General Tree Algebra

Contents

1 Overview

This algebra provides the `distdata`, `hpoint` and `hrect` type constructors and the following operators (`DATA` could be any type constructor).

- `getdistdata(_)`
  Creates `distdata` attributes from any type constructor for which a distdata type is defined in the `DistDataReg` class.

  Signature: `DATA -> distdata`
  Example: `getdistdata(5)`

- `getdistdata2(_, _)`
  Like above, but allows to select another than the default distdata type for the resp. type constructor.

  Signature: `DATA x <distdata-name> -> distdata`
  Example: `getdistdata2(2.5, default)`

- `gethpoint(_)`
  Creates `hpoint` attributes from any type constructor for which a gethpoint function is defined in the `HPointReg` class.

  Signature: `DATA -> hpoint`
  Example: `gethpoint(2.5)`

- `gethpoint2(_, _)`
  Like above, but allows to use another than the default gethpoint function for the resp. type constructor.

  Signature: `DATA x <gethpointfun-name> -> hpoint`
  Example: `gethpoint2(2.5, default)`

- `getbbox(_)`
  Creates `hrect` attributes from any type constructor for which a getbbox function is defined in the `BBoxReg` class.

  Signature: `DATA -> hrect`
  Example: `getbbox(2.5)`

- `getbbox2(_, _)`
  Like above, but allows to use another than the default getbbox function for the resp. type constructor.

  Signature: `DATA x <getbboxfun-name> -> hrect`
  Example: `getbbox2(2.5, default)`

- `gdistance(_, _)`
  Computes the distance between two attributes of the same type - a distance function for the resp. type constructor must be defined in the `DistfunReg` class.

  Signature: `DATA x DATA -> real`
  Example: `gdistance(2.5, 7.5)`

- `gdistance2(_, _, _)`
  Like above, but allows to select another than the default distance function for the resp. type constructor.

  Signature: `DATA x DATA x <distfun-name> -> real`
  Example: `gdistance2(2.5, 7.5, euclid)`

- `gdistance3(_, _, _, _)`
  Like above, but allows to select another than the default distance function and another than the default distdata type for the resp. type constructor.

  Signature: `DATA x <distfun-name> x <distdata-name> -> real`
  Example: `gdistance3(2.5, 7.5, euclid, native)`

1.1 Headerfile `GeneralTreeAlgebra.h`

January-May 2008, Mirko Dibbert

This file combines only some includes to provide a simple way to use the provided classes in other algebras (e.g. the MTreeAlgebra or XTreeAlgebra).

```
#ifndef __GENERAL_TREE_ALGEBRA_H__
#define __GENERAL_TREE_ALGEBRA_H__

#include "StandardTypes.h"
#include "GTA_SpatialAttr.h"
#include "HPointReg.h"
#include "BBoxReg.h"
#include "DistfunReg.h"
#include "GTA_TypeMapUtils.h" // contains some macros for the type
                              // mapping functions
#include "GTree.h"

using namespace std;

#endif // #ifndef __GENERAL_TREE_ALGEBRA_H__
```

1.2 Headerfile `GTA_Config.h`

January-May 2008, Mirko Dibbert

This file contains some constants and defines, which could be used to configurate the algebra.

```
#ifndef __GTA_CONFIG_H__
#define __GTA_CONFIG_H__

/////////////////////////////////////////////////////////////////////
// enable debug mode for the algebra
/////////////////////////////////////////////////////////////////////
// #define __GTA_DEBUG



namespace gta
{

/////////////////////////////////////////////////////////////////////
// domain type for the spatial types in the GeneralTreeAlgebra
// (should be float or double)
/////////////////////////////////////////////////////////////////////
typedef float GTA_SPATIAL_DOM;

} // namespace gta

#endif // #ifndef __GTA_CONFIG_H__
```

1.3 Headerfile `GTA_Spatial.h`

January-May 2008, Mirko Dibbert

This file contains the folllowing classes:

- `Spatial` (base for `HPoint` and `HRect`)
- `HPoint` (models hyper points)
- `HRect` (models hyper rects)
- `SpatialDistfuns` (distance functions for hpoint and hrect)

```
#ifndef __GTA_SPATIAL_H__
#define __GTA_SPATIAL_H__

#include "GTA_Config.h"
#include <assert.h>
#include <cstring>
#include <cmath>
#include <algorithm>

using namespace std;

namespace gta
{

// forward declarations:
class HRect;
```

1.3.1 Class *Spatial*

```
class Spatial
{
  public:
```

Default constructor (used from the 'read' constructors).

```
    inline Spatial()
    {}
```

Constructor.

```
    inline Spatial(unsigned dim)
        : m_dim(dim),
          m_vectorlen(m_dim * sizeof(GTA_SPATIAL_DOM))
    {}
```

Copy constructor.

```
    inline Spatial(const Spatial &e)
        : m_dim(e.dim()),
          m_vectorlen(e.vectorlen())
    {}
```

Destructor.

```
    inline ~Spatial()
    {}
```

Returns the dimension of the spatial object.

```
    inline unsigned dim() const
    { return m_dim; }
```

Returns the lengths of the coordinate vector arrays in bytes.

```
    inline unsigned vectorlen() const
    { return m_vectorlen; }

  protected:
```

Writes the spatial object to `buffer` and increases `offset`.

```
    void inline write(char *buffer, int &offset) const
    {
        memcpy(buffer+offset, &m_dim, sizeof(unsigned));
        offset += sizeof(unsigned);
    }
```

Reads the spatial object from `buffer` and increases `offset`.

```
        void inline read(const char *buffer, int &offset)
        {
            memcpy(&m_dim, buffer+offset, sizeof(unsigned));
            offset += sizeof(unsigned);

            m_vectorlen = m_dim * sizeof(GTA_SPATIAL_DOM);
        }
```

Returns the size of the spatial object on disc in bytes.

```
        inline size_t size() const
        { return sizeof(unsigned); }

        unsigned m_dim; // dimension of the spatal object
        unsigned m_vectorlen; // length of a coordinate vector in bytes
    }; // class Spatial
```

1.3.2 Class *HPoint*

```
    class HPoint
            : public Spatial
    {
      public:
```

Constructor.

```
        inline HPoint(unsigned dim, const GTA_SPATIAL_DOM *coords)
            : Spatial(dim),
              m_coords(new GTA_SPATIAL_DOM[dim])
        { memcpy(m_coords, coords, vectorlen()); }
```

Default Copy constructor.

```
        inline HPoint(const HPoint &e)
            : Spatial(e),
              m_coords(new GTA_SPATIAL_DOM[dim()])
        { memcpy(m_coords, e.m_coords, vectorlen()); }
```

Constructor (reads the given hyper point from buffer and increases offset)

```
        inline HPoint(const char *buffer, int &offset)
        { read(buffer, offset); }
```

Destructor.

```
        inline ˜HPoint()
        { delete m_coords; }
```

Assignment operator.

```
inline HPoint & operator=(const HPoint &rhs)
{
    m_dim = rhs.dim();
    m_vectorlen = m_dim * sizeof(GTA_SPATIAL_DOM);
    delete m_coords;
    m_coords = new GTA_SPATIAL_DOM[dim()];
    memcpy(m_coords, rhs.m_coords, vectorlen());
    return *this;
}
```

Returns the coordinate vector of the hyper point.

```
inline GTA_SPATIAL_DOM *coords() const
{ return m_coords; }
```

Returns the i-th coordinate of the hyper point.

```
inline GTA_SPATIAL_DOM coord(unsigned i) const
{
    #ifdef __GTA_DEBUG
    assert(i < dim());
    #endif

    return m_coords[i];
}
```

Returns the eucledean distance between `this` and `p`.

```
inline double eucl_dist(HPoint *p) const
{
    double result = 1.0;
    for (unsigned i = 0; i < dim(); ++i)
        result  += pow(coord(i) - p->coord(i), 2);
    result = sqrt(result);
    return result;
}
```

Returns the square of eucledean distance between `this` and `p`.

```
inline double eucl_dist2(HPoint *p) const
{
    double result = 1.0;
    for (unsigned i = 0; i < dim(); ++i)
        result  += pow(coord(i) - p->coord(i), 2);
    return result;
}
```

Writes the hyper point to `buffer` and increases `offset`.

```
void inline write(char *buffer, int &offset) const
{
    Spatial::write(buffer, offset);
```

```
            memcpy(buffer+offset, m_coords, vectorlen());
            offset += vectorlen();
        }
```

Reads the hyper point from `buffer` and increases `offset`.

```
        void inline read(const char *buffer, int &offset)
        {
            Spatial::read(buffer, offset);

            m_coords = new GTA_SPATIAL_DOM[dim()];
            memcpy(m_coords, buffer+offset, vectorlen());
            offset += vectorlen();
        }
```

Returns the size of the hyper point on disc in bytes.

```
        inline size_t size() const
        { return Spatial::size()  + vectorlen(); }
```

Returns the bounding box of `this`.

```
        HRect *bbox() const;

      private:
        GTA_SPATIAL_DOM *m_coords; // coordinate vector
    }; // class HPoint
```

1.3.3 Class *HRect*

```
    class HRect
            : public Spatial
    {
      public:
```

Constructor.

```
        HRect(unsigned dim, const GTA_SPATIAL_DOM *lb,
                            const GTA_SPATIAL_DOM *ub)
            : Spatial(dim),
              m_lbVect(new GTA_SPATIAL_DOM[dim]),
              m_ubVect(new GTA_SPATIAL_DOM[dim])
        {
            memcpy(m_lbVect, lb, vectorlen());
            memcpy(m_ubVect, ub, vectorlen());
        }
```

Default copy constructor.

```
inline HRect(const HRect &e)
    : Spatial(e),
      m_lbVect(new GTA_SPATIAL_DOM[dim()]),
      m_ubVect(new GTA_SPATIAL_DOM[dim()])
{
    memcpy(m_lbVect, e.m_lbVect, vectorlen());
    memcpy(m_ubVect, e.m_ubVect, vectorlen());
}
```

Constructor (reads the given hyper rectangle from buffer and increases offset)

```
inline HRect(const char *buffer, int &offset)
{ read(buffer, offset); }
```

Destructor.

```
inline ~HRect()
{ delete m_lbVect; delete m_ubVect; }
```

Assignment operator.

```
inline HRect &operator=(const HRect &rhs)
{
    m_dim = rhs.dim();
    m_vectorlen = m_dim * sizeof(GTA_SPATIAL_DOM);
    delete m_lbVect;
    delete m_ubVect;
    m_lbVect = new GTA_SPATIAL_DOM[dim()];
    m_ubVect = new GTA_SPATIAL_DOM[dim()];
    memcpy(m_lbVect, rhs.m_lbVect, vectorlen());
    memcpy(m_ubVect, rhs.m_ubVect, vectorlen());
    return *this;
}
```

Returns the lower bounds vector of the hyper rectangle.

```
inline const GTA_SPATIAL_DOM *lb() const
{ return m_lbVect; }
```

Returns the upper bounds vector of the hyper rectangle.

```
inline const GTA_SPATIAL_DOM *ub() const
{ return m_ubVect; }
```

Returns the i-th coordinate of the lower bounds vector of the hyper rectangle.

```
inline GTA_SPATIAL_DOM lb(unsigned i) const
{
    #ifdef __GTA_DEBUG
    assert(i < dim());
    #endif

    return m_lbVect[i];
}
```

Returns the i-th coordinate of the upper bounds vector of the hyper rectangle.

```
inline GTA_SPATIAL_DOM ub(unsigned i) const
{
    #ifdef __GTA_DEBUG
    assert(i < dim());
    #endif

    return m_ubVect[i];
}
```

Returns the i-th coordinate of the center vector of the hyper rectangle.

```
inline GTA_SPATIAL_DOM center(unsigned i) const
{
    #ifdef __GTA_DEBUG
    assert(i < dim());
    assert(m_ubVect[i] >= m_ubVect[i]);
    #endif

    return (m_ubVect[i] + m_lbVect[i]) / 2;
}
```

Returns the hpoint of which represents the center of the hyper rectangle.

```
inline HPoint center() const
{
    GTA_SPATIAL_DOM  c[dim()];
    for (unsigned i = 0; i < dim(); ++i)
        c[i] = center(i);
    return HPoint(dim(), c);
}
```

Returns the area of the hyper rectangle.

```
inline double area() const
{
    double result = 1.0;
    for (unsigned i = 0; i < dim(); ++i)
    {
        result *= (ub(i) - lb(i));
        if ((ub(i) - lb(i)) == 0.0)
            return 0.0;
    }
    return result;
}
```

Returns the margin of the hyper rectangle.

```
inline double margin() const
{
    double result = 0.0;
```

```
        for (unsigned i = 0; i < dim(); ++i)
            result += (ub(i) - lb(i));
        result *= pow(2, (double)(dim()-1));
        return result;
    }
```

Returns the partial margin of the hyper rectangle (each side is only added once instead of $2^(dim - 1)$ times - this method should be used, if the margin is only used to order a set of hyper rectangles, since it fulfills the same inequalities (e.g. r1.margin() ¡ r2.margin() iff. r1.margin2() ¡ r2.margin2()).

```
    inline double margin2() const
    {
        double result = 0.0;
        for (unsigned i = 0; i < dim(); ++i)
            result += (ub(i) - lb(i));
        return result;
    }
```

Returns the overlap between `this` and `r`.

```
    inline double overlap(HRect *r) const
    {
        if (!intersects(r))
            return 0.0;
        else
            return Intersection(r).area();
    }
```

Returns `true` if the current hyper rectangle intersects `r`.

```
    bool intersects(HRect *r) const
    {
        for (unsigned i = 0; i < dim(); ++i)
            if ((r->lb(i) > ub(i)) || (r->ub(i) < lb(i)))
                 return false;

        return true;
    }
```

Returns `true` if the current hyper rectangle contains `p`.

```
    bool contains(HPoint *p) const
    {
        for (unsigned i = 0; i < dim(); ++i)
            if ((p->coord(i) < lb(i)) || (p->coord(i) > ub(i)))
                 return false;

        return true;
    }
```

Returns `true` if the current hyper rectangle contains `r`.

```
        bool contains(HRect *r) const
        {
            for (unsigned i = 0; i < dim(); ++i)
                if ((r->lb(i) < lb(i)) || (r->ub(i) > ub(i)))
                     return false;

            return true;
        }
```

Returns `true` if the specified hyper rectangle is equal to the current one.

```
        inline bool operator== (const HRect &r) const
        {
            for(unsigned i = 0; i < dim(); ++i)
                if(lb(i) != r.lb(i) || ub(i) != r.ub(i) )
                    return false;

            return true;
    }
```

Returns `true` if the specified hyper rectangle is not equal to the current one.

```
        inline bool operator!= (const HRect &r) const
        { return !(*this == r); }
```

Returns the intersection between `this` and `r`.

```
        inline const HRect Intersection(HRect *r) const
        {
            GTA_SPATIAL_DOM lbVect[dim()], ubVect[dim()];
            for(unsigned i = 0; i < dim(); ++i)
            {
                lbVect[i] = max(lb(i), r->lb(i));
                ubVect[i] = min(ub(i), r->ub(i));
            }
            return HRect(dim(), lbVect, ubVect);
        }
```

Returns the union of `this` and `r`.

```
        inline const HRect Union(HRect *r) const
        {
            GTA_SPATIAL_DOM lbVect[dim()], ubVect[dim()];
            for(unsigned i = 0; i < dim(); ++i)
            {
                lbVect[i] = min(lb(i), r->lb(i));
                ubVect[i] = max(ub(i), r->ub(i));
            }
            return HRect(dim(), lbVect, ubVect);
        }
```

