Standard Template Library (STL)

33 Wie wird das erste Element gesucht, das nicht in einem anderen Container enthalten ist?

Wir benötigen dazu das Gegenstück zum STL-Algorithmus `find_first_of`.

find_first_not_of

```
template<class Forward1, class Forward2> inline
Forward1 find_first_not_of(Forward1 first1, Forward1 last1,
                           Forward2 first2, Forward2 last2) {
  for (; first1 != last1; ++first1) {
    for (Forward2 mid2 = first2; mid2 != last2; ++mid2)
      if (*first1 == *mid2)
        break;
    if(mid2==last2)
      return(first1);
  }
  return (first1);
}
```

Listing 91: Das Template find_first_not_of

Die Funktion finden Sie auf der CD in der `STLFunctions.h`-Datei.

Das Problem könnte auch gelöst werden, indem das Template `find_first_of` mit dem Prädikat `not_equal_to` eingesetzt würde.

34 Wie kann der Index-Operator bei Vektoren gesichert werden?

Der wahlfreie Zugriff auf Elemente eines STL-Vektors ist im Normalfall auf zwei Arten möglich: entweder mit dem Index-Operator oder mit der Methode `at`.

Der einzige Unterschied liegt darin, dass die Methode `at` den angegebenen Index auf Gültigkeit prüft und gegebenenfalls eine `out_of_range`-Ausnahme wirft, wohin-

gegen der Index-Operator ungebremst den Index verwendet und schlimmstenfalls Speicherbereiche außerhalb des Vektor-Speichers anspricht. Mit meist unangenehmen Folgen.

Es wäre doch schön, wenn der eleganter einzusetzende Index-Operator ebenfalls den Index prüfen und bei Bedarf eine Ausnahme werfen würde.

Wir implementieren dazu eine Klasse `CSaveVector`, die öffentlich von der STL-Klasse `vector` abgeleitet wird.

Klassendefinition

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CSaveVector : public std::vector<Type, Allocator> {

public:
  typedef std::vector<Type, Allocator> base_type;
  typedef CSaveVector<Type, Allocator> my_type;
  typedef base_type::size_type size_type;
  typedef base_type::reference reference;
  typedef base_type::const_reference const_reference;
};
```

Listing 92: Die Klassendefinition von CSaveVector

Konstruktoren

Durch die öffentliche Vererbung stehen in unserer Klasse alle Methoden von `vector` zur Verfügung – bis auf die Konstruktoren. Wir müssen für unsere Klasse eigene Konstruktoren implementieren und über ihre Elementinitialisierungslisten dann die Basisklassen-Konstruktoren aufrufen.

```
explicit CSaveVector(const Allocator &a=Allocator())
  : base_type(a)
{}

CSaveVector(const my_type &v)
  : base_type(v)
{}

CSaveVector(size_type n,
```

Listing 93: Die Konstruktoren von CSaveVector

```
            const Type &obj=Type(),
            const Allocator &a=Allocator())
  : base_type(n,obj,a)
{}

template<typename Input>
CSaveVector(Input begin, Input end, const Allocator &a=Allocator())
  : base_type(begin, end ,a)
{}
```

Listing 93: Die Konstruktoren von CSaveVector (Forts.)

Index-Operatoren

Die Index-Operatoren schließlich sind diejenigen, wegen denen wir den ganzen Aufwand überhaupt betreiben. Um ihren Zugriff zu sichern, benutzen wir für den Zugriff die `at`-Methode.

```
reference operator[](size_type offset) {
  return(at(offset));
}

const_reference operator[](size_type offset) const {
  return(at(offset));
}
```

Listing 94: Die Index-Operatoren von CSaveVector

Die Klasse `CSaveVector` finden Sie auf der CD in der `CSaveVector.h`-Datei.

35 Wie kann der Index-Operator bei Deques gesichert werden?

Motivation dieser Frage und Vorgehensweise der Lösung sind nahezu identisch mit der Klasse `CSaveVector` aus Rezept 34.

Wir implementieren dazu eine Klasse `CSaveDeque`, die öffentlich von der STL-Klasse `deque` abgeleitet wird.

Klassendefinition

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CSaveDeque : public std::deque<Type, Allocator> {

public:
  typedef std::deque<Type, Allocator> base_type;
  typedef CSaveDeque<Type, Allocator> my_type;
  typedef base_type::size_type size_type;
  typedef base_type::reference reference;
  typedef base_type::const_reference const_reference;
};
```

Listing 95: Die Klassendefinition von CSaveDeque

Konstruktoren

Die Basisklassen-Konstruktoren müssen über die Elementinitialisierungslisten der eigenen Konstruktoren aufgerufen werden.

```
explicit CSaveDeque(const Allocator &a=Allocator())
  : base_type(a)
{}

CSaveDeque(const my_type &v)
  : base_type(v)
{}

CSaveDeque(size_type n,
           const Type &obj=Type(),
           const Allocator &a=Allocator())
  : base_type(n,obj,a)
{}

template<typename Input>
CSaveDeque(Input begin, Input end, const Allocator &a=Allocator())
  : base_type(begin, end ,a)
{}
```

Listing 96: Die Konstruktoren von CSaveDeque

Index-Operatoren

Zum Schluss fehlen noch die Index-Operatoren, die ihre Arbeit mit Hilfe der `at`-Methode verrichten.

```
reference operator[](size_type offset) {
  return(at(offset));
}

const_reference operator[](size_type offset) const {
  return(at(offset));
}
```

Listing 97: Die Index-Operatoren von CSaveDeque

Die Klasse `CSaveDeque` finden Sie auf der CD in der *CSaveDeque.h*-Datei.

36 Wie können die find-Algorithmen ohne Beachtung der Groß-/Kleinschreibung eingesetzt werden?

Will man mit Hilfe von `find` ein Zeichen in einem Container finden, dann sieht das im Normalfall so aus[1]:

```
string s1("Dies ist ein Test im Glas");
string::iterator i=find(s1.begin(), s1.end(), 'T');
cout << "Gefunden: " << *i << endl;
```

Das obere Codestück findet den Buchstaben »T« im Wort »Test« und nicht das »t« aus »ist«, weil zwischen Groß- und Kleinschreibung unterschieden wird.

CCharCIPredicate

Wie ist nun aber vorzugehen, wenn es keine Rolle spielen soll, ob nun ein »t« oder ein »T« gefunden wird?

1. In diesem konkreten Fall hätte auch die `find`-Methode von `string` zum Einsatz kommen können. Aber wir wollen die Betrachtung allgemeingültiger halten.

Dazu implementieren wir uns das Prädikat `CCharCIPredicate`, welches die Unterscheidung zwischen Groß- und Kleinschreibung für uns ignoriert. Um konform mit den Prädikaten der STL zu gehen, wird es von `unary_function` angeleitet:

```
class CCharCIPredicate : public std::unary_function<char, bool> {
private:
  char m_char;
public:
  CCharCIPredicate(char c)
    : m_char(tolower(c))
  {}
  bool operator()(char o) {
    return(m_char==tolower(o));
  }
};
```

Listing 98: Das Prädikat CCharCIPredicate

Das Prädikat kann nun über `find_if` eingesetzt werden:

```
  string s1("Dies ist ein Test im Glas");
  string::iterator i=find_if(s1.begin(),
                             s1.end(),
                             CCharCIPredicate('T'));
  cout << "Gefunden: " << *i << endl;
```

Nun wird das kleine »t« aus »ist« gefunden, weil das Prädikat die Groß-/Kleinschreibung ignoriert.

CCharCIBinPredicate

Die gleiche Funktionalität wollen wir nun auch beim Vergleich von Sequenzen zur Verfügung stellen.

Beim Vergleich von Sequenzen brauchen wir ein binäres Prädikat. Es wird aus Kompatibilitätsgründen von `binary_function` abgeleitet:

```
class CCharCIBinPredicate
: public std::binary_function<char, char, bool> {
```

Listing 99: Das binäre Prädikat CCharCIBinPredicate

```
public:
  bool operator()(char o1, char o2) {
    return(tolower(o1)==tolower(o2));
  }
};
```

Listing 99: Das binäre Prädikat CCharCIBinPredicate (Forts.)

Nun lassen sich auch Sequenzen ohne Berücksichtigung der Groß- und Kleinschreibung vergleichen:

```
  string s1("Dies ist ein Test im Glas");
  string s2("tEsT");
  string::iterator i=search(s1.begin(),
                            s1.end(),
                            s2.begin(),
                            s2.end(),
                            CCharCIBinPredicate());
  cout << "Gefunden an Position: " <<
          distance(s1.begin(),i) << endl;
```

Es wird Position 13 ausgegeben.

Die beiden Prädikate finden Sie auf der CD in der *CCharCIPredicate.h*-Datei.

37 Wie kann eine Matrix implementiert werden?

Es gibt viele mögliche Ansätze, eine Matrix zu implementieren. Grundsätzlich sollte eine vernünftige Implementierung auf die Container der STL zurückgreifen, um das Rad nicht neu erfinden zu müssen.

Wenn man versucht, in Begriffen der Objektorientierten Programmierung zu denken, könnte man auf den Ansatz kommen, dass eine Matrix für einen bestimmten Datentyp nichts anderes ist als ein Vektor von Vektoren des Datentyps. Also in etwa wie folgt:

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrix : public std::vector<std::vector<Type> > {
};
```

Dieser Ansatz hat den Vorteil, dass die übliche Index-Schreibweise, wie sie in C++ bei mehrdimensionalen Feldern üblich ist (z.B. `f[2][3]`), verwendet werden kann.

So elegant dieser Ansatz auf den ersten Blick erscheinen mag, hat er jedoch den dramatischen Nachteil, dass die interne Struktur der Matrix nicht gekapselt ist und somit von außen problemlos in einen inkonsistenten Zustand gebracht werden kann. Ein Beispiel:

```
CMatrix2<int> m(5,5);
m[0].clear();
```

Wenn wir davon ausgehen, dass die Matrix mit einem Konstruktor ausgestattet wurde, dem die Größe der Matrix übergeben wird, dann haben wir mit der ersten Zeile eine 5*5-Matrix erzeugt.

Nun ist es möglich, lediglich einen Index zu verwenden und damit alle Methoden der `vector`-Klasse anzusprechen. In diesem Fall wurde der Vektor der ersten Zeile (oder Spalte, je nach interner Implementierung) gelöscht. Die Matrix ist damit nicht mehr funktionstüchtig.

Es sollte also ein Ansatz gewählt werden, bei dem der Benutzer nicht in die interne Struktur eingreifen kann. Andererseits sollte aber der Vorteil der Index-Schreibweise beibehalten werden.

CMatrixVector

Wir werden daher speziell für die Matrix eine Vektor-Klasse erstellen, die nur die Implementierung des üblichen STL-Vektors übernimmt, die Schnittstelle aber bis auf die für den Benutzer wesentlichen Teile (Index-Operator) kapselt.

Klassendefinition

Die Klasse `CMatrixVector` wird später von der Klasse `CMatrix` eingesetzt. Damit die Matrix selbst vollen Zugriff auf die gekapselten Methoden besitzt, wird sie als Freund von `CMatrixVector` deklariert. Dazu muss vorher die spätere Matrix-Klasse deklariert werden.

Wir leiten `CMatrixVector` von der Klasse `CSaveVector` aus Rezept 34 ab, damit ein sicherer Zugriff über den Index-Operator gewährleistet ist. Bei Bedarf kann natürlich auch der übliche STL-Vektor genommen werden.

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrix;

template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrixVector : private CSaveVector<Type, Allocator> {
  friend class CMatrix;
public:
  typedef CSaveVector<Type, Allocator> base_type;
  typedef CMatrixVector<Type, Allocator> my_type;
  typedef base_type::size_type size_type;
  typedef base_type::reference reference;
  typedef base_type::const_reference const_reference;
};
```

Listing 100: Die Klassendefinition von CMatrixVector

Konstruktoren

Von den Konstruktoren der Basisklasse werden lediglich zwei »durchgeschliffen«.

```
explicit CMatrixVector(const Allocator &a=Allocator())
  : base_type(a)
{}

//*********

CMatrixVector(size_type n,
              const Type &obj=Type(),
              const Allocator &a=Allocator())
  : base_type(n,obj,a)
{}
```

Listing 101: Die Konstruktoren von CMatrixVector

Index-Operatoren

Damit der Index-Operator für Außenstehende einsetzbar ist, muss eine öffentliche Variante existieren.

```
reference operator[](size_type idx) {
  return(base_type::operator[](idx));
```

Listing 102: Die Index-Operatoren von CMatrixVector

```
}

//*********

const_reference operator[](size_type idx) const {
  return(base_type::operator[](idx));
}
```

Listing 102: Die Index-Operatoren von CMatrixVector (Forts.)

CMatrix

Die Klasse `CMatrix` besitzt als wichtigstes Attribut einen `CMatrixVector` vom Typ `CMatrixVector<Type>`. Dadurch können wir später die gewohnte Index-Schreibweise einsetzen, haben aber die restlichen Methoden des Vektors in `CMatrixVector` gekapselt und nur für `CMatrix` zugänglich gemacht.

Zusätzlich definieren wir noch die Attribute `m_rows` und `m_columns` für die Dimensionen der Matrix.

Die Dimensionen der Matrix lassen sich über Zugriffsmethoden ermitteln. Darüber hinaus wird noch die Methode `size` implementiert, die die Anzahl der Elemente in der Matrix liefert.

Klassendefinition

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrix {
public:
  typedef CMatrix<Type, Allocator> my_type;
  typedef std::size_t size_type;
  typedef Type& reference;
  typedef const Type& const_reference;

private:
  CMatrixVector<CMatrixVector<Type, Allocator>, Allocator>
    m_elements;
  size_type m_rows;
  size_type m_columns;
```

Listing 103: Klassendefinition und Zugriffsmethoden von CMatrix

```
public:
  size_type getRows() const {
    return(m_rows);
  }

//*********

  size_type getColumns() const {
    return(m_columns);
  }

//*********

  size_type size() const {
    return(m_rows*m_columns);
  }
};
```

Listing 103: Klassendefinition und Zugriffsmethoden von CMatrix (Forts.)

Konstruktoren

Die Matrix erhält einen Standard-Konstruktor, einen Kopier-Konstruktor und einen Konstruktor, mit dem eine Matrix dimensioniert und optional alle Werte mit einem Wert initialisiert werden können.

```
CMatrix()
: m_rows(0), m_columns(0)
{}

//*********

CMatrix(const CMatrix &m)
: m_elements(m.m_elements), m_rows(m.m_rows), m_columns(m.m_columns)
{ }

//*********

CMatrix(size_type rows, size_type columns, const Type &obj=Type()) {
  resize(rows,columns,obj);
}
```

Listing 104: Einige Konstruktoren von CMatrix

Zusätzlich wird ein Konstruktor implementiert, der die Matrix aus einem Teil einer anderen Matrix erzeugt. Dazu werden die Position innerhalb der Matrix und die gewünschte Größe angegeben:

```
CMatrix(const CMatrix &m,
        size_type rbegin, size_type cbegin,
        size_type rows, size_type columns) {
  resize(rows,columns);
  for(size_type r=0; r<rows; ++r)
    for(size_type c=0; c<columns; ++c)
      m_elements[r][c]=m.m_elements[r+rbegin][c+cbegin];
}
```

Listing 105: Ein Konstruktor für Bereiche einer Matrix

Ferner wird noch ein Konstruktor hinzugefügt, der über zwei Iteratoren eine einspaltige bzw. einzeilige Matrix erzeugt:

```
template<typename IType>
CMatrix(bool column, IType beg, IType end) {
  if(column) {
    resize(distance(beg,end),1);
    copy(beg,end,column_begin());
  }
  else {
    resize(1, distance(beg,end));
    copy(beg,end,row_begin());
  }
}
```

Listing 106: Ein Konstruktor für einzeilige/-spaltige Matrizen

Zuweisungsoperator

```
CMatrix &operator=(const CMatrix &m) {
  if(this!=&m) {
    m_elements=m.m_elements;
    m_rows=m.m_rows;
```

Listing 107: Der Zuweisungsoperator von CMatrix

```
    m_columns=m.m_columns;
  }
  return(*this);
}
```

Listing 107: Der Zuweisungsoperator von CMatrix (Forts.)

Index-Operatoren

Die Index-Operatoren von `CMatrix` rufen die Index-Operatoren von `CMatrixVector` auf.

```
CMatrixVector<Type, Allocator> &operator[](size_type idx) {
  return(m_elements[idx]);
}

//*********

const CMatrixVector<Type, Allocator>
&operator[](size_type idx) const {
  return(m_elements[idx]);
}
```

Listing 108: Die Index-Operatoren von CMatrix

Aufruf-Operatoren

Wir wollen die Klasse auch mit Aufruf-Operatoren versehen, um eine alternative Schreibweise für den Elementzugriff zu ermöglichen. Obwohl der Aufruf-Operator bei der gewählten internen Struktur keine Geschwindigkeitsvorteile bringt, kann er insbesondere bei Implementierungen für dünn besetzte Matrizen von Vorteil sein.

