetAddSort, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetAddSort, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetAddSort, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetInsert, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetInsert, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetInsert, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetInsert, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetAppend, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetAppend, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetAppend, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetAppend, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _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, \
    VecFloat2D*: _GSetRemoveFirst, \
    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), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetRemoveFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetRemoveLast, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetRemoveAll, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetRemoveAll, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetRemoveAll, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetRemoveAll, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetRemoveAll, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))

#define GSetGetIndexFirst(Set, Data) _Generic(Set, \
  GSet*: _Generic(Data, \
    default: _GSetGetIndexFirst), \
  const GSet*: _Generic(Data, \
    default: _GSetGetIndexFirst), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetGetIndexFirst, \
    VecFloat2D*: _GSetGetIndexFirst, \
    VecFloat3D*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetGetIndexFirst, \
    VecFloat2D*: _GSetGetIndexFirst, \
    VecFloat3D*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetGetIndexFirst, \
    VecShort2D*: _GSetGetIndexFirst, \
    VecShort3D*: _GSetGetIndexFirst, \
    VecShort4D*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetGetIndexFirst, \
    VecShort2D*: _GSetGetIndexFirst, \
    VecShort3D*: _GSetGetIndexFirst, \
    VecShort4D*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetGetIndexFirst, \
    Facoid*: _GSetGetIndexFirst, \
    Pyramidoid*: _GSetGetIndexFirst, \
    Spheroid*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetGetIndexFirst, \
    Facoid*: _GSetGetIndexFirst, \
    Pyramidoid*: _GSetGetIndexFirst, \
    Spheroid*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetStr*: _Generic(Data, \
    char*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetGetIndexFirst, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
```

```
#define GSetGetIndexLast(Set, Data) _Generic(Set, \
  GSet*: _Generic(Data, \
    default: _GSetGetIndexLast), \
  const GSet*: _Generic(Data, \
    default: _GSetGetIndexLast), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetGetIndexLast, \
    VecFloat2D*: _GSetGetIndexLast, \
    VecFloat3D*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetGetIndexLast, \
    VecFloat2D*: _GSetGetIndexLast, \
    VecFloat3D*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetGetIndexLast, \
    VecShort2D*: _GSetGetIndexLast, \
    VecShort3D*: _GSetGetIndexLast, \
    VecShort4D*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetGetIndexLast, \
    VecShort2D*: _GSetGetIndexLast, \
    VecShort3D*: _GSetGetIndexLast, \
    VecShort4D*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetGetIndexLast, \
    Facoid*: _GSetGetIndexLast, \
    Pyramidoid*: _GSetGetIndexLast, \
    Spheroid*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetGetIndexLast, \
    Facoid*: _GSetGetIndexLast, \
    Pyramidoid*: _GSetGetIndexLast, \
    Spheroid*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetStr*: _Generic(Data, \
    char*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetGetIndexLast, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))

#define GSetFirstElem(Set, Data) _Generic(Set, \
  GSet*: _Generic(Data, \
    default: _GSetFirstElem), \
  const GSet*: _Generic(Data, \
    default: _GSetFirstElem), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetFirstElem, \
    VecFloat2D*: _GSetFirstElem, \
    VecFloat3D*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetFirstElem, \
    VecFloat2D*: _GSetFirstElem, \
    VecFloat3D*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetFirstElem, \
    VecShort2D*: _GSetFirstElem, \
    VecShort3D*: _GSetFirstElem, \
    VecShort4D*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetFirstElem, \
    VecShort2D*: _GSetFirstElem, \
    VecShort3D*: _GSetFirstElem, \
    VecShort4D*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetFirstElem, \
    Facoid*: _GSetFirstElem, \
    Pyramidoid*: _GSetFirstElem, \
    Spheroid*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetFirstElem, \
    Facoid*: _GSetFirstElem, \
    Pyramidoid*: _GSetFirstElem, \
    Spheroid*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetStr*: _Generic(Data, \
    char*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetFirstElem, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))

#define GSetLastElem(Set, Data) _Generic(Set, \
  GSet*: _Generic(Data, \
    default: _GSetLastElem), \
  const GSet*: _Generic(Data, \
    default: _GSetLastElem), \
  GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetLastElem, \
    VecFloat2D*: _GSetLastElem, \
    VecFloat3D*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecFloat*: _Generic(Data, \
    VecFloat*: _GSetLastElem, \
    VecFloat2D*: _GSetLastElem, \
    VecFloat3D*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetLastElem, \
    VecShort2D*: _GSetLastElem, \
    VecShort3D*: _GSetLastElem, \
    VecShort4D*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetVecShort*: _Generic(Data, \
    VecShort*: _GSetLastElem, \
    VecShort2D*: _GSetLastElem, \
    VecShort3D*: _GSetLastElem, \
    VecShort4D*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetBCurve*: _Generic(Data, \
    BCurve*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetSCurve*: _Generic(Data, \
    SCurve*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetLastElem, \
    Facoid*: _GSetLastElem, \
    Pyramidoid*: _GSetLastElem, \
    Spheroid*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetShapoid*: _Generic(Data, \
    Shapoid*: _GSetLastElem, \
    Facoid*: _GSetLastElem, \
    Pyramidoid*: _GSetLastElem, \
    Spheroid*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetKnapSackPod*: _Generic(Data, \
    KnapSackPod*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetPBPhysParticle*: _Generic(Data, \
    PBPhysParticle*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTree*: _Generic(Data, \
    GenTree*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(Data, \
    char*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetStr*: _Generic(Data, \
    char*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  const GSetGenTreeStr*: _Generic(Data, \
    GenTreeStr*: _GSetLastElem, \
    default: PBErrInvalidPolymorphism), \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), (void*)(Data))
```

```
#define GSetPrint(Set, Stream, Fun, Sep) _Generic(Set, \
  GSet*: _GSetPrint, \
  const GSet*: _GSetPrint, \
  GSetVecFloat*: _GSetPrint, \
  const GSetVecFloat*: _GSetPrint, \
  GSetVecShort*: _GSetPrint, \
  const GSetVecShort*: _GSetPrint, \
  GSetBCurve*: _GSetPrint, \
  const GSetBCurve*: _GSetPrint, \
  GSetSCurve*: _GSetPrint, \
  const GSetSCurve*: _GSetPrint, \
  GSetShapoid*: _GSetPrint, \
  const GSetShapoid*: _GSetPrint, \
  GSetKnapSackPod*: _GSetPrint, \
  const GSetKnapSackPod*: _GSetPrint, \
  GSetPBPhysParticle*: _GSetPrint, \
  const GSetPBPhysParticle*: _GSetPrint, \
  GSetGenTree*: _GSetPrint, \
  const GSetGenTree*: _GSetPrint, \
  GSetStr*: _GSetPrint, \
  const GSetStr*: _GSetPrint, \
  GSetGenTreeStr*: _GSetPrint, \
  const GSetGenTreeStr*: _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, \
  GSetKnapSackPod*: _GSetFlush, \
  GSetPBPhysParticle*: _GSetFlush, \
  GSetGenTree*: _GSetFlush, \
  GSetStr*: _GSetFlush, \
  GSetGenTreeStr*: _GSetFlush, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))

#define GSetNbElem(Set) _Generic(Set, \
  GSet*: _GSetNbElem, \
  const GSet*: _GSetNbElem, \
  GSetVecFloat*: _GSetNbElem, \
  const GSetVecFloat*: _GSetNbElem, \
  GSetVecShort*: _GSetNbElem, \
  const GSetVecShort*: _GSetNbElem, \
  GSetBCurve*: _GSetNbElem, \
  const GSetBCurve*: _GSetNbElem, \
  GSetSCurve*: _GSetNbElem, \
  const GSetSCurve*: _GSetNbElem, \
  GSetShapoid*: _GSetNbElem, \
  const GSetShapoid*: _GSetNbElem, \
  GSetKnapSackPod*: _GSetNbElem, \
  const GSetKnapSackPod*: _GSetNbElem, \
  GSetPBPhysParticle*: _GSetNbElem, \
  const GSetPBPhysParticle*: _GSetNbElem, \
  GSetGenTree*: _GSetNbElem, \
  const GSetGenTree*: _GSetNbElem, \
  GSetStr*: _GSetNbElem, \
  const GSetStr*: _GSetNbElem, \
  GSetGenTreeStr*: _GSetNbElem, \
  const GSetGenTreeStr*: _GSetNbElem, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))

#define GSetPop(Set) _Generic(Set, \
  GSet*: _GSetPop, \
  GSetVecFloat*: _GSetVecFloatPop, \
  GSetVecShort*: _GSetVecShortPop, \
  GSetBCurve*: _GSetBCurvePop, \
  GSetSCurve*: _GSetSCurvePop, \
  GSetShapoid*: _GSetShapoidPop, \
  GSetKnapSackPod*: _GSetKnapSackPodPop, \
  GSetPBPhysParticle*: _GSetPBPhysParticlePop, \
  GSetGenTree*: _GSetGenTreePop, \
  GSetStr*: _GSetStrPop, \
  GSetGenTreeStr*: _GSetGenTreeStrPop, \
  default: PBErrInvalidPolymorphism)(Set)

