

# GTree

P. Baillehache

April 20, 2018

## Contents

|          |                           |           |
|----------|---------------------------|-----------|
| <b>1</b> | <b>Interface</b>          | <b>2</b>  |
| <b>2</b> | <b>Code</b>               | <b>9</b>  |
| 2.1      | pbmath.c . . . . .        | 9         |
| 2.2      | pbmath-inline.c . . . . . | 16        |
| <b>3</b> | <b>Makefile</b>           | <b>22</b> |
| <b>4</b> | <b>Unit tests</b>         | <b>23</b> |
| <b>5</b> | <b>Unit tests output</b>  | <b>32</b> |

## Introduction

GTree is a C library providing structures and functions to manipulate tree structures.

A GTree is a structure containing a pointer toward its parent, a void\* pointer toward user's data and a GSet of subtrees. The GTree offers the same interface has a GSet to manipulate its subtrees. It also provides a function to cut the GTree from its parent.

The library provides also three iterators to run through the trees: GTreeIterDepth, GTreeIterBreadth, GTreeIterValue which step, respectively, in depth first order, breadth first order and value (sorting value of the GSet of subtrees) first order.

It uses the PBErr and GSet libraries.

## 1 Interface

```
// ===== GTREE.H =====

#ifndef GTREE_H
#define GTREE_H

// ===== Include =====

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "gset.h"

// ----- GTree

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

// ===== Data structure =====

typedef struct GTree {
    // Parent node
    GTree* _parent;
    // Branches
    // Branch cannot be null, if the user tries to add a null branch
    // nothing happen
    GSetGTree _subtrees;
    // User data
    void* _data;
} GTree;

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

// Create a new GTree
GTree* GTreeCreate(void);

// Create a new static GTree
GTree GTreeCreateStatic(void);

// Create a new GTree with user data 'data'
GTree* GTreeCreateData(void* data);

// Free the memory used by the GTree 'that'
// If 'that' is not a root node it is cut prior to be freed
// Subtrees are recursively freed
// User data must be freed by the user
void GTreeFree(GTree** that);

// Free the memory used by the static GTree 'that'
// If 'that' is not a root node it is cut prior to be freed
// Subtrees are recursively freed
// User data must be freed by the user
void GTreeFreeStatic(GTree* that);
```

```

// Get the user data of the GTree 'that'
#if BUILDMODE != 0
inline
#endif
void* GTreeData(GTree* that);

// Set the user data of the GTree 'that' to 'data'
#if BUILDMODE != 0
inline
#endif
void GTreeSetData(GTree* that, void* data);

// Get the set of subtrees of the GTree 'that'
#if BUILDMODE != 0
inline
#endif
GSetGTree* GTreeSubtrees(GTree* that);

// Disconnect the GTree 'that' from its parent
// If it has no parent, do nothing
void GTreeCut(GTree* that);

// Return true if the GTree 'that' is a root
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GTreeIsRoot(GTree* that);

// Return true if the GTree 'that' is a leaf
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GTreeIsLeaf(GTree* that);

// Return the parent of the GTree 'that'
#if BUILDMODE != 0
inline
#endif
GTree* GTreeParent(GTree* that);

// Return the number of subtrees of the GTree 'that' and their subtrees
// recursively
int GTreeGetSize(GTree* that);

// Wrapping of GSet functions
inline GTree* GTreeSubtree(GTree* that, int iSubtree) {
    return GSetGet(GTreeSubtrees(that), iSubtree);
}
inline GTree* GTreeFirstSubtree(GTree* that) {
    return GSetGetFirst(GTreeSubtrees(that));
}
inline GTree* GTreeLastSubtree(GTree* that) {
    return GSetGetLast(GTreeSubtrees(that));
}
inline GTree* GTreePopSubtree(GTree* that) {
    return GSetPop(GTreeSubtrees(that));
}
inline GTree* GTreeDropSubtree(GTree* that) {
    return GSetDrop(GTreeSubtrees(that));
}

```

```

}
inline GTree* GTreeRemoveSubtree(GTree* that, int iSubtree) {
    return GSetRemove((GSet*)GTreeSubtrees(that), iSubtree);
}

inline void GTreePushSubtree(GTree* that, GTree* tree) {
    if (!tree) return;
    GSetPush(GTreeSubtrees(that), tree);
    tree->_parent = that;
}

inline void GTreeAddSortSubTree(GTree* that, GTree* tree,
    float sortVal) {
    if (!tree) return;
    GSetAddSort(GTreeSubtrees(that), tree, sortVal);
    tree->_parent = that;
}

inline void GTreeInsertSubtree(GTree* that, GTree* tree, int pos) {
    if (!tree) return;
    GSetInsert(GTreeSubtrees(that), tree, pos);
    tree->_parent = that;
}

inline void GTreeAppendSubtree(GTree* that, GTree* tree) {
    if (!tree) return;
    GSetAppend(GTreeSubtrees(that), tree);
    tree->_parent = that;
}

inline void GTreePushData(GTree* that, void* data) {
    GTree* tree = GTreeCreateData(data);
    GSetPush(GTreeSubtrees(that), tree);
    tree->_parent = that;
}

inline void GTreeAddSortData(GTree* that, void* data,
    float sortVal) {
    GTree* tree = GTreeCreateData(data);
    GSetAddSort(GTreeSubtrees(that), tree, sortVal);
    tree->_parent = that;
}

inline void GTreeInsertData(GTree* that, void* data, int pos) {
    GTree* tree = GTreeCreateData(data);
    GSetInsert(GTreeSubtrees(that), tree, pos);
    tree->_parent = that;
}

inline void GTreeAppendData(GTree* that, void* data) {
    GTree* tree = GTreeCreateData(data);
    GSetAppend(GTreeSubtrees(that), tree);
    tree->_parent = that;
}

// ----- GTreeIter

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

// ===== Data structure =====

typedef struct GTreeIter {
    // Attached tree
    GTree* _tree;
    // Current position
    GSetElem* _curPos;
    // GSet to memorize nodes sequence
    // The node sequence doesn't include the root node of the attached tree

