

# GSet

P. Baillehache

November 23, 2017

## Contents

|          |                  |           |
|----------|------------------|-----------|
| <b>1</b> | <b>Interface</b> | <b>2</b>  |
| <b>2</b> | <b>Code</b>      | <b>7</b>  |
| <b>3</b> | <b>Makefile</b>  | <b>24</b> |
| <b>4</b> | <b>Usage</b>     | <b>25</b> |

## Introduction

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

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

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

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

# 1 Interface

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

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

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

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

void GSetIterUnsupported(void*t, ...);
#define GSetIterFree(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardFree, \
    GSetIterBackward*: GSetIterBackwardFree, \
    default: GSetIterUnsupported)(I)
#define GSetIterClone(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardClone, \
    GSetIterBackward*: GSetIterBackwardClone, \
    default: GSetIterUnsupported)(I)
#define GSetIterReset(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardReset, \
    GSetIterBackward*: GSetIterBackwardReset, \
    default: GSetIterUnsupported)(I)
#define GSetIterStep(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardStep, \
    GSetIterBackward*: GSetIterBackwardStep, \
    default: GSetIterUnsupported)(I)
#define GSetIterApply(I, F, P) _Generic((I), \
    GSetIterForward*: GSetIterForwardApply, \
    GSetIterBackward*: GSetIterBackwardApply, \
    default: GSetIterUnsupported)(I, F, P)
#define GSetIterIsFirst(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardIsFirst, \
    GSetIterBackward*: GSetIterBackwardIsFirst, \
    default: GSetIterUnsupported)(I)
#define GSetIterIsLast(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardIsLast, \
    GSetIterBackward*: GSetIterBackwardIsLast, \
    default: GSetIterUnsupported)(I)
#define GSetIterSetGSet(I, S) _Generic((I), \
    GSetIterForward*: GSetIterForwardSetGSet, \
    GSetIterBackward*: GSetIterBackwardSetGSet, \
    default: GSetIterUnsupported)(I, S)
#define GSetIterGet(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardGet, \
    GSetIterBackward*: GSetIterBackwardGet, \
    default: GSetIterUnsupported)(I)
#define GSetIterGetElem(I) _Generic((I), \
    GSetIterForward*: GSetIterForwardGetElem, \
    GSetIterBackward*: GSetIterBackwardGetElem, \
    default: GSetIterUnsupported)(I)

// ===== 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
    double _sortVal;
} GSetElem;

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

// Structures of the GSet iterators
typedef struct GSetIterForward {
    GSet *_set;
    GSetElem *_curElem;
} GSetIterForward;

typedef struct GSetIterBackward {
    GSet *_set;
    GSetElem *_curElem;
} GSetIterBackward;

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

// Function to create a new GSet,
// Return a pointer toward the new GSet, or null if it couldn't
// create the GSet
GSet* GSetCreate(void);

// Function to clone a GSet,
// Return a pointer toward the new GSet, or null if it couldn't
// clone the GSet
GSet* GSetClone(GSet *s);

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

// Function to empty the GSet
void GSetFlush(GSet *s);

// 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 *s, FILE* stream,
    void(*printData)(void *data, FILE *stream), char *sep);

// Function to insert an element pointing toward 'data' at the
// head of the GSet

```

```

// Do nothing if arguments are invalid
void GSetPush(GSet *s, void* data);

// Function to insert an element pointing toward 'data' at the
// position defined by 'v' sorting the set in decreasing order
// Do nothing if arguments are invalid
void GSetAddSort(GSet *s, void* data, double v);

// Function to insert an element pointing toward 'data' at the
// 'iElem'-th position
// If 'iElem' is greater than or equal to the number of element
// in the GSet, elements pointing toward null data are added
// Do nothing if arguments are invalid
void GSetInsert(GSet *s, void* data, int iElem);

// Function to insert an element pointing toward 'data' at the
// tail of the GSet
// Do nothing if arguments are invalid
void GSetAppend(GSet *s, void* data);

// Function to remove the element at the head of the GSet
// Return the data pointed to by the removed element, or null if the
// GSet is empty
// Return null if arguments are invalid
void* GSetPop(GSet *s);

// 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
// Return null if arguments are invalid
void* GSetDrop(GSet *s);

// Function to remove the element at the 'iElem'-th position of the GSet
// Return the data pointed to by the removed element
// Return null if arguments are invalid
void* GSetRemove(GSet *s, int iElem);

// Function to remove the element 'elem' of the GSet
// Return the data pointed to by the removed element
// The GSetElem is freed and *elem == NULL after calling this function
// Return null if arguments are invalid
void* GSetRemoveElem(GSet *s, GSetElem **elem);

// Function to remove the first element of the GSet pointing to 'data'
// Do nothing if arguments are invalid
void GSetRemoveFirst(GSet *s, void *data);

// Function to remove the last element of the GSet pointing to 'data'
// Do nothing if arguments are invalid
void GSetRemoveLast(GSet *s, void *data);

// Function to remove all the selement of the GSet pointing to 'data'
// Do nothing if arguments are invalid
void GSetRemoveAll(GSet *s, void *data);

// Function to get the element at the 'iElem'-th position of the GSet
// without removing it
// Return the data pointed to by the element
// Return null if arguments are invalid
void* GSetGet(GSet *s, int iElem);

// Function to get the element at the 'iElem'-th position of the GSet

```

```

// without removing it
// Return the GSetElem
// Return null if arguments are invalid
GSetElem* GSetGetElem(GSet *s, int iElem);

// Function to get the index of the first element of the GSet
// which point to 'data'
// Return -1 if arguments are invalid or 'data' is not in the GSet
int GSetGetIndexFirst(GSet *s, void *data);

// Function to get the index of the last element of the GSet
// which point to 'data'
// Return -1 if arguments are invalid or 'data' is not in the GSet
int GSetGetIndexLast(GSet *s, void *data);

// Function to sort the element of the gset in increasing order of
// _sortVal
// Do nothing if arguments are invalid or the sort failed
void GSetSort(GSet *s);

