

GTree

P. Baillehache

March 10, 2019

Contents

1	Interface	2
2	Code	14
2.1	gtree.c	14
2.2	gtree-inline.c	22
3	Makefile	28
4	Unit tests	29
5	Unit tests output	39

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"

// ----- GenTree

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

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

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

typedef struct GenTreeIter GenTreeIter;

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

// Create a new GenTree
GenTree* GenTreeCreate(void);

// Create a new static GenTree
GenTree GenTreeCreateStatic(void);

// Create a new GenTree with user data 'data'
GenTree* GenTreeCreateData(void* const data);

// Free the memory used by the GenTree '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 _GenTreeFree(GenTree** that);

// Free the memory used by the static GenTree '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 _GenTreeFreeStatic(GenTree* that);

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

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

// Get the set of subtrees of the GenTree 'that'
#if BUILDMODE != 0
inline
#endif
GSetGenTree* _GenTreeSubtrees(const GenTree* const that);

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

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

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

// Return the parent of the GenTree 'that'
#if BUILDMODE != 0
inline
#endif
GenTree* _GenTreeParent(const GenTree* const that);

// Return the number of subtrees of the GenTree 'that' and their subtrees
// recursively
int _GenTreeGetSize(const GenTree* const that);

// Append a new node with 'data' to the first node containing 'node'
// in the GenTree 'that'
// Uses the iterator 'iter' to search the node
// Return true if we could find 'node', false else
bool _GenTreeAppendToNode(GenTree* const that,
    void* const data, void* const node, GenTreeIter* const iter);

// Search the first node containing 'data' in the GenTree 'that'
// Uses the iterator 'iter' to search the node. Do not reset the
// iterator, thus several calls with the same iterator will return
// eventual successive nodes containing the same data. If one want to
// loop on these nodes, the proper stopping condition is

```

```

// while(GenTreeSearch() != NULL && GenTreeIterIsLast() == false)
// Return the node if we could find 'data', null else
GenTree* _GenTreeSearch(const GenTree* const that,
    const void* const data, GenTreeIter* const iter);

// Wrapping of GSet functions
inline GenTree* _GenTreeSubtree(const GenTree* const that, const int iSubtree) {
    return GSetGet(_GenTreeSubtrees(that), iSubtree);
}
inline GenTree* _GenTreeFirstSubtree(const GenTree* const that) {
    return GSetHead(_GenTreeSubtrees(that));
}
inline GenTree* _GenTreeLastSubtree(const GenTree* const that) {
    return GSetTail(_GenTreeSubtrees(that));
}
inline GenTree* _GenTreePopSubtree(GenTree* const that) {
    return GSetPop(_GenTreeSubtrees(that));
}
inline GenTree* _GenTreeDropSubtree(GenTree* const that) {
    return GSetDrop(_GenTreeSubtrees(that));
}
inline GenTree* _GenTreeRemoveSubtree(GenTree* const that, const int iSubtree) {
    return GSetRemove((GSet*)_GenTreeSubtrees(that), iSubtree);
}

inline void _GenTreePushSubtree(GenTree* const that, GenTree* const tree) {
    if (!tree) return;
    GSetPush(_GenTreeSubtrees(that), tree);
    tree->_parent = that;
}
inline void _GenTreeAddSortSubTree(GenTree* const that, GenTree* const tree,
    const float sortVal) {
    if (!tree) return;
    GSetAddSort(_GenTreeSubtrees(that), tree, sortVal);
    tree->_parent = that;
}
inline void _GenTreeInsertSubtree(GenTree* const that, GenTree* const tree,
    const int pos) {
    if (!tree) return;
    GSetInsert(_GenTreeSubtrees(that), tree, pos);
    tree->_parent = that;
}
inline void _GenTreeAppendSubtree(GenTree* const that, GenTree* const tree) {
    if (!tree) return;
    GSetAppend(_GenTreeSubtrees(that), tree);
    tree->_parent = that;
}

inline void _GenTreePushData(GenTree* const that, void* const data) {
    GenTree* tree = GenTreeCreateData(data);
    GSetPush(_GenTreeSubtrees(that), tree);
    tree->_parent = that;
}
inline void _GenTreeAddSortData(GenTree* const that, void* const data,
    const float sortVal) {
    GenTree* tree = GenTreeCreateData(data);
    GSetAddSort(_GenTreeSubtrees(that), tree, sortVal);
    tree->_parent = that;
}
inline void _GenTreeInsertData(GenTree* const that, void* const data,
    const int pos) {
    GenTree* tree = GenTreeCreateData(data);

```

```

    GSetInsert(_GenTreeSubtrees(that), tree, pos);
    tree->_parent = that;
}
inline void _GenTreeAppendData(GenTree* const that, void* const data) {
    GenTree* tree = GenTreeCreateData(data);
    GSetAppend(_GenTreeSubtrees(that), tree);
    tree->_parent = that;
}

// ----- GenTreeIter

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

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

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

typedef struct GenTreeIterDepth {GenTreeIter _iter;} GenTreeIterDepth;
typedef struct GenTreeIterBreadth {GenTreeIter _iter;} GenTreeIterBreadth;
typedef struct GenTreeIterValue {GenTreeIter _iter;} GenTreeIterValue;

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

// Create a new GenTreeIterDepth for the GenTree 'tree'
GenTreeIterDepth* _GenTreeIterDepthCreate(GenTree* const tree);

// Create a new static GenTreeIterDepth for the GenTree 'tree'
GenTreeIterDepth* _GenTreeIterDepthCreateStatic(GenTree* const tree);

// Create a new GenTreeIterBreadth for the GenTree 'tree'
GenTreeIterBreadth* _GenTreeIterBreadthCreate(GenTree* const tree);

// Create a new static GenTreeIterBreadth for the GenTree 'tree'
GenTreeIterBreadth* _GenTreeIterBreadthCreateStatic(GenTree* const tree);

// Create a new GenTreeIterValue for the GenTree 'tree'
GenTreeIterValue* _GenTreeIterValueCreate(GenTree* const tree);

// Create a new static GenTreeIterValue for the GenTree 'tree'
GenTreeIterValue* _GenTreeIterValueCreateStatic(GenTree* const tree);

// Update the GenTreeIterDepth 'that' in case its attached GenTree has been
// modified
// The node sequence doesn't include the root node of the attached tree
void GenTreeIterDepthUpdate(GenTreeIterDepth* const that);

// Update the GenTreeIterBreadth 'that' in case its attached GenTree has
// been modified
// The node sequence doesn't include the root node of the attached tree
void GenTreeIterBreadthUpdate(GenTreeIterBreadth* const that);

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

```

```

void GenTreeIterValueUpdate(GenTreeIterValue* const that);

// Free the memory used by the iterator 'that'
void _GenTreeIterFree(GenTreeIter** that);

// Free the memory used by the static iterator 'that'
void _GenTreeIterFreeStatic(GenTreeIter* const that);

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

// Reset the iterator 'that' at its end position
#if BUILDMODE != 0
inline
#endif
void _GenTreeIterToEnd(GenTreeIter* const 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 _GenTreeIterStep(GenTreeIter* const 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 _GenTreeIterStepBack(GenTreeIter* const that);

// Apply a function to all elements' data of the GenTree 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 _GenTreeIterApply(GenTreeIter* const that,
    void(*fun)(void* const data, void* const param), void* const param);

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GenTree)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool _GenTreeIterIsFirst(const GenTreeIter* const that);

// Return true if the iterator is at the end of the elements (from
// its point of view, not the order in the GenTree)
// Return false else
#if BUILDMODE != 0
inline
#endif

```

```

bool _GenTreeIterIsLast(const GenTreeIter* const that);

// Change the attached tree of the iterator, and reset it
#if BUILDMODE != 0
inline
#endif
void _GenTreeIterDepthSetGenTree(GenTreeIterDepth* const that,
    GenTree* const tree);
#if BUILDMODE != 0
inline
#endif
void _GenTreeIterBreadthSetGenTree(GenTreeIterBreadth* const that,
    GenTree* const tree);
#if BUILDMODE != 0
inline
#endif
void _GenTreeIterValueSetGenTree(GenTreeIterValue* const that,
    GenTree* const tree);