```
Type &operator()(size_type r, size_type c) {
  return(m_elements.at(r).at(c));
}

//*********

const Type &operator()(size_type r, size_type c) const {
```

Listing 109: Die Aufruf-Operatoren von CMatrix

```
  return(m_elements.at(r).at(c));
}
```

Listing 109: Die Aufruf-Operatoren von CMatrix (Forts.)

resize

Mit der öffentlichen Methode `resize` kann die Größe der Matrix angepasst werden. Einige der Konstruktoren machen von ihr Gebrauch.

```
void resize(size_type rows,
            size_type columns,
            const Type &obj=Type()) {

/*
** Größe des Zeilenvektors anpassen
*/
  m_elements.resize(rows, CMatrixVector<Type,
                    Allocator>(columns,obj) );

/*
** Größe der Spaltenvektoren anpassen
*/
  for(size_type r=0; r<rows; ++r)
    m_elements[r].resize(columns,obj);
  m_columns=columns;
  m_rows=rows;
}
```

Listing 110: Die Methode resize von CMatrix

output

Damit die Matrix einigermaßen strukturiert ausgegeben werden kann, fügen wir die Methode `output` hinzu, die eine maximale Elementbreite für die Ausgabe übergeben bekommt und die Matrix als String zurückliefert.

```
std::string output(int width) const{
  ostringstream os;
  for(size_type y=0; y<m_rows; ++y) {
```

Listing 111: Die Methode output von CMatrix

```
    for(size_type x=0; x<m_columns; ++x)
      os << std::setw(width) << (*this)[y][x] << " ";
    os << "\n";
  }
  return(os.str());
}
```

Listing 111: Die Methode output von CMatrix (Forts.)

Stream-Operatoren

Damit ein Objekt vom Typ `CMatrix` auch mit den binären Strömen aus Rezept 106 zusammenarbeiten kann, überladen wir hier noch die Stream-Operatoren:

```
friend CBinaryOStream &operator<<(CBinaryOStream &os,
                                  const CMatrix &m){
  os << m.m_rows;
  os << m.m_columns;
  const_row_iterator iter=m.row_begin();
  while(iter!=m.row_end())
    os << *(iter++);
  return(os);
}

friend CBinaryIStream &operator>>(CBinaryIStream &is, CMatrix &m){
  is >> m.m_rows;
  is >> m.m_columns;
  m.resize(m.m_rows,m.m_columns);
  row_iterator iter=m.row_begin();
  while(iter!=m.row_end())
    is >> *(iter++);
  return(is);
}
```

Listing 112: Die Stream-Operatoren von CMatrix

row_iterator

Damit wir mit der Matrix-Klasse elegant arbeiten können, ergänzen wir sie um Iteratoren. Die Frage ist nur, wie ein Iterator über die einzelnen Elemente der Matrix iterieren sollte.

In manchen Fällen macht es Sinn, die Matrix zeilenweise zu durchlaufen, in bestimmten Situationen kann aber auch eine spaltenweise Iteration vonnöten sein.

Wir werden deswegen zwei verschiedene Iteratoren programmieren, einen für die zeilenweise Iteration und einen für das spaltenweise Durchlaufen.

Beginnen wir mit `row_iterator`, dem Iterator für die zeilenweise Iteration.

Als Attribute benötigen wir einen Zeiger auf die dazugehörige Matrix und die Koordinaten des aktuellen Elementes.

Klassendefinition

```
class row_iterator;
friend class row_iterator;
class row_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
  size_type m_c;
  CMatrix *m_matrix;
};
```

Listing 113: Die Klassendefinition von row_iterator

Konstruktoren

Es gibt einen Standard-Konstruktor und einen Konstruktor, der das Iterator-Objekt mit den Koordinaten in einer bestimmten Matrix verknüpft.

```
row_iterator()
  :m_matrix(0), m_r(0), m_c(0)
{}

//*********

row_iterator(CMatrix &matrix, size_type r, size_type c)
  : m_matrix(&matrix), m_r(r), m_c(c)
{}
```

Listing 114: Die Konstruktoren von row_iterator

Inkrement-/Dekrement-Operatoren

Die Inkrement- und Dekrement-Operatoren sind das Herzstück eines bidirektionalen Iterators.

Die Post-Operatoren greifen auf die Funktionalität der Prä-Operatoren zurück.

```
row_iterator &operator++() {

/*
** Ende-Position erreicht?
** Ende-Position beibehalten
*/
  if(m_r==m_matrix->m_rows)
    return(*this);

/*
** Nächstes Element der Zeile
*/
  ++m_c;

/*
** Zeile zu Ende?
** => Auf erstes Element der nächsten Zeile setzen
*/
  if(m_c==m_matrix->m_columns) {
    m_c=0;
    ++m_r;
  }
  return(*this);
}

//*********

row_iterator &operator--() {

/*
** Anfangs-Position erreicht?
** Anfangs-Position beibehalten
*/
  if((m_c==0)&&(m_r==0))
    return(*this);
```

Listing 115: Die Inkrement-/Dekrement-Operatoren von row_iterator

```
/*
** Aktuelles Element erstes Element der Zeile?
** => Auf letztes Element der vorhergehenden Zeile setzen
*/
  if(m_c==0) {
    m_c=m_matrix->m_columns-1;
    --m_r;
  }

/*
** Andernfalls auf vorhergehendes Element der Zeile setzen
*/
  else
    --m_c;
  return(*this);
}

//*********

row_iterator operator++(int) {
  row_iterator tmp=*this;
  ++(*this);
  return(tmp);
}

//*********

row_iterator operator--(int) {
  row_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 115: Die Inkrement-/Dekrement-Operatoren von row_iterator (Forts.)

Zugriffsoperatoren

Zum Schluss fehlen noch die Operatoren, mit denen das mit dem Iterator verknüpfte Element der Matrix angesprochen werden kann.

```
Type &operator*() {
  return((*m_matrix)[m_r][m_c]);
}

//*********

Type *operator->() {
  return(&(*m_matrix)[m_r][m_c]);
}
```

Listing 116: Die Zugriffsoperatoren von row_iterator

Vergleichsoperatoren

Bidirektionale Iteratoren müssen die Vergleichsoperationen == und != unterstützen:

```
bool operator==(row_iterator &i) const {
  return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
}

//*********

bool operator!=(row_iterator &i) const {
  return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
}
```

Listing 117: Die Vergleichsoperatoren von row_iterator

column_iterator

Der `column_iterator` ist so konzipiert, dass er die Matrix spaltenweise durchläuft.

Klassendefinition

```
class column_iterator;
friend class column_iterator;
class column_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
```

Listing 118: Die Klassendefinition von column_iterator

```
  size_type m_c;
  CMatrix *m_matrix;
};
```

Listing 118: Die Klassendefinition von column_iterator (Forts.)

Konstruktoren

Die Konstruktoren werden analog zu denen von `row_iterator` implementiert.

```
column_iterator()
  :m_matrix(0), m_r(0), m_c(0)
{}

//*********

column_iterator(CMatrix &matrix, size_type r, size_type c)
  : m_matrix(&matrix), m_r(r), m_c(c)
{}
```

Listing 119: Die Konstruktoren von column_iterator

Inkrement-/Dekrement-Operatoren

Die Inkrement- und Dekrement-Operatoren von `column_iterator` sorgen dafür, dass der Iterator spaltenweise über die Matrix wandert.

```
column_iterator &operator++() {

/*
** Ende-Position erreicht?
** Ende-Position beibehalten
*/
  if(m_c==m_matrix->m_columns)
    return(*this);

/*
** Nächstes Element der Spalte
*/
  ++m_r;
```

Listing 120: Die Inkrement-/Dekrement-Operatoren von column_iterator

```
/*
** Spalte zu Ende?
** => Auf erstes Element der nächsten Spalte setzen
*/
  if(m_r==m_matrix->m_rows) {
    m_r=0;
    ++m_c;
  }
  return(*this);
}

//*********

column_iterator &operator--() {

/*
** Anfangs-Position erreicht?
** Anfangs-Position beibehalten
*/
  if((m_c==0)&&(m_r==0))
    return(*this);

/*
** Ist aktuelles Element erstes Element der Spalte?
** => Auf letztes Element der vorhergehenden Spalte setzen
*/
  if(m_r==0) {
    m_r=m_matrix->m_rows-1;
    --m_c;
  }

/*
** Andernfalls auf vorhergehendes Element der Spalte setzen
*/
  else
    --m_r;
  return(*this);
}

//*********
```

Listing 120: Die Inkrement-/Dekrement-Operatoren von column_iterator (Forts.)

```
column_iterator operator++(int) {
  column_iterator tmp=*this;
  ++(*this);
  return(tmp);
}

//*********

column_iterator operator--(int) {
  column_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 120: Die Inkrement-/Dekrement-Operatoren von column_iterator (Forts.)

Zugriffsoperatoren

```
Type &operator*() {
  return((*m_matrix)[m_r][m_c]);
}

//*********

Type *operator->() {
  return(&(*m_matrix)[m_r][m_c]);
}
```

Listing 121: Die Zugriffsoperatoren von column_iterator

Vergleichsoperatoren

```
bool operator==(column_iterator &i) const {
  return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
}

//*********
```

Listing 122: Die Vergleichsoperatoren von column_iterator

```
bool operator!=(column_iterator &i) const {
  return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
}
```

Listing 122: Die Vergleichsoperatoren von column_iterator (Forts.)

const-Iteratoren

Damit die Iteratoren auch mit konstanten Matrizen arbeiten können, benötigen wir `row_iterator` und `column_iterator` zusätzlich für konstante Objekte.

Wir nennen sie `const_row_iterator` und `const_column_iterator`. Der Vollständigkeit halber sind sie im Folgenden aufgeführt.

```
//*****************************************************************
// const_row_iterator
//*****************************************************************

class const_row_iterator;
friend class const_row_iterator;
class const_row_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
  size_type m_c;
  const CMatrix *m_matrix;
public:
  const_row_iterator()
    :m_matrix(0), m_r(0), m_c(0)
  {}

//*********

  const_row_iterator(const CMatrix &matrix,
                     size_type r,
                     size_type c)
    : m_matrix(&matrix), m_r(r), m_c(c)
  {}

//*********
```

Listing 123: Die const-Iteratoren von CMatrix

```
  const_row_iterator &operator++() {
    if(m_r==m_matrix->m_rows)
      return(*this);
    ++m_c;
    if(m_c==m_matrix->m_columns) {
      m_c=0;
      ++m_r;
    }
    return(*this);
  }

//*********

  const_row_iterator &operator--() {
    if((m_c==0)&&(m_r==0))
      return(*this);
    if(m_c==0) {
      m_c=m_matrix->m_columns-1;
      --m_r;
    }
    else
      --m_c;
    return(*this);
  }

//*********

  const_row_iterator operator++(int) {
    const_row_iterator tmp=*this;
    ++(*this);
    return(tmp);
  }

//*********

  const_row_iterator operator--(int) {
    const_row_iterator tmp=*this;
    --(*this);
    return(tmp);
  }
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

```
//*********

  const Type &operator*() {
    return((*m_matrix)[m_r][m_c]);
  }

//*********

  const Type *operator->() {
    return(&(*m_matrix)[m_r][m_c]);
  }

//*********

  bool operator==(const_row_iterator &i) const {
    return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
  }

//*********

  bool operator!=(const_row_iterator &i) const {
    return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
  }
};


//****************************************************************
// const_column_iterator
//****************************************************************

class const_column_iterator;
friend class const_column_iterator;
class const_column_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
  size_type m_c;
  const CMatrix *m_matrix;
public:
  const_column_iterator()
    :m_matrix(0), m_r(0), m_c(0)
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

```
  {}

//*********

  const_column_iterator(const CMatrix &matrix,
                        size_type r,
                        size_type c)
    : m_matrix(&matrix), m_r(r), m_c(c)
  {}

//*********

  const_column_iterator &operator++() {
    if(m_c==m_matrix->m_columns)
      return(*this);
    ++m_r;
    if(m_r==m_matrix->m_rows) {
      m_r=0;
      ++m_c;
    }
    return(*this);
  }

//*********

  const_column_iterator &operator--() {
    if((m_c==0)&&(m_r==0))
      return(*this);
    if(m_r==0) {
      m_r=m_matrix->m_rows-1;
      --m_c;
    }
    else
      --m_r;
    return(*this);
  }

//*********

  const_column_iterator operator++(int) {
    const_column_iterator tmp=*this;
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

```
    ++(*this);
    return(tmp);
  }

//*********

  const_column_iterator operator--(int) {
    const_column_iterator tmp=*this;
    --(*this);
    return(tmp);
  }

//*********

  const Type &operator*() {
    return((*m_matrix)[m_r][m_c]);
  }

//*********

  const Type *operator->() {
    return(&(*m_matrix)[m_r][m_c]);
  }

//*********

  bool operator==(const_column_iterator &i) const {
    return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
  }

//*********

  bool operator!=(const_column_iterator &i) const {
    return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
  }
};
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

Iterator-Methoden

Damit zu einer Matrix auch Iteratoren erzeugt werden können, benötigen wir die obligatorischen `begin`- und `end`-Methoden.

Wir wollen aber auch hier einen Schritt weiter gehen als von der STL vorgelebt und zusätzliche Methoden implementieren, die Iteratoren für bestimmte Spalten erzeugen.

Ein Beispiel finden Sie im Anschluss an die Methoden.

```
/*
** begin & end für row_iterator
*/

row_iterator row_begin() {
  return(row_iterator(*this,0,0));
}

//*********

row_iterator row_end() {
  return(row_iterator(*this,m_rows,0));
}

//*********

row_iterator row_begin(size_type r) {
  return(row_iterator(*this,r,0));
}

//*********

row_iterator row_end(size_type r) {
  return(row_iterator(*this,r+1,0));
}

/*
** begin & end für const_row_iterator
*/

const_row_iterator row_begin() const {
  return(const_row_iterator(*this,0,0));
}

//*********
```

Listing 124: Die Iterator-Methoden von CMatrix

```
const_row_iterator row_end() const {
  return(const_row_iterator(*this,m_rows,0));
}

//*********

const_row_iterator row_begin(size_type r) const {
  return(const_row_iterator(*this,r,0));
}

//*********

const_row_iterator row_end(size_type r) const {
  return(const_row_iterator(*this,r+1,0));
}

/*
** begin & end für column_iterator
*/

column_iterator column_begin() {
  return(column_iterator(*this,0,0));
}

//*********

column_iterator column_end() {
  return(column_iterator(*this,0,m_columns));
}

//*********

column_iterator column_begin(size_type c) {
  return(column_iterator(*this,0,c));
}

//*********

column_iterator column_end(size_type c) {
  return(column_iterator(*this,0,c+1));
```

Listing 124: Die Iterator-Methoden von CMatrix (Forts.)