```

```

    GSetGTree _seq;
} GTreeIter;

typedef struct GTreeIterDepth {GTreeIter _iter;} GTreeIterDepth;
typedef struct GTreeIterBreadth {GTreeIter _iter;} GTreeIterBreadth;
typedef struct GTreeIterValue {GTreeIter _iter;} GTreeIterValue;

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

// Create a new GTreeIterDepth for the GTree 'tree'
GTreeIterDepth* GTreeIterDepthCreate(GTree* tree);

// Create a new static GTreeIterDepth for the GTree 'tree'
GTreeIterDepth GTreeIterDepthCreateStatic(GTree* tree);

// Create a new GTreeIterBreadth for the GTree 'tree'
GTreeIterBreadth* GTreeIterBreadthCreate(GTree* tree);

// Create a new static GTreeIterBreadth for the GTree 'tree'
GTreeIterBreadth GTreeIterBreadthCreateStatic(GTree* tree);

// Create a new GTreeIterValue for the GTree 'tree'
GTreeIterValue* GTreeIterValueCreate(GTree* tree);

// Create a new static GTreeIterValue for the GTree 'tree'
GTreeIterValue GTreeIterValueCreateStatic(GTree* tree);

// Update the GTreeIterDepth 'that' in case its attached GTree has been
// modified
// The node sequence doesn't include the root node of the attached tree
void GTreeIterDepthUpdate(GTreeIterDepth* that);

// Update the GTreeIterBreadth 'that' in case its attached GTree has
// been modified
// The node sequence doesn't include the root node of the attached tree
void GTreeIterBreadthUpdate(GTreeIterBreadth* that);

// Update the GTreeIterValue 'that' in case its attached GTree has been
// modified
// The node sequence doesn't include the root node of the attached tree
void GTreeIterValueUpdate(GTreeIterValue* that);

// Free the memory used by the iterator 'that'
void _GTreeIterFree(GTreeIter** that);

// Free the memory used by the static iterator 'that'
void _GTreeIterFreeStatic(GTreeIter* that);

// Reset the iterator 'that' at its start position
#if BUILDMODE != 0
inline
#endif
void _GTreeIterReset(GTreeIter* that);

// Reset the iterator 'that' at its end position
#if BUILDMODE != 0
inline
#endif
void _GTreeIterToEnd(GTreeIter* that);

// Step the iterator 'that' at its next position
// Return true if it could move to the next position

```

```

// Return false if it's already at the last position
#if BUILDMODE != 0
inline
#endif
bool _GTreeIterStep(GTreeIter* that);

// Step back the iterator 'that' at its next position
// Return true if it could move to the previous position
// Return false if it's already at the first position
#if BUILDMODE != 0
inline
#endif
bool _GTreeIterStepBack(GTreeIter* that);

// Apply a function to all elements' data of the GTree of the iterator
// The iterator is first reset, then the function is apply sequentially
// 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 _GTreeIterApply(GTreeIter* 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 GTree)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool _GTreeIterIsFirst(GTreeIter* that);

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

// Change the attached tree of the iterator, and reset it
#if BUILDMODE != 0
inline
#endif
void _GTreeIterDepthSetGTree(GTreeIterDepth* that, GTree* tree);
#if BUILDMODE != 0
inline
#endif
void _GTreeIterBreadthSetGTree(GTreeIterBreadth* that, GTree* tree);
#if BUILDMODE != 0
inline
#endif
void _GTreeIterValueSetGTree(GTreeIterValue* that, GTree* tree);

// Return the user data of the tree currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void* _GTreeIterGetData(GTreeIter* that);

```

```

// Return the tree currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GTree* _GTreeIterGetGTree(GTreeIter* that);

// Return the tree associated to the iterator 'that'
#if BUILDMODE != 0
inline
#endif
GTree* _GTreeIterGTree(GTreeIter* that);

// Return the sequence of the iterator
#if BUILDMODE != 0
inline
#endif
GSetGTree* _GTreeIterSeq(GTreeIter* that);

// ===== Polymorphism =====

#define GTreeIterFree(RefIter) _Generic(RefIter, \
    GTreeIter*: _GTreeIterFree, \
    GTreeIterDepth*: _GTreeIterFree, \
    GTreeIterBreadth*: _GTreeIterFree, \
    GTreeIterValue*: _GTreeIterFree, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(RefIter))

#define GTreeIterFreeStatic(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterFreeStatic, \
    GTreeIterDepth*: _GTreeIterFreeStatic, \
    GTreeIterBreadth*: _GTreeIterFreeStatic, \
    GTreeIterValue*: _GTreeIterFreeStatic, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterReset(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterReset, \
    GTreeIterDepth*: _GTreeIterReset, \
    GTreeIterBreadth*: _GTreeIterReset, \
    GTreeIterValue*: _GTreeIterReset, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterToEnd(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterToEnd, \
    GTreeIterDepth*: _GTreeIterToEnd, \
    GTreeIterBreadth*: _GTreeIterToEnd, \
    GTreeIterValue*: _GTreeIterToEnd, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterStep(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterStep, \
    GTreeIterDepth*: _GTreeIterStep, \
    GTreeIterBreadth*: _GTreeIterStep, \
    GTreeIterValue*: _GTreeIterStep, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterStepBack(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterStepBack, \
    GTreeIterDepth*: _GTreeIterStepBack, \
    GTreeIterBreadth*: _GTreeIterStepBack, \
    GTreeIterValue*: _GTreeIterStepBack, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