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

// Return the tree currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GenTree* _GenTreeIterGetGenTree(const GenTreeIter* const that);

// Return the tree associated to the iterator 'that'
#if BUILDMODE != 0
inline
#endif
GenTree* _GenTreeIterGenTree(const GenTreeIter* const that);

// Return the sequence of the iterator
#if BUILDMODE != 0
inline
#endif
GSetGenTree* _GenTreeIterSeq(const GenTreeIter* const that);

// ===== Typed GenTree =====

typedef struct GenTreeStr {GenTree _tree;} GenTreeStr;
#define GenTreeStrCreate() ((GenTreeStr*)GenTreeCreate())
inline GenTreeStr GenTreeStrCreateStatic(void)
    {GenTreeStr ret = {._tree=GenTreeCreateStatic()}; return ret;}
#define GenTreeStrCreateData(Data) ((GenTreeStr*)GenTreeCreateData(Data))
inline char* _GenTreeStrData(const GenTreeStr* const that) {
    return (char*)_GenTreeData((const GenTree* const)that);
}
inline void _GenTreeStrSetData(GenTreeStr* const that, char* const data) {
    _GenTreeSetData((GenTree* const)that, (void* const)data);
}
inline GSetGenTreeStr* _GenTreeStrSubtrees(const GenTreeStr* const that) {
    return (GSetGenTreeStr*)_GenTreeSubtrees((const GenTree* const)that);
}
inline GenTreeStr* _GenTreeStrParent(const GenTreeStr* const that) {
    return (GenTreeStr*)_GenTreeParent((const GenTree* const)that);
}

```

```

inline void _GenTreeStrPushData(GenTreeStr* const that, char* const data) {
    _GenTreePushData((GenTree* const)that, (void* const)data);
}
inline void _GenTreeStrAddSortData(GenTreeStr* const that, char* const data,
    const float sortVal) {
    _GenTreeAddSortData((GenTree* const)that, (void* const)data, sortVal);
}
inline void _GenTreeStrInsertData(GenTreeStr* const that, char* const data,
    const int pos) {
    _GenTreeInsertData((GenTree* const)that, (void* const)data, pos);
}
inline void _GenTreeStrAppendData(GenTreeStr* const that, char* const data) {
    _GenTreeAppendData((GenTree* const)that, (void* const)data);
}
inline GenTreeStr* _GenTreeStrSubtree(const GenTreeStr* const that,
    const int iSubtree) {
    return (GenTreeStr*)_GenTreeSubtree((const GenTree* const)that, iSubtree);
}
inline GenTreeStr* _GenTreeStrFirstSubtree(const GenTreeStr* const that) {
    return (GenTreeStr*)_GenTreeFirstSubtree((const GenTree* const)that);
}
inline GenTreeStr* _GenTreeStrLastSubtree(const GenTreeStr* const that) {
    return (GenTreeStr*)_GenTreeLastSubtree((const GenTree* const)that);
}
inline GenTreeStr* _GenTreeStrPopSubtree(GenTreeStr* const that) {
    return (GenTreeStr*)_GenTreePopSubtree((GenTree* const)that);
}
inline GenTreeStr* _GenTreeStrDropSubtree(GenTreeStr* const that) {
    return (GenTreeStr*)_GenTreeDropSubtree((GenTree* const)that);
}
inline GenTreeStr* _GenTreeStrRemoveSubtree(GenTreeStr* const that,
    const int iSubtree) {
    return (GenTreeStr*)_GenTreeRemoveSubtree((GenTree* const)that, iSubtree);
}
inline void _GenTreeStrPushSubtree(GenTreeStr* const that,
    GenTreeStr* const tree) {
    _GenTreePushSubtree((GenTree* const)that, (GenTree* const)tree);
}
inline void _GenTreeStrAddSortSubTree(GenTreeStr* const that,
    GenTreeStr* const tree, const float sortVal) {
    _GenTreeAddSortSubTree((GenTree* const)that, (GenTree* const)tree, sortVal);
}
inline void _GenTreeStrInsertSubtree(GenTreeStr* const that,
    GenTreeStr* const tree, const int pos) {
    _GenTreeInsertSubtree((GenTree* const)that, (GenTree* const)tree, pos);
}
inline void _GenTreeStrAppendSubtree(GenTreeStr* const that,
    GenTreeStr* const tree) {
    _GenTreeAppendSubtree((GenTree* const)that, (GenTree* const)tree);
}

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

#define GenTreeFree(RefTree) _Generic(RefTree, \
    GenTree*: _GenTreeFree, \
    GenTreeStr*: _GenTreeFree, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(RefTree))

#define GenTreeFreeStatic(Tree) _Generic(Tree, \
    GenTree*: _GenTreeFreeStatic, \
    const GenTree*: _GenTreeFreeStatic, \
    GenTreeStr*: _GenTreeFreeStatic, \

```



```

    const GenTreeStr*: _GenTreeFreeStatic, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeParent(Tree) _Generic(Tree, \
    GenTree*: _GenTreeParent, \
    const GenTree*: _GenTreeParent, \
    GenTreeStr*: _GenTreeStrParent, \
    const GenTreeStr*: _GenTreeStrParent, \
    default: PBErrInvalidPolymorphism) (Tree)

#define GenTreeSubtrees(Tree) _Generic(Tree, \
    GenTree*: _GenTreeSubtrees, \
    const GenTree*: _GenTreeSubtrees, \
    GenTreeStr*: _GenTreeStrSubtrees, \
    const GenTreeStr*: _GenTreeStrSubtrees, \
    default: PBErrInvalidPolymorphism) (Tree)

#define GenTreeData(Tree) _Generic(Tree, \
    GenTree*: _GenTreeData, \
    const GenTree*: _GenTreeData, \
    GenTreeStr*: _GenTreeStrData, \
    const GenTreeStr*: _GenTreeStrData, \
    default: PBErrInvalidPolymorphism) (Tree)

#define GenTreeSetData(Tree, Data) _Generic(Tree, \
    GenTree*: _GenTreeSetData, \
    GenTreeStr*: _GenTreeStrSetData, \
    default: PBErrInvalidPolymorphism) (Tree, Data)

#define GenTreeCut(Tree) _Generic(Tree, \
    GenTree*: _GenTreeCut, \
    GenTreeStr*: _GenTreeCut, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeIsRoot(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIsRoot, \
    const GenTree*: _GenTreeIsRoot, \
    GenTreeStr*: _GenTreeIsRoot, \
    const GenTreeStr*: _GenTreeIsRoot, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeIsLeaf(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIsLeaf, \
    const GenTree*: _GenTreeIsLeaf, \
    GenTreeStr*: _GenTreeIsLeaf, \
    const GenTreeStr*: _GenTreeIsLeaf, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeGetSize(Tree) _Generic(Tree, \
    GenTree*: _GenTreeGetSize, \
    const GenTree*: _GenTreeGetSize, \
    GenTreeStr*: _GenTreeGetSize, \
    const GenTreeStr*: _GenTreeGetSize, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreePushData(Tree, Data) _Generic(Tree, \
    GenTree*: _GenTreePushData, \
    GenTreeStr*: _GenTreeStrPushData, \
    default: PBErrInvalidPolymorphism) (Tree, Data);

#define GenTreeAddSortData(Tree, Data, SortVal) _Generic(Tree, \
    GenTree*: _GenTreeAddSortData, \

```

```

    GenTreeStr*: _GenTreeStrAddSortData, \
    default: PBErriInvalidPolymorphism) (Tree, Data, SortVal);

#define GenTreeInsertData(Tree, Data, Pos) _Generic(Tree, \
    GenTree*: _GenTreeInsertData, \
    GenTreeStr*: _GenTreeStrInsertData, \
    default: PBErriInvalidPolymorphism) (Tree, Data, Pos);

#define GenTreeAppendData(Tree, Data) _Generic(Tree, \
    GenTree*: _GenTreeAppendData, \
    GenTreeStr*: _GenTreeStrAppendData, \
    default: PBErriInvalidPolymorphism) (Tree, Data);