Replaces `this` with the union of `this` and `r`.

```
void unite(HRect *r)
{
    for (unsigned i=0; i < dim(); ++i)
    {
        m_lbVect[i] = min(lb(i), r->lb(i));
        m_ubVect[i] = max(ub(i), r->ub(i));
    }
}
```

Returns `true`, if `this` represents the empty set (could e.g. happen, if the hyper rectangle was created from the `Intersection` method for two non intersecting bounding boxes).

```
inline bool isEmptySet()
{
    for(unsigned i=0; i<dim(); ++i)
        if(lb(i) > ub(i))
            return true;

    return false;
}
```

Writes the hyper rectangle to `buffer` and increases `offset`.

```
void inline write(char *buffer, int &offset) const
{
    Spatial::write(buffer, offset);

    memcpy(buffer+offset, m_lbVect, vectorlen());
    offset += vectorlen();
    memcpy(buffer+offset, m_ubVect, vectorlen());
    offset += vectorlen();
}
```

Reads the hyper rectangle from `buffer` and increases `offset`.

```
void inline read(const char *buffer, int &offset)
{
    Spatial::read(buffer, offset);

    m_lbVect = new GTA_SPATIAL_DOM[dim()];
    memcpy(m_lbVect, buffer+offset, vectorlen());
    offset += vectorlen();

    m_ubVect = new GTA_SPATIAL_DOM[dim()];
    memcpy(m_ubVect, buffer+offset, vectorlen());
    offset += vectorlen();
}
```

Returns the size of the hyper point on disc in bytes.

```
inline size_t size() const
{ return Spatial::size()  + 2*vectorlen(); }
```

Returns the bounding box of `this`.

```
        inline HRect *bbox() const
        { return new HRect(*this); }

      private:
        GTA_SPATIAL_DOM *m_lbVect, *m_ubVect; // lower/upper bounds vect.
    }; // class HRect
```

1.3.4 Class *SpatialDistfuns*

```
    class SpatialDistfuns
    {
      public:
```

Returns the Euclidean distance between `p1` and `p2`.

```
        static double euclDist(HPoint *p1, HPoint *p2);
```

Returns the square of the Euclidean distance between `p1` and `p2` (avoids computing the square root of result).

```
        static double euclDist2(HPoint *p1, HPoint *p2);
```

Returns 0 if `r` contains `p` and the square of the Euclidean distance between `p` and the edge of `r` otherwhise.

```
        static double minDist(HPoint *p, HRect *r);
```

This method computes the square of the Euclidean distance between `p` and the farthest point of each edge of `r` and returns the minimum of all theese distances (needed for the RKV nearast-neighbour search algorithm, e.g. in the XTreeAlgebra).

```
        static double minMaxDist(HPoint *p, HRect *r);
    }; // class SpatialDistfuns



    } // namespace gta
    #endif // #ifndef __GTA_SPATIAL_H__
```

1.4 Headerfile `GTA_SpatialAttr.h`

January-May 2008, Mirko Dibbert

This file implements the `hpoint` (hyper point) and `hrect` (hyper rectangle) type constructors. The methods of the resp. `HPoint` and `HRect` class could be accessed with help of the `hpoint` or `hrect` method.

```
#ifndef __GTA_SPATIAL_ATTR_H__
#define __GTA_SPATIAL_ATTR_H__

#include "StandardAttribute.h"
#include "GTA_Spatial.h"

namespace gta {
```

1.4.1 Class *HPointAttr*

```
class HPointAttr
        : public StandardAttribute
{
  public:
```

Default constructor.

```
    inline HPointAttr()
    {}
```

Constructor (creates an undefined hyper point - the parameter is only needed to distinguish this constructor from the default constructor)

```
inline HPointAttr(size_t size)
    : m_coords(0), m_defined(false)
{}
```

Constructor (initialises the hyper point with the given values)

```
inline HPointAttr(int dim, GTA_SPATIAL_DOM *coords)
    : m_dim(dim),
      m_coords(vectorlen()),
      m_defined(true)
{ m_coords.Put(0, vectorlen(), coords); }
```

Default copy constructor.

```
HPointAttr(const HPointAttr &p)
    : m_dim(p.m_dim),
      m_coords(p.m_coords.Size()),
      m_defined(p.m_defined)
{
    if(IsDefined())
    {
        const char *buffer;
        p.m_coords.Get(0, &buffer);
        m_coords.Put(0, p.m_coords.Size(), buffer);
    }
}
```

Destructor.

```
inline ~HPointAttr()
{}
```

Sets the attribute values to the given values.

```
void set(bool defined, unsigned dim, GTA_SPATIAL_DOM *coords)
{
    m_defined = defined;
    if(m_defined)
    {
        m_dim = dim;
        m_coords.Clean();
        m_coords.Resize(vectorlen());
        m_coords.Put(0, m_coords.Size(), coords);
    }
}
```

Sets the attribute values to the given values.

```
void set(bool defined, HPoint *p)
{
    m_defined = defined;
    if(m_defined)
    {
        m_dim = p->dim();
        m_coords.Clean();
        m_coords.Resize(vectorlen());
        m_coords.Put(0, m_coords.Size(), p->coords());
    }
}
```

Returns the dimension of the hyper point.

```
inline unsigned dim() const
{ return m_dim; }
```

Removes the disc representation of the coordinate vector FLOB.

```
inline void deleteFLOB()
{ m_coords.Destroy(); }
```

Returns a new `HPoint` object which represents `this`.

```
inline HPoint *hpoint() const
{
    const char *buffer;
    m_coords.Get(0, &buffer);
    const GTA_SPATIAL_DOM *coords =
            reinterpret_cast<const GTA_SPATIAL_DOM*>(buffer);
    return new HPoint(m_dim, coords);
}
```

Implementation of virtual methods from the StandardAttribute class:

```
inline virtual bool IsDefined() const
{ return m_defined; }

inline virtual void SetDefined(bool defined)
{ m_defined = defined; }

inline virtual size_t Sizeof() const
{ return sizeof(*this); }

inline virtual bool Adjacent(const Attribute *attr) const
{ return false; }

inline virtual Attribute *Clone() const
{ return new HPointAttr(*this); }

inline virtual int NumOfFLOBs() const
{ return 1; }
```

```
        inline virtual FLOB *GetFLOB(const int i)
        { return &m_coords; }

        inline virtual int Compare(const Attribute *rhs) const
        { return 0; }

        virtual size_t HashValue() const
        { return 0; }

        virtual void CopyFrom(const StandardAttribute *rhs)
        {
            const HPointAttr *p = static_cast<const HPointAttr*>(rhs);

            m_defined = p->IsDefined();
            if(IsDefined())
            {
                m_dim = p->m_dim;
                const char *buffer;

                m_coords.Clean();
                m_coords.Resize(vectorlen());
                p->m_coords.Get(0, &buffer);
                m_coords.Put(0, p->m_coords.Size(), buffer);
            }
        }

    private:
```

Returns the size of the coordinate vector in bytes.

```
        inline unsigned vectorlen() const
        { return m_dim * sizeof(GTA_SPATIAL_DOM); }

        int m_dim;      // dimension of the hyper point
        FLOB m_coords;  // coordinate vector
        bool m_defined; // true, if the attribute is defined

    }; // class HPointAttr
```

1.4.2 Class *HRectAttr*

```
    class HRectAttr
            : public StandardAttribute
    {
      public:
```

Default constructor.

```
        inline HRectAttr()
        {}
```

Constructor (creates an undefined hyper rectangle - the parameter is only needed to distinguish this constructor from the default constructor).

```
inline HRectAttr(size_t size)
    : m_lbVect(0), m_ubVect(0), m_defined(false)
{}
```

Constructor (initialises the hyper rectangle with the given values)

```
inline HRectAttr(int dim, GTA_SPATIAL_DOM *lb,
                          GTA_SPATIAL_DOM *ub)
    : m_dim(dim),
      m_lbVect(vectorlen()),
      m_ubVect(vectorlen()),
      m_defined(true)
{
    m_lbVect.Put(0, vectorlen(), lb);
    m_ubVect.Put(0, vectorlen(), ub);
}
```

Default copy constructor.

```
HRectAttr(const HRectAttr &r)
    : m_dim(r.m_dim),
      m_lbVect(r.m_lbVect.Size()),
      m_ubVect(r.m_ubVect.Size()),
      m_defined(r.m_defined)
{
    if(IsDefined())
    {
        const char *buffer;

        r.m_lbVect.Get(0, &buffer);
        m_lbVect.Put(0, r.m_lbVect.Size(), buffer);

        r.m_ubVect.Get(0, &buffer);
        m_ubVect.Put(0, r.m_ubVect.Size(), buffer);
    }
}
```

Destructor.

```
inline ~HRectAttr()
{}
```

Sets the attribute values to the given values.

```
void set(
        bool defined, unsigned dim,
        GTA_SPATIAL_DOM *lb, GTA_SPATIAL_DOM *ub)
{
    m_defined = defined;
```

```
        if(m_defined)
        {
            m_dim = dim;

            m_lbVect.Clean();
            m_lbVect.Resize(vectorlen());
            m_lbVect.Put(0, m_lbVect.Size(), lb);

            m_ubVect.Clean();
            m_ubVect.Resize(vectorlen());
            m_ubVect.Put(0, m_ubVect.Size(), ub);
        }
    }
```

Sets the attribute values to the given values.

```
    void set(bool defined, HRect *r)
    {
        m_defined = defined;
        if(m_defined)
        {
            m_dim = r->dim();

            // set  lower bounds vector
            m_lbVect.Clean();
            m_lbVect.Resize(vectorlen());
            m_lbVect.Put(0, m_lbVect.Size(), r->lb());

            // set upper bounds vector
            m_ubVect.Clean();
            m_ubVect.Resize(vectorlen());
            m_ubVect.Put(0, m_ubVect.Size(), r->ub());
        }
    }
```

Returns the dimension of the hyper rectangle.

```
    inline unsigned dim() const
    { return m_dim; }
```

Removes the disc representation of the FLOBs.

```
    inline void deleteFLOB()
    { m_lbVect.Destroy(); m_ubVect.Destroy(); }
```

Returns a new `HRect` object which represents `this`.

```
    inline HRect *hrect() const
    {
        const char *buffer;
        m_lbVect.Get(0, &buffer);
        const GTA_SPATIAL_DOM *lb =
                reinterpret_cast<const GTA_SPATIAL_DOM*>(buffer);
        m_ubVect.Get(0, &buffer);
        const GTA_SPATIAL_DOM *ub =
                reinterpret_cast<const GTA_SPATIAL_DOM*>(buffer);
        return new HRect(m_dim, lb, ub);
    }
```

Implementation of virtual methods from the StandardAttribute class:

```
    inline virtual bool IsDefined() const
    { return m_defined; }

    inline virtual void SetDefined(bool defined)
    { m_defined = defined; }

    inline virtual size_t Sizeof() const
    { return sizeof(*this); }

    inline virtual bool Adjacent(const Attribute *attr) const
    { return false; }

    inline virtual Attribute *Clone() const
    { return new HRectAttr(*this); }

    inline virtual int NumOfFLOBs() const
    { return 2; }

    inline virtual FLOB* GetFLOB(const int i)
    {
        if (i == 1)
            return &m_lbVect;
        else
            return &m_ubVect;
    }

    inline virtual int Compare(const Attribute *rhs) const
    { return 0; }

    virtual size_t HashValue() const
    { return 0; }

    virtual void CopyFrom(const StandardAttribute *rhs)
    {
        const HRectAttr *r = static_cast<const HRectAttr*>(rhs);

        m_defined = r->IsDefined();
        if(IsDefined())
        {
            m_dim = r->m_dim;
            const char *buffer;

            // copy lower bounds vector
            m_lbVect.Clean();
            m_lbVect.Resize(vectorlen());
            r->m_lbVect.Get(0, &buffer);
            m_lbVect.Put(0, r->m_lbVect.Size(), buffer);

            // copy upper bounds vector
            m_ubVect.Clean();
            m_ubVect.Resize(vectorlen());
            r->m_ubVect.Get(0, &buffer);
            m_ubVect.Put(0, r->m_ubVect.Size(), buffer);
        }
    }

private:
```

Returns the size of the coordinate vector in bytes.

```
    inline unsigned vectorlen() const
    { return m_dim * sizeof(GTA_SPATIAL_DOM); }

    int m_dim;      // dimension of the hyper rectangle
    FLOB m_lbVect;  // lower bounds vector
    FLOB m_ubVect;  // upper bounds vector
    bool m_defined; // true, if the attribute is defined
}; // class HRectAttr


} // namespace gta
#endif // #ifndef __SPATIAL_ATTR_H__
```

1.5 Headerfile `HPointReg.h`

January-May 2008, Mirko Dibbert

This file contains all defined gethpoint functions. New functions must be registered in the `HPointReg::initialize` method.

1.5.1 Includes and defines

```
#ifndef __HPOINT_REG_H__
#define __HPOINT_REG_H__

#include "SecondoInterface.h"
#include "Symbols.h"
#include "GTA_SpatialAttr.h"

extern SecondoInterface* si;
using symbols::Sym;

namespace gta
{

typedef HPoint* (*GetHPointFun)(const void *data);
```

Refers the default gethpoint function for the each type constructor.

```
Sym HPOINT_DEFAULT("default");
```

This value is returned from `HPointReg::defaultName()`, if no default gethpoint function has been found for the resp. type constructor.

```
Sym HPOINT_UNDEFINED("n/a");

//////////////////////////////////////////////////////////////////////
// constants for the gethpoint function names:
//////////////////////////////////////////////////////////////////////

// native gethpoint function (e.g. used if only one fun is defined)
Sym HPOINT_NATIVE("native");



//////////////////////////////////////////////////////////////////////
// Flags for the HPointInfo class:
//////////////////////////////////////////////////////////////////////

// this flag is set for all defined info objects
// (will automatically assigned from the constructor)
const char HPOINT_IS_DEFINED = (1 << 0);

// if set, the info object will be used as default for the resp. type
// constructor (if more than one info object for the same type
// constructor is specified as default, the least added one will be
// used)
const char HPOINT_IS_DEFAULT = (1 << 1);
```

1.5.2 Class *HPointInfo*

This class stores a pointer to a gethpoint function together with the function name.

```
class HPointInfo
{

public:
```

Default constructor (creates an undefined info object)

```
    inline HPointInfo()
    : m_name("undef"), m_gethpointFun(0), m_flags(0)
    {}
```

Constructor (creates a new info object with the given values)

```
    inline HPointInfo(
            const string &name,
            const string &typeName,
            const GetHPointFun gethpoint_Fun,
            char flags = 0)
        : m_name(name), m_gethpointFun(gethpoint_Fun),
          m_flags(HPOINT_IS_DEFINED | flags)
    { si->GetTypeId(typeName, m_algebraId, m_typeId); }
```

Returns the name of the assigned gethpoint function.

```
        inline string name() const
        { return m_name; }
```

Returns the assigned gethpoint function.

```
        inline GetHPointFun getHPointFun() const
        { return m_gethpointFun; }
```

Generates a new `HPoint` object of with the assigned gethpoint function from `attr`.

```
        inline HPoint* getHPoint(const void *attr) const
        { return m_gethpointFun(attr); }
```

Returns the id of the assigned type constructor.

```
        inline int typeId() const
        { return m_typeId; }
```

Returns the name of the assigned type constructor.

```
        inline string typeName() const
        { return am->Constrs(m_algebraId, m_typeId); }
```

Returns the id of the assigned algebra.

```
        inline int algebraId() const
        { return m_algebraId; }
```

Returns the name of the assigned algebra.

```
        inline string algebraName() const
        { return am->GetAlgebraName(m_algebraId); }
```

