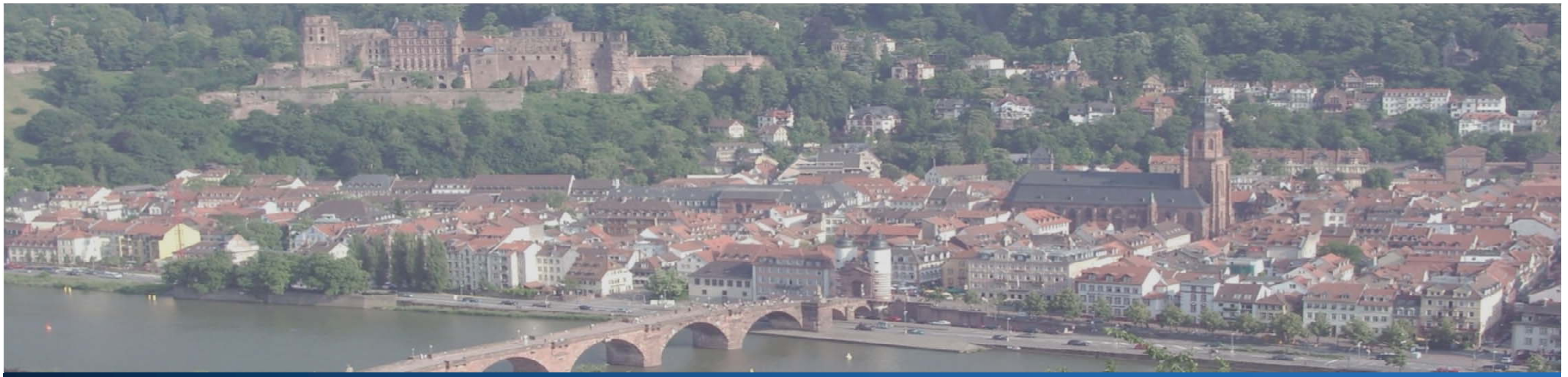
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Containerklassen

- Containerklassen
- Motivation
- Listenschablone
- Iteratoren
- Doppeltverkettete Liste
- Feld
- Stack
- Queue & DeQueue
- Prioritätswarteschlangen
- Set
- Map
- Anwendung: Huffman-Code

Motivation

Bezeichnung: Klassen, die eine Menge anderer Objekte verwalten (man sagt aggregieren) nennt man Containerklassen.

Beispiele: Wir hatten: Liste, Stack, Feld. Weitere sind: binärer Baum (*binary tree*), Warteschlange (*queue*), Abbildung (*map*), . . .

Bemerkung: Diese Strukturen treten sehr häufig als Komponenten in größeren Programmen auf. Ziel von Containerklassen ist es, diese Bausteine in wiederverwendbarer Form zur Verfügung zu stellen (*code reuse*).

Vorteile:

- Weniger Zeitaufwand in Entwicklung und Fehlersuche.
- Klarere Programmstruktur, da man auf einer höheren Abstraktionsebene arbeitet.

Werkzeug: Das Werkzeug zur Realisierung effizienter und flexibler Container in C++ sind Klassenschablonen.

Bemerkung: In diesem Abschnitt sehen wir uns eine Reihe von Containern an. Die Klassen sind vollständig ausprogrammiert und zeigen, wie man Container implementieren $könnte$. In der Praxis verwendet man allerdings die *Standard Template Library* (STL), welche Container in professioneller Qualität bereitstellt.

Ziel: Sie sind am Ende dieses Kapitels motiviert, die STL zu verwenden und können die Konzepte verstehen.

Listenschablone

Bei diesem Entwurf ist die Idee, das Listenelement und damit auch die Liste als Klassenschablone zu realisieren. In jedem Listenelement wird ein Objekt der Klasse `T`, dem Schablonenparameter, gespeichert.

Programm: **Definition und Implementation (Liste.hh)**

```
template <class T>
class List
{
public:
  // Infrastruktur
  List() { _first = 0; }
  ~List();

  // Listenelement als nested class
  class Link
  {
```

```
    Link* _next;
  public:
    T item;
    Link( T& t ) { item = t; }
    Link* next() { return _next; }
    friend class List<T>;
  };

  Link* first() { return _first; }
  void insert( Link* where, T t );
  void remove( Link* where );

private:
  Link* _first;
  // privater Copy-Konstruktor und Zuweisungs-
  // operator da Defaultvarianten zu fehlerhaftem
  // Verhalten fuehren
  List( const List<T>& l ) {};
  List<T>& operator=( const List<T>& l ) {};
```

friend: Zugriff auf private members

```
};

template <class T> List<T>::~List()
{
  Link* p = _first;
  while ( p != 0 )
  {
    Link* q = p;
    p = p->next();
    delete q;
  }
}

template <class T>
void List<T>::insert( List<T>::Link* where, T t )
{
  Link* ins = new Link(t);
  if ( where == 0 )
  {
```

```
    ins->_next = _first;
    _first = ins;
  }
  else
  {
    ins->_next = where->_next;
    where->_next = ins;
  }
}

template <class T>
void List<T>::remove( List<T>::Link* where )
{
  Link* p;
  if ( where == 0 )
  {
    p = _first;
    if ( p != 0 ) _first = p->_next;
  }
```

```
    else
    {
      p = where->_next;
      if ( p != 0 ) where->_next = p->_next;
    }
    delete p;
}
```

Programm: Verwendung (UseListe.cc)

```
#include <iostream>
#include "Liste.hh"

int main()
{
  List<int> list;

  list.insert( 0, 17 );
  list.insert( 0, 34 );
  list.insert( 0, 26 );

  for ( List<int>::Link* l=list.first(); l!=0; l=l->next() )
    std::cout << l->item << std::endl;
  for ( List<int>::Link* l=list.first(); l!=0; l=l->next() )
    l->item = 23;
}
```

Bemerkung:

- Diese Liste ist homogen, d. h. alle Objekte im Container haben den gleichen Typ. Eine heterogene Liste könnte man als Liste von Zeigern auf eine gemeinsame Basisklasse realisieren.

- Speicherverwaltung wird von der Liste gemacht. Listen können kopiert und als Parameter übergeben werden (sofern Copy-Konstruktor und Zuweisungsoperator noch mittels deep copy implementiert werden).

- Zugriff auf die Listenelemente erfolgt über eine offengelegte *nested class*. Die Liste wird als friend deklariert, damit die Liste den `next`-Zeiger manipulieren kann, nicht jedoch der Benutzer der Liste.

Iteratoren

Problem: Grundoperation aller Container sind

- Durchlaufen aller Elemente,
- Zugriff auf Elemente.