#define GenTreeSubtree(Tree, ISubtree) _Generic(Tree, \
    GenTree*: _GenTreeSubtree, \
    const GenTree*: _GenTreeSubtree, \
    GenTreeStr*: _GenTreeStrSubtree, \
    const GenTreeStr*: _GenTreeStrSubtree, \
    default: PBErriInvalidPolymorphism) (Tree, ISubtree)

#define GenTreeFirstSubtree(Tree) _Generic(Tree, \
    GenTree*: _GenTreeFirstSubtree, \
    const GenTree*: _GenTreeFirstSubtree, \
    GenTreeStr*: _GenTreeStrFirstSubtree, \
    const GenTreeStr*: _GenTreeStrFirstSubtree, \
    default: PBErriInvalidPolymorphism) (Tree)

#define GenTreeLastSubtree(Tree) _Generic(Tree, \
    GenTree*: _GenTreeLastSubtree, \
    const GenTree*: _GenTreeLastSubtree, \
    GenTreeStr*: _GenTreeStrLastSubtree, \
    const GenTreeStr*: _GenTreeStrLastSubtree, \
    default: PBErriInvalidPolymorphism) (Tree)

#define GenTreePopSubtree(Tree) _Generic(Tree, \
    GenTree*: _GenTreePopSubtree, \
    GenTreeStr*: _GenTreeStrPopSubtree, \
    default: PBErriInvalidPolymorphism) (Tree)

#define GenTreeDropSubtree(Tree) _Generic(Tree, \
    GenTree*: _GenTreeDropSubtree, \
    GenTreeStr*: _GenTreeStrDropSubtree, \
    default: PBErriInvalidPolymorphism) (Tree)

#define GenTreeRemoveSubtree(Tree, ISubTree) _Generic(Tree, \
    GenTree*: _GenTreeRemoveSubtree, \
    GenTreeStr*: _GenTreeStrRemoveSubtree, \
    default: PBErriInvalidPolymorphism) (Tree, ISubTree)

#define GenTreePushSubtree(Tree, SubTree) _Generic(Tree, \
    GenTree*: _GenTreePushSubtree, \
    GenTreeStr*: _GenTreeStrPushSubtree, \
    default: PBErriInvalidPolymorphism) (Tree, SubTree)

#define GenTreeAddSortSubtree(Tree, SubTree, SortVal) _Generic(Tree, \
    GenTree*: _GenTreeAddSortSubtree, \
    GenTreeStr*: _GenTreeStrAddSortSubtree, \
    default: PBErriInvalidPolymorphism) (Tree, SubTree, SortVal)

#define GenTreeInsertSubtree(Tree, SubTree, Pos) _Generic(Tree, \
    GenTree*: _GenTreeInsertSubtree, \
    GenTreeStr*: _GenTreeStrInsertSubtree, \

```

```

    default: PBErrInvalidPolymorphism) (Tree, SubTree, Pos)

#define GenTreeAppendSubtree(Tree, SubTree) _Generic(Tree, \
    GenTree*: _GenTreeAppendSubtree, \
    GenTreeStr*: _GenTreeStrAppendSubtree, \
    default: PBErrInvalidPolymorphism) (Tree, SubTree)

#define GenTreeAppendToNode(Tree, Data, Node, Iter) _Generic(Tree, \
    GenTree*: _GenTreeAppendToNode, \
    GenTreeStr*: _GenTreeAppendToNode, \
    default: PBErrInvalidPolymorphism) ((GenTree*)Tree, Data, Node, \
    (GenTreeIter*)Iter)

#define GenTreeSearch(Tree, Data, Iter) _Generic(Tree, \
    GenTree*: _GenTreeSearch, \
    const GenTree*: _GenTreeSearch, \
    GenTreeStr*: _GenTreeSearch, \
    const GenTreeStr*: _GenTreeSearch, \
    default: PBErrInvalidPolymorphism) ((GenTree*)Tree, Data, \
    (GenTreeIter*)Iter)

#define GenTreeIterDepthCreate(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIterDepthCreate, \
    const GenTree*: _GenTreeIterDepthCreate, \
    GenTreeStr*: _GenTreeIterDepthCreate, \
    const GenTreeStr*: _GenTreeIterDepthCreate, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeIterDepthCreateStatic(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIterDepthCreateStatic, \
    const GenTree*: _GenTreeIterDepthCreateStatic, \
    GenTreeStr*: _GenTreeIterDepthCreateStatic, \
    const GenTreeStr*: _GenTreeIterDepthCreateStatic, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeIterBreadthCreate(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIterBreadthCreate, \
    const GenTree*: _GenTreeIterBreadthCreate, \
    GenTreeStr*: _GenTreeIterBreadthCreate, \
    const GenTreeStr*: _GenTreeIterBreadthCreate, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeIterBreadthCreateStatic(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIterBreadthCreateStatic, \
    const GenTree*: _GenTreeIterBreadthCreateStatic, \
    GenTreeStr*: _GenTreeIterBreadthCreateStatic, \
    const GenTreeStr*: _GenTreeIterBreadthCreateStatic, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeIterValueCreate(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIterValueCreate, \
    const GenTree*: _GenTreeIterValueCreate, \
    GenTreeStr*: _GenTreeIterValueCreate, \
    const GenTreeStr*: _GenTreeIterValueCreate, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

#define GenTreeIterValueCreateStatic(Tree) _Generic(Tree, \
    GenTree*: _GenTreeIterValueCreateStatic, \
    const GenTree*: _GenTreeIterValueCreateStatic, \
    GenTreeStr*: _GenTreeIterValueCreateStatic, \
    const GenTreeStr*: _GenTreeIterValueCreateStatic, \
    default: PBErrInvalidPolymorphism) ((GenTree*)(Tree))

```

```

#define GenTreeIterFree(RefIter) _Generic(RefIter, \
    GenTreeIter*: _GenTreeIterFree, \
    GenTreeIterDepth*: _GenTreeIterFree, \
    GenTreeIterBreadth*: _GenTreeIterFree, \
    GenTreeIterValue*: _GenTreeIterFree, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter**)(RefIter))

#define GenTreeIterFreeStatic(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterFreeStatic, \
    GenTreeIterDepth*: _GenTreeIterFreeStatic, \
    GenTreeIterBreadth*: _GenTreeIterFreeStatic, \
    GenTreeIterValue*: _GenTreeIterFreeStatic, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterReset(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterReset, \
    GenTreeIterDepth*: _GenTreeIterReset, \
    GenTreeIterBreadth*: _GenTreeIterReset, \
    GenTreeIterValue*: _GenTreeIterReset, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterToEnd(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterToEnd, \
    GenTreeIterDepth*: _GenTreeIterToEnd, \
    GenTreeIterBreadth*: _GenTreeIterToEnd, \
    GenTreeIterValue*: _GenTreeIterToEnd, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterStep(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterStep, \
    GenTreeIterDepth*: _GenTreeIterStep, \
    GenTreeIterBreadth*: _GenTreeIterStep, \
    GenTreeIterValue*: _GenTreeIterStep, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterStepBack(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterStepBack, \
    GenTreeIterDepth*: _GenTreeIterStepBack, \
    GenTreeIterBreadth*: _GenTreeIterStepBack, \
    GenTreeIterValue*: _GenTreeIterStepBack, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterApply(Iter, Fun, Param) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterApply, \
    GenTreeIterDepth*: _GenTreeIterApply, \
    GenTreeIterBreadth*: _GenTreeIterApply, \
    GenTreeIterValue*: _GenTreeIterApply, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter), Fun, Param)

#define GenTreeIterIsFirst(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterIsFirst, \
    const GenTreeIter*: _GenTreeIterIsFirst, \
    GenTreeIterDepth*: _GenTreeIterIsFirst, \
    const GenTreeIterDepth*: _GenTreeIterIsFirst, \
    GenTreeIterBreadth*: _GenTreeIterIsFirst, \
    const GenTreeIterBreadth*: _GenTreeIterIsFirst, \
    GenTreeIterValue*: _GenTreeIterIsFirst, \
    const GenTreeIterValue*: _GenTreeIterIsFirst, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterIsLast(Iter) _Generic(Iter, \

```