#define GSetDrop(Set) _Generic(Set, \
  GSet*: _GSetDrop, \
  GSetVecFloat*: _GSetVecFloatDrop, \
  GSetVecShort*: _GSetVecShortDrop, \
  GSetBCurve*: _GSetBCurveDrop, \
  GSetSCurve*: _GSetSCurveDrop, \
  GSetShapoid*: _GSetShapoidDrop, \
  GSetKnapSackPod*: _GSetKnapSackPodDrop, \
  GSetPBPhysParticle*: _GSetPBPhysParticleDrop, \
  GSetGenTree*: _GSetGenTreeDrop, \
  GSetStr*: _GSetStrDrop, \
  GSetGenTreeStr*: _GSetGenTreeStrDrop, \
  default: PBErrInvalidPolymorphism)(Set)

#define GSetRemove(Set, Pos) _Generic(Set, \
  GSet*: _GSetRemove, \
  GSetVecFloat*: _GSetVecFloatRemove, \
  GSetVecShort*: _GSetVecShortRemove, \
  GSetBCurve*: _GSetBCurveRemove, \
  GSetSCurve*: _GSetSCurveRemove, \
  GSetShapoid*: _GSetShapoidRemove, \
  GSetKnapSackPod*: _GSetKnapSackPodRemove, \
  GSetPBPhysParticle*: _GSetPBPhysParticleRemove, \
  GSetGenTree*: _GSetGenTreeRemove, \
  GSetStr*: _GSetStrRemove, \
  GSetGenTreeStr*: _GSetGenTreeStrRemove, \
  default: PBErrInvalidPolymorphism)(Set, Pos)

#define GSetRemoveElem(Set, Elem) _Generic(Set, \
  GSet*: _GSetRemoveElem, \
  GSetVecFloat*: _GSetVecFloatRemoveElem, \
  GSetVecShort*: _GSetVecShortRemoveElem, \
  GSetBCurve*: _GSetBCurveRemoveElem, \
  GSetSCurve*: _GSetSCurveRemoveElem, \
  GSetShapoid*: _GSetShapoidRemoveElem, \
  GSetKnapSackPod*: _GSetKnapSackPodRemoveElem, \
  GSetPBPhysParticle*: _GSetPBPhysParticleRemoveElem, \
  GSetGenTree*: _GSetGenTreeRemoveElem, \
  GSetStr*: _GSetStrRemoveElem, \
  GSetGenTreeStr*: _GSetGenTreeStrRemoveElem, \
  default: PBErrInvalidPolymorphism)(Set, Elem)

#define GSetGet(Set, Pos) _Generic(Set, \
  GSet*: _GSetGet, \
  const GSet*: _GSetGet, \
  GSetVecFloat*: _GSetVecFloatGet, \
  const GSetVecFloat*: _GSetVecFloatGet, \
  GSetVecShort*: _GSetVecShortGet, \
  const GSetVecShort*: _GSetVecShortGet, \
  GSetBCurve*: _GSetBCurveGet, \
  const GSetBCurve*: _GSetBCurveGet, \
  GSetSCurve*: _GSetSCurveGet, \
  const GSetSCurve*: _GSetSCurveGet, \
  GSetShapoid*: _GSetShapoidGet, \
  const GSetShapoid*: _GSetShapoidGet, \
  GSetKnapSackPod*: _GSetKnapSackPodGet, \
  const GSetKnapSackPod*: _GSetKnapSackPodGet, \
  GSetPBPhysParticle*: _GSetPBPhysParticleGet, \
  const GSetPBPhysParticle*: _GSetPBPhysParticleGet, \
  GSetGenTree*: _GSetGenTreeGet, \
  const GSetGenTree*: _GSetGenTreeGet, \
  GSetStr*: _GSetStrGet, \
  const GSetStr*: _GSetStrGet, \
  GSetGenTreeStr*: _GSetGenTreeStrGet, \
  const GSetGenTreeStr*: _GSetGenTreeStrGet, \
  default: PBErrInvalidPolymorphism)(Set, Pos)

#define GSetHead(Set) _Generic(Set, \
  GSet*: _GSetHead, \
  const GSet*: _GSetHead, \
  GSetVecFloat*: _GSetVecFloatGetHead, \
  const GSetVecFloat*: _GSetVecFloatGetHead, \
  GSetVecShort*: _GSetVecShortGetHead, \
  const GSetVecShort*: _GSetVecShortGetHead, \
  GSetBCurve*: _GSetBCurveGetHead, \
  const GSetBCurve*: _GSetBCurveGetHead, \
  GSetSCurve*: _GSetSCurveGetHead, \
  const GSetSCurve*: _GSetSCurveGetHead, \
  GSetShapoid*: _GSetShapoidGetHead, \
  const GSetShapoid*: _GSetShapoidGetHead, \
  GSetKnapSackPod*: _GSetKnapSackPodGetHead, \
  const GSetKnapSackPod*: _GSetKnapSackPodGetHead, \
  GSetPBPhysParticle*: _GSetPBPhysParticleGetHead, \
  const GSetPBPhysParticle*: _GSetPBPhysParticleGetHead, \
  GSetGenTree*: _GSetGenTreeGetHead, \
  const GSetGenTree*: _GSetGenTreeGetHead, \
  GSetStr*: _GSetStrGetHead, \
  const GSetStr*: _GSetStrGetHead, \
  GSetGenTreeStr*: _GSetGenTreeStrGetHead, \
  const GSetGenTreeStr*: _GSetGenTreeStrGetHead, \
  default: PBErrInvalidPolymorphism)(Set)

#define GSetTail(Set) _Generic(Set, \
  GSet*: _GSetTail, \
  const GSet*: _GSetTail, \
  GSetVecFloat*: _GSetVecFloatGetTail, \
  const GSetVecFloat*: _GSetVecFloatGetTail, \
  GSetVecShort*: _GSetVecShortGetTail, \
  const GSetVecShort*: _GSetVecShortGetTail, \
  GSetBCurve*: _GSetBCurveGetTail, \
  const GSetBCurve*: _GSetBCurveGetTail, \
  GSetSCurve*: _GSetSCurveGetTail, \
  const GSetSCurve*: _GSetSCurveGetTail, \
  GSetShapoid*: _GSetShapoidGetTail, \
  const GSetShapoid*: _GSetShapoidGetTail, \
  GSetKnapSackPod*: _GSetKnapSackPodGetTail, \
  const GSetKnapSackPod*: _GSetKnapSackPodGetTail, \
  GSetPBPhysParticle*: _GSetPBPhysParticleGetTail, \
  const GSetPBPhysParticle*: _GSetPBPhysParticleGetTail, \
  GSetGenTree*: _GSetGenTreeGetTail, \
  const GSetGenTree*: _GSetGenTreeGetTail, \
  GSetStr*: _GSetStrGetTail, \
  const GSetStr*: _GSetStrGetTail, \
  GSetGenTreeStr*: _GSetGenTreeStrGetTail, \
  const GSetGenTreeStr*: _GSetGenTreeStrGetTail, \
  default: PBErrInvalidPolymorphism)(Set)

#define GSetHeadElem(Set) _Generic(Set, \
  GSet*: _GSetHeadElem, \
  const GSet*: _GSetHeadElem, \
  GSetVecFloat*: _GSetHeadElem, \
  const GSetVecFloat*: _GSetHeadElem, \
  GSetVecShort*: _GSetHeadElem, \
  const GSetVecShort*: _GSetHeadElem, \
  GSetBCurve*: _GSetHeadElem, \
  const GSetBCurve*: _GSetHeadElem, \
  GSetSCurve*: _GSetHeadElem, \
  const GSetSCurve*: _GSetHeadElem, \
  GSetShapoid*: _GSetHeadElem, \
  const GSetShapoid*: _GSetHeadElem, \
  GSetKnapSackPod*: _GSetHeadElem, \
  const GSetKnapSackPod*: _GSetHeadElem, \
  GSetPBPhysParticle*: _GSetHeadElem, \
  const GSetPBPhysParticle*: _GSetHeadElem, \
  GSetGenTree*: _GSetHeadElem, \
  const GSetGenTree*: _GSetHeadElem, \
  GSetStr*: _GSetHeadElem, \
  const GSetStr*: _GSetHeadElem, \
  GSetGenTreeStr*: _GSetHeadElem, \
  const GSetGenTreeStr*: _GSetHeadElem, \
  default: PBErrInvalidPolymorphism)((const GSet*)Set)

#define GSetTailElem(Set) _Generic(Set, \
  GSet*: _GSetTailElem, \
  const GSet*: _GSetTailElem, \
  GSetVecFloat*: _GSetTailElem, \
  const GSetVecFloat*: _GSetTailElem, \
  GSetVecShort*: _GSetTailElem, \
  const GSetVecShort*: _GSetTailElem, \
  GSetBCurve*: _GSetTailElem, \
  const GSetBCurve*: _GSetTailElem, \
  GSetSCurve*: _GSetTailElem, \
  const GSetSCurve*: _GSetTailElem, \
  GSetShapoid*: _GSetTailElem, \
  const GSetShapoid*: _GSetTailElem, \
  GSetKnapSackPod*: _GSetTailElem, \
  const GSetKnapSackPod*: _GSetTailElem, \
  GSetPBPhysParticle*: _GSetTailElem, \
  const GSetPBPhysParticle*: _GSetTailElem, \
  GSetGenTree*: _GSetTailElem, \
  const GSetGenTree*: _GSetTailElem, \
  GSetStr*: _GSetTailElem, \
  const GSetStr*: _GSetTailElem, \
  GSetGenTreeStr*: _GSetTailElem, \
  const GSetGenTreeStr*: _GSetTailElem, \
  default: PBErrInvalidPolymorphism)((const GSet*)Set)