Um Container austauschbar verwenden zu können, sollten diese Operationen mit der gleichen Schnittstelle möglich sein. Die Schleife für eine Liste sah aber ganz anders aus als bei einem Feld.

Abhilfe: Diese Abstraktion realisiert man mit Iteratoren. Iteratoren sind zeigerähnliche Objekte, die auf ein Objekt im Container zeigen (obwohl der Iterator nicht als Zeiger realisiert sein muss).

Prinzip:

```
template <class T> class Container
{
public:
  class Iterator
  {  // nested class definition
    ...
  public:
    Iterator();
    bool operator!=( Iterator x );
    bool operator==( Iterator x );
    Iterator operator++();        // prefix
    Iterator operator++( int );   // postfix
    T& operator*() const;
    T* operator->() const;
    friend class Container<T>;
  };
  Iterator begin() const;
  Iterator end() const;
```

```
  ... // Spezialitäten des Containers
};

// Verwendung
Container<int> c;
for ( Container<int>::Iterator i=c.begin(); i!=c.end(); ++i )
  std::cout << *i << std::endl;
```

Bemerkung:

- Der `Iterator` ist als Klasse innerhalb der Containerklasse definiert. Dies nennt man eine geschachtelte Klasse (*nested class*).
- Damit drückt man aus, dass `Container` und `Iterator` zusammengehören. Jeder Container wird seine eigene Iteratorklasse haben.
- Innerhalb von `Container` kann man `Iterator` wie jede andere Klasse verwenden.

- `friend class Container<T>` bedeutet, dass die Klasse `Container<T>` auch Zugriff auf die *privaten* Datenmitglieder der Iteratorklasse hat.

- Die Methode `begin()` des Containers liefert einen Iterator, der auf das erste Element des Containers zeigt.

- `++i` bzw. `i++` stellt den Iterator auf das $nächste$ Element im Container. Zeigte der Iterator auf das letzte Element, dann ist der Iterator gleich dem von `end()` gelieferten Iterator.

- `++i` bzw. `i++` manipulieren den Iterator für den sie aufgerufen werden. Als Rückgabewert liefert `++i` den neuen Wert des Iterators, `i++` jedoch den alten.

- Bei der Definition unterscheiden sie sich dadurch, dass der Postfix-Operator noch ein **int**-Argument erhält, das aber keine Bedeutung hat.

- `end()` liefert einen Iterator, der auf „das Element nach dem letzten Element“ des Containers zeigt (siehe oben).

- `*i` liefert eine Referenz auf das Objekt im Container, auf das der Iterator `i` zeigt. Damit kann man sowohl `x = *i` als auch `*i = x` schreiben.

- Ist das Objekt im Container von einem zusammengesetzten Datentyp (also `struct` oder `class`), so kann mittels `i-><Komponente>` eine Komponente selektiert werden. Der Iterator verhält sich also wie ein Zeiger.

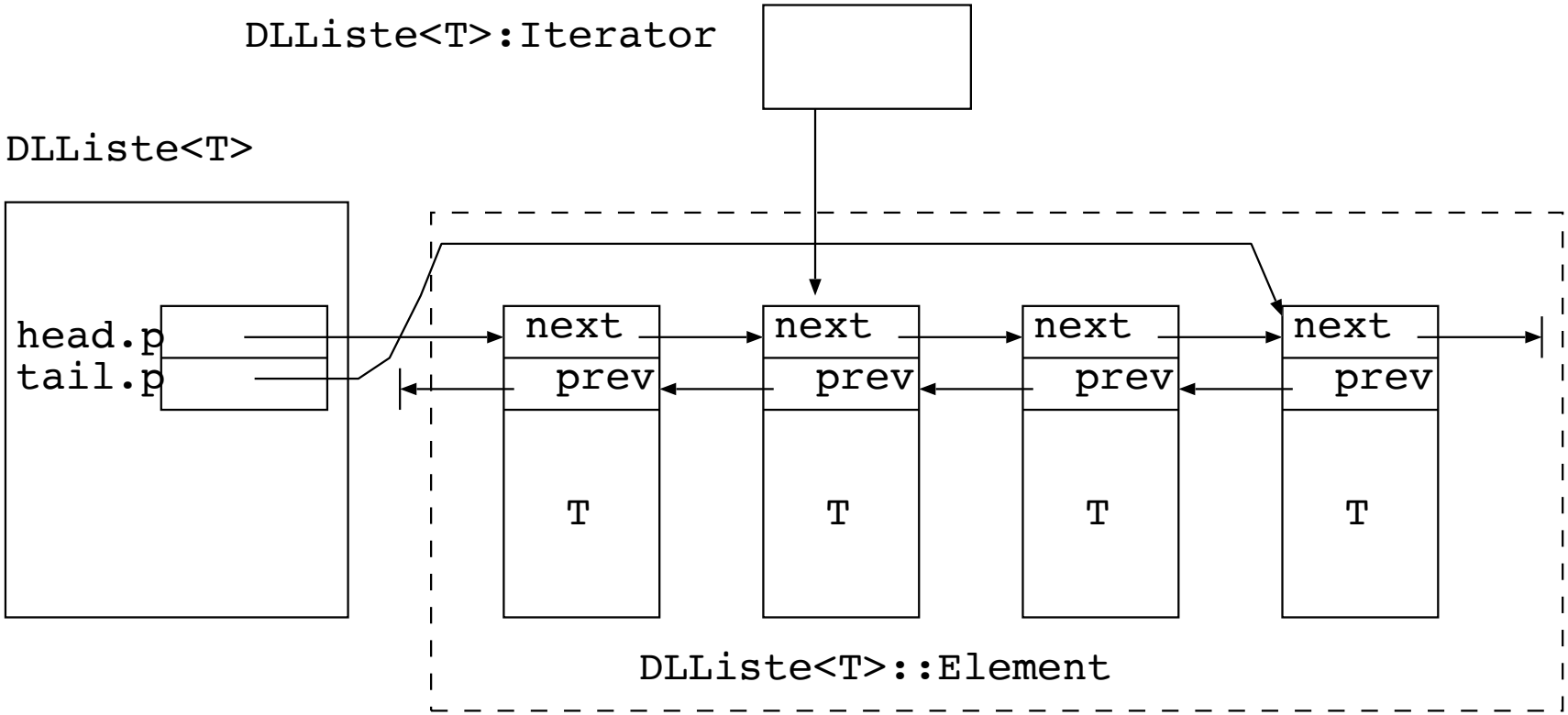
Machen wir ein Beispiel . . .