```
}

/*
** begin & end für const_column_iterator
*/

const_column_iterator column_begin() const {
  return(const_column_iterator(*this,0,0));
}

//*********

const_column_iterator column_end() const {
  return(const_column_iterator(*this,0,m_columns));
}

//*********

const_column_iterator column_begin(size_type c) const {
  return(const_column_iterator(*this,0,c));
}

//*********

const_column_iterator column_end(size_type c) const {
  return(const_column_iterator(*this,0,c+1));
}
```

Listing 124: Die Iterator-Methoden von CMatrix (Forts.)

Sie finden die Klasse `CMatrix` auf der CD in der `CMatrix.h`-Datei.

Wollen wir beispielsweise aus einer 5*5-Matrix `oMatrix` die zweite Spalte in einen Vektor `oVector` kopieren, dann sieht das wie folgt aus:

```
CMatrix<int> oMatrix(5,5);
vector<int> oVector;
copy(oMatrix.column_begin(1),
     oMatrix.column_end(1),
     back_inserter(oVector));
```

Wollten wir die dritte und vierte Zeile in den Vektor kopieren, geschähe dies so:

```
copy(oMatrix.row_begin(2),
     oMatrix.row_end(3),
     back_inserter(oVector));
```


38 Wie kann ein binärer Suchbaum implementiert werden?

Im Gegensatz zu einer linearen Liste, bei der jeder Knoten nur einen Nachfolger haben kann, zeichnet sich ein Suchbaum dadurch aus, dass ein Knoten maximal zwei Nachfolger besitzen darf. Abbildung 26 zeigt ein Beispiel eines Baums, der darüber hinaus noch vollständig ist.

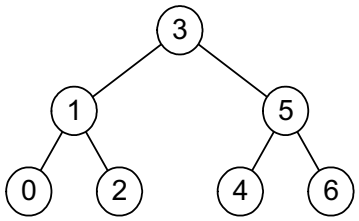

Abbildung 26: Ein Suchbaum

Die Besonderheit solcher Bäume liegt in der Sortierung eines Baums. In unserem Fall gilt die Regel, dass der linke Teilbaum eines Knotens nur Knoten enthält, deren Wert kleiner ist als der Vater des Teilbaums.

Einige STL-Container basieren auf Bäumen (z. B. `set`, `multiset`, `map`, `multimap`), allerdings handelt es sich dabei um ausgeglichene Bäume, denn bei gleicher Knotenanzahl ist der Baum mit der geringsten Höhe für Such-Operationen am effizientesten.

In Abbildung 27 sehen wir einen Baum, dessen Gesamthöhe durch eine Umstrukturierung (in diesem Fall eine Links-Rotation) um eins vermindert wird.

Allerdings geht dabei die ursprüngliche Baumstruktur verloren. Deswegen erzeugt die folgende Implementierung einen Baum, dessen Struktur nicht aus Effizienz-Gründen verändert wird.

Die Implementierung eines höhenbalancierten Baumes finden Sie in Rezept 43.

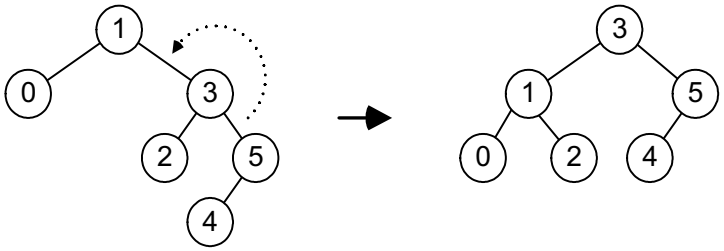

Abbildung 27: Höhenausgleichende Maßnahmen bei Bäumen

CKnot

Um einen einzelnen Knoten des Baumes repräsentieren zu können, legen wir eine Klasse `CKnot` an. Ein `CKnot`-Objekt besitzt Zeiger auf die beiden Söhne und den Vater. Zusätzlich beinhaltet das Objekt mit `m_data` das zu verwaltende Datenobjekt.

Abbildung 28 stellt die Zeigerstruktur grafisch dar.

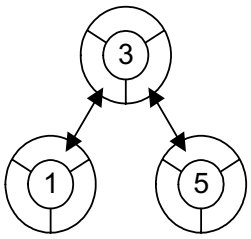

Abbildung 28: Die Zeigerstruktur der CKnot-Objekte

```
class CKnot {
public:
  CKnot *m_father;
  CKnot *m_left;
  CKnot *m_right;
  value_type m_data;

//*********

  CKnot(CKnot *fa, CKnot *li, CKnot *re)
  :m_father(fa),m_left(li),m_right(re)
  { }
```

Listing 125: Die Klasse CKnot

```
//*********

  CKnot(CKnot *fa, CKnot *li, CKnot *re, const value_type &v)
  :m_father(fa),m_left(li),m_right(re),m_data(v)
  {}

//*********

  virtual ~CKnot()
  {}
};
```

Listing 125: Die Klasse CKnot (Forts.)

CTree

Die Klasse `CTree` übernimmt die Verwaltung der Baumstruktur.

Sie wird als Template definiert, wobei ein `CTree`-Objekt an ein Objekt gebunden wird, das die Eigenschaften der zu verwaltenden Daten spezifiziert. Diese `Traits` besprechen wir später bei den konkreten Klassen `CSetTree` und `CMapTree`.

Klassendefinition

Abgesehen von den einzelnen Typendeklarationen, die zum größten Teil aus dem `Traits`-Objekt übernommen werden, besitzt die Klasse ein Attribut für die Wurzel des Baums und eins für die Anzahl der Objekte im Baum.

Die trivialen Methoden `size` und `empty` sind bei der Klassendefinition gleich mit aufgeführt.

```
template<typename Traits>
class CTree {
public:
  typedef CTree<Traits> my_type;
  typedef typename Traits::value_type value_type;
  typedef typename Traits::key_type key_type;
  typedef unsigned long size_type;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
```

Listing 126: Die Klassendefinition von CTree

```
protected:
  class CKnot { /* … */ };

  CKnot *m_root;
  size_type m_size;

//*********

  size_type size() const  {
    return(m_size);
  }

//*********

  bool empty() const {
    return(m_size==0);
  }
};
```

Listing 126: Die Klassendefinition von CTree (Forts.)

Konstruktoren

Der einfachste Konstruktor – der Standard-Konstruktor – erzeugt einen leeren Baum:

```
CTree()
: m_root(0), m_size(0)
{}
```

Listing 127: Der Standard-Konstruktor von CTree

Der Kopier-Konstruktor kopiert einen Baum durch Einsatz der `operator=`-Methode:

```
CTree(const CTree &t)
: m_root(0), m_size(0) {
  *this=t;
}
```

Listing 128: Der Kopier-Konstruktor von CTree

Der aufwändigste Konstruktor erzeugt den Baum aus Objekten eines Bereichs, der mit Iteratoren definiert wird. Für das Einfügen der Objekte in den Baum wird die `insert`-Methode benutzt:

```
template<typename Input>
CTree(Input beg, Input end)
: m_root(0), m_size(0) {
  insert(beg, end);
}
```

Listing 129: Ein weiterer Konstruktor für CTree

Destruktor

Der Destruktor löscht alle Knoten des Baums durch Aufruf der `deleteKnot`-Methode für die Wurzel des Baums:

```
virtual ~CTree() {
  if(m_root)
    deleteKnot(m_root);
}
```

Listing 130: Der Destruktor von CTree

deleteKnot

Die geschützte Methode `deleteKnot` löscht einen Knoten mitsamt der eventuell daran hängenden Teilbäume:

```
void deleteKnot(CKnot *kn) {
  if(kn) {
    if(kn->m_left) deleteKnot(kn->m_left);
    if(kn->m_right) deleteKnot(kn->m_right);
    delete(kn);
  }
}
```

Listing 131: Die geschützte Methode deleteKnot von CTree

Zuweisungsoperator

Der Zuweisungsoperator kopiert einen Baum, indem er zuerst die Wurzel kopiert und die weitere Arbeit dann der Methode `copySons` überlässt:

```
CTree &operator=(const CTree &t) {
  if(&t!=this) {

/*
** Alle Knoten des Baums löschen
*/
    deleteKnot(m_root);
    m_size=t.m_size;

/*
** Besitzt Quellbaum Knoten?
** = Wurzelknoten kopieren und alle weiteren Knoten
** durch Aufruf von copySons kopieren
*/
    if(t.m_root) {
      m_root=new CKnot(0,0,0,t.m_root->m_data);
      copySons(m_root, t.m_root);
    }
  }
  return(*this);
}
```

Listing 132: Der Zuweisungsoperator von CTree

copySons

Die private Methode `copySons` kopiert die Söhne eines Knotens und ruft für sie `copySons` erneut auf:

```
void copySons(CKnot* d, CKnot *s) {
  if(d&&s) {

/*
** Linken Sohn kopieren, falls vorhanden, und
** für ihn copySons rekursiv aufrufen
*/
```

Listing 133: Die private Methode copySons von CTree

```
    if(s->m_left) {
      d->m_left=new CKnot(d,0,0,s->m_left->m_data);
      copySons(d->m_left,s->m_left);
    }

/*
** Rechten Sohn kopieren, falls vorhanden, und
** für ihn copySons rekursiv aufrufen
*/
    if(s->m_right) {
      d->m_right=new CKnot(d,0,0,s->m_right->m_data);
      copySons(d->m_right,s->m_right);
    }
  }
}
```

Listing 133: Die private Methode copySons von CTree (Forts.)

insert

Mit den verschiedenen `insert`-Methoden können Objekte in den Baum eingefügt werden.

Die einfachste Variante fügt ein Objekt in einen Baum ein und liefert die Position im Baum als Iterator-Position zurück. Sie benutzt dazu die `insert`-Methode zum Einfügen von Knoten.

Die Iteratoren werden in den nächsten Rezepten besprochen.

```
virtual inorder_iterator insert(const value_type &v) {

/*
** Datenobjekt in einen Knoten packen
*/
   CKnot *kn=new CKnot(0,0,0,v);

/*
** Knoten in Baum einfügen
*/
  insert(kn);
```

Listing 134: Eine insert-Methode von CTree

```
/*
** Position als Iterator zurückgeben
*/
  return(_iterator(kn,this,0));
}
```

Listing 134: Eine insert-Methode von CTree (Forts.)

Um eine `insert`-Methode zu besitzen, die »kompatibel« mit den `insert`-Methoden der STL-Baum-Container ist, fügen wir die folgende Variante hinzu.

Es ist selbstverständlich, dass die übergebene Iterator-Position ignoriert werden muss, weil das Datenobjekt durch die engen Sortierkriterien des Baums nicht frei im Baum platziert werden kann.

```
inorder_iterator insert(_iterator i, const value_type &v) {
  return(insert(v));
}
```

Listing 135: Eine insert-Methode von CTree

Die folgende `insert`-Methode fügt Objekte in einem durch Iteratoren definierten Bereich in den Baum ein:

```
template<typename Input>
void insert(Input beg, Input end) {
  while(beg!=end)
    insert(*(beg++));
}
```

Listing 136: Eine insert-Methode von CTree

Zum Schluss wird die `insert`-Methode vorgestellt, die den Hauptteil der Einfügearbeit leisten muss, denn sie fügt den Knoten in die Baum-Struktur ein:

```
void insert(CKnot *kn) {

/*
```

Listing 137: Die insert-Methode für Knoten

```
** Baum leer?
** => Einzufügender Knoten wird die Wurzel
*/
  if(!m_root) {
    m_root=kn;
    m_size++;
    return;
  }

  CKnot *cur=m_root;
  while(cur) {
    if(Traits::gt(Traits::getKey(cur->m_data),
                  Traits::getKey(kn->m_data))) {

/*
** Einzufügender Knoten kleiner als aktueller Knoten?
** => Falls vorhanden, im linken Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten linker
**    Sohn des aktuellen Knotens
*/
      if(!cur->m_left) {
        cur->m_left=kn;
        kn->m_father=cur;
        m_size++;
        return;
      }
      else {
        cur=cur->m_left;
      }
    }

/*
** Einzufügender Knoten größer/gleich dem aktuellen Knoten?
** => Falls vorhanden, im rechten Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten rechter
**    Sohn des aktuellen Knotens
*/
    else {
      if(!cur->m_right) {
        cur->m_right=kn;
        kn->m_father=cur;
```

Listing 137: Die insert-Methode für Knoten (Forts.)

```
        m_size++;
        return;
      }
      else {
        cur=cur->m_right;
      }
    }
  }
}
```

Listing 137: Die insert-Methode für Knoten (Forts.)

erase

Mit den `erase`-Methoden können Knoten aus dem Baum entfernt werden.

Die erste Variante bekommt einen Schlüssel übergeben und löscht alle Knoten mit diesem Schlüssel. Sie liefert die Anzahl der gelöschten Knoten zurück:

```
virtual size_type erase(const key_type &k) {

/*
** Ersten Knoten mit Schlüssel k finden
*/
  CKnot *kn=findKnot(k);

/*
** Keine Knoten vorhanden?
** => Kein Knoten gelöscht
*/
  if(!kn)
    return(0);

/*
** Inorder-Iterator auf Knoten hinter dem zu löschenden
** Knoten setzen
*/
  inorder_iterator i=inorder_iterator(kn,this,0);
  ++i;
  size_type a=0;
```

Listing 138: Eine erase-Methode für Schlüssel

```
/*
** So lange laufen, wie noch Knoten mit
** Schlüssel k existieren
*/
  do {

/*
** Schlüssel löschen, neuen Knoten aus Iterator
** holen und Iterator inkrementieren
*/
    erase(kn);
    a++;
    kn=i.m_knot;
    ++i;
  } while((kn)&&(Traits::eq(Traits::getKey(kn->m_data),k)));
  return(a);
}
```

Listing 138: Eine erase-Methode für Schlüssel (Forts.)

Die oben stehende Methode benutzt einen Inorder-Iterator, der nicht vollständig initialisiert ist, denn der dritte Konstruktor-Parameter definiert nicht wie vorgeschrieben den ersten Knoten des Baumes. Letzlich spielt es in diesem Fall keine Rolle, weil der Iterator nur inkrementiert wird und von außen nicht zugänglich ist. Es vermindert aber die Laufzeit, weil eben der erste Knoten nicht bestimmt werden muss.

Die folgende Methode bekommt einen Iterator als Position übergeben und löscht den entsprechenden Knoten. Die Funktion gibt den in Inorder-Reihenfolge dahinter liegenden Knoten als Iterator-Position zurück:

```
virtual inorder_iterator erase(_iterator i) {

/*
** Iterator-Position hinter zu löschendem
** Knoten ermitteln
** (Ohne Anfangs-Position für Iterator zu bestimmen)
*/
  inorder_iterator io(i.m_knot,this,0);
  ++io;
```

Listing 139: Eine erase-Methode für Iteratoren

```
/*
** Konnte Knoten gelöscht werden?
** => Position hinter gelöschtem Knoten zurückgeben
** Andernfalls End-Position zurückgeben
*/
  if(erase(i.m_knot))
    return(_iterator(io.m_knot,this,0));
  else
    return(inorder_end());
}
```

Listing 139: Eine erase-Methode für Iteratoren (Forts.)

Die nächste `erase`-Methode löscht einen durch Iteratoren definierten Bereich. Der zweite Iterator muss dazu bezogen auf die Inorder-Reihenfolge hinter dem ersten Iterator liegen.

```
virtual inorder_iterator erase(inorder_iterator beg,
                               inorder_iterator end) {
  while((beg!=inorder_end())&&(beg!=end)) {
    CKnot *kn=beg.m_knot;
    ++beg;
    erase(kn);
  }
  return(beg);
}
```

Listing 140: Eine erase-Methode für Bereiche

Im Folgenden ist die `erase`-Methode aufgeführt, die einen Knoten tatsächlich aus dem Baum entfernt.

Dabei ist gewährleistet, dass alle Iterator-Positionen, die in Inorder-Reihenfolge auf Knoten hinter dem zu löschenden Knoten verweisen, gültig bleiben.

Aus dieser Garantie schöpfen jedoch nur die internen Methoden einen Vorteil. Für externe Einsätze gilt die alte Iterator-Regel, dass die Gültigkeit von Iteratoren nach Einfüge- oder Löschoperationen nicht gewährleistet wird.

```
bool erase(CKnot *cur) {
  if(!cur) return(false);

  CKnot *father=cur->m_father;

/*
** Zu löschender Knoten hat keine Söhne?
** => Kann problemlos gelöscht werden
*/
  if((!cur->m_left)&&(!cur->m_right)) {

/*
** Zu löschender Knoten ist die Wurzel?
** => Baum leer
*/
    if(cur==m_root) {
      m_root=0;
      delete(cur);
      m_size--;
      return(true);
    }

/*
** Zu löschender Knoten ist nicht die Wurzel?
** => Vater muss berücksichtigt werden
*/
    else {

/*
** Je nachdem, ob zu löschender Knoten der linke oder
** rechte Sohn des Vaters ist, muss entsprechender Sohn
** des Vaters auf 0 gesetzt werden.
*/
      if(father->m_left==cur) {
        father->m_left=0;
      }
      else {
        father->m_right=0;
      }
      delete(cur);
      m_size--;
      return(true);
    }
```

```
  }

/*
** Besitzt zu löschender Knoten zwei Söhne?
** => Zu löschenden Knoten durch symmetrischen
**    Vorgänger ersetzen und symmetrischen Vorgänger
**    löschen
*/
  if((cur->m_left)&&(cur->m_right)) {
    CKnot *sys=symmetricPred(cur);
    cur->m_data=sys->m_data;
    return(erase(sys));
  }

/*
** Besitzt zu löschender Knoten nur einen Sohn?
** => Sohn des zu löschenden Knotens wird Sohn vom
** Vater des zu löschenden Knotens
*/
  CKnot *son;
  if(cur->m_left)
    son=cur->m_left;
  else
    son=cur->m_right;

  if(cur!=m_root) {
    son->m_father=father;
    if(father->m_left==cur) {
      father->m_left=son;
    }
    else {
      father->m_right=son;
    }
  }

/*
** Ist zu löschender Sohn die Wurzel?
** => Sohn des zu löschenden Knotens wird Wurzel
*/
  else {
    son->m_father=0;
    m_root=son;
  }
```

```
  delete(cur);
  m_size--;
  return(true);
}
```

findFirstKnot

Die geschützte Methode `findFirstKnot` sucht bezogen auf die Inorder-Reihenfolge den ersten Knoten des Baumes mit übergebenem Schlüssel.