#define GSetElement(Set, Pos) _Generic(Set, \
  GSet*: _GSetElement, \
  const GSet*: _GSetElement, \
  GSetVecFloat*: _GSetElement, \
  const GSetVecFloat*: _GSetElement, \
  GSetVecShort*: _GSetElement, \
  const GSetVecShort*: _GSetElement, \
  GSetBCurve*: _GSetElement, \
  const GSetBCurve*: _GSetElement, \
  GSetSCurve*: _GSetElement, \
  const GSetSCurve*: _GSetElement, \
  GSetShapoid*: _GSetElement, \
  const GSetShapoid*: _GSetElement, \
  GSetKnapSackPod*: _GSetElement, \
  const GSetKnapSackPod*: _GSetElement, \
  GSetPBPhysParticle*: _GSetElement, \
  const GSetPBPhysParticle*: _GSetElement, \
  GSetGenTree*: _GSetElement, \
  const GSetGenTree*: _GSetElement, \
  GSetStr*: _GSetElement, \
  const GSetStr*: _GSetElement, \
  GSetGenTreeStr*: _GSetElement, \
  const GSetGenTreeStr*: _GSetElement, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), Pos)

#define GSetSort(Set) _Generic(Set, \
  GSet*: _GSetSort, \
  GSetVecFloat*: _GSetSort, \
  GSetVecShort*: _GSetSort, \
  GSetBCurve*: _GSetSort, \
  GSetSCurve*: _GSetSort, \
  GSetShapoid*: _GSetSort, \
  GSetKnapSackPod*: _GSetSort, \
  GSetPBPhysParticle*: _GSetSort, \
  GSetGenTree*: _GSetSort, \
  GSetStr*: _GSetSort, \
  GSetGenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(MergedSet, \
    GSetKnapSackPod*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(MergedSet, \
    GSetPBPhysParticle*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(MergedSet, \
    GSetGenTree*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(MergedSet, \
    GSetStr*: _GSetMerge, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(MergedSet, \
    GSetGenTreeStr*: _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, \
  GSetKnapSackPod*: _GSetSplit, \
  GSetPBPhysParticle*: _GSetSplit, \
  GSetGenTree*: _GSetSplit, \
  GSetStr*: _GSetSplit, \
  GSetGenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(AppendSet, \
    GSetKnapSackPod*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(AppendSet, \
    GSetPBPhysParticle*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(AppendSet, \
    GSetGenTree*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(AppendSet, \
    GSetStr*: _GSetAppendSet, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(AppendSet, \
    GSetGenTreeStr*: _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), \
  GSetKnapSackPod*: _Generic(AppendSet, \
    GSetKnapSackPod*: _GSetAppendSortedSet, \
    default: PBErrInvalidPolymorphism), \
  GSetPBPhysParticle*: _Generic(AppendSet, \
    GSetPBPhysParticle*: _GSetAppendSortedSet, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTree*: _Generic(AppendSet, \
    GSetGenTree*: _GSetAppendSortedSet, \
    default: PBErrInvalidPolymorphism), \
  GSetStr*: _Generic(AppendSet, \
    GSetStr*: _GSetAppendSortedSet, \
    default: PBErrInvalidPolymorphism), \
  GSetGenTreeStr*: _Generic(AppendSet, \
    GSetGenTreeStr*: _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, \
  GSetKnapSackPod*: _GSetSwitch, \
  GSetPBPhysParticle*: _GSetSwitch, \
  GSetGenTree*: _GSetSwitch, \
  GSetStr*: _GSetSwitch, \
  GSetGenTreeStr*: _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, \
  GSetKnapSackPod*: _GSetMoveElem, \
  GSetPBPhysParticle*: _GSetMoveElem, \
  GSetGenTree*: _GSetMoveElem, \
  GSetStr*: _GSetMoveElem, \
  GSetGenTreeStr*: _GSetMoveElem, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), From, To)

#define GSetCount(Set, Data) _Generic(Set, \
  GSet*: _GSetCount, \
  const GSet*: _GSetCount, \
  GSetVecFloat*: _GSetCount, \
  const GSetVecFloat*: _GSetCount, \
  GSetVecShort*: _GSetCount, \
  const GSetVecShort*: _GSetCount, \
  GSetBCurve*: _GSetCount, \
  const GSetBCurve*: _GSetCount, \
  GSetSCurve*: _GSetCount, \
  const GSetSCurve*: _GSetCount, \
  GSetShapoid*: _GSetCount, \
  const GSetShapoid*: _GSetCount, \
  GSetKnapSackPod*: _GSetCount, \
  const GSetKnapSackPod*: _GSetCount, \
  GSetPBPhysParticle*: _GSetCount, \
  const GSetPBPhysParticle*: _GSetCount, \
  GSetGenTree*: _GSetCount, \
  const GSetGenTree*: _GSetCount, \
  GSetStr*: _GSetCount, \
  const GSetStr*: _GSetCount, \
  GSetGenTreeStr*: _GSetCount, \
  const GSetGenTreeStr*: _GSetCount, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set), Data)

#define GSetGetBounds(Set) _Generic(Set, \
  GSetVecFloat*: _GSetVecFloatGetBounds, \
  const GSetVecFloat*: _GSetVecFloatGetBounds, \
  default: PBErrInvalidPolymorphism)(Set)

#define GSetIterForwardCreate(Set) _Generic(Set, \
  GSet*: _GSetIterForwardCreate, \
  const GSet*: _GSetIterForwardCreate, \
  GSetVecFloat*: _GSetIterForwardCreate, \
  const GSetVecFloat*: _GSetIterForwardCreate, \
  GSetVecShort*: _GSetIterForwardCreate, \
  const GSetVecShort*: _GSetIterForwardCreate, \
  GSetBCurve*: _GSetIterForwardCreate, \
  const GSetBCurve*: _GSetIterForwardCreate, \
  GSetSCurve*: _GSetIterForwardCreate, \
  const GSetSCurve*: _GSetIterForwardCreate, \
  GSetShapoid*: _GSetIterForwardCreate, \
  const GSetShapoid*: _GSetIterForwardCreate, \
  GSetKnapSackPod*: _GSetIterForwardCreate, \
  const GSetKnapSackPod*: _GSetIterForwardCreate, \
  GSetPBPhysParticle*: _GSetIterForwardCreate, \
  const GSetPBPhysParticle*: _GSetIterForwardCreate, \
  GSetGenTree*: _GSetIterForwardCreate, \
  const GSetGenTree*: _GSetIterForwardCreate, \
  GSetStr*: _GSetIterForwardCreate, \
  const GSetStr*: _GSetIterForwardCreate, \
  GSetGenTreeStr*: _GSetIterForwardCreate, \
  const GSetGenTreeStr*: _GSetIterForwardCreate, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))

#define GSetIterForwardCreateStatic(Set) _Generic(Set, \
  GSet*: _GSetIterForwardCreateStatic, \
  const GSet*: _GSetIterForwardCreateStatic, \
  GSetVecFloat*: _GSetIterForwardCreateStatic, \
  const GSetVecFloat*: _GSetIterForwardCreateStatic, \
  GSetVecShort*: _GSetIterForwardCreateStatic, \
  const GSetVecShort*: _GSetIterForwardCreateStatic, \
  GSetBCurve*: _GSetIterForwardCreateStatic, \
  const GSetBCurve*: _GSetIterForwardCreateStatic, \
  GSetSCurve*: _GSetIterForwardCreateStatic, \
  const GSetSCurve*: _GSetIterForwardCreateStatic, \
  GSetShapoid*: _GSetIterForwardCreateStatic, \
  const GSetShapoid*: _GSetIterForwardCreateStatic, \
  GSetKnapSackPod*: _GSetIterForwardCreateStatic, \
  const GSetKnapSackPod*: _GSetIterForwardCreateStatic, \
  GSetPBPhysParticle*: _GSetIterForwardCreateStatic, \
  const GSetPBPhysParticle*: _GSetIterForwardCreateStatic, \
  GSetGenTree*: _GSetIterForwardCreateStatic, \
  const GSetGenTree*: _GSetIterForwardCreateStatic, \
  GSetStr*: _GSetIterForwardCreateStatic, \
  const GSetStr*: _GSetIterForwardCreateStatic, \
  GSetGenTreeStr*: _GSetIterForwardCreateStatic, \
  const GSetGenTreeStr*: _GSetIterForwardCreateStatic, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))

#define GSetIterBackwardCreate(Set) _Generic(Set, \
  GSet*: _GSetIterBackwardCreate, \
  const GSet*: _GSetIterBackwardCreate, \
  GSetVecFloat*: _GSetIterBackwardCreate, \
  const GSetVecFloat*: _GSetIterBackwardCreate, \
  GSetVecShort*: _GSetIterBackwardCreate, \
  const GSetVecShort*: _GSetIterBackwardCreate, \
  GSetBCurve*: _GSetIterBackwardCreate, \
  const GSetBCurve*: _GSetIterBackwardCreate, \
  GSetSCurve*: _GSetIterBackwardCreate, \
  const GSetSCurve*: _GSetIterBackwardCreate, \
  GSetShapoid*: _GSetIterBackwardCreate, \
  const GSetShapoid*: _GSetIterBackwardCreate, \
  GSetKnapSackPod*: _GSetIterBackwardCreate, \
  const GSetKnapSackPod*: _GSetIterBackwardCreate, \
  GSetPBPhysParticle*: _GSetIterBackwardCreate, \
  const GSetPBPhysParticle*: _GSetIterBackwardCreate, \
  GSetGenTree*: _GSetIterBackwardCreate, \
  const GSetGenTree*: _GSetIterBackwardCreate, \
  GSetGenTreeStr*: _GSetIterBackwardCreate, \
  const GSetGenTreeStr*: _GSetIterBackwardCreate, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))