Doppelt verkettete Liste

Anforderungen:

- Vorwärts- und Rückwärtsdurchlauf
- Das Einfügen vor oder nach einem Element soll eine $O(1)$-Operation sein. Die Position wird durch einen Iterator angegeben.
- Das Entfernen eines Elementes soll eine $O(1)$-Operation sein. Das zu entfernende Element wird wieder durch einen Iterator angegeben
- Für die Berechnung der Größe der Liste akzeptieren wir einen $O(N)$-Aufwand.

Struktur


DLListe<T>:Iterator
DLListe<T>
head.p
tail.p
next
prev
T
next
prev
T
next
prev
T
next
prev
T
DLListe<T>::Element

Bemerkung:

- Intern werden die Listenelemente durch den Datentyp `Element` repräsentiert. Dieser private, geschachtelte, zusammengesetzte Datentyp ist außerhalb der Klasse nicht sichtbar.

- Die Einfügeoperationen erhalten Objekte vom Typ `T`, erzeugen dynamisch ein Listenelement und $kopieren$ das Objekt in das Listenelement.

- Damit kann man Listen für beliebige Datentypen erzeugen. Die Manipulation gelingt mit Hilfe der Iteratorschnittstelle. Der Iterator kapselt insbesondere den Zugriff auf die außerhalb der Liste nicht bekannten `Element`-Objekte.

Implementation

Programm: DLL.hh

```
template <class T>
class DLList
{
  // das interne Listenelement
  struct Element
  {
    Element* next;
    Element* prev;
    T item;
    Element( T &t )
    {
      item = t;
      next = prev = 0;
    }
  };
```

```
public:
  typedef T MemberType;   // Merke Grundtyp

  // der iterator kapselt Zeiger auf Listenelement
  class Iterator
  {
  private:
    Element* p;
  public:
    Iterator() { p = 0; }
    Iterator( Element* q ) { p = q; }
    bool operator!=( Iterator x ) {
      return p != x.p;
    }
    bool operator==( Iterator x ) {
      return p == x.p;
    }
    Iterator operator++() { // prefix
```

```
    p = p->next;
    return *this;
  }
  Iterator operator++( int ) { // postfix
    Iterator tmp = *this;
    p = p->next;
    return tmp;
  }
  Iterator operator--() { // prefix
    p = p->prev;
    return *this;
  }
  Iterator operator--( int ) { // postfix
    Iterator tmp = *this;
    p = p->prev;
    return tmp;
  }
  T& operator*() { return p->item; }
  T* operator->() { return &(p->item); }
```

```
  friend class DLList<T>; // Liste manip. p
};

// Iteratoren
Iterator begin() const { return head; }
Iterator end() const { return Iterator(); }
Iterator rbegin() const { return tail; }
Iterator rend() const { return Iterator(); }

// Konstruktion, Destruktion, Zuweisung
DLList();
DLList( const DLList<T>& list );
DLList<T>& operator=( const DLList<T>& );
~DLList();

// Listenmanipulation
Iterator insert( Iterator i, T t ); // einf. vor i
void erase( Iterator i );
void append( const DLList<T>& l );
```

```
  void clear();
  bool empty() const;
  int size() const;
  Iterator find( T t ) const;

private:
  Iterator head;     // erstes Element der Liste
  Iterator tail;     // letztes Element der Liste
};


// Insertion
template <class T>
typename DLList<T>::Iterator
DLList<T>::insert( Iterator i, T t )
{
  Element* e = new Element( t );
  if ( empty() )
  {
```

```
    assert( i.p == 0 );
    head.p = tail.p = e;
  }
  else
  {
    e->next = i.p;
    if ( i.p != 0 )
    {    // insert before i
      e->prev = i.p->prev;
      i.p->prev = e;
      if ( head == i )
        head.p = e;
    }
    else
    {    // insert at end
      e->prev = tail.p;
      tail.p->next = e;
      tail.p = e;
    }
```

```
  }
  return Iterator( e );
}

template <class T>
void DLList<T>::erase( Iterator i )
{
  if ( i.p == 0 ) return;

  if ( i.p->next != 0 )
    i.p->next->prev = i.p->prev;
  if ( i.p->prev != 0 )
    i.p->prev->next = i.p->next;

  if ( head == i ) head.p = i.p->next;
  if ( tail == i ) tail.p = i.p->prev;

  delete i.p;
}
```

```
template <class T>
void DLList<T>::append( const DLList<T>& l ) {
  for ( Iterator i=l.begin(); i!=l.end(); i++ )
    insert( end(), *i );
}

template <class T>
bool DLList<T>::empty() const {
  return begin() == end();
}

template <class T>
void DLList<T>::clear() {
  while ( !empty() )
    erase( begin() );
}

// Constructors
```

```
template <class T> DLList<T>::DLList() {}

template <class T>
DLList<T>::DLList( const DLList<T>& list ) {
  append( list );
}

// Assignment
template <class T>
DLList<T>&
DLList<T>::operator=( const DLList<T>& l )
{
  if ( this != &l )
  {
    clear();
    append(l);
  }
  return *this;
}
```

```
// Destructor
template <class T> DLList<T>::~DLList() { clear(); }

// Size method
template <class T> int DLList<T>::size() const
{
  int count = 0;
  for ( Iterator i=begin(); i!=end(); i++ )
    count++;
  return count;
}

template <class T>
typename DLList<T>::Iterator DLList<T>::find( T t ) const
{
  DLList<T>::Iterator i = begin();
  while ( i != end() )
  {
```

```
    if ( *i == t ) break;
    i++;
  }
  return i;
}

template <class T>
std::ostream& operator<<( std::ostream& s, DLList<T>& a )
{
  s << "(";
  for ( typename DLList<T>::Iterator i=a.begin();
        i!=a.end(); i++ )
  {
    if ( i != a.begin() ) s << " ";
    s << *i;
  }
  s << ")" << std::endl;
  return s;
}
```

Verwendung

Programm: UseDLL.cc

```
#include <cassert>
#include <iostream>
#include "DLL.hh"
#include "Zufall.cc"

int main()
{
  Zufall z( 87124 );
  DLList<int> l1, l2, l3;

  // Erzeuge 3 Listen mit je 5 Zufallszahlen
  for ( int i=0; i<5; i=i+1 )
    l1.insert( l1.end(), i );
  for ( int i=0; i<5; i=i+1 )
    l2.insert( l2.end(), z.ziehe_zahl() );
```

```
for ( int i=0; i<5; i=i+1 )
  l3.insert( l3.end(), z.ziehe_zahl() );

// Loesche alle geraden in der ersten Liste
DLList<int>::Iterator i, j;
i = l1.begin();
while ( i != l1.end() )
{
  j = i; // merke aktuelles Element
  ++i;   // gehe zum naechsten
  if ( *j % 2 == 0 ) l1.erase( j );
}

// Liste von Listen ...
DLList<DLList<int> > ll;
ll.insert( ll.end(), l1 );
ll.insert( ll.end(), l2 );
ll.insert( ll.end(), l3 );
std::cout << ll << std::endl;
```

```
    std::cout << "Laenge: " << ll.size() << std::endl;
}
```

Diskussion

- Den Rückwärtsdurchlauf durch eine Liste c erreicht man durch:

```
for ( DLList<int>::Iterator i=c.rbegin();
      i!=c.rend(); --i )
  std::cout << *i << endl;
```

- Die Objekte (vom Typ `T`) werden beim Einfügen in die Liste kopiert. Abhilfe: Liste von Zeigern auf die Objekte, z. B. `DLList<int*>`.