Returns `true`, if the `HPointInfo` object is defined (should only be `false`, if a requested gethpoint function could not be found in the `HPointReg` class).

```
        inline bool isDefined() const
        { return (m_flags & HPOINT_IS_DEFINED); }
```

Returns `true`, if the info object is the default info object for the respective type constructor.

```
        inline bool isDefault() const
        { return (m_flags & HPOINT_IS_DEFAULT); }

    private:
        string m_name;   // unique name (used for selection)
        int m_algebraId; // algebra-id of the assigned type constructor
        int m_typeId;    // type-id of the assigned type constructor
        GetHPointFun m_gethpointFun; // assigned gethpoint function
        char m_flags;
    }; // class HPointInfo
```

1.5.3 Class *HPointReg*

This class manages all defined gethpoint functions.

```
class HPointReg
{

public:
```

Initializes the defined gethpoint functions.

```
    static void initialize();
```

Returns `true`, if the `initialize` function has already been called.

```
    static inline bool isInitialized()
    { return initialized; }
```

Adds a new `HPointInfo` object.

```
    static void addInfo(HPointInfo info);
```

Returns the name of the default gethpoint function for the specified type.

```
    static string defaultName(const string &typeName);
```

Returns the specified `HPointInfo` object.

```
    static HPointInfo &getInfo(
            const string &typeName, const string &gethpoint_FunName);
```

Returns `true`, if the specified `HPointInfo` object does exist.

```
    static inline bool isDefined(
            const string &typeName, const string &gethpoint_FunName)
    { return getInfo(typeName, gethpoint_FunName).isDefined(); }
```

Returns a string with a list of all defined gethpoint functions for the specified type (could e.g. be used in the type mapping functions in case of errors to show possible values).

```
    static string definedNames(const string &typeName);

private:
    typedef map<string, HPointInfo>::iterator hpointIter;

    static bool initialized;
    static HPointInfo defaultInfo;
    static map<string, HPointInfo> hpointInfos;
    static map<string, string> defaultNames;
```

Defined gethpoint functions:

Gethpoint function for the `real` type constructor.

```
        static HPoint *gethpoint_Int(const void *attr);
```

Gethpoint function for the `real` type constructor.

```
        static HPoint *gethpoint_Real(const void *attr);
```

Gethpoint function for the `point` type constructor.

```
        static HPoint *gethpoint_Point(const void *attr);
```

Gethpoint function for the `hpoint` type constructor.

```
        static HPoint *gethpoint_HPoint(const void *attr);
    }; // class HPointReg


    } // namespace gta
    #endif // #ifndef __HPOINT_REG_H__
```

1.6 Headerfile `BBoxReg.h`

January-May 2008, Mirko Dibbert

This file contains all defined getbbox functions. New functions must be registered in the `BBoxReg::initialize` method.

1.6.1 Includes and defines

```
#ifndef __BBOX_REG_H__
#define __BBOX_REG_H__

#include "SecondoInterface.h"
#include "Symbols.h"
#include "RectangleAlgebra.h"
#include "GTA_SpatialAttr.h"

extern SecondoInterface* si;
using symbols::Sym;

namespace gta
{

typedef HRect* (*GetBBoxFun)(const void *data);
```

Refers the default getbbox function for the each type constructor.

```
Sym BBOX_DEFAULT("default");
```

This value is returned from `BBoxReg::defaultName()`, if no default getbbox function has been found for the resp. type constructor.

```
Sym BBOX_UNDEFINED("n/a");

//////////////////////////////////////////////////////////////////////
// constants for the getbbox function names:
//////////////////////////////////////////////////////////////////////

// native getbbox function (e.g. used if only one fun is defined)
Sym BBOX_NATIVE("native");



//////////////////////////////////////////////////////////////////////
// Flags for the BBoxInfo class:
//////////////////////////////////////////////////////////////////////

// this flag is set for all defined info objects
// (will automatically assigned from the constructor)
const char BBOX_IS_DEFINED = (1 << 0);

// if set, the info object will be used as default for the resp. type
// constructor (if more than one info object for the same type
// constructor is specified as default, the least added one will be
// used)
const char BBOX_IS_DEFAULT = (1 << 1);
```

1.6.2 Class *BBoxInfo*

This class stores a pointer to a getbbox function together with the function name.

```
class BBoxInfo
{

public:
```

Default constructor (creates an undefined info object)

```
    inline BBoxInfo()
    : m_name("undef"), m_getbboxFun(0), m_flags(0)
    {}
```

Constructor (creates a new info object with the given values)

```
    inline BBoxInfo(
            const string &name,
            const string &typeName,
            const GetBBoxFun getbbox_Fun,
            char flags = 0)
        : m_name(name), m_getbboxFun(getbbox_Fun),
          m_flags(BBOX_IS_DEFINED | flags)
    { si->GetTypeId(typeName, m_algebraId, m_typeId); }
```

Returns the name of the assigned getbbox function.

```
        inline string name() const
        { return m_name; }
```

Returns the assigned getbbox function.

```
        inline GetBBoxFun getBBoxFun() const
        { return m_getbboxFun; }
```

Generates a new `HRect` object with the assigned getbbox function from `attr`.

```
        inline HRect* getBBox(const void *attr) const
        { return m_getbboxFun(attr); }
```

Returns the id of the assigned type constructor.

```
        inline int typeId() const
        { return m_typeId; }
```

Returns the name of the assigned type constructor.

```
        inline string typeName() const
        { return am->Constrs(m_algebraId, m_typeId); }
```

Returns the id of the assigned algebra.

```
        inline int algebraId() const
        { return m_algebraId; }
```

Returns the name of the assigned algebra.

```
        inline string algebraName() const
        { return am->GetAlgebraName(m_algebraId); }
```

Returns `true`, if the `BBoxInfo` object is defined (should only be `false`, if a requested getbbox function could not be found in the `BBoxReg` class).

```
        inline bool isDefined() const
        { return (m_flags & BBOX_IS_DEFINED); }
```

Returns `true`, if the info object is the default info object for the respective type constructor.

```
        inline bool isDefault() const
        { return (m_flags & BBOX_IS_DEFAULT); }

    private:
        string m_name;   // unique name (used for selection)
        int m_algebraId; // algebra-id of the assigned type constructor
        int m_typeId;    // type-id of the assigned type constructor
        GetBBoxFun m_getbboxFun; // assigned getbbox_ function
        char m_flags;
    }; // class BBoxInfo
```

1.6.3 Class *BBoxReg*

This class manages all defined getbbox functions.

```
class BBoxReg
{

public:
```

Initializes the defined getbbox functions.

```
    static void initialize();
```

Returns `true`, if the `initialize` function has already been called.

```
    static inline bool isInitialized()
    { return initialized; }
```

Adds a new `BBoxInfo` object.

```
    static void addInfo(BBoxInfo info);
```

Returns the name of the default getbbox function for the specified type.

```
    static string defaultName(const string &typeName);
```

Returns the specified `BBoxInfo` object.

```
    static BBoxInfo &getInfo(
            const string &typeName, const string &getbbox_FunName);
```

Returns `true`, if the specified `BBoxInfo` object does exist.

```
    static inline bool isDefined(
            const string &typeName, const string &getbbox_FunName)
    { return getInfo(typeName, getbbox_FunName).isDefined(); }
```

Returns a string with a list of all defined getbbox functions for the specified type (could e.g. be used in the type mapping functions in case of errors to show possible values).

```
    static string definedNames(const string &typeName);

private:
    typedef map<string, BBoxInfo>::iterator bboxIter;

    static bool initialized;
    static BBoxInfo defaultInfo;
    static map<string, BBoxInfo> bboxInfos;
    static map<string, string> defaultNames;
```

Defined getbbox functions:

Getbbox function for the `real` type constructor.

```
        static HRect *getbbox_Int(const void *attr);
```

Getbbox function for the `real` type constructor.

```
        static HRect *getbbox_Real(const void *attr);
```

Getbbox function for spatial attributes.
Need to define seperate functions for each dimension duo to compiler errors under windows:

```
Internal compiler error in c_expand_expr, at c-common.c:3714
```

```
        static HRect *getbbox_Spatial2(const void *attr);
        static HRect *getbbox_Spatial3(const void *attr);
        static HRect *getbbox_Spatial4(const void *attr);
        static HRect *getbbox_Spatial8(const void *attr);
```

Getbbox function for the `hpoint` type constructor.

```
        static HRect *getbbox_HPoint(const void *attr);
```

Getbbox function for the `bbox` type constructor.

```
        static HRect *getbbox_HRect(const void *attr);
    }; // class BBoxReg

    } // namespace gta
    #endif // #ifndef __BBOX_REG_H__
```

1.7 Headerfile `DistDataReg.h`

January-May 2008, Mirko Dibbert

1.7.1 Overview

This headerfile declares the following classes:

- `DistDataId`
  Objects of this class store the id of a distdata-type together with the id of a type constructor (type-constructor id + algebra id).

- `DistData`
  This class is used as parameter type for the distance functions, which are defined in the `DistfunReg` class (headerfile `Distfunreg.h`). Objects of this class contain a char array, which stores all neccesary data for distance computations (usually the memory representation of a value (or an array of values), but it could also contain any other data, e.g. the name of a file, from which the distance function should read the data).

- `DistDataAttribute`
  This class is similar to the `DistData` class, but it implemets the `StandardAttribute` class and the data is stored within a `FLOB` instead of a char array. Additionally it stores a `DistDataId` object, which is needed to select the assigned `DistDataInfo` object from the `DistDataReg` class and to test, if `DistDataAttr` objects belong to the same distdata-type.

- `DistDataInfo`
  Objects of this class store a function pointer to a getdata-function, which transforms `Attribute`

objects into DistData objects. Each type constructor could provide several distdata-types, which are distinguished by the name of the assigned type-constructor and the name of the distdata-type (used e.g. for the picture type constructor to create several histogram-types from pictures).

- DistDataReg
  This class managages all defined DistDataInfo objects. Each DistDataInfo object could be unambigously selected by the name of the distdata-type and the name of the type-constructor or by a DistDataId object.

New distdata-types must be registered in the DistDataReg::initialize method.

1.7.2 Includes and defines

```
#ifndef __DISTDATA_REG_H__
#define __DISTDATA_REG_H__

#include "SecondoInterface.h"
#include <iostream>
#include <string>
#include "StandardTypes.h"
#include "StandardAttribute.h"
#include "Symbols.h"

extern SecondoInterface* si;
using namespace std;

namespace gta
{

class DistData; // forward declaration
```

Refers the default distdata-type for the each type constructor.

```
symbols::Sym DDATA_DEFAULT("default");
```

This value is returned from DistDataReg::defaultName(), if no default distdata-type has been found for the resp. type constructor.

```
symbols::Sym DDATA_UNDEFINED("n/a");

//////////////////////////////////////////////////////////////////////
// Name and short descriprions for the defined distdata-types:
//////////////////////////////////////////////////////////////////////

// native data representation, e.g. an int value for CcInt attributes
// (this type usually used, if only one distdata-type is defined for
// the respective type constructor).
symbols::Sym DDATA_NATIVE("native");
symbols::Sym DDATA_NATIVE_DESCR(
        "native representation of the respective datatype");
```

```
//////////////////////////////////////////////////////////////////////
// Constants for the distdata-type id's
//
// For each distdata-type name above one unique integer constant
// should be defined below. These constants are stored within
// "DistDataAttribute"[4] objects instead of the whole name to save
// some memory (by the way this allows to change the name of the
// distdata-types without affecting the "DistDataAttribute"[4]
// objects, as long as the assigned id will not be changed).
//////////////////////////////////////////////////////////////////////
const int DDATA_NATIVE_ID = 0;

// id's for the picture algebra types (defined in PictureFuns.h)
const int DDATA_HSV64_ID         = 1000;
const int DDATA_HSV128_ID        = 1001;
const int DDATA_HSV256_ID        = 1002;
const int DDATA_LAB256_ID        = 1011;
const int DDATA_HSV64_NCOMPR_ID  = 1020;
const int DDATA_HSV128_NCOMPR_ID = 1021;
const int DDATA_HSV256_NCOMPR_ID = 1022;
const int DDATA_LAB256_NCOMPR_ID = 1031;

typedef DistData* (*GetDataFun)(const void* attr);

//////////////////////////////////////////////////////////////////////
// Flags for the DistDataInfo class
//////////////////////////////////////////////////////////////////////

// this flag is set for all defined info objects
// (automatically assigned from the constructor)
const char DDATA_IS_DEFINED = (1 << 0);
```

1.7.3 Class *DistDataId*

```
class DistDataId
{

public:
```

Default constructor (should not be used).

```
    inline DistDataId()
    {}
```

Constructor (creates an undefined id - the `defined` parameter is only needed to distinguish this constructor from the default constructor).

```
    inline DistDataId(bool defined)
        : m_algebraId(-1)
    {}
```

Constructor.

```
        inline DistDataId(int algebraId, int typeId, int distdataId)
        : m_algebraId(algebraId), m_typeId(typeId),
          m_distdataId(distdataId)
        {}
```

Returns `true`, if the id's are equal.

```
        bool inline operator == (const DistDataId& rhs) const
        {
            return (m_algebraId == rhs.m_algebraId) &&
                   (m_typeId == rhs.m_typeId) &&
                   (m_distdataId == rhs.m_distdataId);
        }
```

Returns `true`, if the id's are not equal.

```
        bool inline operator != (const DistDataId& rhs) const
        { return !operator == (rhs); }
```

Returns `true`, if the assigned distdata type is defined (in this case `m_algebraId` must be positive).

```
        inline bool isDefined() const
        { return (m_algebraId >= 0); }
```

Returns the id of the assigned type constructor.

```
        inline int typeId() const
        { return m_typeId; }
```

Returns the name of the assigned type constructor.

```
        inline string typeName() const
        { return am->Constrs(m_algebraId, m_typeId); }
```

Returns the id of the assigned algebra.

```
        inline int algebraId() const
        { return m_algebraId; }
```

Returns the name of the assigned algebra.

```
        inline string algebraName() const
        { return am->GetAlgebraName(m_algebraId); }
```

Returns the id of the assigned distdata type.

```
        inline int distdataId() const
        { return m_distdataId; }

    private:
        int m_algebraId;  // algebra-id of the assigned type constructor
        int m_typeId;     // type-id of the assigned type constructor
        int m_distdataId; // id of the distdata type
    }; // class DistDataId
```

1.7.4 Class *DistDataId*

```
class DistData
{

public:
```

Constructor (initiates the object with a copy of the given `char` array).

```
    inline DistData(size_t size, const void* value)
        : m_size(size), m_value(new char[m_size])
    { memcpy(m_value, value, m_size); }
```

Constructor (initiates the object with the given string).

```
    inline DistData(const string value)
        : m_size(value.size()+1), m_value(new char[m_size])
    { memcpy(m_value, value.c_str(), m_size); }
```

Constructor (reads the object from `buffer` and increases `offset`).

```
    DistData(const char *buffer, int& offset)
    {
        // read m_size
        memcpy(&m_size, buffer+offset, sizeof(size_t));
        offset += sizeof(size_t);

        // read m_value
        m_value = new char[m_size];
        memcpy(m_value, buffer+offset, m_size);
        offset += m_size;
    }
```

Default copy constructor.

```
    inline DistData(const DistData& e)
    : m_size (e.m_size), m_value(new char[e.m_size])
    { memcpy(m_value, e.m_value, e.m_size); }
```

Destructor.

```
    inline ~DistData()
    { delete m_value; }
```

Returns a copy of `this`.

```
    inline DistData *clone()
    { return new DistData(*this); }
```

Returns a reference to the stored data array.

```
    inline const void *value() const
    { return m_value; }
```

