

GSet

P. Baillehache

September 5, 2017

Contents

1	Interface	1
2	Code	4
3	Makefile	12
4	Usage	12

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.

1 Interface

```

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

// ===== Include =====
#include <stdlib.h>
#include <stdio.h>

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

// ===== 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;

// ===== 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();

// 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
// 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 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 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);

#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() {
    // 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);
            // 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
    int nbAddElement = iElem - s->_nbElem - 1;
    while (nbAddElement > 0) {
        GSetAppend(s, NULL);
        nbAddElement--;
    }
    // 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 (iElem == s->_nbElem + 1) {
            // Append data
            GSetAppend(s, data);
        } else {
            // Else, the insert position is inside the list
            // 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 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;
// 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 to the next element iElem times
    for (int i = iElem; i > 0 && p != NULL; --i, p = p->_next);
    // If the pointer is not null (in case the GSet is empty)
    if (p != NULL)
        // Memorize the data pointed to by the elem
        ret = p->_data;
}
// 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;
}

```

3 Makefile

```
OPTIONS_DEBUG=-ggdb -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)
```

4 Usage

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

struct Test {
    int v;
};

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

int main(int argc, char **argv) {
    GSet *theSet = GSetCreate();
    fprintf(stderr, "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(stderr, "Pushed [1,3,2], nb elem : %d\n", theSet->_nbElem);
    fprintf(stderr, "Print GSet:\n");
    GSetPrint(theSet, stdout, &TestPrint, (char*)"");
    fprintf(stderr, "\n");

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

```

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

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

GSetAppend(theSet, &(amp;data[1]));
GSetAppend(theSet, &(amp;data[3]));
GSetAppend(theSet, &(amp;data[2]));
fprintf(stderr, "Append back and drop elements :\n");
while (theSet->_nbElem > 0) {
    struct Test *p = (struct Test *)GSetDrop(theSet);
    fprintf(stderr, "%d, ", p->v);
}
fprintf(stderr, "\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(stderr, "Add sort [2,3,1] and get elements :\n");
for (int i = 0; i < theSet->_nbElem; ++i) {
    struct Test *p = (struct Test *)GSetGet(theSet, i);
    fprintf(stderr, "%d, ", p->v);
}
fprintf(stderr, "\n");

GSetInsert(theSet, &(amp;data[0]), 0);
GSetInsert(theSet, &(amp;data[0]), 2);
GSetInsert(theSet, &(amp;data[0]), 8);
fprintf(stderr, "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, stderr);
    fprintf(stderr, ", ");
}
fprintf(stderr, "\n");

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

GSetRemove(theSet, 7);
GSetRemove(theSet, 1);
GSetRemove(theSet, 0);
fprintf(stderr, "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, stderr);
        fprintf(stderr, ", ");
    }
    fprintf(stderr, "\n");

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

    GSetRemoveAll(theSet, NULL);
    fprintf(stderr, "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, stderr);
        fprintf(stderr, ", ");
    }
    fprintf(stderr, "\n");

    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 [2,3,1] and get elements :
3, 2, 1,
Insert 0 at 0, 2, 8 and get elements :
0, 3, 0, 2, 1, (null), (null), 0,
Clone the set and print it:
0, 3, 0, 2, 1, (null), (null), 0
Remove at 7,1,0 and get elements :
0, 2, 1, (null), (null),
Index of first null data : 3
Index of last null data : 4
Delete all null and get elements :
0, 2, 1,

```