- Die Schlüsselworte **const** in der Definition von `begin`, `end`, ... bedeuten, dass diese Methoden ihr Objekt nicht ändern.

- Innerhalb einer Template-Definition werden geschachtelte Klassen nicht als Typ erkannt. Daher muss man den Namen explizit mittels **typename** als Typ kennzeichnen.

Beziehung zur STL-Liste

Die entsprechende STL-Schablone heißt `list` und unterscheidet sich von unserer Liste unter anderem in folgenden Punkten:

- Man erhält die Funktionalität durch **#include** <list>.
- Die Iterator-Klasse heißt `iterator` statt `Iterator`.
- Es gibt zusätzlich einen `const_iterator`. Auch unterscheiden sich Vorwärts- und Rückwärtsiteratoren (`reverse_iterator`).
- Sie hat einige Methoden mehr, z. B. `push_front`, `push_back`, `front`, `back`, `pop_front`, `pop_back`, `sort`, `reverse`, . . .
- Die Ausgabe über „`std::cout` <<“ ist nicht definiert.

Feld

Wir fügen nun die Iterator-Schnittstelle unserer `SimpleArray<T>`-Schablone hinzu.

Programm: **(Array.hh)**

```
template <class T> class Array
{
public:
  typedef T MemberType;   // Merke Grundtyp

  // Iterator fuer die Feld-Klasse
  class Iterator
  {
  private:
    T* p; // Iterator ist ein Zeiger ...
    Iterator( T* q ) { p = q; }
  public:
    Iterator() { p = 0; }
    bool operator!=( Iterator x ) {
```

```
    return ( p != x.p );
  }
  bool operator==( Iterator x ) {
    return ( p == x.p );
  }
  Iterator operator++() {
    p++;
    return *this;
  }
  Iterator operator++( int ) {
    Iterator tmp = *this;
    ++*this;
    return tmp;
  }
  T& operator*() const { return *p; }
  T* operator->() const { return p; }
  friend class Array<T>;
};
```

```
// Iterator Methoden
Iterator begin() const {
  return Iterator( p );
}
Iterator end() const {
  return Iterator( &(p[n]) ); // ja, das ist ok!
}

// Konstruktion; Destruktion und Zuweisung
Array( int m ) {
  n = m;
  p = new T[n];
}
Array( const Array<T>& );
Array<T>& operator=( const Array<T>& );
~Array() {
  delete[] p;
}
```

```
  // Array manipulation
  int size() const {
    return n;
  }
  T& operator[]( int i ) {
    return p[i];
  }

private:
  int n;   // Anzahl Elemente
  T* p;    // Zeiger auf built-in array
};

// Copy-Konstruktor
template <class T>
Array<T>::Array( const Array<T>& a ) {
  n = a.n;
  p = new T[n];
  for ( int i=0; i<n; i=i+1 )
```

```
    p[i] = a.p[i];
}

// Zuweisung
template <class T>
Array<T>& Array<T>::operator=( const Array<T>& a ) {
  if ( &a != this )
  {
    if ( n != a.n )
    {
      delete[] p;
      n = a.n;
      p = new T[n];
    }
    for ( int i=0; i<n; i=i+1 ) p[i] = a.p[i];
  }
  return *this;
}
```

```
// Ausgabe
template <class T>
std::ostream& operator<<( std::ostream& s, Array<T>& a ) {
  s << "array " << a.size() << " elements = [" << std::endl;
  for ( int i=0; i<a.size(); i++ )
    s << "    " << i << "  " << a[i] << std::endl;
  s << "]" << std::endl;
  return s;
}
```

Bemerkung:

- Der Iterator ist als Zeiger auf ein Feldelement realisiert.
- Die Schleife

  ```
  for (Array<int>::Iterator i=a.begin(); i!=a.end(); ++i) ...
  ```

  entspricht nach Inlining der Methoden einfach

  ```
  for (int* p=a.p; p!=&a[100]; p=p+1) ...
  ```

  und ist somit nicht langsamer als handprogrammiert!
- Man beachte auch die Definition von `MemberType`. Dies ist praktisch innerhalb eines Template **template** <**class** C>, wo der Datentyp eines Containers `C` dann als `C::MemberType` erhalten werden kann.

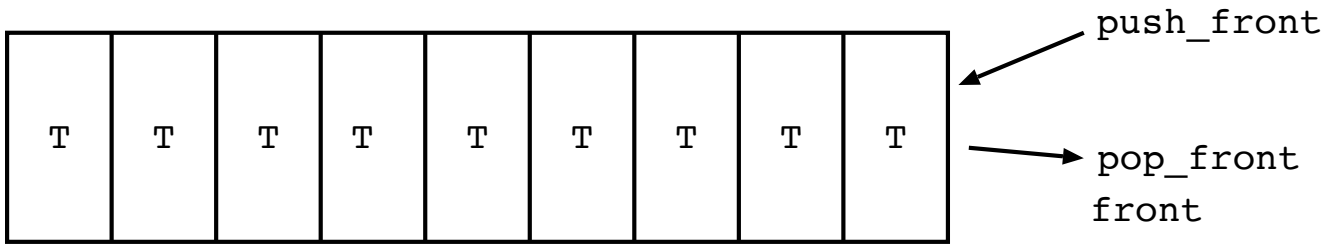
Programm: **Gleichzeitige Verwendung** `DLList/Array` **(UseBoth.cc)**:

```
#include <cassert>
#include <iostream>

#include "Array.hh"
#include "DLL.hh"
#include "Zufall.cc"

int main()
{
  Zufall z( 87124 );
  Array<int> a( 5 );
  DLList<int> l;

  // Erzeuge Array und Liste mit 5 Zufallszahlen
  for ( int i=0; i<5; i=i+1 ) a[i] = z.ziehe_zahl();
  for ( int i=0; i<5; i=i+1 )
    l.insert( l.end(), z.ziehe_zahl() );
```

```
    // Benutzung
    for ( Array<int>::Iterator i=a.begin(); i!=a.end(); i++ )
      std::cout << *i << std::endl;
    std::cout << std::endl;
    for ( DLList<int>::Iterator i=l.begin(); i!=l.end(); i++ )
      std::cout << *i << std::endl;
}
```

Bemerkung: Die STL-Version von `Array` erhält man mit **#include** <vector>. Die Klassenschablone heißt `vector` anstatt `Array`.