#define GSetIterBackwardCreateStatic(Set) _Generic(Set, \
  GSet*: _GSetIterBackwardCreateStatic, \
  const GSet*: _GSetIterBackwardCreateStatic, \
  GSetVecFloat*: _GSetIterBackwardCreateStatic, \
  const GSetVecFloat*: _GSetIterBackwardCreateStatic, \
  GSetVecShort*: _GSetIterBackwardCreateStatic, \
  const GSetVecShort*: _GSetIterBackwardCreateStatic, \
  GSetBCurve*: _GSetIterBackwardCreateStatic, \
  const GSetBCurve*: _GSetIterBackwardCreateStatic, \
  GSetSCurve*: _GSetIterBackwardCreateStatic, \
  const GSetSCurve*: _GSetIterBackwardCreateStatic, \
  GSetShapoid*: _GSetIterBackwardCreateStatic, \
  const GSetShapoid*: _GSetIterBackwardCreateStatic, \
  GSetKnapSackPod*: _GSetIterBackwardCreateStatic, \
  const GSetKnapSackPod*: _GSetIterBackwardCreateStatic, \
  GSetPBPhysParticle*: _GSetIterBackwardCreateStatic, \
  const GSetPBPhysParticle*: _GSetIterBackwardCreateStatic, \
  GSetGenTree*: _GSetIterBackwardCreateStatic, \
  const GSetGenTree*: _GSetIterBackwardCreateStatic, \
  GSetStr*: _GSetIterBackwardCreateStatic, \
  const GSetStr*: _GSetIterBackwardCreateStatic, \
  GSetGenTreeStr*: _GSetIterBackwardCreateStatic, \
  const GSetGenTreeStr*: _GSetIterBackwardCreateStatic, \
  default: PBErrInvalidPolymorphism)((GSet*)(Set))

#define GSetIterSetGSet(Iter, Set) _Generic(Iter, \
  GSetIterForward*: _Generic(Set, \
    GSet*: GSetIterForwardSetGSet, \
    const GSet*: GSetIterForwardSetGSet, \
    GSetVecFloat*: GSetIterForwardSetGSet, \
    const GSetVecFloat*: GSetIterForwardSetGSet, \
    GSetVecShort*: GSetIterForwardSetGSet, \
    const GSetVecShort*: GSetIterForwardSetGSet, \
    GSetBCurve*: GSetIterForwardSetGSet, \
    const GSetBCurve*: GSetIterForwardSetGSet, \
    GSetSCurve*: GSetIterForwardSetGSet, \
    const GSetSCurve*: GSetIterForwardSetGSet, \
    GSetShapoid*: GSetIterForwardSetGSet, \
    const GSetShapoid*: GSetIterForwardSetGSet, \
    GSetKnapSackPod*: GSetIterForwardSetGSet, \
    const GSetKnapSackPod*: GSetIterForwardSetGSet, \
    GSetPBPhysParticle*: GSetIterForwardSetGSet, \
    const GSetPBPhysParticle*: GSetIterForwardSetGSet, \
    GSetGenTree*: GSetIterForwardSetGSet, \
    const GSetGenTree*: GSetIterForwardSetGSet, \
    GSetStr*: GSetIterForwardSetGSet, \
    const GSetStr*: GSetIterForwardSetGSet, \
    GSetGenTreeStr*: GSetIterForwardSetGSet, \
    const GSetGenTreeStr*: GSetIterForwardSetGSet, \
    default: PBErrInvalidPolymorphism), \
  GSetIterBackward*: _Generic(Set, \
    GSet*: GSetIterBackwardSetGSet, \
    const GSet*: GSetIterBackwardSetGSet, \
    GSetVecFloat*: GSetIterBackwardSetGSet, \
    const GSetVecFloat*: GSetIterBackwardSetGSet, \
    GSetVecShort*: GSetIterBackwardSetGSet, \
    const GSetVecShort*: GSetIterBackwardSetGSet, \
    GSetBCurve*: GSetIterBackwardSetGSet, \
    const GSetBCurve*: GSetIterBackwardSetGSet, \
    GSetSCurve*: GSetIterBackwardSetGSet, \
    const GSetSCurve*: GSetIterBackwardSetGSet, \
    GSetShapoid*: GSetIterBackwardSetGSet, \
    const GSetShapoid*: GSetIterBackwardSetGSet, \
    GSetKnapSackPod*: GSetIterBackwardSetGSet, \
    const GSetKnapSackPod*: GSetIterBackwardSetGSet, \
    GSetPBPhysParticle*: GSetIterBackwardSetGSet, \
    const GSetPBPhysParticle*: GSetIterBackwardSetGSet, \
    GSetGenTree*: GSetIterBackwardSetGSet, \
    const GSetGenTree*: GSetIterBackwardSetGSet, \
    GSetStr*: GSetIterBackwardSetGSet, \
    const GSetStr*: GSetIterBackwardSetGSet, \
    GSetGenTreeStr*: GSetIterBackwardSetGSet, \
    const GSetGenTreeStr*: 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 GSetIterStepBack(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardStepBack, \
  GSetIterBackward*: GSetIterBackwardStepBack, \
  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, \
  const GSetIterForward*: GSetIterForwardIsFirst, \
  GSetIterBackward*: GSetIterBackwardIsFirst, \
  const GSetIterBackward*: GSetIterBackwardIsFirst, \
  default: PBErrInvalidPolymorphism)(Iter)