```

GenTreeIter*: _GenTreeIterIsLast, \
const GenTreeIter*: _GenTreeIterIsLast, \
GenTreeIterDepth*: _GenTreeIterIsLast, \
const GenTreeIterDepth*: _GenTreeIterIsLast, \
GenTreeIterBreadth*: _GenTreeIterIsLast, \
const GenTreeIterBreadth*: _GenTreeIterIsLast, \
GenTreeIterValue*: _GenTreeIterIsLast, \
const GenTreeIterValue*: _GenTreeIterIsLast, \
default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterSetGenTree(Iter, Tree) _Generic(Iter, \
    GenTreeIterDepth*: _GenTreeIterDepthSetGenTree, \
    GenTreeIterBreadth*: _GenTreeIterBreadthSetGenTree, \
    GenTreeIterValue*: _GenTreeIterValueSetGenTree, \
    default: PBErrInvalidPolymorphism) (Iter, Tree)

#define GenTreeIterGetData(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterGetData, \
    const GenTreeIter*: _GenTreeIterGetData, \
    GenTreeIterDepth*: _GenTreeIterGetData, \
    const GenTreeIterDepth*: _GenTreeIterGetData, \
    GenTreeIterBreadth*: _GenTreeIterGetData, \
    const GenTreeIterBreadth*: _GenTreeIterGetData, \
    GenTreeIterValue*: _GenTreeIterGetData, \
    const GenTreeIterValue*: _GenTreeIterGetData, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterGenTree(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterGenTree, \
    const GenTreeIter*: _GenTreeIterGenTree, \
    GenTreeIterDepth*: _GenTreeIterGenTree, \
    const GenTreeIterDepth*: _GenTreeIterGenTree, \
    GenTreeIterBreadth*: _GenTreeIterGenTree, \
    const GenTreeIterBreadth*: _GenTreeIterGenTree, \
    GenTreeIterValue*: _GenTreeIterGenTree, \
    const GenTreeIterValue*: _GenTreeIterGenTree, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterGetGenTree(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterGetGenTree, \
    const GenTreeIter*: _GenTreeIterGetGenTree, \
    GenTreeIterDepth*: _GenTreeIterGetGenTree, \
    const GenTreeIterDepth*: _GenTreeIterGetGenTree, \
    GenTreeIterBreadth*: _GenTreeIterGetGenTree, \
    const GenTreeIterBreadth*: _GenTreeIterGetGenTree, \
    GenTreeIterValue*: _GenTreeIterGetGenTree, \
    const GenTreeIterValue*: _GenTreeIterGetGenTree, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterSeq(Iter) _Generic(Iter, \
    GenTreeIter*: _GenTreeIterSeq, \
    const GenTreeIter*: _GenTreeIterSeq, \
    GenTreeIterDepth*: _GenTreeIterSeq, \
    const GenTreeIterDepth*: _GenTreeIterSeq, \
    GenTreeIterBreadth*: _GenTreeIterSeq, \
    const GenTreeIterBreadth*: _GenTreeIterSeq, \
    GenTreeIterValue*: _GenTreeIterSeq, \
    const GenTreeIterValue*: _GenTreeIterSeq, \
    default: PBErrInvalidPolymorphism) ((GenTreeIter*)(Iter))

#define GenTreeIterUpdate(Iter) _Generic(Iter, \
    GenTreeIterDepth*: GenTreeIterDepthUpdate, \

```

```

    GenTreeIterBreadth*: GenTreeIterBreadthUpdate, \
    GenTreeIterValue*: GenTreeIterValueUpdate, \
    default: PBErrInvalidPolymorphism) (Iter)

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

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

#endif

```

2 Code

2.1 gtree.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 GenTreeFreeRec(GSetGenTree* subtrees);

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

// Create a new GenTree
GenTree* GenTreeCreate(void) {
    // Declare the new tree
    GenTree *that = PBErrMalloc(GenTreeErr, sizeof(GenTree));
    // Set properties
    that->_parent = NULL;
    that->_subtrees = GSetGenTreeCreateStatic();
    that->_data = NULL;
    // Return the tree
    return that;
}

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

// Create a new GenTree with user data 'data'

```

```

GenTree* GenTreeCreateData(void* const data) {
    // Declare the new tree
    GenTree *that = PBErrMalloc(GenTreeErr, sizeof(GenTree));
    // Set properties
    that->_parent = NULL;
    that->_subtrees = GSetGenTreeCreateStatic();
    that->_data = data;
    // Return the tree
    return that;
}

// Free the memory used by the GenTree '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 _GenTreeFree(GenTree** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        // Nothing to do
        return;
    // If it's not a root node
    if (!GenTreeIsRoot(*that))
        // Cut the tree
        GenTreeCut(*that);
    // Free recursively the memory
    GenTreeFreeRec(GenTreeSubtrees(*that));
    free(*that);
    *that = NULL;
}

// Free the memory used by 'subtrees' recursively
void GenTreeFreeRec(GSetGenTree* subtrees) {
    while (GSetNbElem(subtrees) > 0) {
        GenTree* tree = GSetPop(subtrees);
        GenTreeFreeRec(GenTreeSubtrees(tree));
        free(tree);
    }
}

// Free the memory used by the static GenTree '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 _GenTreeFreeStatic(GenTree* that) {
    // Check argument
    if (that == NULL)
        // Nothing to do
        return;
    // If it's not a root node
    if (!GenTreeIsRoot(that))
        // Cut the tree
        GenTreeCut(that);
    // Free memory
    GenTreeFreeRec(GenTreeSubtrees(that));
}

// Disconnect the GenTree 'that' from its parent
// If it has no parent, do nothing
void _GenTreeCut(GenTree* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GenTreeErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(GSetErr->_msg, "'that' is null");
        PBErCatch(GSetErr);
    }
#endif
    // If there is no parent
    if (GenTreeParent(that) == NULL)
        // Nothing to do
        return;
    // Remove the tree from the parent's subtrees
    GSetRemoveAll(GenTreeSubtrees(GenTreeParent(that)), that);
    // Cut the link to the parent
    that->_parent = NULL;
}

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

// Append a new node with 'data' to the first node containing 'node'
// in the GenTree 'that'
// Uses the iterator 'iter' to search the node
// Return true if we could find 'node', false else
bool _GenTreeAppendToNode(GenTree* const that,
    void* const data, void* const node, GenTreeIter* const iter) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GenTreeErr->_type = PBErTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PBErCatch(GSetErr);
        }
        if (iter == NULL) {
            GenTreeErr->_type = PBErTypeNullPointer;
            sprintf(GSetErr->_msg, "'iter' is null");
            PBErCatch(GSetErr);
        }
    #endif
    // Declare a variable to memorize the result
    bool res = false;
    // Search the node where to append the data

```



```

    GenTree* nodeTree = GenTreeSearch(that, node, iter);
    // If we could find the node
    if (nodeTree != NULL) {
        // Append the data
        GenTreeAppendData(nodeTree, data);
        // Update the result
        res = true;
    }
    // Return the result
    return res;
}

// Search the first node containing 'data' in the GenTree 'that'
// Uses the iterator 'iter' to search the node. Do not reset the
// iterator, thus several calls with the same iterator will return
// eventual successive nodes containing the same data. If one want to
// loop on these nodes, the proper stopping condition is
// while(GenTreeSearch() != NULL && GenTreeIterIsLast() == false)
// Return the node if we could find 'data', null else
GenTree* _GenTreeSearch(const GenTree* const that,
    const void* const data, GenTreeIter* const iter) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (iter == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'iter' is null");
        PBErrCatch(GSetErr);
    }
#endif
    (void)that;
    // Declare a variable to memorize the result
    GenTree* res = NULL;
    // Loop until we have found or reached the end
    do {
        // If we have found the searched data
        if (GenTreeIterGetData(iter) == data)
            // Memorize the node containing the data
            res = GenTreeIterGetGenTree(iter);
    } while (GenTreeIterStep(iter) && res == NULL);
    // Return the result
    return res;
}

// ----- GenTreeIter

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

// Create recursively the sequence of an iterator for depth first
void GenTreeIterCreateSequenceDepthFirst(GSetGenTree* seq, GenTree* tree);

// Create recursively the sequence of an iterator for breadth first
void GenTreeIterCreateSequenceBreadthFirst(GSetGenTree* seq, GenTree* tree,
    int lvl);

// Create recursively the sequence of an iterator for value first
void GenTreeIterCreateSequenceValueFirst(GSetGenTree* seq, GenTree* tree,
    float val);