```
CKnot *findFirstKnot(const key_type &k) const {

/*
** Baum leer?
** => Knoten nicht gefunden
*/
  if(!m_root) return(0);

/*
** Obersten Knoten mit Schlüssel gleich k suchen
*/
  CKnot *cur=m_root;
  while(cur&&(Traits::ne(Traits::getKey(cur->m_data),k))) {
    if(Traits::lt(k,Traits::getKey(cur->m_data)))
      cur=cur->m_left;
    else
      cur=cur->m_right;
  }

/*
** Um ersten Knoten mit Schlüssel gleich k zu finden
** So lange im linken Ast hinabsteigen, wie Schlüssel gleich k
*/
  while((cur)&&(cur->m_left)&&
        (Traits::eq(Traits::getKey(cur->m_left->m_data),k)))
    cur=cur->m_left;

/*
** Überprüfen, ob symmetrische Vorgänge mit Schlüssel
** gleich k existieren
```

Listing 141: Die geschützte Methode findFirstKnot von CTree

```
*/
  if(cur) {
    CKnot *pred=symmetricPred(cur);
    while((pred)&&(Traits::eq(Traits::getKey(pred->m_data),k))) {
      cur=pred;
      pred=symmetricPred(cur);
    }
  }
  return(cur);
}
```

Listing 141: Die geschützte Methode findFirstKnot von CTree (Forts.)

findLastKnot

Diese Methode findet, bezogen auf die Inorder-Reihenfolge, den letzten Knoten, der gleich dem übergebenen Schlüssel ist:

```
CKnot *findLastKnot(const key_type &k) const{

/*
** Baum leer?
** => Knoten nicht gefunden
*/
  if(!m_root) return(0);

/*
** Obersten Knoten mit Schlüssel gleich k suchen
*/
  CKnot *cur=m_root;
  while(cur&&(Traits::ne(Traits::getKey(cur->m_data),k))) {
    if(Traits::lt(k,Traits::getKey(cur->m_data)))
      cur=cur->m_left;
    else
      cur=cur->m_right;
  }

/*
** Um letzten Knoten mit Schlüssel gleich k zu finden
** So lange im rechten Ast hinabsteigen wie Schlüssel gleich k
*/
```

Listing 142: Die geschützte Methode getLastKnot von CTree

```
  while((cur)&&(cur->m_right)&&
        (Traits::eq(Traits::getKey(cur->m_right->m_data),k)))
    cur=cur->m_right;

/*
** Überprüfen, ob symmetrische Nachfolger mit Schlüssel
** gleich k existieren
*/
  if(cur) {
    CKnot *succ=symmetricSucc(cur);
    while((succ)&&(Traits::eq(Traits::getKey(succ->m_data),k))) {
      cur=succ;
      succ=symmetricSucc(cur);
    }
  }
  return(cur);
}
```

Listing 142: Die geschützte Methode getLastKnot von CTree (Forts.)

symmetricPred

Diese Methode findet den symmetrischen Vorgänger eines Knotens.

```
CKnot *symmetricPred(CKnot *kn) const{
  CKnot *cur=kn->m_left;
  if(!cur)
    return(0);
  while(cur->m_right)
    cur=cur->m_right;
  return(cur);

}
```

Listing 143: Die geschützte Methode symmetricPred

symmetricSucc

Das Gegenstück zu `symmetricPred` bildet die Methode `symmetricSucc`, die den symmetrischen Nachfolger eines Knotens ermittelt:

```
CKnot *symmetricSucc(CKnot *kn) const {
  CKnot *cur=kn->m_right;
  if(!cur)
    return(0);
  while(cur->m_left)
    cur=cur->m_left;
  return(cur);
}
```

Listing 144: Die geschützte Methode symmetricSucc

lower_bound

`lower_bound` ermittelt die Iterator-Position des ersten Elements aus einer Gruppe gleicher Elemente:

```
inorder_iterator lower_bound(const key_type &k) {
  return(_iterator(findFirstKnot(k),this,0));
}

//*********

const_inorder_iterator lower_bound(const key_type &k) const{
  return(_const_iterator(findFirstKnot(k),this,0));
}
```

Listing 145: Die lower_bound-Methoden von CTree

upper_bound

`upper_bound` ermittelt die Iterator-Position hinter dem letzten Element aus einer Gruppe gleicher Elemente:

```
inorder_iterator upper_bound(const key_type &k) {
  inorder_iterator i=_iterator(findLastKnot(k),this,0);
  ++i;
  return(i);
}

//*********
```

Listing 146: Die upper_bound-Methoden von Ctree

```
const_inorder_iterator upper_bound(const key_type &k) const {
  const_inorder_iterator i=_const_iterator(findLastKnot(k),this,0);
  ++i;
  return(i);
}
```

Listing 146: Die upper_bound-Methoden von Ctree (Forts.)

find

Die Methode findet das erste Element mit entsprechendem Schlüssel. Von der Funktionalität her identisch mit `lower_bound`.

```
inorder_iterator find(const key_type &k) {
  return(_iterator(findFirstKnot(k),this,0));
}

//*********

const_inorder_iterator find(const key_type &k) const {
  return(_const_iterator(findFirstKnot(k),this,0));
}
```

Listing 147: Die find-Methoden von CTree

front & back

```
value_type &front() {
  return(inorder_begin().m_knot->m_data);
}

//*********

const value_type &front() const {
  return(inorder_begin().m_knot->m_data);
}

//*********
```

Listing 148: Die Methoden front und back von CTree

```
value_type &back() {
  return(inorder_rbegin().m_knot->m_data);
}

//*********

const value_type &back() const{
  return(inorder_rbegin().m_knot->m_data);
}
```

Listing 148: Die Methoden front und back von CTree (Forts.)

pop_front & pop_back

Die Methoden `pop_front` und `pop_back` entfernen das erste, beziehungsweise das letzte Element des Baums (bezogen auf die Inorder-Reihenfolge):

```
virtual void pop_front() {
  if(!empty())
    erase(inorder_begin().m_knot);
}

//*********

virtual void pop_back() {
  if(!empty())
    erase(inorder_rbegin().m_knot);
}
```

Listing 149: Die Methoden pop_front und pop_back von CTree

push_front & push_back

Diese beiden Methoden sind nur aus Kompatibilitätsgründen vorhanden. Die tatsächliche Einfügeposition des Elements ist durch die Sortierung des Baums vorgegeben.

```
virtual void push_front(const value_type &v) {
  insert(v);
}
```

Listing 150: Die Methoden push_front und push_back von Ctree

```
//*********

virtual void push_back(const value_type &v) {
  insert(v);
}
```

Listing 150: Die Methoden push_front und push_back von Ctree (Forts.)

Sie finden die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

set_tree_traits

Der Baum ist durch die Verwendung der selbst zu definierenden Tree-Traits ausgesprochen flexible. Wir können mit ihm sowohl die Funktionalität eines `set` als auch einer `map` erzeugen.

Schauen wir uns zunächst die `set_tree_traits` an, die das Verhalten von Schlüsseldaten definieren, die nur aus einer Komponente bestehen.

`value_type` und `key_type` sind in diesem Fall identisch.

Die Methode `getKey` liefert das Objekt selbst zurück.

Die Vergleichs-Methoden lassen sich alle auf den ==-Vergleich und das Prädikat `Cmp` zurückführen. Standardmäßig wird für `Cmp` das Prädikat `less` verwendet.

```
template<typename VType, typename Cmp= std::less<VType> >
class set_tree_traits {
public:
  typedef VType value_type;
  typedef VType key_type;

//*********

  static bool eq(const key_type &k1, const key_type &k2) {
    return(k1==k2);
  }

//*********

  static bool lt(const key_type &k1, const key_type &k2) {
```

Listing 151: Die Klasse set_tree_traits

```
    return(Cmp()(k1,k2));
  }

//*********

  static bool gt(const key_type &k1, const key_type &k2) {
    return(lt(k2,k1));
  }

//*********

  static bool ne(const key_type &k1, const key_type &k2) {
    return(!eq(k1,k2));
  }

//*********

  static bool ge(const key_type &k1, const key_type &k2) {
    return(!lt(k1,k2));
  }

//*********

  static bool le(const key_type &k1, const key_type &k2) {
    return(!lt(k2,k1));
  }

//*********

  static const key_type &getKey(const value_type &v) {
    return(v);
  }
};
```

Listing 151: Die Klasse set_tree_traits (Forts.)

CSetTree

Die Klasse `CSetTree` kombiniert nun `CTree` und `set_tree_traits` zu einer funktionstüchtigen Klasse.

Klassendefinition

Das Template besitzt als Parameter den zu verwaltenden Datentyp und das für die Vergleiche eingesetzte Prädikat (im Normalfall less).

Der einfacheren Handhabung wegen werden noch die Typen `base_type` für den Basisklassen-Typ und `my_type` für den eigenen Typ deklariert.

```
template<typename VType, typename Cmp= std::less<VType> >
class CSetTree : public CTree<set_tree_traits<VType, Cmp> > {
public:
  typedef CTree<set_tree_traits<VType, Cmp> > base_type;
  typedef CSetTree<VType, Cmp> my_type;
};
```

Listing 152: Die Klassendefinition von CSetTree

Konstruktoren

Mit den Konstruktoren wird lediglich die Funktionalität von `CTree` durchgeschliffen:

```
CSetTree()
: base_type()
{}

//*********

template<typename Input>
CSetTree(Input beg, Input end)
: base_type(beg,end)
{ }

//*********

CSetTree(const CSetTree &t)
: base_type(t)
{ }
```

Listing 153: Die Konstruktoren von CSetTree

Sie finden die Klassen `set_tree_traits` und `CSetTree` auf der CD in der `CSetTree.h`-Datei.

map_tree_traits

Um den Baum wie eine `map` einsetzen zu können, implementieren wir die `map_tree_traits`, die als zu verwaltenden Datentyp ein `pair` definiert, welches aus einem Schlüssel und den tatsächlichen Daten besteht.

`value_type` definiert damit den Datentyp der Nutzdaten und `key_type` den Datentyp der Schlüsseldaten.

Die Methode `getKey` liefert den Schlüssel, also das erste Element von `pair`, zurück.

Die Vergleichs-Methoden lassen sich alle auf den ==-Vergleich und das Prädikat `Cmp` zurückführen. Standardmäßig wird für `Cmp` das Prädikat `less` verwendet.

```
template<typename KType,
         typename VType,
         typename Cmp= std::less<KType> >
class map_tree_traits {
public:
  typedef typename std::pair<KType, VType> value_type;
  typedef KType key_type;

//*********

  static bool eq(const key_type &k1, const key_type &k2) {
    return(k1==k2);
  }

//*********

  static bool lt(const key_type &k1, const key_type &k2) {
    return(Cmp()(k1,k2));
  }

//*********

  static bool gt(const key_type &k1, const key_type &k2) {
    return(lt(k2,k1));
  }

//*********
```

Listing 154: Die Klasse map_tree_traits

```
  static bool ne(const key_type &k1, const key_type &k2) {
    return(!eq(k1,k2));
  }

//*********

  static bool ge(const key_type &k1, const key_type &k2) {
    return(!lt(k1,k2));
  }

//*********

  static bool le(const key_type &k1, const key_type &k2) {
    return(!lt(k2,k1));
  }

//*********

  static const key_type &getKey(const value_type &v) {
    return(v.first);
  }
};
```

Listing 154: Die Klasse map_tree_traits (Forts.)

CMapTree

Analog zu `CSetTree` erzeugt `CMapTree` mit `CTree` und `map_tree_traits` einen funktionsfähigen Baum.

Klassendefinition

Das Template besitzt als Parameter die Datentypen von Schlüssel- und Nutzdaten und das für die Vergleiche eingesetzte Prädikat (im Normalfall less).

Der einfacheren Handhabung wegen werden noch die Typen `base_type` für den Basisklassen-Typ und `my_type` für den eigenen Typ deklariert.

```
template<typename KType,
         typename VType,
         typename Cmp= std::less<KType> >
```

Listing 155: Die Klassendefinition von CMapTree

```
class CMapTree : public CTree<map_tree_traits<KType, VType, Cmp> > {
public:
  typedef CTree<map_tree_traits<KType, VType, Cmp> > base_type;
  typedef CMapTree<KType, VType, Cmp> my_type;
};
```

Listing 155: Die Klassendefinition von CMapTree (Forts.)

Konstruktoren

Auch hier bestehen die Konstruktoren nur aus Aufrufen der Basisklassen-Konstruktoren:

```
CMapTree()
: base_type()
{}

//*********

template<typename Input>
CMapTree(Input beg, Input end)
: base_type(beg,end)
{ }

//*********

CMapTree(const CMapTree &t)
: base_type(t)
{ }
```

Listing 156: Die Konstruktoren von CMapTree

Sie finden die Klassen `map_tree_traits` und `CMapTree` auf der CD in der *CMapTree.h*-Datei.

39 Wie kann ein Binärbaum mit Iteratoren in Inorder-Reihenfolge durchlaufen werden?

Bevor wir einen konkreten Iterator für den Baum implementieren, legen wir für die wichtigsten Iterator-Arten, die in den folgenden Rezepten noch vorgestellt werden, eine Basisklasse `_iterator` an.

_iterator

Die Klasse `_iterator` besitzt alle für einen Iterator notwendigen Attribute. Um die üblichen Durchlauf-Reihenfolgen implementieren zu können, reicht ein bidirektionaler Iterator. Wir leiten unsere Klasse dazu vom `iterator`-Template der Standardbibliothek ab.

Damit außerhalb des Baums von der Basisklasse kein Exemplar erzeugt werden kann, kommt sie in den geschützten Bereich.

Klassendefinition

Die Basisklasse unserer Iteratoren besitzt folgende Attribute:

- `m_knot` – ein Verweis auf den aktuellen Knoten und damit auf das aktuelle Element, auf das der Iterator zeigt
- `m_tree` – ein Zeiger auf den Baum, zu dem der Iterator gehört
- `m_begin` – die Bestimmung der Anfangsposition kostet je nach Iterator mehr als O(1)-Zeit, deswegen wird der Anfang nicht immer neu bestimmt, sondern im Iterator gespeichert.

```
class _iterator;
friend class _iterator;
class _iterator
: public std::iterator<std::bidirectional_iterator_tag, value_type> {
public:
  CKnot *m_knot;
  CKnot *m_begin;
  CTree *m_tree;
};
```

Listing 157: Die Klassendefinition von _iterator

Konstruktoren

Als Konstruktoren werden der Standard-Konstruktor und ein Konstruktor für alle Attribute implementiert.

```
_iterator()
  :m_knot(0), m_begin(0), m_tree(0)
```

Listing 158: Die Konstruktoren von _iterator

```
{}

//*********

_iterator(CKnot *kn, CTree *tr, CKnot *be)
  : m_knot(kn), m_tree(tr), m_begin(be)
{}
```

Listing 158: Die Konstruktoren von _iterator (Forts.)

operator* & operator->

Auf das mit dem Iterator verknüpfte Baum-Element muss mit Dereferenzierung und dem Zeiger-Operator zugegriffen werden können:

```
reference operator*() {
  return(m_knot->m_data);
}

//*********

pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 159: operator und operator-> von _iterator*

Vergleichsoperatoren

Für bidirektionale Iteratoren müssen die Vergleichsoperatoren == und != zur Verfügung gestellt werden:

```
bool operator==(_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(_iterator &i) const {
```

Listing 160: Die Vergleichsoperatoren von _iterator

```
  return(m_knot!=i.m_knot);
}
```

Listing 160: Die Vergleichsoperatoren von _iterator (Forts.)

isBegin

Um die Überprüfung, ob der Iterator augenblicklich die Anfangsposition seiner Durchlauf-Reihenfolge besitzt, nicht immer über einen Vergleich mit der unter Umständen zeitaufwändigeren `begin`-Methode umsetzen zu müssen, wurde die Methode `isBegin` implementiert.

Sie liefert `true` zurück, wenn der Iterator die Anfangsposition besitzt, andernfalls liefert sie `false`.

```
bool isBegin() const {
  return((m_knot!=0)&&(m_knot==m_begin));
}
```

Listing 161: Die Methode isBegin von _iterator

inorder_iterator

Mit dem `inorder_iterator` wird die Durchlaufreihenfolge Inorder implementiert. Sie entspricht der Sortierung des Baums und lässt sich rekursiv als linker Teilbaum – Knoten – rechter Teilbaum beschreiben.

Für den Baum in Abbildung 26 auf S. 155 beträgt die Inorder-Reihenfolge 0, 1, 2, 3, 4, 5, 6.

Klassendefinition

`inorder_iterator` wird von `_iterator` abgeleitet;

```
class inorder_iterator;
friend class inorder_iterator;
class inorder_iterator : public my_type::_iterator {
};
```

Listing 162: Die Klassendefinition von inorder_iterator

Konstruktoren

Zum einen werden die Iteratoren der Basisklasse durchgeschliffen.

Zusätzlich werden noch ein Kopier-Konstruktor und ein Konstruktor für Basisklassen-Objekte implementiert, die beide auf den Zuweisungsoperator der Klasse zurückgreifen.

```
inorder_iterator()
: _iterator(0,0,0)
{}

//*********

inorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

inorder_iterator(const inorder_iterator &i){
  *this=i;
}

//*********

inorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 163: Die Konstruktoren von inorder_iterator

Zuweisungsoperatoren

Der Zuweisungsoperator für Basisklassen-Objekte muss das Attribut `m_begin` neu bestimmen, weil dies unter Umständen gar nicht definiert wurde oder den Beginn einer anderen Durchlauf-Reihenfolge beinhaltet.

```
inorder_iterator &operator=(const inorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 164: Die Zuweisungsoperatoren von inorder_iterator

```
//*********

inorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->inorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 164: Die Zuweisungsoperatoren von inorder_iterator (Forts.)

operator++

Das erste Element der Durchlauf-Reihenfolge wird ermittelt, indem immer weiter im linken Ast hinabgestiegen wird, bis ein Knoten keinen linken Sohn mehr hat. Das übernimmt die Baum-Methode `inorder_begin`.

Von nun an werden zur Ermittlung des Nachfolgeknotens folgende Fälle unterschieden:

- Die Ende-Position ist erreicht: nichts machen.
- Der aktuelle Knoten besitzt einen rechten Sohn: symmetrischen Nachfolger bestimmen.
- Der aktuelle Knoten besitzt keinen rechten Sohn: so lange den Baum hochsteigen, bis der Vater des linken Astes erreicht ist.