// Merge the GSet '*r' at the end of the GSet '*s'
// '*r' and '*s' can be empty
// After calling this function (*r == null) and *r is freed
// Do nothing if arguments are invalid
void GSetMerge(GSet **s, GSet **r);

// Split the GSet 's' at the GSetElem 'e'
// 'e' must be an element of 's'
// Return a new GSet starting with 'e', or NULL if memory allocation
// failed or arguments are invalid
GSet* GSetSplit(GSet *s, GSetElem *e);

// Switch the 'iElem'-th and 'jElem'-th element of the set
// Do nothing if arguments are invalid
void GSetSwitch(GSet *s, int iElem, int jElem);

// Create a new GSetIterForward for the GSet 'set'
// The iterator is reset upon creation
// Return NULL if memory couldn't be allocated
GSetIterForward* GSetIterForwardCreate(GSet *set);

// Create a new GSetIterBackward for the GSet 'set'
// The iterator is reset upon creation
// Return NULL if memory couldn't be allocated
GSetIterBackward* GSetIterBackwardCreate(GSet *set);

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

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

// Clone a GSetIterForward
// Return NULL if arguments are invalid or memory allocation failed
GSetIterForward* GSetIterForwardClone(GSetIterForward *that);

// Clone a GSetIterBackward
// Return NULL if arguments are invalid or memory allocation failed
GSetIterBackward* GSetIterBackwardClone(GSetIterBackward *that);

```

```

// Reset the GSetIterForward to its starting position
// Do nothing if arguments are invalid
void GSetIterForwardReset(GSetIterForward *that);

// Reset the GSetIterBackward to its starting position
// Do nothing if arguments are invalid
void GSetIterBackwardReset(GSetIterBackward *that);

// Step the GSetIterForward
// Return false if arguments are invalid or we couldn't step
// Return true else
bool GSetIterForwardStep(GSetIterForward *that);

// Step the GSetIterBackward
// Return false if arguments are invalid or we couldn't step
// Return true else
bool GSetIterBackwardStep(GSetIterBackward *that);

// Apply a function to all elements of the GSet of the GSetIterForward
// The iterator is first reset, then the function is apply sequentially
// using the Step funciton of the iterator
// Do nothing if arguments are invalid
// 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
void GSetIterForwardApply(GSetIterForward *that,
    void(*fun)(void *data, void *param), void *param);

// Apply a function to all elements of the GSet of the GSetIterBackward
// The iterator is first reset, then the function is apply sequentially
// using the Step funciton of the iterator
// Do nothing if arguments are invalid
// 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
void GSetIterBackwardApply(GSetIterBackward *that,
    void(*fun)(void *data, void *param), void *param);

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterForwardIsFirst(GSetIterForward *that);

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterBackwardIsFirst(GSetIterBackward *that);

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterForwardIsLast(GSetIterForward *that);

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterBackwardIsLast(GSetIterBackward *that);

```

```

// Change the attached set of the iterator, and reset it
// Do nothing if argument is invalid
void GSetIterForwardSetGSet(GSetIterForward *that, GSet *set);

// Change the attached set of the iterator, and reset it
// Do nothing if argument is invalid
void GSetIterBackwardSetGSet(GSetIterBackward *that, GSet *set);

// Return the data currently pointed to by the iterator
// Return null if arguments are invalid
void* GSetIterForwardGet(GSetIterForward *that);

// Return the data currently pointed to by the iterator
// Return null if arguments are invalid
void* GSetIterBackwardGet(GSetIterBackward *that);

// Return the element currently pointed to by the iterator
// Return null if arguments are invalid
GSetElem* GSetIterForwardGetElem(GSetIterForward *that);

// Return the element currently pointed to by the iterator
// Return null if arguments are invalid
GSetElem* GSetIterBackwardGetElem(GSetIterBackward *that);

#endif

```

## 2 Code

```

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

// ===== Include =====
#include "gset.h"

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

// Function to create a new GSet,
// Return a pointer toward the new GSet, or null if it couldn't
// create the GSet
GSet* GSetCreate(void) {
    // Allocate memory for the GSet
    GSet *s = (GSet*)malloc(sizeof(GSet));
    // If we couldn't allocate memory return null
    if (s == NULL) return NULL;
    // 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, or null if it couldn't
// clone the GSet
GSet* GSetClone(GSet *s) {
    // If the arguments are invalid, return NULL
    if (s == NULL) return NULL;
    // Create the clone

```

```

GSet *c = GSetCreate();
// If the clone could be created
if (c != NULL) {
    // Set a pointer to the head of the set
    GSetElem *ptr = s->_head;
    // While the pointer is not at the end of the set
    while (ptr != NULL) {
        // Append the data of the current pointer to the clone
        GSetAppend(c, ptr->_data);
        // Copy the sort value
        c->_tail->_sortVal = ptr->_sortVal;
        // Move the pointer to the next element
        ptr = ptr->_next;
    }
}
// Return the clone
return c;
}

// Function to free the memory used by the GSet
void GSetFree(GSet **s) {
    // If the arguments are invalid, stop
    if (s == NULL || *s == NULL) return;
    // Empty the GSet
    GSetFlush(*s);
    // Free the memory
    free(*s);
    // Set the pointer to null
    *s = NULL;
}

// Function to empty the GSet
void GSetFlush(GSet *s) {
    // If the arguments are invalid, stop
    if (s == NULL) return;
    // Pop element until the GSet is null
    void *p = NULL;
    while (s->_nbElem > 0) p = GSetPop(s);
    // To avoid warning
    p = p;
}

// 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 *s, FILE* stream,
    void(*printData)(void *data, FILE *stream), char *sep) {
    // If the arguments are invalid, stop
    if (s == NULL || stream == NULL ||
        sep == NULL) return;
    // Set a pointer to the head element
    GSetElem *p = s->_head;
    // While the pointer hasn't reach the end
    while (p != NULL) {
        // If there is a print function for the data
        if (printData != NULL) {
            // Use the argument function to print the data of the
            // current element
            (*printData)(p->_data, stream);
        }
        // Else, there is no print function for the data
    }
}

```



```

    } else {
        // Print the pointer value instead
        fprintf(stream, "%p", p->_data);
    }
    // Flush the stream
    fflush(stream);
    // Move to the next element
    p = p->_next;
    // If there is a next element
    if (p != NULL)
        // Print the separator
        fprintf(stream, "%s", sep);
}
}