```

```

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

// Create a new GenTreeIterDepth for the GenTree 'tree'
GenTreeIterDepth* _GenTreeIterDepthCreate(GenTree* const tree) {
#if BUILDMODE == 0
    if (tree == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GenTreeIterDepth *iter = PBErrMalloc(GenTreeErr, sizeof(GenTreeIterDepth));
    // Set properties
    ((GenTreeIter*)iter)->_tree = tree;
    ((GenTreeIter*)iter)->_seq = GSetGenTreeCreateStatic();
    GenTreeIterDepthUpdate(iter);
    GenTreeIterReset(iter);
    // Return the iterator
    return iter;
}

// Create a new static GenTreeIterDepth for the GenTree 'tree'
GenTreeIterDepth _GenTreeIterDepthCreateStatic(GenTree* const tree) {
#if BUILDMODE == 0
    if (tree == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GenTreeIterDepth iter;
    // Set properties
    ((GenTreeIter*)&iter)->_tree = tree;
    ((GenTreeIter*)&iter)->_seq = GSetGenTreeCreateStatic();
    GenTreeIterDepthUpdate(&iter);
    GenTreeIterReset(&iter);
    // Return the iterator
    return iter;
}

// Create a new GenTreeIterBreadth for the GenTree 'tree'
GenTreeIterBreadth* _GenTreeIterBreadthCreate(GenTree* const tree) {
#if BUILDMODE == 0
    if (tree == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GenTreeIterBreadth *iter =
        PBErrMalloc(GenTreeErr, sizeof(GenTreeIterBreadth));
    // Set properties
    ((GenTreeIter*)iter)->_tree = tree;
    ((GenTreeIter*)iter)->_seq = GSetGenTreeCreateStatic();
    GenTreeIterBreadthUpdate(iter);
    GenTreeIterReset(iter);
    // Return the iterator
    return iter;
}

```

```

// Create a new static GenTreeIterBreadth for the GenTree 'tree'
GenTreeIterBreadth _GenTreeIterBreadthCreateStatic(GenTree* const tree) {
#ifdef BUILDMODE == 0
    if (tree == NULL) {
        GenTreeErr->_type = PErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GenTreeIterBreadth iter;
    // Set properties
    ((GenTreeIter*)&iter)->_tree = tree;
    ((GenTreeIter*)&iter)->_seq = GSetGenTreeCreateStatic();
    GenTreeIterBreadthUpdate(&iter);
    GenTreeIterReset(&iter);
    // Return the iterator
    return iter;
}

// Create a new GenTreeIterValue for the GenTree 'tree'
GenTreeIterValue* _GenTreeIterValueCreate(GenTree* const tree) {
#ifdef BUILDMODE == 0
    if (tree == NULL) {
        GenTreeErr->_type = PErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GenTreeIterValue *iter = PErrMalloc(GenTreeErr, sizeof(GenTreeIterValue));
    // Set properties
    ((GenTreeIter*)iter)->_tree = tree;
    ((GenTreeIter*)iter)->_seq = GSetGenTreeCreateStatic();
    GenTreeIterValueUpdate(iter);
    GenTreeIterReset(iter);
    // Return the iterator
    return iter;
}

// Create a new static GenTreeIterValue for the GenTree 'tree' with
// 'rootval' the value of its root node
GenTreeIterValue _GenTreeIterValueCreateStatic(GenTree* const tree) {
#ifdef BUILDMODE == 0
    if (tree == NULL) {
        GenTreeErr->_type = PErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PErrCatch(GSetErr);
    }
#endif
    // Declare the new iterator
    GenTreeIterValue iter;
    // Set properties
    ((GenTreeIter*)&iter)->_tree = tree;
    ((GenTreeIter*)&iter)->_seq = GSetGenTreeCreateStatic();
    GenTreeIterValueUpdate(&iter);
    GenTreeIterReset(&iter);
    // Return the iterator
    return iter;
}

```

```

// Update the GenTreeIterDepth 'that' in case its attached GenTree has been
// modified
// The node sequence doesn't include the root node of the attached tree
void GenTreeIterDepthUpdate(GenTreeIterDepth* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PErrCatch(GSetErr);
    }
#endif
    // Flush the sequence
    GSetFlush(GenTreeIterSeq(that));
    // Create the sequence with a Depth First run through nodes of the tree
    GenTreeIterCreateSequenceDepthFirst(GenTreeIterSeq(that),
        GenTreeIterGenTree(that));
    // Reset the current position
    GenTreeIterReset(that);
}

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

// Update the GenTreeIterBreadth 'that' in case its attached GenTree has
// been modified
// The node sequence doesn't include the root node of the attached tree
void GenTreeIterBreadthUpdate(GenTreeIterBreadth* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PErrCatch(GSetErr);
    }
#endif
    // Flush the sequence
    GSetFlush(GenTreeIterSeq(that));
    // Create the sequence with a Breadth First run through nodes of
    // the tree
    GenTreeIterCreateSequenceBreadthFirst(GenTreeIterSeq(that),
        GenTreeIterGenTree(that), 0);
    // Reset the current position
    GenTreeIterReset(that);
}

// Create recursively the sequence of an iterator for breadth first
void GenTreeIterCreateSequenceBreadthFirst(GSetGenTree* seq, GenTree* tree,
    int lvl) {
    // Append the current tree to the sequence if it's not root
    if (!GenTreeIsRoot(tree)) GSetAddSort(seq, tree, lvl);
}

```

```

// If there are subtrees
if (GSetNbElem(GenTreeSubtrees(tree)) > 0) {
    // Declare a variable to memorize the next lvl
    int nextLvl = lvl + 1;
    // Append the subtrees recursively
    GSetIterForward iter =
        GSetIterForwardCreateStatic(GenTreeSubtrees(tree));
    do {
        GenTree* subtree = GSetIterGet(&iter);
        GenTreeIterCreateSequenceBreadthFirst(seq, subtree, nextLvl);
    } while (GSetIterStep(&iter));
}
}

// Update the GenTreeIterValue 'that' in case its attached GenTree has been
// modified
// The node sequence doesn't include the root node of the attached tree
void GenTreeIterValueUpdate(GenTreeIterValue* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Flush the sequence
    GSetFlush(GenTreeIterSeq(that));
    // Create the sequence with a Value First run through nodes of the tree
    GenTreeIterCreateSequenceValueFirst(GenTreeIterSeq(that),
        GenTreeIterGenTree(that), 0.0);
    // Reset the current position
    GenTreeIterReset(that);
}

// Create recursively the sequence of an iterator for value first
void GenTreeIterCreateSequenceValueFirst(GSetGenTree* seq, GenTree* tree,
    float val) {
    // Append the current tree to the sequence if it's not root
    if (!GenTreeIsRoot(tree)) GSetAddSort(seq, tree, val);
    // If there are subtrees
    if (GSetNbElem(GenTreeSubtrees(tree)) > 0) {
        // Append the subtrees recursively
        GSetIterForward iter =
            GSetIterForwardCreateStatic(GenTreeSubtrees(tree));
        do {
            GenTree* subtree = GSetIterGet(&iter);
            GenTreeIterCreateSequenceValueFirst(seq, subtree,
                GSetIterGetSortVal(&iter));
        } while (GSetIterStep(&iter));
    }
}

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

```

```

}

// Free the memory used by the static iterator 'that'
void _GenTreeIterFreeStatic(GenTreeIter* const that) {
    // Check argument
    if (that == NULL)
        // Nothing to do
        return;
    // Free memory
    GSetFlush(GenTreeIterSeq(that));
}