Returns the size of the data array

```
        inline size_t size() const
        { return m_size; }
```

Writes the object to `buffer` and increases `offset`.

```
        void write(char *buffer, int& offset) const
        {
            // write m_size
            memcpy(buffer+offset, &m_size, sizeof(size_t));
            offset += sizeof(size_t);

            // write m_value
            memcpy(buffer+offset, m_value, m_size);
            offset += m_size;
        }

    private:
        size_t m_size;  // length of the data array
        char * m_value; // the data array
    };
```

1.7.5 Class *DistDataAttribute*

```
    class DistDataAttribute : public StandardAttribute
    {

    public:
```

Default constructor (should not be used, except for the cast method).

```
        inline DistDataAttribute()
        {}
```

Constructor (creates an undefined distdata object)

```
        inline DistDataAttribute(size_t size)
        : m_data(0), m_defined(false)
        {}
```

Default copy constructor.

```
        DistDataAttribute(const DistDataAttribute& ddAttr)
        : m_data(ddAttr.m_data.Size()),
          m_defined(ddAttr.m_defined),
          m_distdataId(ddAttr.m_distdataId)
        {
            if(IsDefined())
                m_data.Put(0, ddAttr.size(), ddAttr.value());
        }
```

Destructor.

```
        inline ~DistDataAttribute()
        {}
```

Returns a copy of the object.

```
        inline DistDataAttribute *clone() const
        { return new DistDataAttribute(*this); }
```

Sets the attribute values to the given values.

```
        void set(
                bool defined, const char *data, size_t size,
                DistDataId distdataId);
```

Sets the attribute values to the given values.

```
        void set(bool defined, DistData *data, DistDataId distdataId);
```

Returns id of the assigned distance function.

```
        inline DistDataId distdataId() const
        { return m_distdataId; }
```

Returns a reference to the data representation.

```
        inline const char *value() const
        {
            const char *data;
            m_data.Get(0, &data);
            return data;
        }
```

Returns the size of the data object in bytes.

```
        inline size_t size() const
        { return m_data.Size(); }
```

Removes the disc representation of the data FLOB.

```
        inline void deleteFLOB()
        { m_data.Destroy(); }

    //////////////////////////////////////////////////////////////////////
    // virtual methods from the StandardAttribute class:
    //////////////////////////////////////////////////////////////////////
        inline virtual bool IsDefined() const
        { return m_defined; }

        inline virtual void SetDefined(bool defined)
        { m_defined = defined; }
        inline virtual size_t Sizeof() const
        { return sizeof(*this); }

        inline virtual bool Adjacent(const Attribute *attr) const
        { return false; }

        inline virtual Attribute *Clone() const
        { return clone(); }

        inline virtual int NumOfFLOBs() const
        { return 1; }

        inline virtual FLOB *GetFLOB(const int i)
        { return &m_data; }

        inline virtual int Compare(const Attribute *rhs) const
        { return 0; }

        inline virtual size_t HashValue() const
        { return 0; }

        virtual void CopyFrom(const StandardAttribute *rhs);

    private:
        FLOB m_data; // contains the data array
        bool m_defined; // true, if the attribute is defined
        DistDataId m_distdataId;
    }; // class DistDataAttribute
```

1.7.6 Class *DistDataInfo*

```
    class DistDataInfo
    {

    public:
```

Default constructor (creates an undefined info object)

```
        DistDataInfo();
```

Constructor (creates a new info object with the given values)

```
        DistDataInfo(const string& name, const string& descr,
                     int distdataId, const string& typeName,
                     const GetDataFun getDataFun, char flags = 0);
```

Returns the name of the distdata-type.

```
        inline string name() const
        { return m_name; }
```

Returns the description of the distdata-type.

```
        inline string descr() const
        { return m_descr; }
```

Returns the assigned getdata function.

```
        inline GetDataFun getDataFun() const
        { return m_getDataFun; }
```

Generates a new `DistData` object of the assigned type from `attr`.

```
        inline DistData *getData(const void *attr)
        { return m_getDataFun(attr); }
```

Returns the id of the `DistDataInfo` object.

```
        inline DistDataId id() const
        { return m_id; }
```

Returns the id of the assigned type constructor.

```
        inline int typeId() const
        { return m_id.typeId(); }
```

Returns the name of the assigned type constructor.

```
        inline string typeName() const
        { return m_id.typeName(); }
```

Returns the id of the assigned algebra.

```
        inline int algebraId() const
        { return m_id.algebraId(); }
```

Returns the name of the assigned algebra.

```
        inline string algebraName() const
        { return m_id.algebraName(); }
```

Returns the id of the distdata-type.

```
        inline int distdataId() const
        { return m_id.distdataId(); }
```

Returns `true`, if the `DistDataInfo` object is defined (should only be `false`, if a requested distdata-type could not be found in the `DistDataReg` class).

```
        inline bool isDefined() const
        { return (m_flags & DDATA_IS_DEFINED); }

    private:
        string m_name;  // unique name (used for selection)
        string m_descr; // short description
        GetDataFun m_getDataFun; // assigned getdata function
        DistDataId m_id;
        char m_flags;
    }; // class DistDataInfo
```

1.7.7 Class *DistDataReg*

```
class DistDataReg
{

friend class DistfunReg;

public:
```

Initializes the defined distdata types.

```
    static void initialize();
```

Returns `true`, if the `initialize` function has already been called.

```
    static inline bool isInitialized()
    { return initialized; }
```

Adds a new `DistDataInfo` object.

```
    static void addInfo(DistDataInfo info);
```

Returns the name of the default distdata type for the specified type.

```
    static string defaultName(const string& typeName);
```

Returns the `DistDataId` of the specified `DistDataInfo` object.

```
    static DistDataId getId(
            const string& typeName, const string& distdataName);
```

Returns the `DistDataInfo` object for the specified id.

```
    static DistDataInfo& getInfo(DistDataId id);
```

Returns the specified `DistDataInfo` object.

```
    static inline DistDataInfo& getInfo(
            const string& typeName, const string& distdataName)
    { return getInfo(getId(typeName, distdataName)); }
```

Returns `true`, if the specified `DistDataInfo` object does exist.

```
    static inline bool isDefined(
            const string& typeName, const string& distdataName)
    { return getInfo(typeName, distdataName).isDefined(); }
```

Returns a string with a list of all defined distdata types for the specified type (could e.g. be used in the type checking functions in case of errors to show possible values).

```
        static string definedNames(const string& typeName);

    private:
        typedef map<string, DistDataInfo>::iterator distdataIter;

        static bool initialized;
        static map<string, int> distdataIds;
        static DistDataInfo defaultInfo;
        static map<string, DistDataInfo> distdataInfos;
        static map<string, string> defaultNames;

    //////////////////////////////////////////////////////////////////////
    // Defined getdata functions:
    //////////////////////////////////////////////////////////////////////
```

Getdata function for the `int` type constructor.

```
        static DistData *getDataInt(const void *attr);
```

Getdata function for the `real` type constructor.

```
        static DistData *getDataReal(const void *attr);
```

Getdata function for the `hpoint` type constructor.

```
        static DistData *getDataHPoint(const void *attr);
```

Getdata function for the `string` type constructor.

```
        static DistData *getDataString(const void *attr);
    }; // class DistDataReg

    } //namespace gta

    #endif // #ifndef __DISTDATA_REG_H__
```

1.8 Headerfile `DistfunReg.h`

January-May 2008, Mirko Dibbert

1.8.1 Overview

This headerfile declares the following classes:

- `DistfunInfo`
  Objects of this class store a function pointer to a distance function, which computes the distance between two `DistData` objects. Each `DistfunInfo` object is assigned to a distdata-type, which specifies the expected content of the `DistData` objects.
- `DistfunReg`
  This class managages all defined `DistfunInfo` objects. Each `DistfunInfo` object could be unambigously selected by the name of the distance function and the expected distdata-type (e.g. the name of the distance function, the name of the distdata-type and the name of the type constructor or the name of the distance function and a `DistDataId` object).

New distance functions must be registered in the `DistfunReg::initialize` method.

1.8.2 Includes and defines

```
#ifndef __DISTFUN_REG_H__
#define __DISTFUN_REG_H__
```

```
#include "DistDataReg.h"

namespace gta
{

// typedef for distance functions
typedef void (*Distfun)(
        const DistData *data1, const DistData *data2,
        double &result);
```

Refers the default distance function for the each type constructor.

```
symbols::Sym DFUN_DEFAULT("default");
```

This value is returned from `DistfunReg::defaultName()`, if no default distance function has been found for the resp. type constructor.

```
symbols::Sym DFUN_UNDEFINED("n/a");

//////////////////////////////////////////////////////////////////////
// Name and short descriprions for the defined distance functions:
//////////////////////////////////////////////////////////////////////
symbols::Sym DFUN_EUCLID("euclid");
symbols::Sym DFUN_EUCLID_DESCR("euclidean distance function");

symbols::Sym DFUN_EDIT_DIST("edit");
symbols::Sym DFUN_EDIT_DIST_DESCR("edit distance function");

symbols::Sym DFUN_QUADRATIC("quadratic");
symbols::Sym DFUN_QUADRATIC_DESCR(
    "quadratic distance function using a similarity matrix");

//////////////////////////////////////////////////////////////////////
// Flags for the "DistfunInfo" class
//////////////////////////////////////////////////////////////////////

// this flag is set for all defined info objects
// (will automatically assigned from the constructor)
const char DFUN_IS_DEFINED = (1 << 0);

// if set, the info object will be used as default for the resp. type
// constructor (if more than one info object for the same type
// constructor is specified as default, the least added one will be
// used)
const char DFUN_IS_DEFAULT = (1 << 1);

// should be set, if the respective distance function is a metric
const char DFUN_IS_METRIC  = (1 << 2);
```

1.8.3 Class *DistfunInfo*

```
class DistfunInfo
```

```
{

public:
```

Default constructor (creates an undefined info object)

```
        DistfunInfo();
```

Constructor (creates a new info object with the given values)

```
        DistfunInfo(
                const string &name, const string &descr,
                const Distfun distfun, DistDataInfo distdataInfo,
                char flags = 0);
```

Returns the name of the distance function.

```
        inline string name() const
        { return m_name; }
```

Returns the description of the distance function.

```
        inline string descr() const
        { return m_descr; }
```

Returns the assigned distance funcion.

```
        inline Distfun distfun() const
        { return m_distfun; }
```

Returns the assigned getdata function.

```
        inline GetDataFun getDataFun() const
        { return m_distdataInfo.getDataFun(); }
```

Computes the distance between `data1` and `data2` with the assigned distance function.

```
        inline void dist(const DistData *data1, const DistData *data2,
                         double &result) const
        { m_distfun(data1, data2, result); }
```

Returns the assigned `DistDataInfo` object.

```
        inline DistDataInfo data() const
        { return m_distdataInfo; }
```

Generates a new `DistData` object from `attr` by applying the assigned getdata function.

```
        inline DistData *getData(const Attribute *attr)
        { return m_distdataInfo.getData(attr); }
```

Returns `true`, if the `DistfunInfo` object is defined (should only return `false` for the default `DistfunInfo` object, which is returned, if a requested distance function could not be found in the `DistfunReg` class.

```
        inline bool isDefined() const
        { return (m_flags & DFUN_IS_DEFINED); }
```

Returns `true`, if the info object is the default info object for the respective type constructor.

```
        inline bool isDefault() const
        { return (m_flags & DFUN_IS_DEFAULT); }
```

Returns `true`, if the assigned distance function is a metric.

```
        inline bool isMetric() const
        { return (m_flags & DFUN_IS_METRIC); }

    private:
        string m_name;               // unique name (used for selection)
        string m_descr;              // short description
        Distfun m_distfun;           // assigned distance function
        DistDataInfo m_distdataInfo; // assigned distdata info object
        char m_flags;
    }; // class DistfunInfo
```

1.8.4 Class *DistfunReg*

```
    class DistfunReg
    {

    public:
```

Initializes the distance functions and distdata types (in particular sets the default distdata types)

```
        static void initialize();
```

Returns `true`, if the `initialize` function has already been called.

```
        static inline bool isInitialized()
        { return initialized; }
```

Adds a new `DistfunInfo` object.

```
        static void addInfo(DistfunInfo info);
```

Returns the name of the default distance function for the specified type.

```
        static string defaultName(const string &typeName);
```

Returns the specified `DistfunInfo` object.

```
        static DistfunInfo &getInfo(
                const string &distfunName, DistDataId id);
```

Returns the specified `DistfunInfo` object.

```
        static inline DistfunInfo &getInfo(
                const string &distfunName, const string &typeName,
                const string &dataName)
        {
            return getInfo(
                    distfunName, DistDataReg::getId(typeName, dataName));

        }
```

Returns `true`, if the specified `DistFunInfo` object does exist.

```
        static inline bool isDefined(
                const string &distfunName, const string &typeName,
                const string &distdataName)
        {
            return getInfo(
                    distfunName, typeName, distdataName).isDefined();
        }
```

Returns `true`, if the specified `DistFunInfo` object does exist.

```
        static inline bool isDefined(
                const string &distfunName, DistDataId id)
        { return getInfo(distfunName, id).isDefined(); }
```

Returns a string with a list of all defined distance functions for the specified type (could e.g. be used in the type checking functions in case of errors to show possible values).

```
        static string definedNames(const string &typeName);
```

Returns a list with all defined `DistfunInfo` objects.

```
        static void getInfoList(list<DistfunInfo> &result);
```

Prints a structured list with all defined distance functions to cmsg.info().

```
        static string printDistfuns();

    private:
        typedef map<string, DistfunInfo>::iterator distfunIter;

        static bool initialized;
        static bool distfunsShown;
        static map<string, string> defaultNames;
        static DistfunInfo defaultInfo;
        static map<string, DistfunInfo> distfunInfos;

    //////////////////////////////////////////////////////////////////////
    // Defined distance functions:
    //////////////////////////////////////////////////////////////////////
```

Euclidean distance function for the `int` type constructor.

```
        static void EuclideanInt(
                const DistData *data1, const DistData *data2,
                double &result);
```

Euclidean distance function for the `real` type constructor.

```
        static void EuclideanReal(
                const DistData *data1, const DistData *data2,
                double &result);
```

Euclidean distance function for the `hpoint` type constructor.

```
        static void EuclideanHPoint(
                const DistData *data1, const DistData *data2,
                double &result);
```

Edit distance function for the `string` type constructor.

```
        static void EditDistance(
                const DistData *data1, const DistData *data2,
                double &result);
    }; // class DistfunReg

    } //namespace gta
    #endif // #ifndef __DISTFUN_REG_H__
```

2 The general-gree (gtree) framework

January-May 2008, Mirko Dibbert

This framework provides some functionality to build trees of various types, e.g. m-trees or x-trees (see MTreeAlgebra and XTreeAlgebra for examples). All framework classes belong to the `gtree` namespace.

The framework consists of the following files and classes:

- GTree.h (this file)
  includes all other framework files
- GTree_Msg.h
  class `Msg` - contains static methods to print warning and error messages
- GTree_Config.h
  contains some constants, that could be used to configurate the framework
- GTree_EntryBase.h, GTree_InternalEntry.h, GTree_LeafEntry.h
  theese files contain the base classes for node entries
- GTree_Config.h
  class `NodeConfig` (provides some config data for each node type)
- GTree_NodeBase.h, GTree_GenericNodeBase.h, GTree_Internal_Node.h, GTree_LeafNode.h
  theese files contain the base classes for nodes

- GTree_NodeManager.h
  class `NodeManager` (manages node prototypes, could create new nodes of a specific type)
- GTree_FileNode.h
  class `FileNode` (wraps NodeBase pointest to extend them with a persistence mechanism)
- GTree_TreeManager.h
  class `TreeManager`
- GTree_TreeManager.h
  class `TreeManager`
- GTree_Header.h
  class `Header` (default tree header)
- GTree_Tree.h
  class `Tree` (base class for the trees)

The basic procedure to implement such trees is as follows:

1. Define a tree class, which must be derived from `Tree`. The first template parameter must refer to the header class for the tree file, which should be derived from `Header`. This class will automatically be stored to the first page(s) of the file. The only restriction is, that it may not contain any dynamic members (pointers, strings, stl-types, etc), duo to the used persistence mechanism.