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

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

```
#define GSetIterSetData(Iter, Data) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardSetData, \
  const GSetIterForward*: GSetIterForwardSetData, \
  GSetIterBackward*: GSetIterBackwardSetData, \
  const GSetIterBackward*: GSetIterBackwardSetData, \
  default: PBErrInvalidPolymorphism)(Iter, Data)

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

#define GSetIterGetSortVal(Iter) _Generic(Iter, \
  GSetIterForward*: GSetIterForwardGetSortVal, \
  const GSetIterForward*: GSetIterForwardGetSortVal, \
  GSetIterBackward*: GSetIterBackwardGetSortVal, \
  const GSetIterBackward*: GSetIterBackwardGetSortVal, \
  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

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

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

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

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

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

// Function to clone a GSet,
// Return a pointer toward the new GSet
GSet* GSetClone(const GSet* const 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 = (GSetElem*)GSetHeadElem(that);
  // 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, GSetElemData(ptr));
    // Copy the sort value
    GSetElemSetSortVal((GSetElem*)GSetTailElem(c),
      GSetElemGetSortVal(ptr));
    // Move the pointer to the next element
    ptr = (GSetElem*)GSetElemNext(ptr);
  }
  // Return the clone
  return c;
}

// Function to free the memory used by the GSet
void _GSetFree(GSet** that) {
  if (that == NULL || *that == NULL) return;
  // Empty the GSet
  GSetFlush(*that);
  // Free the memory
  free(*that);
  // Set the pointer to null
  *that = NULL;
}

// Function to print a GSet
// Use the function 'printData' to print the data pointed to by
// the elements, and print 'sep' between each element
// If printData is null, print the pointer value instead
// Do nothing if arguments are invalid
void _GSetPrint(GSet* const that, FILE* const stream,
  void(*printData)(const void* const data, FILE* const stream),
  const char* const 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 = (GSetElem*)GSetHeadElem(that);
  // 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)(GSetElemData(p), stream);
    // Else, there is no print function for the data
    } else {
      // Print the pointer value instead
      fprintf(stream, "%p", GSetElemData(p));
    }
    // Move to the next element
    p = (GSetElem*)GSetElemNext(p);
    // 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* const that, void* const data,
  const 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
  GSetElemSetData(e, data);
  // Memorize the sorting value
  GSetElemSetSortVal(e, 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;
    GSetElemSetNext(e, NULL);
    GSetElemSetPrev(e, NULL);
  } else {
    // Set a pointer to the head of the GSet
    GSetElem* p = (GSetElem*)GSetHeadElem(that);
    // While the pointed element has a lower value than the
    // new element, move the pointer to the next element
    while (p != NULL && GSetElemGetSortVal(p) <= v)
      p = (GSetElem*)GSetElemNext(p);
    // Set the next element of the new element to the current element
    GSetElemSetNext(e, p);
    // If the current element is not null
    if (p != NULL) {
      // Insert the new element inside the list of elements before p
      GSetElemSetPrev(e, (GSetElem*)GSetElemPrev(p));
      if (GSetElemPrev(p) != NULL)
        GSetElemSetNext((GSetElem*)GSetElemPrev(p), e);
      else
        that->_head = e;
      GSetElemSetPrev(p, e);
    // Else, if the current element is null
    } else {
      // Insert the new element at the tail of the GSet
      GSetElemSetPrev(e, (GSetElem*)GSetTailElem(that));
      if (GSetTailElem(that) != NULL)
        GSetElemSetNext((GSetElem*)GSetTailElem(that), e);
      that->_tail = e;
      if (GSetHeadElem(that) == 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* const that, void* const data,
  const long 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 (%ld>=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
      GSetElemSetData(e, data);
      // By default set the sorting value to 0.0
      GSetElemSetSortVal(e, 0.0);
      // Set a pointer toward the head of the GSet
      GSetElem* p = (GSetElem*)GSetHeadElem(that);
      // Move the pointer to the iElem-th element
      for (long i = iElem; i > 0 && p != NULL;
        --i, p = (GSetElem*)GSetElemNext(p));
      // Insert the element before the pointer
      GSetElemSetNext(e, p);
      GSetElemSetPrev(e, (GSetElem*)GSetElemPrev(p));
      GSetElemSetPrev(p, e);
      GSetElemSetNext((GSetElem*)GSetElemPrev(e), e);
      // Increment the number of elements
      ++(that->_nbElem);
    }
  }
}

// Function to sort the element of the gset in increasing order of
// _sortVal
// Do nothing if arguments are invalid or the sort failed
static GSet* GSetSortRec(GSet** s);
void _GSetSort(GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create a clone of the original set
  GSet* clone = GSetClone(that);
  // Create recursively the sorted set
  GSet* res = GSetSortRec(&clone);
  // If we could sort the set
  if (res != NULL) {
    // Update the original set with the result one
    GSetFlush(that);
    memcpy(that, res, sizeof(GSet));
    // Free the memory used by the result set
    free(res);
    res = NULL;
  }
}
GSet* GSetSortRec(GSet** s) {
  // Declare a variable for the result
  GSet* res = NULL;
  // If the set contains no element or one element
  if ((*s)->_nbElem == 0 || (*s)->_nbElem == 1) {
    // Return the set
    res = *s;
  // Else, the set contains several elements
  } else {
    // Create two sets, one for elements lower than the pivot
    // one for elements greater or equal than the pivot
    GSet* lower = GSetCreate();
```

```
GSet* greater = GSetCreate();
res = GSetCreate();

// -------------------------
// Selecting the pivot as the middle element seemed to me better
// but test with UnitTestGSetSortBig proved me wrong: 1492/2060/2554
// -------------------------

// Declare a variable to memorize the pivot, which is equal
// to the sort value of the first element of the set
float pivot = GSetElemGetSortVal(GSetHeadElem(*s));
// Pop the pivot and put it in the result
void* data = GSetPop(*s);
GSetAppend(res, data);
GSetElemSetSortVal((GSetElem*)GSetHeadElem(res), 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 = GSetElemGetSortVal((GSetElem*)GSetHeadElem(*s));
  // Pop the head element
  data = GSetPop(*s);
  // If the poped element has a sort value equal to the pivot
  if (fabs(val - pivot) < GSET_EPSILON) {
    // Insert it in the result set
    GSetAppend(res, data);
    // Copy the sort value
    GSetElemSetSortVal((GSetElem*)GSetTailElem(res), val);
  // If the poped element has a sort value lower than the pivot
  } else if (val < pivot) {

    // -------------------------
    // The following seemed to me a good idea but test with
    // UnitTestGSetSortBig proved me wrong: 1496/2054/2626
    // Insert at the beginning if the sort value is lower or equal
    // than the sort value of the head of the lower set, or if it's
    // empty
    // Else, insert at the end of the lower set
    // -------------------------

    // Insert it in the lower set
    GSetAppend(lower, data);
    // Copy the sort value
    GSetElemSetSortVal((GSetElem*)GSetTailElem(lower), val);
  // Else, the poped element has a sort value greater than
  // the pivot
  } else {

    // -------------------------
    // The following seemed to me a good idea but test with
    // UnitTestGSetSortBig proved me wrong: 1496/2054/2626
    // Insert at the beginning if the sort value is lower or equal
    // than the sort value of the head of the greater set, or if it's
    // empty
    // Else, insert at the end of the greater set
    // -------------------------

    // Insert it in the greater set
    GSetAppend(greater, data);
    // Copy the sort value
    GSetElemSetSortVal((GSetElem*)GSetTailElem(greater), 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* const that, const long iElem, const long 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<=%ld<%ld)",
      iElem, GSetNbElem(that));
    PBErrCatch(GenBrushErr);
  }
  if (pos < 0 || pos >= GSetNbElem(that)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is invalid (0<=%ld<%ld)",
      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 moved element
  GSetElem* elem = (GSetElem*)GSetElement(that, iElem);
  //Declare two variables to memorize the sort value and data
  // of the moved element
  float sortVal = GSetElemGetSortVal(elem);
  void* data = GSetElemData(elem);
  // Remove the moved element
  GSetRemove(that, iElem);
  // Insert  new element
  GSetInsert(that, data, pos);
  // Get a pointer to the newly inserted element
  elem = (GSetElem*)GSetElement(that, pos);
  // Correct the sorted value with the original value
  GSetElemSetSortVal(elem, sortVal);
}

// Return the number of (GSetElem._data=='data') in the GSet 'that'
long _GSetCount(const GSet* const that, const void* const 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
  long 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* const 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 = (GSetElem*)GSetHeadElem(set);
  // Return the new iterator
  return ret;
}

// Create a new GSetIterBackward for the GSet 'set'
// The iterator is reset upon creation
GSetIterBackward* _GSetIterBackwardCreate(GSet* const 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(
  const GSetIterForward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Create the clone
  GSetIterForward* ret = GSetIterForwardCreate(that->_set);
  ret->_curElem = that->_curElem;
  // return the clone
  return ret;
}

// Clone a GSetIterBackward
GSetIterBackward* GSetIterBackwardClone(
  const GSetIterBackward* const 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;
}

// Shuffle the GSet 'that'
// The random generator must have been initialized before calling
// this function
// This function modifies the _sortVal of each elements in 'that'
void GSetShuffle(GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // If the set is empty there is nothong to do
  if (GSetNbElem(that) <= 1500)
    GSetShuffleB(that);
  else
    GSetShuffleA(that);
}

void GSetShuffleA(GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Set the sort value randomly then sort the GSet
  // If the set is empty there is nothong to do
  if (GSetNbElem(that) == 0)
    return;
  // Create an iterator on the set
  GSetIterForward iter = GSetIterForwardCreateStatic(that);
  // Loop on the set
  do {
    // Set a random value to the element
    GSetElemSetSortVal((GSetElem*)GSetIterGetElem(&iter), rnd());
  } while (GSetIterStep(&iter));
  // Sort the set
  GSetSort(that);
}

void GSetShuffleB(GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // AddSort each element with a random value in a new GSet
  // Create a temporary set
  GSet shuffled = GSetCreateStatic();
  // Append all the elements of the initial set, sorted with a random
  // value
  while (GSetNbElem(that) > 0) {
    void* data = GSetPop(that);
    GSetAddSort(&shuffled, data, rnd());
  }
  // put back the shuffled set into the original set
  GSetMerge(that, &shuffled);
}

void GSetShuffleC(GSet* const that) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Fischer-Yates algorithm
  for (long i = GSetNbElem(that); i--;) {
     long j = (long)round(rnd() * (float)i);
     GSetSwitch(that, i, j);
  }
}
```