```

```

#define GTreeIterApply(Iter, Fun, Param) _Generic(Iter, \
    GTreeIter*: _GTreeIterApply, \
    GTreeIterDepth*: _GTreeIterApply, \
    GTreeIterBreadth*: _GTreeIterApply, \
    GTreeIterValue*: _GTreeIterApply, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter), Fun, Param)

#define GTreeIterIsFirst(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterIsFirst, \
    GTreeIterDepth*: _GTreeIterIsFirst, \
    GTreeIterBreadth*: _GTreeIterIsFirst, \
    GTreeIterValue*: _GTreeIterIsFirst, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterIsLast(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterIsLast, \
    GTreeIterDepth*: _GTreeIterIsLast, \
    GTreeIterBreadth*: _GTreeIterIsLast, \
    GTreeIterValue*: _GTreeIterIsLast, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterSetGTree(Iter, Tree) _Generic(Iter, \
    GTreeIterDepth*: _GTreeIterDepthSetGTree, \
    GTreeIterBreadth*: _GTreeIterBreadthSetGTree, \
    GTreeIterValue*: _GTreeIterValueSetGTree, \
    default: PBErrInvalidPolymorphism) (Iter, Tree)

#define GTreeIterGetData(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterGetData, \
    GTreeIterDepth*: _GTreeIterGetData, \
    GTreeIterBreadth*: _GTreeIterGetData, \
    GTreeIterValue*: _GTreeIterGetData, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterGTree(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterGTree, \
    GTreeIterDepth*: _GTreeIterGTree, \
    GTreeIterBreadth*: _GTreeIterGTree, \
    GTreeIterValue*: _GTreeIterGTree, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterGetGTree(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterGetGTree, \
    GTreeIterDepth*: _GTreeIterGetGTree, \
    GTreeIterBreadth*: _GTreeIterGetGTree, \
    GTreeIterValue*: _GTreeIterGetGTree, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

#define GTreeIterSeq(Iter) _Generic(Iter, \
    GTreeIter*: _GTreeIterSeq, \
    GTreeIterDepth*: _GTreeIterSeq, \
    GTreeIterBreadth*: _GTreeIterSeq, \
    GTreeIterValue*: _GTreeIterSeq, \
    default: PBErrInvalidPolymorphism) ((GTreeIter*)(Iter))

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

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

```



```
#endif
```

## 2 Code

### 2.1 pbmath.c

```
// ===== GTREE.C =====

// ===== Include =====

#include "gtree.h"
#if BUILDMODE == 0
#include "gtree-inline.c"
#endif

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

// Free the memory used by 'subtrees' recursively
void GTreeFreeRec(GSetGTree* subtrees);

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

// Create a new GTree
GTree* GTreeCreate(void) {
    // Declare the new tree
    GTree *that = PBErrMalloc(GTreeErr, sizeof(GTree));
    // Set properties
    that->_parent = NULL;
    that->_subtrees = GSetGTreeCreateStatic();
    that->_data = NULL;
    // Return the tree
    return that;
}

// Create a new static GTree
GTree GTreeCreateStatic(void) {
    // Declare the new tree
    GTree that;
    // Set properties
    that._parent = NULL;
    that._subtrees = GSetGTreeCreateStatic();
    that._data = NULL;
    // Return the tree
    return that;
}

// Create a new GTree with user data 'data'
GTree* GTreeCreateData(void* data) {
    // Declare the new tree
    GTree *that = PBErrMalloc(GTreeErr, sizeof(GTree));
    // Set properties
    that->_parent = NULL;
    that->_subtrees = GSetGTreeCreateStatic();
    that->_data = data;
    // Return the tree
    return that;
}
```

```

// Free the memory used by the GTree 'that'
// If 'that' is not a root node it is cut prior to be freed
// Subtrees are recursively freed
// User data must be freed by the user
void GTreeFree(GTree** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        // Nothing to do
        return;
    // If it's not a root node
    if (!GTreeIsRoot(*that))
        // Cut the tree
        GTreeCut(*that);
    // Free recursively the memory
    GTreeFreeRec(GTreeSubtrees(*that));
    free(*that);
    *that = NULL;
}

// Free the memory used by 'subtrees' recursively
void GTreeFreeRec(GSetGTree* subtrees) {
    while (GSetNbElem(subtrees) > 0) {
        GTree* tree = GSetPop(subtrees);
        GTreeFreeRec(GTreeSubtrees(tree));
        free(tree);
    }
}

// Free the memory used by the static GTree 'that'
// If 'that' is not a root node it is cut prior to be freed
// Subtrees are recursively freed
// User data must be freed by the user
void GTreeFreeStatic(GTree* that) {
    // Check argument
    if (that == NULL)
        // Nothing to do
        return;
    // If it's not a root node
    if (!GTreeIsRoot(that))
        // Cut the tree
        GTreeCut(that);
    // Free memory
    GTreeFreeRec(GTreeSubtrees(that));
}

// Disconnect the GTree 'that' from its parent
// If it has no parent, do nothing
void GTreeCut(GTree* that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // If there is no parent
    if (GTreeParent(that) == NULL)
        // Nothing to do
        return;
    // Remove the tree from the parent's subtrees
    GSetRemoveAll(GTreeSubtrees(GTreeParent(that)), that);
    // Cut the link to the parent

```

```

    that->_parent = NULL;
}

// Return the number of subtrees of the GTree 'that' and their subtrees
// recursively
int GTreeGetSize(GTree* that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Declare a variable to memorize the result and initialize it with
    // the number of subtrees
    int nb = GSetNbElem(GTreeSubtrees(that));
    // If there are subtrees
    if (nb > 0) {
        // Recursion on the subtrees
        GSetIterForward iter =
            GSetIterForwardCreateStatic(GTreeSubtrees(that));
        do {
            GTree* subtree = GSetIterGet(&iter);
            nb += GTreeGetSize(subtree);
        } while (GSetIterStep(&iter));
    }
    // Return the result
    return nb;
}

// ----- GTreeIter

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

// Create recursively the sequence of an iterator for depth first
void GTreeIterCreateSequenceDepthFirst(GSetGTree* seq, GTree* tree);

// Create recursively the sequence of an iterator for breadth first
void GTreeIterCreateSequenceBreadthFirst(GSetGTree* seq, GTree* tree,
    int lvl);

// Create recursively the sequence of an iterator for value first
void GTreeIterCreateSequenceValueFirst(GSetGTree* seq, GTree* tree,
    float val);

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

// Create a new GTreeIterDepth for the GTree 'tree'
GTreeIterDepth* GTreeIterDepthCreate(GTree* tree) {
#ifdef BUILDMODE == 0
    if (tree == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GTreeIterDepth *iter = PBErrMalloc(GTreeErr, sizeof(GTreeIterDepth));
    // Set properties
    ((GTreeIter*)iter)->_tree = tree;
    ((GTreeIter*)iter)->_seq = GSetGTreeCreateStatic();
    GTreeIterDepthUpdate(iter);
}