// Function to insert an element pointing toward 'data' at the
// head of the GSet
// Do nothing if arguments are invalid
void GSetPush(GSet *s, void* data) {
    // If the arguments are invalid, stop
    if (s == NULL || data == NULL) return;
    // Allocate memory for the new element
    GSetElem *e = (GSetElem*)malloc(sizeof(GSetElem));
    // If we could allocate memory
    if (e != NULL) {
        // Memorize the pointer toward data
        e->_data = data;
        // By default set the sorting value to 0.0
        e->_sortVal = 0.0;
        // Add the element at the head of the GSet
        e->_prev = NULL;
        if (s->_head != NULL) s->_head->_prev = e;
        e->_next = s->_head;
        s->_head = e;
        if (s->_tail == NULL) s->_tail = e;
        // Increment the number of elements in the GSet
        ++(s->_nbElem);
    }
}

// Function to insert an element pointing toward 'data' at the
// position defined by 'v' sorting the set in decreasing order
// Do nothing if arguments are invalid
void GSetAddSort(GSet *s, void* data, double v) {
    // If the arguments are invalid, stop
    if (s == NULL || data == NULL) return;
    // Allocate memory for the new element
    GSetElem *e = (GSetElem*)malloc(sizeof(GSetElem));
    // If we could allocate memory
    if (e != NULL) {
        // Memorize the pointer toward data
        e->_data = data;
        // Memorize the sorting value
        e->_sortVal = v;
        // If the GSet is empty
        if (s->_nbElem == 0) {
            // Add the element at the head of the GSet
            s->_head = e;
            s->_tail = e;
            e->_next = NULL;
            e->_prev = NULL;
        } else {

```

```

    // Set a pointer to the head of the GSet
    GSetElem *p = s->_head;
    // While the pointed element has a greater value than the
    // new element, move the pointer to the next element
    while (p != NULL && p->_sortVal >= v) p = p->_next;
    // Set the next element of the new element to the current element
    e->_next = p;
    // If the current element is not null
    if (p != NULL) {
        // Insert the new element inside the list of elements before p
        e->_prev = p->_prev;
        if (p->_prev != NULL)
            p->_prev->_next = e;
        else
            s->_head = e;
        p->_prev = e;
    } // Else, if the current element is null
    } else {
        // Insert the new element at the tail of the GSet
        e->_prev = s->_tail;
        if (s->_tail != NULL) s->_tail->_next = e;
        s->_tail = e;
        if (s->_head == NULL) s->_head = e;
    }
}
// Increment the number of elements
++(s->_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
// Do nothing if arguments are invalid
void GSetInsert(GSet *s, void* data, int iElem) {
    // If the arguments are invalid, stop
    if (s == NULL || data == NULL || iElem < 0) return;
    // If iElem is greater than the number of elements, append
    // elements pointing toward null data to fill in the gap
    while (iElem > s->_nbElem)
        GSetAppend(s, NULL);
    // If iElem is in the list of element or at the tail
    if (iElem <= s->_nbElem + 1) {
        // If the insert position is the head
        if (iElem == 0) {
            // Push the data
            GSetPush(s, data);
        } // Else, if the insert position is the tail
        } else if (iElem == s->_nbElem) {
            // Append data
            GSetAppend(s, data);
        } // Else, the insert position is inside the list
        } else {
            // Allocate memory for the new element
            GSetElem *e = (GSetElem*)malloc(sizeof(GSetElem));
            // If we could allocate memory
            if (e != NULL) {
                // Memorize the pointer toward data
                e->_data = data;
                // By default set the sorting value to 0.0
                e->_sortVal = 0.0;
            }
        }
    }
}

```

```

        // Set a pointer toward the head of the GSet
        GSetElem *p = s->_head;
        // Move the pointer to the iElem-th element
        for (int i = iElem; i > 0 && p != NULL; --i, p = p->_next);
        // Insert the element before the pointer
        e->_next = p;
        e->_prev = p->_prev;
        p->_prev = e;
        e->_prev->_next = e;
        // Increment the number of elements
        ++(s->_nbElem);
    }
}
}

// Function to insert an element pointing toward 'data' at the
// tail of the GSet
// Do nothing if arguments are invalid
void GSetAppend(GSet *s, void* data) {
    // If the arguments are invalid, stop
    if (s == NULL) return;
    GSetElem *e = (GSetElem*)malloc(sizeof(GSetElem));
    if (e != NULL) {
        e->_data = data;
        e->_sortVal = 0.0;
        e->_prev = s->_tail;
        e->_next = NULL;
        if (s->_tail != NULL) s->_tail->_next = e;
        s->_tail = e;
        if (s->_head == NULL) s->_head = e;
        ++(s->_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
// Return null if arguments are invalid
void* GSetPop(GSet *s) {
    // If the arguments are invalid, return null
    if (s == NULL) return NULL;
    void *ret = NULL;
    GSetElem *p = s->_head;
    if (p != NULL) {
        ret = p->_data;
        s->_head = p->_next;
        if (p->_next != NULL) p->_next->_prev = NULL;
        p->_next = NULL;
        p->_data = NULL;
        if (s->_tail == p) s->_tail = NULL;
        free(p);
        --(s->_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
// Return null if arguments are invalid
void* GSetDrop(GSet *s) {

```

```

// If the arguments are invalid, return null
if (s == NULL) return NULL;
void *ret = NULL;
GSetElem *p = s->_tail;
if (p != NULL) {
    ret = p->_data;
    s->_tail = p->_prev;
    if (p->_prev != NULL) p->_prev->_next = NULL;
    p->_prev = NULL;
    p->_data = NULL;
    if (s->_head == p) s->_head = NULL;
    free(p);
    --(s->_nbElem);
}
return ret;
}