## 2.2 gset-inline.c

```
// *************** GSET-INLINE.C ***************

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

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

// Function to empty the GSet
#if BUILDMODE != 0
inline
#endif
void _GSetFlush(GSet* const 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* const that, void* const 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
  GSetElemSetData(e, data);
  // By default set the sorting value to 0.0
  GSetElemSetSortVal(e, 0.0);
  // Add the element at the head of the GSet
  GSetElemSetPrev(e, NULL);
  if (GSetHeadElem(that) != NULL)
    GSetElemSetPrev((GSetElem*)GSetHeadElem(that), e);
  GSetElemSetNext(e, (GSetElem*)GSetHeadElem(that));
  that->_head = e;
  if (GSetTailElem(that) == 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* const that, void* const 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) {
    GSetElemSetData(e, data);
    GSetElemSetSortVal(e, 0.0);
    GSetElemSetPrev(e, (GSetElem*)GSetTailElem(that));
    GSetElemSetNext(e, NULL);
    if (GSetTailElem(that) != NULL)
      GSetElemSetNext((GSetElem*)GSetTailElem(that), e);
    that->_tail = e;
    if (GSetHeadElem(that) == 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* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  void* ret = NULL;
  GSetElem* p = (GSetElem*)GSetHeadElem(that);
  if (p != NULL) {
    ret = GSetElemData(p);
    that->_head = (GSetElem*)GSetElemNext(p);
    if (GSetElemNext(p) != NULL)
      GSetElemSetPrev((GSetElem*)GSetElemNext(p), NULL);
    GSetElemSetNext(p, NULL);
    GSetElemSetData(p, NULL);
    if (GSetTailElem(that) == 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* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  void* ret = NULL;
  GSetElem* p = (GSetElem*)GSetTailElem(that);
  if (p != NULL) {
    ret = GSetElemData(p);
    that->_tail = (GSetElem*)GSetElemPrev(p);
    if (GSetElemPrev(p) != NULL)
      GSetElemSetNext((GSetElem*)GSetElemPrev(p), NULL);
    GSetElemSetPrev(p, NULL);
    GSetElemSetData(p, NULL);
    if (GSetHeadElem(that) == 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* const that, GSetElem** elem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (elem == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'elem' is null");
    PBErrCatch(GSetErr);
  }
  if (*elem == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'*elem' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Variable to memorize the return value
  void* ret = NULL;
  // Memorize the data at iElem-th position
  ret = GSetElemData(*elem);
  // Remove the element
  if (GSetElemNext(*elem) != NULL)
    GSetElemSetPrev((GSetElem*)GSetElemNext(*elem),
      (GSetElem*)GSetElemPrev(*elem));
  if (GSetElemPrev(*elem) != NULL)
    GSetElemSetNext((GSetElem*)GSetElemPrev(*elem),
      (GSetElem*)GSetElemNext(*elem));
  if (GSetHeadElem(that) == *elem)
    that->_head = (GSetElem*)GSetElemNext(*elem);
  if (that->_tail == (*elem))
    that->_tail = (GSetElem*)GSetElemPrev(*elem);
  GSetElemSetNext(*elem, NULL);
  GSetElemSetPrev(*elem, NULL);
  GSetElemSetData(*elem, 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* const that, const void* const 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 = (GSetElem*)GSetFirstElem(that, data);
  // If we could find an element
  if (elem != NULL)
    // Remove this element
    GSetRemoveElem(that, &elem);
}

// 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* const that, const void* const 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 = (GSetElem*)GSetLastElem(that, data);
  // If we could find an element
  if (elem != NULL)
    // Remove this element
    GSetRemoveElem(that, &elem);
}

// 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* const that, const long 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<=%ld<%ld)",
      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 = (GSetElem*)GSetHeadElem(that);
  // Move the pointer to the iElem-th element
  for (long i = iElem; i > 0 && p != NULL;
    --i, p = (GSetElem*)GSetElemNext(p));
  // Memorize the data at iElem-th position
  ret = GSetElemData(p);
  // Remove the element
  if (GSetElemNext(p) != NULL)
    GSetElemSetPrev((GSetElem*)GSetElemNext(p),
      (GSetElem*)GSetElemPrev(p));
  if (GSetElemPrev(p) != NULL)
    GSetElemSetNext((GSetElem*)GSetElemPrev(p),
      (GSetElem*)GSetElemNext(p));
  if (GSetHeadElem(that) == p)
    that->_head = (GSetElem*)GSetElemNext(p);
  if (that->_tail == p)
    that->_tail = (GSetElem*)GSetElemPrev(p);
  GSetElemSetNext(p, NULL);
  GSetElemSetPrev(p, NULL);
  GSetElemSetData(p, 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* const that, const void* const 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 = (GSetElem*)GSetTailElem(that);
  // Loop on elements until we reached the head of the list
  while (p != NULL) {
    // If the element points toward data
    if (GSetElemData(p) == data) {
      // Memorize the previous element before deleting
      GSetElem* prev = (GSetElem*)GSetElemPrev(p);
      // 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 = (GSetElem*)GSetElemPrev(p);
    }
  }
}

// Function to get the data at the GSetElem
#if BUILDMODE != 0
inline
#endif
void* GSetElemData(const GSetElem* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data
  return that->_data;
}

// Function to get the data at the 'iElem'-th position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetGet(const GSet* const that, const long 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<=%ld<%ld)",
      iElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data of the iElem-th element
  return GSetElemData(GSetElement(that, iElem));
}

// Function to get the data at first position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetHead(const GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data of the first element
  return GSetElemData(GSetHeadElem(that));
}

// Function to get the data at last position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
void* _GSetTail(const GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data of the last element
  return GSetElemData(GSetTailElem(that));
}

// Function to get the GSetElem at first position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
const GSetElem* _GSetHeadElem(const GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the first element
  return that->_head;
}
```

```
// Function to get the GSetElem at last position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
const GSetElem* _GSetTailElem(const GSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the last element
  return that->_tail;
}

// Function to get the element at the 'iElem'-th position of the GSet
// without removing it
#if BUILDMODE != 0
inline
#endif
const GSetElem* _GSetElement(const GSet* const that, const long 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 = PBErrTypeInvalidArg;
    sprintf(GSetErr->_msg, "'iElem' is invalid (0<=%ld<%ld)",
      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 = (GSetElem*)GSetHeadElem(that);
  // Move to the next element iElem times
  for (long i = iElem; i > 0 && ret != NULL;
    --i, ret = (GSetElem*)GSetElemNext(ret));
  // 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
long _GSetGetIndexFirst(const GSet* const that, const void* const 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 = (GSetElem*)GSetHeadElem(that);
  // Set a variable to memorize index
  long index = 0;
  // Loop on elements until we have found the
  // requested data or reached the end of the list
  while (p != NULL && GSetElemData(p) != data) {
    ++index;
    p = (GSetElem*)GSetElemNext(p);
  }
  // If the pointer is null it means the data wasn't in the GSet
  if (p == NULL)
    index = -1;
  // Return the index
  return index;
}

// Function to get the index of the last element of the GSet
// which point to 'data'
// Return -1 if 'data' is not in the set
#if BUILDMODE != 0
inline
#endif
long _GSetGetIndexLast(const GSet* const that, const void* const 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 = (GSetElem*)GSetTailElem(that);
  // Set a variable to memorize index
  long index = that->_nbElem - 1;
  // Loop on elements until we have found the
  // requested data or reached the head of the list
  while (p != NULL && GSetElemData(p) != data) {
    --index;
    p = (GSetElem*)GSetElemPrev(p);
  }
  // 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
const GSetElem* _GSetFirstElem(const GSet* const that,
  const void* const 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 = (GSetElem*)GSetHeadElem(that);
  // Loop on elements until we have found the
  // requested data or reached the end of the list
  while (p != NULL && GSetElemData(p) != data)
    p = (GSetElem*)GSetElemNext(p);
  // 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
const GSetElem* _GSetLastElem(const GSet* const that,
  const void* const 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 = (GSetElem*)GSetTailElem(that);
  // Loop on elements until we have found the
  // requested data or reached the end of the list
  while (p != NULL && GSetElemData(p) != data)
    p = (GSetElem*)GSetElemPrev(p);
  // 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* const that, GSet* const 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));
    // Else, if 'that' is not empty
    } else {
      // Add 'set' to the tail of 'that'
      GSetElemSetNext((GSetElem*)GSetTailElem(that),
        (GSetElem*)GSetHeadElem(set));
      // Add 'that' to the head of 'set'
      GSetElemSetPrev((GSetElem*)GSetHeadElem(set),
        (GSetElem*)GSetTailElem(that));
      // Update the tail of 'that'
      that->_tail = (GSetElem*)GSetTailElem(set);
      // 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* const that, GSetElem* const 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
  long nb = 0;
  // If e is not the head of that
  if (GSetHeadElem(that) != e) {
    GSetElem* ptr = e;
    // While there is an element before e
    do {
      // Increment the number of element
      ++nb;
      // Move to the previous element
      ptr = (GSetElem*)GSetElemPrev(ptr);
    } while (ptr != NULL && ptr != GSetHeadElem(that));
    // 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 = (GSetElem*)GSetTailElem(that);
  // Set the number of element of res
  res->_nbElem = that->_nbElem - nb;
  // Set the tail of s
  that->_tail = (GSetElem*)GSetElemPrev(e);
  // 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
    GSetElemSetNext((GSetElem*)GSetTailElem(that), NULL);
  // Disconnect the head of res
  GSetElemSetPrev((GSetElem*)GSetHeadElem(res), 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* const that, const long iElem, const long jElem) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
  if (iElem < 0 || iElem >= that->_nbElem) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'iElem' is invalid (0<=%ld<%ld)",
      iElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
  if (jElem < 0 || jElem >= that->_nbElem) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'jElem' is invalid (0<=%ld<%ld)",
      jElem, that->_nbElem);
    PBErrCatch(GSetErr);
  }
#endif
  // Get the two elements
  GSetElem* iPtr = (GSetElem*)GSetElement(that, iElem);
  GSetElem* jPtr = (GSetElem*)GSetElement(that, jElem);
  // Switch the elements
  swap(iPtr->_sortVal, jPtr->_sortVal);
  swap(iPtr->_data, jPtr->_data);
}

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

// Set the data of the GSetElem 'that' to 'd'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetData(GSetElem* const that, void* const d) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  that->_data = d;
}

// Set the previous element of the GSetElem 'that' to 'e'
// Do not set the link back in 'e'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetPrev(GSetElem* const that, GSetElem* const e) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  that->_prev = e;
}

// Set the next element of the GSetElem 'that' to 'e'
// Do not set the link back in 'e'
#if BUILDMODE != 0
inline
#endif
void GSetElemSetNext(GSetElem* const that, GSetElem* const e) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  that->_next = e;
}

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

// Create a new GSetIterBackward for the GSet 'set'
// The iterator is reset upon creation
#if BUILDMODE != 0
inline
#endif
GSetIterBackward _GSetIterBackwardCreateStatic(GSet* const 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* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Reset
  that->_curElem = (GSetElem*)GSetHeadElem(that->_set);
}

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

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

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

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

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

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

// 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* const 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(GSetElemData(that->_curElem), 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(const GSetIterForward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return (that->_curElem == GSetHeadElem(that->_set));
}

// 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(const GSetIterBackward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return (that->_curElem == GSetTailElem(that->_set));
}

// 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(const GSetIterForward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return (that->_curElem == GSetTailElem(that->_set));
}

// 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(const GSetIterBackward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return (that->_curElem == GSetHeadElem(that->_set));
}

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

```
// Change the attached set of the iterator, and reset it
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardSetGSet(GSetIterBackward* const that,
  GSet* const 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(const GSetIterForward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  // Return the data
  return GSetElemData(that->_curElem);
}