2. Define the entry classes, which should be derived from `InternalEntry` or `LeafEntry`. Each of these classes must provide at least the following methods:

```
size_t size()
void write(char *const buffer, int &offset) const
void read(const char *const buffer, int &offset)
```

The `write` and `read` method should copy the members to/from `buffer` and increase `offset`. The `size` method must return the number of bytes, that the read/write methods will copy (ensure, that the `size` method returns the correct value, since errors could easily happen here, in particular if further members are added later in the development process). All mentioned methods must call the respective method of the base class.

3. Define the node classes, which should be derived from `InternalNode` or `LeafNode`. If no further functionality is needed, the `gtree` classes could be used directly (e.g. done in the MTreeAlgebra). Otherwhise they could be extended with individual members (e.g. done in the XTreeAlgebra). If only new methods and non-persistent members should be added, the XTreeAlgebra nodes could refer as example. If new members must be stored, the read/write methods must be overwritten, and the `emptySize` parameter of the base class constructor must be called with the size of the new members in bytes (default = 0). Example:

```
class myNode : public gtree::InternalNode
{
  public:
    myNode(gtree::NodeConfigPtr config)
        : gtree::InternalNode(config, sizeof(newMember))
    {}

    myNode(const myNode &node)
       : gtree::InternalNode(node),
         newMember(node.newMember)
   {}

   virtual myNode *clone() const
   { return new myNode(*this); }

    void write(char *const buffer, int &offset) const
    {
       gtree::InternalNode::write(buffer, offset);
       memcpy(buffer, &newMember, sizeof(int));
       offset += sizeof(int);
    }

    void read(const char *const buffer, int &offset)
    {
       gtree::InternalNode::read(buffer, offset);
       memcpy(&newMember, buffer, sizeof(int));
       offset += sizeof(int);
    }

  private:
    int newMember;
};
```

If the default node classes are not sufficient (e.g. since storing the entries into a vector), it is also possible to create complete new node types, since the default classes are not used directely anywhere in the framework. The only restriction is, that these classes must be derived from the `NodeBase` class.

4. Add node prototypes to the framework (e.g. in the constructor or an initialize method) with the `addNodePrototype` method of the `Tree` class. These prototypes are used to create new nodes (by applying the copy constructor) of the respective node. If the node cache should be used, the `treeMngr->enableCache()` should also be called at this point (if some node types should not be cached, set the cacheable flag of the resp. `NodeConfig` object to `false`). It is recommended to define a `NodeTypeId` constant for each node type for easier creation of the nodes.

Now, the framework is ready to be used. For further details see the mentioned files (in particular `GTree_Tree.h` and `GTree_TreeManager.h`) and refer the MTree- and XTreeAlgebra for some practical examples.

Generally, the first action will be the creation of a new root node, which could easily be done by calling the `createRoot` method. This method is implemented in two variants, whereas the second one allows to insert two (internal) entries into the new root, e.g. usefull if a new root must be created during node splits. The resp. counters in the `Header` class (`height`, `internalEntry` and `leafEntry`) will automatically be incremented within these methods.

To traverse the tree use `treeMngr->initPath()` to init a new path. Now, the resp. `treeMngr` methods (`getChield`, `getParent`, `hasChield`, `hasParent`, `curNode`, etc.) could be used to traverse the tree. If a node must be split, the `createNeighbourNode` could be used to create a new node on the same level as `treeMngr->curNode()`.

The last basic one needs to know is the cast mechanism: To access node specific methods, the node object needs to be casted into the appropriate type to access the entry and specific methods. Since usually one will store `NodePtr` pointers (which is equal to `SmartPtr<FileNode>`), the easiest way to do this ist the `cast` method of the `FileNode` class. For example, casting `NodePtr p` to `MyNodeClass` could be done as follows:

```
SmartPtr<MyNodeClass> myNode = p->cast<MyNodeClass>();
MyEntryClass *e myNode->entry(i);
```

(The `entry` method of the default node classes return pointers of the assigned entry type)

```
#ifndef __GTREE_H__
#define __GTREE_H__

#include "GTree_Tree.h"
#include "GTree_LeafNode.h"
#include "GTree_InternalNode.h"

#endif // #define __GTREE_H__
```

2.1 Headerfile `GTree_Config.h`

January-May 2008, Mirko Dibbert

This file contains some constants and defines, which could be used to configurate the framework.

```
#ifndef __GTREE_CONFIG_H__
#define __GTREE_CONFIG_H__

#include "WinUnix.h"
#include "GTree_Msg.h"

namespace gtree
{

////////////////////////////////////////////////////////////////////////
// activate debug mode and open-object-counter for the framework
//
// Warning: Never define __GTREE_DEBUG somewhere else, otherwhise the
// framework is not guaranted to work correct, since it's object
// files could e.g. be compiled with enabled debug mode, which
// would lead to errors if they are used from headers where the debug
// mode is disabled duo to the EntryBase class, which uses a virtual
// destructor in debug mode and a static one otherwhise.
////////////////////////////////////////////////////////////////////////
// #define __GTREE_DEBUG
// #define ENABLE_OBJECT_COUNT
/////////////////////////////////////////////////////////////////////
// show "writing cached nodes to disc" message
/////////////////////////////////////////////////////////////////////
// #define __GTREE_SHOW_WRITING_CACHED_NODES_MSG



/////////////////////////////////////////////////////////////////////
// print hits/misses statistic for the node cache
/////////////////////////////////////////////////////////////////////
// #define __GTREE_SHOW_CACHE_INFOS



/////////////////////////////////////////////////////////////////////
// node pagesize (must be smaller than the system pagesize)
// - a small amount of bytes per page is reserved for maintenance -
//
// if an error like
//
//        DbEnv: Record size of x too large for page size of y
//
// occurs, the pagesize needs to be reduced!
//
// Warning: changing "PAGESIZE"[4] could propably lead to errors on
// existing trees, which are based on this framework (e.g. m-trees
// and x-trees), thus these trees should be rebuild in this case!
/////////////////////////////////////////////////////////////////////
const unsigned PAGESIZE = (WinUnix::getPageSize() - 60);



/////////////////////////////////////////////////////////////////////
// node cache sizes
//
// (if the size of the node cache exceeds maxNodeCacheSize, it would
// remove nodes from the cache, until the size is smaller than
// minNodeCacheSize)
/////////////////////////////////////////////////////////////////////
const unsigned minNodeCacheSize = 7*1024*1024; // default 7 MB
const unsigned maxNodeCacheSize = 9*1024*1024; // default 9 MB

/////////////////////////////////////////////////////////////////////
// maximum size of a node that should be able to be cached
// (should be smaller or equal or maxNodeCacheSize)
/////////////////////////////////////////////////////////////////////
const unsigned maxCacheableSize = 9*1024*1024; // default 9 MB



} // namespace gtree

// must be included after the ENABLE_OBJECT_COUNT define
```

```
#include "ObjCounter.h"

#endif // #define __GTREE_CONFIG_H__
```

2.2 Headerfile `GTree_EntryBase.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_ENTRY_BASE_H__
#define __GTREE_ENTRY_BASE_H__

#include "GTree_Config.h"

namespace gtree
{
```

Class *EntryBase*

This class must be the base of all node entries in this framework.

If `__GTREE_DEBUG` is defined, the class will compile with a virtual destructor, which is used for further type checkings in some methods by using `dynamic_cast` instead of `static_cast` to avoid using incompatible entry and node types, e.g. inserting leaf entries into internal nodes.

```
class EntryBase
    : public ObjCounter<generalTreeNodes>
{

public:
```

Default constructor.

```
    inline EntryBase()
    {}
```

Default copy constructor.

```
    inline EntryBase(const EntryBase &source)
    {}
```

Destructor

```
#ifdef __GTREE_DEBUG
    inline virtual ˜EntryBase()
    {}

#else
    inline ˜EntryBase()
    {}

#endif
}; // class EntryBase

} // namespace gtree
#endif // #define __GTREE_ENTRY_BASE_H__
```

2.3 Headerfile `GTree_InternalEntry.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_INTERNAL_ENTRY_H__
#define __GTREE_INTERNAL_ENTRY_H__

#include "GTree_EntryBase.h"
#include "GTree_FileNode.h"

namespace gtree
{
```

Class *InternalEntry*

This class should be used as base class for all internal entries. The derived classes must overwrite the `read`, `write` and `size` method and call the resp. methods of the base class.

```
class InternalEntry
            : public EntryBase
{

public:
```

Default constructor.

```
    inline InternalEntry(SmiRecordId chield = 0)
            : m_chield(chield)
    {}
```

Constructor (sets chield-id to the record-id of `node`).

```
inline InternalEntry(NodePtr node)
        : m_chield(node->getNodeId())
{}
```

Default copy constructor.

```
inline InternalEntry(const InternalEntry &e)
        : m_chield(e.m_chield)
{}
```

Destructor.

```
inline ~InternalEntry()
{}
```

Writes the entry object to buffer and increses offset.

```
inline void write(char *const buffer, int &offset) const
{
    memcpy(buffer+offset, &m_chield, sizeof(SmiRecordId));
    offset += sizeof(SmiRecordId);
}
```

Reads the entry object from buffer and increses offset.

```
inline void read(const char *const buffer, int &offset)
{
    memcpy(&m_chield, buffer+offset, sizeof(SmiRecordId));
    offset += sizeof(SmiRecordId);
}
```

Returns the size of the entry on disc.

```
inline size_t size()
{ return sizeof(SmiRecordId); }
```

Returns the record-id of the chield node.

```
SmiRecordId chield() const
{ return m_chield; }
```

Sets the chield record-id to `nodeId`.

```
inline void setChield(SmiRecordId nodeId)
{ m_chield = nodeId; }
```

Sets the chield record-id to the record-id of `node`.

```
    inline void setChield(NodePtr node)
    { m_chield = node->getNodeId(); }

private:
    SmiRecordId m_chield; // pointer to chield node
}; // class InternalEntry

} // namespace gtree
#endif // #define __GTREE_INTERNAL_ENTRY_H__
```

2.4 Headerfile `GTree_LeafEntry.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_LEAF_ENTRY_H__
#define __GTREE_LEAF_ENTRY_H__

#include "GTree_EntryBase.h"

namespace gtree
{
```

Class *LeafEntry*

This class should be used as base class for all leaf entries. The derived classes must overwrite the `read`, `write` and `size` method and call the resp. methods of the base class (these methods have currently no affect in this case, but nevertheless they should be called for the case of future extensions).

```
class LeafEntry
            : public EntryBase
{

public:
```

Default constructor.

```
    inline LeafEntry()
    {}
```

Default copy constructor.

```
        inline LeafEntry(const LeafEntry &e)
        {}
```

Destructor.

```
        inline ~LeafEntry()
        {}
```

Writes the entry object to buffer and increses offset.

```
        inline void write(char *const buffer, int &offset) const
        {}
```

Reads the entry object from buffer and increses offset.

```
        inline void read(const char *const buffer, int &offset)
        {}
```

Returns the size of the entry on disc.

```
        inline size_t size()
        { return 0; }
    };

    } // namespace gtree
    #endif // #define __GTREE_LEAF_ENTRY_H__
```

2.5 Headerfile `GTree_NodeConfig.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_NODE_CONFIG_H__
#define __GTREE_NODE_CONFIG_H__

namespace gtree
{



class NodeConfig; // forward declaration
typedef SmartPtr<NodeConfig> NodeConfigPtr;
typedef unsigned char NodeTypeId;
```

Class *NodeConfig*

This class contains the node type id and some data to configurate the node.

The `minEntries` value could be used to enforce a minimum node size, before `Node::insert` is allowed to return `true`, which means that the node should be split (if not, further entries will be inserted nevertheless). The default value guaranties, that the tree does not degenerate to a linked list.

The `maxEntries` and `maxPages` values limit the maximum entrie count for the node. If one of these values is exceeded and the node contains at least `minEntries` entries, the `insert` method will return `true` (split node).

Setting minEntries to 0 would lead to nodes of (potentially) unlimited size (e.g. used for the supernodes in the xtree).

The `userext` pointer is provides a reference to any possible structure, which is avaliable in all nodes (e.g. some data from the tree header). To access this ref. it is neccesary to define individual node types, which then could access the reference with `m_config->userext()` and a respective typecast.

Warning: modifying the config object of a node would affect all nodes of the same type, since all these nodes use a reference to the same config object.

```
class NodeConfig
{

public:
```

Constructor

```
    NodeConfig(
        NodeTypeId type,
        unsigned priority = 0,
        unsigned minEntries = 2,
        unsigned maxEntries = numeric_limits<unsigned>::max(),
        unsigned maxPages = 1,
        bool cacheable = true,
        void *userext = 0)
            : m_type(type), m_priority(priority),
            m_minEntries(minEntries),
            m_maxEntries(maxEntries),
            m_maxPages(maxPages),
            m_cacheable(cacheable),
            m_userext(userext)
    {
        if (m_minEntries == 0)
            m_minEntries = numeric_limits<unsigned>::max();

        if (m_maxEntries < m_minEntries)
            m_maxEntries = m_minEntries;
    }
```

Destructor

```
    inline ~NodeConfig() {}
```

Returns the id of the node type.

```
    inline NodeTypeId type() const
    { return m_type; }
```

Returns the priority of the node.

```
    inline unsigned priority() const
    { return m_priority; }
```

Returns the minimum count of entries in the node.

```
        inline unsigned minEntries() const
        { return m_minEntries; }
```

Returns the maximum count of entries in node.

```
        inline unsigned maxEntries() const
        { return m_maxEntries; }
```

Returns the minimum count of pages of the node.

```
        inline unsigned maxPages() const
        { return m_maxPages; }
```

Returns `true` if the node could be cached.

```
        inline bool cacheable() const
        { return m_cacheable; }
```

Returns a reference to the user extension object.

```
        inline void *userext() const
        { return m_userext; }
```

Sets a new user extension object.

```
        inline void setUserext(void* ext)
        { m_userext = ext; }

    private:
        NodeTypeId m_type;      // id of the node type
        unsigned m_priority;    // priority of the node
        unsigned m_minEntries; // minimum count of entries per node
        unsigned m_maxEntries; // maximum count of entries per node
        unsigned m_maxPages;    // min. size of the node in memory pages
        bool m_cacheable;       // cacheable flag
        void *m_userext;        // reference to user defined object
    }; // class NodeConfig

    } // namespace gtree
    #endif // #define __GTREE_NODE_CONFIG_H__
```

2.6 Headerfile `GTree_NodeBase.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_NODE_BASE_H__
#define __GTREE_NODE_BASE_H__


#include "SmartPtr.h"
#include "GTree_EntryBase.h"
#include "GTree_NodeConfig.h"


namespace gtree
{



class FileNode; // forward declaration
typedef SmartPtr<FileNode> NodePtr;
```

Class *NodeBase*

This is the base class for all nodes in the framework.

```
class NodeBase
    : public ObjCounter<generalTreeNodes>
{
```

```
friend class FileNode;

public:
```

Default constructor.