// Function to remove the element at the 'iElem'-th position of the GSet
// Return the data pointed to by the removed element
// Return null if arguments are invalid
void* GSetRemove(GSet *s, int iElem) {
    // If the arguments are invalid, return null
    if (s == NULL) return NULL;
    // Variable to memorize the return value
    void *ret = NULL;
    // If iElem is a valid index
    if (iElem >= 0 && iElem < s->_nbElem) {
        // Set a pointer to the head of the Gset
        GSetElem *p = s->_head;
        // Move the pointer to the iElem-th element
        for (int i = iElem; i > 0 && p != NULL; --i, p = p->_next);
        // Memorize the data at iElem-th position
        ret = p->_data;
        // Remove the element
        if (p->_next != NULL) p->_next->_prev = p->_prev;
        if (p->_prev != NULL) p->_prev->_next = p->_next;
        if (s->_head == p) s->_head = p->_next;
        if (s->_tail == p) s->_tail = p->_prev;
        p->_next = NULL;
        p->_prev = NULL;
        p->_data = NULL;
        free(p);
        // Decrement the number of elements
        --(s->_nbElem);
    }
    // Return the data
    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
// Return null if arguments are invalid
void* GSetRemoveElem(GSet *s, GSetElem **elem) {
    // If the arguments are invalid, return null
    if (s == NULL || elem == NULL || *elem == NULL) return NULL;
    // Variable to memorize the return value
    void *ret = NULL;
    // Memorize the data at iElem-th position
    ret = (*elem)->_data;
    // Remove the element
    if ((*elem)->_next != NULL) (*elem)->_next->_prev = (*elem)->_prev;

```

```

    if ((*elem)->_prev != NULL) (*elem)->_prev->_next = (*elem)->_next;
    if (s->_head == (*elem)) s->_head = (*elem)->_next;
    if (s->_tail == (*elem)) s->_tail = (*elem)->_prev;
    (*elem)->_next = NULL;
    (*elem)->_prev = NULL;
    (*elem)->_data = NULL;
    free((*elem));
    *elem = NULL;
    // Decrement the number of elements
    --(s->_nbElem);
    // Return the data
    return ret;
}

```

```

// Function to remove the first element of the GSet pointing to 'data'
// Do nothing if arguments are invalid
void GSetRemoveFirst(GSet *s, void *data) {
    // If the arguments are invalid, stop
    if (s == NULL) return;
    // Set a pointer toward the head of the GSet
    GSetElem *p = s->_head;
    // Loop on elements until we have found the
    // requested data or reached the end of the list
    while (p != NULL && p->_data != data) {
        p = p->_next;
    }
    // If the pointer is null it means the data wasn't in the GSet
    if (p != NULL) {
        // Remove the element
        if (p->_next != NULL) p->_next->_prev = p->_prev;
        if (p->_prev != NULL) p->_prev->_next = p->_next;
        if (s->_head == p) s->_head = p->_next;
        if (s->_tail == p) s->_tail = p->_prev;
        p->_next = NULL;
        p->_prev = NULL;
        p->_data = NULL;
        free(p);
        // Decrement the number of elements
        --(s->_nbElem);
    }
}

```

```

// Function to remove the last element of the GSet pointing to 'data'
// Do nothing if arguments are invalid
void GSetRemoveLast(GSet *s, void *data) {
    // If the arguments are invalid, stop
    if (s == NULL) return;
    // Set a pointer toward the tail of the GSet
    GSetElem *p = s->_tail;
    // Loop on elements until we have found the
    // requested data or reached the head of the list
    while (p != NULL && p->_data != data) {
        p = p->_prev;
    }
    // If the pointer is null it means the data wasn't in the GSet
    if (p != NULL) {
        // Remove the element
        if (p->_next != NULL) p->_next->_prev = p->_prev;
        if (p->_prev != NULL) p->_prev->_next = p->_next;
        if (s->_head == p) s->_head = p->_next;
        if (s->_tail == p) s->_tail = p->_prev;
    }
}

```

```

    p->_next = NULL;
    p->_prev = NULL;
    p->_data = NULL;
    free(p);
    // Decrement the number of elements
    --(s->_nbElem);
}
}

// Function to remove all the selement of the GSet pointing to 'data'
// Do nothing if arguments are invalid
void GSetRemoveAll(GSet *s, void *data) {
    // If the arguments are invalid, stop
    if (s == NULL) return;
    // Set a pointer toward the tail of the GSet
    GSetElem *p = s->_tail;
    // Loop on elements until we reached the head of the list
    while (p != NULL) {
        // If the element points toward data
        if (p->_data == data) {
            // Memorize the previous element before deleting
            GSetElem *prev = p->_prev;
            // Remove the element
            if (p->_next != NULL) p->_next->_prev = p->_prev;
            if (p->_prev != NULL) p->_prev->_next = p->_next;
            if (s->_head == p) s->_head = p->_next;
            if (s->_tail == p) s->_tail = p->_prev;
            p->_next = NULL;
            p->_prev = NULL;
            p->_data = NULL;
            free(p);
            // Decrement the number of elements
            --(s->_nbElem);
            // Continue with previous element
            p = prev;
        } else {
            // Continue with previous element
            p = p->_prev;
        }
    }
}

// Function to get the element at the 'iElem'-th position of the GSet
// without removing it
// Return the data pointed to by the element
// Return null if arguments are invalid
void* GSetGet(GSet *s, int iElem) {
    // If the arguments are invalid, return null
    if (s == NULL) return NULL;
    // Set a pointer for the return value
    void *ret = NULL;
    // Get the iElem-th element
    GSetElem* e = GSetGetElem(s, iElem);
    // If we could ge the element
    if (e != NULL)
        // Get the data of the element
        ret = e->_data;
    // Return the data
    return ret;
}

```

```

// Function to get the element at the 'iElem'-th position of the GSet
// without removing it
// Return the GSetElem
// Return null if arguments are invalid
GSetElem* GSetGetElem(GSet *s, int iElem) {
    // If the arguments are invalid, return null
    if (s == NULL) return NULL;
    // Set a pointer for the return value
    GSetElem *ret = NULL;
    // If iElem is a valid index
    if (iElem >= 0 && iElem < s->_nbElem) {
        // Set the pointer to the head of the GSet
        ret = s->_head;
        // Move to the next element iElem times
        for (int i = iElem; i > 0 && ret != NULL; --i, ret = ret->_next);
    }
    // Return the element
    return ret;
}