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

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

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

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

```
  GSetElem *next = (GSetElem*)GSetElemNext(that->_curElem);
  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* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  GSetElem *prev = (GSetElem*)GSetElemPrev(that->_curElem);
  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* const that, const GSet* const 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* const that, const GSet* const 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 = (GSetElem*)GSetIterGetElem(&iter);
      // Append the data of the element according to the sorting value
      GSetAddSort(that, GSetElemData(elem), GSetElemGetSortVal(elem));
    } while (GSetIterStep(&iter));
  }
}

// Return the sort value of GSetElem 'that'
#if BUILDMODE != 0
inline
#endif
float GSetElemGetSortVal(const GSetElem* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return that->_sortVal;
}

// Return the next element of GSetElem 'that'
#if BUILDMODE != 0
inline
#endif
const GSetElem* GSetElemNext(const GSetElem* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return that->_next;
}

// Return the previous element of GSetElem 'that'
#if BUILDMODE != 0
inline
#endif
const GSetElem* GSetElemPrev(const GSetElem* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return that->_prev;
}

// Set the data of the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void GSetIterForwardSetData(const GSetIterForward* const that,
  void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  GSetElemSetData((GSetElem*)GSetIterGetElem(that), data);
}

// Set the data of the element currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void GSetIterBackwardSetData(const GSetIterBackward* const that,
  void* data) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  GSetElemSetData((GSetElem*)GSetIterGetElem(that), data);
}

// Return the sort value of the element currently pointed to by the
// iterator
#if BUILDMODE != 0
inline
#endif
float GSetIterForwardGetSortVal(const GSetIterForward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return GSetElemGetSortVal(GSetIterGetElem(that));
}

// Return the sort value of the element currently pointed to by the
// iterator
#if BUILDMODE != 0
inline
#endif
float GSetIterBackwardGetSortVal(const GSetIterBackward* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GSetErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
  }
#endif
  return GSetElemGetSortVal(GSetIterGetElem(that));
}
```

## 3 Makefile

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

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

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

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

# 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"
#include "pbmath.h"

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

void UnitTestGSetElemGetSet() {
  GSetElem elem;
  GSetElem elemNext;
  GSetElem elemPrev;
  float val = 1.0;
  char data = ' ';
  elem._next = &elemNext;
  elem._prev = &elemPrev;
  elem._sortVal = val;
  elem._data = &data;
  if (GSetElemNext(&elem) != &elemNext) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemNext failed");
    PBErrCatch(GSetErr);
  }
  if (GSetElemPrev(&elem) != &elemPrev) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemPrev failed");
    PBErrCatch(GSetErr);
  }
  if (GSetElemData(&elem) != &data) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemData failed");
    PBErrCatch(GSetErr);
  }
  if (!ISEQUALF(GSetElemGetSortVal(&elem), val)) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemGetSortVal failed");
    PBErrCatch(GSetErr);
  }
  float valb = 2.0;
  char datab = ' ';
  GSetElem elemNextb;
  GSetElem elemPrevb;
  GSetElemSetData(&elem, &datab);
  GSetElemSetSortVal(&elem, valb);
  GSetElemSetNext(&elem, &elemNextb);
  GSetElemSetPrev(&elem, &elemPrevb);
  if (GSetElemData(&elem) != &datab) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemSetData failed");
    PBErrCatch(GSetErr);
  }
  if (!ISEQUALF(GSetElemGetSortVal(&elem), valb)) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemSetSortVal failed");
    PBErrCatch(GSetErr);
  }
  if (GSetElemNext(&elem) != &elemNextb) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemSetNext failed");
    PBErrCatch(GSetErr);
  }
  if (GSetElemPrev(&elem) != &elemPrevb) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetElemSetPrev failed");
    PBErrCatch(GSetErr);
  }

  printf("UnitTestGSetElemGetSet OK\n");
}

void UnitTestGSetElem() {
  UnitTestGSetElemGetSet();
  printf("UnitTestGSetElem OK\n");
}

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

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

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

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

void UnitTestGSetInsertRemove() {
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  GSetIterForward iter = GSetIterForwardCreateStatic(&set);
  GSetInsert(&set, a, 2);
  int *checka[3] = {NULL, NULL, a};
  int i = 0;
  GSetIterReset(&iter);
  do {
    if (checka[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetInsert NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetFlush(&set);
  GSetInsert(&set, a, 0);
  GSetInsert(&set, a + 1, 1);
  GSetInsert(&set, a + 2, 1);
  GSetInsert(&set, a + 3, 1);
  GSetInsert(&set, a + 4, 3);
  int *checkb[5] = {a, a + 3, a + 2, a + 4, a + 1};
  i = 0;
  GSetIterReset(&iter);
  do {
    if (checkb[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetInsert NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetRemove(&set, 0);
  int *checkc[4] = {a + 3, a + 2, a + 4, a + 1};
  i = 0;
  GSetIterReset(&iter);
  do {
    if (checkc[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetRemove NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetRemove(&set, 3);
  int *checkd[3] = {a + 3, a + 2, a + 4};
  i = 0;
  GSetIterReset(&iter);
  do {
    if (checkd[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetRemove NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetRemove(&set, 1);
  int *checke[2] = {a + 3, a + 4};
  i = 0;
  GSetIterReset(&iter);
  do {
    if (checke[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetRemove NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetRemove(&set, 1);
  int *checkf[1] = {a + 3};
  i = 0;
  GSetIterReset(&iter);
  do {
    if (checkf[i] != GSetIterGet(&iter)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetRemove NOK");
      PBErrCatch(GSetErr);
    }
    ++i;
  } while (GSetIterStep(&iter));
  GSetRemove(&set, 0);
  if (set._nbElem != 0 || set._head != NULL || set._tail != NULL) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetRemove NOK");
    PBErrCatch(GSetErr);
  }
  printf("UnitTestGSetInsertRemove OK\n");
}

void UnitTestGSetNbElemGet() {
  int a[5] = {0, 1, 2, 3, 4};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;) {
    GSetPush(&set, a + i);
    if (5 - i != GSetNbElem(&set)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetNbElem NOK");
      PBErrCatch(GSetErr);
    }
  }
  for (int i = 5; i--;)
    if (i != *(int*)GSetGet(&set, i)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetGet NOK");
      PBErrCatch(GSetErr);
    }
  if (*(int*)GSetHead(&set) != 0) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetGetFirst NOK");
    PBErrCatch(GSetErr);
  }
  if (*(int*)GSetTail(&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((GSetElem*)GSetHeadElem(&set), a[j]);
    }
    GSetSort(&set);
    GSetIterReset(&iter);
    int v = *(int*)GSetIterGet(&iter);
    GSetIterStep(&iter);
    do {
      int w = *(int*)GSetIterGet(&iter);
      if (w < v) {
        GSetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GSetErr->_msg, "GSetSort NOK");
        PBErrCatch(GSetErr);
      }
      v = w;
    } while (GSetIterStep(&iter));
    GSetFlush(&set);
  }
  printf("UnitTestGSetSort OK\n");
}

int compare_floats(const void* a, const void* b)
{
  float arg1 = *(const float*)a;
  float arg2 = *(const float*)b;
  return (arg1 > arg2) - (arg1 < arg2);
}

void UnitTestGSetSortBig() {
  srandom(RANDOMSEED);
  int nbTest = 100;
  float sumTime = 0.0;
  float minTime = 100000.0;
  float maxTime = 0.0;
  #define sizeSet 1000000l
  for (int iTest = 0; iTest < nbTest; ++iTest) {
    GSet set = GSetCreateStatic();
    for (long i = 0; i < sizeSet; ++i) {
      GSetPush(&set, NULL);
      GSetElemSetSortVal((GSetElem*)GSetHeadElem(&set),
        rnd() * 100000.0);
    }
    clock_t clockBefore = clock();
    GSetSort(&set);
    clock_t clockAfter = clock();
    float delayMs = ((double)(clockAfter - clockBefore)) /
      CLOCKS_PER_SEC * 1000.0;
    if (minTime > delayMs)
      minTime = delayMs;
    if (maxTime < delayMs)
      maxTime = delayMs;
    sumTime += delayMs;
    GSetFlush(&set);
  }
  printf("Min/Avg/Max time to sort %li elements: %.1f/%.1f/%.1fms\n",
    sizeSet, minTime, sumTime / (float)nbTest, maxTime);

  float floats[sizeSet];
  sumTime = 0.0;
  minTime = 100000.0;
  maxTime = 0.0;
  for (int iTest = 0; iTest < nbTest; ++iTest) {
    for (long i = 0; i < sizeSet; ++i) {
      floats[i] = rnd() * 100000.0;
    }
    clock_t clockBefore = clock();
    qsort(floats, sizeSet, sizeof(int), compare_floats);
    clock_t clockAfter = clock();
    float delayMs = ((double)(clockAfter - clockBefore)) /
      CLOCKS_PER_SEC * 1000.0;
    if (minTime > delayMs)
      minTime = delayMs;
    if (maxTime < delayMs)
      maxTime = delayMs;
    sumTime += delayMs;
  }
  printf("For comparison, using qsort on an array of %li floats:\n",
    sizeSet);
  printf("  Min/Avg/Max time : %.1f/%.1f/%.1fms\n",
    minTime, sumTime / (float)nbTest, maxTime);