```
inorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Existiert rechter Sohn?
** => Den linken Ast des rechten Sohns
**    komplett hinabsteigen
```

Listing 165: Die operator++-Methoden von inorder_iterator

```
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
    while(m_knot->m_left)
      m_knot=m_knot->m_left;
  }

/*
** Existiert kein rechter Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der linke Sohn des Vaters ist.
**     Der Vater ist der neue Knoten.
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_right==son));
  }
  return(*this);
}

//*********

inorder_iterator operator++(int) {
  inorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 165: Die operator++-Methoden von inorder_iterator (Forts.)

operator--

Die Funktionalität der Methode `operator--` ist identisch mit der `operator++`-Methode des Reverse-Iterators und wird dort erklärt.

```
inorder_iterator &operator--() {

/*
```

Listing 166: Die Methoden operator-- von inorder_iterator

```
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->inorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while(m_knot->m_right)
        m_knot=m_knot->m_right;
    }
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Existiert linker Sohn?
** => Den rechten Ast des linken Sohns
**    komplett hinabsteigen
*/
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
    while(m_knot->m_right)
      m_knot=m_knot->m_right;
  }

/*
** Existiert kein linker Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der rechte Sohn des Vaters ist.
**     Der Vater ist der neue Knoten.
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
```

Listing 166: Die Methoden operator-- von inorder_iterator (Forts.)

```
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_left==son));
  }
  return(*this);
}

//*********

inorder_iterator operator--(int) {
  inorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 166: Die Methoden operator-- von inorder_iterator (Forts.)

Iterator-Methoden

Die folgenden Methoden werden der Klasse `CTree` hinzugefügt, um Inorder-Iteratoren erzeugen zu können:

```
inorder_iterator inorder_begin() {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_left)
      kn=kn->m_left;
  }
  return(inorder_iterator(kn,this,kn));
}

//*********

inorder_iterator inorder_end() {
  return(inorder_iterator(0,this,0));
}
```

Listing 167: Die Iterator-Methoden für inorder_iterator

_const_iterator

Um auch Iteratoren auf konstante Bäume einsetzen zu können, implementieren wir eine geschützte Basisklasse `_const_iterator`, die der Klasse `_iterator` recht ähnlich ist.

Klassendefinition

Die Besonderheit des konstanten Iterators liegt im Zeiger auf einen konstanten Baum:

```
class _const_iterator;
friend class _const_iterator;
class _const_iterator
: public std::iterator<std::bidirectional_iterator_tag, value_type> {
public:
  CKnot *m_knot;
  CKnot *m_begin;
  const CTree *m_tree;
};
```

Listing 168: Die Klassendefinition von _const_iterator

Konstruktoren

Die Konstruktoren müssen berücksichtigen, dass der Baum konstant ist:

```
_const_iterator()
  :m_knot(0), m_begin(0), m_tree(0)
{}

//*********

_const_iterator(CKnot *kn, const CTree *tr, CKnot *be)
  : m_knot(kn), m_tree(tr), m_begin(be)
{}
```

Listing 169: Die Konstruktoren von _const_iterator

operator* & operator->

Der Zugriff über die Operatoren `*` und `->` muss einen Verweis auf ein konstantes Objekt zurückliefern.

```
const_reference operator*() {
  return(m_knot->m_data);
```

Listing 170: operator und operator-> von _const_iterator*

```
}

//*********

const_pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 170: operator und operator-> von _const_iterator (Forts.)*

Vergleichsoperatoren

```
bool operator==(_const_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(_const_iterator &i) const {
  return(m_knot!=i.m_knot);
}
```

Listing 171: Die Vergleichsoperatoren von _const_iterator

isBegin

```
bool isBegin() const {
  return((m_knot!=0)&&(m_knot==m_begin));
}
```

Listing 172: Die Methode isBegin von _const_iterator

const_inorder_iterator

Analog zu `inorder_iterator` wird `const_inorder_iterator` von `_const_iterator` abgeleitet.

Klassendefinition

```
class const_inorder_iterator;
friend class const_inorder_iterator;
class const_inorder_iterator : public my_type::_const_iterator {
};
```

Listing 173: Die Klassendefinition von const_inorder_iterator

Konstruktoren

```
const_inorder_iterator()
: _const_iterator(0,0,0)
{}

//*********

const_inorder_iterator(CKnot *kn, const CTree *tr, CKnot *be)
: _const_iterator(kn,tr,be)
{}

//*********

const_inorder_iterator(const const_inorder_iterator &i){
  *this=i;
}

//*********

const_inorder_iterator(const _const_iterator &i){
  *this=i;
}
```

Listing 174: Die Konstruktoren von const_inorder_iterator

Zuweisungsoperatoren

```
const_inorder_iterator &operator=(const const_inorder_iterator &i) {
  m_knot=i.m_knot;
```

Listing 175: Die Zuweisungsoperatoren von const_inorder_iterator

```
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

const_inorder_iterator &operator=(const _const_iterator &i) {
  *this=i.m_tree->inorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 175: Die Zuweisungsoperatoren von const_inorder_iterator (Forts.)

operator++

```
const_inorder_iterator &operator++() {
  if(!m_knot)
    return(*this);

  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
    while(m_knot->m_left)
      m_knot=m_knot->m_left;
  }
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_right==son));
  }
  return(*this);
}

//*********

const_inorder_iterator operator++(int) {
```

Listing 176: Die Methoden operator++ von const_inorder_iterator

```
  const_inorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 176: Die Methoden operator++ von const_inorder_iterator (Forts.)

operator--

```
const_inorder_iterator &operator--() {
  if(!m_knot) {
    *this=m_tree->inorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while(m_knot->m_right)
        m_knot=m_knot->m_right;
    }
    return(*this);
  }
  if(m_knot==m_begin)
    return(*this);

  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
    while(m_knot->m_right)
      m_knot=m_knot->m_right;
  }
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_left==son));
  }
  return(*this);
}

//*********

const_inorder_iterator operator--(int) {
```

Listing 177: Die Methoden operator-- von const_inorder_iterator

```
  const_inorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 177: Die Methoden operator-- von const_inorder_iterator (Forts.)

Iterator-Methoden

Die Iterator-Methoden werden wieder in `CTree` untergebracht.

```
const_inorder_iterator inorder_begin() const {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_left)
      kn=kn->m_left;
  }
  return(const_inorder_iterator(kn,this,kn));
}

const_inorder_iterator inorder_end() const {
  return(const_inorder_iterator(0,this,0));
}
```

Listing 178: Die Iterator-Methoden für const_inorder_iterator

reverse_inorder_iterator

Mit der Klasse `reverse_inorder_iterator` wird die umgekehrte Inorder-Durchlauf-Reihenfolge implementiert. Für Abbildung 26 wäre dies 6, 5, 4, 3, 2, 1, 0.

Klassendefinition

Die Klasse wird von `_iterator` abgeleitet:

```
class reverse_inorder_iterator;
friend class reverse_inorder_iterator;
class reverse_inorder_iterator : public my_type::_iterator {
};
```

Listing 179: Die Klassendefinition von reverse_inorder_iterator

Konstruktoren

Die Konstruktoren verfolgen den bewährten Ansatz, die Funktionalität der Zuweisungsoperatoren zu verwenden.

```
reverse_inorder_iterator()
: _iterator(0,0,0)
{}

//*********

reverse_inorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

reverse_inorder_iterator(const reverse_inorder_iterator &i){
  *this=i;
}

//*********

reverse_inorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 180: Die Konstruktoren von reverse_inorder_iterator

Zuweisungsoperatoren

Der Zuweisungsoperator für `_iterator`-Objekte muss nun `rbegin` verwenden.

```
reverse_inorder_iterator &operator=(const
                                    reverse_inorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 181: Die Zuweisungsoperatoren von reverse_inorder_iterator

```
//*********

reverse_inorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->inorder_rbegin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 181: Die Zuweisungsoperatoren von reverse_inorder_iterator (Forts.)

operator++

Das erste Element der Durchlauf-Reihenfolge wird ermittelt, indem immer weiter im linken Ast hinabgestiegen wird, bis ein Knoten keinen linken Sohn mehr hat. Das übernimmt die Baum-Methode `inorder_begin`.

Von nun an werden zur Ermittlung des Nachfolgeknotens folgende Fälle unterschieden:

- Die Ende-Position ist erreicht: nichts machen.
- Der aktuelle Knoten besitzt einen linken Sohn: symmetrischen Vorgänger bestimmen.
- Der aktuelle Knoten besitzt keinen linken Sohn: so lange den Baum hochsteigen, bis der Vater des rechten Astes erreicht ist.

```
reverse_inorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Existiert linker Sohn?
** => Den rechten Ast des linken Sohns
**    komplett hinabsteigen
*/
```

Listing 182: Die Methoden operator ++ von reverse_inorder_iterator

```
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
    while(m_knot->m_right)
      m_knot=m_knot->m_right;
  }

/*
** Existiert kein linker Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der rechte Sohn des Vaters ist.
**     Der Vater ist der neue Knoten.
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_left==son));
  }
  return(*this);
}

//*********

reverse_inorder_iterator operator++(int) {
  reverse_inorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 182: Die Methoden operator ++ von reverse_inorder_iterator (Forts.)

operator--

Die Methode `operator--` besitzt die gleiche Durchlauf-Reihenfolge wie `operator++` von `inorder_iterator`.

```
reverse_inorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
```

Listing 183: Die Methoden operator-- von reverse_inorder_iterator

```
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->inorder_begin();
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Existiert rechter Sohn?
** => Den linken Ast des rechten Sohns
**    komplett hinabsteigen
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
    while(m_knot->m_left)
      m_knot=m_knot->m_left;
  }

/*
** Existiert kein rechter Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der linke Sohn des Vaters ist.
**     Der Vater ist der neue Knoten
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_right==son));
  }
  return(*this);
}
```

Listing 183: Die Methoden operator-- von reverse_inorder_iterator (Forts.)

```
reverse_inorder_iterator operator--(int) {
  reverse_inorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 183: Die Methoden operator-- von reverse_inorder_iterator (Forts.)

Iterator-Methoden

Folgende Methoden werden der Klasse `CTree` mitgegeben, um Iteratoren des Typs `reverse_inorder_iterator` zu erzeugen:

```
reverse_inorder_iterator inorder_rbegin() {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_right)
      kn=kn->m_right;
  }
  return(reverse_inorder_iterator(kn,this,kn));
}

//*********

reverse_inorder_iterator inorder_rend() {
  return(reverse_inorder_iterator(0,this,0));
}
```

Listing 184: Die Iterator-Methoden für reverse_inorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

40 Wie kann ein Binärbaum mit Iteratoren in Preorder-Reihenfolge durchlaufen werden?

In Preorder-Reihenfolge wird von einem Baum zuerst die Wurzel, anschließend der linke Teilbaum und zum Schluss der rechte Teilbaum ausgegeben.

Damit ergibt der Baum in Abbildung 26 auf Seite 155 die Knoten-Reihenfolge 3, 1, 0, 2, 5, 4, 6.

preorder_iterator

Die Klasse `preorder_iterator` erlaubt es, einen Baum mit Iteratoren in der Preorder-Reihenfolge zu durchlaufen.

Klassendefinition

`preorder_iterator` wird von `_iterator` aus Rezept 39 abgeleitet, um Positionen zwischen den verschiedenen Iteratoren austauschen zu können.

```
class preorder_iterator;
friend class preorder_iterator;
class preorder_iterator : public my_type::_iterator {
};
```

Listing 185: Die Klassendefinition von preorder_iterator

Konstruktoren

Die Konstruktoren stellen die Funktionalität der Basisklasse zur Verfügung. Zusätzlich bieten zwei Konstruktoren unter Zuhilfenahme der Zuweisungsoperatoren noch die Möglichkeit, einen Iterator aus einem `_iterator`-Objekt oder einem `preorder_iterator`-Objekt (Kopier-Konstruktor) zu erzeugen.

```
preorder_iterator()
: _iterator(0,0,0)
{}

//*********

preorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

preorder_iterator(const preorder_iterator &i){
  *this=i;
}

//*********
```

Listing 186: Die Konstruktoren von preorder_iterator

```
preorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 186: Die Konstruktoren von preorder_iterator (Forts.)

Zuweisungsoperatoren

```
preorder_iterator &operator=(const preorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

preorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->preorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 187: Die Zuweisungsoperatoren von preorder_iterator

operator++

Nachdem zum Beispiel mit der Methode `preorder_begin` ein gültiger Iterator erzeugt wurde, ermittelt die `operator++`-Methode den nächsten Knoten der Durchlauf-Reihenfolge durch Unterscheidung der folgenden Fälle:

- Linker Ast noch nicht durchlaufen: ersten Knoten des linken Astes bestimmen.
- Linker Ast durchlaufen, rechten Ast noch nicht: ersten Knoten des rechten Astes bestimmen.
- Beide Äste durchlaufen: im Baum so lange aufsteigen, wie sich der Iterator im rechten Teilbaum befindet oder er sich im linken Teilbaum befindet und kein rechter Ast vorhanden ist.

```
preorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Linker Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
*/
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Linken Ast, aber rechten Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  else if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Beide Äste abgearbeitet?
** => aufsteigen und nächsten Teilast finden
*/
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im linken Ast aufsteigt und
** kein rechter Ast verfügbar ist oder der Iterator im rechten
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
```

Listing 188: Die Methoden operator ++ von preorder_iterator

```
            (((son==m_knot->m_left)&&(!m_knot->m_right) )||
             (son==m_knot->m_right)));
    if(m_knot)
      m_knot=m_knot->m_right;
  }
  return(*this);
}

//*********

preorder_iterator operator++(int) {
  preorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 188: Die Methoden operator ++ von preorder_iterator (Forts.)

operator--

`operator--` besitzt nahezu die gleiche Funktionsweise wie `operator++` des Reverse-Iterators:

```
preorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->preorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
    return(*this);
```

Listing 189: Die Methoden operator-- von preorder_iterator

```
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Linker Ast noch nicht abgearbeitet?
** => Linken Ast so weit wie möglich hinabsteigen,
**    wobei rechte Teiläste Priorität haben
*/
    if((m_knot->m_left)&&((m_knot->m_left!=son))) {
      m_knot=m_knot->m_left;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
  }
  return(*this);
}

//*********

preorder_iterator operator--(int) {
  preorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 189: Die Methoden operator-- von preorder_iterator (Forts.)

Iterator-Methoden

Es folgen die beiden Methoden aus `CTree`, mit denen Preorder-Iteratoren erzeugt werden können:

```
preorder_iterator preorder_begin() {
  return(preorder_iterator(m_root,this, m_root));
}

//*********

preorder_iterator preorder_end() {
  return(preorder_iterator(0,this,0));
}
```

Listing 190: Die Iterator-Methoden für preorder_iterator

const_preorder_iterator

Der Vollständigkeit halber wird hier noch der `const_preorder_iterator` vorgestellt, der die Preorder-Traversion konstanter Bäume erlaubt:

Klassendefinition

`const_preorder_iterator` wird von `_const_iterator` aus Rezept 39 abgeleitet:

```
class const_preorder_iterator;
friend class const_preorder_iterator;
class const_preorder_iterator : public my_type::_const_iterator {
};
```

Listing 191: Die Klassendefinition von const_preorder_iterator

Konstruktoren

```
const_preorder_iterator()
  : _const_iterator(0,0,0)
{}

//*********
```

Listing 192: Die Konstruktoren von const_preorder_iterator

```
const_preorder_iterator(CKnot *kn, const CTree *tr, CKnot *be)
  : _const_iterator(kn,tr,be)
{}

//*********

const_preorder_iterator(const const_preorder_iterator &i){
  *this=i;
}

//*********

const_preorder_iterator(const _const_iterator &i){
  *this=i;
}
```

Listing 192: Die Konstruktoren von const_preorder_iterator (Forts.)