```

```

    GTreeIterReset(iter);
    // Return the iterator
    return iter;
}

// Create a new static GTreeIterDepth for the GTree 'tree'
GTreeIterDepth GTreeIterDepthCreateStatic(GTree* tree) {
    #if BUILDMODE == 0
        if (tree == NULL) {
            GTreeErr->_type = PBErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'tree' is null");
            PBErrCatch(GSetErr);
        }
    #endif
    // Declare the new iterator
    GTreeIterDepth iter;
    // Set properties
    ((GTreeIter*)&iter)->_tree = tree;
    ((GTreeIter*)&iter)->_seq = GSetGTreeCreateStatic();
    GTreeIterDepthUpdate(&iter);
    GTreeIterReset(&iter);
    // Return the iterator
    return iter;
}

// Create a new GTreeIterBreadth for the GTree 'tree'
GTreeIterBreadth* GTreeIterBreadthCreate(GTree* tree) {
    #if BUILDMODE == 0
        if (tree == NULL) {
            GTreeErr->_type = PBErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'tree' is null");
            PBErrCatch(GSetErr);
        }
    #endif
    // Declare the new iterator
    GTreeIterBreadth *iter =
        PBErrMalloc(GTreeErr, sizeof(GTreeIterBreadth));
    // Set properties
    ((GTreeIter*)&iter)->_tree = tree;
    ((GTreeIter*)&iter)->_seq = GSetGTreeCreateStatic();
    GTreeIterBreadthUpdate(iter);
    GTreeIterReset(iter);
    // Return the iterator
    return iter;
}

// Create a new static GTreeIterBreadth for the GTree 'tree'
GTreeIterBreadth GTreeIterBreadthCreateStatic(GTree* tree) {
    #if BUILDMODE == 0
        if (tree == NULL) {
            GTreeErr->_type = PBErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'tree' is null");
            PBErrCatch(GSetErr);
        }
    #endif
    // Declare the new iterator
    GTreeIterBreadth iter;
    // Set properties
    ((GTreeIter*)&iter)->_tree = tree;
    ((GTreeIter*)&iter)->_seq = GSetGTreeCreateStatic();
    GTreeIterBreadthUpdate(&iter);
    GTreeIterReset(&iter);
}

```

```

    // Return the iterator
    return iter;
}

// Create a new GTreeIterValue for the GTree 'tree'
GTreeIterValue* GTreeIterValueCreate(GTree* tree) {
#ifdef BUILDMODE == 0
    if (tree == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GTreeIterValue *iter = PBErrMalloc(GTreeErr, sizeof(GTreeIterValue));
    // Set properties
    ((GTreeIter*)iter)->_tree = tree;
    ((GTreeIter*)iter)->_seq = GSetGTreeCreateStatic();
    GTreeIterValueUpdate(iter);
    GTreeIterReset(iter);
    // Return the iterator
    return iter;
}

// Create a new static GTreeIterValue for the GTree 'tree' with
// 'rootval' the value of its root node
GTreeIterValue GTreeIterValueCreateStatic(GTree* tree) {
#ifdef BUILDMODE == 0
    if (tree == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GTreeIterValue iter;
    // Set properties
    ((GTreeIter*)&iter)->_tree = tree;
    ((GTreeIter*)&iter)->_seq = GSetGTreeCreateStatic();
    GTreeIterValueUpdate(&iter);
    GTreeIterReset(&iter);
    // Return the iterator
    return iter;
}

// Update the GTreeIterDepth 'that' in case its attached GTree has been
// modified
// The node sequence doesn't include the root node of the attached tree
void GTreeIterDepthUpdate(GTreeIterDepth* that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Flush the sequence
    GSetFlush(GTreeIterSeq(that));
    // Create the sequence with a Depth First run through nodes of the tree
    GTreeIterCreateSequenceDepthFirst(GTreeIterSeq(that),
        GTreeIterGTree(that));
    // Reset the current position

```

```

    GTreeIterReset(that);
}

// Create recursively the sequence of an iterator for depth first
void GTreeIterCreateSequenceDepthFirst(GSetGTree* seq, GTree* tree) {
    // Append the current tree to the sequence if it's not root
    if (!GTreeIsRoot(tree)) GSetAppend(seq, tree);
    // If there are subtrees
    if (GSetNbElem(GTreeSubtrees(tree)) > 0) {
        // Append the subtrees recursively
        GSetIterForward iter =
            GSetIterForwardCreateStatic(GTreeSubtrees(tree));
        do {
            GTree* subtree = GSetIterGet(&iter);
            GTreeIterCreateSequenceDepthFirst(seq, subtree);
        } while (GSetIterStep(&iter));
    }
}

// Update the GTreeIterBreadth 'that' in case its attached GTree has
// been modified
// The node sequence doesn't include the root node of the attached tree
void GTreeIterBreadthUpdate(GTreeIterBreadth* that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Flush the sequence
    GSetFlush(GTreeIterSeq(that));
    // Create the sequence with a Breadth First run through nodes of
    // the tree
    GTreeIterCreateSequenceBreadthFirst(GTreeIterSeq(that),
        GTreeIterGTree(that), 0);
    // Reset the current position
    GTreeIterReset(that);
}

// Create recursively the sequence of an iterator for breadth first
void GTreeIterCreateSequenceBreadthFirst(GSetGTree* seq, GTree* tree,
    int lvl) {
    // Append the current tree to the sequence if it's not root
    if (!GTreeIsRoot(tree)) GSetAddSort(seq, tree, lvl);
    // If there are subtrees
    if (GSetNbElem(GTreeSubtrees(tree)) > 0) {
        // Declare a variable to memorize the next lvl
        int nextLvl = lvl + 1;
        // Append the subtrees recursively
        GSetIterForward iter =
            GSetIterForwardCreateStatic(GTreeSubtrees(tree));
        do {
            GTree* subtree = GSetIterGet(&iter);
            GTreeIterCreateSequenceBreadthFirst(seq, subtree, nextLvl);
        } while (GSetIterStep(&iter));
    }
}

// Update the GTreeIterValue 'that' in case its attached GTree has been
// modified
// The node sequence doesn't include the root node of the attached tree