```
        inline NodeBase(NodeConfigPtr config, unsigned emptySize)
            : m_config(config),
              m_emptySize(emptySize +
                     sizeof(NodeTypeId) +  // type-id
                     sizeof(size_t) +      // count of extension records
                     sizeof(SmiRecordId)), // first extension page
              m_curSize(m_emptySize),
              m_modified(true),
              m_cached(false),
              m_pagecount(1),
              m_level(0)
        {}
```

Default copy constructor.

```
        inline NodeBase(const NodeBase& node)
            : m_config(node.m_config),
              m_emptySize(node.m_emptySize),
              m_curSize(node.m_curSize),
              m_modified(node.m_modified),
              m_cached(node.m_cached),
              m_pagecount(node.m_pagecount),
              m_level(node.m_level)
        {}
```

Virtual destructor.

```
        inline virtual ~NodeBase()
        {}
```

Returns the type-id of the node.

```
        inline NodeTypeId typeId() const
        { return m_config->type(); }
```

Returns state of the cached flag.

```
        inline bool isCached() const
        { return m_cached; }
```

Returns true, if the node could be stored in the node cache.

```
        inline bool isCacheable() const
        { return m_config->cacheable(); }
```

Sets the cached flag to `true`.

```
        inline void setCached()
        { m_cached = true; }
```

Sets the cached flag to `false`.

```
        inline void resetCached()
        { m_cached = false; }
```

Returns state of the modified flag.

```
        inline bool isModified() const
        { return m_modified; }
```

Sets the modified flag to `true`.

```
        inline void setModified()
        { m_modified = true; }
```

Sets the modified flag to `false`.

```
        inline void resetModified()
        { m_modified = false; }
```

Returns the count of pages, which this node would need in the file.

```
        inline unsigned pagecount() const
        { return m_pagecount; }
```

Returns the level of the node in the tree.

```
        inline unsigned level() const
        { return m_level; }
```

Increases the level of the node in the tree.

```
        inline void incLevel()
        { ++m_level; }
```

Decreases the level of the node in the tree.

```
        inline void decLevel()
        { --m_level; }
```

Sets the level of the node in the tree.

```
        inline void setLevel(unsigned level)
        { m_level = level; }
```

Returns the priority of the node.

```
        inline unsigned priority() const
        { return m_config->priority(); }
```

Should return `true` for leaf nodes and `false` for internal nodes.

```
virtual bool isLeaf() const = 0;
```

Should read the node from buffer and increase offset.

```
virtual void read(const char *const buffer, int& offset) = 0;
```

Should write the node to buffer and increase offset.

```
virtual void write(char *const buffer, int& offset) const = 0;
```

Should return a copy of the node.

```
virtual NodeBase *clone() const = 0;
```

Should return the `SmiRecordId` of the i-th subtree of the node, if the node is an internal node and 0 for leaf nodes.

```
virtual SmiRecordId chield(unsigned i) const = 0;
```

Should recompute the size of the node and set the modified flag (needed, if the entry representation has been manimpulated directly)

```
virtual void recomputeSize() = 0;
```

Inserts a new entry into the node. Should return `true`, if the node needs to be splitted after insert (the entry should be inserted in any case).

```
virtual bool insert(EntryBase *e) = 0;
```

Like *insert*, but inserts a copy of `e` by calling the copy constructor of the respective entry class.

```
virtual bool insertCopy(EntryBase *e) = 0;
```

Should return the count of all contained entries.

```
virtual unsigned entryCount() const = 0;
```

Should remove all entries.

```
virtual void clear() = 0;
```

Should remove the i-th entry.

```
virtual void remove(unsigned i) = 0;
```

Should replace the i-th emtry with `newEntry`.

```
virtual void replace(unsigned i, EntryBase *newEntry) = 0;
```

Should return the size of the entry im memory, which is needed to update the size of the node-cache, if used. Since this size is only needed to check, if the node cache has reached it's maximum size, it is sufficient to compute an approximately value to avoid complex computations.

If the `recompute` value is false, the last value (stored in m_memSize) should be returned. Otherwhise the memory size should be recomputed first. A call with parameter `recompute = false` is needed from the tree manager in the `recomputeSize` method, which calls respective method of this class.

```
        virtual unsigned memSize(bool recompute = true) const = 0;
```

Returns the i-th entry in the node, needed it the respective node class is not known.

```
        virtual EntryBase *baseEntry(unsigned i) const = 0;

    protected:
        NodeConfigPtr m_config;     // ref. to the NodeConfig object
        unsigned      m_emptySize;  // size of an empty node
        unsigned      m_curSize;    // curent size of the node
        bool          m_modified;   // modified flag
        bool          m_cached;     // cached flag
        unsigned      m_pagecount;  // count of extension pages
        unsigned      m_level;      // level of the node
        mutable unsigned m_memSize; // current size of the node in memory
    }; // class NodeBase

    } // namespace gtree
    #endif // #define __GTREE_NODE_BASE_H__
```

2.7 Headerfile `GTree_GenericNodeBase.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_GENERIC_NODE_BASE_H__
#define __GTREE_GENERIC_NODE_BASE_H__

#include <vector>
#include "GTree_NodeBase.h"

namespace gtree
{
```

Class *GenericNodeBase*

Base class for `InternalNode` and `LeafNode` - implements the `NodeBase` class, using a vector.

The following methods hat been added additionally to the `NodeBase` methods:

- `begin`, `end`, `entryIt`, `reserve` (access to some methods of the entry vector)
- `remove` for iterators
- `fastRemove` methods (faster than remove, but ignores entry order)

```
template<class TEntry>
class GenericNodeBase
    : public gtree::NodeBase
{
public:
    typedef typename vector<TEntry*>::iterator iterator;
    typedef TEntry entryType;
```

Constructor

```
        inline GenericNodeBase(
                NodeConfigPtr config, unsigned emptySize)
            : NodeBase(config, emptySize + sizeof(size_t)),
              m_entries(new vector<TEntry*>())
        {}
```

Default copy constructor.

```
        inline GenericNodeBase(const GenericNodeBase& node)
            : NodeBase(node), m_entries(new vector<TEntry*>())
        {
            // copy entry vector
            for (iterator it = node.begin(); it != node.end(); ++it)
                m_entries->push_back(new TEntry(**it));
        }
```

Virtual destructor.

```
        virtual ~GenericNodeBase()
        {
            for (iterator it = begin(); it != end(); ++it)
                delete *it;

            delete m_entries;
        }
```

Returns an iterator to the begin of the entry vector.

```
        inline iterator begin() const
        { return m_entries->begin(); }
```

Returns an iterator to the end of the entry vector.

```
        inline iterator end() const
        { return m_entries->end(); }
```

Returns an iterator to the i-th entry.

```
        inline iterator entryIt(unsigned i) const
        {
    #ifdef __GTREE_DEBUG
            assert(i < entryCount());
    #endif

            return m_entries->begin() + i;
        }
```

Returns the entry vector for direct access. If entries has been added or removed from the vector, the `recomputeSize` method has to be called.

```
inline vector<TEntry*> *entries() const
{ return m_entries; }
```

Reserves space for `n` entries in the entry vector.

```
inline void reserve(size_t n)
{ entries->reserve(n); }
```

Sets the modified flag and recomputes `m_curSize` and `m_pagesize`, which is nessesary, if the entry vector has been directly manipulated.

```
virtual void recomputeSize();
```

Inserts a new entry into the node and returns `true`, if the node should been splitted.

```
virtual bool insert(EntryBase *newEntry);
```

Like *insert*, but inserts a copy of `e`, using the respective copy constructor.

```
virtual bool insertCopy(EntryBase *newEntry);
```

Returns the count of all entries in the node.

```
virtual inline unsigned entryCount() const
{ return m_entries->size(); }
```

Removes all entries from the node.

```
virtual inline void clear()
{
    m_curSize = m_emptySize;
    m_entries->clear();
    setModified();
}
```

Removes the i-th entry from the node. Since the entries are hold in a vector, all entries after the i-th position have to be moved to the new position, wich could be a performance problem, in particular for nodes with much entries. However this shouldn't be a problem in most cases, since the vector contains only pointers to the entry. If the order of the entries doesen't matter, `fastRemove` could be used instead.

```
virtual void remove(unsigned i);
```

Like above remove method, but expects an iterator to the entry.

```
virtual void remove(iterator it);
```

Removes the i-th entry from the node and moves the last entry to that position. This is faster than the default remove method, but destroyes the entry order.

```
        virtual void fastRemove(unsigned i);
```

Like above fastRemove method, but expects an iterator to the entry.

```
        virtual void fastRemove(iterator it);
```

Replaced the i-th entry with `newEntry`.
Warning: Do not replace an entry with itself, since this would lead to a crash!

```
        virtual void replace(unsigned i, EntryBase *newEntry);
```

Returns a reference to the i-th entry.

```
        inline TEntry *entry(unsigned i) const;
```

Like above entry method, but returns an `EntryBase` pointer (e.g. used in the `Tree` copy constructor).

```
        virtual inline EntryBase *baseEntry(unsigned i) const;
```

Returns the (approximately) size of the entry im memory.

```
        virtual inline unsigned memSize(bool recompute = true) const;

    protected:
```

Reads the node from buffer and increses offset.

```
        virtual void read(const char *const buffer, int &offset);
```

Writes the node to buffer and increses offset.

```
        virtual void write(char *const buffer, int &offset) const;

        vector<TEntry*> *m_entries; // entry vector
    }; // GenericNodeBase



    //////////////////////////////////////////////////////////////////////
    //////////////////////////////////////////////////////////////////////
    //////////////////////////////////////////////////////////////////////
```

Method *recomputeSize*:

```
template<class TEntry>
void GenericNodeBase<TEntry>::recomputeSize()
{
    setModified();

    // recompute size
    m_curSize = m_emptySize;
    for (iterator it = begin(); it != end(); ++it)
        m_curSize += (*it)->size();

    // recompute pagecount
    m_pagecount = 1;
    while (m_curSize > m_pagecount * PAGESIZE)
        ++m_pagecount;
}
```

Method *insert*:

```
template<class TEntry>
bool GenericNodeBase<TEntry>::insert(EntryBase *newEntry)
{
#ifdef __GTREE_DEBUG
    TEntry *e = dynamic_cast<TEntry*>(newEntry);
    if (!e) Msg::wrongEntryType_Error();

#else
    TEntry *e = static_cast<TEntry*>(newEntry);
#endif

    // insert entry and update node size
    m_entries->push_back(e);
    m_curSize += e->size();
    setModified();

    // increase pagecount, if neccessary
    while (m_curSize > (m_pagecount * PAGESIZE))
        ++m_pagecount;

    if ((
            (entryCount() <= m_config->maxEntries()) &&
            (m_pagecount <= m_config->maxPages())
        ) ||
        (entryCount() <= m_config->minEntries()))
    {   // node should not be splitted
        return false;
    }

    // node should be splitted
    return true;
}
```

Method *insertCopy*:

```
template<class TEntry>
bool GenericNodeBase<TEntry>::insertCopy(EntryBase *newEntry)
{
#ifdef __GTREE_DEBUG
    TEntry *e = dynamic_cast<TEntry*>(newEntry);
    if (!e) Msg::wrongEntryType_Error();

#else
    TEntry *e = static_cast<TEntry*>(newEntry);
#endif

    // insert entry and update node size
    m_entries->push_back(new TEntry(*e));
    m_curSize += e->size();
    setModified();

    // increase pagecount, if neccessary
    while (m_curSize > (m_pagecount * PAGESIZE))
        ++m_pagecount;

    if ((
            (entryCount() <= m_config->maxEntries()) &&
            (m_pagecount <= m_config->maxPages())
        ) ||
        (entryCount() <= m_config->minEntries()))
    {   // node should not be splitted
        return false;
    }

    // node should be splitted
    return true;
}
```

Method *remove* (position):

```
template<class TEntry>
void GenericNodeBase<TEntry>::remove(unsigned i)
{
#ifdef __GTREE_DEBUG
    assert(i < entryCount());
#endif

    m_curSize -= entry(i)->size();
    delete entry(i);
    entries()->erase(entryIt(i));

    setModified();
}
```

Method *remove* (iterator):

```
template<class TEntry>
void GenericNodeBase<TEntry>::remove(iterator it)
```

```
{
    m_curSize -= (*it)->size();
    delete *it;
    entries()->erase(it);

    setModified();
}
```

Method *fastRemove* (position):

```
template<class TEntry>
void GenericNodeBase<TEntry>::fastRemove(unsigned i)
{
#ifdef __GTREE_DEBUG
    assert(i < entryCount());
#endif

    m_curSize -= entry(i)->size();
    delete entry(i);
    *entryIt(i) = m_entries->back();
    m_entries->pop_back();

    setModified();
}
```

Method *fastRemove* (iterator):

```
template<class TEntry>
void GenericNodeBase<TEntry>::fastRemove(iterator it)
{
    m_curSize -= (*it)->size();
    delete *it;
    *it = m_entries->back();
    m_entries->pop_back();

    setModified();
}
```

Method *replace*:

```
template<class TEntry>
void GenericNodeBase<TEntry>::replace(
        unsigned i, EntryBase *newEntry)
{
#ifdef __GTREE_DEBUG
    assert(i < entryCount());
    TEntry *e = dynamic_cast<TEntry*>(newEntry);
    if (!e) Msg::wrongEntryType_Error();

#else
    TEntry *e = static_cast<TEntry*>(newEntry);
#endif
```

```
        m_curSize -= entry(i)->size();
        m_curSize += e->size();
        delete entry(i);
        *entryIt(i) = e;
        setModified();
    }
```

Method *entry*:

```
    template<class TEntry>
    TEntry *GenericNodeBase<TEntry>::entry(unsigned i) const
    {
    #ifdef __GTREE_DEBUG
        assert(i < entryCount());
    #endif
        return (*m_entries)[i];
    }
```

Method *baseEntry*:

```
    template<class TEntry>
    EntryBase *GenericNodeBase<TEntry>::baseEntry(unsigned i) const
    {
    #ifdef __GTREE_DEBUG
        assert(i < entryCount());
    #endif
        return (*m_entries)[i];
    }
```

Method *memSize*:

```
    template<class TEntry>
    unsigned GenericNodeBase<TEntry>::memSize(bool recompute) const
    {
        if (recompute)
        {
            m_memSize = m_curSize +
                sizeof(GenericNodeBase<TEntry>) + // size of all members
                entryCount() * sizeof(TEntry*); // size of entry pointers
        }
        return m_memSize;
    }
```

Method *read*:

```
    template<class TEntry>
    void GenericNodeBase<TEntry>::read(
            const char *const buffer, int &offset)
    {
        m_curSize = m_emptySize;
        // read count of stored entries
        size_t count;
        memcpy(&count, buffer+offset, sizeof(size_t));
        offset += sizeof(size_t);

        // read entry vector
        m_entries->reserve(count);
        int old_offset = offset;

        for (size_t i = 0; i < count; ++i)
        {
            m_entries->push_back(new TEntry());
            m_entries->back()->read(buffer, offset);
        }

        // add  size of the entry vector to m_curSize
        m_curSize += (offset - old_offset);
    }
```

Method *write*:

```
    template<class TEntry>
    void GenericNodeBase<TEntry>::write(
            char *const buffer, int &offset) const
    {
        // write count of stored entries
        size_t count = m_entries->size();
        memcpy(buffer + offset, &count, sizeof(size_t));
        offset += sizeof(size_t);

        // write the entry array
        for (iterator it = begin(); it != end(); ++it)
        {
            (*it)->write(buffer, offset);
        }
    }



    } // namespace gtree
    #endif // #define __GTREE_GENERIC_NODE_BASE_H__
```

2.8 Headerfile `GTree_LeafNode.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_LEAF_NODE_H__
#define __GTREE_LEAF_NODE_H__

#include "GTree_GenericNodeBase.h"

namespace gtree
{
```

Class *LeafNode*

This class should be used as base for all leaf nodes.