Zuweisungsoperatoren

```
const_preorder_iterator &operator=(const
                                   const_preorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

const_preorder_iterator &operator=(const _const_iterator &i) {
  *this=i.m_tree->const_preorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 193: Die Zuweisungsoperatoren von const_preorder_iterator

operator++

```
const_preorder_iterator &operator++() {
  if(!m_knot)
    return(*this);

  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }
  else if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }
  else {
    CKnot * son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
            (((son==m_knot->m_left)&&(!m_knot->m_right) )||
              (son==m_knot->m_right)));
    if(m_knot)
      m_knot=m_knot->m_right;
  }
  return(*this);
}

//*********

const_preorder_iterator operator++(int) {
  const_preorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 194: Die Methoden operator ++ von const_preorder_iterator

operator--

```
const_preorder_iterator &operator--() {
  if(!m_knot) {
```

Listing 195: Die Methoden operator-- von const_preorder_iterator

```
    *this=m_tree->preorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
    return(*this);
  }

  if(m_knot==m_begin)
    return(*this);

  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {
    if((m_knot->m_left)&&((m_knot->m_left!=son))) {
      m_knot=m_knot->m_left;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
  }
  return(*this);
}

//*********

const_preorder_iterator operator--(int) {
  const_preorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 195: Die Methoden operator-- von const_preorder_iterator (Forts.)

Iterator-Methoden

Und zum guten Schluss die Iterator-erzeugenden Methoden aus `CTree`:

```
const_preorder_iterator preorder_begin() const {
  return(const_preorder_iterator(m_root,this, m_root));
}

//*********

const_preorder_iterator preorder_end() const {
  return(const_preorder_iterator(0,this,0));
}
```

Listing 196: Die Iterator-Methoden für const_preorder_iterator

reverse_preorder_iterator

Um einen Baum in umgekehrter Preorder-Reihenfolge durchlaufen zu können, was in Abbildung 26 auf Seite 155 der Reihenfolge 6, 4, 5, 2, 0, 1, 3 entspricht, fügen wir die Klasse `reverse_preorder_iterator` hinzu.

Klassendefinition

Wie alle nicht-konstanten Iteratoren bisher wird `reverse_preorder_iterator` von `_iterator` aus Rezept 39 abgeleitet:

```
class reverse_preorder_iterator;
friend class reverse_preorder_iterator;
class reverse_preorder_iterator : public my_type::_iterator {
};
```

Listing 197: Die Klassendefinition von reverse_preorder_iterator

Konstruktoren

```
reverse_preorder_iterator()
: _iterator(0,0,0)
{}
```

Listing 198: Die Konstruktoren von reverse_preorder_iterator

```
//*********

reverse_preorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: iterator(kn,tr,be)
{}

//*********

reverse_preorder_iterator(const reverse_preorder_iterator &i){
  *this=i;
}

//*********

reverse_preorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 198: Die Konstruktoren von reverse_preorder_iterator (Forts.)

Zuweisungsoperatoren

```
reverse_preorder_iterator &operator=(const reverse_preorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

reverse_preorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->preorder_rbegin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 199: Die Zuweisungsoperatoren von reverse_preorder_iterator

operator++

Um den nächsten Knoten zu bestimmen, ermittelt `operator++` zunächst den Vater des aktuellen Knotens.

Sollte der aktuelle Knoten der linke Sohn gewesen sein, dann ist der Vater der neue Knoten.

Sollte der aktuelle Knoten nicht der linke Sohn gewesen sein, so steigt `operator++` immer tiefer im linken Teilbaum hinab.

Sollte ein Knoten zwei Söhne haben, so wird immer der rechte genommen.

Der letzte auf diese Weise ermittelte Knoten ist der neue Knoten.

```
reverse_preorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Linker Ast noch nicht abgearbeitet?
** => Linken Ast so weit wie möglich hinabsteigen,
**    wobei rechte Teiläste Priorität haben
*/
    if((m_knot->m_left)&&((m_knot->m_left!=son))) {
      m_knot=m_knot->m_left;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
```

Listing 200: Die Methoden operator ++ von reverse_preorder_iterator

```
      }
    }
  }
  return(*this);
}

//*********

reverse_preorder_iterator operator++(int) {
  reverse_preorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 200: Die Methoden operator ++ von reverse_preorder_iterator (Forts.)

operator--

Diese Methode verfolgt das gleiche Prinzip wie `operator++` von `preorder_iterator`:

```
reverse_preorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->preorder_begin();
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Linker Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
```

Listing 201: Die Methoden operator-- von reverse_preorder_iterator

```
*/
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Linken Ast, aber rechten Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  else if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Beide Äste abgearbeitet?
** => aufsteigen und nächsten Teilast finden
*/
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im linken Ast aufsteigt und
** kein rechter Ast verfügbar ist oder der Iterator im rechten
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
           (((son==m_knot->m_left)&&(!m_knot->m_right) )||
            (son==m_knot->m_right)));
    if(m_knot)
      m_knot=m_knot->m_right;
  }
  return(*this);
}

//*********

reverse_preorder_iterator operator--(int) {
  reverse_preorder_iterator tmp=*this;
```

Listing 201: Die Methoden operator-- von reverse_preorder_iterator (Forts.)

```
  --(*this);
  return(tmp);
}
```

Listing 201: Die Methoden operator-- von reverse_preorder_iterator (Forts.)

Iterator-Methoden

Mit den folgenden Methoden lassen sich über ein `CTree`-Objekt entsprechende Iteratoren erzeugen:

```
reverse_preorder_iterator preorder_rbegin() {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_right)
      kn=kn->m_right;
  }
  return(reverse_preorder_iterator(kn,this, kn));
}

//*********

reverse_preorder_iterator preorder_rend() {
  return(reverse_preorder_iterator(0,this,0));
}
```

Listing 202: Die Iterator-Methoden für reverse_preorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

41 Wie kann ein Binärbaum mit Iteratoren in Postorder-Reihenfolge durchlaufen werden?

Die Postorder-Reihenfolge ist bezogen auf einen Knoten so definiert, dass zuerst der linke Teilbaum, dann der rechte Teilbaum und zum Schluss der Knoten selbst ausgegeben wird.

Der Baum in Abbildung 26 auf Seite 155 würde in Postorder 0, 2, 1, 4, 6, 5, 3 ausgegeben.

postorder_iterator

Für die Postorder-Funktionalität wird eine neue Klasse `postorder_iterator` angelegt.

Klassendefinition

Auch `postorder_iterator` wird von `_iterator` aus Rezept 39 abgeleitet.

```
class postorder_iterator;
friend class postorder_iterator;
class postorder_iterator : public my_type::_iterator {
};
```

Listing 203: Die Klassendefinition von postorder_iterator

Konstruktoren

Als Konstruktoren steht der Kopier-Konstruktor, ein Konstruktor für `_iterator`-Objekte und die beiden durchgeschliffenen Basisklassen-Konstruktoren zur Verfügung.

```
postorder_iterator()
: _iterator(0,0,0)
{}

//*********

postorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

postorder_iterator(const postorder_iterator &i){
  *this=i;
}

//*********

postorder_iterator(const _iterator &i){
```

Listing 204: Die Konstruktoren von postorder_iterator

```
  *this=i;
}
```

Listing 204: Die Konstruktoren von postorder_iterator (Forts.)

Zuweisungsoperatoren

Die Zuweisungsoperatoren, von denen auch zwei der Konstruktoren Gebrauch machen, weisen einem `postorder_iterator`-Objekt ein anderes `postorder_iterator`-Objekt oder ein `_iterator`-Objekt zu:

```
postorder_iterator &operator=(const postorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

postorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->postorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 205: Die Zuweisungsoperatoren von postorder_iterator

operator++

Um den nächsten Knoten zu bestimmen, ermittelt `operator++` zunächst den Vater des aktuellen Knotens.

Sollte der aktuelle Knoten der rechte Sohn gewesen sein, dann ist der Vater der neue Knoten.

Sollte der aktuelle Knoten nicht der rechte Sohn gewesen sein, so steigt `operator++` immer tiefer im rechten Teilbaum hinab.

Sollte ein Knoten zwei Söhne haben, so wird immer der linke genommen.

Der letzte auf diese Weise ermittelte Knoten ist der neue Knoten.

```
postorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Rechter Ast noch nicht abgearbeitet?
** => Rechten Ast so weit wie möglich hinabsteigen,
**    wobei linke Teiläste Priorität haben
*/
    if((m_knot->m_right)&&((m_knot->m_right!=son))) {
      m_knot=m_knot->m_right;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_left)
          m_knot=m_knot->m_left;
        else
          m_knot=m_knot->m_right;
      }
    }
  }
  return(*this);
}

//*********

postorder_iterator operator++(int) {
  postorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 206: Die Methoden operator ++ von postorder_iterator

operator--

Die Methode `operator--` ist von der Grundidee her identisch mit `operator++` des Reverse-Iterators:

```
postorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->postorder_begin();
    m_knot=m_tree->m_root;
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Rechter Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Rechten Ast, aber linken Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
*/
  else if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Beide Äste abgearbeitet?
```

Listing 207: Die Methoden operator-- von postorder_iterator

```
** => aufsteigen und nächsten Teilast finden
*/
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im rechten Ast aufsteigt und
** kein linker Ast verfügbar ist oder der Iterator im linken
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
           (((son==m_knot->m_right)&&(!m_knot->m_left) )||
            (son==m_knot->m_left)));
    if(m_knot)
      m_knot=m_knot->m_left;
  }
  return(*this);
}

//*********

postorder_iterator operator--(int) {
  postorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 207: Die Methoden operator-- von postorder_iterator (Forts.)

Iterator-Methoden

Um einen `postorder_iterator` zu erzeugen, werden die folgenden Methoden von `CTree` benutzt:

```
postorder_iterator postorder_begin() {
  CKnot *kn=m_root;
  if(kn) {
    while((kn->m_left)||(kn->m_right)) {
```

Listing 208: Die Iterator-Methoden für postorder_iterator

```
      if(kn->m_left)
        kn=kn->m_left;
      else
        kn=kn->m_right;
    }
  }
  return(postorder_iterator(kn,this,kn));
}

//*********

postorder_iterator postorder_end() {
  return(postorder_iterator(0,this,0));
}
```

Listing 208: Die Iterator-Methoden für postorder_iterator (Forts.)

const_postorder_iterator

Natürlich muss auch ein konstanter Baum in Postorder-Reihenfolge durchlaufen werden können, deswegen wird die Klasse `const_postorder_iterator` geschrieben.

Klassendefinition

Um austauschbar mit den anderen Iteratoren für konstante Bäume zu bleiben, wird `const_postorder_iterator` von `_const_iterator` aus Rezept 39 abgeleitet:

```
class const_postorder_iterator;
friend class const_postorder_iterator;
class const_postorder_iterator : public my_type::_const_iterator {
};
```

Listing 209: Die Klassendefinition von const_postorder_iterator

Konstruktoren

```
const_postorder_iterator()
: _const_iterator(0,0,0)
{}
```

Listing 210: Die Konstruktoren von const_postorder_iterator

```
//*********

const_postorder_iterator(CKnot *kn, const CTree *tr, CKnot *be)
: _const_iterator(kn,tr,be)
{}

//*********

const_postorder_iterator(const const_postorder_iterator &i){
  *this=i;
}

//*********

const_postorder_iterator(const _const_iterator &i){
  *this=i;
}
```

Listing 210: Die Konstruktoren von const_postorder_iterator (Forts.)

Zuweisungsoperatoren

```
const_postorder_iterator &operator=(const
                                    const_postorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

const_postorder_iterator &operator=(const _const_iterator &i) {
  *this=i.m_tree->postorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 211: Die Zuweisungsoperatoren von const_postorder_iterator

operator++

```
const_postorder_iterator &operator++() {
  if(!m_knot)
    return(*this);

  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {
    if((m_knot->m_right)&&((m_knot->m_right!=son))) {
      m_knot=m_knot->m_right;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_left)
          m_knot=m_knot->m_left;
        else
          m_knot=m_knot->m_right;
      }
    }
  }
  return(*this);
}

//*********

const_postorder_iterator operator++(int) {
  const_postorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 212: Die Methoden operator ++ von const_postorder_iterator

operator--

```
const_postorder_iterator &operator--() {
  if(!m_knot) {
    *this=m_tree->postorder_begin();
    m_knot=m_tree->m_root;
    return(*this);
  }
```

Listing 213: Die Methoden operator-- von const_postorder_iterator

```
  if(m_knot==m_begin)
    return(*this);

  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }
  else if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }
  else {
    CKnot * son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
           (((son==m_knot->m_right)&&(!m_knot->m_left) )||
            (son==m_knot->m_left)));
    if(m_knot)
      m_knot=m_knot->m_left;
  }
  return(*this);
}

//*********

const_postorder_iterator operator--(int) {
  const_postorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 213: Die Methoden operator-- von const_postorder_iterator (Forts.)

Iterator-Methoden

```
const_postorder_iterator postorder_begin() const {
  CKnot *kn=m_root;
  if(kn) {
    while((kn->m_left)||(kn->m_right)) {
```

Listing 214: Die Iterator-Methoden für const_postorder_iterator

```
      if(kn->m_left)
        kn=kn->m_left;
      else
        kn=kn->m_right;
    }
  }
  return(const_postorder_iterator(kn,this,kn));
}

//*********

const_postorder_iterator postorder_end() const {
  return(const_postorder_iterator(0,this,0));
}
```

Listing 214: Die Iterator-Methoden für const_postorder_iterator (Forts.)

reverse_postorder_iterator

Wir wollen auch die Möglichkeit unterstützen, mit einem Reverse-Iterator den Baum in umgekehrter Postorder-Reihenfolge zu durchlaufen. Für unseren Baum in Abbildung 26 auf Seite 155 wäre dies 3, 5, 6, 4, 1, 2, 0.

Klassendefinition

Wir leiten `reverse_postorder_iterator` von der Klasse `_iterator` aus Rezept 39 ab:

```
class reverse_postorder_iterator;
friend class reverse_postorder_iterator;
class reverse_postorder_iterator : public my_type::_iterator {
};
```

Listing 215: Die Klassendefinition von reverse_postorder_iterator

Konstruktoren

```
reverse_postorder_iterator()
  : _iterator(0,0,0)
{}
```

Listing 216: Die Konstruktoren von reverse_postorder_iterator

```
//*********

reverse_postorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
  : _iterator(kn,tr,be)
{}

//*********

reverse_postorder_iterator(const reverse_postorder_iterator &i){
  *this=i;
}

//*********

reverse_postorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 216: Die Konstruktoren von reverse_postorder_iterator (Forts.)

Zuweisungsoperatoren

```
reverse_postorder_iterator &operator=(const
                                reverse_postorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

reverse_postorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->postorder_rbegin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 217: Die Zuweisungsoperatoren von reverse_postorder_iterator

operator++

Die Methode `operator++` ermittelt den nächsten Knoten der Durchlauf-Reihenfolge durch Unterscheidung der folgenden Fälle:

- Rechter Ast noch nicht durchlaufen: ersten Knoten des rechten Astes bestimmen.
- Rechten Ast durchlaufen, linken Ast noch nicht: ersten Knoten des linken Astes bestimmen.
- Beide Äste durchlaufen: im Baum so lange aufsteigen, wie sich der Iterator im linken Teilbaum befindet oder er sich im rechten Teilbaum befindet und kein linker Ast vorhanden ist.

```
reverse_postorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Rechter Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Rechten Ast, aber linken Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
*/
  else if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Beide Äste abgearbeitet?
** => aufsteigen und nächsten Teilast finden
*/
```

Listing 218: Die Methoden operator ++ von reverse_postorder_iterator

```
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im rechten Ast aufsteigt und
** kein linker Ast verfügbar ist oder der Iterator im linken
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
            (((son==m_knot->m_right)&&(!m_knot->m_left) )||
             (son==m_knot->m_left)));
    if(m_knot)
      m_knot=m_knot->m_left;
  }
  return(*this);
}

//*********

reverse_postorder_iterator operator++(int) {
  reverse_postorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 218: Die Methoden operator ++ von reverse_postorder_iterator (Forts.)

operator--

Die Methode `operator--` verfolgt zur Ermittlung des nächsten Knotens eine ähnliche Strategie wie `operator++` von `postorder_iterator`:

```
reverse_postorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
```

Listing 219: Die Methoden operator-- von reverse_postorder_iterator

```
  if(!m_knot) {
    *this=m_tree->postorder_begin();
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Rechter Ast noch nicht abgearbeitet?
** => Rechten Ast so weit wie möglich hinabsteigen,
**    wobei linke Teiläste Priorität haben
*/
    if((m_knot->m_right)&&((m_knot->m_right!=son))) {
      m_knot=m_knot->m_right;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_left)
          m_knot=m_knot->m_left;
        else
          m_knot=m_knot->m_right;
      }
    }
  }
  return(*this);
}

//*********

reverse_postorder_iterator operator--(int) {
  reverse_postorder_iterator tmp=*this;
```

Listing 219: Die Methoden operator-- von reverse_postorder_iterator (Forts.)

```
  --(*this);
  return(tmp);
}
```

Listing 219: Die Methoden operator-- von reverse_postorder_iterator (Forts.)

Iterator-Methoden

Erzeugt wird ein `reverse_postorder_iterator`-Objekt mit folgenden Methoden, die in `CTree` zu finden sind:

```
reverse_postorder_iterator postorder_rbegin() {
  return(reverse_postorder_iterator(m_root,this,m_root));
}

//*********

reverse_postorder_iterator postorder_rend() {
  return(reverse_postorder_iterator(0,this,0));
}
```

Listing 220: Die Iterator-Methoden für reverse_postorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der *CTree.h*-Datei.

42 Wie kann ein Binärbaum mit Iteratoren in Levelorder-Reihenfolge durchlaufen werden?

Die Durchlauf-Reihenfolge Levelorder erfordert ein anderes Vorgehen als das der anderen Iteratoren. Bei ihr werden die Knoten Baumlevel für Baumlevel ausgegeben. Der Baum in Abbildung 26 auf S. 155 ergibt in Levelorder ausgegeben beispielsweise 3, 1, 5, 0, 2, 4, 6.

Während Durchlauf-Reihenfolgen wie Inorder, Preorder oder Postorder einer Tiefensuche ähneln, baut der Ansatz für eine Levelorder-Reihenfolge auf einer Breitensuche auf.

levelorder_iterator

Um die Levelorder-Funktionalität umsetzen zu können, schreiben wir eine Klasse `levelorder_iterator`.

Klassendefinition

Der Levelorder-Iterator benötigt mit Abstand den größten Speicherbereich. Um ihn dennoch so gering wie möglich zu halten, verzichten wir bei ihm auf operator-- und definieren ihn als Forward-Iterator.

Die Klasse besitzt folgende Attribute:

- `m_knot` – ein Verweis auf den aktuellen Knoten und damit auf das aktuelle Element, auf das der Iterator zeigt.
- `m_tree` – ein Zeiger auf den Baum, zu dem der Iterator gehört.
- `m_queue` – im konkreten Fall eine Deque, in der die Knoten des nächsten Levels gespeichert werden. (Notwendig für ein Breitensuche-Verhalten.)
- `m_level` – der Level, auf dem sich der aktuelle Knoten im Baum befindet.