// Function to get the index of the first element of the GSet
// which point to 'data'
// Return -1 if arguments are invalid or 'data' is not in the GSet
int GSetGetIndexFirst(GSet *s, void *data) {
    // If the arguments are invalid, return -1
    if (s == NULL) return -1;
    // Set a pointer toward the head of the GSet
    GSetElem *p = s->_head;
    // Set a variable to memorize index
    int index = 0;
    // Loop on elements until we have found the
    // requested data or reached the end of the list
    while (p != NULL && p->_data != data) {
        ++index;
        p = p->_next;
    }
    // If the pointer is null it means the data wasn't in the GSet
    if (p == NULL)
        index = -1;
    // Return the index
    return index;
}

// Function to get the index of the last element of the GSet
// which point to 'data'
// Return -1 if arguments are invalid or 'data' is not in the GSet
int GSetGetIndexLast(GSet *s, void *data) {
    // If the arguments are invalid, return -1
    if (s == NULL) return -1;
    // Set a pointer toward the tail of the GSet
    GSetElem *p = s->_tail;
    // Set a variable to memorize index
    int index = s->_nbElem - 1;
    // Loop on elements until we have found the
    // requested data or reached the head of the list
    while (p != NULL && p->_data != data) {
        --index;
        p = p->_prev;
    }
    // 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 sort the element of the gset in increasing order of
// _sortVal
// Do nothing if arguments are invalid or the sort failed
GSet* GSetSortRec(GSet **s);
void GSetSort(GSet *s) {
    // If the arguments are invalid, do nothing
    if (s == NULL) return;
    // Create a clone of the original set
    // GSetSortRec destroys its argument, so if something wrong we need
    // to still have the original set to give it back to the user
    GSet *clone = GSetClone(s);
    // 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(s);
        memcpy(s, res, sizeof(GSet));
        // Free the memory used by the result set
        free(res);
        res = NULL;
    }
}

GSet* GSetSortRec(GSet **s) {
    // If the arguments are invalid, do nothing
    if (s == NULL || *s == NULL) return NULL;
    // 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();
        // If we couldn't allocate memory
        if (lower == NULL || greater == NULL || res == NULL) {
            // Free memory and stop here
            GSetFree(&lower);
            GSetFree(&greater);
            GSetFree(&res);
            GSetFree(s);
            return NULL;
        }
        // Declare a variable to memorize the pivot, which is equal
        // to the sort value of the first element of the set
        float pivot = (*s)->_head->_sortVal;
        // Pop the pivot and put it in the result
        void *data = GSetPop(*s);
        GSetAppend(res, data);
        res->_head->_sortVal = pivot;
        // Pop all the elements one by one from the set
        while ((*s)->_nbElem != 0) {
            // Declare a variable to memorize the sort value of the head

```



```

    // element
    float val = (*s)->_head->_sortVal;
    // Pop the head element
    data = GSetPop(*s);
    // If the popped element has a sort value lower than the pivot
    if (val < pivot) {
        // Insert it in the lower set
        GSetAppend(lower, data);
        // Copy the sort value
        lower->_tail->_sortVal = val;
    }
    // Else, the popped element has a sort value greater then or
    // equal to the pivot
    else {
        // Insert it in the greater set
        GSetAppend(greater, data);
        // Copy the sort value
        greater->_tail->_sortVal = val;
    }
}
// At the end of the loop the original set is empty and we
// don't need it anymore
GSetFree(s);
// Sort the two half
GSet *sortedLower = GSetSortRec(&lower);
GSet *sortedGreater = GSetSortRec(&greater);
if (sortedLower == NULL || sortedGreater == NULL) {
    // Free memory and stop here
    GSetFree(&sortedLower);
    GSetFree(&sortedGreater);
    GSetFree(&res);
    return NULL;
}
// Merge back the sorted two half and the pivot
GSetMerge(&sortedLower, &res);
GSetMerge(&sortedLower, &sortedGreater);
res = sortedLower;
}
// Return the result
return res;
}

// Merge the GSet '*r' at the end of the GSet '*s'
// '*r' and '*s' can be empty
// After calling this function (*r == null) and *r is freed
// Do nothing if arguments are invalid
void GSetMerge(GSet **s, GSet **r) {
    // Check arguments
    if (s == NULL || r == NULL || *r == NULL)
        return;
    // If the set s is empty
    if (*s == NULL || (*s)->_head == NULL) {
        GSetFree(s);
        *s = *r;
        *r = NULL;
    }
    // Else, if the set r is empty
    else if ((*r)->_head == NULL) {
        GSetFree(r);
    }
    // Else, s and r are both not empty
    else {
        // Add r to the tail of s
        (*s)->_tail->_next = (*r)->_head;
        // Add s to the head of r

```

```

    (*r)->_head->_prev = (*s)->_tail;
    // Update the tail of s
    (*s)->_tail = (*r)->_tail;
    // Update the number of element of s
    (*s)->_nbElem += (*r)->_nbElem;
    // Free memory used by r
    free(*r);
    // Set r to null
    *r = NULL;
}
}

// Split the GSet 's' at the GSetElem 'e'
// 'e' must be an element of 's'
// Return a new GSet starting with 'e', or NULL if memory allocation
// failed or arguments are invalid
GSet* GSetSplit(GSet *s, GSetElem *e) {
    // Check arguments
    if (s == NULL || e == NULL)
        return NULL;
    // Check that e is an element of s
    // Declare a variable to count element before e in s
    int nb = 0;
    // If e is not the head of s
    if (s->_head != e) {
        GSetElem *ptr = e;
        // While there is an element before e
        do {
            // Increment the number of element
            ++nb;
            // Move to the previous element
            ptr = ptr->_prev;
        } while (ptr != NULL && ptr != s->_head);
        // If we have reached an element without previous element, this
        // element is not the head of S, meaning e is not in the set s
        if (ptr == NULL) {
            // Stop here
            return NULL;
        }
    }
    // Allocate memory for the result
    GSet *res = GSetCreate();
    // If we could allocate memory
    if (res != NULL) {
        // Set the head of res
        res->_head = e;
        // Set the tail of res
        res->_tail = s->_tail;
        // Set the number of element of res
        res->_nbElem = s->_nbElem - nb;
        // Set the tail of s
        s->_tail = e->_prev;
        // Set the number of element of s
        s->_nbElem = nb;
        // If s is empty
        if (nb == 0) {
            // Update head
            s->_head = NULL;
        }
        // Else, s is not empty
    } else {
        // Disconnect the tail of s
        s->_tail->_next = NULL;
    }
}

```