```
template<class TEntry>
class LeafNode
    : public GenericNodeBase<TEntry>
{

public:
```

Default constructor.

```
    inline LeafNode(NodeConfigPtr config, unsigned emptySize = 0)
            : GenericNodeBase<TEntry>(config, emptySize)
    {}
```

Default copy constructor.

```
        inline LeafNode(const LeafNode& node)
                : GenericNodeBase<TEntry>(node)
        {}
```

Virtual destructor.

```
        inline virtual ˜LeafNode()
        {}
```

Returns a reference to a copy of the node.

```
        virtual LeafNode *clone() const
        { return new LeafNode(*this); }
```

Returns 0, since leaf does not have any chield nodes.

```
        virtual SmiRecordId chield(unsigned i) const
        { return 0; }
```

Returns true, to indicate that this node is a leaf node.

```
        virtual bool isLeaf() const
        { return true; }
    };

    } // namespace gtree
    #endif // #define __GTREE_LEAF_NODE_H__
```

2.9 Headerfile `GTree_InternalNode.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_INTERNAL_NODE_H__
#define __GTREE_INTERNAL_NODE_H__

#include "GTree_GenericNodeBase.h"

namespace gtree
{
```

Class *InternalNode*

This class should be used as base for all internal nodes.

```
template<class TEntry>
class InternalNode
    : public GenericNodeBase<TEntry>
{

public:
```

Default constructor.

```
    inline InternalNode(NodeConfigPtr config, unsigned emptySize = 0)
            : GenericNodeBase<TEntry>(config, emptySize)
    {}
```

Default copy constructor.

```
        inline InternalNode(const InternalNode& node)
                : GenericNodeBase<TEntry>(node)
        {}
```

Virtual destructor.

```
        inline virtual ~InternalNode()
        {}
```

Returns a reference to a copy of the node.

```
        virtual InternalNode *clone() const
        { return new InternalNode(*this); }
```

Returns the record id of the i-th chield node.

```
        virtual SmiRecordId chield(unsigned i) const
        { return (GenericNodeBase<TEntry>::entry(i))->chield(); }
```

Returns false, to indicate that this node is an internal node.

```
        virtual bool isLeaf() const
        { return false; }
    };

    } // namespace gtree
    #endif // #define __GTREE_INTERNAL_NODE_H__
```

2.10 Headerfile `GTree_NodeManager.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_NODE_MANAGER_H__
#define __GTREE_NODE_MANAGER_H__

#include <map>
#include "GTree_NodeBase.h"

namespace gtree
{
```

Class *NodeManager*

This class manages the node prototypes and could create new nodes of a specific type by calling the clone method of the respective prototype node.

```
class NodeManager
{
  public:

    typedef map<NodeTypeId, NodeBase*>::iterator iterator;
```

Constructor.

```
    NodeManager(SmiRecordFile *treeFile)
        : m_file(treeFile)
    {}
```

Destructor.

```
~NodeManager()
{
    for (iterator iter = m_prototypes.begin();
            iter != m_prototypes.end(); ++iter)
    {
        delete iter->second;
    }
}
```

Adds a new prototype node.

```
inline void addPrototype(NodeBase* node)
{ m_prototypes[node->typeId()] = node; }
```

Returns a copy of the prototype node with id `type`. If such a node does not exist, it prints an error message and halts the dbms or returns 0 (if assertions are disabled).

```
NodeBase *createNode(NodeTypeId type)
{
    iterator iter = m_prototypes.find(type);

    if (iter != m_prototypes.end())
    {
        return iter->second->clone();
    }
    else // node type not found in prototype map
    {
        Msg::undefinedNodeType_Error(type);
        return 0;
    }
}
```

Adds all prototype nodes from the `src` objects to this object (used in the `Tree` copy constructor).

```
void copyPrototypes(NodeManager *src)
{
    for (size_t i = 0; i < src->m_prototypes.size(); ++i)
        addPrototype(src->m_prototypes[i]->clone());
}
```

Returns a reference to the tree file.

```
  inline SmiRecordFile *file() const
  { return m_file; }

private:
  SmiRecordFile *m_file;
  // reference to the tree file

  map<NodeTypeId, NodeBase*> m_prototypes;
```

```
    // prototypes for all registered nodes
}; // class NodeManager

} // namespace gtree
#endif // #define __GTREE_NODE_MANAGER_H__
```

2.11 Headerfile `GTree_FileNode.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_FILE_NODE_H__
#define __GTREE_FILE_NODE_H__

#include "SecondoSMI.h"
#include "GTree_NodeManager.h"

namespace gtree
{



class FileNode; // forward declaration
typedef SmartPtr<FileNode> NodePtr;
```

Class *FileNode*

This class adds some persitence abilities to the `NodeBase` class, all methods of `NodeBase` are wrapped within respective methods of this class, thus `NodePtr` (=`SmartPtr<FileNode>`) could be used similar to `NodeBase`, but allows further filemanagement methods for the node,
e.g. `getNodeId`, `get`, `put` or `drop`.

```
class FileNode
{

public:
```

Constructor (creates a new node).

```
inline FileNode(NodeManager *mngr, NodeTypeId type)
    : m_nodeMngr(mngr), m_nodeId(0),
      m_extensionId(0), m_extPageCnt(0)
{ createNode(type); }
```

Constructor (reads the node from page `nodeId` in file).

```
inline FileNode(NodeManager *mngr, SmiRecordId nodeId)
    : m_nodeMngr(mngr)
{ get(nodeId); }
```

Default copy constructor.

```
inline FileNode(const FileNode &node)
    : m_node(node.m_node->clone()), m_nodeMngr(node.m_nodeMngr),
      m_nodeId(node.m_nodeId), m_extensionId(node.m_extensionId),
      m_extPageCnt(node.m_extPageCnt)
{}
```

Destructor (writes the node to file, if neccesary).

```
inline ~FileNode()
{ put(); }
```

Returns a copy of the `FileNode` Object.

```
inline NodePtr clone() const
{ return NodePtr(new FileNode(*this)); }
```

Returns the record id of the node. If no record exist, a new record will be appended to the file.

```
SmiRecordId getNodeId();
```

Creates a new node. If there already exists a node reference, that node would be automatically written to file.

```
inline void createNode(NodeTypeId type)
{ m_node = m_nodeMngr->createNode(type); }
```

Writes the node to file.

```
void put();
```

Reads the node from file. If there already exists a node reference, that node would be automatically written to file.

```
void get(SmiRecordId nodeId);
```

Removes the refered node from file.

```
        void drop();
```

Returns `m_node`.

```
        inline SmartPtr<NodeBase> ptr()
        { return m_node; }
```

Casts the node pointer to the node class, which is specified in the template parameter. In debug mode, an error message is shown and the dbms will terminate, if the cast was not allowed.

```
        template<class TNode>
        inline TNode* cast()
        {
            #ifdef __GTREE_DEBUG
            TNode* result = dynamic_cast<TNode*>(m_node.operator->());
            if (!result)
                Msg::invalidNodeCast_Error();
            return result;
            #else
            return static_cast<TNode*>(m_node.operator->());
            #endif
        }

    private:
        SmartPtr<NodeBase> m_node;         // Pointer to the referred node
        NodeManager        *m_nodeMngr;    // ref. to node manager
        SmiRecordId        m_nodeId;       // header record of the node
        SmiRecordId        m_extensionId;  // id of first extension page
        unsigned           m_extPageCnt;   // count of extension pages

    public:
```

The following methods wraps the respective methods of the `NodeBase` class:

```
        inline unsigned memSize(bool recompute = true) const
        { return m_node->memSize(recompute); }

        inline bool isCached() const
        { return m_node->isCached(); }

        inline void setCached()
        { m_node->setCached(); }

        inline void resetCached()
        { m_node->resetCached(); }

        inline bool isModified() const
        { return m_node->isModified(); }

        inline void setModified()
        { m_node->setModified(); }
```

```
inline void resetModified()
{ m_node->resetModified(); }

inline bool isCacheable() const
{ return m_node->isCacheable(); }

inline bool isLeaf() const
{ return m_node->isLeaf(); }

inline NodeTypeId typeId() const
{ return m_node->typeId(); }

inline SmiRecordId chield(unsigned i) const
{ return m_node->chield(i); }

inline void recomputeSize()
{ m_node->recomputeSize(); }

inline bool insert(EntryBase *e)
{ return m_node->insert(e); }

inline bool insertCopy(EntryBase *e)
{ return m_node->insertCopy(e); }

inline unsigned entryCount() const
{ return m_node->entryCount(); }

inline void clear()
{ m_node->clear(); }

inline void remove(unsigned i)
{ m_node->remove(i); }

inline void replace(unsigned i, EntryBase *newEntry)
{ m_node->replace(i, newEntry); }

inline unsigned pagecount() const
{ return m_node->pagecount(); }

inline unsigned level() const
{ return m_node->level(); }

inline void incLevel()
{ return m_node->incLevel(); }

inline void decLevel()
{ return m_node->decLevel(); }

inline void setLevel(unsigned level)
{ return m_node->setLevel(level); }

inline unsigned priority() const
{ return m_node->priority(); }
```

```
    inline EntryBase *baseEntry(unsigned i) const
    { return m_node->baseEntry(i); }
}; // class FileNode

} // namespace gtree
#endif // #define __GTREE_FILE_NODE_H__
```

2.12 Headerfile `GTree_TreeManager.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_TREE_MANAGER_H__
#define __GTREE_TREE_MANAGER_H__

#include <list>
#include "GTree_FileNode.h"
#include "GTree_InternalEntry.h"
#include "GTree_LeafEntry.h"

namespace gtree
{


typedef unsigned (*PriorityFun)(NodePtr& node);
```

Class *TreeManager*

```
class TreeManager
{

public:
    typedef map<SmiRecordId, NodePtr>::iterator iterator;
```

Default constructor:

```
        inline TreeManager(NodeManager* mngr)
            : m_curNode(), m_parentNode(), nodeSupp(mngr), m_level(0),
              useCache(false), curSize(0), minPriority(0), m_hits(0),
              m_misses(0), getPriority(TreeManager::priorityFun)
        {}

    private:
```

Default copy constructor and assignment operator (not implemented).

```
        TreeManager(const TreeManager&);
        void operator = (const TreeManager&);

    public:
```

Destructor:

```
        inline ~TreeManager()
        { clearCache(); }
```

Default priority function - could be replaced with another one with the `setPriorityFun` method.

```
        inline static unsigned priorityFun(NodePtr &node)
        {
            return 10*node->priority() +
                   5*node->level() +
                   node->pagecount();
        }
```

Sets a new priority function, which replaces the default priority function.

```
        inline void setPriorityFun(PriorityFun pf)
        { getPriority = pf; }
```

Enables the node cache.

```
        inline void enableCache()
        { useCache = true; }
```

Removes all nodes from the node cache and disables it.

```
        inline void disableCache()
        {
            clearCache();
            useCache = false;
        }
```

Removes all nodes from cache.

```
        void clearCache();
```

Returns the current size of the cache.

```
inline unsigned cacheSize()
{ return curSize; }
```

Creates a new node of the specified type. The optional `level` value specifies the level of the node in the tree (whereas leaf level = 0) and should be used, if the node cache has been enabled, since iter is used to determine the priority of the nodes in the cache.

If the node level of a cached node has been changed (e.g. if a new chield has been inserted), iter could be updated with the respective node methods (`incLevel`, `decLevel`, `setLevel`), otherwise the cache priority mechanism would possibly not work as expected.

```
NodePtr createNode(NodeTypeId type, unsigned level = 0);
```

Like *createNode*, but uses current node level as level for the new node.

```
inline NodePtr createNeighbourNode(NodeTypeId type)
{ return createNode(type, m_level); }
```

The optional `level` value specifies the level of the node in the tree (whereas leaf level = 0) and should be used, if the node cache has been enabled, since iter is used to determine the priority of the nodes in the cache.

If the node level of a cached node has been changed (e.g. if a new chield has been inserted), iter could be updated with the respective node methods (`incLevel`, `decLevel`, `setLevel`), otherwise the cache priority mechanism would possibly not work as expected.

```
NodePtr getNode(SmiRecordId nodeId, unsigned level = 0);
```

Initiates a new path from the specified root. The curNode pointer will be set to the root node.

```
void initPath(SmiRecordId rootId, unsigned rootLevel);
```

Sets the curNode pointer to the i-th chield of current node.

```
void getChield(unsigned i);
```

Sets the curNode pointer to the parent of current node.

```
void getParent();
```

Returns `true`, if the current node is not the root.

```
inline bool hasParent()
{ return !path.empty(); }
```

Returns a smart pointer to the current node.

```
inline NodePtr curNode()
{ return m_curNode; }
```

Returns a smart pointer to the parent node.

```
        inline NodePtr parentNode()
        { return m_parentNode; }
```

Returns the level of the current node in path (should be equal to `curNode()->level()`).

```
        inline unsigned curLevel()
        { return m_level; }
```

Returns a pointer to the parent entry. The node type of the parent is excepted as template parameter.

```
        template<class TNode>
        inline typename TNode::entryType* parentEntry()
        {
            // could not use the cast method of m_parerntNode, since
            // calling template member methods from template classes does
            //  not work under windows
            NodeBase* parent = m_parentNode->ptr().operator->();
            TNode* node = 0;
    #ifdef __GTREE_DEBUG
            node = dynamic_cast<TNode*>(parent);
            if (!node) Msg::invalidNodeCast_Error();

    #else
            node = static_cast<TNode*>(parent);
    #endif
            return node->entry(path.back().parentIndex);
        }
```

Replaces the parent entry in parent node with `entry` and updates the used cache size.

```
        void replaceParentEntry(InternalEntry* entry);
```

Calls `node->insert` and updates the used cache size.

```
        bool insert(NodePtr node, EntryBase* entry);
```

Inserts a copy of `entry` into `node` and updates used cache size.

```
        bool insertCopy(NodePtr node, EntryBase* e);
```

Removes all entries from node and updates used cache size.

```
        void clear(NodePtr node);
```

Removes the i-th entry from node and updates used cache size.

```
        void remove(NodePtr node, unsigned i);
```

Calls `node->recomputeSize` and updates the used cache size.

```
        void recomputeSize(NodePtr node);
```

Removes the node from the tree file and the node cache, clears all entries in the node.

```
        void drop(NodePtr node);

    protected:
```

This struct is used to store the id of the parent node together with the respective parent entry index.

```
        struct PathEntry
        {
            PathEntry(SmiRecordId _nodeId, unsigned _parentIndex)
                    : nodeId(_nodeId), parentIndex(_parentIndex)
            {}

            SmiRecordId nodeId;
            unsigned parentIndex;
        };
```

This struct is used in the `cleanupCache` method and stores the priority of the node together with an iterator to the node position in the node cache.

```
        struct PriorityEntry
        {
            PriorityEntry(unsigned _priority, iterator _iter)
                    : priority(_priority), iter(_iter)
            {}

            unsigned priority;
            iterator iter;

            bool operator<(const PriorityEntry rhs) const
            { return priority < rhs.priority; }
        };
```

Inserts `node` into the cache, if the priority of `node` is greater than `minPriority`.

```
        void putToCache(NodePtr node);
```

Removes at least one node from the cache, if the used cache size has grown greater than `maxNode-CacheSize`. In general, many nodes would be removed in one burst, which could be done more efficient than removing only one node per time.