```

```

void GTreeIterValueUpdate(GTreeIterValue* that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Flush the sequence
    GSetFlush(GTreeIterSeq(that));
    // Create the sequence with a Value First run through nodes of the tree
    GTreeIterCreateSequenceValueFirst(GTreeIterSeq(that),
        GTreeIterGTree(that), 0.0);
    // Reset the current position
    GTreeIterReset(that);
}

// Create recursively the sequence of an iterator for value first
void GTreeIterCreateSequenceValueFirst(GSetGTree* seq, GTree* tree,
    float val) {
    // Append the current tree to the sequence if it's not root
    if (!GTreeIsRoot(tree)) GSetAddSort(seq, tree, val);
    // If there are subtrees
    if (GSetNbElem(GTreeSubtrees(tree)) > 0) {
        // Append the subtrees recursively
        GSetIterForward iter =
            GSetIterForwardCreateStatic(GTreeSubtrees(tree));
        do {
            GTree* subtree = GSetIterGet(&iter);
            GSetElem* elem = GSetIterGetElem(&iter);
            GTreeIterCreateSequenceValueFirst(seq, subtree, elem->_sortVal);
        } while (GSetIterStep(&iter));
    }
}

// Free the memory used by the iterator 'that'
void _GTreeIterFree(GTreeIter** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        // Nothing to do
        return;
    // Free memory
    GSetFlush(GTreeIterSeq(*that));
    free(*that);
    *that = NULL;
}

// Free the memory used by the static iterator 'that'
void _GTreeIterFreeStatic(GTreeIter* that) {
    // Check argument
    if (that == NULL)
        // Nothing to do
        return;
    // Free memory
    GSetFlush(GTreeIterSeq(that));
}

```

## 2.2 pbmath-inline.c

```
// ===== GTREE-INLINE.C =====

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

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

// Get the user data of the GTree 'that'
#if BUILDMODE != 0
inline
#endif
void* GTreeData(GTree* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return that->_data;
}

// Get the parent of the GTree 'that'
#if BUILDMODE != 0
inline
#endif
GTree* GTreeParent(GTree* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return that->_parent;
}

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

// Get the set of subtrees of the GTree 'that'
#if BUILDMODE != 0
inline
#endif
GSetGTree* GTreeSubtrees(GTree* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
    }
#endif
}
```



```

        PBErCatch(GSetErr);
    }
#endif
    return &(that->_subtrees);
}

// Return true if the GTree 'that' is a root
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GTreeIsRoot(GTree* that) {
    if BUILDMODE == 0
        if (that == NULL) {
            GTreeErr->_type = PBErTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PBErCatch(GSetErr);
        }
    #endif
    return (that->_parent == NULL ? true : false);
}

// Return true if the GTree 'that' is a leaf
// Return false else
#if BUILDMODE != 0
inline
#endif
bool GTreeIsLeaf(GTree* that) {
    if BUILDMODE == 0
        if (that == NULL) {
            GTreeErr->_type = PBErTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PBErCatch(GSetErr);
        }
    #endif
    return (GSetNbElem(&(that->_subtrees)) == 0 ? true : false);
}

// ----- GTreeIter

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

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

// Reset the iterator 'that' at its start position
#if BUILDMODE != 0
inline
#endif
void _GTreeIterReset(GTreeIter* that) {
    if BUILDMODE == 0
        if (that == NULL) {
            GTreeErr->_type = PBErTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PBErCatch(GSetErr);
        }
    #endif
    that->_curPos = ((GSet*)&(that->_seq))->_head;
}

// Reset the iterator 'that' to its end position
#if BUILDMODE != 0
inline

```

```

#endif
void _GTreeIterToEnd(GTreeIter* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    that->_curPos = ((GSet*)&(that->_seq))->_tail;
}

// Step the iterator 'that' at its next position
// Return true if it could move to the next position
// Return false if it's already at the last position
#if BUILDMODE != 0
inline
#endif
bool _GTreeIterStep(GTreeIter* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (that->_curPos == NULL) {
        GTreeErr->_type = PBErrTypeInvalidArg;
        sprintf(GSetErr->_msg, "'that->_curPos' is null");
        PBErrCatch(GSetErr);
    }
#endif
    if (that->_curPos->_next != NULL) {
        that->_curPos = that->_curPos->_next;
        return true;
    }
    return false;
}

// Step back the iterator 'that' at its next position
// Return true if it could move to the previous position
// Return false if it's already at the first position
#if BUILDMODE != 0
inline
#endif
bool _GTreeIterStepBack(GTreeIter* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (that->_curPos == NULL) {
        GTreeErr->_type = PBErrTypeInvalidArg;
        sprintf(GSetErr->_msg, "'that->_curPos' is null");
        PBErrCatch(GSetErr);
    }
#endif
    if (that->_curPos->_prev != NULL) {
        that->_curPos = that->_curPos->_prev;
        return true;
    }
    return false;
}

```

```

}

// Apply a function to all elements' data of the GTree of the iterator
// The iterator is first reset, then the function is apply sequentially
// 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 _GTreeIterApply(GTreeIter* that,
    void(*fun)(void* data, void* param), void* param) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (fun == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'fun' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Reset the iterator;
    GTreeIterReset(that);
    // If the associated tree is not empty
    if (GSetNbElem(&(amp;that->_seq)) > 0) {
        // For each node of the tree
        do {
            // Apply the user function
            fun(GTreeIterGetData(that), param);
        } while (GTreeIterStep(that));
    }
}

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

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

```

```

    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return (that->_curPos == ((GSet*)&(that->_seq))->_tail);
}

// Change the attached tree of the iterator, and reset it
#if BUILDMODE != 0
inline
#endif
void _GTreeIterDepthSetGTree(GTreeIterDepth* that, GTree* tree) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (tree == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Set the tree
    ((GTreeIter*)that)->_tree = tree;
    // Update the sequence
    GTreeIterDepthUpdate(that);
    // Reset the iterator
    GTreeIterReset(that);
}

#if BUILDMODE != 0
inline
#endif
void _GTreeIterBreadthSetGTree(GTreeIterBreadth* that, GTree* tree) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (tree == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Set the tree
    ((GTreeIter*)that)->_tree = tree;
    // Update the sequence
    GTreeIterBreadthUpdate(that);
    // Reset the iterator
    GTreeIterReset(that);
}

#if BUILDMODE != 0
inline
#endif
void _GTreeIterValueSetGTree(GTreeIterValue* that, GTree* tree) {
#if BUILDMODE == 0
    if (that == NULL) {