```

    }
    // Disconnect the head of res
    res->_head->_prev = NULL;
}
// Return the result
return res;
}

// Switch the 'iElem'-th and 'jElem'-th element of the set
// Do nothing if arguments are invalid
void GSetSwitch(GSet *s, int iElem, int jElem) {
    // Check arguments
    if (s == NULL || iElem < 0 || iElem >= s->_nbElem ||
        jElem < 0 || jElem >= s->_nbElem || iElem == jElem)
        return;
    // Get the two elements
    GSetElem *iPtr = GSetGetElem(s, iElem);
    GSetElem *jPtr = GSetGetElem(s, jElem);
    // If we could get the two elements
    if (iPtr != NULL && jPtr != NULL) {
        // Switch the elements
        float v = iPtr->_sortVal;
        iPtr->_sortVal = jPtr->_sortVal;
        jPtr->_sortVal = v;
        void *dat = iPtr->_data;
        iPtr->_data = jPtr->_data;
        jPtr->_data = dat;
    }
}

// Create a new GSetIterForward for the GSet 'set'
// The iterator is reset upon creation
// Return NULL if memory couldn't be allocated
GSetIterForward* GSetIterForwardCreate(GSet *set) {
    // Check argument
    if (set == NULL)
        return NULL;
    // Allocate memory
    GSetIterForward *ret =
        (GSetIterForward*)malloc(sizeof(GSetIterForward));
    // If we could allocate memory
    if (ret != NULL) {
        // Set properties
        ret->_set = set;
        ret->_curElem = set->_head;
    }
    // Return the new iterator
    return ret;
}

// Create a new GSetIterBackward for the GSet 'set'
// The iterator is reset upon creation
// Return NULL if memory couldn't be allocated
GSetIterBackward* GSetIterBackwardCreate(GSet *set) {
    // Check argument
    if (set == NULL)
        return NULL;
    // Allocate memory
    GSetIterBackward *ret =
        (GSetIterBackward*)malloc(sizeof(GSetIterBackward));
    // If we could allocate memory
    if (ret != NULL) {

```

```

    // 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
// Return NULL if arguments are invalid or memory allocation failed
GSetIterForward* GSetIterForwardClone(GSetIterForward *that) {
    // Check argument
    if (that == NULL)
        return NULL;
    // Create the clone
    GSetIterForward *ret = GSetIterForwardCreate(that->_set);
    // If we could allocate memory
    if (ret != NULL) {
        ret->_curElem = that->_curElem;
    }
    // return the clone
    return ret;
}

// Clone a GSetIterBackward
// Return NULL if arguments are invalid or memory allocation failed
GSetIterBackward* GSetIterBackwardClone(GSetIterBackward *that) {
    // Check argument
    if (that == NULL)
        return NULL;
    // Create the clone
    GSetIterBackward *ret = GSetIterBackwardCreate(that->_set);
    // If we could allocate memory
    if (ret != NULL) {
        ret->_curElem = that->_curElem;
    }
    // return the clone
    return ret;
}

```

```

}

// Reset the GSetIterForward to its starting position
// Do nothing if arguments are invalid
void GSetIterForwardReset(GSetIterForward *that) {
    // Check argument
    if (that == NULL)
        return;
    // Step
    that->_curElem = that->_set->_head;
}

// Reset the GSetIterBackward to its starting position
// Do nothing if arguments are invalid
void GSetIterBackwardReset(GSetIterBackward *that) {
    // Check argument
    if (that == NULL)
        return;
    // Step
    that->_curElem = that->_set->_tail;
}

// Step the GSetIterForward
// Return false if arguments are invalid or we couldn't step
// Return true else
bool GSetIterForwardStep(GSetIterForward *that) {
    // Check argument
    if (that == NULL)
        return false;
    // Step
    if (that->_curElem->_next != NULL)
        that->_curElem = that->_curElem->_next;
    else
        return false;
    return true;
}

// Step the GSetIterBackward
// Return false if arguments are invalid or we couldn't step
// Return true else
bool GSetIterBackwardStep(GSetIterBackward *that) {
    // Check argument
    if (that == NULL)
        return false;
    // Step
    if (that->_curElem->_prev != NULL)
        that->_curElem = that->_curElem->_prev;
    else
        return false;
    return true;
}

// Apply a function to all elements of the GSet of the GSetIterForward
// The iterator is first reset, then the function is apply sequentially
// using the Step funciton of the iterator
// Do nothing if arguments are invalid
// 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
void GSetIterForwardApply(GSetIterForward *that,
    void(*fun)(void *data, void *param), void *param) {
    // Check arguments

```

```

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

// Apply a function to all elements of the GSet of the GSetIterBackward
// The iterator is first reset, then the function is apply sequentially
// using the Step function of the iterator
// Do nothing if arguments are invalid
// 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
void GSetIterBackwardApply(GSetIterBackward *that,
    void(*fun)(void *data, void *param), void *param) {
    // Check arguments
    if (that == NULL || fun == NULL)
        return;
    // Reset the iterator
    GSetIterReset(that);
    // If the set is not empty
    if (that->_curElem != NULL) {
        // Loop on element
        do {
            // Apply the user function
            fun(that->_curElem->_data, param);
        } while (GSetIterStep(that) == true);
    }
}

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterForwardIsFirst(GSetIterForward *that) {
    // Check argument
    if (that == NULL)
        return false;
    if (that->_curElem == that->_set->_head)
        return true;
    else
        return false;
}

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterBackwardIsFirst(GSetIterBackward *that) {
    // Check argument
    if (that == NULL)
        return false;
    if (that->_curElem == that->_set->_tail)
        return true;
}

```