```

2.2 gtree-inline.c

```

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

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

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

// Get the user data of the GenTree 'that'
#if BUILDMODE != 0
inline
#endif
void* _GenTreeData(const GenTree* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GenTreeErr->_type = PErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PErrCatch(GSetErr);
        }
    #endif
    return that->_data;
}

// Get the parent of the GenTree 'that'
#if BUILDMODE != 0
inline
#endif
GenTree* _GenTreeParent(const GenTree* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GenTreeErr->_type = PErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PErrCatch(GSetErr);
        }
    #endif
    return that->_parent;
}

// Set the user data of the GenTree 'that' to 'data'
#if BUILDMODE != 0
inline
#endif
void _GenTreeSetData(GenTree* const that, void* const data) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GenTreeErr->_type = PErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");

```

```

        PBErCatch(GSetErr);
    }
#endif
    that->_data = data;
}

// Get the set of subtrees of the GenTree 'that'
#if BUILDMODE != 0
inline
#endif
GSetGenTree* _GenTreeSubtrees(const GenTree* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErCatch(GSetErr);
    }
#endif
    return (GSetGenTree*)&(that->_subtrees);
}

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

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

// ----- GenTreeIter

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

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

// Reset the iterator 'that' at its start position
#if BUILDMODE != 0
inline

```

```

#endif
void _GenTreeIterReset(GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    that->_curPos = ((GSet*)&(that->_seq))->_head;
}

// Reset the iterator 'that' to its end position
#if BUILDMODE != 0
inline
#endif
void _GenTreeIterToEnd(GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_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 _GenTreeIterStep(GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (that->_curPos == NULL) {
        GenTreeErr->_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 _GenTreeIterStepBack(GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {

```



```

    GenTreeErr->_type = PBErrTypeNullPointer;
    sprintf(GSetErr->_msg, "'that' is null");
    PBErrCatch(GSetErr);
}
if (that->_curPos == NULL) {
    GenTreeErr->_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 GenTree 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 _GenTreeIterApply(GenTreeIter* const that,
    void(*fun)(void* const data, void* const param), void* const param) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
    if (fun == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'fun' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Reset the iterator;
    GenTreeIterReset(that);
    // If the associated tree is not empty
    if (GSetNbElem(&(that->_seq)) > 0) {
        // For each node of the tree
        do {
            // Apply the user function
            fun(GenTreeIterGetData(that), param);
        } while (GenTreeIterStep(that));
    }
}

// Return true if the iterator is at the start of the elements (from
// its point of view, not the order in the GenTree)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool _GenTreeIterIsFirst(const GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(GSetErr->_msg, "'that' is null");
        PBErCatch(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 GenTree)
// Return false else
#if BUILDMODE != 0
inline
#endif
bool _GenTreeIterIsLast(const GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErCatch(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 _GenTreeIterDepthSetGenTree(GenTreeIterDepth* const that,
    GenTree* const tree) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErCatch(GSetErr);
    }
    if (tree == NULL) {
        GenTreeErr->_type = PBErTypeNullPointer;
        sprintf(GSetErr->_msg, "'tree' is null");
        PBErCatch(GSetErr);
    }
#endif
    // Set the tree
    ((GenTreeIter*)that)->_tree = tree;
    // Update the sequence
    GenTreeIterDepthUpdate(that);
    // Reset the iterator
    GenTreeIterReset(that);
}
#if BUILDMODE != 0
inline
#endif
void _GenTreeIterBreadthSetGenTree(GenTreeIterBreadth* const that,
    GenTree* tree) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErCatch(GSetErr);
    }
    if (tree == NULL) {
        GenTreeErr->_type = PBErTypeNullPointer;
    }
#endif
}

```

```

        sprintf(GSetErr->_msg, "'tree' is null");
        PBErrCatch(GSetErr);
    }
#endif
    // Set the tree
    ((GenTreeIter*)that)->_tree = tree;
    // Update the sequence
    GenTreeIterBreadthUpdate(that);
    // Reset the iterator
    GenTreeIterReset(that);
}
#if BUILDMODE != 0
inline
#endif
void _GenTreeIterValueSetGenTree(GenTreeIterValue* const that,
    GenTree* const tree) {
    #if BUILDMODE == 0
        if (that == NULL) {
            GenTreeErr->_type = PBErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'that' is null");
            PBErrCatch(GSetErr);
        }
        if (tree == NULL) {
            GenTreeErr->_type = PBErrTypeNullPointer;
            sprintf(GSetErr->_msg, "'tree' is null");
            PBErrCatch(GSetErr);
        }
    #endif
    // Set the tree
    ((GenTreeIter*)that)->_tree = tree;
    // Update the sequence
    GenTreeIterValueUpdate(that);
    // Reset the iterator
    GenTreeIterReset(that);
}

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

// Return the tree currently pointed to by the iterator
#if BUILDMODE != 0
inline
#endif
GenTree* _GenTreeIterGetGenTree(const GenTreeIter* const that) {
    #if BUILDMODE == 0

```

```

    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return (GenTree*)(that->_curPos->_data);
}

// Return the tree associated to the iterator 'that'
#if BUILDMODE != 0
inline
#endif
GenTree* _GenTreeIterGenTree(const GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return that->_tree;
}

// Return the sequence of the iterator 'that'
#if BUILDMODE != 0
inline
#endif
GSetGenTree* _GenTreeIterSeq(const GenTreeIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GenTreeErr->_type = PBErrTypeNullPointer;
        sprintf(GSetErr->_msg, "'that' is null");
        PBErrCatch(GSetErr);
    }
#endif
    return (GSetGenTree*)&(that->_seq);
}

```

3 Makefile

```

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

all: pbmake_wget main

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

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

# Rules to make the executable

```

```

repo=gtree
$$($(repo)_EXENAME): \
$$($(repo)_EXENAME).o \
$$($(repo)_EXE_DEP) \
$$($(repo)_DEP)
$(COMPILER) 'echo "$($(repo)_EXE_DEP) $($($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u' $(LINK_ARG) $($($(repo)_LINK_ARG)

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

```

4 Unit tests

```

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

#define RANDOMSEED 0

void UnitTestGenTreeCreateFree() {
    GenTree* tree = GenTreeCreate();
    if (tree == NULL ||
        tree->_parent != NULL ||
        GSetNbElem(&(tree->_subtrees)) != 0 ||
        tree->_data != NULL) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeCreate failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeFree(&tree);
    if (tree != NULL) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeFree failed");
        PBErrCatch(GenTreeErr);
    }
    int data = 1;
    tree = GenTreeCreateData(&data);
    if (tree == NULL ||
        tree->_parent != NULL ||
        GSetNbElem(&(tree->_subtrees)) != 0 ||
        tree->_data != &data) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeCreateData failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeFree(&tree);
    GenTree treeStatic = GenTreeCreateStatic();
    if (treeStatic._parent != NULL ||
        GSetNbElem(&(treeStatic._subtrees)) != 0 ||
        treeStatic._data != NULL) {

```

```

        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeCreateStatic failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeFreeStatic(&treeStatic);
    printf("UnitTestGenTreeCreateFree OK\n");
}

void UnitTestGenTreeGetSet() {
    GenTree tree = GenTreeCreateStatic();
    int data = 1;
    tree._data = &data;
    if (GenTreeData(&tree) != &data) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeData failed");
        PBErrCatch(GenTreeErr);
    }
    int data2 = 1;
    GenTreeSetData(&tree, &data2);
    if (GenTreeData(&tree) != &data2) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeSetData failed");
        PBErrCatch(GenTreeErr);
    }
    if (GenTreeSubtrees(&tree) != &(tree._subtrees)) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeSubTrees failed");
        PBErrCatch(GenTreeErr);
    }
    if (GenTreeIsRoot(&tree) == false) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIsRoot failed");
        PBErrCatch(GenTreeErr);
    }
    tree._parent = &tree;
    if (GenTreeIsRoot(&tree) == true) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIsRoot failed");
        PBErrCatch(GenTreeErr);
    }
    if (GenTreeParent(&tree) != &tree) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeParent failed");
        PBErrCatch(GenTreeErr);
    }
    tree._parent = NULL;
    if (GenTreeIsLeaf(&tree) == false) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIsLeaf failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeAppendData(&tree, &data);
    if (GenTreeIsLeaf(&tree) == true) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIsLeaf failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeFreeStatic(&tree);
    printf("UnitTestGenTreeGetSet OK\n");
}

void UnitTestGenTreeCutGetSize() {