```

```

        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (tree == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Set the tree
    ((GTreeIter*)that)->_tree = tree;
    // Update the sequence
    GTreeIterValueUpdate(that);
    // Reset the iterator
    GTreeIterReset(that);
}

// Return the user data of the tree currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
void* _GTreeIterGetData(GTreeIter* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (that->_curPos == NULL) {
        GTreeErr->_type = PBErrTypeInvalidArg;
        sprintf(GSetErr->_msg, "'that->_curPos' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return ((GTree*)(that->_curPos->_data))->_data;
}

// Return the tree currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GTree* _GTreeIterGetGTree(GTreeIter* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return (GTree*)(that->_curPos->_data);
}

// Return the tree associated to the iterator 'that'
#if BUILDMODE != 0
inline
#endif
GTree* _GTreeIterGTree(GTreeIter* that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
    }
#endif
}

```

```

        PBErCatch(GSetErr);
    }
#endif
    return that->_tree;
}

// Return the sequence of the iterator 'that'
#if BUILDMODE != 0
inline
#endif
GSetGTree* _GTreeIterSeq(GTreeIter* that) {
    if BUILDMODE == 0
        if (that == NULL) {
            GTreeErr->_type = PBErTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PBErCatch(GSetErr);
        }
    #endif
    return &(that->_seq);
}

```

### 3 Makefile

```

#directory
PBERRDIR=../PBEr
GSETDIR=../GSet

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

include $(PBERRDIR)/Makefile.inc

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

# compiler
COMPILER=gcc

#rules
all : main

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

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

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

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

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

```

```

clean :
rm -rf *.o main

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

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

```

## 4 Unit tests

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "gtree.h"

#define RANDOMSEED 0

void UnitTestGTreeCreateFree() {
    GTree* tree = GTreeCreate();
    if (tree == NULL ||
        tree->_parent != NULL ||
        GSetNbElem(&(tree->_subtrees)) != 0 ||
        tree->_data != NULL) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeCreate failed");
        PBErrCatch(GTreeErr);
    }
    GTreeFree(&tree);
    if (tree != NULL) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeFree failed");
        PBErrCatch(GTreeErr);
    }
    int data = 1;
    tree = GTreeCreateData(&data);
    if (tree == NULL ||
        tree->_parent != NULL ||
        GSetNbElem(&(tree->_subtrees)) != 0 ||
        tree->_data != &data) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeCreateData failed");
        PBErrCatch(GTreeErr);
    }
    GTreeFree(&tree);
    GTree treeStatic = GTreeCreateStatic();
    if (treeStatic._parent != NULL ||
        GSetNbElem(&(treeStatic._subtrees)) != 0 ||
        treeStatic._data != NULL) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeCreateStatic failed");
        PBErrCatch(GTreeErr);
    }
}

```

```

    GTreeFreeStatic(&treeStatic);
    printf("UnitTestGTreeCreateFree OK\n");
}

void UnitTestGTreeGetSet() {
    GTree tree = GTreeCreateStatic();
    int data = 1;
    tree._data = &data;
    if (GTreeData(&tree) != &data) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeData failed");
        PBErrCatch(GTreeErr);
    }
    int data2 = 1;
    GTreeSetData(&tree, &data2);
    if (GTreeData(&tree) != &data2) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeSetData failed");
        PBErrCatch(GTreeErr);
    }
    if (GTreeSubtrees(&tree) != &(tree._subtrees)) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeSubTrees failed");
        PBErrCatch(GTreeErr);
    }
    if (GTreeIsRoot(&tree) == false) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIsRoot failed");
        PBErrCatch(GTreeErr);
    }
    tree._parent = &tree;
    if (GTreeIsRoot(&tree) == true) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIsRoot failed");
        PBErrCatch(GTreeErr);
    }
    if (GTreeParent(&tree) != &tree) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeParent failed");
        PBErrCatch(GTreeErr);
    }
    tree._parent = NULL;
    if (GTreeIsLeaf(&tree) == false) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIsLeaf failed");
        PBErrCatch(GTreeErr);
    }
    GTreeAppendData(&tree, &data);
    if (GTreeIsLeaf(&tree) == true) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIsLeaf failed");
        PBErrCatch(GTreeErr);
    }
    GTreeFreeStatic(&tree);
    printf("UnitTestGTreeGetSet OK\n");
}

void UnitTestGTreeCutGetSize() {
    GTree tree = GTreeCreateStatic();
    int data = 1;
    GTreeAppendData(&tree, &data);
    GTreeAppendData(&tree, &data);

```



```

GTreeAppendData(GTreeSubtree(&tree, 1), &data);
if (GTreeGetSize(&tree) != 3) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeGetSize failed");
    PBErrCatch(GTreeErr);
}
GTree* cuttree = GTreeSubtree(&tree, 1);
GTreeCut(cuttree);
if (GTreeGetSize(&tree) != 1 ||
    GTreeGetSize(cuttree) != 1) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeCut failed");
    PBErrCatch(GTreeErr);
}
GTreeFreeStatic(&tree);
GTreeFree(&cuttree);
printf("UnitTestGTreeCutGetSize OK\n");
}

void UnitTestGTree() {
    UnitTestGTreeCreateFree();
    UnitTestGTreeGetSet();
    UnitTestGTreeCutGetSize();
    printf("UnitTestGTree OK\n");
}