  printf("UnitTestGSetSortBig 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, (GSetElem*)GSetElement(&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 UnitTestGSetShuffle() {
  srandom(1);
  int a[5] = {1, 2, 3, 4, 5};
  GSet set = GSetCreateStatic();
  for (int i = 5; i--;)
    GSetPush(&set, a + i);
  GSetShuffle(&set);
  int b[5] = {2, 3, 4, 1, 5};
  for (int i = 0; i < 5; ++i) {
    int* j = GSetPop(&set);
    if (*j != b[i]) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "GSetSuffle NOK");
      PBErrCatch(GSetErr);
    }
  }
  printf("UnitTestGSetShuffle OK\n");
}

void UnitTestGSet() {
  UnitTestGSetCreateFree();
  UnitTestGSetClone();
  UnitTestGSetFlush();
  UnitTestGSetPrint();
  UnitTestGSetPushPopAppendDrop();
  UnitTestGSetAddSort();
  UnitTestGSetInsertRemove();
  UnitTestGSetNbElemGet();
  UnitTestGSetGetIndex();
  UnitTestGSetSort();
  UnitTestGSetSortBig();
  UnitTestGSetSplitMerge();
  UnitTestGSetSwitch();
  UnitTestGSetMoveElem();
  UnitTestGSetMergeSet();
  UnitTestGSetCount();
  UnitTestGSetShuffle();
  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, "GSetIterStep NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGetElem(&iter) != elem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterGetElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGet(&iter) != a + 1) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterGet NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterStepBack(&iter);
  if (iter._curElem != set._head) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterStepBack NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  printf("UnitTestGSetIteratorForwardStepGetGetElem OK\n");
}

void FunInc(void* data, void* param) {
  (void)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 UnitTestGSetIteratorForwardSet() {
  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, "GSetIterSetGSet NOK");
    PBErrCatch(GSetErr);
  }
  char c = ' ';
  GSetIterSetData(&iter, &c);
  if (GSetIterGet(&iter) != &c) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterSetData NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  GSetFlush(&setb);
  printf("UnitTestGSetIteratorForwardSet 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();
  UnitTestGSetIteratorForwardSet();
  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, "GSetIterStep NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGetElem(&iter) != elem) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterGetElem NOK");
    PBErrCatch(GSetErr);
  }
  if (GSetIterGet(&iter) != a + 3) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterGet NOK");
    PBErrCatch(GSetErr);
  }
  GSetIterStepBack(&iter);
  if (iter._curElem != set._tail) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterStepBack 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 UnitTestGSetIteratorBackwardSet() {
  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, "GSetIterSetGSet NOK");
    PBErrCatch(GSetErr);
  }
  char c = ' ';
  GSetIterSetData(&iter, &c);
  if (GSetIterGet(&iter) != &c) {
    GSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GSetErr->_msg, "GSetIterSetData NOK");
    PBErrCatch(GSetErr);
  }
  GSetFlush(&set);
  GSetFlush(&setb);
  printf("UnitTestGSetIteratorBackwardSet 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();
  UnitTestGSetIteratorBackwardSet();
  printf("UnitTestGSetIteratorBackward OK\n");
}

void UnitTestSpeedShuffle() {
  GSet set = GSetCreateStatic();
  long setSizeMin = 100;
  long setSizeMax = 10000;
  printf("Start speed test for shuffling algorithms\n\n");
  for (long setSize = setSizeMin; setSize <= setSizeMax; setSize *= 2) {
    for (long i = setSize; i--;)
      GSetPush(&set, i);
    clock_t clockBefore = clock();
    for (int run = 40; run--;)
      GSetShuffle(&set);
    clock_t clockAfter = clock();
    float delayMs = 0.025 * ((double)(clockAfter - clockBefore)) /
      CLOCKS_PER_SEC * 1000.0;
    printf("Delay to shuffle %ld elements with GSetShuffle: %fms\n",
      setSize, delayMs);
    clockBefore = clock();
    for (int run = 40; run--;)
      GSetShuffleA(&set);
    clockAfter = clock();
    float delayMsA = 0.025 * ((double)(clockAfter - clockBefore)) /
      CLOCKS_PER_SEC * 1000.0;
    printf("Delay to shuffle %ld elements with GSetShuffleA: %fms\n",
      setSize, delayMsA);
    clockBefore = clock();
    for (int run = 40; run--;)
      GSetShuffleB(&set);
    clockAfter = clock();
    float delayMsB = 0.025 * ((double)(clockAfter - clockBefore)) /
      CLOCKS_PER_SEC * 1000.0;
    printf("Delay to shuffle %ld elements with GSetShuffleB: %fms\n",
      setSize, delayMsB);
    clockBefore = clock();
    for (int run = 40; run--;)
      GSetShuffleC(&set);
    clockAfter = clock();
    float delayMsC = 0.025 * ((double)(clockAfter - clockBefore)) /
      CLOCKS_PER_SEC * 1000.0;
    printf("Delay to shuffle %ld elements with GSetShuffleC: %fms\n",
      setSize, delayMsC);
    printf("\n");
    GSetFlush(&set);
    if (setSize <= 1500 &&
      (delayMsB > delayMsA ||
      delayMsB > delayMsC)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "UnitTestSpeedShuffle NOK");
      PBErrCatch(GSetErr);
    } else if (setSize > 1500 &&
      (delayMsA > delayMsB ||
      delayMsA > delayMsC)) {
      GSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GSetErr->_msg, "UnitTestSpeedShuffle NOK");
      PBErrCatch(GSetErr);
    }
  }
  printf("UnitTestSpeedShuffle OK\n");
}

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

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

## 5 Unit tests output

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

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

UnitTestGSetPushPopAppendDrop OK
UnitTestGSetAddSort OK
UnitTestGSetInsertRemove OK
UnitTestGSetNbElemGet OK
UnitTestGSetGetIndex OK
UnitTestGSetSort OK
Min/Avg/Max time to sort 1000000 elements: 1285.1/1713.7/1907.4ms
For comparison, using qsort on an array of 1000000 floats:
  Min/Avg/Max time : 208.4/211.7/507.3ms
UnitTestGSetSortBig OK
UnitTestGSetSplitMerge OK
UnitTestGSetSwitch OK
UnitTestGSetMoveElem OK
UnitTestGSetMergeSet OK
UnitTestGSetMergeSet OK
UnitTestGSetShuffle OK
UnitTestGSet OK
UnitTestGSetIteratorForwardCreateFree OK
UnitTestGSetIteratorForwardClone OK
UnitTestGSetIteratorForwardReset OK
UnitTestGSetIteratorForwardStepGetGetElem OK
UnitTestGSetIteratorForwardApply OK
UnitTestGSetIteratorForwardIsFirstIsLast OK
UnitTestGSetIteratorForwardSet OK
UnitTestGSetIteratorForwardRemoveElem OK
UnitTestGSetIteratorForward OK
UnitTestGSetIteratorBackwardCreateFree OK
UnitTestGSetIteratorBackwardClone OK
UnitTestGSetIteratorBackwardReset OK
UnitTestGSetIteratorBackwardStepGetGetElem OK
UnitTestGSetIteratorBackwardApply OK
UnitTestGSetIteratorBackwardIsFirstIsLast OK
```

```
UnitTestGSetIteratorBackwardSet OK
UnitTestGSetIteratorBackward OK
Start speed test for shuffling algorithms

Delay to shuffle 100 elements with GSetShuffle: 0.009150ms
Delay to shuffle 100 elements with GSetShuffleA: 0.042950ms
Delay to shuffle 100 elements with GSetShuffleB: 0.008850ms
Delay to shuffle 100 elements with GSetShuffleC: 0.009825ms

Delay to shuffle 200 elements with GSetShuffle: 0.024150ms
Delay to shuffle 200 elements with GSetShuffleA: 0.093125ms
Delay to shuffle 200 elements with GSetShuffleB: 0.023700ms
Delay to shuffle 200 elements with GSetShuffleC: 0.036475ms

Delay to shuffle 400 elements with GSetShuffle: 0.072900ms
Delay to shuffle 400 elements with GSetShuffleA: 0.210250ms
Delay to shuffle 400 elements with GSetShuffleB: 0.071800ms
Delay to shuffle 400 elements with GSetShuffleC: 0.145675ms

Delay to shuffle 800 elements with GSetShuffle: 0.243775ms
Delay to shuffle 800 elements with GSetShuffleA: 0.464275ms
Delay to shuffle 800 elements with GSetShuffleB: 0.244475ms
Delay to shuffle 800 elements with GSetShuffleC: 0.580300ms

Delay to shuffle 1600 elements with GSetShuffle: 1.023000ms
Delay to shuffle 1600 elements with GSetShuffleA: 1.016900ms
Delay to shuffle 1600 elements with GSetShuffleB: 1.255075ms
Delay to shuffle 1600 elements with GSetShuffleC: 2.997075ms

Delay to shuffle 3200 elements with GSetShuffle: 2.216450ms
Delay to shuffle 3200 elements with GSetShuffleA: 2.205450ms
Delay to shuffle 3200 elements with GSetShuffleB: 10.448700ms
Delay to shuffle 3200 elements with GSetShuffleC: 21.226299ms

Delay to shuffle 6400 elements with GSetShuffle: 4.747125ms
Delay to shuffle 6400 elements with GSetShuffleA: 4.698825ms
Delay to shuffle 6400 elements with GSetShuffleB: 51.958527ms
Delay to shuffle 6400 elements with GSetShuffleC: 100.310150ms

UnitTestSpeedShuffle OK
UnitTestAll OK
```