```
class levelorder_iterator;
friend class levelorder_iterator;
class levelorder_iterator
: public std::iterator<std::forward_iterator_tag, value_type> {
private:
  CKnot *m_knot;
  CTree *m_tree;
  int m_level;
  std::deque<CKnot*> m_queue;
};
```

Listing 221: Die Klassendefinition von levelorder_iterator

Konstruktoren

Es steht sowohl ein Standard-Konstruktor zur Verfügung als auch ein von den Methoden `levelorder_begin` und `levelorder_end` verwendeter Konstruktor mit einem Knoten- und einem Baum-Objekt als Parameter.

```
levelorder_iterator()
:m_knot(0), m_tree(0),m_level(0) {
  m_queue.push_back(0);
}

//*********
```

Listing 222: Die Konstruktoren von levelorder_iterator

```
levelorder_iterator(CKnot *kn, CTree *tr)
: m_knot(kn), m_tree(tr),m_level(0) {
  m_queue.push_back(0);
}
```

Listing 222: Die Konstruktoren von levelorder_iterator (Forts.)

Zuweisungsoperatoren

Als Zuweisungsoperator benutzen wir den standardmäßig vom Compiler zur Verfügung gestellten Zuweisungsoperator für flache Kopien.

operator* & operator->

```
reference operator*() {
  return(m_knot->m_data);
}

//*********

pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 223: Die Methoden operator und operator-> von levelorder_iterator*

Vergleichsoperatoren

```
bool operator==(levelorder_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(levelorder_iterator &i) const {
  return(m_knot!=i.m_knot);
}
```

Listing 224: Die Vergleichsoperatoren von levelorder_iterator

operator++

Bevor die `operator++`-Methode den nächsten Knoten aus der Queue holt, werden die Söhne des aktuellen Knotens an die Queue angehängt. Auf diese Weise liegen die Knoten Level für Level in der Queue.

Jedes Mal, wenn ein neuer Level »angebrochen« wird, fügen wir einen 0-Konten in die Queue ein, um später den Wechsel des Levels erkennen und `m_level` um eins erhöhen zu können.

Weil wir keinen Operator `--` implementieren, können die Knoten tatsächlich aus der Queue entfernt werden, sodass sich in der Queue maximal nur die Knoten von zwei Leveln befinden.

```
levelorder_iterator &operator++() {

/*
** Bevor nächster Knoten aus Queue geholt wird,
** die beiden Söhne (falls vorhanden) an die Queue
** anhängen
*/
  if(m_knot->m_left)
    m_queue.push_back(m_knot->m_left);
  if(m_knot->m_right)
    m_queue.push_back(m_knot->m_right);

/*
** Nächsten Knoten aus Queue holen
*/
  if(m_queue.size()!=0) {
    m_knot=m_queue.front();
    m_queue.pop_front();

/*
** Nächsten Level erreicht?
** => Neue Markierung in die Queue schieben und
**    Level um eins erhöhen
*/
    if(!m_knot) {
      m_level++;
      m_queue.push_back(0);
      m_knot=m_queue.front();
```

Listing 225: Die Methode operator++ von levelorder_iterator

```
      m_queue.pop_front();
    }
  }

/*
** Queue leer?
** => Ende des Baumes erreicht
*/
  else
    m_knot=0;
  return(*this);
}

//*********

levelorder_iterator operator++(int) {
  levelorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 225: Die Methode operator++ von levelorder_iterator (Forts.)

getLevel

Mit `getLevel` kann der Level des Knotens bestimmt werden, auf den der Iterator zeigt.

```
int getLevel() const {
  return(m_level);
}
```

Listing 226: Die getLevel-Methode von levelorder_iterator

Iterator-Methoden

Mit den Methoden `levelorder_begin` und `levelorder_end` kann über ein `CTree`-Objekt ein entsprechender Levelorder-Iterator erzeugt werden:

```
levelorder_iterator levelorder_begin() {
  return(levelorder_iterator(m_root,this));
}

//*********

levelorder_iterator levelorder_end() {
  return(levelorder_iterator(0,this));
}
```

Listing 227: Die Iterator-Methoden für levelorder_iterator

const_levelorder_iterator

`const_levelorder_iterator` implementiert einen Levelorder-Iterator für konstante Bäume.

Klassendefinition

Die Klasse besitzt nun einen Zeiger auf einen konstanten Baum:

```
class const_levelorder_iterator;
friend class const_levelorder_iterator;
class const_levelorder_iterator
: public std::iterator<std::forward_iterator_tag, value_type> {
private:
  CKnot *m_knot;
  const CTree *m_tree;
  int m_level;
  std::deque<CKnot*> m_queue;
};
```

Listing 228: Die Klassendefinition von const_levelorder_iterator

Konstruktoren

```
const_levelorder_iterator()
  :m_knot(0), m_tree(0),m_level(0) {
  m_queue.push_back(0);
}
```

Listing 229: Die Konstruktoren von const_levelorder_iterator

```
//*********

const_levelorder_iterator(CKnot *kn, const CTree *tr)
  : m_knot(kn), m_tree(tr),m_level(0) {
  m_queue.push_back(0);
}
```

Listing 229: Die Konstruktoren von const_levelorder_iterator (Forts.)

operator* & operator->

```
const_reference operator*() {
  return(m_knot->m_data);
}

//*********

const_pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 230: Die Methoden operator und operator-> von const_levelorder_iterator*

Vergleichsoperatoren

```
bool operator==(const_levelorder_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(const_levelorder_iterator &i) const {
  return(m_knot!=i.m_knot);
}
```

Listing 231: Die Vergleichsoperatoren von const_levelorder_iterator

operator++

```
const_levelorder_iterator &operator++() {
  if(m_knot->m_left)
    m_queue.push_back(m_knot->m_left);
  if(m_knot->m_right)
    m_queue.push_back(m_knot->m_right);
  if(m_queue.size()!=0) {
    m_knot=m_queue.front();
    m_queue.pop_front();
    if(!m_knot) {
      m_level++;
      m_queue.push_back(0);
      m_knot=m_queue.front();
      m_queue.pop_front();
    }
  }
  else
    m_knot=0;
  return(*this);
}

//*********

const_levelorder_iterator operator++(int) {
  const_levelorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 232: Die Methoden operator++ von const_levelorder_iterator

getLevel

```
int getLevel() const {
  return(m_level);
}
```

Listing 233: Die getLevel-Methode von const_levelorder_iterator

Iterator-Methoden

```
const_levelorder_iterator levelorder_begin() const {
  return(const_levelorder_iterator(m_root,this));
}

//*********

const_levelorder_iterator levelorder_end() const {
  return(const_levelorder_iterator(0,this));
}
```

Listing 234: Die Iterator-Methoden für const_levelorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

43 Wie kann ein höhenbalancierter Binärbaum implementiert werden?

Wir haben in den letzten Rezepten einen Binärbaum implementiert und ihn mit Iteratoren bestückt. Was aber, wenn wir die Iteratoren auch auf einem ausgeglichenen Baum verwenden wollen?

Wir werden dazu einen ausgeglichenen Baum programmieren, der die wesentliche Funktionalität von `CTree` aus Rezept 38 übernimmt.

Von den verschiedenen Möglichkeiten, einen Baum auszugleichen, wollen wir den AVL-Baum wählen. Bei einem AVL-Baum besitzt jeder Knoten eine so genannte Balance. Die Balance ist definiert als die Differenz der Höhen des rechten und linken Teilbaums.

In einem AVL-Baum besitzt jeder Knoten eine Balance von –1, 0 oder 1.

Sollte diese Balance größer als 1 beziehungsweise kleiner als –1 werden, so ist die AVL-Bedingung verletzt und der Baum wird durch Rotationen wieder in einen AVL-Baum umgewandelt.

Auf die hinter diesen Mechanismen liegende Theorie kann hier leider nicht eingegangen werden.

CAVLKnot

In einem AVL-Baum besitzt jeder Knoten zusätzlich zu den Verweisen auf Vater, rechten und linken Sohn noch eine Balance. Wir leiten dazu die Klasse `CAVLKnot` von der Klasse `CKnot` aus `CTree` ab:

Klassendefinition

```
typedef short balance_type;

class CAVLKnot : public CAVLTree::CKnot {
public:
  balance_type m_balance;
};
```

Listing 235: Die Klassendefinition von CAVLKnot

Konstruktoren

Alle Konstruktoren benutzen zur Initialisierung der Knoten-Verweise und der Nutzdaten den Basisklassen-Konstruktor:

```
CAVLKnot(CAVLKnot *fa, CAVLKnot *li, CAVLKnot *re)
: CKnot(fa,li,re),m_balance(0)
{ }

//*********

CAVLKnot(CAVLKnot *fa, CAVLKnot *li, CAVLKnot *re,
         const value_type &v)
: CKnot(fa,li,re,v),m_balance(0)
{}

//*********

CAVLKnot(CAVLKnot *fa, CAVLKnot *li, CAVLKnot *re,
         const value_type &v, balance_type b)
: CKnot(fa,li,re,v),m_balance(b)
{}
```

Listing 236: Die Konstruktoren von CAVLKnot

Destruktor

Der Destruktor macht eigentlich nichts außer mit seiner Virtualität dafür Sorge zu tragen, dass der Destruktor einer eventuellen Unterklasse ebenfalls aufgerufen wird.

```
virtual ~CAVLKnot()
{}
```

Listing 237: Der Destruktor von CAVLKnot

Zugriffs-Methoden

Um innerhalb der Klasse `CAVLTree` die Problematik etwas zu verringern, dass die Knotenverweise eigentlich vom Typ `CKnot` sind, fügen wir drei Zugriffs-Methoden hinzu, die die Umwandlung übernehmen. Wir entscheiden uns hier für die ungeprüfte Typumwandlung, weil in unserem Kontext nur Knoten vom Typ `CAVLKnot` vorkommen können.

```
CAVLKnot *getLeft() const {
  return(reinterpret_cast<CAVLKnot*>(m_left));
}

//*********

CAVLKnot *getRight() const {
  return(reinterpret_cast<CAVLKnot*>(m_right));
}

//*********

CAVLKnot *getFather() const {
  return(reinterpret_cast<CAVLKnot*>(m_father));
}
```

Listing 238: Die Zugriffs-Methoden von CAVLKnot

CAVLTree

Damit im AVL-Baum ebenfalls die Iteratoren von `CTree` zur Verfügung stehen, leiten wir `CAVLTree` von `CTree` ab.

Klassendefinition

Genau wie `CTree` wird das Verhalten des AVL-Baums von Tree-Traits abhängig gemacht.

```
template<typename Traits>
class CAVLTree : public CTree<Traits> {
public:
  typedef CTree<Traits> base_type;
  typedef CAVLTree<Traits> my_type;
  typedef typename Traits::value_type value_type;
  typedef typename Traits::key_type key_type;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef short balance_type;

  class CAVLKnot : public CAVLTree::CKnot { /* ... */ };
};
```

Listing 239: Die Klassendefinition von CAVLTree

Konstruktoren

Genau wie `CTree` statten wir den AVL-Baum mit einem Standard-Konstruktor, einem Kopier-Konstruktor und einem Konstruktor aus, der den Baum mit Elementen aus einem mit Iteratoren definierten Bereich initialisiert:

```
CAVLTree(void)
: base_type() {
}

//*********

CAVLTree(const CAVLTree &t)
: base_type() {
  *this=t;
}

//*********
```

Listing 240: Die Konstruktoren von CAVLTree

```
template<typename Input>
CAVLTree(Input beg, Input end)
: base_type() {
  insert(beg, end);
}
```

Listing 240: Die Konstruktoren von CAVLTree (Forts.)

Destruktor

Der Destruktor muss keine Arbeit übernehmen, weil alle notwendigen Löschoperationen vom Destruktor der Basisklasse ausgeführt werden.

```
virtual ~CAVLTree()
{}
```

Listing 241: Der Destruktor von CAVLTree

Zuweisungsoperator

Der Zuweisungsoperator kopiert einen Baum knotenweise unter Zuhilfenahme von `copySons`.

```
CAVLTree &operator=(const CAVLTree &t) {
  if(&t!=this) {

/*
** Alten Baum löschen
*/
    deleteKnot(m_root);
    m_size=t.m_size;

/*
** Zuzuweisender Baum nicht leer?
** => alle Knoten kopieren
*/
    if(t.m_root) {

/*
```

Listing 242: Der Zuweisungsoperator von CAVLKnot

```
** Zunächst Wurzel kopieren
*/
      m_root=new CAVLKnot(0,0,0,t.m_root->m_data,
                  reinterpret_cast<CAVLKnot*>(t.m_root)->m_balance);

/*
** Söhne der Wurzel mit copySons kopieren
*/
      copySons(reinterpret_cast<CAVLKnot*>(m_root),
               reinterpret_cast<CAVLKnot*>(t.m_root));
    }
  }
  return(*this);
}
```

Listing 242: Der Zuweisungsoperator von CAVLKnot (Forts.)

copySons

Die Methode `copySons` kopiert die Söhne eines Knotens:

```
void copySons(CAVLKnot* d, CAVLKnot *s) {

/*
** Beide Knoten existent?
*/
  if(d&&s) {

/*
** Linken Sohn kopieren, falls vorhanden, und
** für ihn copySons rekursiv aufrufen
*/
    if(s->m_left) {
      d->m_left=new CAVLKnot(d,0,0,s->m_left->m_data,
         reinterpret_cast<CAVLKnot*>(s->m_left)->m_balance);
      copySons(reinterpret_cast<CAVLKnot*>(d->m_left),
               reinterpret_cast<CAVLKnot*>(s->m_left));
    }

/*
** Rechten Sohn kopieren, falls vorhanden, und
```

Listing 243: Die Methode copySons von CAVLKnot

```
** für ihn copySons rekursiv aufrufen
*/
    if(s->m_right) {
      d->m_right=new CAVLKnot(d,0,0,s->m_right->m_data,
         reinterpret_cast<CAVLKnot*>(s->m_right)->m_balance);
      copySons(reinterpret_cast<CAVLKnot*>(d->m_right),
               reinterpret_cast<CAVLKnot*>(s->m_right));
    }
  }
}
```

Listing 243: Die Methode copySons von CAVLKnot (Forts.)

insert

Damit die beim Einfügen unter Umständen notwendigen Umstrukturierungen ausgeführt werden können, müssen in `CAVLTree` alle `insert`-Methoden der Basis-Klasse überschrieben werden.

Die erste `insert`-Methode erzeugt aus einem Nutzdaten-Objekt einen Knoten und ruft die dafür verantwortliche `insert`-Methode auf:

```
virtual inorder_iterator insert(const value_type &v) {

/*
** Datenobjekt in einen Knoten packen
*/
  CAVLKnot *kn=new CAVLKnot(0,0,0,v);

/*
** Knoten in Baum einfügen
*/
  insert(kn);

/*
** Position als Iterator zurückgeben
*/
  return(_iterator(kn,this,0));
}
```

Listing 244: Die Methode insert für value_type-Objekte

Um eine größere Kompatibilität zu anderen STL-Containern zu erreichen, fügen wir eine `insert`-Methode mit der zusätzlichen Angabemöglichkeit einer Einfüge-Position bei. Diese Position muss natürlich in einem Baum ignoriert werden:

```
inorder_iterator insert(_iterator i, const value_type &v) {
  return(insert(v));
}
```

Listing 245: Die Methode insert mit Einfüge-Position

Als letzte der öffentlichen `insert`-Methoden fehlt noch die Version, die einen mit Iteratoren definierten Bereich in den Baum einfügt:

```
template<typename Input>
void insert(Input beg, Input end) {
  while(beg!=end)
    insert(*(beg++));
}
```

Listing 246: Die insert-Methode für Bereiche

Zum Schluss fehlt noch die Methode, die den mit den anderen `insert`-Methoden erzeugten Knoten in den Baum einfügt.