Stack



Schnittstelle:

- Konstruktion eines Stack.
- Einfügen eines Elementes vom Typ `T` oben (`push`).
- Entfernen des obersten Elementes (`pop`).
- Inspektion des obersten Elementes (`top`).
- Test ob Stack voll oder leer (`empty`).

Programm: **Implementation über DLList (Stack.hh)**

```
template <class T>
class Stack : private DLList<T>
{
public:
  // Default-Konstruktoren + Zuweisung OK

  bool empty() { return DLList<T>::empty(); }
  void push( T t ) {
    insert( begin(), t );
  }
  T top() { return *begin(); }
  void pop() { erase( begin() ); }
};
```

Bemerkung:

- Wir haben den Stack als Spezialisierung der doppelt verketteten Liste realisiert. Etwas effizienter wäre die Verwendung einer einfach verketteten Liste gewesen.

- Auffallend ist, dass die Befehle `top`/`pop` getrennt existieren (und `pop` keinen Wert zurückliefert). Verwendet werden diese Befehle nämlich meist gekoppelt, so dass auch eine Kombination `pop` $\leftarrow$ `top`+`pop` nicht schlecht wäre.

Programm: **Anwendung: (UseStack.cc)**

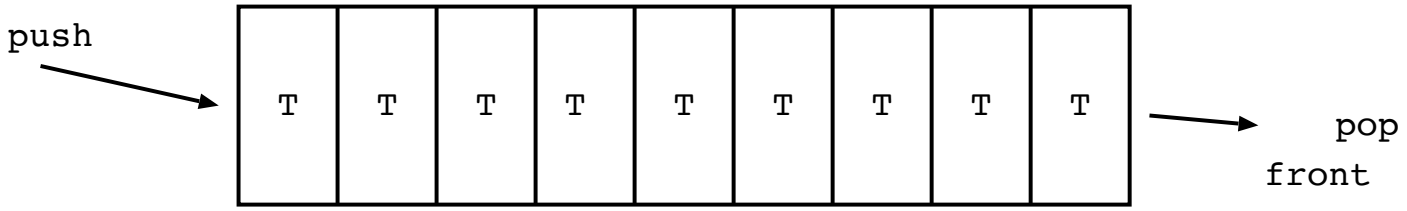
```
#include <cassert>
#include <iostream>

#include "DLL.hh"
#include "Stack.hh"

int main()
{
  Stack<int> s1;
  for ( int i=1; i<=5; i++ )
    s1.push( i );

  Stack<int> s2( s1 );
  s2 = s1;
  while ( !s2.empty() )
```

```
    {
        std::cout << s2.top() << std::endl;
        s2.pop();
    }
}
```

Bemerkung: Die STL-Version erhält man durch **#include** <stack>. Die Klassenschablone heißt dort `stack` und hat im wesentlichen dieselbe Schnittstelle.

Queue

Eine Queue ist eine Struktur, die Einfügen an einem Ende und Entfernen nur am anderen Ende erlaubt:



Anwendung: Warteschlangen.

Schnittstelle:

- Konstruktion einer leeren Queue
- Einfügen eines Elementes vom Typ `T` am Ende
- Entfernen des Elementes am Anfang
- Inspektion des Elementes am Anfang
- Test ob Queue leer

Programm: **(Queue.hh)**

```
template <class T>
class Queue : private DLList<T>
{
public:
  // Default-Konstruktoren + Zuweisung OK
  bool empty() {
    return DLList<T>::empty();
  }
  T front() {
    return *DLList<T>::begin();
  }
  T back() {
    return *DLList<T>::rbegin();
  }
  void push( T t ) {
```

```
    insert( DLList<T>::end(), t );
  }
  void pop() {
    erase( DLList<T>::begin() );
  }
};
```

Bemerkung: Die STL-Version erhält man durch **#include** <queue>. Die Klassenschablone heißt dort queue und hat im wesentlichen dieselbe Schnittstelle wie Queue.

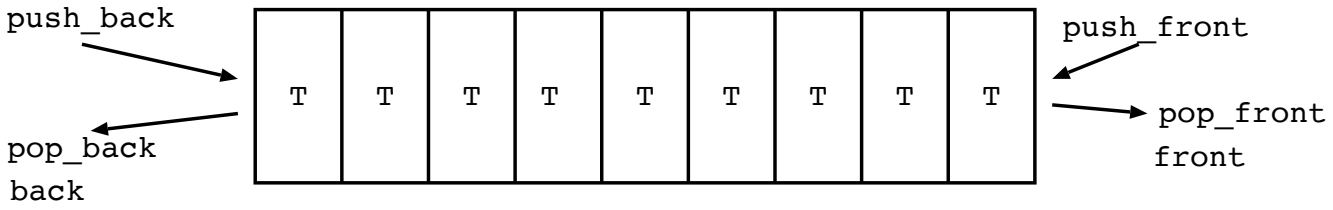
Programm: Zur Abwechslung verwenden wir mal die STL-Version: **(UseQueueSTL.cc)**

```
#include <queue>
#include <iostream>

int main()
{
  std::queue<int> q;
  for ( int i=1; i<=5; i++ )
    q.push( i );

  while ( !q.empty() )
  {
    std::cout << q.front() << std::endl;
    q.pop();
  }
```

}

DeQueue

Eine DeQueue (*double-ended queue*) ist eine Struktur, die Einfügen und Entfernen an beiden Enden erlaubt:



Schnittstelle:

- Konstruktion einer leeren `DeQueue`
- Einfügen eines Elementes vom Typ `T` am Anfang oder Ende
- Entfernen des Elementes am Anfang oder am Ende
- Inspektion des Elementes am Anfang oder Ende
- Test ob `DeQueue` leer