```

```

GenTree tree = GenTreeCreateStatic();
int data = 1;
GenTreeAppendData(&tree, &data);
GenTreeAppendData(&tree, &data);
GenTreeAppendData(GenTreeSubtree(&tree, 1), &data);
if (GenTreeGetSize(&tree) != 3) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeGetSize failed");
    PBErrCatch(GenTreeErr);
}
GenTree* cuttree = GenTreeSubtree(&tree, 1);
GenTreeCut(cuttree);
if (GenTreeGetSize(&tree) != 1 ||
    GenTreeGetSize(cuttree) != 1) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeCut failed");
    PBErrCatch(GenTreeErr);
}
GenTreeFreeStatic(&tree);
GenTreeFree(&cuttree);
printf("UnitTestGenTreeCutGetSize OK\n");
}

void UnitTestGenTreeSearchAppendToNode() {
    GenTree tree = GenTreeCreateStatic();
    int data[4] = {1, 2, 3, 4};
    GenTreeAppendData(&tree, data);
    GenTreeAppendData(&tree, data + 1);
    GenTreeAppendData(GenTreeSubtree(&tree, 1), data + 2);
    GenTreeIterDepth iter = GenTreeIterDepthCreateStatic(&tree);
    if (GenTreeSearch(&tree, data + 2, &iter) !=
        GenTreeSubtree(GenTreeSubtree(&tree, 1), 0)) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeSearch failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeIterReset(&iter);
    if (!GenTreeAppendToNode(&tree, data + 3, data + 1, &iter)) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeAppendToNode failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeIterFreeStatic(&iter);
    iter = GenTreeIterDepthCreateStatic(&tree);
    if (GenTreeSearch(&tree, data + 3, &iter) !=
        GenTreeSubtree(GenTreeSubtree(&tree, 1), 1)) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeAppendToNode failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeIterFreeStatic(&iter);
    GenTreeFreeStatic(&tree);
    printf("UnitTestGenTreeSearchAppendToNode OK\n");
}

void UnitTestGenTree() {
    UnitTestGenTreeCreateFree();
    UnitTestGenTreeGetSet();
    UnitTestGenTreeCutGetSize();
    UnitTestGenTreeSearchAppendToNode();
    printf("UnitTestGenTree OK\n");
}

```

```

int dataExampleTree[10] = {0,1,2,3,4,5,6,7,8,9};
GenTree* GetExampleTree() {
    GenTree* tree = GenTreeCreate();
    GenTreeAddSortData(tree, dataExampleTree + 0, 0);
    GenTreeAddSortData(tree, dataExampleTree + 9, 9);
    GenTree* subtree = GenTreeSubtree(tree, 0);
    GenTreeAddSortData(subtree, dataExampleTree + 1, 1);
    GenTreeAddSortData(subtree, dataExampleTree + 2, 2);
    subtree = GenTreeSubtree(tree, 1);
    GenTreeAddSortData(subtree, dataExampleTree + 3, 3);
    GenTreeAddSortData(subtree, dataExampleTree + 4, 4);
    subtree = GenTreeSubtree(subtree, 0);
    GenTreeAddSortData(subtree, dataExampleTree + 8, 8);
    GenTreeAddSortData(subtree, dataExampleTree + 6, 6);
    subtree = GenTreeSubtree(subtree, 1);
    GenTreeAddSortData(subtree, dataExampleTree + 7, 7);
    GenTreeAddSortData(subtree, dataExampleTree + 5, 5);
    return tree;
}

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

void UnitTestGenTreeIterDepth() {
    GenTree* tree = GetExampleTree();
    GenTreeIterDepth* iter = GenTreeIterDepthCreate(tree);
    if (iter == NULL ||
        iter->_iter._tree != tree ||
        GSetNbElem(&(iter->_iter._seq)) != 10 ||
        iter->_iter._curPos != iter->_iter._seq._set._head) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterDepthCreate failed");
        PBErrCatch(GenTreeErr);
    }
    int check[10] = {0,1,2,9,3,6,8,5,7,4};
    int iCheck = 0;
    do {
        int* data = GenTreeIterGetData(iter);
        if (*data != check[iCheck]) {
            GenTreeErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GenTreeErr->_msg, "GenTreeIterDepth failed");
            PBErrCatch(GenTreeErr);
        }
        ++iCheck;
    } while (GenTreeIterStep(iter));
    GenTreeIterFree(&iter);
    if (iter != NULL) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterFree failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeIterDepth iterstatic = GenTreeIterDepthCreateStatic(tree);
    if (iterstatic._iter._tree != tree ||
        GSetNbElem(&(iterstatic._iter._seq)) != 10 ||
        iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterDepthCreateStatic failed");
        PBErrCatch(GenTreeErr);
    }
    iCheck = 0;
}

```



```

do {
    int* data = GenTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterDepth failed");
        PBErrCatch(GenTreeErr);
    }
    ++iCheck;
} while (GenTreeIterStep(&iterstatic));
check[3] = 12;
dataExampleTree[9] = 12;
GenTreeIterDepthUpdate(&iterstatic);
iCheck = 0;
do {
    int* data = GenTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterUpdate failed");
        PBErrCatch(GenTreeErr);
    }
    ++iCheck;
} while (GenTreeIterStep(&iterstatic));
dataExampleTree[9] = 9;
GenTreeIterReset(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterReset failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsFirst(&iterstatic) == false) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsFirst failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsLast(&iterstatic) == true) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsLast failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterGetGenTree(&iterstatic) != GenTreeSubtree(tree, 0)) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterGetGenTree failed");
    PBErrCatch(GenTreeErr);
}
GenTreeIterToEnd(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._tail) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterToEnd failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsFirst(&iterstatic) == true) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsFirst failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsLast(&iterstatic) == false) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsLast failed");
    PBErrCatch(GenTreeErr);
}
GenTreeIterStepBack(&iterstatic);
if (iterstatic._iter._curPos->_next !=

```

```

    iterstatic._iter._seq._set._tail) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterStepBack failed");
        PBErrCatch(GenTreeErr);
    }
    if (GenTreeIterGenTree(&iterstatic) != tree) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterGenTree failed");
        PBErrCatch(GenTreeErr);
    }
    if (GenTreeIterSeq(&iterstatic) != &(iterstatic._iter._seq)) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterSeq failed");
        PBErrCatch(GenTreeErr);
    }
    char c = ',';
    GenTreeIterApply(&iterstatic, &funApply, &c);
    printf("\n");
    GenTree* treeB = GenTreeCreate();
    GenTreeIterSetGenTree(&iterstatic, treeB);
    if (GenTreeIterGenTree(&iterstatic) != treeB ||
        GSetNbElem(&(iterstatic._iter._seq)) != 0) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterSetGenTree failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeIterFreeStatic(&iterstatic);
    GenTreeFree(&tree);
    GenTreeFree(&treeB);
    printf("UnitTestGenTreeIterDepth OK\n");
}

void UnitTestGenTreeIterBreadth() {
    GenTree* tree = GetExampleTree();
    GenTreeIterBreadth* iter = GenTreeIterBreadthCreate(tree);
    if (iter == NULL ||
        iter->_iter._tree != tree ||
        GSetNbElem(&(iter->_iter._seq)) != 10 ||
        iter->_iter._curPos != iter->_iter._seq._set._head) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterBreadthCreate failed");
        PBErrCatch(GenTreeErr);
    }
    int check[10] = {0,9,1,2,3,4,6,8,5,7};
    int iCheck = 0;
    do {
        int* data = GenTreeIterGetData(iter);
        if (*data != check[iCheck]) {
            GenTreeErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GenTreeErr->_msg, "GenTreeIterBreadth failed");
            PBErrCatch(GenTreeErr);
        }
        ++iCheck;
    } while (GenTreeIterStep(iter));
    GenTreeIterFree(&iter);
    if (iter != NULL) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterFree failed");
        PBErrCatch(GenTreeErr);
    }
    GenTreeIterBreadth iterstatic = GenTreeIterBreadthCreateStatic(tree);
    if (iterstatic._iter._tree != tree ||