int dataExampleTree[10] = {0,1,2,3,4,5,6,7,8,9};
GTree* GetExampleTree() {
    GTree* tree = GTreeCreate();
    GTreeAddSortData(tree, dataExampleTree + 0, 0);
    GTreeAddSortData(tree, dataExampleTree + 9, 9);
    GTree* subtree = GTreeSubtree(tree, 0);
    GTreeAddSortData(subtree, dataExampleTree + 1, 1);
    GTreeAddSortData(subtree, dataExampleTree + 2, 2);
    subtree = GTreeSubtree(tree, 1);
    GTreeAddSortData(subtree, dataExampleTree + 3, 3);
    GTreeAddSortData(subtree, dataExampleTree + 4, 4);
    subtree = GTreeSubtree(subtree, 0);
    GTreeAddSortData(subtree, dataExampleTree + 8, 8);
    GTreeAddSortData(subtree, dataExampleTree + 6, 6);
    subtree = GTreeSubtree(subtree, 1);
    GTreeAddSortData(subtree, dataExampleTree + 7, 7);
    GTreeAddSortData(subtree, dataExampleTree + 5, 5);
    return tree;
}

void funApply(void* data, void* param) {
    printf("%d%c", *(int*)data, *(char*)param);
}

void UnitTestGTreeIterDepth() {
    GTree* tree = GetExampleTree();
    GTreeIterDepth* iter = GTreeIterDepthCreate(tree);
    if (iter == NULL ||
        iter->_iter._tree != tree ||
        GSetNbElem(&(iter->_iter._seq)) != 10 ||
        iter->_iter._curPos != iter->_iter._seq._set._head) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterDepthCreate failed");
        PBErrCatch(GTreeErr);
    }
    int check[10] = {0,1,2,9,3,6,8,5,7,4};

```

```

int iCheck = 0;
do {
    int* data = GTreeIterGetData(iter);
    if (*data != check[iCheck]) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterDepth failed");
        PBErrCatch(GTreeErr);
    }
    ++iCheck;
} while (GTreeIterStep(iter));
GTreeIterFree(&iter);
if (iter != NULL) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterFree failed");
    PBErrCatch(GTreeErr);
}
GTreeIterDepth iterstatic = GTreeIterDepthCreateStatic(tree);
if (iterstatic._iter._tree != tree ||
    GSetNbElem(&(iterstatic._iter._seq)) != 10 ||
    iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterDepthCreateStatic failed");
    PBErrCatch(GTreeErr);
}
iCheck = 0;
do {
    int* data = GTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterDepth failed");
        PBErrCatch(GTreeErr);
    }
    ++iCheck;
} while (GTreeIterStep(&iterstatic));
check[3] = 12;
dataExampleTree[9] = 12;
GTreeIterDepthUpdate(&iterstatic);
iCheck = 0;
do {
    int* data = GTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterUpdate failed");
        PBErrCatch(GTreeErr);
    }
    ++iCheck;
} while (GTreeIterStep(&iterstatic));
dataExampleTree[9] = 9;
GTreeIterReset(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterReset failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsFirst(&iterstatic) == false) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsFirst failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsLast(&iterstatic) == true) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsLast failed");
}

```

```

    PBErCatch(GTreeErr);
}
if (GTreeIterGetGTree(&iterstatic) != GTreeSubtree(tree, 0)) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterGetGTree failed");
    PBErCatch(GTreeErr);
}
GTreeIterToEnd(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._tail) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterToEnd failed");
    PBErCatch(GTreeErr);
}
if (GTreeIterIsFirst(&iterstatic) == true) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsFirst failed");
    PBErCatch(GTreeErr);
}
if (GTreeIterIsLast(&iterstatic) == false) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsLast failed");
    PBErCatch(GTreeErr);
}
GTreeIterStepBack(&iterstatic);
if (iterstatic._iter._curPos->_next !=
    iterstatic._iter._seq._set._tail) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterStepBack failed");
    PBErCatch(GTreeErr);
}
if (GTreeIterGTree(&iterstatic) != tree) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterGTree failed");
    PBErCatch(GTreeErr);
}
if (GTreeIterSeq(&iterstatic) != &(iterstatic._iter._seq)) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterSeq failed");
    PBErCatch(GTreeErr);
}
char c = ',';
GTreeIterApply(&iterstatic, &funApply, &c);
printf("\n");
GTree* treeB = GTreeCreate();
GTreeIterSetGTree(&iterstatic, treeB);
if (GTreeIterGTree(&iterstatic) != treeB ||
    GSetNbElem(&(iterstatic._iter._seq)) != 0) {
    GTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterSetGTree failed");
    PBErCatch(GTreeErr);
}
GTreeIterFreeStatic(&iterstatic);
GTreeFree(&tree);
GTreeFree(&treeB);
printf("UnitTestGTreeIterDepth OK\n");
}

void UnitTestGTreeIterBreadth() {
    GTree* tree = GetExampleTree();
    GTreeIterBreadth* iter = GTreeIterBreadthCreate(tree);
    if (iter == NULL ||
        iter->_iter._tree != tree ||

```

```

    GSetNbElem(&(iter->_iter._seq)) != 10 ||
    iter->_iter._curPos != iter->_iter._seq._set._head) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterBreadthCreate failed");
    PBErrCatch(GTreeErr);
}
int check[10] = {0,9,1,2,3,4,6,8,5,7};
int iCheck = 0;
do {
    int* data = GTreeIterGetData(iter);
    if (*data != check[iCheck]) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterBreadth failed");
        PBErrCatch(GTreeErr);
    }
    ++iCheck;
} while (GTreeIterStep(iter));
GTreeIterFree(&iter);
if (iter != NULL) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterFree failed");
    PBErrCatch(GTreeErr);
}
GTreeIterBreadth iterstatic = GTreeIterBreadthCreateStatic(tree);
if (iterstatic._iter._tree != tree ||
    GSetNbElem(&(iterstatic._iter._seq)) != 10 ||
    iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterBreadthCreateStatic failed");
    PBErrCatch(GTreeErr);
}
iCheck = 0;
do {
    int* data = GTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterBreadth failed");
        PBErrCatch(GTreeErr);
    }
    ++iCheck;
} while (GTreeIterStep(&iterstatic));
check[1] = 12;
dataExampleTree[9] = 12;
GTreeIterBreadthUpdate(&iterstatic);
iCheck = 0;
do {
    int* data = GTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterUpdate failed");
        PBErrCatch(GTreeErr);
    }
    ++iCheck;
} while (GTreeIterStep(&iterstatic));
dataExampleTree[9] = 9;
GTreeIterReset(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterReset failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsFirst(&iterstatic) == false) {

```