Programm: **(DeQueue.hh)**

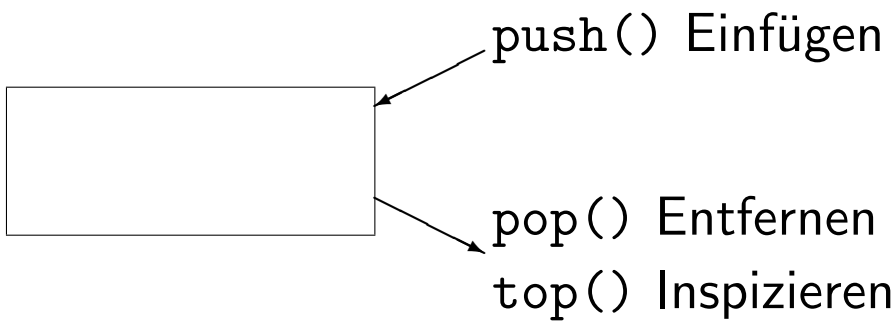
```
template <class T>
class DeQueue : private DLList<T>
{
public:
  // Default-Konstruktoren + Zuweisung ok
  bool empty();
  void push_front( T t );
  void push_back( T t );
  T pop_front();
  T pop_back();
  T front();
  T back();
};
```

Bemerkung: Die STL-Version erhält man auch hier mit **#include** <queue>. Die Klassenschablone heißt deque.

Prioritätswarteschlangen

Bezeichnung: Eine Prioritätswarteschlange ist eine Struktur, in die man Objekte des Grundtyps `T` einfüllen kann und von der jeweils das *kleinste* (MinPriorityQueue) bzw. das *größte* (MaxPriorityQueue) der eingegebenen Elemente als nächstes entfernt werden kann. Bei gleich großen Elementen verhält sie sich wie eine Queue.

Bemerkung: Auf dem Grundtyp `T` muß dazu die Relation $<$ mittels dem **operator**$<$ zur Verfügung stehen.



Schnittstelle:

- Konstruktion einer leeren MinPriorityQueue.
- Einfügen eines Elementes vom Typ `T` (`push`).
- Entfernen des kleinsten Elementes im Container (`pop`).
- Inspektion des kleinsten Elementes im Container (`top`).
- Test ob MinPriorityQueue leer (`empty`).

Programm: Hier die Klassendefinition **(MinPriorityQueue.hh)**:

```
template <class T>
class MinPriorityQueue : private DLList<T>
{
private:
  typename DLList<T>::Iterator find_minimum();
public:
  // Default-Konstruktoren + Zuweisung OK
  bool empty();
  void push( T t );   // Einfuegen
  void pop();         // Entferne kleinstes
  T top();            // Inspiziere kleinstes
};
```

Und die Implementation **(MinPriorityQueueImp.cc)**:

```
template <class T>
bool MinPriorityQueue<T>::empty() {
  return DLList<T>::empty();
```

```
}

template <class T>
void MinPriorityQueue<T>::push( T t ) {
  insert( DLList<T>::begin(), t );
}

template <class T>
typename DLList<T>::Iterator
MinPriorityQueue<T>::find_minimum()
{
  typename DLList<T>::Iterator min = DLList<T>::begin();
  for ( typename DLList<T>::Iterator i=DLList<T>::begin();
        i!=DLList<T>::end(); i++ )
    if ( *i <= *min ) min = i;
  return min;
}

template <class T>
```

```
inline void MinPriorityQueue<T>::pop() {
  erase( find_minimum() );
}

template <class T>
inline T MinPriorityQueue<T>::top() {
  return *find_minimum();
}
```

Bemerkung:

- Unsere Implementierung arbeitet mit einer einfach verketteten Liste. Das Einfügen hat Komplexität $O(1)$, das Entfernen/Inspizieren jedoch $O(n)$.

- Bessere Implementationen verwenden einen Heap, was zu einem Aufwand der Ordnung $O(\log n)$ führt.

- Analog ist die Implementation der `MaxPriorityQueue`.

Bemerkung: Die STL-Version erhält man auch durch **#include** <queue>. Die Klassenschablone heißt `priority_queue` und implementiert eine `MaxPriorityQueue`. Man kann allerdings den Vergleichsoperator auch als Template-Parameter übergeben (etwas lästig).

Set

Ein Set (Menge) ist ein Container mit folgenden Operationen:

- Konstruktion einer leeren Menge.
- Einfügen eines Elementes vom Typ `T`.
- Entfernen eines Elementes.
- Test auf Enthaltensein.
- Test ob Menge leer.

Programm: Klassendefinition **(Set.hh)**:

```
template <class T>
class Set : private DLList<T>
{
public:
  // Default-Konstruktoren + Zuweisung OK

  typedef typename DLList<T>::Iterator Iterator;
  Iterator begin();
  Iterator end();

  bool empty();
  bool member( T t );
  void insert( T t );
  void remove( T t );
  // union, intersection, ... ?
};
```

Implementation **(SetImp.cc)**:

```
template <class T>
typename Set<T>::Iterator Set<T>::begin() {
  return DLList<T>::begin();
}

template <class T>
typename Set<T>::Iterator Set<T>::end() {
  return DLList<T>::end();
}

template <class T>
bool Set<T>::empty() {
  return DLList<T>::empty();
}

template <class T>
inline bool Set<T>::member( T t ) {
```

```
  return find( t ) != DLList<T>::end();
}

template <class T>
inline void Set<T>::insert( T t )
{
  if ( !member( t ) )
    DLList<T>::insert( DLList<T>::begin(), t );
}

template <class T>
inline void Set<T>::remove( T t )
{
  typename DLList<T>::Iterator i = find( t );
  if ( i != DLList<T>::end() )
    erase( i );
}
```

Bemerkung:

- Die Implementierung hier basiert auf der doppelt verketteten Liste von oben (private Ableitung!).
- Einfügen, Suchen und Entfernen hat die Komplexität $O(n)$.
- Wir lernen später Implementierungen kennen, die den Aufwand $O(\log n)$ für alle Operationen haben.
- Auf dem Typ `T` muss der Vergleichsoperator **operator**$<$ definiert sein. (Set gehört zu den sog. sortierten, assoziativen Containern).

Map

Bezeichnung: Eine Map ist ein assoziatives Feld, das Objekten eines Typs `Key` Objekte eines Typs `T` zuordnet.