```

    else
        return false;
}

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterForwardIsLast(GSetIterForward *that) {
    // Check argument
    if (that == NULL)
        return false;
    if (that->_curElem == that->_set->_tail)
        return true;
    else
        return false;
}

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GSet)
// Return false if arguments are invalid
// Return false else
bool GSetIterBackwardIsLast(GSetIterBackward *that) {
    // Check argument
    if (that == NULL)
        return false;
    if (that->_curElem == that->_set->_head)
        return true;
    else
        return false;
}

// Change the attached set of the iterator, and reset it
// Do nothing if argument is invalid
void GSetIterForwardSetGSet(GSetIterForward *that, GSet *set) {
    // Check arguments
    if (that == NULL || set == NULL)
        return;
    // Set the GSet
    that->_set = set;
    // Reset the iterator
    GSetIterReset(that);
}

// Change the attached set of the iterator, and reset it
// Do nothing if argument is invalid
void GSetIterBackwardSetGSet(GSetIterBackward *that, GSet *set) {
    // Check arguments
    if (that == NULL || set == NULL)
        return;
    // Set the GSet
    that->_set = set;
    // Reset the iterator
    GSetIterReset(that);
}

// Return the data currently pointed to by the iterator
// Return null if arguments are invalid
void* GSetIterForwardGet(GSetIterForward *that) {
    // Check argument
    if (that == NULL)
        return NULL;

```

```

    // Return the data
    return that->_curElem->_data;
}

// Return the data currently pointed to by the iterator
// Return null if arguments are invalid
void* GSetIterBackwardGet(GSetIterBackward *that) {
    // Check argument
    if (that == NULL)
        return NULL;
    // Return the data
    return that->_curElem->_data;
}

// Return the element currently pointed to by the iterator
// Return null if arguments are invalid
GSetElem* GSetIterForwardGetElem(GSetIterForward *that) {
    // Check argument
    if (that == NULL)
        return NULL;
    // Return the data
    return that->_curElem;
}

// Return the element currently pointed to by the iterator
// Return null if arguments are invalid
GSetElem* GSetIterBackwardGetElem(GSetIterBackward *that) {
    // Check argument
    if (that == NULL)
        return NULL;
    // Return the data
    return that->_curElem;
}

```

### 3 Makefile

```

OPTIONS_DEBUG=-gddb -g3 -Wall
OPTIONS_RELEASE=-O3
OPTIONS=$(OPTIONS_RELEASE)

all : gset

clean:
rm *.o gset

gset : gset_main.o gset.o Makefile
gcc gset_main.o gset.o $(OPTIONS) -o gset -lm

gset_main.o : gset.h gset_main.c Makefile
gcc -c gset_main.c $(OPTIONS)

gset.o : gset.c gset.h Makefile
gcc -c gset.c $(OPTIONS)

install:
cp gset.h ../Include; cp gset.o ../Include

valgrind :

```



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

## 4 Usage

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "gset.h"

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

struct Test {
    int v;
};

void TestPrint(void *t, FILE *stream) {
    if (t == NULL) {
        fprintf(stream, "(null)");
    } else {
        fprintf(stream, "%d", ((struct Test*)t)->v);
    }
}

void TestApply(void *t, void *p) {
    struct Test *data = (struct Test*)t;
    struct Test *param = (struct Test*)p;
    printf("applying %d on %d\n", param->v, data->v);
}

int main(int argc, char **argv) {
    // Initialize the random generator
    time_t seed = time(NULL);
    srand(seed);

    GSet *theSet = GSetCreate();
    fprintf(stdout, "Created the set, nb elem : %d\n", theSet->_nbElem);
    struct Test data[4];
    for (int i = 0; i < 4; ++i) data[i].v = i;

    GSetPush(theSet, &(data[1]));
    GSetPush(theSet, &(data[3]));
    GSetPush(theSet, &(data[2]));
    fprintf(stdout, "Pushed [1,3,2], nb elem : %d\n", theSet->_nbElem);
    fprintf(stdout, "Print GSet:\n");
    GSetPrint(theSet, stdout, &TestPrint, (char*)" ", "");
    fprintf(stdout, "\n");

    fprintf(stdout, "Pop elements :\n");
    while (theSet->_nbElem > 0) {
        struct Test *p = (struct Test *)GSetPop(theSet);
        fprintf(stdout, "%d, ", p->v);
    }
    fprintf(stdout, "\n");

    GSetPush(theSet, &(data[1]));
    GSetPush(theSet, &(data[3]));
    GSetPush(theSet, &(data[2]));
    fprintf(stdout, "Push back and drop elements :\n");
    while (theSet->_nbElem > 0) {
```

```

    struct Test *p = (struct Test *)GSetDrop(theSet);
    fprintf(stdout, "%d, ", p->v);
}
fprintf(stdout, "\n");

GSetAppend(theSet, &(amp;data[1]));
GSetAppend(theSet, &(amp;data[3]));
GSetAppend(theSet, &(amp;data[2]));
fprintf(stdout, "Append back and pop elements :\n");
while (theSet->_nbElem > 0) {
    struct Test *p = (struct Test *)GSetPop(theSet);
    fprintf(stdout, "%d, ", p->v);
}
fprintf(stdout, "\n");

GSetAppend(theSet, &(amp;data[1]));
GSetAppend(theSet, &(amp;data[3]));
GSetAppend(theSet, &(amp;data[2]));
fprintf(stdout, "Append back and drop elements :\n");
while (theSet->_nbElem > 0) {
    struct Test *p = (struct Test *)GSetDrop(theSet);
    fprintf(stdout, "%d, ", p->v);
}
fprintf(stdout, "\n");

GSetAddSort(theSet, &(amp;data[2]), data[2].v);
GSetAddSort(theSet, &(amp;data[3]), data[3].v);
GSetAddSort(theSet, &(amp;data[1]), data[1].v);
fprintf(stdout, "Add sort (decreasing) [2,3,1] and get elements :\n");
for (int i = 0; i < theSet->_nbElem; ++i) {
    struct Test *p = (struct Test *)GSetGet(theSet, i);
    fprintf(stdout, "%d, ", p->v);
}
fprintf(stdout, "\n");

GSetSwitch(theSet, 0, 2);
fprintf(stdout, "Switch 1st and 3rd elements:\n");
for (int i = 0; i < theSet->_nbElem; ++i) {
    struct Test *p = (struct Test *)GSetGet(theSet, i);
    fprintf(stdout, "%d, ", p->v);
}
fprintf(stdout, "\n");

GSetInsert(theSet, &(amp;data[0]), 0);
GSetInsert(theSet, &(amp;data[0]), 2);
GSetInsert(theSet, &(amp;data[0]), 8);
fprintf(stdout, "Insert 0 at 0, 2, 8 and get elements :\n");
for (int i = 0; i < theSet->_nbElem; ++i) {
    struct Test *p = (struct Test *)GSetGet(theSet, i);
    TestPrint(p, stdout);
    fprintf(stdout, ", ");
}
fprintf(stdout, "\n");

fprintf(stdout, "Split the set at 1:\n");
GSetElem *splitElem = GSetGetElem(theSet, 4);
GSet *split = GSetSplit(theSet, splitElem);
GSetPrint(theSet, stdout, TestPrint, ", ");
fprintf(stdout, " and ");
GSetPrint(split, stdout, TestPrint, ", ");
fprintf(stdout, "\n");