```
void insert(CAVLKnot *kn) {

/*
** Baum leer?
** => Einzufügender Knoten wird die Wurzel
*/
  if(!m_root) {
    m_root=kn;
    m_size++;
    return;
  }

  CAVLKnot *cur=reinterpret_cast<CAVLKnot*>(m_root);
  while(cur) {
```

Listing 247: Die insert-Methode für Knoten

```
    if(Traits::gt(Traits::getKey(cur->m_data),
                  Traits::getKey(kn->m_data))) {

/*
** Einzufügender Knoten kleiner als aktueller Knoten?
** => Falls vorhanden, im linken Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten linker
**    Sohn des aktuellen Knotens
*/
      if(!cur->m_left) {
        cur->m_left=kn;
        cur->m_balance--;
        kn->m_father=cur;
        m_size++;

/*
** Hat sich Balance von 0 auf anderen Wert geändert
** => Höhe hat sich geändert, eventuell umstrukturieren
*/
        if(cur->m_balance)
          rebalanceInsert(cur);
        return;
      }
      else {
        cur=cur->getLeft();
      }
    }
    else {

/*
** Einzufügender Knoten größer/gleich dem aktuellen Knoten?
** => Falls vorhanden, im rechten Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten rechter
**    Sohn des aktuellen Knotens
*/
      if(!cur->m_right) {
        cur->m_right=kn;
        cur->m_balance++;
        kn->m_father=cur;
        m_size++;
```

Listing 247: Die insert-Methode für Knoten (Forts.)

```
/*
** Hat sich Balance von 0 auf anderen Wert geändert
** => Höhe hat sich geändert, eventuell umstrukturieren
*/
        if(cur->m_balance)
          rebalanceInsert(cur);
        return;
      }
      else {
        cur=cur->getRight();
      }
    }
  }
}
```

Listing 247: Die insert-Methode für Knoten (Forts.)

rebalanceInsert

Die Methode `rebalanceInsert` steigt vom eingefügten Knoten aus den Baum nach oben und nimmt mit Hilfe der Methode `rotate` eventuell notwendige Korrekturen der Baumstruktur vor:

```
void rebalanceInsert(CAVLKnot *kn) {
  CAVLKnot *fkn=kn->getFather();

  if(((kn->m_balance==-1)||(kn->m_balance==1))&&(kn!=m_root)) {
    if(kn->m_father->m_left==kn)
      kn->getFather()->m_balance--;
    else
      kn->getFather()->m_balance++;
    rebalanceInsert(kn->getFather());
    return;
  }

  if(kn->m_balance==-2) {
    if(kn->getLeft()->m_balance==-1) {
      rotate(kn);
      return;
    }
    else {
```

Listing 248: Die Methode rebalanceInsert von CAVLTree

```
      rotate(kn->getLeft());
      rotate(kn);
      return;
    }
  }

   if(kn->m_balance==2) {
    if(kn->getRight()->m_balance==1) {
      rotate(kn);
      return;
    }
    else {
      rotate(kn->getRight());
      rotate(kn);
      return;
    }
  }
}
```

Listing 248: Die Methode rebalanceInsert von CAVLTree (Forts.)

erase

Die folgende `erase`-Methode löscht einen Knoten über seinen Schlüssel. Dabei werden alle Knoten mit diesem Schlüssel gelöscht und die Anzahl der gelöschten Knoten zurückgegeben.

```
virtual size_type erase(const key_type &k) {

/*
** Ersten Knoten mit Schlüssel k finden
*/
  CAVLKnot *kn=reinterpret_cast<CAVLKnot*>(findFirstKnot(k));

/*
** Keine Knoten vorhanden?
** => Kein Knoten gelöscht
*/
  if(!kn)
    return(0);
```

Listing 249: Die erase-Methode für key_type-Objekte

```
/*
** Inorder-Iterator auf Knoten hinter dem zu löschenden
** Knoten setzen
*/
  inorder_iterator i=_iterator(kn,this,0);
  ++i;
  size_type a=0;

/*
** So lange laufen, wie noch Knoten mit
** Schlüssel k existieren
*/
  do {

/*
** Schlüssel löschen, neuen Knoten aus Iterator
** holen und Iterator inkrementieren
*/
    erase(kn);
    a++;
    kn=reinterpret_cast<CAVLKnot*>(i.m_knot);
    ++i;
  } while((kn)&&(Traits::eq(Traits::getKey(kn->m_data),k)));
  return(a);
}
```

Listing 249: Die erase-Methode für key_type-Objekte (Forts.)

Die nachstehende Methode löscht den Knoten an der Iterator-Position `i` und liefert den nächsten Knoten der Inorder-Reihenfolge als Iterator zurück:

```
virtual inorder_iterator erase(_iterator i) {

/*
** Iterator-Position hinter zu löschendem
** Knoten ermitteln
** (Ohne Anfangs-Position für Iterator zu bestimmen)
*/
  inorder_iterator io(i.m_knot,this,0);
  ++io;
```

Listing 250: Die erase-Methode für Iterator-Positionen

```
/*
** Konnte Knoten gelöscht werden?
** => Position hinter gelöschtem Knoten zurückgeben
** Andernfalls End-Position zurückgeben
*/
  if(erase(reinterpret_cast<CAVLKnot*>(i.m_knot)))
    return(_iterator(io.m_knot,this,0));
  else
    return(inorder_end());
}
```

Listing 250: Die erase-Methode für Iterator-Positionen (Forts.)

Nun muss von der `CTree`-Klasse noch die `erase`-Methode für zu löschende Bereiche überschrieben werden.

Der Methode werden in Form von Iterator-Positionen der Beginn des zu löschenden Bereichs und die Position hinter dem letzten Element des zu löschenden Bereichs übergeben. Die Methode liefert die Position hinter dem gelöschten Bereich (bezogen auf die Inorder-Reihenfolge) zurück.

```
virtual inorder_iterator erase(inorder_iterator beg,
                               inorder_iterator end) {
  while((beg!=inorder_end())&&(beg!=end)) {
    CKnot *kn=beg.m_knot;
    ++beg;
    erase(reinterpret_cast<CAVLKnot*>(kn));
  }
  return(beg);
}
```

Listing 251: Die Methode erase für zu löschende Bereiche

Um einen Knoten aus dem Baum entfernen zu können, existiert die folgende `erase`-Methode:

```
bool erase(CAVLKnot *cur) {
  if(!cur) return(false);
```

Listing 252: Die erase-Methode für Knoten

```
  CAVLKnot *father=cur->getFather();

/*
** Zu löschender Knoten hat keine Söhne?
** => Kann problemlos gelöscht werden
*/
  if((!cur->m_left)&&(!cur->m_right)) {

/*
** Zu löschender Knoten ist die Wurzel?
** => Baum leer
*/
    if(cur==m_root) {
      m_root=0;
      delete(cur);
      m_size--;
      return(false);
    }

/*
** Zu löschender Knoten ist nicht die Wurzel?
** => Vater muss berücksichtigt werden
*/
    else {

/*
** Je nachdem, ob zu löschender Knoten der linke oder
** rechte Sohn des Vaters ist, muss entsprechender Sohn
** des Vaters auf 0 gesetzt werden
*/
      if(father->m_left==cur) {
        father->m_left=0;
        father->m_balance++;
      }
      else {
        father->m_right=0;
        father->m_balance--;
      }

/*
```

Listing 252: Die erase-Methode für Knoten (Forts.)

```
** Knoten löschen, AVL-Bedingung testen und
** ggfs. wieder herstellen
*/
      delete(cur);
      m_size--;
      rebalanceErase(father);
      return(true);
    }
  }

/*
** Besitzt zu löschender Knoten zwei Söhne?
** => Zu löschenden Knoten durch symmetrischen
**    Vorgänger ersetzen und symmetrischen Vorgänger
**    löschen
*/
  if((cur->m_left)&&(cur->m_right)) {
    CAVLKnot *sys=reinterpret_cast<CAVLKnot*>(symmetricPred(cur));
    cur->m_data=sys->m_data;
    return(erase(sys));
  }

/*
** Besitzt zu löschender Knoten nur einen Sohn?
** => Sohn des zu löschenden Knotens wird Sohn vom
** Vater des zu löschenden Knotens
*/
  CAVLKnot *son;
  if(cur->m_left)
    son=cur->getLeft();
  else
    son=cur->getRight();

  if(cur!=m_root) {
    son->m_father=father;
    if(father->m_left==cur) {
      father->m_left=son;
      father->m_balance++;
    }
    else {
      father->m_right=son;
```

Listing 252: Die erase-Methode für Knoten (Forts.)

```
      father->m_balance--;
    }

/*
** AVL-Bedingung testen und ggfs. wieder herstellen
*/
    rebalanceErase(father);
    }

/*
** Ist zu löschender Sohn die Wurzel?
** => Sohn des zu löschenden Knotens wird Wurzel
*/
  else {
    son->m_father=0;
    m_root=son;
  }

/*
** Knoten löschen
*/
  delete(cur);
  m_size--;
  return(true);
}
```

Listing 252: Die erase-Methode für Knoten (Forts.)

rebalanceErase

`rebalanceErase` stellt eine eventuell verletzte AVL-Bedingung nach dem Entfernen eines Knotens wieder her.

```
void rebalanceErase(CAVLKnot *kn) {
  CAVLKnot *fkn=kn->getFather();

  if((kn->m_balance==-1)||(kn->m_balance==1))
    return;
  if((kn==m_root)&&(kn->m_balance==0))
    return;
```

Listing 253: Die Methode rebalanceErase von CAVLTree

```
if(kn==m_root) {
  if(kn->m_balance==-2) {
    if(kn->getLeft()->m_balance<=0)
      rotate(kn);
    else {
      kn=rotate(kn->getLeft());
      rotate(kn);
    }
  }
  else {
    if(kn->getRight()->m_balance>=0)
      rotate(kn);
    else {
      kn=rotate(kn->getRight());
      rotate(kn);
    }
  }
  return;
}
if(kn->m_balance==2) {
  switch(kn->getRight()->m_balance) {
    case 0:
      rotate(kn);
      return;

    case 1:
      rebalanceErase(rotate(kn));
      return;

    case -1:
      rotate(kn->getRight());
      rebalanceErase(rotate(kn));
      return;
  }
}
if(kn->m_balance==-2) {
  switch(kn->getLeft()->m_balance) {
    case 0:
      rotate(kn);
      return;
```

Listing 253: Die Methode rebalanceErase von CAVLTree (Forts.)

```
      case -1:
        rebalanceErase(rotate(kn));
        return;

      case 1:
        rotate(kn->getLeft());
        rebalanceErase(rotate(kn));
        return;
    }
  }
  if(fkn->m_left==kn) {
    fkn->m_balance++;
    if(fkn->m_balance<2) {
      rebalanceErase(fkn);
      return;
    }
    switch(fkn->getRight()->m_balance) {
      case 0:
        rotate(fkn);
        return;

      case 1:
        rebalanceErase(rotate(fkn));
        return;

      case -1:
        rotate(fkn->getRight());
        rebalanceErase(rotate(fkn));
        return;
    }
  }
  if(fkn->m_right==kn) {
    fkn->m_balance--;
    if(fkn->m_balance>-2) {
      rebalanceErase(fkn);
      return;
    }
    switch(fkn->getLeft()->m_balance) {
      case 0:
        rotate(fkn);
        return;
```

Listing 253: Die Methode rebalanceErase von CAVLTree (Forts.)

```
      case -1:
        rebalanceErase(rotate(fkn));
        return;

      case 1:
        rotate(fkn->getLeft());
        rebalanceErase(rotate(fkn));
        return;
    }
  }
}
```

Listing 253: Die Methode rebalanceErase von CAVLTree (Forts.)

rotate

Die Methode `rotate` wird von `rebalanceInsert` und `rebalanceErase` eingesetzt, um den Baum umzustrukturieren und ihn damit wieder zu einem AVL-Baum zu machen.

Genau wie `rebalanceInsert` und `rebalanceErase` wird `rotate` der Vollständigkeit halber aufgeführt, bleibt aber unkommentiert, weil eine Erklärung der dahinter liegenden Theorie hier den Rahmen sprengen würde.

```
CAVLKnot *rotate(CAVLKnot *kn) {
  CAVLKnot *child;

  if(kn->m_balance<0) {
    child=kn->getLeft();
    kn->m_left=child->m_right;
    if(child->m_right)
      child->m_right->m_father=kn;
    child->m_right=kn;
    child->m_father=kn->m_father;
    kn->m_father=child;
    if(child->m_father) {
      if(child->m_father->m_left==kn)
        child->m_father->m_left=child;
      else
        child->m_father->m_right=child;
```

Listing 254: Die Methode rotate von CAVLTree

```
    }
    else
      m_root=child;

    if(kn->m_balance==-1) {
      if(child->m_balance==1) {
        child->m_balance=2;
        kn->m_balance=0;
        return(child);
      }

      if(child->m_balance==-1)
        kn->m_balance=1;
      else
        kn->m_balance=0;
      child->m_balance=1;
      return(child);
    }
    if(kn->m_balance==-2) {
      if(child->m_balance==-1) {
        kn->m_balance=child->m_balance=0;
        return(child);
      }
      if(child->m_balance==0) {
        kn->m_balance=-1;
        child->m_balance=1;
        return(child);
      }
      if(child->m_balance==-2) {
        kn->m_balance=1;
        child->m_balance=0;
        return(child);
      }
    }
  }
  else {
    child=kn->getRight();
    kn->m_right=child->m_left;
    if(child->m_left) child->m_left->m_father=kn;
    child->m_left=kn;
    child->m_father=kn->m_father;
```

Listing 254: Die Methode rotate von CAVLTree (Forts.)

```
    kn->m_father=child;
    if(child->m_father) {
      if(child->m_father->m_left==kn)
        child->m_father->m_left=child;
      else
        child->m_father->m_right=child;
    }
    else
      m_root=child;

    if(kn->m_balance==1) {
      if(child->m_balance==-1) {
//            Inorder();
        child->m_balance=-2;
        kn->m_balance=0;
        return(child);;
      }

      if(child->m_balance==1)
        kn->m_balance=-1;
      else
        kn->m_balance=0;
      child->m_balance=-1;
      return(child);
    }
    if(kn->m_balance==2) {
      if(child->m_balance==1)  {
        kn->m_balance=child->m_balance=0;
        return(child);
      }
      if(child->m_balance==0)  {
        kn->m_balance=1;
        child->m_balance=-1;
        return(child);
      }
      if(child->m_balance==2)  {
        kn->m_balance=-1;
        child->m_balance=0;
        return(child);
      }
    }
```

Listing 254: Die Methode rotate von CAVLTree (Forts.)

```
  }
  return(child);
}
```

Listing 254: Die Methode rotate von CAVLTree (Forts.)

pop_front & pop_back

Mittels `pop_front` und `pop_back` können Knoten bezogen auf die Inorder-Reihenfolge am Anfang oder am Ende des Baumes entfernt werden.

```
virtual void pop_front() {
  if(!empty())
    erase(reinterpret_cast<CAVLKnot*>(inorder_begin().m_knot));
}

//*********

virtual void pop_back() {
  if(!empty())
    erase(reinterpret_cast<CAVLKnot*>(inorder_rbegin().m_knot));
}
```

Listing 255: Die Methoden pop_front und pop_back

push_front & push_back

Der Kompatibilität wegen dem Baum hinzugefügt, halten sie allerdings nicht, was ihr Name verspricht:

```
virtual void push_front(const value_type &v) {
  insert(v);
}

//*********

virtual void push_back(const value_type &v) {
  insert(v);
}
```

Listing 256: Die Methoden push_front und push_back

Sie finden die Klasse `CAVLTree` auf der CD in der *CAVLTree.h*-Datei.

CAVLSetTree

Um den Baum wie ein `set` der STL nutzen zu können, implementieren wir eine Klasse `CAVLSetTree` und benutzen dazu die `set_tree_traits` aus Rezept 38:

```
template<typename VType, typename Cmp= std::less<VType> >
class CAVLSetTree : public CAVLTree<set_tree_traits<VType, Cmp> > {
public:
  typedef CAVLTree<set_tree_traits<VType, Cmp> > base_type;
  typedef CAVLSetTree<VType, Cmp> my_type;

//*********

  CAVLSetTree()
  : base_type()
  {}

//*********

  template<typename Input>
  CAVLSetTree(Input beg, Input end)
  : base_type(beg,end)
  { }

//*********

  CAVLSetTree(const CAVLSetTree &t)
  : base_type(t)
  { }
};
```

Listing 257: Die Klasse CAVLSetTree

Sie finden die Klasse `CAVLSetTree` auf der CD in der *CAVLSetTree.h*-Datei.

CAVLMapTree

Analog zu `CAVLSetTree` stellen wir die `map`-Funktionalität mit der Klasse `CAVLMapTree` her und benutzen dazu die `map_tree_traits` aus Rezept 38:

```
template<typename KType,
         typename VType,
         typename Cmp= std::less<KType> >
class CAVLMapTree : public CAVLTree<map_tree_traits<KType,
                                                    VType,
                                                    Cmp> > {
public:
  typedef CAVLTree<map_tree_traits<KType, VType, Cmp> > base_type;
  typedef CAVLMapTree<KType, VType, Cmp> my_type;

//*********

  CAVLMapTree()
  : base_type()
  {}

//*********

  template<typename Input>
  CAVLMapTree(Input beg, Input end)
  : base_type(beg,end)
  { }

//*********

  CAVLMapTree(const CAVLMapTree &t)
  : base_type(t)
  { }
};
```

Listing 258: Die Klasse CAVLMapTree

Sie finden die Klasse `CAVLMapTree` auf der CD in der *CAVLMapTree.h*-Datei.