Beispiel: Telefonbuch:

Meier	$\rightarrow$	504423
Schmidt	$\rightarrow$	162300
Müller	$\rightarrow$	712364
Huber	$\rightarrow$	8265498

Diese Zuordnung könnte man realisieren mittels:

```
Map<string, int> telefonbuch;
telefonbuch["Meier"] = 504423;
...
```

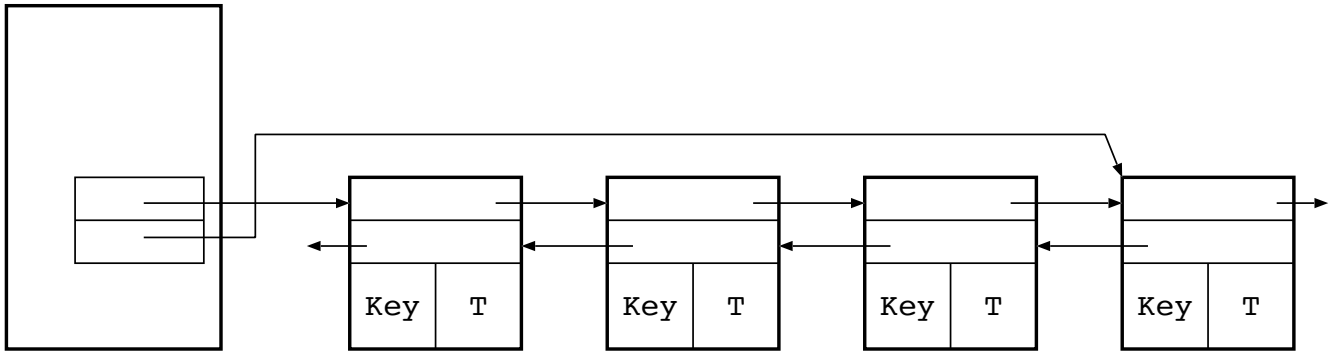
Programm: Definition der Klassenschablone (Map.hh)

```
// Existiert schon als std::pair
// template <class Key, class T>
// struct pair {
//     Key first;
//     T second;
// };

template <class Key, class T>
class Map : private DLList<pair<Key, T> >
{
public:

  T& operator[]( const Key& k );

  typedef typename DLList<pair<Key, T> >::Iterator Iterator;
  Iterator begin() const;
  Iterator end() const;
```

```
    Iterator find( const Key& k );
};
```

Bemerkung:

- In dieser Implementation von Map werden je zwei Objekte der Typen `Key` (*Schlüssel*) und `T` (*Wert*) zu einem Paar vom Typ `pair<Key,T>` kombiniert und in eine doppelt verkettete Liste eingefügt:



- Ein Objekt vom Typ `Key` kann nur einmal in einer Map vorkommen. (Daher ist das Telefonbuch kein optimales Beispiel.)

- Wir haben einen Iterator zum Durchlaufen des Containers.
- Auf dem Schlüssel muss der Vergleichsoperator **operator**$<$ definiert sein.
- `find(Key k)` liefert den Iterator für den Wert, ansonsten `end()`.
- Der Aufwand einer Suche ist wieder $O(n)$. Bald werden wir aber eine Realisierung von `Map` kennenlernen, die Einfügen und Suchen in $O(\log n)$ Schritten ermöglicht.

Wiederholung: Containerklassen

- Container speichern Objekte eines Grundtyps `T`.
- Container erlauben Inspektion der Elemente und Modifikation des Inhalts.
- Diese Operationen werden mit Hilfe von Iteratorklassen abstrahiert.
- Container unterscheiden sich hinsichtlich des Zugriffsmusters und der Komplexität der Operationen.

Folgende Container haben wir besprochen:

- Feld
- Liste (doppelt verkettet)
- Stack
- Queue
- Double-ended Queue
- Priority Queue
- Set
- Map

Anwendung: Huffman-Code

Problem: Wir wollen eine Zeichenfolge, z. B.

'ABRACADABRASIMSALABIM'

durch eine Folge von Zeichen aus der Menge $\{0, 1\}$ darstellen (kodieren).

Dazu wollen wir jedem der neun (verschiedenen) Zeichen aus der Eingabekette eine Folge von Bits zuzuordnen.

Am einfachsten ist es, einen Code fester Länge zu konstruieren. Mit n Bits können wir 2^n verschiedene Zeichen kodieren. Im obigem Fall genügen also vier Bit, um jedes der neun verschiedenen Zeichen in der Eingabekette zu kodieren, z. B.

A	0001	D	0100	M	0111
B	0010	I	0101	R	1000
C	0011	L	0110	S	1010

Die Zeichenkette wird dann kodiert als

$$\underbrace{0001}_{\text{A}}\underbrace{0010}_{\text{B}}\underbrace{1000}_{\text{R}}\ldots$$

Insgesamt benötigen wir $21 \cdot 4 = 84$ Bits (ohne die Übersetzungstabelle!).

Beobachtung: Kommen manche Zeichen häufiger vor als andere (wie etwa bei Texten in natürlichen Sprachen) so kann man Platz sparen, indem man Codes variabler Länge verwendet.

Beispiel: Morsecode.

Beispiel: Für unsere Beispielzeichenkette 'ABRACADABRASIMSALABIM' wäre folgender Code gut:

A	1	D	010	M	100
B	10	I	11	R	101
C	001	L	011	S	110

Damit kodieren wir unsere Beispielkette als

$$\underbrace{1}_{\mathsf{A}}\ \underbrace{10}_{\mathsf{B}}\ \underbrace{101}_{\mathsf{R}}\ \underbrace{1}_{\mathsf{A}}\ \underbrace{001}_{\mathsf{C}}\ldots$$

Schwierigkeit: Bei der Dekodierung könnte man diese Bitfolge auch interpretieren als

$$\underbrace{110}_{\mathsf{S}}\ \underbrace{101}_{\mathsf{R}}\ \underbrace{100}_{\mathsf{M}}\ldots$$

Abhilfe: Es gibt zwei Möglichkeiten das Problem zu umgehen:

1. Man führt zusätzliche Trennzeichen zwischen den Zeichen ein (etwa die Pause beim Morsecode).

2. Man sorgt dafür, dass kein Code für ein Zeichen der Anfang (Präfix) eines anderen Zeichens ist. Einen solchen Code nennt man Präfixcode.

Frage: Wie sieht der optimale Präfixcode für eine gegebene Zeichenfolge aus, d. h. ein Code der die gegebene Zeichenkette mit einer Bitfolge minimaler Länge kodiert.

Antwort: Huffmancodes! (Sie sind benannt nach ihrem Entdecker David Huffman[18], der auch die Optimalität dieser Codes gezeigt hat.)