```

    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsFirst failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsLast(&iterstatic) == true) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsLast failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterGetGTree(&iterstatic) != GTreeSubtree(tree, 0)) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterGetGTree failed");
    PBErrCatch(GTreeErr);
}
GTreeIterToEnd(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._tail) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterToEnd failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsFirst(&iterstatic) == true) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsFirst failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsLast(&iterstatic) == false) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsLast failed");
    PBErrCatch(GTreeErr);
}
GTreeIterStepBack(&iterstatic);
if (iterstatic._iter._curPos->_next !=
    iterstatic._iter._seq._set._tail) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterStepBack failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterGTree(&iterstatic) != tree) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterGTree failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterSeq(&iterstatic) != &(iterstatic._iter._seq)) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterSeq failed");
    PBErrCatch(GTreeErr);
}
char c = ',';
GTreeIterApply(&iterstatic, &funApply, &c);
printf("\n");
GTree* treeB = GTreeCreate();
GTreeIterSetGTree(&iterstatic, treeB);
if (GTreeIterGTree(&iterstatic) != treeB ||
    GSetNbElem(&(iterstatic._iter._seq)) != 0) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterSetGTree failed");
    PBErrCatch(GTreeErr);
}
GTreeIterFreeStatic(&iterstatic);
GTreeFree(&tree);
GTreeFree(&treeB);
printf("UnitTestGTreeIterBreadth OK\n");

```

```

}

void UnitTestGTreeIterValue() {
    GTree* tree = GetExampleTree();
    GTreeIterValue* iter = GTreeIterValueCreate(tree);
    if (iter == NULL ||
        iter->_iter._tree != tree ||
        GSetNbElem(&(iter->_iter._seq)) != 10 ||
        iter->_iter._curPos != iter->_iter._seq._set._head) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterValueCreate failed");
        PBErrCatch(GTreeErr);
    }
    int check[10] = {0,1,2,3,4,5,6,7,8,9};
    int iCheck = 0;
    do {
        int* data = GTreeIterGetData(iter);
        if (*data != check[iCheck]) {
            GTreeErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GTreeErr->_msg, "GTreeIterValue failed");
            PBErrCatch(GTreeErr);
        }
        ++iCheck;
    } while (GTreeIterStep(iter));
    GTreeIterFree(&iter);
    if (iter != NULL) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterFree failed");
        PBErrCatch(GTreeErr);
    }
    GTreeIterValue iterstatic = GTreeIterValueCreateStatic(tree);
    if (iterstatic._iter._tree != tree ||
        GSetNbElem(&(iterstatic._iter._seq)) != 10 ||
        iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
        GTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GTreeErr->_msg, "GTreeIterValueCreateStatic failed");
        PBErrCatch(GTreeErr);
    }
    iCheck = 0;
    do {
        int* data = GTreeIterGetData(&iterstatic);
        if (*data != check[iCheck]) {
            GTreeErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GTreeErr->_msg, "GTreeIterValue failed");
            PBErrCatch(GTreeErr);
        }
        ++iCheck;
    } while (GTreeIterStep(&iterstatic));
    check[9] = 12;
    dataExampleTree[9] = 12;
    GTreeIterValueUpdate(&iterstatic);
    iCheck = 0;
    do {
        int* data = GTreeIterGetData(&iterstatic);
        if (*data != check[iCheck]) {
            GTreeErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GTreeErr->_msg, "GTreeIterUpdate failed");
            PBErrCatch(GTreeErr);
        }
        ++iCheck;
    } while (GTreeIterStep(&iterstatic));
    dataExampleTree[9] = 9;
}

```

```

GTreeIterReset(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterReset failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsFirst(&iterstatic) == false) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsFirst failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsLast(&iterstatic) == true) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsLast failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterGetGTree(&iterstatic) != GTreeSubtree(tree, 0)) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterGetGTree failed");
    PBErrCatch(GTreeErr);
}
GTreeIterToEnd(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._tail) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterToEnd failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsFirst(&iterstatic) == true) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsFirst failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterIsLast(&iterstatic) == false) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterIsLast failed");
    PBErrCatch(GTreeErr);
}
GTreeIterStepBack(&iterstatic);
if (iterstatic._iter._curPos->_next !=
    iterstatic._iter._seq._set._tail) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterStepBack failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterGTree(&iterstatic) != tree) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterGTree failed");
    PBErrCatch(GTreeErr);
}
if (GTreeIterSeq(&iterstatic) != &(iterstatic._iter._seq)) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GTreeErr->_msg, "GTreeIterSeq failed");
    PBErrCatch(GTreeErr);
}
char c = ',';
GTreeIterApply(&iterstatic, &funApply, &c);
printf("\n");
GTree* treeB = GTreeCreate();
GTreeIterSetGTree(&iterstatic, treeB);
if (GTreeIterGTree(&iterstatic) != treeB ||
    GSetNbElem(&(iterstatic._iter._seq)) != 0) {
    GTreeErr->_type = PBErrTypeUnitTestFailed;
}

```

```

        sprintf(GTreeErr->_msg, "GTreeIterSetGTree failed");
        PBErrCatch(GTreeErr);
    }
    GTreeIterFreeStatic(&iterstatic);
    GTreeFree(&tree);
    GTreeFree(&treeB);
    printf("UnitTestGTreeIterValue OK\n");
}

void UnitTestGTreeIter() {
    UnitTestGTreeIterDepth();
    UnitTestGTreeIterBreadth();
    UnitTestGTreeIterValue();
    printf("UnitTestGTreeIter OK\n");
}

void UnitTestAll() {
    UnitTestGTree();
    UnitTestGTreeIter();
    printf("UnitTestAll OK\n");
}

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

```

## 5 Unit tests output

```

UnitTestGTreeCreateFree OK
UnitTestGTreeGetSet OK
UnitTestGTreeCutGetSize OK
UnitTestGTree OK
0,1,2,9,3,6,8,5,7,4,
UnitTestGTreeIterDepth OK
0,9,1,2,3,4,6,8,5,7,
UnitTestGTreeIterBreadth OK
0,1,2,3,4,5,6,7,8,9,
UnitTestGTreeIterValue OK
UnitTestGTreeIter OK
UnitTestAll OK

```