```

```

fprintf(stdout, "Merge back the set:\n");
GSetMerge(&theSet, &split);
GSetPrint(theSet, stdout, TestPrint, " ");
fprintf(stdout, "\n");

GSet *clone = GSetClone(theSet);
fprintf(stdout, "Clone the set and print it:\n");
GSetPrint(clone, stdout, &TestPrint, (char*)" ", "");
fprintf(stdout, "\n");
GSetFree(&clone);

fprintf(stdout, "Sort the elements, before:\n");
GSetElem *elem = theSet->_head;
while (elem != NULL) {
    elem->_sortVal = rnd();
    fprintf(stdout, "%.3f,", elem->_sortVal);
    elem = elem->_next;
}
fprintf(stdout, "\n");
GSetSort(theSet);
fprintf(stdout, "Sort the elements, after:\n");
elem = theSet->_head;
while (elem != NULL) {
    fprintf(stdout, "%.3f,", elem->_sortVal);
    elem = elem->_next;
}
fprintf(stdout, "\n");
for (int i = 0; i < theSet->_nbElem; ++i) {
    struct Test *p = (struct Test *)GSetGet(theSet, i);
    TestPrint(p, stdout);
    fprintf(stdout, " ");
}
fprintf(stdout, "\n");

GSetRemove(theSet, 7);
GSetRemove(theSet, 1);
GSetRemove(theSet, 0);
fprintf(stdout, "Remove at 7,1,0 and get elements :\n");
for (int i = 0; i < theSet->_nbElem; ++i) {
    struct Test *p = (struct Test *)GSetGet(theSet, i);
    TestPrint(p, stdout);
    fprintf(stdout, " ");
}
fprintf(stdout, "\n");

fprintf(stdout, "Index of first null data : %d\n",
    GSetGetIndexFirst(theSet, NULL));
fprintf(stdout, "Index of last null data : %d\n",
    GSetGetIndexLast(theSet, NULL));

GSetRemoveAll(theSet, NULL);
fprintf(stdout, "Delete all null and get elements :\n");
for (int i = 0; i < theSet->_nbElem; ++i) {
    struct Test *p = (struct Test *)GSetGet(theSet, i);
    TestPrint(p, stdout);
    fprintf(stdout, " ");
}
fprintf(stdout, "\n");

fprintf(stdout, "Test iterator forward\n");
GSetIterForward *iterf = GSetIterForwardCreate(theSet);
do {

```

```

    struct Test *p = (struct Test *)GSetIterGet(iterf);
    TestPrint(p, stdout);
    fprintf(stdout, ", ");
} while (GSetIterStep(iterf) == true);
fprintf(stdout, "\n");

fprintf(stdout, "Apply a function on all elements\n");
struct Test param;
param.v = 2;
GSetIterApply(iterf, TestApply, &param);
fprintf(stdout, "\n");
GSetIterFree(&iterf);

fprintf(stdout, "Test iterator backward\n");
GSetIterBackward *iterb = GSetIterBackwardCreate(theSet);
do {
    GSetElem *e = GSetIterGetElem(iterb);
    struct Test *p = (struct Test *)e->_data;
    TestPrint(p, stdout);
    fprintf(stdout, ", ");
} while (GSetIterStep(iterb) == true);
fprintf(stdout, "\n");
GSetIterFree(&iterb);

GSetFree(&theSet);
}

```

Output:

```

Created the set, nb elem : 0
Pushed [1,3,2], nb elem : 3
Print GSet:
2, 3, 1
Pop elements :
2, 3, 1,
Push back and drop elements :
1, 3, 2,
Append back and pop elements :
1, 3, 2,
Append back and drop elements :
2, 3, 1,
Add sort (decreasing) [2,3,1] and get elements :
3, 2, 1,
Switch 1st and 3rd elements:
1, 2, 3,
Insert 0 at 0, 2, 8 and get elements :
0, 1, 0, 2, 3, (null), (null), (null), 0,
Split the set at 1:
0, 1, 0, 2 and 3, (null), (null), (null), 0
Merge back the set:
0, 1, 0, 2, 3, (null), (null), (null), 0
Clone the set and print it:
0, 1, 0, 2, 3, (null), (null), (null), 0
Sort the elements, before:
0.981,0.867,0.697,0.762,0.871,0.394,0.284,0.143,0.292,
Sort the elements, after:
0.143,0.284,0.292,0.394,0.697,0.762,0.867,0.871,0.981,
(null), (null), 0, (null), 0, 2, 1, 3, 0,
Remove at 7,1,0 and get elements :

```

```
0, (null), 0, 2, 1, 0,  
Index of first null data : 1  
Index of last null data : 1  
Delete all null and get elements :  
0, 0, 2, 1, 0,  
Test iterator forward  
0, 0, 2, 1, 0,  
Apply a function on all elements  
applying 2 on 0  
applying 2 on 0  
applying 2 on 2  
applying 2 on 1  
applying 2 on 0  
  
Test iterator backward  
0, 1, 2, 0, 0,
```