```

```

GSetNbElem(&(iterstatic._iter._seq)) != 10 ||
iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterBreadthCreateStatic failed");
PBErrCatch(GenTreeErr);
}
iCheck = 0;
do {
int* data = GenTreeIterGetData(&iterstatic);
if (*data != check[iCheck]) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterBreadth failed");
PBErrCatch(GenTreeErr);
}
++iCheck;
} while (GenTreeIterStep(&iterstatic));
check[1] = 12;
dataExampleTree[9] = 12;
GenTreeIterBreadthUpdate(&iterstatic);
iCheck = 0;
do {
int* data = GenTreeIterGetData(&iterstatic);
if (*data != check[iCheck]) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterUpdate failed");
PBErrCatch(GenTreeErr);
}
++iCheck;
} while (GenTreeIterStep(&iterstatic));
dataExampleTree[9] = 9;
GenTreeIterReset(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterReset failed");
PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsFirst(&iterstatic) == false) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterIsFirst failed");
PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsLast(&iterstatic) == true) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterIsLast failed");
PBErrCatch(GenTreeErr);
}
if (GenTreeIterGetGenTree(&iterstatic) != GenTreeSubtree(tree, 0)) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterGetGenTree failed");
PBErrCatch(GenTreeErr);
}
GenTreeIterToEnd(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._tail) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterToEnd failed");
PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsFirst(&iterstatic) == true) {
GenTreeErr->_type = PBErrTypeUnitTestFailed;
sprintf(GenTreeErr->_msg, "GenTreeIterIsFirst failed");
PBErrCatch(GenTreeErr);
}
}

```

```

if (GenTreeIterIsLast(&iterstatic) == false) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsLast failed");
    PBErrCatch(GenTreeErr);
}
GenTreeIterStepBack(&iterstatic);
if (iterstatic._iter._curPos->_next !=
    iterstatic._iter._seq._set._tail) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterStepBack failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterGenTree(&iterstatic) != tree) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterGenTree failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterSeq(&iterstatic) != &(iterstatic._iter._seq)) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterSeq failed");
    PBErrCatch(GenTreeErr);
}
char c = ',';
GenTreeIterApply(&iterstatic, &funApply, &c);
printf("\n");
GenTree* treeB = GenTreeCreate();
GenTreeIterSetGenTree(&iterstatic, treeB);
if (GenTreeIterGenTree(&iterstatic) != treeB ||
    GSetNbElem(&(iterstatic._iter._seq)) != 0) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterSetGenTree failed");
    PBErrCatch(GenTreeErr);
}
GenTreeIterFreeStatic(&iterstatic);
GenTreeFree(&tree);
GenTreeFree(&treeB);
printf("UnitTestGenTreeIterBreadth OK\n");
}

void UnitTestGenTreeIterValue() {
    GenTree* tree = GetExampleTree();
    GenTreeIterValue* iter = GenTreeIterValueCreate(tree);
    if (iter == NULL ||
        iter->_iter._tree != tree ||
        GSetNbElem(&(iter->_iter._seq)) != 10 ||
        iter->_iter._curPos != iter->_iter._seq._set._head) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterValueCreate failed");
        PBErrCatch(GenTreeErr);
    }
    int check[10] = {0,1,2,3,4,5,6,7,8,9};
    int iCheck = 0;
    do {
        int* data = GenTreeIterGetData(iter);
        if (*data != check[iCheck]) {
            GenTreeErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GenTreeErr->_msg, "GenTreeIterValue failed");
            PBErrCatch(GenTreeErr);
        }
        ++iCheck;
    } while (GenTreeIterStep(iter));
    GenTreeIterFree(&iter);
}

```

```

if (iter != NULL) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterFree failed");
    PBErrCatch(GenTreeErr);
}
GenTreeIterValue iterstatic = GenTreeIterValueCreateStatic(tree);
if (iterstatic._iter._tree != tree ||
    GSetNbElem(&(iterstatic._iter._seq)) != 10 ||
    iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterValueCreateStatic failed");
    PBErrCatch(GenTreeErr);
}
iCheck = 0;
do {
    int* data = GenTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterValue failed");
        PBErrCatch(GenTreeErr);
    }
    ++iCheck;
} while (GenTreeIterStep(&iterstatic));
check[9] = 12;
dataExampleTree[9] = 12;
GenTreeIterValueUpdate(&iterstatic);
iCheck = 0;
do {
    int* data = GenTreeIterGetData(&iterstatic);
    if (*data != check[iCheck]) {
        GenTreeErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenTreeErr->_msg, "GenTreeIterUpdate failed");
        PBErrCatch(GenTreeErr);
    }
    ++iCheck;
} while (GenTreeIterStep(&iterstatic));
dataExampleTree[9] = 9;
GenTreeIterReset(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._head) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterReset failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsFirst(&iterstatic) == false) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsFirst failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterIsLast(&iterstatic) == true) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsLast failed");
    PBErrCatch(GenTreeErr);
}
if (GenTreeIterGetGenTree(&iterstatic) != GenTreeSubtree(tree, 0)) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterGetGenTree failed");
    PBErrCatch(GenTreeErr);
}
GenTreeIterToEnd(&iterstatic);
if (iterstatic._iter._curPos != iterstatic._iter._seq._set._tail) {
    GenTreeErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterToEnd failed");
}

```

```

    PBErCatch(GenTreeErr);
}
if (GenTreeIterIsFirst(&iterstatic) == true) {
    GenTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsFirst failed");
    PBErCatch(GenTreeErr);
}
if (GenTreeIterIsLast(&iterstatic) == false) {
    GenTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterIsLast failed");
    PBErCatch(GenTreeErr);
}
GenTreeIterStepBack(&iterstatic);
if (iterstatic._iter._curPos->_next !=
    iterstatic._iter._seq._set._tail) {
    GenTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterStepBack failed");
    PBErCatch(GenTreeErr);
}
if (GenTreeIterGenTree(&iterstatic) != tree) {
    GenTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterGenTree failed");
    PBErCatch(GenTreeErr);
}
if (GenTreeIterSeq(&iterstatic) != &(iterstatic._iter._seq)) {
    GenTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterSeq failed");
    PBErCatch(GenTreeErr);
}
char c = ',';
GenTreeIterApply(&iterstatic, &funApply, &c);
printf("\n");
GenTree* treeB = GenTreeCreate();
GenTreeIterSetGenTree(&iterstatic, treeB);
if (GenTreeIterGenTree(&iterstatic) != treeB ||
    GSetNbElem(&(iterstatic._iter._seq)) != 0) {
    GenTreeErr->_type = PBErTypeUnitTestFailed;
    sprintf(GenTreeErr->_msg, "GenTreeIterSetGenTree failed");
    PBErCatch(GenTreeErr);
}
GenTreeIterFreeStatic(&iterstatic);
GenTreeFree(&tree);
GenTreeFree(&treeB);
printf("UnitTestGenTreeIterValue OK\n");
}

void UnitTestGenTreeIter() {
    UnitTestGenTreeIterDepth();
    UnitTestGenTreeIterBreadth();
    UnitTestGenTreeIterValue();
    printf("UnitTestGenTreeIter OK\n");
}

void UnitTestAll() {
    UnitTestGenTree();
    UnitTestGenTreeIter();
    printf("UnitTestAll OK\n");
}

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

```

```
    return 0;  
}
```

5 Unit tests output

```
UnitTestGenTreeCreateFree OK  
UnitTestGenTreeGetSet OK  
UnitTestGenTreeCutGetSize OK  
UnitTestGenTreeSearchAppendToNode OK  
UnitTestGenTree OK  
0,1,2,9,3,6,8,5,7,4,  
UnitTestGenTreeIterDepth OK  
0,9,1,2,3,4,6,8,5,7,  
UnitTestGenTreeIterBreadth OK  
0,1,2,3,4,5,6,7,8,9,  
UnitTestGenTreeIterValue OK  
UnitTestGenTreeIter OK  
UnitTestAll OK
```