Beispiel: Für unsere Beispiel-Zeichenkette ist ein solcher Huffmancode

A	11	D	10011	M	000
B	101	I	001	R	011
C	1000	L	10010	S	010

Die kodierte Nachricht lautet hier

$$11101011111000111001111101011110100010000101110010111010010 00$$

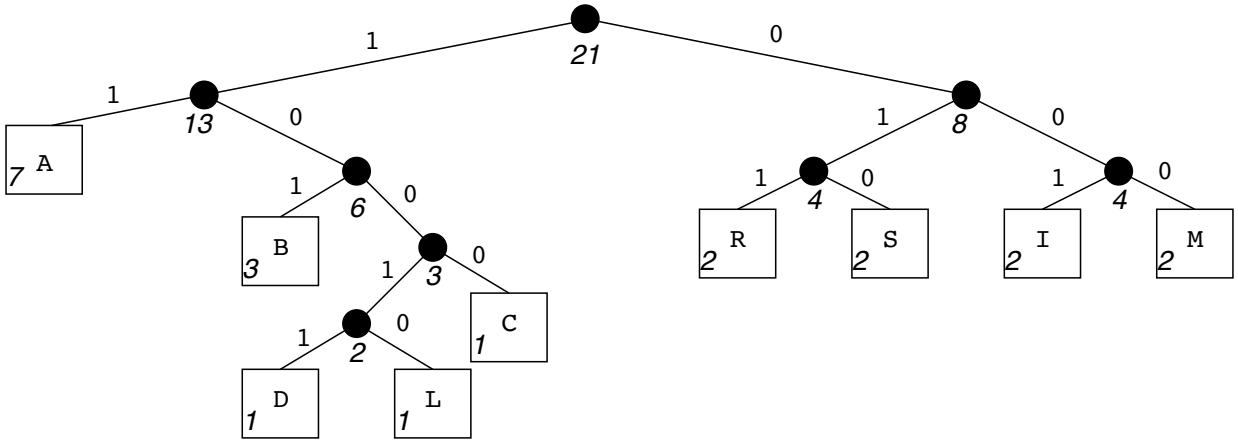
und hat nur noch 61 Bits!

[18] David A. Huffman, 1925–1999, US-amerk. Computerpionier.

Trie

Einem Präfixcode kann man einen binären Baum zuordnen, der Trie (von „retrieval“, Aussprache wie „try“) genannt wird. In den Blättern stehen die zu kodierenden Zeichen. Ein Pfad von der Wurzel zu einem Blatt kodiert das entsprechende Zeichen.

Blätter enthalten die zu kodierenden Zeichen, innere Knoten haben nur Wegweiserfunktion.



Bemerkung: Zeichen, die häufig vorkommen, stehen nahe bei der Wurzel. Zeichen, die seltener vorkommen, stehen tiefer im Baum.

Konstruktion von Huffmancodes

1. Zähle die Häufigkeit jedes Zeichens in der Eingabefolge. Erzeuge für jedes Zeichen einen Knoten mit seiner Häufigkeit. Packe alle Knoten in eine Menge E.

2. Solange die Menge E nicht leer ist: Entferne die zwei Knoten l und r mit **geringster** Häufigkeit aus E. Erzeuge einen neuen Knoten n mit l und r als Söhnen und der Summe der Häufigkeiten beider Söhne. Ist E leer, ist n die Wurzel des Huffmanbaumes, sonst stecke n in E.

Implementation

Programm: Huffman-Kodierung mit STL (HuffmanSTL.cc)

```
#include <iostream>
#include <map>
#include <queue>
#include <string>

using namespace std; // import namespace std

// There are no general binary trees in the STL.
// But we do not use much of this structure anyhow...
struct node
{
  struct node* left;
  struct node* right;
  char symbol;
  int weight;
```

```
  node( char c, int i ) {  // leaf constructor
    symbol = c;
    weight = i;
    left = right = 0;
  }
  node( node* l, node* r ) { // internal node constructor
    symbol = 0;
    weight = l->weight + r->weight;
    left = l;
    right = r;
  }
  bool isleaf() { return symbol != 0; }
  bool operator>( const node& a ) const {
    return weight > a.weight;
  }
};

// construct the Huffman trie for this message
node* huffman_trie( string message )
```

```
{
  // count multiplicities
  map<char, int> cmap;
  for ( string::iterator i=message.begin(); i!=message.end(); i++ )
    if ( cmap.find(*i) != cmap.end() )
      cmap[*i]++;
    else
      cmap[*i] = 1;

  // generate leaves with multiplicities
  priority_queue<node, vector<node>, greater<node> > q;
  for ( map<char, int>::iterator i=cmap.begin(); i!=cmap.end();
        i++ )
    q.push( node( i->first, i->second ) );

  // build Huffman tree (trie)
  while ( q.size() > 1 )
  {
    node* left = new node( q.top() );
```

```
    q.pop();
    node* right = new node( q.top() );
    q.pop();
    q.push( node( left, right ) );
  }
  return new node( q.top() );
}

// recursive filling of the encoding table 'code'
void fill_encoding_table( string s, node* i,
                          map<char, string>& code )
{
  if ( i->isleaf() )
    code[i->symbol] = s;
  else
  {
    fill_encoding_table( s + "0", i->left, code );
    fill_encoding_table( s + "1", i->right, code );
  }
```

```
}

// encoding
string encode( map<char, string> code, string& message ) {
  string encoded = "";
  for ( string::iterator i=message.begin(); i!=message.end(); i++ )
    encoded += code[*i];
  return encoded;
}

// decoding
string decode( node* trie, string& encoded ) {
  string decoded = "";
  node* node = trie;
  for ( string::iterator i=encoded.begin(); i!=encoded.end(); i++ )
  {
    if ( !node->isleaf() )
      node = (*i == '0') ? node->left : node->right;
    if ( node->isleaf() )
```

```
    {
      decoded.push_back( node->symbol );
      node = trie;
    }
  }
  return decoded;
}

int main() {
  string message = "ABRACADABRASIMSALABIM";

  // generate Huffman trie
  node* trie = huffman_trie( message );

  // generate and show encoding table
  map<char, string> table;
  fill_encoding_table( "", trie, table );
  for ( map<char, string>::iterator i=table.begin();
        i!=table.end(); i++ )
```

```
    cout << i->first << " " << i->second << endl;

  // encode and decode
  string encoded = encode( table, message );
  cout << "Encoded: " << encoded <<
    " [" << encoded.size() << " Bits]" << endl ;
  cout << "Decoded: " << decode( trie, encoded ) << endl;

  // the trie is not deleted here ...
}
```

Ausgabe: Wir erhalten einen anderen Huffman-Code als oben angegeben (der aber natürlich genauso effizient kodiert):

```
A 11
B 100
C 0010
D 1010
I 010
```

```
L 0011
M 1011
R 011
S 000
Encoded: 11100011110010111010111100011... [61 Bits]
Decoded: ABRACADABRASIMSALABIM
```