```
        void cleanupCache();

        NodePtr m_curNode;    // ref. to current node in path
        NodePtr m_parentNode; // ref. to parent node in path
        NodeManager* nodeSupp;   // reference to the NodeManager object
        list<PathEntry> path; // path to current node
        unsigned m_level;     // tree level of current node
        bool useCache;        // true, if cache should be used
        unsigned curSize;     // current size of the cache
        unsigned minPriority; /* maximal minimum priority of a node, that
```

```
                            has been removed from cache (only nodes
                            with a priority greater than this value
                            will be inserted into the cache) */
    unsigned m_hits, m_misses; // statistic infos
    map<SmiRecordId, NodePtr> cache; // node cache
    PriorityFun getPriority;
}; // TreeManager

} // namespace gtree
#endif // #define __GTREE_TREE_MANAGER_H__
```

2.13 Headerfile `GTree_Header.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_HEADER_H__
#define __GTREE_HEADER_H__

#include "SecondoSMI.h" // for SmiRecordId

namespace gtree
{
```

Struct *Header*

Default tree header class (provides some often needed members).

```
struct Header
{
    Header()
        : root(0), height(0), entryCount(0),
          internalCount(0), leafCount(0)
    {}

    ~Header()
    {}

    SmiRecordId root;        // page of the root node
    unsigned height;         // height of the tree
    unsigned entryCount;     // count of the entries in the tree
    unsigned internalCount; // count of Internal nodes in the tree
    unsigned leafCount;     // count of leaf nodes in the tree
};

} // namespace gtree
#endif // #define __GTREE_HEADER_H__
```

2.14 Headerfile `GTree_Tree.h`

January-May 2008, Mirko Dibbert

```
#ifndef __GTREE_TREE_H__
#define __GTREE_TREE_H__

#include <stack>
#include "GTree_Header.h"
#include "GTree_FileNode.h"
#include "GTree_TreeManager.h"

namespace gtree
{
```

Class *Tree*

This is the base class for trees.

```
template <class THeader = Header, class TTreeManager = TreeManager>
class Tree : public ObjCounter<generalTrees>
{

public:
```

Default constructor.

```
    Tree(bool temporary = false);
```

Constructor (reads a previously stored tree from file).

```
        Tree(SmiFileId fileId);
```

Default copy constructor.

```
        Tree(const Tree& tree);
```

Destructor.

```
        ~Tree();
```

Deletes the tree file.

```
        inline void deleteFile();
```

Returns the file id of the `SmiRecordFile`, that contains the tree.

```
        inline SmiFileId fileId();
```

Returns the count of all Internal nodes.

```
        inline unsigned internalCount();
```

Returns the count of all leafes.

```
        inline unsigned leafCount();
```

Returns the count of all entries, stored in the tree.

```
        inline unsigned entryCount();
```

Returns the height of the tree.

```
        inline unsigned height();
```

Returns the count of open general trees.

```
        inline unsigned openTrees();
```

Returns the count of open general tree nodes.

```
        inline unsigned openNodes();
```

Returns the count of open general tree entries.

```
        inline unsigned openEntries();
```

```
    protected:
```

Adds a new protoype node.

```
        inline void addNodePrototype(NodeBase* node);
```

Creates a new node.

```
        inline NodePtr createNode(NodeTypeId type, unsigned level = 0);
```

Creates a new node on the same level as `treeMngr->curNode()`.

```
        inline NodePtr createNeighbourNode(NodeTypeId type);
```

Creates a new node on root level.

```
        inline NodePtr createRoot(NodeTypeId type);
```

Creates a new node on root level and inserts the given entries.

```
        inline NodePtr createRoot(
                NodeTypeId type, InternalEntry *e1, InternalEntry *e2);
```

Reads the specified node from file.

```
        inline NodePtr getNode(
            SmiRecordId nodeId, unsigned level = 0) const;
```

Method initPath (initiates a new tree manager path from root)

```
        inline void initPath(SmiRecordId root, int level);
```

Method initPath (initiates a new tree manager path from header.root)

```
        inline void initPath();

    private:
```

Reads the header from `file`.

```
        void readHeader();
```

Writes the header to `file`.

```
        void writeHeader();

    protected:
        THeader header;            // contains the header of the tree file
        SmiRecordFile file;        // contais the tree file

    private:
        bool temporary;            /* true, if the file should be dropped
                                    when the tree is deleted. */
        unsigned headerPageCount;  // count of header pages

    protected:
        NodeManager* nodeMngr;     /* reference to the tree file and the
                                      node prototypes */
        TTreeManager* treeMngr;   // reference to the tree manager
    }; // class Tree


    //////////////////////////////////////////////////////////////////////
    //////////////////////////////////////////////////////////////////////
    //////////////////////////////////////////////////////////////////////
```

Tree constructor (new file):

```
template <class THeader, class TTreeManager>
gtree::Tree<THeader, TTreeManager>::Tree(bool _temporary)
        : header(), file(true, PAGESIZE), temporary(_temporary),
          nodeMngr(new NodeManager(&file)),
          treeMngr(new TTreeManager(nodeMngr))
{
#ifdef __GTREE_DEBUG
    Msg::showDbgMsg();
#endif

    file.Create();

    // create header page(s), compute headerPageCount
    SmiRecordId headerId;
    SmiRecord headerRecord;
    headerPageCount = 0;

    while (sizeof(THeader) > headerPageCount*PAGESIZE)
    {
        ++headerPageCount;
        file.AppendRecord(headerId, headerRecord);
        assert(headerId == headerPageCount);
    }
}
```

Tree constructor (reads from file):

```
template <class THeader, class TTreeManager>
gtree::Tree<THeader, TTreeManager>::Tree(SmiFileId fileId)
        : header(), file(true), temporary(false),
        nodeMngr(new NodeManager(&file)),
        treeMngr(new TTreeManager(nodeMngr))
{
#ifdef __GTREE_DEBUG
    Msg::showDbgMsg();
#endif

    assert(file.Open(fileId));

    // compute headerPageCount
    headerPageCount = 0;

    while (sizeof(THeader) > headerPageCount*PAGESIZE)
```

```
            ++headerPageCount;

        readHeader();
    }
```

Tree copy constructor:

```
    template <class THeader, class TTreeManager>
    gtree::Tree<THeader, TTreeManager>::Tree(
        const Tree<THeader, TTreeManager>& tree)
            : file(true, PAGESIZE), temporary(tree.temporary),
            nodeMngr(new NodeManager(&file)),
            treeMngr(new TTreeManager(nodeMngr))
    {
    #ifdef __GTREE_DEBUG
        Msg::showDbgMsg();
    #endif

        file.Create();

        // create header page(s), compute headerPageCount
        SmiRecordId headerId;
        SmiRecord headerRecord;
        headerPageCount = 0;

        while (sizeof(THeader) > headerPageCount*PAGESIZE)
        {
            ++headerPageCount;
            file.AppendRecord(headerId, headerRecord);
            assert(headerId == headerPageCount);
        }

        // copy header
        memcpy(&header, &tree.header, sizeof(THeader));

        nodeMngr->copyPrototypes(tree.nodeMngr);

        // disable node cache, while copying the tree structure
        treeMngr->disableCache();

        stack<pair<NodePtr, NodePtr> > remaining;
        stack<unsigned> indizes;
        unsigned curIndex = 0;

        // read tree root and create new root
        NodePtr source = tree.getNode(tree.header.root);
        NodePtr target = createNode(source->typeId());
        header.root = target->getNodeId();

        // copy tree structure (copies the tree node by node and updates
        // the chield node pointers, needs enough memory to hold
        // header.height nodes open at one time)
        if (source->isLeaf())
```

```
{
    for (unsigned i = 0; i < source->entryCount(); ++i)
        target->insertCopy(source->baseEntry(i));
}
else
{
    NodePtr chield = tree.getNode(source->chield(curIndex));
    NodePtr newChield = createNode(chield->typeId());
    target->insertCopy(source->baseEntry(curIndex));
    static_cast<InternalEntry*>(target->baseEntry(curIndex))->
    setChield(newChield);

    // push current node to stack, if
    // further entries are remaining
    if (++curIndex < source->entryCount())
    {
        remaining.push(pair<NodePtr, NodePtr>(source, target));
        indizes.push(curIndex);
    }

    // push chield node to stack
    remaining.push(pair<NodePtr, NodePtr>(chield, newChield));
    indizes.push(0);
}

while (!remaining.empty())
{
    source = remaining.top().first;
    target = remaining.top().second;
    curIndex = indizes.top();
    remaining.pop();
    indizes.pop();

    if (source->isLeaf())
    {
        for (unsigned i = 0; i < source->entryCount(); ++i)
            target->insertCopy(source->baseEntry(i));
    }
    else
    {
        NodePtr chield = tree.getNode(source->chield(curIndex));
        NodePtr newChield = createNode(chield->typeId());
        target->insertCopy(source->baseEntry(curIndex));
        static_cast<InternalEntry*>(target->
                baseEntry(curIndex))->setChield(newChield);

        // push current node to stack, if
        // further entries are remaining
        if (++curIndex < source->entryCount())
        {
            remaining.push(
                    pair<NodePtr, NodePtr>(source, target));
            indizes.push(curIndex);
            }

            // push chield node to stack
            remaining.push(
                    pair<NodePtr, NodePtr>(chield, newChield));

            indizes.push(0);
        }
    }

    treeMngr->enableCache();
} // tree copy constructor
```

Tree destructor:

```
template <class THeader, class TTreeManager>
gtree::Tree<THeader, TTreeManager>::~Tree()
{
    delete treeMngr;
    delete nodeMngr;

    if (temporary)
        deleteFile();

    if (file.IsOpen())
    {
        writeHeader();
        file.Close();
    }

#ifdef __GTREE_DEBUG
    /* print count of open objects only for the last remaining tree
      (otherwhise the warning message would appear allways, if more
      than one tree exists, e.g. when calling the copy constructor)*/
    if (openTrees() == 1)
    {
        if (openNodes())
            Msg::memoryLeak_Warning(openNodes(), "nodes");

        if (openEntries())
            Msg::memoryLeak_Warning(openEntries(), "entries");
    }
#endif
} // Tree destructor
```

Method *readHeader*:

```
template <class THeader, class TTreeManager>
void gtree::Tree<THeader, TTreeManager>::readHeader()
{
    // create buffer
    char buffer[headerPageCount*PAGESIZE];
    memset(buffer, 0, headerPageCount*PAGESIZE);
```

```
        // read header-data from file
        SmiRecord record;
        SmiRecordId curPage = 1;

        for (unsigned i = 0; i < headerPageCount; ++i)
        {
            file.SelectRecord(curPage, record, SmiFile::ReadOnly);
            record.Read(buffer + (i*PAGESIZE), PAGESIZE, 0);
            ++curPage;
        }

        // copy buffer to header
        memcpy(&header, buffer, sizeof(THeader));
    } // Tree::readHeader
```

Method *writeHeader*:

```
    template <class THeader, class TTreeManager>
    void gtree::Tree<THeader, TTreeManager>::writeHeader()
    {
        // create buffer and copy header to buffer
        char buffer[headerPageCount*PAGESIZE];
        memset(buffer, 0, headerPageCount*PAGESIZE);
        memcpy(buffer, &header, sizeof(THeader));

        // write header-data to file
        SmiRecord record;
        SmiRecordId curPage = 1;

        for (unsigned i = 0; i < headerPageCount; ++i)
        {
            file.SelectRecord(curPage, record, SmiFile::Update);
            record.Write(buffer + (i*PAGESIZE), PAGESIZE, 0);
            ++curPage;
        }
    } // Tree::writeHeader
```

Method *deleteFile*:

```
    template <class THeader, class TTreeManager>
    void gtree::Tree<THeader, TTreeManager>::deleteFile()
    {
        if (file.IsOpen())
            file.Close();

        file.Drop();
    }
```

Method *fileId*:

```
    template <class THeader, class TTreeManager>
    SmiFileId gtree::Tree<THeader, TTreeManager>::fileId()
    { return file.GetFileId(); }
```

Method *internalCount*:

```
template <class THeader, class TTreeManager>
unsigned gtree::Tree<THeader, TTreeManager>::internalCount()
{ return header.internalCount; }
```

Method *leafCount*:

```
template <class THeader, class TTreeManager>
unsigned gtree::Tree<THeader, TTreeManager>::leafCount()
{ return header.leafCount; }
```

Method *entryCount*:

```
template <class THeader, class TTreeManager>
unsigned gtree::Tree<THeader, TTreeManager>::entryCount()
{ return header.entryCount; }
```

Method *height*:

```
template <class THeader, class TTreeManager>
unsigned gtree::Tree<THeader, TTreeManager>::height()
{ return header.height; }
```

Method *openTrees*:

```
template <class THeader, class TTreeManager>
unsigned gtree::Tree<THeader, TTreeManager>::openTrees()
{ return ObjCounter<generalTrees>::openObjects(); }
```

Method *openNodes*:

```
template <class THeader, class TTreeManager>
unsigned gtree::Tree<THeader, TTreeManager>::openNodes()
{ return ObjCounter<generalTreeNodes>::openObjects(); }
```

Method *openEntries*:

```
template <class THeader, class TTreeManager>
unsigned gtree::Tree<THeader, TTreeManager>::openEntries()
{ return ObjCounter<generalTreeEntries>::openObjects(); }
```

Method *addNodePrototype*:

```
template <class THeader, class TTreeManager>
void gtree::Tree<THeader, TTreeManager>::addNodePrototype(
        NodeBase* node)
{ nodeMngr->addPrototype(node); }
```

Method *createNode*:

```
template <class THeader, class TTreeManager>
NodePtr gtree::Tree<THeader, TTreeManager>::
        createNode(NodeTypeId type, unsigned level /* = 0 */)
{
    if (level == 0)
        ++header.leafCount;
    else
        ++header.internalCount;
    return treeMngr->createNode(type, level);
}
```

Method *createNeighbourNode*:

```
template <class THeader, class TTreeManager>
NodePtr gtree::Tree<THeader, TTreeManager>::
        createNeighbourNode(NodeTypeId type)
{
    if (treeMngr->curNode()->isLeaf())
        ++header.leafCount;
    else
        ++header.internalCount;

    return treeMngr->createNeighbourNode(type);
}
```

Method *createRoot*:

```
template <class THeader, class TTreeManager>
NodePtr gtree::Tree<THeader, TTreeManager>::createRoot(
        NodeTypeId type)
{
    NodePtr root(treeMngr->createNode(type, header.height));
    header.root = root->getNodeId();
    if (header.height == 0)
    {
        ++header.leafCount;
    }
    else
    {
        ++header.internalCount;
    }
    ++header.height;

    return root;
}
```

Method *createRoot*:

```
template <class THeader, class TTreeManager>
NodePtr gtree::Tree<THeader, TTreeManager>::createRoot(
        NodeTypeId type, InternalEntry *e1, InternalEntry *e2)
{
    NodePtr root(treeMngr->createNode(type, header.height));
    treeMngr->insert(root, e1);
    treeMngr->insert(root, e2);
    header.root = root->getNodeId();
    ++header.height;
    ++header.internalCount;

    return root;
}
```

Method *getNode*:

```
    template <class THeader, class TTreeManager>
    NodePtr gtree::Tree<THeader, TTreeManager>:: getNode(
            SmiRecordId nodeId, unsigned level) const
    { return treeMngr->getNode(nodeId, level); }
```

Method initPath:

```
    template <class THeader, class TTreeManager>
    void gtree::Tree<THeader, TTreeManager>::initPath(
            SmiRecordId root, int level)
    {
        treeMngr->initPath(root, level);
    }
```

Method initPath:

```
    template <class THeader, class TTreeManager>
    void gtree::Tree<THeader, TTreeManager>::initPath()
    {
        treeMngr->initPath(header.root, header.height-1);
    }


    } // namespace gtree
    #endif // #define __GTREE_TREE_H__
```